T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE2246

Analysis of Algorithms

HW-2 Report

Ahmet Kerem Akpınar - 150119842

Ceyhun Erdönmez - 150120851

Murat Ünsal - 150121516

Ali Yetim - 150119803

Introduction

We used a clustering algorithm for Half Traveling Salesman Problem. The algorithm utilizes the K-means clustering technique to identify clusters of cities based on their geographical coordinates. The goal is to visit a representative sample of cities while minimizing the total distance traveled.

Algorithm Overview

1. **Input Reading:** The script reads input data from a file specified by sys.argv[1]. Each line in the file represents a city with its unique ID, X-coordinate, and Y-coordinate.

2. **K-means Clustering:** The script applies the K-means clustering algorithm using the KMeans class from the sklearn.cluster module. It specifies n_clusters=2 to divide the cities into two clusters. The algorithm aims to group cities together based on their proximity in the two-dimensional coordinate space. **The reason behind we choose n_cluster = 2 is, our goal is to visit half of the city. N_cluster = 2 allows us to split our data into two.**

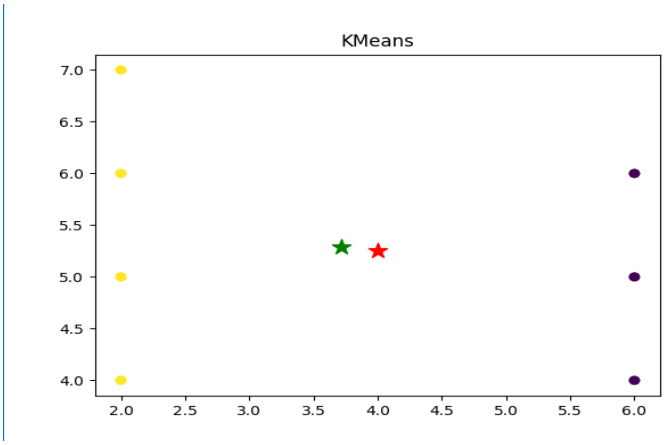
3. **Weighted Midpoint of 2 Cluster:** The script calculates the weights of the clusters based on the number of points they contain. The weight of a cluster is determined by dividing the number of cities in the cluster by the total number of cities. This weighting scheme considers the relative importance of each cluster in the subsequent steps. With this approach we are The code below.

```
#number of points in each cluster
clusterCount =  np.bincount(labels)

# Weights of clusters according to points that they have
cluster1weight = clusterCount[0]/(clusterCount[0]+clusterCount[1])
cluster2weight = 1-cluster1weight

finalX2 = cluster1weight*coordinatesOfClusterCenters[0][0] + cluster2weight*coordinatesOfClusterCenters[1][0]
finalY2 = cluster1weight*coordinatesOfClusterCenters[0][1] + cluster2weight*coordinatesOfClusterCenters[1][1]

# A diagram that shows the dataset and centers of clusters (with weights) *OPTIONAL*
plt.scatter(finalX2, finalY2, marker='*', s=150, c='g')
plt.scatter(cityX, cityY, c=kmeans.labels_)
plt.title('KMeans')
```



Red Star: Non-Weighted Midpoint

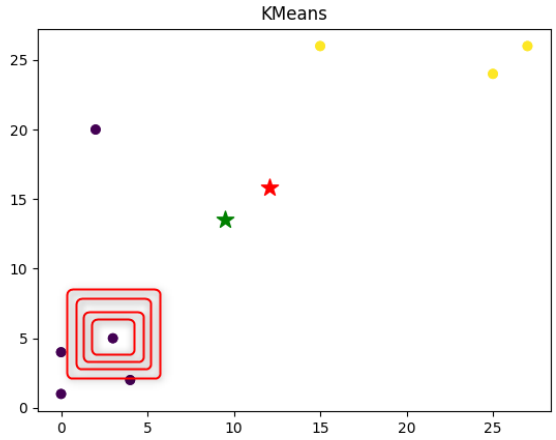
Green Star: Weighted Midpoint

Thanks to this approach, we choose our starting point from the densely populated areas, which is of great benefit to us in reducing the total distance traveled.

4. **Setting a Starting Point and Deciding for Next City:** The main algorithm iteratively selects cities to visit. It starts by expanding a square region around the weighted center (Green Star) of the clusters. The size of the square region increases with each iteration. The algorithm checks if any cities lie within the region and selects the first available city. **This is The Nearest Neighbor Algorithm as an Idea but we are not calculating every point and picking the nearest one. We are using a square method to pick the nearest one.** Therefore, we don't have to visit every city and pick the closest one. This means huge runtime.

```
while(visitedCount< numberOfCities/2):
    bottom_left = (nextPoint[0]-increment, nextPoint[1]-increment)
    top_right = (nextPoint[0]+increment, nextPoint[1]+increment)

    for i in data:
        if(checkIfInSquare(bottom_left,top_right,i)):
            pointsInSquare.append(i)
    if len(pointsInSquare) == 0:
        increment+= 1
        continue
    else:
        indexOfClosestOne = solve(pointsInSquare,nextPoint)
        point = pointsInSquare[indexOfClosestOne]
        value = [point[0],point[1]]
        listToWriteOntoOutput.append(str(get_key(value)))
        nextPoint = value
        data.remove(point)
        visitedCount+=1
        visitedCities.append(value)
        pointsInSquare.remove(point)
        increment = 0.1
```



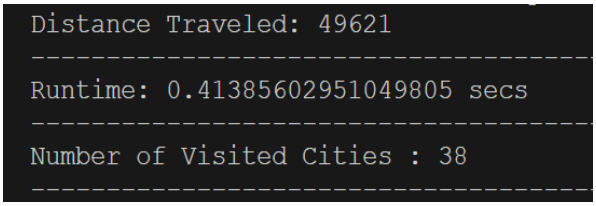
5. **Distance Calculation:** The script calculates the total distance traveled by summing the distances between consecutive visited cities. And turn back to origin city. The distance between two cities is computed using the Euclidean distance formula.

6. **Output Generation:** The script generates an output file specified by sys.argv[2]. The output file contains the distance traveled and the IDs of the visited cities in the order they were visited.

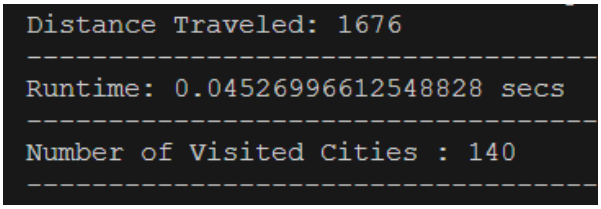
Results and Performance

Test results for given inputs:

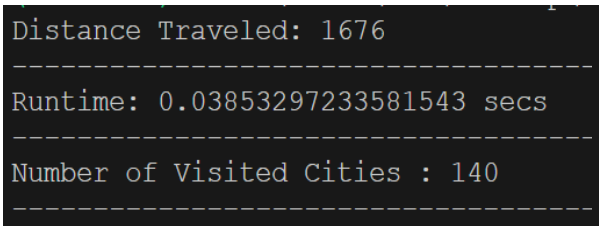
Example Input-1:



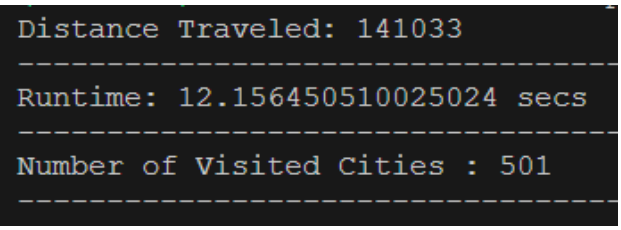
Test-Input-1:



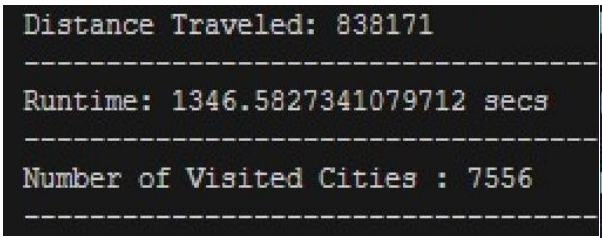
Example Input-2:



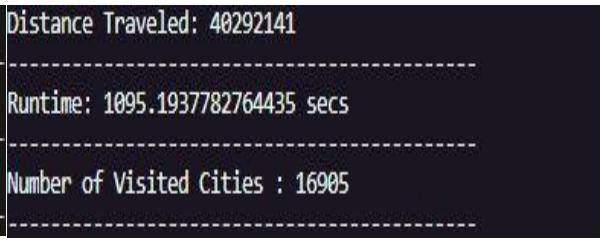
Test-Input-2:



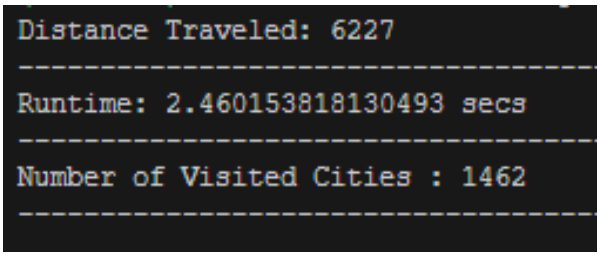
Example Input-3:



Test-Input-3:



Test-Input-4:



The algorithm's runtime and performance depend on the number of cities and the distribution of the data. Larger datasets or non-uniformly distributed cities might affect the runtime and the quality of the selected subset.

Division Of Labor:

Ahmet Kerem Akpınar – Main Algorithm, Input/Output Operations

Ceyhun Erdönmez – Main Algorithm, Input/Output Operations

Murat Ünsal – Main Algorithm, Cluster Operations

Ali Yetim – Main Algorithm, Cluster Operations