Name(last, first): [redacted]

U C L A Computer Science Department

CS 180 **Algorithms & Complexity** **ID (4 digit):** [redacted]

Midterm **Total Time: 1.5 hours** **February 2016**

Each problem has 25 points.

25

1 **A.** Describe Topological Sort (in English, bullet by bullet) on a DAG **B.** Analyze its time complexity and justify your answer. **C.** What will happen when there is a cycle? Prove your answer.

~~Iterate through all edges~~ Let's have E be the no. of edges and n be the no. of nodes.

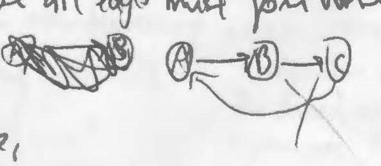
A topological sorting on a DAG is a sorting that preserves precedence relationships.

~~And~~ Any node that has a directed edge from it to another node, has to be ordered before that node in a topological sort. The algorithm is as follows.

✓ Iterate through all edges of the graph. For every edge (u,v), where the edge points from u to v, increase the out-degree of u by 1, increase the in-degree of v by 1. This takes O(E) time. Now we iterate through all nodes to find the nodes with in-degree = 0 to generate a list of sources. Let's call this set of source nodes S. This takes O(n) time.

Now output an arbitrary source from S and delete it from S. Consider all the edges that come out from it. The nodes that are connected to these edges, decrease their in-degree by 1. If any node has ~~an in~~ a resulting in-degree of zero, add it to the list of sources S. Since every time ✓ we pick a source we look at the edges coming out of it, we will look at all edges of the graph. This step takes O(E) time. Our algorithm guarentees that there will always be a source to be picked from S at any point in our algorithm as long as G is DAG. To reconstruct the topologically sorted graph is simple. Every time we output a source and look at its neighbors, we ~~set that source to be~~ make sure that ~~source is sorted before its neighbors~~ the neighbors are placed afted the source in our final data structure that contains the sorting. This takes O(E) time.

Total time = O(n+3E) = O(n+E)

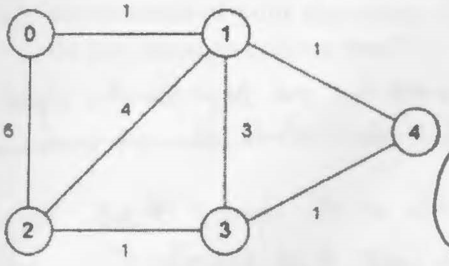
✓ If there is a cycle, there cannot be a topological sort. This is because all edges must point forward in the sorting, with a cycle there will be a backward edge. ⇒ 

Our algorithm will also not work because at some point in time, there will be no sources available to output, since in a cycle, there is no source.

25

Name(last, first): [redacted]

2 . A. Use Kruskal's MST algorithm to find a an MST in this graph. Show each step.
B. If some edges were negative would the algorithm still find an MST. Prove your answer.



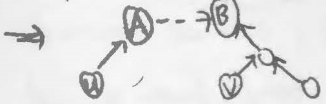
First sort edges by non-decreasing order of weights.
We get the sorting : 1,1,1,1,3,4,6.
~~Since the edges are non distinct~~

~~-Look at the first edge. Arbitrarily pick one of the edges with weight 1, let's say (0,1)~~. Pick an edge according to sorted order. Arbitrarily pick any non-unique edge.

Make-union-set operation: make n disjoint sets where n is the no. of nodes. Each node belongs to each set initially. This takes O(n) time.

-Suppose let's call the n disjoint sets by the names of their nodes. So we have the sets {0,1,2,3,4}.

~~Look at (0,1), it connects nodes the edges~~ For every edge (u,v), look at the sets u & v belong to. Do a find(u) and find(v) operation to find if u and v belong to the same set. ~~Assuming~~ This takes O(lgn) time, which will be explained below. If u and v belong to the same set, ignore edge (u,v), because this edge forms a cycle. If u and v are from different sets, add (u,v) to the MST. Do a make-union of the sets that u and v belong to. Do this by creating a pointer that points from the root node of ~~its set to the root node of its set~~ set which is smaller to the root node of the ~~set~~ that is bigger. This maintains a tree every time that looks like this This takes O(1) time



~~This~~ In this case, root node A represents the set that contains u while root node B represents the set that contains v. We create a pointer from A to B, increasing the height of the tree by at most one, while the no. of nodes at most doubles. This allows the find(u) operation to be O(lgn) time, since it is O(height of tree).

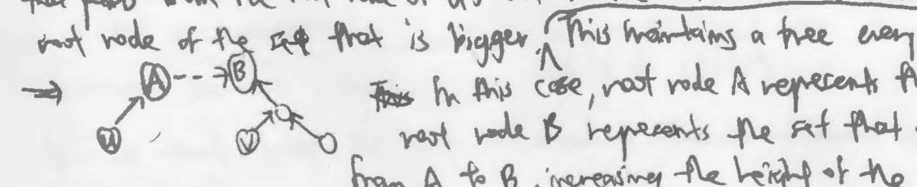
~~Eventually we all remaining edges will connect to nodes in the same set, ~~so we in~~ and the output is the MST.~~

Since we look at all edges and do a find and make union operation each time (at worst), we have time = O(elgn). sorting the edges takes O(elge) = O(elgn). So final time taken = O(elgn)

even if e=n²
O(elgn²) = O(e2lgn) = O(elgn)

with negative edges, Kruskal's algorithm still sorts edges in non-decreasing order. Since Kruskal's algorithm only cares about the sorted order of edges, we can use the same algorithm. Picking edges in this sorted order and doing the same operations will still result in an MST Since the algorithm is guaranteed to produce an MST, an MST will be produced.

In this case the output sequence is as follows:



Final MST

Name(last, first): [redacted]

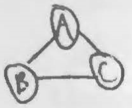
23

3. A graph is two-colorable if we can color its vertices with RED & BLUE such that no two adjacent vertices have the same color. **A.** Are all graphs two-colorable. Prove your answer **B.** Design an efficient algorithm for two coloring a graph. Prove the correctness of your algorithm. **C.** Analyze its time complexity. **D.** How many colors do you need to color a tree?

We prove this by showing that

No. Graphs with odd cycles are not two-colorable. ~~Suppose we have~~ a graph with an odd cycle cannot ~~be~~ be two-colorable

~~We move along the nodes of the cycle, coloring it in an alternate fashion. This means~~ ~~Take the below~~



This ~~is~~ is a graph with an odd cycle. We color A red, so we have to color B blue since it is adjacent to A. Now we color C red since it is adjacent to B. Now A and C have the same color but they are adjacent ⟹ no two-coloring possible.

Do a BFS on the graph. starting from an arbitrary node s. Maintain a visited array to keep track of visited nodes. Maintain lists too that keep track of levels. Maintain also a parent array to keep track of parents

Initially start at s. Add s to L[0]. Set a level counter i=0.

~~Look at~~ ~~Consider every edge (u,v) that connects from~~ s.

~~While~~ For every node u in L[i], consider its edges (u,v).

```
If v is not parent(u) and not visited
    Add v to visited list.
    Add v to ~~to~~ L[i+1].
    Set u = parent(v).  → If v is not parent(u) and visit
                           add v to L[i+1]
Increment counter i by 1.
```

Now we iterate through the levels and attach a ~~color to different color~~ red to odd levels and blue to even levels.

Now we iterate through the nodes in all levels. If any node belongs to 2 levels that have different colors, there is no two-coloring of the graph. If not, this is a valid coloring of the graph.

This algorithm is correct because any node one level apart is adjacent to each other, and hence must be colored differently, which is what our algorithm does. ✓ proof not formal -2

The BFS takes $O(e+n)$ time, where e = no. of edges, n = no. of nodes. Iterating through levels to determine if it is a valid coloring takes $O(n)$ time.

Thus total time = $O(e+n+n) = O(e+n)$. ✓

We need 2 colors to color a tree since it has no cycles, hence it must have a valid 2-coloring.

Name(last, first): [redacted]

0

25

4. Consider a sequence x1, x2, ... , xn of (positive and negative) integers. We want to find two indices i and j such that xi + ... + xj is maximized.

A. Describe in English (bullet by bullet) an O(n) time algorithm (using constant extra space) for solving this problem. For example, if the input is (-2, -2, 5, 7, -3, 4, -4) then i=3 and j = 6 (and the sum xi+…+xj is equal to 13). **B**. Prove the correctness of your algorithm.

-2 -4 1 8 5 9 5

-2 -2 5 7 -3 4 -4

↑ ↑ ↑ ↑

-2 -2 5 7 -3 -3 -9 4 -1

Iterate through the list.

Set an index candidate. Set 0

~~Whenever~~ The first increase we see, add that index to the index candidate.

Every time we ~~decrease~~ see a decrease, we decrease the index ~~input~~.

Maintain a sum. If the sum is the greatest from index i to j.