



## 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

<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	2
<b>3</b>	<b>Dijkstra - Algorithmus</b>	<b>5</b>
3.1	Erklärung	5
3.2	Komplexität	6
3.3	Implementierung	6
3.4	Anwendungsbereiche	6
<b>4</b>	<b>Zusammenfassung</b>	<b>8</b>
	<b>Glossar</b>	<b>9</b>

## 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.

## 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  $u \in U$  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**.<sup>1</sup>

Ein gerichteter Graph besteht somit aus einer Menge aus geordneten Knotenpunkten. Dadurch, dass 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

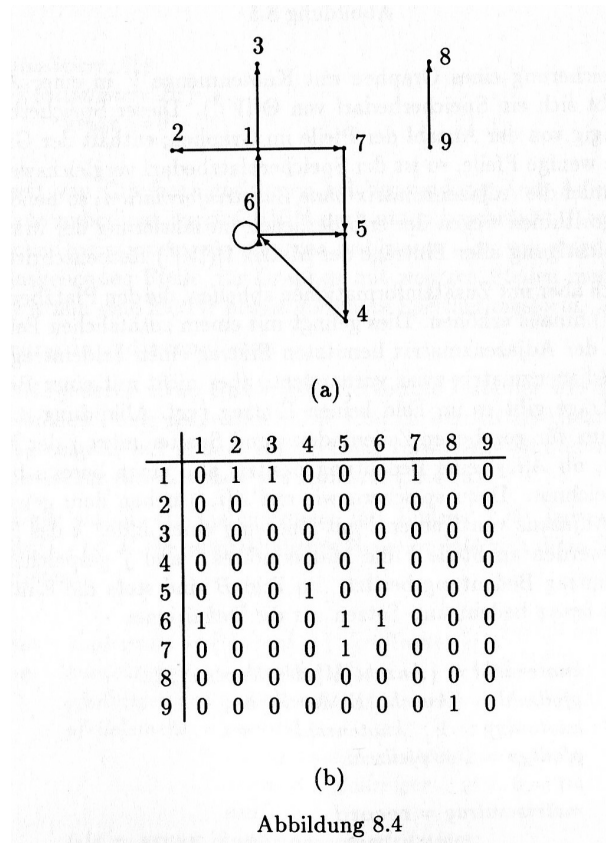
Der Graph  $G=(V,E)$  wird in einer booleschen  $m \times m$  Matrix gespeichert ( $m = \#V$ ), mit 1 falls  $(i, j) \in E$  und 0 falls nicht.

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

---

<sup>1</sup> Bieß, Graphentheorie, S.9

mit wenig Kanten.



**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

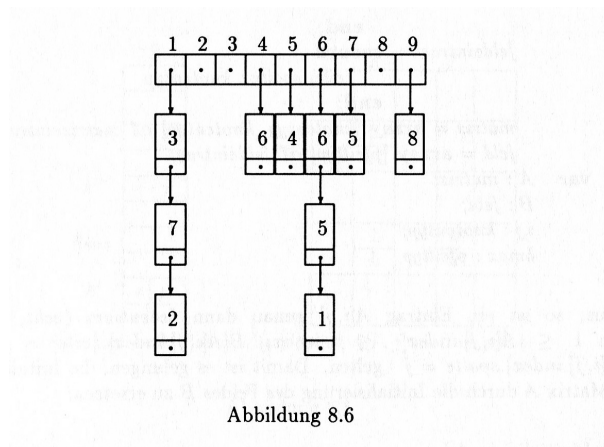
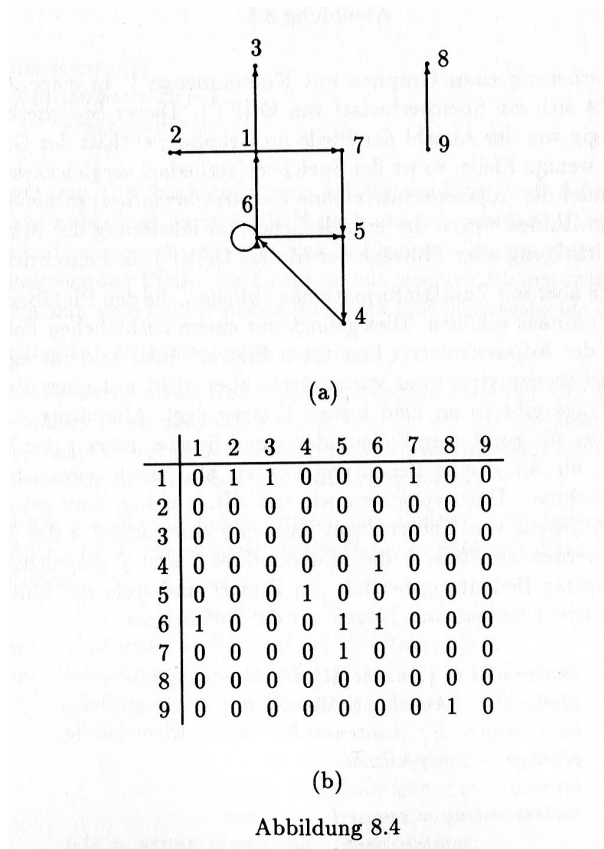


Abbildung 8.6

**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)



**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)

## Dijkstra - Algorithmus

### 3.1 Erklärung

-löst das Shortest-Path Problem (oder Single-Source Shortest Path Problem) [Def Problem einfügen] -Problemstellung: Wie finde ich den kürzesten Pfad von meinem Startknoten  $s$  zu einem Zielknoten  $u$ ? Mit kurz ist nicht die Länge gemeint, sondern die Minimierung der Pfadkosten! -Eingabe sind Digraphen (ungerichtete Graphen lassen sich leicht in Digraphen umwandeln, indem man zu jeder Kante  $(u,v)$  eine Kante  $(v,u)$  mit gleichen Kosten hinzufügt) -Annahme: für jeden Knoten existiert auch ein Pfad -funktioniert nach dem Entwurfsprinzip Greedy

kurze Erklärung zu Greedy-Algorithmen: -greedy = engl. gierig - zu Beginn ist Teillösung  $T = \emptyset$  -im nächsten Schritt werden alle möglichen Erweiterungen in Betracht gezogen und nach ihrem Nutzen bewertet. Zur Bewertung wird ein Greedy-Kriterium formuliert. -Die Erweiterung, die den maximalen Nutzen bringt, wird ausgewählt und die Lösung damit erweitert. - Eine einmal durchgeführte Erweiterung wird nicht mehr rückgängig gemacht! Deshalb sind Greedy-Algorithmen sehr effizient. -Die durch sukzessive Erweiterung erzielte Lösung ist optimal.

Dijkstra -Grundidee: Pfadkosten  $d(u)$  für jeden Knoten  $u$  schätzen aus einer Menge  $Q$  (alle restlichen Knoten). Nach jeder Erweiterung werden die Schätzungen aktualisiert

-genaueres Vorgehen: 1. Initialisierung: Man beginnt mit  $Q = V$ , d.h. alle Knoten sind enthalten,  $d(s) = 0$  (Kosten des Startknoten auf 0 setzen). Alle anderen  $d(u)$  werden auf undefiniert gesetzt, weil deren Schätzwerte noch nicht bekannt sind. 2. Erweiterung: Jetzt wird die Greedy-Auswahlregel verwendet, d.h. man sucht in  $Q$  nach dem Knoten mit dem kleinsten Schätzwert und entfernt diesen aus  $Q$ . (Streng genommen wird die Lösung nicht erweitert, da die Lösungsmenge nicht explizit konstruiert wird.  $T$  entspricht  $V/Q$ ) 3. Aktualisierung: Nachfolger von  $v$  durchgehen und deren Schätzwerte  $d(u)$  ggf. aktualisieren, falls diese nun schneller erreichbar sind.

-wenn man zu jedem Knoten  $u$  den Vorgänger  $p(u) = v$  in einer Liste speichert, lässt sich so zu jedem Knoten der kürzeste Pfad finden ( $s, \dots, p(p(u)), p(u), u$ )



## 3.2 Komplexität

$O(m^2)$  bei schlechter Implementierung - bei Implementierung mit Priority-Queue in geeigneter Datenstruktur (z.B. `heapq` in Python) lässt sich eine deutlich bessere Laufzeit von  $O(k \cdot \log m)$  erreichen [ $k$  = Kantenzahl des Graphen]

## 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 = [
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

Der Dijkstra-Algorithmus wird noch heute oft verwendet und bietet einen großen Nutzen beispielsweise im Verkehrswesen. So wird er oft für die Routenplanung von Warentransporten verwendet, aber auch für die Streckenberechnungen von Zügen und Flugzeugen. Hier bedeutet jeder Umweg höhere Transportkosten, da Treibstoff und Fahrzeit steigen.

Doch auch in Kommunikationsnetzwerken findet der Weg-Findungs-Algorithmus von Dijkstra noch immer Verwendung. Zwischen Rechnernetzen findet Routing statt. Hier kosten Umwege wertvolle Kapazitäten, welche sehr begrenzt sind. Somit ist es sinnvoll, die kürzesten Kommunikationswege zu finden, damit die Kapazitäten nicht unnötig belastet werden.

-Informationsnetzwerke (Anzahl Links Domain Suchmaschine)

DIJKTRA - ALGORITHMUS

## Zusammenfassung

ZUSAMMENFASSUNG + AUSBLICK

# 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)