Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

CMPT 706 - ASSIGNMENT 4

Question 1:

Some of your friends have gotten into the burgeoning field of time series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges — what is being bought — are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them — for example, they want to know if the four events

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible events (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a subsequence of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S' . So, for example, the sequence of four events above is a subsequence of the sequence

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up a short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time O(m + n) whether S' is a subsequence of S.

Solution:

Given: Two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time O(m + n) whether S' is a subsequence of S.

Algorithm:

CheckSubSequenceOfSequence(S, S'):

```
1. set value of i and j to 1
2. when i <= S.length &  j <= S'.length
    a. If S[i] is the same as S'[j] , S[i] == S'[j]
        i. then increment i and  j by 1, j := j + 1 and i := i + 1
        b. Else
            i. just increment i, i := i + 1
3. if j > S'.length and j != 0
    a. Return true
4. else
    a. Return false
```

<u>Running time analysis:</u>

If we see the proposed algorithm it will run in linear time of input's size. Since, there might be cases where S.length == S'.length and S.length > S'.length.

CASE 1: S.length == S'.length

In the this case it would be m= n, where S.length = m and S'.length = n.

Therefore running time in such a situation would be O(m + n) = O(2m) = O(m).

But since its given in the question provided that 'Give an algorithm that takes two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time O(m + n)'. We can ignore this case.

CASE 2: S.length > S'.length

In the this case it would be m > n, where S.length = m and S'.length = n.

Therefore running time in such a situation would be O(m + n).

Why m + n? Because the loop that checks greedily for event by event match of both sequences at each step in order but not consequently occuring would take O(m + n) time in the worst case scenario. The worst case scenario is when we traverse through all elements in S and all elements in S' to either find a sub-sequence or in not finding one.

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Question 2:

Let G = (V, E) be an undirected graph with costs ce ≥ 0 on the edges e ∈ E. Assume you are given a minimum cost spanning tree T of G. Now assume that a new edge is added to G, connecting two nodes v, w ∈ V with cost c.

(a) *Give an efficient algorithm to test if T remains the minimum cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time O(|E|). Can you do it in O(|V |) time.*

Solution:

Given: T(V, E') is a Minimum cost spanning tree of Graph- G(V, E). A new edge e (v, w) is added to G but not to T, where new edge is added to G connecting two nodes v, w ∈ V with cost c.

Algorithm:

CheckIfMST(T, e (v, w, c)):

1. T with v as the current node. startnode = v and endnode = w
2. Weight = c, result = false
3. Do BFS For all e(startnode,t,c) ∈ E' and not (endnode == source(e(v,t,c))), that is when the source node of an edge become our destination node- stop, as we would have then traversed a complete path from v to w.
4. If in the traversed BFS path from v to w there be an edge that has greater weight than 'Weight' => if weight(e) > Weight, (as both edges from v to w cannot be present in MST, since then it will form a cycle- so only either of these should give us the shortest path from v to w.)

   'Print not MST'

5. Else

   'Print MST'

Running time analysis:

This algorithm will run in O(|E|) time as its only going to traverse through the edges on the path from v to w in our MST T, so the worst case would be when the root node is the start node and the leaf node at the highest depth of the tree is the destination node of the path to be travelled.

It can be done in O(|V|) time-> if we instead replace the edge with the longest edge in the cycle formed due to the new added edge e to G for T.

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

(b) *Suppose T is no longer a minimum cost spanning tree. Give a linear time algorithm (time O(|E|) to update the tree T to a new minimum cost spanning tree.*

Solution:
We can modify T(V, E’) to obtain a new MST by deleting the maximum weighted edge on the desired path and replacing it with the new edge. Using the graph for T, we can do a recursive tree traversal in T, starting at vertex v. Once the traversal reaches w, we can open up the recursion, and we look for the highest weighted edge along the path v-w.

Alogrithm:
MakeMST(T, e (v, w, c)):

5. T with v as the current node. startnode = v and endnode = w
6. Weight = c, result = false
7. Do BFS For all e(startnode,t,c) ∈ E’ and not (endnode == source(e(v,t,c))), that is when the source node of an edge become our destination node- stop, as we would have then traversed a complete path from v to w.
8. If in the traversed BFS path from v to w there be an edge that has greater weight than ‘Weight’ => if weight(e) > Weight, (as both edges from v to w cannot be present in MST, since then it will form a cycle- so only either of these should give us the shortest path from v to w.)
    a. Find the highest of such weighted edge between v and w (our desired path here).
9. Replace the found highest weighted edge with the new edge e(v,w,c).
10. Print ‘ MST created’.

Running time analysis:
The running time of the above algorithm will be O(|E|):

The tree traversal part requires O(|E|) (as it is linear in the number of edges of T) time and the update part can be done in constant time O(1).

So, overall running time complexity will be = O(|E|) + O(1) = O(|E|).

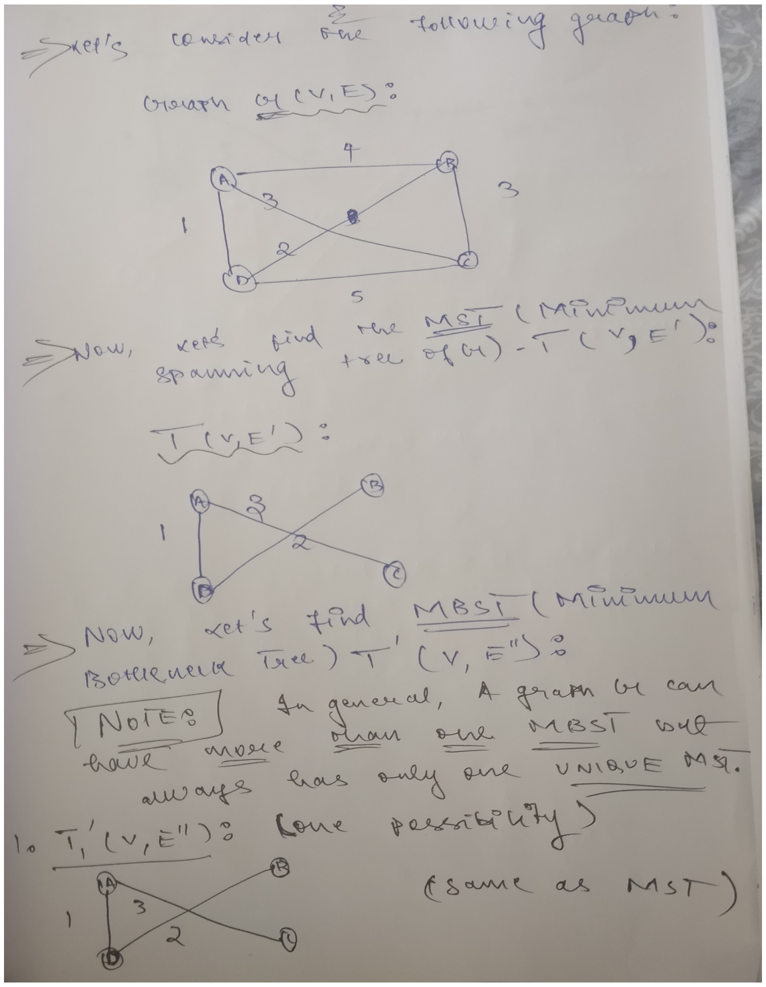
Question 3:
One of the basic motivations behind the Minimum Spanning Tree problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible.

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Specifically, let G = (V, E) be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let T = (V, E’) be a spanning tree of G; we define the bottleneck edge of T to be the edge of T with the greatest cost.
A spanning tree is a minimum-bottleneck spanning tree if there is no spanning tree T’ of G with a cheaper bottleneck edge.

(a) *Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.*

<u>Solution:</u>
No, every minimum-bottleneck tree (MBST) of G is not a minimum spanning tree(MST) of G. Let’s take an example to counter prove the statement given above:



=> Let's consider the following graph:

Graph of (V,E):

=> Now, let's find the MST (Minimum spanning tree of G) - T(V,E'):

T(V,E'):

=> Now, let's find MBST (Minimum Bottleneck Tree) T'(V,E''):

NOTE: In general, A graph G can have more than one MBST but always has only one UNIQUE MST.

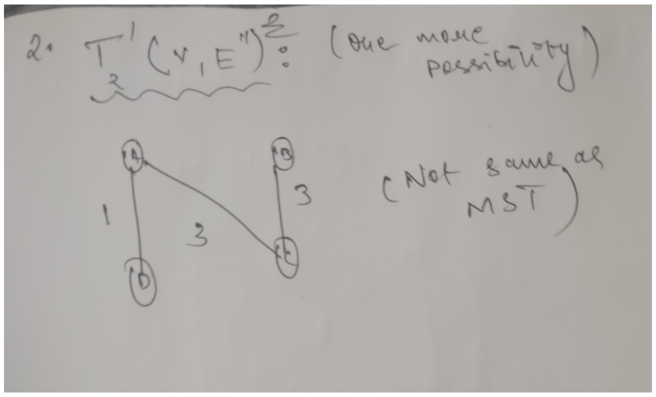
1. $T_1'(V,E'')$: (one possibility)

(same as MST)

Now, in this figure if we see I have taken an example graph and found out the MST of the example graph G(V,E).

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Then, as the next step I'm finding out the possible MBST's (Minimum bottleneck tree) for the graph G. **The first MBST found is the same as our MST is this case and the maximum weighted edge is 3.**



Here, I have found a second MBST for the same graph G. **However, we can clearly see here that this MBST is not same as MST. Though, the maximum weighted edge is 3 again.**

Hence, proved that not all MBST's are MST of a graph G.

(b) *Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.*

__Solution:__
Yes. we can use either the cut property or cycle property of MST to prove this.
Here I will be using the cut property of MST to prove the same. To prove that every MST is also a bottleneck spanning tree (MBST). Let us consider a MST T(V,E). Suppose there is some edge in T, e(u, v) that has a weight that's greater than the corresponding edge connecting u-v nodes in MBST.

Then, let V' be a subset of vertices of V that are reachable from u in T without going through node v.

Now, let V'' be a subset of vertices of V that are reachable from v in T without passing through node u.

Then, consider the cut (by the cut property) that separates subset V' and V''- as both of these are disjoint sets now. The only edge that we could add across this cut is the one of minimum weight so as to retain T to be a MST, so we know that there are no edge across this cut of weight less than w(u, v).

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

However, we have an edge between u-v with less than w(u, v) weight in MBST- according to our previous assumption. This cannot be true, as if it was then T cannot be an MST.

Therefore- it leads to a contradiction. Hence by proof by contradiction every MST is a MBST.

Question 4:
Give a linear time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each vertex exactly once.

Solution:
Given: Input: A tree T, Output: To determine if T has a perfect matching or not.

Algorithm:
Notations used-> Let V and E be the vertices, edges respectively of T and let m = |V | and n = |E|.

```
CheckPerfectMatchingorNot( T(V, E)):
1. M = ∅
2. While E not equal to ∅:
        2.1 Let v ∈ V be a leaf node in T and let (u, v) ∈ E be the unique edge incident on v
        2.2 M = M union (u, v) , add edge (u, v) to M
        2.3 V = V – u – v and remove all edges from E that are incident on vertex u (source
        vertex).
3. End of while
4. If |M| = m/2
        return "T is perfect matching"
5. Else
        return "T is not perfect matching"
```

Why this works?
Perfect matching is to find a unique edge between each pair of vertices in the given tree T.
So, the algorithm above loops until all possible edges of the tree T has been seen. The logic is that, if all possible edges of the T has been visited and still there are unvisited vertices of T left, then T is not perfect matching else it is.
So, After removing u and v from V , the resulting subgraph has a perfect matching and we may recurse on this graph. In this way we find all the edges of M' => hence |M| = |M' | = m/2. (m/2 because each edge is incident on two vertices)

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Running time analysis:
The algorithm is linear in the number of edges of T.
=> O(|E|)

Question 5:

Ternary Huffman. Trimedia Disks Inc. has developed 'ternary' hard disks. Each cell on a disk can now store values 0, 1, or 2 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size n, where the characters occur with known frequencies f1, f2, . . . , fn. *Your algorithm should encode each character with a variable-length codeword over the values 0,1,2 such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Prove that your algorithm is correct.*

Solution:

Given: Ternary disks that can store 0,1,2 values instead of binary values- 0 and 1.

To solve: Deduce an algorithm using huffman encoding that leverages ternary storage. It should follow rules of prefix-code to provide maximum possible compression.

Algorithm:

TrenaryHuffmanAlgo(char[], freq[]):

1. Create frequency table by ordering the unique characters identified in the input as per their increasing order of frequencies.
2. If n (number of unique characters/ nodes in the huffman tree) is not having 3+2k symbols for some integer value k, i.e, if is not odd-> add dummy one or two dummy nodes to the end with 0 frequencies.
3. Instead of picking 2 least possible characters, pick 3 least frequent/probable character at each step. This works when there are 3 + 2k symbols for some integer k. Alternatively, combine fewer than 3 symbols, say combine 2 symbols just in the first iteration if you do not want to add dummy symbols at the end and n is not of the form 3 + 2k, for some integer value k.

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Note: We need to ensure 3 + 2k so that at the end of huffman tree building, we exactly will have one node (root node) left in the frequency table.

4. Recursively perform step 3 to obtain the huffman tree for ternary case.
5. To now get the code of the characters from the built huffman tree, assign 0 to left child of a node, 1 to centre child of a node and 2 to children that are onto right of a node.
6. Starting from root till you reach every vertex/node in the huffman tree, traverse the tree and for every node visited store its corresponding code in an array with index as its node name/number.

Correctness/ Soundness of the algorithm:

The above algorithm is just a modification to the Binary huffman code algorithm in-order to incorporate the ternary behaviour, which ensures that the least frequent nodes end up at the leaf nodes and most frequent nodes are at the top levels of the huffman tree.

So, to prove ternary case- We just have to show that any optimal tree has the lowest three frequencies at the lowest level (If not then we could switch a leaf with a higher frequency from the lowest level with one of the lowest 3 leaves of T, and obtain a lower average length).

Without any loss of generality, we can assume that all the three lowest frequencies are the children of the same node (if they are at the same level). Now, observe that we can treat the contracted leaves (combined nodes) as a new character with frequency equal to the sum of the frequencies of the three characters. By a similar reasoning, we can see that the cost of the optimal tree is the sum of the tree with the three symbols contracted and the eliminated mini tree which had the nodes being subjected to contraction.

Hence, by induction it can be proven that by adding in the combined nodes in pairs of three in the increasing order of their frequency values we will ultimately result with the value of the node tree.