CSE 6331 HOMEWORK 8

BRENDAN WHITAKER

1. *Given a directed* ***acyclic*** *graph $G = (V, E)$, write an $O(V^2)$ algorithm to determine whether for all pairs of nodes $u, v \in V$, there is a path from u to v or a path from v to u. Describe your algorithm in (high-level) pseudo-code and explain it in plain English. (Hint: topological sort.)*

   Note this is not the same as checking whether or not the graph is weakly connected, this condition is stronger than weak connectedness. We give high-level pseudocode for this algorithm:

Algorithm 1: Pathfinder(G)

Data: The set of vertices $V = \{ 1, 2, ..., n \}$ and edges E of the given graph G. The 2D array adjacency matrix $A[1..n][1..n]$.
Result: The truth value of the statement "for all pairs of nodes $u, v \in V$, there is a path from u to v or a path from v to u."

```
1  begin
       /* global array L[1..n]                                          */
2      initialize L[1..n] ⟵ 0;
       /* We first do a topological sort using DFS.                     */
       /* Use depth-first search, with a L of size n initialized to 0.  */
       /* At the end of procedure dfs(v), insert the index of v in A to the
          front of L.                                                   */
3      L[1..n] ⟵ TopSort(G);
       /* Then L gives a topological sort of G.                         */
4      boolean pathExists ⟵ true;
5      for i ⟵ 1 to n − 1 do
6          if A[L[i]][L[i + 1]] == 0 then
7              pathExists ⟵ false;
8          end
9      end
10     return pathExists;
11 end
```

Note that since we have a DAG, we know that we can do topological sort. So we can put a partial ordering on the nodes such that all edges go from a node with lower index in L to a node with higher index in L. So if we want to determine whether or not there is a path from u to v or a path from v to u, if $u = L[i]$ and $v = L[j]$, and if without loss of generality we let $i < j$, then there is a path from u to v or a path from v to u if and only if there is an edge from $L[i]$ to $L[j]$. So if we want to check if

Date: SP18.

this holds for all u, v, clearly we need to check it for all pairs $L[i], L[i+1]$. But is this sufficient? Yes because if there is an path from $L[i]$ to $L[i+1]$ it must be a single edge, else it would contradict our definition of a topological ordering, and if we have such edges for all such $i = 1, ..., n-1$, then we must have a path K that passes through every vertex, starting with $L[1]$ and ending with $L[n]$. Therefore we certainly have a path from either u to v or v to u which is a subset of K for all u, v. So our algorithm finds a topological ordering, and then checks if there is an edge between each pair of adjacent vertices in our ordering. If there is no such edge for any of these adjacent vertices, we must have that the result is false, and else, it is true. The topological sort takes $O(V^2)$ time implemented with an adjacency matrix, and the rest of the algorithm takes $O(n) = O(V)$ time, so the whole thing takes $O(V^2)$ time.

2. *Let $G = (V, E)$ be an undirected graph. Modify the following algorithm so that it answers whether G contains a cycle of odd length. Your modification must not increase the algorithm's time complexity. Do NOT rewrite the whole algorithm, just make the necessary changes.*

Algorithm 2: Search($G = (V, E)$)

```
1  begin
       /* Assume V = { 1, 2, ..., n }.                                          */
       /* global variables: oddCycle, visited[1..n], ancestorLevel[1..n], level */
2      visited[1..n] ⟵ 0;
3      level ⟵ 0;
4      ancestorLevel ⟵ −1;
5      oddCycle ⟵ false;
6      for i ⟵ 1 to n do
7          if visited[i] = 0 then
8              ancestorLevel[i] ⟵ level;
9              dfs(i);
10             ancestorLevel[i] ⟵ −1;
11         end
12     end
13     return oddCycle;
14 end
```

Algorithm 3: dfs(v)

```
1  begin
2      visited[v] ⟵ 1;
3      level++;
4      for each node w such that (v, w) ∈ E do
5          if visited[w] = 0 then
6              ancestorLevel[w] ⟵ level;
7              dfs(w);
8              ancestorLevel[w] ⟵ −1;
9          else if ancestorLevel[w] ≠ −1 && (level − ancestorLevel[w]) ≡ 1 mod 2
               then
10             oddCycle ⟵ true;
11         end
12     end
13     level- -;
14 end
```

3. *Let $G = (V, E)$ be a directed graph. Modify the* ***if statement*** *in the dfs procedure (and modify nothing else) to determine if (v, w) is a tree, forward, back, or cross edge. Your modification must not increase the time complexity of the algorithm. Do NOT rewrite the whole algorithm, just make the necessary changes.*

Algorithm 4: dfs(v)

```
1  begin
2      vn[n] ⟵ time ⟵ time + 1;
3      for each node w such that (v, w) ∈ E do
4          if vn[w] = 0 then
               /* (v, w) is a tree edge.                                  */
5              dfs(w);
6          else if vn[v] < vn[w] then
               /* (v, w) is a forward edge.                               */
7          else if fn[w] = 0 then
               /* (v, w) is a back edge.                                  */
8          else
               /* (v, w) is a cross edge.                                 */
9          end
10     end
11     level- -;
12 end
```