CS320: Assignment 3

Bryan Reilly

March 3, 2015

Problem 1

My solution finds the transitive closure of any graph represented by an adjacency matrix. I iterate through each node u, find what nodes are reachable from u using depth first search, then use that information to build a transitive closure.
This is the Python code:

```
import numpy

def transitiveClosure(array):
        #Initialize returned array
        closure = array
        #iterate through each node n
        for n in range(array.shape[1]):
                #dfs from n to find all reachable nodes from n
                reachable = DFS(n, array)
                #add all reachable nodes to n's edges
                for i in reachable:
                        closure[n][i] = 1
        return closure

def DFS(node, array):
        #Initialize return list
        final = []
        #Remeber initial node
        S = [node]
        #Initialize E to be a list representing the explored state of each node
        E = [0] * array.shape[1]
        #While S not empty
        while len(S) != 0:
                #Take a node u from S
                u = S.pop()
                #If explored[u] = false then
                if E[u] == 0:
                        #Set explored[u] true
                        E[u] = 1
                        #For each edge (u,v) incitent to u
                        for v in range(array.shape[1]):
                                if array[u][v] == 1:
                                        #Add v to the stack S
                                        final.append(v)
```

```
                                        S.append(v)
        #remove duplicates:
        final = list(set(final))
        #remove reference to self:
        if node in final: final.remove(node)
        return final

a = numpy.zeros([6,6])
#[from][to]
#[y][x]
a[0][1] = 1
a[0][3] = 1
a[0][4] = 1
a[1][2] = 1
a[3][5] = 1
a[4][0] = 1
a[5][4] = 1

print a
tc = transitiveClosure(a)
print "\n"
print tc
```

The running time of this code is $O(|N| * (|E| + |N|))$. We come to this via the following argument.
The outer loop of the *transitiveClosure* function will run $|N|$ times. Within that loop we call DFS, which is $O(|E|)$, on the current node. In the same loop we then update the adjacency graph, which happens in $O(|N|)$ time. Thus the inside of our loop runs in $O(|E| + |N|)$ time, and the outer loop runs $O(|N|)$ times. This gives us a final bound of $O(|N| * (|E| + |N|))$.

Problem 2

Since each node has a degree of at least 2, there are no leaf nodes. This means that it will be possible to find a cycle without the need to backtrack. By simply using a depth-first search and checking for run-overs we can find a cycle in $O(|V|)$ time.
Here is the algorithm:

```
def DFS(node, array):
        #Initialize return list
        final = []
        #Remeber initial node
        S = [node]
        #Initialize E to be a list representing the explored state of each node
        E = [0] * array.shape[1]
        #While S not empty
        while len(S) != 0:
                #Take a node u from S
                u = S.pop()
                #If explored[u] = false then
                if E[u] == 0:
                        #Set explored[u] true
                        E[u] = 1
                        #For each edge (u,v) incitent to u
```

```
                for v in range(array.shape[1]):
                        if array[u][v] == 1:
                                #Add v to the stack S
                                final.append(v)
                                S.append(v)
        #remove duplicates:
        final = list(set(final))
        #remove reference to self:
        if node in final: final.remove(node)
        return final
```

EXPLAIN WHY NOT O—N— time!!!!! Each edge "removes" a node

Problem 3

We may show a strongly connected graph also contains any two nodes (u, v) in a cycle via the following argument. Assume G is strongly connected, then there exists a directed path from u to v, and there exists a directed path from v to u. A cycle is defined a a subset of the edge set of G that forms a path such that the first node of the path corresponds to the last. Thus, we may take our path from u to v as the first part of our cycle, and the path from v to u as the second part of our cycle. We know both of these paths exist because the graph is strongly connected. Since we start at u and go through v, then come back to u, we have a cycle that contains u and v. Thus, if a graph is strongly connected, any two nodes are contained in a cycle.

Problem 4