Assignment1

May 12, 2021

1 IN3050/IN4050 Mandatory Assignment 1: Traveling Salesperson Problem

Name: Rohullah Akbari

1.1 Introduction

In this exercise, you will attempt to solve an instance of the traveling salesman problem (TSP) using different methods. The goal is to become familiar with evolutionary algorithms and to appreciate their effectiveness on a difficult search problem. You may use whichever programming language you like, but we strongly suggest that you try to use Python, since you will be required to write the second assignment in Python. You must write your program from scratch (but you may use non-EA-related libraries).

	Barcelona	Belgrade	Berlin	Brussels	Bucharest	Budapest
Barcelona	0	1528.13	1497.61	1062.89	1968.42	1498.79
Belgrade	1528.13	0	999.25	1372.59	447.34	316.41
Berlin	1497.61	999.25	0	651.62	1293.40	1293.40
Brussels	1062.89	1372.59	651.62	0	1769.69	1131.52
Bucharest	1968.42	447.34	1293.40	1769.69	0	639.77
Budapest	1498.79	316.41	1293.40	1131.52	639.77	0

Figure 1: First 6 cities from csv file.

1.2 Problem

The traveling salesperson, wishing to disturb the residents of the major cities in some region of the world in the shortest time possible, is faced with the problem of finding the shortest tour among the cities. A tour is a path that starts in one city, visits all of the other cities, and then returns to the starting point. The relevant pieces of information, then, are the cities and the distances between them. In this instance of the TSP, a number of European cities are to be visited. Their relative distances are given in the data file, *european_cities.csv*, found in the zip file with the mandatory assignment.

(You will use permutations to represent tours in your programs. If you use Python, the **itertools** module provides a permutations function that returns successive permutations, this is useful for exhaustive search)

1.3 Exhaustive Search

First, try to solve the problem by inspecting every possible tour. Start by writing a program to find the shortest tour among a subset of the cities (say, **6** of them). Measure the amount of time your program takes. Incrementally add more cities and observe how the time increases.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import itertools as it
import random as rd
import time
import csv
```

[11]:
```
data = pd.read_csv("european_cities.csv",sep=';').values.tolist()

def totalDistance(cityData,solution):
    #finds the totaldistance for a given solution.
    distance = 0
    for i in range(len(solution)):
        distance += cityData[solution[i-1]][solution[i]]
    return distance

def shortestPath(cityData,cityNum):
    bestDistance = 100000000
    bestSequence = None
    #using itertool to make permutations
    for perm in it.permutations(cityNum):
        #calculating the totalDistance for each permutations
        currentDistance = totalDistance(cityData,perm)
        #checking if the currentDistance is better than best, if thats
        #the case then it changes
        if currentDistance < bestDistance:
            bestDistance = currentDistance
            bestSequence = perm
    return bestDistance,perm
```

What is the shortest tour (i.e., the actual sequence of cities, and its length) among the first 10 cities (that is, the cities starting with B,C,D,H and I)? How long did your program take to find it? Calculate an approximation of how long it would take to perform exhaustive search on all 24 cities?

[12]:
```
cityNum = range(10)
t1 = time.time()
best_distance,best_sequece = shortestPath(data,cityNum)
t2 = time.time()
finalTime = t2-t1
```

```
print(f"Best solution: {best_distance:.3f}")
print(f"The actual sequence of the citites: {best_sequece}")
print(f"Best time: {finalTime:.2f} s")
```

```
Best solution: 7486.310
The actual sequence of the citites: (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
Best time: 5.92 s
```

By looking at the function above, we can estimate the Big-O notation. In this case we have that the first loop does N! primitive steps and the second loop runs N times. This gives us that O-notation is $N! \cdot N$ for this function. We can use this to estimate the time for 24 cities. When I run this algorithm on my Mac, I get around 6 seconds for N = 10. So it gives: $\frac{6.00}{10*10!}$= 1.6710^{-07} second/steps. By multiplying it with N!* N = 24! * 24 will gives the estimate for time the program will take to run:

$$\frac{6.00}{10*10!} * 24! * 24 = 2482614305877049344 = 2.48 \cdot 10^{19}$$

years.

1.4 Hill Climbing

Then, write a simple hill climber to solve the TSP. How well does the hill climber perform, compared to the result from the exhaustive search for the first **10 cities**? Since you are dealing with a stochastic algorithm, you should run the algorithm several times to measure its performance. Report the length of the tour of the best, worst and mean of 20 runs (with random starting tours), as well as the standard deviation of the runs, both with the **10 first cities**, and with all **24 cities**.

[47]:
```
def randomSolution(cityData):
    #this function selects a random city
    cities = list(range(len(cityData)))
    solution = []

    for i in range(len(cityData)):
        #selecting a city randomly using randint
        #adding it in the list of solution
        #making sure to not select it again by removing it from cities
        randomCity = cities[rd.randint(0,len(cities)-1)]
        solution.append(randomCity)
        cities.remove(randomCity)

    return solution

def getNeighbors(solution):
    #this function returns all neighbors of the given solution
    #by recombination/swaping elements of the solution listl
    AllNeighbors = []
    for i in range(len(solution)):
        for j in range(i+1,len(solution)):
            neighbor = solution.copy()
            neighbor[i] = solution[j]
            neighbor[j] = solution[i]
            AllNeighbors.append(neighbor)
    return AllNeighbors


def getBestNeighbor(cityData,AllNeighbors):
    #initialising the current best distances and neighbors
    #this function finds the best neighbor (meaning neighbor with shortest␣
 ↪distance)
    #by searching all of the neighbors
    bestDistance = totalDistance(cityData, AllNeighbors[0])
    bestNeighbor = AllNeighbors[0]
    for neighbor in AllNeighbors:
        currentDistance = totalDistance(cityData,neighbor)
        #checks if the currentDistance is better than the best, if so
        #it changes the best
        if currentDistance < bestDistance:
            bestDistance = currentDistance
            bestNeighbor = neighbor
    return bestNeighbor,bestDistance


def hillClimbing(cityData):
    #calling on the functions above to get the variables
    currentSolution = randomSolution(cityData)
    currentDistance = totalDistance(cityData,currentSolution)
    AllNeighbors = getNeighbors(currentSolution)
    bestNeighbor,bestDistance = getBestNeighbor(cityData,AllNeighbors)
    #checks if the currentDistance is better than the best, if so
    #it changes the best
    while bestDistance < currentDistance:
        currentSolution = bestNeighbor
        currentDistance = bestDistance
        AllNeighbors = getNeighbors(currentSolution)
        bestNeighbor,bestDistance = getBestNeighbor(cityData,AllNeighbors)

    return currentSolution,currentDistance
```

The hill climber performed very well compared to exhaustive search. We got almost the same distance. Hill climbing is a stochastic algorithm, so we had to run the algorithm several times to measure its performance. Although we ended up with the same result for the distance, but the sequel of the cities differs from the exhaustive search.

[48]:
```
amountRuns = 20
listFirstTen = []
```

```
listAllCities = []
for i in range(amountRuns):
    distanceForTen = hillClimbing(data[0:10])[1]
    listFirstTen.append(distanceForTen)

    distanceAll = hillClimbing(data)[1]
    listAllCities.append(distanceAll)

def printing(string,liste):
    print(f"Hill climbing: 20 runs of the {string}")
    print(f"Best length of the tour: {min(liste):.3f}")
    print(f"Worst length of the tour: {max(liste):.3f}")
    print(f"Mean length of the tour: {np.mean(liste):.3f}")
    print(f"Standard deviation of the runs: {np.std(liste):.3f}")
    print("___________________________________________")

printing("first ten cities",listFirstTen)
printing("all cities",listAllCities)
```

```
Hill climbing: 20 runs of the first ten cities
Best length of the tour: 7486.310
Worst length of the tour: 7737.950
Mean length of the tour: 7564.320
Standard deviation of the runs: 113.813
___________________________________________
Hill climbing: 20 runs of the all cities
Best length of the tour: 12504.080
Worst length of the tour: 15528.340
Mean length of the tour: 14368.179
Standard deviation of the runs: 855.119
___________________________________________
```

1.5 Genetic Algorithm

Next, write a genetic algorithm (GA) to solve the problem. Choose mutation and crossover operators that are appropriate for the problem (see chapter 4.5 of the Eiben and Smith textbook). Choose three different values for the population size. Define and tune other parameters yourself and make assumptions as necessary (and report them, of course).

For all three variants: As with the hill climber, report best, worst, mean and standard deviation of tour length out of 20 runs of the algorithm (of the best individual of last generation). Also, find and plot the average fitness of the best fit individual in each generation (average across runs), and include a figure with all three curves in the same plot in the report. Conclude which is best in terms of tour length and number of generations of evolution time.

1.6 The procedure goes as following:

- Initializing and randomly choosing a population while calculating the distance and fitness
- Selection parents: selecting parents with a probabilty proportinal to their fitness scores

- Crossover: generating new children from the parents
- Mutation: mutates the new children
- Selection: sorting and selecting those with highest fitness
- Evaluation: checking if it has been found a solution better than the previous one. If so, the programe terminates, otherwise it starts from the selecting new parents.

[4]:
```python
def calulateDistance(data,combination):
    #calculates the distance btween cities and stores it
    finalDistance = 0
    for i in range(len(combination) - 1):
        fromCity = data[0].index(combination[i])
        toCity = data[0].index(combination[i + 1])
        distances = data[1:]
        distance = float(distances[fromCity][toCity])
        finalDistance += distance

    lastCity = data[0].index(combination[-1])
    firstCity = data[0].index(combination[0])
    lastDistance = float(data[1:][lastCity][firstCity])
    finalDistance += lastDistance
    return finalDistance


def calculateFitness(fitnessOfTour,population,distanceOfTour):
    #calculates fitness and stores it
    for i in range(len(population)):
        fitness = np.sum(distanceOfTour)/distanceOfTour[i]
        fitnessOfTour.append(fitness)
    return fitnessOfTour

def makeAPopulation(data,tourOfCities,populationSize):
    #creating a population by using the random.shuffle()
    distanceOfTour = []
    population = []
    fitnessOfTour = []
    for i in range(populationSize):
        tour = tourOfCities.copy()
        rd.shuffle(tour)

        population.append(tour)
        distanceOfTour.append(calulateDistance(data,tour))

    fitnessOfTour = calculateFitness(fitnessOfTour,population,distanceOfTour)

    return population,distanceOfTour,fitnessOfTour
```

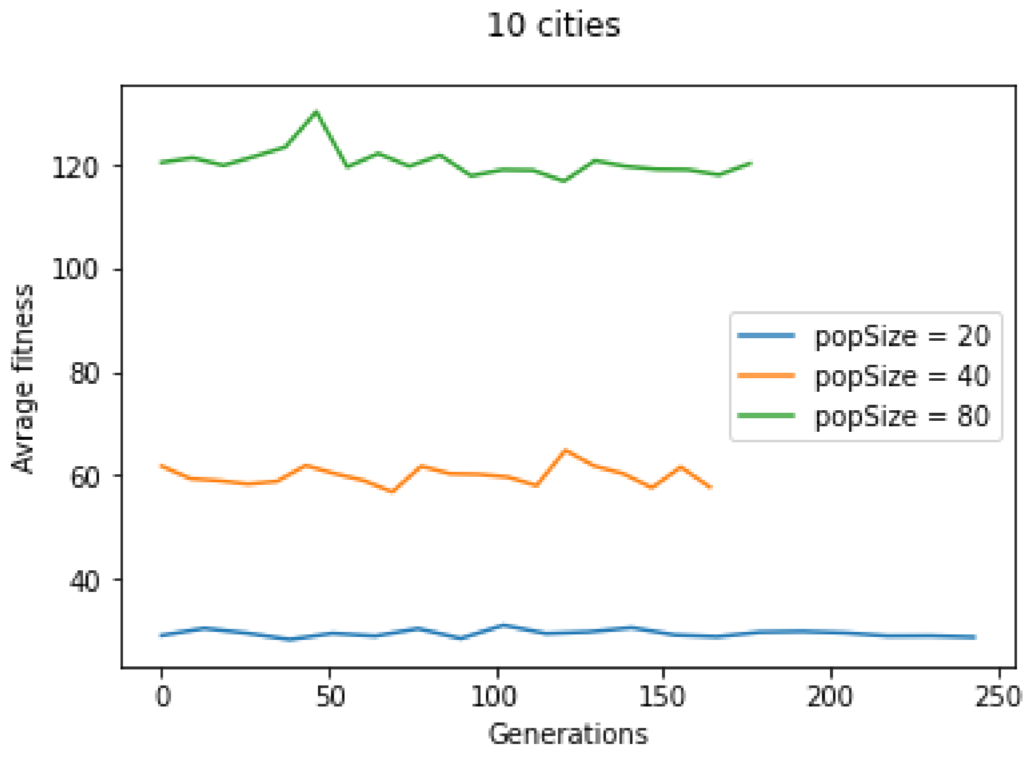
[17]:
```
#Selection of parents
def parentSelection(population, fitnessOfTour):
    #defining the selectedParent as None
    selectedParent = None
    indeks = rd.randint(0, len(population) - 1)
    sumFitness = np.sum(fitnessOfTour)

    #loops until we find a parent
    while selectedParent == None:
        if indeks == len(population):
            indeks = 0
        #selecting the parent with highest fitness
        probability = fitnessOfTour[indeks]/sumFitness
        if probability > rd.uniform(0,1):
            selectedParent = population[indeks]
        indeks += 1

    return selectedParent

#Selection of the survivor:
def survivorSelection(population, populationSize, fitnessOfTour,distanceOfTour):
    fullPopulation = []
    #adding up population,fitness and distance in one list in order to make it␣
 ↪easier for sorting
    for i in range(len(population)):
        fullPopulation.
 ↪append([population[i],distanceOfTour[i],fitnessOfTour[i]])
    #sorting and revoming those with the worst fitness
    fullPopulation.sort(key=lambda x: x[2], reverse=True)
    while len(fullPopulation) > populationSize:
        fullPopulation.pop()

    population = [row[0] for row in fullPopulation]
    distanceOfTour = [row[1] for row in fullPopulation]
    fitnessOfTour = [row[2] for row in fullPopulation]

    return population,distanceOfTour,fitnessOfTour

#partially-mapped crossover
def pmx(a, b, start, stop):
    child = [None]*len(a)
    child[start:stop] = a[start:stop]
    for ind, x in enumerate(b[start:stop]):
        ind += start
        if x not in child:
            while child[ind] != None:
                ind = b.index(a[ind])
            child[ind] = x
    for ind, x in enumerate(child):
        if x == None:
            child[ind] = b[ind]

    return child

def pmxPair(a,b):
    half = len(a) // 2
    start = np.random.randint(0, len(a)-half)
    stop = start+half
    return pmx(a, b, start, stop), pmx(b, a, start, stop)

#mutation
def swapMutation(child):
    cities = np.random.choice(len(child), 2, replace=False)
    child[cities[0]], child[cities[1]] = child[cities[1]], child[cities[0]]
    return child


#this funciton generate parents and childs by mutation and crossover
def nextGeneration(population,fitnessOfTour,data,distanceOfTour):
    populationSize = len(population)
    listOffspring = []
    distOffspring = []
    fitOffspring = []

    for i in range(len(population)//2):
        #calling on and creating parents
        parent1 = parentSelection(population, fitnessOfTour)
        parent2 = parentSelection(population, fitnessOfTour)

        #mutation and crossover
        child1, child2 = pmxPair(parent1,parent2)
        child1 = swapMutation(child1)
        child2 = swapMutation(child2)

        #adding the children into the lists of offspring
        listOffspring.append(child1)
        distOffspring.append(calulateDistance(data,child1))
        fitOffspring.append(0)

        listOffspring.append(child2)
        distOffspring.append(calulateDistance(data,child2))
        fitOffspring.append(0)

    for i in range(len(listOffspring)):
        fitOffspring[i] = np.sum(distOffspring)/distOffspring[i]

    #adding offspring to the population
    population = population + listOffspring
    distanceOfTour = distanceOfTour + distOffspring
    fitnessOfTour = fitnessOfTour + fitOffspring

    return survivorSelection(population, populationSize,␣
 ↪fitnessOfTour,distanceOfTour)


def evaluation(population,fitnessOfTour,data,distanceOfTour):
    bestDist = distanceOfTour
    bestRout = population
    bestFitness = fitnessOfTour

    generation = 0
    counter = 0

    #using while loop to find the best solution by running it many times, if it␣
 ↪runs without
    #improving the programe will terminate
    while counter < 100:
        population,distanceOfTour,fitnessOfTour =␣
 ↪nextGeneration(population,fitnessOfTour,data,distanceOfTour)
        generation += 1

        if distanceOfTour[0] < bestDist[0]:
            bestDist = distanceOfTour
            bestRout = population
            bestFitness = fitnessOfTour
            counter = 0
        else:
            counter += 1
    return bestRout,bestDist,bestFitness,generation
```

[16]:
```
def runningGA(populationSize,numberOfCities,data):
    t1 = time.time()
    tourOfCities = data[0][:numberOfCities]
    population,distanceOfTour,fitnessOfTour =␣
 ↪makeAPopulation(data,tourOfCities,populationSize)

    bestSolution = []
    runs = []
    for i in range(20):
        bestRout,bestDist,bestFitness,generation =␣
 ↪evaluation(population,fitnessOfTour,data,distanceOfTour)
```

```
        bestSolution.append([bestRout[0],bestDist[0],bestFitness[0]])
        runs.append(generation)

    t2 = time.time()

    bestSolution.sort(key=lambda x: x[1], reverse=False)
    avrageFitness = [x[2] for x in bestSolution]
    gen = np.linspace(0,max(runs),20)
    plt.plot(gen,avrageFitness,label = f"popSize = {populationSize}")

    print(f'GA for {numberOfCities} cities with populationSize␣
 ↪={populationSize}:')
    print(f'Shortest distance: {bestSolution[0][1]:.3f}')
    print(f'Mean distance: {np.mean([x[1] for x in bestSolution]):.3f}')
    print(f'Standard deviation: {np.std([x[1] for x in bestSolution]):.3f}')
    print(f'Worst distance: {bestSolution[-1][1]:.3f}')
    finalTime = t2-t1
    print(f"Time: {finalTime:.3f} s")
    print("___________________________________________")

def printing(popSizes,numCities):
    data = list(csv.reader(open("european_cities.csv", "r"), delimiter=";"))
    for p in popSizes:
        runningGA(p,numCities,data)

    plt.suptitle(f"{numCities} cities")
    plt.xlabel("Generations")
    plt.ylabel("Avrage fitness")
    plt.legend()
    plt.show()

printing([20,40,80],10)
printing([20,40,80],24)
```

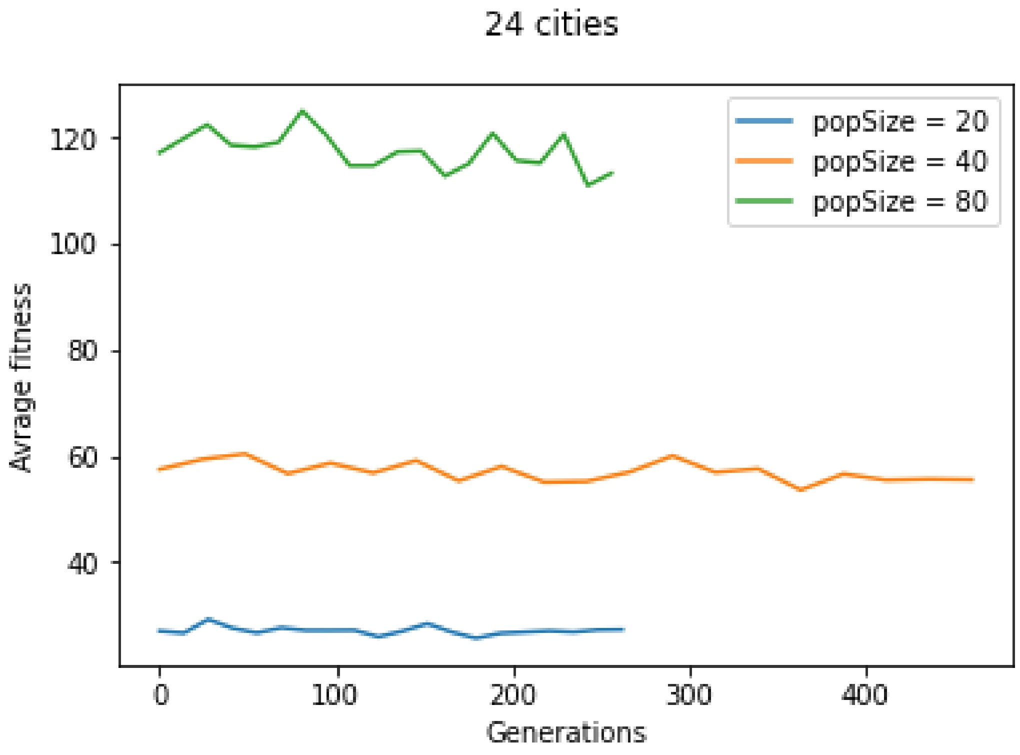
```
GA for 10 cities with populationSize =20:
Shortest distance: 7486.310
Mean distance: 7937.769
Standard deviation: 356.960
Worst distance: 8744.900
Time: 3.523 s
___________________________________________
GA for 10 cities with populationSize =40:
Shortest distance: 7486.310
Mean distance: 7730.195
Standard deviation: 226.066
Worst distance: 8309.610
Time: 7.074 s
```

```
-------------------------------------------
GA for 10 cities with populationSize =80:
Shortest distance: 7486.310
Mean distance: 7729.057
Standard deviation: 162.081
Worst distance: 8003.960
Time: 18.898 s
-------------------------------------------
```



```
GA for 24 cities with populationSize =20:
Shortest distance: 15792.350
Mean distance: 17798.408
Standard deviation: 1354.247
Worst distance: 20856.140
Time: 5.777 s
-------------------------------------------
GA for 24 cities with populationSize =40:
Shortest distance: 13324.900
Mean distance: 16380.903
Standard deviation: 2100.358
Worst distance: 22381.750
Time: 15.641 s
-------------------------------------------
```

```
GA for 24 cities with populationSize =80:
Shortest distance: 13339.810
Mean distance: 15988.198
Standard deviation: 1767.268
Worst distance: 19201.320
Time: 36.208 s
-------------------------------------------
```



1.6.1 The plots

The plot above is showing the avrage fitness per generations. As it can be seen, the different graphs shows fitness for the given population sizes which are 20,40 and 80. Generally, the avrage fitness increase with the population size. In the plot of the ten first cities, we can see that the avrage fitness is quite stable at population size 20. In addition to this, we can also see that the standard deviation drops from 356.960 to 226.066 by changing the population size, and further decreases to 162.081 at population size 80. This clearly tells us that the best choice of population size is around 80 and it is the most stable one.

The interesting thing is that we get a different situation in the plot for 24 cities. Here we can see that the standard deviation increases with increasing population size from 20 to 40, but it drops again at population size 80. This can again be seen in the plot. Note that for 24 cities, the code struggles to find the very shortest distance between the cities, but it is getting close. Unfortunately, I was unable to fix this issue.

Among the first 10 cities, did your GA find the shortest tour (as found by the exhaustive search)? Did it come close?

For both 10 and 24 cities: How did the running time of your GA compare to that of the exhaustive search?

How many tours were inspected by your GA as compared to by the exhaustive search?

1.6.2 Answers:

We see from above that the shortest length was found to be 7486.310 which is the exactly same length as found with exhaustive search. This solution seems to be solid, since the very same distance was found by all different population sizes.

However, the running time for the genetic algorithm is different compared to exhaustive search. We can see that the running time increases with increased population sizes which seems logical. The time for lowest population sizes (20) seems to be a bit faster compared to the exhaustive search, the time we get for 10 cities is around 3.6 seconds and the time for 20 cities is around 5.7 seconds. So the GA is using almost the same to find the shortest distance between all of the cities while the exhaustive search only find for 10 cities. But other than that, the time is greater for the other population sizes and number of cities.

We can calculate the amount of tours completed by the genetic algorithm and the exhaustive search by following:

$$GATours = (populationSize) \cdot (generationCount) \cdot (runs)$$
$$ExhaustiveTours = (numberOfCities)!$$

For the GA, I used population sizes 20, 40, 80 and run the program 20 times. The amount of avrage generation count after this runs was 130. This gives us:

$$GATours = (140) \cdot (130) \cdot (20) = 364000$$
$$ExhaustiveTours = (numberOfCities)! = 10! = 3628800$$

Note that these numbers are not entirely accurate considering that the program can create duplicate children and parents, but may seem like a good approximation.