Assignment 6

I have demonstrated 2 methods: `simulated annealing` and `selection sort` algorithm.

Advantages of selection sort:

- In simulated annealing, `convergence` is not guarenteed. The same code when run again could still give a large value of distance
- It is `deterministic` unlike simulated annealing which is stochastic
- Simulated annealing requires careful `tuning of parameters` for a wide number of cases
- The simulated anneal gives quite an appropriate answer only at about n^3 runs while selection sort has a `complexity` of O(n^2) (It runs only for n-1(n-2)/2 times). Also in simulated annealing for every n^3 runs, calculating total distance would make total complexity is O(n^4).
  However, we might not reach global minima in selection sort, while properly tuned simulated annealing can `escape local optima` and find high-quality solutions especially for large values of n.

Thus for smaller number of cities selection sort is better, while simulated annealing is better in a wide variety of cases

Simulated annealing

The TSP optimization process begins with a randomly generated city order, where the total distance covered is computed. Subsequently, the `sim_anneal()` function is invoked, initially setting the temperature at 3 and the decay rate at 0.8.

During the simulated annealing process, each city in the finalorder is considered for swapping with another city to examine whether the total distance can be reduced. If the distance is reduced, the new order is accepted. However, if the distance does not decrease, it is accepted only if it varies slightly (by less than 0.5) from the current minimum. To introduce randomness into this decision-making process, a random sample is used, which makes the acceptance of a suboptimal solution a probabilistic event.

As the optimization progresses, the probability of accepting suboptimal solutions decreases as the temperature decays, ultimately converging towards an improved solution.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv
import timeit
```

`The distcost()` computes the distance between any 2 points in space

In [ ]:
```
def distcost(x1, y1, x2, y2):
    return np.sqrt((x1-x2)**2 + (y1-y2)**2)
```

`distance()` computes the total distance along the given order

In [ ]:
```
def distance(cities, cityorder):
    total_distance = 0
    N = len(cityorder)
    for i in range(N):
        current_city = cityorder[i]
        next_city = cityorder[(i + 1)%N]
        x1, y1 = cities[current_city]
```

```
        x2, y2 = cities[next_city]
        total_distance += distcost(x1, y1, x2, y2)
    return total_distance
```

`sim_anneal(cities,finalorder)` performs simulated annealing to return a `finalorder` along with its `bestcost`

In [ ]:
```
def sim_anneal(cities,finalorder):
    T = 3
    decayrate = 0.8
    x=T
    bestcost=distance(cities,finalorder)
    N = len(finalorder)
    for i in range(N):
        for j in range(N-1,-1,-1):
            if i != j:  # Exclude cases where i is equal to j
                iter = finalorder.copy()  # Make a copy of finalorder to avoid modi
                iter[i] = finalorder[j]
                iter[j] = finalorder[i]
                present = distance(cities,iter)
                T = T * decayrate
                if present < bestcost:
                    bestcost = present
                    finalorder = iter
                elif np.exp(-(present-bestcost)/T)>0.2*np.random.random_sample() an
                    bestcost = present
                    finalorder = iter
                if(T<x*decayrate**(N**3)):
                    break
    return finalorder,bestcost
```

For selection sort algorithm, we take the first city; find the nearest city to it and select it as second city. Then find next nearest to 2nd city and make it the third..

For this we use `find_nearest_city()` that takes current_city and unvisited_cities as its parameters and returns nearest_cityand min_distance. Then `tsp_selection_sort()` takes x,y coodinates of cities and keeps appending cities into tour an removing cities from unvisited_cities and thus finds an optimum order.

In [ ]:
```
# Function to find the nearest unvisited city to the current city
def find_nearest_city(current_city, unvisited_cities, x, y):
    min_distance = float('inf')
    nearest_city = None

    for city in unvisited_cities:
        distance = distcost(x[current_city], y[current_city], x[city], y[city])
        if distance < min_distance:
            min_distance = distance
            nearest_city = city

    return nearest_city, min_distance

def tsp_selection_sort(x, y):
    num_cities = len(x)
    unvisited_cities = list(range(1, num_cities))  # Start with all cities except t
    tour = [0]  # Initialize the tour with the first city (arbitrary starting point
    total_distance = 0.0

    for _ in range(num_cities - 1):
        nearest_city, distance = find_nearest_city(tour[-1], unvisited_cities, x, y
        tour.append(nearest_city)
        total_distance += distance
        unvisited_cities.remove(nearest_city)

    # Return to the starting city to complete the tour

    total_distance += distcost(x[tour[-2]], y[tour[-2]], x[0], y[0])

    return tour, total_distance
```

Plotting order of cities

In [ ]:
```
def plot(cities, cityorder):
    # Extract the city coordinates in the specified order
    xplot = [cities[i][0] for i in cityorder]
    yplot = [cities[i][1] for i in cityorder]

    # Close the loop for plotting by appending the coordinates of the first city at
    xplot.append(xplot[0])
    yplot.append(yplot[0])

    # Plot the cities
    plt.clf()
    plt.plot(xplot, yplot, 'o-')
    plt.show()
    print("Order:",cityorder)
```

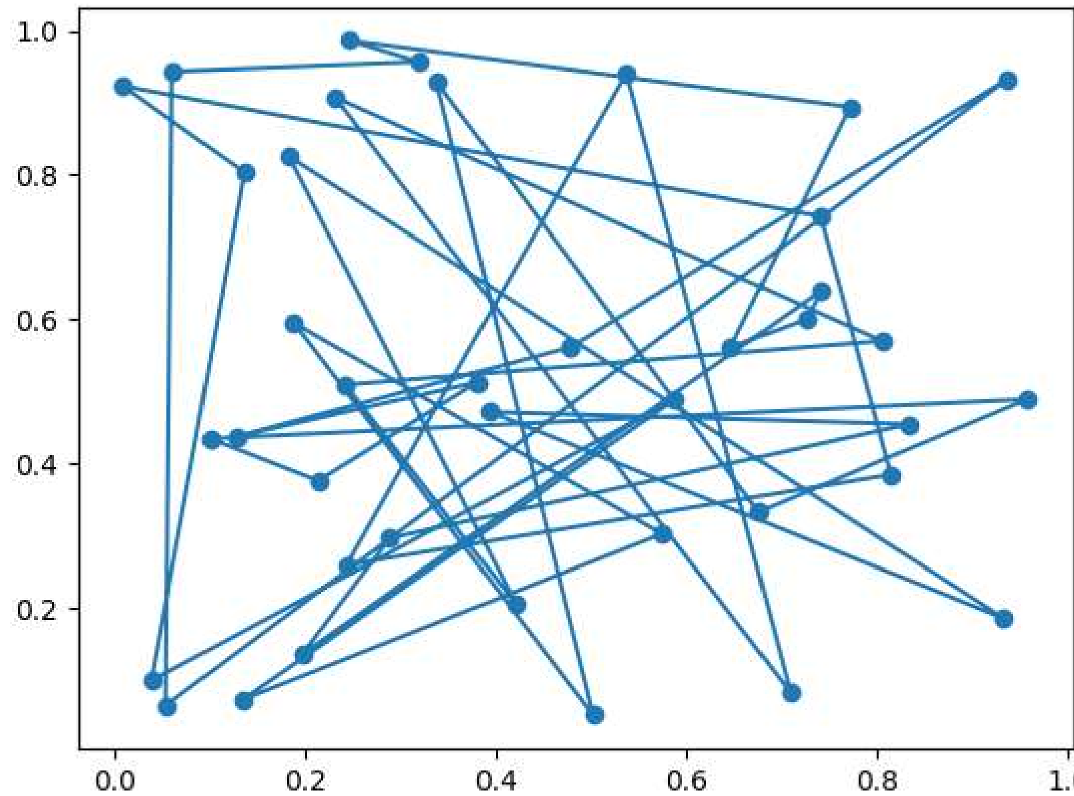
`tsp()` takes the optimum solution among simulated annealing and selection sort and returns the optimum order of cities. It plots the original order and distance and also the solutions after simulated annealing and selection sort.

In [ ]:
```
def tsp(cities):
    N = len(cities)
    x_cities = [cities[i][0] for i in range(N)]
    y_cities = [cities[i][1] for i in range(N)]

    cityorder = np.arange(N)
    np.random.shuffle(cityorder)

    finalorder,cost1 = sim_anneal(cities, cityorder)
    tour, total_distance = tsp_selection_sort(x_cities, y_cities)

    # Calculate the cost using the distance function
    cost = distance(cities, cityorder)
    print(f"Initial distance = {cost}")
    plot(cities,cityorder)
    plot(cities,finalorder)
    print("Distance after simulated annealing:\n",cost1)
    print("Percentage improvement after simulated annealing:\n",(cost-cost1)*100/co
    plot(cities,tour)
    print("Distance after selection sort:\n",total_distance)
    print("Percentage improvement after selection sort:\n",(cost-total_distance)*10
    if(cost1<total_distance ):
        return finalorder
    return tour
```

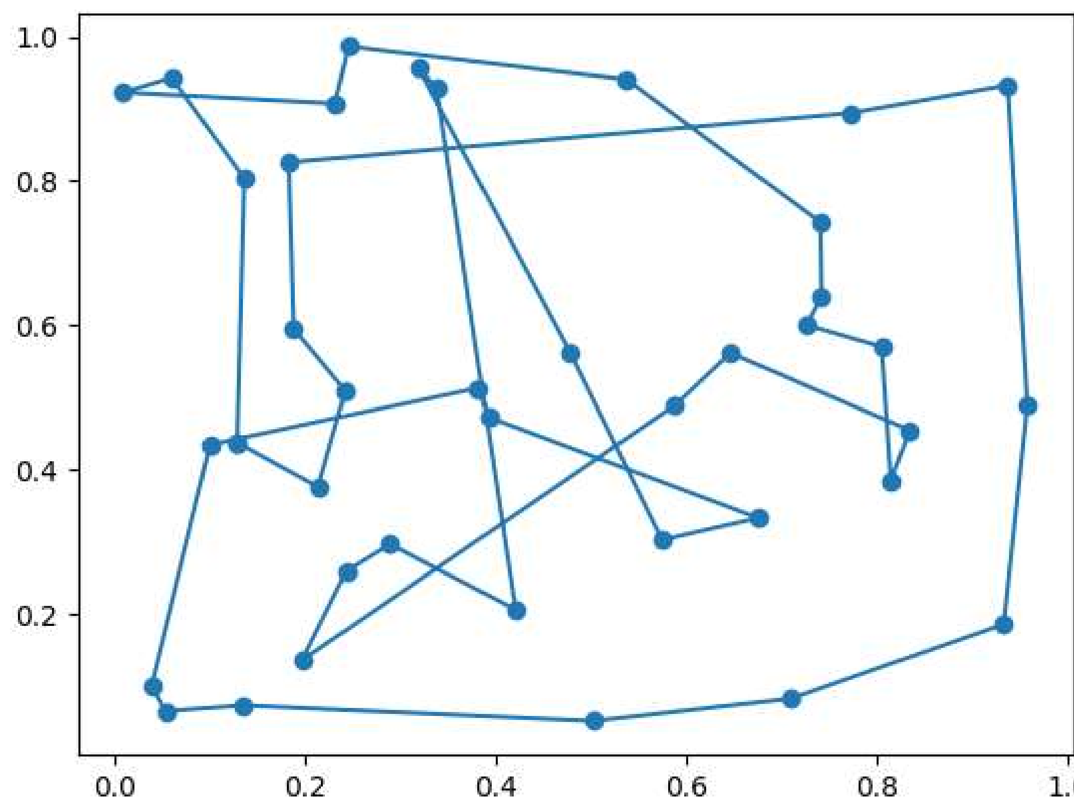
Reading the csv file with first line having number of cities and subsequent lines having (x,y) co-ordinates of cities

In [ ]:
```
with open("C:\\Users\\shrip\\Downloads\\tsp40.txt", 'r') as f:
    reader = csv.reader(f, delimiter=' ')
    cities = []

    # Read city coordinates from the CSV file
    for i, row in enumerate(reader):
        if i == 0:
            # Define the number of cities
            N = int(row[0])
        elif i <= N:
            # Append city coordinates as a tuple to the 'cities' list
            cities.append((float(row[0]), float(row[1])))
def time():
    order=tsp(cities)
timeit.timeit(time,number=1)
```

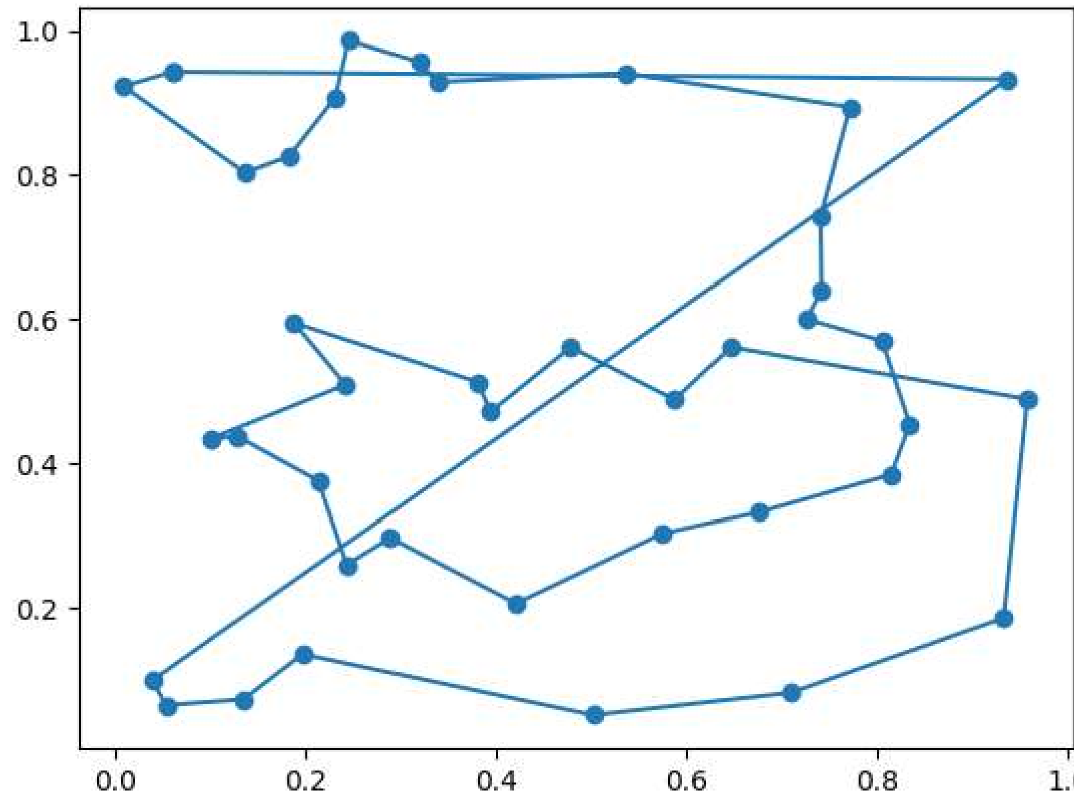
```
Initial distance = 21.64890450647002
```




```
Order: [36 15 16 14 23 33 26  4 19 39 21 38  6 22 28  9 35 18  3 32 20 29 13 25
 17 37 30  1 24 11 34  8 27 31  7  5 12  0 10  2]
```




```
Order: [15 14 25 39 37  7  2 33 30 20 19 38 10 22 23 16  1 26 21 36 12  4 17 11
  3 34  6 31 24 18 13 27  8 35 32  5 29  9  0 28]
Distance after simulated annealing:
 8.727854112276017
Percentage improvement after simulated annealing:
 59.684546117945146
```




```
Order: [0, 9, 28, 37, 29, 5, 12, 4, 32, 7, 35, 8, 27, 13, 24, 18, 26, 21, 17, 11, 3,
14, 15, 23, 25, 39, 16, 1, 36, 6, 31, 33, 30, 20, 19, 34, 38, 10, 22, 2]
Distance after selection sort:
 7.031069374336417
Percentage improvement after selection sort:
 67.52228560925518
```

Out[ ]: 0.9444026000001031

Time taken= 6.2497476 s