Fall 2020 Big Data Analytics 10351

Final Project

Solving NP-Complete Problems Using Evolutionary Algorithms

Mor Benayoun 305281743

Aviv Yampolsky 315028688

Nir Ben Dor 204588388

1 INTRODUCTION

In the computer science field some computational problems are considered NPC, or Non-deterministic Polynomial time Complete.
NP can be defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. a problem *np* in NP is also NP-Complete if every other problem in NP can be transformed into (or be reduced to) *np* in polynomial time.

our project focuses on the issue of finding an approximation to a specific type of problem- the traveling Salesman Problem (TSP).

In order to approximate the optimal solution we will use a specific type of Evolutionary Algorithm - the Genetic Algorithm (GA).

2 TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem is a non-deterministic polynomial-time problem in computational complexity theory which asks the following question: “Given a list of cities and distances between each pair, what is the shortest possible route that visits each city and returns to the origin city?”

There are a lot of different routes to choose from, but finding the best one - the one that will require the least distance, or cost - is the real challenge.

TSP is classified as NP-hard because it has no quick and easy solution.
the complexity of calculating the best route will increase when you add more destinations to the problem.

the problem can be solved by finding every possible route from the source to the destination and then determining the shortest one.
The problem is that as the number of cities increases, the corresponding number of possible routes grow at an exponential rate.

For example - with only 10 destinations, there can be more than 3,000,000 routes. with 15 destinations, the number of possible routes exceeds 1 trillion!

In our project we will deal with MTSP, the Metric type of TSP - which satisfies the Triangle Inequality.

3 GENETIC ALGORITHM

Evolutionary Algorithms uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection.

In our project we will use a specific type of EA - the Genetic Algorithm.
Genetic Algorithms are inspired by Charles Darwin's theory of natural evolution and imitate the process of natural selection, where the fittest individuals in a population are selected for reproduction in order to improve the next generation and produce better offsprings.

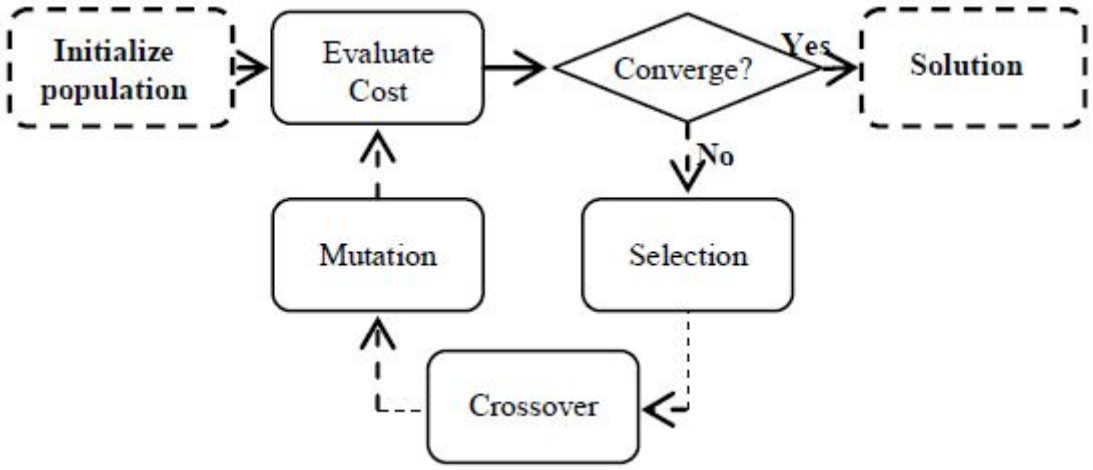
The main use of GA is to generate solutions for optimization and search problems, in order to do that it uses 3 biologically inspired operators:
Selection, Crossover and Mutation.

In biology:

Selection is the preferential survival and reproduction (or preferential elimination) of individuals with certain genotypes, by means of natural or artificial controlling factors.

Chromosomal Crossover is the exchange of genetic material between two non-sister chromosomes that result in recombinant chromosomes.

Mutation is an alteration in the nucleotide sequence of the genome of an organism. mutations play a part in both normal and abnormal biological processes, including: evolution, cancer, and the development of the immune system.



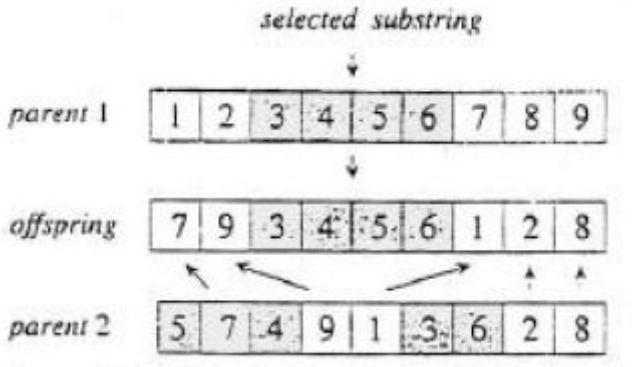
3.1 Crossover Operator

We implemented and compared 3 crossover operators: order, cycle, and position-based.

Order Crossover (OX):

Procedure: OX

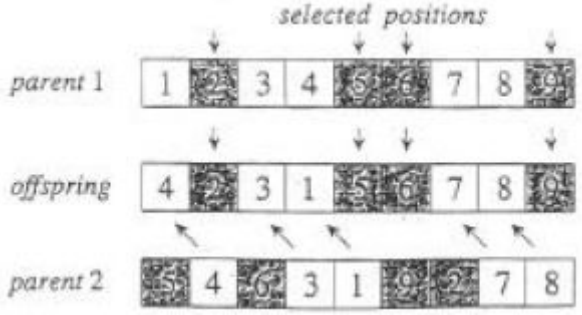
1. Select a substring from a parent at random.
2. Produce a proto-child by copying the substring into the corresponding position of it.
3. Delete the cities which are already in the substring from the 2nd parent. The resulting sequence of cities contains the cities that the proto-child needs.
4. Place the cities into the unfixed positions of the proto-child from left to right according to the order of the sequence to produce an offspring.



Position-Based Crossover (PX):

Procedure: PX

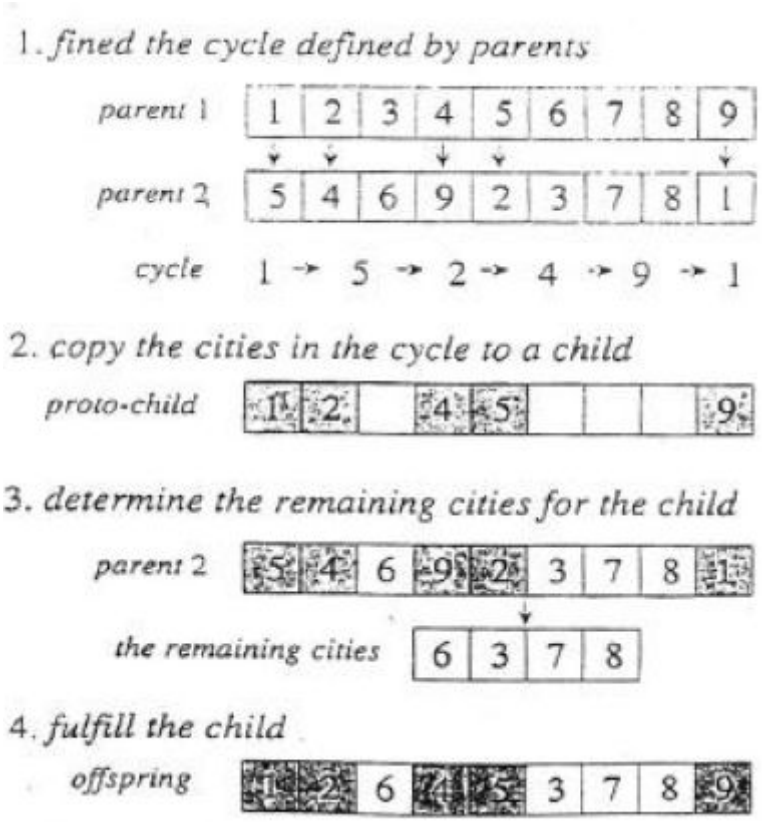
1. Select a set of positions from one parent at random.
2. Produce a proto-child by copying the cities on these into the corresponding position of the proto-child.
3. Delete the cities which are already selected from the second parent. The resulting sequence of cities contains the cities the proto-child needs.
4. Place the cities into the unfixed position of the proto-child from left to right according to the order of the sequence to produce one offspring.



Cycle Crossover (CX):

Procedure: CX

1. Find the cycle which is defined the corresponding positions of cities between parents.
2. Copy the cities in the cycle to a child with the corresponding positions of one parent.
3. Determine the remaining cities from the child by deleting those cities which are already in the cycle from the other parent.
4. Fulfill the child with the remaining cities.

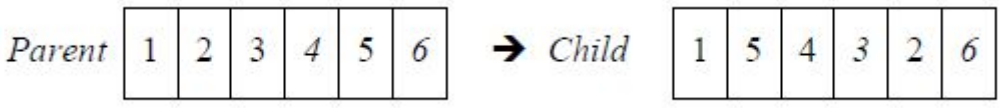


3.2 Mutation Operator

We implemented and compared 5 Mutation operators: reverse sequence, twors, insertion, center inverse, and partial shuffle.

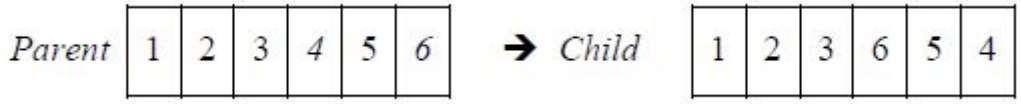
Reverse Sequence Mutation (RSM):

In the reverse sequence mutation operator, we take a sequence S limited by two positions i and j randomly chosen, such that i<j. The gene order in this sequence will be reversed by the same way as what has been covered in the previous operation.



Twors Mutation (TWORS):

Twors mutation allows the exchange of position of two genes randomly chosen.



Insertion Mutation (IM):

Insertion Mutation randomly chooses a node, and inserts it at another random position, shifting the rest of the nodes appropriately.

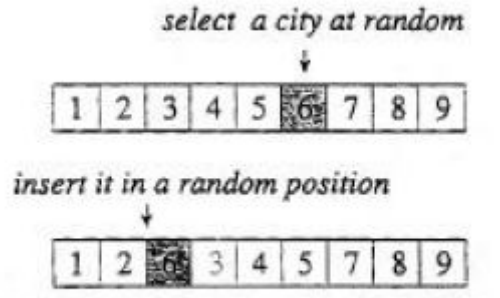
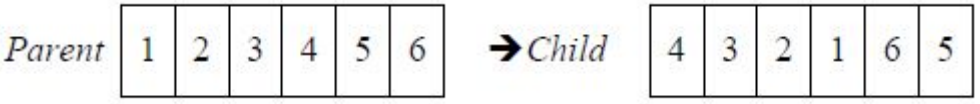


Figure 3.21. Illustration of insertion mutation.

Center Inverse Mutation (CIM):

The chromosome is divided into two sections. All genes in each section are copied and then inversely placed in the same section of a child.



Partial Shuffle Mutation (PSM):

The Partial Transfer Shuffle (PSM), changes part of the order of the genes in the chromosome.

```
Input:  Parents  x=[x1, x2, ....., xn]
        and P is Mutation probability
Output: Children x=[x1, x2, ....., xn]
---------------------------------------

i = 1;

Repeat

      Choose p a random number between 1 and P

      if  p < P  then
        Choose j a random number between 1 and n;
        Permute (xi, xj);
      End if

Until i ≤ n
```

5 ENVIRONMENT

We worked with Python on Ubuntu and Windows OS.
We used Pyspark for parallel computing, Numpy for numerical computations and Matplotlib for data visualization.

6 DATASETS

TSPLIB
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95
https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html

*For our testing we've used a dataset of **48** cities, with an optimal result (route) of **33,523**.*

7 EXPERIMENT GROUP 1

Trying every combination of **3** Crossover operators and **5** Mutation operators,
we conducted **15** different experiments and repeated each one of them **20** times,
finally averaging the results for a representing sample, in order to eliminate random initialization chance, and early convergence to local minima.
Each batch of 20 repeats of an experiment was run using PySpark in parallel.

GA Parameters:

POP_SELECT = 0.2
20% of the best population out of the current generation was used for crossover.

POP_CROSS = 0.9
90% of the next generation was generated using crossover.

POP_MUT = 0.1
A 10% mutation chance was introduced in the population generated using crossover.

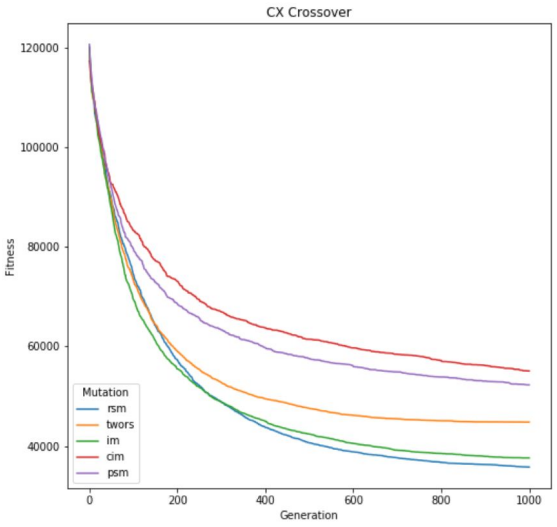
POP_ELITE = 0.1
Finally, in order to achieve constant improvement and to avoid local minimal,
we inserted 10% of the current generation elite population to the next generation.

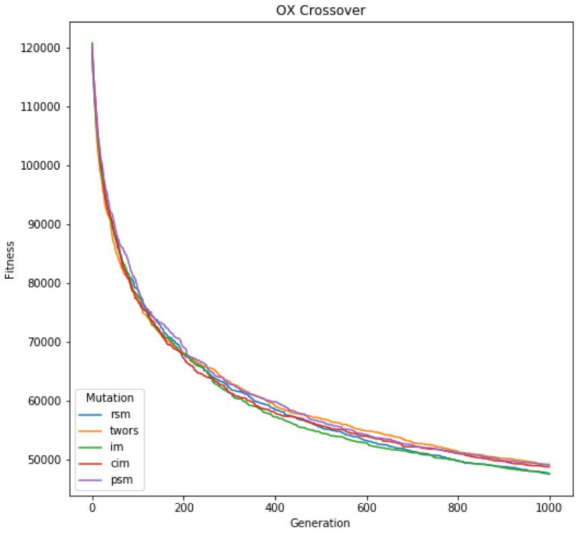
POPULATION: 1000
GENERATIONS: 1000

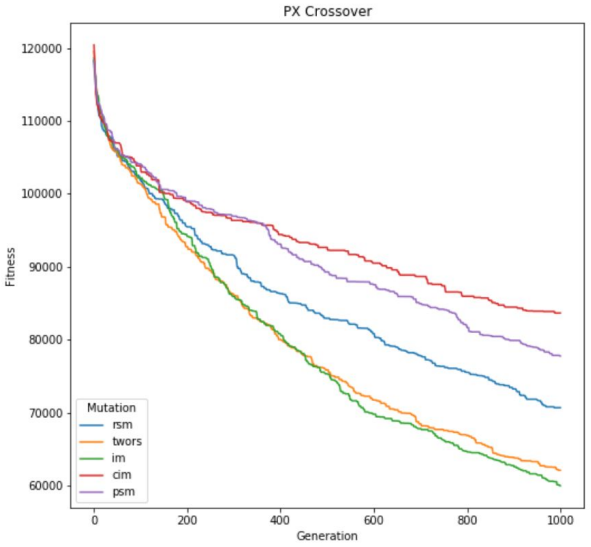
As we can see the GA search space is a tiny fraction of the possible solutions search space.

Number of cities	Search space	GA search space
48	1.241E+61	1E+6
	100%	8E-54 %

7.1 Results







The best result of approximately **36K Fitness Score** (path length)
was achieved using **CX** Crossover operator in combination with **RSM** Mutation operator,
This result is just shy of the optimal result of **33,523.**

Despite only covering less than **1% times -54 orders of magnitude** of the search space,
we managed to get within **7%** of the optimal result.

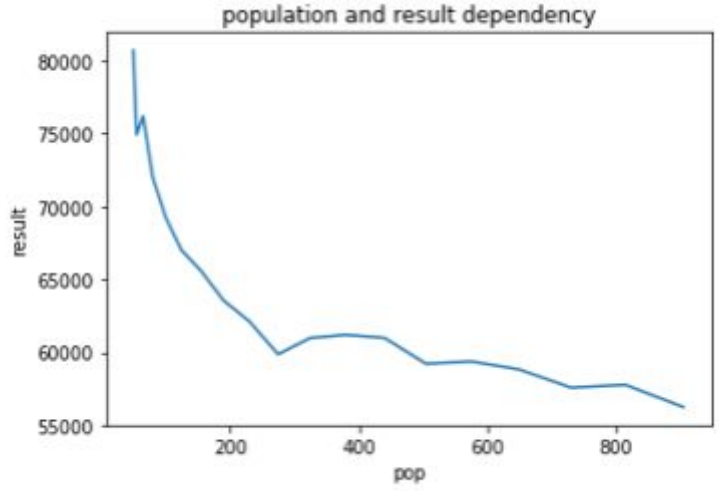
8 EXPERIMENT GROUP 2

In the following experiments we attempted to optimize the achieved result and the alogrithm's runtime. We tried to study the parameter behaviour in different situations.

In each of the following experiments, we've used the **CX** and **RSM** operators that were found to be optimal in the previous experiment. Furthermore, at each step we've continued to build upon the optimal parameters that were found in the previous steps.

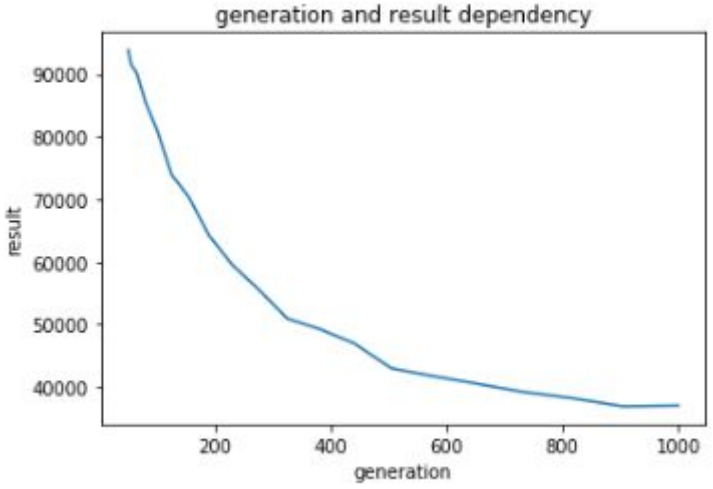
8.1 Population

In this experiment the goal was to observe the effect that the population size has on the result, and find the optimal population that yields the best **cost-benefit ratio**.
A local optimum was chosen, and can be seen at **230**.



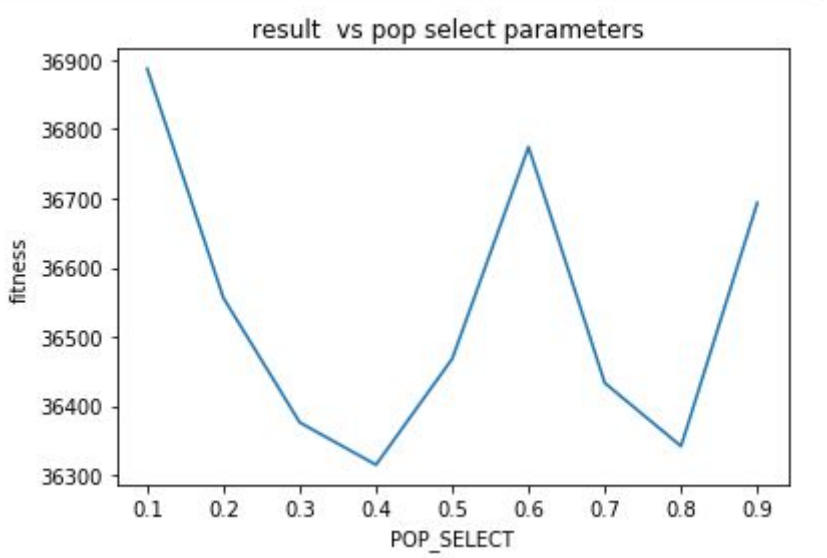
8.2 Target Generation

In this experiment the goal was to observe the effect that the number of generations has on the result.
In the graph below we can clearly see that after **905** there is no real improvement,
and that we've reached a good approximation of **37,025**.



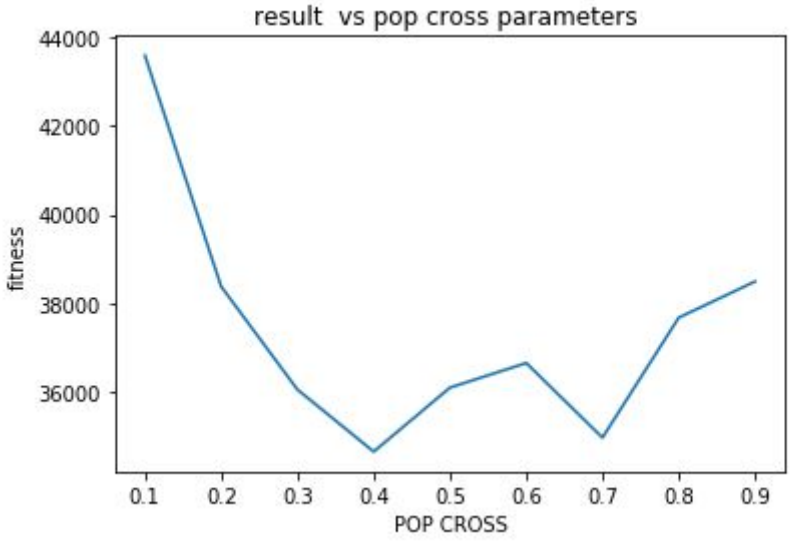
8.3 Selection

In this experiment that goal was to observe the effect that the selection ratio in the population has on the result.
A local optimum was chosen, and can be seen at **0.4**.



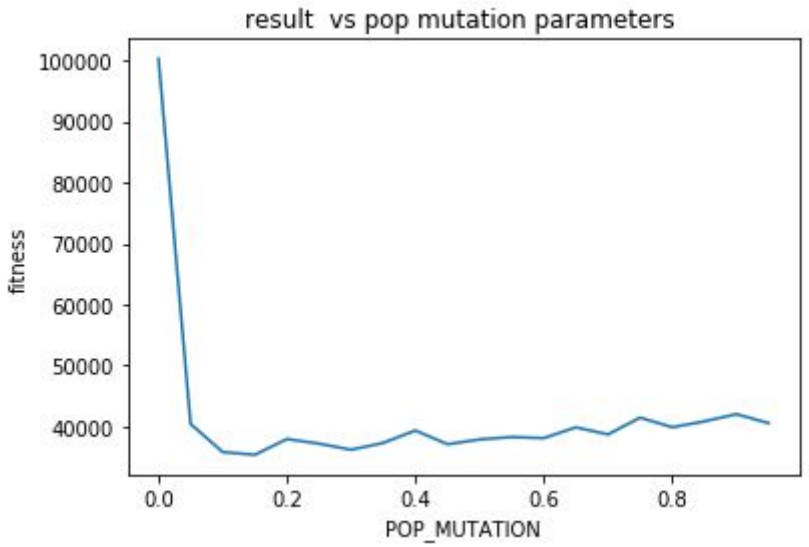
8.4 Crossover

In this experiment that goal was to observe the effect that the Crossover ratio in the population has on the result.
A local optimum was chosen, and can be seen at **0.4**.



8.5 Mutation

In this experiment that goal was to observe the effect that the mutation ratio in the population has on the result.
A local optimum was chosen, and can be seen at **0.15**.



8.6 Elite Insertion

The ratio of elite selection has no real effect on the result, as long as it's not 0.
A ratio of **0.1** was chosen for convenience reasons.

9 RESULTS

Considering the optimal route is **33,523**,
Our optimized approximation algorithm found a route of about **34,500**.
with the following parameters:

Population = 230
Generations = 905
POP_SELECT = 0.4
POP_CROSS = 0.4
POP_MUT = 0.15
POP_ELITE = 0.1

9.1 Runtetime

In the first experiment, the initial algorithm reached a result of approximately **36K** in about 8 minutes. After further parameter optimization, the improved algorithm has reached a result of **34.5K** in less than 1 minute. For comparison, the sequential algorithm took 6 minutes to complete.

10 CONCLUSION

In our project we tried to find a way to **approximate** the TSP Problem efficiently and with minimal **computation time,** all while using the parallel computing of Apache **Spark.**

Approximation:
Using our naive algorithm, we've approximated the optimal route with an error of **7%**.
Using our improved algorithm, we've approximated the optimal route with an error of **3%**, while improving the initial runtime by **8 times**.

Runtime:
Using Apache Spark, We were able to reduce the runtime by 4 times,
compared to the sequential algorithm.

11 REFERENCES

- **An Experimental Comparison Between Genetic Algorithm and Particle Swarm Optimization in Spark Performance Tuning**
  DOI: 10.1145/3129457.3129494

- **Solving NP hard Problems using Genetic Algorithm**
  ISSN: 0975-9646

- **Solving NP-Complete Problems Using Genetic Algorithms**
  DOI: 10.1109/UKSim.2016.65

- **2019 Evolutionary Algorithms Review**
  arXiv:1906.08870v1

- **A Comparative Study of Adaptive Crossover Operators for Genetic Algorithms to Resolve the Traveling Salesman Problem**
  International Journal of Computer Applications (0975 – 8887)

- **A Comparative Study of Crossover Operators for Genetic Algorithms to Solve the Job Shop Scheduling Problem**
  E-ISSN: 2224-2872

- **Comparison of eight evolutionary crossover operators for the vehicle routing problem**
  Math. Commun. 18(2013), 359{375}