Introduction to Algorithms
Assignment 5

Sourabh S. Shenoy
(UIN:225009050)

October 12, 2016

Contents

1. Semi-connected Graph

1.1 Question

A directed graph G=(V,E) is semi-connected if, for all pairs of vertices (u,v), we have u –> v or v –> u. Give an efficient algorithm to determine whether or not G is semi-connected. Prove that your algorithm is correct, and analyze its running time.

1.2 Solution

1.2.1 Idea:

To check if a graph G is semi-connected, we need to follow the following steps:

In a strongly connected component, all vertices are mutually reachable. All such components should be reduced to a node, such that we get a directed acyclic graph in the original graph. In this, we perform topological sorting to see if it is semi-connected. If this graph is semi connected, then the entire graph is semi-connected.

1) Obtain all strongly connected components of the graph G

2) Topologically Sort all the nodes of the Strongly Connected Components

3) Suppose we have 'n' vertices, then we should have atleast n-1 edges, connecting each successive pair of vertices in the topologically sorted graph.

1.2.2 Pseudocode:

Input: Graph G

Step 1: Run StronglyConnectedComponents(G)

Step 2: Treat each Strongly Connected Component as a node and form a graph G'

Step 3: Run TopologicalSort(G')

Step 4: Check if for each consecutive node in G', we have an edge in G'. If yes, return true

Step 5: Return false

StronglyConnectedComponents:

Step 1: Run DFS on G to find the finish time f(u) for each node u belonging to V

Step 2: Run DFS on Gt, where each time we pick a root node, we pick the undiscovered node with the greatest finish time. Return the nodes of each DFS tree as a strongly connected component

TopologicalSort:

Step 1: Run DFS to find the finish time f(n) of each node n belonging to V. As each node is finished, insert it into the front of a linked list.

Step 2: Return the linked list

1.2.3 Proof of Correctness:

If we can prove that there is an edge between any two consecutive vertices, it follows that there is a path between all pairs of vertices. Thus, the graph is semi-connected. The above algorithm proves that there is an edge between consecutive pairs of vertices in the directed acyclic graph obtained by reducing the strongly connected components to nodes. Hence, there also should be a path from all vertices in any component to all vertices in any other component. Thus, the algorithm is correct.

1.2.4 Time Complexity:

The time complexity of this algorithm is the time required to find the strongly connected components, as well as perform topological sorting. Thus, it is O(V+E), since topological sort uses DFS.

2. Euler Tour

2.1 Question Part A

An Euler tour of a strongly connected, directed graph G=(V,E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

Show that G has an Euler tour if and only if in-degree(v) = out-degree(v), for each vertex vϵ V .

2.2 Solution Part A

Since we have if and only if, we need to prove both cases. i.e; A Euler Tour exists => indegree = outdegree and indegree = outdegree => Euler Tour exists.

Let us prove the first case, A Euler Tour exists => indegree = outdegree.
For every vertex v in G, each edge having v as one of it's end points, is included only one in the Euler Path. Each time the path enters v, it has to exit v through another edge. Suppose the path enters vertex v 'n' times, it also has to exit v 'n' times, thus making the indegree(v) = outdegree(v). This is true even if there are multiple paths in the graph, since the logic holds true for each cycle, and even when the cycles are merged at a node, the relation indegree=outdegree will hold true.

Next, we need to prove that if indegree(v) = outdegree(v) for all v => Euler Path exists.
Consider a cycle C1. Suppose it starts and ends at a vertex v, it is evident that each vertex in the cycle has indegree = outdegree. A graph may have multiple such cycles. For each of those cycles, consider a vertex that has a common vertex. Since we can start at any vertex and arrive at the same vertex in a cycle, suppose we have 2 cycles at a vertex, we can start at the vertex, traverse both cycles and return to the same vertex. Thus, indegree = outdegree for each vertex in all those cycles, and we are able to traverse the entire graph without repeating the nodes. Thus, a Euler circuit/Tour exists.

2.3 Question Part B

Describe an O(E) time algorithm to find an Euler tour of G if one exists. (Hint: Merge edge-disjoint cycles.)

2.4 Solution Part B

2.4.1 Idea:

A Euler tour consists of several edge-disjoint cycles, each connected by a common vertex. Hence, we can start from a vertex, and build a cycle. We then find other cycles in the graph, such that it has a common vertex with atleast one already built cycle, and the add it to the Euler Tour. We stop when all edges in the graph are included. In other words, all possible cycles are added to the tour.

2.4.2 Pseudocode:

Input: A Graph G

Step 1: Choose a random vertex v from G

Step 2: Start constructing a cycle C. v is the first vertex in the cycle.

Step 3: While Euler Tour does not include all edges, do

Choose a vertex u from G

Construct a new cycle in G, from u. We can use DFS till we find a back edge to u.

Merge the new cycle with existing cycles in the Euler Tour T

Step 4: Return the Euler Tour T

2.4.3 Proof of Correctness:

We know that a Euler path is composed of several edge-disjoint cycles, connected at a common vertex. The algorithm proposed above, starts from a vertex and adds a cycle to the path. It then proceeds to find other cycles in the graph such that any one vertex u, already exists in the selected cycle. Once all edges are covered, we know that we have several edge-disjoint cycles, all connected by one vertex. Thus, a path can start from a cycle and exit the cycle through the connecting vertex to the next cycle and so on, till the entire graph is traversed. Thus we find a Euler Tour using the above algorithm.

2.4.4 Time Complexity:

Since each node is visited atleast once, the worst case time complexity is O(E), where E is the number of edges in the graph.

3. Minimum Spanning Tree

3.1 Question

Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges not in T. Give an algorithm for finding the minimum spanning tree in the modified graph.

3.2 Solution

3.2.1 Idea:

Suppose the weight of the edge, say uv, not in minimum spanning tree is decreased, then we can have 2 cases:

Case 1: The reduced weight is less than some edge in the path from u to v in the minimum spanning tree.

Case 2: The reduced weight is not less than any edge in the path from u to v in the minimum spanning tree.

3.2.2 Pseudocode:

Input: Graph G

Step 1: Given the edge uv, find a path P in the minimum spanning tree that connects u to v. This can be done using BFS.

Step 2: Traverse through the path and find the maximum cost edge in the path. This can be done by using DFS from u to v

Step 3: If the cost of the maximum edge is less than the cost of edge uv, then do nothing and return

Step 4: Else, delete the maximum cost edge and add edge uv to the minimum spanning tree

3.2.3 Proof of Correctness:

We have to prove that the algorithm works both when we pick or not pick the edge uv. According to Kruskal's algorithm, suppose we dont pick uv, it implies that it was not picked in the minimum spanning tree because it would create a cycle, if picked. Suppose we pick uv, then we are going to pick other edges such that the weights are minimum. This would end up excluding the edge with the maximum weight in the path from u to v. Thus, even if we pick uv, the algorithm above gives the correct answer.

3.2.4 Time Complexity:

The time complexity of this algorithm is O(V), the same as that of DFS for a tree.