Dijkstra Spezifikation
Projektkette Naturwissenschaften

Adrian Boldi Igor Wiedler

13. Januar 2010

Inhaltsverzeichnis

1 Algorithmus **3**
1.1 Problem . . . 3
1.2 Definition . . . 3
1.3 Beispiel . . . 3

2 Design **4**
2.1 Programmablauf . . . 4
2.2 Datenmodell . . . 5
2.3 Anwendungsfälle . . . 5

3 Dateiformat **7**
3.1 Allgemein . . . 7
3.2 Typen . . . 7
3.2.1 Vertex . . . 7
3.2.2 Edge . . . 7
3.3 Beispiel . . . 7
3.4 Dateiendung . . . 7

4 Pseudocode **8**

5 Programmdokumentation **9**
5.1 Generell . . . 9
5.2 Klassen . . . 9
5.2.1 Reader . . . 9
5.2.2 Writer . . . 9
5.2.3 Dijkstra . . . 10
5.2.4 Graph . . . 10
5.2.5 Vertex . . . 10
5.2.6 Edge . . . 10
5.2.7 GraphGUI . . . 10
5.2.8 Point . . . 11
5.2.9 EdgeEditWindow . . . 11
5.2.10 VertexFactory . . . 11
5.2.11 VertexEditWindow . . . 11
5.2.12 ItemSelectWindow . . . 11
5.3 Anwendung . . . 11
5.3.1 Graphen . . . 12

5.3.2 Vertices . . . 12
5.3.3 Edges . . . 12
5.3.4 Algorithmen . . . 12
5.4 Beispiele . . . 13

6 Verwendete Mittel **15**
6.1 Programme . . . 15
6.2 Quellen . . . 15
6.2.1 WWW . . . 15
6.2.2 Literatur . . . 16

1 Algorithmus

1.1 Problem

Das Ziel des Dijkstra Algorithmus ist es, den kürzesten Weg zwischen zwei Punkten innerhalb eines gewichteten Graphes zu finden.

Ein mögliches Szenario hierfür ist eine Reise zwischen zwei Städten A und B. Zwischen diesen Städten befinden sich weitere Städte, welche durch Hauptstrassen verbunden sind. Das befahren dieser Hauptstrassen dauert unterschiedlich lange, was jedoch nicht nur von der eigentlichen Distanz abhängen muss. Das Ziel ist es den schnellsten weg zwischen A und B zu finden.

1.2 Definition

1. Dem Startknoten wird der Wert 0 zugewiesen. Dieser Knoten wird eingerahmt.
2. All jene Knoten, die mit dem zuletzt eingerahmten Knoten verbunden sind, werden mit dem Wert des gerahmten Knoten plus dem Wert der verbindenden Kante temporär beschriftet. Ist der Knoten bereits beschriftet, wird er nur dann überschrieben, wenn der neue Wert tiefer wäre als der bestehende. Bereits eingerahmte Knoten können hierbei ignoriert werden.
3. Von allen temporär beschrifteten, noch nicht eingerahmten Knoten wird derjenige eingerahmt, welcher den tiefsten Wert besitzt. Falls dieser Knoten der Zielknoten ist, gehe zu 5.
4. Gehe zurück zu 2.
5. Der schnellste Weg wurde gefunden.

1.3 Beispiel

Hier soll der Algorithmus an einem einfachen Graphen gezeigt werden. Der Startknoten ist a, der Endknoten lautet f.

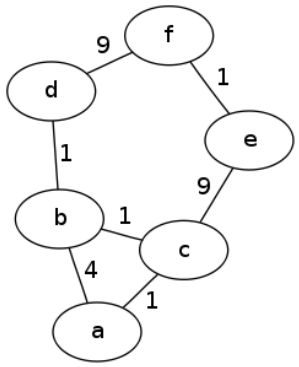


Abbildung 1: Einfacher gewichteter Graph

Knoten den Wert 0 zuweisen und einrahmen.
Verbundene Knoten beschriften: *b = 4*, *c = 1*.

Tiefsten temporär beschrifteten Knoten einrahmen: *c*.
Verbundene Knoten beschriften: *b = 2*, *e = 10*.

Tiefsten temporär beschrifteten Knoten einrahmen: *b*.
Verbundene Knoten beschriften: *d = 3*.

Tiefsten temporär beschrifteten Knoten einrahmen: *d*.
Verbundene Knoten beschriften: *f = 12*.

Tiefsten temporär beschrifteten Knoten einrahmen: *e*.
Verbundene Knoten beschriften: *f = 11*.

Tiefsten temporär beschrifteten Knoten einrahmen: *f*.

Der schnellste Weg lautet: *f*, *e*, *c*, *a*.

Wie man leicht erkennen kann, wird der schnellste Weg gefunden, indem der Pfad welcher zum Einrahmen der jeweiligen Knoten geführt hat, rückwärts durchquert wird. Bei bedarf lässt sich dieser spiegeln, damit er beim Anfangsknoten beginnt und beim Zielknoten endet.

2 Design

2.1 Programmablauf

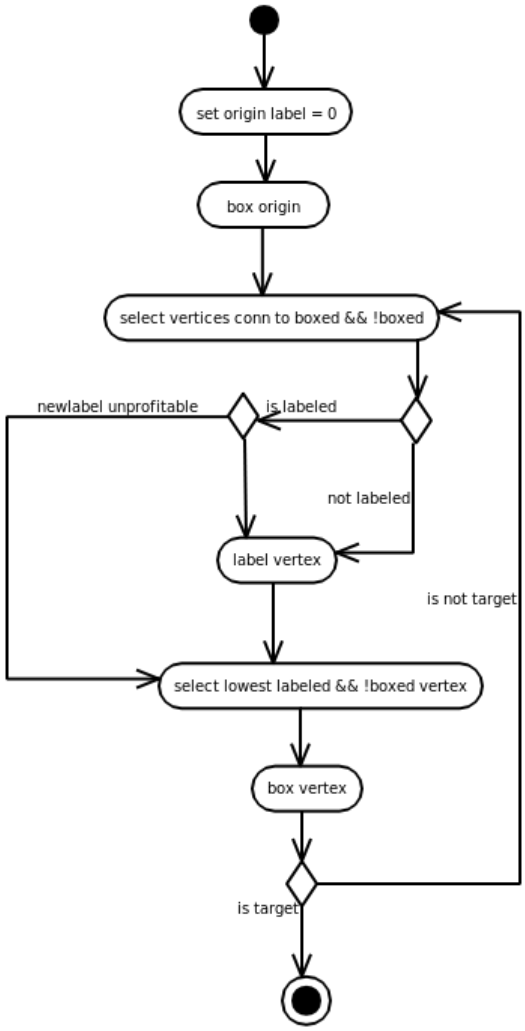


Abbildung 2: UML Aktivitätsdiagramm des Dijkstra

Dieses Diagramm beschreibt den Ablauf des Dijkstra Algorithmus. Es entspricht der obrigen Definition.

Begriffe

vertex Knoten

label temporäre Beschriftung eines Knotens

origin Ursprung; der Ursprungsknoten des Algorithmus

target Ziel; der Zielknoten des Algorithmus

boxed permanent eingerahmt

unprofitable Vertex ist bereits temporär beschriftet, neuer Wert wäre jedoch höher oder gleich dem bestehenden

lowest der/die/das tiefste

2.2 Datenmodell

Es folgt ein Diagramm, welche die verwendeten Klassen, sowie deren Attribute und Beziehungen untereinander beschreibt. Diese Klassen definieren, welche Daten verwendet werden.

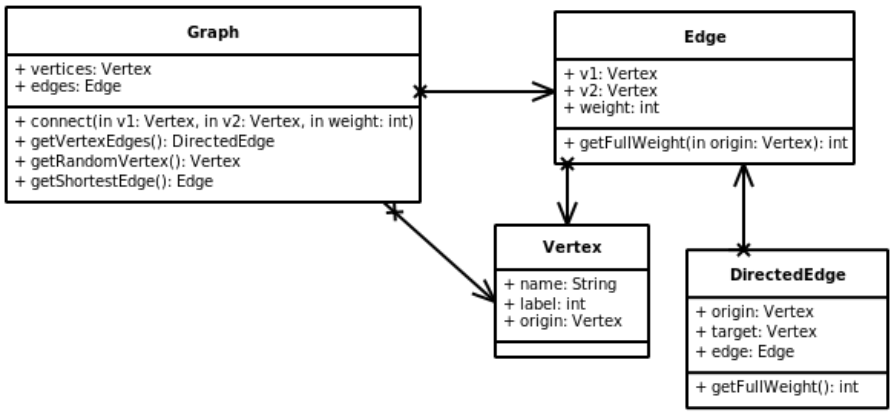


Abbildung 3: UML Klassendiagramm des Modells

Beschreibung

Graph ein Graph beinhaltet Knoten und Kanten

Vertex ein Knoten

Edge eine gewichtete Kante zwischen zwei Knoten

DirectedEdge eine Kante, bei der die Richtung bekannt ist

2.3 Anwendungsfälle

Anwendungsfälle beschreiben die Funktionalitäten, die das Programm dem Benutzer bietet.

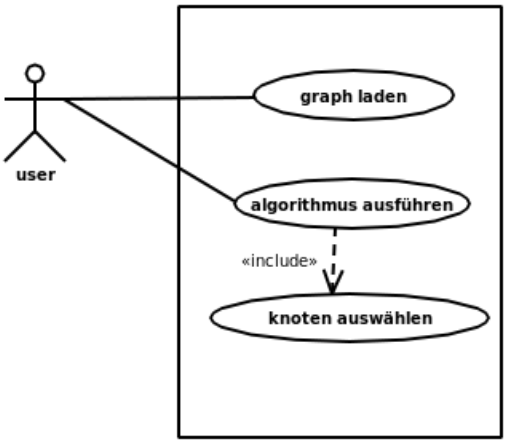


Abbildung 4: UML Anwendungsfall Diagramm

Beschreibung

User der Benutzer des Programms

Graph laden Graphen können aus einer Datei geladen werden

Algorithmus ausführen dies ist die Hauptfunktion

Knoten auswählen bei manchen Algorithmen (zum Beispiel Dijkstra) wird ein Start- und/oder Zielknoten benötigt, dieser soll ausgewählt werden können

3 Dateiformat

3.1 Allgemein

Damit es möglich ist Graphen in das Programm zu laden, braucht es ein Dateiformat. Der Einfachheit halber wird hier ein ASCII basiertes Format verwendet. Eine Zeile in der Datei kann hierbei entweder einen Knoten oder eine Kante beschreiben.

Die Zeilen werden mittels Linefeed (*ASCII 10*) getrennt. Pro Zeile gibt es Tokens, die via Tab (*ASCII 9*) getrennt werden. Dies gleicht einem CSV-Format mit Tab Delimiter.

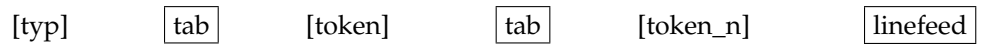


Abbildung 5: Grundgerüst des Dateiformats

3.2 Typen

Ein *v* als Typ steht für einen Knoten (*Vertex*), ein *e* für eine Kante (*Edge*). Je nach Typ gibt es verschiedene Tokens.

3.2.1 Vertex

Name Der Name des Knotens

Position x Die x-Koordinate

Position y Die y-Koordinate

3.2.2 Edge

Von Der Name des Ursprungsknotens

Nach Der Name des Zielknotens

Gewichtung Die Gewichtung der Kante

3.3 Beispiel

v	a	5	5
v	b	5	10
v	c	10	5
e	a	b	3
e	b	c	4

Abbildung 6: Beispiel eines einfachen Grafs

3.4 Dateiendung

Der einfachheit halber wird keine Dateiendung benötigt. Aus Kompatibilitätsgründen (Windows) ist es jedoch möglich eine *txt* Endung zu verwenden.

4 Pseudocode

```
// eingabe entgegennehmen
input: vertices, origin, target

// initialisation
boxed = []

// origin vorbereiten
origin.label = 0
boxedVertex = origin

// endlosschlaufe
while true
        boxed += boxedVertex

        if boxedVertex == target
                // ziel erreicht
                break

        foreach getVertexEdges(boxedVertex) as edge
                // ueberspringen falls boxed
                // ueberspringen falls unprofitabel (nicht ueberschreibbar)
                if !boxed.contains(edge.target) ||
                        (edge.target.label && edge.fullweight >= edge.target.label)

                        edge.target.label = edge.fullweight
                        edge.target.origin = boxedVertex

                        // vertices nach label aufsteigend sortieren
                        // tiefsten selektieren
                        vertices.sort()
                        foreach vertices as vertex
                                if !boxed.contains(vertex) && vertex.label
                                        boxedVertex = vertex
                                        break

// auswertung
result = []

vertex = target
result += vertex

while vertex != origin
        vertex = vertex.origin
        result += vertex

output: result
```

5 Programmdokumentation

5.1 Generell

Die Applikation *UltraGraph* wurde in Java realisiert. Java enthält erweiterte Objektorientierte Features sowie eine einheitliche grafische Oberfläche und gute Unterstützung für die gängigsten Betriebssysteme.

Abstraktion wurde eingesetzt, um die Komponenten so lose wie möglich zu koppeln. Beispielsweise ist das GUI nicht von einem bestimmten Algorithmus abhängig, es kann mehrere verwenden. Ausserdem wurden Teile des Model-View-Controller Patterns eingesetzt, welche die Logik von Daten und Darstellung trennen.

Da der Quellcode weitgehend mit Javadoc kommentiert wurde, ist eine Javadoc-Dokumentation vorhanden.

5.2 Klassen

5.2.1 Reader

Die Klasse Reader ist dazu da, um die gespeicherten Graphen aus einer Text Datei auszulesen, und den Graph zu erzeugen.

Ablaufbeschreibung der Funktion *getGraph*:

1. Aufruf der Funktion mit dem Namen des Files als Parameter.
2. Einlesen der ersten Linie.
3. Die Linien werden nun gesplitet
4. Der erste Character der Linie wird überprüft
   - Wenn *v*, dann einen neuen Vertex mit den drei folgenden Characters
     - Vertex wird zu Graph *g* hinzugefügt.
   - Wenn *e*, dann eine neue Edge mit den drei folgenden Characters
     - Im Graph *g* werden zwei Vertex mit der Edge verbunden.
5. Gibt den Graph *g* zurück

5.2.2 Writer

Die Klasse Writer ist dazu da, um den Graph in eine Text Datei zu speichern.

Ablaufbeschreibung:

1. Aufruf der Funktion mit den Parametern: Graph *g* und des gewünschten Filename
2. Das File wird erstellt
3. Solange Vertices im Graph *g* vorhanden sind
   - Schreibe Identifikation, Name, X-Koordinate und Y-Koordinate in das File
4. Solange Edges im Graph *g* vorhanden sind
   - Schreibe Identifikation, Start Vertex, End Vertex und Gewichtung in das File
5. File wird geschlossen

5.2.3 Dijkstra

In der Klasse Dijekstra ist der eigentliche Algorithmus implementiert. Eine genauere Beschreibung des Algorithmus finden sie in Punkt 2.1.

5.2.4 Graph

- Methode add(Vertex... vertices)
  - fügt alle Vertices in den Vektror Vertices hinzu.
- Methode connect(Vertex v1, Vertex v2, int weight)
  - erzeugt Edges und fügt diesen dem Vektor Edge hinzu.
- Methode connect(Vertex v1, Vertex v2)
  - erzeugt Edges mit einem Defaultwert für weight und fügt diesen dem Vektor Edge hinzu.
- Methode getVertices()
  - gibt alle Vertices zurück
- Methode getEdges()
  - gibt alle Edges zurück
- Methode removeVertex(Vertex v)
  - löscht den Vertex v mit allen angeschlossenen Edges
- Methode removeEdge(Edge e)
  - löscht die Edge e
- Methode removeVertices(Vector<Vertex> vv)
  - löscht alle Vertices vv
- Methode getVertexEdges(Vertex v)
  - gibt alle mit v verbundenen Edges zurück
- Methode getVerticesEdges(Vertex v1, Vertex v2)
  - gibt alle mit v1 und v2 verbundenen Vertices zurück

5.2.5 Vertex

Die Klasse Vertex ist das Model für die Vertices.

5.2.6 Edge

Die Klasse Vertex ist das Model für die Edges.

5.2.7 GraphGUI

Die Klasse GraphGUI enthält das graphische Oberfläche von UltraGraph.

5.2.8 Point

Representiert die Koordinaten der Vertices.

5.2.9 EdgeEditWindow

Das Fenster für die Editierung eines Edges.

5.2.10 VertexFactory

Erstellt Vertex mit einem Namen.

5.2.11 VertexEditWindow

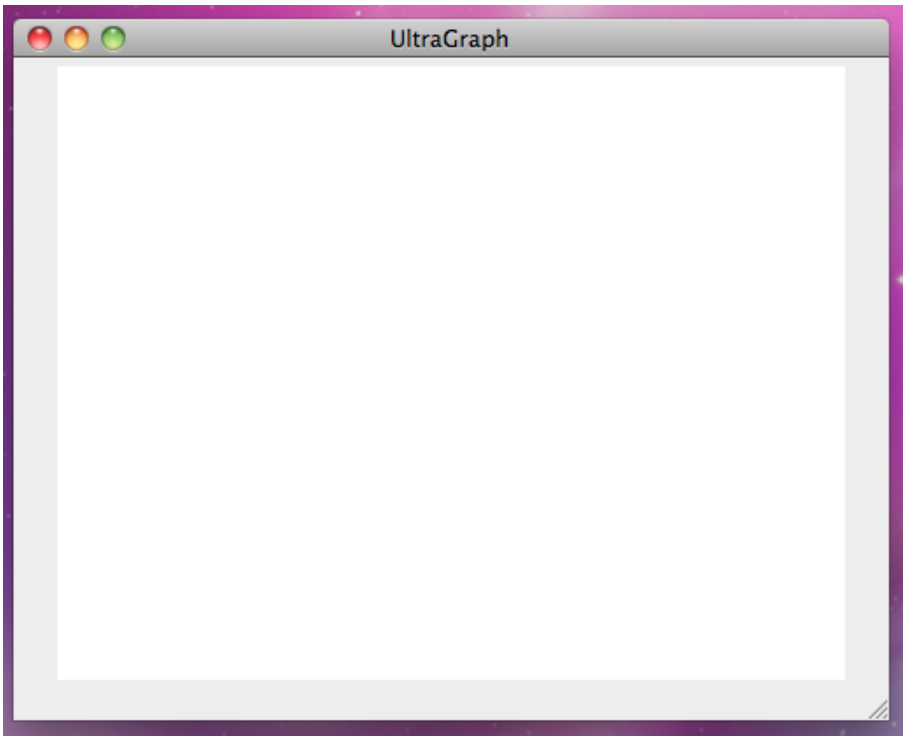
Das Fenster für die Editierung eines Vertex mit X- und Y-Position.

5.2.12 ItemSelectWindow

Auswahlfenster zum auswählen des gewünschten Edges oder Vertices zum editieren.

5.3 Anwendung

Wenn Sie UltraGraph starten erscheint das Hauptfenster. Die leere Fläche bietet Platz für Ihren Graphen.



5.3.1 Graphen

Nach dem Programmstart erscheint ein neuer leerer Graph. Falls Sie den momentanen Graph verwerfen wollen, drücken Sie *Command + N*, um einen neuen zu erzeugen. Über *Graph → New* besteht eine alternative Möglichkeit hierfür.

Falls Sie einen gespeicherten Graphen laden wollen, drücken Sie *Command + O*, und wählen Sie die Datei aus, welche den Graphen enthält. Über *Graph → Open* besteht eine alternative Möglichkeit hierfür.

Um einen Graphen zu speichern, drücken Sie *Command + S*. Ein Graph kann in verschiedenen Formaten gespeichert werden: Das UltraGraph-Format, Graphviz-dot und XStream-XML. Wählen Sie den Ort und den Dateinamen für die Datei. Über *Graph → Save* besteht eine alternative Möglichkeit hierfür.

5.3.2 Vertices

Um einen Vertex hinzuzufügen, muss mit gedrückter Alt-Taste auf die Position geklickt werden, an welcher der neue Vertex erscheinen soll. Über *Vertex → Add* besteht eine alternative Möglichkeit hierfür.

Vertices können verschoben werden, indem der Vertex mit gedrückter Maustaste an die gewünschte stelle gezogen wird. Über *Vertex → Edit* besteht eine alternative Möglichkeit hierfür, bei der es auch möglich ist Vertices umzubenennen.

Um einen Vertex zu löschen, muss mit gedrückten Shift- und Alt-Tasten auf den Vertex geklickt werden, welcher gelöscht werden soll. Über *Vertex → Remove* besteht eine alternative Möglichkeit hierfür.

5.3.3 Edges

Um eine Edge hinzuzufügen, müssen die beiden zu verbindenden Vertices mit gedrückter Shift-Taste verbunden werden. Die neue Edge erscheint dann. Es ist auch möglich mehrere Edges zwischen zwei Vertices zu haben. Über *Edge → Add* besteht eine alternative Möglichkeit hierfür.

Die Gewichtung von Edges kann geändert werden, indem im Menü *Vertex → Edit* ausgewählt wird.

Um eine Edge zu löschen, selektiert man im Menü *Vertex → Remove*.

5.3.4 Algorithmen

UltraGraph beherscht sowohl den Dijkstra-Algorithmus, wie auch Kruskal und Prim. Um den zu verwendenden Algorithmus auszuwählen muss man *Command + A* oder im Menü *Algorithm → Select algorithm* wählen.

Um den ausgewählten Algorithmus zu starten, wird *Command + R* gedrückt, oder im Menü *Algorithm → Start* ausgewählt. Im Falle das Dijkstra, muss noch konfiguriert werden, welches die Start- und Endknoten sein sollen, bei den anderen beiden ist keine Konfiguration nötig. Sobald diese vorgenommen wurde, wird der Algorithmus gestartet. Die Anzeige erfolgt direkt im Hauptfenster.

Es ist möglich, den laufenden Algorithmus via *Algorithm → Pause* zu pausieren.

Falls das Programm auf der Kommandozeile ausgeführt wird, werden dort zusätzliche Informationen angezeigt.

5.4 Beispiele

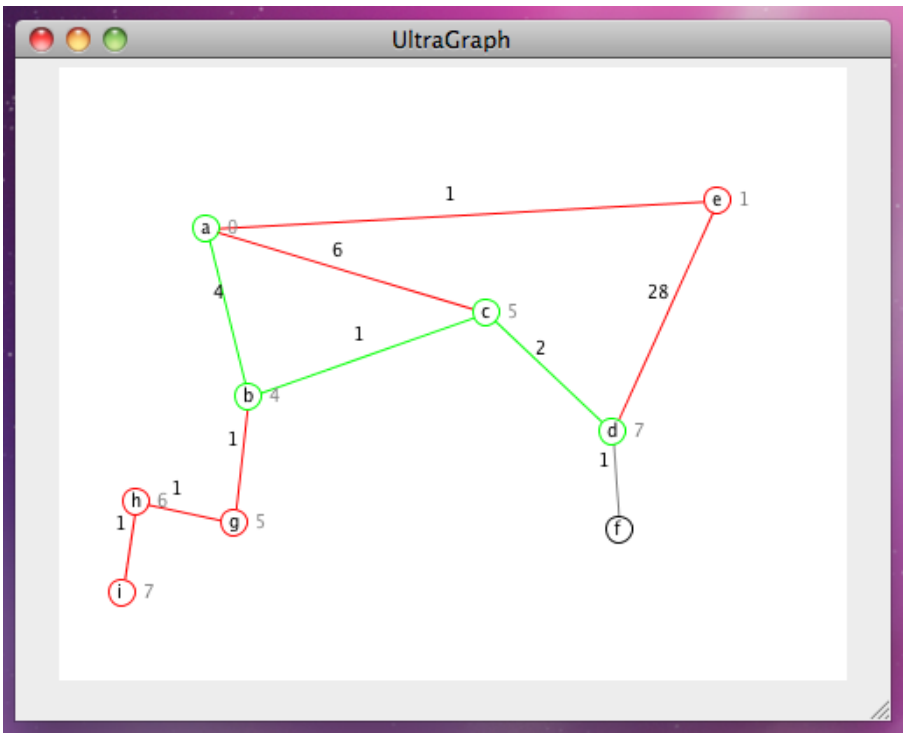


Abbildung 7: Beispielsgraph nach Dijkstra Durchlauf

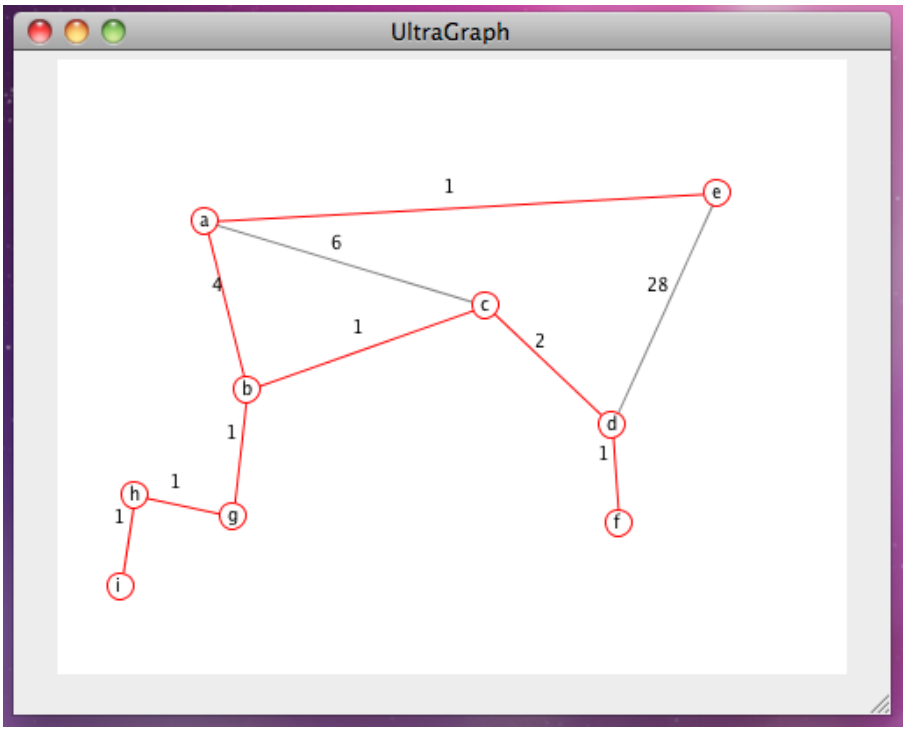


Abbildung 8: Beispielsgraph nach Kruskal Durchlauf

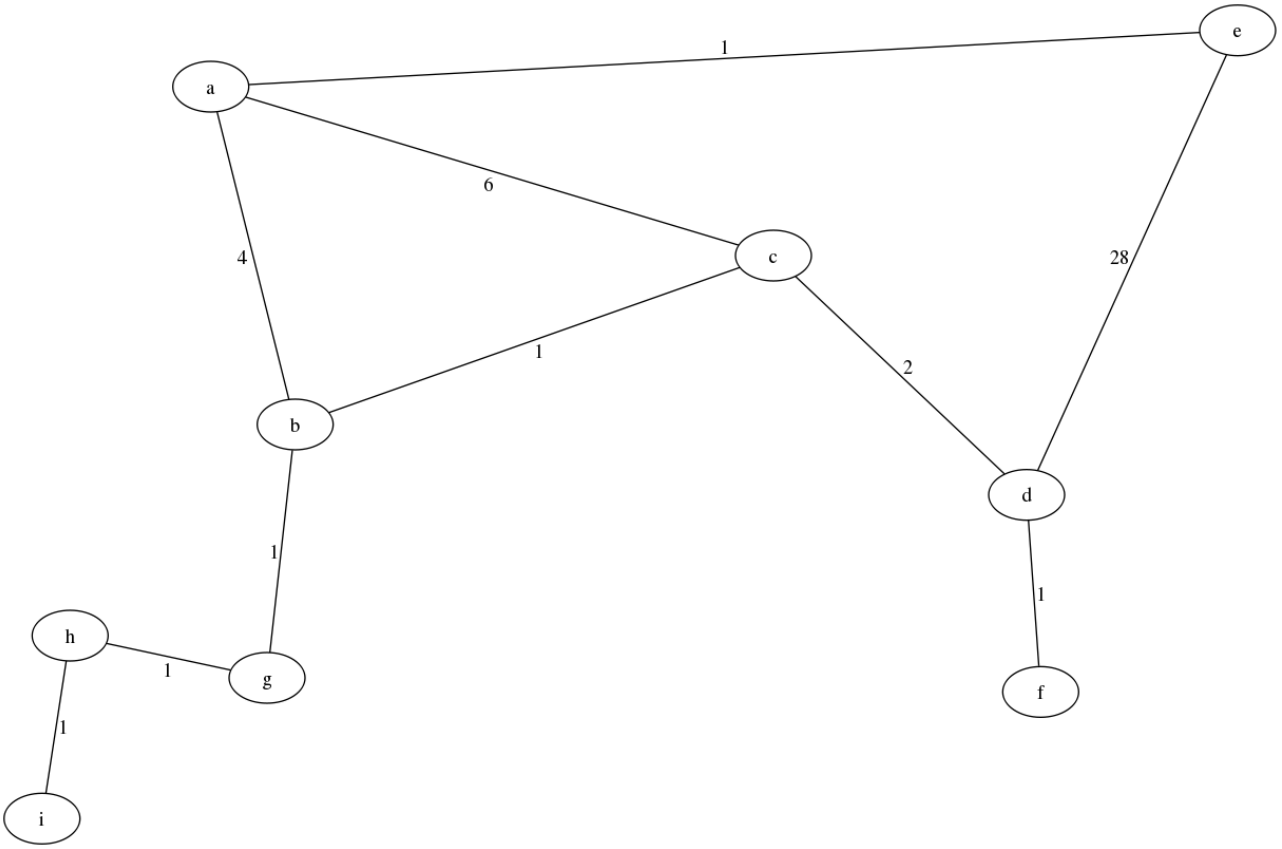


Abbildung 9: Beispielsgraph mit Graphviz visualisiert

6 Verwendete Mittel

6.1 Programme

- *Java:* Programmiersprache und Umgebung
  http://www.eclipse.org
- *Eclipse:* Entwicklungsumgebung für Java
  http://www.eclipse.org
- *Gaphor* UML Diagramm Tool
  http://gaphor.sourceforge.net
- *Graphviz* Graphenvisualisierung
  http://graphviz.org
- LaTeX: Textsatz-Software
  http://www.latex-project.org
- *XStream*: XML-Enkodierungs-Bibliothek für Java
  http://xstream.codehaus.org
  http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/index.html
- *Git*: Version Control System
  http://git-scm.com

6.2 Quellen

6.2.1 WWW

- vkaul: *Dijkstra's Alhorithm*, 23. Januar 2004, 12. Januar, 2010
  http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-082JNetwork-OptimizationSpring2003/FC13EFA1-0FE2-4BFB-B019-8939606EDDCC/0/dijkstrasalgorithm.pdf
- Unbekannter Author: *Der Algorithmus von Dijkstra*, 31. März 1998, 12. Januar, 2010
  http://www.educ.ethz.ch/lehrpersonen/informatik/unterrichtsmaterialien_inf/kommuniation_kryptographie/routing/la3.pdf
- Wikipedia: *Algorithmus von Kruskal*, 18. August 2009, 12. Januar 2010
  http://de.wikipedia.org/wiki/Algorithmus_von_Kruskal
- Jean-Philippe Rameau: *Algorithmus von Kruskal Info*, unbekannte Datierung, 12. Dezember 2009
  http://infofrosch.info/a/al/algorithmus_von_kruskal.html
- Papagelis Athanasios: *Kruskal Algorithm*, 13. Oktober 1997, 12. Januar 2010
  http://students.ceid.upatras.gr/~papagel/project/kruskal.htm
- Papagelis Athanasios: *Pseudocode For Kruskal Algorithm*, 13. Oktober 1997, 12. Januar 2010
  http://students.ceid.upatras.gr/~papagel/project/pseukrus.htm
- Wikipedia: *Algorithmus von Prim*, 13. September 2008, 12. Januar 2010
  http://de.wikipedia.org/wiki/Algorithmus_von_Prim
- Kenji Ikeda, William F. Klostermeyer: *Prim's Algorithm Demo*, 6. Juli 2001, 12. Januar 2010
  http://www.unf.edu/~wkloster/foundations/PrimApplet/PrimApplet.htm
- Jean-Michel Léon: *Implementing Double Buffering with Java*, 27. September 2006, 12. Januar 2010
  http://www.ecst.csuchico.edu/~amk/classes/csciOOP/double-buffering.html

- Unbekannter Author: *The DOT Language*, unbekannte Datierung, 12. Januar 2010
  http://www.graphviz.org/doc/info/lang.html
- Sun Microsystems: *Java^{TM} Platform, Standard Edition 6 API Specification*, 15. August 2008, 12. Januar 2010
  http://java.sun.com/javase/6/docs/api/

6.2.2 Literatur

- *Networks – Student text and unit guide*, Cambridge University Press, 2000