



# 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, 6. Juli 2015

---

## Kurzfassung

Jedes Mal, wenn physikalische Objekte oder Daten von einem Punkt zum anderen gelangen müssen, ist der kürzeste Pfad zwischen diesen Punkten gesucht. Die Schwierigkeit besteht nicht nur darin, diesen Pfad zu finden, sondern auch darin, diesen effizient zu finden, das heißt mit kleinstmöglichem Zeitaufwand. Dijkstra hat zu diesem Problem einen Algorithmus entwickelt, der beide Kriterien erfüllt und deshalb vielfach zum Einsatz kommt.

Aufgrund dessen Relevanz wird in dieser Arbeit ein Gesamtüberblick über diesen Algorithmus vermittelt. Dazu wird zunächst auf Graphen eingegangen, da der Algorithmus auf Graphen ausgelegt ist. Es wird erklärt, wie Graphen repräsentiert werden können, um die Eingabe nachzuvollziehen. Anschließend wird die Funktionsweise des Algorithmus erläutert und dessen Komplexität bezüglich Laufzeit beschrieben. Zur Veranschaulichung wird eine mögliche Implementierung als Codebeispiel gezeigt. Abschließend werden die vielfältigen Anwendungsbereiche beschrieben, um zu verdeutlichen, weshalb der Algorithmus unverzichtbarer Bestandteil des heutigen Zeitalters ist.

When two physical objects or data have to be transmitted from one point to another, it is necessary to find the shortest path between these points. The difficulty of this task is not only finding this path but also finding it in an efficient way, which means as quick as possible. For this problem, Dijkstra has developed an algorithm which meets both criteria. Therefore it is used in many cases.

Due to its relevance, the purpose of this work is to give an overview of Dijkstra's algorithm. First, graphs are discussed because the algorithm is based on graphs. It is explained how they are represented in order to understand the algorithm's input. Afterwards, it is exemplified how it works and the complexity of its runtime. A possible implementation as an example of code helps to illustrate the content. Finally, the multiple scopes of application are described to show why it is an essential part of our modern age.

---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung und Problemstellung</b>	<b>1</b>
<b>2</b>	<b>Graph</b>	<b>2</b>
2.1	Definition Graph	2
2.2	Datenstrukturen zur Repräsentation von Graphen	3
<b>3</b>	<b>Dijkstra - Algorithmus</b>	<b>6</b>
3.1	Erklärung	6
3.1.1	Das Shortest-Path Problem	6
3.1.2	Zur Eingabe	6
3.1.3	Das Optimalitätsprinzip	6
3.1.4	Funktionsweise des Algorithmus	7
3.1.5	Realisierung	8
3.2	Komplexität	9
3.2.1	Ursprüngliche Implementierung	9
3.2.2	Implementierung mit Heap	9
3.3	Implementierung	10
3.4	Anwendungsbereiche	10
<b>4</b>	<b>Ausblick</b>	<b>12</b>
	<b>Literaturverzeichnis</b>	<b>13</b>
	<b>Glossar</b>	<b>14</b>

## Einleitung und Problemstellung

Wir leben in einer modernen, schnelllebigen Welt. Die Globalisierung hat jeden Kontinent verbunden und sowohl Marktwirtschaft als auch Tourismus florieren mehr denn je. Doch in solch einer Welt gibt es auch viele logistische Herausforderungen. Waren sollen ausgeliefert werden, Passagierflugzeuge müssen mit passenden Routen ausgestattet werden, Touristen wollen mit dem Auto oder per Bahn an ihre Urlaubsziele kommen.

Doch wie ist ein solcher Aufwand am effizientesten zu bewältigen? Hierfür hat der Mathematiker Dijkstra bereits in den Fünfziger Jahren einen Algorithmus entwickelt, der noch heute in der Praxis Anwendung findet.

In dieser Arbeit geht es um genau diesen Algorithmus.

## 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 Graph **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]

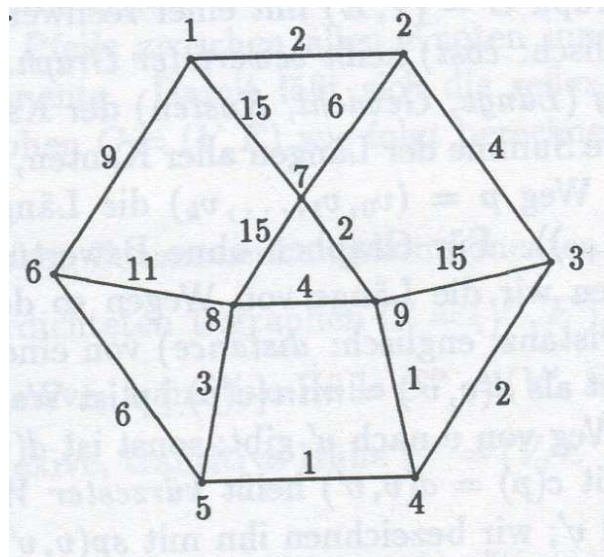


Abb. 2.1. Beispielgraph [TP90, S.572 Abb. 8.18]

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  $m \times m$  Matrix gespeichert ( $m = \#V$ ), 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]

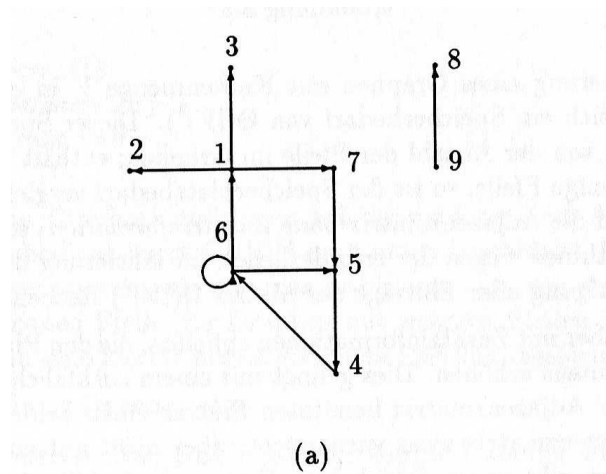
### Speicherung in Adjazenzenlisten

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 Adjazenzenlisten]

### Speicherung in doppelt verketteten Listen

Jedes Element enthält Zeiger auf die beiden Nachbarelemente sowie auf eine Kantenliste (wie bei Adjazenzenliste, s.o.). Diese Darstellung besitzt die den Adjazenzenlisten fehlende Dynamik, ist aber natürlich komplizierter. Vgl. [TP90, S.542-544, Speicherung in einer doppelt verketteten Pfeilliste]



	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]

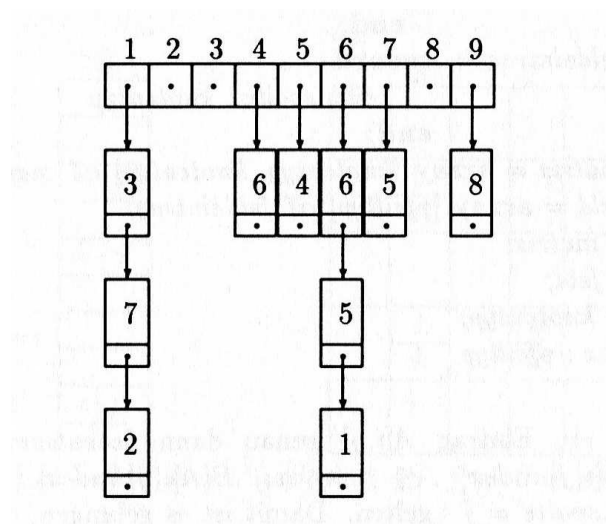
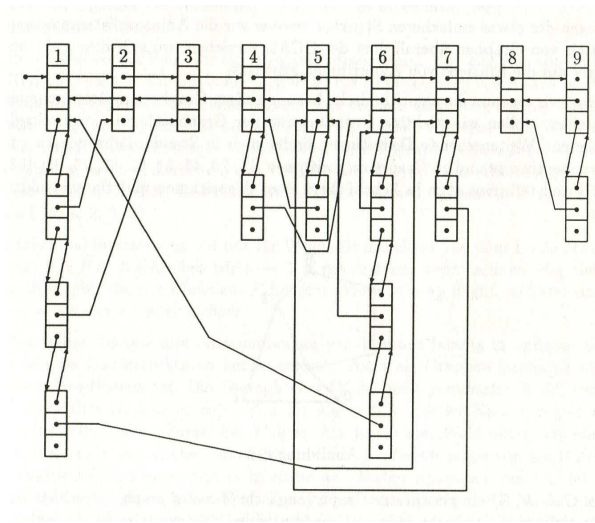


Abb. 2.3. Adjazenzliste [TP90, S.542 Abb. 8.6]



**Abb. 2.4.** Doppelt verkettete Pfeilliste[TP90, S.543 Abb 8.7]



## 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.<sup>1</sup> 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  $p=(v_0,v_1,\dots,v_k)$  von  $v_0$  nach  $v_k$  ist jeder Teilweg  $p'=(v_i,\dots,v_j)$  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 Widerspruchsbeweis 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

---

<sup>1</sup> Vorlesungsskript Algorithmen-Design, Prof.Dr.Heinz Schmitz, Stand 6. April 2015, S.133 Z.8-13

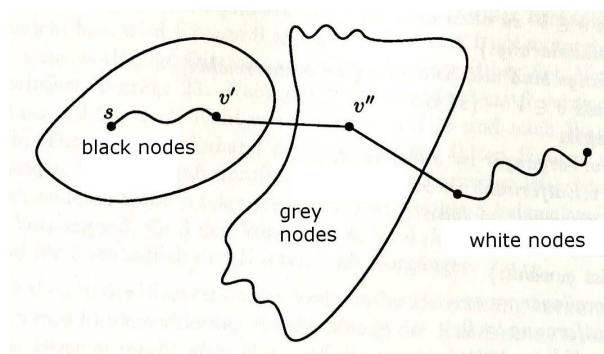
$p''$  ersetzt werden (denn  $p''$  ist kürzer). Damit würde sich auch der Gesamtpfad von  $v_0$  nach  $v_k$  verkürzen. Da unser Pfad  $p$  von  $v_0$  nach  $v_k$  bereits der kürzeste war, ist das ein Widerspruch. [TP90, S. 573, Z.1-5]

### 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 tiefer gehend 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 umgesetzt 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 vernachlässigt 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  $O(\log m)$  bestimmen. Das Einfügen eines Elementes beträgt ebenfalls  $O(\log m)$  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 <sup>2</sup>

```

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 = [
30         {1:1, 4:4, 2:2}, # Nachfolger von s
31         {3:3, 4:1},      # von u
32         {4:2, 5:3},      # von x
33         {},               # von y
34         {3:1, 5:2},      # von v
35         {}                # von z
36     ]
37     return G
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

<sup>2</sup> Vgl. Abs. 3.2.2, "Implementierung 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.

## Ausblick

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)

# Stellungnahme zu den Reviews

Von Gr. 13 zu G11 B3

1. Deckblatt: Ort, Abgabedatum durch Ort und Abgabedatum ersetzen, selbst wenn dies nur eine digitale Ausarbeitung ist  
A: geändert
2. Einleitung und Problemstellung:
  - (a) Keine/wenig Motivation, gleicht eher einer Kurzbeschreibung  
A: Neu geschrieben
  - (b) Es wurde nicht auf die Problemstellung eingegangen  
A: Neu geschrieben
  - (c) Seite 1, erster und zweiter Satz: zweimal folgenden, vielleicht umschreiben  
A: Neu geschrieben
  - (d) Letzter Satz zu geschachtelt, in mehrere Sätze aufteilen  
A: Neu geschrieben
3. Graph
  - (a) Definition Graph
    1. Erster "Satz": "-Für Dijkstra auf gerichtete Graphenausgelegt." Vielleicht als normalen Satz im Text schreiben. Ist die Definition darauf ausgelegt? Soll das nur andeuten, dass es noch andere/allgemeinere Definitionen von Graphen gibt?  
A: Umgeschrieben
    2. Erster und Zweiter Absatz: Was bedeutet hier "geordnetes" bzw. "ungeordnetes Paar von Knoten"? Kurz erläutern oder Verweis/Fußnote einfügen  
A: In Satz 2.1 definiert
    3. Dritter Absatz: Ein gerichteter Graph besteht außerdem aus 2 Mengen, hier ist nun nur die Rede von einer. Was stimmt nun?  
A: Das hängt von der Darstellungsweise ab. Wenn man den Graph als  $G=(V,E)$  aufschreibt, besteht der Graph aus zwei Mengen, da die Kantenmenge  $E$  jedoch eindeutig beschreibt, wie die Knoten einander zugeordnet werden, kann man den Graphen auch als eine Menge betrachten: Einander zugeordnete Punktepaare, d.h. eine Menge aus geordneten Punktepaaren wie im Text beschrieben. Deswegen wurde dies auch nicht geändert.
    4. Dritter Absatz: "[Grafik hier einfügen]"  
A: Verbessert, Grafik eingefügt
    5. Vierter Absatz: "Wenn man diese Punkte als Richtungen nun als Straßennetz sähe, würde sich durch die Punkte ein -

Straßenverlauf abbilden mit unterschiedlichen Verlaufsrichtungen." Ist das "als Richtungen" zu viel? Zur besseren Lesbarkeit umschreiben z.B. in: "Wenn man diese Punkte nun als Straßennetz sähe, würde sich durch diese ein Straßenverlauf mit unterschiedlichen Verlaufsrichtungen ergeben."

A: Kommt nicht mehr vor

(b) Datenstrukturen zur Repräsentation von Graphen:

1. Erster Satz: Andere Bezeichnungen beim Graph, warum nicht bei  $G = (X,U)$  bleiben? Oder direkt  $G = (V,E)$  verwenden!

A: Hat sich bei uns eingeschlichen und haben es seitdem nicht gemerkt. Jetzt ist es sehr umständlich dies wieder zu entfernen. Darum Erklärung eingefügt, dass es sich bei beidem um das Gleiche handelt.

2. Erster Satz: "...mit 1 falls  $(i, j) \in E$  und 0 falls nicht." Was ist hier 1 oder 0?

A: Bezeichner die hier so festgelegt werden.

3. Abb. 2.1. kurz im Text erwähnen, z.b. am Ende des Abschnitts "Speicherung in einer Adjazenzmatrix" mit (siehe Abb. 2.1.).

A: Neu 2.2, kommt direkt nach der Erklärung zur Veranschaulichung, Bezug ist unserer Meinung nach klar gegeben

4. selbiges gilt für alle weiteren vorkommenden Grafiken

A: selbe Antwort wie 3.

5. Abb. 2.1. Die Quelle wie in der Vorlage angeben und ins Literaturverzeichnis verweisen.

A: gemacht

6. selbiges gilt für die Quelle am Ende des Abschnitts "Speicherung in doppelt verketteten Listen" und alle weiteren vorkommenden Quellen

A: gemacht

4. Dijkstra-Algorithmus: s in der Überschrift fehlt

A: korrigiert

(a) Erklärung:

1. Im ersten Abschnitt ("Das Shortest-Path Problem") wird nun zum ersten Mal von Kosten geredet. Diese sollten evtl. in der Definition kurz erläutert werden.

A: geändert

2. Im zweiten Abschnitt ("Zur Eingabe") dasselbe mit "Digraphen". Diese kurz zu erwähnen/erläutern würde nicht schaden.

A: Digraphen wurden schon definiert

3. Abb. 3.1.: Wenn schon Begriffe wie "black nodes", "grey

nodes" und "white nodes" eingeführt werden, warum dann nicht auch in der Grafik verwenden?

A: geändert

4. Im Folgenden wird nun einfach die deutsche Übersetzung dieser Begriffe verwendet. Mir fehlt hierbei der Übergang, also entweder eine kurze Übersetzung oder beim englischen bleiben.

A: Übergang eingefügt (Erklärung, dass ab jetzt deutsche Übersetzungen verwendet werden)

5. Seite 7: "..wird, muss" Komma einfügen

A: Text geändert

6. Seite 7: Die Auflistung der 3 Fälle könnte man eventuell. schöner mit der von LaTeX gegebenen Aufzählung realisieren.

A: Geschmackssache, daher nicht geändert

7. Beim Aufzählen der Schritte beim konkreten Vorgehen im Abschnitt "Realisierung" dasselbe

A: Siehe Punkt 6.

8. Seite 8: erster Absatz: "...entsprechen  $V/Q$ ": woher kommt nun das  $V$ ? Soll  $V/Q$  den Durchschnitt von  $V$  und  $Q$  darstellen? Hier bitte das richtige Zeichen " $V \cap Q$ " verwenden.

A: Gemeint ist nicht der Durchschnitt, sondern die Differenz. Dies wird jetzt auch im Text erklärt.

(b) Komplexität:

1. Was bedeutet " $k \mid m$ "? Rauskopiert ergibt sich " $k < m$ "

A: Wurde korrigiert

2. Welche übrigen Operationen? Hier hätte auch auf Grafiken zu Verschaulichung zurückgreifen können. A: Alle nicht beschriebenen Operationen, durch Codebeispiel nachvollziehbar. Es wäre viel zu aufwändig, jeden Schritt zu beschreiben.

3. Dichte Graphen werden nicht erläutert, dünne Graphen schon. Aus dem Kontext ergibt sich zwar deren Bedeutung, doch eine kurze Erwähnung im Abschnitt "Ursprüngliche Implementierung" (wie bei dünnen Graphen") wäre meiner Ansicht nach hilfreich.

A: Ist das Gleiche, nur die Anzahl der Kanten ist verschieden

(c) Implementierung:

1. Auch hier sollte eine kurze Beschreibung zum Code hin. Nicht unbedingt um eben diesen im einzelnen zu erklären, was auch nicht schaden würde, aber zumindest welche Implementierung das nun ist. Anscheinend ist es die, wie sie in Abschnitt 3.2.2 beschrieben wird, das vielleicht kurz dazuschreiben.

A: geändert

(d) Anwendungsbereiche:

1. Anscheinend behandelt jeder Abschnitt ein Thema. Am Ende jedoch wird in dem letzten Absatz das Thema des vorigen Absatz weiterbehandelt. Hier könnte man einheitlich bleiben und die beiden letzten Absätze zu einem zusammenfügen.

A: geändert

## Von Gr. 13 zu G17 B3

Im Punkt Einleitung und Problemstellung:

1. wird nicht auf die Problemstellung eingegangen

A: neu geschrieben

2. Im ersten und zweiten Satz jeweils einmal „folgenden“ „in der folgenden... wird im folgenden...“.

A: neu geschrieben

3. Wirkt etwas aufgesetzt, „hierzu wird im folgenden... danach wird... zum schluss wird...“

A: Neu geschrieben

Graph

1. Zwischen dem Kapitel 2 und dem Unterkapitel 2.1 steht kein Text.

A: finden wir nicht schlimm

2. Dritter, vierter, fünfter Satz: „wir“ entfernen.

A: überarbeitet

3. Was bedeutet geordnet und ungeordnet?

A: Definition 2.1

4. Im Unterkapitel 2.1 fehlt eine Grafik „[Grafik hier einfügen]“

A: überarbeitet

5. Vorletzter Satz: „Wenn man diese Punkte... würden sich durch die Punkte...“ zweimal Punkte liest sich komisch. Das „man“ sollte entfernt werden.

A: geändert

6. Letzter Satz ist zu sehr geschachtelt. „man“ Entfernen.

A: überarbeitet

7. Speicherung in einer Adjazenzmatrix: Auf den Nachteil sollte etwas genauer bezüglich des „warum“ eingegangen werden.

A: Nachteil wird genannt

8. Speicherung in Adjazenzlisten: Letzter Satz, warum wird kein

Speicherplatz verschwendet?

A: Siehe Satz davor

9. Speicherung in doppelt verketteten Listen: Sollte weiter drauf eingegangen werden. Am Ende steht die Quelle?

A: überarbeitet

10. Seite 2: Quelle enthält kein Auflage / Jahr.

11. A: Jahr hinzugefügt

Dijkstra - Algorithmus:1.

1. Überschrift „Dijkstra -Algorithmus“, der „-„ müsste nicht sein.

A: Ist aber auch nicht falsch finden wir

2. Zwischen Kapitel 3 und Unterkapitel 3.1 kein Text.

A: Finden wir nicht schlimm

3. Das Shortest-Path Problem: Zweiter Satz: „man“ entfernen

A: Warum ?

4. Das Shortest-Path Problem: Letzter -und vorletzter Satz: Was sind Kantenkosten? Was sind Pfadkosten? Das Gleiche?

A: Nein, das geht aber aus dem Text hervor

5. Zur Eingabe: Was sind Digraphen?

A: siehe Kapitel Graphen

6. Zur Eingabe: Zweiter Satz: „man“ entfernen. Zu sehr geschachtelt.

A: geändert

7. Zur Eingabe: Dritter Satz: „man“ entfernen

A: In der neuen Version nicht mehr vorhanden

8. Das Optimalitätsprinzip: Dritter, fünfter, sechster Satz: „man“ entfernen!

A: In der neuen Version nicht mehr vorhanden

9. Funktionsweise des Algorithmus: Erster Satz: „man“ entfernen!

A: In der neuen Version nicht mehr vorhanden

10. Funktionsweise des Algorithmus: Auf Greedy-Entwurfsmuster eingehen.

A: Wird nur erwähnt, da es nicht Hauptthema der Arbeit ist

11. Funktionsweise des Algorithmus: 1. Absatz, zweiter Satz ist lang sagt aber nicht viel. Aus welcher Menge? Was für eine Lösung? Was soll der Nutzen sein?

A: Überarbeitet

12. Funktionsweise des Algorithmus: „Die Knoten, deren Schätzwert... Nachfolger ("grey..." Klammer wieder schließen.  
A: Abschnitt wurde überarbeitet
13. Funktionsweise des Algorithmus: Es wird in einer Klammer von grey nodes gesprochen, danach von grauen Knoten.  
A: „grey nodes“ kommt von anderen, graue Knoten unserer Bezeichner
14. Seite 6: Letzter Satz zu verschachtelt. Letzten beiden Quellen in Fußzeile schwer zu identifizieren.  
A: Literaturverzeichnis angelegt
15. Seite 7: Erster Satz „man“ entfernen. Quelle in Fußzeile schwer zu identifizieren.  
A: Literaturverzeichnis angelegt
16. Funktionsweise des Algorithmus: Dieses Kapitel benötigt dringend weitere Bilder zu Erklärung!  
A: Finden wir nicht
17. Funktionsweise des Algorithmus: Vorletzter Satz „man“ entfernen.  
A: Kommt in der neuen Version nicht vor
18. Seite 8: Erster, zweiter, dritter Satz: „man“ entfernen.  
A: Kommt in der neuen Version nicht vor
19. Zwischen Kapitel 3.2 und 3.2.1 kein Text.  
A: finden wir nicht schlimm
20. Ursprüngliche Implementierung: Fünfer Satz „verkehrtes !“, Zahl 2 ausschreiben. „wir“ entfernen.  
A: Abgeändert
21. Implementierung mit Heap: Erster Satz „man“ entfernen.  
A: Kommt in der neuen Version nicht vor
22. Implementierung mit Heap: Dritter Satz zu sehr geschachtelt, „man“ entfernen!
23. Implementierung mit Heap: Vierter Satz „man“ entfernen.  
A: finden wir ok an der Stelle
24. Implementierung mit Heap: Abbildungen sollten hinzugefügt werden.

- A: Wir finden der Text reicht hier
25. Implementierung: In das Kapitel „Implementierung mit Heap“ einfügen. Und etwas zum Code schreiben.
- A: Der Algorithmus wurde davor besprochen, seine Arbeitsweise und Charakteristiken sollten dem Leser somit klar sein. Das Verständnis des Code-Beispiels sollte daher gegeben sein.
26. Seite 10: Zweiter Satz: „man“ entfernen.
- A: In der neuen Version nicht mehr vorhanden
27. Seite 10: Letzter Absatz, erster Satz: Zweimal „Teilstrecken“
- A: geändert
28. Anwendungsbereiche: Sollte eine Abbildung bekommen.
- A: Können wir nicht nachvollziehen warum, bzw. finden wir nicht.

Ausblick:

Allgemeine Antwort: Wurde neu geschrieben! Daher können diese Anmerkungen vernachlässigt werden.

1. Erster Satz: zweimal „Algorithmen“. „Routenplanung Algorithmen sind eine der Algorithmen...“ hört sich nicht gut an.
2. Zweiter Satz: „Diese Effizienz kann man heutzutage nicht mehr vernachlässigen, es ist etwas das die heutige Menschheit ausmacht.“ Diesen Satz unbedingt nochmal überdenken.
3. Vorletzter Satz: Hinter „können“ ein Komma. „werden“ entfernen.
4. Sollte durch eine Quelle belegt werden auch wenn es sehr wahrscheinlich stimmt.