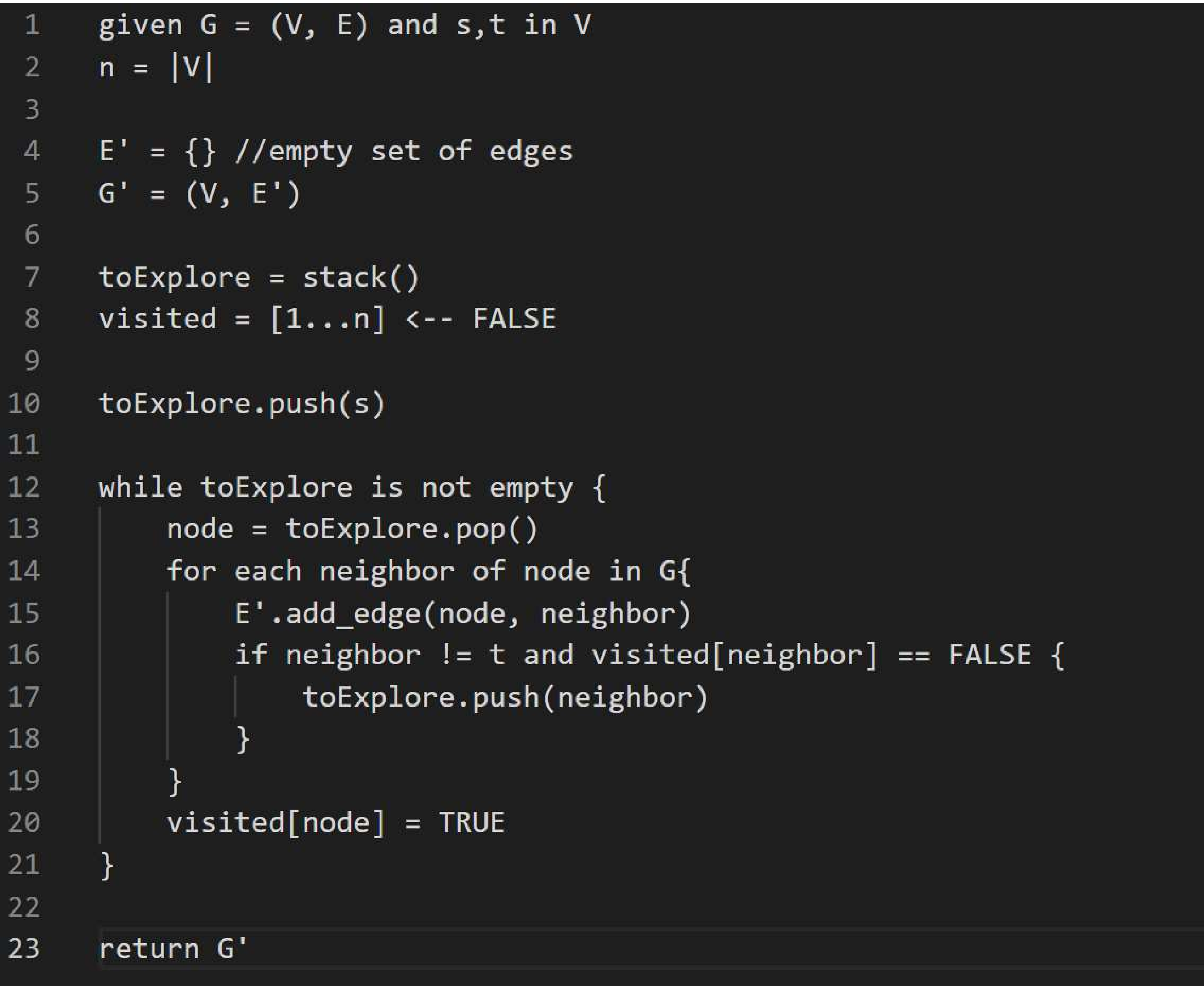
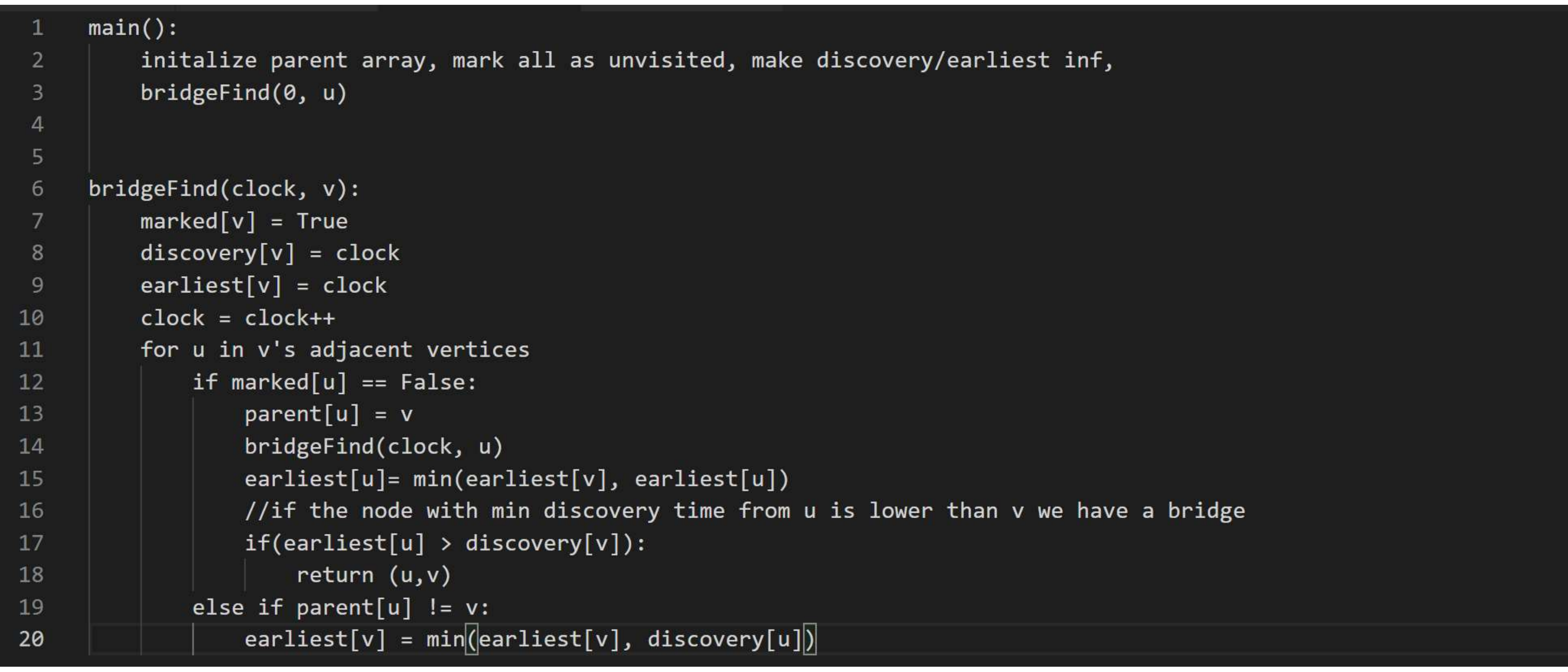
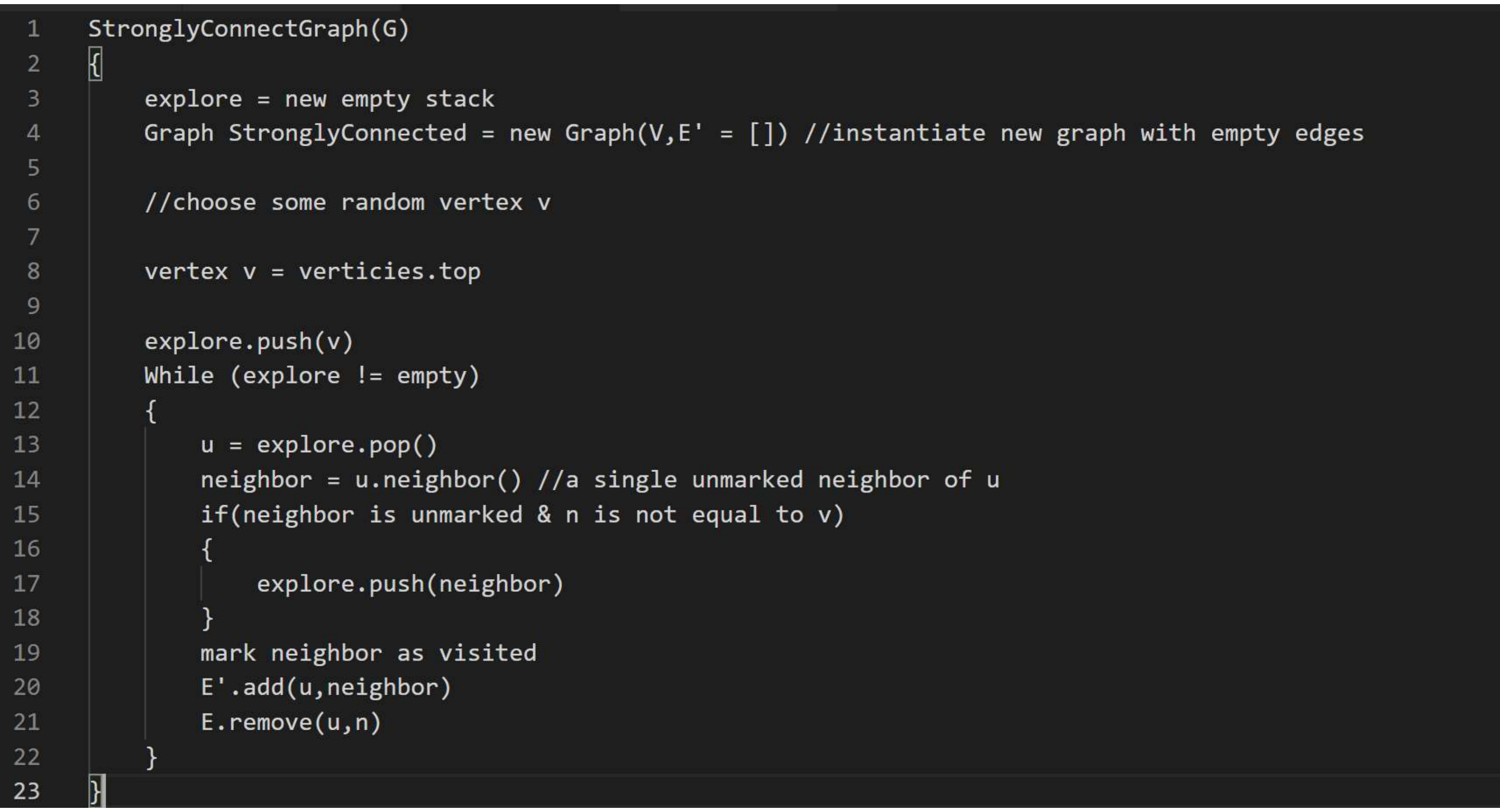

```
given G = (V, E) and s,t in V
n = |V|

E' = {} //empty set of edges
G' = (V, E')

toExplore = stack()
visited = [1...n] <-- FALSE

toExplore.push(s)

while toExplore is not empty {
    node = toExplore.pop()
    for each neighbor of node in G{
        E'.add_edge(node, neighbor)
        if neighbor != t and visited[neighbor] == FALSE {
            toExplore.push(neighbor)
        }
    }
    visited[node] = TRUE
}

return G'
```



```
main():
    initalize parent array, mark all as unvisited, make discovery/earliest inf,
    bridgeFind(0, u)


bridgeFind(clock, v):
    marked[v] = True
    discovery[v] = clock
    earliest[v] = clock
    clock = clock++
    for u in v's adjacent vertices
        if marked[u] == False:
            parent[u] = v
            bridgeFind(clock, u)
            earliest[u]= min(earliest[v], earliest[u])
            //if the node with min discovery time from u is lower than v we have a bridge
            if(earliest[u] > discovery[v]):
                return (u,v)
        else if parent[u] != v:
            earliest[v] = min(earliest[v], discovery[u])
```



```
StronglyConnectGraph(G)
{
    explore = new empty stack
    Graph StronglyConnected = new Graph(V,E' = []) //instantiate new graph with empty edges

    //choose some random vertex v

    vertex v = verticies.top

    explore.push(v)
    While (explore != empty)
    {
        u = explore.pop()
        neighbor = u.neighbor() //a single unmarked neighbor of u
        if(neighbor is unmarked & n is not equal to v)
        {
            explore.push(neighbor)
        }
        mark neighbor as visited
        E'.add(u,neighbor)
        E.remove(u,n)
    }
}
```

HW 8

Due on **Wednesday, March 27, 2019 at 10am**

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.33**

Worked with Jeremy Varghese.

22 (100 PTS.) Graph Morphing

Suppose we are given a DAG G with n vertices and m edges. The vertices represent jobs and whose edges represent precedence constraints. That is, each edge (u, v) indicates that job u must be completed before job v begins. Each node v also has a weight $T(v)$ indicating the time required to execute job v.

In the following, let G be a graph with n vertices and m edges.

22.A. (20 PTS.) Given an *undirected* graph G, and two vertices $s, t \in V(G)$, describe a linear time algorithm that directs each edge of G so that the resulting directed graph is a DAG, s is a source of this DAG, and t is a sink.

22.B. (10 PTS.) For a connected undirected graph G, a ***bridge*** is an edge that its removal disconnects the graph. Using **DFS** describe a linear time algorithm that decides if a bridge exists, and if so it outputs it.

22.C. (20 PTS.) Prove that for an *undirected* connected graph G with n vertices and m edges, the edges can be directed so that the resulting graph is strongly connected $\implies$ there are no bridges in G (the claim holds with $\iff$, but this is the easier direction)..

22.D. (50 PTS.) Given an undirected connected graph G that has no bridges, describe how to modify the **DFS** algorithm so that it outputs an orientation of the edges of G so that the resulting graph is strongly connected. Prove the correctness of your algorithm.

22

Solution:

1. **Idea:** We approached this part by using a modified version of DFS. We setup the problem by instantiating a stack by pushing our source vertex onto it. Next we mark every vertex as unvisited and instantiate an empty set of edges for our final DAG. We then call DFS on the source vertex s. for every vertex, within the DFS, after exploring it, we mark it as visited and make an edge from the current node to all neighbors. If it is T, we do not add it to the stack to be explored, otherwise we will also add it to the stack. The result of traversing this gives us a valid DAG from S to T.

   Psuedocode: See image 1.

   Analysis: Since the algorithm iterates through each node and doesn't visit each one more than once we satisfy the linear time requirement with $O(n)$.

2. **Idea:** The idea behind this algorithm is to use a modified DFS traversal. We first build DFS trees for the graph. We can identify a bridge in our DFS tree by two cases: 1. The edge comes out of the root of the DFS tree and the root has at least two children. 2. The vertex

v in the bridge edge (u,v) has no vertex beneath it such that there is a backedge to one of the ancestors of u.

We instantiate the setup with marking all vertices as unvisited. We then traverse and get DFS trees with discovery times of visited vertices by marking each vertex we visit with visit time and visited. For all neighbors of the current iteration, we now have DFS trees for them. Additionally, we store an array that stores discovery time of each vertex and a second array that gives us the earliest vertex we've visited for each node.

We then recur for all vertices that are adjacent to the current vertex passing in our current min discovery time. We calculate the new earliest vertex value and then go into our two cases. If the earliest vertex visited for v is below u in our tree then we know we have a bridge.
Psuedocode: see image 2

Analysis: This algorithm uses a modified DFS which traverses adjacent vertices while marking them. Since it only visits each node once and all operations within each recursive call is constant our final runtime is $O(m+n)$.

3. We will prove this by contradiction by assuming that there is a bridge in G. If there is a bridge in G, then by definition there exists a node such that if the incoming edge is removed, then the graph breaks into two components. By definition of strongly connected, every vertex is able to reach every other vertex through some path. If the condition is that the vertices of graph G are valid to have some strongly connected orientation of edges, then each vertex must at least have an incoming edge and an outgoing edge that is apart of a path that through an arbitrary number of iterations may reach any other node. Because a bridge requires the condition that one component has only one edge to another component, we have a contradiction as definition of strongly connected does not allow this.

4. **Idea:** We approached this part by using a modified version of DFS. We setup the problem by instantiating a stack by pushing a random source vertex onto it. Next we mark every vertex as unvisited and instantiate an empty set of edges for our final DAG. We then call DFS on the source vertex s. our initial source vertex is not marked as visited. for every vertex, within the DFS, after exploring it, we mark it as visited and make an edge from the current node to an unmarked neighbor. We also remove the node from the original graph's edgeset E therefore we will not revisit a neighbor we have already been to when choosing which neighbor to go to. If the neighbor to be explored is our original vertex v, we make the connection and we are done, otherwise we will continue to direct edges in the graph. The result of traversing this gives us a valid strongly connected DAG from S to T.

   To prove the correctness of this algorithm, we used the assumption that there were no bridges. We can refer back to the proof in part C and the given from the problem statement such that if there are no bridges a strongly connected edge orientation must exist. Because our algorithm makes use of the fact to create a cycle from any arbitrary node that connects all other nodes with a path back to the origin, we get a strongly connected graph. This would not work if there were bridges in the graph.

   Psuedocode: See image 3.

   Analysis: Since the algorithm iterates through each node and doesn't visit each one more than once we get a runtime of $O(n+m)$.