CSCI 3104 Spring 2017 **Problem Set 8**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

1. *(30 pts total) On an overnight camping trip in the Forbidden Forest near the Hogwarts School of Witchcraft and Wizardry, you and your wizard friends Ron and Hermione are woken from a restless sleep by a scream. Crawling out of your tent to investigate, you see a terrified young wizard stumble out of the woods, covered in blood and clutching a crumpled piece of paper to her chest. Reaching your tent, she gasps Get out... while... you..., thrusts the paper into your hands and falls to the ground, dead.*

   Looking at the crumpled paper, you recognize a map of the forest, drawn as an undirected graph, where vertices represent landmarks in the forest, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) Coincidentally, you recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

   On closer examination, you notice that someone (perhaps the dead wizard) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on map's reverse side indicates that these numbers give the probability of encountering a deadly Dementor along the corresponding trail or at the corresponding landmark. The note warns that stepping off the marked trails will surely result in death.

   You glance down at the corpse at your feet. Her death certainly looked painful. On closer examination, you realize that the wizard is not dead at all, or rather, is turning into a Dementor who will surely devour you. After burning the wizard's body, you wisely decide to leave the forest immediately.

   (a) *Give a (small!) example G such that the path from your current location to the EXIT node that minimizes the expected number of encountered Dementors is different from the path that minimizes the probability of encountering any Dementors at all. Explain why, in general, these two criteria lead to different answers.*

   Let each edge x and vertex y represent a Bernoulli Random Variable with probability $w(x)$ or $w(y)$, respectively.

   Expected value: We know from probability that we can get from the expected value of a path over all edges x to the sum of the expected monsters on each path as such,

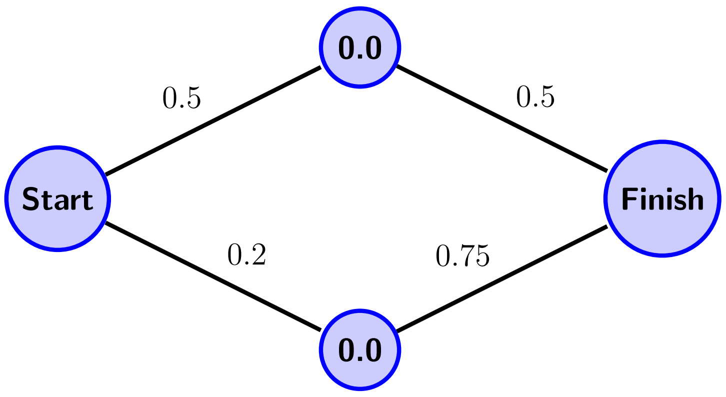
$$E[Path] = E[\sum_x x + \sum_y y] = \sum_x E[x] + \sum_y E[y] = \sum_x w(x) + \sum_y w(y)$$

Because we know that the expectation of a Bernoulli Random Variable is the probability of success.

Encountering any Dementors: The probability of encountering a Dementor on any given vertex x of the path is the probability associated with that path, which we're calling $w(x)$. The probabilities of encountering Dementors on two different paths are independent, but not mutually exclusive events. Therefore the easiest way to find the probability of finding no Dementors is to multiply the probabilities of not seeing a Dementor. Note that since we can also encounter Dementors at clearings, or vertices, we can just combine the next node with the edge we just traversed. Again calling them x and y, the probability of seeing no Dementors is

$$\prod_x (1 - w(x))(1 - w(y))$$

Consider the following graph:



In this case the expected value of the top graph is $0.5 + 0.0 + 0.5 = 1$. So if you traverse the top path many times you will see 1 Dementor on average. The expected value of the bottom path is $0.2 + 0.0 + 0.75 = 0.95$, which means that on average you will see 0.95 Dementors (though you won't see 0.95 on any given trip!). The probability of seeing a Dementor on the top path is $1 - (1 - 0.5)(1 - 0.5)(1 - 0.0) = 1 - (0.5)(0.5)(1) = 1 - 0.25 = 0.75$. This means that %75 of the time you traverse the top path you will see <u>at least</u> one Dementor. The probability of seeing a Dementor on the bottom path is $1-(1-0.2)(1-0.75)(1-0.0) = 1-(0.8)(0.25)(1) = 1-0.2 = 0.8$, so you will see a Dementor %80 of the time on the bottom path. Note how you will see more Dementors less frequently on the top path. In our case, we are just as dead if we encounter 1 Dementor as 2, so we will want to choose the top path.

(b) *Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total expected number of Dementors encountered along the path is as small as possible. Be sure to account for both the vertex probabilities and the edge probabilities.*

Dumbledore's hint: This is clearly an SSSP problem, but you must identify how to reduce the input G to a form that can be solved by SSSP. Remember to include the cost of this transformation in your running-time analysis.

As mentioned above, this is an SSSP problem, but since we know we won't see any Dementors in the first clearing, we have to combine the probabilities of the next vertex with that of the first edge. We account for this in the Relax() function. Instead of just comparing the edge plus the distance of u, we also have to factor in the probability of seeing a Dementor at the vertex. Notice how the vertex's probability is factored in with the edge, and is not overcounted because when a vertex gets relaxed, the new distance only includes the current distance, the edge weight, and the probability of seeing a Dementor at v. The algorithm is correct because it is Dijkstra's Algorithm, adapted to account for weighted vertices. Because the comparison in the Relax() function doesn't increase the asymptotic runtime in any way, the runtime is the same as Dijkstra's, which is $O(E \lg V)$

```
Relax(u, v, w){
    if v.dist > u.dist + w(u, v) + w(v){
        v.dist = u.dist + w(u, v) + w(v)
        v.pred = u
    }
}
```

```
ExpectedValueSSSP(G, w, s) :
    // Initialize all variables
    Q = empty min-heap
    Solved = []
    for each node i{
        d[i] = INF
        pred(i) = NULL
    }
    d[s] = 0
    insert s into Q
    while Q not empty{
        u = Q.extractMin()
        append u to Solved
        // This will be the first and shortest instance of an exit node
        // Note we don't need to solve the whole graph
        if u is Exit{
            return u
        }
        for all edges (u, v){
            Relax(u, v, w)
            insert v into Q
        }
    }
}
```

(c) *Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the probability of encountering any Dementors at all is minimized.*

This is also an SSSP problem. However, instead of using addition to minimize the shortest path, we're going to maximize the probability of running into no Dementors. As mentioned above, finding the probability of encountering at least one Dementor is relatively difficult, and to minimize that we just subtract 1 from the maximum value for encountering no Dementors. Thus, this algorithm is a revised version of Dijkstra's, in which we multiply weights, and find the max value. Note that since all probabilities are less than 1, we will not encounter our equivalent of the "negative weight cycle", where we keep multiplying numbers to increase the value. Appending more edges will always reduce the "distance". Like last time, we are not doing anything to asymptotically change the runtime of Dijkstra's, since the maximization will take as many iterations as a regular minimization. So the runtime will still be $O(E \lg v)$

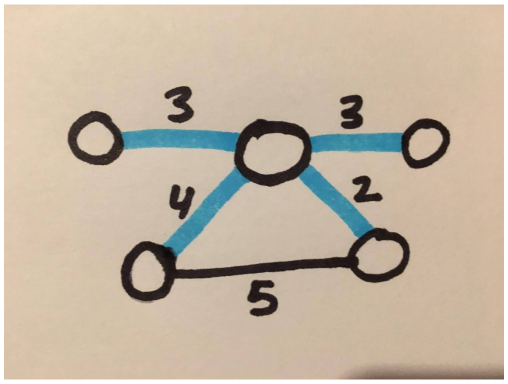
```
Relax(u, v, w){
    if v.dist < u.dist * (1 - w(u, v)) * (1 - w(v))
        v.dist = u.dist * (1 - w(u, v)) * (1 - w(v))
        v.pred = u

}

 ExpectedValueSSSP(G, w, s) :
     // Because we are maximizing
     Q = empty max-heap
     Solved = []
     for each node i{
         d[i] = 0
         pred(i) = NULL
     }
     // There is no probability of seeing Dementors at our camp
     d[s] = 1
     insert s into Q
     while Q not empty{
         u = Q.extractMax()
         append u to Solved
         // When this runs we will find the highest probability
         // of encountering no Dementors
         if u is Exit{
             return u
         }
         for all edges (u, v){
             Relax(u, v, w)
             insert v into Q
         }
     }
 }
```

2. *(30 pts total) In a late-night algorithms study session, you and Draco Malfoy are arguing about the conditions under which a minimum spanning tree is unique. You agree that if all edges in G have unique weights the MST is also unique, but you disagree about how to relax this assumption. Let $w(e)$ be a function that returns the weight of some $e \in E$.*

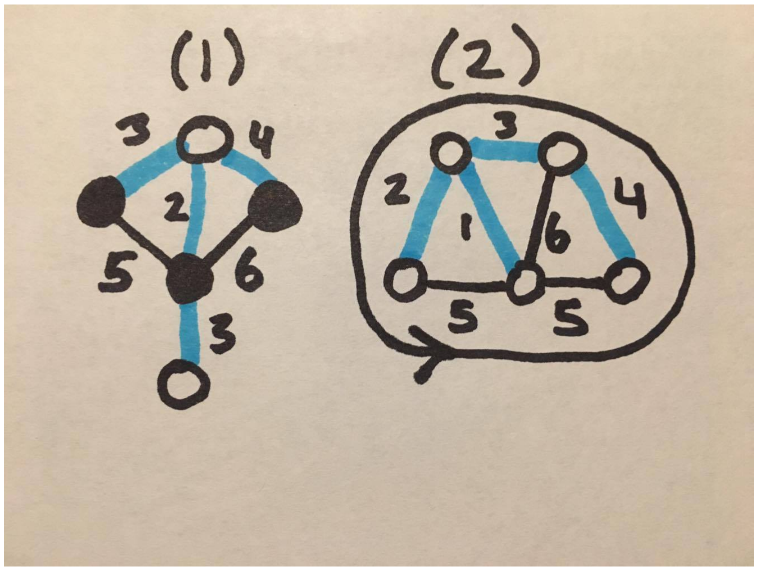
   (a) *Give an example of a (small!) weighted graph that has both a unique MST and some pair of edges e and e' such that $w(e) = w(e') = x$.*

   Above is an example of a weighted graph that has both a unique MST (with tree edges colored blue) and a pair of edges $w(e) = w(e') = 3$.

   (b) *Malfoy claims that the following is true. Prove via (small!) counter examples that it is false.*

   Malfoy's Claim: G has a unique MST if and only if (i) for any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique, and (ii) the maximum-weight edge in any cycle of G is unique.



We will prove that Malfoy's claim is false by examining two different graphs which are counterexamples to his claim.

Counterexample 1 (Graph 1): Malfoy's claim, that G has a unique MST if and only if statement (i) and statement (ii), is equivalent to the claim: if G has a unique MST $\rightarrow$ (statement (i) and (ii)) AND if (statement (i) and (ii)) $\rightarrow G$ has a unique MST. Thus, if we show a counterexample to the statement if G has a unique MST $\rightarrow$ statement (i), then that is enough to disprove Malfoy's entire claim. If we take a look at graph 1, we can see that the graph has a *unique* MST, which has edges colored blue of weights 3 ,4, 2, and 3. This presents a contradiction to statement (i) because, when we partition the vertices of the graph into two different subsets, with one set having white vertices and the other black vertices, we see that we dont have a unique minimum-weight edge that has its endpoints in the two different subsets (we have two edges with weight 1 that satisfy this condition). This is clearly a contradiction to the claim that the minimum-weight edge needs to be unique for any partitioning of the vertices.

Counterexample 2 (Graph 2): Now, we will present another counterexample to Malfoy's claim by proving the statement if G has a unique MST $\rightarrow$ statement (ii) false. Statement (ii) claims that if G has a unique MST, then any cycle of G will have a unique maximum-weight edge. Our example graph shown in diagram 2 has a unique MST that have edges colored blue. If we look at the cycle drawn in diagram 3, which goes through all of the vertices, we can see that there is no unique maximum-weight edge in the cycle. There are two edges in the cycle with weight 5, which is the maximum weight in the cycle. Hence, this is a counterexample to claim(ii), and we have proved Malfoy's claim false.

(c) *Malfoy now demands that you produce the correct relaxed condition, which you claim is the following. Prove that you are correct.*

Your Claim: an edge-weighted graph G has a unique MST T_{mst} if and only if the following conditions hold:
(i) for any bipartition of the vertices induced by removing some edge $e \in T_{mst}$, the minimum-weight edge with one endpoint in each subset is unique, and
(ii) the maximum-weight edge of any cycle constructed by adding one edge f to T_{mst}, where $f \notin T_{mst}$, is unique.

Dumbledore's hint: Note that for any spanning tree T on G, removing some edge $e \in T$ induces a bipartition of the vertices. Consider the edges that span this cut.

To prove this claim we have to prove both the claim that "If an edge-weighted graph has a unique MST $\rightarrow$ statement(i) and statement(ii)", and the claim "If statement(i) and statement(ii) $\rightarrow$ the edge-weighted graph has a unique MST. First proving the If we assume that the graph has a unique MST, then if we remove an edge $e \in T_{mst}$ (which will break the spanning tree into two smaller trees and partition all the vertices $v \in V$ into two sets), then the minimum-weight edge that has an endpoint in both subsets has to be the edge that was just removed because if there was any other minimum-weight edge that connected the separate trees, then that would mean that there's another MST, resulting in a contradiction. So for every different kind of partition that we can make by removing one of the edges in T, the edge that has endpoints in both sets is unique (the removed edge itself). Also, to prove that statement(ii) is true, we consider what happens when we add an edge f to T_{mst} where $f \notin T_{mst}$. Adding such an edge constructs a cycle in some part of the spanning tree because the MST already has the right amount of edges ($V - 1$) to be a spanning tree, and if we add more edges we will go over this amount, creating a cycle somewhere. If our graph has a unique MST, then there *must* be a unique maximum-weight edge in the cycle when we add a non-tree edge to the MST because if there wasnt, then that would mean there is another edge $k \in T_{mst}$ such that $w(k) = w(f)$. If that were the case, however, then edge f could have been part of T_{mst} instead of k, since they have the same weight, and adding f wouldn't have changed $w(T_{mst})$. This contradicts one of our statements that $f \notin T_{mst}$. Thus, we could conclude that if a graph has a unique MST, then statement(ii) needs to be true.

Now we will prove the *only if part*: "If statement(i) and statement(ii) $\rightarrow$ the edge-weighted graph has a unique MST. If statement(i) were true and the graph didnt have a unique MST, then then that would mean there are other minimum-weight edges that can connect the two sets, which would make statement(i) false (a contradiction). Also, if statement(ii) were true and there was no unique MST, then that would mean there is more than one maximum-weight edge in a cycle (also a contradiction). Thus, our claim has been proved.

(d) *Describe and analyze an algorithm that will determine whether an input graph G has a unique MST in (effectively) $O(E \log V)$ time.*
Dumbledore's hint: Think about Kruskal's algorithm.

Description: Our algorithm to find whether a graph G has a unique MST is a slight variation of Kruskal's algorithm. It works like this: The algorithm basically has the same structure as Kruskal's. First, we initialize our MST to NULL and we create a forest of V vertices, where each vertex is a tree and is the first element of its set. Then, we order each edge of G in increasing order, and go through this list of edges. If an edge $(u, v) \in E$ connects two different trees and doesn't violate spanning tree properties (i.e., vertices u and v are in different sets), then we add (u, v) to the spanning tree and merge the vertices in u's set and v's set into one set. But here is how we check if our graph has a unique MST or not: we store the weight of the previous edge in the list in a variable called `prevWeight`, and we also have a boolean variable called `uniqueMST`, which stores `False` if its discovered that G doesn't contain a unique MST, and `True` if it does. If vertices u and v are in the same tree, then we compare $w(u, v)$ to the weight of the previous edge in the list (we cannot add (u, v) to the MST in this case because theres a violation of a property of spanning trees - a cycle). If the weight of the previous edge equals $w(u, v)$, then we know that G cannot have a unique MST because of the claims proven in problem $2c$ - if a cycle induced by adding a non-tree edge into the tree doesn't contain a unique maximum-weight edge, than the graph cannot have a unique spanning tree. Then, we finish up building our MST until the very end and don't worry about finding another violation of unique MSTs. When the algorithm terminates, it returns the MST and the boolean `uniqueMST`.

```
uniqueMST(G):
    MST = NULL
    uniqueMST = True

    for each v in G.V:
        make-set(v)

    for each (u,v) in G.E, order by w(u,v), increasing order:
        if find-set(u) NOT EQUAL to find-set(v):
           MST = MST UNION {(u,v)}
           Union(u,v)
        else if find-set(u) EQUAL to find-set(v) AND uniqueMST = True:
           if w(u,v) = prevWeight:
              uniqueMST = False
        prevWeight = w(u,v)

    return MST, uniqueMST
```

3. *(20 pts) Let G be a graph, w be a weight function for its edges, and T be one of its minimum spanning trees. Now, suppose that we modify G slightly by decreasing the weight of exactly one of the edges in T, producing a new graph G'. Prove that T isalso a minimum spanning tree for G'. More formally, let the edge choice be denoted $(x, y) \in T$, k be a positive number, and define the weight function w' by*

$$w'(u,v) = \begin{cases} w(u,v) & \text{if}(u,v) \neq (x,y) \\ w(x,y) - k & \text{if}(u,v) = (x,y) \end{cases}$$

Prove that T is a minimum spanning tree for G' , whose edge weights are given by w'.

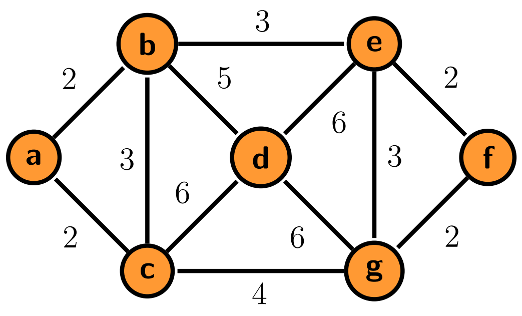
We will prove that T is a minimum spanning tree for G' by using proof by contradiction. We first assume that T is a minimum spanning tree for G but *not* a MST for G'. This would mean that there exists another spanning tree T_2 such that $w^{'}(T_2) < w^{'}(T)$. However, since $w(T) > w^{'}(T)$ this would lead to $w(T_2) < w(T)$, since the weight of every edge other than (x, y) is the same for both G and $G^{'}$. This would mean that there exists a spanning tree of G which has a total weight less than $w(T)$. This would contradict our assumption that T is a MST for G. Therefore, T *has* to be a MST for G'.

4. *(20 pts) Professor Snape gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees and single-source shortest path trees:*

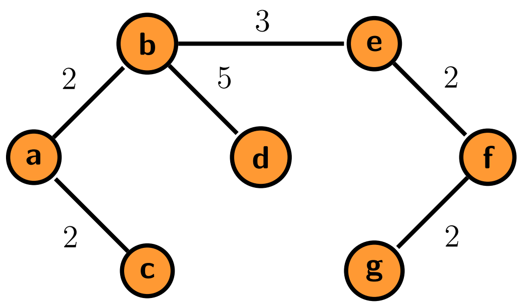
   - *The MST is distinct from any of the seven SSSP trees.*
   - *The order in which Jarník/Prim's algorithm adds the safe edges is different from the order in which Kruskal's algorithm adds them.*
   - *Borůvka's algorithm takes at least two rounds to construct the MST.*

   Justify your solution by (i) giving the edges weights, (ii) showing the corresponding MST and all the SSSP trees, and (iii) giving the order in which edges are added by each of the three algorithms. (For Borůvka's algorithm, be sure to denote which edges are added simultaneously in a single round.)

   Consider the following graph:

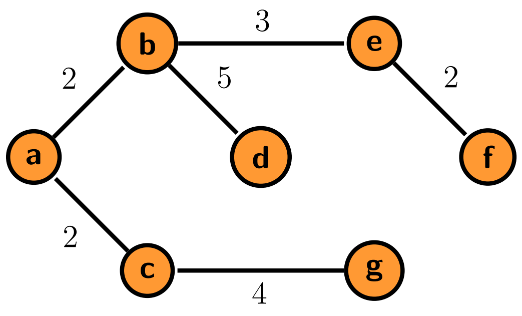


   We would then have the following MST:

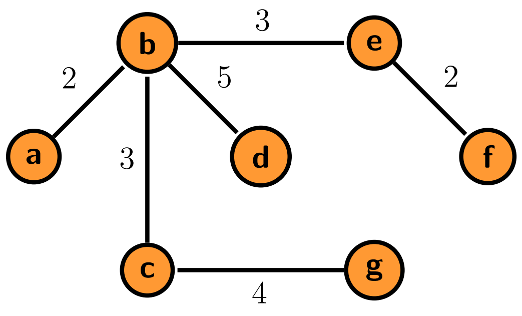


   And the following 7 SSSP trees, starting at a, b, c, d, e, f, and then g, respectively

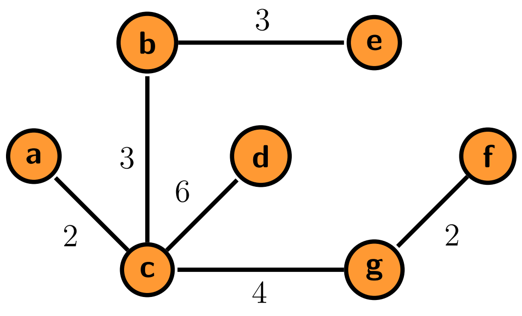
A:



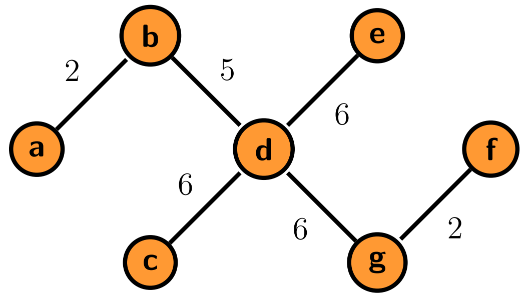
B (Note that inserting (f, g) is same as (c, g), but both are distinct from MST):



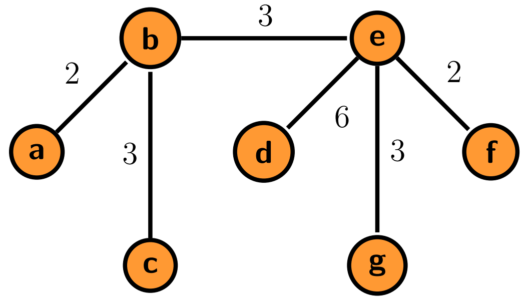
C:



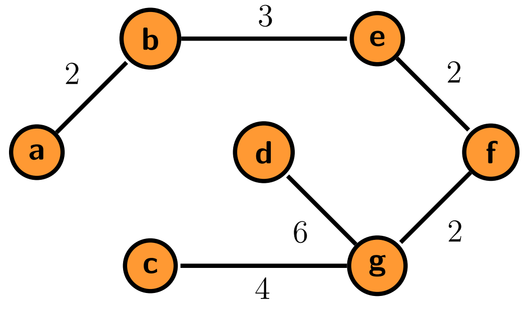
D (Note that inserting (e, f) is same as (g, f), but both are distinct from MST):



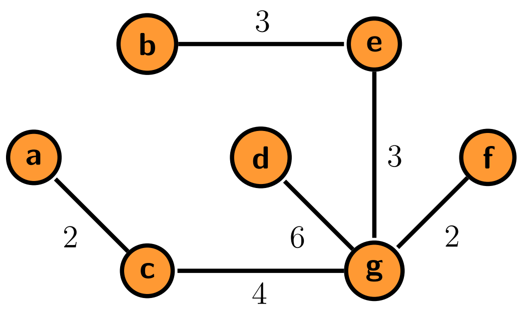
E:



F (Note that inserting (g, d) is same as (e, d), but both are distinct from MST):

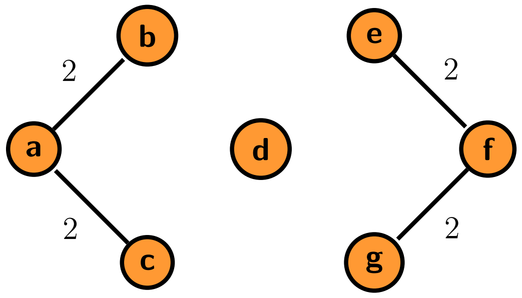


And Finally G:

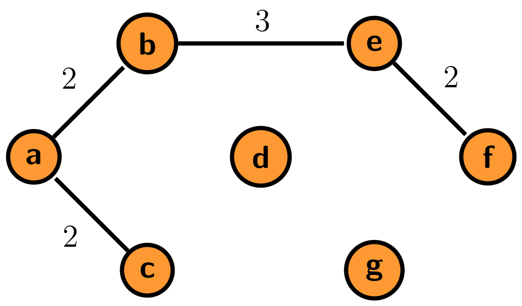


Note how all of the SSSPs are distinct from the MST. This is not a coincidence. Due to the nature of the outside subgraphs, the MST will make the decision to traverse the two outside edges for the smaller overall weight, while the SSSP will simply create a new node on the inside because it is immediately a shorter path to the node.

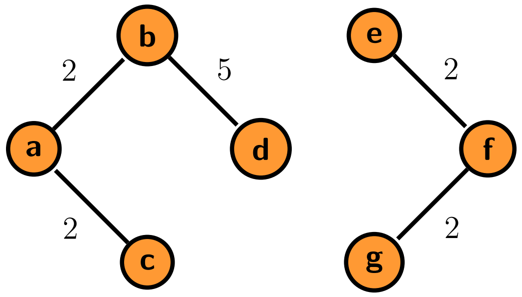
Now we know that Prim's algorithm will add edges in a different order than Kruskal's algorithm because Kruskal's algorithm will add the four outside edges in the first four steps because they are the smallest weight unconnected edges. So after 4 rounds the graph will look like this:



For example, if we start at a, Prim's algorithm will look like the following after 4 rounds:



Finally, Borůvka's Algorithm will construct the following after one round:



Which is not the MST, so the algorithm will require at least one more round (it will finish in exactly one more round). Therefore, the MST is distinct from any of the seven SSSP trees, the order in which Prim's algorithm adds safe edges is different from the order in which Kruskal's algorithm adds them, and Borůvka's algorithm takes at least two rounds to construct the MST

5. *(10 pts extra credit) Currency arbitrage is a form of financial trading that uses discrepancies in foreign currency exchange rates to transform one unit of some currency into more than one unit of the same currency. For instance, suppose 1 U.S. dollar bought 0.82 Euro, 1 Euro bought 129.7 Japanese Yen, 1 Japanese Yen bought 12 Turkish Lira and one Turkish Lira bought 0.0008 U.S. dollars. Then, by converting currencies, a trader could start with 1 U.S. dollar and buy* $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ *U.S. dollars, thus turning a 2% profit. Of course, this is not how real currency markets work because each transaction must pay a commission to a middle-man for making the deal.*

   Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i,j]$ units of currency c_j. A traditional arbitrage opportunity is thus a cycle in the induced graph such that the product of the edge weights is greater than unity. That is, a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1,i_2] \times R[i_2,i_3] \times \ldots \times R[i_{k-1},i_k] \times R[i_k,i_1] > 1$. Each transaction, however, must pay a commission, which is typically some α fraction of the transaction value, e.g., $\alpha = 0.01$ for a 1% rate.

   (a) *Give an efficient algorithm to determine whether or not there exists such an arbitrage opportunity, given a commission rate α. Analyze the running time of your algorithm.*

   Dumbledore's hint: It is possible to solve this problem in $O(n^3)$. Recall that Bellman-Ford can be used to detect negative-weight cycles in a graph.

   One way to solve this problem is to think about it as a graph problem. We are trying to find exchange transactions such that $R[i_1,i_2] \times R[i_2,i_3] \times \ldots \times R[i_{k-1},i_k] \times R[i_k,i_1] > 1$. We can represent the currencies $c_1, c_2, \ldots, c_n$ as the vertices of the graph and each of the vertices can be connected by directed edges which represent the exchange rates between them. We can make use of negative cycles and the Bellman-Ford algorithm which can find negative cycles. However, that would mean that we would have to convert the problem from a multiplication problem where we have to find a value greater than one to an addition problem where we have to find a value less than 0. Changing the value of each transaction would take $O(n^2)$ time and running the graph through Bellman-Ford's algorithm will take $O(n^3)$, giving us a time complexity of $\underline{O(n^3)}$.

   (b) Explain what effect varying α has on the structure of the set of possible arbitrage opportunities your algorithm might identify.

   If the values for α vary, then there will be a change in the possible arbitrage opportunities our algorithm might identify. For example, if the commission rates bring up the sum of the edge weights in a cycle above 0, then that would mean we no longer have a negative cycle. Hence, for such cases, we will not have an arbitrage opportunity.