INFORMS Journal on Computing
Vol. 25, No. 2, Spring 2013, pp. 346–363
ISSN 1091-9856 (print) | ISSN 1526-5528 (online)

http://dx.doi.org/10.1287/ijoc.1120.0506
© 2013 INFORMS

A Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem

Yuichi Nagata, Shigenobu Kobayashi
Department of Computational Intelligence and Systems Science, Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, Yokohama, Kanagawa 226-8503, Japan
{nagata@fe.dis.titech.ac.jp, kobayasi@dis.titech.ac.jp}

This paper presents a genetic algorithm (GA) for solving the traveling salesman problem (TSP). To construct a powerful GA, we use edge assembly crossover (EAX) and make substantial enhancements to it: (i) localization of EAX together with its efficient implementation and (ii) the use of a local search procedure in EAX to determine good combinations of building blocks of parent solutions for generating even better offspring solutions from very high-quality parent solutions. In addition, we develop (iii) an innovative selection model for maintaining population diversity at a negligible computational cost. Experimental results on well-studied TSP benchmarks demonstrate that the proposed GA outperforms state-of-the-art heuristic algorithms in finding very high-quality solutions on instances with up to 200,000 cities. In contrast to the state-of-the-art TSP heuristics, which are all based on the Lin–Kernighan (LK) algorithm, our GA achieves top performance without using an LK-based algorithm.

Key words: genetic algorithm; traveling salesman; crossover; population diversity
History: Accepted by David Woodruff, Area Editor for Heuristic Search and Learning; received March 2011; revised September 2011, January 2012; accepted February 2012. Published online in *Articles in Advance* May 4, 2012.

1. Introduction

The traveling salesman problem (TSP) is one of the most cited NP-hard combinatorial optimization problems because it is so intuitive and easy to understand but difficult to solve. It is a challenging optimization problem of significant academic value as it is often used as a benchmark problem when new solution approaches are developed. Intensive research efforts have therefore been directed toward the development of both exact and heuristic (approximation) algorithms for the TSP.

The current state-of-the-art exact algorithm "Concorde" (available at http://www.tsp.gatech.edu) frequently solves TSPs with up to 1,000 cities to optimality within a reasonable computation time. This algorithm also solves large nontrivial TSPs with at most 85,900 cities (Bixby et al. 2009) with the help of optimal solutions found in advance by powerful heuristic algorithms, but at the cost of a very long computation time. Here, knowledge of a tour that is indeed proved to be optimal enables better configuration of exact methods and substantially reduces the run-time. In this paper, we propose a genetic algorithm (GA) aimed at finding very high-quality solutions (e.g., new best-known solutions) to instances with up to 200,000 cities. Such solutions will provide useful information for powerful exact algorithms for solving unsolved TSP instances.

The simplest heuristic approach for the TSP would be a (greedy) local search with the *k-opt* neighborhood. This neighborhood is defined as a set of solutions that are transformed from a current tour by replacing at most k edges in other possible ways. The solution quality will improve with increasing k when the neighborhood is exhaustively searched at each iteration, but at the cost of rapidly increasing computation time. To handle the problem of the *k-opt* neighborhood with a greater value of k, Lin and Kernighan (1973) developed a variable-depth search where the *k-opt* neighborhood (e.g., $k = 50$) is partially searched with a smart pruning strategy.

According to the extensive survey of heuristic algorithms for the TSP by Johnson and McGeoch (1997) and recent papers, the most effective heuristic algorithms in various periods in the history of the TSP have been the Lin–Kernighan (LK) algorithm and its extensions. The LK algorithm was known as the most effective heuristic approach from its proposal in 1973 until 1989, providing best-known solutions on many benchmark instances simply by restarting the algorithm. Martin et al. (1991) proposed a more effective approach, which is often referred to as chained LK (CLK) or iterated LK (ILK). In this scheme, the LK algorithm is performed iteratively, starting with an initial solution obtained by perturbing a local optimum in the previous iterations. Because of its effectiveness, several enhancements to CLK were proposed by Johnson and McGeoch (1997), Walshaw (2002), and Applegate et al. (2003). Helsgaun (2000) developed another innovative improvement to the LK algorithm, where the variable-depth search is guided on the basis of optimal solutions to relaxation problems (1-tree) of the TSP. Helsgaun's LK (LKH) algorithm has been further improved (Helsgaun 2006, 2009), and the current version of LKH (LKH-2) is known as one of the most effective heuristic algorithms for finding very high-quality solutions. Moreover, recent very powerful approximation methods have incorporated LKH or LKH-2 into the procedures. Cook and Seymour (2003) proposed a tour-merging technique, where a "branch-width" algorithm attempts to find an optimal solution on a sparse graph consisting of the union of the edge sets of the tours (e.g., 10–40 tours) obtained by independent LKH runs. Dong et al. (2009) proposed a technique for reducing the size of TSPs by fixing edges that are likely to be in an optimal tour; the edges to be fixed were determined as the intersection of the edge sets of the tours (e.g., 10–20 tours) obtained by independent LKH runs (pseudo backbone contraction). LKH-2 was then applied to the reduced TSP instance. In fact, most of the current best-known solutions to the well-known TSP benchmarks have been found by the LKH, LKH-2, tour merging, and Dong's approach.

Hybrid genetic algorithms (Hybrid GAs), which are also referred to as memetic algorithms (MAs) or genetic local search algorithms, have also offered promising approaches to the TSP. Early attempts include those by Ulder et al. (1991) and Merz and Freisleben (1997), where a local search algorithm with the 2-*opt* neighborhood and the LK algorithm were incorporated, respectively. Recent more powerful hybrid GAs include those by Tsai et al. (2004) and Nguyen et al. (2007), where a CLK algorithm and a variant of the LKH algorithm were incorporated, respectively. The performance of hybrid GAs strongly depends on the effectiveness and efficiency of the incorporated local search algorithms, and it seems natural to incorporate effective LK-based algorithms in the construction of competitive hybrid GAs.

Genetic algorithms (GAs) have also been applied to the TSP. Starting from early attempts (Goldberg and Lingle 1985, Grefenstette et al. 1985, Whitley et al. 1989), a number of GAs have been developed in order to improve the performance. In particular, much effort has been devoted to developing crossover operators appropriate to the TSP (Mathias and Whitley 1992, Yamamura et al. 1996, Maekawa et al. 1996, Freisleben and Merz 1996, Nagata and Kobayashi 1997, Whitley et al. 2010) because the performance of GAs strongly depends on the design of crossover operators. Despite such efforts, however, no GA that is competitive with effective LK-based local search algorithms has ever been proposed. One inherent drawback of GAs is the lack of local optimization capability, and therefore hybrid GAs have been the focus of the recent trend of constructing powerful heuristic algorithms for combinatorial optimization problems. Another general drawback of GAs is the execution time because GAs usually require longer computation times to exercise their capabilities compared to efficiently implemented local search algorithms. Indeed, many researchers have concluded that the use of GAs is not a promising direction for developing competitive heuristic algorithms, at least for the TSP.

Nevertheless, in this paper, we demonstrate that we can construct a GA that outperforms the state-of-the-art heuristic algorithms in finding very high-quality solutions, without using any LK-based algorithms. In our approach, we use edge assembly crossover (EAX) (Nagata and Kobayashi 1997) as a crossover operator. EAX generates offspring solutions by combining edges from two parent solutions and adding relatively few new short edges, which are determined by a simple search procedure. EAX therefore has the capability of local optimization and is known as one of the most effective crossover operators for the TSP. In fact, a GA using EAX (Nagata and Kobayashi 1997) has been one of the most effective GAs for the TSP, but it still cannot compete with effective LK-based algorithms such as CLK and LKH.

To improve the performance of the original GA using EAX, we have enhanced the original EAX in two ways in our preliminary work: (i) *localization* of EAX together with its efficient implementation (Nagata 2006a), and (ii) an improvement in EAX for generating even better offspring solutions from very high-quality parent solutions (Nagata 2006b). Moreover, we have proposed (iii) an innovative selection model for maintaining population diversity at a negligible computational cost (Nagata 2006a). In addition, this paper presents a new, more sophisticated version of EAX with regard to enhancement (ii) where we introduce a simple local search procedure into EAX to determine good combinations of the parents' edges. This paper gives a full description of these enhancements and provides more instructive analysis of the impact of these enhancements from a unified viewpoint. Moreover, a wider class of benchmarks and larger instances with up to 200,000 cities are now also considered. The program code of the proposed GA is available in the online supplement of this paper (available at http://dx.doi.org/10.1287/ijoc.1120.0506).

Experimental results on 57 well-studied TSP benchmark instances (the largest size is 85,900) show that the proposed GA outperforms the state-of-the-art LK-based algorithms in finding very high-quality solutions; the GA found optimal or best-known solutions for most benchmark instances in a reasonable computation time and improved five best-known solutions. Moreover, the GA improved all best-known solutions for a set of six larger instances called "Art TSPs" with sizes ranging from 100,000 to 200,000.

The remainder of this paper is organized as follows. In §2, we first describe the key ideas and outline the proposed GA. In §3, the enhancements of EAX are presented. Section 4 gives the selection model for maintaining population diversity. In §5, we analyze the impact of the proposed enhancements and present detailed results on all benchmark instances. Conclusions are provided in §6.

2. Basic Framework

In this section, we describe the key ideas and outline the GA proposed in this paper.

2.1. Basic Ideas

GAs for the TSP usually require longer computation times to exercise their capabilities than do efficiently implemented local-search-based algorithms. One reason for this is the nature of the population-based search. We believe, however, that the major reason arises from the fact that crossover operators require more computational effort to generate an offspring solution than do local search operators to evaluate a solution in the neighborhood.

In principle, a crossover operator requires at least $O(N)$ time (N is the number of cities) to generate an offspring solution if it generates an offspring solution by equally combining edges from two solutions selected as parents (some new edges that do not exist in either parent may be included). Most crossover operators for the TSP, including the original EAX (Nagata and Kobayashi 1997), use similar approaches, which makes it possible for population members to escape from deep local optima. In this sense, the original EAX is an efficient crossover operator because it can generate an offspring solution in $O(N)$ time.

Local search operators, on the other hand, usually evaluate a solution in the neighborhood more efficiently. For example, any solution in the *2-opt* neighborhood can be evaluated in $O(1)$ time. Note that it requires $O(\log N)$ time to update the current solution even with the use of an efficient data structure (Fredman et al. 2005), but the number of evaluations is usually much greater than the number of updates in a local search procedure.

To reduce the computational cost of EAX, we proposed *localization* of EAX together with an efficient implementation of this crossover operator (Nagata 2006a). A localized version of EAX generates offspring solutions from one of the parent solutions, denoted as p_A, by replacing relatively few edges with edges selected from the other parent, denoted as p_B (together with a few new edges). This approach enables generation of an offspring solution in less than $O(N)$ time by making use of the fact that it will be similar to p_A. In addition, localization of EAX contributes to maintaining population diversity coupled with an appropriate GA framework where only p_A is replaced with an offspring solution in the selection for survival. We therefore use only a localized version of EAX from the start of the search until it can no longer effectively generate offspring solutions that improve p_A.

When the localized version of EAX cannot generate an offspring solution that improves p_A, p_A will get trapped in a deep local optimum. In this case, the number of edges replaced by EAX should be increased to further improve p_A, and the localized version of EAX should be switched to another version of EAX. In this paper, we use the term "*global* version of EAX" to refer to a version of EAX that exchanges more edges than does the localized version of EAX. In our preliminary work (Nagata 2006a), we employed the original EAX as a global version of EAX. Then, we proposed a more effective global version of EAX (Nagata 2006b). In this paper, we further propose a more sophisticated global version of EAX along with an innovative design concept.

2.2. GA Framework

Algorithm 1 gives the basic framework of our GA. The population consists of N_{pop} solutions, where N_{pop} is a parameter. Individuals in the initial population are generated by an appropriate procedure (line 1). Here, we use a greedy local search algorithm with the *2-opt* neighborhood because it is reasonable to use a simple local search procedure to efficiently obtain N_{pop} solutions with a certain level of quality. Details of the local search algorithm are presented in the online supplement for readers who are interested in the implementation.

In each generation (lines 3–8) of the GA, each of the population members is selected once as parent p_A and once as parent p_B, in random order (lines 3 and 5). For each pair of parents, EAX generates N_{ch} offspring solutions (line 6), where N_{ch} is a parameter. Then, a best solution among the generated offspring solutions and p_A is selected according to a given criterion, and the selected solution replaces the population member selected as p_A (line 7). Therefore, no replacement occurs, if all offspring solutions are worse than p_A. This inhibits the loss of population diversity without improvement of the average solution quality of the population, and therefore no mutation is used in our GA. Iterations of generation are repeated until a termination condition is met (line 9). Finally, a solution with the shortest tour length in the population is returned (line 10).

The search process of the GA consists of two stages. First, we use a localized version of EAX as the crossover operator from the start of the search until no improvement in the best solution is found over a period of generations (stage I). After that, we switch to a global version of EAX and use it until the end of the search (stage II). More precisely, let G be the number of generations at which no improvement in the best solution is found over the recent $1{,}500/N_{\text{ch}}$ generations. If the value of G has already been determined and the best solution does not improve over the last $G/10$ generations, we terminate stage I and proceed to stage II. Stage II is also terminated by the same condition (G and the "stagnation" counter are initialized at the beginning of this stage).

Because a localized version of EAX generates offspring solutions similar to p_A, it is reasonable to replace only parent p_A, rather than both parents (line 7), in order to better maintain population diversity. Here, the offspring solution that replaces parent p_A is selected according to a given evaluation function. The most straightforward evaluation function would be the tour length, but we employ an alternative evaluation function in order to maintain population diversity in a positive manner. We describe this evaluation function in §4.

Algorithm 1 (Procedure GA( ))

```
 1. $\{x_1, \ldots, x_{N_{\text{pop}}}\} := \textsc{Generate\_Initial\_Pop}(\,)$;
 2. repeat
 3.     $r(\cdot) :=$ a random permutation of $1, \ldots, N_{\text{pop}}$;
 4.     for $i := 1$ to $N_{\text{pop}}$ do
 5.         $p_A := x_{r(i)}$, $p_B := x_{r(i+1)}$;
 6.         $\{c_1, \ldots, c_{N_{\text{ch}}}\} := \textsc{Crossover}(p_A, p_B)$;
 7.         $x_{r(i)} := \textsc{Select\_Best}(c_1, \ldots, c_{N_{\text{ch}}}, p_A)$;
 8.     end for
 9. until a termination condition is satisfied
10. return the best individual in the population.
```

3. Edge Assembly Crossover

In §3.1, we first present the EAX algorithm along with the description of the original and localized versions of EAX. Several global versions of EAX are described in §3.2. Because of space limitations, details of the efficient implementation techniques for the localized version of EAX are presented in the online supplement of this paper.

3.1. EAX Algorithm

Let a TSP be defined on a complete graph (V, E) consisting of N vertices and a function d that returns the length of every edge. Let p_A and p_B be two tours selected as a pair of parents. We define E_A and E_B as sets of edges consisting of p_A and p_B, respectively. We also define E_C as a set of edges consisting of an offspring solution or intermediate solution. EAX generates offspring solutions through the following six steps (EAX algorithm), which are also illustrated in Figure 1. Note that all versions of EAX are based on the EAX algorithm described next.

EAX algorithm

1. Let G_{AB} be the undirected multigraph (overlapped edges, one from E_A and one from E_B, are distinguished) defined as $G_{AB} = (V, E_A \cup E_B)$.

2. Partition all edges of G_{AB} into *AB-cycles*, where an *AB-cycle* is defined as a cycle in G_{AB}, such that edges of E_A and edges of E_B are alternately linked. Here, partition of the edges into *AB-cycles* is always possible, because for any vertex in G_{AB} the number of incident edges of E_A is equal to that of E_B. However, the partition is not uniquely determined and we partition the edges randomly into *AB-cycles* in the following way.

The procedure is started by randomly selecting a vertex. Starting from the selected vertex, trace the edges of E_A and E_B in G_{AB} in turn until an *AB-cycle* is found in the traced path, where the edge to be traced next is randomly selected (if two candidates exist) and the traced edges are immediately removed from G_{AB}. If an *AB-cycle* is found in the traced path (a portion of the traced path including the end may form an *AB-cycle*), store it and remove the edges constituting it from the traced path. If the current traced path is not empty, start the tracing process again from the end of the current traced path. Otherwise, start the tracing process by randomly selecting a vertex from among those linked by at least one edge in G_{AB}. If there is no edge in G_{AB}, iterations of the tracing process are terminated.

3. Construct an *E-set* by selecting *AB-cycles* according to a given selection strategy, where an *E-set* is defined as the union of *AB-cycles*. Note that this selection strategy determines a version of EAX.

4. Generate an intermediate solution from p_A by removing the edges of E_A and adding the edges of E_B in the *E-set*, i.e., generate an intermediate solution by $E_C := (E_A \backslash (E\text{-}set \cap E_A)) \cup (E\text{-}set \cap E_B)$. An intermediate solution consists of one or more subtours.

5. Generate an offspring solution by connecting all subtours into a tour (details are given later).

6. If a further offspring solution is generated, then go to Step 3. Otherwise, terminate the procedure.

We define the size of an *AB-cycle* as the number of edges of E_A (or E_B) included in it. Note that some of

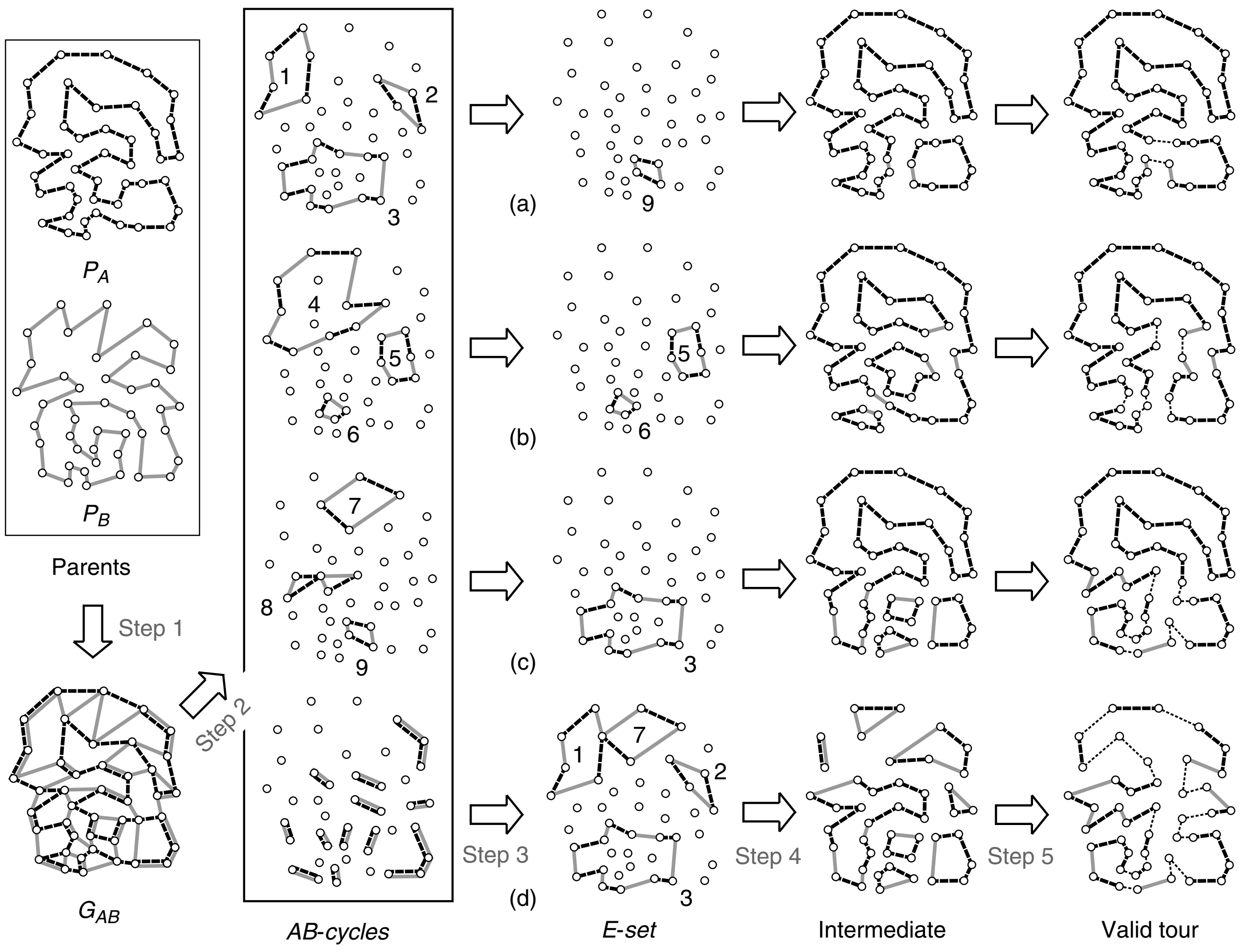

Figure 1 Illustration of the EAX Algorithm

the *AB-cycles* might consist of two overlapping edges, one from E_A and one from E_B. We call such an *AB-cycle* "ineffective" because the inclusion of ineffective *AB-cycles* in an *E-set* does not affect the resulting intermediate solution. We call an *AB-cycle* consisting of more than four edges "effective" (nine effective *AB-cycles* and 15 ineffective *AB-cycles* are formed in the example of Figure 1). In Step 3, we select only effective *AB-cycles* for constructing *E-sets* unless stated otherwise. We define the size of an *E-set* as the number of edges of E_A (or E_B) included in it.

According to the definition of an *E-set* and the procedure in Step 4, EAX generates an intermediate solution from E_A by replacing edges with the same number of edges selected from E_B, under the condition that every vertex is linked by just two edges. An intermediate solution therefore consists of one or more subtours, as illustrated in Figure 1.

In Step 5, an intermediate solution is modified into a tour by connecting the subtours. Let the subtours in the intermediate solution be indexed by $\{1, \ldots, m\}$, where m is the number of subtours. We define $w_{ll'}$ as the increase in the sum of the edge lengths of the intermediate solution when two subtours l and l' are connected. Here, two subtours are connected by removing one edge from each subtour and adding two edges to connect them in the best way possible. Roughly speaking, we connect the subtours into a tour according to the minimum spanning tree in a graph defined by m nodes and weights $w_{ll'}$ $(l, l' \in \{1, \ldots, m\})$. Step 5 can therefore be performed in a greedy manner, described as follows.

Step 5 of EAX algorithm

5-1. Let m be the number of subtours and A_l $(l = 1, \ldots, m)$ the number of vertices in subtour l. Compute m and A_l $(l = 1, \ldots, m)$ from E_C.

5-2. Let r be the index of the smallest subtour (the subtour consisting of the smallest number of edges). Let U be a set of edges in this subtour.

5-3. Find 4-tuples of edges $\{e^*, e'^*, e''^*, e'''^*\} = \arg\max_{e \in U, e' \in E_C \setminus U}\{-d(e) - d(e') + d(e'') + d(e''')\}$, where e and e' denote two edges to be removed, and e'' and e''' denote two edges to be added to connect the breakpoints.

5-4. Let r' be the index of the subtour that includes edge e'^{*}. Connect subtours r and r' by

$$E_C := (E_C \setminus \{e^*, e'^*\}) \cup \{e''^*, e'''^*\}.$$

5-5. Subtract 1 from m (re-index the subtours appropriately). If m equals 1, then terminate the procedure. Otherwise, recompute A_l $(l = 1, \ldots, m)$ and go to Step 5-2.

At each iteration of Step 5, the smallest subtour is selected in Step 5-2 as the one to connect next, in order to reduce the computational effort of Step 5-3. However, performing an exhaustive search in Step 5-3 is too time consuming, and we restrict the search to promising pairs of e and e'. More precisely, for each $e \in U$, the candidates of e' are restricted to a set of edges that satisfy the following condition: at least one end of e' is among the N_{near} closest to either end of e ($N_{\text{near}} = 10$ in our experiments).

EAX can generate various intermediate solutions, depending on the combination of *AB-cycles* selected in Step 3 for constructing *E-sets*. We can construct different versions of EAX by using different selection strategies of *AB-cycles*. For the original EAX (Nagata and Kobayashi 1997), we proposed a simple selection strategy, which we call "random strategy" in this paper. In addition, we proposed another simple selection strategy (Nagata 2006a), which is called the "single strategy" in this paper. The two selection strategies of *AB-cycles* are described below.

Random strategy. Select *AB-cycles* randomly with a probability of 0.5 for each.

Single strategy. Select a single *AB-cycle* randomly without overlapping the previous selections.

The random strategy typically forms an *E-set* like *E-set* (d) in Figure 1, and the resulting intermediate solution tends to contain edges of E_A and edges of E_B equally. In contrast, the single strategy typically forms an *E-set* like *E-set* (a) in Figure 1, and the resulting intermediate solution tends to be similar to p_A. In this paper, we apply the single strategy to the construction of a localized version of EAX.

3.2. Global Versions of EAX

In this subsection, we present three selection strategies of *AB-cycles* for constructing global versions of EAX. Here, the size of *E-sets* generated by a global version of EAX should be greater than that by the localized version of EAX (EAX with the single strategy). However, the increase of the *E-set* size typically increases the number of subtours in intermediate solutions (see the *E-sets* and the resulting intermediate solutions in Figure 1 for an illustration), which degrades the capability to generate good offspring solutions that improve parent solutions when parent solutions are very high-quality tours. This is because the new edges introduced in Step 5 of the EAX algorithm frequently degrade the quality of offspring solutions in this situation. So, it is preferable to decrease the number of subtours while increasing the *E-set* size. A global version of EAX needs to be designed to satisfy these two competing demands.

3.2.1. *K*-Multiple and Block Strategies. In our preliminary work (Nagata 2006a), the random strategy (i.e., the original EAX) was employed for constructing a global version of EAX. However, compared to the single strategy, the random strategy substantially increases the number of subtours in intermediate solutions. One straightforward approach to reduce the number of subtours is to simply restrict the number of *AB-cycles* constituting an *E-set*, and in fact the following strategy, which we call "*K*-multiple strategy" in this paper, works better than the random strategy. We therefore use EAX with the *K*-multiple strategy as a simple global version of EAX to investigate the baseline performance.

K-multiple strategy. Select K *AB-cycle*s randomly, where K is a given parameter ($K = 5$ in our experiments).

In our preliminary work (Nagata 2006b), we proposed a heuristic selection strategy of *AB-cycles*, called "block strategy," to construct an effective global version of EAX. One basic principle of the block strategy is to select geographically close *AB-cycles* for constructing an *E-set*; the resulting intermediate solution is generated from E_A by replacing a block of edges of E_A with a block of edges of E_B in the same region. In fact, the block strategy is clearly superior to the *K*-multiple strategy. However, a main focus of this paper regarding a global version of EAX is to present a new, more sophisticated one, which we call "block2 strategy," and we present details of the block strategy in the online supplement.

3.2.2. Block2 Strategy. The basic idea of the block2 strategy is to construct an *E-set* by selecting *AB-cycles* so that the resulting intermediate solution consists of relatively few and large segments of E_A and E_B, in order to decrease the number of subtours. Figure 3 shows a typical example of such an *E-set* and the resulting intermediate solution.

To describe the block2 strategy, we first define the following notations for an intermediate solution and *E-set* (see Figure 2 for illustrations).

A-block. A maximal sequence of successive edges of E_A in an intermediate solution.

B-block. A maximal sequence of successive edges of E_B in an intermediate solution.

A-vertex. A vertex linked by two edges of E_A in an intermediate solution, i.e., a vertex linked by no edges in an *E-set*.

B-vertex. A vertex linked by two edges of E_B in an intermediate solution, i.e., a vertex linked by four edges (two of E_A and two of E_B) in an *E-set*.

C-vertex. A vertex linked by one edge of E_A and one edge of E_B in an intermediate solution, i.e., a vertex linked by two edges (one of E_A and one of E_B) in an *E-set*.

A-blocks and B-blocks are separated by C-vertices in an intermediate solution, and the number of C-vertices is even. Let the number of C-vertices be denoted by $\#C$. The number of A-blocks is equal to that of B-blocks and is equal to $\#C/2$. If $\#C = 0$, the corresponding intermediate solution is identical to either p_A or p_B, and we do not regard this tour as either A-block or B-block. If $\#C = 2$, the corresponding intermediate solution is always a tour consisting of one A-block and one B-block. If $\#C = 4$, the corresponding intermediate solution consists of one or two subtours (if two subtours exist, each consists of one A-block and one B-block). In the same way, we can see that if $\#C = 2m$ ($m > 1$), the number of subtours is between 1 and $m - 1$. Therefore, given an *E-set*, we have a rough estimation for the number of subtours in the resulting intermediate solution by computing the value of $\#C$. As we will explain later, we can easily compute the value of $\#C$ for a given *E-set*, whereas computing the number of subtours in the resulting intermediate solution requires considerable computational effort.

An intermediate solution, however, may have "excess" C-vertices, which are potentially eliminated without generating any other new C-vertex by converting edges of E_A into the same edges of E_B. Figure 2 shows an example; the value of $\#C$ is decreased from (a) 16 to (b) 12 by converting four edges of E_A into the same edges of E_B (i.e., the corresponding four ineffective *AB-cycles* are added to the *E-set*). For a given intermediate solution, we should calculate the value of $\#C$ after excluding excess C-vertices in order to gain a better estimation of the number of subtours. Note that excess C-vertices exist in an intermediate solution if and only if an intermediate solution includes an A-block whose edges are all able to be converted into the same edges of E_B. If this is the case, there exists a sequence of ineffective *AB-cycles* overlapping this A-block, and we can eliminate the two C-vertices (both ends of this A-block) by adding this sequence of ineffective *AB-cycles* to the *E-set*.

To eliminate excess C-vertices, we redefine *AB-cycles* in a slightly different way, as illustrated in Figure 3, where each sequence of successive ineffective *AB-cycles* is integrated into one of the adjacent effective *AB-cycles*. By constructing *E-sets* using redefined *AB-cycles*, we can definitely eliminate excess C-vertices. We define the size of a redefined *AB-cycle* as that of the original effective *AB-cycle*. After this, we call a redefined *AB-cycle* simply an *AB-cycle*, unless stated otherwise.

The block2 strategy uses a tabu search procedure to search for combinations of *AB-cycles* that construct *E-sets* with small $\#C$ values. Let the *AB-cycles* be indexed by $1, \ldots, l$. As in the case of the block strategy, the block2 strategy constrains an *E-set* to include a relatively large *AB-cycle* (central *AB-cycle*) as a basic component. The tabu search procedure is outlined below (see Figure 3 for an illustration).

TS for selecting *AB-cycles* (Block2 strategy)

1. At the beginning of the procedure, set the iteration counter $t = 0$, and initialize the tabu list as follows: $\text{Tabu}[i] = 0$ ($i = 0, \ldots, l$). Here, the tabu list is represented as a table.

2. Select a relatively large *AB-cycle* (central *AB-cycle*). An initial *E-set* is formed by selecting the central *AB-cycle* and by further randomly selecting from among the *AB-cycles* having at least one contact point with the central *AB-cycle* and that are smaller than the central *AB-cycle*. Note that the central *AB-cycle* is

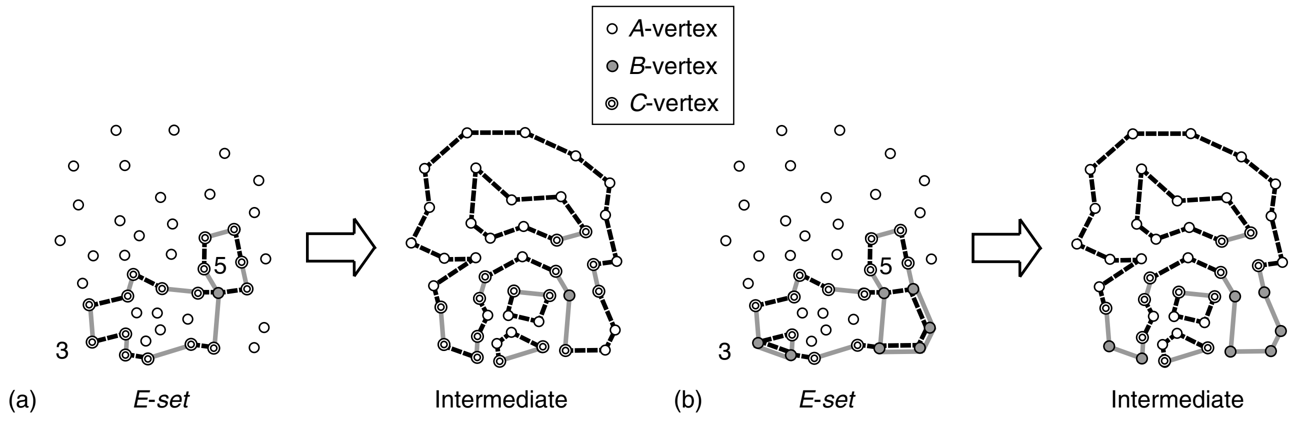

Figure 2 Illustration of A-Blocks, B-Blocks, A-Vertices, B-Vertices, and C-Vertices
Note. The parent solutions and the set of *AB-cycles* are the same as those in Figure 1.

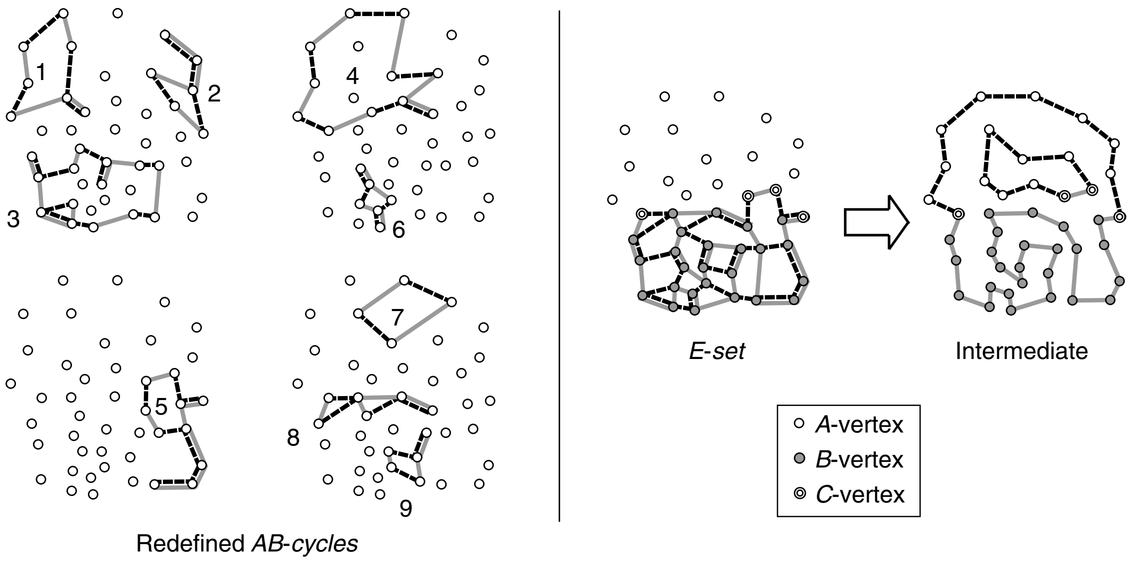

Figure 3 Illustration of the Block2 Strategy
Notes. The parent solutions and the set of original *AB-cycles* are the same as those in Figure 1. *AB-cycle* 3 is a central *AB-cycle*.

selected in descending order of the size when more than one offspring solution is generated.

3. At each iteration, the current *E-set* is moved to the one that has the lowest value of #*C* in the neighborhood and is not a tabu solution. The neighborhood is defined as the set of *E-sets* obtained from the current *E-set* by adding one nonselected *AB-cycle*, or by removing one selected *AB-cycle* other than the central *AB-cycle*. Here, tabu solutions are defined by the attribute of moves; if $t \le \text{Tabu}[i]$, an *E-set* obtained by adding or removing *AB-cycle* i is regarded as a tabu solution. We use a standard aspiration criterion; a tabu solution leading to an *E-set* that is better than the best one found so far is always accepted. After the current *E-set* is moved where *AB-cycle* i is added or removed, the tabu list is updated as follows: $\text{Tabu}[i] := t + \text{Rand}[0, T_{\max}]$, where $T_{\max}$ is a parameter (10 in the experiments) and $\text{Rand}[0, T_{\max}]$ randomly returns an integer value between 0 and $T_{\max}$. Then t is incremented by one.

4. Iterations are terminated when the best *E-set* (with the lowest value of #*C*) found so far is not updated during successive LS_{stag} (20 in the experiments) iterations. Then return the best *E-set*.

At each iteration, the #*C* values of the *E-sets* in the neighborhood can be efficiently computed as described below. Let n_i $(i \in \{1, \dots, l\})$ be the number of C-vertices when an *E-set* consists only of *AB-cycle* i, and let w_{ij} $(i, j \in \{1, \dots, l\})$ be the number of contact points between *AB-cycles* i and j (w_{ii} is defined as zero). For example, $n_5 = 6$ and $w_{35} = 2$ in the example illustrated in Figure 3. These values are computed in advance after forming all redefined *AB-cycles*. Let *E-set* also denote a set of the indices of the selected *AB-cycles* in the current *E-set*. Then, #*C* is calculated as $\#C = \sum_{i \in E\text{-set}} n_i - \sum_{i,j\ (i<j) \in E\text{-set}} w_{ij}$. When *AB-cycle* j is added to the current *E-set*, the difference in #*C*, denoted by $\Delta(\#C)$, is calculated as $\Delta(\#C) = n_j - 2\sum_{i \in E\text{-set}} w_{ij}$. In the same way, when *AB-cycle* j is removed from the current *E-set*, $\Delta(\#C)$ is calculated as $\Delta(\#C) = -n_j + 2\sum_{i \in E\text{-set} \setminus \{j\}} w_{ij}$. For each *E-set* in the neighborhood, we can compute $\Delta(\#C)$ in $O(1)$ time by introducing additional variables W_j $(j \in \{1, \dots, l\})$ storing $\sum_{i \in E\text{-set}} w_{ij}$. These values must be updated according to the definition each time the current *E-set* is moved. Note that we introduce the redefined *AB-cycles* only for defining the values of n_i and w_{ij}, and we can construct an *E-set* only from the original effective *AB-cycles* after a set of the indices of the selected redefined *AB-cycles* is determined.

4. Diversity Preservation

This section presents an effective and efficient way of preserving population diversity. We use the GA framework given in Algorithm 1, where N_{ch} offspring solutions are generated from each of the parent pairs, denoted by p_A and p_B, and a best solution among the generated offspring solutions replaces the population member selected as p_A only if it improves p_A in terms of a given evaluation function. In our preliminary work (Nagata 2006a), we proposed an evaluation function to maintain population diversity in a positive manner at a negligible computational cost.

4.1. Evaluation Function

We employ the edge entropy measure to evaluate population diversity. Maekawa et al. (1996) proposed this measure in a GA framework called thermodynamical GA (TDGA). For each vertex i $(i=1,\dots,N)$, let $\{P_i\}=P_{ij}$ $(j=1,\dots,N)$ be the distribution of the vertices linked to vertex i in the population, i.e., the number of individuals including edge (i,j) in the population divided by $2N_{\text{pop}}$. The edge entropy H of the population is defined as

$$H_i=-\sum_{j=1}^{N}P_{ij}\log(P_{ij})\quad(i=1,\dots,N),\tag{1}$$

$$H=\sum_{i=1}^{N}H_i=-\sum_{i=1}^{N}\sum_{j=1}^{N}P_{ij}\log(P_{ij}).\tag{2}$$

Here, H_i $(i=1,\dots,N)$ represents the entropy with respect to the distribution $\{P_i\}$, and H is the approximate entropy of the population, defined under the assumption that the distributions $\{P_i\}$ $(i=1,\dots,N)$ are independent of each other.

The TDGA framework was introduced in analogy to free energy in physics. Let L be the average tour length of the population and T a positive coefficient controlling the balance between L and H. Note that H tends to decrease (i.e., loss of the population diversity) as L decreases because the fitness landscapes of the TSP typically have a big-valley structure (Boese et al. 1994). At each generation in TDGA, the population members generate offspring solutions by applying crossover and mutation operators. Then, the members of the next population are selected from the union of the current population members and offspring solutions such that the value of $L-TH$ for the next population is approximately minimized. However, the TDGA framework requires a high computational cost in selecting individuals for the next population, and the value of T must be carefully adjusted (T is gradually decreased).

In our approach, the selection of offspring solutions can be performed at a negligible computational cost, and the evaluation function does not include a parameter that needs to be adjusted. Let y be an offspring solution generated from parents p_A and p_B. Let $\Delta L(y)$ and $\Delta H(y)$ be the differences in L and H, respectively, when $x_{r(i)}$ (the population member selected as p_A) is replaced by an offspring solution y. We use the term "entropy-preserving selection" to refer to the selection method of offspring solutions based on the following evaluation function, where the offspring solution with the largest evaluation value is selected to replace $x_{r(i)}$ only if the evaluation value is greater than zero:

$$Eval_{\text{Entropy}}(y)=\begin{cases}\dfrac{\Delta L(y)}{\Delta H(y)} & (\Delta L<0,\ \Delta H<0)\\[2ex] -\dfrac{\Delta L(y)}{\epsilon} & (\Delta L<0,\ \Delta H\ge 0),\\[2ex] -\Delta L(y) & (\Delta L\ge 0)\end{cases}\tag{3}$$

where ϵ is a sufficiently small positive number. If the tour length of an offspring solution y is greater than that of p_A (i.e., $\Delta L(y)\ge 0$ and thus $Eval_{\text{Entropy}}(y)<0$), this offspring solution never replaces $x_{r(i)}$, even if the population diversity is increased. This criterion serves to prevent the population from not converging. When $\Delta L(y)<0$, $\Delta H(y)$ is frequently less than zero (i.e., loss of the population diversity), as mentioned above. In this case, an offspring solution y is evaluated by $\Delta L(y)/\Delta H(y)$, which is the amount of improvement in the average tour length per unit loss of the population diversity. If there is at least one offspring solution such that $\Delta L(y)<0$ and $\Delta H(y)\ge 0$, we can improve the average tour length by the replacement, without loss of the population diversity. In this case, the offspring solution with the smallest value of $\Delta L(y)$ is selected from among these offspring solutions.

4.2. Efficient Computation of $\Delta H(y)$

For every offspring solution y, we must compute $\Delta L(y)$ and $\Delta H(y)$ to obtain the value of $Eval_{\text{Entropy}}(y)$. When the localized version of EAX is used in the GA framework given in Algorithm 1, we can compute both values at a negligible computational cost because an offspring solution is formed from p_A by exchanging a relatively small number of edges. Here, we define E_{ad} and E_{re} as sets of the added and removed edges, respectively, when an offspring solution y is generated from p_A. Then, $\Delta L(y)$ is computed as follows:

$$\Delta L(y)=\frac{1}{N_{\text{pop}}}\left\{\sum_{e\in E_{\text{ad}}}d(e)-\sum_{e\in E_{\text{re}}}d(e)\right\}.\tag{4}$$

For the current population consisting of N_{pop} individuals, let $F(e)$ $(e\in E)$ be the number of individuals having edge e. The population diversity H can be represented as follows:

$$H=-\sum_{e\in E}F(e)\Big/N_{\text{pop}}\log(F(e)/N_{\text{pop}}).\tag{5}$$

Then, $\Delta H(y)$ is computed as follows:

$$\begin{aligned}\Delta H(y)=&\sum_{e\in E_{\text{re}}}\{A(F(e)-1)-A(F(e))\}\\&-\sum_{e\in E_{\text{ad}}}\{A(F(e))-A(F(e)+1)\},\end{aligned}\tag{6}$$

where $A(x)=-x/N_{\text{pop}}\log(x/N_{\text{pop}})$.

When the localized version of EAX is used, $|E_{\text{ad}}|$ and $|E_{\text{re}}|$ are much smaller than N, and we can compute the value of $Eval_{\text{Entropy}}(y)$ at a negligible computational cost. Note that we need to compute $F(e)$ $(e\in E)$ once for the initial population. Each time $x_{r(i)}$ is replaced by a selected offspring solution, it can be updated as follows: $F(e):=F(e)+1$ $(e\in E_{\text{ad}})$ and $F(e):=F(e)-1$ $(e\in E_{\text{re}})$.

4.3. Other Evaluation Functions

To investigate the impact of the proposed evaluation function, we also test two other evaluation functions for comparison. The most straightforward evaluation function would be the tour length, and we use the term "greedy selection" to refer to the selection method of offspring solutions based on the following evaluation function, where the offspring solution with the largest evaluation value is selected to replace $x_{r(i)}$ only if the evaluation value is greater than zero:

$$Eval_{\text{Greedy}}(y) = -\Delta L(y). \tag{7}$$

To investigate the impact of the use of the edge entropy measure for evaluating population diversity, we also test another population diversity measure D. This is defined as the average distance between two solutions in the population, where the distance between two solutions is defined as the number of edges of one solution that do not exist in the other. In fact, D can be computed as follows (the constant factor is ignored):

$$D \propto NN_{\text{pop}}^2 - \sum_{e\in E} F(e)^2. \tag{8}$$

In the same way, $\Delta D(y)$ can be computed at a negligible computational cost as follows (the constant factor is ignored):

$$\Delta D(y) \propto \sum_{e\in E_{\text{re}}} (F(e)-1) - \sum_{e\in E_{\text{ad}}} F(e). \tag{9}$$

By comparing Equations (7) and (9), we can see that one important feature of the edge entropy measure H is the sensitivity to the change of rare edges in the population. On the other hand, the average distance D is not sensitive to the change of rare edges in the population. In the experiments, we also test the evaluation function Equation (4) but with the different population diversity measure D, i.e., D is used instead of H. We use the term "distance-preserving selection" to refer to the selection method of offspring solutions based on this evaluation function.

5. Computational Experiments

The proposed GA was tested on 57 instances with up to 85,900 cities (small instances were ignored) selected from well-known, widely used benchmark sets for the TSP:TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95), National TSP benchmarks (http://www.tsp.gatech.edu/world/countries.html), and VLSI TSP benchmarks (http://www.tsp.gatech.edu/world/countries.html). The number of cities for every instance is indicated by the instance name.

The proposed GA was implemented in C++ and the program code was compiled using GNU gcc compiler. The program code is available in the online supplement. We executed the GA in a virtual machine environment on a cluster with Intel Xeon 2.93 GHz nodes, i.e., a job is executed on a single core, but multiple jobs may be executed in the same node and conflict with each other. However, a program execution time varied substantially, depending on the existence of other jobs in the same node. So, we measured the CPU time by executing the GA just once in an occupied node (10 trials were performed in the virtual machine environment to obtain results).

In this section, we first describe several configurations for the GA in §5.1 and then analyze the impact of the proposed enhancements on a selected group of the instances in §5.2. Detailed results of the GA using all enhancements on all 57 instances are presented in §5.3 along with a comparison to LKH-2. Because of space limitations, we report in the online supplement our results on the 100,000-city Mona Lisa TSP challenge as well as on five larger instances with up to 200,000 cities.

5.1. Configurations of the GA

We apply our GA (Algorithm 1) using several different configurations to analyze the impact of the proposed enhancements on the performance. One configuration is determined by selecting one strategy from each of the items listed below. For each item, the default strategy corresponds to each of the proposed enhancements, and other strategies are also tested for comparison. As for the default parameter values for N_{pop} and N_{ch}, we determined them through preliminary experiments. We refer to the GA using the default configuration "default GA."

- Version of EAX in stage I: EAX with the single strategy (default). Alternatively, EAX with the five-multiple strategy and EAX with the random strategy are used.
- Version of EAX in stage II: EAX with the block2 strategy (default). Alternatively, EAX with the five-multiple strategy and EAX with the block strategy are used.
- Selection method of offspring solutions: entropy-preserving selection (default). Alternatively, greedy selection and distance-preserving selection are used.
- Population size (N_{pop}): 300 (default). Alternatively, set to 600 if greedy selection is used.
- Number of offspring solutions (N_{ch}): 30 (default). Alternatively, set to 5, 10, and 100.

5.2. Impact of the Proposed Enhancements

We apply the GA using each configuration 10 times to a selected group of instances with sizes ranging from 4,000 to 25,000 (four instances are selected from each of TSPLIB, National TSPs, and VLSI TSPs) in order to save space and to avoid numerous experiments. Results are presented in Tables 1–3 in the following format: the instance name (instance), the

Table 1 Effects of Localization of EAX on the Performance

		Random strategy				Five-Multiple strategy				Single strategy			
Instance	Optimum	Su	A-Err	Gen	Time	Su	A-Err	Gen	Time	Su	A-Err	Gen	Time
fnl4461	182,566	0	0.01835	145	960	9	0.00022	647	526	10	0.00000	797	225
fi10639	520,527	0	0.04255	207	4,576	5	0.00037	2,036	2,376	7	0.00006	2,139	983
usa13509	19,982,859	0	0.05946	246	7,381	3	0.00023	2,668	3,891	4	0.00028	2,831	1,538
xvb13584	(37,083)	0	0.05178	196	4,772	7	0.00081	1,468	1,764	9	0.00027	2,017	700
d15112	1,573,084	0	0.08759	270	15,518	2	0.00024	2,880	6,990	8	0.00004	4,024	2,930
it16862	557,315	0	0.06634	260	14,266	1	0.00063	3,084	3,916	2	0.00052	3,733	2,259
pjh17845	(48,092)	0	0.04637	214	7,948	1	0.00270	1,935	2,740	3	0.00187	2,371	1,096
d18512	645,238	0	0.10010	295	20,837	5	0.00020	3,870	8,083	8	0.00009	5,006	4,049
ido21215	(63,517)	0	0.05022	236	12,216	6	0.00142	2,512	3,988	6	0.00094	3,470	2,001
vm22775	569,288	0	0.08579	300	33,244	0	0.00165	3,853	6,413	1	0.00116	5,153	2,798
xrh24104	(69,294)	0	0.04618	244	14,228	5	0.00101	2,419	3,928	9	0.00014	3,140	1,763
sw24978	855,597	0	0.07284	305	31,763	0	0.00133	5,212	8,574	4	0.00028	6,599	3,718

Note. The three versions of EAX are used in stage I (other settings follow the default configuration).

Table 2 Effects of Global Versions of EAX on the Performance

		Stage I (single)				Stage II (five-multiple)				Stage II (block2)			
Instance	Optimum	Su	A-Err	Gen	Time	Su	A-Err	Gen	Time	Su	A-Err	Gen	Time
fnl4461	182,566	0	0.00142	744	210	10	0.00000	65	12	10	0.00000	53	15
fi10639	520,527	0	0.00767	2,076	923	0	0.00200	127	85	7	0.00006	63	60
usa13509	19,982,859	0	0.00415	2,769	1,476	0	0.00057	160	465	4	0.00028	62	62
xvb13584	(37,083)	0	0.01160	1,956	609	1	0.00701	87	185	9	0.00027	61	91
d15112	1,573,084	0	0.00668	3,954	2,815	0	0.00198	227	913	8	0.00004	70	115
it16862	557,315	0	0.00942	3,665	2,145	0	0.00266	175	356	2	0.00052	68	114
pjh17845	(48,092)	0	0.01060	2,302	976	0	0.00707	75	112	3	0.00187	69	120
d18512	645,238	0	0.00801	4,935	3,939	0	0.00262	196	709	8	0.00009	71	110
ido21215	(63,517)	0	0.01448	3,395	1,803	0	0.00992	77	140	6	0.00094	75	198
vm22775	569,288	0	0.00530	5,078	2,489	0	0.00416	95	215	1	0.00116	75	309
xrh24104	(69,294)	0	0.00592	3,081	1,579	0	0.00419	68	129	9	0.00014	59	184
sw24978	855,597	0	0.01009	6,509	3,425	0	0.00553	172	346	4	0.00028	90	293

Note. Stage I is performed using EAX with the single strategy, and stage II is then performed using the two global versions of EAX (other settings follow the default configuration).

Table 3 Effects of the Diversity-Preserving Selections on the Performance

	Greedy $N_{pop} = 600$ $N_{ch} = 30$			Distance-preserving $N_{pop} = 300$ $N_{ch} = 30$			Entropy-preserving $N_{pop} = 300$ $N_{ch} = 5$			Entropy-preserving $N_{pop} = 300$ $N_{ch} = 10$			Entropy-preserving $N_{pop} = 300$ $N_{ch} = 30$			Entropy-preserving $N_{pop} = 300$ $N_{ch} = 100$		
Instance	Su	A-Err	Gen	Su	A-Err	Gen	Su	A-Err	Gen	Su	A-Err	Gen	Su	A-Err	Gen	Su	A-Err	Gen
fnl4461	3	0.00279	399	9	0.00022	825	6	0.00236	1,683	9	0.00022	1,134	10	0.00000	797	10	0.00000	813
fi10639	1	0.00365	986	2	0.00038	2,284	0	0.00169	4,635	3	0.00079	3,214	7	0.00006	2,139	7	0.00017	2,032
usa13509	0	0.00246	1,098	1	0.00083	3,061	0	0.00162	6,163	1	0.00117	3,803	4	0.00028	2,831	5	0.00016	2,762
xvb13584	1	0.00755	857	4	0.00189	2,092	5	0.00351	4,630	3	0.00270	2,828	9	0.00027	2,017	10	0.00000	1,732
d15112	0	0.00113	1,791	6	0.00025	4,193	0	0.00105	8,487	0	0.00055	5,796	8	0.00004	4,024	5	0.00004	3,199
it16862	0	0.00587	1,702	0	0.00126	3,986	0	0.00248	8,883	0	0.00217	5,920	2	0.00052	3,733	4	0.00045	3,366
pjh17845	0	0.00749	1,026	4	0.00125	2,474	2	0.00291	5,457	4	0.00125	3,327	3	0.00187	2,371	8	0.00042	2,229
d18512	0	0.00411	2,143	5	0.00015	5,373	0	0.00169	9,699	2	0.00050	6,317	8	0.00009	5,006	9	0.00002	3,972
ido21215	0	0.00803	1,648	6	0.00094	3,354	4	0.00378	7,797	3	0.00394	4,615	6	0.00094	3,470	8	0.00031	2,900
vm22775	0	0.00824	2,478	0	0.00218	5,333	0	0.00358	15,761	0	0.00230	8,705	1	0.00116	5,153	6	0.00032	4,292
xrh24104	0	0.00447	1,412	5	0.00087	3,289	1	0.00245	6,831	4	0.00159	4,263	9	0.00014	3,140	7	0.00043	2,832
sw24978	0	0.00754	2,880	1	0.00064	6,870	0	0.00353	15,949	0	0.00366	9,169	4	0.00028	6,599	3	0.00034	5,178

Note. The three selection methods are used with several values of N_{ch} (other settings follow the default configuration).

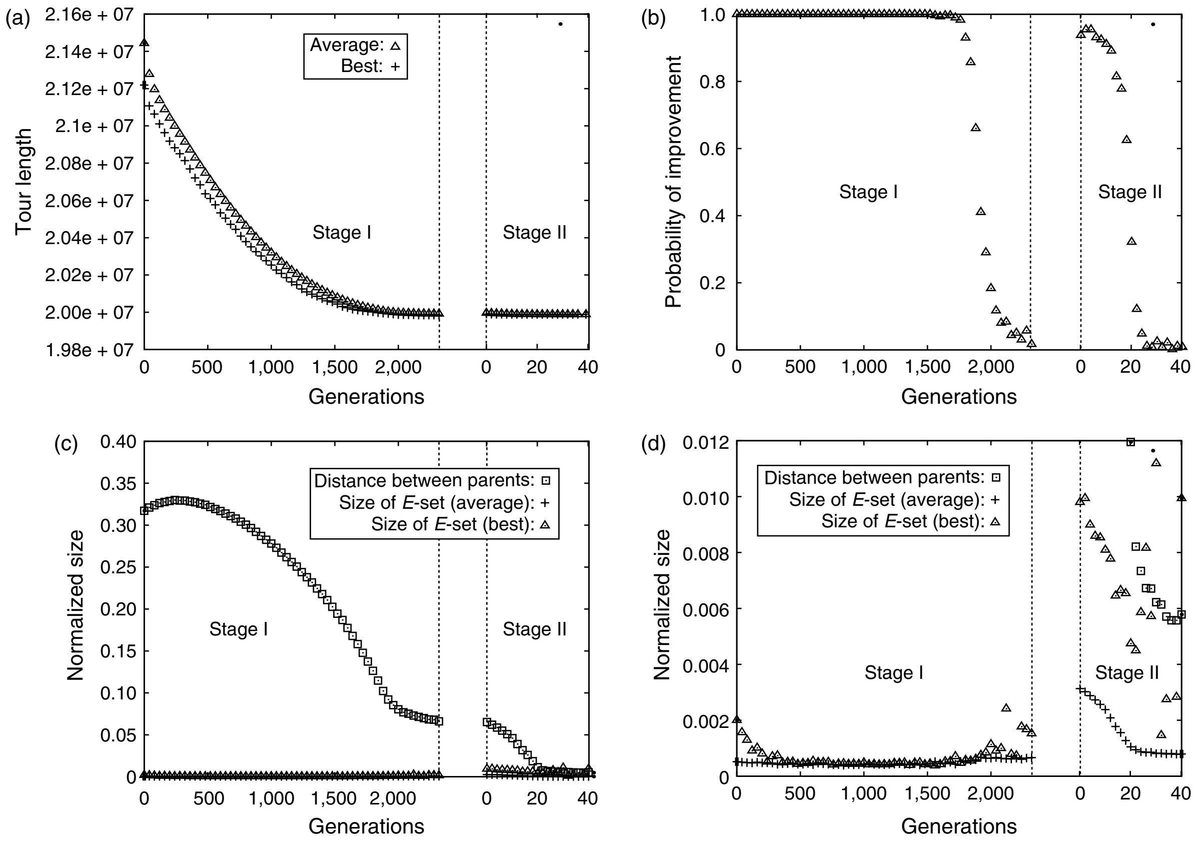

Figure 4 **Typical Behavior of the Default GA (usa13509)**

Notes. (a) The average and best tour lengths in the population. (b) The probability of generating at least one offspring solution that improves p_A for each parent pair. (c) The average size of *E-sets* and the size of an *E-set* that generates the best offspring solution (considered only if it improves p_A), for each parent pair. The distance between the parents is also shown. These values are normalized such that the size (or distance) of N is transformed to 1. (d) An enlarged view of (c) with respect to the y-axis.

number of runs that succeeded in finding the optimal or best-known solution (no better solution was found) over 10 runs (Su), the average percentage excess with respect to the optimal solutions (A-Err), the average number of generations (Gen), and the computation time for a single run in seconds (Time).

5.2.1. Behavior of the Default GA. First, we analyze the behavior of the default GA. In Figure 4, the four graphs depict typical behavior of the default GA for a single run on instance usa13509 ($N = 13{,}509$). These graphs show several feature amounts averaged over each generation where results of stages I and II are separately displayed at every 40 and two generations, respectively (note the different x-axis scales). Note that stage II terminates at generation 72, but the results after generation 40 are omitted.

Graph (a) shows that most of the search is performed in stage I and the final phase of the search is performed in stage II; the number of generations of stage I is more than 20 times greater than that of stage II (see Table 2 for other instances). Graph (b) shows that EAX with the block2 strategy can effectively generate offspring solutions that improve parent p_A at the beginning of stage II after EAX with the single strategy can no longer effectively improve p_A. Graphs (c) and (d) show that EAX with the single strategy is considerably localized in stage I; the average normalized size of the *E-sets* at each generation is 0.0004–0.0007 (5–9 in terms of *E-set* size) throughout stage I. Note that, given a pair of parents, the total size of all effective *AB-cycles* is equal to or slightly greater than the distance between the parents (the number of edges of one parent that do not exist in the other). Therefore, if the random strategy is used, the size of *E-sets* would be about half the distance between the parents.

We can see that the size of an *E-set* constructed by EAX with the single strategy tends to be greater at the end of stage I when a best offspring solution (considered only if it improves p_A) among 30 offspring solutions is generated. This indicates that the size of the *E-sets* constructed by the single strategy is too small to improve parent p_A in this situation. We can also see that the block2 strategy increases the size of the *E-sets* in stage II, and EAX with the block2 strategy effectively improves parent p_A that are no longer improved by EAX with the single strategy.

5.2.2. Impact of Localization of EAX. We analyze the impact of localization of EAX. We compare the GAs with three different configurations where EAX with the single strategy, EAX with the five-multiple strategy, and EAX with the random strategy are used, respectively, in stage I (other settings follow the default configuration).

Table 1 shows the results. The GA with the single strategy is superior to the GA with the random strategy in terms of the computation time, even though the former requires a greater number of generations. This is because EAX with the single strategy can generate offspring solutions much more efficiently than does EAX with the random strategy thanks to localization of EAX together with the efficient implementation. The GA with the five-multiple strategy is also faster than the GA with the random strategy for the same reason but is slower than the GA with the single strategy because EAX with the single strategy is more localized.

Another important advantage of localization of EAX (coupled with Algorithm 1) is the ability of maintaining population diversity. That is because the larger the *E-set* size is, the more likely the two population members selected as the parents (p_A and p_B) become similar to each other after p_A is replaced with an offspring solution. In fact, Table 1 shows that the solution quality of the GA improves in the following order: random, five-multiple, and single strategies. Note that we observed that the use of the block2 strategy in stage I also degraded the solution quality of the default GA.

5.2.3. Impact of the Block2 Strategy. We compare the three global versions of EAX (five-multiple, block, and block2 strategies) in order to investigate their impact on the performance. For each instance, stage I is performed using the default configuration, and stage II is then performed using each of the three global versions of EAX (other settings follow the default configuration). For the three global versions of EAX, stage II starts from the same set of 10 populations obtained by performing stage I 10 times.

Table 2 lists the results of stage I and of stage II using two global versions of EAX (five-multiple and block2). We present a comparison between the block and block2 strategies in the online supplement because it is somewhat lengthy. The table shows that high-quality solutions are obtained through only stage I, but no optimal (or best-known) solution is found for any instance. The solution quality of stage I is clearly improved by further performing stage II with the five-multiple strategy, but again no optimal (or best-known) solution is found for any instance except for fnl4461 and xvb13584. We can see that the block2 strategy is more powerful than the five-multiple strategy, finding optimal (or best-known) solutions for all instances.

In regard to the computation time, stage II with the block2 strategy is more than 10 times faster than stage I in most instances because stage II is performed at the final phase of the search as indicated by the number of generations (Gen) in the table.

Next, we analyze the behavior of the three global versions of EAX. Figure 5 shows plots of the number of C-vertices (excess C-vertices are eliminated) versus the size of *E-set* for offspring solutions generated by each of the three global versions of EAX at the first generation in stage II on usa13509. Here, results of the localized version of EAX (EAX with the single strategy) are also presented for comparison. Note that to make the graphs more visible, one third of all data points are plotted and they are plotted only if the *E-set* size is an even number. The figure shows that the use of each of the five-multiple, block, and block2 strategies increases the size of *E-set* as compared with the single strategy. As already discussed, increasing the size of *E-set* is essential in stage II for the generation of offspring solutions that improve parent p_A. The figure also shows that the number of C-vertices tends to increase as the size of *E-set* increase when the single, five-multiple, and block strategies are used. We can see that EAX with the block strategy generates offspring solutions that improve parent p_A more frequently than EAX with the five-multiple strategy and that such offspring solutions are typically generated when the number of C-vertices is fairly small. This verifies our hypothesis that decreasing the number of C-vertices improves the ability of generating offspring solutions that improves parent p_A when parent solutions are very high-quality tours. Finally, we can see that the block2 strategy limits the increase of the number of C-vertices associated with the increase of the size of *E-set*. As a consequence, EAX with the block2 strategy can generate offspring solutions that improve parent p_A most effectively.

The impact of the use of the tabu search procedure in the block2 strategy should also be described. In fact, almost the same quality solutions are obtained on the instances with up to 25,000 cities, even if the greedy local search (i.e., set $LS_{\text{stag}} = 1$) is used instead of the tabu search. However, the use of the tabu search sometimes improves the solution quality on larger instances, and this tendency becomes more prominent as the number of vertices increases.

5.2.4. Impact of Entropy-Preserving Selection. We compare the three selection methods (greedy, distance-preserving, and entropy-preserving selections) in order to investigate their impact on the performance. For each instance, we performed the

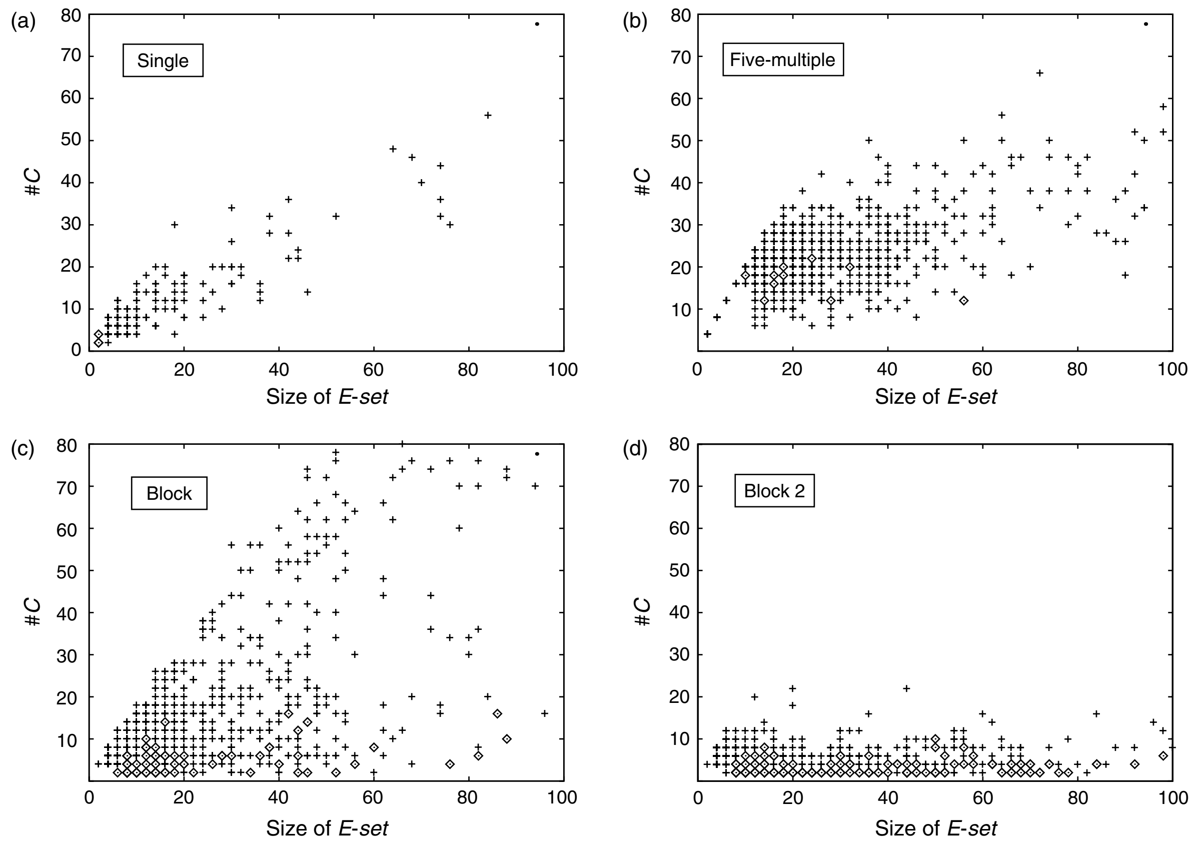

Figure 5 Effects of the Global Versions of EAX (usa13509)

Notes. Plots of the number of C-vertices versus the size of *E-set* for offspring solutions generated by each of the three global versions of EAX (the localized version of EAX is also applied for comparison) at the first generation in stage II. A data point is represented by a diamond if the resulting offspring solution improves parent p_A or by a cross mark if not.

GA using each of the three selection methods (other settings follow the default configuration). Note that when greedy selection is used, the population converges more rapidly, and we thus set N_{pop} to 600 to make the computation time roughly equal to that of the GA using entropy-preserving selection. In addition, we also performed the GA using entropy-preserving selection with N_{ch} set to 5, 10, 30, and 100 because this parameter has a great impact on the performance.

Table 3 shows the results, but omitting the computation time, which is roughly proportional to $N_{\text{pop}} \times N_{\text{ch}} \times \text{Gen}$ (i.e., the total number of offspring solutions generated during a run). First, we make a comparison between the greedy, distance-preserving, and entropy-preserving selections, where $N_{\text{ch}} = 30$ for every selection method. The table shows that the solution quality of the GA improves in the following order: greedy, distance-preserving, and entropy-preserving selections, demonstrating the effectiveness of entropy-preserving selection. Next, we compare the results of entropy-preserving selection with various values of N_{ch}. Table 3 shows that the solution quality tends to improve as the value of N_{ch} increases, although the improvement may be saturated at $N_{\text{ch}} = 30$. The computation time, however, also increases with increasing N_{ch}.

We analyze the effects of the three selection methods and of the different values of N_{ch} on preserving population diversity. Figure 6 shows the behavior of the six GAs for a single run on instance usa13509. Graph (a) depicts the average tour length of the population against the accumulated number of generated offspring solutions. Graph (b) depicts the value of the edge entropy H against the average tour length L of the population at every 20 generation.

We first analyze the impact of the different values of N_{ch} when entropy-preserving selection is used. Graph (b) shows that the population diversity at each solution-quality level of the population is better maintained by increasing N_{ch}. This is because for each pair of parents, our selection model (Algorithm 1) with entropy-preserving selection selects an offspring solution from among the N_{ch} offspring solutions to replace parent p_A such that the amount of improvement in the average tour length ($-\Delta L$) per unit loss of the edge entropy of the population ($-\Delta H$) is maximized. As a result, the solution quality of the GA, in principle,

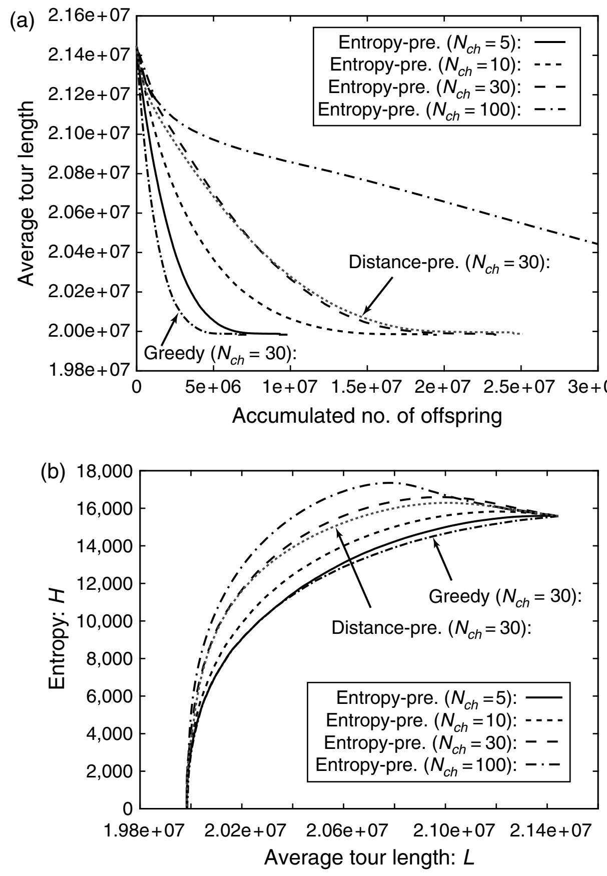

Figure 6 **Effects of the Diversity-Preserving Selections on Maintaining the Population Diversity (usa13509)**

improves with increasing N_{ch}. Graph (a) shows, however, the decreasing speed of the average tour length is slowed down by increasing the value of N_{ch}. Therefore, the parameter N_{ch} should be determined by considering the trade-off between the computation time and the capability of preserving the population diversity; we set $N_{\mathrm{ch}} = 30$ as the default configuration. Graph (b) also shows that entropy-preserving selection is superior to greedy and distance-preserving selections in maintaining the population diversity.

5.3. Detailed Results

We present detailed results of the default GA (we denote it as GA-EAX) for all 57 instances along with a comparison to LKH-2 (Helsgaun 2006, 2009), which is known as one of the most powerful heuristic algorithms for the TSP. For large instances, say more than 25,000 cities, we implemented the GA with the additional techniques to save the amount of memory required to execute it (see Appendix B in the online supplement). We performed GA-EAX and LKH-2 10 times, respectively, for each instance on the same computers.

The program code of LKH-2 (Version 2.0.3), which is implemented in C language, can be downloaded from the website (http://www.akira.ruc.dk/~keld/research/LKH). LKH-2 with the default setting performs an iterated local search with a special perturbation (kicking) strategy. In Helsgaun (2006), the effect of different values for several important parameters were investigated: K (basic K-opt move type) = 4, 5, 6, and 7 (default: 5); patching (control of nonsequential moves) = no patching, simple patching, and full patching (default: no patching); and the number of iterations = 1 and 1,000 (default: N). We performed LKH-2 using each of simple and full patching with several values of K (5, 6, and 7) on the 12 instances used in §5.2, and we found that LKH-2 using full patching with $K = 6$ seems to work best on these instances. Therefore, we present results of LKH-2 using these parameter values (other parameters follow the default setting) where the number of iterations were set to 1,000 and $0.2N$.

Table 4 lists the results for the 57 instances; the sets of the first 11, second 18, and third 28 instances belong to the TSPLIB, National TSPs, and VLSI TSPs, respectively. The first two columns list the instance name (instance) and the optimal tour length (optimal). If an optimal solution is not known, the best-known tour length is listed in parentheses (b-best). For each algorithm, we present the best (B-Err), average (A-Err), and worst (W-Err) results in the form of the percentage excess to the optimal (or best-known) tour length and the computation time for a single run in seconds (Time). LKH-2 requires a preprocessing step, but the computation time presented in the table does not include the preprocessing time. If the best solution out of 10 runs equals to the optimal (or best-known) solution, the corresponding cell in column B-Err shows "=" along with the number of runs that succeeded in finding it. If the best solution out of 10 runs is better than the best-known solution, the value in column B-Err becomes less than zero and is indicated by boldface.

Table 4 shows that GA-EAX is quite an effective approximation algorithm for the TSP, finding optimal or best-known solutions for all instances with up to 39,000 vertices except for pla7397 and pla33810 within a reasonable computation time. Moreover, GA-EAX found five new best solutions, namely, to instances bm33708, ch71009, icx28698, rbz43748, and bna56769. The tour lengths of the new best solutions are listed in the column "Best of 10 runs" of Table 1 in the online supplement.

One drawback of our GA, however, is that it does not seem to work very well on instances in which most vertices are arranged in a lattice pattern. For example, the three instances pla7397, pla33810, and pla85900 in the TSPLIB are VLSI applications, and most vertices are arranged in a lattice pattern. GA-EAX found no optimal solution for these

Table 4 Results of GA-EAX Along with a Comparison to LKH-2

		LKH-2 (10 runs)						GA-EAX (10 runs)			
		No. of iterations = 1,000			No. of iterations = 0.2N			(Default configuration)			
Instance	Optimum	B-Err	A-Err	Time	B-Err	A-Err	Time	B-Err	A-Err	W-Err	Time
fnl4461	182,566	=(7)	0.0008	85	=(7)	0.0009	67	=(10)	0.0000	0.0000	225
rl5915	565,530	0.0007	0.0194	82	0.0007	0.0194	94	=(10)	0.0000	0.0000	153
rl5934	556,045	0.0164	0.0511	188	0.0164	0.0488	208	=(9)	0.0016	0.0164	135
pla7397	23,260,728	=(10)	0.0000	1,804	=(10)	0.0000	2,593	0.0004	0.0044	0.0095	289
rl11849	923,288	0.0014	0.0343	489	=(1)	0.0198	1,066	=(10)	0.0000	0.0000	722
usa13509	19,982,859	0.0022	0.0093	807	0.0010	0.0045	2,178	=(4)	0.0003	0.0008	1,538
brd14051	469,385	0.0009	0.0038	994	0.0009	0.0029	2,522	=(10)	0.0000	0.0000	2,070
d15112	1,573,084	0.0003	0.0029	865	=(1)	0.0026	2,237	=(8)	0.0000	0.0004	2,930
d18512	645,238	0.0009	0.0035	1,449	0.0003	0.0026	5,151	=(8)	0.0001	0.0006	4,049
pla33810	66,048,945	0.0191	0.0266	101,703		Over a time limit		0.0079	0.0123	0.0202	5,637
pla85900	142,382,641	0.0172	0.0460	38,732		Over a time limit		0.0070	0.0109	0.0178	47,933
ca4663	1,290,319	=(1)	0.0042	169	=(1)	0.0045	159	=(9)	0.0003	0.0033	234
pm4951	114,855	=(3)	0.0012	882	=(3)	0.0012	879	=(5)	0.0044	0.0131	195
tz6117	394,718	0.0056	0.0104	14,779	0.0056	0.0094	18,161	=(3)	0.0012	0.0020	247
ar6723	837,479	=(2)	0.0071	219	=(3)	0.0064	295	=(4)	0.0007	0.0012	471
ho7103	177,092	=(5)	0.0015	800	=(8)	0.0005	1,112	=(8)	0.0002	0.0011	464
eg7146	172,386	0.0006	0.6487	976	0.0006	0.6487	1,322	=(9)	0.0001	0.0006	413
ym7663	238,314	=(5)	0.0027	361	=(5)	0.0020	550	=(10)	0.0000	0.0000	418
ei8246	206,171	=(1)	0.0070	287	=(3)	0.0044	475	=(5)	0.0005	0.0015	559
ja9847	491,924	0.0671	0.1036	721	0.0651	0.0941	1,411	=(6)	0.0003	0.0014	751
gr9882	300,899	=(4)	0.0011	751	=(6)	0.0007	1,480	=(4)	0.0007	0.0020	543
kz9976	1,061,881	0.0010	0.0104	377	0.0007	0.0067	706	=(9)	0.0001	0.0005	509
fi10639	520,527	0.0013	0.0087	564	0.0008	0.0061	1,039	=(7)	0.0001	0.0002	983
mo14185	(427,377)	0.0028	0.0122	764	0.0028	0.0062	2,518	=(6)	0.0002	0.0007	1,329
it16862	557,315	0.0043	0.0067	19,832	0.0018	0.0038	67,337	=(2)	0.0005	0.0009	2,259
vm22775	569,288	0.0002	0.0034	8,311	=(2)	0.0020	38,315	=(1)	0.0012	0.0028	2,798
sw24978	855,597	0.0005	0.0077	2,426	0.0001	0.0042	12,597	=(4)	0.0003	0.0007	3,718
bm33708	959,304	0.0019	0.0056	2,890	−**0.0003**	0.0026	19,385	−**0.0011**	−0.0005	0.0000	8,218
ch71009	(4,566,542)	0.0074	0.0169	15,145	0.0018	0.0060	179,190	−**0.0007**	−0.0006	−0.0002	57,582
xmc10150	(28,387)	0.0070	0.0296	403	0.0070	0.0278	978	=(8)	0.0007	0.0035	329
xvb13584	(37,083)	0.0162	0.0348	1,014	0.0027	0.0232	2,506	=(9)	0.0003	0.0027	700
xrb14233	(45,462)	0.0044	0.0244	859	0.0044	0.0178	2,341	=(2)	0.0042	0.0066	599
xia16928	(52,850)	0.0019	0.0189	973	0.0000	0.0144	3,440	=(4)	0.0023	0.0057	924
pjh17845	(48,092)	0.0104	0.0254	1,097	0.0062	0.0160	3,631	=(3)	0.0019	0.0042	1,096
frh19289	(55,798)	0.0108	0.0253	1,228	0.0054	0.0122	6,112	=(7)	0.0005	0.0018	1,339
fnc19402	(59,287)	0.0186	0.0369	1,040	0.0118	0.0273	4,097	=(4)	0.0012	0.0034	1,253
ido21215	(63,517)	0.0173	0.0291	1,613	0.0157	0.0258	7,357	=(6)	0.0009	0.0032	2,001
fma21553	(66,527)	0.0105	0.0347	1,207	0.0045	0.0247	4,782	=(7)	0.0006	0.0030	1,268
lsb22777	(60,977)	0.0016	0.0221	965	0.0016	0.0154	5,394	=(9)	0.0003	0.0033	1,630
xrh24104	(69,294)	0.0101	0.0309	2,058	0.0058	0.0193	9,234	=(9)	0.0001	0.0014	1,763
bbz25234	(69,335)	0.0173	0.0268	1,462	0.0101	0.0196	9,753	=(9)	0.0001	0.0014	3,180
irx28268	(72,607)	0.0124	0.0299	1,205	0.0041	0.0204	6,114	=(9)	0.0003	0.0028	3,647
fyg28534	(78,562)	0.0140	0.0313	2,090	0.0076	0.0207	10,265	=(6)	0.0008	0.0038	3,632
icx28698	(78,089)	0.0346	0.0461	1,389	0.0128	0.0280	9,064	−**0.0013**	0.0006	0.0026	3,524
boa28924	(79,622)	0.0163	0.0237	1,752	0.0088	0.0161	10,517	=(3)	0.0014	0.0025	3,814
ird29514	(80,353)	0.0274	0.0352	1,565	0.0075	0.0236	9,243	=(4)	0.0015	0.0050	3,262
pbh30440	(88,313)	0.0192	0.0298	1,495	0.0136	0.0189	8,653	=(3)	0.0016	0.0034	3,246
xib32892	(96,757)	0.0207	0.0533	2,045	0.0145	0.0346	10,188	=(2)	0.0020	0.0041	4,137
fry33203	(97,240)	0.0062	0.0316	2,411	0.0041	0.0227	15,941	=(4)	0.0012	0.0031	4,407
bby34656	(99,159)	0.0171	0.0278	2,315	0.0131	0.0179	18,236	=(2)	0.0017	0.0030	4,687
pba38478	(108,318)	0.0231	0.0352	2,214	0.0157	0.0236	19,459	=(1)	0.0033	0.0055	5,557
ics39603	(106,819)	0.0187	0.0289	2,951	0.0112	0.0173	26,630	0.0028	0.0048	0.0066	6,852
rbz43748	(125,184)	0.0120	0.0288	2,816	0.0064	0.0165	22,588	−**0.0008**	0.0009	0.0024	8,114
fht47608	(125,106)	0.0328	0.0439	3,428	0.0192	0.0297	32,785	0.0008	0.0021	0.0056	11,102
fna52057	(147,789)	0.0196	0.0351	3,048	0.0156	0.0212	31,668	0.0007	0.0024	0.0054	11,729
bna56769	(158,082)	0.0171	0.0366	3,975	0.0032	0.0197	40,132	−**0.0019**	−0.0008	0.0013	12,537
dan59296	(165,372)	0.0224	0.0401	3,831	0.0139	0.0251	40,560	0.0012	0.0018	0.0042	16,182

instances, even though pla7379 is not a very large instance. The VLSI TSP instances also include a number of vertices arranged in a lattice pattern, and GA-EAX could not reach the best-known solutions for some instances with more than 39,000 vertices. The reason for this drawback, in our opinion, is that there are various different optimal (or near-optimal) solutions for this type of instance, and this would make it difficult for the population to converge into a promising region in the search space.

Next, we compare GA-EAX to LKH-2 (the number of iterations $= 0.2N$). The table shows that the solution quality of GA-EAX is better than that of LKH-2 in most instances, even though the computation time of GA-EAX is basically smaller than that of LKH-2. We have conducted the one-sided Wilcoxon rank sum test, and the solution quality of GA-EAX is significantly better than that of LKH-2 at a significant level of 0.05 for 50 instances out of all 57 instances. The advantage of GA-EAX is especially noticeable for instances with more than 10,000 cities. Given that LKH-2 is known as one of the most powerful heuristic algorithms for the TSP, our experimental results demonstrate the effectiveness of the proposed GA in finding very high-quality solutions.

6. Conclusions

In this paper we have proposed a powerful GA in finding very high-quality solutions for the TSP. The proposed GA has found optimal or best-known solutions for most benchmark instances with up to 200,000 cities and has improved 11 best-known solutions. Given the fact that the champion heuristic algorithms for the TSP have always been based on the LK algorithm for nearly 40 years, our GA has significant academic value because it achieves top performance without using an LK-based algorithm and is conceptually simple.

One of the strengths of our GA is the use of EAX, a high-potential crossover operator for the TSP, although the original GA using EAX is not comparable to state-of-the-art LK-based algorithms. We have demonstrated that localization of EAX significantly reduces the computational cost of EAX, with the help of efficient implementation techniques. This resolves the common problem that GAs (for the TSP) are usually much more time consuming than efficiently implemented local-search-based algorithms. Another important contribution is the development of EAX in generating even better offspring solutions from very high-quality parent solutions at the final phase of the GA. An interesting feature is that we introduce a simple local search procedure into EAX to determine good combinations of parents' edges. In addition to the EAX enhancements, we have proposed a GA model that effectively maintains population diversity without major additional computational costs. We have demonstrated that all enhancements significantly improve the performance of the GA. We believe that the proposed GA provides a good example of a successful GA application for a representative combinatorial optimization problem and that some of the ideas can be successfully applied to the design of GAs for other combinatorial optimization problems.

Currently, we have applied the proposed GA to TSP instances with up to 200,000 cities. On the other hand, some effective LK-based algorithms (Applegate et al. 2003, Helsgaun 2009) have been applied to very large TSP instances with 10,000,000 or even 25,000,000 cities. It will be difficult, however, to fully exercise the capability of our GA for such very large instances within a reasonable computation time. To tackle such large instances, we have to develop an efficient parallel implementation of our GA.

Electronic Companion

An electronic companion to this paper is available as part of the online version at http://dx.doi.org/10.1287/ijoc.1120.0506.

Acknowledgments

This work was partially supported by Grant-in-Aid for Young Scientists (B) [19700134].

References

Applegate D, Cook W, Rohe A (2003) Chained Lin-Kernighan for large traveling salesman problems. *INFORMS J. Comput.* 15(1):82–92.

Bixby RE, Cook W, Espinoza DG, Groycoolea M, Helsgaun K (2009) Certification of an optimal TSP tour through 85,900 cities. *Oper. Res. Lett.* 37(1):11–15.

Boese KD, Kahng AB, Muddu S (1994) A new adaptive multi-start technique for combinatorial global optimizations. *Oper. Res. Lett.* 16(2):101–113.

Cook W, Seymour P (2003) Tour merging via branch-decomposition. *INFORMS J. Comput.* 15(3):233–248.

Dong C, Jäger G, Richter D, Molitor P (2009) Effective tour searching for TSP by contraction of pseudo backbone edges. *Proc. 5th Internat. Conf. Algorithmic Aspects Inform. Management, Lecture Notes in Computer Science*, Vol. 5564 (Springer, Berlin, Heidelberg), 175–187.

Fredman M, Johnson D, McGeoch L, Ostheimer G (2005) Data structures for traveling salesman. *J. Algorithms* 18(3):432–479.

Freisleben B, Merz P (1996) New genetic local search operators for the traveling salesman problem. *Proc. 6th Internat. Conf. Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, Vol. 1141 (Springer, Berlin, Heidelberg), 890–899.

Goldberg D, Lingle R (1985) Allels, loci and the traveling salesman problem. *Proc. 1st Internat. Conf. Genetic Algorithms* (Lawrence Erlbaum, Pittsburgh) 154–159.

Grefenstette JJ, Gopal R, Rosmaita BJ, Gucht DV (1985) Genetic algorithms for the traveling salesman problem. *Proc. 1st Internat. Conf. Genetic Algorithms* (Lawrence Erlbaum, Pittsburgh) 160–168.

Helsgaun K (2000) An effective implementation of the Lin-Kernighan traveling salesman heuristic. *Eur. J. Oper. Res.* 126(1):106–130.

Helsgaun K (2006) An effective implementation of *k*-opt moves for the Lin-Kernighan TSP heuristic. *Datalogiske Skrifter (Writings on Computer Science)*, Vol. 109 (Roskilde University, Roskilde, Denmark).

Helsgaun K (2009) General *k*-opt submoves for the Lin–Kernighan TSP heuristic. *Math. Programming Comput.* 1(2):119–163.

Johnson DS, McGeoch LA (1997) The traveling salesman problem: A case study in local optimization. Aarts EHL, Lenstra JK, eds. *Local Search in Combinatorial Optimization* (John Wiley and Sons, London), 215–310.

Lin S, Kernighan BW (1973) An effective heuristic algorithm for the traveling salesman problem. *Oper. Res.* 21(2):498–516.

Maekawa K, Mori N, Tamaki H, Kita H, Nishikawa Y (1996) A genetic solution for the traveling salesman problem by means of a thermodynamical selection rule. *Proc. 1996 IEEE Conf. Evolutionary Comput.* (ICEC '96), 529–534.

Martin O, Otto SW, Felten EW (1991) Large-step Markov chains for the traveling salesman problem. *Complex Systems* 5(3):299–326.

Mathias K, Whitley D (1992) Genetic operators, the fitness landscape and the traveling salesman problem. *Proc. 2nd Internat. Conf. Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, Vol. 866 (Springer, Berlin, Heidelberg), 219–228.

Merz P, Freisleben B (1997) Genetic local search for the TSP: New results. *Proc. 1997 IEEE Internat. Conf. Evolutionary Comput.* (ICEC '97), 159–164.

Nagata Y (2006a) Fast EAX algorithm considering population diversity for traveling salesman problems. *Proc. 6th Internat. Conf. Evolutionary Comput. Combinatorial Optim., Lecture Notes in Computer Science*, Vol. 3906 (Springer, Berlin, Heidelberg), 171–182.

Nagata Y (2006b) New EAX crossover for large TSP instances. *Proc. 9th Internat. Conf. Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, Vol. 4193 (Springer, Berlin, Heidelberg) 372–381.

Nagata Y, Kobayashi S (1997) Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem. *Proc. 7th Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Francisco), 450–457.

Nguyen HD, Yoshihara I, Yamamori K, Yasunaga M (2007) Implementation of an effective hybrid GA for large-scale traveling salesman problems. *IEEE Trans. Systems, Man, and Cybernetics, Part B* 37(1):92–99.

Tsai HK, Yang JM, Tsai YF, Kao CY (2004) An evolutionary algorithm for large traveling salesman problems. *IEEE Trans. Systems, Man, Cybernetics, Part B* 34(4):1718–1729.

Ulder N, Aarts E, Bandelt HJ, van Laarhoven P, Pesch E (1991) Genetic local search algorithms for the traveling salesman problem. *Proc. 1st Internat. Conf. Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, Vol. 496 (Springer, Berlin, Heidelberg) 109–116.

Walshaw C (2002) A multilevel approach to the travelling salesman problem. *Oper. Res.* 50(5):862–877.

Whitley D, Hains D, Howe A (2010) A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. *Proc. 11th Internat. Conf. Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, Vol. 6323 (Springer, Berlin, Heidelberg), 566–575.

Whitley D, Starkweather T, Fuquay D (1989) Scheduling problems and traveling salesman: The genetic edge recombination. *Proc. 3rd Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Francisco), 133–140.

Yamamura M, Ono I, Kobayashi S (1996) Emergent search on double circle TSPs using subtour exchange crossover. *Proc. 1996 IEEE Internat. Conf. Evolutionary Comput.* (ICEC '96), 535–540.

Online Supplement to “Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem”

Yuichi Nagata, Shigenobu Kobayashi
Department of Computational Intelligence and Systems Science, Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, 4259 Nagatsuta, Midori-ku Yokohama, Kanagawa 226-8502, Japan {nagata@fe.dis.titech.ac.jp, kobayasi@dis.titech.ac.jp}

1. Solving Mona-Lisa TSP Challenge

The largest non-trivial TSP instance that has been solved to optimality is the 85,900-city TSP instance pla85900 (`http://www.tsp.gatech.edu/pla85900/index.html`) included in the TSPLIB. The leading research group on exact algorithms for the TSP is trying to solve larger instances to establish a new world record for the TSP. In particular, this research group has intensively investigated a 100,000-city instance named “Mona-Lisa 100K” for the next challenge (`http://www.tsp.gatech.edu/data/ml/monalisa.html`), and encouraged researchers to report a new best-known solution to this instance because knowledge of a good tour (hopefully an optimal tour) will substantially reduce the computational effort of the exact algorithm. So, a number of powerful heuristic algorithms have been extensively applied to this instance since it was provided in February 2009.

W applied the default GA 10 times to Mona-Lisa TSP. The best tour length of the 10 runs of the GA was 5,757,194, whereas the previous best-known tour length was 5,757,199. The computation time for a single run was approximately two weeks on an ADM Opteron 2.4 GHz processor (34.5 hours on a cluster with Intel Xeon 2.93 GHz nodes). To further improve the solution quality, we additionally performed Stage II of the default GA 10 times, starting from the same initial population constructed by assembling 30 tours from each of the 10 populations that were obtained at the end of the 10 runs of the default GA. Specifically, the top 30 tours were selected from each of the populations, without duplication of the already-assembled individuals. As a result, we found a better solution with a tour length of 5,757,191 (all 10 runs found the same solution). We reported this new best-known solution on March 17, 2009, which has not been improved (as of February 14, 2011).

We also applied the same procedure to the 57 instances to further improve the best solutions found by the 10 runs of the default GA (see Section 5.3 of the paper), which are listed in the column “Best of 10 runs” of Table 1 (the tour length is listed only if a new best-known solution was obtained). The tour lengths for the improved instances are listed in the column “Merge pop” of Table 1, showing that the best solutions were further improved in four instances.

Mona-Lisa 100K TSP is one of the six “Art TSPs” range in size between 100,000 and 200,000 cities (`http://www.tsp.gatech.edu/data/art/index.html`) that provide continuous line drawings of well-known pieces of art. In Table 1, we also present results for the remaining five Art TSPs, showing that new best-known solutions were found for all instances by the 10 runs of the default GA. The average computation times for a single run were about 9.8 days (Vangogh120K), 13.6 days (Venus140K), 19.1 days (Pareja160K), 25.0 days (Curbet180K), and 31.1 days (Earring200K), respectively, in the virtual machine environment on a cluster with Intel Xeon 2.93 GHz nodes. These results were further improved by the same procedure described above as shown in the column Merge pop of Table 1.

Table 1: New best-known solutions found by GA-EAX

instance	best-known	Best of 10 runs	Merge pop
bm33708	959304	**959293**	**959290**
ch71009	4566542	**4566508**	**4566506**
icx28698	78089	**78088**	**78088**
rbz43748	125184	**125183**	**125183**
fht47608	125106	125107	**125104**
bna56769	158082	**158079**	**158078**
Mona-Lisa100K	5757199	**5757194**	**5757191**
Vangogh120K	6543622	**6543616**	**6543610**
Venus140K	6810696	**6810671**	**6810665**
Pareja160K	7619976	**7619955**	**7619953**
Curbet180K	7888759	**7888739**	**7888733**
Earring200K	8171712	**8171686**	**8171677**

2. Implementation details of EAX

We present details of the efficient implementation of EAX (see Section 3 of the paper). This implementation actually makes it possible to execute the localized version of EAX (EAX with the single strategy) in less than $O(N)$ time (per generation of an offspring solution). The efficient implementation also makes the execution of global versions of EAX faster but has relatively small impact. Nevertheless, the efficient implementation has a great impact on the overall execution time of the default GA because most of the search is performed using the localized version of EAX (see Section 5.2.1 of the paper).

Let k denote the size of an *E-set*, and assume that $k \ll N$ when the single strategy is used for constructing *E-sets*. In this case, only Step 5 (in particular Step 5-1) requires $O(N)$ time per generation of an offspring solution, whereas the other steps can actually be performed in less than $O(N)$ time with the help of additional simple implementation techniques. In fact, if $k \ll N$, Step 5 can be also performed in less than $O(N)$ time by using an efficient implementation. In this section, we first describe the implementation of the EAX algorithm excluding Step 5 and then present the efficient implementation of Step 5.

2.1. Implementation excluding Step 5

We represent an individual in the population by a slightly deformed doubly linked list denoted by $Link[v][s]$ $(v = 1, \dots, N; s = 0, 1)$, where $Link[v][0]$ and $Link[v][1]$ store the two vertices adjacent to vertex v. Note that the two vertices can be stored without considering the precedence relation, whereas in a typical doubly linked list, $Link[v][0]$ and $Link[v][1]$ store the vertices that precede and follow vertex v, respectively. In this paper, we simply call this data structure a doubly linked list. Let $List_A$, $List_B$, and $List_C$ be doubly linked lists representing E_A, E_B, and E_C, respectively.

We do not need to do anything to perform Step 1 using $Link_A$ and $Link_B$ directly as G_{AB}.

Step 2 can be performed in $O(N)$ time, but it is not especially time consuming because this step is performed only once while Steps 3–6 are performed N_{ch} times (e.g., 30 in the best configuration). When the single strategy is used for constructing *E-sets*, we can terminate the procedure of this step immediately after N_{ch} effective *AB-cycles* are generated. In this case, this step can actually be performed in less than $O(N)$ time per generation of an offspring solution because only a tiny fraction of all *AB-cycles* is formed (e.g., the number of all effective *AB-cycles* is about 800 at the beginning of Stage I on instance usa13509 $(N = 13509)$).

In Step 3, the computational cost is obviously negligible when the single strategy is used. Note that the same holds when either random or K-multiple strategy is used, but this step requires $O(N)$ time when the block strategy is used. When the block2 strategy is used, the time complexity cannot be analyzed due to the use of the tabu search procedure.

Step 4 is performed through αk elementary operations (α is a constant factor) because E_C is represented as the (deformed) doubly linked list $List_C$. However, this step requires $O(N)$ time if all the elements of $List_A$ are copied to $List_C$ each time an offspring solution is generated. So, we define $Link_C$ as an alias of $Link_A$ and directly change $Link_A$ to generate an offspring solution. We must therefore undo $Link_A$ after an offspring solution is generated and store the changed edges (i.e., differences between an offspring solution and p_A) if necessary.

2.2. Implementation of Step 5

After Step 4, we have an intermediate solution E_C, which is represented by the doubly linked list $Link_C$. In Step 5-1, we compute m and A_l $(l = 1, \dots, m)$ from E_C. To know the number of vertices in a sub-tour, we must trace the vertices according to E_C, starting from a vertex in this sub-tour and counting the number of vertices traced until returning to the starting point. For example, even if an intermediate solution consists of one sub-tour (i.e., a valid tour), we must trace all the vertices to know that. In the naive implementation, this procedure is repeated, each time starting from a vertex that has not yet been visited, until all vertices are visited. Step 5-1 therefore requires $O(N)$ time per generation of an offspring solution.

Recall that EAX generates an intermediate solution from E_A by removing edges of E_A and adding edges of E_B in an *E-set*. We can reduce the computational cost of Step 5-1 by

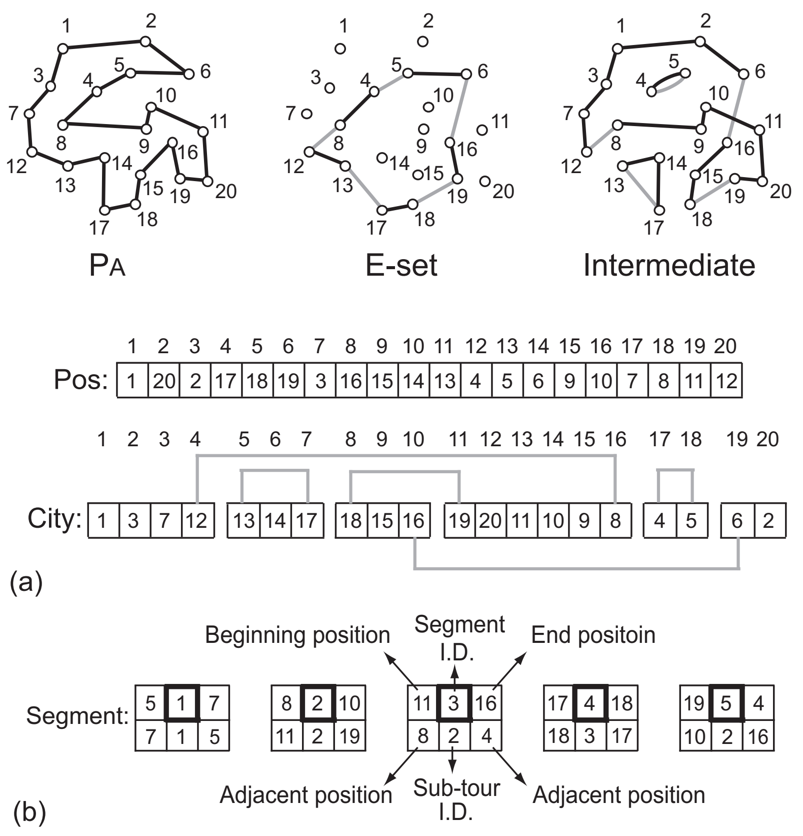

Figure 1: Segment representation of an intermediate solution.

using information about the order of the vertices in the parent solution p_A. Let $City[i]$ ($i = 1, \ldots, N$) be an array representing the sequence of vertices in p_A, and let $Pos[v]$ ($v = 1, \ldots, N$) be the inverse of $City$ (i.e., $Pos[v]$ represents the position of vertex v in $City$). Figure 1(a) illustrates the basic concept of the efficient implementation of Step 5-1, where the sequence of vertices represented by $City$ is divided into k segments ($City[N]$ and $City[1]$ are toroidally connected), and the ends of the segments are linked to each other by gray lines. Here, every cut point in the sequence corresponds to one of the removed edges (e.g., edge $(12, 13)$), and every new link corresponds to one of the added edges (e.g., edge $(12, 8)$). The segments are maintained by a data structure, which we call segment representation, similar to a two-level tree (Fredman et al., 2005). Figure 1(b) illustrates this data structure. Each of the segments is assigned an index value (*segment ID*) arbitrarily. Each segment contains the positions of both ends in $City$ (*beginning position* and *end position*), the positions of the ends of the two linked segments (*adjacent position*), and the index of the sub-tour to which it belongs (*sub-tour ID*). We can easily compute m and A_l ($l = 1, \ldots, m$) after the segment representation is obtained for an intermediate solution.

The efficient implementation of Step 5 is presented below. Let $S[v]$ ($v = 1, \ldots N$) be an array, where all elements are initialized to zero at the beginning of the search.

Implementation of Step 5:

5-1. Let $v_1, \ldots, v_k$ be one ends of the edges of E_A in the *E-set*, each of which is one of the ends of each edge with a greater value of *Pos* (e.g., 13, 18, 19, 4, and 6 in the example shown in Figure 1). Sort $Pos[v_1], \ldots, Pos[v_k]$ (e.g., 5, 8, 11, 17, and 19 in the example) in increasing order; for simplicity, we assume here that $Pos[v_1] <, \ldots, < Pos[v_k]$. Now, we can obtain the values of the *beginning position* and *end position* for the k segments as follows: $(Pos[v_1], Pos[v_2] - 1), \ldots, (Pos[v_k - 1], Pos[v_k] - 1), (Pos[v_k], Pos[v_1] - 1)$. For each segment, assign a *segment ID* arbitrarily, and determine the values of *adjacent positions* according to the edges of E_B in the *E-set* and *Pos*. By tracing the segments according to the values of the *beginning position*, *end position*, and *adjacent position*, we can classify the segments according to the sub-tours to which they belong. Then *sub-tour IDs* are assigned to the segments arbitrarily. Now, we can easily compute m and A_l $(l = 1, \ldots, m)$.

5-2. Let r be the index of the smallest sub-tour. Let V be a set of vertices in this sub-tour, which is obtained by tracing this sub-tour according to $Link_C$, starting from a vertex included in it (we store such a vertex for each sub-tour). Set $S[v] = 1$ $(v \in V)$ (undo S beforehand).

5-3. Find 4-tuples of edges $\{e^*, e'^*, e''^*, e'''^*\}$, where all pairs of edges $e \in U$ and $e' \in E_C \backslash U$ are generated as follows. Let e, e', e'', and e''' be represented as (v_1v_2), (v_3v_4), (v_1v_3), and (v_2v_4), respectively. Generate four vertices as follows:
$v_1 \in \{v | v \in V\}$,
$v_2 \in \{Link_C[v_1][0],\ Link_C[v_1][1]\}$,
$v_3 \in \{v | v \in Near(v_1, 10),\ S[v] \neq 1\}$,
$v_4 \in \{Link_C[v_3][0],\ Link_C[v_3][1]\}$.
$Near(v, N_{near})$ refers to a set of vertices that are among the N_{near} closest to v; we set $N_{near} = 10$ in our experiments because we found that the solution quality was hardly improved by setting $N_{near} = 20$ but was deteriorated by setting $N_{near} = 5$.

5-4. Update $Link_C$ in accordance with $E_C := (E_C \backslash \{e^*, e'^*\}) \cup \{e''^*, e'''^*\}$. Note that we directly update $Link_A$ because $Link_C$ is defined as the alias of $Link_A$.

5-5. Let ${v_3}^*$ be one end of edge e'^*, and find the segment including it (i.e., the segment satisfying *beginning position* $\leq Pos[v_3^*] \leq$ *end position*). Then r' is obtained as the *sub-tour ID* of this segment. Subtract 1 from m, reassign *sub-tour ID*s appropriately, and recompute A_l $(l = 1, \ldots, m)$ in accordance with the resulting intermediate solution. If m equals 1, then terminate the procedure. Otherwise, go to Step 5-2.

Here, we should note that the segment representation is used only for computing m and A_l $(l = 1, \ldots, m)$ in Step 5-1 and is never updated except for the *sub-tour ID* in Steps 5-2 to 5-5. In Step 5-5, we can easily recompute A_l $(l = 1, \ldots, m)$. For example, the size of the new sub-tour formed by connecting sub-tours r and r' is computed simply by adding the sizes of the two sub-tours.

We analyze the computational cost of Step 5. For simplicity, we assume that k is a fixed value. Step 5-1 is performed in $O(k \log k)$ time on average because k integer values must be sorted. Steps 5-2 to 5-5 are iterated $m - 1$ times, and we analyze the total computational cost of each step (per generation of an offspring solution). Although Steps 5-2 and 5-3 each require $O(N)$ time in the worst case, these steps can, in most cases, be performed more efficiently. Let L be the sum of the sizes of the sub-tours selected in Step 5-2 during the $m-1$ iterations. Each of Step 5-2 and Step 5-3 is performed through αL elementary operations, where α is a constant factor (Step 5-3 has a significantly larger constant factor). Indeed, $L \sim N$ in the worst case, but L is substantially smaller than N in most cases when the localized version of EAX is used because an intermediate solution frequently consists of one distinctly large sub-tour and other small sub-tours. For example, if an intermediate solution consists of four sub-tours whose sizes are 5, 7, 30, and 10000, the largest sub-tour is never selected in Step 5-2. Finally, in the worst case, Step 5-4 and Step 5-5 are performed in $O(k)$ and $O(k^2)$ times, respectively, because $m \leq k$.

We should note that obtaining arrays *City* and *Pos* from p_A requires $O(N)$ time. In fact, this computational cost can be reduced by storing these arrays as part of each individual solution in the population, together with the doubly linked list, and updating them appropriately. Nevertheless, we compute them each time a population member is selected as parent p_A because this step is performed only once and is not especially time consuming (Step 3-6 is performed N_{ch} times).

2.3. Impact of the efficient implementation

We executed the default GA on the 57 instances (See Section 5.1 of the paper) using the naive and efficient implementations in order to investigate the impact of the efficient implementation on the computation time. When the efficient implementation is used, (i) segment representation is used to speed up Step 5-1, (ii) $Link_A$ is directly modified to generate offspring solutions, and (iii) the generation of *AB-cycles* in Step 2 is terminated immediately after N_{ch} effective *AB-cycles* are formed (this applies only if EAX with the single strategy is used). The efficient version was applied to the 57 instances, but the naive version was applied to 36 instances with up to 25,000 cities because executing the naive version on larger instances would require a very long computation time. In addition, for large instances say more than 25,000 cities, the efficient version was implemented with the additional techniques to save the amount of memory required to execute it (see Appendix B).

Figure 2 shows the computation time for a single run of the default GA with naive implementation and with the efficient implementation, respectively. Here, the computation time displayed in the graph includes the computation time spent generating the initial population (e.g., 326 seconds for usa13509), even though this is not affected by the efficient implementation. The graph clearly shows a significant reduction in the computation time achieved using the efficient implementation. For example, the efficient version is about 10 times faster than the naive one on instances with about 25,000 cities. The impact of the reduction in the computation time will be more prominent for larger instances.

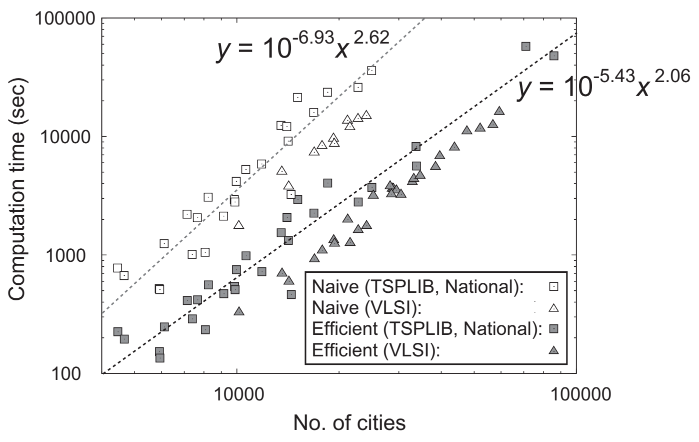

Figure 2: Effect of the efficient implementation: Computation time of the default GA (one run) with the naive implementation and with the efficient implementation, respectively. The x- and y-axes show the number of vertices and the computation time, respectively, on a log scale. The two lines show the results of the linear least squares fitting to the two sets of data points, respectively, where the model function is given by $y = 10^b x^a$. Note that data points for the VLSI benchmarks are excluded in the linear least squares fitting because the computation time for the VLSI benchmarks is evidently less than that for the TSPLIB and National benchmarks due to their specific city arrangement.

3. Comparison between block and block2 strategies

In our preliminary work (Nagata, 2006), we proposed a heuristic selection strategy of *AB-cycles*, called block strategy, to construct an effective global version of EAX. However, we have developed a new, more sophisticated global version of EAX, call block2 strategy, and details of the block strategy are omitted in the paper. This section presents details of the block strategy and a performance comparison between the block and block2 strategy.

3.1. Block strategy

The block strategy is described below (see Figure 3 for an illustration).

Selecting *AB-cycles* (Block strategy):

1. Select a relatively large *AB-cycle* (central *AB-cycle*). A temporal *E-set* is formed by selecting the central *AB-cycle.* Note that the central *AB-cycle* is selected in descending order of size when more than one offspring solution is generated.

2. Apply the temporal *E-set* (Step 4 of the EAX algorithm) to generate a temporal intermediate solution. Let V_1 be a set of vertices in the largest sub-tour (the sub-tour consisting of the largest number of edges) and V_l $(l = 2, \ldots, m)$ a set of vertices in each of the other sub-tours, in the temporal intermediate solution. If the temporal intermediate solution is a tour (i.e., $m = 1$), the temporal *E-set* is used as an *E-set*. Otherwise, go to step 3.

3. Construct an *E-set* by selecting the central *AB-cycle* as well as the *AB-cycles* that satisfy the following conditions; (i) at least one vertex in an *AB-cycles* exists in V_l ($l = 2, \ldots, m$), and (ii) the size of an *AB-cycle* is less than that of the central *AB-cycle*.

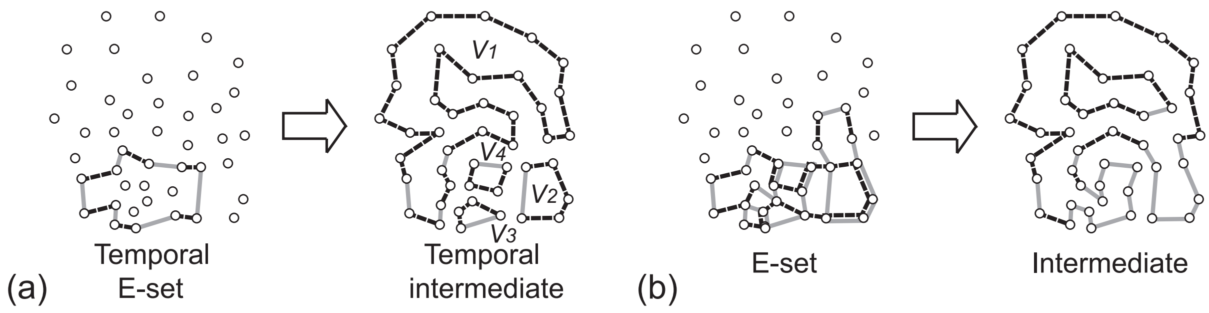

Figure 3: Illustration of the block strategy: The parent solutions and the set of *AB-cycles* are the same as those in Figure 1 of the paper. (a) An example of a temporal *E-set* (*AB-cycle* 3 is a central *AB-cycle*) and the resulting temporal intermediate solution. (b) An example of an *E-set* constructed by the block strategy and the resulting intermediate solution, where effective *AB-cycles* 3, 5, 6, and 9 are selected (the ineffective *AB-cycles* that satisfy the conditions are also selected here for the convenience of the explanation).

One basic principle of the block strategy is to select a relatively large *AB-cycle* (central *AB-cycle*) as a basic component so that the size of an *E-set* is basically greater than those of *E-sets* generated by the localized version of EAX (EAX with the single strategy). However, if an *E-set* consists of only the central *AB-cycle* (i.e., temporal *E-set*), a lot of small sub-tours are frequently formed around the region affected by the *E-set* in the resulting intermediate solution (i.e., temporal intermediate solution). To avoid such a situation, the block strategy additionally selects *AB-cycles* in Step 3 according to the condition (i); the resulting intermediate solution is generated from E_A by replacing a block of edges of E_A with a block of edges of E_B in the region in which the small sub-tours are formed in the temporal intermediate solution (i.e., the region covered by $V_2 \cup \ldots \cup V_m$). As a result, a sub-tour is never formed within this region in the intermediate solution because only edges of E_B are connected to the vertices in this region. At the same time, however, new sub-tours may be formed outside of this region. To reduce the formation of new sub-tours, the condition (ii) in Step 3 is also considered.

3.2. Performance comparison between block and block2 strategies

As is the case in Section 5.2.3 of the paper, we compare the impact of the block and block2 strategies on performance. For each instance, Stage I is performed using the default configuration, and Stage II is then performed using each of the two global version of EAX (other settings follow the default configuration). For the two global versions of EAX, Stage II starts from the same set of 10 populations obtained by performing Stage I 10 times.

Table 2 lists the results of Stage I and of Stage II with the two global versions of EAX. The solution quality of the block2 strategy, however, seems to be closed to that of the block strategy for the instances with up to 25,000 cities. In the table, we also present the number of trials, denoted as "T", in which both GAs found the same quality solution over the 10 runs, starting from the same population in each of the 10 runs. In the same way, "W" and "L" represent the numbers of trials in which the GA with the block2 strategy found a better and worse solution, respectively, than the GA with the block strategy. We can see that both GAs sometimes find the same quality solution. This is because the population members obtained after performing Stage I are fairly similar to each other (e.g., 93% of edges are common between two solutions in the population on instance usa13509), and the possible search space is already fairly reduced. This tendency is particularly true in small instances, say less than 10,000 cities. On the other hand, in larger instance, Stage II with the block2 strategy sometimes finds better solutions than Stage II with the block strategy, and this tendency becomes more prominent as the number of vertices increases. So, we additionally presents in Table 2 results on several large instances with more than 30,000 cities.

Table 2: Comparison between Block and Block2 Strategies

	Stage I (Single)				Stage II (Block)				Stage II (Block2)				B2 vs B		
instance	#S	A-Err	Gen	Time	#S	A-Err	Gen	Time	#S	A-Err	Gen	Time	W	T	L
fnl4461	0	0.00142	744	205	10	0.00000	52	9	10	0.00000	53	15	0	10	0
fi10639	0	0.00767	2076	923	7	0.00006	67	45	7	0.00006	63	60	0	10	0
usa13509	0	0.00415	2769	1476	4	0.00029	69	58	4	0.00028	62	62	3	6	1
xvb13584	0	0.01160	1956	609	8	0.00054	63	61	9	0.00027	61	91	1	9	0
d15112	0	0.00668	3954	2815	6	0.00004	82	108	8	0.00004	70	115	3	6	1
it16862	0	0.00942	3665	2145	3	0.00050	73	118	2	0.00052	68	114	0	9	1
pjh17845	0	0.01060	2302	976	3	0.00250	61	72	3	0.00187	69	120	2	8	0
d18512	0	0.00801	4935	3939	6	0.00012	88	123	8	0.00009	71	110	2	8	0
ido21215	0	0.01448	3395	1803	7	0.00063	75	114	6	0.00094	75	198	0	9	1
vm22775	0	0.00530	5078	2489	1	0.00126	66	160	1	0.00116	75	309	4	5	1
xrh24104	0	0.00592	3081	1579	9	0.00014	59	97	9	0.00014	59	184	0	10	0
sw24978	0	0.01009	6509	3425	2	0.00037	92	186	4	0.00028	90	293	3	7	0
bm33708	0	0.00851	8968	7464	10	-0.00033	109	565	10	-0.00051	124	754	6	2	2
bna56769	0	0.01404	10258	10537	7	-0.00038	121	1691	9	-0.00076	137	2000	4	5	1
dan59296	0	0.02032	10738	13833	0	0.00169	144	1709	0	0.00175	164	2349	1	7	2
ch71009	0	0.01320	27521	54940	10	-0.00046	320	6686	10	-0.00062	164	2642	9	1	0
M-L100K	0	0.00674	53783	117431	2	0.00007	574	13106	7	-0.00001	248	6653	10	0	0

Appendix

A. Implementation details of the local search for the initial population

Algorithm 1 presents the local search algorithm used for generating the initial population. The local search is a hill-climbing method using the *2-opt* neighborhood with well-known speed-up techniques (Johnson and McGeoch, 1997). In the algorithm, a possible *2-opt* move is represented as a 4-tuple of vertices, where (v_1, v_2) and (v_3, v_4) are the edges to be deleted and (v_1, v_3) and (v_2, v_4) are the edges to be added. Let $Neighbor[v][0]$ and $Neighbor[v][1]$ indicate the vertices that precede and follow vertex v, respectively, in the current tour. Let $near[v][j]$ indicate the j-th nearest vertex to vertex v. This algorithm uses a variant of the "don't look bit" strategy, where H is a set of vertices whose don't look bit is zero; H is updated by adding to it a set of vertices that are at most within the 50 nearest to one of the vertices v_1, v_2, v_3, and v_4 when the current tour is moved (line 10). The size of the *2-opt* neighborhood is effectively reduced by the conditional branching in line 8. The tour is represented by a two-level tree (Fredman et al., 2005).

Algorithm 1 : Procedure LOCAL-SEARCH()

```
 1: Randomly generate a tour and set $H := \{1, \dots, N\}$;
 2: **repeat**
 3:    Randomly select $v_1 \in H$;
 4:    **for** $i := 0$ to 1 **do**
 5:       $v_2 := Neighbor[v_1][i]$;
 6:       **for** $j := 1$ to 50 **do**
 7:          $v_3 := near[v_1][j]$;
 8:          **if** $-d(v_1, v_2) + d(v_1, v_3) \geq 0$ **then break**;
 9:          $v_4 := Neighbor[v_3][(i + 1) \bmod 2]$;
10:          **if** $-d(v_1, v_2) - d(v_3, v_4) + d(v_1, v_3) + d(v_2, v_4) < 0$ **then** Update the current tour
             and $H$. Go to line 3;
11:       **end for**
12:    **end for**
13:    $H := H \setminus \{v_1\}$;
14: **until** $H$ becomes empty
15: **return** the current tour;
```

B. Implementation for large instances

In our implementation of the GA, we use an $N \times N$ matrix to store the distances of all edges. In addition, we use another $N \times N$ matrix to stores the frequencies of the edges in the population for entropy-preserving selection. However, for large instances, say more than 25,000 cities, we cannot use these matrices due to the limitation of the memory storage capacity. For large instances, we implement the GA in the following way in order to save the amount of memory required. From our observations, this implementation increases the overall execution time of the default GA by a factor of 1.3–1.5.

Let $near[v][j]$ $(j = 1, \dots, 50)$ indicate the j-th nearest vertex to vertex v. In addition, let $D[v][j]$ $(j = 1, \dots, 50)$ be the distance between v and $near[v][j]$. Each population member is represented by two doubly linked lists, denoted by *Link* and *Order*. The first one is the same doubly linked list as that used in the original implementation, where $Link[v][s]$ $(s = 0, 1)$ stores the two vertices adjacent to v. In the second list, $Order[v][s]$ $(s = 0, 1)$ stores j^* such that $Link[v][s] = near[v][j^*]$. If such j^* does not exist (i.e., the distance between v and $Link[v][s]$ is greater than the distance between v and $near[v][50]$), then null is assigned. Note that, in practice, null is rarely assigned to the population members.

The distance of an edge $(v, Link[v][s])$ included in a population member can be obtained as $D[v][Order[v][s]]$ if $Order[v][s]$ is not null. Otherwise, we compute the Euclidean distance of this edge. In Step 5-3 of the EAX algorithm, we must know the distances of new edges, (v_1, v_3) and (v_2, v_4). Although we can obtain the distance of (v_1, v_3) from D, we must compute the Euclidean distance of (v_2, v_4).

The frequency of the edges in the population is represented by $F[v][j] (v = 1, \dots, N,\ j = 1, \dots, 50)$, where $F[v][j]$ stores the number of individuals in which one of the values of $Order[v][s]$ $(s = 0, 1)$ is j (≤ 50). Therefore, some edges represented as null in *Order* are ignored.

References

Fredman, M., D. Johnson, L. McGeoch, G. Ostheimer. 2005. Data Structures for Traveling Salesman. *Journal of Algorithms* **18** 432–479.

Johnson, David S., Lyle A. McGeoch. 1997. The traveling salesman problem: A case study in local optimization. E. H. L. Aarts, J. K. Lenstra, eds., *Local Search in Combinatorial Optimization*. John Wiley and Sons, London, England, 215–310.

Nagata, Y. 2006. New EAX crossover for large TSP instances. *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature, LNCS 4193*. 372–381.