CSCE 629 - 602 Analysis of Algorithms

October 12, 2016

Homework V **Rishabh Yadav**
Prof. Anxiao (Andrew) Jiang UIN: 425009824

1A. 22.5-7 Solution

Idea

The idea is to find out the Strongly connected components, C_i and making a new graph from this with each Strongly connected components, C_i as a vertex, lets call this graph G'. Now we perform a topological sort on this graph, G'. If there exists an edge between every consecutive vertex in the topologically sorted graph G' then the graph is Semi-Connected, otherwise it is not semi-connected.

Pseudo Code

Semi_connected (G):

1. Find StronglyConnectedComponents of G.
2. Take each SCC as a virtual vertex and make a new SCC vertex graph G'
3. Perform a Topological Sort on G'
4. For every pair of consecutive vertices (V_i, V_{i+1}) in a topological sort order of G'.
   (a) If there is NO edge (V_i, V_{i+1}) in graph G'.
      i. return NOT-SEMI-CONNECTED
5. return SEMI-CONNECTED.

Proof of correctness

In a topologically sorted DAG, we know, if there exists no edge (v_i, v_{i+1}) for any i, then the graph is definitely not semi connected, since there are no back-edges int the graph G'. Hence the check for semi-connected-ness of G' is right.

The second part is to prove this indeed leads to show semi-connected-ness of the given graph G. Since, the edge between the two components in SCC is actually an edge between

two vertices in G. Let us say the (V_i, V_{i+1}) be the edge in G' between any two SCC. Now since each vertex say u in the SCC represented by V_i is connected to one another and each vertex say v in in SCC represented by V_{i+1} is connected to on another. We can say Every vertex in V_i is semi connected to every vertex in V_{i+1}. Thus the semi-connected test on the SCC is also the semi-connected test on the original graph. Since we do this for each of the vertices in the SCC, we have tested if the original graph is semi connected.

Time complexity

Time taken to find out strongly connected components and to perform Topological sort is O(V+E) each.
Time Complexity = $O(V + E)$

2A. 22.3 Solution

Part a

We need to show that a graph G has an Euler tour if and only if in-degree(v)= out-degree(v) for each vertex $v \epsilon$ V. We break this down into two parts.

Part 1

A Euler tour exists $\rightarrow$ in-degree(v) = out-degree(v), $\forall\ v \epsilon$ V

Proof: Euler tour can be thought of as a set of edge-disjoint cycles, which in combination for the complete when combined form the tour.
Now each vertex v int the cycle will have cycle has one edge coming into it and one edge leading out of it. Therefore, in-degree(v)=out-degree(v) $\forall\ v \epsilon$ the cycle under consideration an all edges part of this cycle. This can be said for all the cycles in the graph.
Since the given graph is strongly connected and there exists a Euler Tour, we can generalize the above inference to all the cycles, which ultimately form a Euler Tour. Hence in-degree(v)=out-degree(v) for $\forall\ v \epsilon$ V.

Part 2

in-degree(v) = out-degree(v), $\forall\ v \epsilon$ V $\rightarrow$ A Euler tour exists

Proof: We build cycles in the graph. Start at some vertex v and follow that out going edge from v to get to another vertex. Assuming we get to a vertex other than v, we can leave that vertex (since it has one edge going in, it must have some untouched out going edge). We continue till we arrive back at v (which we eventually end up, as graph G is strongly connected, and v has at least one edge coming in to match its leaving edge with which we began our cycle). We build a cycle for like this for every general vertex $v \epsilon$ V, that starts and ends at v. Moving forward we look at all the untouched edges, say (w, u) and start forming cycles as described above for those edges, say vertex W, We then go to a visited node w connected to said edge, and follow it till we reach back at w. Again, since the graph G is strongly connected and in-degree equals out-degree for every vertex, we are guaranteed that this will happen. Add this new cycle into our existing cycle. When we examine every edge, we are in the situation that every edge has been visited exactly once and is part of the cycle formed. Which is nothing but the Euler tour of the graph.

Part b

Idea

The idea is to take a vertex v and start building cycles in the graph. We then find more edge disjoint cycles unexplored edges and keep adding them into our main cycle, until all edges have been visited. If at some point we cannot make cycle and there exists some unexplored edges we return saying No-Euler-Tour exists. If all the edges have been visited then we return the main cycle as this is us our Euler tour.

Pseudo code

Euler Tour (G)

1. $EulerCircuit = Null$
2. Set of $unexploredEdges = E$ // E are the edges in graph G
3. for all $unexploredEdges$ (u, v):
   (a) newCycle = Null
   (b) Form Cycles starting with v, say pick edge (v, w) then so on and finally (u, v) to arrive back at v.
   (c) remove all the edges used in the previous cycle formation from the unexplored set, *unexploredEdges*.
   (d) if cycle cannot be formed:
      i. return NO-EULER-TOUR
   (e) add cycle to the *EulerCircuit*
4. if *unexploredEdges* is empty:
   (a) return *EulerCircuit*
5. return NO-EULER-TOUR

Proof of correctness

Since we know in an Euler tour all the edges in the graph are visited exactly once. We start out from any vertex, v and make a cycle ending at the same vertex v. Now all the edges used in making this cycle are removed from the unexplored set of edges. This way we make sure we traverse each edge exactly once. Two cases arise here: Case:1, that at some point we cannot form a cycle. Since we are unable to complete the cycle then we can say NO-EULER-TOUR exists. Case:2, that we form a cycle every time but there exists some edges in Unexplored set of edges which are not part of any cycle. Since we are unable to traverse each edge exactly then we can say NO-EULER-TOUR exists.
Since we are able to traverse each edge and make a big cycle out of them which is checked by maintaining a main cycle always and the unexplored set of edges in the last step of algorithm. We can say there exists a Euler tour, which is returned in the penultimate step.

Time complexity

Since every edge is visited exactly once, the algorithm runs in $O(E)$ time.

$Time\ Complexity = O(E)$

2A. 22.1-11 Solution

Idea

The idea is to check if the edge whose weight has been decreased can potentially become part of Minimum spanning tree, T or not. We simply add this edge to the Minimum spanning tree. Now we have got exactly 1 cycle in T. In this Cycle present in MST, T we need to find and remove edge with highest value that is on that cycle. The new tree generated will be the new Minimum Spanning Tree since no other edges in the tree can possibly be changed or modified except the one which is described above. This way we get the new Minimum spanning tree, T'.

Pseudo code

find_Modified_MST(G, T, $e(u, v)$): //$e(u, v)$ is the edge whose weight is decreased

1. add e to T.
2. find cycle in T using DFS.
3. remove the edge with highest weight from the cycle found in previous step, to break the cycle. Say this edge is e'. //$T' = T + e$ - e'
4. return T'. //This is the new Minimum spanning tree.

Proof of correctness

We know that the edge, say $e(u, v)$ whose weight has been modified is not part of the original Minimum spanning tree, T. We also know that in a tree if we add an edge it forms a cycle.
We also know that this edge $e(u, v)$ can only be part of MST if there exist an edge with higher weight in the potential cycle which may form when $e(u, v)$ is added to the MST. When we add this edge in MST T we get a cycle. in order for us to restore $T + e$ to form a MST we need to break the cycle formed by adding $e(u, v)$. If we break this cycle by removing edge with highest weight in this cycle we are done.

Time complexity

All steps except Step 2 can be done in constant time. Finding the Cycle using DFS take O(V+E) time. Hence,
Time Complexity $= O(V + E)$

References

Introduction to Algorithms by T. Cormen, C. Leiserson, R. Rivest, C. Stein