Katholieke
Universiteit
Leuven

**Department of
Computer Science**

PROJECT: TSP

Genetic Algorithms and Evolutionary Computing
(B-KUL-H02D1A)

Verbois-Halilovic

Sten Verbois (r0680459)
Armin Halilovic (r0679689)

Academic year 2017-2018

Contents

Introduction

In this report we discuss our solutions and results for the given tasks. We ran experiments for each task to evaluate our solutions. Unless stated otherwise, the experiments were executed with a certain set of parameters and functions. This was done so that we would have a consistent basis to compare results on. The parameters were also chosen in order to leave enough room for improvement so that the effects of different methods can be compared, while at the same time reducing variation of experiments that are too short or do too little work. The default parameters and functions are as follows:

- `number of individuals` = 100
- `maximum number of generations` = 250
- `probability of mutation` = 0.05
- `probability of crossover` = 0.95
- `percentage of elite population` = 0.05
- `subpopulations` = 1
- `loop detection` = off
- `parent selection function` = sus
- `crossover function` = cross_alternating_edges
- `mutation function` = mut_inversion
- `custom stopping criterion` = on
- `custom survivor selection function` = off

The results shown in all tables except for the benchmarks of task 6 are the average results of 10 runs. Every experiment is ran 10 times so that the effects of local optima would be reduced.

The appendix includes extra tables that contain results of experiments and our code that is relevant to the tasks.

Tasks

1.1 Task 2: Initial experiments

The impact of the following parameters on the results of the existing genetic algorithm was tested by varying them while keeping the rest of the parameters at their default values:

- number of individuals (NIND)
- maximum number of generations (MAXGEN)
- percentage of the elite population (ELITIST)
- probability of crossover (PR_CROSS)
- probability of mutation (PR_MUT)
- local loop removal (LOCALLOOP)

The parameter values were chosen so that evenly spread out values from low to high could be tested. The experiments for this task were executed on a subset of the given datasets to keep the tables readable. The datasets range from ones with a small amount of cities to ones with a large amount of cities. The tables for the results of the experiments can be found in appendix 4.1.1. The custom stopping criterion was not used during these experiments, as it is discussed in section 1.2.

1.1.1 Individuals

The mininum path lengths clearly decrease as the number of individuals increases. This is to be expected, as a larger amount of individuals causes a larger amount of mutations and crossover which can positively impact path lengths. Analogously, the maximum path lengths slightly increase as the number of individuals increases. Because of this effect, the mean path lengths remain relatively constant after 100 individuals. Improvements to the minima are higher when increasing a lower number of individuals than when increasing an already high number of individuals.

1.1.2 Generations

Increasing the number of generations shows effects similar to the ones of increasing the number of individuals. However, these results show that increasing an already high number of generations improves the minima a relatively equal amount to increasing a lower number of generations. Also, the means and maxima are lower for high numbers of generations than they are for high numbers of individuals. This can be explained by the survivor selection having more opportunities to select better paths when the amount of generations is higher. Note that the algorithm stops at around 180 generations for dataset "rondrit016.tsp". This is due to the stopping criterion of the existing algorithm being triggered.

1.1.3 Elitism

Judging from the jump in results between 0 elitism and all other values for elitism, it is better to have elitism than to not have it at all. The results improve as the elitism grows from low to medium values, but then worsen again as it goes to high values. A percentage of around 30 to 50 performs best. The results become worse again at high values because crossover and mutation happens a lot less at those values.

1.1.4 Crossover

The crossover results show that the default very high probability performs poorer than all other options for crossover. Furthermore, it is better to have a probability of crossover greater than zero. Similarly to elitism, the results are best at medium probabilities around 30 to 50 percent. When the probability of crossover is high, many more children are created, which creates a lower probability for good parents to be preserved in later generations.

1.1.5 Mutation

The results don't change much for different values of mutation. Judging from the minima, means, and maxima, the best performance again lies around probabilities of 30 to 50 percent. TODO: test nog eens met lagere crossover?

1.1.6 Loop removal

See section 1.4

1.1.7 Mix

After some parameter tuning with the above information in mind, we have come up with a configuration of parameters that performs very well. The results can be seen in table 1.1. The parameters were: `number of individuals` = 450, `maximum number of generations` = 400, `probability of mutation` = 0.4, `probability of crossover` = 0.5, `percentage of elite population` = 0.45, `subpopulations` = 1, `loop detection` = on, `selection function` = sus, `crossover function` = cross_alternating_edges, `mutation function` = mut_inversion, `custom stopping criterion` = off, `custom survivor selection function` = off.

Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	400.0	3.35	3.47	4.87
rondrit018.tsp	400.0	2.93	3.06	4.73
rondrit023.tsp	400.0	3.24	3.42	5.80
rondrit025.tsp	400.0	4.02	4.21	6.80
rondrit048.tsp	400.0	4.42	4.61	7.57
rondrit050.tsp	400.0	5.96	6.21	9.69
rondrit051.tsp	400.0	6.40	6.61	9.63
rondrit067.tsp	400.0	4.58	4.75	7.63
rondrit070.tsp	400.0	7.12	7.45	11.19
rondrit100.tsp	400.0	9.02	9.57	15.69
rondrit127.tsp	400.0	7.37	7.75	11.42

Table 1.1: Existing genetic algorithm after parameter tuning.

1.2 Task 3: Stopping criterion

To implement a new stopping criterion, we looked at the commonly used termination conditions outlined by the textbook. There we see the following suggestions:

1. Maximally allowed CPU time elapses.
2. Total number of fitness evaluations reaches limit.
3. Fitness improvement remains under threshold for a given period of time.
4. Population diversity drops under threshold.

The first and second criteria are useful, either to guarantee the evaluations do not go on forever, or when there is some kind of constraint on system resource usage. In the project template, we already have the guarantee of eventual termination because of the limit on the number of generations, and we do not have to account for system resource constraints in this context.

The fourth criterion is also already present in the template. The default value is so strict (95% equal individuals), it practically is never reached.

We decided to implement the third criterion. With this condition, termination occurs when the fitness of the best individual does not improve above a threshold for a given period of time. This period of time is expressed in terms of a certain number of generations. We chose to define this number of generations to be 10% of the specified maximum number of generations.

The results of our experiments with this new termination condition are displayed in table 1.2. We clearly see that it does succeed in avoiding computation of useless generations where the fitness does not improve for a long time. As expected, the achieved score is often slightly worse in comparison to the default stopping criterion. This is because of the fact that improvements may still happen at a much later point in time. With a more strict stopping criterion, we make a trade-off between avoiding almost useless calculations and the chance for slightly better scores.

	Default stopping criterion				Custom stopping criterion			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	184.0	3.3752	3.4664	4.4159	75.2	3.5061	4.6191	6.1368
rondrit018.tsp	232.4	3.0469	4.3285	5.8945	92.6	3.3740	5.1082	7.1152
rondrit023.tsp	250.0	4.0076	6.6840	9.6150	84.3	4.5584	7.0113	9.6520
rondrit025.tsp	250.0	5.0209	8.7816	12.2293	113.0	5.7750	9.1623	12.6250
rondrit048.tsp	250.0	9.6163	14.4618	18.6610	83.5	10.6749	15.1442	19.1976
rondrit050.tsp	250.0	13.9410	19.4818	24.0982	120.1	15.2353	20.3705	24.6610
rondrit051.tsp	250.0	13.2179	18.3265	22.4049	90.3	14.7031	19.0930	23.3291
rondrit067.tsp	250.0	13.6788	18.6416	22.5891	136.5	14.3745	18.7647	22.5262
rondrit070.tsp	250.0	21.7930	28.4651	34.0860	99.3	23.3636	29.3783	34.7131
rondrit100.tsp	250.0	34.1562	43.1992	50.1761	100.6	36.6088	44.4320	50.6472
rondrit127.tsp	250.0	22.0104	26.3718	29.7679	119.6	22.8799	26.7850	30.2722

Table 1.2: Comparison between default and custom stopping criteria.

1.3 Task 4: Other representation

The given project template uses adjacency representation by default for TSP paths. We have chosen to use path representation as the alternative one. Conversion between the two representations was already possible thanks to the `adj2path` and `path2adj` functions in the template. To do crossover with path representation, we implemented the Order Crossover method (function `cross_order`) as described in the textbook. Simple Inversion Mutation, which is a mutation operator for path representation, was already provided in the template (`mut_inversion`). We have decided to extend this and have added a function for Inversion Mutation (`mut_inversion2`). Our custom stopping criterion was not used during these experiments in order to allow for the different crossover and mutation operators to have a stronger impact.

Table 1.3 contains the results of experiments with different crossover operators. The function `cross_alternating_edges` implements Alternating Edge Crossover and is provided in the template. Order Crossover performs significantly better than Alternating Edge Crossover; all of the path lengths with Order Crossover are lower for every dataset. This can be attributed to Order Crossover retaining longer sequences of genetic material of parents than Alternating Edge Crossover. Also, this results in the path lengths converging to a value earlier, which is why the numbers of generations are lower.

	Alternating Edge Crossover				Order Crossover			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	146.1	3.41	3.50	4.25	48.3	3.37	3.43	4.28
rondrit018.tsp	249.9	3.02	4.56	6.27	57.1	3.01	3.04	3.84
rondrit023.tsp	250.0	3.94	6.75	9.26	83.3	3.55	3.57	4.57
rondrit025.tsp	250.0	5.15	8.79	12.73	81.1	4.46	4.48	5.41
rondrit048.tsp	250.0	9.43	14.56	19.01	159.8	5.55	5.56	6.38
rondrit050.tsp	250.0	13.61	19.62	24.45	180.3	7.87	7.98	9.21
rondrit051.tsp	250.0	13.53	18.37	22.48	179.3	8.23	8.25	9.19
rondrit067.tsp	250.0	13.54	18.41	22.31	218.5	7.03	7.10	7.96
rondrit070.tsp	250.0	21.31	28.53	34.72	212.9	11.37	11.66	13.33
rondrit100.tsp	250.0	33.95	42.81	49.54	236.7	17.47	18.00	20.00
rondrit127.tsp	250.0	21.97	26.19	29.68	249.3	12.46	12.68	13.67

Table 1.3: Results of different crossover functions.

Table 1.4 contains the results of experiments with different mutation functions. The function `mut_inversion` implements Simple Inversion Mutation and is provided in the template. While both functions perform roughly the same, the one provided in the template performs slightly better when looking at the minima, means, and maxima. This is likely due to Inversion Mutation changing genetic material more than Simple Inversion Mutation; moving the randomly chosen subpath to a different position in the path after reversing it is more likely to have a negative effect when the path is near a (local) optimum.

	Simple Inversion Mutation				Inversion Mutation			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	154.0	3.39	3.57	4.42	184.3	3.40	3.84	4.95
rondrit018.tsp	214.2	3.04	4.18	5.69	236.3	3.03	4.33	5.94
rondrit023.tsp	250.0	3.94	6.57	9.38	250.0	3.92	6.68	9.35
rondrit025.tsp	250.0	5.14	8.87	12.55	250.0	5.20	8.91	12.47
rondrit048.tsp	250.0	9.49	14.49	19.10	250.0	9.95	14.71	19.19
rondrit050.tsp	250.0	13.94	19.57	24.39	250.0	13.79	19.61	24.56
rondrit051.tsp	250.0	13.41	18.25	22.38	250.0	13.81	18.69	22.49
rondrit067.tsp	250.0	14.27	18.63	22.49	250.0	13.66	18.67	22.49
rondrit070.tsp	250.0	21.99	28.78	34.46	250.0	21.52	28.77	34.76
rondrit100.tsp	250.0	34.80	43.18	50.27	250.0	34.24	43.06	49.31
rondrit127.tsp	250.0	21.90	26.26	29.66	250.0	21.94	26.44	29.55

Table 1.4: Results of different mutation functions.

Table 1.5 shows results after some parameter tuning with the `cross_order` and `mut_inversion` operators. The parameters used were: `number of individuals` = 1000, `maximum number of generations` = 1000, `probability of mutation` = 0.30, `probability of crossover` = 0.50, `percentage of elite population` = 0, `subpopulations` = 1, `loop detection` = on, `selection function` = sel_tournament, `crossover function` = cross_order, `mutation function` = mut_inversion, `custom stopping criterion` = on, `custom survivor selection function` = on (round robin tournament).

Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	105.8	3.36	3.38	4.03
rondrit018.tsp	107.7	2.94	2.96	3.78
rondrit023.tsp	109.3	3.29	3.31	4.43
rondrit025.tsp	110.4	4.05	4.08	5.21
rondrit048.tsp	133.3	4.52	4.54	5.54
rondrit050.tsp	135.0	6.19	6.22	7.37
rondrit051.tsp	134.3	6.59	6.62	7.70
rondrit067.tsp	156.5	4.74	4.77	5.66
rondrit070.tsp	159.6	7.48	7.51	8.71
rondrit100.tsp	217.8	8.58	8.62	9.86
rondrit127.tsp	282.2	6.37	6.38	6.97

Table 1.5: Results for operators for path representation after parameter tuning.

1.4 Task 5: Local optimisation

For this task, we are testing the local optimization already present in the template. This optimization takes a path and tries to remove local loops with a path length of up to 3.

The results of our experiments with local optimisation disabled and enabled are displayed in table 1.6. With default values for other parameters, this results in major improvements to both path lenghts and number of generations needed. This is the case for every single dataset.

	Local optimisation disabled				Local optimisation enabled			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	101.2	3.4868	4.7997	6.3890	38.0	3.3500	3.4509	4.3388
rondrit018.tsp	111.8	3.2462	4.9590	6.8692	46.1	2.9420	3.3895	4.7746
rondrit023.tsp	111.6	4.3773	6.8276	9.5882	63.4	3.2868	5.2676	7.7579
rondrit025.tsp	98.3	6.0037	9.4285	12.7233	67.6	4.1675	6.9177	10.1071
rondrit048.tsp	124.8	10.5785	15.0395	19.1395	82.5	6.3315	10.7464	14.4598
rondrit050.tsp	94.2	15.1419	20.1793	24.5441	78.6	9.6336	14.6699	19.1436
rondrit051.tsp	130.6	14.2812	18.7077	22.6377	67.0	9.2661	13.7145	17.3316
rondrit067.tsp	79.5	15.1933	19.1011	22.8508	59.7	8.7378	13.2005	17.0004
rondrit070.tsp	109.5	22.8502	29.4624	35.2209	63.2	13.9933	20.5239	26.3143
rondrit100.tsp	112.9	36.2358	44.1065	50.3106	65.8	19.9940	29.3512	36.1405
rondrit127.tsp	96.3	22.7452	26.8796	30.0337	68.4	13.8480	18.6564	22.1228

Table 1.6: Comparison between local optimisation disabled (left) and local optimisation enabled (right).

1.5 Task 7: Optional tasks

1.5.1 7a: Parent selection

Additional parent selection methods we implemented are Fitness Proportional Selection (`sel_fit_prop`) and Tournament Selection (`sel_tournament`). Both of them use the same parameters as the existing implementation of Stochastic Universal Sampling so we could easily swap them in.

The results of our experiments with all three parent selection methods are displayed in tables 1.7, 1.8, and 1.9. For Tournament Selection, the tournament size parameter k was set to 5. We see that it achieves much better path lengths than SUS, especially when dealing with a high number of cities. The number of generations needed is quite a bit higher so it seems that SUS gets stuck in local optima while tournament selection is able to select better parents for future generations. This is not the case for Fitness Proportional Selection. This parent selection method seems to match the Stochastic Universal Sampling in terms of path lengths and number of generations needed. This is probably because the methods are very similar to each other.

	Stochastic Universal Sampling			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	76.5	3.5071	4.7287	6.2973
rondrit018.tsp	112.0	3.2458	4.9674	6.9154
rondrit023.tsp	89.8	4.3998	6.9255	9.6426
rondrit025.tsp	81.6	5.8037	9.2203	12.4602
rondrit048.tsp	115.7	10.5298	15.0273	18.9340
rondrit050.tsp	74.1	15.5617	20.4526	24.5841
rondrit051.tsp	101.3	14.7763	19.2497	23.2194
rondrit067.tsp	98.2	14.7791	19.1254	22.6560
rondrit070.tsp	117.4	22.9912	29.3627	34.8783
rondrit100.tsp	91.4	37.0343	44.2892	51.0612
rondrit127.tsp	120.9	22.9939	26.6807	30.0838

Table 1.7: Results when using Stochastic Universal Sampling as parent selection method.

	Tournament Selection			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	47.6	3.4307	3.4489	4.1168
rondrit018.tsp	69.8	3.0751	3.1000	3.9496
rondrit023.tsp	97.2	3.4223	3.4448	4.3663
rondrit025.tsp	107.1	4.2080	4.2289	5.2293
rondrit048.tsp	220.2	5.2384	5.3288	6.7016
rondrit050.tsp	228.5	7.4060	7.4469	8.7495
rondrit051.tsp	223.4	7.7256	7.8091	9.0010
rondrit067.tsp	246.7	6.5806	6.6540	7.7929
rondrit070.tsp	250.0	10.4302	11.1550	14.0566
rondrit100.tsp	241.3	18.2543	19.2141	22.2169
rondrit127.tsp	250.0	13.2633	13.8539	15.8462

Table 1.8: Results when using Tournament Selection as parent selection method.

	Fitness Proportional Selection			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	87.6	3.5253	4.7702	6.31735983363770
rondrit018.tsp	106.5	3.1629	5.0059	6.88030465197992
rondrit023.tsp	72.1	4.6849	7.0675	9.61113827405700
rondrit025.tsp	115.5	5.7095	9.0493	12.57024535746879
rondrit048.tsp	93.0	10.6731	15.1775	19.06378066242915
rondrit050.tsp	84.1	15.6907	20.4983	24.46309616398185
rondrit051.tsp	114.1	14.6637	19.0878	22.88529001223297
rondrit067.tsp	101.7	14.8841	19.0997	22.56887763300833
rondrit070.tsp	101.4	23.2645	29.7146	35.05263230223609
rondrit100.tsp	95.0	37.2210	43.9292	50.80513760208527
rondrit127.tsp	114.7	22.9952	26.8508	29.97739386912141

Table 1.9: Results when using Fitness Proportional Selection as parent selection method.

1.5.2 7b: Survivor selection

Round robin tournament was chosen as the other strategy for survivor selection. The results for different percentages of elitism for survival selection can be found in appendix 4.1.1 in table 4.13. To evaluate how round robin tournament performs compared to the already implemented elitism, the elitism percentage is set to 0% in an experiment. Also, an experiment is done where round robin tournament is combined with an elitism percentage of 30%. Table 1.10 contains the results of the experiments for this task. The size of the tournaments was set to 10, as recommended by the textbook.

The round robin tournament strategy outperforms every percentage of elitism. Combining round robin tournament with elitism results in longer path lengths overall. This happens because the elitism causes the round robin tournament to be executed on a smaller population; the two methods compete with each other. Therefore, it is better to set elitism to zero when using round robin tournament.

	Round robin tournament				Elitism + round robin tournament			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	61.2	3.39	3.39	3.39	75.9	3.38	3.39	3.49
rondrit018.tsp	74.3	3.00	3.00	3.16	91.4	3.01	3.01	3.13
rondrit023.tsp	110.8	3.41	3.41	3.78	118.7	3.53	3.56	3.82
rondrit025.tsp	126.9	4.18	4.18	4.33	153.9	4.36	4.41	4.58
rondrit048.tsp	231.9	5.52	5.60	5.77	241.3	6.17	6.37	6.98
rondrit050.tsp	250.0	7.67	7.77	8.33	224.5	9.37	9.68	10.51
rondrit051.tsp	250.0	7.88	7.99	8.28	234.5	9.14	9.35	9.92
rondrit067.tsp	250.0	7.62	7.83	8.31	238.0	9.41	9.69	10.47
rondrit070.tsp	241.3	12.76	13.04	14.16	249.2	14.50	14.88	15.58
rondrit100.tsp	240.0	23.07	23.56	24.62	172.0	28.16	28.98	31.47
rondrit127.tsp	250.0	15.67	15.92	16.60	216.4	17.52	18.07	19.24

Table 1.10: Results for the already implemented elitism combined with our round robin tournament survivor selection.

1.5.3 7c: Diversity preservation

In order to preserve population diversity, we adapted a few of the functions in the template to work with subpopulations, simulating the island model. In addition to the number of supopulations, the number of individuals was also increased from 100 to 800 in these experiments. This way there is a reasonable amount of individuals in each subpopulation for each experiment.

The results displayed in the appendix 4.1.2 in tables 4.16 through 4.20 show tests performed with 1, 2, 5, 10 and 20 subpopulations or islands. The results show a clear downwards trend in terms of path length when increasing the number of subpopulations. When the number of subpopulations is high, the number of individuals decreases enough to negatively impact path length. When the number of subpopulations is lower (e.g. 2 or 5) the performance is comparable to results of experiments performed for task 2 with the corresponding number of individuals in a single subpopulation.

We did not implement any communication mechanisms between subpopulations. We suspect that the addition of migration might improve the results we observed.

1.6 Task 6: Benchmark problems

For this task, we have selected a set of parameters and methods based on all of the results above. Our algorithm is evaluated by running it on the given benchmark problems and calculating the relative error of the results of the algorithm to the known optimal paths of the benchmark problems.

The parameters used were: `number of individuals` = 1000, `maximum number of generations` = 1000, `probability of mutation` = ???, `probability of crossover` = ???, `percentage of elite population` = ???, `subpopulations` = 1, `loop detection` = 1, `selection function` = sel_tournament, `crossover function` = cross_order, `mutation function` = mut_inversion, `custom stopping criterion` = off, `custom survivor selection function` = on (round robin tournament).

The results are shown in table **??**. Table**??** contains results for the benchmark problems when using the default parameters mentioned in the introduction. Judging from the relative errors, we can conclude that our solutions are TODO

1.7 Conclusion

For this project, we ran and evaluated many experiments. In task 2, we looked at the effects of parameter tuning on the 'bare' genetic algorithm. We mainly noticed that by themselves, the parameters all have a relatively small impact. However, tuning the parameters together can give much better results on the datasets.

The new stopping criterion developed for task 3 allows us to gain on computation time while sacrificing a bit in minimal path length. Task 4 showed us that Order Crossover on chromosomes in path representation outperforms Alternating Edge Crossover for every dataset, and that Inversion Mutation is not really better than its Simple version in this case. Also, local loop removal is a must according to the experiments we performed for task 5. It decreases both path length and number of generations required.

For the optional tasks, we decided to implement Fitness Proportional Selection and Tournament Selection for task 7a, where we saw that only tournament selection had an impact on performance. Furthermore, we implemented RR Tournament survivor selection for task 7b. It had a positive effect when used by itself, but this effect decreased when used with the already implemented elitism. Lastly, we experimented with multiple subpopulations for task 7c in an attempt to avoid premature convergence, but it ended up not being helpful in the context of this project.

We put the knowledge gained from doing all these tasks to the test when tuning the final parameters for the benchmark tests, and achieved ... TODO resultaten.

Appendix

4.1 Tables

4.1.1 Task 2

Dataset	# Generations	Min	Mean	Max
number of individuals = 50				
rondrit016.tsp	240.3	3.43	4.79	6.15
rondrit048.tsp	250.0	10.75	15.48	18.92
rondrit067.tsp	250.0	14.85	19.37	22.58
rondrit127.tsp	250.0	22.80	27.03	29.92
number of individuals = 100				
rondrit016.tsp	191.7	3.40	3.82	4.70
rondrit048.tsp	250.0	9.88	14.72	18.82
rondrit067.tsp	250.0	13.49	18.52	22.12
rondrit127.tsp	250.0	21.59	26.30	29.80
number of individuals = 250				
rondrit016.tsp	250.0	3.37	4.75	6.53
rondrit048.tsp	250.0	8.94	14.62	19.69
rondrit067.tsp	250.0	12.50	18.28	22.59
rondrit127.tsp	250.0	20.64	25.96	30.16
number of individuals = 500				
rondrit016.tsp	246.5	3.36	4.60	6.49
rondrit048.tsp	250.0	7.98	14.06	19.49
rondrit067.tsp	250.0	11.81	17.95	22.71
rondrit127.tsp	250.0	19.49	25.47	29.85
number of individuals = 1000				
rondrit016.tsp	243.1	3.35	4.52	6.55
rondrit048.tsp	250.0	7.81	14.13	19.84
rondrit067.tsp	250.0	11.15	17.86	23.32
rondrit127.tsp	250.0	18.92	25.26	30.12

Table 4.11: Existing genetic algorithm with varying amount of individuals.

Dataset	# Generations	Min	Mean	Max
max number of generations = 100				
rondrit016.tsp	100.0	3.46	4.74	6.28
rondrit048.tsp	100.0	10.59	15.11	19.11
rondrit067.tsp	100.0	15.02	19.27	22.78
rondrit127.tsp	100.0	22.90	26.87	30.22
max number of generations = 250				
rondrit016.tsp	176.6	3.40	3.50	4.23
rondrit048.tsp	250.0	9.60	14.58	18.97
rondrit067.tsp	250.0	13.52	18.59	22.90
rondrit127.tsp	250.0	21.99	26.49	30.43
max number of generations = 500				
rondrit016.tsp	184.2	3.38	3.40	4.28
rondrit048.tsp	500.0	8.41	13.89	18.68
rondrit067.tsp	500.0	12.46	17.94	22.27
rondrit127.tsp	500.0	20.26	25.27	28.98
max number of generations = 1000				
rondrit016.tsp	180.7	3.38	3.40	4.24
rondrit048.tsp	1000.0	7.63	13.77	18.62
rondrit067.tsp	1000.0	11.20	17.04	21.68
rondrit127.tsp	1000.0	18.83	24.86	28.70

Table 4.12: Existing genetic algorithm with varying amount of maximum generations.

Dataset	# Generations	Min	Mean	Max
percentage of the elite population = 0.00				
rondrit016.tsp	250.0	4.50	5.69	7.18
rondrit048.tsp	250.0	14.03	17.31	20.50
rondrit067.tsp	250.0	18.16	21.13	23.96
rondrit127.tsp	250.0	26.28	28.72	31.22
percentage of the elite population = 0.05				
rondrit016.tsp	198.6	3.38	3.81	4.80
rondrit048.tsp	250.0	8.92	14.23	18.51
rondrit067.tsp	250.0	13.33	18.46	22.14
rondrit127.tsp	250.0	22.02	26.30	29.37
percentage of the elite population = 0.10				
rondrit016.tsp	73.2	3.38	3.39	4.01
rondrit048.tsp	250.0	7.29	9.67	13.38
rondrit067.tsp	250.0	11.04	14.38	18.90
rondrit127.tsp	250.0	19.11	22.87	27.30
percentage of the elite population = 0.30				
rondrit016.tsp	52.8	3.39	3.41	4.08
rondrit048.tsp	247.7	6.42	7.43	11.04
rondrit067.tsp	250.0	9.19	10.32	13.71
rondrit127.tsp	250.0	17.56	18.95	22.25
percentage of the elite population = 0.50				
rondrit016.tsp	68.3	3.40	3.41	4.12
rondrit048.tsp	250.0	7.03	7.61	10.62
rondrit067.tsp	250.0	10.36	11.28	14.48
rondrit127.tsp	250.0	18.36	19.33	22.32
percentage of the elite population = 0.70				
rondrit016.tsp	112.3	3.37	3.38	3.97
rondrit048.tsp	250.0	8.15	8.87	11.99
rondrit067.tsp	250.0	11.75	12.51	15.15
rondrit127.tsp	250.0	19.56	20.56	23.64
percentage of the elite population = 0.95				
rondrit016.tsp	250.0	3.57	3.89	4.71
rondrit048.tsp	250.0	11.56	12.56	15.06
rondrit067.tsp	250.0	15.43	16.75	18.79
rondrit127.tsp	250.0	23.75	24.80	26.80

Table 4.13: Existing genetic algorithm with varying percentage of the elite population.

Dataset	# Generations	Min	Mean	Max
probability of crossover = 0.00				
rondrit016.tsp	33.0	4.17	4.18	4.64
rondrit048.tsp	159.7	8.31	8.33	9.39
rondrit067.tsp	177.2	11.80	11.81	12.50
rondrit127.tsp	197.7	20.77	20.82	21.12
probability of crossover = 0.10				
rondrit016.tsp	42.3	3.63	3.65	4.38
rondrit048.tsp	163.1	7.21	7.23	8.04
rondrit067.tsp	192.5	9.84	9.90	10.85
rondrit127.tsp	250.0	16.72	16.88	18.18
probability of crossover = 0.30				
rondrit016.tsp	36.9	3.57	3.58	4.29
rondrit048.tsp	196.8	6.14	6.17	7.44
rondrit067.tsp	245.8	7.79	7.95	9.63
rondrit127.tsp	220.5	15.97	16.14	17.26
probability of crossover = 0.50				
rondrit016.tsp	39.9	3.55	3.57	4.33
rondrit048.tsp	199.1	6.09	6.31	8.08
rondrit067.tsp	217.4	8.57	8.91	10.84
rondrit127.tsp	250.0	15.25	15.81	18.10
probability of crossover = 0.70				
rondrit016.tsp	58.6	3.42	3.44	4.27
rondrit048.tsp	221.3	6.43	6.90	9.16
rondrit067.tsp	243.0	9.16	10.09	13.20
rondrit127.tsp	250.0	16.82	19.07	23.18
probability of crossover = 0.95				
rondrit016.tsp	159.1	3.39	3.42	4.18
rondrit048.tsp	250.0	9.67	14.65	18.68
rondrit067.tsp	250.0	13.77	18.74	22.76
rondrit127.tsp	250.0	21.76	26.14	29.46

Table 4.14: Existing genetic algorithm with varying probability of crossover.

Dataset	# Generations	Min	Mean	Max
probability of mutation = 0.00				
rondrit016.tsp	122.1	3.38	3.39	3.66
rondrit048.tsp	250.0	9.95	14.56	18.82
rondrit067.tsp	250.0	14.24	18.42	22.35
rondrit127.tsp	250.0	22.66	26.70	30.03
probability of mutation = 0.05				
rondrit016.tsp	167.0	3.39	3.41	4.09
rondrit048.tsp	250.0	9.72	14.68	18.93
rondrit067.tsp	250.0	13.91	18.67	22.32
rondrit127.tsp	250.0	21.63	26.11	29.80
probability of mutation = 0.10				
rondrit016.tsp	241.0	3.39	4.19	5.40
rondrit048.tsp	250.0	9.09	14.32	18.58
rondrit067.tsp	250.0	13.58	18.66	22.57
rondrit127.tsp	250.0	21.53	26.12	29.56
probability of mutation = 0.30				
rondrit016.tsp	250.0	3.45	4.99	6.64
rondrit048.tsp	250.0	8.88	14.69	19.14
rondrit067.tsp	250.0	12.81	18.22	22.51
rondrit127.tsp	250.0	21.28	26.02	29.34
probability of mutation = 0.50				
rondrit016.tsp	250.0	3.41	5.14	6.70
rondrit048.tsp	250.0	8.98	14.70	19.21
rondrit067.tsp	250.0	12.50	18.55	22.42
rondrit127.tsp	250.0	20.78	25.97	29.68
probability of mutation = 0.70				
rondrit016.tsp	250.0	3.41	5.30	6.86
rondrit048.tsp	250.0	9.17	15.13	19.73
rondrit067.tsp	250.0	12.63	18.44	22.64
rondrit127.tsp	250.0	20.91	26.20	29.78
probability of mutation = 0.95				
rondrit016.tsp	250.0	3.45	5.46	7.07
rondrit048.tsp	250.0	9.24	15.34	19.50
rondrit067.tsp	250.0	13.06	18.83	22.93
rondrit127.tsp	250.0	20.91	26.24	29.66

Table 4.15: Existing genetic algorithm with varying probability of mutation.

4.1.2 Task 7c

	# subpopulations = 1			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	87.3	3.3789	4.8268	6.8628
rondrit018.tsp	92.0	3.0807	5.1325	7.5719
rondrit023.tsp	132.2	3.8457	6.8379	10.2530
rondrit025.tsp	117.7	4.9808	9.0435	13.4739
rondrit048.tsp	158.8	8.7074	14.5176	19.8265
rondrit050.tsp	105.7	13.6085	20.0173	25.5480
rondrit051.tsp	117.9	12.9100	18.5933	23.6349
rondrit067.tsp	102.1	13.5305	18.8152	23.4651
rondrit070.tsp	129.1	20.1628	28.6489	35.5801
rondrit100.tsp	121.7	33.4117	43.3215	52.2091
rondrit127.tsp	139.6	20.8300	26.1699	30.2266

Table 4.16: Results when using a single subpopulation.

	# subpopulations = 2			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	86.5	3.7069	5.3631	7.6527
rondrit018.tsp	104.1	3.3891	5.6142	8.1424
rondrit023.tsp	119.4	4.2318	7.5442	11.2955
rondrit025.tsp	172.2	5.1527	9.7101	14.4419
rondrit048.tsp	109.0	10.6332	16.4698	22.6468
rondrit050.tsp	130.0	14.5412	21.7655	27.6456
rondrit051.tsp	163.0	13.6866	20.2262	25.9560
rondrit067.tsp	145.5	14.3458	20.4498	25.6300
rondrit070.tsp	141.1	22.5607	31.5559	39.1275
rondrit100.tsp	128.5	36.5586	47.6320	57.1317
rondrit127.tsp	153.6	22.8718	28.8169	33.7587

Table 4.17: Results when using two subpopulations.

	# subpopulations = 5			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	96.7	3.7377	5.3840	7.5751
rondrit018.tsp	110.8	3.3874	5.6606	8.4666
rondrit023.tsp	149.4	4.2469	7.5998	11.2493
rondrit025.tsp	166.8	5.4264	9.9891	14.8461
rondrit048.tsp	150.1	10.2174	16.3796	22.4803
rondrit050.tsp	116.8	15.0743	22.0950	28.6421
rondrit051.tsp	125.3	14.4879	20.6224	26.1835
rondrit067.tsp	113.1	14.9047	20.8687	26.0722
rondrit070.tsp	128.7	23.1888	32.0881	39.9062
rondrit100.tsp	107.8	38.2434	48.5536	57.3213
rondrit127.tsp	176.2	22.6508	28.8988	33.6638

Table 4.18: Results when using five subpopulations.

	# subpopulations = 10			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	101.6	3.7363	5.4124	7.7952
rondrit018.tsp	113.0	3.4010	5.6424	8.4560
rondrit023.tsp	134.2	4.3630	7.7145	11.3325
rondrit025.tsp	117.1	5.7916	10.2213	14.8791
rondrit048.tsp	162.2	9.8346	16.3824	22.4776
rondrit050.tsp	135.3	14.4837	22.0135	28.4629
rondrit051.tsp	133.6	14.3287	20.6799	26.4687
rondrit067.tsp	183.9	13.6331	20.3402	25.7013
rondrit070.tsp	128.9	22.9472	32.1190	39.3304
rondrit100.tsp	151.4	36.4110	47.9459	57.1262
rondrit127.tsp	189.3	22.6947	28.9111	33.7984

Table 4.19: Results when using ten subpopulations.

	# subpopulations = 20			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	97.1	3.7418	5.4420	7.7813
rondrit018.tsp	101.2	3.4532	5.7598	8.4451
rondrit023.tsp	120.9	4.4710	7.7258	11.2595
rondrit025.tsp	138.6	5.5890	10.2273	15.0354
rondrit048.tsp	128.2	10.4201	16.5968	22.4393
rondrit050.tsp	133.5	14.9730	22.1471	28.3386
rondrit051.tsp	140.1	14.3330	20.6143	26.3868
rondrit067.tsp	135.2	14.8073	20.9061	26.2982
rondrit070.tsp	142.1	23.0770	32.1201	39.8267
rondrit100.tsp	144.2	36.8373	48.0742	57.2916
rondrit127.tsp	136.0	23.5869	29.2883	34.0429

Table 4.20: Results when using twenty subpopulations.

4.2 Code

Listing 4.1: The main algorithm - src/run_ga.m

```
% run_ga.m (RUN GENETIC ALGORITHM)
%
% Input parameters:
% x, y - coordinates of the cities
% NIND - number of individuals
% MAXGEN - maximal number of generations
% ELITIST - percentage of elite population
% STOP_PERCENTAGE - percentage of equal fitness (stop criterium)
% PR_CROSS - probability for crossover
% PR_MUT - probability for mutation
% CROSSOVER - the crossover operator
% MUTATION - the mutation operator
% LOCALLOOP - local loop removal on/off
% CUSTOMSTOP - custom stopping criterion on/off
% CUSTOMSS - custom survivor selection on/off
% SELECTION - the parent selection function (sus, sel_tournament,
% sel_fit_prop, ...)
% ah1, ah2, ah3 - axes handles to visualise tsp
%
% Output parameters:
% best - vector of the best result of every iteration
% mean_fits - vector of the mean result of every iteration
% worst - vector of the worst result of every iteration

function [best, mean_fits, worst] = run_ga(x, y, NIND, MAXGEN, NVAR,
    ELITIST, STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, MUTATION,
    LOCALLOOP, CUSTOMSTOP, CUSTOMSS, SELECTION, SUBPOP, ah1, ah2, ah3)

    GGAP = 1 - ELITIST;

    best = zeros(1, MAXGEN);
    mean_fits = zeros(1,MAXGEN+1);
    worst = zeros(1,MAXGEN+1);

    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,:) = path2adj(randperm(NVAR));
    end
    % evaluate initial population
    ObjV = tspfun(Chrom, Dist);

```

```
48      % number of individuals of equal fitness needed to stop
49      stopN = ceil(STOP_PERCENTAGE*NIND);
50
51      gen = 0;
52      % generational loop
53      while gen < MAXGEN
54          sObjV = sort(ObjV);
55          best(gen+1) = min(ObjV);
56          minimum = best(gen+1);
57          mean_fits(gen+1) = mean(ObjV);
58          worst(gen+1) = max(ObjV);
59          for t = 1:size(ObjV,1)
60              if (ObjV(t) == minimum)
61                  break;
62              end
63          end
64
65          if nargin == 19
66              visualizeTSP(x, y, adj2path(Chrom(t,:)), minimum, ah1, gen,
                    best, mean_fits, worst, ah2, ObjV, NIND, ah3);
67          end
68
69          % stopping criterion
70          if CUSTOMSTOP == 1
71              % Stop if the minimum has not improved in the interval of the
72              % last 0.1*MAXGEN generations. The change in fitness is
73              % calculated by taking the sum of the differences between the
74              % interval and the current best fitness.
75              idx = floor(gen-0.1*MAXGEN);
76              if (idx > 1) && (sum(abs(best(idx:gen) - minimum)) <= 1e-15)
77                  break;
78              end
79          elseif (sObjV(stopN)-sObjV(1) <= 1e-15)
80              % Stop if the difference between the best and the stopN'th
81              % best fitness value is very small.
82              break;
83          end
84
85          % assign fitness values to entire population
86          FitnV = ranking(ObjV);
87
88          % select individuals for breeding
89          SelCh = select(SELECTION, Chrom, FitnV, GGAP);
90
91          % recombine individuals (crossover)
92          SelCh = crossover_tsp(CROSSOVER, SelCh, PR_CROSS, SUBPOP);
93          SelCh = mutate_tsp(MUTATION, SelCh, PR_MUT, SUBPOP);
94
95          % evaluate offspring, call objective function
96          ObjVSel = tspfun(SelCh, Dist);
97
98          % reinsert offspring into population
99          if CUSTOMSS == 0
```

```
            [Chrom, ObjV] = reins(Chrom, SelCh, SUBPOP, 1, ObjV, ObjVSel);
        else
            [Chrom, ObjV] = sur_sel_rr_tournament(Chrom, SelCh, ObjV,
                ObjVSel, 10);
        end

        Chrom = tsp_improve_population(NIND, NVAR, Chrom, LOCALLOOP, Dist)
            ;

        gen = gen+1;
    end
end
```

Listing 4.2: Order crossover for task 4- src/cross_order.m

```
% CROSS_ORDER.M (ORDER CROSSOVER)
%
% Order crossover for TSP.
%
% Input parameters:
% ParentOne, ParentTwo - The TSP individuals to apply crossover on in a
% certain representation.
% Representation - The representation the given parents are in.
% If omitted, 2 (path) is assumed.
%
% Output parameters:
% ChildOne, ChildTwo - Chromosomes created by mating, ready to be
% mutated and/or evaluated, in the same format
% as OldChrom.

function [ChildOne, ChildTwo] = cross_order(ParentOne, ParentTwo)

ParentOne = adj2path(ParentOne);
ParentTwo = adj2path(ParentTwo);

[~, cols] = size(ParentOne);
ChildOne = zeros(1, cols);
ChildTwo = zeros(1, cols);

rnd = sort(randi(cols, [1, 2]));
a = rnd(1);
b = rnd(2);

ChildOne(a:b) = ParentOne(a:b);
ChildTwo(a:b) = ParentTwo(a:b);

childOneIdx = rem(b, cols) + 1;
childTwoIdx = rem(b, cols) + 1;
for i = 1:cols
    current = rem(b + i-1, cols) + 1;

    if (all(ChildTwo ~= ParentOne(current)))
        ChildTwo(childTwoIdx) = ParentOne(current);
        childTwoIdx = rem(childTwoIdx, cols) + 1;
```

```
    end

    if (all(ChildOne ~= ParentTwo(current)))
        ChildOne(childOneIdx) = ParentTwo(current);
        childOneIdx = rem(childOneIdx, cols) + 1;
    end
end

ChildOne = path2adj(ChildOne);
ChildTwo = path2adj(ChildTwo);

end
```

Listing 4.3: High level crossover function- src/crossover_tsp.m

```
% CROSSOVER_TSP.M (Crossover for TSP high-level function)
%
% This function performs recombination (crossover) between pairs of
% individuals and returns the new individuals after mating.
% The function handles multiple populations and calls a given low-level
% function for the actual recombination process.
%
% Input parameters:
% CROSS_F - String containing the name of the crossover function
% OldChrom - Matrix containing the chromosomes of the old
% population. Each line corresponds to one individual
% PR_CROSS - (optional) Scalar containing the probability of
% recombination/crossover occurring between pairs
% of individuals. If omitted or NaN, 0.95 is assumed.
% SUBPOP - (optional) Number of subpopulations.
% If omitted or NaN, 1 subpopulation is assumed.
%
% Output parameter:
% NewChrom - Matrix containing the chromosomes of the population
% after recombination in the same format as OldChrom.

function NewChrom = crossover_tsp(CROSS_F, OldChrom, PR_CROSS, SUBPOP)

% Check parameter consistency
if nargin < 2; error('Not enough input parameter'); end

% Probability of crossover
if nargin < 3; PR_CROSS = 0.95;
elseif nargin > 2
  if isempty(PR_CROSS), PR_CROSS = 0.7;
  elseif isnan(PR_CROSS), PR_CROSS = 0.7;
  elseif length(PR_CROSS) ~= 1, error('PR_CROSS must be a scalar');
  elseif (PR_CROSS < 0 | PR_CROSS > 1), error('PR_CROSS must be a scalar
      in [0, 1]');
  end
end

% Population size
[rows, cols] = size(OldChrom);
```

```
NewChrom = zeros(rows, cols);

if nargin < 4; SUBPOP = 1;
elseif nargin > 3
  if isempty(SUBPOP), SUBPOP = 1;
  elseif isnan(SUBPOP), SUBPOP = 1;
  elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar');
  end
end

if (rows/SUBPOP) ~= floor(rows/SUBPOP)
    rows
    SUBPOP
    error('OldChrom and SUBPOP disagree');
end
rows = rows/SUBPOP; % Compute number of individuals per subpopulation

% Select individuals of subpopulations and call low level function
for subpop = 1:SUBPOP
    SubChrom = OldChrom((subpop-1)*rows+1:subpop*rows, :);

    for row = 1:2:rows
        if row == rows
            NewChrom((subpop-1)*rows + row, :) = SubChrom(rows, :);
        elseif rand < PR_CROSS
            if strcmp(CROSS_F, 'cross_alternating_edges')
                NewChrom((subpop-1)*rows + row,:) = cross_alternating_edges
                    ([ SubChrom(row,:) ; SubChrom(row+1,:) ]);
                NewChrom((subpop-1)*rows + row + 1,:) =
                    cross_alternating_edges([ SubChrom(row+1,:) ; SubChrom(
                    row,:) ]);
            else
                [ChildOne, ChildTwo] = feval(CROSS_F, SubChrom(row, :),
                    SubChrom(row+1, :));
                NewChrom((subpop-1)*rows + row, :) = ChildOne;
                NewChrom((subpop-1)*rows + row + 1, :) = ChildTwo;
            end
        else
            NewChrom((subpop-1)*rows + row, :) = SubChrom(row, :);
            NewChrom((subpop-1)*rows + row + 1, :) = SubChrom(row+1, :);
        end
    end
end

end
```

Listing 4.4: Inversion mutation for task 4 - src/mut_inversion2.m

```
% low level function for TSP mutation

function NewChrom = mut_inversion2(OldChrom)

NewChrom = adj2path(OldChrom);

```

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

% reverse a subpath in the chrom
reversed_subpath = NewChrom(rndi(2) : -1 : rndi(1));

tmp = [ NewChrom(1:rndi(1)-1) NewChrom(rndi(2)+1:size(NewChrom, 2)) ];
if (isempty(tmp))
    NewChrom = reversed_subpath;
else
    idx = randi(size(tmp, 2));
    NewChrom = [ tmp(1:idx) reversed_subpath tmp(idx+1:size(tmp, 2)) ];
end

NewChrom = path2adj(NewChrom);

end
```

Listing 4.5: High level mutation function - src/mutate_tsp.m

```
% MUTATE_TSP.M (Mutation for TSP high-level function)
%
% This function takes a matrix OldChrom containing the
% representation of the individuals in the current population,
% mutates the individuals and returns the resulting population.
%
% Input parameters:
% MUT_F - String containing the name of the mutation function
% OldChrom - Matrix containing the chromosomes of the old
% population. Each line corresponds to one individual.
% Representation - The TSP representation the given population is in.
% PR_MUT - (optional) Scalar containing the probability of
% mutation. If omitted, 0.05 is assumed.
% SUBPOP - (optional) Number of subpopulations.
% if omitted or NaN, 1 subpopulation is assumed
%
% Output parameter:
% NewChrom - Matrix containing the chromosomes of the population
% after mutation in the same format as OldChrom.


function NewChrom = mutate_tsp(MUT_F, OldChrom, PR_MUT, SUBPOP)

% Check parameter consistency
if nargin < 2; error('Not enough input parameters'); end

% Probability of mutation
if nargin < 3; PR_MUT = 0.05; end

% Population size
```

```
[rows, cols] = size(OldChrom);
NewChrom = zeros(rows, cols);

if nargin < 4; SUBPOP = 1;
elseif nargin > 3
  if isempty(SUBPOP), SUBPOP = 1;
  elseif isnan(SUBPOP), SUBPOP = 1;
  elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar');
  end
end

if (rows/SUBPOP) ~= fix(rows/SUBPOP)
    error('OldChrom and SUBPOP disagree');
end

rows = rows/SUBPOP; % Compute number of individuals per subpopulation

% Select individuals of subpopulations and call low level function
for subpop = 1:SUBPOP
    SubChrom = OldChrom((subpop-1)*rows+1:subpop*rows,:);

    for row = 1:rows
        if rand < PR_MUT
            NewChrom((subpop-1)*rows + row, :) = feval(MUT_F, SubChrom(row
                , :));
        else
            NewChrom((subpop-1)*rows + row, :) = SubChrom(row, :);
        end
    end
end

end
```

Listing 4.6: Fitness proportional selection for task 7a - src/sel_fit_prop.m

```
% SEL_FIT_PROP.m (FITNESS PROPORTIONAL SELECTION)
%
% This function performs fitness proportional selection.
%
% Syntax: NewChrIx = fitpropsel(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 parameter:
% 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 = sel_fit_prop(FitnV, NSel)

```

```
NewChrIx = zeros(NSel, 1);

fitSum = sum(FitnV);
[selProp, I] = sort(FitnV / fitSum);

for i = 1:NSel
    s = find(cumsum(selProp) >= rand, 1, 'first');
    NewChrIx(i) = I(s);
end

end
```

Listing 4.7: Tournament selection for task 7a - src/sel_tournament.m

```
% SEL_TOURNAMENT.m (TOURNAMENT SELECTION)
%
% This function performs tournament selection.
%
% Input parameters:
% FitnV - Column vector containing the fitness values of the
% individuals in the population.
% Nsel - number of individuals to be selected
%
% Output parameter:
% 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 = sel_tournament(FitnV, NSel)

NewChrIx = zeros(NSel, 1);

[NInd, ~] = size(FitnV);

k = max(2, floor(0.05*NInd));

for i = 1:NSel
    randIdx = randi(NInd, [k, 1]);
    [~, I] = max(FitnV(randIdx));
    NewChrIx(i) = randIdx(I);
end

end
```

Listing 4.8: Round robin tournament survival selection for task 7b - src/sur_sel_rr_tournament.m

```
% SUR_SEL_RR_TOURNAMENT.M (ROUND ROBIN TOURNAMENT SURVIVOR SELECTION)
%
% Reinserts offspring in the population by round-robin tournament
% survivor selection.
%
% Input parameters:
```

```
% Chrom - Matrix containing the individuals (parents) of the current
% population. Each row corresponds to one individual.
% SelCh - Matrix containing the offspring of the current population.
% Each row corresponds to one individual.
% ObjVCh - Column vector containing the objective values of the
% individuals (parents - Chrom) in the current population,
% needed for fitness-based insertion saves recalculation of
% objective values for population
% ObjVSel - Column vector containing the objective values of the
% offspring (SelCh) in the current population, needed for
% partial insertion of offspring, saves recalculation of
% objective values for population
% q - The amount of other individuals that each individual is
% to be evaluated against
%
% Output parameters:
% Chrom - Matrix containing the individuals of the current
% population after reinsertion.
% ObjVCh - if ObjVCh and ObjVSel are input parameter, than column
% vector containing the objective values of the individuals
% of the current generation after reinsertion.

function [Chrom, ObjVCh] = sur_sel_rr_tournament(Chrom, SelCh, ObjVCh,
    ObjVSel, q)

pop = [Chrom; SelCh];
popFit = [ObjVCh; ObjVSel];
[Npop, ~] = size(pop);
[NIND, ~] = size(Chrom);
wins = zeros(Npop, 1);

for i = 1:Npop
    wins(i) = sum(popFit(i) >= popFit(randi(Npop, [q 1])));
end

[~, I] = sort(wins);

Chrom = pop(I(1:NIND), :);
ObjVCh = popFit(I(1:NIND));

end
```

Listing 4.9: Template function for testing the algorithm. Other testing functions are omitted because they are very similar to this one - src/test_template.m

```
NIND=100; % Number of individuals
MAXGEN=250; % Maximum no. of generations
ELITIST=0.05; % percentage of the elite population
STOP_PERCENTAGE=.95; % percentage of equal fitness individuals for
    stopping
PR_CROSS=.95; % probability of crossover
PR_MUT=.05; % probability of mutation
LOCALLOOP=1; % local loop removal
CROSSOVER = 'cross_alternating_edges'; % crossover operators
```

```
MUTATION = 'mut_inversion'; % mutation operators
SELECTION = 'sus'; % parent selection algorithm
SUBPOP = 1; % Amount of subpopulations
SCALING = 1; % City location scaling on/off
CUSTOMSTOP = 0; % Custom stopping criterion on/off
CUSTOMSS = 0; % Custom survivor selection on/off
RUNS = 1; % Number of ga runs in tests


datasetslist = dir('datasets/');
Ndatasets = size(datasetslist, 1) - 2;

results = zeros([Ndatasets 4]);

out = fopen('./table.tex', 'w');
fprintf(out, 'A & B & C & D & E\n\\midrule\n');

for ds = 1:Ndatasets
    datasetslist(ds + 2).name
    data = load(['datasets/' datasetslist(ds + 2).name]);

    x = data(:,1);
    y = data(:,2);

    if SCALING == 1
        x = x / max([data(:,1); data(:,2)]);
        y = y / max([data(:,1); data(:,2)]);
    end

    NVAR=size(data,1);

    for i = 0:RUNS
        [best, mean, worst] = run_ga(x, y, NIND, MAXGEN, NVAR, ELITIST,
            STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, MUTATION,
            LOCALLOOP, CUSTOMSTOP, CUSTOMSS, SELECTION, SUBPOP);
        Ngen = find(best, 1, 'last');
        B = best(Ngen);
        M = mean(Ngen);
        W = worst(Ngen);

        results(ds, :) = results(ds, :) + [Ngen B M W];
    end

    results(ds, :) = results(ds, :) / RUNS;

    fprintf(out, '%s & %d & %d & %d & %d \\\\\n', datasetslist(ds + 2).
        name, results(ds, 1) - 1, results(ds, 2), results(ds, 3), results(
        ds, 4));

end

fclose(out);

```

```
58 results
```