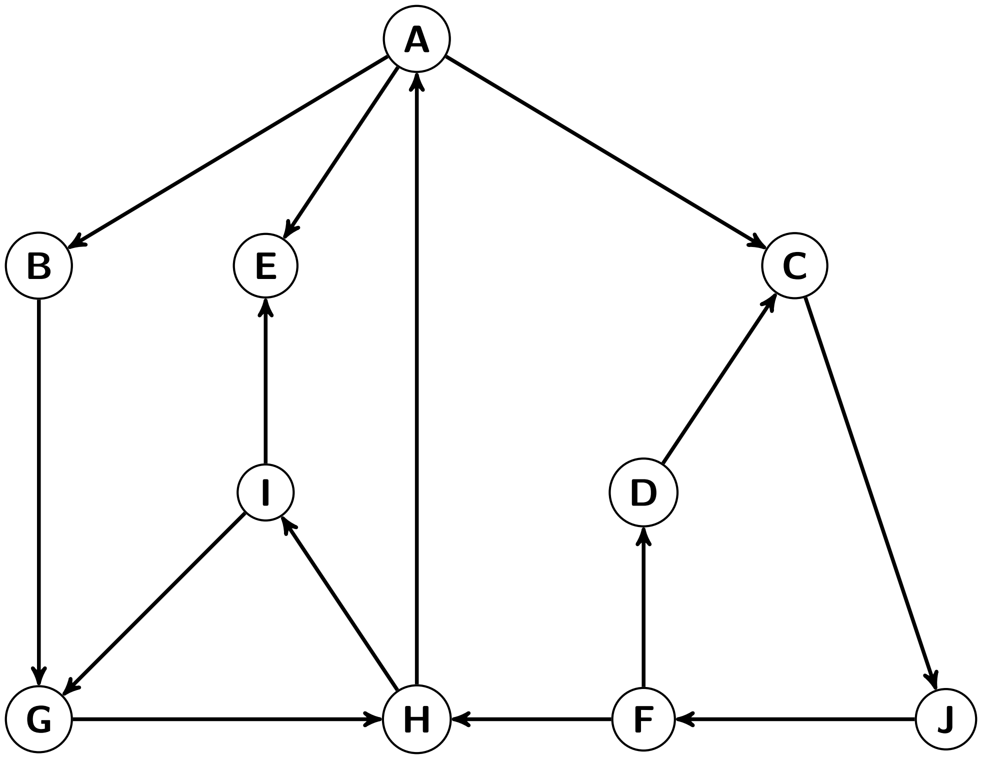
Graph Traversals



1. What are all the strongly connected components? (i.e. groups of vertices such that there exists a path between any two vertices in the group)

SOLUTION: $\{E\}, \{A, B, C, D, F, G, H, I, J\}$.

Problem Solving Notes:

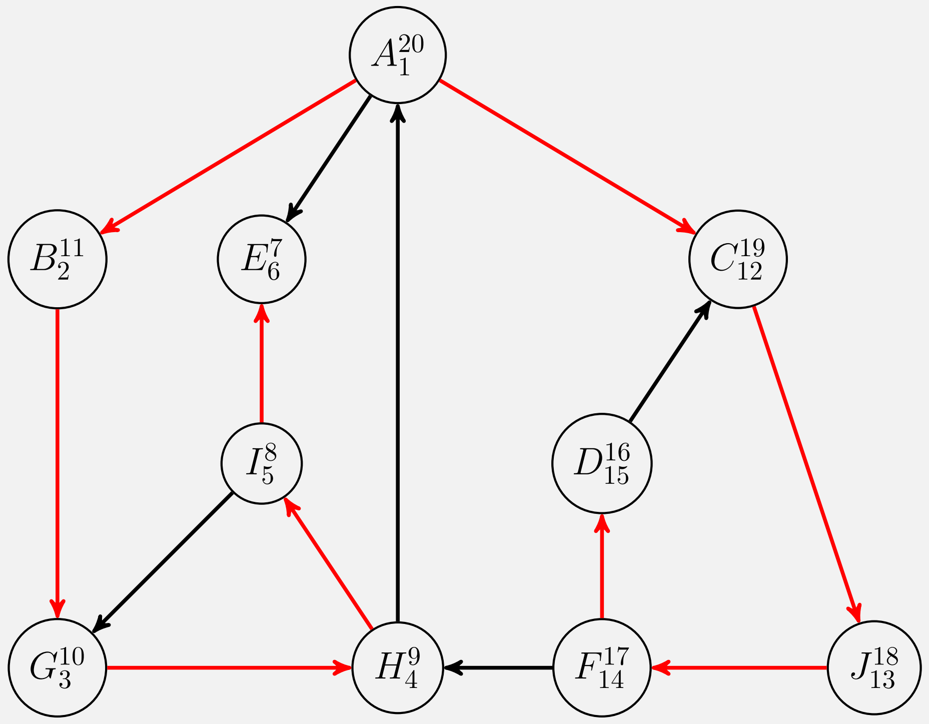
(a) *Read and Interpret:* We are asked for the strongly connected components of the graph, so we must identify sets of vertices that satisfy the path between any two vertices condition.

(b) *Information Needed:* How can we ensure each SCC is as large as it needs to be (i.e. includes all vertices it can)? Is there a way to "rule out" a particular node from a potential SCC?

(c) *Solution Plan:* Since the graph is sufficiently small, we can start with the "sinkiest" vertices and see which nodes are reachable from it. We can then eyeball if every node reached by this vertex is reachable from any other vertex connected to it. Repeating this procedure will identify the SCCs without needing to run Kosaraju's explicitly.

2. Perform DFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the start time and the finish time. Highlight the edges in the tree generated from the search.

SOLUTION:



Problem Solving Notes:

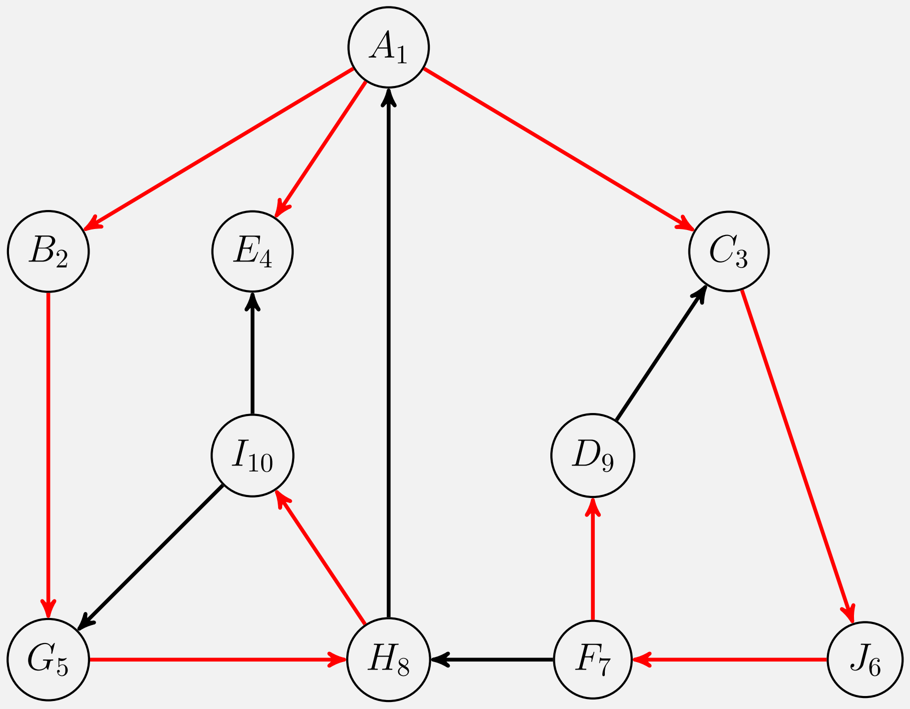
(a) *Read and Interpret:* We are asked to perform DFS with an explicit condition for breaking ties. Following the lecture or section slide examples on this is a fantastic place to start!

(b) *Information Needed:* When should we be updating our counter to track start and end times? What conditions should be met before we mark a node as completed?

(c) *Solution Plan:* Beginning at node A, proceed to the children of A in lexigraphical order, recursively calling DFS on the children nodes. We can mark a node as finished once each of its children have at least been marked as started (in most cases, they will all be marked as finished; see exception in True or False part 2). Our counter is updated any time we mark the start or end of a particular node in the graph.

3. Perform BFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the discovery order. Highlight the edges in the tree generated from the search.

SOLUTION:



Problem Solving Notes:

(a) *Read and Interpret:* We are asked to perform BFS with an explicit condition for breaking ties. Following the lecture or section slide examples on this is a great place to start!

(b) *Information Needed:* When should we be updating our discovery order counter? How does this traversal differ from DFS?

(c) *Solution Plan:* Beginning at node A, proceed to the children of A in lexigraphical order, immediately marking each node as it's discovered. Repeat this procedure for each of the child nodes, continuing until all nodes are explored.

True or False

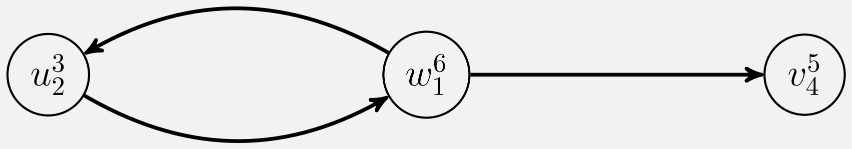
1. If (u, v) is an edge in an undirected graph and during DFS and $finish(v) < finish(u)$, then u is an ancestor of v in the DFS tree.

SOLUTION: True. When we do DFS, we annotate "visited" nodes with starting/finishing times to keep track of the order in which they were visited. There are two scenarios by which $finish(v) < finish(u)$ could happen. It could be the case that u was visited, then v was visited, then v was marked as finished, then u was marked as finished. This makes u an ancestor of v, since we both started and finished marking v before we finished marking u.

The only other scenario is if v was marked as started and finished before u was marked as started (and finished). However, since there is an edge between u and v, this scenario would never happen in DFS since you explore all neighbors before marking yourself as finished.

2. In a directed graph, if there is a path from u to v and $start(u) < start(v)$ then u is an ancestor of v in the DFS tree.

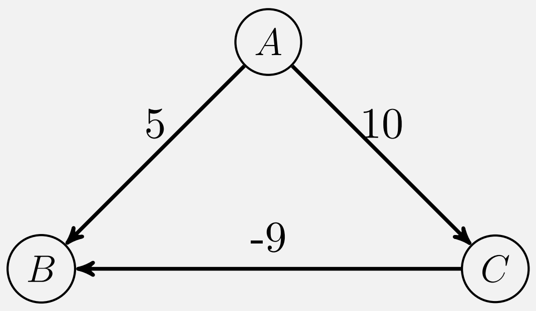
SOLUTION: False. Consider the following case:



Here, u and v are siblings since we started DFS at node w, the parent of u and v. Note: notice that part 1 had an edge between u and v while this does not specify such an edge (only a path).

3. Dijkstra's can be used for successfully finding the shortest path from a source to all other vertices in a graph with negative edges, but not with negative cycles.

SOLUTION: False. Consider the following case:



Once Dijkstra has marked node B as "I'm sure" with cost 5, it will not try to find another path to B. Thus, it will never update the cost to 1.

4. Bellman-Ford's algorithm can be used for successfully finding the shortest path from a source to all other vertices in a graph with negative edges, but not with negative cycles.

SOLUTION: True. Unlike Dijkstra's algorithm, Bellman-Ford does not mark nodes as "done" until computing the shortest path to all nodes from a single source over the maximum path length, thus making it tolerant of negative weights. That is, in the example given above, Bellman-Ford first identifies the shortest path from A to B as 5 when scanning over length-1 paths. It then proceeds to check all length-2 paths and sees that A to C to B produces a shorter path with value 1. Since 2 is the maximal path length for this graph, the algorithm terminates, producing the correct answer.

It's worth noting that Bellman-Ford can additionally identify the presence of a negative cycle (Do you see why?)

5. To find the shortest path from one vertex to another in an unweighted graph, you should use Dijkstra's algorithm as it is the most efficient solution.

SOLUTION: False. For an unweighted graph, you should use BFS. It is more efficient because it does not need a data structure that can handle fast operations of type extractMin() and decreaseKey().

6. Adding a new positive edge to an undirected weighted graph with positive edges cannot lead to the output values of Dijkstra's increasing.

SOLUTION: True. Adding a new edge to graph can have the following possibilities:

(a) If the edge is in the new shortest path to a vertex v from source s, then the minimum-cost will reduce.

(b) If the edge is not in the new shortest path to a vertex v from source s, then the minimum cost will not be affected.

In either case, the cost of the shortest path can only decrease.

Russian Boxes

You have n boxes. The i-th box has dimensions $w_i \times h_i$. Box i can fit inside box j if and only if $w_i < w_j$ and $h_i < h_j$. A sequence of boxes $b_1, b_2, ..., b_k$ form a chain if box b_i fits inside box b_{i+1} for each $1 \leq i < k$. Design an algorithm which takes as input a list of dimensions $w_i \times h_i$ and returns the length of a longest possible chain of boxes. You must construct a directed graph as part of your solution.

SOLUTION: Our algorithm will construct a directed graph whose vertices are boxes, and such that there is an edge (v_i, v_j) iff box v_i fits inside box v_j. Notice that this graph is a DAG, since you can only go in one direction between boxes. Our goal is now to find the longest path.

1. First start by topologically sorting the graph using the procedure outlined in lecture (run DFS starting at an arbitrary box, then sort the boxes by decreasing finish times).

2. Linearize (relabel) these boxes so that the first box in topologically sorted order is v_1, the second is v_2, and so on. We see that under this labelling, whenever there is an edge from v_j to v_i, it holds that $j < i$.

3. For every node v_i, define ℓ_i be the length of the longest path ending at v_i (e.g. $\ell_1 = 0$, since there are no incoming edges to v_1). We can compute ℓ_i recursively as follows:

$$\ell_i = 1 + \max_{(v_j, v_i) \in E} \ell_j$$

4. Because we have linearized the graph, ℓ_i depends only on the longest path of smaller boxes ℓ_j for $j < i$. Thus, we proceed by computing the ℓ_i values in increasing order of i.

The answer is given by $\max_{i=1}^{n} \ell_i$.

Problem Solving Notes:

1. *Read and Interpret:* We are asked to compute the length of a longest possible chain of boxes using a directed graph construction. What would each vertex represent within our graph? What about each edge?

2. *Information Needed:* Which algorithms have we seen that can help us find the length of a path? Is there anything we've seen that specifies an ordering between features of a graph? Once we've specified an ordering, how can we ensure that the chain we create is the longest?

3. *Solution Plan:* Noting that the Russian box condition imposes that our graph is a DAG, we begin by topologically sorting our vertices. Since there are no time requirements for our algorithm, we can begin by naively computing all paths that traverse nodes in topologically sorted order, taking the maximum of all such paths. As an improvement to this initial step, we note that we can express the longest path length from node i in terms of the longest path length from node i's parents, from which the linearizing algorithm follows.

Bipartite Graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) connects a vertex from U to V or a vertex from V to U. A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. In lecture, we saw an algorithm using BFS to determine where a graph is bipartite. Design an algorithm using DFS to determine whether or not an undirected graph is bipartite.

SOLUTION: The algorithm is essentially the same as that of DFS, except at every node we visit, we either color it if it hasn't been visited before, or check its color if it has been visited before. The rough algorithm is as follows:

1. Start DFS from any node and color it RED.
2. Color the next node BLUE.
3. Continue coloring each successive node the opposite color until the end of the tree is reached.
4. If at any point a current node is the same color as one of its neighbors, then return false.
5. Once every node has been visited, if we haven't returned false, then the graph is bipartite.

Problem Solving Notes:

1. *Read and Interpret:* We are asked solve the bipartite graph problem using DFS instead of BFS. Since we aren't asked for a new algorithm, we simply need to figure out what additional bookkeeping we should do to DFS to solve this different problem.
2. *Information Needed:* Are the start/end times sufficient for determining whether the graph is bipartite? Is there a way to rule out with certainty a graph's bipartiteness given the procedure of our DFS algorithm?
3. *Solution Plan:* We follow the same coloring scheme procedure that BFS does by enforcing that no edge connects two vertices of the same color. Starting with an arbitrary color, each time we encounter a new node, we are forced to color that node a specific color to ensure bipartiteness. If we reach an impossible situation (i.e. where either coloring of a particular vertex will break the graph's bipartiteness), we can safely say that no such coloring exists.

Source Vertices

A source vertex in a graph $G = (V, E)$ is a vertex v such that all other vertices in G can be reached by a path from v. Say we have a directed, connected graph that has at least one source vertex.

1. Describe a naive algorithm to find a source vertex.

SOLUTION: You can simply perform DFS/BFS on every vertex and find whether we can reach all the vertices from that vertex. This approach takes $O(V(E+V))$ time, which is very inefficient for large graphs.

2. Describe an algorithm that operates in $O(V + E)$ time to find a source vertex.

SOLUTION: If there exists a source vertex (or vertices), then one of the source vertices is the last finished vertex in DFS. (Or a source vertex has the maximum finish time in DFS traversal).

A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

How does the above idea work?
Let the last finished vertex be v. Basically, we need to prove that there cannot be an edge from another vertex u to v if u is not another source vertex (or alternatively phrased, there cannot exist a non-source vertex u such that $u \to v$ is an edge). There can be two possibilities.

(a) Recursive DFS call is made for u before v. If an edge $u \to v$ exists, then v must have finished before u because v is reachable through u and a vertex finishes after all its descendants.

(b) Recursive DFS call is made for v before u. In this case also, if an edge $u \to v$ exists, then either v must finish before u (which contradicts our assumption that v is finished at the end) OR u should be reachable from v (which means u is another source vertex).

Algorithm:

(a) Do DFS traversal of the given graph, restarting at an unvisited vertex if DFS completes with vertices remaining. While doing traversal keep track of last finished vertex 'v', This step takes $O(V + E)$ time.

(b) If there exists a source vertex (or vertices), then v must be one (or one of them). Check if v is a source vertex by doing DFS/BFS from v. This step also takes $O(V + E)$ time.

Problem Solving Notes:

(a) *Read and Interpret:* We are asked to describe an algorithm that locates a source vertex (given one exists) in $O(V + E)$ time. What graph algorithms have we seen before that operate in time $O(V + E)$?

(b) *Information Needed:* Can we leverage the fact that our graph is directed and connected? The problem states that the graph will have at least one source vertex present. Is there a way we can check that a candidate vertex is a source within the time bound $O(V+E)$?

(c) *Solution Plan:* We can lean on our intuition from lecture that "sourcier" vertices will finish later than their "sinkier" counterparts in DFS. In lecture 9, we saw that this is evident in the case that we start at a source vertex. It remains to show that this holds more generally, like if we don't start at a source vertex, which we can do using a contradiction argument. A proof by contradiction seems like a good fit here because of the inherent "for all" in the source vertex definition. That is, it's easier to prove the negation (i.e. there exists some vertex w that is unreachable from v) instead of proving all vertices are indeed reachable from v.