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

Trier, 5. Juli 2015

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. Desweiteren wird auf den Algorithmus direkt eingegangen, mit einer einführenden Erklärung, der Klärung der Komplexität und anschließender Implementierung in Python Pseudo-Code. Abschließend 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.

In the course of this work, Dijkstra's algorithm is presented.
For this, a rough explanation of graphs and their representation with data structures are given. The next topic is the algorithm itself, explained by an introducing declaration, a discussion of its complexity and a subsequent implementation written in Python pseudo-code. Finally, its scopes of application are presented in order to illustrate the algorithm's current pertinence.

2

Graph

2.1 Definition Graph

Satz 2.1. *Ein **Graph** G besteht aus einer Menge X [deren Elemente Knotenpunkte genannt werden] und einer Menge U, wobei jedem Element $u \in U$ in eindeutiger Weise ein geordnetes oder ungeordnetes Paar von [nicht notwendig verschiedenen] Knotenpunkten, $x,y \in X$ zugeordnet ist. Ist jedem $u \in U$ 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 $u \in U$ 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.***
[Bie76, S. 9]

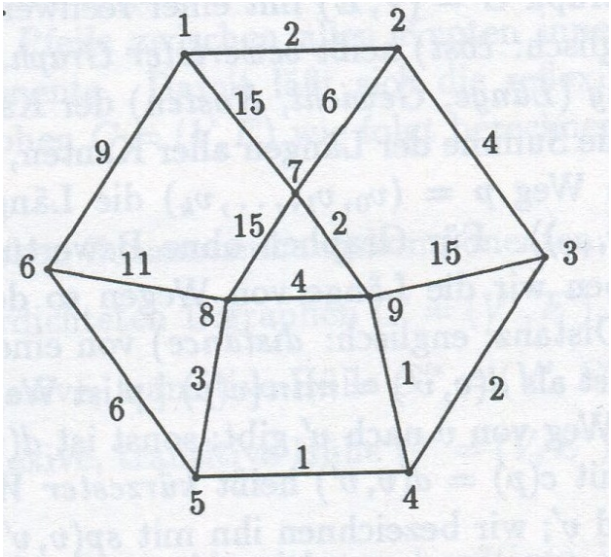


Abb. 2.1. Beispielgraph [TP90, S.572 Abb. 8.18]

Ein gerichteter Graph, auch Digraph genannt, besteht somit aus einer Menge aus geordneten Knotenpunkten [TP90, S. 590] . Dadurch, dass diese Punkte ge-

ordnet sind, also zu jedem Element U genau ein Paar Knotenpunkte zugeordnet wird, ist eine Verlaufsrichtung festgelegt, in welcher der Graph zeigt.

Die Verbindungslinien zwischen zwei Knotenpunkten werden Kanten genannt, welchen Kosten zugeordnet sind. Diese Kosten sind im Vornherein festgelegte, nicht negative Werte. Sie sind nicht mit der Kantenlänge zu verwechseln.

Somit kann sich eine geometrische Figur ergeben.

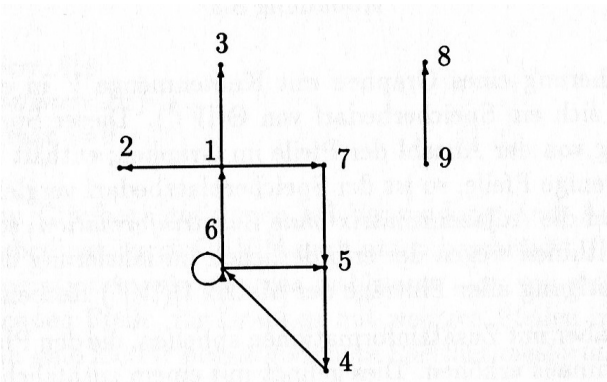
Weiterhin gibt es auch ungerichtete Graphen, welche jedoch in dieser Ausarbeitung keine Bedeutung haben und somit an dieser Stelle nicht weiter erläutert werden. In der folgenden Arbeit wird jedoch mit anderen Bezeichnungen gearbeitet. Anstelle von der von Gieß gebrauchten Bezeichnung G=(X,U), wird G=(V,E) verwendet.

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 mit wenig Kanten. [TP90, S. 539 - 541]

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. Vgl. [TP90, S.541-542 Speicherung in Adjazenzlisten]

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. Vgl. [TP90, S.542-544, Speicherung in einer doppelt verketteten Pfeilliste]



(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

Abb. 2.2. Adjazenzmatrix [TP90, S.539 Abb. 8.4]

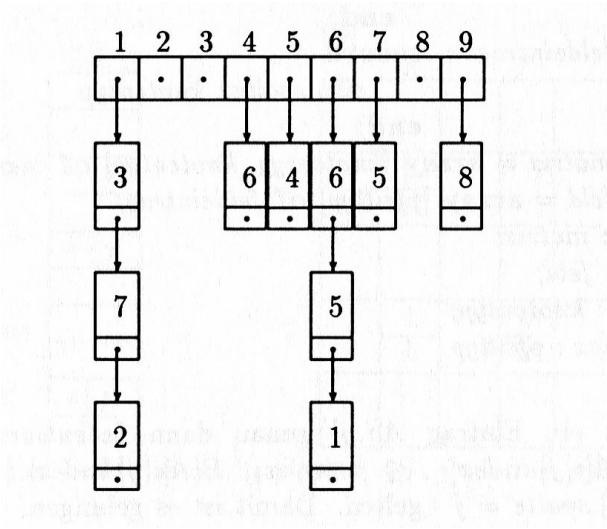


Abb. 2.3. Adjazenzliste [TP90, S542 Abb 8.6]

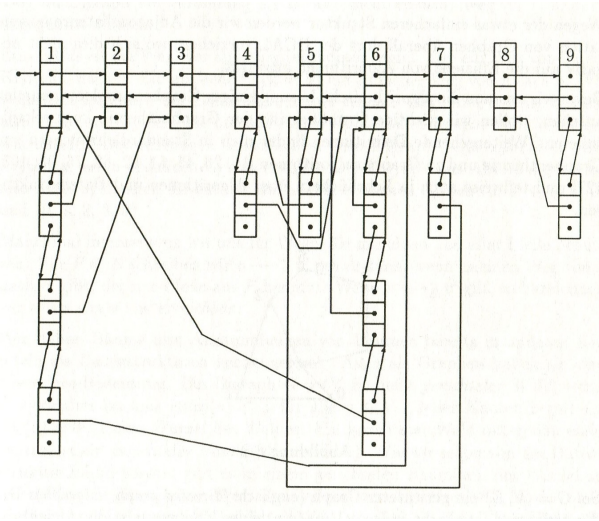


Abb. 2.4. doppeltVerketteтePfeilliste.jpg [TP90, S.543 Abb 8.7]

3

Dijkstra - 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. [TP90, S. 572, Z. 19-20] 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 für jede Kante {u,v} die Kanten (u,v) und (v,u) mit gleichen Kosten eingeführt werden. [1] Somit findet der Dijkstra-Algorithmus auch für ungerichtete Graphen Anwendung, solange sie vorher in einen Digraphen überführt werden.
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 für alle möglichen Teilpfade angenommen wird, dass sie optimal sind. Dass diese Annahme stimmt, lässt sich mittels Widerspruchbeweis nachweisen:
Wird vom Gegenteil ausgegangen, 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 p' durch

[1] Vorlesungsskript Algorithmendesign, Prof.Dr.Heinz Schmitz, Stand 6. April 2015, S.133 Z.8-13

p" ersetzen werden (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. [TP90, S. 573, Z.1-5]

3.1.4 Funktionsweise des Algorithmus

Auf welche Weise kann eine optimale Lösung gefunden werden? 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. Auf das Greedy-Prinzip wird in dieser Arbeit jedoch nicht tiefergehend eingegangen.
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"[W.88, S.107 Z.24] , deren Nachfolger ("grey nodes"Vgl. [TP90, S. 108, Z.2-3]) sowie unerreichbare Knoten, deren Schätzwerte gänzlich unbekannt sind ("white nodes"Vgl. [TP90, S. 108, Z.3]).

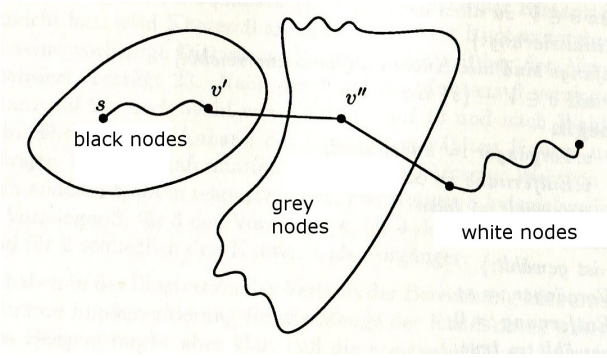


Abb. 3.1. Knotentypen Vgl. [TP90, S.573 Abb. 8.19]

Im folgenden werden zur besseren Lesbarkeit für „black nodes", „grey nodes"und „white nodes"jeweils deren deutsche Übersetzungen verwendet.
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.
Um zu bestimmen, welcher graue Knoten gewählt wird, müssen die Kosten seines speziellen Pfades bestimmt werden Vgl. [TP90, S.108,Z.8-12] , d.h. den Pfad aus-

gehend 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 ein kostengünstigerer Pfad für den entsprechenden Knoten gefunden wurde.
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.

Natürlich kann auch eine Alternative zu Dijkstras Vorgehen mit Speicherung der Farbstufen gewählt werden. Der Algorithmus kann auch umsetzen werden, indem die Schätzwerte in einem Array gespeichert werden und die Knoten in einer Menge Q. Am Anfang werden alle Schätzwerte auf undefiniert gesetzt, das entspricht

den weißen Knoten. Die Knoten aus Q, deren Schätzwert feststeht, werden aus Q entfernt, d.h. die schwarzen Knoten entsprechen der Differenz zwischen V und Q (V/Q). Alle übrigen Knoten in der Menge sind mit den grauen Knoten gleichzusetzen, welche jedoch erst vorläufige Schätzwerte darstellen. Das grundlegende Vorgehen ist bei einer solchen Implementierung gleich.

3.2 Komplexität

3.2.1 Ursprüngliche Implementierung

Der folgende Abschnitt bezieht sich direkt auf Dijkstras eigene Implementierung [W.88, S.110-111] , die von Ottman und Widmayer aufgegriffen wurde [TP90, S.573-574, Algorithmus] 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 zwei Knoten), kann dies vernachlässt werden, ebenso wie die übrigen verwendeten Operationen, die konstant sind. Insgesamt ergibt sich so eine Laufzeit von $O(m^2)$, [TP90, S.576 Z.13, Algorithmus] 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 zur Speicherung der grauen Knoten ein Heap eingesetzt wird, 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". Vgl. [TP90, S.576 Z.29-31, Algorithmus] Damit dies funktioniert, muss gespeichert werden, 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 $O(k*log(m))$ Vgl. [TP90, S. 576 Z.36] , was den Algorithmus bei dünnen Graphen sehr effizient macht. Bei sehr dichten Graphen, also Graphen mit vielen Kanten, 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 $O(k + m * log(m))$ [TP90, S. 577 Z.10] erreichen, wodurch die Implementierung mit einem Heap auch bei dichten Graphen effizienter ist.

3.3 Implementierung

Das folgende Codebeispiel bezieht sich auf die bereits beschriebene Implementierung mit Heap [2]

```
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.
Allgemein gesprochen ist dies überall, wo Routenplanung vonnöten ist.
Das kann z.B. Warentransport aller Art sein. Ganz gleich, ob der Transport der Güter per Flugzeug, Bahn oder LKW geschieht, die Routen in dem jeweiligen Verkehrsnetz müssen entsprechend geplant werden, damit die Waren schnell und

[2] Vgl. Abs. 3.2.2, Ïmplementierung mit Heap"

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. [TAN12, S.420 Z. 1-7] Zur Auswahl dieser Hops wird unter anderem Dijkstra verwendet.
Als Maß zur Bewertung der Teilstrecken kann deren Anzahl gewählt werden, was allerdings bedingt durch das Fehlen weiterer Informationen (beispielsweise Bandbreite) 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. [TP90, S.424, Z.1-5] Allgemein werden die Kosten jedoch als "Funktion von Entfernung, Bandbreite, Durchschnittsverkehr, Übertragungskosten, gemessener Übertragungszeit und weiterer Faktoren"[TP90, S.424 Z.6-8] berechnet.

4

Ausblick

ALT Routenplanung Algorithmen sind eine der Algorithmen, ohne die die Welt wie wir sie kennen nicht existieren könnte. Sie sind eine der Grundbausteine auf der die heutige Effizienz der Menschheit aufbaut!
Diese Effizienz kann man heutzutage nicht mehr vernachlässigen, es ist etwas das die heutige Menschheit ausmacht.

Dijkstra macht mit seinem Algorithmus unser Leben effizienter, stressfreier, erlebenswerter. Denn wer will schon lange auf die Verbindung zu einem Server warten, oder mit dem Auto einen Umweg fahren?
Die Vorteile, welche der Dijkstra-Algorithmus der Menschheit bietet, sind für uns schon längst Alltag. Sie sind für moderne Menschen essenziell und dies wird sich in der Zukunft auch nichtmehr ändern. Algorithmen welche das "Kürzeste Wege Problem"lösen können werden immer gebraucht werden.
Daher werden sich auch in der Zukunft Leute mit diesem Problem befassen und den Dijstra-Algorithmus benutzen bzw. möglicher weise auch einen neuen Algorithmus entwickeln.

NEU Es hat sich herausgestellt, dass der Dijkstra-Algorithmus trotz seiner Bedeutung von überschaubarer Komplexität ist und daher leicht verständlich ist. Dijkstras ursprüngliche Implementierung wurde im Laufe der Zeit im Hinblick auf Laufzeit und Codelänge bzw. Codekomplexität optimiert. Es ist abzusehen, dass auch in Zukunft noch Optimierungen stattfinden werden. Dabei ist das Grundprinzip jedoch immer gleich geblieben. Die Bedeutung und Gegenwart des Algorithmus ist uns zwar oft nicht bewusst, dennoch ist er aus dem modernen Alltag nicht mehr wegzudenken. Überall, wo Transportwege und Kommunikationsnetze im Spiel sind, ist Dijkstra gefragt. Wer will schon ewig warten, bis seine Ware geliefert wird? Oder bis eine Internetseite lädt? Man stelle sich einmal vor, wie langsam das Internet wäre, wenn die kürzesten Pfade nicht so schnell gefunden würden. Alles in allem lässt sich sagen, dass der Algorithmus gerade in der modernen Zeit, in der es oft auf Geschwindigkeit und Effizienz ankommt, unabdingbar ist.

Literaturverzeichnis

Bie76. Biess, Prof. Dr. G.: *Graphentheorie*. Verlag Harri Deutsch, Thun und Frankfurt/Main, 1976.

TAN12. TANENBAUM, Andrew; WETHERALL, David: *Computernetzwerke*. Pearson Deutschland GmbH,2012, 2012.

TP90. Thomas, Prof. Dr. Ottmann und Prof. Dr. Widmayer Peter: *Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70*. BI-Wissenschaftsverlag Mannheim/Wien/Zürich, 1990.

W.88. W., DIJKSTRA Edsger: *A Method of Programming*. Addison-Wesley Publishing Company, 1988.

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)