EECS-214 Worksheet 6

# Problem 1

Here’s a directed graph with nodes labeled with integers:



Write the class definitions for two different data structures that could be used for representing the graph (there are many, we just want you to do two different ones).

## Answers:

As we said, there are a bunch of ways of doing this, but here are two:

class GraphRepresentedWithNodeObjects {  
 class Node {  
 int weight;  
 List<Node> neighbors;  
 }  
 List<Node> nodes;  
}

class GraphRepresentedWithArrayOfAdjacencyLists {  
 // This assumes nodes are numbered with unique numbers starting at 0  
 public List<int> Weights; // Weights[i] is the weight of node i.  
 public List<List<int>> AdjacencyLists; // AdjacencyLists[i] is the adjacency list of node i.  
 // Note: the List data type in C# is a dynamic array, not a linked list.  
}

# Problem 2

Give an algorithm to print the shortest path between two nodes in an undirected graph. You may assume whatever graph representation you like.

## Answer:

We’ll assume the array of adjacency lists representation above.

**PrintPath**(graph, start, end) {  
 // Start with a breadth-first search  
 // We search from the end rather than the start because BFS computes the predecessors  
 // of nodes, not the successors. By searching backward, the predecessor links end up pointing  
 // in the direction of the forward path.  
 nodeCount = graph.AdjacencyLists.Count;  
 visited = new bool[nodeCount];  
 predecessor = new int[nodeCount];  
 q = new Queue();

visited[end] = true;  
 predecessor[end] = -1;  
 q.Enqueue(end);  
 while (q not empty) {  
 n = q.Dequeue();  
 foreach (neighbor in graph.AdjacencyLists[n])  
 if (!visited(neighbor)) {  
 q.Enqueue(neighbor);  
 visited[neighbor] = true;  
 predecessor[neighbor] = n;  
 if (neighbor == end)  
 break;  
 }  
  
 // Now print the path  
 for (current = start; current != -1; current = predecessor[current])  
 print current;  
}

Note that I didn’t bother having it compute distances here because we were just computing the path, not the distances.

# Problem 3

Suppose you have a DAG (directed acyclic graph) represented using the following data structure.

class DAG {  
 class Node {  
 List<Node> Neighbors;  
 }  
 List<Node> Nodes;  
}

Suppose you wanted to test whether two nodes in the DAG are connected. As you know, you could do this with DFS or BFS. Normally for a graph, you’d use a version of DFS or BFS that keeps track of visited nodes. Is that necessary for a DAG? That is, is it possible to write an algorithm that works and doesn’t keep visited information? If so, is there a cost of doing so?

## Answer

Yes to both. The visited information is usually needed for detecting cycles in the graph, since a naïve algorithm will go into an infinite loop or recursion when there’s a cycle. However, a DAG (Directed **Acyclic** Graph), by definition, doesn’t have cycles. So a version that doesn’t keep track of visited information won’t loop forever. That said, it can still search the same areas of the graph repeatedly; it just does so only a finite number of times (rather than infinite, if there’s a cycle). For example, in the graph:



If we search starting at node 1, then node 4 will get searched twice, because it’s a neighbor of both 2 and 3. But that means nodes 5 and 6 also get searched twice, and node 7 will get searched once for each time 5 or 6 is searched. So node 7 is searched 4 times. If this splitting-and-merging shape, making the graph longer, then you end up with a search that technically runs in finite time, but can end up requiring exponential time (in the length of the graph) to search.