**Report about GA when applied over N queen and TS problems**

**By Ephi Frankel and Yarden Levinger**

GA approach

As discussed throughout the lectures, GA requires the following main building blocks in order to be applied:

*Chromosome:* a possible solution of the problem.

*Fitness function:* measures how “fit” is a chromosome.

*Selection mechanism:* defines how does 2 chromosomes are selected for cross-over.

*Cross-over function:* defines a re-combination over 2 chromosomes into 2 offspring.

*Mutation function*: applied over a chromosome causes at a very low probability random change at the chromosome.

Additionally, *elitism* is a mechanism in which a percent of the strongest chromosomes passed into the next generation at the expanse of the same percent from the weakest chromosomes.

**Part1 – N Queen Problem**

Introduction

A very well known problem in the domain of optimization problems is the N queen problem. Given a board at size N X N, find a placement for N chess queens such that no queen will threat another queen.

In this report we aim to show the results when trying to solve this problem for N=8 with GA. We will also use a randomized CS (Constraints Satisfaction) algorithm that finds an optimal solution as our baseline results comparison.

Baseline approach

There are many possible ways to find a viable solution for the N queen problem. Let us review some of them in terms of speed/performance in order to select the most reasonable as baseline comparison to the GA algorithm.

1. Naïve Brute Force: Generate all possible permutations of N queens at the N X N board for each permutation, check if there exists a threat in the board, select a permutation that has no threats.

Complexity analysis: Factorial at N

Algorithm is complete – that is, a solution is promised to be found.

1. Recursive Algorithm: Recursively generate boards that maintain the valid solution throughout their construction (if there is a violation, try another path).

Complexity analysis: Exponential at N

Algorithm is complete – that is, a solution is promised to be found.

1. Randomized Algorithm: Generate a random permutation that is a possible solution for the N queen problem such that there is a queen at each row. For each queen from row i: N-2 to 0, move it on the same row if it threats any queen at row i+1,i+2, .., N-1, if a valid position is not found after checking each cell at row i, re-apply the algorithm, if valid position were found for all rows return the solution.

Complexity analysis: C\*N^2 where C is the number of random permutations needed to find a solution. Note: C is not bounded – this implies that the algorithm is not complete, the algorithm will only halt if a solution was found, otherwise, will continue indefinitely. However, if the algorithm halts, the solution is optimal. Also, in practice C is usually small.

Note: We might bound C with a factorial of N by only generating new permutations at each iteration of the algorithm, in this case the algorithm will be complete, with worst case complexity of Factorial at N. However, in this report we are using the unbounded version, and thus, it is an incomplete algorithm.

Since in practice C is small, then, the runtime of (3) is preferred over the other options. So, we chose it as our baseline.

Baseline results:

When the baseline approach is applied, optimized solution might be found as fast as 0.004 seconds. However, due to the reasons discussed above, it is possible that no solution might be found at all.

Fig. 1 shows an outcome given the baseline method.

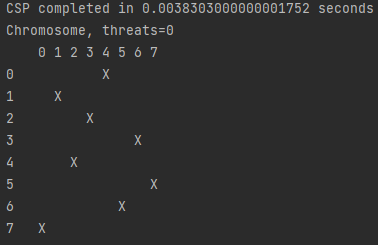


Fig. 1 – An example for a baseline result.

8-Queen GA

*Chromosome:*

First, we consider the following 1-D representation for the problem such that each queen is allocated a single row. To be precise, we say that each queen belongs specifically to row and that queens by definition cannot share rows. By this definition we reduce a dimension of the problem since now no 2 queens can ever be located on the same row, so for each we define the location of at row . So, we have that as our representation.

Next, we consider that then each can be represented with exactly 3 bits (000 = 0, 001 = 1, …, 111 = 7). Due to this, we define the Chromosome to be such that is the 3 bit representation for the location of , is the 3 bit representation for the location of , …, is the 3 bit representation for the location of . We note that by construction the chromosome is valid for any sequence of bits , then, **there exists no cross-over/mutation that could cause such chromosome to become invalid.**

*Fitness function:*

For the fitness function we consider the number of threats generated on the board. We do it as follows:

For each count the number of that are either on the same columns or on the same diagonals such that

We say that and are on the same column if =

We say that and are on the same diagonal if = or =

By this definition **a chromosome is more fit if the number returned by this function is lower.**

*Selection mechanism:*

The selection mechanism we used is a roulette wheel selection. Before applying this selection, we are inverting the fitness function such that each chromomere fitness F will correspond to where is a number close to 0 to avoid division by 0.

*Cross-over function:*

The cross-over function we chose is the uniform binary cross-over. That is, given two chromosomes and the cross-over function creates two offsprings:

and such that at probability p=0.5, and with probability 1-p, for all i.

*Mutation function:*

The mutation function we chose is binary flip, that is, given chromosome , each is flipped with probability p.

*Elitism:*

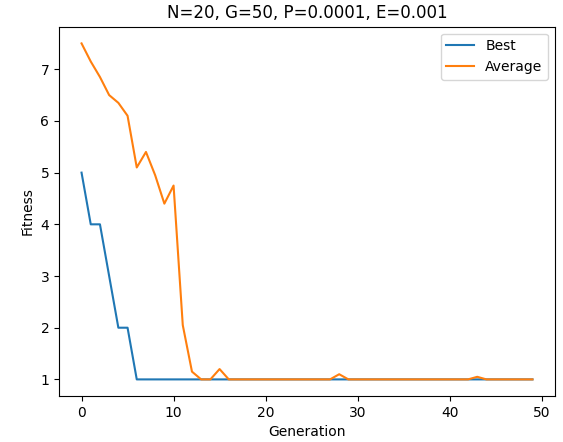
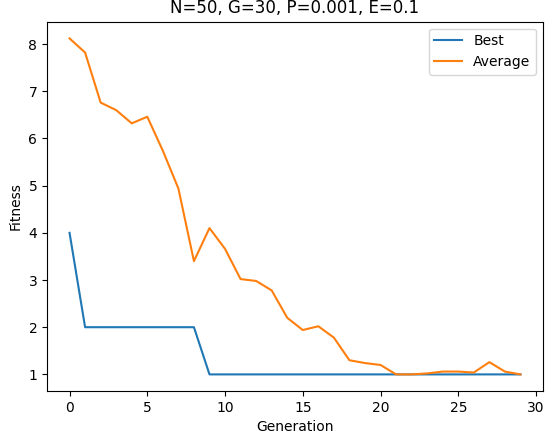
Elitism is defined over a percentage of the population, but at least 2 are always selected.

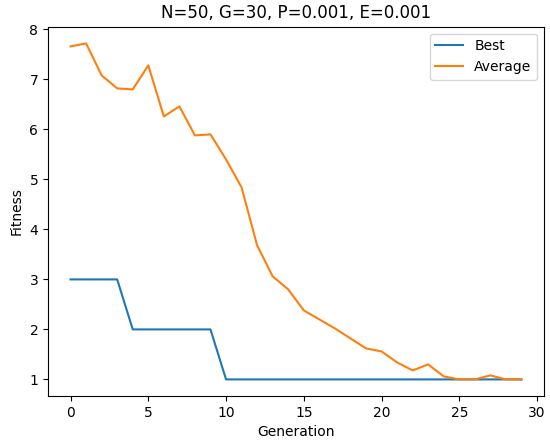
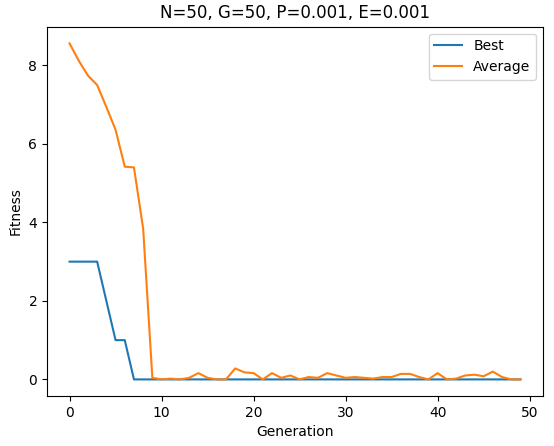
Experiments

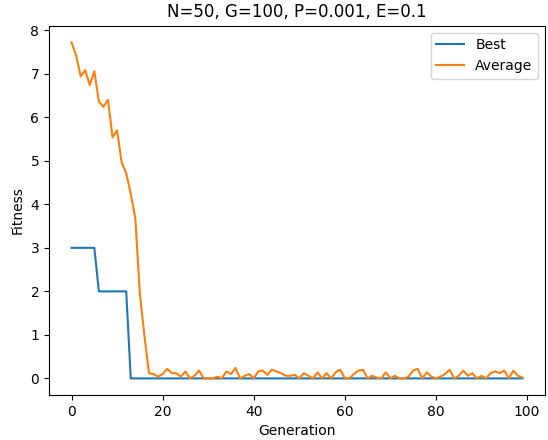
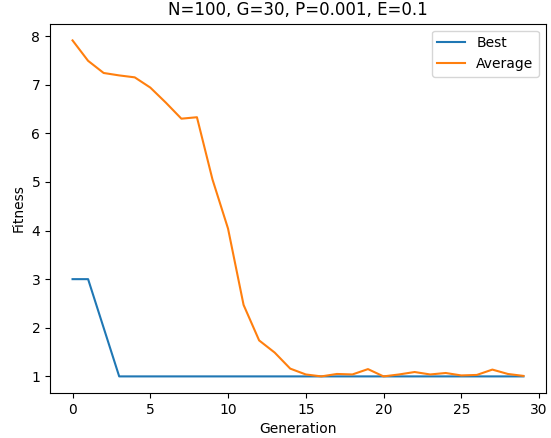
The GA paradigm provides many hyper-parameters to fine tune when performing experiments. In fact, we may consider the chromosome, fitness, cross-over, mutation and selection mechanism as hyper-parameters as well. We are justifying the chromosome design and fitness we selected by their simplicity and correctness, so we have not experimented with different possibilities. Additionally, we justify the selection of cross-over and mutation function due to the knowledge we gained in the lectures, that is, that the standardized cross-over and mutation of bitwise chromosome is always valid is as described above.

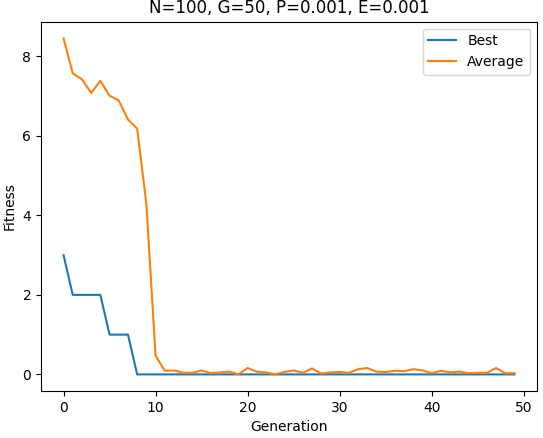
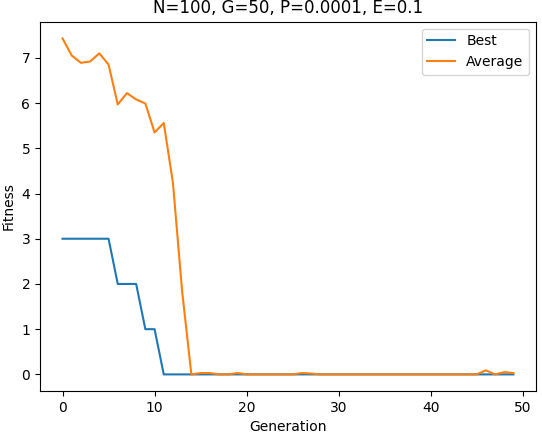
The hyper-parameters we adjusted then are as follows: N = the size of the chromosome population, G = the number of generations, P = the probability for mutation, E = the elitism percentage. Below are the promising results when we experimented over following:

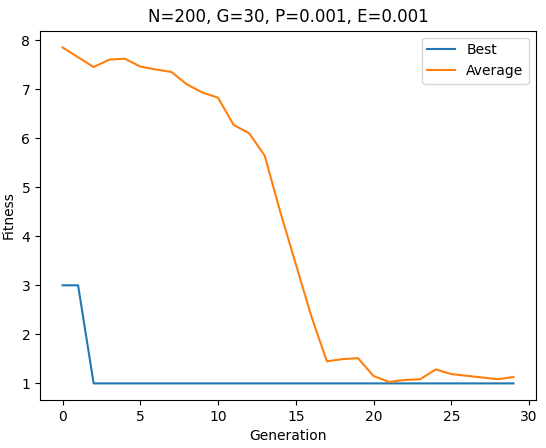
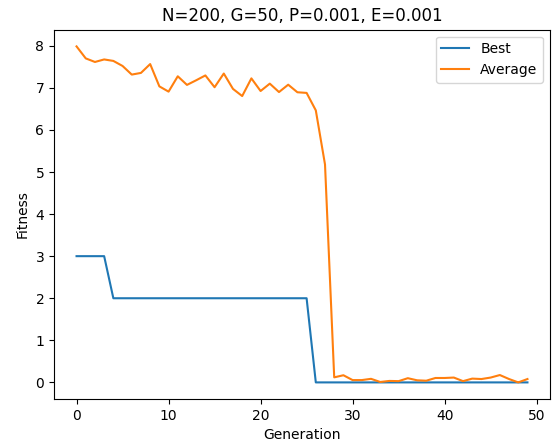
N = [10, 20, 50, 100, 200, 500]  
G = [10, 30, 50, 100, 200]  
P = [0.01, 0.001, 0.0001]  
E = [0.1, 0.001]

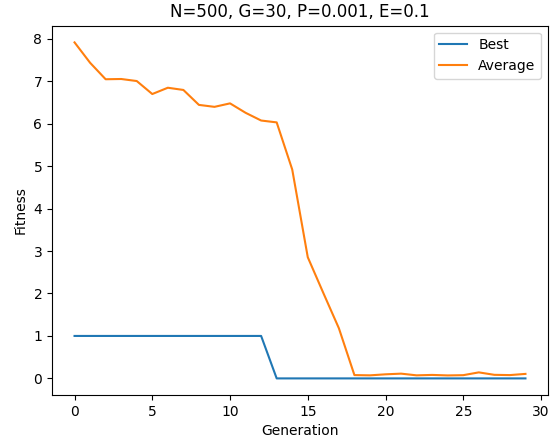
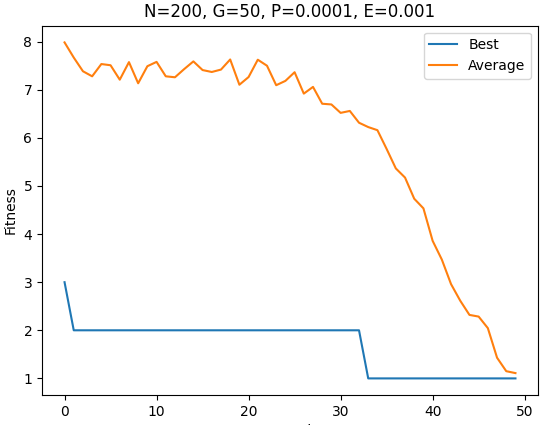
 

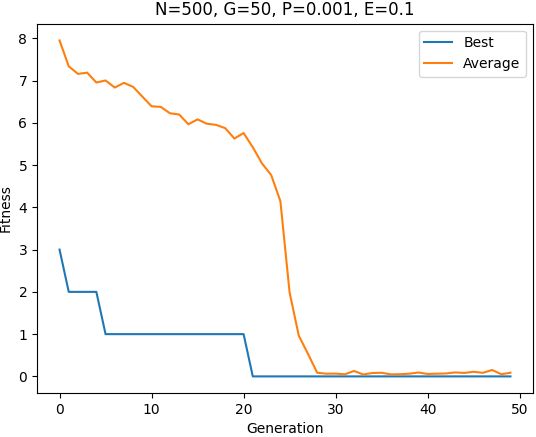
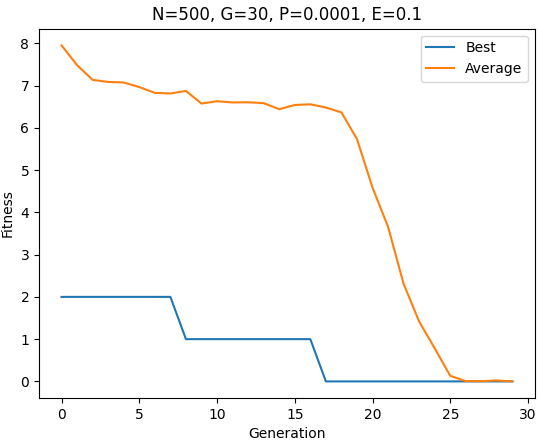
 

**** ****

****

****

Results analysis

As can be seen over the graphs of the promising results, many of the configurations reached sub-optimal solutions (i.e: fitness = 1). In many of the results where an optimal solution was reached, it was probably due to a powerful mutation that almost immediately spread throughout the population (for example, consider the result for N=200, G=50, P=0.001, E=0.001). It seems that as the population size increases, so does the stability of the algorithm resulting with gradual improvement (for example, consider the result for N=500, G=30, P=0.0001, E=0.1). However, in terms of speed, even the quickest configurations (e.g: N=20, G = 50) took 0.02 seconds which is more than an order of magnitude slower than the average run time of baseline algorithm, while the slower and more stable configurations at the GA perform in the order of seconds.

Conclusion

By analyzing the results, we may understand that the baseline algorithm is on average better than using the GA approach for the 8-Queen problem. However, we consider the following in mind: first, that the baseline approach is not complete, that is, it MAY stuck (or at least timeout with an invalid result). Second, the GA algorithm MOST of the time will return a good approximation of the optimal result (i.e: solution with only a single threat). We also note that the GA was applied here for N=8, under the assumption that as N increases the probability of generating boards such that a viable solution is reachable from them decreases, we have that the C in our baseline algorithm complexity analysis increases. Then, due to the increase of C we assume that for big enough N the performance of the GA in term of run-time will be better than that of the baseline algorithm presented in this report while providing good solution approximation.

**Part 2 – Travelling Salesman Problem**

Introduction

The Travelling Salesman Problem (also called **TSP**) asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” (Taken from TSP article in Wikipedia).

It is an NP-hard problem, which means that it is suspected that there are no polynomial-time algorithms for solving this problem, but that has not been proven.

In this report we aim to show the results when trying to solve this problem with N=48 cities using Genetic Algorithm.

Baseline approach

There are many possible ways to find the best solution for TSP. Most of them are just a modified version of the Naïve Brute Force, with valid permutations. Let us review some of them in terms of speed/performance to select the most reasonable as baseline comparison to the GA algorithm.

1. Naïve Brute Force: Generate all possible permutations of N cities that satisfy the constrain of the problem (each city appears exactly one). for each permutation, calculate the full route Euclidean distance, and then find the route with the minimal distance.

Complexity analysis: Factorial at N (for example, in our report N = 48 -> N! = 1.24139 possible permutations).

Algorithm is complete – that is, a solution is promised to be found.

1. Greedy Algorithm: Creating N paths in this order: path number i is a path that built using the city i first and then always adds the city that is the closest to the last city added, and is available (not added already) – very fast, have no lower bound to the correctness of the solution and as no option for improvement (with our cities, the best path found is with length of 39236.8848).

TSP GA

*Chromosome:*

We represent the chromosome in the simplest and most naive way, as a list of the cities the salesman is passing in his travel. This representation way will create some constrains on the genetic functions.

**Worse mention: Because of the circular path there are 48 different ways to represents the same path, which may cause repetitions of chromosomes in a population (distance-wise). To prevent that, We also tried to represent a chromosome using only 47 cities while one city has a constant place, but for some reason(what is the reason?), this representation degraded the performance.**

*Fitness function:*

For the fitness function we consider the sum of the Euclidean distances of the path (including the return to the first city).

By this definition **a chromosome is more fit if the number returned by this function is lower.**

So, **the selection probabilities are inversely proportional to this fitness function**.

That gives another benefit, a route that is two times shorter get a selection probability two time bigger.

*Selection mechanism:*

The selection mechanism we used is a roulette wheel selection. Before applying this selection, we are inverting the fitness function such that each chromomere fitness F will correspond to and then normalized for the sum of probabilities will be 1.

*Cross-over function:*

The cross-over here is more like circular double point crossover, by partially changing the order of genes in a parent’s chromosome according to the parent’s chromosome. That is, for each cross over function generates 3 random integer numbers, where the first number is the range’s length we want to swap that is in range , the second number is the first index to swap in the first parent ( ) and the third number is the first index to swap in the second parent ( ). Both the second and the third random numbers are in range .

If a range is gone after the chromosome size, we keep it circularly and then we take the cities in those ranges.

For example, with chromosome size of 10 (6 cities in a route), we generate the following random numbers: range’s length = 4, first index in = 2, , first index in = 7.

And the parents’ chromosomes are:

Now, after taking the two range we want to swap between, we are looking for intersection of them (the numbers that appear in both, in our example, [2 4 5]).

After finding the intersection, we start building the two children. Each child is built using the corresponding parent with part of the range we want to swap from the other parent (as much as we can without becoming invalid).

For child 1: we set the chromosome values:

Then erase the numbers in the range we want to swap:

Then we insert the numbers from the selected range from the other parent that are also appears in the intersection, keeping the same places as in the other parent’s range (in our example: [4 2 \_ 5]):

And then we insert the numbers from selected range that don’t appear in the intersection, in the empty places, keeping the same order as before:

Now we got a new child. Now we are doing the same for the second child.

The validity of the children is a result of swapping only numbers that are appears in both ranges.

We assume, from the problem definition that the parents are valid chromosomes. Therefore, no index is missing, and no index is appearing twice. So, our cross-over, by only changing the genes’ order, cannot make the chromosome invalid.

*Mutation function:*

The mutation function we chose is a randomly two cities swap with probability of p.

*Elitism:*

Elitism is defined over a percentage of the population.

Experiments

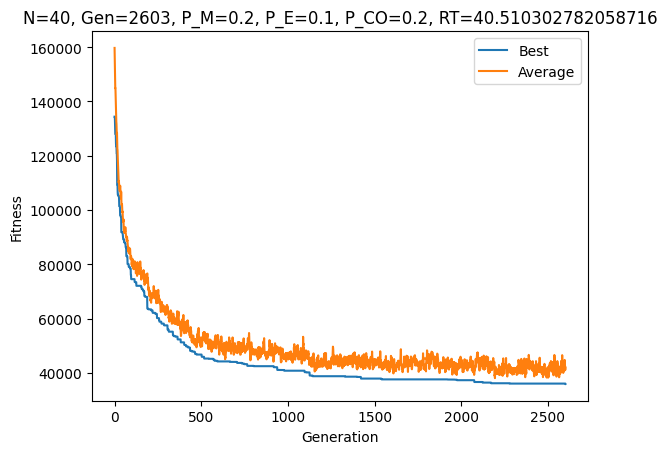
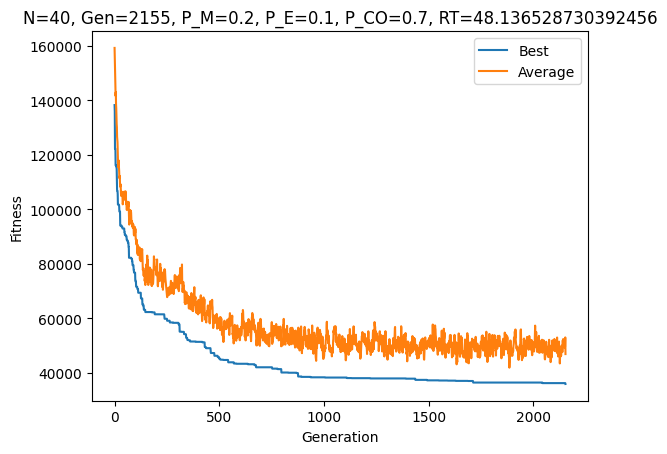
The GA paradigm provides many hyper-parameters to fine tune when performing experiments. In fact, we may consider the chromosome, fitness, cross-over, mutation and selection mechanism as hyper-parameters as well. We are justifying the chromosome design and fitness we selected by their simplicity and correctness, so we have not experimented with different possibilities.

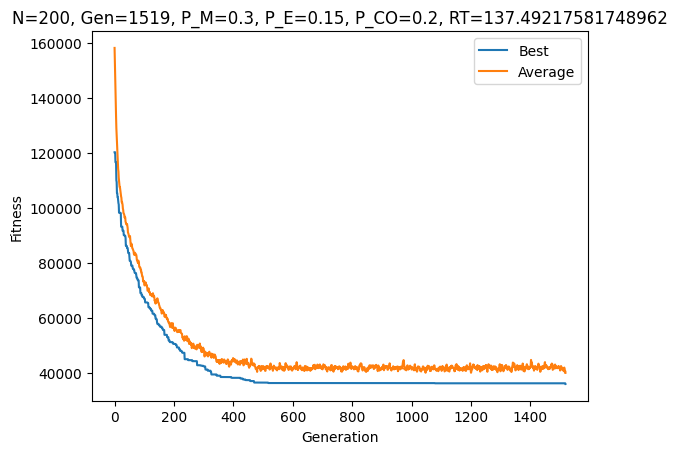
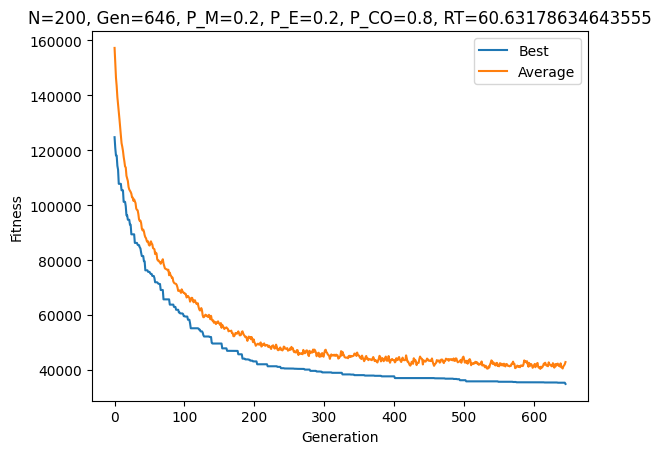
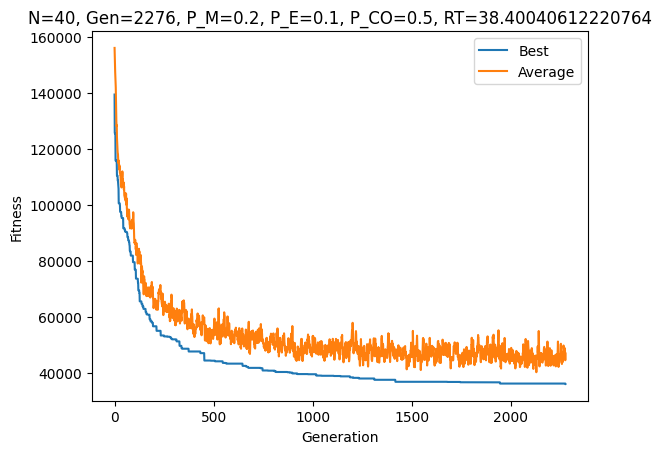
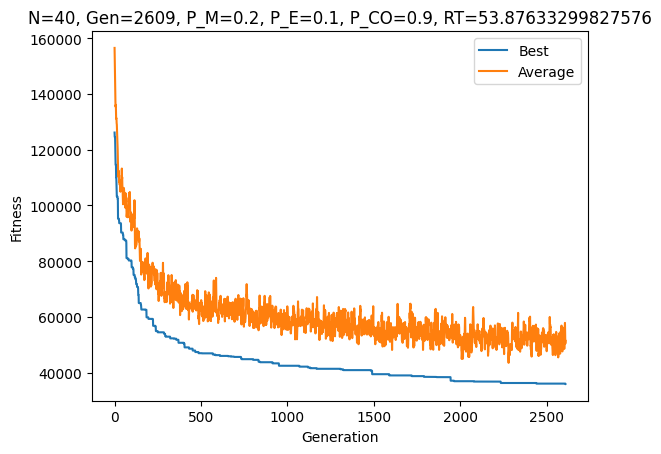
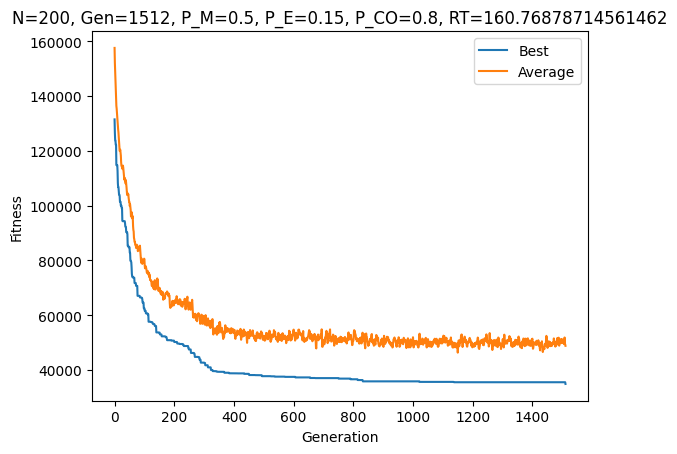
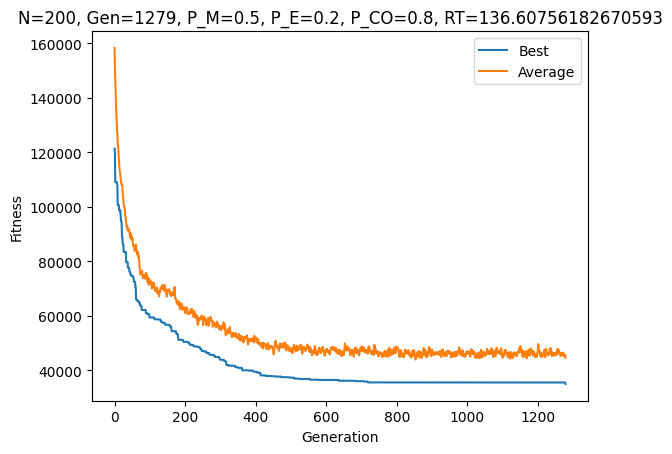
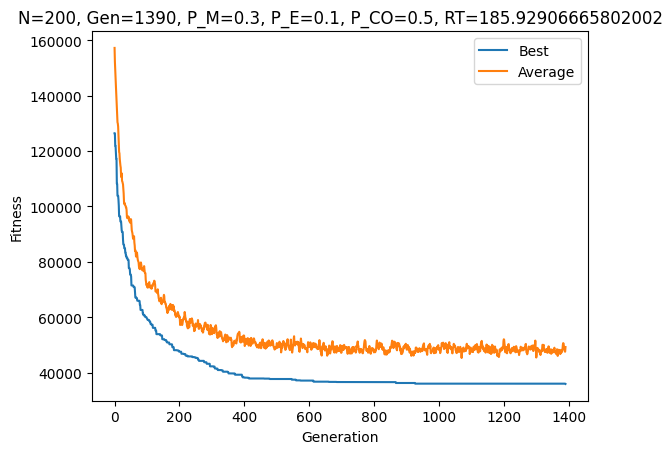
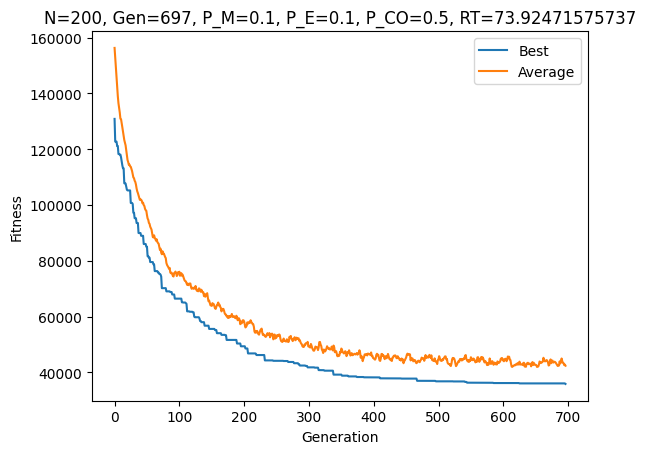
As said before, after trying using a constant place for one of the cities, to pervert double path chromosomes and to reduce the search space, we got worse results then the naïve representation.

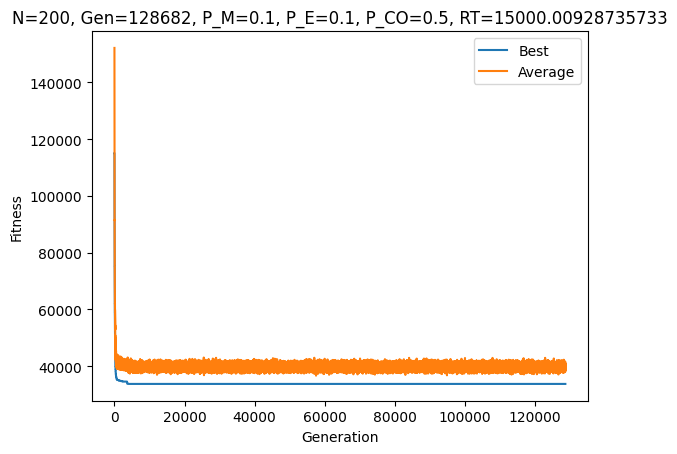
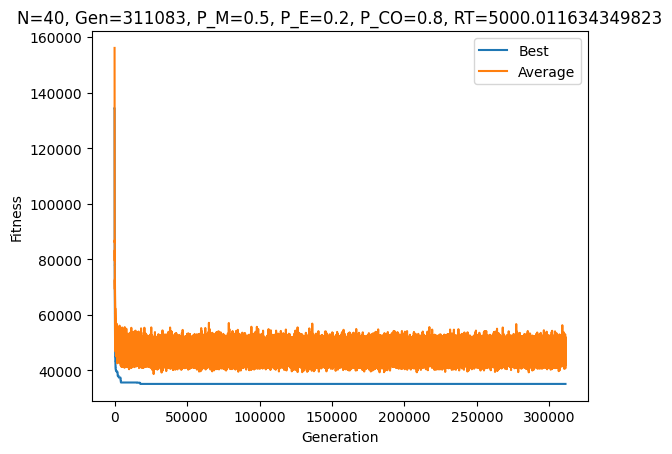
The hyper-parameters we adjusted then are as follows: N = Number of chromosomes in our population, P\_M = Mutation Rate, P\_E = Elitism Ratio, P\_CO = Cross-Over Rate. We run the hyper parameters until reaching a path of length 36000 or below. In the graphs’ titles you can see beside the hyper parameters, the following:

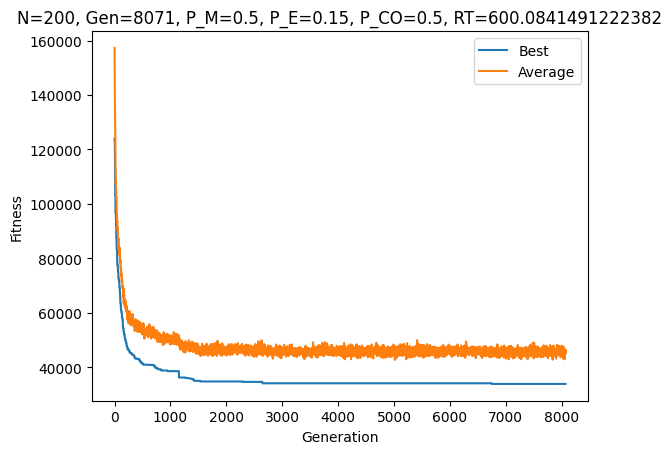
Gen – number of generations required to reach the threshold (= 36000), RT – Runtime measurement (in seconds)

Below are the promising results:





Try To Reach Best Results:



Results analysis

As can be seen we had promising results (we consider a result as promising if the best fitness is no more than 36000) for **population at sizes 40 and 200**. We also experimented on populations of sizes in range (40,200), but, the results were either the same or degraded (as the population size increased until stabilizing again for a good result when reached 200 and degrading when further increased – we assumed such stabilization for large population due to divergence of chromosomes). Surprisingly, we saw that as the **population size was smaller** (N=40) we needed to **reduce the cross over rate** (P\_CO = 0.2) while **keep** **relatively high mutation rate** (P\_M=0.2) in order to have better convergence of the GA algorithm in relatively quickly (RT=40.5). We assume the reason for this is that when the population is small the type of cross-over we used wouldn’t allow good enough divergence in the chromosomes while the mutation did. Moreover, we didn’t expect low population size to yield any good approximation let alone a promising one. We believe the reason for this is that mutation is not too “strong” in the sense that mutation only causes 2 cities on a tour to be swapped allowing possibly allowing gradual improvement in a small population. However, we **weren’t able to achieve best fitness <= 35000 over such population size most of the time**.

When the **population size was larger** (N=200), we needed to **increase the cross over rate** (P\_CO=0.8), but still **keep relatively high mutation rate** (P\_M=0.2), while having **significant elitism** (P\_E=0.2). We observed good convergence of the GA algorithm, however, slowly ( RT = 60 ) in comparison to smaller population size. Additionality, the results with larger population were more stable, that is, **most of time we were able to achieve best fitness <= 35000** and at some instances **even lower than 34,000**.

We assume the **key reason for this performance is the cross over function when applied in conjunction with large population size**. To be precise, we speculate that the cross-over function somehow allows “building blocks” to be carried over the generations in this problem, however, when the population is too small the probability for such building block to formulate is very small –we think that this explains the reason for the better performance with high population size. We also deduce that for this problem using a relatively non-destructive mutation aids in finding better approximations.

After a lot of configurations have been tried, we saw two were much better than others: N=40, N=200.

With N = 40 we got much better runtime performance to reach 36,000, but when trying to go below that, we got stuck most of the times. In the other hand, N = 200 have worse runtime performance (not too much) but is more stabilize most of the time we get better result, even lower than 34,000.

Conclusion

By analyzing the results, we can see a big advantage when applying GA trying to find a good solution to the TS problem. While the Naïve algorithm (using brute-force) has a complexity of N! = 1.24139 (N=48) while providing optimal solution, the greedy algorithm provides an approximated solution as a tour of length 39236.884841918945 (with our input cities), with no option for improvement. However, as reported above the GA find approximations as low as 34,000 (we believe that for our problem the optimal tour length is no more than 33,600) at a run-time which is much closer to that of the greedy algorithm than to that of the Naïve algorithm.