Datenstrukturen und effiziente Algorithmen
Blatt 12

Markus Vieth, David Klopp, Christian Stricker

29. Januar 2016

Aufgabe 1

a)

Definiere zwei Farben (hier rot und blau)
Wähle einen beliebigen Knoten als ”CurrentVertex” und setze ihn auf blau.

```
Wiederhole {
  Füge alle Nachbarknoten in eine Warteschlange.
  Falls CurrentVertex blau und mindestens einer der Nachbarknoten auch blau {
    Rückgabe: FALSE
  } ansonsten {
    setze alle Nachbarknoten auf rot
  }

  Falls CurrentVertex rot und mindestens einer der Nachbarknoten auch rot {
    Rückgabe: FALSE
  } ansonsten {
    setze alle Nachbarknoten auf blau
  }
  Wähle den nächsten Knoten aus der Warteschlange als CurrentVertex
}

Falls die Liste leer ist {
  Rückgabe: TRUE
}
```

b)

BipartiterGraph.java

```java
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Vector;

public class BipartiterGraph implements UndirectedGraph{
  HashMap<Integer,Vertex> map = new HashMap<Integer, Vertex>(); //Speichert alle Knoten
  LinkedList<Vertex> list = new LinkedList<Vertex>();    //Warteschlange für die noch zu untersuchenden Knoten
  Vertex beginVertex;                  //Ein Knoten, mit dem der Algo beginnt

  class Edge{
    Vertex start;  //Speichert seine beiden Knoten
    Vertex end;

    public Edge(Vertex start, Vertex end){
      this.start = start;
      this.end = end;
    }
  }

  //Klasse Vertex, Speichert all seine Kanten in edge, speichert seine Prüffarbe und ein value
  class Vertex{

    Vector<Edge> edge = new Vector<Edge>();

    String colour;
    int value;
```

```
28      public Vertex(int value){ //Konstruktor mit eindeutigem Key/Value
29        this.value = value;
30      }

32      public void addEdge(Edge edge){ //Fügt dem Knoten eine seiner Kanten hinzu
33        this.edge.add(edge);
34      }
35    }

37    /*class Graph{
38      Vector<Vertex> vertex;

40      public Graph(){
41        this.vertex = new Vector<Vertex>();
42      }

44      public void addVertex(Vertex vertex){
45        this.vertex.add(vertex);
46      }


49    }*/

51    public BipartiterGraph(){  //Fügt Kanten und Knoten hinzu und überprüft den Graphen
52      insertVertex(1);
53      insertVertex(2);
54      insertVertex(3);
55      insertVertex(4);
56      insertVertex(5);
57      insertVertex(6);
58      insertVertex(7);
59      insertVertex(8);
60      insertEdge(1,1);
61      insertEdge(1,2);
62      insertEdge(2,3);
63      insertEdge(3,5);
64      insertEdge(5,8);
65      insertEdge(2,8);
66      insertEdge(2,4);
67      insertEdge(4,6);
68      insertEdge(1,7);
69      System.out.println(isBipartite());
70    }

72    @Override
73    public void insertVertex(int value) {
74      Vertex vertex = new Vertex(value);  //Erzeugt einen Knoten
75      this.map.put(value, vertex);   //Fügt einen Knoten der Hashmap hinzu
76      if(this.beginVertex == null){   //Falls noch kein Knoten existiert, wird dieser als Startknoten definiert
77        this.beginVertex = vertex;
78      }
79    }

81    @Override
82    public void insertEdge(int value1, int value2) {
83      Vertex start = this.map.get(value1); //Sucht einen Endknoten der Kante
84      Vertex end = this.map.get(value2);  //Sucht den anderen Endknoten der Kante
85      if(start != null && end != null){  //Wenn beide existieren, füge beiden Knoten die Kante hinzu
86        Edge edge = new Edge(start, end);
87        start.addEdge(edge);
88        end.addEdge(edge);
89      }
90      else                //Ansonsten gibt Hinweis aus, das die Kante nicht im Algo berücksichtigt wird
91        System.out.println(" Mindestens einer der Knotenenden der Kante \" "+ value1 +"---" + value2 + " \"
              existiert nicht! \n Diese Kante wird nicht dem Graphen hinzugefügt!");
92    }

94    @Override
95    public boolean isBipartite() {
96      this.beginVertex.colour = "blue"; //Setzt den Anfangsknoten auf eine bestimmte Farbe (blue)
```

```
97      list.add(this.beginVertex);   //Fügt den ersten Knoten der Warteschlange hinzu
98      Vertex next;          //Hilfsvariablen
99      Vertex currentVertex = beginVertex;
100     if(!this.map.isEmpty()){   //Wenn ein Graph existiert, starte
101       do{
102         currentVertex = list.getFirst(); //Der erste Knoten wird aus der Liste genommen

104         String colour = currentVertex.colour;
105         for(Edge edge : currentVertex.edge){ //Iteriert über alle Kanten des derzeitigen Knotens
106           if(!edge.start.equals(currentVertex) || !edge.end.equals(currentVertex)){ //

108             if(edge.start.equals(currentVertex)){ //Bestimme den Nachbarknoten, schließe den derzeitigenn
                    Knoten als Nachbarknoten aus
109               next = edge.end;
110             }else
111               next = edge.start;

113             if(next.colour == null)  //Wenn der Knoten bisher noch nciht in der Warteschlange gewesen ist, d.h.
                    die Farbe ist noch null, füge den Knoten dort ein
114               list.add(next);

116             if(next.colour == colour) //Wenn die Farbe zu dem derzeitigen Knoten gleich ist, haben zwei
                    Nachbarknoten die gleihce Farbe => kien Bipartiter Graph
117               return false;
118             else if(currentVertex.colour.equals("red")) //Ansonsten färbe den Knoten in der zum currentVertex
                    verschiedenen Farbe
119               next.colour = "blue";
120             else
121               next.colour = "red";
122           }
123         }
124         list.remove(currentVertex); //Wenn alle Kanten abgehackt sind, lösche den derzeitigen Knoten aus der
                Liste(Listenanfang)
125       }while(!list.isEmpty());  //Wenn die Warteschlange leer ist und bisher kein Konflikt enstanden ist, gebe
              true zurück.
126     }
127     return true;
128   }

130   public static void main(String[] args) {
131     BipartiterGraph i = new BipartiterGraph();  //Der Konstruktor baut den Graphen auf und überprüft ihn.
          Funktionen müssen nicht static sein
132   }

134 }
```

Aufgabe 2

Aufgabe 3

a)

Knoten	Input	Output	Differenz
0	12	$5+7$	0
1	5	5	0
2	9	$6+3$	0
3	$6+7$	13	0
4	0	0	0
5	$0+3$	3	0

Der Input ist niemals größer als die maximale Kapazität. Für alle Knoten bis auf s und t stimmt die Anzahl des Inputs mit der des Outputs überein. Es handelt sich daher um einen gültigen Fluss.

b)

Nein, der eingezeichnete Fluss ist nicht maximal.

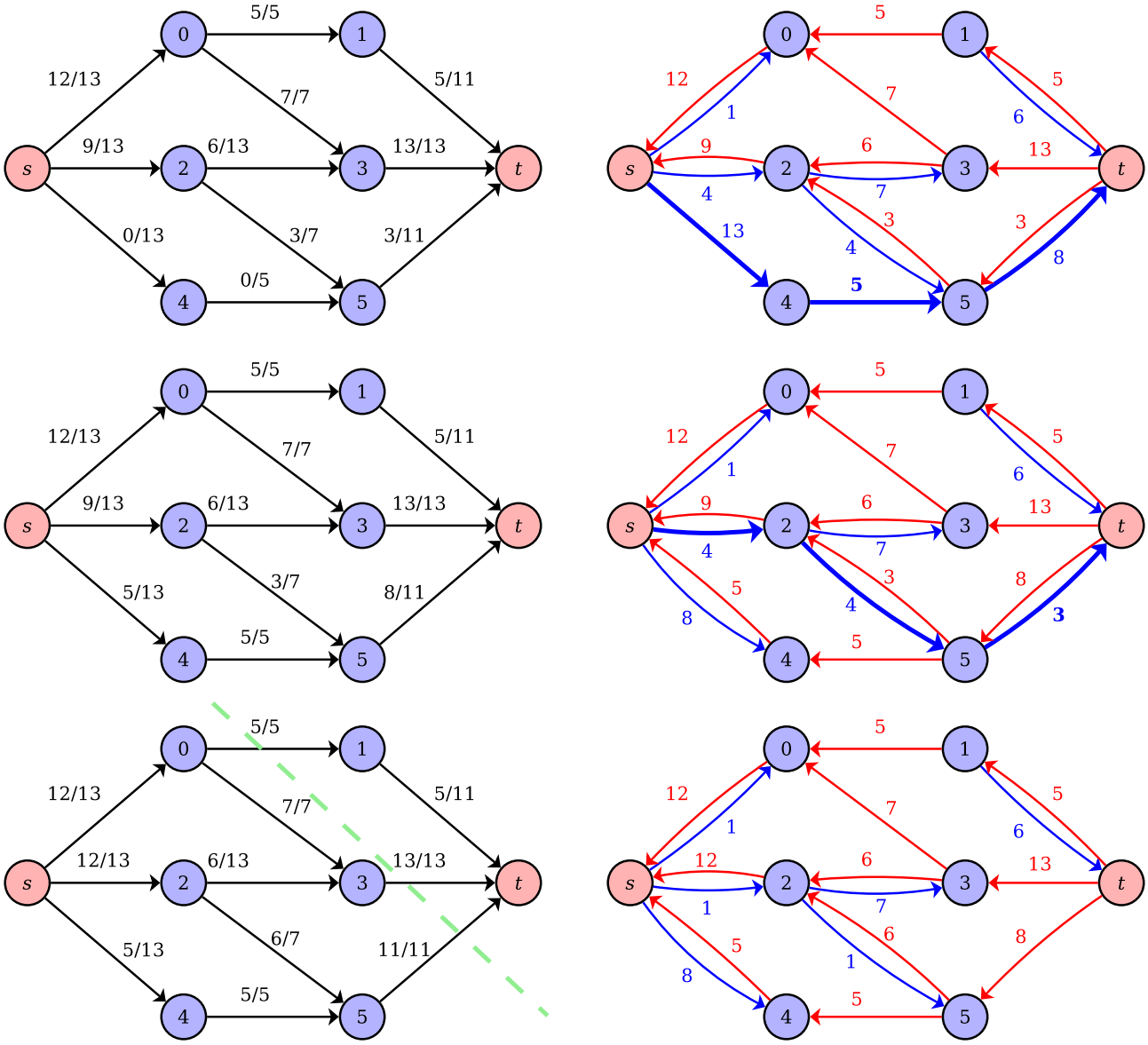

Abbildung 1: Ford-Fulkerson

c)

Siehe grüne, gestrichelte Linie. Der Schnitt geht durch die Kanten 0-1,3-t,5-t und entspricht dem maximalen Fluss (29).

d)

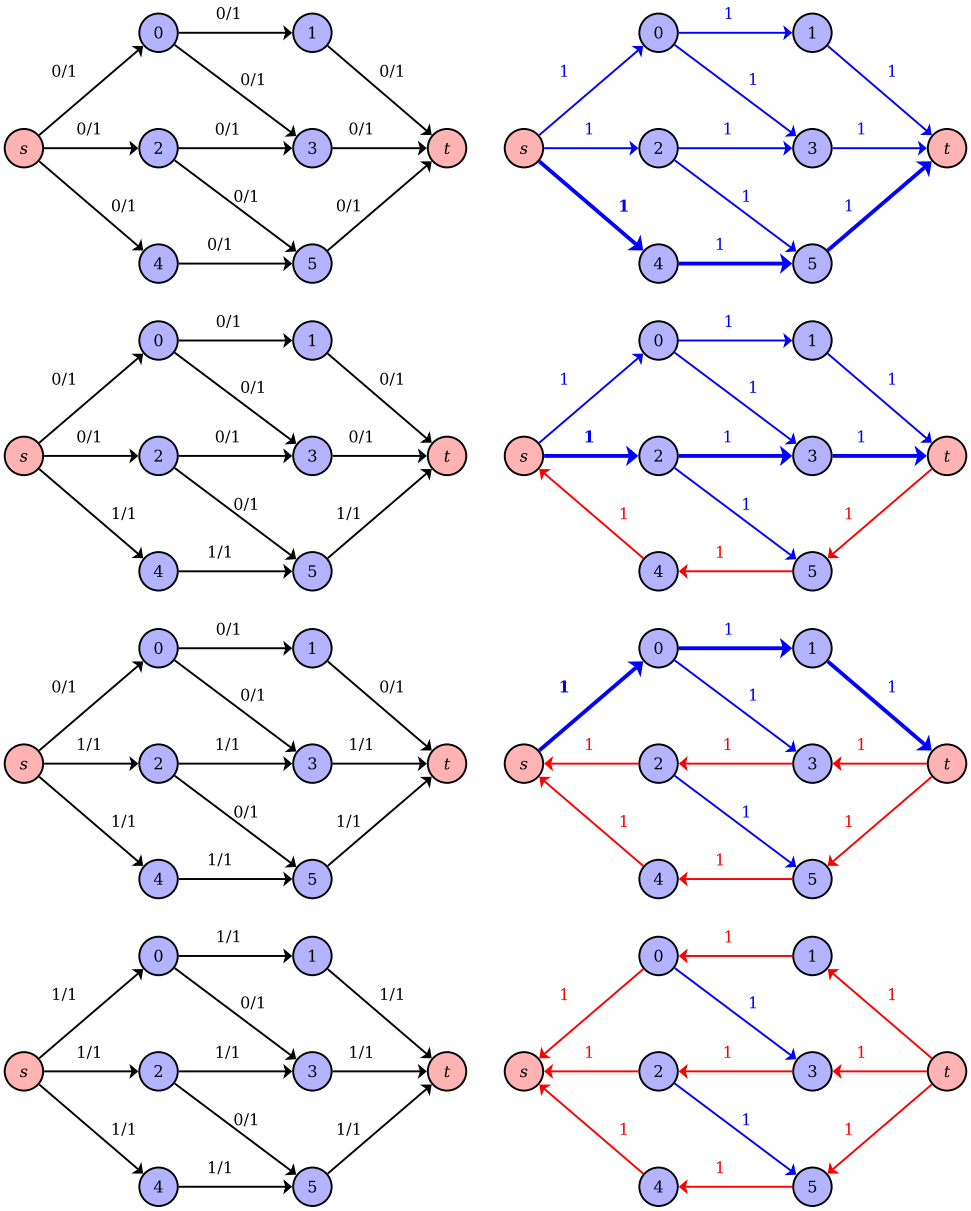

Abbildung 2: Matching

In diesem Fall handelt es sich um ein Matching Problem. Wir haben die beiden Partitionen $A := \{0, 2, 4\}$ $B := \{1, 3, 5\}$. Die lässt sich durch folgende Feststellungen belegen:

- Es gibt nur Pfade von A nach B
- Es gibt keine Pfade innerhalb von A oder B
- s ist mit allen Knoten aus A verbunden und zu t führen nur Kanten aus B

Entfernt man die Knoten s und t, sowie alle mit diesen verbunden Kanten, erhält man einen bipartiten Graphen. Entfernt man nun noch alle Kanten mit einem Durchfluss von 0 und macht aus den gerichteten Kanten ungerichtete, hat man ein maximales Matching.

Markus Vieth, David Klopp, Christian Stricker

Zusatzaufgabe: ResidualGraph.java

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Vector;


public class ResidualGraph {
    //Iterierbare Liste der Knoten
    Vector<Vertex> vertexs;
    //Zuordnung Name -> Knoten
    HashMap<String, Vertex> map;

    /**
     * Constructor
     */
    public ResidualGraph() {
        this.vertexs = new Vector<>();
        this.map = new HashMap<>();
    }

    ////////////////////////////////////////////////////////

    public static void main(String... args) {
        System.out.println("Tailaufgabe b)");
        ResidualGraph test = new ResidualGraph();
        test.addVertex("s");
        for (int i = 0; i < 6; i++)
            test.addVertex(i + "");
        test.addVertex("t");
        test.addEdge("0", "1", 5, 5);
        test.addEdge("0", "3", 7, 7);
        test.addEdge("1", "t", 11, 5);
        test.addEdge("2", "3", 13, 6);
        test.addEdge("2", "5", 7, 3);
        test.addEdge("3", "t", 13, 13);
        test.addEdge("4", "5", 5, 0);
        test.addEdge("5", "t", 11, 3);
        test.addEdge("s", "0", 13, 12);
        test.addEdge("s", "2", 13, 9);
        test.addEdge("s", "4", 13, 0);
        System.out.println("Ausgangssituation");
        test.print();

        test.findMaxFlow("s", "t");
        System.out.println("Ergebnis");

        test.print();

        System.out.println("Tailaufgabe c)");
        System.out.println("Minimaler Schnitt");
        Vector<Edge> cut = test.findMinCut("s", "t");
        //int sum = 0;
        for (Edge e : cut) {
            System.out.println(e.toString());
            //sum += e.capacity;
        }

        System.out.println("Tailaufgabe d)");
        test = new ResidualGraph();
        test.addVertex("s");
        for (int i = 0; i < 6; i++)
            test.addVertex(i + "");
        test.addVertex("t");
        test.addEdge("0", "1", 1);
        test.addEdge("0", "3", 1);
        test.addEdge("1", "t", 1);
        test.addEdge("2", "3", 1);
        test.addEdge("2", "5", 1);
```

```
69          test.addEdge("3", "t", 1);
70          test.addEdge("4", "5", 1);
71          test.addEdge("5", "t", 1);
72          test.addEdge("s", "0", 1);
73          test.addEdge("s", "2", 1);
74          test.addEdge("s", "4", 1);
75          System.out.println("Ausgangssituation");

77          test.print();

79          test.findMaxFlow("s", "t");
80          System.out.println("Ergebnis");

82          test.print();

84          System.out.println("Minimaler Schnitt");
85          cut = test.findMinCut("s", "t");
86          //sum = 0;
87          for (Edge e : cut) {
88              System.out.println(e.toString());
89              //sum += e.capacity;
90          }
91      }

93      ////////////////////////////////////////////////////////

95      /**
96       *
97       * @param name name of the vertex
98       * @return true if successful, false if name is already in use
99       */
100     public boolean addVertex(String name) {
101         Vertex u = new Vertex(name);

103         //Prüfe, ob Name bereits vergeben
104         if (this.map.containsKey(name))
105             return false;

107         this.map.put(name, u);
108         this.vertexs.add(u);
109         return true;
110     }

112     /**
113      * Erzeugt Kante ohne vorhandenen Fluss
114      * @param start name of the start vertex
115      * @param end name of the end vertex
116      * @param capacity max. capacity of the edge
117      */
118     public void addEdge(String start, String end, int capacity) {
119         map.get(start).addEdge(map.get(end), capacity);
120     }

122     /**
123      * Erzeugt Kante mit Fluss
124      * @param start name of the start vertex
125      * @param end name of the end vertex
126      * @param capacity max. capacity of the edge
127      * @param used used capacity of the edge
128      */
129     public void addEdge(String start, String end, int capacity, int used) {
130         map.get(start).addEdge(map.get(end), capacity, used);
131     }

133     /**
134      * Tiefensuche, welche einen Pfad von start nach end sucht
135      * @param start name of source
136      * @param end name of the sink
137      * @return LinkedList with a path from source to sink or null if no path exists
138      */
```

```
139     private LinkedList<Vertex> findPath(String start, String end) {
140         if (start.equals(end))
141             return new LinkedList<>();

143         //Tiefensuche
144         HashSet<Vertex> visited = new HashSet<>();
145         LinkedList<Vertex> stack = new LinkedList<>();
146         HashMap<Vertex, Vertex> pi = new HashMap<>();

148         Vertex origin = this.map.get(start);
149         Vertex sink = this.map.get(end);

151         visited.add(origin);
152         pi.put(origin, null);

154         stack.push(origin);

156         while (stack.size() > 0) {
157             Vertex u = stack.pop();
158             Vector<Edge> edges = u.edges;
159             edges.addAll(u.restEdges);
160             for (int i = 0; i < edges.size(); i++) {
161                 Edge toNext = edges.get(i);
162                 if (toNext.capacity <= 0)
163                     continue;
164                 Vertex next = toNext.getEnd();
165                 if (!visited.contains(next)) {
166                     visited.add(next);
167                     pi.put(next, u);
168                     stack.push(next);
169                 }

171                 //Stoppe, wenn sink erreicht
172                 if (next == sink) {
173                     LinkedList<Vertex> temp = new LinkedList<>();
174                     temp.push(next);
175                     while (pi.get(next) != null) {
176                         next = pi.get(next);
177                         temp.push(next);
178                     }
179                     return temp;
180                 }
181             }
182         }
183         return null;
184     }

186     /**
187      * Suche des minimalen Schnitts über eine Breitensuche
188      * @param start name of the source node
189      * @param end name of the sink node
190      * @return A Vector of the edges, which are connecting partitions S and T.
191      */
192     public Vector<Edge> findMinCut(String start, String end) {
193         if (start.equals(end))
194             return new Vector<>();

196         //Erzeuge maximalen Fluss
197         this.findMaxFlow(start, end);

199         //Breitensuche
200         HashSet<Vertex> visited = new HashSet<>(); //entspricht S
201         LinkedList<Vertex> queue = new LinkedList<>();

203         Vertex origin = this.map.get(start);

205         visited.add(origin);

207         queue.add(origin);
```

```
209         while (queue.size() > 0) {
210             Vertex u = queue.poll();
211             Vector<Edge> edges = u.edges;
212             edges.addAll(u.restEdges);
213             for (int i = 0; i < edges.size(); i++) {
214                 Edge toNext = edges.get(i);
215                 if (toNext.capacity <= 0)
216                     continue;
217                 Vertex next = toNext.getEnd();
218                 if (!visited.contains(next)) {
219                     visited.add(next);
220                     queue.add(next);
221                 }
222             }
223         }
224
225         //Teile Knoten in jene von S im residual Graphen erreichbare und nicht erreichbare ein
226         Vector<Vertex> setT = new Vector<>();
227         for (Vertex v : this.vertexs) {
228             if (!visited.contains(v))
229                 setT.add(v);
230         }
231
232         //Sammel alle Kanten, welche die beiden Partitionen verbinden
233         Vector<Edge> cut = new Vector<>();
234         for (Vertex v : visited) {
235             for (Vertex u : setT) {
236                 Edge temp = v.getEdge(u);
237                 if (temp != null)
238                     cut.add(temp);
239             }
240         }
241
242         return cut;
243
244
245     }
246
247     /**
248      * Find max flow
249      * @param start name of the source
250      * @param end name of the sink
251      */
252     public void findMaxFlow(String start, String end) {
253         if (start.equals(end)) {
254             System.out.println("Start == End");
255             return;
256         }
257         //Finde einen flussverbessernden Pfad
258         LinkedList<Vertex> p = this.findPath(start, end);
259
260         // Solange ein flussverbessernder Pfad existiert
261         while (p != null) {
262             int c = Integer.MAX_VALUE;
263             LinkedList<Edge> pTemp = new LinkedList<>();
264             Vertex last = p.removeFirst();
265
266             //Finde die kleinste Kapazität
267             while (!p.isEmpty()) {
268                 Vertex next = p.removeFirst();
269                 Edge edge = last.getEdge(next);
270                 c = Math.min(edge.capacity, c);
271                 pTemp.add(edge);
272                 last = next;
273             }
274
275             // aktualisiere den Durchfluss
276             while (!pTemp.isEmpty()) {
277                 Edge edge = pTemp.removeFirst();
278                 edge.capacity -= c;
```

```
279                 edge.partner.capacity += c;
280             }
281             p = this.findPath(start, end);
282         }
283     }

285     /**
286      * Erzeugt String zur grafischen Ausgabe
287      * @return String with adjacency matrix
288      */
289     @Override
290     public String toString() {
291         StringBuilder out = new StringBuilder();
292         out.append("from\\to\t");
293         for (int i = 0; i < this.vertexs.size(); i++) {
294             out.append(" ");
295             out.append(this.vertexs.get(i).toString());
296             out.append(" ");
297             out.append('\t');
298         }
299         out.append('\n');
300         for (int i = 0; i < this.vertexs.size(); i++) {
301             out.append(" ");
302             out.append(this.vertexs.get(i).toString());
303             out.append(" ");
304             out.append('\t');
305             Vertex u = this.vertexs.get(i);
306             for (int j = 0; j < this.vertexs.size(); j++) {
307                 Vertex v = this.vertexs.get(j);
308                 Edge edge = u.getEdge(v);
309                 if (i == j) {
310                     out.append(" - ");
311                     out.append('\t');
312                 } else if (edge != null && !edge.isRest) {
313                     out.append(" ");
314                     out.append(edge.CAP - edge.capacity);
315                     out.append('/');
316                     out.append(edge.CAP);
317                     out.append('\t');
318                 } else {
319                     out.append(" ");
320                     out.append('-');
321                     out.append(" ");
322                     out.append('\t');
323                 }
324             }
325             out.append('\n');
326         }
327         return out.toString();
328     }

330     /**
331      *  Prints this.toString()
332      */
333     public void print() {
334         System.out.println(this.toString());
335     }

337     ////////////////////////////////////////////////////////

339     private class Vertex {
340         Vector<Edge> edges;
341         Vector<Edge> restEdges;
342         String name;

344         /**
345          *
346          * @param name name of the vertex
347          */
348         public Vertex(String name) {
```

```
349             this.edges = new Vector<>();
350             this.restEdges = new Vector<>();
351             this.name = name;
352         }
353
354         /**
355          *
356          * @param end name of the vertex at the end
357          * @param capacity max. capacity of the edge
358          * @param used used capacity
359          */
360         public void addEdge(Vertex end, int capacity, int used) {
361             Edge edge = new Edge(this, end, capacity - used, false, capacity);
362             Edge restEdge = new Edge(end, this, used, true, capacity);
363             this.edges.add(edge);
364             end.restEdges.add(restEdge);
365             edge.setPartner(restEdge);
366             restEdge.setPartner(edge);
367         }
368
369         /**
370          *
371          * @param end name of the vertex at the end
372          * @param capacity max. capacity of the vertex
373          */
374         public void addEdge(Vertex end, int capacity) {
375             this.addEdge(end, capacity, 0);
376         }
377
378         /**
379          *
380          * @param end name of the vertex at the end
381          * @return the edge with end at the end or null
382          */
383         public Edge getEdge(Vertex end) {
384             for (Edge e : edges) {
385                 if (e.end == end)
386                     return e;
387             }
388             for (Edge e : restEdges) {
389                 if (e.end == end)
390                     return e;
391             }
392
393             return null;
394         }
395
396         /**
397          *
398          * @return hash code
399          */
400         @Override
401         public int hashCode() {
402             int hash = 17;
403             int mult = 59;
404             hash = hash * mult + name.hashCode();
405             return hash;
406
407         }
408
409         @Override
410         public boolean equals(Object o) {
411             if (o instanceof Vertex) {
412                 return this.hashCode() == o.hashCode();
413             }
414             return false;
415         }
416
417         @Override
418         public String toString() {
```

```
419             return name;
420         }
421     }

423     ////////////////////////////////////////////////////////

425     private class Edge {
426         final int CAP;
427         int capacity;
428         Vertex end;
429         Edge partner;
430         boolean isRest;
431         Vertex start;

433         /**
434          *
435          * @param start name of the vertex at the start
436          * @param end name of the vertex at the end
437          * @param capacity not used capacity of the edge
438          * @param isRest true if edge is only in the residual graph
439          * @param cap max. capacity of the edge
440          */
441         public Edge(Vertex start, Vertex end, int capacity, boolean isRest, int cap) {

443             this.capacity = capacity;
444             this.CAP = cap;
445             this.end = end;
446             this.isRest = isRest;
447             this.start = start;
448         }

450         /**
451          *
452          * @return name of the vertex at the end
453          */
454         public Vertex getEnd() {
455             return this.end;
456         }

458         /**
459          *
460          * @param partner the "twin edge" in the residual graph
461          */
462         public void setPartner(Edge partner) {
463             this.partner = partner;
464         }

466         public String toString() {
467             if (isRest)
468                 return end + "-" + start + '\t' + capacity + "/" + CAP;
469             return start + "-" + end + '\t' + capacity + "/" + CAP;
470         }

472         @Override
473         public int hashCode() {
474             int hash = 42;
475             int mult = 43;

477             hash = hash * mult + CAP;
478             hash = hash * mult + end.hashCode();
479             if (isRest) {
480                 hash = hash * mult + 1;
481             } else {
482                 hash = hash * mult + 0;
483             }
484             hash = hash * mult + start.hashCode();
485             return hash;
486         }
487     }
488 }
```

Zusatzaufgabe: ResidualGraph.java

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Vector;


public class ResidualGraph {
    //Iterierbare Liste der Knoten
    Vector<Vertex> vertexs;
    //Zuordnung Name -> Knoten
    HashMap<String, Vertex> map;

    /**
     * Constructor
     */
    public ResidualGraph() {
        this.vertexs = new Vector<>();
        this.map = new HashMap<>();
    }

    ////////////////////////////////////////////////////////

    public static void main(String... args) {
        System.out.println("Tailaufgabe b)");
        ResidualGraph test = new ResidualGraph();
        test.addVertex("s");
        for (int i = 0; i < 6; i++)
            test.addVertex(i + "");
        test.addVertex("t");
        test.addEdge("0", "1", 5, 5);
        test.addEdge("0", "3", 7, 7);
        test.addEdge("1", "t", 11, 5);
        test.addEdge("2", "3", 13, 6);
        test.addEdge("2", "5", 7, 3);
        test.addEdge("3", "t", 13, 13);
        test.addEdge("4", "5", 5, 0);
        test.addEdge("5", "t", 11, 3);
        test.addEdge("s", "0", 13, 12);
        test.addEdge("s", "2", 13, 9);
        test.addEdge("s", "4", 13, 0);
        System.out.println("Ausgangssituation");
        test.print();

        test.findMaxFlow("s", "t");
        System.out.println("Ergebnis");

        test.print();

        System.out.println("Tailaufgabe c)");
        System.out.println("Minimaler Schnitt");
        Vector<Edge> cut = test.findMinCut("s", "t");
        //int sum = 0;
        for (Edge e : cut) {
            System.out.println(e.toString());
            //sum += e.capacity;
        }

        System.out.println("Tailaufgabe d)");
        test = new ResidualGraph();
        test.addVertex("s");
        for (int i = 0; i < 6; i++)
            test.addVertex(i + "");
        test.addVertex("t");
        test.addEdge("0", "1", 1);
        test.addEdge("0", "3", 1);
        test.addEdge("1", "t", 1);
        test.addEdge("2", "3", 1);
        test.addEdge("2", "5", 1);
```

```
69          test.addEdge("3", "t", 1);
70          test.addEdge("4", "5", 1);
71          test.addEdge("5", "t", 1);
72          test.addEdge("s", "0", 1);
73          test.addEdge("s", "2", 1);
74          test.addEdge("s", "4", 1);
75          System.out.println("Ausgangssituation");

77          test.print();

79          test.findMaxFlow("s", "t");
80          System.out.println("Ergebnis");

82          test.print();

84          System.out.println("Minimaler Schnitt");
85          cut = test.findMinCut("s", "t");
86          //sum = 0;
87          for (Edge e : cut) {
88              System.out.println(e.toString());
89              //sum += e.capacity;
90          }
91      }

93      ////////////////////////////////////////////////////////

95      /**
96       *
97       * @param name name of the vertex
98       * @return true if successful, false if name is already in use
99       */
100     public boolean addVertex(String name) {
101         Vertex u = new Vertex(name);

103         //Prüfe, ob Name bereits vergeben
104         if (this.map.containsKey(name))
105             return false;

107         this.map.put(name, u);
108         this.vertexs.add(u);
109         return true;
110     }

112     /**
113      * Erzeugt Kante ohne vorhandenen Fluss
114      * @param start name of the start vertex
115      * @param end name of the end vertex
116      * @param capacity max. capacity of the edge
117      */
118     public void addEdge(String start, String end, int capacity) {
119         map.get(start).addEdge(map.get(end), capacity);
120     }

122     /**
123      * Erzeugt Kante mit Fluss
124      * @param start name of the start vertex
125      * @param end name of the end vertex
126      * @param capacity max. capacity of the edge
127      * @param used used capacity of the edge
128      */
129     public void addEdge(String start, String end, int capacity, int used) {
130         map.get(start).addEdge(map.get(end), capacity, used);
131     }

133     /**
134      * Tiefensuche, welche einen Pfad von start nach end sucht
135      * @param start name of source
136      * @param end name of the sink
137      * @return LinkedList with a path from source to sink or null if no path exists
138      */
```

```
139     private LinkedList<Vertex> findPath(String start, String end) {
140         if (start.equals(end))
141             return new LinkedList<>();
142
143         //Tiefensuche
144         HashSet<Vertex> visited = new HashSet<>();
145         LinkedList<Vertex> stack = new LinkedList<>();
146         HashMap<Vertex, Vertex> pi = new HashMap<>();
147
148         Vertex origin = this.map.get(start);
149         Vertex sink = this.map.get(end);
150
151         visited.add(origin);
152         pi.put(origin, null);
153
154         stack.push(origin);
155
156         while (stack.size() > 0) {
157             Vertex u = stack.pop();
158             Vector<Edge> edges = u.edges;
159             edges.addAll(u.restEdges);
160             for (int i = 0; i < edges.size(); i++) {
161                 Edge toNext = edges.get(i);
162                 if (toNext.capacity <= 0)
163                     continue;
164                 Vertex next = toNext.getEnd();
165                 if (!visited.contains(next)) {
166                     visited.add(next);
167                     pi.put(next, u);
168                     stack.push(next);
169                 }
170
171                 //Stoppe, wenn sink erreicht
172                 if (next == sink) {
173                     LinkedList<Vertex> temp = new LinkedList<>();
174                     temp.push(next);
175                     while (pi.get(next) != null) {
176                         next = pi.get(next);
177                         temp.push(next);
178                     }
179                     return temp;
180                 }
181             }
182         }
183         return null;
184     }
185
186     /**
187      * Suche des minimalen Schnitts über eine Breitensuche
188      * @param start name of the source node
189      * @param end name of the sink node
190      * @return A Vector of the edges, which are connecting partitions S and T.
191      */
192     public Vector<Edge> findMinCut(String start, String end) {
193         if (start.equals(end))
194             return new Vector<>();
195
196         //Erzeuge maximalen Fluss
197         this.findMaxFlow(start, end);
198
199         //Breitensuche
200         HashSet<Vertex> visited = new HashSet<>(); //entspricht S
201         LinkedList<Vertex> queue = new LinkedList<>();
202
203         Vertex origin = this.map.get(start);
204
205         visited.add(origin);
206
207         queue.add(origin);
```

```
209         while (queue.size() > 0) {
210             Vertex u = queue.poll();
211             Vector<Edge> edges = u.edges;
212             edges.addAll(u.restEdges);
213             for (int i = 0; i < edges.size(); i++) {
214                 Edge toNext = edges.get(i);
215                 if (toNext.capacity <= 0)
216                     continue;
217                 Vertex next = toNext.getEnd();
218                 if (!visited.contains(next)) {
219                     visited.add(next);
220                     queue.add(next);
221                 }
222             }
223         }
224
225         //Teile Knoten in jene von S im residual Graphen erreichbare und nicht erreichbare ein
226         Vector<Vertex> setT = new Vector<>();
227         for (Vertex v : this.vertexs) {
228             if (!visited.contains(v))
229                 setT.add(v);
230         }
231
232         //Sammel alle Kanten, welche die beiden Partitionen verbinden
233         Vector<Edge> cut = new Vector<>();
234         for (Vertex v : visited) {
235             for (Vertex u : setT) {
236                 Edge temp = v.getEdge(u);
237                 if (temp != null)
238                     cut.add(temp);
239             }
240         }
241
242         return cut;
243
244
245     }
246
247     /**
248      * Find max flow
249      * @param start name of the source
250      * @param end name of the sink
251      */
252     public void findMaxFlow(String start, String end) {
253         if (start.equals(end)) {
254             System.out.println("Start == End");
255             return;
256         }
257         //Finde einen flussverbessernden Pfad
258         LinkedList<Vertex> p = this.findPath(start, end);
259
260         // Solange ein flussverbessernder Pfad existiert
261         while (p != null) {
262             int c = Integer.MAX_VALUE;
263             LinkedList<Edge> pTemp = new LinkedList<>();
264             Vertex last = p.removeFirst();
265
266             //Finde die kleinste Kapazität
267             while (!p.isEmpty()) {
268                 Vertex next = p.removeFirst();
269                 Edge edge = last.getEdge(next);
270                 c = Math.min(edge.capacity, c);
271                 pTemp.add(edge);
272                 last = next;
273             }
274
275             // aktualisiere den Durchfluss
276             while (!pTemp.isEmpty()) {
277                 Edge edge = pTemp.removeFirst();
278                 edge.capacity -= c;
```

```
279                 edge.partner.capacity += c;
280             }
281             p = this.findPath(start, end);
282         }
283     }

285     /**
286      * Erzeugt String zur grafischen Ausgabe
287      * @return String with adjacency matrix
288      */
289     @Override
290     public String toString() {
291         StringBuilder out = new StringBuilder();
292         out.append("from\\to\t");
293         for (int i = 0; i < this.vertexs.size(); i++) {
294             out.append(" ");
295             out.append(this.vertexs.get(i).toString());
296             out.append(" ");
297             out.append('\t');
298         }
299         out.append('\n');
300         for (int i = 0; i < this.vertexs.size(); i++) {
301             out.append(" ");
302             out.append(this.vertexs.get(i).toString());
303             out.append(" ");
304             out.append('\t');
305             Vertex u = this.vertexs.get(i);
306             for (int j = 0; j < this.vertexs.size(); j++) {
307                 Vertex v = this.vertexs.get(j);
308                 Edge edge = u.getEdge(v);
309                 if (i == j) {
310                     out.append(" - ");
311                     out.append('\t');
312                 } else if (edge != null && !edge.isRest) {
313                     out.append(" ");
314                     out.append(edge.CAP - edge.capacity);
315                     out.append('/');
316                     out.append(edge.CAP);
317                     out.append('\t');
318                 } else {
319                     out.append(" ");
320                     out.append('-');
321                     out.append(" ");
322                     out.append('\t');
323                 }
324             }
325             out.append('\n');
326         }
327         return out.toString();
328     }

330     /**
331      *  Prints this.toString()
332      */
333     public void print() {
334         System.out.println(this.toString());
335     }

337     ////////////////////////////////////////////////////////

339     private class Vertex {
340         Vector<Edge> edges;
341         Vector<Edge> restEdges;
342         String name;

344         /**
345          *
346          * @param name name of the vertex
347          */
348         public Vertex(String name) {
```

```
349             this.edges = new Vector<>();
350             this.restEdges = new Vector<>();
351             this.name = name;
352         }

354         /**
355          *
356          * @param end name of the vertex at the end
357          * @param capacity max. capacity of the edge
358          * @param used used capacity
359          */
360         public void addEdge(Vertex end, int capacity, int used) {
361             Edge edge = new Edge(this, end, capacity - used, false, capacity);
362             Edge restEdge = new Edge(end, this, used, true, capacity);
363             this.edges.add(edge);
364             end.restEdges.add(restEdge);
365             edge.setPartner(restEdge);
366             restEdge.setPartner(edge);
367         }

369         /**
370          *
371          * @param end name of the vertex at the end
372          * @param capacity max. capacity of the vertex
373          */
374         public void addEdge(Vertex end, int capacity) {
375             this.addEdge(end, capacity, 0);
376         }

378         /**
379          *
380          * @param end name of the vertex at the end
381          * @return the edge with end at the end or null
382          */
383         public Edge getEdge(Vertex end) {
384             for (Edge e : edges) {
385                 if (e.end == end)
386                     return e;
387             }
388             for (Edge e : restEdges) {
389                 if (e.end == end)
390                     return e;
391             }

393             return null;
394         }

396         /**
397          *
398          * @return hash code
399          */
400         @Override
401         public int hashCode() {
402             int hash = 17;
403             int mult = 59;
404             hash = hash * mult + name.hashCode();
405             return hash;

407         }

409         @Override
410         public boolean equals(Object o) {
411             if (o instanceof Vertex) {
412                 return this.hashCode() == o.hashCode();
413             }
414             return false;
415         }

417         @Override
418         public String toString() {
```

```
419             return name;
420         }
421     }

423     //////////////////////////////////////////////////////

425     private class Edge {
426         final int CAP;
427         int capacity;
428         Vertex end;
429         Edge partner;
430         boolean isRest;
431         Vertex start;

433         /**
434          *
435          * @param start name of the vertex at the start
436          * @param end name of the vertex at the end
437          * @param capacity not used capacity of the edge
438          * @param isRest true if edge is only in the residual graph
439          * @param cap max. capacity of the edge
440          */
441         public Edge(Vertex start, Vertex end, int capacity, boolean isRest, int cap) {

443             this.capacity = capacity;
444             this.CAP = cap;
445             this.end = end;
446             this.isRest = isRest;
447             this.start = start;
448         }

450         /**
451          *
452          * @return name of the vertex at the end
453          */
454         public Vertex getEnd() {
455             return this.end;
456         }

458         /**
459          *
460          * @param partner the "twin edge" in the residual graph
461          */
462         public void setPartner(Edge partner) {
463             this.partner = partner;
464         }

466         public String toString() {
467             if (isRest)
468                 return end + "-" + start + '\t' + capacity + "/" + CAP;
469             return start + "-" + end + '\t' + capacity + "/" + CAP;
470         }

472         @Override
473         public int hashCode() {
474             int hash = 42;
475             int mult = 43;

477             hash = hash * mult + CAP;
478             hash = hash * mult + end.hashCode();
479             if (isRest) {
480                 hash = hash * mult + 1;
481             } else {
482                 hash = hash * mult + 0;
483             }
484             hash = hash * mult + start.hashCode();
485             return hash;
486         }
487     }
488 }
```