Fabián Ontiveros Juárez 173428
Artificial Intelligence

Genetic Algorithms

Introduction

The Travelling Salesman Problem is a classic problem in computer science and operations research, seeking the shortest route that visits a list of cities exactly once and returns to the starting point. It's a challenging problem, classified as NP-hard, with applications in various fields like logistics, manufacturing, and DNA sequencing. Despite its computational complexity, numerous algorithms exist to tackle it efficiently, making it a benchmark for optimization methods. The problem's variations, like the travelling purchaser problem and vehicle routing problem, further extend its relevance in practical scenarios (Wikipedia).

Problem

Ciudad de México is one the world's largest cities with about 9 million inhabitants according to the 2020 population and housing national census done by INEGI (INEGI,2020). A very important element in the city's transport is the metro system, which was created about 50 years ago.

Given a map of the metro stations from Ciudad de México, we want to determine the path of minimum length between "El Rosario" and "San Lázaro" using a genetic algorithm.

Figure 1. CDMX metro system.

Graph representation

I solved this problem by using an adjacency matrix representation for a non-directed weighted graph $G = (V, E)$ such that each node represents one of the stations seen in the figure above that are drawn with a white circle and a black contour.

In Power Point, I drew black circles over each of the stations and eliminated the map image, then I exported the slide as a .jpg and cut the edges that did not formed part of the original

map shape. The idea is to extract the positions of the circles with a clustering algorithm in order to visualize the graph nodes and edges in Python, by using matplotlib.

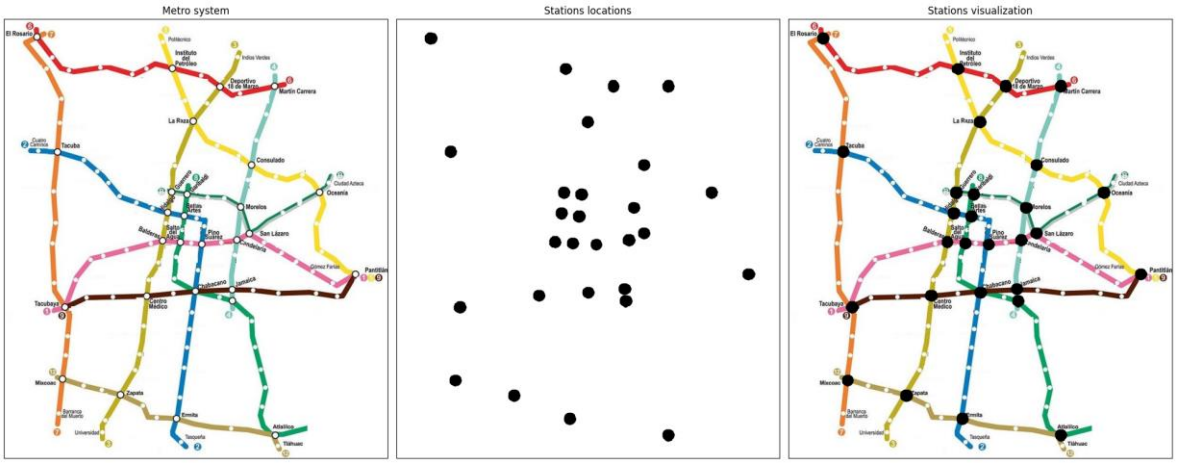


Figure 2. Metro system, black circles, merged images.

Now, by including the location information (x,y coordinates) to the matrix that represents the black circles image and applying KMeans clustering to all pixels whose color is not white I obtained the centers of the 28 clusters (one for each station) and manually built the adjacency matrix A with the following indexing.

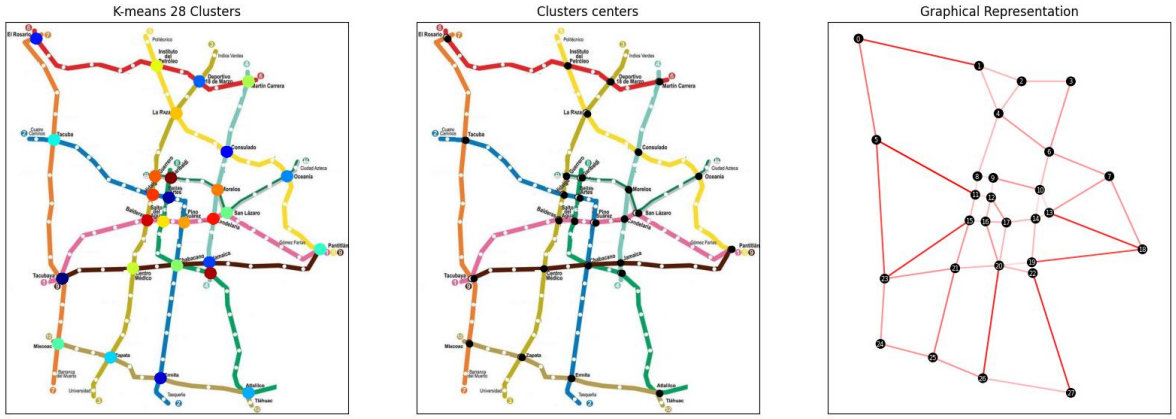


Figure 3. Graph visualization.

The genetic Algorithm

Let $v_1, v_2 \in V$ such that P: $v_1 = v^1 \rightarrow \cdots \rightarrow v^n = v_2$is a simple path between $v_1$and v_2. We define the distance of the path P as follows:

$$d(P) = \sum_{j=1}^{n} w\left(v^j, v^{j+1}\right)$$

where $w\left(v^j, v^{j+1}\right)$ represnts the weight from the edge $\{v^j, v^{j+1}\} \in E$.

With this, we define the **fitness function** f of P as

$$f(P) = \frac{1}{d(P)}$$

In our context, any simple path P from "El Rosario" (v_0) to "San Lázaro" (v_{13}) will be considered a **genome**. Now, we define a procedure to generate a simple random walk from "El Rosario" to "San Lázaro" (a simple walk is a walk that do not repeats vertices), see the code for the implementation of this procedure.

With this, we now may define some other functions as part of the genetic approach:

1. A generator of a random sample of genomes
2. A selection procedure (we will use the ranking method)
3. A breeding procedure
4. A mutation procedure

The hardest part in this activity is to ensure that modifications of the current genomes leave a new **valid** genome.

> ***Note.*** *See the code for such procedures, in this report I only comment some important factors taken in account to solve this problem.*

Let $P_1: a = v^1 \to \cdots \to v^n = b$ and $P_2: a = u^1 \to \cdots \to u^m = b$ be two simple paths from a to b. To breed P_1 and P_2 we will procede as follows

Case 1

Suppose that $P_1 \cap P_2 \neq \emptyset$ then, $\forall\, \mu \in P_1 \cap P_2, \exists\, v^i \in P_1, u^j \in P_2$ such that $\mu = v^i = u^j$.

If $v^1 \to \cdots \to v^{i-1} \cap u^{j+1} \to \cdots \to u^m \neq \emptyset$ we **breed** P_1 and P_2 to create a new simple path P given by

$$P(\mu): v^1 \to \cdots \to v^{i-1} \to \mu \to v^{j+1} \to \cdots \to u^m$$

which is a valid genome because none of the vertices at the right appears at the left and it starts and ends at the same places.

If the former is not true, we skip such μ and pass to the next one (if exists).

Suppose that this process is valid for some $\mu_1, \dots \mu_k \in P_1 \cap P_2$, so we form childs $P(\mu_1), \dots, P(\mu_k)$. Now, in the sake of omptimality, we return the path $P(\mu_s)$ that maximices the fitness function value.

Case 2

Suppose that $P_1 \cap P_2 = \emptyset$ or Case 1 is false for all $\mu \in P_1 \cap P_2$ then we say that P_1 and P_2 are **not breedable** and simply return one of both at random.

Now, we implement a mutation process as follows.

Let $P: a = v^1 \to \cdots \to v^n = b$ be a simple path from a to b. Now, with probability p select some random vertex $v^i \in P - \{a, b\}$ and define $P^{(i)} = v^1 \to \cdots \to v^{i-1} \subset P$ as the i-th cut of P. Now, we generate a path from v^i to v^n with a visited nodes list $P^{(i)}$ and return the result.

Note that there exists at least one path that connects v^i to b without passing by the former nodes, which is $v^i \to \cdots \to v^n$ itself. Hence, this process will always return a valid genome.

In the Python notebook, I implemented these processes and the extension of these to apply them to all the current population (See the code for more information about this).

Implementation

I represented each path with a list of the node's indices of figure 3. For example, the path

$$P: v_0 \to v_1 \to v_4 \to v_6 \to v_{10} \to v_{13}$$

which is actually, the optimum one, is represented as $[0,1,4,6,10,13]$.

The adjacency matrix A was implemented with a **NumPy** array and all the processes mentioned before (path generating, breeding and mutation) used NumPy-based operations combined with **Itertools** used for efficient looping and **Random** for random number generation during the mutation process and the path generation. I created all the figures and animations with **Matplotlib** and joined them with **Imageio**.

Execution and results

The code is presumably fast and even can find the global optimum in a small number of iterations. Again, I recommend seeing the jupyter notebook to get a deeper explanation about this and the parameters used. I strongly recommend running the **Execution** section cells multiple times to get different results.

As an example, lets study the following execution of the code.

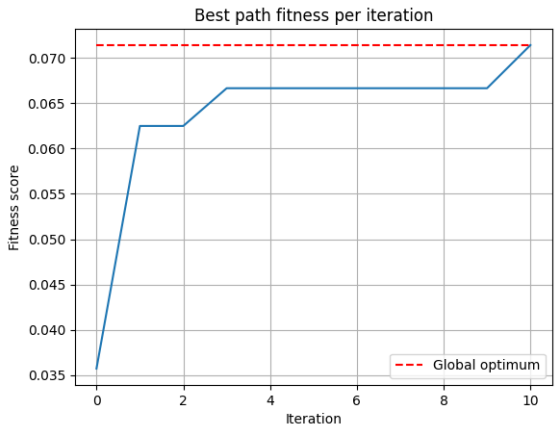


Figure 4. Best- model performance per iteration.

Note that in this case, the system reached the global optimum (path 0,1,4,6,10,13) in just 10 iterations, this shows the power of genetic algorithms.

Note: Iteration 0 is just non-ranked a random walk. Hence, it may be anything.

In the following figure, see how the system evolves and gets a solution closer to the optimum one as iterations increases.

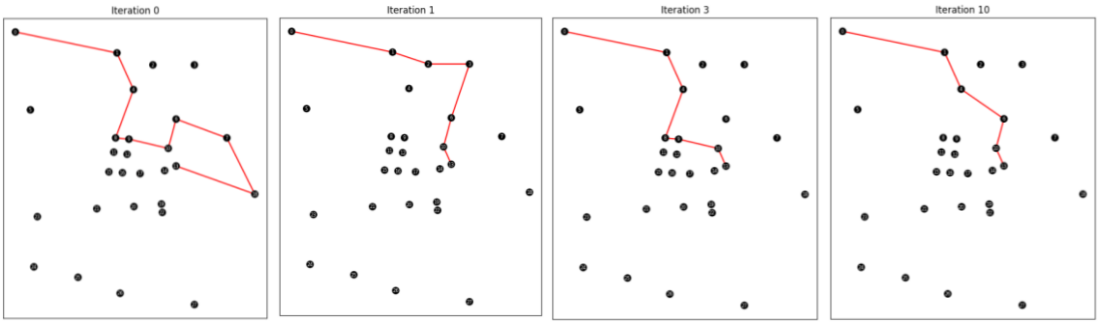
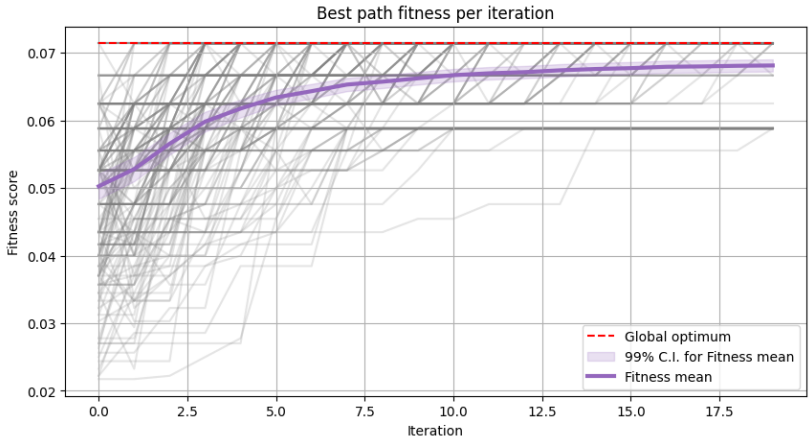


Figure 5. Best different genome per iteration.

Now, to study the model's general performance, I generated 250 populations and run the genetic algorithm for 20 iterations. With the data recovered we form the following figure, where I show the mean of the i-th iteration and a 99% Confidence Interval for the mean.



See that the mean fitness value after the 6-th iteration is closer than or as good as 90.0% from the value achieved by the global optimum. Hence, in just 7 iterations, the genetic algorithm may find a very good solution in average.

Conclusions

Genetic algorithms are a great alternative to other discrete-optimization algorithms that may need to compute compute many or even all possible combinations to achieve a good solution. In our case, we studied a small graph route problem, that may have also been solved efficiently by Dijkstra's or A* algorithms, for example. The problem is that if the graph size increases dramatically, such algorithms may become too expensive to consider them when implementing a feasible solution in computation power and time terms.

References

INEGI (2020). Ciudad de México: Población y número de habitantes. https://cuentame.inegi.org.mx/monografias/informacion/df/poblacion/

Kirschning, I. (2024). Genetic Algorithms. [PowerPoint slides]. School of Engineering, UDLAP.

Soltz, E. (2018). Evolution of a salesman: A complete genetic algorithm tutorial for Python. https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35

Wikipedia (n.d.). Travelling salesman problem. https://en.wikipedia.org/wiki/Travelling_salesman_problem#:~:text=The%20travelling%20salesman%20problem%2C%20also,returns%20to%20the%20origin%20city%3F%22