Algorithms & Datastructures
Laboratory Exercise Sheet 11

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

June 8th, to be discussed on June 15th

Exercise 1) Graphs – sequence of operations You have an unlimited water source (such as a river) and only two buckets B_1 and B_2. The bucket B_1 has a maximum volume of 5 liters and the bucket B_2 has a maximum volume of 3 liters. You are allowed to perform following operations: you can fill in a bucket with water from the water source, you can pour over the water from one bucket to the second bucket and vice-versa, and you can empty a bucket (pour the water from the bucket back to the water source).

1. Draw a graph that represents a model of this exercise (think about what vertices and edges can represent).
2. How can you get exactly 4 liters of water in the bucket B_1? Show a path in the graph that represents the solution.

Exercise 1) Solution A vertex in the graph is a state, that is, a B_1 and B_2 with certain amount of water, and an edge is a transition between two states, that is, an operation you perform. Note, that it makes sense to perform only operations after which one of the buckets are either full or empty.

For example, a state $s_0 = (0, 0)$ denotes that B_1 and B_2 have both 0 liters of water, a state $s_1 = (0, 3)$ denotes that B_1 has 0 liters and B_2 has 3 liters, etc. A transition between s_1 and s_2 denotes an operation in which you fill the bucket B_2 with the water from the water source.

You have to find a sequence of states that lead to a state $s_n = (4, 0)$ where n is the number of operations you have to perform. One possible solution is the following sequence of states with $n = 8$:

- $s_0 = (0, 0)$, both buckets are empty, initial state.
- $s_1 = (0, 3)$, after you fill in B_2.
- $s_2 = (3, 0)$, after you pour the water from B_2 over to B_1.
- $s_3 = (3, 3)$, after you fill in B_2.
- $s_4 = (5, 1)$, after you pour 2 liters of the water from B_2 over to B_1 (in B_1 you have 5 liters and in B_2 it remains 1 liter).
- $s_5 = (0, 1)$, after you empty B_1.
- $s_6 = (1, 0)$, after you pour the water from B_2 over to B_1.
- $s_7 = (1, 3)$, after you fill in B_2.
- $s_8 = (4, 0)$, after you pour the water of B_2 over to B_1 resulting in 4 liters in B_1, final state.

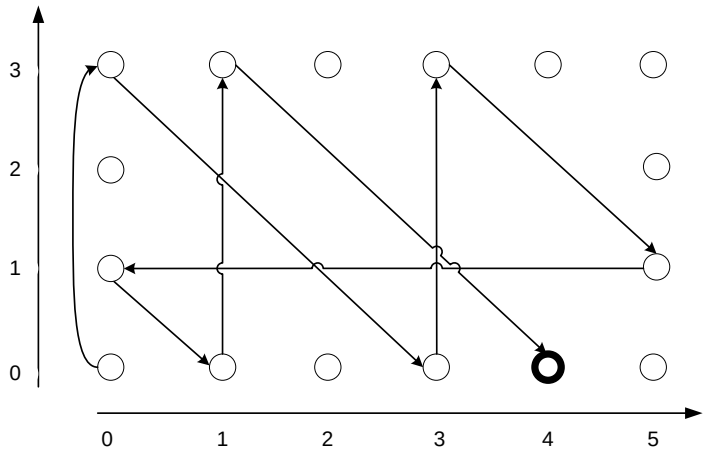
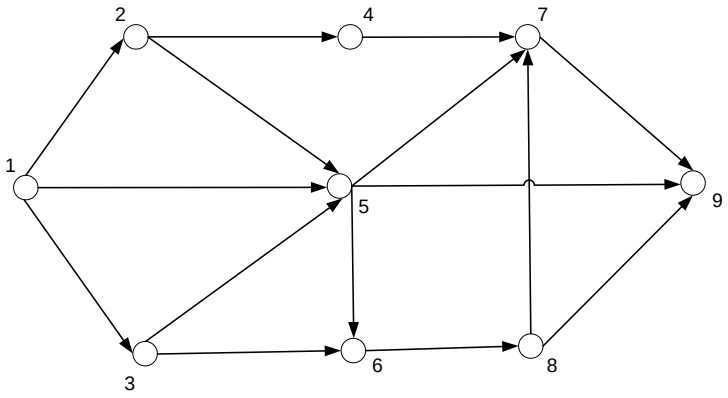


Figure 1: Sequence of states leading to B_1 having exactly 4 liters of water

Exercise 2) Topological Sort You have an acyclic graph in the following figure.



1. Explain how a topological sort algorithm works and show how it runs on the given graph. Show a topological sort of both vertices and edges.

2. Implement the topological sort algorithm in the method `TS(List<Vertex> ts_v, List<Edge> ts_e)` that fills the `ts_v` list (topologically sorted vertices) and `ts_e` list (topologically sorted edges). Use the following code skeleton.

```
public class Vertex {
    int id;

    public Vertex(int id) {
        this.id = id;
    }

    public boolean equals(Object other) {
        if (other instanceof Vertex) {
            return this.id == ((Vertex)other).id;
        }
        return false;
    }

    public int hashCode() {
        return id;
    }

    public String toString() {
        return new Integer(id).toString();
    }
}
```

```
public class Edge {
    Vertex SV;
    Vertex EV;

    public Edge(Vertex SV, Vertex EV) {
        this.SV = SV;
        this.EV = EV;
    }

    public String toString() {
        return "(" + SV.toString() + "," + EV.toString() + ")";
    }
}
```

```
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import java.util.Map.Entry;

public class Graph {

    Hashtable<Integer, Vertex> vertices = new Hashtable<Integer,
        Vertex>();
    ArrayList<Edge> edges = new ArrayList<Edge>();

    /* add an egde to the graph */
    public Edge addEdge(int startV, int endV) throws Exception {
        // check if the edge already exist
        for (Edge e : edges)
            if (e.SV.id == startV && e.EV.id == endV)
                throw new Exception("The edge (" + startV + "," + endV
                    + ") already exists!");

        // get the starting and ending vertex of the edge if they
            exist
        Vertex SV = vertices.get(startV);
        Vertex EV = vertices.get(endV);

        // create vertices if they do not exist
        if (SV == null) {
            SV = new Vertex(startV);
            vertices.put(startV, SV);
        }
        if (EV == null) {
            EV = new Vertex(endV);
            vertices.put(endV, EV);
        }

        // create the edge
        Edge e = new Edge(SV, EV);
        edges.add(e);
        return e;
    }

    /* topological sort */
    public void TS(List<Vertex> ts_v, List<Edge> ts_e) {
        // implement me!
    }

    public static void main(String[] args) throws Exception {
        Graph g = new Graph();
        g.addEdge(1, 2); g.addEdge(1, 5); g.addEdge(1, 3);
        g.addEdge(2, 4); g.addEdge(2, 5); g.addEdge(5, 7);
        g.addEdge(5, 9); g.addEdge(5, 6); g.addEdge(3, 5);
        g.addEdge(3, 6); g.addEdge(4, 7); g.addEdge(7, 9);
        g.addEdge(6, 8); g.addEdge(8, 7); g.addEdge(8, 9);

        ArrayList<Vertex> ts_v = new ArrayList<Vertex>();
        ArrayList<Edge> ts_e = new ArrayList<Edge>();

        g.TS(ts_v, ts_e);

        for (Vertex v : ts_v)
            System.out.println(v);
```

```
            for (Edge e : ts_e)
                System.out.println(e);
        }
    }
```

3. What is the complexity of the algorithm?

4. Prove that there exists a topological sorting for a graph iff the graph is acyclic.

Exercise 2) Solution

1. Topological sort of vertices is: 1, 2, 3, 4, 5, 6, 8, 7, 9. Topological sort of edges is: (1,2), (1,5), (1,3), (2,4), (2,5), (3,5), (3,6), (4,7), (5,7), (5,9), (5,6), (6,8), (8,7), (8,9), (7,9).

2. Implementation: First, we add a helper attribute c to the `Vertex` class:

```
public class Vertex {
    int id;
    int c = 0;  // helper attribute for topological sort

    public Vertex(int id) {
        this.id = id;
    }

    public boolean equals(Object other) {
        if (other instanceof Vertex) {
            return this.id == ((Vertex)other).id;
        }
        return false;
    }

    public int hashCode() {
        return id;
    }

    public String toString() {
        return new Integer(id).toString();
    }
}
```

The implementation of `TS(List<Vertex> ts_v, List<Edge> ts_e)`:

```
    public void TS(List<Vertex> ts_v, List<Edge> ts_e) {
        // value c for a vertex x is the number of edges of which x is
            the ending vertex
        // and that having a starting vertex not yet in the
            topological sort list (ts_v)
        // set c for every vertex to 0
        for (Entry<Integer, Vertex> e : vertices.entrySet())
            e.getValue().c = 0;

        // calculate c of every vertex that is the ending vertex of an
            edge
        for (Edge e : edges)
            e.EV.c = e.EV.c + 1;

        // initialize helper M list; initially add every vertex that
            has c equal to 0
        ArrayList<Vertex> M = new ArrayList<Vertex>();
        for (Entry<Integer, Vertex> e : vertices.entrySet())
            if (e.getValue().c == 0)
                M.add(e.getValue());

        while (M.size() != 0) {
            // get the first vertex from M, remove it from M and add
                it to ts_v
            Vertex x = M.remove(0);
            ts_v.add(x);

            // for all edges of which x is the starting vertex do
```

```
            for (Edge e : edges)
                if (e.SV.id == x.id) {
                    ts_e.add(e);
                    e.EV.c = e.EV.c - 1;
                    if (e.EV.c == 0)
                        M.add(e.EV);
                }
        }
    }
```

3. It is $\mathcal{O}(|V|\cdot|E|)$, so not optimal. The main reason for not obtaining the optimal complexity of $\mathcal{O}(|V|+|E|)$ is the expensive lookup of finding all successors of given vertex.

4. Let $E(G)$ and $V(G)$ be all edges and vertices of G respectively. Proof there exists a topological sorting for a graph G iff the graph is acyclic.

   (a) Part 1: G has a topological sorting if G is acyclic.

   Let's assume that the topological sorting exists for G, thus for every edge $e = (v_i, v_j), e \in E(G), v_i, v_j \in V(G)$ it holds that $i < j$. Further, let's assume that G is cyclic (contradiction). Let $SV(e)$ and $EV(e)$ be a starting and an ending vertex of $e \in E(G)$ and $e_1, e_2, ..., e_k \in E(G)$ be a cycle in G such that $EV(e_i) = SV(e_{i+1}), i = 1, 2, ..., k-1$ and $EV(e_k) = SV(e_1)$. Since the topological sorting exist for G, for indices $i_1, i_2, ..., i_k$ of vertices $v_1, v_2, ..., v_k$ of edges from the cycle it must hold that $i_1 < i_2 < ... < i_k < i_1$. Such ordering is not possible.

   (b) Part 2: G is acyclic if there exist a topological sorting.

   We perform induction on the number of vertices of G. If $V(G) = \emptyset$ then the graph is obviously acyclic. Otherwise, let $v_1 < v_2 < \cdots < v_n$ be the topological sorting of vertices of $G, v_i \in V(G), 1 \leq i \leq n, n = |V(G)|$. Hence, $v_1 < v_2 < \ldots v_{n-1}$ is a topologic sorting of $G - v_n$. By the induction hypothesis we know that $G - v_n$ is acyclic. Hence, to show that G is acyclic we only have to prove that there are no cycles which involve v_n. But there cannot be such a cycle since v_n is the last element in the topologic sorting of G and thus, has no outgoing edges.