Theorie und JAVA-Realisierung ausgewählter Algorithmen zur Bestimmung kürzester Wege in Graphen

Diplomarbeit

im Studiengang Informatik

der Fachhochschule Dortmund

von

Robin Quast
geb. am 29.12.1970
Matrikelnr. 7034489

Dortmund, den 12. Mai 2004

Betreuer: Prof. Dr. Burkhard Lenze

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I
Einleitung

1 Überblick

Jeder kennt das Problem, wie kommen wir auf dem schnellsten Weg in den Urlaub, nach Hause oder zu einem anderen Ziel. Ob mit dem Auto, der Bahn oder mit dem Fahrrad, die Auswahl der geeigneten Strecke wird meistens durch die Intuition oder den Blick auf eine Karte bestimmt. Aber auch hier kann der Computer uns helfen. Im Internet gibt es zahlreiche Routenplaner, meist liegen diese auch Atlanten bei oder wir können sie in der Bibliothek ausleihen. Natürlich können wir uns auch einen aktuellen Routenplaner kaufen.

Bei der Beförderung von Paketen oder Briefen taucht ein ähnliches Problem auf, wie können wir möglichst kostengünstig die Post von A nach B befördern? Was für eine Technologie steckt also hinter der gesamten Problematik der kürzesten Wege? Wie können wir einfach und effizient einen kürzesten, schnellsten oder kostengünstigsten Weg von einem Start- zu einem oder mehreren Zielpunkten finden? Zur einfachen Darstellung des Problems benutzen wir die „Sprache“ der Graphentheorie (siehe Kapitel II ab Seite 2).

Auf dieser Basis wollen wir uns die Algorithmen zur Berechnung der kürzesten Wege verdeutlichen und anhand von Beispielen aufzeigen (siehe Abschnitt 5). Im weiteren Teil der Diplomarbeit wollen wir aufzeigen, wie wir unser Tool *Yet another Visualiser*[15] entwickelt haben (siehe Kapitel III). Ziel des Tools ist es, dem Benutzer die hier vorgestellten Graphalgorithmen zu veranschaulichen. Zum Abschluss möchten wir noch einen kleinen Ausblick auf Programmverbesserungen, weiterführende Theorie und andere Programme, die sich mit dem Thema Graphalgorithmen befassen, geben.

Teil II
Graphentheorie

In diesem Teil der Diplomarbeit wollen wir grundlegende Begriffe erklären und definieren, sowie die Darstellung von Graphen aufzeigen. Weiterhin besprechen wir benötigte Datenstrukturen und eine Auswahl von kürzeste Wege Algorithmen. Teilweise Anlehnung an die Bücher von Ralf Hartmut Güting „Datenstrukturen und Algorithmen“[4], Jørgen Bang-Jensen und Gregory Gutin „Digraphs: Theory, Algorithms and Applikations“[3] und von Volker Turau „Algorithmische Graphentheorie“[6].

2 Grundlagen

Ein Graph stellt eine Menge von Objekten und Relationen auf diesen Objekten dar. Beispiele sind:

- Ein Stammbaum ist ein Graph. Die Objekte sind durch Personen dargestellt und eine Relation zwischen den Objekten x und y ist z.B. x ist Kind von y.
- Eine Straßenkarte ist ein Graph. Die Städte sind die Objekte und die Relationen sind die Straßen zwischen diesen Städten.
- Die Beziehung zwischen Personen kann als Graph dargestellt werden. Die Personen sind die Objekte und die Relationen sind die Beziehungen der Personen zueinander, z.B. Person x ist Freund von y.

Die Objekte werden durch Knoten (Kreise) und Relationen durch gerichtete Kanten (Pfeile) dargestellt. Eine ungerichtete Kante (Strecke) ist ein Spezialfall und kann einfach durch zwei entgegengesetzt gerichtete Kanten ersetzt werden (siehe Abbildung 1 auf Seite 3). Aus diesem Grund wollen wir uns im Weiteren nur noch mit gerichteten Graphen (auch Digraphen genannt) befassen.

2.1 Gerichteter Graph/ Digraph

Ein gerichteter Graph D, oder einfach Digraph, besteht aus einer Knotenmenge V und einer Kantenmenge E mit gerichteten Kanten. Wir schreiben $D = (V, E)$, welches bedeutet, dass V und E die Knoten-, bzw. Kantenmenge von D sind[1]. Es gilt (siehe auch [4]):

1. V ist endlich, nichtleer und enthält nur Elemente aus der Menge der ganzen Zahlen, einschließlich Null.

[1] V und E wurden aus dem Englischen abgeleitet, wobei *v* für *vertices* und *e für* edges *steht*

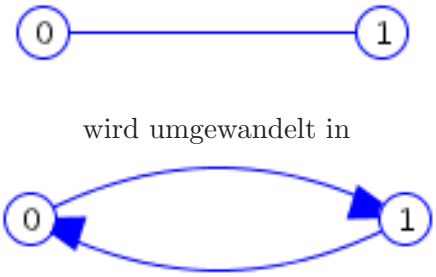


Abbildung 1: Umwandlung ungerichteter Graph → Digraph

2. $E \subseteq V \times V$

Die Elemente aus V sind Knoten, die wir in dem theoretischen Teil der Diplomarbeit mit kleinen Buchstaben oder mit Buchstaben und Indizes, z.B. v_j bezeichnen wollen. Im praktischen Teil und in den Abbildungen werden wir die Knoten nummerieren[2]. Eine gerichtete Kante $e_{v,w}$ ist also ein geordnetes Paar und es gilt (siehe auch [5]):

$$e_{v,w} = (v, w) \text{ mit } v, w \in V$$

Dabei wird v als Startknoten (oder tail) und w als Zielknoten (head) bezeichnet. Man sagt die Kante $e_{v,w}$ geht von v nach w und ist **inzident** zu v und w. Die Knoten v und w sind **adjazent**, **benachbart** oder **Nachbarn**[3]. Für den Graph aus Abbildung 4 (siehe Seite 6) gilt: die Knoten 1 und 4 sind inzident zu der Kante $e_{4,1} = (4, 1)$ und der Knoten 1 ist adjazent zu 4, bzw. 4 ist adjazent zu 1.

2.2 Gewichteter Digraph

Ein gewichteter Digraph D ist ein Digraph mit einer Kostenfunktion $c : E \to \Re$. Es gilt dann (siehe [5]):

$$D = (V, E, c).$$

Man sagt auch der Digraph ist kantenbewertet. Eine **gewichtete Kante** wird durch ein Tupel dargestellt, wobei $c_{v,w}$ das Kantengewicht repräsentiert.

$$e_{v,w} = (v, w, c_{v,w}) \text{ mit } v, w \in V \text{ und } c_{v,w} \in \Re$$

[2] 0,1,...

[3] In der Literatur wird auch von manch anderen Autoren vereinbart, dass v nur dann adjazent zu w ist, wenn eine gerichtete Kante von v zu w existiert, also $e_{v,w} = (v, w)$. Wir werden durch die Adjazenz zweier Knoten lediglich voraussetzen, dass eine Kante $e_{v,w} = (v, w)$ oder $e_{w,v} = (w, v)$ existiert. Die Richtung ist für die Adjazenz nicht relevant.

Weiterhin soll für die Kante $c_{v,w} = c(e_{v,w})$ gelten. Bezüglich der Kante $e_{v,w}$ ist v der Vorgänger von w und entsprechend w der Nachfolger von v. Kanten, deren tail und head gleich sind, nennt man **parallele** Kanten. Wir wollen das Vorhandensein paralleler Kanten ausschließen, da für das Finden des kürzesten Weges nur eine der parallelen Kanten, nämlich die mit den kleinsten Gewicht, also die mit den minimalen Kosten relevant ist. Alle anderen Kanten sind teurer und können nicht auf dem kürzesten Weg liegen.

Der **Grad** eines Knotens ist die Anzahl aller ein- und ausgehenden Kanten. Entsprechend ist der **Eingangsgrad** (**Ausgangsgrad**) eines Knotens die Anzahl aller eingehenden (ausgehenden) Kanten.

Ein **Pfad**[4] ist eine Knotenfolge $P = (v_1, v_2, \ldots, v_n)$, so dass für $1 \leq i \leq n-1$ gilt: $(v_i, v_{i+1}) \in E$. v_1 wollen wir als **Start-** und v_n als **Zielknoten** bezeichnen. Die **Länge** des Pfades ist die Anzahl der Kanten auf dem Pfad (also n-1). Alle Kanten, die auf dem Pfad P liegen, fassen wir in der Menge $E(P)$ zusammen. Wenn alle Knoten auf dem Pfad paarweise unterschiedlich sind, dann heißt der Pfad **einfach** (Ausnahme $v_1 = v_n$). Ein Pfad ist **geschlossen**, wenn Start- und Zielknoten identisch sind ($v_1 = v_n$).

2.3 Weg

Ein Pfad P heißt **Weg**[5] im Graphen D, wenn gilt (siehe [5]):

$$\forall e_{x,y}, e_{v,w} \in E(P) \text{ mit } e_{x,y} = (x, y) \text{ und } e_{v,w} = (v, w) :$$

$$e_{x,y} \neq e_{v,w} \text{ mit } x, y, v, w \in V$$

Sind also alle verwendeten Kanten des Pfades verschieden, handelt es sich um einen Weg. Auch hier gilt: Ist Startknoten gleich Zielknoten, ist der Weg geschlossen, andernfalls offen. Beispiel: Die farbig markierten Kanten $(5, 3, 2)$, $(3, 4, 22)$, $(4, 2, 12)$ und $(2, 0, 12)$ in Abbildung 2 bilden einen Weg[6] $P = (5, 3, 4, 2, 0)$.

Ein Weg, der nur aus einem Knoten besteht ($P = (v)$) ist ein trivialer Weg. Die **Länge** eines Weges ist gleich der Anzahl der Kanten, die auf diesem Weg liegen. Die **Kosten** eines Weges P bilden sich aus der Summe der Kosten der Kanten, die diesen Weg bilden. Es gilt für die Weglänge n:

$$c(P) := \sum_{i=0}^{n} c(e_i)$$

Beispiel: Die Kosten des Weges in Abbildung 2 sind 48.

Ein Weg P heißt **einfach**, wenn alle verwendeten Knoten, bis auf Start- und Zielknoten, verschieden sind[7]. Beispiel: Der Weg in Abbildung 2 ist einfach.

[4] englisch walk
[5] englisch path
[6] Dieser Weg ist aber nicht der kürzeste Weg.
[7] siehe auch einfacher Pfad

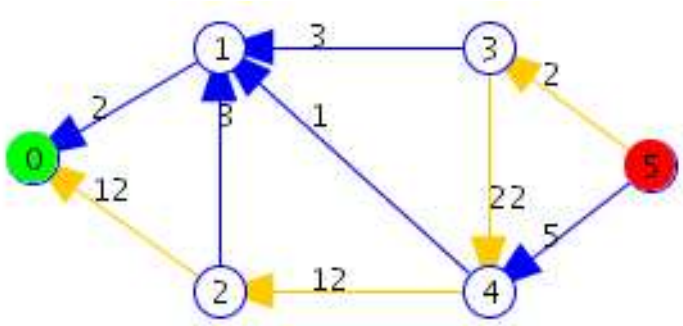


Abbildung 2: Ein Weg mit Startknoten 5 und Zielknoten 0

Eine **Schleife**, bzw. **trivialer Zyklus**, ist eine Kante, deren tail und head gleich sind ($e_{x,x} = (x, x)$). Hat die Schleife positive Kosten oder sind die Kosten der Schleife 0, dann kann man diese im Allgemeinen für die Berechnung der kürzesten Wege vernachlässigen, da angenommen wird, dass der Weg von dem Knoten zu sich selber immer 0 ist und somit der Weg über die Schleife länger wäre. Hat die Schleife negative Kosten, dann handelt es sich um einen negativen Zyklus, mit dem wir bei nochmaliger Benutzung den kürzesten Weg immer wieder verkürzen könnten. Durch das nochmalige Benutzen einer Kante würde aber das Wegkriterium (siehe 2.3) verletzt, da die Kanten auf dem gefundenen „Weg“ nicht paarweise verschieden sind. Wir wollen für unsere Graphalgorithmen negative Zyklen ausschließen, so dass wir Schleifen mit negativen Kosten vernachlässigen können.

Ein Digraph D heißt **azyklisch**, wenn es keinen Zyklus in D gibt. Beispiel: Der Graph in Abbildung 4 (Seite 6) ist azyklisch. Die **Ordnung** eines Graphen D ist die Anzahl der Knoten in D. Sie wird oft auch mit $\mid D \mid$ bezeichnet. Der Graph aus Abbildung 4 (siehe Seite 6) hat die Ordnung 6. Ein Digraph, der Schleifen enthalten darf nennen wir gerichteten **Multigraph**. Beispiel: Der Graph aus Abbildung 4 (siehe Seite 6) ist natürlich auch ein gerichteter Multigraph, der jedoch keine Schleifen enthält.

Ein **Teilgraph** eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$. G' ist also ein Teilgraph von G, wenn die Knoten- und Kantenmenge von G' eine Teilmenge der Knoten- und Kantenmenge von G ist[8]. Beispiel: Der Graph $G' = (V', E')$ mit $V' = \{1, 3, 5\}$ und $E' = \{(3, 1, 3), (5, 3, 2)\}$ (siehe Abbildung 3) ist ein Teilgraph von G (siehe Abbildung 4, Seite 6).

[8]Die Kanten bestehen alle aus Knoten der Knotenmenge V'.

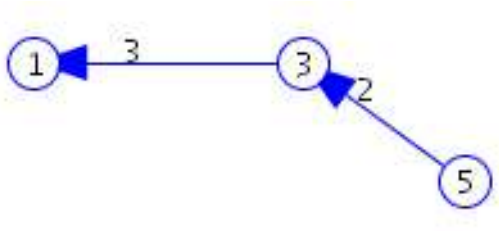


Abbildung 3: Beispiel eines Teilgraphen

3 Darstellung

Graphen können auf unterschiedlichste Weise dargestellt werden. Die gebräuchlichsten Darstellungen seien hier anhand des Beispielgraphens aus Abbildung 4 aufgeführt.

3.1 Graphische Darstellung

Ein Graph kann als eine Graphik dargestellt werden. Knoten werden dabei durch Kreise und Kanten durch Verbindungen zwischen den Kreisen dargestellt. Soll es sich bei der Kante um eine gerichtete Kante handeln, erhält diese eine Pfeilspitze am Ende des Zielknotens.

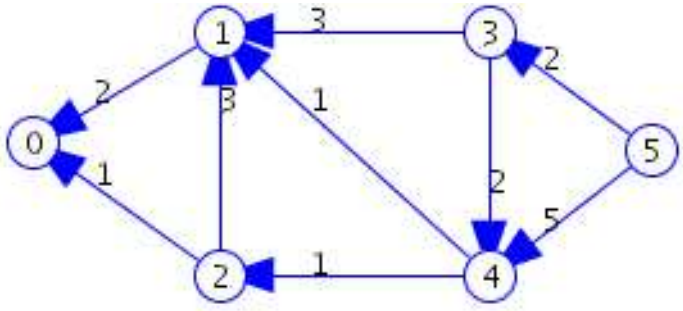


Abbildung 4: graphische Darstellung eines gerichteten Graphen

3.2 Adjazenzliste

Ein Graph kann durch seine Adjazenzliste dargestellt werden. Dazu wird zu jedem Knoten eine Liste von direkt über eine Kante erreichbaren Knoten erstellt. Als Ergebnis haben wir zu jedem Knoten eine Kantenliste der ausgehenden Kanten. Für das obige Beispiel (siehe Abbildung 4) ist die Adjazenzliste in Tabelle 4 angegeben.

0:	
1:	0
2:	0 1
3:	1 4
4:	1 2
5:	3 4

Tabelle 1: Adjazenzliste eines Digraphen ohne Kostenfunktion

Von dem Knoten 1 gibt es also eine Kante zu dem Knoten 0 (in der Abbildung 4 wäre das die Kante $e_{1,0} = (1, 0, 2)$).

0:	
1:	0,2
2:	0,1 1,3
3:	1,3 4,2
4:	1,1 2,1
5:	3,2 4,5

Tabelle 2: Adjazenzliste eines Digraphen mit Darstellung der Kostenfunktion

Enthalten die Kanten des Digraphen eine Kostenfunktion, so werden die Kosten einer Kante nach dem Zielknoten durch Komma getrennt aufgeführt. Die Kante $e_{1,0}$ wird nun durch das Tupel $(1, 0, 2)$ repräsentiert. Es existiert also eine Kante von Knoten 1 zu Knoten 0 mit den Kosten 2 (in der Abbildung 4 wäre das die Kante $e_{1,0} = (1, 0, 2)$).

Vorteil dieser Darstellung ist der geringe Speicherbedarf $O(n + m)$ für $n =\mid V \mid$ und $m =\mid E \mid$. Alle Nachbarn eines Knotens können in Linearzeit erreicht werden. Ein Test, ob v und w benachbart sind, kann nicht in $O(1)$ erfolgen, da dazu die Kantenliste von v nach der Suche von w durchlaufen werden müsste. Falls zu einem Knoten alle adjazenten Vorgängerknoten gefunden werden sollen, muss sogar die ganze Adjazenzliste durchsucht werden. Alternativ könnte man zusätzlich für dieses Problem eine **inverse** Adjazenzliste verwalten, um so schnelleren Zugriff auf die benachbarten Vorgänger zu haben.

3.3 Adjazenzmatrix

Die Darstellung mittels einer Adjazenzmatrix spiegelt die Kanten eines Graphen in einer Matrix wieder. Zu der gerichteten, ungewichteten Kante $e_{1,2}$ gibt es in der Matrix an der Stelle (1,2) eine 1 als Eintrag. Existiert keine Kante zwischen den Knoten, ist an der entsprechenden Stelle eine 0 (oder ein anderes vereinbartes Element) zu finden. Die Größe der Matrix entspricht

der Anzahl der Knoten. Hat also ein Graph 4 Knoten, so benötigen wir eine 4x4 Adjazenzmatrix. Diese Darstellung hat den Nachteil, dass auch bei Graphen mit wenig Kanten und vielen Knoten die Größe der Matrix nicht verringert werden kann und somit viele Einträge mit Nullen in der Adjazenzmatrix zu finden sind. Vorteil dieser Darstellung, ist unter Anderem dass die Existenz einer Kante in einer Laufzeit von $O(1)$ festgestellt werden kann.

$$Digraph: \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Tabelle 3: Adjazenzmatrixdarstellung ohne Kostenfunktion

Soll die Adjazenzmatrix eines gewichteten Digraphen angegeben werden, so werden statt einer 1 die Kosten der entsprechenden Kante an die Stelle in der Matrix gesetzt. Die Kante $e_{1,0} = (1, 0, 2)$ wird durch den Eintrag 2 an der Stelle (1, 0) dargestellt[9]. Sind Kanten mit den Kosten 0 zugelassen, so muss bei der Implementierung ein anderes Element vereinbart werden, dass anzeigt, dass eine Kante nicht existiert, dies könnte z.B. auch die größte Integer Zahl[10] sein.

$$Digraph: \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 2 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 5 & 0 \end{pmatrix}$$

Tabelle 4: Adjazenzmatrixdarstellung mit Kostenfunktion

Die Adjazenzmatrix- und die Adjazenzlistendarstellung sind meist als **statische** Datenstrukturen zu sehen. Diese werden vor dem Start eines Algorithmus aufgebaut und während des Algorithmus in der Regel nicht verändert. Dadurch spielt die Laufzeit der Änderungsmethoden, wie z.B. insert, delete, für den Algorithmus keine Rolle.

[9] Die Spalten und Zeilen der Matrix wollen wir mit 0 beginnend nummerieren.
[10] Integer.MAX_VALUE bei Benutzung von Java

3.4 Weitere Darstellungen

Andere Darstellungsmöglichkeiten bieten die Inzidenzmatrix- und die Forward-, bzw. Reverse, Star- Darstellung. Die Inzidenzmatrix stellt spaltenweise eine Kante dar. Die beteiligten Knoten werden durch den entsprechenden Eintrag in der Zeile durch eine -1, für den Startknoten, und eine 1 für den Zielknoten markiert (für gewichtete Kanten entsprechend mit dem Gewicht der Kante). Diese Darstellung ist für Graphen mit wenig Kanten geeignet. Für Graphen mit vielen Kanten enthält die Matrix viele Spalten und wird dadurch sehr groß. Die Forward Star Darstellung repräsentiert den Graphen durch eine Kantenliste, die durch ein Indexarray (fpt) ergänzt wird. Dieser Index gibt an, ab welcher Stelle in der Kantenliste die Kanten eines entsprechenden Knotens zu finden sind. Tabellen 5 und 6 zeigen die Inzidenzmatrix und die Forward Star Darstellung für den Graphen aus Abbildung 4.

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 3 & 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & -3 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -3 & -2 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 & -1 & -1 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -5 \end{pmatrix}$$

Tabelle 5: Inzidenzmatrix Digraph

Knoten	Arrayindex
0:	10
1:	1
2:	2
3:	4
4:	6
5:	8

Index	tail	head	cost
1:	1	0	2
2:	2	0	1
3:	2	1	3
4:	3	1	3
5:	3	4	2
6:	4	1	1
7:	4	2	1
8:	5	3	2
9:	5	4	5

Tabelle 6: Forward Star Darstellung

4 Datenstrukturen

4.1 Heap

Für die Algorithmen *Bellman-Ford-Moore* und *Dijkstra* benötigen wir schnelle Zugriffe auf Heapelemente[11] bzgl. folgender Operationen:

- *decreaseKey*, ändert den Schlüssel eines Knotens innerhalb der Datenstruktur.
- *insert*, fügt einen Knoten, dem ein Schlüssel zugeordnet wurde, in eine Datenstruktur ein.
- zusätzlich für Bellman-Ford-Moore
  - *findFirstElement*, gibt das erste Heapelement in der Datenstruktur zurück.
  - *deleteFirstElement*, löscht das erste Heapelement in der Datenstruktur.
- zusätzlich für Dijkstra
  - *findMin*, gibt das Element mit dem kleinsten Schlüssel innerhalb der Datenstruktur zurück.
  - *deleteMin*, löscht Heapelement mit dem kleinsten Schlüssel aus der Datenstruktur.

Da die Sortierung innerhalb der Datenstruktur für die Algorithmen *Bellman-Ford-Moore* und *Dijkstra* unwichtig ist, bietet sich hier eine Prioritätswarteschlange an[12].

Diese könnte auf unterschiedliche Weise implementiert werden (siehe Tabelle 7). Zu der Tabelle noch ein paar Bemerkungen: Die sortierte, lineare Liste sei in dem Beispiel eine Implementation mit Zeiger auf das Minimum. Dadurch sind die Operationen *findMin* und *deleteMin* schnell. Die Operation *decreaseKey* sei durch *delete* und *insert* realisiert. Die Tabelle ist aus den Quellen [7] und [5] zusammengetragen worden.

Für das Bearbeiten von kleinen Graphen, in unserem Fall bis maximal 100 Knoten, reicht eine einfache Heap- Implementation. Unsere Implementierung auf Vektorbasis, sucht das kleinste Element, indem es den Heap sortiert und das erste Element zurückgibt. Für die Sortierung wird die Methode *sort* der Klasse *Collections* verwendet. Da diese auf den mergesort-

[11] Ein Heapelement ist, in unserem Fall, ein Element, dass einen Knoten und einen Schlüssel enthält. Dieser Schlüssel stellt, innerhalb der Algorithmen Bellman-Ford-Moore und Dijkstra, den aktuell gefundenen kürzesten Weg dar.

[12] Für *Bellman-Ford-Moore* würde auch eine einfache Warteschlange reichen (z.B. ein Stack). Da hier nicht der Zugriff auf das Element mit dem kleinsten Schlüssel erfolgen muss.

	sortierte, lineare Liste	Heap (Vektorbasis)	Skipliste	Fibonacci Heap
create	O(n log n)	O(n log n)	O(n log n)	O(1)
insert	O(n)	O(1)	O(log n)	O(1)
findMin	O(1)	O(n log n)	O(1)	O(1)
deleteMin	O(1)	O(n log n)	O(log n) wc O(1) ac	O(log n)
decreaseKey	O(n)	O(n)	O(log n)	O(1)
delete	O(n)	O(n)	O(log n)	O(log n)

Tabelle 7: Datenstrukturen und deren Laufzeit

Algorithmus basiert (siehe JDK Beschreibung), erhalten wir für die Sortierung die Laufzeit *O(n log n)* und für das anschließende Ausgeben des ersten Elements, dass das Element mit dem kleinsten Schlüssel ist, *O(1)*, also eine Laufzeit von *O(n log n)* für *findMin* und *deleteMin* . Die Operationen *decreaseKey* und *delete* benötigen *O(n)*, da für die Elementsuche im worst case alle Elemente geprüft werden müssen[13]. Damit ist unser Heap zwar im Vergleich schlechter als eine sortierte, lineare Liste, doch für unsere Datenmengen vollkommen ausreichend. Vorteil dieser Implementierung ist, dass der Quellcode auf bereits vorhandene Methoden der Klassen *Vector* und *Collections* aufbaut und somit sehr robust und einfach ist.

Knoten	Schlüssel	Knoten	Schlüssel
0	∞	4	4
2	∞	1	5
1	5	0	∞
4	4	2	∞

Tabelle 8: Heapimplementation

In der Tabelle 8 sind zwei Heaps zu sehen. Der linke Heap zeigt den Inhalt nach einer Einfügeoperation an. Der rechte Heap zeigt den Inhalt vor der Ausgabe, bzw. dem Löschen, des Elements mit dem kleinsten Schlüssel an. In diesem Fall wird der Heap vorher nach den Schlüsselelementen sortiert und das erste Element zurückgegeben.

Skiplisten Skiplisten sind 1990 von William Pugh erdacht worden. Sie sind normal verkettete Listen mit zusätzlicher Verzeigerung, um über Elemente hinweg springen (skippen) zu können. Dadurch ergibt sich eine baumähnliche Struktur. Jedoch wird bei Skiplisten, im Unterschied zu

[13]Es kann dabei nicht vorausgesetzt werden, dass der Heap sortiert ist.

Bäumen, mittels stochastischer Methoden „balanciert“. Durch die zusätzliche Verzeigerung entsteht etwa der doppelte Aufwand durch Zeiger- und Höhenverwaltung, aber dadurch wird das Suchen eines Elementes schneller[14]. Ein weiterer Vorteil ist das Auslesen der Elemente, da diese auf der untersten Ebene als „normale“, sortierte Liste erhalten bleiben. Skiplisten werden im Folgenden kurz veranschaulicht. Die Laufzeit von Skiplisten beträgt für die *deleteMin* Operation im worst case (wc) *O(log n)*, aber im average case (ac) *O(1)* (siehe auch Tabelle 7). Weiteres zu Skiplisten ist im Internet oder in [5] zu finden.

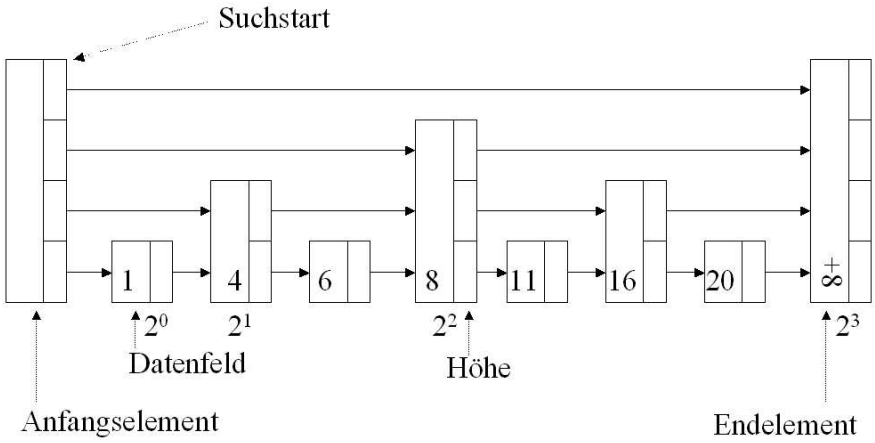


Abbildung 5: Skipliste

Fibonacci Heaps Fibonacci Heaps sind 1986 von Fredman und Tarjan erdacht worden. Unter Fibonacci Heaps kann man sich einen Wald von heapgeordneten Bäumen, ähnlich den Binomial Heaps[15], vorstellen, wobei diese jedoch weniger starr strukturiert sind. Die Idee von Fibonacci Heaps ist, die schnelle Laufzeit der *merge*- Operation, bei einer linearen Liste *O(1)*, auszunutzen, aber eine starre Struktur zu verlassen. Der Einstiegszeiger[16] zeigt auf ein Element mit dem kleinsten Schlüssel[17] (siehe Abbildung 6). Die Bäume sind auf der ersten Ebene und ebenso die Knoten innerhalb der Bäume durch eine ringförmig, doppelt verkettete Liste, miteinander verknüpft. Jeder Knoten besteht aus einer Verknüpfung zum Nachfolger, Vorgänger, dem linken und dem rechten Kind, einem Knotenwert, eine Marke und einem Knotenrang. Näheres zu Fibonacci Heaps ist unter anderem im dem Buch

[14] ähnlich binärer Suche, in $O(logn)$

[15] 1978

[16] root- oder Wurzelzeiger

[17] Es können wie im Beispiel mehrere Elemente den kleinsten Schlüssel besitzen, dann ist es egal, auf welches kleinste Element gezeigt wird.

von Niklaus Wirth[7] zu finden.

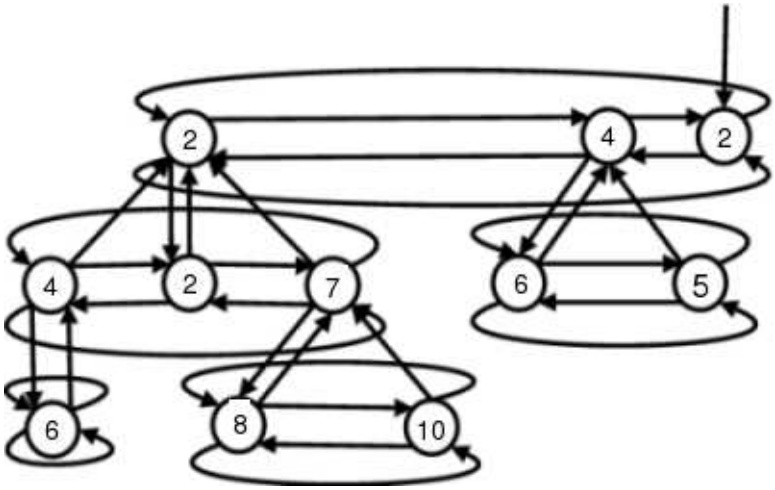


Abbildung 6: Fibonacci Heap

Weiterhin würden sich Leftist Heaps oder diverse Baumstrukturen , z.B. AVL-Bäume (siehe [4] und [7]) eignen.

4.2 Vorgängerliste

Wir benutzen eine Vorgängerliste, um den kürzesten Weg von einem Startknoten zu allen anderen Knoten wiederzugeben. Diese Vorgängerliste ist als Array von Integerzahlen (int[] vorgaenger;) implementiert. Wird zu einem Knoten x also der Vorgänger gesucht, dann kann einfach $vorgaenger[x]$ abgefragt werden. Wir könnten folgende Vorgängerliste für den Graphen aus Abbildung 4 erhalten, wobei wir den Startknoten 5 und den Zielknoten 0 angegeben haben: Beginnend mit dem Zielknoten wollen wir nun den kürzesten

Arrayindex bzw. Knoten	Vorgänger
0	2
1	3
2	4
3	5
4	3
5	-

Tabelle 9: Beispiel einer Vorgängerliste

Weg von Knoten 5 nach 0 ablesen. Der Vorgänger von Knoten 0 ist Knoten 2. Davon der Vorgänger ist Knoten 4. Dessen Vorgänger der Knoten 3 ist. 5

ist der Vorgänger von Knoten 3. Knoten 5 ist der Startknoten. Also haben wir den kürzesten Weg $P = (5, 3, 4, 2, 0)$ gefunden. Die Algorithmen werden uns aber keine Vorgängerliste zurückgeben, sondern einen Vektor von allen Kanten, die auf dem ermittelten kürzesten Weg liegen. Diese Kantenliste wird nach dem Algorithmus aus der Vorgängerliste aufgebaut und der aufrufenden Methode zurückgegeben.

4.3 Vorgängermatrix

Für den Algorithmus von Floyd-Warshall reicht eine Vorgängerliste nicht, da wir die kürzesten Wege zwischen allen Knoten berechnen. Also benötigen wir eine Vorgängermatrix. Die Vorgängermatrix ist ein Objekt der Klasse *Matrix* und basiert auf ein zweidimensionales Array von Objekten. Die

Array Index	0:	1:	2:	3:	4:	5:
0:	0					
1:	1	1				
2:	2	2	2			
3:	2	3	4	3	3	
4:	2	4	4		4	
5:	2	3	4	5	3	5

Tabelle 10: Implementierung der Vorgängermatrix

Vorgängermatrix ist im Prinzip nur eine Ansammlung von Vorgängerlisten. Wir haben in Zeile 0 die Vorgängerliste von dem Knoten 0, als Startknoten. Für Knoten 1 als Startknoten finden wir die benötigte Vorgängerliste in Zeile 1 usw... Genau wie bei der Vorgängerliste ermitteln wir hier den Weg von Knoten 5 nach Knoten 0. Dazu müssen wir uns also Zeile 5 anschauen und fangen bei dem Eintrag für den Zielknoten 0 an. Daraus ergibt sich, dass 2 der Vorgänger von 0 ist. Der Vorgänger von 2 ist 4. Davon ist 3 der Vorgänger. Der Startknoten 5 ist der Vorgänger von 3. Damit haben wir den kürzesten Weg von 5 nach 0 gefunden mit $P = (5, 3, 4, 2, 0)$.

4.4 Adjazenzliste

Für die Verwaltung des Graphen wollen wir u.a. die Adjazenzlistendarstellung benutzen. Dazu verwenden wir ein Array von Vektoren. Der Knotenname ist zugleich Index des Arrays. Über den Arrayindex erhalten wir einen Kantenvektor. Dieser enthält also alle Kanten zu dem Knoten. Dabei speichern wir in dem Kantenvektor alle Kanteninformationen, also ein Objekt der Klasse *Edge*. Dadurch speichern wir auch den Startknoten der Kante, ab, obwohl wir diesen ja durch den Arrayindex schon kennen. Vorteil ist,

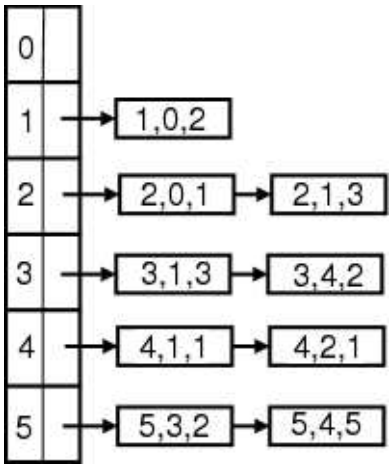


Abbildung 7: Adjazenzlistenimplementierung

dass wir dadurch ein einfaches Löschen und Einfügen von Kanten realisieren können. In Abbildung 7 haben wir unsere Implementierung der Adjazenzliste dargestellt. Wollen wir nun auf die Kante $e_{3,4}$ zugreifen, müssen wir uns die Kantenliste vom Index 3 geben lassen (Methode *getEdgesWithTail*) und die einzelnen Kanten durchlaufen, bis wir eine Kante finden, die die gleichen Knoten enthält (dies ist z.B. auch die Vorgehensweise der Methode *getCost*). Dabei ist zu beachten, dass die Kantenliste nicht unbedingt sortiert sein muss.

4.5 Adjazenzmatrix

Array Index	0:	1:	2:	3:	4:	5:
0:	0	0	0	0	0	0
1:	2	0	0	0	0	0
2:	1	3	0	0	0	0
3:	0	3	0	0	2	0
4:	0	1	1	0	0	0
5:	0	0	0	2	5	0

Tabelle 11: Implementierung der Adjazenzmatrix

Die Adjazenzmatrix wollen wir als zweidimensionales Array von Objekten verwalten, wobei wir Elemente vom Typ *Integer* in die einzelnen Zellen schreiben. Beim Auslesen und Setzen der Zellenwerte werden wir aber *int* Zahlen an die Methoden der Klasse *Matrix* übergeben. Zu beachten ist, dass der Arrayindex bei *0* und nicht bei *1* anfängt. Da wir Knoten mit ganzen

Zahlen größer gleich 0 benennen, stellt dies für uns kein Problem dar. Ist die Knotenbenennung lückenhaft, werden zu den Lücken (im Graphen nicht vorhandene Knoten) leere Einträge verwaltet, genau so als würden die Knoten ohne Kanten im Graphen vorkommen. Um das zu verhindern muss der Anwender für eine lückenlose Knotenbenennung sorgen.

5 Kürzeste Wege Algorithmen

Auf der Suche nach einem kürzesten Weg zwischen zwei Knoten könnte man zunächst auf folgende Idee[18] kommen. Wir könnten alle Wege eines Graphen vom Start- zum Zielknoten sammeln, und dann aus dieser endlichen Menge von Wegen, den Kürzesten bestimmen. Bei dem Graphen aus Abbildung 8 gibt es $2 + 2 * 4 = 10$ Knoten. Um nun den kürzesten Weg von 0 nach 9 zu bestimmen, müssten wir bei Knoten 0 starten und hätten bei den ersten vier Knoten, die auf einem möglichen Weg liegen, also bei den Knoten 0,1,2,3,5,6,7, jeweils 2 Möglichkeiten über eine andere Kante zu verzweigen. Daraus ergibt sich, dass wir $2^4 = 16$ Wege von 0 nach 9 haben. Verallgemeinern wir das Beispiel auf $2+2*50$, also auf 102 Knoten, ergeben sich zwischen dem Start- und dem Zielknoten 2^{50}, also mehr als 10^{15} Wege. Die Anzahl möglicher Wege kann somit exponentiell in der Anzahl der Knoten wachsen. Die Ermittlung des kürzesten Weges würde also auch auf sehr schnellen Rechnern viel Zeit in Anspruch nehmen. Fazit: Wir benötigen einen Algorithmus, der kürzeste Wege effizient und in annehmbarer Zeit berechnet.

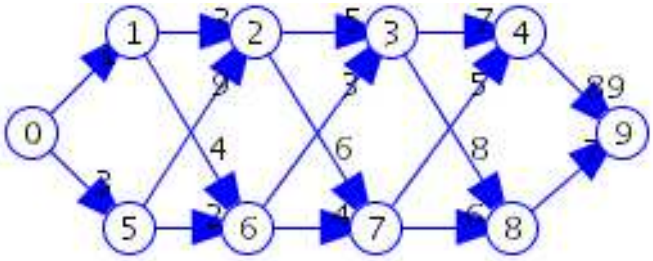


Abbildung 8: erster Lösungsversuch

Wir unterteilen nun unser kürzestes Wege Problem in drei Klassen:

1. Berechnung des kürzesten Weges zwischen zwei Knoten.
2. Berechnung der kürzesten Wege von einem Startknoten zu allen anderen Knoten[19].

[18] Das Einführende Beispiel wurde in geänderter Form von[8] übernommen.
[19] single source shortest path problem (SSSP)

3. Berechnung der kürzesten Wege zwischen allen Knoten[20].

Beispiel zu den einzelnen Problemklassen sind (siehe Graph Abbildung 4):

zu 1 Startknoten sein 5 und der Zielknoten sei 0. Ein Algorithmus der das Problem aus Klasse 1 löst wird einen kürzesten Weg von Knoten 5 nach Knoten 0 finden.

zu 2 Startknoten sei 5. Bei Lösung des Problems aus Klasse 2 wird der Algorithmus für jede Knotenkombination einen kürzesten Weg ermitteln. Also einen kürzesten Weg von Knoten 5 zum Knoten 0, einen weiteren kürzesten Weg von Knoten 5 nach Knoten 1, einen nach Knoten 2 usw...

zu 3 Bei der Problemklasse 3 wird keine Knotenvorgabe gemacht. Ist ein Graph mit n Knoten gegeben, dann wird im Prinzip zu jedem Knoten v_i, mit $1 \leq i \leq n$, das Problem der Klasse 2 gelöst. Jeder Knoten v_i wird also einmal zum Startknoten gemacht. Der Algorithmus wird dann z.B. kürzeste Wege zwischen Knoten 5 und allen anderen Knoten ermitteln. Weiterhin zwischen Knoten 0 und allen anderen Knoten, zwischen Knoten 1 und allen anderen Knoten usw...

Da bislang noch kein Algorithmus für die Berechnung eines kürzesten Weges zwischen 2 Knoten bekannt ist, dessen worst case Zeit geringer ist als die des effizientesten Algorithmus für die Berechnung zwischen einem Knoten und allen anderen Knoten, wollen wir hier zwei Algorithmen aus Problemklasse 2 und einen aus Klasse 3 vorstellen[21].

Vorab wollen wir noch einige Definitionen festlegen:

$p(x_0, x)$ gibt den Vorgänger des Knotens x zurück, der auf den bislang gefundenen kürzesten Weg vom angegebenen Startknoten x_0 zu dem Knoten x liegt. Ist der Algorithmus durchlaufen, kann man an der Vorgängerliste den kürzesten Weg zu dem Knoten x ablesen (siehe 4.2).

$k(x_0, x)$ steht für die minimalen Kosten, die auf den bislang gefundenen kürzesten Weg von x_0 zu dem Knoten x ermittelt wurden. Ist der Algorithmus durchgelaufen, kann man an $k(x_0, x)$ direkt die Länge des kürzesten Weges von Startknoten x_0 zum Knoten x ablesen.

$c(e_{v,w})$ ist die Kostenfunktion, für die Kante $e_{v,w}$.

$create(h)$ erzeugt einen leeren Heap h.

[20] all pairs shortest path problem (APSP)

[21] Lösen wir ein Problem der Klasse 2 oder 3 haben wir natürlich auch das entsprechende Problem aus den niedrigeren Klassen gelöst.

$insert(h, x, c)$ fügt einen Knoten x mit dem Schlüssel c in den Heap h ein.

$replace(h, x, c)$ ersetzt den Heapeintrag zu dem Knoten x, falls vorhanden, durch einen Eintrag des Knotens x mit dem Schlüssel c in den Heap h. Im Einzelnen bedeutet das, das Heapelement mit dem Knoten x wird zunächst aus dem Heap gelöscht. Anschließend wird es mit dem Schlüssel c angefügt. Dadurch wird unter Umständen das Heapelement in der Reihenfolge verschoben.

$findFirstElement(h)$ ist eine Methode, die das erste Element des Heaps h zurückgibt.

$deleteFirstElement(h)$ löscht das erste Element aus dem Heap h.

$findMin(h)$ gibt das Element mit dem kleinsten Schlüssel aus dem Heap h zurück.

$deleteMin(h)$ löscht das Element mit dem kleinsten Schlüssel aus dem Heap h.

$decreaseKey(h, x, c)$ ordnet dem Knoten x einen neuen Schlüssel c im Heap h zu.

$Adlist(x)$ bezeichnet die Liste der von x ausgehenden Kanten.

$Integer.MAX\_VALUE$ steht für den höchsten, möglichen Wert und wird zur Initialisierung von $k(x_0, x)$ benötigt. Alle gefundenen kürzesten Wege sind kleiner. Es muss also gewährleistet sein, dass dieser Wert durch die Länge des kürzesten Weges nie erreicht wird. $Integer.MAX\_VALUE$ wird in den Beispielen, zwecks einfacherer Lesbarkeit, auch durch ∞ dargestellt.

5.1 Bellman-Ford-Moore

Der folgende Algorithmus wurde von Bellman, Ford und Moore erdacht und entstammt ursprünglich aus den Quellen[31],[37] und[40]. Er gehört zur Problemklasse 2 und findet zu einem Startknoten die kürzesten Wege zu allen anderen Knoten, die von dem Startknoten aus erreichbar sind.

Voraussetzung Der Graph D ist ein gewichteter Digraph, mit Kantengewichten größer gleich Null[22]. Ein Startknoten v_0 muss festgelegt sein.

[22]Der Digraph darf für den Algorithmus, in der Regel aber auch negative Kantenbewertungen haben, jedoch keine negative Zyklen. Um negative Zyklen auszuschließen setzen wir, der Einfachheit halber, positive Kantengewichte vorraus.

Idee Beginnend mit dem Startknoten v_0 werden in einem Durchgang alle zu einem Knoten v_i existierende Nachfolger v_j, die über eine gerichtete Kante $e_{i,j} = (v_i, v_j)$ erreichbar sind, überprüft, ob die Distanz über diese Kante $e_{i,j}$ kürzer ist, als die bisher gefundene Distanz zu diesem Knoten v_j . Ist die Distanz kürzer, wird diese kürzere Distanz mit dem Knoten v_j in eine Warteschlange h geschrieben. Falls ein Eintrag in der Warteschlange zu dem Knoten v_j enthalten ist, wird dieser durch den neuen Eintrag ersetzt. Ein Knoten ist also nicht doppelt in der Warteschlange enthalten. Der Vorgänger des Knotens v_j wird anschließend auf v_i gesetzt. Für den nächsten Durchlauf wird nun der erste Knoten aus der Warteschlange entnommen und dessen ausgehenden Kanten überprüft. Ist die Warteschlange leer, dann ist kein weiterer Knoten mehr erreichbar und der Algorithmus terminiert. Der kürzeste Weg kann dann anhand der Vorgängerliste abgelesen werden (siehe 4.2 auf Seite 13).

Bemerkung: Aus der Warteschlange wird z.B. immer der erste Knoten entfernt und alle Nachfolger untersucht. Es kann aber auch ein beliebig anderer Knoten entfernt werden, z.B. der zuletzt Hinzugefügte. Dabei ist die bereits gefundene Distanz, die der Knotenschlüssel in der Warteschlange darstellt, nicht von Belang[23].

Bellman-Ford-Moore in Pseudocode Vorbedingungen: $\{c(e_{v,w}) \geq 0, \forall e_{v,w} \in E\}$ und es existieren keine parallelen Kanten. Der Graph liegt als Adjazenzliste vor. Der Startknoten v_0 ist gegeben.

1. alg BellmanFordMoore(v_0);
2. insert $(create(h), v_0, 0)$;
3. $p(v_0, v_0) \leftarrow v_0$;
4. $\forall v \neq v_0$ do
5. $\quad k(v_0, v) = Integer.MAX\_VALUE$;
6. od;
7. $k(v_0, v_0) = 0$
8. while $h \neq \{\}$
9. $\quad v \leftarrow$ findFirstElement (h);
10. $\quad$ deleteFirstElement (h);
11. $\quad \forall w \in$Adlist(v) do

[23] im Gegensatz zu Dijkstra, bei dem der Knoten mit dem kleinsten Schlüssel aus dem Heap entfernt wird

12. if $k(v_0, v)$+ c$(e_{v,w}) < k(v_0, w)$ then
13. decreaseKey $(h, w, k(v_0, v) + c(e_{v,w}))$;
14. replace $(h, w, k(v_0, w))$;
15. $p(v_0, w) \leftarrow v$;
16. fi;
17. od;
18. od;

In Zeile 2 wird eine leere Warteschlange erzeugt und der Startknoten v_0 in die Warteschlange eingefügt. Der Vorgänger von v_0 wird auf v_0 gesetzt. Ebenso die Kosten zu allen anderen Knoten auf $Integer.MAX\_VALUE$ (Zeile 4-6). Die Kosten des Startknotens werden mit 0 initialisiert (Zeile 7). $k(v_0, v)$ wurde in diesem Algorithmus als Array implementiert, da sich der Bezug auf den Startknoten v_0 in dem Algorithmus nicht mehr ändert und so ein speichern dieser Werte in einer Matrix nicht nötig ist.

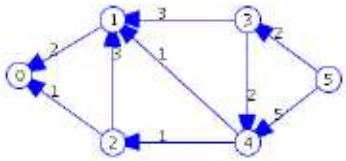
Solange die Warteschlange h nun nicht leer ist, wird das erste Warteschlangenelement aus der Warteschlange genommen und gelöscht (findFirstElement Zeile 9 und deleteFirstElement Zeile 10). Danach werden alle ausgehenden Kanten von diesem Knoten untersucht. Sind die Kosten des Zielknotens einer untersuchten Kante kleiner als die bisher gefundenen, dann setze die Kosten des Zielknotens auf die des Startknotens plus der Kosten der Kante (decreaseKey Zeile 13). Anschließend setze den Vorgänger des Zielknotens auf den Startknoten ($p(v_0, w) \leftarrow v$, Zeile 15). Danach wird der Zielknoten in die Warteschlange eingefügt, wenn dieser nicht schon in der Warteschlange enthalten ist, sonst wird nur der Schlüssel des Zielknotens überschrieben. Ist die Warteschlange leer terminiert der Algorithmus, da kein weiterer Knoten mehr erreichbar ist.

Die Warteschlange kann durch eine beliebig andere Warteschlange, mit den gleichen Funktionalitäten, ersetzt werden, z.B. eine Queue oder einem Stack. In der Implementierung verwenden wir einen Heap, wobei wir die Heapeigenschaft, das Element mit dem minimalen Schlüssel ist schnell erreichbar, nicht benutzten. Der Pseudocode ist aus dem Buch von Volker Turau[6] (Seite 248) übernommen, wobei wir die Notation aus dem Pascal ähnlichem Stil des Buches in einen allgemeineren Stil geändert haben.

Beispiel: Startknoten $v_0 = 5$

Adjazenzliste:

0:		
1:	0,2	
2:	0,1	1,3
3:	1,3	4,2
4:	1,1	2,1
5:	3,2	4,5



1. Schritt: (Initialisierung)
   Warteschlange h:5

$k(v_0, v)$:

0	1	2	3	4	5
∞	∞	∞	∞	∞	0

$p(v_0, v)$:

0	1	2	3	4	5
					5

2. Schritt:
   Warteschlange h:3,4

v=5; $k(v_0, v)$:

0	1	2	3	4	5
∞	∞	∞	2	5	0

$p(v_0, v)$:

0	1	2	3	4	5
			5	5	5

3. Schritt:
   Warteschlange h: 1,4

v=3; $k(v_0, v)$:

0	1	2	3	4	5
∞	5	∞	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
	3		5	3	5

4. Schritt:
   Warteschlange h: 4,0

v=1; $k(v_0, v)$:

0	1	2	3	4	5
7	5	∞	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
1	3		5	3	5

5. Schritt:
   Warteschlange h:0,2

v=4; $k(v_0, v)$:

0	1	2	3	4	5
7	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
1	3	4	5	3	5

6. Schritt:
   Warteschlange h:2

v=0 $k(v_0, v)$:

0	1	2	3	4	5
7	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
1	3	4	5	3	5

7. Schritt:
   Warteschlange h:0

v=2; $k(v_0, v)$:

0	1	2	3	4	5
6	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
2	3	4	5	3	5

8. Schritt:
   Warteschlange h ist leer

v=0; $k(v_0, v)$:

0	1	2	3	4	5
6	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
2	3	4	5	3	5

Nach dem Durchlauf des Algorithmus können wir an der Vorgängerliste $p(v_0, v)$ den kürzesten Weg zwischen dem Startknoten 5 und einem anderen Knoten, z.B. 0 ablesen (siehe dazu Abschnitt 4.2). Der kürzeste Weg zwischen Knoten 5 und Knoten 0 wäre also $P = (5, 3, 4, 2, 0)$.

Laufzeit Der Algorithmus Bellman-Ford-Moore hat eine Laufzeit von $O(mn)$, wobei m die Anzahl der Kanten und n die Anzahl der Knoten im Graph ist. Beweisidee: Wird für jeden Kantenvergleich die Distanz des bisher gefundenen Weges geändert, dann wird jeder Knoten maximal m mal in die Warteschlange h eingefügt. Hinzu kommt die Tiefe t des kürzesten Wege- Teilgraphen mit $t < n$. Es ergibt sich dann eine Laufzeit von $O(mn)$. Den ausführlichen Beweis findet man in dem Buch von Volker Turau[6] auf Seite 248-249.

5.2 Dijkstra

Der folgende Algorithmus wurde von Dijkstra erdacht und entstammt ursprünglich aus den Quellen[34]. Er löst das Problem aus Klasse 2, von einem festen Knoten aus kürzeste Wege zu sämtlichen Knoten zu bestimmen.

Idee Wir suchen einen Weg vom Startknoten v_0 zu einem Zielknoten v_n. Dazu vergrößern wir schrittweise eine Menge M von Knoten für deren Elemente v_i, mit $0 \leq i \leq n$, wir bereits einen kürzesten Weg von v_0 nach v_i gefunden haben. Allen anderen Knoten v_j, mit $i + 1 < j \leq n$, die nicht in M liegen, ordnen wir die Länge des bisher gefundenen kürzesten Weges zu.

Der Algorithmus benutzt obere Schranken ($k(v_0, v)$). $k(v_0, v)$ bezeichnet die Länge des bis dahin gefundenen kürzesten Weges von v_0 zum Knoten v. Falls noch kein Weg gefunden wurde, setzt man $k(v_0, v) = Integer.MAX\_VALUE$. Man organisiert die Menge der Knoten, die man im nächsten Schritt erreichen kann, als Prioritätswarteschlange [24] bzgl. eines Schlüssels[25]. Die Operationen, die zur Ermittlung kürzester Wege benötigt werden, sind auf Seite 17 erklärt.

[24] z.B. Heap oder Skipliste
[25] $k(v_0, v)$: kleiner Schlüssel = große Priorität

Um kürzeste Wege zurückzuverfolgen, merkt man sich zu jedem Knoten v den Vorgänger auf dem (bis dahin gefundenen) kürzesten Weg bezüglich des Startknotens v_0: $p(v_0, v)$ (für den Startknoten gilt: $p(v_0, v_0) = v_0$)

Dijkstra in Pseudocode Vorbedingungen: $\{c(e_{v,w}) \geq 0, \forall e_{v,w} \in E\}$. Der Graph liegt in Form einer Adjazenzliste vor und besitzt keine parallelen Kanten. Der Startknoten v_0 ist gegeben.

1. alg Dijkstra(v_0);
2. insert $(create(h), v_0, 0)$;
3. $p(v_0, v_0) \leftarrow v_0$;
4. $k(v_0, v_0) = 0$;
5. $\forall v \neq v_0$ do
6. insert $(h, v, Integer.MAX\_VALUE)$;
7. $k(v_0, v) = Integer.MAX\_VALUE$;
8. od;
9. while $h \neq \{\}$ and $min(h) \neq Integer.MAX\_VALUE$
10. $v \leftarrow$ findMin (h);
11. deleteMin (h);
12. $\forall w \in$Adlist(v) do
13. if $k(v_0, v)$+ c($e_{v,w}$) $< k(v_0, w)$ then
14. decreaseKey $(h, w, k(v_0, v) + c(e_{v,w}))$;
15. $p(v_0, w) \leftarrow v$;
16. fi;
17. od;
18. od;

In Zeile 2 wird ein leerer Heap erzeugt und der Startknoten v_0 in den Heap eingefügt. Der Vorgänger und die Kosten von v_0 werden auf v_0 gesetzt. Ebenso werden die Kosten zu allen anderen Knoten mit $Integer.MAX\_VALUE$ (Zeile 4-8) initialisiert. Alle Knoten v_i werden mit dem Kostenschlüssel $k(v_0, v_i)$ in den Heap eingefügt (Zeile 5-8).

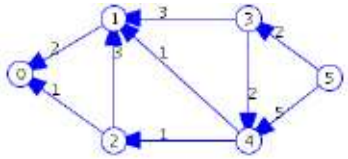
Solange der Heap h nun nicht leer ist, wird das Heapelement mit dem kleinsten Schlüssel aus dem Heap genommen und gelöscht (Zeile 10).

Dann werden alle ausgehenden Kanten zu diesem Knoten untersucht (Zeile 11). Sind die Kosten des Zielknotens einer untersuchten Kante kleiner als die bisher gefundenen, dann setze die Kosten des Zielknotens auf die des Startknotens plus den Kosten der Kante (Zeile 14). Der Vorgänger des Zielknotens wird auf den Startknoten gesetzt(Zeile 15). Ist der Heap leer oder der Schlüssel des minimalen Heapelements ist gleich dem Wert $Integer\_MAXVALUE$, terminiert der Algorithmus, da kein weiterer Knoten mehr erreichbar ist (Zeile 9). Wird also ein Knoten aus dem Heap gewählt, welcher als Schlüssel $Integer\_MAXVALUE$ hat ist dieser Knoten vom Startknoten v_0 nicht erreichbar. Terminiert der Algorithmus wenn der Heap noch ein Heapelement mit dem Schlüssel $Integer\_MAXVALUE$ enthält, dann existiert kein Weg vom Startknoten v_0 zu den jeweiligen Knoten mit dem Wert $Integer\_MAXVALUE$. $k(x, y)$ wurde in diesem Algorithmus als Array implementiert, da sich der Bezug auf den Startknoten v_0 in dem Algorithmus nicht mehr ändert. Die Vorlage für den Pseudocode stammt aus der Vorlesung „Algorithmen und Datenstrukturen“ von W. Hauenschild[5] und wurde von uns leicht überarbeitet.

Beispiel: Startknoten 5

Adjazenzliste:

0:		
1:	0,2	
2:	0,1	1,3
3:	1,3	4,2
4:	1,1	2,1
5:	3,2	4,5



1. Schritt: (Initialisierung)
   Heap h:5,0,1,2,3,4

$k(v_0, v)$:

0	1	2	3	4	5
∞	∞	∞	∞	∞	0

$p(v_0, v)$:

0	1	2	3	4	5
					5

2. Schritt:
   Heap h:0,1,2,3,4

v=5; $k(v_0, v)$:

0	1	2	3	4	5
∞	∞	∞	2	5	0

$p(v_0, v)$:

0	1	2	3	4	5
			5	5	5

3. Schritt:
   Heap h:0,2,1,4

v=3; $k(v_0, v)$:

0	1	2	3	4	5
∞	5	∞	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
	3		5	3	5

4. Schritt:
   Heap h:1,2,0

v=4; $k(v_0, v)$:

0	1	2	3	4	5
	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
	3	4	5	3	5

5. Schritt:
   Heap h:2,0

v=1; $k(v_0, v)$:

0	1	2	3	4	5
7	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
1	3	4	5	3	5

6. Schritt:
   Heap h:0

v=2 $k(v_0, v)$:

0	1	2	3	4	5
6	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
2	3	4	5	3	5

7. Schritt:
   Heap h ist leer

v=0; $k(v_0, v)$:

0	1	2	3	4	5
6	5	5	2	4	0

$p(v_0, v)$:

0	1	2	3	4	5
2	3	4	5	3	5

Genau wie bei dem Bellman-Ford-Moore Algorithmus erhalten wir als kürzesten Weg zwischen Knoten 5 und Knoten 0: $P = (5, 3, 4, 2, 0)$.

Laufzeit Da $c(e_{v_i,v_j}) > 0 \; \forall v_i, v_j \in V$ gilt, kann man die kürzesten Wege unter den einfachen Wegen suchen. Aus der Tatsache, dass nur endlich viele einfache Wege existieren, gibt es mindestens einen kürzesten Weg zwischen zwei Knoten oder gar keinen Weg. Sei nun $l(v_0, v)$ die Länge eines kürzesten Weges von v_0 nach v_i mit $l(v_0, v) = Integer.MAX\_VALUE$, wenn es keinen Weg gibt.

Beweisidee Wir benennen die Durchläufe mit dem Knoten $v_i = findMin(h)$. Weiterhin wollen wir beweisen, dass die folgenden beiden Schleifeninvarianten gelten:

1. $\forall v_i \neq v_0$ mit $k(v_0, v_i) < Integer.MAX\_VALUE$ gibt es einen Weg P von v_0 nach v_i der Länge $k(v_0, v_i)$ mit Vorgänger $p(v_0, v_i)$.

2. $\forall v_i \notin h$ gilt: $k(v_i) = l(v_i)$. Die kürzeste Weglänge wurde gefunden. Diese wird nicht mehr geändert, weil der Knoten v_i nicht mehr im Heap h enthalten ist und auch nicht mehr in den Heap eingefügt wird..

Den Beweis führt man durch Induktion. Nach dem letzten Durchlauf gilt:

$\forall v_i \in V : k(v_0, v_i) = l(v_0, v_i)$ und

$\forall v_i \in V$ mit $l(v_0, v_i) < Integer.MAX\_VALUE$ liefert

$p(v_0, v_i)$ den Vorgänger von v_i auf einem kürzesten Weg.

5.3 Floyd-Warshall

Der Algorithmus von Floyd-Warshall errechnet, im Gegensatz zu Bellman-Ford-Moore und Dijkstra, die kürzesten Wege zwischen allen Knoten. Diese könnte man natürlich auch errechnen, indem man Bellman-Ford-Moore oder Dijkstra n mal[26], jeweils mit anderem Startknoten startet. Als Ergebnis würde man n Distanz- und Vorgängelisten erhalten, die jeweils wieder n Elemente lang sind. Diese Listen könnten auch durch eine $n \times n$ Distanz- und Vorgängermatrix dargestellt werden. Ergebnis des Floyd-Warshall Algorithmus ist dementsprechend eine Vorgängermatrix und eine entsprechende Distanzmatrix.

Idee [27] Gegeben sei ein Digraph D mit n Knoten in Form einer Adjazenzmatrix A. Die Knoten seien fortlaufend und lückenlos nummeriert. Der Algorithmus errechnet nun aus der Adjazenzmatrix A in n Schritten die Distanzmatrix A_n. Dabei erhält man nach jedem Schritt j mit $1 \leq j \leq n$ eine Distanzmatrix A_j. Diese Matrix hat die Eigenschaft, dass der Eintrag an der Stelle (i, k) die Länge des kürzesten Weges von i nach k enthält, der nur die Knoten aus der Menge $\{1, \ldots, j\}$ benutzt. Die Matrix A_0 enthält die Längen der Kanten und ist gleich der Adjazenzmatrix A. Um uns den Übergang von A_{j-1} nach Aj zu veranschaulichen, wollen wir uns noch mal aufzeigen, wie der kürzeste Weg W in der Adjazenzmatrix A_j von Knoten i nach Knoten k aussieht, der nur die Knoten $\{1, \ldots, j\}$ verwendet:

1. W verwendet nicht den Knoten j. Dann verwendet W nur Knoten aus $\{1, \ldots, j-1\} \Rightarrow$ der Eintrag (i, j) von A_{j-1} enthält schon die Länge von W.

2. W verwendet den Knoten j. Da der Graph keine negativen Zyklen hat, kommt der Knoten j auf dem Weg W nur einmal vor. Der Weg wird nun in zwei Teile W_1 und W_2 aufgeteilt und zwar so, dass der Zielknoten aus W_1 und der Startknoten aus W_2 ist. Dann gilt: Die Länge von W ist gleich der Länge von W_1 plus der Länge von W_2. W_1 und W_2 sind kürzeste Wege. Beide Wege verwenden nur Knoten aus $\{1, \ldots, j-1\}$. Die Längen ergeben sich also aus der Matrix A_{j-1}.

[26] so oft, wie Knoten im Graphen vorhanden sind

[27] Siehe Volker Turau [6] Seite 269

Der Übergang von A_{j-1} nach A_j ergibt sich also aus $A_j(i,k) = Minimum(A_{j-1}(i,k), A_{j-1}(i,j)+A_{j-1}(j,k))$. Nach n Schritten ist der Algorithmus durchlaufen und liefert uns die Matrix A_n. Diese enthält die Längen der kürzesten Wege bzgl. des Problems aus Klasse 3.

Es fehlt noch die Vorgängermatrix, wenn man die Kanten der kürzesten Wege erhalten will. Dazu initialisieren wir die Vorgängermatrix aus der Adjazenzmatrix, indem wir zu jeder Kante $e_{x,y}$ des Graphen den Matrixeintrag (x,y) auf x setzen (x ist Vorgänger von y). Wird nun die Distanzmatrix geändert, und zwar nur in dem Fall, dass $A_j(i,k) = Minimum(A_{j-1}(i,k), A_{j-1}(i,j) + A_{j-1}(j,k)) = A_{j-1}(i,j) + A_{j-1}(j,k)$ gilt, dann setzen wir den Eintrag (i,k) der Vorgängermatrix auf den Inhalt des Eintrags (j,k) der Vorgängermatrix.

Floyd-Warshall in Pseudocode Vorbedingungen: $\{c(e_{v,w}) \geq 0, \forall e_{v,w} \in E\}$. Die Knoten sind fortlaufend durchnummeriert. Der Graph ist als Adjazenzliste gegeben[28].

1. alg FloydWarshall();
2. $\forall v_i \neq v_j$ do
3. $\quad k(v_i, v_j) = Integer.MAX\_VALUE$;
4. $\quad v(i,k) = Integer.MAX\_VALUE$;
5. od;
6. $\forall v_i = v_j$ do
7. $\quad k(v_i, v_j) = 0$
8. $\quad v(v_i, v_j) = v_i$;
9. od
10. $\forall v \in V$ do
11. $\quad \forall w \in$Adlist(v) do
12. $\quad\quad$ setze $k(v,w) = c(e_{v,w})$
13. $\quad\quad$ setze $v(v,w) = v$
14. $\quad$ od
15. od
16. for $j = 0$ to n do

[28]Die Adjazenzmatrix wird erst noch aufgebaut (Zeilen 10-15).

17. for $i = 0$ to n do
18. for $k = 0$ to n do
19. if $k(i,j) + c(e_{j,k}) < k(i,k)$ then
20. $k(i,k) = k(i,j) + c(e_{j,k})$;
21. $v(i,k) = v(j,k)$;
22. fi;
23. od;
24. od;
25. od;

Ein Eintrag der Vorgängermatrix wird durch $v(x,y)$ und die Distanzmatrix durch $k(x,y)$ dargestellt, wobei gilt: $0 \leq x,y \leq n$. Beide Matrizen werden zunächst initialisiert (Zeile 2 bis 9). Die Einträge auf den Matrizendiagonalen werden, in Zeile 6 bis 9, entsprechend gesetzt. In Zeile 10 bis 15 werden die Kanten des Graphen eingetragen. Anschließend wird die Distanzmatrix, nach dem schon erwähnten Muster, durchlaufen und dabei die Einträge der einzelnen Matrizen verändert (Zeile 16-25). Endergebnis sind zwei Matrizen, die Distanzmatrix und die Vorgängermatrix. Der Pseudocode ist aus dem Buch von Volker Turau[6] (Seite 270) übernommen, wobei wir die Notation aus dem Pascal ähnlichem Stil des Buches in einen allgemeineren Stil geändert und den Aufbau der Adjazenzmatrix eingefügt haben.

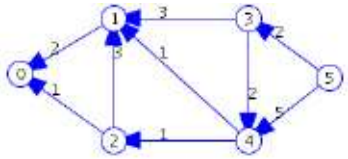
Laufzeit Die Laufzeit des Algorithmus ist einfach festzustellen. Die geschachtelten *For* Schleifen haben die Laufzeit $O(n^3)$. Alle anderen Operationen ergeben zusammen $O(n+m)$. Daraus ergibt sich eine Gesamtkomplexität von $O(n^3 + n + m) = O(n^3)$. Hier wird deutlich, dass die Laufzeit des Algorithmus abhängig von der Anzahl der Knoten in dem Graphen ist.

Negativer Zyklus An dem Ergebnis des Floyd Warshall Algorithmus kann man erkennen, ob ein Graph negative Zyklen hat, die den kürzesten Weg zwischen zwei Knoten immer wieder verkürzen würden, je öfter sie im kürzesten Weg eingebunden werden. Ein gewichteter Digraph hat dann nach dem Durchlauf des Floyd-Warshall Algorithmus einen negativen Zyklus, wenn es einen Knoten i gibt mit: $k^m(i,i) < 0$ für $m, i \in 1, 2, \ldots, n$.

Beispiel: Ein Startknoten muss nicht weiter bestimmt werden, da der Algorithmus die kürzesten Wege zwischen allen Knoten errechnet.

Adjazenzliste:

0:		
1:	0,2	
2:	0,1	1,3
3:	1,3	4,2
4:	1,1	2,1
5:	3,2	4,5



Adjazenzmatrix:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 2 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 5 & 0 \end{pmatrix}$$

Auch der Floyd-Warshall Algorithmus findet den kürzesten Weg zwischen dem Knoten 5 und 0 mit $P = (5, 3, 4, 2, 0)$ (siehe auch 4.3).

1 Schritt: (Initialisierung)
Als platzsparende Darstellung werden hier Distanzmatrix und Vorgängermatrix nebeneinander geschrieben[29].

Knoten	0	1	2	3	4	5	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
1	2	0	∞	∞	∞	∞	1	1	∞	∞	∞	∞
2	1	3	0	∞	∞	∞	2	2	2	∞	∞	∞
3	∞	3	∞	0	2	∞	∞	3	∞	3	3	∞
4	∞	1	1	∞	0	∞	∞	4	4	∞	4	∞
5	∞	∞	∞	2	5	0	∞	∞	∞	5	5	5

2 Schritt: untersuche die erste Zeile 1 der Matrix (alle Kanten von Knoten 0 ausgehend). Da hier keine Kanten von Knoten 0 ausgehen, werden hier auch keine Änderungen vorgenommen.

Knoten	0	1	2	3	4	5	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
1	2	0	∞	∞	∞	∞	1	1	∞	∞	∞	∞
2	1	3	0	∞	∞	∞	2	2	2	∞	∞	∞
3	∞	3	∞	0	2	∞	∞	3	∞	3	3	∞
4	∞	1	1	∞	0	∞	∞	4	4	∞	4	∞
5	∞	∞	∞	2	5	0	∞	∞	∞	5	5	5

3 ...

[29] Distanzmatrix links, Vorgängermatrix rechts

9 Schritt: Eine sichtbare Veränderung an den Matrizen tritt z.B. nach dem neunten Schritt auf. Da wir für j=1 und i=3 in einem zweiten Durchlauf die Kante $(i,k) = (3,0) = \infty$ mit der Kante $(i,j) + (j,k) = (3,1) + (1,0) = 3 + 2 = 5$ vergleichen. Dies führt zu einer Änderung der Matrizen an der Stelle (i,k) auf den Wert 5.

Knoten	0	1	2	3	4	5	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
1	2	0	∞	∞	∞	∞	1	1	∞	∞	∞	∞
2	1	3	0	∞	∞	∞	2	2	2	∞	∞	∞
3	5	3	∞	0	2	∞	1	3	∞	3	3	∞
4	3	1	1	∞	0	∞	1	4	4	∞	4	∞
5	∞	∞	∞	2	5	0	∞	∞	∞	5	5	5

letzter Schritt: Matrizen nach kompletten Durchlauf durch den Algorithmus

Knoten	0	1	2	3	4	5	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞
1	2	0	∞	∞	∞	∞	1	1	∞	∞	∞	∞
2	1	3	0	∞	∞	∞	2	2	2	∞	∞	∞
3	4	3	3	0	2	∞	2	3	4	3	3	∞
4	2	1	1	∞	0	∞	2	4	4	∞	4	∞
5	6	5	5	2	4	0	2	3	4	5	3	5

Soll nun der kürzeste Weg zwischen zwei Knoten ausgegeben werden, dann muss wie folgt verfahren werden.

5.4 weitere Algorithmen

Die bisher besprochenen Algorithmen sind in unserem Programm *YAV* implementiert. Jedoch gibt es noch andere Algorithmen. Eine Variante des Dijkstra Algorithmus ist der im Bereich der Künstlichen Intelligenz angesiedelte A^* Algorithmus [39]. Eine Übersicht über Algorithmen zur Berechnung von kürzesten Wegen ist in [35] und in [33] zu finden. Bücher, die speziellere Aspekte der Graphentheorie behandeln sind: [32] (Distanzprobleme) und [38] (kürzeste Wege in planare Graphen). Natürlich ist auch weitergehende Theorie in den von uns benutzten Büchern [4], [3] und [6] zu finden.

6 Übersicht

6.1 Datenstrukturen

Adjazenzliste	Darstellung von Graphen in Form von Kantenlisten, die in Bezug zu einem Knotenschlüssel stehen
Adjazenzmatrix	Darstellung von Graphen in Form einer Matrix, deren Dimension gleich der Anzahl von Knoten in dem Graphen ist. Die Knoten müssen durchlaufend nummeriert sein.
Fibonacci Heap (Fredman und Tarjan 1986)	Sehr schnelle Datenstruktur, die Schlüssel heapgeordnet verwaltet. In unserem Fall ideal bei großen Datenmengen in Verbindung mit dem Dijkstra Algorithmus.
Heap	Erlaubt uns hier Knoten in Beziehung zu einem Schlüssel zu verwalten.
Skipliste (William Pugh 1991)	Schnelle Datenstruktur, die Elemente sortiert in einer Liste hält, wobei die Möglichkeit besteht über Elemente hinwegzuspringen. Sie kann in unserem Fall als Heap verwendet werden.
Vorgängerliste	Liste, die zu jedem Knoten den Vorgängerknoten auf dem kürzesten Weg zum Startknoten wiederspiegelt.
Vorgängermatrix	Matrix, die zu jedem Knoten die Vorgängerknoten auf den kürzesten Wegen wiedergibt.

6.2 Graphalgorithmen

Algorithmus	Jahr	Laufzeit	Problemklasse	Seite
Bellman-Ford-Moore	1956	$O(nm)$	2	18
Dijkstra	1959	$O(nlogn)$	2	22
Floyd-Warshall	1962	$O(n^3)$	3	26

Teil III
Analyse, Entwurf und Implementierung

Um nun den Ablauf der Algorithmen an einem Graphen aufzuzeigen benötigen wir ein Programm, welches eine einfache, intuitive Oberfläche bietet und die einzelnen Algorithmenschritte aufzeigt. Also haben wir uns dazu entschlossen ein solches Werkzeug komplett neu zu entwickeln.

7 Pflichtenheft

7.1 Ziel-Bestimmung

Das Programm soll eine visuelle Darstellung von Graphen ermöglichen. Anhand dieses Graphen und ggf. der Festlegung eines Start- und Zielknotens sollen die einzelnen Schritte der Graphalgorithmen zur Bestimmung kürzester Wege veranschaulicht werden.

7.1.1 Muss-Kriterien

M–I Interaktives Erstellen/ Bearbeiten eines Graphen,

M–II Visuelle Darstellung eines Graphen,

M–III Laden und Speichern eines Graphen,

M–IV Visualisierung des Bellman-Ford-Moore Algorithmus,

M–V Visualisierung des Dijkstra Algorithmus,

M–VI Visualisierung des Floyd-Warshall Algorithmus,

M–VII Integrierte Onlinehilfe

7.1.2 Kann-Kriterien

K–I Druckerausgabe,

K–II Postscriptausgabe,

K–III exportieren einer Graph-Darstellung, z.B. in eine Textdatei,

K–IV exportieren des Graphen als Bild, z.B. JPEG,

7.1.3 Abgrenzungskriterien

Es werden nur die oben angegebenen Algorithmen zur Bestimmung kürzester Wege in Graphen implementiert.

7.2 Einsatz

Das Programm soll Graph-Algorithmen (im Speziellen die ausgewählten Algorithmen) auf einfache Weise visualisieren und verdeutlichen.

7.2.1 Anwendungsgebiete

Es könnte z.B. in schulischen Einrichtungen zur Veranschaulichung der Algorithmen genutzt, oder über das Internet jedem Interessierten frei zugänglich gemacht werden.

7.2.2 Zielgruppen

Z–I Studenten,

Z–II Schüler,

Z–III Lehrer,

Z–IV Professoren,

Z–V jeder der sich mit Graphentheorie beschäftigen möchte, muss oder soll.

7.2.3 Betriebsbedingungen

Es wird eine interaktive Bedienung durch den Benutzer vorausgesetzt. Das Programm muss lokal vorhanden sein oder es muss ein Internet Zugang bestehen.

7.3 Umgebung

7.3.1 Software

Minimale Voraussetzung ist auf Seiten des Anwenders die Installation eines Betriebsystems, auf dem die Java Laufzeitumgebung 1.4.2 installiert ist. Das Programm sollte lokal auf dem Rechner zur Verfügung stehen.

Da es möglich ist auch über das Internet auf die Applikation zuzugreifen, kann die lokale Verfügbarkeit entfallen, wenn ein Internetzugang besteht und ein Browser installiert ist (z.B. Mozilla, Netscape, Opera, Konqueror, Internet Explorer...).

Das Test- und Entwicklungssystem besteht aus Windows XP, Internet Explorer, Mozilla, Java 1.4.2 SDK, sowie Knoppix 3.2 und SuSE Linux 9 mit dem Konqueror.

7.3.2 Hardware

Es wird keine bestimmte Hardware vorausgesetzt. Lediglich die Mindestanforderungen des Betriebsystems und der Java Laufzeitumgebung, müssen erfüllt sein. Für die Applikation wird ein Festplattenplatz von ca. 200kb benötigt, das komplette Paket inklusive Dokumentation und Hilfe benötigt insgesamt 4MB. Das Entwicklungssystem besteht aus einem PC mit Intel Celeron 300 MHz Prozessor, 192MB Speicher, 6G Festplatte. Weiterhin wurde es auf einem AMD 750 MHz Prozessor mit 640MB Speicher, 40G Festplatte getestet.

7.3.3 Orgware

Zu Überlegen ist, wo das Programm bei der Installation abgelegt und wo die erstellten Graphen gespeichert werden sollen. Vor dem Programmeinsatz oder während der Grapherstellung sollte der zu betrachtende Graph geplant werden.

7.4 Funktionalität

Zunächst muss der Benutzer einen Graphen „eingeben“, z.B. durch interaktives „Zusammenklicken“ des Graphen oder Laden eines bereits gespeicherten Graphen. Anschließend wird dieser graphisch dargestellt. Um nun die kürzesten Wege berechnen zu lassen muss ein Start- und auf Wunsch ein Zielknoten festgelegt werden. Die Algorithmen können dann entweder in einem Modus gestartet werden, der die einzelnen Algorithmusschritte aufzeigt, oder es wird nur das Ergebnis des Algorithmus angegeben. Es sollten also folgende Funktionalitäten zur Verfügung stehen:

F–I Eingabe und Löschen einzelner Kanten und Knoten mit Hilfe der Maus,

F–II Eingabe der Kantengewichte per Tastatur,

F–III interaktives Erstellen eines Graphen,

F–IV Auswählen des Modus für den Ablauf des Algorithmus (Ergebnis oder Berechnung mit Zwischenschritte),

F–V Ablauf des Algorithmus in dem gewählten Modus,

F–VI Ausgabe des Ergebnisses,

F–VII Speichern/ Laden des Graphen,

F–VIII Erstellung eines Beispielgraphens (eventuell durch Vorgabe einer gespeicherten Datei).

7.5 Daten

Es soll möglich sein den Graphen in einer Datei zu speichern und wieder einzulesen. Die Anbindung an eine Datenbank macht aufgrund der Datenmenge keinen Sinn. Zu erwarten ist eine maximale Datenmenge von 100 Knoten. Dadurch sind maximal 10 000 Kanten möglich. Jedoch wird diese Datenmenge von dem Anwender kaum erzeugt werden, da das Verwalten von 10 000 Kanten und 100 Knoten zu unübersichtlich wird. Zu erwarten sind Graphen bis ca. 20 oder 30 Knoten.

7.6 Leistung

Es werden keine besonderen Leistungen an das System gestellt. Natürlich darf die Ermittlung der kürzesten Wege nicht „Ewigkeiten“ dauern, dies wird jedoch durch die Laufzeit der Algorithmen[30] eingegrenzt.

7.7 Benutzeroberflächen

Das Programm soll als Applikation realisiert werden und im Internet herunterladbar sein[15]. Weiterhin soll es von einem Browser aus startbar sein und eine graphische Oberfläche mit Menü und Maus-Bedienung enthalten. Die allgemein üblichen Gestaltungsvorschriften sollten hier beachtet werden, z.B. Datei oder File Menü links auf der Menüleiste, das Hilfe Menü ganz rechts. Die Fensterelemente sollten einen Hinweistext (Tooltip) besitzen, usw...

7.8 Qualitätsziele

Q–I Das Programm sollte durch die Implementierung in Java eine hohe Portabilität besitzen.

Q–II Weiterhin soll es möglich sein andere Graph-Algorithmen in dem Programm einzubinden.

Q–III Die Benutzer-Freundlichkeit soll durch einfache, intuitive Bedienung ermöglicht werden.

Q–IV Um den Benutzer zu unterstützen soll es ein Hilfesystem, eine API Dokumentation, eine Klassendiagrammübersicht sowie die Sourcecode Dokumentation geben.

[30] siehe Algorithmen im Teil Graphentheorie)

8 Entwurf und Design

8.1 Geschäftsprozesse

Es sind zwei entscheidende Geschäftsprozesse erkennbar. Zum Einen *Graph erstellen/ bearbeiten* zum Anderen *kürzeste Wege ermitteln* (siehe Abbildung 9).

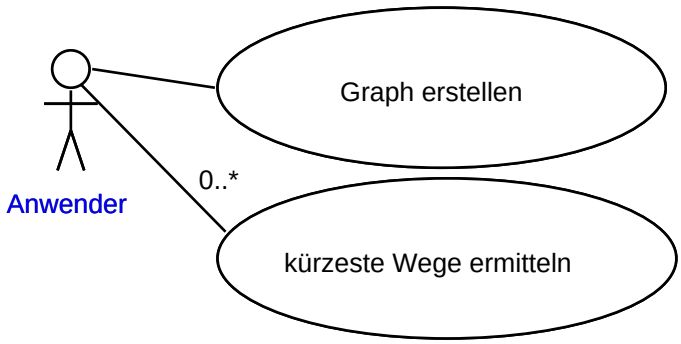


Abbildung 9: Geschäftsprozesse

Graph erstellen/ bearbeiten

- Ziel: Interaktives Erstellen und Bearbeiten eines Graphen, insbesondere das Erstellen von Knoten und Kanten.
- Kategorie: primär
- Vorbedingung: Es ist kein Graph vorhanden, oder ein Graph der bearbeitet werden soll ist schon geladen.
- Nachbedingung Erfolg: Der gewünschte Graph ist erstellt worden und steht zur Berechnung der kürzesten Wege bereit.
- Nachbedingung Fehlschlag: Es wird ein Fehlerhinweis ausgegeben, dass das Erstellen/ Bearbeiten (teilweise) fehlgeschlagen ist.
- Akteure: Der Anwender.
- Auslösendes Ereignis: Der Anwender hat das Programm gestartet, befindet sich im Bearbeitungsmodus und möchte einen Graphen erstellen/ bearbeiten.
- Beschreibung:
  - Erstellen von Knoten, interaktiv, mittels Mausklick.

  - Interaktives Erstellen von Kanten, mittels Mausklick auf den tail und head Knoten.
  - Eingeben der Kantengewichte mit Hilfe der Tastatur.
  - Interaktives Löschen eines Knotens.
  - Interaktives Löschen einer Kante durch Mausklick auf die Kante oder die beteiligten Knoten.
  - Ändern der Kantengewichte durch erneute Eingabe der Werte.
  - Drucken des Graphen auf einen Drucker.

kürzeste Wege ermitteln

- Ziel: Ermitteln, ob es einen kürzesten Weg gibt und wie lang dieser ist.
- Kategorie: primär
- Vorbedingung: Es existiert ein Graph, der die Vorbedingungen der Algorithmen erfüllt.
- Nachbedingung Erfolg: Ein oder mehrere kürzeste Wege sind errechnet worden und werden ausgegeben.
- Nachbedingung Fehlschlag: Es wird ein Fehlerhinweis ausgegeben, dass das Berechnen der kürzesten Wege (teilweise) fehlgeschlagen ist.
- Akteure: Der Anwender.
- Auslösendes Ereignis: Der Anwender hat das Programm gestartet und es existiert ein Graph, für den die kürzesten Wege berechnet werden sollen.
- Beschreibung:
  - Ggf. wählen der Start- und Zielknoten.
  - Auswahl des Algorithmus.
  - Durchlaufen des Algorithmus.
  - Ausführen eines Algorithmusschrittes.
  - Ausgeben des Ergebnisses nach Ablauf des Schrittes oder des ganzen Algorithmus.

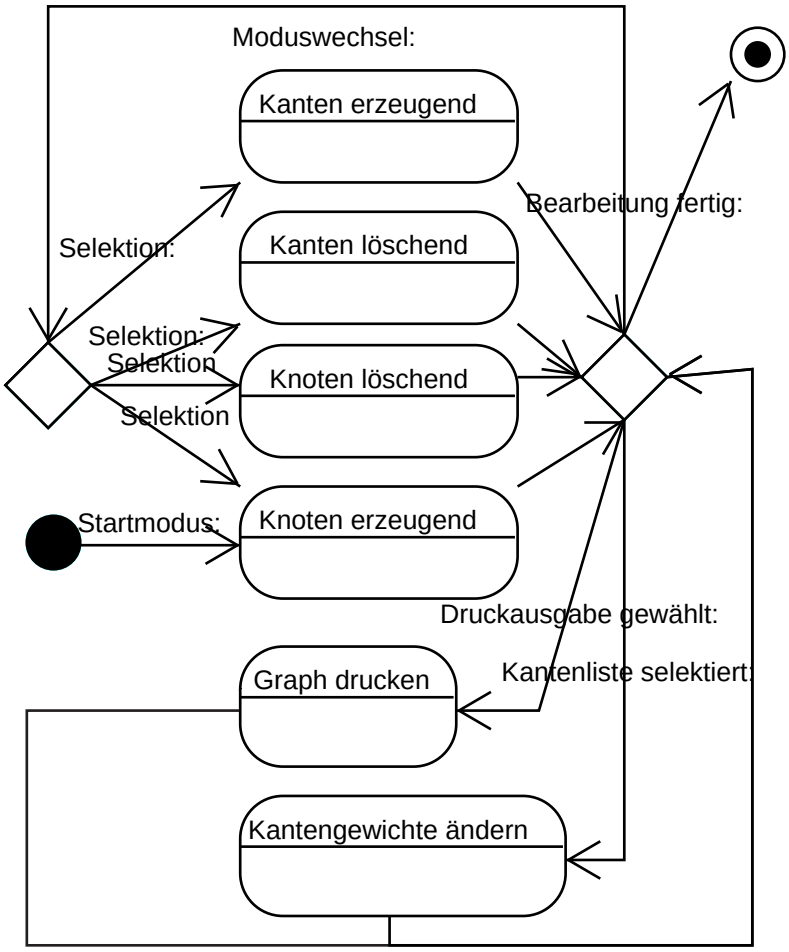


Abbildung 10: Graph erstellen

8.2 Zustandsdiagramme

Zu den Geschäftsprozess *Graph erstellen* wollen wir Zustandsdiagramme angeben. Dadurch soll deutlicher werden, wie ein Graph erstellt werden muss und was dabei Programmintern geschehen wird.

Abbildung 10 zeigt, dass es im Programm beim Erstellen des Graphen vier Hauptzustände (Modi) gibt. Einmal der Zeichenmodus, zum Anderen der Löschmodus, jeweils für Knoten und Kanten. Da das Erstellen und Löschen eines Knoten trivial ist, es wird die entsprechende Aktion durch einen Mausklick ausgelöst und das Ergebnis sofort umgesetzt, geben wir für diese Modi kein weiteres Zustandsdiagramm an. Das Erstellen und Löschen der Kante wird durch das Selektieren von Start- und Zielknoten vorgenommen (siehe Abbildung 11 und 12). In dem Moment, in dem der Zielknoten selektiert wird, wird die Kante, wenn zwischen den Knoten vorhanden,

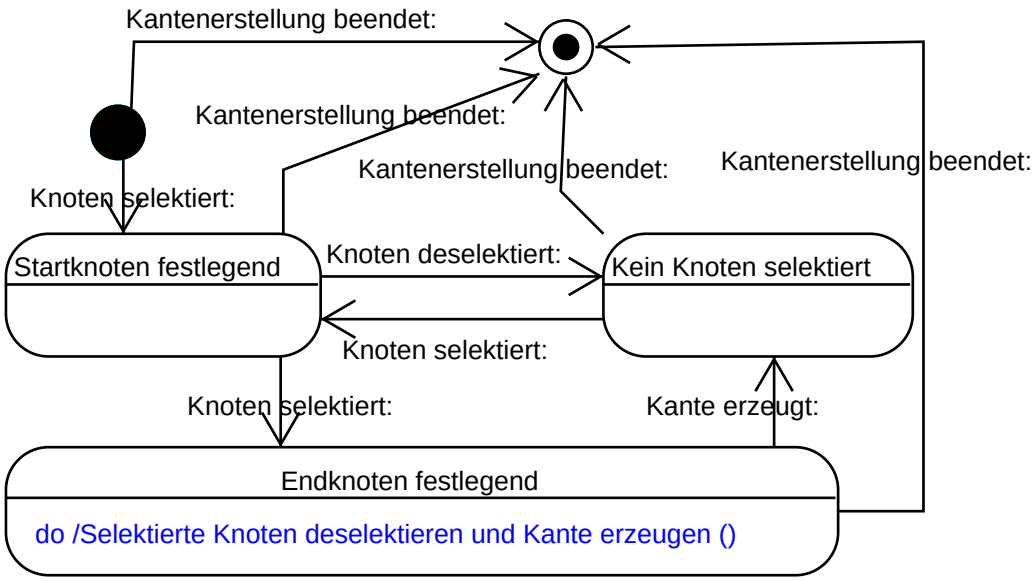


Abbildung 11: Kanten erzeugen

gelöscht und die selektierten Knoten deselektiert. Das Löschen der Kante wird auch auf Ebene der Kanten- und Adjazenzliste durchgeführt.

8.3 Klassendiagramme

Die Klassen *Heap*, *Node*, *Edge*, *GraphNode*,*GraphEdge* und *HeapElement* müssen alle die Eigenschaften der Interfaces *Comparable* und *Serializable* besitzen, damit die Sortierung über die Klasse *Collections* und die Dateiausgabe funktioniert. Das ist wichtig, da die Suche nach dem Heap- Minimum mittels Sortierung und anschließender Ausgabe des ersten Elements geschieht. Die Algorithmen sind Kindklassen von *Adjazenzliste* oder *Adjazenzmatrix*, je nachdem auf welcher Datenbasis sich der Algorithmus bezieht. Die Klasse *Heap* ist, wie schon erwähnt, aus der Klasse *Vector* abgeleitet. Diese Implementierung ist für den zu erwartenden Datenumfang ausreichend schnell. Für das Speichern der Daten werden Objekte der Klasse *Adjazenzliste* und *Graphnodelist* benötigt. Die *Graphnode*- Objekte enthalten die Koordinaten der Knoten.

Elementare Klassen sind also *Node* und *Edge*. Beide Klassen implementieren die Interfaces *Comparable* und *Serializable*, damit die Listen, auf Vektorbasis, aufgrund der *CompareTo* Methode die Sortierung nutzen können und die Objekte in Dateien gespeichert werden können. Die Methode *headCost2String* der Klasse Edge wird benötigt, um die Adjazenzliste als String darzustellen. Dazu wird z.B. von der Kante $(1, 2, 50)$ nur der Head Knoten und die Kostengröße ausgegeben z.B. $2, 50$.

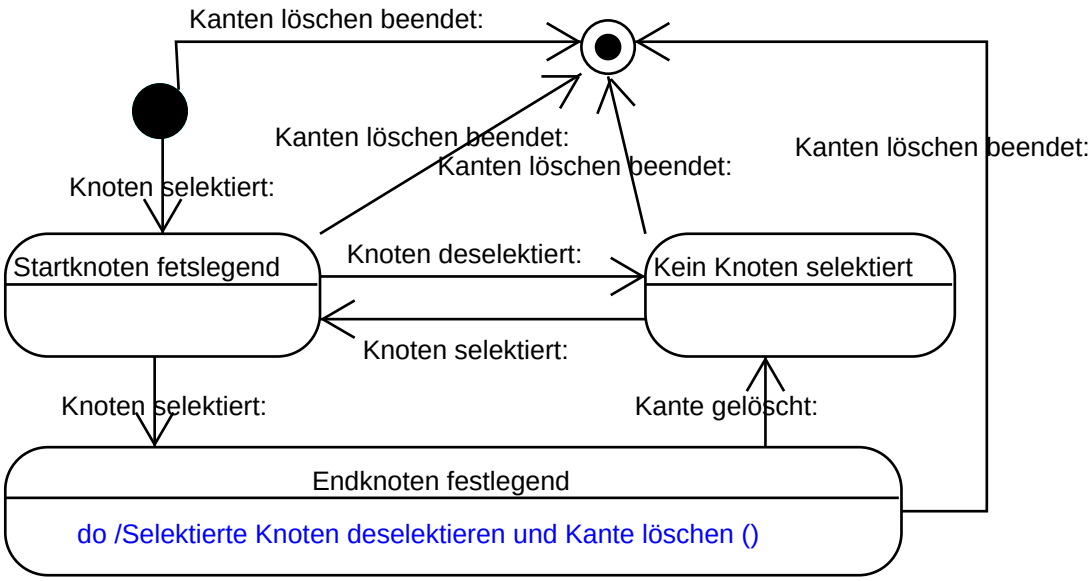


Abbildung 12: Kanten löschen

Die Klasse *Adjazenzliste* stellt die Basis für die Algorithmen *Dijkstra* und *Bellman-Ford-Moore* dar (siehe Abbildung 12, 12 und 12). Die Algorithmen durchlaufen die Adjazenzliste des Graphen und ermitteln so die kürzesten Wege. Wichtige Methoden sind *add*, *remove*, *contains*, *getEdgesWithTail*, die das Löschen, Einfügen, Enthaltensein durchführen und die Kantenliste zu einem Vektor ermitteln. Für die Datenhaltung sind *writeTo* und *readFrom* zuständig, die das Speichern auf Dateiebene realisieren.

GUI Klassen Für den Programmeinstieg verwenden wir die Klasse *YAV* (Methode Main). Dort werden das Menü, die Toolbar und das Panel mit den Reitern[31] aufgebaut. Für die einzelnen Reiter sind jeweils Objekte von bestimmten Klassen verantwortlich: *Graph* für die graphische Ansicht, *Daten* für die Daten Ansicht, *DijkstraViewer* , *BellmanFordMooreViewer* und *FloydWarshallViewer* für den jeweiligen Algorithmus. Die grundlegenden Methoden zur graphischen Darstellung des Graphen sowie der Kantenliste sind in der Klasse *Graph* vorhanden. Diese werden durch die Klassen *DijkstraViewer* , *BellmanFordMooreViewer* und *FloydWarshallViewer* mittels Vererbung übernommen und für die farbige Ausgabe der Kanten und Knoten erweitert. Dabei wird die Darstellung noch um die Vorgängerliste und den Heap, bzw. um die Vorgängermatrix und die Distanzmatrix, ergänzt. Die Tabellen werden durch die Klassen mit der Endung *Table* und *TableModel* implementiert, die meist Kindklassen von *JTable* und *AbstraktTableModel*

[31] JTabbedPane

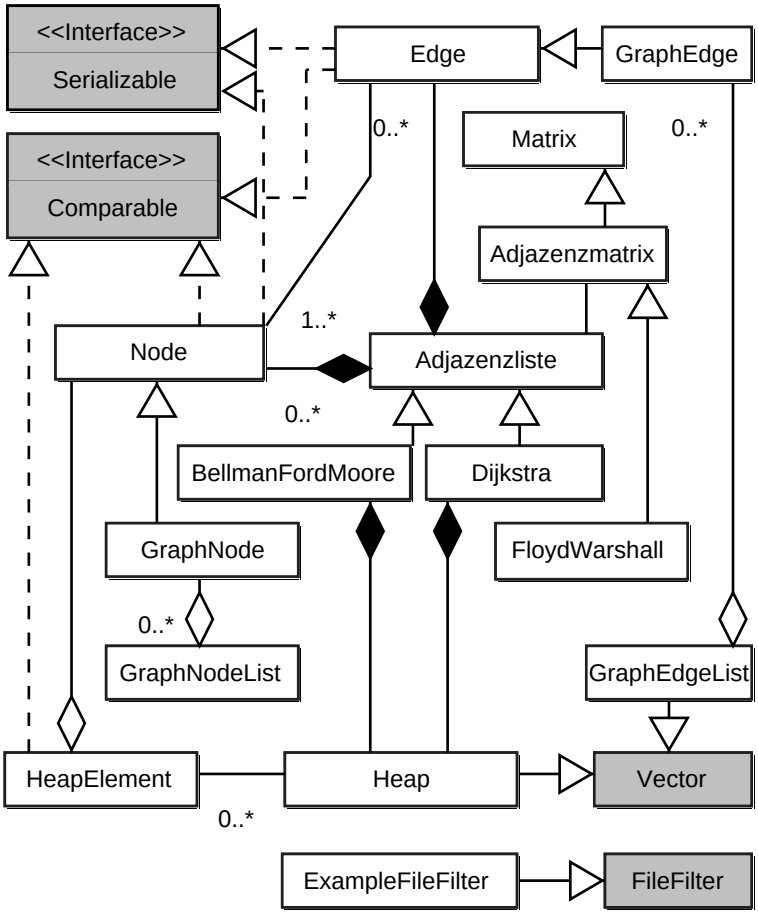


Abbildung 13: Übersicht der Klassen in UML Notation

sind. In Abbildung 14 haben wir eine GUI Klassenübersicht in UML Notation angegeben. Diese ist nicht ganz vollständig da z.B. alle Assoziationen fehlen und die Attribute und Operationen weggelassen wurden. Der Aufbau der Oberfläche sollte aber dennoch deutlich werden. Die Klassen des JDKs sind grau gefärbt.

Fremde Klassen In unsere Applikation wollen wir fremde Klassen einbinden. Zum Einen benötigen wir noch eine Klasse, die ähnlich eines Browsers HTML Code darstellt. Dazu bietet sich die Klasse *MetalworksHelp* aus der JDK Demo *Metalworks*. Wir haben lediglich die Klasse in *YAVHelp* umbenannt und den Pfad, an der das Einstiegsdokument zu finden ist angepasst. Zum Anderen wollen wir für das Dateihandling einen Filter erzeugen, der

nur **.YAV* Dateien anzeigt. Auch hier gibt es eine Klasse, die sich zum Einbinden anbietet: *ExampleFileFilter* von Jeff Dinkins aus der *SwingSet2* Demo des JDKs. Diese Klasse haben wir gar nicht geändert, sondern im Original übernommen.

8.4 Oberfläche

Es galt eine Oberfläche zu entwerfen, die einfach und leicht verständlich ist. Da fast jeder Malprogramme kennt, bei denen nach dem Start sofort eine Zeichenfläche angezeigt wird, ist es sicherlich sinnvoll die Zeichenfläche nach dem Start auch gleich anzubieten. Weiterhin enthalten die meisten Programme eine Toolbar, um die wichtigsten Programmfunktionen zu erreichen. Die einzelnen Symbole der Toolbar mussten auch selbsterklärend sein. Also wurde für die Knotenerzeugung ein Kreis und für die Kantenerzeugung ein Pfeil verwendet. Die Löschmodi ergeben sich aus dem durchgestrichene Erzeugungssymbol (durchgestrichener Kreis oder Pfeil). Der Startmodus wird durch ein Turnschuhsymbol und der Stepmodus durch einzelne Fußabdrücke dargestellt. Das Druckersymbol und das Fragezeichen für die Hilfe sind auch intuitiv verständlich. Um dem Benutzter Nachrichten oder Fehlermeldungen anzuzeigen, wollten wir nicht jedesmal einen Dialog aufrufen, der erst vom Anwender wieder weggeklickt werden muss. Deshalb werden dem Benutzer Meldungen und Fehler in einer Statusleiste am unteren Bildschirmrand angezeigt. Um nun die einzelnen Algorithmen unter der Oberfläche zu vereinen, haben wir diese auf einzelne Reiter (JTabbedPane) verteilt. Weiterhin sollten Menüeinträge zu finden sein. Fast jedes Programm hat ein Datei- und Hilfemenü. Weiterhin bot es sich an, da wir Java verwenden, ein Menü für das Look&Feel anzugeben. Jeder Algorithmus sollte zudem per Menu und Tastaturkürzel startbar sein. Die aktuelle Oberfläche ist in Abbildung 19 auf Seite 53 zu sehen.

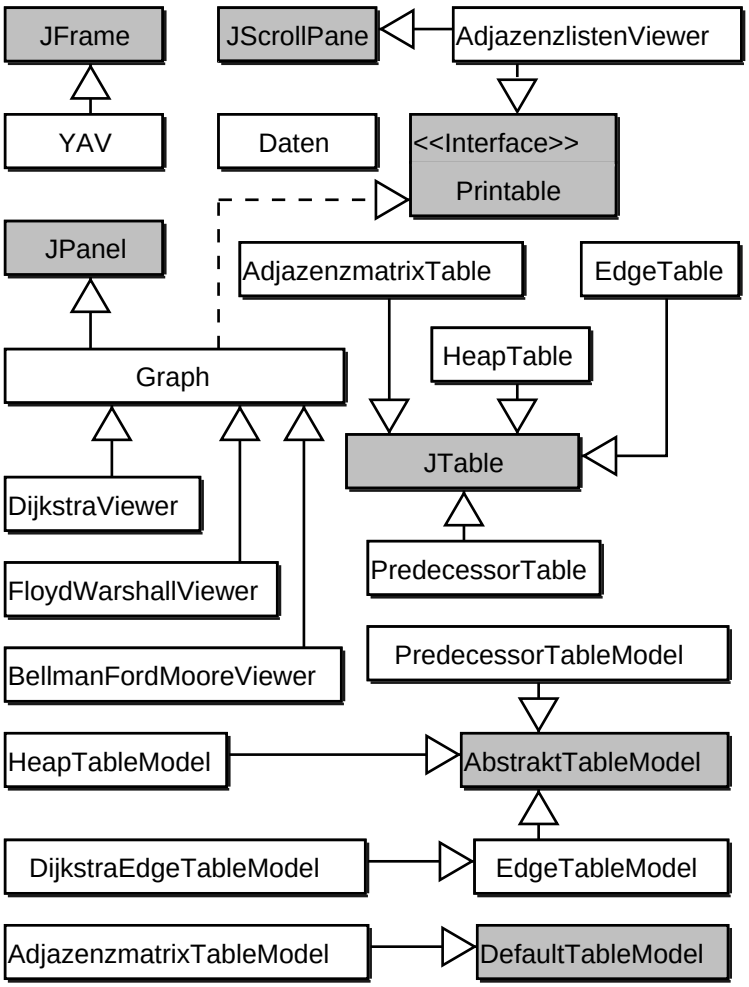


Abbildung 14: Übersicht der GUI Klassen in UML Notation

9 Entwicklung

9.1 Entwicklungsumgebung

Da das Endprodukt plattformunabhängig und mit einer modernen, objektorientierten Programmiersprache entwickelt werden sollte, kam eigentlich nur die Programmiersprache *Java*[19] von SUN in Betracht. Bei der Sprache *C++* hätten wir das Problem, dass für jedes Zielsystem ein kompilierter Code zu Verfügung stehen muss. Dieses stellt einen hohen Aufwand für Entwickler oder Anwender dar.

Als Entwicklungsumgebung wurde neben dem *JDK 1.4.2*, die *JCreator* IDE von Xinox Software[22] in der Freeware Version für Windows benutzt. Durch das JDK-nahe Arbeiten mit JCreator ist ein problemloses Kompilieren und Editieren auch mit anderen Tools, wie z.B. einen Editor, dem JDK, Eclipse[16] und ähnlichem möglich, natürlich auch auf anderen Plattformen, z.B. unter Linux. JCreator bindet nicht wie J++ oder Visual Cafe eigene Klassen ein, sondern stellt im Prinzip einen Editor mit JDK Integration dar. Weiterhin ist die IDE in Windows native Code verfügbar und somit schneller als z.B. Forte von SUN, so dass die Entwicklung auch auf einem Notebook mit Pentium II (300MHz) Prozessor möglich war. Für die Implementation wurde das JDK 1.4.2 vorausgesetzt, da die Druckausgabe in dieser Version, im Vergleich zur Vorherigen, verbessert wurde. Es sollte also mindestens die Java Laufzeitumgebung in der Version 1.4.2 benutzt werden.

Für die Erstellung der Installationsroutine unter Windows wurde *b1gSetup*[13]. Dieses Programm erlaubt mit einfachem Mitteln ein Installationssetup zuerstellen. Weiter Vorteil ist, dass das Programm in der Version 1.3.0.132 frei verfügbar ist.

Für die Dokumentation wurde die LaTeX Umgebung *Kile*[23] für Linux, verwendet. Diese ist frei erhältlich[32] und bietet eine sehr gute Unterstützung für LaTeX. LaTeX ist gerade für wissenschaftliche oder technische Dokumente hervorragend geeignet und hat sich an den Universitäten und Fachhochschulen bewährt. Für die UML Dokumentation wurde *ArgoUML*[12] benutzt. Dieses Tool hat z.B. gegenüber Rational Rose den Vorteil, dass es frei verfügbar ist und, durch Implementierung in Java, plattformunabhängig. Um eine Übersicht über alle Klassen zu generieren, wurde das Windows Programm *EssModel*[17] in der Version 2.2 benutzt. Dieses bietet eine einfache Möglichkeit aus vorhandenen Klassen eine HTML Klassenübersicht zu generieren und wird zur Zeit als Open Source veröffentlicht. Diese Klassenübersicht ist im Verzeichnis *klassen* des Programmpfads zu finden, oder kann über die Hilfe aufgerufen werden.

Graphiken wurden mit den Werkzeugen *Dia*[14] und *The Gimp*[18] erstellt. Der Quellcode wurde mit *Java2HTML*[20] (Version 4.0) nach HTML

[32] Kile wird unter der GNU GENERAL PUBLIC LICENSE, veröffentlicht

umgewandelt. Durch die Realisierung in Java ist auch dieses Tool plattformunabhängig und dadurch unter Linux und Windows nutzbar.

9.2 Vorgehensweise

Bei der Entwicklung sind wir nach der Top- Down Methode vorgegangen. Wir haben zunächst das Gesamtproblem dargestellt und anschließend die einzelnen Teilprobleme gelöst.

Der erste Prototyp stellte lediglich die Oberfläche dar und gab einen schwachen Eindruck vom späteren System. Es war die Technik der einzelnen TabPanes und der Toolbar zu erkennen. Einzelne Knoten konnten schon erzeugt werden. Der nächste Prototyp konnte zusätzlich Knoten und Kanten erzeugen. Nun hatten wir ein Rahmenprogramm und mussten noch die einzelnen Funktionalitäten implementieren. Als nächstes haben wir dann die Klassen *Node*, *Edge* und *Adjazenzliste* implementiert und unabhängig vom Programm getestet. Diese Testroutinen sind noch vereinzelt in den Klassen (*Main* Methoden) vorhanden. So konnte der nächste Prototyp schon die Adjazenzliste anzeigen, speichern und laden. Als nächstes wurden dann der Dijkstra Algorithmus implementiert und in die Oberfläche eingebunden. Weiterhin kam noch die Kantenverwaltung und die Druckausgabe hinzu. Im nächsten Schritt wurde das Hilfesystem eingebunden und die Menüstruktur verbessert. In einem der letzten Prototypen wurden noch die Algorithmen Bellman-Ford-Moore und Floyd-Warshall implementiert und zur Oberfläche hinzugefügt. Für die Implementation von Bellman-Ford-Moore waren wenig Anpassungen erforderlich, da der Algorithmus ähnlich Dijkstra funktioniert und so über Vererbung die grundlegenden Funktionalitäten übernommen werden konnten. Anders war es bei dem Algorithmus von Floyd-Warshall. Da die Datenbasis die Adjazenzmatrix ist, mussten erst neue Klassen geschaffen werden. Die Einbindung in die Oberfläche konnte jedoch ähnlich gestaltet werden. Neu hinzu kam in dem Zuge auch, unter dem Reiter *Daten*, die Anzeige der Adjazenzmatrix in Form einer Tabelle[33]. Jeder Algorithmus kann nun über das Menü gestartet werden. Der Quellcode ist über das Hilfesystem oder einen HTML-Browser[34] zu erreichen. Die Druckausgabe wurde auf die Methoden des JDK 1.4.2 umgestellt, da in dieser Version deutliche Verbesserungen bzgl. der Druckerausgabe zu finden sind.

9.3 Probleme bei der Entwicklung

Bei der Entwicklung eines Programms treten zwangsläufig Probleme auf. Die Wichtigsten sind hier kurz aufgeführt. Das erste Problem war, die Kanten darzustellen. Soweit die Kanten keine Gegenkanten haben, braucht ja nur eine Gerade zwischen den Punkten gezeichnet werden. Existiert zu der Kante

[33] JTable

[34] Unterverzeichnis *src* für den Quellcode und *hlp* für die Hilfe

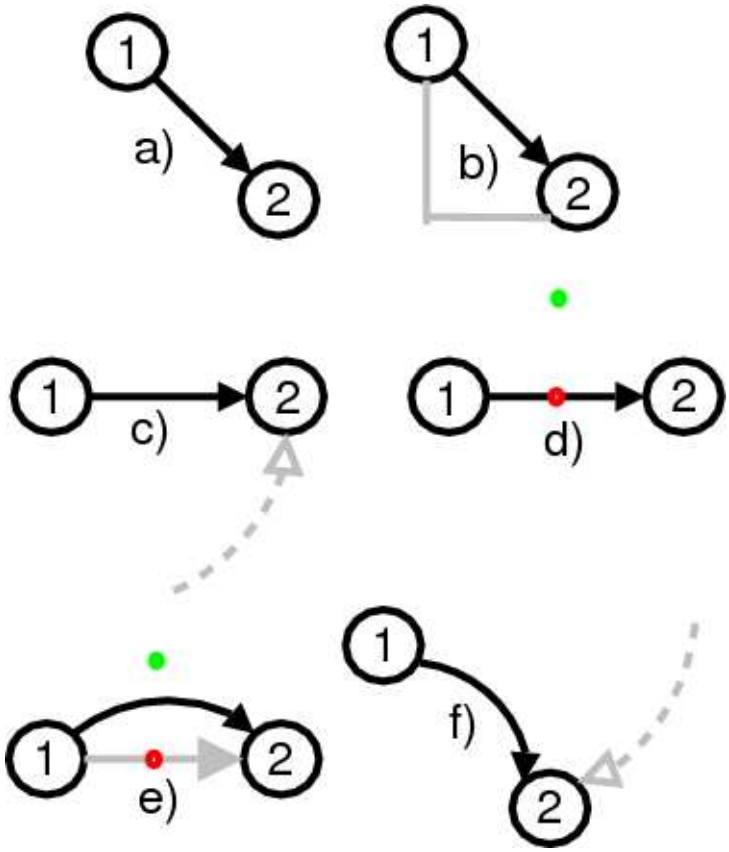


Abbildung 15: Problem der Kantendarstellung als Kurve

eine Gegenkante, dann würden die zwei Geraden sich aber überlappen. Also wird bei Vorhandensein einer Gegenkante eine Kurve[35] erzeugt und durch Angabe des Stützpunkts dafür gesorgt, dass die Kanten sich nicht überlappen (siehe hierzu auch Abbildung 15). Jetzt stellt sich natürlich die Frage, wie wir den Stützpunkt ermitteln. Dazu nehmen wir an, dass die Knoten horizontal zueinander liegen. Dies lässt sich durch eine einfache Transformation erledigen, indem die Länge der Strecke zwischen den beiden Punkten mit Pythagoras ermittelt (Abbildung 15 Schritt b) und auf den x- Wert eines Punktes aufaddiert wird (Abbildung 15 c). Nun lässt sich einfach ein Stützpunkt errechnen. Zunächst werden wir den Mittelpunkt auf der Gerade (Geradenlänge wurde schon durch Pythagoras ermittelt) zwischen den zwei Punkten bestimmen (Abbildung 15 Schritt d, roter Punkt). Danach addieren wir einfach einen festgelegten Wert hinzu und haben den Stützpunkt (grüner Punkt) für die Kurve, in dem Fall, dass die Knoten horizontal zueinander stehen (Abbildung 15 Schritt e). Nun müssen wir nur noch

[35] ein Objekt der Klasse QuadCurve2.Double

die soeben berechnete Kante um einen Punkt (Start- oder Zielknoten) rotieren, um das gewünschte Ergebnis zu erhalten (Abbildung 15 Schritt f). Der Rotationswinkel wird im Prinzip aufgrund folgender Formel errechnet und an die Methode *rotate* eines Objekts der Klasse *Graphics2d* übergeben: $rotation = \pi * \frac{3}{4} - atan(\frac{-(y_{head}-y_{tail})}{x_{head}-x_{tail}})$ mit $head = (x_{head}, y_{head})$ und $tail = (x_{tail}, y_{tail})$. Dabei müssen natürlich solche Fälle beachtet werden, dass der x-, oder y- Wert des head oder tail Punktes 0 sind, oder die Punkte horizontal, bzw. vertikal zueinander stehen. In den beiden letzteren Fällen muss keine Rotation vorgenommen werden.

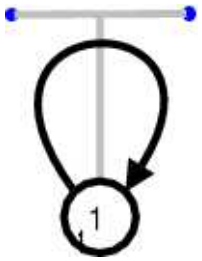


Abbildung 16: Problem der Darstellung einer Schlinge

Ein weiteres Problem in diesem Zusammenhang war natürlich, wie eine Schlinge dargestellt werden soll. Dazu haben wir einfach ein Objekt der Klasse *CubicCurve2D.Double* erzeugt, wobei der Start- und Zielknoten identisch ist. Die Angabe zweier Stützpunkte (blaue Punkte) ermöglicht dann das Aussehen einer Schlinge (siehe Abbildung 16). Die Stützpunkte können durch feste Vorgaben errechnet werden. Soll zu einem Knoten $v = (x_v, y_v)$ eine Schlinge gezeichnet werden, dann werden im Programm die Stützpunkte s_1 und s_2 durch folgende Formeln errechnet: $s_1 = (x_v - 40, y_v - 60)$ und $s_2 = (x_v + 40, y_v - 60)$.

Zusätzlich zur Kantendarstellung muss noch der Pfeil der Kante gezeichnet werden. Der Pfeil wird als ein Polygon gezeichnet, dessen Punkte wie folgt festgelegt und berechnet werden:

```
x[0]=head.x;x[1]=head.x+20*vorzeichen;x[2]=head.x+10*vorzeichen;
y[0]=head.y;y[1]=head.y+10*vorzeichen;y[2]=head.y+20*vorzeichen;
Polygon pfeil = new Polygon(x,y,3);
g2d.rotate( drehung+k,head.x,head.y);
```

Es besteht also aus drei Punkten und wird je nach *vorzeichen* in die Richtung *drehung* gedreht. Besitzt die Kante eine Gegenkante, dann muss weiter gedreht werden. Dies wird durch die Korrektur k bestimmt, die im Normalfall 0 und im Falle einer Gegenkante 0.3 ist.

Das Löschen der Kanten machte auch Probleme. Zuerst haben wir aufgrund der Selektion in der Kantentabelle eine Löschroutine implementiert, diese war aber wenig intuitiv, da das Erstellen der Kanten anders verläuft.

Danach haben wir genau wie beim Erstellen der Kanten die Selektion der zwei beteiligten Knoten und das Auswählen des Kantenlöschmodus als Anlass zum Löschen einer Kante genommen. Probleme gab es bei dem Versuch des Löschens durch direktes Anklicken der Kanten. Bei einer Kante, die keine Gegenkante enthält und somit durch eine Strecke dargestellt wird, gab es keine Probleme. Probleme gab es nur bei Kanten, die als *QuadCurve2D.Double* mittels Rotation erzeugt wurden. Durch das Erzeugen mittels Rotation konnte das Objekt die Koordinaten der Kurve nach der Rotation nicht erkennen. Deshalb konnte auch die Methode *intersects* der Klasse *QuadCurve2D.Double* nicht so ohne weiteres erfolgreich genutzt werden. Im Prinzip müsste die Rotation noch einmal nachvollzogen werden oder der Punkt, der überprüft werden soll, selbst in entgegengesetzter Richtung rotiert werden. Da das aber so ohne weiteres nicht machbar ist, bleibt die Implementierung dieser Löschroutine vorerst inaktiv (Methode *intersects* der Klasse *GraphEdge* muss dazu überarbeitet werden.).

Ein weiteres Problem war die Eingabe der Kosten einer Kante. Es konnte leicht passieren, dass der Eingabefokus nicht auf der editierbaren Zelle in der Kostenspalte lag, sondern z.B. in der Spalte *tail*. Der Anwender hat nun Kosten eingegeben und nicht bemerkt, dass die falsche Spalte editiert wurde. Um dies zu verhindern enthält nun die Klasse *EdgeTable* in den *KeyListener* Eventroutinen die Methode $setColumnSelectionInterval(2, 2)$. Diese bewirkt, dass der Eingabefokus bei den entsprechenden Events[36] auf die letzte Spalte springt und dort die eingegebenen Tasten in die Zelle geschrieben werden.

Wurde nun ein Graph erstellt und das Fenster verkleinert, wie sollte das Programm dann reagieren? Sollte es die graphischen Daten in den sichtbaren Bereich verschieben? Dies haben die ersten Programmversionen gemacht. Da jedoch bei einem anschließenden Vergrößern des Fensters die Knoten und Kanten nicht wieder zurückverschoben wurden, war diese Methode auf Dauer unbrauchbar. Lösung des Problems war ein Split- und ein ScrollPane, die nun das Fenster unterteilen und das Scrollen des sichtbaren Bereichs ermöglichen.

Die Hilfe sollte natürlich auf jeder Plattform zur Verfügung stehen und auch ohne Java aufrufbar sein. Dazu bot sich HTML[37] an. Wie sollte jedoch die Hilfe von Java aus aufgerufen und angezeigt werden? Die Suche in den Demo Quellcode des JDK zeigte, dass die Klasse $MetalworksHelp$, aus der Metalworksdemo, HTML in einem Fenster darstellen kann. Also brauchte diese Klasse nur in das eigene Programm integriert werden. Dies stellte sich sehr einfach heraus. Es wurde lediglich der Klassenname und der Hilfepfad (Unterverzeichnis *help* im Programmpfad) geändert.

Für das Speichern und Laden der Graphen werden drei Dateien benutzt.

[36] drücken einer Taste, loslassen einer Taste und tippen einer Taste

[37] hypertext markup language, siehe auch [27]

Die Datei mit der Endung *.yav* kann zur Versionskontrolle genutzt werden. Die Datei mit der Endung *.yal* enthält die Adjazenzliste ohne die graphischen Informationen. Die Datei *.ygd* enthält die graphischen Daten in Form der GraphNode Objekte. Damit der Anwender nicht andere statt *.yav* Dateien im Programm öffnet, benötigen wir einen Datei Filter. Dieser ist aus dem Beispiel *SwingSet2* des JDK entnommen (Klasse *ExampleFileFilter.java* und Übergabe an ein Objekt der Klasse *JFileChooser*).

Demselben Beispiel ist auch die Klasse *CodeViewer* entnommen, die aber nicht mehr zum Einsatz kommt. Sie diente zu Anfang dazu den Quellcode formattiert in einem eigenen Panel anzuzeigen. Dies ist nun nicht mehr nötig, da der Quellcode über die Hilfe im HTML Format zur Verfügung steht. Zur Formatierung des Quellcodes wurde das Tool Java2HTML[20] benutzt, welches Java Quellcode nach HTML, LaTeX, RTF[38] usw. konvertiert.

Unabhängig davon hatten wir mit ArgoUML Schwierigkeiten. So hat ArgoUML nach Anlegen eines Sequenzdiagramms die Daten nicht richtig gespeichert, bzw. einen Fehler beim Speichern ausgegeben, wenn wir auf denselben Dateinamen gespeichert haben. Dies haben wir durch das Speichern auf einen neuen Dateinamen umgangen. Dadurch konnten wir bei Problemen auf die vorher gespeicherte Datei zurückgreifen. Probleme gab es auch bei der Klassenübersicht. Dort haben wir im Klassendiagramm bei jeder Klasse die Attribute und die Operationen ausgeblendet. Nach dem Speichern und erneuten Laden wurden diese aber wieder angezeigt. Mit der Version 1.2 hatten wir das Problem nicht, doch lief diese Version im Bezug auf die Diagrammerstellung nicht so stabil. Also haben wir für die Klassendiagrammübersicht die Version 1.2 und für die restlichen Diagramme die Version 1.4 bzw. 1.4.1 benutzt. Alle drei Programmversionen sind auf der beigefügten CD zu finden und können direkt von CD aufgerufen werden. Die letzte ArgoUML Projektdatei zur Diplomarbeit ist im Verzeichnis *Ausarbeitung/ArgoUML/* zu finden.

9.4 Hardwarevoraussetzung

Das Programm wurde auf einem IBM-kompatiblen PC[39] mit x86-Prozessor (Pentium II 300MHz), 192 MB Hauptspeicher, 4MB Grafikkarte und einer 6G großen Festplatte erstellt und getestet. Weiterhin wurde das Programm auf einem System mit AMD Duron Prozessor 750MHz, 640MB Hauptspeicher, 40G Festplatte und AGP Grafikkarte (Elsa TNT2 64MB) getestet. Die folgenden Mindestanforderungen, insbesondere die an Geschwindigkeit und Speicher, sind mangels Testmöglichkeiten Schätzwerte. Wahrscheinlich sind auch niedrigere Werte möglich, wodurch das Programm aber zu stark verlangsamt würde.

[38] Rich Text Format
[39] Notebook Compaq Armada 7400

- IBM-kompatibler PC (mindestens 300 MHz Prozessor empfohlen)
- Monitor oder Monitorersatz (z.B. Fernsehr, Beamer...) mindestens jedoch eine Auflösung von 800x600 Pixel.
- VGA-Grafikkarte
- 64 MB Hauptspeicher (abhängig vom Betriebsystem, mindestens 192 MB für Windows XP und SuSE Linux empfohlen)
- 4 MB Festplattenkapazität für das Programm inklusive Hilfe, Sourcecode, Klassendiagramm und API Dokumentation. Das Programm an sich benötigt nur ca. 158k an Plattenspeicher.
- Zusätzlich wird natürlich Festplattenplatz für das Betriebsystem und die Laufzeitumgebung, sowie den verwendeten Programmen benötigt.
- Als Eingabegeräte werden Maus (oder Maus-Emulation) und Tastatur benötigt.

9.5 Softwarevoraussetzungen

Minimale Voraussetzung ist auf Seiten des Anwenders die Installation eines Betriebsystems, auf dem die Java Laufzeitumgebung 1.4.2 oder höher installiert (JDK 1.4.2 auf der CD im Ordner *Zusatz/JDK*) ist. Das Programm YAV muss lokal auf dem Rechner zur Verfügung stehen. Zusätzlich können die Java Entwicklungsumgebung, ein Web Browser, JCreator, Java2HTML, LaTeX, Kile und andere Hilfsmittel installiert sein.

9.6 Installation

Es gibt mehrere Möglichkeiten das Programm zu installieren:

- eine Installationsdatei „YAVsetup.exe“ für Windows,
- ein selbstextrahierendes Archiv „YAV.exe“, im RAR Format[26], nur unter Windows ausführbar,
- ein ZIP- Archiv, „YAV.zip“, im ZIP Format[30]
- das Verzeichnis „Java“ enthält die Projektdateien für JCreator und im Unterverzeichnis „YAV“ den Quellcode („.java“ Dateien) und kompilierten Zwischencode („.class“ Dateien).

Wird unter Windows die CD eingelegt, dann wird die Datei „YAVsetup.exe“ gestartet und es erscheint der Begrüßungsdialog (siehe Abbildung 17). Nun kann der Anwender sich durch Klick auf die *weiter*- Schaltfläche durch die einzelnen Dialoge nacheinander arbeiten. Folgende Dialoge werden angezeigt:

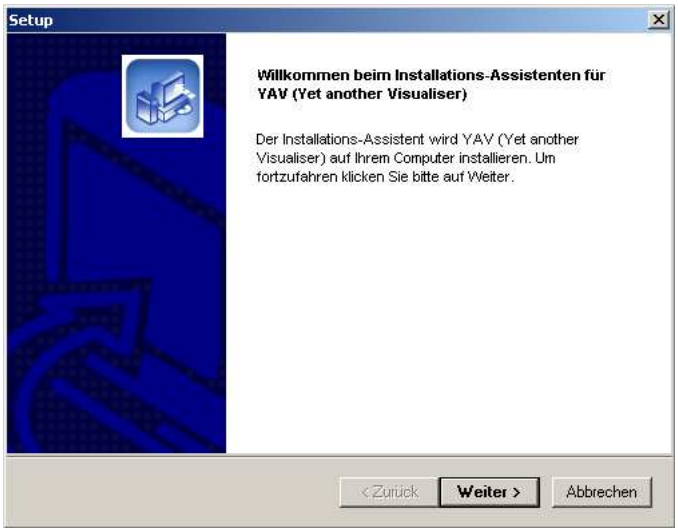


Abbildung 17: Startdialog des Setups

1. Willkommensbildschirm (Abbildung 17)
2. Lizenzbestimmungen (Anzeige und Annahme der Lizenzbedingungen)
3. Verzeichnisauswahl (Auswahl des Installationsortes)
4. Zusammenfassung (Anzeigen der selektierten Optionen)
5. Installation/ Fortschrittsanzeige (durchführen der Installation)
6. Installationsabschluss (Bestätigung, dass die Installation durchgeführt wurde)

Ergebnis der Installation ist, dass in dem *Start-* Menü eine Programmgruppe mit dem Namen *YAV* eingetragen ist. Diese Programmgruppe enthält Verknüpfungen zu dem Programm, der Hilfe, der API Dokumentation, der Lizenz, dem Source Code und der UML Übersicht.

Eine weitere Möglichkeit ist unter Windows die Datei „YAV.exe“ zu starten. Es erscheint ein Dialog zur Pfadauswahl (Abbildung 18). Nun muss nur noch der Pfad angegeben werden, in dem das Programm extrahiert werden soll. Geht der Anwender den anderen Weg über das ZIP- Archiv, dann muss er bei der Extraktion den selbstextrahierenden Pfad vorgeben. Dies ist abhängig vom verwendeten Archivierungsprogramm[40]. Alternativ dazu kann auch das Verzeichnis „Java“ auf die Festplatte kopiert werden. Möchten Sie den Zwischencode neu erzeugen, sollten Sie anschließend den Befehl *javac YAV* ausführen.

[40]Mögliche Archivierungsprogramme sind z.B. KArchiver, WinRar, InfoZIP...

Abbildung 18: Entpackdialog des selbstextrahierenden Archivs

Ergebnis aller Vorgehensweisen ist, dass das Programm als *jar* oder *class* Datei in dem Verzeichnis „YAV“ zur Verfügung steht[41]. Wenn die Java Laufzeitumgebung 1.4.2 installiert ist, kann das Programm von dem Installationspfad aus gestartet werden. Der Zwischencode der *class* Datei wird mit dem Befehl *java YAV* und die *jar* Datei mit *java -jar YAV.jar* gestartet. Alternativ kann das Programm auch von der CD aus gestartet werden. Dazu muss auf der CD das Verzeichnis *JAVA/YAV* gewählt und die Datei *YAV.class* mit dem Befehl *java YAV* gestartet werden. Unter manchen Betriebsystemen, z.B. unter Windows, ist es auch möglich eine *jar* Datei per Doppelklick mit der Maus zu starten.

Hinweis Bitte beachten Sie die Lizenzbestimmungen bzgl. der Installation von *YAV*. Wir haben uns entschlossen *YAV* unter der *GNU General Public License Version 2* zu veröffentlichen und hoffen dadurch einen hohen Verbreitungsgrad zu erzielen. Die Lizenz wird bei der Installation im Lizenzdialog angezeigt, oder ist im Programmverzeichnis von *YAV* zu finden (Dateien „license.txt“ und „lizenz.html“).

9.7 Beispiel zur Programmbedienung

Wir wollen an einem Beispiel die Programmbedienung veranschaulichen. Nach dem Programmstart bietet sich dem Anwender das Bild aus Abbildung 19.

[41] Bei der Installationsroutine unter Windows wird zusätzlich die Programmgruppe YAV mit entsprechenden Verknüpfungen angelegt.

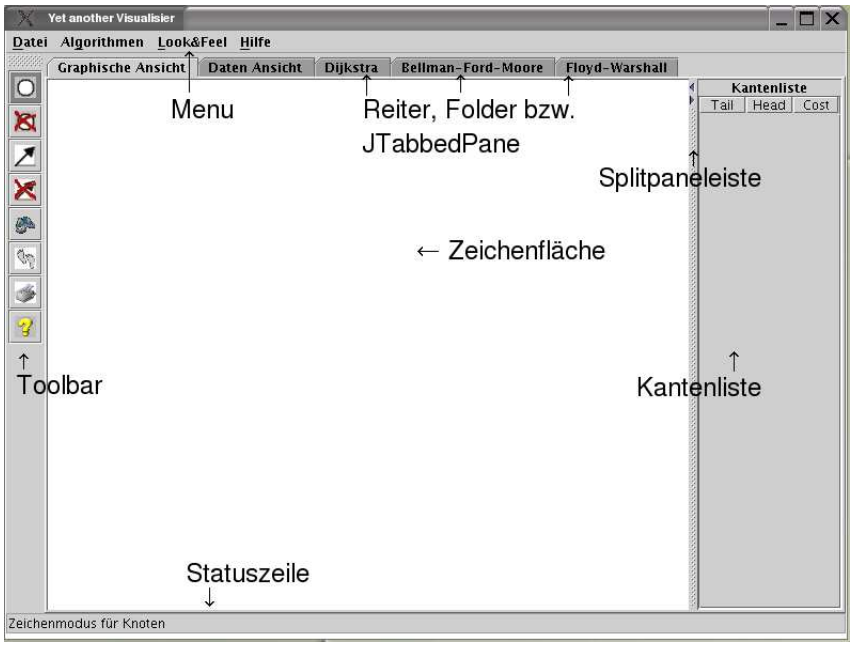


Abbildung 19: YAV Startbildschirm

Am linken Bildschirmrand befindet sich die Toolbar, die der Anwender auf Wunsch auch an einem anderen Fensterrand „einklinkt“ oder „schwebend“ über die Applikation legt. Der Anwender kann nun über die Toolbar die entsprechenden Modis anwählen. Folgende Modis sind wählbar (siehe Toolbar von oben nach unten in Abbildung 19):

- Zeichenmodus für Knoten (Kreissymbol)
- Löschmodus für Knoten (durchgestrichenes Kreissymbol)
- Zeichenmodus für Kanten (Pfeilsymbol)
- Löschmodus für Kanten (durchgestrichenes Pfeilsymbol)
- Startmodus für Algorithmen (Schuhsymbol)
- Stepmodus für Algorithmen (Fußsymbol)
- Druckerausgabe (Druckersymbol)
- Hilfeaufruf (Fragezeichen)

Weiterhin sind die Funktionen für Neu, Laden, Speichern, Drucken, Beenden, das Starten der einzelnen Algorithmen in dem Start- und Stepmodus, das Setzen des Look&Feels, sowie die Hilfe und der „About“- Dialog über Menüpunkte erreichbar. Die weiße Fläche stellt die Zeichenfläche dar.

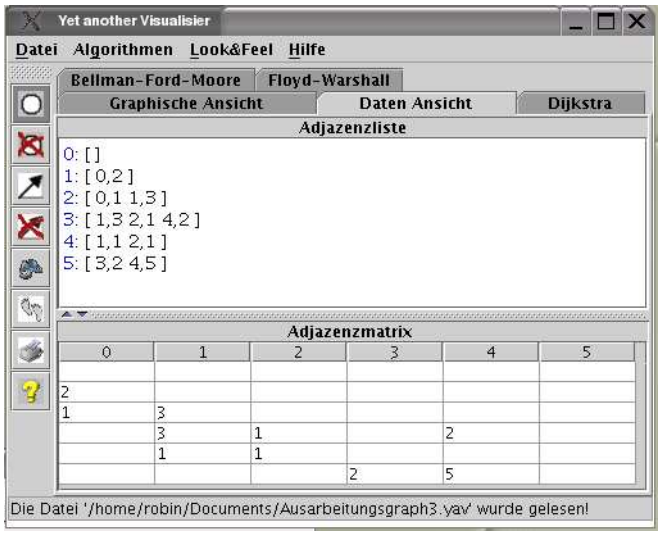


Abbildung 20: Datenansicht

Die Kantenliste wird am rechten Fensterrand ausgegeben. Nach dem Programmstart ist der Zeichenmodus für Knoten aktiviert. Der Anwender kann nun durch Mausklick auf der Zeichenfläche Knoten erzeugen. Knoten werden durch Drag and Drop verschoben. Sollen Kanten erzeugt werden, dann muss der Anwender in den Zeichenmodus für Kanten wechseln. Nun können Kanten erzeugt werden, indem zunächst der Startknoten und danach der Zielknoten angeklickt wird. Knoten und Kanten werden äquivalent zur Erstellung in dem jeweiligen Löschmodus entfernt. Dabei ist zu beachten, dass beim Löschen eines Knotens auch alle Kanten von und zu diesem Knoten gelöscht werden. Wird mehr Platz für die Zeichenfläche benötigt, dann kann die Splitpaneleiste verschoben, oder das Applikationsfenster vergrößert werden.

Nachdem der Anwender nun den Graphen erzeugt oder geladen hat, kann er sich die Adjazenzliste und -matrix in dem *Daten-* Reiter ansehen (siehe Abbildung 20).

Anschließend kann der Anwender zu einem Algorithmus wechseln, z.B. Dijkstra (siehe Abbildung 21). Falls gewünscht können Start- und Zielknoten gesetzt werden. Dazu selektiert man mit der Maus die entsprechenden Knoten. Es wird im Wechsel nun Startknoten (Rot) und Zielknoten (Grün) festgelegt. Sobald die gewünschte Selektion vorgenommen wurde, kann man den Algorithmus im Start- oder Stepmodus starten. Im Startmodus läuft der Algorithmus durch und zeigt das Endergebnis mittels gelber Kanten, der Tabelle *kürzester Weg* und der Vorgängerliste an. Im Stepmodus wird jeweils nur ein Algorithmusschritt ausgeführt und das Ergebnis des Schritts visualisiert. Ist der Algorithmus durchgelaufen, dann erscheint eine entsprechende Meldung in der Statusleiste. Dabei wird die benötigte Zeit ausgegeben. Die-

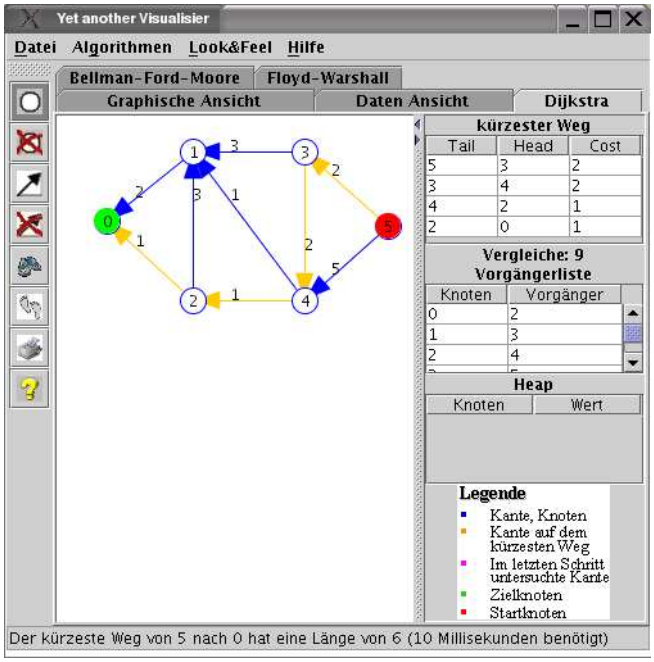


Abbildung 21: Dijkstra Algorithmus

se Zeit bezieht sich immer nur auf den zuletzt gemachten Algorithmusschritt (Stepmodus) oder auf den ganzen Durchlauf (Startmodus).

9.8 Deinstallation

Soll das Programm deinstalliert werden, steht bei Benutzung der Installationsroutine unter Windows ein Deinstallationsprogramm zur Verfügung, dass über die Software- Systemeinstellung aufgerufen werden kann (siehe Abbildung 22). Wurde die Installationsroutine nicht genutzt, dann reicht es aus den Programmpfad zu löschen, in dem das Programm installiert wurde.

Teil IV
Ausblick

Obwohl die verwendeten Graphalgorithmen schon vor ca. 40 Jahren und mehr erdacht wurden sind sie immer noch aktuell. In Verbindung mit neueren Datenstrukturen (Fibonacci Heap, Skipliste), schnelleren Computern mit mehr Speicher, ergeben die Graphalgorithmen immer bessere Laufzeiten.

Durch die Implementierung haben wir ein Tool, das auf einfache und intuitive Weise die vorgestellten Graphalgorithmen verdeutlicht und in Verbindung mit dieser Ausarbeitung, insbesondere dem Pseudocode, erklärt.

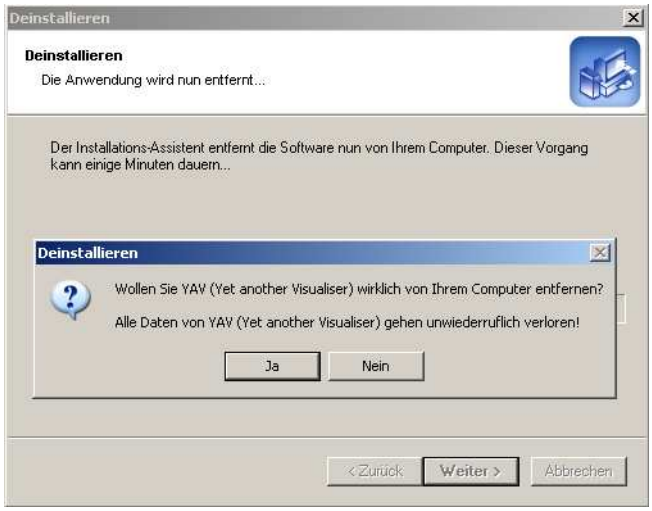


Abbildung 22: Deinstallation unter Windows

Wir hoffen unser Tool *YAV* findet einen großen Verbreitungsgrad und dass viele Leute daran gefallen haben.

10 Verbesserungs- und Erweiterungsmöglichkeiten

Die Ausgabe auf dem Drucker könnte durch das Drucken der Kantenliste und der Algorithmenergebnisse ergänzt werden. Weiterhin könnte man das Löschen der Kanten durch direktem Mausklick implementieren (siehe dazu den Abschnitt 9.3). Eine Undo Funktion, die den letzten Bearbeitungsschritt am Graphen rückgängig macht, wäre in manchen Situationen nötig. Auch eine Mehrfachselektion von Knoten, um diese gemeinsam verschieben zu können wäre sicherlich denkbar. Um bei der Grapherstellung die Übersicht zu behalten und das Layout der Graphen zu verbessern, könnte man die Kanten als Pfad realisieren, so dass der Anwender verschiebbare Zwischenknoten einfügen und löschen kann. Wird das Look& Feel des Programms geändert und/ oder die Toolbar verschoben, dann werden diese Einstellungen bei Programmende nicht gespeichert. Dies könnte man dadurch realisieren, dass allgemeine Programmeinstellungen in einer festgelegten Datei gespeichert werden. Das Sichern des Graphen kann auch durch ein einfaches Speichern erweitert werden, das einen geladenen Graphen unter dem schon angegebenen Namen speichert, ohne den Dateidialog aufzurufen.

Weiterhin könnte es vorteilhaft sein die Daten nicht nur auf einem Drucker auszugeben, sondern z.B. direkt als Postscript oder HTML, wobei die Ausgabe in HTML von jedem Algorithmusschritt ein Bild mit entsprechenden Erklärungen generieren könnte, so dass man die einzelnen Algorith-

musschritte erkennen kann.

Durch die Implementierung anderer Algorithmen könnte das Programm zusätzlich an Qualität gewinnen. So könnten z.B. Algorithmen implementiert werden, die einen Eulerschen Pfad oder einen Sperrflusses[42] errechnen. Die Implementation weiterer Datenstrukturen z.B. die Forward-Reverse Star Darstellung wäre auch eine sinnvolle Erweiterung. Als weiteres Feature könnten wir uns das automatische Ausrichten des Graphen, z.B. so dass Kanten sich möglichst nicht überlappen, vorstellen.

Eine einfachere Anbindung/ Erweiterung in Form von Plugins wäre sicherlich förderlich, um das Wachsen und die einfache Funktionserweiterung des Programms zu unterstützen. Der Export/ Import in ein anderes Format, wie z.B. VCG wäre genauso interessant, da dadurch der Datenaustausch zu anderen Programmen möglich wäre.

11 Weitere Programme

Im Internet wird man schnell fündig, wenn man nach Graphalgorithmen oder kürzeste Wege sucht. Einen kleinen Ausschnitt dieser Suche wollen wir kurz vorstellen.

Gato Weitere Programme, die sich mit den Erzeugen von Graphen und der Ausgabe der kürzesten Wege beschäftigen sind z.B. Gato, das auf Phyton basiert und Graphalgorithmen schrittweise visualisiert.

Leda LEDA (Library of Efficient Data Types and Algorithms) ist eine C++-Bibliothek die eine Vielzahl von effizienten Datentypen und Algorithmen enthält. Sie wurde ab Ende der 80er Jahre am Max-Planck-Institut für Informatik im Rahmen eines Forschungsprojekts entwickelt. Derzeit betreibt die Weiterentwicklung, Pflege und den Vertrieb von LEDA die Firma Algorithmic Solutions Software GmbH[9].

Webseiten mit Applets und Graphentheorie Eine in Englisch verfasste Seite, die ein Applet zur Visualisierung von Graphalgorithmen zur Verfügung stellt wird von Biliana Kaneva und Dominique Thiébaut[10] angeboten. Die Veranschaulichung des Dijkstra Algorithmus und eines Algorithmus zur Berechnung eines minimalen Spannbaums bietet H.W. Lang[11].

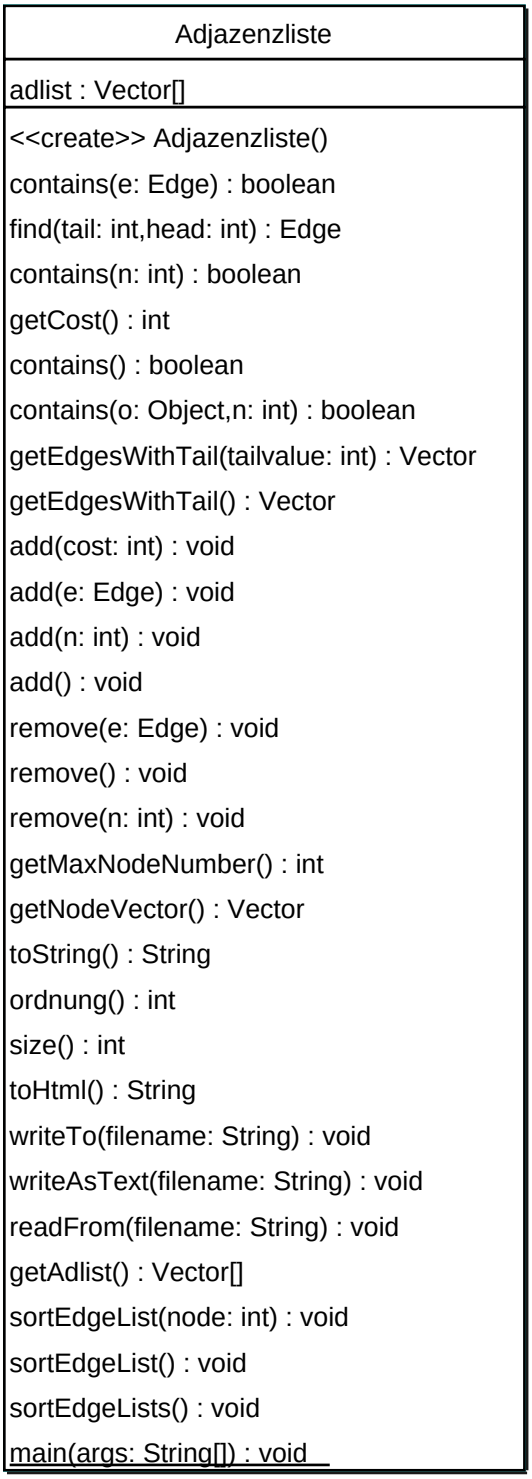
[42] Blocking flow

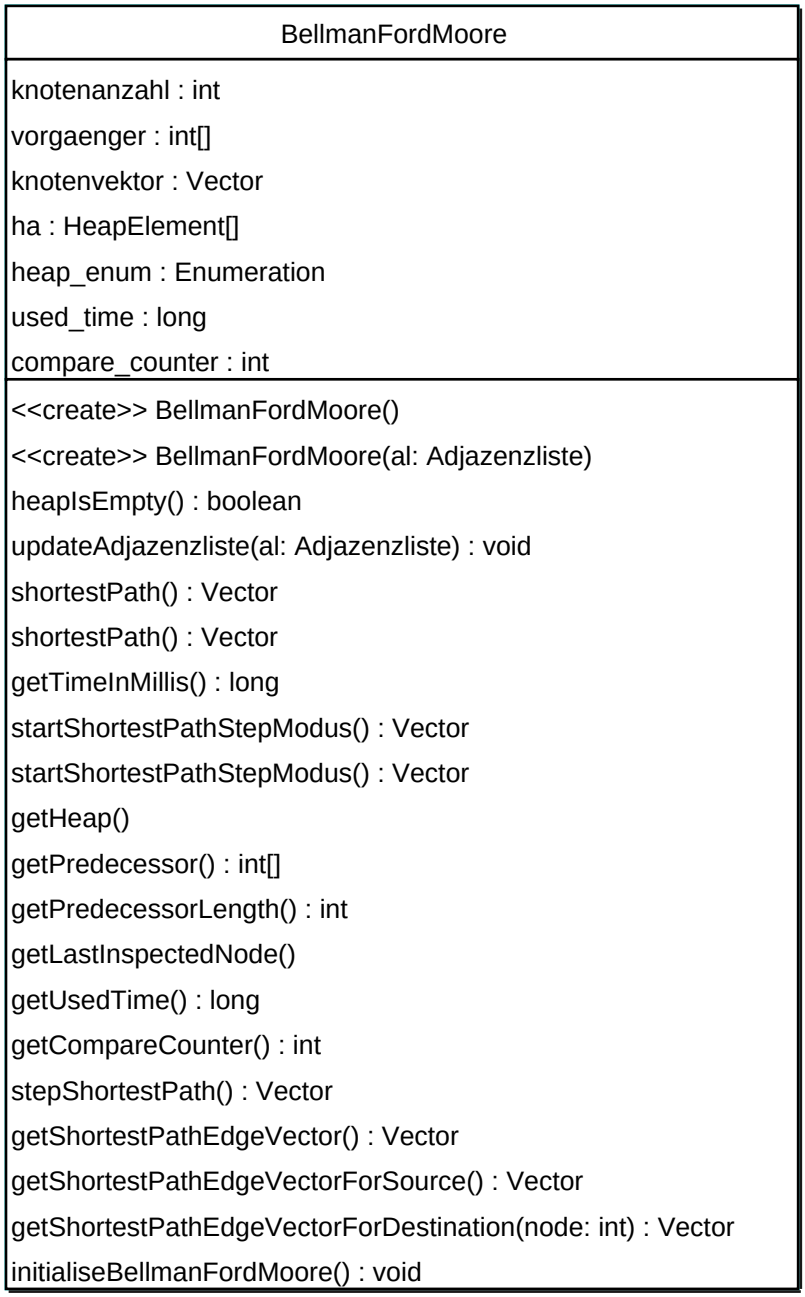
Teil V
Anhang

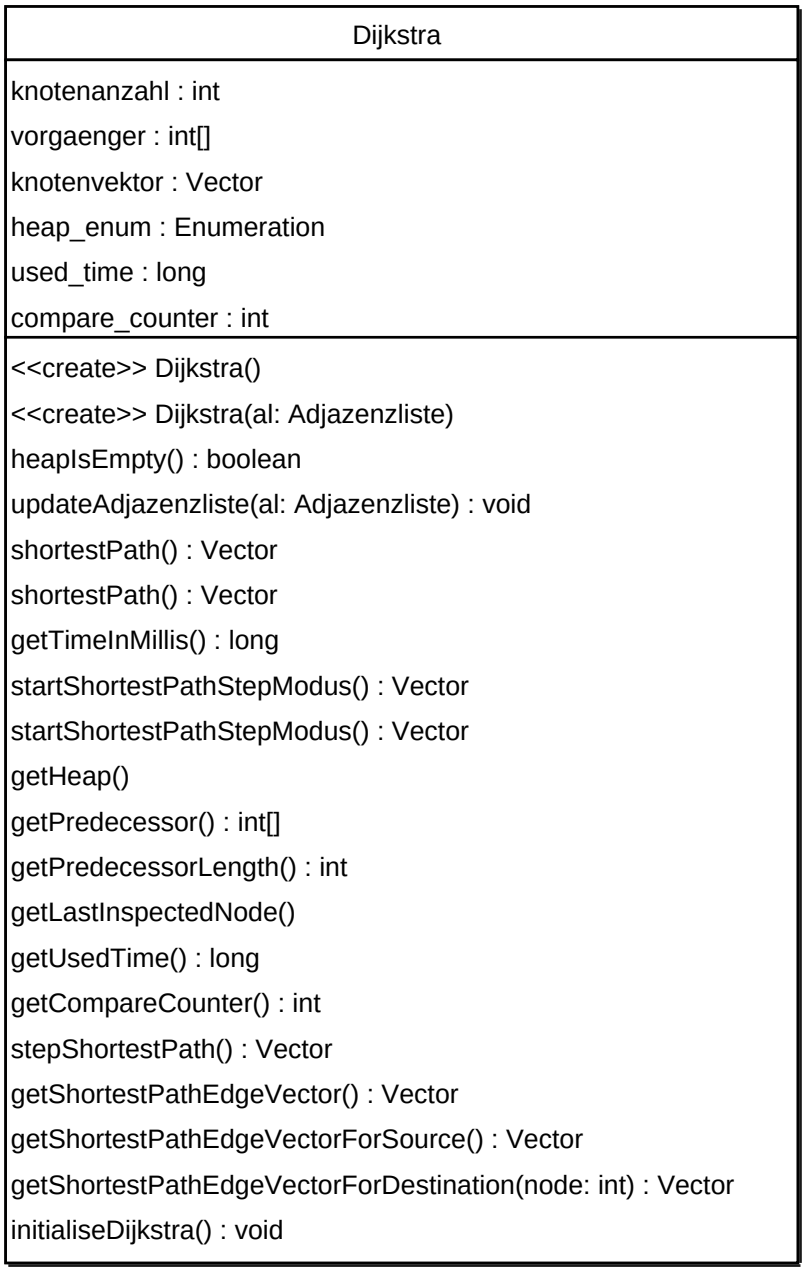
12 Anhang A ausgewählte Klassendiagramme

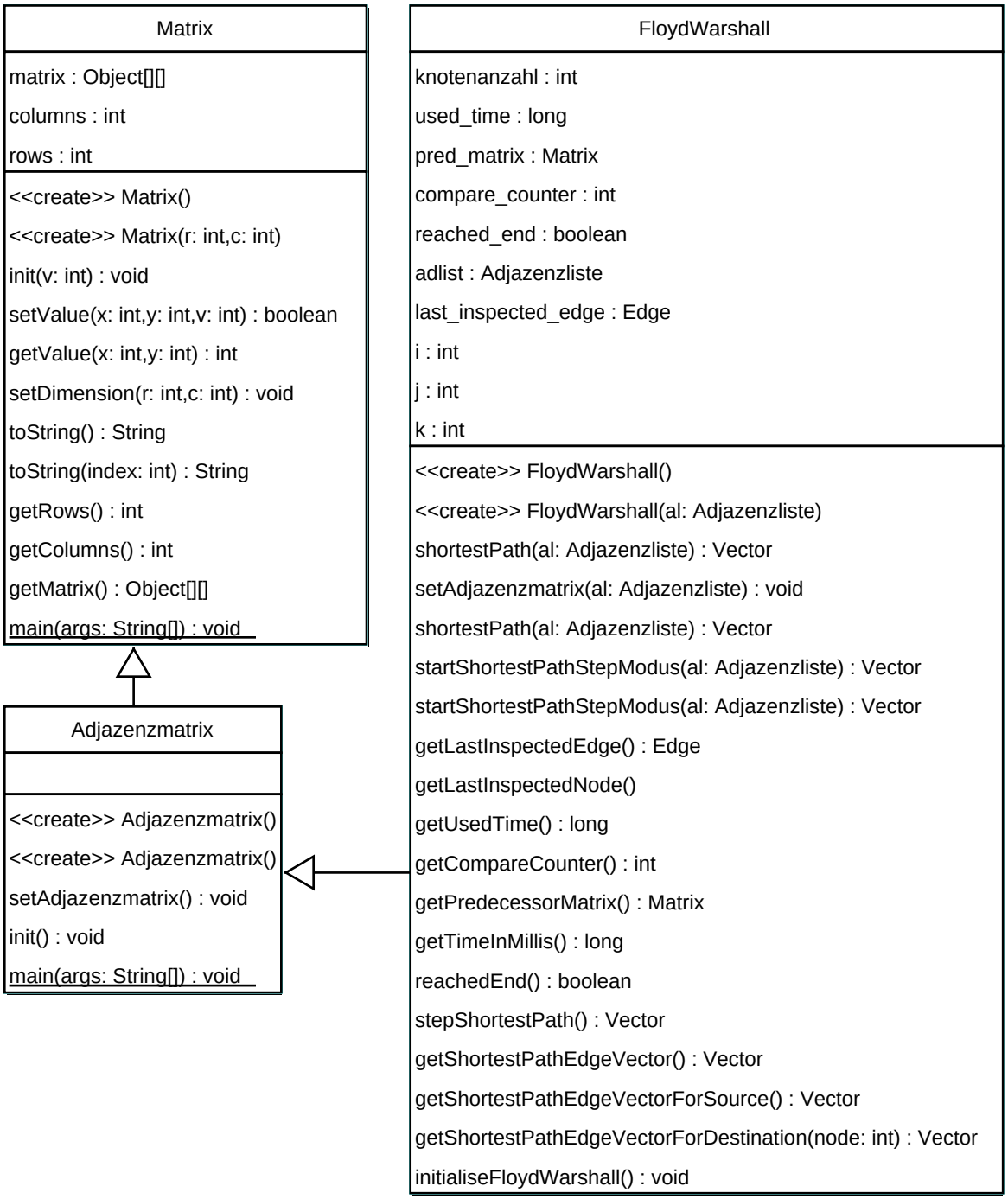
Ein vollständiges Klassendiagramm ist auf der beiliegenden CD zu finden (Verzeichnis *YAV/klassen* oder im Programmpfad Unterverzeichnis *klassen*). Weiterhin findet man auch dort das Programm ArgoUML und die ArgoUML Datei „Diplomarbeit.zargo“, die neben den Klassendiagrammen auch die Zustandsdiagramme enthält[43]. Eine kurze Erläuterung zu den Klassen haben wir bereits in Abschnitt 8.3 gegeben. An dieser Stelle wollen wir noch einmal die Klassendiagramme der wichtigsten Klassen aufzeigen. Diese wären *Adjazenzliste*, *Adjazenzmatrix*, *BellmanFordMoore*, *Dijkstra*, *FloydWarshall* und *Matrix*.

[43] Verzeichnis *Zusatz/ArgoUML/* mit den entsprechenden Unterverzeichnissen für die ArgoUML Programme und Verzeichnis *Ausarbeitung/ArgoUML/* für die Projektdatei


Adjazenzliste
adlist : Vector[]
<<create>> Adjazenzliste()
contains(e: Edge) : boolean
find(tail: int,head: int) : Edge
contains(n: int) : boolean
getCost() : int
contains() : boolean
contains(o: Object,n: int) : boolean
getEdgesWithTail(tailvalue: int) : Vector
getEdgesWithTail() : Vector
add(cost: int) : void
add(e: Edge) : void
add(n: int) : void
add() : void
remove(e: Edge) : void
remove() : void
remove(n: int) : void
getMaxNodeNumber() : int
getNodeVector() : Vector
toString() : String
ordnung() : int
size() : int
toHtml() : String
writeTo(filename: String) : void
writeAsText(filename: String) : void
readFrom(filename: String) : void
getAdlist() : Vector[]
sortEdgeList(node: int) : void
sortEdgeList() : void
sortEdgeLists() : void
main(args: String[]) : void


BellmanFordMoore
knotenanzahl : int
vorgaenger : int[]
knotenvektor : Vector
ha : HeapElement[]
heap_enum : Enumeration
used_time : long
compare_counter : int
<<create>> BellmanFordMoore()
<<create>> BellmanFordMoore(al: Adjazenzliste)
heapIsEmpty() : boolean
updateAdjazenzliste(al: Adjazenzliste) : void
shortestPath() : Vector
shortestPath() : Vector
getTimeInMillis() : long
startShortestPathStepModus() : Vector
startShortestPathStepModus() : Vector
getHeap()
getPredecessor() : int[]
getPredecessorLength() : int
getLastInspectedNode()
getUsedTime() : long
getCompareCounter() : int
stepShortestPath() : Vector
getShortestPathEdgeVector() : Vector
getShortestPathEdgeVectorForSource() : Vector
getShortestPathEdgeVectorForDestination(node: int) : Vector
initialiseBellmanFordMoore() : void


Dijkstra
knotenanzahl : int
vorgaenger : int[]
knotenvektor : Vector
heap_enum : Enumeration
used_time : long
compare_counter : int
<<create>> Dijkstra()
<<create>> Dijkstra(al: Adjazenzliste)
heapIsEmpty() : boolean
updateAdjazenzliste(al: Adjazenzliste) : void
shortestPath() : Vector
shortestPath() : Vector
getTimeInMillis() : long
startShortestPathStepModus() : Vector
startShortestPathStepModus() : Vector
getHeap()
getPredecessor() : int[]
getPredecessorLength() : int
getLastInspectedNode()
getUsedTime() : long
getCompareCounter() : int
stepShortestPath() : Vector
getShortestPathEdgeVector() : Vector
getShortestPathEdgeVectorForSource() : Vector
getShortestPathEdgeVectorForDestination(node: int) : Vector
initialiseDijkstra() : void


Matrix
matrix : Object[][]
columns : int
rows : int
<<create>> Matrix()
<<create>> Matrix(r: int,c: int)
init(v: int) : void
setValue(x: int,y: int,v: int) : boolean
getValue(x: int,y: int) : int
setDimension(r: int,c: int) : void
toString() : String
toString(index: int) : String
getRows() : int
getColumns() : int
getMatrix() : Object[][]
main(args: String[]) : void
Adjazenzmatrix
<<create>> Adjazenzmatrix()
<<create>> Adjazenzmatrix()
setAdjazenzmatrix() : void
init() : void
main(args: String[]) : void
FloydWarshall
knotenanzahl : int
used_time : long
pred_matrix : Matrix
compare_counter : int
reached_end : boolean
adlist : Adjazenzliste
last_inspected_edge : Edge
i : int
j : int
k : int
<<create>> FloydWarshall()
<<create>> FloydWarshall(al: Adjazenzliste)
shortestPath(al: Adjazenzliste) : Vector
setAdjazenzmatrix(al: Adjazenzliste) : void
shortestPath(al: Adjazenzliste) : Vector
startShortestPathStepModus(al: Adjazenzliste) : Vector
startShortestPathStepModus(al: Adjazenzliste) : Vector
getLastInspectedEdge() : Edge
getLastInspectedNode()
getUsedTime() : long
getCompareCounter() : int
getPredecessorMatrix() : Matrix
getTimeInMillis() : long
reachedEnd() : boolean
stepShortestPath() : Vector
getShortestPathEdgeVector() : Vector
getShortestPathEdgeVectorForSource() : Vector
getShortestPathEdgeVectorForDestination(node: int) : Vector
initialiseFloydWarshall() : void

13 Anhang B ausgewählter Quelltext

13.1 Adjazenzliste

Grundlage der Datenstruktur des Programms ist der Datentyp *int*. Die Klasse *Node* baut auf diesen auf und bietet die elementare Knotenobjekte, die von der Klasse *Edge* und schließlich von der Adjazenzliste wieder benutzt werden.

Die Adjazenzliste verwaltet zu jedem Knotennamen (int- Wert entspricht Index im Array von Vektoren) eine Kantenliste (Vector mit Edge- Objekten). Ist ein Knoten nicht im Graphen enthalten und gibt es einen Knoten mit einem größeren Namen, dann ist die Kantenliste des Knotenindizes gleich dem Element *null*. Ist jedoch ein Knoten enthalten, von dem keine Kanten ausgehen, dann ist die Kantenliste des Knotens der leere Vektor (new Vector).

```
public class Adjazenzliste {

  public Vector[] adlist; // Array von Vectoren
...
  public boolean contains (Edge e) {
    boolean rc=false;
    if (e.tail.value >= adlist.length) return false;
    Enumeration tmp_enum = adlist[e.getTailValue()].elements();

    while (tmp_enum.hasMoreElements()) {
      Edge tmp_edge=new Edge();
      Object tmp_element=tmp_enum.nextElement();

      if (tmp_element.getClass().isInstance((Object)tmp_edge))  {
        tmp_edge = (Edge) tmp_element;
        if ((tmp_edge.getTailValue()==e.getTailValue()) &&
            (tmp_edge.getHeadValue()==e.getHeadValue()))
              rc=true;
      }
    }
    return rc;
  }
  public Edge find(int tail,int head) {
    Edge e = null;
    if (tail >= adlist.length) return e;
    Enumeration tmp_enum = adlist[tail].elements();

    while (tmp_enum.hasMoreElements()) {
      Edge tmp_edge=new Edge();
```

```
    Object tmp_element=tmp_enum.nextElement();

    if (tmp_element.getClass().isInstance((Object)tmp_edge))  {
      tmp_edge = (Edge) tmp_element;
      if ((tmp_edge.getTailValue()==tail) &&
          (tmp_edge.getHeadValue()==head))
            e=tmp_edge;
    }
  }
  return e;
}
public Vector getEdgesWithTail (int tailvalue) {
  if (tailvalue < 0) return null;
  if (tailvalue>=adlist.length) return null;
  if (adlist[tailvalue]==null) return null;
  Vector ewt = new Vector (adlist[tailvalue]);
  return ewt;
}
public void add (Edge e) {
  Vector[] tmp_adlist;
  if (e.tail.value > adlist.length - 1) {
    tmp_adlist = new Vector [e.tail.value + 1];
    System.arraycopy(adlist,0,tmp_adlist,0,adlist.length);
    adlist=new Vector [e.tail.value + 1];
    adlist=tmp_adlist;
    adlist[e.tail.value]=new Vector();
  }
  if (adlist[e.tail.value]==null) adlist[e.tail.value]=new Vector();
  boolean test = contains(e);
  if (!test) {
    adlist[e.tail.value].addElement(e);
    if (!contains(e.head)) add(e.head);
  }

}
public void add (int n) {
  Vector[] tmp_adlist;
  if (n > adlist.length - 1) {
    tmp_adlist = new Vector [n + 1];
    System.arraycopy(adlist,0,tmp_adlist,0,adlist.length);
    adlist=new Vector [n + 1];
    adlist=tmp_adlist;
    adlist[n]=new Vector();
  }
```

```
    if (adlist[n]==null) adlist[n]=new Vector();

  }
  public void remove (Edge e) {
    if (e.getTailValue()<adlist.length) {
      if (adlist[e.getTailValue()]!=null) {
        Enumeration tmp_enum = adlist[e.getTailValue()].elements();
          while (tmp_enum.hasMoreElements()) {
          Edge tmp_edge=new Edge();
          Object tmp_element=tmp_enum.nextElement();
          if (tmp_element.getClass().isInstance((Object)tmp_edge))  {
            tmp_edge = (Edge) tmp_element;
            if (tmp_edge.equal(e))
              adlist[e.getTailValue()].remove(tmp_edge);
          }
        }
      }
    }
  }
   public void remove (int n) {
    if (n < adlist.length) {
      adlist[n]=null;
      int i;
      for (i=0; i<adlist.length;i++) {
        if (adlist[i]!=null) {
          Enumeration tmp_enum = adlist[i].elements();
            while (tmp_enum.hasMoreElements()) {
            Edge tmp_edge=new Edge();
            Object tmp_element=tmp_enum.nextElement();
            if (tmp_element.getClass().isInstance((Object)tmp_edge))  {
              tmp_edge = (Edge) tmp_element; // Hole Element aus Vector
              if (tmp_edge.hasNode(n)) {
                remove(tmp_edge);
              }
            }
          }
        }
      }
    }
   }
...
}
```

13.2 Matrix

Grundlage der Klasse Matrix ist ein zweidimensionales Array von Objekten. Mittels *int* Werten werden in der Methode *setValue* Objekte der Klasse *Integer* erzeugt und in der Matrix an der entsprechenden Stelle gesetzt. Genauso werden die *Integer* Objekte wieder ausgelesen und *int* Werte von der Methode *getValue* zurückgegeben. Die Methode *setDimension* verändert die Matrixgrösse. Dabei wird bei einer Erweiterung die Matrix mit undefinierten Feldinhalten vergrößert. Bei einer Verkleinerung jedoch werden eventuell vorhandene Felder, die außerhalb der neuen Dimension liegen, einfach verworfen.

```
...
public class Matrix {
  Object[][] matrix;
  int columns=0;
  int rows=0;
...
  public void setDimension(int r,int c) {
    Object [][] tmp_m= new Object [r][c];
    for (int i=0;i<Math.min(r,getRows());i++) {
      for (int j=0;j<Math.min(c,getColumns());j++) {
        tmp_m[i][j]=matrix[i][j];
      }
    }
    matrix=tmp_m;
    columns=c;
    rows=r;
  }
...
}
```

13.3 Adjazenzmatrix

Die Adjazenzmatrix ist Kindklasse von *Matrix* und bietet somit die gleiche Funktionalität. Hinzu kommt z.B. noch die Methode *setAdjazenzmatrix* die aus einer Adjazenzliste eine Adjazenzmatrix aufbaut. Dabei werden vorhandene Einträge der Adjazenzmatrix überschrieben und die entsprechenden Einträge aus der Adjazenzliste eingetragen. Die Adjazenzliste wird Knotenweise durchlaufen. Zuerst wird die Kantenliste des Knotens 0 bearbeitet und in der Adjazenzmatrix eingetragen, dann die des Knotens 1 usw...

```
...
public class Adjazenzmatrix
extends Matrix
```

```
{
...
  public void setAdjazenzmatrix(Adjazenzliste al) {
    int nodes=al.getMaxNodeNumber();
    setDimension(nodes+1,nodes+1);
    init();
    for (int i=0;i<nodes+1;i++) {
      Vector kantenliste =al.getEdgesWithTail(i);
      if (kantenliste!=null) {
        Enumeration tmp_enum = kantenliste.elements();
        while (tmp_enum.hasMoreElements()) {
          Edge tmp_edge=(Edge)tmp_enum.nextElement();
          setValue(tmp_edge.getTailValue(),
                   tmp_edge.getHeadValue(),
                   tmp_edge.getCost());
        }
      }
    }
  }
 ...
}
```

13.4 Heap

In der Heap Klasse ist gut zu erkennen, wie stark sich diese auf die vererbten Methoden der Java-Klasse *Vector* stützt. Einzig bei der Methode *find* wird nicht direkt auf eine Methode der Elternklasse zugegriffen. Die Methoden *findFirstElement* und *deleteFirstElement* sind in der Methode *findAndRemoveFirstElement* zusammengefasst, da diese Methoden in dem Algorithmus Bellman-Ford-Moore nur zusammen aufgerufen werden. Ebenso gibt es ein *findAndRemoveMin*, das im Dijkstra Algorithmus benutzt wird. Die Namensgebung der Methoden haben wir an die der Elternklasse *Vector* angepasst, so haben wir die Methoden des Pseudocodes *insert* und *delete* in *add* und *remove* umbenannt.

```
...
public class Heap extends Vector {
  public void add(HeapElement he) { //insert
    this.add((Object) he);
  }
  public void addOrReplace(HeapElement hel) {
    HeapElement he=find(hel.getNode());
    if (he.getNode()!=null) this.remove((Object) he);
    this.add((Object) hel);
```

```
}
public void remove(HeapElement he) {
  this.remove((Object) he);
}
public HeapElement findAndRemoveMin() {
  HeapElement he = getMin();
  remove(he);
  return he;
}
public HeapElement findAndRemoveFirstElement() {
  HeapElement he =(HeapElement) firstElement();
  remove(he);
  return he;
}
public HeapElement getMin() {
  HeapElement min_node = null;
  if (this.size()!=0){
    Collections.sort(this);
    min_node=(HeapElement)this.firstElement();
  }
  return min_node;
}
public HeapElement findMin() {
  return getMin();
}
public void decreaseKey(HeapElement he, int new_key) {
  remove (he);
  he.setKey(new_key);
  add(he);
}
public HeapElement find (Node n) {
  HeapElement he = new HeapElement();
  boolean gefunden=false;
  Enumeration heap_enum = elements();
  while (heap_enum.hasMoreElements() && !(gefunden)) {
    HeapElement tmp_element=(HeapElement)heap_enum.nextElement();
    if (tmp_element.getNodeValue()==n.getValue()) {
      he=tmp_element;
      gefunden=true;
    }
  }
  return he;
}
public int getCost(Node n) {
```

```
    HeapElement he = find(n);
    if (he.knoten!=null) return he.getKey();
    else return Integer.MAX_VALUE;
  }
  public void setCost(Node n,int cost) { // decreasekey
    HeapElement he = find(n);
    if ((he.getNodeValue()!=-1) && (he.getNode()!=null)) {
      remove(he);
      he.setKey(cost);
      add(he);
    }

  }
...
}
```

13.5 Algorithmen

Die Algorithmen haben wir ja bereits in Abschnitt 5 anhand des Pseudocodes erklärt. Hier wollen wir, der Vollständigkeit halber, den wichtigsten Teil des Quellcodes aufzeigen. Zu beachten ist, dass im Vergleich zum Pseudocode weitergehende Informationen, wie die Anzahl der Vergleiche[44] oder die benötigten Millisekunden gesammelt werden. Für den Bellman-Ford-Moore Algorithmus wird ein Array von Heapelementen (Variable $ha[..]$) geführt, dieses enthält den bislang gefunden kürzesten Weg[45]. Bei Dijkstra brauchen wir dieses Array nicht, da wir die Informationen dort im Heap führen und außerhalb des Heaps nicht mehr benötigen[46]. Die wichtigsten Variablen sind als globale Variablen[47] realisiert, damit beim Algorithmusdurchlauf im Stepmodus der aktuelle Stand gehalten werden kann und für den nächsten Schritt zur Verfügung steht.

Bellman-Ford-Moore Voraussetzung für den Algorithmus ist, dass ein Graph in der Adjazenzliste vorliegt und ein Startknoten (start, bzw. source) übergeben wurde. Zunächst werden die entsprechenden Variablen initialisiert *initialiseBellmanFordMoore*. Dann wird der Algorithmus gemäß dem Pseudocode (siehe Abschnitt 5.1) durchlaufen. Unterschied zum Pseudocode ist, dass das Finden und Löschen des ersten Elements in einer Methode

[44] compare Zähler

[45] $k(v_0, v)$ im Pseudocode

[46] Bei Bellman-Ford-Moore kann ein Knoten mehrmals in dem Heap eingefügt werden, deshalb benötigen wir die Information immer. Bei dem Dijkstra Algorithmus hingegen wird ein Knoten niemals doppelt in dem Heap eingefügt. Deshalb wird die Information über die Länge des bislang gefundenen kürzesten Weges nur im Heap geführt.

[47] Klassenattribute

zusammengefasst ist. Da bei diesem Algorithmus die Knoten mehrmals in dem Heap eingefügt werden können, müssen wir die Länge des bislang gefundenen kürzesten Weges in einer zusätzlichen Struktur halten. Es reicht also nicht aus diese Information in dem Heap mitzuspeichern. Dazu verwenden wir ein Array von *int*- Werten (ha[...]). Zu jedem Knoten gibt es also einen Arrayeintrag, der ein entsprechendes *HeapElement* speichert.

```
...
public class BellmanFordMoore
extends Adjazenzliste {

  int knotenanzahl = 0;
  int[] vorgaenger = new int[knotenanzahl+1];
  Vector knotenvektor = new Vector();
  Heap h = new Heap();
  HeapElement[] ha = null;
  Enumeration heap_enum = knotenvektor.elements();
  Node source = new Node();
  Node destination = new Node();
  Node last_inspected_node;
  long used_time;
  int compare_counter=0;

...
  public Vector shortestPath (Node start,Node end) {
    source = new Node(start);
    if (end!=null) destination = new Node(end);
    else destination=null;
    compare_counter=0;
    last_inspected_node = new Node();
    used_time=getTimeInMillis();
    if (!(source==null) ) {
      initialiseBellmanFordMoore();
      while (!(h.isEmpty())) {
        HeapElement fe=h.findAndRemoveFirstElement();
        int c=((HeapElement)ha[fe.getNodeValue()]).getKey();
        Vector kanten = getEdgesWithTail(fe.getNode());
        if (kanten!=null) {
          Enumeration kanten_enum = kanten.elements();
          while (kanten_enum.hasMoreElements()) {
            Object tmp_element=kanten_enum.nextElement();
            Edge tmp_edge = (Edge) tmp_element;
            Node head = tmp_edge.getHead();
            int cost = (ha[fe.getNodeValue()]).getKey()
```

```
                    +tmp_edge.getCost();
            compare_counter++;
            if (cost<((HeapElement)ha[head.getValue()]).getKey()) {
              vorgaenger[head.getValue()]=fe.getNodeValue();
              h.addOrReplace(new HeapElement(head,cost));
              ha[head.getValue()]=new HeapElement(head,cost);
            }
          }
        }
        kanten=null;
      }
    }
    used_time=getTimeInMillis()-used_time;
    return getShortestPathEdgeVector();
  }
...
}
```

Dijkstra Voraussetzung für den Algorithmus ist, dass ein Graph in der Adjazenzliste vorliegt und ein Startknoten (start, bzw. source) übergeben wurde. Zunächst werden die entsprechenden Variablen initialisiert (*initialiseDijkstra*). Dann wird der Algorithmus gemäß dem Pseudocode (siehe Abschnitt 5.2) durchlaufen. Unterschied zum Pseudocode ist, dass das Finden und Löschen des minimalen Elements zu einer Methode zusammengefasst wurde (*findAndRemoveMin*).

```
...
public class Dijkstra
extends Adjazenzliste {

  int knotenanzahl = 0;
  int[] vorgaenger = new int[knotenanzahl+1];
  Vector knotenvektor = new Vector();
  Heap h = new Heap();
  Enumeration heap_enum = knotenvektor.elements();
  Node source = new Node();
  Node destination = new Node();
  Node last_inspected_node;
  long used_time;
  int compare_counter=0;

  public Vector shortestPath (Node start,Node end) {
    source = new Node(start);
    if (end!=null) destination = new Node(end);
```

```
    else destination = null;

    compare_counter=0;
    last_inspected_node = new Node();
    used_time=getTimeInMillis();
      if (!(source==null)) {
      initialiseDijkstra();
      while (!(h.isEmpty())) {
        HeapElement min=h.findAndRemoveMin();
        Vector kanten = getEdgesWithTail(min.getNode());
        if (!(min.getKey()==Integer.MAX_VALUE)) {
          Enumeration kanten_enum = kanten.elements();
          while (kanten_enum.hasMoreElements()) {
            Object tmp_element=kanten_enum.nextElement();
            Edge tmp_edge = (Edge) tmp_element;
            Node head = tmp_edge.getHead();
            int cost = min.getKey()+tmp_edge.getCost();
            compare_counter++;
            if (cost<h.getCost(head)) {
              h.setCost(head,cost);
              vorgaenger[head.getValue()]=min.getNodeValue();
            }
          }
        }
      }
    }
    used_time=getTimeInMillis()-used_time;
    return getShortestPathEdgeVector();
  }
...
}
```

Floyd-Warshall Voraussetzung für den Algorithmus ist, dass ein Graph in der Adjazenzliste vorliegt. Eine Knotenvorgabe kann hier entfallen, da der Algorithmus die kürzesten Wege zwischen allen Knoten berechnet. Zunächst werden die entsprechenden Variablen initialisiert und die Adjazenzmatrix anhand der Adjazenzliste aufgebaut (*setAdjazenzmatrix*). Dann wird der Algorithmus gemäß dem Pseudocode (siehe Abschnitt 5.3) durchlaufen. Die bislang gefundene Länge des Weges zu einem Knoten wird in der Matrix verwaltet. Die Vorgängermatrix *pred_matrix* wird parallel erzeugt.

```
...
public class FloydWarshall
extends Adjazenzmatrix {
```

```
  int knotenanzahl = 0;
  long used_time;
  Matrix pred_matrix = new Matrix();
  int compare_counter=0;
  boolean reached_end=true;

  Adjazenzliste adlist = null;

  Node source = new Node();
  Node destination = new Node();
  Node last_inspected_node=new Node();

  int i=0;
  int j=0;
  int k=0;

...
  public Vector shortestPath (Node start,Node end,Adjazenzliste al) {
    if (start!=null)source = new Node(start);
    else source=null;
    if (end!=null) destination = new Node(end);
    else destination=null;

    compare_counter=0;
    used_time=getTimeInMillis();
    setAdjazenzmatrix(al);
    for (j=0; j<knotenanzahl;j++) {
      for (i=0; i<knotenanzahl;i++) {
        for (k=0;k<knotenanzahl;k++) {
          compare_counter++;
          if   ((getValue(i,j)!=Integer.MAX_VALUE)
             && (getValue(j,k)!=Integer.MAX_VALUE)) {
               if ((long)getValue(i,k)>
                  (long)getValue(i,j)+(long)getValue(j,k)) {
                    setValue(i,k,getValue(i,j)+getValue(j,k));
                    pred_matrix.setValue(i,k,pred_matrix.getValue(j,k));
               }
          }
        }
      }
    }
    reached_end=true;
    used_time=getTimeInMillis()-used_time;
```

```
    if (!(source==null) || (destination==null))
       return getShortestPathEdgeVector();
    else return null;
  }
...
}
```

Step-Modus Am Beispiel des Bellman-Ford-Moore Algorithmus wollen wir zeigen, wie der Stepmodus implementiert wurde. Es gibt im Prinzip zwei Methoden, die nötig sind. Eine Methode initialisiert den Algorithmus (*startShortestPathStepModus*) und eine führt einen Algorithmusschritt aus (*stepShortestPath*). Die benötigten Objekte werden als Klassenattribute implementiert, damit diese nach einem Algorithmusschritt wieder zur Verfügung stehen. So muss z.B. der Heap (Objekt h) für den nächsten Schritt vorhanden sein.

```
...
  public Vector startShortestPathStepModus (Node start,Node end) {
    source = new Node(start);
    if (end!=null) destination = new Node(end);
    else destination=null;
    last_inspected_node = new Node();
    compare_counter=0;

    used_time=getTimeInMillis();
    if (!(source==null) ) {
      initialiseBellmanFordMoore();
    }
    used_time=getTimeInMillis()-used_time;
    return getShortestPathEdgeVector();
  }

  public Vector stepShortestPath (){
    if (!(h.isEmpty())) {
      HeapElement fe=h.findAndRemoveFirstElement();
      last_inspected_node = new Node(fe.getNode());
      int c=((HeapElement)ha[fe.getNodeValue()]).getKey();
      Vector kanten = getEdgesWithTail(fe.getNode());
      if (kanten!=null) {
        Enumeration kanten_enum = kanten.elements();
        while (kanten_enum.hasMoreElements()) {
          Object tmp_element=kanten_enum.nextElement();
          Edge tmp_edge = (Edge) tmp_element;
          Node head = tmp_edge.getHead();
```

```
          int cost = (ha[fe.getNodeValue()]).getKey()+tmp_edge.getCost();
          compare_counter++;
          if (cost<((HeapElement)ha[head.getValue()]).getKey()) {
            vorgaenger[head.getValue()]=fe.getNodeValue();
            h.addOrReplace(new HeapElement(head,cost));
            ha[head.getValue()]=new HeapElement(head,cost);
          }
        }
      }
    }
    return getShortestPathEdgeVector();
  }
...
```

13.6 Ermitteln des Kantenvektors

Für Dijkstra und Bellman-Ford-Moore wird der Kantenvektor (Vektor der Kanten, die auf dem kürzesten Weg liegen) aus der Vorgängerliste erzeugt. Wie man eine Vorgängerliste durchläuft, haben wir im Abschnitt 4.2 auf Seite 13 gesehen. Die Kantenliste wird abhängig davon aufgebaut, ob ein Start- oder ein Start- und Zielknoten vorgegeben wurde. Wird ein Start- und ein Zielknoten vorgegeben, dann wird die Kantenliste vom Zielknoten zum Startknoten durchlaufen und die entsprechenden Kanten erzeugt und in den Vector eingefügt (Methode *getShortestPathEdgeVectorForDestination*). Ist kein Zielknoten vorgegeben muss diese Methode einfach nur für alle möglichen Zielknoten aufgerufen und die so entstehenden Kantenlisten zusammengefügt werden (Methode *getShortestPathEdgeVectorForSource*).

```
  private Vector getShortestPathEdgeVector () {
    Vector kuerzester_weg = new Vector();
    if (source==null) return kuerzester_weg;
    if (destination==null) {
      return getShortestPathEdgeVectorForSource();
    }
    if (destination.getValue()==-1) {
      return getShortestPathEdgeVectorForSource();
    }
    int i_knoten = destination.getValue();
    kuerzester_weg.addAll(
         getShortestPathEdgeVectorForDestination(i_knoten)
         );
    return kuerzester_weg;
  }
```

```
private Vector getShortestPathEdgeVectorForSource() {
    Vector kuerzester_weg = new Vector();
    int maxnnum = getMaxNodeNumber();
    for (int i=0;i<maxnnum+1;i++) {
      kuerzester_weg.addAll(
           getShortestPathEdgeVectorForDestination(i)
           );
    }
    return kuerzester_weg;
}

private Vector getShortestPathEdgeVectorForDestination(int node) {
  Vector kw=new Vector();
  while (node!=source.getValue() && vorgaenger[node]!=-1) {
    Edge tmp_e=find(vorgaenger[node],node);
    if (tmp_e!=null) kw.add(tmp_e);
    node=vorgaenger[node];
  }
  return kw;
}
```

14 CD Inhalt

Die Dateien haben wir kursiv dargestellt. Verzeichnisse sind in normaler Schrift angegeben. In Klammern haben wir Kommentare zu den Verzeichnissen und Dateien angegeben. Angegeben sind nur die wichtigsten Dateien und Verzeichnisse.

- Ausarbeitung
  - ArgoUML (enthält die ArgoUML Projektdatei zur Diplomarbeit)
  - Diplomarbeit Latex DVI (diese Ausarbeitung in LaTeX und DVI)
  - *diplomarbeit.pdf* (diese Ausarbeitung im PDF Format)
- Zusatz (enthält einen Teil der Programme, die bei der Entwicklung der Diplomarbeit benutzt wurden)
  - ArgoUML (ArgoUML Programmversionen)
  - EssModel(Windows)
  - GPL (GNU General Puplic License)
  - Java2html (Programm zum Formattieren von Quellcode)
  - JCreator(Windows) (Java IDE zur Entwicklung unter Windows)
  - JDK (Java 1.4.2 JDK für Linux und Windows)

  - Kile(SuSELinux) (Latex IDE für Linux)

- Java (JCreator Projektdateien, Quellcode, Hilfe ... zu unserm Tool YAV)
- YAV (Jar Datei komplett mit Hilfe-, Sourcecode-, API- usw. Verzeichnis)
- *Autorun.inf* (startet unter Windows automatisch die Datei YAVsetup.exe)
- *Java.zip* (gepackte Version des Verzeichnisses Java)
- *YAV.exe* (Selbstextrahierendes RAR Verzeichnis, unter Windows als Alternative zu YAVsetup.exe)
- *YAV.zip* (Zip Archiv für die Installation unter allen gängigen Betriebsystemen)
- *YAVsetup.exe* (Windows Installationsroutine)

Studentische Erklärung

Hiermit erkläre ich, dass die vorliegende Diplomarbeit von mir selbstständig verfasst und angefertigt wurde, nur die angegebenen Quellen und Hilfsmittel benutzt und Zitate kenntlich gemacht wurden.

Dortmund, den 12. Mai 2004

Literatur

[1] Heide Balzert, *Lehrbuch der Objektmodellierung*, Spektrum Akademischer Verlag Heidelberg Berlin, 1999

[2] Reinhard Diestel, *Grahentheorie*, Springer-Verlag, http://www.math.uni-hamburg.de/home/diestel/books/graphentheorie/ (2000)

[3] Jørgen Bang-Jensen and Gregory Gutin, *Digraphs: Theory, Algorithms and Applikations*, Springer-Verlag London Berlin Heidelberg, 2001

[4] Ralf Hartmut Güting, *Datenstrukturen und Algorithmen*, B.G. Teubner Stuttgart, 1992

[5] W. Hauenschild, *Vorlesung Algorithmen und Datenstrukturen* Universität Paderborn, http://www.uni-paderborn.de/∼hauenschild/,

[6] Volker Turau, *Algorithmische Graphentheorie*, Addison Wesley (Deutschland) GmbH Bonn, 1996

[7] Niklaus Wirth, *Algorithmen und Datenstrukturen (Pascal-Version)*, B.G. Teubner Stuttgard-Leipzig-Wiesbaden, 5. Auflage, Oktober 2000

[8] *Internetseiten der Uni Köln*, Kombinatorische Optimierung http://www.zaik.uni-koeln.de/∼buzdemir/ kombinatorische_opt/sections.html

[9] Algorithmic Solutions Software GmbH, *LEDA*,http://www.algorithmic-solutions.com, 01.05.2004

[10] Biliana Kaneva, Dominique Thiébaut, *Applet zur Verdeutlichung von Graphalgorithmen*, http://cs.smith.edu/∼thiebaut/java/graph/select.html, 04.05.2004

[11] H.W. Lang, *Applet zur Verdeutlichung des Dijkstra und eines Algorithmus zur Berechnung eines minimalen Spannbaums*, http://www.iti.fh-flensburg.de/lang/algorithmen/graph/shortest.htm, 04.05.2004

Entwicklung und Software

[12] University of California, *ArgoUML*, http://argouml.tigris.org, 01.03.2004

[13] B1G Software, *b1gSetup*, Programm zur Erstellung einer Installationsdatei unter Windows, http://www.b1gsetup.info, 05.05.2004

[14] Alexander Larsson, *Dia*, Programm zur einfachen Diagrammerstellung, Version 0.91, http://www.lysator.liu.se/∼alla/dia, 30.03.2004

[15] Robin Quast, *Diplomarbeit "Theorie und JAVA-Realisierung ausgewählter Algorithmen zur Bestimmung kürzester Wege in Graphen"*, http://www.robinquast.de/diplom, 13.05.2004

[16] IBM Corp und andere, *Eclipse*, umfangreiche Entwicklungsumgebung, http://www.eclipse.org, 01.05.2004

[17] Eldean AB Sweden, *Essmodel*, UML Dokumentationprogramm, http://www.essmodel.com oder http://essmodel.sourceforge.net, 06.05.2004

[18] Spencer Kimball und Peter Mattis, *The Gimp*, Version 1.3.20, Bildbearbeitungsprogramm, http://www.gimp.org, 30.03.2004

[19] SUN, *Java 1.42* SDK, http://www.sun.de, 01.05.2004

[20] Markus Gebhard, *Java2HTML*, http://www.java2html.de, 04.03.2004

[21] SUN, *Forte for Java*, Java Entwicklungsumgebung, http://www.sun.de, 04.05.2004

[22] Xinox Software, *JCreator*, Freeware Version 3.0, Java Entwicklungsumgebung unter Windows, http://www.jcreator.com, 01.04.2004

[23] Jeroen Wijnhout, *Kile*, Version 1.6, KDE LaTeX-Umgebung, http://kile.sourceforge.net, 20.02.2004

[24] Klaus Knopper, *Knoppix Live Linux on CD*, Linux Distribution, die von CD startet, http://www.knopper.net, 01.05.2004

[25] Free Software Foundation, *Miktex*, Latexumgebung für Windows, http://www.miktex.org, 01.04.2004

[26] Eugene Roshal, *RAR*, Archivierungsprogramm, Version 3.30, http://www.winrar.de, 01.05.2004

[27] Stefan Münz, *Selfhtml*, HTML Informationen und Tutorial, http://www.teamone.de, 02.05.2004

[28] SuSE, *SuSE Linux 9.0*, Linux Distribution mit KDE Overfläche Version 3.1.4, http://www.suse.de, 01.05.2004

[29] Microsoft, *Windows*, Betriebsystem, http://www.microsoft.de, 01.05.2004

[30] Info Zip Gruppe, *Info-ZIP*, Zip und Unzip Programm, http://www.info-zip.de, 01.05.2004

Aus[3], [6] und [4] übernommene Referenzen

[31] R.E. Bellman, On a routing problem, *Quart. Appl. Math.*, 16:87-90, 1958

[32] F. Buckley und F. Harary „Distance in Graphs“, Addison-Wesley Publishing Co. Reading, Massachusetts, 1990

[33] B.B. cherkassky, A.V. Goldberg and T. Radzik, „Shortest Paths Algorithms: Theory and Experimental Evaluation“, Stanford University, Technical Report, 1993

[34] E.W. Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik*, 1:269-271, 1959

[35] N. Deo und C. Pang, „Shortest-Path Algorithms: Taxonomy and Annotation“, Networks 14, 275-323, 1984

[36] R.W. Floyd, Algorithm 97, shortest path, *Comm. ACM*, 5:345, 1962

[37] Jr. Ford, L.R. Network flow theory, Technical Report P-923, The Rand Corp., 1956

[38] G.N. Frederickson, „Fast algorithms for shortest paths in planar graphs, with applications“, SIAM J. on Computing, Vol. 16, 1004-1022, 1987

[39] P. Hart, N. Nilsson and B. Raphael, „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“, IEEE Trans. Sys. Sci. Cybern. 2, 100-107, 1968

[40] E.F. Moore, The shortest path through a maze, *In proc. of the Int. symp. on the Theory of Switching*, pages 285-292, Harward University Press, 1959

[41] S. Warshall, A theorem on boolean matrices, *J.ACM*, 9:11-12, 1962