Markiyan Varhola
A2032471

Homework Assignment 7

1. Exercise 22.5-3 on page 620 of CLRS3.

The suggestion that using the original graph in the second depth-first search and scanning the vertices in order of increasing finishing times would not work. A counterexample to this suggestion would be if the graph has vertices {1,2,3} and edges (2,1),(2,3),(3,2). The strongly connected components of the graph would be {1} and {2,3}.

One possible depth-first search that starts at 2 could visit 3 before 1, which would mean that 3 has a shorter finishing time than 1 and 2. If the depth-first search was to start at 3, however, then the search would be able to reach all vertices and return a single strongly connected component. This does not work since there is no direct path from 1 to 2 or from 1 to 3.

2. Problem 22-1 (a) on page 621 of CLRS3.

<u>Property 1: There are no back edges and no forward edges.</u>
If there is a back edge, it means that there must be two vertices: an ancestor and a child of that vertex. However, there is a path from the ancestor to the child which does not require moving up the tree. This contradicts the fact that the only children in the tree are one edge away, which means that there cannot be other paths as it would make the number of edges to the children more than one.

A forward edge would mean that from a given vertex there is a child vertex that has been processed. For a child vertex to be processed, the algorithm would need to go through a different path to reach the child, and since children are only one edge away, this is a contradiction.

Property 2: For each tree edge (u,v), we have v.d = u.d + 1

A tree edge is put on a list of edges to be processed if it goes to a vertex that has not been considered. The path from that vertex to the root must be at least the distance from the current vertex + 1. Another possible path is going to the current vertex and using its path to the root.

Property 3: For each cross edge (u,v), we have v.d = u.d or v.d = u.d + 1

A cross edge cannot go farther than depth-1, as it would be used as a tree edge instead. It cannot go to a depth of one more because a vertex that far from the root would not have been processed yet. Because the depths of the cross edge vertices cannot be more than one apart, we can interchange v or u.

3. Given a graph G with weighted edges and a minimum spanning tree T of G, give and analyze an algorithm to update the minimum spanning tree when the weight of an edge e in G decreased.

If an edge *e* is already in *T*, then nothing needs to be done. *e* still belongs to the minimum spanning tree after the weight is decreased. If we are deleting the edge e from T, we will create an intermediate spanning forest with two components. Since e is a safe edge before we decrease its weight, it is not impacted. If the edge e is not part of T, then there must be a path in T that forms a cycle between the endpoints of e. Thus, the new minimum spanning tree will include the maximum-weight edge of the cycle. To find this maximum-weight edge, we can perform either a breadth-first search or a depth-first search, which is O(V nodes + E edges).

4. (a) Describe and analyze an algorithm to find the best shortest path from s to t.

Begin by initializing an array **dist**, which will contain the distances from the starting vertex s to each vertex in the graph. **dist(s)** will be initialized to 0, while for all the other vertices v, **dist(v)** will be initialized to infinity. The distance from the source to each vertex will be finalized when the shortest distance to that vertex is found.

Next, we initialize a queue **Q,** which will contain all of the vertices in the graph. We also initialize a set **S,** which will initially be empty. The set **S** will be used to indicate which vertices have been visited.

The following steps are then performed:

1. While Q is not empty: pop the vertex **v** (that is not in **S**) from Q which has the smallest dist(**v**). The starting vertex **s** will be chosen first as dist(**s**) was initialized to 0.
2. Add the vertex **v** to **S**, indicating that v has been visited.
3. Update dist values of adjacent vertices of the current node **v**:
    a. If dist(**v**) + weight(u,v) < dist(u), there is a new minimal distance found for u, so update dist(u) to the new distance value.

Once the algorithm visits all of the vertices in the graph, then it will find the smallest distance to each node. **dist** will now contain the shortest distance from every vertex to the s. It is only necessary to look at the parent vertices of a vertex v to trace it back to the start.

Initially building a priority queue takes O(V) time, since we need to add every vertex in the graph to the queue. We then choose vertices (nodes) with only the smaller distance, which totals to O(V log V). A for loop is executed for each edge in the graph, and then updates the minimum distance of each vertex. This takes a total of O(E log V) time. Together, the total runtime of the implementation takes O((E+V) log (V)) time.

(b) Implement your algorithm (in whatever programming language you like) and submit the program, sample data (graphs with at least 15 vertices and 25 edges), and its output. The TAs will select students at random to demonstrate their programs.

The algorithm program is attached.

References

1. Bodnar, Michelle, and Andrew Lohr. "CLRS Solutions." *CLRS Solutions*, Rutgers University, sites.math.rutgers.edu/~ajl213/CLRS/CLRS.html.
2. Thaddeus Abiy, Hannah Pang, Jimin Khim. *Dijkstra's Shortest Path Algorithm*, https://brilliant.org/wiki/dijkstras-short-path-finder/.