Структури от Данни и Обектно-Ориентирано Програмиране
спец. Компютърни Науки
Упражнение №14
02.06.2009г.

ТЕМА: Графи

Задача 1: Да се напише клас на езика Java за представяне на ориентиран граф.

```
import java.util.Set;
import java.util.HashSet;

/**
 *  DiGraph class represents an directed graph of vertices named 0 through V-1.
 *  Parallel edges and self-loops are permitted.
 */
public class DiGraph {

        private final int numV;
        private int numE;
        private Set<Integer>[] adjacents;

        /**
         * Create an empty digraph with V vertices.
         */
        public DiGraph(int v) {
                if (v < 0)
                        throw new RuntimeException("Number of vertices
                                                        must be nonnegative");
                this.numV = v;
                this.numE = 0;
                adjacents = (Set<Integer>[]) new Set[numV];
                for (int i = 0; i < v; i++) {
                        adjacents[i] = new HashSet<Integer>();
                }
        }

        /**
         * Return the number of vertices in the digraph.
         */
        public int numberOfVerteces() {
                return numV;
        }
```

```
    /**
     * Return the number of edges in the digraph.
     */
    public int numberOfEdges() {
        return numE;
    }

    /**
     * Add the directed edge v-w to the digraph.
     */
    public void addEdge(int v, int w) {
        adjacents[v].add(w);
        numE++;
    }

    /**
     * Return the list of neighbors of vertex v as in Iterable.
     */
    public Iterable<Integer> adjacents(int v) {
        return adjacents[v];
    }

    /**
     * Return a string representation of the digraph.
     */
    public String toString() {
        StringBuilder s = new StringBuilder();
        //String NEWLINE = System.getProperty("line.separator");
        s.append(numV + " " + numE + "\n");
        for (int i = 0; i < numV; i++) {
            s.append(i + ": ");
            for (int w : adjacents[i]) {
                s.append(w + " ");
            }
            s.append("\n");
        }
        return s.toString();
    }

}
```

```
/**
 *  Determine single-source or multiple-source reachability in a digraph
 *  using depth first search.
 */
public class DiGraphDFSearch {
        private boolean[] visited;

        // single-source reachability
        public DiGraphDFSearch(DiGraph G, int s) {
                visited = new boolean[G.numberOfVerteces()];
                dfs(G, s);
        }

        // multiple-source reachability
        public DiGraphDFSearch(DiGraph G, Iterable<Integer> sources) {
                visited = new boolean[G.numberOfVerteces()];
                for (int v : sources)
                        dfs(G, v);
        }

        private void dfs(DiGraph G, int v) {
                visited[v] = true;
                for (int w : G.adjacents(v)) {
                        if (!visited[w]) dfs(G, w);
                }
        }

        // is there a directed path from the source (or sources) to v?
        public boolean visited(int v) {
                return visited[v];
        }

}



import java.util.Stack;

/**
 *  Determine reachability in a digraph from a given vertex using
 *  depth first search.
 */
public class DiGraphDFSPath {
   private boolean[] visited;  // visited[v] = true if v is reachable from s
   private int[] edgeTo;      // edgeTo[v] = last edge on path from s to v
   private final int s;       // source vertex
```

```
   public DiGraphDFSPath(DiGraph G, int s) {
      visited = new boolean[G.numberOfVerteces()];
      edgeTo = new int[G.numberOfVerteces()];
      this.s = s;
      dfs(G, s);
   }

   private void dfs(DiGraph G, int v) {
      visited[v] = true;
      for (int w : G.adjacents(v)) {
         if (!visited[w]) {
            edgeTo[w] = v;
            dfs(G, w);
         }
      }
   }

   // is there a directed path from s to v?
   public boolean hasPathTo(int v) {
      return visited[v];
   }

   // return path from s to v; null if no such path
   public Iterable<Integer> pathTo(int v) {
      if (!hasPathTo(v)) return null;
      Stack<Integer> stack = new Stack<Integer>();
      for (int x = v; x != s; x = edgeTo[x])
         stack.push(x);
      stack.push(s);
      return stack;
   }

}

import java.util.Queue;
import java.util.LinkedList;
import java.util.Stack;

/*************************************************************************
 *  Run breadth first search on a digraph.
 */
public class DiGraphBFSPath {
   private static final int INFINITY = Integer.MAX_VALUE;
   private boolean[] visited;  // visited[v] = is there an s-v path?
   private int[] distTo;      // distTo[v] = length of shortest s-v path
   private int[] edgeTo;      // edgeTo[v] = last edge on shortest s-v path
   private final int s;       // the source
```

```
   public DiGraphBFSPath(DiGraph G, int s) {
      this.s = s;
      visited = new boolean[G.numberOfVerteces()];
      distTo = new int[G.numberOfVerteces()];
      edgeTo = new int[G.numberOfVerteces()];
      for (int v = 0; v < G.numberOfVerteces(); v++) distTo[v] = INFINITY;
      distTo[s] = 0;
      visited[s] = true;
      bfs(G, s);
   }

   private void bfs(DiGraph G, int s) {
      Queue<Integer> q = new LinkedList<Integer>();
      q.add(s);
      while (!q.isEmpty()) {
         int v = q.poll();
         for (int w : G.adjacents(v)) {
            if (!visited[w]) {
               q.add(w);
               edgeTo[w] = v;
               distTo[w] = distTo[v] + 1;
               visited[w] = true;
            }
         }
      }
   }

   // length of shortest path from s to v
   public int distanceTo(int v) {
      return distTo[v];
   }

   // is there a directed path from s to v?
   public boolean hasPathTo(int v) {
      return visited[v];
   }

   // return shortest path from s to v; null if no such path
   public Iterable<Integer> pathTo(int v) {
      if (!hasPathTo(v)) return null;
      Stack<Integer> stack = new Stack<Integer>();
      for (int x = v; x != s; x = edgeTo[x])
         stack.push(x);
      stack.push(s);
      return stack;
   }

}
```

Задача 2: Да се напише клас на езика Java за представяне на неориентиран граф.

Задача 3: Да се напише клас на езика Java за представяне на ориентиран граф с тегла на ребрата..