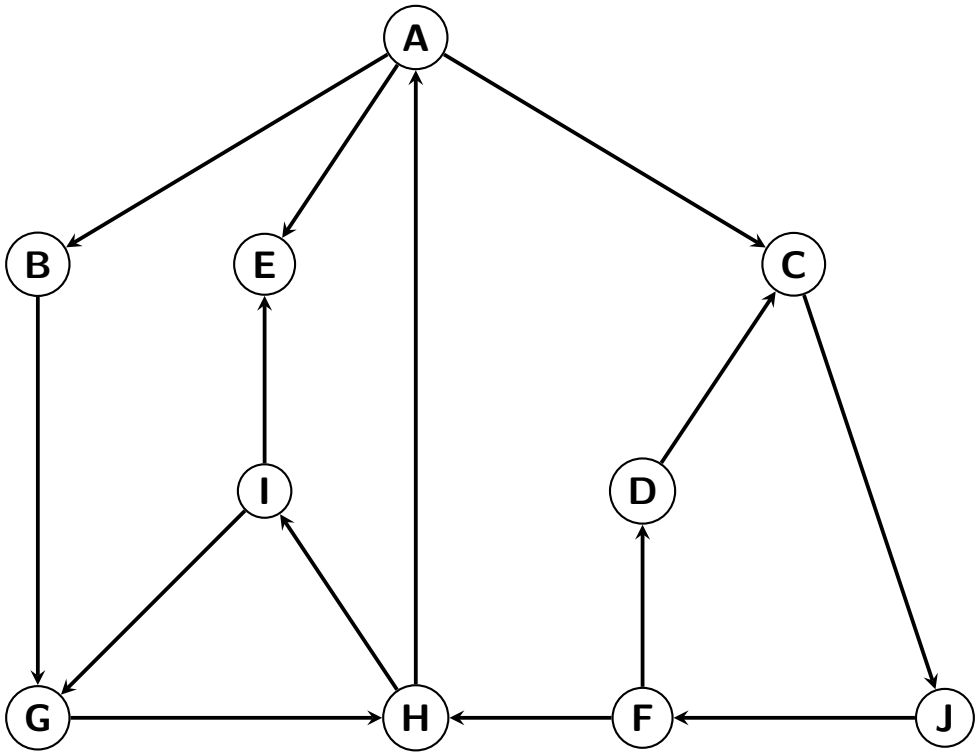
CS 161 Winter 2020 Section 5

February 13-14, 2020

1 Strongly Connected Components

Ten "friends" decided to make a Valentine's Day graph in which each person is a node, and an edge points from v_i to v_j if Person j has a crush on Person i. Below is their graph:



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

{E}, {A, B, C, D, F, G, H, I, J}. The only strategy here is to first notice that E is a lonely node, and the rest is just visually verifying that the rest of nodes can all reach each other (i.e. satisfy the definition of an SCC).

2 Russian Boxes

Through hard work and a small stroke of good luck you have glowed up over the past year and are suddenly inundated with admirers for Valentine's Day, who have each sent you a box of chocolates (if you are allergic

to chocolates, you may assume they sent you strawberries or something else nice that comes in a box). After you finish consuming the spoils of your attractiveness, you are left with n empty rectangular boxes (you may assume n VERY large), and you decide to nest some of them within each other for easy storage. 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 the longest possible chain of boxes. You must construct a directed graph as part of your solution.

BONUS: Having found the length of the longest possible chain, how can you use your directed graph to return the chain itself? (If there is more than one longest chain, you may return any of them)

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 (it is impossible for some box i to fit in box j and also contain j). Our goal is now to find the length of the longest path.

We're going to approach this by trying to compute the length of the longest chain that ends at any node v_i. We think to ourselves, "for any box i, it would be amazing if only we knew the max chain lengths that end at any of the boxes that can fit into box i - we'd then just pop that chain of boxes into box i and we would have extended that chain by 1!". This is the idea of *DYNAMIC PROGRAMMING* (DP), which is a concept we'll cover in the first lecture of week 7. Basically, DP is all about re-using answers to subproblems ('memoization', if you've heard it from 106B), and the key thing is to figure out what that subproblem structure is. In our case, we'd like to solve for 'smaller' boxes first, or really, we want to solve in order of what the topoSort result of our graph would give us (think of it as each node depending only on the boxes that could fit in it). Thus we linearize (i.e., create an ordering $v_1, \ldots, v_n$ of the nodes via topological sort) the graph, so whenever there is an edge from v_i to v_j , $i < j$.

For every node v_i, let ℓ_i be the length of the longest path ending at v_i. We can compute ℓ_i as follows:

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

Because we have linearized the graph, ℓ_i depends only on ℓ_j for $j < i$. That is, the length of a chain ending at some box i only depends on the lengths of the chains ending at all boxes j that can fit inside i. Thus we can compute the ℓ_i values in order of "size," and our longest chain has length $\ell^* = \max_{i=1}^{n} \ell_i$. One way to do this is to start at node 1 and add "weights" to each edge of our directed graph, where weights are equivalent to the length of the longest path to the source node (i.e., the weight of edge (v_i, v_j) is ℓ_i).
(This is really a great example of a dynamic programming problem–we will cover more details about dynamic programming later next week, so stay tuned for that, and don't worry if it's not an obvious approach to you yet!)

To return the chain itself, we can do a modified version of DFS (or BFS) in which we keep track of the length of our current paths and return once we have found a path of the desired length ℓ^*. Here's one method: construct a starting set S of all nodes that have no edges in. For each v in S, begin a recursive depth-first-search from v which takes two arguments: the current node and the current length of the chain. Rather than having a specific target node to catalyze returning a solution, we return only when the current length is equivalent to our precomputed maximal length.

Other things to think about on your own: is it better to use DFS or BFS? In what situations would one be faster than the other?

3 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) either 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.

You are attending a talk on Valentine's Day in hopes of making an intellectual connection with someone. The speaker is presenting empirical research on romantic relationships in classroom settings, with a case study of students who attend CS 161 section. They are presenting their findings using a graph of student relationships, but you are somewhat skeptical that they may have not observed everything. Your first step to testing your hypothesis is to check if the graph they present is bipartite.

1. Design an algorithm using DFS to determine whether or not a graph is bipartite.

   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:

   (a) Start DFS from any node and color it RED

   (b) Color the next node BLUE

   (c) Continue coloring each successive node the opposite color until the end of the tree is reached

   (d) If at any point a current node is the same color as its parent node, then return false

   (e) Once every node has been visited, if we haven't returned false, then the graph is bipartite

2. Design an algorithm using BFS to determine whether or not a graph is bipartite.

   (a) Start BFS from any node and color it RED

   (b) For each of it's neighbors:

   - If it hasn't been visited before, color it the opposite color as its parent
   - If it has been visited before, check to make sure its color is the opposite of its parent, otherwise return false

   (c) Once every node has been visited, if we haven't returned false, then the graph is bipartite