CS-3250

Section 01

-

Take Home Exam #2

Patrick Pei

Professor Jeremy Spinrad

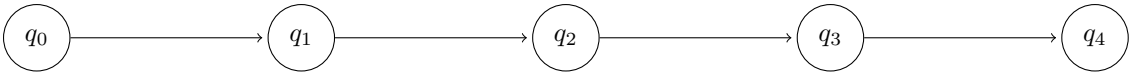
Due 2-6-2017

Exercises

1

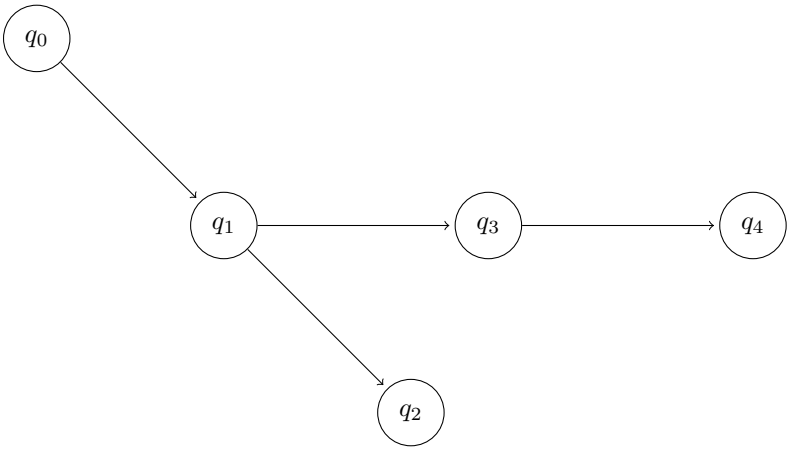
1a) Draw a graph G on 5 vertices, and assign each vertex a number so that each vertex number is equal to both the depth first search number that each vertex number is equal to both the depth first search number and the breadth first search number in some BFS and some DFS and G.

G



1b) Draw a graph G' on 5 vertices such that numbers cannot be assigned to vertices such that vertex numbers correspond to both depth first search numbers and breadth first search numbers for any DFS and BFS of G', and explain why it is not possible to assign such numbers for G'.

G'



It is not possible to assign numbers such that vertex numbers correspond to both depth first search numbers and breadth first search numbers for any DFS and BFS of G' because of the very definitions of DFS and BFS. At every step, BFS examines a vertex v and and subsequently visits all vertices that v has a path to. On the contrary, DFS examines a vertex v and subsequently visits a neighbor of v and then the

same process is applied to the neighbor and it is evidently that this process (as aptly named) will constantly expand outward. Hence, in a graph G', as long as there exists a vertex v such that v has at least 2 neighbors and any one of these neighbors has a neighbor that is not one of v's neighbors, it will not be possible to assign such numbers due to the nature of BFS and DFS.

2 Design an $O(n+m)$ algorithm to arrange all lists in an adjacency list representation of G so that vertices occur in increasing order of vertex name on each list.

In order to design an $O(n+m)$ algorithm to arrange all lists in an adjacency list representation of G so that vertices occur in increasing order of vertex name on each list, extra memory must be used. In other words, the algorithm will not be performed in place on the current adjacency list representation of G. Next, the current adjacency list representation of will be traversed by vertex number starting from the lowest vertex number to the highest, and for each of these vertices, the adjacency list will be traversed and the vertex name will be added to the new adjacency lists. In other words, because the traversal is starting from the lowest numbered vertices, there can be no vertex before it on any adjacency list. The idea is extended on step so that after the lowest vertex's adjacency list is traversed, the new lists will have lists such that the only vertex in the list is the lowest. This means that for the lowest vertex, it is first in all the lists that contain it. Finally, it is clear that the next lowest vertex is then taken and because it is guaranteed to be the lowest apart from every vertex traversed before it, the same process can be applied until all lists are formed such that vertices occur in increasing order of vertex name on each list. This algorithm runs in $O(n+m)$ time because each step takes $O(n+m)$ time. As discussed in class, forming the initial adjacency lists (the ones that do not guarantee increasing order in every list) takes $O(n+m)$ time since every vertex is examined and edges out of each of these vertices is examined (and it is evident that no vertex or edge needs to be looked at more than once). Finally, it is enough to show that since appending to a list is clearly $O(1)$, traversing the original adjacency lists in order will have the same property - no vertex is examined more than once, and each of these lists will only be traversed once (the sum of these lists is m). Thus, since the runtime is now bound by some constant c * (n + m), the overall runtime in order notation is $O(n+m)$.

3 Design an $O(n+m)$ algorithm to find all vertices which are on every path from x to y in a graph G. Hint: create a graph of the biconnected components, similar to the graph of strongly connected components.

The key point to note is that if such a vertex v exists such that v is on every path from x to y in a graph G, there is obviously no way to get from x to y without v. Thus, v must be an articulation point. Next, in order to find these above mentioned vertices, the algorithm shown in class to identify articulation points will be used and each of these articulation points will be kept track of. This algorithm correctly identified all articulation points in a graph G in $O(n+m)$ time by essentially performing a DFS but calculating a LOW property where LOW is recursively defined as

$$min(DFS\#(neighbors(v)), LOW(children(v)))$$

. Finally, to identify the articulation points, if LOW(v) was greater than or equal to DFS#(parent(v)), the parent is an articulation point. The special case here is for the root which would only be marked as an articulation point if it had more than 1 child. Next, a DFS will be performed on x and throughout the traversal, all articulation points, which were previously calculated, that the DFS is passing through will be kept track of as well. At the conclusion of the DFS, if y is not found, evidently there is no path from x to y. If y is found, then every articulation point that was on the path from x to y is a vertex on every path from x to y in G. This can also be thought of as a graph of biconnected components, similar to the graph of strongly connected components shown in class, such that each vertex is a biconnected component. In this simplified graph, it can be seen that within a biconnected component, there are no such vertices that are on every path within two vertices of the biconnected component. Conversely, if x and y are not within the same biconnected component, it is clear that every articulation point in between will be on every path from x to y. The runtime of this algorithm is $O(n+m)$ since the algorithm learned in class correctly identifies articulation points in $O(n+m)$ time and a DFS as previously proven in class as well takes $O(n+m)$. Thus, the algorithm described runs in $O(n+m)$ time.

4 Someone in class last year suggested a new notion of connectivity; a directed graph is sort-of-connected if there is some vertex which has a path to all other vertices.

Design an $O(n + m)$ algorithm to determine whether a graph is sort-of-connected. Hint: First build the graph of strongly connected components.

First, the graph of strongly connected components will be built by using the algorithm covered in class where the premise was based on the hint given from the movie about how the object had to be flipped upside-down: the graph with every edge between vertices reversed has the same strongly connected components as the original graph. This algorithm first performs an initial DFS in $O(n+m)$ time and in the second traversal of the graph (the graph such that the edges are reversed), if LOW(x) ¿= DFS#(x), then the component would be broken off as a separate strongly connected component. This second traversal of the tree is evidently also in $O(n+m)$ time. Finally, now that the graph of strongly connected components is built, the properties of this graph can be utilized to find some vertex v which has a path to all other vertices, if it exists. Since the resulting graph is connected (if not connected, it is immediately clear that no such vertex exists and that graph is not "sort of connected"), directed, and acyclic (every acyclic graph **does** have a topological sort). To actually find the vertex, topological sort as taught in lecture can be applied. The key point in this step is that since, as described in lecture, topological sort will store lists of vertices by in-degree. Lastly, the only point left to be realized is that in this directed, acyclic graph, this property that some vertex v has a path to all other vertices can be checked by simply examining the lowest level of the lists formed by topological sort. There must be a list L_0 formed by toplogical sort such that L_0 is the list where all vertices in L_0 contain in-degrees of 0. Again, since this is directed and acyclic, seeing a single vertex v in L_0 means that this vertex can reach all others, this is the very property of directed and acyclic. If there are multiple vertices in L_0 it is clear that without cycles, there is no way for one vertex in L_0 to reach any other vertex in L_0. Thus, since the runtime described in the construction of the graph of strongly connected components was shown to be $O(n+m)$ and the runtime of toplogical sort is $O(n+m)$, and since examining the previously mentioned L_0 can take no longer than n time, the overall runtime of this algorithm is $O(n+m)$.