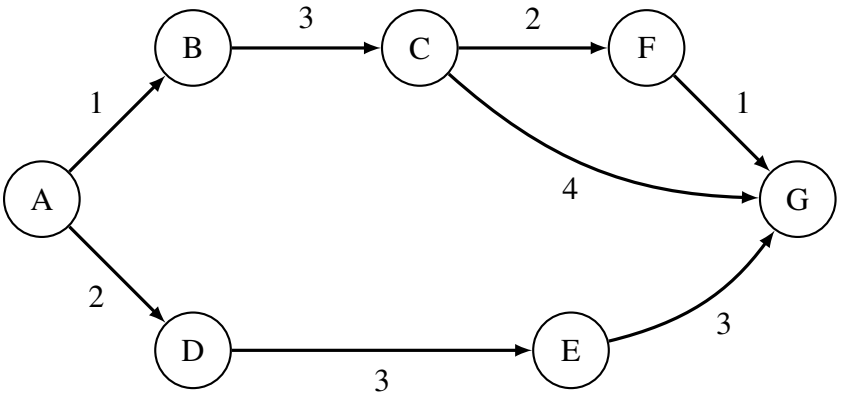
1 A* Search

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`. Remember the heuristic serves to estimate the distance between two nodes `u` and `v`.




```
Edge weights:    Heuristics:
g(A, B) = 1      h(A, G) = 8
g(B, C) = 3      h(B, G) = 6
g(C, F) = 2      h(C, G) = 5
g(C, G) = 4      h(F, G) = 1
g(F, G) = 1      h(D, G) = 6
g(A, D) = 2      h(E, G) = 3
g(D, E) = 3
g(E, G) = 3
```

a) Given the weights and heuristic values for the graph below, what path would A* search return, starting from A and with G as a goal?

A* would return A-D-E-G.

A* is different from Dijkstra's because it finds the shortest distance from a start node to a specific goal node g (rather than to all nodes).

Instead of choosing the node with the smallest `dist(v)` value to pop off the fringe, in A* we choose the node with the smallest `dist(v) + h(v, g)` sum. Remember, `dist(v)` represents the distance from the start node to node v.

The search is finished when we pop the goal node off of the fringe.

In the chart below, we keep track of dist values at each iteration. Note that we stop as soon as we pop G.

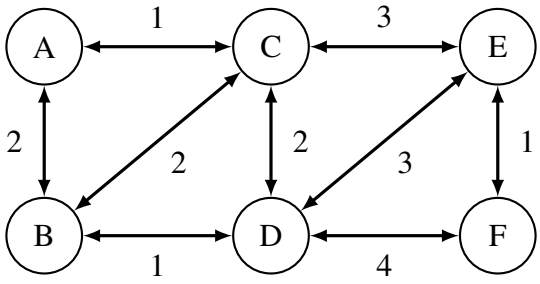
	dist(*v*)					
v	Init	Pop A	Pop B	Pop D	Pop E	Pop G
A	**0**	0	0	0	0	
B	∞	**1**	1	1	1	
C	∞	∞	4	4	4	
D	∞	2	**2**	2	2	
E	∞	∞	∞	**5**	5	
F	∞	∞	∞	∞	∞	
G	∞	∞	∞	∞	**8**	

In order to now derive the path from this table, we start with G and find the node that we traversed right before G. This will be the node that was popped when G obtained its final dist value (the value it has when it's popped). We notice this node is E. We then find the node we traversed right before E and so on. This gets us A-D-E-G.

b) Is the heuristic admissible? Why or why not? The heuristic is not admissible because `h(C, G) = 5`, but the shortest path from C to G has length 3.

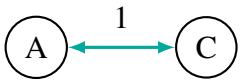
This is why A* returns A-D-E-G (total path distance of 8) when A-B-C-F-G (total path distance of 7) is actually the shortest path. Without an admissible heuristic, A* cannot guarantee it'll return the shortest path.

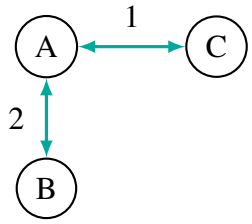
2 Minimum Spanning Trees

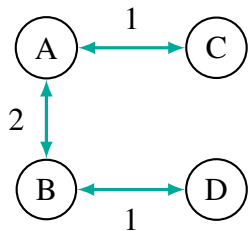


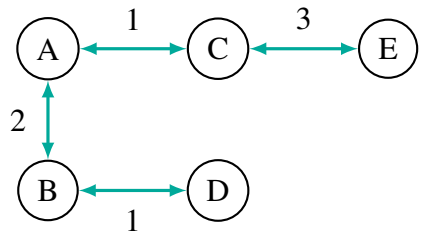
a) Perform Prim's algorithm to find the minimum spanning tree of the above graph. Pick A as the initial node. Whenever there are more than one node with the same cost, process them in alphabetical order.

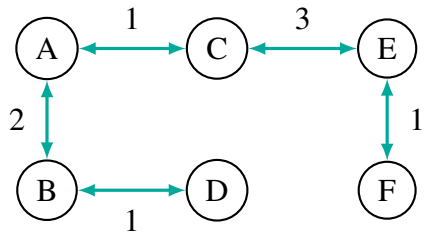
Prim's algorithm adds the shortest edge connecting some node already in the tree to one that isn't yet. This continues until all nodes are in the tree (note this happens when there are n - 1 edges, where n is the number of nodes). Initially, A is the only node in our tree.





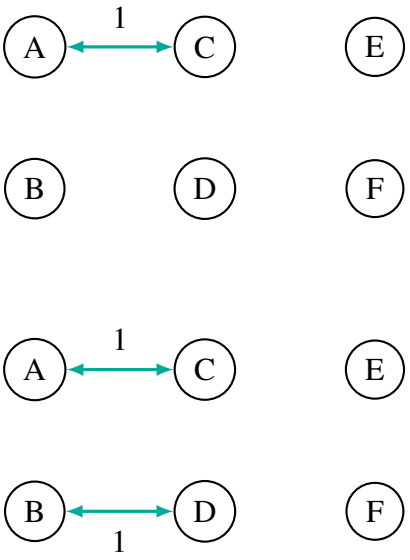


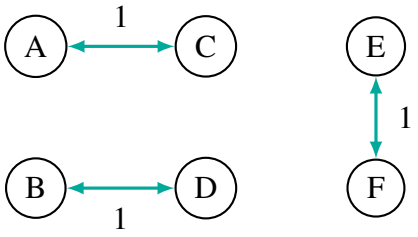


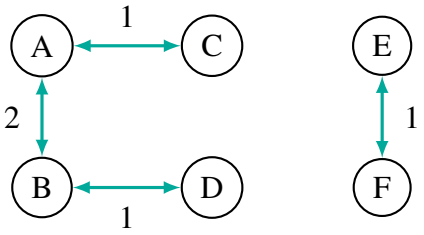


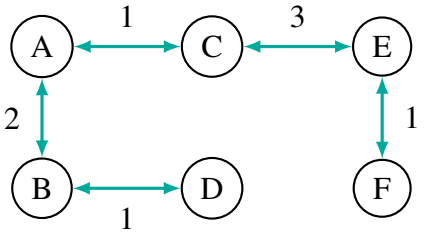
b) Use Kruskal's algorithm to find a minimum spanning tree. Is it the same as the one found by Prim's?

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

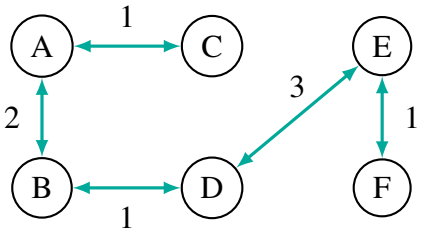








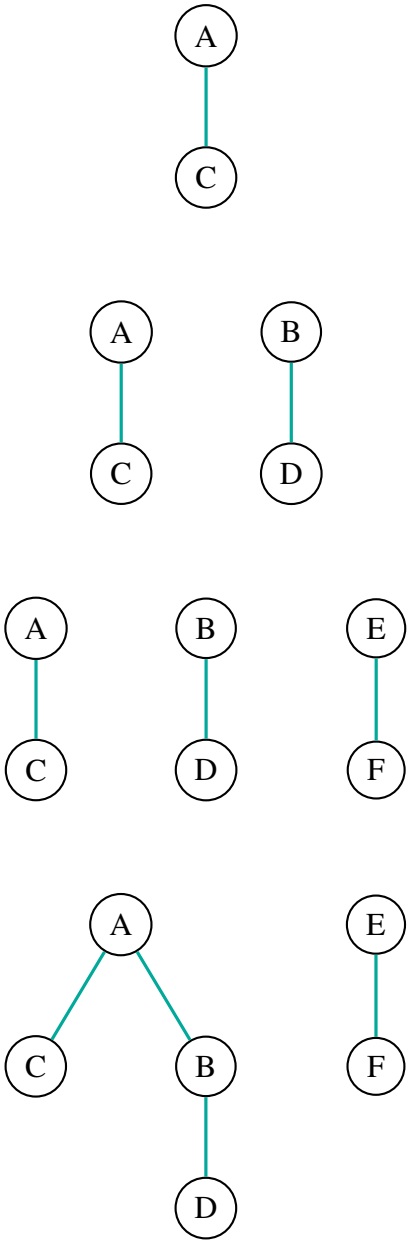
Prim's and Kruskal's happen to output the same MST here, but it is also possible for Kruskal's to return a different MST based on how its sorting breaks ties. For example, instead of the C-E edge that was chosen last, we could have chosen the D-E edge. This would create the following MST:



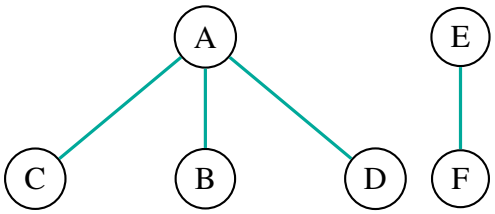
c) Draw the final state of the tree that results from union find operations executed when running Kruskal's on the graph above. When there are ties, choose the root to be the alphabetically first node.

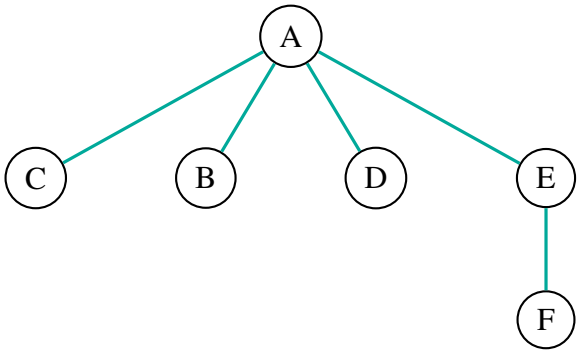
To run Kruskal's we need to execute union find operations. This is so we can **find** which subtree a node currently belongs to as well as **union** two subtrees when we connect them by an edge.

Remember, 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:


A
E
C
B
D
F


A
C
B
D
E
F