FC3: Genetic Algorithm in Python

Heriberto Espino Montelongo

Universidad de las Américas Puebla

P24-LIS3082-2: Inteligencia Artificial

Dra. Ingrid Kirsching

April 2024

Table of Contents

Introduction

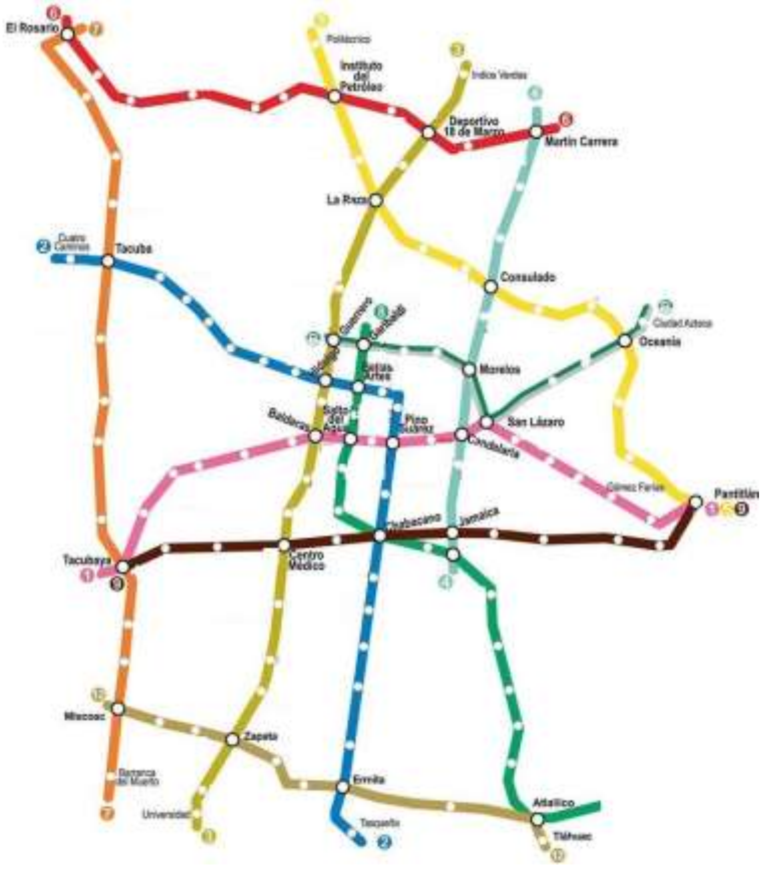
The Traveling Salesman Problem is about finding the shortest possible route for a salesman to visit each city exactly once and return to the starting city. The problem has practical implications in logistics, route planning, and organization of circuits in electronic design (DataCamp, 2023).

More formally, TSP can be looked upon as an undirected graph with cities being considered as nodes and edges attached with weights, which usually represent travel costs or distances. The given approach aims at minimizing the weight of the overall tour, while the traveler passes through all the nodes, returning to the node of origin. In another way, the problem belongs to the NP-hard class—the class for which there actually doesn't exist any efficient algorithm solving all of its instances within polynomial time (Reinelt, 1994).

In this case, we do not want to visit all cities of a state, we only want to go from a starting point to an end point in the shortest time possible, and, instead of cities, we will use *estaciones de metro* from Mexico City.

Figure 1

Map of *rutas* from Mexico City



Map of rutas for the problem

For the problem to solve, The starting point is ‘El Rosario’ and we want to get to ‘San Lázaro’, still, in the shortest time possible using a genetic algorithm.

List of *estaciones*

The estaciones are:

- Atlalilco
- Balderas
- Barranca del Muerto
- Bellas Artes
- Candelaria
- Centro Médico
- Chabacano
- Ciudad Azteca
- Consulado
- Cuatro Caminos
- Deportivo 18 de Marzo
- El Rosario
- Ermita
- Garibaldi
- Gómez Farías
- Guerrero
- Hidalgo
- Indios Verdes
- Instituto del Petroleo
- Jamaica
- La Raza
- Martín Carrera
- Mixcoac
- Morelos
- Oceanía
- Pantitlán
- Pino Suarez
- Politécnico
- Salto del Agua
- San Lázaro
- Tacuba
- Tacubaya
- Tasqueña
- Tláhuac
- Universidad
- Zapata

List of *Rutas*

Línea 1:

From Tacubaya to Balderas: 6

From Balderas to Salto del Agua: 1

From Salto Del Agua to Pino Suarez: 2

From Pino Suarez to Candelaria: 2

From Candelaria to San Lazaro: 1

From San Lazaro to Gomez Farias: 4

From Gomez Farias to Pantitlan: 2

Línea 2:

From Cuatro Caminos to Tacuba: 1

From Tacuba to Hidalgo: 7

From Hidalgo to Bellas Artes: 1

From Bellas Artes to Pino Suárez: 3

From Pino Suárez to Chabacano: 2

From Chabacano to Ermita: 6

From Ermita to Tasqueña: 1

Línea 3:

Indios Verdes to Deportivo 18 de Marzo: 1

From Deportivo 18 de Marzo to La Raza: 2

From La Raza to Guerrero: 2

From Guerrero to Hidalgo: 1

From Hidalgo to Balderas: 2

From Balderas to Centro Médico: 3

From Centro Médico to Zapata: 4

From Zapata to Universidad: 2

Línea 4:

From Martín Carrera to Consulado: 3

From Consulado to Morelos: 2

From Morelos to Candelaria: 1

From Candelaria to Jamaica: 2

Línea 5:

From Politécnico to Instituto del Petróleo: 1

From Instituto del Petróleo to La Raza: 2

From La Raza to Consulado: 3

From Consulado to Oceanía: 3

From Oceanía to Pantitlán: 3

Línea 6:

From El Rosario to Instituto del Petróleo: 6

From Instituto del Petróleo to Deportivo 18 de Marzo: 2

From Deportivo 18 de Marzo to Martín Carrera: 2

Línea 7:

From El Rosario to Tacuba: 4

From Tacuba to Tacubaya: 5

From Tacubaya to Mixcoac: 3

From Mixcoac to Barranca del Muerto: 1

Línea 8:

From Garibaldi to Bellas Artes: 1

From Bellas Artes to Salto del Agua: 2

From Salto del Agua to Chabacano: 3

From Chabacano to Atlalilco: 8

Línea 9:

From Tacubaya to Centro Médico: 3

From Centro Médico to Chabacano: 2

From Chabacano to Jamaica: 1

From Jamaica to Pantitlán: 5

Línea 12:

From Mixcoac to Zapata: 3

From Zapata to Ermita: 3

From Ermita to Atlalilco: 2

From Atlalilco to Tláhuac: 1

Línea B:

From Guerrero to Garibaldi: 1

From Garibaldi to Morelos: 3

From Morelos to San Lázaro: 1

From San Lázaro to Oceanía: 3

From Oceanía to Ciudad Azteca: 1

Description of representation

The rutas.py file stores this information, it has Python classes and data structures, with the data and behavior needed to manage the metro routes. Here are some points describing its content and its functions:

Class *Ruta*: This class represents a route between two *estaciones*. It has attributes for the starting station (start), ending station (end), travel time (time), and the *linea* it belongs to.

List *estaciones*: This list contains the names of all *estaciones*. Each *estacion* is identified by its index in the list. This provides a convenient way to reference stations when creating routes.

List *rutas*: This list contains Instances of the *Ruta* class, representing different rutas between *estaciones*. Each *ruta* is defined by specifying the starting and ending stations, travel time, and the line it belongs to. The list is twice the size of the total of *rutas*, it represents the same *ruta* in the opposite direction, for the orientation of a start point and an end point.

`generate_genome`: Builds a genome, which is basically a path from a from a starting point to an endpoint, using a list of *rutas*. The genome consists of indices from this list, representing selected *rutas*, each index specifies the *ruta* total travel time; the time; the *linea* involved, and the *estaciones* of the path.

`generate_population`: Takes two parameters: *rutas* and size_pop, where size_pop is the desired size of the first population. The function returns a list of genomes.

`fitness`: Calculates the fitness of each genome, by dividing one over the total time of the path, in order to later choose the genomes with a higher fitness value.

`elitism`: Selects the top genomes from the population based on their fitness. The number of genomes to select is determined by the elite_size parameter. If elite_size is less than 2, it is set to 2 to ensure that at least two genomes are selected. The genomes are sorted in descending order of fitness, and the top genomes are returned.

`mutate_genome`: This is used to introduce variation into the population of genomes by altering parts of the path. It randomly selects a point in the genome's path, generates a new path from that point to the end point, and replaces the part of the original path from the chosen point with the newly created path.

`crossover`: Combine two parent genomes to produce a child genome. It does this by finding a common stage between the parents and creating a new path that includes the part of the first parent's

path up to the common stage and the part of the second parent's path after the common stage.

`create_new_population`: Creates a new population for the next generation of the genetic algorithm. The algorithm here selects the top genomes from the current population (elitism), generating new genomes by combining pairs of elite genomes (crossover), creates new genomes through pairing elite genomes, making sure it introduces variation by randomly altering some parts of these elite genomes (mutation). The new population consists of the elite genomes, their copies, and the children genomes.

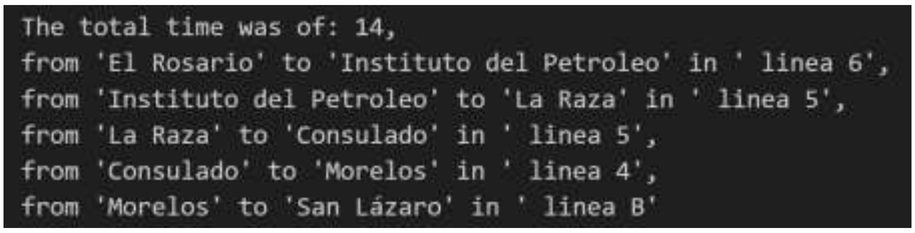
`genetic_algorithm`: Manages the execution of the genetic algorithm through different generations.

Result of the execution of the program

Every run is considered successful if the time is equal to 14 and the output is the one of the figure below.

Figure 2

Successful result of the execution of the program



```
The total time was of: 14,
from 'El Rosario' to 'Instituto del Petroleo' in ' linea 6',
from 'Instituto del Petroleo' to 'La Raza' in ' linea 5',
from 'La Raza' to 'Consulado' in ' linea 5',
from 'Consulado' to 'Morelos' in ' linea 4',
from 'Morelos' to 'San Lázaro' in ' linea B'
```

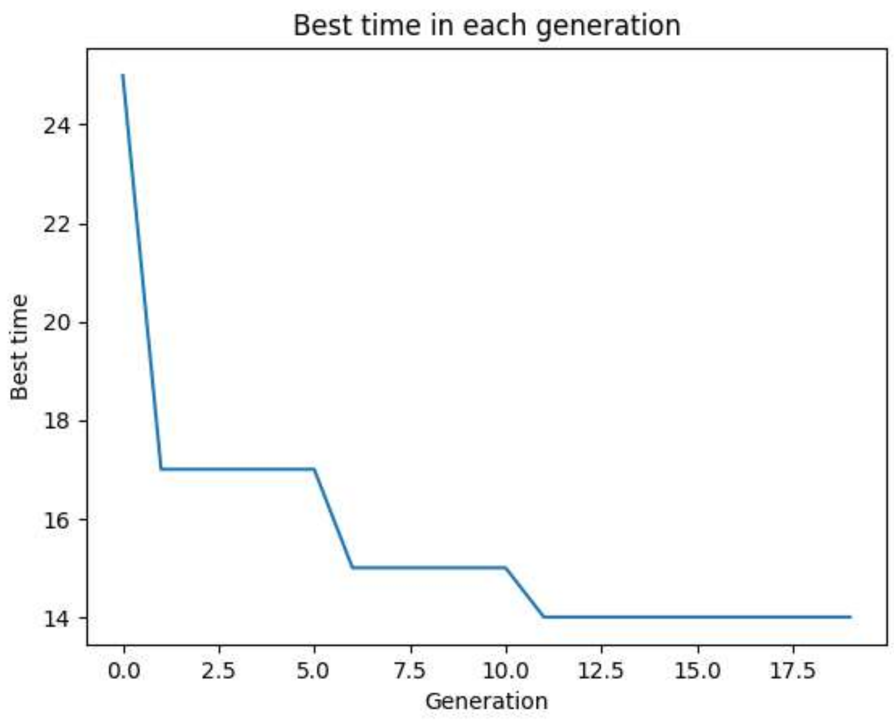
The shortest path has a time of 14, with those estaciones and lineas.

Result of the execution of the program given the parameters

Results differ given the parameters, in this section we find a graph that shows the best time of each generation, considering a run successful when the time 14. Some of the results were the following:

Figure 3

Best time in each generation.



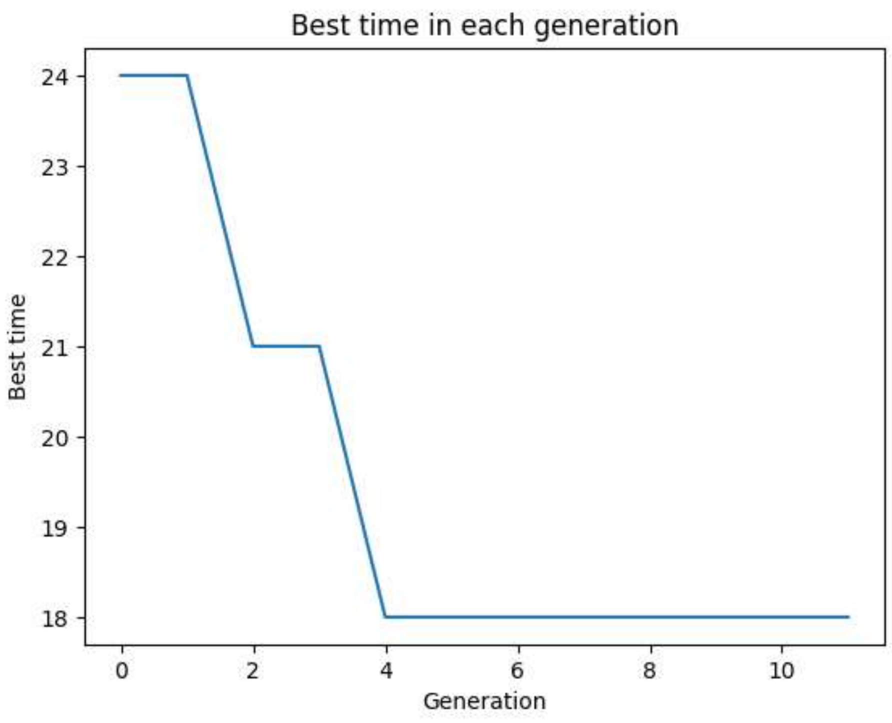
Successful run with number of generations equals to 20.

```
start_point = 'El Rosario'
end_point = 'San Lázaro'
size_pop = 20
num_generations = 20
elite_size = 6
mutation_rate = 1
time = 0.4s
```

Almost always finds the shortest path, around a 70% of the times, however, it last 0.4s, we will see what happens if we want a lower time.

Figure 4

Best time in each generation, but faster.



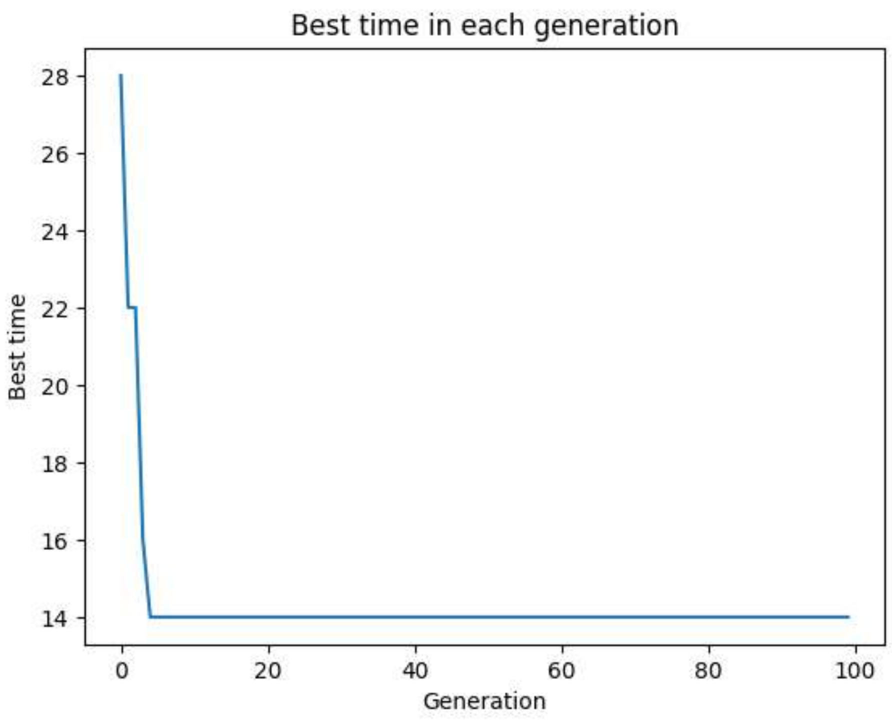
Successful run with number of generations equals to 12.

```
start_point = 'El Rosario'

end_point = 'San Lázaro'

size_pop = 12

num_generations = 12

elite_size = 4

mutation_rate = 1

time = 0.1s
```

With this parameter, it finds the shortest path around 50% of times, which is not ideal, but it is fast, with a time of 0.1s. We can see that it is ideal to increase the number of generations to ensure with higher probability that the shortest path was found.

Figure 5

Best time in each generation, but with 100 generations.



Successful run with number of generations equals to 100.

```
start_point = 'El Rosario'

end_point = 'San Lázaro'

size_pop = 12

num_generations = 100

elite_size = 4

mutation_rate = 1

time = 0.6s – 2s
```

The algorithm almost always finds the shortest path before generation 20, however, the time does not increase significantly unless the genomes take a long time to arrive to the end point. Therefore, I will consider these parameters as the better ones.

Conclusions

Genetic algorithms are effective for problems with a high complexity, being able to even find the optimal solution, or near the optimal, reducing the time of execution by large amounts. It is curious, as well as amazing , how GA imitate the evolutionary process, being influenced by parameters, such as population size, crossover, mutation rates, elitism and number of generations. Achieving optimal performance often requires careful calibration of these parameters.

Moreover, the implementation of a genetic algorithm has not only provided us with solutions but also deepened our understanding of the problem. By tracing the evolution of routes over successive generations. As a result, GA serve not only to optimize solutions but also to enhance our understanding of network structures.

References

DataCamp. (2023, April). *Travelling Salesman Problem using Dynamic Programming.* Travelling Salesman Problem using Dynamic Programming - GeeksforGeeks

Kie Codes. (2020, July 20). *Genetic Algorithm from Scratch in Python (tutorial with code)*. YouTube. Retrieved from [https://www.youtube.com/watch?v=nhT56blfRpE&t=312s]

OpenAI. (2024). ChatGPT [Large language model]. https://chat.openai.com/chat

Reinelt, G. (1994). *The traveling salesman: computational solutions for TSP applications.* https://archive.org/details/travelingsalesma0000rein/page/1/mode/1up

The Coding Train. (2016, July 29). *Genetic Algorithm*. YouTube. Retrieved from [https://www.youtube.com/watch?v=9zfeTw-uFCw&list=PLRqwX-V7Uu6bJM3VgzjNV5YxVxUwzALHV]

GitHub Repository

Personal repository: https://github.com/heritaco

Complete code: https://github.com/heritaco/Inteligencia-Artificial/tree/main/5%20Genetic%20Algorithm/FC3%20Genetic%20Algorithm

Jupyter Notebook: https://github.com/heritaco/Inteligencia-Artificial/blob/main/5%20Genetic%20Algorithm/FC3%20Genetic%20Algorithm/genetic%20algorithm.ipynb

During the development of the algorithm, I drew inspiration from Kie Codes work (2020) and consulted a series of instructional videos by The Coding Train (2016). Additionally, I received guidance and assistance from ChatGPT during the process.

Throughout the process, I got assistance from ChatGPT to enhance the code's clarity, resulting, for example. in the removal of unnecessary elements.