Graphenalgorithmen: Blatt 6

Lukas Kalbertodt, Elena Resch, Mirko Wagner

29. Mai 2015

A12	A13	Σ	P4
3/8	5/6	7/14	9,5/10

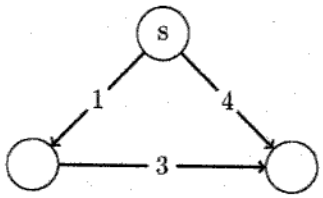
Aufgabe 12

(a) Angenommen es gibt eine Weglänge $d[i]$ die die Bellmannschen Gleichungen nicht erfüllt, d.h. es gibt ein j für das gilt $d'[i] = d[j] + c_{ij} < d[i]$ und sei k der Knoten, der auf dem Weg mit der Weglänge $d'[i]$ von s zu i vor i kam. Dann gilt $d'[i] = d[j] + c_{ji} < d[k] + c_{ki} = d'[i]$ womit $d[i]$ nicht die kürzeste Weglänge für i ist, was der Vorraussetzung widerspricht.

(b) Angenommen es gibt einen Knoten i für den es einen Weg mit der Weglänge $d'[i]$ gibt, mit $d'[i] < d[i]$. Dann gibt es ein j für das gilt $d[j] + c_{ij} = d'[i] < d[i] = \min_{(i,j)\in E} d[i] + c_{ij}$, was der Vorraussetzung, dass alle Markierungen die Bellmannschen Gleichungen erfüllen, widerspricht.
Wenn G Kreise mit Kosten 0 enthält, gilt die Aussage immernoch, jedoch ist eine optimale Markierung nun in jedem Fall nicht mehr eindeutig.

Aufgabe 13:

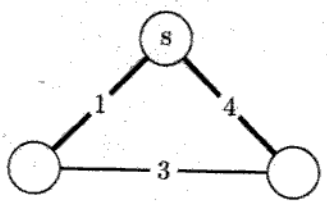
(a) Der Kürzeste-Wege-Baum ist nicht eindeutig bestimmt. Die Kantenkosten sind zwar alle unterschiedlich, aber d.h. nicht, dass es nicht zwei Wege zum selben Knoten mit den selben Kosten geben kann. Beispiel:



(b) Offentsichtlich ist erstmal jeder Kürzeste-Wege-Baum ein Spannbaum. Dieser Spannbaum ist aber nicht immer minimal. Im folgenden Beispiel sind die markierten Kanten zwar ein gültiger Kürzeste-Wege-Baum, aber kein minimaler Spannbaum:

$-\frac{1}{2}$

Und was ist der MST in diesem Fall?



```
397                 print("graph not connected.");
398                 return null;
399             }
400             int i = minimum.node;
401             s.add(i);
402             d[i] = minimum.weight;
403             edges.add(new Edge(pred[i], i, g.getWeight(pred[i], i)));
404
405             // update the successors from i
406             List<Integer> n = g.getSuccessors(i);
407             for (int k = 0; k < n.size(); k++) {
408                 int j = n.get(k);
409                 if (!s.contains(j)) {
410                     double c_ij = g.getWeight(i, j);
411                     if (d[j] > (d[i] + c_ij)) {
412                         d[j] = d[i] + c_ij;
413                         pred[j] = i;
414                         Edge edge = new Edge(i, j, d[j]);
415                         // both methods for updateHeap
416                         heap.remove(edge);
417                         heap.add(edge);
418                     }
419                 }
420             }
421         }
422
423         // create the graph
424         GraphImpl dijkstra = new GraphImpl(true, true);
425         for (int i = 0; i < vertexcnt; i++) {
426             dijkstra.addVertex();
427         }
428         for (int i = 0; i < edges.size(); i++) {
429             Edge tmp = edges.get(i);
430             dijkstra.addEdge(tmp.pred, tmp.node,
431                     g.getWeight(tmp.pred, tmp.node));
432         }
433
434         // print all paths
435         printPaths(pred, d, start);
436         return dijkstra;
437     }
438
439     /**
440      * main-function to start the whole process of dijkstra
441      *
442      * @param args
443      */
444     public static void main(String[] args) {
445         if (args.length != 1) {
446             print("java -jar Dijkstra.jar <filename>");
447             return;
448         }
```



```
449         createDijkstraFromFile(args[0]);
450     }
451
452 }
453
```

P4: 9.5/10

```
296      *
297      * @param args
298      *            filename of the .gra-file to be read
299      */
300     private static void createDijkstraFromFile(String args) {
301         print("reading " + args + ".");
302         GraphImpl graph = readGraFile(args);
303         if (graph != null) {
304             String filename = graToPngString(args);
305             try {
306                 RenderGraph.renderGraph(graph, filename);
307                 print(filename + " created.");
308                 print("--------------------------------");
309                 print("dijktra: ");
310                 GraphImpl dijkstra = dijkstra(graph, 0);
311                 if (dijkstra != null) {
312                     RenderGraph.renderGraph(dijkstra, "shortest_path_"
313                             + filename);
314                     print("shortest_path_" + filename + " created.");
315                 }
316             } catch (IOException e) {
317                 print("file could not be created");
318             }
319         } else {
320             print("error in creating the graph.");
321         }
322         print("Program exit.....");
323     }
324
325     /**
326      * implementation of dijkstra algorithm for the single-source shortest
    path
327      * problem
328      *
329      * @param g
330      *            directed and weighted graph
331      * @param start
332      *            vertex to be started from
333      * @return 'reduced" graph to be rendered
334      */
335     private static GraphImpl dijkstra(GraphImpl g, int start) {
336         if (g == null) {
337             print("g is null");
338             return null;
339         }
340         if (!(g instanceof Graph)) {
341             print("g is not instance of Graph");
342             return null;
343         }
344         if (g.getNodeCount() == 0) {
345             print("g has no nodes");
346             return new GraphImpl(true, true);
```



```
347         }
348         if (start >= g.getNodeCount()) {
349             print("start node not in g");
350             return null;
351         }
352
353         int vertexcnt = g.getNodeCount();
354
355         Set<Integer> s = new HashSet<Integer>(); // set of vertices
356         double[] d = new double[vertexcnt]; // distances
357         int[] pred = new int[vertexcnt]; // predecessors
358         List<Edge> edges = new ArrayList<Edge>(); // list of edges to
    create
359                                                   // graph
360         // heap with specified comparator for weights
361         PriorityQueue<Edge> heap = new PriorityQueue<>(vertexcnt,
362                 new Comparator<Edge>() {
363
364                     @Override
365                     public int compare(Edge e1, Edge e2) {
366                         if (e1.weight < e2.weight)
367                             return -1;
368                         if (e1.weight > e2.weight)
369                             return 1;
370                         return 0;
371                     }
372                 });
373
374         // init distances to infinity
375         for (int i = 0; i < vertexcnt; i++) {
376             d[i] = Double.MAX_VALUE;
377         }
378
379         // start of dijkstra
380         s.add(start);
381         d[start] = 0;
382         List<Integer> neighbors = g.getSuccessors(start);
383         // add all neighbors to the heap
384         for (int i = 0; i < neighbors.size(); i++) {
385             int n = neighbors.get(i);
386             d[n] = g.getWeight(start, n);
387             pred[n] = start;
388             heap.add(new Edge(0, n, d[n]));
389         }
390
391         // while-loop
392         while (s.size() != vertexcnt) {
393             // get a node that has the minimal distance to the currently
    built
394             // tree
395             Edge minimum = heap.poll();
396             if (minimum == null) {
```

```
195         return null;
196     }
197
198     /**
199      * helper function to convert the numbers
200      *
201      * @param line
202      * @param number
203      * @return
204      */
205     private static double[] convertLine(String line, int number) {
206
207         double[] values;
208         line.trim();
209         String[] splitted = line.split(" ");
210         if (splitted.length != number) {
211             System.out
212                     .println("number of vertices is not equal to matrix
size");
213             return null;
214         }
215         values = new double[number];
216         for (int i = 0; i < number; i++) {
217             try {
218                 values[i] = Double.parseDouble(splitted[i]);
219                 if (values[i] < 0) {
220                     System.out.println("weight negative");
221                 }
222             } catch (NumberFormatException nfe) {
223                 System.out.println("file has wrong format");
224                 return null;
225             }
226         }
227         return values;
228
229     }
230
231     /**
232      * replaces the .gra suffix with .png
233      *
234      * @param grafile
235      *            string = "*.gra"
236      * @return *.png filename
237      */
238     private static String graToPngString(String grafile) {
239         char[] filename = grafile.toCharArray();
240         int i = 0;
241         filename[filename.length - 1 - i++] = 'g';
242         filename[filename.length - 1 - i++] = 'n';
243         filename[filename.length - 1 - i++] = 'p';
244         StringBuffer name = new StringBuffer();
245         name.append(filename);
```



```
246         return name.toString();
247     }
248
249     /**
250      * Helper function to not write the long expression(laziness)
251      *
252      * @param str
253      *            line to be printed
254      */
255     private static void print(String str) {
256         System.out.println(str);
257     }
258
259
260     /**
261      * helper function to print all possible path
262      *
263      * @param pred
264      *            integer array of predecessors
265      * @param d
266      *            double array with the sum of distances for each vertex
from
267      *            the start
268      * @param start
269      *            vertex where dijkstra algorithm started
270      */
271     private static void printPaths(int[] pred, double[] d, int start) {
272
273         for (int i = 0; i < pred.length; i++) {
274             String path = "";
275             double costs = 0;
276             if (i != start) {
277                 int j = i;
278                 costs = d[j];
279                 if (costs == Double.MAX_VALUE) {
280                     print("c = infty: no path from " + start + " to " +
i);
281                 } else {
282                     path = " -> " + j;
283                     while (pred[j] != start) {
284                         path = " -> " + pred[j] + path;
285                         j = pred[j];
286                     }
287                 }
288             }
289             path = start + path;
290             print(String.format("c = %5.1f: %s", costs, path));
291         }
292     }
293
294     /**
295      * Function reads the file, creates a graph, and writes the .png-files
```

```
102         } else if (filename.contains("dir")) {
103             directed = true;
104         } else { // if no information is given, than graph is assumed to
    be not
105                     // directed
106             directed = true;
107         }
108 
109         try (BufferedReader rd = new BufferedReader(new FileReader(new
    File(
110                 filename)))) {
111 
112             // first line(s) is(are) a comment
113             while ((line = rd.readLine()) != null && line.startsWith("#"))
    {
114             }
115 
116             // line is null
117             if (line == null) { // wrong format, file empty
118                 rd.close();
119                 System.out.println("file has wrong format");
120                 return null;
121             }
122 
123             line.trim();
124 
125             // if regex for the count of vertices line n <number> does not
126             // matches the standard
127             if (!line.matches("n [1-9][0-9]*")) {
128                 rd.close();
129                 System.out.println("file " + filename + " has wrong
    format.");
130                 return null;
131             }
132 
133             String[] splitted = line.split(" ");
134             // convert the string number to int
135             try {
136                 vertexNumber = Integer.parseInt(splitted[1]);
137             } catch (NumberFormatException nfe) { // is not possible to
    be, as a
138                                                   // regex assures that
    the
139                                                   // right format is
    used
140                 rd.close();
141                 System.out.println("file " + filename + " has wrong
    format.");
142                 return null;
143             }
144             // ignore the comment lines
145             while ((line = rd.readLine()) != null && line.startsWith("#"))
```



```
    {
146             }
147 
148             // now must come the adjacency matrix
149             // regex: "\\d+|\\s+[ : ][\\d ]+"
150             matrix = new double[vertexNumber][];
151             for (int i = 0; i < vertexNumber; i++) {
152                 line.trim();
153                 if ((!line.matches(".+ : [[[0-9]+ ]+|[[0-9]+\\.[0-9]+ ]+]
    +"))) {
154 
155                     System.out.println("file " + filename
156                             + " has wrong format.");
157                     rd.close();
158                     return null;
159                 }
160                 // if everything ok, split the line on ":"
161                 splitted = line.split(" : ", 2);
162 
163                 // get the double[] of weights.
164                 matrix[i] = convertLine(splitted[1], vertexNumber);
165                 if (matrix[i] == null) {
166                     rd.close();
167                     return null;
168                 }
169                 line = rd.readLine();
170             }
171 
172         } catch (FileNotFoundException e) {
173             System.out.println(filename + " not found.");
174             return null;
175         } catch (IOException e) {
176             System.out.println("error while opening/reading" + filename +
    ".");
177         }
178 
179         if (matrix != null) {
180             // printMatrix(matrix);
181             // matrix read, instantiate the graph
182             try {
183                 graph = new GraphImpl(directed, weighted, matrix);
184 
185             } catch (IllegalArgumentException iae) {
186                 System.out.println("1. wrong data in file " + filename);
187                 return null;
188             } catch (RuntimeException re) {
189                 System.out.println("2. wrong data in file " + filename);
190                 return null;
191             }
192             return graph;
193 
194         }
```

```java

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

import renderGraph.RenderGraph;
import P1.Graph;
import P1.GraphImpl;

/**
 *
 * @author Elena Resch
 * @author Lukas Kalbertodt
 * @author Mirko Wagner
 *
 */

public class Dijkstra {

    /**
     * internal representation of an edge, most important is the overwritten
     * equals-method
     */
    private static class Edge {
        int pred, node; // node numbers
        double weight; // weight of this edge

        /**
         * C-tor
         *
         * @param pred
         *            predecessor of node
         * @param node
         *            vertexnumber
         * @param weight
         *            double value >= 0.0, no checking of value
         */
        public Edge(int pred, int node, double weight) {
            this.pred = pred;
            this.node = node;
            this.weight = weight;
        }

```



```java
        /**
         * for the heap-remove and -add-method as updateHeap
         */
        public boolean equals(Object obj) {
            if (obj == null) {
                return false;
            }
            if (!(obj instanceof Edge)) {
                return false;
            }
            Edge tmp = (Edge) obj;
            if (this.node == tmp.node) {
                return true;
            }
            return false;
        }

        public String toString() {
            String tmp = "((" + pred + ", " + node + "), " + weight + ")";
            return tmp;
        }
    }

    /**
     * checks if the given filename is a .gra-file and reads its content while
     * checking for correct .gra-file format
     *
     * @param filename
     *            *.gra-file
     * @return graph if file could be read, null on error
     */
    private static GraphImpl readGraFile(String filename) {
        // check if filename is a .gra-file
        if (!filename.endsWith(".gra")) {
            System.out.println(filename
                    + " has wrong suffix.  only '*.gra'-files allowed");
            return null;
        }
        boolean directed, weighted;
        String line;
        int vertexNumber;
        double[][] matrix = null;
        GraphImpl graph;
        directed = true;
        weighted = true;

        // check if filename gives information whether graph is directed or
        // undirected
        if (filename.contains("undir")) {
            directed = false;
```