

21/04/2017

---

---

# Report Tutorial Course 4 (Part A)

*Algorithmics and Advanced Programming*

---

---

---

WU Zheng

# Report Tutorial Course 4 (Part A)

## *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

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

```

### A non-recursive implementation of DFS

```
1 procedure DFS-iterative( $G, v$ ):
2     let  $S$  be a stack
3      $S$ .push( $v$ )
4     while  $S$  is not empty
5          $v = S$ .pop()
6         if  $v$  is not labeled as discovered:
7             label  $v$  as discovered
8             for all edges from  $v$  to  $w$  in  $G$ .adjacentEdges( $v$ ) do
9                  $S$ .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

```
1  procedure BFS-iterative( $G, v$ ):
2      let  $Q$  be a queue
3      let  $D$  be a queue
4       $Q.enqueue(v)$ 
5       $D.enqueue(0)$ 
6      mark( $v$ )
7      while  $Q$  is not empty
8           $t = Q.dequeue()$ 
9           $td = D.dequeue()$ 
10         for all edges  $e$  from  $m$  to  $n$  in  $G.adjacentEdges(t)$  do
11             if  $n$  is not marked:
12                 mark( $n$ )
13                  $Q.enqueue(n)$ 
14                  $D.enqueue(td + 1)$ 
```

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.

```
import java.util.*;
```

```
public class BFSShortestPaths {
```

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

```
6 •
```

---

```

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?

```

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:

- 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  $v$ .
- 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.
- Add a boolean function `hasPathTo(int v)` which returns true if there is a path from  $s$  to  $v$ .
- Add the function `distTo(int v)` which returns the length of the shortest path from  $s$  to  $v$ .
- Add the function `printSP(int v)` which prints the shortest path from  $s$  to  $v$ .
- 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;
```

```
8 •
```

---

```

    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]
```