Problem Set 11 - Solution

Graphs: Representation, Traversal

---

1. Suppose a weighted undirected graph has *n* vertices and *e* edges. The weights are all integers. Assume that the space needed to store an integer is the same as the space needed to store an object reference, both equal to one unit. *What is the minimum value of e* for which the adjacency matrix representation would require less space than the adjacency linked lists representation? Ignore the space needed to store vertex labels.

   SOLUTION

   Space for adjacency matrix (AMAT) is *n^2*. Space for adjacency linked lists (ALL) is *n + 3*2e = n + 6e*. (Each node needs 3 units of space: 1 for the neighbor number, 1 for the edge weight, and 1 for the next node reference. And there are 2*e* nodes.) The space required by AMAT and ALL is the same when *n^2 = n + 6e*, i.e. when *e = (n^2 - n)/6*.

   The minimum value of *e* for which the adjacency matrix representation would require less space than the adjacency linked lists representation is one more than the *e* above, which would be *(n^2 - n)/6+1*.

---

2. The complement of an **undirected** graph, **G**, is a graph **GC** such that:
   - **GC** has the same set of vertices as **G**
   - For every edge *(i,j)* in **G**, there is no edge *(i,j)* in **GC**
   - For every pair of vertices *p* and *q* in **G** for which there is no edge *(p,q)*, there is an edge *(p,q)* in **GC**.

   Implement a method that would return the complement of the **undirected** graph on which this method is applied.

```
class Edge {
  int vnum;
  Edge next;
}

public class Graph {
  Edge[] adjlists;  // adjacency linked lists
  ...
  public Graph complement() {
    // FILL IN THIS METHOD
    ...
  }
}
```

   What would be the worst case running time (big O) of an implementation for a graph with *n* vertices and *e* edges?

SOLUTION

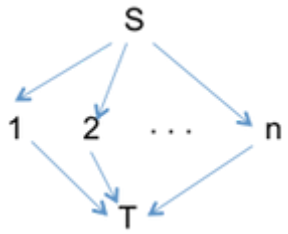
```
public Graph complement() {
  boolean[][] temp = new boolean[adjlists.length][adjlists.length];
  // in temp, fill in trues for the edges
  for (int v=0;v < adjlists.length;v++) {
    for (Edge e=adjlists[v];e != null;e = e.next) {
      temp[v][e.vnum] = true;
    }
  }
  // complement temp
  for (int i=0; i < adjlists.length; i++) {
     for (int j=0; j < adjlists.length; j++) {
        if (i != j) { // leave out the diagonal
          temp[i][j] = !temp[i][j];
        }
     }
  }
  // now create the adjacency linked lists for the complement graph
  Edge[] compall = new Edge[adjlists.length];
  for (int v=0;v < compall.length;v++) {
    for (int j=0; j < adjlists.length; j++) {
      if (temp[v][j]) {
        Edge e = new Edge();
        e.vnum = j;
        e.next = compall[v];
        compall[v] = e;
      }
    }
  }
  // create new Graph and return
  Graph comp = new Graph();
  comp.adjlists = compall;
  return comp;
}
```

Running time is *O(n^2)* - this is the time needed to compute the complement matrix. (A more abstract way of reasoning about this is to note that the original graph and its complement would involve all possible edges between the *n* vertices, which is O(n^2).)

---

3. **WORK OUT THE SOLUTION TO THIS PROBLEM AND TURN IT IN AT RECITATION**

Consider this graph:



This graph has *n+2* vertices and 2*n* edges. For every vertex labeled *i*, *1* <= *i* <= *n*, there is an edge from *S* to *i*, and an edge from *i* to *T*.

1. How many different depth-first search sequences are possible if the start vertex is *S*?
2. How many different breadth-first search sequences are possible if the start vertex is *S*?

SOLUTION

1. *n!*, for the different permutations of the vertices 1 through n. (Note: If a vertex v in this set is visited immediately after S, then T would be immediately visited after v.)

   For instance, say *n = 3*. Here are all possible DFS sequences (3! = 6):

   S 1 T 2 3
   S 1 T 3 2
   S 2 T 1 3
   S 2 T 3 1
   S 3 T 1 2
   S 3 T 2 1

2. *n!*, similar to DFS. The only difference is that T will be the last vertex to be visited. So, if *n = 3*, the possible BFS sequences are:

   S 1 2 3 T
   S 1 3 2 T
   S 2 1 3 T
   S 2 3 1 T
   S 3 1 2 T
   S 3 2 1 T

---

4. * You can use DFS to check if there is a path from one vertex to another in a directed graph.

   Implement the method **hasPath** in the following. Use additional class fields/helper methods as needed:

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

public class Graph {
  Neighbor[] adjLists;  // adjacency linked lists for all vertices

  // returns true if there is a path from v to w, false otherwise
  public boolean hasPath(int v, int w) {
    // FILL IN THIS METHOD
    ...
  }
}
```

SOLUTION

```
  public boolean hasPath(int v, int w) {
    if (v == w) return false;
    int n = adjlists.length;
    boolean[] visited = new boolean[n];
    for (int i=0; i < n; i++) {
       visited[i] = false;
    }
    return pathDFS(v,w,visited);
  }

  private boolean pathDFS(int current, int w, boolean[] visited) {
    if (current == w) return true;
    visited[current] = true;
    for (Neighbor ptr=adjlists[current]; ptr != null; ptr=ptr.next) {
       if (!visited[ptr.vertex]) {
         if (pathDFS(ptr.vertex, w, visited)) {
           return true;
         }
       }
    }
    return false;
  }
```

5. An *undirected* graph may be disconnected, if there are certain vertices that are unreachable from other vertices. In a disconnected graph, each island of vertices is called a *connected component* - in each island, every vertex can reach all other vertices.

   You are given the same Graph class as in problem #4, but this time it represents an undirected graph, and it does not have the hasPath method.

   Implement a method in this class that will use dfs to number all connected components (0..), and return an array that holds, for each vertex, the number of the connected component to which it belongs. Implement helper methods as necessary. What is the big O running time of your implementation?

```
public class Graph {

  Neighbor[] adjLists; // adjacency linked lists for all vertices

  // returns an array of connected component membership of vertices,
  // i.e. return[i] is the number of the connected number to which a vertex belongs
  // connected components are numbered 0,1,2,...
  public int[] connectedComponents() {
     // FILL IN THIS IMPLEMENTATION
  }

  ...
}
```

SOLUTION

```
  // recursive dfs modified to deal out component number
  private void dfs(int v, boolean[] visited, int[] compNums, int num) {
    visited[v] = true;
    compNums[v] = num;
    for (Neighbor nbr=adjLists[i]; nbr != null; nbr=nbr.next) {
      if (!visited[nbr.vertex]) {
        dfs(nbr.vertex, visited, compNums, num);
      }
    }
  }

  // returns an array of connected component membership of vertices,
```

```
  // i.e. return[i] is the number of the connected number to which a vertex belongs
  // connected components are numbered 0,1,2,...
  public int[] connectedComponents() {
    boolean[] visited = new boolean[adjLists.length];
    int[] compNums = new int[adjLists.length];

    for (int i=0,num=0; i < adjLists.length; i++,num++) {
      if (!visited[i]) {
        dfs(i, visited, compNums, num);
      }
    }
    return compNums;
  }
```

The only addition to the basic dfs implementation is the assignment of a component number to a vertex immediately after it is marked as visited. This adds O(n) to the original running time of O(n+e)), which does not make a difference. In the connectedComponents method, there is a scan of the visited array, which takes O(n) time. Again this does not change the big O time. So the total running time is still O(n+e), same as basic dfs.