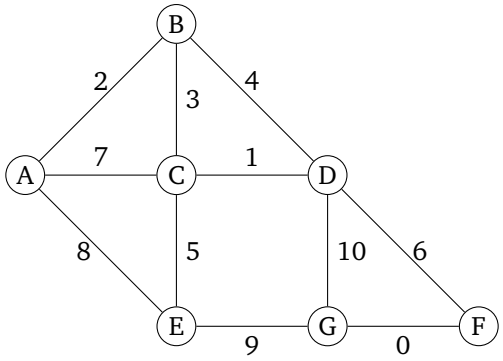
Section 09: Solutions

1. MSTs: Unique Minimum Spanning Trees

Consider the following graph:



(a) What happens if we run Prim's algorithm starting on node A? What are the final costs and edges selected? Give the set of edges in the resulting MST.

Solution:

Step	Components	Edge
1	{A} {B} {C} {D} {E} {F} {G}	(A,B)
2	{A,B} {C} {D} {E} {F} {G}	(B,C)
3	{A,B,C} {D} {E} {F} {G}	(C,D)
4	{A,B,C,D} {E} {F} {G}	(C,E)
5	{A,B,C,D,E} {F} {G}	(D,F)
6	{A,B,C,D,E,F} {G}	(F,G)

(b) What happens if we run Prim's algorithm starting on node E? What are the final cost and edges selected? Give the set of edges in the resulting MST.

Solution:

Step	Components	Edge
1	{A} {B} {C} {D} {E} {F} {G}	(E,C)
2	{C,E} {A} {B} {D} {F} {G}	(C,D)
3	{C,D,E} {A} {B} {F} {G}	(C,B)
4	{B,C,D,E} {A} {F} {G}	(B,A)
5	{A,B,C,D,E} {F} {G}	(D,F)
6	{A,B,C,D,E,F} {G}	(F,G)

(c) What happens if we run Prim's algorithm starting on *any* node? What are the final costs and edges selected? Give the set of edges in the resulting MST.

Solution:

The answer would be the same as the one we get above, since for each node, we always choose the smallest-weight edge that links to it.

(d) What happens if we run Kruskal's algorithm? Give the set of edges in the resulting MST.

Solution:

We'll use this table to keep track of components and edges we processed. The edges are listed in an order sorted by weight.

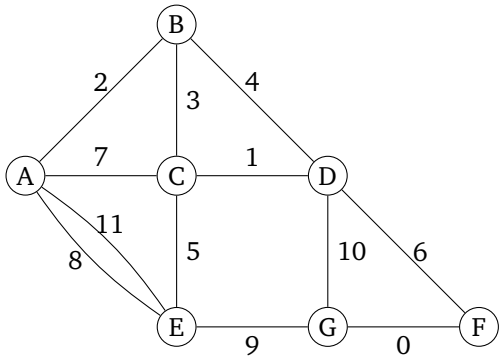
Step	Components	Edge	Include?
1		(F,G)	
2		(C,D)	
3		(A,B)	
4		(B,C)	
5		(B,D)	
6		(C,E)	
7		(D,F)	
8		(A,C)	
9		(A,E)	
10		(E,G)	
11		(D,G)	

After executing Kruskal's algorithm on the above graph, we get

Step	Components	Edge	Include?
1	{A} {B} {C} {D} {E} {F} {G}	(F,G)	Yes
2	{A} {B} {C} {D} {E} {F,G}	(C,D)	Yes
3	{A} {B} {C,D} {E} {F,G}	(A,B)	Yes
4	{A,B} {C,D} {E} {F,G}	(B,C)	Yes
5	{A,B,C,D} {E} {F,G}	(B,D)	No
6	{A,B,C,D} {E} {F,G}	(C,E)	Yes
7	{A,B,C,D,E} {F,G}	(D,F)	Yes
8	{A,B,C,D,E,F,G}	(A,C)	No
9	{A,B,C,D,E,F,G}	(A,E)	No
10	{A,B,C,D,E,F,G}	(E,G)	No
11	{A,B,C,D,E,F,G}	(D,G)	No

The resulting MST is a set of all edges marked as *Include* in the above table.

(e) Suppose we modify the graph above and add a heavier parallel edge between A and E, which would result in the graph shown below. Would your answers for above subparts (a, b, c, and d) be the same for this following graph as well?



Solution:

The steps are exactly the same, since we don't consider the heavier edge when there are parallel edges. The reason is that the heavier edge would never be considered as the best edge when there is a lighter one (of weight 8) that can be added to the graph instead.

2. MSTs: True or False

Answer each of these true/false questions about minimum spanning trees.

(a) A MST contains a cycle.

Solution:

False. Trees (including minimum spanning trees) never contain cycles.

(b) If we remove an edge from a MST, the resulting subgraph is still a MST.

Solution:

False, the set of edges we chose will no longer connect everything to everything else.

(c) If we add an edge to a MST, the resulting subgraph is still a MST.

Solution:

False, an MST on a graph with n vertices always has $n - 1$ edges.

(d) If there are V vertices in a given graph, a MST of that graph contains $|V| - 1$ edges.

Solution:

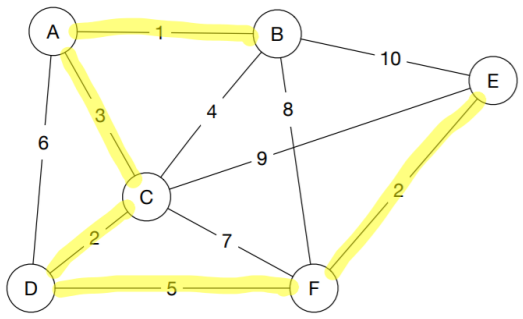
This is true (assuming the initial graph is connected).

3. MSTs: Kruskal's Algorithm

Answer these questions about Kruskal's algorithm.

(a) Execute Kruskal's algorithm on the following graph. Fill the table.

Solution:



Step	Components	Edge	Include?
1	{A} {B} {C} {D} {E} {F}	A, B	Yes
2	{A, B} {C} {D} {E} {F}	D, B, C	Yes
3	{A, B} {C, D} {E} {F}	E, F	Yes
4	{A, B} {C, D} {E, F}	A, C	Yes
5	{A, B, C, D} {E, F}	B, C	No
6	— " —	D, F	Yes
7	{A, B, C, D, E, F}	A, D	No
8	— " —	C, F	No
9	— " —	B, F	No
10	— " —	C, E	No
11	— " —	E, B	NO

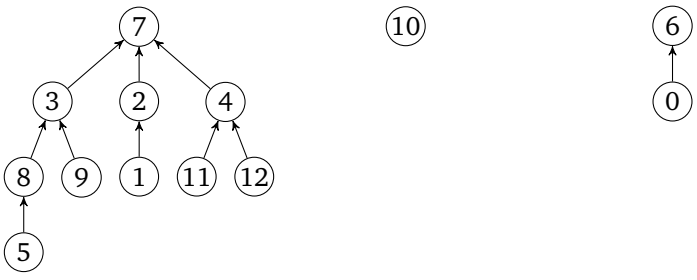
(b) In this graph there are 6 vertices and 11 edges, and the for loop in the code for Kruskal's runs 11 times, a few more times after the MST is found. How would you optimize the pseudocode so the for loop terminates early, as soon as a valid MST is found.

Solution:

Use a counter to keep track of the number of edges added. When the number of edges reaches $|V| - 1$, exit the loop.

4. Disjoint Sets

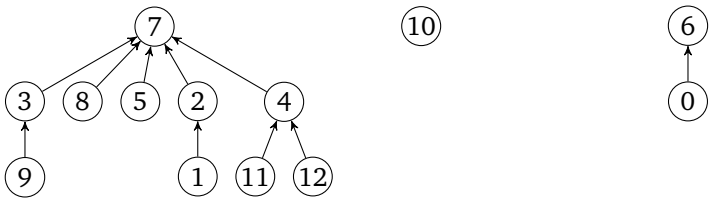
(a) Consider the following trees, which are a part of a disjoint set:



For these problems, use both the **union-by-rank** and **path compression** optimizations. Assume that the first tree has rank 3, the second has rank 0 and the last has rank 1.

(i) Draw the resulting tree(s) after calling `findSet(5)` on the above. What value does the method return?

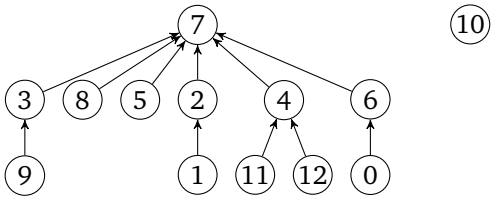
Solution:



The method returns 7, which is the root of the tree.

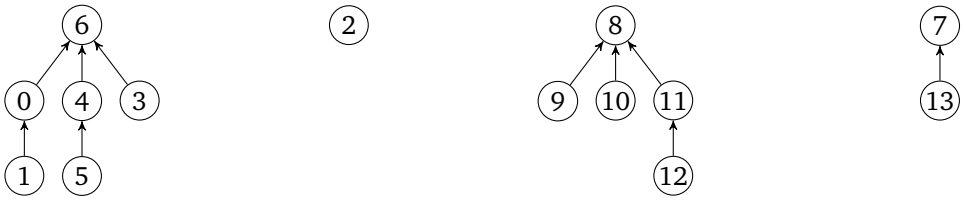
(ii) Draw the final result of calling `union(2,6)` on the result of part (i).

Solution:



We merge the second tree to the first since the rank of the first tree is 3, while that of the second tree is 1. The combined tree has rank 3 – since the inital ranks were unequal, the combined tree gets the max of the previous two. Notice that the rank is not the same as the height.

(b) Consider the disjoint-set shown below:

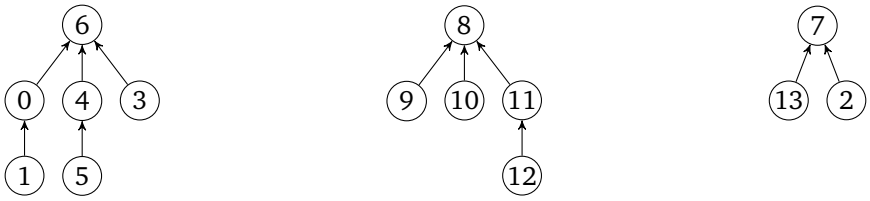


The ranks of trees from left to right are 2, 0, 2, 1.

What would be the result of the following calls on union if we add the "union-by-rank" and "path compression optimizations. Draw the forest at each stage with corresponding ranks for each tree

(i) `union(2, 13)`

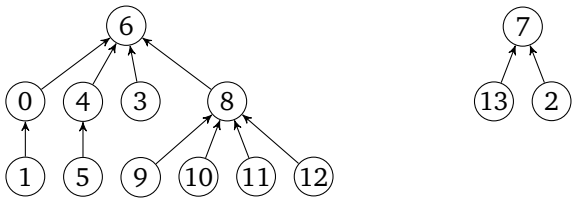
Solution:



The ranks from left to right are 2, 2, 1.

(ii) `union(4, 12)`

Solution:

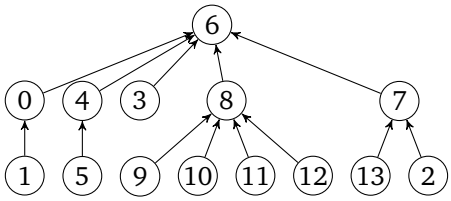


The ranks from left to right are 3, 1.

The ranks were tied, so we arbitrarily chose 6 to be the root; 8 could be chosen as well. Notice that the rank of the new tree is 3 (since the initial ranks were equal, the new rank is one more than the previous) even though the height is 2.

(iii) `union(2, 8)`

Solution:



The final rank of the remaining tree is 3.

5. Graph Modeling 1: DJ Kistra

You've just landed your first big disk jockeying job as "DJ Kistra."

During your show you're playing "Shake It Off," and decide you want to slow things down with "Wildest Dreams." But you know that if you play two songs whose tempos differ by more than 10 beats per minute or if you play only a portion of a song, that the crowd will be very disappointed. Instead you'll need to find a list of songs to play to gradually get you to "Wildest Dreams." Your goal is to transition to "Wildest Dreams" as quickly as possible (in terms of seconds).

You have a list of all the songs you can play, their speeds in beats per minute, and the length of the songs in seconds.

(a) Describe a graph you could construct to help you solve the problem. At the very least you'll want to mention what the vertices and edges are, and whether the edges are weighted or unweighted and directed or undirected.

Solution:

Have a vertex for each song. Draw a directed edge from song A to song B if (and only if) song B is slower than A, but the difference between their speeds is at most 10 beats per minute. Add a weight equal to the length of song B to each such edge.

(b) Describe an algorithm to construct your graph from the previous part. You may assume your songs are stored in whatever data structure makes this part easiest. Assume you have access to a method `makeEdge(v1, v2,`

`w`) which creates an edge from `v1` to `v2` of weight `w`.

Solution:

```
foreach(Song s1){
      foreach(Song s2){
           if( s2.bpm < s1.bpm && |s1.bpm - s2.bpm| <= 10)
                insert(s1, s2, s2.songLength)
      }
}
```

As long as our data structure as an efficient iterator this algorithm will run in $O(|V|^2)$ time. If your song is stored in a data structure that can be sorted by bpm, you can increase the speed to $O(S \log S + E)$ where S is the number of songs and E is the number of edges in the resulting graph by adding an early exit to the loop.

(c) Describe an algorithm you could run on the graph you just constructed to find the list of songs you can play to get to "Wildest Dreams" the fastest without disappointing the crowd.

Solution:

Run Dijkstra's from "Shake It Off." When the algorithm finishes, use back pointers from "Wildest Dreams" (and reverse the order) to find the songs to play.

(d) What is the running time of your plan to find the list of songs? You should include the time it would take to construct your graph and to find the list of songs. Give a simplified big-O running time in terms of whatever variables you need.

Solution:

The answer will depend on what you chose in the previous parts. The sorted list approach gives a running time of $O(S \log S + E \log S)$

6. Graph Modeling 2: Snow Day

After 4 snow days this year, UW has decided to improve its snow response plan. Instead of doing "late start" days, they want an "extended passing period" plan. The goal is to clear enough sidewalks that everyone can get from every classroom to every other **eventually** but not necessarily very quickly.

Unfortunately, UW has access to only one snowplow. Your goal is to determine which sidewalks to plow and whether it can be done in time for Kasey's 8:30 AM lecture.

You have a map of campus, with each sidewalk labeled with the time it will take to plow to clear it.

(a) Describe a graph that would help you solve this problem. You will probably want to mention at least what the vertices and edges are, whether the edges are weighted or unweighted, and directed or undirected.

Solution:

Have a vertex for each building and an edge for each section of sidewalk. The edges should be undirected, and weighted by the time it will take the snowplow to clear it.

(b) What algorithm would you run on the graph to figure out which sidewalks to plow? Explain why the output of your algorithm will be able to produce a "extended passing period" plowing plan.

Solution:

Run an MST algorithm (either Kruskal's or Prim's). Whatever edges are chosen are the sidewalks the plow should clear. Why is this valid for the extended passing period plan? For example, why can students get from every classroom to every other?

(c) How can you tell whether the plow can actually clear all the sidewalks in time?

Solution:

Look at the weight of the MST. That's how long it will take to plow. If the plow can start in time to finish by 8:30, then we can start on time!

7. Graph Modeling 3: Video Game

(a) Suppose we are trying to design a maze within a 2d top-down video-game. The world is represented as a grid, where each tile is either an impassable wall, an open space a player can pass through, or a *wormhole*. On each turn, the player may move one space on the grid to any adjacent open tile. If the player is standing on a wormhole, they can instead use their turn to teleport themselves to the other end of the wormhole, which is located somewhere else on the map.

Now, suppose the there are several coins scattered throughout the map. Your goal is to design an algorithm that finds a path between the player and some coin in the fewest number of turns possible.

Describe how you would represent this scenario as a graph (what are the vertices and edges? Is this a weighted or unwighted graph? Directed or undirected?). Then, describe how you would implement an algorithm to complete this task.

Solution:

We can represent this as an undirected, unweighted graph where each tile is a vertex. Edges connect tiles we can travel between. When we have a wormhole, we add an extra edge connecting that wormhole tile to the corresponding end of the wormhole.

Because it takes only one turn to travel to each adjacent tile, there is actually no need to store edge weights: it costs an equal amount to move to the next vertex.

All paths are bidirectional, so we can also use an undirected graph. (If there are paths or wormholes that are one-way, we can switch to using a directed graph).

To find the shortest path, we can run BFS starting with the player and stop the moment we hit a coin.

(We can use other algorithms like DFS or Dijkstra's algorithm if we're careful, but those would be less efficient.)

(b) **Challenge:** Suppose the map now stuffed with a huge number of players. We now want an algorithm that finds the shortest path between every single player and the closest coin.

A naive way of doing this would be to run the same algorithm from the previous part on every single player. However, this would be prohibitively expensive. Design a more efficient algorithm that does the same thing.

When designing your algorithm, you may assume that the map is relatively small/that there are only a few coins, relative to the number of players.

Solution:

One idea is to start at each coin and run BFS or DFS radiating outwards. Every time we visit a tile, save the distance from it to the coin along with a backpointer that leads towards the coin.

If we visit a tile that already has a distance associated with it, see if that distance is higher or lower then the current one. If it's higher, replace it with the shorter distance/the new backpointer. Otherwise, leave it alone.

Now, once we've computed a distance and backpointer for every tile, we can simply just loop through each player, check the backpointer for the tile they're standing on, and move them in that direction.

We now need to run BFS/DFS just once over the entire map, rather then per each player.

Once a player find a coin, we can re-run BFS/DFS again and update the backpointers.