Lachlan Sinclair

Homework 5

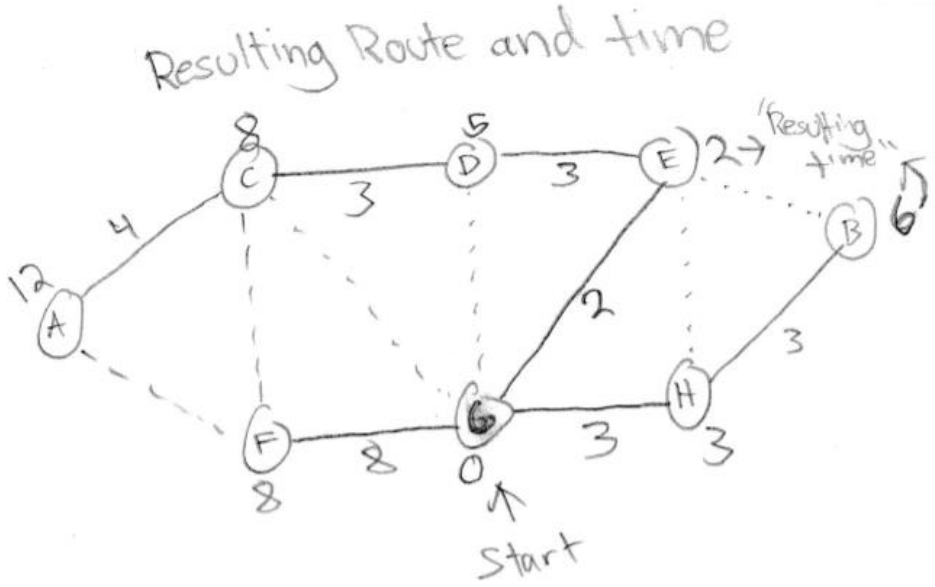
7/27/2019

Problem 1:

a) I recommend using Dijkstra's shortest path algorithm.
Running algorithm can be modeled by the table below.

Name	Step	Distance(Time)	Parent
A	8	12	C
B	5	6	H
C	7	8	D
D	4	5	E
E	2	2	G
F	6	8	G
G	1	0	NIL
H	3	3	G

In this table the distance represents the total time to the intersection via the shortest route from G, the route taken can be produced by backtracking through the parent nodes until G is reached.



b) To find the optimal location in the graph I would run Dijkstra's shortest path algorithm using an adjacency list and a min priority queue using a min binary heap, testing every vertex f in graph as the origin and selecting the intersection that has the lowest max distance to any other intersection.

In order to track the optimal intersection, I would create a struct that contains an int property and a pointer to an intersection. The int will represent the maximum distance required to travel any other intersection when the intersection pointed to by the struct is selected as the location. The int will initially be set to infinity and the pointer to an intersection will be set to null. This will insure the first intersection processed gets set as the optimal intersection to use. At the end of each run of Dijkstra's algorithm, the distance to last intersection processed will be compared to the int value of this struct, if it is smaller the pointer will be changed to point to this intersection and the int will be set to the new smallest max distance. This comparison will be done in constant time therefor not affecting the runtime.

The complexity of this algorithm depends on Dijkstra's algorithm, which when implemented with an adjacency list and a min priority queue using min binary heap, has a runtime of O(r*lg(f)). Dijkstra's algorithm gets ran f times therefor the overall runtime of my algorithm will be O(f*r*lg(f)).

c) E is the optimal intersection. Selecting intersection E results in the lowest maximum distance 10 to any other intersection.

d) The algorithm I would use would be to alter Dijkstra's algorithm to begin with two starting vertices. The major difference between my algorithm and the standard is how to start it. I would still use an adjacency list and a min priority queue that uses min binary heap. In this problem there will be two vertices with 0 distance which could cause problems down the road. So before entering the normal while loop I would add both starting vertices to the Set (removing them from the queue as well), then relax all the adjacent vertices (intersections) of both. If there aren't any other intersections the solution is trivially true, and the while loop will be skipped. After that different start Dijkstra's algorithm will be able to function normally once in the while loop.

This paragraph is pretty much a verbatim analysis of problem b just with two intersection being tracked rather than one. In order to track the optimal intersections, I would create a struct that contains an int property and two pointers to intersections. The int will represent the maximum distance required to travel any intersection when the two intersections pointed to by the struct are selected as the locations. The int will initially be set to infinity and the pointers to the intersections will be set to null. This will insure the first intersection pair processed gets set as the optimal intersections to use. At the end of each run of Dijkstra's algorithm, the distance to last intersection processed will be compared to the int value of this struct, if it is smaller the pointers will be changed to point to these intersections and the int will be set to the new smallest max distance. This comparison will be done in constant time therefor not affecting the runtime.

With two starting points the run time of Dijkstra's algorithm will still be O(r*lg(f)) when implemented with an adjacency list and a min priority queue using a min binary heap. Dijkstra's algorithm will need to be run for every possible pair of intersections, which will be (f-1)+(f-2)+…+(1) total combinations which is trivially bound above by f^2 there for we can say it will take

$O(f^2)$ runtime to look at every pairing. Which then means the overall runtime of my algorithm will be $O(f^2*r*lg(f))$.

e) C,H are the optimal locations for two fire stations. The max distance to any intersection is 5, which is the lowest achievable max distance.

Problem 2:

a) An efficient algorithm to solve this problem would be to use a breadth first search starting at s, and only processing edges with at least a weight of W, once t is found the algorithm terminates and the path can be determined by backtracking through the parent nodes from t. I would use an adjacency list to record the edges of the graph, to be explicit the adjacency list would also store the weight of the edges. All other the data structures used would match the ones describe in the text book in section 22.2.
b) The run time of this algorithm will be the runtime of the breadth first search algorithm which is O(V+E). The only change my algorithm would make would be a simple inequality check while in the for loop on line 12, any vertex v adjacent to u with a weight less than W wouldn't be processed. This check would occur in constant time therefor not affecting the run time.

Problem 3:

a) Pseudocode:

```
Struct wrestler: 3 properties: string name, int distance, int finished, link *next
Struct link: 2 properties: struct wrestler* wres, link * next
numOfWrestlers<- read in from file

wrestlers[numOfWrestlers]: new array of structs

for each wrestler in wrestlers
        wrestler[].name <- read in from file
        wrestler[].finished=0
        wrestler[].distance=-1
        wrestler[].link=NULL

mat[numOfWrestlers][numOfWrestlers]: create adjacency matrix, initialize to all 0s

numOfRivals <-  read in from file

for i=0 to numOfRivals

        rival1 <-index of first rival in wrestlers array

        rival2 <-index of second rival in wrestlers array

        wrestler[rival1]->next = wrestler[rival2] //add to front of list
```

```
        wrestler[rival2]->next = wrestler[rival1] //add to front of list
for i=0 to numOfWrestlers
        if wrestlers[i].finished =0
                int check = BFS(wrestlers, I, numOfWrestlers)
if check = 1
        return;
for i =0 to numOfWrestlers
        if wrestler[i]%2==0
        display wrestler as baby face
else
        display wrestler as heels



Int BFS(wrestlers, index, numOfWrestlers)
        wrestlers[index] .distance=0
        Q <- create queue of wrestlers
        Q.add(wrestlers[index])
        While(Q is not empty)
                temp = Q.front()
                Q.pop()
                tempN = temp.next
                while(tempN != null)
                        If tempN.wres.finished == 0
                                tempN.wres finished=1
                                tempN.wres.distance=temp.distance+1
                                Q.push(tempN.wres)
                        If tempN.wres.finished ==1 || tempN.wres.finished ==2 && tempN.wres
                        .distance has the same parity as temp.distance
                                Print “no solution”
```

Return 1;

tempN=tempN.next

temp.finished=2

Return 0

b) The run time of my algorithm is O(V+E), where V is the number of wrestlers and E is the number of rivalries. My algorithm is a simple implementation of the BFS algorithm provided in section 22.2 of the text book. It uses an adjacency list, and the c++ STL queue data structure which adds to the back and removes from the front in constant time and there for is consistent with the ADT described in the book.