Remi Galvez
CSci 3212 - Exercise IV
Monday, October 26, 2015

A. Given an adjacency-list representation of a directed graph, explain how you would (give high-level pseudocode):

- Compute the out-degrees of all vertices.

Given: - adjList → List<Edges>[Vtx]: array of lists where each array index corresponds to a vertex, and each list stores the directed edges leaving that vertex.

```
// iterate through vertices
for i in 0 : adjList.length                                   O(V)
    // compute number of edges leaving vertex i
    outDegree(i) = adjList(i).size                            O(1)
    // sum total number of degrees
    totalDegrees = totalDegrees + outDegree(i)                O(1)
end for

return [ outDegree, totalDegrees ]
```

Answer: $\boxed{O(V)}$

- Compute the in-degrees of all vertices.

Given: - adjList → List<Edges>[Vtx]: array of lists where each array index corresponds to a vertex, and each list stores the directed edges leaving that vertex.

```
// iterate through vertices
for i in 0 : adjList.length                                          O(V)
    // iterate through edges
    for e in adjList(i)                                              O(E(v))
        // increment count for end vertex (as the edge only will add an in-degree to the end vertex)
        inDegree(e.endVertex) = inDegree(e.endVertex) + 1            O(1)
        // sum total number of degrees
        totalDegrees = totalDegrees + outDegree(i)                   O(1)
    end for
end for
```

Essentially, we are iterating through each edge once and incrementing the count for each end vertex. Hence,
Answer: $\boxed{O(E)}$

C. Describe, in pseudocode, an algorithm to detect whether a given undirected graph is bipartite, assuming the graph is input as an adjacency matrix. Provide an order-notation analysis of the running time. A graph is called bipartite if its vertices (the set V) can be partitioned into two sets V_1 and V_2 such that:
(1) no vertex is in both V_1 and V_2;
(2) no edge has both its end vertices in V_1;
(3) no edge has both its end vertices in V_2. The graph below is an example:

Algorithm: detectBipartite(g)
Input: A graph's adjacency matrix

// Label will alternate between "0" and "1" so initialize to -1
for $i = 0$ **to** $n - 1$
    visitOrder[i] = -1
    label[i] = -1
end for

// Standard queue data structure.
Create queue;

// Look for an unvisited vertex and explore its tree.
// We need this because the graph may have multiple components.
for $i = 0$ **to** $n - 1$
    if visitOrder[i] < 0
        // We call this "iterative" because other searches are recursive.
        detectBipartiteIterative(i)
    end if
end for

Algorithm: detectBipartite(u)
Input:Vertex u, adjMatrix is assumed to be global.

    // Queue needs to be reset for each tree.
    Clear queue;

    //Place root of tree on the queue.
    queue.addToRear(u);
    label[u] = 1

    // Continue processing vertices until no more can be added.
    while queue **not** empty

        // Remove a vertex.
        v = remove item at front of queue;
        // If it hasn't been visited ...
        if visitOrder[v] < 0
            // Visite the vertex.

```
            visitCount = visitCount + 1
            visitOrder[v] = visitCount
            // Look for neighbors to visit.
            for i = 0 to n − 1
                // If two neighboring vertices have equal labels, there is an odd cycle
                if adjMatrix[v][i] = 1 and label[v] == label[i]
                    // Hence, the graph is not bipartite
                    return false
                end if

                if adjMatrix[v][i] = 1 and i != v        // Check self-loop: i != v
                    queue.addToRear(i)
                    // Assign opposite label to neighboring vertices
                    label[i] = −label[v]
                end if
            end for
        end if
    end while

    return true
```

Idea - Run breadth-first search, and assign label to every new node labels alternating between neighbors. If two neighbors are assigned the same label, the graph has an odd cycle, and therefore cannot be bipartite. Labelling the vertices essentially represents assigning them to a partition. If two neighbors are in the same partition, the graph is not bipartite.

D. Transaction problem, part 2. Recall the transaction problem from Exercise 3, in which we were given an array of prices. In this variation, we will allow multiple transactions, as many as one likes. The transactions must all have different buy dates, but can share common sell dates. Show how to use the ideas in part 1 (Exercise 3) along with a heap to determine the maximum possible profit, with the output listing the profitable transactions in profit order (most profitable, followed by next most profitable etc). What is the running time of this combined algorithm? Explain your analysis of the running time.

Such an algorithm can be broken down into 3 steps:

1. Instantiate nodes for all possible transactions $\frac{n\cdot(n-1)}{2} = O(n^2)$
2. Build heap $O(n^2)$ (As there can be up to $\frac{n\cdot(n-1)}{2}$ transactions)
3. Print elements $O(n^2)$ (Again n^2 for the same reason)

The time complexities are all worst case, as there would only be up to $\frac{n\cdot(n-1)}{2}$ transactions if the given array were sorted in increasing order.

Answer: $\boxed{O(n^2)}$