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....

1.1.1 Art der Dokumentation

Die Dokumentation des Projekts ist auf der Website schilda.node5.de zu finden. Alternativ wird automatisiert aus den Markdown-Dateien eine PDF-Dokumentation erstellt, die über diesen Link herunter geladen werden kann.

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 Rahmenbedingungen

1.3.1 Eingabe

Alle Algorithmen benötigen **einen Graph als Eingabe** und liefern **einen Graph als Ausgabe**. Die Testgraphen werden je nach Art des Graphen folgende Form haben:

1. ungerichteter und ungewichteter Graph
2. ungerichteter und gewichteter Graph
3. gerichteter und ungewichteter Graph
4. gerichteter und gewichteter Graph

Alle Graphen werden als einfache Textdateien gegeben. Die erste Zeile beginnt mit zwei Leerzeichen, gefolgt von den **Namen der Knoten**, jeweils getrennt mit einem Leerzeichen. Alle weiteren Zeilen starten mit einem **Knotennamen, gefolgt von einer Folge von Zahlen**, jeweils **getrennt durch ein Leerzeichen**. Für ungerichtete Graphen werden jeweils nur die Werte unterhalb der Diagonalen angegeben.

Wichtig: Um neue Daten für die Eingabe zu verwenden, muss der Inhalt der txt-Dateien im Ordner "code/data/problemX.txt" ersetzt werden.

1.3.2 Ausgabe

Die Ausgabe erfolgt in derselben Form wie die Eingabe.

1.3.3 Abgabe Projekt

Das Projekt ist als PDF-Datei mit folgender Benennung "PT3-2022-FelixMöhler-JulianThiele.pdf" bis zum 20.01.2023 abzugeben.

1.4 Planungstool Aufbau

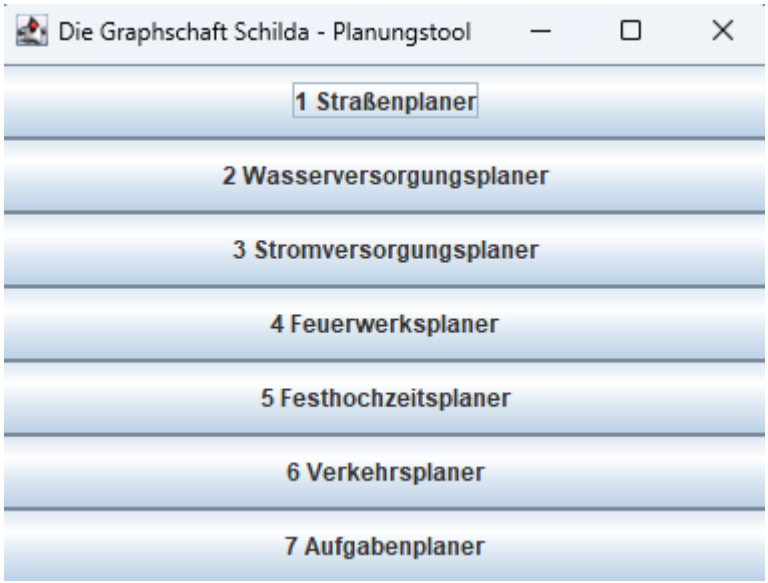
In den folgenden Abschnitten wird der Aufbau des Planungstools beschrieben.

1.4.1 Ordnerstruktur des Projektes

```
GraphschafSchilda/
├─ code/
│  ├─ data/
│  │  └─ Eingabe txt-Dateien
│  ├─ output/
│  │  └─ Ausgabe txt-Dateien
│  ├─ old/
│  │  └─ Frühere Versionen der Algorithmen
│  ├─ utils/
│  │  ├─ AdjazentMatrix.java
│  │  ├─ BasicWindow.java
│  │  ├─ Edge.java
│  │  ├─ FileHandler.java
│  │  ├─ JGraphPanel.java
│  │  └─ Vertex.java
│  ├─ Main.java
│  ├─ Problem1.java
│  ├─ Problem2.java
│  ├─ Problem3.java
│  ├─ Problem4.java
│  ├─ Problem5.java
│  ├─ Problem6.java
│  └─ Problem7.java
├─ documentation/
│  └─ docs/
└─ libs/
```

1.4.2 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.



1.4.3 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.

1.5 Das Team

Felix Möhler - GitHub - Aufgabenteilung: Dokumentation und Code für Problem 2, 5, 6 und 7

Julian Thiele - GitHub - Aufgabenteilung: Dokumentation und Code für Problem 1, 3, 4, 6 und 7

1.6 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.

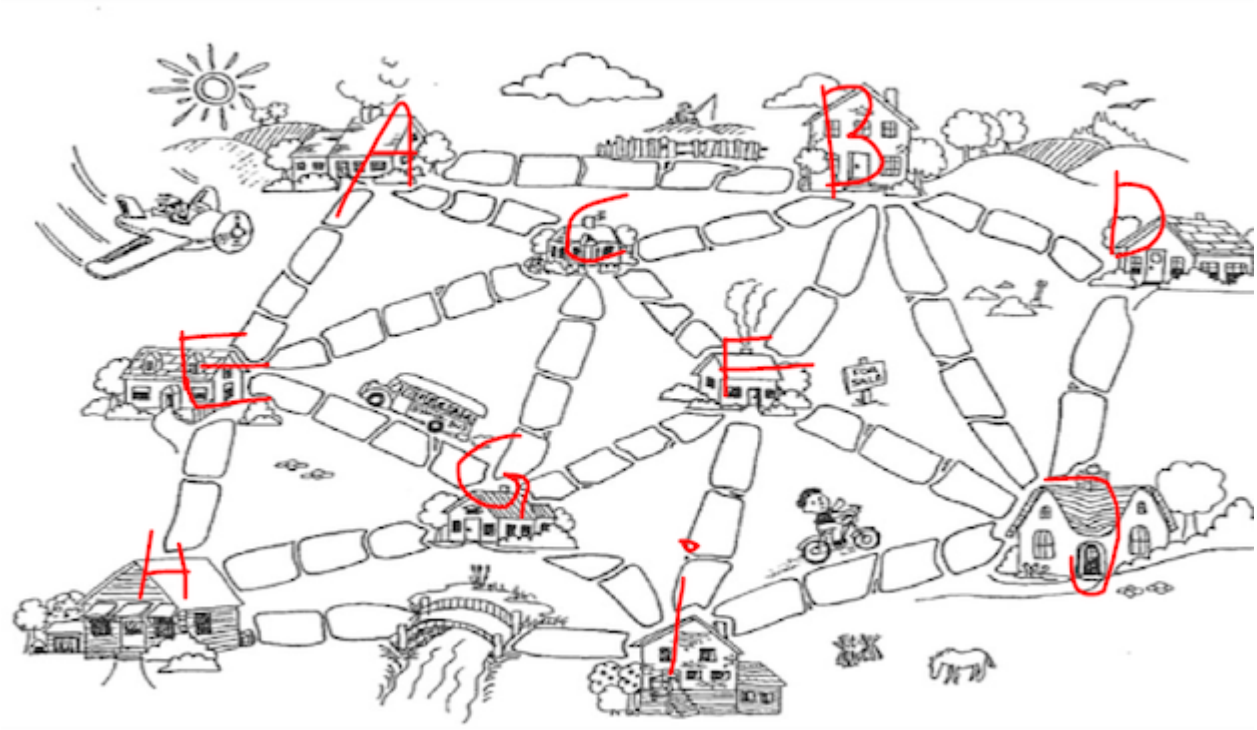
Es wird eine Konfiguration an Kanten gesucht, die eine minimale Anzahl an Pflastersteinen benötigt.

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.



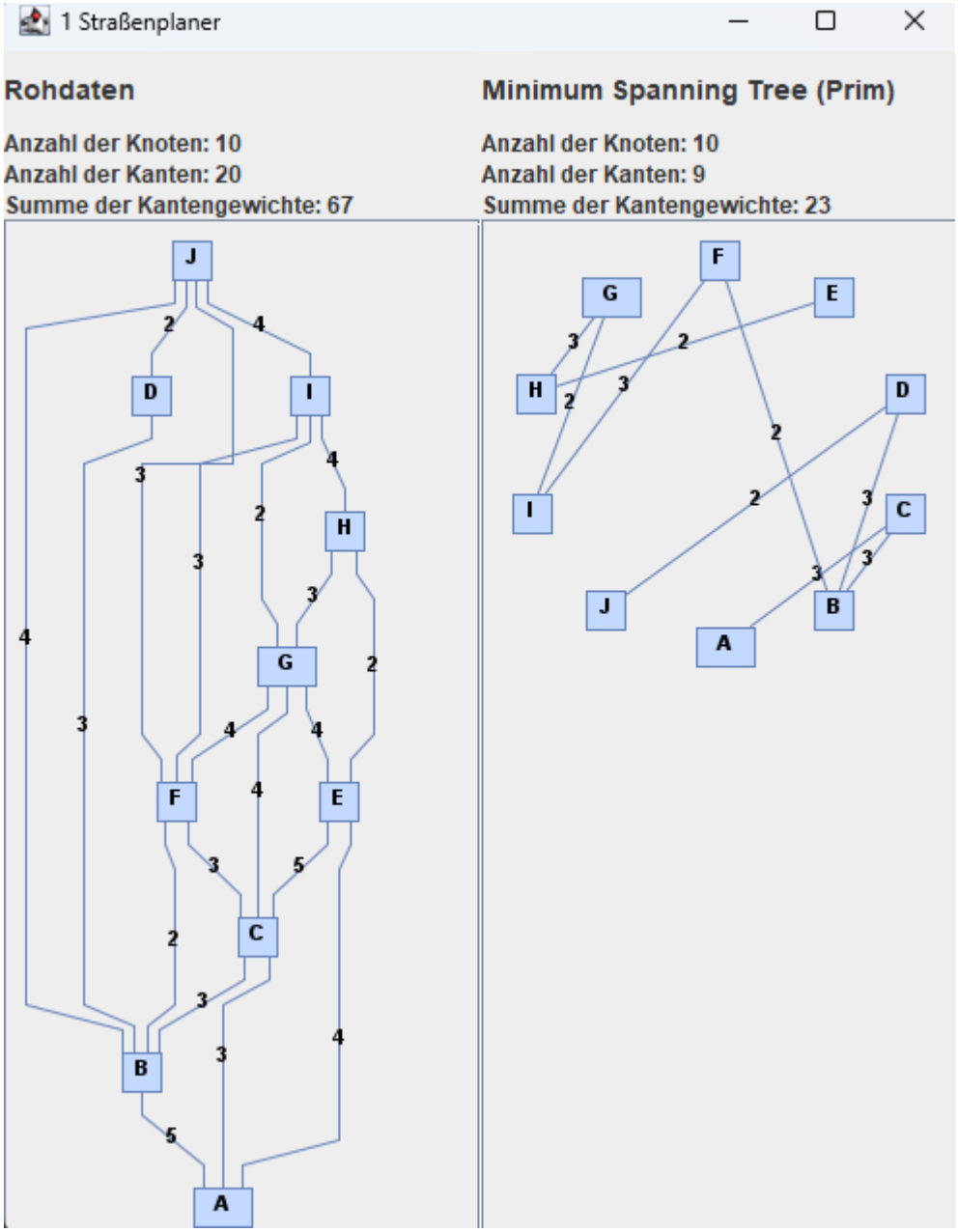
```
// code/data/problem1.txt
  A B C D E F G H I J
A 0
B 5 0
C 3 3 0
D 0 3 0 0
E 4 0 5 0 0
F 0 2 3 0 0 0
G 0 0 4 0 4 4 0
H 0 0 0 0 2 0 3 0
I 0 0 0 0 0 3 2 4 0
J 0 4 0 2 0 3 0 0 4 0
```

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.

Eine 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.




```
1  // code/output/1 Straßenplaner.txt
2    A B C D E F G H I J
3  A 0
4  B 0 0
5  C 3 3 0
6  D 0 3 0 0
7  E 0 0 0 0 0
8  F 0 2 0 0 0 0
9  G 0 0 0 0 0 0 0
10 H 0 0 0 0 2 0 3 0
11 I 0 0 0 0 0 3 2 0 0
12 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.

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

Das Problem lässt sich als Graphenmodell mit gerichteten und gewichteten Kanten darstellen. Jedes Haus ist ein Knoten, die Wasserleitungen zwischen den Häusern sind die Kanten. Das Gewicht der Kanten wird durch den maximal möglichen Volumenstrom in m^3/s dargestellt.

Es wird nach dem maximalen Fluss im gegebenen Flussnetzwerk von einer Quelle zu einer Senke gesucht.

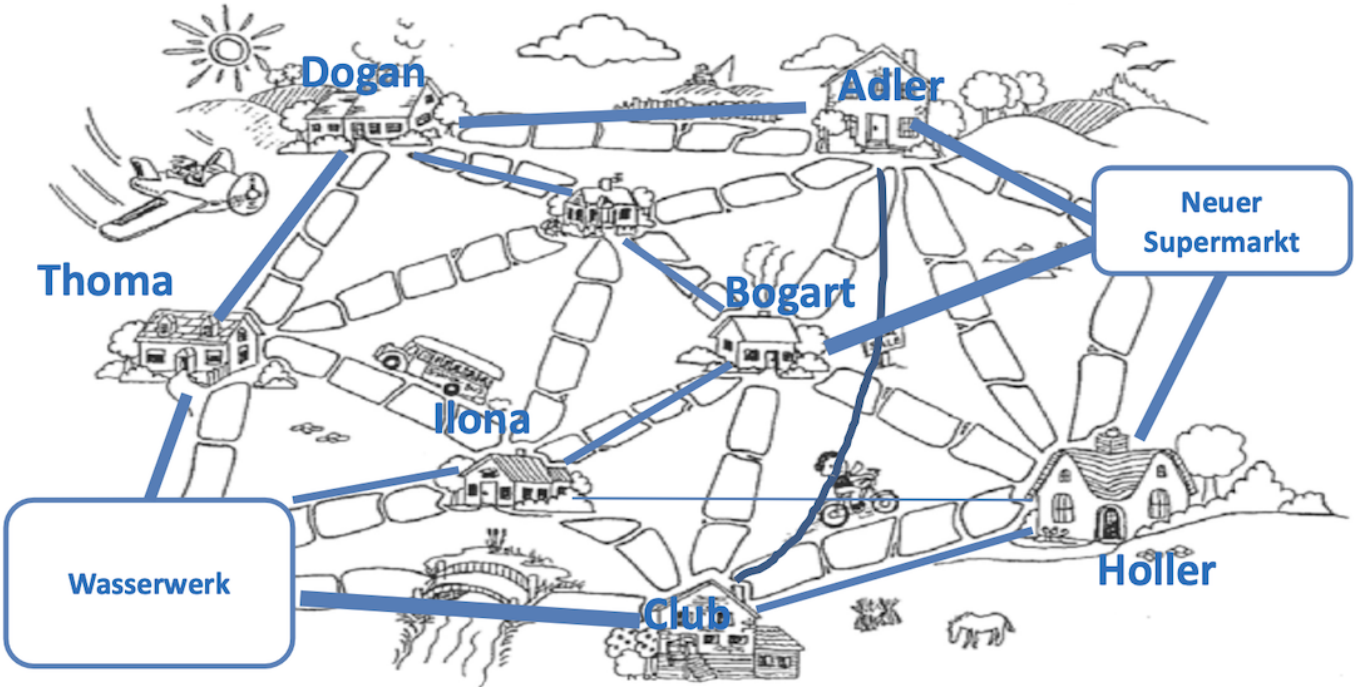
Durch die Kombination der Kanten von einer Quelle (Startpunkt) zu einer Senke (Endpunkt) erhält man mögliche Flusspfade. Wenn ein Pfad gefunden wurde, wird das maximal mögliche Kantengewicht (Gewicht der kleinsten Kante) von allen Kanten aus dem Pfad abgezogen. Dieser Vorgang wird so lange wiederholt, bis es keinen möglichen Flusspfad mit freier Kapazität von der Quelle zur Senke mehr gibt. Die einzelnen möglichen Flusspfade werden zu einem Gesamtdurchfluss addiert. Durch Verwenden des Edmonds-Karp-Algorithmus wird stehts der kürzeste Weg mit freien Kapazitäten gewählt und somit auch das schnellstmögliche Abarbeiten des Graphen gewährleistet.

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.

3.2 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 maximalen Flusses von Quelle zu Senke.

Für die Matrix in der Datei `problem2.txt` wurden jeweils die Anfangsbuchstaben W, A, B, C, D, H, I, T, S, der Gebäude aus dem Bild als Bezeichnung genutzt. Wobei W für das Wasserwerk als Quelle steht und S für den Supermarkt als Senke.



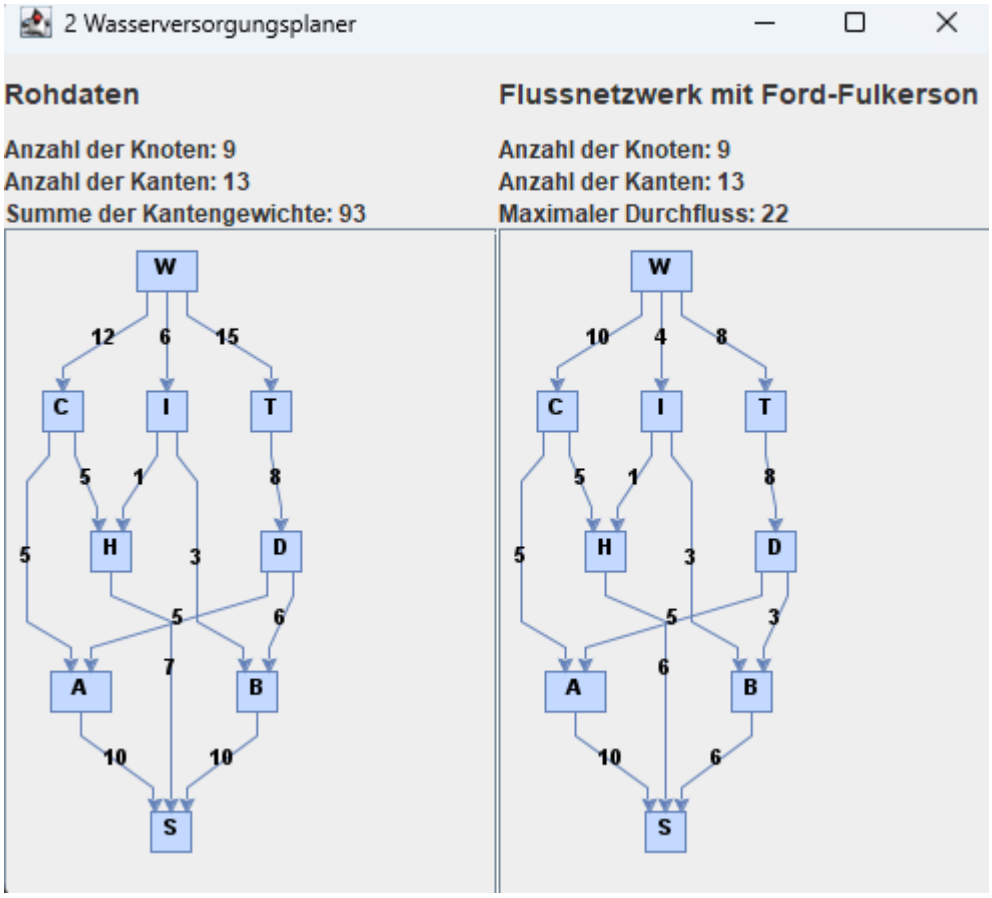

```
// code/data/problem2.txt
  W A B C D H I T S
W 0 0 0 12 0 0 6 15 0
A 0 0 0 0 0 0 0 0 10
B 0 0 0 0 0 0 0 0 10
C 0 5 0 0 0 5 0 0 0
D 0 5 6 0 0 0 0 0 0
H 0 0 0 0 0 0 0 0 7
I 0 0 3 0 0 1 0 0 0
T 0 0 0 0 8 0 0 0 0
S 0 0 0 0 0 0 0 0 0
```

3.3 Die Ausgabe

Die Ausgabe wird als Graph in einem Fenster dargestellt und in die Datei `2 Wasserversorgungsplaner.txt` geschrieben. Die erzeugte Datei wird im Projekt im Ordner `output` abgelegt und kann mit einem Text Editor geöffnet werden. Das Fenster zur Visualisierung der Graphen besteht aus zwei Hälften. Auf der linken Seite wird der Eingabegraph dargestellt. Auf der rechten Seite wird der berechnete Ausgabegraph dargestellt. In diesem Graph entsprechen die Werte der Kanten dem maximal nutzbaren Volumenstrom pro Kante von Wasserwerk zu Supermarkt.

Eine korrekte Ausgabe erfüllt folgende Eigenschaften:

- Der Fluss von Quelle zu Senke muss maximal sein
- Alle nutzbaren Pfade für Flüsse müssen gefunden werden
- Der maximal mögliche Fluss pro Pfad muss von den Pfadkanten abgezogen werden
- Die Kanten müssen gerichtet sein und der Rückfluss aus dem Ausgabegraph entfernt
- Es muss ersichtlich sein welcher Volumenstrom an der Quelle möglich ist




```
// code/data/2 Wasserversorgungsplaner.txt
  W A B C D H I T S
W 0 0 0 10 0 0 4 8 0
A 0 0 0 0 0 0 0 0 10
B 0 0 0 0 0 0 0 0 6
C 0 5 0 0 0 5 0 0 0
D 0 5 3 0 0 0 0 0 0
H 0 0 0 0 0 0 0 0 6
I 0 0 3 0 0 1 0 0 0
T 0 0 0 0 8 0 0 0 0
S 0 0 0 0 0 0 0 0 0
```

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.