Problem Set 12 - Solution

Graphs: Topological Sorting, Traversal

---

1. You are given a directed graph:

```
class Neighbor {
   public int vertex;
   public Neighbor next;
   ...
}

class Vertex {
   String name;
   Neighbor neighbors; // adjacency linked lists for all vertices
}

public class Graph {
   Vertex[] vertices;

   // returns an array of indegrees of the vertices, i.e. return[i] is the
   // number of edges that are directed IN TO vertex i
   public int[] indegrees() {
     // FILL IN THIS METHOD
     ...
   }
   ...
}
```

   Assuming that the graph has already been read in, complete the `indegrees` method.

   SOLUTION

```
   public int[] indegrees() {
      int[] indeg = new int[vertices.length];
      for (int i=0; i < vertices.length; i++) {
          for (Neighbor nbr=vertices[i].neighbors; nbr != null; nbr=nbr.next) {
              indeg[nbr.vertex]++;
          }
      }
      return indeg;
   }
```

---

2. What is the big O running time of your `indegrees` implementation if the graph has *n* vertices and *e* edges? Show your analysis.

   SOLUTION

   - Accessing the front of a vertex's neighbors list, updating the indegree of a vertex, and accessing the neighbor of a vertex are each unit time operations.
   - There are *e* neighbors in all, for all vertices put together, so the neighbor access part contributes *e* units of time. Accessing the front of a vertex's neighbors list is done *n* times in all, once per vertex. There are *e* indegree updates, one per edge.
   - Total is $e + n + e = n + 2e$, which is $O(n+e)$

---

3. With the same `Graph` class as in the previous example, assuming that the graph is acyclic, and that that the `indegrees` method has been implemented, implement a `topsort` method to toplogically sort the vertices using **using BFS (breadth-first search)** (see algorithm in Section 14.4.4 of text):

```
public class Graph {
   ...
   public String[] indegrees() {
      ... // already implemented
   }

   // returns an array with the names of vertices in topological sequence
   public String[] topsort() {
     // FILL IN THIS METHOD
     ...
   }
   ...
}
```

   You may use the following `Queue` class:

```
public class Queue<T> {
   ...
   public Queue() {...}
   public void enqueue(T item) {...}
   public T dequeue() throws NoSuchElementException {...}
   public boolean isEmpty() {...}
   ...
}
```

   SOLUTION

```
   // returns an array with the names of vertices in topological sequence
   public String[] topsort()
   throws Exception {

      // compute indegrees
      int[] indeg = indegrees();

      int topnum=0;
      String[] tops = new String[vertices.length];
      Queue queue = new Queue();

      // find all vertices with indegree zero, assign them topological numbers, and enqueue
      for (int i=0; i < indeg.length; i++) {
          if (indeg[i] == 0) {
             tops[topnum++] = vertices[i].name;
             queue.enqueue(i);
          }
      }

      // loop until queue is empty
      while (!queue.isEmpty()) {
          int v = queue.dequeue();
          for (Neighbor nbr=vertices[v].neighbors; nbr != null; nbr=nbr.next) {
              indegrees[nbr.vertex]--;
              if (indegrees[nbr] == 0) {
                 tops[topnum++] = vertices[nbr.vertex].name;
                 queue.enqueue(nbr.vertex);
              }
          }
      }

      return tops;
   }
```

---

4. An undirected graph has `n` vertices and `e` edges, and is stored in adjacency linked lists. The edges DO NOT have weights. What would be the *fastest* algorithm (in the big O worst case sense) to find the shortest path from vertex numbered `x` to vertex numbered `y`, assuming `y` can be reached from `x`? Describe the algorithm, and state its big O worst case running time.

SOLUTION

Algo:
Do a BFS starting at `x`. Set distance of `x` to 0. When an edge `a--b` is seen and `b` is enqueued, make distance of `b` equal to distance of `a` plus 1. When `y` is reached, stop.

Worst case running time is O(n+e) since in the worst case, we would need to run BFS over the entire graph (i.e. `y` is the last vertex seen), and the running time of BFS is O(n+e).

---

5. A *strongly connected* directed graph is one in which every vertex can reach all other vertices. In the following `Graph` class, implement a method `stronglyConnected` that returns true if the graph is strongly connected, and false otherwise. What is the worst case big O running time of your implementation?

```
public class Graph {
   Vertex[] vertices;

   // performs a recursive dfs starting at vertex v
   private void dfs(int v, boolean[] visited) {
       // already implemented
     ...
   }

   public boolean stronglyConnected() {
       // FILL IN THIS IMPLEMENTATION
   }

   ...
}
```

SOLUTION

```
   public boolean stronglyConnected() {
      boolean[] visited = new boolean[vertices.length];
      for (int i=0; i < vertices.length; i++) {
          for (int j=0; j < visited.length; j++) {
              visited[i] = false;
          }

          dfs(i, visited);

          for (int j=0; j < visited.length; j++) {
              if (!visited[j]) {
                 return false;
              }
          }
      }
      return true;
   }
```

In the worst case, every vertex can reach all other vertices. The dfs method is called once for each vertex, and the time for a dfs run is O(*n*+*e*). So the total time is *n**O(*n*+*e*) = O(n^2+*ne*). (Note: since *e* can be anywhere between *0* and O(n^2), we cannot simplify the big O result any further.