The Use of Dynamic Programming in Genetic Algorithms for Permutation Problems

Mutsunori YAGIURA and Toshihide IBARAKI

Department of Applied Mathematics and Physics

Faculty of Engineering

Kyoto University

Kyoto-city, 606-01 Japan

Abstract To deal with computationally hard problems, approximate algorithms are used to provide reasonably good solutions in practical time. *Genetic algorithms* are an example of the meta-heuristics which were recently introduced and which have been successfully applied to a variety of problems. 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, the optimal linear arrangement problem and the traveling salesman problem, all of which ask to compute optimum permutations of n objects and are known to be NP-hard. Computational results for randomly generated problem instances exhibit encouraging features of genetic DP algorithms.

Keywords: Genetic algorithm; Dynamic programming; Heuristics; Combinatorial optimization; Single machine scheduling problem

1 Introduction

There are numerous combinatorial optimization problems, for which computing exact optimal solutions is computationally intractable, e.g., those known as *NP-hard* [5]. However, in practice, we are usually satisfied with sufficiently good solutions, and in this sense, *approximate* (or *heuristic*) *algorithms* are very important.

There are several useful tools used to design approximate algorithms. The most common one is perhaps the *greedy* method, 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*, which starts from an initial feasible solution x and repeats replacing x with a better solution in its *neighborhood* $N(x)$ until no better solution is found in $N(x)$.

The so-called *meta-heuristics* combine these tools into more sophisticated algorithms. Among the well-known meta-heuristics are *multi-start local search* [4, 15], *simulated annealing*, *tabu search*, *genetic algorithm* [3, 7, 10] 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 x with a worse solution in $N(x)$ 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 $P \geq 1$ candidate solutions and improve them in the process of evolution. Among various modifications [3, 25], it is reported in [12, 17, 28] that introducing local search technique in the evolution process is quite effective. This is called the *genetic local search*. This may suggest that combining some of other techniques with genetic algorithms is also worth trying.

In this paper, we propose a variant of the genetic algorithm called *genetic DP* [29, 30]. 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) [11], the *optimal linear arrangement* problem (abbreviated as OLAP) [2] and the *traveling salesman* problem (abbreviated as TSP) [14], 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 [15] 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.

2 Genetic Algorithms

Before presenting the framework of genetic DP, we explain *genetic local search*, which reportedly gives good results to various combinatorial optimization problems.

2.1 Genetic Local Search

GENETIC LOCAL SEARCH (Positive integers P and Q ($> P$) are program parameters to be specified beforehand)

1 (Initialize): Construct P initial candidate solutions.

2 (Crossover and Improve): Repeat the following steps (a) and (b), or (a$'$) and (b) until Q candidate solutions are obtained.

- (a) (Crossover): Crossover two or more candidate solutions to generate a new solution.
- (a$'$) (Mutate): Mutate a candidate solution to generate a new solution.
- (b) (Local search): Apply local search to the solution of (a) to obtain a local optimal solution. Add the obtained solution to the set of candidate solutions unless it is already in the set.

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

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

In the traditional genetic algorithm, solutions are represented as binary strings of length n (i.e., genes), and crossover and mutation are defined as operations on these strings. However, in the above framework of genetic local search, any type of representations, and any operations of crossover and mutation can be used, since a variety of crossover and mutation operations have been proposed until now [18, 25].

Step 3 chooses P solutions from Q candidate solutions. For example, we may choose the best P of Q candidate solutions, or may choose P solutions randomly according to some biased probability so that better solutions are more likely to be chosen [3, 7, 29, 30].

2.2 Genetic DP Algorithm

In place of crossover and local search in step 2 of GENETIC LOCAL SEARCH, 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) and is described as follows.

GENETIC DP (Positive integers P and Q ($> P$) are program parameters to be specified beforehand)

1 (Initialize): Construct P initial candidate solutions.

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.

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

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

2.2.1 Step 2 (Crossover by DP)

Let a solution be a permutation $\sigma = \langle \sigma(1), \ldots, \sigma(n) \rangle$, i.e., an ordered sequence of n different elements, chosen from set $\{1, 2, \ldots, n\}$, 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

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

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 [15] that about 85% pairs of cities on the average are commonly adjacent in two tours obtained by using their algorithm, and $60 \sim 80\%$ 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 3.2, 4.1 and 5.1 as it depends on the particular problem being solved), it is possible to compute the best solution consistent with D.

2.2.2 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:

$$D := D - \{(i,k) \mid \sigma_1^{-1}(i) < \sigma_1^{-1}(k) \leq \sigma_1^{-1}(j))\}, \tag{2}$$

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

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.

2.2.3 Step 3 (Select)

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

$$\text{cost}(\sigma_1) \leq \text{cost}(\sigma_2). \tag{3}$$

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

(i) Select P solutions after the Q candidate solutions are formed in step 2.

(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 [30] 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.

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

$$\Delta_i = \text{cost}(\sigma_i) - \text{cost}(\sigma_{new}),\ i \in \{1,2\}, \tag{4}$$

$$p(x) = \min\{\frac{1}{a}x, 1\}. \tag{5}$$

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 1. 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.

2.2.4 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 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: $\{1,\ldots,n\} \rightarrow V$ 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

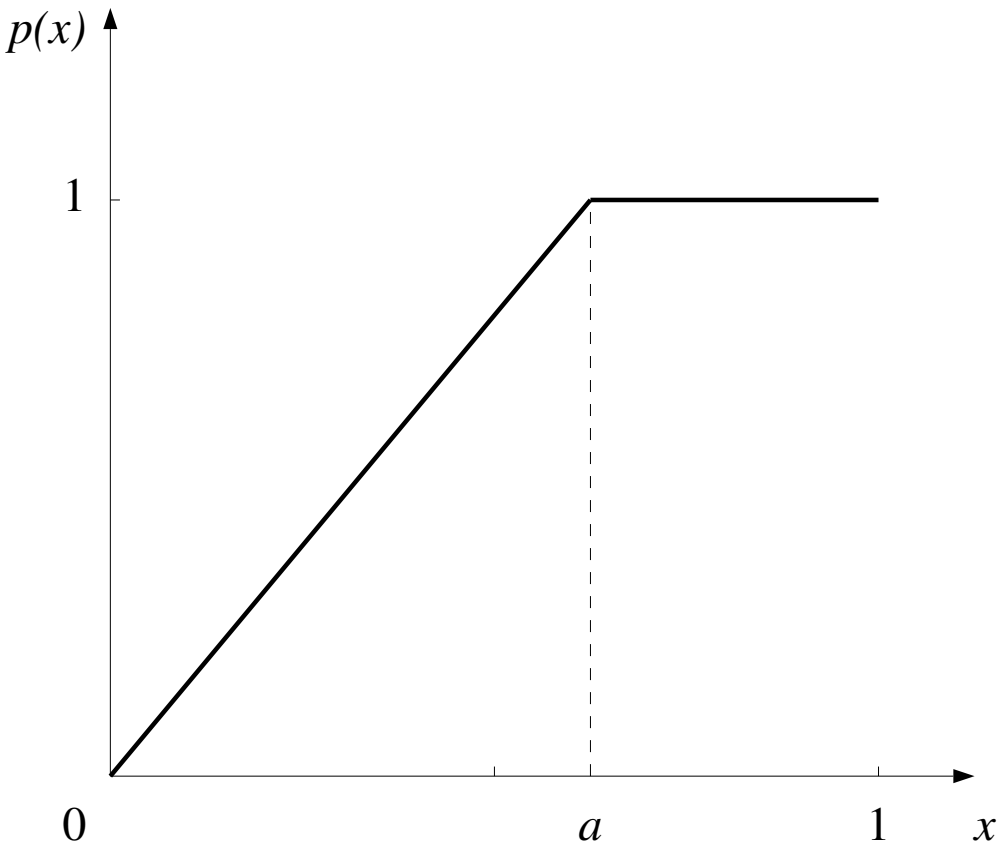

Figure 1: The probability function p(·).

$$\text{cost}(\sigma) = \sum_{i \in V} g_i(c_i). \tag{6}$$

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

$$g_i(c_i) = h_i \max\{d_i - c_i, 0\} + w_i \max\{0, c_i - d_i\}, \tag{7}$$

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.

3.1 Exact Algorithms

The basic dynamic programming recursion due to [9] can solve SMP exactly. Let $S \subseteq V$ be an arbitrary subset of jobs, and let $f^*(S)$ denote the minimum of cost function (6) 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

$$f^*(\phi) = 0, \tag{8}$$

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

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 (8) 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 [21]. Another type of algorithm *SSDP* (*successive sublimation dynamic programming*) was proposed in [11]. 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.

3.2 Genetic DP Algorithm

In this section, we specialize the genetic DP of section 2.2 to the SMP, and describe the details of steps 1 and 2.

3.2.1 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

$$M = \{i \in V \mid i \text{ is not scheduled yet}\}. \tag{9}$$

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

$$e_1(i,t) = g_i(t + p_i + \bar{p})\delta_i(t) - g_i(t + p_i)(1 - \delta_i(t)), \tag{10}$$

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

$$\bar{p} = \sum_{i \in V} p_i / n, \tag{11}$$

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

$$\delta_i(t) = \begin{cases} 1, & t + \bar{p} + p_i \geq d_i, \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

The second and third evaluation functions are

$$e_2(i,t) = w_i\delta_i(t) - h_i(1 - \delta_i(t)), \tag{13}$$

and

$$e_3(i,t) = \frac{w_i}{p_i}\delta_i(t) - \frac{h_i}{p_i}(1 - \delta_i(t)). \tag{14}$$

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

$$e_4(i,t) = -g_i(t + p_i) - \sum_{l=1}^{k} g_{j_l}(c_{j_l}), \tag{15}$$

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 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 [4], in the framework of multi-start local search.

3.2.2 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.

$$f^*(\phi) = 0, \tag{16}$$

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

where

$$V^*(D) = \{S \subseteq V \mid j \in S \text{ and } (i,j) \in D \Rightarrow i \in S\}, \tag{17}$$

and

$$I(S) = \{i \in S \mid \text{no } j \in S \text{ satisfies } j \neq i \text{ and } (i,j) \in D\}. \tag{18}$$

While the DP recursion by (8) generates all 2^n subsets S of V, the recursion by (16) 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

$$D := D \cup \{(i,j) \mid \sigma_1^{-1}(i) < k \text{ and } k \leq \sigma_1^{-1}(j)\}. \tag{19}$$

The value $|V^*(D)|$ is important in estimating computation time, since it gives the number of states in dynamic programming recursion (16). 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 $|V^*(D)|$ [27]. It is known [23] 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 [26], which holds true if neither mutation (section 2.2.2) nor augmentation (19) 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 (16) will never be greater than that of σ_1 (recall assumption (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 $\sigma : \{1, \ldots, n\} \rightarrow V$, meaning that vertex $\sigma(i)$ is placed in the i-th position in a straight line. The cost of a permutation σ is

$$\sum_{i=1}^{m} w_i \max_{k,l \in S_i} \{|\sigma^{-1}(k) - \sigma^{-1}(l)|\}, \tag{20}$$

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 [1, 2, 13, 24]. 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 [6].

4.1 Genetic DP Algorithm

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

4.1.1 Step 1 (Initialize)

To generate different types of initial candidate solutions, we use two heuristics, Kang's greedy method [13] and the clustering method [2, 24].

Although Kang's method was stated in [13] 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 netcut$(\{i\}, V - \{i\})$, where netcut(L, R) denotes the sum of net weights between L and R, i.e.,

$$\text{netcut}(L, R) = \sum_{i=1}^{m} w_i \delta_i(L, R), \tag{21}$$

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

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 netcut$(V - M, \{i\})$ − netcut$(\{i\}, M - \{i\})$ at each iteration, where

$$M = \{i \in V \mid i \text{ has not been placed yet}\}. \tag{22}$$

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 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.

4.1.2 Step 2 (Crossover by DP)

Here we consider the following two methods.

(A) Compute the common partial order D of (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

$$f^*(L) = 0, \ |L| \leq 1, \tag{23}$$

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

where

$$V^*(D) = \{L \subseteq V \mid |L| > 1, (j \in L \text{ and } (i, j) \in D \Rightarrow i \in L)\}, \tag{24}$$

$I(\cdot)$ is defined in (18), and $f^*(L)$ denotes the minimum cost (20) 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 (23), we add the chain constraints to method (A) in the following manner. Call that vertices i and j are adjacent in σ if

$$|\sigma^{-1}(i) - \sigma^{-1}(j)| = 1, \tag{25}$$

and denote

$$AD(\sigma) = \{(i,j) \mid \text{vertices } i \text{ and } j \text{ are adjacent}\}. \tag{26}$$

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 (23) 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 (19) is applied.

A computational comparison of these two methods for various b is shown in figure 2, in which 11 test instances of up to $n = 40$ are solved. It shows how the total cost (20) 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 2.2 and 3.2 for the details of these parameters) and b is varied from 0 to up to 100. Figure 2 shows superiority of (B) to (A). In the computational experiment in section 6, where genetic DP is compared with other approximate methods, we adopt (B).

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 $V = \{1, \ldots, n\}$ and each edge $(i,j) \in E$ has length d_{ij}. The symmetric traveling salesman

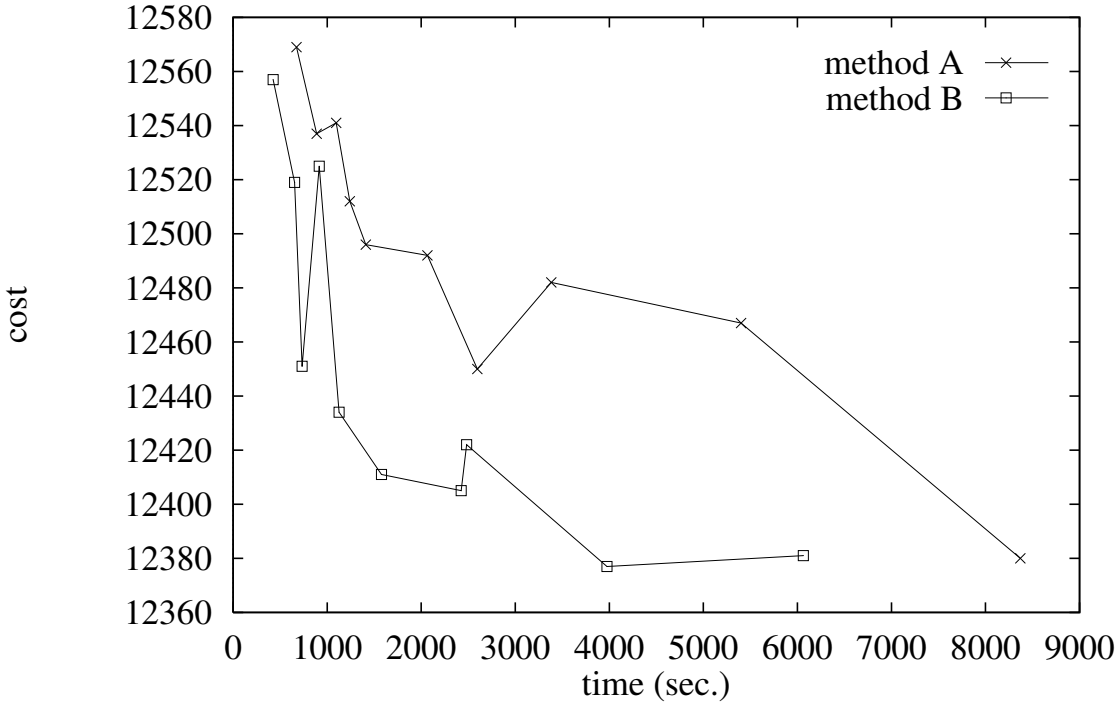

Figure 2: The effect of chain constraints.

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

$$\sum_{i=1}^{n-1} d_{\sigma(i)\sigma(i+1)} + d_{\sigma(n)\sigma(1)}. \tag{27}$$

Numerous exact and approximate algorithms have been proposed for this problem [14], and it is reported that exact optimal solutions have been obtained for problem instances of up to $n = 2000 \sim 3000$ [8, 19] (500,000 in the case of asymmetric version [16]).

5.1 Genetic DP Algorithm

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

5.1.1 Step 1 (Initialize)

We use the *arbitrary insertion* method [14, 22] 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 [14], 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.

5.1.2 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 (1). The best tour, which is consistent with D, and its cost $f^*(V)$, can be obtained by solving

$$\begin{aligned} & f^*(\{1\}, 1) = 0, \\ & f^*(S, i) = \min_{j \in I(S) - \{i\}} \{f^*(S - \{i\}, j) + d_{ji}\}, \ S \in V^*(D), \\ & f^*(V) = \min_{i \in I(V)} \{f^*(V, i) + d_{i1}\}, \end{aligned} \tag{28}$$

where

$$V^*(D) = \{S \subseteq V \mid j \in S \text{ and } (i,j) \in D \Rightarrow i \in S\}, \tag{29}$$

and $I(\cdot)$ is the same as (18). $f^*(S,i)$ 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 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.

6 Computational Results for Three Problems

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., [20]) 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,

$$\begin{aligned} RDD &= 0.2, 0.4, 0.6, 0.8, 1.0, \\ LF &= 0.2, 0.4, \end{aligned}$$

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

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

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 2.2.4),

a: parameter in (5), 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 3.2.2),

s: number of mutations (see section 2.2.2).

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 (6), (20) or (27) 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 1 unless otherwise stated. Only the parameter examined is changed.

First we examined the effect of r. Figure 3 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 [29].

Second we examined the effect of a. The results for SMP are shown in figure 4. 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,

Table 1: 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.

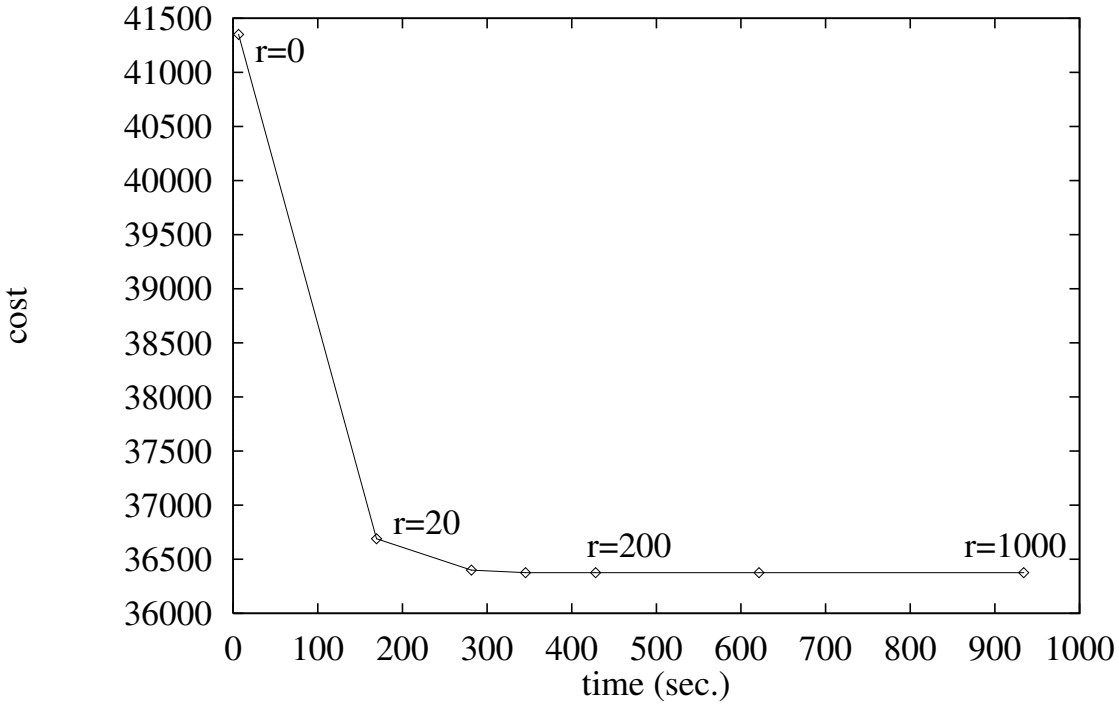

Figure 3: The effect of the number of iterations r (SMP).

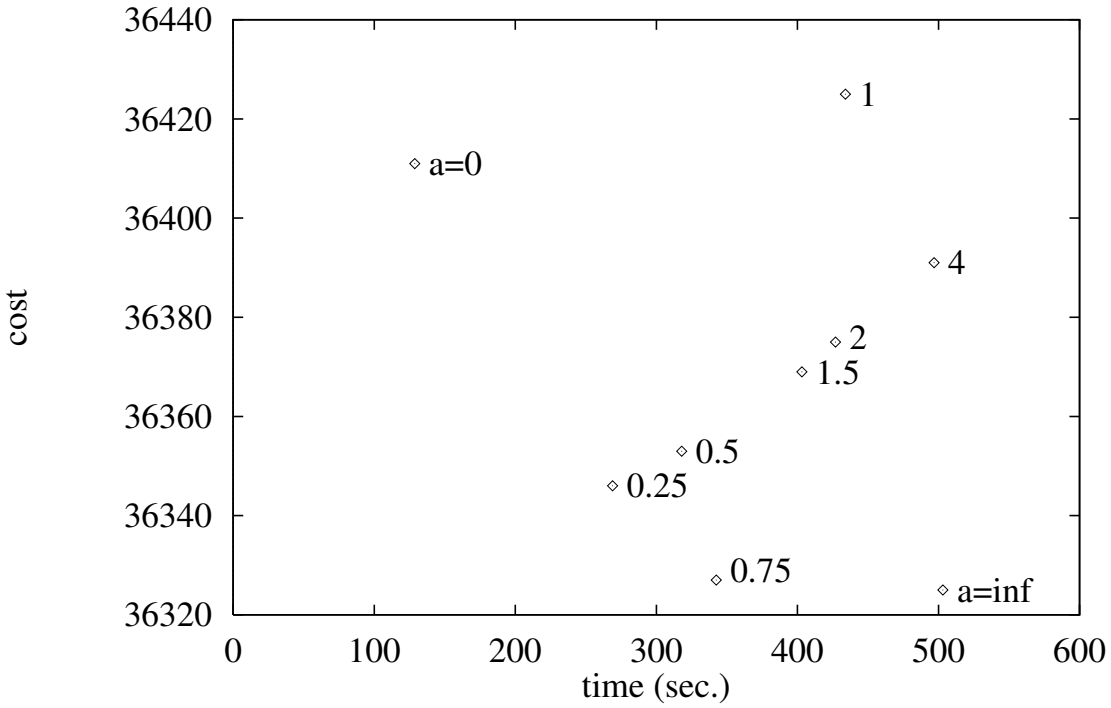

Figure 4: The effect of parameter a in the probability function p(·) (SMP).

setting a beyond 0.75 seemed to consume more time without gaining much improvement. The results for OLAP and TSP are similar [29].

Next we examined P, b and s. The results are exhibited in figures 5, 6 and 7 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 2.

Table 2: 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

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

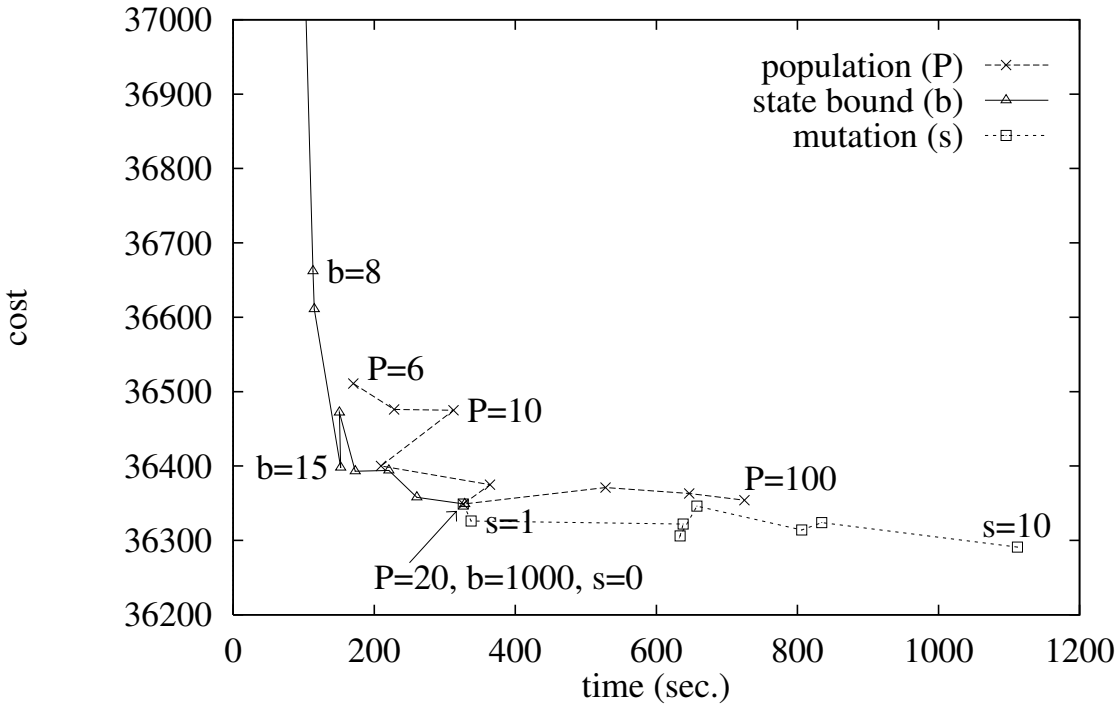

Figure 5: The effect of parameters P, b, s (SMP).

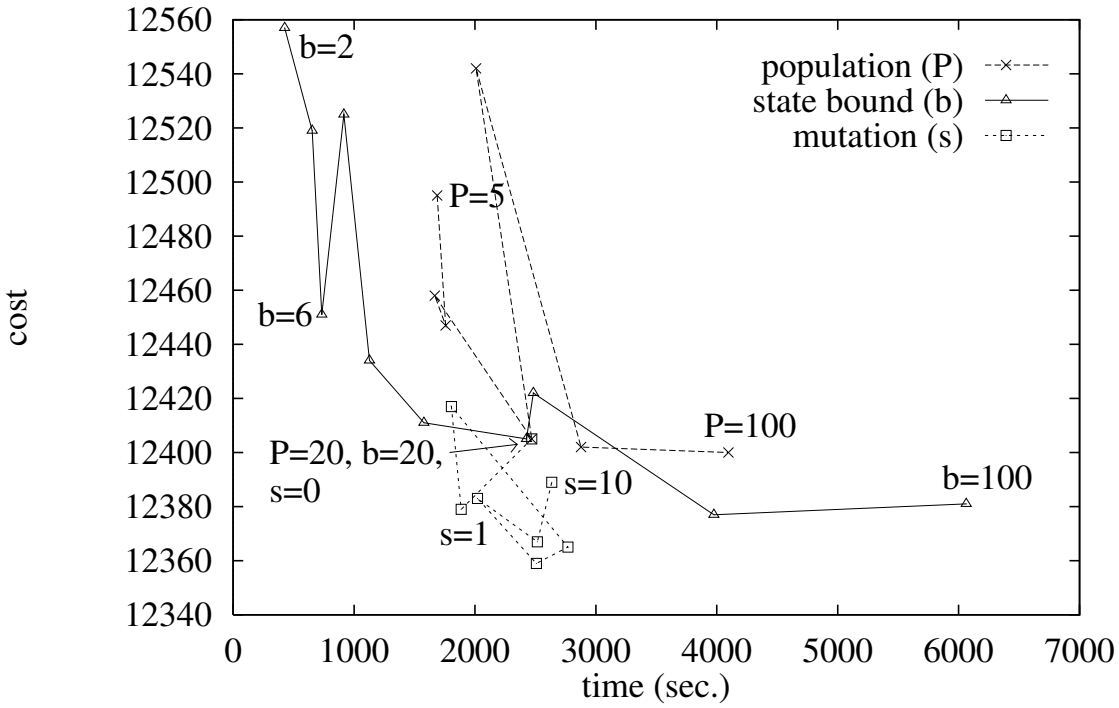

Figure 6: The effect of parameters P, b, s (OLAP).

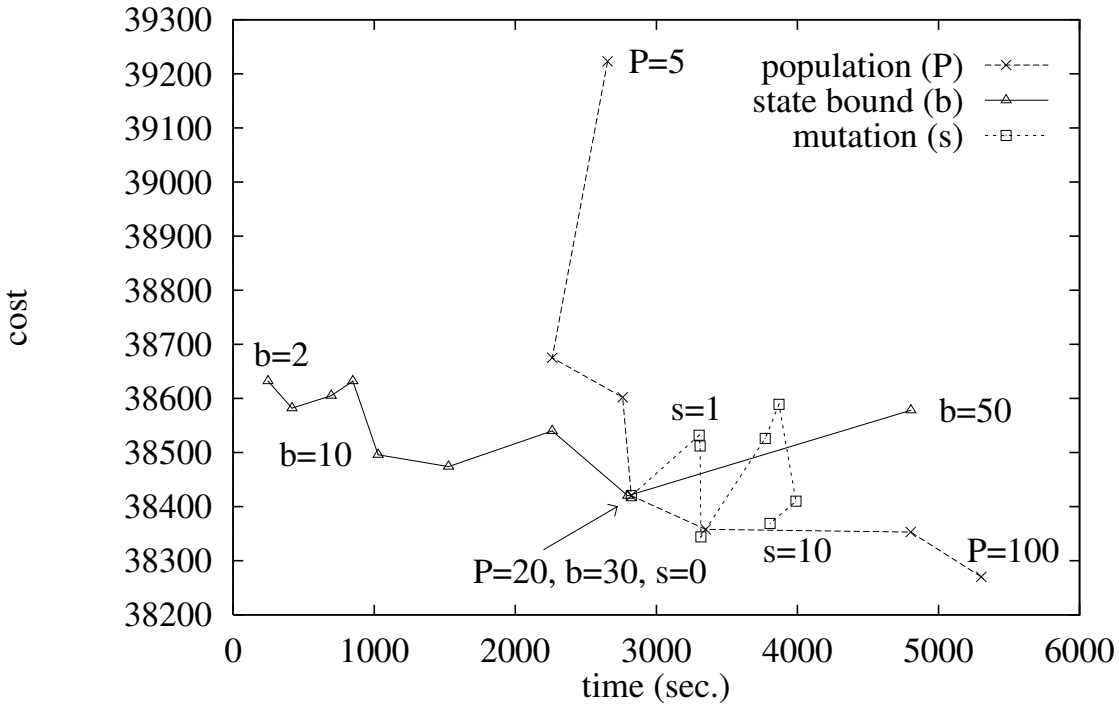

Figure 7: The effect of parameters P, b, s (TSP).

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 8 in seconds.

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 in seconds.

Table 3: The computational time (in seconds) to solve OLAP by algorithms DP and genetic DP.

n	DP	Genetic DP
15	115	54
20	6554	120

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 [15], as will be reported in the next subsection.

6.4 Comparison with Other Heuristic Algorithms

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

6.4.1 Details of Other Heuristic Algorithms

The first heuristic algorithms tested are the following two, already explained in section 1:

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

(2) Genetic local search algorithm (Genetic-Local).

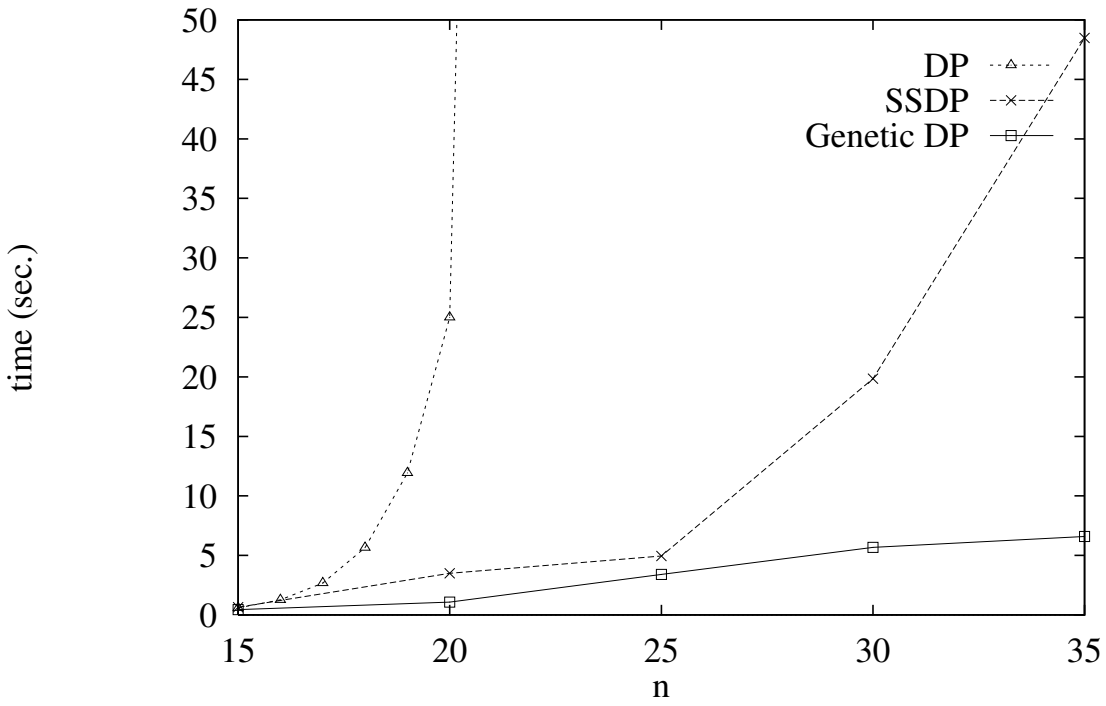

Figure 8: The computational time (in seconds) to solve SMP by algorithms DP, SSDP and genetic DP.

In Multi-Local and Genetic-Local, 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 9). The crossover operation for Genetic-Local is performed as follows (see figure 10). 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 [17] for TSP. Step 3 (select) and step 4 (iterate) for Genetic-Local are the same as genetic DP.

This Genetic-Local is a bit different from the genetic local search algorithm proposed in other literature, such as [17, 28], in which all initial candidate solutions are improved by local search, before applying crossover operation. This original type of genetic local search algorithm (Genetic-Local#) 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 (16), (23) or (28) to all pairs in P initial candidates, and then halts. Initial solutions are the same as those used in genetic DP. This algorithm is tested to see the effect of only the DP part of genetic DP.

(4) *Or-opt* procedure [14] and *Lin-Kernighan* algorithm (abbreviated as LK) [15]. 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.

6.4.2 Results and Discussions

Figures from 11 to 15 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 11 shows a comparison of simple GA and Genetic-Local for SMP. Similar results are obtained for OLAP and TSP [29]. These results indicate a rather discouraging feature of simple GA, which is also observed in other references such as [28]. Figure 12 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 13, 14 and 15 compare genetic DP with other heuristic algorithms. In the case of

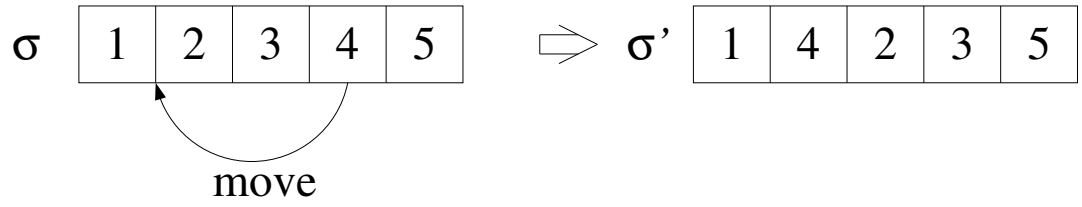

Figure 9: A solution in neighborhood $N(\sigma)$, corresponding to $i = 4$ and $j = 2$.

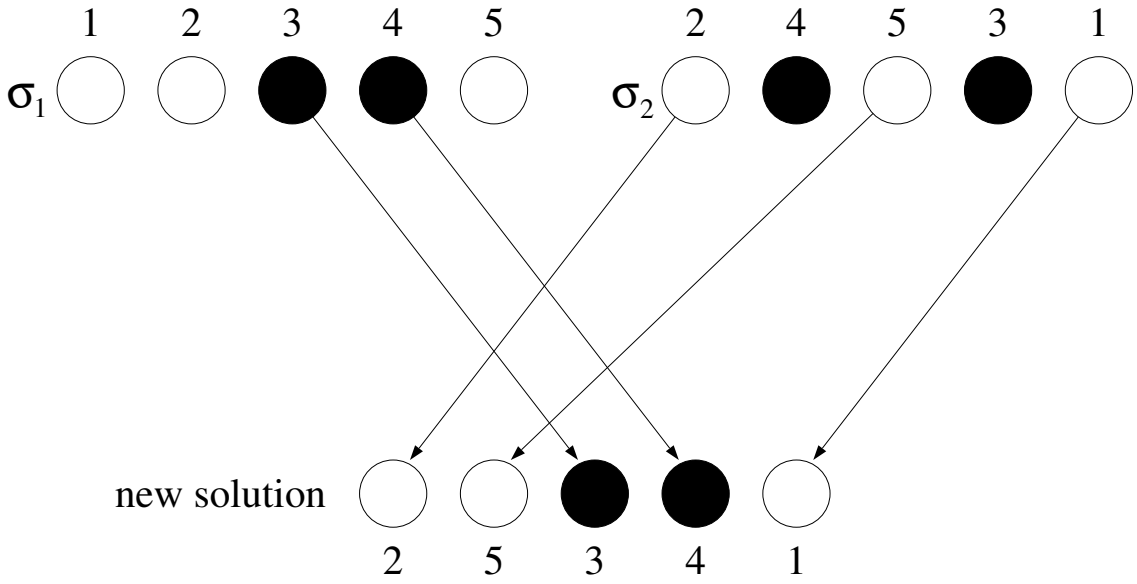

Figure 10: An example of crossover operation for Genetic-Local, where $l = 2$, $k = 3$.

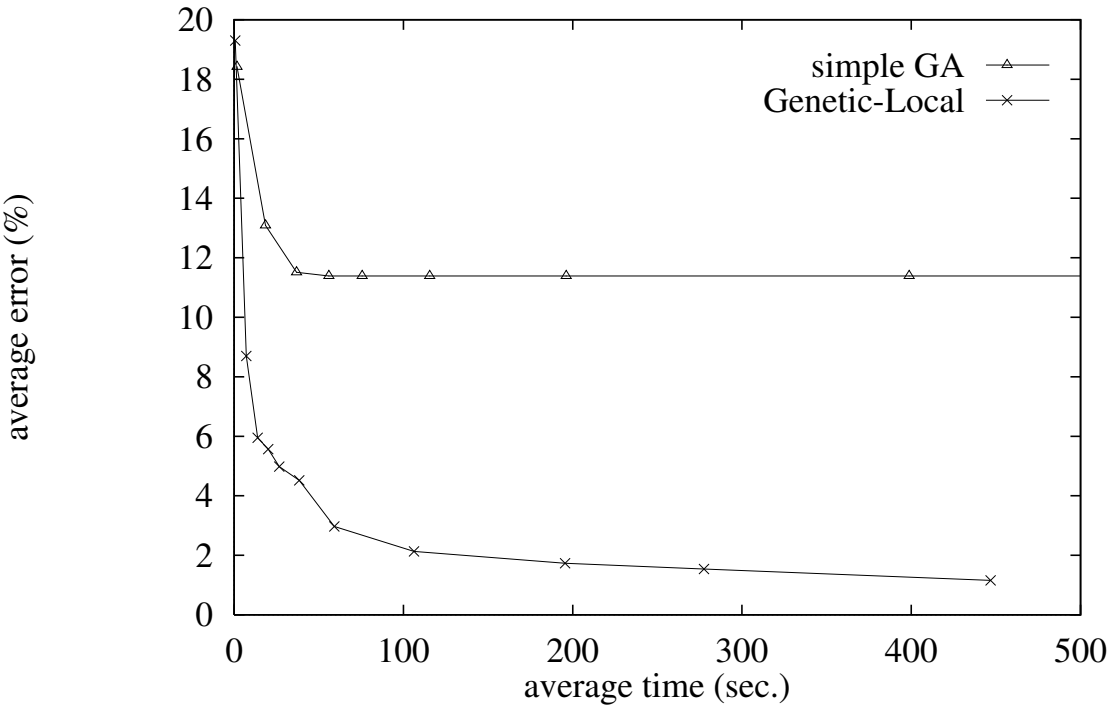

Figure 11: Comparison of simple GA and Genetic-Local (SMP).

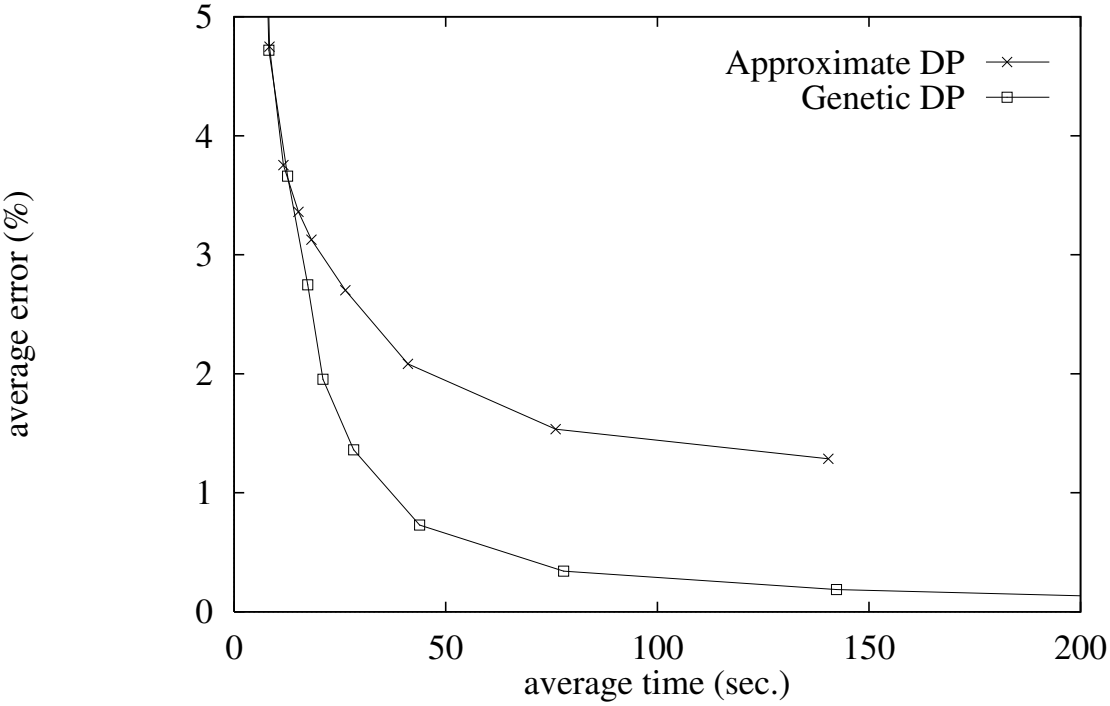

Figure 12: Comparison of approximate DP and genetic DP (SMP).

SMP, genetic DP triumphed over Multi-Local and Genetic-Local. Genetic DP could obtain better solutions in shorter time. In the case of OLAP, genetic DP obtained better solutions than Multi-Local, but when longer computational time was allowed, Genetic-Local obtained slightly better solutions. In the case of TSP, genetic DP could get better solutions than Multi-Local, Genetic-Local and Or-opt when sufficient computational time was allowed. But LK could obtain much better solutions in shorter time.

Finally, Figure 16 shows how the computational time for SMP increases as the size of 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 Multi-Local and Genetic-Local increase a bit more rapidly than genetic DP. Similar results were obtained for OLAP and TSP [29]. This is because the number of states necessary for the DP recursion of genetic DP was bounded by bn using operation (19) and the computational time for each DP recursion (16) 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.

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. Combination of such special purpose heuristics with genetic algorithms may be an important subject of future study. It is noted, however, that an advantage of general meta-heuristics, 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.

Acknowledgments The authors are grateful to Professor Erwin Pesch, currently with

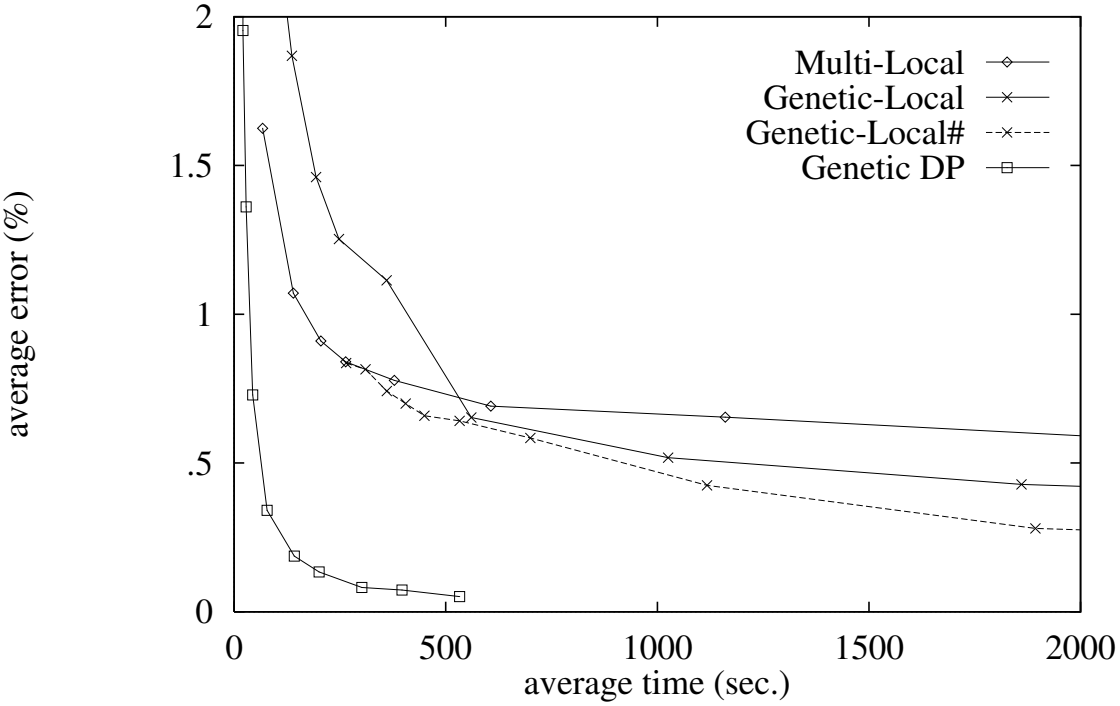

Figure 13: A comparison of the four algorithms (SMP).

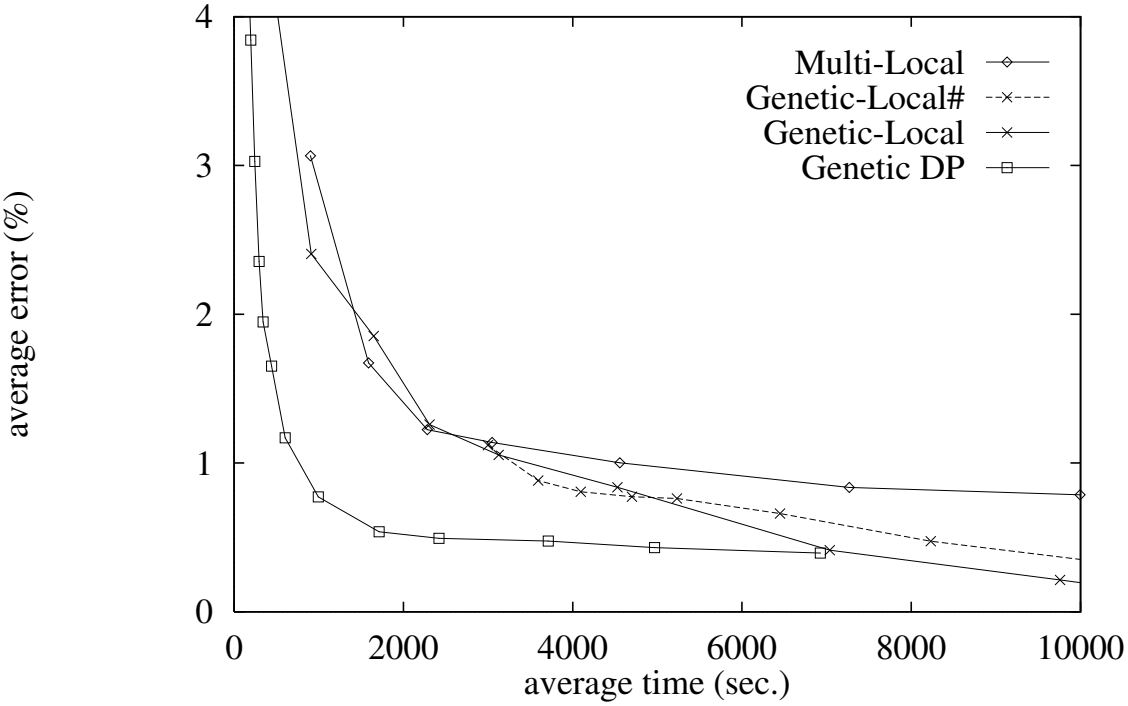

Figure 14: A comparison of the four algorithms (OLAP).

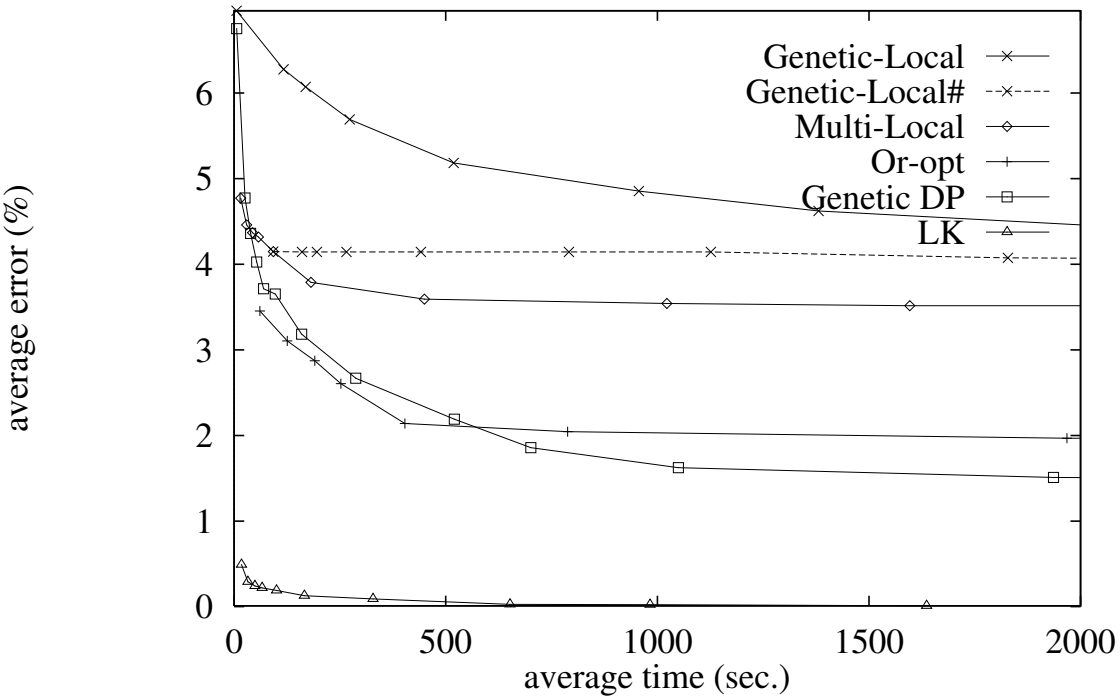

Figure 15: A comparison of the six algorithms (TSP).

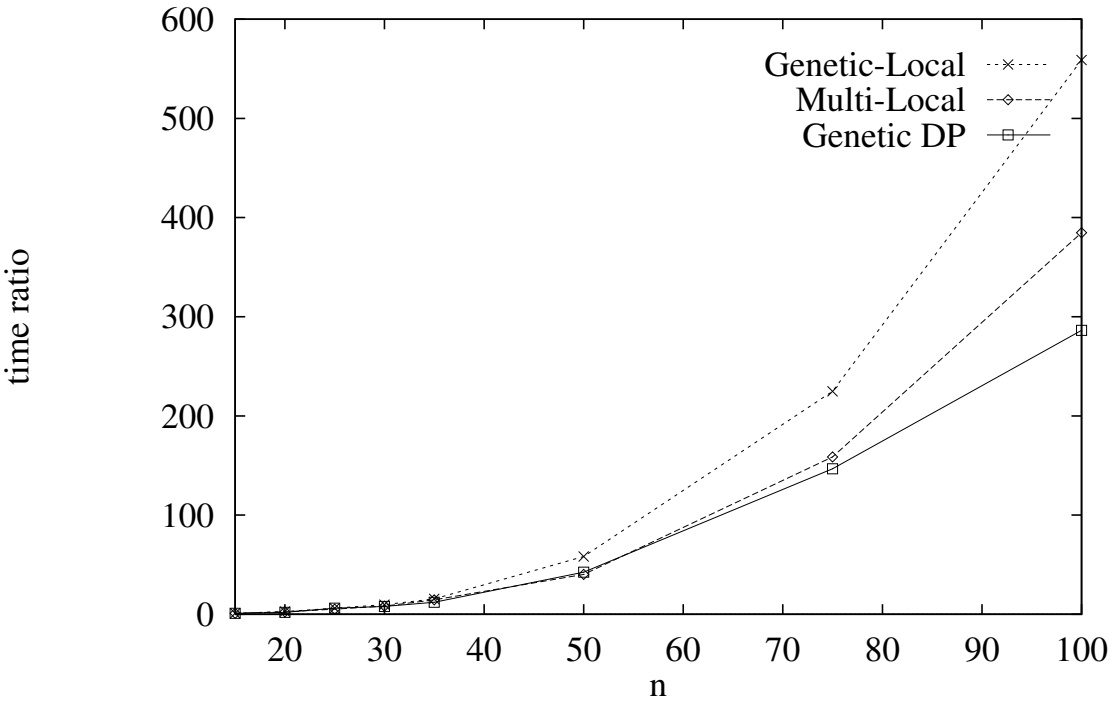

Figure 16: Time ratio with respect to n (time for n=15 is regarded as 1) of the three algorithms (SMP).

Limburg University, for his providing us with the original code of Lin-Kernighan's algorithm.

References

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

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

[3] Davis, L., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[4] Feo, T.A., Venkatraman, K., and Bard, J.F., "A GRASP for Single Machine Scheduling with Due Dates and Earliness Penalties", Operations Research Group, Department of Mechanical Engineering, University of Texas at Austin, Austin, Texas 78712–1063, 1989.

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

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

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

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

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

[10] Holland, J., *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, 1975., and MIT Press, 1992.

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

[12] Jog, P., Suh, J.Y., and Van Gucht, D., "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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] Ulder, N.L.J., Pesch, E., Van Laarhouven, P.J.M., Bandelt, H.-J., and Aarts, E.H.L., "Genetic Local Search Algorithms for the Traveling Salesman Problem," *Proceedings of the 1st International Workshop on Parallel Problem Solving from Nature* (1990).

[29] Yagiura, M., "Genetic Algorithms for Solving Some Combinatorial Optimization Problems," Master thesis, Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, 1993.

[30] Yagiura, M., and Ibaraki, T., "A Genetic Algorithm for Solving the Single Machine Scheduling Problem (in Japanese)," *IEICE Technical Report*, COMP91–17 (1991) 43–52.