Seattle Pacific University

Department of Engineering and Computer Science

CSC 3430 Algorithm Design and Analysis

Winter 2018

NP Problems and Heuristics

Bomin Kim

Mary Kate La Bar

Sion Park

Table of Contents

Introduction

When big name musical groups go on tour, they sometimes visit cities all over the country. These tours can take months to complete and can be very costly. Our goal is to analyze a concert tour and optimize their travel cost and time. Vampire Weekend recently announced their tour list which will be visiting 30 cities in the United States. Assuming they will be driving a tour bus from city to city, we want to find the shortest route without visiting a city more than once, and ideally returning to the city in which they started. This problem is similar to the Travelling Salesman Problem (TSP). By finding the optimal route the band would save costs such as gas for the bus, lodging in cities they've already stayed in, or time spent on the road.

Background

Travelling Salesman Problem (TSP) is a problem trying to solve the cheapest route for a trip for multiple cities, but the condition is that the person should not visit one city more than once. Comparing to a graph, it is going to be finding the path that has minimum weight of edges that visit all the nodes just once. TSP is an NP (nondeterministic polynomial time) problem which means that the problem could be verified with a proof in a polynomial time even though the problem itself is very hard and time consuming to solve. There exist several heuristic solutions to TSP. Among them, we chose to use "Nearest Neighbor Algorithm", which is a kind of Greedy Algorithm that seeks and selects the best choice for each step. Nearest neighbor algorithm starts with a node and marks that node as visited, then, it will choose to visit a city that has the cheapest edge and repeat that process until all nodes have been marked visited. This will give the path with minimum weight of edges where the node will only be visited once.

Description

We chose to examine Vampire Weekend's tour because it is a large cross-country tour that will take them about 9 months to complete, and it is relevant as the tour will be beginning this month. We collected the list of cities they will be visiting from their website (Sony Music Entertainment, 2019). Then we removed all cities from the list that were outside of the United States, to follow the assumption they will just be driving for travel. This also gives us the ability to change the order of travel, without having to consider cost factors such as plane tickets.

Next, to optimize the route according to TSP, we need to know how far each city is from every other city. Using the scripting language R and a package called `gmapsdistance` (Melo, Rodriguez, & Zarruk,

2018), we were able to calculate the distances in meters. The package utilized the Google Maps API, and we could determine the distance between any two cities in meters, as well as specify mode of transportation (driving, in this case). After running this for all combinations we ended up with 900 distances. For optimizing our route for TSP, we will consider each city as a vertex and each distance as the weight of an edge between two vertices. For the purpose of the algorithm, we needed to choose a starting point. We chose San Francisco because the band is from Los Angeles, this starting point is arbitrary, but we figure the band would like to end close to home.

We coded the TSP algorithm using Python and ran it on a MacBook Air with the following specifications: Processor 1.6GHz Intel Core i5, Memory 4GB 1600 MHz DDR3. Finding the distances for the cities, as well as creating a visual of the final path was coded in R using RStudio and ran on a Lenovo Y40-80 with the following specifications: Intel Core i7-5500U CPU @ 2.40GHz, Memory 8.00 GB.

The data format that we used is:

```
origin, destination, distance
```

We then organized our data using dictionary(map) data structure in our python program:

```
{ “City 1”: { “City 1”: 10000,
              “City 2”: 23485,
              “City 3”:  12345}
 “City 2” : {     …}
}
```

The algorithm starts with a given city, finds a city closest in distance, and repeats until all cities have been added to the route.

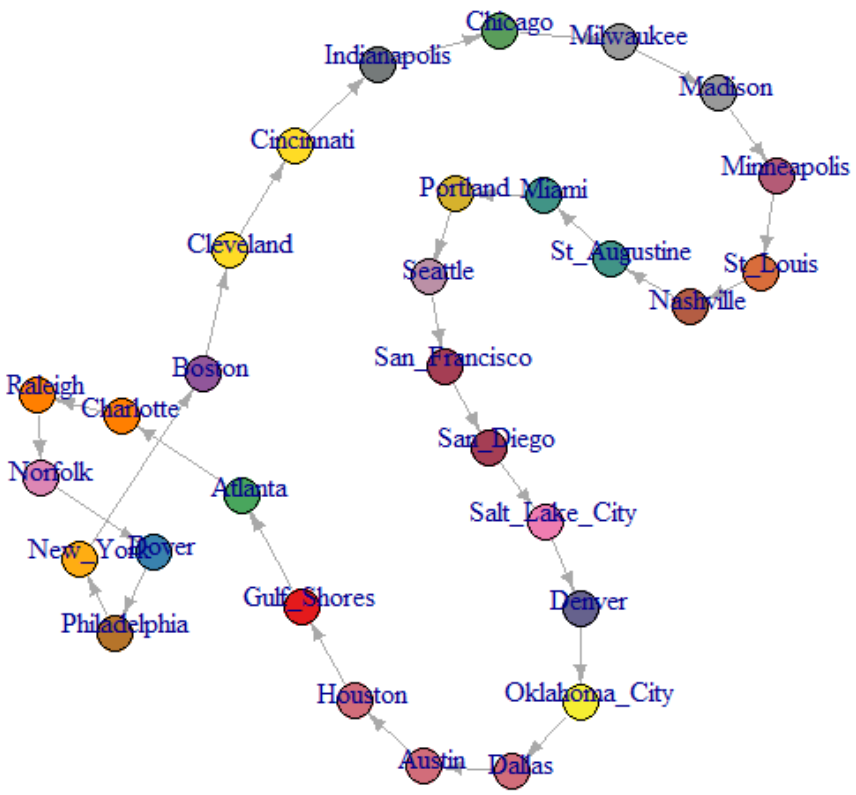
Results

Our program returns the route that suggests one efficient way to tour cities for concert and the total distance of the whole trip. The distance and route changes depending on the starting city.

Below is the screenshot of the results after running the algorithm to solve TSP with our dataset. The shortest route starting from San Francisco returned is 19229013 meters, which is equivalent to 19229.013 kilometers.

```
(['SanFrancisco', 'SanDiego', 'SaltLakeCity', 'Denver', 'OklahomaCity', 'Dallas'
, 'Austin', 'Houston', 'GulfShores', 'Atlanta', 'Charlotte', 'Raleigh', 'Norfolk
', 'Dover', 'Philadelphia', 'NewYork', 'Boston', 'Cleveland', 'Cincinnati', 'Ind
ianapolis', 'Chicago', 'Milwaukee', 'Madison', 'Minneapolis', 'StLouis', 'Nashvi
lle', 'StAugustine', 'Miami', 'Portland', 'Seattle', 'SanFrancisco'], 20489554)
```

We also used RStudio to visualize the final route in graph form. It doesn't reflect geographical location of cities.



Our algorithm is written based on Nearest Neighbor Algorithm. It does not give us an optimal solution, but it is one of the simplest heuristic algorithms to solve TSP (Haider A Abdulkarim, 2015).

In our program, the `tsp(city)` function is called thirty times as there are thirty cities that we need to connect. In each tsp(city) function we call another function `nearestneighbor(city)`, which finds and returns the closest city to the city that is passed as an argument of the function. In the `nearestneighbor(city)` function, we use a dictionary that stores destination cities as keys and its distance as values. The shortest distance is set to a large number as default and we update shortest distance as we compare the current distance to the shortest distance. Then we add the returned shortest distance to the total distance and add the city to the visited array. We repeat this step for every city in the list and it results in O(n^2) run time complexity.

In the Nearest Neighbor Heuristic Algorithm, every time it checks the distance between cities, it skips comparison for the cities that are already visited. This brings down the algorithm to O(n log n) time

complexity (The Traveling Salesman Problem, n.d.). However, our algorithm doesn't account for the case where the city is already visited. We don't have any operation to remove visited cities in the dictionary, so it traverses all thirty cities for each city to find the shortest distance, giving O(n^2) time complexity.

It is also worth mentioning that the city chosen to start with will affect the overall distance of the route. Since we start in San Francisco, we cannot guarantee that the final city will be anywhere near San Francisco. This could add a large distance at the end of the route we do not want. Another idea to optimize the solution could be to pick a starting city that is in a more centralized location with respect to the other cities.

Conclusion

Even though the solution we found is not the optimal one, our algorithm was able to solve the problem in polynomial time O(n^2) using parts Nearest Neighbor Algorithm. By researching the heuristic solution and implementing it, we learned that it is not always easy to solve a problem even though we understand its components. When discussing algorithms and time complexity in class, it seems obvious when we find the most efficient implementation. Even though we fully understood the concepts of TSP problem and the researched heuristic solution, it was very hard to explain and implement that to solve an actual problem with real piece of data. In practice with real data, we realized that it is difficult to execute. Sometimes it seemed we got stuck in one way of viewing a problem and cannot think of a better way to solve it. Even if our current solution is very slow. It takes a lot of time to analyze and understand how best to solve a problem. With this experience, we learned a valuable lesson that we actually have to sit and think about what we are writing rather than just coding right away as programmers. We now will aim to think about ways to solve the problems efficiently rather than solving the problem with whatever comes to mind first.

Bibliography

Csardi G, Nepusz T: The igraph software package for complex network research, InterJournal, Complex Systems 1695. 2006. http://igraph.org

Erich Neuwirth (2014). RColorBrewer: ColorBrewer Palettes. R package version 1.1-2. https://CRAN.R-project.org/package=RColorBrewer

Khushboo Arora, S. A. (2016, January). Solving TSP using Genetic Algorithm and Nearest Neighbour Algorithm and their Comparison. *International Journal of Scientific & Engineering Reserach, 7*(1).

Melo, R. A., Rodriguez, D., & Zarruk, D. (2018). *gmapsdistance: Distance and Travel Time Between Two Points from Google Maps. R package version 3.4*. Retrieved from The Comprehensive R Archive Network: https://CRAN.R-project.org/package=gmapsdistance

Sony Music Entertainment. (2019). Retrieved from Vampire Weekend: https://www.vampireweekend.com/

Appendix

TSP Algorithm (coded in Python)

```
#Read inputfile
file = open("cities_cp.txt","r")
lines = file.read().split("\n")

#Neighbor to store each city
neighbor =""
#Initialize visited and route list
visited = []
route = []
#Map cities and other cities into dictionary
cities = dict()

countCities = 0
totalWeight =0

#Find closest city
def nearestNeighbor(city):
    route = cities[city]
    #Default shortest distance
    shortestDistance = float("inf")

    for neighbor, weight in route.items():
        weight = int(weight)
        #update shortest distance if city is not visited
        #and less than the current shortest distance
        if weight < shortestDistance and neighbor not in visited: #compare
distance, if new dist is smaller enter if
            city = neighbor
            shortestDistance = weight

    return shortestDistance, city

#Inputfile data into dictionaries.
def map(lines):
    global countCities
    for line in lines:
        line =line.split()
        if countCities ==0 or countCities%30 ==0:
            neighbor = line[0]
            cities[neighbor]= dict()
            cities[neighbor][line[1]] = line[2]
            countCities =0
        else:
```

```
            cities[neighbor][line[1]] = line[2]
        countCities += 1
        # if neighbor in
    return cities


def tsp(city):
    global totalWeight
    if len(visited) ==30:
        #add the first city to the last visited list
        lastDistance = int(cities[visited[29]][visited[0]])
        totalWeight += lastDistance
        visited.append(visited[0])
        return visited, totalWeight
    else:
        minimumDistance, closestCity = nearestNeighbor(city)
        visited.append(closestCity)
        totalWeight += minimumDistance
        tsp(closestCity)

#put cities into a map
map(lines)
print(cities)
#starting city
tsp("SanFrancisco")
print (visited, totalWeight)
```

Finding Distances (coded in R)

```
install.packages("gmapsdistance")
set.api.key("A**************4") #censored for security
cities<-c(TourCities$`City+State`)
results = gmapsdistance(origin = cities, destination = cities, combinations =
"all", mode = "driving", shape = "long")

write.csv(results$Distance, file = "Distances.csv")
```

Creating Map Visual (coded in R)

```
install.packages("igraph")

nodes <- read.csv("citiesNodes.csv", header=T, as.is=T)
links <- read.csv("citiesEdges.csv", header=T, as.is=T)

library(igraph)
```

```
library(RColorBrewer)
path <- graph_from_data_frame(d=links, vertices=nodes, directed=T)

colorCount = length(unique(nodes$State))
getPalette = colorRampPalette(brewer.pal(9, "Set1"))
V(path)$color <- palette(getPalette(colorCount))[ as.factor(V(path)$State)]

plot(path, vertex.label=nodes$City,
     vertex.size=20,
     vertex.label.dist=1,
     edge.arrow.size=0.4,
     margin =-.4,
     layout=layout_with_dh(path, coords =as.matrix(nodes[,c("long","lat")])))
```