Kimberly Tom

CS 325

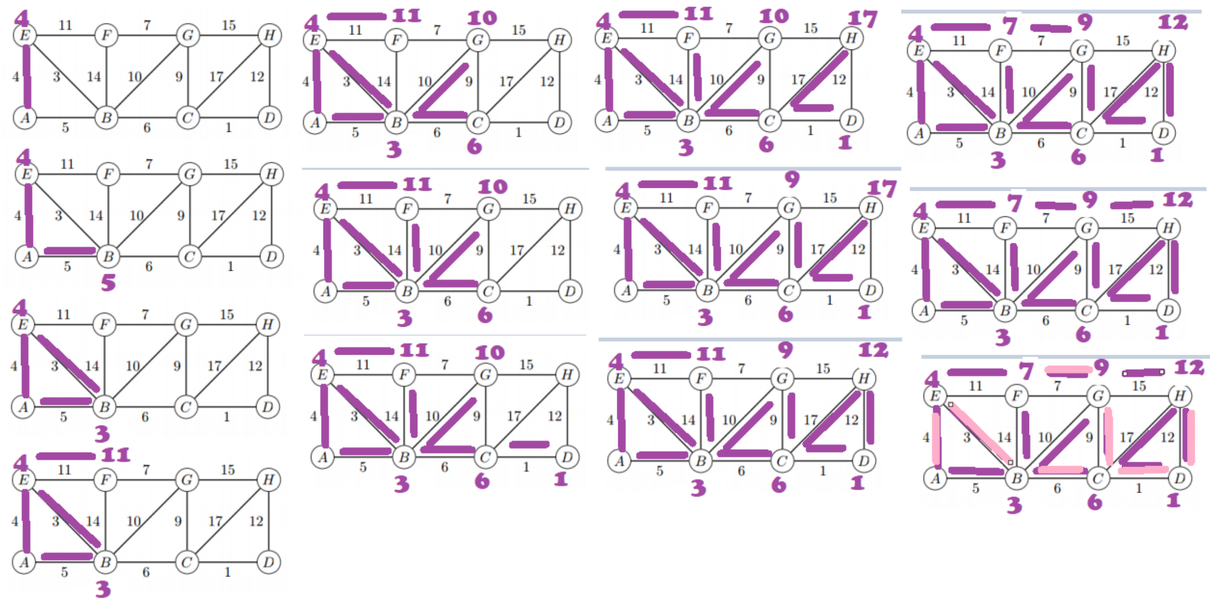
2/16/19

Homework 5

1)

a) (A,E), (E,B), (B,C), (C,D), (C,G), (G,F), (D,H)

Work is shown in columns, starts from first column, top to bottom, after hit bottom row, go to next column to the right. Also, initialize all vertexes without purple numbers to infinity.



b) If each edge weight was increased by 1, this will not change the minimum spanning tree (MST) because we always update the vertex to the lowest edge weight, so if all edges are increased by one, the lowest edge weight for that vertex before we increased the weights by 1, will still be the lowest edge weight for that vertex after we increase the weights by 1.

2)

a) I would use Dijkstra's algorithm because this algorithm is good for finding the shortest paths from a single source to a single destination. It can find a shortest path from a given source (vertex G), to each of the vertices.

	S	d_v	p_v
A	F	infinity	-

B	F	infinity	-
C	F	infinity	-
D	F	infinity	-
E	F	infinity	-
F	F	infinity	-
G	F	infinity	-
H	F	infinity	-

	S	d_v	p_v
A	T	12	C
B	T	6	H
C	T	8	D
D	T	5	E
E	T	2	G
F	T	8	G
G	T	0	-
H	T	3	G

b)

Assuming we already have Dijkstra's algorithm, which takes in a graph of the city, and the source, from the lecture slides.

```
SSSP-Dijkstra(G, w, s)
    InitializeSingleSource(G, s)
    S ← ∅
    Q ← V[G]
    while Q ≠ 0 do
        u ← ExtractMin(Q)
        S ← S ∪ {u}
        for u ∈ Adj[u] do
            Relax(u,v,w)

InitializeSingleSource(G, s)
for v ∈ V[G] do
    d[v] ← ∞
    p[v] ← 0
d[s] ← 0

Relax(u, v, w)
    if d[v] > d[u] + w(u,v) then
        d[v] ← d[u] + w(u,v)
        p[v] ← u
```

source: CS325 Part 3: chapter 24 lecture

SSSP-Dijkstra(G, w, s)

```
optimalLocationFinder(G) {
// G is the graph
// f is the number of possible locations for the fire station
// r is the possible roads
int optimalLocation = -1; // set to negative 1 as no optimal location is found yet
int currentOptimalDistance = infinity; // really high number, as we will be  trying to minimize this
// for each location in the graph
for (int i = 0; i < f – 1; i++) {
        int sum = 0; // this holds the total distance of the paths for each graph
        distance = SSSP-Dijkstra(G, f, i); // distance[] will have an array of best paths from source i
```

```
        sum = MAX(distance) // find the maximum total distance of the shortest paths

        // if the sum of the distances for the graph is less than our current optimal sum

        // then update the currentOptimalDistance and optimalLocation

        if (sum < currentOptimalDistance) {

                currentOptimalDistance = sum;

                optimalLocation = i;

        }


}

return optimalLocation;

}
```

Assume Dijkstra’s algorithm takes theta(V^2) in our optimalLocationFinder, where V is number of vertexes. The loop from 0 to f is theta(V). Inside the loop, we have Dijkstra’s algo which takes theta(V^2), finding the maximum distance which must loop through all the locations from the vertex, so theta(V), and the if statement takes theta(1) to update the optimal distance and location. Therefore, V * (V^2 + V + 1) = V * V^2 = theta(V^3).

c) Since we are finding a location that minimizes the distance to the farthest intersection, we need to look at each vertex and calculate the shortest distance to each of the other vertexes. Then, we will look at the table’s greatest distance from the starting vertex and compare it to the other tables. The table with the shortest maximum will be the optimal location. After calculating for each vertex below, location E as the starting vertex has the shortest furthest distance, so E is the optimal location for my fire station. See work below.

	S	d_v	p_v
A	T	0	-
B	T	**18**	B
C	T	4	A
D	T	7	C
E	T	10	D
F	T	6	C
G	T	12	E
H	T	15	G

	S	d_v	p_v
A	F	**18**	C

B	T	0	-
C	T	14	D
D	T	11	E
E	T	8	G
F	T	14	G
G	T	6	H
H	T	3	B

	S	d_v	p_v
A	T	4	C
B	T	**14**	H
C	T	0	-
D	T	3	C
E	T	6	D
F	T	2	C
G	T	8	E
H	T	11	G

	S	d_v	p_v
A	T	7	C
B	T	**11**	H
C	T	3	D
D	T	0	-
E	T	3	D
F	T	5	C
G	T	5	G
H	T	8	G

	S	d_v	p_v
A	T	**10**	C
B	T	8	H
C	T	6	D
D	T	3	E
E	T	0	-
F	T	8	C
G	T	2	E
H	T	5	G

	S	d_v	p_v
A	T	6	C
B	T	**14**	H
C	T	2	F
D	T	5	C

E	T	8	E
F	T	0	-
G	T	8	F
H	T	11	G

	S	d_v	p_v
A	T	**12**	C
B	T	6	H
C	T	8	D
D	T	5	E
E	T	2	G
F	T	8	G
G	T	0	-
H	T	3	G

	S	d_v	p_v
A	T	**15**	C
B	T	3	H
C	T	11	D
D	T	8	E
E	T	5	G
F	T	11	G
G	T	3	H
H	T	0	-

3)

a)
```
def main():
    wrestlerGraph = parse()

    # Repeat until all wrestlers have been looked at

    while True:
        wrestler = findUnvisited(wrestlerGraph['graph'])
```

```
    if wrestler == "all wrestlers looked at":
        break

    # call bfs algorithm
    bfs(wrestlerGraph['graph'], wrestler)

# vertices with even type are baby faces and all vertices with odd type are heels
# Check that every edge goes between an even and odd type

rivalriesOK = True

graph = wrestlerGraph['graph']
edges = wrestlerGraph['edges']

# each edge needs to be between a odd and even vertex to be true
for edge in edges:
    dist1 = graph[edge[0]]['type']
    dist2 = graph[edge[1]]['type']
    result = dist2 - dist1

    # if there is a result that does not go between an odd or even, then rivalriesOK is set to false
    if result % 2 == 0:
        rivalriesOK = False

# if rivalries OK is false, then print impossible
if not rivalriesOK:
    print("Impossible")

# else, print the teams
```

```
    else:
        print("Yes Possible")

        teamBabyFace = ""
        teamHeels = ""

        # if type is even, then wrestler is a babyface, else wrestler is a heel
        for wrestler in wrestlerGraph['graph']:
            if wrestlerGraph['graph'][wrestler]['type'] % 2 == 0:
                teamBabyFace += wrestler + " "
            else:
                teamHeels += wrestler + " "



        print("Babyfaces: " + teamBabyFace)
        print("Heels: " + teamHeels)

# Parse to create a graph where each wrestler is a key in a dictionary
def parse():
    # read the file name that the user types
    textFile = sys.argv[1]

    # assert that it exists
    assert os.path.exists(textFile), "Error: A file with that name is not in the same directory."

    # open the text file
    readFile = open(textFile, "r")

    # Read the lines
```

```
lines = readFile.readlines()

graph = {}  # create empty dictionary
wrestlersCount = 0  # number of wrestlers
edgesCount = 0  # edges between wrestler
totalEdges = 0  # total edges between wrestlers
edges = []  # create array of edges between wrestlers
loopCount = 0  # loop count

for line in lines:
    # Remove all leading and trailing spaces from string
    line = line.strip()

    # if we are at the first line, then it is the number of wrestlers
    if loopCount == 0:
        wrestlersCount = int(line)

    # create graph that has wrestlers as keys, and values as visited, type, and edges
    if 0 < loopCount <= wrestlersCount:
        if loopCount == 1:
            beginning = line

        # each line represents a wrestler, if it is not a number
        # for each wrestler, initialize visited to false, type to zero, and edges to empty
        graph[line] = {
            'visited': False,
            'type': 0,
            'edges': []
        }
```

```
    # once we hit the next integer, it is not a wrestler, but the number of edges
    # store number in totalEdges
    if loopCount == wrestlersCount + 1:
        totalEdges = int(line)

    # create edges between wrestlers, this simulates the pairings
    if loopCount > (wrestlersCount + 1) and edgesCount < totalEdges:
        # split the string into an array of substring
        # https://www.pythonforbeginners.com/dictionary/python-split
        wrestler = line.split()
        edges.append(wrestler)
        graph[wrestler[0]]['edges'].append(wrestler[1])
        graph[wrestler[1]]['edges'].append(wrestler[0])
        # increment edge count
        edgesCount += 1

    # increment loop count
    loopCount += 1

# close the file
readFile.close()

# return the parsed info
return {
    'beginning': beginning,
    'graph': graph,
    'edges': edges
}
```

```
# BFS algorithm
# with help from https://www.programiz.com/dsa/graph-bfs
def bfs(graph, beginning):
    queue = [beginning]

    # set wrestlers we visited from false to true
    graph[beginning]['visited'] = True

    # while there are still wrestlers in the queue
    while queue:
        wrestler = queue.pop(0) # pop the wrestler
        type = graph[wrestler]['type'] + 1

        for neighborWrestler in graph[wrestler]['edges']:
            if not graph[neighborWrestler]['visited']:
                graph[neighborWrestler]['visited'] = True
                graph[neighborWrestler]['type'] = type
                queue.append(neighborWrestler)

    return graph




# finds unvisited wrestler in the graph
def findUnvisited(graph):
    for wrestler in graph:
```

```python
        if not graph[wrestler]['visited']:
            return wrestler

    # return all wrestlers looked at once all have been visited
    return "all wrestlers looked at"



# call main function to start program
if __name__ == '__main__':
    main()
```

b) The running time of my algorithm is O(n+r) because I need to run through the list of n wrestlers which is O(n). This is done to assign each wrestler as a Babyface or a Heel. Then I need to check the r edges and see if each babyface wrestler has an edge to a heel wrestler which is O(r). The other things like printing will run in O(1) time. Therefore running time is O(n+r).

c) see zipped folder in teach