Tutorial course 4: Algorithms for traversing a graph and shortest paths

---

Guillaume Lachaud

Contents

Listings

In this tutorial, we use the implementation of the adjacency list of a graph that we created in the third tutorial. It is contained in the file **"GraphList.java"**[1].
The implementations of DFS, BFS and the Connected Complements class follow the ones in [1].

Depth First Search Algorithm (DFS)

The *Depth First Search* algorithm can be found in the class **"DepthFirstPaths"**

Listing 1: DepthFirstPaths class

```
1  public class DepthFirstPaths<T> {
2    private Map<Node, Boolean> marked;
3    private Map<Node, Node> edgeTo;
4    private final Node s;
5    // ...
6    }
```

```
int main() {
printf("hello, world");
return 0;
}
```

The variable **s** refers to the node chosen as the starting point of the graph.
The variable **marked** keeps trace of all the nodes that have been traversed.
The variable **edgeTo** keeps records of paths leading from each point of the graph to the source node (when such a path exists).

The *Depth First Search* (DFS) algorithm can then be implemented as follows:

Listing 2: Depth First Search algorithm

```
1    private void dfs(GraphList G, Node v) {
2      marked.put(v, true);
3      Iterator<Node<T>> iterator = v.getListNeighbors().iterator();
4      while (iterator.hasNext()) {
5        Node nodeIteration = iterator.next();
6        if (!marked.containsKey(nodeIteration)) {
7          edgeTo.put(nodeIteration, v);
8          dfs(G, nodeIteration);
9        }
10     }
11   }
```

1. If we want to print the vertices in the order of their first encounter, we could add the line

```
1  System.out.println(G.getNode(v).getNodeName());
```

just after line 2 in Listing 2.

[1]The file **"GraphFunctions.java"** provides some complement needed to define the adjacency list in **"GraphList.java"**.

2. To search for connected components in a graph, we created the class **"CC.java"** which is implemented as follows:

Listing 3: CC class

```
1 public class CC<T> {
2   private Map<Node, Boolean> marked;
3   private Map<Node, Integer> id;
4   private int count;
5   // ...
6 }
```

The variable **count** keeps track of the number of connected components in the graph.
The variable **marked** keeps trace of all the nodes that have been traversed.
The variables **id** keeps track of which component each node belongs to.

3. The nodes are traversed in this order: 5 - 2 - 1 - 3 - 4 - 6 - 7
The graph has **one component**. It is **connected**.

Breadth First Search (BFS)

The *Breadth First Search* (BFS) algorithm can be implemented as follows:

Listing 4: Breadth First Search algorithm

```
1   private void bfs(GraphList G, Node s) {
2     Queue<Node> queue = new LinkedList<>();
3     marked.put(s, true);
4     queue.add(s);
5     while (!queue.isEmpty()) {
6       Node v = queue.remove();
7       Iterator<Node<T>> iterator = v.getListNeighbors().iterator();
8       while (iterator.hasNext()) {
9         Node nodeIteration = iterator.next();
10        if (!marked.containsKey(nodeIteration)) {
11          edgeTo.put(nodeIteration, v);
12          marked.put(nodeIteration, true);
13          queue.add(nodeIteration);
14        }
15      }
16    }
17  }
```

To contrast with the DFS implementation, we decided to implement the BFS in a non-recursive fashion.
The algorithm is in the class **"BreadthFirstPaths.java"**. It shares the same class variables as **"DepthFirstPaths.java"**, i.e.

Listing 5: BreadthFirstPaths class

```
1 public class BreadthFirstPaths<T> {
2   private Map<Node, Boolean> marked;
```

```
3     private Map<Node, Node> edgeTo;
4     private final Node s;
5     //
6  }
```

1. If we want to print the vertices in the order of their first encounter, we could add the line

```
1  System.out.println(G.getNode(s).getNodeName());
```

   just after line 3 in Listing 4 and

```
1  System.out.println(G.getNode(nodeIteration).getNodeName());
```

   just after line 12 in listing 4.

2. The nodes are traversed in this order: 5 - 2 - 6 - 1 - 3 - 7 - 4
   The graph has **one component**. It is **connected**.

References

[1] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley Professional, 4th ed., 2011.