Institut für Theoretische Informatik
Peter Widmayer
Antonis Thomas

27. April 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 8 FS 16

Cycle detection with DFS

The objective of this exercise was to perform cycle detection on an input graph $G = (V, E)$. Let the set of vertices be $V = \{0, 1, \ldots, n-1\}$. The graph is given by its adjacency list. If the input is written as we suggested in the template, it is stored in an array of ArrayLists named `graph`. The result is that `graph[i]` is an ArrayList that contains all the vertices that are adjacent to vertex `i`.

The solution we explore here is to decide cycle detection with the use of DFS. Below we give two versions of cycle detection with DFS. The first is recursive, while the second is non-recursive.

Recursive version

```
class Main {
    boolean[] visited;
    ArrayList<Integer>[] graph;

    //return true iff there is a cycle in graph.
    boolean dfs_cycle() {
        //A boolean array that holds if a vertex has been visited
        visited = new boolean[graph.length];
        Arrays.fill(visited, false);

        return dfs_rec(0,0);
    }

    //recursive function. Returns true iff there is a cycle.
    //v is the current vertex on the DFS
    //u is the previous vertex, i.e. the vertex from which v was called.
    boolean dfs_rec(int v, int u) {
        visited[v] = true;

        for (int w : graph[v]) {
            if(!visited[w]) {
                //if the recursive call returns true then we want it to return true
                //but if the recursive call returns false then we want the for loop to
                    continue
                if (dfs_rec(w,v)) return true;
            }
            //if you see a visited vertex that is not the previous one then there is a
                cycle
            else if (w != u) {
                return true;
            }
        }
        return false;
    }
    ...
```

Non-Recursive version

This version uses a Stack to avoid recursion.

```
class Main {
    boolean[] visited;
    ArrayList<Integer>[] graph;

    //return true iff there is a cycle in graph.
    boolean dfs_cycle() {
        int n = graph.length;

        //A boolean array that holds if a vertex has been visited
        visited = new boolean[n];
        Arrays.fill(visited, false);

        Stack<Integer> stack = new Stack();
        stack.push(0); //0 is treated as the root

        //An array that holds which is the previous vertex
        //w.r.t. dfs for each vertex
        int[] previous = new int[n];
        Arrays.fill(previous, -1);

        while (!stack.empty()) {
            int v = stack.pop();
            visited[v] = true;

            for (int w : graph[v]) {
                if (visited[w]) {
                    //if w is visited and it's not the
                    //previous of v then we found a cycle
                    if (w != previous[v]) return true;
                }
                else {
                    stack.push(w);
                    //we set the previous of w to be v
                    //only if the previous of v is not w
                    if (previous[v]!=w) previous[w] = v;
                }
            }
        }
        //if this point is reached the whole graph has
        //been explored and no cycles have been found
        return false;
    }
    ...
```

Main function

We include once again the main function from the given template. Now it also includes the call to the appropriate function for cycle detection.

```
    ...
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int test = in.nextInt();

        for (int t=0; t<test; t++) {
            Main main = new Main();
            int n = in.nextInt();
            in.nextLine(); //read the extra new line character
            //The above is because nextInt() ignores the \n character
            // that comes after the integer. nextLine() reads and
            //in this case discards this character.

            main.graph = new ArrayList[n];
            for (int i = 0; i < n; i++)
                main.graph[i] = new ArrayList<Integer>();

            for (int j=0; j<n; j++) {
                String adj = in.nextLine();

                //The split() functions splits the string
                //into an array of strings according the
                // split character " "  (space)
                for (String s : adj.split(" ")) {
                    //tries to parse each string to an integer
                    //and adds this integer to the adjacency list
                    //of the right vertex
                    try {main.graph[j].add(Integer.parseInt(s));}
                    catch (java.lang.NumberFormatException nfe) {
                        //do nothing, just ignore garbage characters
                        //for example the new line character \n
                    }
                }
            }

            if (main.dfs_cycle()) System.out.println("y");
            else System.out.println("n");
        }
    }
}
```