# The Travelling Salesman Problem

Thomas Setzler and Shaina Mainar

April 27, 2019

## 1 Introduction

The traveling salesman problem is a famous NP-complete problem, attempting to solve the most efficient method of traveling between a specified number of cities with varying distances between each of the cities. This problem is an application of weighted graphs with the weighted edges indicating distances between cities and the vertices indicating the cities. An NP-complete problem implies that there is no existing solution for the problem therefore, creating and implementing a solution can be difficult and inefficient in comparison to other problems. Solving this has been done with multiple methods, but none of them have been determined to be the most efficient means of solving this infamous case.

## 2 The Approach

In order to solve this problem, a combination of approaches were used. A greedy algorithm known as the nearest-neighbor algorithm is used to calculate the shortest path between the first city and other

nearby, non-visited cities. This process is repeated with each city to be visited selected as a starting city and their paths and distances stored in an array. After each city has been a starting city, the array is parsed for the shortest distance, and that path is selected to be the shortest path. This choice was made because once the shortest path is found, it is possible to start anywhere on that path seeing as it is a complete circuit.

## 3 Main Method

First, and foremost is the main() method. This method is passed the latitude, longitude, and city name of each city in one continuous list and the total number of cities to visit as parameters. The list is formatted in the same method as if it were to come from a CSV, or Comma Separated Value file. From here, a distance matrix is created to store the distance between each city and every other city. Afterwards, the nearest neighbors algorithm is performed on each city and stored into the array. Upon the conclusion, the shortest path is selected and printed out onto the terminal along with the total distance in kilometers.

```
def main(coords, n):

  distMatrix = matr(coords,n)
  cityPathMatrix = [[0] * n for k in range(n+1)]
  minPath, minIndex, cityPath = algorithm(distMatrix, n, cityPathMatrix)
  print("City Path: \nStarting City: " + str(coords[minIndex * 3 + 2]))
  for i in range(n):
    temp = cityPathMatrix[minIndex][i]
    print("cityNo: " + str(temp) +" "+ str(coords[temp*3+2]))
  print("The minimum cycle through all " + str(n) + " US cities is: "
        + str(minPath) + " kilometers")
  print("The starting and ending city is: " + str(coords[(minIndex*3) + 2]))

  print("done\n")
```

## 4 Distances

Next is the distance calculations. The distance matrix is populated using the matr() method. This method accepts the list of coordinates and the number of cities to visit as parameters. It iterates through each city, populating all of its distances first, then proceeding onto the next distance. The distances are calculated using the geopy library's distance function and given in kilometers. Each distance is the direct distance between the two points, not accounting for any roads or factors involved in the means of transportation for the traveling salesman.

```
def matr(coords,n):
  Matrix = [ [0] * n for k in range(n)]
  i = 0
  j = 0
  xValue = 0
  yValue = 1
  xFixed = 0
  yFixed = 1
  for i in range(n):
    xValue = 0
    yValue = 1
    for j in range(n):
      if i == j:
        Matrix[int(i)][int(j)] = 0
        #do nothing
      else:
        dist = distanceCalc(coords[int(xFixed)], coords[int(yFixed)],
                            coords[int(xValue)], coords[int(yValue)])
        Matrix[int(i)][int(j)] = dist
      xValue += 3
      yValue += 3
    xFixed +=3
    yFixed +=3
  return Matrix
```

## 5 Nearest Neighbor Algorithm

Most importantly is the nearest neighbor algorithm named algorithm(). It accepts a distance matrix, the total number of cities to visit, and an array for storing each city path. This algorithm begins in one city and chooses the closest, non-visited city to it, and marks that city as visited. The city is considered visited when its Boolean value in the array is 1. The closest city is selected with a newMin() function. Next it repeats the same process, but with the previously chosen city until each and every city has been visited. Upon the

visiting of all cities, the algorithm returns directly to the starting city. This process is repeated n number of times, where n is the total number of cities. The shortest path is returned by the function and printed to the terminal in the main method.

```
def algorithm(matrix, n, cityMatrix):
  pathArray = [] #append to the end of pathArray
  visitedList= [False] * n
  visitedCt = 1
  for i in range(n):
    #resetting visitedList
    visitedCt = 0
    k = 0
    for k in range(n):
      visitedList[k] = False
      if i == k:
        visitedList[k] = True
    minPath = 0
    minValue, minIndex = newMin(matrix, i, n, visitedList)
    visitedList[minIndex] = True
    visitedCt +=1
    minPath += minValue
    cityMatrix[i][0] = i
    for j in range(n-1):

      if j != i:
        minValue, minNext = newMin(matrix, minIndex, n, visitedList)
        minPath += minValue
        cityMatrix[i][j] = minNext
        visitedList[minNext] = True
        visitedCt +=1
    # once the algorithim is finished, replaces last value with return to starti
    j +=1
    minPath += matrix[minNext][i]
    cityMatrix[i][j] = i
    pathArray.append(minPath)
  print(pathArray)
  return min(pathArray), pathArray.index(min(pathArray)), cityMatrix
```

## 6 Minimum Value

A key part of the algorithm is the newMin() function. This function accepts the distance matrix, city to travel from, number of cities, and a list of visited cities as parameters. The visited list is actually a list of Booleans set to true or false at the position of the corresponding

city in the distance matrix. It iterates through the entire list of city distances from the chosen city and determines the closest city that doesn't have a corresponding value of 'True', or 1, in the visited list. The function returns the newly found city's distance and index in the list.

```
def newMin(matrix, startPos, n, visited):
  #this function find a minimum value that hasn't been visited yet
  #forward declared variables

  i = 0
  compareVal = 0
  minVal = 0
  minIndex = 0

  while(i < n):
    if(matrix[startPos][i] == 0):
      #if 0, move to the next city
      pass
    elif (visited[i] == False):
      if (minVal == 0):
        #if minvalue has no value and the city hasn't been visited, use it f
        minVal = matrix[startPos][i]
        minIndex = i
      elif (matrix[startPos][i] < minVal):
        #If the city is closer, set it to minVal
        minVal = matrix[startPos][i]
        minIndex = i
      else:
        pass
    else:
      pass
    i += 1

  return minVal, minIndex
```

## 7 Results

Using the program and a list of capitals of the 48 continental states of the United States of America, the program selected the path be-

ginning and ending in Jefferson City, Missouri with a distance of approximately 60,206 kilometers. While this might not necessarily be the desired starting city, because the path is a cycle, one can begin at any point in the cycle and complete their round trip through the cities. The final path is as follows: Jefferson City, Indianapolis, Madison, Des Moines, Frankfort, Nashville, Topeka, Lansing, Columbus, Lincoln, Little Rock, Saint Paul, Charleston, Atlanta, Jackson, Oklahoma City, Montgomery, Columbia, Pierre, Baton Rouge, Raleigh, Richmond, Harrisburg, Montgomery, Annapolis, Tallahassee, Dover, Trenton, Austin, Cheyenne, Denver, Bismarck, Albany, Hartford, Montpelier, Santa Fe, Providence, Concord, Boston, Augusta, Salt Lake City, Helena, Phoenix, Boise, Carson City, Sacramento, Olympia, Salem, Jefferson City. The 48 continental state capitals were chosen because of their connectedness by land and lack of major outlying distances. While the algorithm still works with cities such as Honolulu and Juneau, they were purposefully left out due to their outlying nature in comparison to the 48 continental states. If they are included, the starting city of the shortest path is still Jefferson City, but this time with a distance of approximately 74,501 kilometers. The only difference between this path and the 48 city path, is that Honolulu and Juneau are appended onto the end of the 48 city path immediately before the return to Jefferson City.

## 8 Sources

https://www.geeksforgeeks.org/k-nearest-neighbours/

https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

https://www.sciencedirect.com/science/article/pii/S0166218X01001950

https://www.xfront.com/us_states/