21/04/2017

Report Tutorial Course 4

Algorithmics and Advanced Programming

WU Zheng

Report Tutorial Course 4

Algorithmics and Advanced Programming

Weighted Directed Graph

To continue with the following steps, I created a class named *WDGraph* to represent weighted directed graph. We will consider the adjacency list representation for unweighted and weighted digraphs. For all unweighted directed graph, the weight of each edge will be 1.

Create a class called *DirectedEdge* containing three attributes as follows:

```
public class DirectedEdge {

    private final int v;
    private final int w;
    private final double weight;

    DirectedEdge(int from, int to, double power) {
        v = from;
        w = to;
        weight = power;
    }

    public int from() {
        return v;
    }

    public int to() {
        return w;
    }

    public double weight() {
        return weight;
    }

}
```

Where v represents the source vertex, w the destination vertex and weight the edge-weight. The functions **from**(), **to**() and **weight**() constitute the getter functions for the source node, the destination node and the edge-weight.

Create a class called *WDgraph* to represent weighted-digraphs. This class will use an adjacency list representation of the graph, where the entry corresponding to vertex v will contain the list of all the outgoing arcs and the associated weights. In other words, that will be a list of objects of type *DirectedEdge*.

```
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

// This class represents a directed graph using adjacency list
```

```
// representation
class WDGraph {

    public int V;   // No. of vertices

    // Array of lists for Adjacency List Representation
    public LinkedList<DirectedEdge> adj[];

    // Constructor
    WDGraph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    // Constructor: build graph from file
    WDGraph(String filePath) throws IOException {
        List<String> lines = Files.readAllLines(Paths.get(filePath),
                StandardCharsets.UTF_8);
        HashSet<Integer> set = new HashSet<Integer>();
        for (String line : lines) {
            String[] nodesId = line.trim().split("\\s+");
            if (nodesId.length >= 2) {
                set.add(Integer.parseInt(nodesId[0]));
                set.add(Integer.parseInt(nodesId[1]));
            }
        }
        ArrayList<Integer> nodeIds = new ArrayList<Integer>(set);
        V = Collections.max(nodeIds) + 1;
        adj = new LinkedList[V];
        for (int i = 0; i < V; ++i)
            adj[i] = new LinkedList();
        for (String line : lines) {
            String[] nodesId = line.split("\\s+");
            if (nodesId.length == 2) {
                addEdge(Integer.parseInt(nodesId[0]), Integer.parseInt(nodesId[1]),
1);
            } else if (nodesId.length == 3) {
                addEdge(Integer.parseInt(nodesId[0]), Integer.parseInt(nodesId[1]),
Double.parseDouble(nodesId[2]));
            }
        }
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w, double power) {
        adj[v].add(new DirectedEdge(v, w, power));  // Add w to v's list.
    }

    // Print the whole graph
    void print() {
        for (int i = 0; i < V; i++) {
            System.out.print(i + ": ");
            for (DirectedEdge n : adj[i]) {
                System.out.print("(" + Integer.toString(n.to()) + ", " +
Double.toString(n.weight()) + "), ");
            }
            System.out.println();
        }
    }
```

```
}
```

The *WDGraph* can be read from a formatted file like:

```
1 2 9
1 6 14
1 7 15
```

The file will be interpreted as follows:

- The first line indicates that there is an arc from vertex 1 to vertex 2 and its weight is 9.
- The second line implies that there is an arc from vertex 1 to vertex 6 and its weight is 14.
- ... and so on.

The Depth Search First algorithm (DFS)

Input: A graph G and a vertex v of G.
Output: All vertices reachable from v labeled as discovered.

A recursive implementation of DFS

```
procedure DFS(G,v):
    label v as discovered
    for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

A non-recursive implementation of DFS

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Here are the tasks:

1. Given a graph, create a function called dfs(*Graph* G) that performs the deep first search (DFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.

```
import java.util.ArrayList;
import java.util.ListIterator;
import java.util.Stack;

/**
 * Created by Zheng on 21/04/2017.
 *
 */
public class DFSSearch {

    // The function to do DFS traversal.
    public ArrayList<Integer> dfs(WDGraph G, int startNode) {
        // Mark all the vertices as not visited (set as false by default in java).
        boolean[] visited = new boolean[G.V + 1];
        // Use an array list to record visit orders.
        ArrayList<Integer> visitOrder = new ArrayList<>();
        // Perform search algorithm without recursive calls.
        // DFS uses Stack data structure.
        Stack<Integer> stack = new Stack<>();
        // Put root node into stack.
        stack.push(startNode);
        while (!stack.isEmpty()) {
            int node = stack.pop();
            visited[node] = true;
            visitOrder.add(node);
            // In case of choice, the vertex with the smallest identifier will be
chosen.
            ListIterator<WDGraph.DirectedEdge> iter =
G.adj[node].listIterator(G.adj[node].size());
            while (iter.hasPrevious()) {
                WDGraph.DirectedEdge childNode = iter.previous();
                if (!visited[childNode.to()]) {
                    stack.push(childNode.to());
                }
            }
        }
        return visitOrder;
    }

}
```

2. One important application of the depth first search algorithm is to find the connected components of a graph. Write a function cc(*Graph* G) that takes as input a simple graph and determines the number of connected components. The function CC(*Graph* G) must use the DFS algorithm. Write a function called isConnected() that returns true if the graph is connected, false otherwise.

```
import java.util.*;

public class SearchAlgorithm {

    interface SearchCallbackInterface {
        ArrayList<Integer> search(WDGraph G, int startNode);
    }
```

```
    // The function to determine the number of connected vertex.
    private static int cc(WDGraph G, int startNode, SearchCallbackInterface
callback) {
        return callback.search(G, startNode).size();
    }

    // To determine whether the graph is a connected graph.
    private static boolean isConnected(WDGraph G, int startNode,
SearchCallbackInterface callback) {
        return cc(G, startNode, callback) == G.V;
    }

}
```

3. Test the dfs(.) and cc(.) functions with the graph is the graph-DFS-BFS.txt file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

```
public static void main(String[] args) {
    try {

        // We assume that the vertex id should be consecutive from 0 to
max(nodeIds).
        // If some vertex ids are missing, we consider these vertex are isolated.

        // Test DFS and BFS
        WDGraph g = new WDGraph("graph-DFS-BFS.txt");
        g.print();

        testDFS(g);

    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static void testDFS(WDGraph g) {
    // Test DFS
    SearchCallbackInterface dfsCallback = (G, startNode) -> {
        DFSSearch dfsSearch = new DFSSearch();
        return dfsSearch.dfs(G, startNode);
    };
    System.out.println(new DFSSearch().dfs(g, 1));
    System.out.println(cc(g, 1, dfsCallback));
    System.out.println(isConnected(g, 1, dfsCallback));
    System.out.println(new DFSSearch().dfs(g, 5));
    System.out.println(cc(g, 5, dfsCallback));
    System.out.println(isConnected(g, 5, dfsCallback));
}
```

The result of this test is:

```
0:
1: (2, 1.0),
2: (3, 1.0), (5, 1.0),
3: (4, 1.0),
4:
```

```
5: (6, 1.0),
6: (7, 1.0),
7:
[1, 2, 3, 4, 5, 6, 7]
7
false
[5, 6, 7]
3
false
```

If we start from the node 5, the order of the first encounter of the nodes is [5, 6, 7], the graph have 3 connected components. It is not connected.

Breadth Search First algorithm (BFS)

Input: A graph G and a starting vertex root v of G.
Output: Goal state. The parent links trace the shortest path back to root.

A non-recursive implementation of BFS

```
procedure BFS-iterative(G,v):
    let Q be a queue
    let D be a queue
    Q.enqueue(v)
    D.enqueue(0)
    mark(v)
    while Q is not empty
        t = Q.dequeue()
        td = D.dequeue()
         for all edges e from m to n in G.adjacentEdges(t) do
             if n is not marked:
                 mark(n)
                 Q.enqueue(n)
                 D.enqueue(td + 1)
```

1. Given a graph, create a function called bfs(*Graph* G) that performs the breadth first search (BFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.

```java
import java.util.*;

public class BFSShortestPaths {

    private int sourceNode;
    private boolean[] marked;
    private int[] previous;
```

```java
    private int[] distance;

    // The function to do BFS traversal.
    public ArrayList<Integer> bfs(WDGraph G, int s) {
        sourceNode = s;
        int v = G.V + 1;
        marked = new boolean[v];
        previous = new int[v];
        distance = new int[v];
        for (int i = 0; i < v; i++) {
            previous[i] = -1; // UNDEFINED
            distance[i] = Integer.MAX_VALUE; // +INFINITY
        }
        // Use an array list to record visit orders.
        ArrayList<Integer> visitOrder = new ArrayList<>();
        // Perform search algorithm without recursive calls.
        // BFS uses Queue data structure.
        Queue<Integer> queue = new LinkedList<>();
        Queue<Integer> distanceQueue = new LinkedList<>();
        // Put root node into queue
        // Put distance into queue (correspond to node)
        queue.add(s);
        marked[s] = true;
        previous[s] = -1;
        distanceQueue.add(0);
        distance[s] = 0;
        while (!queue.isEmpty()) {
            int node = queue.remove();
            int nodeDistance = distanceQueue.remove();
            visitOrder.add(node);
            // In case of choice, the vertex with the smallest identifier will be
chosen.
            for (WDGraph.DirectedEdge childEdge : G.adj[node]) {
                int thisNode = childEdge.to();
                if (!marked[thisNode]) {
                    queue.add(thisNode);
                    // Mark child node
                    marked[thisNode] = true;
                    previous[thisNode] = node;
                    // Update distance
                    distanceQueue.add(nodeDistance + 1);
                    distance[thisNode] = nodeDistance + 1;
                }
            }
        }
        return visitOrder;
    }

}
```

2. Test the bfs(.)function with the graph is the graph-DFS-BFS.txt file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

```java
private static void testBFS(WDGraph g) {
    // Test BFS
    SearchCallbackInterface bfsCallback = (G, startNode) -> {
        BFSShortestPaths bfsSearch = new BFSShortestPaths();
        return bfsSearch.bfs(G, startNode);
    };
    System.out.println(new BFSShortestPaths().bfs(g, 1));
```

```
    System.out.println(cc(g, 1, bfsCallback));
    System.out.println(isConnected(g, 1, bfsCallback));
    System.out.println(new BFSShortestPaths().bfs(g, 5));
    System.out.println(cc(g, 5, bfsCallback));
    System.out.println(isConnected(g, 5, bfsCallback));
}
```

The result of this test is:

```
[1, 2, 3, 5, 4, 6, 7]
7
false
[5, 6, 7]
3
false
```

If we start from the node 5, the order of the first encounter of the nodes is [5, 6, 7], the graph have 3 connected components. It is not connected.

Breadth Search First (BFS) for shortest paths in unweighted (di)graphs

Create a class called BFSShortestPaths. This class will implement the BFS algorithm for shortest paths from a given vertex s (seen in lecture 4). This class will contain:

1. 3 vertex-indexed arrays: **boolean**[] marked, **int**[] previous and **int**[] distance. marked[v] is set to true if v has been visited (false otherwise). previous[v] indicated the vertex preceding v on the shortest path. Distance[v] represents the distance (in number of edges) from the source vertex s to vertex v.
2. Modify the function bfs(*Graph* G) called bfs(*Digraph* G, **int** s) which executes the BFS algorithm to calculate all the shortest paths from the root vertex s. This function will update the marked, previous and distance arrays. It will be of void type.
3. Add a boolean function hasPathTo(**int** v) which returns true if there is a path from s to v.
4. Add the function distTo(**int** v) which returns the length of the shortest path from s to v.
5. Add the function printSP(**int** v) which prints the shortest path from s to v.
6. Test the previous functions with the graph graph-BFS-SP.txt. Find all the shortest paths and deduce the excentricity of each vertex, the diameter and the radius of the graph.

```
import java.util.*;

public class BFSShortestPaths {

    private int sourceNode;
    private boolean[] marked;
    private int[] previous;
    private int[] distance;

    // The function to do BFS traversal.
    public ArrayList<Integer> bfs(WDGraph G, int s) {
        sourceNode = s;
        int v = G.V + 1;
```

```
        marked = new boolean[v];
        previous = new int[v];
        distance = new int[v];
        for (int i = 0; i < v; i++) {
            previous[i] = -1; // UNDEFINED
            distance[i] = Integer.MAX_VALUE; // +INFINITY
        }
        // Use an array list to record visit orders.
        ArrayList<Integer> visitOrder = new ArrayList<>();
        // Perform search algorithm without recursive calls.
        // BFS uses Queue data structure.
        Queue<Integer> queue = new LinkedList<>();
        Queue<Integer> distanceQueue = new LinkedList<>();
        // Put root node into queue
        // Put distance into queue (correspond to node)
        queue.add(s);
        marked[s] = true;
        previous[s] = -1;
        distanceQueue.add(0);
        distance[s] = 0;
        while (!queue.isEmpty()) {
            int node = queue.remove();
            int nodeDistance = distanceQueue.remove();
            visitOrder.add(node);
            // In case of choice, the vertex with the smallest identifier will be
chosen.
            for (WDGraph.DirectedEdge childEdge : G.adj[node]) {
                int thisNode = childEdge.to();
                if (!marked[thisNode]) {
                    queue.add(thisNode);
                    // Mark child node
                    marked[thisNode] = true;
                    previous[thisNode] = node;
                    // Update distance
                    distanceQueue.add(nodeDistance + 1);
                    distance[thisNode] = nodeDistance + 1;
                }
            }
        }
        return visitOrder;
    }

    public boolean hasPathTo(int v) {
        return marked[v];
    }

    public int distTo(int v) {
        return distance[v];
    }

    public void printSP(int v) {
        ArrayList<Integer> shortestPath = new ArrayList<>();
        int thisNode = v;
        while (thisNode > -1) {
```

```
            shortestPath.add(thisNode);
            thisNode = previous[thisNode];
            if (thisNode == sourceNode) {
                shortestPath.add(sourceNode);
                break;
            }
        }
        Collections.reverse(shortestPath);
        System.out.println(shortestPath);
    }

}
```

The result of this test is:

```
0: (1, 1.0), (3, 1.0),
1: (2, 1.0),
2: (3, 1.0), (4, 1.0),
3:
4: (5, 1.0), (7, 1.0),
5: (6, 1.0), (7, 1.0),
6:
7:
true
4
[0, 1, 2, 4, 7]
```

Dijkstra algorithm for weighted digraphs

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
```

```
          u ← vertex in Q with min dist[u]    // Node with the least distance will be selected first
          remove u from Q

          for each neighbor v of u:           // where v is still in Q.
              alt ← dist[u] + length(u, v)
              if alt < dist[v]:               // A shorter path to v has been found
                  dist[v] ← alt
                  prev[v] ← u

      return dist[], prev[]
```

Create a class called DijkstraSP. This class will implement the Dijkstra algorithm for detecting shortest paths in weighted-digraphs. This class will contain the following functions:

1. 3 arrays: **boolean**[] marked, **int**[] previous and **int**[] distance as for the unweighted graphs.
2. A function called verifyNonNegative(*WDGraph* G) which takes as input a weidhted-directed graph and verifies than all weights in the graph are non negative.
3. Create a function called DijkstraSP(*WDgraph* G, **int** s)which implements the Dijkstra algorithm for shortest paths studied in lecture 4. The input arguments are a weighted-digraph and a root vertex s.
4. As for the previous section, create the functions hasPathTo(**int** v), distTo(**int** v) and printSP(**int** v).

```java
import java.util.*;

public class DijkstraSP {

    private int sourceNode;
    private boolean[] marked;
    private int[] previous;
    private double[] distance;

    private boolean verifyNonNegative(WDGraph G) {
        for (LinkedList<WDGraph.DirectedEdge> edges: G.adj) {
            for (WDGraph.DirectedEdge n : edges) {
                if (n.weight() <= 0) {
                    return false;
                }
            }
        }
        return true;
    }

    public ArrayList<Integer> DijkstraSP(WDGraph G, int s) {
        // To ensure all weights of edges are positive.
        if (!verifyNonNegative(G)) {
```

```
            return null;
        }
        sourceNode = s;
        int v = G.V + 1;
        marked = new boolean[v];
        previous = new int[v];
        distance = new double[v];
        // Use an array to store unvisited nodes
        HashSet<Integer> openedNodes = new HashSet<>();
        // Open all nodes
        for (int i = 0; i < v; i++) {
            previous[i] = -1; // UNDEFINED
            distance[i] = Double.MAX_VALUE; // +INFINITY
            openedNodes.add(i);
        }
        // Use an array list to record visit orders.
        ArrayList<Integer> visitOrder = new ArrayList<>();
        // Distance from source to source
        distance[s] = 0; // distance
        marked[s] = true; // mark
        visitOrder.add(s); // visit
        while (!openedNodes.isEmpty()) {
            // Choose the smallest distance.
            double smallestDistance = Double.MAX_VALUE;
            int smallestNode = -1;
            for (Integer thisNode : openedNodes) {
                if (distance[thisNode] < smallestDistance) {
                    smallestDistance = distance[thisNode];
                    smallestNode = thisNode;
                }
            }
            // Go to the smallest one.
            openedNodes.remove(smallestNode);
            visitOrder.add(smallestNode);
            // If remained nodes are not available, it is not a connected graph,
terminate the progress.
            if (smallestNode == -1) {
                break;
            }
            // Check all neighbours and update distances
            for (WDGraph.DirectedEdge directedEdge : G.adj[smallestNode]) {
                int childNode = directedEdge.to();
                double alt = distance[smallestNode] + directedEdge.weight();
                if (alt < distance[childNode]) {
                    marked[childNode] = true;
                    previous[childNode] = smallestNode;
                    distance[childNode] = alt;
                }
            }
        }
        return visitOrder;
    }

    public boolean hasPathTo(int v) {
        return marked[v];
    }

    public double distTo(int v) {
        return distance[v];
    }

    public void printSP(int v) {
```

```
        ArrayList<Integer> shortestPath = new ArrayList<>();
        int thisNode = v;
        while (thisNode > -1) {
            shortestPath.add(thisNode);
            thisNode = previous[thisNode];
            if (thisNode == sourceNode) {
                shortestPath.add(sourceNode);
                break;
            }
        }
        Collections.reverse(shortestPath);
        System.out.println(shortestPath);
    }

}
```

Test the previous functions with the graph graph-WDG.txt. The result of this test is:

```
0:
1: (2, 9.0), (6, 14.0), (7, 15.0),
2: (3, 24.0),
3: (5, 2.0), (8, 19.0),
4: (3, 6.0), (8, 6.0),
5: (4, 11.0), (8, 16.0),
6: (3, 18.0), (5, 30.0), (7, 5.0),
7: (5, 20.0), (8, 44.0),
8:
true
50.0
[1, 6, 3, 5, 8]
```