Hindawi Publishing Corporation
Applied Computational Intelligence and Soft Computing
Volume 2014, Article ID 137928, 17 pages
http://dx.doi.org/10.1155/2014/137928

Research Article

Developing Programming Tools to Handle Traveling Salesman Problem by the Three Object-Oriented Languages

Hassan Ismkhan[1] and Kamran Zamanifar[2]

[1] *University of Bonab, Bonab, East Azerbaijan, Iran*
[2] *University of Isfahan, Isfahan, Isfahan, Iran*

Correspondence should be addressed to Hassan Ismkhan; esmkhan@gmail.com

Received 3 September 2014; Revised 3 October 2014; Accepted 8 October 2014; Published 23 December 2014

Academic Editor: Baoding Liu

The traveling salesman problem (TSP) is one of the most famous problems. Many applications and programming tools have been developed to handle TSP. However, it seems to be essential to provide easy programming tools according to state-of-the-art algorithms. Therefore, we have collected and programmed new easy tools by the three object-oriented languages. In this paper, we present ADT (abstract data type) of developed tools at first; then we analyze their performance by experiments. We also design a hybrid genetic algorithm (HGA) by developed tools. Experimental results show that the proposed HGA is comparable with the recent state-of-the-art applications.

1. Introduction

The objective of TSP is to find the shortest tour among a set of cites. Given the distance matrix $D(d_{ij})$ where d_{ij} stands for distance between the city i and j, the problem is called symmetric TSP (STSP) when $d_{ij} = d_{ji}$ and, otherwise, it is named asymmetric TSP (ATSP).

Since TSP is NP-Complete, there is no exact algorithm with time complexity better than an exponential time. It means that exact algorithms are not practical for the large-scale instances in reasonable running times, so we have to use approximate algorithms to find the semioptimal solutions in acceptable running times. Recently, many approximate algorithms have been developed to handle TSP instances [1–4]. The types of metaheuristics like genetic algorithms (GA) [5–7], simulated annealing [8], swarm based algorithm [9], artificial bee colony algorithm [10], ant colony algorithms [11, 12], and combination of these algorithms have been applied to the TSP [13, 14]. However, if we consider the experiment sections of these references, we observe that almost all of these algorithms have not been applied to the instances with size of more than 1000. Among these metaheuristics, surly, Lin-Kernighan (LK) which is a type of local search algorithms (LSAs) (in this paper, LS points to the local search) is one of the best algorithms in which its extended types have been successfully applied to the large-scale instances with size of more than 85000 nodes [3, 15]. In addition, in many cases, these algorithms have been used in other metaheuristics and have increased their performance [2, 11, 16].

LSAs include 2- and 3-opt and Lin-Kernighan (LK) algorithms have been based on edges exchange process [1, 3, 15, 17]. GAs are population-based and their efficiency depends on their operators [4]. To easily use these algorithms, we have programmed objective tools by three object-oriented languages which include C++, C#, and Java. These tools allow the researchers or developers to exploit these metaheuristics easily and create their own hybrid algorithms (these tools will be available via email request to esmkha@gmail.com (subject: TSP_Tools)).

Developed tools in this paper mainly focus on types of genetic operators and LSAs; however, types of ant colony optimization (ACO) have been implemented separately and will be available. The implementation of LSAs has been based on LKH implementation [1, 15, 18] which is one of the most famous and effective implementations of LK. In addition, some famous initial-solution constructors like

Suppose tour T with E_T edges that is defined on graph G(V, E):
(1) Suppose direction for T.
(2) If there are not nodes like A, B, C and D with below conditions then go to end.
(i) AB, CD $\in E_T$
(a) In supposed direction, B and D are right nodes of A and C respectively.
(ii) Cost(AB) + cost(CD) > cost(AC) + cost(BD)
(3) Remove AB and CD form E_T and add AC and BD to it (2-opt-move).
(4) Go to (2).
(5) End.

Algorithm 1: General algorithm for 2-opt.

Quick-Boruvka and nearest neighbor (NN) strategy have been included in these tools. The genetic operators have been selected from literature. These operators include the PMX [19], EPMX [20], VGX [21], IGX [5], GX (description of this operator and its versions can be found in [5, 19, 21, 22]), GSX-0, GSX-1, GSX-2, DPX [16], and OX [23]. The implementations of these operators are effective. Experimental results show that, in almost all cases, the performance (in the terms of running time and accuracy) of developed operators is even better than reported results in their references.

This paper is not limited to the developed tools only. A type of hybrid GA which is proposed in this paper and uses a two-storey strategy is fast and accurate. Experimental results show that performance of proposed hybrid algorithm outperforms one of the recent state-of-the-art algorithms.

With these descriptions, this paper is organized as follows: in the rest of this paper, we briefly describe LSAs and the ADT of their programming pack. We review GA, its operators, and the ADT of their class in the third section. In Section 4, we combine the LK into the GA and design a hybrid GA. We put forward experimental results of these algorithms in Section 5 and finally summarize the paper in Section 6.

2. LSAs

Majority of LSAs for TSP have been based on the edges exchange process. The 2- opt, 3-opt, and LK are three important algorithms that are categorized in LSAs. We have programmed these heuristics by C++, C#, and Java. In this section, we review their algorithms briefly, and then we state ADTs of their programming tools.

2.1. The 2-Opt. The 2-opt is a special case of the K-OPT. A tour is named K-OPT, if it is impossible to decrease the cost of tour by changing K number of edges. The 2-opt converts an input tour to its possible 2-opt case. Algorithm 1 shows the general algorithm for the 2-opt.

Instruction 3 in Algorithm 1 is named 2-opt-move or 2-change that is shown in Figure 1. In the 2-opt algorithm the 2-opt-move occurs when conditions in instruction 2 are satisfied. Time complexity of running exact 2-opt is high, so, to improve speed of the 2-opt algorithm, researchers usually use two important rules which have been proposed by Bentley.

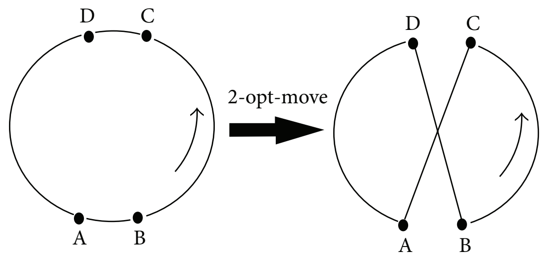


Figure 1: 2-opt-move.

(1) For each node A in line 2, we only consider its candidate nodes. Usually the five nearest neighbors are selected to make a candidate set of each node. These sets can be approximately calculated by k-d-tree [25] in $O(n \log^2 n)$.
(2) In the instruction 2, only the active nodes are considered. The nodes which have participated in tour cost reduction in previous iteration are activated for the next iteration. This heuristic is known as "do not-look bits" [26].

2.2. 3-OPt. The 3-opt operates like the 2-opt but its conditions to exchanging edges is rather complicated (see [27]). Algorithm for the 3-opt in each step probes 3 edges to exchanging, so when three edges are deleted, six considerable cases appear and probing these cases increases time complexity and algorithm becomes more complicated to implement.

2.3. LK. The types of LK may be the best heuristics that have been successfully applied to TSP. Furthermore, other metaheuristics like GAs widely use variant versions of this heuristic to improve their solutions. For more description about LK, we recommend readers to refer to [15, 17] but here we present this algorithm in brief. The LK can be introduced by the three words: "break," "link," and "condition test". The LK algorithm is done in some iterations. In each iteration, it exchanges some edges by another to reduce tour cost. Appendix A shows a simple algorithm for LK.

2.4. Review ADT of Class for LSAs. To implement LSAs, we need to define some primitive data structure like graph and tour at first, because these data structure definitions are necessary for other parts of program and classes. In the rest

```
      //nodes indexes start at 0 and go up to graph dimenion − 1
(1)   class Graph
(2)   {
(3)   public:
(4)     Graph(char* path);
(5)     ~Graph();
(6)     int D(int node1,int node2);
(7)     int D(int x1,int y1,int x2,int y2);
(8)     int Dimension();
(9)     double X(int node);
(10)    double Y(int node);
(11)  };
```

Algorithm 2: Graph ADT.

```
(1)   class Tour
(2)   {
(3)   public:
(4)       Tour(Graph* graph);//tour constructor gives a gaph object pointer as argument
(5)       ~Tour();
(6)       int Add_Right(int node); //exends uncomlete tour from right
(7)       int Add_Left(int node); //exends uncomlete tour from left
(8)       int Right(int node); // return right neighbour of node in complete tour
(9)       int Left(int node); //return left neighbour of node in complete tour
(10)      unsigned long long Cost(); //computes and returns cost
(11)      void InitiateRandomly();//forms tour by sequence: 0, 1, ..., dimenion − 1
(12)      Tour* Copy();
(13)      short IsComplete();//if tour is complet, this function returns 1 otherwise 0.
(14)      void reset();
(15)  };
```

Algorithm 3: Tour ADT.

of this subsection, we define class for graph and tour at first; then, we present class for heuristic methods.

2.4.1. Graph Class. Our graph implementation has been packed in Graph class. It can read .tsp files and compute distance between nodes. It supports all known TSP formats like GEO, GEOM, ATT, EU-2D, and CEIL-2D. Algorithm 2 shows Graph ADT. Graph class object should read TSP file by its constructor as soon as it is created (line 4).

2.4.2. Tour Class. Tour ADT is shown in Algorithm 3. Tour object is created to belong to Graph object. Tour object is completed after adding n (= dimension) nodes, either adding to the right (by using function in line 6) or adding to the left (by using function in line 7).

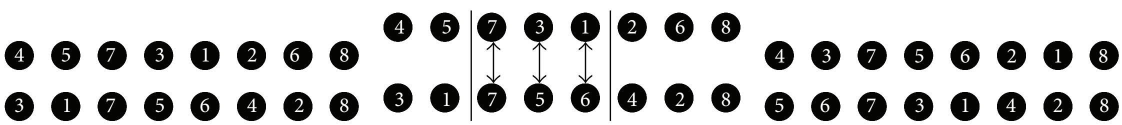
2.4.3. Heuristics Class. We have packed 2-opt, 3-opt, LK, Quick-Boruvka, and double-bridge into the Heuristics class. To manipulate the candidate sets, we have also added some functions into the Heuristics class. Quick-Boruvka is effective tour constructor algorithm. Double-bridge is usually used to mutate. Algorithm 4 shows Heuristics class ADT. The LK method in Heuristics class has been based on latest version of LKH, so-called LKH-2, and its source code is in C language and free for academic use [18].

3. Genetic Algorithm

Genetic algorithm is one of the search algorithms that is inspired by evolutionary process of nature. In recent years, researchers have solved many NP-Complete problems by GA, scheduling [28, 29], routing [30], and assignment [31, 32] and many other problems have been solved by GA effectively in recent years. GA works with population of solutions and, in each step, new solution is created by the crossover operator, or one or more solutions are changed by the mutation operator. The crossover operators usually get two solutions from the population. These two solutions are so-called parents (or the father and mother). The crossover creates new solution(s) based on the parents. The new solution is called child or offspring. There is question in which solutions are suitable to submit to mutation or crossover operators. There are some papers answering this question [33, 34]. Crossover and mutation are two operators of GA which play an important role in evolution of solutions of GA. Generally, LSAs include LK extensions such as iterated LK (ILK) [3] and LKH versions [1, 15] are very powerful in dealing with TSP. However, there are some effective GA or extensions like Nagata's one [4] that uses very efficient crossover operator, so-called edge assembly crossover (EAX). In this section, we review some of these crossover operators which have been included in the developed tools.

```
/* We implement this class function according to LKH[18]that is free for academic use.*/
(1)   class Heuristics
(2)   {
(3)   public:
        /*Heuristic object compute candidate sets as soon as created, second argument in
       constructor is the number of candidates in each set.*/
(4)    Heuristics(Graph* graph, int NumberOfCandidates);
(5)    ~Heuristics();
        /*Both of lines(6)and(7)shows the 2-OPT but function in line(6)consider all of   nodes
       as active but function(7)supposes only nodes in ActiveNodes array are active   */
(6)    void TwoOpt(Tour* tour);
(7)    void TwoOpt (Tour* tour, int *ActiveNodes, int NumberOfActiveNodes);
       /*Both of lines(8)and(9)shows the 3-OPT but function in line(8)consider all of   nodes
       as active but function(9)supposes only nodes in ActiveNodes array are active   */
(8)    void ThreeOpt(Tour* tour);
(9)    void ThreeOpt (Tour* tour, int *ActiveNodes, int NumberOfActiveNodes);
        /*Both of lines(10)and(11)shows the LK but function in line(10)consider all of   nodes
       as active but function(11)supposes only nodes in ActiveNodes array are   active */
(10)    void LinKernighan(Tour* tour);
(11)    void LinKernighan(Tour* tour, int *ActiveNodes, int NumberOfActiveNodes);
(12)    void DoubleBridge(Tour *tour);
(13)    Tour* Q_Boruvka();
(14)    int SetCandidates(int node, int candidate, int index);
(15)    int GetCandidates(int node, int index);
(16)    void SetBestTour(Tour *best_tour);
(17)  };
```

Algorithm 4: Heuristics ADT.



(a) Parents (b) Selections of arbitrary points (c) Children

Figure 2: PMX example.

3.1. Genetic Operators Review. Many GA crossover operators have been invented by researchers because the performance of GA depends on an ability of these operators. PMX [19] is one of the first crossovers which have been proposed by Goldberg and Lingle in 1985. Reference [20] states some shortcomings for PMX and to overcome them, proposing extended PMX (EPMX). DPX [16] is another crossover that produces child with greedy reconnect of common edges in two parents. Greedy subtour crossovers (GSXs) [24, 35, 36] family is another group of crossovers that operate fast. GSX-2 [36] is improved version of GSX-0 [35] and GSX-1 [24]. Order crossover (OX) proposed by Davis is another one in which its extensions not only have been applied on TSP [23] but also solved many other NP-Completes [32, 37].

In this subsection, we represent some of the recent GA crossovers and introduce them by examples. In these examples, we use the graph with eight nodes as this set: {1, 2, 3, 4, 5, 6, 7, 8} that its edges weight is as shown in Table 1.

Table 1: Distance matrix for examples.

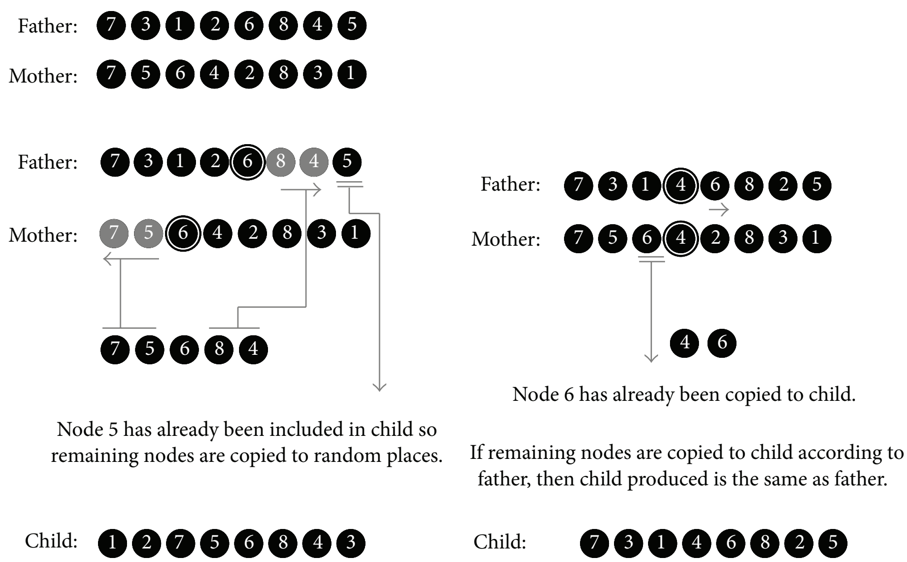
	1	2	3	4	5	6	7	8
1	0	12	19	31	22	17	23	12
2	12	0	15	37	21	28	35	22
3	19	15	0	50	36	35	35	21
4	31	37	50	0	20	21	37	38
5	22	21	36	20	0	25	40	33
6	17	28	35	21	25	0	16	18
7	23	35	35	37	40	16	0	14
8	12	22	21	38	33	18	14	0

3.1.1. PMX Crossover. Partially mapped crossover (PMX) is one of the first genetic operators. It produces two children according to two parents by exchanging nodes between two arbitrary points.

PMX is unable to detect the same nodes from mapped areas. In Figure 2, it can be easily seen that PMX cannot determine that node 7 is common in both mapped areas. PMX is double point crossover and these crossovers are not suitable to solve TSP. These defects can cause repetitive children production by this crossover [20].

```
Node X ← select random node;
Copy X to child;
Node R ← X;
Node L ← X;
while(true)
{
  R ← right neighbor of R in father tour;
  L ← left neighbor of L in mother tour;
  if R is in child then break while loop;
  if L is in child then break while loop;
  Add node R to child right side;
  Add node L to child left side;
}
Complete child by remaining nodes (nodes haven't been copied to child tour yet) in random;
return child;
```

Algorithm 5: Pseudocodes for GSX-0.



(a) GSX-0 example (b) GSX-1 example

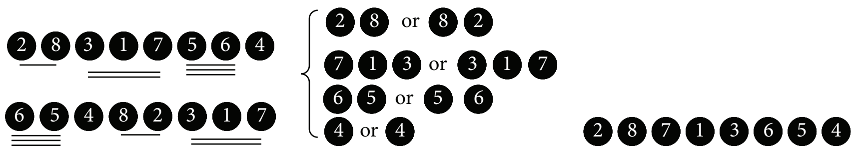
Figure 3: Examples for GSX-0 and GSX-1.

3.1.2. EPMX Crossover. Reference [20] tries to overcome PMX's shortcomings and proposes extended PMX (EPMX). It selects one arbitrary point and exchanges unique nodes before this arbitrary point and produces two children. As example of EPMX, given father = 1-2-3-4-5-6-7-8 and mother = 1-4-8-6-2-3-5-7, suppose that arbitrary point = 4 so father = 1-2-3-4|5-6-7-8 and mother = 1-4-8-6|2-3-5-7 are divided to two sublists. Nodes 2 and 3 from first sublist of father are not repeated in first sublist of mother and nodes 8 and 6 from first sublist of mother are not repeated in first sublist of father so $\{(2 \leftrightarrow 6), (3 \leftrightarrow 8)\}$ form exchanges so children are produced as child1 = 1-4-8-6-5-3-7-2 and child2 = 1-2-3-4-8-6-5-7.

3.1.3. Greedy Crossovers (GXs). Some versions of GX like very greedy crossover (VGX) [21] and improved greedy crossover [5] have been proposed by researchers in recent years. To review these crossovers, readers can refer to [5].

3.1.4. Improved Greedy Subtour Crossover (GSX-2). GSX-2 [36] is improved version of GSX-0 [35] and GSX-1 [24]. GSX-0 is first version of GSX family. Algorithm 5 shows GSX-0 algorithm.

Figure 3(a) shows GSX-0 example. In this example, after node 5 that has been included in child is met again, GSX-0 fills remaining places with random nodes but, same as Figure 3(b)

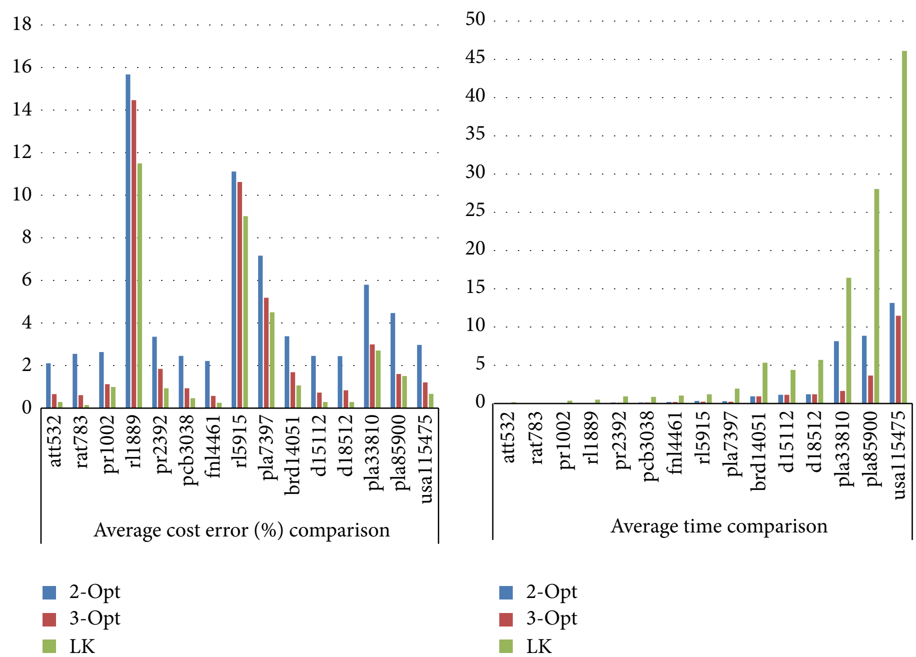


(a) Parents (b) Common subpaths (c) Child

Figure 4: DPX example.

```
(1)   class Crossovers
(2)   {
(3)   public:
(4)       CrossoversGraph(Graph *);
          //Original greedy crossover's function definition.
(5)       void GX(Tour *, Tour *, Tour *);
          //Another version of greedy crossover's function definition [17, 21].
(6)       void GX_4(Tour *, Tour *, Tour *);
          //Function definition of another version of GX [17, 21].
(7)       void GX_4_Pool(Tour *, Tour *, Tour *);
          /* lines (8) to (15) show proposed function definitions of crossovers that proposed in
          [3, 5, 16, 20, 21, 23, 24] respectively. */
(8)       void VGX(Tour *, Tour *, Tour *);
(9)       void IGX(Tour *, Tour *, Tour *);
(10)      void DPX(Tour *, Tour *, Tour *);
(11)      void GSX(Tour *, Tour *, Tour *);
(12)      void OX(Tour *, Tour *, Tour *, Tour *);
(13)      void PMX(Tour *, Tour *, Tour *, Tour *);
(14)      void EPMX(Tour *, Tour *, Tour *, Tour *);
(15)  };
```

Algorithm 6: Crossovers class ADT.



(a) (b)

Figure 5: (a) Average cost error percent. (b) Average of twenty runtimes of each instance.

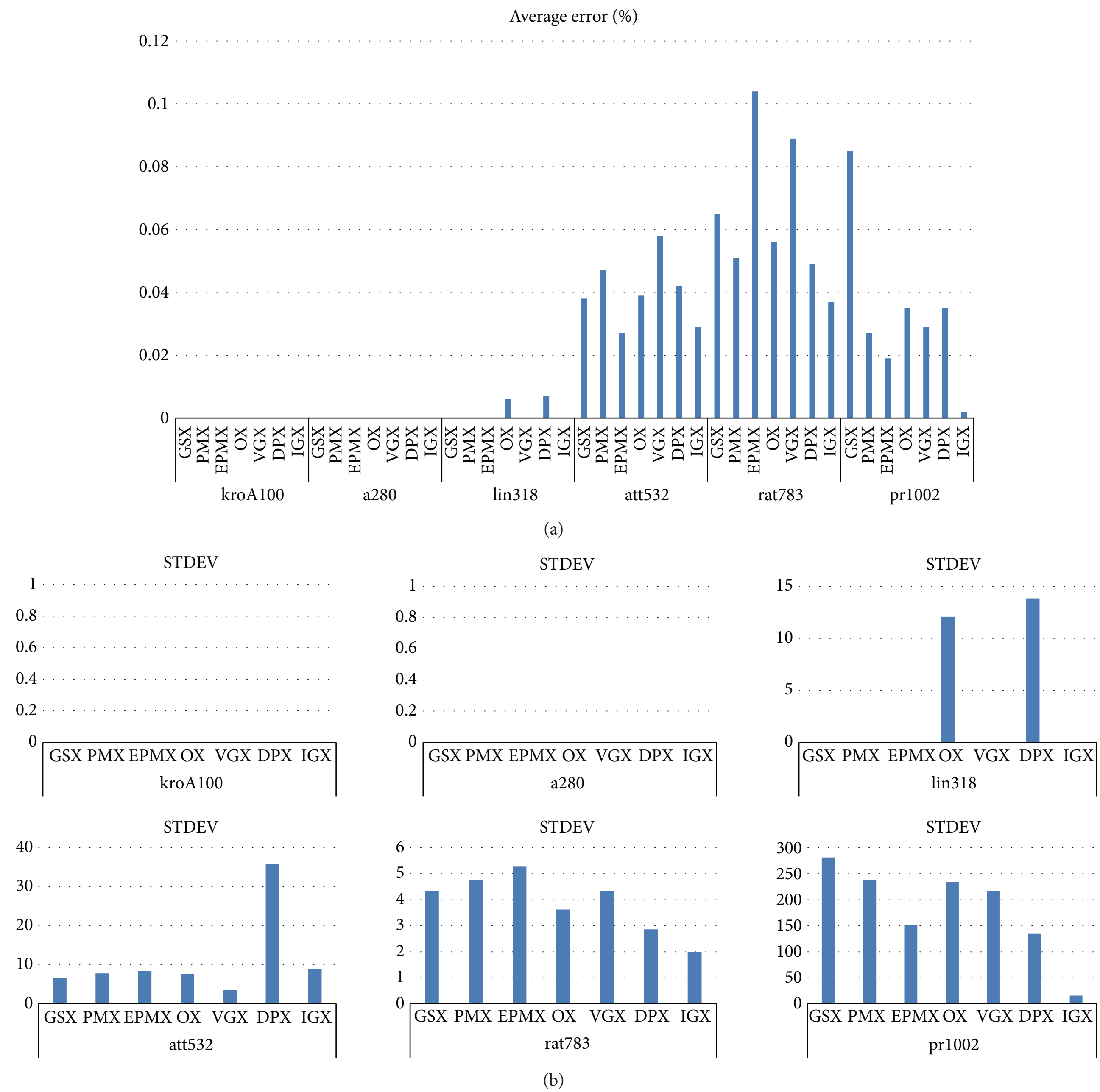


Figure 6: (a) Average percent and (b) standard deviation.

that shows GSX-1 example, it fills remaining nodes in order of one of parents.

In general, GSX-1 operates better than GSX-1 because it can preserve order of remaining nodes but, in some cases, it produces repetitive tours; same as Figure 3(b), child tour is the same as father tour. Reference [36] states some shortcomings of GSX-0 and GSX-1 and, to overcome these shortages, proposes GSX-2.

3.1.5. Distance Preserving Operator (DPX). DPX [16] operates as follows: it detects common subpaths of two parents at first and then reconnects them greedily and produces child. Figure 4 shows DPX example that uses presented edges weight of graph in Table 1.

3.2. ADT of Class for Crossovers. We have packed crossover operators into the *Crossover class*. Algorithm 6 shows ADT of *Crossover class*. Lines 5 to 14 show definitions of crossovers functions. Functions in lines 5 to 12 show crossovers which take two tours as father and mother and produce only one child, so their functions take three tour-pointers as the input arguments. The first two arguments are to point to the parent

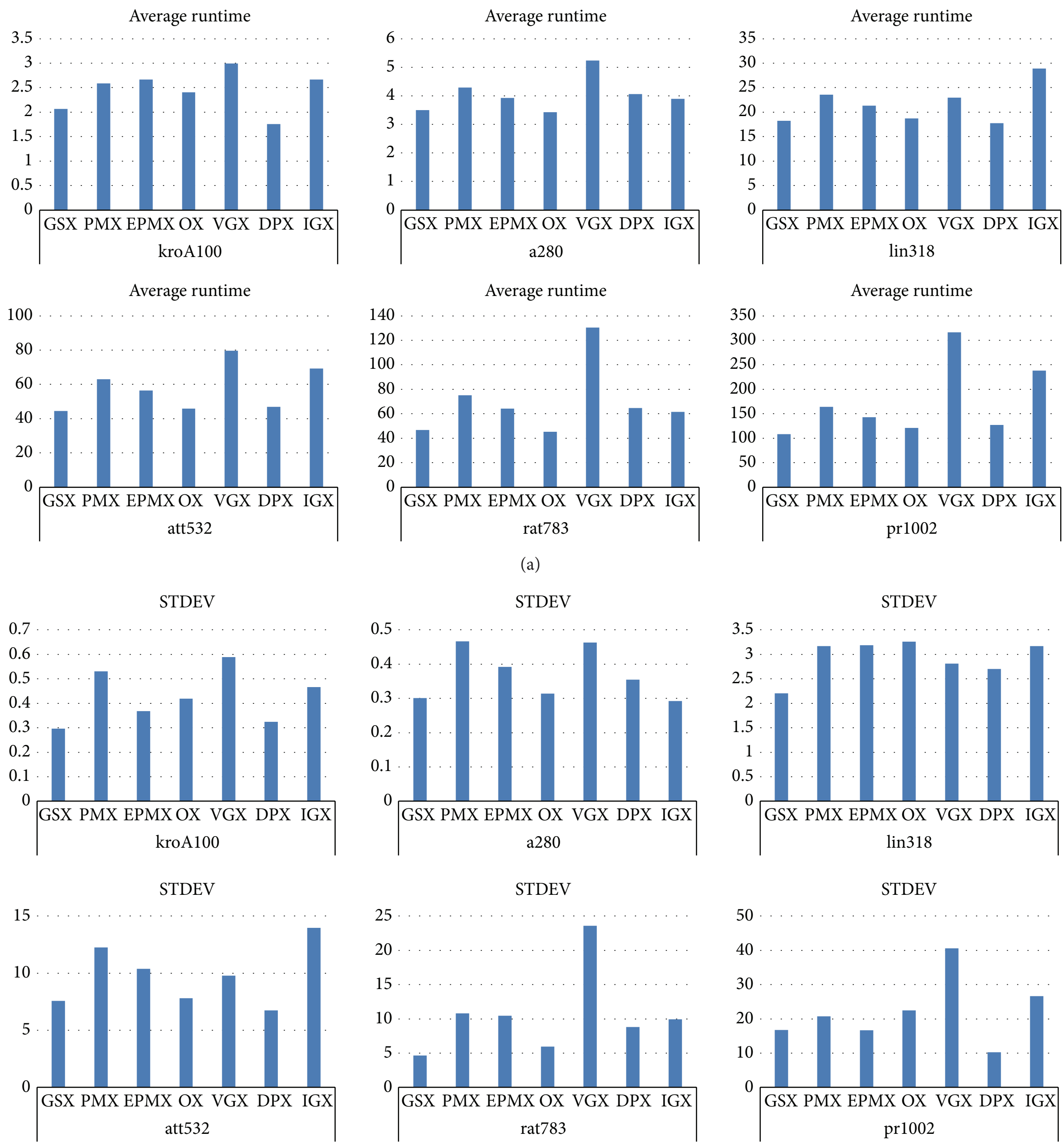


Figure 7: (a) Average required runtime and (b) STDEV of required time.

tours and third argument points to the child tour. Lines 12, 13, and 14 show OX, PMX, and EPMX which produces two children so their functions take four arguments. The first two arguments point to the parent tours and the second two arguments point to the two children tours.

The performance of these crossovers, which are based on speed and accuracy, has been analyzed in [5]. Results in [5] show that heuristic crossovers like IGX and DPX have more accuracy than others. These results also show that the crossovers like GSX family have more diversity than others which mean that these crossovers can produce wide range of different solutions.

3.3. Types of Genetic Algorithm. There are two major models for GA: generational and steady-state GA. The main deference between generational and steady-state GA is that, in

```
List<Node> ActiveNodes;// to implement “don’t-look bits”
void LK(Tour tour)
{
  for each node X
      Add X to ActiveNodes List;
  while(active node is existed)
  {
      Node N = remove and return first node in ActiveNodes;
      if(inner-LK(tour, X) <= 0)
       inactive X;
   }
 }

int inner-LK(Tour tour, Node x)
{
  Y ← neighbor of X;
  int partial-gain = |XY|;
  break XY from tour;
  Add Y to ActiveNodes List;
  for each Z ∈ candidate set of X
  {
      add YZ to tour;
       Add Z to ActiveNodes List;
       partial-gain = partial-gain + |YZ|;
       if tour is feasible (tour closing up by one edge is possible)
       {
           if (partial-gain − last added edge cost > 0 then)
           {
               close up tour;
               return partial-gain − last added edge cost;
           }
           else
           {
               int g = inner-LK(tour, x);
               if (g < 0)
               {
                     break YZ; //test another.
               }
               else
               {
                   return g;
               }
           }
       }
   }
   add XY to tour; //breaking XY was unsuccessful.
   remove Y from ActiveNodes List;
   return 0;
 }
```

Algorithm 7: Abstract pseudocodes for LK.

generational GA, new solutions are added to population and, after some steps, population size is normalized by removing worse individuals but, in steady-state GA, the new solution is replaced by one of old solution of population. In both algorithms, mutation operation may be applied on one or more solutions of population periodically.

Recently, researchers add solution improvement function such as 2- opt, 3-opt, and LK into their GA. These functions usually are applied to new solutions after they are created or changed by the crossover or mutation operations. These GAs are called memetic or hybrid GA (HGA). Memetic is general concept and points to the

```
while (some conditions are satisfied)
{
  for (i = 0; i < gen-size; i++)
  {
    Get father and mother from population;
    Child(s) <- crossover(father, mother);
    LS may be apply on child tour(s);
    Add child to population;
  }
Normalized population size by removing some solutions;
}
```

Algorithm 8: Generational GA pseudocodes.

```
for (i = 0; i < gen-size; i++)
{
  Get father and mother from population;
  Child(s) <- crossover(father,mother);
  LS may be apply on child tour(s);
  index <- get_inddex();
  population[index] <- child;
}
```

Algorithm 9: Steady-state GA pseudocodes.

```
Create population of random tours;
//First storey is GA and increases tours'quality. It uses IGX as its crossover [5].
Use steady-state GA by heuristic crossover to improve population;
Sort population according to cost in ascendant order;
Tour *best-so-far ← population[0];
//Second storey is HGA and uses GSX as its crossover, Double-Bridge as its mutation and LK
as its LS.
for (i = 1; i <= gen-size; i++)
{
  Tour *child;
  int index;
  if(rand_01() < crossover-rate)
  {
      index = linear selection from population;
      father ← population[index];
      index = linear selection from population;
      mother ← population[index];
      crossover(father, mother, child);
      Improve child by lk;
  }
  else
  {
      child←linear selection from population;
      mutate child by double-bridge;
      Improve child by lk;
  }
  Sort population according to cost in ascendant order;
  if (i % period == 0)
      update candidate set according to edges density in population;
  if(best-so-far cost > population[0] cost)
      best-so-far cost ← population[0];
}
Report best-so-far;
```

Algorithm 10: Our HGA's pseudocodes.

TABLE 2: (a) 2-Opt performance based on solution costs. (b) 3-Opt performance based on solution costs. (c) LK performance based on solution costs.

(a)

	Optimum cost	Best		Average		Worst		STDEV
		Cost	Error (%)	Cost	Error (%)	Cost	Error (%)	
att532	27686	28004	1.149	28271	2.113	28466	2.817	135.82961
rat783	8806	9001	2.214	9030.35	2.548	9079	3.1	24.598406
pr1002	259045	263489	1.716	265864.85	2.633	267891	3.415	1053.8556
rl1889	316536	362397	14.488	366151.5	15.675	370419	17.023	2649.2821
pr2392	378032	388057	2.652	390727.5	3.358	393472	4.084	1552.5029
pcb3038	137694	140534	2.063	141086.6	2.464	141533	2.788	269.47071
fnl4461	182566	186205	1.993	186608.85	2.214	187116	2.492	250.18104
rl5915	565530	612142	8.242	628403.15	11.118	642120	13.543	7820.3035
pla7397	23260728	24613322	5.815	24926239	7.16	25373211	9.082	192676.28
brd14051	469385	480038	2.27	485242.3	3.378	491392	4.688	3402.7965
d15112	1573084	1607543	2.191	1611675.8	2.453	1627080	3.432	4269.2295
d18512	645238	659009	2.134	660992.65	2.442	666349	3.272	1859.8221
pla33810	66048945	69456797	5.16	69879698	5.8	70333800	6.487	281635.59
pla85900	142382641	148589968	4.36	148751673	4.473	148868721	4.555	68119.888
usa115475	6283142	6424750	2.254	6469988.7	2.974	6504607	3.525	18159.159

(b)

	Optimum cost	Best		Average		Worst		STDEV
		Cost	Error (%)	Cost	Error (%)	Cost	Error (%)	
att532	27686	27809	0.444	27869.35	0.662	27984	1.076	46.671613
rat783	8806	8826	0.227	8860.3	0.617	8901	1.079	21.555559
pr1002	259045	261013	0.76	261966.95	1.128	263477	1.711	676.60316
rl1889	316536	353866	11.793	362309.1	14.461	369220	16.644	3502.7353
pr2392	378032	382036	1.059	385028.6	1.851	390991	3.428	2673.4602
pcb3038	137694	138687	0.721	138978.65	0.933	139684	1.445	239.93624
fnl4461	182566	183351	0.43	183628.35	0.582	183918	0.741	163.06514
rl5915	565530	614092	8.587	625596.05	10.621	640140	13.193	7676.6391
pla7397	23260728	24153629	3.839	24466610	5.184	24899249	7.044	212019.08
brd14051	469385	474243	1.035	477311.15	1.689	482885	2.876	2821.3266
d15112	1573084	1581463	0.533	1584579	0.731	1589468	1.042	2568.7097
d18512	645238	649143	0.605	650639.05	0.837	655523	1.594	1533.274
pla33810	66048945	66980982	1.411	68025924	2.993	68519773	3.741	411992.49
pla85900	142382641	144593580	1.553	144680542	1.614	144750832	1.663	45980.685
usa115475	6283142	6334053	0.81	6359277.6	1.212	6395483	1.788	19890.006

(c)

	Optimum cost	Best		Average		Worst		STDEV
		Cost	Error (%)	Cost	Error (%)	Cost	Error (%)	
att532	27686	27712	0.094	27765.2	0.286	27897	0.762	49.990104
rat783	8806	8806	0	8818.95	0.147	8833	0.307	8.9117014
pr1002	259045	260359	0.507	261630.45	0.998	265127	2.348	1434.3942
rl1889	316536	338432	6.917	352930.7	11.498	366537	15.796	6490.877
pr2392	378032	378870	0.222	381562.35	0.934	385833	2.064	1432.2577
pcb3038	137694	138010	0.229	138346.35	0.474	138604	0.661	162.46466
fnl4461	182566	182872	0.168	183035	0.257	183134	0.311	77.290974
rl5915	565530	597425	5.64	616541.85	9.02	629056	11.233	7045.3808
pla7397	23260728	24090910	3.569	24308120	4.503	24588866	5.71	140266.08
brd14051	469385	471566	0.465	474418.05	1.072	479274	2.107	1767.5788
d15112	1573084	1576636	0.226	1577681.8	0.292	1579279	0.394	669.32031

(c) Continued.

	Optimum cost	Best		Average		Worst		STDEV
		Cost	Error (%)	Cost	Error (%)	Cost	Error (%)	
d18512	645238	646770	0.237	647091.95	0.287	648439	0.496	373.13882
pla33810	66048945	67107824	1.603	67841467	2.714	68576654	3.827	361018.43
pla85900	142382641	144296674	1.344	144528830	1.507	144944342	1.799	220016.09
usa115475	6283142	6283142	0	6325336.4	0.672	6350265	1.068	23558.074

all evolutionary algorithms that incorporate local searches to improve their solutions.

4. Developing HGA by the Developed Objective Tools

To show applicability of proposed objective tools, we develop new model of HGA which differs with another versions in two main cases (See Appendices).

(1) The proposed HGA is two-storey GA. It means that the proposed HGA has been formed from two storeys of GA. First storey of GA uses heuristic genetic operator such as GX versions. This storey increases quality of population, so LSA can operate quickly in second storey. It should be tended that LSAs can operate quickly on high quality solutions. Therefore, this storey affects the second storey where LK is utilized. The LK operates quickly when it is applied to high quality tour.

(2) The second storey of the HGA is also HGA itself. Like other HGA algorithms that incorporate LSA to increase tours quality, proposed HGA also does and exploits LK as its LSA but

   (I) it is updating LK candidates' sets periodically while these instructions of storey are executing;

   (II) in order to produce wide variety of solutions it should use quick crossover operator with high diversity same as classical PMX, GSX-1, or EPMX instead of heuristic crossovers that are usually slow. Notice that LK guarantees solutions' quality so it is not reasonable to use time consumer heuristic crossovers.

5. Experiments

In this section, we show objective tools performance. We divide this section to three subsections. In first subsection, we focus on LSAs tools, in second subsection, we put forward experimental results for the crossovers, and, finally, we exhibit experimental results of HGA designated by developed objective tools.

5.1. Performance of the Developed Tools. To test developed LS tools including 2-opt, 3-opt, and LK, we apply them on fifteen TSPLIB instances twenty times. Users may need to be informed about accuracy and speed of LS tools, so here we report best, worst, average, and standard deviation of recorded costs and runtimes for LS tools per each instance in each of the twenty runs. Table 2 shows average, best, worst, and standard deviation of twenty solution costs for each instance achieved by each of the stated heuristics. Moreover, this table shows error percent of best, average, and worst solution costs which is calculated by (cost − optimum cost) × (100/optimum cost). Please consider that optimum cost for usa115475 is unknown so we have used best solution cost (6283142) that is obtained by LK tool.

Table 3 presents runtime information of each heuristic applying to each instance in twenty runs. The minimum, average, maximum, and standard deviation of required runtimes have been listed in this table.

To make comparison among heuristic tools easy, we have introduced average error percent column of Table 2 and average time column of Table 3 by diagrams in Figure 5.

5.2. Performance of the Developed Crossovers' Tools. To present crossover performance, we should show effect of crossover in GA accuracy, convergence speed, and ability of crossover in generating wide range of various solutions. To achieve these goals, we have to use generational GA because, in steady-state GA, generation size is constant but, in generational GA, the total generated solutions depend on ability of crossover in generating various solutions; if crossover can generate different solutions, so it delays generational GA convergence; then total generations increase. On the other hand, when generated solutions count is high, it shows that crossover diversity is high and it can produce wide range of various solutions. We used each of stated crossovers in generational GA to solve some instances from TSPLIB twenty times and Table 4 shows results of this experiment.

Table 4 shows information about best, worst, average, and standard deviation of solution costs for each of the twenty runs by each crossover when solving each instance. Figure 6 summarizes average error percent and STDV columns of Table 4.

In Figure 6 it can be easily seen that IGX has better performance in both average error percent and standard deviation. In average error percent and for kroA100, a280, lin318, rat783, and pr1002, IGX has first best rank and, only in att532, it has second minimum error percent. For standard deviation, also IGX has minimum in dealing with kroA100, a280, lin318, rat783, and pr1002. In solving att532, IGX has second minimum STDEV.

Table 3: (a) 2-Opt runtime results. (b) 3-Opt runtime results. (c) LK runtime results.

(a)

	Min time	Average time	Max time	STDEV
att532	0.015	0.02185	0.032	0.0077817
rat783	0.015	0.0187	0.031	0.0063254
pr1002	0.031	0.05075	0.078	0.0122039
rl1889	0.062	0.0803	0.109	0.0136617
pr2392	0.094	0.14515	0.203	0.0291047
pcb3038	0.093	0.12325	0.234	0.03034
fnl4461	0.109	0.18565	0.234	0.0349801
rl5915	0.234	0.3494	0.452	0.0488439
pla7397	0.218	0.2965	0.39	0.0448371
brd14051	0.578	0.93515	1.435	0.2534248
d15112	0.748	1.152	1.81	0.2831689
d18512	0.936	1.2106	1.904	0.2183212
pla33810	6.271	8.16745	12.543	2.2148945
pla85900	7.16	8.84205	10.343	0.8042973
usa115475	9.984	13.1663	20.951	2.6126377

(b)

	Min time	Average time	Max time	STDEV
att532	0.015	0.0281	0.047	0.0096185
rat783	0.015	0.0234	0.047	0.0095499
pr1002	0.031	0.04525	0.063	0.0101508
rl1889	0.047	0.07495	0.125	0.0156994
pr2392	0.093	0.1171	0.171	0.0227917
pcb3038	0.093	0.1303	0.156	0.0163453
fnl4461	0.156	0.20675	0.266	0.0258739
rl5915	0.188	0.2404	0.344	0.0380808
pla7397	0.172	0.2434	0.281	0.0255021
brd14051	0.733	0.9384	1.56	0.1754526
d15112	0.967	1.1318	1.248	0.0875338
d18512	0.983	1.1941	1.435	0.1246983
pla33810	1.248	1.65835	2.012	0.2394011
pla85900	3.182	3.66755	4.384	0.3271961
usa115475	9.313	11.47305	15.21	1.6807505

(c)

	Min time	Average time	Max time	STDEV
att532	0.109	0.1812	0.296	0.0440438
rat783	0.032	0.0687	0.156	0.0293493
pr1002	0.265	0.3553	0.515	0.0719716
rl1889	0.406	0.5171	0.734	0.0887841
pr2392	0.562	0.954	1.669	0.2803806
pcb3038	0.592	0.86495	1.217	0.1730511
fnl4461	0.671	1.0308	1.622	0.2252083
rl5915	0.921	1.2207	1.544	0.1467128
pla7397	1.295	1.94995	2.839	0.3957677
brd14051	3.666	5.322	6.646	0.7302357
d15112	3.541	4.3859	5.335	0.5414395
d18512	4.68	5.70405	7.909	0.711076
pla33810	12.371	16.43415	23.946	2.7950968
pla85900	23.4	28.04495	34.991	3.1503448
usa115475	37.815	46.1041	54.506	3.7665248

Table 4: Crossovers performance analysis on solution costs.

	Optimum cost	Crossover name	Best		Average		Worst		STDEV
			Cost	Error (%)	Cost	Error (%)	Cost	Error (%)	
kroA100	21282	GSX	21282	0	21282	0	21282	0	0
		PMX	21282	0	21282	0	21282	0	0
		EPMX	21282	0	21282	0	21282	0	0
		OX	21282	0	21282	0	21282	0	0
		VGX	21282	0	21282	0	21282	0	0
		DPX	21282	0	21282	0	21282	0	0
		IGX	21282	0	21282	0	21282	0	0
a280	2579	GSX	2579	0	2579	0	2579	0	0
		PMX	2579	0	2579	0	2579	0	0
		EPMX	2579	0	2579	0	2579	0	0
		OX	2579	0	2579	0	2579	0	0
		VGX	2579	0	2579	0	2579	0	0
		DPX	2579	0	2579	0	2579	0	0
		IGX	2579	0	2579	0	2579	0	0
lin318	42029	GSX	42029	0	42029	0	42029	0	0
		PMX	42029	0	42029	0	42029	0	0
		EPMX	42029	0	42029	0	42029	0	0
		OX	42029	0	42031.7	0.006	42083	0.128	12.075
		VGX	42029	0	42029	0	42029	0	0
		DPX	42029	0	42032.1	0.007	42091	0.148	13.864
		IGX	42029	0	42029	0	42029	0	0
att532	27686	GSX	27686	0	27696.55	0.038	27704	0.065	6.716
		PMX	27686	0	27699	0.047	27705	0.069	7.773
		EPMX	27686	0	27693.35	0.027	27706	0.072	8.4
		OX	27686	0	27696.85	0.039	27706	0.072	7.638
		VGX	27693	0.025	27702.1	0.058	27706	0.072	3.432
		DPX	27686	0	27697.5	0.042	27847	0.582	35.881
		IGX	27686	0	27694.1	0.029	27706	0.072	8.896
rat783	8806	GSX	8807	0.011	8811.7	0.065	8822	0.182	4.342
		PMX	8806	0	8810.5	0.051	8826	0.227	4.763
		EPMX	8807	0.011	8815.15	0.104	8828	0.25	5.274
		OX	8806	0	8810.9	0.056	8818	0.136	3.626
		VGX	8808	0.023	8813.8	0.089	8824	0.204	4.324
		DPX	8806	0	8810.3	0.049	8815	0.102	2.867
		IGX	8807	0.011	8809.25	0.037	8815	0.102	1.997
pr1002	259045	GSX	259045	0	259264.3	0.085	260066	0.394	281.477
		PMX	259045	0	259115.8	0.027	260046	0.386	237.75
		EPMX	259045	0	259093.5	0.019	259600	0.214	150.998
		OX	259045	0	259136.75	0.035	259908	0.333	233.981
		VGX	259045	0	259119.35	0.029	259949	0.349	216.241
		DPX	259045	0	259134.6	0.035	259588	0.21	134.744
		IGX	259045	0	259050.9	0.002	259099	0.021	15.758

Tables 5 and 6 show experimental results information about required runtime and total generations count in each of the twenty runs. These tables list best, average, worst, and standard deviation of required runtime and minimum, average, maximum, and standard deviation of total generations count. Average and STDEV columns of both tables have been introduced in Figure 7.

5.3. HGA Performance Analysis. Comparing developed HGA with other state-of-the-art methods is not our purpose here but we want to show that it is possible to design and develop new memetic algorithms by our objective tools. Therefore, to achieve this goal we compare HGA with latest windows based version of LKH in period of 100000 seconds in solving pla85900 that is the largest problem in TSPLIB. Diagram in

Table 5: Required runtime for generational GA with each crossover.

		Best	Average	Worst	STDEV
kroA100	GSX	1.762	2.065	2.839	0.296
	PMX	1.623	2.586	3.619	0.531
	EPMX	2.137	2.666	3.354	0.368
	OX	1.888	2.406	3.042	0.419
	VGX	2.464	2.992	4.587	0.589
	DPX	1.264	1.755	2.169	0.324
	IGX	2.106	2.665	3.916	0.467
a280	GSX	3.229	3.502	4.118	0.301
	PMX	3.9	4.289	5.038	0.467
	EPMX	3.635	3.928	4.789	0.392
	OX	2.621	3.429	4.056	0.314
	VGX	4.898	5.242	6.349	0.463
	DPX	3.76	4.063	4.82	0.355
	IGX	3.635	3.895	4.773	0.292
lin318	GSX	14.18	18.257	23.338	2.206
	PMX	16.832	23.554	28.922	3.171
	EPMX	15.818	21.304	27.301	3.186
	OX	13.588	18.713	26.894	3.262
	VGX	18.798	22.938	30.031	2.813
	DPX	13.026	17.735	23.743	2.702
	IGX	24.351	28.89	34.617	3.172
att532	GSX	32.962	44.434	60.84	7.568
	PMX	46.301	63.016	95.753	12.268
	EPMX	42.354	56.373	83.382	10.394
	OX	33.056	45.853	69.763	7.809
	VGX	64.786	79.721	101.447	9.776
	DPX	38.891	46.856	64.693	6.75
	IGX	50.731	69.304	99.185	13.961
rat783	GSX	37.456	46.628	56.301	4.659
	PMX	57.97	75.219	100.823	10.811
	EPMX	48.704	64.15	92.259	10.477
	OX	36.457	45.407	58.406	5.957
	VGX	101.26	130.538	173.285	23.599
	DPX	51.355	64.816	81.713	8.804
	IGX	50.84	61.569	92.618	9.933
pr1002	GSX	74.443	108.473	148.091	16.73
	PMX	137.483	164.007	210.804	20.737
	EPMX	121.586	142.91	176.171	16.68
	OX	83.429	120.811	180.774	22.516
	VGX	247.057	316.199	403.245	40.551
	DPX	106.158	127.228	146.437	10.286
	IGX	174.814	238.014	279.912	26.608

Table 6: Total generations count.

		Min	Average	Max	STDEV
kroA100	GSX	1500	1725	2500	302.403
	PMX	500	862.5	1250	206.394
	EPMX	750	975	1250	160.181
	OX	750	975	1250	197.017
	VGX	1500	1850	3000	432.252
	DPX	1000	1525	2000	379.577
	IGX	1500	1950	3000	394.034
a280	GSX	1500	1625	2000	222.131
	PMX	750	825	1000	117.541
	EPMX	750	800	1000	102.598
	OX	500	775	1000	111.803
	VGX	1500	1575	2000	183.174
	DPX	1500	1600	2000	205.196
	IGX	1500	1550	2000	153.897
lin318	GSX	3000	4025	5500	617.188
	PMX	1250	1887.5	2500	319.076
	EPMX	1250	1862.5	2500	339.068
	OX	1250	1912.5	3000	415.767
	VGX	2500	3250	4500	550.12
	DPX	2500	3575	5500	693.485
	IGX	4000	4875	6000	646.346
att532	GSX	3000	4425	6500	892.586
	PMX	1750	2487.5	3750	509.612
	EPMX	1750	2512.5	4250	620.245
	OX	1500	2337.5	3750	501.806
	VGX	4000	5150	7000	812.728
	DPX	3500	4375	7500	1024.374
	IGX	3500	5250	8000	1208.522
rat783	GSX	3500	4300	5500	571.241
	PMX	2000	2925	4250	544.711
	EPMX	2000	2687.5	4000	479
	OX	1750	2287.5	3000	337.122
	VGX	5500	7400	10000	1391.705
	DPX	3500	4825	6500	892.586
	IGX	3500	4500	7000	842.927
pr1002	GSX	5000	8950	14500	2199.88
	PMX	3750	4650	6000	640.723
	EPMX	3750	4575	6000	702.907
	OX	3500	4887.5	7750	1119.431
	VGX	11000	13750	19000	1956.77
	DPX	5500	7175	9000	831.533
	IGX	10000	14400	17500	1923.538

Figure 8 shows result of this competition. This diagram shows that HGA produces better tours than LKH during 100000(s) and its prominence is noticeable.

6. Conclusion

In this paper, we present highlight of our TSP programming tools that have been based on LKH implementation. In fact, these tools are source codes in three object-oriented languages: C++, C#, and JAVA. These tools can help engineers, researchers, and those who are dealing with TSP to write and develop their TSP applications more easily by one of the stated programming languages arbitrarily. Here, we tried to show our tools' performance by experiments. In order to show their applicability, we designed a hybrid algorithm that

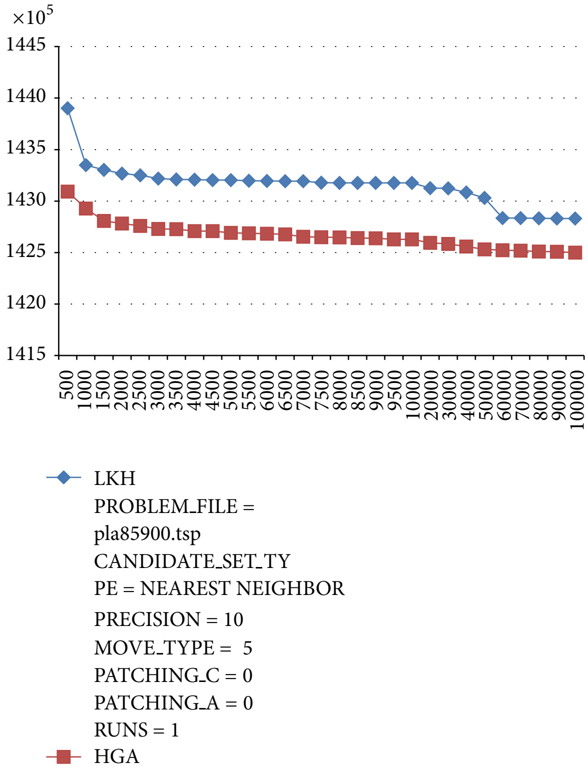


Figure 8: Competition between HGA and LKH during 100000 seconds.

was effective and beat the LKH-2 software in dealing with largest TSPLIB instance.

Appendices

A.

See Algorithm 7.

B.

See Algorithms 8 and 9.

C.

See Algorithm 10.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] K. Helsgaun, "General k-opt submoves for the Lin–Kernighan TSP heuristic," *Mathematical Programming Computation*, vol. 1, no. 2-3, pp. 119–163, 2009.

[2] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Implementation of an effective hybrid GA for large-scale traveling salesman problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, no. 1, pp. 92–99, 2007.

[3] O. Martin, S. W. Otto, and E. W. Felten, "Large-step Markov chains for the traveling salesman problem," *Complex Systems*, vol. 5, no. 3, pp. 299–326, 1991.

[4] Y. Nagata and S. Kobayashi, "A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem," *Informs Journal on Computing*, vol. 25, no. 2, pp. 346–363, 2013.

[5] H. Ismkhan and K. Zamanifar, "Developing improved greedy crossover to solve symmetric traveling salesman problem," *International Journal of Computer Science Issues*, vol. 9, no. 4, pp. 121–126, 2012.

[6] M. Albayrak and N. Allahverdi, "Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms," *Expert Systems with Applications*, vol. 38, no. 3, pp. 1313–1320, 2011.

[7] A. Chowdhury, A. Ghosh, S. Sinha, S. Das, and A. Ghosh, "A novel genetic algorithm to solve travelling salesman problem and blocking flow shop scheduling problem," *International Journal of Bio-Inspired Computation*, vol. 5, no. 5, pp. 303–314, 2013.

[8] X. Geng, Z. Chen, W. Yang, D. Shi, and K. Zhao, "Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search," *Applied Soft Computing Journal*, vol. 11, no. 4, pp. 3680–3689, 2011.

[9] S. M. Chen and C. Y. Chien, "Solving the traveling salesman problem based on the genetic simulated annealing ant colony system with particle swarm optimization techniques," *Expert Systems with Applications*, vol. 38, no. 12, pp. 14439–14450, 2011.

[10] D. Karaboga and B. Görkemli, "A combinatorial Artificial Bee Colony algorithm for traveling salesman problem," in *Proceedings of the International Symposium on Innovations in Intelligent Systems and Applications (INISTA '11)*, pp. 50–53, Istanbul, Turkey, June 2011.

[11] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[12] J. B. Escario, J. F. Jimenez, and J. M. Giron-Sierra, "Ant colony extended: experiments on the travelling salesman problem," *Expert Systems with Applications*, vol. 42, no. 1, pp. 390–410, 2015.

[13] G. Dong, W. W. Guo, and K. Tickle, "Solving the traveling salesman problem using cooperative genetic ant systems," *Expert Systems with Applications*, vol. 39, no. 5, pp. 5006–5011, 2012.

[14] D. Whitley, D. Hains, and A. Howe, "A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover," in *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature*, 2010.

[15] K. Helsgaun, "An effective implementation of the Lin-Kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[16] B. Freisleben and P. Merz, "Genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems," in *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC '96)*, pp. 616–621, May 1996.

[17] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[18] K. Helsgaun, "LKH source codes," 2009, http://www.akira.ruc.dk/~keld/research/LKH.

[19] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: a review of representations and operators," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 129–170, 1999.

[20] Z. Tao, "TSP problem solution based on improved genetic algorithm," in *Proceedings of the 4th International Conference on Natural Computation (ICNC '08)*, pp. 686–690, October 2008.

[21] B. A. Julstrom, "Very greedy crossover in a genetic algorithm for the traveling salesman problem," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 324–328, February 1995.

[22] H. Ismkhan and K. Zamanifar, "Study of some recent crossovers effects on speed and accuracy of genetic algorithm, using symmetric travelling salesman problem," *International Journal of Computer Applications*, vol. 80, no. 6, pp. 1–6, 2013.

[23] S. S. Ray, S. Bandyopadhyay, and S. K. Pal, "Genetic operators for combinatorial optimization in TSP and microarray gene ordering," *Applied Intelligence*, vol. 26, no. 3, pp. 183–195, 2007.

[24] A. Takeda, S. Yamada, K. Sugawara, I. Yoshihara, and K. Abe, "Optimization of delivery route in a city area using genetic algorithm," in *Proceedings of the 4th International Symposium on Artificial Life and Robotics*, 1999.

[25] J. L. Bentley, "K-d trees for semidynamic point sets," in *Proceedings of the 6th Annual Symposium on Computational Geometry*, pp. 187–197, June 1990.

[26] J. L. Bentley, "Experiments on traveling salesman heuristics," in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[27] H. Ismkhan and K. Zamanifar, "Using ants as a genetic crossover operator in GLS to solve STSP," in *Proceedings of the International Conference of Soft Computing and Pattern Recognition (SoCPaR '10)*, pp. 344–348, Paris, France, December 2010.

[28] B. M. Ombuki and M. Ventresca, "Local search genetic algorithms for the job shop scheduling problem," *Applied Intelligence*, vol. 21, no. 1, pp. 99–109, 2004.

[29] Y. M. Wang, H. L. Yin, and J. Wang, "Genetic algorithm with new encoding scheme for job shop scheduling," *The International Journal of Advanced Manufacturing Technology*, vol. 44, no. 9-10, pp. 977–984, 2009.

[30] F. T. Hanshar and B. M. Ombuki-Berman, "Dynamic vehicle routing using genetic algorithms," *Applied Intelligence*, vol. 27, no. 1, pp. 89–99, 2007.

[31] N. Yalaoui, H. Mahdi, L. Amodeo, and F. Yalaoui, "A new approach for workshop design," *Journal of Intelligent Manufacturing*, vol. 22, no. 6, pp. 933–951, 2011.

[32] A. S. Ramkumar, S. G. Ponnambalam, N. Jawahar, and R. K. Suresh, "Iterated fast local search algorithm for solving quadratic assignment problems," *Robotics and Computer-Integrated Manufacturing*, vol. 24, no. 3, pp. 392–401, 2008.

[33] K. S. Goh, A. Lim, and B. Rodrigues, "Sexual selection for genetic algorithms," *Artificial Intelligence Review*, vol. 19, no. 2, pp. 123–152, 2003.

[34] D. Whitley, "The genitor algorithm and selective pressure: why rank-based allocation of reproductive trials is best," in *Proceedings of the 3rd International Conference on Genetic Algorithm*, 1989.

[35] S. Hiroaki and I. Yoshihara, "A fast TSP solver using GA on java," in *Proceedings of the 3rd International Symposium on Artificial Life and Robotics*, 1999.

[36] N. H. Dinh, Y. Ikuo, Y. Kunihito, and Y. Moritoshi, "Greedy genetic algorithms for symmetric and asymmetric TSPs," *Information Processing Society of Japan, Transactions on Mathematical Modeling and Its Applications*, vol. 43, no. 10, pp. 165–175, 2002.

[37] J. Majumdar and A. K. Bhunia, "Genetic algorithm for asymmetric traveling salesman problem with imprecise travel times," *Journal of Computational and Applied Mathematics*, vol. 235, no. 9, pp. 3063–3078, 2011.