STUDIES ON METAHEURISTIC ALGORITHMS FOR COMBINATORIAL OPTIMIZATION PROBLEMS

Mutsunori YAGIURA

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

KYOTO UNIVERSITY
KYOTO, JAPAN
JANUARY, 1999

Preface

There are numerous combinatorial optimization problems, for which computing exact optimal solutions is computationally intractable, e.g., those problems known as *NP-hard*. However, in practice, we are often asked to deal with large scale instances of such difficult problems. One possibility to overcome this difficulty is that, in most practical cases, we do not need exact optimal solutions and are satisfied with sufficiently good solutions. In this sense, *approximate* (or *heuristic*) *algorithms*, which provide reasonably good solutions in practically meaningful time, are very important and have been well studied recently.

There are several useful tools used to design approximate algorithms, such as *greedy* method and *local search*. The so-called *metaheuristics* combine these tools into more sophisticated algorithms. Among the well-known metaheuristics are *multi-start local search*, *simulated annealing*, *tabu search*, *genetic algorithm* and so on. Many variants of these, such as *GRASP*, *threshold accepting*, *iterated local search* and others, have also been proposed and extensively studied.

One of the attractive features of these metaheuristics is in its flexibility. They can be hybridized with other heuristic or exact algorithms to create more powerful tools. As an example of such hybrid algorithms, we propose to use dynamic programming (DP) to improve candidate solutions within the framework of genetic algorithm, which is called the *genetic DP* algorithm. Good prospects of the proposed algorithm are observed by the computational experiments to three representative NP-hard problems: single machine scheduling problem, optimal linear arrangement problem and traveling salesman problem.

During the experience of developing the genetic DP, we realized that crossover is one of the most important operators in genetic algorithms, on which the overall performance of the algorithms critically depends. To pursue this direction, we review a variety of crossover operators proposed for sequencing problems, and analyze the relationship between characteristics of the operator and performance of the algorithm. Based on this analysis, we propose simple criteria for measuring the quality of crossover operators. Some computational analysis on single machine scheduling problem is then added to validate the effectiveness of the proposed criteria.

Another attractive feature of metaheuristics is in its robustness and simplicity. They can be developed even if deep mathematical properties of the problem domain are not at hand, and still can provide reasonably good solutions, much better than those obtainable by simple heuristics. To investigate this direction, we compare representative metaheuristic algorithms using rather simple inner operators to observe general tendency of their performance. From these results, we propose a recommendation about the use of metaheuristics as simple optimization tools.

We then consider a problem arising from the implementation issue of a crossover operator. Three types of fast algorithms are proposed, and analyses of these algorithms and of the problem structure are given.

The main aim of this thesis is to establish a guideline to construct good metaheuristic algorithms. The author hopes that the research in this dissertation will help advance the understanding of this significant field.

January, 1999
Mutsunori Yagiura

Acknowledgement

This thesis would not have been possible without the help of many others. First of all, I am heartily grateful to Professor Toshihide Ibaraki of Kyoto University for his enthusiastic guidance, discussion and persistent encouragement. He commented in detail on the whole work in the manuscript, which significantly improved the accuracy of the arguments and quality of the exposition. Without his considerable help, none of this work could have been completed.

A great deal of gratitude goes to Professor Tatsuya Akutsu of University of Tokyo, Hajime Ase of NKK Corporation, Professor Fred Glover of University of Colorado, Nobuyuki Kawai of NKK Corporation, and Professor Takeaki Uno of Tokyo Institute of Technology, with whom I wrote joint papers. Their great collaboration was very important in forming the idea of this thesis.

I am also indebted to Professor Masao Fukushima of Kyoto University, Professor Masamitsu Ohnishi of Osaka University, and Professor Hiroshi Nagamochi of Kyoto University for a number of helpful suggestions. I also wish to express my gratitude to Professor Yushi Uno of University of Osaka Prefecture, Professor Satoru Ibaraki of Nagoya City University, Professor Kouichi Taji of Osaka University, Professor Yoshiyuki Karuno of Kyoto Institute of Technology, Professor Dao-Zhi Zeng of Kagawa University, Professor Kazuhisa Makino of Osaka University, and all members in Professor Ibaraki's laboratory for many enlightening discussions on the area of this work. I especially mention two members with whom I wrote joint papers, namely Shinji Katoh and Takashi Yamaguchi.

My deepest gratitude is to my parents for their heartfelt cooperation and encouragement. Finally, I must acknowledge my wife, Mariko. Her cooperation and steady encouragement were essential especially in the period when I studied at Kyoto University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Historical Background

The optimization problems we consider in this thesis are generally defined as follows:

$$\begin{array}{ll} \text{minimize} & cost(\sigma) \\ \text{subject to} & \sigma \in F. \end{array} \tag{1.1.1}$$

F is the set of solutions σ that satisfy all the constraints. F is called the *feasible region* and each $\sigma \in F$ is called a *feasible solution.* A feasible solution $\sigma^* \in F$ is *optimal* if $cost(\sigma^*) \leq cost(\sigma)$ holds for all $\sigma \in F$, and $cost(\sigma^*)$ is called the *optimal value.* When F is combinatorial in some sense, we call problem (1.1.1) a *combinatorial optimization* problem.

Combinatorial optimization problems frequently appear in the real-world such as machine scheduling, vehicle routing, and their importance has widely been recognized in recent years. Many of such combinatorial optimization problems are computationally intractable, e.g., those problems known to be *NP-hard* [38]. However, in practice, we are often asked to deal with large scale instances of such difficult problems. One possibility to overcome this seemingly impossible difficulty is that, in most practical cases, we do not need exact optimal solutions and are satisfied with sufficiently good solutions. In this sense, *approximate* (or *heuristic*) *algorithms*, which provide reasonably good solutions in practically meaningful time, are very important and have been intensively studied recently.

There are several useful tools used to design approximate algorithms. The most common one is perhaps the *greedy* method [75, 95], which directly constructs approximate solutions by successively determining the values of variables on the basis of some local information. Another important tool is the *local search* [3, 95], which starts from an initial feasible solution σ and repeats replacing it with a better solution in its *neighborhood* $N(\sigma)$ until no better solution is found in $N(\sigma)$, where $N(\sigma)$ is a set of solutions obtainable from σ by a slight perturbation.

The so-called *metaheuristics* [3, 92, 103, 104] combine these tools into more sophisticated algorithms. Among the well-known metaheuristics are *multi-start local search* [68, 78], *simulated annealing* [1, 16, 69], *tabu search* [41, 44], *genetic algorithm* [22, 47, 58] and so on. Multi-start local search applies the local search to a number of initial solutions and outputs the best solution found during the entire search. Simulated annealing and tabu search try to enhance the local search by allowing the replacement of the current solution σ with a worse solution in $N(\sigma)$ thereby avoiding to be trapped into bad local optimals. The genetic algorithm is a probabilistic algorithm that simulates the evolution process, by repeating the operations such as crossover, mutation and selection. An important feature of this algorithm is that it keeps P (≥ 1) candidate solutions and improve them in the process of evolution. Many variants, such as *GRASP* (*greedy randomized adaptive search procedure*) [31, 32], *threshold accepting* [28], *iterated local search* [63, 82], *genetic local search* [71, 89, 126] and so on, have also been proposed and extensively studied. These algorithms are summerized in Chapter 2.

1.2 Research Objectives and Outline of the Thesis

One of the attractive features of these metaheuristics is in its flexibility. They can be hybridized with other heuristic or exact algorithms to make them more powerful. As an example of such hybrid algorithms, we propose to use *dynamic programming* in the process of obtaining new generation solutions in the genetic algorithm, and call it a *genetic DP* algorithm. To evaluate the effectiveness of this approach, we choose three representative combinatorial optimization problems, the single machine scheduling problem (SMP), the optimal linear arrangement problem (OLAP) and the traveling salesman problem (TSP), all of which ask to compute optimum permutations of n objects and are known to be NP-hard. Computational experiments of genetic DP algorithms are conducted to compare them with exact algorithms, the conventional genetic algorithms and multi-start local search algorithms. Algorithms of genetic DP could obtain optimal solutions to 47 out of 50 SMP instances with up to $n = 35$ jobs, and 23 out of 24 OLAP instances with up to $n = 20$ components, in a very short time compared to the exact algorithms. They also exhibit superiority to other meta-heuristics such as multi-start local search algorithms and genetic local search algorithms. However, in the case of TSP, the Lin-Kernighan heuristic [78] exhibits much better performance than all others including genetic DP algorithm.

During the experience of developing the genetic DP, we realized that crossover is one of the most important operators in genetic algorithms, on which the overall performance of the algorithms critically depends. To pursue this direction, we review a number of crossover operators proposed so far for sequencing problems. We then consider a general framework of crossover operators and analyze the relationship between characteristics of the operator and performance of the algorithm. Based on this analysis, we propose simple criteria for

measuring the quality of crossover operators. Computational experiments for the single machine scheduling problem (SMP) using a simple framework of GA is conducted, and it is observed that the following two criteria are important for crossover operators: (1) inherit as many elements as possible from the parents, and (2) keep the diversity of children obtained from the parents.

Another attractive feature of metaheuristics consists in its robustness and simplicity. They can be developed even if deep mathematical properties of the problem domain are not at hand, and still can provide reasonably good solutions, much better than those obtainable by simple heuristics. We pursue this direction more carefully, by implementing various metaheuristics and comparing their performance. The objective is not to propose the most powerful algorithm but to compare general tendencies of various algorithms. The emphasis is placed not to make each ingredient of such metaheuristics too sophisticated, and to avoid detailed tuning of the program parameters involved therein, so that practitioners can easily test the proposed framework to solve their problems of applications. As a concrete problem to test, we choose the single machine scheduling problem (SMP). The results indicate that: (1) MLS is usually good enough for practical purposes, considering its simplicity, (2) a variant of MLS, called GRASP, is effective; however, its performance is sensitive to greedy methods used to generate initial solutions, (3) a variant of MLS, called iterated local search, is quite effective, (4) GA combined with local search is also competitive if longer computational time is allowed, and its performance is not sensitive to crossovers, (5) SA (and its variants called the threshold accepting and the great deluge algorithm) is another competitive method assuming that longer computational time is allowed, and its performance is not much dependent on inside parameter values, (6) there are cases in which TS is more effective than MLS; however, its performance depends on how to define the tabu list and parameter values, and (7) the definition of neighborhood is critical for all of the tested algorithms except GA. These results lead to a simple description of the guideline for designing metaheuristic algorithms.

We then consider a problem arising from the implementation issue of a crossover operator. One of the crossover operators proposed for sequencing problems includes the following problem: Given two permutations of n elements, enumerate all pairs of intervals consisting of the same set of elements. We call this problem as the *common interval enumeration problem*, and propose three types of fast algorithms: i) a simple $O(n^2)$ time algorithm (LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations of n elements, ii) a practically fast $O(n^2)$ time algorithm (MNG) using the reverse Monge property, and iii) an $O(n+K)$ time algorithm (RC), where K $(\leq \binom{n}{2})$ is the number of common intervals. It will be also shown that the expected number of common intervals for two random permutations is $O(1)$. This result gives a reason for the phenomenon that the expected time complexity $O(n)$ of the algorithm LHP is independent of K. Among the proposed algorithms, RC is most desirable from the theoretical point of view; however, it is quite complicated compared to LHP and MNG. Therefore, it is possible that RC is slower than the other two algorithms in some cases. For this reason, computational experiments for various types of problems with up to $n = 10^6$ are conducted. The results indicate that i) LHP and MNG are much faster than RC for two randomly generated permutations, and ii) MNG is rather slower than LHP for random inputs; however, there are cases that LHP requires $\Omega(n^2)$ time, but MNG runs in $o(n^2)$ time and is faster than both LHP and RC. We also consider the enumeration of all *common subtrees*, i.e., given two trees with labels on their leaves find the pairs of subtrees having the same set of leaf labels. This problem has an application in constructing evolutionary trees. By using the algorithm RC, we can derive a fast randomized algorithm with $O(n \log^2 n)$ expected running time if we are given two binary trees of depth $\log_2 n$, where n is the number of leaves. The expected running time becomes $O(n)$ if the same two binary trees of depth $\log_2 n$ are given as the input. The latter special case is a trivial instance; however, this case is intuitively considered to be tough for this algorithm, and hence, it is expected that the proposed algorithm runs in $O(n)$ expected time for most of the practical instances, although the worst case running time is $O(n^2)$.

The thesis is organized as follows. In Chapter 2, we review various metaheuristic algorithms. In Chapter 3, the genetic DP, in which the genetic algorithms and the dynamic programming are combined, is proposed and computational results are shown. In Chapter 4, various crossover operators are compared and simple criteria for measuring the quality of crossover operators are proposed. In Chapter 5, various metaheuristic algorithms are compared and a guideline for the use of metaheuristic algorithms is discussed. In Chapter 6, three algorithms for the common interval enumeration problem are proposed. Finally, in Chapter 7, we summarize our study in this thesis and list the contribution of our study. The importance of metaheuristic algorithms is evident, as the sizes of the real-world problem instances are always increasing. The author hopes that the work in this thesis will be helpful to make metaheuristic algorithms more effective.

Chapter 2

Metaheuristic Algorithms: An Overview

2.1 Metaheuristics

In this chapter, we describe frameworks of various metaheuristic algorithms. For simplicity, we restrict our attention to the problems whose feasible solutions are easily obtained. The basic frameworks of metaheuristics are the same for those problems whose feasible solutions are not easily obtained, but we need some slight modifications. For example, we often allow the search into the infeasible region and add a penalty term to the cost (1.1.1) to evaluate the degree of infeasibility. Some other approaches are possible, but we omit them here.

Among basic strategies of approximate algorithms are

- greedy method,
- local search (LS).

The greedy method is a one-path algorithm that constructs a feasible solution step by step, on the basis of the effectiveness computed by a local evaluator. The idea of the greedy method may be best explained by examples. Some examples of the greedy methods for sequencing problems, SMP, OLAP and TSP, are explained in Sections 3.3.2, 3.4.1 and 3.5.1, respectively.

The LS starts from an initial solution σ and repeats replacing σ with a better solution in its *neighborhood* $N(\sigma)$ until no better solution is found in $N(\sigma)$, where $N(\sigma)$ is a set of solutions obtainable by slight perturbations. The local search from an initial solution σ_0, in which the neighborhood N is used, is formally described as follows.

```
Algorithm $LS(N, \sigma_0)$
Step 0 Set $\sigma := \sigma_0$.
Step 1 If there is a feasible solution $\sigma' \in N(\sigma)$ such that $cost(\sigma') < cost(\sigma)$, set $\sigma := \sigma'$ and
       return to Step 1. Otherwise go to Step 2.
Step 2 ($cost(\sigma') \geq cost(\sigma)$ holds for all $\sigma' \in N(\sigma)$.) Output $\sigma$ and stop.
```

A solution σ is called *locally optimal*, if no better solution exists in $N(\sigma)$. We call the computation of obtaining a locally optimal solution from an initial σ_0 as a *trial* of LS, and call the replacement of the current solution σ by a better solution as a *move*. One of the following two move strategies are commonly used: *First admissible* move strategy (abbreviated as FA) and *best admissible* move strategy (abbreviated as BA). FA scans the neighborhood $N(\sigma)$ according to a prespecified random order and moves to the first improved solution. BA scans the entire neighborhood and move to the best solution in $N(\sigma)$.

In general, if only one trial of LS is applied, many solutions of better quality may remain unvisited. Therefore, LS may be enhanced by:

- trying many initial solutions,
- using a sophisticated neighborhood or a larger neighborhood,
- using a sophisticated search strategy, sometimes allowing moves to worse solutions in $N(\sigma)$.

Metaheuristics such as

- multi-start local search (MLS),
- genetic algorithm (GA),
- simulated annealing (SA),
- tabu search (TS)

can be viewed as such variants of LS. In the following sections, we briefly summarize these metaheuristic algorithms, along with their variants. More details are found in survey papers and books such as [3, 91, 92, 98, 99, 103, 104], and hybrid approaches (e.g., hybrids of two metaheuristics, hybrids of exact algorithms and metaheuristics, etc.) are summarized in [59]. Comparisons of metaheuristic algorithms are found in, e.g., [2, 20, 63, 117, 126].

2.2 Multi-Start Local Search

In the *multi-start local search* (MLS), LS is repeated from a number of initial solutions and the best solution found during the entire search is output. This is one of the most commonly

used techniques for combinatorial optimization problems [68, 78, 95]. The initial solution may be generated randomly or by using greedy methods. The MLS, in which initial solutions are generated randomly, is formally described as follows.

```
Algorithm MLS
Step 1 (initialize) Set $best := \infty$.
Step 2 (generate an initial solution) Generate a solution $\sigma$ randomly.
Step 3 (improve by LS) Improve $\sigma$ by LS, i.e., set $\sigma := LS(N, \sigma)$.
Step 4 (update the best cost) If $cost(\sigma) < best$, set $best := cost(\sigma)$ and $\sigma^* := \sigma$.
Step 5 (halt or random restart) If some stopping criterion is satisfied, output $\sigma^*$ and stop;
       otherwise return to Step 2.
```

In Step 5, various stopping criteria are possible. Among common ones are:

- stop if a prespecified computational time is reached,
- stop if a prespecified computational time is spent without improving *best*.

Some other measures, such as

- the number of repetitions of Steps 2 to 4,
- the number of moves,
- the number of cost evaluations,

are also commonly used instead of the computational time. These stopping criteria are also used in other metaheuristic algorithms.

The *greedy randomized adaptive search procedure* (GRASP) is a variant of MLS, in which the initial solutions are generated by randomized greedy methods. In the greedy method, a feasible solution is usually constructed step by step by choosing the element with the best evaluation. Although better initial solutions than random ones are usually obtained, the variety of solutions constructed by this method is quite limited, which is not preferable for MLS. To overcome this, in GRASP, a feasible solution is constructed by, in each step, randomly choosing an element from the candidate list CA composed of those elements with good local evaluations. The size $|CA|$ of the candidate list is a prespecified parameter. If $|CA| = 1$, the algorithm is equivalent to the ordinary greedy method. Some examples of GRASP for SMP will be examined in Chapter 5. In GRASP, it is expected that LS can start from good initial solutions while keeping the diversity of the search. The framework of GRASP is described as follows.

Algorithm GRASP

(Steps 1, 3, 4 and 5 are the same with MLS.)

Step 2 (generate an initial solution) Generate a solution σ by using randomized greedy method.

GRASP was proposed by Feo et al., e.g., [34], and applied to various combinatorial optimization problems by themselves and others, e.g., [30, 32, 33, 73, 76, 106]. The basic idea of GRASP has appeared in early papers such as [31, 55].

Another variant of MLS called the *iterated local search* (ILS) [53, 63] is also possible, where the initial solutions are generated by slightly perturbing a solution σ_{seed}, which is a good (not necessarily the best) solution found during the search.

Algorithm ILS

Step 1 (initialize) Set $best := \infty$ and generate a solution σ_{seed} randomly.

Step 2 (generate an initial solution) Generate a solution σ by slightly perturbing σ_{seed}.

Step 3 (improve by LS) Improve σ by LS, i.e., set $\sigma := LS(N, \sigma)$.

Step 4 (update the best and seed solutions) If $cost(\sigma) < best$, set $best := cost(\sigma)$ and $\sigma^* := \sigma$. If some accepting criterion is satisfied, set $\sigma_{seed} := \sigma$.

Step 5 (halt or random restart) If some stopping criterion is satisfied, output σ^* and stop; otherwise return to Step 2.

In Step 2, the new solution σ is usually generated by randomly choosing a solution in the neighborhood $N'(\sigma)$. For N', we can use the same neighborhood as LS (i.e., $N' = N$); however, the search may return to σ_{seed} by LS and cycling may occur, since the neighborhood is usually symmetric (i.e., $\sigma_a \in N(\sigma_b) \Leftrightarrow \sigma_b \in N(\sigma_a)$). To avoid this, a larger neighborhood (i.e., $|N'| > |N|$) or a different neighborhood is often used as N'. There is a variant of this, in which the neighborhood N' is gradually enlarged if the search fails to improve σ^*, and N' is reset to the original size (usually small) if σ^* is updated. Such variants are called *variable neighborhood search* algorithms [14, 15, 86]. Another variant is to generate σ in Step 2 by applying LS to σ_{seed}, in which a randomized cost function is used to evaluate solutions instead of the original cost. Such algorithms are called *noising method* or *perturbation* [17, 18, 121].

In Step 4, one of the simplest rules of accepting a new σ_{seed} is: Set $\sigma_{seed} := \sigma$ if $cost(\sigma) < best$ (i.e., $\sigma_{seed} = \sigma^*$). In [81, 82], a variant, called *chained local optimization*, is proposed. In this method, σ_{seed} is chosen randomly according to the following rule, whose idea is taken from the simulated annealing: If $cost(\sigma) < cost(\sigma_{seed})$, set $\sigma_{seed} := \sigma$; otherwise set $\sigma_{seed} := \sigma$ with probability $e^{-\Delta/t}$, where $\Delta = cost(\sigma) - cost(\sigma_{seed})$ and t is a prespecified parameter (t can be adaptively changed during the search).

2.3 Genetic Algorithm

The genetic algorithm (GA) is a probabilistic algorithm, whose idea comes from evolution. GA repeatedly applies the operations such as *crossover*, *mutation* and *selection* to the set of candidate solutions $\mathcal{P}$. This algorithm can be viewed as a generalization of LS, in which the neighborhood $N(\mathcal{P})$ is defined to be the set of solutions obtainable from $\mathcal{P}$ by crossover and mutation operators. A crossover operator generates one or more solutions (*children*) by combining two or more candidate solutions (*parents*), and a mutation operator generates a solution by slightly perturbing a candidate solution. The GA starts from an initial candidate solutions $\mathcal{P}$ and repeat replacing $\mathcal{P}$ with $\mathcal{P}' \subseteq \mathcal{P} \cup N(\mathcal{P})$ according to the selection rule.

Algorithm GA

(Positive integers P and Q are program parameters to be specified beforehand.)

Step 1 (initialize) Construct the set $\mathcal{P}$ of P initial candidate solutions. Let σ^* be the best solution among $\mathcal{P}$.

Step 2 (crossover and improve): Repeat the following steps (a) and/or (b) until the set $\mathcal{Q}$ of Q candidate solutions are obtained.

a (crossover) Crossover two or more candidate solutions to generate a new solution.

b (mutate) Mutate a candidate solution to generate a new solution.

Step 3 (update the best solution) If a solution σ with $cost(\sigma) < cost(\sigma^*)$ is found in Step 2, set $\sigma^* := \sigma$.

Step 4 (select) Select P solutions $\mathcal{P}'$ from the resulting $\mathcal{P} \cup \mathcal{Q}$, and set $\mathcal{P} := \mathcal{P}'$.

Step 5 (iterate) If some stopping criterion is satisfied, output σ^* and stop; otherwise return to Step 2.

GA was originally introduced by Holland [58]. For details, see [22, 47]. There is a recent survey by Reeves [105], in which various ideas and applications are discussed from the view point of "GA as a tool for operations researchers."

A variant of GA in which solutions generated by the crossover and mutation operators are improved by LS is called the *genetic local search* (GLS) [71, 89, 126]. GLS is different from MLS in that GLS generates the initial solutions from the current P by crossover and/or mutation, while MLS generates them randomly from scratch.

Algorithm GLS

(Steps 1, 3, 4 and 5 are the same as GA.)

Step 2 (crossover and improve): Repeat the following steps (a) and/or (b), and (c) until the set $\mathcal{Q}$ of Q candidate solutions are obtained.

a (crossover) Crossover two or more candidate solutions to generate a new solution.

b (mutate) Mutate a candidate solution to generate a new solution.

c (local search) Apply local search to the solution of (a) and/or (b) to obtain a locally optimal solution.

Early references such as [13, 22, 47, 61, 62, 84, 87, 89, 123, 126] have already mentioned the idea of GLS. Some other successful applications are found in [37, 71].

2.4 Simulated Annealing

This is a variant of LS, in which test solutions are randomly chosen from $N(\sigma)$ and accepted with probability that is 1 if the test solution is better than σ, and positive even if the test solution is worse than σ. By giving a positive probability to a move to a worse solution, the search is able to escape from poor locally optimal solutions. The acceptance probability is judiciously controlled by a parameter called *temperature*, whose idea stems from the physical annealing process.

Algorithm SA

Step 1 (initialize) Generate a solution σ, set $\sigma^* := \sigma$ and specify an initial temperature t.

Step 2 (check a neighborhood solution) Generate a solution $\sigma' \in N(\sigma)$ randomly, and set $\Delta := cost(\sigma') - cost(\sigma)$. If $\Delta < 0$ (i.e., a better solution is found), set $\sigma := \sigma'$; otherwise set $\sigma := \sigma'$ with probability $e^{-\Delta/t}$.

Step 3 (update the best cost) If $cost(\sigma) < cost(\sigma^*)$, set $\sigma^* := \sigma$.

Step 4 (halt or further search) If some stopping criterion is satisfied, output σ^* and stop; otherwise update t according to some rule and return to Step 2.

SA was proposed in [16, 69]. For details of SA, see [1]. Extensive computational results are found in the series of papers [64].

The *threshold accepting* (TA), originally introduced in [28], is a variant of SA. In TA, Step 2 of SA is replaced by

Step 2′ (check a neighborhood solution) Generate a solution $\sigma' \in N(\sigma)$ randomly, and set $\Delta := cost(\sigma') - cost(\sigma)$. If $\Delta < \tau$, set $\sigma := \sigma'$.

The parameter τ, called threshold, is controlled instead of the temperature t. Comparisons with other metaheuristics are found in [2, 77, 126].

There is another variant of SA, called *great deluge algorithm* (GDA), which was proposed in [27]. In GDA, Step 2 of SA is replaced by

Step 2″ (check a neighborhood solution) Generate a solution $\sigma' \in N(\sigma)$ randomly. If $cost(\sigma') < W$, set $\sigma := \sigma'$.

The parameter W, called water level, is controlled instead of the temperature t. Comparisons with other metaheuristics are found in [117].

Similar (but much simpler) approach is applied to the satisfiability problem, which is called the WALKSAT algorithm [115]. In this method, the algorithm either moves to the best solution, or to a solution randomly chosen, in the (randomly restricted) neighborhood.

2.5 Tabu Search

The tabu search tries to enhance LS by using the memory of the previous search. Basically the best solution in $N(\sigma)\backslash(\{\sigma\} \cup T)$ is chosen as the next solution, where the set T, called *tabu list* (or *short term memory*), is a set of solutions which includes those solutions most recently visited. Within this restricted neighborhood, a move to a new solution is always executed even if the current solution is already locally optimal, but cycling of a short period can be avoided as a result of introducing T. Another type of memory, called *long term memory*, is often employed in the framework of TS, which memorizes the past search information such as the frequency that each decision variable has been changed, the frequency that a solution has been visited and so on. This memory is used to direct the search to the unvisited region (i.e., *diversification*).

Algorithm TS

Step 1 (initialize) Generate a solution σ, set $\sigma^* := \sigma$ and $T := \emptyset$.

Step 2 (decide a move) Find the best solution σ' in $N(\sigma)\backslash(\{\sigma\} \cup T)$, and set $\sigma := \sigma'$.

Step 3 (update the best cost) If $cost(\sigma) < cost(\sigma^*)$, set $\sigma^* := \sigma$.

Step 4 (halt or further search) If some stopping criterion is satisfied, output σ^* and stop; otherwise update T according to some rule and return to Step 2.

TS was proposed in [41]. For details, see books and tutorials such as [42, 43, 44, 45].

2.6 Other Metaheuristic Algorithms

In this section, we briefly review some other metaheuristic algorithms: (1) variable depth search, (2) ant system and (3) guided local search.

The *variable depth search*, originally proposed in [68, 78] and introduced with this name in [95], is a generalization of local search, in which the neighborhood is defined to be the set of solutions obtainable by a sequence of simple neighborhood moves. This idea is slightly extended and studied with the name *ejection chain* in combination with the tabu search [26, 67, 74, 97]. Recently, we successfully applied the variable depth search to the generalized assignment problem [137, 138], which is one of the representative combinatorial optimization problems that is known to be NP-hard.

The *ant system* algorithm, originally introduced in [19, 24], is a randomized algorithm inspired by the behavior of ants. Ants are able to find good solutions to shortest path problems between a food source and their home colony by communicating via pheromones. If many ants choose a certain path and lay down pheromones, the intensity of the trails increases and thus this trail attracts more and more ants. This mechanism is imitated to store the information of good solutions found in the previous search, and to bias the later search to these promising directions. For details, see [25]. Combination with local search is also possible, and good prospects of such approaches are reported in [80, 122, 124]. Boese [12] proposed a similar (but much simpler) multi-start local search approach based on a different motivation.

The *guided local search* [102, 125, 128] is a variant of local search, in which solutions are evaluated with the modified cost based on the previous search information. In this method, the element (e.g., tour edge for the TSP) with the largest cost included in the locally optimal solution in the last trial is penalized in the next search so that different solutions are visited. This algorithm can be considered as a special case of TS, in which only the long term memory is used. Although the motivation is rather different and the algorithm is specific to a certain problem, similar idea is applied to the satisfiability problem [114], which is called the weighting strategy. (The objective of the satisfiability problem is to find a solution by which all the given clauses are satisfied.) In this method, the weights of unsatisfied clauses in the locally optimal solution of the previous trial are modified so that they get more chance to be satisfied in the next iteration.

2.7 Theoretical Results

Although not much is known about the theoretical aspects of metaheuristics, we briefly mention here some of such results.

From the view point of computational complexity, it is even not clear whether finding a

locally optimal solution is possible in polynomial time or not. To investigate this direction, Johnson et al. [65] proposed a complexity class called PLS (polynomial-time local search). Other related topics are found in [72, 94, 111], and a recent survey is in [141]. This research direction is quite important; however, not much attention is payed by practitioners, since the computational time of LS to find a locally optimal solution is usually small.

It is well-known that the search of SA converges to a global optimum under certain conditions, if the temperature is decreased very slowly, e.g., [79]. Similar result is also known for TA [7]. On the other hand, it is shown in [110] that exponential time is needed for such convergence of SA for the matching problem, for which efficient polynomial time algorithms exist. Therefore, these results do not give a support to the success of metaheuristic algorithms within limited amount of computational time, although they are quite interesting from the theoretical point of view.

It is shown in [6, 23] that, under certain conditions, a global optimum is found in polynomial time with high probability by a multi-point search called "go with the winners", which is a simplified search model proposed for the analysis and is similar to GA to some extent. The drawback of these results is that the conditions needed for the theorem to hold are rather unnatural. However, the algorithm itself is quite simple and its algorithmic (not theoretical) idea is applicable to many problems.

2.8 Conclusion

In this chapter, we briefly reviewed representative metaheuristic algorithms, such as multi-start local search (MLS), greedy randomized adaptive search procedure (GRASP), iterated local search (ILS), simulated annealing (SA) and tabu search (TS). We also mentioned some variants of them. There are many other approaches we did not mention in this chapter, e.g., multispace search [52, 54], incomplete construction/improvement [109], etc. However, it is very hard to cover all of known algorithms in this rapidly growing field.

We also did not explain details of each metaheuristic algorithm or hybrid approaches of them, which are often important to achieve fruitful results. The readers who would like to know more about metaheuristics, see, e.g., [3, 91, 92, 98, 99, 103, 104].

Chapter 3

The Use of Dynamic Programming in Genetic Algorithms

3.1 Introduction

An important feature of the genetic algorithm is that it keeps $P \geq 1$ candidate solutions and improve them in the process of evolution. Among various modifications [22, 118, 139, 142], it is reported in [61, 89, 126] that introducing local search technique in the evolution process, i.e., GLS explained in Section 2.3, is quite effective. This may suggest that combining some of other techniques with genetic algorithms is also worth trying.

In this chapter, we propose a variant of the genetic algorithm called *genetic DP* [129, 130, 134]. It uses *dynamic programming* (abbreviated as DP) to improve the candidate solutions. To evaluate the effectiveness of this approach, we choose three representative combinatorial optimization problems: the *single machine scheduling* problem (abbreviated as SMP) [60], the *optimal linear arrangement* problem (abbreviated as OLAP) [10] and the *traveling salesman* problem (abbreviated as TSP) [75], all of which are known to be NP-hard. The SMP asks to determine an optimal sequence of n jobs that minimizes a cost function defined for jobs, e.g., total weighted sum of earliness and tardiness. The OLAP asks to determine an optimal arrangement of n components in a straight line, which minimizes the total wire length needed for connecting all components in a prespecified manner. The TSP asks to find the shortest tour (i.e., a closed path that visits every city exactly once). These problems all ask to find an optimal permutation of n elements. The genetic DP can be applied to these optimization problems to find an optimal permutation of n elements.

Computational experiments of genetic DP algorithms are conducted to compare them with exact algorithms, the conventional genetic algorithms and multi-start local search algorithms. Algorithms of genetic DP could obtain optimal solutions to 47 out of 50 SMP instances with up to $n = 35$ jobs, and 23 out of 24 OLAP instances with up to $n = 20$ components, in a very short time compared to the exact algorithms. They also exhibit superiority to other meta-heuristics such as multi-start local search algorithms and genetic local search algorithms. However, in the case of TSP, the Lin-Kernighan heuristic [78] exhibits much better performance than all others including genetic DP algorithm.

From these results, we can conclude that genetic DP is one of the most powerful meta-heuristics useful for general combinatorial optimization problems, though it does not exclude the possibility that some heuristics specialized to the given problem, such as Lin-Kernighan algorithm, may turn out to be the winner.

3.2 Genetic DP Algorithm

In place of crossover and local search in Step 2 of GLS, genetic DP applies dynamic programming (DP) in order to generate a new solution from given two candidate solutions. It is primarily considered for the problem of finding optimum permutations (though it can be generalized to other types of optimization problems). The general framework of genetic DP is first described, and then each step is explained more in details.

Algorithm GENETIC DP

(Positive integers P and Q are program parameters to be specified beforehand)

Step 1 (Initialize): Construct P initial candidate solutions.

Step 2 (Crossover by DP and Improve): Get Q candidate solutions by repeating the following steps, where step (Mutate) is optional.

(Crossover): Pick up two candidate solutions and compute the partial order D common to both solutions.

(Mutate): Perturb the obtained D randomly.

(DP): Apply dynamic programming (DP) to the resulting D to obtain the best solution that does not violate D. Add the obtained solution to the set of candidate solutions unless it is already in the set.

Step 3 (Select): Select P solutions from the resulting $P + Q$ solutions.

Step 4 (Iterate): Repeat Steps 2 to 3 until some stopping criterion is satisfied.

Step 2 (Crossover by DP)

Let a solution be a permutation $\sigma = \langle \sigma(1), \dots, \sigma(n) \rangle$, i.e., an ordered sequence of n different elements, chosen from set $\{1, 2, \dots, n\}$, where $\sigma(i)$ denotes the i-th element in the sequence

and $\sigma^{-1}(j)$ denotes the location of element j. Denote the two candidate solutions picked up in Step 2 by σ_1 and σ_2. The partial order common to σ_1 and σ_2 is defined by

$$D = \{(i,j) \mid \sigma_1^{-1}(i) \le \sigma_1^{-1}(j) \text{ and } \sigma_2^{-1}(i) \le \sigma_2^{-1}(j)\}. \tag{3.2.1}$$

The idea of crossover by DP is based on the fact that good solutions tend to have a lot of common structure. For example, it is reported in Lin and Kernighan [78] that about 85% pairs of cities on the average are commonly adjacent in two tours obtained by using their algorithm, and 60 ~ 80% pairs are commonly adjacent in 7 or 8 such tours. They succeeded in speeding up their algorithm by fixing such pairs common in k tours (k is a given positive integer) without greatly losing the solution quality. In our formulation, the common partial order D of given two candidate solutions is a description of the common structure. By using DP (details of its computation will be described later in Sections 3.3.2, 3.4.1 and 3.5.1 as it depends on the particular problem being solved), it is possible to compute the best solution consistent with D.

Here we give an example of a partial order D. For two solutions $\sigma_1 = \langle 1, 2, 3, 4, 5, 6 \rangle$ and $\sigma_2 = \langle 2, 3, 1, 5, 4, 6 \rangle$, D is given by

$$\begin{aligned} D = \{&(1,1),(1,4),(1,5),(1,6),\\ &(2,2),(2,3),(2,4),(2,5),(2,6),\\ &(3,3),(3,4),(3,5),(3,6),\\ &(4,4),(4,6),\\ &(5,5),(5,6),\\ &(6,6)\}. \end{aligned} \tag{3.2.2}$$

A partial order can be represented by a directed graph, where a vertex represents an element and an arc represents an order. In such a graph, arcs for

$$\{(i,i) \in D\} \cup \{(i,k) \in D \mid \exists j \text{ such that } (i,j) \in D \text{ and } (j,k) \in D\}$$

are omitted. The graph representing the above D is given in Figure 3.1.

Step 2 (Mutate)

In our computational experiment, we realized the mutation by randomly perturbing the common partial order D before applying DP computation. After trying several, we employed the following method of perturbation: randomly choose a pair $i, j \in V$ such that $\sigma_1^{-1}(i) < \sigma_1^{-1}(j)$, and relax D by one of the following two operations:

$$D := D - \{(i,k) \mid \sigma_1^{-1}(i) < \sigma_1^{-1}(k) \le \sigma_1^{-1}(j)\}. \tag{3.2.3}$$

$$D := D - \{(k,j) \mid \sigma_1^{-1}(i) \le \sigma_1^{-1}(k) < \sigma_1^{-1}(j)\}.$$

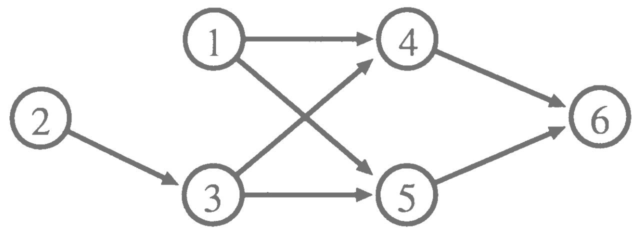


Figure 3.1: The graph representing the partial order D for two solutions $\sigma_1 = \langle 1,2,3,4,5,6\rangle$ and $\sigma_2 = \langle 2,3,1,5,4,6\rangle$.

This operation is repeated s times, where s is a prespecified positive integer.

The mutation relaxes the constraint D, and enlarges the search space of DP computation; hence, the solution quality may improve at the cost of spending more computation time.

Step 3 (Select)

Denote the cost of solution σ by $\text{cost}(\sigma)$. We suppose without loss of generality that

$$\text{cost}(\sigma_1) \leq \text{cost}(\sigma_2). \tag{3.2.4}$$

To maintain P candidate solutions, we tested the following two methods.

Method (i): Select the best P solutions after the Q candidate solutions are formed in Step 2.

Method (ii): The selection is conducted immediately after the new solution is generated in Step 2. More precisely, at each execution of crossover by DP in Step 2, let σ_{new} be the solution obtained by DP from σ_1 and σ_2. Replace σ_2 with the new solution σ_{new} if $\text{cost}(\sigma_{new})<\text{cost}(\sigma_1)$, otherwise σ_{new} is discarded.

It was observed in [130] that method (ii) usually perform better than method (i). However, since σ_{new} and σ_1 tend to become very close, repeating (ii) many times may lose the diversity of P candidate solutions. In order to prevent this, method (ii) is modified as follows.

Method (iii): Replace σ_2 with σ_{new} with probability $p(\Delta_1/\Delta_2)$; otherwise replace σ_1 with σ_{new}, where

$$\Delta_i = \text{cost}(\sigma_i) - \text{cost}(\sigma_{new}),\ i \in \{1,2\}, \tag{3.2.5}$$

$$p(x) = \min\left\{\frac{1}{a}x, 1\right\}. \tag{3.2.6}$$

Note that $\Delta_2 \geq \Delta_1 \geq 0$ by the definition, and hence $0 \leq x \leq 1$ (we consider $x = 1$ if $\Delta_2 = \Delta_1 = 0$). The above $p(x)$ is illustrated in figure 3.2. The positive constant a is a program parameter. If $a = 0$ then σ_{new} always replaces σ_2, and if $a = \infty$ then σ_{new} always replaces σ_1.

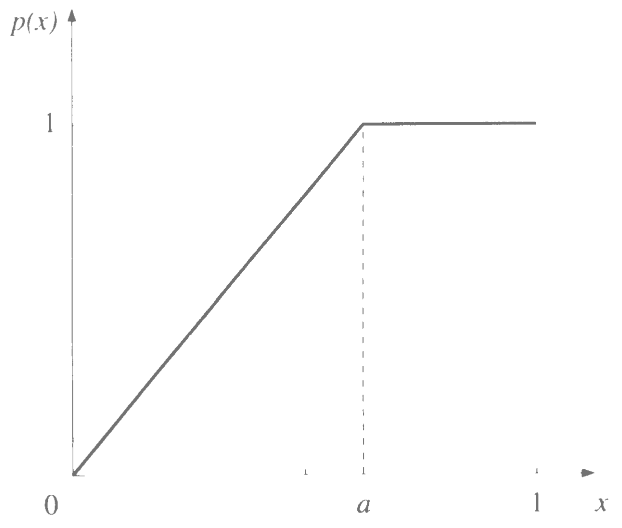


Figure 3.2: The probability function p(·).

Step 4 (Iterate)

The algorithm terminates after r (a given positive integer) successive iterations of Steps 2 and 3 without improvement of the best solution in the P candidates.

3.3 Single Machine Scheduling Problem

The single machine scheduling problem (SMP) asks to determine an optimal sequence of n jobs in $V = \{1,\ldots,n\}$, which are processed on a single machine without idle time. A sequence σ: $\{1,\ldots,n\} \to V$ is a one-to-one mapping such that $\sigma(i) = j$ (or $\sigma^{-1}(j) = i$) means that job j is the i-th job processed on the machine. Each job i becomes available at time 0, requires integer processing time p_i and incurs cost $g_i(c_i)$ if completed at time c_i, where $c_i = \sum_{j=1}^{\sigma^{-1}(i)} p_{\sigma(j)}$. All jobs are processed in time interval $[0, \sum_{i\in V} p_i]$. A sequence σ is optimal if it minimizes

$$\text{cost}(\sigma) = \sum_{i\in V} g_i(c_i). \tag{3.3.7}$$

The single machine scheduling problem is known to be NP-hard for most of the interesting forms of $g_i(\cdot)$. We consider in particular

$$g_i(c_i) = h_i \max\{d_i - c_i, 0\} + w_i \max\{0, c_i - d_i\}, \tag{3.3.8}$$

where $d_i \in Z_+$ (set of nonnegative integers) is the due date of job i, and $h_i, w_i \in Z_+$ are respectively the weights given to earliness and tardiness of job i.

3.3.1 Exact Algorithms

The basic dynamic programming recursion due to [56] can solve SMP exactly. Let $S \subseteq V$ be an arbitrary subset of jobs, and let $f^*(S)$ denote the minimum of cost function (3.3.7) over S when the jobs in S are sequenced in the first $|S|$ positions of the whole sequence. Then $f^*(V)$ defines the cost of an optimal sequence of all jobs, and is obtained by solving

$$f^*(\phi) = 0, \tag{3.3.9}$$

$$f^*(S) = \min_{i \in S}\{f^*(S - \{i\}) + g_i(\sum_{j \in S} p_j)\}, \ \ S \subseteq V.$$

The computational time required to obtain $f^*(V)$ is $O(n2^n)$, since all 2^n subsets S of V need to be generated and the computation of each $f^*(S)$ by (3.3.9) requires $O(n)$ time. This DP reduces the size of the solution space from $n!$ to 2^n. However, the time complexity is still exponential, and this approach is limited to small problem instances, e.g., $n \leq 20$.

A number of exact algorithms, which are based on branch-and-bound, have been studied so far [101]. Another type of algorithm *SSDP* (*successive sublimation dynamic programming*) was proposed in [60]. The essence of SSDP is to execute a series of DP recursions, such that the underlying state-space is progressively refined at each iteration, until an exact optimal sequence of jobs is computed. The number of generated states can be kept within manageable level at each iteration by eliminating those states that are concluded from the information of previous iterations not to lead to optimal sequences. The computational experiment shows that problem instances of up to $n = 35$ can be practically solved.

3.3.2 Genetic DP Algorithm

In this section, we specialize the genetic DP of Section 3.2 to the SMP, and describe the details of Steps 1 and 2.

Step 1 (Initialize)

It is important to generate different types of good solutions as initial candidates. Here we adopt greedy heuristics for this purpose. At each step, let

$$M = \{i \in V \mid i \text{ is not scheduled yet}\}. \tag{3.3.10}$$

Then a job $i \in M$ is chosen as the next job according to some evaluation criterion (e.g., the i with the smallest d_i). There are two types of algorithms, corresponding to whether a schedule is constructed forward or backward. A forward schedule starts with the job to be processed at time 0, and continues adding the next job to be processed until M becomes empty. A backward schedule is symmetrically defined from the last job to the first job. We

will describe below the forward schedule only.

```
GREEDY
  1. Set $M := V$ and $t := 0$.
  2. Choose $i \in M$ that maximizes the local gain $e(i,t)$ as the next job. Let $M := M - \{i\}$
     and $t := t + p_i$.
  3. Repeat Step 2 until $M = \phi$.   □
```

Here the local gain function $e(i,t)$ represents the heuristic used. We employed the following six functions (and hence 12 solutions corresponding to forward and backward constructions).

The first evaluation function e_1 is given by

$$e_1(i,t) = g_i(t + p_i + \bar{p})\delta_i(t) - g_i(t + p_i)(1 - \delta_i(t)), \tag{3.3.11}$$

where $\bar{p}$ is the average processing time

$$\bar{p} = \sum_{i \in V} p_i / n, \tag{3.3.12}$$

and $\delta_i(t)$ indicates whether the due date d_i is urgent or not, i.e.,

$$\delta_i(t) = \begin{cases} 1, & t + \bar{p} + p_i \geq d_i, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3.13}$$

The second and third evaluation functions are

$$e_2(i,t) = w_i\delta_i(t) - h_i(1 - \delta_i(t)), \tag{3.3.14}$$

and

$$e_3(i,t) = \frac{w_i}{p_i}\delta_i(t) - \frac{h_i}{p_i}(1 - \delta_i(t)). \tag{3.3.15}$$

If there are jobs with urgent due dates, the above functions put priority on the job among them, whose cost will increase most rapidly if it becomes tardy. If there is no urgent job, then a job whose cost will decrease most slowly or a job whose current cost is smallest is selected.

Other evaluation functions e_4, e_5, e_6 are also used. Suppose $k = |M| - 1$, i is the job to be evaluated and jobs $j \in M - \{i\}$ are sorted in nondecreasing order of d_j, i.e., $d_{j_1} \leq \ldots \leq d_{j_k}$. Then

$$e_4(i,t) = -g_i(t + p_i) - \sum_{l=1}^{k} g_{j_l}(c_{j_l}), \tag{3.3.16}$$

where $c_{j_l} = t + p_i + \sum_{h=1}^{l} p_{j_h}$. This gives the sum of $g_j(c_j)$ of all jobs in M when i is scheduled next, and the rest is scheduled after i in nondecreasing order of d_j. The functions e_5 and e_6 are variants of e_4 in that e_5 (e_6) uses nonincreasing order of w_i (respectively, nondecreasing order of h_i) instead of nondecreasing order of d_i.

In these six evaluation functions, e_3 often produces the best approximate solutions whose error from the optimals are within 10%, and e_4 usually produces solutions of reasonable quality. Although other schemes are usually not as good as e_3 or e_4, we adopted all six in order to maintain the diversity of initial candidate solutions.

If more than 12 initial solutions are necessary (e.g., $P = 100$ solutions are generated in the experiment of Section 3.6.2), we introduce randomness into the above greedy algorithms. That is, in Step 2 of GREEDY, choose a candidate set $C \subseteq M$ of k jobs (k is a prespecified positive integer) in the decreasing order of the local gain (instead of a single job i), and then randomly choose i from set C. This idea of randomized greedy methods is extensively studied in [34], in the framework of multi-start local search.

Step 2 (Crossover by DP)

We first compute the common partial order D of σ_1 and σ_2, and introduce the constraint that job i must be processed before j if $(i,j) \in D$. Then the best solution with cost $f^*(V)$, among those which are consistent with D, can be obtained by solving the following dynamic programming recursion.

$$f^*(\varphi) = 0, \tag{3.3.17}$$

$$f^*(S) = \min_{i \in I(S)} \{f^*(S - \{i\}) + g_i(\sum_{j \in S} p_j)\}, \quad S \in V^*(D),$$

where

$$V^*(D) = \{S \subseteq V \mid j \in S \text{ and } (i,j) \in D \Rightarrow i \in S\}, \tag{3.3.18}$$

and

$$I(S) = \{i \in S \mid \text{no } j \in S \text{ satisfies } j \neq i \text{ and } (i,j) \in D\}. \tag{3.3.19}$$

Here we give an example of $V^*(D)$ and $I(S)$. For the partial order D of (3.2.2), all the sets $S \in V^*(D)$ and $I(S)$ for each S are shown in Table 3.1. Sets $S = \{1,2,3\}$ and $I(S) = \{1,3\}$ are also shown in the graph representing D in Figure 3.3.

While the DP recursion by (3.3.9) generates all 2^n subsets S of V, the recursion by (3.3.17) generates only those subsets in $V^*(D)$, i.e., those consistent with D. This implies that the computational time and space can be substantially reduced.

It is, however, possible that the number of subsets in $V^*(D)$ is still too large to handle. In such a case, we randomly augment D until the estimated number of states $|V^*(D)|$ becomes less than bn (b is a prespecified positive constant): Randomly choose $k \in \{2, \ldots, n\}$ and let

Table 3.1: All the sets $S \in V^*(D)$ and $I(S)$ for the partial order D of (3.2.2).

S	$I(S)$
$\{1\}$	$\{1\}$
$\{2\}$	$\{2\}$
$\{1,2\}$	$\{1,2\}$
$\{2,3\}$	$\{3\}$
$\{1,2,3\}$	$\{1,3\}$
$\{1,2,3,4\}$	$\{4\}$
$\{1,2,3,5\}$	$\{5\}$
$\{1,2,3,4,5\}$	$\{4,5\}$
$\{1,2,3,4,5,6\}$	$\{6\}$

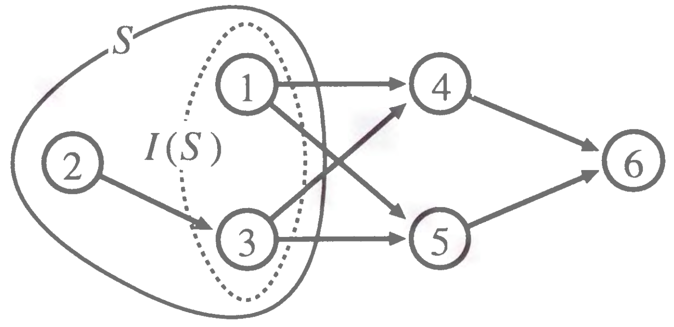


Figure 3.3: Sets $S = \{1,2,3\}$ and $I(S) = \{1,3\}$ are shown on the graph of Figure 3.1.

$$D := D \cup \{(i,j) \mid \sigma_1^{-1}(i) < k \text{ and } k \leq \sigma_1^{-1}(j)\}. \tag{3.3.20}$$

The value $|V^*(D)|$ is important in estimating computation time, since it gives the number of states in dynamic programming recursion (3.3.17). The $V^*(D)$ is known as the set of ideals of partial order D, and much effort has been devoted to the study of estimating $|V^*(D)|$ [120]. It is known [112] that a rather accurate estimation of $|V^*(D)|$ can be obtained in $O(n^2)$ time. This estimation is exact when D has dimension two [119], which holds true in our case if neither mutation (Section 3.2) nor augmentation (3.3.20) is applied.

Note that the sequence σ_1 is always consistent with D, whether it is mutated or augmented, and hence the optimum cost of (3.3.17) will never be greater than that of σ_1 (recall assumption (3.2.4)).

3.4 Optimal Linear Arrangement Problem

In the optimal linear arrangement problem (OLAP), we are given a weighted hypergraph $H = (V, S, W)$, where $V = \{1, \ldots, n\}$ is a set of vertices, $S = \{S_1, \ldots, S_m\}$ is a collection of subsets of V, and $W = \{w_1, \ldots, w_m\}$ is a set of weights given to subsets in S. A *linear arrangement* is a permutation $\sigma : \{1, \ldots, n\} \to V$, meaning that vertex $\sigma(i)$ is placed in the i-th position in a straight line. The cost of a permutation σ is

$$\sum_{i=1}^{m} w_i \max_{k,l \in S_i} \{|\sigma^{-1}(k) - \sigma^{-1}(l)|\}, \tag{3.4.21}$$

and it is asked to find a permutation σ with the minimum cost. The applications of OLAP are abundant in VLSI design and other areas [4, 10, 66, 113]. A special case of OLAP in which the hypergraph H is a graph (each S_i contains exactly two vertices) is referred to as the *Graph Optimal Linear Arrangement* problem (abbreviated as GOLAP). It is known that GOLAP with edge weights equal to 1 is already NP-hard [39]. GOLAP on rooted trees (the root is always placed at the left most position) is solvable in $O(n \log n)$ (where $n = |V|$) time [4], and GOLAP on undirected trees with edge weights equal to 1 is solvable in $O(n^{2.2})$ time [116].

3.4.1 Genetic DP Algorithm

In this section, we explain the details of Steps 1 and 2 of genetic DP for OLAP.

Step 1 (Initialize)

To generate different types of initial candidate solutions, we use two heuristics, Kang's greedy method [66] and the clustering method [10, 113].

Although Kang's method was stated in [66] for the case of unit weights ($w_i = 1$), it is easily extended to arbitrary weights. It begins with a vertex $i \in V$ that minimizes $\text{netcut}(\{i\}, V - \{i\})$, where $\text{netcut}(L, R)$ denotes the sum of net weights between L and R, i.e.,

$$\text{netcut}(L, R) = \sum_{i=1}^{m} w_i \delta_i(L, R), \tag{3.4.22}$$

$$\delta_i(L, R) = \begin{cases} 1, & \text{if } S_i \text{ has vertices in both } L \text{ and } R, \\ 0, & \text{otherwise,} \end{cases}$$

and place it at the leftmost position. It then builds a linear arrangement from left to right by adding one vertex $i \in M$ that maximizes $\text{netcut}(V - M, \{i\}) - \text{netcut}(\{i\}, M - \{i\})$ at each iteration, where

$$M = \{i \in V \mid i \text{ has not been placed yet}\}. \tag{3.4.23}$$

The clustering method has two phases. A cluster $CL_i \subset V$ is a set of vertices. The first phase is executed as follows.

1. Let $CL_i := \{i\}$ $(i = 1, \ldots, n)$, $k := n$, and $S := V$ (S stores the indices of all clusters).
2. Let $k := k + 1$. Find $i, j \in S$ that maximizes $\text{netcut}(CL_i, CL_j)$, let $CL_k := CL_i \cup CL_j$, and $S := S \cup \{k\} - \{i, j\}$.
3. Repeat Step 2 until $|S| = 1$ holds. □

The process of combining two clusters CL_i and CL_j into one cluster CL_k (Step 2) can be represented as a binary tree, called a cluster tree, in which each cluster CL_l is represented as vertex l, and vertices i and j are the two sons of vertex k. It is decided arbitrarily which of i and j becomes the left son. The resulting left to right order of all leaf vertices of the whole cluster tree is a linear arrangement and is output as a heuristic solution obtained in the first phase. In the second phase, the above solution is improved by applying the local search whose neighborhood $N(\sigma)$ is the set of solutions obtained by exchanging left and right sons of any inner vertex of the cluster tree.

Step 2 (Crossover by DP)

Here we consider the following two methods.

(A) Compute the common partial order D of (3.2.1) for two candidate solutions σ_1 and σ_2. Note that $(i, j) \in D$ in this case denotes that vertex i is placed to the left of j in both σ_1 and σ_2. Call $L \subset V$ a left segment if the vertices in L are arranged to the left of the rest of vertices $R = V - L$. Then we can find the best arrangement among those consistent with D, by solving

$$f^*(L) = 0, \ |L| \le 1, \tag{3.4.24}$$

$$f^*(L) = \min_{j \in I(L)} \{f^*(L - \{j\}) + \text{netcut}(L - \{j\}, R \cup \{j\})\}, \ L \in V^*(D),$$

where

$$V^*(D) = \{L \subseteq V \mid |L| > 1, (j \in L \text{ and } (i,j) \in D \Rightarrow i \in L)\}, \tag{3.4.25}$$

$I(\cdot)$ is defined in (3.3.19), and $f^*(L)$ denotes the minimum cost (3.4.21) when the vertices in L are arranged in the left $|L|$ positions. Obviously $f^*(V)$ denotes the minimum cost of all vertices, which we want to compute.

(B) In order to reduce the computation time of (3.4.24), we add the chain constraints to method (A) in the following manner. Call that vertices i and j are adjacent in σ if

$$|\sigma^{-1}(i) - \sigma^{-1}(j)| = 1, \tag{3.4.26}$$

and denote

$$AD(\sigma) = \{(i,j) \mid \text{vertices } i \text{ and } j \text{ are adjacent}\}. \tag{3.4.27}$$

We impose the constraint that every pair of vertices adjacent in both σ_1 and σ_2 are forced to be adjacent in the new solution σ_{new}, i.e., $AD(\sigma_1) \cap AD(\sigma_2) \subset AD(\sigma_{new})$. Note that each connected component in graph $G(\sigma_1, \sigma_2) = (V, AD(\sigma_1) \cap AD(\sigma_2))$ is a chain. The DP computation of (3.4.24) can be carried out more efficiently by applying it after contracting each chain into a single vertex, though the values of netcut$(\cdot,\cdot)$ must be calculated for the original set of vertices.

In both methods (A) and (B), when the estimated number of states $|V^*(D)|$ exceeds bn, the operation of (3.3.20) is also applied.

A computational comparison of these two methods for various b is shown in figure 3.4, in which 11 test instances of up to $n = 40$ are solved. It shows how the total cost (3.4.21) changes against the amount of time required (which is determined by parameter b). Program parameters are set to $P = 20$, $r = 300$, $a = 0.5$, $s = 0$ (see Sections 3.2 and 3.3.2 for the details of these parameters) and b is varied from 0 to up to 100. Figure 3.4 shows superiority of (B) to (A). In the computational experiment in Section 3.6, where genetic DP is compared with other approximate methods, we therefore adopt (B).

3.5 Traveling Salesman Problem

The traveling salesman problem is one of the most well-known combinatorial optimization problems. It asks to find the shortest *tour* (i.e., Hamiltonian circuit, that is, a path that visits every vertex exactly once and returns to the first vertex) in a given graph $G = (V, E)$, where

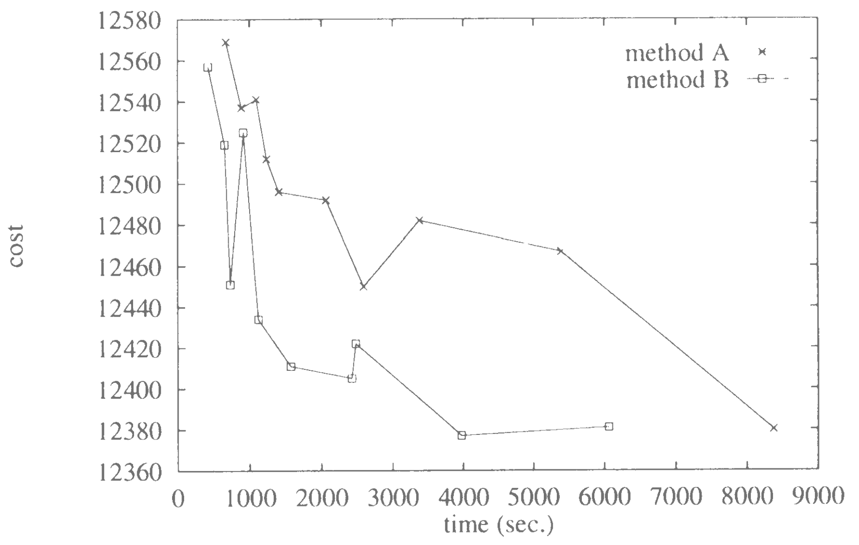


Figure 3.4: The effect of chain constraints.

$V = \{1, \ldots, n\}$ and each edge $(i,j) \in E$ has length d_{ij}. The symmetric traveling salesman problem we consider assumes $d_{ij} = d_{ji}$ for all pairs of i and j. Let σ: $\{1, \ldots, n\} \to V$ be a tour, where $\sigma(i)$ denotes the i-th vertex in a tour σ. A tour is optimal if it minimizes

$$\sum_{i=1}^{n-1} d_{\sigma(i)\sigma(i+1)} + d_{\sigma(n)\sigma(1)}. \tag{3.5.28}$$

Numerous exact and approximate algorithms have been proposed for this problem [75], and it is reported that exact optimal solutions have been obtained for problem instances of up to $n = 7397$ [8, 51, 93] (500,000 in the case of asymmetric version [85]).

3.5.1 Genetic DP Algorithm

In this section, we explain the details of Steps 1 and 2 of genetic DP for TSP.

Step 1 (Initialize)

We use the ***arbitrary insertion*** method [75, 108] for generating initial candidate solutions. It is a greedy method and can create reasonably good solutions in short time. A path that visits every vertex in a subset S of V exactly once, and returns to the first vertex in S is called a *subtour*. The arbitrary insertion procedure begins with a randomly chosen subtour consisting of only two vertices, and iterates the insertion of the vertex k between randomly chosen adjacent vertices i and j in the current subtour, where k minimizes $d_{ik} + d_{kj} - d_{ij}$, until a tour is formed. There are a number of variations of insertion heuristics [75], such as the farthest insertion and convex hull insertion procedures. We have chosen the arbitrary insertion for our computational experiment because it is simple and can produce a variety of solutions.

Step 2 (Crossover by DP)

Suppose that a tour always starts from vertex 1 (i.e., $\sigma_1(1) = \sigma_2(1) = 1$) without loss of generality. The definition of the partial order D common to σ_1 and σ_2 is the same as (3.2.1). The best tour, which is consistent with D, and its cost $f^*(V)$, can be obtained by solving

$$f^*(\{1\}, 1) = 0,$$

$$f^*(S, i) = \min_{j \in I(S)-\{i\}} \{f^*(S - \{i\}, j) + d_{ji}\}, \ S \in V^*(D), \tag{3.5.29}$$

$$f^*(V) = \min_{i \in I(V)} \{f^*(V, i) + d_{i1}\},$$

where

$$V^*(D) = \{S \subseteq V \mid j \in S \text{ and } (i, j) \in D \Rightarrow i \in S\}, \tag{3.5.30}$$

and $I(\cdot)$ is the same as (3.3.19). $f^*(S, i)$ denotes the length of the shortest path that starts from vertex 1, visits all the vertices in S, ends with vertex i, and is consistent with D. In the computational experiment of Section 3.6, we also added the chain condition of method (B) of OLAP, to speed up the DP computation. The modification required is similar to the case of OLAP.

3.6 Computational Results for Three Problems

3.6.1 Generation of Problem Instances

Computational experiments were performed on SUN SPARC station IPX using C language for SMP and OLAP, and using FORTRAN 77 for TSP. The tested problem instances are generated as follows.

SMP: For each n, coefficients p_i, h_i, w_i for $i \in V$ $(= \{1, \ldots, n\})$ are generated by randomly selecting integers from interval [1, 10]. It has been observed in the literature (e.g., [100]) that problem hardness is related to two parameters RDD and LF, called the relative range of due dates and the average lateness factor, respectively. In our experiment,

$$RDD = 0.2, 0.4, 0.6, 0.8, 1.0,$$
$$LF = 0.2, 0.4,$$

are used. Corresponding to each of these $5 \times 2 = 10$ cases, one problem instance is generated by selecting integer due dates d_i, $i \in V$, from interval

$$[(1 - LF - RDD/2)T, \ (1 - LF + RDD/2)T].$$

OLAP: For each n, $2^3 = 8$ types of instances depending on (1) whether H is a hypergraph or a graph, (2) whether weighted or unweighted, and (3) whether $m = 2n$ or $m = 4n$ when $n > 20$ ($m = 6n$ when $n \leq 20$), are generated. For each $i = 1, 2, \ldots, m$, an integer $|S_i|$ is randomly chosen from interval [2, 5] in the case of a hypergraph ($|S_i| = 2$ in the case of a graph) and then $|S_i|$ vertices are randomly chosen from V as the elements in S_i. S_i are generated so that $S_i \neq S_j$ holds for $i \neq j$. The weight w_i (integer) of S_i is chosen randomly from interval [1, 5] ($w_i = 1$ in the unweighted case).

TSP: We considered only the Euclidean case (i.e., all vertices are located in the plane and edge lengths are given by the Euclidean distances between their end vertices). A coordinate pair (x_i, y_i) of each vertex is first generated by randomly selecting two integers from interval [0, 1000], and the length between vertices i and j is set to

$$d_{ij} = \lfloor \{\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + 0.5\} \rfloor.$$

3.6.2 The Effect of Program Parameters

Implementation of genetic DP contains the following parameters:

r: number of iterations without improvement before termination (see Section 3.2),

a: parameter in (3.2.6), which decides the frequency of replacing σ_1 in Step 3 (select),

P: number of candidate solutions (population),

b: bn is the upper bound on the number of states generated in DP recursion (see Section 3.3.2),

s: number of mutations (see Section 3.2).

To know appropriate values of these parameters, 10 SMP instances of $n = 50$, 11 OLAP instances of up to $n = 40$, and 5 TSP instances of $n = 100$ were generated and tested. We examined how the performance changes according to the amount of time invested (which is determined by program parameters). Every parameter has a tendency that the larger it becomes, the more computational time is needed. In the figures explained below, "cost" denotes the total cost (3.3.7), (3.4.21) or (3.5.28) of the solutions obtained and "time" is the total time in seconds required to solve all test instances of each problem. Parameters are set as given in table 3.2 unless otherwise stated. Only the parameter examined is changed.

Table 3.2: Parameter values used for the parameter tuning.

	SMP	OLAP	TSP
r	200	300	1000
a	0.5*	0.5*	0.5*
P	20	20	20
b	1000	20	30
s	0	0	0

* $a = 2$ when r is examined.

First we examined the effect of r. Figure 3.5 shows that, for SMP, great improvement is achieved only in the early stage of increasing r. Similar results are also observed for OLAP and TSP [129].

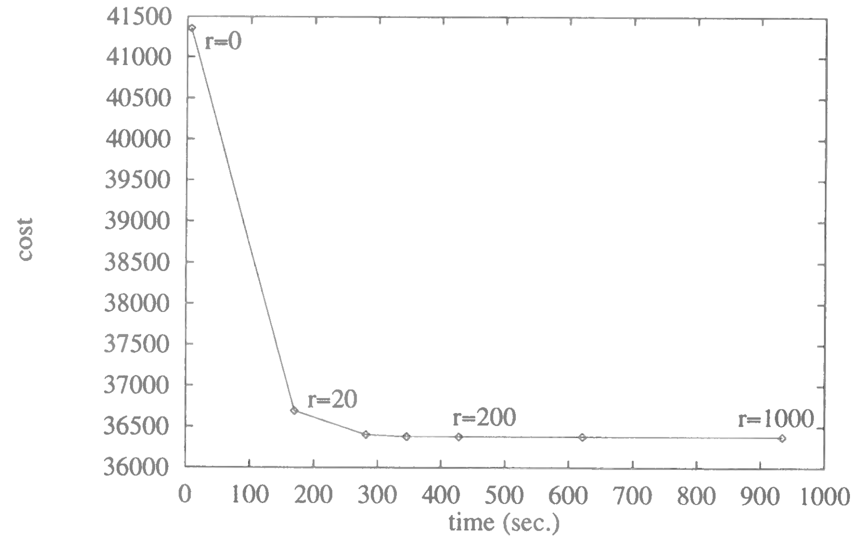


Figure 3.5: The effect of the number of iterations r (SMP).

Second we examined the effect of a. The results for SMP are shown in figure 3.6. Although the behavior seems to be quite erratic, it may indicate that the quality improves (by consuming more computational time) when the parameter a is increased from 0 to up to 0.75. However, setting a beyond 0.75 seemed to consume more time without gaining much improvement. The results for OLAP and TSP are similar [129].

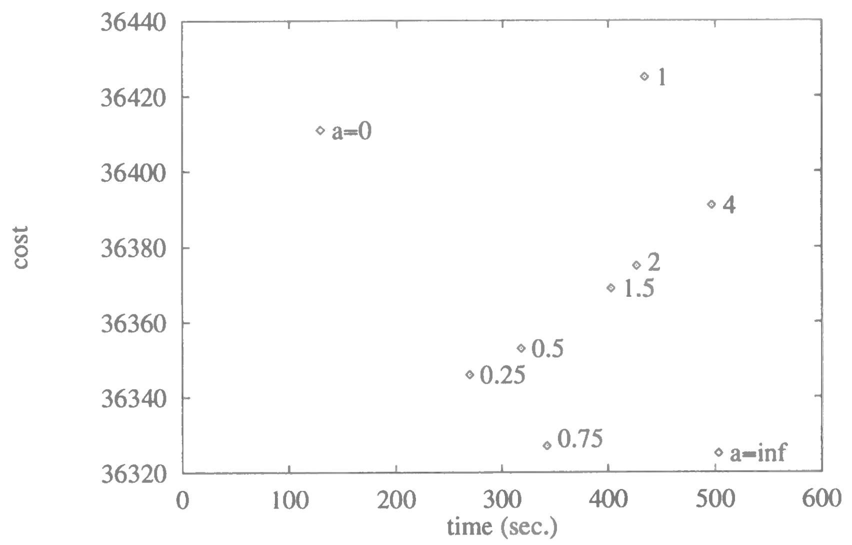


Figure 3.6: The effect of parameter a in the probability function p(·) (SMP).

Next we examined P, b and s. The results are exhibited in figures 3.7, 3.8 and 3.9 for SMP, OLAP and TSP, respectively. For SMP, parameter b has the largest effect and P is less effective. The introduction of mutation (i.e., larger s) improves the solution quality to some extent, but considering the amount of time consumed, a small s appears to be preferable. For OLAP, the effect of parameters P and b were almost the same. The introduction of mutation improved the solution quality, but the magnitude of improvements and the amount of time consumed were insensitive to s. This is because the state bound $b = 20$ was a bit too strict and the mutations could not enlarge the search space of DP (because of the bound imposed by b). For this reason, we adopted a small s. For TSP, the parameter P was the most effective and b was less effective. In this case, the effect of mutation was not clear and sometimes worse solutions were produced while spending greater amount of time.

From these results, we concluded to set the parameters as in table 3.3.

3.6.3 Performance of Genetic DP

After fixing the parameters of genetic DP as discussed above, we solved SMP instances of $n = 15, 20, 25, 30, 35, 50, 75,\ 100$ (10 instances for each n), and OLAP instances of $n = 15, 20, 40, 60,\ 80$ (8 instances for each n, except that 12 instances for $n \leq 20$), and TSP instances of $n = 100, 200,\ 300$ (5 instances for each n).

The solutions with proved optimality could be obtained by exact algorithm SSDP only for

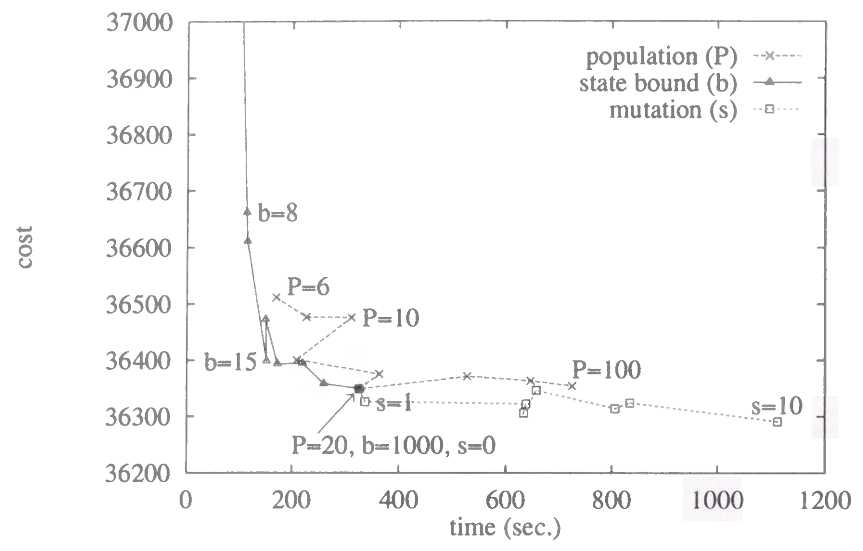


Figure 3.7: The effect of parameters P, b, s (SMP).

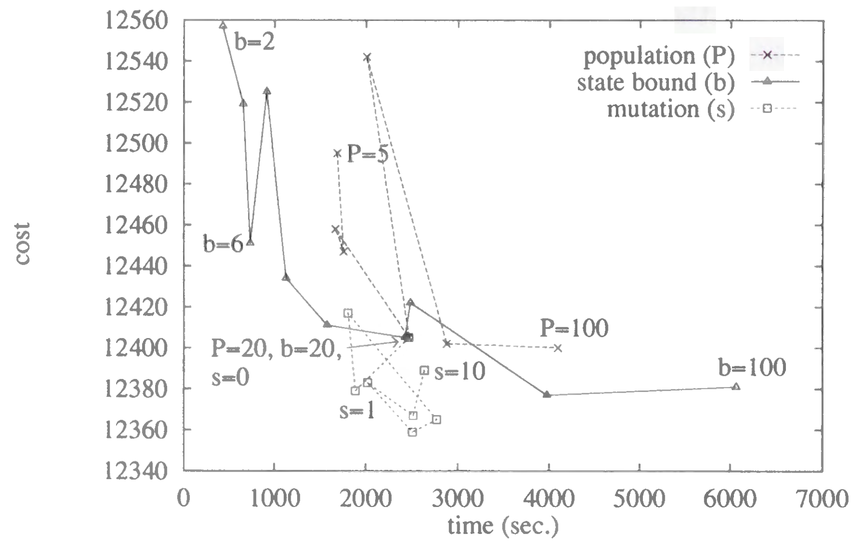


Figure 3.8: The effect of parameters P, b, s (OLAP).

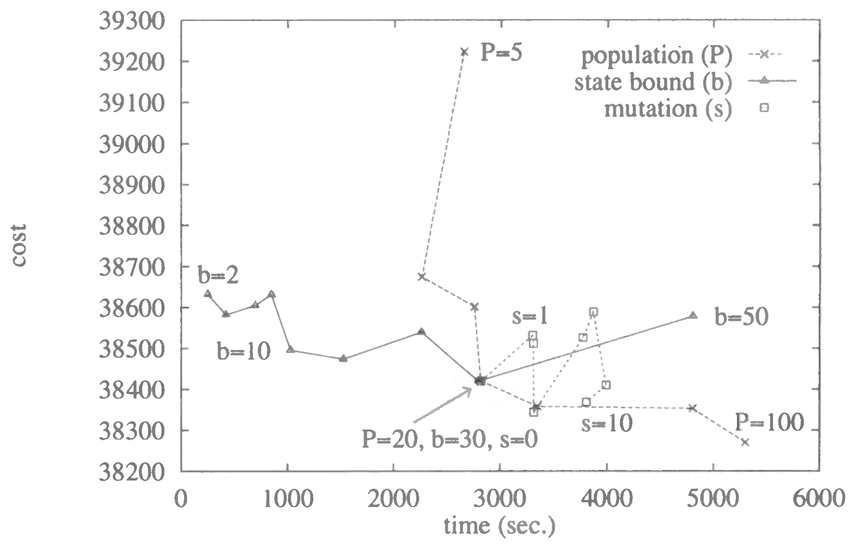


Figure 3.9: The effect of parameters P, b, s (TSP).

Table 3.3: Parameter values adopted after the parameter tuning.

	SMP	OLAP	TSP
r	200	300	1000
a	0.5	0.5	0.5
P	20	20	30
b	100	20	10
s	1	1	0

small instances of up to $n = 35$. Genetic DP succeeded in obtaining exact optimal solutions for 47 out of 50 instances with $n \leq 35$. The maximum error from the optimum values of the 3 instances, which genetic DP failed to find, was less than 0.45%. Genetic DP could get exact optimal solutions for all the 50 instances when different parameter values were used. The average time of genetic DP, and the two exact algorithms DP and SSDP are shown in figure 3.10 in seconds.

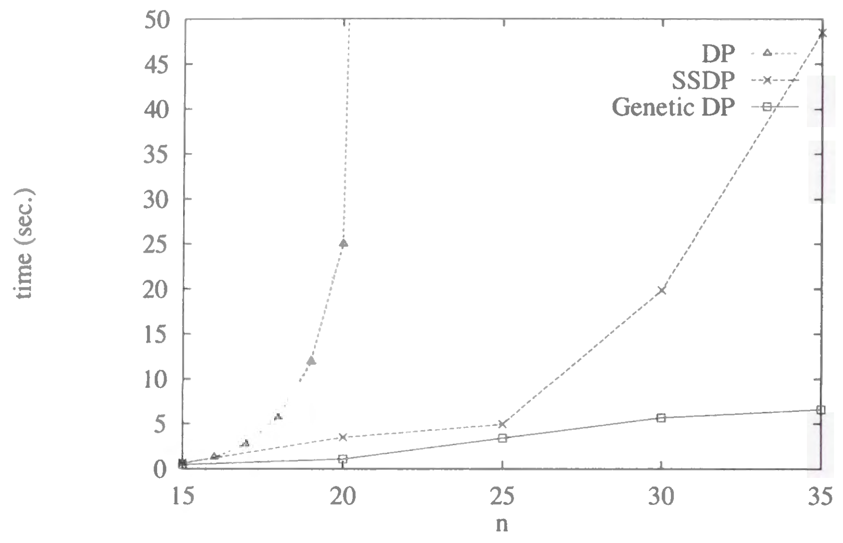


Figure 3.10: The computational time (in seconds) to solve SMP by algorithms DP, SSDP and genetic DP.

In the case of OLAP, exact optimal solutions can be obtained by using DP only for small instances of up to $n = 20$. Genetic DP succeeded in obtaining optimal solutions for 23 out of 24 OLAP instances of these sizes. The error from the optimum value of the only one instance, which genetic DP failed to solve exactly, is about 1.51%. Genetic DP could get an optimal solution to this instance by using different parameter values. The average time of genetic DP and the exact algorithm DP are shown in table 3.4 in seconds.

For TSP, the comparison with the exact algorithms was not attempted, because the performance of genetic DP is rather poor even against the heuristic algorithm of Lin and Kernighan [78], as will be reported in the next subsection.

3.6.4 Comparison with Other Heuristic Algorithms

The performance of genetic DP is compared with that of other heuristic algorithms.

Table 3.4: The computational time (in seconds) to solve OLAP by algorithms DP and genetic DP.

n	DP	Genetic DP
15	115	54
20	6554	120

Details of Other Heuristic Algorithms

The first heuristic algorithms tested are the following two, already explained in Section 3.1:

(1) Multi-start local search algorithm (MLS),

(2) Genetic local search algorithm (GLS).

In MLS and GLS, the neighborhood $N(\sigma)$ is defined to be the set of solutions which can be obtained by moving a single element $\sigma(i)$ to the location between $\sigma(j-1)$ and $\sigma(j)$, for all pairs of i and j $(i \neq j)$ (see figure 3.11). The crossover operation for GLS is performed

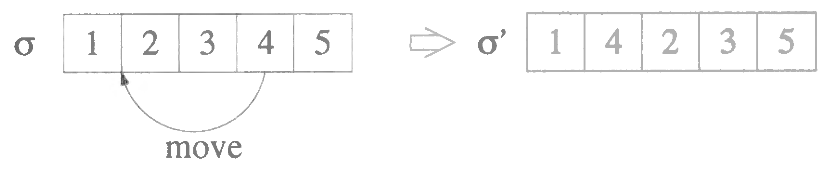


Figure 3.11: A solution in neighborhood $N(\sigma)$, corresponding to $i = 4$ and $j = 2$.

as follows (see figure 3.12). For two solutions σ_1 and σ_2, choose randomly an integer l from interval $[n/4, n/2]$ and an integer k from interval $[1, n-l+1]$. Then the new solution σ_{new} is constructed by $\sigma_{new}(i) := \sigma_1(i)$ $(i \in [k, k+l-1])$, and setting $\sigma_{new}(i)$ $(i \notin [k, k+l-1])$ according to the order of σ_2 (i.e., $i < j \Rightarrow \sigma_{new}^{-1}(\sigma_2(i)) < \sigma_{new}^{-1}(\sigma_2(j))$ for all $i, j \notin [k, k+l-1]$). This is a variation of the crossover operation mentioned in [89] for TSP. Step 3 (select) and Step 4 (iterate) for GLS are the same as genetic DP.

This GLS is a bit different from the genetic local search algorithm proposed in other literature, such as [89, 126], in which all initial candidate solutions are improved by local search, before applying crossover operation. This original type of genetic local search algorithm (GLS#) is also tested. Further, we examined the performance of the traditional genetic algorithms (simple GA), which do not include the improvement by local search.

(3) Approximate DP. This applies DP recursions of (3.3.17), (3.4.24) or (3.5.29) to all pairs in P initial candidates, and then halts. Initial solutions are the same as those used in

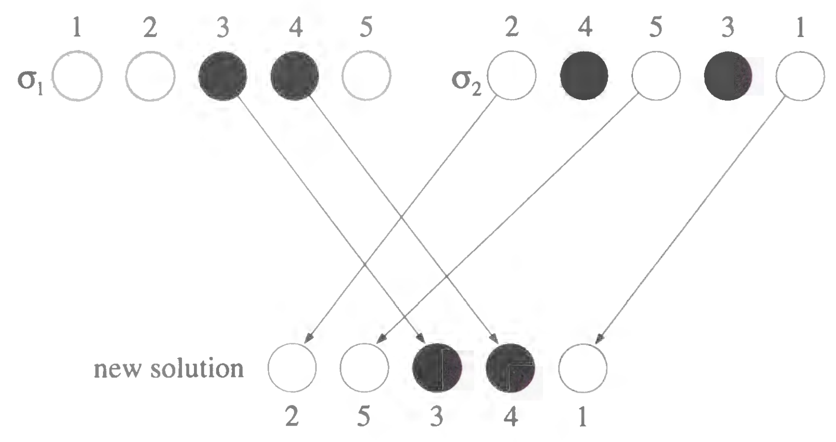


Figure 3.12: An example of crossover operation for GLS, where $l = 2$, $k = 3$.

genetic DP. This algorithm is tested to see the effect of only the DP part of genetic DP.

(4) *Or-opt* procedure [75] and *Lin-Kernighan* algorithm (abbreviated as LK) [78]. These are examined only in the case of TSP. In LK, random tours uniformly chosen from the set of all possible permutations are used for the initial solutions.

Results and Discussions

Figures from 3.13 to 3.17 show how the average error (%) from the best cost found during our experiment decreases with time, where the average time used for the largest instances ($n = 100$ for the SMP, $n = 80$ for the OLAP and $n = 300$ for the TSP) is used.

Figure 3.13 shows a comparison of simple GA and GLS for SMP. Similar results are obtained for OLAP and TSP [129]. These results indicate a rather discouraging feature of simple GA, which is also observed in other references such as [126]. Figure 3.14 exhibits a comparison of approximate DP and genetic DP. Similar results were obtained for OLAP and TSP. From these, we can see clear dominance of genetic DP over simple GA and approximate DP.

Figures 3.15, 3.16 and 3.17 compare genetic DP with other heuristic algorithms. In the case of SMP, genetic DP triumphed over MLS and GLS. Genetic DP obtained better solutions in shorter time. In the case of OLAP, genetic DP obtained better solutions than MLS in most cases; but when longer computational time was allowed, GLS obtained slightly better solutions. In the case of TSP, genetic DP could get better solutions than MLS, GLS and Or-opt when sufficient computational time was allowed. But LK could obtain much better solutions in shorter time.

Finally, Figure 3.18 shows how the computational time for SMP increases as the size of

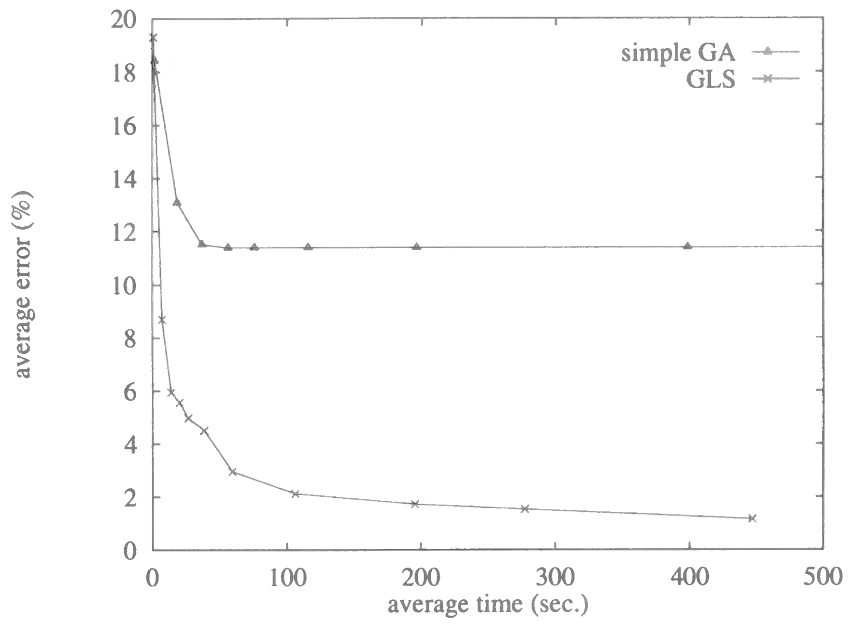


Figure 3.13: Comparison of simple GA and GLS (SMP).

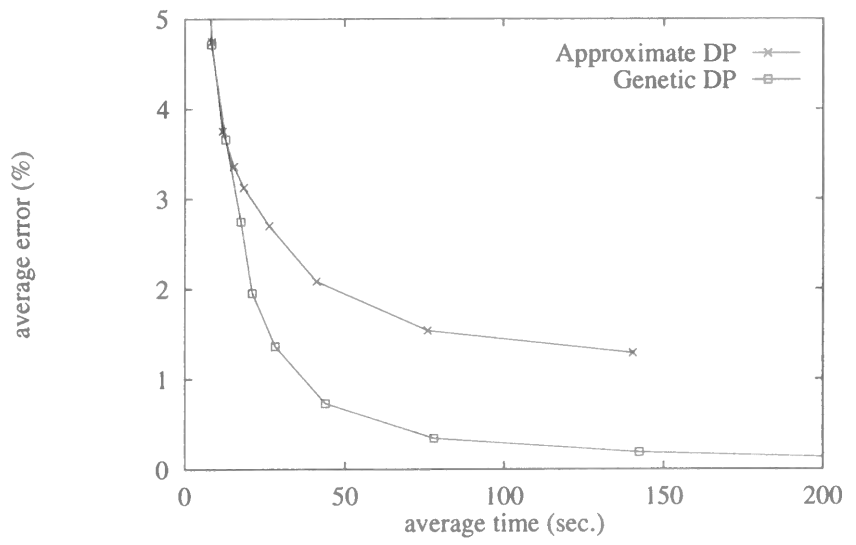


Figure 3.14: Comparison of approximate DP and genetic DP (SMP).

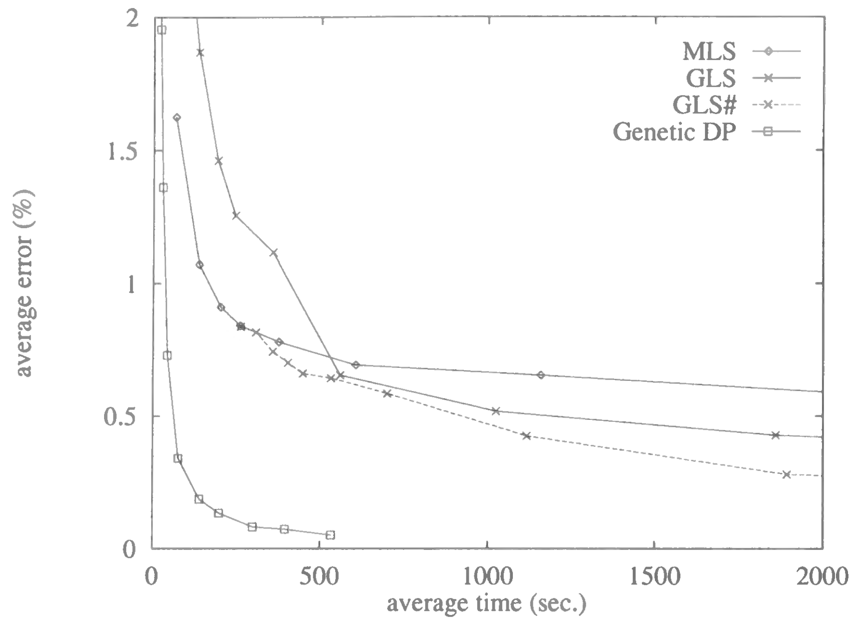


Figure 3.15: A comparison of the four algorithms (SMP).

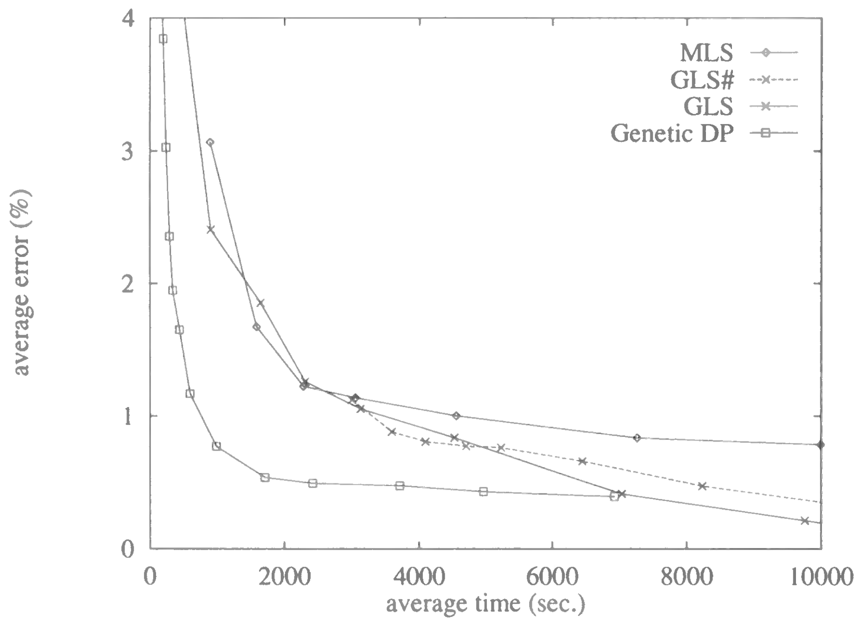


Figure 3.16: A comparison of the four algorithms (OLAP).

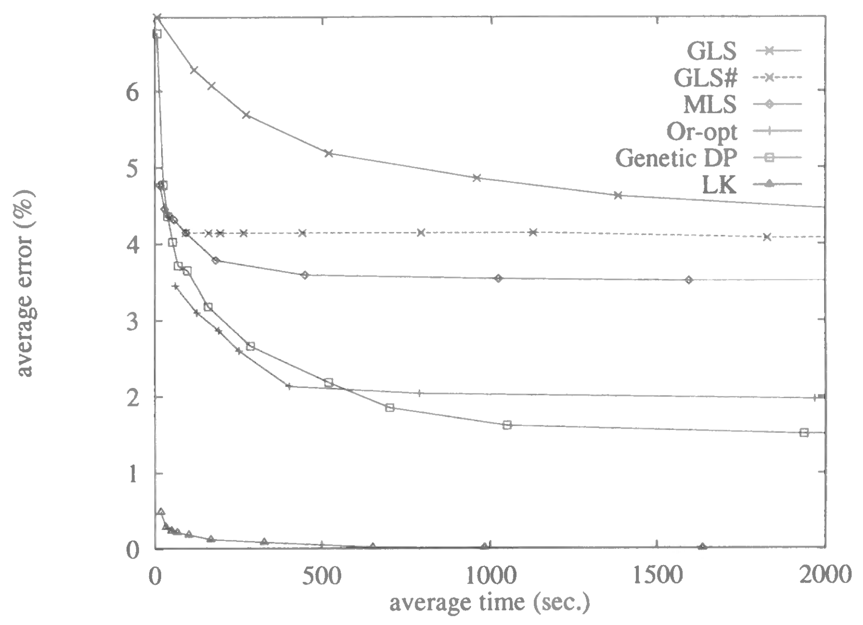


Figure 3.17: A comparison of the six algorithms (TSP).

problem instance n grows, when all the algorithms are terminated with 500 iterations, where the time for $n = 15$ is normalized to 1. It is observed that the computational time for MLS and GLS increase slightly more rapidly than genetic DP. Similar results were obtained for OLAP and TSP [129]. This is because the number of states necessary for the DP recursion of genetic DP was bounded by bn by operation (3.3.20) and the computational time for each DP recursion (3.3.17) was $O(bn^2)$.

From the above results, we may conclude that genetic DP is one of the most powerful meta-heuristics for general purposes. However, it is also noted that very efficient heuristic algorithms, such as LK, may exist if the algorithms are tailored to the given problems.

3.7 Conclusion

We proposed a framework of approximate algorithms, called genetic DP, and evaluated its effectiveness by conducting computational experiments for three problems SMP, OLAP and TSP, all of which ask to obtain optimal permutations of n elements. Genetic DP tends to attain better solution quality than traditional multi-start local search and genetic local search algorithms when sufficiently long time is allowed, though performance of these algorithms depends on problem characteristics. However, if some efficient heuristics specially designed to the given problem, such as Lin-Kernighan method, are available, we recommend to use them.

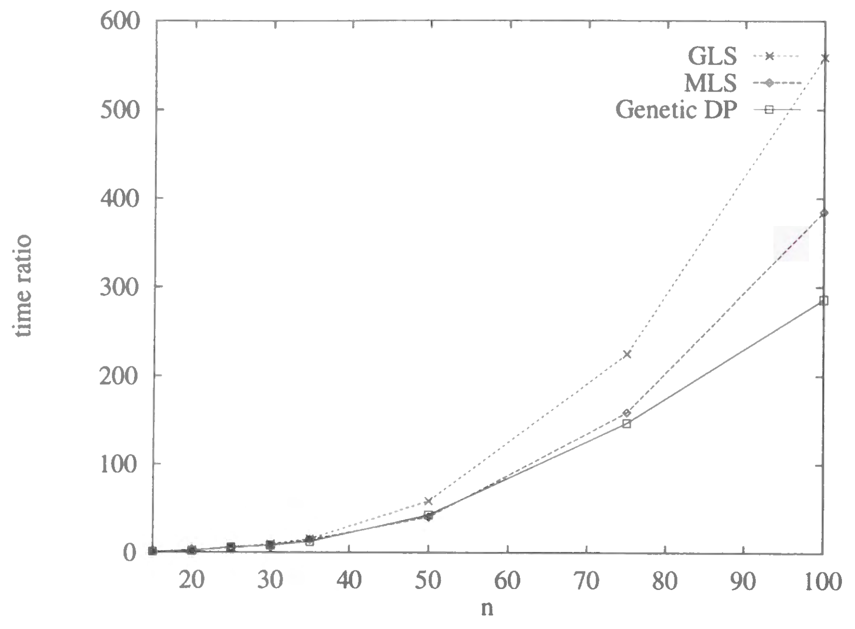


Figure 3.18: Time ratio with respect to n (time for n=15 is regarded as 1) of the three algorithms (SMP).

Combination of such special purpose heuristics with genetic algorithms may be an important subject of future study. It is emphasized, however, that an advantage of general metaheuristics, including genetic DP, is that they can be easily adapted to many problems, while problem specific algorithms, such as Lin-Kernighan, are hardly adapted to other problems.

Recently, similar hybrid approach of combining exact methods and metaheuristic methods are tried in [5, 83].

Chapter 4

On Genetic Crossover Operators for Sequencing Problems

4.1 Introduction

Crossover is one of the basic operators of genetic algorithm (GA), and has a great influence on the performance of the algorithm [90, 118]. New solutions, called ***children***, are generated from more than one candidate solution, called ***parents***, by crossover operators. Many crossover operators have been proposed, e.g., 1-point, 2-point, multi-point and uniform crossover operators for binary strings, and those crossover operators applicable to more general objects such as figures and graphs [22]. Most of combinatorial optimization problems have constraints on the solution space, and the feasibility of the generated children should be taken into account when crossover operators are designed. For example, many crossover operators have been proposed for the traveling salesman problem (TSP) whose feasible solutions are permutations of the given n cities. A reason for this is that keeping the feasibility of the children is not trivial for this problem [46, 50, 142]. We will illustrate some of them in the next section.

In this chapter, we first review various crossover operators proposed for the combinatorial optimization problems whose feasible solutions are given by permutations. We call such a crossover operator as *permutation crossover*. We then consider a general framework of crossover operators and analyze the relationship between characteristics of the operator and performance of the algorithm. Based on this analysis, we propose simple criteria for measuring the quality of crossover operators. Computational experiments for the single machine scheduling problem (SMP) using a simple framework of GA is conducted, and it is observed that the following two criteria are important for crossover operators: (1) inherit as many elements as possible from the parents, and (2) keep the diversity of the children obtainable from the parents.

4.2 Crossover Operators for Sequencing Problems

In this section, we review various crossover operators proposed in the literature for sequencing problems, where we restrict our attention to those without solution improvement mechanisms such as heuristics and local search. Most of the crossover operators introduced in this section are originally proposed for TSP. However, we sometimes slightly modify them so that they also fit to SMP. Here we assume that one child C is generated from two parents A and B. Let $V = \{1, \dots, n\}$ be a set of n elements, and $\sigma(i) = j$ (equivalently $\sigma^{-1}(j) = i$) denote that the i-th element of the permutation σ is j. The permutations of the parents A, B and the child C are denoted σ_A, σ_B and σ_C, respectively.

PMX (partially mapped crossover): Randomly generate an n bit 0-1 mask msk, where $msk(i) \in \{0,1\}$. For each i with $msk(i) = 0$, set $\sigma_C(i) := \sigma_A(i)$ and $\sigma_B(j) := \sigma_B(i)$ for j with $\sigma_B(j) = \sigma_A(i)$. Then for each i with $msk(i) = 1$, set $\sigma_C(i) := \sigma_B(i)$ (see Figure 4.1).

The crossover operators in which the elements are inherited according to randomly generated masks are called *uniform crossover*. If the masks are restricted to those in which 0 and 1 are adjacent in at most k positions, then they are called *k-point crossover*. For example, masks 11000 and 10011 correspond to 1-point and 2-point crossover operators, respectively. Here we consider 1-point, 2-point and uniform crossover operators for PMX, and denote them as PMX(1), PMX(2) and PMX(U), respectively.

PMX was originally proposed as the 2-point crossover operator in [46]. It is also introduced in other literature such as [47, 90, 118].

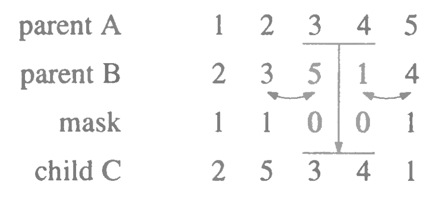


Figure 4.1: An example of PMX(2).

CX (cycle crossover): In this method, the child σ_C is constructed so that

$$\sigma_C(i) = \sigma_A(i) \text{ or } \sigma_B(i) \tag{4.2.1}$$

hold for all i. First, cycle number $cycle(i)$ is computed for each position i by the following algorithm (see Figure 4.2), where $cycle(i) = 0$ means that position i has not numbered yet.

1. Set $k := 1$ and $cycle(i) := 0$ for all i.
2. Set $i_0 := \min\{i \mid cycle(i) = 0\}$ and $i := i_0$.

3. Set $cycle(i) := k$ and $i := \sigma_A^{-1}(\sigma_B(i))$.
4. Return to Step 3 unless $i = i_0$ holds.
5. If $cycle(i) > 0$ hold for all i, then halt; otherwise set $k := k + 1$ and return to Step 2.

The n elements are partitioned into cycles $CYC_k = \{i \mid cycle(i) = k\}$ by their cycle numbers. Therefore, condition (4.2.1) can be satisfied by inheriting the elements in a cycle from the same parent (see Figure 4.2). Here we consider the following three methods to choose the parent for each cycle: (1) the parent is randomly chosen for each cycle, denoted CX(U), (2) one cycle is randomly chosen from parent A and others are taken from B, denoted CX(1), and (3) cycles with odd indices are taken from parent A and others are taken from B, i.e., cycles are alternately chosen, denoted CX(A).

CX was originally proposed as CX(U) in [90]. It is also introduced in other literature such as [47, 118].

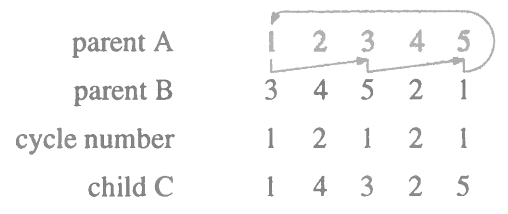


Figure 4.2: An example of CX(U).

FLX (free list crossover): In this method, a permutation is represented by using the list of n elements (e.g., $(n, n-1, \dots, 1)$). A permutation is coded by determining the position of each element in the list from left to right, where the used elements are removed from the list. Here we use the ordered list $(1, 2, \dots, n)$. Then the code $\bar{\sigma}$ of a permutation σ is formally defined as

$$\bar{\sigma}(i) = \sigma(i) - |\{j \mid \sigma(j) < \sigma(i) \text{ and } j < i\}| \tag{4.2.2}$$

(see Figure 4.3). The original σ can be obtained from $\bar{\sigma}$ by the following decoding algorithm.

```
Algorithm Decode_FLX
Line 1: for $i = 1, 2, \dots, n$ do
Line 2:     $list(i) := i$;
Line 3: end for;
Line 4: for $i = 1, 2, \dots, n$ do
Line 5:     $\sigma(i) := list(\bar{\sigma}(i))$;
Line 6:     for $j = \bar{\sigma}(i), \bar{\sigma}(i) + 1, \dots, n - i$ do
Line 7:                 $list(j) := list(j+1)$
Line 8:             end for
Line 9:     end for.
```

If $\bar{\sigma}(i) \le n-i$ holds for all i, algorithm Decode_FLX outputs a permutation σ, that is, every element appears exactly once in the resulting σ, since, at the i-th iteration, the elements $\sigma(1), \sigma(2), \ldots, \sigma(i-1)$ have already been removed from *list*. This is the one to one mapping between σ and $\bar{\sigma}$. Then an n bit 0-1 mask $msk \in \{0,1\}^n$ is randomly generated, and a coded child $\bar{\sigma}_C$ is produced by setting $\bar{\sigma}_C(i) := \bar{\sigma}_A(i)$ for i with $msk(i) = 0$, and setting $\bar{\sigma}_C(i) := \bar{\sigma}_B(i)$ for i with $msk(i) = 1$. The resulting $\bar{\sigma}_C$ is then decoded to make the child σ_C. As in PMX, we consider 1-point, 2-point and uniform crossover operators, and denote them FLX(1), FLX(2) and FLX(U), respectively. FLX was originally proposed as FLX(1) in [50].

	σ							$\bar{\sigma}$			
parent A	2	3	1	5	4	$\xrightarrow{\text{code}}$	parent A	2	2	1	2	1
parent B	3	1	5	4	2		parent B	3	1	3	2	1
							mask	1	1	0	0	0
child C	3	1	2	5	4	$\xleftarrow{\text{decode}}$	child C	3	1	1	2	1

Figure 4.3: An example of FLX(1) with list (1, 2, 3, 4, 5).

POPX (partial order preserving crossover): In this method, the child is a linear extension of the partial order defined by the two parents (i.e., the child does not conflict with the precedence relation common to both parents). Let D_A be

$$D_A = \{(i,j) \mid \sigma_A^{-1}(i) \le \sigma_A^{-1}(j)\}. \tag{4.2.3}$$

Sets D_B and D_C are similarly defined. Then the partial order of the two parents A and B is defined by $D = D_A \cap D_B$. (This is the same D as in (3.2.1) defined on σ_A and σ_B.) Then a child C is generated so that $D \subseteq D_C$ holds.

We consider two methods to generate a child. The first method POPX1 is described as follows. For a subset $S \subseteq V$, let $M_D(S)$ be

$$M_D(S) = \{i \in S \mid (j,i) \notin D \text{ for all } j \in S - \{i\}\}.$$

Then the child σ_C is generated as follows.

1. Set $S := V$ and $i := 1$.

2. Randomly choose $j \in M_D(S)$, and set $\sigma_C(i) := j$.
3. If $i = n$ holds, then halt; otherwise set $i := i+1$, $S := S - \{j\}$ and return to Step 2.

This method is motivated by [134]. Similar idea is also introduced in [35].

In the second method POPX2, the element j is chosen from the set $\{\sigma_A(i_A^S), \sigma_B(i_B^S)\}$ in Step 2 instead of $M_D(S)$, where i_A^S is defined by $i_A^S = \min\{i \mid \sigma_A(i) \in S\}$ and i_B^S is similarly defined. See Figure 4.4, where the partial order D is represented by directed arcs of the graph in the same manner as Figure 3.1. In the example, initially $M_D(\{1,2,3,4,5\}) = \{1,2\}$ holds and 2 is chosen as $\sigma_C(1)$. In the second iteration, $M_D(\{1,3,4,5\}) = \{1\}$ holds and 1 is chosen as $\sigma_C(2)$. Then, in the third iteration, $M_D(\{3,4,5\}) = \{3,5\}$ holds and 3 is chosen as $\sigma_C(3)$. Similar steps are repeated until σ_C is completed.

parent A	1	2	3	4	5
parent B	2	1	5	3	4
child C	2	1	3	5	4

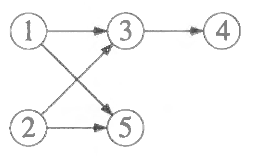
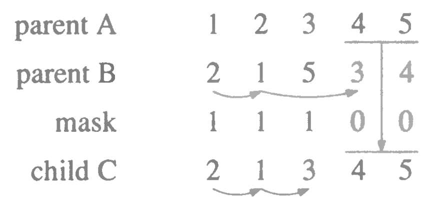

$$\begin{tikzcd} 1 \arrow[r] \arrow[dr] & 3 \arrow[r] & 4 \\ 2 \arrow[r] \arrow[ur] & 5 & \end{tikzcd}$$

Figure 4.4: An example of POPX1.

OX (order crossover): First randomly generate an n bit 0-1 mask $msk \in \{0,1\}^n$, and set $\sigma_C(i) := \sigma_A(i)$ for i with $msk(i) = 0$. Let S_{msk} be $S_{msk} = \{\sigma_A(i) \mid msk(i) = 0\}$, then $D'_B = D_B - \{(i,j) \mid i \in S_{msk} \text{ or } j \in S_{msk}\}$ gives a total order of $V - S_{msk}$. The child σ_C is completed by assigning elements to positions i with $msk(i) = 1$ according to the order of D'_B (see Figure 4.5). As in PMX, we consider 1-point, 2-point and uniform crossover operators, and call them OX(1), OX(2) and OX(U), respectively.

OX was originally proposed in [21] as OX(1) and in [89] as OX(2) independently. It is also introduced in other literature such as [47, 90] as OX(2). In [22] (p. 342~), OX(U) is introduced as two different crossover operators; however, they are the same in our framework.



parent A	1	2	3	4	5
parent B	2	1	5	3	4
mask	1	1	1	0	0
child C	2	1	3	4	5

Figure 4.5: An example of OX(1).

AEX (alternating edge crossover): In this method, a solution is represented by a pointer *next*, where $next(i) = j$ means that element j is ordered next to i. For convenience, we include a dummy element 0 in both ends of the sequence. Then a permutation σ is represented by *next* as

$$\begin{aligned} next(0) &= \sigma(1), \\ next(\sigma(i)) &= \sigma(i+1), \quad i = 1, \ldots, n-1 \\ next(\sigma(n)) &= 0. \end{aligned} \qquad (4.2.4)$$

Let $next_A$, $next_B$ and $next_C$ be the pointer representations of parents A, B and child C, respectively. Then AEX is described as follows, where S is the set of elements not appeared in $next_C$ yet (see Figure 4.6).

1. Set $i := 0$ and $S := V$.

2. Randomly choose j from the set$\{next_A(i), next_B(i)\} \cap S$ if it is not empty; otherwise randomly choose $j \in S$. Then set $next_C(i) := j$.

3. Set $i := j$ and $S := S - \{j\}$. If $S = \emptyset$ holds, set $next_C(i) := 0$ and stop; otherwise return to Step 2.

AEX was originally proposed in [50], which is slightly different from the above definition. In [50], the pointer $next_C(i)$ is chosen from $next_A(i)$ or $next_B(i)$ alternately (from which the name 'alternating' comes) if possible; otherwise randomly chosen from S. Modified versions of this is also proposed in [50, 61].

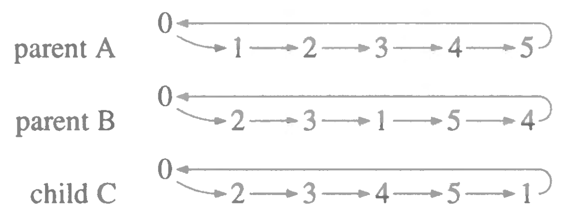


Figure 4.6: An example of AEX.

ERX (edge recombination crossover): In this method, Step 2 of AEX is modified as follows. If $next_A(i)$, $next_B(i) \in S$ holds, instead of choosing the next element j randomly, the parent W (W is A or B) with smaller nonzero value of $|\{next_A(next_W(i)), next_B(next_W(i))\} \cap S|$ is chosen. By using this rule, the element for which fewer pointers are left in S is preferred; hence it is expected that the number of random pointers in $next_C$ is reduced. ERX was originally proposed for TSP in [142], in which the adjacent two elements for each parent (instead of only one next element as above) are considered and is slightly different from the one we explained. It is also introduced in [22] and a modified version is proposed in [118].

Other crossover operators: There are some other permutation crossover operators proposed for TSP, such as subtour exchange crossover [127, 136, 140], sorted match [89] and a similar one [13]. We also tested these operators after modifying them to fit SMP; however, we do not include the results as they are discouraging. Here we note that the original versions of these are reported to be quite effective for TSP and some other sequencing problems. This may be because of the difference in the problem structures. There are also some other permutation crossover operators such as [11, 48, 88]; however, we did not test them, since they are similar to one of the tested crossover operators or combinations of them.

4.3 A General Framework of Crossover

The crossover operators in the previous section are captured by the following general framework.

1. Represent the two parents A and B by the sets of components Π_A and Π_B with which they are defined. Set the component set Π_C of the child to be empty.

2. Choose a new component e to include in Π_C, i.e., $\Pi_C := \Pi_C \cup \{e\}$. Here e is chosen either (i) from $\Pi_A \cup \Pi_B$, or (ii) from those components consistent with the current Π_C. Then, the components conflicting with the resulting Π_C are then removed from $\Pi_A \cup \Pi_B$. (The rules of how to choose e and which of (i) and (ii) is used depends on the crossover operator.)

3. Repeat Step 2 until the child C is uniquely determined by Π_C.

We call the components in $\Pi_C - (\Pi_A \cup \Pi_B)$ as ***non-inherited components***, which are the components in the child C but not in the parents A and B. The crossover operators in the previous section are all described in the above framework by choosing appropriate components from the following definitions:

i) PsR: $\Pi_A = \{(i, \sigma_A(i)) \mid i = 1, \ldots, n\}$,
ii) FLR: $\Pi_A = \{(i, \bar{\sigma}_A(i)) \mid i = 1, \ldots, n\}$,
iii) OR: $\Pi_A = D_A$,
iv) PtR: $\Pi_A = \{(i, next_A(i)) \mid i = 0, \ldots, n\}$,

where the set Π_B is similarly defined, and $\bar{\sigma}$, D_A and *next* were defined in (4.2.2), (4.2.3) and (4.2.4). For example, the component $(i, \sigma_A(i))$ in i) means that the i-th element of parent A is $\sigma_A(i)$. We call the above four as i) position-based representation (denoted PsR), ii) free-list-based representation (denoted FLR), iii) order-based representation (denoted OR) and iv) pointer-based representation (denoted PtR).

For example, PMX can be explained by the above framework by representing the two parents with PsR, that is, $\Pi_A = \{(i, \sigma_A(i)) \mid i \in V\}$ (Π_B is similarly defined). First, set

$\Pi_C := \{(i, \sigma(i)) \in \Pi_A \mid msk(i) = 0\}$, which corresponds to setting $\sigma_C(i) := \sigma_A(i)$ for i with $msk(i) = 0$. Then Π_B is modified by setting $\Pi_B := \Pi_B - \{(j, \sigma_B(j)) \mid \sigma_B(j) = \sigma_A(i)$ for some i with $msk(i) = 0\}$. Set $\Pi_C := \Pi_C \cup \{(i, \sigma_B(i)) \in \Pi_B \mid msk(i) = 1\}$. The child C is completed by adding the non-inherited components by $\Pi_C := \Pi_C \cup \{(i, \sigma_B(\sigma_A^{-1}(\sigma_B(i)))) \mid (i, j) \notin \Pi_C$ for all $j \in V\}$. CX is also explained by representing the parents by PsR. In this case, the rules are designed so that $\Pi_C \subseteq \Pi_A \cup \Pi_B$ holds. OX is explained by representing the parent A by PsR and representing the parent B by OR. Other crossover operators in the previous section are similarly explained within the above framework.

As a natural implementation of a given framework, we can randomly choose the component e in Step 2. Crossover operators different from those in the previous section are sometimes defined by this rule. For PsR, for example, the following crossover operator is made, where $\{W_1, W_2\} = \{A, B\}$ and Π_C is initially set empty.

1. Randomly choose a component $(i, \sigma_{W_1}(i)) \in \Pi_A \cup \Pi_B$ and add it to Π_C. Set $\Pi_{W_1} := \Pi_{W_1} - \{(i, \sigma_{W_1}(i))\}$ and $\Pi_{W_2} := \Pi_{W_2} - (\{(i, \sigma_{W_2}(i))\} \cup \{(j, \sigma_{W_2}(j)) \mid \sigma_{W_2}(j) = \sigma_{W_1}(i)\})$.

2. Repeat Step 1 until $\Pi_A \cup \Pi_B$ becomes empty.

3. Complete the child C by randomly adding into Π_C those components which do not conflict with the current Π_C.

We call this crossover operator as the *position-based random crossover*, which is denoted as PsRND. For OR and PtR, the *order-based random crossover* (denoted ORND) and the *pointer-based random crossover* (denoted PtRND) are similarly defined by the above rule. For FLR, the previous FLX(U) corresponds to this random rule. For PtR, as the pointer *next* is a permutation of $\{0, 1, \ldots, n\}$, we can design a crossover similar to CX, although care must be taken to avoid creating subcycles. We call this as the *pointer-based cycle crossover*, which is denoted as PtCX.

The crossover operators explained in this chapter are categorized into five groups as shown in Table 4.1. From the above consideration, we can conclude that the crossover operators are defined by (i) the representation of the components and (ii) the rule of choosing the components added to the child.

4.4 The Role of Crossover in GA

In this section, we investigate the role of crossover operators in genetic algorithms. Let $C(\chi; A, B) \subseteq F$ (F is the set of all feasible solutions) denote the set of solutions obtainable from the parents A and B by the crossover operator χ (e.g., χ is PMX(2), ERX, etc.). Then, an execution of a crossover can be viewed as the operation of randomly choosing a solution σ

Table 4.1: Classification of crossover operators.

representation	crossover operators
PsR	PMX, CX, PsRND
FLR	FLX
OR	POPX, ORND
PtR	AEX, ERX, PtCX, PtRND
PsR+OR	OX

from $C(\chi; A, B)$. (Note that the probability of choosing a solution is not necessarily uniformly distributed.)

One of the important roles of $C(\chi; A, B)$ is to restrict the search to a promising region. On the other hand, it is not meaningful to restrict $C(\chi; A, B)$ without reason; that is, the set $C(\chi; A, B)$ should include as variety of solutions as possible if they are considered to be promising. We call the achievement of these roles as Objectives 1 and 2, which are summarized as:

Objective 1: Restrict the search to a promising region,

Objective 2: Include as variety of solutions as possible if they are considered to be promising.

The tradeoff between these two objectives is considered to be a key to the success of GA.

To achieve Objective 1, it would be meaningful to inherit as many components as possible from the parents. Therefore, one of the criteria is to include in $C(\chi; A, B)$ those children containing non-inherited components as few as possible (Criterion 1). In GA, as the parents A and B are usually good solutions, it is expected that good solutions are included in $C(\chi; A, B)$ by achieving Criterion 1, if the components used to define the solution reflect the problem characteristics well.

To achieve Objective 2, one of the conceivable criteria is to make the size $|C(\chi; A, B)|$ as large as possible (Criterion 2).

Criteria 1 and 2 usually conflict with each other. That is, if we keep the number of non-inherited components small, the size $|C(\chi; A, B)|$ also becomes small, and if we make $|C(\chi; A, B)|$ large, the number of non-inherited components also becomes large. To evaluate the achievement of the two objectives, we use the following two criteria:

Criterion $1'$: (the smaller the better) the average cost of solutions in $C(\chi; A, B)$,

Criterion $2'$: (the larger the better) the standard deviation of the costs in $C(\chi; A, B)$.

Criteria 1 and 2 can be estimated before we implement the crossover operators, while Criteria $1'$ and $2'$ are not available beforehand.

Criterion 1 can be achieved to some extent by using the general framework of crossover operations in Section 4.3 and putting higher priority to the rule of choosing component e from $\Pi_A \cup \Pi_B$. Actually, it is theoretically shown that the expected number of non-inherited components $|\Pi_C - (\Pi_A \cup \Pi_B)|$ is $c|\Pi_C|$ for some constant c with $0 \le c \le 1$ for most of the crossover operators explained in this chapter (e.g., $c \simeq 1/4$ for PMX(U)). This tendency is also confirmed by the computational experiment in Section 4.5.

To see the achievement of Criterion 2, here we evaluate the size $|C(\chi; A, B)|$. Let $|C(\chi)|$ denote the expectation of $|C(\chi; A, B)|$ if the parents A and B are generated randomly. For PMX, as the number of possible children is determined by the number of possible masks, $|C(\text{PMX}(1))| = O(n)$, $|C(\text{PMX}(2))| = O(n^2)$ and $|C(\text{PMX}(\text{U}))| = O(2^n)$ hold. (Precisely speaking, the same children may be generated from different masks; however, we consider such cases are rare and neglect the effect of them. Actually, even if we take this into account, we can show, for example, that $|C(\text{PMX}(1))| = n - O(\log n) = O(n)$.) For CX, we can show that the expected number of cycles is $O(\log n)$, and hence, $|C(\text{CX}(\text{A}))| = O(1)$, $|C(\text{CX}(1))| = O(\log n)$ and $|C(\text{CX}(\text{U}))| = O(n)$ (the size of CX(U) is based on the experimental data) hold. For PsRND, $|C(\text{PsRND})|$ is considered to be about $k!2^{n-k}$, where k is the number of non-inherited components. The expected number of non-inherited components k is shown to be $k \le n/5$ analytically, and is observed to be about $n/7$ experimentally.

Therefore, the crossover operators of PsR sorted by non-decreasing order of $|C(\chi)|$ are:

CX(A), CX(1), CX(U), PMX(1), PMX(2), PMX(U), PsRND.

For FLX, by the similar discussion with PMX, $|C(\text{FLX}(1))| = O(n)$, $|C(\text{FLX}(2))| = O(n^2)$ and $|C(\text{FLX}(\text{U}))| = O(2^n)$ hold. Therefore, the operators of FLR are sorted as:

FLX(1), FLX(2), FLX(U).

For the operators of OR, we could only show that $|C(\text{POPX2})| = 2^{n-O(\log n)}$ and $|C(\text{ORND})| = 2^{O(n^2)}$ hold. By the relation $C(\text{POPX2}; A, B) \subseteq C(\text{POPX1}; A, B) \subseteq C(\text{ORND}; A, B)$, the order is:

POPX2, POPX1, ORND.

For operators of PtR, $|C(\text{PtCX})| \simeq |C(\text{CX}(\text{U}))|$ and $|C(\text{PtRND})| \simeq |C(\text{PsRND})|$ hold, and sizes $|C(\text{AEX})|$ and $|C(\text{ERX})|$ are considered to be close to $|C(\text{PtRND})|$. By the relation

$C(\text{PtCX}; A, B) \subseteq C(\text{ERX}; A, B) \subseteq C(\text{AEX}; A, B) \subseteq C(\text{PtRND}; A, B)$, the order of these operators is:

PtCX, ERX, AEX, PtRND.

For OX, by the similar discussion with PMX, $|C(\text{OX}(1))| = O(n)$, $|C(\text{OX}(2))| = O(n^2)$ and $|C(\text{OX}(\text{U}))| = O(2^n)$ hold. Therefore, the order is:

OX(1), OX(2), OX(U).

4.5 Computational Results

As the objective of this experiment is to evaluate the crossover operators, the following simple framework of GA is used so that we can avoid interference from other operations and observe the performance of crossover operators as clearly as possible. The population P is set to 100, and all the initial solutions are generated randomly. It is not allowed to include the same solution in the candidate solutions. Selection is executed whenever a crossover is executed, and the worst solution in the candidate solutions is replaced if the child is not already included in the current candidate solutions. Mutation and local search are not incorporated.

The algorithms were coded in C language and run on a workstation Sun SPARC station IPX. The problem instances of SMP were generated according to Subsection 3.6.1. We tested 10 instances for each of $n = 35$ and 100, where the optimal values are known for the instances of $n = 35$ by the exact algorithm SSDP [60].

Table 4.2 shows the following data:

i) The average error in % from the best (optimal for $n = 35$) solution found during the experiments if the algorithms are terminated after 10000 (30000 for PtR) crossover operations for $n = 35$, and 30000 (90000 for PtR) crossover operations for $n = 100$.

ii) Ratio of non-inherited components in the child C.

iii) Analytical order of the expected size $|C(\chi)|$ of the set of children.

iv) The normalized average quality $(cost(\sigma) - \mu_{AB}) / \left(\frac{1}{2}|cost(\sigma_A) - cost(\sigma_B)|\right)$ of $\sigma \in C(\chi; A, B)$ for fixed A and B, where μ_{AB} is the average cost of the parents defined by $\mu_{AB} = (cost(\sigma_A) + cost(\sigma_B))/2$.

v) The standard deviation of the above normalized solution quality for $\sigma \in C(\chi; A, B)$.

The crossover operators are ordered in nondecreasing order of the size $|C(\chi)|$ for each representation (see Table 4.1). We also include the results by the random search (denoted as RND), in which solutions are generated randomly, to give a basis to observe the effectiveness of crossover operators. Data ii) is the results for 10000 independent samples. For data iv) and v), the best one in the initial candidate solutions is chosen as A, and 10-th best solution is chosen as B. Then, the average for 10 problem instances are shown, where 1000 samples were taken for each instance.

Table 4.2: A comparison of various crossover operators.

representation	crossover operator	i) average error from the best (%)		ii) non-inherited components (%)		iii) $\|C(\chi)\|$	iv) average quality in $C(\chi; A, B)$		v) standard deviation in $C(\chi; A, B)$	
	χ	$n = 35$	$n = 100$	$n = 35$	$n = 100$		$n = 35$	$n = 100$	$n = 35$	$n = 100$
PsR	CX(A)	46.5	97.7	0.0	0.0	$O(1)$	0.062	−0.001	1.5	1.4
	CX(1)	47.9	97.0	0.0	0.0	$O(\log n)$	0.007	0.018	1.2	1.1
	CX(U)	61.9	100.0	0.0	0.0	$O(n)$	0.028	−0.004	1.2	1.2
	PMX(1)	54.4	136.3	14.9	15.9	$O(n)$	0.171	0.187	1.1	1.2
	PMX(2)	16.3	47.2	16.1	16.5	$O(n^2)$	0.713	0.798	1.5	1.4
	PMX(U)	8.0	27.7	22.8	24.2	$2^{O(n)}$	0.806	0.807	1.7	1.4
	PsRND	9.4	25.9	12.7	13.2	$2^{O(n)}$	0.467	0.426	1.6	1.4
FLR	FLX(1)	110.2	181.2	0.0	0.0	$O(n)$	0.733	0.649	1.4	1.4
	FLX(2)	82.3	152.6	0.0	0.0	$O(n^2)$	1.048	0.882	1.6	1.4
	FLX(U)	56.5	100.8	0.0	0.0	$2^{O(n)}$	2.069	1.880	1.9	1.6
OR	POPX2	105.9	205.8	0.0	0.0	$2^{O(n)}$	0.023	−0.020	0.5	0.3
	POPX1	67.6	160.9	0.0	0.0	$2^{O(n)}$	−0.155	−0.356	0.8	0.5
	ORND	8.3	12.3	6.5	7.8	$2^{O(n)}$	0.549	0.561	1.6	1.3
PtR	PtCX	116.4	206.9	0.0	0.0	$O(n)$	0.608	0.442	1.4	1.3
	ERX	76.8	122.6	17.2	17.2	$2^{O(n)}$	2.842	2.937	2.0	1.8
	AEX	79.4	135.2	19.5	19.5	$2^{O(n)}$	2.905	3.089	2.1	1.8
	PtRND	16.2	56.9	14.0	13.7	$2^{O(n)}$	2.429	3.120	2.1	1.7
PsR	OX(1)	110.1	190.9	0.0	0.0	$O(n)$	−0.077	−0.117	0.9	1.0
+	OX(2)	10.7	36.0	0.0	0.0	$O(n^2)$	0.379	0.474	1.3	1.2
OR	OX(U)	1.4	3.8	0.0	0.0	$2^{O(n)}$	0.135	0.032	1.5	1.2
	RND	182.0	224.5	100.0	100.0	$O(n!)$	3.440	3.170	2.1	1.7

From the table, it is observed that the ratio of the non-inherited components are small constants for all the crossover operators (except RND). This tendency is also analytically shown, as mentioned in Section 4.4. It is evident that, for crossover operators within the same representation, the quality of solutions becomes better as the order of $|C(\chi)|$ becomes larger. Note that, for most of crossover operators χ with $|C(\chi)| = 2^{O(n)}$, the coefficient of $O(n)$ is close to one (i.e., $|C(\chi)| \approx O(2^n)$) and not much differences exist between them. However, for the three operators of OR, the sizes are $|C(\text{POPX2})| \approx O(2^n)$, $|C(\text{POPX1})| \approx O(2^{2n})$ and $|C(\text{ORND})| \approx O(2^{4n})$, which are quite different. This would be a reason for a big performance change among the three operators in OR.

The average solution quality in $C(\chi; A, B)$ are about the same for all the crossover operators within the same representation. The quality of crossover operators of FLR and PtR are rather poor compared to other representations. Within the same representation, the solution quality becomes worse as the ratio of non-inherited components becomes large, although the differences of the quality are much smaller than those between different representations. The standard deviation of the quality are very small for POPX1 and POPX2, and are rather large

for ERX, AEX and PtRND, but are about the same for the rest of the crossover operators.

These results support the discussions in the previous section. That is, as the number of non-inherited components are small (Criterion 1) for all the crossover operators except RND, the quality of the solution obtained by GA are better for larger $|C(\chi)|$ (Criterion 2) within the same representation. For SMP, the size of order $|C(\chi)| \geq 2^{O(n)}$ seems necessary. Although $|C(\chi)|$ is large, the solution quality of RND is not good, since the number of non-inherited components are quite large (Criterion 1). Rather poor results are observed for crossover operators within FLR and PtR, for which the solution quality in $C(\chi; A, B)$ is also poor (Criterion 1′). The results of POPX1 and POPX2 are not good, for which the standard deviations of the quality of solutions in $C(\chi; A, B)$ are quite small (Criterion 2′).

Here we compared the performance of each criterion only on the basis of the initial candidate solutions. To draw more reliable conclusion, it would also be necessary to observe how the performance changes as the search of GA proceeds. However, it is usually quite hard to evaluate such changes (analytically or numerically) beforehand. Moreover, the objective of this research is to give simple criteria to design good crossover operators. Therefore, we did not consider further details; but it is one of the important future research directions.

Figures from 4.7 to 4.11 show the behavior of average error in % from the best solutions found during the experiments. The results are shown against the number of crossover operations. We chose the number of crossover operations as the horizontal axis instead of computational time, since computational time is affected by the programming skills, which is not essential in this experiment. The crossover operators in each figure (except RND) are arranged from the top in non-decreasing order of the size $|C(\chi)|$. From the figures, it is confirmed that the solution quality becomes better as $|C(\chi)|$ becomes larger within the same representation (Criterion 2). It is also observed that the convergence of crossover operators of OR are fast, but those of PtR are slow. This indicate that the convergence would be faster if the standard deviation of the quality of solutions within $C(\chi; A, B)$ is smaller. Note that good solution is not necessarily obtained by a fast convergence.

As a whole, good performances are observed for OX(U) and crossover operators in which the rule of choosing components are random (e.g., PsRND, ORND, etc.). The definition of these crossover operators are quite natural, and they achieve the proposed two criteria to some extent. On the other hand, as in the case of 1-point and 2-point crossover operators, it is meaningless to restrict the size $|C(\chi)|$ by the rules which are not essential.

We conclude that achieving the proposed two criteria is important to design good crossover operators. However, in general, as the size $|C(\chi)|$ and the standard deviation of the quality of solutions in $C(\chi; A, B)$ become larger, the number of non-inherited components also becomes larger and the average solution quality in $C(\chi; A, B)$ becomes worse. Therefore, it is important to evaluate the tradeoff between the two criteria. From these considerations, we propose the following guideline for the design of crossover operators:

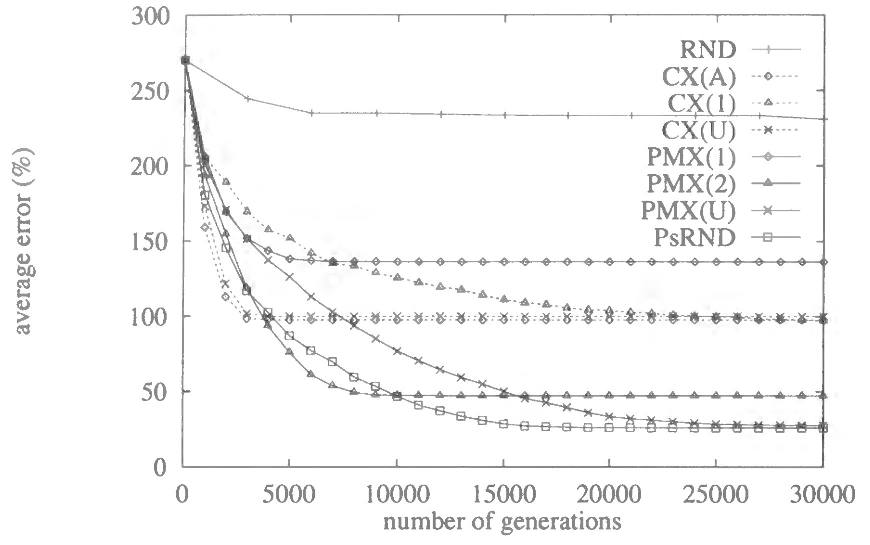


Figure 4.7: A comparison of crossovers (PsR).

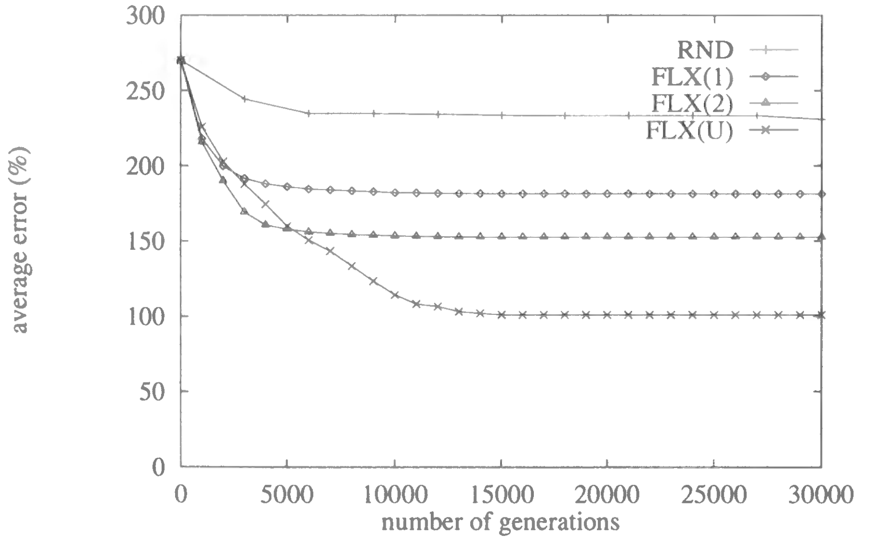


Figure 4.8: A comparison of crossovers (FLR).

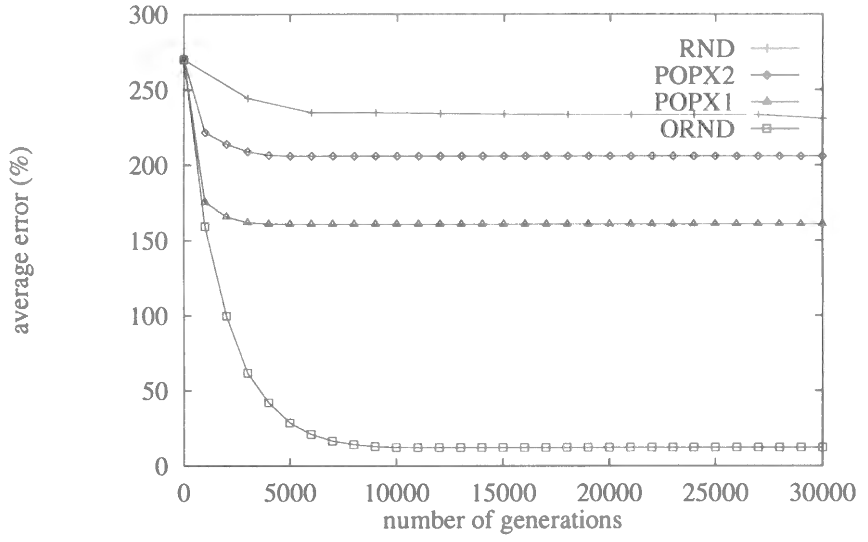


Figure 4.9: A comparison of crossovers (OR).

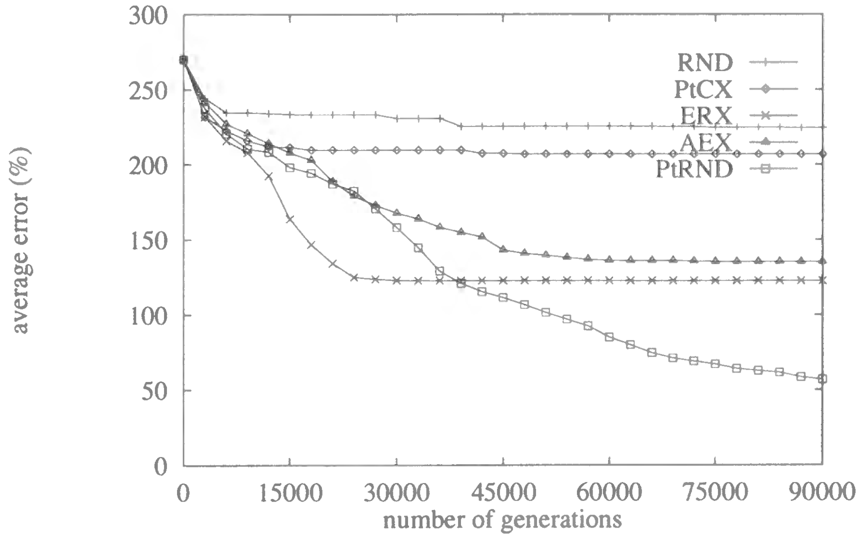


Figure 4.10: A comparison of crossovers (PtR).

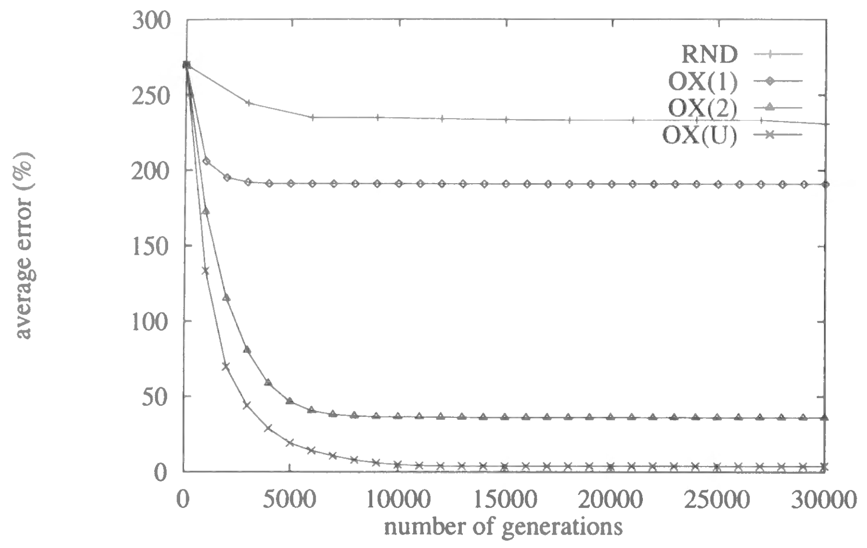


Figure 4.11: A comparison of crossovers (PsR+OR).

1. Make the size $|C(\chi)|$ larger than $2^{O(n)}$ while keeping the number of non-inherited components as small as possible.

2. Choose good solution representation which capture the problem characteristics well so that the average solution quality in $C(\chi; A, B)$ becomes better.

It would also be worth trying to combine more than one representation as in the case of OX. Between the above two rules, 1 can be evaluated before designing crossover operators; however, 2 is difficult to predict and can only be evaluated after crossover operators are implemented. Therefore, the above guideline may be useful to compare crossover operators within the same representation; however, deep insight into the problem structure is needed to choose good solution representation.

4.6 Conclusion

In this chapter, we compared various crossover operators proposed for sequencing problems from the view point of general framework. It is confirmed that the performance of the crossover operators can be evaluated by some simple criteria related to characteristics of the set $C(\chi; A, B)$ of children obtainable from the parents A and B. These criteria are expected to give a useful guideline in designing good crossover operators for genetic algorithms.

The results in Section 4.5 indicate that crossover operators with larger $|C(\chi)|$ are preferable. This result is partially due to the framework of GA used in our experiments. We did not incorporate mutations and used the selection strategy with high selection pressure. Therefore, to keep the divergence in the candidate solutions, it was important to have variety of solutions in the set $C(\chi, A, B)$ of children. There are other strategies to increase the variety of candidate solutions, such as incorporating mutations, using larger population, employing the selections with lower pressure, and so on. Comparing the effectiveness of such strategies is one of the important future research directions. Incorporating other strategies, such as local search (i.e., genetic local search [126]) and exact algorithms [134], is essential to make GA competitive with other optimization tools. Examining such hybrid approaches is also important.

The framework of GA is quite flexible and there are various ways to improve its performance. This robustness is one of the attractive features of GA; however, from the view point of users, the algorithms should be as simple as possible. In this sense, it is important to simplify the framework and analyze the effect of each basic operation to the performance of GA. The research of this chapter may contribute in this research direction.

Chapter 5

Metaheuristics as Robust and Simple Optimization Tools

5.1 Introduction

One of the attractive features of metaheuristics is in its simplicity and robustness. They can be developed even if deep mathematical properties of the problem domain are not at hand, and still can exhibit reasonably good performance, much better than those obtainable by simple heuristics. In this chapter, we pursue this direction more carefully, by implementing various metaheuristics and comparing their performance. The objective is not to propose the most powerful algorithm but to compare general tendencies of various algorithms. The emphasis is placed not to make each ingredient of such metaheuristics too sophisticated, and to avoid detailed tuning of the program parameters involved therein, so that practitioners can easily test the proposed framework to solve their problems of applications. As a concrete problem to test, we solve in this chapter the single machine scheduling problem (SMP).

We test various metaheuristics, such as random multi-start local search (MLS), genetic algorithm (GA), simulated annealing (SA) and tabu search (TS), using rather simple inside operators. The results indicate that: (1) simple implementation of MLS is usually competitive with (or even better than) GA, (2) GA combined with local search is quite effective if longer computational time is allowed, and its performance is not sensitive to crossovers, (3) SA is also quite effective if longer computational time is allowed, and its performance is not much dependent on parameter values, (4) there are cases in which TS is more effective than MLS; however, its performance depends on how to define the tabu list and parameter values, and (5) the definition of neighborhood is very important for all of MLS, SA and TS.

5.2 Design of Metaheuristic Algorithms

Some details of the tested algorithms and the computational results are discussed in this section. All the tested algorithms were coded in C language and run on a Sun SPARC station IPX. The quality of the obtained solutions is evaluated by the average error from the best cost values, which were found in the entire experiment. The efficiency of algorithms is measured on the basis of the number of the solution samples evaluated, rather than the computational time, since the computational time depends on the computers used and other factors such as programming skill. Ten problem instances for each of $n = 35$ and 100 are generated as described in Subsection 3.6.1.

Initial solutions are generated randomly except for GRASP, and the neighborhood $N(\sigma)$ is always scanned according to a prespecified random order.

5.2.1 Random Multi-Start Local Search

The performance of LS and MLS critically depends on: (1) the definition of $N(\sigma)$ and (2) the search strategy (i.e., how to search the solutions in $N(\sigma)$). In our experiment, only the following strategies are examined from the view point of simplicity.

(1) Neighborhoods: $N_{ins}(\sigma) = \{\sigma_{k \leftarrow l} \mid k \neq l\}$ and $N_{swap}(\sigma) = \{\sigma_{k \leftrightarrow l} \mid k \neq l\}$. Here $\sigma_{k \leftarrow l}$ is the sequence obtained from σ by moving the l-th job to the location before the k-th job, while $\sigma_{k \leftrightarrow l}$ is obtained by interchanging the k-th job and l-th job of σ.

(2) Search strategies: FA scans $N(\sigma)$ and selects the first improved solution σ' satisfying $cost(\sigma') < cost(\sigma)$, and BA selects the solution σ' having the best cost in the entire area of $N(\sigma)$.

The average error (%) of the best solutions obtained by these four combinations are shown in Table 5.1, where 3×10^5 and 3×10^6 samples were generated for each test run with $n = 35$ and 100, respectively. Table 5.1 also shows the average number of trials (i.e., the number of initial solutions) in parentheses.

Table 5.1: Average error in % of the best solutions (average number of initial solutions) with MLS.

	$n = 35$; 3×10^5 samples		$n = 100$; 3×10^6 samples	
	N_{ins}	N_{swap}	N_{ins}	N_{swap}
FA	0.000 (97.6)	0.000 (131.0)	0.669 (103.6)	0.182 (126.5)
BA	0.289 (8.6)	0.047 (13.9)	4.696 (3.2)	0.624 (5.0)

These results indicate that: (1) Search strategy FA obtains good solutions earlier than

search strategy BA. (Based on this, the search strategy is fixed to FA in the remaining experiments.) (2) The quality of solutions obtained by neighborhood N_{swap} is better than that obtained by N_{ins}.

5.2.2 Greedy Randomized Adaptive Search Procedure

This procedure is called GRASP and was explained in Section 2.2. The initial solutions are generated as follows. At each step, let $M = \{i \in V \mid i \text{ is not scheduled yet}\}$. A candidate set $CA \subseteq M$ of a fixed number of jobs ($|CA|$ is a prespecified positive integer) is chosen according to a criterion based on a local gain function that represents greedy heuristics, and then a job $i \in CA$ is randomly chosen as the next job.

A total of 12 local gain functions $e_i(\text{f})$ and $e_i(\text{b})$ defined in [132] (see also Subsection 3.3.2) and parameter values $|CA| = 1, 2, 4, 7, 10, 20$ are tested to generate initial solutions of GRASP, where $|CA| = 1$ means the conventional greedy methods. In this subsection, only the results with the neighborhood N_{swap} are shown; however, similar tendencies were observed for N_{ins}. Table 5.2 shows the average error (%) of the best solutions obtained by

Table 5.2: Average error in % of the best solutions with GRASP using N_{swap}.

$\lvert CA\rvert$	1	2	4	7	10	20	init
e_1(f)	0.183	0.156	0.228	0.170	0.169	0.151	50.5
e_1(b)	0.205	0.309	0.193	0.223	0.230	0.154	53.8
e_2(f)	1.524	1.292	1.034	0.653	0.446	0.214	28.6
e_2(b)	1.339	1.117	0.650	0.423	0.270	0.150	33.6
e_3(f)	0.383	0.335	0.202	0.136	0.077	0.163	8.8
e_3(b)	0.568	0.398	0.193	0.220	0.137	0.183	16.0
e_4(f)	0.170	0.099	0.121	0.097	0.153	0.189	18.2
e_4(b)	0.466	0.392	0.214	0.308	0.205	0.241	72.7
e_5(f)	0.120	0.171	0.117	0.147	0.069	0.145	73.2
e_5(b)	0.138	0.109	0.122	0.119	0.095	0.131	90.6
e_6(f)	0.172	0.135	0.053	0.106	0.109	0.107	116.0
e_6(b)	0.158	0.123	0.129	0.135	0.088	0.138	64.3
MLS				0.182			314.6

various GRASP algorithms within 3×10^6 samples. For comparison purpose, the last row, MLS, indicates the average error when initial solutions are generated randomly. The last column, init, indicates the average error of the initial solutions generated by greedy methods, i.e., with $|CA| = 1$, and the bottom is the average error of the initial solutions generated randomly.

The results indicate that: (1) Performance of GRASP critically depends on the local gain functions used for generating initial solutions. (2) If the local gain function is properly chosen, GRASP improves the performance of MLS to some extent. However the performance is hardly affected by the parameter $|CA|$. (3) A local gain function which produces better initial solutions does not always lead to better performance of GRASP. In other words, GRASP is simple and can be powerful than MLS, but not robust with the local gain functions used.

5.2.3 Iterated Local Search

Here we employed the framework of chained local optimization (see Section 2.2). In Step 2 of ILS, a solution σ is randomly chosen from $N'(\sigma_{seed})$, where either N_{ins} of N_{swap} is used for N' and is denoted as INS or SWAP, respectively. In Step 4, σ_{seed} is chosen randomly according to the following rule: If $cost(\sigma) < cost(\sigma_{seed})$, set $\sigma_{seed} := \sigma$; otherwise set $\sigma_{seed} := \sigma$ with probability $e^{-\Delta/t}$, where $\Delta = cost(\sigma) - cost(\sigma_{seed})$ and t is a prespecified parameter. In our experiment, parameter t is fixed and the adaptive control of t such as used in simulated annealing is not incorporated. Here we tested parameter values $t = 0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, \infty$. Table 5.3 shows the average error (%) of the best solutions, where 3×10^6 samples were generated for $n = 100$. For $n = 35$, optimal solutions were found for most cases, and the result is omitted here.

The results indicate that: (1) ILS is effective compared to MLS, (2) ILS is more effective for N_{swap} than for N_{ins}, (3) the performance of ILS is not sensitive to the perturbation rule in Step 2, i.e., both INS and SWAP gives good results, and (4) smaller t gives better results and sufficient quality is usually obtained with $t = 0$. In conclusion, ILS is simple (even simpler than GRASP) and can be more powerful than MLS.

5.2.4 Genetic Algorithm

The framework of GA (see Section 2.3) we examined is as follows: at each generation, generate a set of solutions $\mathcal{Q} \subseteq N(\mathcal{P})$ and select a set $\mathcal{P}'$ of P solutions from $\mathcal{P} \cup \mathcal{Q}$, and set $\mathcal{P} := \mathcal{P}'$. Recall that $N(\mathcal{P})$ is the set of solutions obtainable from $\mathcal{P}$ by crossover and mutation operators. Among various types of crossover, mutation and selection operators, we considered the following representative operators.

Crossover: The order crossover OX [21, 89] (see also Section 4.2) is employed here. We assume that one child σ_C is produced from two parents σ_A and σ_B. Generate randomly an n-bit mask $msk \in \{0,1\}^n$. Set $\sigma_C(k) := \sigma_A(k)$ for all k satisfying $msk(k) = 0$. Define $D'_B = D_B - \{(i,j) \mid i \in S_{msk} \text{ or } j \in S_{msk}\}$, where $D_B = \{(i,j) \mid \sigma_B^{-1}(i) \le \sigma_B^{-1}(j)\}$ and $S_{msk} = \{\sigma_A(k) \mid msk(k) = 0\}$. Then D'_B is the total order of σ_B restricted to $V - S_{msk}$.

Table 5.3: Average error in % of the best solutions with ILS for $n = 100$ after 3×10^6 samples.

	N_{ins}		N_{swap}	
t	INS	SWAP	INS	SWAP
0	0.312	0.360	0.008	0.001
1	0.424	0.401	0.015	0.024
2	0.362	0.570	0.012	0.023
4	0.523	0.334	0.025	0.001
8	0.422	0.205	0.012	0.005
16	0.668	0.272	0.009	0.027
32	0.726	0.660	0.023	0.009
64	0.830	0.527	0.054	0.011
128	0.892	0.689	0.046	0.060
256	1.136	0.661	0.075	0.032
512	0.936	1.064	0.125	0.056
1024	0.926	1.158	0.138	0.064
∞	1.211	1.249	0.105	0.061
MLS	0.669		0.182	

Complete σ_C by assigning jobs to all the positions k satisfying $msk(k) = 1$ according to D'_B. Crossover operators based on arbitrary masks are called uniform, and those based on restricted masks having at most k adjacent 0-1 pairs are called k-point; e.g., masks 11000 and 10011 are 1-point and 2-point respectively. We call 1-point, 2-point and uniform crossover operators of this type as OX(1), OX(2) and OX(U) respectively.

Other types of crossover operators, such as partially mapped crossover [46] and cycle crossover [90, 132] (see Section 4.2), were also examined; however, the results are omitted here, since the results for OX are better and the tendency is similar with other operators.

Mutation: Mutation employed in this experiment perturbs a candidate solution σ by a random selection $\sigma' \in N(\sigma)$ and $\sigma := \sigma'$. Two types of neighborhood $N_{ins}(\sigma)$ and $N_{swap}(\sigma)$ are used as $N(\sigma)$; the resulting mutations are denoted as INS and SWAP respectively.

Selection: In our experiment, the set of generated candidate solutions $\mathcal{Q} \subseteq N(\mathcal{P})$ is determined as follows. We use $Q = 1$ and the child $\sigma_C \in \mathcal{Q}$ is obtained by randomly selecting two parents σ_A, $\sigma_B \in \mathcal{P}$, mutating either σ_A or σ_B, and then applying crossover of σ_A and σ_B. Then with a solution σ_{worst} satisfying $cost(\sigma_{worst}) \ge cost(\sigma)$ for all $\sigma \in \mathcal{P} \cup \mathcal{Q}$, let $\mathcal{P}' := \mathcal{P} \cup \{\sigma_C\} - \{\sigma_{worst}\}$. Note that the selection is executed only if the child $\sigma_C \in \mathcal{Q}$ is not in $\mathcal{P}$.

Various GA defined by the above crossover, mutation and selection operators are compared, in which P is always set to 100. Note that exactly one sample (i.e., cost evaluation) occurs during one generation, since $Q = 1$ is used. Table 5.4 shows the average error (%) of the best solutions obtained by the tested algorithms, where 3×10^4 (resp., 3×10^5) samples were allowed for $n = 35$ (resp., 100). Here, 'MUT only' means that crossover is not used and 'no MUT' means that mutation is not used. The results of MLS are also included for comparison, where the same neighborhood as mutation is used.

Table 5.4: Average error (%) of the best solutions with various versions of GA with $P = 100$.

	$n = 35$; 3×10^4 samples			$n = 100$; 3×10^5 samples		
	no MUT	INS	SWAP	no MUT	INS	SWAP
OX(1)	110.1	5.1	3.5	190.9	3.1	2.1
OX(2)	10.5	0.2	0.8	35.9	2.6	0.7
OX(U)	1.4	0.3	1.1	3.8	0.8	0.8
MUT only	—	2.3	4.0	—	1.4	1.5
MLS	—	0.1	0.06	—	1.0	0.5

These results indicate that: (1) Using mutations is essential to get good solutions within the framework of GA. (2) Performance of GA is not sensitive against the types of crossover operators if combined with mutation, though it critically depends on the types of crossover operators if mutation is not used. (3) Crossover is also effective to improve GA, since GA with MUT only needs slightly more samples than GA with crossover to obtain solutions of similar quality. (4) MLS performs better than GA of this sort.

GA using different population sizes P are also tested. Table 5.5 shows the average error in % of the best solutions after 3×10^6 samples were generated, where $n = 100$ and crossover type is OX(U). For comparison purpose, the results of MLS are also included in Table 5.5, where the same neighborhood as the mutation is used.

Table 5.5: Average error in % of the best solutions with GA after 3×10^6 samples.

P	10^1	10^2	10^3	10^4	MLS
INS	4.876	0.641	0.595	0.524	0.669
SWAP	0.804	0.802	0.538	0.327	0.182

The results indicate that: (1) Better solutions are obtained on average as P increases, at the cost of testing more number of samples. (2) The quality of solutions obtained by MLS is still slightly better than those results of GA.

Note that much more computational time is needed to sample a solution with GA compared to other algorithms, such as MLS. We may conclude that the effectiveness of simple GA is in question.

5.2.5 Genetic Local Search

Genetic local search (GLS) is a variation of GA, in which the new candidate solutions in Q are improved by LS. Other operators are the same as GA. Various GLS were compared, in which P is set to 20. The results for $n = 35$ are omitted, since almost all the tested algorithms could obtain exact optimal solutions for all the instances. Table 5.6 shows the average error (%) of the best solutions for $n = 100$, where 3×10^6 samples were generated. The results of MLS are also included for comparison purposes.

Table 5.6: Average error (%) of the best solutions with GLS in which $P = 20$.

neighbor	N_{ins}			N_{swap}		
mutation	noMUT	INS	SWAP	noMUT	INS	SWAP
OX(1)	0.281	0.361	0.303	0.090	0.065	0.069
OX(2)	0.220	0.327	0.272	0.038	0.052	0.047
OX(U)	0.203	0.164	0.266	0.015	0.069	0.038
MUT only	—	0.598	0.510	—	0.103	0.057
MLS		0.669			0.182	

We can summarize these results as follows. (1) GLS can obtain solutions of higher quality than GA and MLS, particularly when long computational time is allowed. (2) GLS is rather insensitive to the types of crossover and mutation. Between crossover and mutation, crossover appears slightly more effective. On the other hand, GLS only with mutations is much easier to implement, since mutation can be realized by using the neighborhood of LS, and hence the additional efforts required is very little. (3) The solution quality critically depends on the type of neighborhood.

ILS can be viewed as a special case of GLS in which $P = 1$ and crossover is not incorporated. By comparing Tables 5.3 ($R = 0$) and 5.6 (column 'MUT only'), we can conclude that the performances of GLS and ILS are similar. More computational results for MLS, GA and GLS are found in [132].

5.2.6 Simulated Annealing

The SA used in this experiment is similar to the one in [64] (see also Section 2.4). Our algorithm includes parameters IP, TR, SF and TF. The initial temperature t is determined so that

$$\left(\sum_{\sigma' \in UP(\sigma)} e^{-\{cost(\sigma') - cost(\sigma)\}/t} \right) / |UP(\sigma)| \simeq IP$$

for randomly chosen initial solutions σ, where $UP(\sigma) = \{\sigma' \in N(\sigma) \mid cost(\sigma') > cost(\sigma)\}$. Then the following loop is executed, where k is initially set to 0.

```
While $k < TR \cdot |N|$ do the following.
    (a) Perform the following loop $SF \cdot |N|$ times.
        i. Pick a random neighbor $\sigma' \in N(\sigma)$.
        ii. Let $\Delta = cost(\sigma') - cost(\sigma)$.
        iii. If $\Delta \le 0$, set $\sigma = \sigma'$.
        iv. If $\Delta > 0$, set $\sigma = \sigma'$ with probability $e^{-\Delta/t}$.
        v. If $\Delta < 0$, set $k = 0$; otherwise set $k = k + 1$.
    (b) Set $t = TF \cdot t$.
```

Upon termination, the search is restarted from a randomly chosen initial solution unless sufficient number of solution samples has been tested.

The parameter TR is set to 1 for N_{ins} and 2 for N_{swap} according to a preliminary experiment. The parameter TF is fixed to 0.95 as suggested in [64].

First the effect of IP is examined, and it is observed that: (1) the quality of the obtained solutions becomes better as IP increases up to 0.1; however, it does not change much if $IP \geq 0.1$, and (2) the number of samples needed for one trial of the algorithm becomes larger as IP increases.

Next the effect of SF is examined with $IP = 0.3$. It is observed that: (1) the quality of the solutions becomes better as SF increases up to 1; however, it does not change much if $SF \geq 1$, and (2) the number of samples needed for one trial of the algorithm becomes larger as SF increases.

From these results, we examined four combinations of IP and SF: $IP = 0.1, 0.3$ and $SF = 1, 2$. Table 5.7 shows the average error (%) of the best solutions, where 3×10^5 (resp., 3×10^6) samples were allowed for $n = 35$ (resp., $n = 100$). The results of MLS are also included for comparison.

We can summarize these results as follows. (1) SA can obtain solutions of higher quality than MLS, provided that rather long computational time is allowed. (2) The quality of the

Table 5.7: Average error (%) of the best solutions with SA.

		$n = 35$; 3×10^5 samples		$n = 100$; 3×10^6 samples	
IP	SF	N_{ins}	N_{swap}	N_{ins}	N_{swap}
0.1	1	0.144	0.000	0.184	0.034
0.1	2	0.124	0.003	0.217	0.032
0.3	1	0.035	0.000	0.153	0.029
0.3	2	0.033	0.000	0.227	0.014
MLS		0.000	0.000	0.669	0.182

solutions obtained by SA is rather insensitive to the parameter values, though the number of samples needed for one trial critically depends on them. (3) The solution quality critically depends on the type of neighborhood.

5.2.7 Threshold Accepting and Great Deluge Algorithm

In the threshold accepting (TA) (see Section 2.4), we use four parameters IP, TR, SF, TF, as in the case of simulated annealing. The initial threshold τ is determined so that

$$\{\sigma' \in UP(\sigma) \mid cost(\sigma') - cost(\sigma) \le \tau\} / |UP(\sigma)| \simeq IP$$

holds for randomly chosen initial solution σ. Then a similar loop with SA is repeated, where Step (a)-iii is replaced with

iii′. If $\Delta < \tau$, set $\sigma := \sigma'$.

and parameter t is replaced with τ.

As in the case of SA, the parameter TF is set to 0.95, and the parameter TR is set to 1 for N_{ins} and 2 for N_{swap} according to a preliminary experiment. Then the effect of the parameters IP and SF are examined; however, the tendency is not very clear.

Based on these observations, we examined ten combinations of IP and SF: $IP = 0.1, 0.3$ and $SF = 0.25, 0.5, 1, 2, 4$. Table 5.8 shows the average error (%) of the best solutions, where 3×10^5 (resp., 3×10^6) samples were allowed for $n = 35$ (resp., $n = 100$). The results of MLS are also included for comparison. We can observe that the performance of TA is competitive with SA.

The framework of the great deluge algorithm (GDA) (see Section 2.4), employed in this experiment is as follows. Two parameters RS (called *rain speed*) and TR are included. The first water level is set to $W := cost(\sigma)$ for a randomly chosen initial solution σ. Then the following loop is executed, where k is initially set to 0.

Table 5.8: Average error (%) of the best solutions with TA.

		$n = 35$; 3×10^5 samples		$n = 100$; 3×10^6 samples	
IP	SF	N_{ins}	N_{swap}	N_{ins}	N_{swap}
0.1	0.25	0.003	0.000	0.093	0.031
0.1	0.5	0.000	0.000	0.155	0.037
0.1	1	0.243	0.003	0.196	0.006
0.1	2	0.376	0.000	0.326	0.041
0.1	4	0.364	0.132	0.385	0.019
0.3	0.25	0.000	0.000	0.130	0.031
0.3	0.5	0.101	0.018	0.165	0.031
0.3	1	0.082	0.023	0.200	0.039
0.3	2	0.146	0.023	0.401	0.029
0.3	4	0.246	0.023	0.336	0.040
MLS		0.000	0.000	0.669	0.182

While $k < TR \cdot |N|$ holds, do the following.

(a) Randomly choose a solution σ' from $N(\sigma)$.

(b) If $cost(\sigma') < W$ holds, set $\sigma := \sigma'$ and $W := W - RS(W - cost(\sigma'))$.

(c) If $cost(\sigma') - cost(\sigma) < 0$ holds, set $k := 0$; otherwise set $k := k + 1$.

The parameter TR is set to 1 for N_{ins} and 2 for N_{swap} according to a preliminary experiment. Then the effect of the parameter RS is examined within the range of $RS = 0.00125 \sim 0.64$, and it is observed that (1) the solution quality becomes better as RS becomes smaller, (2) the number of samples needed for one iteration becomes larger as the RS becomes smaller.

Based on these, we examined the parameter values $RS = 0.005, 0.01$. Table 5.9 shows the average error (%) of the best solutions, where 3×10^5 (resp., 3×10^6) samples were allowed for $n = 35$ (resp., $n = 100$). The results of MLS are also included for comparison. We can observe that the performance of GDA is also competitive with SA.

It is well-known that the search of SA converges to a global optimum under certain conditions (e.g., [79]), and this result is sometimes considered to give support for the success of SA. However, similar (but rather weaker) result is also known for TA [7]. In such convergence results, the asymptotic behavior of algorithms when the number of iterations tends to infinity are discussed, and such analyses do not necessarily explain the performance within limited computational time. In addition, TA and GDA gave competitive results with SA in our experiments.

Table 5.9: Average error (%) of the best solutions with GDA.

	$n = 35$; 3×10^5 samples		$n = 100$; 3×10^6 samples	
RS	N_{ins}	N_{swap}	N_{ins}	N_{swap}
0.005	0.069	0.228	0.239	0.048
0.01	0.000	0.015	0.227	0.027
MLS	0.000	0.000	0.669	0.182

5.2.8 Tabu Search

In the tabu search (see Section 2.5), we scan $N(\sigma)$ and select the first solution σ' satisfying $cost(\sigma') < cost(\sigma)$ and $\sigma' \notin T \cup \{\sigma\}$, or satisfying $cost(\sigma') < best$ (i.e., *aspiration criterion*), where *best* is the cost of the best solution found duing the past search. If none of the solutions in $N(\sigma)$ is selected by the above rule (i.e., σ is locally optimal), the next solution is selected as follows. A counter MC and a parameter R are used to control the process of generating the next initial solution when one trial of tabu search ends. If $MC < R$ holds, then the best solution in $N(\sigma) \backslash (\{\sigma\} \cup T)$ is chosen as the next initial solution, and MC is incremented by one; otherwise the solution $\sigma' \in N(\sigma) \backslash (\{\sigma\} \cup T)$ that minimizes $cost(\sigma') + \alpha \cdot penalty(\sigma')$ is chosen as the next solution and MC is reset to zero, where α is a prespecified program parameter and $penalty(\sigma')$ is a cost of the long term memory. Finally, if a solution better than the past best solution is found (aspiration criterion), MC is reset to zero.

Two types of tabu lists T_{job} and T_{pos} are considered, where $T_{job}(\sigma) = \{\sigma' \in N(\sigma) \mid$ the move from σ to σ' changes the position of a job whose position has been changed in the last TT moves$\}$ and $T_{pos}(\sigma) = \{\sigma' \in N(\sigma) \mid$ the move from σ to σ' assigns a job to the position where it has been assigned in the last TT moves$\}$. The parameter TT is a prespecified nonnegative integer called *tabu tenure*. We examined two types of penalties of long term memory, which are called $penalty_{move}$ and $penalty_{period}$. Let $LT_{move}(i, k)$ be the number of moves of job i from position k which have been made during the past search, and $LT_{period}(i, k)$ be the period that job i has been scheduled at position k so far. Then we define $penalty_{move}(\sigma') = \sum_{j \in CH(\sigma')} LT_{move}(j, \sigma^{-1}(j))$ and $penalty_{period}(\sigma') = \sum_{k=1}^{n} LT_{period}(\sigma'(k), k)$, where $CH(\sigma')$ is the set of jobs whose positions are changed by the move from σ to σ'.

First α is set to 0 in order to examine the effect of tabu lists (short term memory). Two types of tabu list T_{job} and T_{pos}, and various TT values are tested. It is observed that: (1) TS can obtain solutions of higher quality than MLS if N_{swap} and T_{pos} are used and $TT = 1 \sim 5$, and (2) the performance of TS is worse than MLS with other combinations. From these, we consider only T_{pos} in the remaining experiments.

Second the effect of the long term memory is examined. Two types of penalties $penalty_{move}$ and $penalty_{period}$, and various α values are tested. It is observed that: (1) TS can obtain solutions of better quality than MLS if N_{swap} and $penalty_{period}$ are used with $\alpha \geq 10^4$, and (2) the performance of TS is worse than MLS with other combinations. From these, we consider only $penalty_{period}$ in the remaining experiments.

Finally the short and long term memories are combined together. Here the parameter α is set to 10^3 (resp., 10^4) for $n = 35$ (resp., 100). Parameter values $TT = 1, 3, 5, 7, 10, 20$ are examined and R is set to TT. Table 5.10 shows the average error (%) of the best solutions obtained by TS, where 3×10^5 (resp., 3×10^6) samples were generated for $n = 35$ (resp., 100).

Table 5.10: Average error (%) from the best solutions with TS.

	$n = 35$; 3×10^5 samples		$n = 100$; 3×10^6 samples	
TT	N_{ins}	N_{swap}	N_{ins}	N_{swap}
1	0.000	0.000	1.449	0.086
3	0.037	0.000	0.965	0.150
5	0.084	0.000	1.647	0.093
7	0.358	0.000	1.689	0.167
10	0.360	0.000	1.684	0.148
20	0.542	0.000	1.899	0.204
MLS	0.000	0.000	0.669	0.182

We can summarize these results as follows. (1) TS can obtain solutions of higher quality than MLS if N_{swap} is used, and its performance is not sensitive to parameter TT. (2) TS does not improve the performance of MLS if N_{ins} is used. (3) The performance of TS critically depends on the type of memories (the tabu list and the long term memory).

5.3 Comparison of Metaheuristics

We conclude this chapter by comparing four metaheuristic algorithms tested so far. Figures 5.1 and 5.2 show how the average error (%) of the best solutions improves as the number of samples increases for problem instances with $n = 100$. Both neighborhoods N_{ins} (Fig. 5.1) and N_{swap} (Fig. 5.2) are examined, where in the case of GA, mutation operators INS and SWAP are used corresponding to N_{ins} and N_{swap}, respectively. For GRASP with neighborhood N_{ins} (resp., N_{swap}), local gain function e_6 (b) (resp., e_6 (f)) is used and parameter $|CA|$ is set to 7 (resp., 4). For ILS with neighborhood N_{ins} (resp., N_{swap}), parameter t is

set to 8 (resp., 0). Crossover operator OX(U) is used for GA and GLS. The parameter P is set to 1000 for GA and 20 for GLS, and mutation is not incorporated for GLS. For SA with neighborhood N_{ins} (resp., N_{swap}), parameters are set to $IP = 0.3$, $TR = 1$, $SF = 1$ and $TF = 0.95$ (resp., $IP = 0.3$, $TR = 2$, $SF = 2$ and $TF = 0.95$). For TA with neighborhood N_{ins} (resp., N_{swap}), parameters are set to $IP = 0.1$, $TR = 1$, $SF = 0.25$ and $TF = 0.95$ (resp., $IP = 0.1$, $TR = 2$, $SF = 1$ and $TF = 0.95$). For GDA with neighborhood N_{ins} (resp., N_{swap}), parameters are set to $TR = 1$ and $RS = 0.01$ (resp., $TR = 2$ and $RS = 0.01$). For TS with neighborhood N_{ins} (resp., N_{swap}), tabu list T_{pos} and penalty $penalty_{period}$ are used and the parameters are set to $R = 3$, $TT = 3$ and $\alpha = 10^4$ (resp., $R = 5$, $TT = 5$ and $\alpha = 10^4$).

From these results, we can conclude that: (1) Performance of GA is robust about mutation; however, its performance is rather poor. (2) GRASP, ILS, GLS and SA improve the performance of MLS further, among which ILS, SA and GLS appear more powerful if the same neighborhood is used. (3) Performance of MLS, GRASP, ILS, GLS, SA and TS critically depends on the type of neighborhood used.

In view of these, we can summarize our recommendation about the use of metaheuristic algorithms as 'simple optimization tools' as follows.

1. If the simplicity is our first concern, use MLS. In this case, the component to be defined is only the neighborhood.

2. If obtaining solutions of higher quality is important, first try ILS, since ILS is simpler than other metaheuristics.

3. If the performance of ILS is not sufficient, use SA or GLS.

5.4 Conclusion

In this chapter, various metaheuristic algorithms were compared from the view point of robustness and simplicity. As a concrete problem to test, we chose the single machine scheduling problem (SMP) and metaheuristics such as the multi-start local search (MLS), the genetic algorithm (GA), the simulated annealing (SA), the tabu search (TS), and some of their variants were examined. A guideline to design metaheuristic algorithms was proposed in the previous section, based on the computational results. These results were limited to a single problem, and it is important to conduct similar comparisons on basis of various types of problems so that we can understand the general tendencies of the metaheuristic algorithms.

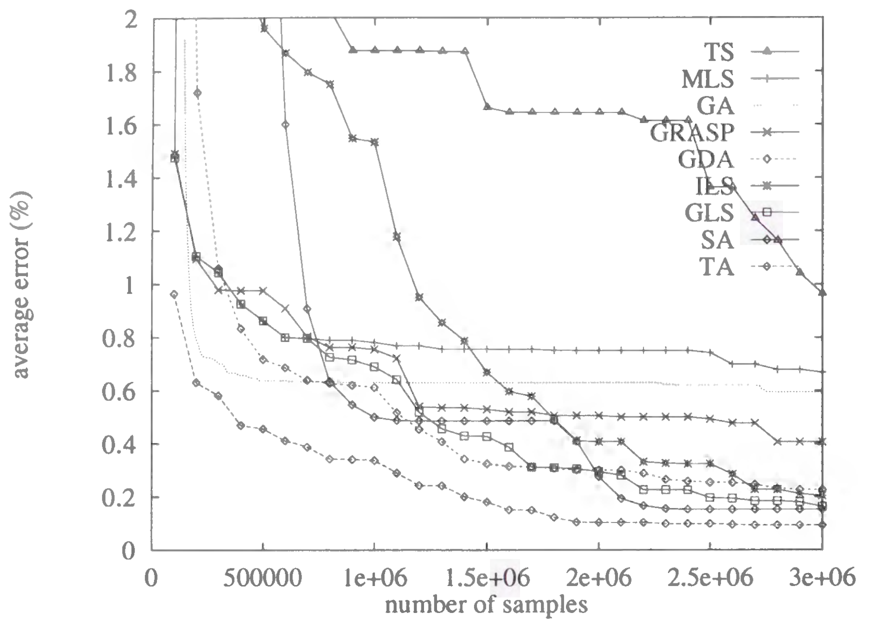


Figure 5.1: Average error (%) from the best solution (N_{ins}).

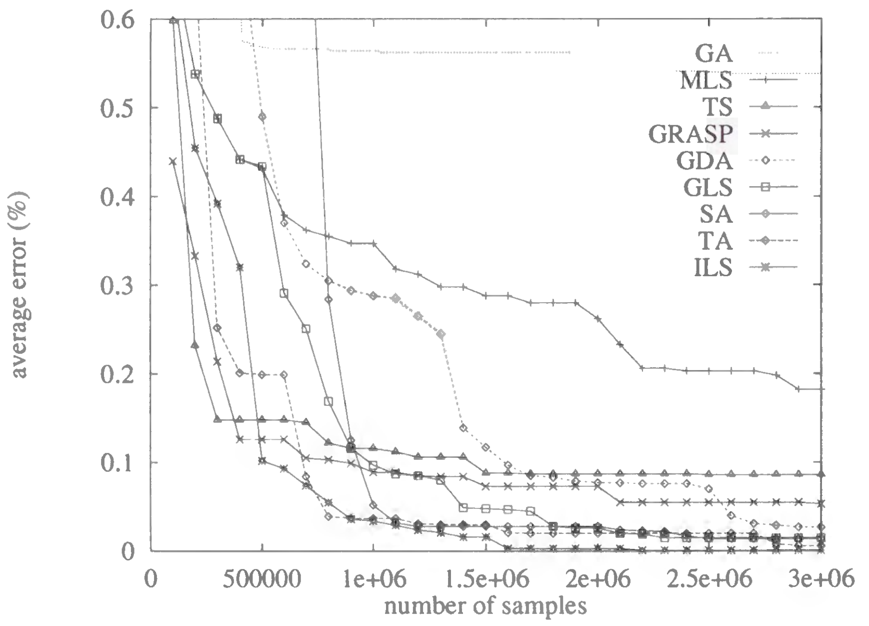


Figure 5.2: Average error (%) from the best solution (N_{swap}).

Chapter 6

Enumerating All Common Intervals of Two Permutations

6.1 Introduction

Two permutations σ_A and σ_B of set $V = \{1, \ldots, n\}$ are given as the input, where $\sigma_A(i) = j$ (or $\sigma_A^{-1}(j) = i$) denotes that j is the i-th element of σ_A (σ_B is similarly defined). Let $[x, y]$ denote the index set $\{x, x+1, \ldots, y\}$. We call a pair of intervals $([x_A, y_A], [x_B, y_B])$ $(1 \le x_A < y_A \le n, 1 \le x_B < y_B \le n)$ a *common interval* if it satisfies

$$\{\sigma_A(i) \mid i \in [x_A, y_A]\} = \{\sigma_B(i) \mid i \in [x_B, y_B]\}. \tag{6.1.1}$$

The length of a common interval $([x_A, y_A], [x_B, y_B])$ is defined to be $y_A - x_A + 1$.

Some genetic algorithms based on common intervals have been proposed for sequencing problems (e.g., traveling salesman problem, job shop scheduling problem, etc.) and have exhibited good prospects [13, 70, 89, 140].

In this chapter, we consider enumeration of all common intervals of length 2 to n. Three algorithms are proposed, which are improved versions of a simple $O(n^2)$ time algorithm proposed in [136]:

1. A simple $O(n^2)$ time algorithm (called LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations.
2. A practically fast $O(n^2)$ time algorithm (called MNG) using the reverse Monge property.
3. An $O(n + K)$ time algorithm (called RC), where K $(\le \binom{n}{2})$ is the number of outputs.

It will be also shown that the expected number of common intervals of length 2 to $n-2$ for two random permutations is $2 + O(n^{-1})$. This implies that the expected number of common intervals of length 2 to n is $O(1)$, since the number of common intervals of length $n-1$ or n is at most 3. This result gives a reason for the phenomenon that the expected time complexity $O(n)$ of the algorithm LHP is independent of the output length K. We also give an example for which both LHP and MNG requires $\Omega(n^2)$ time, although $K = O(n)$.

Among the three algorithms proposed in this chapter, RC is most desirable from the theoretical point of view; however, it is quite complicated compared to LHP and MNG. Therefore, it is possible that RC is slower than the other two algorithms in some cases. For this reason, computational experiments for various types of problems with up to $n = 10^6$ are conducted. The results indicate that

1. LHP and MNG are much faster than RC for two randomly generated permutations (e.g., LHP is about 13 times faster than RC).
2. MNG is rather slower than LHP for random inputs; however, there are cases that LHP requires $\Omega(n^2)$ time, but MNG runs in $o(n^2)$ time and is faster than both LHP and RC.

A recommendation about the use of the three algorithms is discussed in Section 6.8, based on the computational results.

These results are also applicable to similar problems defined on two cyclic permutations [136, 131].

6.2 Basic Algorithm

Here, we describe the basic $O(n^2)$ time algorithm [136], which is the starting point of all the algorithms proposed in this chapter. For convenience, we denote the function $\sigma_B^{-1} \cdot \sigma_A$ by π_{AB} (i.e., $\pi_{AB}(i) = \sigma_B^{-1}(\sigma_A(i))$ holds for all i, and $\pi_{AB}(i) = j$ means that the i-th element of σ_A is located in the j-th position of σ_B) throughout this chapter, which can be calculated from σ_A and σ_B in $O(n)$ time. We also define the following functions for an interval $[x, y]$ of σ_A:

$$l(x,y) = \min_{i\in[x,y]} \pi_{AB}(i) \qquad (6.2.2)$$

$$u(x,y) = \max_{i\in[x,y]} \pi_{AB}(i) \qquad (6.2.3)$$

$$f(x,y) = u(x,y) - l(x,y) - (y-x). \qquad (6.2.4)$$

Since $f(x,y)$ is the number of elements in $\{\sigma_B(i) \mid i \in [l(x,y), u(x,y)]\}\backslash\{\sigma_A(i) \mid i \in [x,y]\}$, a pair $([x,y], [l(x,y), u(x,y)])$ is a common interval if and only if $f(x,y) = 0$. Then all common intervals can be enumerated by calculating $f(x,y)$ for all (x,y) pairs satisfying $1 \le x < y \le n$. This gives rise to the following algorithm.

Algorithm BSC

```
Line 1: for $x = 1, \ldots, n-1$ do
```

```
Line 2:     $l$:=$u$:=$\pi_{AB}(x)$;
Line 3:     for $y = x+1, \ldots, n$ do
Line 4:         $l$:=min$\{l, \pi_{AB}(y)\}$;
Line 5:         $u$:=max$\{u, \pi_{AB}(y)\}$;
Line 6:         if $u - l - (y-x) = 0$ then
Line 7:             output ($[x, y]$, $[l, u]$)
Line 8:     end for
Line 9: end for.
```

The variables u and l in BSC correspond to the function values $u(x,y)$ and $l(x,y)$ defined above. The time complexity of this algorithm is $O(n^2)$, since Lines 4, 5, 6 and 7 can be executed in $O(1)$ time.

6.3 Simple Improvements of the Basic Algorithm

In this section, we propose two improved versions of BSC, called LHP and MNG, both of which detect some redundant inner loop iterations from Line 3 to 8 of BSC by simple tests, and remove them from execution. They still require $O(n^2)$ time in the worst case; however, it is observed that they are practically much faster than BSC for many types of problems.

6.3.1 The Algorithm LHP

Here we describe the algorithm LHP. It is shown in Section 6.5 that the expected running time of this algorithm for two randomly generated permutations is $O(n)$. For convenience, only the common intervals of length 2 to $n-2$ are considered in this subsection, and Line 3 of BSC is modified as

"Line 3′: **for** $y = x+1, \ldots, \min\{n, x+n-3\}$ **do**".

Modification of the algorithm to the original problem (where common intervals of length 2 to n are considered) is easy and the results of this chapter are not affected by this assumption by the following reasons. The pair of intervals of length n (i.e., $([1,n],[1,n])$) is always a common interval. There are four pairs of intervals, $([1,n-1],[1,n-1])$, $([1,n-1],[2,n])$, $([2,n],[1,n-1])$ and $([2,n],[2,n])$, which are the candidates for common intervals of length $n-1$. The pair of intervals $([1,n-1],[1,n-1])$ is a common interval if and only if $\pi_{AB}(n) = n$. The other cases are similar. Therefore, we can enumerate all common intervals of length $n-1$ in constant time by checking if $\pi_{AB}(1) = 1$, $\pi_{AB}(1) = n$, $\pi_{AB}(n) = 1$ or $\pi_{AB}(n) = n$ holds. We improve the basic algorithm BSC in the following two respects.

The first is that, if

$$u - l > \min\{n-x, n-3\} \qquad (6.3.5)$$

is satisfied just before entering Line 6 of BSC in the x-th iteration, then the rest of current inner loop can be omitted, and we move into the $(x+1)$st iteration immediately. Note that $u-l$ is monotonically nondecreasing during the x-th iteration. Condition (6.3.5) implies that the length of interval $[l,u]$ of σ_B exceeds the maximum length of interval $[x,y]$ of σ_A when y is increased up to $\min\{n, x+n-3\}$. We call this condition *length condition.*

Let HP be the set

$$\begin{aligned} HP &= V\backslash\{\pi_{AB}(w) \mid w = x, x+1, \ldots, \min\{n, x+n-3\}\} \\ &= \{\pi_{AB}(w) \mid w \in [1, x-1] \text{ or } w \equiv x-2 \pmod n \text{ or } w \equiv x-1 \pmod n\}. \end{aligned}$$

The second is that, if an $h \in HP$ satisfies

$$l < h < u \tag{6.3.6}$$

just before entering Line 6 of BSC, then the rest of the current inner loop can be omitted. HP is the set of indices of the elements which will not be included in any interval $[x,y]$ ($y = x+1, \ldots, \min\{n, x+n-3\}$) of σ_A. We call each element of HP a *hole point*, and call condition (6.3.6) *HP condition.* It is not advantageous to check the HP condition for all $h \in HP$, since the whole running time increases to $O(n^3)$. Hence, we check the HP condition for only a sufficiently small portion of HP, which we call HP', so that the original worst case time complexity $O(n^2)$ is preserved. For this, $|HP'|$ should be kept constant. After trying several in preliminary computational experiments, we choose HP' as follows:

$$HP' = \{\pi_{AB}(w) \mid w \equiv x-2 \pmod n \text{ or } w \equiv x-1 \pmod n\}. \tag{6.3.7}$$

As other natural candidates, one may consider

$$\begin{aligned} HP_1 &= \{\pi_{AB}(w) \mid w \in [1,n] \text{ and } w \equiv x-1 \pmod n\} \text{ or} \\ HP_2 &= \{\text{an element randomly chosen from } HP\}. \end{aligned} \tag{6.3.8}$$

However, it is observed that $O(n \log n)$ average time is needed for two randomly generated permutations if we use HP_1, and it is also observed that the algorithm becomes slower if we use HP_2 (one of the conceivable reasons for this phenomenon is that generating random values frequently is too expensive). More discussion is in [131].

6.3.2 The Algorithm MNG

Here we describe the second algorithm MNG. It uses the fact that the function f defined by (6.2.4) satisfies the reverse Monge property, that is,

$$f(x',y) + f(x,y') \geq f(x',y') + f(x,y) \tag{6.3.9}$$

holds for all x', x, y, y' satisfying $x' < x \leq y < y'$ (see Appendix A for the proof). From (6.3.9), we have

$$\begin{aligned} f(x,y') &\geq f(x,y) - \{f(x',y) - f(x',y')\} \\ &\geq f(x,y) - \{f(x',y) - \min_{z\in[y+1,n]} f(x',z)\}. \end{aligned} \tag{6.3.10}$$

Since the above inequalities hold for every x' $(< x)$,

$$f(x,y') \geq f(x,y) - \min_{w\in[1,x-1]}\{f(w,y) - \min_{z\in[y+1,n]} f(w,z)\} \tag{6.3.11}$$

holds. The value of $\min_{w\in[1,x-1]}\{f(w,y) - \min_{z\in[y+1,n]} f(w,z)\}$ gives an upper bound for the decrease of $f(x,y)$ when y is increased up to n. Hence, if $x \geq 2$ and

$$f(x,y) - \min_{w\in[1,x-1]}\{f(w,y) - \min_{z\in[y+1,n]} f(w,z)\} > 0 \tag{6.3.12}$$

holds just before entering Line 8 of BSC in the x-th iteration, then the rest of the current inner loop can be omitted, and we can move to the $(x+1)$st iteration immediately.

Now let y_{last} be defined as the value of y at Line 9 when we exit the inner loop. If $y_{last} \leq n-1$, then we will not complete computing $\min_{z\in[y_{last}+1,n]} f(x,z)$. Hence, we may fail to check condition (6.3.12) for larger x. Thus we define a function

$$LD(x,y) = \begin{cases} \infty, & (x=1, y=2,3,\ldots,n-1) \\ \min\{LD(x-1,y), f(x-1,y) - \min_{z\in[y+1,n]} f(x-1,z)\}, & \\ & (x\geq 2, y=x,\ldots,y_{last}-1, y_{last}=n) \\ \min\{LD(x-1,y), f(x-1,y) - \min\{\min_{z\in[y+1,y_{last}]} f(x-1,z), & \\ \quad f(x-1,y_{last}) - LD(x-1,y_{last})\}\}, & \\ & (x\geq 2, y=x,\ldots,y_{last}, y_{last}\leq n-1) \\ LD(x-1,y), & (x\geq 2, y=y_{last}+1,\ldots,n-1, y_{last}\leq n-1). \end{cases} \tag{6.3.13}$$

The function $LD(x,y)$ can be calculated even if $y_{last} \leq n-1$, and satisfies

$$LD(x,y) \geq \min_{w\in[1,x-1]}\{f(w,y) - \min_{z\in[y+1,n]} f(w,z)\}. \tag{6.3.14}$$

An inner loop can be terminated if condition

$$f(x,y) - LD(x,y) > 0 \tag{6.3.15}$$

holds. The correctness of the algorithm is retained even after this modification, since condition (6.3.15) implies condition (6.3.12). We call condition (6.3.15) *Monge condition.*

We defined LD as a function of both x and y for convenience; however, the value of $LD(x,y)$ can be overwritten on the memory space of $LD(x-1,y)$ in the actual execution. Such an update of LD is executed every time we exit the inner loop, which is possible in

$O(y_{last}-x)$ time. Hence, the worst case running time $O(n^2)$ of the algorithm BSC is preserved for MNG.

We further set a parameter $R \in (0,1]$, and do not exit the inner loop for $y > R(n-x)+x$ even if Monge condition is satisfied. ($R=1$ means the case we do not use this modification.) Once $y > R(n-x)+x$ holds, y_{last} is forced to be n and we can update LD by using the second formula of (6.3.13); hence, LD value may improve by this modification. The total time spent to inner loops increases at most $1/R$ times compared to the case with $R=1$. We set R to 0.5 in the computational experiments, since remarkable improvement was observed in some problem instances compared to $R=1$.

6.3.3 Remarks about the Two Algorithms

Two algorithms LHP and MNG can be combined; however, slight modifications are needed in updating LD. It would be worth trying to terminate the inner loop by length condition, HP condition or Monge condition only if $y < R(n-x)+x$ for a parameter $R \in (0,1]$. Since the computational time gains at most $1/R$ times of the algorithm LHP, expected running time of this combined algorithm remains $O(n)$ for two randomly generated permutations. It is also noted that some LD values may become larger than those realized by MNG alone, and this combined algorithm will not necessarily improve the performance of MNG.

Although it is observed that algorithms of this type are much faster than the algorithm BSC for many types of problems, they always require $\Omega(n^2)$ time for some problem instances. For example, consider the problem given by setting $\sigma_A(i) = i$ $(i=1,\ldots,n)$ and

$$\sigma_B(i) = \begin{cases} 2i-1, & i \le \lceil n/2 \rceil \\ 2(n-i+1), & i \ge \lceil n/2 \rceil + 1. \end{cases}$$

The function f then takes

$$\begin{aligned} f(x,y) &> 0, \quad x=1,\ldots,n-1, \quad y=x+1,\ldots,n-1 \\ f(x,n) &= 0, \quad x=1,\ldots,n-1 \end{aligned}$$

and the number of outputs is $K=O(n)$. Any algorithm improved from BSC by "omitting redundant loops" requires $\Omega(n^2)$ time for this example, since the inner loop must be repeated until y becomes n for all x. It shows a limitation of the algorithms of this type.

6.4 An Algorithm with $O(n+K)$ Worst Case Running Time

In this section, we propose an algorithm called the *reduce candidate algorithm* (abbreviated as RC) which runs in $O(n+K)$ time in the worst case. Since the algorithm runs in time proportional to the number of inputs and outputs, it is optimal in the sense of the worst case

time complexity. On the other hand, those algorithms proposed in the previous section may take much time, e.g., $\Omega(n^2)$ time even if the number of outputs K is $O(n)$, though they are very simple and fast for most of the tested problem instances.

For a fixed x, we call a y *unnecessary* if it satisfies $f(x',y) > 0$ for all $x' \le x$. By definition, if y is unnecessary for x, y is also unnecessary for x'' for all $x'' \le x$. The main idea of the algorithm RC is to save the time to check whether $f(x,y)=0$ or not for some y which can be concluded as unnecessary from the past search information. The framework of the algorithm is described as follows.

```
Algorithm RC
Line 1: $Y := \{n\}$.
Line 2: for $x = n-1, \cdots, 1$ do
Line 3:     Output all $y$ $(> x)$ in $Y$ satisfying $f(x,y)=0$.
Line 4:     Set $Y := (Y \cup \{x\}) \backslash W$
              where $W \subseteq \{y \in N \mid y \ge x$ and $f(x',y) > 0$ for all $x' < x\}$.
Line 5: end for.
```

The key to this algorithm is how to find unnecessary y's. The following lemmas help us to identify them. Note that $u(x,y) \le u(x',y')$ and $l(x,y) \ge l(x',y')$ hold for $[x,y] \subseteq [x',y']$.

Lemma 4.1 *Suppose that we are given $x>1$ and $y>x$. If $u(x,y) < u(x,y')$ and $u(x-1,y) = u(x-1,y')$ hold for some $y' > y$, y satisfies $f(x',y) > 0$ for all $x' < x$.*

Proof. From $u(x,y) < u(x,y')$, there exists a $y'' \in [y+1,y']$ satisfying $\pi_{AB}(y'') \in [u(x,y)+1, u(x,y')]$. By $u(x-1,y) = u(x-1,y')$, we have $[u(x,y)+1, u(x,y')] \subseteq [l(x',y), u(x',y)]$ and $\pi_{AB}(y'') \in [l(x',y), u(x',y)]$. As y'' is not included in $[x',y]$, $f(x',y)$ is greater than 0. □

Lemma 4.2 *Suppose that we are given $x>1$ and $y>x$. If $f(x,y) > f(x,y')$ hold for some $y' > y$, y satisfies $f(x',y) > 0$ for all $x' \le x$.*

Proof. From $f(x,y) > f(x,y')$, there exists a $y'' \in [y+1,y']$ which satisfies $\pi_{AB}(y'') \in [l(x,y), u(x,y)]$. Since y'' is not included in $[x',y]$, $f(x',y)$ is greater than 0. □

We can find a part of unnecessary y from these properties. We will show an algorithm that removes all y that satisfy the conditions of Lemma 4.1 or 4.2 from the set Y at Line 4 of algorithm RC.

To maintain Y, the algorithm uses a doubly linked list, *ylist*, composed of the cells $y_1, \ldots, y_r$ corresponding to the elements $y \in Y$. The cells are sorted in increasing order of their values. Initially, the *ylist* is composed of only one element n. Then Line 4 of algorithm RC is realized by adding an element x at the head of *ylist* and executing algorithm TRIMMING_YLIST(x,y)

explained below. For simplicity, we consider only the case with $\pi_{AB}(x-1) > \pi_{AB}(x)$ throughout this section. The opposite case can be treated similarly. The algorithm for trimming the wastful y from *ylist* is as follows.

```
Algorithm TRIMMING_YLIST($x, y$)
($x$ and $y$ are set to the values in Line 4 of algorithm RC.)
Step 1: Find $y^* \in N$ which is maximum among those $y$ satisfying $u(x, y) < u(x-1, y)$.
Step 2: If the cell $y$ on the head of ylist satisfies $u(x, y) < u(x, y^*)$, then remove it from
        ylist (from Lemma 4.1) and go to Step 2; otherwise go to Step 3.
Step 3: Let $y_i$ and $y_{i+1}$ be adjacent in ylist and satisfy $y_i \leq y^* < y_{i+1}$. If $f(x-1, y_i)$
        $> f(x-1, y_{i+1})$ then remove $y_i$ from ylist (from Lemma 4.2) and go to Step 3.
```

Algorithm TRIMMING_YLIST(x, y) correctly remove all the elements concluded as unnecessary by Lemmas 4.1 and 4.2 by the following reasons. In Step 2, if there exists a $y' \leq y^*$ satisfying $u(x, y') < u(x, y^*)$, then the head y of *ylist* also satisfies $u(x, y) < u(x, y^*)$, since $u(x, y)$ is monotonically nondecreasing with y. Therefore, all y satisfying $u(x, y) < u(x, y^*)$ are removed from *ylist* during the iteration of Step 2, i.e., all the elements concluded as unnecessary by Lemma 4.1 are removed.

For Lemma 4.2, we claim that $f(x-1, y_i) \leq f(x-1, y_{i+1})$ hold for all y_i $(\geq x)$ which remain in the *ylist* at the end of the algorithm TRIMMING_YLIST. This is proved by induction on x. Suppose that $f(x, y_i) \leq f(x, y_{i+1})$ holds for all i just before x is added at the head of *ylist* in Line 4 of algorithm RC. This hypothesis is true for $x = n - 1$, since the *ylist* is initially composed of only one element n. As $f(x, x) = 0$ and $f(x, y) \geq 0$ $(\forall x \leq y)$ hold, $f(x, y_i) \leq f(x, y_{i+1})$ still holds for all i after x is added at the head of *ylist*. For every $y_i > y^*$, $f(x-1, y_i) - f(x, y_i) = -1$ holds, since $u(x, y_i)$ and $l(x, y_i)$ are unchanged for such i. For every $y_i \leq y^*$ which is not removed in Step 2 of TRIMMING_YLIST, $f(x-1, y_i) - f(x, y_i) = c$ (c is a constant satisfying $c \geq 0$) holds (i.e., c is the same for all $y_i \leq y^*$), since $u(x, y_i)$ are the same for all such i and $l(x, y_i)$ are unchanged. Thus the claim was proved.

Given x and y, we have to spend $O(y - x)$ time to calculate $u(x, y)$ if no particular data structure is used. To achieve linear time, we have to obtain them in shorter time. In our algorithm, we represent the functions u and l by lists called *ulist* and *llist*. For a fixed x, $u(x, y)$ (resp., $l(x, y)$) is monotonically nondecreasing (resp., nonincreasing) in y. (See Figure 6.1.) We now describe the construction of the linked list only for u, since the construction of *llist* is similar. The interval $[x+1, n]$ is decomposed into intervals $[y'_0 = x+1, y'_1 - 1], [y'_1, y'_2 - 1], \ldots, [y'_{r-1}, y'_r = n]$ where $u(x, y'') = u(x, y''')$ holds if and only if both y'' and y''' are included in $[y'_i, y'_{i+1} - 1]$. From this decomposition, we represent u by *ulist* composed of the cells which correspond to these intervals. Each cell keeps the corresponding interval and the value $u(x, y)$ for y which the interval includes. A pair of cells are doubly

linked by pointers if they correspond to adjacent intervals. We say that y is included in the cell of *ulist* if the corresponding interval includes y.

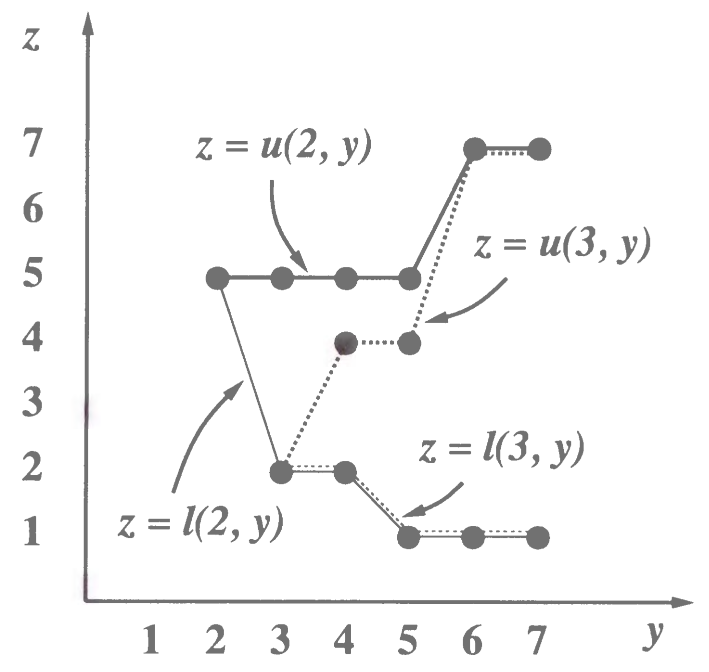


Figure 6.1: Functions $u(2, y)$, $u(3, y)$, $l(2, y)$ and $l(3, y)$ corresponding to permutations $\sigma_A = \langle 1, 2, 3, 4, 5, 6, 7\rangle$ and $\sigma_B = \langle 5, 3, 1, 4, 2, 7, 6\rangle$.

To get the value of $u(x, y)$, we have to find the cell in *ulist* which includes y. To realize this operation in short time, we prepare a pointer from each cell y_i of *ylist* to the cell of *ulist* which includes y_i. We also prepare a pointer from each cell of *ulist* to the cell y_i of *ylist*, where y_i is the maximum among those included in the same cell of *ulist*. (See Figure 6.2.)

The update of *ulist* and *llist* when x changes to $x - 1$ is executed as follows. We update *llist* by adding a cell corresponding to interval $[x-1, x-1]$ on its head. (Recall that we treat only the case $\pi_{AB}(x-1) > \pi_{AB}(x)$.) We delete all the cells of *ulist* which include a y such that $u(x, y) < u(x, y^*)$. For the cell including y^*, we change its interval to $[x-1, y^*]$ and its value from $u(x, y^*)$ to $u(x-1, y^*)$. (See Figure 6.3.) Note that we do not remove the cell representing $u(x, y^*)$, but use it to represent $u(x-1, y^*)$. By doing this, pointers from all y included in the cell corresponding to $u(x, y^*)$ to *ulist* need not to be changed. This is a key point in speeding up of the algorithm.

In Step 2 of TRIMMING_YLIST, if the pointer from a cell y of *ylist* indicates a deleted cell of *ulist*, we remove it from *ylist*, since this implies $u(x, y) < u(x, y^*)$. Thus it is not necessary to update the pointers between *ylist* and *ulist*.

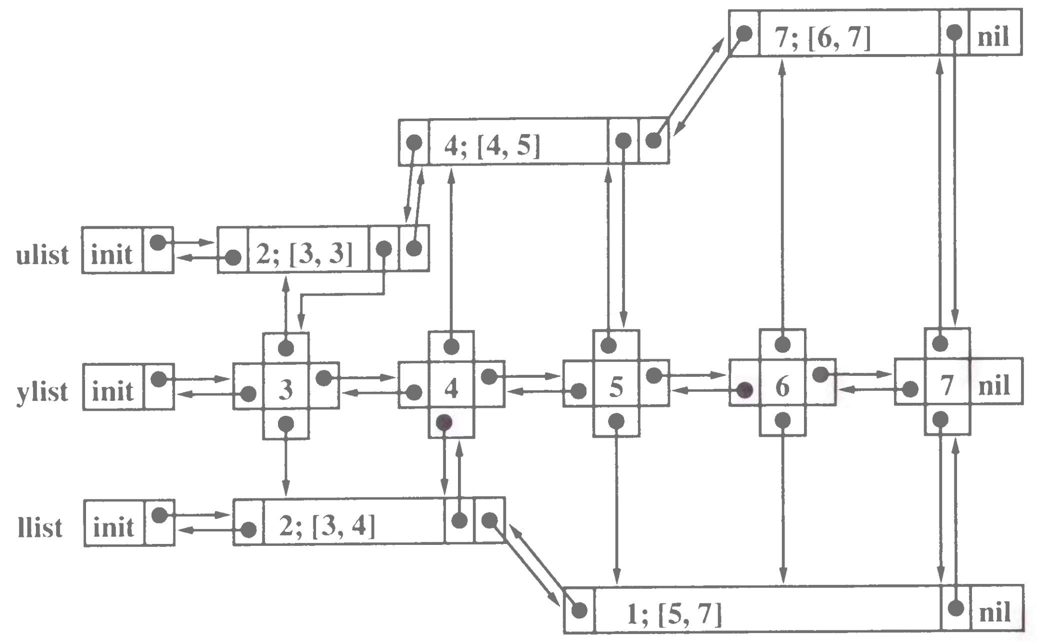


Figure 6.2: Examples of *ulist* and *llist* corresponding to $u(3, y)$ and $l(3, y)$ of Figure 6.1.

Now we consider the time complexity of algorithm RC. For this purpose, let us consider the time to update *ylist*, *ulist* and *llist* and the time to scan *ylist* to output common intervals in the entire algorithm of RC. Since those update operations of *ulist* are done by tracing *ulist* from its head to the cell including y^*, Step 1 and 2 of the algorithm TRIMMING_YLIST take $O(d+1)$ time, where d is the number of deleted cells in Step 2. The total number of deleted cells during the execution of the algorithm RC can not exceed the number of created cells, which is $O(n)$, and thus the total time of those operations in the algorithm RC is $O(n)$.

In Step 3 of the algorithm TRIMMING_YLIST, we can find y_i and y_{i+1} in $O(1)$ time by tracing a pointer from the cell of *ulist* including y^* to the cell of *ylist*. (See Figure 6.3.) Step 3 is repeated while the current cell is deleted. This is done in time proportional to the number of the deleted cells. Thus the total time spent in Step 3 of the algorithm TRIMMING_YLIST in all iterations of algorithm RC is proportional to the total number of the deleted cells. It can not exceed the number of created cells, and the total time is $O(n)$.

In Line 3 of algorithm RC, the cells $y_1, \cdots, y_r$ of *ylist* satisfy $f(x, y_i) \leq f(x, y_{i+1})$ $(i = 1, \ldots, r-1)$. Therefore we can enumerate all y satisfying $f(x, y) = 0$ by tracing *ylist* from its head without scanning y with $f(x, y) > 0$ in the middle. When we encounter a y with $f(x, y) > 0$, we stop the tracing since $f(x, y') > 0$ holds for all $y' > y$. It takes time proportional to the number of outputs, which is $O(n+K)$.

As a result, the following theorem holds.

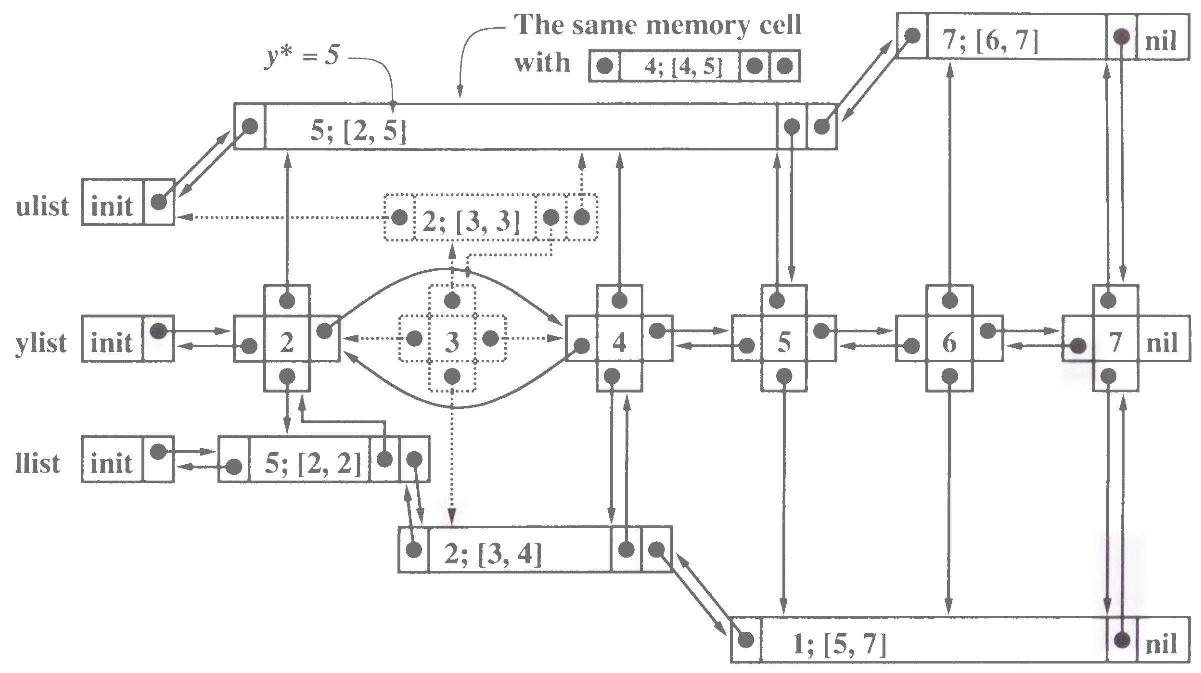


Figure 6.3: The process of updating *ulist*, *llist* and *ylist*. The cells represented by dotted lines are deleted when *ulist* is updated.

Theorem 4.1 *Algorithm RC with* TRIMMING_YLIST *outputs all common intervals in* $O(n+K)$ *time.*

6.4.1 Enumerating Common Intervals within a Specified Length

Given $b_l \leq b_u \leq n$, we consider the problem of enumerating all the common intervals of two permutations whose length are not smaller than b_l and not greater than b_u. This problem is motivated by the following reason. If the given two permutations are similar, the number of common intervals of length 2 to n will be very large (e.g., $O(n^2)$). Even in such cases, the number of common intervals of length b_l to b_u may be much smaller if $b_u - b_l$ is small (e.g., the number of outputs is $O(n)$ if $b_u - b_l = O(1)$). Of course we can enumerate common intervals of length b_l to b_u by first enumerating all common intervals of length 2 to n and then outputting those with the specified lengths, but this algorithm requires $O(K)$ time, where K is the number of common intervals of length 2 to n. However, we can do better by using algorithm RC with slight modifications.

In each iteration, we keep the minimum cell $\hat{y}$ of *ylist* among those satisfying $\hat{y} - x + 1 \geq b_l$. At the end of Line 4 of algorithm RC, we find the minimum cell $\hat{y}'$ satisfying $\hat{y}' - x + 2 \geq b_l$ and set $\hat{y} := \hat{y}'$. Since $\hat{y}'$ is either adjacent to $\hat{y}$ in the *ylist* or $\hat{y}' = \hat{y}$, this update can be done in $O(1)$ time. The enumeration of y satisfying $f(x,y)=0$ and $b_l \le y-x+1 \le b_u$ can be done in $O(n+K')$ time by tracing *ylist* from $\hat{y}$, where K' is the number of outputs for this problem.

6.4.2 Finding the Common Interval of Maximum Length within a Specified Length

In this subsection, we consider the problem finding a common interval of the maximum length whose length is less than or equal to a given number b_u $(< n)$. The motivation of considering this problem is similar to that explained in Subsection 6.4.1.

The basic idea is similar to the above algorithm. We keep the maximum cell $\bar{y}$ satisfying $\bar{y}-x+1 > b^*$ and *ylist* is scanned from $\bar{y}$ in Line 3 of algorithm RC, where $b^* \le b_u$ is the maximum length of the common intervals which the algorithm found so far.

At the end of each iteration of algorithm RC, we update b^* if the common interval whose length is not more than b_u and is larger than b^* is found. In such a case, we update $\bar{y}$ to it by tracing *ylist* from b^* while the cell satisfies $f(x,y)=0$. Otherwise we find the minimum $\bar{y}'$ satisfying $\bar{y}'-x > b^*$ and set $\bar{y} := \bar{y}'$, which is done in $O(1)$ time. Since the number of forward scans of *ylist* can not exceed b_u $(< n)$ and the number of backward scans can not exceed n, the algorithm is executed in $O(n)$ time.

6.5 Random Inputs

In this section we consider the case in which two permutations are generated uniformly at random (i.e., every permutation appears with probability $1/n!$), and show the following two properties.

i) Expected number of common intervals is $O(1)$.

ii) Expected running time of algorithm LHP is $O(n)$.

For convenience, only the common intervals of length 2 to $n-2$ are considered in this section. This assumption does not change the above results as discussed in Subsection 6.3.1.

6.5.1 Expected Number of Common Intervals

We define two types of random variables as follows. A variable X_{kx} $(x=1,\ldots,n-k+1$, $k=2,\ldots,n-2)$ takes value 1 if $f(x,x+k-1)=0$, and 0 otherwise. We also define $X_k = \sum_{x=1}^{n-k+1} X_{kx}$ and $X=\sum_{k=2}^{n-2} X_k$. These variables represent the number of common intervals of length k and the number of common intervals of lengths from 2 to $n-2$, respectively.

Theorem 5.2 *For $n \ge 5$, $E(X)=2+O(n^{-1})$. To be more precise, $E(X_2)=2-\frac{2}{n}$, and $E(\sum_{k=3}^{n-2} X_k)=O(n^{-1})$.*

Proof. For fixed x_A and x_B,

$$Pr(\{\sigma_A(i) \mid i \in [x_A, x_A+k-1]\} = \{\sigma_B(i) \mid i \in [x_B, x_B+k-1]\}) = \frac{(n-k)!k!}{n!}. \quad (6.5.16)$$

Since possible values of x_B is from 1 to $n-k+1$, we have

$$E(X_{kx_A}) = \frac{(n-k)!k!}{n!} \times (n-k+1). \quad (6.5.17)$$

By the linearity of expectation, this implies

$$E(X_k) = \sum_{x=1}^{n-k+1} E(X_{kx}), \quad (6.5.18)$$

$$E(X) = \sum_{k=2}^{n-2} E(X_k). \quad (6.5.19)$$

To analyze the behavior of $E(X_k)$, we consider the solution of

$$E(X_k)/E(X_{k-1}) < 1. \quad (6.5.20)$$

From

$$\frac{E(X_k)}{E(X_{k-1})} = \frac{k(n-k+1)}{(n-k+2)^2} < 1, \quad (6.5.21)$$

we obtain

$$2k^2-(3n+5)k+(n^2+4n+4) > 0, \quad (6.5.22)$$

and get the solution $k < \alpha_-(n), \alpha_+(n) < k$ for (6.5.20), where $n \ge 4$ and

$$\alpha_-(n) = \frac{3n+5-\sqrt{n^2-2n-7}}{4}, \quad (6.5.23)$$

$$\alpha_+(n) = \frac{3n+5+\sqrt{n^2-2n-7}}{4}. \quad (6.5.24)$$

It is easy to check that $0 < \alpha_-(n) \le n$ holds. By the fact

$$(n-3)^2 \le n^2-2n-7 \quad (6.5.25)$$

for $n \ge 4$, we have $\alpha_+(n) > n$. Therefore, $E(X_k)$ is monotonically nonincreasing in k when $2 \le k \le \alpha_-(n)$ holds, and is monotonically nondecreasing in k when $\alpha_-(n) \le k \le n$ holds. By using $E(X_4) < \frac{24}{n^2}$ and $E(X_{n-2}) \le \frac{24}{n^2}$ $(n \ge 4)$, we have

$$E(X_k) \le \frac{24}{n^2}, \quad (k=4,5,\ldots,n-2, n \ge 4), \quad (6.5.26)$$

and from (6.5.18),

$$E(X_2) = 2 - \frac{2}{n}, \tag{6.5.27}$$

$$E(X_3) = \frac{6(n-2)}{n(n-1)}. \tag{6.5.28}$$

Hence, we can conclude for $n \geq 5$ that

$$E(\sum_{k=3}^{n-2} X_k) \leq E(X_3) + (n-5) \cdot \frac{24}{n^2} \tag{6.5.29}$$

$$= O(n^{-1}), \tag{6.5.30}$$

$$E(X) = E(X_2) + E(\sum_{k=3}^{n-2} X_k) \tag{6.5.31}$$

$$= 2 + O(n^{-1}). \square \tag{6.5.32}$$

By estimating the variance of X_2 and using Chebyshev bound and Markov inequality, the following theorem is also shown [136].

Theorem 5.3 *If $n \geq 5$, $Pr(X \geq \sqrt{2}t + 3) \leq \frac{1}{t^2} + O(n^{-1})$ holds for arbitrary $t > 0$.*

6.5.2 Expected Running Time of the Algorithm LHP

For each x ($x = 1, \ldots, n-1$), let T_x be the random variable representing the number of iterations in the inner loop of LHP for x. We also define $T = \sum_{x=1}^{n-1} T_x$, which represents the total number of inner loop iterations.

Theorem 5.4 *For $n \geq 4$, $E(T) \leq 3n$ holds.*

Theorem 5.4 holds even if we do not incorporate the length condition (6.3.5) into LHP.

Before proving this theorem, we consider the following problem. Suppose that we have k white balls and $m-k$ black balls ($0 \leq k \leq m-1$, $m \geq 1$) in an urn. The probability of taking out a ball is the same for all balls. Take out one ball. If it is white, we do not replace the ball into the urn and continue the same trial; otherwise (i.e., once a black ball is taken) we terminate the trial. Let $E_{\text{urn}}(m,k)$ denotes the expected number of trials until a black ball is taken. Then

$$E_{\text{urn}}(m,k) = \frac{m+1}{m-k+1} \tag{6.5.33}$$

holds (see Appendix B). We define $E_{\text{urn}}(m,m) = m$ for convenience. Now let $E^*_{\text{urn}}(m,k,j)$ denotes the expected number of trials until a black ball is taken or the number of trials becomes j. Then

$$E^*_{\text{urn}}(m,k,j) \leq E_{\text{urn}}(m,k) \tag{6.5.34}$$

holds for $j \geq 1$, $0 \leq k \leq m$ and $m \geq 1$ (see also Appendix B). These facts are used in the proof.

Proof. By linearity of expectation, we have

$$E(T) = \sum_{x=1}^{n-1} E(T_x). \tag{6.5.35}$$

For a fixed x, let $r(x)$ be $\min\{n-x, n-3\}$, which is the maximum number of inner loop iterations for x. Since the two permutations are generated uniformly at random, $HP' = \{i,j\}$ holds with probability $\binom{n}{2}^{-1}$ for any i and j ($i,j \in [1,n]$, $i < j$). For such i and j, probability that $1 \leq \pi_{AB}(x) \leq i-1$ holds is $\frac{i-1}{n-2}$, and in this case, the expected number of inner loop iterations is $E^*_{\text{urn}}(n-3, i-2, r(x))$. Secondly, the probability that $i+1 \leq \pi_{AB}(x) \leq j-1$ holds is $\frac{j-i-1}{n-2}$, and in this case, the expectation is $E^*_{\text{urn}}(n-3, j-i-2, r(x))$. Thirdly, the probability that $j+1 \leq \pi_{AB}(x) \leq n$ holds is $\frac{n-j}{n-2}$, and in this case, the expectation is $E^*_{\text{urn}}(n-3, n-j-1, r(x))$. Therefore,

$$\begin{aligned}
E(T) &= \sum_{x=1}^{n-1} \binom{n}{2}^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left\{ \frac{i-1}{n-2} E^*_{\text{urn}}(n-3, i-2, r(x)) \right. \\
&\quad \left. + \frac{j-i-1}{n-2} E^*_{\text{urn}}(n-3, j-i-2, r(x)) + \frac{n-j}{n-2} E^*_{\text{urn}}(n-3, n-j-1, r(x)) \right\} \\
&\leq \frac{n-1}{\binom{n}{2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left\{ \frac{i-1}{n-2} E_{\text{urn}}(n-3, i-2) \right. \\
&\quad \left. + \frac{j-i-1}{n-2} E_{\text{urn}}(n-3, j-i-2) + \frac{n-j}{n-2} E_{\text{urn}}(n-3, n-j-1) \right\} \\
&= \frac{n-1}{\binom{n}{2}} \cdot 3 \cdot \sum_{i=1}^{n-1} (n-i) \cdot \frac{i-1}{n-2} E_{\text{urn}}(n-3, i-2) \\
&= \frac{6}{n} \left\{ \sum_{i=1}^{n-2} (n-i) \cdot \frac{i-1}{n-2} \cdot \frac{n-2}{n-i} + (n-3) \right\} \\
&= 3n - 9 \leq 3n. \square
\end{aligned}$$

6.6 Computational Results

In this section, we compare algorithms BSC, LHP, MNG and RC by applying them to six types of problem instances of sizes up to $n = 10^6$.

6.6.1 Generation of Problem Instances

The following six types of problem instances are examined.

RAND: Two permutations σ_A and σ_B are randomly generated (i.e., any permutation is chosen with probability $1/n!$).

SWAP: Initially two permutations σ_A and σ_B are set as $\sigma_A(i) = \sigma_B(i) = i$ for $i = 1, \dots, n$. Then we repeat s times a swap of two elements $\sigma_B(i)$ and $\sigma_B(j)$ for two integers i and j $(i \neq j)$ randomly chosen from $[1, n]$. We set $s = n$ in the experiment.

NBRAND: The permutation σ_A is set as $\sigma_A(i) = i$ for $i = 1, \dots, n$. For an integer k, let p and q be the integers satisfying $n = kp + q$ and $0 \leq q < k$. For each i $(i = 0, 1, \dots, k)$, a permutation $\sigma_i : V_i \to V_i$ is randomly generated, where $V_i = \{ip+1, ip+2, \dots, \min\{(i+1)p, n\}\}$, and σ_B is set as $\sigma_B = \sigma_0\sigma_1 \cdots \sigma_k$. We use $k = \lfloor\sqrt{n} + 0.5\rfloor$ in the experiment.

NBSWAP: Initially two permutations σ_A and σ_B are set as $\sigma_A(i) = \sigma_B(i) = i$ for $i = 1, \dots, n$. Then a swap of two elements $\sigma_B(i+j)$ and $\sigma_B(j)$ for an integer i randomly chosen from $[1, k]$ and an integer j randomly chosen from $[1, n-i]$ is repeated s times, where k is a parameter to restrict the swap distance. We set $k = \lfloor\sqrt{n} + 0.5\rfloor$ and $s = n$ in the experiment.

SLIDE: For an integer k, let p and q be the integers satisfying $n = kp + q$ and $0 \leq q < k$. Two permutations are set as $\sigma_A(i) = i$ and

$$\sigma_B(i) = \begin{cases} i - 2k - 1 \pmod{kp} + 1, & i \equiv 0 \pmod{k} \\ i, & \text{otherwise,} \end{cases}$$

for $i = 1, \dots, n$. An example with $n = 20$ and $k = 3$ is exhibited in Figure 6.4. We set $k = 4$ in the experiment.

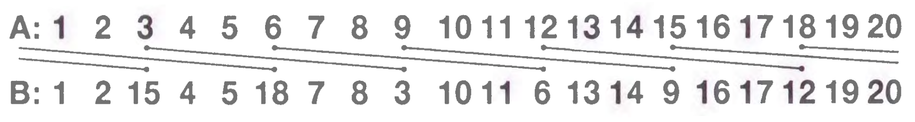

A: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
B: 1 2 15 4 5 18 7 8 3 10 11 6 13 14 9 16 17 12 19 20

Figure 6.4: An example of type SLIDE instance with $n = 20$ and $k = 3$.

NET: Two permutations are set as $\sigma_A(i) = i$ and

$$\sigma_B(i) = \begin{cases} (i+1)/2, & i\text{: odd} \\ \lceil n/2 \rceil + i/2, & i\text{: even,} \end{cases}$$

for $i = 1, 2, \dots, n$. An example with $n = 10$ is shown in Figure 6.5.

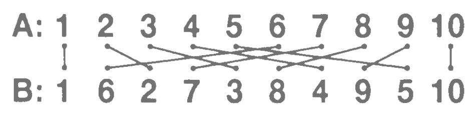


Figure 6.5: An example of type NET instance with $n = 10$.

For type RAND instances, the expected number of common intervals is $2 + O(n^{-1})$ as shown in Section 6.5. By the similar discussion, we can show that the expected number of common intervals for type NBRAND instances is at most $k^2/2 + o(k^2)$ if $k = o(n)$. Recall that we choose $k = O(\sqrt{n})$ in the experiment, and hence, the expected number of outputs is $O(n)$.

For type SWAP and NBSWAP instances, it is observed that the number of common intervals is $O(n)$ as shown in Table 6.1, where each entry is the average of five instances examined in the next subsection.

Table 6.1: Average number of common intervals divided by n for type SWAP and NBSWAP instances.

	K/n	
n	SWAP	NBSWAP
1000	0.022	0.084
10000	0.021	0.050
100000	0.021	0.032
1000000	0.021	0.026

For type SLIDE instances, the number of common intervals is at most

$$p\binom{k-1}{2} + \binom{q}{2} + k(q+1) \leq \frac{1}{2}kn + \frac{3}{2}k^2.$$

Recall that we choose $k = 4$, hence, the number of outputs is $O(n)$. For type NET instances, the number of common intervals is at most one.

6.6.2 Computational Results

All the tested algorithms were coded in C language and run on a workstation Sun SPARC classic. A simple multiplicative congruential method was used to generate random sequences. For each type of problem (except for type SLIDE and NET problems), we generate five instances for each $n = 10^3 \sim 10^6$, and exhibit the average computational time (etc.) of each tested algorithm. Although type SLIDE and NET problems include no randomness, we exhibit the average data of three runs for each tested algorithms, since the CPU time returned by the computer includes errors.

Table 6.2 shows the average number of inner loop iterations of BSC, LHP and MNG divided by n, where $n = 10^4$ is used. (This implies the average number of iterations for an inner loop.) The mark '$*$' is put if this value does not increase more than 5% when $n = 10^6$.

and for others, we mark '△' if the instances with $n = 10^6$ was solved in one minute. Table 6.3 shows the average of the total number of scans on *ulist*, *llist* and *ylist* of the algorithm RC divided by n, where $n = 10^6$ is used. Figures 6.6 ~ 6.11 show the average computational time (in μ secs.) divided by n. (Note that the data are identical to the average computational time in seconds when $n = 10^6$.)

Table 6.2: Average number of inner loop iterations of BSC, LHP and MNG divided by n ($n = 10^4$).

	RAND	SWAP	NBRAND	NBSWAP	SLIDE	NET
BSC	4999.50	4999.50	4999.50	4999.50	4999.50	4999.50
LHP	*1.99	*2.33	99.62	*11.13	2498.50	1876.00
MNG	*3.40	*3.66	53.50	△4.39	*6.25	△8.68

Table 6.3: Average of the total number of scans on *ulist*, *llist* and *ylist* of RC divided by n ($n = 10^6$).

	RAND	SWAP	NBRAND	NBSWAP	SLIDE	NET
RC	27.45	27.44	28.94	27.60	29.75	28.00

From these, we can observe the following:

- In Table 6.2, the marks '$*$' and '△' imply the effectiveness of the speed up techniques proposed in Section 6.3. Especially for those with '$*$' marks, it may be concluded that the problem instances were solved in $O(n)$ time on the average. For each of those with '△' marks, the value increases about 13% (resp., 38%) for NBSWAP (resp., NET) when $n = 10^6$. For NBSWAP, this is because the variance of the data of MNG is rather large. The same tendency was observed for LHP. Indeed, the value decreases about 23% for LHP with NBSWAP when $n = 10^6$. It is known that MNG needs $O(n \log n)$ time for type NET instances, as evidenced by the increase of about 38%.

- The performances of BSC and RC are hardly affected by the type of instances: BSC always requires $O(n^2)$ time, while RC always runs in $O(n)$ time (recall that $K = O(n)$ for all tested problem instances). Note that the values in Table 6.3 are almost the same for other tested sizes.

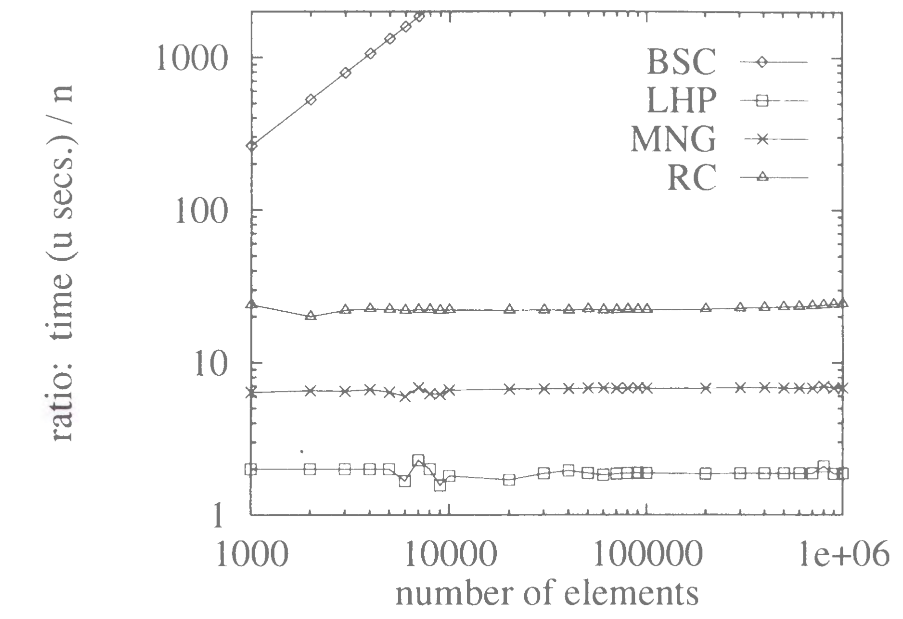


Figure 6.6: Computational time against n (type RAND).

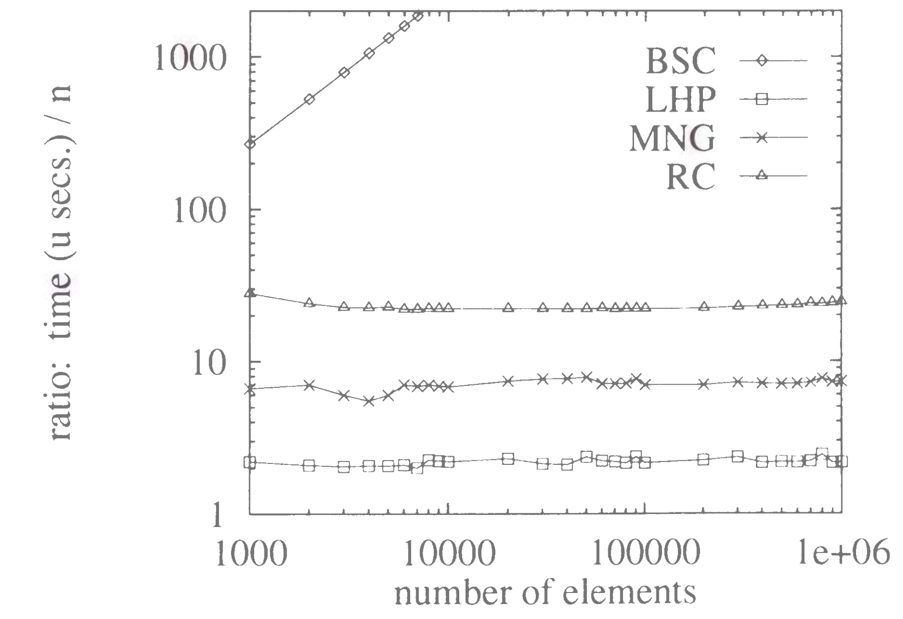


Figure 6.7: Computational time against n (type SWAP).

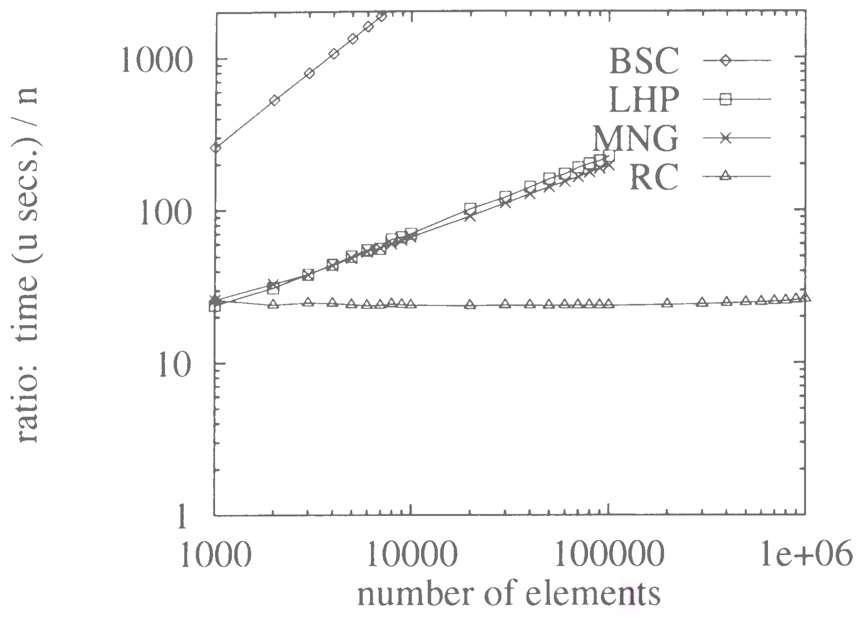


Figure 6.8: Computational time against n (type NBRAND).

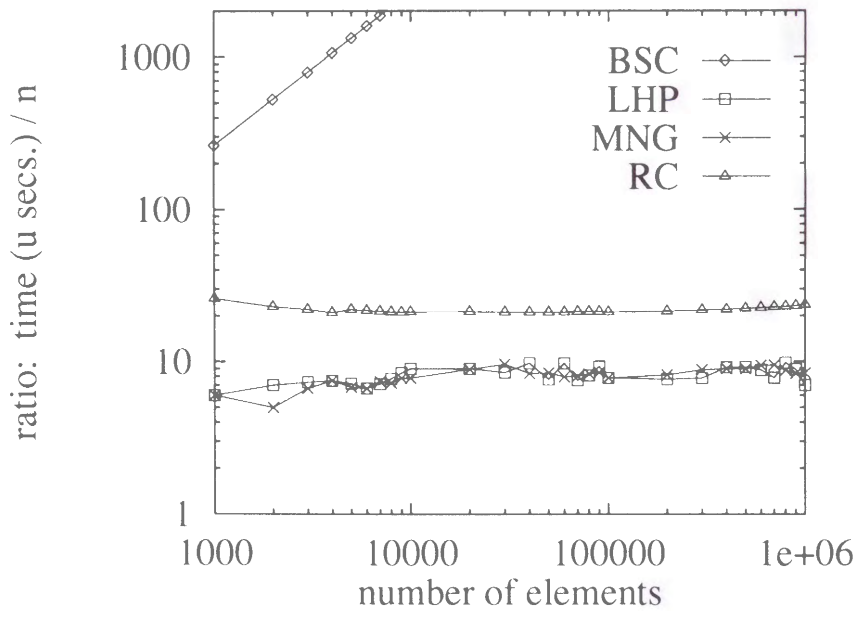


Figure 6.9: Computational time against n (type NBSWAP).

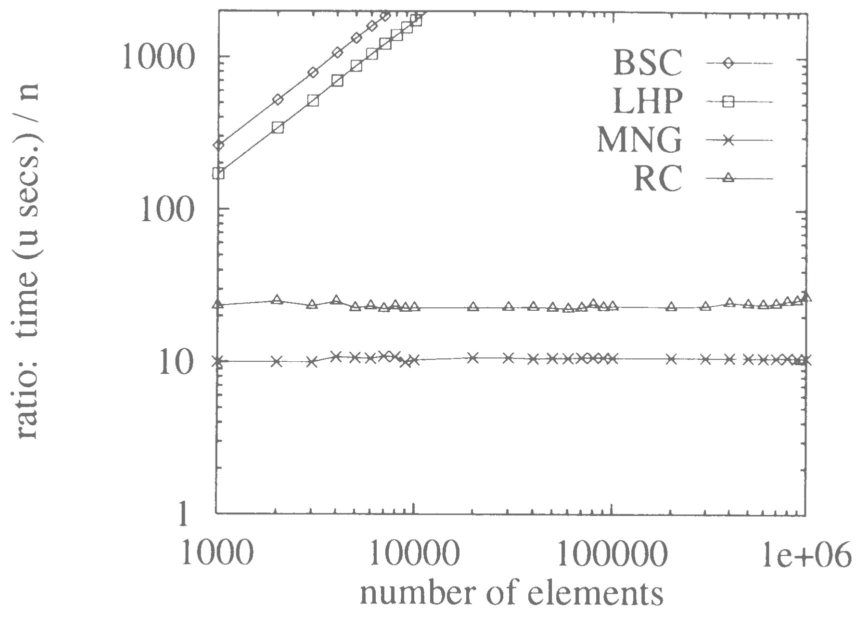


Figure 6.10: Computational time against n (type SLIDE).

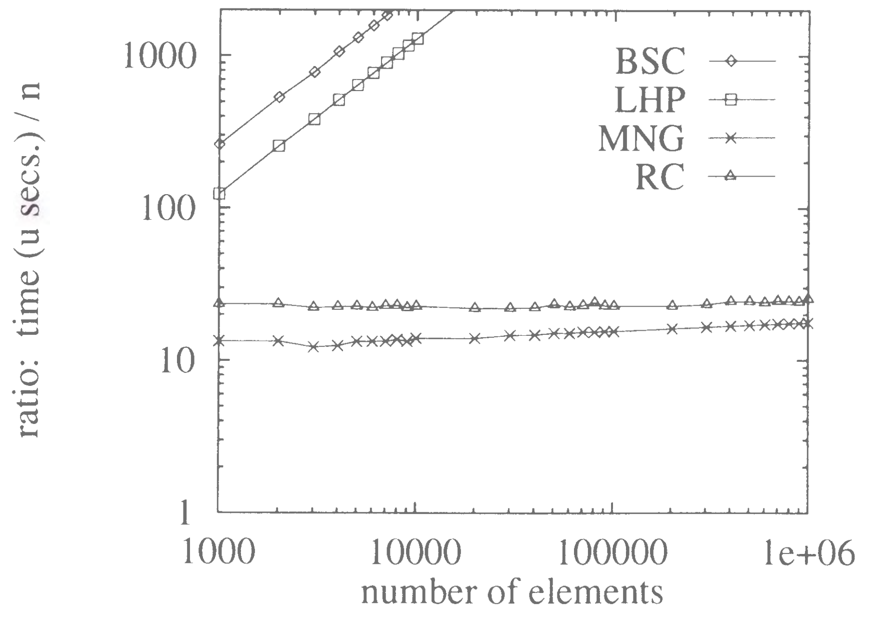


Figure 6.11: Computational time against n (type NET).

- The algorithm LHP is quite effective for type RAND and SWAP instances. It is also effective for type NBSWAP instances, though about three times slower than the cases of RAND and SWAP instances. On the contrary, we can show that $O(n^{3/2})$ time is needed for type NBRAND instances and $O(n^2)$ time is needed for type SLIDE and NET instances.

- The algorithm MNG is quite effective for almost all types of problems except for NBRAND, for which we can show that it requires $O(n^{3/2})$ time. It is noted, however, that the running time of MNG is about three times larger than that of LHP for RAND and SWAP instances, and we can show that MNG requires $O(n \log n)$ time for type NET instances. It is also noted that problem types SLIDE and NET are quite artificial, and these results do not necessarily imply that MNG is more robust than LHP.

6.7 Common Subtrees

In this section, we consider an application of the algorithms for the common interval enumeration problem proposed in the previous sections to the following problem: given two trees with labels on their leaves, enumerate all *common subtrees*, i.e., pairs of subtrees having the same set of leaf labels. By using algorithm RC, we can derive a fast randomized algorithm with $O(n \log^2 n)$ expected running time if we are given two binary trees of depth $\log_2 n$, where n is the number of leaves. The expected running time becomes $O(n)$ if the same two binary trees of depth $\log_2 n$ are given as the input. The latter special case is a trivial instance; however, this case is intuitively considered to be tough for this algorithm, and hence, it is expected that the proposed algorithm runs in $O(n)$ expected time for most of the practical instances, although the worst case running time is $O(n^2)$.

The problem is formally defined as follows. Two rooted trees Υ_A and Υ_B are given as the input, each of which has n leaves labeled with $1, 2, \ldots, n$. A subtree $\Upsilon_A(u)$ is defined to be the subgraph of Υ_A induced by u and all descendants of u. Let $L_A(u)$ be the set of labels of the leaves in $\Upsilon_A(u)$. $\Upsilon_B(u)$ and $L_B(u)$ are similarly defined. We call a pair of subtrees $(\Upsilon_A(u), \Upsilon_B(v))$ a *common subtree* if it satisfies

$$L_A(u) = L_B(v),$$

where u and v are neither a root nor a leaf. We assume that every inner vertex of Υ_A or Υ_B has at least two children, so that the number of inner vertices is $O(n)$.

Genetic algorithms based on common subtrees are proposed for VLSI design. Common subtree also has an application in evolutionary trees for species sets, which are used in biology. There are many proposals for constructing evolutionary trees, which are then compared to

form consensus. The number of common subtrees is one of the basic measures for consensus [57, 96, 107], among others [29, 40].

The proposed algorithm is based on the following observation. Let σ_{Υ_A} be the permutation of leaf labels of Υ_A defined by the order where they are scanned by depth-first search, in which the left to right order of choosing the children of each inner vertex is determined arbitrarily. Let $l_A(u)$ (resp., $r_A(u)$) be the label of the left (resp., right) most leaf of $\Upsilon_A(u)$. σ_{Υ_B}, $l_B(u)$ and $r_B(u)$ are similarly defined. Then $(\Upsilon_A(u), \Upsilon_B(v))$ is a common subtree if and only if $([\sigma^{-1}_{\Upsilon_A}(l_A(u)), \sigma^{-1}_{\Upsilon_A}(r_A(u))], [\sigma^{-1}_{\Upsilon_B}(l_B(v)), \sigma^{-1}_{\Upsilon_B}(r_B(v))])$ is a common interval of two permutations σ_{Υ_A} and σ_{Υ_B}. Note that there may be common intervals of σ_{Υ_A} and σ_{Υ_B} that do not correspond to any subtrees.

The basic framework of the algorithm is as follows.

1. Apply depth-first search to Υ_A and Υ_B, choosing randomly the order of the children at each inner vertex. Denote the two permutations of the leaf labels of Υ_A and Υ_B by σ_{Υ_A} and σ_{Υ_B}, respectively.

2. Enumerate all common intervals of σ_{Υ_A} and σ_{Υ_B} one by one, and if the two intervals corresponding to each common interval define subtrees of Υ_A and Υ_B, respectively, then output the corresponding pair of subtrees.

Step 1 can be executed in $O(n)$ time. We can check in Step 2 if an interval of σ_{Υ_A} (resp., σ_{Υ_B}) defines a subtree of Υ_A (resp., Υ_B) in $O(1)$ worst case time by using the data structure called perfect hash [36], which can be constructed in $O(n)$ expected time and in $O(n^2)$ worst case time.

Let $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ be the number of common intervals of two permutations σ_{Υ_A} and σ_{Υ_B}. $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ may be the dominating factor of the running time of our algorithm. Note that $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B}) = \binom{n}{2}$ in the worst case, although the number of common subtrees is $O(n)$. It is also noted that the result about the expected number of common intervals for two random permutations stated in Section 6.5 is not applicable in this case, since the probability space is different. We can show that the expected value of $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ is $O(n \log^2 n)$ if the given two trees are binary and the depth of them is $\log_2 n$. We can also show that the expected value of $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ is $O(n)$ if the same two binary trees of depth $\log_2 n$ are given as the input. The latter special case is a trivial instance as the common subtree enumeration problem. However, in this case, $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B}) = \binom{n}{2}$ if we do not randomize the children order of each inner vertex, which is the largest possible value of $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$. Hence, this is considered to be a tough instance for our algorithm. Therefore, we believe that the expected value of $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ is small (e.g., $O(n)$) for most of the practical instances, although theoretical results are limited to the above special cases.

Actually, it is observed by computational experiments on some types of randomly generated trees with up to $n = 10^6$ that the average value of $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ is $O(n)$ for all the tested instances, in which trees of depth $\Omega(n)$ are included.

If we use algorithm RC to enumerate common intervals in Step 2, the expected running time of the above algorithm is $O(n+K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B}))$, which is $O(n \log^2 n)$ if the given two trees are binary and the depth of them is $\log_2 n$, and $O(n)$ if the same two binary trees of depth $\log_2 n$ are given as the input. The worst case running time is $O(n^2)$, since $K(\sigma_{\Upsilon_A}, \sigma_{\Upsilon_B})$ is $O(n^2)$ in general.

Similar algorithms are applicable to the subtree problems defined on two *unrooted* trees, in which two connected components defined by deleting an edge are considered as subtrees.

6.8 Conclusion

For the common interval enumeration problem, we proposed the following three algorithms: i) a simple $O(n^2)$ time algorithm (LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations, ii) a practically fast $O(n^2)$ time algorithm (MNG) using the reverse Monge property, and iii) an $O(n+K)$ time algorithm (RC). It was observed in the computational experiment that: 1) LHP is very fast for randomly generated problem instances, 2) MNG is rather slower than LHP for random instances; however, there are cases that MNG can run in $o(n^2)$ time while LHP needs $\Omega(n^2)$ time, and 3) the performance of RC is quite robust against the types of problem instances, though it is rather slower than MNG for many of the tested problem instances. It is noted that LHP and MNG are very simple and easy to program (LHP is much simpler than MNG), while RC is rather complicated. On the other hand, it is also noted that there are cases that both LHP and MNG require $\Omega(n^2)$ time as mentioned in the end of Section 6.3. From these, we recommend RC if one wants to solve large instances (e.g., $n \geq 10^5$), and LHP if one wants to solve the instances which seem to include randomness. MNG is recommended if LHP fails to solve efficiently some problem instances one wants to solve.

Appendix A

Here we prove the *reverse Monge property* of $f(\cdot,\cdot)$, that is,

$$f(x',y) + f(x,y') \geq f(x,y) + f(x',y')$$

holds for all x', x, y, y' satisfying $x' < x \leq y < y'$. Subtracting right-hand side from left-hand side, we get

$$u(x',y) + u(x,y') - \{u(x,y) + u(x',y')\} + l(x,y) + l(x',y') - \{l(x',y) + l(x,y')\}.$$

It is sufficient to show that $u(\cdot,\cdot)$ and $l(\cdot,\cdot)$ satisfy

$$\begin{array}{rcll} u(x',y) + u(x,y') & \geq & u(x,y) + u(x',y') & \text{(reverse Monge property)} \\ l(x',y) + l(x,y') & \leq & l(x,y) + l(x',y') & \text{(Monge property).} \end{array}$$

We prove this only for $u(\cdot,\cdot)$, since the latter case is symmetrically proven. Either $u(x',y') = u(x,y')$ or $u(x',y') = u(x',y)$ holds, since

$$\max_{z\in[x',x-1]} \pi_{AB}(z) < u(x,y') \quad\Rightarrow\quad u(x',y') = u(x,y')$$
$$\max_{z\in[x',x-1]} \pi_{AB}(z) \geq u(x,y') \quad\Rightarrow\quad u(x',y') = u(x',y).$$

This fact, combined with $u(x,y') \geq u(x,y)$ and $u(x',y) \geq u(x,y)$, implies that $u(\cdot,\cdot)$ satisfies reverse Monge property, and hence, reverse Monge property of $f(\cdot,\cdot)$ is proven.

Appendix B

Here, we prove that

$$E_{\text{urn}}(m,k) = \frac{m+1}{m-k+1} \tag{6.8.36}$$

for $0 \leq k \leq m-1$ and $m \geq 1$, and

$$E^*_{\text{urn}}(m,k,j) \leq E_{\text{urn}}(m,k) \tag{6.8.37}$$

for $1 \leq j$, $0 \leq k \leq m$ and $m \geq 1$, where $E_{\text{urn}}(\cdot,\cdot)$ and $E^*_{\text{urn}}(\cdot,\cdot,\cdot)$ are defined in Section 6.5. Let us define a random variable Z representing the number of trials until a black ball is taken out. The probability that a black ball is taken out after i trials or more is equal to the probability that white balls are taken in the first $i-1$ trials, so

$$Pr(Z \geq i) = \frac{[k]_{i-1}}{[m]_{i-1}}, \quad i = 1, \ldots, k+1 \tag{6.8.38}$$

holds, where

$$[m]_i = \begin{cases} 1, & i = 0, \\ m(m-1)\cdots(m-i+1), & i > 0. \end{cases} \tag{6.8.39}$$

By using this fact, we can conclude

$$\begin{aligned} E_{\text{urn}}(m,k) &= \sum_{i=1}^{k+1} iPr(Z=i) \\ &= \sum_{i=1}^{k+1} Pr(Z \geq i) \\ &= \sum_{i=0}^{k} \frac{[k]_i}{[m]_i} \\ &= \sum_{i=0}^{k} \frac{\binom{m-i}{k-i}}{\binom{m}{k}} \\ &= \frac{m+1}{m-k+1}. \end{aligned} \tag{6.8.40}$$

See for example [49] for the last sigma calculation. When $k \leq m-1$, if $1 \leq j \leq k$, then

$$\begin{aligned} E^*_{\mathrm{urn}}(m,k,j) &= \sum_{i=1}^{j-1} iPr(Z=i) + jPr(Z \geq j) \\ &= \sum_{i=1}^{j} Pr(Z \geq i) \\ &\leq E_{\mathrm{urn}}(m,k), \end{aligned} \tag{6.8.41}$$

and if $j \geq k+1$, then $E^*_{\mathrm{urn}}(m,k,j) = E_{\mathrm{urn}}(m,k)$. When $k=m$, $E^*_{\mathrm{urn}}(m,m,j) = j \leq m = E_{\mathrm{urn}}(m,k)$. (Recall that we defined $E_{\mathrm{urn}}(m,m) = m$ for convenience.)

Chapter 7

Conclusion

Throughout this thesis, we have considered various metaheuristic algorithms for the combinatorial optimization problems. The contribution of this thesis is summerized as follows.

First, we proposed a framework of approximate algorithms, called *genetic DP*, in which dynamic programming is incorporated into the genetic algorithm. Its effectiveness was evaluated by computational experiments for three problems: the single machine scheduling problem (SMP), the optimal linear arrangement problem (OLAP) and the traveling salesman problem (TSP), all of which are known to be NP-hard. Genetic DP tends to attain better solution quality than traditional multi-start local search (MLS) and genetic local search (GLS) algorithms when sufficiently long time is allowed, though performance of these algorithms depends on problem characteristics. Recently, similar hybrid approach of combining exact methods and metaheuristic methods are tried in [5, 83].

Second, we compared various crossover operators proposed for sequencing problems using a general framework of crossover operators. It was confirmed that the performance of the crossover operators can be evaluated by some simple criteria related to characteristics of the set of children obtainable from the parents. These criteria are expected to give useful guidelines to design good crossover operators for genetic algorithms. The flexibility is one of the attractive features of metaheuristics; however, from the view point of users, the algorithms should be as simple as possible. In this sense, it is important to simplify the framework and analyze the effect of each basic operation to the performance of the algorithm. This second result may be useful from the view point of this research direction.

Next, various metaheuristic algorithms were compared from the view point of robustness and simplicity. As a concrete problem to test, we chose the single machine scheduling problem (SMP) and metaheuristics such as the multi-start local search (MLS), the genetic algorithm (GA), the simulated annealing (SA), the tabu search (TS), and some of their variants were examined. A guideline to design metaheuristic algorithms was proposed in Section 5.3, based on the obtained computational results. These results were limited to a single problem, and it is important to conduct similar comparisons on the basis of various types of problems so that we can understand the general tendencies of the metaheuristic algorithms.

Finally, we considered the common interval enumeration problem, which stems from a basic operation of genetic algorithms for sequencing problems. For this problem, we proposed the following three algorithms: i) a simple $O(n^2)$ time algorithm (LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations, ii) a practically fast $O(n^2)$ time algorithm (MNG) using the reverse Monge property, and iii) an $O(n+K)$ time algorithm (RC). Application of these algorithms to the common subtree enumeration problem was also discussed. Designing efficient implementations for basic operations of metaheuristic algorithms is practically very important; however, not much research has been done in this direction. In this sense, the above results may be useful from the view point of this research direction. As another example, we recently proposed efficient neighborhood implementations for the maximum satisfiability problem [135], in which the worst-case and average-case time complexities are analyzed. Such analyses of basic operations of metaheuristics are considered to be one of the important future research directions.

Recently, the border lines between metaheuristic algorithms become subtle, since many variants and hybrid approaches of more than one metaheuristic algorithm have been proposed and are given different names. Therefore, as a whole, metaheuristic algorithms become quite complicated. However, it is important to understand the roles of basic components of these approaches and provide a guideline to design effective metaheuristic algorithms which can exploit the structures of given problems. Moreover, not much research has been done on the theoretical aspects of metaheuristics. The author hopes that this dissertation will provide some assistance to the community of metaheuristic algorithms.

Bibliography

[1] E.H.L. Aarts and J.H.M. Korst, *Simulated Annealing and Boltzmann Machines*, (John Wiley & Sons, 1989).

[2] 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*, 6 (1994) 118–125.

[3] E.H.L. Aarts and J.K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, (John Wiley & Sons, 1997).

[4] D. Adolphson and T.C. Hu, "Optimal Linear Ordering," *SIAM Journal on Applied Mathematics*, 25 (1973) 403–423.

[5] C.C. Aggarwal, J.B. Orlin and R.P. Tai, "Optimized Crossover for the Independent Set Problem," *Operations Research*, 45 (1997) 226–234.

[6] D. Aldous and U. Vazirani, ""Go With the Winners" Algorithms," *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (1994) 492–501.

[7] I. Althöfer and K.-U. Koschnick, "On the Convergence of "Threshold Accepting"," *Applied Mathematics and Optimization*, 24 (1991) 183–195.

[8] D. Applegate, R. Bixby, V. Chvátal and W. Cook, "Finding Cuts in the TSP (A Preliminary Report)," manuscript taken from `http://achille.research.att.com/netlib/att/math/applegate/TSP/`.

[9] K.R. Baker and G.D. Scudder, "Sequencing with Earliness and Tardiness Penalties: A Review," *Operations Research*, 38 (1990) 22–36.

[10] J. Bhasker and S. Sahni, "Optimal Linear Arrangement of Circuit Components," *Proceedings of the 20th Annual Hawaii International Conference on System Sciences* (1987) 99–111.

[11] J.N. Bhuyan, V.V. Raghavan and V.K. Elayavalli, "Genetic Algorithm for Clustering with an Ordered Representation," *Proceedings of the 4th International Conference on Genetic Algorithms* (1991) 408–415.

[12] K.D. Boese, A.B. Kahng and S. Muddu, "A New Adaptive Multi-Start Technique for Combinatorial Global Optimizations," *Operations Research Letters*, 16 (1994) 101–113.

[13] R.M. Brady, "Optimization Strategies Gleaned from Biological Evolution," *Nature*, 317 (1985) 804–806.

[14] J. Brimberg, P. Hansen, N. Mladenović and É.D. Taillard, "Improvements and Comparison of Heuristics for Solving the Multisource Weber Problem," Technical Report IDSIA-33-97, IDSIA, Corso Elvezia 36, CH-6900 Lugano, Switzerland, 1997 (available as `http://www.idsia.ch/~eric/articles.dir/IDSIA-33-97.ps.Z`).

[15] J. Brimberg and N. Mladenović, "A Variable Neighborhood Algorithm for Solving the

Continuous Location-Allocation Problem," *Studies in Locational Analysis*, 10 (1996) 1–12.

[16] V. Černý, "A Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Applications*, 45 (1985) 41–51.

[17] I. Charon and O. Hudry, "The Noising Method: a New Method for Combinatorial Optimization," *Operations Research Letters*, 14 (1993) 133–137.

[18] B. Codenotti, G. Manzini, L. Margara and G. Resta, "Perturbation: An Efficient Technique for the Solution of Very Large Instances of the Euclidean TSP," *INFORMS Journal on Computing*, 8 (1996) 125–133.

[19] A. Colorni, M. Dorigo and V. Maniezzo, "Distributed Optimization by Ant Colonies," *Proceedings of European Conference on Artificial Life* (1991) 134–142.

[20] H.A.J. Crauwels, C.N. Potts and L.N. Van Wassenhove, "Local Search Heuristics for Single-Machine Scheduling with Batching to Minimize the Number of Late Jobs," *European Journal of Operational Research*, 90 (1996) 200–213.

[21] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains," *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (1985) 162–164.

[22] L. Davis (ed.), *Handbook of Genetic Algorithms*, (Van Nostrand Reinhold, 1991).

[23] T. Dimitriou and R. Impagliazzo, "Towards an Analysis of Local Optimization Algorithms," *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing* (1996) 304–313.

[24] M. Dorigo, *Optimization, Learning, and Natural Algorithms*, PhD Thesis, Politecnico di Milano, 1992.

[25] M. Dorigo, V. Maniezzo and A. Colorni, "The Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on System, Man, and Cybernetics - Part B*, 26-1 (1996) 29–41.

[26] U. Dorndorf and E. Pesch, "Fast Clustering Algorithms," *ORSA Journal on Computing*, 6 (1994) 141–153.

[27] G. Dueck, "New Optimization Heuristics: the Great Deluge Algorithm and the Record-to-Record Travel," Heidelberg Scientific Center Research Report TR89.06.011, IBM, Germany (1989).

[28] G. Dueck and T. Scheuer, "Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing," *Journal of Computational Physics*, 90 (1990) 161–175.

[29] M. Farach and M. Thorup, "Sparse Dynamic Programming for Evolutionary-Tree Comparison," *SIAM Journal on Computing*, 26 (1997) 210–230.

[30] T.A. Feo and J.L. González-Velarde, "The Intermodal Trailer Assignment Problem," *Transportation Science*, 29 (1995) 330–341.

[31] T.A. Feo and M.G.C. Resende, "A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem," *Operations Research Letters*, 8 (1989) 67–71.

[32] T.A. Feo and M.G.C. Resende, "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, 6 (1995) 109–133.

[33] T.A. Feo, M.G.C. Resende and S.H. Smith, "A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set," *Operations Research*, 42 (1994) 860–878.

[34] T.A. Feo, K. Venkatraman and J.F. Bard, "A GRASP for a Difficult Single Machine Scheduling Problem," *Computers and Operations Research*, 18 (1991) 635–643.

[35] B.R. Fox and M.B. McMahon, "Genetic Operators for Sequencing Problems," in: *Foundations of Genetic Algorithms (FOGA)*, (Morgan Kaufmann, 1991) p. 284–300.

[36] M.L. Fredman, J. Komlós and E. Szemerédi, Storing a Sparse Table with $O(1)$ Worst Case Access Time, *Journal of the Association for Computing Machinery*, 31 (1984) 538–544.

[37] B. Freisleben and P. Merz, "A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems," in: *Proceedings of IEEE International Conference on Evolutionary Computation* (1996) 616–621.

[38] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, (Freeman, 1979).

[39] M.R. Garey, D.S. Johnson and L. Stockmeyer, "Some Simplified NP-Complete Graph Problems," *Theoretical Computer Science*, 1 (1976) 237–267.

[40] L. Gąsieniec, J. Jansson, A. Lingas and A. Östlin, "On the Complexity of Computing Evolutionary Trees," *Proceedings of the 3rd Annual International Conference on Computing and Combinatorics* (1997) 134–145.

[41] F. Glover, "Tabu Search — Part I," *ORSA Journal on Computing*, 1 (1989) 190–206; Part II, ditto, 2 (1990) 4–32.

[42] F. Glover, "Tabu Search and Adaptive Memory Programming — Advances, Applications and Challenges," in: *Interfaces in Computer Science and Operations Research*, (Kluwer Academic Publishers, 1996).

[43] F. Glover and M. Laguna, "Tabu Search," a chapter in: *Modern Heuristic Techniques for Combinatorial Problems*, (Blackwell Scientific Publications, 1993; re-issued by McGraw-Hill, 1995).

[44] F. Glover and M. Laguna, *Tabu Search*, (Kluwer Academic Publishers, 1997).

[45] F. Glover, E. Taillard and D. de Werra, "A User's Guide to Tabu Search," *Annals of Operations Research*, 41 (1993) 3–28.

[46] D.E. Goldberg and R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem," *Proceedings of the 1st International Conference on Genetic Algorithms* (1985) 154–159.

[47] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, (Addison-Wesley, 1989).

[48] M. Gorges-Schleuter, "ASPARAGOS An Asynchronous Parallel Genetic Optimization Strategy," *Proceedings of the 3rd International Conference on Genetic Algorithms* (1989) 422–427.

[49] R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics — A Foundation for Computer Science*, (Addison-Wesley, 1989).

[50] J. Grefenstette, R. Gopal, B. Rosmaita and D. Van Gucht, "Genetic Algorithms for the Traveling Salesman Problem," *Proceedings of the 1st International Conference on Genetic Algorithms* (1985) 160–168.

[51] M. Grötschel and O. Holland, "Solution of Large-Scale Symmetric Traveling Salesman Problems," *Mathematical Programming*, 51 (1991) 141–202.

[52] J. Gu, "Optimization by Multispace Search," Technical Report UCECE-TR-90-001, Department of Electrical and Computer Engineering, University of Calgary, 1990.

[53] J. Gu, "Efficient Local Search for Very Large-Scale Satisfiability Problems," *SIGART Bulletin*, 3 (1992) 8–12.

[54] J. Gu and X. Huang, "Efficient Local Search with Search Space Smoothing: A Case Study of the Traveling Salesman Problem (TSP)," *IEEE Transactions on Systems, Man, and Cybernetics*, 24 (1994) 728–735.

[55] J. Hart and A. Shogan, "Semi-Greedy Heuristics: An Empirical Study," *Operations Research Letters*, 6 (1987) 107–114.

[56] M. Held and R.M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *SIAM Journal on Applied Mathematics*, 10 (1962) 196–210.

[57] M.D. Hendy, C.H.C. Little and D. Penny, "Comparing Trees with Pendant Vertices Labeled," *SIAM Journal on Applied Mathematics*, 44 (1984) 1054–1065.

[58] J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, (The University of Michigan Press, 1975, and MIT Press, 1992).

[59] T. Ibaraki, "Combination with Other Optimization Methods," a chapter in: *Handbook of Evolutionary Computation*, (IOP Publishing Ltd and Oxford University Press, 1996).

[60] T. Ibaraki and Y. Nakamura, "A Dynamic Programming Method for Single Machine Scheduling," *European Journal of Operational Research*, 76 (1994) 72–82.

[61] P. Jog, J.Y. Suh and D. Van Gucht, "The Effects of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem," *Proceedings of the 3rd International Conference on Genetic Algorithms* (1989) 110–115.

[62] P. Jog, J.Y. Suh and D. Van Gucht, "Parallel Genetic Algorithms Applied to the Traveling Salesman Problem," *SIAM Journal on Optimization*, 1 (1991) 515–529.

[63] D.S. Johnson, "Local Optimization and the Traveling Salesman Problem," *Proceedings of the 17th Colloquium on Automata, Languages and Programming* (1990) 446–461.

[64] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, "Optimization by Simu-

lated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, 37 (1989) 865–892; "Part II, Graph Coloring and Number Partitioning," ditto, 39 (1991) 378–406.

[65] D.S. Johnson, C.H. Papadimitriou and M. Yannakakis, "How Easy Is Local Search?," *Journal of Computer and System Sciences*, 37 (1988) 79–100.

[66] S. Kang, "Linear Ordering and Application to Placement," *Proceedings of the 20th Design Automation Conference* (1983) 457–464.

[67] J.P. Kelly, M. Laguna and F. Glover, "A Study of Diversification Strategies for the Quadratic Assignment Problem," *Computers and Operations Research*, 21 (1994) 885 893.

[68] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, 49 (1970) 291–307.

[69] S. Kirkpatrick, C.D. Gelatt, Jr. and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, 220 (1983) 671–680.

[70] S. Kobayashi, I. Ono and M. Yamamura, "An Efficient Genetic Algorithm for Job Shop Scheduling Problems," *Proceedings of the 6th International Conference on Genetic Algorithms* (1995) 506–511.

[71] A. Kolen and E. Pesch, "Genetic Local Search in Combinatorial Optimization," *Discrete Applied Mathematics*, 48 (1994) 273–284.

[72] M.W. Krentel, "On Finding and Verifying Locally Optimal Solutions," *SIAM Journal on Computing*, 19 (1990) 742–749.

[73] M. Laguna, T.A. Feo and H.C. Elrod, "A Greedy Randomized Adaptive Search Procedure for the Two-Partition Problem," *Operations Research*, 42 (1994) 677–687.

[74] M. Laguna, J.P. Kelly, J.L. González-Velarde and F. Glover, "Tabu Search for the Multilevel Generalized Assignment Problem," *European Journal of Operational Research*, 82 (1995) 176–189.

[75] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, (John Wiley & Sons, 1985).

[76] Y. Li, P.M. Pardalos and M.G.C. Resende, "A Greedy Randomized Adaptive Search Procedure for the Quadratic Assignment Problem," *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 16 (1994) 237–261.

[77] C.K.Y. Lin, K.B. Haley and C. Sparks, "A Comparative Study of both Standard and Adaptive Versions of Threshold Accepting and Simulated Annealing Algorithms in Three Scheduling Problems," *European Journal of Operational Research*, 83 (1995) 330-346.

[78] S. Lin and B.W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research*, 21 (1973) 498–516.

[79] M. Lundy and A. Mees, "Convergence of an Annealing Algorithm", *Mathematical Pro-*

gramming, 34 (1986) 111–124.

[80] V. Maniezzo, M. Dorigo and A. Colorni, "The Ant System Applied to the Quadratic Assignment Problem," Technical Report IRIDIA/94-28, Universit Libre de Bruxelles, Belgium, 1994.

[81] O.C. Martin and S.W. Otto, "Combining Simulated Annealing with Local Search Heuristic," *Annals of Operations Research*, 63 (1996) 57–75.

[82] O.C. Martin, S.W. Otto and E.W. Felten, "Large-Step Markov Chains for the TSP Incorporating Local Search Heuristic," *Operations Research Letters*, 11 (1992) 219–224.

[83] T. Mautor and P. Michelon, "MIMAUSA: A New Hybrid Method Combining Exact Solution and Local Search," *Proceedings of the 2nd International Conference on Metaheuristics* (1997) 15 16.

[84] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, (Springer-Verlag, 1992).

[85] D.L. Miller and J.F. Pekny, "Exact Solution of Large Asymmetric Traveling Salesman Problem," *Science*, 251 (1991) 754–761.

[86] N. Mladenović and P. Hansen, "Variable Neighborhood Search," *Les Cahiers du GERAD*, G-96-49, 1996 (to appear in *Computers and Operations Research*).

[87] H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," *Proceedings of the 3rd International Conference on Genetic Algorithms* (1989) 416–421.

[88] H. Mühlenbein, "Parallel Genetic Algorithms in Combinatorial Optimization," in: *Computer Science and Operations Research*, (Pergamon Press, 1992) p. 441–453.

[89] H. Mühlenbein, M. Gorges-Schleuter and O. Krämer, "Evolution Algorithms in Combinatorial Optimization," *Parallel Computing*, 7 (1988) 65–85.

[90] I.M. Oliver, D.J. Smith and J.R.C. Holland, "A Study of Permutation Crossover Operators on the Traveling Salesman Problem," *Proceedings of the 2nd International Conference on Genetic Algorithms* (1987) 224–230.

[91] I.H. Osman and J.P. Kelly, "Meta-Heuristics: An Overview," in: *Meta-Heuristics: Theory and Applications*, (Kluwer Academic Publishers, 1996) p. 1–21.

[92] I.H. Osman and J.P. Kelly (eds.), *Meta-Heuristics: Theory and Applications*, (Kluwer Academic Publishers, 1996).

[93] M. Padberg and G. Rinaldi, "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems," *SIAM Review*, 33 (1991) 60–100.

[94] C.H. Papadimitriou, "The Complexity of the Lin-Kernighan Heuristic for the Traveling Salesman Problem," *SIAM Journal on Computing*, 21 (1992) 450–465.

[95] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, (Prentice-Hall, 1982).

[96] D. Penny and M.D. Hendy, "The Use of Tree Comparison Metrics," *Systematic Zoology*,

34 (1985) 75–82.

[97] E. Pesch and F. Glover, "TSP Ejection Chains," *Discrete Applied Mathematics*, 76 (1997) 165-181.

[98] M. Pirlot, "General Local Search Heuristics in Combinatorial Optimization: A Tutorial," *Belgian Journal of Operations Research, Statistics and Computer Science*, 32 (1992) 7-67.

[99] M. Pirlot, "General Local Search Methods," *European Journal of Operational Research*, 92 (1996) 493–511.

[100] C.N. Potts and L.N. Van Wassenhove, "A Decomposition Algorithm for the Single Machine Total Tardiness Problem," *Operations Research Letters*, 1 (1982) 177-181.

[101] C.N. Potts and L.N. Van Wassenhove, "A Branch and Bound Algorithm for the Total Weighted Tardiness Problem," *Operations Research*, 33 (1985) 363–377.

[102] P. Prosser and P. Shaw, "Guided Local Search for the Vehicle Routing Problem," *Proceedings of the 2nd International Conference on Metaheuristics* (1997) 89–90.

[103] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves and G.D. Smith (eds.), *Modern Heuristic Search Methods*, (John Wiley & Sons, 1996).

[104] C.R. Reeves (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, (Blackwell Scientific Publications, 1993; re-issued by McGraw-Hill, 1995).

[105] C.R. Reeves, "Genetic Algorithms for the Operations Researcher," *INFORMS Journal on Computing*, 9 (1997) 231–250.

[106] M.G.C. Resende and T.A. Feo, "A GRASP for Satisfiability," *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, (to appear).

[107] F.J. Rohlf, "Consensus Indices for Comparing Classifications," *Mathematical Biosciences*, 59 (1982) 131–144.

[108] D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis II, "An Analysis of Several Heuristics for the Traveling Salesman Problem," *SIAM Journal on Computing*, 6 (1977) 563 581.

[109] R.A. Russell, "Hybrid Heuristics for the Vehicle Routing Problem with Time Windows," *Transportation Science*, (to appear).

[110] G.H. Sasaki and B. Hajek, "The Time Complexity of Maximum Matching by Simulated Annealing," *Journal of the Association for Computing Machinery*, 35 (1988) 387–403.

[111] A.A. Schäffer and M. Yannakakis, "Simple Local Search Problems That Are Hard to Solve," *SIAM Journal on Computing*, 20 (1991) 56–87.

[112] M. Schrage and K.R. Baker, "Dynamic Programming Solution of Sequencing Problems with Precedence Constraints," *Operations Research*, 26 (1978) 444–449.

[113] D.M. Schuler and E.G. Ulrich, "Clustering and Linear Placement," *Proceedings of the 9th Design Automation Conference*, (1972) 50–56.

[114] B. Selman and H.A. Kautz, "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems," *Proceedings of of the 13th International Joint Con-*

ference on Artificial Intelligence (1993) 290 295.

[115] B. Selman, H.A. Kautz and B. Cohen, "Noise Strategies for Improving Local Search," *Proceedings of the 12th National Conference on Artificial Intelligence* (1994) 337–343.

[116] Y. Shiloach, "A Minimum Linear Arrangement Algorithm for Undirected Trees," *SIAM Journal on Computing*, 8 (1979) 15 32.

[117] M. Sinclair, "Comparison of the Performance of Modern Heuristics for Combinatorial Optimization on Real Data," *Computers and Operations Research*, 20 (1993) 687 695.

[118] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley and C. Whitley, "A Comparison of Genetic Sequencing Operators," *Proceedings of the 4th International Conference on Genetic Algorithms* (1991) 69 76.

[119] G. Steiner, "Single Machine Scheduling with Precedence Constraints of Dimension 2," *Mathematics of Operations Research*, 9 (1984) 248 259.

[120] G. Steiner, "On Estimating the Number of Order Ideals in Partial Orders, with Some Applications," *Journal of Statistical Planning and Inference* 34 (1993) 281–290.

[121] R.H. Storer, S.D. Wu and R. Vaccari, "New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling," *Management Science*, 38 (1992) 1495- 1509.

[122] T. Stützle and H. Hoos, "$\mathcal{MAX}$-$\mathcal{MIN}$ Ant System and Local Search for the Traveling Salesman Problem," *Proceedings of 1997 IEEE International Conference on Evolutionary Computation* (1997).

[123] J.Y. Suh and D. Van Gucht, "Incorporating Heuristic Information into Genetic Search," *Proceedings 2nd International Conference on Genetic Algorithms* (1987) 100- 107.

[124] E.D. Taillard and L.M. Gambardella, "An Ant Approach for Structured Quadratic Assignment Problems," *Proceedings of the 2nd International Conference on Metaheuristics* (1997) 217 222.

[125] E. Tsang and C. Voudouris, "Fast Local Search and Guided Local Search and Their Application to British Telecom's Workforce Scheduling Problem," *Operations Research Letters*, 20 (1997) 119-127.

[126] N.L.J. Ulder, E.H.L. Aarts, H.-J. Bandelt, P.J.M. Van Laarhouven and E. Pesch, "Genetic Local Search Algorithms for the Traveling Salesman Problem," *Proceedings of the 1st International Workshop on Parallel Problem Solving from Nature* (1990) 109 116.

[127] T. Uno and M. Yagiura, "Fast Algorithms to Enumerate All Common Intervals of Two Permutations," Technical Report #96015, Department of Applied Mathematics and Physics, Graduate School of Engineering, Kyoto University, 1996.

[128] C. Voudouris, and E. Tsang, "Guided Local Search," Technical Report CSM-247, Department of Computer Science, University of Essex, 1995

[129] M. Yagiura, "Genetic Algorithms for Solving Some Combinatorial Optimization Problems," Master thesis, Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, 1993.

[130] M. Yagiura and T. Ibaraki, "A Genetic Algorithm for Solving the Single Machine Scheduling Problem (in Japanese)," *IEICE Technical Report*, COMP91 17 (1991) 43 52.

[131] M. Yagiura and T. Ibaraki, "Fast Algorithms to Enumerate All Common Intervals of Two Permutations (in Japanese)," *Technical Report of IEICE* (COMP94-83), 94 (1995) 65 74.

[132] M. Yagiura and T. Ibaraki, "Genetic and Local Search Algorithms as Robust and Simple Optimization Tools," in: *Meta-Heuristics: Theory and Applications*, (Kluwer Academic Publishers, 1996) p. 63 82.

[133] M. Yagiura and T. Ibaraki, "Metaheuristics as Robust and Simple Optimization Tools," *Proceedings of IEEE International Conference on Evolutionary Computation* (1996) 541 546.

[134] M. Yagiura and T. Ibaraki, "The Use of Dynamic Programming in Genetic Algorithms for Permutation Problems," *European Journal of Operational Research*, 92 (1996) 387 401.

[135] M. Yagiura and T. Ibaraki, "Efficient 2 and 3-Flip Neighborhood Search Algorithms for the MAX SAT," *Proceedings of the 4th Annual International Computing and Combinatorics Conference* (1998) 105 116.

[136] M. Yagiura, H. Nagamochi and T. Ibaraki, "Two Comments on the Subtour Exchange Crossover Operator (in Japanese)," *Journal of Japanese Society for Artificial Intelligence*, 10 (1995) 464 467.

[137] M. Yagiura, T. Yamaguchi and T. Ibaraki, "A Variable Depth Search Algorithm for the Generalized Assignment Problem," in: *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, (Kluwer Academic Publishers, 1999) p. 459 471.

[138] M. Yagiura, T. Yamaguchi and T. Ibaraki, "A Variable Depth Search Algorithm with Branching Search for the Generalized Assignment Problem," *Optimization Methods and Software* (to appear).

[139] T. Yamada and R. Nakano, "A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems," *Proceedings of the 2nd International Workshop on Parallel Problem Solving from Nature* (1992) 281 290.

[140] M. Yamamura, T. Ono and S. Kobayashi, "Character-Preserving Genetic Algorithms for Traveling Salesman Problem (in Japanese)," *Journal of Japanese Society for Artificial Intelligence*, 7 (1992), 117 127.

[141] M. Yannakakis, "Computational Complexity," in: *Local Search in Combinatorial Optimization*, (John Wiley & Sons, 1997).

[142] D. Whitley, T. Starkweather and D. Fuquay, "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator," *Proceedings of the 3rd International Conference on Genetic Algorithms* (1989) 133 140.

A List of Author's Work

Journals and Books

1. M. Yagiura and T. Ibaraki, "On Genetic Crossover Operators for Sequencing Problems (in Japanese)," *T. IEE Japan*, 114-C (1994) 713 720.

2. M. Yagiura, H. Nagamochi and T. Ibaraki, "Two Comments on the Subtour Exchange Crossover Operator (in Japanese)," *Journal of Japanese Society for Artificial Intelligence*, 10 (1995) 464 467.

3. M. Yagiura and T. Ibaraki, "The Use of Dynamic Programming in Genetic Algorithms for Permutation Problems," *European Journal of Operational Research*, 92 (1996) 387 401.

4. M. Yagiura and T. Ibaraki, "Genetic and Local Search Algorithms as Robust and Simple Optimization Tools," in: *Meta-Heuristics: Theory and Applications*, (Kluwer Academic Publishers, 1996) p. 63 82.

5. N. Kawai, H. Ase, T. Ibaraki and M. Yagiura, "Scheduling of Shift Operations in a Container Terminal (in Japanese)," *Transactions of the Institute of Systems, Control and Information Engineers* 10 (1997) 182 190.

6. M. Yagiura, T. Yamaguchi and T. Ibaraki, "A Variable Depth Search Algorithm for the Generalized Assignment Problem," in: *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, (Kluwer Academic Publishers, 1999) p. 459 471.

7. T. Uno and M. Yagiura, "Fast Algorithms to Enumerate All Common Intervals of Two Permutations," *Algorithmica* (to appear).

8. M. Yagiura and T. Ibaraki, "Analyses on the 2 and 3-Flip Neighborhoods for the MAX SAT," *Journal of Combinatorial Optimization* (to appear).

9. M. Yagiura, T. Yamaguchi and T. Ibaraki, "A Variable Depth Search Algorithm with Branching Search for the Generalized Assignment Problem," *Optimization Methods and*

Software (to appear).

International Conferences

1. S. Katoh, M. Yagiura and T. Ibaraki, "Application of Genetic Algorithms to the Single Machine Scheduling Problem under Changing Environment," *Proceedings of International Conference on Advances in Production Management Systems* (1996) 543–546.

2. M. Yagiura and T. Ibaraki, "Metaheuristics as Robust and Simple Optimization Tools," *Proceedings of 1996 IEEE International Conference on Evolutionary Computation* (1996) 541–546.

3. T. Akutsu and M. Yagiura, "Linear Programming Based Approach for Learning Score Functions in Molecular Biology," *JAPAN-KOREA Joint Workshop on Algorithms and Computation* (1997) 144–151.

4. T. Akutsu and M. Yagiura, "On the Complexity of Deriving Score Functions from Examples for Problems in Molecular Biology," *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming* (1998) 832–843.

5. M. Yagiura and T. Ibaraki, "Efficient 2 and 3-Flip Neighborhood Search Algorithms for the MAX SAT," *Proceedings of the 4th Annual International Computing and Combinatorics Conference* (1998) 105–116.

Tutorials

1. M. Yagiura and T. Ibaraki, "Metaheuristics as Robust and Simple Optimization Tools (in Japanese)," *Proceedings of the Eighth RAMP Symposium* (1996) 109–124.

Unpublished Manuscripts

1. H. Ase, T. Ibaraki and M. Yagiura, "Removal Scheduling of a Multi-Story Mechanical Parking Facility (in Japanese)," submitted for publication.

2. F. Glover, T. Ibaraki and M. Yagiura, "An Ejection Chain Approach for the Generalized Assignment Problem," prepared for publication.

3. M. Yagiura and T. Ibaraki, "Efficient 2 and 3-Flip Neighborhood Search Algorithms for the MAX SAT: Experimental Evaluation," submitted for publication.