Assignment - 6

Rahul Motan

1.)

Let’s look at this problem as the problem of finding bridges in a graph. A bridge is an edge in an undirected/directed graph that if removed disconnects the graph.  We can solve this problem by following a simple method…

$•$      Start at any node and do a Depth First Search labelling the nodes with an increasing values as we go starting from 1.

$•$     We’ll keep track of the ids of the nodes as we go and we’ll use a simple data structure such as an array (low-cycle[]) to label the         nodes that are present in a cycle and we’ll label those nodes with lowest ids among them.

$•$     Bridges will be found where the id of the node your edge is coming from is less than the low-cycle value of the node your edge is         going to.

ALGORITHM

$id\leftarrow 0$

$adjList \leftarrow  adjacency list$

$totalNodes\leftarrow size of the graph$

$nodeIds\left[totalNodes\right]\leftarrow \left[0,0,.....0\right] $

$lowCycle\left[totalNodes\right]\leftarrow \left[0, 0, 0,......0\right]$

$visited\left[totalNodes\right]\leftarrow \left[false,false,...false\right]$

$locateBridges\left(\right)$    {

    $HashMap\left(bridges\right)$

$for each node i in graph$:

        $if not visited\left[i\right] then$

            $modDfs\left(i,−1,bridges\right)$

$return bridges$   }

// at : the current node we’re looking at

$modDfs\left(cur, parent, bridges\right)$    {

    $visited\left[cur\right]\leftarrow  true$

    $id\leftarrow id+1$

    $lowCycle\left[cur\right]\leftarrow nodeIds\left[cur\right]\leftarrow id$

    $for \left(to : aList\left[cur\right]\right)${

            $if to == parent$

                        $continue$

            $if !visited\left[to\right]${

                // do dfs again

                    $mod−dfs\left(to, cur, bridges\right)$

                    $lowCycles\left[cur\right]\leftarrow $$min\left(lowCycles\left[cur\right], lowCycles\left[to\right]\right)$

                    $if nodeIds\left[cur\right]<lowCycles\left[to\right]$

                            $bridges.put\left(cur, to\right)$

            $else:$

                    $lowCycles\left[cur\right]\leftarrow min\left(lowCycles\left[cur\right], nodeIds\left[to\right]\right)$

Time Complexity: $O\left(V+E\right)$

Space Complexity: $ O\left(V\right)$

$$

2. )

An articulation point is a point in an undirected graph on removal of which disconnects the graph or we can say if vertex (u, v) form a bridge then either ‘u’ or ‘v’ is an articulation point. So in the given problem the group of islands could get disconnected if the islands that serve as articulation points get bombed.

The algorithm to find the articulation point is as follows:-

$•$      Start at any node and do a Depth First Search labelling the nodes with an increasing values as we go starting from 1.

$•$     We’ll keep track of the ids of the nodes as we go and we’ll use a simple data structure such as an array (low-cycle[]) to label the         nodes that are present in a cycle and we’ll label those nodes with lowest ids among them.

$•$     On the callback,  if  the id of the node we’re currently at is equal to the low-cycle(toNode) then there  was a cycle.

$•$     The indication of a cycle back to the original node implies an articulation point.

$•$     For the root node or the starting node we need to see if it has atleast two outgoing edges then it can be an articulation point.

PSUEDOCODE:

$id\leftarrow 0$

$adjList \leftarrow  adjacency list$

$totalNodes\leftarrow size of the graph$

$noOfOuterEdges\leftarrow 0$

$nodeIds\left[totalNodes\right]\leftarrow \left[0,0,.....0\right] $

$lowCycle\left[totalNodes\right]\leftarrow \left[0, 0, 0,......0\right]$

$visited\left[totalNodes\right]\leftarrow \left[false,false,...false\right]$

$articulationPoints\leftarrow \left[false, false,...false\right]$

$locateArticPoints\left(\right)$    {

    $HashMap\left(bridges\right)$

$for each node i in graph$:

        $if not visited\left[i\right] then$

             $noOfOuterEdges \leftarrow 0$

            $modDfs\left(i, i,−1\right)$

            $if noOfOuterEdges>1$

                        $articulationPoints\left[i\right]\leftarrow true$

$return bridges$   }

// at : the current node we’re looking at

$modDfs\left(root, curNode, parent\right)$    {

      $if root equal to parent$

            $noOfOuterEdges\leftarrow  noOfOuterEdges+1$

    $visited\left[curNode\right]\leftarrow  true$

    $id\leftarrow id+1$

    $lowCycle\left[curNode\right]\leftarrow nodeIds\left[curNode\right]\leftarrow id$

    $for \left(to : aList\left[curNode\right]\right)${

            $if to == parent$

                        $continue$

            $if !visited\left[to\right]${

                // do dfs again

                    $mod−dfs\left(root, to, curNode\right)$

                    $lowCycles\left[curNode\right]\leftarrow $$min\left(lowCycles\left[curNode\right], lowCycles\left[to\right]\right)$

                    $if nodeIds\left[curNode\right]\leq lowCycles\left[to\right]$

                            $articulationPoints\left[at\right]\leftarrow true$

            $else:$

                    $lowCycles\left[curNode\right]\leftarrow min\left(lowCycles\left[curNode\right], nodeIds\left[to\right]\right)$

  **Time Complexity**:$ O\left(V+E\right)$

  **Space Complexity**:$O\left(V\right)$

3.)

We know a connected acyclic graph is called a Tree.  So the given alkane or saturated hydrocarbon can be intuitively deemed as a tree.

Following are some facts that we would need to devise an algorithm for the given problem:-

1.    Any tree with $v$ vertices has $v−1$ edges.

2.    Sum of degrees of all the vertices is equal to twice the number of edges in the Tree.

            $≡deg\left(v\_{1}\right)+deg\left(v\_{2}\right)+....+deg\left(v\_{k}\right) = 2\left(k−1\right)$

So considering each of carbon and hydrogen atoms as vertices and the bonds between them as the edges we can consider the molecule to be a tree.

We are given $k$ carbon atoms and $l $ hydrogen atoms. So we have $k+l$ vertices and thus with $k+l−1$ edges.

And we know the degree of Carbon atom at maximum can be 4 and for Hydrogen atom it can be 1.

Since there are k vertices with degree 4 and l vertices with degree 1, we can write

                                                    $4k+l=2\left(k+l−1\right)$

                                                $⇒l=2k+2$

                                                $⇒k=\frac{\left(l−2\right)}{2}$

ALGORITHM

$if l<4 or l is odd then$

        $return null$

$else$

         $return \frac{\left(l−2\right)}{2}$

4.)

Technical Problem: Given a graph find its topological sort.

The given problem is of scheduling the inter-dependencies between the sub-processes. We can consider each sub-process as a vertex and create a directed graph by creating an edge from one sub-process to the other sub-process that is dependent on it. We can find out the dependencies among the sub-processes and list them or order them in such a way that would delineate the relationship among the processes using Topological Sorting.

Topological Sorting can only be applied to DAGs so if such an ordering is not possible then we will be able to identify such a scenario. This scenario basically occurs when there is a presence of a cycle in the graph.

Kahn’s Topological Sort

In this algorithm we start doing BFS on the node whose in-degree is 0 i.e there are no incoming edges to that node. The vertices on the outgoing edge are considered and that edge is removed. The vertex is checked again for more incoming edges if there are none then its added to the list of sorted vertices.

**ALGORITHM**

$L\leftarrow $ An Empty list for storing the vertices

$S\leftarrow $ Set of all vertices with no incoming edges

$while s is not empty do${

    remove a vertex $n$ from $S$

    insert $n $ into $L$

    $for$ each vertex $m$ with an edge $e$ from $n$ to $m$ do

        remove edge $e$ from the graph

        if  $m$ has no other incoming edges

            insert $m$ into $S$

if graph has edges then

        return error( graph has cyclic dependencies)

else

        return $L$

5.)

We are given a few sub-processes, we need to group them together to the same category to which they belong. To achieve this we will need to save some additional information.

We will make two tables, one table will contain A(vertex, in-degree) and the other table will contain B(category, in-degree, noOfVertices) .

We will create an adjacency list such that we will save the category and with that category we will save all the nodes belonging to that category.

ALGORITHM:

We will follow Kahn’s Topological Sort Algorithm but we will have to modify it slightly and use the information we have stored in our data structure.

In this algorithm we start doing BFS on the node whose group’s in-degree is 0 i.e there are no incoming edges to that group. We will pick the category from Table B.

The vertices on the outgoing edge are considered and that edge is removed. Table A is updated if the removed edge is within the same category.

If the removed edge was to another node belonging to a different category then we update the Table B and Table A.

The vertex is checked again for more incoming edges if there are none then its added to the list of sorted vertices and removed from the table or queue.

PSUEDOCODE:

$\rightarrow $ TSORT(V,E){

            Create an array in-degree[sizeOfGraph]

            $L\leftarrow  list to store vertices in the grouped order$

              for(each vertex u in v){

                    in-degree( u ) = 0

                }

            for( each edge (u, v) $\in $ E){

                    in-degree(v) += 1

            }

        Create a queue $Q\_{c}$

          Create a queue $Q\_{v}$

         for( each category $label$ in B){

                    if in-degree of B[$label$] $==$ $0$

                        $Q\_{c}.push\left(B\left[label\right]\right)$

        while $Q\_{c }is not empty$:

                    $prevCateg \leftarrow  Q\_{c}.pop\left(\right)$

                    $for each node v \in A\_{list}\left[prevCateg\right]$

                             $if$ in-degree[v] $==$ $0$

                                   $Q\_{v}.push\left(v\right)$

                            $while Q\_{v} is not empty$

                                    $u\leftarrow Q\_{v}.pop\left(\right)$

                                    $L.add\left(u\right)$

                                    $for edge\left(u,v\right) \in A$

                                            in-degree[v] = in-degree[v]-1

                                            $if $ in-degree[v] == 0 and $prevCateg ==v.category$

                                                        $Q\_{v}.push\left(v\right)$

                                               $B\left[v.Category\right]\leftarrow B\left[v.Category\right]−1$

                                                $if B\left[v.Category\right]==0$:

                                                          $Q\_{c}.push\left(B\left[v.Category\right]\right)$

                                        $L.add\left(u\right)$

                                     $if$ graph has edges then:

                                            return error(graph has cycles)

                                    $else:$

                                            return  $L$