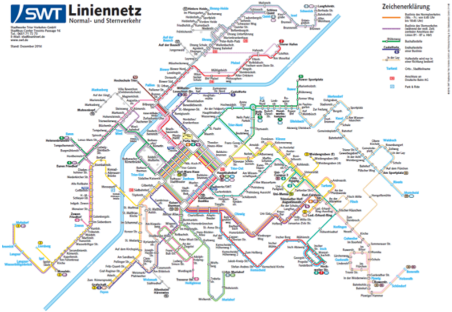
test

Max Mustermann

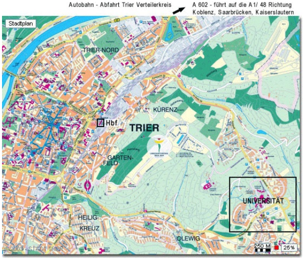
Fachhochschule Trier

22. Juli 2015

Problemstellung



(Quelle: `http://www.vrt-info.de/images/liniennetzplan_trier_2014.png`)



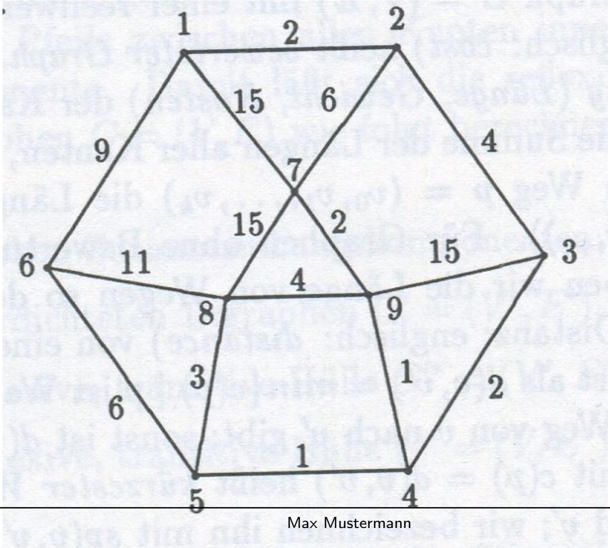
(Quelle: `http://www.uni-trier.de/index.php?id=27631`)

Table of content

Definition

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)

Beispiel



Dijkstra

Erweiterung der Knoten um ...

Dijkstra

Erweiterung der Knoten um ...

- einen Schätzwert.

Dijkstra

Erweiterung der Knoten um ...

- einen Schätzwert.
- einen Vorgänger.

Dijkstra

white node

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

Dijkstra

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. Dieser Weg muss aber noch nicht optimal sein.

Dijkstra

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. Dieser Weg muss aber noch nicht optimal sein.

black node

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

Berechnung von Knotenwerten

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

Berechnung: $k_A = k_V + g$

Dijkstra

Initialisierung

Initialisierung

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

Dijkstra

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

Schritt

- Der "grey node" mit dem niedrigsten Schätzwert wird gesucht.

Schritt

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

Dijkstra

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 wird der niedrigere Wert dem Knoten zugeordnet (relaxieren).

Ende

Dijkstra

Ende

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

Komplexität

Komplexität

Ursprüngliche Implementierung

Komplexität

Ursprüngliche Implementierung

- Einzelschritte:

Komplexität

Ursprüngliche Implementierung

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

Komplexität

Ursprüngliche Implementierung

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

Komplexität

Ursprüngliche Implementierung

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

Komplexität

Ursprüngliche Implementierung

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

Komplexität

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 $O(m^2)$

Komplexität

Implementierung mit Heap

- Vorteile:

Komplexität

Implementierung mit Heap

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

Komplexität

Implementierung mit Heap

- Vorteile:
  - Heapoperationen in $O(logm)$
  - Bestimmen des Minimums in $O(logm)$

Komplexität

Implementierung mit Heap

- Vorteile:
  - Heapoperationen in $O(logm)$
  - Bestimmen des Minimums in $O(logm)$
- insgesamt Komplexität $O(k * logm)$

Eigenschaften

Implementierung

Eigenschaften

- Programmiersprache: Python

Implementierung

Eigenschaften

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

Implementierung

Eigenschaften

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

Implementierung

Eigenschaften

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

Kompletter Code

```
from heapq import heappush, heappop

def dijkstra_pq(G,s):
    m = len(G)                                  #O(1)
    pq = []         #priority queue             #O(1)
    d = [None]*m    #kosten                     #O(m)
    p = [None]*m    #vorgaenger               #O(m)
    d[s] = 0                                    #O(1)
    heappush(pq, (0,s))                         #O(log(m))
    while pq:                                   #O(m)
        (_,v) = heappop(pq)                     #O(log m)
        for u in G[v]:                          #O(deg(v)) —> O(k)
            alt = d[v] + G[v][u]                #O(1)
            if d[u]== None or alt < d[u]:       #O(1)
                d[u] = alt                    #O(1)
                  p[u] = v                        #O(1)
            heappush(pq, (alt,u))               #O(log m)
    return d,p

def shortest_path(s,v,p):
    if v == None:
        return []
    else:
        return shortest_path(s,p[v],p) + [v]

#Graph:

def define_G():
    G = [   {1:1, 4:4, 2:2},    # Nachfolger von s
            {3:3, 4:1},         # von u
            {4:2, 5:3},         # von x
            {},                 # von y
            {3:1, 5:2},         # von v
            {}                  # von z
        ]
    return G


G = define_G()
d,p = dijkstra_pq(G,0)
print( shortest_path(0,5,p))
print( shortest_path(0,3,p))
```

Eingabe

```
#Graph:

def define_G():
    G = [    {1:1, 4:4, 2:2},     # Nachfolger von s
             {3:3, 4:1},          # von u
             {4:2, 5:3},          # von x
             {},                  # von y
             {3:1, 5:2},          # von v
             {}                   # von z
        ]
    return G
```

Algorithmus-Initialisierung

```
from heapq import heappush, heappop

def dijkstra_pq(G, s):
    m = len(G)                                  #O(1)
    pq = []         #priority queue             #O(1)
    d = [None]*m    #kosten                     #O(m)
    p = [None]*m    #vorgaenger              #O(m)
```

Algorithmus-Initialisierung

```python
from heapq import heappush, heappop

def dijkstra_pq(G, s):
    m = len(G)                                    #O(1)
    pq = []          #priority queue              #O(1)
    d = [None]*m     #kosten                      #O(m)
    p = [None]*m     #vorgaenger              #O(m)
    d[s] = 0                                      #O(1)
    heappush(pq, (0, s))                          #O(log m)
```

Algorithmus-Erweitern

```
from heapq import heappush, heappop

def dijkstra_pq(G, s):
    m = len(G)                                      #O(1)
    pq = []            #priority queue              #O(1)
    d = [None]*m       #kosten                      #O(m)
    p = [None]*m       #vorgaenger              #O(m)
    d[s] = 0                                        #O(1)
    heappush(pq, (0,s))                             #O(log m)
    while pq:                                       #O(m)
        (_,v) = heappop(pq)                         #O(log m)
```

HOCHSCHULE TRIER
Trier University of Applied Sciences
Informatik - Computer Science

Algorithmus-Erweitern

```
from heapq import heappush , heappop

def dijkstra_pq(G, s ):
    m = len(G)                                        #O(1)
    pq = []            #priority queue                #O(1)
    d = [None]*m       #kosten                        #O(m)
    p = [None]*m       #vorgaenger                #O(m)
    d[s] = 0                                          #O(1)
    heappush(pq, (0,s))                               #O(log m)
    while pq:                                         #O(m)
        (_,v) = heappop(pq)                           #O(log m)
        for u in G[v]:                                #O(deg(v)) --> O(k)
            alt = d[v] + G[v][u]                      #O(1)
```

Algorithmus-Aktualisieren

```
from heapq import heappush, heappop

def dijkstra_pq(G,s):
    m = len(G)                                   #O(1)
    pq = []          #priority queue             #O(1)
    d = [None]*m     #kosten                     #O(m)
    p = [None]*m     #vorgaenger              #O(m)
    d[s] = 0                                     #O(1)
    heappush(pq, (0,s))                          #O(log m)
    while pq:                                    #O(m)
        (_,v) = heappop(pq)                      #O(log m)
        for u in G[v]:                           #O(deg(v)) --> O(k)
            alt = d[v] + G[v][u]                 #O(1)
            if d[u]== None or alt < d[u]:        #O(1)
                d[u] = alt                     #O(1)
                  p[u] = v                         #O(1)
                heappush(pq, (alt,u))            #O(log m)
    return d,p

```

Implementierung

Rekursives Bestimmen des Pfades

```python

def shortest_path(s,v,p):
    if v == None:
        return []
    else:
        return shortest_path(s,p[v],p) + [v]

```

Aufruf

```python
d,p = dijkstra_pq(G,0)
print( shortest_path(0,5,p))
print( shortest_path(0,3,p))
```