Internal Report 2013–19 August 2013

Universiteit Leiden
Opleiding Informatica

A Genetic Algorithm for the Travelling Salesman Problem with Area Constraints

Ruud Heesterbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The travelling salesman problem (TSP) is a very widely-studied problem up until today. Numerous solutions have been proposed to solve the problem, including genetic algorithms. In this paper, a solution method is created that is also based on a genetic algorithm. In addition, a special constraint is added to the TSP: the area constraint. Borders can be defined in TSP instances, to mark areas that are more difficult to travel through, or more expensive to enter. It will be shown that the proposed genetic algorithm can achieve good solutions for problems with and without these constraints. Also, a comparison is made between a number of genetic operators, some of which use local knowledge to get to a good solution in a very short amount of time.

1 The Travelling Salesman Problem

The travelling salesman problem (TSP) is a very widely studied problem in the field of computer science and mathematics. The problem description is simple: given a number of points (cities), find the shortest route that passes through every city exactly once. The tour needs to end in the city it started in. It turns out, the problem becomes very hard to solve for a large amount of cities, because there are $N!$ possible solutions. In fact, the TSP is an NP-complete problem, which means currently no algorithm exists, that can solve the TSP in polynomial time, and if such an algorithm is found, there exist algorithms to solve every problem in NP in polynomial time. While on the one hand the TSP is a popular problem for academic research, it is also applicable to problems in real life. One can easily think of parcel delivery or mail delivery problems, where a mail carrier has to visit every customer, after which he/she has to return to the post office. Another example of an application of the TSP to a real life problem, is the design of a cable network, where cables need to pass every house and the beginning and end of the cables needs to be at the main station. The TSP is also a sub-problem in fields where you would not necessarily expect it, for example, it is even used in the field of DNA sequencing.

It is easy to understand that the TSP has been a heavily-researched problem. Many algorithms have been proposed to solve the problem. The current state-of-the-art solver is "Concorde" [2]. Its algorithm is designed to find the exact optimal solution for instances of the TSP. It has been used to obtain the optimal solution for the largest instance in TSPLIB [7], which contains 85,900 cities. This is the largest non-trivial TSP-instance that has been solved to optimality. In this paper, a heuristic algorithm is proposed, that tries to give an approximation of the optimal solution, as close as possible to the real optimal solution.

2 Genetic Algorithms

For implementing our TSP-solver, a genetic algorithm will be used. Genetic algorithms are a special type of a larger group of algorithms that are called evolutionary algorithms. These algorithms are inspired by biological evolution and they are based on the process of natural selection. In a genetic algorithm,

there is a population that consists of candidate solutions, often called individuals. As the algorithm runs, these individuals "evolve" towards better solutions. In standard genetic algorithms, individuals are represented as a binary string. Individuals can evolve, by applying crossover and mutation. In a crossover, two individuals are combined into a new individual, usually by taking a portion of the bits of one individual, and a portion of the bits of the other. This process is illustrated in Figure 1. A mutation slightly changes the bits of one individual, usually by inverting one bit. One can imagine that the standard crossover and mutation methods are not very well suited for the TSP, because when you change a bit string that was a legal solution to the TSP, chances are high that the new bit string will not represent a legal solution anymore. This is because solutions to the TSP must contain all cities exactly one time. For the TSP, it is therefore common practice to represent the individuals as a string of integer numbers. It also requires special mutation and crossover operators. These operators will be covered in the next section.

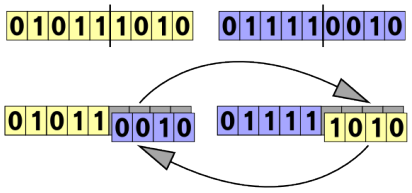

Figure 1: An example of a crossover between two individuals

After new individuals have been created, a new population has to be selected. This can be done by selecting only individuals from the children, or by selecting individuals from both parents and children. The principle of selection is based on Darwin's "survival of the fittest", which means that better solutions have a greater chance of surviving, or making it to the next generation. The fitness of every individual will be calculated by a so called fitness function. For the TSP, the fitness value is the length of the tour that the individual represents. This means that lower fitness values are better. The are a few selection operators to select a new group of individuals based on their fitness value. These will also be discussed in the next section.

3 A Genetic Algorithm for the TSP

In this section, the genetic algorithm that we implemented, will be explained in detail. First, the global structure of the algorithm is described. In the following subsections, all implemented operators will be described in detail.

It is easier to describe the global structure of the algorithm by using pseudocode. The pseudocode can be found in Algorithm 1. The algorithm is started by initializing the population: new individuals are added to the population. The size of the population can be determined by the user. Then, for a fixed number of iterations, a loop will be executed. In that loop, new individuals are created by applying crossover and mutation operators. The crossover rate and mutation rate, both numbers between 0 and 1, determine how often a crossover and a mutation is performed on the population. The algorithm will stop creating

new individuals when the number of new individuals is equal to the population size. After that, individuals are selected from both the old and new generation to form the new population. The advantage of this is that good solutions can survive for multiple generations, which should improve the overall quality of the population. Individuals will be selected based on their fitness. During execution of the loop, the algorithm keeps track of the best individual and when the loop ends, this individual will be the final solution.

```
initialize population;
while iterations left >0 do
    while new generation size <population size do
        if random <crossover rate then
            crossover parents;
            if random <mutation rate then
                mutate child;
            end
        else if random <mutation rate then
            mutate parent;
        end
    end
    update best solution;
    select population;
end
return best solution;
```

Algorithm 1: The Genetic Algorithm

3.1 Initialization

A few well known operators have been implemented to initialize the population.

- Random. This is the easiest and most straightforward way to initialize a population. The route will be a random permutation of the cities in the particular TSP instance. This operator obviously does not require a lot of computation time, but the solutions will be very poor.

- Nearest Neighbour. This operator generates tours by always picking the city that is closest to the current city. Starting at a random node, the next city in the tour will be the city that has the smallest distance to the previous city. This process will be repeated until every city has been added to the tour.

- Nearest Insertion. First, a random permutation of the cities has to be generated. Cities will then be added to the tour in the order of the permutation. However, their position in the tour will be where the increase in length of the tour is the smallest. In other words, city i will be placed between the cities a and b for which distance(a,i) + distance(i,b) - distance(a,b) has the smallest value. This principle of inserting a city in a tour is called nearest insertion.

The last two initialization operators require a lot more computation time — $O(N^2)$ — than random intitialization, which only takes $O(N)$ time, but the tours they generate are of a significantly higher level.

3.2 Crossover

Three different crossover operators have been implemented.

- Edge Recombination Crossover. This operator was first introduced by Whitley et al. [9]. It tries to preserve as much of the edges of the parent tours as possible. To avoid cities being left out, it will select cities with the least amount of different edges in the parent tours first. These cities endure the highest risk of becoming isolated. The ER operator works with a so called edge map. The edge map is a list that tells for every city which edges it has in the parent tours. If we take for example the two tours ABCDEF and FBAEDC, the algorithm will start with the edge map in Table 1. As you see, city F has four entries in the edge map, because it is connected to city A and E in the first tour and to city C and B in the second tour. City D only has two entries in the edge map, because it is connected to city C and E in both tours. City D will probably be selected fairly early by the algorithm, because it has the least amount of edges in its edge map.

  Table 1: An example of an edge map

City	Edges
A	B, E, F
B	A, C, F
C	B, D, F
D	C, E
E	A, D, F
F	A, B, C, E

  The edge map will be updated constantly while the tour is constructed, according to the cities that have been chosen: only edges that still have the possibility to be added to the tour will be kept in the list. At the moment that a city has been chosen by the algorithmin for the new tour, all its edges will be erased from the edge map. The algorithm works as follows:

  1. Choose a random starting city.
  2. Remove all occurrences of edges that contain the current city from the edge map.
  3. Pick the city in the current city's edge list, that has the least amount of cities in its own edge list. At a tie, decide randomly. If there are no cities in the current city's edge list, choose a random remaining city as the next current city.
  4. Repeat step 2 and 3 until all cities have been chosen.

- Order Crossover. The order crossover operator was introduced by Davis [3]. The algorithm assumes that only the order of the cities in a tour

is important, not the position that they are in. It chooses a subtour of cities from one parent, and puts these cities in the order that they appear in the other parent. First, a subtour of the first parent is selected, that will be copied to the child. The remaining cities will be added after this subtour, in the order that they appear in the second parent. Figure 2 further illustrates this process. The cities of subtour 7-8-9-1-2 are put in the order in which they appear in the second parent.

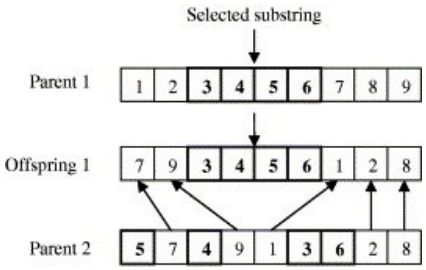

Figure 2: An example of order crossover

- NI-combined Crossover. This operator was proposed by Sakurai et al. [8]. The basic principle is the same as in the previously discussed order crossover, but to improve convergence speed of the genetic algorithm, cities are now added using the nearest insertion method. The first step of the algorithm is to select a subtour of cities from the first parent. These cities will be copied to the child. The remaining nodes will be added to the child using nearest insertion, in the order that the cities occur in the second parent. To improve performance in larger problem instances, we have made a slight alteration to the original operator. The first subtour that is copied to the child, will be chosen in such a way, that the remaining cities will not be more than fifty in total. By doing this, the operator will preserve a larger part of the parent's tour, which increases the chance that the new tour is an improvement over the old tours. Moreover, this alteration to the algorithm decreases the computation time that is needs, because the number of cities that has to be added with the nearest insertion method, is now limited.

Notice that the first two crossover operators are very fast, traditional genetic operators. They do not use any extra knowlegde of the problem and they only require $O(N)$ computation time. The NI-combined crossover method is different. It uses local knowledge (the distance between each pair of cities) to create new individuals and the computation time is a lot longer. Because the number of cities to insert is limited to fifty, the operator can finish in $O(N)$ computation time, but with a large coefficient before the N ($O(50N)$).

3.3 Mutation

Four different mutation operators have been implemented in the solver.

- Insertion Mutation. The insertion mutation operator is based on a very simple principle. It randomly selects a city, removes it from the parent's

tour and then chooses a random position to insert the city back in the tour. It was proposed by Fogel [4].

- Displacement Mutation. This operator starts by selecting a random subtour. This subtour is then removed from the parent's tour and inserted at a random place. It was proposed by Michalewicz [6]. Figure 3 shows an example of how the operator works.

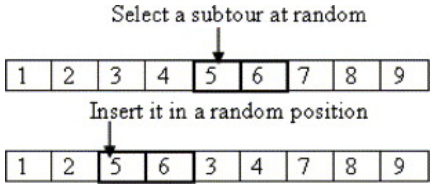

Figure 3: An example of the displacement mutation operator

- 2-Opt Mutation. This method has been proposed by Sakurai et al. [8]. It is based on simple inversion mutation, Figure 4 (Holland [5]), but it has been slightly changed to improve the convergence speed of the genetic algorithm. First, a random subtour will be selected from the parent. The cities in this subtour will then be put in reverse order. The change over the original algorithm is the following: if the new tour is an improvement over the old tour, the operator will be applied again. It will stop when no improvement has been made, and the end result will be the last tour that has been an improvement over the previous one.

Figure 4: An example of simple inversion mutation

- Block-type Mutation. This method has also been proposed by Sakurai et al. [8]. The operator is applied on a random geographical block in the tour. First, a random city is chosen as the centre of the block. After that, the algorithm determines a random neighbourhood by taking the distance from the city in the centre to the next city in the tour and multiplying it by a random number from 1 to 5. This number will be the neighbourhood size. Every city that has a distance to the centre node that is smaller than the neighbourhood size, will be removed from the tour. After that, all nodes that have been removed, will be inserted into the remaining tour using the nearest insertion method. We have made a small alteration to the operator, the same alteration we made to the NI-combined crossover operator: we only allow the operator to remove a maximum of fifty cities from the tour. Removing more cities will destroy too much of the original tour, but more importantly, it is very time-consuming to insert a high number of cities back into the tour using the nearest insertion method.

Just as with the crossover operators, a few key differences between the operators can be pointed out. The first two mutation operators (insertion and displacement) are fast, traditional operators that only require $O(N)$ computation time.

The last two operators use local knowlegde to create a new individual. It is hard to comment on the computation time of the 2-opt operator. There are $O(N^2)$ pairs of edges, and inverting a pair of edges requires $O(N)$ computation time, so the operator will certainly finish in $O(N^3)$ computation time. However, in practice the operator will often be applied to very good tours, so that it will only execute one or two iterations. In practice, block-type mutation will be the slowest of the four, because it uses the nearest insertion method, which is very time consuming. Just as with the NI-combined crossover operator, the number of cities that is inserted is limited to fifty, so the operator can finish in $O(N)$ computation time, but again with a large coefficient before the N ($O(50N)$).

3.4 Selection

Two different selection operators have been implemented in the solver. The operators will select the new individuals from both the parents and the children. As an extra rule, copies of the same individual are only allowed once in the population, to keep the population more diverse. The implemented operators are the following:

- Tournament Selection. In tournament selection, a number of individuals, which we call the tournament size, will be randomly picked from the candidate solutions. The fittest individual, the one with the best solution, will be put into the new population. This process is repeated until the population has reached the required size.

- Exponential Ranking Selection. This operator starts by ranking all candidate solutions according to their fitness value. After that, there will be several rounds in which an individual is picked and put into the population. At each round, the fittest individual has the highest chance to be picked and the individual with the worst solution has the lowest chance to be picked. We have chosen the distribution of the chances over the individuals to be exponential (Blickle, [1]). The probability p_i that individual i will be picked, can be calculated by the following formula:

$$p_i = \frac{c-1}{c^{N-1}} c^{N-i} \tag{1}$$

  i is the rank of the individual from 1 to N: the fittest individual gets rank N, while the worst individual gets rank 1. The parameter c is a value between 0 and 1 and is called the selection bias. A value closer to 0 leads to more exponential behaviour of the operator, while a value closer to 1 distributes the chances over the individuals more equally.

4 Area constraints

In this section, the area constraint that has been added to the TSP will be discussed. A feature has been added in the solver to add borders to a TSP instance. The borders make it more expensive to travel from a certain area to another. It relates closely to scenarios that you see in real life. For example, it could be seen as travelling to different countries. It often costs more time or even money to travel from one country to another, than to travel between two

cities in the same country. It could also be seen as an obstacle in nature, that is more difficult to pass during a travel. For example a river, or mountains, often require more time or effort to cross. It could also be the case that there simply is no direct route from one city to another, because it is not possible to travel through the area between the cities. In that case, the borders can be used to block that part of the map, so that specific paths cannot appear in the final solution.

The borders in the TSP solver are defined by a few simple rules. As a first rule, travelling from one city to another can only happen in a straight line. If the resulting path crosses one or more borders, a penalty will be added to the distance of the path, for each border that it crosses. This penalty is customizable in the algorithm, but it should be the same penalty for each individual border.

5 Experiments and Results

In this section, the results of the experiments that have been performed to compare and evaluate the operators, will be shown. The algorithm has so many parameters that one can tune, that it is impossible to test everything. A selection of tests have been made and these tests were run on eight different problems. Four of these eight problems have additional area constraints. Each combination of operators and parameters was run 20 times and from these 20 runs, the average solution quality and the standard deviation were calculated. All runs in which a crossover operator was tested, were run with the same combination of mutation operators: insert mutation, displacement mutation and 2-opt mutation. These operators were alternated by the algorithm on a random basis. In the runs in which a mutation operator was tested, the edge recombination operator was used as crossover operator. This operator is fast and gives relatively good solutions, so it is a good test to compare the different mutation operators against each other. Each combination has been tested with different population sizes, and two different combinations of initialization. Finally, tournament selection was used as the selection operator and each test was performed with the following values of the remaining operators:

- Crossover rate: 0.9
- Mutation rate: 0.2
- Tournament size: 4
- Penalty for crossing a border: 1000
- Number of iterations: 200 times the number of cities. Because the complexity of the problem is directly correlated to the number of cities, the algorithm was given more time on larger problem instances. However, when the best tour did not improve for 10,000 consecutive generations, the algorithm was stopped prematurely to start the next run.

Four different problems have been selected from TSPLIB, a library of TSP problems from various sources [7], and each of these problems has been tested with and without area constraints. The problems that were picked were: berlin52, bier127, gil262 and rd400. The numbers in these problem names indicate the number of cities. The results of the bier127 problem can be found in Tables 2

and 3. The results of the other problems can be found in the Appendix section. Note that for the problems without constraints, the optimal solution is known. The optimal solution for the problems with area constraints is unknown, but it is possible to compare the results to the best solution that has been found by our algorithm.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	124914	123824	121463	120664	122687	121814	120108	119113
	Std. dev.	2207.1	1822.7	1429.2	1035.2	2455.3	2228.4	1066.8	382.2
Order C.	Average	191559	173269	162948	155878	127218	125408	125411	124444
	Std. dev.	5707.1	3443.2	3584.0	2735.1	1396.3	1257.4	868.9	832.6
NI-Co. C.	Average	118704	118557	118500	118411	118545	118544	118448	**118402**
	Std. dev.	410.3	290.1	224.6	148.4	284.0	404.6	153.1	145.3
Insert M.	Average	127204	125089	123349	121681	123632	121889	120785	119690
	Std. dev.	3261.8	2750.6	1947.9	1500.7	2549.1	1308.5	1580.0	1026.0
Disp. M	Average	131474	125539	122503	121340	123946	121515	120439	119256
	Std. dev.	2823.6	2978.9	1647.1	1227.8	2003.1	1716.9	1570.1	713.3
2-OPT M.	Average	128731	128148	127153	124223	123495	122427	120474	119476
	Std. dev.	2801.3	2938.1	2581.4	1653.8	1723.6	2017.2	1442.0	484.8
Block M.	Average	119131	120113	120820	120603	119194	119894	120910	120671
	Std. dev.	699.9	974.6	325.1	378.9	658.2	1242.2	300.0	432.4

Table 2: Results of the bier127 problem without constraints. The optimal solution is known to be 118282. It has in fact been found numerous times by our algorithm.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	136581	133279	132377	131292	134191	132306	130338	129783
	Std. dev.	2879.1	2502.0	2163.8	1123.9	2330.7	1988.0	1329.4	1104.7
Order C.	Average	217019	196294	181517	171465	138161	136938	136232	135202
	Std. dev.	7449.7	5469.9	6593.2	5360.0	2396.1	2073.2	1553.5	981.1
NI-Co. C.	Average	128445	128474	128157	128067	128610	128464	128218	**127971**
	Std. dev.	545.3	586.5	566.5	369.1	642.7	534.3	443.6	222.6
Insert M.	Average	139208	133758	133056	131835	135090	133349	131783	130294
	Std. dev.	4413.4	3166.5	2172.5	2416.4	2078.2	2134.7	1172.0	1037.2
Disp. M	Average	139903	135594	133623	131844	133165	132417	130997	130006
	Std. dev.	4185.5	2980.6	3382.6	1370.2	2465.2	2335.5	1433.7	1178.1
2-OPT M.	Average	139987	139489	135527	133856	135010	132332	131678	130173
	Std. dev.	2282.8	3525.2	2850.8	2314.5	2050.3	1911.8	1704.4	889.0
Block M.	Average	128818	130960	130957	130404	129367	130195	130981	130586
	Std. dev.	883.6	932.6	455.2	501.9	888.4	1292.2	351.2	502.4

Table 3: Results of the bier127 problem with constraints. The best result found by the algorithm in one run was 127762.

It can be concluded from Tables 2 and 3 that the NI-combined crossover operator performes best. It outperformed the other two crossover operators on every combination of operators and parameters that was tested. Its best average

result in our experiments was only 1.0% away from the optimal solution. Order crossover performed worst of the three. It needs a very large population size to achieve a decent result. With a population size of 20 and random initialization, its average result was 62% worse than the optimal solution on the problem without constraints. With constraints, its average result was even 69% worse than the best solution that was found by our algorithm. It can also be concluded that the best performing mutation operator is block-type mutation. Some interesting behaviour by the operator can be observed: it performes better when the population size is smaller. Its results with a population size of 20 are better than what the other operators achieve with a population size of 200, although the differences are small. There are only small differences between the results of the other three mutation operators. It can be said that the displacement mutation operator slightly outperformes the other two when population sizes are large, but all three achieve very similar results. Tables 2 and 3 also show that initializing the population with the nearest neighbour and nearest insertion operators, almost always improves the solution quality of the algorithm.

From comparing the problem without constraints to the problem with constraints, the following can be concluded: the added constraints do not affect the performance of the operators. The NI-combined crossover operator is the best operator on both problems and the block-type mutation operator is the highest performing mutation operator on both problems. As can be seen from Tables 2 and 3, the operators that performed well on the problem without constraints, also performed well on the problem with constraints, and the same can be said for operators did not perform well.

The results of the other problems that were tested, can be found in the Appendix section. A lot of the results are similar to the results of the bier127 problem, in the sense that the same operators got the best results, so they support the conclusions that were drawn based on the bier127 problem. However, a few interesting differences can be pointed out. Tables 7, 8, 9 and 10 show that the edge recombination crossover operator performes a lot worse when the population size is large and random initialization is used. The mutation operators were tested with the edge recombination operator as well and therefore the results of these operators are rather strange as well. The 2-opt operator often achieves the best results on these occasions. It can also be seen that on these problems, the results when a population is initialized by neighrest neighbour and nearest insertion are all very similar. This is probably because the operators struggle to improve the solutions that were created by these initialization operators. It can be seen that here the solution quality of the NI-combined crossover operator is higher than the solution quality of the other operators as well.

A final experiment will be shown, in which we compare the results of the algorithm to the optimal solution that is known for the four problems without constraints. For this experiment the NI-combined crossover operator is used, combined with two mutation operators, namely 2-opt and block-type mutation. The population size has been set to 25, to keep computation time reasonable. Again, tournament selection is used as the selection operator and the other parameters are the same as in the previous experiment. Each combination has been run 20 times and Table 4 shows the results of these experiments.

Only for the rd400 problem, the algorithm has not found the optimal solution. The optimal solution for rd400 is known to be 15281, according to TSPLIB.

Initialization		Random	N. Neighbour	N. Insertion	N. N. + N. I.
berlin52	Best	**7542**	**7542**	**7542**	**7542**
	Average	**7542.0**	**7542.0**	**7542.0**	**7542.0**
	Std. dev.	0.0	0.0	0.0	0.0
bier127	Best	**118282**	**118282**	**118282**	**118282**
	Average	118628	118654	**118496**	118630
	Std. dev.	420.3	478.3	214.8	410.0
gil262	Best	**2378**	2380	2383	2380
	Average	2408.3	2398.2	2404.0	**2396.9**
	Std. dev.	23.8	13.4	11.6	9.4
rd400	Best	15502	15385	15327	**15314**
	Average	15572	15553	15453	**15440**
	Std. dev.	43.1	84.0	61.0	58.6

Table 4: Results of the final experiments in which it is tried to achieve near-optimal solutions with the algorithm.

For the other three problems, the algorithm found the optimal solution at least once. The method of initialization has little influence on the end result. On the largest problem, we can see that nearest insertion initialization achieves a slightly better average solution than initialization by nearest neighbour, while a combination of the two achieved the best average on three of the four problems. On the largest problem, the best combination of operators was only 1.0% away from the final solution, on average. It can be concluded from the standard deviations that the algorithm performs very consistent as well. On every run it will come very close to the optimal solution.

6 Conclusion

A genetic algorithm has been presented to solve the travelling salesman problem with the addition of area constraints. The experiments show that NI-combined crossover and block-type mutation generate the highest quality tours. They are both based on the nearest insertion method and while they require a bit more computation time, they improve tours very quickly. It has been shown that the algorithm can achieve very good results, even for problems with up to 400 cities. On the rd400 problem, the algorithm was only 1.0% away from the optimal solution on average. It was also shown that the algorithm can achieve good results when area constraint are added to the TSP. NI-combined crossover and block-type mutation also achieved the best results in this situation. The low standard deviations also show that these operators are very consistent: they produce a high quality tour on every run. While the optimal solutions of these problems with area constraints is unknown, it is possible to assess the solutions by hand and it can be seen that the solutions look good. Figure 5 shows a picture of the solver. It has been solving a problem with area constraints and the solution looks like it is very close to the optimal solution.

7 Future Work

While the current algorithm achieves good results, the genetic operators could still be improved further. The experiments suggest that the operators that

are based on nearest insertion work well. It would be a good addition to the solver to add more of these operators. Right now, it only has one crossover and one mutation operator that utilizes this method. It could also be possible to hybridize operators with the nearest neighbour method. For example, the edge recombination crossover operator currently chooses a random city to go to next, when a city's edge list is empty. Maybe it would improve the operator, if it instead would choose the city that is nearest to the current city.

It would also be interesting to add more real life constraints to the solver. A good example would be precendence constraints, where city a needs to be visited before city b. It could also be turned into a time constraint, where the constraint would be that city a and b need to be visited within the first 20 cities, or within the first 50 kilometres travelled.

References

[1] T. Blickle & L. Thiele. A Comparison of Selection Schemes used in Genetic Algorithms. *TIK-Report No. 11*. Computer Engineering and Communication Networks Lab (TIK). Swiss Federal Institute of Technology (ETH) Zürich, Switzerland (1995).

[2] W. Cook. Concorde TSP Solver. Mathematics — University of Waterloo, `www.math.uwaterloo.ca/tsp/concorde/` (2011).

[3] L. Davis. Applying Adaptive Algorithms to Epistatic Domains. *Proceedings of the International Joint Conference on Artificial Intelligence*, 162–164. Morgan Kaufmann Publishers, San Francisco, CA (1985).

[4] D. B. Fogel. An Evolutionary Approach to the Traveling Salesman Problem. *Biological Cybernetics* 60: 139–144. Springer-Verlag, Berlin (1988).

[5] J. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975).

[6] Z. Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, Berlin (1992).

[7] G. Reinelt. TSPLIB. Institut für Informatik — Universität Heidelberg, `comopt.ifi.uni-heidelberg.de/software/TSPLIB95/` (2013).

[8] Y. Sakurai, T. Onoyama, S. Kubota, S. Tsuruta. A multi-inner-world Genetic Algorithm to optimize delivery problem with interactive-time. In D. Davendra, *Traveling Salesman Problem, Theory and Applications*: 137–154. In-Tech (2010).

[9] D. Whitley, T. Starkweather & D'Ann Fuquay. Scheduling Problems and Travelling Salesman: The Genetic Edge Recombination Operator. In J. Schaffer (ed.) *Proceeding on the Third International Conference on Genetic Algorithms*: 133–140. Morgan Kaufmann Publishers, Los Altos, CA (1989).

8 Appendix

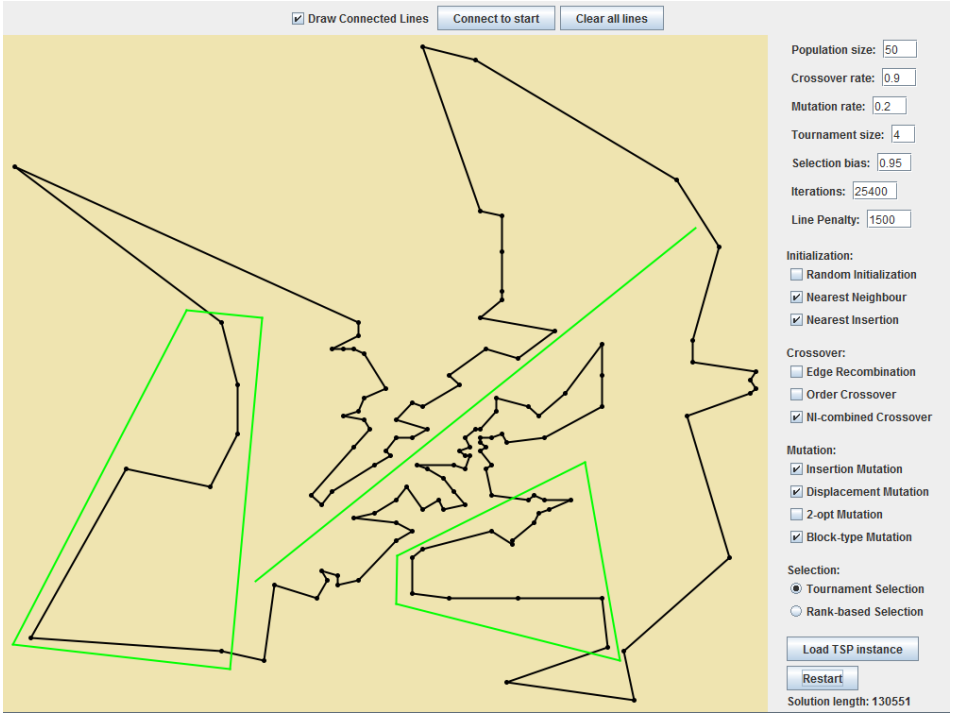

Figure 5: A solution of the solver on the bier127 problem with area constraints.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	7861.9	7679.9	7632.9	7588.9	7627.5	7553.7	**7542.0**	**7542.0**
	Std. dev.	287.7	197.9	148.3	115.2	136.0	50.8	0.0	0.0
Order C.	Average	8946.3	8379.9	8115.7	7838.4	7982.5	7821.6	7742.4	7646.5
	Std. dev.	283.5	215.5	196.3	198.9	158.8	154.3	115.9	76.3
NI-Co. C.	Average	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**
	Std. dev.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Insert M.	Average	7890.0	7756.7	7691.8	7570.0	7603.0	**7542.0**	**7542.0**	**7542.0**
	Std. dev.	217.2	228.6	206.7	90.6	131.5	0.0	0.0	0.0
Disp. M	Average	7852.2	7740.6	7693.8	**7542.0**	7616.4	**7542.0**	**7542.0**	**7542.0**
	Std. dev.	202.1	144.1	211.6	0.0	120.7	0.0	0.0	0.0
2-Opt M.	Average	8205.8	8100.1	7776.0	7700.9	7651.0	7559.4	7553.7	**7542.0**
	Std. dev.	295.0	288.8	150.8	158.6	146.4	75.6	50.8	0.0
Block M.	Average	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**	**7542.0**
	Std. dev.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 5: Results of the berlin52 problem without constraints. The optimal solution is known to be 7542 and the algorithm found that solution a lot of times.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	12279	12127	12114	12035	12136	12047	12035	12016
	Std. dev.	539.0	195.8	249.5	58.4	135.8	56.5	58.4	11.7
Order C.	Average	13502	13014	12535	12524	12675	12375	12282	12214
	Std. dev.	393.1	434	76.5	293.6	340.9	123.6	149.6	80.5
NI-Co. C.	Average	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**
	Std. dev.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Insert M.	Average	12702	12166	12138	12029	12154	12088	12022	12014
	Std. dev.	892.6	221.3	299.9	58.5	139.8	105.2	16.9	8.5
Disp. M	Average	12712	12345	12088	12027	12142	12058	12044	12018
	Std. dev.	747.4	605.2	165.0	58.4	137.1	89.1	66.8	13.9
2-OPT M.	Average	12706	12264	12250	12056	12151.1	12099	12048	12026
	Std. dev.	814.6	218.4	275.8	82.7	158.9	130.7	69.0	52.5
Block M.	Average	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**	**12012**
	Std. dev.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 6: Results of the berlin52 problem with constraints. The best result found by the algorithm in one run was 12012. Because this solution was found so many times by the algorithm, it is very likely that this is in fact the optimal solution.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	2655.5	2570.2	7075.9	7708.4	2554.2	2543.4	2546.0	2530.1
	Std. dev.	51.4	38.7	2964.7	154.4	34.7	29.2	19.2	23.8
Order C.	Average	8523.7	7629.3	6976.5	6347.7	2576.9	2552.9	2541.7	2540.0
	Std. dev.	178.1	159.1	180.0	131.9	24.4	22.3	21.3	14.7
NI-Co. C.	Average	2414.1	2403.1	2398.0	**2387.4**	2404.7	2389.8	2391.0	2390.0
	Std. dev.	13.9	19.6	9.5	7.7	10.3	8.3	8.4	6.3
Insert M.	Average	2686.6	2639.7	9307.2	7869.0	2561.1	2544.3	2542.7	2524.9
	Std. dev.	52.9	71.1	216.3	155.2	34.0	23.1	19.2	21.8
Disp. M	Average	2997.8	2742.9	9022.2	7959.5	2563.4	2560.1	2540.9	2529.6
	Std. dev.	91.3	69.0	1500.3	197.4	26.5	20.6	22.2	27.5
2-OPT M.	Average	2690.3	2682.5	6528.6	5196.1	2563.7	2544.7	2545.2	2536.5
	Std. dev.	54.9	52.0	2841.8	2331.7	21.6	23.2	19.5	24.2
Block M.	Average	2674.3	2819.8	2809.1	2789.2	2529.1	2558.7	2544.0	2539.6
	Std. dev.	202.5	41.8	33.6	48.0	65.3	20.0	13.6	13.1

Table 7: Results of the gil262 problem without constraints. The optimal solution is known to be 2378 and the algorithm was able to find that solution as well, with the NI-combined crossover operator.

Population Size		20	50	100	200	20	50	100	200
Initialization		Random				N. Neighbour + N. Insertion			
Operator	Result								
Edge R. C.	Average	10980	9434	64897	67487	8792	8740	8735	8735
	Std. dev.	1519.5	957.6	32504	2390	63.3	60.1	41.7	31.8
Order C.	Average	55339	40994	33622	21573	8823	8777	8750	8739
	Std. dev.	3249.9	4290.9	3338.0	2241.6	124.9	35.2	28.9	25.3
NI-Co. C.	Average	8568	8558	8544.7	**8543**	8565	8569	8550	8544
	Std. dev.	26.6	20.4	18.9	13.2	21.9	15.8	16.4	11.9
Insert M.	Average	12521	17678	84262	68341	8784	8762	8746	8732
	Std. dev.	2090.7	19095	3827.0	2783.8	49.1	48.1	34.2	33.1
Disp. M	Average	10626	20427	85262	68655	8797	8764	8758	8736
	Std. dev.	1250.4	24928	2586.2	3180.5	46.7	31.7	19.2	33.0
2-OPT M.	Average	12371	11492	61793	63687	8773	8760	8750	8737
	Std. dev.	1684.6	1612.8	29504	3000.1	55.6	36.2	30.2	31.2
Block M.	Average	13314	12187	11119	10586	8798	8754	8737	8734
	Std. dev.	1272.8	1295.8	749.6	880.4	95.4	60.3	26.2	22.1

Table 8: Results of the gil262 problem with constraints. The best result found by the algorithm in one run was 8525.

Population Size		20	50	100	20	50	100
Initialization		Random			N. Neighbour + N. Insertion		
Operator	Result						
Edge R. C.	Average	17992	32945	80857	16687	16581	16471
	Std. dev.	211.0	32539	27397	120.3	105.3	203.1
Order C.	Average	69745	63337	58539	16692	16577	16516
	Std. dev.	1035.0	1224.4	1393	122.0	91.1	48.3
NI-Co. C.	Average	15616	15606	15496	15476	15449	**15411**
	Std. dev.	98.7	102.9	68.2	62.5	51.5	48.5
Insert M.	Average	17734	101850	96322	16542	16601	16527
	Std. dev.	280.9	1884.4	1229.3	210.0	103.4	99.4
Disp. M	Average	21517	93497	95957	16645	16587	16554
	Std. dev.	535.5	24928	1372.2	107.1	83.1	85.6
2-OPT M.	Average	17511	17175	85680	16635	16582	16545
	Std. dev.	216.2	331.5	22890	168.2	108.7	97.0
Block M.	Average	27044	26783	25505	16277	16535	16541
	Std. dev.	2797.3	1264.2	1202.6	486.3	292.0	75.0

Table 9: Results of the rd400 problem without constraints. The optimal solution is known to be 15281, but the algorithm was not able to find this solution in this experiment.

Population Size		20	50	100	20	50	100
Initialization		Random			N. Neighbour + N. Insertion		
Operator	Result						
Edge R. C.	Average	32895	43916	322611	28190	26918	27032
	Std. dev.	2159.3	64180	98678	937.7	781.3	657.5
Order C.	Average	213199	183856	160577	28337	27635	27244
	Std. dev.	6131.6	5559.2	3760.5	693.6	672.0	641.1
NI-Co. C.	Average	24661	24677	24397	24581	24562	**24270**
	Std. dev.	446.8	483.9	390.5	471.7	453.9	110.5
Insert M.	Average	36446	376305	361834	28335	27509	26962
	Std. dev.	3344.2	18602	5001.7	814.6	725.4	613.7
Disp. M	Average	38583	361940	362843	28401	27686	27250
	Std. dev.	2905.6	77861	7686.8	757.8	957.0	549.2
2-OPT M.	Average	34539	78564	311444	28436	27396	26883
	Std. dev.	2836.0	109103	93939	1433.1	678.8	519.4
Block M.	Average	83473	79458	72993	27587	27384	26832
	Std. dev.	10212	6056.8	5834.3	1531.6	969.4	399.3

Table 10: Results of the rd400 problem with constraints. The best result found by the algorithm in one run was 24076.