Genetic Algorithms and Evolutionary Computing: Genetic algorithm performance on TSP

Verheyen Willem R0624869
Vandebeek Sjaan R0624482

Contents

Abstract

This report handles a solution to the TSP problem using genetic algorithms. Experiments with different variation parameter settings are experimented upon to increase the efficiency and resulting fitness values. Different methods such as order based crossover, reverse sequence mutation, different selection methods are discussed. The influence of multiple stop criteria and an approach to improve the solution using a local optimisation are handled. Different methods are tested against a benchmark.

1. Existing genetic algorithm

Different parameters settings for elitism, mutation, crossover and population are tested on the "rondrit127" dataset which consists of 127 cities. The algorithm runs 10 times and the results are averaged over those 10 runs. In the results, the mean fitness value of the 10 runs and the best fitness value of these 10 runs are considered. Figures are labelled using the following format: [Population, elitism, crossover rate, mutation rate].

The following parameters and their associated values are are considered in the experiments:

- Maximum generations: 200
- Elitism: [0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1]
- Crossover rate: [0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1]
- Mutation rate: [0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1]
- Population : [20 50 100 200]

Crossover produces two off-springs and typically has a higher probability ranging form [0.8-0.99]. While mutation rate has a typically low mutation, [0.01-0.2]. As we can see in figure 1, 2 having a higher mutation rate pays of when the population has a lower number of individuals, this is because with a low population there is not a significant diversity in the population, mutation allows slight differences in the genotypes to encourage diversity. However because variation parameter settings depend on the problem we experiment with larger intervals.

In genetic algorithms it is important to find a proper balance between exploration (searching in the search space) and exploitation (concentrating on a part of the search space, hopefully a global optimum). Mutation enhances exploration while crossover is more exploitative, therefore we want a good trade-off between exploitation and exploration. At the start, exploration is more important since we want to explore much more of the search space to ensure we cover a large part of the possible solutions. In this manner, a high population diversity is assured. Later on in the training process we want to focus on exploitation, such that our better individuals can converge into a global optimum. Therefore we use a higher crossover rate and a lower mutation rate.
If we have a very small mutation rate we might end up with premature convergence. However both mutation rate and crossover rate are very problem specific and a high crossover and low mutation are not always considered the best settings. Therefore we now try to find a good combination of the parameters.
While testing different parameters we came to the conclusion that when dealing with very large populations a low mutation rate resulted in , as we previously stated, premature convergence, therefore we tested out a larger interval for both mutation and combination. We experiment using a GENERATE-and-TEST approach.

When using a higher population, the amount of generations decreases significantly before obtaining a good result. Using smaller populations also decreases diversity which will result in premature convergence and thus not utilizing a large search space. Populations smaller than 100 resulted in very slow increase in fitness or even premature convergence, larger population sizes than 100 had a faster increase of fitness but also increased computation time significantly. Therefore we decided the optimal population size of 100 individuals.

We determine the best parameter for elitism. Figure 5 shows the influence of the elitism parameter on the best fitness value of the population. It is obvious that a lower elitism results in longer computation time, this is also shown in table 1. Table 1 shows the best fitness values for corresponding elitism settings, we can conclude that the best setting for elitism is 0.1.

Figure 3 shows the correlation between crossover and mutation on the resulting time of one training cycle. Figure 4 shows the correlation between crossover and mutation on the resulting best fitness of all 10 training cycles using a population of 100 and elitism 0.1. We see for a low mutation rate and low crossover rate there is less computing time necessary than for large crossover and mutation rates.
The optimal values for the crossover and the mutation rate can be derived from figure 4. A lower value for crossover

gives better results than higher values therefore we consider the interval [0.2-0.4] for the crossover as optimal. We can also see that the mutation rate does not influence the result as much as crossover does.
When dealing with smaller populations, it pays of to have a larger mutation rate, but when dealing with larger population sizes, the mutation rate does not influence the resulting fitness for a significant amount.

We use the best parameters discussed above and run the algorithm 10 times. The results of the best run are shown in figure 6. We reach a fitness value of 6.3058 after 2000 generations in 13.6 seconds.

- Elitism: 0.1
- Crossover rate: 0.3
- Mutation rate: 0.5
- Population: 100

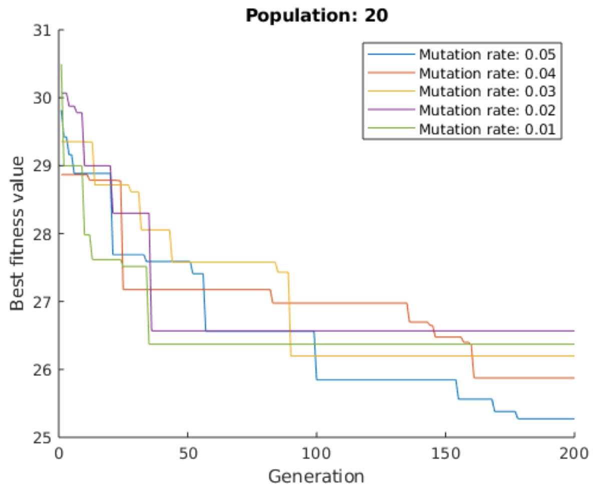

Fig. 1: [20, 0.5, 0.9, var], Best fitness for different mutation parameters

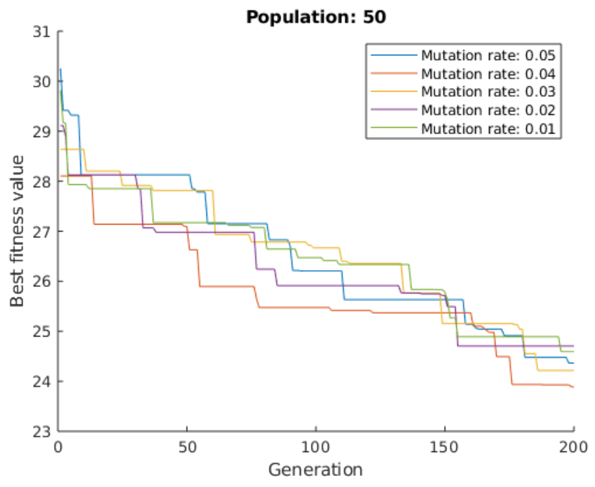

Fig. 2: [50, 0.5, 0.9, var], Best fitness for different mutation parameters

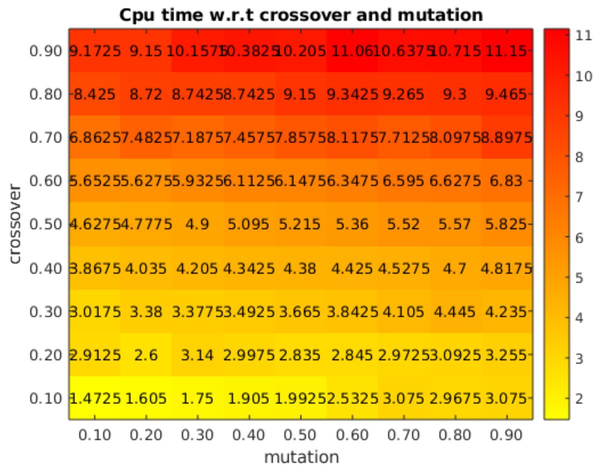

Fig. 3: Correlation of crossover and mutation on average cpu time

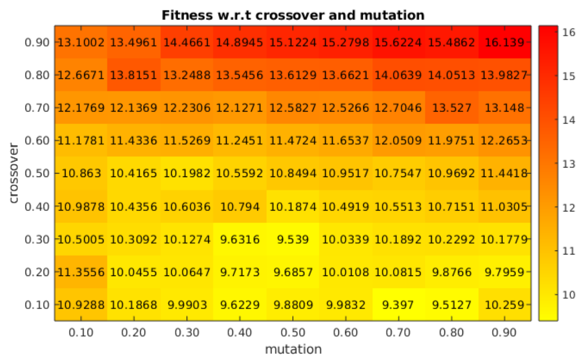

Fig. 4: Correlation of crossover and mutation on best fitness

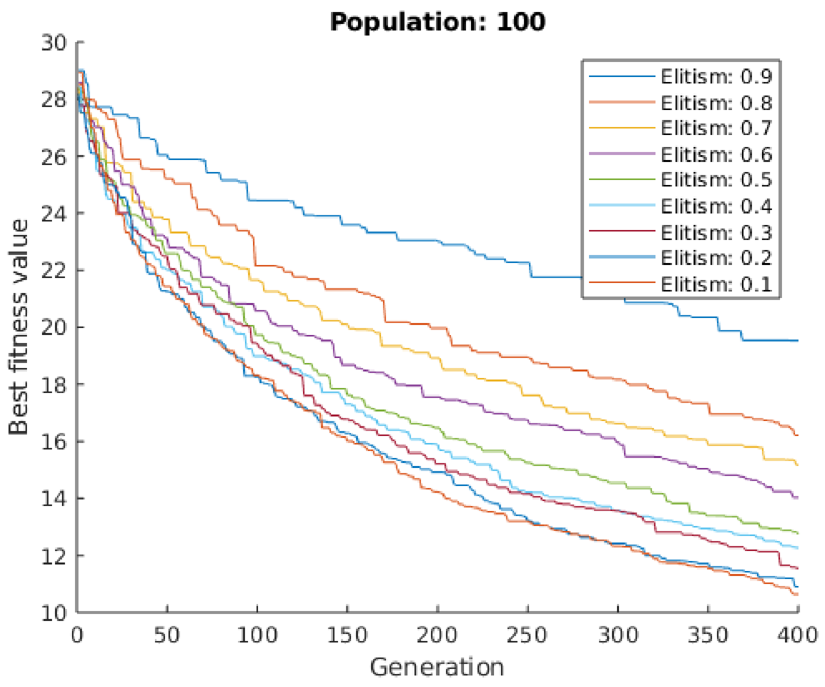

Fig. 5: [400,var,0.4,0.4] Best fitness w.r.t elitism parameter

		Elitism								
		0.90	0.80	0.70	0.60	0.50	0.40	0.30	0.20	0.10
200 generations	Best Fitness	22.635	20.0518	18.067	17.196	15.999	16.1332	15.2926	15.119	14.5294
	Cpu time	0.375	0.517	0.695	0.9579	1.034	1.2640	1.4340	1.5510	1.766
400 generations	Best Fitness	19.525	16.209	15.17	14.033	12.756	12.271	11.561	10.9031	10.644
	Cpu time	0.643	0.930	1.387	1.704	2.011	2.377	2.725	3.243	3.746

Table 1: Result of elitism parameter on best fitness for 200 and 400 generations.

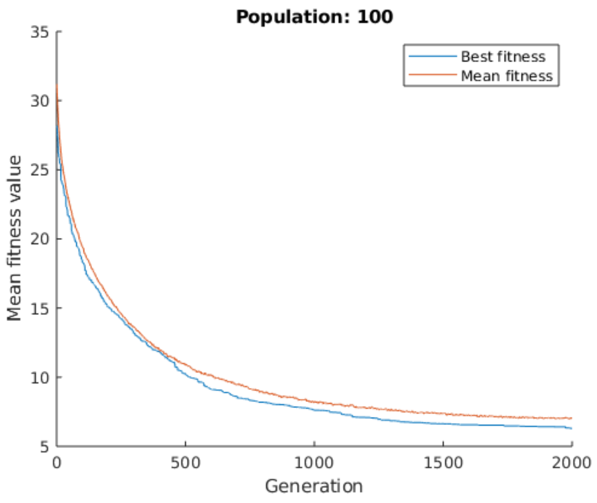

Fig. 6: [100, 0.1, 0.3, 0.5], Best parameters settings for default method

2. Stopping criterion

When does a genetic algorithm have to stop? For a np-hard problem where no optimal solution is known, an unambiguous answer does not exist. In this section we consider stop criteria that are deterministic. We do not contemplate stochastic or self-adaptive stopping criteria.
First some stop criteria are proposed and a summary of their characteristics is given. Then some criteria are fine tuned with respect to the efficiency of a GA.

2.1. Stop criteria

In our project we consider stop criteria of the following kind: the algorithm is stopped when a property of the population reaches a predefined limit or doesn't change for a predefined value of generations. The properties that we use are:

1. The percentage of chromosomes that are equal to the best chromosome of the population.
2. The mean fitness value of the population.
3. The fitness value of the best solution.

These properties are visualized in figure 7. With these three properties the following three stopping criteria are constructed:

1. Stop when X percent of the population has the same fitness as the best chromosome.
2. Stop when the mean fitness of the population did not improve for X generations.
3. Stop when the fitness of the best chromosome did not improve for X generations.

Running the same experiment as above for different sets of parameters gave us the following insights about the three stop criteria.
The first stop criteria is highly susceptible to the size of the population and the percentage of elitism. Therefor it is a rather difficult criterion to tune. When the population has little variety it is a good time to stop, but even when the whole generation is equal to the best chromosome, we still can't be sure if we are stuck in an local optimum, Let alone know how long we will be stuck in in this place. It could be that we reached a good optimum, but it could also mean that we just had little luck of finding a better optimum.
Another disadvantage is that often when there is a peak in the percentage of chromosomes that is equally to the best, the fittest solution has not been improved for a long time. Nevertheless when the whole population is fit, there might even be a bigger chance on finding an optimisation. Those remarks lead to the conclusion that this stop criteria is good for the tsp when you are looking for a really good solution and plan on running the GA for many generations.

The second stop criterion is also highly susceptible to the size of the population and the percentage of elitism. When the population is high and elitism is low, the mean fitness of the population will converge slowly and smoothly to the fitness of the best chromosome, while a low population and high elitism will converge fast with a lot of fluctuations. Furthermore a GA is a stochastic processes. X shall therefore have a certain minimum value to prevent the GA of stopping early because of bad luck. The main disadvantage of this criterion is that often when the mean fitness doesn't improve for a large amount of generations, the best solution has not been improved for a even longer time. A lot of useless generations are generated. Those can be omitted with the third criterion.

The tsp does only care for one good solution, it is therefore rather obvious to only consider the fittest solution of each generation. Useless generations are omitted but we do not consider information about the whole generation. Which makes this criterion perfect for large problems where a low cpu time is key.

We use the third criterion when we want to stop early. We exploratively search for a good solution and then stop the algorithm. A combination of the second and third stopping criteria is used when we look for a good solution but do want to omit useless generations. In this combination the GA stops when both the mean as the best solution have not improved for X generations. The first stopping criteria is used when we prefer quality over cpu time.

We visualized the three stop criteria in figure 9. The criteria used the following values:

1. Stop when 50 percent of the population has the same fitness as the best chromosome.
2. Stop when the mean fitness of the population did not improve for 100 generations.
3. Stop when the fitness of the best chromosome did not improve for 10 generations.

2.2. Efficiency of a GA

Every generation can contribute to a better solution. The question is: "Which generations are worth the cpu time?". When we look at the graph of the mean and best fitness values in figure 7, it is unambiguous where the perfect stopping point is located. Therefore we introduce four units to measurement to quantify these contributions:

1. The relative efficiency with no time penalty:
   $E = \frac{f_n - f_{n-1}}{f_{n-1}}$
2. The relative efficiency with a logarithmic time penalty:
   $E_{ln(n)} = \frac{f_n - f_{n-1}}{f_{n-1}} / ln(n)$
3. The relative efficiency with a linear time penalty:
   $E_n = \frac{f_n - f_{n-1}}{f_{n-1}} / n$
4. The relative efficiency with a exponential time penalty:
   $E_{exp(n)} = \frac{f_n - f_{n-1}}{f_{n-1}} / exp(n)$

Figure reffig:ex2map3 visualizes these four efficiency functions for the first 100 generations.

The efficiency function E, that doesn't penalize time, is considered useless for the TSP since a GA wants to approximate a good route and is not interested in the best solution. It is only makes sense to penalize time. The degree of penalization determines the way the efficiency evaluation can be used. A logarithmic penalization allows the GA to increase it generations as long as it is able to leave local optima and improve it's solution. Exponential penalization does not allow these extra generations and will cut the algorithm to an end when a good local optimum is reached.

Both the efficiency evaluation function for the mean and the best solutions can be used. efficiency evaluation function for the best chromosome of each generation can be used for determining the best generation to stop the GA at, while the mean can be used for an online approximation. The mean is especially good when the population size is large.

One could create an optimisation problem to fine tune the parameters of the stop criteria. Collect the data that is generated by multiple runs of the GA on multiple test data. Use the efficiency functions the determine the optimal stop point and change the parameters so that the optimal stopping point is approximated.

2.3. Summary

The best fitness value holds little information for a stop criterion. It is best used when in little time a loose approximation has to be found. Properties that take the whole generation into account are better for a long time search of a very good approximation of the best solution.

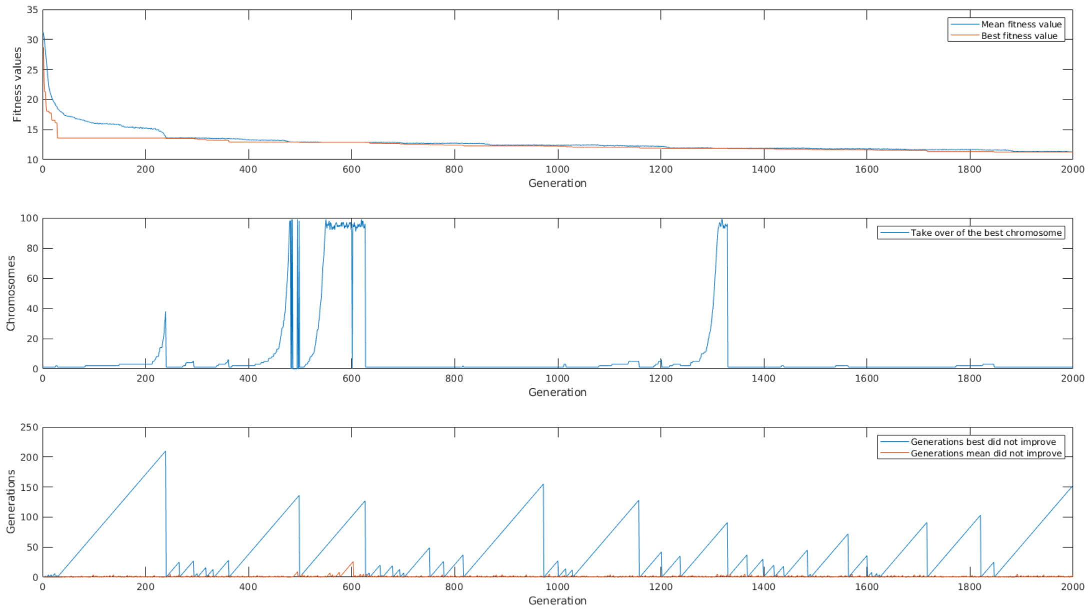

Fig. 7: [100, 0.9, 0.95, 0.5]-The properties of the stop criteria.

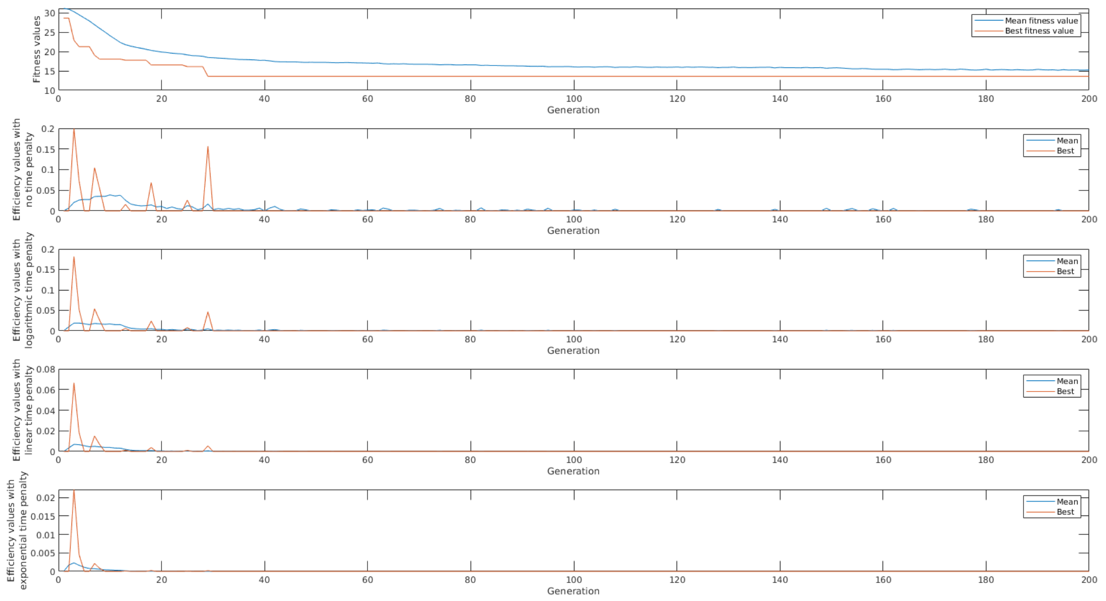

Fig. 8: [100, 0.9, 0.95, 0.5]-The efficiency values for the first 200 generations.

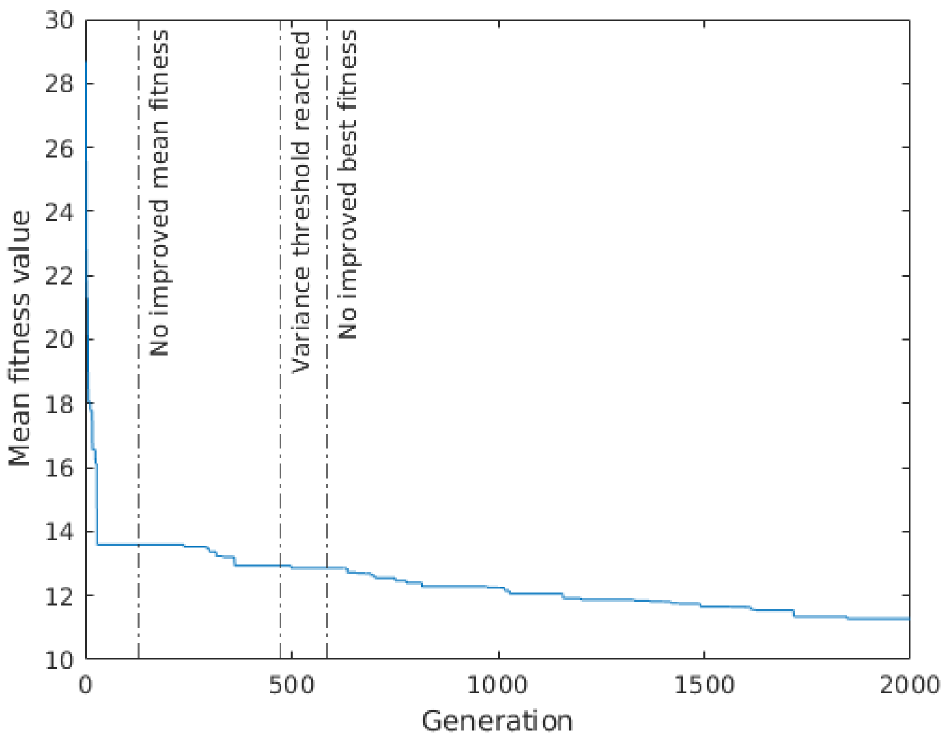

Fig. 9: [100, 0.9, 0.95, 0.5]-Stop criteria visualized on the best fitness and mean fitness value graph.

3. Other representation and appropriate operators (main task)

In this section another representation is implemented and a appropriate crossover and mutation operator is purposed. Some parameter tuning is performed to identify the proper combinations for an optimal result.

3.1. Representation

We adapt the algorithm to use path representation. Figure 10 shows a possible path found by the tsp algorithm for 5 cities. This solution in is represented in path representation as [1,2,3,4,5]. Each individual consists of an array where the elements are the cities visited in order.

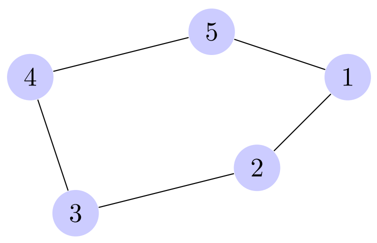

Fig. 10: TSP path example

3.2. Crossover operator

We decided to use the ordered crossover (OX), described in [1], as the new crossover operator. The code is shown in listing 2. Ordered crossover takes a substring from one parent and keeps the relative order of the other parent, this ensures that no drastic changes are made in the order of visited cities ensuring explotation over exploration. This is important because we do not want to completely scramble the order of cities visited, we want to keep the good subtours.

3.3. Mutation operator

We introduce reverse sequence mutation (RSM) as an alternative mutation operator. In RSM a sequence S, limited by positions i and j, is randomly chosen from the genotype of an individual. This order will be reversed and placed back again in the individual. The code is shown in listing 1.

3.4. Parameter tuning and testing

Figure 13 and table 2 show the influence of different values of elitism on the result. A good value for elitism is in the range [0.1-0.5]. We decide to choose 0.3 as elitism value because of the acceptable complexity.

Figure 14 shows different population sizes. Small populations will have no sufficient diversity and will encounter premature convergence, while a too large population size does not improve the fitness significantly and therefore generates a lot of overhead. We consider 100 as a good population size because there is not much increase in fitness for larger populations.

The same method of finding the optimal parameter settings are used as in section 1. Starting of by finding a good combination of crossover rate and mutation rate. Figure 12 shows the best fitness values for different combinations of crossover and mutation rate using a population of 100 and a elitism of 0.4. As shown in the figure, a medium crossover rate and mutation rate gives the best results. It is even more the obvious than in the default method. The resulting values are a crossover rate of 0.6 and a mutation rate of 0.5.

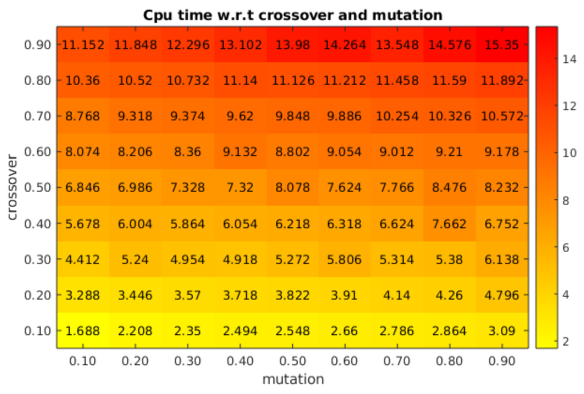

Fig. 11: Correlation of crossover and mutation on average cpu time

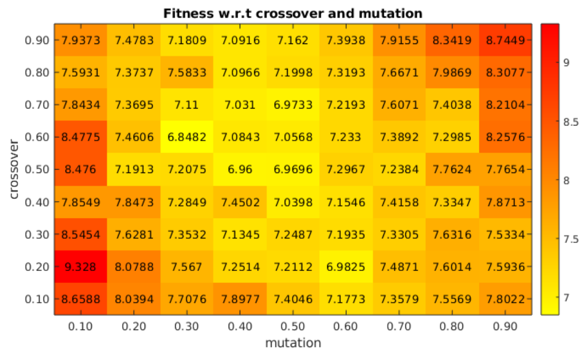

Fig. 12: Correlation of crossover and mutation on best fitness

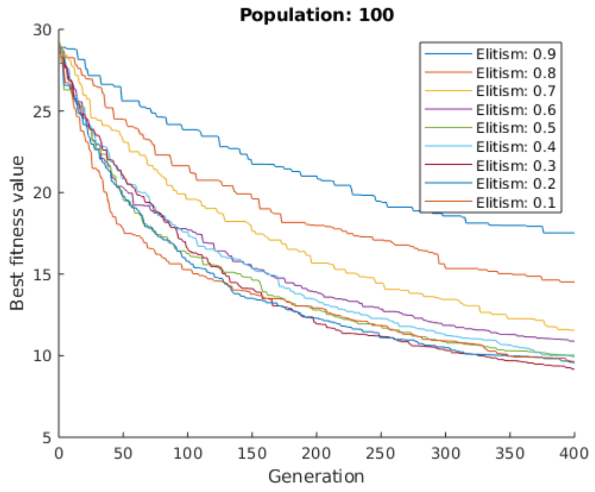

Fig. 13: [100,var,0.9,0.1]-Best fitness for different elitism values

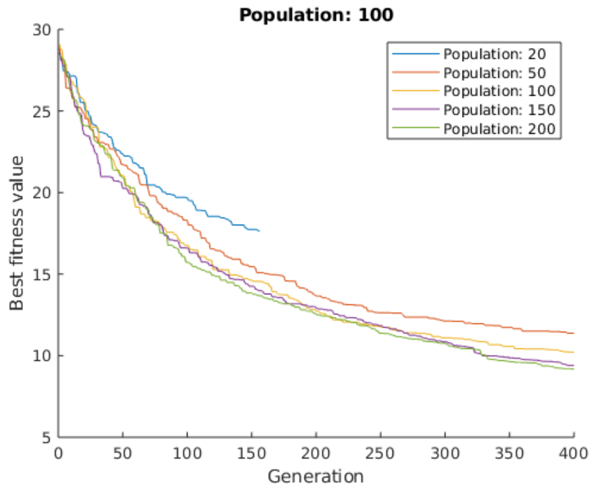

Fig. 14: [var,0.4,0.9,0.1]-Best fitness for different population sizes

		Elitism								
		0.90	0.80	0.70	0.60	0.50	0.40	0.30	0.20	0.10
400	Best Fitness	17.524	14.519	11.555	10.8922	10.0242	9.9248	9.1856	9.575	9.598
generations	Cpu time	1.446	1.956	2.07330	3.6730	4.5380	5.768	6.651	7.674	8.32

Table 2: Result of elitism parameter on best fitness for 400 generations.

4. Local optimisation

In this section a local search heuristic genetic algorithm is implemented. The algorithm described in [2] is used as a guidance. The algorithm is implemented and tested. The section ends with a discussion of the results.

4.1. Local Search Heuristic Genetic Algorithm

The algorithm described in [2] works as follows:

1. A random population with a path representation is generated.
2. A local search in the neighbourhood is used as an asexual crossover operator. (see listing 7)
3. Mutation is done by the a MUT3 operator. (see listing 8)
4. 2k-tournament selection is used to select the fittest individuals for survival. (see listing 6)
5. Crossover rate, mutation rate and elitism are self adaptive.

The whole algorithm is shown in listing 9.

The results of the algorithm are shown in figure 15, figure 16, figure 17 and figure 18.
The graphs can be explained as followed:

- At initialization the population is filled with random individuals. The crossover rate is high, therefore the population will slowly be optimized. The fitness of the best and the mean fitness of the population will slowly decrease.
- When a path reaches it's local optimum, new paths will be introduced through mutation. The mean fitness may rise because those new path have a high chance to have a worse solution than the mean.
- When those new solution are optimized by crossover operation, there is a chance that a new best is found.

The greatest disadvantage of the algorithm is that crossover is only asexual. Information gained by one individual of the population is never shared with the other individuals. If a local optimum is reached through crossover, the only way to improve it is to mutate and hope that optimization by crossover yields a better result.
Since 2k-tournament selection is used, the chance that the best solution leaves the breading pool is real. This is shown in figure 15.

We use no improved best as the stop criterion since the other two that where purposed are rather useless for this algorithm. The means always decreases except for the occasional peak that happens in just one generation and the population will almost never consist of equally individuals since asexual reproduction is used.

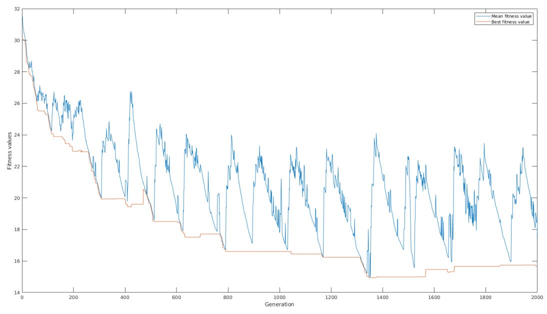

Fig. 15: 2000 generations for a population size of 10.

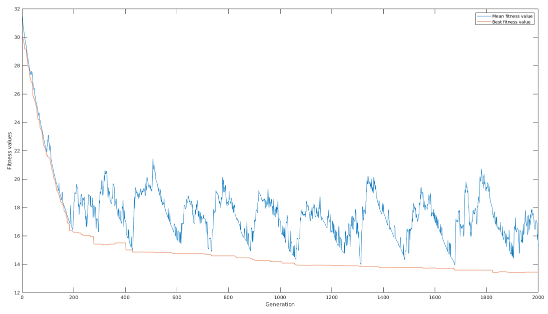

Fig. 16: 2000 generations for a population size of 20.

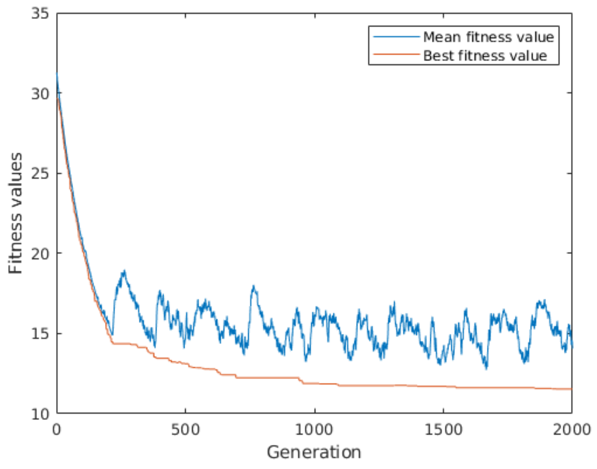

Fig. 17: 2000 generations for a population size of 50.

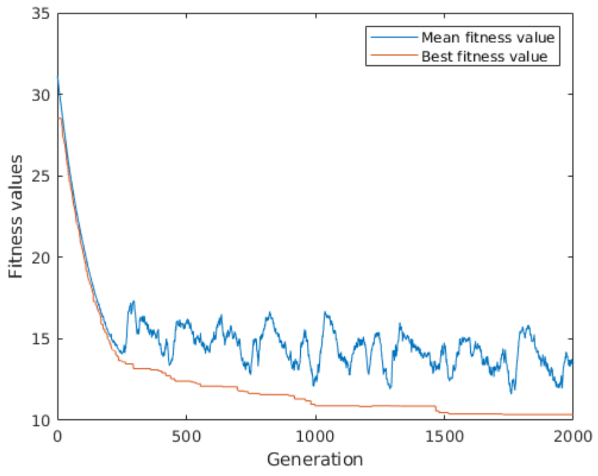

Fig. 18: 2000 generations for a population size of 100.

4.2. discussion results

We ran the algorithm 10 times on the "rondrit127" dataset with varying population sizes. As a stop criterion we used a non improved best for 5 generations. The algorithm is run for a population of [5, 10, 20, 50, 100] and the results are shown in figures 19, 20, 21, 22 and 23. The best run for each population size is combined in figure 24. The result of those runs are also shown in table 3.

Table 3 shows that the time does not grow linear with the population size. We therefore made some changes to their algorithm. We set a fixed mutation rate of 1. This means that when no better solution is found after crossover, the individual will always mutate. A lot of time is saved and it only seems logical to mutate when there are no more improvements to be made.

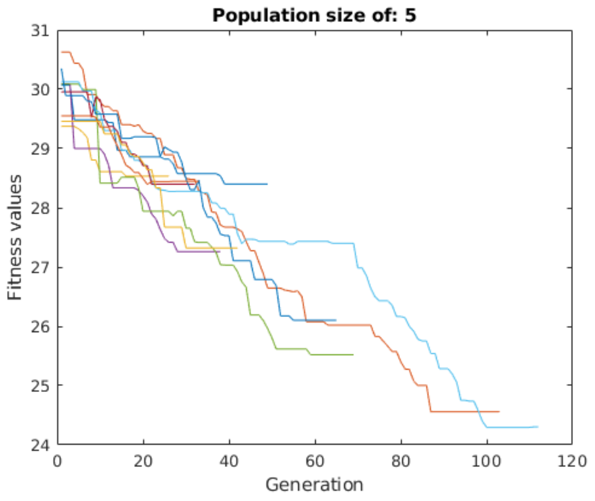

Fig. 19: 10 runs for a population size of 5.

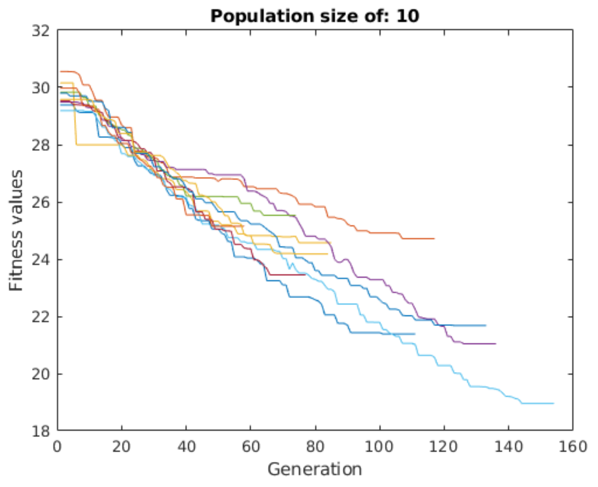

Fig. 20: 10 runs for a population size of 10.

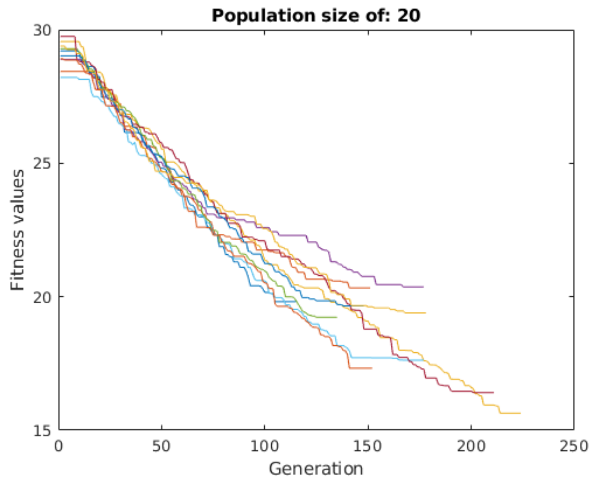

Fig. 21: 10 runs for a population size of 20.

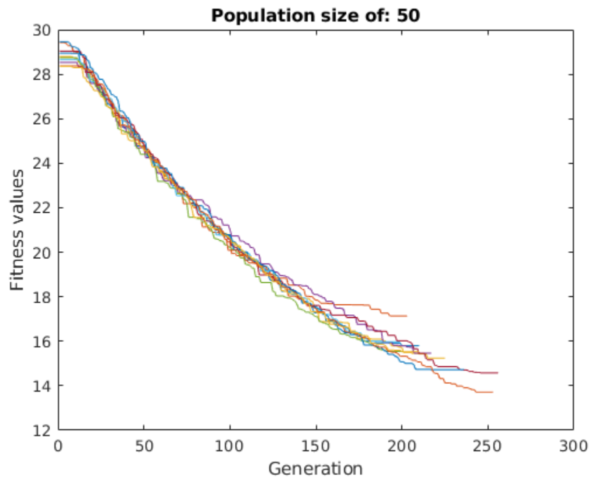

Fig. 22: 10 runs for a population size of 50.

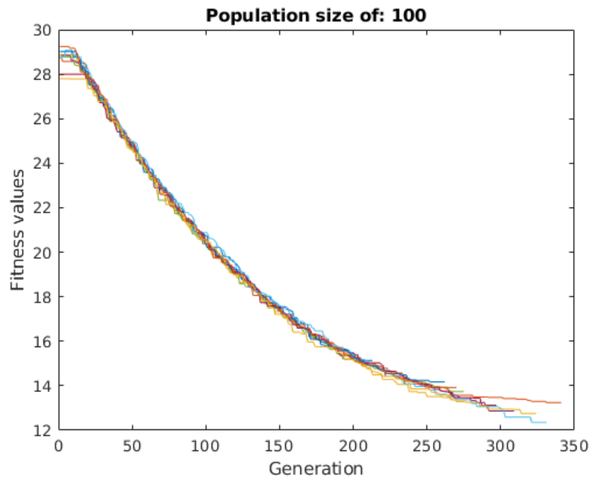

Fig. 23: 10 runs for a population size of 100.

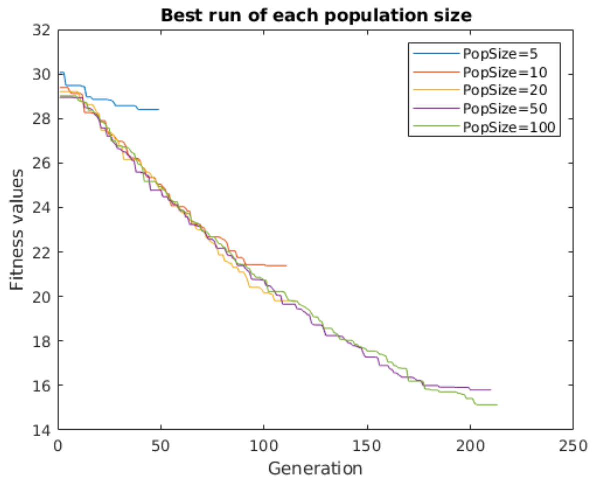

Fig. 24: The best run for each population size.

Population size	5	10	20	50	100
Time [s]	0.01309	0.037998	0.084409	1.30991	10.895986

Table 3: The time needed for a population size of [5,10,20,50,100].

5. Benchmark problems

While testing out different parameter values, we found out that we had a bad performance on the datasets with a large number of cities, this was due to our small population size and low amount of generations considered. To improve the resulting tour length we either enlarged the number of generations or population size and searched for a good trade-off between both. The results show that a larger population is more beneficial than a large number of generations but takes a much longer time, figure 25,26,27,28 and 29. Therefore we decided to test our benchmarks with more generations and a low population size.

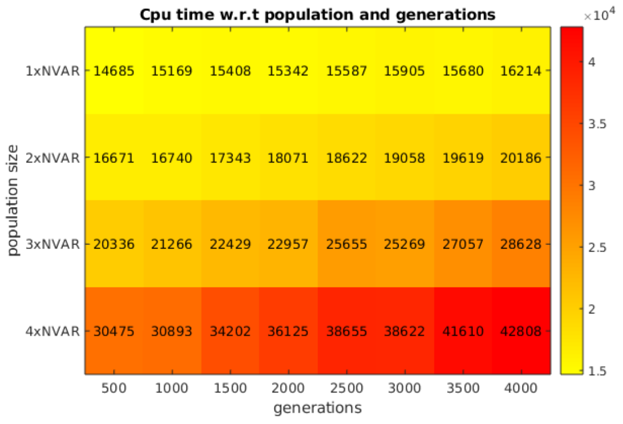

Fig. 25: Cpu time of benchmark rbx711 on variable generations and population sizes

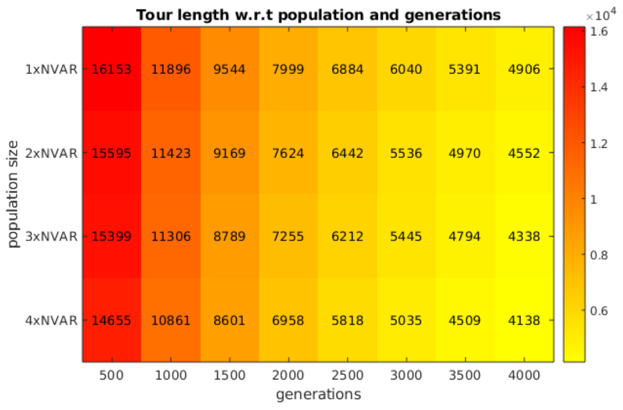

Fig. 26: Tour length of benchmark rbx711 on variable generations and population sizes

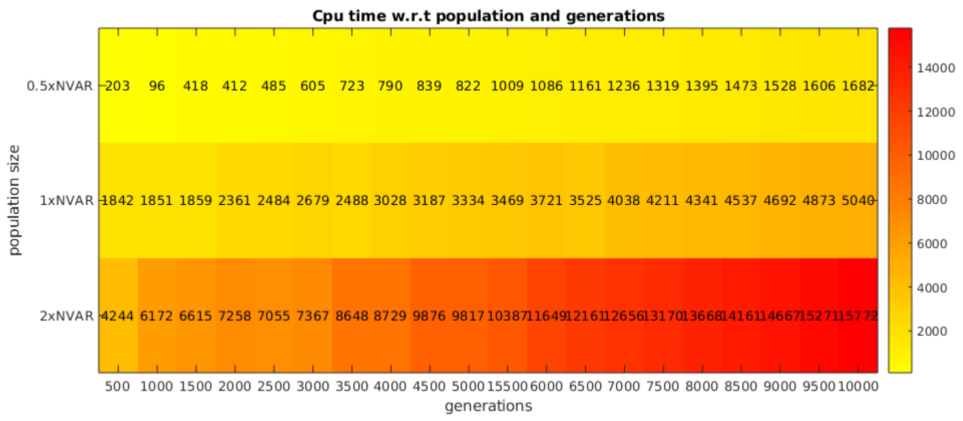

Fig. 27: Cpu time of benchmark rbx711 on variable generations and population sizes

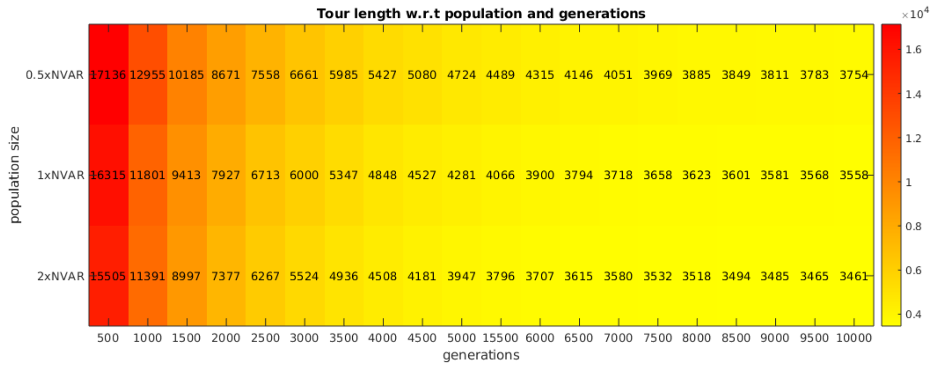

Fig. 28: Tour length of benchmark rbx711 on variable generations and population sizes

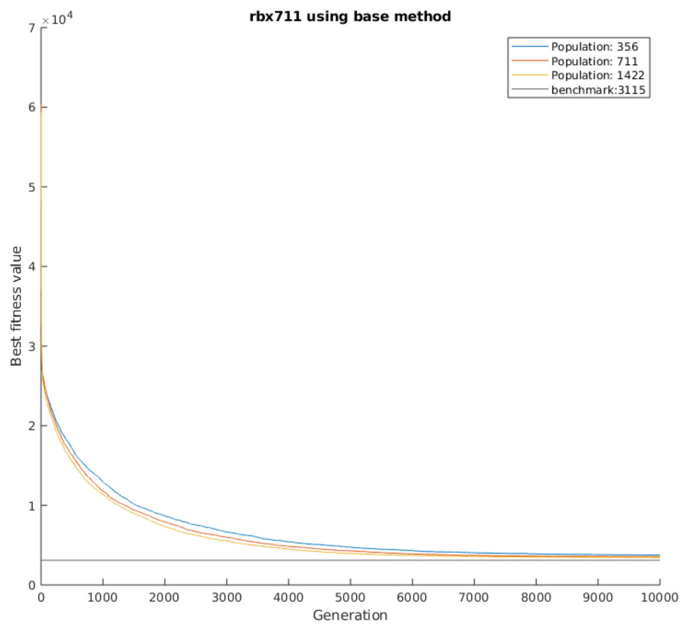

Fig. 29: Tour length of benchmark rbx711 on variable population sizes

		Method									
		Base		OX/RSM/pathrep		Tournament		SRS		LSHGA	
Dataset	Optimal tour	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]
xqf131	564	610	57	622	92	613	43	617	52	1409	251
bcl1380	1621	1837	378	1946	1083	2049	264	2107	362	7745	1649
xql662	2513	41000	820	3240	3244	4240	464	4512	396	30373	8985
rbx711	3115	5291	990	4001	3901	5329	548	5967	464	37993	10303

Paramaters	Nb of generations	10000	10000	10000	10000	10000
	Population size	#cities/2	#cities/2	#cities/2	#cities/2	#cities/2
	Elitism	0.1	0.3	0.3	0.4	NaN
	Crossover rate	0.3	0.6	0.2	0.2	NaN
	Mutation rate	0.5	0.5	0.	0.6	NaN
	Stop percentage	1	1	1	1	NaN
	Loop detection	0	0	0	0	NaN
	Elitism constant	NaN	NaN	NaN	NaN	0.1
	Crossover constant	NaN	NaN	NaN	NaN	1

Table 4: The five algorithms tested on the benchmarks with the previously discovered optimal parameters and a population size that is equal to half of the amount of cities.

		Method									
		Base		OX/RSM/pathrep		Tournament		SRS		LSHGA	
Dataset	Optimal tour	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]	Tour	Time [s]
xqf131	564	587	77	589	132	606	49	621	42	1483	513
bcl1380	1621	1901	549	1844	1534	1901	319	1918	271	7920	4591
xql662	2513	3659	1795	2975	6767	3824	1026	4044	872	30844	19124
rbx711	3115	4601	2109	3683	7927	4784	1196	5354	1000	38493	21687

Paramaters	Nb of generations	10000	10000	10000	10000	10000
	Population size	#cities	#cities	#cities	#cities	#cities
	Elitism	0.1	0.3	0.3	0.4	NaN
	Crossover rate	0.3	0.6	0.2	0.2	NaN
	Mutation rate	0.5	0.5	0.	0.6	NaN
	Stop percentage	1	1	1	1	NaN
	Loop detection	0	0	0	0	NaN
	Elitism constant	NaN	NaN	NaN	NaN	0.1
	Crossover constant	NaN	NaN	NaN	NaN	1

Table 5: The five algorithms tested on the benchmarks with the previously discovered optimal parameters and a population size that is equal to the amount of cities.

6. Other task(s)

For the extra task we decided to implement 2 new parent selection methods: tournament selection with/without replacement (described as in the book) and stochastic remainder selection.
Stochastic remainder selection works as following: First every fitness value is scaled using :

$$f_{i,new} = \frac{f_i \times n_{pop}}{\sum_{j=1}^{n_{pop}} f_i}$$

All fitness that had a fitness value above the average fitness will have a scaled fitness greater than 1. Every individual is then copied into the mating pool a number of times equal to the integer past of the scaled fitness. Next we take the residual fitness as the decimal parts of the scaled fitness. We then perform roulette wheel selection based on the residual fitness to fill the remaining places in the mating pool. Stochastic remainder selection combines a deterministic step that ensures that individuals with a good fitness value are used as parents for the new population combined with a stochastic step that offers all individuals a chance to become a parent.
The implementation of the selection methods are shown in Algoritme 3, 4 and 5.

After testing the different methods we found that for both tournament selection methods a population of 100 gives the best results, for stochastic remainder selection a population of 50 gives the best results, the reason a lower population is sufficient lies in the algorithm using deterministic and stochastic elements.

The best parameters of elitism are: 0.3 for both tournament selections and 0.4 for the stochastic remainder selection.

During our testing of good parameters for crossover and mutation we found out that we want a lower crossover rate while having an average mutation rate for all three different selection methods.

In figure 38 we can see that the different selection methods have a similar result We can see that the stochastic remainder selection performs slightly better than stochastic remainder selection, and stochastic remainder selection performs slightly better than tournament selection.

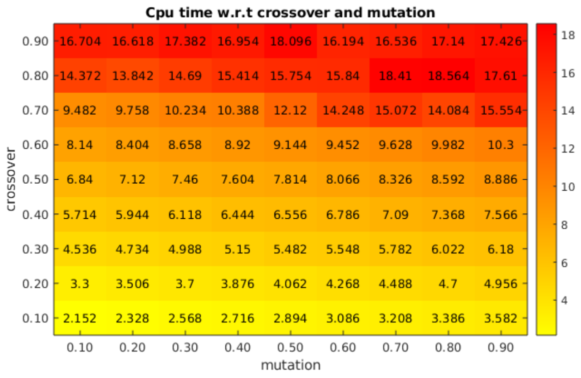

Fig. 30: Correlation of crossover and mutation on average cpu time

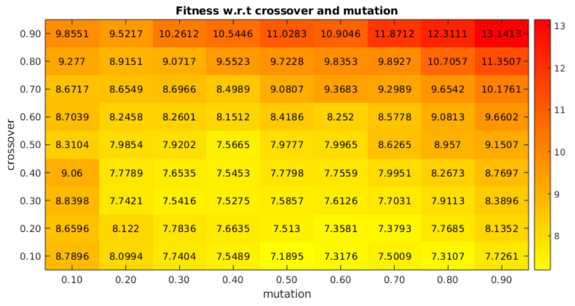

Fig. 31: Correlation of crossover and mutation on best fitness

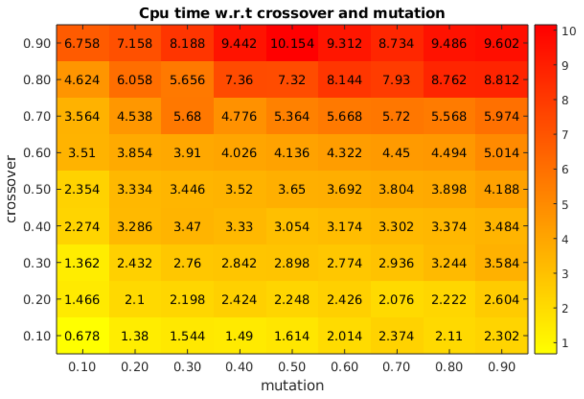

Fig. 32: Correlation of crossover and mutation on average cpu time

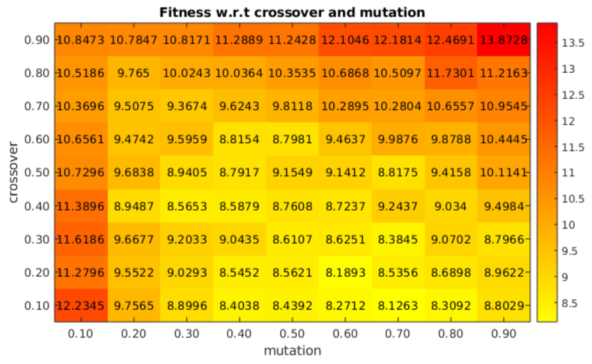

Fig. 33: Correlation of crossover and mutation on best fitness

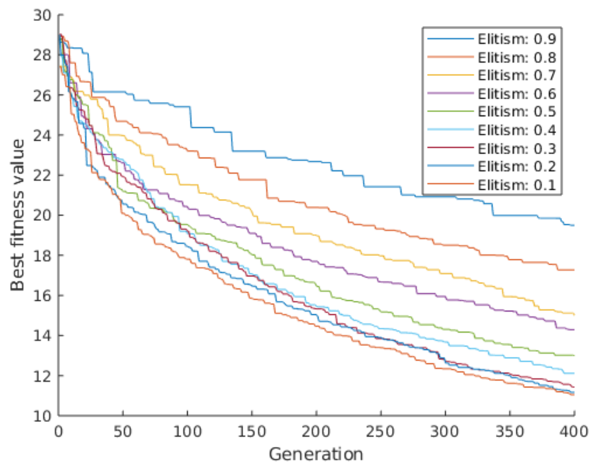

Fig. 34: Influence of elitism on tournament selection without replacement

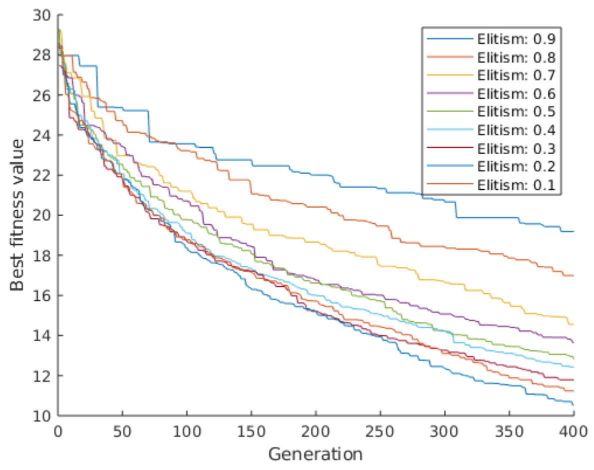

Fig. 35: Influence of elitism on tournament selection with replacement

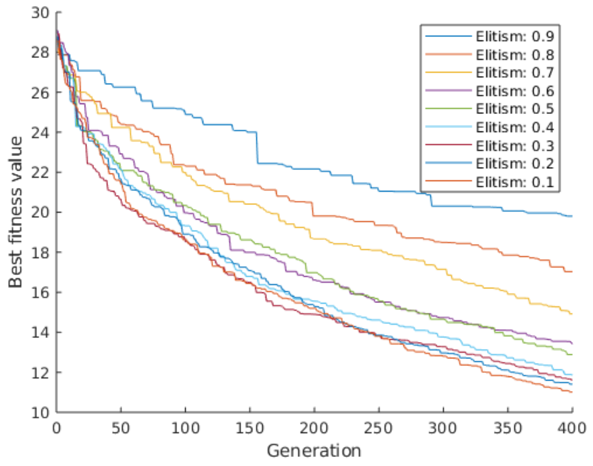

Fig. 36: Influence of elitism on stochastic remainder selection

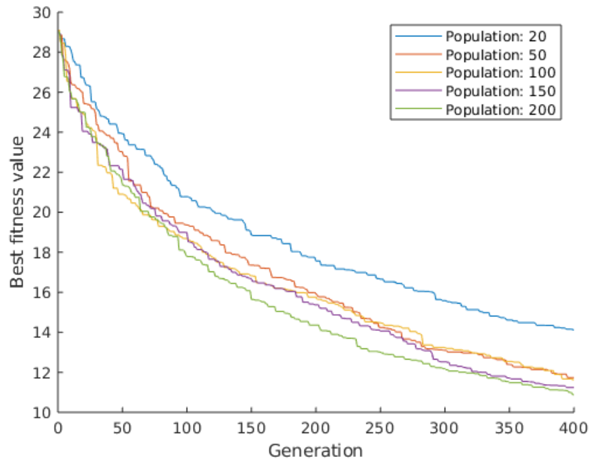

Fig. 37: Influence of population size on stochastic remainder selection

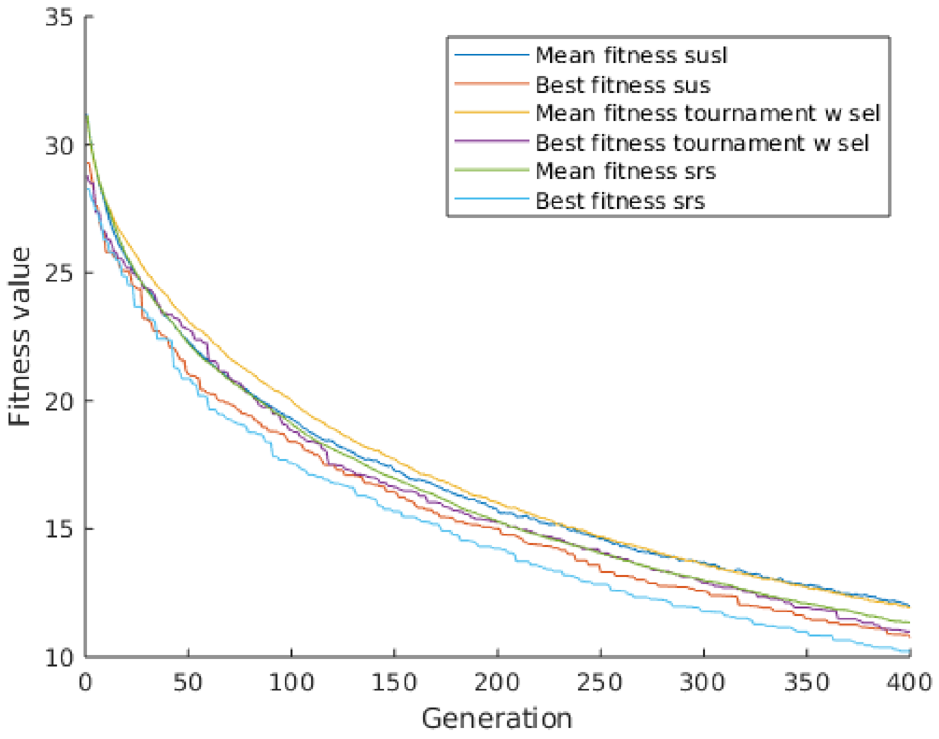

Fig. 38: Fitness using different selection methods

			Elitism								
			0.90	0.80	0.70	0.60	0.50	0.40	0.30	0.20	0.10
tournament wh replacement	400 generations	Best Fitness	19.4994	17.277	14.999	14.294	12.982	12.114	11.422	11.169	11.048
		Cpu time	1.059	1.352	1.728	2.321	2.80	3.323	3.705	4.741	4.801
tournament with replacement	400 generations	Best Fitness	19.190	16.988	14.558	13.649	12.821	12.431	11.789	10.535	11.2393
		Cpu time	0.812	1.317	1.828	2.369	2.747	3.2530	3.617	4.102	4.650
stochastic remainder selection	400 generations	Best Fitness	19.8031	17.030	14.920	13.406	12.889	11.853	11.627	11.404	11.010
		Cpu time	0.766	1.354	1.705	2.299	2.779	3.667	4.247	4.7640	5.1320
			Population								
			20	50	100	150	200				
	400 generations	Best Fitness	14.116	11.375	11.6327	11.2357	10.883				
		Cpu time	0.889	1.912	3.832	5.619	7.4760				

Table 6: Influence of elitism and population on different selection methods

		Base	New operators	Ex4	Tournament	Stochastic remainder
results	Best fitness	10.565	9.598	13.4387	11.048	10.883
	Cpu time/training cycle	3.172	8.32	36.8750	4.801	4.650
parameter settings	Elitism	0.1	0.3	NaN	0.3	0.4
	Crossover rate	0.3	0.6	NaN	0.2	0.2
	Mutation rate	0.5	0.5	NaN	0.6	0.6
	Population size	100	100	100	100	50

Table 7: Best fitness and respective cpu time for all considered methods in this paper and their best parameters for 400 generations on the 127 cities dataset.

Time spent on the project

1. For each student of the team: estimate how many hours spent on the project (NOT including studying textbook and other reading material).
   (a) Verheyen Willem: 80 hours
   (b) Vandebeek Sjaan: 60 hours
2. Briefly discuss how the work was distributed among the team members.

   Willem Verheyen handled implementation, testing and reporting exercise 1,3 and 6. Sjaan Vandebeeck did implementation, testing and reporting of exercise 2 and 4. Exercise 5 was made by both students.

References

[1] A. Eiben and Jim Smith. *Introduction To Evolutionary Computing*, volume 45. 01 2003.

[2] J. Zhang and C. Tong. Solving tsp with novel local search heuristic genetic algorithms. In *2008 Fourth International Conference on Natural Computation*, volume 1, pages 670–674, Oct 2008.

7. Source code

Listing 1: RSM Mutation

```
function NewChrom = mutate_RSM(OldChrom, Representation)
    %implements the reverse sequence mutation operator
    if Representation==1
        OldChrom=adj2path(OldChrom);
    end

    [¬,Chromsize] = size(OldChrom);
    i = randi([1 Chromsize]);
    j = randi([i Chromsize]);
    NewChrom = zeros(1,Chromsize);
    for x = 1:Chromsize
        if ( x ≥ i && x ≤ j)
            NewChrom(1,x) = OldChrom(j+(i−x));
        else
            NewChrom(1,x) = OldChrom(x);
        end
    end
    if Representation==1
        NewChrom=path2adj(NewChrom);
    end
end
```

Listing 2: OX crossover

```
function NewChrom = OX(OldChrom)
%EX3_ORDERED_CROSSOVER implements ordered based crossover for the path
%presentation in the tsp.

    [Parents,GenSize]=size(OldChrom);
    NewChrom = zeros(Parents,GenSize); % Allocate memory for the offspring

    RandPerms = randperm(GenSize);
    CrossP1 = RandPerms(1,1); %Random crossover point 1
    CrossP2 = RandPerms(1,2); %Random crossover point 2

    if CrossP1 > CrossP2
        Temp = CrossP1;
        CrossP1 = CrossP2;
        CrossP2 = Temp;
    end

    MidPart1 = OldChrom(1,CrossP1:CrossP2);
    MidPart2 = OldChrom(2,CrossP1:CrossP2);

    %Set the inner parts
    NewChrom(1,CrossP1:CrossP2) = MidPart1;
    NewChrom(2,CrossP1:CrossP2) = MidPart2;

    tmp = [1:1:GenSize];

    restOf1 = [OldChrom(1,CrossP2+1 : GenSize)  OldChrom(1,1 : CrossP2)];
    restOf2 = [OldChrom(2,CrossP2+1 : GenSize)  OldChrom(2,1 : CrossP2)];

    %Set the rest
    for index = CrossP2+1:GenSize;
        if NewChrom(1,index) == 0
            for ind = 1 : length(restOf2)
                if (¬ismember(NewChrom(1,:),restOf2(ind)))
```

```
                    NewChrom(1,index) = restOf2(ind);
                    restOf2 = restOf2(ind:end);
                    break;
                end
            end
        end
    end
    for index = CrossP2+1:GenSize;
        if NewChrom(2,index) == 0
            for ind = 1 : length(restOf1)
                if (¬ismember(NewChrom(2,:),restOf1(ind)))
                    NewChrom(2,index) = restOf1(ind);
                    restOf1 = restOf1(ind:end);
                    break;
                end
            end
        end
    end

    for index = 1:CrossP1;
        if NewChrom(1,index) == 0
            for ind = 1 : length(restOf2)
                if (¬ismember(NewChrom(1,:),restOf2(ind)))
                    NewChrom(1,index) = restOf2(ind);
                    restOf2 = restOf2(ind:end);
                    break;
                end
            end
        end
    end
    for index = 1:CrossP1;
        if NewChrom(2,index) == 0
            for ind = 1 : length(restOf1)
                if (¬ismember(NewChrom(2,:),restOf1(ind)))
                    NewChrom(2,index) = restOf1(ind);
                    restOf1 = restOf1(ind:end);
                    break;
                end
            end
        end
    end
end
```

Listing 3: Tournament selection with replacement

```
% This function performs selection tournament with replacement.
%
% Syntax:  NewChrIx = sus(FitnV, Nsel)
%
% Input parameters:
%    FitnV     — Column vector containing the fitness values of the
%                individuals in the population.
%    Nsel      — number of individuals to be selected
%
% Output parameters:
%    NewChrIx  — column vector containing the indexes of the selected
%                individuals relative to the original population, shuffled.
%                The new population, ready for mating, can be obtained
%                by calculating OldChrom(NewChrIx,:).

function NewChrIx = tournament(FitnV,Nsel);
% tournament size
```

```
    k = 2;
% Identify the population size (Nind)
   [Nind,anss] = size(FitnV);
    NewChrIx = zeros(Nind,anss);
    listInd = [1:1:Nind];
% Perform tournament selection
    for i = 1 : Nsel
        TempFit = zeros(1,k);
        Indexes = zeros(1,k);
        Randx = listInd(randperm(length(listInd)));
        Rand = Randx(1:k);
        for c=1:k;
            TempFit(c) = FitnV(Rand(1,c));
            Indexes(c) = Rand(1,c);
        end
        Ind = find(max(TempFit) == TempFit);
        NewChrIx(i,:) = Indexes(Ind(1,1));
        listInd(listInd == Indexes(Ind(1,1))) = [];
        Nind = Nind-1;
        if (Nind < k) k = k - 1; end
    end

% Shuffle new population
   [ans, shuf] = sort(rand(Nsel, 1));
   NewChrIx = NewChrIx(shuf);


% End of function
```

Listing 4: Tournament selection without replacement

```
% This function performs selection tournament without replacement.
%
% Syntax:  NewChrIx = sus(FitnV, Nsel)
%
% Input parameters:
%    FitnV     - Column vector containing the fitness values of the
%                individuals in the population.
%    Nsel      - number of individuals to be selected
%
% Output parameters:
%    NewChrIx  - column vector containing the indexes of the selected
%                individuals relative to the original population, shuffled.
%                The new population, ready for mating, can be obtained
%                by calculating OldChrom(NewChrIx,:).


function NewChrIx = tournament2(FitnV,Nsel);
% tournament size
    k = 2;
% Identify the population size (Nind)
   [Nind,anss] = size(FitnV);
    NewChrIx = zeros(Nind,anss);
    listInd = [1:1:Nind];
% Perform tournament selection
    for i = 1 : Nsel
        TempFit = zeros(1,k);
        Indexes = zeros(1,k);
        Randx = listInd(randperm(length(listInd)));
        Rand = Randx(1:k);
        for c=1:k;
            TempFit(c) = FitnV(Rand(1,c));
            Indexes(c) = Rand(1,c);
        end
        Ind = find(max(TempFit) == TempFit);
        NewChrIx(i,:) = Indexes(Ind(1,1));
    end

% Shuffle new population
   [ans, shuf] = sort(rand(Nsel, 1));
   NewChrIx = NewChrIx(shuf);


% End of function
```

Listing 5: Stochastic remainder selection

```
% This function performs stochastic remainder selection.
%
% Syntax:  NewChrIx = sus(FitnV, Nsel)
%
% Input parameters:
%    FitnV     — Column vector containing the fitness values of the
%                individuals in the population.
%    Nsel      — number of individuals to be selected
%
% Output parameters:
%    NewChrIx  — column vector containing the indexes of the selected
%                individuals relative to the original population, shuffled.
%                The new population, ready for mating, can be obtained
%                by calculating OldChrom(NewChrIx,:).

function NewChrIx = srs(FitnV,Nsel);
%FitnV
%Nsel
% Identify the population size (Nind)
   [Nind,anss] = size(FitnV);
    NewChrIx = zeros(Nind,anss);
    expcounts = zeros(Nind,1);
    mantexp = zeros(Nind,1);

% Perform stochastic remainder selection
    for i = 1 : Nind
        mantexp(i,1) = FitnV(i,1)*Nind/sum(FitnV);
    end
    [mantexp,order] = sort(mantexp);
    cnt = 1;
    for i = 1: Nind
        while (mantexp(i,1) ≥ 1 && cnt ≤ Nsel)
            NewChrIx(cnt,1) = order(i);
            cnt = cnt + 1;
            mantexp(i,1) = mantexp(i,1) — 1;
        end
    end
    mantisses = zeros(Nind,1);
    for i = 1 : Nind
        mantisses(i,1) = mantexp(i,1) — floor(mantexp(i,1));
    end
    r = randi(100);
    while (cnt ≤ Nsel)
        i = 1;
        while(mantisses(i,1) < r && i < Nind)
            i = i + 1;
        end
        NewChrIx(cnt,1) = order(i);
        cnt = cnt+1;
    end

% Shuffle new population
   [ans, shuf] = sort(rand(Nsel, 1));
   NewChrIx = NewChrIx(shuf);


% End of function
```

Listing 6: 2k-tournament selection

```
unction [SelectedPopulation,SelectedObjV] = binary_tournament_selection_LSHGA(Population,ObjV,ATS)
% BINARY_TOURNAMENT_SELECTION select ATS individuals in a binary
% (k=2) tournament selection.
%   Population: the population to cut in half with BTS.
%   SelectedPopulation: the population selected with BTS.
%   ATS: the amount of individuals to select.
% Example: binary_tournament_selection_LSHGA([1:20]',[1:20]',8)

    % Size of population and amount of cities.
    [PopSize,Cities] = size(Population);

    if PopSize == ATS
        SelectedPopulation = Population;
        SelectedObjV = ObjV;
        return;
    end

    % Allocate memory for SelectedPopulation and for SelectedObjV.
    SelectedPopulation = zeros(ATS,Cities);
    SelectedObjV = zeros(ATS,1);

    % Use 2k Tournament selection to reduce the population in half
    for i = 1:ATS
        sft = randi(PopSize,1,2);
        if ObjV(sft(1)) < ObjV(sft(2))
            SelectedPopulation(i,:) = Population(sft(1),:);
            SelectedObjV(i,1) = ObjV(sft(1),1);
            Population(sft(1),:) = [];
            ObjV(sft(1)) = [];
        else
            SelectedPopulation(i,:) = Population(sft(2),:);
            SelectedObjV(i,1) = ObjV(sft(2),1);
            Population(sft(2),:) = [];
            ObjV(sft(2)) = [];
        end
        PopSize = PopSize -1;
    end
end
```

Listing 7: LSH-crossover

```
function [Child,Distance] = crossover_LSHGA(Dist,Parent,Distance)
%CROSSOVER_LSHGA is the crossover operator that is described in
%[Zhang,Tong].
%   Parent: the path representation of the single parent selected for
%   crossover.
%   Child: the path representation of the single child that is produced by
%   applying the crossover operator.
%   Dist: the matrix that represent the distance between the cities.

    %% Step 0: preperations
    [¬,cities] = size(Parent);% Number of cities to visist

    %% STEP 1: choose a random city to visist first in the cycle
    % The random city to put first in the path.
    i = randi(cities);
    % Switch the order of the cities such that the i-th city appears first
    % in the path.
    ParentX = [Parent(1,i+1:cities),Parent(1,1:i)];
    % j is the last city in the path.

    for n = 1: cities
        %% STEP 2: Search the city that is the closest to the last city
        % The distance between the city that is the closest to j.
        shortest2j = Dist(2,cities);
        % The index of the city that is the closest to j
        s2ji = 2;
        % determine the closest city to j and the distance to j
        for i = 3:cities-1
            if Dist(i,cities) < shortest2j
                shortest2j = Dist(i,cities);
                s2ji = i;
            end
        end

        %% STEP3: Rearange the tour
        Child = [ParentX(1,1:s2ji-1), ParentX(1,s2ji+1:cities),ParentX(1,s2ji)];

        %% STEP4: Evaluation step
        DeltaDistance = ...
            Dist(ParentX(1),ParentX(s2ji))+Dist(ParentX(s2ji-1),ParentX(s2ji+1))+Dist(ParentX(s2ji),ParentX(citie
        if DeltaDistance < 0
            Distance = Distance + DeltaDistance;
            if Distance == 0
            end
            return;
        end

        %% STEP5: Repeat until child is found or cities are exhausted
        ParentX = [ParentX(1,cities),ParentX(1,1:cities-1)];

    end
    Child = Parent;
end
```

Listing 8: MUT3-mutation

```
function [Mutation,Distance] = mutation_LSHGA(Dist,Chromosome,Distance)
% Mutate the Chromosome with a MUT3 operator
%   Dist: the matrix that represent the distance between the cities.
%   Chromosome: the chromosome to mutate.
%   Distance: the distance of the tour of the given chromosome.
%   Mutation: the mutated chromosome.

    % The amount of cities in the tour
    [¬,cities] = size(Chromosome);
    % Two random points to use the MUT3 operator on
    indices = sort(randi(cities-1,1,2));

    % a = Chromosome(1,1:indices(1));
    % b = Chromosome(1,indices(1)+1:indices(2));
    % c = Chromosome(1,indices(2)+1:cities);
    % Mutation = [b,flip(c),a];
    Mutation = ...
        [Chromosome(1,indices(1)+1:indices(2)),flip(Chromosome(1,indices(2)+1:cities)),Chromosome(1,1:indices(1))

    % The difference in distance between the old chromosome and the
    % mutation
    ∆Distance = Dist(Chromosome(indices(2)),Chromosome(cities)) + ...
        Dist(Chromosome(indices(2)+1),Chromosome(1)) - ...
        Dist(Chromosome(indices(2)),Chromosome(indices(2)+1)) - ...
        Dist(Chromosome(cities),Chromosome(1));
    % The distance of the mutation
    Distance = Distance + ∆Distance;
end
```

Listing 9: LSHGA

```
function [mean_fits,minimum,best] = run_ga_with_local_heuristics(x, y, NIND, MAXGEN, NVAR, Kc, ...
    Km, Ke)
% usage: run_ga(x, y,
%               NIND, MAXGEN, NVAR,
%               ELITIST, STOP_PERCENTAGE,
%               PR_CROSS, PR_MUT, CROSSOVER,
%               ah1, ah2, ah3)
%
% x, y: coordinates of the cities
% NIND: number of individuals
% NVAR: number of cities
% MAXGEN: maximal number of generations
% ELITIST: percentage of elite population
% fitness (stop criterium)
% PR_CROSS: probability for crossover
% PR_MUT: probability for mutation
% CROSSOVER: the crossover operator
% calculate distance matrix between each pair of cities

    % Generation
    gen=1;

    % Best fitness value of all generations.
    best=zeros(1,MAXGEN);
    % Mean fitness values of all generations.
    mean_fits = zeros(1,MAXGEN);
    % Worst fitness value of all generations.
    worst = zeros(1,MAXGEN);

    % Distance between cities
    Dist = zeros(NVAR,NVAR);
    for i=1:size(x,1)
        for j=1:size(y,1)
            Dist(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
        end
    end

    % Initialize population
    Chrom = zeros(NIND,NVAR);
    for row=1:NIND
        Chrom(row,:)=randperm(NVAR);            % Path representation is used
        %Chrom(row,:)=path2adj(randperm(NVAR));  % Adjacency representation
    end
    % Evaluate initial population
    ObjV = tspfun2(Chrom,Dist);

    % Generational loop
    while gen<MAXGEN+1

        % Generation statistics
        best(gen)=min(ObjV); % The best chromosome
        mean_fits(gen)=mean(ObjV); % Mean fitness value
        worst(gen)=max(ObjV); % Worst fitness value
        fmax = fitness_LSHGA(best(gen),NVAR);
        fmean = fitness_LSHGA(mean_fits(gen),NVAR);

        % Louche shit in hun code
        minimum=best(gen);
        for t=1:size(ObjV,1)
            if (ObjV(t)==minimum)
                break;
            end
        end

        % Create the selection pool
        Parents = zeros(NIND,NVAR);
        ParentsObjV = zeros(NIND,1);
        Children = zeros(NIND,NVAR);
        ChildrenObjV = zeros(NIND,1);

        % For each chromosome in the population
        for i = 1 : NIND
            % Crossoverrate Pc
            if fitness_LSHGA(ObjV(i),NVAR) < fmean % f < f_gem
                Pc = Kc;
            else %f > f_gem
                Pc = Kc * (fmax-fitness_LSHGA(ObjV(i),NVAR))/(fmax-fmean);
            end

            if rand() < Pc % Do crossover
                [Child,ChildDistance] = crossover_LSHGA(Dist,Chrom(i,:),ObjV(i,1));

                % If after crossover a better solution is not found,
                if ChildDistance == ObjV(i,1)
                    % Chrom without Chrom(i,:)
                    RestChrom = Chrom;
                    RestChrom(i,:) = [];
                    % Mutation rate Pm
                    Pm = Km * min_distance_LSHGA(Chrom(i,:),RestChrom) / (NVAR);

                    if rand() < Pm
                        % Mutate the parent with mutation rate Pm
                        [MutatedParent,MutatedDistance] = mutation_LSHGA(Dist,Chrom(i,:),ObjV(i,1));
                        % Add parent to the Parents pool
                        Parents(i,:) = Chrom(i,:);
                        ParentsObjV(i,1) = ObjV(i,1);
                        % Add a child to the Children pool
                        Children(i,:) = MutatedParent;
                        ChildrenObjV(i,1) = MutatedDistance;
                    else
                        Parents(i,:) = randperm(NVAR);
                        ParentsObjV(i,1) = tspfun2(Parents(i,:),Dist);
                        Children(i,:) = Chrom(i,:);
                        ChildrenObjV(i,1) = ObjV(i,1);
                    end
                else
                    % Add parent to the Parents pool
```

```
                        Parents(i,:) = Chrom(i,:);
                        ParentsObjV(i,1) = ObjV(i,1);
                        % Add a child to the Children pool
                        Children(i,:) = Child;
                        ChildrenObjV(i,1) = ChildDistance;
                    end
                else % Don't do crossover
                    % Add parent to the Parents pool
                    % Parents(i,:) = Chrom(i,:);
                    % ParentsObjV(i,1) = ObjV(i,1);
                    % Add a random new child to the Children pool
                    % Children(i,:) = randperm(NVAR);
                    % ChildrenObjV(i,1) = tspfun2(Children(i,:),Dist);
                    Parents(i,:) = randperm(NVAR);
                    ParentsObjV(i,1) = tspfun2(Parents(i,:),Dist);
                    Children(i,:) = Chrom(i,:);
                    ChildrenObjV(i,1) = ObjV(i,1);
                end
            end

            % The reservation rate Pe
            SquaredMean = mean(fitness_LSHGA(ObjV,NVAR).^2);
            Pe = Ke * (SquaredMean - (fmean^2)) / ((fmax^2) - (fmean^2));

            % Select parents
            AmountOfParantsToSelect = ceil(Pe * NIND);
            [SelectedParents,SelectedParentsObjV] = ...
                binary_tournament_selection_LSHGA(Parents,ParentsObjV,AmountOfParantsToSelect);
            %ind = sus(ParentsObjV,AmountOfParantsToSelect);
            %SelectedParents(:,:) = Parents(ind,:);
            %SelectedParentsObjV = ParentsObjV(ind);

            % Select children
            AmountOfChildrenToSelect = NIND - AmountOfParantsToSelect;
            [SelectedChildren,SelectedChildrenObjV] = ...
                binary_tournament_selection_LSHGA(Children,ChildrenObjV,AmountOfChildrenToSelect);
            %ind = sus(ChildrenObjV,AmountOfChildrenToSelect);
            %SelectedChildren(:,:) = Children(ind,:);
            %SelectedChildrenObjV = ChildrenObjV(ind);

            % Create the new population
            Chrom = [SelectedParents;SelectedChildren];
            ObjV = [SelectedParentsObjV;SelectedChildrenObjV];
            % Increment generation counter
            gen=gen+1
        end
    end
```