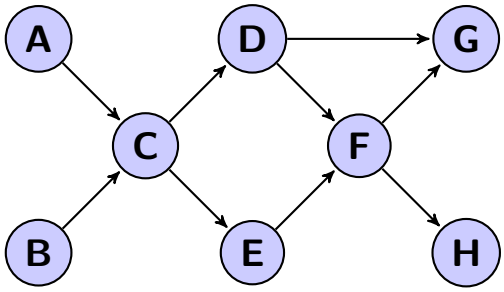
This homework is due by the start of class on Friday, February 19th. You must submit the homework via the course page on T-Square, and you may hand in a printed version with any graph drawings, if you choose not to include them in the online copy.

Homework Policies:

- Your work will be graded on correctness and clarity. Write complete and precise answers.
- You may collaborate with up to *three* other classmates on this problem set. However, you must *write your own solutions* in your own words and *list your collaborators*.
- You may portions of our two textbooks that we have covered. The purpose of these homeworks is NOT only to teach you the algorithms that we cover, but also to teach you how to problem-solve. Therefore using *ANY* outside resources like the internet, previous students etc. is *not* allowed!
- **Optional** questions are for you to practice and learn the basics before you advance. They may ask you to follow the steps of an algorithm covered in class, or review material from an earlier class. Solutions to these problems will not be graded, but you must be are able to do them, and you are responsible for material that is addressed by these optional questions.
- **Basic** questions *must be solved and written up alone*; you may not collaborate with others. These will help you build the framework needed to solve the harder questions. Please do these first, before meeting in groups, for both you and your group's benefit.
- **Group** questions may be solved in collaboration with up to ***three*** other classmates. However, you must still *write up your own solutions* alone and in your own words. These will form the bulk of your homework questions.
- **Bonus** questions are like group questions, but they will be more challenging and will be graded more rigorously. Bonus points are tallied separately from your normal points, and will be applied to your grade *after* the curve. Bonus points are worth more than normal points, and can have a significant positive affect on your grade. Most importantly, bonus problems are fun!

1. (optional) Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



   (a) (optional) Indicate the pre and post numbers of the nodes.

> **Solution:** In order of pre number, (the order of visitation), we have:
>
> - A: 1, 14

- C: 2, 13
- D: 3, 10
- F: 4, 9
- G: 5, 6
- H: 7, 8
- E: 11, 12
- B: 15, 16

(b) (optional) What are the sources and sinks of the graph?

Solution: The sources are A, B, the sinks are G, H.

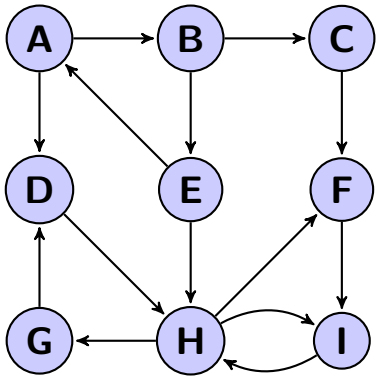
(c) (optional) What topological ordering is found by the algorithm?

Solution: In order of decreasing post number, B, A, C, E, D, F, H, G.

(d) (optional) How many topological orderings does this graph have? List them all.

Solution: There are 8 orderings - we can choose which of A, B, D, E, and G, H to place in front of the other, but everything else is fixed.

2. (20 points) (basic) Run the strongly connected components algorithm on the following directed graph G. When doing DFS on G^R : whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.



(a) In what order are the strongly connected components (SCCs) found?

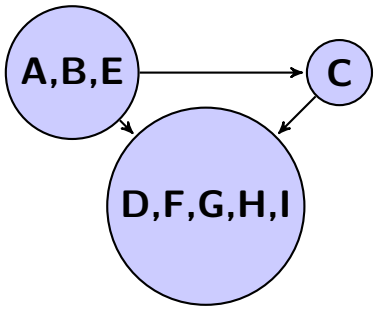
Solution: We "find" SCCs when we finish exploring the reachable nodes at each location, in order of decreasing post number in the reverse graph. If you do this, you will first complete the component D, F, G, H, I, then C, and finally A, B, E.

(b) Which are source SCCs and which are sink SCCs?

Solution: A, B, E is the source SCC, D, F, G, H, I is the sink.

(c) Draw the "metagraph" (each meta-node is a SCC of G).

Solution:



(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

Solution: I could add one edge from D to A say, and thus make a cycle in the meta graph. This would collapse that cycle into a single strongly connected component.

3. (20 points) (basic) The police department in the city of Computopia has made all streets one-way. The mayor claims that there is still a way to drive legally from any intersection in the city to any other intersection. However, the city elections are coming up soon, and we need to be able to decide this as fast as possible.

(a) Formulate this problem graph-theoretically and give an algorithm to solve it. Correct and faster asymptotically is worth more points.

Solution: Treat each intersection as a vertex, and each street as an directed edge between vertices. By asking if we can move from any intersection to any other intersection, we are asking if this graph is strongly connected.

Tarjan's SCC algorithm is linear time, and we can tell that the mayor is right if there is a single SCC, and wrong if there is more than one.

(b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and show how it too can decided as fast as possible.

Solution: In our graph representation, the mayor is claiming that everything reachble from the town hall also has a path back to it. Consider the SCC meta graph of our graph. If the node containing the town hall had any outgoing edges, then the endpoint of that node would NOT have any path back to the town hall (as the meta graph is acyclic). If that node had NO outgoing edges, then the only set reachable from the town hall are vertices in its own SCC, which by definition always have a path back to the town hall. Thus, the property we are looking for is equivalent to the SCC

containing the town hall being a sink node. We can also read this from the SCC meta graph returned by Tarjan's algorithm, which again runs in linear time.

4. (20 points) (group) Give an efficient algorithm that takes as input a directed graph G and determines whether there is a vertex from which all other vertices are reachable. Prove your algorithm is correct and compute its running time. Faster (in O notation) and correct is worth more credit. (Note: The input graph G may not be acyclic.)

Solution: We'll start with the algorithm, and then we'll explain why it's correct.

Run Tarjan's SCC algorithm in linear time. We note that the question is equivalent to asking "Is there an SCC from which all other SCC's are reachable," since every vertex belongs to some SCC, and all vertices inside an SCC can reach all other vertices in that SCC. Check each component in the meta graph MG to see how many sources there are (no incoming edges) in linear time in the size of the meta graph, which is at most as large as the original graph. Return TRUE if there is only one source in MG, and FALSE if there is more than one.

It's not obvious why there is only one source if and only an SCC (that source) reaches everything, so we give a short proof of correctness. First, consider the case that there are at least two source SCCs; then one source can't reach a different source, as there are no incoming edges to a source. Thus the property doesn't hold.

Now consider the case where there is only one source SCC in MG, call it X. Consider deleting all the SCC's reachable from X from the meta graph to obtain new meta graph MG'. The elements we deleted remaining had no incoming edges to anything in MG', since otherwise those elements would be reachable from X and would have been deleted. Thus any source in MG' was also a source in MG. We claimed that MG had only one source, and it was deleted in MG'. Since every non-empty DAG has some source vertex, this means that MG' must be empty, which means that everything in MG was reachable from the source.

An alternate algorithm could do two passes of a DFS, one to obtain all the post numbers, and then a second DFS from the element with highest post number, which is a "necessary candidate" for an element that reaches everything. You would need to prove that if some element reached every vertex, then so does the element with highest post number to ensure correctness - this ensures that the element with highest post number is all that you need to check.

5. (40 points) (group) To get in shape, you have decided to start biking to work. Every street in your city is one-way, and is either uphill or downhill. You want a route that goes uphill and then entirely downhill, so that you can work up a sweat going uphill and then get a nice breeze at the end of your run as you ride downhill.

   Your run will start at home and end at work and you are given as input a map detailing the roads with n intersections and m one-way road segments, each road segment is marked uphill or downhill. You can assume home is intersection 1 and work is intersection n, and the input is given by two sets: E_U representing the uphill edges and E_D representing the downhill edges, where each set is given in adjacency list representation.

Give an algorithm that determines if there is a route (regardless of the length), which starts at home, only goes uphill, and then only goes downhill, and finishes at work. You want the uphill and downhill components to include at least one edge each.

You should give a clear description of your algorithm in words, and you should explain its running time. Faster (in O notation) and correct is worth more credit.

Here is an example with 5 vertices, where vertex 1 is home and vertex 5 is work. The input are the following arrays of linked lists:

$E_U = 1 \to 2; 3 \to 4, 3 \to 5.$
$E_D = 1 \to 3; 2 \to 3, 2 \to 4; 4 \to 5.$
There is one valid path to go from 1 to 5: $1 \to 2 \to 4 \to 5$.

Solution: We will present two algorithms for this problem. The first is easier to think of, but the second is somewhat more clever, and the ideas in the second algorithm will be useful for you in the future.

First run a DFS starting at home using only uphill edges, and mark all vertices reached. These are the vertices we can reach using only uphill edges. Then, run a DFS on the reverse graph starting at work - these are the vertices we can descend down to get to work. If we ever come across a marked vertex, return "True", else return "False". This requires two runs of DFS/reachability, which takes linear time.

Alternatively, we construct a NEW graph G' as follows. For each vertex $v \in V(G)$, make two copies, v_{up}, v_{down}. For every uphill edge in G, connect the corresponding vertices labeled *up*, and for every down hill edge, connect the corresponding vertices labeled *down*. Finally, put a directed arrow from v_{up} to $vdown$ to represent a transition from going uphill to downhill. Now, we run normal DFS to check if $home_{up}$ can reach $work_{down}$. Constructing the graph copy takes $O(V)$ work to copy the vertices, and $O(E)$ work to transplace edges in G to G'. Running a DFS on the new graph takes $O((2V) + (V + E)) = O(V + E)$ time.

IF we want to be rigorous, we need to show that G' has a path from $home_{up}$ to $work_{down}$ IF AND ONLY IF G has a valid uphill/downhill path from home to work. That way, we know that if there is a valid path in G, we will find it AND we know that we won't find paths that aren't actually valid in G. In this case, this statement is fairly clear from the construction of G', and a proof could be omitted, but in future problems of this type, this step will be necessary and less obvious.

Say G' has a path from $home_{up}$ to $work_{down}$. Such a path would have to "transition" fromm uphill to downhill vertices, since the only edges connecting uphill to downhill edges are transition edges by construction. Before this transition, this path only travelled between "up" vertices, which means by construction, the corresponding edges in G were all uphill. Similarly after the transition, all the corresponding edges in G were downhill. Thus all the non-transition edges correspond to a a valid uphill/downhill path in G.

Now, say there is a uphill/downhill path in G, then the corresponding uphill path in G' exists among the up vertices, the transition vertex exists, and the corresponding downhill path in G' exists among the downhill vertices, connecting $home_{up}$ and $work_{down}$.

6. (5 points) (bonus) We are given a set of boolean variables $\{p_1, \ldots, p_n\}$, as well as a list L of logi-

cal statements of the form $x = true$, $x = false$, or $x \implies y$, where $x, y \in \{p_1, \ldots, p_n, \bar{p_1}, \ldots, \bar{p_n}\}$. Describe an efficient algorithm than can determine if L represents a consistent set of implications, (i.e. there are no logical contradictions.)

Pseudocode is optional, but a clear and precise description of the algorithm should be given, along with proof of correctness and analysis of running time. Faster (in O notation) and correct is worth more credit.

Hint: Start by creating a graph with $2n$ vertices, one for each positive and negative literal. Add edges to represent the logical statements, and formulate logical consistency as a property about this graph.

Solution: First, we deal with the implications. For each proposition p_n, create two nodes representing that node being true or false - we call them p_n and $\bar{p_n}$. For convention, we say $\bar{\bar{x}} = x$

For each implication $x \implies y$, we construct *two* edges, from x to y, and for $\bar{y} \to \bar{x}$, as both will be needed to make inferences. For convenience, we will (for now), translate statements like $x = true$ to mean $\bar{x} \implies x$, which is logically equivalent.

Now, run Tarjan's SCC algorithm on this graph to obtain metagraph MG of the SCC's. An SCC in this graph represents a set of vertices that must have the same truth value (everything implies each other). An implication in the metagraph means that all the truth values in one SCC imply all the truth values in the other (in one direction only).

Once we have the metagraph, consider the following algorithm for assigning these SCC's to true or false.

Here is a hypothetical algorithm to create a potential assignment. Take a source SCC, and set all the truth values in it to false. By symmetry, the complements of these variables are in a sink SCC that will be set to true. Since False can legally imply anything, and anything can imply True, we cannot create any logical inconsistensies by doing this assignment. We can now remove these two nodes from the graph, and repeat. Since an SCC metagraph is acyclic, there will always be a source/sink to use at every iteration.

The only way this wouldn't work is if a vertex and its complement were both in the same SCC. We can check for this in linear time after we've run Tarjan's SCC algorithm. Thus the whole process of determining consistency can be done in linear time.