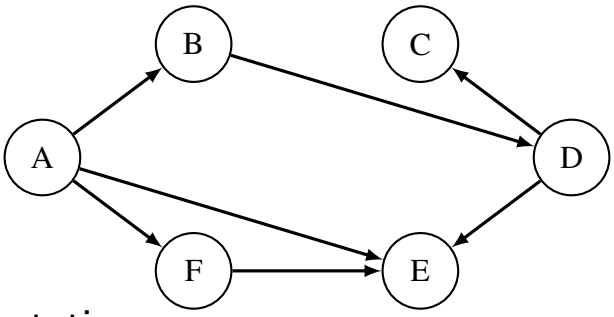


1 Graph Representation

Represent the graph above with an adjacency list and an adjacency matrix representation.

Depending on the convention being used, nodes may or may not have edges to themselves. For this problem, we stick with the convention that nodes do **not** have an edge to themselves.

Adjacency List

A → [B, E, F]
B → [D]
C → []
D → [C, E]
E → []
F → [E]

FROM / TO

	A	B	C	D	E	F
A	0	1	0	0	1	1
B	0	0	0	1	0	0
C	0	0	0	0	0	0
D	0	0	1	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	1	0

(in the above matrix 0 means false and 1 means true)

Note: Edge lists and adjacency lists are **not** the same! An edge list is a list stored in each node that contains all successors and possibly predecessors of that node (see lecture). An adjacency list is more of a table (or map data structure) that lists the adjacent vertices for each vertex in the graph; it is a representation of the graph as a whole. Graphs are commonly represented using adjacency lists and matrices but be aware of what is meant by an edge list.

2 Searches and Traversals

Run depth first search (DFS) preorder, DFS postorder, and breadth first search (BFS) on the graph above, starting from node A. List the order in which each node is first visited. Whenever there is a choice of which node to visit next, visit nodes in alphabetical order.

DFS has both a preorder and postorder traversal. Preorder means we visit the node *before* visiting its children. Postorder order means we visit the node only *after* visiting its children.

DFS preorder: A, B, D, C, E, F
DFS postorder: C, E, D, B, F, A
BFS: A, B, E, F, D, C

3 Topological Sorting

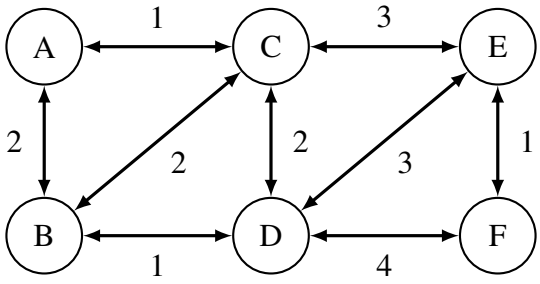
Give a valid topological ordering of the graph. Is the topological ordering of the graph unique?

A topological ordering is a linear ordering of nodes such that for every directed edge S $\rightarrow$ T, S is listed before T. For this problem, the topological ordering of the graph is **not** unique. Below, we list two valid topological orderings for the graph.

- One valid ordering: A, B, D, C, F, E
  - Explanation: One way to approach this problem is to take any node with no edges leading to it and return it as the next node. After returning a node, we delete it and any edges leaving from it and look for a node with no incoming edges in the updated graph. We can repeat this until we have no nodes left. If at any point in this process we have a multiple choices for which node to return then the topological ordering is not unique.
- Another possible valid ordering: A, F, B, D, E, C
  - Explanation: Note that this ordering is just the reverse of DFS postorder traversal. Reverse DFS postorder will always be a valid topological ordering. This is because a DFS postorder traversal visits nodes only after all successors have been visited, so the reverse traversal visits nodes only after all predecessors have been visited.

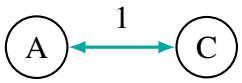
Note: Only Directed Acyclic Graphs (DAGs), which are directed graphs that do not contain any cycles, have topological orderings. This is because within any given cycle, no one node comes before another. There are no valid topological orderings for undirected graphs because there is no direction associated with any edge. No one node comes before another, so it does not make sense to have a topological ordering for undirected graphs.

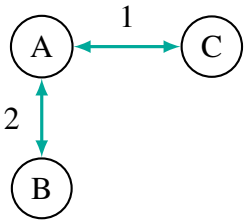
4 Minimum Spanning Trees

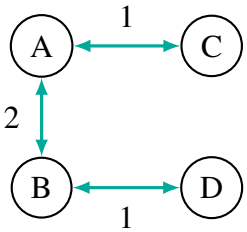


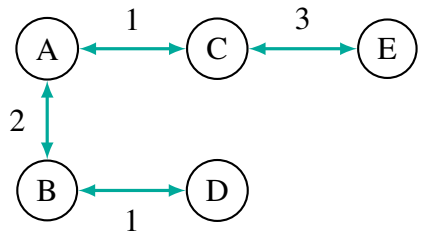
(a) Perform Prim's algorithm to find a minimum spanning tree of the graph above. Pick A as the initial node. If there are multiple nodes with the same cost, process them in alphabetical order.

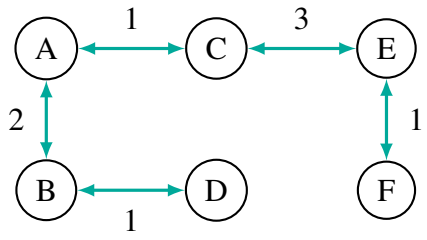
Prim's algorithm adds the shortest edge connecting some node already in the tree to one that isn't yet. This continues until all nodes are in the tree (note this happens when there are n - 1 edges, where n is the number of nodes). Initially, A is the only node in our tree.





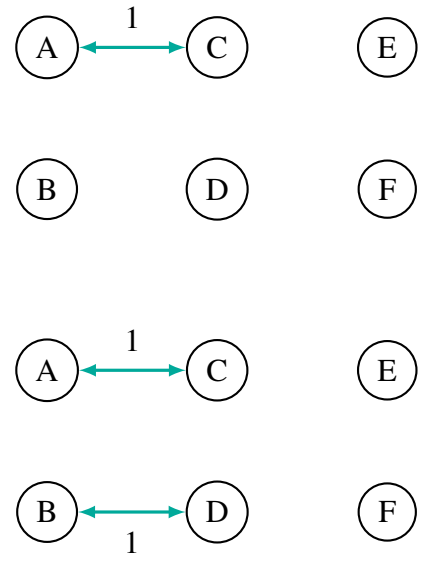


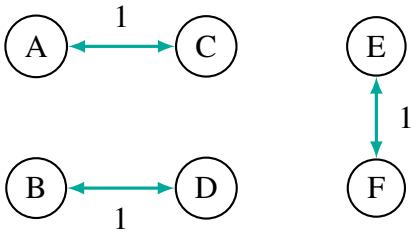


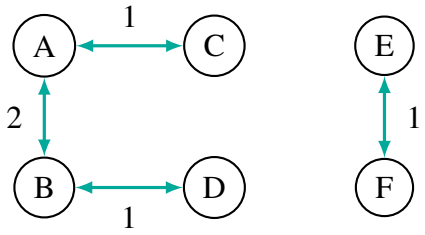


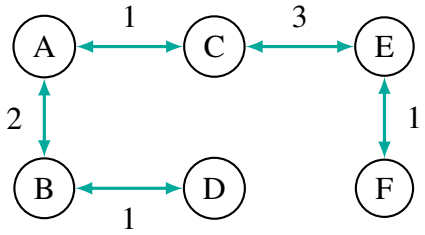
(b) Use Kruskal's algorithm to find a valid minimum spanning tree of the graph above. If there are multiple edges with the same cost, process them in alphabetical order. Will Prim's and Kruskal's always return the same MST?

Kruskal's algorithm goes through every edge in increasing order of weight. If the edge selected connects two subtrees that haven't yet been connected, it adds that edge to the MST. This continues until all nodes are connected into one tree.

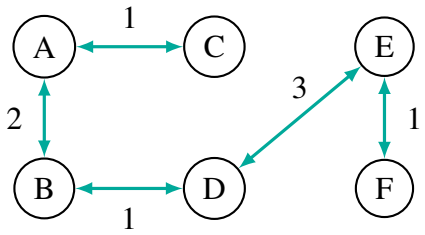






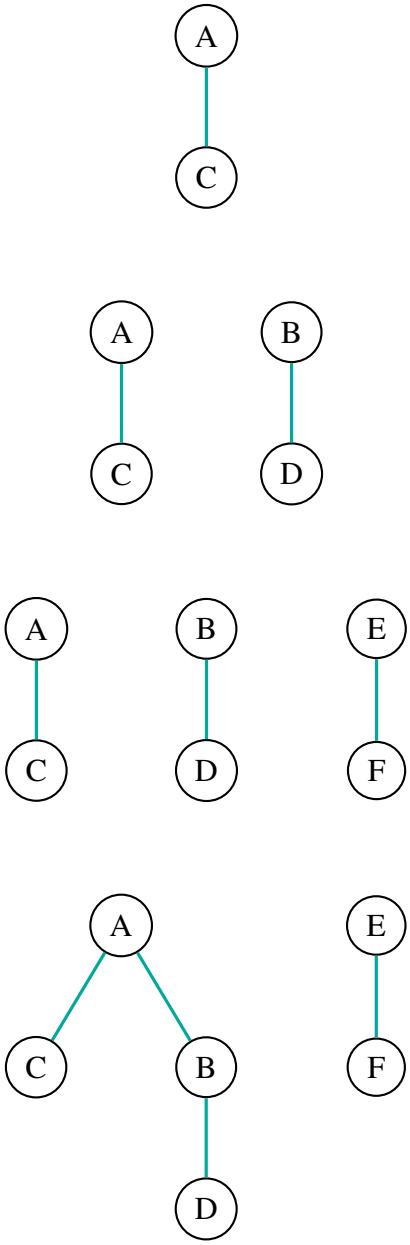


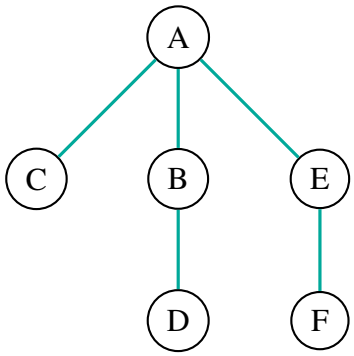
Prim's and Kruskal's happen to output the same MST here, but it is also possible for Kruskal's algorithm to return a different MST based on how its sorting breaks ties. For example, instead of the C-E edge that was chosen last, we could have chosen the D-E edge. This would create the following MST:



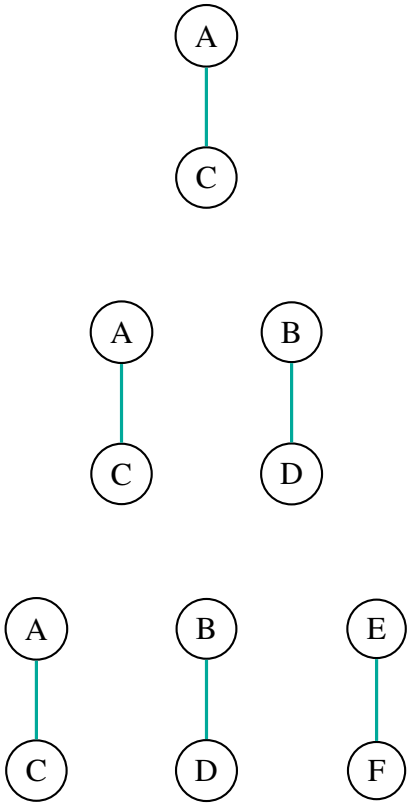
(c) Draw the final state of the tree that results from union find operations executed when running Kruskal's on the graph above. When selecting a new root, break ties alphabetically.

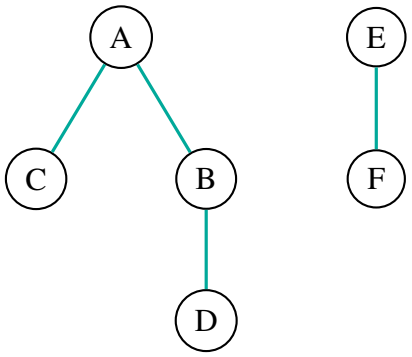
Without path compression: To run Kruskal's we need to execute union find operations. This is so we can **find** which subtree a node currently belongs to as well as **union** two subtrees when we connect them by an edge.





With path compression: Just like before, to run Kruskal's we need to execute union find operations. This is so we can **find** which subtree a node currently belongs to as well as **union** two subtrees when we connect them by an edge. However, with path compression we also want to compress paths whenever possible. This means that whenever we run a **find** operation we compress the path to the root.





When we call find(D) to check whether we want to add edge (C, D), we compress D's path to the root: