CS 61B Spring 2024

Graphs II, Tries

Exam-Level 10: April 1, 2024

1 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 with at least 3 vertices.

1. 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:

2. 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:

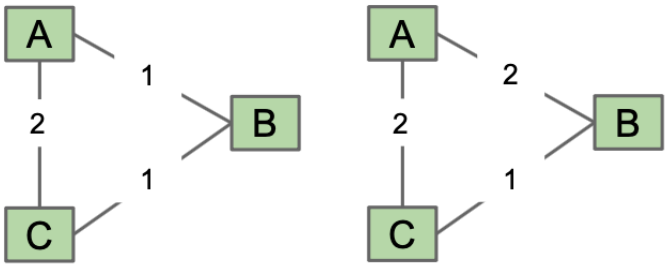
1. If **some** of the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ■ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

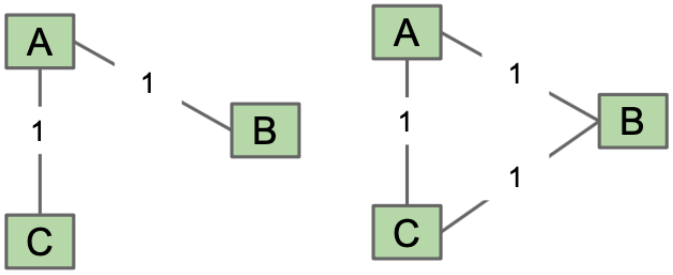


In the graph on the left, the only MST is [AB, BC]. In the graph on the right, two MSTs exist — [AB, BC] and [AC, BC].

2. 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.

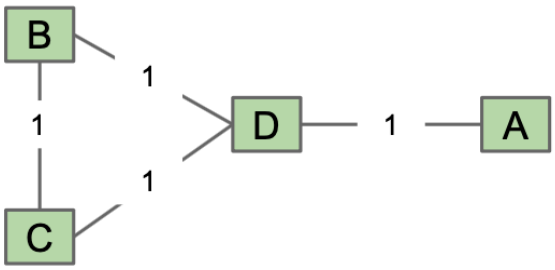
(c) It is possible that Prim's and Kruskal's find **different** MSTs on the same graph `G` (as an added exercise, construct a graph where this is the case!). Given any graph `G` with integer edge weights, modify the edge weights of G to **ensure** that (1) Prim's and Kruskal's will output the same results, and (2) the output edges still form a MST correctly in the original graph. You may not modify Prim's or Kruskal's, and you may not add or remove any nodes/edges.
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. This is to ensure that the edges picked in the modified graph is still a correct MST in the original graph, 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.



2 Topological Sorting for Cats

The big brain cat, Duncan, is currently studying topological sorts! However, he has a variety of **curiosities** that he wishes to satisfy.

(a) Describe at a high level in plain English how to perform a topological sort using an algorithm we already know (hint: it involves DFS), and provide the time complexity.

Solution: Reverse the edges of the graph, then perform a postorder traversal from any unvisited vertices until you visit all of them. The runtime is linear with respect to the number of vertices and edges.

(b) Duncan came up with another way to possibly do topological sorts, and he wants you to check him on its correctness and tell him if it is more efficient than our current way! Let's derive the algorithm.

1. First, provide a logical reasoning for the following claim (or a proof!): Every DAG has at least one

source node, and at least one sink node.

Solution: Suppose a DAG had no sinks. Then every node has a outgoing edge. If we traversed the graph, then given that there are no sinks, there is always another vertex we can go to from the current one, and therefore after —V— steps we must be repeating vertices. Therefore, the graph has a cycle, and is not a DAG. Suppose a DAG had no sources. Then every node has at least one edge going into it. Note that then, if we reverse the graph, this graph has no sinks, but should still be a DAG. However, we proved in the previous part this cannot be the case, so this too cannot be a DAG.

2. Duncan wishes to extend from the Graph class to create a DAG class. He wants to eventually add a method that enables topological sorting, but needs to write some helper methods first! Complete the following instance methods `computeInDegrees` and `findAllSourceNodes()`.

Solution:

```
public class Graph {
    public Graph(int V) // Create empty graph with v vertices, numbered 0 to V - 1
    public void addEdge(int v, int w) // Adds edge from v to w
    Iterable<Integer> adj(int v) // Gets vertices adjacent to v
    int V() // Number of vertices
    int E() // Number of edges
}

public class DAG extends Graph {
    // Computes the number of incoming edges to a vertex
    public int[] computeInDegrees() {
        int[] indegree = new int[V()];
        for (int i = 0; i < V(); i++) {
            for (int w : adj(i)) {
                indegree[w]++;
            }
        }
        return indegree;
    }

    // Finds all source nodes in the graph
    public List<Integer> findAllSourceNodes(int[] indegree) {
        List<Integer> sources = new ArrayList<>();
        for (int i = 0; i < V(); i++) {
            if (indegree[i] == 0) {
                sources.add(i);
            }
        }
        return sources;
    }
}
```

The idea behind `computeInDegrees()` is to create an array of size V, where `indegree[i]` represents

the number of incoming edges to vertex i. Then, we consider each edge coming out of each vertex. Each time, we increment the indegree of the destination vertex by one.

The idea behind `findAllSourceNodes()` is to iterate through the `indegree` array, and if the `indegree` of a vertex is 0, add it to the list of sources.

The runtime of `computeInDegrees` is $O(V + E)$, and the runtime of `findAllSourceNodes` is $O(V)$. This is because we iterate through all the vertices and edges in the graph in the first method, and only the vertices in the second method.

3. Now, make the following observation: If we remove all of the source nodes from a DAG, we are guaranteed to have at least one new source node. Inspired by this fact, and using the previous parts, complete the `topologicalSort()` method. What is its runtime?

   Solution:

```
public class DAG extends Graph {
    public int[] computeInDegrees() { ... }
    public List<Integer> findAllSourceNodes(int[] indegree) { ... }

    public List<Integer> topologicalSort() {
        List<Integer> sorted = new ArrayList<>();
        int[] indegree = computeInDegrees();

        // Hint: add elements from another iterable here
        Queue<Integer> sources = new ArrayDeque<>(findAllSourceNodes(indegree));

        while (!sources.isEmpty()) {

            int source = sources.poll();
            sorted.add(source);

            for (int w : adj(source)) {
                indegree[w]--;
                if (indegree[w] == 0) {
                    sources.add(w);
                }
            }
        }
    }
}
```

   The algorithm is as follows. Create the indegree array described in the previous part. Get all of the source nodes from it, which are guaranteed to exist by part a. Add them to a queue. Then, until the set is queue, remove any vertex from the queue, output it, and "removes" it from the graph (we are not really removing the edge and vertex here, just decrementing the array `indegree`): For each edge that that vertex has outgoing, decrement the indegree of the destination vertex by one. If that vertex now has 0 indegree in the array, add it to the queue.

   The runtime of `topologicalSort` is $O(V + E)$. We made one call to `computeInDegrees`, which is $O(V + E)$, and `findAllSourceNodes`, which is $O(V)$. The while loop attempts to "remove" all vertices and its outgoing edges from the graph when it becomes a source, and will run in $O(V + E)$ time.

   Wow, it actually works! Nice work Duncan! However, this algorithm has already been discovered and is known as Kahn's algorithm, and so we cannot call it Duncan's algorithm :(

4. Venti, the bard from Mondstadt is allergic to cats. He wanted to trick Duncan and created a DAG object, but it actually represents a graph with a cycle! How can you modify the method `topologicalSort()` above to detect whether the graph has a cycle?

Solution: If the graph has a cycle, at some point we will not be able to find any more sources, but there will still be things that we have not "removed" from the graph. Therefore, we can check if there are any non-zero elements in the `indegree` array after the while loop. If there are, then the graph has a cycle.

3 A Wordsearch

Given an N by N wordsearch and N words, devise an algorithm (using pseudocode or describe it in plain English) 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.

Solution: 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 $O(N^3)$.