Course: CMPUT366
Student Name: Dexter Dai
Date: Nov 28,2021
Prof: Levi Santana de Lelis

Project Report

Introduction:

The project will focus on solving an TSP problem by using genetic algorithm (GA). The requirements are as follows:

The traveling salesman has to visit n cities and eventually return to the departure city, requiring that each city be visited only once, with the optimization goal of minimizing the sum of the distances traveled.[1]

Since the model of the TSP problem can be applied to many areas. Such as the current take-out delivery worker industry, family outing schedule, Amazon, Canada Post's mailman industry and so on. All of these areas need to get the goods or travelers to the appropriate destination in the shortest possible time. The shorter the total distance traveled, the shorter the time spent if other factors are not taken into account.

Basic genetic algorithm (GA) is generally composed of two processes. The first process is selection of individuals for the production of the next generation and the second process is manipulation of the selected individuals to form the next generation by crossover and mutation techniques. The selection mechanism determines which individuals are chosen for mating (reproduction) and how many offspring each selected individual produces. The main principle of selection strategy is “the better is an individual; the higher is its chance of being parent.” Generally, crossover and mutation explore the search space, whereas selection reduces the search area within the population by discarding poor solutions. However, worst individuals should not be discarded and they have some chances to be selected because it may lead to useful genetic material. A good search technique must find a good trade-off between exploration and exploitation in order to find a global optimum [2].

Design and environment requirements:

This project involves tournament selection, sequential crossover, mutation, and one-to-one survivor competition.

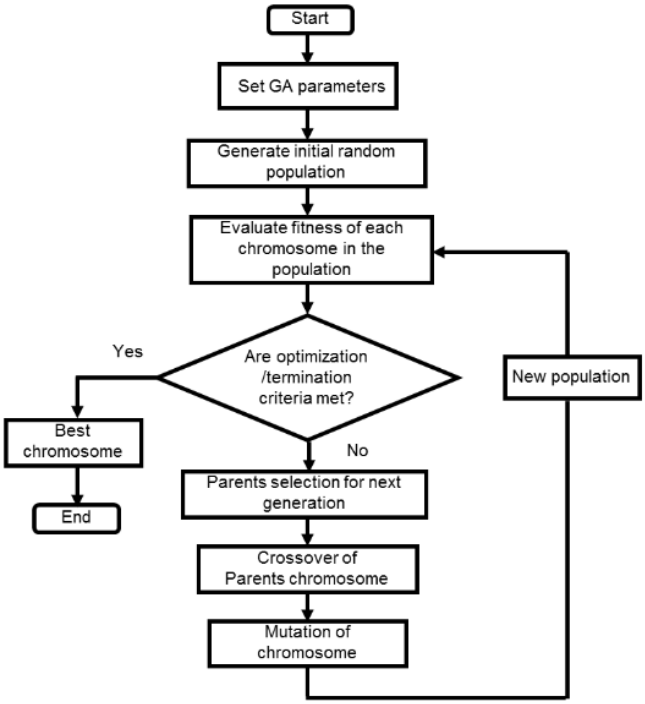
The runtime environment is going to use the pandas module, and matplotlib to generate the resulting images.

Methods:

This section provides the general overview of the genetic algorithm component and operation for solving TSP. Genetic algorithm is an optimization method that uses a stochastic approach to randomly search for good solutions to a specified problem. These stochastic approaches use various analogies of natural systems to build promising solutions, ensuring greater efficiency than completely random search. The basic principles of GA were first proposed by Holland in 1975 [3]. The GA operation is based on the Darwinian principle of „survival of the fittest“ and it implies that the fitter individuals are more likely to survive and have a greater chance of passing their good genetic features to the next generation.

The procedure for solving the TSP can be viewed as a process given in Figure 1. The GA

process starts with the provision of important information such as the location of the city, the maximum number of generations, the population size, the crossover probability and the mutation probability. An initial random population of chromosomes is elicited and the fitness of each chromosome is evaluated. This population is then transformed into a new population (next generation) using three genetic operators: selection, crossover, and mutation. The selection operator is used to select two parents from the current generation in order to produce a new child by crossover and/or mutation. The new generation contains a higher percentage of the traits possessed by the "good" members of the previous generation, and in this way the good traits spread through the population and mix with other good traits. After each generation, a new set of chromosomes evolves that is equal in size to the initial population size. This transformation process from one generation to the next continues until the population converges to an optimal solution, which usually occurs when a certain percentage of the population (e.g., 95%) has the same optimal chromosomes, with the best individuals being used as the optimal solution.



Therefore, the program will initialize several variables such as cities' coordinates (population) and so on. Here we set the probability of cross over is 95%, and the mutation probability is 10%.

The whole loop will iterate 100 times. The maximum value of the cities' coordinates is 100 and minimum is 0.

```
CityNum = 20
MinCoordinate = 0
MaxCoordinate = 101
#GA
generation = 100
popsize = 100
tournament_size = 5
pc = 0.95   #crossOver proba
pm = 0.1    #mutation proba

# set city size, we use a set of numbers for test, while we can generate a set of random city Coordinates.
# CityCoordinates = [(random.randint(MinCoordinate,MaxCoordinate),random.randint(MinCoordinate,MaxCoordinate)) for i in range(CityNum)]
CityCoordinates = [(88, 16),(42, 76),(5, 76),(69, 13),(73, 56),(100, 100),(22, 92),(48, 74),(73, 46),(39, 1),(51, 75),(92, 2),(101, 44),
dis_matrix = pd.DataFrame(data=None,columns=range(len(CityCoordinates)),index=range(len(CityCoordinates)))
for i in range(len(CityCoordinates)):
    xi,yi = CityCoordinates[i][0],CityCoordinates[i][1]
    for j in range(len(CityCoordinates)):
        xj,yj = CityCoordinates[j][0],CityCoordinates[j][1]
        dis_matrix.iloc[i,j] = round(math.sqrt((xi-xj)**2+(yi-yj)**2),2)  #generate a matrix with coordinate's distance.
```

Next, we will be evaluating the fitness value from the pops where we input the value of distance in the dis_matrix. Finally, the minimum value of fitness will be found, that is the best_fit, list as the first optimal solution (first shortest total distance)

```
iteration = 0
pops = [random.sample([i for i in list(range(len(CityCoordinates)))],len(CityCoordinates)) for j in range(popsize)]


fits = [None]*popsize
for i in range(popsize):
    fits[i] = calFitness(pops[i],dis_matrix)

best_fit = min(fits)
best_pop = pops[fits.index(best_fit)]
print('first opimal %.1f' % (best_fit))
best_fit_list = []
best_fit_list.append(best_fit)
```

Then the program will get into the most important part, that is the iteration part, also the most vital part of the whole Genetic Algorithm. The process goes as follows:

Tournament selection is probably the most popular selection method in genetic algorithms because of its efficiency and simplicity of implementation [3]. In tournament selection, n individuals are randomly selected from a large population, and the selected individuals compete with each other. The individual with the highest fitness wins and will be included in the next generation population. Then the individual selected will make a crossover of their chromosome for the next generation, the parents will also make a mutation with certain probability (10%). Then we will evaluate the fitness again to get the new generation (parents again) until the end of the loop.

```
while iteration <= generation:

    pop1,fits1 = tournament_select(pops,popsize,fits,tournament_size)
    pop2,fits2 = tournament_select(pops,popsize,fits,tournament_size)
    #cross over
    child_pops = crossover(popsize,pop1,pop2,pc)
    #mutation
    child_pops = mutate(child_pops,pm)
    #calculate fitness
    child_fits = [None]*popsize
    for i in range(popsize):
        child_fits[i] = calFitness(child_pops[i],dis_matrix)
    #keep the optimal
    for i in range(popsize):
        if fits[i] > child_fits[i]:
            fits[i] = child_fits[i]
            pops[i] = child_pops[i]

    if best_fit>min(fits):
        best_fit = min(fits)
        best_pop = pops[fits.index(best_fit)]

    best_fit_list.append(best_fit)

    print('the %d generation optimal %.1f' % (iteration, best_fit))
    iteration += 1
```

Next major function is crossover. The mating starts with the crossover operation according to the function below. This function accepts the parents and the offspring size. It uses the children's size to know the number of offspring to produce from such parents. The parents are selected in a way that is loop after loop, where pc is the probability of crossover, so if the random number generated is greater than the probability, the crossover will not be implemented, otherwise implement the crossover with a start and end point exchange of two parents.

```
def crossover(popsize,parent1_pops,parent2_pops,pc):
    child_pops = []
    for i in range(popsize):

        child = [None]*len(parent1_pops[i])
        parent1 = parent1_pops[i]
        parent2 = parent2_pops[i]
        if random.random() >= pc:
            child = parent1.copy()
            random.shuffle(child)
        else:
            start_pos = random.randint(0,len(parent1)-1)
            end_pos = random.randint(0,len(parent1)-1)
            if start_pos>end_pos:
                tem_pop = start_pos
                start_pos = end_pos
                end_pos = tem_pop
            child[start_pos:end_pos+1] = parent1[start_pos:end_pos+1].copy()
            # parent2 -> child
            list1 = list(range(end_pos+1,len(parent2)))
            list2 = list(range(0,start_pos))
            list_index = list1+list2
            j = -1
            for i in list_index:
                for j in range(j+1,len(parent2)):
                    if parent2[j] not in child:
                        child[i] = parent2[j]
                        break
        child_pops.append(child)
    return child_pops
```

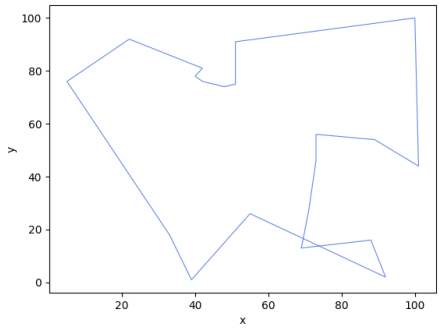
Next function is mutation function, that is the function accepts the crossover offspring and return them after mutation. The function parameter pm is the probability of mutation (10%). Therefore, every time the random number generated less than (0.1, that is really low) will perform a mutation with random integer change.

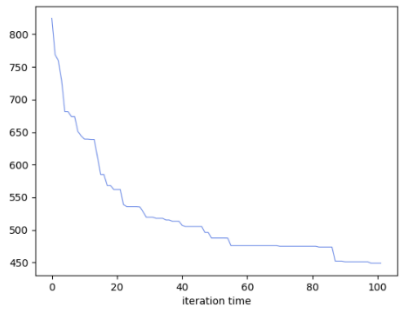
```
def mutate(pops,pm):
    pops_mutate = []
    for i in range(len(pops)):
        pop = pops[i].copy()
        t = random.randint(1,5)
        count = 0
        while count < t:
            if random.random() < pm:
                    mut_pos1 = random.randint(0,len(pop)-1)
                    mut_pos2 = random.randint(0,len(pop)-1)
                    if mut_pos1 != mut_pos2:
                        tem = pop[mut_pos1]
                        pop[mut_pos1] = pop[mut_pos2]
                        pop[mut_pos2] = tem
            pops_mutate.append(pop)
            count +=1
    return pops_mutate
```

Finally, the TSP problem will generate an image that contains all cities with corresponding coordinates. So that draw_path function will be implemented. Another line graph will show after 100 iterations, the final sum distance we find every time.

(All comments in python file are similar as this file explained)

Evaluation:





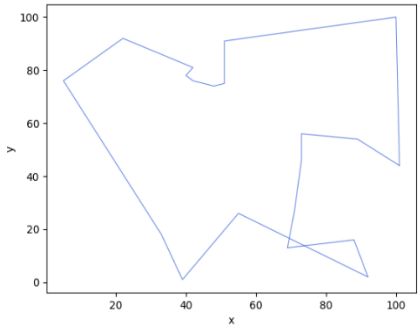
The image generated above are coordinates and the optimal line (dots generated each time of iteration).

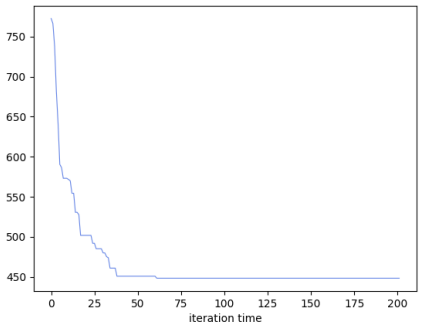
Since the problem asks for the optimal solution, therefore:

The first image shows how the salesman should travel in these 20 cities so that the final distance is optimal, the second image shows after 100 iterations, the "accuracy", that is the total distance traveled of each iteration. The second image indicates clearly that after each iteration, the final generation is much closer to the optimal solution.

The algorithmic processing of the TSP ensures that the approximation algorithm is not guaranteed to find the optimum, but it is the only technique that can find good solutions to large problem instances. In order to assess the quality of a solution, we must be able to compute a lower bound on the value of the shortest Hamiltonian cycle.

If double the generation time, that is increase the value "generation" to 200, the new graphs will be generated.





The result of the bottom line in second image is 448.3 which **is highly likely** to be (very close to) the final optimal solution since it never decreased during last 100 iteration.

From this result, another conclusion can be made, that is the more the generation time become, the more difficult to find a shorter distance (more difficult for a new generation to find the next optimal solution.

On average for this program only, the total number of for loop is 150-200 times as it contains random module of Python. Therefore, the more optimal generation will be lying in this for loop range.

Reference:

1. Razali, Noraini Mohd, and John Geraghty. "Genetic algorithm performance with different selection strategies in solving TSP." Proceedings of the world congress on engineering. Vol. 2. No. 1. Hong Kong: International Association of Engineers, 2011.
2. D. Beasley, D. Bull, and R. Martin, An Overview of genetic algorithms: Part 1, Fundamentals, University Computing, vol. 2, pp. 58-69, 1993.
3. J. H. Holland, Adaptation in natural and artificial systems, The University of Michigan press, 1975.