



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	2
Tasks	3
1.1 Task 2: Initial experiments	3
1.1.1 Individuals	3
1.1.2 Generations	3
1.1.3 Elitism	4
1.1.4 Crossover	4
1.1.5 Mutation	4
1.1.6 Loop removal	4
1.1.7 Mix	4
1.2 Task 3: Stopping criterion	4
1.3 Task 4: Other representation	7
1.4 Task 5: Local optimisation	8
1.5 Task 7: Optional tasks	10
1.5.1 7a: Parent selection	10
1.5.2 7b: Survivor selection	11
1.5.3 7c: Diversity preservation	11
1.6 Task 6: Benchmark problems	13
Appendix	15
4.1 Tables	15
4.1.1 Task 2	15
4.2 Code	19

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. The custom stopping criterion was not used during these experiments, as it is discussed in section 1.2.

1.1.1 Individuals

The minimum 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.

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.

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.

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: TODO

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.

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.

Dataset	Default stopping criterion				Custom stopping criterion			
	# 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. Also, the path lengths converted to a value earlier with Order Crossover, which is why the numbers of generations are lower. This can be attributed to ... ???

Dataset	Alternating Edge Crossover				Order Crossover			
	# 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 'mut_inversion' function 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.

Dataset	Simple Inversion Mutation				Inversion Mutation			
	# 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. 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 1.6.

Dataset	Local optimisation disabled				Local optimisation enabled			
	# 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.

Dataset	Stochastic Universal Sampling			
	# Generations	Min	Mean	Max
rondrit016.tsp	76.5	3.5071	4.7287	6.29733152234251
rondrit018.tsp	112.0	3.2458	4.9674	6.91548211300174
rondrit023.tsp	89.8	4.3998	6.9255	9.64269815478443
rondrit025.tsp	81.6	5.8037	9.2203	12.46022806183646
rondrit048.tsp	115.7	10.5298	15.0273	18.93401510803271
rondrit050.tsp	74.1	15.5617	20.4526	24.58418303604441
rondrit051.tsp	101.3	14.7763	19.2497	23.21947103058057
rondrit067.tsp	98.2	14.7791	19.1254	22.65609716605926
rondrit070.tsp	117.4	22.9912	29.3627	34.87830719868566
rondrit100.tsp	91.4	37.0343	44.2892	51.06123059198230
rondrit127.tsp	120.9	22.9939	26.6807	30.08382590490890

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

Dataset	Tournament Selection			
	# Generations	Min	Mean	Max
rondrit016.tsp	47.6	3.4307	3.4489	4.11682588428370
rondrit018.tsp	69.8	3.0751	3.1000	3.94962726552862
rondrit023.tsp	97.2	3.4223	3.4448	4.36636737136705
rondrit025.tsp	107.1	4.2080	4.2289	5.22930916811214
rondrit048.tsp	220.2	5.2384	5.3288	6.70164030513469
rondrit050.tsp	228.5	7.4060	7.4469	8.74950293351244
rondrit051.tsp	223.4	7.7256	7.8091	9.00104215733518
rondrit067.tsp	246.7	6.5806	6.6540	7.79291635925291
rondrit070.tsp	250.0	10.4302	11.1550	14.05669608369397
rondrit100.tsp	241.3	18.2543	19.2141	22.21694990667605
rondrit127.tsp	250.0	13.2633	13.8539	15.84629676598155

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

Dataset	Fitness Proportional Selection			
	# 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 the appendix in table 4.18. 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 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. Therefore, it is better to set elitism to zero.

Dataset	Round robin tournament				Elitism + round robin tournament			
	# 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 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. In addition to the number of supopulations, the number of individuals was also increased from 100 to 315 in these

experiments. This way there is a reasonable amount of individuals in each subpopulation for each experiment.

The results displayed in Table 1.11 through Table ?? show tests performed with 1, 2, 5, 10 and 20 subpopulations or islands.

Dataset	# subpopulations = 1			
	# Generations	Min	Mean	Max
rondrit016.tsp	95.5	3.4074	4.7767	6.6380
rondrit018.tsp	115.2	3.1761	5.0546	7.2900
rondrit023.tsp	101.9	4.0955	6.9523	10.0988
rondrit025.tsp	99.4	5.5454	9.2036	13.2600
rondrit048.tsp	111.9	10.0535	15.0388	19.9039
rondrit050.tsp	142.2	13.7133	19.8064	25.4397
rondrit051.tsp	123.7	13.3033	18.5275	23.2007
rondrit067.tsp	149.8	13.1947	18.4811	22.9918
rondrit070.tsp	110.0	22.0561	29.2306	35.7109
rondrit100.tsp	139.6	33.9588	43.5204	51.1411
rondrit127.tsp	144.7	21.5389	26.3386	30.2041

Table 1.11: Results when using a single subpopulation.

Dataset	# subpopulations = 2			
	# Generations	Min	Mean	Max
rondrit016.tsp	86.6	3.3963	4.7597	6.5871
rondrit018.tsp	98.2	3.1664	5.0665	7.2148
rondrit023.tsp	137.1	3.9422	6.8060	10.1056
rondrit025.tsp	103.9	5.3210	9.2062	13.1897
rondrit048.tsp	103.0	10.0800	15.0969	19.9683
rondrit050.tsp	136.6	13.7321	19.8016	25.0207
rondrit051.tsp	141.3	13.3589	18.6720	23.2114
rondrit067.tsp	131.9	13.5762	18.7582	23.3741
rondrit070.tsp	99.9	21.7707	29.1058	35.0688
rondrit100.tsp	94.3	35.1090	43.9147	51.3187
rondrit127.tsp	138.8	21.6849	26.4038	30.1507

Table 1.12: Results when using two subpopulations.

Dataset	# subpopulations = 5			
	# Generations	Min	Mean	Max
rondrit016.tsp	92.5	3.3886	4.7337	6.4869
rondrit018.tsp	108.3	3.0714	4.9820	7.2249
rondrit023.tsp	120.3	4.1053	6.8448	9.9840
rondrit025.tsp	129.1	5.0991	8.9414	12.9755
rondrit048.tsp	151.9	9.4921	14.7340	19.6760
rondrit050.tsp	136.2	13.5967	19.7958	25.2649
rondrit051.tsp	99.2	13.6028	18.7847	23.4240
rondrit067.tsp	143.1	13.4945	18.6712	23.2590
rondrit070.tsp	142.7	21.0303	28.8698	35.5137
rondrit100.tsp	144.7	33.5632	42.8819	50.8222
rondrit127.tsp	152.2	21.6000	26.3773	30.1659

Table 1.13: Results when using five subpopulations.

Dataset	# subpopulations = 10			
	# 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 1.14: 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 1.15. 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 1.15: 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	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 4.16: 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 4.17: 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 4.18: 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 4.19: 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 4.20: Existing genetic algorithm with varying probability of mutation.

4.2 Code

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

```

1 % run_ga.m (RUN GENETIC ALGORITHM)
2 %
3 % Input parameters:
4 % x, y - coordinates of the cities

```

```

5 % NIND - number of individuals
6 % MAXGEN - maximal number of generations
7 % ELITIST - percentage of elite population
8 % STOP_PERCENTAGE - percentage of equal fitness (stop criterium)
9 % PR_CROSS - probability for crossover
10 % PR_MUT - probability for mutation
11 % Crossover - the crossover operator
12 % MUTATION - the mutation operator
13 % LOCALLOOP - local loop removal on/off
14 % CUSTOMSTOP - custom stopping criterion on/off
15 % CUSTOMSS - custom survivor selection on/off
16 % SELECTION - the parent selection function (sus, sel_tournament,
17 % sel_fit_prop, ...)
18 % ah1, ah2, ah3 - axes handles to visualise tsp
19 %
20 % Output parameters:
21 % best - vector of the best result of every iteration
22 % mean_fits - vector of the mean result of every iteration
23 % worst - vector of the worst result of every iteration
24
25 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)
26
27 GGAP = 1 - ELITIST;
28
29 best = zeros(1, MAXGEN);
30 mean_fits = zeros(1, MAXGEN+1);
31 worst = zeros(1, MAXGEN+1);
32
33 Dist = zeros(NVAR, NVAR);
34 for i = 1:size(x,1)
35     for j = 1:size(y,1)
36         Dist(i,j) = sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
37     end
38 end
39
40 % initialize population
41 Chrom = zeros(NIND, NVAR);
42 for row = 1:NIND
43     Chrom(row,:) = path2adj(randperm(NVAR));
44 end
45 % evaluate initial population
46 ObjV = tspfun(Chrom, Dist);
47
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,
67             best, mean_fits, worst, ah2, ObjV, NIND, ah3);
68     end
69
70     % stopping criterion
71     if CUSTOMSTOP == 1
72         % Stop if the minimum has not improved in the interval of the
73         % last 0.1*MAXGEN generations. The change in fitness is
74         % calculated by taking the sum of the differences between the
75         % interval and the current best fitness.
76         idx = floor(gen-0.1*MAXGEN);
77         if (idx > 1) && (sum(abs(best(idx:gen) - minimum)) <= 1e-15)
78             break;
79         end
80     elseif (sObjV(stopN)-sObjV(1) <= 1e-15)
81         % Stop if the difference between the best and the stopN'th
82         % best fitness value is very small.
83         break;
84     end
85
86     % assign fitness values to entire population
87     FitnV = ranking(ObjV);
88
89     % select individuals for breeding
90     SelCh = select(SELECTION, Chrom, FitnV, GGAP);
91
92     % recombine individuals (crossover)
93     SelCh = crossover_tsp(CROSSOVER, SelCh, PR_CROSS, SUBPOP);
94     SelCh = mutate_tsp(MUTATION, SelCh, PR_MUT, SUBPOP);
95
96     % evaluate offspring, call objective function
97     ObjVSel = tspfun(SelCh, Dist);
98
99     % reinsert offspring into population
100     if CUSTOMSS == 0
101         [Chrom, ObjV] = reins(Chrom, SelCh, SUBPOP, 1, ObjV, ObjVSel);
102     else
103         [Chrom, ObjV] = sur_sel_rr_tournament(Chrom, SelCh, ObjV,
104             ObjVSel, 10);
105     end
106
107     Chrom = tsp_improve_population(NIND, NVAR, Chrom, LOCALLOOP, Dist)
108     ;

```

```

106
107     gen = gen+1;
108 end
109 end

```

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

```

1  % CROSS_ORDER.M (ORDER CROSSOVER)
2  %
3  % Order crossover for TSP.
4  %
5  % Input parameters:
6  % ParentOne, ParentTwo - The TSP individuals to apply crossover on in a
7  % certain representation.
8  % Representation - The representation the given parents are in.
9  % If omitted, 2 (path) is assumed.
10 %
11 % Output parameters:
12 % ChildOne, ChildTwo - Chromosomes created by mating, ready to be
13 % mutated and/or evaluated, in the same format
14 % as OldChrom.
15
16 function [ChildOne, ChildTwo] = cross_order(ParentOne, ParentTwo)
17
18 ParentOne = adj2path(ParentOne);
19 ParentTwo = adj2path(ParentTwo);
20
21 [~, cols] = size(ParentOne);
22 ChildOne = zeros(1, cols);
23 ChildTwo = zeros(1, cols);
24
25 rnd = sort(randi(cols, [1, 2]));
26 a = rnd(1);
27 b = rnd(2);
28
29 ChildOne(a:b) = ParentOne(a:b);
30 ChildTwo(a:b) = ParentTwo(a:b);
31
32 childOneIdx = rem(b, cols) + 1;
33 childTwoIdx = rem(b, cols) + 1;
34 for i = 1:cols
35     current = rem(b + i-1, cols) + 1;
36
37     if (all(ChildTwo ~= ParentOne(current)))
38         ChildTwo(childTwoIdx) = ParentOne(current);
39         childTwoIdx = rem(childTwoIdx, cols) + 1;
40     end
41
42     if (all(ChildOne ~= ParentTwo(current)))
43         ChildOne(childOneIdx) = ParentTwo(current);
44         childOneIdx = rem(childOneIdx, cols) + 1;
45     end
46 end
47

```

```

48 ChildOne = path2adj(ChildOne);
49 ChildTwo = path2adj(ChildTwo);
50
51 end

```

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

```

1  % CROSSOVER_TSP.M (Crossover for TSP high-level function)
2  %
3  % This function performs recombination (crossover) between pairs of
4  % individuals and returns the new individuals after mating.
5  % The function handles multiple populations and calls a given low-level
6  % function for the actual recombination process.
7  %
8  % Input parameters:
9  % CROSS_F - String containing the name of the crossover function
10 % OldChrom - Matrix containing the chromosomes of the old
11 % population. Each line corresponds to one individual
12 % PR_CROSS - (optional) Scalar containing the probability of
13 % recombination/crossover occurring between pairs
14 % of individuals. If omitted or NaN, 0.95 is assumed.
15 % SUBPOP - (optional) Number of subpopulations.
16 % If omitted or NaN, 1 subpopulation is assumed.
17 %
18 % Output parameter:
19 % NewChrom - Matrix containing the chromosomes of the population
20 % after recombination in the same format as OldChrom.
21
22 function NewChrom = crossover_tsp(CROSS_F, OldChrom, PR_CROSS, SUBPOP)
23
24 % Check parameter consistency
25 if nargin < 2; error('Not enough input parameter'); end
26
27 % Probability of crossover
28 if nargin < 3; PR_CROSS = 0.95;
29 elseif nargin > 2
30     if isempty(PR_CROSS), PR_CROSS = 0.7;
31     elseif isnan(PR_CROSS), PR_CROSS = 0.7;
32     elseif length(PR_CROSS) ~= 1, error('PR_CROSS must be a scalar');
33     elseif (PR_CROSS < 0 | PR_CROSS > 1), error('PR_CROSS must be a scalar
34         in [0, 1]');
35     end
36 end
37
38 % Population size
39 [rows, cols] = size(OldChrom);
40 NewChrom = zeros(rows, cols);
41
42 if nargin < 4; SUBPOP = 1;
43 elseif nargin > 3
44     if isempty(SUBPOP), SUBPOP = 1;
45     elseif isnan(SUBPOP), SUBPOP = 1;
46     elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar');
47     end

```



```

47 end
48
49 if (rows/SUBPOP) ~= floor(rows/SUBPOP)
50     rows
51     SUBPOP
52     error('OldChrom and SUBPOP disagree');
53 end
54 rows = rows/SUBPOP; % Compute number of individuals per subpopulation
55
56 % Select individuals of subpopulations and call low level function
57 for subpop = 1:SUBPOP
58     SubChrom = OldChrom((subpop-1)*rows+1:subpop*rows, :);
59
60     for row = 1:2:rows
61         if row == rows
62             NewChrom((subpop-1)*rows + row, :) = SubChrom(rows, :);
63         elseif rand < PR_CROSS
64             % TODO: adapt crossover functions so that feval can be used
65             % with all of them
66             if strcmp(CROSS_F, 'cross_alternating_edges')
67                 NewChrom((subpop-1)*rows + row,:) = cross_alternating_edges
68                     ([ SubChrom(row,:) ; SubChrom(row+1,:) ]);
69                 NewChrom((subpop-1)*rows + row + 1,:) =
70                     cross_alternating_edges([ SubChrom(row+1,:) ; SubChrom(
71                         row,:) ]);
72             else
73                 [ChildOne, ChildTwo] = feval(CROSS_F, SubChrom(row, :),
74                     SubChrom(row+1, :));
75                 NewChrom((subpop-1)*rows + row, :) = ChildOne;
76                 NewChrom((subpop-1)*rows + row + 1, :) = ChildTwo;
77             end
78         else
79             NewChrom((subpop-1)*rows + row, :) = SubChrom(row, :);
80             NewChrom((subpop-1)*rows + row + 1, :) = SubChrom(row+1, :);
81         end
82     end
83 end
84 end

```

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

```

1 % low level function for TSP mutation
2
3 function NewChrom = mut_inversion2(OldChrom)
4
5 NewChrom = adj2path(OldChrom);
6
7 % select two positions in the tour
8 rndi = zeros(1,2);
9 while rndi(1) == rndi(2)
10     rndi = randi(size(NewChrom, 2), [1, 2]);
11 end
12 rndi = sort(rndi);

```

```

13
14 % reverse a subpath in the chrom
15 reversed_subpath = NewChrom(rndi(2) : -1 : rndi(1));
16
17 tmp = [ NewChrom(1:rndi(1)-1) NewChrom(rndi(2)+1:size(NewChrom, 2)) ];
18 if (isempty(tmp))
19     NewChrom = reversed_subpath;
20 else
21     idx = randi(size(tmp, 2));
22     NewChrom = [ tmp(1:idx) reversed_subpath tmp(idx+1:size(tmp, 2)) ];
23 end
24
25 NewChrom = path2adj(NewChrom);
26
27 end

```

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

```

1 % MUTATE_TSP.M (Mutation for TSP high-level function)
2 %
3 % This function takes a matrix OldChrom containing the
4 % representation of the individuals in the current population,
5 % mutates the individuals and returns the resulting population.
6 %
7 % Input parameters:
8 % MUT_F - String containing the name of the mutation function
9 % OldChrom - Matrix containing the chromosomes of the old
10 % population. Each line corresponds to one individual.
11 % Representation - The TSP representation the given population is in.
12 % PR_MUT - (optional) Scalar containing the probability of
13 % mutation. If omitted, 0.05 is assumed.
14 % SUBPOP - (optional) Number of subpopulations.
15 % if omitted or NaN, 1 subpopulation is assumed
16 %
17 % Output parameter:
18 % NewChrom - Matrix containing the chromosomes of the population
19 % after mutation in the same format as OldChrom.
20
21
22 function NewChrom = mutate_tsp(MUT_F, OldChrom, PR_MUT, SUBPOP)
23
24 % Check parameter consistency
25 if nargin < 2; error('Not enough input parameters'); end
26
27 % Probability of mutation
28 if nargin < 3; PR_MUT = 0.05; end
29
30 % Population size
31 [rows, cols] = size(OldChrom);
32 NewChrom = zeros(rows, cols);
33
34 if nargin < 4; SUBPOP = 1;
35 elseif nargin > 3
36     if isempty(SUBPOP), SUBPOP = 1;

```

```

37 elseif isnan(SUBPOP), SUBPOP = 1;
38 elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar');
39 end
40 end
41
42 if (rows/SUBPOP) ~= fix(rows/SUBPOP)
43     error('OldChrom and SUBPOP disagree');
44 end
45
46 rows = rows/SUBPOP; % Compute number of individuals per subpopulation
47
48 % Select individuals of subpopulations and call low level function
49 for subpop = 1:SUBPOP
50     SubChrom = OldChrom((subpop-1)*rows+1:subpop*rows,:);
51
52     for row = 1:rows
53         if rand < PR_MUT
54             NewChrom((subpop-1)*rows + row, :) = feval(MUT_F, SubChrom(row
55                 , :));
56         else
57             NewChrom((subpop-1)*rows + row, :) = SubChrom(row, :);
58         end
59     end
60 end
61 end

```

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

```

1 % SEL_FIT_PROP.m (FITNESS PROPORTIONAL SELECTION)
2 %
3 % This function performs fitness proportional selection.
4 %
5 % Syntax: NewChrIx = fitprosel(FitnV, NSel)
6 %
7 % Input parameters:
8 % FitnV - Column vector containing the fitness values of the
9 % individuals in the population.
10 % NSel - number of individuals to be selected
11 %
12 % Output parameter:
13 % NewChrIx - column vector containing the indexes of the selected
14 % individuals relative to the original population, shuffled.
15 % The new population, ready for mating, can be obtained
16 % by calculating OldChrom(NewChrIx,:).
17
18 function NewChrIx = sel_fit_prop(FitnV, NSel)
19
20 NewChrIx = zeros(NSel, 1);
21
22 fitSum = sum(FitnV);
23 [selProp, I] = sort(FitnV / fitSum);
24
25 for i = 1:NSel

```

```

26     s = find(cumsum(selProp) >= rand, 1, 'first');
27     NewChrIx(i) = I(s);
28 end
29
30 end

```

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

```

1 % SEL_TOURNAMENT.m (TOURNAMENT SELECTION)
2 %
3 % This function performs tournament selection.
4 %
5 % Input parameters:
6 % FitnV - Column vector containing the fitness values of the
7 % individuals in the population.
8 % Nsel - number of individuals to be selected
9 %
10 % Output parameter:
11 % NewChrIx - column vector containing the indexes of the selected
12 % individuals relative to the original population, shuffled.
13 % The new population, ready for mating, can be obtained
14 % by calculating OldChrom(NewChrIx,:).
15
16 function NewChrIx = sel_tournament(FitnV, Nsel)
17
18 NewChrIx = zeros(Nsel, 1);
19
20 [NInd, ~] = size(FitnV);
21
22 k = max(2, floor(0.05*NInd));
23
24 for i = 1:Nsel
25     randIdx = randi(NInd, [k, 1]);
26     [~, I] = max(FitnV(randIdx));
27     NewChrIx(i) = randIdx(I);
28 end
29
30 end

```

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

```

1 % SUR_SEL_RR_TOURNAMENT.M (ROUND ROBIN TOURNAMENT SURVIVOR SELECTION)
2 %
3 % Reinserts offspring in the population by round-robin tournament
4 % survivor selection.
5 %
6 % Input parameters:
7 % Chrom - Matrix containing the individuals (parents) of the current
8 % population. Each row corresponds to one individual.
9 % SelCh - Matrix containing the offspring of the current population.
10 % Each row corresponds to one individual.
11 % ObjVCh - Column vector containing the objective values of the
12 % individuals (parents - Chrom) in the current population,

```

```

13 % needed for fitness-based insertion saves recalculation of
14 % objective values for population
15 % ObjVSel - Column vector containing the objective values of the
16 % offspring (SelCh) in the current population, needed for
17 % partial insertion of offspring, saves recalculation of
18 % objective values for population
19 % q - The amount of other individuals that each individual is
20 % to be evaluated against
21 %
22 % Output parameters:
23 % Chrom - Matrix containing the individuals of the current
24 % population after reinsertion.
25 % ObjVCh - if ObjVCh and ObjVSel are input parameter, than column
26 % vector containing the objective values of the individuals
27 % of the current generation after reinsertion.
28
29 function [Chrom, ObjVCh] = sur_sel_rr_tournament(Chrom, SelCh, ObjVCh,
    ObjVSel, q)
30
31 pop = [Chrom; SelCh];
32 popFit = [ObjVCh; ObjVSel];
33 [Npop, ~] = size(pop);
34 [NIND, ~] = size(Chrom);
35 wins = zeros(Npop, 1);
36
37 for i = 1:Npop
38     wins(i) = sum(popFit(i) >= popFit(randi(Npop, [q 1])));
39 end
40
41 [~, I] = sort(wins);
42
43 Chrom = pop(I(1:NIND), :);
44 ObjVCh = popFit(I(1:NIND));
45
46 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

```

1 NIND=100; % Number of individuals
2 MAXGEN=250; % Maximum no. of generations
3 ELITIST=0.05; % percentage of the elite population
4 STOP_PERCENTAGE=.95; % percentage of equal fitness individuals for
    stopping
5 PR_CROSS=.95; % probability of crossover
6 PR_MUT=.05; % probability of mutation
7 LOCALLOOP=1; % local loop removal
8 CROSSOVER = 'cross_alternating_edges'; % crossover operators
9 MUTATION = 'mut_inversion'; % mutation operators
10 SELECTION = 'sus'; % parent selection algorithm
11 SUBPOP = 1; % Amount of subpopulations
12 SCALING = 1; % City location scaling on/off
13 CUSTOMSTOP = 0; % Custom stopping criterion on/off
14 CUSTOMSS = 0; % Custom survivor selection on/off

```

```

15 RUNS = 1; % Number of ga runs in tests
16
17
18 datasetslist = dir('datasets/');
19 Ndatasets = size(datasetslist, 1) - 2;
20
21 results = zeros([Ndatasets 4]);
22
23 out = fopen('./table.tex', 'w');
24 fprintf(out, 'A & B & C & D & E\n\\midrule\n');
25
26 for ds = 1:Ndatasets
27     datasetslist(ds + 2).name
28     data = load(['datasets/' datasetslist(ds + 2).name]);
29
30     x = data(:,1);
31     y = data(:,2);
32
33     if SCALING == 1
34         x = x / max([data(:,1); data(:,2)]);
35         y = y / max([data(:,1); data(:,2)]);
36     end
37
38     NVAR=size(data,1);
39
40     for i = 0:RUNS
41         [best, mean, worst] = run_ga(x, y, NIND, MAXGEN, NVAR, ELITIST,
42             STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, MUTATION,
43             LOCALLOOP, CUSTOMSTOP, CUSTOMSS, SELECTION, SUBPOP);
44         Ngen = find(best, 1, 'last');
45         B = best(Ngen);
46         M = mean(Ngen);
47         W = worst(Ngen);
48
49         results(ds, :) = results(ds, :) + [Ngen B M W];
50     end
51
52     results(ds, :) = results(ds, :) / RUNS;
53
54     fprintf(out, '%s & %d & %d & %d & %d \\\n', datasetslist(ds + 2).
55         name, results(ds, 1) - 1, results(ds, 2), results(ds, 3), results(
56             ds, 4));
57
58 end
59
60 fclose(out);
61
62 results

```