HOCHSCHULE **TRIER**
Trier University of Applied Sciences
Informatik - Computer Science

Funktionsprinzip und Anwendungsbeispiele des Dijkstra-Algorithmus

Bearbeiter 1: Thomas Jürgensen
Bearbeiter 2: Annika Kremer
Bearbeiter 3: Tobias Meier

Gruppe: 13

Ausarbeitung zur Vorlesung Wissenschaftliches Arbeiten

Ort, Abgabedatum

Kurzfassung

kurzfassung Gr 13

Inhaltsverzeichnis

1

Einleitung und Problemstellung

In der folgenden Arbeit geht es um die Vorstellung des Dijkstra-Algorithmus. Hierzu wird im Folgenden grob erklärt, was Graphen sind und in welchen Datenstrukturen sie repräsentiert werden können. Danach wird auf den Algorithmus direkt eingegangen, mit einer einführenden Erklärung, der Klärung der Komplexität und anschließender Implementierung in Python Pseudo-Code. Zum Schluss wird noch auf aktuelle Anwendungsbeispiele des Algorithmus eingegangen, um zu zeigen, dass dieses Verfahren zwar schon vor längerer Zeit entwickelt wurde, doch immer noch in der Praxis relevant ist.

2

Graph

2.1 Definition Graph

-Für Dijkstra auf gerichtete Graphen ausgelegt.

Ein **Graph** G besteht aus einer Menge X [deren Elemente Knotenpunkte genannt werden] und einer Menge U, wobei jedem Element u ∈ U in eindeutiger Weise ein geordnetes oder ungeordnetes Paar von [nicht notwendig verschiedenen] Knotenpunkten, x,y ∈ X zugeordnet ist. Ist jedem u ∈ U ein geordnetes Paar von Knoten zugeordnet, so heißt der Graf **gerichtet**, und wir schreiben $G = (X, U)$. Die Elemente von U werden in diesem Fall als **Bögen** bezeichnet.

Ist jedem u ∈ U ein ungeordnetes Paar von Knotenpunkten zugeordnet, so heißt der Graph **ungerichtet** und wir schreiben $G = [X, U]$. Die Elemente von U bezeichnen wir dann als **Kanten.** [1]

Ein gerichteter Graph besteht somit aus einer Menge aus geordneten Knotenpunkten. Dadurch, das diese Punkte geordnet sind, ist eine Verlaufsrichtung festgelegt, in welcher der Graph zeigt. Somit kann sich eine geometrische Figur ergeben.

[Grafik hier einfügen]

Wenn man diese Punkte als Richtungen nun als Straßennetz sähe, würde sich durch die Punkte ein Straßenverlauf abbilden mit unterschiedlichen Verlaufsrichtungen. Somit verbünde nicht jede Strecke direkt jeden Punkt. Um jenen kürzesten Weg zu finden, der zwei bestimmte Punkte miteinander verbindet, kann man den Dijkstra-Algorithmus anwenden, welcher im Folgenden erklärt wird.

2.2 Datenstrukturen zur Repräsentation von Graphen

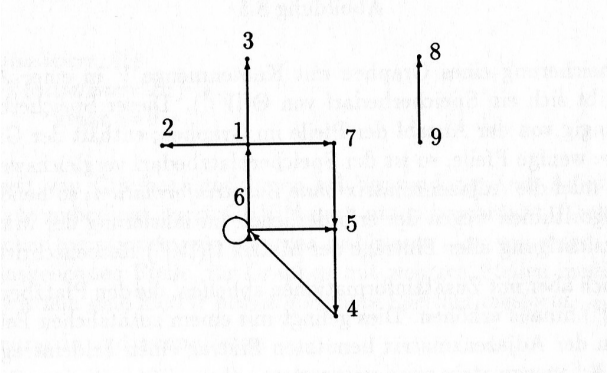
Speicherung in einer Adjazenzmatrix

Der Graph G=(V,E) wird in einer booleschen m*m Matrix gespeichert ($m = \sharp V$), mit 1 falls $(i, j) \in E$ und 0 falls nicht.

Nachteil: Dies erfordert unverhältnismäßig viel Speicher, wenn der Graph nur wenig Kanten hat. Abhilfe schafft eine Zusatzmatrix, die nur bedeutsame Einträge speichert. Insgesamt ist eine Adjazenzmatrix aber trotzdem ineffizient bei Graphen

[1] Bieß, Graphentheorie, S.9

mit wenig Kanten.



(a)

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

(b)

Abbildung 8.4

Abb. 2.1. Adjazenzmatrix (Quelle: OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S.539 Abb. 8.4)

Speicherung in Adjazenzlisten

Für jeden Knoten wird eine lineare, verkettete Liste seiner ausgehenden Kanten gespeichert. Die Knoten werden als lineares Feld gespeichert (d.h. jeder Knoten im Feld zeigt auf eine Liste).
Dies ist effizienter als eine Adjazenzmatrix, weil kein Speicherplatz verschwendet wird.

Speicherung in doppelt verketteten Listen

Jedes Element enthält Zeiger auf die beiden Nachbarelemente sowie auf eine Kantenliste (wie bei Adjazenzliste, s.o.). Diese Darstellung besitzt die den Adjazenzlisten fehlende Dynamik, ist aber natürlich komplizierter.

Quelle: Algorithmen und Datenstrukturen, T.Ottmann/P.Widmayer, Kap. 8 Graphenalgorithmen, S.539-544

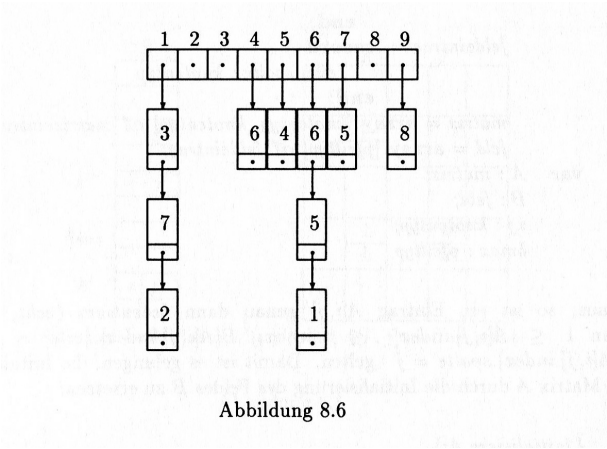
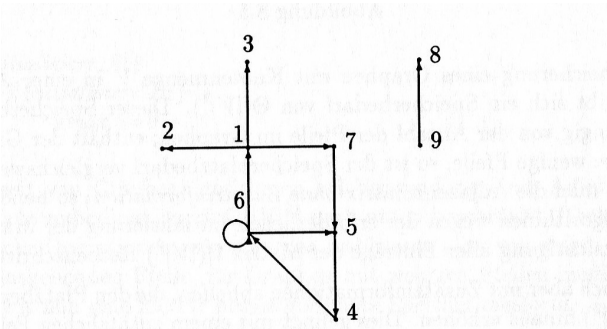


Abbildung 8.6

Abb. 2.2. Adjazenzliste (Quelle: OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S.542 Abb. 8.6)



(a)

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

(b)

Abbildung 8.4

Abb. 2.3. doppeltVerkettetePfeilliste.jpg (Quelle: OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S.543 Abb. 8.7)

3

Dijktra - Algorithmus

3.1 Erklärung

3.1.1 Das Shortest-Path Problem

Das Shortest-Path Problem ist in der Literatur manchmal auch unter dem Namen Single-Source Shortest Path Problem zu finden. Es behandelt die Frage, wie man in einem Graphen ausgehend von einem gegebenen Anfangsknoten s (engl. Source) zu jedem anderen Knoten des Graphen den kürzesten Pfad findet [1]. Mit kurz ist hierbei nicht die Länge gemeint, sondern die Minimierung der Kantenkosten. Im Allgemeinen ist mit Länge nicht die tatsächliche Länge gemeint, sondern die Pfadkosten.

3.1.2 Zur Eingabe

Die Eingabe besteht stets aus gerichteten Graphen. Hierbei muss jedoch angemerkt werden, dass sich jeder ungerichtete Graph leicht in einen gerichteten Graph umwandeln lässt, indem man für jede Kante {u,v} die Kanten (u,v) und (v,u) mit gleichen Kosten einführt.[2] Somit findet der Dijkstra-Algorithmus auch für ungerichtete Graphen Anwendung, solange man sie vorher in einen Digraphen überführt.
Zudem wird davon ausgegangen, dass die Kantenkosten nicht negativ sind.

3.1.3 Das Optimalitätsprinzip

Dem Dijkstra-Algorithmus liegt das Optimalitätsprinzip zugrunde:
Für jeden kürzesten Pfad p=(v_0,v_1,...,v_k) von v_0 nach v_k ist jeder Teilweg p'=(v_i, ..., v_j) mit $0 \leq i < j \leq k$ ein kürzester Weg von v_i nach v_j.
Das bedeutet kurz gesagt, dass man für alle möglichen Teilpfade annimmt, dass sie optimal sind. Dass diese Annahme stimmt, lässt sich mittels Widerspruchbeweis nachweisen:
Man geht vom Gegenteil aus, nämlich dem Fall, dass neben dem Pfad p' von v_i nach v_j ein anderer, kürzerer Pfad p" von v_i nach v_j existiert. Dann müsste man

[1] OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S. 572, Z. 19-20
[2] Vorlesungsskript Algorithmendesign, Prof.Dr.Heinz Schmitz, Stand 6. April 2015, S.133 Z.8-13

p' durch p" ersetzen (denn p" ist kürzer). Damit würde sich auch der Gesamtpfad von v_0 nach v_k verkürzen. Da unser Pfad p von v_0 nach v_k bereits der kürzeste war, ist das ein Widerspruch. [3]

3.1.4 Funktionsweise des Algorithmus

Auf welche Weise findet man nun zur optimalen Lösung? Die Grundidee besteht darin, für jeden Knoten die Pfadkosten zu schätzen und die bestehende Lösungsmenge mit dem Knoten zu erweitern, welcher den geringsten Schätzwert aufweist.
Diese Arbeitsweise entspricht dem Greedy-Entwurfsmuster (engl. gierig): Nach einem Auswahlkriterium (die sogenannte Greedy-Regel) wird aus einer Menge das Element zur Erweiterung der Lösung gewählt, das den meisten Nutzen verspricht. In diesem Fall ist die Greedy-Regel das Verlangen nach dem kleinsten Schätzwert.
Dijkstra geht bei seinem Algorithmus folgendermaßen vor: Er unterteilt die Knoten des Graphen zunächst einmal in drei Gruppen: Die Knoten, deren Schätzwert bereits feststehen ("black nodes"[4]), deren Nachfolger ("grey nodes"[5] sowie unerreichbare Knoten, deren Schätzwerte gänzlich unbekannt sind ("white nodes"[6]).

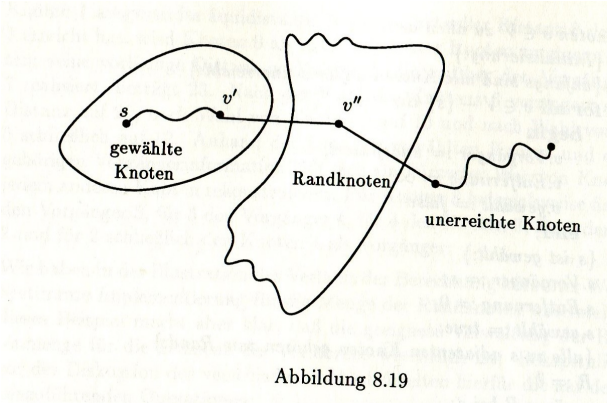


Abb. 3.1. Knotentypen (Quelle: OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S.573 Abb. 8.19)

Wenn keine grauen Knoten vorhanden sind, lässt sich die Lösungsmenge nicht erweitern.
Ansonsten muss aus den grauen Knoten einer ausgewählt werden, der schwarz werden soll, d.h. der Knoten, mit dem die Lösungsmenge erweitert wird. Alle vorherigen Pfade bestehen dabei ausschließlich aus schwarzen Knoten, d.h. Knoten die mit endgültig festgelegten Schätzwerten der Lösungsmenge bereits hinzugefügt wurden.

[3] OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990, S.573, Z.1-5
[4] DIJKSTRA Edsger W., FEIJEN W.H.J.: A Method of Programming,Cornwall: Addison-Wesley Publishing Company,1988, S.107 Z.24
[5] vergl. Ebd., S.108, Z.2-3
[6] vergl. Ebd., S.108, Z.3

Um zu bestimmen, welcher graue Knoten gewählt wird muss man die Kosten seines speziellen Pfades bestimmten [7], d.h. den Pfad ausgehend vom Startknoten s bis hin zur Kante vom letzten schwarzen Knoten zum aktuellen grauen Knoten. Der Knoten, dessen spezieller Pfad am kostengünstigsten ist, wird in die Menge der schwarzen Knoten aufgenommen und aus der Menge der grauen entfernt. Ist dieser Schritt vollzogen, müssen noch seine Nachfolger behandelt werden. Dabei gibt es für jeden Nachfolger drei Möglichkeiten:
Fall 1: Der Nachfolger ist weiß, d.h. er hat noch keinen Schätzwert. Dann wird dieser jetzt bestimmt und der Knoten wird grau.
Fall 2: Der Nachfolger ist grau. Dann muss verglichen werden, ob der aktuelle Schätzwert geringer ist als der vorherige, und wenn dies zutrifft, wird der Wert entsprechend aktualisiert. (Das ist immer dann der Fall, wenn man einen kostengünstigeren Pfad für den entsprechenden Knoten gefunden hat.)
Fall 3: Der Nachfolger ist schwarz. Dann steht sein Schätzwert bereits fest und er kann ignoriert werden.

3.1.5 Realisierung

Um dies umzusetzen, geht Dijkstra folgendermaßen vor: In einem Array werden die Farben der Knoten gespeichert. Ein weiteres Array dient der Speicherung der Schätzwerte.
Da der Knoten mit minimalen Pfadkosten nur aus den grauen gewählt wird, ist es sinnvoll, ein Feld für diese anzulegen, wobei eine Variable, die zugleich als Zeiger dient, deren Anzahl speichert.
Des weiteren werden zu jedem Knoten deren unmittelbare Vorgänger gespeichert, sodass sich der vollständige Pfad zu jedem Knoten zurückverfolgen lässt (Rekursion).
Eingegeben wird ein Graph zusammen mit Startknoten s und Zielknoten v. Ist v nicht von s aus erreichbar, sind die zurückgegeben Pfadkosten 0. Ansonsten wird die Länge zurückgegeben.
Das konkrete Vorgehen sieht nun folgendermaßen aus:

1. Initialisierung: Alle Knoten werden auf weiß gesetzt, der Startknoten s wird grau.
2. Erweiterung: Der Knoten mit minimalem Schätzwert wird aus der Menge der grauen Knoten ausgewählt und herausgenommen. Er wird auf schwarz gesetzt, sein Schätzwert ist nun endgültig.
3. Aktualisierung: Nun werden alle Nachfolger betrachtet. Deren Schätzwerte werden gegebenenfalls aktualisiert, wenn deren Pfadkosten nun geringer sind.

Der Algorithmus terminiert, wenn alle grauen Knoten abgearbeitet sind bzw. wenn der letzte Knoten schwarz ist.

[7] vergl. Ebd., S.108,Z.8-12

Natürlich muss man nicht exakt wie Dijkstra vorgehen und die entsprechenden Farbstufen speichern. Man kann den Algorithmus auch umsetzen, indem man die Schätzwerte in einem Array speichert und die Knoten in einer Menge Q. Am Anfang setzt man alle Schätzwerte auf undefiniert (das entspricht den weißen Knoten). Die Knoten aus Q, deren Schätzwert feststeht, werden aus Q entfernt (d.h. die schwarzen Knoten entsprechen V/Q). Alle übrigen Knoten in der Menge sind mit den grauen Knoten gleichzusetzen (vorläufige Schätzwerte). Das grundlegende Vorgehen ist bei einer solchen Implementierung gleich.

3.2 Komplexität

3.2.1 Ursprüngliche Implementierung

Das Initialisieren der Arrays benötigt jeweils $O(m)$ Schritte, wobei m die Kardinalität der Knotenmenge des Graphen ist. Die äußere Schleife arbeitet alle grauen Knoten ab und benötigt daher ebenfalls $O(m)$ Schritte. Das Bestimmen des Minimums braucht ebenfalls $O(m)$ Schritte. Das Aktualisieren der Nachfolger benötigt $O(deg(v))$ Schritte, also die Anzahl der ausgehenden Kanten des Knotens v, die maximal k beträgt (Gesamtzahl Kanten). Da in jedem Fall gilt, dass k ¡ m (jede Kante mündet in 2 Knoten), können wir dies vernachlässigen, ebenso wie die übrigen Operationen, die konstant sind. Insgesamt ergibt sich so eine Laufzeit von $O(m^2)$,[8] da m mal das Minimum in $O(m)$ bestimmt wird. Das ist bei dünnen Graphen, also Graphen mit wenig Kanten im Verhältnis zur Knotenzahl sehr ineffizient. Im Vergleich zu günstigeren Implementierungen ist die Laufzeit recht hoch.

3.2.2 Implementierung mit Heap

Die von Dijkstra vorgeschlagene Implementierung lässt sich verbessern, indem man zur Speicherung der grauen Knoten einen Heap einsetzt, denn in einem Heap lässt sich das Bestimmen des Minimums und dessen Entfernen in $O(logm)$ bestimmen. Das Einfügen eines Elementes beträgt ebenfalls $O(logm)$ Schritte.
Zwar ist das gezielte Löschen von Knoten mit veralteten Werten nicht möglich, sodass eventuell mehrere Knoten mit gleichen Werten existieren, dies ist jedoch nicht weiter schlimm, "wenn man für jeden Knoten nur die erste Entnahme dieses Knotens aus dem Heap beachtet und alle weiteren ignoriert".[9] Damit dies funktioniert, muss man speichern, welche Knoten bereits betrachtet wurden und welche nicht.
Jede Kante erhält maximal einen Eintrag im Heap, also enthält dieser nie mehr als k Kanten. Damit kostet das Einfügen $k * log(m)$ Rechenschritte, ebenso wie das Entfernen. Somit ergibt sich für die Implementierung mit Heap eine Laufzeit von $O(k * log(m))$[10], was den Algorithmus bei dünnen Graphen sehr effizient macht.

[8] OTTMANN, Thomas; WIDMAYER Peter: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70, Mannheim: BI-Wissenschaftsverlag, 1990,S.576,Z.13
[9] vergl. Ebd.S.576 Z.29-31
[10] vergl. Ebd., S. 576 Z.36

Bei sehr dichten Graphen mit $k = \Omega(m^2)$ schneidet der ursprüngliche Dijkstra dagegen besser ab.
Durch die Verwendung eines speziellen Heaps, dem Fibonacci-Heap, lässt sich eine Laufzeit von $O(k + m * log(m))$ [11] erreichen, wodurch die Implementierung mit einem Heap auch bei dichten Graphen effizienter ist.

3.3 Implementierung

```
1  from heapq import heappush, heappop
2
3  def dijkstra_pq(G,s):
4      m = len(G)                                  #O(1)
5      pq = []          #priority queue            #O(1)
6      d = [None]*m  #kosten                       #O(m)
7      p = [None]*m  #vorgaenger                #O(m)
8      d[s] = 0                                    #O(1)
9      heappush(pq, (0,s))                         #O(m)
10     while pq:                                   #O(m)
11         (_,v) = heappop(pq)                     #O(log m)
12         for u in G[v]:                          #O(deg(v)) --> O(k)
13             alt = d[v] + G[v][u]                #O(1)
14             if d[u]== None or alt < d[u]:       #O(1)
15                 d[u] = alt                   #O(1)
16                   p[u] = v                      #O(1)
17                 heappush(pq, (alt,u))           #O(log m)
18     return d,p
19
20 def shortest_path(s,v,p):
21    if v == None:
22       return []
23    else:
24       return shortest_path(s,p[v],p) + [v]
25
26 #Graph:
27
28 def define_G():
29     G = [    {1:1, 4:4, 2:2},    # Nachfolger von s
30              {3:3, 4:1},         # von u
31              {4:2, 5:3},         # von x
32              {},                 # von y
33              {3:1, 5:2},         # von v
34              {}                  # von z
35          ]
36     return G
37
38
39 G = define_G()
40 d,p = dijkstra_pq(G,0)
41 print( shortest_path(0,5,p))
42 print( shortest_path(0,3,p))
```

3.4 Anwendungsbereiche

Es stellt sich natürlich die Frage nach dem praktischen Nutzen des Dijkstra-Algorithmus. Diese Frage lässt sich gar nicht so schnell beantworten, da er vielseitig einsetzbar ist und in mehreren Bereichen Anwendung findet.

[11] vergl. Ebd., S.577 Z.10

Allgemein gesprochen ist dies überall, wo Routenplanung vonnöten ist.
Das kann z.B. Warentransport aller Art sein. Ganz gleich, ob man die Güter per Flugzeug, Bahn oder LKW transportiert, die Routen in dem jeweiligen Verkehrsnetz müssen entsprechend geplant werden, damit die Waren schnell und günstig am Zielort ankommen. Die Kosten einer Route müssen dabei nicht unbedingt die Entfernung an sich sein, auch andere Faktoren wie z.B. Mautkosten können hinzukommen. Diese spiegeln sich dann in den Kantenkosten wieder.
Was für Warentransport gilt, lässt sich auch auf den Personentransport übertragen. Wer eine möglichst günstige Reiseroute sucht, mit der er sein Reiseziel am schnellsten erreicht, kann diese mit Dijkstra errechnen.
Ein weiteres bedeutendes Anwendungsfeld ist Routing bei Rechnernetzen. Auf der Ebene der Vermittlungsschicht werden Pakete von der Quelle zum Ziel weitergeleitet. Dieser Weg besteht aus mehreren Teilstrecken, sogenannten Hops. [12] Zur Auswahl dieser Hops wird unter anderem Dijkstra verwendet.
Als Maß zur Bewertung der Teilstrecken kann die Anzahl der Teilstrecken gewählt werden, was allerdings sehr unpräzise ist, oder die physikalische Entfernung. Ein anderes Maß stellt die mittlere Übertragungszeit einer Teilstrecke für ein Paket dar. Die mittlere Übertragungszeit muss dazu regelmäßig aktualisiert werden. Im letzten Fall wäre der "kürzeste"Pfad der schnellste.[13] Allgemein werden die Kosten jedoch als "Funktion von Entfernung, Bandbreite, Durchschnittsverkehr, Übertragungskosten, gemessener Übertragungszeit und weiterer Faktoren"[14] berechnet.
DIJKTRA - ALGORITHMUS

[12] TANENBAUM,Andrew; WETHERALL, David: Computernetzwerke, München: Pearson Deutschland GmbH,2012, Kap. 5.2 Routing Algorithmen, S.420 Z. 1-7
[13] Ebd.,S.424, Z.1-5
[14] Ebd.,S.424 Z.6-8

4

Zusammenfassung

ZUSAMMENFASSUNG + AUSBLLICK

A

Glossar

DisASTer	DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms
DSM	Distributed Shared Memory
AC	Linearisierbarkeit (atomic consistency)
SC	Sequentielle Konsistenz (sequential consistency)
WC	Schwache Konsistenz (weak consistency)
RC	Freigabekonsistenz (release consistency)