

# Dijkstra Spezifikation

## Projektkette Naturwissenschaften

Adrian Boldi

Igor Wiedler

24. Oktober 2009

# Inhaltsverzeichnis

<b>1</b>	<b>Algorithmus</b>	<b>2</b>
1.1	Problem . . . . .	2
1.2	Definition . . . . .	2
1.3	Beispiel . . . . .	2
<b>2</b>	<b>Design</b>	<b>3</b>
2.1	Programmablauf . . . . .	3
2.2	Datenmodell . . . . .	4
2.3	Anwendungsfälle . . . . .	4
<b>3</b>	<b>Dateiformat</b>	<b>6</b>
3.1	Allgemein . . . . .	6
3.2	Typen . . . . .	6
3.2.1	Vertex . . . . .	6
3.2.2	Edge . . . . .	6
3.3	Beispiel . . . . .	6
3.4	Dateiendung . . . . .	6
<b>4</b>	<b>Pseudocode</b>	<b>7</b>
<b>5</b>	<b>Programmdokumentation</b>	<b>8</b>
<b>6</b>	<b>Projektablauf</b>	<b>9</b>
<b>7</b>	<b>Verwendete Mittel</b>	<b>10</b>
7.1	Programme . . . . .	10
7.2	Quellen . . . . .	10
7.2.1	WWW . . . . .	10
7.2.2	Literatur . . . . .	10

# 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.

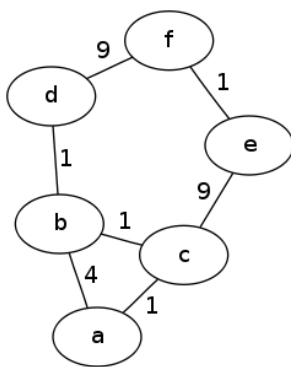


Abbildung 1: Einfacher gewichteter Graph

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:  $f = 11$ .

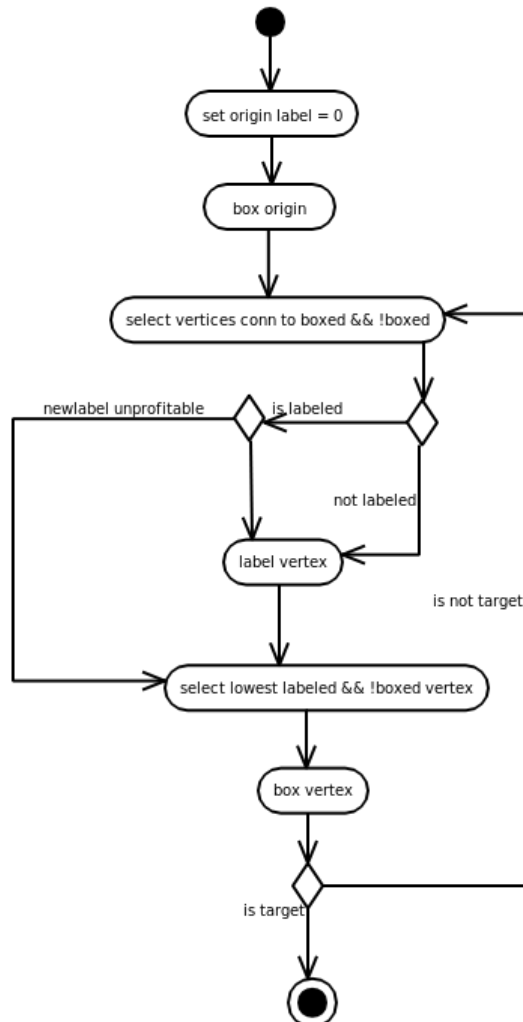
Tiefsten temporär beschrifteten Knoten einrahmen:  $f$ .

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



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

Abbildung 2: UML Aktivitätsdiagramm des Dijkstra

## 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.

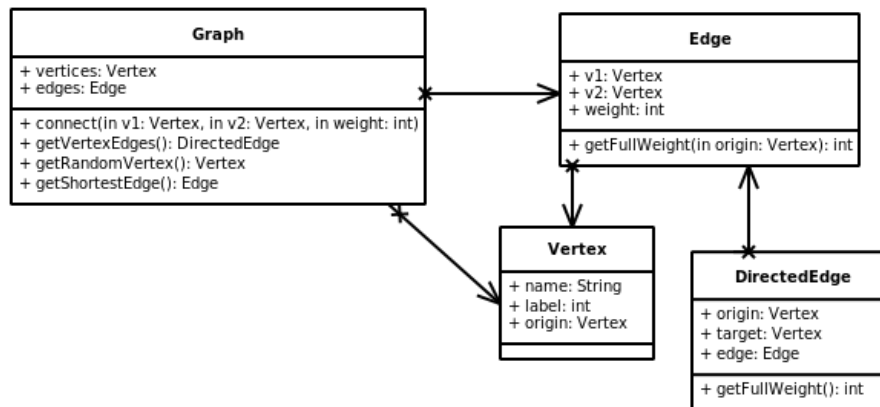


Abbildung 3: UML Klassendiagramm des Modells

### 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.

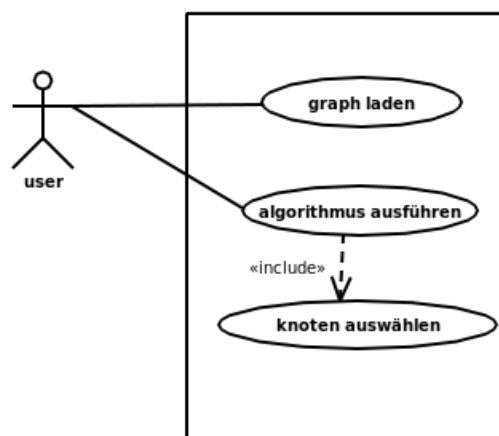


Abbildung 4: UML Anwendungsfall Diagramm

**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.

[typ]      [tab]      [token]      [tab]      [token\_n]      [linefeed]

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

// endlosschleife
while true
    boxed += boxedVertex
    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

            if boxedVertex == target
                // ziel erreicht
                break

        // 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 *Boldi-Wiedler-Graph* wurde in Java realisiert.

## 6 Projektablauf

## 7 Verwendete Mittel

### 7.1 Programme

- *Eclipse*: Entwicklungsumgebung für Java
- *Gaphor*: UML Diagramm Tool
- *Graphviz*: Graphenvisualisierung
- *Gifsicle*: Animierte gifs erstellen
- $\text{\LaTeX}$ : Textsatz-Software

### 7.2 Quellen

#### 7.2.1 WWW

- *Dijkstra's Algorithm (PDF)*
- *Der Algorithmus von Dijkstra (PDF)*
- *ASCII – Wikipedia*
- *LaTeX – Wikibooks*

#### 7.2.2 Literatur

- *Networks – Student text and unit guide, Cambridge 2000*
- andere Unterlagen aus dem Unterricht