Evolutionary Algorithm - Traveling Salesman Problem (Final Project)

By Afrah Abdulmajid, Gillian Mensah, Sylvia Unimna
Department of Computer Science
Memorial University

December 11, 2018

Contents

1 Introduction

1.1 Assigned Problem

The aim for the final project is to solve the Travelling Salesman Problem using Evolutionary Algorithms. Given a set of cities the task of the TSP is to find the shortest path to all cities, given that each city is visited once and we have to return back to the starting point. A Travelling Salesman Problem is a well known Optimization Problem in which we have a model of our problem and we seek the correct inputs that give us the right path. Given n number of cities, there are

$$\frac{(n+1)!}{2} \tag{1}$$

number of solutions and the goal is to find the most optimal solution among all.

For an EA the following are needed:

1. A population: Population is given by the diverse tours that can be taken by the Salesman, and a single tour is an individual.
2. Competition/Selection: The individuals compete with one another in order to be to be selected to produce an offspring/or to be a part of the next generation.
3. Variation/changes: A single individual will be selected at random to be mutated, this helps maintain a diverse population. How often variations take place are controlled by variation operators.
4. Reproduction/Death: For every generation, the next generation of individuals is created through recombination, and we choose what happens to the older generation, they either get to move on to the next generation or the new off-springs fully replace the older individuals.

The Main Components to any EA are given by the following:

Representation	Permutation of City Indexes
Fitness	Euclidean Distance
Recombination	Order Crossover, Modified Order Crossover and Greedy Crossover
Mutation	Swap Mutation & Inversion Mutation
Parent Selection	Tournament
Survivor Selection	$\mu + \lambda$ selection (without replacement)

What we would also like our EA to do, is go through all the phases in the fitness landscape. The phases are as follows:

1. Early Phase: This is when our population's fitness is at different points on the fitness landscape. This gives rise to exploration of the fitness landscape of our population.
2. Middle Phase: This is when our EA starts to evolve from exploration to exploitation.
3. Late Phase: This is the final phase of the EA, where our EA has discovered all the highest points on the landscape, and it is fully exploiting all the high values on the landscape.

1.2 Algorithm Design

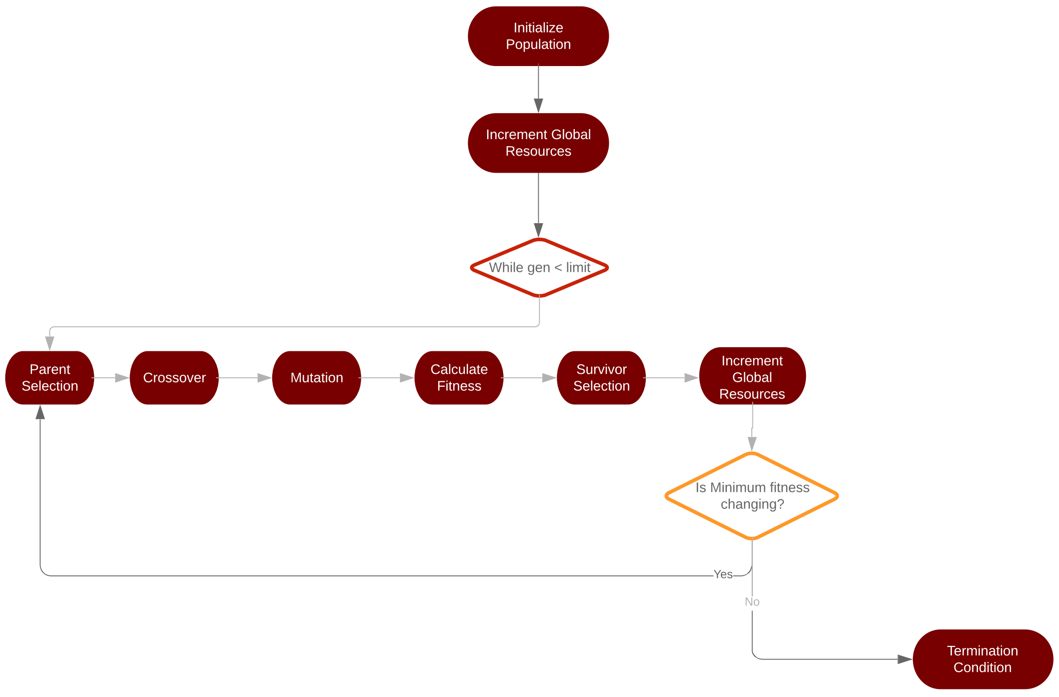


Figure 1: Top-Down Design of our EA

We followed a very similar design from our previous assignments in the class, we only made slight changes to fit the design of the TSP problem.
It starts by calling the function **city.py** to read the specified file and stores the coordinates in a numpy array of vectors, it then proceeds to pre-calculalate the distances of each coordinate and store them in a multi-dimensional list.
We then initialized our population by creating popSize amount of clusters. Through this we calculate the initial fitness our population and then it begins the normal workflow of an Evolutionary Algorithm which we explained further below.
We have a solid termination condition that only terminates when we haven't noticed a change in our fitness for **(0.25 * popSize)** iterations. We first switch our Mutation and Recombination operators and if we still don't see a change then we terminate the program, and print out a visualization of our optimal solutions and their corresponding fitness.

1.3 Parameter Settings

We listed some parameters below that we noticed made a huge difference in the efficiency and how optimal our code is.

Parameter	Purpose
pop_size	Max. number of individuals in a population
chrom_length	Number of cities in a file
gen_limit	Max. no of iterations
tournament_size	Number of individuals that compete in a tournament
mut_rate	Mutation rate
xover_rate	Crossover rate
change	Used to determine if our fitness has remained stagnant

Variation Parameters Used:

1. Mutation rate - The mutation rate would determine how often we choose to Explore or Exploit. With a mutation rate of 0.2, we would be exploring 20% of the time by making slight changes to our individuals even after the crossover and exploiting 80% of the time. Our preference in our problem with a mutation rate of
2. Crossover rate - This works in a very similar way to the mutation rate except it changes up an entire offspring and not just a segment, but it still carries the Exploration vs Exploitation concept.

2 User Guide

This program coded in python will try to solve the Travelling Salesman Problem by mimicking the workflow of an EA.

2.1 System Requirement

The Final Project folder contains the main EA.py file which can be edited with any IDE's (NetBeans, Visual Studio, Eclipse, Sublime Text etc.) or by using Notepad.
The program should be compatible with any operating system but it does require the user to at least have Python 3.7 or later since we are using some modules that the older versions do not have. Some knowledge of python is required.

2.2 Format

There is no input format required for this program. The user however has to edit sections of the code to select what city they would like to run.
The initial city we set was Western Sahara.

2.3 How to Run

In the main program we have imported all the modules needed, this makes running the program straight-forward. There are numerous ways to run a python program but we will be mainly focusing on the two popular ways, through the Console and through the IDLE.
Using the Console:
Step 1: Open Command Prompt
Step 2: Navigate to the directory where the main file is (.../FinalProject)
Step 3: Type in; python3 EA.py
The program will now execute

Using the IDLE: This method is alot more straight forward, we only need to open the file EA.py using the Python 3.7 IDE and click the F5 key to run it. This was also stated in the README attached.

3 Methods

3.1 Main Components of EA

3.1.1 Initialization

For the representation, initially we had decided to represent an individual as a 2D-array of 2 x N, where N is the number of cites, and every row in the array corresponds to the x-y co-ordinates of a city. However for ease of coding we decided to change our representation to a permutation of numbers from 1-N. Since we are representing our population for the EA as a permutation of numbers, where each number represents a city, we just generate a random permutation of numbers.

3.1.2 Fitness

In order to compute how good or bad a salesman's tour is given by the total distance travelled by the salesman.Therefore the shorter the distance travelled for an entire tour the better the tour is.
Since the fitness evaluation requires a lot of computation for tours with a lot of cities, to save the

amount of computation done, we pre-calculate all the possible distances for all the cities and store it in a 2-D matrix. Therefore when we are evaluating the fitness from one city to the next we just refer to the 2-D matrix of distances.
The distances between two cities is given by:

$$\text{Distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The Python v_3.7 library for Numpy made using arrays simpler, especially 2D arrays. This made calculating the distance vector more straight forward.
Once the matrix is computed fitness of an individual is given by the sum distances between a single tour(individual). Therefore an individual is evaluated with that sum.

3.1.3 Recombination

We had initially started with Greedy Crossover for our recombination method, but we noticed our algorithm reaching local optima extremely quick and combine the powers of three very popular crossover methods, Order Crossover, Modified Order Crossover and Greedy Crossover.

1. **Order Crossover** - Order crossover picks two crossover points randomly and copies those into the offspring. It then fills out the rest of the offspring by using cities(points) from the order parent.

2. **Modified Order Crossover** - Modified Crossover works in a very similar except it picks the crossover point by going through all points in the individual and picking two points with the shortest distance. [3] quick demo on how this works can be seen below:
   Parent 1: 1 2 5 6 4 3 8 7
   Parent 2: 1 4 2 3 6 5 7 8
   Detect shortest edge from second parent using the distance formula. The shortest edge would be **3, 5**. We pick our first crossover point as 3 and then proceed to choose a second one by picking any random point after 5, we then copy those into the parent just, and then follow the same steps from order crossover [4].

3. **Greedy Crossover** - starts by selecting a random city as the start and selects the next city based on the city closest to it. It does this by constructing an Edge Table and then using the distance formula to compute the edge with the shortest distance to it and then repeating the same steps for whichever city got picked. if the city with the shortest distance already exists in the offspring, it just picks a random city.

3.1.4 Mutation

Mutation is used to introduce completely different offspring from that of the parent in order to keep our algorithm from converging too soon, it also used to introduce some diverse individuals. We found that when we used a single mutation operator we sometimes would find our solution has entered a local optimum and it converges at that local optimum. However to get out of the local optimum we found that introducing a second mutation operator to replace the one being used helped us move up the landscape and reach our global maximum. The following mutation operators are used:

1. Swap Mutation
2. Inversion Mutation

3.1.5 Parent Selection

Parent selection is very crucial in determining the convergence rate of an EA, as good parents bring about good offspring.
We chose to use Tournament selection without replacement. We select K individuals to compete in the tournament and it picks the individual with the overall best fitness.

3.1.6 Survivor Selection

For our algorithm we decided to follow an elitist approach where the fittest individuals from the parents and offspring are reserved a seat in the next generation. We would like to maintain the better individuals given that our search space is very large in some cases. The elitist survivor operator that we decided to go with is $\mu + \lambda$ selection. In which the parents with the best fitness are always included in the new generation.

3.2 Advanced Technique

3.2.1 Modified Order Crossover

We decided to implement MOX as our advanced technique. We explained how this works above.

3.2.2 K-Means Clustering

K-Means transforms the problem from a 1 person problem to an n person problem. "we use K-means clustering to divide n cities into m clusters in accordance with the number of salesmen" [5]
We introduced Clustering when we ran our EA on Uruguay and noticed we were starting at a distance way bigger than what the optimal solution is, and getting the optimal path would cost us a lot of iterations thus increasing the running time of our algorithm. Using clustering reduced our Starting fitness by over half what it initially used to be.
Below is the algorithm we used to compute our clusters:

1. We begin by picking m number of points to set as our initial centroids(means)
2. For every city in the file, we compute the distance to each centroid and then we place the city in the cluster with the minimum distance.
3. We update the centroid by computing the new average whenever a new city is added into a cluster.

3.2.3 2-Opt

We decided to try and optimize our initial population by using 2-opt after we perform k-mean clustering. We found that the initial fitness after clustering gets reduced to half when 2-opt is used. Therefore the initial population we start with would not have a bad fitness to begin with.
Using 2-opt increased the run time of our algorithm and so we commented out the section of code, but we did attach it in the files submitted.
How 2-opt works: If we treat our individual in a population as an undirected graph we use 2-opt, also known as 2-optimal, to resolve any intersecting edges as long as the resulting new graph has a better fitness than the what it had previously. [2]

3.3 Runtime Optimization

We optimized by using vectors(Numpy array in python) to represent each x,y coordinate of a city. Through some research, we found out that vectors have a way lower time complexity than tuples or lists and this would decrease the runtime. The image below shows a comparison of Python's most common data structures and their total runtime.

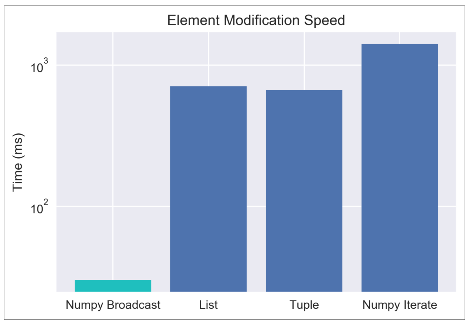


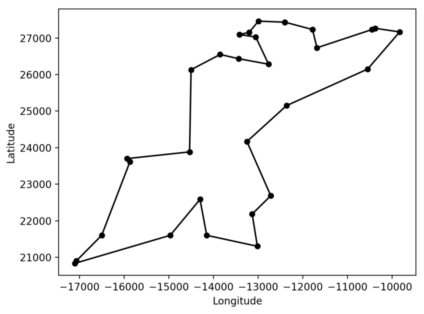
Figure 2: Comparing Python Data Structures [1]

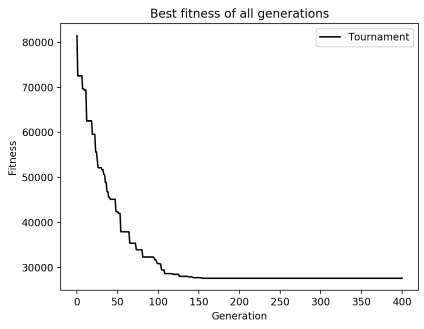
We originally started our algorithm with the hopes of using numpy arrays for all operations but we often found ourselves using lists for most of the components of an EA. We then decided to represent our population as list of city indexes but still store the coordinates as a vector array to increase the runtime of our fitness computation.

4 Results

4.1 Western Sahara - 29 Cities

As seen in the figures below , our optimal solution is the true optimal path. Our solution takes about 25 seconds to reach the optimal path, and below is a table we made to show our success rate





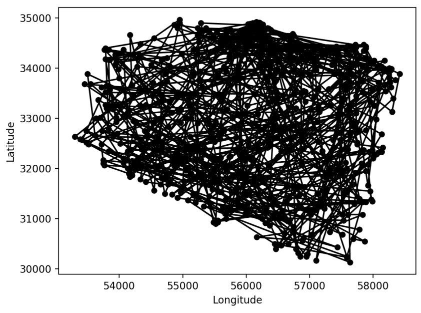
and the average number of evaluations we had to a solution. We also have a table below comparing the performance of Greedy Crossover and Order Crossover.

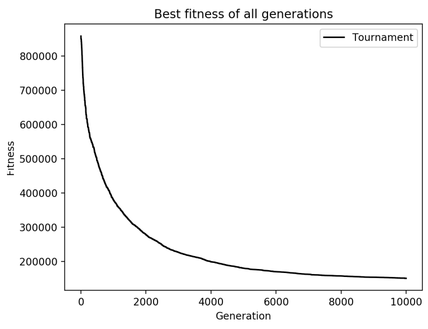
Run	Minimum Distance
1	28864
2	27601
3	27601
4	27601
5	28113
6	27601
7	27601
8	28864
9	27748
10	27601
11	27601
12	27601
13	27601
14	27748
15	27643
16	27601
17	27601
18	27748
19	27601
20	28972

The numbers highlighted in red show the Non-Optimal solutions we got during 20 runs. As seen, We have a success rate of **60%**.
Our program previously ran for 10 seconds but once we implemented all the extra features (i.e Clustering, 2-opt, switching mutation operators), we noticed it slowed down our algorithm and now we have a runtime of approximately 35 seconds.

4.2 Uruguay - 734 Cities

Initially when we ran our algorithm on Uruguay we found that it ran but it was very time consuming and our fitness was no where near optimal. (We initially started out with a fitness of 1.5 Million and ended up with a fitness of approximately 800,000) which is no where near the optimal. When we introduced Clustering we were able to trim down our initial fitness to about half what it used to be, but it still took 14 hours to get to a fitness of approximately 150,000, and this again is still not optimal. Attached is the result of a sample run and the progression of our fitness over 10,000 generations.





Through researching our code, we found that the most expensive part of our algorithm was the crossover because this used about 3 seconds per generation.

5 Discussions

Our proposed team management plan was to meet-up twice in one week, we managed to achieve a meeting once a week, and each meeting comprised of each member working on the tasks they were assigned.
We initially started out the project misunderstanding the approach to solve the problem, and this definitely contributed to our results; we started out thinking we didn't have to read all lines in the code and we didn't figure out the correct way till after all the demos were done. Given more time for the project (considering the time spent on implementing the wrong idea), we would have introduced Dictionaries into any part of our code that requires a look-up. Dictionaries have an access time of O(1) while lists have an access time of O(n) and we're sure this would have made a huge difference in the time spent on crossovers during our code.
We figured run time optimization was not on our side and so we made our main goal to be finding the optimal path .

5.1 Conclusion

Our algorithm has tried to solve the Travelling Salesman Problem. Although the TSP was solved, the optimal solution for Uruguay was not attained. We did however obtain the Optimal solution for western Sahara.

References

[1] Are tuples faster than lists? `http://zwmiller.com/blogs/python_data_structure_speed.html`, 2017.

[2] Maria Antonia Carravilla Jose Fernando Oliveira. Heuristics and local search. `https://paginas.fe.up.pt/~mac/ensino/docs/OR/CombinatorialOptimizationHeuristicsLocalSearch.pdf`, 2001.

[3] Jean-Yves Potvin. Genetic algorithms for the traveling salesman problem. `https://iccl.inf.tu-dresden.de/w/images/b/b7/GA_for_TSP.pdf`, 1996.

[4] Monica Sehrawat and Sukhvir Singh. Modified order crossover(ox) operator. *International Journal on Computer Science and Engineering*, 3(5):2019–2023, 2011.

[5] Lutfiani Safitri Sri Mardiyati and Jihan. Solving multiple traveling salesman problem using k-means clustering-genetic ant colony system algorithm.