Traveling Salesman Problem

Genetic Algorithms and Evolutionary Computing (|H08M3a|)

Li Quan Wout Swinnen

January 13, 2011

Contents

List of Figures

List of Tables

1. Introduction

In this project, some of the key concepts of solving combinatorial optimization problems with the use of genetic algorithms are explored. To be more precise, we will implement genetic operators and investigate their characteristics when applied to the traveling salesman problem (TSP).

2. Traveling salesman problem

2.1. Description

The TSP is the classic example of a route planning problem. The aim of the TSP is to find a route between cities that is as short as possible, where every city has to be visited once, and where the final city has to be the same one as the starting city. There are many more complicated forms of this TSP, e.g., [1, 2, 3], but these will not be discussed in this project.

2.2. TSP representation

First, we will discuss how to represent a TSP tour. This will have an impact on the possible crossover and/or mutation operators and their performance.

2.2.1. Adjacency representation

The adjacency representation describes a tour with a list of n cities where city j is listed in position i if and only if the tour leads from city i to city j. The adjacency representation allows schemata analysis, but it has many disadvantages. The use of normal crossover operators will very likely introduce illegal tours in the population, and they have to be complemented with repair operators. On the other hand, the use of crossover operators designed for the adjacency representation will generally lead to very poor results.

2.2.2. Ordinal representation

When using the ordinal presentation, a tour is also represented as a list of n cities; the ith element of the list is a number in the range from 1 to $n - i + 1$, and there an ordered list of cities serving as a reference point is also used.

The main advantage of this rather complicated ordinal representation lies in the fact that the classical crossover can be used. (This follows from the fact that the ith element of the tour representation is always a number in the range from 1 to $n - i + 1$.) The results obtained using ordinal representation however have been generally poor.

2.2.3. Path representation

The most straightforward, natural representation for describing TSP tours is the path representation. In the path representation, once again a list of n cities is used. In this list, the jth element with value i denotes that the jth city to be visited is city i. Also in this case, the classical crossover operators will not be usable. However using this representation, specific crossover operators have been designed that do give very good results.

2.3. Tasks

The main objective of this project is to investigate the performance of various GA operators and discuss possible improvements. In particular, we change the original adjacency representation (with alternating edge crossover and simple inversion[1] mutation) to *path representation* and implement appropriate operators. We implemented all crossover and mutation operators mentioned in [1] for the path representation.

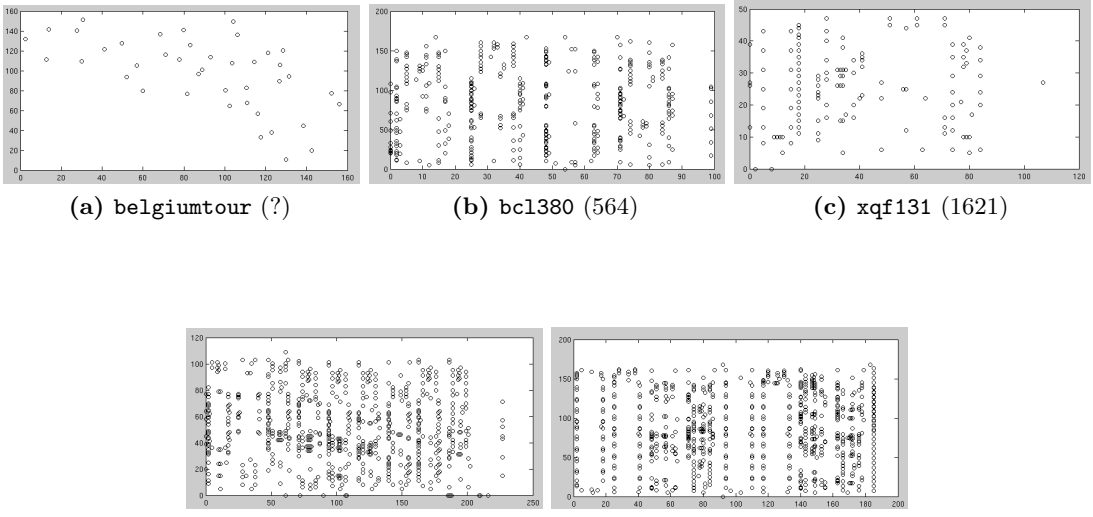
More specifically, this consists of the following crossover operators: *Partially Matched Crossover (PMX)*, *Order Crossover (OX)*, *Cyclic Crossover (CX)* and *(Enhanced) Edge Recombination Crossover ((E)ERX)*; and the following mutation operators: *inversion*, *exchange* and *insertion* mutation.

As an optional task, we have chosen to analyze the selection processes of *proportional (roulette wheel)* and *tournament* selection.

These operators and processes will be tested by comparing solution speeds, minimal tour distances, genetic diversity, etc. Finally, we will also briefly compare with some alternative approaches to solve TSP problems.

2.4. Benchmark problems

In general, the optimal solution of TSP problems is not known a priori. However, to study the performance of the algorithms it is crucial to have several benchmark problems with a known optimal solution in order to establish a baseline.



(a) `belgiumtour` (?) **(b)** `bcl380` (564) **(c)** `xqf131` (1621)

(d) `xql662` (2513) **(e)** `rbx711` (3115)

Figure 1: The five benchmark problems. In parentheses their optimal solutions are given.

The following experiments will therefore be based on 5 benchmark problems, shown in Figure 1.

[1] In the original implementation this was somewhat confusingly named inversion mutation, which is a slightly more complicated variant.

In particular, these problems consist of respectively 131, 380, 662 and 711 nodes, and one with 41 nodes based on Belgium's most important cities. For all of these except for `belgiumtour`, we know the optimal solution.

3. Experiments

In this section, we will perform experiments to analyze the properties of the various operators for the genetic algorithms with respect to the TSP problem. First, we will investigate the crossover operators, followed by the mutation operators and finally the selection operators.

We will measure the performance by running the genetic algorithm several times for each of the 5 benchmark problems. To be precise, we will run the algorithm 5 times for each problem, and record the computation time and solution quality of each run. By repeatedly solving different problems, we will also get a measure of how much the applied operators depend on the problem size. And, because benchmark problems have been used, we can also express the quality of the best solution in a relative manner with respect to the global optimum solution. As a measure for this, we use the relative error, i.e., $\epsilon_{\text{rel}} = \frac{|x_{\text{opt}} - \hat{x}|}{x_{\text{opt}}}$.

All experiments were executed with MATLAB version 2010b (Linux 64 bit version) on a laptop with an Intel Core i3-370M 2.4 Ghz and 4 GB RAM.

3.1. Crossover operators

The parameters for the experiments concerning the crossover operators are as follows. Crossover operator with 90% probability, simply inversion mutation with 10% probability, 5% elitism, stochastic uniform sampling for selection and no further applied heuristics. The number of individuals in each generation is 50, and the algorithm is stopped after 100 generations. (The crossover probability is set high so that the outcome of the algorithm is very much characterized by the respective applied crossover operator.)

The results of the alternating edge crossover, partially matched crossover and order crossover operators are given in Table 1, and of the cycle crossover, edge recombination crossover and its enhanced variant in Table 2. (The implementation of these crossover operators is found in Appendix A.1 on page 17.)

3.1.1. Alternating edge crossover

Description The alternating edge crossover chooses an edge from the first parent at random. Then, the partial tour created in this way is extended with the appropriate edge of the second parent. This partial tour is extended by the adequate edge of the first parent, etc.

Results It can be seen that the computation time does not increase linearly with the problem size. The solutions drift away from the optimal solution as the problem size increases, because the amount of generations allowed becomes too limited to give good results.

We conclude that both the solving time and solution quality of this crossover operator are of very poor quality (as will be seen when compared to the other crossover operators). This is because the alternating edges crossover often destroys good subtours.

3.1.2. Partially matched crossover

Description The partially matched crossover (PMX) operator passes on ordering and value information from the parent tours to the offspring tours: A part of one parent's string is mapped

	belgiumtour		xqf131		bcl380		xql662		rbx711	
Run	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance
AEX										
1	24.0	1486	32.3	3250	69.5	21046	110.0	45983	111.3	56082
2	23.6	1682	33.3	3267	71.5	21223	111.2	45244	120.1	56562
3	24.3	1498	32.2	3250	69.5	21046	116.6	45952	113.3	56522
4	23.7	1635	31.9	3338	68.8	21757	105.9	46538	114.5	56538
5	23.6	1519	32.3	3380	70.3	21781	107.3	46732	119.1	55268
Worst	24.3	1682 (?)	33.3	3380 (5.0)	71.4	21781 (12.4)	116.6	46732 (18.0)	120.1	56562 (17.2)
Average	23.8	1564 (?)	32.4	3297 (4.8)	69.9	21371 (12.2)	110.2	46090 (17.3)	115.7	56194 (17.0)
Best	23.6	1486 (?)	31.9	3250 (4.7)	68.8	21046 (12.0)	105.9	45244 (17.0)	111.3	55268 (16.7)
PMX										
1	7.0	1090	8.7	2462	16.0	18317	30.6	39367	35.2	48260
2	7.1	1226	8.7	2242	16.5	18581	31.3	39728	34.2	50096
3	7.0	984	8.6	2535	16.6	18315	32.1	40302	35.1	50821
4	7.1	1238	8.5	2404	16.6	18373	32.7	41374	33.3	50139
5	7.2	910	8.6	2232	16.7	18244	29.6	39863	33.5	49296
Worst	7.2	1238 (?)	8.7	2535 (3.5)	16.7	18581 (10.5)	32.7	41374 (15.5)	35.2	50821 (15.3)
Average	7.1	1090 (?)	8.6	2375 (3.2)	16.5	18366 (10.3)	32.3	40127 (15.0)	34.3	49722 (15.0)
Best	7.0	910 (?)	8.5	2232 (3.0)	16.0	18244 (10.2)	29.6	39367 (14.7)	33.3	48260 (14.5)
OX1										
1	6.8	952	8.3	2332	7.8	17290	8.3	38775	8.5	48131
2	6.8	931	7.1	2314	7.9	16928	8.2	39335	8.6	48709
3	6.8	1002	7.0	2389	7.7	17081	8.3	39317	8.7	47980
4	6.8	985	7.0	2309	7.8	17066	8.4	38891	8.7	48180
5	6.9	928	7.0	2186	7.8	16670	8.3	38486	8.6	48918
Worst	6.9	1002 (?)	8.3	2389 (3.2)	7.9	17290 (10.1)	8.4	39335 (14.7)	8.7	48918 (14.7)
Average	6.8	959 (?)	7.3	2306 (3.1)	7.8	17007 (9.5)	8.3	38960 (14.5)	8.6	48384 (14.5)
Best	6.8	928 (?)	7.0	2186 (2.9)	7.7	16670 (9.3)	8.2	38486 (14.3)	8.5	47980 (14.4)

Table 1: TSP performance on benchmark problems of crossover operators: *alternating edge crossover* (AEX), *partially matched crossover* (PMX) and *order crossover* (OX1).

	belgiumtour		xqf131		bcl380		xql662		rbx711	
Run	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance
CX										
1	6.6	1381	6.9	3032	8.0	21334	9.6	44788	9.3	56008
2	6.7	1223	7.1	2847	8.2	21104	10.3	43846	10.2	53786
3	6.6	1244	7.0	2828	8.4	20150	10.0	44187	10.2	55655
4	6.8	1354	7.0	2965	8.2	20936	9.5	46068	10.9	55655
5	6.6	1195	7.0	2858	8.6	20274	10.6	43828	9.3	55974
Worst	6.8	1381 (?)	7.1	3032 (4.4)	8.6	21334 (12.2)	10.6	46068 (17.3)	10.9	56008 (17.0)
Average	6.7	1279 (?)	7.0	2906 (4.2)	8.2	20759 (11.8)	10.0	44543 (16.7)	10.0	55472 (16.8)
Best	6.6	1195 (?)	6.9	2828 (4.0)	8.0	20150 (11.4)	9.5	43828 (16.4)	9.3	53786 (16.3)
ERX										
1	9.9	889	18.4	1824	119.3	13298	272.8	23010	288.0	33963
2	10.0	926	18.8	1537	118.7	14512	277.7	22584	289.0	34295
3	9.9	1021	18.9	1716	120.4	13190	264.3	22552	288.8	32114
4	10.6	971	18.7	1619	110.0	14352	269.4	23115	289.7	34034
5	10.0	979	18.8	1666	108.9	14826	265.3	22918	290.6	33003
Worst	10.6	1021 (?)	18.9	1824 (2.2)	120.4	14826 (8.1)	277.7	23115 (8.2)	290.6	34295 (10.0)
Average	10.1	957 (?)	18.7	1683 (2.0)	115.5	14036 (7.7)	269.9	22836 (8.1)	289.2	33482 (9.7)
Best	9.9	889 (?)	18.4	1573 (1.8)	108.9	13190 (7.1)	264.3	22552 (8.0)	288.0	32114 (9.3)
EERX										
1	10.9	892	22.0	1619	115.7	13947	267.7	24128	308.0	34244
2	10.7	946	21.4	1699	118.8	12968	281.7	24498	291.0	33877
3	10.8	964	22.0	1780	119.8	13688	253.8	25435	286.0	34136
4	11.0	901	21.2	1780	109.3	13049	257.0	24897	289.0	33511
5	10.8	896	22.2	1613	111.4	13516	252.8	24617	293.3	33780
Worst	11.0	964 (?)	22.2	1824 (2.2)	119.8	13947 (7.6)	281.7	25435 (9.1)	308.0	34244 (10.0)
Average	10.8	920 (?)	21.8	1683 (2.0)	115.0	13443 (7.3)	262.6	24715 (8.8)	293.5	33910 (9.9)
Best	10.7	892 (?)	21.2	1573 (1.8)	109.3	12968 (7.0)	252.8	24128 (8.6)	286.0	33511 (9.7)

Table 2: TSP performance on benchmark problems of crossover operators: *cyclic crossover* (CX), *edge recombination crossover* (ERX) and *enhanced edge recombination crossover* (EERX).

onto a part of the other parent's string and the remaining information is exchanged.

Results We already see that this operator consistently gives better results than the alternating edge crossover operator. For computation time, we see a speedup with a factor of $\approxeq 4$; the solution quality is slightly better with roughly 20%. We see the same issues with the scalability of the TSP sizes as before, only the solution time dependence on the problem size is much smaller in this case.

The PMX operator therefore tries to keep the positions of the cities in the path representation; these are rather irrelevant in the context of the TSP problem where the most important goal is to keep the sequences. Thus, the performance of this operator for the TSP is rather poor, as shown by the solution quality on the benchmark problems.

3.1.3. Order crossover

Description The order crossover (OX1) operator constructs an offspring by choosing a subtour of one parent preserving the relative order of the other parent.

Results The solution quality of OX1 is better than PMX for all benchmark problems. But more importantly, we see that (this particular implementation of) the operator scales very well with the problem size in terms of computational complexity.

The solution quality is high because it employs the essential property of the path representation, that the order of cities is important and not their position. Its computational complexity is low because the algorithm is very simple. Overall, this is an operator with a good trade-off between solution quality and computational complexity.

3.1.4. Cycle crossover

Description The cyclic crossover (CX) operator attempts to create an offspring from the parents where every position is occupied by a corresponding element from one of the parents.

Results This operator is somewhat slower than the order crossover and the solution quality is worse than PMX (but still better than AEX). This operator is thus not really interesting because it performs mediocre without any benefits.

In [5], it was concluded from theoretical and empirical results that the CX operator gave slightly better results than the PMX operator. In our experiments we did not see this result.

However our basic assumption that in the context of the TSP it is much more important to keep sequences rather than positions, is fortified, as we see that OX is definitely better than PMX and CX.

3.1.5. (Enhanced) edge recombination crossover

Description Even if the main aim of the OX operator is to keep the sequence of at least one parent there are still quite a lot of new edges in the offspring.

The edge recombination operator (ERX) has been designed with the objective of keeping as many edges defined by the parents as possible.

As common sequences of the parent tours are not taken into account by the ERX operator an enhancement, the enhanced edge recombination crossover (EERX) has been proposed.

The EERX additionally gives priority to those edges starting from the current city which are present in both parents.

Results The solution quality of these operators is undoubtly the best. It is clear that by keeping as many edges defined by the parents as possible, these operators give very good solutions, but the drawback is that these operators need a very complex and time consuming procedure.

We can also see the improvement of EERX in solution quality versus ERX by sacrificing a relative small amount of computation time.

3.2. Mutation operators

The parameters for the experiments concerning the mutation operators are as follows. Crossover operator with 10% probability, mutation operator with 90% probability, 5% elitism, stochastic uniform sampling for selection and no further applied heuristics. The number of individuals in each generation is 50, and the algorithm is stopped after 100 generations. (The mutation probability is set high so that the outcome of the algorithm is very much characterized by the respective applied mutation operator.)

The results of the simple inversion, inversion, exchange and insertion mutation operators are given in Table 3. (The implementation of these mutation operators is found in Appendix A.2 on page 20.)

3.2.1. Simple inversion mutation

Description The simple inversion mutation operator randomly selects two cut points and simply reverses the string between them.

Results We cannot yet compare with the other mutation operators, but we can already compare with the results of the OX1 results in Table 1, where the same parameters were used except for a higher crossover rate and lower mutation rate.

The solution distances now are significantly worse. This is definitely because of the low crossover probability, which inhibits a decent exploration of the search space. Also, we can already predict that the type of mutation operator will not influence the results as much as the type of crossover operator used. This is a straightforward result, since the differences in implementation of the crossover operators are way more pronounced, and thus affect the efficiency of the search space exploration in a profound manner. The same does not hold for the mutation operators, for which these small differences in implementation do not have a strong effect on the way the search space is exploited. Also, for the crossover operators it is way easier to counteract the building block theory and destroy relevant alleles. Mutation operators only introduce small changes, and do not threaten the genetic building blocks as much.

3.2.2. Inversion mutation

Description The inversion mutation operator randomly selects a subtour, removes it, and inserts it in reverse order at a randomly chosen position.

Results This mutation operator requires slightly more calculation time than the previous operator. This is due to the fact that calculation of one extra random position has to be done and the insertion of the subtour in the result. For the solution quality, we see no significant differences.

	belgiumtour		xqf131		bcl380		xql662		rbx711	
Run	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance
simple inversion										
1	6.8	1090	6.8	2530	7.5	19014	8.5	41125	8.4	53679
2	6.7	1090	6.8	2404	7.7	18897	8.0	42498	8.8	54119
3	6.7	1081	6.8	2496	7.4	18592	8.9	42802	8.5	52536
4	6.7	1004	7.2	2444	8.6	18303	9.3	42033	9.4	51366
5	7.3	1182	7.6	2446	8.3	19469	9.1	43261	9.5	53439
Worst	7.3	1182 (?)	7.6	2530 (3.5)	8.6	19469 (11.0)	9.3	43261 (16.2)	9.5	54119 (16.4)
Average	6.8	1089 (?)	7.0	2464 (3.4)	7.9	18856 (10.6)	8.8	42424 (15.9)	8.9	53028 (16.0)
Best	6.7	1004 (?)	6.8	2404 (3.3)	7.4	18303 (10.3)	8.0	41125 (15.3)	8.4	51366 (15.5)
inversion										
1	7.0	1144	7.1	2454	8.6	18562	9.2	42705	9.6	51351
2	7.1	1200	7.0	2460	8.7	18396	9.0	42384	9.2	52843
3	6.9	1378	7.0	2523	8.3	18950	9.9	40665	9.5	52511
4	7.1	1230	7.6	2484	8.7	18402	8.9	41613	9.4	51824
5	7.7	1265	7.5	2434	8.1	18975	9.0	42298	9.3	52342
Worst	7.7	1378 (?)	7.6	2523 (3.5)	8.7	18975 (10.7)	9.9	42705 (16.0)	9.6	52843 (16.0)
Average	7.2	1243 (?)	7.2	2471 (3.4)	8.5	18657 (10.5)	9.2	41933 (15.7)	9.4	52174 (15.7)
Best	6.9	1144 (?)	7.0	2434 (3.3)	8.1	18396 (10.3)	8.9	40665 (15.2)	9.2	51351 (15.5)
insertion										
1	7.2	1254	7.2	2689	8.6	19387	9.2	42436	9.5	52299
2	7.2	1163	7.3	2778	8.0	18499	8.9	42602	9.1	52499
3	7.0	1265	7.4	2483	7.6	19252	8.9	42040	8.8	53086
4	7.2	1260	7.1	2558	8.3	19212	8.9	43277	9.4	52389
5	7.3	1081	7.3	2512	8.2	18510	9.0	43329	9.1	53047
Worst	7.3	1265 (?)	7.4	2778 (3.9)	8.6	19252 (10.9)	9.2	43329 (16.0)	9.5	53086 (16.0)
Average	7.2	1205 (?)	7.3	2604 (3.6)	8.1	18972 (10.7)	9.0	42737 (16.0)	9.2	52664 (15.9)
Best	7.0	1081 (?)	7.1	2483 (3.4)	7.6	18499 (10.4)	8.9	42040 (15.7)	8.8	52299 (15.8)
exchange										
1	7.5	1274	7.2	2742	8.5	18994	9.4	42085	9.4	52763
2	7.6	1212	7.3	2706	7.9	18645	8.7	41072	9.4	50406
3	7.0	1241	7.4	2619	8.2	18570	8.8	41692	9.4	52166
4	7.2	1205	7.1	2624	8.2	19286	8.8	42493	9.2	51147
5	7.2	1269	7.3	2782	7.9	18601	9.0	41614	9.2	52047
Worst	7.6	1274 (?)	7.4	2782 (3.9)	8.5	19286 (10.9)	9.4	42493 (15.9)	9.4	52763 (15.9)
Average	7.3	1240 (?)	7.3	2695 (3.8)	8.1	18819 (10.6)	8.9	41791 (15.6)	9.3	51706 (15.8)
Best	7.0	1205 (?)	7.1	2619 (3.6)	7.9	18570 (10.5)	8.8	41072 (15.3)	9.2	50406 (15.2)

Table 3: TSP performance on benchmark problems of mutation operators: *simple inversion*, *inversion*, *insertion* and *exchange*.

3.2.3. Insertion mutation

Description The insertion mutation operator randomly chooses a city, removes it from the tour and inserts it at a randomly selected place.

Results Also for the insertion mutation, we see no substantial improvements in solution quality. As for the time cost, it is on the same order as the inversion mutation.

3.2.4. Exchange mutation

Description The exchange mutation operator selects two cities of the tour randomly and simply exchanges them.

Results This operator has the same performance as the insertion and inversion mutation.

3.3. Selection operators

The selection operators implemented are stochastic uniform sampling, which was already given in the toolbox and used, a roulette wheel selection (which was also part of the toolbox) and a tournament selection. For the experiments we use OX1 crossover with 90% probability, simple inversion mutation with 10% probability, and 20% elitism, and 100 consecutive generations with 50 individuals each. No extra heuristics were applied.

The results of all selection operators are given in Table 4. (The implementation of these selection operators is found in Appendix A.3 on page 22.)

3.3.1. Stochastic uniform sampling

Description The stochastic uniform sampling is the parent selection technique that was already given in the MATLAB toolbox and originally used. This sampling technique provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection (see further). Here equally spaced pointers are placed over the line as many as there are individuals to be selected.

Results We can compare with Table 1, where the same parameters and operators were used, except for the lower elitism percentage of 5%. We see that increasing the elitism rate gives rise to longer calculation times. The quality of the solutions does not improve by this boost of elitism from 5 to 20%.

3.3.2. Roulette wheel selection

Description The simplest selection scheme is roulette wheel selection, also called stochastic sampling with replacement. This is a stochastic algorithm and involves the following technique. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness.

	belgiumtour		xqf131		bcl380		xql662		rbx711	
Run	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance
SUS										
1	7.2	951	7.3	2268	8.1	17675	8.7	39012	8.7	47856
2	7.1	1048	7.3	2357	8.1	17451	8.6	39186	8.6	49247
3	6.8	927	8.0	1979	8.2	17274	8.5	38810	8.7	47947
4	7.2	992	7.4	2329	8.1	17879	8.6	40101	8.9	47876
5	7.2	1097	7.3	2306	8.1	16838	8.8	39240	8.8	47260
Worst	7.2	1097 (?)	8.0	2357 (3.2)	8.8	17879 (10.0)	8.4	39335 (14.7)	8.9	49247 (14.8)
Average	7.1	1003 (?)	7.5	2248 (3.0)	8.6	17423 (9.7)	8.3	38960 (14.5)	8.7	48037 (14.4)
Best	6.8	927 (?)	7.3	1979 (2.5)	8.5	16838 (9.4)	8.2	38486 (14.3)	8.6	47260 (14.2)
RWS										
1	7.3	973	7.4	2345	8.2	17723	8.8	40211	8.8	47788
2	5.4	1255	7.5	2357	8.1	17534	8.8	38354	8.8	48266
3	7.1	1043	7.4	2241	8.4	18026	8.5	39751	8.7	49763
4	6.1	1027	7.3	2281	8.1	17928	8.6	39514	8.9	49749
5	7.2	1055	7.3	2341	8.0	18087	8.6	39514	8.8	49030
Worst	7.3	1097 (?)	7.5	2345 (3.2)	8.4	18087 (10.2)	8.8	40211 (15.0)	8.9	49763 (15.0)
Average	6.6	1061 (?)	7.4	2313 (3.1)	8.2	17860 (10.0)	8.6	39469 (14.7)	8.8	48919 (14.7)
Best	5.4	973 (?)	7.3	2241 (3.0)	8.0	17534 (9.8)	8.5	38354 (14.3)	8.7	47788 (14.3)
TS										
1	7.6	807	7.6	1912	8.2	15656	8.8	40211	9.0	46020
2	7.4	829	7.7	2021	8.6	15619	8.8	36780	9.1	47460
3	7.4	862	7.5	1919	8.2	15976	8.9	36479	9.1	45846
4	7.5	864	7.5	1895	8.3	15883	8.9	36939	9.3	47556
5	7.5	899	7.6	1927	8.3	16227	9.0	38224	9.4	45521
Worst	7.3	899 (?)	7.7	2021 (2.6)	8.6	16227 (9.0)	8.9	38224 (14.2)	9.4	47556 (14.3)
Average	6.6	852 (?)	7.6	1935 (2.4)	8.3	15872 (8.8)	8.6	37201 (13.8)	9.2	46481 (13.9)
Best	5.4	807 (?)	7.5	1895 (2.3)	8.2	15619 (8.6)	8.5	36479 (13.5)	9.0	45521 (13.6)

Table 4: TSP performance on benchmark problems of selection operators: *stochastic uniform sampling* (SUS), *roulette wheel selection* (RWS) and *tournament selection* (TS).

Results There is no real difference in time cost for this operator when compared to the stochastic uniform sampling. Also, the quality of the solutions is slightly worse. The reason is that stochastic uniform sampling ensures a selection of offspring which is closer to what is "deserved" (concerning the fitness function) than roulette wheel selection.

3.3.3. Tournament selection

Description In tournament selection a number of individuals (K) is chosen randomly from the population and the best individual from this group is selected as parent. This process is repeated as often as individuals must be chosen. These selected parents produce uniform at random offspring.

Results Since tournament selection also requires the definition of the variable K (size of the tournament subpopulation), we have chosen a value of 10% of the population size (i.e., in these experiments $K = 5$). Also, when running the algorithm for different mutation rates, we noticed that the tournament selection attained better results when the mutation rate was made larger. This was not the case for the two previous selection operators. Optimal results with tournament selection were attained when mutation rate was set to approximately 35%.

This is probably due to the character of the tournament selection. When mutation is boosted, the search space is exploited more efficiently at first. However, when mutation rates get too high, it becomes too difficult to preserve relevant building blocks in the genetic code. Tournament selection tends to buffer this effect, since winners are chosen from many different randomly chosen subpopulations, which provides a greater genetic diversity in the selected parents. This decreases the chance that the relevant and valuable building blocks die out.

We can see that running the algorithm has become more cost-expensive with respect to the two previous cases. This is a result of the higher mutation rate, and the increased number of applied mutation operators. We can also see that indeed the quality of the solutions is slightly better than in the two previous cases, due to the cooperation of mutation and tournament selection.

3.4. Comparison with other techniques

There are of course many other techniques to solve TSP problems and combinatorial problems in general. Among those, we will discuss the nearest neighbor and simulated annealing algorithms.

The results of these alternative methods are shown in Table 5 on page 14.

3.4.1. Nearest neighbor heuristic

Description The nearest neighbor algorithm is a typical representative of a route building heuristics. It simply considers a city as its starting point and takes the nearest city in order to build up the path.

Results This approach gives very fast and good solutions. The most important drawback is that although this strategy works out quite well in the beginning of the path construction, adverse stretches have to be inserted when only a few cities are left. This is illustrated in Figure 2.

3.4.2. Simulated annealing

Description Simulated annealing [4] (SA) is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. Wikipedia [6] gives following description:

	belgiumtour		xqf131		bcl380		xql662		rbx711	
Run	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance	Time (s)	Distance
NN										
1	< 0.01	782	< 0.01	654	0.01	2021	0.05	3072	0.05	3618
2	< 0.01	809	< 0.01	668	0.02	2069	0.05	3167	0.06	3815
3	< 0.01	799	< 0.01	715	0.02	2086	0.04	3248	0.06	3707
4	< 0.01	807	< 0.01	753	0.01	2048	0.05	3203	0.05	3959
5	< 0.01	778	< 0.01	737	0.02	2072	0.04	3216	0.06	3879
Worst	< 0.01	809 (?)	< 0.01	753 (0.34)	0.02	2086 (0.29)	0.05	3248 (0.29)	0.06	3959 (0.27)
Average	< 0.01	795 (?)	< 0.01	705 (0.25)	0.02	2059 (0.27)	0.05	3181 (0.27)	0.05	3796 (0.22)
Best	< 0.01	778 (?)	< 0.01	654 (0.16)	0.01	2021 (0.25)	0.04	3072 (0.22)	0.05	3248 (0.04)
SA										
1	1.0	861	9.0	1180	17.5	6217	17.0	13540	17.1	17807
2	1.4	1006	9.3	1079	17.5	5837	16.6	13749	16.2	17576
3	1.3	964	7.6	1143	17.7	6019	16.9	14192	16.5	17738
4	1.3	870	13.0	973	17.3	5810	16.5	13458	16.8	16431
5	1.2	744	7.9	1202	17.3	6051	16.8	13710	16.7	16247
Worst	1.4	1006 (?)	13.0	1202 (1.13)	17.7	6217 (2.84)	17.0	14192 (4.64)	17.1	17807 (4.72)
Average	1.2	889 (?)	9.4	1115 (0.98)	17.5	6987 (3.31)	16.8	13730 (4.46)	16.7	15840 (4.09)
Best	1.0	744 (?)	7.6	973 (0.73)	17.3	5810 (2.58)	16.5	13458 (4.36)	16.2	14192 (3.56)
NN+SA										
1	0.6	777	1.6	685	2.9	1968	0.6	3182	0.7	3707
2	0.6	763	3.7	690	0.5	1953	0.6	3117	0.6	3775
3	0.9	733	0.7	715	0.5	1955	0.6	3135	0.8	3862
4	0.5	758	0.8	688	0.5	2056	0.6	3226	0.6	3959
5	0.7	741	2.1	681	0.4	2110	0.5	3164	1.0	3851
Worst	0.9	777 (?)	3.7	715 (0.27)	2.9	2110 (0.30)	0.6	3226 (0.28)	0.8	3959 (0.27)
Average	0.7	754 (?)	1.8	692 (0.23)	0.9	2008 (0.24)	0.6	3165 (0.26)	0.7	3831 (0.23)
Best	0.5	733 (?)	0.7	681 (0.21)	0.4	1953 (0.20)	0.5	3135 (0.25)	0.6	3707 (0.19)

Table 5: TSP performance on benchmark problems using other techniques: *nearest neighbor algorithm* (NN) and *simulated annealing* (using random path as initial solution: SA; using path generated by NN heuristic as initial solution: SA+NN).

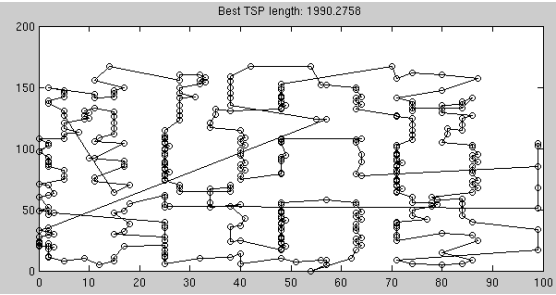


Figure 2: The nearest neighbor heuristic on `bcl380` shows it limitations: while overall a very good TSP solution, there are a few adverse stretches (built at the end of the construction of the path).

> The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.
>
> By analogy with this physical process, each step of the SA algorithm attempts to replace the current solution by a random solution (chosen according to a candidate distribution, often constructed to sample from solutions near the current solution). The new solution may then be accepted with a probability that depends both on the difference between the corresponding function values and also on a global parameter T (called the temperature), that is gradually decreased during the process. The dependency is such that the choice between the previous and current solution is almost random when T is large, but increasingly selects the better or "downhill" solution (for a minimization problem) as T goes to zero. The allowance for "uphill" moves potentially saves the method from becoming stuck at local optima-which are the bane of greedier methods.

We initialized the TSP path for the simulated annealing algorithm[2] with either a random path or one attained by the NN heuristic. For neighborhood candidate solution generation, we simply used the exchange mutation operator. The initial temperature was 1, the cooling schedule was set so the temperature decreases with a factor $\beta = 0.9$. Maximum consecutive rejections was set to 2500 and maximum tries to 500.

Results Overall, we see very good results both solution quality wise as computational complexity wise. In general these algorithms require less parameters to set. The use of an path obtained by the NN heuristic as a initial solution for the simulated annealing algorithm does not result in significantly better solutions in the same parameter settings.

[2]We used the simulated annealing toolbox found at MATLAB File Exchange, `http://www.mathworks.com/matlabcentral/fileexchange/10548`.

4. Conclusion

For the crossover operators, our results give the following hierarchies in performance: when considering speed, OX is definitely the fastest, followed by CX, and then PMX. The slowest are by far the powerful ERX and its even slower version EERX. When considering only the quality of the best solutions for the same number of allowed generations, the best crossover operators are clearly EERX and ERX. Next in order is the fast subtour preserving OX, followed by the position preserving PMX and CX. Overall, OX1 gives a good trade-off between solution quality and computational complexity.

Within the mutation operators, there were no real differences in solution quality nor time complexity. All four operators performed more or less the same. This shows that the choice of mutation operator is overall less important than the choice of the crossover operator.

When comparing purely the selection operator, the differences in performance were again not very distinct. We could only see that the tournament selection operator works slightly slower than the stochastic uniform sampling and the proportional selection, which work at more or less the same speed. If we compared only the different selection operators, also no differences in solution quality came to view. However, the tournament selection seems to outperform the other operators when working in a high mutation environment.

Considering the available crossover operators and mutation operators, it is very clear that the path representation is a massive improvement with respect to the adjacency representation. It is a more natural way of representing the phenotype of the tour, it allows crossover operators that have way better computational speeds, and it also allows operators that provide solutions that are much closer to the globally optimal solution. The path representation clearly outperforms the adjacency representation for the TSP problem.

Comparing genetic algorithms with other approaches such as simulated annealing resulted in following interesting conclusion. In general, other heuristics such as SA or NN were easier to use and gave better solutions in less time.

A. Matlab code

A.1. Crossover operators

A.1.1. PMX

```
function Offspring=partially_matched_crossover(Parents)
cols = size(Parents,2);

%select two different positions in the tour for the splitpoints
rndi = zeros(cols, [1,2])
while (rndi(2) == rndi(1)), rndi(2) = randi(cols); end
rndi = sort(rndi);

subtour1 = Parents(1, rndi(1):rndi(2)); %the subtour between splitpoints ...
    from parents 1
subtour2 = Parents(2, rndi(1):rndi(2)); %the corresponding mapping values

%put the subtour in the offspring
Offspring = Parents(2,:);
Offspring(rndi(1):rndi(2)) = subtour1;

%part before subtour
for k=1:rndi(1)-1
    while (sum(Offspring == Offspring(k)) > 1)
        idx = find(subtour1 == Offspring(k), 1, 'first');
        Offspring(k) = subtour2(idx);
    end
end
%part after subtour
for k=rndi(2)+1:cols
    while (sum(Offspring == Offspring(k)) > 1)
        idx = find(subtour1 == Offspring(k), 1, 'first');
        Offspring(k) = subtour2(idx);
    end
end
end
```

A.1.2. OX

```
function Offspring=order_crossover(Parents)
cols = size(Parents,2);

% select two different positions in the tour for the splitpoints
rndi = randi(cols,[1,2]);
while (rndi(2) == rndi(1)), rndi(2) = randi(cols); end
rndi = sort(rndi);

%the subtour between splitpoints from parents 1
subtour = Parents(1, rndi(1):rndi(2));
```

```
%filter out elements already in subtour from parent2
par2_filtered = Parents(2, ~ismember(Parents(2,:), subtour));

%create the resulting offspring
Offspring = [par2_filtered(1:rndi(1)-1) subtour par2_filtered(rndi(1):end)];
end
```

A.1.3. CX

```
function Offspring=cyclic_crossover(Parents)
Offspring = zeros(1,size(Parents,2));

i = 1; %index into first parent
%loop while offspring does not contain selected node
while (all(Offspring ~= Parents(1,i)))
    Offspring(i) = Parents(1,i);
    i = find(Parents(1,:) == Parents(2,i), 1, 'first');
end

% fill up the rest of the child with information from parent 2
stillToFill = (Offspring == 0);
Offspring(stillToFill) = Parents(2,stillToFill);
end
```

A.1.4. (E)ERX

```
% calculates the edgemap given the different paths in paths
% (each row given in path representation)
function edgemap = edge_map(parents)
[m,n] = size(parents);
edgemap = zeros(n);

%copy the first and last column for easy loop processing
parents2 = [ parents(:,end) parents parents(:,1) ];

%fill in the edge map
for p=1:m
    for k=2:n+1
        ingoing = parents2(p,k-1);
        curr = parents2(p,k);
        outgoing = parents2(p,k+1);

        edgemap(curr, ingoing) = edgemap(curr, ingoing) + 1;
        edgemap(curr, outgoing) = edgemap(curr, outgoing) + 1;
    end
end

if n >= 150 %convert to sparse representation if better performance
    edgemap = sparse(edgemap);
```

```
end
```

```
function Offspring=edge_recombination_crossover(Parents)
n = size(Parents,2);
Offspring = zeros(1,n);

edgemap = edge_map(Parents); %create edgemap
unvisited = true(n,1); %contains the unvisited cities

%choose random city to start
curr = randi(n);
unvisited(curr) = false;
Offspring(1) = curr;

for k=2:n
    %remove all occurences of current city from connected cities
    edgemap(:,curr) = 0;

    %check if current city has entries
    if any(edgemap(curr, :))
       %determine which of the cities
       %in the edgelist of current city has
       %fewest entitities in own edge list
       others = find(edgemap(curr, :), 4, 'first');
       edge_lengths = sum( edgemap(others, :) ~= 0,  2);
       [~, idx] = min(edge_lengths);
       curr = others(idx);

    else
        %select random unvisited city
        unvisitedIdx = find(unvisited, n-k+1, 'first');
        curr = unvisitedIdx(randi(length(unvisitedIdx)));
    end

    %add to offspring and set visited
    Offspring(k) = curr;
    unvisited(curr) = false;
end
end
```

```
function Offspring=enhanced_edge_recombination_crossover(Parents)
n = size(Parents,2);
Offspring = zeros(1,n);

edgemap = edge_map(Parents); %create edgemap
unvisited = true(n,1); %contains the unvisited cities

%choose random city to start
curr = randi(n);
unvisited(curr) = false;
Offspring(1) = curr;
```

```
for k=2:n
    %remove all occurences of current city from connected cities
    edgemap(:,curr) = 0;

    %check if current city has edges in both parents
    if any(edgemap(curr,:) >= 2)
        inbothIdx = find(edgemap(curr, :) >= 2, 4, 'first');
        curr = inbothIdx(randi(length(inbothIdx)));
    %check if current city has edges
    elseif any(edgemap(curr, :))
        others = find(edgemap(curr, :), 4, 'first');
        edge_lengths = sum( edgemap(others, :) ~= 0,  2);
        [~, idx] = min(edge_lengths);
        curr = others(idx);
    else
        %select random unvisited city
        unvisitedIdx = find(unvisited, n-k+1, 'first');
        curr = unvisitedIdx(randi(length(unvisitedIdx)));
    end

    %add to offspring and set visited
    Offspring(k) = curr;
    unvisited(curr) = false;
end
end
```

A.2. Mutation operators

A.2.1. Simple inversion

```
function NewChrom = simple_inversion(OldChrom)
n = size(OldChrom,2);
NewChrom = OldChrom;

% select two positions in the tour
rndi = randi(n,[1,2]);
while (rndi(2) == rndi(1)), rndi(2) = randi(n); end
rndi = sort(rndi);

NewChrom(rndi(1):rndi(2)) = NewChrom(rndi(2):-1:rndi(1));
```

A.2.2. Inversion

```
function NewChrom = inversion(OldChrom)
n = length(OldChrom);

% select two positions in the tour
rndi = randi(n,[1,2]);
```

```
while (rndi(2) == rndi(1)), rndi(2) = randi(n); end
rndi = sort(rndi);

%get and remove the subtour
subtour = OldChrom(rndi(2):-1:rndi(1));
NewChrom = [OldChrom(1:rndi(1)-1) OldChrom(rndi(2)+1:end)];

% select the index to insert the subtour
subtourI = randi(length(NewChrom)+1); %+1 because it is an insertion

NewChrom = [ NewChrom(1:subtourI-1) subtour NewChrom(subtourI:end) ];
```

A.2.3. Insertion

```
function NewChrom = insertion(OldChrom)
n = length(OldChrom);
NewChrom = OldChrom;

% select a random city in the given tour and remove it in result
rndInd = randi(n);
selected = NewChrom(rndInd);
NewChrom(rndInd) = [];

% select a random position in the tour and perform the insertion
rndInd = randi(n);
NewChrom = [NewChrom(1:rndInd-1) selected NewChrom(rndInd:end)];
```

A.2.4. Exchange

```
function NewChrom = exchange(OldChrom)
NewChrom = OldChrom;
n = size(NewChrom,2);

%create indices of swapped cities
rndi = randi(n,[1,2]);
while (rndi(2) == rndi(1)), rndi(2) = randi(n); end

%swap the cities in the tour
buffer = NewChrom(rndi(1));
NewChrom(rndi(1)) = NewChrom(rndi(2));
NewChrom(rndi(2)) = buffer;
```

A.3. Selection operators

A.3.1. Tournament selection

```
function NewChrIx = tournament_selection(FitnV,Nsel,K)
   Nind = size(FitnV,1);
   % K = tournament size, default is 10% of population size;
   % can not be easily integrated with gatbx toolbox functions
   % that is why it is for now here hacked in
   if (nargin < 3), K = ceil(0.1*Nind); end

   NewChrIx = zeros(Nsel,1);
   for k=1:Nsel
     tournament_indices = randsample(Nind,K);
     [~,max_idx] = max(FitnV(tournament_indices));
     NewChrIx(k) = tournament_indices(max_idx);
   end
end
```

A.4. Other algorithms

A.4.1. Objective function

```
function ObjVal = tspfun(Phen, Dist)
    [nbIndividuals, nbCities] = size(Phen);

    %% vectorized function works ~10 times faster
    Phen = Phen';                           %column order
    Phen2 = [ Phen(2:end,:) ; Phen(1,:) ]; %shifted version
    ObjVal = sum(reshape(Dist(sub2ind(size(Dist), Phen(:), Phen2(:))), ...
                     nbCities, nbIndividuals))';  %transpose to get a row ...
                         vector

    %% using a for loop, semi-vectorized
    %ObjVal = zeros(nbIndividuals,1);
    %for k=1:nbIndividuals
    %  ObjVal(k) = sum(Dist(sub2ind(size(Dist), Phen(k,:), [Phen(k,2:end) ...
        Phen(k,1)])));
    %end
end
```

A.4.2. Nearest neighbor heuristic

```
function [ path ] = nn_heuristic(dist, start)
%NN_HEURISTIC creates a path using Nearest Neighbor heuristic
% Dist is the distance matrix
% start is the city to start from (optional argument)
```

```
n = size(dist,1);
path = zeros(1,n);
unvisited = true(1,n);

[~, dind] = sort(dist,2);
dind = dind(:,2:end); %remove indices of eigencity

%choose random starting city
if (nargin < 2 || start > n) start = randi(n);  end
path(1) = start;
unvisited(start) = false;

for k=2:n
    currInd = find( unvisited(dind(path(k-1),:)), 1, 'first' );
    path(k) = dind(path(k-1), currInd);
    unvisited(path(k)) = false;
end
```

References

[1] M. Affenzeller and S. Winkler. *Genetic algorithms and genetic programming: modern concepts and practical applications*. Numerical insights. CRC Press, 2009.

[2] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, June 2006.

[3] G. Gutin and A. Punnen. *The traveling salesman problem and its variations*. Combinatorial optimization. Kluwer Academic Publishers, 2002.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[5] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[6] Wikipedia, the free encyclopedia. Simulated annealing. `http://en.wikipedia.org/wiki/Simulated_annealing`.