Genetic Algorithm Model: Traveling Salesmen Problem

Nianyi Wang

Section I: Statement and Overview

Traveling Salesmen Problem is a classic NP-complete problem in computer science that we are given some number of cities and distances between each pair of cities and find the shortest possible route that visits each city exactly once and returns to where we start[1].

The main focus of this problem is to find the shortest route that visits all cities exactly once and returns. In order to find it, the order of visiting those cities matters. We first need to find all possible solutions that visit all the cities exactly once and there are up to (number of cities)! possible solutions. Then we have to calculate the total traveling distance for each solution according to the distance and the order of that solution and find the minimum one among them.

Suppose we are given an integer n representing the total number of cities we want to visit, a nxn matrix with d[i][j] indicating the distance from city i to city j and the size of population we want to generate in our genetic algorithm, we will have enough information to calculate the total distance of one possible route. To simplify the problem, we will mark the cities with numbers 1 to n. For example, if we have 5 cities and mark them as 1,2,3,4,5, and the distance between each pair of cities is 1, then the total traveling distance for the route 1->2->3->4->5 and then return to city 1 is 5. Therefore, in order to solve the problem, we not only need to find the shortest distance but also the route that generates that shortest distance.

[1] Wikipedia contributors. "Travelling salesman problem." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 21 Apr. 2017. Web. 3 May. 2017.

Although there are n! possible solutions if we are given n cities, there are some solutions that are not valid under the background of this problem. Since we need to visit all the cities exactly once, the solution that does not contain all the cities or contains some city more than once are not a valid solution. For example, if we are given 5 cities, the solution [1,2,3,4,3] is not a valid solution since it does not visit city 5 and visits city 3 twice. Although we need to return to the start city, we will not include it in our solution string. We automatically assume that we will return to the start city in the end in order to get rid of the duplication of the start city and make the rule consistent. In this case, when we find a duplication in a solution, we don't need to check whether it's a duplication of the start city or not. We can simply claim that it's an invalid solution.

Generally, this model can be really useful when we try to make some traveling plans ahead of time. If we plan to travel by car during the vocation and we already know how many cities we want to go and how far these cities are between each other, we can use this model to find the shortest route that visits all the cities on our traveling list and finally come back home. The shortest route will largely save us lots of time and gas during traveling and maximize the time we can spend in each city.

Section II: Model Description

Recall that we are given an integer n, a distance matrix and a population size to implement this model. n represents the total number of cities we want to visit, the distance matrix contains all the information between two cities and population size indicates how large the population should be.

String

To construct the string, I would make the position represents the visiting order and the value of number in that position represents the city to be visited. The number of bits of

the string represents the total number of cities provided as we are required to visit all the cities. If the population size is m and the number of cities is n, then the initial population that will be generated is m n-bit strings. If we have a 5-bit string 14523, it means we start at city 1 and then visit city 4,5,2,3 in order and then return back to city 1. However, since string is immutable in python and java, I would implement it as an array of integers but still call it a “string” in order to keep the notation consistent. It is much easier to do mutation and crossover with integer array than string. Therefore, with my implementation, the index of the array represents the visiting order of the route and the value at that index represents which city to visit. Take the same example as above, a 5-bit string would be an integer array of length 5. The “string” [1,4,5,2,3] means we start at city 1 and then visit city 4,5,2,3 in order and then return back to city 1. In this case, the good “trait ”, a good visiting order, will be closer to each other and would be less likely to be broken and get modified. Therefore, the good trait will be preserved.

Fitness

For the fitness function, we are given a solution X and we already know the distance matrix. We have to look at the route of the solution provided and figure out what is the total distance of visiting all the cities in the order that the solution indicates. To calculate the total distance, we only need to get the distance of adjacent pairs of the solution from the distance matrix, i.e d[i][i+1] for i from 1 to n-1, and add all of them together. What’s more, we need to include the return distance in our total distance. Since the distance should be some positive number, the total distance should also be positive value, so we don’t need to worry about adding some constant to avoid negative values. However, since we want to get the shortest route, we want to minimize the total distance instead of maximizing it. Thus I would modify the return value to be 1/total_distance. In this case, if the total distance is larger, the return value is smaller and has smaller possibility during reproduction.

Reproduction

When we are given the population of one generation, we first calculate the total fitness of this generation. We find the fitness for each solution of the population and add them up. Then we calculate the percentage of total fitness of each solution in the population and figure out the cumulative sum of each solution according to the percentage. Because we scale the cumulative sum to 1000 to get rid of any decimals and make each number integer, we generate a random number between 1 and 1000. In this case, smaller distance will have larger range of the cumulative sum and thus have higher possibility to be reproduced and bigger distance will be less likely to be reproduced. We then need to check which range this random number falls into so we know which solution should be reproduced. From there, we create a new population of next generation and store those solutions in an array. This new array is our population of next generation.

Mutation

Since we don't allow duplications in the solution, we cannot simply mutate one position to any other city. What we want is to randomly swap the order of two cities instead of removing one city from the route or adding one city to the route. Thus, when we get the chance to do mutation, we randomly generate two integers between 1 and n indicating two positions and swap the values of two positions. By our implementation, we maintain the validity of the solution such that no duplications and no missing cities. In the end, we return the modified solution.

Take a 5-bit solution as an example:

Before mutation, the route is [1,2,3,4,5] and we randomly pick position 2 and 4(we need to make sure that once the mutation should occur, the positions randomly generated are not same):

1	2	3	4	5

After mutation, the route becomes [1,4,3,2,5], which does not contain any duplications and still covers all the cities:

1	4	3	2	5

In this case, the string gets mutated without any duplications.

Crossover

If we simply replace the sections of two solutions in this problem, we might face the situation that after crossover at some position, the solution generated might be invalid, i.e, the duplications occur or some cities are missing from the solution. It's more complicated than the mutation because we cannot avoid invalid solutions during the crossover so that the only thing we can do is to modify our invalid solutions to be valid. We first check whether we should do crossover according to the possibility provided. If crossover needs to occur, after crossover for some random section, we check whether duplications exist for each value in that section. If so, we replace the duplications with the values that are exchanged to make sure that no cities are missing. In the end, we return a pair of the modified solutions.

Take a 9-bit string as an example:

Before crossover:

string1:

1	2	3	4	5	6	7	8	9

string2:

9	8	7	6	5	4	3	2	1

After crossover:

string1:

1	2	3	4	5	4	3	2	9

Note: 4,3,2 are duplications and 6,7,8 are missing.

string2:

9	8	7	6	5	6	7	8	1

Note: 6,7,8 are duplications and 4,3,2 are missing.

After modification:

string1:

1	~~2~~8	~~3~~7	~~4~~6	5	4	3	2	9

string2:

9	~~8~~2	~~7~~3	~~6~~4	5	6	7	8	1

Order of operations

Now we have all the methods and parameters clarified and defined. The next step is to put them together in order. For each generation, we call reproduction to generate a new population. When we use reproduction method, we need to use fitness method to calculate the fitness for each solution in the population, then we can use the logic described above to generate a new generation. After reproduction, we have some new population. Then we randomly choose one solution from the population with some chance to mutate and randomly choose a pair from the population to do crossover and update the population again. The mutation and crossover will generate some new solutions other than the initial population of solutions with either high or low fitness to give us higher opportunity to find the best solution. Although solutions with low fitness is not what we want, these solutions will eventually extinct during reproduction as they have really low possibility to be reproduced. After we repeat these operations over and over again, we will encounter more and more different solutions compared to the original population and the average fitness of population will gradually increase. Because during these operations, the algorithm will automatically select high-fitness solutions and eliminate low-fitness solutions. Therefore, if we can repeat these operations for enough time, the solution that we are looking for, i.e the shortest route that visits all cities exactly once and return back to the start city, will eventually survive.

Appendix

For all the methods below, I would suppose that n, distance matrix and size are some private instances of the class so that I can use them directly in the class instead of calling string.length every time to get n or calling population.length to get population size.

All the helper functions are defined at the end of this section.

Constructor

Initialize the values of instances of the class which include the number of cities, the distance matrix and the population size.

```
constructor(n,distance,population_size)
    this.n ← n
    this.distance ← distance
    this.size ← population_size
```

Initialize the population

This initialize function is used to initialize the population, i.e, the initial solutions. When we get the first generation, we can then calculate the fitness for each solution and get percentage of total fitness for each solution to reproduce the next generation. This function will return the initial population.

```
initialize()
    //create an array of arrays,first index indicates which solution to
    //look and the array at that index indicates the solution
    population ← new int[1..size][1..n]
    for i ← 1 to size
        population[i] ← {1,2,3,...,n}
        //randomly shuffle the solution
        population[i] ← shuffle(population[i])
    end for
    return population
```

Fitness function

This function is used to calculate the fitness of a given solution.

```
fitness(string)
    total_distance ← 0
    for i ← 1 to n-1
        //get the distance between each cities
        current ← distance[string[i]][string[i+1]]
        //add it to the total distance
        total_distance ← total_distance + current
    end for
    //add the last distance that go from nth city back to the start city
    total_distance ← total_distance + distance[string[n]][string[1]]
    return 1/(double)total_distance
```

Reproduction

This function will take a population of generation and reproduce a new generation according to the fitness of each solution of the population provided. However, the cumulative sum of fitness might have some minor problems such that the total sum might not added up to 1000. I will just leave it as it is in this pseudo code but deal with the details when actually implementing this function.

```
reproduction(population)
    fitness_array ← new double[1..size]
    fitness_sum ← new int[1..size]
    total_fitness ← 0
    new_population ← new int[1..size][1..n]
    //calculate fitness for each string and find the total fitness
    for i ← 1 to size
        fit ← fitness(population[i])
        total_fitness ← total_fitness + fit
        fitness_array[i] ← fit
    end for
    //calculate the fraction of fitness for each string and find cumulative
    //sum of fitness
    for i ← 1 to size
        perc ← fitness_array[i]/(double)total_fitness
        fitness_sum[i] ← round(perc*1000)
    end for
    fitness_sum[0] ← 0
    for i ← 1 to size
```

```
        r ← random(1,1000)
        solution_idx ← find_solution(fitness_sum,r)
        new_population[i] ← population[solution_idx]
    end for
    return new_population
```

Mutation function

Note: This function only represents the mutation process for one string of the population. When using it in the main function, we will have another for loop to loop through all strings of the population and call this function each time.

This function takes a percentage of mutation to be occurred and the string that needs to be mutated and return the mutated string.

```
mutation(α, string)
    //α represents the chance of mutation occurring and it should be
    //a number between 0 and 1.
    r ← random(0,100) //get a random number between 0 and 100
    if r < α*100 then
        //loop through each bit of the string
        for i ← 1 to n
            //get one random position
            r1 ← random(1,n)
            p1 ← min(r1,i)
            p2 ← max(r1,i)
            //make sure that mutation actually happens
            if p1 = p2 and p2 = n then
                p1--
            else if p1 = p2 and p1 = 1 then
                p2++
            end if
            //swap the values in two positions
            temp ← string[p1]
            string[p1] ← string[p2]
            string[p2] ← temp
        end for
    end if
    return string
```

Crossover function

Note: This function only represents the crossover process for one string of the population. When using it in the main function, we will have another for loop to loop through all strings in the population, find a random string other than the current string and call crossover function for these two strings each time.

This function takes a possibility of crossover and two strings that are chosen to crossover and returns the two strings after crossover.

```
crossover(α, string1,string2)
    r ← random(0,100) //get a random number between 0 and 100
    if r < α*100 then
        //get two random positions
        r1 ← random(1,n)
        r2 ← random(1,n)
        p1 ← min(r1,r2)
        p2 ← max(r1,r2)
        //swap two sections
        temp ←string1[p1:p2]
        string1[p1:p2] ← string2[p1:p2]
        string2[p1:p2] ← temp
        for i ← p1 to p2
            //call helper function checkdup
            idx1 ← checkdup(string1,p1,p2,i)
            idx2 ← checkdup(string2,p1,p2,i)
            if idx1 > -1 then
                string1[idx1] ← string1[i]
            end if
            if idx2 > -1 then
                string2[idx2] ← string2[i]
            end if
        end for
    end if
    return (string1, string2)
```

Although the process of finding the corresponding index from the array can be inefficient when the number of cities is really large as the running time is O(n) and the worst case could be as large as $O(n^2)$ if we happen to get a really large section, this is the only way I can think of to remove all the duplications.

Helper functions

Checkdup function

This function is used to determine whether there is duplication of the section exchanged. If duplication exists, return the index of where the duplication occurs. Otherwise, return -1 indicating that there is no duplication.

```
checkdup(string,p1,p2,i)
    //return the index of duplicated value, otherwise return -1.
    for j ← 1 to p1
        if string[j] = string[i] then
            return j
        end if
    end for
    for k ← p2 to n
        if string[k] = string[i] then
            return k
        end if
    end for
    return -1
```

Shuffle function

This function is used to randomly shuffle the solution. It is used to initialize the population, i.e, generating some initial possible solutions.

```
shuffle(string)
    r ← random(1,n)
    for i ← i to n
        temp ←  string[i]
        string[i] ← string[r]
        string[r] ← temp
    end for
    return string
```

Find_solution function

This function is used when we try to reproduce the solution. The parameters are the cumulative fitness sum and the random number. The return value is which solution should be reproduced.

```
find_solution(fitness,r)
    for i ← 1 to size
        if r <= fitness[i] and r > fitness[i-1] then
            return i
        end if
    end for
```