23/03/2017

Report Tutorial Course 3

Algorithmics and Advanced Programming

WU Zheng

Report Tutorial Course 3

Algorithmics and Advanced Programming

2.1 Adjacency list representation of an undirected graph

Add the following functions to the class graph:

1. create a class node and a class edge. Then each entry of the adjacency list becomes a list of objects of class node (instead of a list of integer numbers).

```
private class Edge {
    int ivex;
    Edge nextEdge;
}
private class Node {
    String id;
    Edge firstEdge;
};
private Node[] adj;
```

2. Initialize an empty graph of order N (input parameter) and 0 edges.

```
public Graph(int capacity) {
    adj = new Node[capacity];
    for (int i = 0; i < capacity; i++) {
        adj[i] = new Node();
        adj[i].id = Integer.toString(i);
        adj[i].firstEdge = null;
    }
}
```

3. Initializes a graph from a specified input stream. You will test this function by reading the graph from the graph.txt file.

```
public Graph(String filePath) throws IOException {
    List<String> lines = Files.readAllLines(Paths.get(filePath),
            StandardCharsets.UTF_8);
    addNodesToGraph(lines);
    addEdgesToGraph(lines);
}

private int addNodesToGraph(List<String> lines) {
    HashSet<String> set = new HashSet<String>();
    for (String line : lines) {
        String[] nodesId = line.split(" ");
        set.add(nodesId[0]);
        set.add(nodesId[1]);
    }
    adj = new Node[set.size()];
    int i = 0;
    for (String nodeId : set) {
        Node node1 = new Node();
        node1.id = nodeId;
        node1.firstEdge = null;
```

```
            adj[i] = node1;
            i++;
        }
        return set.size();
    }

    private void addEdgesToGraph(List<String> lines) {
        for (String line : lines) {
            String[] nodesId = line.split(" ");
            addEdgeBetweenNode(nodesId[0], nodesId[1]);
        }
    }
```

4. A function called addEdge(int u, int v) that takes as input parameters two integers representing two vertex labels, the endpoints of the new edge. This function will add an edge between two existing nodes to the graph.

```
    private void addEdgeBetweenNode(String node1Id, String node2Id) {
        int node1Pos = getNodePosition(node1Id);
        int node2Pos = getNodePosition(node2Id);
        Edge edge1 = new Edge();
        edge1.ivex = node2Pos;
        edge1.nextEdge = null;
        if (adj[node1Pos].firstEdge == null) {
            adj[node1Pos].firstEdge = edge1;
        } else {
            addEdgeToNode(edge1, adj[node1Pos]);
        }
        Edge edge2 = new Edge();
        edge2.ivex = node1Pos;
        edge2.nextEdge = null;
        if (adj[node2Pos].firstEdge == null) {
            adj[node2Pos].firstEdge = edge2;
        } else {
            addEdgeToNode(edge2, adj[node2Pos]);
        }
    }

    private void addEdgeToNode(Edge edge1, Node node1) {
        Edge tEdge = node1.firstEdge;
        if (tEdge == null) {
            node1.firstEdge = edge1;
            return;
        }
        while (tEdge.nextEdge != null) {
            tEdge = tEdge.nextEdge;
        }
        tEdge.nextEdge = edge1;
    }

    private int getNodePosition(String nodeId) {
        for (int i = 0; i < adj.length; i++) {
            if (adj[i].id.equals(nodeId)) {
                return i;
            }
        }
        return -1;
    }
```

5. Create a function Neighbors(int v) that takes a as input a given vertex and prints all the neighbors of that vertex.

```
public void printNeighbors(Node node1) {
    Edge tEdge = node1.firstEdge;
    while (tEdge != null) {
        System.out.print(adj[tEdge.ivex].id + ", ");
        tEdge = tEdge.nextEdge;
    }
}
```

6. Add a function that prints the Adjcency list representation of the graph.

```
public void print() {
    for (Node node1: adj) {
        System.out.print(node1.id + ": [");
        Edge tEdge = node1.firstEdge;
        while (tEdge != null) {
            System.out.print(adj[tEdge.ivex].id + ", ");
            tEdge = tEdge.nextEdge;
        }
        System.out.println(")");
    }
}
```

Now, you are going to study some structural properties of the graph. Create a function called Degree(int v) that returns for a given vertex v, its degree. Answer the following questions:

1. What is the average, minimal and maximal degree of the graph? What is the edge-density? Do you consider this graph dense or sparse?
   The average degree of the graph is 3, the minimum degree is 1 and the maximum degree is 4. The edge-density is 0.75, which means the graph is dense.
2. Are there any isolated nodes? If yes, which ones?
   There is no isolated node.
3. Are there any loops?
   Yes, there is an edge between node 1 and node 1 itself.
4. Verify your answers by using the *Neighbors(int v)* function.
   1: [1, 1, 2, 3,]
   2: [1, 3, 3,]
   3: [1, 2, 2, 4,]
   4: [3,]

Finally, write a function that allows to read data graph from the keyboard input. The program will ask the user the total number of vertices, the total number of edges and user will have to type each edge as in the following example:

```
Enter the number of vertices:
5
Enter the number of edges:
4
Enter the edges in graph: <to> <from>
1 2
```

```
public Graph() throws IOException {
    Scanner scanner1 = new Scanner(System.in);
    System.out.println("Enter the number of vertices: ");
    int num_nodes = scanner1.nextInt();
    adj = new Node[num_nodes];
    for (int i = 0; i < num_nodes; i++) {
```

```
            adj[i] = new Node();
            adj[i].id = Integer.toString(i + 1);
            adj[i].firstEdge = null;
        }
        System.out.println("Enter the number of edges: ");
        int num_edges = scanner1.nextInt();
        scanner1.nextLine();
        System.out.println("Enter the edges in graph: <to> <from>");
        for (int i = 0; i < num_edges; i++) {
            String line = scanner1.nextLine();
            String[] nodesId = line.split(" ");
            addEdgeBetweenNode(nodesId[0], nodesId[1]);
        }
}
```

2.2 Adjacency matrix representation of an undirected graph

Define a class GraphAdjMatrix containing three attributes : the order of the graph (number of nodes), the size of the graph (the number of edges) and an adjacency matrix.

Create the following functions:

1. Initialize an empty graph of order N (input parameter) and 0 edges.
2. Initializes a graph from a specified input stream. All the entries of the matrix adj must be initialiazed. You will test this function by reading the graph from the graph.txt file.

```
public class MatrixGraph {
    private int N;
    private int M;
    private boolean[][] adj;

    public MatrixGraph(int capacity) {
        adj = new boolean[capacity][capacity];
        for (int i = 0; i < capacity; i++) {
            for (int j = 0; j < capacity; j++) {
                adj[i][j] = false;
            }
        }
    }

    public MatrixGraph(String filePath) throws IOException {
        List<String> lines = Files.readAllLines(Paths.get(filePath),
                StandardCharsets.UTF_8);
        int capacity = lines.size();
        adj = new boolean[capacity][capacity];
        for (int i = 0; i < capacity; i++) {
            for (int j = 0; j < capacity; j++) {
                adj[i][j] = false;
            }
        }
        for (String line : lines) {
            String[] nodesId = line.split(" ");
            adj[Integer.parseInt(nodesId[0])][Integer.parseInt(nodesId[1])] =
true;
            }
        }
}
```

Why is it preferably to use an adjacency list representation in practical contexts?

An adjacency matrix uses $O(n^2)$ memory. It has fast lookups to check for presence or absence of a specific edge, but slow to iterate over all edges.

Adjacency lists use memory in proportion to the number edges, which might save a lot of memory if the adjacency matrix is sparse. It is fast to iterate over all edges, but finding the presence or absence specific edge is slightly slower than with the matrix. That's why is it preferably to use an adjacency list representation in practical contexts.

3 Practical application

Why is it possible to say that the nodes 1 and 34 occupy a central position?

```
public class Main {
    public static void main(String[] args) throws Exception {
        Graph graph = new Graph("karate.txt");
        graph.print();
    }
}
```

Nodes with high degree occupy a central position, and Node 1 and Node 34 have the highest degrees. So we can say that the nodes 1 and 34 occupy a central position. Here is the output:

22: [1, 2,]
23: [33, 34,]
24: [26, 28, 30, 33, 34,]
25: [26, 28, 32,]
26: [24, 25, 32,]
27: [30, 34,]
28: [3, 24, 25, 34,]
29: [3, 32, 34,]
30: [24, 27, 33, 34,]
31: [2, 9, 33, 34,]
32: [1, 25, 26, 29, 33, 34,]
10: [3, 34,]
11: [1, 5, 6,]
33: [3, 9, 15, 16, 19, 21, 23, 24, 30, 31, 32, 34,]
12: [1,]
34: [9, 10, 14, 15, 16, 19, 20, 21, 23, 24, 27, 28, 29, 30, 31, 32, 33,]
13: [1, 4,]
14: [1, 2, 3, 4, 34,]
15: [33, 34,]
16: [33, 34,]
17: [6, 7,]
18: [1, 2,]
19: [33, 34,]
1: [2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 18, 20, 22, 32,]
2: [1, 3, 4, 8, 14, 18, 20, 22, 31,]
3: [1, 2, 4, 8, 9, 10, 14, 28, 29, 33,]
4: [1, 2, 3, 8, 13, 14,]

5: [1, 7, 11,]
6: [1, 7, 11, 17,]
7: [1, 5, 6, 17,]
8: [1, 2, 3, 4,]
9: [1, 3, 31, 33, 34,]
20: [1, 2, 34,]
21: [33, 34,]

All source code in this chapter can be found here: https://github.com/Lessica/Advanced-Algorithm