CSCI 270

Discussion Week 4

Let S be a subset of nodes that is neither empty nor equal to V, and let edge e(v,w) be the minimum cost edge with one end in S and the other end in V-S. Then every MST contains the edge e.

Proof

Let T be a spanning tree that does not contain e. The ends of e are v and w. Because T is a spanning tree we can say that there is a path from v to w in T.

Following the path v to w in T we assume that the first node in V-S along that path is w’. Let e’ = (v’,w’) be the edge joining S and V-S in T.

If we exchange e’ with e, we get a set of edges T’. Clearly T’ will be connected because any path in T that used e’ can now be rerouted to use e. This can be done by tracing the path between v and v’ and w and w’ and using that.

Also, T’ would be acyclic because when we add e to T we create a cycle which has e’. T’ is created by then removing e’ from T+{e}. this proves that T’ is a spanning tree.

We know that e is the cheapest edge and so cost of e < cost of e’. Thus, by exchanging we improve the cost of the spanning tree.

Proof for Kruskal's

Consider any edge $e = (v, w)$ added by Kruskal's Algorithm, and let S be the set of all nodes to which v has a path at the moment just before e is added. Clearly v belongs to S and w belongs to V-S. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal's Algorithm. Thus e is the cheapest edge with one end in S and the other in $V - S$, and so it belongs to every minimum spanning tree.

Proof for Prim’s

For Prim’s Algorithm, in each iteration of the algorithm, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node *v* and edge *e* are added that take the minimum edge with one end in S and the other in V-S. So by the Cut Property, it is in every minimum spanning tree.

Proof of Reverse-Delete

Cycle property:

The highest weight edge (assuming unique edge costs) in a cycle cannot belong to any MST

In every iteration of reverse delete, we only removes highest weight edges in a cycle, so it only removes edges that cannot belong to any MST according to the above property.

Implementation of Dijkstra’s

```
S = NULL
Initialize priority Queue Q with all nodes V with d(v) as the key value(all d(v) are infinite except d(s) = 0)
While S!=V:
    v = Extract-Min(Q)
    S = S U {v}
    for all u in adj(v):
        if d(u) > d(v) + cost of edge e:
            Decrease_Key(Q, u, d(v) + cost of edge e)
    end for
end while
```

Implementation of Prim’s

```
S = NULL
Initialize priority Queue Q with all nodes V with d(v) as the key value(all d(v) are infinite except d(s) = 0)
While S!=V:
    v = Extract-Min(Q)
    S = S U {v}
    for all u in adj(v):
        if d(u) > **cost of edge e**:
            Decrease_Key(Q, u, **cost of edge e**)
    end for
end while
```

Implementation of Kruskal’s

```
Create an independent set for each node
A = NULL
Sort edges in non-decreasing order of weight
for each edge (u,v) in E, take in order, if u and v not in the same set
    Add (u,v) to A
    merge the two sets
    endif
endfor
```

Union-find data structure

Make set	O(1) for set size = 1
Find set	O(1) or O(logn)
Union	O(logn) or O(1)

Complexity Analysis

Making a set for each of the nodes - O(n)

Sorting the edges = O(mlogm)

Checking if two sets are same and if not merging the two sets takes O(logn). this step is performed O(m) times

Hence, TC = O(n) + O(mlogm) + O(mlogn) = O(mlogm)

Problem

Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Analyze the amortized runtime of the operation in question.

Solution

This is exactly the push-back function, and has the same analysis. Over the first n operations, we will have $\log_2 n$ operations where i is a power of 2 , and these will take $\sum^{\log n} 2^i = \Theta\ (2^{\log n}) = \Theta(n)$ time.

The rest of the $n - \log_2 n$ operations each take constant time, for a total of $n - \log_2 n = \Theta(n)$ time.

So, the total time is $\Theta(n) + \Theta(n) = \Theta(n)$. If we amortize this cost over all n operations, we get $\Theta(n)/n = \Theta(1)$ amortized runtime.

Problem

You are given a minimum spanning tree T in a graph G = (V, E). Suppose we remove an edge from G creating a new graph G1. Assuming that G1 is still connected, devise a linear time algorithm to find a MST in G1.

Solution

Suppose the removed edge is named e_r.

Case 1: If e_r is not in T. The MST is T. This takes linear time.

Case 2: If e_r is in T. Once we delete an edge from T, the tree becomes disconnected. We need to find the minimum weight edge that connects two components, T1 and T2. Pick any component say T2 and find all edges going to T1. Among them choose the one which has the smallest cost. Runtime O(E).

Problem

Hardy decides to start running to work in San Francisco city to get in shape. He prefers a route that goes entirely uphill and then entirely downhill so that he could work up a sweat uphill and get a nice, cool breeze at the end of his run as he runs faster downhill. He starts running from his home and ends at his workplace. To guide his run, he prints out a detailed map of the roads between home and work with k intersections and m road segments (any existing road between two intersections). The length of each road segment and the elevations at every intersection are given. Assuming that every road segment is either fully uphill or fully downhill, give an efficient algorithm to find the shortest path (route) that meets Hardy's specifications. If no such path meets Hardy's specifications, your algorithm should determine this. Justify your answer.

Solution

Use a directed graph to represent the map of the city. The edge direction will indicate the uphill direction of the road segment. Then run Dijkstra's algorithm with Home as the starting point. This will find us the shortest paths and shortest distances from Home to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S1.

Run Dijkstra's algorithm with Work as the starting point. This will find us the shortest paths and shortest distances from Work to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S2. $S = S1 \cap S2$

If nodes corresponding to Work or Home fall into S remove them - If S is null then there is no such solution, otherwise, for each node i in S, add up the shortest uphill distance from Home to i and shortest uphill distance from Work to i. Select the node j in S that gives us the smallest total distance. The path we are interested in will be the shortest uphill path from Home to j followed by the shortest downhill path from j to Work.

Problem

A network of n servers under your supervision is modeled as an undirected graph G = (V, E) where a vertex in the graph corresponds to a server in the network and an edge models a link between the two servers corresponding to its incident vertices. Assume G is connected. Each edge is labeled with a positive integer that represents the cost of maintaining the link it models. Further, there is one server (call its corresponding vertex as S) that is not reliable and likely to fail. Due to a budget cut, you decide to remove a subset of the links while still ensuring connectivity. That is, you decide to remove a subset of E so that the remaining graph is a spanning tree. Further, to ensure that the failure of S does not affect the rest of the network, you also require that S is connected to exactly one other vertex in the remaining graph. Design an algorithm that given G and the edge costs efficiently decides if it is possible to remove a subset of E, such that the remaining graph is a spanning tree where S is connected to exactly one other vertex and (if possible) finds a solution that minimizes the sum of maintenance costs of the remaining edges.

Solution

First we need to check the possibility of node S having only one neighbor in a spanning tree of the underlying graph. The best way to check this is to remove node S and all its adjacent edges to form a graph G′ . If G′ is a connected graph, then we can claim it is possible to have a spanning tree where node S has only one neighbor. To check the connectivity of G′ , the simplest way is to run a DFS or BFS algorithm on G′.

Considering that G′ is connected, we need to find the spanning tree that minimizes the maintenance cost. Therefore we need to find the MST with the additional constraint that S should be a leaf. Therefore, we remove S and all its adjacent edges to form G′ . We run Prim’s (or any other MST algorithm) to find the MST of G′ . Among all edges adjacent to S, we find the one with the minimum maintenance cost and connect S to the MST using this edge. The resulting graph will still be a spanning tree and in this spanning tree S will be a leaf.

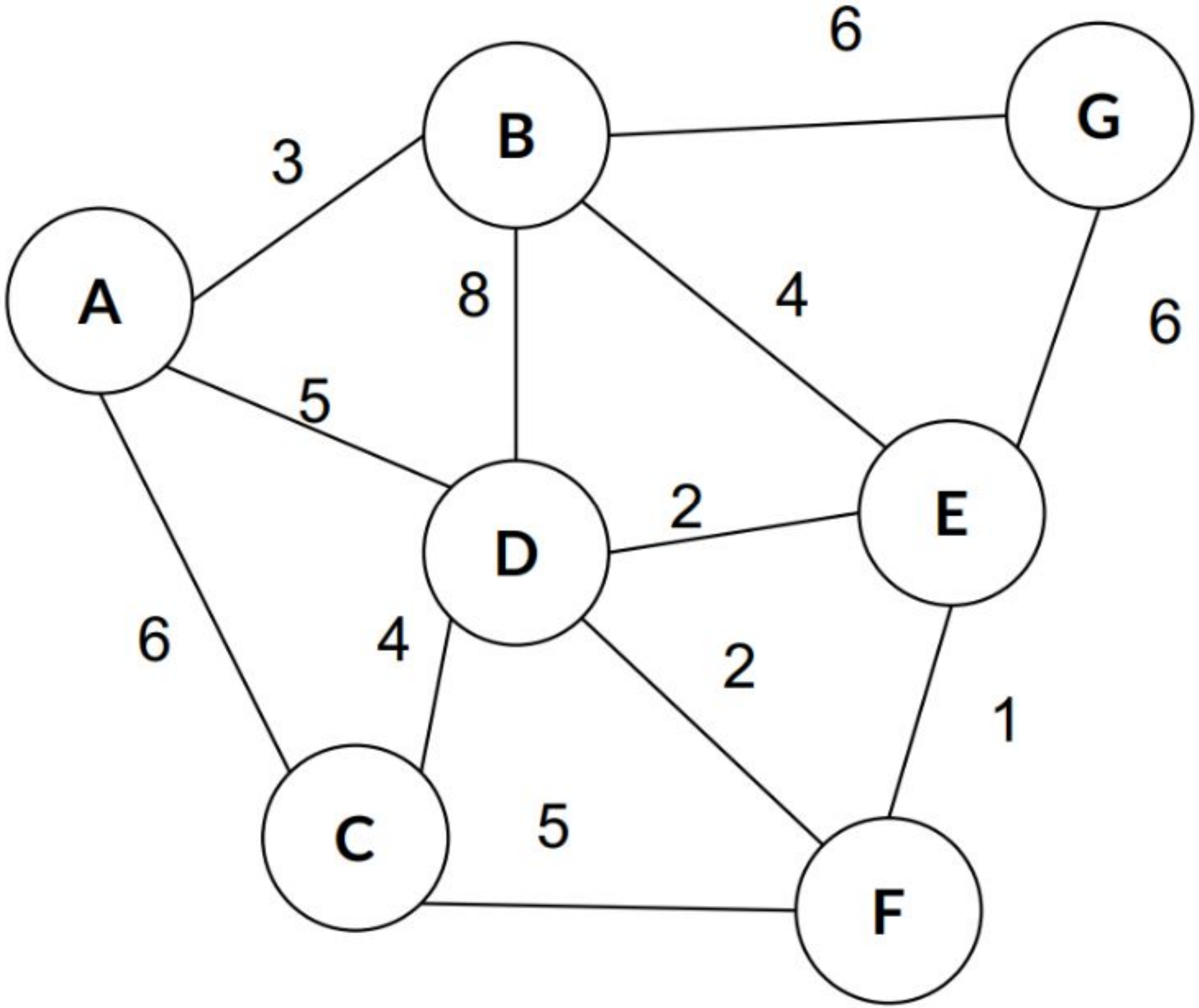
Problem

Considering the following graph G:

a) In graph G, if we use Kruskal's Algorithm to find the MST, what is the third edge added to the solution? Select all correct answers

b) In graph G, if we use Prim's Algorithm to find MST starting at A, what is the second edge added to the solution?

c) What is the cost of the MST in the Graph?


A
B
C
D
E
F
G
3
6
8
4
6
5
2
6
4
2
1
5

Solution

(a) A-B

(b) B-E

(c) 20