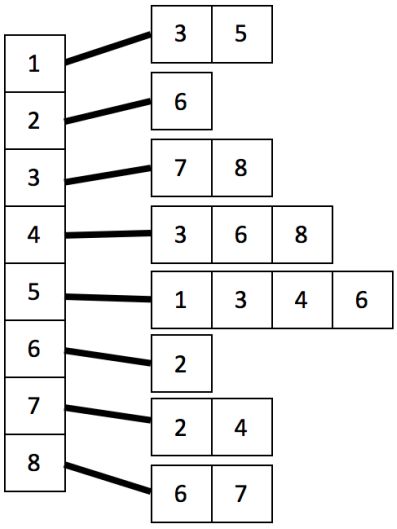
16:332:573

Midterm Exam III

Duration: 150 minutes, a closed book closed note exam.

Q1. DFS/BFS: Consider the directed graph who adjacency structure is as follows. [20pt]



a) List the vertices in the order they would be first visited by bread-first search (BFS) (starting from vertex 1, visit a node with a smaller id to break tie)

Sol) 1 3 5 7 8 4 6 2

b) List the vertices in the order they would be visited by depth-first search (DFS preorder) (starting from vertex 1, visit a node with a smaller id to break tie)

Sol) 1 3 7 2 6 4 8 5

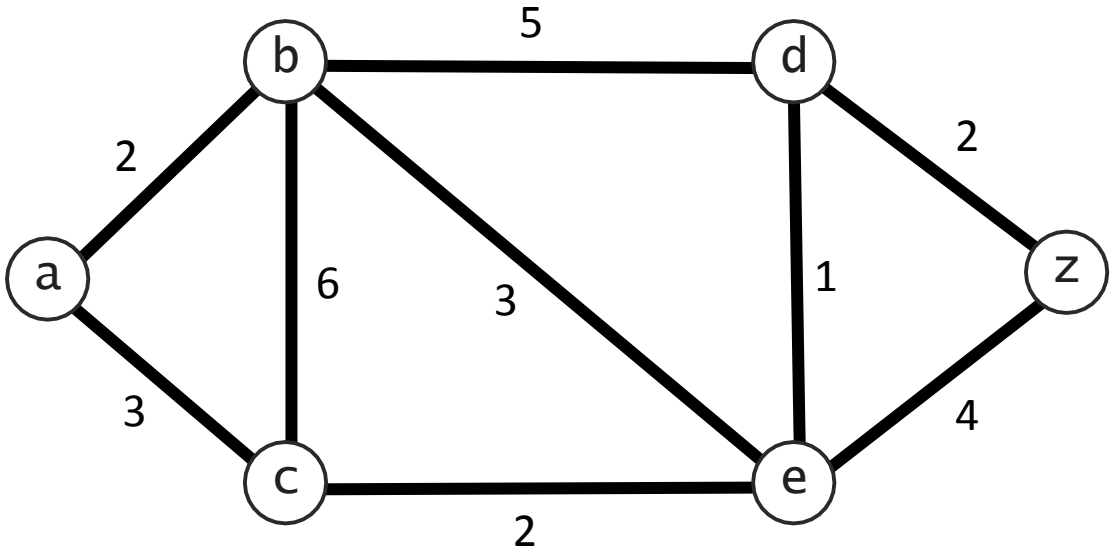
c) Does the graph have a topological order? If so, list the vertices in a topological order. If not, explain why.

Sol) No topological order since there exist cycles in the graph.

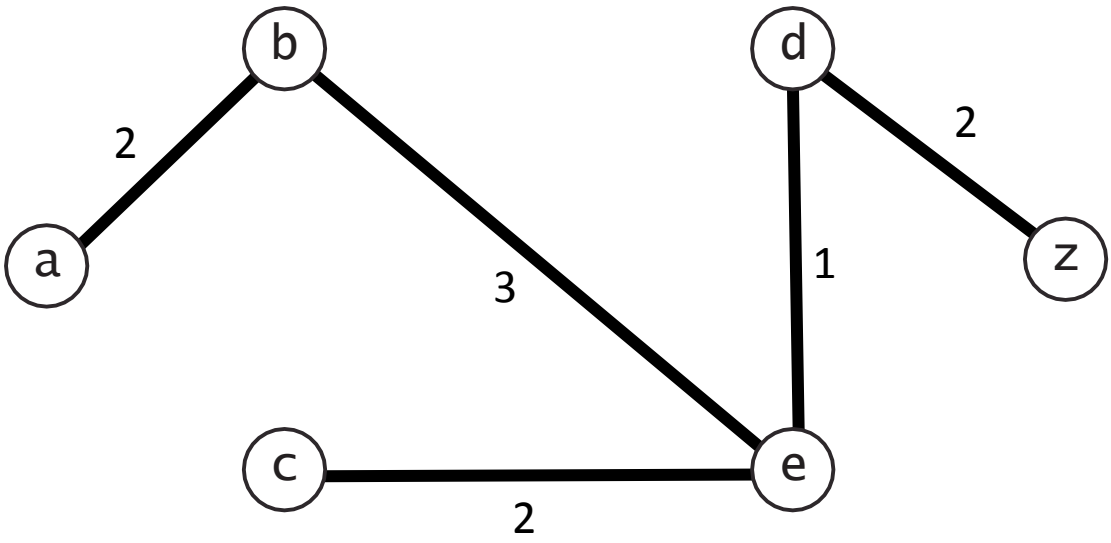
d) Find the vertex sets of the strongly connected components.

Sol) {1, 5} {2, 6} {3, 4, 7, 8}

Q2. Prim's Algorithm: Describe how Prim's algorithm would find a minimum spanning tree in the following weighted graph. Start from node e and use alphabetical order to break ties. [20pt]



Sol) Prim's algorithm will proceed as follows. First we add edge {d, e} of weight 1. Next, we add edge {c, e} of weight 2. Then we add edge {d, z} of weight 2. After that, we add edge {b, e} of weight 3. Finally, we add edge {a, b} of weight 2. This produces a minimum spanning tree of weight 10.



Q3. T/F: Mark either T (True) or F (False) for each statement. [20pt]

a) [F] Let's assume we use DFS for checking cycles when computing MST. Then Kruskal's algorithm computes it in time proportional to E logE in the worst case.

b) [T] The performance of Prim's algorithm depends on PQ implementation used internally.

c) [F] In order to improve the performance of Bellman-Ford algorithm, we could maintain a queue of vertices whose known distance from the source node changed in pass i. Then its *worst case* running time becomes E + V instead of E x V.

d) [F] Dijkstra's algorithm can find a short path tree (SPT) even if there exist negative edges in the graph.

e) [F] If there exist negative cycles in the graph, Bellman-Ford algorithm can find SPT.

f) [F] Mergesort is in-place sorting algorithm with N log N worst case.

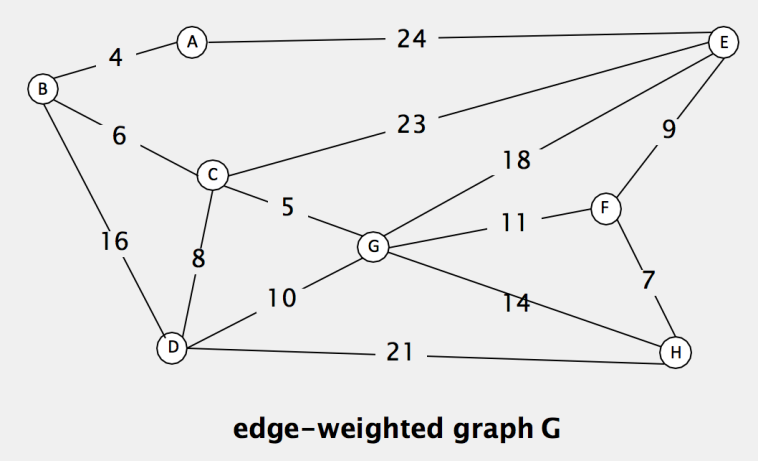
g) [F] Topological sort algorithm can detect negative cycles.

h) [F] Binary search tree is perfectly balanced all the time.

i) [F] BFS, Kruskal, and Dijkstra algorithms all maintain a set of explored vertices S and grow S by exploring edges with exactly one endpoint leaving S.

j) [F] Queue implements Last-In, First-Out strategy and Stack implements First-In, First-Out strategy.

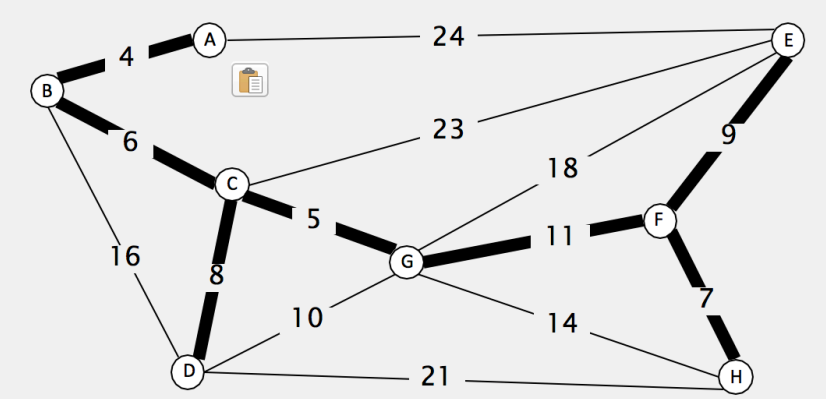
Q4. Kruskal-MST: Consider the following edge-weighted graph G.



edge-weighted graph G

a) Apply Kruskal's algorithm to find a minimum spanning tree. List the edges in the order that the algorithm examined (make sure to include all examined edges even if they are not included in the resulting MST). Calculate the cost, i.e., sum of all edges in the resulting MST. [15pt]

Sol) A-B, C-G, B-C, F-H, C-D, E-F, ~~D-G (cycle)~~, G-F (reaches V-1 nodes in MST, so finish the algorithm) (5pt)



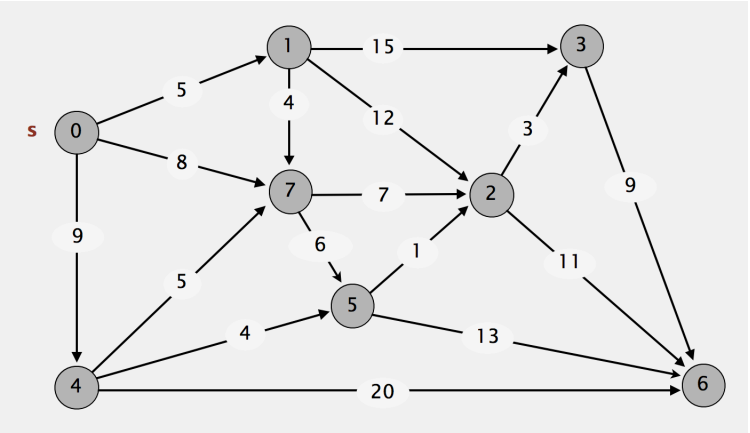
minimum spanning tree T
(cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)

MST: 5pt, Cost: 5pt

b) How would you find a maximum spanning tree of the graph G? [5pt]

Sol) Negative the weight of each edge and run any of the algorithms to find minimum spanning tree (or reverse the sense of comparison in the compareTo() method.).

Q5. Dijkstra's algorithm: Consider the following edge-weighted digraph G. Let's assume that we are running Dijkstra algorithm on the graph and has the following intermediate state table. distTo[] array has the known shortest path length for each node as of now and edgeTo[] array stores the last node connected to the vertex. In the table, vertex 0, 1, 7 (colored in blue) have been already visited by the algorithm and corresponding array elements are updated.



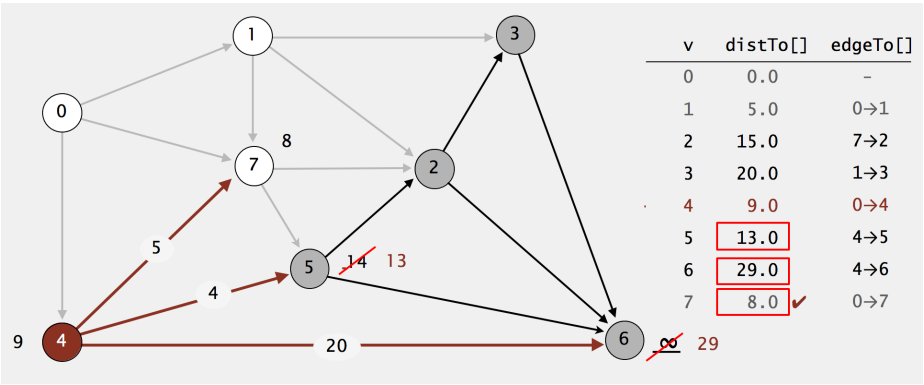
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

a) Which vertex will the algorithm visit next? Why? [10pt]

Sol) Vertex 4 (5pt) since it has the minimum distance from the source node. (distTo[4] = 9.0) (5pt)

b) Let's say the answer to the question a) is vertex X. Relax each edge connected to the vertex X and update the above table (i.e., redraw the table with distTo[] and edgeTo[] entries updated.) [10pt]

Sol)



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0 ✔	0→7

Q6. Advanced Shortest Path: We are given an edge-weighted digraph G = (V, E). Each edge's weight can be represented as r(u, v). Edge weight r(u, v) is a real number in the range $0 \le r(u, v) \le 1$ that represent the reliability of a communication channel from vertex u to vertex v. We interpret r(u, v) as the probability that the channel from u to v will not fail, and we assume that

these probabilities are independent. The end-to-end reliability from vertex u to vertex v can be calculated as the multiplication of all edges involved in between the two vertices. Give an efficient algorithm to find the most reliable path between two given vertices, i.e., path that gives the highest end-to-end reliability. [20pt]

* In this problem, you can assume that the grader of this exam has an enough knowledge about how BFS/DFS/Topological-sort/Kruskal/Prim/Dijkstra/Bellman-Ford algorithms work.

Sol) This problem is similar to the arbitrage detection example that we discussed in the last class.

To find the most reliable path between s and t, run Dijkstra's algorithm with edge weights $w(u, v) = -\lg r(u, v)$ to find shortest paths from s in $O(E+V \lg V)$ time. The most reliable path is the shortest path from s to t, and that path's reliability is the product of the reliabilities of its edges.

Here's why this method works. Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path $s \overset{p}{\rightsquigarrow} t$ such that $\prod_{(u,v)\in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg(\prod_{(u,v)\in p} r(u, v)) = \sum_{(u,v)\in p} \lg r(u, v)$, which is in turn equivalent to minimizing $\sum_{(u,v)\in p} -\lg r(u, v)$. (Note: $r(u, v)$ can be 0, and $\lg 0$ is undefined. So in this algorithm, define $\lg 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\lg r(u, v)$, we have a shortest-path problem.

Since $\lg 1 = 0$, $\lg x < 0$ for $0 < x < 1$, and we have defined $\lg 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \lg V)$ time.

Alternate answer

You can also work with the original probabilities by running a modified version of Dijkstra's algorithm that maximizes the product of reliabilities along a path instead of minimizing the sum of weights along a path.

In Dijkstra's algorithm, use the reliabilities as edge weights and substitute

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue,
- $\times$ for $+$ in relaxation,
- 1 (identity for $\times$) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max).

For example, the following is used instead of the usual RELAX procedure:

RELAX-RELIABILITY(u, v, r)
if $d[v] < d[u] \cdot r(u, v)$
  then $d[v] \leftarrow d[u] \cdot r(u, v)$
    $\pi[v] \leftarrow u$

This algorithm is isomorphic to the one above: It performs the same operations except that it is working with the original probabilities instead of the transformed ones.

Q7. Bellman-Ford: Let's consider the following pseudo code to implement a Bellman-Ford algorithm. Let V be the number of vertices in the graph G and E be the number of edges in the graph.

```
Boolean justdo(Graph G, Vertex source)
{
1:    initialize variables: source, distTo[], edgeTo[], etc.
2:    for i=1 to V-1
3:        do for each edge e: u → v in G
4:            relax(e) and update distTo[] and edgeTo[].
5:    for each edge e: u → v in G
```

```
6:          do if distTo[v] > distTo[u] + weight(e)
7:                  then return FALSE
8:      return TRUE
}
```

a) What does the above algorithm do? If it returns TRUE, what would be the outcome of the algorithm? (10pt)

Sol) This code implements the Bellman-Ford algorithm. (5pt). If it returns TRUE, it will give a shortest path tree (SPT) from a source to every other vertex. (5pt).

b) What is its worst case complexity? [5pt]

Sol) O(EV).

c) If the algorithm returns FALSE, what does that mean? [5pt]

Sol) There exists a negative cycle. (5pt)

Q8. Recursion: Recall the Tower of Hanoi problem discussed in the class. An implementation of solveTowers() was as follows. [10pt]

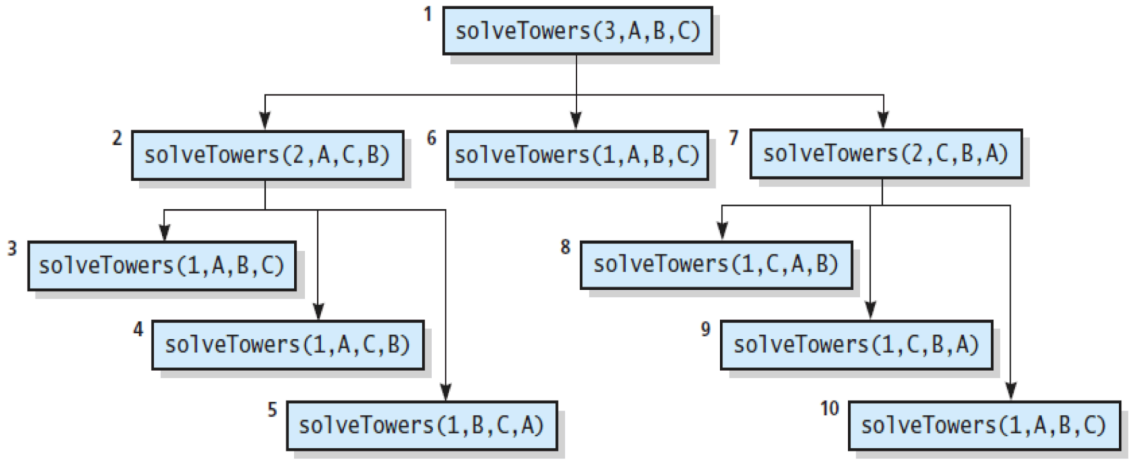
```
void solveTowers(int count, char source, char destination, char spare)
{
    if (count == 1)
    {
        cout << "Move top disk from pole " << source
```

```
                << " to pole " << destination << endl;
        }
        else
        {
                solveTowers(count – 1, source, spare, destination);
                solveTowers(1, source, destination, spare);
                solveTowers(count – 1, spare, destination, source);
        }
}
```

Let's consider the case in which the number of poles is 3, i.e., solveTowers(3, 'A', 'B', 'C') is called in the beginning. Whenever the recursive function solveTowers() returns, it writes a log to the screen (see cout statement above) to instruct the necessary action, e.g., "move top disk from pole X to pole Y". Trace the function call solveTowers(3, 'A', 'B', 'C') and identify the 6th instruction.

Sol)

Trace:



(5pt)

6th step is "Move top disk from pole C to pole B." (5pt)

Q9. Binary Search: Provide a function that implements binary search. Your function will take an integer array and an integer key value as input. Return a location of the key if there is a match. If no match, return -1. [10pt]

Sol) Give full credit if the operation is correct.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
   }
   return -1;
}
```