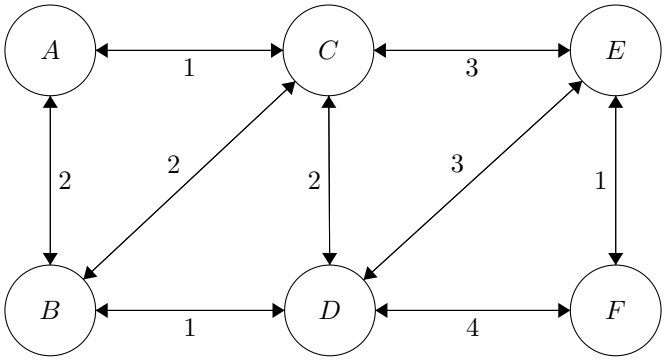
CS 61B Fall 2020

Graphs, MSTs

Discussion 11: November 2, 2020

1 Minimum Spanning Trees



(a) Perform Prim's algorithm to find the minimum spanning tree. Pick A as the initial node. Whenever there is more than one option with the same cost, process them in alphabetical order.

Prim's operates by engulfing nodes. Begin by engulfing A. Consider all the edges outgoing from the engulfed nodes: out of AC and AB, AC is the cheapest. So we engulf C (using edge AC).

Now, we take our engulfed set (AC), and look at outgoing edges. Candidates: AB, BC, CD, CE. We take the cheapest, AB, and engulf B.

Out of (ABC), candidates: BD, CD, CE. We take the cheapest, BD, and engulf D.

Out of (ABCD), candidates: CE, DE, DF. We take the cheapest and lexicographically smallest, CE, and engulf E.

Finally, we take EF.

Edges picked: AC, AB, BD, CE, EF.

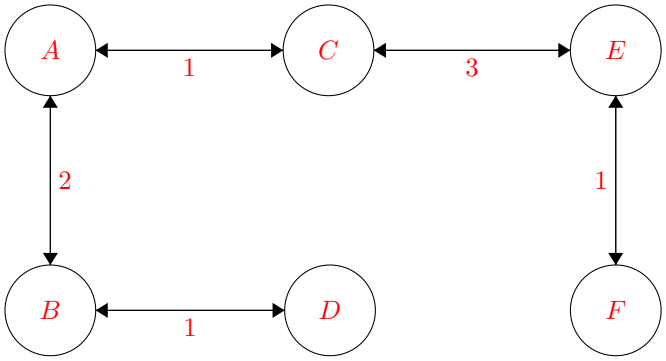
(b) Use Kruskal's algorithm to find a minimum spanning tree. When deciding between equiweighted edges, alphabetically sort the edges, and then pick in lexicographic order.

For instance, edges are always written as AB or AC, never BA or CA. If deciding between AB and AC, pick AB first.

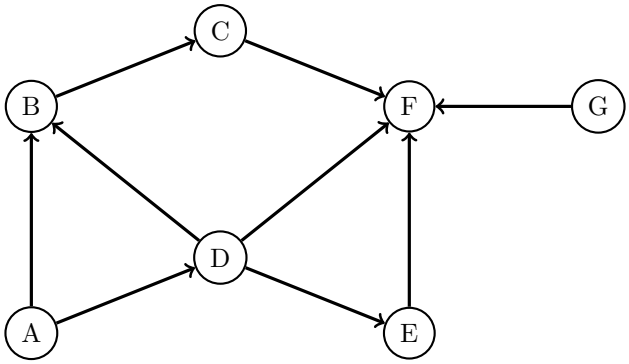
Kruskal's first considers the weight 1 edges. Out of AC, BD and EF, we first pick AC, then BD, then EF since picking all 3 does not create cycles. Next, we consider weight 2 edges. We start with AB, and pick it since it doesn't create any cycles. Next, we consider BC which creates a cycle (ABCA), so we skip it. Next, we consider CD, which creates a cycle (ABDCA), so we skip it.

Next, we consider weight 3 edges, starting with CE. It does not create a cycle, so we pick it. At this point, we stop because we have a spanning tree.

In this case, Prim and Kruskal's output the same MST. This is not always the case.



2 Topological Sorting

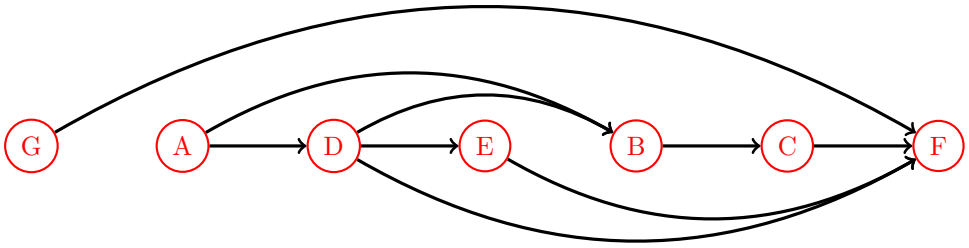


(a) Give a valid topological sort of the graph above. For your reference, some orderings of the graph are provided below.

```
DFS preorder: ABCFDE (G)
DFS postorder: FCBEDA (G)
BFS: ABDCEF (G)
```

A valid topological sorting can be obtained by reversing the DFS postorder.

One valid topological sort is $G-A-D-E-B-C-F$. There are many others. In particular, G can go anywhere except after F, since it has no incoming edges and only one outgoing edge (to F).



(b) There are two requirements that a graph must satisfy in order for there to be a valid topological sorting of the graph. What are they?

1. The graph must be directed. Topological sorting does not make sense for an undirected graph.
2. The graph must not have cycles. If a cycle was to exist, say, A, B, C, A, which node should come first in the topological sort?

Graphs that satisfy both properties are called **Directed Acyclic Graphs (DAGs)**.

(c) *Extra:* Why does using a reverse post-order DFS to compute the topological sort work?

We will show that it works by considering every possible situation in which `dfs` could be called. Consider any edge u, v, and what happens when `dfs`(u)

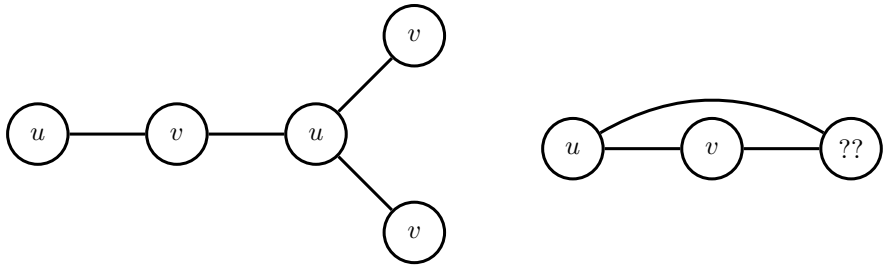
is called. Below, we enumerate the possible cases that could happen on such a `dfs` call.

1. case: `dfs`(v) was called in the past, and it already returned. In this event, v takes place before u.
2. case: `dfs`(v) was called in the past, but hasn't returned. Observe that this means there is a path from v to u (because a series of edges let from v to u through the `dfs` calls on the neighbors), and now there is an edge from u to v. This is impossible in a DAG, therefore, this case is impossible.
3. case: `dfs`(v) is yet to be called. Eventually `dfs`(v) will definitively get called before `dfs`(u) returns (because there is an edge from u to v). And `dfs`(v) must return before `dfs`(u) does because this is a DAG. (See case 2 for justification). Therefore v takes place before u.

Convince yourself that no other cases exist and therefore our proof is complete. Furthermore, observe that we make the arguments above for every single `dfs` call that the algorithm will ever call, and therefore, our entire argument must be correct.

3 Graph Algorithm Design

(a) An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets U and V such that every edge connects an item in U to an item in V. For example below, the graph on the left is bipartite, whereas on the graph on the right is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?



To solve this problem, we run a special version of a traversal from any vertex. This can be implemented with both DFS and BFS. This special version marks the start vertex with a u, then each of its neighbors with a v, and each of their neighbors with a u, and so forth. If at any point in the traversal we want to mark a node with u but it is already marked with a v (or vice versa), then the graph is not bipartite.

If the graph is not connected, we repeat this process for each connected component.

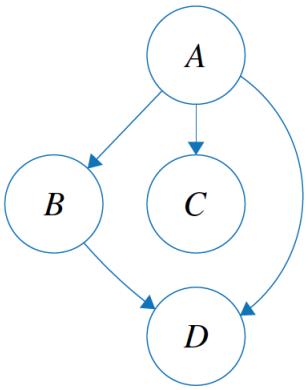
If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

The runtime of the algorithm is the same for BFS and DFS: $\Theta(E + V)$.

(b) Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

Give an example of a graph where this algorithm may not traverse in DFS order.



For the graph above, it's possible to visit in the order $A-B-C-D$ (which is not depth-first) because D won't be put into the fringe after visiting B, since it's already been marked after visiting A. One should only mark nodes when they have actually been visited; in this example, we mark them before we visit them, as we put them into the fringe.

(c) *Extra:* Provide an algorithm that finds the shortest cycle (in terms of the number of edges used) in a directed graph in $O(EV)$ time and $O(E)$ space, assuming $E > V$.

The key realization here is that the shortest directed cycle involving a particular source vertex s is just the shortest path to a vertex v that has an edge to s, along with that edge. Using this knowledge, we create a shortestCycleFromSource(s) subroutine. This subroutine runs BFS on s to find the shortest path to every vertex in the graph. Afterwards, it iterates through all the vertices to find the shortest cycle involving s: if a vertex v has an edge back to s, the length of the cycle involving s and v is one plus distTo(v) (which was computed by BFS).

This subroutine takes $O(E+V)$ time because it uses BFS and a linear pass through the vertices. To find the shortest cycle in an entire graph, we simply call the subroutine on each vertex, resulting in an $V \cdot O(E+V) = O(EV+V^2)$ runtime. Since $E > V$, this is still $O(EV)$, since $O(EV+V^2) \in O(EV+EV) \in O(EV)$.