# Die Graphschaft Schilda

**Felix Möhler und Julian Thiele**

*Felix Möhler und Julian Thiele*

# Inhaltsverzeichnis

# 1. Die Graphschaft Schilda

## 1.1 Abstract

Dieses Dokument 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 Das Team

- Felix Möhler - GitHub
- Julian Thiele - GitHub

## 1.4 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 Insanz der Klasse `GraphData.java` geladen. Diese Insanz dient als Basis für die Berechnung des günstigsten Weges.

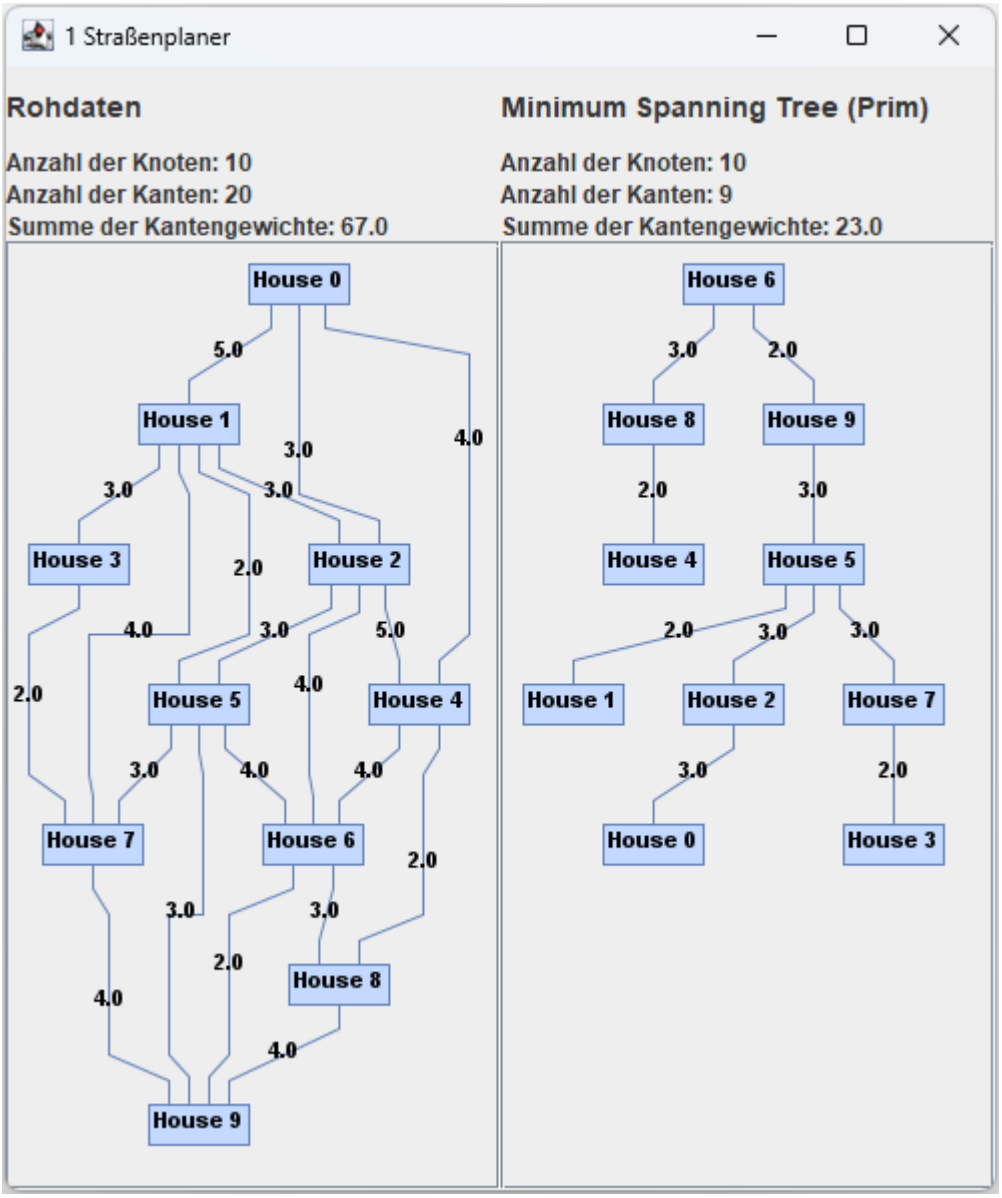
```json
{
  "directed_edges": false,
  "vertices": [
    { "label": "House 0" },
    { "label": "House 1" },
    ...
  ],
  "edges": [
    { "source": "House 0", "target": "House 1", "weight": 5 },
    { "source": "House 0", "target": "House 2", "weight": 3 },
    { "source": "House 0", "target": "House 4", "weight": 4 },
    ...
  ]
}
```

## 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 `GraphVertex` beinhaltet:

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

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

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

Aufgrund der Struktur der `GraphVertex` und `GraphEdge` Klassen werden die zusätzlichen Funktionen `getNeighbors()` und `getEdgesBetweenTwoVertices()` benötigt. Diese Funktionen benötigen zusätzlicehe Laufzeit und werden in der Klasse `GraphData` implementiert.

```
// Initialisiere alle Knoten mit ∞, setze den Vorgänger auf null
for (GraphVertex v : vertices) {
  v.setValue(Integer.MAX_VALUE);
  v.setPredecessor(null);
}

// Starte mit beliebigem Startknoten
// Startknoten bekommt den Wert 0
GraphVertex start = vertices.get(6);
start.setValue(0);

// Speichere alle Knoten in einer geeigneten Datenstruktur Q
// -> Prioritätswarteschlange
PriorityQueue<GraphVertex> queue = new PriorityQueue<GraphVertex>(
    Comparator.comparingInt(GraphVertex::getValue));
queue.addAll(vertices);

// Solange es noch Knoten in Q gibt...
while (!queue.isEmpty()) {
  // Wähle den Knoten aus Q mit dem kleinsten Schlüssel (v)
  GraphVertex vertex = queue.poll();

  // Speichere alle Nachbarn von v in neighbours
  ArrayList<GraphVertex> neighbors = GraphData.getNeighbors(vertex, vertices, edges);

  for (GraphVertex n : neighbors) {
    // Finde Kante zwischen v und n
    for (GraphEdge edge : GraphData.getEdgesBetweenTwoVertices(vertex, n, edges)) {
      // Wenn der Wert der Kante kleiner ist als der Wert des Knotens und der Knoten
      // noch in Q enthalten ist
      if (edge.getWeight() >= n.getValue() || !queue.contains(n))
        continue;

      // Speichere v als vorgänger von n und passe wert von n an
      n.setValue((int) edge.getWeight());
      n.setPredecessor(vertex);
      // Aktualisiere die Prioritätswarteschlange
      queue.remove(n);
      queue.add(n);
    }
  }
}
```

# 3. Problem 2 - "Wasserversorgung"

## 3.1 Modellierung des Problems

## 3.2 Die Eingabe

## 3.3 Die Ausgabe

## 3.4 Der Alrogithmus

## 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.

```
{
  "directed_edges": false,
  "vertices": [
    { "label": "Pole 0" },
    { "label": "Pole 1" },
    { "label": "Pole 2" },
    ...
  ]
}
```

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

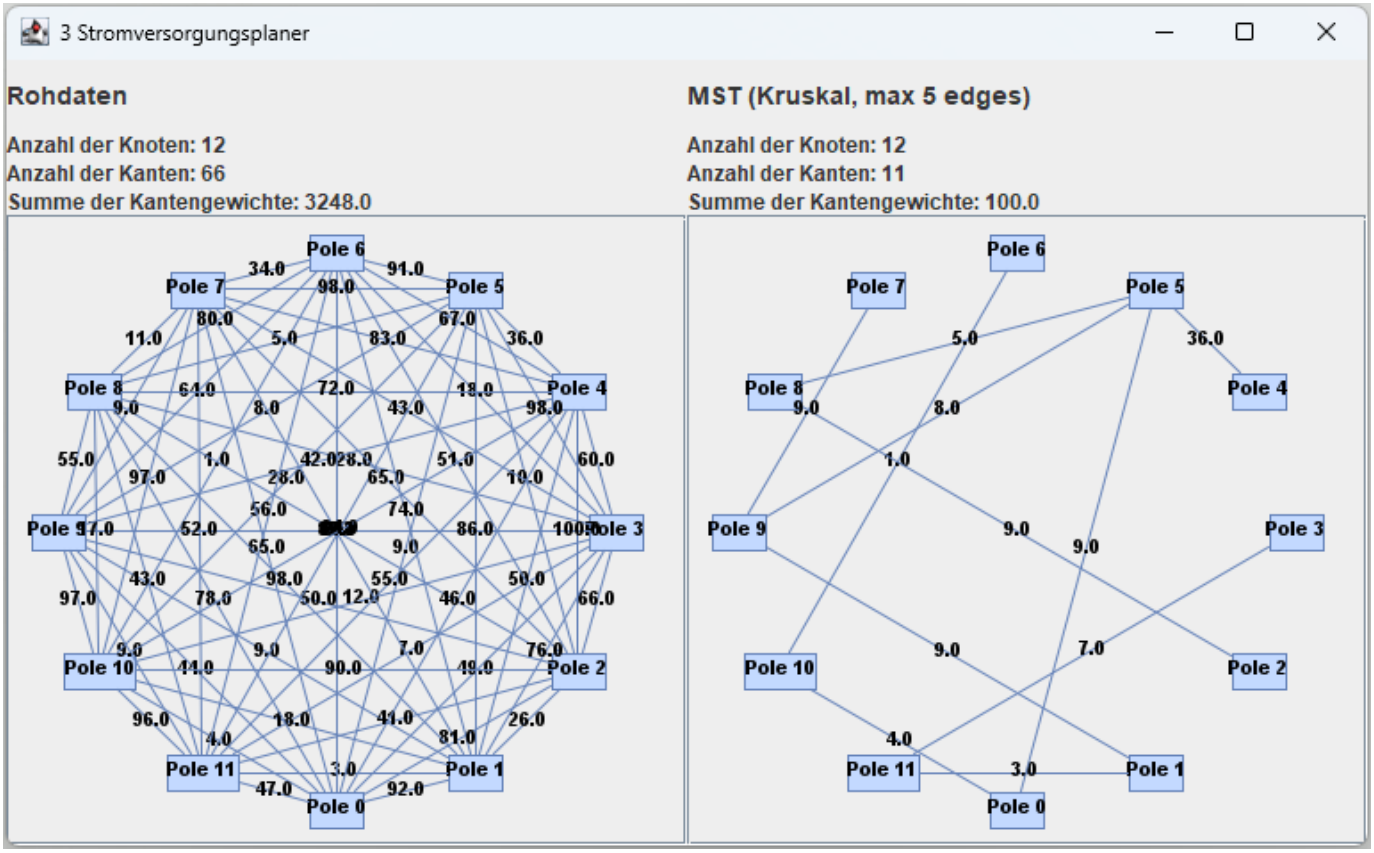
```
private ArrayList<GraphEdge> generate_all_edges(ArrayList<GraphVertex> vertices) {
  ArrayList<GraphEdge> output = new ArrayList<GraphEdge>();
  for (int i = 0; i < vertices.size(); i++) {
    for (int j = i + 1; j < vertices.size(); j++) {
      // generate random weight and add new edge to output
      output.add(new GraphEdge(vertices.get(i).getLabel(),
          vertices.get(j).getLabel(), Math.round(Math.random() * 100.0)));
    }
  }
  return output;
}
```

## 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 Alrogithmus

TODO

```
// Überspringe die Kante e, wenn sie von einem Knoten ausgeht, der bereits mehr als 5 Kanten hat
ArrayList<GraphEdge> source_edges = GraphData.getAdjacentEdges(e.getSource(), output_edges);
ArrayList<GraphEdge> target_edges = GraphData.getAdjacentEdges(e.getTarget(), output_edges);

if (source_edges.size() >= max_edges || target_edges.size() >= max_edges)
    continue;
```

## 4.5 Die Laufzeit des Algorithmus

TODO

## 4.6 Die Implementierung des Algorithmus

```
// Lese die Knoten und Kanten aus den Rohdaten
ArrayList<GraphVertex> vertices = input.getVertices();
ArrayList<GraphEdge> edges = input.getEdges();

// Sortiere die Kanten nach Gewicht
edges.sort(Comparator.comparingDouble(GraphEdge::getWeight));

// erstelle einen wald 'forest' (eine menge von bäumen), wo jeder knoten ein
// eigener baum ist
ArrayList<ArrayList<GraphVertex>> forest = new ArrayList<ArrayList<GraphVertex>>();
for (GraphVertex v : vertices) {
    ArrayList<GraphVertex> tree = new ArrayList<GraphVertex>();
    tree.add(v);
    forest.add(tree);
}

// erstelle eine liste mit den kanten des minimum spanning trees
ArrayList<GraphEdge> forest_edges = new ArrayList<GraphEdge>(edges);

// erstelle eine liste für die Ausgabe
ArrayList<GraphEdge> output_edges = new ArrayList<GraphEdge>();

// solange der wald nicht leer ist und der baum noch nicht alle knoten enthält
while (forest_edges.size() > 0) {
    // entferne eine kante (u, v) aus forest
    GraphEdge e = forest_edges.remove(0);

    // finde die bäume, die mit der Kante e verbunden sind
    ArrayList<GraphVertex> tree_u = null;
    ArrayList<GraphVertex> tree_v = null;
    for (ArrayList<GraphVertex> t : forest) {
      if (t.contains(GraphData.getSourceVertexFromEdge(e, vertices)))
          tree_u = t;
      if (t.contains(GraphData.getTargetVertexFromEdge(e, vertices)))
          tree_v = t;
    }

    // Prüfe ob die kante e von einem vertex ausgeht, der bereits mehr als 5 kanten
    // hat
    ArrayList<GraphEdge> source_edges = GraphData.getAdjacentEdges(e.getSource(), output_edges);
    ArrayList<GraphEdge> target_edges = GraphData.getAdjacentEdges(e.getTarget(), output_edges);

    if (source_edges.size() >= max_edges || target_edges.size() >= max_edges)
        continue;

    // wenn u und v in gleichen Bäumen sind -> skip
    if (tree_u == tree_v)
        continue;

    // füge kante von u und v zur Ausgabe hinzu
    output_edges.add(e);

    // füge baum von v zu baum von u hinzu (merge)
    for (GraphVertex v : tree_v)
        tree_u.add(v);

    forest.remove(tree_v);
}
```

# 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

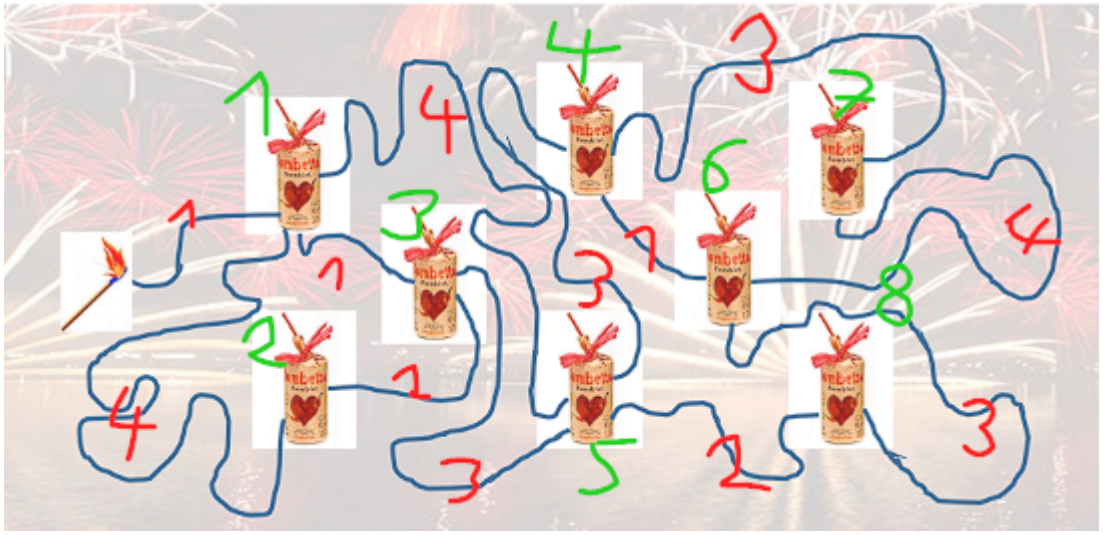
Das Problem lässt sich als Graphenmodell mit ungerichteten Kanten darstellen. Das Steichholz und die Feuerwerkskörper sind die Knoten, die Zündschnüre sind die Kanten. Die Kosten der Kanten sind die Länge der Zündschnüre.

Es wird die korrekte Reihenfolge gesucht, in der die Feuerwerkskörper gezündet werden wenn das Streichholz 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 Zünschnüre vorgenommen und in einer Grafik dargestellt.



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

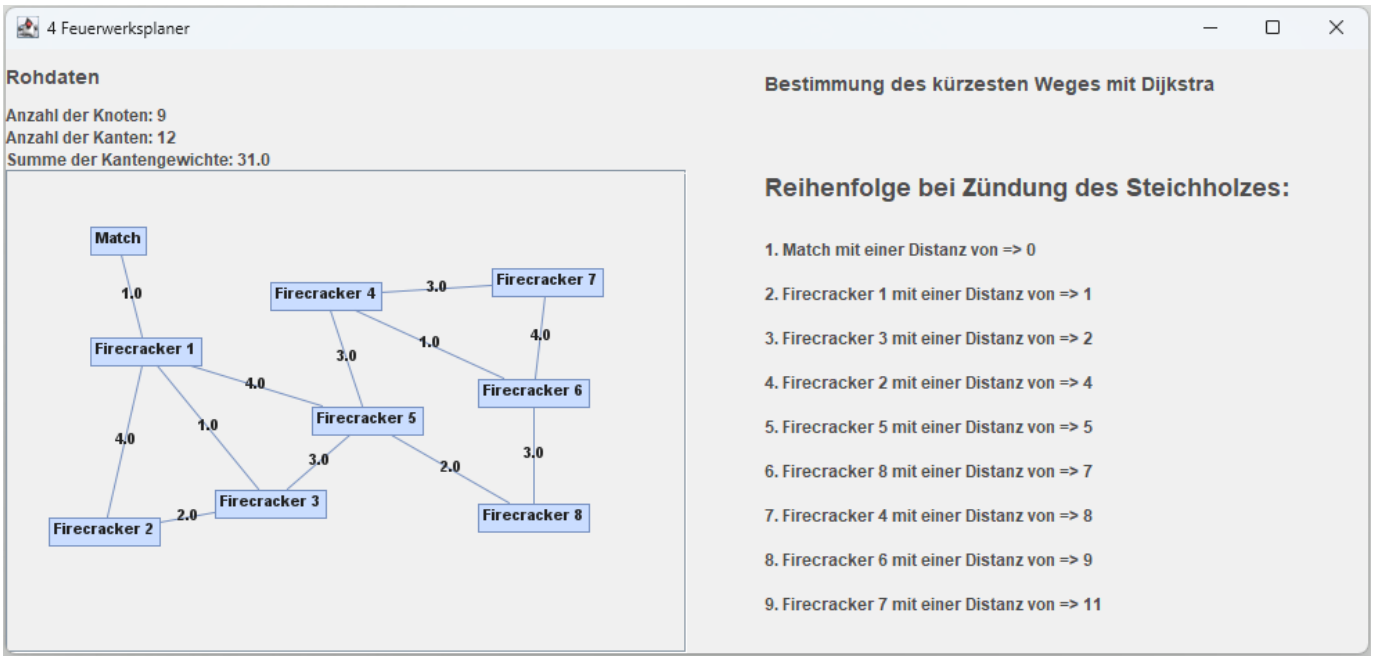
```
{
  "directed_edges": false,
  "vertices": [
    { "label": "Match" },
    { "label": "Firecracker 1" },
    { "label": "Firecracker 2" },
    ...
  ],
  "edges": [
    { "source": "Match", "target": "Firecracker 1", "weight": 1 },
    { "source": "Firecracker 1", "target": "Firecracker 2", "weight": 4 },
    { "source": "Firecracker 1", "target": "Firecracker 3", "weight": 1 },
    ...
  ]
}
```

## 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 Alrogithmus

TODO

## 5.5 Die Laufzeit des Algorithmus

TODO

## 5.6 Die Implementierung des Algorithmus

TODO

```
// Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten
// mit ∞.
for (GraphVertex vertex : vertices) {
  vertex.setValue(Integer.MAX_VALUE);
  vertex.setPredecessor(null);
}
vertices.get(0).setValue(0);

// Speichere alle Knoten in einer Prioritätswarteschlange queue
PriorityQueue<GraphVertex> queue = new PriorityQueue<GraphVertex>(
    Comparator.comparingInt(GraphVertex::getValue));
queue.addAll(vertices);

// Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit
// minimaler Distanz aus und
while (!queue.isEmpty()) {
  // Nehme den Knoten mit dem kleinsten Wert aus der Warteschlange
  GraphVertex v = queue.poll();

  // 1. speichere, dass dieser Knoten schon besucht wurde
  v.setVisited(true);

  // 2. berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen
  // Kantengewichtes und der Distanz im aktuellen Knoten
  for (GraphVertex n : GraphData.getNeighbors(v, vertices, edges)) {

    // 3. ist dieser Wert für einen Knoten kleiner als die
    // dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten
    // als Vorgänger. (Dieser Schritt wird auch als Update bezeichnet. )
    double sum = v.getValue() + GraphData.getWeightSum(v, n, edges);

    if (sum >= n.getValue())
      continue;

    n.setValue((int) sum);
    n.setPredecessor(v);
    // Aktualisiere die Prioritätswarteschlange
    queue.remove(n);
    queue.add(n);
  }
}

// Sortiere Knoten nach Distanz
vertices.sort(Comparator.comparingInt(GraphVertex::getValue));
```

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

## 6.1 Modellierung des Problems

## 6.2 Die Eingabe

## 6.3 Die Ausgabe

## 6.4 Der Alrogithmus

## 6.5 Die Laufzeit des Algorithmus

## 6.6 Die Implementierung des Algorithmus

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

## 7.1 Modellierung des Problems

## 7.2 Die Eingabe

## 7.3 Die Ausgabe

## 7.4 Der Alrogithmus

## 7.5 Die Laufzeit des Algorithmus

## 7.6 Die Implementierung des Algorithmus

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

## 8.1 Modellierung des Problems

## 8.2 Die Eingabe

## 8.3 Die Ausgabe

## 8.4 Der Alrogithmus

## 8.5 Die Laufzeit des Algorithmus

## 8.6 Die Implementierung des Algorithmus