TheoryPrep **(Due: 7/26/20)**

Assignment #8: Algorithms

Name: Student name(s)

We've taken some problems/solutions from the course texts and CS 121/124/125 material. Some problems are marked as (Challenge) or (Additional Problems). Do the rest first, and if you have time, return to these problems. It's **much more important** to have a rigorous understanding of the core problems and how to prove them than to simply finish all the additional problems.

Problem 1: Algorithms

Learning goal: This problem set is different from previous ones. The goal for this problem set is to help you prepare for 124, and one of the main difficulties in 124 is that the problems are longer. This means you'll need to use your longer proof writing skills (lemmas, breaking down proofs into smaller parts, applying multiple proof techniques in one proof, et cetera).

Thus, Pset 8 has only one problem (taken from 124–but don't let this scare you since you did a couple of 124 problems last week!). 124 problems often follow the structure below

1. Learn about an algorithm in class. See its proof of runtime and correctness (a proof of correctness means it correctly gives the answer to a problem, like yielding a topological sort).
2. You are given a problem that you can solve with the algorithm from class (or a slightly modified version).
3. Give the pseudocode for the algorithm (you can abstract away the function that you learned in class–so on this problem set you could just "get the Topological Sort" or "return the SCC graph").
4. Give some runtime bound $f(n)$ and prove that your code runs in $O(f(n))$.
5. Prove that it correctly solves the problem.

You can follow the above outline for this problem set

(a) In one of your math classes, the professor wanted to show that the following 4 statements are equivalent:

(1) $A \subseteq B$

(2) $A \cap B = A$

(3) $A \cup B = B$

(4) $\bar{B} \subseteq \bar{A}$ (where $\bar{A}$ denotes the complement of A)

Since time was running out in lecture, the professor decided to only show the proofs that (1)⇒(2), and that (3)⇔(4), and she asked you to think about how to finish the equivalence proof when you return home. Seeing as how proving implications is time-consuming, you wanted to prove as few as possible to show equivalence of all statements, and you realized that you could do so by further showing just two more statements: namely (2)⇒(3), and (3)⇒(1). Since you're also taking CS 124, you decided to challenge yourself by trying to develop an algorithm for the following generalized variant:

Given n statements that you want to show are logically equivalent, and given proofs of implications of some subset S of the $n(n-1)$ possible implications, output a minimum number of extra implications one needs to prove that, when combined with those already proven in S, implies equivalence of all n statements. You should give an algorithm (pseudocode/python level is fine, but you should explain how to use the data structures you choose to do the operations you want your program to do), give its runtime (you can do so in terms big-oh of n and $|S|$), and prove that it returns the correct result. Again, the begin{lstlisting} environment may be helpful for writing code.

Solution:
Solution Idea: Since this is a long proof, students will not immediately know how to do it. This first paragraph details a way you might guide students through the exploration phase of the problem (I mean the part where they are trying things out and before they have a concrete solution, when they may feel lost). I think an important lesson for students to learn is that especially with these longer problems, it's okay to not be making tangible progress towards a solution; if you've spent the last half hour trying solutions that don't work, that's huge progress towards understanding the structure of the problem and its eventual solution. It's helpful to first draw out some examples, and students will probably realize that an SCC can really be treated as one vertex since all of its members are equivalent to each other. This means we are thinking about a DAG, so students may want to draw some of those and think about how to turn a DAG into an SCC (because every vertex being connected to each other is exactly what it means for everything to be logically equivalent). Sometimes students can get hung up on finding "the solution" and therefore might be unwilling to write anything (perfection is the enemy of the good). Some helpful prompts may be, "what if we don't restrict ourselves to finding the best solution? Then what's A way to turn this DAG into an SCC? What patterns do you see there? How can we turn that into a solution?" or "Why is this DAG not an SCC? What makes it that way? How can we fix that by adding edges? Is that the fewest number of edges? What patterns do you see that could indicate an algorithm?" This may help students see the idea of taking the maximum of the number sources and sinks, since the sources and sinks are "the reason" the DAG is not an SCC.

The approach is to first create an SCC graph and then we can search the SCC graph for the number of sinks and sources and return the maximum of the number of sinks and sources. The pseudocode is below (as noted on Piazza, since we went over the algorithm to convert to an SCC graph, we can abstract it out here–we assume adjacency list representation through nested list)–we assume that the input is a list of lists, each sublist containing the statements implied by that index which also happens to be an adjacency list representation for a graph where the nodes are the mathematical statements and edges are implications.

```
1  def finishProof(G):
2      SCC_g = SCC(G)
3      if len(SCC_g) == 1:
4          return 0
5      source = [1] * len(SCC_g)
6      sink = [1] * len(SCC_g)
7      for node in the SCC_g:
8          if len(node) != 0:
9              sink[node] = 0
10         for edge in node:
11             sink[edge] = 0
12     return max(sum(source), sum(sink))
```

First runtime analysis. We know that the turning a graph into an SCC graph takes $O(|V| + |E|)$. Let $n = |V|, m = |E|$. We have that line 2 is $O(n + m)$. Lines 3 and 4 take constant time. In particular, our SCC_g has at most n nodes, so lines 5 and 6 take $O(n)$, and the code in lines 7-11 is run n times. Lines 8 and 9 take constant time, and the for loop in line 10 is run at most m total times in all the runs of 7-11 (since it's run once for each edge in each node which is to say once for each edge). We note that line 11 is constant time, so we have that lines 7-11 are $O(n + m)$. Line 12 takes at most $O(n)$, so overall we are at $O(n + m)$.

Now to prove correctness. We see that if the graph is strongly connected already, line 4 will return 0, so we exclude that case from the rest of our analysis.

We will induct on the maximum of the number of sources in the SCC graph and prove the statement that given a DAG with n sources and k sinks, then we need to add exactly $\max(n, k)$ edges to make all components connected. First we will show that we need at least $\max(n, k)$ edges. Suppose that we add fewer than $\max(n, k)$ edges. If

$\max(n, k) = n$, then there exists a source SCC in the resultant graph, and by definition, if we select a point in another SCC, we cannot get from there to the source SCC. If $\max(n, k) = k$, then there exists a sink SCC in the resultant graph, and by definition if we select a point in another SCC, we cannot get from the sink SCC to this new point. Thus, we see that we need at least $\max(n, k)$ edges.

Now we will show that adding $\max(n, k)$ edges is enough to form a strongly connected graph.

First to prove a lemma that in a DAG, every node is reachable from at least one source and there exists a sink that you can reach from that node.

Pf: Fix a DAG, and fix a nonsink, nonsource node n_1 (we will prove first for nonsink, nonsource and then generalized). We see the graph has some number of vertices m. First to prove that there exists a source that it is reachable from. Since n_1 is a nonsource, there exists some n_2 such that (n_2, n_1) is an edge. We see either n_2 is a source and we are done, or there exists some edge (n_3, n_2) ($n_3 \neq n_1$ since then there would be a cycle). We can repeat this argument to form a chain of connected nodes $n_m, ..., n_1$. We see n_m must be a source node since if it has an incoming edge, since there are only m vertices, its incoming edge originates from another member of the list, forming a cycle. Thus, we see that there exists a source node which n_1 is reachable from. To prove that there exists a sink you can reach from n_1 is very similar. Since n_1 is a nonsink, it has an out edge (n_1, n_2). n_2 is either a sink and we are done, or we can find and edge (n_2, n_3) (where $n_3 \neq n_1$ since that would form a cycle). This gives us connected nodes $n_1, ..., n_m$ and n_m must be a sink since otherwise the edge coming out of n_m would connect to a previous node and form a cycle. We can generalize to sources since any source is reachable from itself, and since it is a source, it has an edge to a nonsource, which is either a sink or is a nonsink, nonsource, so leads to a sink by our above result. We can generalize to sinks since any sink can reach itself, and since it is a sink, it is reachable by a nonsink which is either a source or is a nonsink, nonsource, so that is reachable from a source by our above result.

The lemma implies that there does not exist a DAG with no sources since every node in a DAG has an ancestor that is a source.

Thus, the base case is that $n = 1$. Suppose that our SCC graph has 1 source node and k sink nodes. We can add k edges (the maximum of the sinks and sources), each from the source node to one of the sink nodes to form graph G'. I claim that G' is a singular strongly connected component. In particular, fix arbitrary nodes A, B. We see by the lemma that B can be reached by the source node s, and the lemma also tells us there is a path from A to a sink l. Since we added edges from all the sinks to the source, we can get from l to s, so we have a path

$$A \to l \to s \to B$$

Thus, the graph is a single SCC.

Now for the inductive step. I will prove that for an SCC graph of n sources and k sinks, we need to add exactly $\max(n, k)$ edges to make all components connected. We will split into two cases

- Every source can reach every sink: Label the sources $a_1, ..., a_n$ and the edges $b_1, ..., b_k$. Connect b_i to a_i until you run out of one or the other–connect leftover bs to a_1 or if there are leftover as, add edges from b_1 to them. This is a total of $\max(n, k)$ edges. Fix arbitrary, u, v in the graph. By the lemma, u can reach a sink s. Since we connected all the sinks to the sources, we can get from s to some source a_i. By the lemma, v is reachable from some source a_j. We see that by construction, we connected either b_1 or b_j to a_j. Since all sources connect to all sinks, a_i connects to both b_1/b_j, so we have a path

  $$u \to s \to a_i \to b_1/b_j \to a_j \to v$$

  Thus, we see the graph is strongly connected.

- There exists a sink not reachable from a source: In this case, there exists a source w from which we cannot get to a sink s. Thus, we can add an edge (s, w), which is guaranteed not to create a cycle since there is no way to get from w to s. In particular, this edge also makes w not a source and s not a sink, so we now have $n-1$ sources and $k-1$ sinks, so by the inductive hypothesis, since we have a DAG with $n-1$ sources, this can be transformed into a strongly connected graph in $\max(n-1, k-1)$ edges, so we can transform the original graph into a strongly connected graph with the addition of $\max(n-1, k-1)+1 = \max(n, k)$ edges.

This completes the proof.

(b) (Additional Question) Design an efficient algorithm to find the *longest* path in a directed acyclic graph given in adjacency list representation. (Partial credit will be given for a solution where each edge has weight 1; full credit for solutions that handle general real-valued weights on the edges, including *negative* values; you can assume the adjacency list representation is a list of lists of tuples (u, i) where u is the destination of the edge an i is the weight of the edge). You should again give an algorithm (pseudocode/python level is fine but it should be granular enough to use the adjacency list representation), give its runtime (in terms of the number of vertices and edges in the graph), and prove that it returns the correct result.

Solution:
Solution Idea: Since this is a long proof, students will not immediately know how to do it. This first paragraph is details a way you might guide students through the exploration phase of the problem (I mean the part where they are trying things out and before they have a concrete solution, when they may feel lost). I think an important lesson for students to learn is that especially with these longer problems, it's okay to not be making tangible progress towards a solution; if you've spent the last half hour trying solutions that don't work, that's huge progress towards understanding the structure of the problem and its eventual solution. Again, a great starting point is for students to try example DAGs (with different weights, but maybe starting with nonnegative weights). Especially with more complex DAGs, they may label the maximum length paths from each node in reverse topological order, which gives rise to the algorithm! Another way to think about this problem is that a DAG should always make you think of topological sort. Questions for students along the lines of "how could we use the structure of a topological sort to identify the longest path in the graph" or "if we write out the topological sort as $v_1, ..., v_n$, how could we identify the longest path starting at a vertex v_i? What does that depend on?" are great. This may help students see that the longest path from a vertex just depends on the lengths of paths from vertices later in the topological sort and the weights of outgoing edges. This suggests an algorithm of calculating maximum distances locally in reverse topologically sorted order.

First to give the algorithm. Let n the number of vertices and m the number of edges. First do a topological sort of the graph. Let the points resulting from the topological sort be labeled $a_1, ..., a_n$. Now initialize an array $\mathit{start}[1, ..., n] = 0$ and a variable $l = 0$. Starting with a_n and descending, for all edges (a_i, v), set $\mathit{start}[a_i] = \max(\mathit{start}[a_i], \mathit{start}[v] + w(a_i, v))$ (we see this makes sense since by topologically sorting, we ensured we already set $\mathit{start}[v]$). Then before moving on to the next a_i, set $l = \max(l, \mathit{start}[a_i])$. At the end of this process, l will be the longest path in the DAG.

First to analyze runtime. We see a topological sort is $O(n+m)$, and initializing variables is $O(n)$. Then we do various arithmetic and variable setting in a for loop that loops over all vertices, and then all outgoing edges from that vertex. Thus, we have $\sum_{i=1}^{n}$ outgoing edges from $i + 1 = O(m+n)$. Thus, the overall runtime is $O(n+m)$.

We will prove correctness by induction on the number of itterations of the a_i for loop (let $\phi(a)$ denote the true longest path starting from a). Let k be the number of iterations through the for loop. I will prove that for all $i \leq k$, $\mathit{start}[a_i] = \phi(a_i)$, and m is the maximal value of $\{\mathit{start}[a_i] | i \leq k\}$. The base case is $k = 0$, in which case the statement is true vacuously since there are no a_i such that $i \leq 0$.

Now for the inductive step. Assume our inductive hypothesis is true for some k. First I must show that $\mathit{start}[a_{k+1}] = \phi(a_{k+1})$. We will first show that $\mathit{start}[a_{k+1}] \geq \phi(a_{k+1})$.

Two cases

1. Let the longest possible path starting at a_{k+1} be $a_{k+1}, b_1, ..., b_r$. This implies that the longest path starting from b_1 is $b_1, ..., b_r$, since if there was a longer path p starting from b_1 we could obtain an even longer path starting from a_{k+1} by substituting in p. Furthermore, we see that since we topologically sorted the vertices $b_1 = a_j$ for some $j \leq k$ (otherwise the edge a_{k+1}, b_1 would violate the top sort). Thus, $start[a_j]$ is equal to the length of the path $b_1, ..., b_r$, so when we iterate through the edges of the form (a_{k+1}, v), we will eventually hit (a_{k+1}, a_j), and this ensures that $start[a_{k+1}] \geq \phi(a_{k+1})$.

2. The other case is that the "longest" path is just the point a_{k+1}. We see that $start[a_{k+1}] \geq 0$ since it is initialized to 0, and the line $start[a_{k+1}] = \max(start[a_{k+1}], start[v] + w(a_{k+1}, v))$ can only increase its value.

We should also check that $start[a_{k+1}] \not> \phi(a_{k+1})$. We see that $start[a_{k+1}]$ is only set on the line $start[a_{k+1}] = \max(start[a_{k+1}], start[v] + w(a_{k+1}, v))$. Suppose, towards a contradiction that this line sets $start[a_{k+1}] = c > \phi(a_{k+1})$. By the topological sort $v = a_j$ for $j < k$, so $start[v]$ is the longest possible path starting from v, so in particular it is the length of a path starting from v, so this implies there is a path of length $c > \phi(a_{k+1})$ starting at a_{k+1}, which is not possible. Thus, we see that $start[a_{k+1}] = \phi(a_{k+1})$

Now to show that l is the maximal value of $\{start[a_i] | i \leq k+1\}$. There are two possibilities

1. $\max(\{start[a_i] | i \leq k+1\}) = \max(\{start[a_i] | i \leq k\})$. In this case, none of the updates to l change its value, since l is always updated as the maximum of itself and some new value, and by the inductive hypothesis it already had the value $\max(\{start[a_i] | i \leq k\})$.

2. $\max(\{start[a_i] | i \leq k+1\}) > \max\{start[a_i] | i \leq k\})$. This implies that $\max(\{start[a_i] | i \leq k+1\}) = start[a_{k+1}]$, so after we finish going over the edges (a_{k+1}, v), we get to the line $l = \max(m, start[a_{k+1}])$, at which point l is set to $\max(\{start[a_i] | i \leq k+1\})$.

This completes the inductive step. To see why this inductive proof implies the correctness of the algorithm, we see that after running the algorithm n times, $l = \max(\{start[a_i] | i \leq n\})$, and $start[a_i] = \phi(a_i)$, so $l = \max(\{\phi(a_i) | i \leq n\})$, or the maximum of the lengths of the longest paths starting at any vertex in the graph, which is exactly the length of the longest path in the graph.

Problem 2: Review

Learning goal: The goal for this section is to reflect on how you can improve your proof skills!

(a) Select the problem from a previous week that you think you would learn the most from redoing. Copy the problem and your instructor's feedback below, rewrite the proof, and add a short reflection on the changes you made.

(b) (Additional Question) Choose another problem you received feedback on and copy the problem and feedback you received, rewrite the proof, and write a sentence about why you made your changes (so do Part A again) OR solve a problem you left blank from a previous week (try to pick from a week you felt least secure on).

Problem 3: Feedback (Mini Q)

Purpose: Your instructor will have a survey link for you to fill out. This will be really helpful to us if we run this course (or other versions) in the future, so please fill it out authentically!