



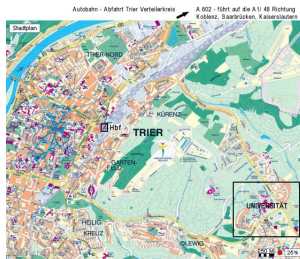
Funktionsprinzip und Anwendungsbeispiele des Dijkstra-Algorithmus

Annika Kremer, Thomas Jürgensen, Tobias Meier

Fachhochschule Trier

22. Juli 2015

Problemstellung



(Quelle:

(Quelle: <http://www.vrt-info.de/images/>

liniennetzplan_trier_2014.png)

<http://www.uni-trier.de/index.php?id=27631>)

Table of content

Problemstellung

Graphendefinition

Dijkstra - Algorithmus

Komplexität

Implementierung

Anwendungsbereiche

Quellen

Graphendefinition

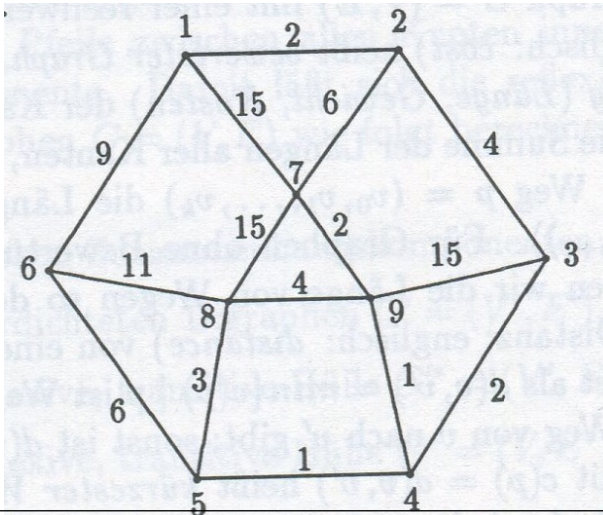
Ein **Graph** G besteht aus einer Menge X [deren Elemente Knotenpunkte genannt werden] und einer Menge U , wobei jedem Element $u \in U$ in eindeutiger Weise ein geordnetes oder ungeordnetes Paar von [nicht notwendig verschiedenen] Knotenpunkten, $x, y \in X$ zugeordnet ist. Ist jedem $u \in U$ ein geordnetes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

(Quelle: Bieß, Graphentheorie)

Graphendefinition

Beispiel



Erweiterung der Knoten um ...

Dijkstra - Algorithmus

Erweiterung der Knoten um ...

- ▶ einen Schätzwert.

Dijkstra - Algorithmus

Erweiterung der Knoten um ...

- ▶ einen Schätzwert.
- ▶ einen Vorgänger.

Dijkstra - Algorithmus

Erweiterung der Knoten um ...

- ▶ einen Schätzwert.
- ▶ einen Vorgänger.

Einschränkung der Kanten

Dijkstra - Algorithmus

Erweiterung der Knoten um ...

- ▶ einen Schätzwert.
- ▶ einen Vorgänger.

Einschränkung der Kanten

Eine Kante darf nur ein positives Gewicht besitzen.

Dijkstra - Algorithmus

white node

Ein “white node” ist ein Knoten über den weder ein Schätzwert noch ein Vorgänger bekannt ist.

Dijkstra - Algorithmus

white node

Ein “white node” ist ein Knoten über den weder ein Schätzwert noch ein Vorgänger bekannt ist.

grey node

Ein “grey node” ist ein Knoten zu dem bereits ein Weg gefunden wurde. Er besitzt also einen Schätzwert und Nachfolger. Diese müssen aber noch nicht optimal sein.

Dijkstra - Algorithmus

white node

Ein “white node” ist ein Knoten über den weder ein Schätzwert noch ein Vorgänger bekannt ist.

grey node

Ein “grey node” ist ein Knoten zu dem bereits ein Weg gefunden wurde. Er besitzt also einen Schätzwert und Nachfolger. Diese müssen aber noch nicht optimal sein.

black node

Ein “black node” ist ein Knoten zu dem bereits der optimale Weg gefunden wurde.

Dijkstra - Algorithmus

Berechnung von Knotenwerten

Der Schätzwert S_A eines Knotens K_A kann berechnet werden, wenn es einen Knoten K_V gibt, dessen Schätzwert S_V bekannt ist und eine gerichtete Kante von K_V nach K_A mit bekannten Gewichtung g existiert.

Berechnung: $S_A = S_V + g$

Initialisierung

Initialisierung

- ▶ Der Startknoten ist ein “black node” mit dem Schätzwert 0.
Einen Vorgänger besitzt der Startknoten nicht.

Initialisierung

- ▶ Der Startknoten ist ein “black node” mit dem Schätzwert 0. Einen Vorgänger besitzt der Startknoten nicht.
- ▶ Alle seine Nachfolgerknoten werden berechnet. Sie sind somit alle “grey nodes”.

Schritt

Dijkstra - Algorithmus

Schritt

- ▶ Der “grey node” mit dem niedrigsten Schätzwert wird gesucht.

Dijkstra - Algorithmus

Schritt

- ▶ Der “grey node” mit dem niedrigsten Schätzwert wird gesucht.
- ▶ Dieser “grey node” wird ab sofort als “black node” angesehen.

Schritt

- ▶ Der “grey node” mit dem niedrigsten Schätzwert wird gesucht.
- ▶ Dieser “grey node” wird ab sofort als “black node” angesehen.
- ▶ Alle Nachfolgerknoten dieses Knotens werden berechnet. Falls die Nachfolger bereits einen Schätzwert hat wird der niedrigere Wert dem Knoten zugeordnet (relaxieren), der Vorgänger wird dabei auch geändert.

Ende

Ende

- ▶ Wenn der Zielknoten ein “black node” ist, so hat man den idealen Weg vom Startknoten zum Endknoten gefunden.

Dijkstra - Algorithmus

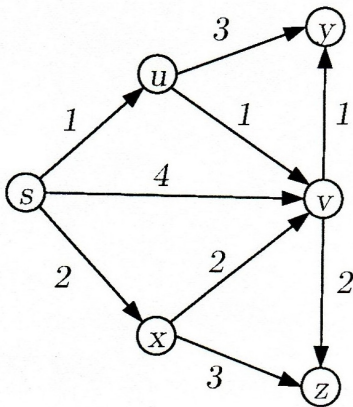


Abbildung: Pr. Dr. Schmitz Algorithmen-Design

Dijkstra - Algorithmus

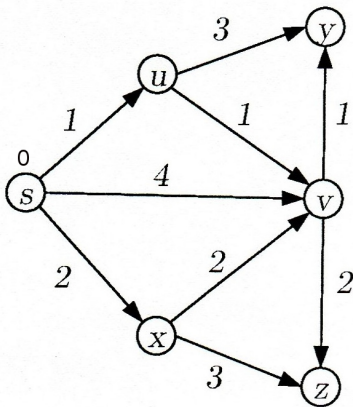


Abbildung: Pr. Dr. Schmitz Algorithmen-Design

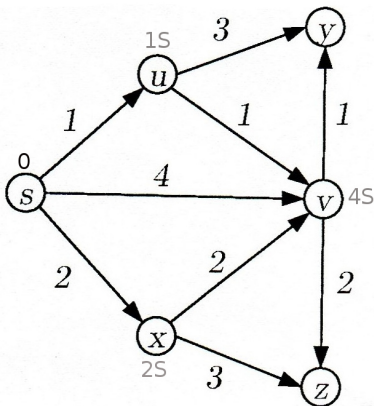


Abbildung: Pr. Dr. Schmitz Algorithmen-Design

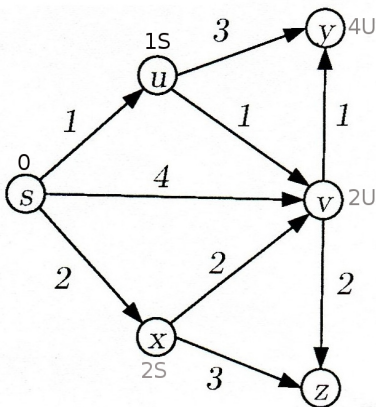


Abbildung: Pr. Dr. Schmitz Algorithmen-Design

Dijkstra - Algorithmus

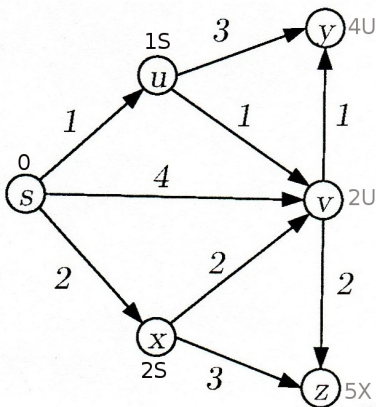


Abbildung: Pr. Dr. Schmitz Algorithmen-Design

Komplexität

Ursprüngliche Implementierung

Ursprüngliche Implementierung

- ▶ Einzelschritte:

Ursprüngliche Implementierung

- ▶ Einzelschritte:
 - ▶ Initialisieren der Arrays je $O(m)$

Ursprüngliche Implementierung

- ▶ Einzelschritte:
 - ▶ Initialisieren der Arrays je $O(m)$
 - ▶ Abarbeiten der grauen Knoten $O(m)$

Ursprüngliche Implementierung

- ▶ Einzelschritte:
 - ▶ Initialisieren der Arrays je $O(m)$
 - ▶ Abarbeiten der grauen Knoten $O(m)$
 - ▶ Bestimmen des Minimums $O(m)$

Ursprüngliche Implementierung

- ▶ Einzelschritte:
 - ▶ Initialisieren der Arrays je $O(m)$
 - ▶ Abarbeiten der grauen Knoten $O(m)$
 - ▶ Bestimmen des Minimums $O(m)$
 - ▶ Aktualisieren der Nachfolger $O(\deg(v)) \rightarrow O(k)$

Ursprüngliche Implementierung

- ▶ Einzelschritte:
 - ▶ Initialisieren der Arrays je $O(m)$
 - ▶ Abarbeiten der grauen Knoten $O(m)$
 - ▶ Bestimmen des Minimums $O(m)$
 - ▶ Aktualisieren der Nachfolger $O(\deg(v)) \rightarrow O(k)$
- ▶ insgesamt Komplexität $O(m^2)$

Implementierung mit Heap

- ▶ Vorteile:

Implementierung mit Heap

- ▶ Vorteile:
 - ▶ Heapoperationen in $O(\log m)$

Implementierung mit Heap

- ▶ Vorteile:
 - ▶ Heapoperationen in $O(\log m)$
 - ▶ Bestimmen des Minimums in $O(\log m)$

Implementierung mit Heap

- ▶ Vorteile:
 - ▶ Heapoperationen in $O(\log m)$
 - ▶ Bestimmen des Minimums in $O(\log m)$
- ▶ insgesamt Komplexität $O(k * \log m)$

Eigenschaften

Eigenschaften

- ▶ Programmiersprache: Python

Eigenschaften

- ▶ Programmiersprache: Python
- ▶ Umsetzung mit Heap (Priority Queue)

Eigenschaften

- ▶ Programmiersprache: Python
- ▶ Umsetzung mit Heap (Priority Queue)
- ▶ keine Speicherung der Farbstufen wie bei Dijkstra

Eigenschaften

- ▶ Programmiersprache: Python
- ▶ Umsetzung mit Heap (Priority Queue)
- ▶ keine Speicherung der Farbstufen wie bei Dijkstra
 - ▶ kürzerer und übersichtlicherer Code

Implementierung

Kompletter Code

```

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(log(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 # Knotennummern: s=0, u=1, x=2, y=3, v=4, z=5
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))

```

Implementierung

Eingabe

```
26 #Graph:
27 # Knotennummern: s=0, u=1, x=2, y=3, v=4, z=5
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
```

Implementierung

Algorithmus-Initialisierung

```
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
9
10
11
12
13
14
15
16
17
18
19
```


Implementierung

Algorithmus-Initialisierung

```
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(log m)
10
11
12
13
14
15
16
17
18
19
```

Implementierung

Algorithmus-Erweitern

```
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(log m)
10     while pq:                                   #O(m)
11         (_, v) = heappop(pq)                   #O(log m)
12
13
14
15
16
17
18
19
```

Implementierung

Algorithmus-Erweitern

```

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(log 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
15
16
17
18
19

```

Implementierung

Algorithmus-Aktualisieren

```

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(log 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

```

Implementierung

Rekursives Bestimmen des Pfades

```
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
```

Aufruf

```
40 d, p = dijkstra_pq(G, 0)
41 print( shortest_path(0, 5, p))
42 print( shortest_path(0, 3, p))
```



- ▶ Wegfindung in Realität und Spiel

Anwendungsbereiche

- ▶ Wegfindung in Realität und Spiel
- ▶ Routing-Algorithmus in verschiedenen Protokollen

Anwendungsbereiche

- ▶ Wegfindung in Realität und Spiel
- ▶ Routing-Algorithmus in verschiedenen Protokollen
- ▶ weitere: z.B. Münzproblem

- ▶ Prof. Dr. G. Biess: Graphentheorie. Verlag Harri Deutsch, Thun und Frankfurt/Main, 1976.
- ▶ Tanenbaum, Andrew; WETHERALL, David: Computernetzwerke. Pearson Deutschland GmbH, 2012, 2012.
- ▶ Prof. Dr. Ottmann und Prof. Dr. Widmayer: Algorithmen und Datenstrukturen, Reihe Informatik Bd. 70. BI-Wissenschaftsverlag Mannheim/Wien/Zürich, 1990.
- ▶ Dijkstra, Edsger: A Method of Programming. Addison-Wesley Publishing Company, 1988.
- ▶ Vorlesungsskript Algorithmen-Design Prof. Dr. Heinz Schmitz Stand 6. April 2015 Lektion 6 Beispiel 6.1 (Bild)
- ▶ Prof. Dr. Heinz Schmitz, Hochschule Trier, Vorlesung Algorithmen-Design, Code aus Lektion 6, S.137, dijkstra.py