Genetic Algorithms and Evolutionary Computing
Project Report

Benedek József Tasi & Koralp Catalsakal

January 5, 2020

1 Existing genetic algorithm

Perform *a limited set* of experiments by varying the parameters of the existing genetic algorithm (population size, probabilities, ...) and evaluate the performance (quality of the solutions).

1. Data set(s) used & explain why you selected these data set(s)

   We have performed the initial experiments with the existing TSP-solver genetic algorithm implementation on two of the provided datasets, namely rondrit25 and rondrit100. The reason for selecting these is straightforward: testing various parameter combinations results in running the simulation a great number of times, so to speed up the process and keep it computationally inexpensive, testing on a small dataset (25 cities) is preferable. However, the effectiveness of the algorithm, and thus the quality of the results can possibly gradually change with the size of the dataset, therefore, testing on a larger dataset (100 cities) is also required.

2. Parameters considered and intervals/values

   Our aim in this section was to test various different parameter configurations, to get an idea about the expected behavior of the algorithm under each different condition. Table 1 contains the parameter configurations we were testing the original GA implementation on (parameters hold with the larger dataset of 100 cities: for the smaller dataset of 25 cities, the number of individuals was tested at 10 and 50 (used throughout the rest), and the number of generations at 50 and 100 (used throughout the rest).

Tests	#Individuals	#Generations	Pr Mutation	Pr Crossover	Pr Elitism
I+G+C+M Test	50	50	10	90	5
I+G+C+M Test	200	200	10	90	5
I+G+C+MTest	10	50	10	90	5
C+M Test	200	200	90	10	5
C+M+E Test	200	200	90	10	0
C+M+E Test	200	200	10	90	0
C+M+E Test	200	200	90	10	90
C+M+E Test	200	200	10	90	90
C+M+E Test	200	200	90	10	1
C+M+E Test	200	200	10	90	1

Table 1: Tested parameter values for the initial GA implementation, with dataset *rondrit100*.

3. Performance criteria used

   To measure performance, we were mainly examining 3 properties:

   - The final fitness value, that is, the best found route length as an end result, averaged over several trials (MBF);
   - The speed of convergence, e.g. how fast does the algorithm reach our acquired optimum;
   - And the nature of convergence, meaning the expression of curve smoothness or a step-like structure in the time-fitness plot.

4. Test results

   For testing purposes, a simple, looping encapsulating algorithm was implemented, which runs the genetic algorithm with the specified parameter configuration 5 times, logs the results in a file, and plots the final fitness curves in a shared figure to facilitate comparison (**plotAllResults.m**, dummy test data is attached in **test_results.txt**). The results of the initial testing process were plotted with this method, as shown in Figure 1. This loop algorithm will also be used throughout the report for result evaluation purposes. Aside from this, the minimal, maximal and average fitness value plot of the GUI (top right corner) was also examined to formulate the observations in the discussion sections.

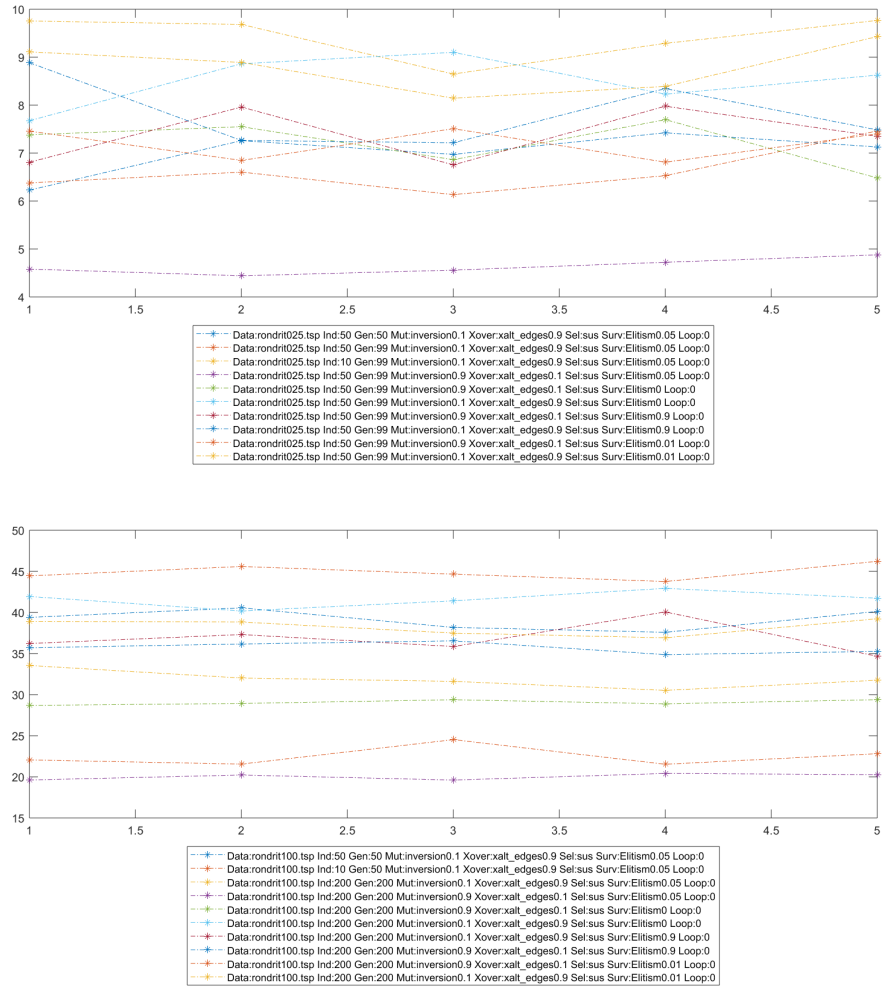

Figure 1: Test results of the initial GA testing. Fitness values of rondrit25 (top) and rondrit100 (bottom), each performed on the specified parameters 5 times.

5. Discussion of test results

   Generally, we can observe that larger datasets require running through more generations before converging to an optimum. The required number of generations can vary, since after a while the curve becomes flat, when the algorithm essentially converged. This is where a stopping criterion could be utilized to prevent computing rather unnecessary iterations.

   Using a larger population, and thus increasing the search space generally makes the results gradually better, and speeds up convergence, following a logarithmic pattern: if we raise the number of individuals beyond a set point, this increase will become less prominent. Using too many individuals thus makes no difference in result improvement above a certain point, but increases computational costs (gradually reduces speed). Lack of individuals, however, strongly affects the effectiveness of the outcome: the patterns will get much more random, as the effect of each crossover and mutation operation increases significantly, and result quality will be reduced.

Regarding the crossover and mutation probabilities, we can observe that a high mutation percentage with low crossover leads to better results than the opposite, and even than using 50-50%. Generally, if crossover>>mutation, we get a slowly converging, ineffective, staircase-like pattern with abrupt steps, while if mutation>>crossover (which is effectively a much more randomized search, that better explores the search space), the convergence is much smoother (c>>m reaches the same result in 1000 generations as m>>c does in 100). This is assumed to be caused by a sub-optimal default crossover function, as xalt_edges does not prioritize the preservation of parent sequences (edges): we expect improvement after the completion of our main task.

High elitism causes a very early convergence to a bad result, as expected, since a selected best proportion of the parent population will be preserved throughout the entire process, reducing diversity and inhibiting improvement. No elitism, however, results in a larger degree of stochasticity: with mutation>>crossover, it stays convergent, but with a much more random pattern, with crossover>>mutation, the high-amplitude oscillation is so random that there is barely any observable convergence. Using 1% elitism already makes a large difference, resulting in a smoother curve with better results, as the best individuals are already preserved. The optimal amount was found to be around 5%: increasing it any further will not improve the results significantly, and after a while, decrease in effectiveness will be observable. This is in line with our expectations, as the few most fit individuals will represent the best currently attainable fitness, and keeping them alive for further reproduction facilitates finding an optimal solution.

2 Stopping criterion

Implement a stopping criterion that avoids that rather useless iterations (generations) are computed.

1. Stopping criterion & explain why you selected this criterion

   Out of the 4 stopping criteria mentioned in [1]/pp.34, we opted for the implementation of the 3rd one. The reasoning behind this decision is that this is the one with the most adaptivity: if we were to restrict the maximally allowed CPU time or keep limiting the total number of fitness evaluations, we would introduce unwanted constraints into a testing process where we are interested in the algorithm's capability of freely reaching the best possible optimum. Stopping under a certain population diversity would only be optimal with a really low threshold, as low diversity does not necessarily mean that the results will not improve further, as we willsee later. Our choice of stopping criterion, on the other hand, stops the execution only once the fitness has converged to a region where it will remain relatively stabilized for a given period of time (here experimentally determined to be *#Generations/5*, not to stop too early, but maximized at 100 generations), meaning that it is very unlikely to significantly improve anymore, so it returns an approximately optimal path. Our stopping criterion implementation can be found in **run_ga.m**, starting from line 60, and a textbox was added to the GUI where the fitness threshold value can be assigned (default 0.05, determined experimentally, suited to run with scaled fitness values).

2. Test results (incl. performance criteria and parameter settings)

   We tested the stopping criterion with multiple configurations. In this regard, the exact parameters are less important, the only requirement is that the process must converge to a stable region earlier than reaching the maximal number of generations, otherwise the stopping criterion would not trigger. Figure 2 shows a fitness curve plot after running this test.

3. Discussion of test results

   Our results clearly show a successful implementation of the stopping criterion: the algorithm stops after having converged to a relatively stable region, where only marginal changes would be applied to the resulting fitness value, which renders the remaining iterations pointless cost-wise. The optimal threshold value is subject to experimentation: for most of the example datasets, 0.05 was appropriate. Increasing it further would result in it triggering too early before stabilization, and if its value is too small, triggering is effectively prevented.

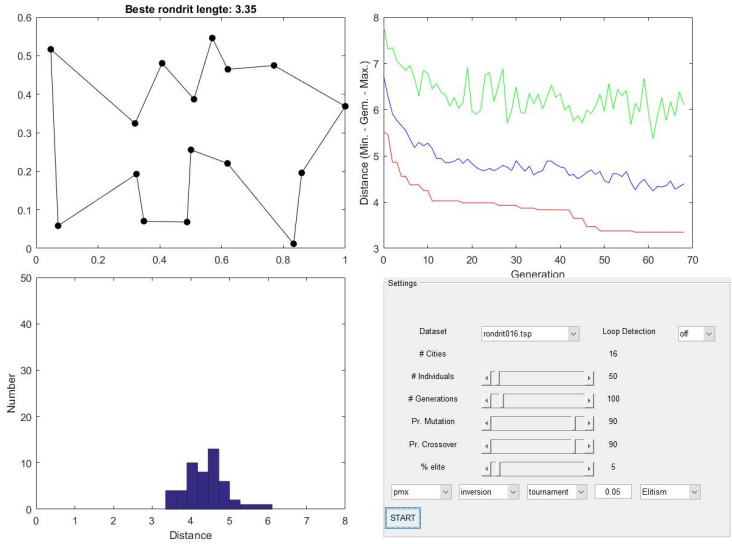

Figure 2: Fitness plot showing the triggered stopping criterion at generation #68. The parameter configuration used in the example can be seen on the configuration panel (bottom right)

3 Other representation and appropriate operators (main task)

Implement and use another representation and appropriate crossover and mutation operators. Perform some parameter tuning to identify proper combinations of the parameters.

1. Representations

   In the TSP, keeping the sequences of genes intact are also of high importance, even more so than positions. Therefore, the most suitable and widely used options for this problem are the permutation representations [1]/pp.67, because these eliminate the risk of producing illegal paths. In the provided implementation, the adjacency representation is used with appropriate operators. However (partially based on the course material and on oral discussions during the exercises), as the path representation is deemed to be more natural, and has the most available operator options, we decided that using it would be the most efficient in our own solution. The conversion functions for this can already be found in **adj2path.m** and **path2adj.m**.

   We were also interested in other optimization considerations: storing the huge amount of data that is often required by running such a genetic algorithm on a large dataset could be made more memory-efficient: for this purpose, and for the sake of the task, we decided on implementing another, binary representation [1]/pp.51. as well, despite it not being suitable for calculations for the problem at hand. The conversion functions can be found in **path2bin.m** and **bin2path.m**. Our algorithm dynamically determines, based on the size of the used dataset, that how many bits are required for storing a value. Then an individual path representation (vector of integers) is converted to a single bitstream consisting of words of this set size. For example, the path representation "2 1 7 3 5 4 6" with bitstring length 3 converts to the binary "001000110010100011101" stream. All data is stored in this format between operations, aiming for reducing memory usage, but operators and evaluation are still performed on path representation.

2. Crossover operators & explain why you selected the operators

   In literature, several contradictory statements can be found regarding the performance of certain crossover operators. Some state that PMX (partially mapped crossover, [1]/pp.70) is an efficient method for solving most problems, including the TSP [2],[3]/Crossover Operators, but others (course notes, especially for the TSP) mention that it is generally outperformed by the OX (order 1 crossover, [1]/pp.73, which is the fastest crossover operator for permutation representation, and acknowledges that the ordering of the cities is more important than their positions [4]. In some sources, it is stated that the Edge recombination crossover [1]/pp.72, [3]/Edge recombination, although being much more heavy computationally, typically outperforms both of the above on

the TSP on a generation by generation basis, due to avoiding the introduction of stray edges, while others do not mention this kind of superiority.

We were interested to see how these statements hold up with our own algorithm. To this end, all three mentioned crossover operators were implemented. Code can be found in **partially_mapped_crossover.m**, **order1_xover.m**, and **edge_recombin.m** respectively.

3. Mutation operators & explain why you selected the operators

   The mutations [1]/pp.69. implemented by default either exert a controlled sequence change (inversion, which only cuts two edges to revert the substring in between), or have linkage-breaking effects (swap). We were interested in seeing more options: the scramble mutation is expected to have a much more randomizing effect on the chromosome, which can help to effectively leave a local optima by increasing variance. The insertion mutation, on the other hand, should have more limited effects, but instead of swapping two cities, it moves one city next to another, which might serve as a more optimal choice for fixing a suboptimal tour segment. The implementations of these can be found in **scramble.m** and **insertion.m** respectively.

4. Parameter tuning: parameters considered and intervals/values

   Based on our initial testing, we have tested the crossovers and mutations by the following factors (performance measured in terms of mean best fitness, speed and smoothness of convergence):

   (a) Genetic Algorithm performance with each crossover operator under different xover and mut probabilities (high-low, low-high, and high-high);
   (b) Effect of the chosen mutation operator on performance, when it has a primary role (case of xalt_edges with low xover and high mutation probability);
   (c) Behavior and performance of the crossover/mutation operator combinations.

   The exact parameter values used in the tuning tests are shown in Tables 2 and 3.

5. Data set(s) used & explain why you selected these data set(s)

   We performed tests on three datasets: first on sizes of 25 and 100, in order to see how they compare to the default implementation, and how the dataset size affects their performance. Once we determined the general behavior, and verified that they are consistent over both datasets, further, comparative testing was done on the 50 cities set to speed up the tests computationally. The number of individuals and generations were tuned and adjusted accordingly (50/100 for 25 cities, 100/200 for 50 cities, 200/350 for 100 cities), in line with our observations in Section 1.

6. Test results (incl. performance criteria and parameter settings)

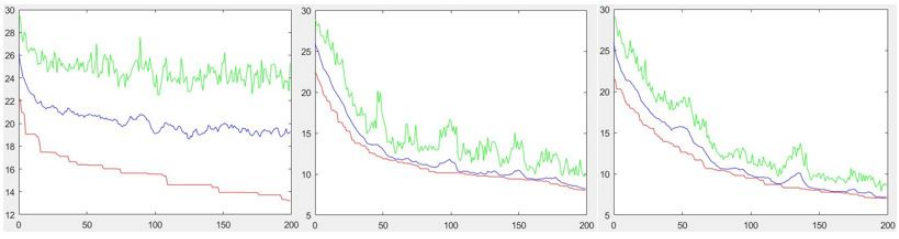

Figure 3: Comparison of the crossover operator convergence plots: xalt_edges (left), pmx (middle), ox (right), using high crossover and low mutation, on dataset of size 50. The plot of edge_recombin is very similar to xalt_edges, and is therefore omitted.

Test results are shown in Tables 2 and 3, with the best tour length highlighted in bold, and Figures 3 and 4, showing a few selected convergence patterns. Note that in [3], the implementation of OX is slightly different (one step detail) than in the course material: both were tested, but the former's performance was much more effective and efficient, so that is the one we kept for further analysis (the latter behaves similarly to xalt_edges, best results with high mutation

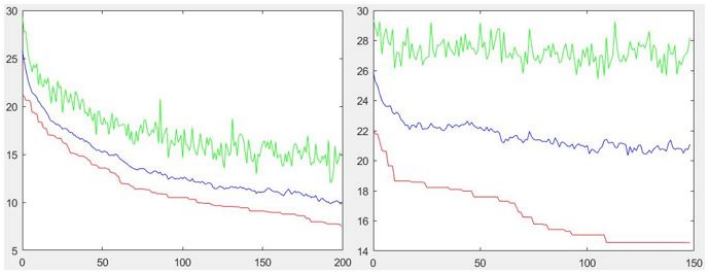

Figure 4: Comparison of the mutation operator convergence plots: inversion (left), scramble (right), using high crossover and low mutation, on dataset of size 50. The plots of insertion and interchange strongly resembles that of inversion, and are therefore not shown.

probability, as tested in line with Table 2 (not shown)). The behavior was also analyzed in terms of convergence nature (smoothness).

7. Discussion of test results

   Our results clearly show that our implementation of PMX performs significantly better than the provided xalt_edges, and it also shows more gradual improvement on bigger datasets, with much more prominent differences between the tested parameter configurations. PMX generally converges much faster to a much better result than xalt_edges, but it is generally slightly outperformed by OX both in terms of convergence speed and MBF (with OX, a larger initial drop is observable in the curve, as it is better at keeping the sequence intact in a subpath of random length, and transmits information about the relative sequence from the second parent). It was found that PMX and OX perform best and converge the smoothest with high crossover and low mutation probabilities, which is a clear improvement regarding crossover utility compared to the default xalt_edges, which was best with low crossover and high mutation (that is, when randomness takes up most of the responsibility for improvement). Edge recombination was also tested: although it generally outperforms xalt_edges using its optimal parameters, it takes significantly more time to compute, converges much slower with a step-like pattern with very abrupt drops, and usually to a local optima. It provides its best results if crossover probability is low, similarly to xalt_edges, but is significantly outperformed by both PMX and OX in their optimal parameter configurations. Generally, we would favor the selection of quickly converging operators: although the slow step-like patterns can also produce good results in time, this is computationally heavy, slow, and is therefore usually eliminated by our stopping criterion. To this end, PMX and OX are kept as best-performing crossover operators.

   Regarding mutations, inversion is the most efficient (with any crossover option), as this introduces the smallest possible change by flipping a subpath, preserving the most adjacency information. The random, undeterministic behavior of scramble is clearly visible in Figure 4. This mutation also produced a wide range of results, sometimes very good, sometimes very poor, proving it to be rather unreliable. Insertion converges smoothly, similarly to interchange, but provides better results, and it leads to even faster convergence than inversion, especially on smaller datasets (below 50 cities) due to its larger, link-breaking effects, but does not outperform inversion in terms of final results, and inversion also produces a smoother curve with less abrupt drops. The last three options all perform aggressive actions, being irrespective of the edges (sequence of cities), therefore, inversion was kept as the most efficient mutation operator.

   The chosen parent and survivor selection strategies also strongly influence the efficiency of our operators. In this stage, the default settings were used, but further discussion of the crossover and mutation operators is warranted together with testing our implementations for these selections. This will be detailed further in the next section.

Configuration					
# C	Crossover	Pr	Mutation	Pr	Pathlength
25	XALT	90	inversion	10	6.8435
		10		90	**4.8511**
		90		90	6.6317
	PMX	90		10	**4.5546**
		10		90	4.8579
		90		90	5.8377
	OX	90		10	**4.3964**
		10		90	4.7084
		90		90	5.5392
	E_REC	90		10	7.1129
		10		90	**4.7597**
		90		90	7.7198
100	XALT	90	inversion	10	31.5466
		10		90	**18.5044**
		90		90	29.9696
	PMX	90		10	**15.2089**
		10		90	17.1758
		90		90	25.2305
	OX	90		10	**14.6179**
		10		90	17.0628
		90		90	23.7826

Table 2: Crossover operator testing parameters and results. All possible combinations were tested with these parameter configuration options, and the results stayed characteristically the same over differently sized datasets.

Configuration					
# C	Crossover	Pr	Mutation	Pr	Pathlength
25	XALT	10	inversion	90	**4.7213**
			interchange		5.7803
			scramble		6.2357
			insertion		5.2148
100	XALT	10	inversion	90	**16.2907**
			interchange		21.1885
			scramble		29.5902
			insertion		19.8219
50	PMX	90	inversion	10	**7.897**
			interchange		9.962
			scramble		11.558
			insertion		9.3211
		90	inversion	90	**10.0837**
			interchange		12.3296
			scramble		14.0708
			insertion		11.7384
	OX	90	inversion	10	**7.5806**
			interchange		8.3285
			scramble		11.0521
			insertion		9.0607
		90	inversion	90	**9.7509**
			interchange		12.0282
			scramble		15.5161
			insertion		11.5401

Table 3: Mutation operator testing parameters and results. Top area contains separated mutation testing, while bottom area shows the most efficient crossover-mutation combinations for comparison. Tests were performed for all possible combinations of probabilities for the comparisons, but the observations made on the individual crossover/mutation experiments were unaltered, and so these results were omitted from this table.

4 Other tasks

You should select *at least one* task from the list.

We have decided to solve *two* tasks, which we were most interested in from the given options. Based on our initial tests, we estimated that these could significantly improve our algorithm's performance.

Parent selection methods:

1. Description of implementation

   The default parent-selection method used by the provided algorithm is sus (stochastic universal sampling), performed on a ranked set of chromosomes. We have implemented three other methods for comparison: rank-based (simple, linear), roulette wheel (also a fitness proportionate selection) and tournament selection (no replacement for increased determinism, with k experimentally tuned to *#individuals/10* for adaptivity) [1]/pp.80-86, [5]. These can be found in **rankbased.m**, **roulette.m** and **tournament.m** respectively.

2. Description of the experiments

   The purpose of our experiments was to compare the implemented parent selection methods, and examine their induced change in performance when paired with our operators. For this purpose, each selection method was tested with xalt_edges and PMX as well, on dataset sizes 16 and 50. Inversion was used, due to being the most efficient mutation operator. The number of individuals was chosen in accordance with the observations made in the first and second sections. The tuned operator probabilities are presented with the test results.

3. Test results

   Table 4 shows the averaged test results for this phase. For each selection method, the most optimal probability configuration is shown. For sus, roulette, rank-based: low mutation (10%) and high crossover (90%). For tournament: high mutation and high crossover (both at 90%).

Configuration			
#Cities	Crossover Function	Selection Strat	Tour Distance
16	XALT	Sus	3.6027
		Tournament	**3.4834**
		Roulette	4.1368
		Rank-based	3.8207
50	XALT	Sus	13.3603
		Tournament	**6.7184**
		Roulette	12.4762
		Rank-based	11.3895
16	PMX	Sus	3.9549
		Tournament	**3.2862**
		Roulette	3.9714
		Rank-based	4.0032
50	PMX	Sus	8.475
		Tournament	**6.1455**
		Roulette	8.659
		Rank-based	7.8725

Table 4: Test results for the implemented parent selection strategies

4. Discussion of test results

   Roulette wheel selects the parent based on their fitness proportion of the entire population. This is a significant difference from simple rank-based selection, where the individuals are ranked based on their fitness, but onto a linear scale. Consequentially, roulette has a higher probability for selecting the fittest as parents, but the selection probability curve of rank-based is much smoother. This reflects in their convergence: both produce a step-like pattern (similar to sus), with larger drops between approximately horizontal segments, and reach similar optimas, but the drops in roulette are much more abrupt. Simple rank-based performs generally better than fitness-proportionate selections (roulette and sus), as expected, since it preserves a constant

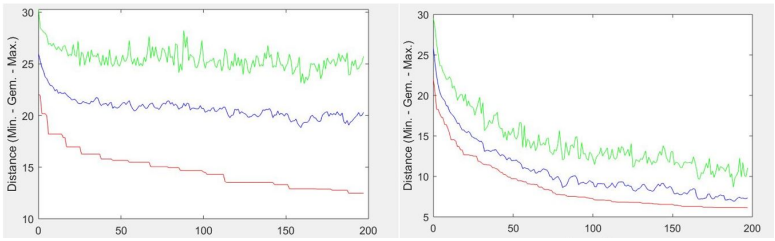

Figure 5: Convergence curves for the implemented **SUS** (step-like, left) & **Tournament** (smoother, right) based parent selection methods, using PMX

selection pressure by its sorting method, unlike FP selections, which have low pressure (meaning that selection is uniformly random) if the difference between fitness values is very small, nearing the (local) optimum.

Tournament selection, on the other hand, compares the relative fitness of a few individuals, and selects the best ones: the only random/probabilistic factor is in selecting the pool for comparison, but the k-1 least fit individuals can never be selected. This increases efficiency (less susceptible to premature convergence to a local optima), and produces a much smoother curve. This comparison can be observed more concretely in Figure 5. From this, and the results in Table 4, one can interpret that for small datasets, there is no significant difference between the results when using the different methods, but on larger datasets, tournament converges much faster to a much better optimum. Another addition is that while PMX and OX generally worked well with low mutation and high crossover probabilities, using our tournament selection provides the best results with these crossovers if both probabilities are high, as it highly relies on mutation to increase competition and exploits this diversity-increasing stochasticity in its aggressive (no replacement or winner-selection probability) selection process (it even increases the performance difference between inversion and the other mutations). Low mutation with tournament selection thus results in early convergence and loss of diversity. It is possible that this could be further improved by using a tournament implementation with probabilistic conditions for replacement or winner selection, and controlling the value of k more (it was tuned for safety now, as too low k would result in a more random search, and too high would quickly reduce diversity). An important note is that all of these observations hold deterministically for our crossover implementations: PMX and OX still outperform xalt_edges with all selection strategies, but tournament selection significantly improves the results with all operators.

Survivor selection strategies:

1. Description of implementation

   In the Section 1 and 2, we have highlighted the importance of survivor selection regarding the performance of the GA, based on the tests with the default elitism implementation. For this reason, we were interested in how other survivor selection methods would perform, as the available options significantly differ in their approach. Age-based selection was considered, but as all individuals start from age 0 in our example, the only way we could have introduced this properly was to assign a random age to each individual during algorithm initialization, which would introduce unwanted stochasticity. Moreover, age-based selection d Moreover, age-based selection does not depend on the fitness values of the individuals, so this survivor selection strategy is executed on a different metric, which is in hindsight not directly related to the objective function that is to be optimized. (μ, λ) was not chosen, as $\mu > \lambda$ does not hold for our case. Thus, the uniform, $\mu + \lambda$ and round-robin ($q = 10$, as suggested by [1]) strategies were chosen to be implemented [1]/pp.89, as these represent two different approaches to selection. These can be found in **reins.m**, starting from line 100.

2. Description of the experiments

   For this section, we have conducted experiments over three independent variables: the dataset size (16 and 50), our best crossover operators (PMX and OX), and the 4 implemented survivor

selection strategies, for all possible combinations. For the GA specifications, we used: 50/150 individuals , 150/250 generations , 90% mutation and crossover probability, **tournament** selection and 5% Generation-gap. Loop detection was kept turned OFF.

3. Test results

   The results of these experiments are displayed in this section. Distances were formulated as the MBF averages of 5 trials.

Configuration				
#Cities	Crossover Function.	Selection Strat	Tour Distance	Convergence Time
16	PMX	Elitism	**3.4353**	55 Generations
		Uniform	3.9006	NaN
		$\mu + \lambda$	3.4815	35 Generations
		Round-Robin	3.464	40 Generations
50	PMX	Elitism	6.2587	130 Generations
		Uniform	6.9654	NaN
		$\mu + \lambda$	**6.0834**	120 Generations
		Round-Robin	6.2834	160 Generations
16	OX	Elitism	3.4756	60 Generations
		Uniform	3.7714	NaN
		$\mu + \lambda$	3.4813	40 Generations
		Round-Robin	**3.4541**	40 Generations
50	OX	Elitism	6.3251	130 Generations
		Uniform	6.3586	NaN
		$\mu + \lambda$	6.2875	120 Generations
		Round-Robin	**6.1143**	150 Generations

Table 5: Test results for given survivor selection strategy

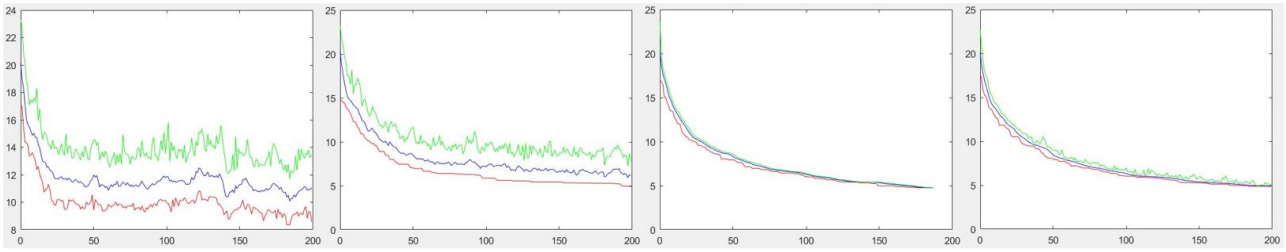

Figure 6: Convergence curve comparison for the implemented survivor selection strategies, from left to right: **Uniform** & **Elitism** & $\mu + \lambda$ & **Round-robin**.

4. Discussion of test results

   As expected, uniform selection produces a very randomized pattern with only initially observable convergence, as there is no fitness-based argument behind random survivor selection. The general purpose of survivor selection, however, would be to enhance the new genetic pool with the fittest individuals. Elitism keeps only a few of the fittest individuals, allowing others to be replaced, thus maintaining a higher population diversity (which can be observed in the large ripples in the worst individuals' curve in the generation). $\mu + \lambda$ and round-robin, however, assemble a new population from the fittest members of the shared parent and offspring pools. This will quickly lead to the lowering of diversity, as shown in Figure 6, but the crossovers between the increasingly fit individuals of the population will result in a gradual decrease in total tour distance.

   Due to these properties, elitism is slower to converge than the more deterministic $\mu + \lambda$ and round-robin, but if let to run long enough, it eventually reaches similar results (but it needs to be terminated by the stopping criterion, as it never stops automatically due to hitting a diversity bottom, as the other methods). Generally, $\mu + \lambda$ reaches a bit better results than round-robin, as the latter still introduces a bit of probabilistic selection (which, in turn, maintains more diversity), as shown in Figure 6. In contrast with our observations in Section 1, for $\mu + \lambda$ and round-robin, the smoothness of the curve and the speed of convergence does not depend so heavily on mutation probability, because these strategies always keep most of the best individuals, reducing the impact of destructive mutation. However, the stopping criterion for population

diversity below a certain threshold tends to kill the process earlier due to the reduced diversity and fast convergence, so mutation still helps to avoid stopping at a local optimum. Round robin can also sustain this diversity at a bit higher level despite low mutation probabilities, when compared to $\mu + \lambda$. Having a high crossover probability helps these deterministic methods to perform better. Elitism can endure low crossover and low mutation probabilities better than $\mu + \lambda$ or round-robin, due to the heavier randomness it introduces.

Regarding crossover selection, it was observed that elitism works better with PMX than $\mu + \lambda$ or rr in smaller datasets. However, in bigger datasets, the advantage of $\mu + \lambda$ can be observed. On the other hand, OX produces slightly better results with $\mu + \lambda$ or rr than with elitism. The best configurations were deemed to be elitism + PMX for smaller tours and $\mu + \lambda$ + PMX/OX for bigger tours, but this setting is again dependent on the *#cities*, as seen in Table 5.

5 Local optimisation

Test to which extent a local optimisation heuristic can improve the result.

1. Local optimisation heuristic

   Loop detection was used as a local optimization heuristic, which removes local loops from a TSP path, with the maximal treated subpath length specified as a parameter. We were interested in seeing how this specified pathlength influences the effectiveness of the algorithm: therefore, it was tested at multiple values, set within the algorithm in **improve_path.m**.

2. Test results (incl. performance criteria and parameter settings)

   The heuristic was tested with subpath lengths of 2 and 3. As the purpose of a local optimization heuristic is to further improve the results, we tested this primarily with our most efficient parameter setup (PMX/OX, inversion, 90% crossover and mutation probabilities, 5% $\mu + \lambda$/elitism, tournament, on datasets #16 and #50 with 50/100 individuals and 100/300 generations, 0.05 stopping criterion threshold), and compared it to its effects on the default configuration:

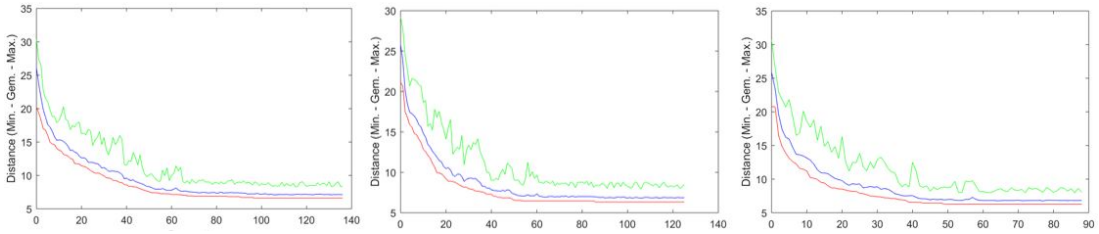

Figure 7: Convergence curves for: Loop detection off (left); Loop detection for subpath length of 2 (middle); Loop detection for subpath length 3 (right)

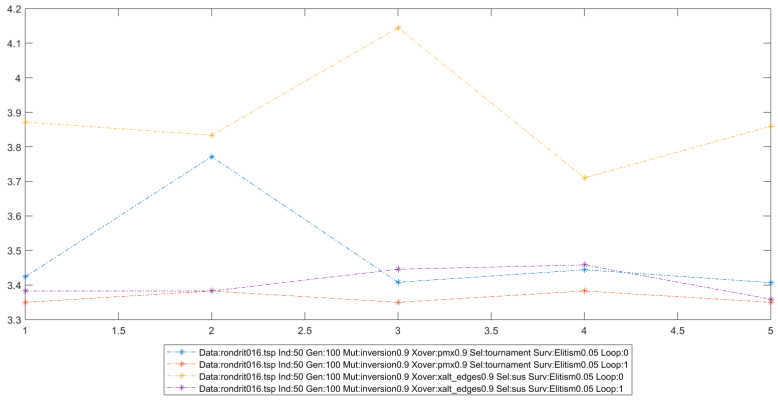

Figure 8: Results of tests with loop detection heuristics tested on rondrit16 dataset

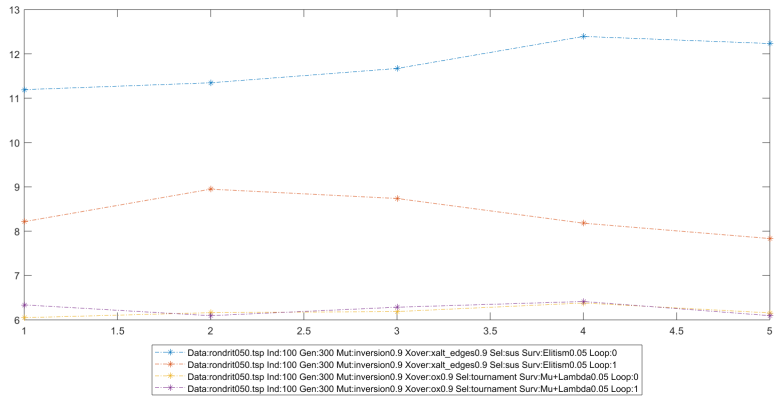

Figure 9: Results of tests with loop detection heuristics tested on rondrit50 dataset

3. Discussion of test results

   We have observed that when using loop detection on a generally well-performing algorithm (tested with our best configurations), the final result (MBF) does not change much (only slight improvement is observable), as shown in the plots in Figure 8 and 9.

   The speed of convergence, however, is massively improved: it reaches the optimum significantly faster (under up to 3-4 times less generations, thus exploiting our stopping criterion), by a very abrupt initial fitness increase, due to the immediate removal of sub-optimal, looping subpaths, which otherwise would need to be eliminated by crossovers and mutations as time progresses.

   From the graphs of Figure 7, we observe that turning the Loop detection heuristic on generally allows for faster convergence. Moreover, comparing a subpath length of 2 vs. 3, we observe that in the latter version, the GA converges around #90 generations, whereas in the former, it takes around #120 generations. Hence, we can also conclude that the higher subpath length check leads to more efficient detection, and faster convergence. However, the source code limits the loop detection in subpaths **upto** length 3. Therefore, we were not able to test further, and cannot comment on whether the correlation between subpath length and convergence still holds for higher values. Similarly, even for the slowly-converging default xalt_edges and sus combination (which is usually stopped by the stopping criterion, as no fitness change is observable between the rare pathlength drops), and less optimized crossover and survivor selection strategies, using loop detection significantly improves both the convergence speed, and the resulting total pathlength. The improvements over the path distance can again be observed in Figure 8 and 9.

6 Benchmark problems

Test the performance of your algorithm using *some* benchmark problems (available on Toledo) and critically evaluate the achieved performance.

1. List of benchmark problems

   The algorithm's best performing configurations (PMX/OX, inversion, 90% crossover and mutation probabilities, tournament selection, 5% elitism/$\mu + \lambda$, with loop detection ON) were tested on *belgiumtours.tsp*, *xqf131.tsp* & *xqf380.tsp* benchmark problems shared on Toledo. A comment section was added to **tspgui.m**, lines 123-126, to enable switching pathlength scaling on/off. Results were compared with the known best routes.

2. Test results (incl. performance criteria and parameter settings)

   Results are also dependent on the number of individuals, generations, and the stop condition threshold. Generally, as the benchmark datasets are rather large, and we were interested in the best results we can reach, we do not want to restrict the algorithm by these parameters. The stopping condition threshold was therefore set to very low (0.001), and #Generations was

maximized. #Individuals was fine-tuned for the best performance for each problem. Results of the experiments can be observed in Table 6 for the corresponding configurations.

Benchmark	Optimal Solution	Test Results	# Individuals	#Generations
belgiumtours	N/A	664.0276 ± 13.86	150	300
xqf131	564	640.8642 ± 23.14	350	350
bcl380	1621	2165.4618 ± 126.4	800	550

Table 6: Test results of the best GA configuration over benchmark problems

Figure 10: Optimal tour for *xqf131* (Left) vs. the path constructed in the best configuration of our GA (Right)

3. Discussion of test results

   Naturally, the algorithm performed better on the smaller benchmark datasets: computation time is raised exponentially for larger sets, and they require more generations to properly converge. On the first problem (*xqf131.tsp*), our algorithm was capable of reaching the vicinity of the known global optimum, reaching up to below 10% deviance from it. We estimate that this could be further improved by introducing a small probabilistic condition for replacement into our most efficient tournament selection. Another interesting observation was that while both elitism, rr and $\mu + \lambda$ performed well and within these boundaries, the "shape" of the final, best path was usually more comparable to the known optimum if we used $\mu + \lambda$. This phenomenon was clearly observed using the *xqf131.tsp* dataset: the comparison of the known optimal path solution and our best result is shown in Figure 10. For the *bcl380* dataset, similar results were observed too, so they are not further discussed. Moreover, we have also included our best result for the *belgiumtours* dataset; we leave it to the reader to perform the comparison for that example.

Time spent on the project

1. The amount of time spent with working on the project, per team member:
   (a) Benedek: approx. 45 hours
   (b) Koralp: approx. 45 hours
2. Work distribution among the team members: We have met weekly to implement each task together, discussing our options and solutions. Each of us prepared for the meetings individually by looking up the appropriate topics in the textbook and other available literature (used sources referenced). The correctness of every implemented algorithm was verified by performing manual calculations (pen+paper) on test input data. After validating the implementations, we distributed the required testing tasks between each other, performed them, and plotted/logged the results individually, with occasional and scheduled online discussions. We have merged them into the report, and formulated the descriptions, figures and result discussions together.

References

[1] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[2] Kulbhushan Verma and Manpreet Kaur. Palvee,"comparative analysis of various types of genetic algorithms to resolve tsp". *International Journal of Electronics and Communication Engineering & Technology (IJECET)*, 4(5):111–116, 2013.

[3] Rubicite Interactive. Genetic Algorithms Tutorial. [Online] Available at: `http://www.rubicite.com/Tutorials/GeneticAlgorithms.aspx`, last accessed 20-december-2019.

[4] Keivan Borna and Vahid Haji Hashemi. An improved genetic algorithm with a local optimization strategy and an extra mutation level for solving traveling salesman problem. *arXiv preprint arXiv:1409.3078*, 2014.

[5] Tutorialspoint. Genetic Algorithms - Parent Selection. [Online] Available at: `https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm`, last accessed 20-december-2019.