Genetic Algorithms and Evolutionary Computing Project

Jesse Smits & David Schotmans

January 3, 2019

Abstract

In this paper, the impact of different parameter sets in TSP problems are discussed using the 'GA Toolbox from University of Sheffield' together with a 'Template matlab program'. The performance of different TSP representations are compared with one another. Heuristics are a good tool for direct improvements, however they should be used in combination with other techniques to prevent ending up in a local optimum of the search space.

1 Introduction

In this paper, we will discuss our adaptations and conclusions for the genetic algorithm of the Travelling Salesman problem (TSP).

Varying the parameters of the genetic algorithm and combining them in various ways allowed us to obtain a stable set of good parameters. These observations are discussed in section 3.

The existing genetic algorithm was lacking a stronger stopping condition to avoid rather useless iterations. The decisions in designing this more complex stopping criteria will be discussed in section 4.

In addition to the already present adjacency representation, a path representation was implemented in combination with edge crossover and inversion mutation. The implementation details will be discussed in section 5. The matlab code can be viewed in the accompanying archive file.

Section 6 discusses the impact of a local optimization heuristic on the performance of the genetic algorithm.

Finally, two new parent selection methods were introduced in order to compare the performance of different selection mechanisms on the TSP.

2 Overall approach

Starting from the given implementation of the genetic algorithm, we adapted the code so that it allows for automated batch processing. This way, we can setup experiments with varying parameters for the same genetic algorithm and extract those sets of parameters that produce promising results.

2.1 Assumptions

To compare the results obtained from exploring parameter sets, we make the following assumptions:

- For every parameter set, the number of runs the genetic algorithm is executed equals `R`
- All runs have the same number of generations `G`

These assumptions are important for comparing the results, as one is interested in how quickly (i.e. convergence time) a certain parameter configuration can produce optimal results.

The parameters `R` and `G` are determined by making a trade-off between resources (e.g. time, hardware capabilities, ...) and accuracy of the results. A higher number of runs will produce more data, which leads to a higher probability that the observed optimal parameter set corresponds with the true optimal parameter set. Increasing the number of generations for a run sheds more light on the time to converge to an optimal result and which parameter configuration comes closest to the optimal result.

When testing the benchmark problem this way, we set `R` to `10` and `G` to `100`. This is on the one hand motivated by only having a limited amount of hardware resources. On the other hand we will test benchmark problems in two ways, so that we can still determine an approximately optimal parameter configuration with a high probability.

The benchmark problems are evaluated in the following two ways:

- Batch processing parameter sets for the benchmark problem using the above mentioned parameters `R` and `G`.
- Single long run without a limit on the number of generations (except the stopping condition)

In the second case, we can evaluate how good the result is produced by a specific parameter configuration.

2.2 Performance evaluation

To improve the performance of the genetic algorithm for TSP, we first must define how performance will be measured. We can measure performance by several dimensions:

- The best solution found by the genetic algorithm, i.e. the minimal tour
- The execution time of the genetic algorithm, i.e. time needed to reach the stopping condition

An approximately optimal tour that requires a relatively long execution time is not very useful in production environments where having a reasonable tour as quick as possible is often more important.

2.3 Visualization

Having the results stored in a data folder scattered in multiple *.csv* files we have designed a visualization tool in python that extracts the essential data from each file belonging to the same test scenario. Now, since we have all the necessary data, we can visualize individual runs similarly as the tspgui dynamic plot and create a boxplot for each scenario. To create the boxplot the tool takes the best result from each test run and combines them. These plots will be used to see whether a change in the parameter setting space has a significant impact or not.

3 Comparing parameter sets

In this section, we briefly discuss the impact of several parameters on the performance of the genetic algorithm. If appropriate, claims will be supported by corresponding plots of the observed data.

3.1 Crossover rate and mutation rate

Experiments with different crossover and mutation rates have shown that a low crossover rate in combination with a higher mutation can lead to better results. However, this observation only holds in the absence of problem specific knowledge (i.e. a heuristic). Without the use of a heuristic, the genetic algorithm must first explore more areas of the search space to find good candidate solutions. This corresponds to a relatively high mutation rate. Only when good candidate solutions are reached, more exploitation is needed.

Figure 1 shows the minimal tour lengths of 10 runs of the genetic algorithm with a high crossover rate and a low mutation rate. Figure 2 shows the minimal tour lengths of 10 runs of the genetic algorithm with a low crossover rate and a high mutation rate. As can be seen from figure 2, a high mutation rate leads to smaller minimal tour lengths in the same amount of time (i.e. same number of generations).

The results in the presence of a local optimisation heuristic are discussed in section 6.

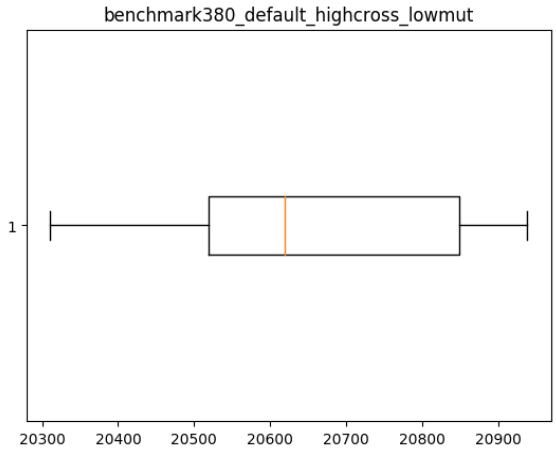

Figure 1: Boxplot showing the minimal tour lengths for 10 runs of the `benchmark131` benchmark problem, with `crossover rate = 0.95` and `mutation rate = 0.35`

.

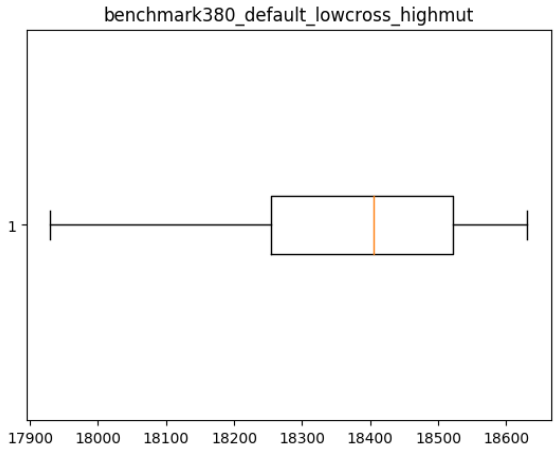

Figure 2: Boxplot showing the minimal tour lengths for 10 runs of the `benchmark131` benchmark problem, with `crossover rate = 0.05` and `mutation rate = 0.55`

.

3.2 Elitism

When varying the rate of elitism, we concluded that a low rate works best as this will prevent premature convergence of the population in terms of fitness values. However, it makes sure that various good solutions remain in the population. This helps the genetic algorithm to find an area in the search space where the population consists on average of good candidate solutions.

4 Stopping Criteria

The already existing termination condition was limited to the population diversity of a generation and the maximum number of life cycles specified by the user. To improve this condition another criterion was added to reduce the computational load on the system.
Evaluating the fitness improvement over a certain period of time: This condition will observe the evolution of the best results generated by the algorithm. If the current best solution does not improve by a certain threshold ε in a specific period of time the execution will terminate. Having this extra condition is very helpful when doing multiple test runs on our local machines because it makes the termination condition stronger which was a requirement when testing on benchmark problems that could take up to hours having a rather weak stopping condition. Depending on the strength of the hardware components we can adapt these parameters to have a stopping condition on an agreeable time limit.

5 Path representation

The path representation of the TSP was used as an alternative representation. In this representation, a tour is represented as a permutation. For the crossover operator, the edge crossover operator was used. The existing mutation operator (inversion mutation) was reused in this representation.

To integrate this new representation with the existing genetic algorithm, the provided functions `adj2path` and `path2adj` are used to convert between the adjacency representations and path representations.

5.1 Implementation

The edge crossover operator is implemented in Matlab. To create a new child chromosome, we use two parents to construct an edge table, which shows the connections for every gene to its neighbours in both parents. From this edge table, a new chromosome is constructed according to the algorithm presented in figure 3. We create two child chromosomes by changing the order of the parents in the call this algorithm.

1. Construct the edge table
2. Pick an initial element at random and put it in the offspring
3. Set the variable *current_element* = *entry*
4. Remove all references to *current_element* from the table
5. Examine list for *current_element*
   - If there is a common edge, pick that to be the next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - Ties are split at random
6. In the case of reaching an empty list, the other end of the offspring is examined for extension; otherwise a new element is chosen at random

Figure 3: Pseudo-code adapted from [1] showing the creation of the edge table.

.

5.2 Performance

The alternative path representation in combination with edge crossover produces relatively better results than the default adjacency representation. The path representation leads to smaller minimal tour lengths in a smaller number of generations. These observations are shown in figure 4. When compared with the boxplot shown in figure 2, one can see that our path representation produces on average smaller minimal tour lengths.

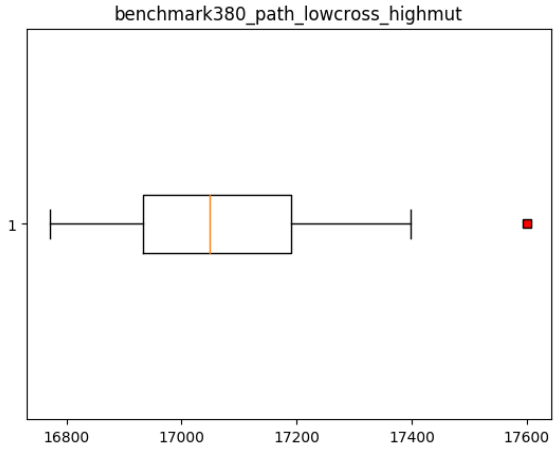

Figure 4: Boxplot showing the minimal tour lengths for 10 runs of the `benchmark131` benchmark problem, with `crossover rate = 0.05` and `mutation rate = 0.55` and path representation.

.

6 Local heuristic optimisation

Exploring the search space in an efficient manner becomes more important when considering relatively large instances of the TSP. A heuristic approach can help to improve performance by pruning parts of the search space. Tours that contain loops are not very useful and should be removed or improved. The given `localloop` heuristic will adapt tours with loops in them by removing these loops.

6.1 Exploration and exploitation

Experiments performed with the `localloop` heuristic showed that having a higher crossover rate in combination with a lower mutation rate leads to better results. This effect can be explained by observing that the heuristic prevents "bad tours" (i.e. tours with loops in them) are part of the produced offspring. This leads in turn to more rapid convergence of the population in terms of fitness values, as the offspring is only limited to tours without local loops. By using this heuristic, the genetic algorithm effectively needs less exploration and more exploitation. This way, the genetic algorithm uses problem-specific knowledge to reduce the time to reach a population with on average more good candidate

solutions. Figure 5 and figure 6 show the evolution of the fitness values when running the genetic algorithm on the `benchmark131` benchmark problem with high and low crossover respectively.

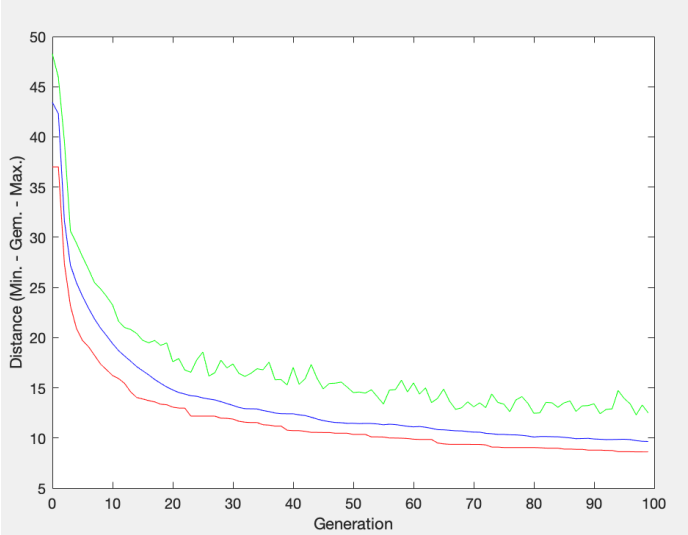

Figure 5: 100 generations of the genetic algorithm for the `benchmark131` benchmark problem with `localloop` heuristic and `crossover rate = 0.05` and `mutation rate = 0.8`

.

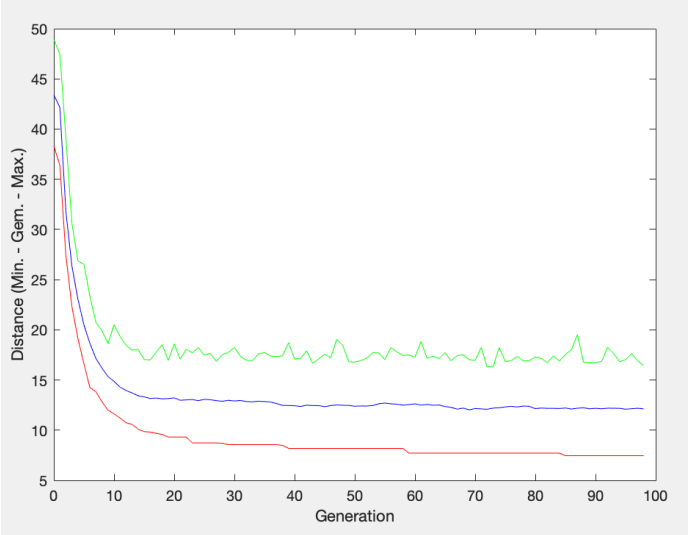

Figure 6: 100 generations of the genetic algorithm for the `benchmark131` benchmark problem with `localloop` heuristic and `crossover rate = 0.95` and `mutation rate = 0.35`

.

These observations also explain why the genetic algorithm needs a higher mutation rate and a relatively low crossover rate, when running it without this heuristic. As no problem-specific information is included, the algorithm must first explore large parts of the search space to reach an environment with relatively good solutions. Only when such an area of the search is reached, more exploitation is needed.

6.2 Local optima problem

When using a heuristic such as the `localloop` heuristic, the mutation rate must not be too low. Especially when a TSP problem with lots of local optima is considered, the genetic algorithm must still have the ability to explore the search space for possibly better solutions. As this heuristic is rather greedy and the initial population is determined randomly, one needs to take care that the result is not sub-optimal (i.e. reaching a local optimum).

7 Parent Selection Algorithm

Originally the only Parent Selection algorithm present in the GA is the Rank-Based selection. We chose to add:

- Tournament Selection
- Fitness Proportional Selection

7.1 Tournament Selection

Following the pseudo code in figure 11, we chose to have a binary tournament selection within the current generation. This means, randomly selection 2 parent candidates and selecting 1 winner based on the highest fitness score. After selecting 2 winners 1 new chromosome can be created.
The reason for having a binary tournament selection is the avoid premature convergence of a population. It is clear that having a rather large value for k a relative good chromosome will be selected multiple times with a high probability.

```
BEGIN
  /* Assume we wish to select λ members of a pool of μ individuals */
  set current_member = 1;
  WHILE ( current_member ≤ λ ) DO
    Pick k individuals randomly, with or without replacement;
    Compare these k individuals and select the best of them;
    Denote this individual as i;
    set mating_pool[current_member] = i;
    set current_member = current_member + 1;
  OD
END
```

Figure 7: Pseudo Code Tournament Selection (Pseudo-code adapted from [1])

.

7.2 Fitness Proportional Selection

Mimicking the Pseudo code in figure 8, we have to define a cumulative probability distribution for the current generation. Now, we have a vector containing relative fitness values smaller than 1 with a total equal to 1 (relative to the summation of all fitness values in that generation). After, we generate a random number from [0,1]. Having this random number we can determine in which interval of a chromosome this number belongs. That chromosome will be selected as a parent.
This sampling technique has the quality that it converges fast. Therefore we can conclude this selection mechanism will give good results in the local search space. especially in our test scenarios, having a rather low number of generations this algorithm helps generating an optimal result in a short time.

```
BEGIN
  /*  Given the cumulative probability distribution a */
  /*  and assuming we wish to select λ members of the mating pool */
  set current_member = 1;
  WHILE ( current_member ≤ λ ) DO
    Pick a random value r uniformly from [0, 1];
    set i = 1;
    WHILE (  a_i < r ) DO
      set i = i + 1;
    OD
    set mating_pool[current_member] = parents[i];
    set current_member = current_member + 1;
  OD
END
```

Figure 8: Pseudo Code Fitness Proportional Selection (Pseudo-code adapted from [1])

.

8 Benchmark problems

The genetic algorithm has been run on several benchmark problems. The parameter configuration was optimal according to our experiments discussed in section 3. This configuration is shown below:

- Crossover operator = edge crossover
- Crossover rate = 0.95
- Mutation rate = 0.35
- Elitism = 0.05
- Local Heuristic = enabled
- Parent selection = Fitness Proportional Selection

We chose to run the benchmark problems with a maximum of 2000 generations consisting each of 1000 individuals. The stopping criteria remained the same which was the original plus the evaluation of the fitness improvement over time.

8.1 Benchmark with 131 cities

The genetic algorithm performs very good on this benchmark, as the optimal tour has a length of 564, while the solution found by the genetic algorithm using our optimal settings finds a tour of length 581. The algorithm runs for about 370 iterations and as can be seen from figure 9, the heuristic provides an enormous improvement over only small number of iterations.

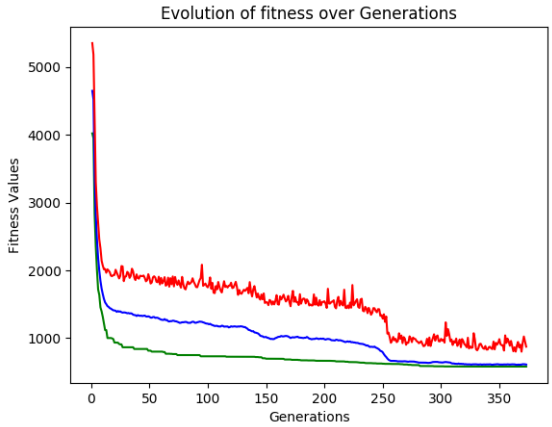

Figure 9: Run of the genetic algorithm with optimal settings on the `benchmark131` benchmark problem

.

8.2 Benchmark with 380 cities

When running the genetic algorithm on the benchmark problem with 380 cities, the solution generated by the genetic algorithm is not as close to the optimal solution as we expected. However, the number of iterations needed to come to the solution of the genetic algorithm is surprisingly fast: only about 70 iterations are needed to reach a tour of length of about 5000. The optimal tour has a length of 1621 which is still a lot less then the solution found by the algorithm. This difference can be explained by observing the following characteristics about the search space:

- The search space of 131 cities is quite small compared to the search space of 380 cities.
- The larger the search space, the more likely it is that the heuristic will end up in a local optimum.

The heuristic is greedy and as such it will prefer large and direct improvements over smaller and indirect improvements. Figure 10 shows this rather rapid convergence in the first 20 iterations. If the algorithm combined iterations of

more exploration (i.e. higher mutation rate) followed by iterations of more exploitation (using the heuristic), we would expect an improved result as it will not focus too soon on a seemingly optimal area of the search space.

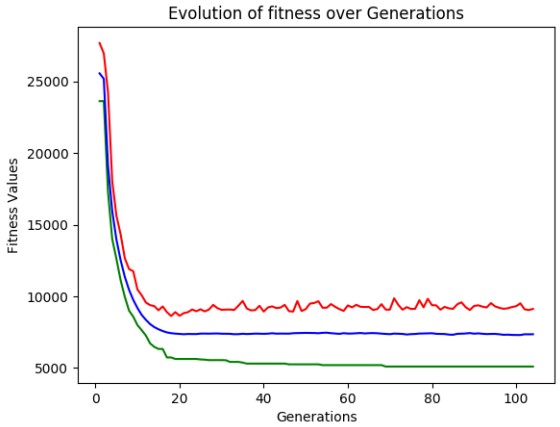

Figure 10: Run of the genetic algorithm with optimal settings on the `benchmark380` benchmark problem

.

8.3 Benchmark with 662 cities

The algorithm still performs well even for this benchmark tour problem. The search space for this problem is gigantic and yet a tour of length 6872 was found, with the optimal solution being 2513. The Algorithm made a total of 1895 generations before meeting the termination condition. The time consumption needed to generate a estimate to the 662 city tour was much higher than the previous benchmarks. The time needed to create this solution was close to 10 hours. Considering the immense search space (662 factorial), the high crossover cost (tables of 662 rows has to be created for each individual crossover) and the high number of generations and individuals, the generated solution is definitely acceptable.

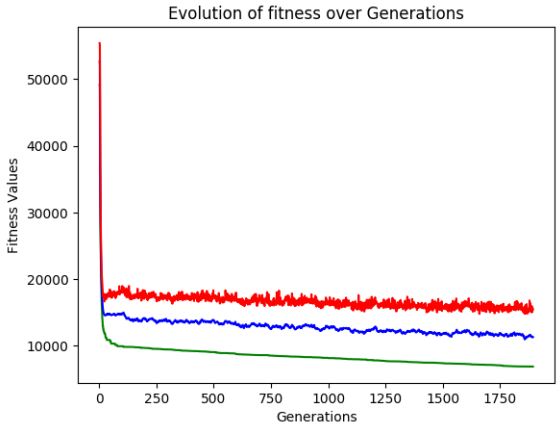

Figure 11: Run of the genetic algorithm with optimal settings on the `benchmark662` benchmark problem

.

9 Conclusion

After performing a lot of scenarios with different tour sizes, we came to the conclusion that it is challenging to find an optimal set of parameters on many different problems. We experienced that a particular set worked well for a specific TSP problem but had rather weak results on others. In addition, our path representation and edge crossover algorithm slightly increased the performance, which was in part due to a higher transferal of aspects of good solutions over generations than with the adjacency representation. Heuristics definitely helped to improve the best result by pruning illegal or invalid intermediate results. The Heuristic made fast convergences to local optima which has the disadvantage to ignore other parts of the search space. A trade-off between exploration and exploitation has to be made (preferably even dynamical, i.e. during the run) so that the heuristic can be used to make large improvements where possible, while still having the possibility to discover even possibly better parts of the search space through mutation.

References

[1] Introduction to evolutionary computing, Eiben, Agoston E and Smith, James E and others, 2003, Springer