EC 504
Spring, 2022
HW 4

1. (15 pts) Consider an undirected graph $G = (V, E)$.

   (a) Describe a linear-time algorithm $O(|V| + |E|)$ for determining whether the graph is bipartite.

   Solution: Do a Breadth-First-Search algorithm, with an extra twist. Color the first vertex you insert into the queue as red. Subsequently, when you insert a vertex into the queue, color the vertex as black if its parent is a red vertex, and red if its parent is a black vertex. At each step in the BFS algorithm, when you scan the neighbors of a vertex (including those that have left the queue), if you have a neighbor that is of the same color as the vertex, then the graph is not bipartite. If you reach the end where all the neighbors have left the queue and no edges between different-colored nodes are found, the graph is bipartite.

   The complexity of the algorithm is $O(|E| + |V|)$. Each edge is examined twice (undirected graph), and a queue of length $|V|$ is used.

   (b) Describe an $O(|V|)$-time algorithm for determining whether the graph contains a cycle. .

   Solution: Do a Depth-First-Search algorithm on the graph. If, while doing DFS, you find an edge that points to a node already in the stack for DFS, this is a back edge and therefore has the graph has a cycle. If you don't find one, then there are no cycles.

   This is $O(|V|)$ because, if the graph has no cycles, it only has at most $|V| - 1$ edges, so DFS is $O(|V|)$. If there is a cycle, a back edge will be found no later than when DFS reaches the end of a connected component, which is $O(|V|)$.

   (c) Describe an algorithm to find the minimum number of edges that must be removed from a given undirected graph in order to make it acyclic. Note the graph may not be connected.

   Solution: If the graph is connected, this is a simple algorithm: remove $|E| - |V| + 1$ edges. Note the question does not ask to find which edges to remove. The algorithm is to do BFS or DFS to find the number of connected components k. Then, remove $|E| - |V| + k$ edges. Complexity of this is $O(|V|)$.

2. (15 pts) The two arrays below contain a description of a directed, graph, stored as a forward star. The vertices V are numbered $1, \ldots, 7$, and the edges are numbered $1, \ldots, 10$.

```
Array Firstout:  {1,3,5,7,8,10,11}
Array Endvertex:   {2,3,6,4,4,5,6,4,7,7}
```

   (a) Draw the graph.

   Solution: Reading the forward star, the arcs out of vertex 1 have end vertices in positions 1 and 2 of the edge array, so edges are (1,2),(1,3); those out of vertex 2 have end vertices in positions 3 and 4 of the edge array, which are edges (2,6), (2,4); those out of vertex 3 have end vertices in positions 4 and 6, so arcs are (3,4),(3,5); Out of vertex 4 in position 7, so arc is (4,6); out of vertex 5 in positions 8 and 9, so arcs are (5,4), (5,7); out of vertex 6 in position 10, so arc is (6,7); out of vertex 7, there are no arcs, since position 11 is after the end of the array.

   (b) Is the graph acyclic?

   Solution: Yes. Just draw it and check.

   (c) Use depth first search, implemented by recursion, examining the arcs in the order in which they appear in the array Endvertex whenever there is a tie, to generate the order in which the vertices will be visited. Draw also the spanning tree constructed by the relation that the parent vertex spawns the recursive function which examines the child vertex.

   Solution: Depth first search, starting from vertex 1, leads to the following order of the vertices: Fill up the stack in order from bottom up: First vertex 1, then look at arc (1,2) and add vertex 2 on top; look

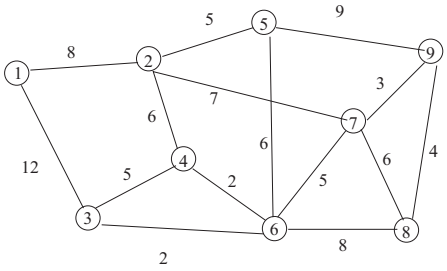
at arc (2,6) (the first arc out of vertex 2 in the above order) and add vertex 6; arc (6,7) adds vertex 7. Since there are no more arcs, pop vertex 7 out first, then vertex 6.

Now, look at the next arc out of vertex 2, which is (2,4) and add vertex 4 to stack. The arc out of vertex 4 is (4,6), but vertex 6 is already scanned, so don't put it back. Instead, pop vertex 4, then vertex 2.

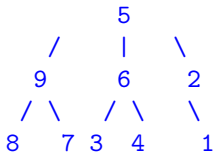
Look at next arc out of vertex 1, (1,3) and add vertex 3 to stack. Look at arcs out of vertex 3 to stack, (3,4) but don't add 4 because it was already popped, and (3,5), so add vertex 5 to stack. Arcs out of vertex 5 lead to vertices which have already been popped, so pop vertices 5, then 3, then 1 out of the stack.

The pop order is: 7 - 6 - 4 - 2 - 5 - 3 - 1.

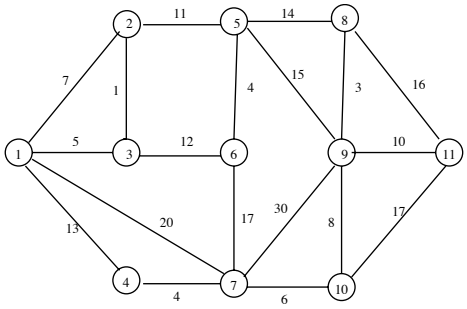
3. (10 pts) Consider the undirected, weighted graph on the right. List the order in which the nodes will be visited in a Breadth-First Search, starting from node 5. Assume that the arcs out of a node will be visited in clockwise order, starting from straight up direction. Show the tree which is generated by the relation that node i's parent is the node which puts node i into the stack.



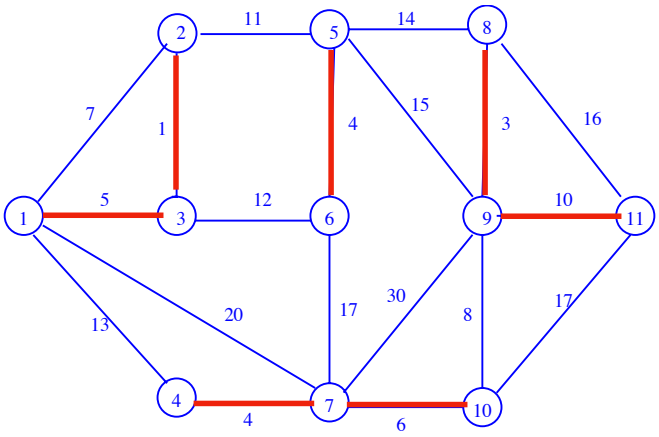
Solution: Starting from node 5, we have, into the queue: Add node 5 to queue and mark it as visited. Remove 5, and add its neighbor nodes 9, 6, 2 to queue and mark as visited. Remove 9, and add 8, 7, mark as visited. Remove 6, and add 3, 4, and mark as visited. Remove 2, and add unvisited neighbor 1 to the queue. Remove nodes in order from queue, to get order 5 - 9 - 6 -2 - 8 - 7 - 3 - 4 - 1 The tree is



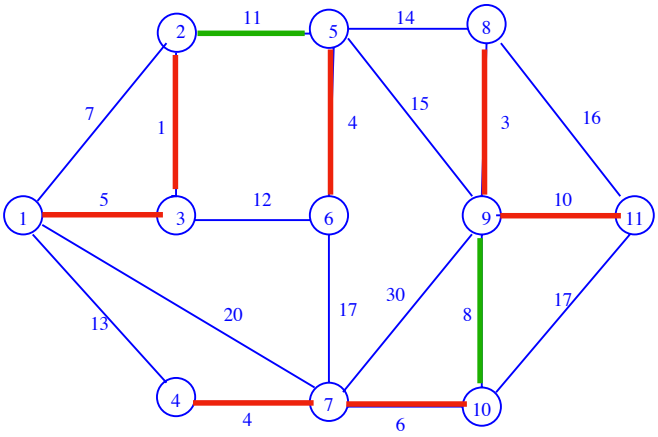
4. (10 pts) For the graph on the right, compute the minimum spanning tree. Identify the algorithm you are using, and show some of the partial work. Compute the total weight of the minimum spanning tree.



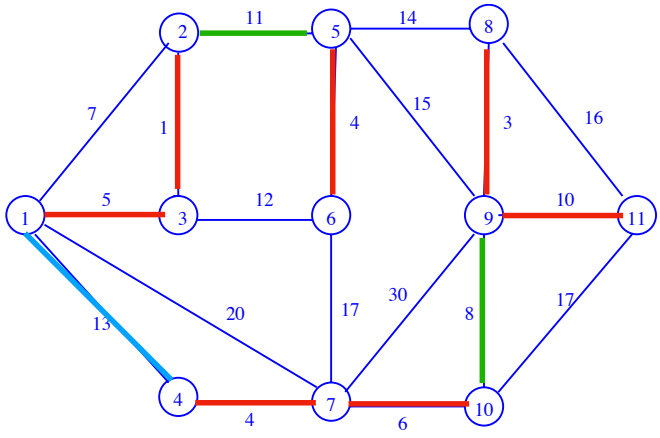
Solution: We will show Boruvska's algorithm, as it takes the least number of pictures. At step 1, we find the least-weight edges edges connected to each vertex, and add them to the tree. this is shown below:



This step reduces it to 4 connected components. At step 2, we find the smallest weight edge that leaves each connected component, and add it to the tree. These are the green edges below:



Now we need one more edge. We find the smallest weight edge that connects the two remaining components, which is the edge $\{1, 4\}$. The final MST is shown below.



with a total weight of 65.

5. (10 pts) Consider a path between two vertices s and t in an undirected, *connected* weighted graph G where no two edges have the same weight. The bottleneck length of this path is the maximum weight of any edge in the path. The bottleneck distance between s and t is the minimum bottleneck length of any path from s to t. Describe an algorithm to compute the bottleneck distance between every pair of vertices in an arbitrary undirected weighted graph.

Solution: Find a MST (V, T). The bottleneck distance between any two vertices will be the maximum edge weight in the unique path between two vertices in the MST.

Why is this true? Assume it is not: Let d^* be the bottleneck distance between s and t in (V, E), and let $d > d^*$ be the bottleneck distance between s and t in the MST (V, T). Then, there is an edge e in the path p from s to t in (V, T) such that $w(e) = d > d^*$. There is also a path p_1 from s to t in (V, E) where the weight of all edges is less than or equal to d^*. Thus, if one were to execute Kruskal's algorithm, vertices s, t would be in the same connected component once all edges of weight less than or equal to d^* were considered. Thus, the edge e with weight $d > d^*$ cannot be part of an MST.

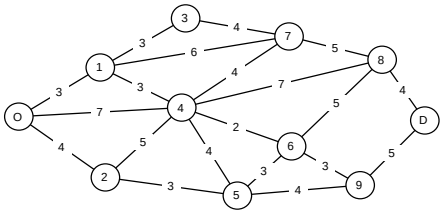
The complexity for finding an MST is $O(|E| + |V| \log(|V|))$. Now, how can we find the bottleneck distance for every pair of vertices from the MST? If you think about this, you are generating $O(|V|^2)$ outputs, so any solution is at least $\Omega(|V|^2)$. There are many ways of doing this. One easy way is to pick the largest weight edge in the MST, and find the two connected components in the rest of the MST. Then, the bottleneck distance between every pair of vertices, one in one component, one in the other is the largest weight. We now have two smaller equivalent problems, with one less edge. The worst case is when only one vertex is in one connected component, so we reduce the size of the problem by 1. We have to do this iteration $|V|$ times, and the complexity of each iteration is $O(|V|)$, so we have an $O(|V|^2)$ algorithm, which was the best we could hope for.

6. (10 pts) Suppose you have computed a minimum spanning tree T for a graph $G(V, E)$. Assume you now add a new vertex n and undirected arcs $E_n = \{\{n, v_i\} \text{ for some } v_i \in V\}$, with new weights $w(n, v_i)$. Provide the pseudocode for an algorithm to find the minimum weight spanning tree in the augmented graph $G_a(V \cup \{n\}, E \cup E_n)$. Estimate the complexity of this algorithm.

Solution: Let $G(V, T)$ be the MST in the original graph. A new minimum spanning must be formed from edges of the old MST plus the new edges to vertex n.

Form the graph $G_b(V \cup \{n\}, E \cup E_n)$. Find an MST in this graph, which has $2|V| - 1 edges$, and $|V| + 1$ vertices, using either Kruskal's or Prim's algorithm with a simple binary heap, to get an algorithm of $O(|V| \log(|V|))$ complexity.

7. (10 pts) Consider the undirected weighted graph in the right. Show each step of Boruvka's algorithm for finding a minimum spanning tree in the graph.



Solution: Boruvka generates an MST in one iteration, of weight 33. The MST is shown below.