Katholieke
Universiteit
Leuven

**Department of
Computer Science**

PROJECT

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. For each task, experiments were ran 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 tables 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 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 options 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.

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.

1.1.2 Generations

TODO: COMMENTS

1.1.3 Elitism

TODO: COMMENTS

1.1.4 Crossover

TODO: COMMENTS

1.1.5 Mutation

TODO: COMMENTS

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 **??**. The parameters were:

1.2 Task 3: Stopping criterion

To implement a new stopping criterion, we looked at the commonly used termination conditions outlined by the book. 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.

The fourth criterion is also already present in the template and can be adjusted via the GUI. 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 a percentage of the specified maximum number of generations. When testing this termination condition, we see that it does succeed in avoiding computation of useless generations where the fitness does not improve for a long time. Because of the fact that improvements may still happen at a later point in time, the score will be slightly worse with this new condition.

The results of our experiments with this new termination condition are displayed in Table 1.1.

	Default stopping criterion				Custom stopping criterion			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	60.6000	3.8289	3.8405	4.3167	51.9000	3.8598	4.0645	4.9485
rondrit018.tsp	59.9000	3.6123	3.6526	4.3262	62.4000	3.5116	3.7063	4.6899
rondrit023.tsp	92.8000	3.9902	4.2086	5.2684	77.5000	4.3064	4.4586	5.5427
rondrit025.tsp	82.0000	5.3085	5.4600	6.4753	79.2000	5.3707	5.7207	7.3034
rondrit048.tsp	108.7000	7.8736	8.2540	9.5729	109.2000	8.1605	8.9541	10.9159
rondrit050.tsp	101.5000	12.3558	12.9267	14.6198	104.0000	12.0748	12.8018	14.4246
rondrit051.tsp	109.6000	11.8400	12.3779	13.6785	108.9000	11.7734	12.2508	13.6669
rondrit067.tsp	110.0000	11.5868	12.2748	13.8317	107.2000	11.4511	12.4336	14.0862
rondrit070.tsp	110.0000	17.8292	18.6919	20.5209	109.6000	18.3206	18.9595	20.7340
rondrit100.tsp	110.0000	29.6034	31.6596	34.6374	108.7000	29.1263	30.6127	33.1294
rondrit127.tsp	110.0000	19.6930	20.5928	21.9205	110.0000	19.0723	20.0230	21.4847

Table 1.1: 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').

Table 1.2 contains the results of experiments with different crossover operators. The 'cross_alternate_edges' function implements Alternating Edge Crossover and is provided in the template. It is clear that Order Crossover performs significantly better than Alternating Edge Crossover; all of the path lengths with Order Crossover are lower for every dataset.

	Alternating Edge Crossover				Order Crossover			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	182.4	3.39	3.55	4.32	50.6	3.44	3.46	4.08
rondrit018.tsp	244.5	2.98	4.53	6.46	55.4	3.05	3.06	3.87
rondrit023.tsp	250.0	3.90	6.59	9.43	79.0	3.57	3.59	4.45
rondrit025.tsp	250.0	5.03	8.64	11.94	82.4	4.48	4.51	5.75
rondrit048.tsp	250.0	9.68	14.45	18.81	188.4	5.49	5.54	6.67
rondrit050.tsp	250.0	13.99	19.79	24.57	162.4	8.20	8.23	9.53
rondrit051.tsp	250.0	13.08	18.29	22.54	155.9	8.67	8.69	9.61
rondrit067.tsp	250.0	13.95	18.41	22.28	188.9	7.11	7.14	7.92
rondrit070.tsp	250.0	21.89	28.98	34.40	219.6	10.94	11.06	12.49
rondrit100.tsp	250.0	34.40	43.21	50.05	232.6	17.78	18.22	19.91
rondrit127.tsp	250.0	21.93	26.30	29.49	239.8	12.77	13.03	14.02

Table 1.2: Results of different crossover functions.

Table 1.3 contains the results of experiments with different mutation operators. The 'mut_inversion' function implements Simple Inversion Mutation and is provided in the template. There is no clear difference between the two methods in these results.

	Simple Inversion Mutation				Inversion Mutation			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	168.1	3.39	3.54	4.48	180.5	3.41	3.58	4.57
rondrit018.tsp	237.4	3.06	4.55	6.44	245.2	3.04	4.58	6.36
rondrit023.tsp	250.0	3.91	6.54	9.14	250.0	4.07	6.86	9.53
rondrit025.tsp	250.0	5.05	8.78	12.26	250.0	5.01	8.77	12.53
rondrit048.tsp	250.0	9.34	14.57	19.28	250.0	9.65	14.59	19.29
rondrit050.tsp	250.0	13.98	19.60	24.39	250.0	13.57	19.43	24.01
rondrit051.tsp	250.0	13.49	18.57	22.72	250.0	13.56	18.58	22.65
rondrit067.tsp	250.0	13.95	18.73	22.73	250.0	13.63	18.56	22.24
rondrit070.tsp	250.0	21.58	28.43	34.15	250.0	21.50	28.89	34.42
rondrit100.tsp	250.0	35.01	43.47	50.80	250.0	34.13	42.89	49.85
rondrit127.tsp	250.0	22.00	26.48	29.43	250.0	21.90	26.30	29.70

Table 1.3: Results of different mutation functions.

Table 1.4 shows results after some parameter tuning with the 'cross_order' and 'mut_inversion2' operators. The parameters used were: a = b, c = d, ... TODO check tuning

Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	43.3	3.42	3.44	4.25
rondrit018.tsp	51.7	3.13	3.16	4.29
rondrit023.tsp	70.2	3.68	3.71	4.65
rondrit025.tsp	72.5	4.43	4.48	6.07
rondrit048.tsp	134.6	6.12	6.15	7.51
rondrit050.tsp	160.2	8.68	8.71	10.28
rondrit051.tsp	144.0	8.92	8.96	10.46
rondrit067.tsp	168.9	7.99	8.01	9.03
rondrit070.tsp	184.4	12.25	12.28	13.56
rondrit100.tsp	230.1	18.31	18.59	20.26
rondrit127.tsp	249.2	12.84	13.23	14.30

Table 1.4: Results with 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 up to path length 3. With default values for other parameters, this results in major improvements to the score.

The results of our experiments with local optimisation disabled and enabled are displayed in Table 1.5.

	Local optimisation disabled				Local optimisation enabled			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	108.8	3.9087	5.3409	6.6707	63.4	3.6923	3.7783	4.4343
rondrit018.tsp	109.0	3.8205	5.9391	7.9101	82.7	3.2285	3.8159	4.8886
rondrit023.tsp	109.0	5.2503	8.0243	10.4586	109.0	3.6688	6.0516	8.2630
rondrit025.tsp	109.0	6.8383	10.5710	13.9566	109.0	4.6353	7.9255	11.2084
rondrit048.tsp	109.0	12.9322	17.3083	21.2216	109.0	7.4032	12.2117	15.8290
rondrit050.tsp	109.0	17.5460	23.0187	27.4702	109.0	10.6862	16.3413	19.9854
rondrit051.tsp	109.0	17.0520	21.6487	25.5342	109.0	9.8788	15.0773	19.2524
rondrit067.tsp	109.0	17.2930	21.6320	25.0256	109.0	9.4921	14.8637	18.2992
rondrit070.tsp	109.0	27.0916	33.4376	38.6964	109.0	14.9605	22.7446	27.9257
rondrit100.tsp	109.0	41.7305	49.6807	55.5533	109.0	22.4411	32.7396	38.8860
rondrit127.tsp	109.0	26.5206	30.3975	33.3563	109.0	15.3001	20.8402	24.3830

Table 1.5: 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.

	Stochastic Universal Sampling			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	109.0	3.9268	5.5816	6.95230040943754
rondrit018.tsp	109.0	3.8098	5.8874	7.64505139464696
rondrit023.tsp	109.0	5.2107	8.1510	10.66363338096259
rondrit025.tsp	109.0	6.9584	10.6573	13.64919455732690
rondrit048.tsp	109.0	12.5354	17.1587	20.98354804718375
rondrit050.tsp	109.0	17.6555	23.3466	27.70294938389831
rondrit051.tsp	109.0	16.8619	21.5905	24.99887566024666
rondrit067.tsp	109.0	17.3626	21.8921	25.01576819902206
rondrit070.tsp	109.0	27.3491	33.7980	39.02844367980951
rondrit100.tsp	109.0	42.5563	50.3468	56.14968388227629
rondrit127.tsp	109.0	26.3171	30.1653	32.82588637833199

Table 1.6: Results when using Stochastic Universal Sampling as parent selection method.

	Tournament Selection			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	109.0	3.8584	5.2429	6.65136434345961
rondrit018.tsp	109.0	3.8044	5.8175	7.60837957298278
rondrit023.tsp	109.0	5.0251	7.8150	10.25462477584426
rondrit025.tsp	109.0	6.8520	10.6018	13.97083528139636
rondrit048.tsp	109.0	12.3467	16.3804	20.33995051462599
rondrit050.tsp	109.0	17.7767	23.0010	26.94691760670842
rondrit051.tsp	109.0	16.6423	21.4503	24.88148805771897
rondrit067.tsp	109.0	17.4205	21.8030	24.91090949012749
rondrit070.tsp	109.0	27.0049	33.5066	38.57674586236878
rondrit100.tsp	109.0	42.1082	49.7688	55.34138052048054
rondrit127.tsp	109.0	26.5709	30.3721	33.34923179147698

Table 1.7: Results when using Tournament Selection as parent selection method.

	Fitness Proportional Selection			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	109.0	3.9268	5.5816	6.95230040943754
rondrit018.tsp	109.0	3.8098	5.8874	7.64505139464696
rondrit023.tsp	109.0	5.2107	8.1510	10.66363338096259
rondrit025.tsp	109.0	6.9584	10.6573	13.64919455732690
rondrit048.tsp	109.0	12.5354	17.1587	20.98354804718375
rondrit050.tsp	109.0	17.6555	23.3466	27.70294938389831
rondrit051.tsp	109.0	16.8619	21.5905	24.99887566024666
rondrit067.tsp	109.0	17.3626	21.8921	25.01576819902206
rondrit070.tsp	109.0	27.3491	33.7980	39.02844367980951
rondrit100.tsp	109.0	42.5563	50.3468	56.14968388227629
rondrit127.tsp	109.0	26.3171	30.1653	32.82588637833199

Table 1.8: 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 values of the ELITISM parameter can be found in table 4.17. To evaluate how round robin tournament performs compared to the already implemented elitism, the ELITIST parameter is set to 0 in an experiment. Also, an experiment is done where elitism is combined with round robin tournament. Table 1.9 contains the results of the experiments for this task. TODO: evaluate results after verifying they are correct. The custom stopping criterion needs to be updated first. Last X values should all be equal, instead of just current and current-X.

	Already implemented elitism				Round robin tournament			
Dataset	# Generations	Min	Mean	Max	# Generations	Min	Mean	Max
rondrit016.tsp	91.3	3.46	4.66	6.19	0.0	0.00	0.00	0.00
rondrit018.tsp	83.9	3.40	5.20	7.09	0.0	0.00	0.00	0.00
rondrit023.tsp	84.3	4.50	6.91	9.45	0.0	0.00	0.00	0.00
rondrit025.tsp	102.9	5.79	9.11	12.60	0.0	0.00	0.00	0.00
rondrit048.tsp	101.0	10.56	15.16	19.12	0.0	0.00	0.00	0.00
rondrit050.tsp	137.6	14.68	19.80	24.21	0.0	0.00	0.00	0.00
rondrit051.tsp	111.3	14.70	19.24	22.88	0.0	0.00	0.00	0.00
rondrit067.tsp	90.4	15.17	19.11	22.88	0.0	0.00	0.00	0.00
rondrit070.tsp	123.4	23.12	29.61	35.04	0.0	0.00	0.00	0.00
rondrit100.tsp	106.5	36.29	43.94	50.20	0.0	0.00	0.00	0.00
rondrit127.tsp	113.6	23.02	26.88	30.19	0.0	0.00	0.00	0.00

Table 1.9: Results for the already implemented elitism and 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. The results displayed in Table 1.10 through Table **??** show tests performed with 1, 2, 5, 10 and 20 subpopulations or islands.

	# subpopulations = 1			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	99.9	3.7370	5.3014	7.3443
rondrit018.tsp	120.7	3.4504	5.5736	7.9594
rondrit023.tsp	124.4	4.6149	7.6039	10.7075
rondrit025.tsp	134.5	5.8402	9.9015	13.9843
rondrit048.tsp	129.7	10.9175	16.1203	21.3953
rondrit050.tsp	194.6	14.7423	21.4762	26.9975
rondrit051.tsp	169.1	14.7132	20.3833	25.4901
rondrit067.tsp	139.0	15.3320	20.6747	25.3712
rondrit070.tsp	187.9	23.1837	31.6961	39.3717
rondrit100.tsp	174.5	38.1144	48.0221	55.8977
rondrit127.tsp	195.0	23.8434	29.0456	33.1707

Table 1.10: Results when using a single subpopulation.

	# subpopulations = 2			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	118.9	4.1112	5.6874	7.9026
rondrit018.tsp	147.5	3.8393	6.0843	8.5831
rondrit023.tsp	159.4	5.0312	8.3282	11.8303
rondrit025.tsp	152.9	6.6344	10.9143	15.5315
rondrit048.tsp	204.9	11.4129	17.6468	23.8143
rondrit050.tsp	216.0	16.5176	23.7049	29.4113
rondrit051.tsp	168.2	16.6110	22.5015	27.8662
rondrit067.tsp	203.0	16.4217	22.3292	27.4292
rondrit070.tsp	129.3	27.3049	35.4823	42.9792
rondrit100.tsp	251.9	40.3992	51.8574	60.8410
rondrit127.tsp	265.4	25.6029	31.5892	36.3884

Table 1.11: Results when using two subpopulations.

	# subpopulations = 5			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	126.6	4.1295	5.8593	7.9499
rondrit018.tsp	121.0	4.0043	6.2663	8.9579
rondrit023.tsp	138.2	5.1783	8.4623	12.0132
rondrit025.tsp	166.0	6.4178	11.0127	15.7281
rondrit048.tsp	117.9	12.4779	18.4335	24.2895
rondrit050.tsp	164.8	17.1913	24.2682	30.5998
rondrit051.tsp	166.4	17.3803	23.2727	28.6140
rondrit067.tsp	225.4	16.5562	22.6888	27.6837
rondrit070.tsp	183.2	26.8295	35.8003	42.9375
rondrit100.tsp	199.2	41.8170	53.1547	62.3695
rondrit127.tsp	218.5	26.4587	32.1475	36.5978

Table 1.12: Results when using five subpopulations.

	# subpopulations = 10			
Dataset	# Generations	Min	Mean	Max
rondrit016.tsp	97.4	4.1405	5.5019	7.7501
rondrit018.tsp	140.1	3.8233	5.5875	8.1174
rondrit023.tsp	180.4	4.7633	7.8333	11.6950
rondrit025.tsp	186.0	6.1102	10.4032	15.4008
rondrit048.tsp	276.3	10.6240	17.0398	23.1415
rondrit050.tsp	245.8	15.9304	22.8300	29.9988
rondrit051.tsp	190.7	16.0860	21.8967	27.5143
rondrit067.tsp	254.4	15.8431	21.9523	27.1892
rondrit070.tsp	272.4	24.4098	33.8360	41.4091
rondrit100.tsp	250.7	39.4631	51.1833	61.1145
rondrit127.tsp	282.1	25.3591	31.1691	36.3321

Table 1.13: Results when using ten subpopulations.

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 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 and methods used were as follows:

- TODO

The results are shown in table 1.14. Judging from the relative errors, we can conclude that our solutions are TODO

Dataset	Optimal length	# Generations	Min	Mean	Max	Error Min	Error Mean	Error Max
bcl380.tsp	1621	250.0	10119.10	14190.44	16270.37	524.25%	775.41%	1003.66%
belgiumtour.tsp	1	250.0	884.34	1425.84	2048.85	88333.65%	142483.67%	204785.29%
rbx711.tsp	3115	250.0	27069.22	34898.93	39431.08	769.00%	1020.35%	1265.81%
xqf131.tsp	564	250.0	1752.34	2528.96	3232.48	210.70%	348.40%	572.96%
xql662.tsp	2513	250.0	21848.41	29270.98	33471.24	769.42%	1064.78%	1331.88%

Table 1.14: Results for benchmark problems with our final algorithm.

Appendix

4.1 Tables

4.1.1 Task 2

Dataset	# Generations	Min	Mean	Max
number of individuals = 50				
rondrit016.tsp	257.2	3.79	4.98	6.35
rondrit048.tsp	275.0	12.05	17.20	21.45
rondrit067.tsp	275.0	16.44	21.32	24.98
rondrit127.tsp	275.0	25.40	29.92	33.49
number of individuals = 100				
rondrit016.tsp	204.8	3.71	4.04	5.06
rondrit048.tsp	275.0	10.62	15.99	20.74
rondrit067.tsp	275.0	14.92	20.27	24.63
rondrit127.tsp	275.0	24.24	28.98	32.68
number of individuals = 250				
rondrit016.tsp	250.4	3.69	4.99	6.79
rondrit048.tsp	275.0	9.65	15.87	21.31
rondrit067.tsp	275.0	14.19	20.33	25.16
rondrit127.tsp	275.0	22.88	28.67	33.30
number of individuals = 500				
rondrit016.tsp	269.7	3.68	5.02	7.07
rondrit048.tsp	275.0	8.70	15.66	21.39
rondrit067.tsp	275.0	12.71	19.60	25.26
rondrit127.tsp	275.0	21.89	28.25	32.98
number of individuals = 1000				
rondrit016.tsp	262.0	3.69	4.92	6.95
rondrit048.tsp	275.0	8.44	15.49	21.66
rondrit067.tsp	275.0	12.00	19.52	25.46
rondrit127.tsp	275.0	21.23	27.93	33.13

Table 4.15: Existing genetic algorithm with varying amount of individuals.

Dataset	# Generations	Min	Mean	Max
max number of generations = 100				
rondrit016.tsp	110.0	3.96	5.59	7.16
rondrit048.tsp	110.0	12.55	17.37	21.65
rondrit067.tsp	110.0	17.23	21.73	25.31
rondrit127.tsp	110.0	26.64	30.43	33.49
max number of generations = 250				
rondrit016.tsp	266.0	3.78	5.11	6.40
rondrit048.tsp	275.0	11.76	17.25	21.25
rondrit067.tsp	275.0	16.49	21.51	24.93
rondrit127.tsp	275.0	25.72	29.78	32.79
max number of generations = 500				
rondrit016.tsp	455.4	3.74	4.75	5.79
rondrit048.tsp	550.0	11.36	16.91	20.90
rondrit067.tsp	550.0	15.73	21.11	24.53
rondrit127.tsp	550.0	24.90	29.81	32.72
max number of generations = 1000				
rondrit016.tsp	785.0	3.75	4.54	5.49
rondrit048.tsp	1100.0	10.44	16.51	20.78
rondrit067.tsp	1100.0	14.92	20.82	24.94
rondrit127.tsp	1100.0	23.76	29.38	32.65

Table 4.16: Existing genetic algorithm with varying amount of maximum generations.

Dataset	# Generations	Min	Mean	Max
percentage of the elite population = 0.00				
rondrit016.tsp	275.0	5.04	6.26	7.63
rondrit048.tsp	275.0	15.64	19.03	22.63
rondrit067.tsp	275.0	20.45	23.39	26.17
rondrit127.tsp	275.0	29.51	31.75	34.26
percentage of the elite population = 0.05				
rondrit016.tsp	269.5	3.77	5.08	6.73
rondrit048.tsp	275.0	11.55	16.92	21.22
rondrit067.tsp	275.0	16.37	21.52	25.24
rondrit127.tsp	275.0	25.54	30.00	32.93
percentage of the elite population = 0.10				
rondrit016.tsp	60.2	3.90	3.91	4.55
rondrit048.tsp	250.7	8.29	9.14	11.35
rondrit067.tsp	270.5	12.25	14.56	18.86
rondrit127.tsp	256.3	21.44	23.87	27.30
percentage of the elite population = 0.30				
rondrit016.tsp	49.7	3.90	3.92	4.40
rondrit048.tsp	201.5	7.90	8.14	9.68
rondrit067.tsp	259.2	11.16	11.81	14.16
rondrit127.tsp	241.0	20.46	21.16	23.62
percentage of the elite population = 0.50				
rondrit016.tsp	59.0	3.87	3.88	4.34
rondrit048.tsp	254.7	8.12	8.35	10.08
rondrit067.tsp	275.0	11.57	12.10	14.30
rondrit127.tsp	228.9	21.65	22.03	23.64
percentage of the elite population = 0.70				
rondrit016.tsp	91.1	3.92	3.92	4.12
rondrit048.tsp	262.1	8.97	9.31	11.04
rondrit067.tsp	250.1	13.05	13.45	15.26
rondrit127.tsp	275.0	21.93	22.55	24.69
percentage of the elite population = 0.95				
rondrit016.tsp	262.1	4.10	4.23	4.71
rondrit048.tsp	257.6	12.80	13.41	14.57
rondrit067.tsp	275.0	17.23	18.00	19.44
rondrit127.tsp	274.8	26.61	27.18	27.80

Table 4.17: Existing genetic algorithm with varying percentage of the elite population.

Dataset	# Generations	Min	Mean	Max
probability of crossover = 0.00				
rondrit016.tsp	31.5	5.14	5.15	5.47
rondrit048.tsp	83.4	13.74	13.75	14.18
rondrit067.tsp	90.9	18.62	18.63	18.92
rondrit127.tsp	141.2	26.71	26.71	26.90
probability of crossover = 0.10				
rondrit016.tsp	37.0	4.41	4.42	4.83
rondrit048.tsp	120.0	10.21	10.23	10.79
rondrit067.tsp	151.2	14.42	14.43	14.94
rondrit127.tsp	199.4	22.26	22.30	22.58
probability of crossover = 0.30				
rondrit016.tsp	40.9	4.16	4.17	4.67
rondrit048.tsp	163.6	8.41	8.44	9.43
rondrit067.tsp	207.7	11.44	11.60	12.54
rondrit127.tsp	259.8	18.98	19.29	20.44
probability of crossover = 0.50				
rondrit016.tsp	46.8	4.07	4.07	4.44
rondrit048.tsp	171.2	8.09	8.15	9.11
rondrit067.tsp	246.2	10.36	10.57	11.99
rondrit127.tsp	267.9	18.47	18.87	20.62
probability of crossover = 0.70				
rondrit016.tsp	53.2	3.86	3.87	4.27
rondrit048.tsp	262.3	7.83	8.73	11.27
rondrit067.tsp	275.0	11.87	13.67	17.33
rondrit127.tsp	267.4	20.65	22.54	25.76
probability of crossover = 0.95				
rondrit016.tsp	275.0	3.82	5.45	6.99
rondrit048.tsp	275.0	12.01	17.23	21.33
rondrit067.tsp	275.0	16.50	21.51	25.12
rondrit127.tsp	275.0	25.45	30.02	33.20

Table 4.18: Existing genetic algorithm with varying probability of crossover.

Dataset	# Generations	Min	Mean	Max
probability of mutation = 0.00				
rondrit016.tsp	250.4	3.78	5.04	6.39
rondrit048.tsp	275.0	11.94	16.97	21.15
rondrit067.tsp	275.0	16.57	21.50	24.87
rondrit127.tsp	275.0	25.79	30.13	33.28
probability of mutation = 0.05				
rondrit016.tsp	248.6	3.74	5.13	6.34
rondrit048.tsp	275.0	11.58	17.10	21.07
rondrit067.tsp	275.0	16.55	21.47	25.15
rondrit127.tsp	275.0	25.69	30.07	33.22
probability of mutation = 0.10				
rondrit016.tsp	275.0	3.78	5.41	6.94
rondrit048.tsp	275.0	11.91	16.94	20.74
rondrit067.tsp	275.0	16.34	21.52	24.83
rondrit127.tsp	275.0	25.56	30.13	33.08
probability of mutation = 0.30				
rondrit016.tsp	275.0	3.92	5.71	7.34
rondrit048.tsp	275.0	11.48	17.30	21.32
rondrit067.tsp	275.0	15.89	21.31	25.08
rondrit127.tsp	275.0	25.10	29.95	32.99
probability of mutation = 0.50				
rondrit016.tsp	275.0	3.87	5.90	7.47
rondrit048.tsp	275.0	11.61	17.31	21.55
rondrit067.tsp	275.0	15.80	21.55	25.25
rondrit127.tsp	275.0	24.48	29.71	32.90
probability of mutation = 0.70				
rondrit016.tsp	275.0	3.86	6.01	7.59
rondrit048.tsp	275.0	11.03	17.27	21.42
rondrit067.tsp	275.0	15.59	21.50	25.28
rondrit127.tsp	275.0	25.01	30.03	32.92
probability of mutation = 0.95				
rondrit016.tsp	275.0	3.95	6.12	7.67
rondrit048.tsp	275.0	11.50	17.51	21.43
rondrit067.tsp	275.0	15.84	21.66	25.35
rondrit127.tsp	275.0	24.53	29.79	33.13

Table 4.19: Existing genetic algorithm with varying probability of mutation.

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);

    % number of individuals of equal fitness needed to stop
    stopN = ceil(STOP_PERCENTAGE*NIND);

    gen = 0;
    % generational loop
    while gen < MAXGEN
        sObjV = sort(ObjV);
        best(gen+1) = min(ObjV);
```

```
        minimum = best(gen+1);
        mean_fits(gen+1) = mean(ObjV);
        worst(gen+1) = max(ObjV);
        for t = 1:size(ObjV,1)
            if (ObjV(t) == minimum)
                break;
            end
        end

        if nargin == 19
            visualizeTSP(x, y, adj2path(Chrom(t,:)), minimum, ah1, gen,
                best, mean_fits, worst, ah2, ObjV, NIND, ah3);
        end

        % stopping criterion: stop when the minimum of the last stopN
        % generations has not improved
        if CUSTOMSTOP == 1
            if (gen-0.1*MAXGEN > 1) && ((best(floor(gen-0.1*MAXGEN)) -
                minimum) <= 1e-15)
                break;
            end
        else
            if (sObjV(stopN)-sObjV(1) <= 1e-15)
                break;
            end
        end

        % assign fitness values to entire population
        FitnV = ranking(ObjV);

        % select individuals for breeding
        SelCh = select(SELECTION, Chrom, FitnV, GGAP);

        %recombine individuals (crossover)
        SelCh = crossover_tsp(CROSSOVER, SelCh, PR_CROSS, SUBPOP);
        SelCh = mutate_tsp(MUTATION, SelCh, PR_MUT, SUBPOP);

        %evaluate offspring, call objective function
        ObjVSel = tspfun(SelCh, Dist);

        %reinsert offspring into population
        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)
```

```
    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
            % TODO: adapt crossover functions so that feval can be used
                with all of them
            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
```

```matlab
% 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

```matlab
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);

results
```