Algorithmen und Datenstrukturen
Musterlösung 9

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

24. Mai, zur Besprechung am 31. Mai

Aufgabe 1) Graphen

1. (a)

```java
import java.util.Set;
import java.util.TreeSet;

public class AdjacencyMatrixGraph implements Graph {
    static final byte FREE=0;
    static final byte USED=1;

    // Sei G = (V,E) der repraesentierte Graph
    byte[] status; // v in V genau dann wenn status[v] == USED
    byte[][] matrix; // (v,w) in E genau dann wenn matrix[v][w] == 1
                     // weiters gilt: wenn matrix[v][w] == 1 dann ist v,w in V
    int cnt;

    public AdjacencyMatrixGraph(int size) {
    matrix = new byte[size][size];
    status = new byte[size];
    cnt = 0;
    }


    public int size() {
    return cnt;
    }

    public boolean containsNode(int node) {
    return 0 <= node && node < status.length && status[node]==USED;
    }

    public boolean containsEdge(int source, int target) {
    return matrix[source][target] == 1;
    }
```

```
32
33      public int freshNode() {
34      int node = -1;
35      int size = this.status.length;
36
37      // Suche nach dem ersten freien Platz
38      for (int i = 0; i < size; i++) {
39          if (status[i] == FREE) {
40          node = i;
41          break;
42          }
43      }
44
45      // Wenn kein freier Platz, Dimension verdoppeln
46      if (node < 0) {
47          int newsize = size * 2;
48          byte[] newstatus = new byte[newsize];
49          byte[][] newmatrix = new byte[newsize][newsize];
50
51          System.arraycopy(status, 0, newstatus, 0, size);
52          for(int i = 0; i < size; i++) {
53          System.arraycopy(matrix[i], 0, newmatrix, 0, size);
54          }
55          matrix = newmatrix;
56          status = newstatus;
57      }
58
59      cnt++;
60      status[node] = USED;
61      return node;
62      }
63
64      public void addEdge(int source, int target) throws
            NodeNotFoundException {
65      if (! containsNode(source)) {
66          throw new NodeNotFoundException(source);
67      } else if (! containsNode(target)) {
68          throw new NodeNotFoundException(target);
69      } else {
70          matrix[source][target] = 1;
71      }
72      }
73
74      public void removeEdge(int source, int target) {
75      if (containsNode(source) && containsNode(target)) {
76          matrix[source][target] = 0;
77      }
78      }
79
80      public void removeNode(int node) {
81      if(containsNode(node)) {
82          status[node] = FREE;
83          for (int j = 0; j < status.length; j++) {
84          matrix[node][j] = 0;
85          matrix[j][node] = 0;
86          }
87          cnt--;
88      }
89      }
90
91
92      public Set<Integer> successors(int node) {
```

```
93      Set<Integer> s = new TreeSet();
94      if(containsNode(node)) {
95          for (int i = 0; i < status.length; i++) {
96          if (containsEdge(node, i)) {
97              s.add(i);
98          }
99          }
100     }
101     return s;
102     }
103
104     public Set<Integer> predecessors(int node) {
105     Set<Integer> s = new TreeSet();
106     if(containsNode(node)) {
107         for (int i = 0; i < status.length; i++) {
108         if (containsEdge(i, node)) {
109             s.add(i);
110         }
111         }
112     }
113     return s;
114     }
115
116     public int inDegree(int node) {
117     return predecessors(node).size();
118     }
119
120     public int outDegree(int node) {
121     return successors(node).size();
122     }
123
124 }
```

```
1  import java.util.Set;
2  import java.util.TreeSet;
3
4  public class AdjacencyListGraph implements Graph {
5
6      // Sei G = (V,E) der repraesentierte Graph
7      TreeSet<Integer>[] list; // list[u] == { v | (u,v) in
           Eckenmenge E }
8                               // weiters gilt: list[u] != null
                                    genau dann wenn u in V
9      int cnt;                 // Anzahl der Knoten in G
10
11     public AdjacencyListGraph(int size) {
12     list = (TreeSet<Integer>[]) new Object[size];
13     cnt = 0;
14     }
15
16     public int size() {
17     return cnt;
18     }
19
20     public boolean containsNode(int node) {
21     return 0 <= node && node < list.length && list[node]!=null;
22     }
23
24     public boolean containsEdge(int source, int target) {
25     return containsNode(source) && containsNode(target) && list[
           source].contains(target);
26     }
```

```
27
28      public int freshNode() {
29      int node = -1;
30      int length = this.list.length;
31
32      // Suche nach dem ersten freien Platz
33      for (int i = 0; i < length; i++) {
34          if (! containsNode(i)) {
35          node = i;
36          break;
37          }
38      }
39
40      // Wenn kein freier Platz, Liste verdoppeln
41      if (node < 0) {
42          int newlength = length * 2;
43          TreeSet<Integer>[] newlist = (TreeSet<Integer>[]) new
                Object[newlength];
44
45          System.arraycopy(list, 0, newlist, 0, length);
46
47          list = newlist;
48          node = length;
49      }
50
51      cnt++;
52      list[node] = new TreeSet<Integer>();
53          return node;
54      }
55
56      public void addEdge(int source, int target) throws
            NodeNotFoundException {
57      if (! containsNode(source)) {
58          throw new NodeNotFoundException(source);
59      } else if (! containsNode(target)) {
60          throw new NodeNotFoundException(target);
61      } else {
62          list[source].add(target);
63      }
64      }
65
66      public void removeEdge(int source, int target) {
67      if (containsNode(source) && containsNode(target)) {
68          list[source].remove(target);
69      }
70      }
71
72      public void removeNode(int node) {
73      if(containsNode(node)) {
74          list[node] = null;
75          for (int j = 0; j < list.length; j++) {
76          removeEdge(node, j);
77          }
78          cnt--;
79      }
80      }
81
82      public Set<Integer> successors(int node) {
83      if(containsNode(node)) {
84          return list[node];
85      } else {
86          return new TreeSet<Integer>();
```

```
87      }
88      }
89
90      public Set<Integer> predecessors(int node) {
91      Set<Integer> s = new TreeSet();
92      if(containsNode(node)) {
93          for (int i = 0; i < list.length; i++) {
94          if (containsEdge(i, node)) {
95              s.add(i);
96          }
97          }
98      }
99      return s;
100     }
101
102     public int inDegree(int node) {
103     return predecessors(node).size();
104     }
105
106     public int outDegree(int node) {
107     return successors(node).size();
108     }
109 }
```

Aufgabe 2) Topologisches Sortieren

(h) (a) $G = (V, E)$ ist azyklisch $\Rightarrow$ es existiert eine topologische Sortierung:
Wir beweisen dies mittels Induktion über $|V|$. Im Basisfall ist $|V| = 0$ und es existiert trivialerweise eine topologische Sortierung.

Wir betrachten den Induktionsschritt. Die Induktionshypothese besagt dass für alle azyklischen Graphen mit n Knoten eine topologische Sortierung besteht. Dies ist nun für alle Graphen mit $n + 1$ Knoten zu zeigen. Sei G ein beliebiger azyklischer Graph mit $n + 1$ Knoten. Da E endlich ist, ist leicht zu sehen dass ein Knoten $v \in V$ existiert dessen Eingangsgrad 0 ist (ansonsten lässt sich ein unendlicher Weg in G durch "rückwärtsspazieren" in G konstruieren). Der (azyklische) Graph $G - v$ beinhaltet genau n Knoten, somit existiert per Induktionshypothese eine topologische Sortierung ord_I fuer $G-v$. Da v so gewählt ist dass keine Kante $(v, w) \in E$ existiert ist die Funktion $\mathsf{ord}(v) := 1$ und sonst $\mathsf{ord}(w) = \mathsf{ord}_I(w) + 1$ eine topologische Sortierung für G.

(b) Es existiert eine topologische Sortierung $\Rightarrow$ G ist azyklisch:
Sei G ein beliebiger Graph mit topologischer Sortierung ord. Wir führen einen Widerspruchsbeweis und nehmen an dass G einen Zykel $v_0, v_1, \ldots, v_0$ enthält. Dann gilt aber $\mathsf{ord}(v_0) < \mathsf{ord}(v_1) < \cdots < \mathsf{ord}(v_0)$ und darum insbesonder $\mathsf{ord}(v_0) < \mathsf{ord}(v_0)$. Widerspruch, somit kann unsere Annahme nicht stimmen, d.h. G azyklisch.

2.
```java
import java.util.Stack;
import java.util.Iterator;

public class TopologicalSort {
    public int[] topsort(Graph graph) {
    int[] ord = new int[graph.size()];
    int[] indeg = new int[graph.size()];
    Stack<Integer> indegZero = new Stack<Integer>();

    for(int i = 0; i < graph.size(); i++) {
        int d = graph.inDegree(i);
        indeg[i] = d;
        if (d == 0) { indegZero.push(d); }
    }


    int i = 0;
    while (! indegZero.empty()) {
        int v = indegZero.pop();
        i++;
        ord[v] = i;

        Iterator<Integer> it = graph.successors(v).iterator();
        while( it.hasNext() ) {
        int w = it.next();
        indeg[w]--;
        if (indeg[w] == 0) {
            indegZero.push(w);
        }
        }
    }
    if (i == graph.size()) {
        return ord;
    } else {
        return null;
    }
```

```
37          }
38  }
```