EE360C: Algorithms
University of Texas at Austin
Dr. Evdokia Nikolova & Dr. Pedro Santacruz

Problem Set #5
Due: October 16, 2014 (*in class quiz*)

Problem Set #5

You should try to solve these problems by yourself. I recommend that you start early and get help in office hours if needed. If you find it helpful to discuss problems with other students, go for it. **You do not need to turn these problems in. The goal is to be ready for the in class quiz that will cover the same or similar problems.**

Problem 1: Graph Theory

A connected, undirected graph is *vertex biconnected* if there is no vertex whose removal disconnects the graph. A connected, undirected graph is *edge biconnected* if there is no edge whose removal disconnects the graph. Give a proof or counterexample for each of the following statements:

(a) A vertex biconnected graph where $|E| > 1$ is edge biconnected.

Solution

The graph with two nodes and a single edge is vertex biconnected but is not edge biconnected. If we remove the single edge, the two nodes are no longer connected. If we remove either one of the vertices, the graph (of a single) node is still connected (it's the degenerate case).

Assume our graph has more than one edge. Suppose it is not the case. That is, suppose there is a graph that is vertex biconnected but is not edge biconnected. Then no matter which vertex I remove, the graph stays connected, but there is some edge that, when I remove it, the graph is no longer connected. Let's call that edge (u, v). When I remove (u, v) from the graph, the result is a partition of the graph such that u is in one partition and v is in the other partition, and there is no edge across the partition. The subgraph that contains u is connected; the subgraph that contains v is connected. (u, v) is the only edge across the partition in the original graph. If, in the original graph, I removed either u or v instead of (u, v), that would also remove (u, v). Since our graph has more than one edge, one of u and v must have additional adjacent vertices. Let's assume u has adjacent vertices (other than v). Removing u from the original graph should leave it connected (that's the premise). However, removing u also removes (u, v). Since (u, v) was the only edge across the two partitions, this leaves v disconnected from u's other adjacent nodes. This is a contradition (since by removing vertex u, we disconnected the graph).

(b) An edge biconnected graph is vertex biconnected.

Solution

A graph with five nodes A, B, C, D, X that contains the edges (A, B), (C, D), (A, X), (B, X), (C, X), and (D, X). This graph is edge biconnected but not vertex biconnected (we could remove any edge, and the graph will stay connected, but if we remove X, the graph is disconnected).

Problem 2: 2-Colorable Graphs

An undirected graph $G = (V, E)$ is said to be k-colorable if all of the vertices of G can be colored one of k different colors such that no two adjacent vertices are assigned the same color. Design an algorithm based on BFS that either colors a graph with 2 colors or determines that two colors are not sufficient. Argue that your algorithm is correct.

Solution

We are given a graph G in which all nodes are designated as "uncolored". We must develop an algorithm to color the graph G with two colors or determine that two colors are not sufficient. The general algorithm is as follows: Pick any arbitrary vertex v in the graph G and color it red. Then, for every single neighbor of the vertex v, we will stick the node in a queue Q. Then we will proceed with the following algorithm:

```
BFS 2-COLOR($G$)
1  while $Q$ is not empty
2      do $v \leftarrow$ pop ($Q$)
3         for each neighbor $n$ of the vertex $v$
4             do if $n$ is uncolored
5                   then color it the opposite color of vertex $v$ and then push $n \rightarrow$ ($Q$)
6                if $n$ is colored && color($n$) == color($v$)
7                   then return FALSE
8  return TRUE
```

If it returns TRUE, then G has been colored into 2 colors. If returned FALSE, we know 2 colors is not sufficient

Problem 3: Depth First Search

(a) During the execution of depth first search, we refer to an edge that connects a vertex to an ancestor in the DFS-tree as a *back edge*. Either prove the following statement or provide a counter-example: if G is an undirected, connected graph, then each of its edges is either in the depth-first search tree or is a back edge.

Solution

This is true. Suppose it were not true. Then there would be an edge in G that is not in the DFS-tree that is also not a back edge. This edge connects a node u to a node v. When u was included in the DFS-tree, we chose not to include (u, v) in the DFS-tree. The only reason we would do this was if v had already been included in the DFS-tree. But v is not an ancestor of u in the DFS-tree, so v must be in some other branch of the DFS-tree. But then, when we included v in the DFS-tree (which was before we examined u and u's neighbors), we chose not to include the edge (v, u). But the only reason we would have done this would have been if u had already been in the DFS-tree. So there's the contradiction.

(b) Suppose G is a connected undirected graph. An edge whose removal disconnects the graph is called a *bridge*. Either prove the following statement or provide a counter-example: every bridge e must be an edge in a depth-first search tree of G.

Solution

This is true. Suppose it were not true. Consider the edge e that connects vertices u and v, where e is a bridge. That is, if e were removed from the graph G, then we would have a partition of the vertices of G such that no edge crosses the partition. Start a depth first search in the partition that includes u. Eventually, since G is connected, this DFS must reach vertex u. When it does, it will examine all of the vertices adjacent to u and incorporate any who have not yet been "touched" into the DFS-tree. This includes v. We are guaranteed that v has not yet been examined since the edge (u, v) is the only edge that crosses our partition of vertices, and we started our DFS in the partition that included u. Therefore, the edge (u, v) must be included in the DFS, giving us our contradiction.

Problem 4: Discipline in Groups of Children

Your job is to arrange n rambunctious children in a straight line, facing front. You are given a list of m statements of the form "i hates j". If i hates j, then you do not want to put i somewhere behind j because then i is capable of throwing something at j.

(a) Give an algorithm that orders the line (or says it's not possible) in $O(m + n)$ time.

Solution

Create a graph G in which each child is a vertex and every statement of the form "i hates j" results in a directed edge from i to j. Topologically sort the vertices. Use a modified topological sort that returns "false" or some other marker if the graph has a cycle (if the graph has a cycle, it's not possible to line the children up safely). If the topological sort succeeds, line the children up according to the topological sort result.

```
TOPOLOGICAL-SORT-CHILDREN($G$)
1  select $v \in G.V$ with no incoming edges
2  **if** $v =$ NIL and $|G.V| \neq 0$
3      **then return** false
4  output $v$
5  $G.V = G.V - v$
6  TOPOLOGICAL-SORT-CHILDREN($G$)
```

The order of the output vertices (children) is the reverse order to put them in line; you can reverse the order in $O(n)$ time (or you could have made edges from j to i instead of from i to j) Creating the graph takes $O(n + m)$ time; running topological sort takes $O(V + E) = O(m + n)$ time.

(b) Suppose instead that you want to arrange the children in rows such that if i hates j, then i must be in a lower numbered row than j. Give an efficient algorithm to find the minimum number of rows needed, if it is possible.

Solution

The minimum number of rows is the longest path in the graph constructed in part a. For general graphs, finding the longest path is an NP-complete problem, meaning it cannot be solved in polynomial time. However, for topologically sorted graphs, this can be done in $O(E)$ time.

We will assume the graph can be topologically sorted and that it has no cycles. We will run the following algorithm on the first node in our topologically sorted graph:

```
LongestPath($s$)
1  dist[] ← {−1, −1, −1, ...}
2  dist[$s$] ← 0
3  for each vertex $v$ in topological sort
4       do for each adjacent vertex $v$ of $u$
5              do if dist[$v$] < dist[$u$] + 1
6                    then dist[$v$] ← dist[$u$] + 1
7  return max(dist)
```

For each node, we are checking if the nodes adjacent to it provide a longer path. Thus, the running time of this algorithm is $O(N^2)$ or $O(E)$.