Evolutionary Algorithms: Group report

Fatjon Barci, Brent Bottin, and Jonathan Van Den Berckt

November 9, 2023

1 An basic evolutionary algorithm

1.1 Representation

In solving the Travelling Salesman Problem (TSP) using genetic algorithms, the representation of potential solutions—the routes—is a foundational aspect that directly influences the effectiveness and efficiency of the algorithm. The chosen representation for this implementation is a permutation-based representation, where each individual in the population is a unique permutation of cities. The primary motivation for employing this representation is its direct correspondence to the problem's nature: every city must be visited once and only once, akin to a Hamiltonian cycle in graph theory. This representation simplifies the evaluation of routes since the fitness function calculates the total distance by traversing the permutation sequentially, thus ensuring a straightforward and natural encoding of the problem.

Alternative representations may include an adjacency-based representation, where each gene represents a city and its value to the next city in the route, or an edge-based representation that encodes edges between cities instead of the cities themselves. These methods, while potentially beneficial in preserving certain properties of routes during crossover and mutation operations, often require additional complexity in the genetic operators to ensure that the offspring are valid TSP solutions.

From an implementation standpoint in Python, a permutation-based representation benefits from the language's strong support for list operations. Python lists can easily be shuffled, indexed, sliced, and mutated, which aligns well with the operations required by genetic algorithm operators like crossover and mutation. Moreover, leveraging Python's libraries such as random for generating permutations or selecting individuals for genetic operations can greatly streamline the development process. The fitness function, crossover, and mutation operations, which are central to the genetic algorithm, are all implemented with consideration of the permutation representation, ensuring the creation of valid and potentially optimal TSP routes through successive generations.

```
    individual = [0, 2, 1, 3, 4]
```

1.2 Initialization

Initializing the population in TSP involves generating random permutations of city visits, ensuring a diverse starting point for the genetic algorithm. This randomness avoids initial bias and promotes exploration of the solution space. Only routes that form valid cycles—visiting all cities once—are kept, focusing the algorithm's efforts on viable solutions from the outset. This approach establishes a broad and feasible base for subsequent evolutionary optimization.

```
    random_initialization(num_individuals, n, distanceMatrix)
```

1.3 Selection operators

In addressing the selection method for the Travelling Salesman Problem (TSP), a comparison between top-*k* selection and *k*-tournament selection was conducted. The adoption of *k*-tournament selection is attributed to its emulation of natural selection, where individuals partake in direct competition for survival. This mechanism is particularly suitable for TSP as it ensures a consistent focus on the most viable solutions, akin to the principle of 'survival of the fittest'.

The *k*-tournament selection operates by randomly drawing a subset of the population, equivalent to the tournament size, and subsequently selecting the optimal individual from this subset. It adeptly strikes a balance between exploration and exploitation; the tournament size dictates the selection pressure. A diminutive tournament size fosters diversity (exploration), providing an opportunity for weaker individuals to be chosen, whereas an augmented size biases the selection towards stronger individuals (exploitation), thereby accelerating convergence.

The parameters demanding consideration are:

1. ***k***: It determines the count of individuals to be selected for the succeeding generation. The selection of k is pivotal—it should be ample enough to retain genetic diversity yet constrained enough to guarantee that predominantly the fittest individuals are chosen.
2. **Tournament size**: An escalated tournament size augments the predilection towards electing the best individuals, which may precipitate swifter convergence but also raises the specter of premature convergence. Conversely, a reduced tournament size renders the selection process more stochastic, which can aid in upholding genetic diversity and averting local optima.

Parameter values for k-tournament selection were smartly chosen in relation to the problem size to balance between diversity and selection pressure. For a 50 x 50 matrix, a tournament size of 5 with k = 25 was found to be a good starting point. As the matrix size increases to 200 x 200, the tournament size and k are scaled to 20 and 100, respectively, and further to 40 and 500 for a 1000 x 1000 matrix. These choices are based on preliminary experimentation that indicated a balance between exploration and exploitation necessary for the varied TSP landscapes.

```
    k_tournament_selection(population, k, tournament_size)
```

1.4 Mutation operator

As a first mutation operator we implemented the swap mutation in which we selected two random indices and swap the cities. Later on we changed the mutation operator to an inversion mutation. Here we select a subvector of the permutation with a random length and reverse the order of the cities. We chose this mutation operator because it introduces more variation compared to the swap mutation and thus a chance to have some more local randomness. At the moment we kept the probability of mutation constant at 5%, when we increased the mutation probability it did not lead to significant differing results.

1.5 Recombination operator

For the recombination parameter we chose for partial mapped crossover and later used a variation of it, namely order crossover. The partial mapped crossover randomly choose a start and end point (index). The subsequences of both parents, based on the start and end point, are then swapped such that parent one has the subsequence of parent two and vice versa. The order crossover works similarly, the difference is that the subsequence is now transferred to an empty child and the empty spots in the child sequence are now filled with the remaining cities of the other parent (in the same order as they appear in the parent). For example: P1 = 1-3-5-2-4; P2 = 5-4-1-3-2; SubS1 = - -5-2- ; C1 = 4-1-5-2-3. This way of doing the crossover prevents the problem of duplicate cities that could appear when using the partial mapped crossover. The order crossover allows the transfer of potentially good links between cities of one parent while also taking into account the order of the remaining cities based on the other parent. Furthermore, the randomness of the length of the subsequence might lead to better exploration. Even when there is little overlap this method will ensure that the order of cities is inherited by the children. The main parameter to play with is the length of the selected subsequence, at the moment we used a random length between 1 and the whole sequence. This might be changed by setting the boundaries more strictly.

1.6 Elimination operators

We did not try out a lot of different elimination procedures. The one we chose at the start was lambda + mu elimination which seems to work fine. We also tried out the lambda, mu elimination in which the entire population is replaced with the best children but this required the generation of a lot of children in order to get good results. In the elimination we do not have any parameters that can be chosen because we have opted for a constant population size.

1.7 Stopping criterion

A good criterion to stop could be based on the improvement of the mean objective value of the iterations. For instance look at five subsequent iterations, if the mean objective value does not improve you could halt the process. We did not implement this stopping criterion but instead use a parameter for the number of iterations. We set it at the maximum of 1000 iterations and checked the convergence results of our algorithm.

2 Numerical experiments

2.1 Chosen parameter values

The main parameters to be set are: the population size, the offspring size, the number of iterations and the probability of mutation. For now we based the choice of the parameter values on the knowledge gained from the course lectures and the project details. We played around with the mutation probability in order to better see the effects of the mutation operator and test whether it has significant effects on the performance of the algorithm. In the test run we chose for a population size of 100 individuals, an offspring size of 60% (60 children), a mutation probability of 5% and we let the algorithm run for a 1000 iterations.

2.2 Preliminary results

The evolutionary algorithm completed the 1000 iterations roughly in 9 seconds. From figure 1, it is clear that there is a fast decline in objective value which slows down around 100 iterations. Around 400 iterations there seems to be no further improvement, which means the algorithm got stuck in a local minimum. This can also be seen by the fact that almost from the get go, the mean objective value and best objective value are as good as equal to each other. We flagged this rapid convergence as something that needs to be altered and improved in the future steps of this project. As can be seen in figure 2, our best result in 50 runs was just above 27500 which is as good as the greedy heuristic result. But then again, we only got this value once in 50 runs, so we cannot yet say that our algorithm is proficient in finding the best solutions. The greedy solution is most likely also not the optimal solution anyway.

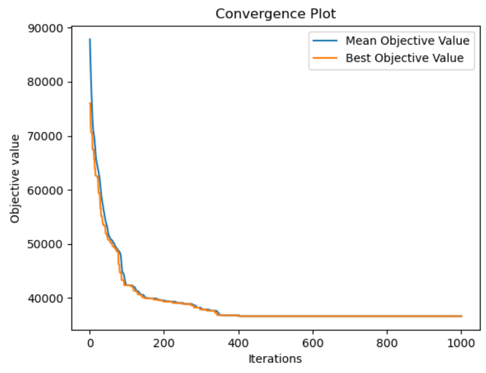

Figure 1: Convergence plot of 1 run over 1000 iterations

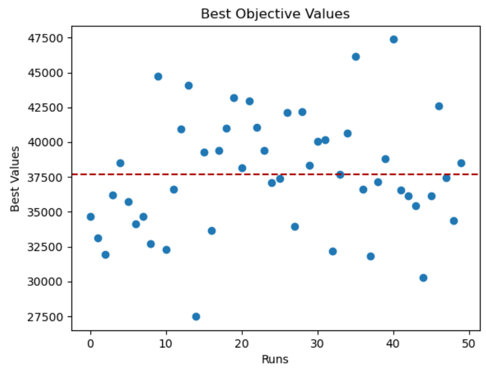

Figure 2: Best objective values over 50 runs

The best values over 50 runs differ quite a lot (a difference of approximately 20000 between the best and the worst result). The average over these 50 runs is around 37500 which is still 10000 above the greedy heuristic result. These results probably indicate that our solutions are trapped in local minima.