JONATHAN PETERSEN
A01236750

# Homework 6

## Graph Algorithms 1

1.a. Design an O(m + n) time algorithm to determine whether there exists a path from s to every other vertex of G.

First, traverse the graph using the Breadth-First Search (BFS) described in class. Then check each vertex to find any that are still white. If there are still white vertices after performing a BFS on s, then there exists no path from s to the white vertex. This is true because BFS explores every possible path from a starting node. Since the BFS runs in O(m + n) time, and searching the vertices runs in O(n) time, this algorithm has a total running time of O(m + n) time.

1.b. Design an O(m+n) time algorithm to determine whether there exists a path from every other vertex of G to s.

For each entry in the adjacency list, create a reverse entry in a new list, creating an inverse-adjacency list. This list will have entries in s for every vertex that connects to s, so simply perform a BFS on s and once more check if there are any unvisited nodes. This algorithm takes O(m + n) time to compute the inverse-adjacency list, O(m + n) time to perform the BFS, and so overall takes O(m + n) time.

1.c. Prove the following statement: G is strongly connected if and only if there is a path in G from s to every other vertex and there is a path from every other vertex to s.

Having a path from every vertex to s and to every vertex from s is clearly a sufficient condition for the existence of a path from u to v, as one such path must exist from the combination of paths from u to s and from s to v. Therefore, G is strongly connected since there exists a path from any vertex u to any other vertex v.

Proving the necessity of the above statement is likewise straightforward. Simply let u = s, and then, by the definition of a strongly connected graph, if there is not a path from s to every other vertex and from every other vertex to s then G is not strongly connected. Therefore, the existence of a path s to and from every other vertex is necessary for G to be strongly connected.

Since the existence of a path from s to every other vertex and from every other vertex to s is both a necessary and sufficient condition of G being strongly connected, we can show that the provided statement is indeed true. ∎

2. Design a non-recursive Depth-First Search (DFS) algorithm by using a stack, and your algorithm should also run in O(m + n) time.

Given a graph G and a starting node S, we first color all of the nodes in G white and set their predecessors to null. Then, we set the next node to be looked at to the starting node and push it onto the stack.

With that initial setup complete, we will start a loop over the elements in the stack, first popping off the top element, changing its color to blue, inspecting its adjacency list and pushing any unvisited (white) nodes onto the stack and setting their predecessor to the node we just popped off the stack. Once we have completed searching the adjacency list for our node, we set its color to red and repeat this behavior with the next element on the stack.

After the loop has completed, we must check if there are any other nodes that are not descendants of S, and repeat the above process with them as well.

Pseudocode is as follows:

```
dfs(Graph G, Node S) {

  Stack toTraverse;

  for each vertex v of G {
                v.color = white;
                v.pre = NULL;
  }

  next = s;
  while (next.color == white) {
          toTraverse.push(next);

          while (toTraverse not empty) {
                  Node u = toTraverse.pop();
                  u.color = blue;
                  print "u";
                  for each vertex v in Adj[u]{
                        if (v.color == white) {
                                toTraverse.push(v);
                                v.pre = u;
                        }
                }
                u.color = red;
        }

        for each vertex u {
                if (u.color == white) {
                        next = u;
                        break;
                }
        }
    }
}
```

3. Design an O(m + n) time algorithm to compute the number of different shortest paths from s to t (your algorithm does not need to list all the paths, but only report the number of different paths).

Given the BFS algorithm provided in class, modify the algorithm such that for each node added to the queue, we also maintain the distance to the node. Then, once the first path to t is found, stop adding nodes into the queue and start a count of the number of times t has been reached. Once the queue has been emptied, all of the shortest paths to t will have been computed, and you simply need to return the number of times t was visited.

Pseudocode for the above is as follows:

```
int countShortestPaths(Graph G, Node S, Node T) {
  int numPaths = 0;
  int shortestDist = infinity;
  Queue Q;

  for each vertex v :
        v.color = white;
        v.dist = infinity;

  Q.enqueue(S);
  S.color = blue;
  S.dist = 0;
```

```
    while (Q not empty) {
        Node u = Q.dequeue();
        if (dist > shortestDist) {
            continue;
        }
        for each vertex v in Adj[u] :
            if (v == T) {
                if (shortestDist == infinity) {
                    shortestDist = v.dist;
                }
                numPaths++;
                continue;
            }
            if (v.color == white) {
                Q.enqueue(v);
                v.color = blue;
                v.dist = u.dist + 1;
            }
        u.color = red;
    }
    return numPaths;
}
```

4. Given a directed-acyclic-graph (DAG) G, design an O(m + n) time algorithm to compute the number of different paths in G from s to t.

First, perform a topological sort on G. We then know that all paths from s to t will travel through at least some of the nodes in the range from [s, t] in the sorted order (at the very least s and t themselves). Therefore, we can end our search for paths once we reach t in the topological order.

We will then create a running count of the number of paths from s to t, and initialize it to the number of outgoing edges from s, minus any paths that go past t in the topological order. We will then traverse down every edge from s in topological order, and for each node we encounter we add the number of outgoing nodes minus one to the number of paths to t, and subtract one for each edge that goes past t. We'll repeat this process until we reach t, at which point we can simply return the number of paths found so far.