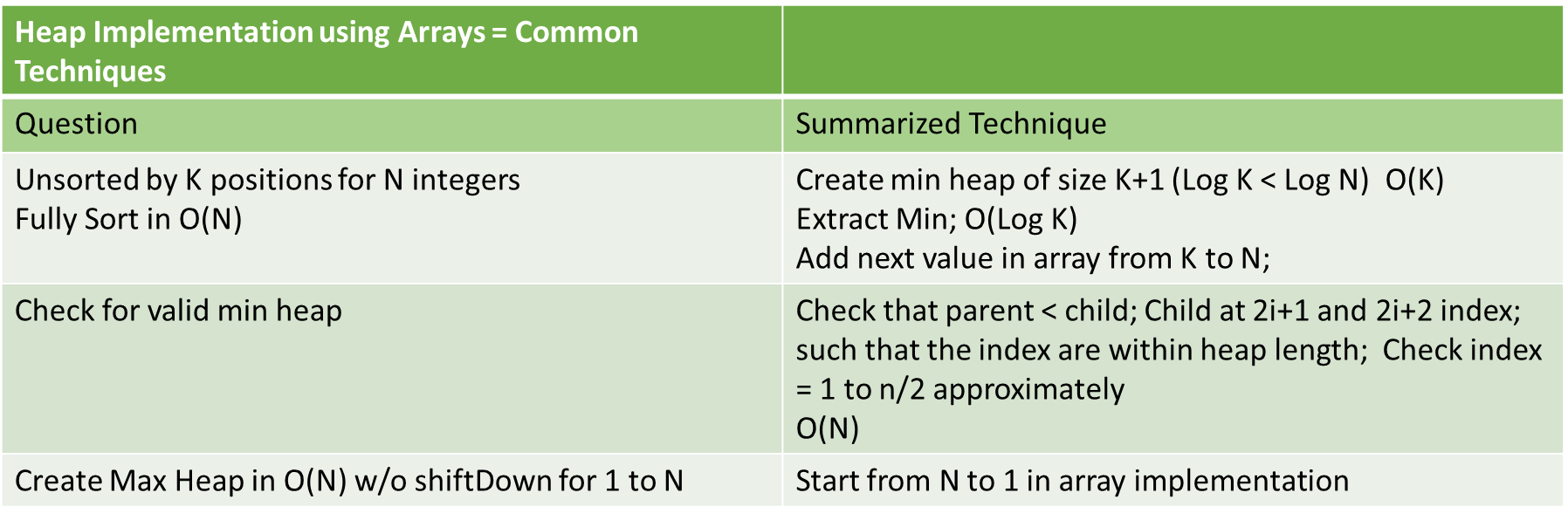
Questions and Algo

HEAP



1. a) Given an unsorted array 𝐴 of 𝑁 unique floating point values of no fixed precision, where each value may only differ from its correct position in the sorted array by no more than 𝐾 positions (𝐾 is much smaller than 𝑁), give an algorithm that will fully sort 𝐴𝐴 in 𝑂𝑂(𝑁) time. For example if 𝐾 = 3 and in sorting A, the value 10.3 is at index 10, then in the original unsorted array 10.3 can be possibly found at indices 7,8,9,10,11,12,13. [10 marks]

Create Min Heap then keep Extracting Min --> O(Log(N))

Min heap creation using array implementation from lecture MinH;

Naive Approach

Insert All Values into Min Heap; O(N) or O(NLog(N)) depending on heap create method;

A = new ArrayList;

while minheap != empty{

A.insert(minheap.extract()); O(Log(N));



}

Overall O(N Log N);

Log K Implies Heap size of K; Important!!

Insert 1st K+1 of A into MinH = O(KLog(K)) or O(K) depending on method to create heap;

Cant insert only 1 to K values;

Eg if we have 1,4,3,5,2 – and we only insert 1,4,3 rather than 1,4,3,5; when we extract 1 to get 4,3,5 🡪 we are missing the 2 which is 3 position away from 4;

A white board with writing on it

Description automatically generated

Time complexity is actually K+1 Log K+1 or O(K+1) which can be approximated to K Log K / O(K);

for i = k to N; //until end of array O(N-k) = O(N)

A.insert(minheap.extract());

minheap.add(Original[k]);

while minheap != empty{ O(KLog(K))

A.insert(minheap.extract());

}

Total time complexity is approximately O(N log k);

Create Max heap in O(N) time from 1 to N w/o shiftDown;

Start from max integer and store in array(heap implementation)

Basically do a reverse from 1 to N to N to 1 in an array;

1. Given a 𝐾𝐾 by 𝑁𝑁 matrix 𝐿𝐿, where each row represents a sorted array of 𝑁𝑁 unique floating point values of no fixed precision (thus 𝐾𝐾 sorted arrays), give an algorithm to merge them into 1 sorted array of 𝐾𝐾 ∗ 𝑁𝑁 elements in 𝑂𝑂(𝐾𝐾 ∗ 𝑁𝑁 ∗ 𝑙𝑙𝑙𝑙𝑙𝑙𝑙𝑙 ) time.

Continuation from 9.

Take Note that Each row is already sorted;

A = final array/List;

D is array of size k such that D = [0,0,0] if k =3; Keeps track of the number of elements that have been extracted from min heap from A[Row];

KLog(K)

for i = 1 to K{

minheap.insert(i, L[i][0]); Inserts the first column //i = Which row it belongs to

}

while minheap != not empty{ tree is size Log K; and N\*K extractions; so N\*K Log K;

check = minheap.extractMin()

A.add(check);

D[check.i] +=1 ; D = [1,0,0] for eg.;

if D[check.i] < N{

minheap.insert(i, L[i][D[i]])

}

}

Prof Solution

A screenshot of a computer program

Description automatically generated

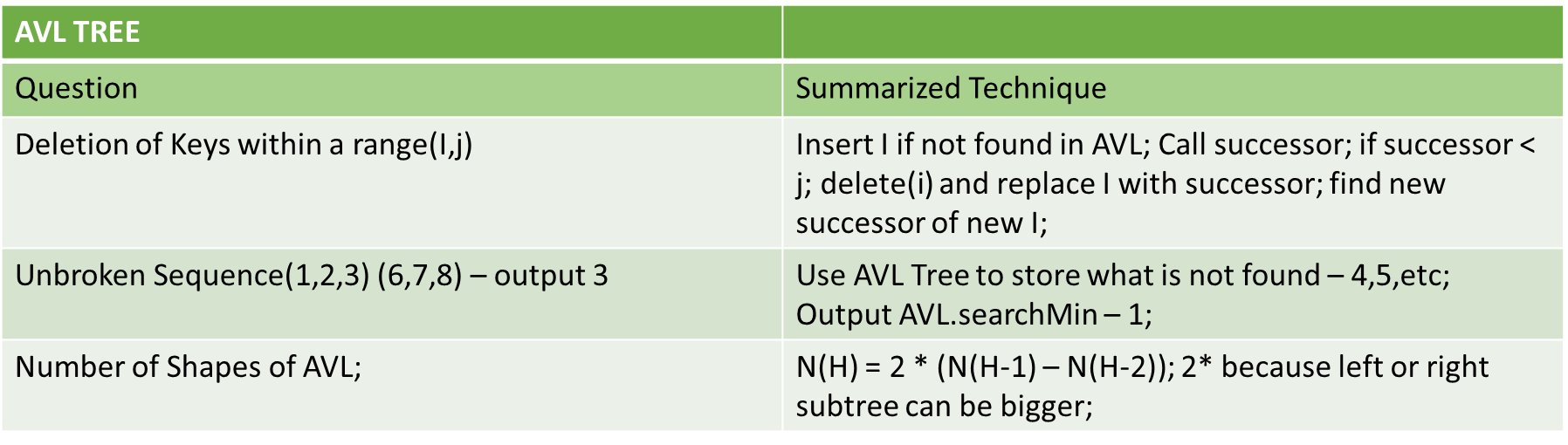
Prof code assumes that our min heap is NEVER 1 element;

Look below – if it becomes 1 element, then we simply go to the go to the last array left;’

A white paper with purple writing on it

Description automatically generated

AVL TREE



1. Given an AVL storing N distinct integer keys, give an algorithm to delete all keys that fall within the range from i to j (i <= j, i and j inclusive) in worst case time <= O(KlogN), where K is the number of keys to be deleted. You cannot use Java library TreeSet or TreeMap in your algorithm. You can only use the AVL operations discussed in lecture. If you need to modify any of the operations, describe your modification.

Search(I,AVL.root); return -1 if not found, else return 1;

If (-1){ insert i}

Snode = successor(i,AVL)

While i < j{

Delete(i)

I = snode;

Snode = successor(i);

}

Stops when I > j

1. Unbroken Sequence, Tutorial 6;

2 Methods; UFDS or AVL Tree;

AVL Tree method;

Create AVL Tree of All Candies NEEDED FROM 1 TO M= labelled AVL; O(MLogM time)

Store what he DOESNT have;

if (T.search(1)){ //means candy 1 not found;

return -1;

}

else{

if T.search(candy J){

T.delete(candy J); can directly delete candy J;

}

return T.searchMin -1;

}

UFDS Method;

Report size of disjoint Set containing 1;

UFDS ufds;

class UFDS;

UFDS();

Largest;

Parent;

UFDS ufds = new ufds(size L);

for i = 1 to L{

LParent = findParent(L);

if L-1 in UFDS 1 and L+1 in UFDS;

ufds.add(L) and L+1 parent;

1.Largest = L+1.largest

else if L-1 in 1 but not L; add L.parent; 1.Largest = L

}

[12 marks] The DDEA (Department to discover extraterrestrial activity) has modified

their tracking of signals from space.

(Recall that DDEA will record a sequence of positive integers representing signals

from space at regular intervals).

The DDEA will first "clean up" each of the recorded signals so that they will be

unique from each other.

Also now the DDEA requires you to implement an ADT that consists of the following

operations:

1. insert(int x): insert a signal x into the ADT as the latest signal being recorded.

2. insertAndTrack(int x): insert a signal x into the ADT as the latest signal being

recorded and also track the largest signal from this point onwards until the next call

to insertAndTrack. This is called a tracking.

For example if the following signals are inserted with the bolded one being inserted

and tracked

1, 2431, 14, 41, 33\*, 8178, 221, 13, 11

then at the point when 11 is inserted, the largest signal being tracked will be 8178

Another example, if the following signals are inserted with the bolded one being

inserted and tracked

231, 1411\*, 13, 131, 111

at the point when 111 is inserted, the largest signal being tracked will be 1411

Note that multiple trackings can be done. An example is as follows

321, 32131, 132\*, 3113, 773111, 13119\*, 3178111, 1313

when 1313 is inserted, the tracking that starts at 132 will have 773111 as the largest

signal being tracked, while the tracking that starts at 13119 will have 3178111 as the

largest signal being tracked. This is because the 1st tracking starting at 132 ended at

773111 and thus will not include 3178111.

3. NumSmalllerThanLatestTracking(): return the number of trackings with the

largest signal less than the largest signal of the latest tracking. If there is no tracking

return -1

For examples:

1, 2431, 14, 41, 33\*, 8178, 221, 13, 11

There is only 1 tracking with largest signal tracked being 8178, so there is no other

tracking thus 0 is returned.

321, 32131, 132\*, 3113, 773111, 13119\*, 3178111, 1313

There are 2 trackings one with largest signal of the 1st tracking being 773111 and the

other (the latest tracking) with largest signal being 3178111. Since the largest signal

of the latest tracking is larger than the largest signal of 1 other tracking so 1 is

returned.

321, 321312\*, 132, 133\*, 3113, 77311, 45, 222\*, 98982

There are 3 trackings, the first one with largest signal being 321312, the second one

with largest signal being 77311, and the third (latest tracking) with largest signal

being 98982. Thus 1 is returned since 98982 is only bigger than 77311 and not

321312.

4. delete(): This will delete the latest recorded signal. If there is a tracking associated

with the signal (i.e the signal was inserted using insertAndTrack()) remove the

tracking too.

Implement each of the above operations so that they will run in worst case O(logn)

time, where n is the current number of signals recorded.

Ans:

Use a Stack S and AVL A:

S stores an entry for each signal as a pair <largest, tracking>. tracking is 1 if the signal is the

start of a new tracking, 0 otherwise, and largest is the value of the largest signal so far for

the tracking that this is signal is associated with. If the signal is not associated with any

tracking largest is -1.

A stores has a node for each tracking and the key for the node is the current largest signal

associated with the tracking.

Let z be a reference to the node representing the latest tracking in the AVL, which is

initialized to null

1. insertAndTrack(x)

a) S.push(<x,1>)

b) Create a node v containing x.

c) Insert v into the A → O(logn)

d) z = v

Total time complexity = O(log n)

2. insert (x)

a) if z is null, S.push(<-1,0>)

b) else S.push(<max(S.peek.largest,x),0>)

c) if z is not null and z.key < x

Delete z from A → O(log n)

Create new node v containing x

Insert v into A → O(log n)

z = v

Total time complexity = O(log n)

3. NumSmalllerThanLatestTracking()

a) If A is empty return -1

b) Return (rank of z)-1 → O(log n)

Total time complexity = O(log n)

4. delete()

a) If S.peek().tracking == 0

S.pop()

If S is not empty and -1 < S.peek().largest < z.key

Delete z from A → O(log n)

Create new node v containing S.peek().largest

Insert v into A → O(log n)

z = v

else if S is empty

z = null

clear A

b) else

S.pop()

Delete z from AVL → O(log n)

If A is not empty

Find node v where key == S.peek().largest → O(log n)

z = v

else

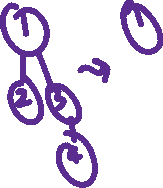
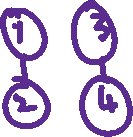
z = null

Total time complexity = O(log n)

Avl TREE + STACK

UFDS

UnionSet eg



GRAPH STRUCTURE

A screenshot of a green and black text

Description automatically generated

1. Given an unweighted graph (with at least 1 edge) stored in an adjacency list, give an algorithm that will return 1 if it is an undirected graph, 2 if it is a directed graph and 3 if it is a hybrid graph (a mix of directed and undirected edges). Your algorithm must run in average time <= O(V+E) where V is number of vertices and E is the number of edges in the graph. \*Note: bi-directed edges will not be present in the graph given.

Form edgelist using hashSet or ArrayList

For (U in adjList){ O(V+E) Time

For (V in adjList.get(V)){

Hashset.add(u,v);

}

} #EdgeList

A = [0,0] as flag; A[0] = undirected, A[1] = directed

For (U in adjList){ O(V+E) Time

For (V in adjList.get(V)){

If Hashset.contains(u,v){

If (Hashset.contains(v,u)){ A[0] = 1; continue; } // implies undirected

Else{A[1] = 1};

}

If A[0] & A[1] == 1; return 3

Else return 1 or 2 respectively;

Time complexity; in AdjList, Loops through all Vertex and Edges with is O(V+E) time – If directed its O(V+E) but undirected is O(V + 2E) which is O(V+E);

If AdjMatrix = O(V^2);

1. There is a game where you have N dots numbered from 1 to N, and for some pairs of dots A,B there is an arrow going from A to B (if there is an arrow going from A to B then there will not be an arrow going from B to A).

The objective of the game is to place the pencil at some starting dot and trace the arrows in such a way as to go through as many dots as possible without lifting the pencil (you may traverse some arrows multiple times to achieve this).

Given there is at least 1 starting dot A' which allows to you to trace the arrows in the above stated way and go through all the other dots, model the game as a graph and give the most efficient algorithm you can think of in terms of worst case time complexity to return one such A'.

Graph Modelling Algo – Name the vertex and edges and whether its weighted/unweighted or undirected/directed;

1st Algo O(V(V+E)) time algo: Run counting component algorithm, once on each vertex as source. If for any particular vertex as source the number of component is 1 then that vertex is an A'

CC Algo – DFS and label which number it belongs to

Do a for loop to check the number of each CC;

2nd Algo Similar to Dominos THA, do counting of SCC’s without incoming edge, Toposort via DFS or BFS which is O(V+E) time. Use a stack to keep track and return the top of the stack

Via DFS it’s a RECURSIVE function;

Add to stack at the return line;

We can use a double stack if we are randomly choosing a vertex to do topoSort

A drawing of a diagram

Description automatically generated A red line with black text

Description automatically generated with medium confidence

A whiteboard with red writing

Description automatically generated

1. Routers in a network are often susceptible to failure. Dr Ferdaus of Ferulock fame/infamy has developed his own protocol to test if a router in a network is still functioning.

Given a graph representing the network, vertices are the routers (there are V number of them numbered from 0 to V-1) and the routers are connected by E directed edges ( 𝑉𝑉(𝑉𝑉−1) 10 ≤ 𝐸𝐸 ≤ 𝑉𝑉(𝑉𝑉−1) 2 ). For any pair of router A and B connected by a directed edge from A to B with integer weight W(A,B) (W(A,B) >= 1), A will send a ping request to B at intervals of W(A,B) seconds (you can assume the time to send the request to B is 0 seconds). If B is alive it will have to send a default message back to A. All the routers in the network is synchronized to a global clock, so they will start counting of their intervals to send pings at the same time.

Now given that the graph as described above is stored in an adjacency list, give the best algorithm in terms of worst case time complexity you can think of to answer the following query (there can be many of such queries):

RequestList(i,K) - Return the 1st K vertices that sends a ping request to vertex i ordered by increasing time of request. Number of vertices sending pings to i <= K <= V. If two different vertices sends a request at the same time, they should be ordered by increasing vertex number.

If required, you can perform pre-processing that takes no more than O(V+E) time. Describe the algorithm for your pre-processing too.

In the example graph given below that represents a network of routers, RequestList(2,4) will return {3,4,3,1} as the first 4 vertices sending pings to vertex 2 in order of increasing time of request. The time of their requests are {2 secs, 3 secs, 4 secs, 5 secs} respectively.

A diagram of a triangle with numbers and lines

Description automatically generated

Both are O(V+E)

Do preprocessing to QUICKLY ACCESS the edges that are pointed to 2,

Two ways to do this, do BFS and store into Edge List

Let Visited be an array initialized to all 0

EL = edgelist that is empty

Let Q be a Queue;

StartingNode = 0;

Q.add(StartingNode);

While (Q not empty){

Node = q.pop();

For (neighbour V of U){

If (V == i){

EL.add(W,U,V); //don’t set to visited so it can continue to be visited;

Else{

q.add(V);

visited[V] = 1;

}

Visited[Node] = 1; //true

}

Return EL;

OR

Like in Kosaraju, do a TRANSPOSE of the graph and store edges is adjListT.

adjList’ = empty adjList

For (U in adjList)

For (neighbour V in adjList)

adjList’[v] = (w,u)

RequestList(I,k)

PQ pq;

ArrayList T;

Intialization

For each neighbour V of I in adjList’ or EL{ O(K) time which is the number of neighbours

Pq.add(w,v,i); // Log(K) time

Use comparator function to order then according to weight;

} Overall KLogK time;

Algo

While K!= 0{ O(K) since the rest are O(1) Time;

Next = pq.poll();

T.add(Next.w);

Next.w = w+ adjList[next.i].get[next.v]

Pq.add(Next);

k--;

Overall O(KLogK + K) = O(KLogK) NOT INCLUDING PREPROCESSING ;

MST

1. Given the graph data structure 𝐷 as modelled in a), island 𝐴 of an attacker and island 𝐵 of the enemy to attack, give the best algorithm you can think of to find the best valid path to get from 𝐴 to 𝐵 such that the weakest bridge (based on its difficulty value of being destroyed) is maximized

and output the difficulty value of that weakest bridge.

If there is no valid path from 𝐴 to 𝐵 output -1. [9 marks]

Use Kruskals(Uses UFDS) but Prims uses PQ and T {Set}

Let weakest = 0; //weakest bridge

Get EdgeList(U,V,W); Sort by descending order – to find maximin;

Initialize UFDS of all Vertices //which uses int[] parent array;

For each (u,v,w) in edgeList{

If (u.nation = A & v.nation = A){

If (!UFDS.isSameSet(u,v){

UFDS.unionSet(u,v);

Weakest = w //because we are doing in descending order; so definitely less

Else(if u/v.nation = A & u/v.nation = B){

If (!UFDS.isSameSet(u,v)){

UFDS.unionSet(u,v);

Weakest = w + BREAK;

UFDS.isSameSet(ATOCHECK,BTOCHECK);

If true;break and return weakest;

Else return -1 if after for loop and not sameSet;

Same time complexity as standard Kruskal’s : O(MlogN)

A diagram of a diagram with red circles and numbers

Description automatically generated

1. Assuming 𝐶 is the set of islands owned by the attacker and given a fixed enemy island 𝐵𝐵. For each of the islands 𝐴’ in 𝐶 with a valid path to B, the best path (path that maximizes the minimum edge) has been found and edges in the best path from 𝐴’ to 𝐵 has been put into an edge list 𝐸. However, the enemy has just destroyed a weakest bridge (𝑢𝑢’, 𝑣𝑣’) along the best path from an attacker island 𝐴𝐴 to 𝐵𝐵, so the edge representing that bridge has been removed from 𝐸. Given 𝐷 and 𝐸, now provide an algorithm to find the next best valid path from 𝐴 to 𝐵 in 𝑂(𝑁 + 𝑀) time. Specify any modifications you need to make to the graph 𝐺(𝑉, 𝐸) and 𝐷 and any extra data structures you use. Again, if there is no more valid path output -1, otherwise output the weight of the weakest bridge in that next best valid path. [10 marks]

A whiteboard with a diagram and numbers

Description automatically generated with medium confidence

Find the best edge that joins two CC’s

* Convert D from adjList to EdgeList O(V+E)

EL’ = edgeList

For (u in adjList){

For (neighbour v in u){

EL.add(u,v,w);

}

}

* DFS for counting component algorithm. O(V+E);
* Num of CC = 1; return deleted edge weight as it implies cycle exist

Else;

Best edge = infinity;

For edge(u,v,w) in EL’{

If (edge(u,v,w) != (u’,v’,w)(destroyed bridge) and u and v diff components){

If (u,v,w) < best edge; bestedge = (w)

Time Complexity is O(V+E) above;

Finding Lightest Edge between two components A and B

DFS on A and B respectively to label which component they belong to;

Iterate through edge list to find the lightest edge connecting components A and B;

19. [10 marks] You are given an undirected weighted connected graph G with V vertices and V+c number of edges, where c is some constant > 1, and where V number of the edges are of weight 1 and c number of them have weights > 1. G is stored in an adjacency list AL. Give an algorithm to find the cost of the MST of G in worst case O(V) time.

Recall MST has V-1 Edges;

Counting Components; if edge >1; ignore, so we only account for edges that == 1; O(V+c) = O(V) since c is a constant; Label each vertex to which component it belongs to;

If same component; return V-1

If broken into 2 components; implies that there is an edge >1 that connects two components;

Add next largest edge that connects 2 components or two components;

Run kruskals on edge list that connects the two component; BUT SINCE C IS A CONSTANT; O(1) TIME

1. Given a graph G that is a DAG with V vertices and E edges (where E = O(V)) stored in an adjacency list, you want to find the weight of the largest edge along the minimax path from a given source vertex A to a given destination vertex B. Give the best algorithm you can think of in terms of worst case time complexity to do it. You may assume there is at least 1 path from A to B

Run kruskals or prims algo which takes O(ELog(V)) to get MST

DFS or BFS from A and check edges which are in MST, until it reaches vertex B

so Time is V+E + ELog(V) = O(V+E)

Corrections : Don’t need to run DFS/BFS, simply run PRIMS STRICTLY using A as source, Store the largest edge as a variable, return that when we reach B; which takes (ELogV) / VLogV;

Second Algo; Uses SSSP Algo instead – I don’t quite get this

A whiteboard with writing on it

Description automatically generated

1. TopoSort of DAG using DFS and Stacks;
2. In Topological ordering, run 1-Pass Bellman Ford

Distance[] array;

For each (u in topological ordering){

For each(neighbour v of u){

D[v] = min(D[v], MAX(D[u], w(u,v)));

Return D[B]

Time taken is time complexity of 1-pass bellman ford = O(V+E) = O(V) since E = O(V)

A white board with red writing

Description automatically generated

SSSP

There are no graphs where the SSSP spanning tree is also the same as the MST (in terms of the edges used). False. If the graph is a tree then the SSSP spanning tree (from any source vertex) will be the same as the MST

1. The salesman from tutorial 11 has begun another round of travelling around different cities and peddling his wares. This time he has set aside funds to pay the toll fee for every city he passes when getting from some source city A to some destination city B. Of course he still wants the shortest route to get from A to B as time is of the essence. However he has calculated that he has only enough money to pay the toll fee of at most K cities where 1<= K <= 10, thus he cannot pass through more than K cities when getting from A to B (including A and B themselves). Given the value K and a graph G with V vertices and E edges, where the vertices are cities and bidirectional edges are roads connecting pairs of cities, and edge weight is the travel time (same in both direction), give the best algorithm you can think of in terms of worst case time complexity to find the cost of the shortest path the salesman should use to get from some city A to some other city B that does not involve more than K cities. If no such path exists output "no valid path from A to B".

Using Tut 11 Algorithm (Easier); Simply find the shortest path with the shortest distance, CHECK if H < K;

Let H be array to store number of hops;

Let D be array to store distance;

In the PQ, store triplets (D[u], H[u], Vertex);

A group of red letters

Description automatically generatedPQ = (0,0,A), (∞,∞,B), (∞,∞,C), (∞,∞,D), (∞,∞,E);

1. PQ = {(0, 1, A),(∞, ∞, B),(∞, ∞, C),(∞, ∞, D),(∞, ∞, E)}. Process (0, 1, A).

D = [0,7,10,∞,∞]

H = [0, 1,1,∞,∞]

1. PQ = {(7, 2, C),(10, 2, B),(∞, ∞, D),(∞, ∞, E)}. Process (7, 2, C).

D = [0,7,10,17,∞]

H = [0, 1,1,2,∞]

1. PQ = {(10, 2, B),(17, 3, D),(∞, ∞, E)}. Process (10, 2, B).

D = [0,7,10,17,25];

H = [0, 1,1,2,2];

4. PQ = {(17, 3, D),(25, 3, E)}. Process (17, 3, D). Note that it will not change (25, 3, E) as

(25, 3, E) is better than (25, 4, E) from (17, 3, D).

D = [0,7,10,17,25];

H = [0, 1,1,2,2];

17,3,D – results in 25,4,E but check H[E] = 2 > 4;(Lazy to change but u get the idea)

5. PQ = {(25, 3, E), (25, 4, E)}. Process (25, 3, E), but E has no outgoing edges.

6. PQ = (25, 4, E}.

End Algo;

If H[E] > K; return no valid path;

//Sorts according to edgeweight then H

PriorityQueue pq;

Array D[];

Array H[];

D[0] = H[0] = 0;

Rest of array is ∞;

While (pq not empty){

u = pq.poll();

For each neighbour V{

weight = adjList[u]

D[v] = min(D[v], D[u] + weight);

If D[v] == D[u] + Weight; H[v] = min between H[u]+1 and H[v]

H[v] = min(H[v], H[u]+1); //infinite vs 1 for starting nodes

Pq.add(D[v],H[v],v);

}

}

If H[E] < k; return D[E];

Else return no valid path.

11 while Q is not empty:

12 u := vertex in Q with smallest distance in dist[] and has not been visited;

13 remove u from Q;

14 visited[u] := true

15

16 for each neighbor v of u:

17 alt := dist[u] + dist\_between(u, v);

alt\_edges := dist\_edges[u] + 1; //Note the increment by 1

if (alt = dist[v] && alt\_edges < dist\_edges[v])

previous[v] := u;

dist\_edges[v]= alt\_edges

18 if alt < dist[v]:

19 dist[v] := alt;

dist\_edges[v] := alt\_edges;

20 previous[v] := u;

21 if !visited[v]:

22 insert v into Q;

Alternative: Running modified Dijkstra;

initSSSP(A) = initialize D[V][K] to ∞ except D[A][1] which is 0;

PQ.enqueue(0,1,A); Similar to above algo;

Now scan D[B][1] to D[B][K] to find the smallest among them and return that. If all are

still +inf means there is no valid path involving <= K vertices from A to B

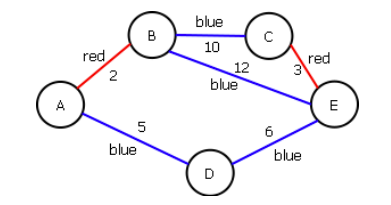
Colored paths [16 marks]

City Z is a very unique city. Its roads are painted either red or blue. Junctions and roads

of Z can be represented as an undirected weighted graph, where in addition to the

weight (positive value), an edge which represents a road also has an extra parameter

color which indicates what color the road is. An example is shown below



Only process those of same color;

Time complexity of AdjLIST O(E+V)\*O(LogV) which is O((E+V)\*LogV)

O(E + VLogV) using Fibonacci Heap;

For ADJMATRIX is O(V^2)

PriorityQueue pq; which is a min heap

Array[] Distance;

Initialize Distance to inf except Distance[x] = 0;

Create adjList of City Z via BFS or DFS;

Starting sNode = X;

Color Xcolor = X.color;

pq.enqueue(X,0);

while (pq!= empty){

    u = pq.dequeue();

    for (neighbour v of u){

        if (v.color == X.color){ //same color

            Distance[v] = min(Distance[v], Distance[u] + w(u,v) from adjList);

            pq.enqueue(v,Distance[v])

        }

    }

}

return Distance[Y];

[10 marks] District X is the central business district of city Y and people from other districts in Y will go to X for work in the morning and go back home in the evening. These people are so familiar with how to get from their home to X that they will always use the fastest route to get there. You may assume that is only 1 fastest route from each district to district X. In fact all fastest route to get from any district to X is used by someone. One day in the middle of the night, there was an earthquake and many of the roads in city Y have been damaged. In the morning, you hear from the news that the roads used by people to get from their district to X are the most badly affected since those roads are already under stress from over usage. Since you still have to get to work in district X (you are from district Z), you are worried that some roads along your usual route may not be passable. Thus you want to find an alternate route which is the following: The fastest route made up of only roads which is not used by anyone to get from their district to X. The road network of city Y is represented as a graph of V vertices and E edges where each district is a vertex and and each road (which is 2-way) linking 2 districts are undirected edges linking the respective vertices. The weight of each edge is a positive integer value representing the time to travel the edge/road. It is guaranteed that you can always get from any district to any other district in city Y via the road network. Come up with the most efficient algorithm to find the best alternative route as given above. If there is no such route return “no such route”. State the time complexity of your algorithm. You may assume the graph is stored in an adjacency list AL, and you will be given the 2 vertices X and Z (representing district X and Z respectively)

Technique; Double SSSP;

A white board with red text and numbers

Description automatically generated

17) Find the shortest path of ALL places to X;

remove edges from shortest path from AL;

Rerun SSSP algorithmn on new AL;

Repeat FOR ALL Districts; --> Least Efficient therefore, Just use X as a source;

PriorityQueue pq;

Distance Array D; Initialize to infinite except for Z;

Path Array P; Initialize to Respective index; eg Z set to Z;

Finding SSSP with the path stored via Dijkstra;

pq.add(Z);

while (pq!= empty){ O((V+E)logV)

    u = pq.dequeue();

    for (neighbour v of u){

        if (D[v] < D[u] + w(u,v)){

            D[v] = min(D[v], D[u] + w(u,v)); // or directly do it

            P[v] = u;

        }

    }

}

Make new AL^ O(V+E) time

AL^ does not include edges from P[X]; till P[Z] = Z; //Store these edges in Edge List; but it can take O(V^2) to make edgelist

Better Approach

Iterate through AL O(V+E) time;

if p[i] = j and p[j] = k and j!=k; include e(i,j);

Rerun SSSP USING Z as the source; O((V+E)logV);

If D[Z] = infinite, return no such route;

If we have a parent array => Set to either all -1

Don’t set to itself; 1,2,3,4; 🡪 Could cause issues; if we have no SSSP from 1 to 4; p[4] = 4; we only stop our algo when p[1] = 1; to find the shortest path from the destination to 1; via backtracking

APSP

Floyd Warshall;

for i = 0; i < V; i++{ i is my intermediate node

    for j = 0; j < V; j++{

        for k = 0; k < V; k++{

            adjMatrix[j][k] = min(adjMatrix[j][i] + adjMatrix[i][k], adjMatrix[j][k]);

        }

    }

}