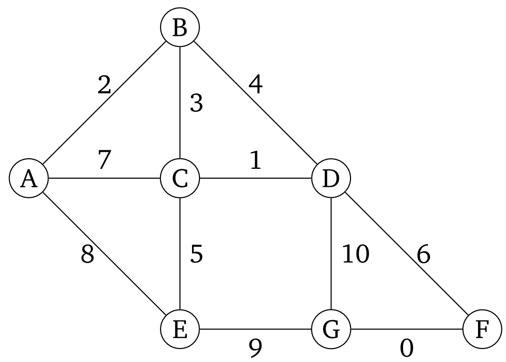
# Section 08: Solutions

## Section Problems

## 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.

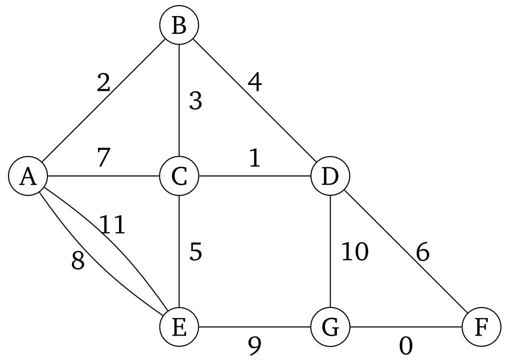
| Step | Components | Edge | Include? |
|---|---|---|---|
| 1 | | (F,G) | |
| 2 | | (C,D) | |
| 3 | | (A,B) | |
| 4 | | (B,C) | |
| 5 | | (B,D) | |
| 6 | | (C,E) | |
| 7 | | (D,F) | |
| 8 | | (A,C) | |
| 9 | | (A,E) | |
| 10 | | (E,G) | |
| 11 | | (D,G) | |

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: Unique Minimum Spanning Trees

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:**

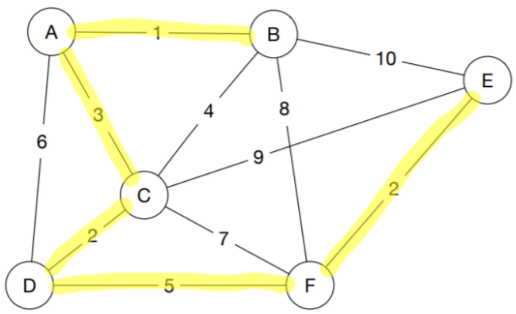
This is true (assuming the initial graph is connected).

## 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 |

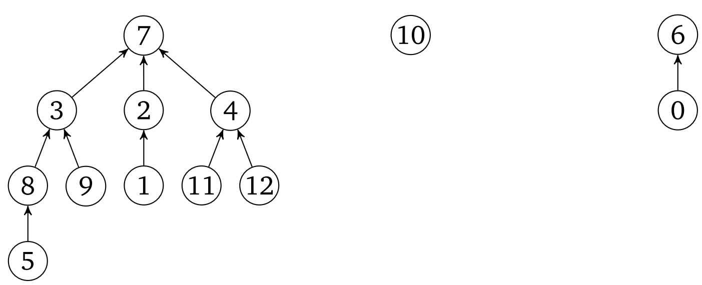
(b) In this graph there are 6 vertices and 11 edges, and the for loop in the code for Kruskal's runs 11 times, a few more times after the MST is found. How would you optimize the pseudocode so the for loop terminates early, as soon as a valid MST is found.

**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

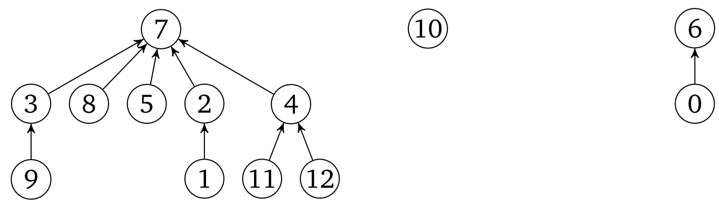
(a) Consider the following trees, which are a part of a disjoint set:




For these problems, use both the **union-by-size** and **path compression** optimizations.

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

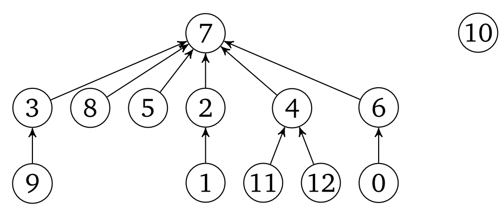
**Solution:**




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

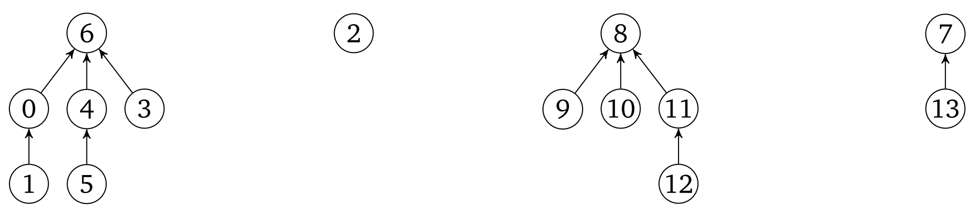
(ii) Draw the final result of calling `union(2,6)` on the result of part (i).

**Solution:**




We attach the second tree to the first since it has a smaller size.

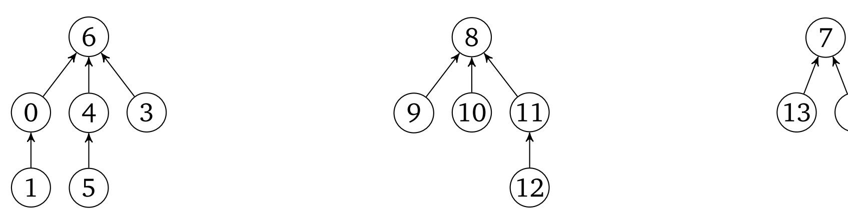
(b) Consider the disjoint-set shown below:




What would be the result of the following calls on union if we add the **union-by-size** and **path compression** optimizations? Draw the forest at each stage.

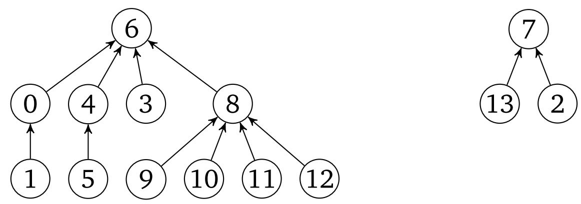
(i) `union(2, 13)`

**Solution:**



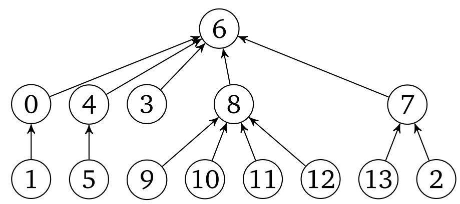
(ii) `union(4, 12)`

**Solution:**



(iii) `union(2, 8)`

**Solution:**



## 5. Graph Modeling 1: Snowed In

After 4 snow days last 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 the first 8:30 AM lecture of the day.

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!

## 6. Graph Modeling 2: Snowden

Consider the following problems, which we can both model and solve as graph problems.

For each problem, describe (i) what your vertices and edges are and (ii) a short (2-3 sentence) description of how to solve the problem.

We will also include more detailed pseudocode in the solutions.

Your description does not need to explain how to implement any of the algorithms we discussed in lecture. However, if you *modify* any of the algorithms we discussed, you must discuss what that modification is.

(a) Suppose you have a bunch of computers networked together (haphazardly) using wires. You want to send a message to every other computer as fast as possible. Unfortunately, some wires are being monitored by some shadowy organization that wants to intercept your messages.

After doing some reconnaissance, you were able to assign each wire a "risk factor" indicating the likelihood that the wire is being monitored. For example, if a wire has a risk factor of zero, it is extremely unlikely to be monitored; if a wire has a risk factor of 10, it is more likely to be monitored. The smallest possible risk factor is 0; there is no largest possible risk factor.

Implement an algorithm that selects wires to send your message such that (a) every computer receives the message and (b) you minimize the total risk factor. The total risk factor is defined as the sum of the risks of all of the wires you use.

**Solution:**

This problem basically boils down to finding the MST of the graph.

**Setup:** We make each computer a node and each wire (with the risk factor) a weighted, undirected edge.

**Algorithm:** Once we form the graph, we can use either Prim's or Kruskal's algorithm as we implemented them in lecture, with no further modifications.

(b) Explain how you would implement an algorithm that finds any computers where sending a message (from a given start computer) would force you to transmit a message over a wire with a risk factor of $k$ or higher.

**Solution:**

**Setup:** We have the same graph as the last part.

**Algorithm:** Run either DFS or BFS on the graph, but modify it so we no longer traverse down edges that have a risk factor of $k$ or higher. We then return all vertices we were unable to visit.

**Pseudocode:**

```
Set<Computer> getAllUnreachable(graph, start, k):
    unreachable = copy(graph.vertices)

    stack = new Stack()
    stack.push(start)

    while stack is not empty:
        curr = stack.pop()
        unreachable.remove(curr)
```

```
        for edge in graph.getNeighbors(start):
            if edge.dest not in unreachable:
                skip iteration (already visited)

            if edge.weight >= k:
                skip iteration (risk factor too high)

            stack.push(edge.dest)

    return unreachable
```