Die Graphschaft Schilda

Felix Möhler und Julian Thiele

Felix Möhler und Julian Thiele

Inhaltsverzeichnis

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

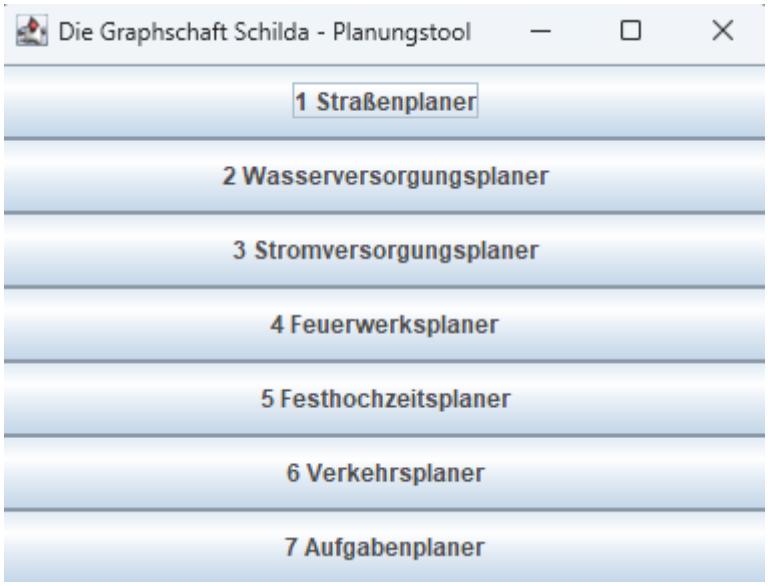
Entwickeln Sie ein Planungstool, dass der Graphschaft Schilda bei der Lösung ihrer Probleme hilft.

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. Die Laufzeit um eine Datei einzulesen ist `O(n)` mit n = Anzahl der Zeichen in der Datei. Die Laufzeit des Schreibens ist `O(V^2)` mit V = Anzahl der Knoten des Graphen.
- `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
  A B C ...               A B C ...
A 0                     A 0 0 1
B 1 0                   B 0 0 1
C 2 3 0                 C 1 0 0
...                     ...
```

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.

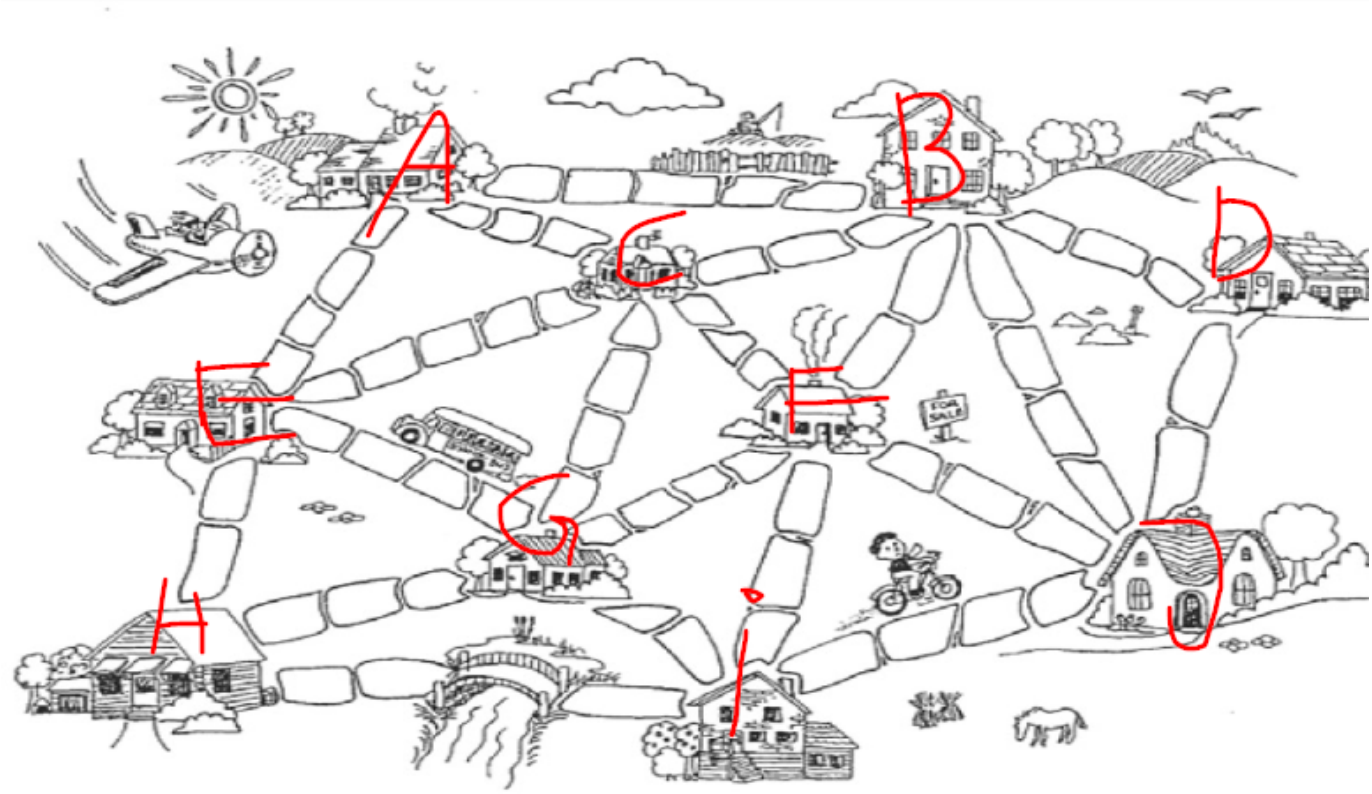
Es wird eine Konfiguration an Kanten gesucht, die eine minimale Anzahl an Pflastersteinen benötigt.

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 `.txt` Datei gelesen und in eine Instanz der Klasse `AdjazenzMatrix.java` geladen. Diese Instanz dient als Basis für die Berechnung des minimalen Spannbaums.

Das Bild aus der Aufgabenstellung wurde mit Buchstaben von A bis J beschriftet und daraus wurde die Datei `problem1.txt` erstellt.



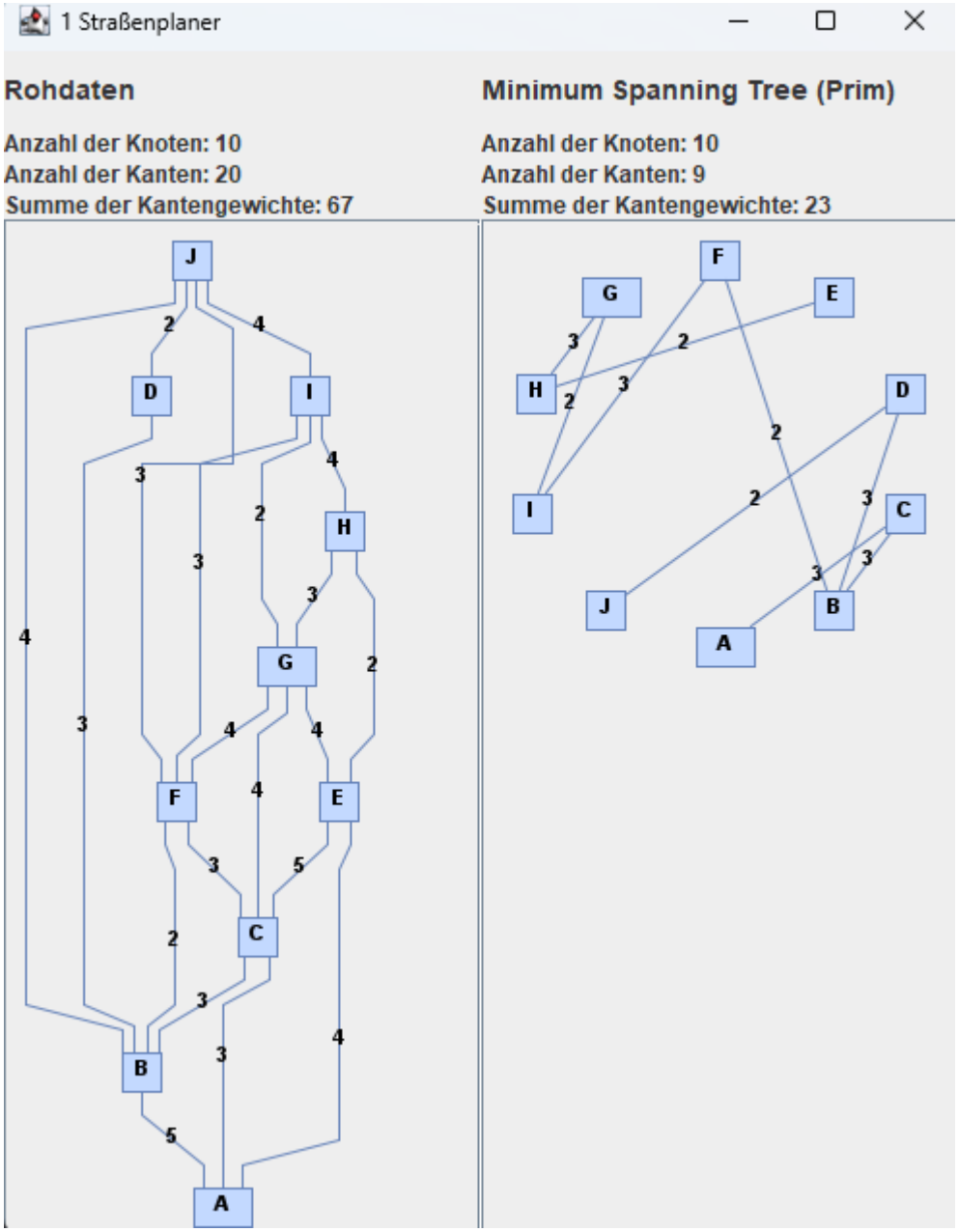

```
// code/data/problem1.txt
  A B C D E F G H I J
A 0
B 5 0
C 3 3 0
D 0 3 0 0
E 4 0 5 0 0
F 0 2 3 0 0 0
G 0 0 4 0 4 4 0
H 0 0 0 0 2 0 3 0
I 0 0 0 0 0 3 2 4 0
J 0 4 0 2 0 3 0 0 4 0
```

2.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `1 Straßenplaner.txt` geschrieben. 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:

- Die Summe der Kantengewichte muss minimal sein.
- Alle Knoten müssen über Kanten erreichbar sein.
- Der Graph muss zusammenhängend und zyklusfrei sein.
- Die Kanten müssen ungerichtet sein.
- Alle Knoten des Eingabe-Graphen müssen im Ausgabe-Graphen enthalten sein.




```
// code/output/1 Straßenplaner.txt
  A B C D E F G H I J
A 0
B 0 0
C 3 3 0
D 0 3 0 0
E 0 0 0 0 0
F 0 2 0 0 0 0
G 0 0 0 0 0 0 0
H 0 0 0 0 2 0 3 0
I 0 0 0 0 0 3 2 0 0
J 0 0 0 2 0 0 0 0 0 0
```

2.4 Geeignete Algorithmen

Es gibt verschiedene Algorithmen, die verwendet werden können, um den minimalen Spannbaum eines ungerichteten Graphen zu berechnen. Einige dieser Algorithmen sind:

Kruskal-Algorithmus: Dieser Algorithmus sortiert alle Kanten des Graphen nach ihrem Gewicht und fügt sie dann eine nach der anderen dem minimalen Spannbaum hinzu, wobei sichergestellt wird, dass keine Schleife entsteht. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

Prim-Algorithmus: Dieser Algorithmus beginnt mit einem beliebigen Knoten des Graphen und fügt nacheinander Kanten hinzu, die den aktuellen minimalen Spannbaum mit einem neuen Knoten verbinden. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

Borůvka-Algorithmus: Dieser Algorithmus ist eine Variation des Kruskal-Algorithmus und wird häufig verwendet, wenn der Graph sehr groß ist. Der Algorithmus sortiert die Kanten des Graphen nicht nach ihrem Gewicht, sondern sucht in jedem Schritt nach der leichtesten Kante, die einen Knoten mit dem minimalen Spannbaum verbindet. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

2.5 Die Laufzeit des Algorithmus

Die Laufzeit der Funktion `prim()` hängt von der Anzahl der Knoten (V) und der Anzahl der Kanten (E) im Graph ab.

Die Funktion `getEdges(matrix, vertexLetters)` hat eine Laufzeit von O(V^2), da sie eine Schleife über alle V^2 möglichen Kanten des Graphs durchführt.

Die Funktion `getNeighbors(u, vertices, edges)` hat eine Laufzeit von O(V * E), da sie eine Schleife über alle V Vertices und eine Schleife über alle E Kanten durchführt, um alle Nachbarn von u zu finden.

Die while-Schleife hat eine Laufzeit von O(V), da alle Knoten in der Prioritätswarteschlange einmal durchlaufen werden können. Innerhalb der while-Schleife wird ein Element aus der Warteschlange genommen O(log(V)) und die Funktion `getNeighbors()` aufgerufen.

Daraus resultiert eine Laufzeit von O(V^2) + O(V) * (O(log(V) + O(V * E)). Umgeformt ergibt sich eine Laufzeit von O(V^2 + V^2 * E + V * log(V)).

Dies kann auf O(V^2 * E) vereinfacht werden.

2.6 Die Implementierung des Algorithmus

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

Zuerst wird eine Liste aller Knoten und Kanten erstellt. Die Knoten werden mit dem maximalen Wert für Integer und ohne Vorgänger initialisiert. Anschließend wird ein beliebiger Knoten als Startknoten gewählt (In diesem Fall der Erste). Der Startknoten bekommt den Wert `0`. Alle Knoten werden in eine Prioritätswarteschlange `Q` eingefügt.

Danach wird eine while-Schleife verwendet um alle Knoten zu durchlaufen. In der Schleife wird der Knoten mit dem kleinsten Wert aus der Warteschlange `Q` entfernt. Anschließend wird für jeden Nachbarknoten des aktuellen Knotens überprüft, ob der Wert des Nachbarknotens größer als der Wert des aktuellen Knotens plus der Kosten der Kante ist. Wenn dies der Fall ist, wird der Wert des Nachbarknotens auf den Wert des aktuellen Knotens plus die Kosten der Kante gesetzt und der Vorgänger des Nachbarknotens auf den aktuellen Knoten gesetzt.

Am Ende wird die Liste aller Knoten in eine Adjazenzmatrix umgewandelt und zurückgegeben.

```
private int[][] prim(int[][] matrix, char[] vertexLetters) {

  ArrayList<Vertex> vertices = new ArrayList<>();
  ArrayList<Edge> edges = getEdges(matrix, vertexLetters); // O(V^2)

  // Generiere eine Liste aller Knoten mit dem Wert unendlich und ohne Vorgänger
  for (int i = 0; i < matrix.length; i++)
    vertices.add(new Vertex(vertexLetters[i], Integer.MAX_VALUE, null));

  // Starte mit beliebigem Startknoten, Startknoten bekommt den Wert 0
  vertices.get(0).setKey(0);

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

  // Solange es noch Knoten in Q gibt...
  while (!q.isEmpty()) {
    // Entnehme den Knoten mit dem kleinsten Wert
    Vertex u = q.poll();

    // Für jeden Nachbarn n von u
    for (Vertex n : getNeighbors(u, vertices, edges)) { // O(V * E)
      // Finde die Kante (u, n)
      Edge e = null;
      for (Edge edge : edges)
        if ((edge.getSource() == u.getLetter() && edge.getTarget() == n.getLetter())
            || (edge.getSource() == n.getLetter() && edge.getTarget() == u.getLetter()))
          e = edge;

      // Wenn n in Q und das Gewicht der Kante (u, n) kleiner ist als der Wert von n
      if (!q.contains(n) || e.getWeight() >= n.getKey())
        continue;

      // Setze den Wert von n auf das Gewicht der Kante (u, n)
      n.setKey(e.getWeight());
      // Setze den Vorgänger von n auf u
      n.setPredecessor(u);
      // Aktualisiere die Position von n in Q
      q.remove(n);
      q.add(n);
    }
  }

  // Erstelle die Adjazenzmatrix für den Minimum Spanning Tree
  int[][] matrix_output = new int[matrix.length][matrix.length];
  for (Vertex v : vertices) {
    if (v.getPredecessor() == null)
      continue;

    int i = v.getLetter() - 'A';
    int j = v.getPredecessor().getLetter() - 'A';
    matrix_output[i][j] = matrix[i][j];
    matrix_output[j][i] = matrix[j][i];
  }
  return matrix_output;
}
```

3. Problem 2 - "Wasserversorgung"

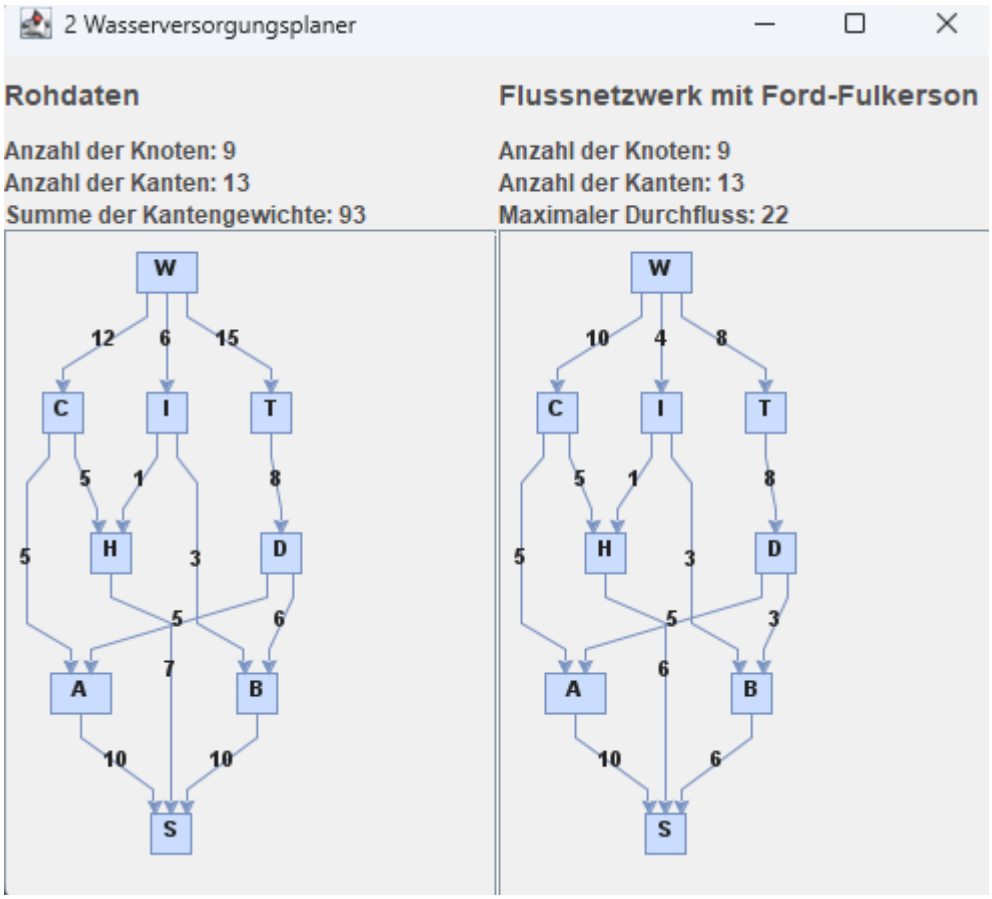
Der Straßenbau in der Graphschaft Schilda war erfolgreich, die Stadt blüht und gedeiht wieder! Selbst ein neuer Supermarkt soll eröffnet werden. Nun muss dieser aber mit Wasser versorgt werden, und da Sie bereits das Straßenbauprojekt so erfolgreich durchgeführt haben, werden Sie nun auch damit beauftragt,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

Es gibt verschiedene Algorithmen, die verwendet werden können, um den maximalen Fluss in einem gerichteten Graph zu berechnen. Einige dieser Algorithmen sind:

Ford-Fulkerson-Algorithmus: Dieser Algorithmus ist ein iterativer Algorithmus, der in jedem Schritt den Fluss durch einen Pfad erhöht, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Der Algorithmus endet, wenn kein solcher Pfad mehr existiert.

Dinic-Algorithmus: Dieser Algorithmus ist ebenfalls ein iterativer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zum Ford-Fulkerson-Algorithmus verwendet der Dinic-Algorithmus jedoch eine Heuristik, um schneller zum Ergebnis zu gelangen.

Edmonds-Karp-Algorithmus: Dieser Algorithmus ist eine Variation des Ford-Fulkerson-Algorithmus und verwendet auch eine Heuristik, um schneller zum Ergebnis zu gelangen. Im Gegensatz zum Dinic-Algorithmus verwendet der Edmonds-Karp-Algorithmus jedoch eine Breitensuche statt einer Tiefensuche, um Pfade im Graph zu finden.

Preflow-Push-Algorithmus: Dieser Algorithmus ist ein schneller, parallelisierbarer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zu den anderen Algorithmen, die hier aufgeführt sind, ist der Preflow-Push-Algorithmus jedoch nicht iterativ, sondern arbeitet in einem einzelnen Durchgang.

3.5 Die Laufzeit des Algorithmus

Die Laufzeit der Funktion `bfs()` ist O(V + E). In jedem Schritt wird ein Knoten aus der Warteschlange entfernt und die Nachbarknoten des Knotens werden in die Warteschlange aufgenommen. Da jeder Knoten nur einmal in die Warteschlange aufgenommen wird und jede Kante nur einmal betrachtet wird, beträgt die Laufzeit O(V + E).

Die Laufzeit des Ford-Fulkerson-Algorithmus ist O(V * E^2). Der Algorithmus wird in jedem Schritt iterativ ausgeführt, bis kein Pfad mehr vom Quellknoten zum Zielknoten verfügbar ist, der dessen Kapazität noch nicht vollständig ausgeschöpft hat. In jedem Schritt wird eine Breitensuche ausgeführt, um einen solchen Pfad zu finden.

Da am Ende der `fordFulkerson(int[][] matrix)` Funktion noch eine Ausgabematrix erzeugt wird erhöht sich die Laufzeit um O(V^2). Mit der gleichen Laufzeit werden zusätzlich noch die inversen Kanten des Graphen entfernt.

```
1  // Filtere die inversen Kanten
2  for (int i = 0; i < matrix_output.length; i++)
3    for (int j = 0; j < matrix_output[i].length; j++)
4      if (matrix_output[i][j] < 0)
5        matrix_output[i][j] = 0;
```

Daraus folgt eine Laufzeit von O(V * E^2 + V^2).

3.6 Die Implementierung des Algorithmus

Zur Lösung des Problems wurde der Ford-Fulkerson-Algorithmus verwendet. Genauer gesagt wurde der Edmonds-Karp-Algorithmus verwendet, da dieser eine Breitensuche verwendet, um Pfade im Graph zu finden.

Zuerst wird die Matrix `matrix` in eine echte Kopie `output` kopiert. Die echte Kopie wird später als Ausgabe verwendet.

Danach wird ein Eltern-Array `parent` erstellt, das die Elternknoten der Knoten im Graph speichert. Dieses Array wird später verwendet, um den Pfad vom Quellknoten zum Zielknoten zu finden.

Als nächstes wird eine Breitensuche ausgeführt, um einen Pfad vom Quellknoten zum Zielknoten zu finden, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Die Breitensuche wird durch die Funktion `bfs()` ausgeführt. Die Funktion `bfs()` gibt `true` zurück, wenn ein Pfad gefunden wurde, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Andernfalls wird `false` zurückgegeben.

Als Datenstruktur der `bfs()` Funktion wird eine LinkedList verwendet. Die Laufzeit der `poll()` Funktion beträgt O(1), da die LinkedList eine doppelt verkettete Liste ist. Die Laufzeit der `add()` Funktion beträgt ebenfalls O(1), da die LinkedList eine doppelt verkettete Liste ist.

Wenn ein solcher Pfad gefunden wurde, wird der minimale Fluss des Pfades berechnet. Der minimale Fluss des Pfades ist die kleinste Kapazität, die noch nicht vollständig ausgeschöpft ist. Dieser Wert wird dann zum maximalen Fluss des Graphen addiert.

Zuletzt wird eine neue Ausgabematrix erstellt, die nur aus dem positiven Fluss des Graphen besteht.

```
95  /**
96   * Laufzeit: O(V * E^2 + V^2)
97   *
98   * @param matrix
99   * @return
100  */
101 private int[][] fordFulkerson(int[][] matrix) {
102
103   // Anzahl der Knoten im Graph
104   int nodes = matrix[0].length;
105   // Die Quelle is der erste Knoten
106   int source = 0;
107   // Die Senke ist der letzte Knoten
108   int sink = nodes - 1;
      // Flow ist zu Beginn 0
      max_flow = 0;

      int u, v;

      // Erzeuge echte Kopie der Matrix für Output
      int output[][] = new int[nodes][nodes];
      for (u = 0; u < nodes; u++)
        for (v = 0; v < nodes; v++)
          output[u][v] = matrix[u][v];

      // Erzeuge ein Eltern Array zum speichern der möglichen BFS-Pfade
      int parent[] = new int[nodes];

      // Wenn für einen Pfad der BFS möglich ist, überprüfe seinen maximalen Fluss
      while (bfs(matrix, output, source, sink, parent)) {

        // Setze den Pfad Fluss auf unendlich
        int path_flow = Integer.MAX_VALUE;
        // Finde den maximalen Fluss durch die möglichen Pfade
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          path_flow = Math.min(path_flow, output[v][u]);
        }

        // aktualisiere die Kanten aus dem Eltern Array
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          // Ziehe den Fluss-Pfad den Kanten ab
          output[v][u] -= path_flow;
          // Addiere den Fluss-Pfad auf die Inversen Kanten
          output[u][v] += path_flow;
        }

        // Addiere die einzelnen Flusspfade auf den maximalen Fluss
        max_flow += path_flow;
      }

      // Ziehe von der Eingabe Matrix die übrigen Flussgewichte ab
      int[][] outputGraph = new int[nodes][nodes];
      for (int i = 0; i < matrix[0].length; i++)
        for (int j = 0; j < matrix[0].length; j++)
          outputGraph[i][j] = matrix[i][j] - output[i][j];

      return outputGraph;
    }

    /**
     * Laufzeit: O(V + E)
     *
     * @param matrix
     * @param output
     * @param s
     * @param t
     * @param parent
     * @return
     */
    private boolean bfs(int[][] matrix, int output[][], int s, int t, int parent[]) {

      // Anzahl der Knoten im Graph
      int nodes = matrix[0].length;

      // Array das alle Knoten als nicht besucht markiert
      boolean visited[] = new boolean[nodes];
      for (int i = 0; i < nodes; ++i)
        visited[i] = false;

      // Warteschlange, die besuchte Knoten als true markiert
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.add(s);
      visited[s] = true;
      parent[s] = -1;

      // Standard BFS-Loop, entfernt Knoten aus der Warteschlange die ungleich 0 sind
      while (queue.size() != 0) {
        int u = queue.poll();

        for (int v = 0; v < nodes; v++) {
          if (visited[v] == false && output[u][v] > 0) {
            // Wenn wir einen möglichen Pfad von s nach t finden geben wir true zurück
```

```
        if (v == t) {
          parent[v] = u;
          return true;
        }
        // Wenn wir keinen möglichen Pfad finden fügen wir den Knoten zur Warteschlange
        // und markieren ihn als besichtigt
        queue.add(v);
        parent[v] = u;
        visited[v] = true;
      }
    }
  }
  return false;
}
```

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 einem Graphen, der aus Kanten und Knoten besteht. Diese werden aus einer `.txt` Datei gelesen und in eine Instanz der Klasse `AdjazenzMatrix.java` geladen. Diese Instanz dient als Basis für die Berechnung des minimalen Spannbaums.

Der Eingabegraph besteht aus 12 Knoten und 66 Kanten. Die Kanten werden mit aufsteigenden Gewichten generiert. Die Knoten werden mit Buchstaben von A - L bezeichnet.

Zusätzlich ist im Code hinterlegt, dass die maximale Anzahl an Nachbarn pro Knoten 5 ist.

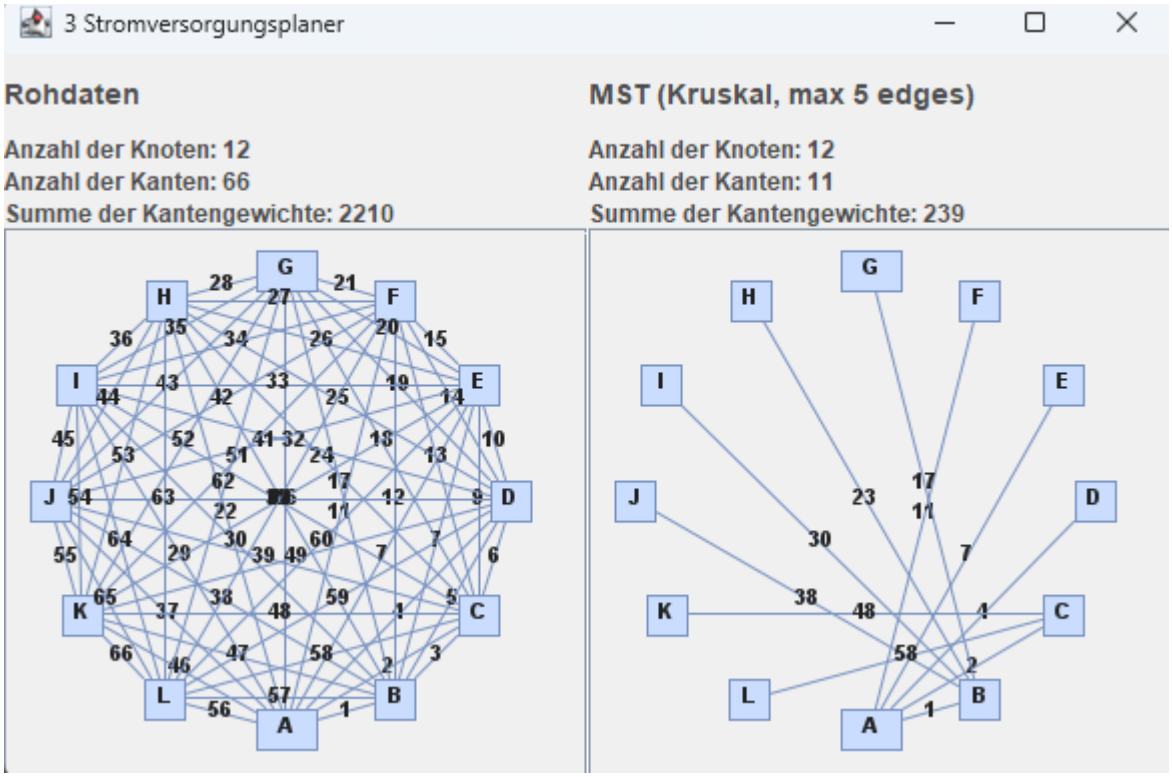
```
// code/data/problem3.txt
  A B C D E F G H I J K L
A 0
B 1 0
C 2 3 0
D 4 5 6 0
E 7 7 9 10 0
F 11 12 13 14 15 0
G 16 17 18 19 20 21 0
H 22 23 24 25 26 27 28 0
I 29 30 31 32 33 34 35 36 0
J 37 38 39 40 41 42 43 44 45 0
K 46 47 48 49 50 51 52 53 54 55 0
L 56 57 58 59 60 61 62 63 64 65 66 0
```

4.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `3 Stromversorgung.txt` geschrieben. 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:

- Die Summe der Kantengewichte muss minimal sein.
- Die Anzahl der Kanten darf maximal 5 sein.
- Alle Knoten müssen über Kanten erreichbar sein.
- Der Graph muss zusammenhängend und zyklusfrei sein.
- Die Kanten müssen ungerichtet sein.
- Alle Knoten des Eingabe-Graphen müssen im Ausgabe-Graphen enthalten sein.




```
// code/output/3 Stromversorgungsplaner.txt
  A B C D E F G H I J K L
A 0
B 1 0
C 2 0 0
D 4 0 0 0
E 7 0 0 0 0
F 11 0 0 0 0 0
G 0 17 0 0 0 0 0
H 0 23 0 0 0 0 0 0
I 0 30 0 0 0 0 0 0 0
J 0 38 0 0 0 0 0 0 0 0
K 0 0 48 0 0 0 0 0 0 0 0
L 0 0 58 0 0 0 0 0 0 0 0 0
```

4.4 Geeignete Algorithmen

Es gibt verschiedene Algorithmen, die verwendet werden können, um den minimalen Spannbaum eines ungerichteten Graphen zu berechnen. Einige dieser Algorithmen sind:

Kruskal-Algorithmus: Dieser Algorithmus sortiert alle Kanten des Graphen nach ihrem Gewicht und fügt sie dann eine nach der anderen dem minimalen Spannbaum hinzu, wobei sichergestellt wird, dass keine Schleife entsteht. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

Prim-Algorithmus: Dieser Algorithmus beginnt mit einem beliebigen Knoten des Graphen und fügt nacheinander Kanten hinzu, die den aktuellen minimalen Spannbaum mit einem neuen Knoten verbinden. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

Borůvka-Algorithmus: Dieser Algorithmus ist eine Variation des Kruskal-Algorithmus und wird häufig verwendet, wenn der Graph sehr groß ist. Der Algorithmus sortiert die Kanten des Graphen nicht nach ihrem Gewicht, sondern sucht in jedem Schritt nach der leichtesten Kante, die einen Knoten mit dem minimalen Spannbaum verbindet. Der Algorithmus endet, wenn alle Knoten des Graphen Teil des Spannbaums sind.

Allerdings müssen diese Algorithmen modifiziert werden, da die maximale Anzahl an Nachbarn pro Knoten maximal 5 betragen darf.

4.5 Die Laufzeit des Algorithmus

Die Funktion `getEdges()` hat eine Laufzeit von O(V^2), da sie jedes Element der Adjazenzmatrix durchläuft.

Die Funktionen `getSourceVertexFromEdge()` und `getTargetVertexFromEdge()` haben beide eine Laufzeit von O(V), da sie durch die Liste der Vertices iterieren und den gesuchten Vertex suchen.

Die Funktion `getAdjacentEdges()` hat eine Laufzeit von O(E), da sie durch die Liste der Edges iteriert und jede Kante überprüft, ob sie den angegebenen Vertex enthält.

Die Hauptschleife in `kruskal()` wird so lange ausgeführt, wie es noch Kanten in forest_edges gibt, wodurch sie eine Laufzeit von O(E) hat.

Innerhalb dieser Schleife werden in einem For-Loop die Funktionen `getSourceVertexFromEdge()` und `getTargetVertexFromEdge()` aufgerufen, die jeweils eine Laufzeit von O(V) haben.

In der Hauptschleife wird zusätzlich die Funktion `getAdjacentEdges()` aufgerufen, die eine Laufzeit von O(E) hat und ein weiterer For-Loop mit der Laufzeit O(V).

Daraus resultiert eine Laufzeit von O(V^2 + E * (V^3 + E)). Dies kann auf O(V^3 * E) vereinfacht werden.

4.6 Die Implementierung des Algorithmus

Zur Lösung dieses Problems wurde der Algorithmus von Kruskal verwendet. Der Algorithmus ist ein Greedy-Algorithmus. Er berechnet den minimalen Spannbaum eines Graphen.

Zuerst wird eine Liste aller Kanten des Graphen erstellt. Diese Liste wird nach Gewicht sortiert. Dann wird ein Wald aus Bäumen erstellt, wobei jeder Knoten ein eigener Baum ist. Dann wird die Liste der Kanten durchlaufen.

Wenn die Kanten zwei Knoten aus verschiedenen Bäumen verbindet, wird die Kante dem minimalen Spannbaum hinzugefügt. Die Bäume werden dann zusammengeführt.

Die Funktion gibt eine Adjazenmatrix zurück, die aus der Liste der Ausgabe-Kanten erstellt wird.

```
private int[][] kruskal(int[][] matrix, char[] vertexLetters, int max_edges) {

  ArrayList<Vertex> vertices = new ArrayList<>();

  // Generiere eine Liste aller Knoten
  for (int i = 0; i < matrix.length; i++)
    vertices.add(new Vertex(vertexLetters[i], 0));

  ArrayList<Edge> edges = getEdges(matrix, vertexLetters); // O(V^2)

  // Sortiere die Kanten nach Gewicht
  edges.sort(Comparator.comparingInt(Edge::getWeight));

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

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

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

  // 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
    Edge e = forest_edges.remove(0);

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

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

    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 (Vertex v : tree_v)
      tree_u.add(v);

    forest.remove(tree_v);
  }

  // erstelle die Ausgabe-Adjazenzmatrix
  int[][] output_matrix = new int[matrix.length][matrix.length];
  for (Edge e : output_edges) {
    int source = e.getSource() - 'A';
    int target = e.getTarget() - 'A';
    output_matrix[source][target] = matrix[source][target];
    output_matrix[target][source] = matrix[target][source];
  }
  return output_matrix;
}
```

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.

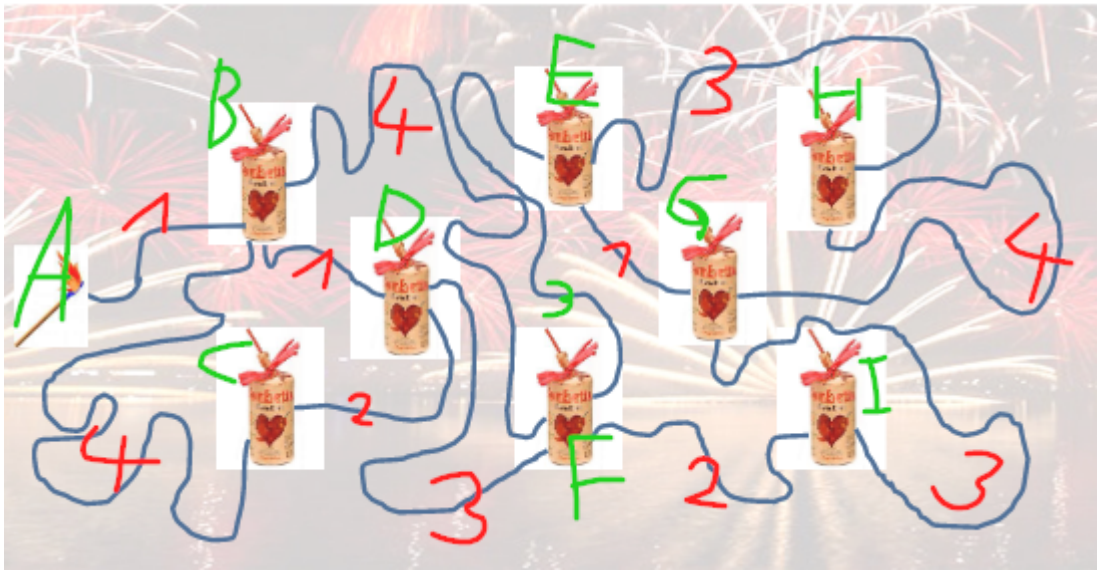
Es wird die korrekte Reihenfolge gesucht, in der die Feuerwerkskörper gezündet werden wenn das Streichholz den ersten Feuerwerkskörper zündet.

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 Zündschnüre vorgenommen und in einer Grafik dargestellt. Die Knoten wurden mit Buchstaben von A - I beschriftet.

Die Eingabe besteht aus einem Graphen, der aus Kanten und Knoten besteht. Diese werden aus einer `.txt` Datei gelesen und in eine Instanz der Klasse `AdjazenzMatrix.java` geladen. Diese Instanz dient als Basis für die Berechnung des minimalen Spannbaums.



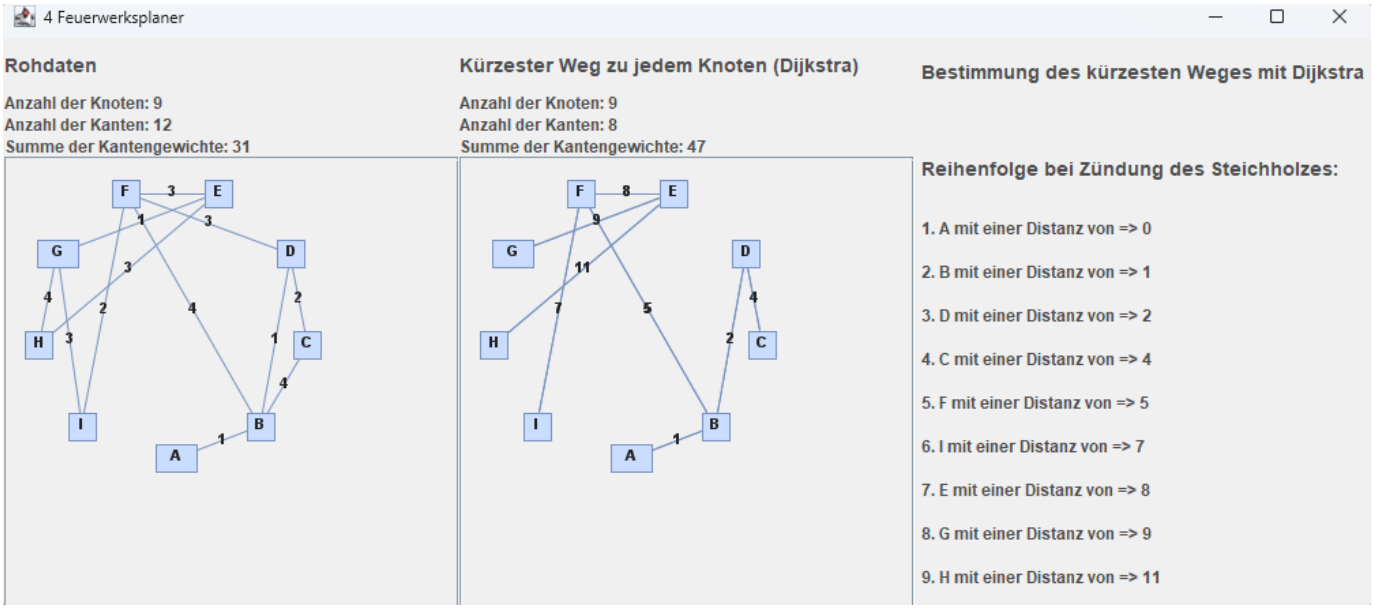

```
// code/data/problem4.txt
  A B C D E F G H I
A 0
B 1 0
C 0 4 0
D 0 1 2 0
E 0 0 0 0 0
F 0 4 0 3 3 0
G 0 0 0 0 1 0 0
H 0 0 0 0 3 0 4 0
I 0 0 0 0 0 2 3 0 0
```

5.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `4 Feuerwerksplaner.txt` geschrieben. Das Fenster besteht aus drei Teilen. Auf der linken Seite wird der Eingabegraph dargestellt. In der Mitte wird der berechnete Graph dargestellt mit dem kürzesten Weg von A zu jedem anderen Knoten. Die Gewichte der Kanten zu den jeweiligen Knoten stellen die minimalen Gesamtkosten zum jeweiligen Knoten. Im rechten Teil wird eine Liste der Knoten dargestellt, in der die Knoten in der Reihenfolge aufgelistet sind, in der sie gezündet werden.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- Die Kantengewichte müssen die minimalen Gesamtkosten zum jeweiligen Knoten sein.
- Alle Knoten müssen über Kanten erreichbar sein.
- Der Graph muss zusammenhängend und zyklusfrei sein.
- Die Kanten müssen ungerichtet sein.
- Alle Knoten des Eingabe-Graphen müssen im Ausgabe-Graphen enthalten sein.




```
// code/output/4 Feuerwerksplaner.txt
  A B C D E F G H I
A 0
B 1 0
C 0 0 0
D 0 2 4 0
E 0 0 0 0 0
F 0 5 0 0 8 0
G 0 0 0 0 9 0 0
H 0 0 0 0 11 0 0 0
I 0 0 0 0 0 7 0 0 0
```

5.4 Geeignete Algorithmen

Für dieses Problem eignen sich die Algorithmen Dijkstra, Bellman-Ford und A*.

Der Bellman-Ford-Algorithmus ist ein dynamischer Algorithmus, der iterativ die Entfernungen von dem Startknoten zu allen anderen Knoten im Graphen aktualisiert, bis sie stabil geworden sind. Der Dijkstra-Algorithmus hingegen ist ein statischer Algorithmus, der alle Entfernungen auf einmal berechnet.

Der A *Algorithmus ist ein Best-First-Suchalgorithmus, der dazu verwendet wird, den kürzesten Pfad von einem Startknoten zu einem Zielknoten in einem Graphen zu finden. Der A*-Algorithmus verwendet dabei eine Heuristik, um zu entscheiden, welche Knoten als nächstes untersucht werden sollen. Diese Heuristik basiert auf einer Schätzung der Entfernung des Knotens vom Ziel

und wird verwendet, um den Algorithmus dazu zu bringen, sich auf die Knoten zu konzentrieren, die wahrscheinlich zum Ziel führen.

5.5 Die Laufzeit des Algorithmus

Die Laufzeit der Funktion `dijkstra()` hängt von der Anzahl der Knoten (V) und der Anzahl der Kanten (E) im Graph ab.

Die Funktion `getEdges(matrix, vertexLetters)` hat eine Laufzeit von O(V^2), da sie eine Schleife über alle V^2 möglichen Kanten des Graphs durchführt.

Die Funktion `getNeighbors(u, vertices, edges)` hat eine Laufzeit von O(V * E), da sie eine Schleife über alle V Vertices und eine Schleife über alle E Kanten durchführt, um alle Nachbarn von u zu finden.

Die Funktion `getWeightSum(u, v, edges)` hat eine Laufzeit von O(E), da sie eine Schleife über alle E Kanten durchführt, um die Gewichte zu addieren.

Die while-Schleife hat eine Laufzeit von O(V), da alle Knoten in der Prioritätswarteschlange einmal durchlaufen werden können. Innerhalb der while-Schleife wird ein Element aus der Warteschlange genommen O(log(V)) und die Funktion `getNeighbors()` aufgerufen. Für jeden Nachbarn wird die Funktion `getWeightSum()` aufgerufen.

Daraus resultiert eine Laufzeit von O(V^2) + O(V) * (O(log(V) + (O(V * E) * O(E))). Umgeformt ergibt sich eine Laufzeit von O(V^2 + V * log(V) + V^3 * E).

Dies kann auf O(V^3 * E) vereinfacht werden.

5.6 Die Implementierung des Algorithmus

Zur Lösung des Problems wurde der Dijkstra-Algorithmus verwendet. Der Dijkstra-Algorithmus ist ein Greedy-Algorithmus.

Der Algorithmus verwendet dabei eine Prioritätswarteschlange, um die Knoten zu sortieren, die als nächstes untersucht werden sollen. Die Prioritätswarteschlange wird mit den Knoten initialisiert, die direkt mit dem Startknoten verbunden sind. Die Knoten werden dann in der Prioritätswarteschlange nach ihrer Entfernung vom Startknoten sortiert. Der Algorithmus wählt dann den Knoten mit der geringsten Entfernung aus der Prioritätswarteschlange aus und aktualisiert die Entfernungen aller Knoten, die mit diesem Knoten verbunden sind.

Dadurch bekommt jeder Knoten einen Wert, der die Entfernung vom Startknoten angibt. Der Algorithmus wird dann wiederholt, bis alle Knoten in der Prioritätswarteschlange untersucht wurden.

```
private int[][] dijkstra(int[][] matrix, char[] vertexLetters) {

  // Generiere eine Liste aller Knoten
  for (int i = 0; i < matrix.length; i++)
    vertices.add(new Vertex(vertexLetters[i], 0));

  ArrayList<Edge> edges = getEdges(matrix, vertexLetters); // O(V^2)

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

  // Speichere alle Knoten in einer Prioritätswarteschlange queue
  PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>(
      Comparator.comparingInt(Vertex::getKey));
  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
    Vertex 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 (Vertex n : 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. )
      int sum = v.getKey() + getWeightSum(v, n, edges);

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

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

  // Sortiere Knoten nach Distanz
  vertices.sort(Comparator.comparingInt(Vertex::getKey));

  // Erstelle eine neue Adjazenzmatrix, die den jeweils kürzesten Weg zu jedem
  // Knoten enthält
  int[][] matrix_output = new int[matrix.length][matrix.length];

  for (Vertex vertex : vertices) {
    if (vertex.getPredecessor() == null)
      continue;
    matrix_output[vertex.getPredecessor().getLetter() - 'A'][vertex.getLetter() - 'A'] = vertex.getKey();
    matrix_output[vertex.getLetter() - 'A'][vertex.getPredecessor().getLetter() - 'A'] = vertex.getKey();
  }

  return matrix_output;
}
```

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 Alrogithmus

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

Das Problem lässt sich als Graphenmodell mit gerichteten Kanten darstellen. Die Autobahn, die Ampeln und der Parkplatz sind die Knoten, die Straßen sind die Kanten. Die Kosten der Kanten sind die Kapazität für Fahrzeuge der Straßen.

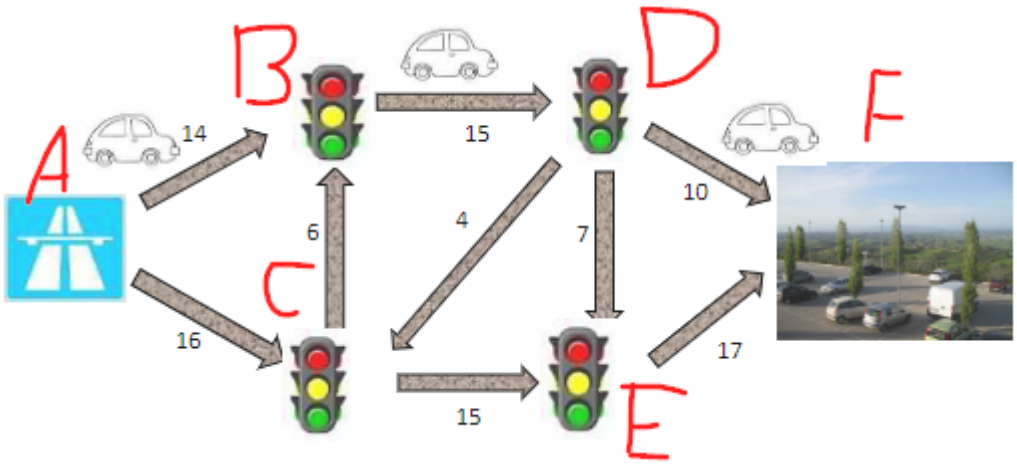
Es wird die jeweilige Anzahl an Fahrzeugen gesucht, die über die einzelnen Kanten geführt werden sollen, damit die Fahrzeuge ohne Stau zum Parkplatz gelangen.

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.

7.2 Die Eingabe

Das Bild der Aufgabenstellung wurde mit Buchstaben von A - F beschriftet und dienen als Bezeichnung für die Knoten. Die Werte der Kanten werden auch aus der Grafik übernommen.

Die Eingabe besteht aus einem Graphen, der aus Kanten und Knoten besteht. Diese werden aus einer `.txt` Datei gelesen und in eine Instanz der Klasse `AdjazenzMatrix.java` geladen. Diese Instanz dient als Basis für die Berechnung des minimalen Spannbaums.




```
// code/data/problem6.txt
  A B C D E F
A 0 14 16 0 0 0
B 0 0 0 15 0 0
C 0 6 0 0 15 0
D 0 0 4 0 7 10
E 0 0 0 0 0 17
F 0 0 0 0 0 0
```

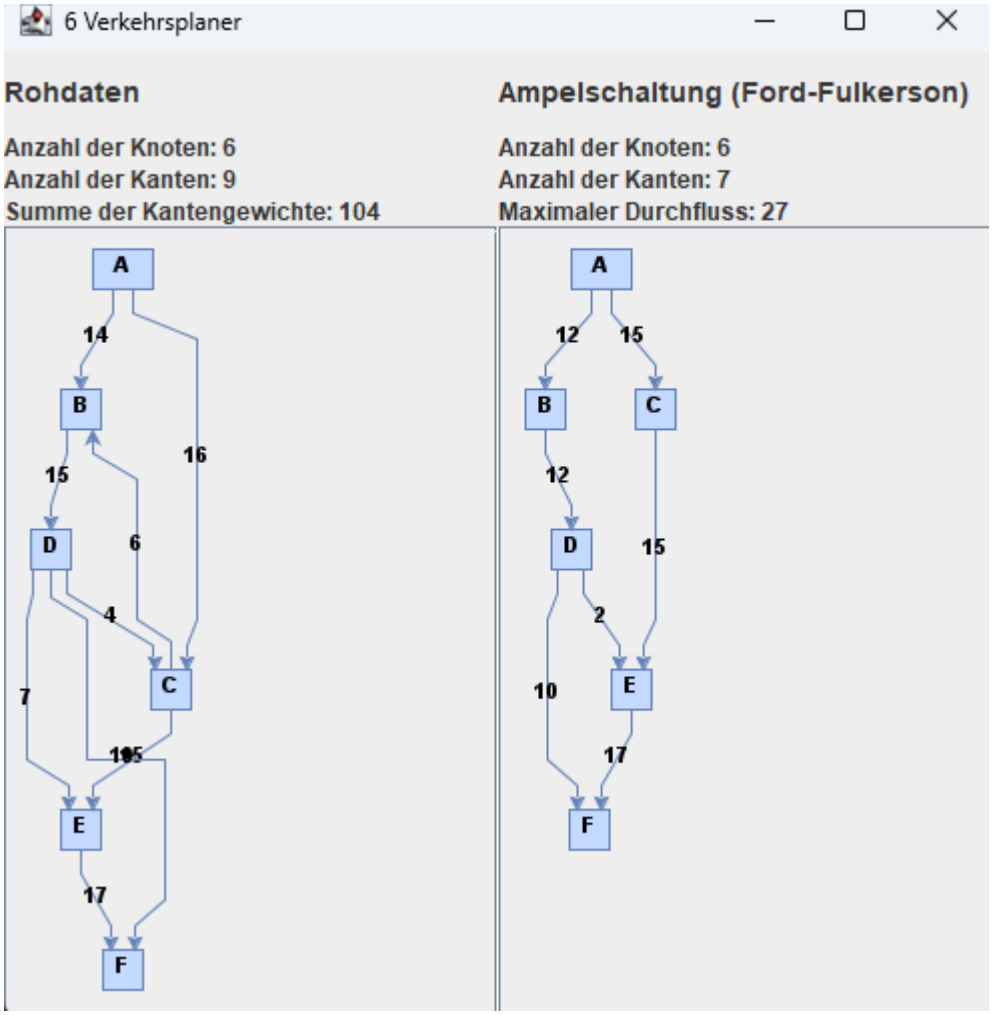
7.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `6 Verkehrsplaner.txt` geschrieben. Das Fenster besteht zwei Hälften. Auf der linken Seite wird der Eingabegraph dargestellt. Auf der rechten Seite wird der Ausgabegraph dargestellt. In diesem Graphen entsprechen die Werte der Kanten die Anzahl der Autos, die über die jeweilige Straße geführt werden sollen.

Vor der Ausgabe werden die inversen/negativen Kanten entfernt. Diese werden nicht benötigt, da sie keine Rolle spielen.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- Alle Knoten müssen von dem ersten Knoten A erreichbar sein.
- Der Fluss der Kanten muss positiv und ganzzahlig sein.
- Der Fluss der Kanten darf nicht größer als deren Kapazität sein.
- Der maximale Fluss muss gleich der Summe der Kapazitäten der Kanten sein, die von A ausgehen und in dem letzten Knoten enden.
- Der maximale Fluss muss maximal sein, also es darf keine andere Flussverteilung geben, die einen höheren Fluss ergibt.
- Der Graph muss zusammenhängend und zyklusfrei sein.
- Die Kanten müssen gerichtet sein.
- Alle Knoten des Eingabe-Graphen müssen im Ausgabe-Graphen enthalten sein.




```
// code/output/6 Verkehrsplaner.txt
  A B C D E F
A 0 12 15 0 0 0
B 0 0 0 12 0 0
C 0 0 0 0 15 0
D 0 0 0 0 2 10
E 0 0 0 0 0 17
F 0 0 0 0 0 0
```

7.4 Geeignete Algorithmen

Es gibt verschiedene Algorithmen, die verwendet werden können, um den maximalen Fluss in einem gerichteten Graph zu berechnen. Einige dieser Algorithmen sind:

Ford-Fulkerson-Algorithmus: Dieser Algorithmus ist ein iterativer Algorithmus, der in jedem Schritt den Fluss durch einen Pfad erhöht, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Der Algorithmus endet, wenn kein solcher Pfad mehr existiert.

Dinic-Algorithmus: Dieser Algorithmus ist ebenfalls ein iterativer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zum Ford-Fulkerson-Algorithmus verwendet der Dinic-Algorithmus jedoch eine Heuristik, um schneller zum Ergebnis zu gelangen.

Edmonds-Karp-Algorithmus: Dieser Algorithmus ist eine Variation des Ford-Fulkerson-Algorithmus und verwendet auch eine Heuristik, um schneller zum Ergebnis zu gelangen. Im Gegensatz zum Dinic-Algorithmus verwendet der Edmonds-Karp-Algorithmus jedoch eine Breitensuche statt einer Tiefensuche, um Pfade im Graph zu finden.

Preflow-Push-Algorithmus: Dieser Algorithmus ist ein schneller, parallelisierbarer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zu den anderen Algorithmen, die hier aufgeführt sind, ist der Preflow-Push-Algorithmus jedoch nicht iterativ, sondern arbeitet in einem einzelnen Durchgang.

7.5 Die Laufzeit des Algorithmus

Die Laufzeit der Funktion `bfs()` ist O(V + E). In jedem Schritt wird ein Knoten aus der Warteschlange entfernt und die Nachbarknoten des Knotens werden in die Warteschlange aufgenommen. Da jeder Knoten nur einmal in die Warteschlange aufgenommen wird und jede Kante nur einmal betrachtet wird, beträgt die Laufzeit O(V + E).

Die Laufzeit des Ford-Fulkerson-Algorithmus ist O(V * E^2). Der Algorithmus wird in jedem Schritt iterativ ausgeführt, bis kein Pfad mehr vom Quellknoten zum Zielknoten verfügbar ist, der dessen Kapazität noch nicht vollständig ausgeschöpft hat. In jedem Schritt wird eine Breitensuche ausgeführt, um einen solchen Pfad zu finden.

Da am Ende der `fordFulkerson(int[][] matrix)` Funktion noch eine Ausgabematrix erzeugt wird erhöht sich die Laufzeit um O(V^2). Mit der gleichen Laufzeit werden zusätzlich noch die inversen Kanten des Graphen entfernt.

```
// Filtere die inversen Kanten
for (int i = 0; i < matrix_output.length; i++)
  for (int j = 0; j < matrix_output[i].length; j++)
    if (matrix_output[i][j] < 0)
      matrix_output[i][j] = 0;
```

Daraus folgt eine Laufzeit von O(V * E^2 + V^2).

7.6 Die Implementierung des Algorithmus

Zur Lösung des Problems wurde der Ford-Fulkerson-Algorithmus verwendet. Genauer gesagt wurde der Edmonds-Karp-Algorithmus verwendet, da dieser eine Breitensuche verwendet, um Pfade im Graph zu finden.

Zuerst wird die Matrix `matrix` in eine echte Kopie `output` kopiert. Die echte Kopie wird später als Ausgabe verwendet.

Danach wird ein Eltern-Array `parent` erstellt, das die Elternknoten der Knoten im Graph speichert. Dieses Array wird später verwendet, um den Pfad vom Quellknoten zum Zielknoten zu finden.

Als nächstes wird eine Breitensuche ausgeführt, um einen Pfad vom Quellknoten zum Zielknoten zu finden, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Die Breitensuche wird durch die Funktion `bfs()` ausgeführt. Die Funktion `bfs()` gibt `true` zurück, wenn ein Pfad gefunden wurde, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Andernfalls wird `false` zurückgegeben.

Als Datenstruktur der `bfs()` Funktion wird eine LinkedList verwendet. Die Laufzeit der `poll()` Funktion beträgt O(1), da die LinkedList eine doppelt verkettete Liste ist. Die Laufzeit der `add()` Funktion beträgt ebenfalls O(1), da die LinkedList eine doppelt verkettete Liste ist.

Wenn ein solcher Pfad gefunden wurde, wird der minimale Fluss des Pfades berechnet. Der minimale Fluss des Pfades ist die kleinste Kapazität, die noch nicht vollständig ausgeschöpft ist. Dieser Wert wird dann zum maximalen Fluss des Graphen addiert.

Zuletzt wird eine neue Ausgabematrix erstellt, die nur aus dem positiven Fluss des Graphen besteht.

```
95  /**
96   * Laufzeit: O(V * E^2 + V^2)
97   *
98   * @param matrix
99   * @return
100  */
101 private int[][] fordFulkerson(int[][] matrix) {
102
103   // Anzahl der Knoten im Graph
104   int nodes = matrix[0].length;
105   // Die Quelle is der erste Knoten
106   int source = 0;
107   // Die Senke ist der letzte Knoten
108   int sink = nodes - 1;
      // Flow ist zu Beginn 0
      max_flow = 0;

      int u, v;

      // Erzeuge echte Kopie der Matrix für Output
      int output[][] = new int[nodes][nodes];
      for (u = 0; u < nodes; u++)
        for (v = 0; v < nodes; v++)
          output[u][v] = matrix[u][v];

      // Erzeuge ein Eltern Array zum speichern der möglichen BFS-Pfade
      int parent[] = new int[nodes];

      // Wenn für einen Pfad der BFS möglich ist, überprüfe seinen maximalen Fluss
      while (bfs(matrix, output, source, sink, parent)) {

        // Setze den Pfad Fluss auf unendlich
        int path_flow = Integer.MAX_VALUE;
        // Finde den maximalen Fluss durch die möglichen Pfade
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          path_flow = Math.min(path_flow, output[v][u]);
        }

        // aktualisiere die Kanten aus dem Eltern Array
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          // Ziehe den Fluss-Pfad den Kanten ab
          output[v][u] -= path_flow;
          // Addiere den Fluss-Pfad auf die Inversen Kanten
          output[u][v] += path_flow;
        }

        // Addiere die einzelnen Flusspfade auf den maximalen Fluss
        max_flow += path_flow;
      }

      // Ziehe von der Eingabe Matrix die übrigen Flussgewichte ab
      int[][] outputGraph = new int[nodes][nodes];
      for (int i = 0; i < matrix[0].length; i++)
        for (int j = 0; j < matrix[0].length; j++)
          outputGraph[i][j] = matrix[i][j] - output[i][j];

      return outputGraph;
    }

    /**
     * Laufzeit: O(V + E)
     *
     * @param matrix
     * @param output
     * @param s
     * @param t
     * @param parent
     * @return
     */
    private boolean bfs(int[][] matrix, int output[][], int s, int t, int parent[]) {

      // Anzahl der Knoten im Graph
      int nodes = matrix[0].length;

      // Array das alle Knoten als nicht besucht markiert
      boolean visited[] = new boolean[nodes];
      for (int i = 0; i < nodes; ++i)
        visited[i] = false;

      // Warteschlange, die besuchte Knoten als true markiert
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.add(s);
      visited[s] = true;
      parent[s] = -1;

      // Standard BFS-Loop, entfernt Knoten aus der Warteschlange die ungleich 0 sind
      while (queue.size() != 0) {
        int u = queue.poll();

        for (int v = 0; v < nodes; v++) {
          if (visited[v] == false && output[u][v] > 0) {
            // Wenn wir einen möglichen Pfad von s nach t finden geben wir true zurück
```

```
        if (v == t) {
          parent[v] = u;
          return true;
        }
        // Wenn wir keinen möglichen Pfad finden fügen wir den Knoten zur Warteschlange
        // und markieren ihn als besichtigt
        queue.add(v);
        parent[v] = u;
        visited[v] = true;
      }
    }
  }
  return false;
}
```

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

Das Problem lässt sich als Graphenmodell mit gerichteten Kanten darstellen. Die Mitarbeiter und Kompetenzen werden als Knoten dargestellt. Die Fähigkeit eines Mitarbeiters eine bestimmte Kompetenz zu besitzen wird als Kante dargestellt. Die Kanten werden mit der Kapazität 1 versehen.

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.

8.2 Die Eingabe

Die Buchstaben für die Mitarbeiter werden mit A - G bezeichnet und die Kompetenzen mit T - Z. Hierbei ist es wichtig anzumerken, dass die Anzahl der Mitarbeiter gleich der Anzahl der Kompetenzen sein muss und Jede Kompetenz von mindestens einem Mitarbeiter besetzt sein muss.

```
A: Frau Maier       T: Straßenbau
B: Frau Müller      U: Verkehrsplanung
C: Frau Augst       V: Archäologie
D: Frau Schmidt     W: Gesamtkoordination
E: Herr Kunze       X: Festplanung
F: Herr Hof         Y: Wasserversorgung
G: Frau Lustig      Z: Wettkampfausrichtung
```

Die Eingabe besteht aus einem Graphen, der aus Kanten und Knoten besteht. Diese werden aus einer `.txt` Datei gelesen und in eine Instanz der Klasse `AdjazenzMatrix.java` geladen. Diese Instanz dient als Basis für die Berechnung des minimalen Spannbaums.

```
// code/data/problem7.txt
  A B C D E F G T U V W X Y Z
A 0 0 0 0 0 0 0 1 1 1 0 0 0 0
B 0 0 0 0 0 0 0 0 0 0 1 1 0 0
C 0 0 0 0 0 0 0 1 0 0 0 0 1 0
D 0 0 0 0 0 0 0 1 1 0 0 0 0 1
E 0 0 0 0 0 0 0 0 0 1 0 1 0 0
F 0 0 0 0 0 0 0 1 0 0 1 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 1 0 1
T 0 0 0 0 0 0 0 0 0 0 0 0 0 0
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0
V 0 0 0 0 0 0 0 0 0 0 0 0 0 0
W 0 0 0 0 0 0 0 0 0 0 0 0 0 0
X 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Y 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Z 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

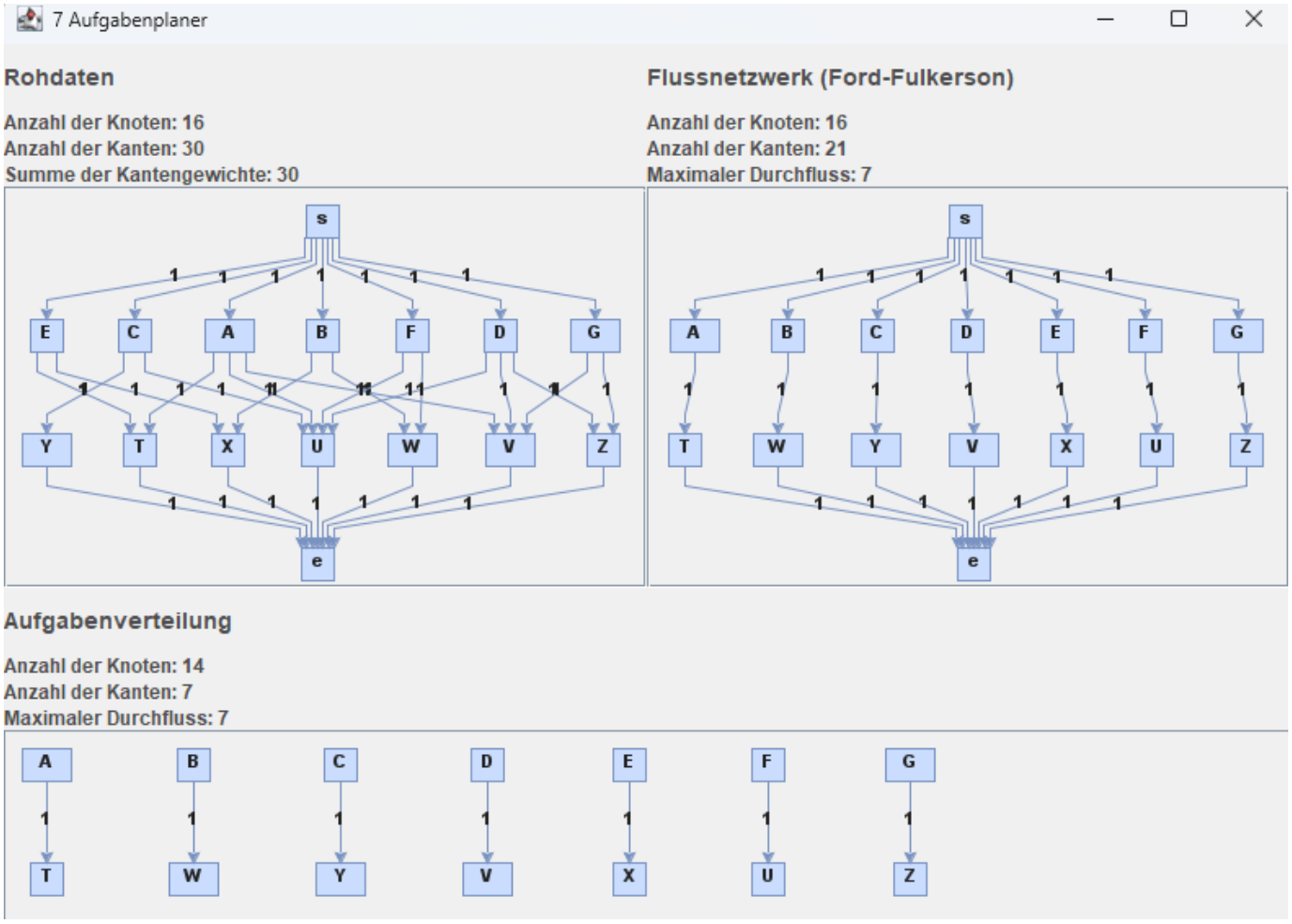
8.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `7 Aufgabenplaner.txt` geschrieben. Das Fenster besteht aus drei Teilen. Im oberen linken Teil wird der Eingabe-Graph mit zusätzlichen Start- und Endknoten dargestellt. Im oberen rechten Teil wird der Ausgabe-Graph mit Start- und Endknoten dargestellt. Im unteren Teil wird die resultierende Aufgabenverteilung ausgegeben.

Vor der Ausgabe werden die inversen/negativen Kanten, Startknoten, Endknoten und deren Kanten entfernt. Diese werden nicht benötigt, da sie keine Rolle spielen.

Ein korrekte Ausgabe erfüllt folgende Eigenschaften:

- Es müssen alle Kompetenzen und Mitarbeiter in der Ausgabe vorkommen.
- Jeder Mitarbeiter muss genau eine Kompetenz besitzen.
- Es darf keine Kante geben, die von einem Mitarbeiter zu einem anderen Mitarbeiter führt.
- Es darf keine Kante geben, die von einer Kompetenz zu einer anderen Kompetenz führt.
- Es darf keine Kante geben, die von einer Kompetenz zu einem Mitarbeiter führt.
- Alle Kanten müssen gerichtet und ungewichtet sein, also müssen den Fluss 1 besitzen




```
// code/output/7 Aufgabenplaner.txt
  A B C D E F G T U V W X Y Z
A 0 0 0 0 0 0 0 0 1 0 0 0 0 0
B 0 0 0 0 0 0 0 0 0 0 1 0 0 0
C 0 0 0 0 0 0 0 0 0 0 0 0 1 0
D 0 0 0 0 0 0 0 0 0 0 0 0 0 1
E 0 0 0 0 0 0 0 0 1 0 0 0 0 0
F 0 0 0 0 0 0 0 1 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 0 0 0 1 0 0
T 0 0 0 0 0 0 0 0 0 0 0 0 0 0
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0
V 0 0 0 0 0 0 0 0 0 0 0 0 0 0
W 0 0 0 0 0 0 0 0 0 0 0 0 0 0
X 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Y 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Z 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

8.3.1 Resultierende Aufgabenverteilung

```
A: Frau Maier     ->   U: Verkehrsplanung
B: Frau Müller    ->   W: Gesamtkoordination
C: Frau Augst     ->   Y: Wasserversorgung
D: Frau Schmidt   ->   Z: Wettkampfausrichtung
E: Herr Kunze     ->   V: Archäologie
F: Herr Hof       ->   T: Straßenbau
G: Frau Lustig    ->   X: Festplanung
```

8.4 Geeignete Algorithmen

Es gibt verschiedene Algorithmen, die verwendet werden können, um den maximalen Fluss in einem gerichteten Graph zu berechnen. Einige dieser Algorithmen sind:

Ford-Fulkerson-Algorithmus: Dieser Algorithmus ist ein iterativer Algorithmus, der in jedem Schritt den Fluss durch einen Pfad erhöht, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Der Algorithmus endet, wenn kein solcher Pfad mehr existiert.

Dinic-Algorithmus: Dieser Algorithmus ist ebenfalls ein iterativer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zum Ford-Fulkerson-Algorithmus verwendet der Dinic-Algorithmus jedoch eine Heuristik, um schneller zum Ergebnis zu gelangen.

Edmonds-Karp-Algorithmus: Dieser Algorithmus ist eine Variation des Ford-Fulkerson-Algorithmus und verwendet auch eine Heuristik, um schneller zum Ergebnis zu gelangen. Im Gegensatz zum Dinic-Algorithmus verwendet der Edmonds-Karp-Algorithmus jedoch eine Breitensuche statt einer Tiefensuche, um Pfade im Graph zu finden.

Preflow-Push-Algorithmus: Dieser Algorithmus ist ein schneller, parallelisierbarer Algorithmus, der den Fluss durch den Graph in jedem Schritt erhöht, indem er einen Pfad vom Quellknoten zum Zielknoten sucht, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Im Gegensatz zu den anderen Algorithmen, die hier aufgeführt sind, ist der Preflow-Push-Algorithmus jedoch nicht iterativ, sondern arbeitet in einem einzelnen Durchgang.

8.5 Die Laufzeit des Algorithmus

Zuerst wird aus der Eingabematrix eine neue Matrix erstellt, die einen zusätzlichen Start- und Endknoten enthält. Für jeden Knoten aus der ersten Hälfte der Knoten wird eine Kante vom Startknoten zu diesem Knoten hinzugefügt. Das gleiche wird umgekehrt für die zweite Hälfte der Knoten mit dem Endknoten gemacht. Die Laufzeit hierfür beträgt O(V^2).

Die Laufzeit der Funktion `bfs()` ist O(V + E). In jedem Schritt wird ein Knoten aus der Warteschlange entfernt und die Nachbarknoten des Knotens werden in die Warteschlange aufgenommen. Da jeder Knoten nur einmal in die Warteschlange aufgenommen wird und jede Kante nur einmal betrachtet wird, beträgt die Laufzeit O(V + E).

Die Laufzeit des Ford-Fulkerson-Algorithmus ist O(V * E^2). Der Algorithmus wird in jedem Schritt iterativ ausgeführt, bis kein Pfad mehr vom Quellknoten zum Zielknoten verfügbar ist, der dessen Kapazität noch nicht vollständig ausgeschöpft hat. In jedem Schritt wird eine Breitensuche ausgeführt, um einen solchen Pfad zu finden.

Da am Ende der `fordFulkerson(int[][] matrix)` Funktion noch eine Ausgabematrix erzeugt wird erhöht sich die Laufzeit um O(V^2). Mit der gleichen Laufzeit werden zusätzlich noch die inversen Kanten des Graphen entfernt.

```java
// Filtere die inversen Kanten
for (int i = 0; i < matrix_output.length; i++)
  for (int j = 0; j < matrix_output[i].length; j++)
    if (matrix_output[i][j] < 0)
      matrix_output[i][j] = 0;
```

Nachdem die inversen Kanten entfernt wurden, wird eine neue Ausgabematrix ohne Start- und Endknoten und ohne deren Kanten erstellt. Die Laufzeit hierfür beträgt O(V^2).

```
// Erstelle eine neue Ausgabe-Adjazenzmatrix ohne Start- und Endknoten und ohne
// die Kanten zu diesen Knoten
int[][] matrix_output2 = new int[matrix_output.length - 2][matrix_output[0].length - 2];
char[] vertex_letters_new2 = new char[vertex_letters_new.length - 2];

for (int i = 0; i < matrix_output2.length; i++) {
  for (int j = 0; j < matrix_output2[i].length; j++)
    matrix_output2[i][j] = matrix_output[i + 1][j + 1];
  vertex_letters_new2[i] = vertex_letters_new[i + 1];
}
```

Daraus folgt eine Laufzeit von O(V * E^2 + V^2).

8.6 Die Implementierung des Algorithmus

Zur Lösung des Problems wurde der Ford-Fulkerson-Algorithmus verwendet. Genauer gesagt wurde der Edmonds-Karp-Algorithmus verwendet, da dieser eine Breitensuche verwendet, um Pfade im Graph zu finden.

Zuerst wird aus der Eingabematrix eine neue Matrix erstellt, die einen zusätzlichen Start- und Endknoten enthält. Für jeden Knoten aus der ersten Hälfte der Knoten wird eine Kante vom Startknoten zu diesem Knoten hinzugefügt. Das gleiche wird umgekehrt für die zweite Hälfte der Knoten mit dem Endknoten gemacht.

Danach wird die Matrix `matrix` in eine echte Kopie `output` kopiert. Die echte Kopie wird später als Ausgabe verwendet.

Danach wird ein Eltern-Array `parent` erstellt, das die Elternknoten der Knoten im Graph speichert. Dieses Array wird später verwendet, um den Pfad vom Quellknoten zum Zielknoten zu finden.

Als nächstes wird eine Breitensuche ausgeführt, um einen Pfad vom Quellknoten zum Zielknoten zu finden, dessen Kapazität noch nicht vollständig ausgeschöpft ist. Die Breitensuche wird durch die Funktion `bfs()` ausgeführt. Die Funktion `bfs()` gibt `true` zurück, wenn ein Pfad gefunden wurde, der vom Quellknoten zum Zielknoten führt und dessen Kapazität noch nicht vollständig ausgeschöpft ist. Andernfalls wird `false` zurückgegeben.

Als Datenstruktur der `bfs()` Funktion wird eine LinkedList verwendet. Die Laufzeit der `poll()` Funktion beträgt O(1), da die LinkedList eine doppelt verkettete Liste ist. Die Laufzeit der `add()` Funktion beträgt ebenfalls O(1), da die LinkedList eine doppelt verkettete Liste ist.

Wenn ein solcher Pfad gefunden wurde, wird der minimale Fluss des Pfades berechnet. Der minimale Fluss des Pfades ist die kleinste Kapazität, die noch nicht vollständig ausgeschöpft ist. Dieser Wert wird dann zum maximalen Fluss des Graphen addiert.

Anschließend wird eine neue Ausgabematrix erstellt, die nur aus dem positiven Fluss des Graphen besteht.

Zuletzt wird eine neue Ausgabematrix ohne Start- und Endknoten und deren Kanten erstellt. Diese Matrix wird dann als Ausgabe angezeigt und in eine Datei gespeichert.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
```

```
95  /**
96   * Laufzeit: O(V * E^2 + V^2)
97   *
98   * @param matrix
99   * @return
100  */
101 private int[][] fordFulkerson(int[][] matrix) {
102
103   // Anzahl der Knoten im Graph
104   int nodes = matrix[0].length;
105   // Die Quelle is der erste Knoten
106   int source = 0;
107   // Die Senke ist der letzte Knoten
108   int sink = nodes - 1;
      // Flow ist zu Beginn 0
      max_flow = 0;

      int u, v;

      // Erzeuge echte Kopie der Matrix für Output
      int output[][] = new int[nodes][nodes];
      for (u = 0; u < nodes; u++)
        for (v = 0; v < nodes; v++)
          output[u][v] = matrix[u][v];

      // Erzeuge ein Eltern Array zum speichern der möglichen BFS-Pfade
      int parent[] = new int[nodes];

      // Wenn für einen Pfad der BFS möglich ist, überprüfe seinen maximalen Fluss
      while (bfs(matrix, output, source, sink, parent)) {

        // Setze den Pfad Fluss auf unendlich
        int path_flow = Integer.MAX_VALUE;
        // Finde den maximalen Fluss durch die möglichen Pfade
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          path_flow = Math.min(path_flow, output[v][u]);
        }

        // aktualisiere die Kanten aus dem Eltern Array
        for (u = sink; u != source; u = parent[u]) {
          v = parent[u];
          // Ziehe den Fluss-Pfad den Kanten ab
          output[v][u] -= path_flow;
          // Addiere den Fluss-Pfad auf die Inversen Kanten
          output[u][v] += path_flow;
        }

        // Addiere die einzelnen Flusspfade auf den maximalen Fluss
        max_flow += path_flow;
      }

      // Ziehe von der Eingabe Matrix die übrigen Flussgewichte ab
      int[][] outputGraph = new int[nodes][nodes];
      for (int i = 0; i < matrix[0].length; i++)
        for (int j = 0; j < matrix[0].length; j++)
          outputGraph[i][j] = matrix[i][j] - output[i][j];

      return outputGraph;
    }

    /**
     * Laufzeit: O(V + E)
     *
     * @param matrix
     * @param output
     * @param s
     * @param t
     * @param parent
     * @return
     */
    private boolean bfs(int[][] matrix, int output[][], int s, int t, int parent[]) {

      // Anzahl der Knoten im Graph
      int nodes = matrix[0].length;

      // Array das alle Knoten als nicht besucht markiert
      boolean visited[] = new boolean[nodes];
      for (int i = 0; i < nodes; ++i)
        visited[i] = false;

      // Warteschlange, die besuchte Knoten als true markiert
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.add(s);
      visited[s] = true;
      parent[s] = -1;

      // Standard BFS-Loop, entfernt Knoten aus der Warteschlange die ungleich 0 sind
      while (queue.size() != 0) {
        int u = queue.poll();

        for (int v = 0; v < nodes; v++) {
          if (visited[v] == false && output[u][v] > 0) {
            // Wenn wir einen möglichen Pfad von s nach t finden geben wir true zurück
```

```
        if (v == t) {
          parent[v] = u;
          return true;
        }
        // Wenn wir keinen möglichen Pfad finden fügen wir den Knoten zur Warteschlange
        // und markieren ihn als besichtigt
        queue.add(v);
        parent[v] = u;
        visited[v] = true;
      }
    }
  }
  return false;
}
```