Genetic Algorithms Report

Daniel Loscos Barroso
<daniel.loscosbarroso@student.kuleuven.be>

August 10, 2019

Implementation Resume

Methodology

Before discussing my implementation of the GA I wanted to discuss my methodology. First I did the coding for all tasks and only after all the code was written I began the testing for this report. This ,made the most sense to me while organizing the work as I could allot times for testing after knowing how much time I had invested on the implementation. However, since this was an application with incremental steps, from an engineering perspective, the best methodology would have been a set of code int test iterations. This second methodology would have allowed to detect problems with the implementation and fix them in the next iteration.

Also, after seeing the results of task 6, I would probably had decided to implement task 7(d) instead of task 7(a) (later in the report we will see how the main problem I encountered during task 6 testing was stagnation of the algorithms and slow evolution in the later stages. The main fix to this is self adaptability in some parameters.).

For the testing I used a script `testing.m` that i overwrote with each test. A better practice for reproducibility of my results would have been to comment old tests instead of rewriting them. The main goal of this file was to allow me to run an specific configuration of the GA several times, store the results and to prepare batteries of tests that could run while I worked in other subjects. Using this script allowed me to make the GUI redundant and to only print graphs and stats after the GA had finished, this saves a lot of computation time and leads to faster testing.

Finally, the redaction of this report was made at the same time of testing, to help me make decisions on the next tests to perform and to explain my thought process in a more accurate manner.

Implementation

Stopping criteria

The code provided to as already had an early termination criterion based on the diversity of the population. Seeing this I decided to implement another one (and to allow the combination of the two) based on the number of iterations without improvement (stalling or stagnation). Instead of setting a number of iterations as a parameter to define how many iterations where allowed, it felt like a percentage of the maximum number of generations was a better approach.

It was at this stage that I also decided to allow resetting on early termination, a mechanic that later proved useful during testing, as it allows the GA to get out of local optima in a crude but effective manner.

Representation

I chose to work with the path representation and to implement most of the crossover and mutation operators covered during the course: order and edge recombination crossover with simple inversions, inversion, scramble, swap and insert mutations.

The reason I chose this representation was that it felt much more comfortable to code and work with. However, some problems appeared during testing, as most of this mutation operators have an impact based on the number of TSP nodes. a good solution to this would have been an 'impact of mutation' parameter setting the number of swaps or inversions to perform, or even coding this functions to take into account the size of the problem and tune the impact automatically.

Also, my implementation of the edge recombination crossover was too slow for it to ever be useful in a professional context.

Local optimization

Since loop detection was already implemented in the code we were provided with, I decided to use it and not implement any other strategy. Now, after the testing, I regret my decision, as I feel that loop detection more than proved its worth and that a hungry strategy that connected nodes to their nearest ones for the first iterations would have greatly improved the results of the tested benchmarks.

Selection methods

The option that I chose for task 7 was to implement different parent selection methods: the ones I implemented were fitness proportional and tournament selection.

Fitness proportional is slower than the already implemented SUS, but it represents better the populations and it fitness. To prevent high fitness values from disrupting this method's behavior the fitness values are re-scaled before selection. Regarding tournament selection I decided to implement two default tournament sizes: 3 and 5 individuals, so the user could adapt to its population size.

Again, a possibly better approach could have been to set the tournament size as a parameter or make it adaptive to the size of the population.

Final note

The rest of the techniques described in task 7 where not implemented due to time constraints. I believe that the two most effective amongst them would have been the introduction of self-adaptative parameters and diversity preserving techniques.

I argue this without any data to confirm my theory, but based on the lack of diversity showed by my Task 6 and 7 experiments.

Task 2

I decided to use aggregated data from 20 runs with the same configuration for this experiments. Our variables are going to be the maximum number of generations, the population size and the mutation chance. Tests were performed on the same computer and on two different TSP instances: `rondrit016.tsp` and `rondrit050.tsp`.

Since the goal of this tests was to evaluate the performance of different parameter settings I wanted to test under the design and the production perspectives. Therefore I decided to take a look at the best solution found in 20 runs of that configuration and the average and standard deviation of the found solutions.

The elitism parameter was set to 5% of the population and the crossover chance was 0.95 for all experiments. the effect of this parameters was analyzed later. To help with the testing I wrote a testing script in Matlab that only runs the GA without the GUI. this allows for faster and iterative testing with different parameters.

The effect of population size and number of generations

As expected the GA obtains better solutions the more evaluations it does. More generations yield better results and more individuals lead to better results too. We can see this results both in the best solution found across 20 runs and in the averaged results.

Of course, the bigger our population is and the longer the Ga runs, more computational power is needed to obtain the results, so we have to find a balance between performance and resources.

Finally, the computational cost of the algorithm is a function of $N_{gen} * P_{size}$ where N_{gen} and P_{size} are the number of generation and the size of the population. However the topology of the search space and the fitness function can be explored by viewing how the GA performs with a constant product $N_{gen} * P_{size}$ but different values of N_{gen}.

Assuming our GA can properly preserve diversity across generations, more individuals in less generations lead to a wider search of the search space. While more generations with a smaller population lead to deeper searches (refining solutions after every iteration to find better one near them). If our fitness function over the search space has several local optima a wider search can better explore the vicinity of all of them while a deeper search is better for optimization problems with a single optimum that we are trying to approach iteratively.

In this case, two of the test $N_{gen} = 100, P_{size} = 50$ and $N_{gen} = 50, P_{size} = 100$ had the same computational cost, however the wider one ($N_{gen} = 50, P_{size} = 100$) scored better results for both problems and for almost every $mut\%$ value. This applies to the best and average of the solutions. I explain it by the nature of the problem. Small changes in the genotype can lead to big changes in the fitness of the solutions and the complexity of the TSP lead to have a lot of good solutions with completely different genotypes. The higher amount of individuals allows us to explore a search space where refinement of solutions is clearly limited.

The effect of mutation

The main goal of mutation is to preserve the diversity of the population and to break out of local optima. However, too much mutation can erase the information about the solutions gathered from the previous iterations. Another interesting parameter is the strength of mutation. The way mutations is implemented here, its impact is bigger the smaller the number of cities is.

The results we got show that 20% mutation chance is clearly too much for `rondrit016.tsp` but may have a case for `rondrit050.tsp` and the mean best fitness is generally better. For the smaller problem, the balance seems to be somewhere between 5 and 10% if we look at it form a production perspective. But the design perspective would imply that 10% is desirable since it achieves the best results at least once.

In the bigger problem, the overall best solutions is sometimes found with a mutation chance of 10% and other times with a 20%. So probably the sweet spot is between those values.

The effect of elitism

To test the role of elitism in retaining the best individuals in the gene pool I performed tests on `rondrit050.tsp`, with a constant mutation chance of 20% and crossover rate of 0.95. Elitism has proven powerful in the literature, but too much elitism can kill diversity and the evolutionary process.

The tests we performed under this conditions clearly show this trend: with such high mutation chance a small elite is more subject to disappear and less likely to overtake the population and kill diversity. The experiment shows that increasing the elite size yields better results up until 35% but after that the diversity trade-off kicks in leading to worse results.

The effect of crossover rate

Crossover rates is related to elitism as is one of the factors that determines how fast the evolution process goes. While elitism preserves the best individuals, a low crossover rate preserves random individuals from evolving by recombination (this can also be achieved through different replacement policies). Also, the crossover rate and mutation chance must be balances since they are the two power horses of the genetic algorithm exploration of the search space.

The tests performed had a mutation chance of 20% and elitism of 5% of the population.

The results show what we describe, the lower the crossover rate, the bigger the impact of N_{gen}, its shows a slower but steadier evolution. The fact that a high crossover performs worse than a low one make me think that the crossover operator may not be suited for the problem and may be introducing more noise than anything else.

It is notable that the 25% crossover rate test yielded the best results for the problem across all of the experiments perrformed.

Critique to my methodology

Even if my experiments isolated variables and shows the expected theoretical results, parameter tuning is much more complex. Parameter tuning is itself a multidimensional optimization problem and to explore its search spaces better a different parsing method must be used.

One cannot take my results and apply a linearity argument to say that a mutation chance of 20% with 35% elitism and 25% crossover rate is going to provide the best results. The interconnection between the different parameters and between them and the problem is much more complex and definitely not multilinear.

Task 4

Once the different stopping criteria, crossover and mutation operators were implemented it was time to do some parameter tuning. To do this I decided to run 8 different parameter combinations, 20 times for each combination of crossover and mutation operator. The test were run on the same problem `rondrit050.tsp` with both early stop criteria active (faster testing).

Since the final goal was to compare with benchmarks for specific instances of TSP, I decided to focus only on the best result obtained, instead of in the MBF. To keep things fair between the two crossover operators (`order` was 100 times faster than `edge_recombination`) I capped the computation time for this battery of tests but the results then were awful for `edge_recombination`. Finally I resorted to only test with `order` and then do a fair competition between them in the fine tuning stage. The computation time was capped to 10 seconds per run: which lead to 24 minutes of testing per mutation operator.

Since the best results were achieved by `simple_inversion` and `insert` I decided to test both on a second battery of test. The results showed better performance for `simple_inversion`, elite percentages of 15 and 20, crossover ratios of 0.65 and 0.75 and a mutation chance of 5%.

For the final test I fixed the mutation as `simple_inversion` 5%, the computation time to 1 minute per run and limited the number of runs to 10. The test was between the two crossover operators with different crossover chances. the chosen elite percentage was 15. This experiment run for 1 full hour.

The results showed that the `order` crossover performed better, and that the main problem to its evolution, was eventually stalling (no improvement after an amount of generations). To do the final tuning I decided to test different mutation chances with a crossover chance of 75%. Apparently the best performance was attained by the lowest mutation rate (2%), however some higher ones like 7% and 10% showed potential, perhaps an adaptive rate could solve this problem.

Task 5

To test the effect of loop removal, I repeated the final test with it active. Also I turned on the reset mechanism in case of stalling. The stalling was defined as no improvement in 200 iterations.

The effect was clear, better results were achieved overall under the same computation restrictions. Also, the 10% mutation chance proved to be the best under this circumstances, achieving the lowest value: 5.8283 two times in 10 runs, I think this may be the global optimum for the problem.

Seeing these results I decided to not perform any statistical tests to see if I was dealing with the same underlying distribution. But If the results had not been so different, a T-test or a Wilcoxon test could have been useful.

Task 6

XQF131

I decided to test the configurations from Task 5 on the `xqf131.tsp` benchmark with loop detection enabled, but the results where far from the known optimum: the best result I got in 60, 1 minute runs was 654.2329 while the known optimum is 564. Seeing this results I decided to test the algorithm increasing the population to 400 and allowing it to run for 10 minutes without resetting. I set the mutation chance to 15% to help in the later stages of evolution and prevent stalling. It reached a minimum of 590.2922 but the improvement in the latest iterations was too slow. A final test was performed with the same limits but a population of 800 individuals, the test reached a minimum of 595.5334, which is a worse result. The comparison between the two runs also showed slower evolution for this later test.

Over all, with ten minutes of computation some decent results where achieved, however the evolution still felt stagnated in the latest iterations of the algorithm. This may pose a problem when trying to optimize the solution to reach the actual minimum for the problem. The way the graph looks, it doesn't seem that running the same algorithm for a longer time period would find the global optimum easily.

BCL380

Seeing the results from the previous benchmark I decided to replicate the 400 individuals, 10 minute experiment with exactly the same parameters parameters, but the results were not as good as expected. In fact they were really bad: the GA found a solution with a cost of 4642.4 when the optimum was 1621.

To understand this result I thought of the impact of mutations: right now my GA was using simple inversion, that means that only two edges of the graph change with each mutation. The plot shows the lack of diversity that this generates in the algorithm. Since mutations have a low impact (changing only 0.5% of the edges), the main tool for introducing diversity is recombination through crossover. I run a second test with a 95% chance for crossover and using a different mutation operator: scramble.

The result obtained was even worse than the first one, even if the evolution looked more diverse (more difference between the best and average) it also looked slower than the previous one. Perhaps we were introducing too much noise. For the third run I decided to bring the crossover ratio back to 75% and decreasing the mutation chance to 5%. The result achieved was better than the one from the first test but still far from acceptable: 4593.99.

Seeing this progress I though that maybe a slower but steadier evolution would be best suited for this problem. I decided to increase the elite (30%) and reduce the crossover ratio even further 55%. The result obtained showed steady evolution and better results by a fair margin: 3697.61. I still consider this a bad result, even if it is the best one that I could attain with 10 minutes of computation, a human made solution connecting the dots manually would probably obtain a better score.

I think that the problems that we encountered exporting the parameters optimized for a different instance can be adduced mainly to the change in the number of cities. The local optimization heuristic for the GA may not be as effective for big instances of TSP.

I became curious of the trade off between number of generations and individuals per generation, so I decided to test different population sized for the next instances.

XQL662 and RBX711

I decided to use the same parameters from the last run of BCL380 on these problems, changing the number of individuals per generation: I tried with populations of size 200, 400 and 600. Also, I allowed the GA to run for 20 minutes.

The results showed better performance for the smaller populations in both cases, but still were overall bad. I decided to perform a final test with populations of 100 and 200 but 30 minutes of computation time. The best result for XQL662 was 7079 when the optimum is 2513, and the best for RBX711 was 9554 when the best is 3115. Again, a hand drawn solution would probably have been better.

We can notice a trend of how my algorithm gets worse and worse the bigger the problem is. I tried several configurations but none of them seemed fit to tackle problems this big: either they got lost trying to untangle the mesh of paths from the original generation solutions (like this two problems) or lacked adaptability in the later stages of evolution to reach the actual optimum and not get stagnated (like with XQF131).

Seeing this results and having invested too many ours tuning my parameters I decided to move to the last small problem.

Belgium Tour

To confirm my theory that simple inversion performed better for small problems I decided to test it against scramble mutation with different population sizes and high computation limits. Most of them got stuck in local minimums fairly early in the computation. even if the s.inversion algorithms showed better performance, it looked more due to luck than to parameter tuning.

I decided to activate the reset on stall mechanic of my GA to see if I could achieve better results. I set the stall reset to 400 generations and after 3 minutes of computation 10 results had been reached, 4 of them where tied by being the lowest values: 659.816. I think this proves the usefulness of the resetting mechanic.

Reflection: No free lunch for optimization

After seeing the results from these tests, it is clear that different instances of TSP require different parameter setting to behave optimally. Certain configurations do not even show a sense of locality where small changes to the parameters show small changes in the behavior of the GA.

Therefore, we can conclude that performing parameter tuning for an specific instance of TSP will not help us that much when tackling a different instance of the problem. This phenomenon has been studied in the area of search and optimization algorithms and the widely accepted outcome is that there is no heuristic that is best for all problems; or, in this case, there is no optimal parameter configuration across all TSP instances.

A formal demonstration of this argument was made by David H. Wolpert and William G. Macready in the late 90s in their no free lunch articles for search and optimization heuristics [1] [2]. This does not justify my total incompetence when tuning for the big TSP instances (as Wolper and Macready explain in [3]), I just decided that is was not worth the time sink.

Critique to my methodology

My methodology for parameter tuning was completely unstructured and the results speak for themselves. I am aware this is an unsolved problem in the GA academia, but certainly there better ways to confront the problem than following my instinct at the point of deciding which configurations to run next.

Task 7

(a) Parent selection methods

I decided to add two different parent selection methods to the GA in addition to the ranking stochastic one already implemented. Those methods were fitness proportional selection and tournament selection with two tournament sizes: 3 and 5 individuals.

To test the effect of this changes in the selection policy I run a fast test on the benchmark `xqf131.tsp`. The goal was to run the beginning of the evolution with the different selection mechanisms and see if a trend could be established. The parameters for the test were set to 200 individuals, 10% scramble mutation, 55% crossover and 30% elite. The computation cap was 10 s. per experiment and 20 experiments where made per selection policy.

Since the mean and standard deviations of the results were fairly similar across all selection policies I decided to run t-tests and look at their P-values to check how close the distributions really were. There was a high correlation between ranking universal substitution and the new implemented policies (p-values of 0.4 and higher), but the highest correlations where between the new methods: The test between the 5 individuals tournament and fitness proportional selection had a p-value of 0.96 which is an almost identical distribution.

Also, the new selection policies outperformed the original one, but only by a slight margin. However, I got the feeling that with this high elite percentage and low crossover chance, the selection policy would be of marginal importance to the algorithm and I decided to make a second test: Same configuration but 10% elitism and 85% crossover. In this case the mean values obtained where also very similar but the worst performance was showed by the tourney selection policies. Also the results were much worse than those from the previous experiment, but this is not relevant to study the effect of selection policies.

Perhaps the biggest change in the outcome can be found in the t-test p-value matrix: in this case the main out-layer was the 5 individual tournament selection. In fact its p-value vs the fitness proportional selection was 0.18 while Ranking SUS showed high similarity with the other distributions, having a p-value of 0.95 with the 3 individuals tournament.

The main conclusion is from this results is that one cannot obtain a general conclusion about the effect of selection across all problems and parameter settings. Even if the effect of the changes for the tests performed was limited, one cannot assume that it would be the same for other settings or instances of TSP. Again, I feel compelled to bring up the No Free Lunch theorems.

More extensive testing could be performed to single out the properties of the different selection policies, but the time limitations on this assignment made me not deem it worth it.

Final Note

I tried to not repeat information presented on the course material and to select the right experiments that would allow me to illustrate important properties of GAs. I am aware that there are plenty of improvements I could have made to my methodology and code, starting with the 3 optional tasks from task 7 that I did not implement. Again, as the only student in this project, I did my best within my capabilities and time restrictions.

If there is any part of my code or report that requires clarification or discussion, please contact me and I will reply as soon as I can.

Code

All the code for the project can be found in the zip file that goes with this report. The code is divided in different folders with versions of the incremental design. The final version is inside the folder Task 7.

Results

Task 2

First battery of tests.

	`rondrit016.tsp`: best solution					
	`50 individuals`		`100 individuals`		`150 individuals`	
`mut.`	`50 gen`	`100 gen`	`50 gen`	`100 gen`	`50 gen`	`100 gen`
5%	3.5013	3.4238	3.3828	3.3500	3.4458	3.3500
10%	3.4342	3.3828	3.4658	3.3500	3.4230	3.3500
20%	3.5926	3.4230	3.4962	3.3770	3.4313	3.3500

	`rondrit016.tsp`: average $\pm$ standard deviation					
	`50 individuals`		`100 individuals`		`150 individuals`	
`mut.`	`50 gen`	`100 gen`	`50 gen`	`100 gen`	`50 gen`	`100 gen`
5%	3.7787 ± 0.1564	3.6124 ± 0.1358	3.5820 ± 0.1207	3.4439 ± 0.0626	3.5520 ± 0.1046	3.4352 ± 0.0514
10%	3.7819 ± 0.1116	3.5585 ± 0.1241	3.5956 ± 0.0925	3.4581 ± 0.0748	3.5649 ± 0.1073	3.4473 ± 0.0594
20%	3.8071 ± 0.1618	3.6727 ± 0.1241	3.6283 ± 0.1229	3.4642 ± 0.0711	3.5968 ± 0.1167	3.4762 ± 0.0936

	`rondrit050.tsp`: best solution					
	`50 individuals`		`100 individuals`		`150 individuals`	
`mut.`	`50 gen`	`100 gen`	`50 gen`	`100 gen`	`50 gen`	`100 gen`
5%	15.9577	14.6841	15.4301	12.9806	14.8164	14.3775
10%	15.134	15.1911	14.6667	13.9423	14.0962	13.4207
20%	15.384	14.6714	14.6799	13.493	14.6432	13.9971

	`rondrit050.tsp`: average $\pm$ standard deviation					
	`50 individuals`		`100 individuals`		`150 individuals`	
`mut.`	`50 gen`	`100 gen`	`50 gen`	`100 gen`	`50 gen`	`100 gen`
5%	16.891 ± 0.5357	16.324 ± 0.6801	16.139 ± 0.4650	15.128 ± 0.7237	15.724 ± 0.4486	15.116 ± 0.5216
10%	16.876 ± 0.6948	16.404 ± 0.4934	16.01 ± 0.5650	15.203 ± 0.7913	15.624 ± 0.5492	14.892 ± 0.6623
20%	16.833 ± 0.5507	16.170 ± 0.6547	16.016 ± 0.5858	14.97 ± 0.6961	15.700 ± 0.6047	14.900 ± 0.5845

Elitism tests

	rondrit050.tsp: best solution					
	50 individuals		100 individuals		150 individuals	
eli.	50 gen	100 gen	50 gen	100 gen	50 gen	100 gen
5%	15.384	14.6714	14.6799	13.493	14.6432	13.9971
10%	14.5388	12.5619	14.4372	12.4687	13.624	12.0606
20%	14.1508	11.4103	13.6454	11.4587	12.9849	12.0408
35%	14.0986	11.435	13.6668	11.8153	13.7814	11.2964
50%	14.4856	12.0615	13.8385	12.0259	13.5558	12.2438

	rondrit050.tsp: average ± standard deviation					
	50 individuals		100 individuals		150 individuals	
eli.	50 gen	100 gen	50 gen	100 gen	50 gen	100 gen
5%	16.833 ± 0.5507	16.170 ± 0.6547	16.016 ± 0.5858	14.97 ± 0.6961	15.700 ± 0.6047	14.900 ± 0.5845
10%	15.755 ± 0.6375	13.842 ± 0.8191	15.290 ± 0.508	13.621 ± 0.6573	14.734 ± 0.5177	13.282 ± 0.6981
20%	15.226 ± 0.5298	12.921 ± 0.7952	14.731 ± 0.5455	12.749 ± 0.7304	14.507 ± 0.6202	12.771 ± 0.4492
35%	15.005 ± 0.5739	12.867 ± 0.6125	14.576 ± 0.4545	12.661 ± 0.5683	14.562 ± 0.3777	12.491 ± 0.6652
50%	15.412 ± 0.6216	13.274 ± 0.5073	15.063 ± 0.5114	13.233 ± 0.5247	14.745 ± 0.5019	12.964 ± 0.4130

Crossover tests

	rondrit050.tsp: best solution					
	50 individuals		100 individuals		150 individuals	
XO.	50 gen	100 gen	50 gen	100 gen	50 gen	100 gen
95%	15.384	14.6714	14.6799	13.493	14.6432	13.9971
75%	14.6812	13.5921	13.2881	11.7148	12.8455	11.6605
50%	13.2726	10.9088	12.4115	9.9108	12.5393	9.7074
25%	13.4538	10.5345	12.6164	9.6757	11.6744	9.4606

	rondrit050.tsp: average ± standard deviation					
	50 individuals		100 individuals		150 individuals	
XO.	50 gen	100 gen	50 gen	100 gen	50 gen	100 gen
95%	16.833 ± 0.5507	16.170 ± 0.6547	16.016 ± 0.5858	14.97 ± 0.6961	15.700 ± 0.6047	14.900 ± 0.5845
75%	16.051 ± 0.5173	14.601 ± 0.6539	14.973 ± 0.6856	12.828 ± 0.5517	14.705 ± 0.6468	12.698 ± 0.5969
50%	14.721 ± 0.6225	11.860 ± 0.5445	13.770 ± 0.6211	11.018 ± 0.6344	13.553 ± 0.5503	10.743 ± 0.4598
25%	14.687 ± 0.5657	11.575 ± 0.5361	13.508 ± 0.5236	10.691 ± 0.4626	13.121 ± 0.5903	10.294 ± 0.3731

Task 4

First battery of tests.

	rondrit050.tsp: order crossover							
	Elite percentage: 5%				Elite percentage: 15%			
	Crossover: 95%		Crossover: 65%		Crossover: 95%		Crossover: 65%	
Mutation:	5%	15%	5%	15%	5%	15%	5%	15%
inversion	8.9604	9.3957	8.3333	8.5046	7.6965	8.0226	7.7279	7.7115
simple_inversion	9.2294	9.6845	8.0095	8.1719	7.6743	8.059	7.474	7.7258
scramble	9.0211	9.6716	7.9784	8.5365	8.0939	8.2355	7.693	7.7071
swap_mutation	9.3886	9.5913	8.2924	8.0507	7.5039	8.1126	7.6844	7.9567
insert	9.3122	9.3702	8.2057	8.0318	7.8147	8.0765	7.692	7.8171

Second battery of tests.

	rondrit050.tsp: order crossover							
	Elite percentage: 10%				Elite percentage: 20%			
	Crossover: 75%		Crossover: 50%		Crossover: 75%		Crossover: 50%	
Mutation:	5%	10%	5%	10%	5%	10%	5%	10%
simple_inversion	8.07	7.9925	7.6818	7.7779	7.4455	8.0986	7.6829	7.7309
insert	7.9601	8.0544	7.6964	7.9262	7.9591	7.7151	8.2697	7.7708

Crossover test

	rondrit050.tsp: crossover test							
	Crossover: order				Crossover: edge_recombination			
XO chance:	80%	75%	70%	65%	80%	75%	70%	65%
Best result:	5.992	5.8728	5.9643	5.9232	9.1368	8.6139	9.6555	9.9544
Main cause of stop:	Stall	Stall	Stall	Stall	Time	Time	Time	Time

Final test

	rondrit050.tsp: mutation tuning					
Mut.:	2%	5%	7%	10%	15%	20%
Best:	5.8283	6.0493	5.9159	5.8913	5.9232	5.9214
Mean:	6.058 ± 0.1652	6.229 ± 0.1139	6.108 ± 0.1427	6.117 ± 0.1491	6.041 ± 0.0560	6.1892 ± 0.1586

Task 5

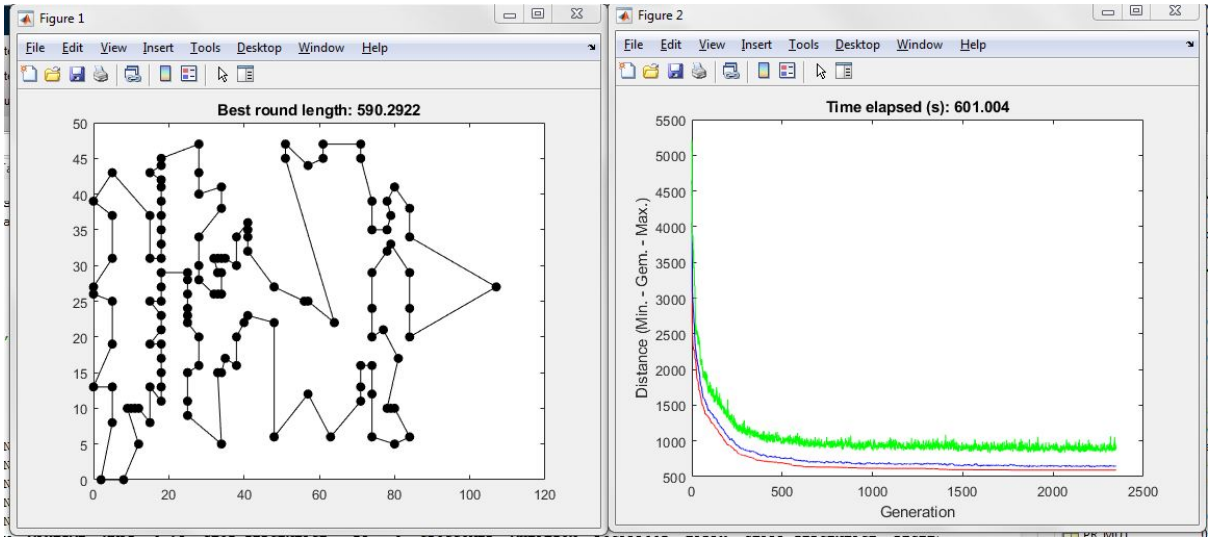
	rondrit050.tsp: effect of loop detection					
Mut.:	2%	5%	7%	10%	15%	20%
Best:	5.8283	5.8283	5.8455	5.8283	5.8455	5.8283
Mean:	5.920 ± 0.069	5.936 ± 0.052	5.908 ± 0.076	5.900 ± 0.060	5.910 ± 0.052	5.938 ± 0.085

Task 6

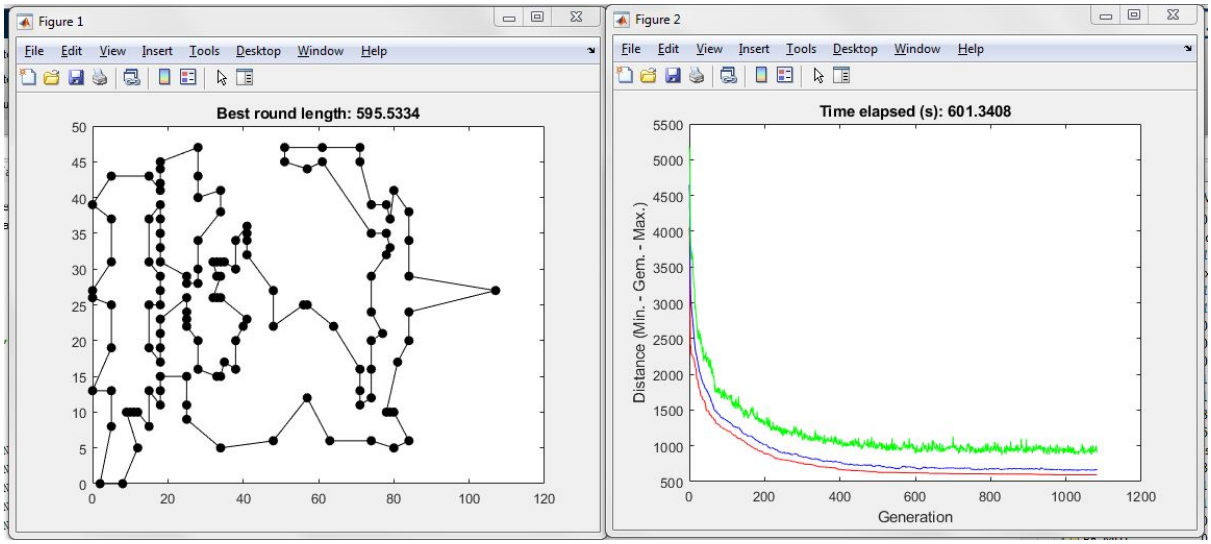
XQF131

Global optimum: 564

	xqf131.tsp with loop detection					
Mut.:	2%	5%	7%	10%	15%	20%
Best:	654.2329	648.3232	679.9358	662.383	680.2317	685.6766



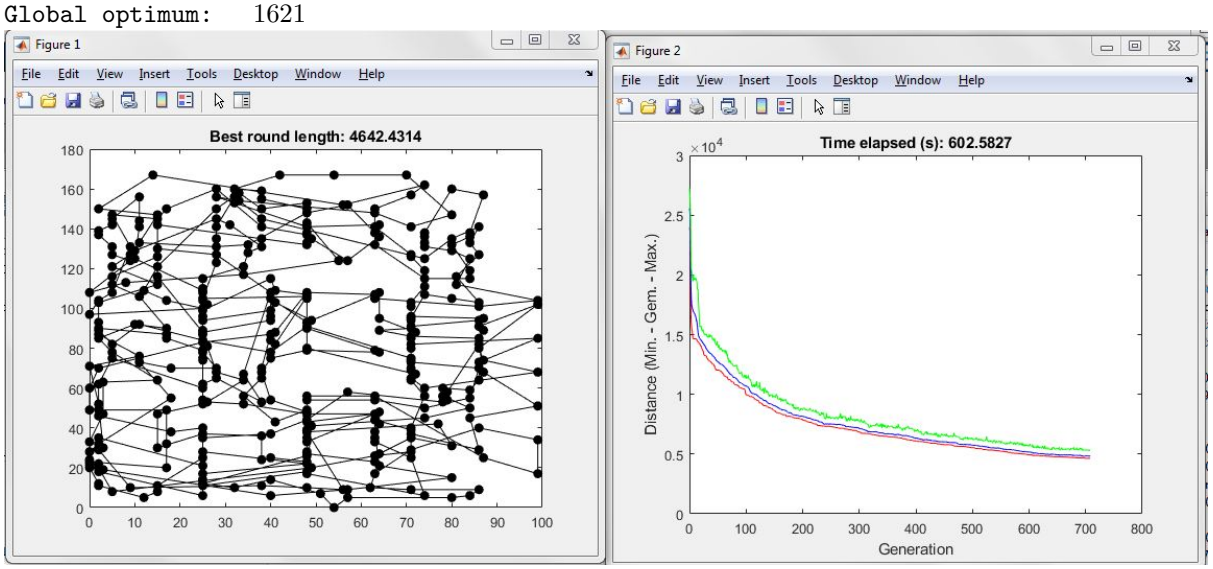
10 minute run with 400 individuals: 590.2922



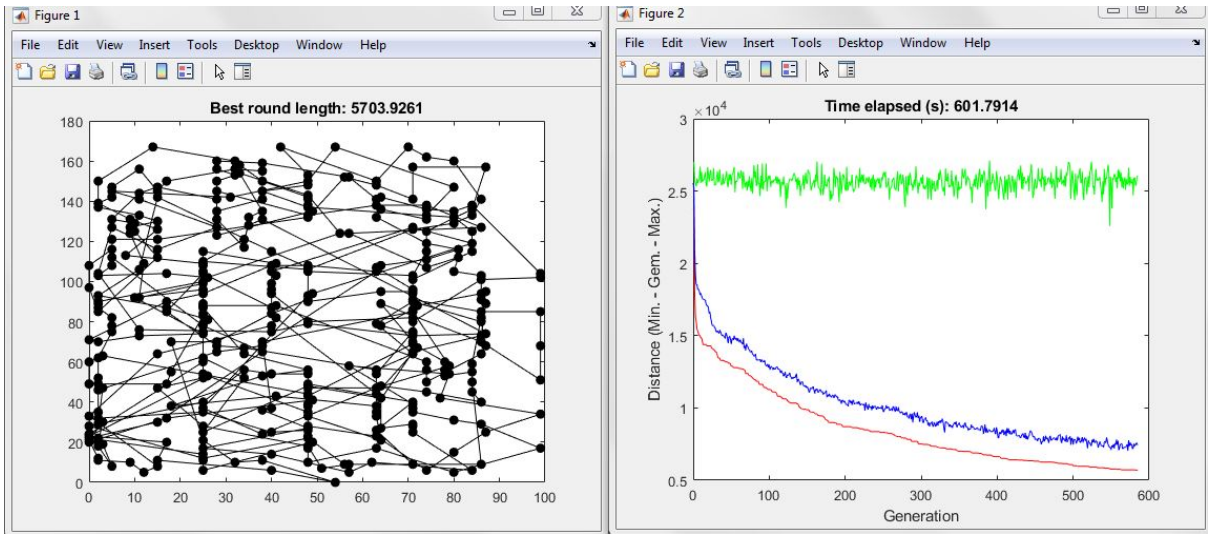
10 minute run with 800 individuals: 595.5334

BCL380

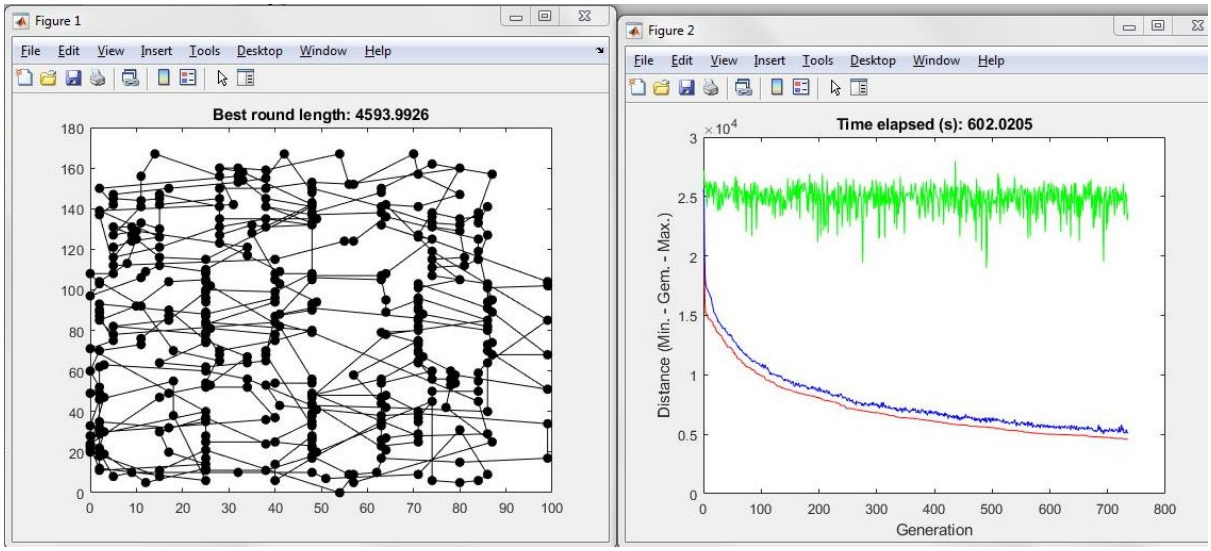
`Global optimum:` 1621



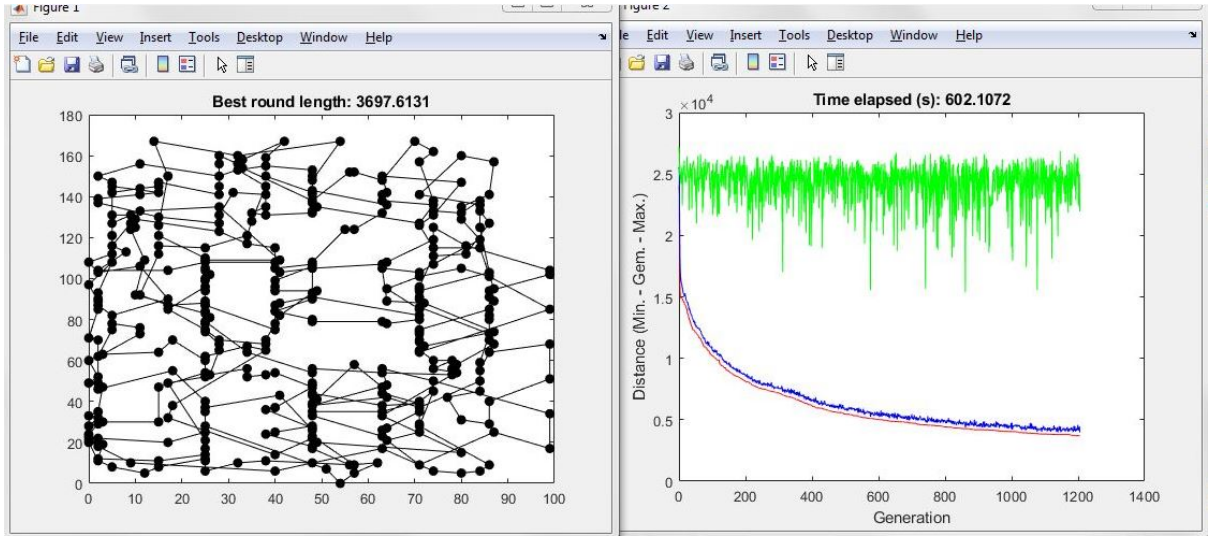
`10 minute run with 400 individuals, 75% XO and 15% simple_inversion mutation:` 4642.43



`10 minute run with 400 individuals, 95% XO and 15% scramble mutation:` 5703.92



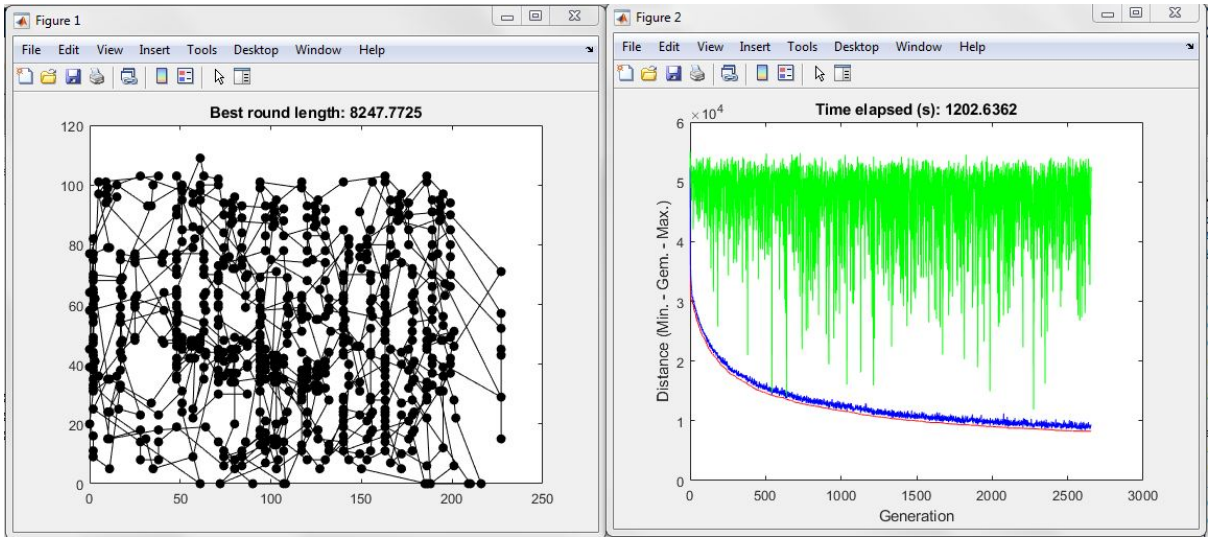
10 minute run with 400 individuals, 75% XO and 5% scramble mutation: 4593.99



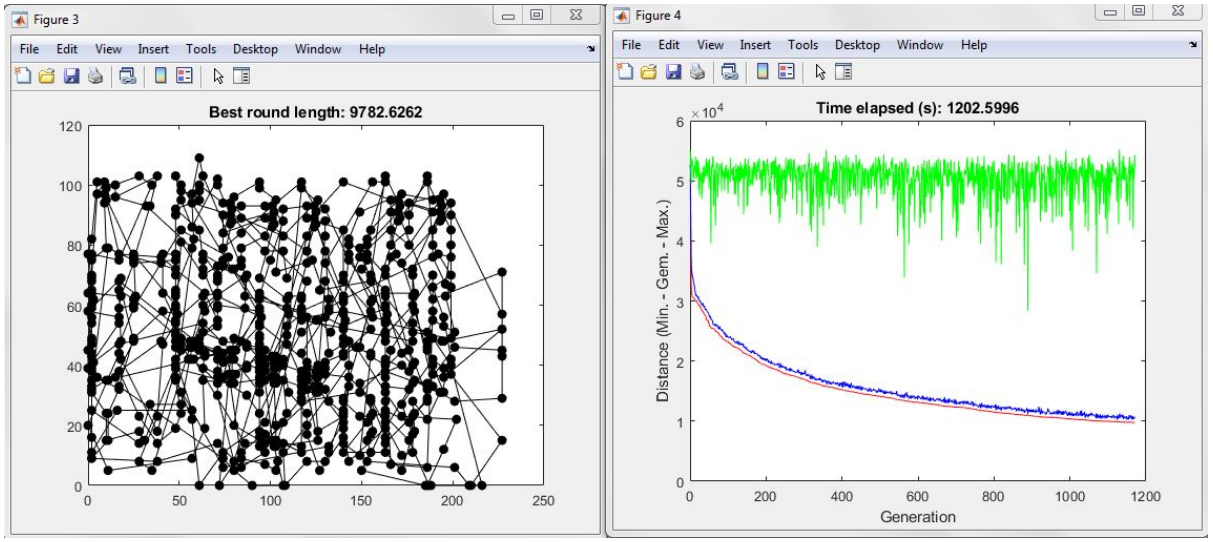
10 minute run with 400 individuals, 30% elite, 55% XO and 5% scramble mutation: 3697.61

XQL662 and RBX711

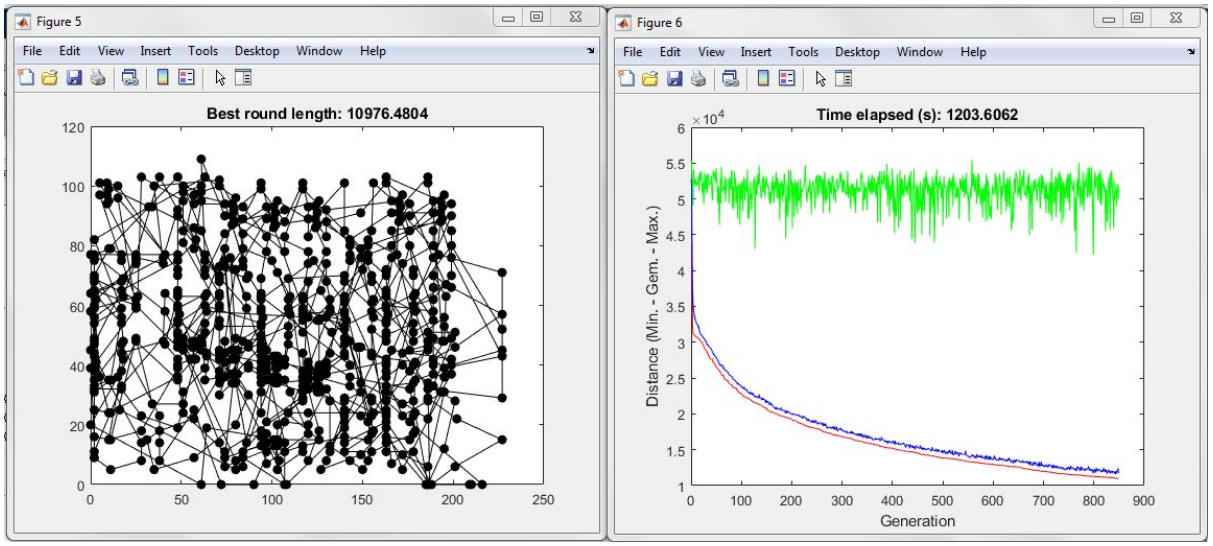
Global optimum XQL662: 2513



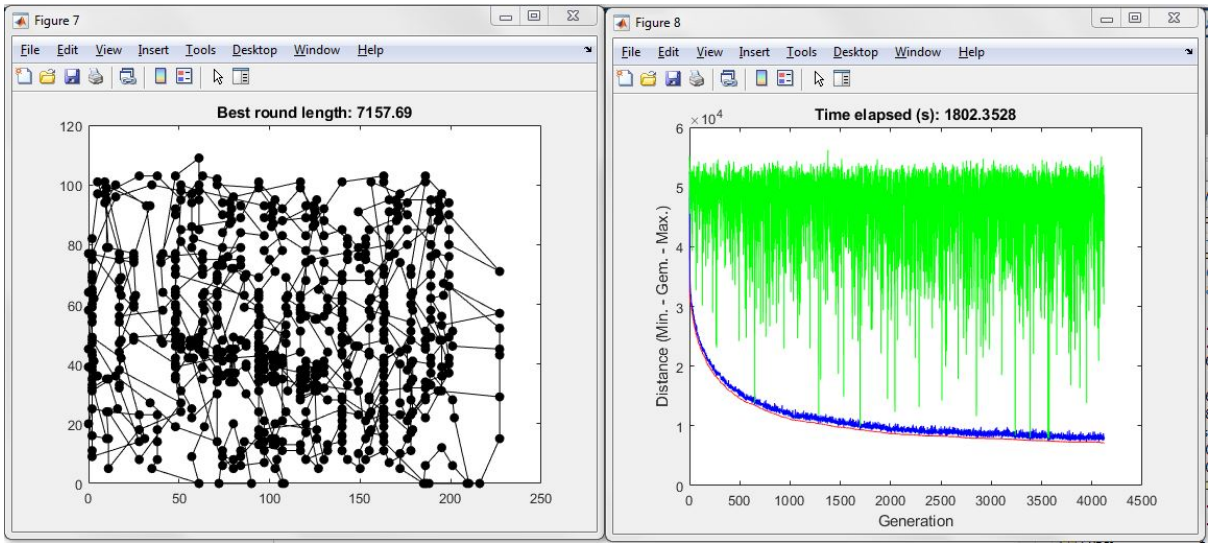
20 minute run with 200 individuals, 30% elite, 55% XO and 5% scramble mutation: 8247.77



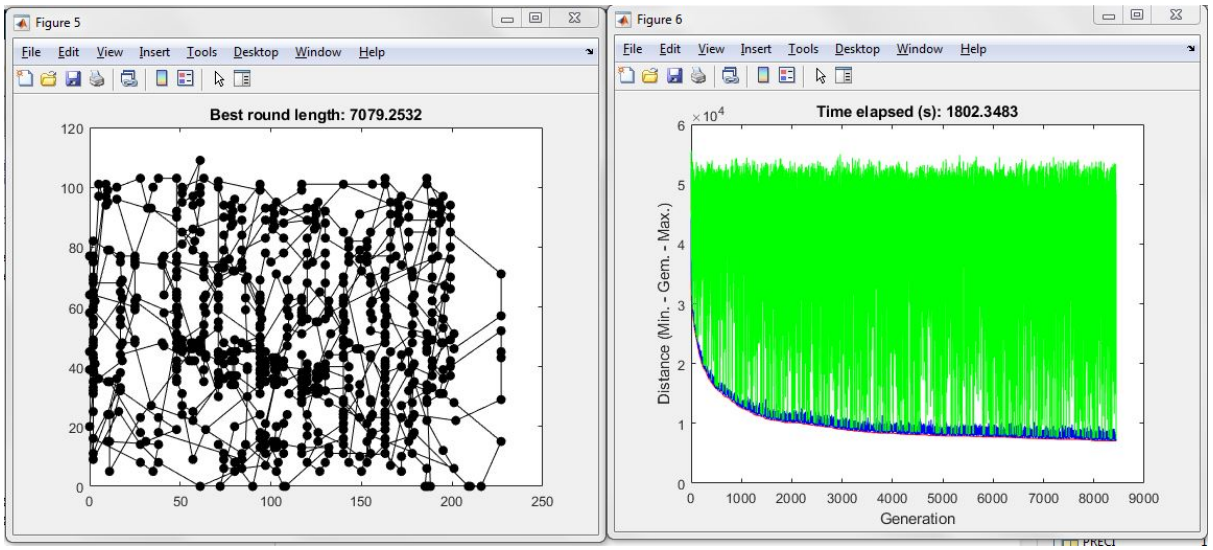
20 minute run with 400 individuals, 30% elite, 55% XO and 5% scramble mutation: 9782.62



20 minute run with 600 individuals, 30% elite, 55% XO and 5% scramble mutation: 10976.5

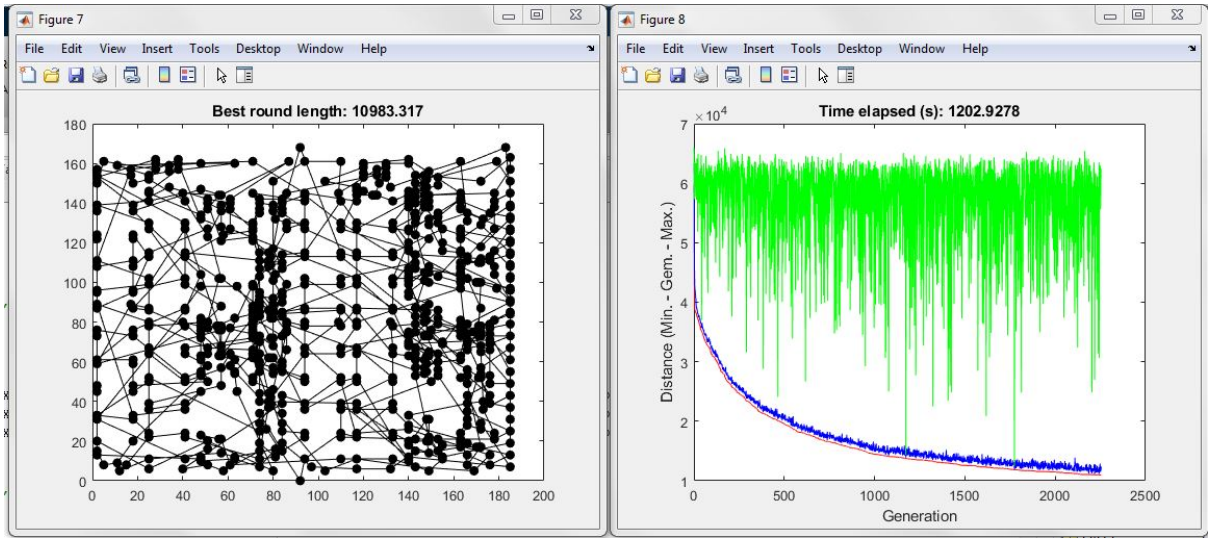


30 minute run with 200 individuals, 30% elite, 55% XO and 5% scramble mutation: 7157.69

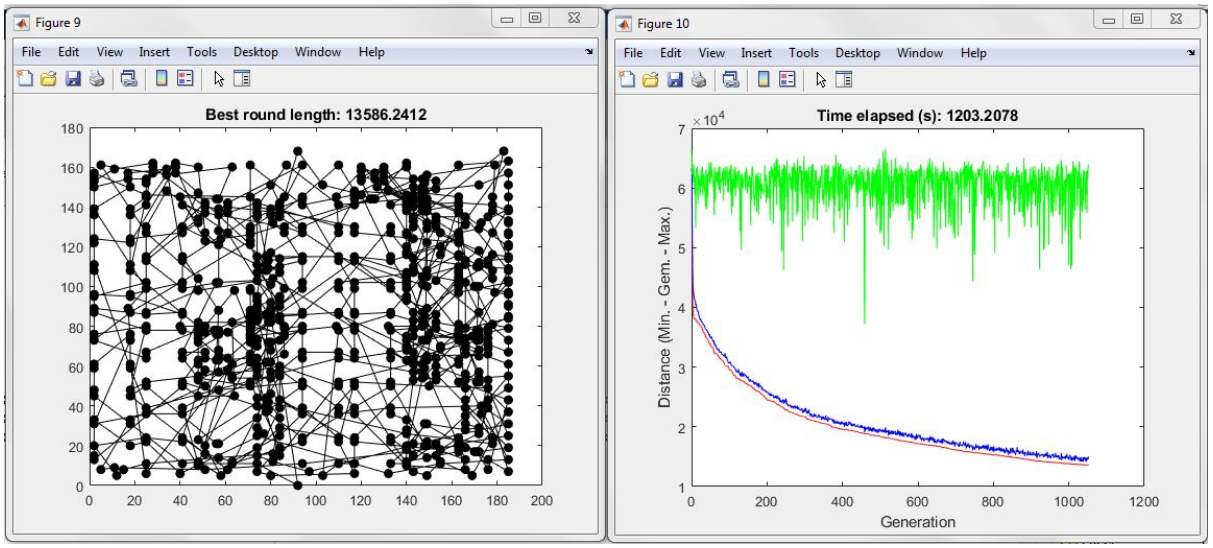


30 minute run with 100 individuals, 30% elite, 55% XO and 5% scramble mutation: 7079.25

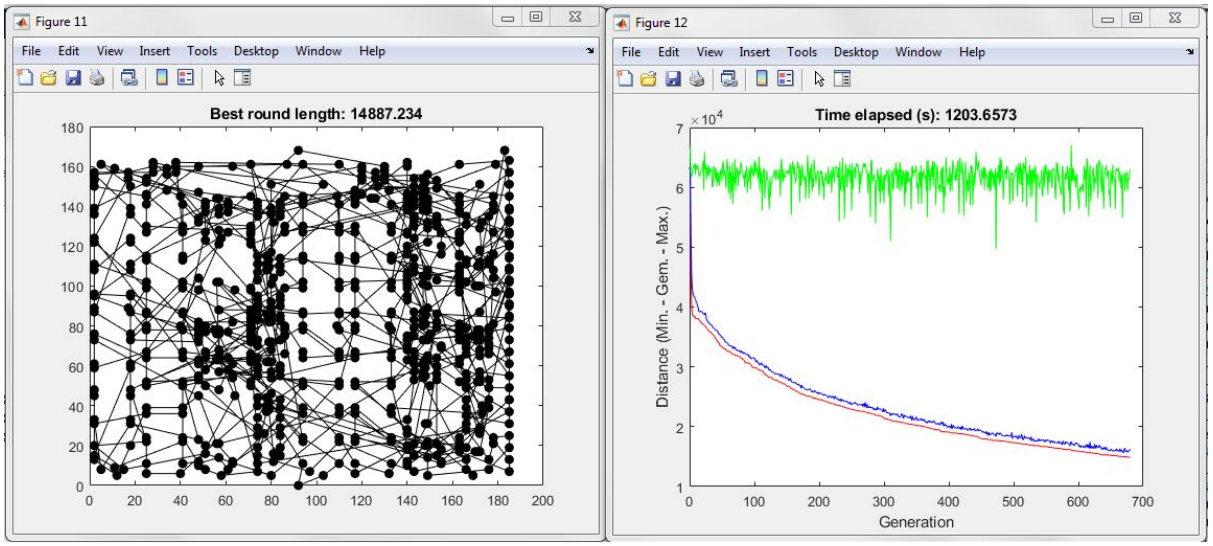
Global optimum RBX711: 3115



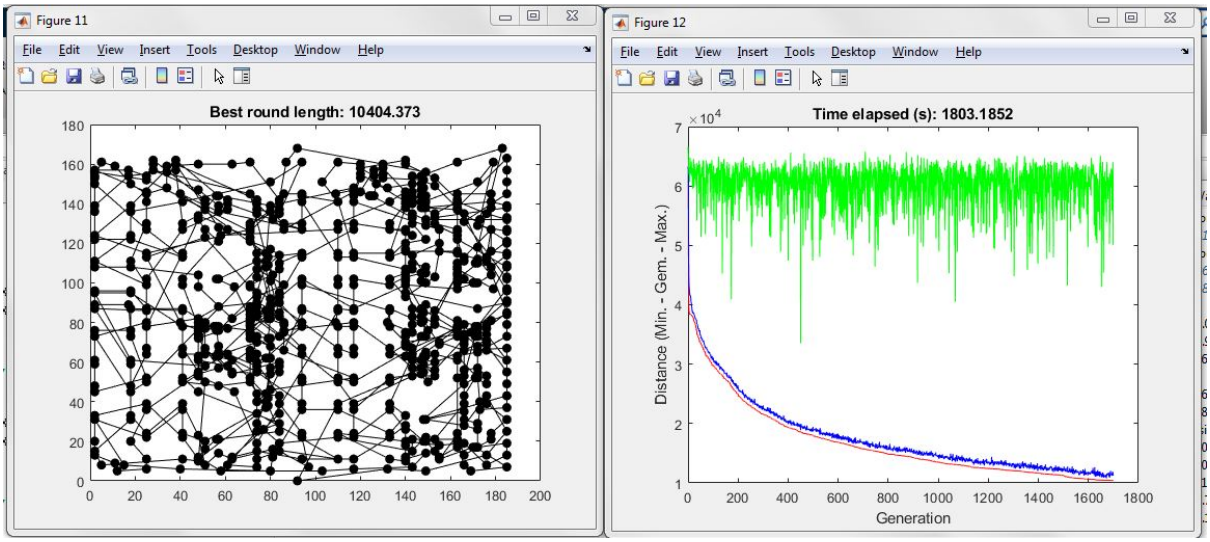
20 minute run with 200 individuals, 30% elite, 55% XO and 5% scramble mutation: 10983.3



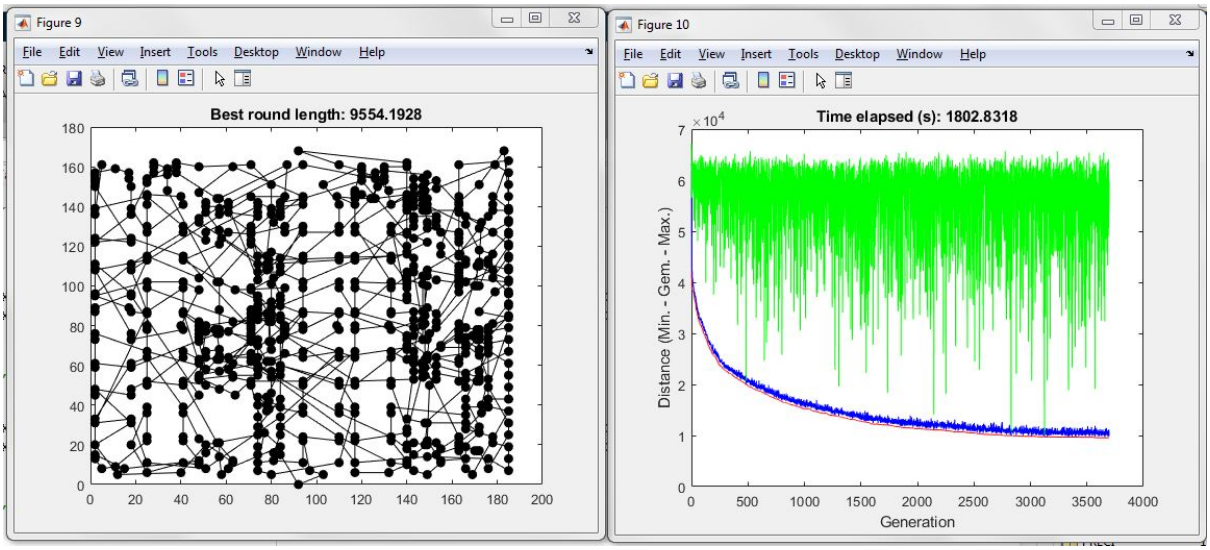
20 minute run with 400 individuals, 30% elite, 55% XO and 5% scramble mutation: 13586.2



20 minute run with 600 individuals, 30% elite, 55% XO and 5% scramble mutation: 14887.25

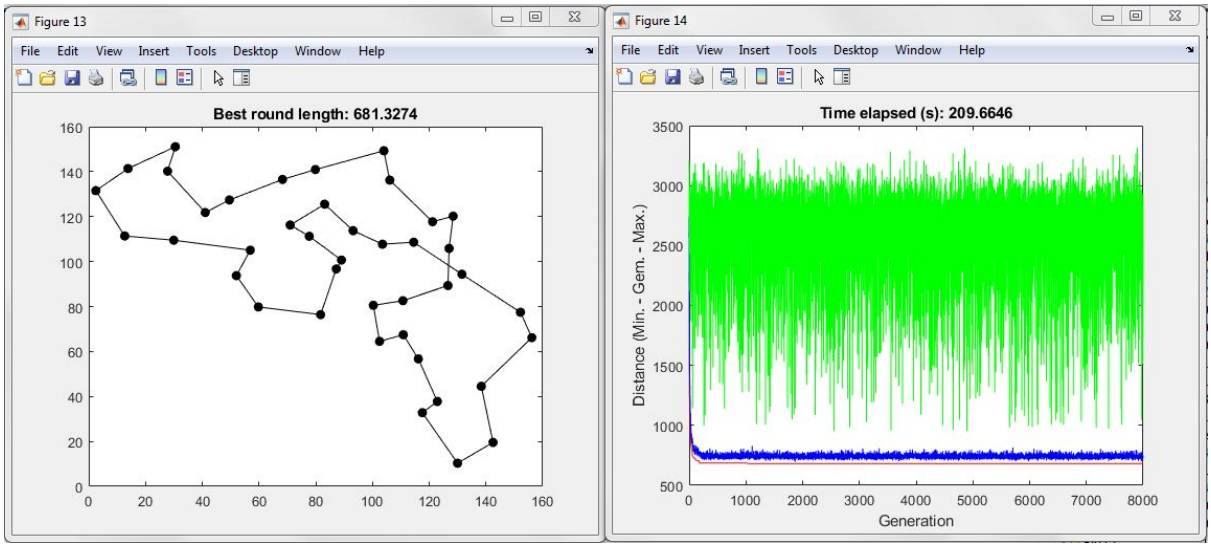


30 minute run with 200 individuals, 30% elite, 55% XO and 5% scramble mutation: 10404.4

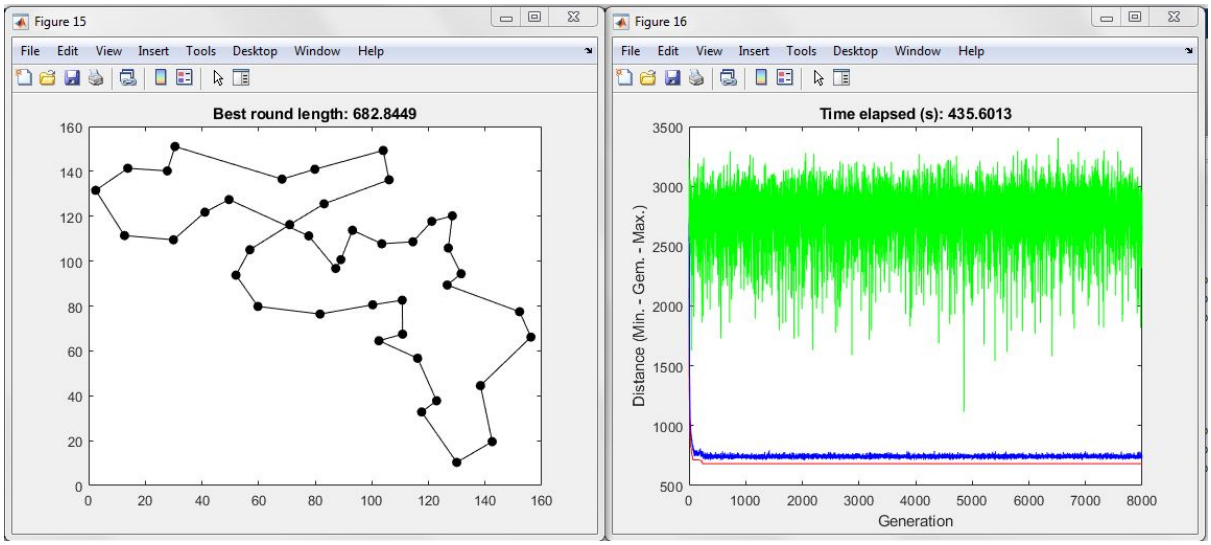


30 minute run with 100 individuals, 30% elite, 55% XO and 5% scramble mutation: 9554.19

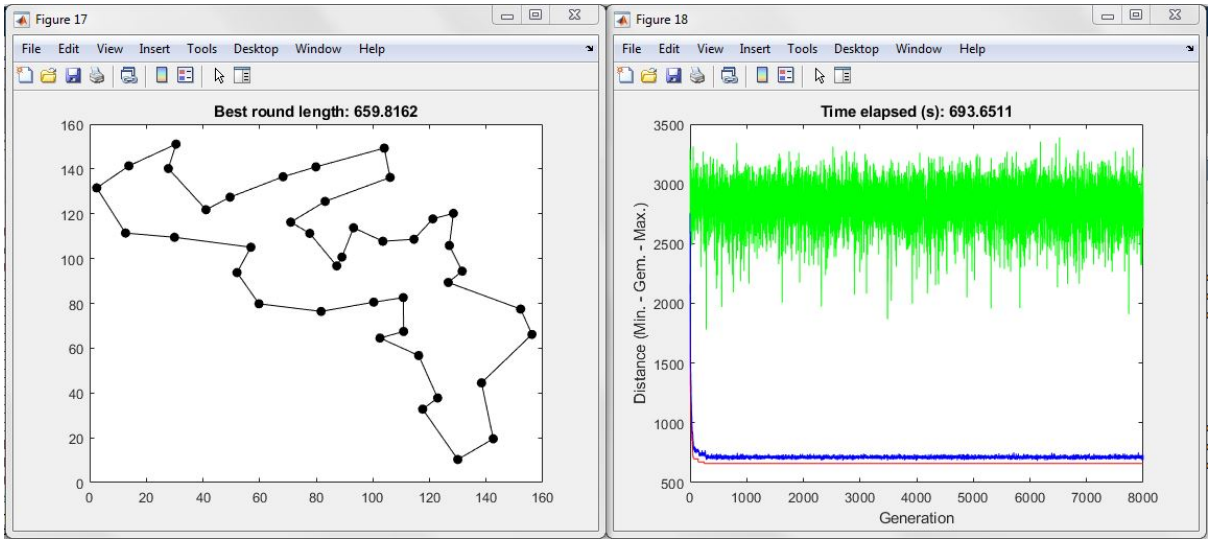
Belgium Tour



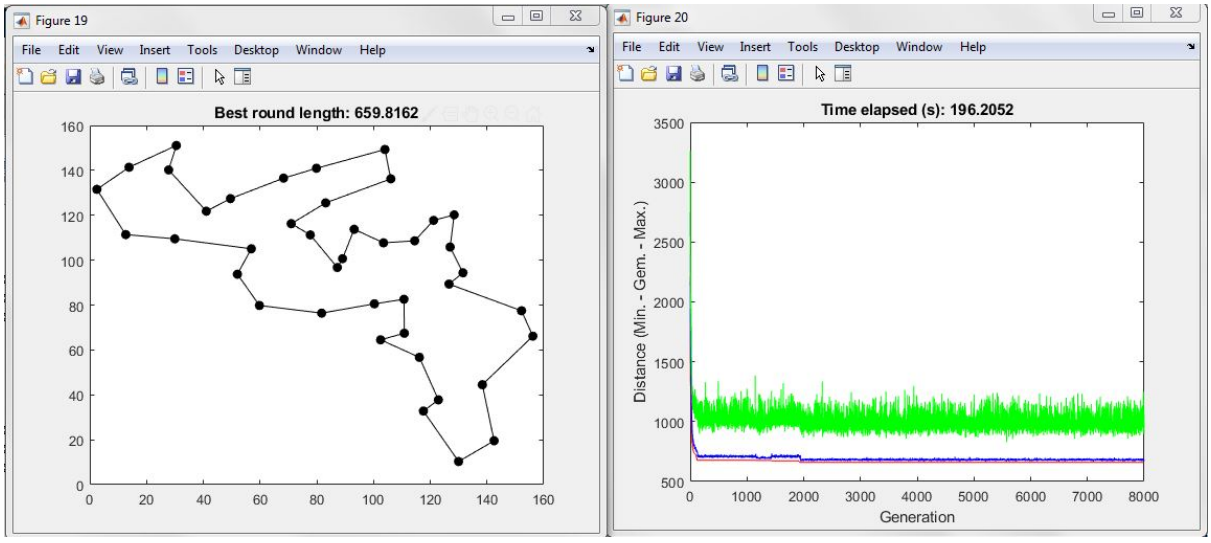
8000 gen run with 200 individuals, 30% elite, 55% XO and 5% scramble mutation: 681.327



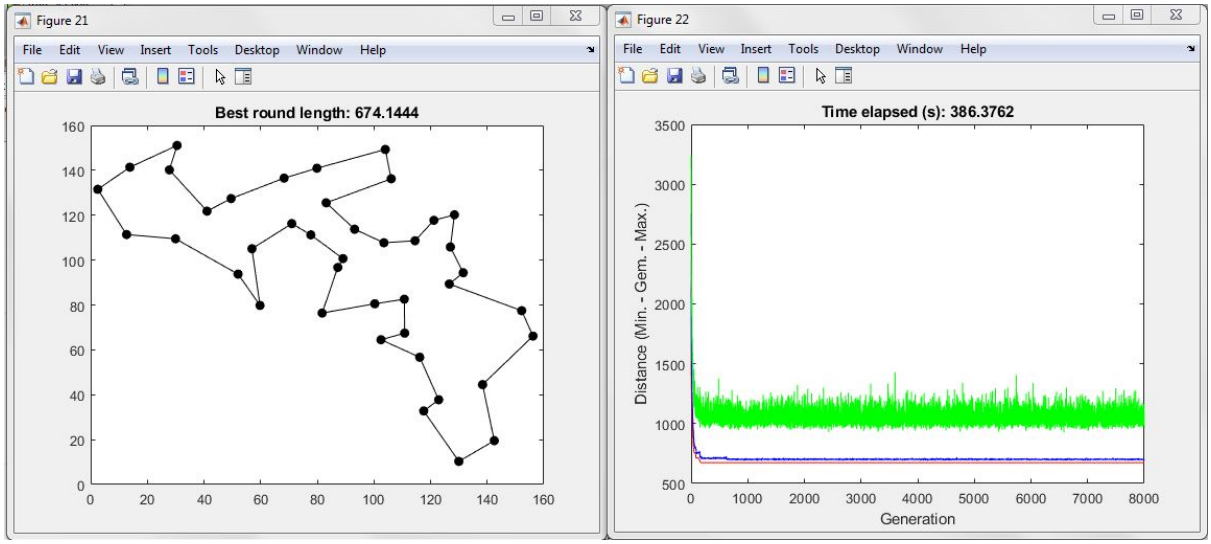
8000 gen run with 400 individuals, 30% elite, 55% XO and 5% scramble mutation: 682.845



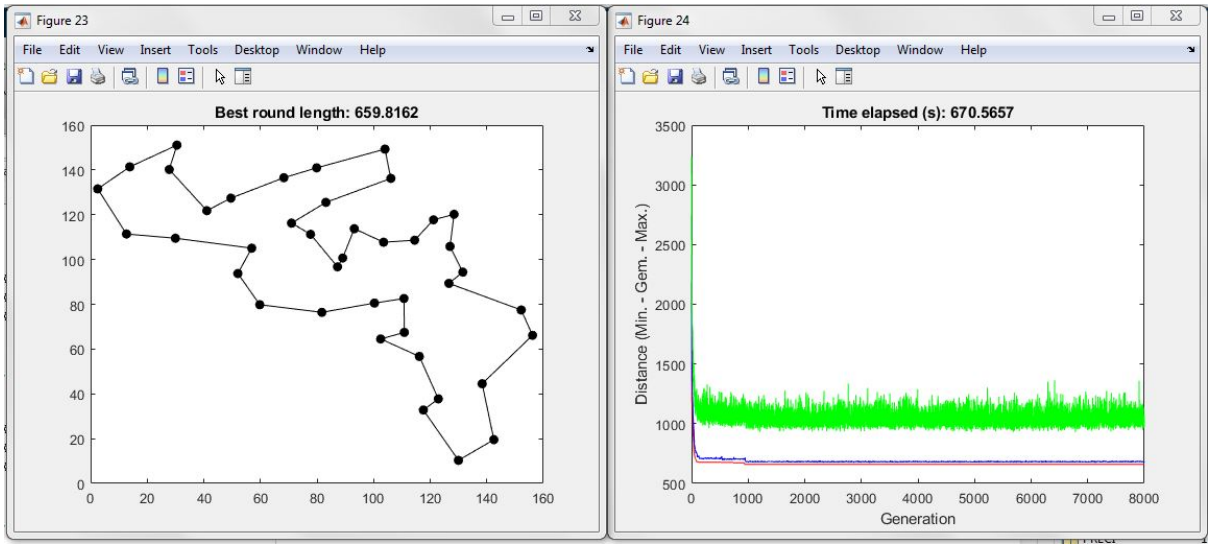
8000 gen run with 600 individuals, 30% elite, 55% XO and 5% scramble mutation: 659.816



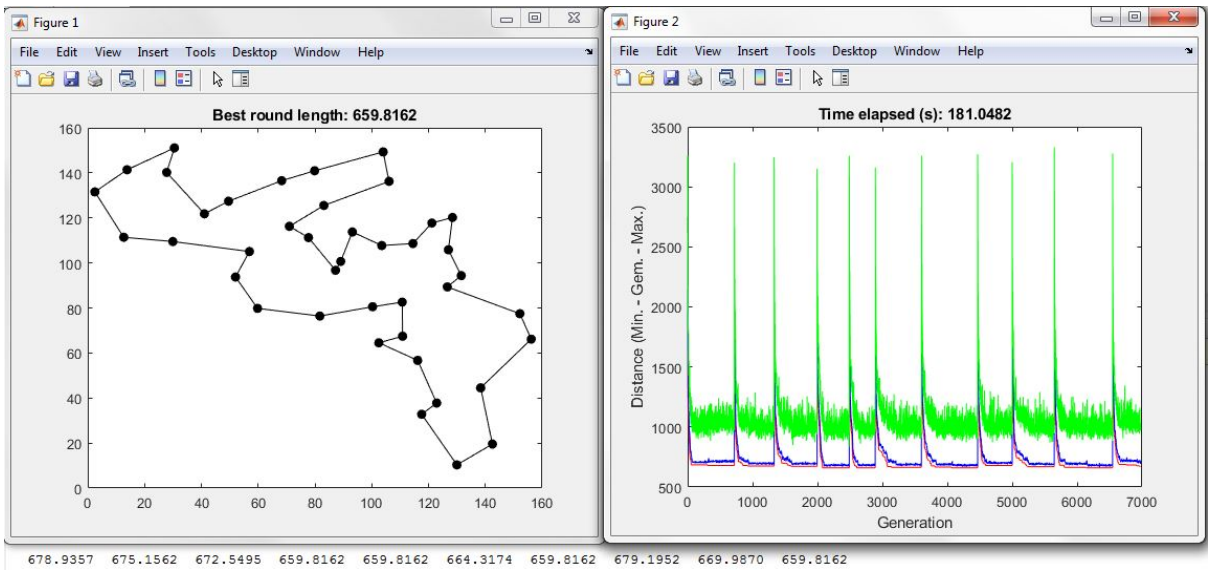
8000 gen run with 200 individuals, 30% elite, 55% XO and 5% s.inversion mutation: 659.816



8000 gen run with 400 individuals, 30% elite, 55% XO and 5% s.inversion mutation: 674.144



8000 gen run with 600 individuals, 30% elite, 55% XO and 5% s.inversion mutation: 659.816



Resetting test: resets after 400 generations of no improvement: 659.816

Task7

First battery of tests

	xqf131.tsp: effect of parent selection			
Selection:	Rank SUS	Fit. Prop.	Tourney 3	Tourney 5
Mean:	1448.6014 ± 73.8687	1431.1215 ± 67.7634	1434.831 ± 78.9882	1429.9666 ± 58.9298

	xqf131.tsp: p-values across different selections			
Selection:	Rank SUS	Fit. Prop.	Tourney 3	Tourney 5
Rank SUS	NaN	0.4084	0.6374	0.4189
Fit. Prop.	0.4084	NaN	0.8871	0.9610
Tourney 3	0.6374	0.8871	NaN	0.8256
Tourney 5	0.4189	0.9610	0.8256	NaN

Second battery of tests

	xqf131.tsp: effect of parent selection			
Selection:	Rank SUS	Fit. Prop.	Tourney 3	Tourney 5
Mean:	1703.1289 ± 75.2572	1694.5057 ± 80.8721	1705.3109 ± 112.7382	1726.1235 ± 66.8885

	xqf131.tsp: p-values across different selections			
Selection:	Rank SUS	Fit. Prop.	Tourney 3	Tourney 5
Rank SUS	NaN	0.7354	0.9480	0.3154
Fit. Prop.	0.7354	NaN	0.7274	0.1799
Tourney 3	0.9480	0.7274	NaN	0.4367
Tourney 5	0.3154	0.1799	0.4367	NaN

References

[1] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, July 1995.

[2] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[3] David H. Wolpert and William G. Macready. What the no free lunch theorems really mean: How to improve search algorithms. In *Ubiquity Symposium: Evolutionary Computation and the Processes of Life*, pages 2:1–2:15, New York, NY, USA, 2013. ACM.