Design and Analysis of Algorithms

Q1) You are given a weighted undirected graph G(V, E) and its MST T(V, E'). Now suppose an edge (a, b) ϵ E' has been deleted from the graph. You need to devise an algorithm to update the MST after deletion of (a, b).

Describe an algorithm for updating the MST of a graph when an edge (a, b) is deleted from the MST (and the underlying graph). It's time complexity must be better than running an MST algorithm from scratch. State and explain the time complexity of your algorithm. Analyze time complexity of your algorithm.

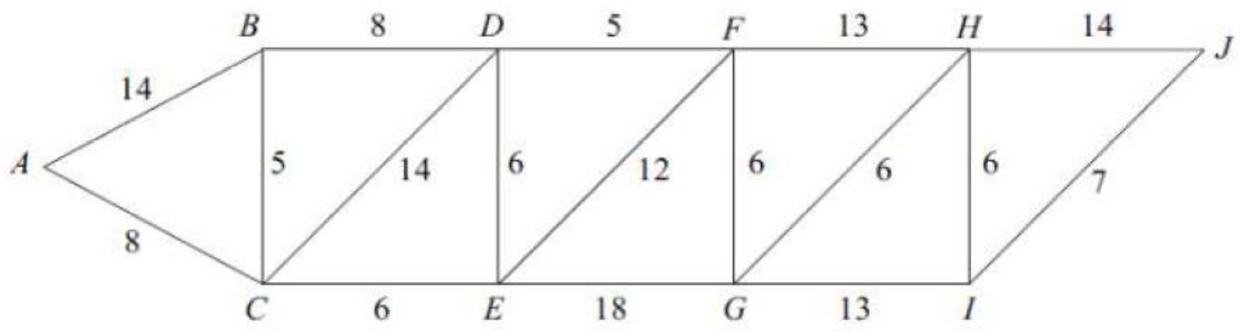
You can assume that all edge weights are distinct, that the graph has E edges and V vertices after the deletion, that the graph is still connected after the deletion of the edge, and that your graph and your MST are represented using adjacency lists.

Solution

The following algorithm has time complexity O(E). Run DFS twice on what remains of the MST, once starting on a and once starting on b to determine the two connected components A and B, which you store in two HashSets. This takes O(E). Then iterate through all edges going out from the elements of A. If an edge leads to an element of B (can be check in O(1) using HashSet), check whether it is the cheapest edge so far. If it is cheaper than any other edge so far, store it. So, finding the cheapest edge can also be done in O(E). In the end, add the cheapest edge to the MST.

Q2) Can you use the DFS algorithm to compute the number of distinct paths between two given nodes, s and t? Two paths are considered distinct if they differ by 1 or more edges. If you think the answer is yes, then provide the pseudo-code for a DFS based algo that computes this number in O(|V|+|E|). If you think the answer is no, draw a graphwith eight vertices in total,with two of its vertices labeleds and t, in which DFS will fail to compute the number of distinct pathsbetween s and t. [5 Marks]

Q3) The following network show times, in minutes, to travel between 10 towns. [3+2 Marks]



a) Use Dijekstra’s algorithm on above figure to find minimum travel time from A to J.

b) State the corresponding route.

Q4) CM of Punjab has decided to build a new road in Lahore. He has a choice of k roads from which he can only build one. However, secretly, CM is only interested in two places in the city: s and t (his office and home). He wishes to build the road which, when added into the city, reduces the cost of the shortest path from s to t the most. He has hired you to write a computer program to tell him which of the k roads to build for this purpose. The input to your program is going to be a directed weighted graph and list of k weighted edges not already in the graph.

(a)Write an algorithm that does the job in O(k(|E|)lg|V|).Note that since k itself is O(|V|^2)this is a pretty slow algorithm. [4 Marks]

Solution

Run Dijkstra on the graph and find shortest paths by keeping house of CM as source vertex.

```
For(i = 1 to k)
  Add road i to the graph
  Run Dijkstra on the graph and find shortest paths by keeping house of CM as source vertex.
  Update shortest path from CM house to his office is less than the previous, also save road i.
```

(b) CM is in a hurry. He wants you to write an algorithm that works in O (k+|E|lg|V|), only. Can you do it?

1) Run Dijkstra by keep s as source. Save distances from s to any other vertex v in d1[v] $O((E)\lg V)$
2) Calculate reverse of the graph G^R in linear time. O(V+E)
3) Run Dijkstra by keeping t as source in G^R. Save distances from t to any other vertex v for G^R in d2[v] $O\big((E)\lg V\big)$
4) For(i = 1 to k)

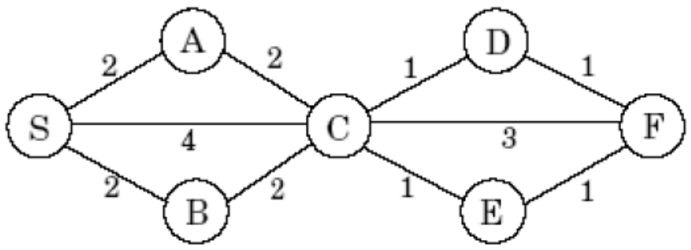
```
    Road i is from u to v
    If(d1[u] + w(u,v) + d2[v]) < d1[t]) then update shortest path to t and save road i.
```

Total Time: O (k+|E|lg|V|),

Q5) In cases where there are several different shortest paths between two nodes (and edges have varying lengths), the most convenient of these paths is often *the one with fewest edges*. For instance, if nodes represent cities and edge lengths represent costs of flying between cities, there might be many ways to get from city s to city t which all have the same cost. The most convenient of these alternatives is the one which involves the fewest stopovers. Accordingly, for a specific starting node s, define

best[*u*] = minimum number of edges in a shortest path from *s* to *u*

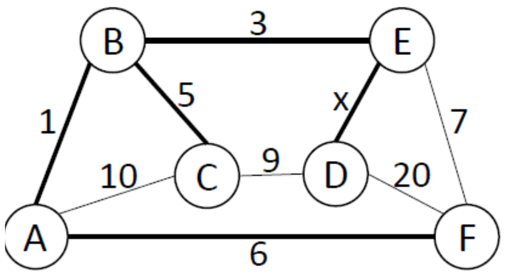
In the example below the best values for nodes S, A, B, C, D, E, F are 0,1,1,1,2,2,3 respectively.



Give an efficient algorithm for the following problem and analyze its time complexity.
Input: Directed Graph G = (V, E); positive edge lengths l_e; starting node *s* ∈ V
Output: The values of best[*u*] should be set for all nodes *u* ∈ V

Q6) For the following graph the bold edges form a Minimum Spanning Tree. What can you tell about the range of values for x?

a. Less than 7, greater than 10
b. Less or equal to 9
c. Less or equal to 3
d. Less or equal to 7



Solution:

Less or equal to 9
The edge with weight x connects D to rest of the spanning tree. It cannot be greater than 9 since otherwise the edge C,D with weight 9 would be selected to connect D.

Q7) Suppose that all edge weights in a graph are integers in the range from 1 to |V|. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Solution

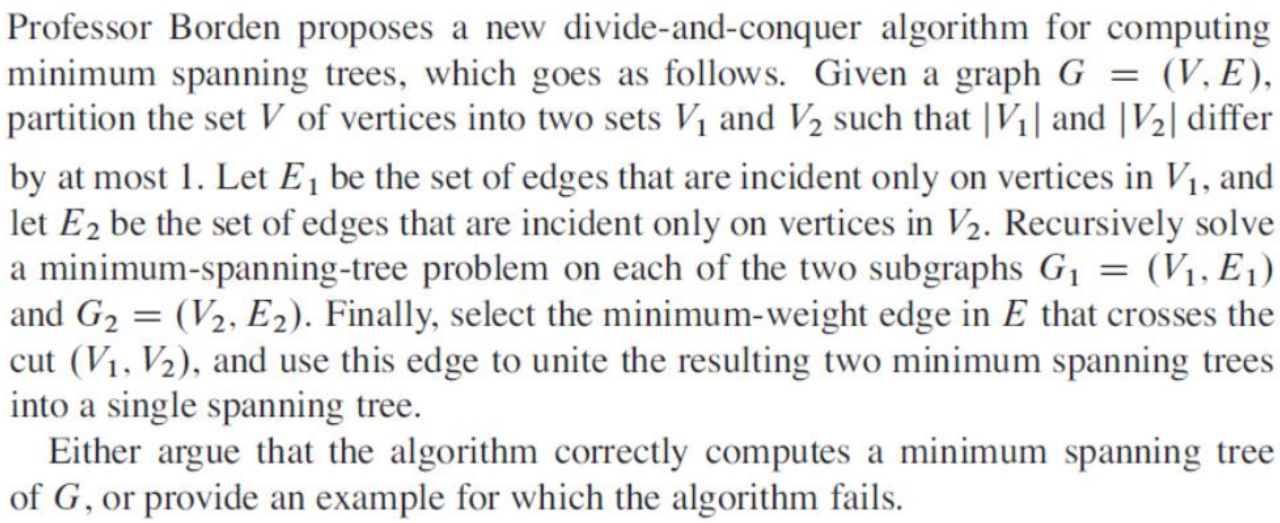
If w is a constant we can use **counting sort**

• Sorting the edges: O(E lg E) time.

• O(E) operations on a disjoint-set forest taking O(Eα(V)).

The sort dominates and hence the total time is O(E lg E). Sorting using counting sort when the edges fall in the range 1, . . . , |V | yields O(V + E) = O(E) time sorting. The total time is then O(Eα(V)). If the edges fall in the range 1, . . . , W for any constant W we still need to use Ω(E) time for sorting and the total running time cannot be improved further.

Q8)

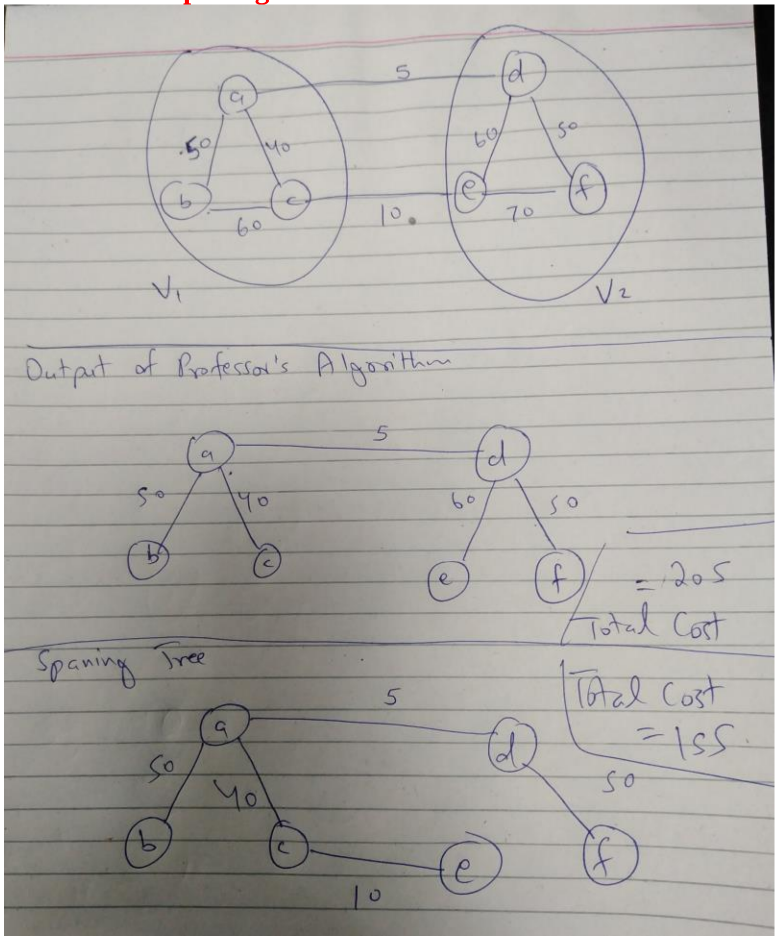


Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

Solution

Counter example is given



Q9)

What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

Solution:

$O(V^2)$ where V = number of vertices

Q10) Given a weighted directed graph in adjacency list representaiton, write an efficient algorithm to calculate in-degree (number of incoming edges), out-degree (number of outgoing edges), sum of weight of incoming edges, and sum of weight of outgoing edges for each vertex of the graph. Analyze time complexity of your algorithm. It should run in linear time O(V+E).

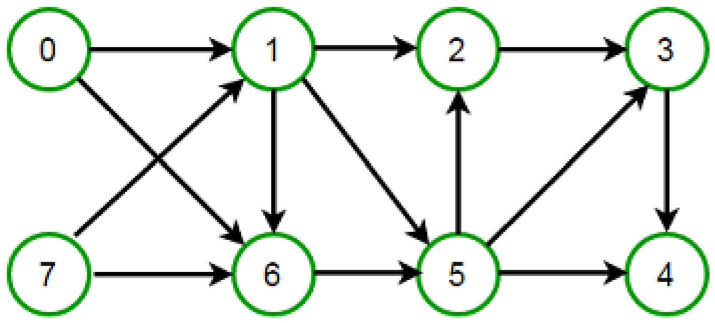
Solution

Run BFS and use 4 different arrays of size n (n = total vertices) to store in-degree (number of incoming edges), out-degree (number of outgoing edges), sum of weight of incoming edges, and sum of weight of outgoing edges for each vertex.

```
BFS(G, start)
   Create new queue Q
   Q.push(start)
   color[1..n] = White

   while Q is not empty
      u = Q.pop()
      for each node v adjacent to u
         if color[v] = White then
            color[v] = Black
            indegree[v]++
            outdegree[u]++
            sumOfIndegree[v] += weight(u,v)
            sumOfOutdegree[u] += weight(u,v)
            Q.push(v)
```

Time complexity = O(V+E)

Q11) Given a directed graph, find the total number of routes to reach the destination from a given source with exactly m edges. For example, consider the following graph: [10 Marks]



Let source = 0, destination = 3, number of edges m = 4. The graph has 3 routes from source 0 to destination 3 with 4 edges. The solution should return the total number of routes 3. Give an efficient algorithm for solving this problem and analyze its time complexity.

```
0 —> 1 —> 5 —> 2 —> 3
0 —> 1 —> 6 —> 5 —> 3
0 —> 6 —> 5 —> 2 —> 3
```

Solution

The idea is to do a BFS traversal from the given source vertex. BFS is generally used to find the shortest paths in graphs/matrices, but we can modify normal BFS to meet our requirements. Usually, BFS doesn't explore already discovered vertices again, but here we do the opposite. To cover all possible paths from source to destination, remove this check from BFS. But if the graph contains a cycle, removing this check will cause the program to go into an infinite loop. We can easily handle that if we don't consider nodes having a BFS depth of more than `m`. Basically, we maintain two things in the BFS queue node:

- The current vertex number.
- The current depth of BFS (i.e., how far away from the current node is from the source?).

So, whenever the destination vertex is reached and BFS depth is equal to `m`, we update the result. The BFS will terminate when we have explored every path in the given graph or BFS depth exceeds `m`.

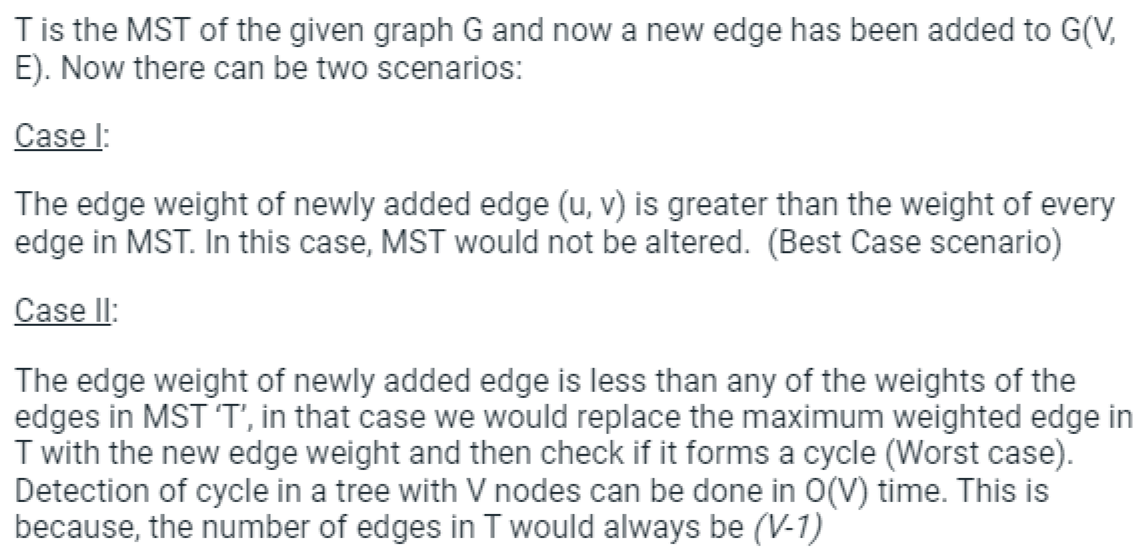
```
int findTotalPaths(Graph const &graph, int src, int dest, int m)
{
    // create a queue for doing BFS
    queue<Node> q;

    // enqueue source vertex
    q.push({src, 0});

    // stores number of paths from source to destination having exactly `m` edges
    int count = 0;

    // loop till queue is empty
    while (!q.empty())
    {
        // dequeue front node
        Node node = q.front();
        q.pop();

        int v = node.vertex;
        int depth = node.depth;

        // if the destination is reached and BFS depth is equal to `m`, update count
        if (v == dest && depth == m) {
            count++;
        }

        // don't consider nodes having a BFS depth more than `m`.
        // This check will result in optimized code and handle cycles
        // in the graph (otherwise, the loop will never break)
        if (depth > m) {
            break;
        }

        // do for every adjacent vertex `u` of `v`
        for (int u: graph.adjList[v])
        {
            // enqueue every vertex (discovered or undiscovered)
            q.push({u, depth + 1});
        }
    }

    // return number of paths from source to destination
    return count;
}
```

Time complexity of your algorithm:

O(V+E)

Q12) Let G=(V,E) be a weighted undirected graph and let T be a Minimum Spanning Tree (MST) of G maintained using adjacency lists. Suppose a new weighed edge (u,v) is added to G. Give an efficient algorithm for determining of T is still MST of resultant graph and analyze its time complexity.

Solution



T is the MST of the given graph G and now a new edge has been added to G(V, E). Now there can be two scenarios:

Case I:

The edge weight of newly added edge (u, v) is greater than the weight of every edge in MST. In this case, MST would not be altered. (Best Case scenario)

Case II:

The edge weight of newly added edge is less than any of the weights of the edges in MST 'T', in that case we would replace the maximum weighted edge in T with the new edge weight and then check if it forms a cycle (Worst case). Detection of cycle in a tree with V nodes can be done in O(V) time. This is because, the number of edges in T would always be *(V-1)*