Chapter 7: Job-shop scheduling (pp. 134–160)
Genetic algorithms in engineering systems
Edited by A.M.S. Zalzala and P.J. Fleming
IEE control engineering series 55
©1997: The Institution of Electrical Engineers, ISBN: 0 85296 902 3

Chapter 7

Job-shop scheduling

Takeshi Yamada and Ryohei Nakano

7.1 Introduction

Scheduling is the allocation of shared resources over time to competing activities. It has been the subject of a significant amount of literature in the operations research field. Emphasis has been on investigating *machine scheduling problems* where *jobs* represent activities and *machines* represent resources; each machine can process at most one job at a time.

Table 7.1: A 3×3 problem

job	Operations routing (processing time)		
1	1 (3)	2 (3)	3 (3)
2	1 (2)	3 (3)	2 (4)
3	2 (3)	1 (2)	3 (1)

The $n \times m$ *minimum-makespan* general job-shop scheduling problem, hereafter referred to as the JSSP, can be described by a set of n jobs $\{J_i\}_{1 \leq j \leq n}$ which is to be processed on a set of m machines $\{M_r\}_{1 \leq r \leq m}$. Each job has a technological sequence of machines to be processed. The processing of job J_j on machine M_r is called the *operation* O_{jr}. Operation O_{jr} requires the exclusive use of M_r for an uninterrupted duration p_{jr}, its processing time. A *schedule* is a set of completion times for each operation $\{c_{jr}\}_{1 \leq j \leq n, 1 \leq r \leq m}$ that satisfies those constraints. The time required to complete all the jobs is called the *makespan* L. The objective when solving or optimizing this general problem is to determine the schedule which minimizes L. An example of a 3×3 JSSP is given in Table 7.1. The data includes the

routing of each job through each machine and the processing time for each operation (in parentheses).

The Gantt-Chart is a convenient way of visually representing a solution of the JSSP. An example of a solution for the 3×3 problem in Table 7.1 is given in Figure 7.1.

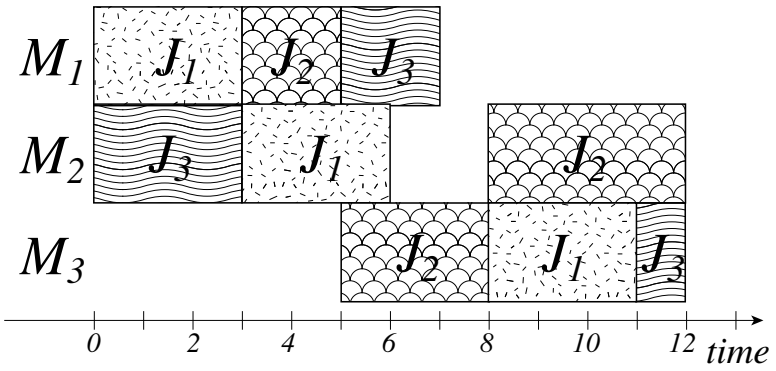

Figure 7.1: A Gantt-Chart representation of a solution for a 3×3 problem

The JSSP is not only $\mathcal{NP}$-hard , but it is one of the worst members in the class. An indication of this is given by the fact that one 10×10 problem formulated by Muth and Thompson [18] remained unsolved for over 20 years.

Besides exhaustive search algorithms based on branch and bound methods, several approximation algorithms have been developed. The most popular ones in practice are based on priority rules and active schedule generation [21]. A more sophisticated method called *shifting bottleneck* (SB) has been shown to be very successful [1]. Additionally, stochastic approaches such as simulated annealing (SA), tabu search [11, 33] and genetic algorithms (GAs) have been recently applied with good success.

This chapter reviews a variety of GA applications to the JSSP. We begin our discussion by formulating the JSSP by a disjunctive graph. We then look at domain independent binary and permutation representations, and then an active schedule representation with GT crossover and the genetic enumeration method. Section 7.7 discusses a method to integrate local optimization directly into GAs. Section 7.8 discusses performance comparison using the well-known Muth and Thompson benchmark and the more difficult "ten tough" problems.

7.2 Disjunctive graph

The JSSP can be formally described by a disjunctive graph $G = (V, C \cup D)$, where

- V is a set of nodes representing operations of the jobs together with two special nodes, a *source* (0) and a *sink* $\star$, representing the beginning and end of the schedule, respectively.

- C is a set of conjunctive arcs representing technological sequences of the operations.
- D is a set of disjunctive arcs representing pairs of operations that must be performed on the same machines.

The processing time for each operation is the weighted value attached to the corresponding nodes. Figure 7.2 shows this in a graph representation for the problem given in Table 7.1.

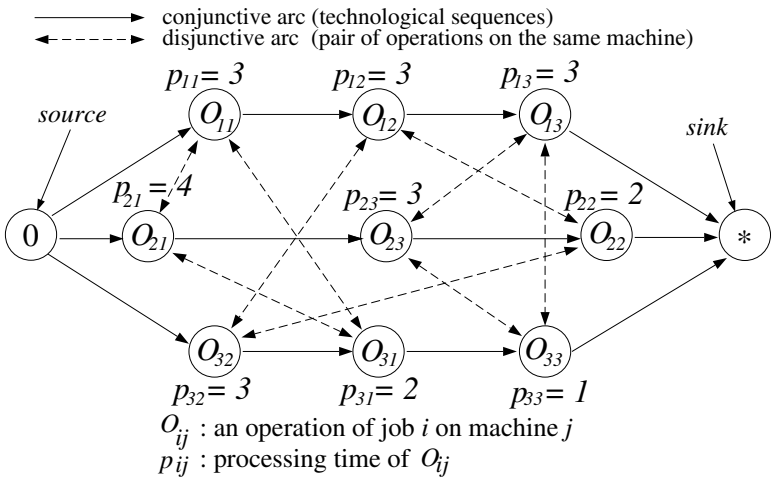

Figure 7.2: A disjunctive graph of a 3×3 problem

Job-shop scheduling can also be viewed as defining the ordering between all operations that must be processed on the same machine, i.e. to fix precedences between these operations. In the disjunctive graph model, this is done by turning all undirected (disjunctive) arcs into directed ones. A *selection* is a set of directed arcs selected from disjunctive arcs. By definition, a selection is *complete* if all the disjunctions are selected. It is *consistent* if the resulting directed graph is acyclic.

A schedule uniquely obtained from a consistent complete selection by sequencing operations as early as possible is called a *semi-active* schedule. In a semi-active schedule, no operation can be started earlier without altering the machining sequences. A consistent complete selection and the corresponding semi-active schedule can be represented by the same symbol S without confusion. The makespan L is given by the length of the longest weighted path from source to sink in this graph. This path $\mathcal{P}$ is called a *critical path* and is composed of a sequence of *critical operations*. A sequence of consecutive critical operations on the same machine is called a *critical block*.

The distance between two schedules S and T can be measured by the number of differences in the processing order of operations on each machine [19]. In other words, it can be calculated by summing the disjunctive arcs whose directions

Algorithm 7.2.1 GT algorithm

1. Let D be a set of all the earliest operations in a technological sequence not yet scheduled and O_{jr} be an operation with the minimum EC in D: $O_{jr} =$ arg min$\{O \in D \mid EC(O)\}$.
2. Assume $i-1$ operations have been scheduled on M_r. A *conflict set* $C[M_r, i]$ is defined as: $C[M_r, i] = \{O_{kr} \in D \mid O_{kr}$ on $M_r, ES(O_{kr}) < EC(O_{jr})\}$.
3. Select an operation $O \in C[M_r, i]$.
4. Schedule O as the i-th operation on M_r with its completion time equal to $EC(O)$.

are different between S and T. We call this distance the *disjunctive graph* (DG) *distance*. Figure 7.3 shows the DG distance between two schedules. The two disjunctive arcs drawn by thick lines in schedule (b) have directions that differ from those of schedule (a), and therefore the DG distance between (a) and (b) is 2.

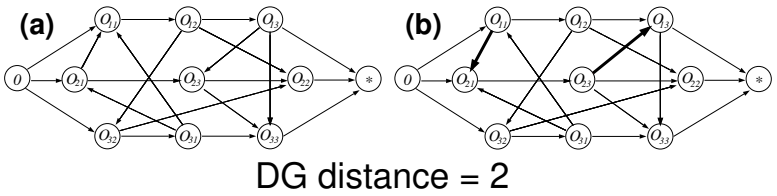

Figure 7.3: The DG distance between two schedules

7.2.1 Active schedules

The makespan of a semi-active schedule may often be reduced by shifting an operation to the left without delaying other jobs. Such reassigning is called a *permissible left shift* and a schedule with no more permissible left shifts is called an *active schedule*. An optimal schedule is clearly active so it is safe and efficient to limit the search space to the set of all active schedules. An active schedule is generated by the *GT algorithm* proposed by Giffler and Thompson [13], which is described in Algorithm 7.2.1. In the algorithm, the *earliest starting time* $ES(O)$ and *earliest completion time* $EC(O)$ of an operation O denote its starting and completion times when processed with the highest priority among all currently schedulable operations on the same machine. An active schedule is obtained by repeating the algorithm until all operations are processed. In Step 3, if all possible choices are considered, all active schedules will be generated, but the total number will still be very large.

Figure 7.4 shows how the GT algorithm works. In the figure, O_{11} is identified as O_{jr} and M_1 as M_r. Then O_{31} is selected from the conflict set and scheduled. After that, the conflict set and earliest starting times of operations are updated.

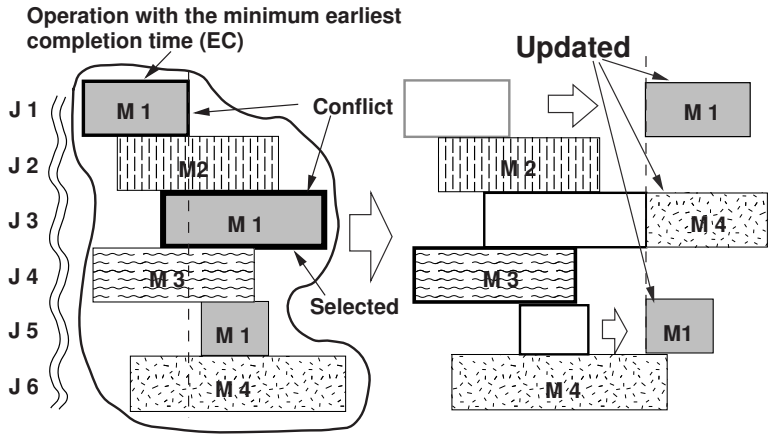

Figure 7.4: Giffler and Thompson's active schedule generation

7.3 Binary representation

As described in the previous section, a (semi-active) schedule is obtained by turning all undirected disjunctive arcs into directed ones. Therefore, by labeling each directed disjunctive arc of a schedule as 0 or 1 according to its direction, a schedule can be represented by a binary string of length $mn(n-1)/2$. Figure 7.5 shows a labeling example, where an arc connecting O_{ij} and O_{kj} $(i < k)$ is labeled as 1 if the arc is directed from O_{ij} to O_{kj} (so O_{ij} is processed prior to O_{kj}) or 0, otherwise. It should be noted that the DG distance between schedules and the Hamming distance between the corresponding binary strings can be identified through this binary mapping.

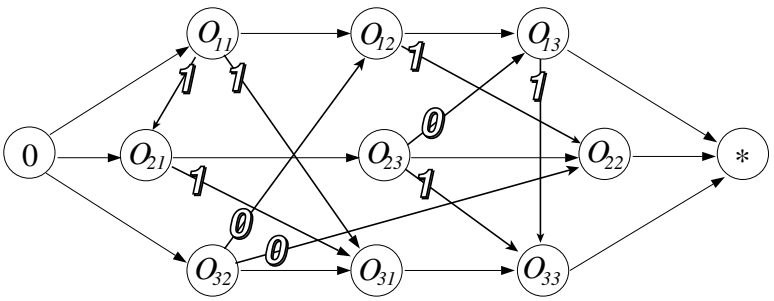

Figure 7.5: Labeling disjunctive arcs

A conventional GA using this binary representation was proposed by Nakano and Yamada [19]. An advantage of this approach is that conventional genetic operators, such as 1-point, 2-point and uniform crossovers can be applied without any modification. However, a resulting new bit string generated by crossover may not represent a schedule, and such a bit string would be called *illegal*. There are two approaches for this problem: one is to repair an illegal string and the other is to impose a penalty for the illegality. The following sections will elaborate on one example of the former approach.

7.3.1 Local harmonization

A repairing procedure that generates a feasible bit string, as similar to an illegal one as possible, is called the *harmonization algorithm* [19]. The Hamming distance is used to assess the similarity between two bit strings. The harmonization algorithm goes through two phases: *local harmonization* and *global harmonization*. The former removes the ordering inconsistencies within each machine, while the latter removes the ordering inconsistencies between machines. This section explains the former and the next section will explain the latter.

The local harmonization works separately for each machine. Thus the following merely explains how it works for one machine. Here we are given an original illegal bit string. The bit string indicates the processing priority on the machine and may include an ordering inconsistency within the machine, for example, job 1 must be prior to job 2, job 2 must be prior to job 3, but job 3 must be prior to job 1. The local harmonization can eliminate such a local inconsistency. At first, the algorithm regards the operation having the highest priority as the one to process first. When there is more than one candidate, it selects one of them. Then it removes the priority inconsistencies relevant to the top operation. By repeating the above, the local inconsistency can be completely removed. The local harmonization goes halfway in generating a feasible bit string.

7.3.2 Global harmonization

The global harmonization removes ordering inconsistencies between machines. It is embedded in a simple scheduling algorithm. First, the scheduling algorithm is explained. Given the processing priority generated by the above local harmonization as well as the technological sequences and processing time for each operation, the scheduling algorithm polls jobs checking if any job can be scheduled, and schedules an operation of a job that can be scheduled. It stops if no more jobs can be scheduled due to a global inconsistency, i.e. a deadlock happens. The global harmonization is called whenever such a deadlock occurs.

The algorithm works as follows. For each job j the algorithm considers $next(j)$, the job j operation to be scheduled next, and $next(j).machine$, the machine which processes $next(j)$. The algorithm calculates how far in the processing priority it

is from $next(next(j).machine)$, the next operation on the machine, to $next(j)$. The algorithm selects the job with the minimum distance. When there is more than one candidate, it selects one of them. Then it removes the priority inconsistencies relevant to the permutation, and returns control to the scheduling algorithm.

Thus the scheduling algorithm generates a feasible bit string in cooperation with the global harmonization. It is not always guaranteed that the above harmonization will generate a feasible bit string closest to the original illegal one, but the resulting one will be reasonably close and the harmonization algorithms are quite efficient.

7.3.3 Forcing

An illegal bit string produced by genetic operations can be considered as a genotype, and a feasible bit string generated by any repairing method can be regarded as a phenotype. Then the former is an inherited character and the latter is an acquired one. Note that the repairing stated above is only used for the fitness evaluation of the original bit string; that is, the repairing does not mean the replacement of bit strings.

Forcing means the replacement of the original string with a feasible one. Hence forcing can be considered as the inheritance of an acquired character, although it is not widely believed that such inheritance occurs in nature. Since frequent forcing may destroy whatever potential and diversity of the population, it is limited to a small number of elites. Such limited forcing brings about at least two merits: a significant improvement in the convergence speed and the solution quality. Experiments have shown how it works [19].

7.4 Permutation representation

As described in Section 7.2, the JSSP can be viewed as an ordering problem just like the Traveling Salesman Problem (TSP). For example, a schedule can be represented by the set of permutations of jobs on each machine, in other words, m-partitioned permutations of operation numbers, which is called a *job sequence matrix*. Table 7.6 shows a job sequence matrix of the same solution as that given in Figure 7.1. The advantage of this representation is that the GA operators used to solve the TSP can be applied without further modifications, because each job sequence is equivalent to the path representation in the TSP.

M_1	M_2	M_3
1 2 3	3 1 2	2 1 3

Figure 7.6: A job sequence matrix for a 3×3 problem

7.4.1 Subsequence exchange crossover

A crossover operator called the *Subsequence Exchange Crossover* (SXX) was proposed by Kobayashi, Ono and Yamamura [15]. The SXX is a natural extension of the subtour exchange crossover for TSPs presented by the same authors [14]. Let two job sequence matrices be p_0 and p_1. A pair of subsequences, one from p_0 and the other from p_1 on the same machine, is called *exchangeable* if and only if they consist of the same set of jobs. The SXX searches for exchangeable subsequence pairs in p_0 and p_1 on each machine and interchanges each pair to produce new job sequence matrices k_0 and k_1. Figure 7.7 shows an example of the SXX for a 6×3 problem.

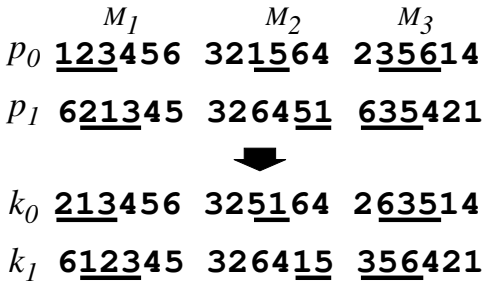

Figure 7.7: Subsequence Exchange Crossover (SXX)

If all jobs in a job subsequence s_0 in p_0 on a machine are positioned consecutively in s_1 in p_1, s_0 and s_1 are exchangeable. By checking for all s_0 in p_0 systematically, if there exists a corresponding s_1 in p_1, all of the exchangeable subsequence pairs in p_0 and p_1 on the machine can be enumerated in $O(n^2)$ [29], so the SXX requires a computational complexity of $O(mn^2)$.

Although a job sequence matrix obtained from the SXX always represents valid job permutations, it not necessarily represents a schedule. To obtain a schedule from illegal offspring, some repairing mechanism such as the global harmonization described in Section 7.3 is also required. Instead of using the global harmonization, the GT algorithm is used as a repairing mechanism together with the described forcing, to modify any job sequence matrix into an active schedule. A small number of swap operations designated by the GT algorithm are applied to repair job sequence matrices.

7.4.2 Permutation with repetition

Instead of using an m-partitioned permutation of operation numbers, like the job sequence matrix defined in the previous subsection, another representation that uses an *unpartitioned permutation with m-repetitions* of job numbers was employed by Bierwirth [6]. In this permutation, each job number occurs m times. By scanning the permutation from left to right the k-th occurrence of a job number refers to

the k-th operation in the technological sequence of this job (see Figure 7.8). In this representation, it is possible to avoid schedule operations whose technological predecessors have not been scheduled yet. Therefore any individual is decoded to a schedule, but two or more different individuals can be decoded to an identical schedule.

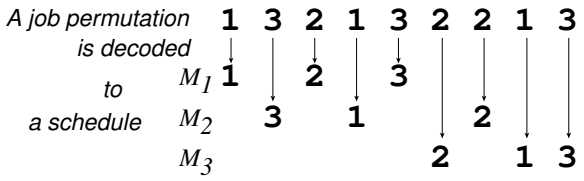

Figure 7.8: A job sequence (permutation with repetition) for a 3×3 problem is decoded to a schedule, which is equivalent to the one in Figure 7.1.

The well used Order Crossover and Partially Mapped Crossover for TSP are naturally extended for this representation (they are called the *Generalized Order Crossover* (GOX) and *Generalized Partially Mapped Crossover* (GPMX)). A new *Precedence Preservative Crossover* (PPX) is also proposed in [7]. The PPX perfectly respects the absolute order of genes in parental chromosomes. A template bit string h of length mn is used to define the order in which genes are drawn from p_0 and p_1. A gene is drawn from one parent and it is appended to the offspring chromosome. The corresponding gene is deleted in the other parent (See Figure 7.9). This step is repeated until both parent chromosomes are empty and the offspring contains all genes involved. The idea of forcing described in Section 7.3 is combined with the permissible left shift described in Subsection 7.2.1: new chromosomes are modified to active schedules by applying permissible left shifts.

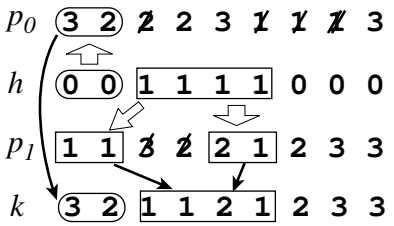

Figure 7.9: Precedence Preservative Crossover (PPX)

7.5 Heuristic crossover

The earlier sections were devoted to representing solutions in generic forms such as bit strings or permutations so that conventional crossover operators could be

Algorithm 7.5.1 GT crossover

1. Same as Step 1. of Algorithm 7.2.1.
2. Same as Step 2. of Algorithm 7.2.1.
3. Select one of the parent schedules $\{p_0, p_1\}$ according to the value of H_{ir} as $p = p_{H_{ir}}$. Select an operation $O \in C[M_r, i]$ that has been scheduled in p earliest among $C[M_r, i]$.
4. Same as Step 4. of Algorithm 7.2.1.

applied without further modifications. Because of the complicated constraints of a problem, however individuals generated by a crossover operator are often infeasible and require several steps of a repairing mechanism. The following properties are common to these approaches:

- Crossover operators are problem independent and they are separated from schedule builders.
- An individual does not represent a schedule itself but its gene codes give a series of decisions for a schedule builder to generate a schedule.

Obviously one of the advantages of the GA is its robustness over a wide range of problems with no requirement of domain specific adaptations. Therefore the crossover operators should be domain independent and separated from domain specific schedule builders. However from the viewpoint of performance, it is often more efficient to directly incorporate domain knowledge into the algorithm to skip wasteful intermediate decoding steps. Thus the GT crossover proposed by Yamada and Nakano [30] has the following properties instead.

- The GT crossover is a problem dependent crossover operator that directly utilizes the GT algorithm. In the crossover, parents cooperatively give a series of decisions to the algorithm to build new offsprings, namely active schedules.
- An individual represents an active schedule, so there is no repairing scheme required.

7.5.1 GT crossover

Let H be a binary matrix of size $n \times m$ [30, 10]. Here $H_{ir} = 0$ means that the i-th operation on machine r should be determined by using the first parent and $H_{ir} = 1$ by the second parent. The role of H_{ir} is similar to that of h described in Section 7.4.2. Let the parent schedules be p_0 and p_1 as always. The GT crossover

can be defined by modifying Step 3 of Algorithm 7.2.1 as shown in Algorithm 7.5.1. It tries to reflect the processing order of the parent schedules to their offspring. It should be noted that if the parents are identical to each other, the resulting new schedule is also identical to the parents'. In general the new schedule inherits partial job sequences of both parents in different ratios depending on the number of 0's and 1's contained in H.

The GT crossover generates only one schedule at once. Another schedule is generated by using the same H but changing the roles of p_0 and p_1. Thus two new schedules are generated that complement each other. The outline of the GT crossover is described in Figure 7.10.

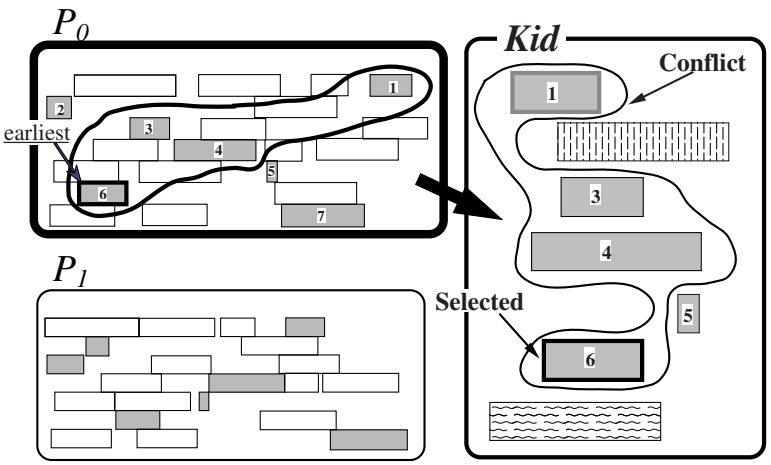

Figure 7.10: GT crossover

Mutation can be put in Algorithm 7.5.1 by occasionally selecting the n-th ($n > 1$) earliest operation in $C[M_{r^*}, i]$ with a low probability inversely proportional to n in Step 3 of Algorithm 7.5.1.

7.6 Genetic enumeration

A method to use bit string representation and simple crossover used in simple GAs, and at the same time to incorporate problem specific heuristics was proposed by Dorndorf and Pesch [12, 22]. They interpret an individual solution as a sequence of decision rules for domain specific heuristics such as the GT algorithm and the shifting bottleneck procedure.

7.6.1 Priority rule based GA

Priority rules are the most popular and simplest heuristics for solving the JSSP. They are rules used in Step 3 of Algorithm 7.2.1 to resolve a conflict by selecting

Algorithm 7.6.1 The shifting bottleneck procedure (SB I)

1. Set $S = \emptyset$ and make all machines unsequenced.
2. Solve a one-machine scheduling problem for each unsequenced machine.
3. Among the machines considered in Step 2, find the bottleneck machine and add its schedule to S. Make the machine sequenced.
4. Reoptimize all sequenced machines in S.
5. Go to Step 3 unless S is completed; otherwise stop.

an operation O from the conflict set $C[M_r, i]$. For example, a priority rule called "SOT-rule" (shortest operation time rule) selects the operation with the shortest processing time from the conflict set. Twelve such simple rules are used in [12, 22] including the SOT-rule, LRPT-rule (longest remaining processing time rule) and FCFS-rule (first come first serve rule) such that they are partially complementary in order to select each member in the conflict set.

Each individual of the *priority rule based GA* (P-GA) is a string of length $mn - 1$, where the entry in the i-th position represents one of the 12 priority rules used to resolve the conflict in the i-th iteration of the GT algorithm. A simple crossover that exchanges the substrings of two cut strings are applied.

7.6.2 Shifting bottleneck based GA

The Shifting bottleneck (SB) proposed by Adams et al. [1] is a powerful heuristic for solving the JSSP. In the method, a one-machine scheduling problem (a relaxation of the original JSSP) is solved for each machine not yet sequenced, and the outcome is used to find a bottleneck machine, i.e. a machine having the longest makespan. Every time a new machine has been sequenced, the sequence of each previously sequenced machine is subject to reoptimization. The SB consists of two subroutines: the first one (SB I) repeatedly solves one-machine scheduling problems; the second one (SB II) builds a partial enumeration tree where each path from the root to a leaf is similar to an application of SB I. The outline of the SB I is described in Algorithm 7.6.1. Please refer to [1, 2, 33] as well as [12, 22] for more details.

Besides using the genetic algorithm as a metastrategy to optimally control the use of priority rules, another genetic algorithm described in Dorndorf and Pesch [12, 22] controls the selection of nodes in the enumeration tree of the shifting bottleneck heuristic; it is called the *shifting bottleneck based genetic algorithm* (SB-GA). Here an individual is represented by a permutation of machine numbers $1 \ldots m$, where the entry in the i-th position represents the machine selected in Step 3 in place of

a bottleneck machine in the i-th iteration of Algorithm 7.6.1. A cycle crossover operator is used as the crossover for this permutation representation.

7.7 Genetic local search

It is well known that GAs can be enhanced by incorporating local search methods, such as neighborhood search into them. The result of such an incorporation is often called *Genetic Local Search (GLS)* [26]. In this framework, an offspring obtained by a recombination operator, such as crossover, is not included in the next generation directly but is used as a "seed" for the subsequent local search. The local search moves the offspring from its initial point to the nearest locally optimal point, which is included in the next generation.

This section briefly reviews the basics of neighborhood search, neighborhood structures for the JSSP and an approach to incorporate a local neighborhood search into a GA to solve the problems.

7.7.1 Neighborhood search

Neighborhood search is a widely used local search technique to solve combinatorial optimization problems. A solution x is represented as a point in the search space, and a set of solutions associated with x is defined as neighborhood $N(x)$. $N(x)$ is a set of feasible solutions reachable from x by exactly one transition, i.e. a single perturbation of x.

An outline of a neighborhood search for minimizing $V(x)$ is described in Algorithm 7.7.1, where x denotes a point in the search space and $V(x)$ denotes its evaluation value.

Algorithm 7.7.1 Neighborhood search

- Select a starting point: $x = x_0 = x_{best}$.

do

1. Select a point $y \in N(x)$ according to the given criterion based on the value $V(y)$. Set $x = y$.
2. If $V(x) < V(x_{best})$ then set $x_{best} = x$.

until some termination condition is satisfied.

The criterion used in Step 1 in Algorithm 7.7.1 is called the *choice criterion*, by which the neighborhood search can be categorized. For example, a descent method selects a point $y \in N(x)$ such that $V(y) < V(x)$. A stochastic method probabilistically selects a point according to the Metropolis Criterion, i.e. $y \in N(x)$

Algorithm 7.7.2 Multi-Step Crossover Fusion (MSXF)

- Let p_0, p_1 be parent solutions.
- Set $x = p_0 = q$.

do
- For each member $y_i \in N(x)$, calculate $d(y_i, p_1)$.
- Sort $y_i \in N(x)$ in ascending order of $d(y_i, p_1)$.

do
1. Select y_i from $N(x)$ randomly, but with a bias in favor of y_i with a small index i.
2. Calculate $V(y_i)$ if y_i has not yet been visited.
3. Accept y_i with probability one if $V(y_i) \leq V(x)$, and with $P_c(y_i)$ otherwise.
4. Change the index of y_i from i to n, and the indexes of y_k ($k \in \{i+1, i+2, \ldots, n\}$) from k to $k-1$.

until y_i is accepted.
- Set $x = y_i$.
- If $V(x) < V(q)$ then set $q = x$.

until some termination condition is satisfied.

- q is used for the next generation.

is selected with probability 1 if $V(y) < V(x)$; otherwise, with probability:

$$P(y) = exp(-\Delta V/T), \text{ where } \Delta V = V(y) - V(x) \,. \tag{7.1}$$

Here P is called the *acceptance probability*. Simulated Annealing (SA) is a method in which parameter T (called the *temperature*) decreases to zero following an annealing schedule as the iteration step increases.

7.7.2 Multi-Step Crossover Fusion

Reeves has been exploring the possibility of integrating local optimization directly into a Simple GA with bit string representations and has proposed the Neighborhood Search Crossover (NSX) [23]. Let any two individuals be x and z. An individual y is called *intermediate* between x and z, written as $x \diamond y \diamond z$, if and only if $d(x,z) = d(x,y) + d(y,z)$ holds, where x, y and z are represented in binary strings and $d(x,y)$ is the Hamming distance between x and y. Then the *k^{th}-order 2 neighborhood* of x and z is defined as the set of all intermediate individuals at a Hamming distance of

k from either x or z. Formally,

$$N_k(x,z) = \{y \mid x \diamond y \diamond z \text{ and } (d(x,y) = k \text{ or } d(y,z) = k)\}.$$

Given two parent bit strings p_0 and p_1, the neighborhood search crossover of order k (NSX$_k$) will examine all individuals in $N_k(p_0, p_1)$, and pick the best as the new offspring.

Yamada and Nakano extended the idea of the NSX to make it applicable to more complicated problems such as job-shop scheduling and proposed the Multi-Step Crossover Fusion (MSXF): a new crossover operator with a built-in local search functionality [31, 34, 32]. The MSXF has the following characteristics compared to the NSX.

- It can handle more generalized representations and neighborhood structures.
- It is based on a stochastic local search algorithm.
- Instead of restricting the neighborhood by a condition of intermediateness, a biased stochastic replacement is used.

A stochastic local search algorithm is used for the base algorithm of the MSXF. Although the SA is a well-known stochastic method and has been successfully applied to many problems as well as to the JSSP, it would be unrealistic to apply the full SA to suit our purpose because it would consume too much time by being run many times in a GA run. A restricted method with a fixed temperature parameter $T = c$ might be a good alternative. Accordingly, the acceptance probability used in Algorithm 7.7.1 is rewritten as:

$$P_c(y) = exp\left(-\frac{\Delta V}{c}\right), \Delta V = V(y) - V(x), c : \text{const.} \tag{7.2}$$

Let the parent schedules be p_0 and p_1, and let the distance between any two individuals x and y in any representation be $d(x,y)$. If x and y are schedules, then $d(x,y)$ is the DG distance. Crossover functionality can be incorporated into Algorithm 7.7.1 by setting $x_0 = p_0$ and adding a greater acceptance bias in favor of $y \in N(x)$ having a small $d(y,p_1)$. The acceptance bias in the MSXF is controlled by sorting $N(x)$ members in ascending order of $d(y_i,p_1)$ so that y_i with a smaller index i has a smaller distance $d(y_i,p_1)$. Here $d(y_i,p_1)$ can be estimated easily if $d(x,p_1)$ and the direction of the transition from x to y_i are known; it is not necessary to generate and evaluate y_i. Then y_i is selected from $N(x)$ randomly, but with a bias in favor of y_i with a small index i. The outline of the MSXF is described in Algorithm 7.7.2.

In place of $d(y_i,p_1)$, one can also use $sign(d(y_i,p_1) - d(x,p_1)) + r_\epsilon$ to sort $N(x)$ members in Algorithm 7.7.2. Here $sign(x)$ denotes the sign of x: $sign(x) = 1$ if $x > 0$, $sign(x) = 0$ if $x = 0$, $sign(x) = -1$ otherwise. A small random fraction

r_ϵ is added to randomize the order of members with the same sign. The termination condition can be given, for example, as the fixed number of iterations in the outer loop.

The MSXF is not applicable if the distance between p_0 and p_1 is too small compared to the number of iterations. In such a case, a mutation operator called the *Multi-Step Mutation Fusion* (MSMF) is applied instead. The MSMF can be defined in the same manner as the MSXF is except for one point: the bias is reversed, i.e. sort the $N(x)$ members in descending order of $d(y_i, p_1)$ in Algorithm 7.7.2.

7.7.3 Neighborhood structures for the JSSP

For the JSSP, a neighborhood $N(S)$ of a schedule S can be defined as a set of schedules that can be reached from S by exactly one transition (a single perturbation of S).

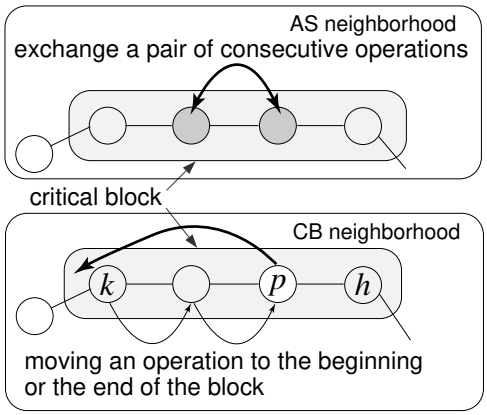

Figure 7.11: Permutation of operations on a critical block

As shown in Section 7.3, a set of solutions of the JSSP can be mapped to a space of bit strings by marking each disjunctive arc as 1 or 0 according to its direction. The DG distance and the Hamming distance in the mapped space are equivalent, and the neighborhood of a schedule S is a set of all (possibly infeasible) schedules whose DG distances from S are exactly one. Neighborhood search using this binary neighborhood is simple and straightforward but not very efficient.

More efficient methods can be obtained by introducing a transition operator that exchanges a pair of consecutive operations only on the critical path and forms a neighborhood [16, 25]. The transition operator was originally defined by Balas in his branch and bound approach [4]. We call this the *adjacent swapping* (AS) neighborhood. DG distances between a schedule and members of its AS neighborhood are always one, so the AS neighborhood can be considered as a subset of the bit string neighborhood.

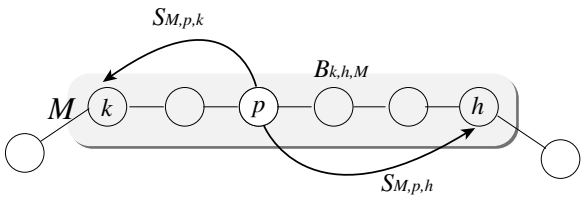

Figure 7.12: $S_{M,p,k}$ and $S_{M,p,h}$ generation

Another very powerful transition operator was used in [9, 11]. The transition operator permutes the order of operations in a critical block by moving an operation to the beginning or end of the critical block, thus forming a ***CB neighborhood***.

A schedule obtained from S by moving an operation within a block to the front of the block is called a *before candidate*, and a schedule obtained by moving an operation to the rear of the block is called an *after candidate*. A set of all before and after candidates ${N'}^C(S)$ may contain infeasible schedules. The CB neighborhood is given as:

$$N^C(S) = \{S' \in {N'}^C(S) \mid S' \text{ is a feasible schedule}\}.$$

It has been experimentally shown by [35] that the CB neighborhood is more powerful than the former one.

Active CB neighborhood

As explained above, before or after candidates are not necessarily executable. In the following, a new neighborhood similar to the CB neighborhood is used, each element of which is not only executable, but also active and close to the original. Let S be an active schedule and $B_{k,h,M}$ be a critical block of S on a machine M, where the front and the rear operations of $B_{k,h,M}$ are the k-th and the h-th operations on M, respectively. Let $O_{p,M}$ be an operation in $B_{k,h,M}$ that is the p-th operation on M. Algorithm 7.7.3 generates an active schedule $S_{M,p,k}$ (or $S_{M,p,h}$) by modifying S such that $O_{p,M}$ is moved to the position as close to the front position k (or the rear position h) of $B_{k,h,M}$ as possible. Parts of the algorithm are due to [11]. The new *active CB neighborhood* $AN^C(S)$ is now defined as a set of all $S_{M,p,k}$'s and $S_{M,p,h}$'s over all critical blocks:

$$AN^C(S) = \bigcup_{B_{k,h,M}} \{S' \in \{S_{M,p,k}\}_{k<p<h} \cup \{S_{M,p,h}\}_{k<p<h}, S' \neq S\}.$$

7.7.4 Scheduling in the reversed order

Algorithm 7.2.1 and all its variations determine the job sequences from left to right in temporal order. This is because active schedules are defined to have no extra idle

Algorithm 7.7.3 Modified GT algorithm generating $S_{M,p,k}$ or $S_{M,p,h}$

1. Same as Step 1. of Algorithm 7.2.1.
2. Same as Step 2. of Algorithm 7.2.1.
3. Do **CASE 1** (or **CASE 2**) to generate $S_{M,p,k}$ (or $S_{M,p,h}$).

   CASE 1: $S_{M,p,k}$ generation

   - *If* $k \leq i \leq p$ and $O_{p,M} \in C[M_r, i]$, then set $O = O_{p,M}$.
   - *Otherwise*, select an operation $O \in C[M_r, i]$ that has been scheduled in S earliest among $C[M_r, i]$.

   CASE 2: $S_{M,p,h}$ generation

   - *If* $i = h$ or $C[M_r, i] = \{O_{p,M}\}$, then set $O = O_{p,M}$.
   - *Otherwise*, select an operation $O \in C[M_r, i] \setminus O_{p,M}$ that has been scheduled in S earliest among $C[M_r, i] \setminus O_{p,M}$.
4. Same as Step 4. of Algorithm 7.2.1.

periods of machines *prior to* their operations. However the idea described below enables the same algorithms to determine the job sequences from right to left with only small modifications.

In general, a given problem of the JSSP can be converted to another problem by reversing all of the technological sequences. The new problem is equivalent to the original one in the sense that reversing the job sequences of any schedule for the original problem results in a schedule for the reversed problem with the same critical path and makespan. It can be seen, however, that an active schedule for the original problem may not necessarily be active in the reversed problem: the activeness is not necessarily preserved.

job	Routing	
1	2(3)	1(4)
2	2(2)	1(4)

Figure 7.13: A simple 2×2 problem

For example, the simple 2×2 problem described in Table 7.13 is considered. Figure 7.14(1) shows a solution of this problem, which is active and no more left shifts can improve its makespan. Figure 7.14(2), obtained by reversing

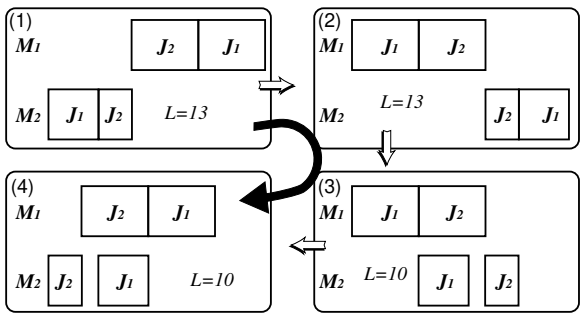

Figure 7.14: Schedule reversal and activation

Figure 7.14(1), is not active and can be improved by a left shift that moves job 1 prior to job 2 on machine 2, resulting in Figure 7.14(3). Finally Figure 7.14(4) is obtained by reversing Figure 7.14(3) again, which is optimal. As things turn out, Figure 7.14(1) is improved by moving job 1 *posterior* to job 2 on machine 2, resulting in Figure 7.14(4).

Although repairing a semi-active schedule to the active one improves the makespan, it can be seen from the example above that there sometimes are obvious improvements that cannot be attained only by left shifts. We call a schedule *left* active if it is an active schedule for the original problem and *right* active if it is such for the reversed problem. It sometimes happens that a reserved problem is easier to solve compared to the original. Searching only in the set of left (or right) active schedules may bias the search toward the wrong direction and result in poor local minima. Therefore left active schedules as well as right active ones should be taken into account together in the same algorithm. In most local search methods, many schedules are generated in a single run; therefore it would be better to apply this reversing and repairing method periodically to change the scheduling directions rather than to reverse and repair every schedule each time it is generated.

7.7.5 MSXF-GA for Job-shop Scheduling

The MSXF is applied to the JSSP by using the active CB neighborhood and the DG distance previously defined. Algorithm 7.7.4 describes the outline of the MSXF-GA routine for the JSSP using the steady state model proposed in [28, 24]. To avoid premature convergence even under a small-population condition, an individual whose fitness value is equal to someone's in the population is not inserted into the population in Step 4.

A mechanism to search in the space of both the left and right active schedules is introduced into the MSXF-GA as follows. First, there are equal numbers of left and right active schedules in the initial population. The schedule q generated from p_0 and p_1 by the MSXF ought to be left (or right) active if p_0 is left (or right) active, and with some probability (0.1 for example) the direction is reversed.

Algorithm 7.7.4 MSXF-GA for the JSSP

- Initialize population: randomly generate a set of *left* and *right* active schedules in equal number and apply the local search to each of them.

do

1. Randomly select two schedules p_0, p_1 from the population with some bias depending on their makespan values.
2. Change the direction (*left* or *right*) of p_1 by reversing the job sequences with probability P_r.
3. Do step (3a) with probability P_c, or otherwise do Step (3b).
   (a) **If** the DG distance between p_1, p_2 is shorter than some predefined small value, apply MSMF to p_1 and generate q.
   Otherwise, apply MSXF to p_1, p_2 using the active CB neighborhood $N(p_1)$ and the DG distance and generate a new schedule q.
   (b) Apply Algorithm 7.7.1 with acceptance probability given by Equation 7.2 and the active CB neighborhood.
4. If q's makespan is shorter than the worst in the population, and no one in the population has the same makespan as q, replace the worst individual with q.

until some termination condition is satisfied.

- Output the best schedule in the population.

Figure 7.15 shows all of the solutions generated by an application of (a) the MSXF and (b) a stochastic local search computationally equivalent to (a) for comparison. Both (a) and (b) started from the same solution (the same parent p_0), but in (a) transitions were biased toward the other solution p_1. The x axis represents the number of disjunctive arcs whose directions are different from those of p_1 on machines with odd numbers, i.e. the DG distance was restricted to odd machines. Similarly, the y axis representing the DG distance was restricted to even machines.

7.8 Benchmark problems

The two well-known benchmark problems with sizes of 10×10 and 20×5 (known as mt10 and mt20) formulated by Muth and Thompson [18] are commonly used as test beds to measure the effectiveness of a certain method. The mt10 problem used to be called a "notorious" problem, because it remained unsolved for over 20 years; however it is no longer a computational challenge.

Applegate and Cook proposed a set of benchmark problems called the "ten

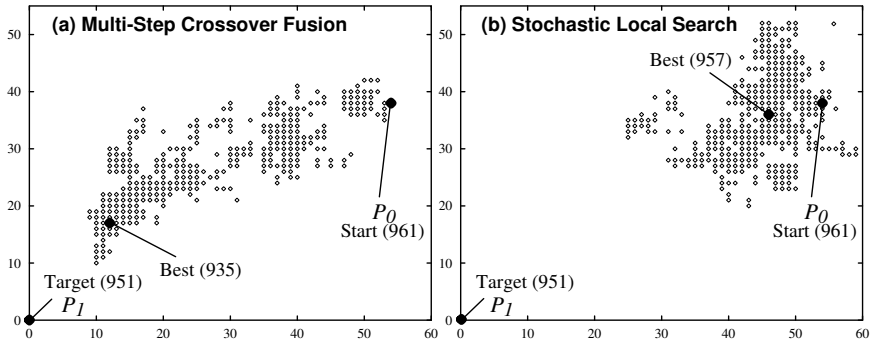

Figure 7.15: Distribution of solutions generated by an application of (a) MSXF and (b) a short-term stochastic local search

tough problems" as a more difficult computational challenge than the mt10 problem, by collecting difficult problems from literature, some of which still remain unsolved [3].

7.8.1 Muth and Thompson benchmark

Table 7.2 summarizes the makespan performance of the methods described in this chapter. This table is partially cited from [6]. The Conventional GA has only limited success and is outdated. It would be improved by being combined with the GT algorithm and/or the schedule reversal. The other results excluding the MSXF-GA results are somewhat similar to each other, although the SXX-GA is improved over the GT-GA in terms of speed and the number of times needed to find optimal solutions for the mt10 problem. The SB-GA produces better results using the very efficient and tailored shifting bottleneck procedure. The MSXF-GA which combines a GA and local search obtains the best results.

For the MSXF-GA, the population size $= 10$, constant temperature $c = 10$, number of iterations for each MSXF $= 1000$, $P_r = 0.1$ and $P_c = 0.5$ are used. The MSXF-GA experiments were performed on a DEC Alpha 600 5/226 which is about four times faster than a Sparc Station 10, and the programs were written in the C language. The MSXF-GA finds the optimal solutions for the mt10 and mt20 problems almost every time in less than five minutes on average.

7.8.2 The ten tough benchmark problems

Table 7.3 shows the makespan performance statistics of the MSXF-GA for the ten difficult benchmark problems proposed in [3]. The parameters used here were the same as those for the MT benchmark except for the population size $= 20$. The algorithm was terminated when an optimal solution was found or after 40 minutes of cpu time passed on the DEC Alpha 600 5/266. In the table, the column named lb shows

Table 7.2: Performance comparison using the MT benchmark problems

1963	Muth-Thompson	Test problems	10×10	20×5
1991	Nakano/Yamada	Conventional GA	965	1215
1992	Yamada/Nakano	Giffler-Thompson GT-GA	930	1184
	Dorndorf/Pesch	Priority-Rule based P-GA	960	1249
	Dorndorf/Pesch	Shifting-Bottleneck SB-GA	938	1178
1995	Kobayashi/Ono /Yamamura	Subsequence Exchange Crossover SXX-GA	930	1178
1995	Bierwirth	Generalized-Permutation GP-GA	936	1181
1996	Yamada/Nakano	Multi-step Crossover Fusion MSXF-GA	930	1165

the known lower bound or known optimal value (for la40) of the makespan, and the columns named bst, avg, var and wst show the best, average, variance and worst makespan values obtained, over 30 runs respectively. The columns named n_{opt} and t_{opt} show the number of runs in which the optimal schedules are obtained and their average cpu times in seconds. The problem data and lower bounds are taken from the OR-library [5]. Optimal solutions were found for half of the ten problems, and four of them were found very quickly. The small variances in the solution qualities indicate the stability of the MSXF-GA as an approximation method.

Table 7.3: Results of the 10 tough problems

prob	size	lb	bst	avg	var	wst	n_{opt}	t_{opt}
abz7	20×15	655	678	692.5	0.94	703	–	–
abz8	20×15	638	686	703.1	1.54	724	–	–
abz9	20×15	656	697	719.6	1.53	732	–	–
la21	15×10	–	*1046	1049.9	0.57	1055	9	687.7
la24	15×10	–	*935	938.8	0.34	941	4	864.1
la25	20×10	–	*977	979.6	0.40	984	9	765.6
la27	20×10	–	*1235	1253.6	1.56	1269	1	2364.75
la29	20×10	1130	1166	1181.9	1.31	1195	–	–
la38	15×15	–	*1196	1198.4	0.71	1208	21	1051.3
la40	15×15	*1222	1224	1227.9	0.43	1233	–	–

Figure 7.16 shows a performance comparison of with and without the MSXF using the la38 problem. A total of 100 runs were done for each under the same conditions used in Table 7.3. The solid line gives the MSXF-GA's results and the dotted line gives the equivalent GLS's results using a short-term stochastic local search. The y axis shows the cpu time at which each run is terminated and the x

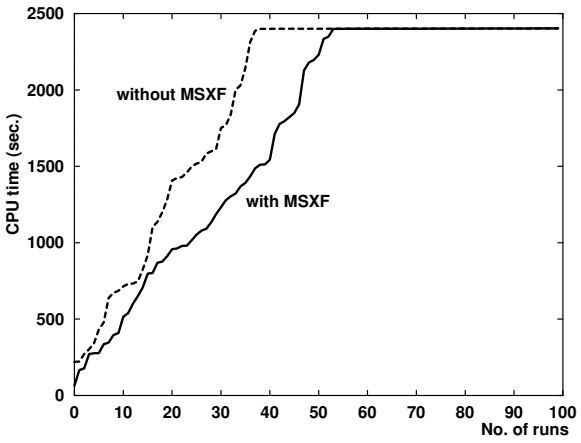

Figure 7.16: Performance comparison using the la38 15×15 problem

axis shows the run numbers which are sorted in ascending order according to the cpu times. The cpu time value $= 2400$ means that the run was terminated before it found the optimal schedule. The experiments with the MSXF outperformed those without the MSXF both in terms of the cpu time and in the number of successful runs.

7.9 Other heuristic methods

Local search based meta-heuristics are commonly applied to the JSSP such as simulated annealing (SA) and tabu search (TS). Van Laarhoven et al. [16] proposed a SA approach by using the AS neighborhood described in Subsection 7.7.3. Matsuo et al. proposed a similar SA approach but with more control. Taillard proposed a TS approach that uses the same neighborhood. Dell'Amico and Trubian extended and improved Taillard's TS method using CB neighborhood. More recently, Nowicki and Smutnicki [20] proposed a still more powerful TS method. Yamada and Nakano proposed a SA approach combined with the shifting bottleneck and improved the best solutions for the two problems abz9 and la29 of the 10 tough problems [33]. Balas and Vazacopoulos proposed the guided local search procedure and combined it with the shifting bottleneck, which at present outperforms most existing methods. For more comprehensive reviews, please refer to [17], [27] and [8].

7.10 Conclusions

The first serious application of GAs to solve the JSSP was proposed by Nakano and Yamada using a bit string representation and conventional genetic operators. Although this approach is simple and straightforward, it is not very powerful. The

idea to use the GT algorithm as a basic schedule builder was first proposed by Yamada and Nakano [30] and by Dorndorf and Pesch [12, 22] independently. The approaches by both groups and other active schedule-based GAs are suitable for middle-size problems; however, it seems necessary to combine each with other heuristics such as the shifting bottleneck or local search to solve larger-size problems.

To solve larger-size problems effectively, it was crucial to incorporate local search methods that use domain specific knowledge. The multi-step crossover fusion (MSXF) was proposed by Yamada and Nakano as a unified operator of a local search method and a recombination operator in genetic local search. The MSXF-GA outperforms other GA methods in terms of the MT benchmark and is able to find near-optimal solutions for the ten difficult benchmark problems, including optimal solutions for five of them.

Bibliography

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Mgmt. Sci.*, 34(3):391–401, 1988.

[2] D. Applegate. Jobshop benchmark problem set. Personal Communication, 1992.

[3] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA J. on Comput.*, 3(2):149–156, 1991.

[4] E. Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Oper. Res.*, 17:941–957, 1969.

[5] J.E. Beasley. Or-library: distributing test problems by electronic mail. *E. J. of Oper. Res.*, 41:1069–1072, 1990.

[6] C. Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *OR Spektrum*, 17:87–92, 1995.

[7] C. Bierwirth, D. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems (to appear). In *4th PPSN*, 1996.

[8] J. Blazewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *E. J. of Oper. Res.*, pages 1–33, 1996.

[9] P. Brucker, B. Jurisch, and B. Sievers. A branch & bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 1994.

[10] Y. Davidor, T. Yamada, and R. Nakano. The ecological framework II: Improving GA performance at virtually zero cost. In *5th ICGA*, pages 171–176, 1993.

[11] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[12] U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers Ops Res*, 22:25–40, 1995.

[13] B. Giffler and G.L. Thompson. Algorithms for solving production scheduling problems. *Oper. Res.*, 8:487–503, 1960.

[14] M. Kobayashi, T. Ono, and S. Kobayashi. Character-preserving genetic algorithms for traveling salesman problem (in Japanese). *Journal of Japanese Society for Artificial Intelligence*, 7:1049–1059, 1992.

[15] S. Kobayashi, I. Ono, and M. Yamamura. An efficient genetic algorithm for job shop scheduling problems. In *6th ICGA*, pages 506–511, 1995.

[16] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Oper. Res.*, 40(1):113–125, 1992.

[17] Dirk C. Mattfeld. *Evolutionary Search and the Job Shop; Investigations on Genetic Algorithms for Production Scheduling*. Physica Verlag, Heidelberg, Germany, 1996.

[18] J.F. Muth and G.L. Thompson. *Industrial Scheduling*. Prentice-Hall, Englewood Cliffs, N.J., 1963.

[19] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In *4th ICGA*, pages 474–479, 1991.

[20] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem, to appear. *Mgmt. Sci.*, 1995.

[21] S. S. Panwalkar and Wafix Iskander. A survey of scheduling rules. *Oper. Res.*, 25(1):45–61, 1977.

[22] E. Pesch. *Learning in Automated manufacturing: a local search approach*. Physica-Verlag, Heidelberg, Germany, 1994.

[23] C. R. Reeves. Genetic algorithms and neighbourhood search. In *Evolutionary Computing, AISB Workshop (Leeds, U.K.)*, pages 115–130, 1994.

[24] G. Syswerda. Uniform crossover in genetic algorithms. In *3rd ICGA*, pages 2–9, 1989.

[25] E.D. Taillard. Parallel taboo search techniques for the job-shop scheduling problem. *ORSA J. on Comput.*, 6(2):108–117, 1994.

[26] N.L.J. Ulder, E. Pesch, P.J.M. van Laarhoven, H.J. Bandelt, and E.H.L. Aarts. Genetic local search algorithm for the traveling salesman problem. In *1st PPSN*, pages 109–116, 1994.

[27] R.J.M. Vaessens. Generalized job shop scheduling: Complexity and local search. Dissertation, University of Technology Eindhoven, 1995.

[28] D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *3rd ICGA*, pages 116–121, 1989.

[29] M. Yagiura, H. Nagamochi, and T. Ibaraki. Two comments on the subtour exchange crossover operator. *Journal of Japanese Society for Artificial Intelligence*, 10:464–467, 1995.

[30] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In *2nd PPSN*, pages 281–290, 1992.

[31] T. Yamada and R. Nakano. A genetic algorithm with multi-step crossover for job-shop scheduling problems. In First IEE/IEEE International Conference on Genetic ALgorithms in Engineering Systems: Innovations and Applications (GALESIA '95), pages 146–151, 1995.

[32] T. Yamada and R. Nakano. A fusion of crossover and local search (to appear). In *IEEE International Conference on Industrial Technology (ICIT '96)*, 1996.

[33] T. Yamada and R. Nakano. *Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search*. Kluwer academic publishers, MA, USA, 1996.

[34] T. Yamada and R. Nakano. Scheduling by genetic local search with multi-step crossover (to appear). In *4th PPSN*, 1996.

[35] T. Yamada, B.E. Rosen, and R. Nakano. A simulated annealing approach to job shop scheduling using critical block transition operators. In *Proc. IEEE Int. Conf. on Neural Networks (Orlando, Florida.)*, pages 4687–4692, 1994.