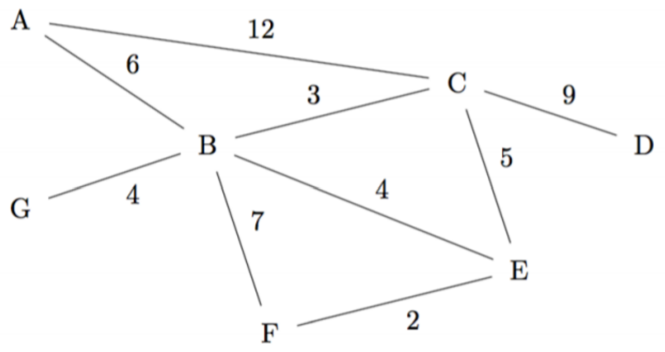
CS 61B Fall 2021 — More Shortest Paths and MSTs

Exam Prep Discussion 14: November 29, 2021

1 Introduction to MSTs



(a) For the graph above, list the edges in the order they're added to the MST by Kruskal's and Prim's algorithm. Assume Prim's algorithm starts at vertex A. Assume ties are broken in alphabetical order. Denote each edge as a pair of vertices (e.g. AB is the edge from A to B)

Prim's algorithm order:
Kruskal's algorithm order:

Prim's algorithm order: AB, BC, BE, EF, BG, CD
Kruskal's algorithm order: EF, BC, BE, BG, AB, CD

(b) Is there any vertex for which the shortest paths tree from that vertex is the same as your Prim MST? If there are multiple viable vertices, list all.

Vertex B, A, or G

(c) True/False: Adding 1 to the smallest edge of a graph G with unique edge weights must change the total weight of its MST

True, either this smallest edge (now with weight +1) is included, or this smallest edge is not included and some larger edge takes its place since there was no other edge of equal weight. Either way, the total weight increases.

(d) True/False: The shortest path from vertex A to vertex B in a graph G is the same as the shortest path from A to B using only edges in T, where T is the MST of G.

False, consider vertices C and E in the graph above

(e) True/False: Given any cut, the maximum-weight crossing edge is in the maximum spanning tree.

True, we can use the cut-property proof as seen in class, but replace ”smallest” with ”largest”.

2 Multiple MSTs

Recall a graph can have multiple MSTs if there are multiple spanning trees of minimum weight.

(a) For each subpart below, select the correct option and justify your answer. If you select “never” or “always,” provide a short explanation. If you select “sometimes”, provide two graphs that fulfill the given properties — one with multiple MSTs and one without. Assume G is an undirected, connected graph.

1. If **none** the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ◯ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

2. If **some** of the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ◯ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

3. If **all** of the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ◯ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

Solution:

1. If **none** the edge weights are **identical**, there will

   [✓ never be multiple MSTs in G.]

   sometimes be multiple MSTs in G.

   always be multiple MSTs in G.

Justification:
To prove this, we can leverage the cut property. Recall the cut property states that the cheapest edge in any cut is in *some* MST. However, if the cheapest edge in any cut is unique, then we get a stronger claim — the cheapest edge must be in *the* MST. As such, if none of the edge weights are identical, i.e. they are all unique, then the cheapest edge in any cut will always be unique, and we will only have one MST.

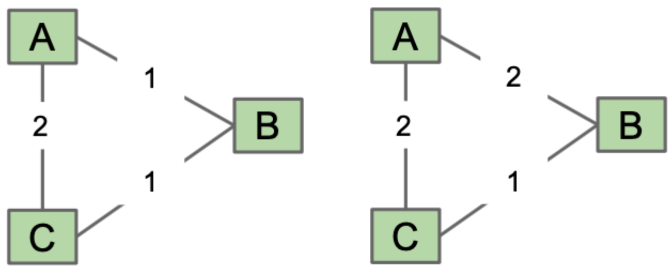
2. If **some** of the edge weights are **identical**, there will

   never be multiple MSTs in G.

   [✓ sometimes be multiple MSTs in G.]

   always be multiple MSTs in G.

Justification:



In the graph on the left, the only MST is [AB, BC]. In the graph on the right, two MSTs exist — [AB, BC] and [AC, BC].

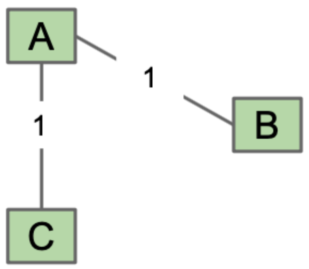
3. If **all** of the edge weights are **identical**, there will

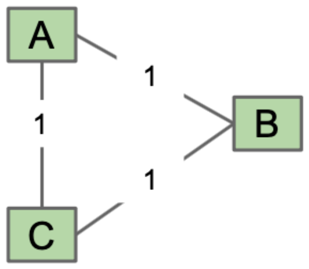
   never be multiple MSTs in G.

   [✓ sometimes be multiple MSTs in G.]

   always be multiple MSTs in G.

Justification:





In the graph on the left, the only MST is [AB, AC]. Note that for any tree, we only have one MST, since the tree itself is the MST! In the graph on the right, three MSTs exist — [AB, BC], [AC, BC], and [AB, AC].

(b) Suppose we have a connected, undirected graph G with N vertices and N edges, where all the **edge weights are identical**. Find the maximum and minimum number of MSTs in G and explain your reasoning.

Minimum: _________
Maximum: _________

Justification:

Solution: Minimum: 3, Maximum: N

Justification: Notice that if all the edge weights are the same, an MST is just a spanning tree. Let's begin by creating a tree, i.e. a connected graph with $N-1$ edges. Now, notice that there is only one spanning tree, since the graph is itself a tree.

As such, the problem reduces to: how many spanning trees can the insertion of one edge create? If we add an edge to a tree, it will create a cycle that can be of length at minimum 3 and at maximum N. Then, notice that we can only remove **any** edge from a cycle to create a spanning tree, so we have at minimum 3 and at maximum N possible MSTs in G.

(c) It is possible that Prim's and Kruskal's find **different** MSTs on the same graph `G` (as an added exercise, construct a graph where this is the case!). Given any graph `G` with integer edge weights, modify G to **ensure** that Prim's and Kruskal's will always find the same MST. You may not modify Prim's or

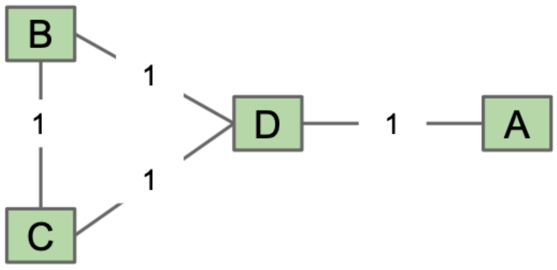
Kruskal's.
Hint: Look at subpart 1 of part a.

Solution: To ensure that Prim's and Kruskal's will always produce the same MST, notice that if `G` has unique edges, only one MST can exist, and Prim's and Kruskal's will always find that MST! So, what if we modify `G` to ensure that all the edge weights are unique?

To achieve this, let's strategically add a small, unique `offset` between 0 and 1, exclusive, to each edge. It is important that we choose an `offset` between 0 and 1 so that this added value doesn't change the MST, since all the edge weights are integers. It is also important that the offset is unique for each edge, because then we ensure each weight is distinct. Pseudocode for such a change is shown below:

```
E = number of edges in the graph
offset = 0
for edge in graph:
    edge.weight += offset
    offset += 1 / E
```

In regard to the added exercise, here is a simple graph G where Prim's and Kruskal's produce different MSTs. Prim's starting from A will select AD, BD, and CD, whereas Kruskals will select AD, BC, and BD.



3 Graph Algorithm Design

Here is a video walkthrough of the solutions. Note that the order of the subparts have changed. In the video, we first go over part c, then part a, then part b, and finally part d.

Given a **undirected**, **weighted** graph G with **positive**, **integer** edge weights, we want to find a path from **u** to **v** that minimizes the total cost. For each "catch" below, find the path of optimal cost no slower than $O(ElogV)$.

(a) Excluding the start and end vertex, we partition the vertices into 5 subsets, and we must visit vertices in order of their subset. That is, if we are in subset **k**, the next vertex we visit must be in subset **k + 1**.

Solution:
Modify Dijkstra's algorithm so that when we visit a vertex in a subset **k**, we *only* consider neighbors in the subset **k + 1**.

Alternate Solution:
Modify the graph by removing all edges that do **not** connect vertices of adjacent subsets. Next, for each remaining edge, we know it must connect vertices in adjacent subsets, let's call these subsets **k** and **k + 1**. Replace each undirected edge with a directed edge from **k** to **k + 1**. Run Dijkstra's from **u** to **v**.

(b) We must visit two designated vertices s and k on our path.

Explanation:
Notice that the shortest path from **u** to **v** will either go **u** → **s** → **k** → **v** or **u** → **k** → **s** → **v**, where **s** → **t** corresponds to taking a path from **s** to **t**. Since we want the final path of minimum cost, each of these paths should be shortest paths. As such, we want the following shortest paths:

- **u** → **s**
- **s** → **k**
- **k** → **v**
- **u** → **k**
- **k** → **s**
- **s** → **v**

However, since the graph is undirected, we know that the shortest path from **s** to **k** is the same as the shortest path from **k** to **s**, and we can reduce the required shortest paths to the below:

- **s** → **u**
- **s** → **k**
- **s** → **v**
- **k** → **v**
- **k** → **v**

Well, we did a bit more than "reduce" in the step above, since we also changed the order of some of the shortest paths to highlight that all of the shortest paths we need either start from **s** or **k**.

Solution:

Run Dijkstra's from **s** and from **k** to find the needed shortest paths in the previous list. Plug in the calculated shortest paths into the expressions below.

1. $\mathbf{u} \rightarrow \mathbf{s} \rightarrow \mathbf{k} \rightarrow \mathbf{v}$
2. $\mathbf{u} \rightarrow \mathbf{k} \rightarrow \mathbf{s} \rightarrow \mathbf{v}$

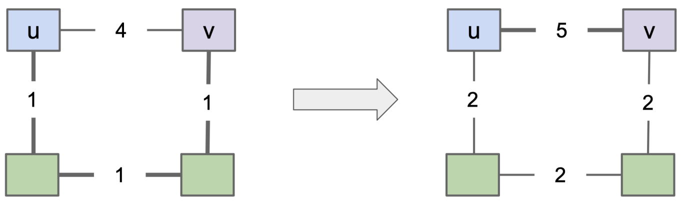
Return the path with the minimum total weight.

(c) If two paths from u to v are of the same cost, we will choose the path with fewer edges.

Solution:

Add $1/E$ to the weight of each edge where E is the number of vertices in the graph. Run Dijkstra's from **u** to **v**.

So, why does this work, and where did we get the idea to add $1/E$ to each edge? Let's begin with the second question. Intuitively, adding a little to each edge **discourages** taking paths with many edges. So why not add 1 to each edge? Looking at the graph below, if we add 1 to every edge, we have now **changed** the shortest path from **u** to **v**! The *only* purpose of the added weight should bes to break ties between two paths of equal length.



Okay, so we want the amount added to be really small, so why not 0.001, or even 0.00001? The problem with any constant is that the same problem showed above may occur for a graph that is really, really big.

Okay, so what if we add an offset that takes in consideration the number of edges in the graph, like $1/E$ (or anything smaller than this proportional to E, e.g. $1/E^2$).

This would work! If we ever have two paths of equal cost, the smallest one path can be is 1 edge and the largest the other path can be is $E-1$ edges. On the larger path, notice that the sum of all the offsets comes out to $(E-1)/E$, which is less than 1! Thus, since we are using **integer** edge weights, this added offset can only serve to break ties, and is not susceptible to the problem described above.

(d) Instead of starting from u and ending at v, we can start from any vertex in a subset of vertices and end at any vertex in a subset of vertices. Each subset is of size k.

Solution:
Create two dummy nodes d_1 and d_2. Connect d_1 to every vertex in the start subset with an edge of zero weight. Connect d_2 to every vertex in the ending subset with an edge of zero weight. Run Dijkstra's from d_1 to d_2. Ignore the edges connected to d_1 and d_2 in the shortest path.