Universidad de las Américas Puebla

Artificial Intelligence

FC 3: Genetic Algorithm in Python
Traveling Salesman Problem:
Mexico City Subway

Sofía Graham Coello

174291

April 22°, 2024.

Introduction

In this project we try to use a genetic algorithm to find the shortest path between two stations on the Mexico City subway based on the famous Traveling Salesman Problem (TSP). According to Adewole and Akinwale (2011), this problem is a classic optimization problem in the field of computer science and operations research. It involves finding the shortest possible route that a traveling salesman can take to visit each city in its rute once and return to the original city but in contrast of the simple description, this problem is one of the nondeterministic polynomial-time complete (NP- complete) problems.

On the other hand, a genetic algorithm is a search heuristic inspired by the process of natural selection from evolutionary biology. It mimics the mechanisms of genetics and natural selection; this algorithm starts with a randomly generated set of solutions called a population. Each solution, or individual in the population is evaluated using a fitness function that quantifies how good the solution is in solving the problem. Based on their fitness, individuals are selected to reproduce and create a new generation of solutions so over successive generations, the population evolves towards an optimal solution.

Based on the complexity of the TSP we approach its solution with a genetic algorithm as they are well-suited for searching through large and complex spaces for good solutions, and due to the versatility, the TSP can present (different spaces), the complexity varies and the use of this kind of algorithm allows finding the solution in acceptable time.

Genetic Algorithm Elements

To work the algorithm, we first need to stablish the base elements to imitate the biological process of evolution, consisting of the chromosomes, the crossover, mutation, and the fitness of each chromosome.

The basic start was making a non-directed graph, were the nodes work as the stations and the weight are the distances between in one, this is later represented as an adjacency matrix, where two unconnected stations are marked wight an infinite weight.

As we are working with traveling routes, stablishing the base of all this algorithm is making chromosomes that make sense and enable all the other processes to work properly, initially there were a few ideas on how to represent the chromosomes, all around the idea of an stablished size and just randomizing the in-between, but it limited the solutions in making them unnecessarily long and didn’t arrive to a short route at all. In the end the best way to represent was suggested by Villareal (2024)

making a chromosome a random length array of random numbers, were each number is a node in the graph, and lock in the first and last number in the array, the start, and the finish. This way we could have chromosomes of three numbers or chromosomes with twenty, were the first and last number is the same, and we could select when running the code were to star and were to finish, it was defined that the maximin number of elements in our chromosome was the total number of nodes in the grah minus two, taking into account the locked numbers. All the code images are from Villareal (2024) solution to the TSP with a genetic algorithm.

```python
def random_chromosome(self):
    # Ensure a small chance of generating a chromosome with only start and end nodes
    if random.random() < 0.1:
        return [self.start_node, self.end_node]
    # Generate a random chromosome with intermediate nodes
    length = random.randint(1, self.max_length - 2)
    middle_nodes = [random.randint(0, len(self.Adjacency) - 1) for _ in range(length)]
    return [self.start_node] + middle_nodes + [self.end_node]
```

Code 1 Shuffling the chromosomes.

Once we had our building blocks (chromosomes) we need to make a function that allows the crossover between two chromosomes (parents) to make a viable child. As there were different lengths of chromosomes, the split of the chromosome is random, and it swaps the first part of one parent with the second half of the other, the remaining halves are united in a second child. This random swap of the chromosomes is where it might combine beneficial traits of both parents. Additionally, there are restrictions in the probability to create a child which is only the start and end node, as the crossover need the parents to be at least four or more numbers including the locked numbers.

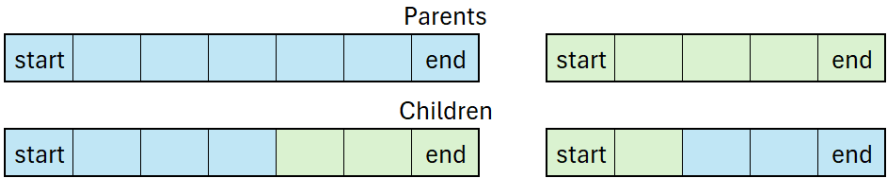


Figure 1 Chromosome splitting and descendants.

```python
def crossover(self, parent1, parent2):
    # Determine the minimum length of parent chromosomes
    min_len = min(len(parent1), len(parent2)) - 2
    if min_len < 1:
        # If parents are too short, return copies of them
        return parent1[:], parent2[:]

    # Perform crossover at a random point
    crossover_point = random.randint(1, min_len)
    # Create two children by exchanging parts of parents' chromosomes
    child1 = [self.start_node] + parent1[1:crossover_point+1] + parent2[crossover_point+1:-1] + [self.end_node]
    child2 = [self.start_node] + parent2[1:crossover_point+1] + parent1[crossover_point+1:-1] + [self.end_node]
    return child1, child2
```

Code 2 Use selected parents to split and make two children.

Next to add a level of realism and up the chances to find an optimal solution we need to take into account the mutations chromosomes might experience when they are a descendant, to simulate this we made three different kind of changes the child may experience, and random **insertion** of a number on a random slot of the array, a random **deletion** of a number in the array or a random **change** in the number it has on a slot, where the only numbers never to be touch are still the first number and the last. This ensures the diversity of our genetic pool within the new population.

```python
def mutate(self, chromosome):
    # Ensure the chromosome has enough length for mutation
    if len(chromosome) > 2:
        # Randomly select mutation type: insert, delete, or change
        mutation_type = random.choice(['insert', 'delete', 'change'])
        # Randomly select position for mutation
        position = random.randint(1, len(chromosome) - 2)
        if mutation_type == 'insert' and len(chromosome) < self.max_length:
            # Insert a random node at the chosen position
            node = random.randint(0, len(self.Adjacency) - 1)
            chromosome.insert(position, node)
        elif mutation_type == 'delete' and len(chromosome) > 3:
            # Delete a node at the chosen position
            chromosome.pop(position)
        elif mutation_type == 'change':
            # Change the node value at the chosen position
            node = random.randint(0, len(self.Adjacency) - 1)
            chromosome[position] = node
```

Code 3 Function used to mutate chromosomes.

As we have the three basic functions to imitate the biological process of evolution, we need to make a way for the algorithm to measure the worthiness of each chromosome, this is what we call the fitness function, that counts the total weight of the route stablished in the number array using the adjacency matrix defined at the beginning. By now it may have crossed your mind that the randomness in which we made our chromosomes doesn't ensure that the numbers in an array are a possible path, in other word, the nodes represented in the chromosomes may not have an edge to connect them, this is taken into account in this part of the algorithm as it assigns the fitness of an impossible path as infinite, which reduces significantly the chances of it making it to the next generation.

```python
def fitness(self, chromosome):
    # Calculate the total cost of the given chromosome
    total_cost = 0
    for i in range(len(chromosome) - 1):
        cost = self.Adjacency[chromosome[i], chromosome[i + 1]]
        # If cost is infinite, return infinity indicating infeasible solution
        if cost == np.inf:
            return np.inf
        total_cost += cost
    return total_cost
```

Code 4 Function that calculates the distance between stations.

With the fitness calculated, we can now know what pair of chromosomes are best to make parents for the next generation, but the algorithm needs to select which ones to use, for this we have the next function. To do this, the fitness scores are converted into a NumPy array for efficient operations and considering there may be only fitness scores of infinites, indicating no feasible solutions, a new random population is generated. Otherwise, selection probabilities are calculated as the inverse of fitness scores where solutions with infinite scores are given zero probability.

These probabilities are normalized so their sum equals one and randomly selects chromosomes to make into parents, allowing for possible repeated selection of the same chromosome. These processes are biases towards fitter individuals mimicking natural selection and making the new generation according to these adjusted probabilities, promoting genetic diversity and the potential for improved solutions with the fittest parents.

```python
def select(self, population, fitness_scores):
    # Convert fitness scores to numpy array for efficient operations
    fitness_scores = np.array(fitness_scores)

    # If all chromosomes have infinite fitness scores, generate a new population
    if np.all(np.isinf(fitness_scores)):
        return [self.random_chromosome() for _ in range(self.Npop)]

    # Calculate selection probabilities based on fitness scores
    probabilities = 1 / (fitness_scores + 1e-6)
    probabilities[np.isinf(fitness_scores)] = 0

    total_probability = np.sum(probabilities)
    if total_probability > 0:
        probabilities /= total_probability
    else:
        probabilities = np.ones_like(fitness_scores) / len(fitness_scores)

    # Select new population by roulette wheel selection
    selected_indices = np.random.choice(len(population), size=self.Npop, replace=True, p=probabilities)
    return [population[i] for i in selected_indices]
```

Code 5 Function to select best parents in the population.

As we have all the component to make a generation and the next, we continue by making the part of the algorithm that combines all this together and iterates the selection, crossover and mutation to make new populations with new solutions in each one, taking into account the fitness in each chromosome to narrow the paths into the optimal solution. The number of generations it generates are user input, and it constantly measures the fitness of all the chromosomes inside the population, selects the best one according to the probabilities, choses randomly between the best ones and makes the act as the parents and add the children to the next generation of the population. This until it fulfills the user input length of each population.

```python
def evolve(self):
    # Execute genetic algorithm for a given number of iterations
    for _ in range(self.Nit):
        # Calculate fitness scores for current population
        fitness_scores = [self.fitness(chrom) for chrom in self.population]
        # Select new population based on fitness scores
        self.population = self.select(self.population, fitness_scores)
        new_population = []
        # Generate new population through crossover and mutation
        while len(new_population) < self.Npop:
            parent1, parent2 = random.sample(self.population, 2)
            child1, child2 = self.crossover(parent1, parent2)
            self.mutate(child1)
            self.mutate(child2)
            new_population.extend([child1, child2])

        # Update population with new generation
        self.population = new_population[:self.Npop]
```

Code 6 Function to generate new populations.

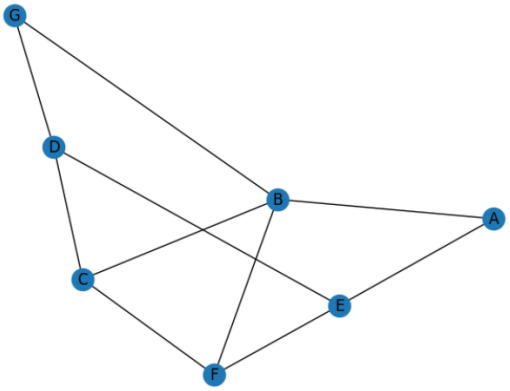
Finally, we have the functions that run all this together and prints the best result it could produce, writing the stations it runs by and the fitness function of the chosen path. These functions summarize what the evolve function comes up with and choses the best chromosome based on the fitness. Additionally, if for any reason of pure bad luck, the best solution is still a path with an infinite cost, it deletes everything and runs the algorithm again.

```python
def answer(self, chrom):
    # Print the stations to take for the given chromosome
    stations = [self.nodes[node] for node in chrom]
    print('Stations to take:', stations)

def run(self):
    # Run the genetic algorithm and print the best solution found
    self.evolve()
    final_fitness = [self.fitness(chrom) for chrom in self.population]
    best_index = np.argmin(final_fitness)
    best_chromosome = self.population[best_index]
    best_fitness = final_fitness[best_index]

    # If the best fitness is infinity, run the algorithm again
    if best_fitness == np.inf:
        self.run()
    else:
        print("Best chromosome:", best_chromosome)
        print("Best fitness (path cost):", best_fitness)
        # Print the stations to take for the best chromosome
        self.answer(best_chromosome)
```

Code 7 Print the result of the algorithm.

Small scale test

The main problem to solve in this project includes thirty-six nodes, which represents the stations in the subway of Mexico City, making testing the code slow for testing while developing the code, so for better understanding and making manually verifying the optimal solution easier, a smaller scale test was develop.



Code 8 Graph of small test scale.

```
[[inf  5. inf inf  4. inf inf]
 [ 5. inf  4. inf inf  2.  1.]
 [inf  4. inf  3. inf  4. inf]
 [inf inf  3. inf  2. inf  4.]
 [ 4. inf inf  2. inf  6. inf]
 [inf  2.  4. inf  6. inf inf]
 [inf  1. inf  4. inf inf inf]]
```

Figure 2 Adjacency matrix of small test scale.

Depending on where we want to go and from where, as there are few nodes and edges, we can verify if the code works as intended. To evaluate it, we want to move from A to F, and from G to C, where manually there is one obvious optimal solution where one is to move from A to B to F, with a distance of 7, and the second is to move from G to B to C, with a distance of 5. Now we try the code by Villareal, and we can see it works as intended.

Example 1

Moving from A to F

```
ga = GeneticAlgorithm(Npop=100, Nit=50, nodes=nodes_simple,
                      max_length=len(Adjacency), Adjacency=Adjacency, start_node=0, end_node=5)
ga.run()
```

✓ 0.0s

```
Best chromosome: [0, 1, 5]
Best fitness (path cost): 7.0
Stations to take: ['A', 'B', 'F']
```

Example 2

Moving from G to C

```
ga = GeneticAlgorithm(Npop=100, Nit=50, nodes=nodes_simple,
                      max_length=len(Adjacency), Adjacency=Adjacency, start_node=6, end_node=2)
ga.run()
```

✓ 0.0s

```
Best chromosome: [6, 1, 2]
Best fitness (path cost): 5.0
Stations to take: ['G', 'B', 'C']
```

Figure 3 Results of genetic algorithm in small test scale, with 50 generations and a population of 100 each.

Mexico City Subway.

As we now know the algorithm works, we can move onto the larger graph that is the subway system in Mexico City, were manually finding an optimal solution is more complex as there are too many ways in which we can move, each with a different advantage in distance between them. First, we model our graph after this map:

Figure 4 Map of Mexico City's subway.

As we can appreciate, the nodes and edges in this map is a lot more that our small test scale and the weights in it varies a lot more, there are some that are only one unit apart whereas others are up to seven units, so finding an optimal solution no matter our starting point and our goal is difficult. For our project we are interested in the shortest path between “El Rosario” and “San Lázaro”. The python non-directed graph of the map is the following:

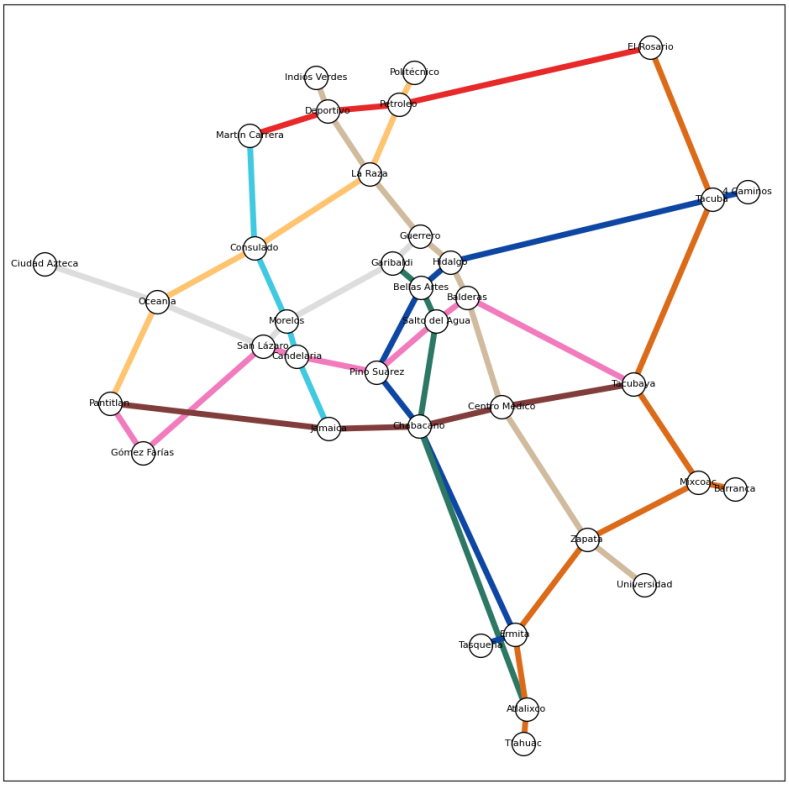


Figure 5 Python graph of Mexico City subway system.

In our case, “El Rosario” was defined as node 0 and “San Lázaro” is 21, as we already have the general algorithm, we only need to modify the input values in our code to use the adjacency matrix of the subway and as this is a larger scale, we can increase the population per generation and the number of generations we create, so we made several iterations to observe the results.

```
subway = GeneticAlgorithm(Npop=1000, Nit=500, nodes=nodes,
                          max_length=len(Subway_Adjacency), Adjacency=Subway_Adjacency,
                          start_node=0, end_node=21)
subway.run()
```

✓ 19.8s Python

```
Best chromosome: [0, 2, 8, 9, 14, 21]
Best fitness (path cost): 14.0
Stations to take: ['El Rosario', 'Petroleo', 'La Raza', 'Consulado', 'Morelos', 'San Lázaro']
```

```
subway = GeneticAlgorithm(Npop=500, Nit=100, nodes=nodes,
                          max_length=len(Subway_Adjacency), Adjacency=Subway_Adjacency,
                          start_node=0, end_node=21)
subway.run()
```

✓ 3m 5.7s Python

```
Best chromosome: [0, 2, 8, 9, 14, 21]
Best fitness (path cost): 14.0
Stations to take: ['El Rosario', 'Petroleo', 'La Raza', 'Consulado', 'Morelos', 'San Lázaro']
```

```
subway = GeneticAlgorithm(Npop=500, Nit=500, nodes=nodes,
                          max_length=len(Subway_Adjacency), Adjacency=Subway_Adjacency,
                          start_node=0, end_node=21)
subway.run()
```

✓ 2m 14.4s

```
Best chromosome: [0, 2, 8, 9, 10, 21]
Best fitness (path cost): 17.0
Stations to take: ['El Rosario', 'Petroleo', 'La Raza', 'Consulado', 'Oceania', 'San Lázaro']
```

Figure 6 Different iterations for the problem.

As we can see, the algorithm found a solution which is only 14 units of distance between our desired stations, even when we modify the number of generations and the population, we even appreciate the difference in time it took for it to arrive to the same conclusion, where one was in 19 seconds and the other were almost 4 minutes, and to illustrate how the algorithm can find different results depending on the iteration, we can see it found a completely different “optimal” solution, which is longer than the first answers, which is interesting.

Conclusion

Personally, the hardest part was defining a viable way to make the chromosomes even when there are a number of resources on the internet, as these are the basic and most fundamental part of this algorithm and having a flimsy base in the construction makes progressing and developing an optimal solution almost as complex as a brute force method on certain scales.

Additionally, the crossover method was not sunshine and rainbows even with a good chromosome selection methos, as it may vary how we produce the next generation if the chromosomes are all the same length or different (as is in this case), were the randomness benefits the evolution but can also make some really time consuming iterations, as it is shown in the results of our project, were the algorithm arrived at the same optimal solution but in two very different times.

I also find interesting how it does not always arrive to the best solution, which is the one that weights fourteen units, even when we made the best possible environment to arrive to the shortest path, as it sometimes throws a solution with seventeen units of distance and sometimes eighteen units. Additionally, it is interesting that sometimes with a larger number of generations to create with more population in each, the algorithm reaches the solution faster that on a smaller number of generations with less population.

Overall, this algorithm is fascinating, specially with the way it imitates an actual biological process and work, it is marvelous that code can imitate reality to such a degree and have a conclusion that evolution and natural selection is a valid way to optimize complex problems, where to many variables are involved or the best solution can be subjective to the user.

GitHub Link

For the full code:

https://github.com/SofiaGC13/Artificial_Intelligence_Repository/tree/a88907718eaed41dd77c5c4106d9dfe36a01402c/FC3%3A%20Genetic%20Algorithm%20in%20Python

References

- Adewole, P., & Akinwale, A. (2011). A Genetic Algorithm for Solving Travelling Salesman Problem. ResearchGate. Recuperated from https://www.researchgate.net/publication/49613263_A_Genetic_Algorithm_for_Solving_Travelling_Salesman_Problem

-Villareal, D. (2024). FC3. GitHub. Recuperated from https://github.com/DiegoVilla03/Artificial-intelligence/blob/main/Genetic%20algorithm/FC3.ipynb

- OpenAI. (2024). ChatGPT interaction. TSP Genetic Algorithm. OpenAI. Recuperated from https://chat.openai.com

- Stoltz, E. (2018). Evolution of a Salesman: A Complete Genetic Algorithm Tutorial for Python. Towards Data Science. Recuperated from https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35