CS 325 Group 10 Adam Cankaya, Kevin Kozee, Ismail Orabi
Project 5 Report, Due Dec 5, 2014

Introduction

We decided to each individually write an algorithm and then compare our results to determine an optimal solution. Adam decided to write an algorithm that would produce a quick solution. Kevin's algorithm takes longer to run, but produces valid tours with shorter total distance. Although Kevin's algorithm takes longer, the runtime is not exponential and is considered acceptable. Based on this we have selected Kevin's algorithm as our final submission.

Adam's Algorithm: Greedy TSP Starting From First City

The algorithm takes in a list, cities[] and then creates a new list, tour[], to which it adds the first element from cities[]. It makes a copy of cities[] called unvisited[] and removes the first city from unvisited[]. The algorithm then starts a continuous loop until unvisited[] is empty in which the last added city in tour[], tour[-1], is compared to every city in unvisited[] and the neighbor to tour[-1] with the shortest distance is added to tour[] and removed from unvisited[].

To determine the neighbor with the shortest distance to a city, the algorithm passes in tour[-1] and the current unvisited[] list to a separate function called closestNeighbor(). This function takes in a list of cities, along with a starting point and loops through every element in the list, to which it calculates the distance between the two, and returns the city element from unvisited[] that is closest to the starting city. Each distance calculation is done by another function, pointDistance, which treats the two cities as x,y points on a 2D cartesian plane.

The biggest improvement that could be made would be to calculate tours starting from a different city besides the first city. The algorithm could be run once for every city in cities[], using each element as a starting point, and saving the total distance along with the tour to a 2D array called solutions[]. Once complete the program would just return the element in solutions[] with the shortest distance.

```
def greedyTSP(cities):
        start = cities[0]
        remove cities[0]
        tour[0] = start
        unvisited = cities
        while unvisited is not empty:
        neighbor = closestNeighbor(tour[-1],
           unvisited)
        append neighbor to tour[]
        remove neighbor from unvisited[]
        return tour
```

```
def closestNeighbor(start, cities):
        neighbor = cities[0]
        bestDistance = pointDistance(start,
           neighbor)
        for i between 0 and length of cities:
                tempDistance =
                  pointDistance(start, cities[i[)
                if tempDistance > 0:
                        if tempDistance < bestDistance:
                                bestDistance = tempDistance
                                best_i = i
        return cities[best_i]
```

Kevin's Algorithm: Better Enumeration with Greedy Ending

This algorithm enumerates through each index of coordinates and calculates the distance between them. The enumeration is slightly better than O(n^2) because it only calculates the distance once for each pair of indices. For example, if the distance from a to b has been calculated the algorithm does not then calculate the distance from b to a. The greedy part kicks in after each distance is calculated and the algorithm checks to see if that distance is the shortest from that particular index (or node). If the most recent calculated distance is the shortest from that node, it is stored in an array of shortest distances. Because we do not iterate back over indices that have already been checked, we know that each new shortest distance take us to a node that we have not yet visited, thus giving us a connected graph.

```
for i in range(len(c)):
    for j in range((i + 1), len(c)):
        d = calculateDistance(c[i][0], c[j][0], c[i][1], c[j][1])
        if (j == (i + 1)):
            s = d
        elif (d < s):
            s = d
    shortest.append(s)
```

Final Algorithm: Better Enumerative Algorithm with Sort

The selected algorithm takes a bit more time than the one's above but gives more accurate and consistent results. This algorithm enumerates over each index i and j calculating the distance between each only once, just as the algorithm above. After storing each distance and set of "cities" in a list, the list is sorted from shortest distance to longest distance. After the list is sorted the first city in the list is added to a "visited" list - it will be the starting point as it has the shortest distance to any given location. The next city (the one connect to the first by the shortest distance) becomes the "current" city and we iterate through the list searching the first instance of the current city that connects us to a city that is not in the visited list; because the list is sorted, we are guaranteed to get the shortest next possible distance between two cities (the current city and the next unvisited city). We continue doing this until all cities are contained in the visited list. As a future update/improvement, we could change the code so that it removes items from the list of distances once one of the cities has been visited as to not continually reiterate over each during second loop

```
def tsp(c):
    c = data[0]
    shortest = []
    visited = []
    distances = []
    for i in range(len(c)):
        for j in range(0, len(c)):
            if (i == j):
                continue
            d = distance([c[i][0], c[i][1]], [c[j][0], c[j][1]])
            distances.append([i, j, d])
    distances = sorted(distances, key=operator.itemgetter(2))
    current = distances[0][0]
    for j in range(len(distances)):
        for i in distances:
            if (((i[0] == current) and (i[1] not in visited)) or ((i[1] == current) and (i[0] not in visited))):
                if ((i[0] == current) and (i[1] not in visited)):
                    shortest.append(i[2])
                    visited.append(i[0])
                    current = i[1]
                    break
                if ((i[1] == current) and (i[0] not in visited)):
                    shortest.append(i[2])
                    visited.append(i[1])
                    current = i[0]
                    break
```