Shortest path in graphs with Genetic Algorithms

Diego VIllarreal De La Cerda

Universidad de las Americas Puebla UDLAP

Artificial Intelligence LIS3082

Abstract*—The following report shows step by step how i solve a modification of the Travelling Salesman problem(TSP), known to be a NP-Complete problem with genetic algorithms a type of programming paradigm based on the evolution theory proposed by Charles Darwin, where only the most capable beings survive, the use of genetic algorithms result in a lower computational complexity with the drawback of rely in probability*

***Keywords*—** *The salesman Problem, NP-Complete, Genetic Algorithms*

1. Introduction

In the field of computer science there exist many algorithms to solve a wide range of problems that have a computational cost usually denoted by the ***Big O*** notation that indicates an algebraically function in terms of time consuming, a particular group of problems known as **Non Polynomial** are those labeled as **O(n!)** with a factorial complexity

1.1. The Salesman Problem

The *"Travelling Salesman Problem"* also known as TSP is about a man that wants to sale his products in many cities to get the most profit from it, then given a certain number of cities to visit and and the connections between them with distance the goal is to visit all cities only once and return to his start point to start again the next day this in the field of mathematics and graph theory is called a **"Hamiltonian Cycle"** this type of problem can be solved by deterministic algorithms like Depth-First Search (DFS) or brute force, and is guaranteed by Dirac's Theorem: If G is a simple graph with n vertices (where n > 2) such that every vertex has degree at least n/2, then G has a Hamiltonian cycle, but computationally find this solution may take a lot of time specially when the nodes reach the thousands or hundred of thousands [1]

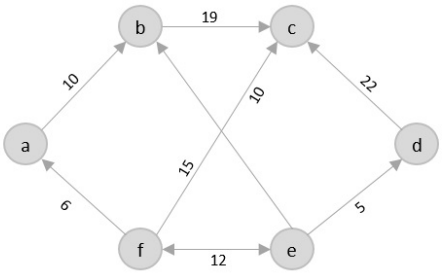

Figure 1. *Salesman Graph example*

1.2. Genetic Algorithms

The approach of genetic algorithms reduces the time complexity by giving an approximating of the solution sthocastical methods based in the Darwin's evolution theory

the way it works is given a structure to represent a Chromosome and a population of Chromosomes they are evaluated by a fitness function that indicates a numerical way of how good and individual is and through the definition of process of mutation and combination, the Chromosomes "Evolves" and converge to a optimal or near optimal solution.[2]

2. The Shortest Path Problem

For this report I am focusing in a similar problem as TSP this is to find the shortest path between two nodes in a given non directed graph the solution may be a ordered sequence of nodes to visit before getting to the end and this solution may be variable in length some scenarios can get a elemental solution in 1 step or may have a solution in at most n steps with n the number of nodes so our code may also contemplate this, also we are working in a non dense graph that means that each node is not connected in one step with other nodes so many connection does not exist and is needed a way to represent it, with that in mind i define a solution as follows

2.1. Representing the Chromosome

i decided to represent the chromosome like an array of numbers, each node will be denoted by a number that identifies it, so the sequence of numbers represent the path to follow, also to make it dynamic and capable of handle variable solution size, like in the nature each chromosome will be generated with a random length limited by the number of nodes in the graph,

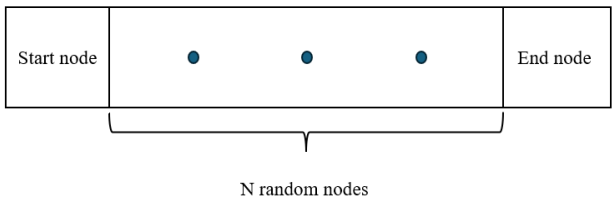

Figure 2. *Chromosome Structure*

2.2. Representing the connections

A very simple way to represent a weighted graph is with an adjacency matrix the rows and columns representing the vertices between ordered nodes

$$\begin{bmatrix} & A & B & C & D \\ A & 0 & 3 & 2 & 0 \\ B & 3 & 0 & 0 & 4 \\ C & 2 & 0 & 0 & 1 \\ D & 0 & 4 & 1 & 0 \end{bmatrix}$$

in the example matrix above it says that the connection between node A and B is 3 is important to notice that the matrix is symmetrical to the diagonal due to is a non directed graph, and the connections that don not exist are 0 in our code this

null connections are represented as infinite with the help of numpy np.inf

3. The Algorithm

```
class GeneticAlgorithm:
    def __init__(self, Npop, Nit, mu, xrat, max_length, Adjacency, start_node, end_node):
        self.Npop = Npop  # Number of chromosomes
        self.Nit = Nit  # Number of iterations
        self.mu = mu  # Mutation ratio
        self.xrat = xrat  # Crossover ratio
        self.max_length = max_length  # Maximum chromosome length
        self.Adjacency = np.array(Adjacency)  # Graph adjacency matrix
        self.start_node = start_node  # Fixed start node
        self.end_node = end_node  # Fixed end node
        self.population = [self.random_chromosome() for _ in range(Npop)]

    def random_chromosome(self):
        if random.random() < 0.1:
            return [self.start_node, self.end_node]
        length = random.randint(2, 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]

    def crossover(self, parent1, parent2):
        min_len = min(len(parent1), len(parent2)) - 2
        if min_len < 1:
            return parent1[:], parent2[:]

        crossover_point = random.randint(1, min_len)
        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

    def mutate(self, chromosome):
        if len(chromosome) > 2:
            mutation_type = random.choice(['insert', 'delete', 'change'])
            position = random.randint(1, len(chromosome) - 2)
            if mutation_type == 'insert' and len(chromosome) < self.max_length:
                node = random.randint(0, len(self.Adjacency) - 1)
                chromosome.insert(position, node)
            elif mutation_type == 'delete' and len(chromosome) > 3:
                chromosome.pop(position)
            elif mutation_type == 'change':
                node = random.randint(0, len(self.Adjacency) - 1)
                chromosome[position] = node

    def fitness(self, chromosome):
        total_cost = 0
        for i in range(len(chromosome) - 1):
            cost = self.Adjacency[chromosome[i], chromosome[i + 1]]
            if cost == np.inf:
                return np.inf
            total_cost += cost
        return total_cost

    def select(self, population, fitness_scores):
        fitness_scores = np.array(fitness_scores)

        if np.all(np.isinf(fitness_scores)):
            return population

        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)

        selected_indices = np.random.choice(len(population), size=self.Npop, replace=True, p=probabilities)
        return [population[i] for i in selected_indices]

    def evolve(self):
        for _ in range(self.Nit):
            fitness_scores = [self.fitness(chrom) for chrom in self.population]
            self.population = self.select(self.population, fitness_scores)
            new_population = []
            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])

            self.population = new_population[:self.Npop]

    def run(self):
        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]
        print("Best chromosome:", best_chromosome)
        print("Best fitness (path cost):", best_fitness)
```

Code 1. *Genetic Algorithms class*

The code works by the object oriented programming, with a given parameters, the *"init"* method establish all the needed parameters the Npop is the population size, Nit is the number of iterations or evolution process that the chromosomes will go through before giving an output, max length determine how big the chromosome may be ,Adjacency is the adjacency

matrix of the graph we want to find an path, the start and end node as fixed nodes self explanatory and the population that is a list with chromosomes, the length of this list is delimited by Npop

3.1. Methods

3.1.1. Random Chromosome. this method is used to generate the initial population, each chromosome is generated with a random length between 1 and the maximum length minus 2 corresponding to the fixed start and end nodes then to the empty array is filled with random numbers in the range of number of nodes and finally append the start and end node at the beginning and end of the array

also to cover all scenarios there is a small probability to do not generate middle nodes and just put together the start and end node that lead us to elemental solutions that can also be possible

3.1.2. crossover. the method takes 2 chromosomes from the population and crossover between them, due the dynamic length of the chromosomes a security measure i take is avoid crossover between chromosomes if one of them has length less than 1 because the crossover point need to be at least 2

if both parents pass the condition then it select a random point between 1 and the minimum length of the shortest chromosome and swap it

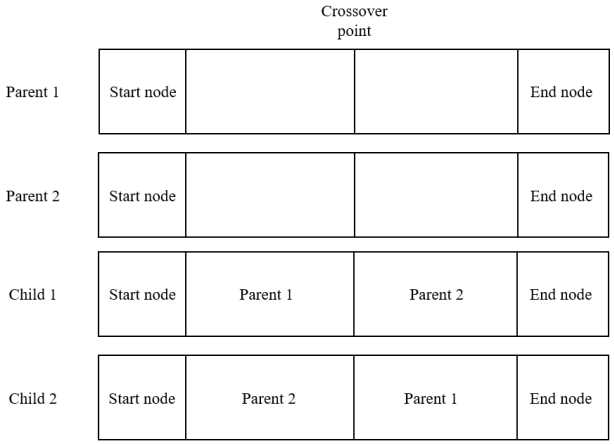

Figure 3. *Crossover representation*

3.1.3. mutate. the mutate method i build is heavily inspired in the how nature works and define 3 types of mutation ***Insert, Delete and change*** i will explain briefly, from a given random point in the chromosome, that chromosome will change depending on the type of mutation the **Insert** mutation will take that random position in the chromosome and add a new random node as long as the new chromosome length does not exceed the maximum allowed length the **Delete** mutation does is the inverse of Insert, takes a random position of the chromosome and pop it from the chromosome resulting in a shorter Chromosome this can only be done if the length of the chromosome is at least 3 The **Change** mutation takes also a random position of the chromosome and replace it with another random number

this mutations are only done to the middle nodes so in all the process the start and end are kept

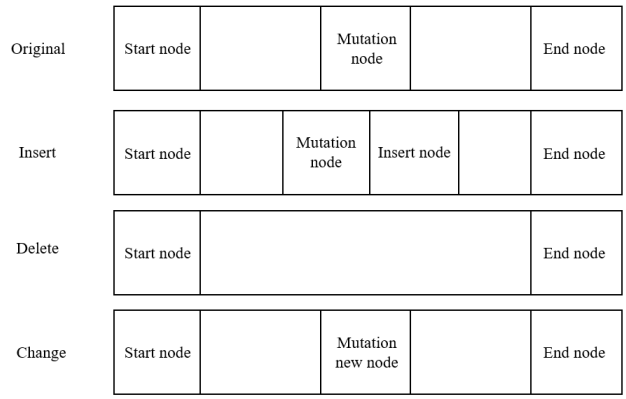

Figure 4. *Mutation Representation*

3.1.4. Fitness. the fitness function calculates the sum of all edges and assign that sum to the corresponding chromosome when one of the edges does not exist and the weight is infinite the algorithm directly assign that chromosome as infinite

3.1.5. Select. The Select function gives the algorithm the concept of elitism the fitness score of each chromosome is ponderated in a probability of being chose those chromosomes with a fitness score of infinite the probability of being selected is 0 so the new population is composed of the most fit chromosomes

3.1.6. Evolve. in a range of number of iterations are done the the crossover operations and the mutations over the selected chromosomes of the population the convergence of the algorithm highly depends in the start parameters of the Number of population and number of iterations with less iterations and number of populations is less probable to get a valid solution

4. Example

Let's test the code with a simple Graph

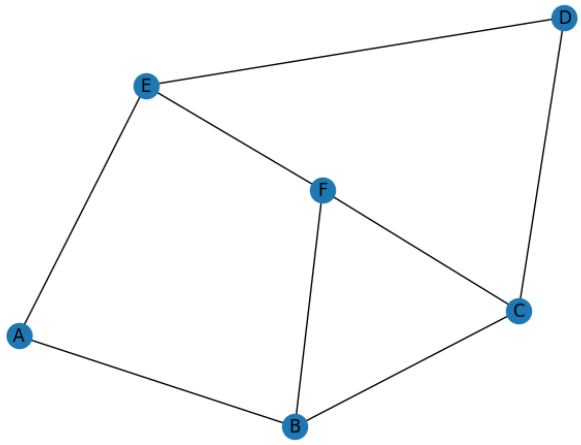

Figure 5. *Graph example*

lets suppose we want to go from node D to node F this are represented as node 3 to node 5

```
ga = GeneticAlgorithm(Npop=100, Nit=50, mu=0.1,
    xrat=0.5, max_length=len(Adjacency),
    Adjacency=Adjacency, start_node=3, end_node
    =5)
ga.run()


Best chromosome: [3, 2, 5]
Best fitness (path cost): 7.0
```

Code 2. *Example 1*

then the optimal path is starting on D go to C and then to F

Now to test the performance of the algorithm with simple solutions lets suppose we want to from node A to node E there exist a simple path of one step to get there

```
ga = GeneticAlgorithm(Npop=100, Nit=50, mu=0.1,
    xrat=0.5, max_length=len(Adjacency),
    Adjacency=Adjacency, start_node=0, end_node
    =4)
ga.run()

Best chromosome: [0, 4]
Best fitness (path cost): 4.0
```

Code 3. *Example 2*

The Code gets to the optimal as expected in one step so now lets move on to our real problem

5. Mexico City subway problem

The Mexico city subway is a big mess in all terms so there is a need to optimize and find the shortest path between stations, so we are contemplating a reduced and simplified representation of the subway lines

Figure 6. *Mexico City Subway*

the goal is to get from the station **"El Rosario"** to **"San Lázaro"** so the representation of the graph is the following

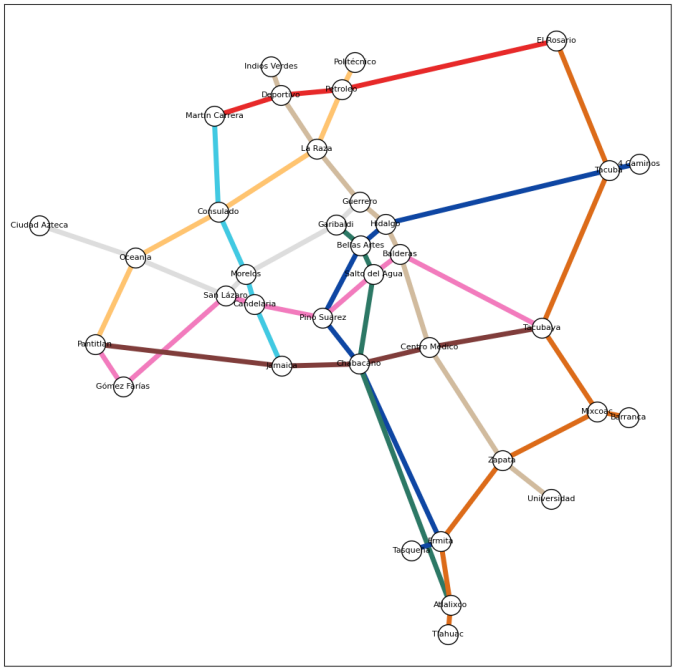

Figure 7. *Mexico City graph*

then in our representation the node associated to El Rosario is 0 and the node associated to San Lázaro is 21

```
subway = GeneticAlgorithm(Npop=1000, Nit=500, mu
    =0.1, xrat=0.5,
                          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
```

Code 4. *shortest path*

the output represents the path from El rosario to instituto del petroleo then to La Raza and Consulado, from Consulado to Morelos and Finally to San Lázaro (More info on how to translate from numbers to stations in the github repository)

6. Conclusions

The genetic Algorithms provide a very good solution that can be optimal or near optimal, the only issue i found is that the convergence of the algorithm heavily depend on the initial parameters may exist cases where a whole population has a cost of infinite and the algorithm does not find a solution, in any case is better than trying by brute force or a deterministic algorithm if the odds is in our favor

References

[1] G. Bezalel, "The travelling salesman problem and related problems", *Massachusetts Institute of Technology, Operations Research Center*, 1978.

[2] K. Codes, "Genetic algorithms", https://github.com/kiecodes/genetic-algorithms, Tech. Rep., 2021.