Studies
on
Metaheuristics for
Jobshop and Flowshop Scheduling Problems

Takeshi YAMADA

Studies
on
Metaheuristics for
Jobshop and Flowshop Scheduling Problems

Takeshi YAMADA

Submitted in partial fulfillment of
the requirement for the degree of
DOCTOR OF INFORMATICS
(Applied Mathematics and Physics)

KYOTO UNIVERSITY
KYOTO, JAPAN
November, 2003

Preface

Scheduling has been a subject of a significant amount of literature in the operations research field since the early 1950s. The main objective of scheduling is an efficient allocation of shared resources over time to competing activities. Emphasis has been on investigating machine scheduling problems where jobs represent activities and machines represent resources. The problem is not only $\mathcal{NP}$-hard, but also has a well-earned reputation of being one of the most computationally difficult combinatorial optimization problems considered to date. This intractability is one of the reasons why the problem has been so widely studied. The problem was initially tackled by "exact methods" such as the branch and bound method (BAB), which is based on the exhaustive enumeration of a restricted region of solutions containing exact optimal solutions. Exact methods are theoretically important and have been successfully applied to benchmark problems, but sometimes they are quite time consuming even for moderate-scale problems.

With a rapid progress in computer technology, it has become even more important to find practically acceptable solutions by "approximation methods" especially for large-scale problems within a limited amount of time. Stochastic local search methods are such approximation methods for combinatorial optimization. They provide robust approaches to obtain high-quality solutions to problems of realistic sizes in reasonable amount of time. Some of stochastic local search methods are proposed in analogies with the processes in nature, such as statistical physics and biological evolution, and others are proposed in the artificial intelligence contexts. They often work as an iterative master process that guides and modifies the operations of subordinate heuristics; thus they are also called *metaheuristics*. Metaheuristics have been applied to wide variety of combinatorial optimization problems with great successes.

The primary focus of this thesis is applications of metaheuristics, especially Genetic Algorithms (GAs), Simulated Annealing (SA) and Tabu Search (TS), to the jobshop scheduling problem (and the flowshop scheduling problem as its special case) which is among the hardest combinatorial optimization problems. The author hopes that the research in this dissertation will help advance in the understanding of this significant field.

November, 2003
Takeshi Yamada

Acknowledgements

I wish to express my sincere gratitude to Professor Toshihide Ibaraki of Kyoto University for his supervising this thesis. He read the manuscript very carefully and made many valuable suggestions and comments, which improved the accuracy and quality of this thesis. I also thank Professor Masao Fukushima, Professor Yutaka Takahashi, Professor Hiroyuki Kawano, Professor Mutsunori Yagiura and Professor Koji Nonobe of Kyoto University for their useful comments.

The research reported in this thesis was supported by Nippon Telegraph and Telephone Corporation (NTT). I am grateful to Professor Seishi Nishikawa, Professor Tsukasa Kawaoka, Dr. Kohichi Matsuda, Professor Yoh'ichi Tohkura, Professor Kenichiro Ishii, former directors of NTT Communication Science Laboratories, Dr. Noboru Sugamura and Dr. Shigeru Katagiri, director and vice director of NTT Communication Science Laboratories, for their warm encouragement and for providing me the opportunity to study these interesting subjects.

I am deeply indebted to Professor Ryohei Nakano of Nagoya Institute of Technology. He had been my supervisor for more than ten years since I first started my research career at NTT fifteen years ago. This thesis would not have been possible without his support and encouragement. I am also indebted to Professor Colin Reeves of Coventry University. Some of the work have been done while I was working with him as a visiting researcher at the university in 1996. I wish to express my many thanks to Professor Bruce Rosen of University of California. The collaboration with him while he was visiting NTT in 1994 is very important especially for the early stage of the work.

I am also grateful to Dr.Ueda and Dr.Saito of NTT Communication Science Laboratories for their encouragement and long standing friendship.

Finally, I thank my parents for their endless support and encouragement, and my wife Kazumi to whom I dedicate this work, for everything else.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In late 1950s, B.Giffler and G.L.Thompson first showed in their paper titled “Algorithms for solving production scheduling problems” [2] that it is not necessary to search for an optimal schedule over all possible schedules, but only over a subset of the feasible schedules, called active schedules. They then proposed the GT algorithm, which is described later in Section 2.2, to iteratively enumerate all active schedules for a given problem. H.Fisher and G.L.Thompson proposed three well-known benchmark problems known as mt06 (6 jobs 6 machines), mt10 (10 jobs 10 machines) and mt20 (20 jobs 5 machines) [3] in a book titled “Industrial Scheduling” edited by J.F.Muth and G.L.Thompson in 1963 [4]. The “notorious” mt10 problem has been unsolved for over 20 years. Their paper is also important in the sense that they first applied a stochastic approximation method based on priority dispatching rules and active schedules.

As a pioneering work in the jobshop scheduling research, G.H.Brooks and C.R.White first proposed a branch and bound method, a tree-based exact method to solve the problem optimally, based on the GT algorithm [5]. E.Balas first pointed out the fact that the adjacent pairwise exchange of operations on a certain part of the schedule, called “critical path”, of a feasible schedule will always results in another new feasible schedule [6]. This fact will later play an important role in metaheuristics context.

A great deal of efforts by Barker and McMahon [7] and then Carlier [8] among others have contributed the progress of the exact approaches, which are mainly based on the branch and bound method. They have commonly used the mt benchmark problems (especially the mt10 problem) as a computational challenge to demonstrate the effectiveness of their algorithms, and the best known solution for the mt10 problem has been improved. Finally in 1985, Carlier and Pinson succeeded in solving the mt10 problem optimally by a branch and bound algorithm [8]. Since then, Brucker [9], Martin and Shmoys [10], and Carlier again [11] have improved the performance of exact approaches. However, the NP-hardness of the problem barriers the efficient application of exact methods to larger-scale problems.

In addtion to those exact methods, many approximation methods have been developed. Simulated annealing (SA) is one of the well-known stochastic local search methods, based on an

analogy with the physical process of annealingl; heating up a solid in a heat bath until it melts, then cooling it down slowly until it solidifies into a low-energy state results in a pure lattice structure. Laarhoven et al. proposed a simulated annealing algorithm for the jobshop scheduling problem, using the pairwise exchange of operations on the critical path proposed by Balas, as a transition operator [12]. However, very similar idea had already been proposed by Matsuo et al. [13].

Adams et al. proposed a very powerful method to find reasonably efficient schedules known as shifting bottleneck heuristic in 1988 [14]. This method, as its name suggests, iteratively identifies a bottleneck machine and optimize its job sequence. The details of the algorithm are described in Section 2.6. In 1991, Applegate have combined the "shifting bottleneck heuristic" and a branch and bound method to develop a powerful algorithm and demonstrated that the mt10 problem is no more a computational challenge. Instead, they proposed a new set of benchmark problems known as the "ten tough benchmarks", which contained the ten difficult problems including seven open problems that were not solved even by their approach [1].

In 1990s, Tabu Search (TS), proposed by Fred Glover [15, 16], has been used by many researchers including Taillard [17], Dell'Amico, Trubian [18], Nowicki and Smutnicki [19, 20]. TS adopts a deterministic local search, whereby a 'memory' is implemented by the recording of previously-seen solutions. Instead of storing solutions explicitly, this record is often an implicit one in a sense that it stores the moves, or the modifications of the solution, that have been made in the recent past of the search, and which are 'tabu' or forbidden for a certain number of iterations. This prevents cycling, and also helps to promote a diversified coverage of the search space. Taillard also proposed a new benchmark set consisting of 80 jobshop and 120 flowshop problems known as "Taillard benchmark" [21].

Genetic Algorithms (GAs) model biological processes to optimize highly complex objective functions. They allow a population composed of many individuals to evolve under specified selection rules to a state that maximizes the "fitness". The method was developed by John Holland over the course of the 1960s and 1970s [22], and popularized by one of his students, David Goldberg who successfully applied a GA to the control of gas-pipeline transmission. He is also well-known for a book titled "Genetic Algorithms in Search, Optimization, and Machine Learning" [23].

GAs have been used successfully in various fields of computer science, including machine learning, control theory and combinatorial optimization. GAs can be uniquely characterized by their population-based search strategies and their operators: mutation, selection and crossover. Nakano and Yamada were among the first who applied a conventional GA that uses binary representation of solutions, to the jobshop scheduling problem [24]. Yamada and Nakano [25] proposed a GA that uses problem-specific representation of solutions with crossover and mutation, which are based on the GT algorithm. The details of these approaches are described later in Chapters 4 and 5.

Ulder and others first proposed Genetic Local Search (GLS), which is a hybridization of GAs and local search [26]. In this framework, each individual, or search agent, performs local search independently, while crossover is performed occasionally to the solutions of two selected individuals and a new solution is produced, which is then used as an initial solution for the subsequent local search performed by an offspring. In this context, the embedded local search is a

main search engine to effectively improve solutions and crossover provides information exchange between individuals who are performing independent local search in parallel.

1.2 Outline of the Thesis

This thesis is devoted to jobshop and flowshop scheduling by metaheuristics, especially by Genetic Algorithms, Simulated Annealing and Tabu Search. In Chapter 2, besic concepts and notations of the jobshop scheduling problem are described such as the semi-active and active schedules, the disjunctive graph representation and critical path and blocks. The main focus throughout this thesis is the *minimum-makespan* problem, in which makespan, maximum completion time of all the operations, is used as an objective function to be minimized. This is denoted as C_{max}. The sum of completion times of all operations, denoted as C_{sum}, is also considered as an alternative objective function of the flowshop scheduling problem. The GT (Giffler & Thompson's) algorithm for generating active schedules and the well-known shifting bottleneck heuristic that generates moderately good schedules by repeatedly solving one-machine scheduling problems are also reviewed as well as some well-known benchmark problems.

In Chapter 3, Genetic Algorithms are reviewed with a major emphasis on conventional binary models for combinatorial optimization, in which a solution is encoded into a binary string of fixed length and binary genetic operators, such as one-point, two-point and uniform crossover and bit-flip mutation, are used. In Chapter 4, a conventional GA using a binary representation is applied to the jobshop scheduling problem. By converting a solution of the problem into a bit-string, conventional GAs can be applied without major modification.

In Chapter 5, a more advanced GA approach is described as the GT-GA method, which involves a non-binary representation of a solution schedule and domain knowledge, namely, active schedules and the GT algorithm. GT-GA method consists of GT crossover and GT mutation that are defined as simple modifications of the GT algorithm. One of the advantages of the GA is its robustness over a wide range of problems with no requirement of domain specific adaptations. From this point of view, the conventional GA approach with binary encoding and binary crossover that is obviously domain independent is reasonable. However, it is often more desireble to directly incorporate problem specific knowledge such as the GT algorithm into GA, resulting the GT-GA method, from the performance point of view.

In Chapter 6, the concept of neighborhood search is described as a widely used local search technique to solve combinatorial optimization problems and is extended to include metaheuristics. Especially, it is shown that Simulated Annealing (SA) and Tabu Search (TS) can be considered as advanced meta strategies for neighorhood search to avoid local optima. An efficient neighborhood for the jobshop scheduling problem, called Critical Block (CB) neighborhood, that is defined based on the critical path and blocks, is also described.

In Chapter 7, a SA method for the jobshop scheduling problem that utilizes the CB neighborhood is described, and then in Chapter 8, it is further extended by incorporating the shifting bottleneck heuristic, which can be regarded as a problem specific deterministic local search method.

In Chapter 9, it is shown that Genetic Algorithms (GAs) can be reagarded as a variant of

neighorhood search, that is called Genetic Local Search (GLS), and an approach called Multi-Step Crossover Fusion (MSXF) method is proposed. In the MSXF method, one of the parent solutions is used as an initial point of the new local search, while the other is used to define an orientation for the search. In other words, it is a local search that traces out a path from one solution to another. The MSXF is applied to the jobshop scheduling problem in Chapter 9 and applied to the C_{max} and C_{sum} flowshop scheduling problems in Chapter 10 and in Chapter 11 respectively.

In Chapter 12, a TS method with a data structure called the Pruning Pattern List (PPL) for the C_{max} flowshop scheduling problem is described. A pruning pattern is constructed from a solution of the flowshop scheduling problem represented by a permutation of jobs numbers by replacing some of its job numbers by a wild card. A list of pruning patterns generated from good schedules collected in the course of a search process is used to inhibit the search to visit already searched and no longer interesting region again and again and it is embedded into a TS method.

Finally, in Chapter 13, the study in this thesis is summerized and the future directions are suggested.

Chapter 2

The Job Shop Scheduling Problem

Scheduling is the allocation of shared resources over time to competing activities. It is convenient to adopt manufacturing terminology, where *jobs* represent activities and *machines* represent resources, while the range of application areas for scheduling theory is not limited to computers and manufacturing but includes transportation, services, etc. In this chapter, we restrict our attention to deterministic jobshop scheduling, where all the data that define a problem instance are known *in advance*.

2.1 The Problem Description

The $n \times m$ *minimum-makespan* general jobshop scheduling problem, designated by the symbols $n/m/G/C_{max}$ and hereafter referred to as the JSP, 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 $\{M_r\}_{1 \leq r \leq m}$. The problem can be characterized as follows:

1. Each job must be processed on each machine in the order given in a pre-defined technological sequence of machines.

2. Each machine can process only one job at a time.

3. The processing of job J_j on machine M_r is called the *operation* O_{jr}.

4. Operation O_{jr} requires the exclusive use of M_r for an uninterrupted duration p_{jr}, its processing time; the preemption is not allowed.

5. The starting time and the completion time of an operation O_{jr} is denoted as s_{jr} and c_{jr} respectively. 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 above constraints.

6. The time required to complete all the jobs is called the *makespan*, which is denoted as C_{max}. By definition, $C_{max} = \max_{1 \leq j \leq n, 1 \leq r \leq m} c_{jr}$.

The problem is "general", hence the symbol G is used, in the sense that the technological sequence of machines can be different for each job as implied in the first condition and that the order of jobs to be processed on a machine can be also different for each machine. The predefined technological sequence of each job can be given collectively as a matrix $\{T_{jk}\}$ in which $T_{jk} = r$ corresponds to the k-th operation O_{jr} of job J_i on machine M_r. The objective of optimizing the problem is to find a schedule that minimizes C_{max}.

An example of a 3×3 JSP is given in Table 2.1. The data include the technological sequence of machines for each job with the processing times in parentheses. In the table, operations for job 1, for example, are processed in the order of $O_{11} \rightarrow O_{12} \rightarrow O_{13}$; i.e., job 1 is first processed on machine 1 with its processing time 3, and then processed on machine 2 with its processing time 3, and then processed on machine 3 with its processing time 3. The problem is equivalently represented by the job sequence matrix $\{T_{jk}\}$ and processing time matrix $\{p_{jk}\}$ as given in Figure 2.1.

Table 2.1: An example of the jobshop scheduling problem with 3 jobs and 3 machines. Each column represents the technological sequence of machines for each job with the processing times in parentheses.

job	machine (processing time)		
1	1 (3)	2 (3)	3 (3)
2	1 (2)	3 (3)	2 (4)
3	2 (3)	1 (2)	3 (1)

$$\{T_{jk}\} = \begin{vmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \end{vmatrix}, \qquad \{p_{jk}\} = \begin{vmatrix} 3 & 3 & 3 \\ 2 & 3 & 4 \\ 3 & 2 & 1 \end{vmatrix}$$

Figure 2.1: The job sequence matrix $\{T_{jk}\}$ and the processing time matrix $\{p_{jk}\}$ for the 3×3 problem given in Table2.1. $T_{jk} = r$ means that k-th operation for job J_j is processed on machine M_r for p_{jk} time units.

Instead of minimizing the makespan C_{max}, other objectives such as minimizing the sum of the completion times of all the operations C_{sum} ($C_{sum} = \sum_{1 \le j \le n, 1 \le r \le m} c_{jr}$) can also be considered and is designated by the symbols $n/m/G/C_{sum}$.

The Gantt chart is a convenient way of visually representing a solution of the JSP. The Gantt chart shows time units at the abscissa and machine numbers at the axis of ordinate. An example of a solution for the 3×3 problem in Table 2.1 is given in Figure 2.2. In the figure, each square box represents an operation O_{jr} with its left edge placed at s_{jr} as its x coordinate and with its horizontal length representing processing time p_{jr}. The makespan of this schedule is $C_{max} = 19$ time unit.

A schedule is called *semi-active* when no operation can be started earlier without altering the operation sequences on any machine. Operation O_{31} in Figure 2.2, for example, can be started as early as at 5 time unit, as indicated by dotted lines, without altering the order of operations on any machine, and the new schedule is semi-active. By definition, a semi-active schedule is uniquely obtained by specifying the operation sequences for all machines. In other words, an semi-active schedule is represented by a $m \times n$ matrix $S = \{S_{rk}\}$ where $S_{rk} = j$ corresponds that the k-th operation on M_r is job J_j. Figure 2.3 shows the matrix representation of a solution for the 3×3 problem in Table 2.1. In the figure, operations on machine 2, for example, are processed in the order of $O_{23} \rightarrow O_{22} \rightarrow O_{12}$. Given a matrix $S = \{S_{rk}\}$, it is straightforward to obtain an associated semi-active schedule. Each operation O has two predecessor operations: job predecessor and machine predecessor. The *job predecessor* of O, denoted by $PJ(O)$, is the direct predecessor of O in the technological sequence. The *machine predecessor* of O, denoted by $PM(O)$, is the direct predecessor of O in the solution matrix. For example, if we have a problem as given in Table 2.1, we have $PJ(O_{12}) = O_{11}$ and if the solution is given as in Figure 2.3, then $PM(O_{12}) = O_{22}$. An operation O is scheduled at time unit 0 if it has no job and no machine predecessors. If only one of job and machine predecessors exists, then O is scheduled immediately when the predecessor is completed. Otherwise it is scheduled when both job and machine predecessors are completed such that: $s(O) = \max\{c(PJ(O)), c(PM(O))\}$, where $s(O)$ and $c(O)$ are the starting time and completion time of operation O respectively. The solution given in Figure 2.3 corresponds to the Gannt chart representation given in Figure 2.2 except that O_{31} is started at 5 time unit.

Throughout this thesis, we assume that a schedule is always semi-active if not stated otherwise. By this formulation, jobshop scheduling can be interpreted as defining the ordering between all operations that must be processed on the same machine, i.e., to fix precedences between these operations. In short, the jobshop scheduling problem is formulated as an ordering problem.

As a special case, when the technological sequence of machines is the same for all jobs and the order in which each machine processes the jobs is also same for all machines, then a schedule is uniquely represented by a permutation of jobs. This simplified problem is called the permutation flowshop scheduling problem and it is designated by the symbols $n/m/P/C_{max}$ (or $n/m/P/C_{sum}$ when the objective is minimizing C_{sum}).

2.2 Active Schedules

The makespan of a semi-active schedule may often be reduced by shifting an operation to the left without delaying other jobs. Consider a semi-active schedule S and two operations O_{jr} and O_{kr} in S that use the same machine M_r. If O_{kr} is processed prior to O_{jr} and the machine M_r has an idle period longer than p_{jr} before processing O_{kr}, then reassignment is possible so that operation O_{jr} is processed prior to O_{kr} without delaying any other operations. Such reassignment is called a *permissible left shift* and a schedule with no more permissible left shifts is called an *active schedule*. Figure 2.4 shows an example of permissible left shift. The schedule in the upper picture of Figure 2.4 is identical to the one given in Figure 2.3 but O_{31} which is started at 5 time unit and its makespan = 19. On machine M_2 in the schedule, O_{12}, the operation of job J_1, can be shifted to

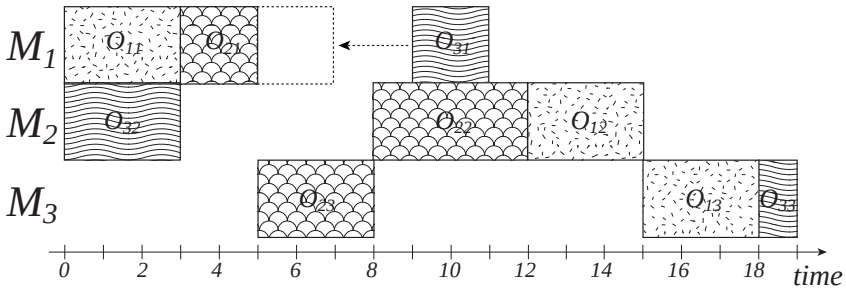

Figure 2.2: A Gantt chart representation of a solution for the 3×3 problem given in Table 2.1. Operation O_{31} can be shifted as early as at 5 time unit, as indicated by dotted lines, without altering the order of operations on any machine, and the new schedule becomes semi-active.

$$\{S_{rk}\} = \begin{vmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{vmatrix}$$

Figure 2.3: The solution matrix S_{rk} for the solution given in Figure2.2. $S_{rk} = j$ means that the k-th operation on machine M_r is job J_j.

the front of O_{32}, the operation of J_3, without delaying any other operations. Operation O_{13} is then shifted to the point immediately after the completion of its job and machine predecessors O_{12} and O_{23}. O_{33} on machine 3 is also shifted. The resulting schedule with improved makespan = 12 is shown in the lower picture of Figure 2.4. Because there always exists an optimal schedule that is active, it should be safe and efficient to restrict the search space to the set of all active schedules.

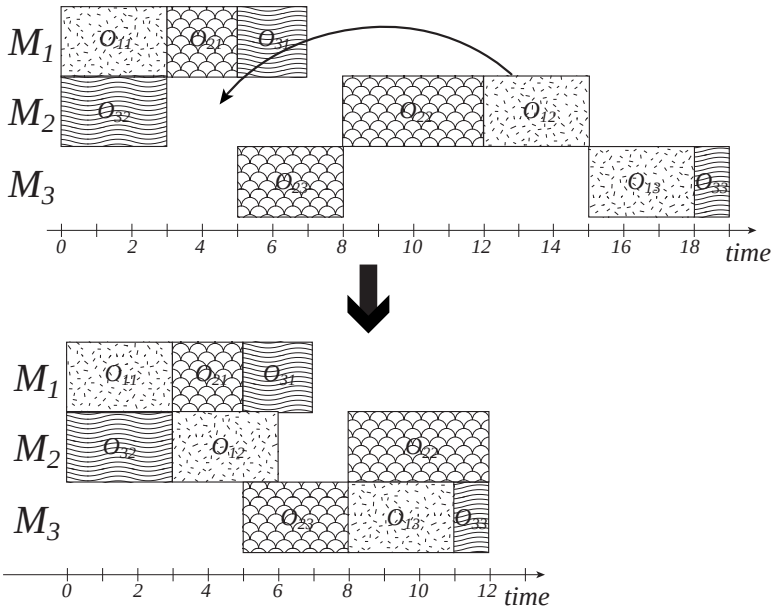

Figure 2.4: An example of a permissible left shift, where in the upper picture, O_{12} can be shifted to the front of O_{32} without delaying any other operations resulted in a much improved schedule given in the lower picture.

An active schedule can be generated by using the *GT algorithm* proposed by Giffler and Thompson [2]. The algorithm is described in Algorithm 2.2.1. In the algorithm, the following notations are used:

- As in the previous section, an operation O has job and machine predecessors. The job predecessor denoted by $PJ(O)$ is the direct predecessor of O in the technological sequence. The definition of the machine predecessor $PM(O)$ is slightly modified and defined as the last scheduled operation on the machine, that is, an operation with the largest completion time among already scheduled operations on the same machine as O.

- An operation O that is not yet scheduled is called *schedulable* when both its job predecessor $PJ(O)$ and machine predecessor $PM(O)$, if they exist, have already been scheduled. The set of all schedulable operations is denoted as G.

- The *earliest starting time* $ES(O)$ of O is defined as the maximum of the completion times of $PJ(O)$ and $PM(O)$: $ES(O) := \max\{c(PJ(O)), c(PM(O))\}$, where $c(O)$ is the completion

time of O. The *earliest completion time* $EC(O)$ is defined as $ES(O)$ plus its processing time $p(O)$.

- The *earliest completable operation* $O_{\star r}$ in D with its machine M_r, is an operation whose earliest completion time $EC(O_{\star r})$ is the smallest in D (break ties arbitrarily) :

$$O_{\star r} = \arg\min\{EC(O) \mid O \in D\}. \tag{2.1}$$

- Given an earliest completable operation $O_{\star r}$ and if there are $i-1$ operations that have already been scheduled on M_r, a *conflict set* $C[M_r, i]$ is a set of candidate operations for the next (i.e. i-th) processing on M_r defined as:

$$C[M_r, i] = \{O_{kr} \in D \mid ES(O_{kr}) < EC(O_{\star r})\}. \tag{2.2}$$

  Note that $O_{\star r} \in C[M_r, i]$.

The essence of GT algorithm is scheduling operations while avoiding an idle period that is long enough to allow a permissible left shift. For this purpose, one has to carefully select the next operation that does not introduce such a long idle period among the set of schedulable operations D. Hence the conflict set $C[M_r, i] \subset D$ is maintained. As long as the next operation is selected from the conflict set, an idle period is kept sufficiently short and the resulting schedule is guaranteed to be active.

An active schedule is obtained by repeating Algorithm 2.2.1 until all operations are scheduled. In Step 4, instead of choosing one operation randomly, if all possible choices are considered, all active schedules will be generated, but the total number will still be very large. Practically, random choice is replaced by the application of so-called *priority dispatching rules* [27], which are the most popular and simplest heuristics for solving the JSP. For example, a rule called *SOT* (shortest operation time) or *SPT* (shortest processing time) selects an operation with the shortest processing time from the conflict set, a rule called *MWKR* (most work remaining) selects an operation associated with the job with the longest total processing time remaining, a rule called *FCFS* (first come first serve rule) selects the first available operation among operations on the same machine. Dorndorf and Pesch [28] proposed a priority rule-based GA for the JSP using twelve such priority dispatching rules.

In the notations above, if the definition of the conflict set $C[M_r, i]$ is simplified as $C[M_r, i] = G_r$, then the generated schedule is an semi-active schedule. Otherwise, if the definition of $C[M_r, i]$ is altered as $C[M_r, i] = \{O_{kr} \in G_r \mid ES(O_{kr}) < ES(O_{\star r})\}$, then the generated schedule is called a *non-delay* schedule. Unlike active schedules, an optimal schedule is not always a non-delay schedule.

Figure 2.5 shows how the GT algorithm works. The figure shows a snapshot in the middle of scheduling where operations $O_{11}, O_{22}, O_{31}, O_{43}, O_{51}$ and O_{64} are schedulable and constitute G. The earliest completable operation is identified as O_{11}, which results in $G_1 = \{O_{11}, O_{31}, O_{51}\}$. In G_1, only O_{11} and O_{31} satisfy the inequality in (2.2), therefore $C[M_1, i] = \{O_{11}, O_{31}\}$. If O_{31} is selected from $C[M_1, i]$, then O_{11} and O_{51} are shifted forward according to Step 6 in Algorithm 2.2.1.

Algorithm 2.2.1 (GT algorithm)

A scheduling problem represented by $\{T_{jk}\}$, the technological sequence matrix, and $\{p_{jk}\}$, the processing time matrix is given as an input.

1. Initialize G as a set of operations that are first in the technological sequence; i.e., $G = \{O_{1T_{11}}, O_{2T_{21}}, \ldots, O_{2T_{n1}}\}$. For each operation $O \in G$, set $ES(O) := 0$ and $EC(O) := p(O)$.

2. Find the earliest completable operation $O_{*r} \in G$ by (2.1) with machine M_r. A subset of G that consists of operations processed on machine M_r is denoted as G_r.

3. Calculate the conflict set $C[M_r, i] \subset G_r$ by (2.2), where $i-1$ is the number of operations already scheduled on M_r.

4. Select one of operations in $C[M_r, i]$ randomly. Let the selected operation be O_{kr}.

5. Schedule O_{kr} as the i-th operation on M_r; i.e. $S_{ri} := k$, with its starting and completion times equal to $ES(O_{kr})$ and $EC(O_{kr})$ respectively: $s(O_{kr}) = ES(O_{kr}), c(O_{kr}) = E(CO_{kr})$.

6. For all $O_{jr} \in G_r \setminus \{O_{kr}\}$, Update $ES(O_{jr})$ as

   $ES(O_{jr}) := \max\{ES(O_{jr}), EC(O_{kr})\}$ and $EC(O_{jr})$ as $EC(O_{kr}) := ES(O_{kr}) + p(O_{kr})$.

7. Remove O_{kr} from G (and therefore from G_r), and add an operation O_{ks} that is the next to O_{kr} in the technological sequence to G if such O_{ks} exits; i.e., if $r = T_{ki}$ and $i < m$, then $s := T_{ki+1}$ and $G := (G \setminus \{O_{kr}\}) \cup \{O_{ks}\}$.

   Calculate $ES(O_{ks})$ and $EC(O_{ks})$ as:

   $ES(O_{ks}) := \max\{EC(O_{kr}), EC(PM(O_{ks}))\}$ and $EC(O_{ks}) := ES(O_{ks}) + p(O_{ks})$ respectively.

8. Repeat from Step 1 to Step 7 until all operations are scheduled.

9. Output the solution matrix $\{S_{rk}\}$ as the active schedule obtained with the set of starting and completion times $\{s(O_{jr})\}$ and $\{c(O_{jr})\}$ respectively where $j = S_{rk}$.

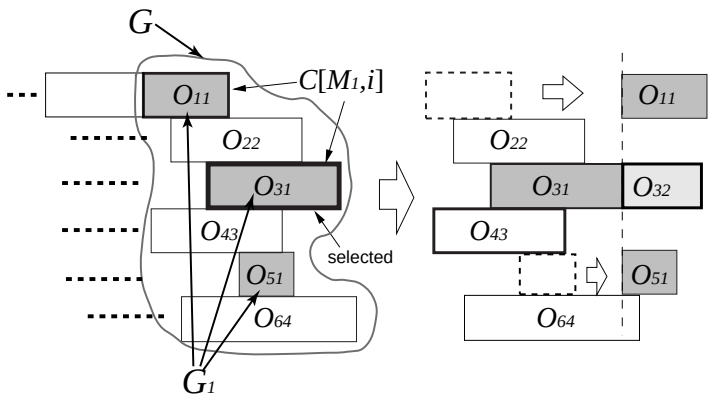

Figure 2.5: A snapshot in the middle of scheduling using Giffler and Thompson's algorithm, where $O_{11}, O_{22}, O_{31}, O_{43}, O_{51}$ and O_{64} are schedulable. O_{11} is the earliest completable operation, and O_{11} and O_{31} are in conflict with O_{11}. O_{31} is selected for the next operation to be scheduled and then O_{11} and O_{51} must be shifted forward to avoid overlapping. O_{31}, which is the next operation to O_{31} in the technological sequence, now becomes schedulable.

2.3 Disjunctive Graph Representation

The Gantt chart representation and the matrix representation described in the previous section are simple and straightforward to identify a schedule. However it is not obvious to see whether the resulting schedule is feasible or not: i.e., whether the job sequence on each machine does not contradict with the pre-defined technological sequence of machines. A more informative problem formulation based on a graph representation is first introduced by Roy and Sussman [29]. In this section, we review a graph representation for the JSP using disjunctive graph formulation. The following descriptions and notations are due to Adams et. al. [14].

The JSP can be 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 machines for each job.
- $D = \bigcup_{r=1}^{m} D_r$, where D_r is a set of disjunctive arcs representing pairs of operations that must be performed on the same machine M_r.

The processing time for each operation is the weighted value p_v attached to the corresponding node v, and for the special nodes, $p_0 = p_* = 0$. Figure 2.6 shows a disjunctive graph representation of the problem given in Table 2.1.

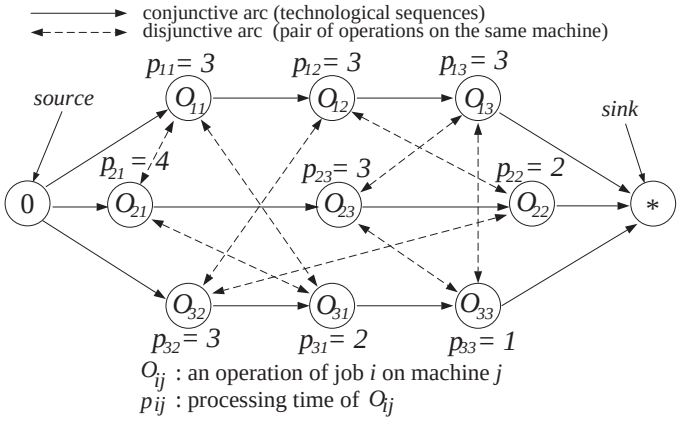

Figure 2.6: A disjunctive graph representation of a 3×3 problem

Let s_v be the starting time of an operation corresponding to node v. By using the disjunctive graph notation, the jobshop scheduling problem can be formulated as a mathematical programming model as follows:

$$\begin{aligned} &\text{minimize: } s_* \\ &\text{subject to: } \quad s_w - s_v \geq p_v, && (v,w) \in C, \\ &\qquad\qquad\quad\ s_v \geq 0, && v \in V, \\ &\qquad\qquad\quad\ s_w - s_v \geq p_v \vee s_v - s_w \geq p_w, && (v,w) \in D_r, 1 \leq r \leq m. \end{aligned} \tag{2.3}$$

The formula $A \vee B$ means that either A or B is to be satisfied (but not both), thus it is called disjunctive constraint. Note that $*$ is the dummy sink node that has a zero processing time. This means that s_* is equal to the completion time of the very last operation of the schedule, which is therefore equal to C_{max}. The first inequality in (2.3) ensures that when there is a conjunctive arc from a node v to a node w, w must wait at least p_v time period after v is started, thus the predefined technological sequence of machines for each job is not violated. The second condition is equivalent to $s_0 \geq 0$. According to the third constraints, when there is a disjunctive arc between a node v and a node w, one has to select either v to be processed prior to w (and w waits for at least p_v time period) or the other way around. This avoids any pair of operations on the same machine to overlap in time. In the disjunctive graph, the selection corresponds to fixing the undirected (disjunctive) arc into a directed one.

To summarize, jobshop scheduling is to define the ordering between all operations that must be processed on the same machine, as described in the previous section. This corresponds to the third constraints in (2.3), and this is done by fixing all undirected (disjunctive) arcs into directed ones: thus the disjunctive graph is turned into a conjunctive graph. A *selection* is defined as a set of directed arcs selected from the set of disjunctive arcs D. By definition, a selection is *complete* if all the disjunctions in D are selected. It is *consistent* if the resulting directed graph is acyclic. When a complete selection is consistent, one can define a unique and consistent ordering of operations on the same machine, namely a solution matrix $\{S_{rk}\}$ and this matrix corresponds to a feasible (semi-active) schedule. Hence a consistent complete selection, the obtained conjunctive graph, and the corresponding (semi-active) schedule are all identified and represented by the same symbol S without confusion. Given a selection, a path starting from a node v to any destination node w is defined by following directed arcs from v to w (if they exist), and the length of the path is defined as the sum of the weights of all the nodes on the path including v and w. It is clear that the makespan C_{max} of a schedule S is given by the length of the longest weighted path from source to sink in the graph of the corresponding complete selection. This path $\mathcal{P}$ (not necessarily unique) is called a *critical path* and is composed of a sequence of *critical operations*.

By using the disjunctive graph model, we can easily show a well-known fact known as feasibility property for adjacent exchanges of operations on a critical path: any exchange of two adjacent operations on a critical path will never lead to an infeasible schedule. Based on this fact, Laarhoven et al. have proposed a simulated annealing algorithm using the pairwise exchange of operations on the critical path as a transition operator [12]. Taillard has proposed a Tabu Search method by using the same transition operator [17].

Theorem 1 (feasibility for adjacent exchange) *Let S be a consistent complete selection and $\mathcal{P}(S)$ be a critical path in S. Consider a pair of adjacent critical operations (u, v) on a same machine on $\mathcal{P}(S)$, i.e.,there is an arc selected from u to v. Then a complete selection S^{uv} obtained from S by reversing the direction of the arc between u and v is always acyclic (thus the corresponding schedule is always feasible).*

Proof: Assume the contrary, then the exchange introduces a cycle in S^{uv}. This means that there is a path $P^{u,v}$ from u to v in S^{uv}, and this $P^{u,v}$ also exists in S. $\mathcal{P}(S)$ can be represented as $\mathcal{P}(S) = (0, \ldots, t, u, v, w, \ldots, *)$ and $(0, \ldots, t, P^{u,v}, w, \ldots, *)$ is also a path from source to sink in

S but clearly longer than $\mathcal{P}(S)$. This contradicts the assumption of the theorem that $\mathcal{P}(S)$ is a critical path of S. □

2.4 DG Distance and Binary Representation

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 [24]. In other words, it can be calculated by counting the directed (originally disjunctive) arcs whose directions are different between S and T. We call this distance the *disjunctive graph* (DG) *distance*. Figure 2.7 shows the DG distance between two schedules. The two directed arcs marked by thick lines in schedule T have directions that differ from those of schedule S, and therefore the DG distance between S and T is 2.

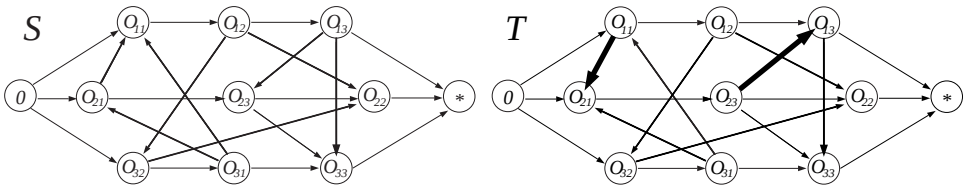

Figure 2.7: The DG distance between two schedules: the distance = 2.

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 (originally) disjunctive arc of a schedule as 0 or 1 according to its direction, and rearrange them as a one dimensional vector, a schedule can be represented by a binary string of length $mn(n-1)/2$. Figure 2.8 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. Nakano and Yamada have proposed a simple Genetic Algorithm based on this binary coding and using standard genetic operators [24].

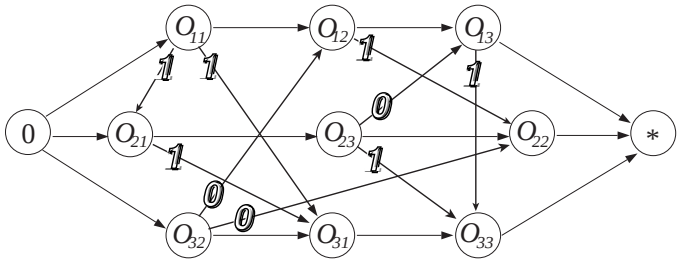

Figure 2.8: Labeling disjunctive arcs

By using the notion of DG distance, the following so called *connectivity property* for adjacent exchanges on the critical path can be derived easily from Theorem 1 as follows:

Theorem 2 (connectivity for adjacent exchange) *Let S and $\mathcal{P}(S)$ be an arbitrarily schedule and its critical path, then it is possible to construct a finite sequence of adjacent exchange on the critical path that will lead to an optimal schedule.*

Proof: Let S^* be an optimal schedule. Because S is not optimal, $C_{max}(S) > C_{max}(S^*)$, and the DG distance, denoted by d, between S and S^* is $d(S, S^*) > 0$. Moreover, there is at least one pair of consecutive critical operations (u, v) in S such that (u, v) is processed in S in this order but in S^*, v is processed prior to u. This is true because if such pair does not exist, then the critical path $\mathcal{P}(S)$ in S exists also in S^* as a path and this means that $C_{max}(S) < C_{max}(S^*)$ which contradicts the assumption that S^* is optimal. Reverse the direction of (u, v) and obtain S^{uv}. Theorem 1 guarantees that S^{uv} is feasible. It is clear that $d(S^{uv}, S^*) = d(S, S^*) - 1$, i.e., S^{uv} is closer to S^* than S by one step. Replace S by S^{uv} and repeat this process at most $d(S, S^*)$ times until there is no such (u, v), then S becomes identical with S^*, or at least the critical paths of S and S^* become identical, therefore $C_{max}(S) = C_{max}(S^*)$ and so S is optimal. □

2.5 Block Property

As described in the previous section, an operation on a critical path is called a critical operation. A sequence of more than one consecutive critical operations on the same machine is called a *critical block*. More formally, let S be a feasible schedule associated with a disjunctive graph $G(S) = G(V, C \cup D)$ with all the disjunctive arcs in D being directed. Let $\mathcal{P}(S)$ be a critical path in $G(S)$ and $L(S)$ be the length of this critical path, which is equal to the makespan. A sequence of successive nodes in P is called a *critical block* or just *block* if the following two properties are satisfied:

1. The sequence contains at least two nodes,

2. The sequence represents a maximal number of operations to be processed on the same machine.

The j-th block on the critical path is denoted by B_j. Figure 2.9 shows an example of critical blocks on a critical path. The following so-called *block property* gives us crucial information in improving a current schedule by simple modifications and thus forms a basis for many of the jobshop scheduling solvers [30].

Theorem 3 (Block property) *Let S, $\mathcal{P}(S)$, $L(S)$ be a complete selection, its critical path, and the length of the critical path respectively. If there exists another complete selection S' such that $L(S') < L(S)$, then at least one operation of some block B of $G(S)$ has to be processed in S' before the first or after the last operation of B.*

Proof: Assume the contrary that $L(S') < L(S)$ and that there is no operation that satisfies the conclusion of the theorem; i.e., *there is no operation of any block of S that is processed before*

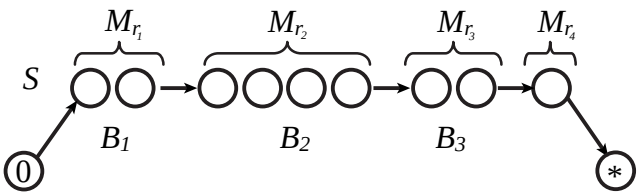

Figure 2.9: An example in which there are three critical blocks is illustrated. The blocks B_1, B_2 and B_3 are on the critical path $\mathcal{P}(S)$ and are corresponding to the different machines M_{r_1}, M_{r_2} and M_{r_3} respectively. The adjacent critical blocks are connected by a conjunctive arc. The last operation on machine M_{r_3} is not a critical block, which must contain at least two operations.

the first or after the last operation of the corresponding block. Then, there is a path $P(S')$ from source to sink in S' that contains all the operations on $\mathcal{P}(S)$:

$$\{P(S')\} \supset \{\mathcal{P}(S)\} \tag{2.4}$$

where $\{P(S)\}$ denotes a set of all nodes on $P(S)$ (See Figure 2.10). Let $l(P)$ be the length of P, then from (2.4) we have:

$$l(P(S')) \geq L(S) \tag{2.5}$$

The definition of the critical path indicates:

$$L(S') \geq l(P(S')) \tag{2.6}$$

From (2.5) and (2.6), we have:

$$L(S') \geq L(S) \tag{2.7}$$

This contradicts the assumption of the theorem. □

The theorem gives us an important criterion about how to improve a current schedule. Namely, if we wish to improve a schedule S, then either one of the following two situations must happen:

1. At least one operation in one block B, that is not the first one in B, has to be processed *before* all the other operations in B.

2. At least one operation in one block B, that is not the last one in B, has to be processed *after* all the other operations in B.

Let S be a complete selection, A critical path $\mathcal{P}(S)$ in $G(S)$ defines critical blocks $B_1, \ldots, B_k$. Roughly speaking, the *before-candidates* B_j^B is defined as a set of all but the first operations in B_j and *after-candidates* B_j^A a set of all but the last operations in B_j. More precisely, the first and the

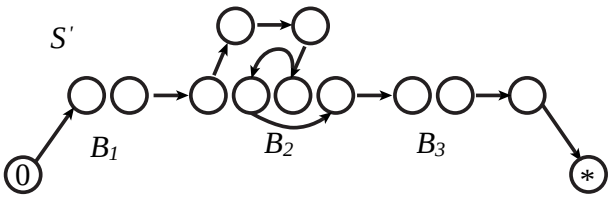

Figure 2.10: The condition that no operation in a block in S is processed before the first or after the last operation of the block in S' implies that all the operations in B_i are processed prior to those in B_j both in S and S' if $i < j$, because each adjacent blocks are connected by a conjunctive arc that cannot be altered. Hence, there is a path $P(S')$ in S' that contains all the operations on $\mathcal{P}(S)$ and the length of $P(S')$ is greater than the length of $\mathcal{P}(S)$.

last blocks B_1 and B_k need special treatment. If the first operation u_1^1 of the first block B_1 is also the very first operation of the critical path $\mathcal{P}(S)$, then we set B_1^B as empty. Likewise, if the last operation $u_{m_k}^k$ of the last block B_k is also the very last operation of the critical path $\mathcal{P}(S)$, then we set B_k^A as empty.

$$B_j^B = \begin{cases} \emptyset & \text{if } j = 1 \text{ and } u_1^1 \text{ is the first in } \mathcal{P}(S) \\ B_j \setminus \{u_1^j\} & \text{otherwise.} \end{cases} \tag{2.8}$$

$$B_j^A = \begin{cases} \emptyset & \text{if } j = k \text{ and } u_{m_k}^k \text{ is the last in } \mathcal{P}(S) \\ B_j \setminus \{u_{m_j}^j\} & \text{otherwise.} \end{cases} \tag{2.9}$$

where u_1^j and $u_{m_j}^j$ are the first and the last operations in B_j. Note that B_j contains at least two operations, and so $B_j^B \cup B_j^A$ is always non-empty.

Brucker et al. have proposed an efficient branch and bound method based on the block property [9]. It is natural to consider generating a new schedule by moving an operation in B_j^B (or B_j^A) to the front (or rear) of B_j aiming for possible improvements. The adjacent exchange of critical operations discussed in the end of Section 2.3 is a special case of this kind of transition. Many metaheuristics approaches have been proposed based on this transition operators [18, 19]. Note that unlike the simpler adjacent exchange of critical operations, in which the feasibility is guaranteed by Theorem 1, applying this transition may result in an infeasible schedule.

2.6 The Shifting Bottleneck Heuristic

The shifting bottleneck heuristic (SB) proposed by Adams, Balas and Zawack [14] is one of the most powerful heuristic methods for the jobshop scheduling problem. Assume we have a partial schedule or corresponding selection S_p, in which only some of the machines are already scheduled and the ordering of operations on those machines has been determined and fully fixed.

For this partial schedule, a critical path is defined exactly the same way as in the complete case; the longest path from source to sink in the graph of the corresponding selection S_p. Hence the makespan $L(S_p)$ is also defined as the length of the critical path.

Given a partial schedule S_p, then we focus on a machine M_k not yet scheduled. When we schedule operations on machine M_k, we have to take into accounts the constraints imposed by the already scheduled operations on other machines. For example, assume that we have a problem given in Figure 2.6 as a disjunctive graph, and that operations on machine M_1 and M_3 are scheduled and their starting and completion times are fixed. Then, operation O_{12} for job J_1 on machine M_2, which is not yet scheduled, has to be started at least after the completion of O_{11} and ended before the start of O_{13}. In short we have to start and complete the processing of O_{12} within a given limited time interval. Similar constraints are imposed to O_{22} and O_{32} as well, and we have to determine the order of O_{12}, O_{22} and O_{32} on machine M_2 such that these constraints are not violated.

In general, we define *head* and *tail* for each unscheduled operation. Let i be an operation on machine M_k not yet scheduled. Operation i can be identified as a node i in the graph corresponding to selection S_p. Let r_i be the length of the longest path from source to the node i: i.e., $r_i = L(0, i)$, where $L(i, j)$ is the length of the longest path from node i to node j. r_i is called the *release time* or the *head* of operation i. In a similar fashion, let q_i be the length of the longest path from i to the sink minus processing time of i, i.e., $q_i = L(i, *) - p_i$, where p_i is the processing time of i. q_i is called the *tail* of operation i. The *due date* d_i is defined as $d_i = L(0, *) - q_i$. Note that $L(0, *)$ is the makespan of S_p.

When the head r_i, tail q_i and the processing time p_i, summarized as $\{r_i, p_i, q_i\}$ are given for each operation i on a machine M_k, and let C as a set of operations on machine M_k, we have a one machine scheduling problem formulated as a mathematical programming model as follows:

$$\begin{array}{lll} \text{minimize the makespan:} & \max_{i \in C}\{s_i + p_i + q_i\} & \\ \text{subject to:} & s_i \geq r_i & (i \in C) \\ \text{and the disjunctive constraints:} & s_j - s_i \geq p_i \vee s_i - s_j \geq p_j & (i, j \in C) \end{array} \tag{2.10}$$

Starting from a partial schedule S_p, which is initially set as empty, we solve a one-machine scheduling problem for each machine not yet scheduled to optimality, and find a bottleneck machine: a machine with the longest one-machine makespan. The algorithm to solve the one-machine scheduling problem will be described in the next section. The bottleneck machine is regarded as scheduled and S_p is updated using the job sequence on the bottleneck machine obtained above. Every time a new machine has been scheduled, the job sequence on each previously scheduled machine is subject to reoptimization. The original SB consists of two subroutines: the first one (SBI) repeatedly solves one-machine scheduling problems; the second one (SBII) builds a partial enumeration tree where each path from the root to a leaf is similar to an application of SBI.

The rough outline of SBI can be summarized as follows:

1. For each of unscheduled machines, solve the one-machine scheduling problem and obtain the best makespan and corresponding job sequence.

2. Identify the most bottleneck machine which has the longest one-machine makespan obtained above.
3. Make the most bottleneck machine scheduled using the job sequence obtained above.
4. Reoptimize all the scheduled machines.
5. Repeat the above until all the machines are scheduled.

The more complete SBI algorithm is given in Algorithm 2.6.1.

Instead of considering the most bottleneck machine in Step 3 of Algorithm 2.6.1, if we consider the n-*th* highest bottleneck machines and apply the remaining steps of SBI for each bottleneck candidate, we have SBII.

2.7 The One-machine Scheduling Problem

In the SBI heuristic, we have to repeatedly solve the one-machine scheduling problem. Although the problem is $\mathcal{NP}$-hard, Carlier has developed an efficient branch and bound method [31]. In this section, we focus on the one-machine problem and describe the algorithm proposed by Carlier. In the one-machine case, each job has only one operation, so an operation and corresponding job is identified. Furthermore, a simplified notation is used to identify job number and corresponding job; i.e., we just say "job i" instead of saying "operation O_i of job J_i ". The disjunctive graph of the problem is defined just as a special case of the jobshop scheduling problem. A schedule is obtained by determining the starting (or completion) times of all jobs, or equivalently, turning all undirected disjunctive arcs into directed ones, resulting in a conjunctive graph, or simply, a job sequence.

Consider a one-machine scheduling problem P with n jobs $I = \{1, 2, \ldots, n\}$ characterized by $\{r_i, p_i, q_i\}$, where r_i, p_i and q_i are the head, processing time and tail of each job i respectively. Hereafter, we omit a set of n jobs I when it is obvious and just say "P is defined by $\{r_i, p_i, q_i\}$". The formal definition of the one-machine scheduling problem was already given by (2.10) and we do not repeat it here. Table 2.2 shows an example of the one-machine scheduling problem with 7 jobs and Figure 2.11 shows an example of a schedule represented by a conjunctive graph for this problem. Note that the conjunctive graph is simplified such that only $n-1$ arcs are presented between adjacent job nodes instead of drawing $n(n-1)/2$ arcs between all job node pairs, which are apparently redundant. The number on each arc from source to a job node i is the head r_i and the numbers on each arc from a job node i to sink is the processing time p_i "+" the tail q_i. The schedule presented in the figure is called a Schrage schedule, the definition of which will be presented shortly.

A lower bound of the makespan for a one-machine scheduling problem defined by $I = \{J_1, \ldots, J_n\}$ and $\{r_i, p_i, q_i\}$ is calculated as follows:

Theorem 4 *Let I_1 be a subset of I, then*

$$h(I_1) = \mathrm{Min}_{i \in I_1} r_i + \sum_{i \in I_1} p_i + \mathrm{Min}_{i \in I_1} q_i \tag{2.11}$$

Algorithm 2.6.1 (SBI heuristic)

A scheduling problem given as a set of technological sequences of machines for each job with processing times (preferably represented by a disjunctive graph) are given as inputs. Let $\mathcal{M}$ be the set of all machines: $\mathcal{M} = \{M_1, \ldots, M_r\}$.

1. A partial schedule S_p and a set of already scheduled machines $\mathcal{M}_0$ in S_p are initialized as $S_p := \{\}$ and $\mathcal{M}_0 := \{\}$ respectively.
2. For each $M_k \in \mathcal{M} \setminus \mathcal{M}_0$, do the following
   (a) Compute head r_i and tail q_i for each operation i on M_k given S_p. Let p_i be the processing time of i.
   (b) Solve one-machine scheduling problem $\{r_i, q_i, p_i\}$ to optimality for machine M_k and obtain the best one-machine makespan $v(M_k)$ and corresponding job sequence (more precisely, corresponding selection) S_{M_k}.
3. Let M^* be the bottleneck machine such that $v(M^*) \geq v(M)$ for all $M \in \mathcal{M} \setminus \mathcal{M}_0$
4. Set $\mathcal{M}_0 := \mathcal{M}_0 \cup \{M^*\}$ and $S_p := S_p \cup S_{M^*}$.
5. Order the elements of $\mathcal{M}_0$ as $\{M_1, M_2, \ldots, M_l\}$ ($l = |\mathcal{M}_0|$) in the order of its inclusion to $\mathcal{M}_0$ above.
6. **[local optimization of S_p with already scheduled machines $\mathcal{M}_0 = \{M_1, M_2, \ldots, M_l\}$] do**
   (a) For $m = 1, 2, \ldots, l$, do the following:
      i. Reset the sequence of operations on M_m, i.e., let $S_p' = S_p \setminus S_{M_m}$.
      ii. Compute head r_i and tail q_i for each operation i on M_m of S_p'.
      iii. Solve one-machine scheduling problem $\{r_i, q_i, p_i\}$ and obtain new $v(M_m)$ and S_{M_m}.
      iv. Let $S_p'' = S_p' \cup S_{M_m}$ and compute makespan $L(S_p'')$ for partial schedule S_p''. If $L(S_p'') < L(S_p)$, then set $S_p := S_p''$, otherwise keep the original S_p.
   (b) Reorder $\{M_1, M_2, \ldots, M_l\}$ according to decreasing order of the new $v(M_m)$s.

   while any improvements are made in Step 6a.
7. Repeat from Step 2 to Step 5 until $\mathcal{M}_0 = \mathcal{M}$.
8. Output S_p as a obtained complete selection.

Table 2.2: An example of the one-machine scheduling problem with 7 jobs

i (= job no.)	1	2	3	4	5	6	7
r_i (head)	10	13	11	20	30	0	30
p_i (processing time)	5	6	7	4	3	6	2
q_i (tail)	7	26	24	21	8	17	0

is a lower bound on the optimal makespan.

Proof: In the conjunctive graph associated with the optimal schedule, there is a path P_{I_1} from source to sink, passing through every job in I_1. It is clear that the length of the path P_{I_1} is longer than or equal to $h(I_1)$. Whereas, by definition of the critical path, the length of the path P_{I_1} is shorter than or equal to the length of the critical path and which is equal to the makespan of the optimal schedule. Thus, $h(I_1)$ is a lower bound. □

There is a method to generate a reasonably good schedule called the *longest tail heuristic* in which a job with the longest tail q_i, therefore with the earliest due date, among ready jobs, is regarded as the most *urgent* and scheduled first. The resulting schedule is called a Schrage schedule. An algorithm to generate a Schrage schedule is described in Algorithm 2.7.1. The name "longest tail" heuristic stems from (2.13).

Figure 2.11 shows a Schrage schedule obtained by the longest tail heuristic to the problem given in Table 2.2. The starting times of each job is: $s_1 = 10$, $s_2 = 15$, $s_3 = 21$, $s_4 = 28$, $s_5 = 32$, $s_0 = s_6 = 0$, $s_7 = 35$, $s_\star = 53$. The critical path is 0, 1, 2, 3, 4, $\star$ and the makespan is equal to 53. This example is taken from [31].

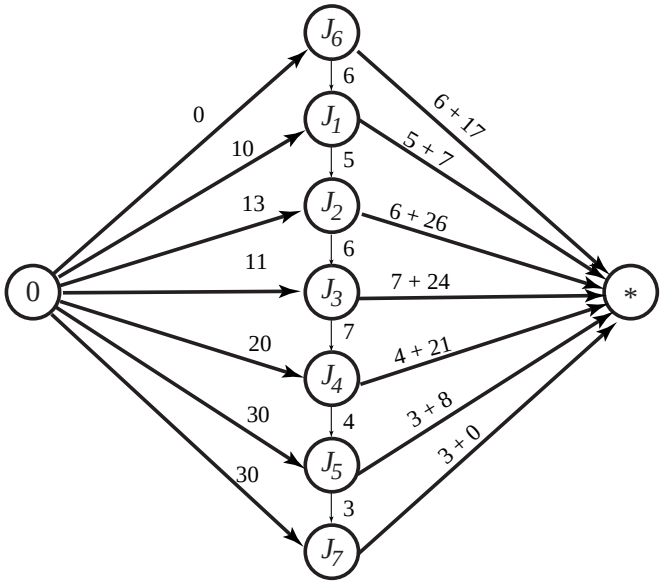

Figure 2.11: A Schrage schedule represented by a conjunctive graph

Algorithm 2.7.1 (longest tail heuristic)

A one-machine scheduling problem defined by $\{r_i, p_i, q_i\}$ for each job $i \in I = \{1, 2, \ldots, n\}$ is given as input

1. Initialize a set of scheduled jobs $U := \{\}$, and the most recently scheduled job $k := 0$. The starting times and processing times of the two dummy nodes 0 and $\star$ are defined as $s_0 := s_\star := 0$ and $p_0 := p_\star := 0$ respectively.

2. Identify R, a set of ready jobs as $R := \{i \in I \setminus U \mid r_i \leq s_k + p_k\}$.

3. If R is empty, then k', the next job to be scheduled, is selected as

$$k' := \arg \mathrm{Min}_{i \in I \setminus U} r_i \tag{2.12}$$

   and schedule k' as $s_{k'} = r_{k'}$. Otherwise k' is selected as

$$k' := \arg \mathrm{Max}_{i \in R} q_i \tag{2.13}$$

   and schedule k' as $s_{k'} = s_k + p_k$. Update $k := k'$.

4. Update $U := U \cup \{k\}$ and update $s_\star$ as $s_\star := \max\{s_\star, s_k + p_k + q_k\}$.

5. Repeat Step 2 to Step 4 until $U = I$.

6. Output the set of obtained starting times $\{s_0, s_1, \ldots, s_n, s_\star\}$.

The following theorem implies that the Schrage schedule is in a sense "close" to the optimal schedule, and to improve it, a particular job c has to be rescheduled.

Theorem 5 *Let L be the makespan of the Schrage schedule.*
(a) If this schedule is not optimal, there is a critical job c and a critical set $\mathcal{J}$ such that:

$$h(\mathcal{J}) = \mathrm{Min}_{i\in\mathcal{J}} r_i + \sum_{i\in\mathcal{J}} p_i + \mathrm{Min}_{i\in\mathcal{J}} q_i > L - r_c$$

(b) If this schedule is optimal, there exists $\mathcal{J}$ such that $h(\mathcal{J}) = L$.

Proof: Let G be the directed graph associated with the Schrage schedule and consider a critical path of G that contains maximal number of jobs. By renumbering jobs if necessary, the critical path P is denoted as $P = (0, 1, 2, \ldots, l, \star)$ without loss of generality.
The length of the critical path L is:

$$L = s_1 + \sum_{i=1,\ldots,l} p_i + q_l. \tag{2.14}$$

If job 1 was scheduled immediately when its predecessor k, if exists, was just finished; i.e., if $s_k + p_k$ equals to s_1, then k must be on the critical path, too. This is in contradiction with the maximality of P. Hence $s_k + p_k < s_1$, and this means R was empty and 1 is selected by (2.12) in Algorithm 2.7.1. Therefore,

$$s_1 = r_1 = \mathrm{Min}_{i=1,\ldots,l} r_i. \tag{2.15}$$

If $q_l = \mathrm{Min}_{i=1,\ldots,l} q_i$, then $L = h(\mathcal{J})$ with $\mathcal{J} = \{1, \ldots, l\}$ from (2.14) and (2.15). Because $h(\mathcal{J})$ is a lower bound from Theorem 4, the schedule is optimal.
Otherwise, there exists $i \in \{1, \ldots, l\}$ such that $q_i < q_l$. Let c be the greatest number among such i and let $\mathcal{J} = \{c + 1, \ldots, l\}$, then:

$$q_c < q_k \text{ for all } k \in \mathcal{J} \tag{2.16}$$

and

$$q_l = \mathrm{Min}_{k\in\mathcal{J}} q_k. \tag{2.17}$$

From (2.16), c has the "shortest" tail among $\{c\} \cup \mathcal{J}$: the least urgent job in the "longest" tail heuristic. This means that when c was scheduled, c was not selected by (2.13) but selected by (2.12), meaning:

$$s_c = r_c < r_k \text{ for all } k \in \mathcal{J}. \tag{2.18}$$

Because c is on the critical path and $s_1 = r_1$ from (2.15),

$$s_c = s_1 + p_1 + \ldots p_{c-1} = r_1 + p_1 + \ldots p_{c-1}. \tag{2.19}$$

From (2.18) and (2.19), we have

$$r_1 + p_1 + \ldots p_{c-1} < r_k \text{ for all } k \in \mathcal{J}. \tag{2.20}$$

Together with (2.17) and (2.20), we have:

$$h(\mathcal{J}) = \text{Min}_{k\in\mathcal{J}} r_k + \sum_{k\in\mathcal{J}} p_k + \text{Min}_{k\in\mathcal{J}} q_k > r_1 + p_1 + \ldots p_{c-1} + p_c + \sum_{k\in\mathcal{J}} p_k + q_l - p_c. \tag{2.21}$$

The right hand side of (2.21) is equal to $L - p_c$ □

As a corollary of this theorem, it can be seen that in an optimal schedule, either c is processed before all the jobs in $\mathcal{J}$ or after all the jobs in $\mathcal{J}$. By using this fact, we consider two new one-machine scheduling problems S_L and S_R by modifying head or tail of the original problem S. Let P be a one-machine scheduling problem corresponding to $\{r_i, p_i, q_i\}$. Apply the longest tail heuristic to P and obtain a Schrage schedule S and the critical job c and critical set $\mathcal{J}$. P_L requires job c scheduled before all jobs in $\mathcal{J}$. Thus P_L is obtained from P using the same $\{r_i, p_i, q_i\}$ but only modifying q_c as follows:

$$q_c := \text{Max}(q_c, \sum_{k\in\mathcal{J}} p_r + q_l). \tag{2.22}$$

Likewise P_R requires job c scheduled after all jobs in $\mathcal{J}$. Therefore, P_R is obtained from P by modifying r_c as follows:

$$r_c := \text{Max}(r_c, \text{Min}_{k\in\mathcal{J}} r_r + \sum_{k\in\mathcal{J}} p_r). \tag{2.23}$$

Now we are ready to describe the branch and bound algorithm. In the branch and bound algorithm, we consider a search tree in which each node is associated with a one-machine scheduling problem P defined by $\{r_i, p_i, q_i\}$. The root node P_0 corresponds to the original problem to be solved. The active node P_a is initialized as P_0. The upper bound μ is initialized as $\mu := \infty$.

We apply the longest tail heuristic to the active node P_a and obtain the Schrage schedule and its makespan $L_s(P_a)$, the critical operation c and the critical set $\mathcal{J}$. The upper bound μ is updated as $\mu := \min\{\mu, L_s(P_a)\}$. Then P_L and P_R are generated by using (2.22) and (2.23).The lower bound λ of the two new nodes will be $\lambda(P_L) := \max\{L_s(P_a), h_L(\mathcal{J}), h_L(\mathcal{J} \cup \{c\})\}$ and $\lambda(P_R) := \max\{L_s(P_a), h_R(\mathcal{J}), h_R(\mathcal{J} \cup \{c\})\}$ respectively, where h_L and h_R correspond to h in (2.11) but calculated with modified $\{r_i, p_i, q_i\}$ by (2.22) and (2.23) respectively. A new node will be added to the tree only if its lower bound is less than the upper bound μ. The next active node is identified as a node with the lowest bound.

2.8 The Well-known Benchmark Problems

The three well-known benchmark problems with sizes of 6×6, 10×10 and 20×5 (known as mt06, mt10 and mt20) formulated by Muth and Thompson [4] are commonly used as test beds to

Algorithm 2.7.2 (Carlier's branch and bound algorithm)

A one-machine scheduling problem P_0 defined by $\{r_i, p_i, q_i\}$ is given as input

1. Initialize $P_a = P_0$ and $\mu := \infty$.
2. Apply the longest tail heuristic to P_a and obtain the Schrage schedule and its makespan $L_s(P_a)$, the critical operation c and the critical set $\mathcal{J}$.
3. If $L_s(P_a) < \mu$, then store P_a as the best schedule obtained so far:$P_b := P_a$, and update $\mu := L_s(P_a)$. Make P_a visited.
4. Generate P_L and P_R defined by $\{r_i^L, p_i^L, q_i^L\}$ and $\{r_i^R, p_i^R, q_i^R\}$ respectively using (2.22) and (2.23).
5. Calculate $\lambda(P_L)$, the lower bound for P_L, as $\lambda(P_L) := \max\{L_s(P_a), h_L(\mathcal{J}), h_L(\mathcal{J} \cup \{c\})\}$, where h_L is calculated by (2.11) with $h_L = h$ and $\{r_i, p_i, q_i\} = \{r_i^L, p_i^L, q_i^L\}$. Calculate $\lambda(P_R)$ in the same way as $\lambda(P_L)$.
6. Add the new node P_L to the search tree as a child node of P_a if $\lambda(P_L) < \mu$ and add P_R if $\lambda(P_R) < \mu$.
7. Update P_a as a node with the lowest bound among nodes not yet visited.
8. Repeat Step 2 to Step 7 until there is no node that is not yet visited.
9. Output P_b as the optimal schedule for P_0.

measure the effectiveness of a certain method. Figure 2.3 shows the mt06 problem, which is the easiest in size and structure. Indeed, it employs only 6 jobs and 6 machines, and the technological sequence of jobs on each machine is similar to each other.

Table 2.3: Muth and Thompson's 6×6 problem (mt06)

job	machine (processing time)					
1	3 (1)	1 (3)	2 (6)	4 (7)	6 (3)	5 (6)
2	2 (8)	3 (5)	5 (10)	6 (10)	1 (10)	4 (4)
3	3 (5)	4 (4)	6 (8)	1 (9)	2 (1)	5 (7)
4	2 (5)	1 (5)	3 (5)	4 (3)	5 (8)	6 (9)
5	3 (9)	2 (3)	5 (5)	6 (4)	1 (3)	4 (1)
6	2 (3)	4 (3)	6 (9)	1 (10)	5 (4)	3 (1)

Table 2.4: Muth and Thompson's 10×10 problem (mt10)

job	machine (processing time)									
1	1 (29)	2 (78)	3 (9)	4 (36)	5 (49)	6 (11)	7 (62)	8 (56)	9 (44)	10 (21)
2	1 (43)	3 (90)	5 (75)	10 (11)	4 (69)	2 (28)	7 (46)	6 (46)	8 (72)	9 (30)
3	2 (91)	1 (85)	4 (39)	3 (74)	9 (90)	6 (10)	8 (12)	7 (89)	10 (45)	5 (33)
4	2 (81)	3 (95)	1 (71)	5 (99)	7 (9)	9 (52)	8 (85)	4 (98)	10 (22)	6 (43)
5	3 (14)	1 (6)	2 (22)	6 (61)	4 (26)	5 (69)	9 (21)	8 (49)	10 (72)	7 (53)
6	3 (84)	2 (2)	6 (52)	4 (95)	9 (48)	10 (72)	1 (47)	7 (65)	5 (6)	8 (25)
7	2 (46)	1 (37)	4 (61)	3 (13)	7 (32)	6 (21)	10 (32)	9 (89)	8 (30)	5 (55)
8	3 (31)	1 (86)	2 (46)	6 (74)	5 (32)	7 (88)	9 (19)	10 (48)	8 (36)	4 (79)
9	1 (76)	2 (69)	4 (76)	6 (51)	3 (85)	10 (11)	7 (40)	8 (89)	5 (26)	9 (74)
10	2 (85)	1 (13)	3 (61)	7 (7)	9 (64)	10 (76)	6 (47)	4 (52)	5 (90)	8 (45)

The mt10 and mt20 problems are like brothers. They are processing the same set of operations and technological sequences are similar, but in the mt20 problem, the number of machines available is reduced to half of that of the mt10 problem. For example, the first operation of each job in mt10 is exactly same as the first operation of each of the first 10 jobs in mt20 and the second operation of each job in mt10 is exactly same as the first operation of each of the second 10 jobs in mt20.

The mt10 and mt20 problems had been a good computational challenges for a long time. Indeed, the mt10 problem has been referred as "notorious", because it remained unsolved for over 20 years. The mt20 problem has also been considered as quite difficult. However they are no longer a computational challenge.

Applegate and Cook proposed a set of benchmark problems called the "ten tough problems" as a more difficult computational challenge than the mt10 problem, by collecting difficult problems from literature, some of which still remain unsolved [1]. The ten tough problems consist of

Table 2.5: Muth and Thompson's 20×5 problem

job	machine (processing time)				
1	1 (29)	2 (9)	3 (49)	4 (62)	5 (44)
2	1 (43)	2 (75)	4 (69)	3 (46)	5 (72)
3	2 (91)	1 (39)	3 (90)	5 (12)	4 (45)
4	2 (81)	1 (71)	5 (9)	3 (85)	4 (22)
5	3 (14)	2 (22)	1 (26)	4 (21)	5 (72)
6	3 (84)	2 (52)	5 (48)	1 (47)	4 (6)
7	2 (46)	1 (61)	3 (32)	4 (32)	5 (30)
8	3 (31)	2 (46)	1 (32)	4 (19)	5 (36)
9	1 (76)	4 (76)	3 (85)	2 (40)	5 (26)
10	2 (85)	3 (61)	1 (64)	4 (47)	5 (90)
11	2 (78)	4 (36)	1 (11)	5 (56)	3 (21)
12	3 (90)	1 (11)	2 (28)	4 (46)	5 (30)
13	1 (85)	3 (74)	2 (10)	4 (89)	5 (33)
14	3 (95)	1 (99)	2 (52)	4 (98)	5 (43)
15	1 (6)	2 (61)	5 (69)	3 (49)	4 (53)
16	2 (2)	1 (95)	4 (72)	5 (65)	3 (25)
17	1 (37)	3 (13)	2 (21)	4 (89)	5 (55)
18	1 (86)	2 (74)	5 (88)	3 (48)	4 (79)
19	2 (69)	3 (51)	1 (11)	4 (89)	5 (74)
20	1 (13)	2 (7)	3 (76)	4 (52)	5 (45)

abz7, abz8, abz9, and *la21, la24, la25, la27, la29, la38, la40*. The *abz* problems are proposed by Adams in [14]. *la* problems are parts of 40 problems *la01-la40* originally from [32]. Table 2.6, which is taken from [1], shows for each of the ten tough benchmark problems, the problem size, best solution reported in [1] and best lower bound. Those gaps between the best solution and the best lower bound suggest the difficulties of the problems and no gap means the problem is *solved*. The more recent status of the best solutions will be reported in the later sections.

Table 2.6: The ten tough benchmark problems (status reported by [1] in 1991)

Prob	size ($n \times m$)	Best Solution	Best Lower Bound
abz7	20×15	668	654
abz8	20×15	687	635
abz9	20×15	707	656
la21	15×10	1053	1040
la24	15×10	935	935
la25	20×10	977	977
la27	20×10	1269	1235
la29	20×10	1195	1120
la38	15×15	1209	1184
la40	15×15	1222	1222

Taillard proposed a set of 80 JSP and 120 FSP benchmark problems. They cover various range of sizes and difficulties. They are randomly generated by a simple algorithm. It has become more common to use the ten tough problems and/or Taillard benchmark than to use the mt10 and mt20 problems as benchmark problems.

Chapter 3

Genetic Algorithms

Genetic Algorithms (GAs) are search strategies designed after the mechanics of natural selection and natural genetics to optimize highly complex objective functions. GAs have been quite successfully applied to optimization problems including scheduling. In this chapter, the basic concepts of GAs are reviewed.

3.1 Basic Concepts

Genetic Algorithms use a vocabulary borrowed from natural genetics. We have a set of *individuals* called *population*. An individual has two representations called *phenotype* and *genotype*. The phenotype represents a potential solution to the problem to be optimized in a straightforward way used in the original formulation of the problem. The genotype, on the other hand, gives an *encoded* representation of a potential solution by the form of a *chromosome*. A chromosome is made of *genes* arranged in linear succession and every gene controls the inheritance of one or several characters or features. For example, a chromosome consists of a sequence of 0 or 1 (i.e. a bit string), and the value at a certain position corresponds to *on* (the value = 1) or *off* (the value = 0) of a certain feature. More complicated forms such as a sequence of symbols and a permutation of alphabets are chosen for chromosomes depending of the target problem.

Each individual has its *fitness*, which measures how suitable is the individual for the local environment. The Darwinian theory tells us that among individuals in a population, the one that is the most suitable for the local environment is most likely to survive to have greater numbers of offspring. This is called a rule of "survival of the fittest."

The objective function f of the target optimization problem plays the role of an environment, therefore, the fitness of an individual F measures how "good" is the corresponding potential solution in terms of the original optimization criteria. When the target optimization is the maximization of the objective function, then fitness may be identical to the objective function value:

$$F(x) = f(x) \tag{3.1}$$

where x is an individual in the current population P. When the target is the minimization, then the objective function must be converted so that an individual with a small objective function value

has a high fitness. The most obvious way to deal with it is to define the fitness as the maximum of objective function over the current population minus its own objective function value:

$$F(x) = \max_{y \in P}\{f(y)\} - f(x). \tag{3.2}$$

Another method is known as ranking. In the ranking method, each individual in the current population P is sorted in the descending order of its objective function value so that the worst individual is numbered as x_1 and the best as x_n, where n is the size of P. Then the fitness F of an individual x_i, the i-th worst individual, is defined as

$$F(x_i) = i. \tag{3.3}$$

3.2 A Simple Genetic Algorithm

Meanwhile, let us consider a simple case in which the genotype is a bit string of length n. A simple genetic algorithm is composed of the following three operators:

1. Crossover
2. Mutation
3. Reproduction

Crossover and Mutation are genetic recombination operators. Each individual in a population is coupled to pairs which is called parents at random. Each pair of individuals undergoes *crossover* described as follows.

Crossover operates on genotype (i.e. chromosomes) of two individuals called parents. It generates new (usually two) individuals called offspring whose genes are inherited from either parents. This can be done by splitting each of the two chromosomes into fragments and recombining them again to form new chromosomes.

Now we assume that the genotype is a bit string of length n. The 1-point crossover sets one crossover point on a string at random and takes a section before the point from one parent and takes another section after the point from the other parent and recombines two sections to form a new bit string. For example, consider A_1 and A_2 being bit strings of length $n = 5$ as parents as follows:

$$\begin{aligned} A_1 &= 0000 \mid 0 \\ A_2 &= 1111 \mid 1. \end{aligned} \tag{3.4}$$

The symbol | indicates a crossover point, and in this case it is set after the fourth bit. The resulting 1-point crossover yields two new individuals A_1' and A_2' as follows:

$$\begin{aligned} A_1' &= 0000 \mid 1 \\ A_2' &= 1111 \mid 0. \end{aligned} \tag{3.5}$$

The two-point crossover sets two crossover points at random, and takes a section between the points from one parent and other sections outside the points from the other parent and recombines them. In the following example, the two crossover points are set after the first and fourth bits respectively.

$$\begin{aligned} A_1 &= 0 \mid 000 \mid 0 \\ A_2 &= 1 \mid 111 \mid 1. \end{aligned} \tag{3.6}$$

The resulting two-point crossover yields the following two individuals:

$$\begin{aligned} A'_1 &= 0 \mid 111 \mid 0 \\ A'_2 &= 1 \mid 000 \mid 1. \end{aligned} \tag{3.7}$$

The uniform crossover is a generalization of the two above. A random mask bit vector of length n is given, and the positions where the mask bit is zero are taken from one parent and the other positions where the mask bit is one are taken from the other. In the following example, the mask bit vector M' is given as $M = 01010$.

$$\begin{aligned} M &= 01010 \\ A_1 &= 00000 \\ A_2 &= 11111. \end{aligned} \tag{3.8}$$

The resulting uniform crossover yields the following two individuals:

$$\begin{aligned} A'_1 &= 01010 \\ A'_2 &= 10101. \end{aligned} \tag{3.9}$$

Mutation operates on genotype of single individual. It corresponds to an error occurred when chromosome is copied and duplicated. When exact copies are always guaranteed, then the mutation rate is zero. However in real life, copy error can happen under some circumstances such as the presence of noise. Mutation changes values of certain genes with small probability. An example of a typical bit-flip mutation is shown in (3.2), where the third gene from the left in A is selected with a small probability and its bit is flipped resulting in A':

$$\begin{aligned} A &= 00000 \\ A' &= 00100. \end{aligned} \tag{3.10}$$

Reproduction is a process in which individuals in a population are copied according to their fitness values to form a new population. The individuals *evolve* through successive iterations of reproduction, called *generations*. Each individual makes number of its copies proportional to its fitness, therefore, an individual with higher fitness makes more copies than that with lower fitness.This is an artificial version of natural selection; a Darwinian survival of the fittest among string creatures.

A simple reproduction operator is called a *roulette wheel* selection where each individual in a population has a roulette wheel slot sized in proportion to its fitness. To reproduce, we

simply spin the weighted roulette wheel and obtain a reproduction candidate with probability proportional to its fitness. Each time we require another offspring, a simple spin of the weighted roulette wheel yields the reproduction candidate. An example of the roulette wheel selection is shown in Figure 3.1. In the upper left picture in the figure, there are total seven individuals, ID 1 to ID 7, with fitness value assigned. A roulette wheel is created and shown in the right picture. The number in each wheel slot corresponds to the individual ID. The first individual with ID 1, for example, has fitness value 9, which is the highest and therefore the largest slot is assigned. We spin the roulette wheel seven times to select seven individuals. Individual ID 1 and ID 2 are likely to be selected more than once but individual ID 4 and ID 6 are not likely to be selected resulting in the seven individuals with some duplicates shown in the lower picture.

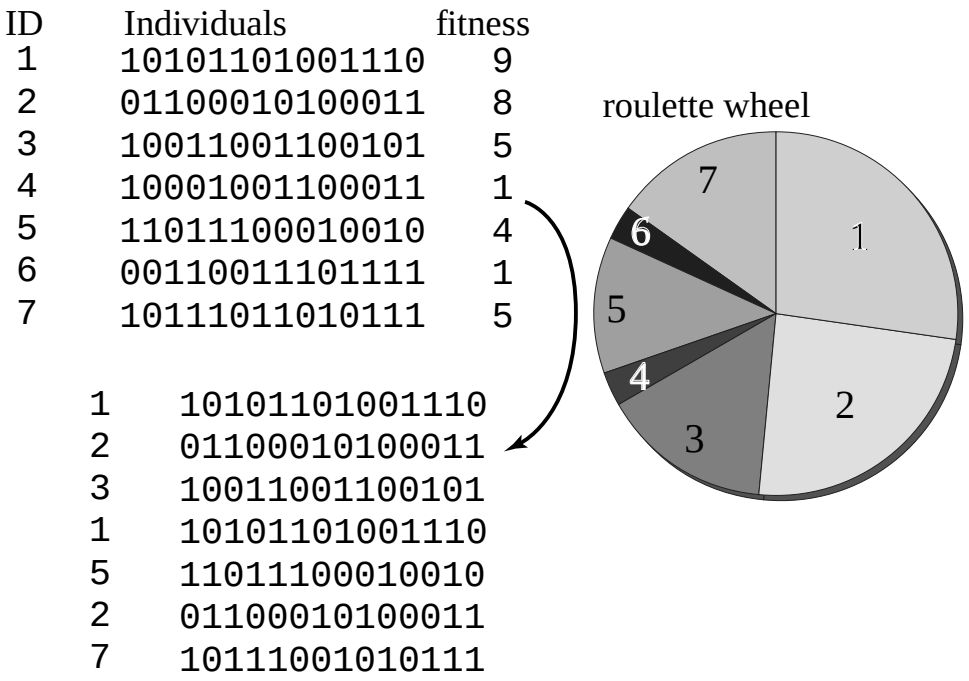

Figure 3.1: An example of the roulette wheel selection, where the roulette wheel is created according to the fitness value of each individual shown in the upper left picture

3.3 The Procedure of a Simple Genetic Algorithm

The general procedure of a simple GA can be summarized as in Algorithm 3.3.1. In the algorithm, we start from a random initial population $P(0)$. $P(t)$ is a population at generation t with N individuals. $R_c \times N$ members are randomly selected from $P(t)$ and crossover is applied to generate new $R_c \times N$ individuals that join into a new population $P'(t)$ in Step 2, where $R_c < 1$ is called

Algorithm 3.3.1 (Simple Genetic Algorithm)

1. Initialize $P(t)$ as a random population $P(t = 0)$
2. Recombine $P(t)$ to yield $P'(t)$ by crossover and mutation
3. Evaluate $P'(t)$
4. Reproduce $P(t + 1)$ from $P'(t)$ by selection
5. Set $t \leftarrow t + 1$
6. repeat from 2 to 5 until some termination condition is met.
7. Output the best individual in $P(t)$.

a crossover ratio. The rest of $P(t)$ is just copied to $P'(t)$. $R_m \times N$ members are then randomly selected from $P'(t)$ and mutation is applied to generate new individuals that replace the original, where $R_m < 1$ is call a mutation ratio. When the best individual in $P(t)$ is preserved and copied to $P'(t)$ without modification, it is called *elitist strategy*.

$P'(t)$ is evaluated in Step 3 and the new population $P(t + 1)$ is obtained after the reproduction using, for example, the roulette wheel selection in step4. The termination condition is usually given as: when t is sufficiently large, when the best or average fitness in $P(t)$ exceeds certain value, or when the variation of the fitness in $P(t)$ is small.

While the process described above is repeated for a sufficient number of generations, the recombination operators keep producing possibly new individuals with new fitness where some of them are possibly better than those of ever existing ones. The reproduction phase focuses on such good individuals and replicate them as occurred in the natural evolution. Eventually an individual with a high fitness value is expected to emerge in a population. The natural evolution process requires enormous amount of time. However, this *simulated* evolution process on a computer runs much faster.

Chapter 4

A Simple Genetic Algorithm for the Jobshop Scheduling Problem

In this chapter, the simple GA described in the previous chapter is applied to the jobshop scheduling problem. The approach described in this chapter was proposed by Nakano and Yamada [24]. An advantage of this approach is that conventional genetic operators, such as one-point, two-point and uniform crossovers can be applied without any modification. However, one drawback is that a new individual generated by crossover may not represent a feasible schedule. In other words, such genotype is called *fatal* or *illegal*. There are two approaches to solve this situation: one is to repair a fatal genotype to a normal one, and the other is to impose a penalty for the fatality and to lower the fitness. One example of the former approach will be elaborated in this chapter.

4.1 Genetic Encoding of a Solution Schedule

We have a $n \times m$ jobshop scheduling problem (JSP). As described in Chapter 2, a solution of a JSP can be represented as a directed graph. Therefore, by labeling each directed arc as 0 or 1 according to its direction, it can be represented as a bit string of length $mn(n-1)/2$. For example, consider a 3×3 problem given in Table 2.1 and a solution given in Figure 2.2 in Chapter 2. According to Figure 2.8, each arc of the graph has a labelling 0 or 1. The only thing we need to do is to specify the order of arcs. Note that each arc represents the precedence relation between two jobs J_i and J_j on the same machine M_r; hence an arc is specified by a triplet (i, j, r). An intuitive ordering between two arcs (i, j, r) and (i', j', r') is a machine-based ordering defined as:

$$(i, j, r) < (i', j', r') \Longleftrightarrow (r < r' \text{ or } (r = r' \text{ and } (i, j) < (i', j'))) \tag{4.1}$$

where,

$$(i, j) < (i', j') \Longleftrightarrow (i < i' \text{ or } (i = i' \text{ and } j < j')) \tag{4.2}$$

The solution schedule given in Figure 2.2 and Figure 2.8 is encoded as follows:

$$111 \mid 100 \mid 011 \tag{4.3}$$

The encoding from a schedule to a bit string based on the machine-based ordering of arcs in the disjunctive graph corresponding to the schedule is called machine-based encoding and decoding.

Another ordering is called a job-based ordering defined as follows:

$$\begin{aligned}(i, j, r) &< (i', j', r') \\ &\Longleftrightarrow ((i, j) < (i', j') \text{ or } ((i, j) = (i', j') \text{ and } o_j(r) < o_j(r')))\end{aligned} \tag{4.4}$$

where $o_j(r) < o_j(r')$ indicates that J_j is processed on M_r prior to $M_{r'}$. In other words, two arcs corresponding to the same job pair (J_i, J_j) are ordered according to the processing order of the first job J_i. If we use the job-based ordering, then solution schedule given in Figure 2.2 and Figure 2.8 is encoded as follows:

$$100 \mid 101 \mid 101 \tag{4.5}$$

where the vertical bars are inserted as job-pair delimiters; the partitions correspond to job pair $(1, 2)$, $(1, 3)$ and $(2, 3)$ from left to right respectively. The encoding from a schedule to a bit string based on the job-based ordering of arcs in the disjunctive graph corresponding to the schedule is called job-based encoding and decoding.

As another example, Figure 4.1 shows a simplified Gantt chart representation of an optimal schedule for the mt06 problem defined in Table 2.3. In the figure, the number indicates job number and consecutive sequence of the same number represents an operation for the job. The repetitions of the same number represents the processing time. For example, the left most sequence 111 on machine M_1 represents an operation for job J_1 on machine M_1 with processing of 3 time units and it starts at time unit 7. Figure 4.2 shows a binary representation of the optimal solution given in Figure 4.1 using the job-based ordering. For the ease of understanding, one long bit string is partitioned and divided per each job pair. For example, the first bit substring represents that J_1 is processed prior to J_2 on M_3 and also on M_1 but J_2 is prior to J_1 on M_2. Note that in the optimal schedule, like the one in this example, there is a tendency that the same bit continues in each substring of the same job pair. This confirms a heuristic that the processing priority for each job pair tends to be unchanged. This is especially true for easy problems in which each technological sequence of jobs on each machine is similar to each other.

As described in the previous chapter, the simple one-point or two-point crossover exchange chunks of bit sequences between parents. By using the job-based ordering, the consecutive same bits are likely to be exchanged together and thus this tendency, once acquired, is not destroyed easily.

4.2 Local harmonization

In the previous section, we have seen a couple of encoding methods to convert a solution schedule into a bit string. In those methods, different solution schedules are mapped into different bit strings. However, an arbitrary bit string generated by hand or crossover or mutation may not necessarily mapped back into a feasible solution schedule. In fact, the directed graph obtained from any bit string by selecting each arc's direction according to zero or one of corresponding bit

```
M1:       111    44444333333333 66666666662222222222555
M2: 2222222244444666111111555  3
M3: 333331  2222255555555544444                        6
M4:      3333       666        4441111111           22225
M5:              2222222222  555553333333444444446666111111
M6:          33333333  666666666222222222255551114444444444
```

Figure 4.1: An optimal schedule for the mt06 (6×6) problem (makespan = 55)

(J_1, J_2) : 110100
(J_1, J_3) : 011000
(J_1, J_4) : 110010
(J_1, J_5) : 111100
(J_1, J_6) : 110000
(J_2, J_3) : 101000
(J_2, J_4) : 111100
(J_2, J_5) : 111111
(J_2, J_6) : 111000
(J_3, J_4) : 111001
(J_3, J_5) : 111100
(J_3, J_6) : 111101
(J_4, J_5) : 110100
(J_4, J_6) : 111010
(J_5, J_6) : 101000

Figure 4.2: A binary representation of a solution schedule using the job-based ordering corresponding to the solution given in Figure 4.1. The first line corresponds to the precedence relation between J_1 and J_2. The first three digits of the bit-string on the first line are 110. This corresponds to the fact that J_1 is processed prior to J_2 on J_1's first and second machines M_3 and M_1, but is not prior to J_2 on J_1's third machine M_2 and so on.

value may contain cycles. In such cases, the bit string, i.e. genotype, is called *fatal* or *illegal*, and a bit string is called *feasible* when it corresponds to an executable schedule with corresponding directed graph being acyclic.

A repairing procedure that generates a feasible bit string, as similar to an illegal one as possible, is called the *harmonization algorithm* [24]. 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 harmonization.

The local harmonization works separately for each machine and resolves the cycle within each machine by changing directions of arcs. Assume that we are resolving cycles on machine M_r. A set of nodes G_r of machine M_r is initialized as $G_r = \{O_{1r}, O_{2r}, \ldots, O_{nr}\}$ (operations and nodes are identified here). First the least priority node $O_{lr} \in G_r$ is identified as a node that has the highest number of incoming arcs from $G_r \setminus O_{lr}$ (break ties arbitrarily). If this node has any outgoing arc to any node in $G_r \setminus O_{lr}$, then the direction of the arc is reversed so that O_{lr} has only incoming nodes from $G_r \setminus O_{lr}$, and G_r is updated as $G_r := G_r \setminus O_{lr}$. This process is repeated until G_r becomes empty, and as a result, the local inconsistensy is completely removed. Figure 4.3 shows an example of the local harmonization for machine M_1. In the figure, O_{31} is first identified as the least priority node, and the arc $O_{31} \rightarrow O_{61}$ is reversed such that O_{31} becomes the last operation on machine M_1. O_{21} is then identified as the second least priority node, and the arc $O_{21} \rightarrow O_{41}$ is reversed such that O_{21} becomes the second last operation on machine M_1 and so on. The obtained consistent ordering is $O_{41} \rightarrow O_{61} \rightarrow O_{51} \rightarrow O_{11} \rightarrow O_{21} \rightarrow O_{31}$.

4.3 Global harmonization

The global harmonization removes ordering inconsistencies between machines. Even after the local harmonization, there may exist cycles in the graph. In Figure 4.4, for example, there is a cycle connecting $O_{23} \rightarrow O_{22} \rightarrow O_{32} \rightarrow O_{31} \rightarrow O_{33}$, and again, $\rightarrow O_{23}$. The global harmonization changes the directions of minimum number of arcs so that there exists no cycles. 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.

4.4 Forcing

An illegal bit string produced by genetic operations can be considered as a genotype, and a feasible schedule 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

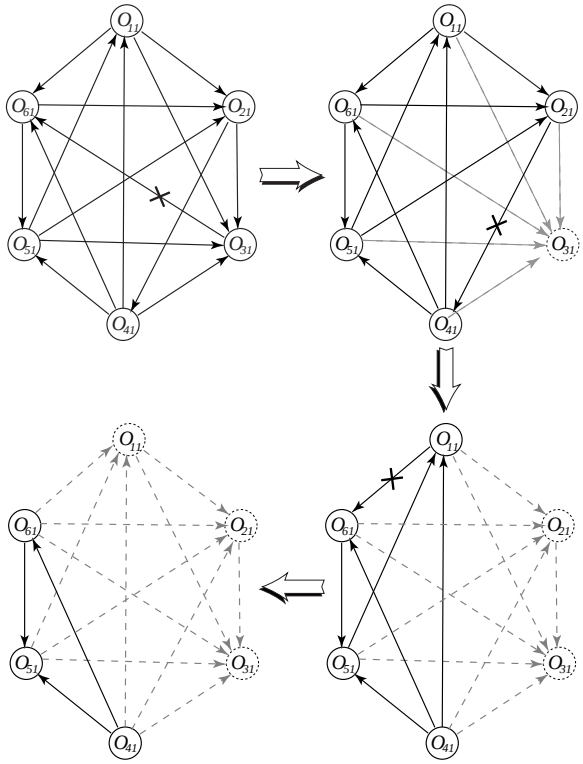

Figure 4.3: An example of the local harmonization resolving cycles within six operations $O_{11}, O_{21}, \ldots, O_{61}$ on the same machine M_1 where the arcs $O_{31} \rightarrow O_{61}$, $O_{21} \rightarrow O_{41}$ and $O_{11} \rightarrow O_{61}$ are reversed in this order and a consistent ordering $O_{41} \rightarrow O_{61} \rightarrow O_{51} \rightarrow O_{11} \rightarrow O_{21} \rightarrow O_{31}$ is eventually obtained.

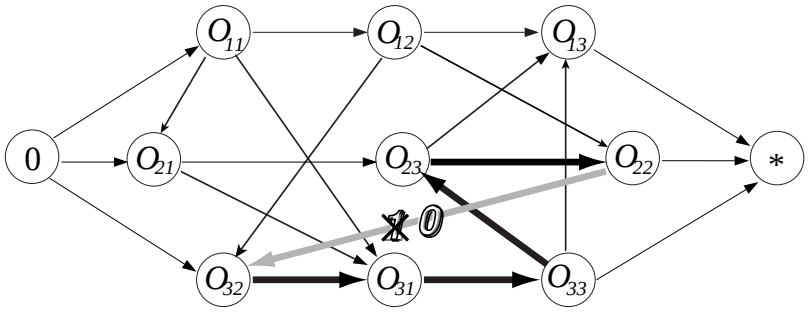

Figure 4.4: An example of global harmonization where a cycle $O_{23} \rightarrow O_{22} \rightarrow O_{32} \rightarrow O_{31} \rightarrow O_{33} \rightarrow O_{23}$ is resolved by reversing an arc $O_{22} \rightarrow O_{32}$.

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, usually the best 5% in the population. Such limited forcing brings about at least two merits: a significant improvement in the convergence speed and the solution quality.

4.5 Simple GA for the JSP

Using the job-based encoding, standard crossover/mutation, global/local harmonizations and forcing, a simple GA for the JSP can be constructed. Because the JSP is a minimization problem, the fitness is defined by the ranking method (3.1) and standard roulette wheel selection is utilized. The outline of the simple GA for the jobshop scheduling problem is described in Algorithm 4.5.1.

4.6 The Limitation of the Simple Approach

The simple GA approach described in this chapter can be applied to small problems such as 6×6 problem given in Table 2.3. Table 4.1 summarizes the experimental results for the mt benchmark problems. The column labeled SGA shows the best makespans obtained by the SGA and the column labeled Optimal shows the known optimal makespans. In fact, the optimal schedule shown in Figure 4.1 is obtained by the GA. However, larger problems such as 10×10 and 20×5 are not tractable by this simple approach.

Table 4.1: Experimental results of the simple GA for mt benchmark problems

Prob. (size)	mt06 (6×6)	mt10 (10×10)	mt20 (20×5)
SGA	55	965	1215
Optimal	55	930	1165

Algorithm 4.5.1 (Simple GA for the JSP)

A $n \times m$ scheduling problem is given as an input. The GA parameters: N, the population size, R_c, crossover ratio, R_m, mutation ratio are also given.

1. Initialize $P(t)$ as a random population $P(t = 0)$ of size N, where each random individual is a bit string of length $m \times n \times (n-1)/2$.
2. Modify $P(t)$ by applying one-point (3.2), two-point (3.2) or uniform (3.2) crossover to the randomly selected $R_c \times N$ members of $P(t)$ and obtain $P'(t)$.
3. Modify $P'(t)$ by apply bit-flip mutation (3.2) to the randomly selected $R_m \times N$ members of $P'(t)$ and obtain $P''(t)$.
4. Evaluate $P''(t)$ by the following steps;
   (a) Decode each individual p in $P''(t)$ by using the job-based decoding based on (4.1) into S, with the local and global harmonization methods to repair illegal bit strings.
   (b) Calculate the objective function f of p as $f(p) = C_{max}(S)$
   (c) Calculate fitness F of p by using the ranking method shown in (3.1).
   (d) Apply forcing to retain the phenotype of small number of elitest individuals to the next generation.
5. Reproduce $P(t+1)$ from $P''(t)$ by the roulette wheel selection
6. Set $t \leftarrow t + 1$
7. repeat from 2 to 5 until some termination condition is met.
8. Output the best individual in $P(t)$.

Chapter 5

GT-GA: A Genetic Algorithm based on the GT Algorithm

As seen in the previous section, conventional GAs can be applied to the jobshop scheduling problem in a rather straightforward way without major difficulties; a solution is represented as a bit string and conventional genetic operators such as 1-point, 2-point and uniform crossover and bit-flip mutation are applied. Because of the complicated constraints of the problem, however, an individual generated by such genetic operators is often infeasible; its phenotype does not represent an executable solution, and requires several steps of repairing process such as local and global harmonizations.

Obviously one of the advantages of the GA is its robustness over a wide range of problems with no requirement of domain specific adaptations. Hence genetic operators deal with genotype, which is domain independent, and are separated from domain specific decoding process from genotype to phenotype. However from the performance viewpoint, it is often more efficient to directly incorporate domain specific features into the genetic operators and skip wasteful intermediate decoding steps. Thus the GT crossover and the genetic algorithm based on GT crossover, denoted as GT-GA, has been proposed by Yamada and Nakano [25] and has the following properties.

- The GT crossover is a problem dependent crossover operator that utilizes the GT algorithm.
- The crossover operates directly on phenotype.
- In the crossover, parents cooperatively give a series of decisions; as which operation should be processed next, to build a new schedule. These decisions are made based on their own scheduling orders.
- The offspring represent active schedules, so there is no repairing process required.

Before describing the GT crossover, let us review some other crossover operators based on non-binary encodings for comparisons and discuss their advantages and disadvantages in the following sections.

5.1 Subsequence Exchange Crossover

As shown in Figure 2.3 in Section 2.1, a schedule of the JSP can be represented by a solution matrix, in other words, the set of permutations of jobs on each machine. When the matrix is expanded in one dimensional array as shown in Figure 5.1, it is called an *m*-partitioned permutation, where the permutation in the *k*-th partition (from the left) corresponds to the processing order of jobs on machine M_k. A solution represented by a *m*-partitioned permutation is regarded as genotype to which a crossover operator is applied.

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

Figure 5.1: The solution given in Figure 2.3 is converted to an *m*-partitioned permutation for $m = 3$, where the permutation in the *k*-th partition corresponds to the processing order of jobs on machine M_k

The *Subsequence Exchange Crossover* (SXX) was proposed by Kobayashi, Ono and Yamamura [33]. The SXX is a natural extension of the subtour exchange crossover for TSPs presented by the same authors [34]. Let two *m*-partitioned permutations be p_0 and p_1, which correspond to two feasible solution schedules. 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 job numbers. The SXX first identifies exchangeable subsequence pairs in p_0 and p_1 on each machine and interchanges each pair to produce new *m*-partitioned permutations k_0 and k_1. Figure 5.2 shows an example of the SXX for a 6×3 problem. In the figure, each underlined subsequence pair is identified as exchangeable and interchanged.

The SXX ensures that k_0 and k_1 are always valid *m*-partitioned permutations and therefore, there are no inconsistencies within each machine to be resolved by the local harmonization described in the previous section. However, there may exist inconsistencies between machines that must be resolved by the global harmonization.

5.2 Precedence Preservative Crossover

Another representation that uses an *unpartitioned permutation of n job numbers with m-repetitions* has been proposed by Bierwirth [35]. In this representation, we consider a permutation of *n* job numbers but each identical job number occurs *m* times. When such a permutation with repetitions is given, it is decoded into a feasible schedule by scanning the permutation from left to right and referring the *k*-th occurrence of a job number to the *k*-th operation in the technological sequence of this job. Figure 5.3 shows an example of this decoding process. In the figure, a permutation of three job numbers with three repetitions is given in the top and there are three rows labeled M_1, M_2 and M_3 in the bottom. We consider decoding this permutation into a solution schedule of 3×3 problem given in Table 2.1 and Figure 2.1. From the job sequence matrix $\{T_{jk}\}$ given in Figure 2.1, we see that job J_1, for example, is first processed on M_1, therefore, the first

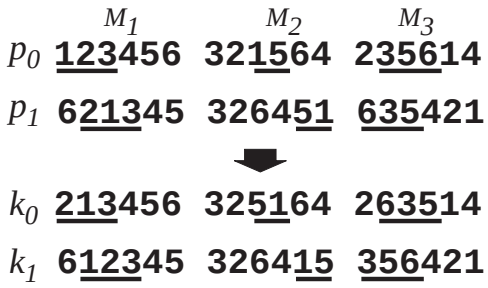

Figure 5.2: An example of subsequence exchange crossover (SXX), where each underlined subsequence pair one from p_0 and the other from p_1 on each machine is identified as exchangeable and interchanged to generate k_0 and k_1

occurrence of 1 in the permutation should be moved straight down to the row of M_1. Likewise, J_1 is then processed on M_2 and M_3 in this order, so the second and third occurrences of 1 should be moved down to the rows of M_2 and M_3 respectively. By moving all the job numbers down to one of M_1, M_2, or M_3 rows, a solution schedule is obtained. In this case the schedule obtained is identical to the one given in Figure 2.3.

The advantage of this representation is that an arbitrary permutation with repetitions can be decoded into a feasible schedule. Therefore, no repairing processes such as local and global harmonizations are required.

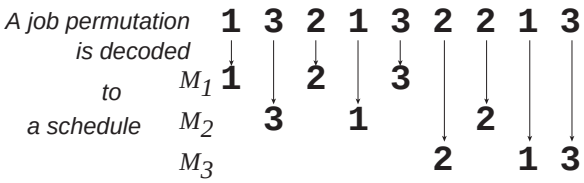

Figure 5.3: A job sequence (permutation with repetition) for a 3×3 problem defined in Figure 2.1 is decoded to a schedule, which is equivalent to the one in Figure 2.3.

A crossover operator called *Precedence Preservative Crossover* (PPX) is proposed for this representation in [36]. The PPX perfectly respects the absolute order of genes in parental chromosomes as follows. Assume we have two parents p_0 and p_1 encoded in permutation with repetitions representation and consider generating a new individual k also represented in a permutation with repetitions. First, a template bit string h of length mn is given that determines from which parent, p_0 or p_1, should genes to be drawn to generate a new individual. The bit value is zero means that the corresponding gene should be copied from p_0 and one from p_1. When a gene is drawn from one parent and then appended to the offspring chromosome, it is deleted from the parent and the corresponding gene is also deleted from the other parent. This step is repeated until both parent chromosomes are empty and the offspring contains all genes involved.

Figure 5.4 shows an example of this crossover. Starting from the top left picture, the first two bits of h are both zero, so the first and second job numbers 3 and 2 in k are copied from p_0 as shown in the round boxes. The leftmost occurrences of job numbers 3 and 2 are deleted from both p_0 and p_1 in the top right picture. The first two bits in h are deleted as well. Then the leftmost non-deleted bits in h become four ones shown in the square box, which means that the next four job numbers should be copied from p_1. The leftmost non-deleted four job numbers in p_1 are 1121 which are copied to k as shown in the square boxes. In the bottom picture, the leftmost non-deleted occurrences of job numbers 1121 are then deleted from p_0 as well as from p_1. The four ones in the square box are also deleted from h. The remaining non-deleted bits in h are three zeros, which indicates that the remaining job numbers should be copied from p_0 (however the remaining non-deleted permutation 233 is identical both in p_0 and p_1 in this example).

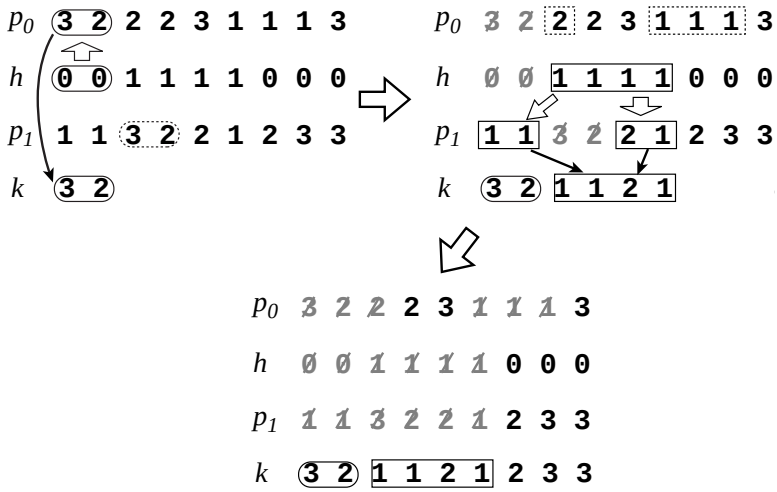

Figure 5.4: An example of the precedence preservative crossover (PPX), where k is generated from p_0 and p_1 using h

5.3 GT Crossover

Unlike other crossover operators described in the previous two sections, the GT crossover, GTX in short, directly operates on the solution matrix representation of a schedule given in Figure 2.3. In this sense, we have no distinction between genotype and phenotype here. Assume we have two parents p_0 and p_1 both represented by solution matrices and consider generating a new individual k also represented in a solution matrix. Let H be a binary matrix of size $m \times n$,where $H_{ri} = 0$ means that the i-th operation on machine r should be determined by the first parent p_0 and $H_{ri} = 1$ by the second parent p_1 [25, 37]. H is called a inheritance matrix. The role of H_{ri} is similar to that of h described in Section 5.2. In fact, the idea of the GT crossover including the use of H_{ri} is first proposed and later adopted to the precedence preservative crossover.

Algorithm 5.3.1 (GT crossover)

A scheduling problem represented by $\{T_{jk}\}$, the technological sequence matrix, and $\{p_{jk}\}$, the processing time matrix as well as two solution schedules p_0 and p_1 represented by solution matrices $S^0 = \{S^0_{rk}\}$ and $S^1 = \{S^1_{rk}\}$ respectively, are given as inputs.

1. Initialize G as a set of operations that are first in the technological sequence; i.e., $G = \{O_{1T_{11}}, O_{2T_{21}}, \ldots, O_{2T_{n1}}\}$. For each operation $O \in G$, set $ES(O) := 0$ and $EC(O) := p(O)$.

2. Find the earliest completable operation $O_{*r} \in G$ by (2.1) with machine M_r. A subset of G that consists of operations processed on machine M_r is denoted as G_r.

3. Calculate the conflict set $C[M_r, i] \subset G_r$ by (2.2), where $i-1$ is the number of operations already scheduled on M_r.

4. Select one of the parents $\{p_0, p_1\}$ as p according to the value of H_{ri}, that is, $p := p_{H_{ri}}$ and $S^p := S^{H_{ri}}$. For each $O_{jr} \in C[M_r, i]$ with job number j, there exists an index l such that $S_{rl} = j$. Let l_m be the smallest index number among them; i.e., $l_m := \min\{l \mid S_{rl} = j \text{ and } O_{jr} \in C[M_r, i]\}$ and let $k := S_{rl_m}$. This results in selecting an operation $O_{kr} \in C[M_r, i]$ that has been scheduled in p earliest among the members of $C[M_r, i]$.

5. Schedule O_{kr} as the i-th operation on M_r; i.e. $S_{ri} := k$, with its starting and completion times equal to $ES(O_{kr})$ and $EC(O_{kr})$ respectively: $s(O_{kr}) = ES(O_{kr}), c(O_{kr}) = E(CO_{kr})$.

6. For all $O_{jr} \in G_r \setminus \{O_{kr}\}$, Update $ES(O_{jr})$ as

   $ES(O_{jr}) := \max\{ES(O_{jr}), EC(O_{kr})\}$ and $EC(O_{jr})$ as $EC(O_{kr}) := ES(O_{kr}) + p(O_{kr})$.

7. Remove O_{kr} from G (and therefore from G_r), and add operation O_{ks} that is the next to O_{kr} in the technological sequence to G if such O_{ks} exits; i.e., if $r = T_{ki}$ and $i < m$, then $s := T_{ki+1}$ and $G := (G \setminus \{O_{kr}\}) \cup \{O_{ks}\}$.

   Calculate $ES(O_{ks})$ and $EC(O_{ks})$ as:

   $ES(O_{ks}) := \max\{EC(O_{kr}), EC(PM(O_{ks}))\}$ and $EC(O_{ks}) := ES(O_{ks}) + p(O_{ks})$ respectively.

8. Repeat from Step 1 to Step 7 until all operations are scheduled.

9. Output the solution matrix $\{S_{rk}\}$ as the active schedule obtained with the set of starting and completion times $\{s(O_{jr})\}$ and $\{c(O_{jr})\}$ respectively where $j = S_{rk}$.

The GT crossover can be defined by modifying Step 4 of Algorithm 2.2.1, where the choice of the next operation from the conflict set $C[M_r, i]$ was at random. In the GT crossover, the choice is made by looking at the processing order in one of the parents specified by H and an operation that has been scheduled in the parent earliest among the members of the conflict set is selected. By doing so, it tries to reflect the processing order of the parent schedules to their offspring. Note that if the parents are identical to each other, the resulting new schedule is also identical to those of the parents. In general the new schedule inherits partial job sequences of both parents in different ratios depending on the number of zeros and ones contained in H.

Algorithm 5.3.1 describes the GT crossover. For the purpose of self-completeness, it is presented as a complete form, but the differences between the GT algorithm and the GT crossover are only the inputs and Step 4. The other steps are just the exact copies of the GT algorithm. 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.

Figure 5.5 shows an example of the GT crossover applied to the two parents p_0 and p_1 represented by solution matrices with an inheritance matrix H and generating k as an offspring when a problem is given as shown in Figure 2.3. For better understanding, correspondig solutions represented by the simplified gantt chart introduced in Figure 4.1 are also shown in the square boxes. Each number with an arrow pointing to an operation indicates the order of the corresponding operation selected in Step 4 of Algorithm 5.3.1. This order is dynamically assigned in the algorithm. For example, the first operation on machine M_1 should be first determined in this case. The corresponding bit in H is consulted and here, we see $H_{11} = 1$ which means the operation should be determined from p_1. Because the first operation on machine M_1 in p_1 is O_{11}, O_{11} is also scheduled as the first operation on machine M_1 in k. The next operation to be determined is the first operation on machine M_2 and $H_{21} = 1$ indicates that this operation should be determined again from p_1, thus O_{32} is selected in k. In the similar way, the next operation to be determined is the second operation on machine M_1 and H_{12} indicates that this operation should be determined this time from p_0, thus O_{21} is selected in k. Repeating this process, the complete schedule shown in the right is finally obtained as k. One can see, for example, that the whole job sequence on M_3 is consequently copied from p_0 because corresponding bits in H are all zeros.

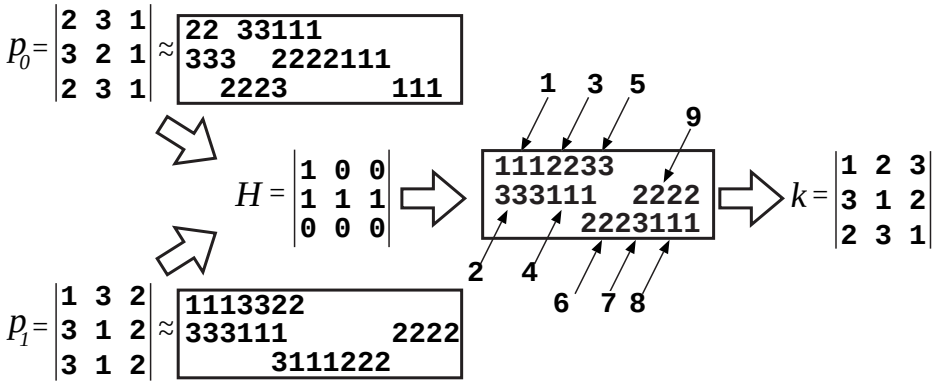

Figure 5.5: GT crossover

While applying the GT crossover, simulated random *copy error* is incorporated as mutation built into the GT crossover. More precisely, in Step 4 of Algorithm 5.3.1, the n-th ($n > 1$) smallest index number ${l^n}_m$ is selected instead of $l_m = {l^1}_m$ and the corresponding operation in $C[M_{r^*}, i]$ that is the n-th earliest scheduled operation in p among the members of $C[M_{r^*}, i]$ is selected with a small probability R_m When the two parents p_0 and p_1 are identical, then the offspring k generated by the GT crossover is also identical to p_0 and p_1, however, with the mutation incorporated, k can be slightly different from the parents.

5.4 GT-GA

A Genetic Algorithm based on the GT crossover is straightforward. The general procedure described in Section 3.3 is used without major modifications. The following points should be mentioned.

1. In the GT-GA, each individual is always a feasible schedule represented by a solution matrix. In fact, each individual is not only feasible but also an active schedule. As described earlier, we have no distinction between genotype and phenotype here.

2. Because the problem is a minimization problem, the rank-based roulette wheel selection method is used.

3. An elitest strategy to preserve the best individual in the current population to the next generation is used.

5.5 Computational Experiments

GT-GA is applied to the mt benchmark problems to explore its efficiencies and limitations. Table 5.1 shows the best solutions obtained by the GT-GA for each problem. For the mt06 problem the optimal schedule with makespan 55 is immediately obtained even with small population. For the mt10 and mt20 problems, 600 trials are performed with different random number seeds in each trial.

For the GA parameters, the population size $N < 100$ is used for the mt06 problem, $N = 1000$ and $N = 2000$ are used for the mt10 and mt20 problems respectively with high crossover rate $R_c > 0.9$ and low mutation rate $R_m < 0.01$. Each GT-GA run is terminated after 200 generations. Figure 5.6 shows the histgram of the obtained best solutions of the mt10 problem for 600 trials. For example, the optimal schedule of the mt10 problem with makespan 930 was obtained four times among 600 trials.

From the experimental results, we can observe that notorious mt10 problem can be solved to optimally even with this simple algorithm. The success is limited in a sense that the optimal makespan for the mt10 problem is obtained only occasionally among many trials and for the mt20 problem, the optimal makespan cannot be obtained. However, considering the simplicity of the algorithm, the results are still interesting.

Algorithm 5.4.1 (A Genetic Algorithm using the GT crossover)

As always, we are given a jobshop scheduling problem represented by $\{T_{jk}\}$, the technological sequence matrix, and $\{p_{jk}\}$, the processing time matrix. Besides, the following GA parameters are given: population size N, crossover rate R_c and mutation rate R_m.

1. A random initial population $P(t = 0)$ of size N is constructed in which each individual is generated using the GT algorithm with randomly selecting operations in Step 4 of Algorithm 2.2.1. The makespan of each individual is automatically calculated as an output of the GT algorithm.

2. Select randomly $N \times R_c$ individuals from $P(t)$ and pair them randomly. Apply the GT crossover (with built-in mutation of probability R_m) to each pair and generate new $N \times R_c$ individuals that are inserted into $P'(t)$. The rest of $P(t)$ members are just copied to $P'(t)$. As a result of the GT crossover, the makespan of each individual is automatically calculated.

3. If the best makespan in $P'(t)$ is not as good as that in $P(t)$, then the worst individual in $P'(t)$ is replaced by the best individual in $P(t)$ (elitest strategy).

4. Reproduce $P(t + 1)$ from $P'(t)$ by using the rank-based roulette wheel selection, in which each individual in $P'(t)$ is sorted in the descending order of its makespan so that the worst individual is numbered as x_1 and the best as x_N. Then the roulette wheel selection is applied with the fitness f of an individul x_i defined as $f(x_i) = i$ to obtain $P(t + 1)$.

5. Set $t \leftarrow t + 1$

6. Repeat from Step 2 to 5 until some termination condition is met.

7. Output the best individual in $P(t)$ as the obtained best solution.

Table 5.1: Experimental results of the GT-GA for mt benchmark problems

Prob. (size)	mt06 (6×6)	mt10 (10×10)	mt20 (20×5)
SGA	55	965	1215
GTGA	55	930	1184
Optimal	55	930	1165

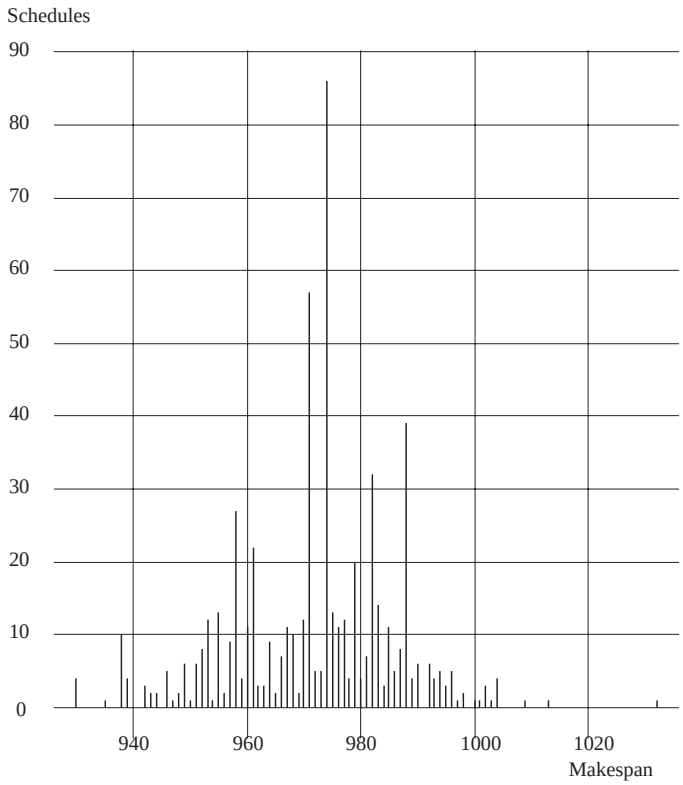

Figure 5.6: The histgram of the best makespans obtained by the GT-GA after 200 generations among 600 trials for the mt10 problem

5.6 Concluding Remarks

Figure 5.7 summarizes the relationship between various crossover operators discussed in this and previous chapter. The ordinate indicates the level of problem independence of each solution representation. The SGA described in the previous chapter uses the binary codings and does not use any domain knowledge of the scheduling problem, therefore, gives the most general representation. The SXX uses the property that the scheduling problem can be represented by the permutation problem but does not incorporate the fact that permutations on each machine are not mutually independent. The PPX utilizes this fact but does not directly use the fact that in the makespan minimizing scheduling problem, the optimal schedules are always active schedules, therefore, we can restrict the search for the optimal schedule within the set of all active schedules. The GT crossover denoted as GTX in the figure incorporates all these domain knowledges therefore the most efficient but still simple enough.

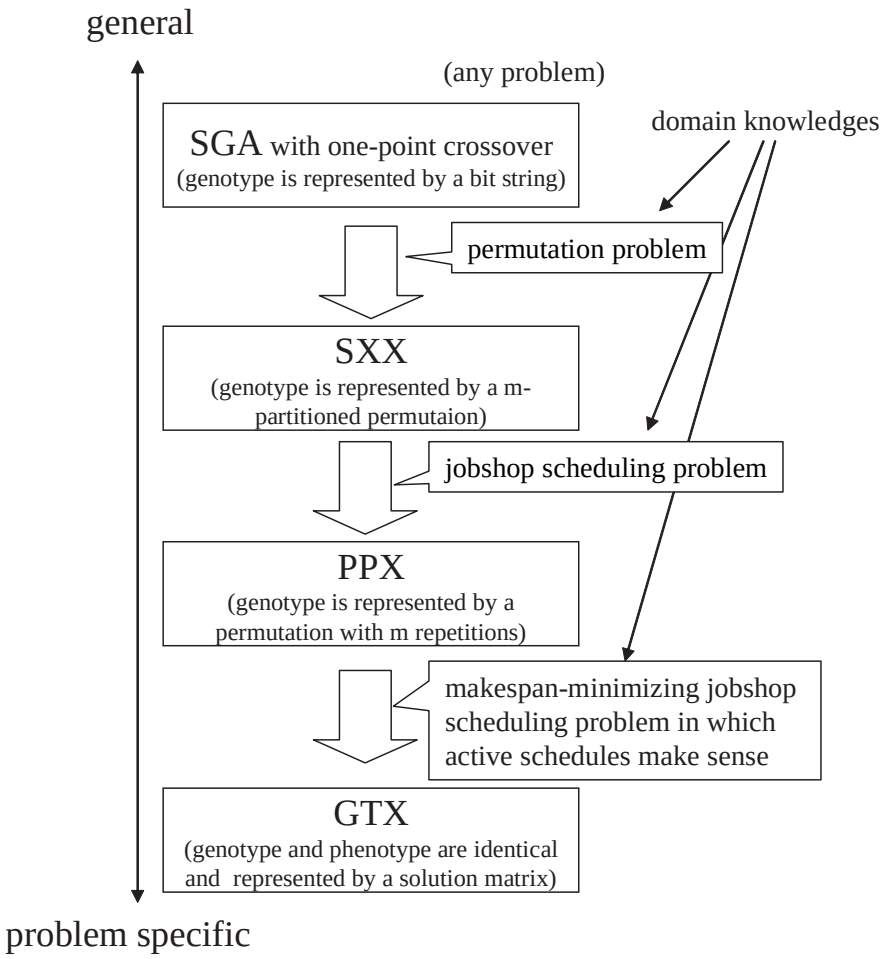

Figure 5.7: relationship between various crossover operators

Chapter 6

Neighborhood Search

As is now universally appreciated, it is not really likely that optimal solutions to large combinatorial problems will be found reliably by any exact method, although it is possible to find classes of instances where problem-specific methods can achieve good results. However, for problems that are NP-hard [38], it is now customary to rely on the application of heuristic techniques [39, 40]. These techniques include what some call the 'metaheuristics'—simulated annealing (SA) and tabu search (TS)—as well as genetic algorithms (GAs) which are already discussed in the earlier chapters. Central to most heuristic search techniques is the concept of neighborhood search (NS). In this chapter, the general concept of the neighborhood search is first reviewed and the well-known instances of metaheuristics, SA, TS, and GAs are formulated in this context so that the differences and characteristics of those methods become clear.

6.1 The Concept of the Neighborhood Search

If we assume that a solution is specified by a vector $\mathbf{x}$, that the set of all (feasible) solutions is denoted by $\mathbf{X}$ (which we shall also call the *search space*), and the cost of solution $\mathbf{x}$ is denoted by $f(\mathbf{x})$, then every solution $\mathbf{x} \in \mathbf{X}$ has an associated set of *neighbors*, $N(\mathbf{x}) \subset \mathbf{X}$, called the neighborhood of $\mathbf{x}$. Each solution $\mathbf{x}' \in N(\mathbf{x})$ can be reached directly from $\mathbf{x}$ by an operation called a *move*, a single perturbation of $\mathbf{x}$. Many different types of move are possible in any particular case, and we can view a move as being generated by the application of a transition operator ω. For example, if $\mathbf{X}$ is the n-dimensional binary hypercube $\mathbb{Z}_2^n$, a simple transition operator is the bit flip $\phi(k)$

$$\phi(k) : \mathbb{Z}_2^n \to \mathbb{Z}_2^n \quad \begin{cases} z_k \mapsto 1 - z_k \\ z_i \mapsto z_i & \text{if } k \neq i \end{cases} \tag{6.1}$$

where $\mathbf{z}$ is a binary vector of length l.

As another example, we can take the forward shift operator for the case where $\mathbf{X}$ is $\mathbf{\Pi}_n$—the

space of permutations $\boldsymbol{\pi}$ of length n. The operator $\mathcal{FSH}(i, j)$ (where we assume $i < j$) is

$$\mathcal{FSH}(i, j) : \mathbf{\Pi}_n \rightarrow \mathbf{\Pi}_n \quad \begin{cases} \pi_k \mapsto \pi_{k-1} & \text{if } i < k \leq j \\ \pi_i \mapsto \pi_j & \\ \pi_k \mapsto \pi_k & \text{otherwise} \end{cases} \tag{6.2}$$

The permutation flowshop scheduling problem with n jobs and m machines and with any objective function, such as $n/m/P/C_{max}$ or $n/m/P/C_{sum}$ introduced in Chapter 2 can be considered as a typical example of this permutation space.

An analogous backward shift operator $\mathcal{BSH}(i, j)$ can similarly be described; the composite of $\mathcal{BSH}$ and $\mathcal{FSH}$ is denoted by $\mathcal{SH}$. Another alternative for such problems is an exchange operator $\mathcal{EX}(i, j)$ which simply exchanges the elements in the ith and jth positions.

Algorithm 6.1.1 A general structure of Neighborhood Search

A cost function of $\mathbf{x} \in \mathbf{X}$ is given as $f(\mathbf{x})$ and neighborhood of $\mathbf{x}$ as $N(\mathbf{x})$. Certain criteria are given to select $\mathbf{y} \in N(\mathbf{x})$ based on the value $f(\mathbf{y})$.

1. Select a starting point $\mathbf{x_0} \in \mathbf{X}$ at random and set $\mathbf{x} = \mathbf{x_{best}} = \mathbf{x_0}$.
2. **do**
   (a) a candidate $\mathbf{y}$ is chosen from $N(\mathbf{x})$ and is accepted or rejected according to the given criteria based on the value $f(\mathbf{y})$. Set $\mathbf{x} = \mathbf{y}$ if $\mathbf{y}$ is accepted, otherwise repeat this step until some $\mathbf{y}$ is accepted .
   (b) If $f(\mathbf{x}) < f(\mathbf{x_{best}})$ then set $\mathbf{x_{best}} = \mathbf{x}$.

   until termination conditions are satisfied.
3. Output $\mathbf{x_{best}}$ as the best solution obtained.

A typical neighborhood search (NS) heuristic procedure is shown in Figure 6.1.1. As shown in the figure, NS operates by generating neighbors in an iterative process where a move to a new solution is made whenever certain criteria are fulfilled in Step 2a. There is a great variety of ways in which candidate moves can be chosen for consideration, and in defining criteria for accepting candidate moves. Perhaps the most common case is that of *ascent*, in which the only moves accepted are to neighbors that improve the current solution. *Steepest* ascent corresponds to the case where all neighbors are evaluated before a move is made—that move being the best available. *Next* ascent is similar, but the next candidate (in some pre-defined sequence) that improves the current solution is accepted, without necessarily examining the complete neighborhood. Normally, the search terminates when no moves can be accepted.

6.2 Avoiding Local Optima

The trouble with NS is that the solution it generates is usually only a *local* optimum—a point in the search space none of whose neighbors offer an improved solution and NS does not guarantee to find the *global* optimum, the very best solution in the entire search space. In recent years many techniques have been suggested for the avoidance of local optima. At the most basic level, we could use *iterative restarts* of NS from many different initial points, thus generating a collection of local optima from which the best can be selected. There are more popular and intelligent principles. For completeness, We refer here briefly to some of the most popular ones. *Simulated annealing* uses a controlled randomization strategy—inferior moves are accepted probabilistically, the chance of such acceptance decreasing slowly over the course of a search. By relaxing the acceptance criterion in this way, it becomes possible to move out of the basin of attraction of a local optimum. *Tabu search* adopts a deterministic approach, whereby a 'memory' is implemented by the recording of previously-seen solutions. This record could be explicit, but is often an implicit one, making use of simple but effective data structures. These can be thought of as a 'tabu list' of moves which have been made in the recent past of the search, and which are 'tabu' or forbidden for a certain number of iterations. This prevents cycling, and also helps to promote a diversified coverage of the search space. *Perturbation methods* improve the restart strategy: instead of retreating to an unrelated and randomly chosen initial solution, the current local optimum is perturbed in some way and the heuristic restarted from the new solution. Perhaps the most widely-known of such techniques is the 'iterated Lin-Kernighan' (ILK) method introduced by Johnson [41] for the travelling salesman problem. On reaching a local optimum, a set of links is randomly chosen for removal and re-connection, in such a way that a new search can start relatively close to a new local optimum. Such techniques can perhaps best be described as *perturbation* methods. *Genetic algorithms* differ in using a population of solutions rather than moving from one point to the next. Furthermore, new solutions are generated from two (or, rarely) more solutions by applying a 'crossover' operator. However, they can also be encompassed within an NS framework, as we shall discuss later in this thesis.

6.3 The Neighborhood Structure for the Jobshop Scheduling Problem

As shown in Section 5.1 and 5.2, the jobshop scheduling problem with n jobs and m machines can be considered itself as a permutation problem; namely we have a permutation of n jobs on each machine, which results in m-partitioned n-job permutations. However, the simple $\mathcal{SH}$ and $\mathcal{EX}$ operators, for example, are not efficient because of the two reasons: (1) the size of the neighborhood becomes too large and (2) the resulting new permutation does not always correspond to a feasible schedule. One way to resolve these problems is to construct a neighborhood structure based on Theorem 1 in Chapter 2. Namely, given a schedule S, a transition operator that exchanges a pair of adjacent critical operations (i.e., operations on a critical path) on a same machine in S as shown in Figure 6.1 forms a neighborhood which we call the *AE (adjacent exchange) neighborhood* and denote $AE(S)$. Theorem 1 guarantees that $AE(S)$ members

are always feasible and Theorem 2 guarantees that an optimal schedule is reachable from any initial schedule by applying finite number of transitions. The transition operator was originally proposed by Balas in his branch and bound approach[6] and has been used as a neighborhood structure for SA in [12] and for TS in [17].

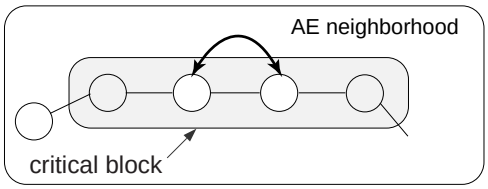

Figure 6.1: $AE(S)$, adjacent exchange neighborhood of S, consists of schedules obtained from S by exchanging a pair of adjacent operations within a same critical block.

Another very powerful transition operator was proposed in [9] using the notions of *before candidate* and *after candidate* introduced in Section 2.5 of Chapter 2. Let a schedule be S and let its critical blocks be $B_1, \ldots, B_k$, then *before candidate* B_j^B and *after candidate* B_j^A in a critical block B_j are defined by Equation 2.5. Let $NB^{B_j}(S)$ and $NA^{B_j}(S)$ be sets of (maybe infeasible) schedules obtained by moving each operation in B_j^B (or B_j^A) to the front (or rear) of B_j respectively as shown in Figure 6.2. Because we have Theorem 2.5, it is tempted to define the CB neighorhood as a set of all the shedules obtained from before and after candidates as follows:

$$CB'(S) = \bigcup_{B_j} \{NB^{B_j}(S) \cup NA^{B_j}(S)\}. \tag{6.3}$$

However unfortunately, there is no theorem similar to Theorem 1 that guarantees the feasibility of CB' members. In fact, CB' may contain infeasible schedules. Therefore the CB neighborhood is given as follows:

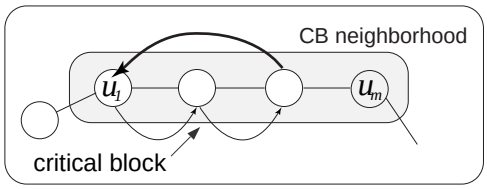

Figure 6.2: $CB(S)$, critical block neighborhood of S, consists of schedules obtained from S by moving an operation in a critical block to the front or the rear of the block.

$$CB(S) = \{S' \in CB'(S) \mid S' \text{ is a feasible schedule}\}. \tag{6.4}$$

In the next chapter, we will see that using the CB neighorhood, an efficient simulated annealing algorithm can be constructed.

Chapter 7

Critical Block Simulated Annealing for the Jobshop Scheduling Problem

7.1 Simulated Annealing

Consider that we have a nonlinear optimization function $f(\mathbf{x})$ defined over a continuous variable $\mathbf{x}$ in multi-dimensional Euclidean space $\mathbf{X}$. Such a nonlinear optimization function may be likened to a mountainous state space landscape, with the algorithm's objective being to locate the lowest valley. Simulated Annealing (SA) methods[1] are analogous to searching a state space landscape by bouncing a rubber ball around the terrain. The ball bounces around the landscape, in and out of different valleys, probabilistically sampling different locations. As the degree of bounce is reduced it becomes more difficult for the ball to bounce out of low valleys into higher ones, than vice versa. Finally, when there is no bounce left in the ball, the ball will settle in the lowest valley. This is mathematically guaranteed given time and a proper bounce (annealing) reduction schedule [42].

Thus, SA algorithms employ *noise* to choose new parameter values. They generate a new state $\mathbf{x}'$ in the neighborhood of $\mathbf{x}$, probabilistically. When $\mathbf{x}$ is a continuous variable, there are infinite number of candidate states in the neighborhood of $\mathbf{x}$ and thus, a new state $\mathbf{x}'$ is generated using a given distribution function $g()$ which will be described shortly. The algorithms calculate the value of the function cost $E = f(\mathbf{x}')$, and then probabilistically decides to *accept* or reject it. If accepted, the new state becomes the current state. The new state may be accepted even if it has a larger function cost than the current state. The criteria for acceptance is determined by an acceptance function $h()$, the temperature parameter T, and the difference in the function values of the the two states. Initially, T is large, and a new state is accepted quite frequently. As the algorithm progresses, T is reduced, lowering the probability that the acceptance function will accept a new state if it's functional cost is greater than that of the current state.

The general SA procedure [43] is defined below.

1. Choose an initial (high) temperature T_0 and a random state $\mathbf{x_0}$.

[1] Annealing (as in metallurgical annealing) refers to the process involving the slow reduction of a temperature.

$T_{k=0} \leftarrow T_0,\ \mathbf{x} \leftarrow \mathbf{x_0}$

2. Calculate the cost function value of the starting state.
   $E_{k=0} \leftarrow f(\mathbf{x_0})$
3. For each iteration k, $k = 1 \ldots k_f$ do the following:
   (a) Choose a new state x', using a generating function.
   $\mathbf{x}' \leftarrow g(\mathbf{x})$
   (b) Calculate the cost of $\mathbf{x}'$.
   $E' \leftarrow f(\mathbf{x}')$
   (c) Set $\mathbf{x} \leftarrow \mathbf{x}'$ and $E \leftarrow E'$ with probability determined by the acceptance function $h()$.
   (d) Reduce the temperature T by annealing
   (e. g. $T_{k+1} \leftarrow \gamma T_k,\ 0 < \gamma < 1$).
   (e) When T is lower than a sufficiently small value T_f, exit the loop.
4. Return $\mathbf{x}$ and E as the (near) optimal state and function cost value.

Because the algorithm occasionally chooses states *uphill* from its current state (i.e. chooses states with higher functional values than the current states), it can escape from local minima and more effectively search the function space to find the global minimum.

The Simulated Annealing method in general consists of a system state $\mathbf{x}$ and the following functional relationships:

1. $f(\mathbf{x})$: a cost function to minimize,
2. $g(\mathbf{x})$: a generating probability density function of new states.
3. $P(\mathbf{x})$: an acceptance function that decides if a new state should become the current state, and
4. $T(k)$: an annealing temperature (T) schedule.

For numeric optimization problems, $\mathbf{x}$ is often defined as an integer or real parameter vector, $\mathbf{x} = \{x^i; i = 1 \ldots D\}$, and Boltzmann Annealing is used to generate new states. Boltzmann Annealing employs a Gaussian probability density function,

$$g(\mathbf{x}') = \frac{1}{(2\pi T)^{D/2}} e^{-(\mathbf{x}-\mathbf{x}')^2/(2T)}$$

where $g(\mathbf{x}')$ is the probability of generating $\mathbf{x}'$ from the currently accepted state $\mathbf{x}$, and where the temperature T is a measure of the fluctuations of the Boltzmann distribution.

A Gaussian probability density distribution is not applicable for the generation of new states for combinatorial optimization problems including the job shop scheduling problem. Instead, a uniform random distribution is often used

$$g(\mathbf{x}') = 1/n, \mathbf{x}' \in N(\mathbf{x}) \tag{7.1}$$

where n is the number of states that can be directly generated by the generating function, i.e. n is the number of states in the neighborhood of $\mathbf{x}$, $n = |N(\mathbf{x})|$.

The acceptance probability function $P(\mathbf{x})$ is based on the chances of accepting a new state $\mathbf{x}'$ relative to the current state $\mathbf{x}$, i.e. the difference of their function values

$$P(\mathbf{x}') = \frac{e^{-f(\mathbf{x}')/T}}{e^{-f(\mathbf{x}')/T} + e^{-f(\mathbf{x})/T}} = \frac{1}{1 + e^{(f(\mathbf{x}')-f(\mathbf{x}))/T}}. \tag{7.2}$$

If lower cost states are always accepted, as in [44], the acceptance function above can be redefined as

$$P(\mathbf{x}') = \begin{cases} 1 & \text{if} \quad f(\mathbf{x}') \leq f(\mathbf{x}) \\ e^{(-f(\mathbf{x}')-f(\mathbf{x}))/T} & \text{otherwise.} \end{cases} \tag{7.3}$$

The practical annealing schedule, T_k, most often used to find the global minimum is of the form

$$T_k = T_0 e^{-ck} \tag{7.4}$$

where c is a positive constant.

7.2 Critical block Simulated Annealing

For the JSP, a state $\mathbf{x}$ is defined by a particular schedule S, and the cost $f(\mathbf{x})$ is defined by the makespan $C_{max}(S)$. A neighborhood $N(S)$ of a schedule S can be defined as the set of feasible schedules that can be reached from S by exactly one transition (a single perturbation of S). We use the critical block neighborhood $CB(S)$ defined by (6.4) in the previous chapter as the neighborhood structure.

The algorithm begins by setting the annealing temperature to an initial value and generating a random schedule S. The makespan and critical path of S is then calculated. Next, a new schedule S' in the neighborhood of S is randomly generated. The new schedule S' is compared with the current schedule S, and probabilistically accepted according to the makespan difference between the two schedules, and the annealing temperature. The temperature, initially quite high, is decreased according to a given annealing schedule. This process is repeated until 1) the temperature is sufficiently low, 2) a given number of iterations have occurred, or 3) a schedule having a (near) optimal makespan is found. Finally, the best generated schedule and its makespan are printed. The algorithm is described in 7.2.1.

Algorithm 7.2.1 (The Critical Block Simulated Annealing Algorithm)
We are given a jobshop scheduling problem to optimize, and the initial temperature T_0.

1. Set $S = S_0$ a randomly generated initial schedule, iteration step number $k = 0$, and $T_{k=0} = T_0$ the initial temperature.
2. **do**
   (a) **do**
      i. Pick $S' \in CB(S)$
      ii. Accept S' probabilistically according to the Metropolis Criterion distribution, i.e. choose S' with probability one if $C_{max}(S') \leq C_{max}(S)$, and $e^{-(C_{max}(S')-C_{max}(S))/T}$ otherwise, i.e., the probability P to accept S' is defined as follows:

$$P(S') = \begin{cases} 1 & \text{if } C_{max}(S') \leq C_{max}(S) \\ e^{-(C_{max}(S')-C_{max}(S))/T} & \text{otherwise.} \end{cases} \quad (7.5)$$

   until S' is accepted.
   (b) Set $S = S'$, increase k, and decrease T according to the annealing schedule.

   until termination (i.e. T is sufficiently small or k is sufficiently large or the minimal makespan or lower bound is found).
3. Print the best schedule found and its makespan.

7.3 Reintensification

Often during the search process 1) the system state wanders far from states that are leading to the global minima, or 2) the system state may become trapped in a deep local minima. Then all schedules generated from the current schedule will have longer makespans. If the acceptance temperature is low, it may be difficult for the system to escape from this local minimum and continue the search.

Occasionally a reintensification strategy may be applied to improve the search. This reintensification is similar to reannealing [45], [46] which is used in numerical function optimization to occasionally reset the system temperature and the state of the system after a sufficiently long period of time without finding a new global minima.

It is useful to reintensify the search when a large number of acceptances have occurred without improving the problem makespan, or when the recent *acceptance to generated ratio* (*AG* ratio) becomes lower than a prescribed threshold, indicating that the system is caught in a minimum. The reintensification process replaces the current state (schedule) with the best state found so far, removing the system state from a local minimum if it has become trapped in a basin of attraction. Reintensification also alters the annealing temperature to a more current and appropriate value. A new annealing temperature is calculated from the standard deviation of the functional cost of states in the best neighborhood. If the new resulting temperature is greater than the current temperature, then the current temperature is reset to the new temperature.

7.4 Parameters

Simulated annealing algorithms often require some parameter specific values be determined *a priori*. These annealing scheduling parameters include the initial and final temperatures (T_0, T_f), and the number of annealing steps (k_f). Reasonable values for the reintensification frequency and the *AG* threshold must be chosen as well. For the annealing schedule, appropriate choices of both the initial and final (lower bound) temperatures, and the maximum number of annealing steps, must be determined.

An appropriate temperature reduction function is also needed. Since both inverse logarithmic and inverse linear annealing schedules are too slow for practical consideration, it is useful to apply the exponential annealing schedule given in equation 7.4, with constant c determined by

$$c = -\log(T_f/T_0)/k_f. \tag{7.6}$$

For example, if the annealing schedule is defined so that $T_0 = 1$, $T_f = 10^{-20}$ and $k_f = 10000$, then $c = -\log(10^{-20})/10000 = 0.0046$.

Since scheduling problems have different characteristics, constraints, and differing degrees of difficulty, different annealing schedules must be chosen to fit different problems. Because the initial and final lower bound temperatures are problem dependent parameters, they are difficult to determine *a priori*. By defining these temperatures in terms of the desired *uphill AG* ratios, the temperatures can be determined adaptively from problem independent values. The initial uphill *AG* ratio should be relatively large so that a large number of uphill transitions are accepted.

Later, at the end of the annealing schedule, the final uphill *AG* ratio should be small forcing most schedules having inferior makespans to be rejected.

The adaptive determination of the initial or final temperature can be incorporated in a short *warmup* sequence at the beginning of a simulation. A desired uphill *AG* threshold is chosen *a priori* which is used to adaptively determine the annealing temperatures. The reverse annealing process is implemented by starting at a sufficiently low temperature such that no uphill states are accepted, and increasing the temperature by a small percentage (e.g. 5%) until the actual uphill *AG* ratio is greater than or equal to the desired threshold. This approach is similar to that defined by Aarts [47] (p59), however only *uphill* generated and accepted states determine the *AG* ratio.

For the initial temperature, T_0, an uphill *AG* threshold of 50% of the makespans generated that were larger than the current schedule's makespan was found to be appropriate. For the final (lower bound) temperature, T_f, an uphill *AG* threshold of 0.2% proved most effective. T_f provides a lower bound for the final temperature. This value would only be arrived at if all generated states would be accepted. Realistically this never occurs.

The total number of annealing steps k_f can be chosen empirically, but should be governed by the desires 1) to have reasonable small differences between successive temperatures, and 2) to have non-excessive trial run times, i.e. at most one or four hour per trial, and 3) to generate as many potential schedules as possible within the time limits. Concerning to 1), the acceptance temperature is assumed to be lowered according to equation 7.4 sufficiently slowly such that a *detailed balance* is maintained, and that the resulting distribution of the inhomogeneous Markov chain generated using this schedule between temperatures $T+\delta$ and $T-\delta$, ($\delta \ll 1$) approximates the stationary distribution of a finite length homogeneous Markov chain at temperature T.

If reintensification is to be used, two parameters must be specified to determine when it must be performed. The first parameter, the reintensification frequency R, determines how often reintensification should be performed. Reintensification can be applied after a set number of new schedules are accepted without finding an improved makespan. The second parameter is an *AG* threshold limit. Reintensification is performed when the current *AG* ratio falls below this value.

7.5 Methodology and Results

The performance of the CBSA algorithms was tested by running several simulation trials with and without reintensification. For the reintensification trials, two reintensification frequencies $R = 3,000$ and $R = 10,000$ were tested, both with an AG ratio threshold of 10^{-3}. Although reintensification violates the theoretical ergodicity of simulated annealing by resetting the state of the system, performances were found to be improved when reintensification was incorporated into the system.

Table 7.1 shows the minimum makespans of the first ten trials when the CBSA algorithm (with reintensification every 3,000 acceptances) was initially applied to the mt10 problem. Different random number seeds were used in each trial, resulting in each trial starting from a different randomly generated schedule. The CBSA algorithm was executed for a warm up period to generate new schedules and to gather acceptance rate statistics. The statistics were used to adaptively determine appropriate values for T_0 and T_f. T_f given in equation 7.6 was used to

Run	Min	Evaluations	Generations	Initial Temp	Last Temp	Time
1	⋆930	481429	548175	51.837858	6.194249	38m 0s
2	⋆930	510050	537087	44.833024	10.612196	41m 28s
3	⋆930	507396	579957	47.036501	6.203178	40m 8s
4	⋆930	344341	331749	49.404748	17.333848	28m 45s
5	⋆930	366680	403856	44.760441	7.891267	28m 40s
6	⋆930	459323	472286	47.085840	14.834713	37m 59s
7	⋆930	371984	405170	38.693533	9.067814	29m 8s
8	⋆930	649431	711167	38.756278	9.568809	51m 39s
9	⋆930	316954	352034	36.828203	8.406492	25m 0s
10	938	459372	1000000	54.385611	0.500000	36m 6s

Table 7.1: Ten Trials using the Simulated Annealing Method ($R = 3,000$).

determine reasonable values for c.

The table also shows the number of actual schedules evaluated, the number of new schedules generated, and the cpu time for each of the trials[2].

Since new schedules are often regenerated from the same current schedule, their makespans need not be reevaluated. Hence the actual number of schedules evaluated is always less than or equal to the number of generations. Optimal schedules are indicated by a star $\star$.

Table 7.2 shows the initial and final temperatures of the ten trials. The last temperature of the successful runs shown can vary considerably depending upon when the algorithm terminated. In table 7.2 the algorithm was terminated when an optimal solution was found, or when k was equal to k_f. Since T_f was determined from accepted, rather than generated states, any solution found on or before the last generation k_f will have an actual final temperature larger than the initially determined T_f.

[2]One test simulation performed during the initial programming of the mt10 problem found an optimal schedule in 47 seconds on a Sparcstation 2, however this was not to be representative of other trials.

Run	Initial Temp	Last Temp
1	51.837858	6.194249
2	44.833024	10.612196
3	47.036501	6.203178
4	49.404748	17.333848
5	44.760441	7.891267
6	47.085840	14.834713
7	38.693533	9.067814
8	38.756278	9.568809
9	36.828203	8.406492
10	54.385611	0.5

Table 7.2: Initial and Last Temperatures. Last temperature is the temperature when an optimal makespan was found, or the temperature after 1,000,000 iterations.

Although the last mt10 trial did not find the optimal solution, it did terminate with a makespan of 938. Trial ten's initial temperature, which was adaptively determined, was the highest of all of the trials. Hence it is likely that the system spent excessive amounts of annealing time performing random search at high temperatures. The cpu time and the number of function evaluations performed during the execution of trial ten was quite comparable with those of the successful trials. It is likely that the the system state became caught in a deep local minimum, allowing few if any new states to be explored.

7.5.1 Random Search

The effects of randomly searching the schedule space was investigated to determine if the transition operations were solely responsible for the generation of the high quality schedules shown in table 7.1. Ten trials of the simulated annealing algorithm were performed by setting the initial and final temperatures to large values, i.e. $T_0 = 100.0$ and $T_f = 99.0$. Results of this random search are shown in table 7.3.

Run	Min	Generations	Time	Acceptances
1	994	1000000	1h 41m 16s	844798
2	998	1000000	1h 39m 34s	845644
3	997	1000000	1h 40m 9s	846223
4	999	1000000	1h 39m 13s	847081
5	993	1000000	1h 39m 15s	846005
6	998	1000000	1h 39m 13s	846835
7	992	1000000	1h 39m 30s	845302
8	995	1000000	1h 40m 33s	846163
9	989	1000000	1h 41m 19s	845688
10	1010	1000000	1h 40m 36s	846533

Table 7.3: Ten *High Temperature* Random Trials.

Approximately 84% of all schedules generated were accepted. Schedules with shorter makespans were always accepted, i.e. from equation 7.5, S' is always accepted when $L(S') \leq L(S)$. Since high temperatures, $T_0 \approx T_f \gg 1$, result in a large numbers of inferior schedules being accepted, the method essentially performs like a random search with the critical block transition operators being used to generate the new schedules. Performances of these random searches was noticeably poorer than those in table 7.1.

7.5.2 Low Temperature Greedy Search

Searching with a very low temperature for a small number of generations essentially implements a *greedy* (downhill only) search. When the critical block transition operators are applied with this greedy algorithm, many of the generated schedules of poor quality were observed. Ten thousand schedules were generated by randomly generating initial schedules and applying the CBSA algorithm to them for 1,000 iterations at very low temperatures, i.e. $T_0 = 0.01$, $T_f \approx 0.001$, and $k_f = 1000$. Figure 7.1 shows the a histogram of the 10,000 makespans generated. It is clear from those performances that the low temperature greedy algorithm is not solely responsible for the performances in table 7.1.

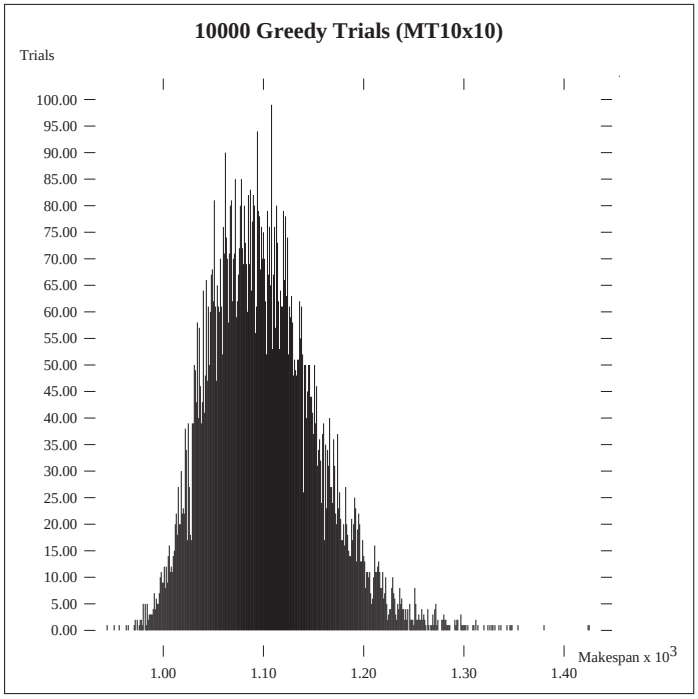

Figure 7.1: Generated Makespans of 10,000 *Greedy* (mt10) Schedules.

Van Laarhoven et. al. [12] describe a similar method called iterative improvement that consists of repeated generation of random schedules using the same neighborhood structure. They tested that method using the previously described Balas transition operator and the best makespan they found over 5 macro runs (averaging 9441.2 trials each) was 1006. In contrast, the best cost makespan found during the ten thousand greedy CBSA trials was 944. The relative difference is indicative of the power of the respective transition operators.

Figure 7.2 shows the time evolution of the makespans of two typical trials with and without reintensification. The abscissa shows the number of schedules generated, and the ordinate shows the makespan differences between the current and optimal schedules. The highly oscillatory behavior of the reintensification trial is due to reintensification.

Annealing Method	$R = 0$		$R = 3,000$		$R = 10,000$	
CBSA min/max	930	945	930	938	930	938
CBSA mean/std.	937.80	4.19	930.80	2.40	933.10	3.81
AESA min/max	938	972	930	951	934	970
AESA mean/std.	951.60	10.20	939.50	5.12	944.40	10.24
GREEDY min/max	971	1491				
GREEDY mean/std.	1171.45	66.15				

Table 7.4: Comparisons between CBSA and AESA.

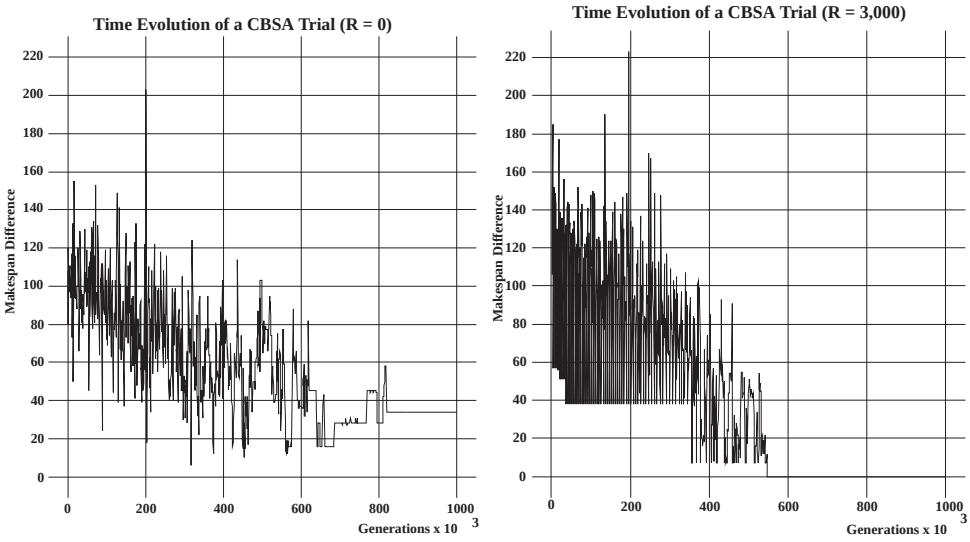

Figure 7.2: Successive makespan differences between the current and optimal solution of the mt10 problem, without reintensification (R=0) and with reintensification ($R = 3,000$).

The power of a reintensification and Critical Block neighborhood structure is shown in table 7.5.2. We show comparative performances of 10 trials of the CBSA and the simulated annealing algorithms using the AE (adjacent exchange) neighborhood (AESA), which was described in Section 6.3, proposed by van Laarhoven et. al. [12] . All performances conditions were identical, except for the reintensification frequencies, (R=0 (no reintensification), R=3,000, and R=10,000), and the neighborhood structure. The performances are best when R=3,000 and the CBSA is used. Not shown are the cpu times which were similar for all runs given.

7.6 Performance on Benchmarks Problems

A set of benchmark problems has been established to judge the effectiveness of algorithms on the JSP. [3]

[3]We are grateful for the benchmark problem set given to us by D. Applegate [48].

PROB	NxM	LB	CBSA	Appl	Laar	Mats	Adams	Bruck
abz7	20x15	655	685	668			710	
abz8	20x15	638	679	687			716	
abz9	20x15	656	701	707			735	
la21	15x10	1046	1050	1053	1063	1071	1084	1059
la24	15x10	935	943	⋆935	952	973	976	935
la25	15x10	977	985	⋆977	992	991	1017	977
la27	20x10	1235	1262	1269	1269	1274	1291	1270
la29	20x10	1130	1188	1195	1203	1196	1239	1202
la38	15x15	1196	1209	1217	1215	1231	1255	1232
la40	15x15	1222	1235	⋆1222	1234	1235	1269	1238

Table 7.5: Ten difficult Benchmark Job Shop Scheduling Problems.

The problem set includes the problems *mt06, mt10,* and *mt20* from [4], and problems *car1-car8* from [49]. Problems *abz5-9* are those given in Adams [14]. Also included is a set of 40 job shop scheduling benchmark problems *la01-la40* originally from [32], The first column in both tables gives the problem name. The next column, NxM, indicates the size of the problem, i.e. N jobs by M machines. The LB column indicates the lower bound of the problem if the optimal makespan is unknown. The *CBSA* column indicates the best makespan found from 5 CBSA trials. (Each trial used a reintensification frequency $R = 3,000$ and was run for one million generations or until the known optimal minimum makespan or lower bound was found.) The column headings *Appl, Laar, Mats, Adams* and *Bruck* indicate the best performances of Applegate [1], Van Laarhoven [12], Matsuo [13], Adams [14], and Brucker [9] respectively. A single star, ⋆, indicates the optimum or best known minimum. Table 7.6 shows the performances of the CBSA and some of the best known job shop algorithms on the *la* problem test set.

PROB	NxM	LB	CBSA	Appl	Laar	Mats	Adams	Bruck
la01	10x5	—	⋆666	⋆666	666	—	666	666
la02	10x5	—	⋆655	⋆655	655	655	669	655
la03	10x5	—	⋆597	⋆597	606	597	605	597
la04	10x5	—	⋆590	⋆590	590	590	593	590
la05	10x5	—	⋆593	⋆593	593	—	593	593
la06	15x5	—	⋆926	⋆926	926	—	926	926
la07	15x5	—	⋆890	⋆890	890	—	890	890
la08	15x5	—	⋆863	⋆863	863	863	863	863
la09	15x5	—	⋆951	⋆951	951	—	951	951
la10	15x5	—	⋆958	⋆958	958	—	958	958
la11	20x5	—	⋆1222	⋆1222	1222	—	1222	1222
la12	20x5	—	⋆1039	⋆1039	1039	—	1239	1039
la13	20x5	—	⋆1150	⋆1150	1150	—	1150	1150
la14	20x5	—	⋆1292	⋆1292	1292	—	1292	1292
la15	20x5	—	⋆1207	⋆1207	1207	—	1207	1207
la16	10x10	—	⋆945	⋆945	956	959	978	945
la17	10x10	—	⋆784	⋆784	784	784	787	784
la18	10x10	—	⋆848	⋆848	861	848	859	848
la19	10x10	—	⋆842	⋆842	848	842	860	842
la20	10x10	—	907	⋆902	902	907	914	902
la21	15x10	1046	1050	1053	1063	1071	1084	1059
la22	15x10	—	935	⋆927	938	927	944	927
la23	15x10	—	⋆1032	⋆1032	1032	1032	1032	1032
la24	15x10	935	943	⋆935	952	973	976	935
la25	15x10	977	985	⋆977	992	991	1017	977
la26	20x10	—	⋆1218	⋆1218	1218	1218	1224	1218
la27	20x10	1235	1262	1269	1269	1274	1291	1270
la28	20x10	—	⋆1216	⋆1216	1224	1216	1250	1276
la29	20x10	1130	1188	1195	1203	1196	1239	1202
la30	20x10	—	⋆1355	⋆1355	1355	1355	1355	1355
la31	30x10	—	⋆1784	⋆1784	1784	—	1784	1784
la32	30x10	—	⋆1850	⋆1850	1850	—	1850	1850
la33	30x10	—	⋆1719	⋆1719	1719	—	1719	1719
la34	30x10	—	⋆1721	⋆1721	1721	—	1721	1721
la35	30x10	—	⋆1888	⋆1888	1888	—	1888	1888
la36	15x15	—	1291	⋆1268	1293	1292	1305	1268
la37	15x15	—	1420	⋆1397	1433	1435	1423	1424
la38	15x15	1196	1209	1209	1215	1231	1255	1232
la39	15x15	—	1243	⋆1233	1248	1251	1273	1233
la40	15x15	1222	1235	⋆1222	1234	1235	1269	1238

Table 7.6: Performances of the 40 Benchmark Job Shop Scheduling Problems.

7.7 Concluding Remarks

According to Aarts [47], The success of an approximation algorithm depends on a number of aspects including performances, ease of implementation, and applicability and flexibility. It is clear that the SA algorithm is very simple and easy to implement, taking only a few hundred lines of code to implement. Regarding flexibility, we found that our SA code could be used to implement both Aarts SA approach and the CBSA method by changing only the call to the neighborhood generation procedure.

Matsuo [13] has indicated that Aarts SA method has very few adjacent pairs that improve the makespan by exactly one interchange (Aarts SA transition). Compared with Aarts neighborhood structure, the CB neighborhood contains more (or at least the same number of) transitions that can immediately improve the a schedules makespan by exactly one transition.

Undoubtedly new and more powerful approaches will be developed to solve the JSP, methods having better performance and convergence characteristics then the methods described here. Unlike SA methods, other approximation approaches have no theoretical guarantee that they will converge to to an optimal solution.

Chapter 8

Critical Block Simulated Annealing with Shifting Bottleneck Heuristics

In this chapter, we consider to improve the CBSA described in the previous chapter in two aspects. One is to employ a new neighborhood by incorporating the notion of active schedule described in Section 2.2 and the other is to combine with a deterministic heuristic called "shifting bottleneck" described in Section 2.6.

8.1 Active Critical Block Simulated Annealing

As explained in the previous chapter, a solution obtained from a before or an after candidate is not necessarily executable. In the following, we propose a new neighborhood by modifying the CB neighborhood. Each element in the new neighborhood is not only executable, but also active and structurally close to the corresponding member in the original CB neighborhood. Let S be an active schedule and $B_\gamma^{\alpha,\beta}$ be a critical block of S on a machine M_γ, where the first and the last operations of $B_\gamma^{\alpha,\beta}$ are the α-th and the β-th operations on M_γ respectively. Let O_γ^λ be a "moving" operation that is the λ-th operation on M_γ such that $\alpha < \lambda < \beta$ (i.e., O_γ^λ is inside $B_\gamma^{\alpha,\beta}$). We consider to generate an active schedule $S_\gamma^{\lambda,\alpha}$ (or $S_\gamma^{\lambda,\beta}$) by moving O_γ^λ into position α (or position β). If the resulting schedule is active, then we will use it. Otherwise, we try to find an alternate position that is as close to α (or β) as possible and is inside $B_\gamma^{\alpha,\beta}$ such that the resulting schedule becomes active and use it instead. This can be done by adopting the GT algorithm described in Algorithm 2.2.1 and modifying Step 4 of the algorithm, where the choice of the next operation from the conflict set $C[M_r, i]$ was at random. Here, the choice is made by looking at the processing order in S. An operation that is in $C[M_r, i]$ and that was scheduled earliest in S is selected when the operation is located outside $B_\gamma^{\alpha,\beta}$ (i.e., $i \notin [\alpha, \beta]$). This way, the processing order of operations in $S_\gamma^{\lambda,\alpha}$ or $S_\gamma^{\lambda,\beta}$ is kept mostly unchanged from that in S outside $B_\gamma^{\alpha,\beta}$. However inside $B_\gamma^{\alpha,\beta}$ instead, the "moving" operation O_γ^λ must be chosen from the conflict set with the top most priority when generating $S_\gamma^{\lambda,\alpha}$ and with the bottom least priority when generating $S_\gamma^{\lambda,\beta}$. The details are described in Algorithm 8.1.1. In the algorithm, if we are generating $S_\gamma^{\lambda,\alpha}$ (i.e., $d = f$) and when $i = \alpha$, O_γ^λ should be chosen from $C[M_r, i]$ as soon as i becomes equal to α. However

at that point, O_γ^λ may not yet exist in $C[M_r, i]$. This means that if we move O_γ^λ to the front position α ignoring the fact that $O_\gamma^\lambda \notin C[M_r, \alpha]$, then the resulting schedule becomes non-active, or may even become infeasible. Hence, we have to "pass" this time and wait until O_γ^λ appeares in $C[M_r, i]$. Because S is an active schedule, it is guaranteed that O_γ^λ appeares in $C[M_r, i]$ at the latest when $i = \lambda$, in which case resulting schedule becomes identical to S. If we are generating $S_\gamma^{\lambda,\beta}$ instead (i.e., $d = r$), then O_γ^λ should be moved to the rear postion β, in other words, O_γ^λ should be chosen from $C[M_r, i]$ only when i becomes equal to β and not while $i < \beta$. In fact, O_γ^λ is guaranteed to appear in $C[M_r, i]$ when $i = \beta$ at the latest because S is active. However O_γ^λ may become the only element of $C[M_r, i]$ and then it is unavoidable to choose O_γ^λ even when $i < \beta$. This means that if we move O_γ^λ to the rear position β ignoring the fact that $O_\gamma^\lambda \notin C[M_r, \beta]$, then the resulting schedule becomes non-active, or may be even infeasible. Hence we have to choose O_γ^λ even when $i < \beta$. The new neighborhood $ACB(S)$ is now defined as a set of all $S_\gamma^{\lambda,\alpha}$ and $S_\gamma^{\lambda,\beta}$ over all critical blocks:

$$ACB(S) = \left(\bigcup_{B_\gamma^{\alpha,\beta}} \{S_\gamma^{\lambda,\alpha}\}_{\alpha<\lambda<\beta} \cup \{S_\gamma^{\lambda,\beta}\}_{\alpha<\lambda<\beta} \right) \setminus \{S\}. \tag{8.1}$$

Once we have the neighborhood structure $ACB(S)$ defined, the basic framework of the simulated annealing algorithm described in Algorithm 7.2.1 can be applied without major modifications by using $ACB(S)$ in place of $CB(S)$ in Step 2(a)i of Algorithm 7.2.1.

In Algorithm 7.2.1, if the acceptance probabilities are low for all members in $ACB(S)$, the system will remain trapped in a local minimum S and it will take a long time to move to a new state. The algorithm may stay in S even after all members are selected in Step 2(a)i and evaluated in Step 2(a)ii. To avoid this, relative acceptance probability defined by (8.1)

$$\overline{P(S_i)} = \frac{P(S_i)}{\sum_{S_j \in ACB(S)} P(S_j)} \qquad \text{for each } S_i \in ACB(S) \tag{8.2}$$

will be introduced after all members in $ACB(S)$ are visited and evaluated without being chosen. The member S_i in $ACB(S)$ is then randomly selected in proportion to $\overline{P}(S_i)$, and the system moves unconditionally to S_i. This modification is effective when, like $ACB(S)$, the neighborhood size is limited. Therefore, Algorithm 7.2.1 is modified as in Algorithm 8.1.2.

8.2 Active CBSA Enhanced by Shifting Bottleneck

As described in Section 2.6, Shifting bottleneck (SB) proposed by [14] is a powerful heuristic method for solving a JSP. Here we consider to incorporate the local optimization procedure used in the SBI heuristic described in Algorithm 2.6.1 into the active CBSA described in the previous section.

SBI is a constructive method that generates a complete schedule from scratch. Modifying the method is necessary in order to refine a certain complete schedule for improvement. The *BottleRepair* shown in Algorithm 8.2.1 describes an iterative version of the basic SB. The reoptimization process used here is the same as used in Algorithm 2.6.1. The basic idea of *BottleRepair*

Algorithm 8.1.1 (An algorithm to generate $S_\gamma^{\lambda,\alpha}$ or $S_\gamma^{\lambda,\beta}$ from S)

A scheduling problem is given as in Algorithm 2.2.1. An original active schedule S of the problem and a moving operation O_γ^λ on a critical block $B_\gamma^{\alpha,\beta}$ of S is given, where $\alpha < \lambda < \beta$. The direction $d = f$ (front, when generating $S_\gamma^{\lambda,\alpha}$) or $d = r$ (rear, when generating $S_\gamma^{\lambda,\beta}$) is given.

1. Initialize G (as in Step 1 of Algorithm 2.2.1). Initialize S' as an empty schedule.
2. Find the earliest completable operation $O_{*r} \in G$ (as in Step 2 of Algorithm 2.2.1).
3. Calculate the conflict set $C[M_r, i] \subset G_r$ (as in Step 3 of Algorithm 2.2.1).
4. **If** $r \neq \lambda$, then select and schedule from $C[M_r, i]$ the earliest operation O_{kr} in S as the i-th operation on M_r in S', i.e., $S'_{ri} := k$, where k is the job number of O_{kr}.

   If $r = \lambda$,

   - If $d = f$ (i.e., generating $S_\gamma^{\lambda,\alpha}$),
     - If $\alpha \leq i \leq \lambda$ and $O_\gamma^\lambda \in C[M_r, i]$, then select and schedule O_γ^λ as the i-th operation on M_r in S.
     - Otherwise, select and schedule from $C[M_r, i]$ the earliest operation in S as the i-th operation on M_r in S'.
   - If $d = r$ (i.e., generating $S_\gamma^{\lambda,\beta}$),
     - If $\lambda \leq i \leq \beta$ and $C[M_r, i]$ contains any operation other than O_γ^λ, then select and schedule from $C[M_r, i] \setminus \{O_\gamma^\lambda\}$ the earliest operation in S as the i-th operation on M_r in S'.
     - If $C[M_r, i] = \{O_\gamma^\lambda\}$ or $i = \beta$ and $O_\gamma^\lambda \in C[M_r, i]$, then select and schedule O_γ^λ as the i-th operation on M_r in S'.
     - Otherwise, select and schedule from $C[M_r, i]$ the earliest operation in S as the i-th operation on M_r in S'.

   Let O_{kr} be the operation selected above, where k is the job number of O_{kr}, then $S'_{ri} := k$.
5. Do Step 6 and Step 7 of Algorithm 2.2.1.
6. Repeat from Step 1 to Step 5 until all operations are scheduled.
7. Output the active schedule S' as $S_\gamma^{\lambda,\alpha}$ if $d = f$, or as $S_\gamma^{\lambda,\beta}$ if $d = r$.

Algorithm 8.1.2 (The Active Critical Block Simulated Annealing)

We are given a jobshop scheduling problem to optimize, and the initial temperature T_0.

1. Set $S = S_0$ a random initial active schedule, generated by the GT algorithm in Algorithm 2.2.1. Set $k = 0$, and $T_{k=0} = T_0$.

2. **do**

   (a) Set N be the size of $ACB(S)$ and set $n = 0$, the number of elements in $ACB(S)$ that are already evaluated.

   (b) **do**

      i. Pick S_i from $ACB(S)$ randomly and if S_i is first time to be picked and not yet evaluated, then evaluate S_i by calculating $C_{max}(S_i)$ and set $n = n + 1$.

      ii. Accept S_i with probability:

$$P(S_i) = \begin{cases} 1 & \text{if } C_{max}(S_i) \leq C_{max}(S) \\ e^{-(C_{max}(S_i)-C_{max}(S))/T} & \text{otherwise.} \end{cases}$$

      until S_i is accepted or $n = N$.

   (c)
   - **If** S_i is accepted, then set $S = S_i$ and $n = 0$.
   - **Otherwise** (i.e., if $n = N$) select S' from $ACB(S)$ in proportion to the probability

$$\overline{P(S_i)} = \frac{P(S_i)}{\sum_{S_j \in ACB(S)} P(S_j)}$$

     and set $S = S_i$ and $n = 0$.

   (d) Set $k = k + 1$ and decrease T_k according to the annealing schedule.

   until termination (i.e. T is sufficiently small or k is sufficiently large or the minimal makespan or lower bound is found).

3. Print the best schedule found and its makespan.

Algorithm 8.2.1 (*BottleRepair*: Iterative SB)

A complete selection S is given as an input. Let $\mathcal{M}$ be a set of all machines: $\mathcal{M} = \{M_1, \ldots, M_r\}$.

1. Let $\mathcal{M}_0 = \{M_1, M_2, \ldots, M_l\}$ be a set of all the critical machines (machines that contain parts of critical path in S).
2. Reset all sequences of all machines in $\mathcal{M} \setminus \mathcal{M}_0$ and make the machines unsequenced. Let S_p be a partial selection obtained from S by this resetting.
3. Reoptimize all sequenced machines in $\mathcal{M}_0$ by applying Step 6 of Algorithm 2.6.1 with $\mathcal{M}_0$ and S_p obtained above and obtain new S_p.
4. Do Step 2 to Step 8 of Algorithm 2.6.1 with $\mathcal{M}_0$ and S_p obtained above and obtain a complete selection.
5. Output the complete selection as an obtained schedule.

Algorithm 8.2.2 (SB incorporation for Active CBSA)

2(b)iii. If S_i is rejected, apply *BottleRepair* to S_i and obtain S_i^*.

Accept S_i^* and set $S_i = S_i^*$ *if* $C_{max}(S_i^*) < C_{max}(S)$.

comes from the original paper of SB [14] where the last α noncritical machines are temporarily removed for the reoptimization.

As shown in Algorithm 8.1.2, S_i is selected from $ACB(S)$ and is probabilistically accepted. *BottleRepair* is applied to S_i only when S_i is rejected. The resulting schedule S_i^* is accepted if its makespan is shorter than that of S. To summarize, Step **2(b)iii** as defined in Algorithm8.2.2 is added to Algorithm 8.1.2 just after Step 2(b)ii.

BottleRepair gives a systematic way to inspect the schedule's critical path and permutes operations again and again by repeatedly solving one machine problems in a deterministic manner. If it generates an improved schedule S' from S, the critical path of S' becomes different from S and the difference is much greater than that between S and its active CBSA neighbor. On the other hand, active CBSA gives a stochastic more focused local search around the current critical path. The proposed integration of active CBSA and SB is expected to have the synergistic effect as: SB gives a long jump to active CBSA so that it can omit many time-consuming inferior transitions and active CBSA adds stochastic perturbations to SB so that it can escape from the local minima.

8.3 Experimental Results

8.3.1 Muth and Thompson's Benchmark

A 10×10 problem (mt10) and 20×5 problem (mt20) formulated by [4] (*MT benchmarks*) are well known benchmark JSPs. CBSA with and without SB modification was evaluated using these problems. Table 8.1 shows the results of 20 trials with different random number seeds on a SUN SPARC station 10. All programs are written in the C language.

Table 8.1: Comparisons between CBSA and CBSA+SB using MT benchmarks

Prob	$n \times m$	CBSA ($R = 6,000$)				CBSA+SB			
		best	mean	std	BT	best	mean	std	BT
mt10	10×10	930	933.65	4.04	190	930	932.45	3.01	786
mt20	20×5	1178	1179.45	1.94	235	1165	1165.00	0.00	449

std: standard deviation
BT: average cpu time (sec.) to find the best solution

Results for the mt10 problem using CBSA without SB show that the optimal solutions of $L = 930$ were found in 11 trials. The average cpu time was 3 min. 10 sec., and the fastest was 1 min. 21 sec. Although the solutions of $L = 930$ were found in only half of the trials, the cpu time in successful runs were satisfactorily short. If the temperature is more slowly lowered, though it takes longer, the rate of finding optimal solutions will become higher as in [50]. CBSA without SB could not find any optimal solution for mt20 problem. In most cases, solutions of $L = 1178$ were found instead of the optimal $L = 1165$.

The results for the mt10 problem using active CBSA with SB modification show that the number of trials finding optimum solutions increased slightly, but the average cpu time increased about four times. This fact indicates that CBSA without SB is powerful enough to solve mt10 problem. On the other hand, all 20 trials with SB modification for mt20 problem found the optimal solutions of $L = 1165$ in an average cpu time of 7 min. 29 sec., and 1 min. 22 sec. was the best time. The effect of SB enhancement is obvious from this problem. Reintensification did not work well because the optimal or near optimal solutions were obtained at an early stage of the search.

8.3.2 Other Benchmarks

Results in the previous section indicate that if CBSA without SB can solve a problem skillfully, applying CBSA+SB has no advantage. However, if CBSA fails to work well, CBSA+SB may improve solution quality and compensate for the extra cpu time needed with SB enhancement. A set of benchmark problems has been established to evaluate different algorithms for JSPs. Table 8.2 shows the makespan performances of CBSA+SB and various other algorithms for the ten difficult benchmark JSPs. All experiments of CBSA+SB runs were done on a HP 730 (HP 730 is about 1.5 times faster than SUN SPARC station 10).

The LB column indicates the theoretical lower bound of the problem if the optimal makespan is unknown. The CBSA+SB column indicates the best makespans found from ten trials. Each trial used a reintensification frequency of $R = 1,000$ and ran for three hours or until the known optimal makespan or lower bound was found. The column headings Aart, Matt, Appl and Tail indicate the best performances of those in [51], [52], [1] and [17] respectively.

Table 8.2: Results of 10 tough JSPs

Prob	$n \times m$	LB	CBSA+SB				Aart	Matt	Appl	Tail
			best	mean	std.	BT				
abz7	20×15	654	665	671.0	3.92	7814	668	672	668	665
abz8	20×15	635	675	680.0	3.13	8775	670	683	687	676
abz9	20×15	656	686	698.6	7.42	8749	691	703	707	691
la21	15×10	1040	1046	1049.3	3.32	361	1053	1053	1053	1047
la24	15×10	–	935	939.2	1.99	6226	935	938	935	
la25	20×10	–	977	979.3	1.62	4117	983	977	977	
la27	20×10	1235	1235	1242.4	6.15	7805	1249	1236	1269	1240
la29	20×10	1120	1154	1162.4	7.10	5434	1185	1184	1195	1170
la38	15×15	1184	1198	1206.8	4.53	3479	1208	1201	1209	1202
la40	15×15	–	1228	1230.2	2.32	3331	1225	1228	1222	

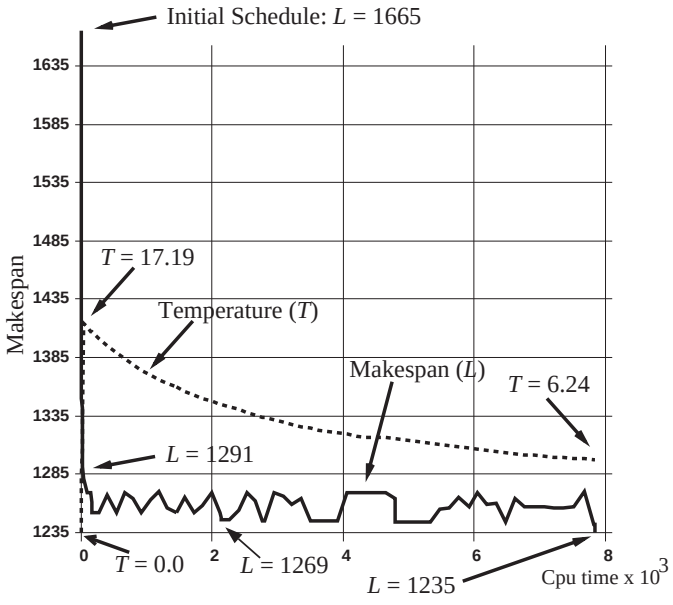

Figure 8.1: The time evolution of CBSA+SB trial for the la27 problem

Figure 8.1 shows the time evolution of the makespan (L) and temperature (T) of the best trial of CBSA+SB for the la27 problem. The abscissa shows the cpu time in seconds, and the

Table 8.3: An optimal solution of la27 problem

m	Job sequences on each machine																			
1	18	9	4	2	11	3	12	8	10	7	13	15	20	1	6	19	5	14	17	16
2	14	9	8	12	3	17	15	4	10	13	6	7	11	19	1	2	18	16	20	5
3	11	14	3	19	10	4	9	5	6	2	16	17	13	15	7	20	18	12	8	1
4	6	11	10	1	19	3	8	13	15	18	2	12	14	17	4	7	9	16	20	5
5	12	3	5	6	20	13	11	8	1	17	9	19	7	14	2	16	18	15	10	4
6	19	18	11	8	14	7	3	16	2	17	1	10	9	5	13	6	20	4	12	15
7	17	2	13	15	9	8	19	7	6	20	10	18	14	16	5	4	3	1	11	12
8	2	12	16	14	11	10	3	5	19	20	6	7	13	17	8	18	15	1	4	9
9	18	12	3	14	11	10	16	8	19	13	4	6	15	2	17	20	1	9	7	5
10	13	20	18	9	4	5	11	6	3	8	2	19	10	14	1	17	7	15	16	12

m: machine

solid and dotted lines show the makespan and the temperature respectively. Starting from the makespan value $L = 1665$, it rapidly decreases to $L = 1291$ during the first warmup interval. After twelve times of reintensification, it finally reached the optimal value $L = 1235$. In this experiment, 56059 schedules were generated and 8850 schedules accepted. About 25% of the accepted schedules were accepted by *BottleRepair* to add CBSA long jumps and the rest served as stochastic perturbations to *BottleRepair*. The oscillatory behavior is due to reintensification.

Although CBSA+SB outperformed other methods most of the cases in Table 8.2, the required computational time is much longer. For example, [53] reported that Applegate's method in the Appl colum found a schedule of $L = 1269$ in 604.2 sec. on SUN SPARC station ELC which is about 10 times slower than HP 730. This is because each CBSA+SB experiment includes a lot of unsuccessful *BottleRepair* trials. But this gap can be filled to some extent by the fact that in the same experiment, CBSA+SB passed a point $L = 1269$ in 164 sec.

8.4 Concluding Remarks

The proposed method CBSA+SB is an improved CBSA enhanced by integrating with a problem specific method called shifting bottleneck. The performance of CBSA+SB was evaluated using difficult benchmark problems. The results show that for eight problems of ten difficult benchmark problems, CBSA+SB could find schedules better than or equal to the best schedules published so far in the literature, when enough computational time is given. A new solution of $L = 1235$ was found for the la27 problem; it is optimal because the value equals the theoretical lower bound. Further research is necessary to reduce the computational time.

Chapter 9

Scheduling by Genetic Local Search with Multi-Step Crossover Fusion

As we have seen in Chapter 5, Genetic Algorithms can be applied to the job-shop scheduling problem with good success. However, the well-known fact that GAs are, in general, not well suited for fine-tuning structures which are very close to optimal solutions also applies to this case and obstructs further improvements. The general remedy to this problem is to incorporate local search methods, such as neighborhood search described in Chapter 6, into GAs. The result of such 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" (initial solution) for the subsequent local search. The local search moves the offspring from its current point to the nearest locally optimal point, which is included in the next generation.

In solving combinatorial optimization problems, it is often difficult to construct an efficient crossover operator, because a crossover operator that "recombines" solutions, allows to cause global changes that alters the structure of a solution in large parts and therefore may violate the constraints of the target problem, resulting in generating many infeasible solutions.

On the other hand, a neighborhood search operator which modifies a solution only locally is rather easier to construct as we have seen in Chapter 7 and Chapter 8. A simple neighborhood search operator exchanges a pair of consecutive jobs in a job sequence, another operator removes a job from its original position and re-insert it in another position on the same job sequence. As we have seen in the previous chapters, they should be improved by focusing on the jobs on the critical path. Unfortunately the same method cannot be applied directly to construct an effective crossover operator.

9.1 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) [54]. Let any two individuals be x and z. An individual y is called *intermediate* between

Algorithm 9.1.1 (Multi-Step Crossover Fusion — MSXF)

Parent individuals p_0, p_1 are given as inputs. The distance measure d is defined. For a given individual $\mathbf{x}$, $f(\mathbf{x})$ is the function to be minimized.

1. Initialize both current solution $\mathbf{x}$ and output q as p_0; $\mathbf{x} = q = p_0$.
2. **do**
   (a) For each member $\mathbf{y_i} \in N(\mathbf{x})$, estimate $d(\mathbf{y_i}, p_1)$.
   (b) Sort $\mathbf{y_i} \in N(\mathbf{x})$ in ascending order of the $d(\mathbf{y_i}, p_1)$ estimation.
   do
      i. Select $\mathbf{y_i}$ from $N(\mathbf{x})$ randomly, but with a bias in favor of $\mathbf{y_i}$ with a small index i.
      ii. Evaluate $\mathbf{y_i}$ and obtain $f(\mathbf{y_i})$ if $\mathbf{y_i}$ has not yet been evaluated.
      iii. Accept $\mathbf{y_i}$ with probability one if $f(\mathbf{y_i}) \leq f(\mathbf{x})$, and with $P_c(\mathbf{y_i})$ otherwise.
      iv. (optional) Change the index of y_i from i to n, and the indexes of $\mathbf{y_k}$ ($k \in \{i+1, i+2, \ldots, n\}$) from k to $k-1$.
   until $\mathbf{y_i}$ is accepted.
   (c) Set $\mathbf{x} = \mathbf{y_i}$.
   (d) If $f(\mathbf{x}) < f(q)$ then set $q = \mathbf{x}$.

   until some termination condition is satisfied.
3. Output q as an individual for the next generation.

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.

In this chapter, we extend the idea of the NSX to make it applicable to more complicated problems such as job-shop scheduling and propose the Multi-Step Crossover Fusion (MSXF): a new crossover operator with a built-in local search functionality [55, 56, 57]. The MSXF has the following characteristics compared to the NSX.

- It can handle more general class of representations (i.e., it is not limited to the binary representation) and neighborhood structures.
- It is based on a stochastic local search algorithm.
- Instead of restricting the neighborhood by the intermediateness, a biased stochastic replacement method 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 JSP, it would be unrealistic to apply SA repeatedly in a GA run that would consume too much time. In such a case, a restricted method with a fixed temperature parameter $T = c$ is a good alternative. Accordingly, the acceptance probability defined in (7.1) is modified as:

$$P(\mathbf{x}') = \begin{cases} 1 & \text{if} \quad f(\mathbf{x}') \leq f(\mathbf{x}) \\ e^{(-f(\mathbf{x}')-f(\mathbf{x}))/c} & \text{otherwise.} \end{cases} \tag{9.1}$$

Let parent individuals be p_0 and p_1, and let the distance between any two individuals $\mathbf{x}$ and $\mathbf{y}$ in any representation be $d(\mathbf{x},\mathbf{y})$. If $\mathbf{x}$ and $\mathbf{y}$ are schedules, then $d(\mathbf{x},\mathbf{y})$ is the DG distance defined in Section 2.4. Let f be the function to be minimized. In the case of the scheduling problem, f is the makespan C_{max}. The basic idea to incorporate crossover functionality into the neighborhood search described in Algorithm 6.1.1 is to set the initial starting point as one of the parent: $\mathbf{x_0} = p_0$ and when choosing a candidate y from $N(\mathbf{x})$, give a greater acceptance bias for $\mathbf{y}$ with small $d(\mathbf{y}, p_1)$, which we call biased stochastic replacement. This bias control in the MSXF is achieved easily by sorting $N(\mathbf{x})$ members in ascending order of $d(\mathbf{y_i}, p_1)$ so that $\mathbf{y_i}$ with a smaller index i has a smaller distance $d(\mathbf{y_i}, p_1)$. The calculation of $d(\mathbf{y_i}, p_1)$ is not an expensive task if $d(\mathbf{x}, p_1)$ and the the nature of the move from $\mathbf{x}$ to $\mathbf{y_i}$ are known; it is not necessary to actually generate and evaluate $\mathbf{y_i}$. Then $\mathbf{y_i}$ is chosen from $N(\mathbf{x})$ randomly, but with a bias for $\mathbf{y_i}$ with small index i. The outline of the MSXF is described in Algorithm 9.1.1.

In place of $d(\mathbf{y_i}, p_1)$, one can also use $sign(d(\mathbf{y_i}, p_1) - d(\mathbf{x}, p_1)) + r_\epsilon$ to sort $N(\mathbf{x})$ members in Algorithm 9.1.1. 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 $N(\mathbf{x})$ members in descending order of $d(\mathbf{y_i}, p_1)$ in Algorithm 9.1.1.

9.2 Scheduling in the reversed order

The GT algorithm in Algorithm 2.2.1 and all its variants determine the job sequences from left to right in temporal order. This is because active schedules are defined to have no extra idle 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 JSP 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 9.1: A simple 2×2 problem

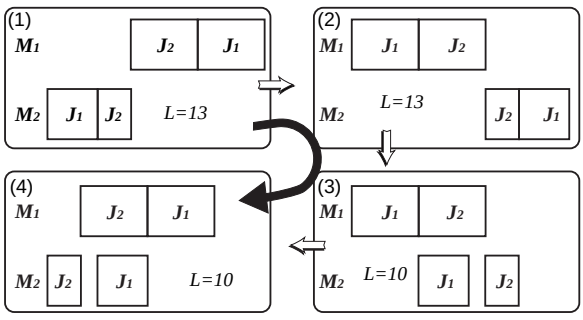

Figure 9.2: Schedule reversal and activation

For example, the simple 2×2 problem described in Table 9.1 is considered. Figure 9.2(1) shows a solution of this problem, which is active and no more left shifts can improve its makespan.

Figure 9.2(2), obtained by reversing Figure 9.2(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 9.2(3). Finally Figure 9.2(4) is obtained by reversing Figure 9.2(3) again, which is optimal. As things turn out, Figure 9.2(1) is improved by moving job 1 *posterior* to job 2 on machine 2, resulting in Figure 9.2(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.

9.3 MSXF-GA for Job-shop scheduling

The MSXF is applied to the JSP by using the active CB neighborhood and the DG distance defined in Section 2.4. Algorithm 9.3.1 describes the outline of the MSXF-GA routine for the JSP using the steady state model proposed in [58, 59]. 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 2d.

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.

Figure 9.3 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.

9.4 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 [4] 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

Algorithm 9.3.1 (MSXF-GA for the JSP)

1. Initialize population: randomly generate a set of *left* and *right* active schedules in equal number and apply the local search to each of them.
2. **do**
   (a) Randomly select two schedules p_0, p_1 from the population with some bias depending on their makespan values.
   (b) Change the direction (*left* or *right*) of p_1 by reversing the job sequences with probability P_r.
   (c) Do step 2(c)i with probability P_c, or otherwise do Step 2(c)ii.
      i. **If** the DG distance between p_1, p_2 is shorter than some predefined small value d_{min}, 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.
      ii. Apply Algorithm 6.1.1 with acceptance probability given by (9.1) and the active CB neighborhood.
   (d) 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.
3. Output the best schedule in the population.

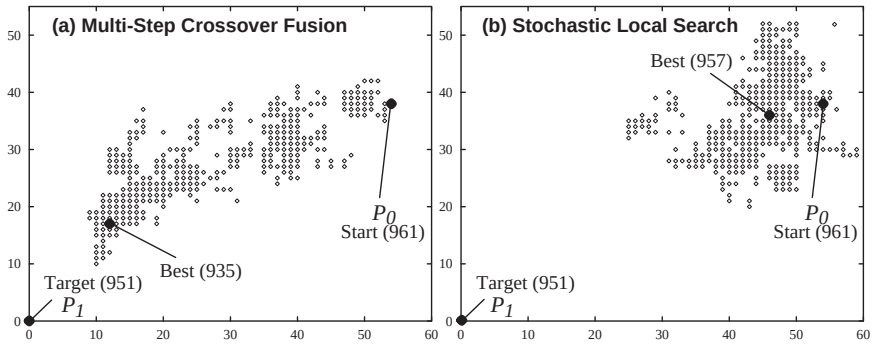

Figure 9.3: Distribution of solutions generated by an application of (a) MSXF and (b) a short-term stochastic local search

Table 9.1: Performance comparison using the MT benchmark problems

1963	Muth-Thompson	Test problems	10×10	20×5
1991	Nakano/Yamada	Simple 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	Subsequence Exchange Crossover	930	1178
	/Yamamura	SXX-GA		
1995	Bierwirth	Generalized-Permutation GP-GA	936	1181
1996	Yamada/Nakano	Multi-step Crossover Fusion MSXF-GA	930	1165

challenge.

Applegate and Cook proposed a set of benchmark problems called the “ten tough problems” as a more difficult computational challenge than the mt10 problem, by collecting difficult problems from literature, some of which still remain unsolved [1].

9.4.1 Muth and Thompson benchmark

Table 9.1 summarizes the makespan performance of the methods described in this chapter. The Simple GA described in Chapter 4 has only limited success. 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 (a GA with Subsequence Exchange Crossover described in Section 5.1) is improved over the GT-GA described in Section 5.4 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 Sparcstation 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.

9.4.2 The Ten Tough Benchmark Problems

Table 9.2 shows the makespan performance statistics of the MSXF-GA for the ten difficult benchmark problems proposed in [1]. 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 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 [60]. 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 9.3 shows the performance comparisons with various heuristic methods for the 10 tough problems. The column headings Nowi and Dell indicate the best performances of TABU search proposed in [19] and [18]. CBSA+SB indicates SA results described in Chapter 8. Aart, Matt, Appl indicate SA results proposed in [51], GA results in [52] and results in [1] respectively. It is interesting to observe from those results that the two approaches CBSA+SB are mutually complementary. For the problem instances abz7, abz8, abz9 and la29, SA+SB outperforms MSXF-GA. In fact, SA+SB performs the best for these problems. However, for the problem instance la38 for which SA+SB fails to find the global optimum, MSXF-GA successfully and frequently find the global optimum. MSXF-GA also shows reasonably good performances for the problem instance la40 for which SA+SB performs relatively poorly.

Table 9.2: 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 9.4 shows a performance comparison of GLS with and without MSXF using the la38 problem. A total of 100 experiments (runs) were performed for each under the same conditions used in Table 9.2 but with different random seeds. In the figure, the solid line gives the results of MSXF-GA (in other words, GLS with MSXF) and the dotted line gives the equivalent results of GLS without MSXF (i.e., in place of MSXF, CPU equivalent short-term stochastic local search is used). Each of the 100 runs is numbered from No.1 to No.100 in ascending order of cpu time at which each run is terminated. For example, run No.1 successfully found the optimal schedule and was terminated the most quickly. The cpu time value = 2400 means that the run was terminated before it found the optimal schedule. In the figure, instead of standard time evolution graph, the x axis represents run numbers and the y axis represents the cpu time. The

Table 9.3: Performance comparisons with various heuristic methods on the 10 tough problems

prob	our bst	Nowi	Dell	CBSA+SB	Aarts	Matt	Appl
abz7	678	–	667	665	668	672	668
abz8	686	–	678	675	670	683	687
abz9	697	–	692	686	691	703	707
la21	⋆1046	1047	1048	⋆1046	1053	1053	1053
la24	⋆935	939	941	⋆935	⋆935	938	⋆935
la25	⋆977	⋆977	979	⋆977	983	⋆977	⋆977
la27	⋆1235	1236	1242	⋆1235	1249	1236	1269
la29	1166	1160	1182	1154	1185	1184	1195
la38	⋆1196	⋆1196	1203	1198	1208	1201	1209
la40	1224	1229	1233	1228	1225	1228	⋆1222

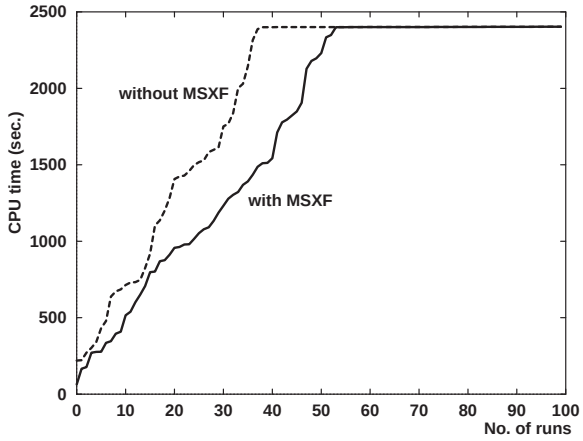

Figure 9.4: Performance comparison using the la38 15×15 problem

fact that the solid line increases slower and has shorter horizontal tail part than the dotted line means that the experiments with MSXF outperforms those without MSXF both in terms of the cpu time and in the number of successful runs.

Chapter 10

Permutation Flowshop Scheduling by Genetic Local Search

So far, we have mainly considered the jobshop scheduling problem (JSP), but hereafter, we will shift our focus to the permutation flowshop scheduling problem, abbreviated as PFSP or just FSP, which is a special case of the JSP in a sense that the technological sequence of machines is the same for all jobs and the order in which each machine processes the jobs is also same for all machines. A solution schedule is then represented by a permutation of n jobs, instead of m permutations of n jobs. We do not have the concept of active schedule, but still we have critical path and blocks, if the objective is C_{max} but not if the objective is C_{sum}, the sum of the completion times of all the operations. In this chapter, the MSXF method described in the previous chapter is applied to the $n/m/P/C_{max}$ and in the next chapter, the MSXF is applied to the $n/m/P/C_{sum}$.

10.1 The Neighborhood Structure of the FSP

Because the FSP is a special case of the JSP, the concepts of critical path and blocks also apply to the FSP in a similar but much simpler fashion. We briefly review those concepts again in the FSP context for the better understanding.

The permutation flowshop scheduling problem designated by the symbols $n/m/P/C_{max}$ has n jobs that have to be processed on m machines in the same order. Because of this simplicity against the general JSP, we can assume that the machines are indexed in the processing order of jobs, so that the M_1 is the first machine to process jobs and M_m is the last. We are given the processing time p_{jr} of each operation of job J_j on machine M_r. A schedule can be represented by a permutation of jobs, or to put it more simply, a permutation of job numbers π.

A critical path is a sequence of operations starting from the first operation on the first machine M_1 and ending with the last operation on the last machine M_m. The starting time of each operation on the path, except for the first one, is equal to the completion time of its preceding operation—that is, there is no idle time along the path. Thus, the length of the critical path is the sum of the processing times of all the operations on the path and equals to C_{max}. There can be more than one critical path on a schedule.

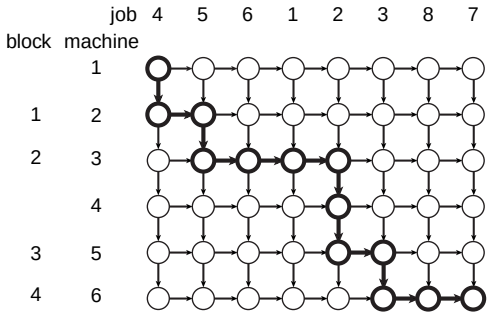

Figure 10.1: A grid graph representation of a solution to a problem of 8 jobs and 6 machines.

The operations on a critical path can be partitioned into subsequences, called *critical blocks*, according to their associated machines. A critical block consists of maximal consecutive operations on the same machine, as in the JSP, but here we make it simple and define a critical block as a subsequence of associated jobs, instead of operations.

Consider a schedule represented by a permutation π. Let $B_1, \ldots B_k$ be a set of all critical blocks that contains more than one job and let m_l be the index of the machine associated with B_l. Let $\pi(u_l)$ be the first job of B_l (and the last job of B_{l-1}). Then the 'inside' of B_l, denoted by $\hat{B}_l$, is defined as follows:

$$\hat{B}_l = \begin{cases} B_l \setminus \{\pi(u_{l+1})\} & \text{if } l = 1 \text{ and } m_l = 1 \\ B_l \setminus \{\pi(u_l)\} & \text{if } l = k \text{ and } m_l = n \\ B_l \setminus \{\pi(u_l), \pi(u_{l+1})\} & \text{otherwise.} \end{cases} \tag{10.1}$$

Figure 10.1 shows an example of schedule $\pi = 4, 5, 6, 1, 2, 3, 8, 7$ for a problem with $n = 8$ jobs and $m = 6$ machines represented by so-called a grid graph, which is a simplification of the disjunctive graph for the JSP. In the figure, the vertical axis corresponds to machines and the horizontal axis to jobs. Each circle represents an operation, and arrows precedence relation between operations. A critical path is marked by thick lines. In this example, there are four critical blocks $B_1, \ldots, B_4$ that contain more than one job. B_2 on machine 3, for example, consists of four jobs $5, 6, 1$ and 2, and $\hat{B}_2$ consists of jobs 6 and 1. Likewise $\hat{B}_4$ on machine 6 consists of jobs 8 and 7. Note that the machines are indexed in the processing order of jobs.

As described in Chapter 6, a neighborhood $N(x)$ of a point x in a search space can be defined as a set of new points that can be reached from x by exactly one transition or move (a single perturbation of x). One of the well-known transition operators for PFSP is the *shift move* that takes a job from its current position and re-inserts it in another position. Let $v = (a, b)$ be a pair of positions in π. Here, v defines a move that removes the job $\pi(a)$ from a position a and re-inserts it in a position b. If $a < b$, the resulting schedule is represented by $\pi_v = \pi(1), \ldots, \pi(a-1)\pi(a+1), \ldots, \pi(b), \pi(a), \pi(b+1), \ldots, \pi(n)$, and if $a > b$, $\pi_v = \pi(1), \ldots, \pi(b), \pi(a), \pi(b+1), \ldots, \pi(a-1)\pi(a+1), \ldots, \pi(n)$. A neighborhood $N(V, \pi)$ is defined as the set of all schedules obtained by shift moves in $V = \{(a, b) : b \notin \{a-1, a\}, a, b \in \{1, \ldots, n\}\}$.

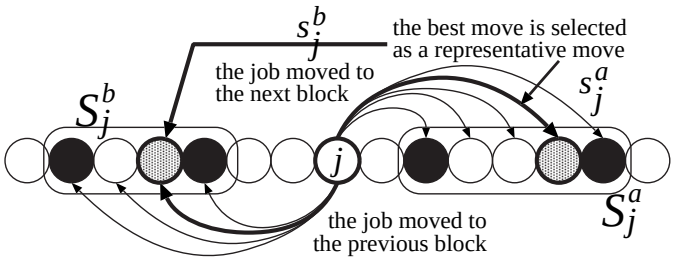

Figure 10.2: The best move to the next/previous block is selected as a representative.

Let $W_l(\pi)$ be a set of moves restricted to the inside of B_l, namely

$$W_l(\pi) = \{(a, b) \in V | a, b \in \hat{B}_l\} \tag{10.2}$$

and

$$W(\pi) = \bigcup_{l=1}^{k} W_l(\pi) \tag{10.3}$$

then the block property described in Theorem 2.5 for the JSP is reformulated as follows:

Corollary 1 (Block property for the FSP) *For any schedule β,*
if $\beta \in \mathcal{N}(W(\pi), \pi)$ *then* $C_{max}(\beta) \geq C_{max}(\pi)$.

According to Corollary 1 above, no move in $W(\pi)$ can directly improve schedule π. Therefore, it is reasonable, for computational efficiency, to reduce the size of the neighborhood $N(V, \pi)$ by eliminating moves in $W(\pi)$, and to use a new neighborhood $N(V \setminus W(\pi), \pi)$, which we call here a "critical block neighborhood".

10.2 Representative Neighborhood

Nowicki and Smutnicki have proposed the *representative* neighborhood method[20], where a *reduced* neighborhood is generated from the original neighborhood by first patitioning the original neighborhood members into clusters and then picking up the best move (representative) from each cluster as a representative. A new neighborhood is the set of all representative moves.

In this chapter, the representative neighorhood method is applied as follows. For each job j in a critical block, let S_j^a be a set of moves that shift the job j to some position in the next block; similarly S_j^b shifts j to the previous block. Evaluate schedules obtained from each move in S_j^a and denote the best one by s_j^a. Similarly s_j^b is obtained from S_j^b. Then the representative neighborhood is defined as a set of all schedules obtained by representative moves $\{s_j^a, s_j^b\}$ for all jobs j in all critical blocks (see Figure 10.2).

10.3 Distance Measures

To measure the difference between two permutation schedules S and T, an appropriately defined distance is required. In the case of the JSP, we have used the DG distance introduced in Section 2.4. For the FSP, two well-known distances are considered as follows:

precedence-based: This distance counts the number of job pairs $\{i, j\}$ in which j is preceded by i in S but not in T.

position-based: This distance sums up the positional differences for each job in S and T.

The first precedence-based distance is equivalent to the DG distance if the FSP is viewed as a special case of the JSP. The relationship between these two distances will be discussed in the later section.

10.4 Landscape analysis

According to Höhn and Reeves [61], a landscape is defined by a triple of a search space, an objective function and a distance measure. The link between landscape and search algorithm is given by the NS operators used in the algorithm. Because these operators generate new points in the search space relative to a given point, they define a distance $d_{\mathcal{N}}(s,t)$ on the search space given by the minimum number of applications of operator $\mathcal{N}$ that will convert element t into element s.

One can understand the degree of difficulty of the given combinatorial optimization problem by looking at its landscape: if the landscape is simple and has only one peak, it is very easy to find the global optimum by using simple best ascent search. Unfortunately most $\mathcal{NP}$-hard combinatorial optimization problems, including PFSP, have very 'rugged' landscapes with many false peaks under any NS operator.

Recently, Boese et al. [62] have shown that an appropriate choice of NS operator introduces some neat structure into the landscape. In this 'big valley' structure, local optima occur in clusters – good candidate solutions are usually to be found 'fairly close' to other good solutions. If a landscape has this structure, it would support the idea of generating new starting points for search from a previous local optimum rather than from a random point in the search space.

Before we apply our GLS method to PFSP, we investigate whether there is a big valley structure for the PFSP and the NS operator using the representative neighborhood and a stochastic search using (9.1) (constant temperature). For the same PFSP but with simpler NS operators, similar experiments reported in Reeves [63] found such a landscape did occur.

As discussed in [62, 64, 63], the existence of a big valley structure can be examined by first generating a set of random local optima and then observing the correlation between their objective function values and their distances to the nearest global optimum, and/or their average distances to other local optima. The distance used here should be $d_{\mathcal{N}}$ for an operator $\mathcal{N}$. However this distance is difficult to compute, and precedence-based distance is used here as an approximation.

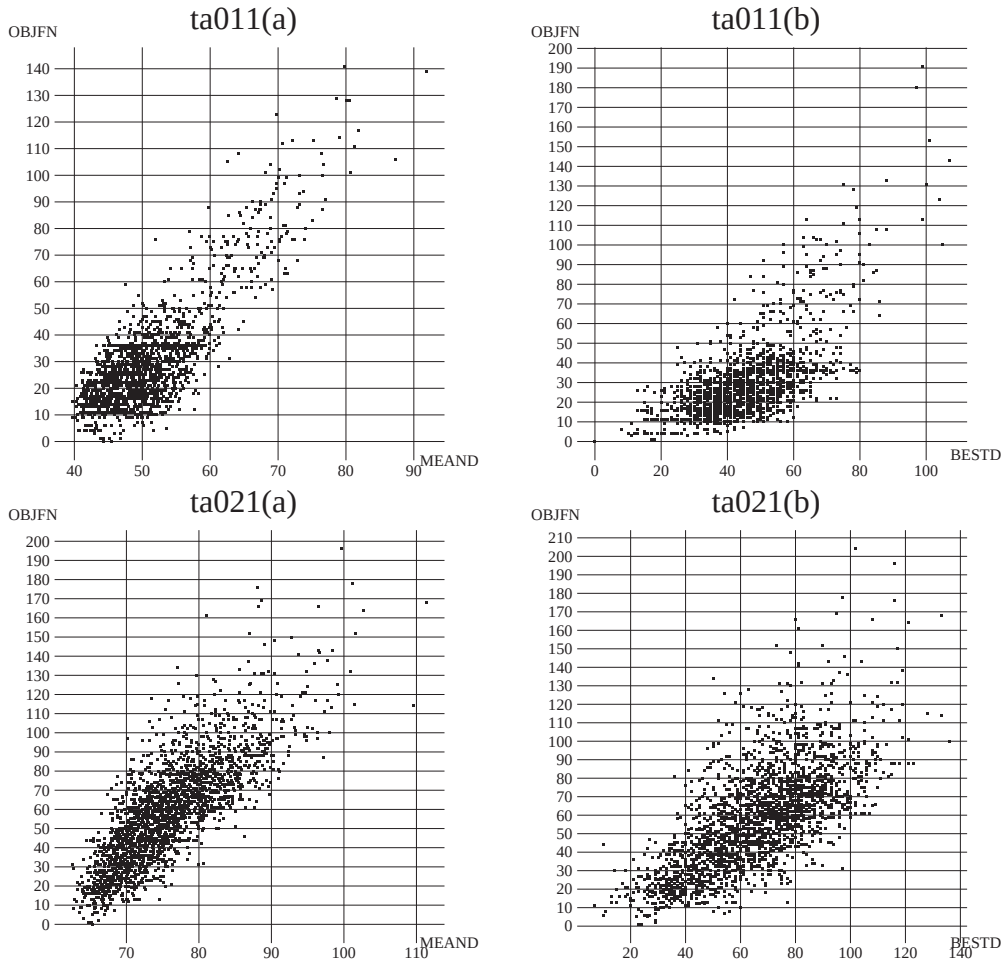

Figure 10.3: 1841 distinct local optima obtained from 2500 short term local search for the ta011 (20×10) problem and 2313 distinct local optima for the ta021 (20×20) problem are plotted in terms of (a) average distance from other local optima and (b) distance from global optima (x-axis), against their relative objective function values (y-axis).

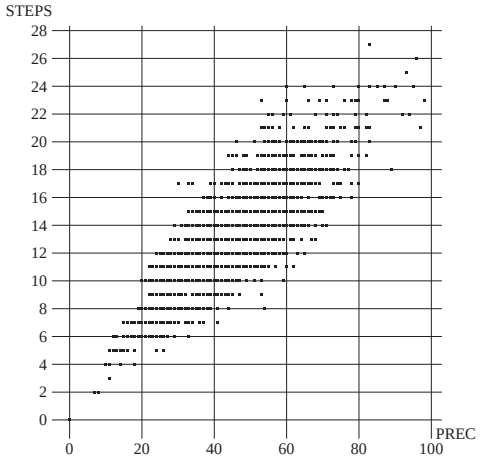

Figure 10.4: The correlation between the precedence-based distance (PREC) and the approximate number of steps (STEPS)

Figure 10.3 shows a scatter plot of random local optima for problems ta011 and ta021, being respectively the first of Taillard's 20×10 and 20×20 groups of problems [21]. Each local optimum is generated by running the neighborhood search described in Algorithm 6.1.1 with $L = 5000$ based on the stochastic method with acceptance probability P_c, $c = 5$. Extensive preliminary experiments found only two distinct global optima for the ta011 problem, very close to each other in terms of the precedence-based distance (the distance is two) and only one global optimum for ta021 problem; although one cannot rule out the possibility of finding other different global optima by continuing the search. However, more than 2500 global optima were found for the smaller ta001 (20×5) problem by spending the same amount of CPU time.

The x-axis in Figure 10.3 represents (a) the average precedence-based distance from other local optima (MEAND), and (b) the precedence-based distance from one of the nearer global optima (BESTD). The y-axis represents their objective function values relative to the global optimum. These plots clearly show that there are good correlations between the distances and objective function values. The calculated correlation coefficients for each plot are: ta011(a): 0.74, ta011(b): 0.50, ta021(a): 0.62 and ta021(b): 0.44. These values are statistically significant at the 0.1% level, on the basis of 1000 replications in a randomization test [63]. These high correlations suggest that the local optima are radially distributed in the problem space with the global optima as the centre, and the more distant are the local optima from the centre, the worse are their objective function values. Hence, by tracing local optima step by step, moving from one optimum to nearby slightly better one, without being trapped, one can eventually reach a near global optimal solution.

In the analysis above, the precedence-based distance is used as a surrogate for d_N, because the minimum number of steps for the neighborhood operator to reach the global optimum is difficult to compute. Although the precedence-based distance seems to be a good alternative, the approximation still need to be justified. For this purpose, the approximate number of steps to

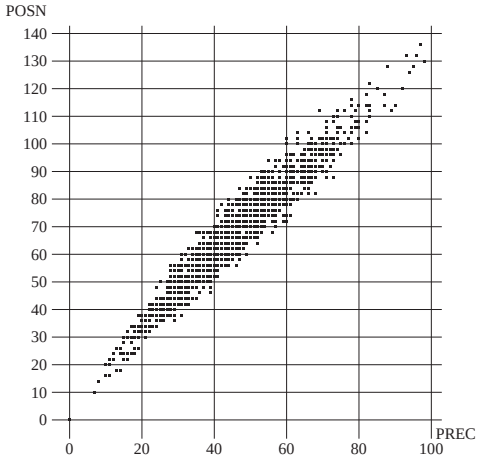

Figure 10.5: The correlation between the precedence-based distance (PREC) and the position-based distance (POSN)

reach the global optimum from each local optimum was calculated by choosing the closest move to the global optimum each time from the neighborhood. While this does not necessarily give the best distance between two points, it seems likely to give a fairly close upper bound.

Figure 10.4 shows the correlation between the precedence-based distance (PREC) and the approximate number of steps (STEPS) for the local optima shown in Figure 10.3 ta011(a) (correlation coefficient is 0.66). Figure 10.5 shows that there is a strong correlation between the precedence-based distance (PREC) and the position-based distance (POSN) for the same local optima (correlation coefficient is 0.91). Thus it does not matter which distance is used. The same kind of experiments were carried out for all Taillard's 20×10 and 20×20 benchmarks, and similar results were obtained in every case. Therefore, the use of the easily-computed precedence-based distance appears to be justified, and the 'big valley' structure can be assumed to hold for this neighborhood.

10.5 MSXF-GA for PFSP

As described in Section 9.1, the MSXF operator is designed to find a new local optimum based on previous ones. MSXF-GA provides a framework for traversing local optima without being trapped, by concentrating its attention on the area between the parent solutions and thus eventually finding a very good solution under the assumption of a 'big valley'. MSXF-GA was applied to PFSP using the representative neighborhood described in Section 10.1 and the precedence-based distance.

Algorithm 10.5.1 describes the outline of the MSXF-GA routine for the PFSP. In this model, the population is ranked according to the makespan values, and the parents are selected from the population with a probability inversely proportional to their ranks. The newly generated

solution q is inserted into the population only if its makespan is better than the worst in the current population. To avoid premature convergence even under a small-population condition, if an individual with the same makespan already exists in the population, then q is not inserted into the population in Step 2c.

Algorithm 10.5.1 MSXF-GA for the PFSP

The population size P is given.

1. Initialize population: randomly generate a set of permutation schedules. Sort the population members in descending order of their makespan values.
2. **do**
   (a) Select two schedules p_1, p_2 from the population with a probability inversely proportional to their ranks.
   (b) Do Step 2(b)i with probability P_X, or otherwise do Step 2(b)ii.
      i. **If** the precedence-based distance between p_1, p_2 is less than d_{min}, apply MSMF to p_1 and generate q.
      Otherwise, apply MSXF to p_1, p_2 using the representative neighborhood and the precedence-based distance and generate a new schedule q.
      ii. Apply Algorithm 6.1.1 with acceptance probability P_c and the representative neighborhood.
   (c) If q's makespan is less than the worst in the population, and no member of the current population has the same makespan as q, replace the worst individual with q.

   until some termination condition is satisfied.
3. Output the best schedule in the population.

10.6 Experimental results

In Section 10.4, the existence of a big valley structure became clear for the relatively small-size PFSP instances. An adaptive multi-start method (AMS) in which new local search is concentrated in a region between previously found local optima should be effective at least for these problems. MSXF-GA for PFSP is especially designed as one of the AMS approaches for PFSP. Preliminary experiments show that MSXF-GA is very effective for the problem instances discussed in Section 10.4, and the global optima are found very quickly. In this section we will extend our investigations to larger-size problems and apply MSXF-GA to a subset of Taillard's benchmark problems.

Table 10.1 summarizes the performance statistics of MSXF-GA for a subset of Taillard's benchmark problems together with the results found by Nowicki and Smutnicki using their tabu

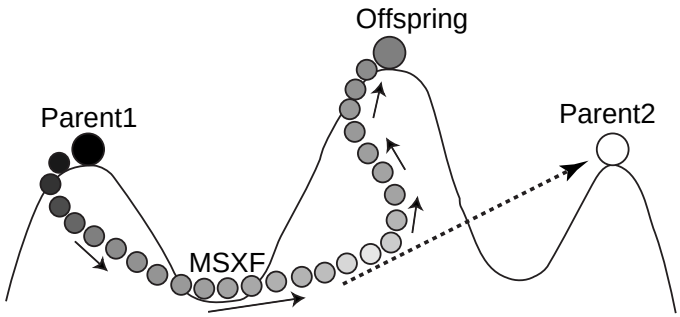

Figure 10.6: Navigated local search by MSXF-GA: A new search is started from one of the parents and while no other good solutions are found, the search 'navigates' towards the other parent. In the middle of the search, good solutions would be eventually found somewhere between the parents. That direction is then pursued to the top of a hill (or a bottom of the valley, if it is a minimization problem) — a new local optimum.

Table 10.1: Results of the Taillard benchmark problems

	50×20				100×20				200×20			
No.	best	avg.	nowi	lb – ub	best	avg.	nowi	lb – ub	best	avg.	nowi	lb – ub
1	**3861**	3880	3875	3771–3875	6242	6259	6286	6106–6228	11272	11316	11294	11152–11195
2	**3709**	3716	3715	3661–3715	6217	6234	6241	6183–6210	11299	11346	11420	11143–11223
3	**3651**	3668	3668	3591–3668	6299	6312	6329	6252–6271	11410	11458	11446	11281–11337
4	**3726**	3744	3752	3631–3752	6288	6303	6306	6254–6269	11347	11400	11347	11275–11299
5	**3614**	3636	3635	3551–3635	6329	6354	6377	6262–6319	11290	11320	11311	11259–11260
6	3690	3701	3698	3667–3687	**6380**	6417	6437	6302–6403	11250	11288	11282	11176–11189
7	3711	3723	3716	3672–3706	6302	6319	6346	6184–6292	11438	11455	11456	11337–11386
8	**3699**	3721	3709	3627–3700	6433	6466	6481	6315–6423	11395	11426	11415	11301–11334
9	3760	3769	3765	3645–3755	6297	6323	6358	6204–6275	11263	11306	11343	11145–11192
10	3767	3772	3777	3696–3767	6448	6471	6465	6404–6434	11335	11409	11422	11284–11313

best, avg.: our best and average makespan values
nowi: results of Nowicki and Smutnicki
lb, ub: theoretical lower bounds and best known makespans taken from OR-library

search implementation[20] and the lower and upper bounds, taken from the OR-library [60]. (Upper bounds are the currently best-known makespans, most of them found by a branch and bound technique with computational time unknown). In all, 30 runs were completed for each problem under the same conditions but with different random number seeds. For each MSXF-GA run, population size = 15, constant temperature $c = 3$, number of iterations for each MSXF = 1000, $d_{min} = n/2$ and $P_X = 0.5$ are used. Each run is terminated after 700 iterations, which takes about 12, 21 and 47 minutes of CPU time respectively for each 50×20, 100×20 and 200×20 problems on a DEC Alpha 600 5/226.

It can be seen that the results for 50×20 problems are remarkable: the solution qualities of our best results are improved over those found in [20] for most of the problems, and some results (marked in bold letters) are even better than the existing best results reported in the OR-library. The results for larger problems are not as impressive as those of 50×20 problems, but still good enough to support our hypothesis. The degradation is probably due to the increasing complexity of the neighborhood calculation. In fact for problems where the ratio $n/m > 3$, Nowicki and Smutnicki abandoned their representative neighborhood and used a simple one instead: just moving a job to the beginning or the end of its critical block. They also implemented an efficient way of evaluating all the members in the neighborhood in a specific order. This method is useful for the tabu search, but not directly applicable to our stochastic search.

10.7 Concluding Remarks

The landscape for the Permutation Flowshop Scheduling Problem with stochastic local search and the representative neighborhood structure has been investigated. The experimental analysis using 20×10 and 20×20 Taillard benchmark problems shows the existence of a ‘big valley’ structure for PFSP. This suggests a well-designed AMS method, such as MSXF-GA in which new local search is concentrated in a region between previously found local optima should be effective in finding near-optimal solutions. MSXF-GA for the PFSP is implemented using the neighborhood operator and applied to more challenging benchmark problems. Experimental results demonstrates the effectiveness of the proposed method.

Chapter 11

C_{sum} Permutation Flowshop Scheduling by Genetic Local Search

11.1 Introduction

We have already proposed an efficient method based on the genetic local search with the MSXF to solve the $n/m/P/C_{max}$ in the previous chapter. In this chapter we deal with the $n/m/P/C_{sum}$. Compared to the C_{max} problem, the C_{sum} problem is more difficult to optimize, mainly because the calculation of the objective function is more time consuming, and problem specific knowledge such as critical blocks is not applicable. However, this difficulty can be partly overcome by extending the idea of the representative neighborhood discussed in Section 10.2.

11.2 Representative Neighborhood

Let s be a job sequence of the current solution, and $s[i,k]$ be a new job sequence obtained from s by moving a job from the i^{th} position in s and re-inserting it in the k^{th} position. $N_i^a(s)$ and $N_i^b(s)$, subsets of $N(s)$, are defined as follows:

$$N_i^a(s) = \{s[i,k] \mid i < k \leq n\}, N_i^b(s) = \{s[i,k] \mid 1 \leq k < i\}.$$

Thus the original neighborhood $N(s)$ is divided into clusters consisting of $N_i^a(s)$ and $N_i^b(s)$. Let $\overline{N_i^a(s)}$ and $\overline{N_i^b(s)}$ be one of the best members in $N_i^a(s)$ and $N_i^b(s)$ respectively. The representative neighborhood $N^*(s)$ can be denoted as:

$$N^*(s) = \{\overline{N_i^a(s)} \mid 1 \leq i < n\} \cup \{\overline{N_i^b(s)} \mid 1 < i \leq n\}.$$

In most local search algorithms, the NS operators to choose a new member from the neighborhood of the current solution can be categorized into two types according to their choice criteria discussed in Section 6.1; one is best descent, and the other is first descent. The best descent method scans all the members in the neighborhood and choose the best as a new current solution. This is suitable when the neighborhood size is small and the cost of evaluating all the

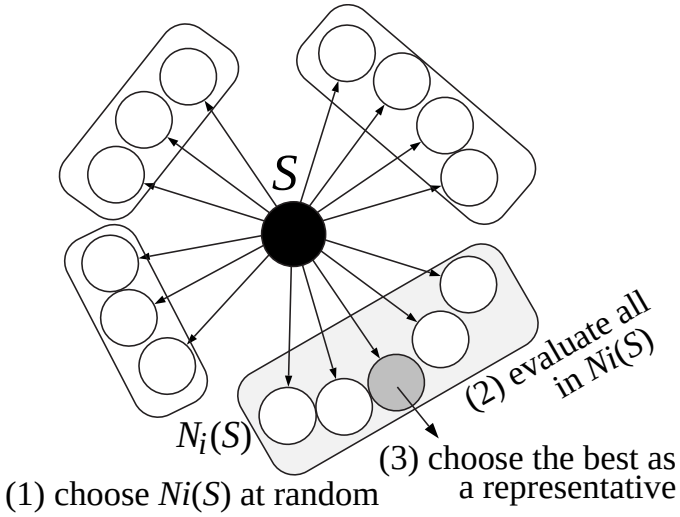

Figure 11.1: Representative neighborhood

members is negligible. Tabu search can be seen as an extension of this method. The first descent method selects one member (at random) and accepts it if it is sufficiently good, otherwise selects another one. This can be used even when the neighborhood size is large. The stochastic sampling in stochastic local search including SA can be seen as an extension of this method. The representative neighborhood fills the gap between these two criteria: a cluster $N_i(s)$ is chosen randomly by using first descent, then best decent is applied to evaluate all the members in $N_i(s)$ of which the best is chosen as a representative. Figure 11.1 illustrates this process. As we will see in the later section, this enables the TS and a stochastic local search method to integrate into a single unified method.

11.3 Tabu List Style Adaptive Memory

As described in Chapter 6, Tabu Search (TS) adopts a deterministic local search approach with a 'memory' implemented as a 'tabu list' of moves which have been made in the recent past of the search, and which are 'tabu' or forbidden for a certain number of iterations. The use of the representative neighborhood makes it easy to have 'tabu list' style adaptive memory.

If a solution $s[i,k]$ generated from the current solution s by moving a job $j = s[i]$ to the k^{th} position is accepted, the pair (j,i), i.e. the job and its original position, is stored on the top of a list of length l and recorded as *tabu*. The oldest element in the list is then deleted. In the subsequent iterations, a solution generated by moving job j to the i^{th} position should not be accepted as long as (j,i) is on the list. In our representative neighborhood scheme, this is achieved by excluding the tabu solution from the calculation of the representative best solution ($\overline{N_i^a(s)}$ and $\overline{N_i^b(s)}$) so that the representative neighborhood will contain no tabu solutions. Because the tabu solutions have already been excluded from the representative neighborhood, there is no need to modify the stochastic local search procedure described in Section 10.5 by this modification.

11.4 Experimental Results

We applied our method to some of Taillard's benchmark problems (ta problems, in short) [21]. First it was applied to relatively easy problems from ta001 to ta030 (the number of jobs is 20 and the number of machines is 5, 10 and 20: denoted by 20x5, 20x10 and 20x20). Six runs were carried out for each problem with different random seeds. The parameters used in these experiments are: $P = 5, L_1 = 1000, L_2 = 700, P_X = 0.5$ and the length of the tabu list $l = 7$.

Here quite consistent results were obtained, i.e. almost all of the 6 runs converged to the same job sequence in a short time (from a few seconds to a few minutes) before the limit of $L_2 = 700$ was reached on a HP workstation. The best results (and they are also the average results in most cases) are reported in Table 11.1 together with the results obtained by the constructive method (NSPD) due to J.Liu [65].

Table 11.1: Taillard's benchmark results (ta001 – ta030)

prob	best	NSPD	prob	best	NSPD	prob	best	NSPD
001	14033	14281	011	20911	21520	021	33623	34119
002	15151	15599	012	22440	23094	022	31587	32706
003	13301	14121	013	19833	20561	023	33920	35290
004	15447	15925	014	18710	18867	024	31661	32717
005	13529	13829	015	18641	19580	025	34557	35367
006	13123	13420	016	19245	20010	026	32564	33153
007	13548	13953	017	18363	19069	027	32922	33763
008	13948	14235	018	20241	21048	028	32412	33234
009	14295	14552	019	20330	21138	029	33600	34416
010	12943	13054	020	21320	22212	030	32262	33045

Problems ta031 to ta050 (50x5 and 50x10 problems) are much more difficult and the best results were different in each run. Ten runs were carried out for each problem with different random seeds. The parameters used in these experiments were: $P = 30, L_1 = 10000, L_2 = 700, P_X = 0.5$. It takes 45 minutes per run for 50x5 problems (ta031 to ta040) and 90 minutes for 50x10 problems (ta041 to ta050).

It is difficult to say how good these solutions are, in other words, how far they are from the global optima. Even for the easier problems in Table 11.1, there is no guarantee that the best solutions obtained so far are optimal, although we believe that they are at least very close to being so. For the problems in Table 11.2, it is almost certain that our best results are not optimal. In fact we found one solution of $C_{sum} = 64803$ for problem ta031 by an overnight run.

11.5 Concluding Remarks

A new genetic local search method is proposed to solve the C_{sum} permutation flowshop scheduling problem. This method effectively integrates the stochastic sampling of Simulated Annealing,

Table 11.2: Taillard's benchmark results (ta031 – ta040)

prob	best	average	NSPD	prob	best	average	NSPD
031	64860	64934.8	66590	041	87430	87561.4	90373
032	68134	68247.2	68887	042	83157	83305.8	86926
033	63304	63523.2	64943	043	79996	80303.4	83213
034	68259	68502.7	70040	044	86725	86822.4	89527
035	69491	69619.6	71911	045	86448	86703.7	89190
036	67006	67127.6	68491	046	86651	86888.0	91113
037	66311	66450.0	67892	047	89042	89220.7	93053
038	64412	64550.1	66037	048	86924	87180.5	90614
039	63156	63223.8	64764	049	85674	85924.3	91289
040	68994	69137.4	69985	050	88215	88438.6	91622

the adaptive memory using the tabu list of Tabu Search and the population-based search of Genetic Algorithms into a single unified framework as summarized in Figure 11.2. The method is applied to Taillard's benchmark problems. Experimental results demonstrate the effectiveness of the proposed method.

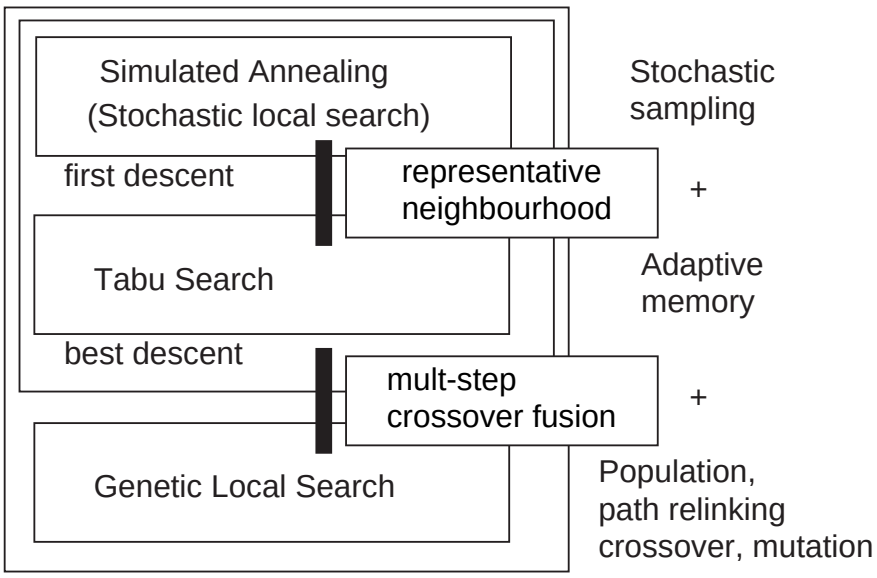

Figure 11.2: The framework of the proposed method

Chapter 12

Tabu Search with a Pruning Pattern List for the Flowshop Scheduling Problem

12.1 Introduction

In this chapter, an approximation method based on Tabu Search with an additional memory structure called “pruning pattern list” is described [66]. A pruning pattern is constructed from a solution, which is represented by a permutation of job numbers, by replacing some of its job numbers by a “wild card” or a “don’t care” symbol. The job numbers to be replaced are determined by investigating the critical path of the schedule. A list of pruning patterns are generated from “good” schedules that are obtained in the course of a search process, and maintained. The list is used to inhibit the search to visit already searched and no longer interesting region again and again.

12.2 Tabu Search

As described in Chapter 6, Tabu Search (TS) adopts a deterministic local search approach with a ‘memory’ implemented as a ‘tabu list’ of moves which have been made in the recent past of the search, and which are ‘tabu’ or forbidden for a certain number of iterations. A tabu move may be accepted (even if it is ’tabu’) if certain criteria are satisfied, such as the solution obtained by the application of the move being better than the best solution obtained so far. Such criteria are called aspiration criteria.

Nowicki and Smutnicki [20] proposed a Tabu Search (TS) method for the Permutation Flowshop Scheduling Problem (PFSP). The tabu list used in their approach is described as follows by using the notations used in Chapter 10.

Let π be a permutation of job numbers representing a schedule and $v = (a, b)$ be a move that removes the job $\pi(a)$ from position a and re-inserts it into position b in π. When $v = (a, b)$ with $a < b$ is performed on π, the precedence relation between a pair of jobs $(\pi(a), \pi(a + 1))$ is reversed. To inhibit the future recovery, i.e., re-reversal, of the precedence relation, $(\pi(a), \pi(a+1))$ is stored as ‘tabu’ in the tabu list T of length tl. Then, a move $v' = (a', b')$ with $a' < b'$ cannot be

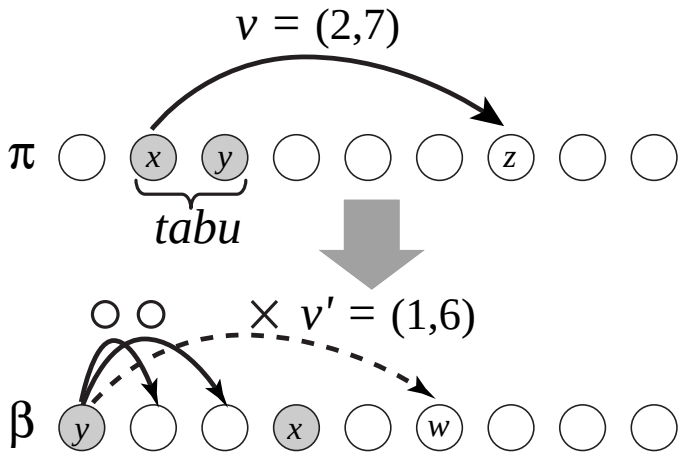

Figure 12.1: When $v = (2,7)$ is applied to π, $(\pi(2),\pi(3)) = (x,y)$ is stored in T as tabu. Later, $v' = (1,6)$ is not allowed to apply to β because it will restore the previously banned precedence relation between (x,y).

performed on a new schedule β if v' induces a recovery of any precedence relation in T, i.e., if $\{(\beta(j),\beta(a')) \mid j = a'+1,\ldots,b'\} \cap T \neq \phi$. Likewise, $(\pi(a-1),\pi(a))$ is stored as 'tabu' if $v = (a,b)$ with $a > b$ is performed on π.

The neighborhood $N(V \setminus W(\pi),\pi)$, which is a subset of $N(V,\pi)$, is still too big for problems of even moderate size. Thus, it is impracticable to evaluate all the neighbors in $N(V \setminus W(\pi),\pi)$, so Nowicki and Smutnicki use a subset of $N(V \setminus W(\pi),\pi)$ as a neighborhood structure. Alternatively, one can make the evaluation probabilistic: a member α from $N(V \setminus W(\pi),\pi)$ is selected at random and accepted if $C_{max}(\alpha) < C_{max}(\pi)$, otherwise probabilistically accepted according to the Metropolis probability $P_c(\alpha) = \exp(-\Delta C_{max}/c)$, where $\Delta C_{max} = C_{max}(\alpha) - C_{max}(\pi)$.

12.3 Pruning Pattern

A pruning pattern $[\pi]_l$ associated with π and its critical block B_l is derived from π by replacing the jobs that belong to $\hat{B}_l$ by $\star$ as follows:

$$[\pi]_l(j) = \begin{cases} \star & \text{if } j \in \hat{B}_l, \\ \pi(j) & \text{otherwise.} \end{cases}$$

For example, the pruning pattern corresponding to $\pi = 4,5,6,1,2,3,8,7$ and B_2 in Figure 10.1 is $[\pi]_2 = 4,5,\star,\star,2,3,8,7$. The makespan of $[\pi]_l$ is defined as the makespan of π, namely, $C_{max}([\pi]_l) := C_{max}(\pi)$. The $\star$ means a wild card, and a permutation β 'matches' $[\pi]_l$ if $\beta(j) = [\pi]_l(j)$ at all but $\star$ positions. For example $4,5,6,1,2,3,8,7$ and $4,5,1,6,2,3,8,7$ both match $[\pi]_2$. $[\pi]_l$ and a set of all permutations that match $[\pi]_l$ are identified. It is clear that $\mathcal{N}(W_l(\pi),\pi) \subset [\pi]_l$. The block property described in Corollary 1 in Section 10.1 can be reformulated using $[\pi]_l$

as follows:

Corollary 2 *For any schedule β and any pruning pattern $[\pi]_l$, $\beta \in [\pi]_l \Longrightarrow C_{max}(\beta) \geq C_{max}([\pi]_l)$.*

This again can be reformulated as follows:

Corollary 3 *For any two pruning patterns $[\pi]_i$ and $[\phi]_j$, $[\pi]_i \subset [\phi]_j \Longrightarrow C_{max}([\pi]_i) \geq C_{max}([\phi]_j)$.*

Corollary 2 suggests that when a new, possibly good, solution π is found during the search, $[\pi]_l$ identifies a region where no solutions are better than π, and that excluding $[\pi]_l$ from the search space can reduce the size of the search space without eliminating the global optima.

12.4 Pruning Pattern List Approach

The basic idea of the pruning pattern list approach is to reduce the size of the search space effectively through storing the 'important' pruning patterns that correspond to a long critical block of a good solution. Let $[\pi]$ (without suffix) be the pruning pattern that corresponds to the longest critical block of schedule π. The pruning pattern list PL with length pl is maintained and updated as shown in Algorithm 12.4.1. Here, $N(\pi)$ represents the neighborhood of π. $N(V \setminus W(\pi), \pi)$ or its subset is normally used as $N(\pi)$. There is no good reason to keep patterns that are rarely accessed in the list. Therefore such patterns are replaced by new patterns. According to Corollary 3, it is also not necessary to keep pruning patterns in the list that are subsets of other patterns. Thus they are removed.

Starting from an initial solution, the local search iteratively replaces the current solution with one of its neighbors. In advanced local search strategies, such as Simulated Annealing (SA) and TS, cost-increasing neighbors can be accepted as well as cost-decreasing ones. Accepting cost-increasing moves enables the search to escape from the local optima, but this may also cause revisiting of previously evaluated points in the search space – something that is wasteful of computing resources in itself, and which also means the search is not adequately diversified. One of the main aims of tabu search is to discourage such revisiting. This can be accomplished by means of an explicit 'tabu list' of points previously visited, but normally it is easier, and more efficient, to record specific attributes of such points, or of moves that would lead towards them. Nevertheless, recording attributes of the points, and not the points themselves, can risk treating even moves that are better than any solution obtained so far as tabu. This is one of the main reasons why aspiration criteria should be introduced into TS.

In the case of the pruning pattern list approach, Corollary 2 guarantees that a move that matches a pruning pattern is never better than the best solution obtained so far. This means that an aspiration criterion is not necessary, and it also means that the pruning pattern list can serve as a longer-term memory to prevent the search getting stuck in an already searched and no longer interesting region. Unlike the tabu list, an old pattern can remain in the pruning list as long as it is accessed frequently. If $N(V \setminus W(\pi), \pi)$ or its subset is used as the neighborhood of the current solution π, it does not contain any solution that matches $[\pi]_l$, even without PL. However, it is still possible that the region $[\pi]_l$ would be revisited in some later stages without any better

Algorithm 12.4.1 The pruning pattern list approach to the PFSP.

A flowshop scheduling problem is given as an input. The size of PL is given as pl.

1. PL is initialized as a list of empty pl elements.
2. Generate a starting solution schedule π_0 at random, and set $\pi = \pi_0$.
3. **do**
   (a) Calculate $N(\pi)$ from π.
   (b) $N^P(\pi)$ is initialized as $N(\pi)$. For each $\alpha \in N^P(\pi)$, if α matches any pattern $[\beta] \in PL$, then α is removed from $N^P(\pi)$, resulting $N^P(\pi) := N^P(\pi) \setminus \alpha$, and the 'access count' of $[\beta]$ is incremented. $N^P(\pi)$ is used as the new neighborhood of π.
   (c) **do**
      i. A candidate schedule ϕ is chosen from $N^P(\pi)$ and is accepted or rejected based on the value $C_{max}(\phi)$.

      until ϕ is accepted.
   (d) When the accepted ϕ is a cost-decreasing solution (i.e., $C_{max}(\phi) < C_{max}(\pi)$), then its longest pruning pattern $[\phi]$ is stored in PL.
   (e) All the existing patterns $[\beta]$ in PL such that $[\beta] \subset [\phi]$ are removed and substituted by empty patterns.
   (f) If the number of non-empty patterns on PL exceeds pl, then the least accessed pattern $[\gamma]$ is removed from the list.
   (g) Set $\pi = \phi$.

   until termination conditions are satisfied.
4. Output the best solution obtained.

solutions than π being found. The size of PL is fixed to pl mainly for computational efficiency. Theoretically, $N^P(\pi)$ may not satisfy the connectivity property that $N(V \setminus W(\pi), \pi)$ does. It should be noted that the pruning pattern list is treated in a similar way that the population is treated in Genetic Algorithms (GAs), especially in the steady state model [67]. The access count of the pruning pattern corresponds to the fitness function of the individual in GAs.

12.5 Experimental Results

The pruning pattern list approach described in the previous section can be embedded into any local search method, including TS, SA, and even GAs [68, 69]. Here a simple probabilistic version of the TS described in Section 12.2 is used as a test case. The programs are coded in *C*.

The graph on the left side of Figure 12.2 shows the time evolution of the makespan averaged over 30 runs of the TS with and without the pruning pattern list. This uses the ta041 problem, which is one of Taillard's benchmarks for size $n = 50, m = 10$ [21]. The parameters used are $c = 3.0, tl = 7$ and $pl = 10$. One run takes about 10 minutes on an HP 700 workstation. The best and worst makespans obtained among 30 runs are 3000 and 3016 respectively when TS with the pruning pattern list is used, and 3010 and 3025 when TS without the pruning pattern list is used. It can be seen that the pruning pattern list approach improves both the solution quality and the speed. Without the pruning pattern list 12 runs out of 30 were trapped at a local minimum of makespan = 3025, which Nowicki and Smutnicki reported as the new reference makespan for this problem. This suggests that their TS method can also be improved by incorporating the pruning pattern list approach.

The graph on the right side of Figure 12.2 shows the results of the ta051 problem (50 jobs and 20 machines) based on 10 replications rather than 30. It also includes the computationally equivalent MSXF-GA results reported in [69] for comparison. The pruning pattern list approach is also applied to other Taillard's benchmarks of the same sizes, and the similar behaviors were observed for most of the cases. The results are summarized in Figure 12.3. The parameters used are the same as in the ta041 and ta051 cases and the results are averaged over 10 replications. The results of TS with the pruning pattern list generally outperform the results of TS without the pruning pattern list , except for the ta057 problem, where the difference is not so significant.

12.6 Concluding Remarks

The pruning pattern list approach to the makespan-minimizing permutation flowshop scheduling problem is proposed and embedded into a Tabu Search method. The preliminary experimental results demonstrate the effectiveness of the proposed approach. Future research will aim to implement an way to efficiently scan the pruning pattern list to find a match quickly, because currently the scanning time becomes not negligible as the problem size increases. The proposed approach can also be embedded into GAs. However, the implementation details including whether each individual should have its own pruning pattern list and should occasionally exchange it with others, or a single list should be shared by all the members in the population are yet to be investigated. It is also expected to apply the pruning pattern list approach to the jobshop scheduling problem.

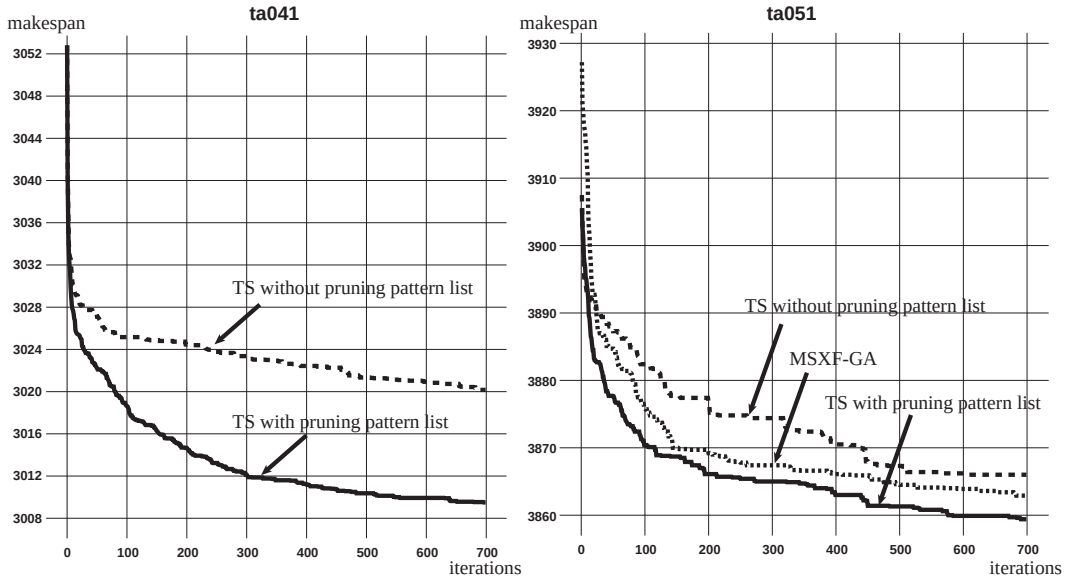

Figure 12.2: The time evolutions of makespans for the ta041 (50 jobs and 10 machines) problem averaged over 30 tabu search runs with and without the pruning pattern list (left). The time evolutions for the ta051 (50 jobs and 20 machines) problem averaged over 10 tabu search runs and the computationally equivalent MSXF-GA runs for comparison (right).

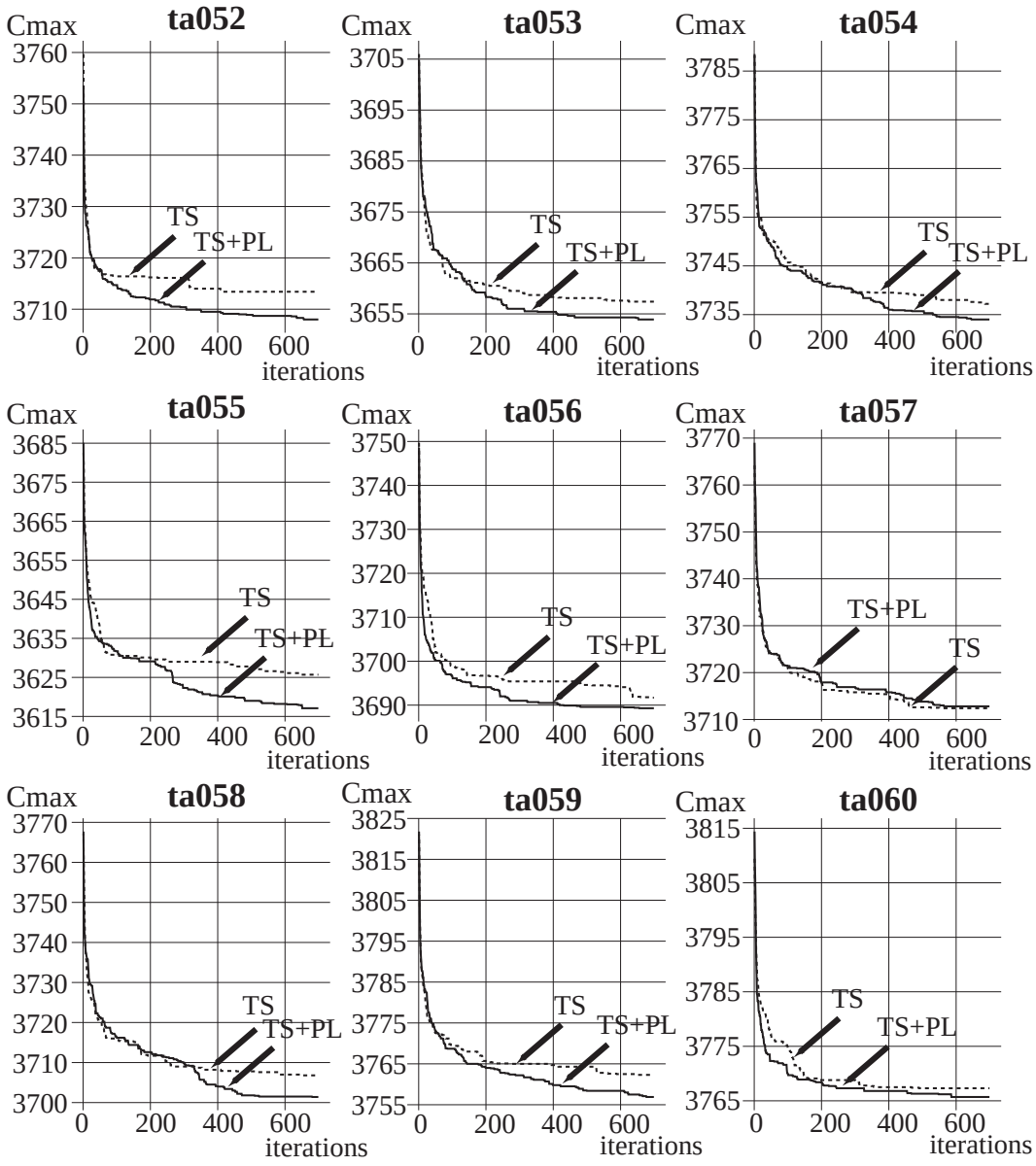

Figure 12.3: The time evolutions for the other nine Taillard problems of 50 jobs and 20 machines (ta052 – ta060) averaged over 10 tabu search runs with (labeled TS+PL) and without (labeled TS) the pruning pattern list.

Chapter 13

Conclusions

In this thesis, we have investigated various approaches to solve scheduling problems by meta-heuristics, including Genetic Algorithms, Simulated Annealing and Tabu Search, and demonstrated the effectiveness of the proposed methods. We have basically started from a simple problem-independent approach to more tailored problem-specific ones that involve more domain specific knowledge of the scheduling problem.

The key features of the proposed methods include the effective use of the concept of active schedule and the GT algorithm as well as the concept of critical path and blocks. The use of these problem-specific knowledge greatly improves the performance of the proposed methods. This is because the use of problem-specific knowledge enables to reduce the size of the search space.

It appears to be always desirable to keep the size of the search space as small as possible. For example, it is more efficient to search in the sub space of active schedules rather than the space of all the semi-active schedules. The use of the critical block neighborhood reduces the size of the neighorhood. TS also reduces the neighborhood size by temporally eliminating some of the members in the neighborhood as "tabu". Likewise, the pruning pattern list introduced in Chapter 12 also provides a mechanism to reduce the neighborhood size by eliminating already searched and no longer interesting solutions from the neighborhood. In GAs, a redundant encoding of phenotype to genotype should be avoided because it increases the size of the search space in which more than one genotypes correspond to the same phenotype. A time-consuming gene decoding process as well as allowing fatal genotypes to be generated and later to be repaired must be avoided.

That said, a simple approach presented in Chapter 4 cannot be dismissed. Because it is implemented easily and sometimes it is more robust, especially in the real-world situation the objective function becomes more complicated and incorporating its domain specific knowledge into the search structure may become more difficult.

Another key feature is the hybrid of GAs and other local search methods such as SA and TS. It appears more promising to consider such hybrid rather than adhering to a single approach. In fact, GAs are known to be unsuited for fine-tuning structures which are very close to optimal solutions as opposed to SA and TS. On the other hand, SA and TS are inherently serial algorithms and not straight-forward to be parallelized. Thus, the MSXF method proposed in Chapter 9 as a

new approach for Genetic Local Search would be promising. In this framework, each individual, or search agent in the population, performs local search using SA and/or TS as a main search engine. Whereas crossover operates occasionally on the solutions of two selected individuals in the population and produces a new solution, which is then used as an initial solution for the subsequent local search. Here, the role of the crossover is to exchange informations between the search agents which otherwise perform independent local search in parallel, rather than to perform search itself as in the conventional GAs. One promising future direction might be to consider multiple parents in the MSXF.

Recently, the factory automation has been so advanced that each machine has self diagnostic sensors as well as a network interface and is monitored online even remotely. The flexibility and the accuracy of controlling machines for reconfiguration and rescheduling are also improved. Therefore, the need for generating an efficient and fine-tuned schedule in reasonable time is more envisaged. However unfortunately, the jobshop and flowshop scheduling problems investigated in this thesis might be too simplistic compared to the real-world problems that have more complicated constraints, more flexible objective functions and more dynamic features. It is highly expected to extend these approaches to incorporate more realistic settings. The author believes that the ideas presented in this thesis, at least some of them, will be helpful for future research.

Bibliography

[1] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, Vol. 3, No. 2, pp. 149–156, 1991. (document), 1.1, 2.8, 2.6, 7.6, 8.3.2, 9.4, 9.4.2

[2] B. Giffler and G.L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, Vol. 8, pp. 487–503, 1960. 1.1, 2.2

[3] C. Fischer and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In *Industrial Scheduling*, pp. 225–251, Englewood Cliffs, N.J., 1963. Prentice-Hall. 1.1

[4] J.F. Muth and G.L. Thompson. *Industrial Scheduling*. Prentice-Hall, Englewood Cliffs, N.J., 1963. 1.1, 2.8, 7.6, 8.3.1, 9.4

[5] G.H. Brooks and C.R. White. An algorithm for finding optimal or near optimal solutions to the production scheduling problem. *The Journal of Industrial Engineering*, Vol. 16, No. 1, pp. 34–40, 1969. 1.1

[6] E. Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations Research*, Vol. 17, pp. 941–957, 1969. 1.1, 6.3

[7] J.R. Barker and G.B. McMahon. Scheduling the general job-shop. *Management Science*, Vol. 31, No. 5, pp. 594–598, 1985. 1.1

[8] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, Vol. 35, No. 2, pp. 164–176, 1989. 1.1

[9] P. Brucker, B. Jurisch, and B. Sievers. A branch & bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, Vol. 49, pp. 107–127, 1994. 1.1, 2.5, 6.3, 7.6

[10] P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *the 5th International IPCO Conference*, pp. 389–403, 1996. 1.1

[11] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, Vol. 78, pp. 146–161, 1994. 1.1

[12] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, Vol. 40, No. 1, pp. 113–125, 1992. 1.1, 2.3, 6.3, 7.5.2, 7.5.2, 7.6

[13] H. Matsuo, C.J. Suh, and R.S. Sullivan. A controlled search simulated annealing method for the general jobshop scheduling problem. *Department of Management, The University of Texas at Austin*, Vol. Working Paper, 03-04-88, , 1988. 1.1, 7.6, 7.7

[14] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, Vol. 34, No. 3, pp. 391–401, 1988. 1.1, 2.3, 2.6, 2.8, 7.6, 8.2

[15] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, Vol. 1, No. 3, pp. 190–206, 1989. 1.1

[16] F. Glover. Tabu search - part 2. *ORSA Journal on Computing*, Vol. 2, No. 1, pp. 4–32, 1990. 1.1

[17] E.D. Taillard. Parallel taboo search techniques for the job-shop scheduling problem. *ORSA Journal on Computing*, Vol. 6, No. 2, pp. 108–117, 1994. 1.1, 2.3, 6.3, 8.3.2

[18] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, Vol. 41, pp. 231–252, 1993. 1.1, 2.5, 9.4.2

[19] E. Nowicki and C. Smutnicki. A fast tabu search algorithm for the job-shop problem. *Management Science*, Vol. 42, pp. 797–813, 1996. 1.1, 2.5, 9.4.2

[20] E. Nowicki and C. Smutnicki. A fast tabu search algorithm for the permutation flow-shop problem. *European Jouranl of Operational Research*, Vol. 91, pp. 160–175, 1996. 1.1, 10.2, 10.6, 12.2

[21] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, Vol. 64, pp. 278–285, 1993. 1.1, 10.4, 11.4, 12.5

[22] J.H. Holland. *Adaptation in Natural and Artificial Systems*. Unuv. of Michigan Press, 1975. 1.1

[23] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading,Mass., 1986. 1.1

[24] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In *Proceedings of International Conference on Genetic Algorithms (ICGA '91)*, pp. 474–479, 1991. 1.1, 2.4, 2.4, 4, 4.2

[25] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In *Proceedings of The Second International Conference on Parallel Problem Solving from Nature PPSN '92*, pp. 281–290, 1992. 1.1, 5, 5.3

[26] N.L.J. Ulder, E. Pesch, P.J.M. van Laarhoven, J. Bandelt, H, and E.H.L. Aarts. Genetic local search algorithm for the traveling salesman problem. In *Parallel Problem Solving from Nature, 1*, pp. 109–116, 1994. 1.1, 9

[27] S. S. Panwalkar and Wafix Iskander. A survey of scheduling rules. *Operations Research*, Vol. 25, No. 1, pp. 45–61, 1977. 2.2

[28] U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers Ops Res*, Vol. 22, pp. 25–40, 1995. 2.2

[29] B. Roy and B. Sussmann. Les probl'emes d'ordonnancement avec contraintes disjonctives. *Note DS no 9 bis, SEMA, Paris*, 1964. 2.3

[30] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, Berlin, 1995. 2.5

[31] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, Vol. 11, pp. 42–47, 1982. 2.7, 2.7

[32] S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, 1984. 2.8, 7.6

[33] S. Kobayashi, I. Ono, and M. Yamamura. An efficient genetic algorithm for job shop scheduling problems. In *Proceedings of 6th International Conference on Genetic Algorithms and their Applications (Pittsburgh, PA)*, pp. 506–511, 1995. 5.1

[34] I. Yamamura, M. Ono and S. Kobayashi. Character-preserving genetic algorithms for traveling salesman problem (in japanese). *Journal of Japanese Society for Artificial Intelligence*, Vol. 7, pp. 1049–1059, 1992. 5.1

[35] C. Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *OR Spektrum*, Vol. 17, pp. 87–92, 1995. 5.2

[36] C. Bierwirth, D. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems. In *Parallel Problem Solving from Nature, 4*, pp. 310–318, 1996. 5.2

[37] Y. Davidor, T. Yamada, and R. Nakano. The ecological framework II: Improving ga performance at virtually zero cost. In *Proceedings of 5th International Conference on Genetic Algorithms and their Applications (Urbana - Champaign)*, pp. 171–176, 1993. 5.3

[38] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman and Company, New York, 1979. 6

[39] C.R. (ed.) Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, Oxford, UK; re-issued by McGraw-Hill, London, UK., 1995. 6

[40] C.R. Reeves. Heuristic search methods: A review. In *D.Johnson and F.O'Brien Operational Research: Keynote Papers*, pp. 122–149. Operational Research Society, Birmingham, UK, 1996. 6

[41] D.S. Johnson. Local optimization and the traveling salesman problem. In *G.Goos and J.Hartmanis (Eds.) Automata, Languages and Programming, Lecture Notes in Computer Science 443*, pp. 446–461. Springer-Verlag, Berlin, 1990. 6.2

[42] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, Vol. 21, No. 6, pp. 1087–1092, 1953. 7.1

[43] P. D. Wasserman. *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, New York, 1989. 7.1

[44] S. Kirkpatrick, C.D.Jr. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, Vol. 220, pp. 671–680, 1983. 7.1

[45] L. Ingber and B E.Rosen. Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical and Computer Modeling*, Vol. 16, No. 11, pp. 87–100, 1992. 7.3

[46] B. E. Rosen. Function optimization based on advanced simulated annealing. *Workshop on Physics and Computation, PhysComp 92*, 1992. 7.3

[47] E. H. L. Aarts and J. H. M. Korst. *Simulated Annealing and Boltzmann machines*. Wiley, Chichester, 1989. 7.4, 7.7

[48] D. Applegate. Jobshop benchmark problem set. Personal Communication, 1992. 3

[49] J. Carlier. Ordonnancements à contraintes disjonctives. *RAIRO*, Vol. 12, pp. 333–351, 1978. 7.6

[50] T. Yamada, B.E. Rosen, and R. Nakano. A simulated annealing approach to job shop scheduling using critical block transition operators. In *Proceedings of IEEE International Conference on Neural Networks (Orlando, Florida.)*, pp. 4687–4692, 1994. 8.3.1

[51] E.H.L. Aarts, P.J.M. van Laarhoven, J.K. Lenstra, and N.L.J. Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, Vol. 6, No. 2, pp. 118–125, 1994. 8.3.2, 9.4.2

[52] H. Kopfer D.C. Mattfeld and C. Bierwirth. Control of parallel population dynamics by social-like behavior of ga-individuals. In *Parallel Problem Solving from Nature, 3*, 1994. 8.3.2, 9.4.2

[53] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. Technical report, Eindhoven University of Technology, Dpt. of Math. and CS, February 1994. 8.3.2

[54] C. R. Reeves. Genetic algorithms and neighbourhood search. In *Evolutionary Computing, AISB Workshop (Leeds, U.K.)*, pp. 115–130, 1994. 9.1

[55] T. Yamada and R. Nakano. A genetic algorithm with multi-step crossover for job-shop scheduling problems. In *Proceedings of the 1st IEE/IEEE International Conference on Genetic ALgorithms in Engineering Systems (GALESIA '95)*, pp. 146–151, 1995. 9.1

[56] T. Yamada and R. Nakano. Scheduling by genetic local search with multi-step crossover. In *Proceedings of The Fourth International Conference on Parallel Problem Solving from Nature (PPSN '96)*, pp. 960–969, 1996. 9.1

[57] T. Yamada and R. Nakano. A fusion of crossover and local search. In *Proceedings of IEEE International Conference on Industrial Technology*, pp. 426–430, 1996. 9.1

[58] D. Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of 3rd International Conference on Genetic Algorithms and their Applications (Arlington,VA)*, pp. 116–121, 1989. 9.3

[59] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of 3rd International Conference on Genetic Algorithms and their Applications (Arlington,VA)*, pp. 2–9, 1989. 9.3

[60] J. E. Beasley. Or-library: distributing test problems by electronic mail. *European Journal of Operational Research*, Vol. 41, pp. 1069–1072, 1990. 9.4.2, 10.6

[61] C. R. Reeves and C. Höhn. Are long path problems hard for genetic algorithms? In *4th International Conference on Parallel Problem Solving from Nature*, pp. 134–153, 1996. 10.4

[62] K. D. Boese, A. B. Kahng, and S. Muddu. A new adaptive multi-start technique for combinatorial global optimization. *Operations Research Letters*, Vol. 16, pp. 101–113, 1994. 10.4

[63] C. R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, Vol. to appear, , 1998. 10.4

[64] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for gas. In *Proceedings of 6th International Conference on Genetic Algorithms and their Applications (Pittsburgh, PA)*, pp. 184–192, 1995. 10.4

[65] J. Liu. A new heuristic algorithm for csum flowshop scheduling problems. Personal Communication, 1997. 11.4

[66] T. Yamada. *A pruning pattern list approach to the permutation flowshop scheduling problem*. Kluwer academic publishers, MA, USA, 2002. 12.1

[67] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings of 3rd International Conference on Genetic Algorithms and their Applications (Arlington,VA)*, pp. 133–140, 1989. 12.4

[68] T. Yamada and C.R. Reeves. Permutation flowshop scheduling by genetic local search. In *Proceedings of the 2nd IEE/IEEE International Conference on Genetic ALgorithms in Engineering Systems (GALESIA '97)*, pp. 232–238, 1997. 12.5

[69] C.R. Reeves and T. Yamada. Genetic algorithms, path relinking and the flowshop sequencing problem. *Evolutionary Computation journal (MIT press)*, Vol. 6, No. 1, pp. 45–60, 1998. 12.5

A List of Author's Work

Journals

1. T.Yamada, B.E.Rosen and R.Nakano, "Critical Block Simulated Annealing for Job Shop Scheduling (in Japanese)," *The Transaction of The Institute of Electrical Engineers of Japan*, Vol.114-C, No.4, pp.476–482 (1994).

2. T.Yamada and R.Nakano, "Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search (in Japanese)," *Transactions of Information Processing Society of Japan*, Vol.37 No.4, pp. 597–604 (1996).

3. T.Yamada and R.Nakano, "Job-Shop Scheduling by Genetic Local Search (in Japanese)," *Transactions of Information Processing Society of Japan*, Vol.38 No.6, pp. 1126–1138 (1997).

4. T.Yamada and C.R.Reeves, "Landscape Analysis of the Flowshop Scheduling Problem and Genetic Local Search (in Japanese)," *Transactions of Information Processing Society of Japan*, Vol.39 No.7, pp. 2112–2123 (1998).

5. C.R.Reeves and T.Yamada, "Genetic Algorithms, Path Relinking and the Flowshop Sequencing Problem," *Evolutionary Computation journal*, MIT press, Vol.6 No.1, pp.45–60, Spring (1998).

Books

1. T.Yamada and R.Nakano, "Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search," in *Meta-Heuristics: Theory & Applications*, Kluwer academic publishers, pp. 237–248 (1996).

2. T.Yamada and R.Nakano, "Chapter 7: Job Shop Scheduling," in *Genetic algorithms in Engineering Systems*, The Institution of Electrical Engineers, pp.134–160 (1997).

3. C.R.Reeves and T.Yamada, "Goal-Oriented Path Tracing Methods," in *New Ideas in Optimization*, The MacGraw-Hill Companies, pp.341–355 (1999).

4. T.Yamada "A Pruning Pattern List Approach to the Permutation Flowshop Scheduling Problem," in *Essays and Surveys in Metaheuristics*, Kluwer academic publishers, pp. 641–651 (2002).

International Conferences

1. R.Nakano and T.Yamada, "Conventional Genetic Algorithm for Job Shop Problems," *Proceedings of International Conference on Genetic Algorithms (ICGA '91)*, pp.474–479 (1991).

2. T.Yamada and R.Nakano, "A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems," *Proceedings of The Second International Conference on Parallel Problem Solving from Nature (PPSN '92)*, pp.281–290 (1992).

3. Y.Davidor, T.Yamada and R.Nakano, "The ECOlogical Framework II: Improving GA Performance At Virtually Zero Cost," *Proceeding of International Conference on Genetic Algorithms (ICGA '93)*, pp.171–176 (1993).

4. T.Yamada, B.E.Rosen and R.Nakano, "A Simulated Annealing Approach to Job Shop Scheduling using Critical Block Transition Operators," *Proceedings of IEEE International Conference on Neural Networks (ICNN '94)*, pp.4687–4692 (1994).

5. T.Yamada and R.Nakano, "A Genetic Algorithm with Multi-Step Crossover for Job-Shop Scheduling Problems," *Proceedings of the 1st IEE/IEEE International Conference on Genetic ALgorithms in Engineering Systems (GALESIA '95)*, pp.146–151 (1995).

6. T.Yamada and R.Nakano, "Scheduling by Genetic Local Search with Multi-Step Crossover," *Proceedings of The Fourth International Conference on Parallel Problem Solving from Nature (PPSN '96)*, pp.960–969 (1996).

7. T.Yamada and R.Nakano, "A Fusion of Crossover and Local Search," *Proceedings of IEEE International Conference on Industrial Technology*, pp.426–430 (1996).

8. T.Yamada and C.R.Reeves, "Permutation Flowshop Scheduling by Genetic Local Search," *Proceedings of the 2nd IEE/IEEE International Conference on Genetic ALgorithms in Engineering Systems (GALESIA '97)*, pp. 232–238, 1997.

9. T.Yamada and C.R.Reeves, "Solving the Csum Permutation Flowshop Scheduling Problem by Genetic Local Search", *Proceedings of 1998 IEEE International Conference on Evolutionary Computation*, pp.230–234 (1998).

10. C.R.Reeves and T.Yamada, "Implicit Tabu Search Methods for Flowshop Sequencing," *Proceedings of IMACS International Conference on Computational Engineering in Systems Applications*, pp.78–81 (1998).

11. T. Yamada, K. Yoshimura and R. Nakano, "Information Operator Scheduling by Genetic Algorithms," Simulated Evolution and Learning: *Proceedings of the Second Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'98)*, Lecture Notes in Computer Science 1585, pp. 50–57 (1999).