Nathan Aun

7/30/19

Lab #5 – Graphs

CS2302 – Data Structures

Summer 2019

Introduction

The problem posed for this lab was to determine the lowest cost path for drones to fly from one city to another. We are supposed to do this by using Dijskta’s algorithm on all available routes and we are supposed to do it again with a minimum spanning tree found by using Prim's algorithm. We will then compare the weights between these two paths, and see how much longer a minimum spanning tree would take when compared to a more direct path.

Proposed Solution

The first method to implement reads an excel spreadsheet and stores the info in two different lists. One list has all the cities in it and the other list has the graph represented as an adjacency list. The next method was Dijkstra’s method was implemented in order to find the minimum cost path from one city to every other. This method returns two different lists, one that holds the paths and another that holds the weights to each city.

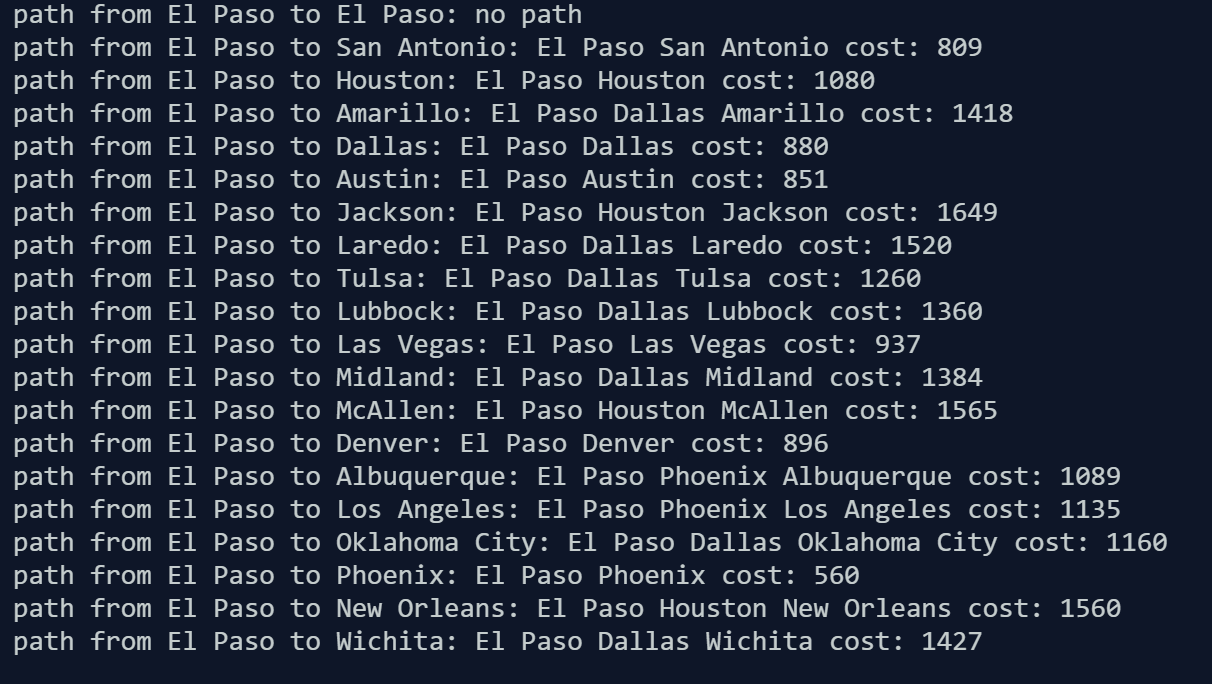
The next important method implemented was one that did a lot of things that admittedly could’ve been in the main method, however I abstracted these functions out to make the code a little more clean. The handlePaths method takes our adjacency list and cities list made by the first function and runs Dijkstra’s algorithm on every city. It adds the weights to a new list that holds the weights for all cities, and the paths to a list that holds the paths to all cities. Once those two things are taken care of, it will run the printPath method which will print the path and the cost from one city to every other. It does this in a loop so that every path from every city to every other city will be printed alongside their cost. It also returns the weight of the most recently created path for use later.

The next method to implement is Prim’s minimum spanning tree algorithm. There are many ways that this can be implemented and, at the time of writing the code, there was none on the class website so the implementation I followed was very close to the Kruskal implementation we did in class, however I also used minHeaps in order to make it more efficient. Prim’s algorithm uses a disjoint set forest to determine if a city has been added to the tree yet, it uses a minheap of edges to determine what is the next minimum cost edge and it will add the minimum cost edge to a city that hasn’t been visited yet to a new adjacency list. Once it has gone through every city, it will return the new adjacency list.

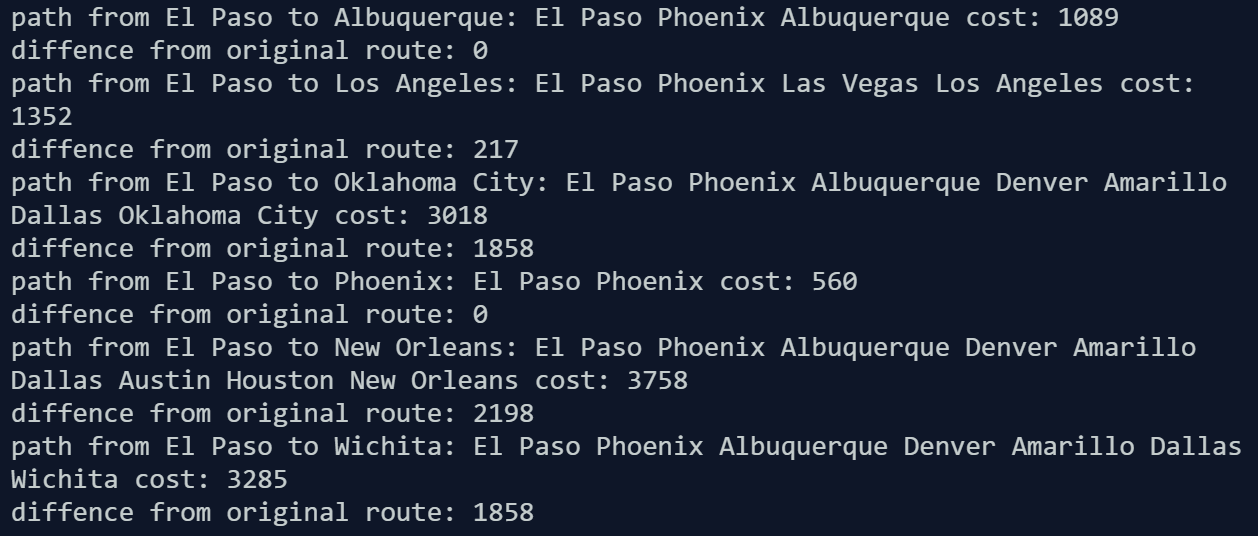
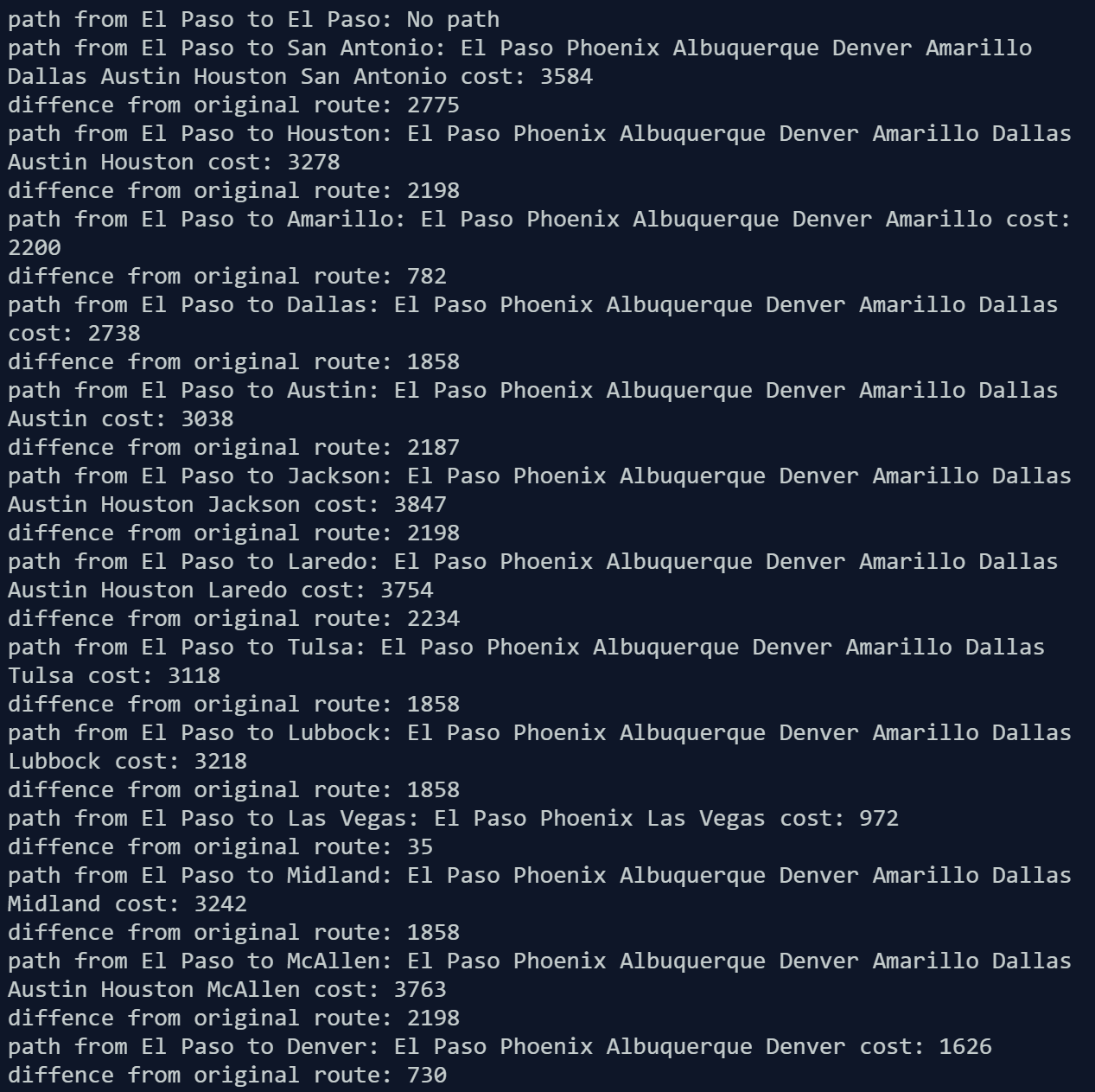
Once this new adjacency list is created, it will run the handlePaths method again, this time passing the extra parameter of the weight from the original run of handlePaths. The program will execute the same up to the point where weights are being printed. This run will also print the difference between the new cost and the original cost of flying from city A to city B.

Conclusions

Since there are a lot of different things that are printed in the output of the lab this section will only show one example of the differences between the first run and the second run of handlePaths, but it will use this one example to talk about all the other outputs.



Run 1 of the program

 Run2

As you can see from the above program outputs for El Paso, the cost after Prim’s algorithm has been run is never faster than the original run of the program, and that is because a lot more direct routes were cut out in order to make the minimum spanning tree. Most of the time the run on prim’s minimum spanning tree is a lot worse than it was on the original, but these are just the sacrifices that need to be made in order to reduce the overall number of paths that can be taken. In no case on the entirety of the output does the path on prim’s algorithm become faster than the original, the closest case is when it ends up taking the same path. This makes sense, because limiting the amount of paths available would not make going from one city to another any faster, but instead the drone would have to fly through more cities in order to get to the same destination.

Appendix

Source code is as follows:

'''

link to report: https://docs.google.com/document/d/1f\_2ij0A9bN8O1Ijv0eAM\_nzhIV-\_2kOhZTNZhz2hSZ4/edit?usp=sharing

Course: CS2302 Data Structures

Author: Nathan Aun

Assignment: Lab 5 - Graphs

Instructor: Dr. Fuentes

TA: Ismael Villanueva-Miranda

Date of Last Modification: 7/30/2019

Purpose of Program: To determine the lowest cost path for a drone to get from one city to another

 '''

import numpy as np

import math

import dsf

import csv

#reads the spreadsheet given and will make two lists, one of the cities and the other of the costs it takes to get from one city to another

def readSpreadsheet(fileName):

    f = open(fileName, 'r')

    wb = csv.reader(f)

    costs = []

    cities = []

    temp = []

    for i in wb:

      temp.append(i)

    for i in range(1,len(temp)):

        costs.append([])

        for j in range(1,len(temp[i])):

            if int(temp[i][j]) > -1:

                costs[i-1].append([j-1,int(temp[i][j])])

    for i in range(1, len(temp)):

        cities.append(temp[0][i])

    return costs, cities

#Dijksra will run Dijkstra's algorithm to get the paths from one starting vertex to every other city

def Dijkstra(startV,G):

  visited = [False for v in G]

  dist = [math.inf for v in G]

  path = [-1 for v in G]

  dist[startV] = 0

  while (False in visited):

    smallest = math.inf

    vertex = -1

    for i in range(len(dist)):

      if dist[i] < smallest and visited[i] != True:

        smallest = dist[i]

        vertex = i

    visited[vertex] = True

    for i in G[vertex]:

      edgeWeight = i[1]

      alternativePathDistance = dist[vertex] + edgeWeight

      if (alternativePathDistance < dist[i[0]]):

        dist[i[0]] = alternativePathDistance

        path[i[0]] = vertex

  return path, dist

#HandlePaths is used in order to handle the information recieved from readSpreadsheet and make paths out of it. This is where most of the program takes place

def handlePaths(AL, cities, compWeight = []):

    path = []

    weight = []

    for i in range(len(cities)):

        tempPath, tempWeight = Dijkstra(i,AL)

        path.append(tempPath)

        weight.append(tempWeight)

    for i in range(len(path)):

        for j in range(len(path[i])):

            print("path from", cities[i],"to", cities[j], end = ': ')

            if compWeight == []:

                cost = printPath(cities, j, i, path[i], weight[i])

                if cost != None:

                    print("cost:", cost)

                else:

                    print("no path")

            else:

                cost = printPath(cities, j, i, path[i], weight[i], )

                if cost != None:

                    print("cost:", cost)

                    cost = cost - compWeight[i][j]

                    print("diffence from original route:", cost)

                else:

                    print("No path")

        print()

    return weight

#printPath will actually print the path that is taken by dijkstra's algorithm

def printPath(cities, j, i, path, weight):

    if path[j] == i:

        print(cities[i], end = ' ') #depends on what we want to display, if the path contain the og city

        print(cities[j], end = ' ')

        return weight[j]

    elif path[j] == -1:

        return None

    printPath(cities, path[j], i, path, weight)

    print(cities[j], end = ' ')

    return weight[j]

#The next few methods were taken from an in-class exercise on heaps in order to make the implementation of prim's algorithm better

#finds the parent of an index in a heap

def parent(i):

    return (i-1)//2

#finds left child of an index

def leftChild(i):

    return 2\*i + 1

#finds right child of an index

def rightChild(i):

    return 2\*i + 2

#inserts a value in a heap while maintaing heap structure

def HeapInsert(H,item):

    H.append(item)

    i = len(H)-1

    while i > 0 and item[2] < H[parent(i)][2]:

        H[i] = H[parent(i)]

        i = parent(i)

    H[i] = item

#removes the top value in a heap while maintaining heap structure

def ExtractMin(H):

    if len(H) <1:

        return None

    if len(H) ==1:

        return H.pop()

    root = H[0]

    p = H.pop()

    i = 0

    m = math.inf

    minVal = math.inf

    while True:

        m = math.inf

        L = [p]

        if leftChild(i) <len(H):

            L.append(H[leftChild(i)])

            if rightChild(i) <len(H):

                L.append(H[rightChild(i)])

        for j in L:

            if j[2] < m:

                m = j[2]

                minVal = j

        H[i] = minVal

        if minVal == p: #  Parent is larger that both children

            break

        elif minVal == L[1]:

            i = leftChild(i)

        else:

            i = rightChild(i)

    return root

#prim's algorithm will find the minimum spanning tree for a graph given an adjacency list.

def primsMinTree(AL):

    forest = dsf.dsf(len(AL))

    AL2 = [[] for i in range(len(AL))]

    heap = []

    vertex = 0

    for i in range(len(AL)-1):

        for j in range(len(AL[vertex])):

            HeapInsert(heap,[vertex] + AL[vertex][j])

        while True:

            pointWeight = ExtractMin(heap)

            if pointWeight == None:

                break

            if forest[pointWeight[1]] == -1 and pointWeight[1] != 0:

                AL2[pointWeight[0]].append(pointWeight[1:])

                AL2[pointWeight[1]].append([pointWeight[0], pointWeight[2]])

                dsf.union(forest,pointWeight[0],pointWeight[1])

                vertex = pointWeight[1]

                print()

                break

    return AL2

if \_\_name\_\_ == "\_\_main\_\_":

    AL, cities = readSpreadsheet("cities.csv")

    #print(AL)

    #graphs.draw\_weighted\_graph(AL)

    originalWeight = handlePaths(AL, cities)

    print()

    minSpanAL = primsMinTree(AL)

    #graphs.draw\_weighted\_graph(minSpanAL)

    #graphs.draw\_directed\_weighted\_graph(minSpanAL)

    #print(minSpanAL)

    handlePaths(minSpanAL, cities, originalWeight)

Statement of Academic Honesty

“I certify that this project is entirely my own work, I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_