Georgia Institute of Technology
College of Computing
March 4, 2016 Bhakta

CS 3510 **Name:** ______________________________

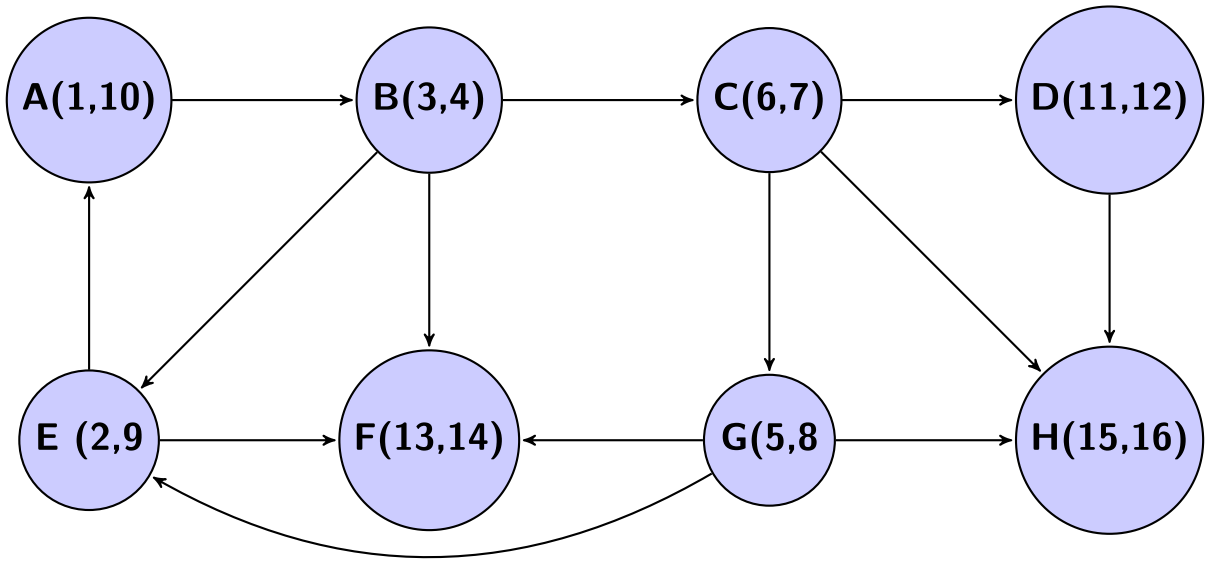
Test 2 **Id:** ______________________________

Ask questions if you are not sure what a problem is asking.

GOOD LUCK!!

Problem	Points	Score
1	20	
2	20	
3	20	
4	40	
Bonus Points	10	
Total	100	

1. Run the strongly connected components algorithm given in class on the following directed graph G. When doing DFS, whenever there is a choice of vertices to explore, always pick the one that is **alphabetically first**.



(a) Indicate the pre and post numbers for each vertex from the first run of DFS on G^R. (You are beginning at A). You may just mark the pre and post numbers on the above graph.

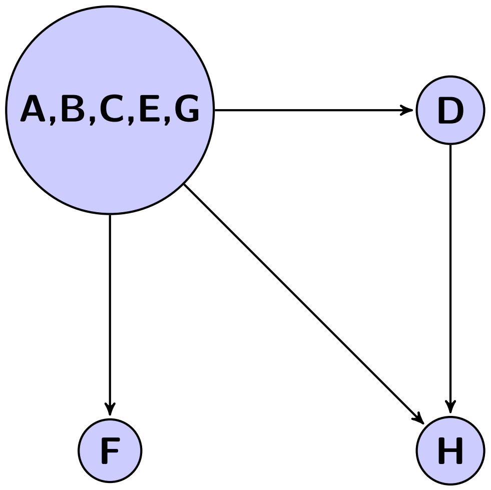
Solution: See above

(b) List the strongly connected components in the order they are discovered, and list each vertex in the SCC in the order **it** is discovered. As before, when you have a choice of vertices to explore, always choose the one which is alphabetically first.

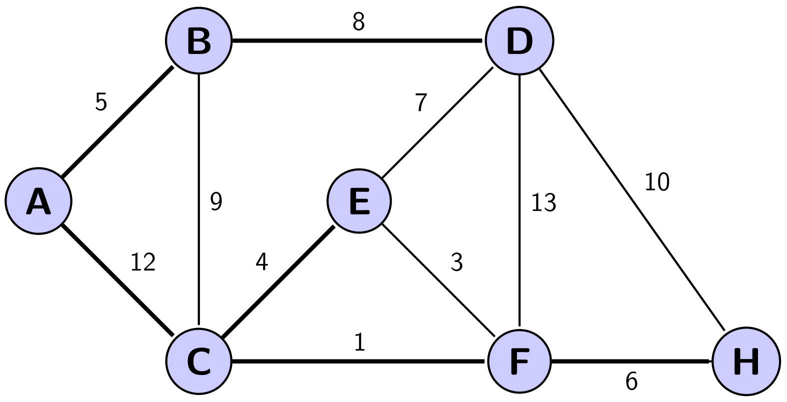
Solution: First we find (H). Then (F), then (D), then (A,B,C,G,E)

(c) Draw the final SCC meta-graph.

Solution:



2. Run Dijkstra's algorithm on the following graph, with source node A. For each addition of one vertex, clearly indicate which vertex you've added, and show the elements of the priority queue with their current dist values (No need to show them in a heap). Finally, mark the edges of the shortest path tree on the below graph.



Solution: Below are the values in the priority queue at each step, with the chosen vertex at each step circled. The shortest path tree is drawn above.

A	B	C	D	E	F	H
0	∞	∞	∞	∞	∞	∞
[✓ 0]	**5**	**12**	∞	∞	∞	∞
0	[✓ 5]	12	**13**	∞	∞	∞
0	5	[✓ 12]	13	**16**	**13**	∞
0	5	12	[✓ 13]	16	13	**23**
0	5	12	13	16	[✓ 13]	**19**
0	5	12	13	[✓ 16]	13	19
0	5	12	13	16	13	[✓ 19]

3. A *bottleneck* edge in a flow network is one that, if you increased its capacity, the graph would have a larger max flow.

   Conversely, a *critical* edge in a flow network is one that, if you reduced its capacity, the graph would have a smaller max flow.

   Prove or disprove each of the below statements, and state clearly whether you are proving them true or false:

   (a) Every flow graph must have at least one bottleneck edge.

   Solution: False: A graph that is a simple path with two edges of capacity 1 has no bottleneck edges.

   (b) Every flow graph must have at least one critical edge.

   Solution: True: Consider the min-cut in the graph. If you lowered the capacity of any edge on this min cut, it would lower the value of the min cut, and therefore the value of the max flow, by the max flow/min cut theorem.

   (c) There exists a graph with a bottleneck edge that is not a critical edge.

   Solution: False. If I increased the capacity of a bottleneck edge, and this increased the max-flow, then this must also increase the min cut. In other words, this means that this edge must be a part of every min-cut of the graph. Since this edge is in *every*, it is certainly in *some* min-cut, and if we decreased it, the max flow would also decrease by the same argument as part b). Therefore, this is impossible.

   (d) There exists a graph with a critical edge that is not a bottleneck edge.

   Solution: True. The graph in part a) is an example. Both edges are critical, but neither are bottleneck edges.

   Hint: Two of the above are true.

   Solution: Note: All of the above statements are basically consequences of the following characterization: An edge is a critical edge iff it is in some min-cut of G. An edge is a bottleneck edge iff it is in every min-cut of G.

4. Prove or disprove the following statements. State clearly whether you are proving the statement True or False.

   (a) It is impossible to have an acyclic, strongly connected directed graph with at least two vertices.

   Solution: True: Say for the sake of contradiction that such a graph existed. Let u and v be two such vertices in G. As G is acyclic, it can be topologically sorted. Say wlog that u appears first in the list. Since G is strongly connected, there must be some path from v to u. This would require some edge going "backwards" in the list, since v appears after u in the topological sort. This is a contradiction, since no such edges can exists in a topological sort.

   (b) In every undirected, connected graph G, the edges of a shortest path between any two vertices is a part of *some* MST of G.

   Solution: False. Consider a graph that is a cycle with at least four vertices, where all edges have weight 1, except for one edge (u,v) of weight 2. The shortest path from u to v is to take the edge of weight 2, however this edge is not a part of the unique MST for G.

   (c) If you added a constant c to the weight of every edge in a graph G, then the every MST of this graph is also an MST of G.

   Solution: True. Unlike the shortest path case, this property is true for MSTs. Every spanning tree of a graph has $|V| - 1$ edges, thus adding a constant c, positive or negative, will change the weight of **every** spanning tree by $c(|V| - 1)$. Thus the edges of any MST of the original graph will also be an MST of the modified graph.

   (d) If there is a unique s-t Max Flow in a directed graph (only one solution), then there is a unique s-t Min Cut in that graph.

   Solution: False. Consider the graph with only a path from s to t of two edges, each with capacity 1. The unique max flow has value 1, while there are two possible min-cuts of value 1, one for each edge.

5. (5 points) (Bonus) You are given an $n \times m$ matrix with real entries in the range $[0, 1)$. The sum of the entries in every row and in every column are an integer. Your goal is to round the *values* in the matrix up or down to 0 or 1 in such a way that the row and column sums stay the same. Values that are already 0 can't be changed.

   Give an algorithm that can perform the above task, with explanation of correctness and running time. Hint: Think max flow.

Solution: Let's denote the rowsum at row i by r_i, and the column sum at column j by c_j. Create one source, one sink, one vertex u_r for each row r, and one vertex v_c for each column c. Connect each row r to a source vertex with capacity equal to the row sum of r, r_i. Also connect each column c to the sink vertex with capacity equal to the column sum of c, c_j. Finally, for every non-zero entry $(i, j) \in (0, 1)$ in the matrix, add an edge of capacity 1 from u_i to v_j.

So the entries of the matrix are represented as edges between the row vertices and column vertices. Each row i must output r_i amount of flow to its outgoing edges (i.e. have r_i entries in it's row set to 1), and similarly each column j must receive c_j amount of flow (i.e. have c_j entries in its column set to 1). There are cuts coming out of the source and sink of size $\sum_i r_i = \sum_j c_j$, so a max flow that reaches this amount corresponds to valid assignment of entries with the desired row and column sums.

Finally, because our original non-integer flow (the original values of the matrix) was a flow of size $\sum_i r_i = \sum_j c_j$, the min cut can't be smaller than this. Thus that is the value of the min cut, and the Ford-Fulkerson algorithm, using shortest paths, will find an integral flow of that capacity in $O(|V||E|^2)$. For an $n \times n$ matrix, we have $O(n)$ vertices and $O(n^2)$ edges, for a total $O(n^5)$ algorithm.

6. (5 points) (Bonus) Given a directed graph $G = (V, E)$, we are interested in finding the *fewest* edges we need to add in order to make G strongly connected.

(a) (2 points) Give a *linear* time algorithm that will return a set of edges to add to G that are *at most twice* as many as optimal. You need to prove that the resulting graph will be strongly connected, and that you use at most twice the optimal number of edges. (What's a lower bound on the optimal number of edges?)

Solution: As above, we repeat the following idea. Run the SCC algorithm and identify all source and sink components in linear time. In order for the graph to be strongly connected, we MUST at least add outgoing edges from each sink node, and incoming edges to each source node. Thus we MUST use at least $\max |Sources|, |Sinks|$ additional edges to make the graph strongly connected. By connecting all the sources and sinks into a single cycle, we would make the graph strongly connected by the following argument: For any vertices u and v, follow a DFS from u to some sink node, and v in reverse to some source node. There is a path from that sink node to that source node in the new added cycle, thus there is a path from u to v. This would use $|Sources| + |Sinks|$ edges, which you can see is at most twice $\max |Sources|, |Sinks|$.

(b) (3 points) Give a *polynomial* time algorithm that can find an *optimal* set of edges to add to G to make it strongly connected. You need to prove that the resulting graph will be strongly connected, and that the number of edges you use is optimal.

Solution: Run the SCC algorithm and identify all source and sink components in linear time. In order for the graph to be strongly connected, we MUST at least add outgoing edges from each sink node, and incoming edges to each source node. Thus we MUST use at least $\max |Sources|, |Sinks|$ additional edges to make the graph strongly connected. We now describe how to optimally decide how use exactly $\max |Sources|, |Sinks|$ additional edges to do so.
We repeatedly do the following

- Pick a source node, and run DFS to see which of the sink nodes it can reach.
- If there is a sink node that it cannot reach, add an edge from that sink node to this source. Since one sink can reach the other source, this causes the source node to no longer be a source, and the sink node to no longer be a sink. Thus by adding one edge, we create a graph G' where both $|Sources|$ and $|Sinks|$ have decreased by 1. We may now repeat this argument.
- If this source could reach all sink nodes, then adding this edge could potentially only decrease the number of sinks by 1, which is a problem if there are more sinks than sources. Thus instead we start at a sink, and look to see which sources can reach it. By a similar argument, if there is a source that cannot reach this sink, then we may add a single edge while reducing the total amount of additional edge by 1.
- In the final case, say all sources can reach all sinks. Then simply add back edges from any sink to any source as needed, using exactly $\max |Sources|, |Sinks|$ edges.

Scrap

Scrap