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
# Graphs represented by numpy multidimensional arrays, ie:
a = numpy.zeros([6,6])
# Set the edges
# [from][to]
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

def transitiveClosure(array):
        finalClosureArray = array
        # Iterate through each node in the graph
        for node in range(array.shape[1]):
                reachable = DFS(node, array)
                # Add all reachable nodes to n's edges
                for reachedNode in reachable:
                        finalClosureArray[node][reachedNode] = 1
        return finalClosureArray

def DFS(node, array):
        final = []
        # Stack of nodes to visit
        visitStack = [node]
        # List representing explored state of each node
        explored = [0] * array.shape[1]
        while len(visitStack) != 0:
                boundaryNode = visitStack.pop()
                # If boundaryNode hasnt been explored
                if explored[boundaryNode] == 0:
                        explored[boundaryNode] = 1
```

```
                        # For each neighbor of boundaryNode
                        for possibleNeighbor in range(array.shape[1]):
                                if array[boundaryNode][possibleNeighbor] == 1:
                                        #Add neighbor to the visitStack
                                        final.append(possibleNeighbor)
                                        visitStack.append(possibleNeighbor)
        # Remove duplicates
        final = list(set(final))
        # Remove reference to self
        if node in final: final.remove(node)
        return final

# Example
print transitiveClosure(a)
```

The running time of this code is $O(|N| * (|E| + |N|))$. I've 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 $O(|E|)$ since at worst it will run through each edge in the graph. In the same loop we then update the adjacency graph, which happens in $O(|N|)$ time because at worst it must update $|N| - 1$ entries in the matrix. 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 (a leaf node is by definition a node of degree 1). This means that it will be possible to find a cycle without the need to backtrack (we can not hit a dead end). By using a depth-first search and checking for twice explored nodes, we can find a cycle in $O(|V|)$ time. The running time is $O(|V|)$ because every time we travel down an edge we "remove" a node from our remaining search pool of nodes.
Here is the algorithm:

```
import numpy
# Graphs represented by numpy multidimensional arrays, ie:
a = numpy.zeros([9,9])
# Set the edges
# [from][to]
a[0][1] = 1
a[0][2] = 1
a[1][0] = 1
a[1][2] = 1
a[2][0] = 1
a[2][1] = 1

def findCycle(array):
        exploredNodes = []
        # Starter node
        visitStack = [0]
        # Initialize exploredState to be a list representing the explored state of each node
        exploredState = [0] * array.shape[1]
        while len(visitStack) != 0:
                # Take a node boundaryNode from visitStack
                boundaryNode = visitStack.pop()
```

```
            # If explored[boundaryNode] = false then
            if exploredState[boundaryNode] == 0:
                    # Set explored[boundaryNode] true
                    exploredState[boundaryNode] = 1
                    exploredNodes.append(boundaryNode)
                    # For each edge (u,v) incitent to boundaryNode
                    for v in range(array.shape[1]):
                            if array[boundaryNode][v] == 1:
                                    # Add v to the stack visitStack
                                    visitStack.append(v)
            elif exploredState[boundaryNode] == 1 and boundaryNode != exploredNodes[len(exploredNode
                    # We have found a cycle
                    exploredNodes.append(boundaryNode)
                    # Remove non-cyclic portion
                    while exploredNodes[0] != boundaryNode: exploredNodes.remove(0)
                    return exploredNodes
    # No cycle found
    return []

# Example
print findCycle(a)
```

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

Adjacency Matrix

An adjacency matrix implementation allows us to check in $O(1)$ time whether or not a given edge exists in a graph. The graph must take $|V|^2$ space since we have $\Omega(|V|^2)$ edges. It takes $\Omega(|V|)$ time to check all incident edges of a node, even if there are less than $|V|$ incident edges.

Adjacency List

An adjacency list requires only $O(|E|+|V|)$ space since we need $|V|$ arrays, where the length of all the arrays is at most $O(|E|)$. Since we have $\Omega(|V|^2)$ edges, we will still require $O(|V|^2)$ space, and thus we save no space over an adjacency matrix representation. The time it takes to find edge e is proportional to the lowest degree'd node that e is attatched to. Once a node is found, its neighbors can be found in constant time per neighbor.

Preference

Since the graph has $\Omega(|V|^2)$ edges, both implementations will take the same amount of space. It will take $O(|V|)$ time to find all the neighbors of a node in both implementations. The adjacency matrix will be able to check if an edge (u, v) exists in the graph in $O(1)$ time, while the adjacency list does the same operation in proportion to the degree u and v. Thus, the adjacency matrix will be the more suitable representation for this type of graph, since it will have faster edge checking.