Problem Set 12 - Solution

Graphs: Topological Sorting, Traversal, Dijkstra's Algorithm

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
     ...
   }
   ...
}
```

   1. 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($n+e$)

2. 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
```

```java
    public String[] topsort() {
      // FILL IN THIS METHOD
      ...
    }
    ...
  }
```

You may use the following Queue class:

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

SOLUTION

```java
    // 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.vertex] == 0) {
            tops[topnum++] = vertices[nbr.vertex].name;
            queue.enqueue(nbr.vertex);
          }
        }
      }

      return tops;
    }
```

3. 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?

```java
    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: The worst case would occur when the graph is strongly connected, otherwise the algorithm terminates earlier. And any strongly connected graph must have at least n edges. Which means that the term ne in the big O is at least n^2, so the other n^2 term can be discarded. Which means the big O can be simplified to *O(ne)*.

---

4. Suppose you are given this undirected graph in which the vertices are towns, and the edges are toll roads between them. The weight of an edge is the dollar amount of toll.
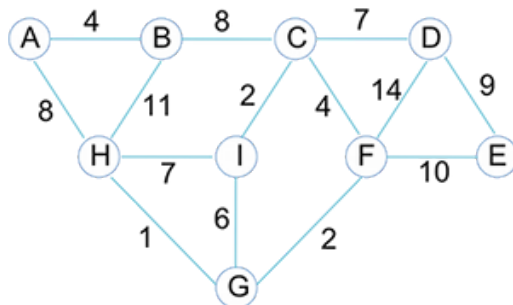
Use Dijsktra's shortest paths algorithm to determine the minimum toll route from A to all other cities.

o Show each step of the algorithm in tabular form. Here's the table after the initial step:

Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]	D[H]	D[I]
A	4,A	∞	∞	∞	∞	∞	8,A	∞

Note that along with the distance, the "previous" vertex is also shown.

SOLUTION

Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]	D[H]	D[I]
A	4,A	∞	∞	∞	∞	∞	8,A	∞
B		12,B	∞	∞	∞	∞	8,A	∞
H		12,B	∞	∞	∞	9,H		15,H
G		12,B	∞	∞	11,G			15,H

F	12,B	25,F	21,F	15,H
C		19,C	21,F	14,C
I		19,C	21,F	
D			21,F	
E				

Note that along with the distance, the "previous" vertex is also shown.

- Draw the shortest path tree induced on the graph. The shortest path tree is a general tree (a node can have any number of children) that consists of all the vertices of the graph, and the edges that are in the shortest paths to all the vertices. The root of the tree is the source vertex for the algorithm.

SOLUTION

The shortest path tree induced on the graph: