CS 61B Spring 2021 — MSTs, Dijkstra's, and Tries

Exam Prep Discussion 11: April 5, 2021

1 Oracle Dijkstra's

In some graph G, we are given a sorted list of nodes, sorted by their distances from some start vertex A. Design an *efficient* algorithm to find the shortest paths tree starting from A.
Hint: Your algorithm should be more efficient than Dijkstra's.

Solution: This algorithm essentially removes the purpose of the priority queue in normal Dijkstra's. When a node is removed from the PQ normally, this signifies we have found the shortest path from the source to that node, AND that this node is the next closest node to the source that hasn't been visited yet. In a sorted list of nodes, we can simply traverse through the nodes in order. Therefore, our algorithm is simply to run Dijkstra's, but instead of keeping a priority queue we go through our sorted list of nodes in order. Our runtime is O(V+E).

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

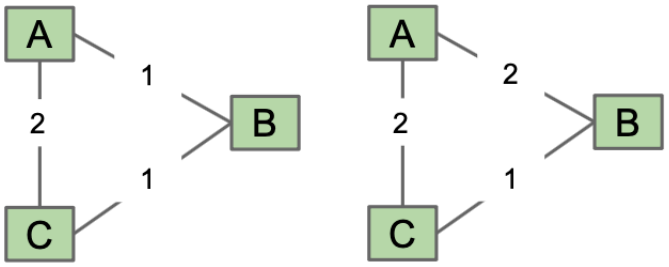
   - [✓ never be multiple MSTs in G.]
   - ○ sometimes be multiple MSTs in G.
   - ○ always be multiple MSTs in G.

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:

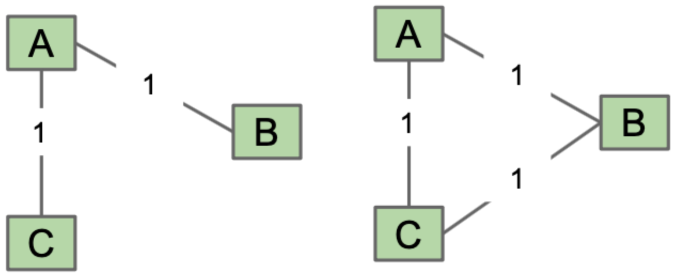


In the graph on the left, the only MST is [AB, BC]. In the graph on the right, two MSTs exist — [AB, BC] and [AC, BC].

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.

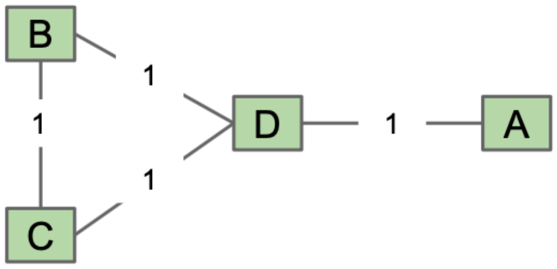
(c) It is possible that Prim's and Kruskal's find **different** MSTs on the same graph `G` (as an added exercise, construct a graph where this is the case!). Given any graph `G` with integer edge weights, modify G to **ensure** that Prim's and Kruskal's will always find the same MST. You may not modify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

For each of the following scenarios, write a brief description for an algorithm for finding the MST in an undirected, connected graph G.

(a) If all edges have edge weight 1. Hint: Runtime is O(V+E)

The key idea here is that any tree which connects all nodes is an MST. We can run DFS and take the DFS tree. You could also take a BFS tree, or run Prim's algorithm with a queue or stack instead of a priority queue (this would be equivalent to BFS/DFS). Unfortunately, a modified Kruskal's will be slightly slower, because even if we don't need to sort edges, the union-find operations will take additional time.

(b) If all edges have edge weight 1 or 2. Hint: Use your algorithm from part (a)

Remove weight 2 edges from the graph so only weight 1 edges remain. Now run an algorithm from part (a) as far as possible (e.g. find a DFS forest). We will have some number of connected components. Use these connected components as nodes in a new graph G*. Look at the weight 2 edges in G. For each edge, if the nodes containing the two endpoints are not already connected in G*, add an edge between the two containing nodes in G*. Now we can run our algorithm from part (a) again to complete the MST.

4 A Wordsearch

Given an N by N wordsearch and N words, devise an algorithm to solve the wordsearch in O(N^3). For simplicity, assume no word is contained within another, i.e. if the word "bear" is given, "be" wouldn't also be given.

If you are unfamiliar with wordsearches or want to gain some wordsearch solving intuition, see below for an example wordsearch. Note that the below wordsearch doesn't follow the precise specification of an N by N wordsearch with N words, but your algorithm should work on this wordsearch regardless.

Example Wordsearch:

C	M	U	H	O	S	A	E	D
T	R	A	T	H	A	N	K	A
O	C	Y	E	S	R	T	U	T
N	I	R	S	A	I	O	L	S
Y	R	R	M	T	N	N	H	R
Y	E	A	E	V	A	R	U	E
A	A	A	I	M	E	L	C	R
N	H	D	J	Y	U	A	C	I
T	Y	S	A	A	R	S	U	C
A	R	S	I	G	Y	E	S	A

ajay	anton
crystal	eric
grace	isha
luke	naama
rica	sarina
sherry	shreyas
sohum	sumer
tony	vidya

Hint: Add the words to a Trie, and you may find the `longestPrefixOf` operation helpful. Recall that `longestPrefixOf` accepts a `String key` and returns the longest prefix of `key` that exists in the `Trie`, or **`null`** if no prefix exists.

Algorithm: Begin by adding all the words we are querying for into a `Trie`. Next, we will iterate through each letter in the wordsearch and see if any words *start* with that letter. For a word to start with a given letter, note that it can go in one of eight directions — N, NE, E, SE, S, SW, W, NW.

Looking at each direction, we will check if the string going in that direction has a prefix that exists in our `Trie`, which we can do using `longestPrefixOf`. Note that words are not nested inside of others, so *at most* one word can start from a given letter in a given direction. As such, if `longestPrefixOf` returns a word, we know it is the only word that goes in that direction from that letter.

For instance, if we are at the letter "S" in the middle of the top row of the wordsearch above and are considering the direction west, we would want to see if the string `"SOHUMC"` has a prefix that exists in the given wordsearch. To efficiently perform this query, we call `longestPrefixOf("SOHUMC")`, which, in this case, returns `"SOHUM"`, and we proceed by removing `"SOHUM"` from our `Trie` to signal that we found the word `"SOHUM"`.

We will repeat this process until the all the words have been found, i.e. when the `Trie` is empty. Finally, note that this is a very open ended problem, so this is one of *many* possible solutions.

Runtime: We look at N^2 letters. At each letter, we execute eight calls to `longestPrefixOf` which runs in time linear to the length of the inputted string, which can be of at most length N, since that is the height and width of the wordsearch. Thus, if we perform on the order of N work per letter and we look at N^2 letters, the runtime is $O(N^3)$.