23-4-2024

FC3: Genetic Algorithm

Diego Pardo Gonzalez

Artificial Intelligence

Introducción

The Traveling Salesman Problem, or TSP, is a classic quandary in optimization and computer science that presents an apparently simple yet very complex challenge. Picture a salesman who needs to visit a series of cities. The goal is to figure out the shortest possible route that allows the salesman to visit each city once and return to the starting point. This problem might sound straightforward, but its complexity goes up fast, as the number of cities increases.

The TSP is about finding the most efficient path through a network of interconnected points (cities), where each connection (road) has a cost (distance). The problem is represented using graph theory, where cities are vertices and roads are weighted edges. The weight reflects the cost of traveling between cities.

The difficulty of the TSP lies in its NP-hardness, a classification in computational complexity that indicates there's no known way to solve the problem efficiently for all possible configurations as the size of the problem grows. This has significant implications in real-world scenarios like logistics for delivery routes or scheduling where optimal solutions are crucial for efficiency and cost reduction.

Despite the theoretical challenges, the TSP is not just an academic exercise but has practical applications across various fields. From logistics companies optimizing delivery routes to manufacturers improving the movements of robotic arms on the assembly line, solutions to the TSP can offer substantial economic benefits. The problem also appears in scientific research, such as plotting telescopes' movements in astronomy or sequencing tasks in genetics.

Solving the TSP can be approached in different ways. Exact methods like dynamic programming or branch-and-bound algorithms are used to find the definitive shortest path, but these methods can become computationally expensive as the number of cities grows. This limitation has led to the popularity of heuristic and metaheuristic approaches like genetic algorithms, simulated annealing, and ant colony optimization—which trade some accuracy for speed and are often good enough for practical purposes.

Map

In the following map we have the subway system of Mexico city, we can notice that we have 13 lines in our map with an amount of 36 stations. The problema we need to solve is to try to get to the station “San Lazaro” when our starting point is the station “El Rosario”, if we can do it the objective is to reduce the time that it takes to get to “San Lazaro” station. Otherwise if our algorithm cannot do it explain why.

Stations list

1. El Rosario
2. Politecnico
3. Instituto del petróleo
4. Indios Verdes
5. Deportivo 18 de Marzo
6. Martin Carrera
7. La Raza
8. Tacuba
9. Cuatro Caminos
10. Consulado
11. Ciudad Azteca
12. Oceania
13. Morelos
14. Garibaldi
15. Gerrero
16. Hidalgo
17. Bellas Artes
18. Balderas
19. Salto de agua
20. Pino Suarez
21. San Lazaro
22. Candelaria
23. Gomez Farias
24. Pantitlan
25. Jamaica
26. Chabacano
27. Centro medico
28. Tacubaya
29. Mixcoac
30. Barranca del Muerto
31. Universidad
32. Zapata
33. Ermita
34. Taxqueña
35. Tlahuac
36. Atlalilco

Composicion of every Subway line with costs of traveling (distance)

Line 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

Line 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

Line 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

Line 4:

From Martín Carrera to Consulado: 3

From Consulado to Morelos: 2

From Morelos to Candelaria: 1

From Candelaria to Jamaica: 2

Line 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

Line 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

Line 7:

From El Rosario to Tacuba: 4

From Tacuba to Tacubaya: 5

From Tacubaya to Mixcoac: 3

From Mixcoac to Barranca del Muerto: 1

Line 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

Line 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

Algorithm Analysis

`__init__` (Constructor)

This function initializes the genetic algorithm class with specific parameters. It sets up the fundamental components of the genetic algorithm such as the population size (`Npop`), number of iterations (`Nit`), graph nodes (`nodes`), the maximum length of a chromosome (`max_length`), adjacency matrix of the graph (`Adjacency`), and specific start and end nodes. After initializing these attributes, it also generates an initial population of random chromosomes using the `random_chromosome` method.

```
def __init__(self, Npop, Nit, nodes, max_length, Adjacency, start_node, end_node):
    # Number of chromosomes
    self.Npop = Npop
    # Number of iterations
    self.Nit = Nit
    # List of nodes in the graph
    self.nodes = nodes
    # Maximum chromosome length
    self.max_length = max_length
    # Graph adjacency matrix
    self.Adjacency = np.array(Adjacency)
    # Fixed start node
    self.start_node = start_node
    # Fixed end node
    self.end_node = end_node
    # Initialize population with random chromosomes
    self.population = [self.random_chromosome() for _ in range(Npop)]
```

`random_chromosome`

This method creates a single random chromosome. A chromosome in this context is a potential solution to the problem, representing a path from a start node to an end node. There's a 10% chance of creating a direct path from the start to the end node. Otherwise, it generates a path that includes a random number of intermediate nodes chosen randomly. The length of the path is constrained by `max_length`.

```
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]
```

`crossover`

This function takes two parent chromosomes and combines their genetic material to produce two new child chromosomes. It first determines a random crossover point and ensures the crossover point is valid (i.e., does not result in an empty segment). Then, it splits each parent at this point and swaps their segments to generate children, ensuring each child starts at the start node and ends at the end node. This mimics biological reproduction, where genetic material is mixed from two parents to produce offspring.

```
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
```

`mutate`

Mutation introduces random changes to a chromosome, which is crucial for maintaining genetic diversity in the population. Depending on the selected mutation type ('insert', 'delete', 'change'), this method randomly alters the chromosome by inserting a new node, deleting an existing node, or changing a node to another random node. This helps to explore new genetic combinations and potentially improve solutions.

```
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
```

`fitness`

This function calculates the "fitness" of a chromosome, which represents how good the solution is. In this algorithm, fitness is defined as the total cost of traveling through the path represented by the chromosome. It sums the costs of traveling between consecutive nodes as specified in the adjacency matrix. A path with a non-existent edge (denoted by `np.inf` in the adjacency matrix) is deemed infeasible and assigned an infinite fitness, indicating it's a poor solution.

```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
```

`select`

Selection is a process where chromosomes are chosen from the current population to create a new generation. This method uses a technique known as "roulette wheel selection," where the probability of each chromosome being selected is inversely proportional to its fitness score (lower cost paths are more likely to be chosen). It handles special cases like all chromosomes being infeasible by generating a completely new population.

```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]
```

`evolve`

This method runs the genetic algorithm over the specified number of iterations (`Nit`). In each iteration, it evaluates the current population's fitness, selects the best candidates to form a new generation, and applies crossover and mutation to create new chromosomes. This iterative process is aimed at improving the population's overall fitness over time.

```
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]
```

`answer`

After finding a solution, this method formats and prints the path in a human-readable form by translating node indices back to their respective node names (stations). This makes the output more understandable for users, showing the sequence of stations along the best path found.

```
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)
```

`run`

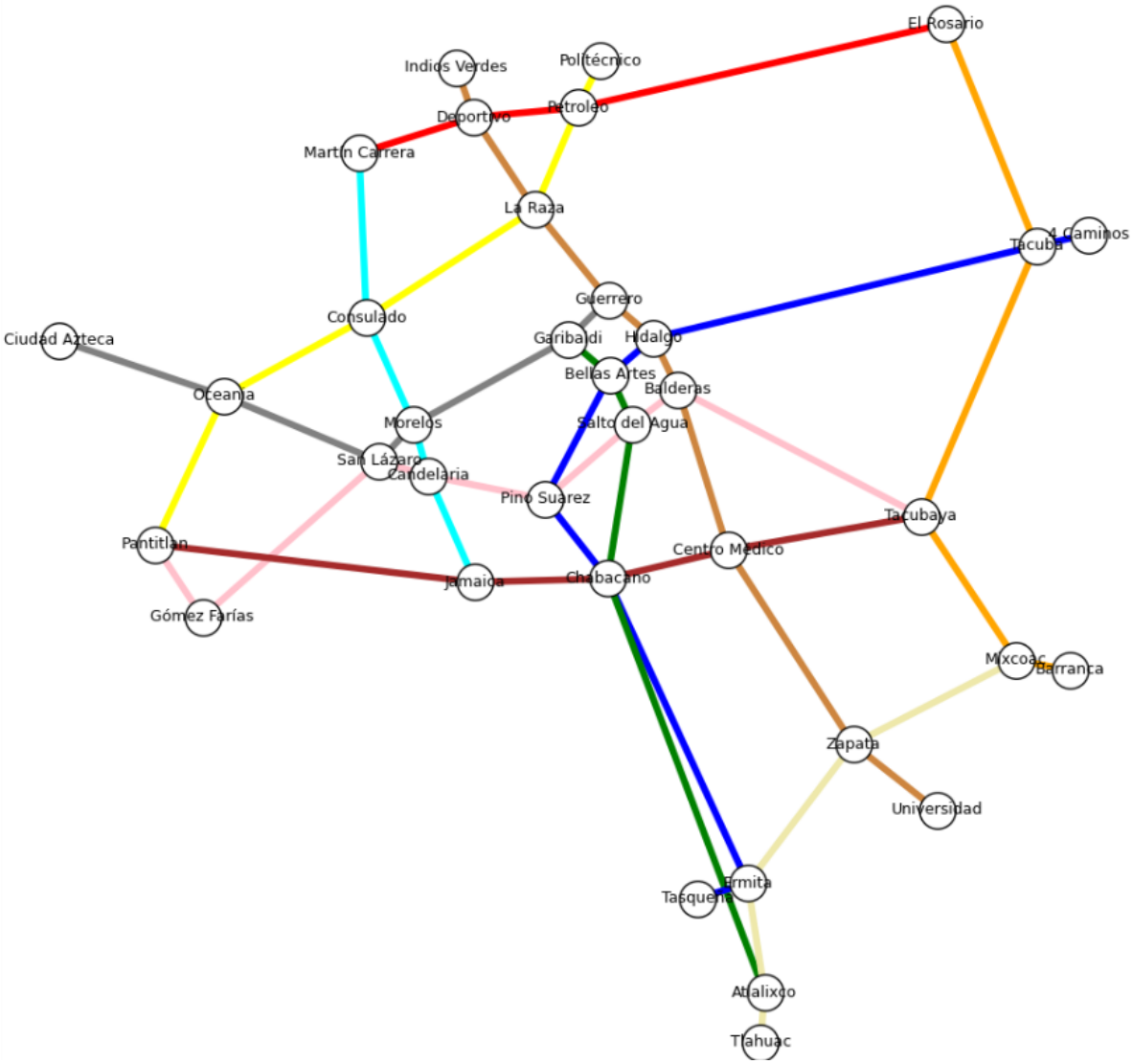
This is the main method that kicks off the genetic algorithm. It repeatedly calls `evolve` until a satisfactory solution is found or the algorithm decides no better solution can be found within the current genetic material. If the best solution found has an infinite cost, indicating no valid path has been found, the algorithm restarts. This method also prints the best path and its cost at the end of the execution.

```
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)
```

These functions collectively encapsulate the genetic algorithm operations tailored for solving shortest path problems in graphs represented as adjacency matrices. Each part plays a role in evolving solutions over generations, aiming to find an optimal or near-optimal path from a start node to an end node in a graph.

The grph that is created with the specifications of the adyancency matrix is the following one:




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

```
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']
```

In this case the result of the best path was actually the best path (14), but if you run it is very probable to give you another result, it depends on how the mutation is storing the optimal answar everytime you run it.

References

https://www.researchgate.net/publication/49613263_A_Genetic_Algorithm_for_Solving_Travelling_Salesman_Problem

file:///C:/Users/Propietario/Downloads/FC%203_Genetic%20Algorithm%20in%20Python_Sof%C3%ADaGraham.pdf

OpenAI. (2024). *ChatGPT*. Chat.openai.com; OpenAI. https://chat.openai.com/