Evolutionary Algorithms: Group report

Andru Onciul, Alexios Konstantopoulos, and Alexandros Kyrloglou

November 12, 2021

1 A basic evolutionary algorithm

1.1 Representation

Each candidate is represented with an array of size $numberOfCities - 1$ this means that such a path will always start from the same city and will return to the same one, this city is the first one defined in the file. We considered having all the cities in the array but that seemed redundant and decided to save some space by omitting one of them as well as reduce the dimension of the search space. It should not matter which city is chosen as the path is cyclic. In python the population is represented by the numpy array $population = np.ndarray(dtype = int, shape = (self.populationsize, numberOfCities - 1))$.

1.2 Initialization

The population is initialised by making random permutations of the available cities. Namely initialising the population array with random permutations ($np.random.permutation(individual)$) of the array with every city listed ($individual = np.arange(1, numberOfCities, dtype = int, shape = (numberOfCities - 1))$).

1.3 Selection operators

The ranking selection operator was initially considered, as it provides more control over the selective pressure. Ultimately it was not chosen as it is very computationally expensive. In order for a selection to be made every individual must have its fitness calculated and the whole population must be sorted.

The k-Tournament selection was chosen as the Selection operator because it has both reasonable selection power, meaning it chooses the best candidates, while keeping quite some variation to the population in order to search a big part of the space. The value of k is important as if set too high, only the best solutions will be picked and the algorithm will converge without searching the whole space, but if set too small then the selection is basically just randomly picking candidates.

1.4 Mutation operator

Firstly the scramble mutation was considered as it searches a big part of the population space. Quickly it was discovered that this mutation was mutating too much and as such information in good individuals seemed to be lost, making the algorithm not converge.

Since the algorithm is trying to find a minimal path, the mutation chosen was swap mutation. In this way good candidates keep most of their information without being mutated much and the algorithm is sufficiently spanning the search space in order to find good results. Not all members of the population are mutated as there is a chance of mutation randomly mutating a part of the data-set. We created a parameter *percentageOfSwitches*, that is used to calculate the number of switches per mutation (by multiplying this parameter with *numberOfCities*). A big value will increase the exploration but a too big value will change too much about the good individuals.

1.5 Recombination operator

The cycle crossover operator was considered to be implemented. This operator tries to preserve as much information as possible about the absolute position in which elements occur. It starts by dividing the elements into cycles, which are subsets of elements that have the property that each element always occurs paired with another element of the same cycle when the two parents are aligned [2]. We ultimately decided against this as we found limited documentation on implementing this particular crossover.

For the recombination we need to keep the good information shared by the two selected individuals. We chose a Partially Mapped Crossover (PMX) as seen in figure[1]. This keeps a randomly chosen chunk of the path of each parents in the offspring. From a practical point of view, choose two index at random to define a crossover range.

To create the off-springs, switch the genetic information in that crossover range between the parents. We now have a problem because the offspring are not legal anymore. To fix it, we generate a mapping between the values of in the crossover range of each parent. We can now replace the illegals cities in the offspring by their mapped city.

PMX creates off-springs that are half of each parent, in that way we keep most of the good information from both parents but we keep exploring because the half of each parent that is kept is chosen randomly by choosing the crossover range randomly. If the two parents have similar sub-paths, it is highly likely that the off-spring will have the same sub-path. We do not have parameters for PMX so we can not do self-adaptivity.

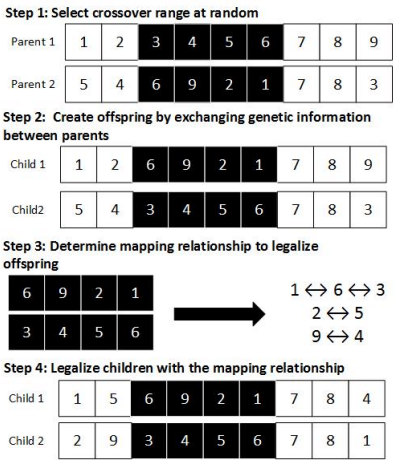

Figure 1: Partially mapped crossover (PMX) [1]

1.6 Elimination operators

The elimination algorithm that was chosen is $(\lambda + \mu)$-elimination with k-tournament in this way both the best elements are chosen and if k is kept low then diversity is also conserved.

We chose not to go with (λ, μ)-elimination because our crossover operator to create offspring is not efficient enough computationally. It would be to slow to compute an offspring population five times bigger than the initial population (as advised in the slides).

1.7 Stopping criterion

In the code there are two parts that make the stopping criterion. The first part is a maximum number of generations, this is defined as *Iterations* and if reached the algorithm will stop as it has gone on for too long, this limit should normally not be reached as it should converge before that. The second and more impactfull part of the stopping criterion is the combination of the parameters *genForConvergence* and *stoppingConvergenceSlope*. Here the *meanFitness* for the past number of generations, defined as *genForConvergence*, is stored. This array is used in a linear regression and the slope of that line is stored. This slope shows how much the *meanFitness* has changed in the past generations. Normalising this slope is also necessary, as the number of cities and the size of the distances will influence the mean value of the path's cost, as such the value of the mean fitness and its rate of change. By comparing it with the initialised *stoppingConvergenceSlope* used as a minimum allowed value, initialised in the beginning, the stopping criterion is fully defined.

We tried at first to compute the difference between the meanFitness value with the bestFitness for each generation. The stopping criteria was a minimum value for this difference. We did this thinking that the program should stop once it converges. By trying it, we realised that it was hard to choose a good minimum value because it changed for a different number of cities. The technique we chose stops when the generation are not getting better anymore without having to manually choose this minimum difference between meanFitness and bestFitness.

2 Numerical experiments

2.1 Chosen parameter values

All the parameters to be chosen and their final values can be seen in figure [2]. Some experimentation in the start was done and a good enough solution compared to the simple greedy heuristic value was not constantly found. Thus it was decided to draw a convergence graph to get some more insight on the workings of the algorithm. By drawing the graph it was found out that the algorithm was converging too fast and as such more exploration was necessary. Then starting with an algorithm that was fully just exploring some convergence was slowly tuned up. By decreasing the values of the *k_selection* and *k_elimination* as well playing with the value of the stopping criterion a good result was reached. Additionally the number of iterations was increased, this value should normally not be reached as the algorithm converges well before that, but it is set there as a limit so the algorithm does not go on forever. Finally the values of the population size was experimented to be low enough that the algorithm can finish at an acceptable time and high enough that multiple experiments will not give a hugely varied results.

2.2 Preliminary results

In the figure [2.2] the convergence graph of the algorithm can be seen. As seen the algorithm runs on average for 10.64 seconds or 234.44 iterations. The algorithm does a fine job on finding a small path in the data-set. It always converges and finds a path with reasonable cost but rarely finds the global optimum. From a memory usage point of view the algorithm is only storing the current population and that with a representation that minimises memory size, as explained in 1.1. The algorithm converges pretty fast and still searches a big part of the data set as the population has quite a big mutation operator. But at the same time this convergence might be too aggressive as the algorithm usually doesn't the global minimum but instead is drawn to local minima. After running the algorithm and testing the best fitness value found was 27805.24097369539 this is not the global minimum as we know the optimal path has an approximate cost of 27,200.

The variation of the final results when running the algorithm multiples times is not too big as the population size is kept high. In this way we have a good representation of the whole data-set as a starting point. The actual values of these test can be seen in figure [4] and [5]

```
#VALUES OF HYPER-PARAMETERS
    #INITIALISATION
    populationsize = 300 #size of population
    iterations = 500 #maximum number of iterations
    numberOfCities = 29 #number of cities (automatically filled in with the *.csv name)
    #SELECTION
    k_selection = 2 # the value of k in k-tournament during the selection operation
    #VARIATION
    percentageOfSwitches = 0.1
    mutation_proba = 0.4 # the probability of mutation occurring
    crossover_proba = 0.8 # the probability of crossover occurring
    #ELIMINATION
    k_elimination = 3 # the value of k in k-tournament during the elimination operation
    #STOPPING CRITERION
    genForConvergence = 5 # maximum number of consecutive generations
                          # where there is little to no progress
    stoppingConvergenceSlope = 1e-05 # minimum value of the slope
#RESULTS OF EXECUTION
I: 263 timeleft: 291.0384874343872
meanObjective: 28239.617417098056 , bestObjective: 27805.24097369539 diff: 434.37644340266706
```

Figure 2: Values of the parameters chosen and result

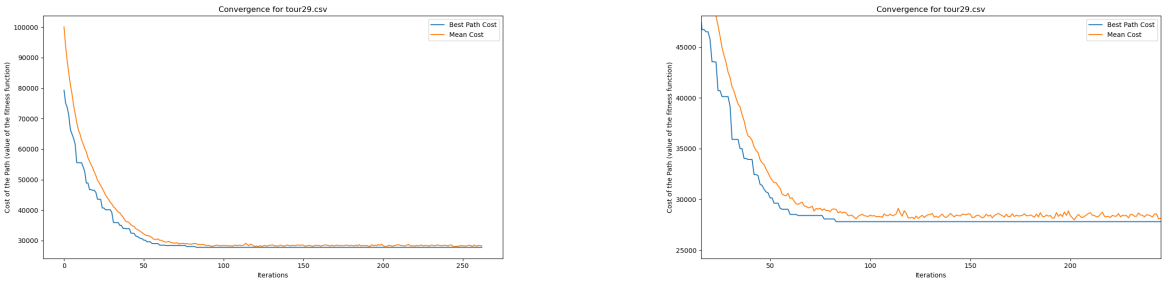

Figure 3: Convergence graph

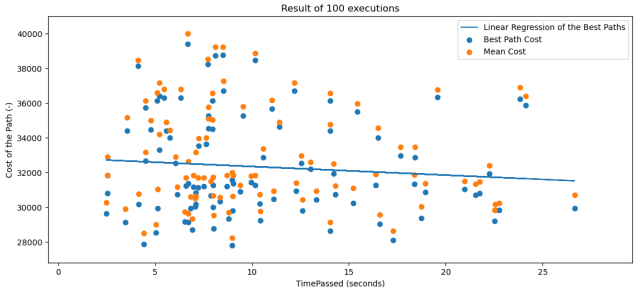

Figure 4: 100 executions of Genetic Algorithm

```
#Values of 100 results
Mean Cost: min max mean variance
27805.24097369539 40002.51489255131 32892.
                  231254832644 8037074.
                  256234057
best Cost: min max mean variance
27805.24097369539 39429.6148096179 32322.
                  382711219685 8267312.
                  49599916
timepassed: min max mean variance
2.487497329711914 26.640767574310303 10.
                  637593846321106 33.
                  19708040318136
```

Figure 5: Results of 100 test

References

[1] Example of PMX implementation:
`https://www.researchgate.net/figure/Example-of-partially-mapped-crossover_fig1_312336654`

[2] Introduction to Evolutionary Computing. A.E. Eiben, J.E. Smith. 2015 Natural Computing Series.