

Die Graphschaft Schilda

Felix Möhler und Julian Thiele

Felix Möhler und Julian Thiele

© 2022 Felix Möhler und Julian Thiele

Inhaltsverzeichnis

1. Die Graphschaft Schilda	4
1.1 Abstract	4
1.2 Aufgabenstellung	4
1.3 Planungstool Aufbau	4
1.4 Das Team	5
1.5 Auftraggeber	5
2. Problem 1 - "Straßen müssen her!"	6
2.1 Modellierung des Problems	6
2.2 Die Eingabe	6
2.3 Die Ausgabe	6
2.4 Geeignete Algorithmen	7
2.5 Die Laufzeit des Algorithmus	7
2.6 Die Implementierung des Algorithmus	8
3. Problem 2 - "Wasserversorgung"	9
3.1 Modellierung des Problems	9
3.2 Die Eingabe	9
3.3 Die Ausgabe	9
3.4 Der Algorithmus	10
3.5 Die Laufzeit des Algorithmus	10
3.6 Die Implementierung des Algorithmus	10
4. Problem 3 - "Stromversorgung"	11
4.1 Modellierung des Problems	11
4.2 Die Eingabe	11
4.3 Die Ausgabe	11
4.4 Der Algorithmus	12
4.5 Die Laufzeit des Algorithmus	12
4.6 Die Implementierung des Algorithmus	13
5. Problem 4 - "Historische Funde"	14
5.1 Modellierung des Problems	14
5.2 Die Eingabe	14
5.3 Die Ausgabe	15
5.4 Der Algorithmus	15
5.5 Die Laufzeit des Algorithmus	15
5.6 Die Implementierung des Algorithmus	16

6. Problem 5 - "Die Festhochzeit - das Verteilen der Einladungen"	17
6.1 Modellierung des Problems	17
6.2 Die Eingabe	17
6.3 Die Ausgabe	17
6.4 Der Algorithmus	17
6.5 Die Laufzeit des Algorithmus	17
6.6 Die Implementierung des Algorithmus	17
7. Problem 6 - "Wohin nur mit den Gästen?"	18
7.1 Modellierung des Problems	18
7.2 Die Eingabe	18
7.3 Die Ausgabe	18
7.4 Der Algorithmus	19
7.5 Die Laufzeit des Algorithmus	19
7.6 Die Implementierung des Algorithmus	19
8. Problem 7 - "Es gibt viel zu tun! Wer macht's"	20
8.1 Modellierung des Problems	20
8.2 Die Eingabe	20
8.3 Die Ausgabe	20
8.4 Der Algorithmus	21
8.5 Die Laufzeit des Algorithmus	21
8.6 Die Implementierung des Algorithmus	21

1. Die Graphschaft Schilda

1.1 Abstract

Diese Website ist die Dokumentation des Projektes "Graphschaft Schilda" für das Modul Programmiertechnik III an der TH Aschaffenburg.

Die Graphschaft Schilda ist ein beschauliches Örtchen irgendwo im Nichts. Lange Zeit blieb diese Graphschaft unbehelligt vom Fortschritt, nichts tat sich in dem Örtchen. Eines Tages jedoch machte sich dort plötzlich das Gerücht breit, dass fernab der Graphschaft intelligente Menschen leben, die (fast) alle Probleme der Welt mit mächtigen Algorithmen lösen könnten. Die Bürger der Graphschaft machten sich also auf den Weg um diese intelligenten Menschen mit der Lösung ihrer Probleme zu beauftragen....

1.2 Aufgabenstellung

Entwickeln Sie ein Planungstool, dass der Graphschaft Schilda bei der Lösung ihrer Probleme hilft.

1. Analysieren Sie jedes der Probleme: Welche Daten sollen verarbeitet werden? Was sind die Eingaben? Was die Ausgaben? Welcher Algorithmus eignet sich? Welche Datenstruktur eignet sich?
2. Implementieren Sie den Algorithmus (in Java), so dass bei Eingabe der entsprechenden Daten die gewünschte Ausgabe berechnet und ausgegeben wird.
3. Geben Sie für jeden implementierten Algorithmus die Laufzeit an. Da Sie sich nun schon so viel Mühe mit dem Tool geben, wollen Sie das Tool natürlich auch an andere Gemeinden verkaufen. Die Eingaben sollen dafür generisch, d.h., für neue Orte, Feiern und Planungen anpassbar sein. Sie können diese Aufgabe ein 2er oder 3er Teams lösen. Bitte geben Sie dann die Arbeitsteilung im Dokument mit an. Die 15minütige Einzelprüfung wird auf die Projektaufgabe eingehen.

1.3 Planungstool Aufbau

1.3.1 Das Interface

Sobald das Programm startet, öffnet sich ein Fenster mit einem Menü für die Auswahl der verschiedenen Probleme. Wenn ein Knopf gedrückt wird, öffnet sich ein neues Fenster mit dem entsprechenden Problem.



1.3.2 Die Aufgaben

Jedes Problem besitzt eine eigene Klasse `ProblemX.java`, die sich im Ordner `code` befinden.

Der Ordner `code/utils` enthält die Klassen: - `AdjazenzMatrix.java`: Speichert eine `int[][]` Matrix, `char[]` Buchstaben-Array und ob es sich um einen gerichteten oder ungerichteten Graphen handelt. - `FileHandler.java`: Stellt Methoden zum Einlesen und Schreiben von Dateien bereit. - `Vertex.java`: Speichert einen Buchstaben, einen Vertex-Vorgänger, einen 'key' und eine Option ob der Vertex bereits besucht wurde. - `Edge.java`: Speichert zwei Buchstaben für den Start- und Endknoten der Kante und ein Gewicht. - `JGraphPanel.java`: Eine Klasse, die ein `JPanel` erweitert und mit Hilfe der `mxgraph` und `jgrapht` Bibliotheken einen Graphen zeichnet. - `BasicWindow.java`: Eine Klasse, die ein `JFrame` erweitert und als Grundlage für die Fenster der einzelnen Probleme dient.

Die Eingabe

Die Eingabedateien befinden sich in dem Ordner `data` und folgen dem Namensschema `problemX.txt`. Die Dateien werden mit der Klasse `FileHandler.java` eingelesen und in einer Instanz der Klasse `AdjazenzMatrix.java` gespeichert.

	Ungerichteter Graph	Gerichteter Graph
1	A B C ...	A B C ...
2	A 0	A 0 0 1
3	B 1 0	B 0 0 1
4	C 2 3 0	C 1 0 0
5
6		

Die Ausgabe

Die Ausgabedateien befinden sich in dem Ordner `output` und werden automatisch generiert oder überschrieben sobald ein Punkt aus dem Menü ausgewählt wird.

Beim Betätigen eines Menübuttons wird die Ein- und Ausgabe graphisch dargestellt und zusätzlich in der Konsole ausgegeben.

In der Ausgabedatei werden die Graphen in einer Adjazenzmatrix gespeichert, die dem Schema der Eingabedatei entspricht.

1.4 Das Team

- Felix Möhler - [GitHub](#)
- Julian Thiele - [GitHub](#)

1.5 Auftraggeber

Prof. Barbara Sprick - Professorin für Praktische Informatik bei TH Aschaffenburg

2. Problem 1 - "Straßen müssen her!"

Lange Zeit gab es in der Graphschaft Schilda einen Reformstau, kein Geld floss mehr in die Infrastruktur. Wie es kommen musste, wurde der Zustand der Stadt zusehends schlechter, bis die Bürger der Graphschaft den Aufbau Ihrer Stadt nun endlich selbst in die Hand nahmen. Zunächst einmal sollen neue Straßen gebaut werden. Zur Zeit gibt es nur einige schlammige Wege zwischen den Häusern. Diese sollen nun gepflastert werden, so dass von jedem Haus jedes andere Haus erreichbar ist.

Da die Bürger der Stadt arm sind, soll der Straßenbau insgesamt möglichst wenig kosten. Die Bürger haben bereits einen Plan mit möglichen Wegen erstellt. Ihre Aufgabe ist nun, das kostengünstigste Wegenetz zu berechnen, so dass alle Häuser miteinander verbunden sind (nehmen Sie dabei pro Pflasterstein Kosten von 1 an):

2.1 Modellierung des Problems

Das Problem lässt sich als Graphenmodell mit ungerichteten Kanten darstellen. Jedes Haus ist ein Knoten, die Straßen sind die Kanten. Die Kosten der Kanten sind die Kosten für die Pflastersteine.

Es wird eine Konfiguration an Kanten gesucht, die eine minimale Anzahl an Pflastersteinen benötigt.

Um den Graph zu modellieren werden die Java-Bibliotheken `JGraphT` und `JGraphX` verwendet. Mit `JGraphT` wird der Graph als Datenstruktur modelliert. Mit `JGraphX` wird der Graph als Grafik dargestellt und auf dem Bildschirm dargestellt.

2.2 Die Eingabe

Die Eingabe besteht aus einem Graphen, der aus Kanten und Knoten besteht. Diese werden aus einer `.json` Datei gelesen und in eine Instanz der Klasse `Graph.java` geladen. Diese Instanz dient als Basis für die Berechnung des günstigsten Weges.

```

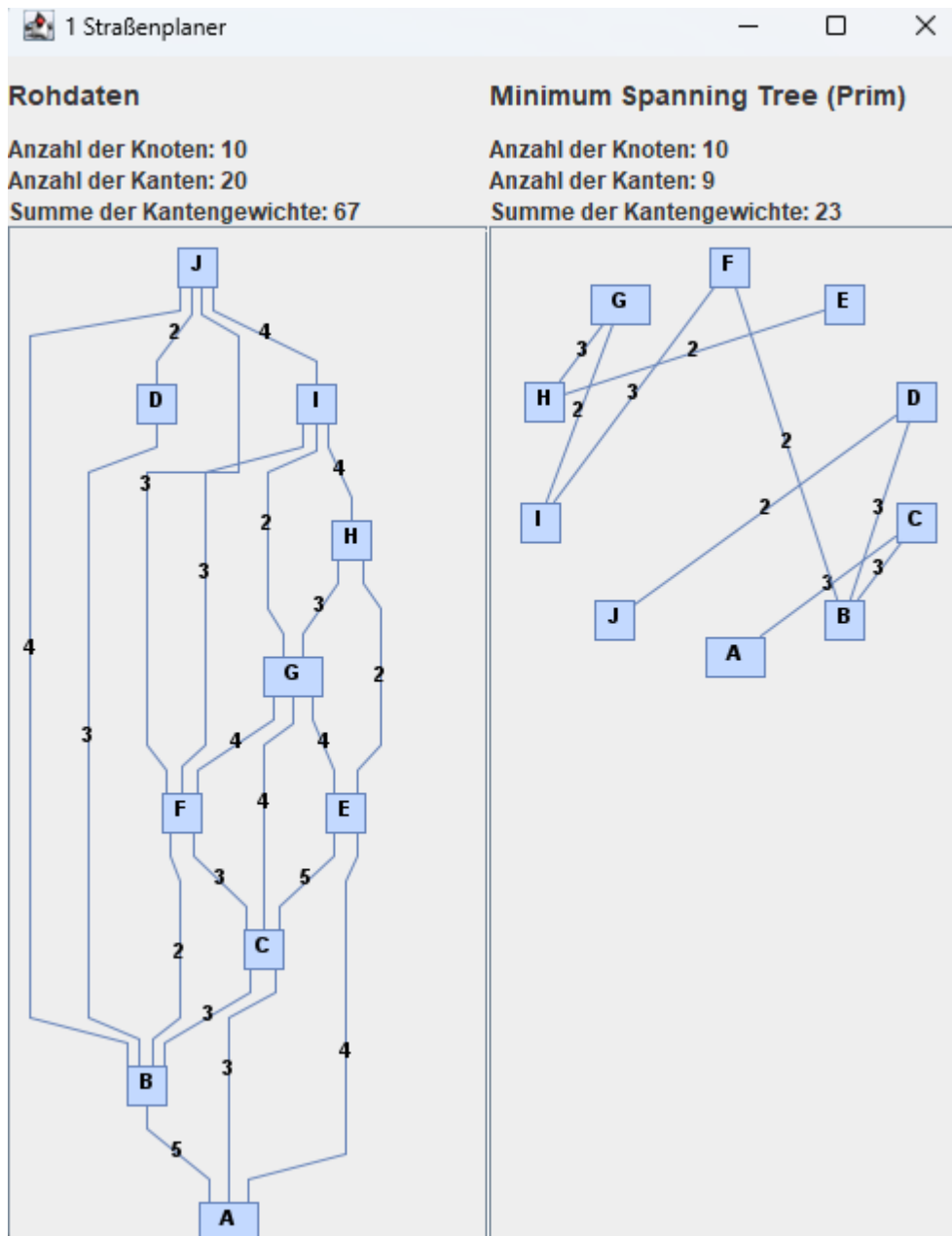
1  {
2    "directed_edges": false,
3    "vertices": [
4      { "label": "House 0" },
5      { "label": "House 1" },
6      ...
7    ],
8    "edges": [
9      { "source": "House 0", "target": "House 1", "weight": 5 },
10     { "source": "House 0", "target": "House 2", "weight": 3 },
11     { "source": "House 0", "target": "House 4", "weight": 4 },
12     ...
13   ]
14 }
```

2.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt. Das Fenster besteht aus zwei Hälften. Auf der linken Seite wird der Eingabegraph dargestellt. Auf der rechten Seite wird der berechnete Graph dargestellt.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- TODO



2.4 Geeignete Algorithmen

TODO Beschreibung MST mit Prim kruskal

2.5 Die Laufzeit des Algorithmus

TODO Laufzeitberechnung $O(|E| + |V| \log |V|)$ TODO (Hier bitte auch eine Begründung einfügen, ein ausführlicher Beweis ist nicht notwendig.)

2.6 Die Implementierung des Algorithmus

Zur Lösung des Problems wurde der Algorithmus von Prim implementiert. Als Datenstruktur wurde eine Prioritätswarteschlange verwendet, die Instanzen der Klasse `Vertex` beinhaltet:

```
1 PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>(  
2     Comparator.comparingInt(Vertex::getValue));
```

Für den Umgang mit Knoten und Kanten wurden drei Klassen implementiert:

- `Vertex.java`: Beinhaltet die Eigenschaft `int value`, welche den Key für den Algorithmus von Prim darstellt und das Objekt `Vertex predecessor`, der vom Algorithmus gesetzt wird.
- `Edge.java`: Beinhaltet die Eigenschaften `String source`, `String target` und `double weight`.
- `Graph.java`: Behinhaltet die Listen `ArrayList<Edge>` und `ArrayList<Vertex>`

Aufgrund der Struktur der `Vertex` und `Edge` Klassen werden die zusätzlichen Funktionen `getNeighbors()` und `getEdgeBetweenTwoVertices()` benötigt. Diese Funktionen benötigen zusätzliche Laufzeit und werden in der Klasse `Graph` implementiert.

```
1 // Initialisiere alle Knoten mit ∞, setze den Vorgänger auf null  
2 for (Vertex v : vertices) {  
3     v.setValue(Integer.MAX_VALUE);  
4     v.setPredecessor(null);  
5 }  
6  
7 // Starte mit beliebigem Startknoten  
8 // Startknoten bekommt den Wert 0  
9 Vertex start = vertices.get(0);  
10 start.setValue(0);  
11  
12 // Speichere alle Knoten in einer geeigneten Datenstruktur Q  
13 // -> Prioritätswarteschlange  
14 PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>(  
15     Comparator.comparingInt(Vertex::getValue));  
16 queue.addAll(vertices);  
17  
18 // Solange es noch Knoten in Q gibt...  
19 while (!queue.isEmpty()) {  
20     // Wähle den Knoten aus Q mit dem kleinsten Schlüssel (v)  
21     Vertex vertex = queue.poll();  
22  
23     // Für jeden Nachbarknoten n von v...  
24     for (Vertex n : Graph.getNeighbors(vertex, vertices, edges)) {  
25  
26         // Finde Kante zwischen v und n  
27         Edge edge = Graph.getEdgeBetweenTwoVertices(vertex, n, edges);  
28  
29         // Wenn der Wert der Kante kleiner ist als der Wert des Knotens und der Knoten  
30         // noch in Q enthalten ist  
31         if (edge.getWeight() < n.getValue() || !queue.contains(n))  
32             continue;  
33  
34         // Speichere v als Vorgänger von n und passe wert von n an  
35         n.setValue((int) edge.getWeight());  
36         n.setPredecessor(vertex);  
37  
38         // Aktualisiere die Prioritätswarteschlange  
39         queue.remove(n);  
40         queue.add(n);  
41     }  
42 }
```


3. Problem 2 - "Wasserversorgung"

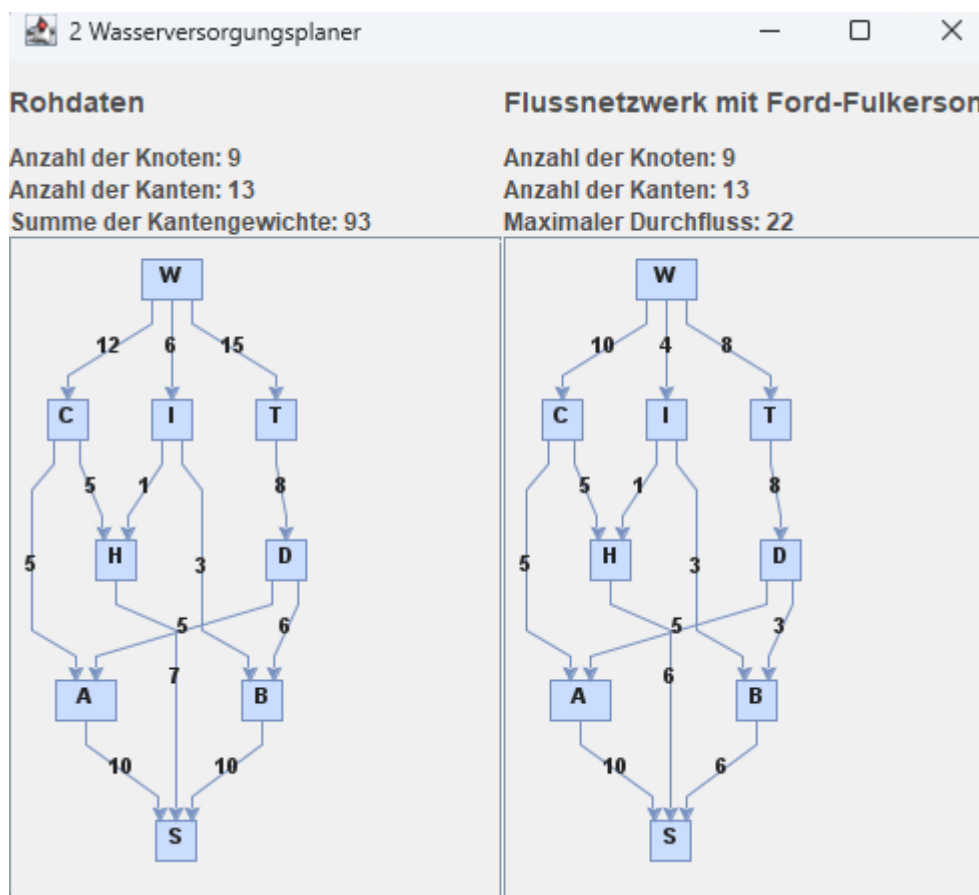
Der Straßenbau in der Graphschaft Schilda war erfolgreich, die Stadt blüht und gedeiht wieder! Selbst ein neuer Supermarkt soll eröffnet werden. Nun muss dieser aber mit Wasser versorgt werden, und da Sie bereits das Straßenbauprojekt so erfolgreich durchgeführt haben, werden Sie nun auch damit beauftragt, den neuen Supermarkt an die Wasserversorgung anzuschließen.

Da die Stadt nach wie vor kein Geld verschwenden möchte, müssen Sie zunächst feststellen, ob das bestehende Leitungsnetz noch ausreichend Kapazität für den zusätzlichen Wasserverbrauch hat, oder ob neue Leitungen benötigt werden. Da die Graphschaft Schilda noch keine Pumpen kennt, kann das Wasser nur bergab fließen. Als Vorarbeit haben Ihnen die Bürger die bestehende Wasserversorgung und die Lage des neuen Supermarktes aufgezeichnet:

3.1 Modellierung des Problems

3.2 Die Eingabe

3.3 Die Ausgabe



3.4 Der Algorithmus

3.5 Die Laufzeit des Algorithmus

3.6 Die Implementierung des Algorithmus

4. Problem 3 - "Stromversorgung"

Die Stadt floriert, alles wird moderner und so muss auch die Stromversorgung erneuert werden. Die Stadt hat bereits eruiert, wo Strommasten aufgestellt werden können. Sie haben auch festgestellt, dass es keine Barrieren in der Stadt gibt, d.h., prinzipiell könnten alle Strommasten miteinander verbunden werden. Aber natürlich wollen wir lange Leitungen möglichst vermeiden.

Deswegen schränken wir von vornherein ein, dass jeder Strommast nur mit maximal 5 nächsten Nachbarn verbunden werden darf. Es stellt sich heraus, dass dies immer noch zu teuer ist. Deswegen soll dieses Netz noch einmal so reduziert werden, dass zwar alle Strommasten miteinander verbunden sind, aber Kosten insgesamt minimal sind. Wir nehmen dabei an, dass die Kosten ausschließlich von der Leitungslänge abhängen.

4.1 Modellierung des Problems

Das Problem lässt sich als Graphenmodell mit ungerichteten Kanten darstellen. Jeder Strommast ist ein Knoten, die Verbindungen sind die Kanten. Die Kosten der Kanten sind die Länge der Stromleitungen.

Es wird eine Konfiguration an Leitungen zwischen den Strommasten gesucht, die eine minimale Gesamtlänge besitzt, und jeder Strommast mit maximal mit 5 weiteren Masten verbunden sein darf.

Um den Graph zu modellieren werden die Java-Bibliotheken `JGraphT` und `JGraphX` verwendet. Mit `JGraphT` wird der Graph als Datenstruktur modelliert. Mit `JGraphX` wird der Graph als Grafik dargestellt und auf dem Bildschirm dargestellt.

4.2 Die Eingabe

Die Eingabe besteht aus Knoten, die aus einer `.json` Datei ausgelesen werden.

```
1  {
2    "directed_edges": false,
3    "vertices": [
4      { "label": "Pole 0" },
5      { "label": "Pole 1" },
6      { "label": "Pole 2" },
7      ...
8    ]
9  }
```

Mit der Funktion `generate_all_edges()` werden alle möglichen Kanten mit zufälligen Gewichten generiert. Diese werden dann dem Input hinzugefügt.

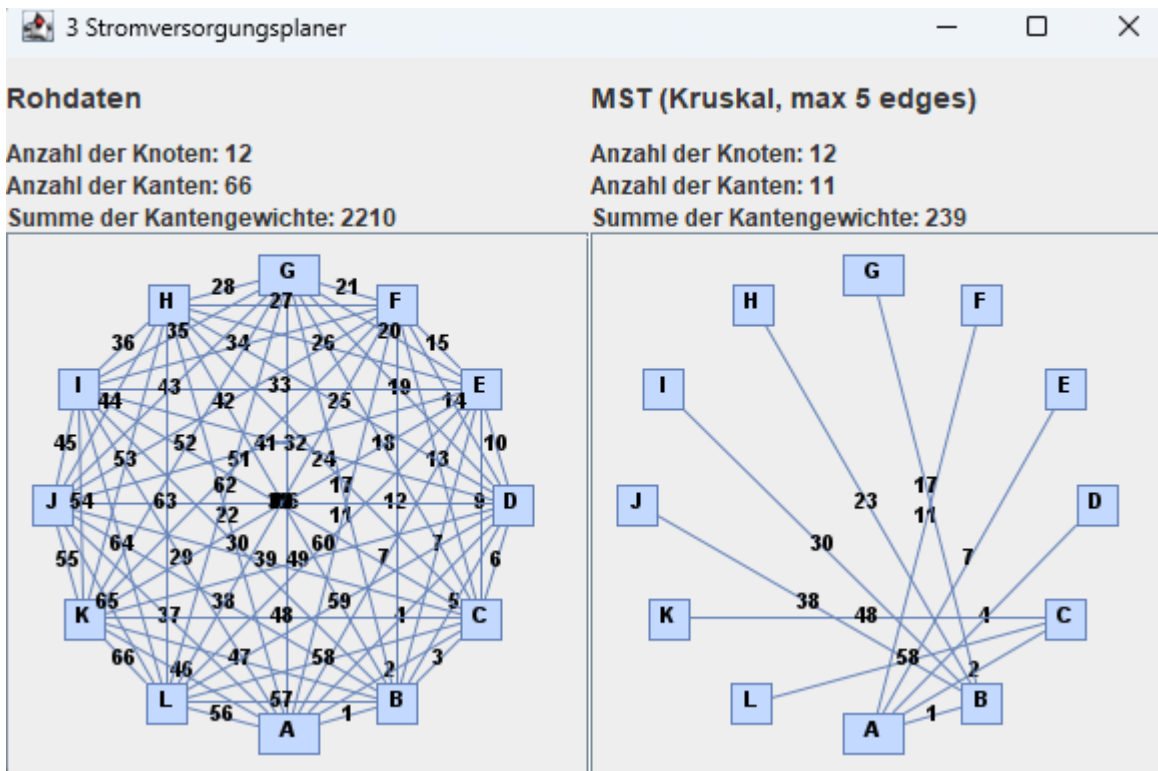
```
1  private ArrayList<Edge> generate_all_edges(ArrayList<Vertex> vertices) {
2    ArrayList<Edge> output = new ArrayList<Edge>();
3    for (int i = 0; i < vertices.size(); i++) {
4      for (int j = i + 1; j < vertices.size(); j++) {
5        // generate random weight and add new edge to output
6        output.add(new Edge(vertices.get(i).getLabel(),
7          vertices.get(j).getLabel(), Math.round(Math.random() * 100.0)));
8      }
9    }
10   return output;
11 }
```

4.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt. Das Fenster besteht aus zwei Hälften. Auf der linken Seite wird der Eingabegraph dargestellt. Auf der rechten Seite wird der berechnete Graph dargestellt.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- TODO



4.4 Der Alrogorithmus

TODO

```

1 // Überspringe die Kante e, wenn sie von einem Knoten ausgeht, der bereits mehr als 5 Kanten hat
2 ArrayList<Edge> source_edges = Graph.getAdjacentEdges(e.getSource(), output_edges);
3 ArrayList<Edge> target_edges = Graph.getAdjacentEdges(e.getTarget(), output_edges);
4
5 if (source_edges.size() >= max_edges || target_edges.size() >= max_edges)
6     continue;

```

4.5 Die Laufzeit des Algorithmus

TODO

4.6 Die Implementierung des Algorithmus

```

1 // Lese die Knoten und Kanten aus den Rohdaten
2 ArrayList<Vertex> vertices = input.getVertices();
3 ArrayList<Edge> edges = input.getEdges();
4
5 // Sortiere die Kanten nach Gewicht
6 edges.sort(Comparator.comparingDouble(Edge::getWeight));
7
8 // erstelle einen wald 'forest' (eine menge von bäumen), wo jeder knoten ein
9 // eigener baum ist
10 ArrayList<ArrayList<Vertex>> forest = new ArrayList<ArrayList<Vertex>>();
11 for (Vertex v : vertices) {
12     ArrayList<Vertex> tree = new ArrayList<Vertex>();
13     tree.add(v);
14     forest.add(tree);
15 }
16
17 // erstelle eine liste mit den kanten des minimum spanning trees
18 ArrayList<Edge> forest_edges = new ArrayList<Edge>(edges);
19
20 // erstelle eine liste für die Ausgabe
21 ArrayList<Edge> output_edges = new ArrayList<Edge>();
22
23 // solange der wald nicht leer ist und der baum noch nicht alle knoten enthält
24 while (forest_edges.size() > 0) {
25     // entferne eine kante (u, v) aus forest
26     Edge e = forest_edges.remove(0);
27
28     // finde die bäume, die mit der Kante e verbunden sind
29     ArrayList<Vertex> tree_u = null;
30     ArrayList<Vertex> tree_v = null;
31     for (ArrayList<Vertex> t : forest) {
32         if (t.contains(Graph.getSourceVertexFromEdge(e, vertices)))
33             tree_u = t;
34         if (t.contains(Graph.getTargetVertexFromEdge(e, vertices)))
35             tree_v = t;
36     }
37
38     // Prüfe ob die kante e von einem vertex ausgeht, der bereits mehr als 5 kanten
39     // hat
40     ArrayList<Edge> source_edges = Graph.getAdjacentEdges(e.getSource(), output_edges);
41     ArrayList<Edge> target_edges = Graph.getAdjacentEdges(e.getTarget(), output_edges);
42
43     if (source_edges.size() >= max_edges || target_edges.size() >= max_edges)
44         continue;
45
46     // wenn u und v in gleichen Bäumen sind -> skip
47     if (tree_u == tree_v)
48         continue;
49
50     // füge kante von u und v zur Ausgabe hinzu
51     output_edges.add(e);
52
53     // füge baum von v zu baum von u hinzu (merge)
54     for (Vertex v : tree_v)
55         tree_u.add(v);
56
57     forest.remove(tree_v);
58 }

```

5. Problem 4 - "Historische Funde"

Beim Ausheben der Wege während des Straßenbaus wurde ein antiker Feuerwerksplan gefunden. Die Lage der pyrotechnischen Effekte und die Zündschnüre sind noch sehr gut zu erkennen.

Wie aber ist die Choreographie des Feuerwerks? In welcher Reihenfolge zünden die Bomben? Können Sie den Bürgern der Graphschaft Schilda helfen? (Unter der Annahme, dass die Zündschnur immer mit gleichbleibender Geschwindigkeit abbrennt...)

5.1 Modellierung des Problems

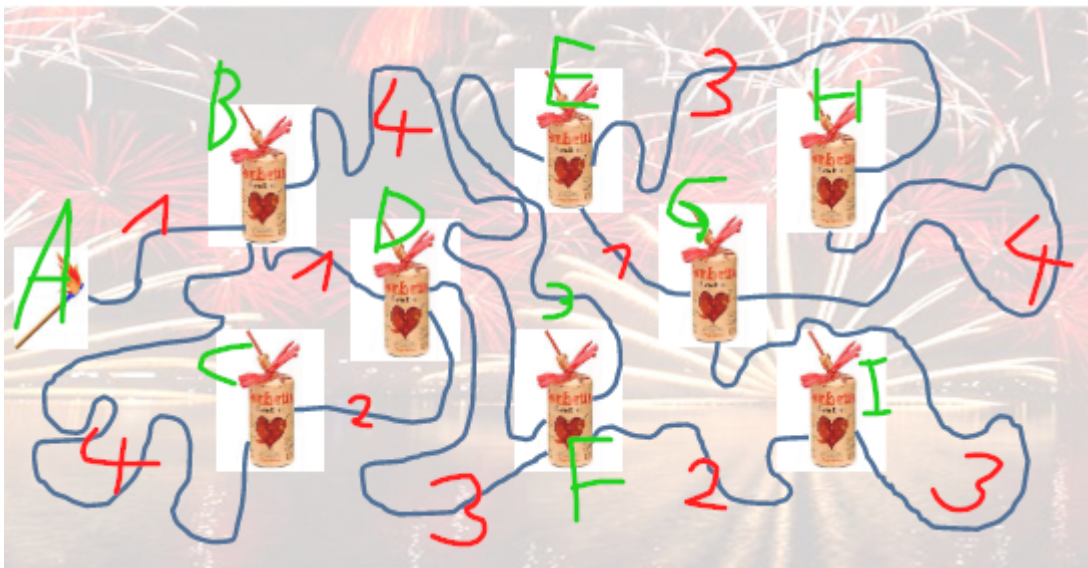
Das Problem lässt sich als Graphenmodell mit ungerichteten Kanten darstellen. Das Steichholz und die Feuerwerkskörper sind die Knoten, die Zündschnüre sind die Kanten. Die Kosten der Kanten sind die Länge der Zündschnüre.

Es wird die korrekte Reihenfolge gesucht, in der die Feuerwerkskörper gezündet werden wenn das Steichholz den ersten Feuerwerkskörper zündet.

Um den Graph zu modellieren werden die Java-Bibliotheken `JGraphT` und `JGraphX` verwendet. Mit `JGraphT` wird der Graph als Datenstruktur modelliert. Mit `JGraphX` wird der Graph als Grafik dargestellt und auf dem Bildschirm dargestellt.

5.2 Die Eingabe

Um das Bild der Aufgabenstellung in konkrete Daten zu übersetzen, wurden hier Schätzungen der Länge der Zündschnüre vorgenommen und in einer Grafik dargestellt.



Die Eingabe besteht aus Knoten und Kanten, die aus einer `.json` Datei ausgelesen werden.

```

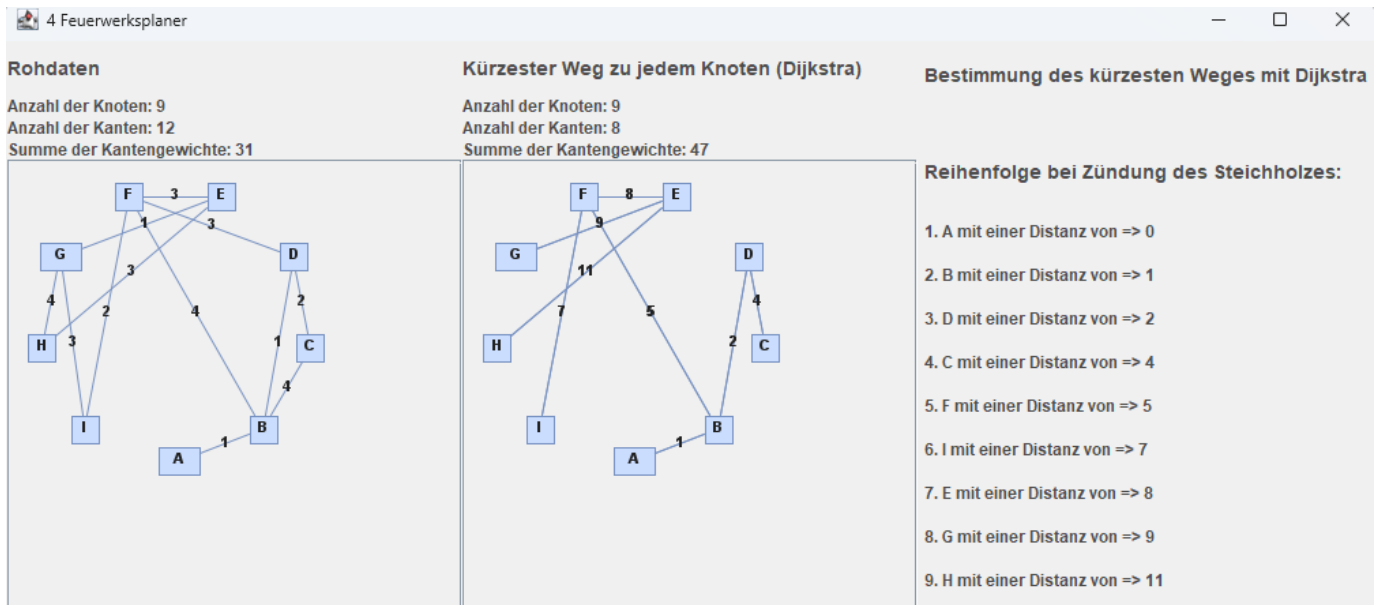
1  {
2    "directed_edges": false,
3    "vertices": [
4      { "label": "Match" },
5      { "label": "Firecracker 1" },
6      { "label": "Firecracker 2" },
7      ...
8    ],
9    "edges": [
10     { "source": "Match", "target": "Firecracker 1", "weight": 1 },
11     { "source": "Firecracker 1", "target": "Firecracker 2", "weight": 4 },
12     { "source": "Firecracker 1", "target": "Firecracker 3", "weight": 1 },
13     ...
14   ]
15 }
```

5.3 Die Ausgabe

Die Ausgabe wird als Liste der Knoten dargestellt, in der die Knoten in der Reihenfolge aufgelistet sind, in der sie gezündet werden.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- TODO



5.4 Der Algorithmus

TODO

5.5 Die Laufzeit des Algorithmus

TODO

5.6 Die Implementierung des Algorithmus

TODO

```

1 // Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten
2 // mit ∞.
3 for (Vertex vertex : vertices) {
4     vertex.setValue(Integer.MAX_VALUE);
5     vertex.setPredecessor(null);
6 }
7 vertices.get(0).setValue(0);
8
9 // Speichere alle Knoten in einer Prioritätswarteschlange queue
10 PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>(
11     Comparator.comparingInt(Vertex::getValue));
12 queue.addAll(vertices);
13
14 // Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit
15 // minimaler Distanz aus und
16 while (!queue.isEmpty()) {
17     // Nimm den Knoten mit dem kleinsten Wert aus der Warteschlange
18     Vertex v = queue.poll();
19
20     // 1. speichere, dass dieser Knoten schon besucht wurde
21     v.setVisited(true);
22
23     // 2. berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen
24     // Kantengewichtes und der Distanz im aktuellen Knoten
25     for (Vertex n : Graph.getNeighbors(v, vertices, edges)) {
26
27         // 3. ist dieser Wert für einen Knoten kleiner als die
28         // dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten
29         // als Vorgänger. (Dieser Schritt wird auch als Update bezeichnet.)
30         double sum = v.getValue() + Graph.getWeightSum(v, n, edges);
31
32         if (sum <= n.getValue())
33             continue;
34
35         n.setValue((int) sum);
36         n.setPredecessor(v);
37         // Aktualisiere die Prioritätswarteschlange
38         queue.remove(n);
39         queue.add(n);
40     }
41 }
42
43 // Sortiere Knoten nach Distanz
44 vertices.sort(Comparator.comparingInt(Vertex::getValue));

```


6. Problem 5 - "Die Festhochzeit - das Verteilen der Einladungen"

Zum großen Hochzeitfest in Schilda sollen natürlich alle Ortsbewohner eingeladen werden.

Da in der letzten Zeit in Schilda viele neue Straßen gebaut worden waren, muss der Briefträger eine neue Route finden. Glücklicherweise haben alle Häuser ihre Briefkästen an die neuen Straßen gestellt, so dass der Briefträger nur noch den kürzesten Weg finden muss, der ausgehend von der Post durch alle Straßen und dann wieder zurück zur Post führt.

Hier sind Sie wieder gefragt: Unterstützen Sie den Briefträger und entwerfen Sie die neue Route für den Briefträger. Den Stadtplan kennen Sie ja bereits, schließlich haben Sie die Straßen gebaut.

6.1 Modellierung des Problems

6.2 Die Eingabe

6.3 Die Ausgabe

6.4 Der Algorithmus

6.5 Die Laufzeit des Algorithmus

6.6 Die Implementierung des Algorithmus

7. Problem 6 - "Wohin nur mit den Gästen?"

Zum Einweihungsfest werden zahlreiche auswärtige Gäste eingeladen. Reisen diese allerdings alle mit dem Auto an, dann ist ohne hervorragende Verkehrsplanung ein Stau in der Innenstadt vorprogrammiert. Parken können die Autos auf dem Parkplatz des neuen Supermarktes.

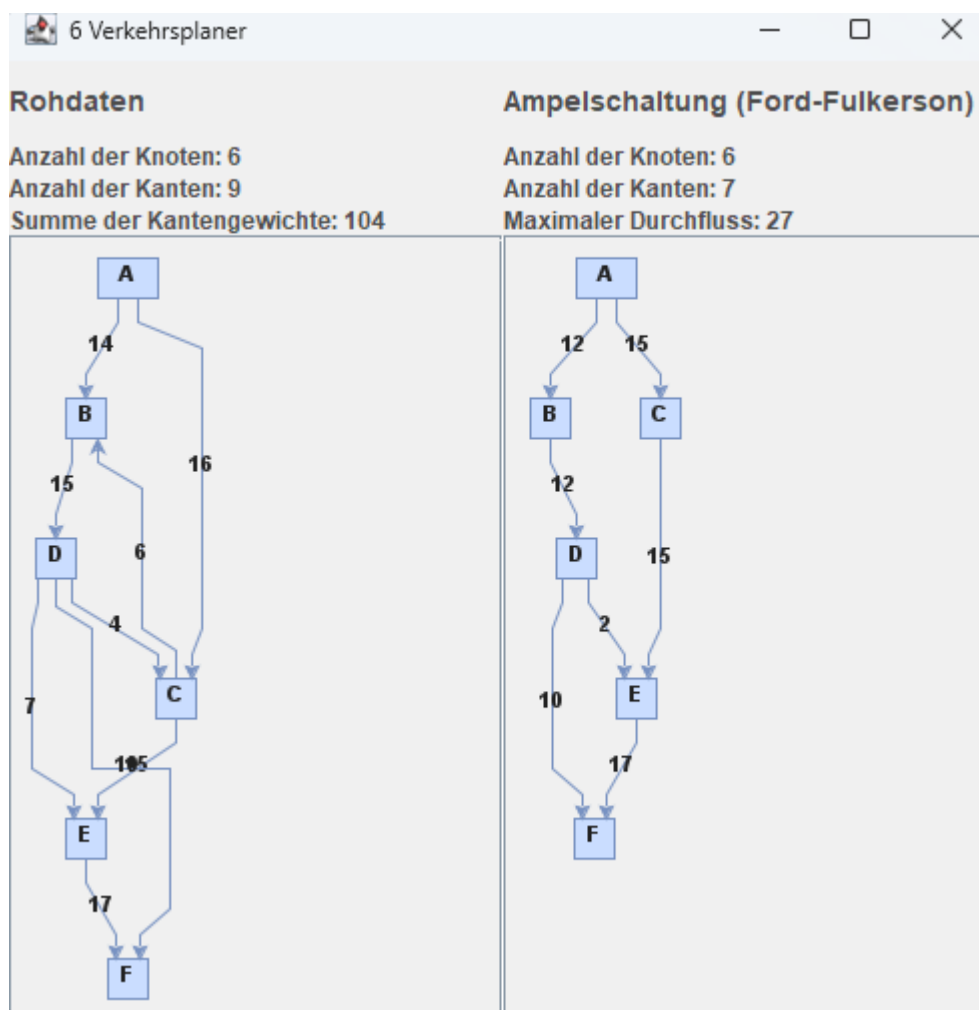
Doch wie soll der Verkehr durch die Stadt geleitet werden, dass möglichst viele Fahrzeuge von der Autobahn zum Parkplatz gelangen können, ohne dass sich lange Schlangen vor den Ampeln bilden? Die Kapazität der einzelnen Straßen haben Ihnen die Bürger der Stadt bereits aufgezeichnet.

Sie sollen nun planen, wie viele Wagen über die einzelnen Wege geleitet werden sollen.

7.1 Modellierung des Problems

7.2 Die Eingabe

7.3 Die Ausgabe



7.4 Der Algorithmus

7.5 Die Laufzeit des Algorithmus

7.6 Die Implementierung des Algorithmus

8. Problem 7 - "Es gibt viel zu tun! Wer macht's"

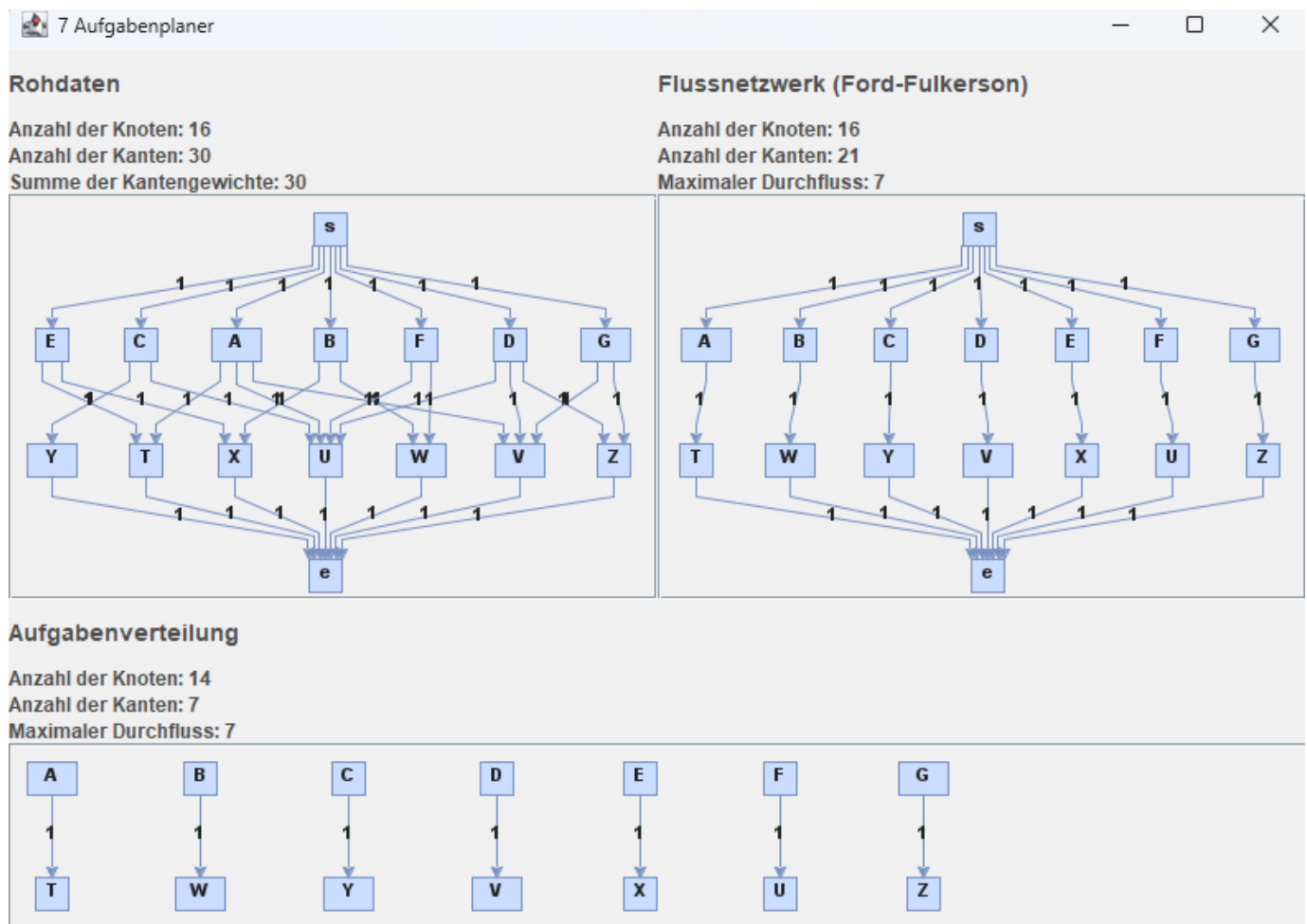
Während die Bürger der Stadt Schilda ganz begeistert von Ihnen sind, bekommen Sie immer mehr Aufträge, die Sie gar nicht mehr alleine bewältigen können. Sie stellen also neues Personal für die Projektleitung ein. Jeder Mitarbeiter hat unterschiedliche Kompetenzen und Sie wollen die Mitarbeiter so auf die Projekte verteilen, dass jedes Projekt von genau einem Mitarbeiter oder einer Mitarbeiterin mit den notwendigen Kompetenzen geleitet wird.

Wie ordnen Sie die Mitarbeiter den Projekten zu? (Genau ein Mitarbeiter pro Projekt) (Auch diesen Algorithmus integrieren Sie in Ihr Tool – schließlich möchte auch die Graphschaft ihre Kräfte gut einsetzen!)

8.1 Modellierung des Problems

8.2 Die Eingabe

8.3 Die Ausgabe



8.4 Der Algorithmus

8.5 Die Laufzeit des Algorithmus

8.6 Die Implementierung des Algorithmus
