

---

# Memetic Algorithm for the Generalized Asymmetric Traveling Salesman Problem

Gregory Gutin<sup>1</sup>, Daniel Karapetyan<sup>1</sup>, and Natalio Krasnogor<sup>2</sup>

<sup>1</sup> Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK, [gutin@cs.rhul.ac.uk](mailto:gutin@cs.rhul.ac.uk), [daniel.karapetyan@gmail.com](mailto:daniel.karapetyan@gmail.com)

<sup>2</sup> Automatic Scheduling and Planning group, School of Computer Science and I.T., University of Nottingham, Nottingham, NG8 1BB, UK, [Natalio.Krasnogor@nottingham.ac.uk](mailto:Natalio.Krasnogor@nottingham.ac.uk)

**Summary.** The generalized traveling salesman problem (GTSP) is an extension of the well-known traveling salesman problem. In GTSP, we are given a partition of cities into groups and we are required to find a minimum length tour that includes exactly one city from each group. The aim of this paper is to present a new memetic algorithm for GTSP which clearly outperforms the state-of-the-art memetic algorithm of Snyder and Daskin [21] with respect to the quality of solutions. Computational experiments conducted to compare the two heuristics also show that our improvements come at a cost of longer running times, but the running times still remain within reasonable bounds (at most a few minutes). While the Snyder-Daskin memetic algorithm is designed only for the Symmetric GTSP, our algorithm can solve both symmetric and asymmetric instances. Unlike the Snyder-Daskin heuristic, we use a simple machine learning approach as well.

## 1 Introduction

The *generalized traveling salesman problem* (GTSP) is defined as follows. We are given a weighted complete directed or undirected graph  $G$  and a partition  $V = V_1 \cup V_2 \cup \dots \cup V_c$  of its vertices; the subsets  $V_i$  are called *clusters*. The objective is to find a minimum weight cycle containing exactly one vertex from each cluster. Such a cycle is called a *GTSP tour*. There are many publications on GTSP (see, e.g., the surveys [5, 7] and the references there) and the problem has many applications, see, e.g., [3, 14]. The problem is NP-hard, since a special case of GTSP is the *traveling salesman problem* (TSP) when  $|V_i| = 1$  for each  $i$ . We call GTSP and TSP *Symmetric* if the complete graph  $G$  is undirected and *Asymmetric* if  $G$  is directed. Often instead of the term weight we use the term *length*.

A collection of vertex-disjoint cycles such that each cluster has only one vertex in the collection is called a *cycle cover*. It is well-known [10] that the

problem of finding a minimum weight cycle cover is polynomial-time solvable if each cluster has just one vertex. However, even when each cluster has just two vertices, the minimum weight cycle cover problem is NP-hard [8]. Since all tours of GTSP and TSP are special cycle covers (consisting of just one cycle), the above difference in complexities, which we call the *cycle cover complexity gap*, indicates that GTSP is somewhat harder than TSP.

Various approaches to GTSP have been studied. There are exact algorithms such as branch-and-bound and branch-and-cut algorithms in [6]. While exact algorithms are very important, they are unreliable with respect to their running time that can easily reach many hours or even days. For example, the well-known TSP solver CONCORDE could easily solve some TSP instances with several thousand cities, but it could not solve several asymmetric instances with 316 cities within the time limit of  $10^4$  sec. (in fact, it appears it would fail even if significantly much more time was allowed) [6].

Several researchers use transformations from GTSP to TSP [3] as there exists a large variety of exact and heuristic algorithms for the TSP, see, e.g., [9, 15]. However, while the known transformations normally allow to produce optimal GTSP tours from the obtained optimal TSP tours, all known transformations do not preserve suboptimal solutions. Moreover, conversions of near-optimal TSP tours may well result in infeasible GTSP solutions. Thus, there is necessity for specific GTSP heuristics. Not every TSP heuristic can be extended to GTSP; for example, so-called subtour patching heuristics often used for the Asymmetric TSP, see, e.g., [10], cannot be extended to GTSP due to the above-mentioned cycle cover complexity gap.

It appears that the only metaheuristic algorithms that can compete with Lin-Kirnighan-based local search for TSP are memetic algorithms cf. [16, 20] that combine powers of genetic and local search algorithms [11, 23]. Thus, it is no coincidence that the current state-of-the-art GTSP heuristic is a memetic algorithm of Snyder and Daskin [21].

The aim of this paper is to present a new memetic algorithm for GTSP that clearly outperforms the Snyder-Daskin heuristic with respect to the quality of solutions. Unlike the Snyder-Daskin heuristic that can be used only for the Symmetric GTSP, our algorithm can be used for both Symmetric and Asymmetric GTSPs. Unlike Snyder and Daskin [21], following [12, 13, 18] we use a simple machine learning approach that makes our heuristic significantly more robust. The computational experiments conducted to compare our heuristic with that of Snyder and Daskin show that our improvements come at a cost of longer running times, but the running times still remain within reasonable bounds (at most a few minutes). Notice that longer running times can hardly improve solution produced by the Snyder-Daskin heuristic due to a rather limited power of the heuristic's local search.

## 2 General Scheme, Solution Coding and Stopping Criterion

Our heuristic is a memetic algorithm, which combines powers of a genetic algorithm with that of a local search. A simple machine learning approach is also used. We start with a general scheme of our heuristic, which is similar to the general schemes of many memetic algorithms.

- Step 1, *initialize*. Construct an initial population of 300 solutions using fast construction heuristics.
- Step 2, *improve*. Use local search to replace the 300 solutions in the population by 300 local optima. Among all equivalent solution eliminate all but one. As a result, we obtain the first generation population of  $p \leq 300$  solutions.
- Step 3, *produce next generation*. Use reproduction, crossover and mutation genetic operators to produce the non-optimized next generation of  $m$  solutions.
- Step 4, *improve next generation*. Use local search to replace the  $m$  solutions by  $m$  local optima. Among all equivalent solution eliminate all but one. As a result, we obtain the next generation population of  $p \leq m$  solutions.
- Step 5, *evolute*. Repeat Steps 3-4 until a stopping criterion is satisfied.

Part of our heuristic is a *genetic algorithm* and one of the most important issues in the design of a genetic algorithm is *solution coding* choice. Coding is a process of converting a feasible solution into *chromosome*, a sequence of so-called *genes*. Gene is an atom for genetic operators. Usually gene is represented by one number.

The most traditional and natural coding for TSP is to use the vertex sequence of a cycle as a chromosome. This method can be used for GTSP unchanged or it can be easily modified: each gene may contain two numbers, the cluster number and the vertex number within the cluster. However, such coding is not preserved by many simple and useful genetic operators [2, 21]. Thus, we adopt the random keys method of Bean [2] (this method was used by Snyder and Daskin [21] as well). Random key coding is based on nonnegative real numbers. Each  $i$ 'th number codes vertex in  $i$ 'th cluster. The order of clusters is determined by the fractional part of the numbers: the smaller fractional part is, the earlier in the tour the corresponding cluster is. The integer part of each number determines which vertex of the cluster is in the solution (cycle).

Random keys require only one restriction for correct solution coding: the integer part of the  $i$ 'th number should be no larger than  $|V_i|$ , where  $V_i$  is the set of vertices in cluster  $i \in \{1, 2, \dots, c\}$ . The Bean's random keys method allows us to change the genes arbitrary subject to the constraints above and makes possible to use standard genetic operators in the algorithm. The main random keys disadvantage is the fact that the decoding procedure is difficult: it is necessary to explore full chromosome even to detect neighbors of a given

vertex in the cycle. However the decoding procedure takes  $O(c \ln c)$  time as it needs gene sorting. We always store both coded and decoded solutions: the genetic operators use coded chromosome, and the decoded solution is used to have a fast access to the information about cluster sequence.

A generation is called *idle* if the best solution in this generation has the same length as the best solution in the previous generation. Our algorithm terminates when the following two conditions are satisfied: there have been at least  $c$  generations before the current one, and  $n_{\text{idle}} > \min\{0.025c \cdot n_{\text{idlemax}}, 2c\}$ , where  $n_{\text{idle}}$  is the current number of consequent idle generations and  $n_{\text{idlemax}}$  is the maximum number of consequent idle generations ever happened. The Snyder-Daskin memetic algorithm terminates after 10 consequent idle generations.

### 3 First Generation and Genetic Operators

#### 3.1 First Generation

We use 300 tries to produce the first generation. Each try runs the nearest neighbor or random construction heuristics and then optimizes the solution using the improvement procedure, which is explained in the next section.

The *nearest neighbor heuristic (NN)* is a natural modification of well-known TSP construction heuristic with the same name. NN chooses a random vertex at first and then it searches for the nearest vertex among unused clusters in each iteration till all clusters have been used.

The *random heuristic* generates a solution (cycle) with random cluster order and a random vertex in each cluster. Obviously this heuristic usually produces cycles far from optimal. However, it is fast and its cycles are without any regularity. The latter is important as each deterministic heuristic can cause solutions uniformity and as a result some solution branches can be lost.

#### 3.2 Next Generations

Each generation except the first one is based on the previous generation. To produce next generation one uses genetic operators, which are algorithms that construct a solution or two from one or two so-called parent solutions. In our heuristic, only one solution is produced from one or two parent solutions. Parent solutions are chosen from the previous generation using some *selection strategy*. In our algorithm, some reproduction, crossover, and mutation genetic operators are employed. We perform  $r$  tries for reproduction, 60 tries for crossover, and 60 tries for mutation operator, where  $r$  is not a constant and it depends on the previous generation. As a result, we obtain at most  $r + 120$  solutions in each generation but the first one.

### 3.3 Reproduction

*Reproduction* is a process of simply copying solutions from the previous generation. Reproduction operator requires a selection strategy to select the solutions from the previous generation to be copied. Before applying our reproduction operator we order the solutions of the previous generation from the best to the worst such that the first solution is the best. We copy the  $\min\{n, 10\}$  shortest solutions from the previous generation to the current one, where  $n$  is the number of solutions in the previous generation. Now we determine the number  $b$  of solutions in the previous generation that have the minimum length. We choose  $\lfloor b/2 \rfloor$  more solutions to copy using some selection strategies. Notice that a solution can be selected more than once for reproduction and this is acceptable as we use a nondeterministic improvement procedure, i.e., its results may well differ even if we start from the same solution. We use the following selection strategies:

- *Random* strategy selects solution randomly and uniformly.
- *Elitist* strategy chooses the solution index as  $i = knr^2 + 1$ , where  $n$  is the previous generation size,  $r$  is a random number,  $r \in [0, 1)$ , and  $k$  is the strategy parameter that specifies the maximum value of  $i$ . We use  $k \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ .

The random strategy is applied with probability  $1/6$ , the same holds for the elitist strategy with fixed  $k$ .

### 3.4 Crossover

A *crossover* operator combines two different solutions from the previous generation. If the current generation has at most four solutions, we do not use crossover. Otherwise, we use one algorithm for crossover with different values of its parameter and several selection strategies to choose parent solutions. In our heuristic  $\rho$  is chosen randomly and uniformly from the set

$$\{0.02, 0.03, 0.05, 0.08, 0.1, 0.15, 0.2, 0.3, 0.5\}$$

and a selection strategy is also chosen randomly and uniformly from the five strategies described below (the elitists strategies with different value of  $k$  are considered to be distinct strategies). Before applying crossover we rotate one of the parents to superpose the parent cycles by the first cluster. Otherwise the crossover would have no meaning.

We use the *uniform* crossover algorithm [22]. It combines two parents using random crossover mask, i.e.,  $r_i = p'_i$  with probability  $\rho$  and  $r_i = p''_i$  with probability  $1 - \rho$ , where  $r$  is the target chromosome and  $p'$  and  $p''$  are parents.

We use the following selection strategies:

- *Random* strategy takes two different random solutions from the previous generation.

- *Elitist* strategy chooses randomly two different solutions among the best  $\beta$  solutions from the previous generation, where  $\beta = \lfloor kn \rfloor$ , where  $k$  is the strategy parameter. We use  $k \in \{0.2, 0.4\}$ .
- *Near* strategy chooses solutions with the similar weight function values. In practice it is required just to take solutions with indices close to each other as we store solutions sorted by weight for each generation. So the strategy generates random solution pairs until  $0 < |i_1 - i_2| < dn$ , where  $i_1$  and  $i_2$  are the generated indices,  $n$  is the previous generation size, and  $d$  is the strategy parameter. We use  $d = 0.3$ .
- *Far* strategy works very similar to near strategy except for the condition:  $|i_1 - i_2| > dn$ . We use  $d = 0.3$ .

### 3.5 Mutation

A *mutation* operator changes partially one solution  $x^{\pi(1)}x^{\pi(2)} \dots x^{\pi(c)}x^{\pi(1)}$  ( $x^i \in V_i$ ) from the previous generation. We choose solutions to perform mutation using either random strategy or the elitist strategy from Subsection 3.3; the probability of using either strategy is 0.5. The elitist strategy parameter value is chosen with the uniform probability from the set  $\{0.1, 0.2, 0.3, 0.4, 0.5\}$ .

We consider the same mutation algorithm with different values of its parameter as distinct mutation strategies. Different mutation and selection strategies are required as each of them is good only for part of instance types. This approach slows the algorithm down, but allows us to make it universal. We use the following mutations:

- *Insert* mutation chooses a random vertex  $x^{\pi(i)}$  in the cycle

$$x^{\pi(1)}x^{\pi(2)} \dots x^{\pi(c)}x^{\pi(1)},$$

deletes it from the cycle and reinserts it into the cycle

$$x^{\pi(1)}x^{\pi(2)} \dots x^{\pi(i-1)}x^{\pi(i+1)} \dots x^{\pi(c)}x^{\pi(1)}$$

in a random place. The mutation is performed  $t$  times for each mutation run, where  $t$  is chosen randomly and uniformly between 1, 2, 3, 5, and 10.

- *Local Insert* mutation considers only a fragment  $x^{\pi(i)}x^{\pi(i+1)} \dots x^{\pi(j)}$  of the cycle. The fragment starts at a random vertex  $x^{\pi(i)}$  and contains  $\lfloor a \cdot c \rfloor$  vertices, where  $c$  is the number of clusters (i.e., vertices in the cycle) and  $a \in \{0.05, 0.1, 0.2, 0.3, 0.5\}$ . For each vertex  $x^{\pi(k)}$ ,  $i \leq k \leq j$  with probability 0.2 we perform the following: remove the vertex  $x^{\pi(k)}$  and insert it into the fragment

$$x^{\pi(i)} \dots x^{\pi(k-1)}x^{\pi(k+1)} \dots x^{\pi(j)}$$

in a random place.

- $k$ -opt mutation splits the tour into  $k$  chains (i.e., fragments) and combines them in a new random order. It also reverses each chain with probability 0.5. We use  $k \in \{4, 5\}$ .

The probability of taking a mutation from the list together with a particular value of its parameter ( $t$ ,  $a$  or  $k$ ) is  $\frac{1}{15}$  apart from two exceptions: the probability of choosing Insert with  $t = 1$  ( $t = 2$ ) is  $\frac{3}{15}$  ( $\frac{2}{15}$ ).

### 3.6 Differences with Snyder-Daskin Heuristic

Our and the Snyder-Daskin heuristics are both memetic algorithms and as such they have many similarities. However, there are many differences that allow our algorithm to perform significantly better. The most important differences are in the improvement part of our heuristic, but there are many differences in the genetic part as well. Now we will list the main differences in the genetic parts of the two heuristics.

Our first generation contains up to 300 solutions and all other generations have at most  $r + 120$  solutions ( $r$  was defined above). Each generation in [21] contains 100 solutions. To produce the first generation we use nearest neighbor and random construction heuristics. In [21] only random heuristic is used. We use reproduction, crossover, and mutation genetic operators; in [21] reproduction, crossover, and new random tours generation are used. In [21] reproduction operator just copies best 20 solutions from the previous generation. No improvement procedure is applied to these solutions as they were optimized once and the second run of the improvement procedure can not do any optimization. In our heuristic optimization is performed any time a new solution is added to the next generation. This is because our improvement procedure is nondeterministic. The termination criterion in our and the Snyder-Daskin heuristics are completely different.

## 4 Local Improvement Part

We use a local improvement procedure for each solution added to the current generation. Snyder and Daskin [21] use a deterministic improvement procedure where the solution is optimized with 2-opt and swap heuristics. The swap heuristic chooses a cluster  $V_j$ , removes the vertex  $x^{\pi(i)}$  of  $V_j$  ( $j = \pi(i)$ ) from the current cycle  $x^{\pi(1)}x^{\pi(2)} \dots x^{\pi(c)}x^{\pi(1)}$  and inserts a vertex from  $V_j$  into the new cycle  $C' = x^{\pi(1)}x^{\pi(2)} \dots x^{\pi(i-1)}x^{\pi(i+1)} \dots x^{\pi(c)}x^{\pi(1)}$ . The insertion is done to minimize the weight of the obtained cycle (all vertices of  $V_j$  and all possibilities of their insertion into  $C'$  are considered). The 2-opt and swap are repeated one after another several times depending on the length of the initial solution that is optimized.

Unlike the local improvement procedure [21], our improvement procedure is nondeterministic. It has several iterations and each iteration applies a certain improvement heuristic with a certain value of its parameter (if it has one)

to the given solution. We have implemented several improvement heuristics and each time we choose one of them (together with a value of its parameter) randomly. The particular heuristic choice probability depends on its previous results as follows. We partition all possible solution lengths into intervals  $[b_i, b_{i+1})$ ,  $i = 0, 1, \dots$ , where  $b_i = 10 \times 1.1^i$ . For each improvement heuristic and for each length's interval, we store the total running time (measured in processor ticks), run number, and total improvement. Each time an improvement heuristic finishes its work, we do the following for the corresponding stored values: (i) increase the run number by one, (ii) add the heuristic running time to its total running time, (iii) add  $l_{\text{before}} - l_{\text{after}}$  to the total improvement, where  $l_{\text{after}}$  and  $l_{\text{before}}$  are the solution lengths after and before the iteration.

We calculate the heuristic quality (for the given length's interval) as the total improvement divided by the total running time. If we have run the particular heuristic for the particular length's interval at most twice, we set the quality to be 1. (While there is not enough statistics gathered we suppose that the heuristic quality is very high.) The particular heuristic choice probability is proportional to  $q + Q/10$ , where  $q$  is the quality of the heuristic for the particular length's interval and  $Q$  is the average heuristic quality for the particular length's interval.

The improvement procedure makes use of the following heuristics:

- *Best Swap* tries to swap each vertex pair in the given solution and applies the best swap found if it improves the tour.
- *First Swap* does the same as Best Swap does, but it stops when the first vertex pair that improves the solution is found.
- *Random Swap* does the same as First Swap does but it chooses the vertex pairs randomly and the iteration number is a parameter with values 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000. Random Swap applies all the improvements found, not just the first one.
- *Neighbor Swaps* does the same as First Swap does but it considers only neighbor vertex pairs and it applies all the improvements found, not just the first one.
- *Insert* heuristic removes one vertex from the given solution and inserts it into another place. The insertion is performed only if it leads to a shorter cycle. The vertex and insertion place are chosen randomly. We use the Insert heuristic for  $i \in \{5, 10, 20, 50, 100, 500\}$  tries in a run.
- *k-opt* (similarly to the standard *k-opt* TSP heuristic, see, e.g., [10, 11]) divides randomly the given solution into  $k$  chains and searches the best permutation of them. As we consider both symmetric and asymmetric instances, *k-opt* heuristic tries also both directions for each chain. We use *k-opt* for  $k \in \{2, 3, 4, 5, 6\}$ .
- *3-opt* is a variation of *k-opt* with  $k = 3$  except that it first selects  $s \in \{5, 10, 20, 30, 50\}$  positions in the cycle and tries all possible combinations of splitting the cycle by these positions into three chains. The directions of the chains are not changed.

- *Direct-k-opt* does the same as *k-opt* does, but it first sorts all arcs/edges of the cycle in the nonincreasing order of their weights  $e_1, e_2, \dots, e_c$  ( $w(e_i) \leq w(e_j)$  for each  $i > j$ ) and deletes arcs/edges  $e_{\tau(1)}, e_{\tau(2)}, \dots, e_{\tau(k)}$ , where  $\tau(1) = 1$  and  $\tau(i) = \tau(i-1) + \xi$  such that  $\xi = 1$  or  $2$  with probability  $1/2$  each.
- *Full Vertex in Cluster* optimization uses the shortest  $(s, t)$ -path algorithm for acyclic digraphs (see, e.g., [1]) to find the best vertex for each cluster when the order of clusters is fixed. This heuristic was introduced by Fischetti, Salazar-Gonzalez and Toth [6] (see its detailed description also in [5]).
- *Fragment Vertex in Cluster* optimization does the same as Full Vertex in Cluster optimization does, but it considers only a fragment of the given tour. The fragment's first cluster is chosen randomly and uniformly and the fragment's length  $\ell$  is a parameter of the heuristic such that  $\ell = \min\{c, \ell'\}$ , where  $\ell' \in \{5, 10, 15, 20, 30, 40, 50\}$ .

## 5 Results of Computational Experiments

We tested our heuristic using GTSP instances from [24] which were generated from some TSPLIB instances by applying the clustering procedure of Fischetti, Salazar and Toth [6]. The GTSP instances in [24] are both symmetric and asymmetric. Snyder and Daskin [21] used all symmetric instances from [24], but two, that allows us to compare our heuristic with theirs. It is rather unfortunate that results for the instances 46gr226 and 87gr431 are not used in [21] since our computational experiments show that the two instances are significantly harder than the rest of the symmetric instances. Since our heuristic is designed for medium and large instances in mind, we only used instances with at least 40 clusters, but we used all such instances (including the two mentioned above).

The computer we used in our experiments has Pentium D processor with 2.8 GHz frequency. The processor used in [21] is Pentium IV 3.2 GHz. We may consider the test computers as equivalent. However, the computer languages used for our heuristic and the Snyder-Daskin heuristic are different: while they used C++, we coded our heuristic using the high-level C# language (the use of C# has simplified coding). However, C# programs are usually slower than C++ programs. The most important language instructions in coding both heuristics are integer arithmetic and floating point arithmetic operations, single loops, and nested loops. According to several works (see, e.g., [4]) these instructions are 1.1 to 4 times slower in C# than in C++. Thus, we can use the following conservative estimate: if we coded our heuristic in C++, we would speed up the computations two times.

The table below shows our test results. Each test includes five algorithm runs. The headings are as follows:

Opt Optimal objective value for the instance.

# Opt The number of tests, out of five, in which the heuristic have found the optimal value.

Min, Mean, and Max Value The minimum, average, and maximum solution lengths.

Min, Mean, and Max Error% The minimum, average, and maximum values of errors in per cent. The error is calculated as follows:  $\frac{value-opt}{opt} \times 100\%$ .

SL The ‘short list’, i.e., the set of instances on which both the Snyder-Daskin heuristic and our heuristic have been tested.

Every odd row shows results obtained by our heuristic and every even row presents results obtained by the Snyder-Daskin heuristic. Note that 40d198 means that the instance has 198 vertices and 40 clusters.

Instance	Opt	# Opt	Min		Mean		Max	
			Value	Error %	Value	Error %	Value	Error %
40d198 (euc2d)	10557	5	10557	0.00	10557	0.00	10557	0.00
from [21]		5	10557	0.00	10557	0.00	10557	0.00
40kroa200 (euc2d)	13406	5	13406	0.00	13406	0.00	13406	0.00
from [21]		5	13406	0.00	13406	0.00	13406	0.00
40krob200 (euc2d)	13111	5	13111	0.00	13111	0.00	13111	0.00
from [21]		4	13111	0.00	13112	0.01	13114	0.02
45ts225 (euc2d)	68340	4	68340	0.00	68352	0.02	68400	0.09
from [21]		4	68340	0.00	68352	0.02	68400	0.09
46gr229 (geo)	71641	0	71972	0.46	71972	0.46	71972	0.46
from [21]			—	—	—	—	—	—
46pr226 (euc2d)	64007	5	64007	0.00	64007	0.00	64007	0.00
from [21]		5	64007	0.00	64007	0.00	64007	0.00
53gil262 (euc2d)	1013	3	1013	0.00	1016.2	0.32	1021	0.79
from [21]		0	1014	0.10	1021	0.79	1025	1.18
53pr264 (euc2d)	29549	5	29549	0.00	29549	0.00	29549	0.00
from [21]		5	29549	0.00	29549	0.00	29549	0.00
60pr299 (euc2d)	22615	5	22615	0.00	22615	0.00	22615	0.00
from [21]		0	22620	0.02	22639	0.11	22677	0.27
64lin318 (euc2d)	20765	5	20765	0.00	20765	0.00	20765	0.00
from [21]		2	20765	0.00	20894	0.62	21026	1.26
65rbg323 (explicit)	471	4	471	0.00	471.4	0.08	473	0.42
from [21]			—	—	—	—	—	—
72rbg358 (explicit)	693	5	693	0.00	693	0.00	693	0.00
from [21]			—	—	—	—	—	—
80rd400 (euc2d)	6361	1	6361	0.00	6385.4	0.38	6408	0.74
from [21]		0	6416	0.86	6436	1.18	6448	1.37
81rbg403 (explicit)	1170	5	1170	0.00	1170	0.00	1170	0.00
from [21]			—	—	—	—	—	—
84fl417 (euc2d)	9651	5	9651	0.00	9651	0.00	9651	0.00
from [21]		0	9654	0.03	9656	0.05	9658	0.07
87gr431 (geo)	101523	0	101946	0.42	102404	0.87	103097	1.55
from [21]			—	—	—	—	—	—
88pr439 (euc2d)	60099	2	60099	0.00	60178	0.13	60230	0.22
from [21]		0	60100	0.00	60258	0.26	60492	0.65
89pcb442 (euc2d)	21657	5	21657	0.00	21657	0.00	21657	0.00
from [21]		0	21941	1.31	22026	1.70	22131	2.19
89rbg443 (explicit)	632	2	632	0.00	632.8	0.13	634	0.32
from [21]			—	—	—	—	—	—
our max error				0.46%		0.87%		1.55%
our max error for SL				0.00%		0.38%		0.79%
max error of [21]				1.31%		1.70%		2.19%

Instance	Distance type	Min time (s)	Mean time (s)	Max time (s)
40d198	euc2d	40.0	41.9	42.8
40kroA200	euc2d	35.4	36.5	37.7
40kroB200	euc2d	33.8	35.8	38.1
45ts225	euc2d	38.1	38.5	38.8
46gr229	geo	44.4	45.0	46.6
46pr226	euc2d	28.2	29.3	31.0
53gil262	euc2d	46.4	54.8	79.9
53pr264	euc2d	47.5	48.5	49.3
60pr299	euc2d	55.8	56.3	57.2
64lin318	euc2d	63.0	64.3	66.0
65rbg323	explicit	67.6	80.5	100.8
72rbg358	explicit	77.3	80.1	84.1
80rd400	euc2d	89.5	119.8	187.9
81rbg403	explicit	71.3	73.0	75.3
84fl417	euc2d	107.1	111.4	116.6
87gr431	geo	124.1	127.3	134.4
88pr439	euc2d	111.2	113.5	117.2
89pcb442	euc2d	108.2	110.9	112.5
89rbg443	explicit	100.9	102.3	105.4

Now we can compare our results with the results from [21]. For all instances considered in [21] we have equal or better solution quality and, moreover, the best of our five tries always gave an optimal solution. The error of [21] algorithm average 0.36% whereas our algorithm error average is only 0.07% for the same instance set or 0.13% for the full instance set. Our algorithm has reached the optimal value in 75% of all tries and in 85% of tries for instances considered in [21] whereas the heuristic from [21] has reached the optimum value only in 46% of tries. The computation time of our heuristic is about 10–20 times higher than that of the Snyder-Daskin heuristic; our heuristic would be 5–10 times slower than that of the Snyder-Daskin heuristic if we used C++ (see above).

## 6 Acknowledgement

We would like to thank Michael Basarab for helpful discussions on memetic algorithms. Research of Gutin was supported in part by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778.

## References

1. Bang-Jensen J, Gutin G (2000) Digraphs: Theory, Algorithms and Applications. Springer, London

2. Bean JC (1994) *ORSA J Computing* 6:154-160
3. Ben-Arieh D, Gutin G, Penn M, Yeo A, Zverovitch A (2003) *Operations Research Letters* 31:357-365
4. Bruckschlegel T (2005) Microbenchmarking C++, C#, and Java, Dr. Dobbs, [www.ddj.com/184401976](http://www.ddj.com/184401976)
5. Fischetti M, Salazar-González JJ, Toth P (2002) The generalized traveling salesman and orientering problems. In: Gutin G, Punnen AP (eds) *The Traveling Salesman Problem and its Variations*. Kluwer, Dordrecht
6. Fischetti M, Salazar-González JJ, Toth P (2003) In: Gross J, Yellen J (eds) *Handbook of Graph Theory*. CRC Press
7. Gutin G (2003) *Traveling Salesman Problems*. In: Gross J, Yellen J (eds) *Handbook of Graph Theory*. CRC Press
8. Gutin G, Yeo A (2003) *Australasian J Combinatorics* 27:149–154
9. Gutin G, Punnen AP (eds) (2002) *Traveling Salesman Problem and its Variations*. Kluwer, Dordrecht
10. Johnson DS, Gutin G, McGeoch L, Yeo A, Zhang X, Zverovitch A (2002) Experimental Analysis of Heuristics for ATSP. In: Gutin G, Punnen AP (eds) *The Traveling Salesman Problem and its Variations*. Kluwer, Dordrecht
11. Johnson DS, McGeoch L (2002) Experimental Analysis of Heuristics for STSP. In: Gutin G, Punnen AP (eds) *The Traveling Salesman Problem and its Variations*. Kluwer, Dordrecht
12. Krasnogor N, Smith JE (2001) Emergence of profitable search strategies based on a simple inheritanc mechanism. In: Spector L et al (eds) *Intern Genetic and Evolut Comput Conf (GECCO2001)*. Morgan Kaufman, San Francisco
13. Krasnogor N, Smith JE (2005) *IEEE Trans Evolut Comput* 9:474–488
14. Laporte G, Asef-Vaziri A, Sriskandarajah C (1996) *J Operational Research Society* 47: 1461-1467
15. Lawler EL, Lenstra JK, Rinooy Kan AHG, Shmoys DB (eds) (1985) *Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*. Wiley, Chichester
16. Moscato P, Memetic algorithms: A short introduction. In: Corne D, Glover F, Dorigo M (eds) (1999) *New Ideas in Optimization*. McGraw-Hill
17. Motta L, Ochi LS, Martinhon C (2001) Grasp metaheuristics for the generalized covering tour problem. In: *MIC2001-4th Metaheuristics Int Conf*. Porto, Portugal
18. Ong YS, Lim MH, Zhu N, Wong W (2006) *IEEE Trans Systems Man Cybern Part B* 36:141–152
19. Reinelt G (1991) *ORSA J Computing* 3:376-384
20. Samanlioglu F, Kurz MB, Ferrell WG, Tangudu S (2006) *Intern J Operational Res* 2:47–63
21. Snyder LV, Daskin MS (2006) *Europ J Operational Research* 174:38–53
22. Spears WM, De Jong KA (1991) On the virtues of parameterized uniform crossover. In: *Proc 4th Int Conf Genetic Algorithms*
23. Tsai H-K, Yang J-M, Tsai Y-F, Kao C-Y (2004) *IEEE Transactions on SMC-part B* 34:1718–1729
24. Zverovitch A (2002) GTSP instances. [www.cs.rhul.ac.uk/home/zvero/GTSPLIB/](http://www.cs.rhul.ac.uk/home/zvero/GTSPLIB/)