Graphenalgorithmen: Blatt 3

Lukas Kalbertodt, Elena Resch, Mirko Wagner

7. Mai 2015

A6	A7	Σ	P1
10/10	7,5/8	17,5/18	12,5/15

Aufgabe 6:

(a) Es handelt sich um einen vollständigen Graphen mit Schlingen mit $V = \{1, \dots, k\}$ als Knoten- und $E := \{(i, j) : i \leq j, \quad i, j \in V\}$ als Kantenmenge. Das Problem entspricht dem Euler-Pfad auf vollständigen Graphen. Die Knoten des Graphen repräsentieren die Zahl, die gerade am Ende der Kette liegt. Eine Kante verbindet immer zwei Zahlen und stellt somit einen Stein dar. Jeder Knoten hat eine Schleife, die aber irrelevant für das Euler-Pfad Problem ist. Der Graph ist vollständig, weil es für jede Kombination von zwei Zahlen $\leq k$ genau einen Spielstein gibt. Daher muss man auch jede Kante exakt einmal besuchen.

(b) Das Problem mit $k = 2, 3$ ist lösbar, für $k = 4$ nicht. $k = 2$ ist dabei ein Sonderfall und eine Ausnahme für die in (c) genannte Regel.

(c) Vollständige Graphen besitzen immer genau dann einen Eulerweg, wenn sie eine ungerade Anzahl von Knoten k haben, weil jeder Knoten $k - 1$ Kanten besitzt und für den Eulerpfad 0 oder 2 Knoten ungeraden Knotengrad haben müssen (Skript Satz 1.3). Der Graph mit $k = 2$ bildet dabei eine Ausnahme, denn seine zwei Knoten besitzen alle Knotengrad 3, da Schlingen doppelt gezählt werden, während für $k > 2$, k gerade gilt, dass alle k Knoten $deg(i) = k - 1, i \in V$, ungerade ist.

(d) Eine geschlossene Kette entspricht einem Eulerkreis im Graphen. Da in dem vollständigen Graphen *alle* Knoten den selben Grad $k - 1$ haben, ist ein Eulerkreis immer dann möglich, wenn der Knotengrad bei allen Knoten gerade ist, also $k - 1$ gerade, insgesamt also k ungerade ist (Skript Satz 1.2).

Aufgabe 7:

Ein „Heap“ ist eine Datenstruktur, die als PriorityQueue genutzt werden kann. Eine PriorityQueue verwaltet eine Menge von Items bestehend aus `key` und `data`. Die PriorityQueue soll jetzt in der Lage sein, möglichst schnell das Objekt mit der höchsten Priorität (niedrigstem Key) zu finden, zurückzugeben und aus der PriorityQueue zu löschen. Die gewünschten Funktionen des ADTs PriorityQueue:

- `min() -> item*`: Gibt eine Referenz auf das Item mit dem kleinsten Key zurück

- `deleteMin() -> item`: Findet und extrahiert das Item mit dem kleinsten Key
- `insert(item)`: Fügt ein Item ein
- `decreaseKey(item*, newValue)`: Verringert den Key eines bestimmten Items
- `delete(item*)`: Löscht ein Item
- `merge(Heap)`: vereint den Heap mit einem anderen (wird nur z.t. gefordert)

„Heap" ist jetzt der Überbegriff für mehrere Datenstrukturen, die die oben genannten Funktionen schnell ausführen können. Der einfachste und bekannteste Heap ist der BinaryHeap, der alle Items in einem binären Baum organisiert. Innerhalb des Baumes gilt die Invarianz, dass der Key in den beiden Söhnen eines Knoten j größer ist, als der Key in j. Somit ist das Element mit dem kleinsten Key in der Wurzel des Baums. Diese Variante des Heaps ist bereits recht effizient in der Praxis. Eine Abwandlung sind die sog. „d-ary Heaps", die die selbe asymptotischen Laufzeiten haben (d konstant), aber in der Praxis besser sein können. Hier werden die Items nicht in einem binären Baum, sondern in einem d-ären Baum gespeichert.

Darüber hinaus gibt es aber weitere und sehr exotische Varianten, mit teilweise deutlich besseren asymptotischen Laufzeiten. Zu erwähnen wäre beispielsweise der PairingHeap, der `insert` in $\mathcal{O}(1)$ und `decreaseKey` in armotisiert $o(log\ n)$ schafft und dabei noch praxistauglich ist. Der FibonacciHeap schafft `decreaseKey` zwar in amortisiert $\mathcal{O}(1)$, dies zahlt sich jedoch erst bei sehr großen Probleminstanzen gegenüber dem Pairing-Heap aus. Der BrodalHeap schafft sogar zusätzlich `decreaseKey` in konstanter Zeit, ist aber in der Praxis absolut untauglich.

Was die Laufzeiten angeht, so kann man eine sinnvolle untere Schranke ableiten: Das Sortierverfahren HeapSort nutzt einen Heap, um per Vergleich zu sortieren. Für vergleichbasierte Sortierverfahren ist ja bereits die untere Schranke $\Omega(n \cdot \log n)$ bekannt. Heapsort ruft je n mal die Funktionen `insert` und `deleteMin` auf. Somit muss entweder `insert` oder `deleteMin` mindestens die Laufzeit $\mathcal{O}(\log n)$ haben. Die restlichen Laufzeiten im Überblick ($*$ armortisiert):

Funktion	Binary	d-ary	Fibonacci	Pairing	Brodal
`min`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`decreaseKey`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)*$	$o(\log n)*$	$\mathcal{O}(1)$
`deleteMin`	$\mathcal{O}(\log n)$	$\mathcal{O}(d \cdot \log_d n)$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)$
`delete`	$\mathcal{O}(\log n)$	$\mathcal{O}(d \cdot \log_d n)$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)$

Implementation als Adjazenz- (-2)
matrix fehlt

P1. 12,5/15

GraphImpl.java

```java
package P1;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import renderGraph.RenderableGraph;

/**
 * Representation of a directed and undirected graph. If graph is not weighted,
 * all weights are set to 0.0. Vertex names begin from 0. Internally the graph
 * is structured as a directed weighted graph through an adjacency list which
 * gives the opportunity to the graph to grow dynamically.
 *
 * @author Elena Resch
 * @author Lukas Kalbertodt
 * @author Mirco Wagner
 *
 */
public class GraphImpl implements Graph, RenderableGraph {

    private boolean m_weighted;
    private boolean m_directed;
    private int m_vertexCount;
    private List<Map<Integer, Double>> m_adjacencylist; // internal
                                                        // representation of the
                                                        // graph

    /**
     * C-tor creates an empty graph, needs informations if the graph should be
     * directed and if it is weighted
     *
     * @param weighted
     *            true if graph should be weighted
     * @param directed
     *            true if graph should be directed
     */
    public GraphImpl(boolean directed, boolean weighted) {
        this.m_weighted = weighted;
        this.m_directed = directed;
        this.m_vertexCount = 0;
        this.m_adjacencylist = new ArrayList<Map<Integer, Double>>();
    }

    /**
```



```java
     * C-tor makes a graph from given values
     *
     * @param matrix
     *            quadratic double array
     * @param directed
     *            boolean value if the graph should be directed
     * @param weighted
     *            boolean value if the graph should be weighted
     * @throws RuntimeException
     *             if matrix is not quadratic
     * @throws IllegalArgumentException
     *             if there is a negative value in the matrix, if there is a
     *             self loop in the matrix, if undirected graph wished but
     *             matrix for a directed graph is given, if matrix is null
     */
    public GraphImpl(boolean directed, boolean weighted, double[][] matrix) {
        this(directed, weighted);
        if (matrix == null) {
            throw new IllegalArgumentException("matrix is null");
        }
        this.graphFromMatrix(matrix);
    }

    /**
     * @return true if graph is weighted, else false
     */
    @Override
    public boolean isWeighted() {
        return this.m_weighted;
    }

    /**
     * @return true if this graph is directed, else false
     */
    @Override
    public boolean isDirected() {
        return this.m_directed;
    }

    /**
     * returns the number of vertices in this graph
     *
     * @return number of vertices
     */
    @Override
    public int getNodeCount() {
        return this.m_vertexCount;
    }

    /**
```

```
98      * is forwarded to getWeightOfEdge(start, end)
99      *
100     * @param x
101     *              start vertex of the edge
102     * @param y
103     *              end vertex of the edge
104     * @return c(start, end) in a weighted graph, 0.0 in a not weighted
    graph
105     *          Double.POSITIVE_INFINITY if the edge is not graph
106     * @throws IllegalArgumentException
107     *              if wrong node number(s) is(are) given
108     */
109    @Override
110    public double getWeight(int x, int y) throws IllegalArgumentException
    {
111        return this.getWeightOfEdge(x, y);
112    }
113
114    /**
115     * is forwarded to addEdge(start, end, weight) with weight = 0.0
116     *
117     * @param start
118     *              start vertex in the graph
119     * @param end
120     *              end vertex in the graph
121     * @throws IllegalArgumentException
122     *              if node(s) is(are) not in this graph, if start equal
    end
123     *              (selfloop),if weight is negative.
124     */
125    @Override
126    public void addEdge(int start, int end) {
127        this.addEdge(start, end, 0.0);
128
129    }
130
131    /**
132     * Adds an edge to the graph.
133     *
134     * @param start
135     *              start vertex in the graph
136     * @param end
137     *              end vertex in the graph
138     * @param weight
139     *              of the edge, must be positive or 0.0
140     * @throws IllegalArgumentException
141     *              if node(s) is(are) not in this graph, if start equal
    end
142     *              (selfloop),if weight is negative.
143     */
144    @Override
145    public void addEdge(int start, int end, double weight) {
```

Wenn kein Gewicht mitgegeben wird, sollte es vllt standardmäßig besser auf 1.0 gesetzt werden


```
146         if (start < 0 || start >= this.m_vertexCount || end < 0
147                 || end >= this.m_vertexCount) {
148             throw new IllegalArgumentException("node(s) not in graph");
149         }
150         if (start == end) { // no selfloops in this kind of graph allowed
151             throw new IllegalArgumentException("no selfloops allowed");
152         }
153         if (weight < 0.0) {
154             throw new IllegalArgumentException("weight is < 0.0");
155         }
156         (this.m_adjacencylist.get(start)).put(end, weight);
157         if (!this.m_directed) {
158             (this.m_adjacencylist.get(end)).put(start, weight);
159         }
160     }
161
162     /**
163      * Adds a consecutively numbered vertex to the graph with an empty
164      * neighborhood
165      */
166     @Override
167     public void addVertex() {
168         Map<Integer, Double> newVertex = new HashMap<Integer, Double>();
169         this.m_adjacencylist.add(newVertex);
170         this.m_vertexCount++;
171     }
172
173     /**
174      * returns the whole neighborhood of this vertex
175      *
176      * @param v
177      *              vertex in the graph
178      * @return list of Integer which are neighbors of the given vertex, if
    v has
179      *          no neighbors an empty list is returned
180      *
181      * @throws IllegalArgumentException
182      *              if v is not in graph
183      */
184     @Override
185     public List<Integer> getNeighbors(int v) throws
    IllegalArgumentException {
186         List<Integer> successors = this.getSuccessors(v);
187         if (this.m_directed) {
188             List<Integer> predecessors = this.getPredecessors(v);
189             successors.addAll(predecessors);
190             return successors;
191         } else {
192             return successors;
193         }
194     }
195
```

-1/2

Achtung wenn im gerichteten Fall Hin- und Rückkante vorhanden sind → Knoten doppelt in Liste

```java
    /**
     * if the graph is undirected all predecessors are the whole neighborhood,
     * else check for all vertices in the adjacency list if there is v their
     * neighborhood, and add these vertices to the list of predecessors
     *
     * @param v
     *            vertex in the graph
     * @return list of Integers which are predecessors of the given vertex, if v
     *         has no predecessors an empty list is returned
     * @throws IllegalArgumentException
     *             if v is not in the graph
     */
    @Override
    public List<Integer> getPredecessors(int v) {
        if (v < 0 || v >= this.m_vertexCount) {
            throw new IllegalArgumentException("vertex not in graph");
        }
        if (this.m_directed) { // predecessors != successors
            List<Integer> predecessors = new ArrayList<Integer>();
            // run throw the list and check if the key is in the hashmap, if so
            // add the actual index to the list of predecessors
            for (int i = 0; i < this.m_adjacencylist.size(); i++) {
                List<Integer> successors = this.getSuccessors(i);
                if (successors.contains(new Integer(v))) {
                    predecessors.add(i);
                }
            }
            return predecessors;
        } else {
            // predecessors = successors, so just get successors
            return this.getSuccessors(v);
        }
    }

    /**
     * if this graph is directed then successors are generally not equal to
     * predecessors, but in an undirected graph successors are equal to
     * predecessors, so they are all neighbors.
     *
     * @param v
     *            vertex in the graph
     * @return list of Integers which are successors of the given vertex, if v
     *         has no successors an empty list is returned
     * @throws IllegalArgumentException
     *             if v not in the graph
     */
```



```java
    @Override
    public List<Integer> getSuccessors(int v) {
        if (v < 0 || v >= this.m_vertexCount) {
            throw new IllegalArgumentException("vertex not in graph");
        }

        // get the corresponding set of this vertex
        Map<Integer, Double> successors = this.m_adjacencylist.get(v);
        Set<Integer> set = successors.keySet();

        // return the set converted to a list
        return new ArrayList<Integer>(set);
    }

    /**
     * returns the number of vertices in this graph
     */
    @Override
    public int getVertexCount() {
        return this.m_vertexCount;
    }

    /**
     * returns the weight of the given edge
     *
     * @param start
     *            start vertex of the edge
     * @param end
     *            end vertex of the edge
     * @return c(start, end) in a weighted graph, 0.0 in a not weighted graph
     *         Double.POSITIVE_INFINITY if the edge is not graph
     * @throws IllegalArgumentException
     *             if wrong node number(s) is(are) given
     */
    @Override
    public double getWeightOfEdge(int start, int end)
            throws IllegalArgumentException {
        if (!this.hasEdge(start, end)) { // if edge is not in graph
            if (start == end) {
                return 0.0;
            }
            return Double.POSITIVE_INFINITY;
        }
        // edge is in this graph
        Map<Integer, Double> successors = this.m_adjacencylist.get(start);
        // get the corresponding value
        double value = successors.get(end);
        return value;
    }

    /**
```

```java
     * checks if the edge is in graph
     *
     * @return true if edge in graph else false
     * @throws IllegalArgumentException
     *             if given node(s) is(are) not in graph
     */
    @Override
    public boolean hasEdge(int start, int end) {
        if (start < 0 || start >= this.m_vertexCount || end < 0
                || end >= this.m_vertexCount) {
            throw new IllegalArgumentException("node(s) not in graph");
        }
        if (start == end) {
            return false;
        }
        List<Integer> successors = this.getSuccessors(start);
        if (successors.contains(new Integer(end))) {
            return true;
        }
        return false;
    }

    /**
     * removes the edge from the graph
     *
     * @param start
     *             number of the start vertex
     * @param end
     *             number of the start vertex
     * @throws IllegalArgumentException
     *             if node(s) is(are) not in graph
     */
    @Override
    public void removeEdge(int start, int end) throws
IllegalArgumentException {
        if (!this.hasEdge(start, end)) {
            return;
        }
        Map<Integer, Double> successors = this.m_adjacencylist.get(start);
        successors.remove(end);
        this.m_adjacencylist.set(start, successors);
        if (!this.m_directed) {
            successors = this.m_adjacencylist.get(end);
            successors.remove(start);
            this.m_adjacencylist.set(end, successors);
        }
    }

    /**
     * removes the vertex with the last number. If graph has no vertices
nothing
     * happens
```



```java
     */
    @Override
    public void removeVertex() {
        if (!this.m_adjacencylist.isEmpty()) {
            this.m_adjacencylist.remove(--this.m_vertexCount);
        }
    }

    /**
     * returns graph information as a 2D-double-matrix
     *
     * @return 2D-double-matrix with weights in it, if graph is not
weighted the
     *         edges in the graph are represented through 1.0 value
     */
    public double[][] getMatrix() {
        double matrix[][] = new double[this.m_vertexCount]
[this.m_vertexCount];
        // fill the matrix with weights
        for (int i = 0; i < this.m_vertexCount; i++) {
            Map<Integer, Double> neighbors = this.m_adjacencylist.get(i);
            for (int j = 0; j < this.m_vertexCount; j++) {
                if (i == j) {
                    matrix[i][j] = 0.0;
                } else if (neighbors.containsKey(j)) {
                    if (this.m_weighted) {
                        matrix[i][j] = neighbors.get(j);
                    } else {
                        matrix[i][j] = 1.0;
                    }
                } else {
                    matrix[i][j] = Double.POSITIVE_INFINITY;
                }
            }
        }
        return matrix;
    }

    /**
     * makes a graph from given values, default values are undirected and
     * weighted
     *
     * @param matrix
     *             quadratic double array
     * @xhrows RuntimeException
     *             if matrix is not quadratic
     * @throws IllegalArgumentException
     *             if there is a negative value in the matrix, if there is
a
     *             self loop in the matrix if undirected graph wished but
matrix
     *             for a directed graph is given
```

```
391     */
392    private void graphFromMatrix(double[][] matrix) {
393
394        for (int i = 0; i < matrix.length; i++) {
395            this.addVertex();
396        }
397        for (int i = 0; i < matrix.length; i++) {
398            // create new vertex
399            for (int j = 0; j < matrix[i].length; j++) {
400                // check if matrix is quadratic
401                if (matrix.length != matrix[i].length) {
402                    throw new RuntimeException("matrix is not quadratic");
403                }
404                // check values in the matrix
405                if (matrix[i][j] < 0.0) {
406                    throw new IllegalArgumentException(
407                            "negative value in the matrix");
408                }
409                if (i == j && matrix[i][j] != 0.0) {
410                    throw new IllegalArgumentException(
411                            "selfloops in graph are not allowed.");
412                }
413                // check values if an edge exists
414                if (matrix[i][j] > 0.0) {
415                    this.addEdge(i, j, matrix[i][j]);
416                }
417            }
418        }
419        // check if a matrix for an undirected graph is given
420        if (!this.m_directed) {
421            for (int i = 0; i < matrix.length; i++) {
422                for (int j = i; j < matrix.length; j++) {
423                    if (matrix[i][j] != matrix[j][i]) {
424                        throw new IllegalArgumentException(
425                                "given matrix is directed");
426                    }
427                }
428            }
429        }
430    }
431
432    /**
433     * prints the 2D-double-matrix
434     */
435    public void printMatrix() {
436        double matrix[][] = this.getMatrix();
437
438        System.out.print("_| ");
439        for (int i = 0; i < matrix.length; i++) {
440            System.out.print("_" + i + "__");
441        }
442        System.out.println();
```

Fehlerbehandlung doppelt da auch in addEdge vorhanden, aber ok


```
443        for (int i = 0; i < matrix.length; i++) {
444            System.out.print(i + "| ");
445            for (int j = 0; j < matrix[i].length; j++) {
446                if (matrix[i][j] == Double.POSITIVE_INFINITY) {
447                    System.out.print(" x  ");
448                } else {
449                    if (this.m_weighted) {
450                        System.out.print(matrix[i][j] + " ");
451                    } else {
452                        System.out.println("1.0 ");
453                    }
454                }
455            }
456            System.out.println();
457        }
458    }
459
460    /**
461     * prints the adjacency list with (key, value)-pairs if graph is
    weighted,
462     * else only key
463     */
464    public void printList() {
465        for (int i = 0; i < this.m_adjacencylist.size(); i++) {
466            Map<Integer, Double> entries = this.m_adjacencylist.get(i);
467            System.out.print("list(" + i + ") = {");
468            for (Map.Entry<Integer, Double> entry : entries.entrySet()) {
469                if (this.m_weighted) {
470                    System.out.print("(" + entry.getKey() + ", "
471                            + entry.getValue() + ")");
472                } else {
473                    System.out.print(entry.getKey() + " ");
474                }
475            }
476            System.out.println("}");
477        }
478    }
479}
480
```