Artificial intelligence

Project 3: Genetic Algorithm Traveling Salesman

Alexander Beauregard Bravo

169206

Imagine a traveler, let's say Sam, that as a part of their journey they want to travel to different cities in Europe. Now, Sam is a very organized and efficient person, but also incredibly adventurous. This means that of course, they want to know the best possible route that they have to take, that is to say, the route that costs the least amount of money that manages to travel through every city in their destination list once. Let us consider the case where they only want to travel through a total of three cities.

As we can see in Figure 1, a representation of the cities is made using a mathematical structure called a graph. The interpretation is obvious: each city is represented as a vertex (so the names of the cities would be #1, #2, and #3) and each edge represents a connection between those respective cities, along with the cost of traveling. It is possible to see that the best strategy, in this case, would be to start at city #2, then move to city #1 and finally to city #3 (or start at #3, then #1, then #2), with a total cost of 18 units.

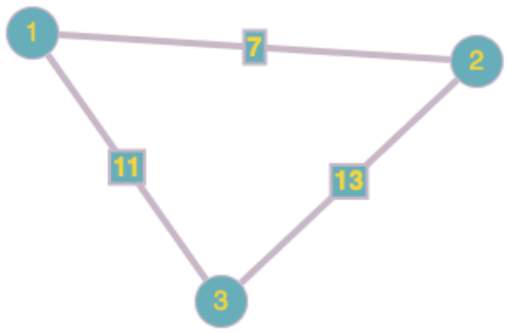


Figure 1: Graph representing three cities with their cost

This problem is called the Traveling Salesman problem. More succinctly, this problem can be summarized in the following way: "Given a list of cities and a cost of traveling between each pair of cities, what is the 'shortest' possible route that visits each city exactly once and returns to the origin city?". This problem was first formulated in 1930, and it is one of the most intensively studied problems in optimization.

Let us return to Sam. Since the amount of cities that they want to travel through is very small, it is very easy to just see what the optimal solution would be. However, what happens if Sam decides to, let's say, travel to 9 different cities?

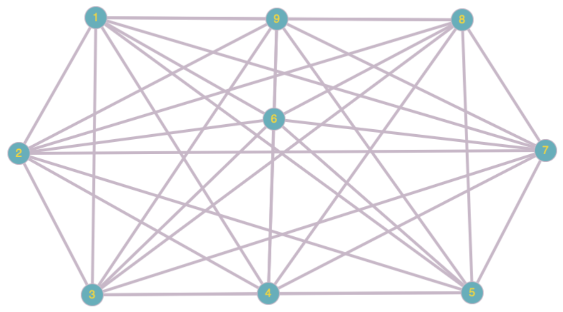


Figure 2: Representation of 9 cities

The situation gets a lot more complicated, as shown in Figure 2. Since it is no longer possible to just see what the optimal solution is, we turn to computation. However, how can a computer get to the right answer? Well, one easy way would be to simply use a brute-force algorithm. That is, an algorithm that actually “travels” through each possible path and each time a path is completed, it records the total cost and compares it to the one in the next iteration and it keeps the minimum one. This would certainly work, yet what happens when we there is a big amount of vertices that the computer has to travel through?

Number of cities	**Number of distinct tour paths**
4	3
5	12
6	60
…	…
9	20160
…	..
12	19958400

Figure 3: Number of possible paths as the number of cities increases.

From Figure 3, we observe that just by considering 12 cities, the computer would have to check nearly 20 million possible paths! Clearly, this only gets worse as the number of cities increases. It seems then, that we don’t really have a good and time-efficient algorithm to solve this problem.

Luckily, this is not true and we have several ways in which we can solve this problem, mainly by using heuristics or some other clever way to circumvent the problem of the number of cities. The particular way in which the problem will be approached will be by considering evolutionary computing, mainly genetic algorithms (GAs). GAs work by leveraging one of the most powerful forces in the world: evolution.

Consider natural selection. The process of it starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If the parents have better fitness, their offspring will be better than the parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. This notion is applied to TSP, that is to say, for a search problem. Five phases are considered in a genetic algorithm:

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

The problem that was attempted can be seen in Figure 4. There are several differences between this connection of cities and the standard connection of cities in the original TSP, for example, this graph is not complete! That is to say, you can not move freely from one city to another (i.e. London and Rome). This represents a problem, especially when the crossover happens. Also, we have two constraints in this problem. One of them is to minimize the total cost of traveling and the other is to waste no more than 72 hours. Details will be explained shortly.

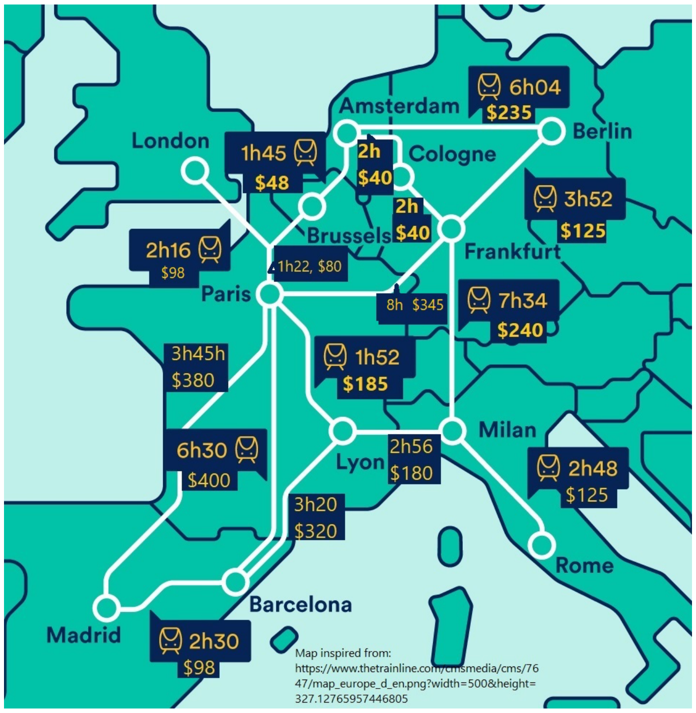


Figure 4: Problem attempted for this project

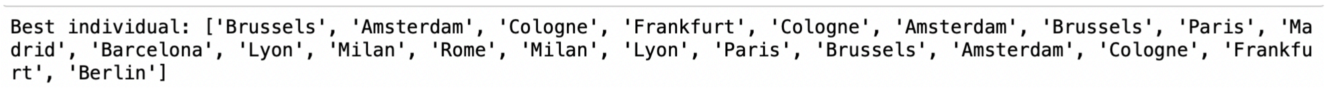
First, a way of making the computer read this connection of cities is required. There are several ways to do this, for example, just a simple graph object in Python (using the *networkx* library, for example), using adjacency matrices, etc. In this case, a simple dictionary was used, with each key being a city which contained several other keys, representing the edges and with tuples as values, representing the cost in the first index of the tuple and the time in the second.

After this, we declared several functions. These were created with readability and order in mind, so while some functions could be merged together and eliminate the necessity of two different ones, simplicity was the main goal here. In detail:

1. *dijkstra(graph, start, end):* Implementation of Dijkstra's algorithm for finding the shortest path between two vertices. This was done in order to calculate the minimum distance from two cities that are not connected with each other.
2. *remove_adjacent(nums):* Helper function for *make_viable(graph, psol)*.
3. *make_viable(graph, psol):* Transforms a possible solution into an actual viable one. This is needed, since when generating a random population, we must make sure that each adjacent cities in the path are actually connected, and we must make sure this keeps being the case when crossover happens.
4. *create_individual():* Initializes the population of potential solutions.
5. *calculate_fitness(individual):* Calculates fitness of each path in our population by taking the inverse of the total distance, making this problem into one that maximizes fitness. It also assigns a 0 to a solution in the population if it exceeds the 72 hour limit.

6. *selection(population):* Uses tournament selection to choose the parents for the next generation.
7. *crossover(parent1, parent2):* Creates the offspring by swapping the tails of the parents.
8. *mutation(individual, mutation_rate):* Swaps two cities with a given probability.
9. *replacement(population, offspring):* Replaces the least fit individuals with the offspring.

Finally, the algorithm ran for 5 generations with a population size of 1000. The results can be seen in Figure 5.



```
Best individual: ['Brussels', 'Amsterdam', 'Cologne', 'Frankfurt', 'Cologne', 'Amsterdam', 'Brussels', 'Paris', 'Ma
drid', 'Barcelona', 'Lyon', 'Milan', 'Rome', 'Milan', 'Lyon', 'Paris', 'Brussels', 'Amsterdam', 'Cologne', 'Frankfu
rt', 'Berlin']
```

Figure 5: Solution found

It must be said, that while this is a very powerful method of solving TSP, the constraints in this specific scenario makes it very difficult for it to converge to an actual solution. This is mainly because the graph is not a complete. This means that when a crossover or a mutation happens, since not all cities are connected with each other, then there is a big probability that the new offspring will contain two adjacent cities that are not connected with each other and thus, not be a viable solution. This is fixed by the *make_viable* function described above. However, even if all adjacent cities are connected to each other, this now does not guarantee that the solution actually passed through all cities. For this reason, and several others, this algorithm does not always converge, and as a matter of fact it has a hard time doing so.

This project helped me understand deeply how genetic algorithms actually worked. They are a very powerful tool in several applications of machine learning and AI. Several problems were found and while the code is far from perfect, it was a good attempt at solving this NP-hard problem.

References

“Combinatorics - Counting the Number of Paths in the “Travelling Salesman Problem.”” *Mathematics Stack Exchange*, 2022, math.stackexchange.com/questions/4373325/counting-the-number-of-paths-in-the-travelling-salesman-problem. Accessed 26 Apr. 2023.

Mallawaarachchi, Vijini. “Introduction to Genetic Algorithms — Including Example Code.” *Medium*, 1 Mar. 2020, towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3#:~:text=A%20genetic%20algorithm%20is%20a. Accessed 25 Apr. 2023.

“Travelling Sales Person Problem - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/travelling-sales-person-problem. Accessed 25 Apr. 2023.

Wikipedia Contributors. “Travelling Salesman Problem.” *Wikipedia*, Wikimedia Foundation, 14 July 2019, en.wikipedia.org/wiki/Travelling_salesman_problem. Accessed 25 Apr. 2023.