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 $Link[v][s]$ $(v = 1, \ldots, N; s = 0, 1)$, 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 $(l = 1, \ldots, m)$ 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

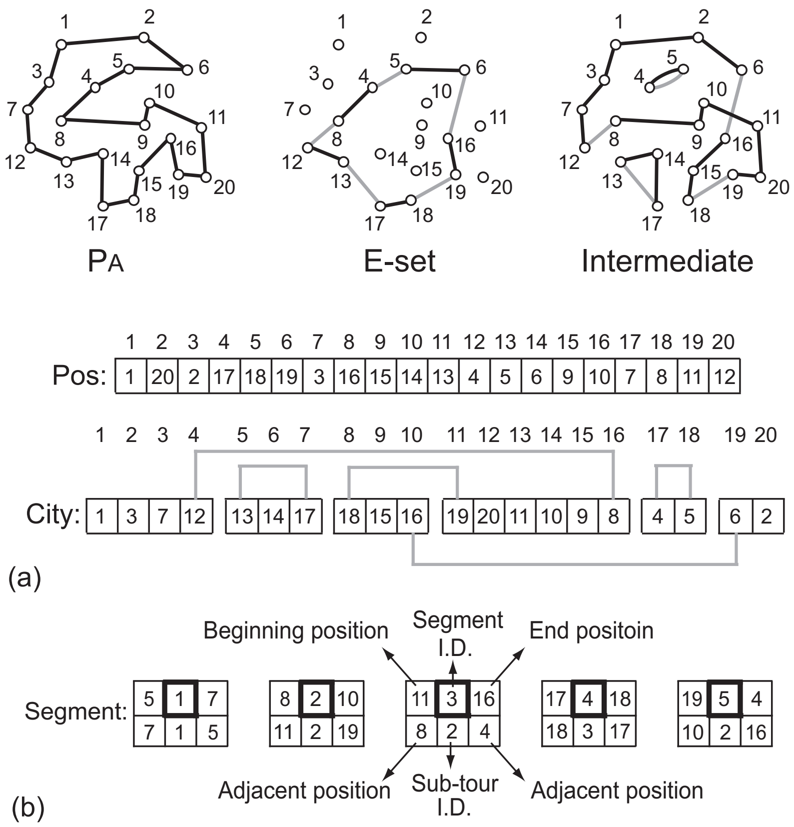

Figure 1: Segment representation of an intermediate solution.

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 $(v_1 v_2)$, $(v_3 v_4)$, $(v_1 v_3)$, and $(v_2 v_4)$, 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.

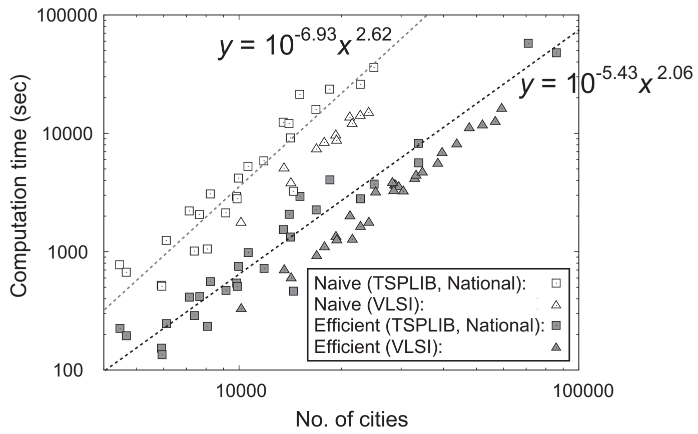

Figure 2: Effect of the efficient implementation: Computation time of the default GA (one run) with the naive implementation and with the efficient implementation, respectively. The x- and y-axes show the number of vertices and the computation time, respectively, on a log scale. The two lines show the 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*.

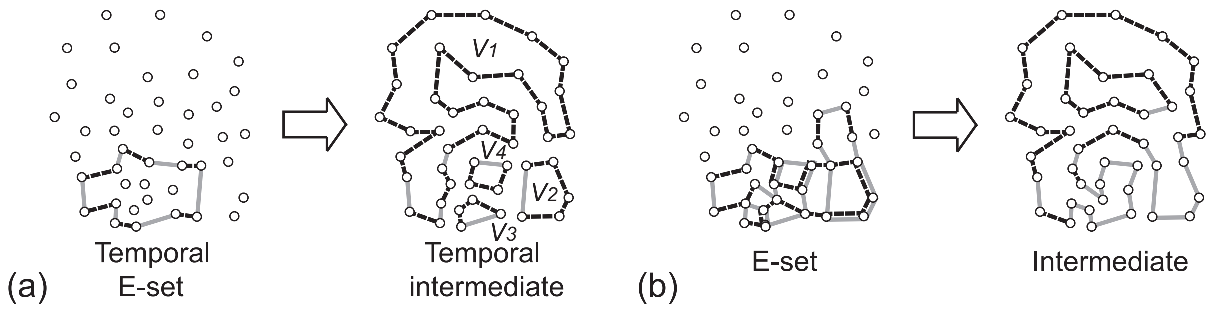

Figure 3: Illustration of the block strategy: The parent solutions and the set of *AB-cycles* are the same as those in Figure 1 of the paper. (a) An example of a temporal *E-set* (*AB-cycle* 3 is a central *AB-cycle*) and the resulting temporal intermediate solution. (b) An example of an *E-set* constructed 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 $H := \{1, \ldots, N\}$;
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:   $H := H \backslash \{v_1\}$;
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 $near[v][j]$ $(j = 1, \ldots, 50)$ indicate the j-th nearest vertex to vertex v. In addition, let $D[v][j]$ $(j = 1, \ldots, 50)$ 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 $F[v][j] (v = 1, \ldots, N,\ j = 1, \ldots, 50)$, 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.