Genetic Algorithms and Evolutionary Computing

Sieben Bocklandt and Ruben Broekx

6th of January

The Traveling Salesman Problem (TSP) was properly written down the first time in 1832, but no concise solution was formulated. Since then, a lot of new mathematical and computer based fields have originated, which led to better and better solution approximations for this well known problem. The problem at hand states that a salesman wants to visit a set of cities exactly once before returning to its starting position, whilst minimizing its total distance travelled.

The main problem with the TSP is that it doesn't scale well for larger datasets, implying a drastic decrease in performance. To handle these larger datasets, more complicated algorithms are needed. *Genetic algorithms* (GA) provide a way to tackle this problem since they provide an efficient way of traversing the search space. The algorithm is based on a population of individuals that each roam the search space in an evolutionary manner. The best performing individuals are used each generation to setup the next generation. By doing so, GA's manage to find close approximations of the solution fairly quick and elegant.

In the first section of this report, the provided algorithm is analyzed to form a baseline for the project. Section 2 extends the catalogue of possible stopping criteria which are used to prevent the algorithm from performing redundant calculations. Next, Section 3 discusses an alternative way of representing the cities in the algorithm, together with appropriate crossover and mutation operators. Section 4 introduces the notion of local heuristics, which try to optimize only select parts of the solution. Section 5 discusses small additions to the algorithm which further increase its performance. The last section, Section 6, bundles all our findings together and compares the performance of our best performing algorithm to that of the baseline. In the end, the our algorithm finds on average a path for the benchmark dataset which is 9.12 times shorter than the baseline when evolved for only one hundred generations. Furthermore, the found path is on average only 11.83% longer than its optimum.

1 Existing genetic algorithm

For this exercise, three experiments were done: the impact that crossover and mutation probabilities have the average distance for a fixed number of generations, the impact of the population size on the time it takes to find the solution, and the impact that the loop detection algorithm has on the number of generations it takes to find the solution.

1. **Data set(s) used & explain why you selected these data set(s)**
   For our tests, we've created our own data set which consists out of forty cities placed equally spaced on a circle that covers both the X and Y axis from 0 to 1. By using this dataset, it's easy to visualize how close the algorithm is to solving the problem (i.e. solution is close to 3.14).

2. **Parameters considered and intervals/values**
   - `Experiment 1 -` This experiment maps out the result of the crossover and mutation probabilities over their full spectrum. All the possible combinations of both mutation and crossover probabilities between zero and one with a step-size of 5% will be used.
   - `Experiment 2 -` This experiment takes the number of individuals into account for a varying size (see Table 1). For the other parameters, the same are used as in the previous experiment, with crossover and mutation probabilities both set to 20%.

- `Experiment 3` - The last experiment takes the influence of loop-detection into account. This experiment is a complete copy of experiment 2 for each of the other parameters.

3. **Performance criteria used**

   - `Experiment 1` - The performance criteria is the smallest distance of the final solution after one hundred generations. Population size equals 128, with elitism 5%, crossover and mutation operators are `xalt_edges` and `inversion` respectively, no loop-detection is used.
   - `Experiment 2` - Since it is a trivial conclusion that larger populations find the solution in fewer generations relative to smaller ones, we've chosen to evaluate this experiment based on the time it takes to find the solution.
   - `Experiment 3` - The last experiment is expressed as the number of generations it takes to find the solution. These results are compared to that of experiment 2 (no loop-detection).

4. **Test results**
   All three the experiments are done 20 times and averaged out, this to prevent outliers.

   - `Experiment 1` - Figure 1 shows the shortest distance after 100 generations for each of the crossover-mutation combinations.
   - `Experiment 2` - Table 1 shows for each of the considered population sizes the average time (in seconds) and number of generations it took to find the solution.
   - `Experiment 3` - Table 2 shows for each of the considered population sizes the average time (in seconds) and number of generations it took to find the solution, as well as the reduction in generations needed compared to experiment 2.

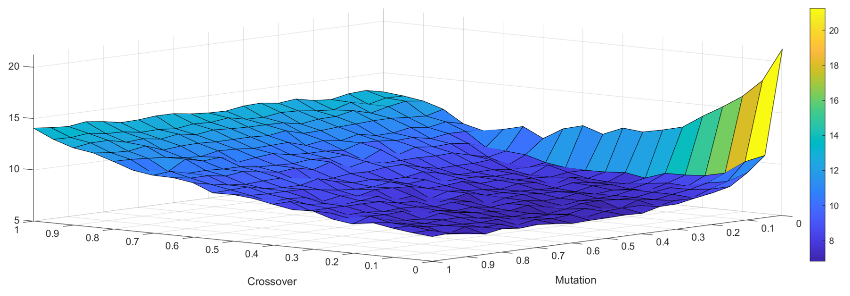


Figure 1: Average distance - crossover vs mutation (lower is better)

Table 1: Duration to find solution relative to population size - no loop-detection

	12	16	24	32	48	64	96	128	192	256	384	512
Time (s)	0.53	0.48	0.40	0.380	**0.377**	0.49	0.50	0.60	0.76	1.00	2.08	2.85
Generations	1678	1331	900	672	459	454	312	288	242	232	223	191

Table 2: Duration to find solution relative to population size - with loop-detection

	12	16	24	32	48	64	96	128	192	256	384	512
Time (s)	0.15	0.14	0.096	**0.091**	0.12	0.10	0.13	0.12	0.18	0.23	0.40	0.60
Generations	336	314	159	121	111	69	63	43	41	38	33	33
Reduction (%)	499	424	566	555	414	658	495	670	590	611	676	579

5. **Discussion of test results**

   - `Experiment 1` - As expected, a mutation rate close to zero doesn't lead to satisfactory results, mainly because this limits the traversal of the search-space in unknown areas. What came as a surprise is that high crossover rates lead to a destructive effect on the performance of the algorithm. Furthermore, the introduction of crossover doesn't even seem to result in significant better results. Another surprising result is to see how little impact the change in mutation rate has when situated between 40% and 90%.

- `Experiment 2 -` Smaller populations seem to converge faster to the optimum. This is because the overhead of evaluating each genome is reduced significantly. Even though a population of size 48 needs almost double the generations that a population of size 256 to find the solution, still, the smaller generations finds the solution almost three times as fast. It can be noted that there is a trade-off between the ability to cover the search-space (more individuals is better) and the ability to evaluate and evolve faster (less is better).
- `Experiment 3 - Loop-detection` has a highly significant effect on the performance of the algorithm since it reduces the number of generations needed by a rough five-to-six-fold. Even though the detection of loops comes at a price, we can safely conclude that the prevention of loops is definitely worth this overhead.

2 Stopping criterion

Stopping criteria are used to prevent the algorithm from computing useless generations. Besides the provided stopping criterion which stops the algorithm when a certain percentage of the population's fitness equals the population's best fitness, two other stopping criteria are implemented.

1. **Stopping criterion**
   - `Threshold -` Stop when the population's minimal distance exceeds a certain threshold (i.e. is smaller or equal to this threshold). This stopping criterion is useful to benchmark the algorithm on data samples for which the minimal distance is known. We've used this criterion in question 1 to determine the number of generations needed to find the solution and will therefore not go further into detail for this criterion.
   - `Stagnation -` This stopping criteria concerns the progress made by the algorithm. If for the last N generations no progress is made, the algorithm will terminate. This criterion has its clear benefits when it isn't known in advance how much generations are needed. Furthermore it can also be used to find a *good approximation* of the solution. It could sometimes take indefinitely more the generations to compute the optimal solution rather than a good approximation (e.g. a 5% longer path).
2. **Test results**
   To test the stagnation criterion, we've run experiments for each of the eleven provided *rondrit* data samples. For this experiment `MAXGEN` was put to infinity and the `STAGNATION` variable (width of the sliding window) is put to one hundred. In other words, if the minimum distance found by the population doesn't decrease after one hundred generations, the algorithm will halt. The goal of this experiment is to see how quick the algorithm will halt and what the computed *shortest distance* is relative to the optimum. The other parameters are: 128 individuals encoded using the `adjacency` representation, `alternating edges` crossover operator with 10% crossover rate, `inversion` mutation operator with 40% mutation rate, elite population of 5%, and `loop-detection` enabled. The other stopping criteria were disabled. Each of the tests are done twenty times and the obtained results were averaged out.
3. **Discussion of test results**
   Table 3 presents the results found for this experiment. In six of the eleven tests, the algorithm finds the solution when being stopped by the stagnation stopping criterion. For the first four test-samples (12, 18, 23, 25), the algorithm finds the solution within the first 60 generations, which shows the benefits of the stopping criterion (i.e. for samples as `127` more than a thousand generations would make sense, but for samples as `12` it clearly doesn't).

Table 3: Stagnation

rondrit	12	18	23	25	48	50	51	67	70	100	127
Generations	113	117	138	154	305	282	303	387	485	702	1006
Found minimum	3.35	2.93	3.24	4.02	4.33	5.96	6.31	4.38	6.82	7.92	6.10
Real minimum	3.35	2.93	3.24	4.02	4.33	5.83	6.22	4.37	6.82	7.84	5.93
Optimum found	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	No	No	**Yes**	No	No

3 Other representation and appropriate operators

For this exercise, we've chosen to compare the performance of the adjacency representation to that of the path representation, with both implementing a suiting crossover and mutation operator. Note that both these representations can be transformed from one to another at each step, meaning that each crossover and mutation operator can be used for both representations. We've also considered implementing the ordinal representation, but research [1] has shown that this representation receives poor results relative to its peers.

1. **Representation**

   - `Adjacency -` The adjacency representation represents a route between cities as a list where each `(index, value)` tuple represents an edge between the outgoing and receiving city respectively. For example, a list `[3,1,2]` would represent the path `1->3->2(->1)`.
   - `Path -` The path representation is the most intuitive representation since a list of cities represents the actual order in which the cities are traversed. For example, a list `[3,1,2]` would represent the path `3->1->2(->3)`.

   Both the representations are supported by the Toolbox out of the box, as well as the transformations between them (`adj2path` and `path2adj`). Note that we've extended these scripts to support full populations instead of only a single genome.

2. **Crossover operators**
   A great help to decide which crossover operator to use was K. Puljic and R. Manger's paper *"Comparison of eight evolutionary crossover operators for the vehicle routing problem"* [2].

   - `Alternating edges (AEX) -` This operator is typically used in the adjacency representation. During crossover, offspring are created by choosing the first edge at random from the first parent and from then on extending the partial tour with the appropriate edge from the other parent. If the addition of a certain edge would result in a cycle, another edge that does not create a cycle will be chosen at random. Two offspring will be created based on two parents each iteration. This operator is supported by the Toolbox (`xalt_edges`).
   - `Heuristic (HGreX) -` This crossover operator uses the path representation. As with the previous operator, the algorithm starts by selecting a random city. From then on, the edges connected to the current city are compared between both parents and the shortest edge is chosen. If both edges lead to a cycle, the shortest edge (of all possible edges) that doesn't lead to a cycle is chosen. A single offspring is created based on two parents each iteration. Although simple, the heuristic operator performs similar or even better than the other operators in related studies [2]. This operator is a small variation of the `sequential constructive crossover (SCX)` operator [3] since the heuristic operator will only search for a substituting edge when both parents fail. The script `heuristic_crossover` implements this operator.

3. **Mutation operators**
   All three mutation operators considered use the path representation, which implies that a chromosome encoded in the adjacency representation must first be transformed to the path representation and decoded back the the adjacency representation after mutation.

   - `Inversion -` This operator will select two cities at random and reverse the order in which the cities and all the intermediate cities appear (textbook page 69). This operator is interesting for our problem since the majority of the sequence's order will be preserved. The Toolbox already has this operator implemented (`inversion`).
   - `Reciprocal exchange -` This mutation operator will swap the positions of two cities at random (textbook page 69). This operator is already supported by the Toolbox via the script `reciprocal_exchange`, which is why we've chosen to evaluate the performance of this operator as well.

- `Scramble` - As an extra, we've added the scramble operator as well (textbook page 69), which is implemented in the `scramble` script. There is no real motivation in why we've chosen to implement this operator besides the fact that we would feel guilty if we didn't implement a mutation operator ourselves.

Our hypothesis is that inversion will outperform the other two operators since the majority of the sequence is kept by only reversing a sub-sequence (see it as twisting a part of the map upside-down). We think that the `scramble` operator will perform worst because it's a very stochastic process that has the potential to change much in a chromosome's state. Reciprocal exchange is expected to perform somewhere in between since it doesn't take the sequences into account as inversion does, but has less impact on the overall order as with the `scramble` operator. Because only two cities are swapped, we expect the impact of this operator being limited, hence we expect it to perform more similar to `inversion` as to the `scramble` operator.

4. **Parameter tuning: parameters considered and intervals/values**
As previously mentioned, at each step the `path` representation can be transformed to the `adjacency` representation and vice versa. This means that each of the discussed operators can be used for both representations. To keep things simple, we've decided to couple the crossover operators to their corresponding representations, and compare each of the mutation operator in combination with each of the crossover operator.

For each of the experiments, a maximum generation of one hundred is used with no other stopping criteria. The population size equals 128 with 5% elitism. Although the clear benefit of using `loop-detection` has been shown in previous experiments, we've chosen to disable it in these experiments since we only want focus on the influence of the operators. Four different crossover (left) and mutation (right) probability combinations will be tested: (20%, 20%), (40%, 20%), (20%, 40%), and (40%, 40%).

5. **Data set(s) used**
We've chosen to do our experiments on the provided *rondrit* data samples. We've chosen these samples since they provide a good variety in the number of cities used and in the map's topology.

6. **Test results**
In a first experiment, each data sample is evaluated twenty times for each possible crossover-mutation probability-combination. For each data sample, the geometric mean is taken, which is then compared to the real optima of the sample. From this comparison we conduct a *performance-score* which expresses how much longer (%) the found solution is compared to the real optimum. For example, if a distance of 1.1 is found and the optimal distance is equal to 1, a performance-score of 110 is obtained since the found solution is ten percent longer. If the shortest distance is found, the performance-score is equal to 100. Each of the performance-scores for all the *rondrit* samples are then averaged which results in the overall performance-score. The results of this experiment are given in Tables 4, 5, 6, and 7.

Table 4: Crossover 20%, Mutation 20%

	inversion	swap	scramble
AEX	187.9	190.8	210.9
HGreX	**105.1**	106.2	107.1

Table 5: Crossover 20%, Mutation 40%

	inversion	swap	scramble
AEX	184.5	192.3	222.4
HGreX	**103.5**	105.4	105.9

Table 6: Crossover 40%, Mutation 20%

	inversion	swap	scramble
AEX	188.0	190.8	209.1
HGreX	**102.8**	103.6	103.7

Table 7: Crossover 40%, Mutation 40%

	inversion	swap	scramble
AEX	191.3	198.1	221.5
HGreX	**102.3**	103.5	103.6

Appendix B contains two visualizations of the previously discussed experiment. In this appendix, Figure 2 show the effect map-complexity has on the performance of the `HGreX` operator. The second visualization, Figure 3, shows the impact each of the mutation operators has when used alongside the `HGreX` crossover operator.

A second experiment checks if the transformation from `adjacency` to `path` (or vice versa) has a significant influence in the calculation time of the algorithm. Since all three of the mutation operators are based on the `path`-representation, the overhead of transforming from `adjacency` to `path` and back is the same for each of them (note that this transformation happens in the higher level script `mutateTSP`). The only difference is in the crossover operators. To test this overhead that comes with the `path2adj` and `adj2path` functions, we've compared the calculation times between the two representations and the two crossover operators for the `rondrit127` data sample. We ran for each combination the algorithm one hundred times over a thousand generations. The results from this experiment are shown in Table 8.

Table 8: Time overhead representation transformation

	adjacency	path
AEX	18.44	18.83
HGreX	22.00	21.58

7. **Discussion of test results**
The first experiment clearly shows the benefit of using the `heuristic crossover` operator (`HGreX`) over the `alternating edges` operator (`AEX`). In the *worst case* comparison, `HGreX` outperforms `AEX` by a stunning 80%. Furthermore, `HGreX` consistently finds good approximating solutions to the real optimum within one hundred generations, with it's worst result being only 7% longer than the optimal solution. When we look at the mutation operators, we see that our hypothesis holds. It is indeed the case that `inversion` consistently outperforms the other two operators and that the `scramble` operator is in general the worst operator out of the three. When comparing the four tables, we see that the `HGreX` operator benefits the higher crossover probability where the `AEX` operator does not. All three the mutation operators benefit more from the 40% mutation probability then they do from the 20% probability.

From the second experiment we can conclude that there is no significant time difference between the two representations. This was to expect since the transformation happens rather effortless (i.e. O(n^2) since each genome is transformed separately with a linear time-complexity) in comparison to other parts of the algorithm. Based on these measures, we conclude that there is no real benefit to use one representation above the other since all the implemented operators (both crossover as mutation) can be used with either representation. Personally, we prefer using the `path` representation simply since it's easier to comprehend.

4 Local optimisation

For this task, we've implemented and compared two local heuristics: `2-opt` and `inversion`. Each heuristic will bring a change to the chromosome's configuration, after which this configuration will be checked. If the changed chromosome performs better than its original, the change is kept. Otherwise, the chromosome gets reverted back to its original.

1. **Local optimisation heuristic & explain why you selected this heuristic**
   - `2-opt -` This operator, also called the *four vertices and three lines inequality* [4], will change two neighbouring cities and checks if this change was beneficial or not. The algorithm traverses each genome sequentially and halts after the first beneficial change. We've implemented this operator since we noticed that often a "*two city swap*" occurred during evolution. This operator is implemented in the `two_opt` script.
   - `inversion -` This is the same as the `inversion` (mutation) operator, with the only difference that this time the inversion will be reverted if it isn't beneficial. We've decided to include this operator since related research [4] shows the benefit of this operator.

2. **Test results**
   The data used for this experiment were all the `rondrit` samples. We kept a similar configuration conform with previous experiments: each data sample is evaluated twenty times over one hundred generations. Afterwards, the geometric mean is taken to express the algorithm's performance on a certain data sample. This intermediate performance expresses how much longer the found solution is relative to the data sample's global optimum. The overall performance is determined as the average performance for each of the data samples. The algorithm is configured as follows: 128 individuals with 5% elitism, `path` representation with `HGreX` and `inversion` operators, both used with a 20% probability. `loop-detection` and `stopping criteria` are disabled.

3. **Discussion of test results**
   Table 9 shows that each local heuristic results in a significant increase in performance. The *improvement* is calculated as the relative performance increase towards the optimum to that of the *no local heuristic* scenario. Figure 4 of Appendix C gives a more in depth comparison between the *no local heuristic* and *both local heuristics* for the `rondrit 016` and `rondrit 127` samples. Although the local heuristics have a positive impact on the `rondrit127` data sample, this impact isn't as pronounced as for the `rondrit016` sample.

Table 9: Performance increase by choice of local heuristic

	off	2-opt	inversion	both
performance	104.30	103.67	102.70	**102.20**
improvement	0%	15%	37%	**49%**

5 Other task(s)

We've decided to implement all four of the optional tasks. Before addressing these additions, a brief discussion is given on how the experiments are done, since this is the same for each of the four additions. We've decided to discuss this section first before heading into the benchmarks since some methods implemented here are used in our final algorithm.

5.1 Description of the experiments

For each of the experiments, the *rondrit* data samples are used to evaluate the algorithm's performance. An approach similar to previous experiments is chosen since each data sample is evaluated twenty times. Afterwards, the geometric mean is taken to express the algorithm's performance on each specific data sample. This performance metric expresses how much longer the found solution is relative to the data sample's global optimum. Each experiments uses a population of size 128 with 5% elitism, represented using the `path` representation. `HGreX` is used as the crossover operator and `inversion` as the mutation operator, both with a 20% probability. None of the stopping criteria are used.

5.2 Parent selection alternatives

1. **Description of implementation**
   - `Scaling` - This method scales the fitness of each genome with a value based on the best, worst and mean fitness of the population. Instead of using the fitness of the genome normalized by the total fitness of the population, *windowing* is used. This method is an answer to the limited selection pressure that occurs when the fitness of two genomes is almost the same. The fitness function is defined as follows:

$$f = \begin{cases} \frac{mean}{max-mean} * f + \frac{mean*max-2*mean}{max-*mean}, & \text{if } min > 2*mean - max \\ \frac{mean}{mean-min} * f - \frac{min*mean}{mean-min}, & otherwise \end{cases}$$

     This method is implemented in the `scaling` script.

- `Tournament` - This selector selects a given number of parents by randomly choosing two genomes and obtaining the most fit one. This is repeated until a predefined threshold is crossed. Note that the selection of genomes happens at random, which implies that a genome can be chosen multiple times. The implementation is based on page 84 in the book and can be found in the `tournament_selection` script.

2. **Description of experiments**
A hypothesis we have is that `scaling` will perform best since it takes the relative difference in fitness into account, where the two other selectors do not. As noted earlier, `scaling` suffers from a low selection pressure when two candidates perform similar to each other, which would be a counterargument for its performance. However, as already discussed, *windowing* should resolve this problem. Although we think that `scaling` performs best, we still wanted to implement `tournament` selection since this selector operates stochastic where the other are deterministic selectors. Since each of the data samples stagnate rather quickly, an evolution of only one hundred generations will suffice for this experiments.

3. **Test results**
Table 10 shows the performance for each of the three parent selectors.

Table 10: Performance by choice of parent selection

	Ranking	Scaling	Tournament
performance	104.85	**103.63**	105.51

4. **Discussion of test results**
As shown in Table 10, `scaling` finds a solution that is only 3.63% longer than the optimum, which is a significant increase in performance over the baseline's 4.85%. The added randomization introduced in the `tournament` selector didn't seem to bring any benefit to its performance. Figure 5 in Appendix D compares the selectors for each generation on the `rondrit 016` and `rondrit 127` data samples

5.3 Survivor selection alternatives

1. **Description of implementation**
As alternative for the provided `elitism` survivor selector, we chose to implement the `round-robin` method. Instead of taking the x fittest genomes, the algorithm goes over the population whilst letting each genome "fight". In this fight, the fitness of the genome is compared to the fitness of ten random genomes. For each of these that the genome has a strictly better fitness, it gets a point. In this way, the whole population is mapped to a score between 1 and 10. The x survivors are the x genomes with the highest score. This method is implemented in `round_robin` and is based on page 89 in the book.

2. **Description of experiments**
Since `elitism` is deterministic, only the most fit candidates will be kept to the next generation. Due to `round-robin`'s stochastic nature, there is still the chance that a less fit candidate will be chosen, which can benefit the population's diversity. The algorithm is executed one hundred generations for each run.

3. **Test results**
Table 11 shows the performance for the two survivor selection methods.

Table 11: Performance by choice of survivor selection

	Elitism	Round-robin
performance	104.18	**103.68**

4. **Discussion of test results**
   `Round-robin` seems to be an improvement over `elitism`, but only incremental. This is most likely since the probability of an elite candidate being kept in the `round-robin` selector remains high. Figure 6 in Appendix D shows the distance by generation for both the `rondrit 016` and `rondrit 127` data samples.

5.4 Preserving population diversity

1. **Description of implementation**
   Preserving population diversity longer could benefit the algorithm since more useful crossovers can be done in later generations. Curious about the difference between two different approaches, we implemented both *Crowding* and *Islands*.

   - `Crowding -` This algorithm tries to keep diversity by changing the way the offspring is reinserted into the population. Instead of just reinserting all the genomes in the population, they will have to contest their place with their parent genomes. This is done by making children-parent pairs using the genome distance (shared edges [5]). The fittest of each pair will be reinserted into the population. This is called *deterministic crowding* [6] and is based on page 93 in the book. The script `crowding` implements this code.
   - `Islands -` Instead of having one population, islands are built on having multiple subpopulations, which are numbered and separated from each other. This implies that each of the population operators works on each subpopulation individually. Genomes are shared between subpopulations each twenty generations by default. This swap is done in a neighbourhood-like fashion, which means that each subpopulation will swap with its left and right neighbour. When a swap occurs, 20% of the genomes will be swapped which are chosen by `fitness proportional selection`. This algorithm is based on page 95 in the book.

2. **Description of experiments**

   - `Crowding -` The implemented algorithm will enforce genomes to stay in a local maximum. Our hypothesis is that this will increase the diversity, as not all individuals will be "pushed" towards the same genome. In this experiment, one hundred generations are used.
   - `Islands -` For the experiment on islands, it is important that the subpopulations contain enough individuals to preserve the effect of crossover after a number of generations. A higher population size also increases the chance of beneficial mutations. To meet both these remarks, we used 256 individuals for this experiment. A last point to take into account is that the effect of the swaps will be most visible when stagnation occurs, which can be enforced by using loop detection and increasing the number of generations to two hundred. One big advantage of using subpopulations is that it can be parallelized, without significant loss of performance.

3. **Test results** Table 12 shows the performance for each of newly introduced methods.

Table 12: Performance by choice of diversity keeping method

	no measure	Crowding	2 islands	4 islands	8 islands
performance	101.23	105.96	**100.93**	100.97	101.05

4. **Discussion of test results**

   - `Crowding -` As can be seen in table 12, trying to preserve diversity by using `crowding` affects the performance in a negative way. Our hypothesis was wrong, as the algorithm converges a lot quicker with `crowding`, which is the opposite of preserving diversity. Figure 7 in appendix D visualizes this.
   - `Islands -` Using 8 subpopulations increased the performance and had a slower convergence, as expected. Figure 8 in appendix D shows this behaviour.

5.5 Adaptive parameter control strategy

1. **Description of implementation**
The main problem we had with our current algorithm is that for harder problems, the algorithm tends to converge towards a local optimum rather quickly from which it hardly escapes. The main reason why is due to a lack of diversity in the population. When fully converged, the best performing genome dominates the population which limits the exploration towards others optima. To enforce the exploration of new genome-configurations, we've implemented a form of *adaptive mutation*, which is enabled in the algorithm by enabling the Boolean `ADAPTIVE_MUT`. When enabled, an additional mutation happens before crossover (`mutateAdaptiveTSP`). The algorithm iterates over all the genomes and remembers each of the genome-configurations it has seen. When a certain genome-configuration occupies more than 5% of the population, the following equally configured genomes will be mutated an arbitrarily number of times based on the mutation probability. The likelihood that a genome is mutated at least `X` times is equal to: $P(\#mut \geq X) = P(pr\_mut)^X$. We've deliberately chosen to implement this extra mutation mechanism before crossover to make the crossing of different genomes more likely, hence enforcing exploration.

2. **Description of the experiments**
Since the effect of this mechanism is targeted towards escaping suboptimal results, we've decided to use datasets in which it is likely to obtain these suboptima. Hence, we've chosen to use the provided *benchmark* dataset (`bcl380`, `belgiumtour`, `rbx711`, `xqf131`, `xql662`). The best configuration found up till now (excluding other methods discussed in this section) is used on a population with size 128 that ran for two hundred generations (hence providing more possibilities to search in the stagnated search-space, compared to the previously used one hundred generations). The other parameters are configured as such: path representation, `HGreX` crossover (20%), inversion mutation (20%), and loop-detection enabled. Stopping-criteria are disabled to give the algorithm enough time to improve in a stagnated scenario.

3. **Test results**
Table 13 visualizes the impact *adaptive mutation* has on the performance of our algorithm. The values in the *enabled* and *disabled* rows denote how much longer the found solution is relative towards the optimal solution. The *performance improvement* is calculated as the relative improvement between *disabled* and *enabled*: $\frac{disabled-enabled}{disabled-optimum}$.

Table 13: Performance increase due to introduction adaptive mutation

	bcl380	belgiumtour	rbx711	xqf131	xqf662
disabled	119.30	101.91	124.32	108.23	126.27
enabled	117.24	101.62	122.34	107.93	125.76
improvement	11%	15%	8%	4%	2%

4. **Discussion of test results**
Since this method only influences the stagnated part of the evolution, the performance increase is only limited, but, as shown in Table 13, still significant on average.

5.6 Improving GUI

As an additional task, we've decided to update `tspgui` to support all our newly introduced features, as well as to give us the freedom to compare different configuration efficiently. Figure 9 in Appendix D shows the result of this.

6 Benchmark problems

1. **List of benchmark problems**
   We used each of the five provided benchmarks for the experiments, which are: `bcl380`, `belgiumtour`, `rbx711`, `xqf131`, `xql662`.

2. **Test results**
   To keep the consistency within this report, we used the same approach for the benchmark problems. We 've run experiments over all five of the benchmark data samples. Each data sample was run twenty times with three hundred generations. For each data sample, the geometric mean is taken, which is then compared to the real optima of the sample. This experiment compares the provided baseline with our best performing configuration. The difference between the two is visualized in Table 14. Both configurations use a population size of 128 with 5% elitism, represented using the `path` representation. As mutation operator, `inversion` is used with a 40% probability. No crowding and subpopulations are used, since these didn't seem to contribute to the algorithm's performance.

Table 14: Performance with and without improvements.

	crossover	crossover %	loop detection	adaptive mutation	parent selection	survivor selection	local heuristics
no impr.	AEX	10%	off	off	ranking	elitism	off
all impr.	HGreX	40%	on	on	scaling	round-robin	on

Table 15: Performance with and without improvements.

generation	without improvements	all improvements
5	1274.05	**244.18**
10	1244.29	**142.70**
25	1179.97	**119.34**
50	1113.49	**114.85**
100	1019.81	**111.83**

3. **Discussion of test results**
   Table 15 shows the significant increase in performance our algorithm obtains relative to the baseline. We noted that the algorithm with improvements converges a lot faster than the default one, as shown in Figure 10 (Appendix E).

References

[1] P. Larranage and C. Kuijpers. Genetic algorithms for the travelling salesman problem: A review of representations and operators.

[2] K. Puljic and R. Manger. Comparison of eight evolutionary crossover operators for the vehicle routing problem.

[3] Zakir H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator.

[4] K. Borna and V. H. Hashemi. An improved genetic algorithm with a local optimization strategy and an extra mutation level for solving traveling salesman problem.

[5] Daniel Angus. Crowding population-based ant colony optimisation for the multi-objective travelling salesman problem.

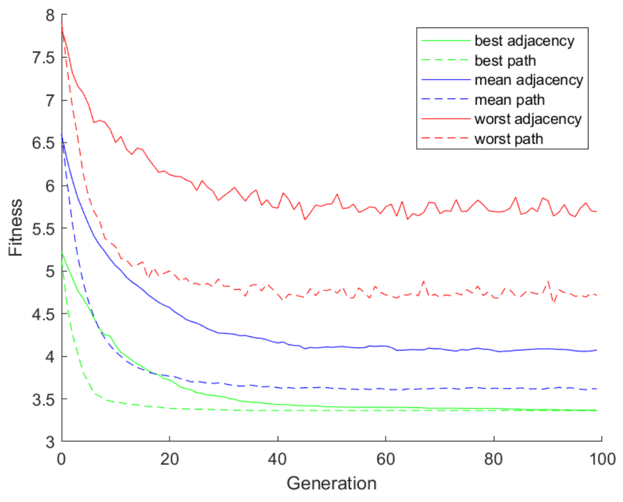
[6] Severino F. Galan and Ole J. Mengshoel. Generalized crowding for genetic algorithms.

Appendices

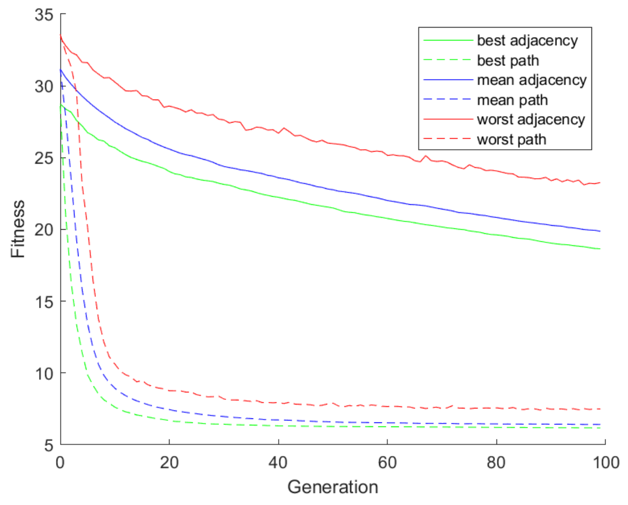
A Time spent on the project

- **Sieben Bocklandt**
  Hours worked on the project: 43
  - Helped with the second exercise (*Stopping criterion*)
  - Helped with implementing the fourth exercise (*Local optimisation*)
  - Implemented fifth exercise parts one to three (*Parent selection*, *Survivor selection*, *Preserving diversity*) and wrote the report for it
  - Implemented the sixth exercise (*Benchmark problems*) and wrote the report for it
  - Revisited and gave feedback on other parts of the report
- **Ruben Broekx**
  Hours worked on the project: 55
  - Implemented the first exercise (*Existing genetic algorithm*) and wrote the report for it.
  - Implemented the second exercise (*Stopping criterion*) and wrote the report for it.
  - Implemented the third exercise (*Other representation and appropriate operators*) and wrote the report for it.
  - Helped with implementing the fourth exercise (*Local optimisation*) and wrote the report for it.
  - Helped with the implementation of the islands for the fifth exercise's third task (*Preserving diversity*).
  - Implemented the fifth exercise's fourth task (*Adaptive parameter control*) and wrote the report for it.
  - Created the test-interface to do experiments efficiently
  - Rewrote `tspgui` to improve visualizations and to provide an interface for all our implemented functionality.
  - Updated `run_ga` such that given parameters were optional (handled via a *containers.Map*), which improved the testing-experience.
  - Revisited and gave feedback on other parts of the report

B Representation experiment visualizations

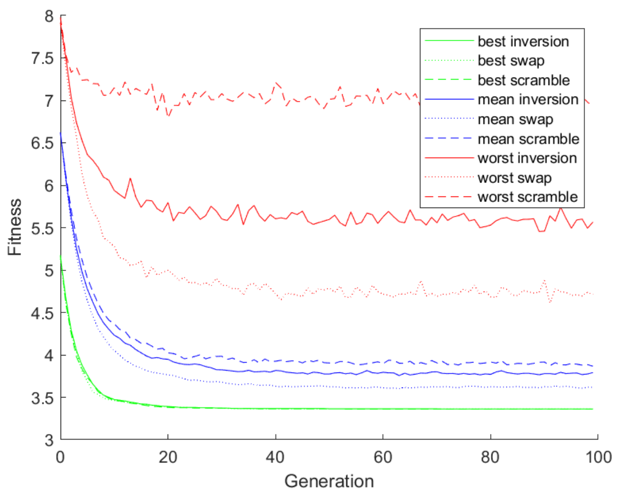


(a) Rondrit - 016

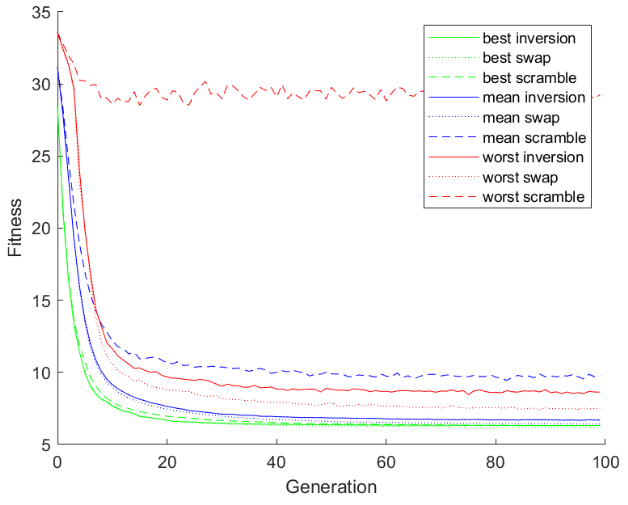


(b) Rondrit - 127

Figure 2: Visualization of the impact the heuristic crossover operator `HGreX` has on the fitness convergence. The difference between the two subfigures visualizes the influence the map-complexity, expressed as the number of cities, has on the performance of the operator. It can be noted that `HGreX` is a clear improvement over `AEX`.



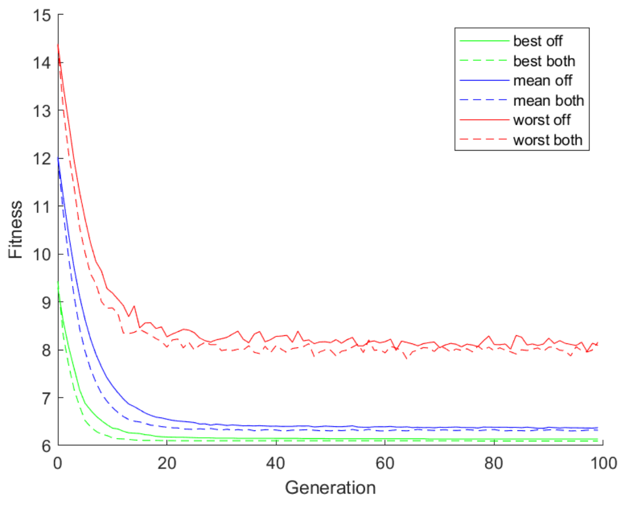
(a) Rondrit - 016



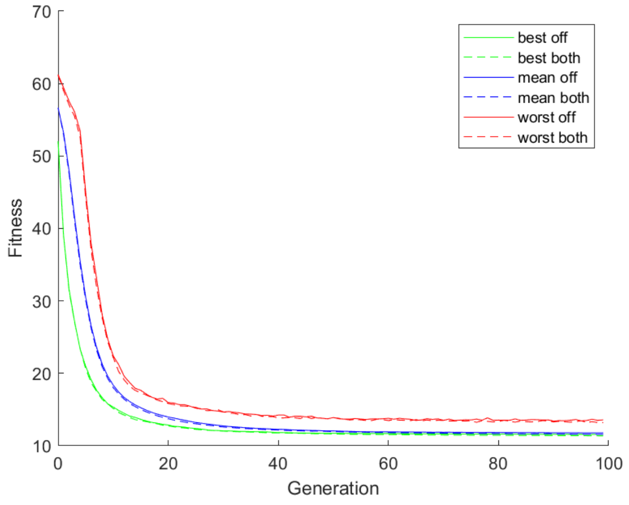
(b) Rondrit - 127

Figure 3: Visualization of the impact each different mutation operator has in collaboration with the `HGreX` crossover operator, both the `inversion` as the `swap` operator outperform the `scramble` operator clearly

C Local optimisation



(a) Rondrit - 016

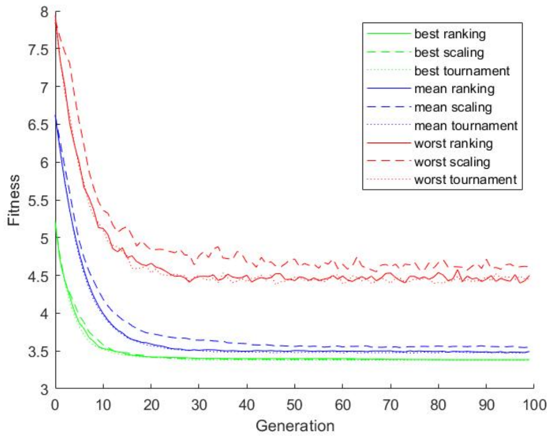


(b) Rondrit - 127

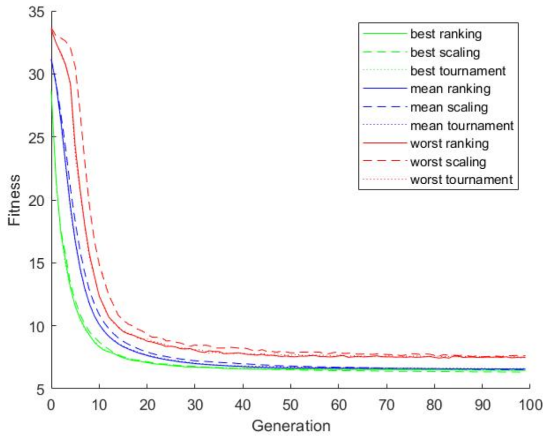
Figure 4: Visualization of how pronounced the impact of the combined local heuristic is relative to the generation in the evolutionary process. Since the local heuristic affects only small parts of the genome in a positive way, this effect is more pronounced for smaller data samples

D Other tasks

D.1 Parent selection



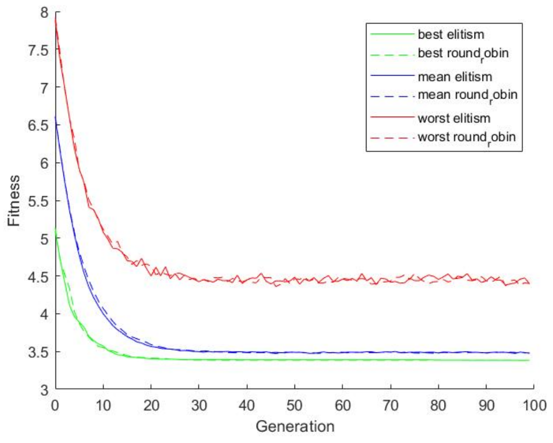
(a) Rondrit - 016



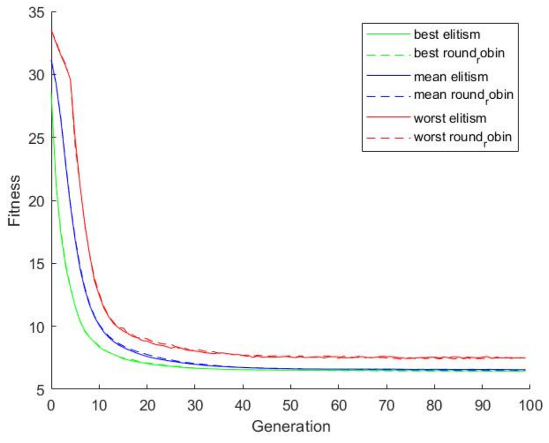
(b) Rondrit - 127

Figure 5: Visualization of the difference between the three parent selection methods: scaling, ranking and tournament selection. No significant difference between the 3 methods is visible.

D.2 Survivor selection



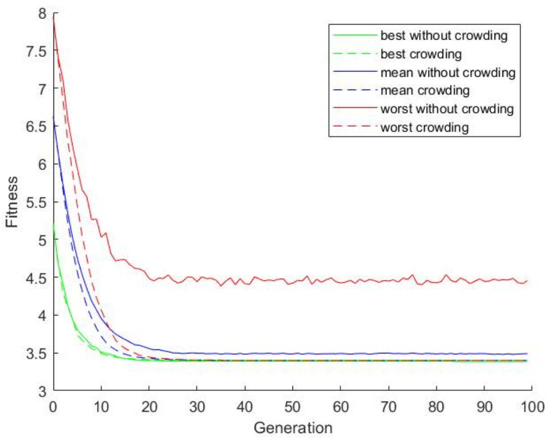
(a) Rondrit - 016



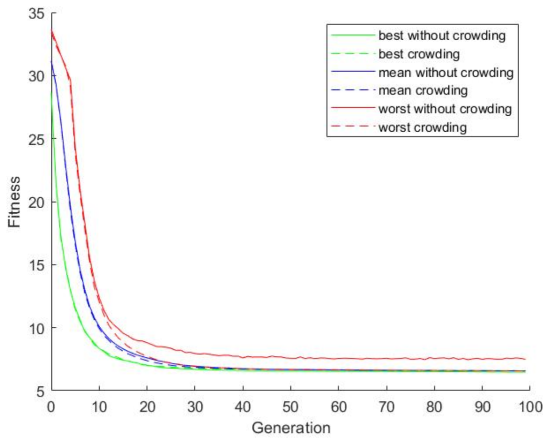
(b) Rondrit - 127

Figure 6: Visualization of the difference between the two survivor selection methods: elitism and round-robin. No significant difference between the two methods is shown.

D.3 Preserving diversity

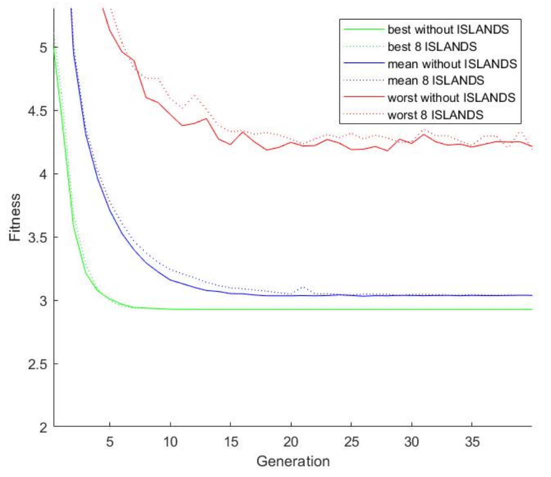


(a) Rondrit - 016

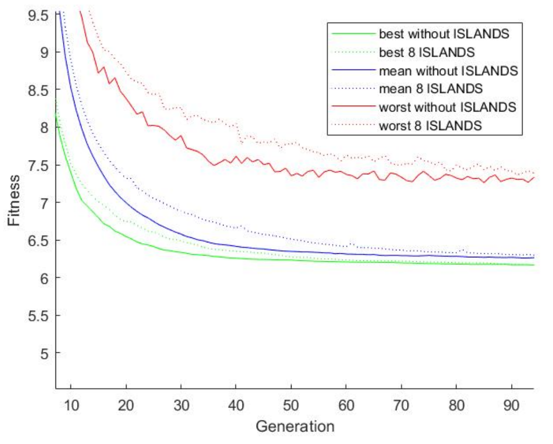


(b) Rondrit - 127

Figure 7: Visualization of the impact of crowding on the performance. No significant difference between crowding and its baseline can be seen.



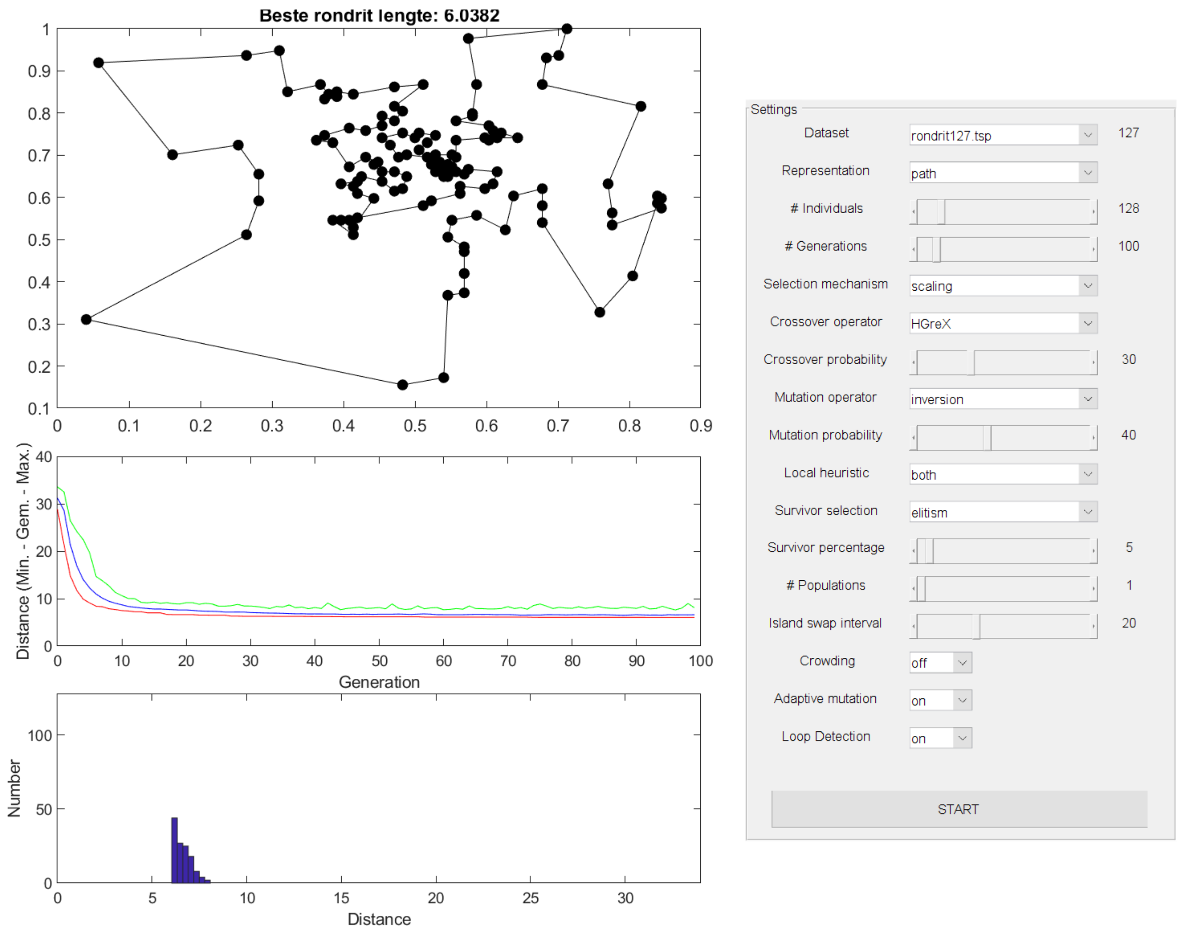
(a) Rondrit - 016



(b) Rondrit - 127

Figure 8: Visualization of the impact of eight islands on the performance. In these plots, it is visible that after each twenty generations the subpopulations swap individuals, indicated by the short spikes in their mean score. Furthermore, it can be noted that the islands stagnate slightly slower, but cross the single-island baseline before fully converging.

D.4 Improving GUI

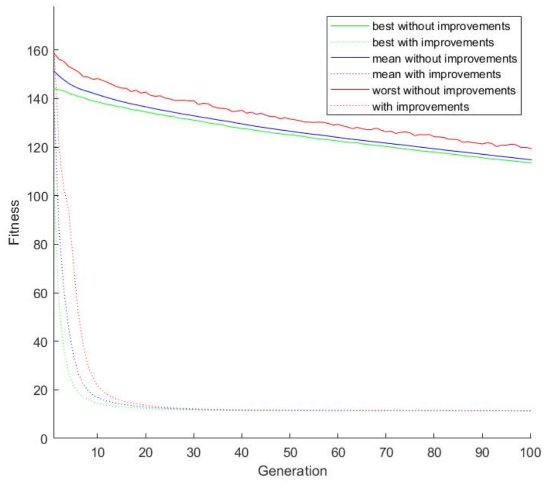


Settings

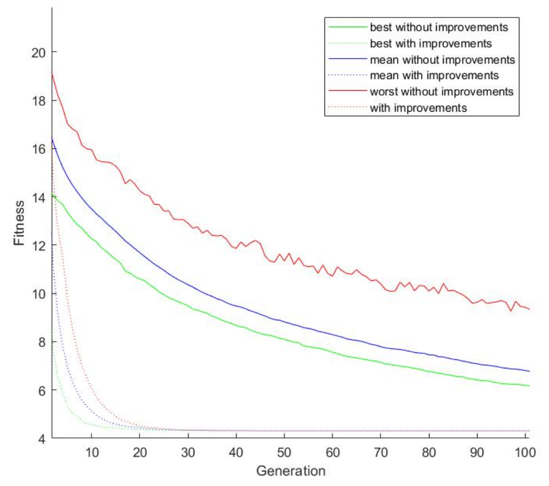
Dataset	rondrit127.tsp	127
Representation	path	
# Individuals		128
# Generations		100
Selection mechanism	scaling	
Crossover operator	HGreX	
Crossover probability		30
Mutation operator	inversion	
Mutation probability		40
Local heuristic	both	
Survivor selection	elitism	
Survivor percentage		5
# Populations		1
Island swap interval		20
Crowding	off	
Adaptive mutation	on	
Loop Detection	on	

Figure 9: Redesign of `tspgui` to support each of the newly introduced features.

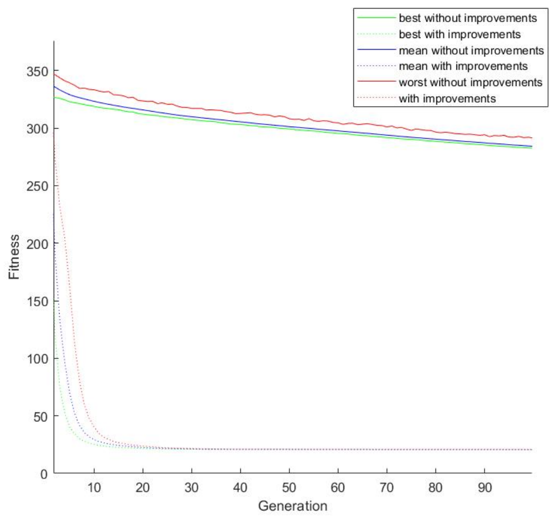
E Benchmarks



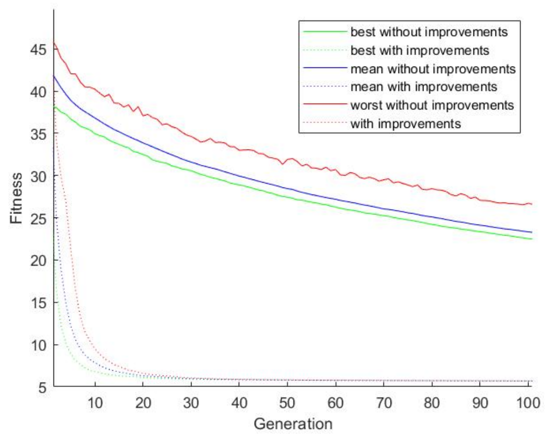
(a) bcl380



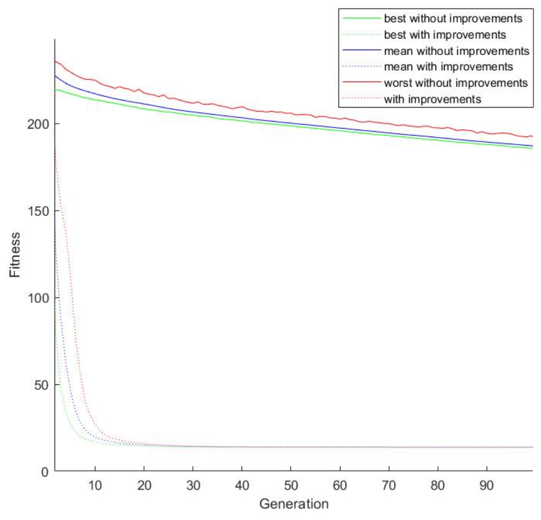
(b) belgiumtour



(c) rbx711



(d) xqf131



(e) xql662

Figure 10: Visualization of the difference in performance between the adapted algorithm and the baseline for each of the given benchmark problems.