CSCE 629-601 Homework 5
12th October, 2016

Name: Adil Hamid Malla **UIN:** 425008306

1. Exercise 22.5-7 on page 621

INPUT: We have a Directed $Graph = G(V, E)$
OUTPUT: Whether the Graph G is semi-connected or not.

Solution: The problem is asking whether a directed Graph is semi-connected or not. By semi-connected it means that for each vertex u, v $\in E$ we have either a path from $u \to v$ or $v \to u$. If we use our knowledge of the chapter 22 till now, we are talking about whether a graph is semi connected or not. To prove the same we have to check whether there is a path from each path v1 to v2 or v2 to v1. We can leverage the property of the graph whether it is strongly conencted or not and then we can check whether the strongly connected graphs have an edge between them. Using the Strongly Connected Components theorem and its corollaries we can prove that the components of the graph form the Directed acyclic graph. And then if there is an edge between the components then the graph can be sorted topologically. After sorting we can check of there is an edge between adjacent vertices, this will be suffice to prove that there is an edge betweeen each vertex.

The Algorithm can be defined as:

Algorithm 1 Semi-Connected Graph

1: **procedure** SEMI-CONNECTED(G)
2: CALL DFS on our Graph G
3: Compute Transpose of graph G given by $G' = (V, E')$
4: Call DFS on our G' considering decreasing order of finish time in first DFS
5: Now we will form the component graph of all the BFS trees
6: From the Lemma 22.14 and Corollary 22.15 If we have an edge from C1 to C2 then $f(C1) > f(C2)$
7: Topologically sort component graph, as Component Graphs make a DAG
8: Verify that the vertices forms the linear chain in each component graph
9: Check verify whether we have edges like (v_1, v_2), $(v_1, v_2) \ldots (v_{n-1}, v_n)$
10: If the vertices form the linear chain then the graph is semi-connected otherwise not
11: **end procedure**

Correctness: The correctness of this algorithm is very easy. First of all lets talk about the individual component graphs, as from the concept of the strongly connected graph which we computed in Algorithm step 2,3,4 and 5, We are ensuring the vertices among the components graphs are reachable from each other which is the definition of the Strongly Connected Components. Now also we know that the components of the graphs will make a Directed Acyclic Graph and/or Forest. Thus we only have to check whether there is a an edge between the two components which follows our requirement to prove the original Graph is semi-connected.

Time Complexity: As far as the time complexity of this algorithm is concerned, we can see that we are applied the DFS two times and then linearly sorted the components graphs in topological order using their finish times and also inside each component graph we have used the topological sorting using the finish times, thus the time complexity of our algorithm is $O(|V| + |E|)$

2. Problem 22-3 on page 623

INPUT: We have a strongly connected Directed Graph $G = (V, E)$.
Definition of Euler Tour : Euler tour of strongly connected graph is a cycle that traverses each edge of G exactly once, although it might visit a vertex more than once.

($a.$) G has an euler tour if and only if in-degree of vertex is equal to out degree of the vertex for all vertices in V
Solution: The proof to this problem is tricky we need to prove two things since we are proving if and only if condition here.

1. The first is going in the forward direction, where if the graph is Euler tour then above claim should hold. If our Graph G has an euler tour ET, then we have either simple cycle in ET or complex cycle. Cycle will be present since our graph is strongly connected graph. Simple cycle is a cycle which doesn't

intersect itself. Now if we are following that the our ET has simple cycle then it means that the in-degree = out-degree =1, so our basic claim remains true.
Now if we have ET which doesn't not simple cycle, i.e ET contains complex cycle, then also the complex cycle is comprised of the simple cycles. We will start removing the simple cycles from the ET and the Graph G until no edges are left. Basically what we are doing is that at every node we are removing one incoming edge and outgoing edge of the cycle. After a cycle of the deletion we have decreased the in-degree and out-degree of a node on the cycle by exactly one. At the end we of this procedure we are left with the nodes where the in-degree and out-degree are 0. So all the vertices v must have started with in-deg = out-degree.

2. The second part of the proof is other way around, if we have in-degree = out-degree then we have an Euler Tour. If every vertex v has in-degree equal to out-degree, then we assume that for all these vertices there will be path which comes back to this vertex following a cycle. If our assumption holds then we find a cycle C in out graph randomly from a any vertex which comes back to the same vertex. Now we delete all the edges in this cycle C in graph G. Now in the updated G we will still have in-degree = out-degree as we have deleted the one incoming and one outgoing edge of each node in cylce C. Now from the Cycle we pick a vertex v-new on Cycle C that has edge incident and repeat the procedure. Following this procedure we will find first one cycle then another Cycle C' which will have atleast one vertex common with C and so on and so forth. From repeating this procedure we can build a big cycle that goes around C jumps into C' and then C" and then comes back to C.

   Now we have asssumed simple thing that for any vertex v, there must be a cycle that contains v. Start from v, and chose any outgoing edge of v, say (u, v). Since in-degree $(u) = out - degree\,(u)$ we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v, at the time we visit an entering edge, there must be an outgoing edge left unvisited, since in-degree = out-degree for all vertices. The only vertex for which there may not be an unvisited outgoing edge is vbecause we started the cycle by visiting one of vs outgoing edges. Since theres always a leaving edge we can visit for any vertex other than v, eventually the cycle must return to v, thus proving the above assumption.

(*b.*) Describe an efficient $O\,(|E|)$ algorithm to find the Euler tour of G if one exists.
Solution: Now from the above proof of part a. we conclude that the graph which has in-degree of each node equal to out-degree then the Euler tour exists for that graph. we will leverage the same concept we used in the first part to check the same.
Now to solve this we will check whether the graph given has the in-degree for each vertex equal to out-degree, this will take $O\,(|E|)$ time to find it. If the above condition is true then we know that the Euler tour exists for this graph else we return no Euler tour exists. Now to get the Euler tour we will do the same procedure we did in the first part of this question, finding and removing the cycles one by one.
The algorithm's pseudo code is given:

Algorithm 2 Euler Tour of Graph

```
1:  procedure EulerTour(G)
2:      Check if the in-degree of each vertex is equal to out-degree
3:      If check return true then we proceed to find the Euler Tour
4:      Find Euler tour
5:  end procedure
6:  procedure EulerTourFind(G)
7:      Pick a vertex v and perform DFS on it until we find back edge to v
8:       traverse the cycle
9:      while traversing we delete the corresponding edge from the adjaceny list of Graph
10:     we delete the edges in constant time using the midify the dfs, we store each edge we traverse in a pointer
11:     we repeat till no edges are left
12: end procedure
```

The Correctness is already proved in the part first and the time complexity of this algorithm is $O\,(|E|)$ as we are running the dfs only E times as the number of edges present.

3. Problem 23.1-11 on page 630

INPUT: We have Graph $G = (V, E)$ and a minimum spanning tree T of the graph.
OUTPUT: If we decrease the weight of one edge in graph G not in T, Algorithm to find new spanning tree of the modified graph.

Solution: The algorithm for updating the new spanning tree is easy but proving it for all the cases is troublesome. Now say we have new edge **e** $= (u, v)$ whose weight has been decreased, then adding **e** to T creates a cycle C in T. Let **e'** be maximum weighted edge in the cycle C. if $w(e') > w(e)$, then $T' = T \cup \{e\} - \{e'\}$ otherwise T' = T. The Algorithm is defined as:

Algorithm 3 Minimum Spanning Tree Update

1: **procedure** MINIMUM SPANNING TREE($G, w(u, v)$)
2: Insert the new edge $w(u, v)$ in the Tree T to form T'
3: Detect the cycle C in the T'
4: Calculate the heights weight edge from the Cycle C and remove the edge from the T'
5: return the new T' as updated minimum spanning tree
6: **end procedure**

Correctness Here we have T as the minimum spanning tree of the graph G. Now lets assume that we modify the edge **e** to get **G'**. According to the minimum spanning tree algorithm we can assume that **T'** is the new MST which may be equal or less than that of weight of T. .
Firstly, T' is a tree - we create exactly one cycle in the algorithm, and break it, so we have no cycles in T'. Secondly T' is a spanning tree of G'. Let e' be the edge removed and e" be the edge added in the algorithm (we have either e"=e or e"=e) . To be a spanning tree, we must have a path between every pair of vertices u, v using only edges of T'. Suppose that in T (which is a spanning tree), the path from u to v do not involve e', then the same path exists in T'. Alternatively, suppose that it did use e, then there is a path (without loss of generality) from u to one endpoint of e' and from the other endpoint of e' to v. There is also a path from one endpoint of e' to the other endpoint via e" (around the cycle), all within T'. Then we can construct a path from u to v via e" in T' by merging these three paths and removing the overlap (although a walk is sufficient for connectivity).
Now, the important part, we wish to prove that T' is a minimum spanning tree for G'. There are two cases for that

1. Case 1: The algorithm does not add e to the tree. In this case T'=T. Suppose that there is a minimum spanning tree H for G' that is different to T'. If H has the same weight as T', we're done. Now suppose for contradiction that the weight of H is less than the weight of T'. There must be some edge e' of lowest weight that is in H but not in T' (there must be some edge that does better, otherwise G would not be of lower weight than T', moreover we can assume that the edge that does better is the lowest weight edge that's not in T' - we can take any H' that is a lower weight tree than T' and look at the candidate for e', if it is not smaller than any edge in its cycle, then either H' is not an MST, or we can create a new H' where we swap the e' for some edge of T', this process must terminate with an edge e' which has the property that it is the edge that does better).

   If e'!= e, then consider the tree obtained by adding e' to T (note, not T'), and removing the highest weight edge on the cycle formed. This new tree has weight less than that of T and is a spanning tree for G, contradicting the fact that T is an MST for G - so we know this can't happen. If e'=e, consider the cycle formed by adding e' = e to T' (i.e. the one the algorithm considered). All other edges in the cycle have lower weight than e' (otherwise the algorithm would've included e as an edge), and hence must be in H (as e' is the lowest weight edge that is not already in T'), but then H must contain a cycle, so isn't a tree and we have a contradiction.

2. Case 2: The algorithm adds e to T'. Let x be the edge in T that is removed by the algorithm (and hence not in T') Again assume we have another MST H as before. If the weight is the same, we're happy. So assume for contradiction that H has lower weight, and as before e' is the lowest weight edge in H that's not in T'. We can make similar arguments as before with x.

   If e' != x, (note also that e'!= e), then we can improve T as before, but we know that T is an MST, and recalling the property that we can assume e' has lower weight that at least one edge in the cycle its addition induces, this gives a contradiction and H can't exist. If e'= x, then again e' must have higher weight than all the other edges in the cycle, hence H must contain all these edges and H is not a tree, and we derive a contradiction.

So in every case we derive a contradiction, therefore there can be no spanning tree of lower weight that T', hence T' is a minimum spanning tree for G'.
The running time of this algorithm will be time spent in detecting the cycle in a tree T'. i.e $O(|V| + |E|)$