Shortest Path Algorithmus von Edsger Dijkstra
Für die PK-Note!

Michael Dienert

17. September 2024

Inhaltsverzeichnis

1 Shortest Path Algorithmus **1**
1.1 Graphen 1
1.2 Knoten 1
1.3 Pseudocode 2

2 Programmierung **3**
2.1 Link-States-Matrix 3
2.1.1 Testdaten 4
2.2 Aufteilung in Module 4
2.2.1 adjacencies 4
2.2.2 dijkstra-Algorithmus 4
2.2.3 Start- und Testmodul 5

1 Shortest Path Algorithmus

1.1 Graphen

Der Shortest Path Algorithmus von Edsger Dijkstra sucht in einem *gewichteten Graphen* (= Netz, bestehend aus Knoten und Verbindungen mit Kosten) nach den günstigsten Pfaden von einem Startknoten zu allen anderen Knoten.

Das Beispielnetz in diesem Dokument sieht so aus:

Es enthält die vier Knoten **a**,**b**,**c**,**d**. Die Ziffern stellen die **Kosten** der Verbindung dar. In der Netzwerktechnik werden die Kosten aus dem Kehrwert der Bandbreite (Bitrate) brechnet, verwendet man den Dijkstra-Algorithmus um Wegstrecken zu opimieren, entsprechen die Kosten der Entfernung zwischen den Knoten.

1.2 Knoten

Jeder Knoten des Graphen hat beim Anwenden des Algorithmus folgende Eigenschaften:

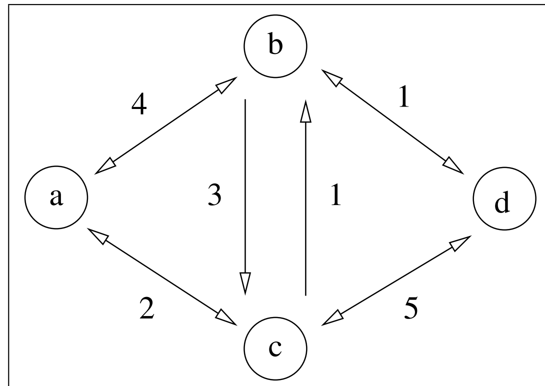


Abbildung 1: Ein Beispielnetz mit 4 Knoten

Distanz: Die beste angenommene Distanz eines Knoten zum Startpunkt

Vorgänger: Der Name des Vorgängerknotens auf dem bislang besten Pfad zum Startort

Offen/Erledigt: Diese Eigenschaft (im Folgenden Variable `qs`) eines Knotens kann 3 Werte annehmen:

- **-** der Knoten wurde noch nicht besucht
- **q** der Knoten wurde besucht, seine kürzeste Distanz zum Startort ist aber noch nicht bekannt
- **s** der Knoten ist erledigt (*settled*), d.h. seine kürzeste Distanz zum Start ist bekannt.

1.3 Pseudocode

Folgender Pseudocode war vor einiger Zeit bei Wikipedia angegeben. Inzwischen 2024) ist dort ein leicht geänderter Pseudocode veröffentlicht.

```
//initialisieren
setze die Distanzen aller Knoten auf unendlich: z.B. $dist = 1000000$
setze den Vorgaengerknoten aller Knoten auf "" (leerer String)
setze die offen/erledigt-Eigenschaft (qs) aller Knoten auf "-"

setze qs des Startknotens auf "q"
setze die Distanz dist des Startknotens auf dist=0

while( es gibt noch knoten, bei denen qs=="q" ist ){
   knoten u = extractMinimum();
   relaxNeighbors(u);
}
```

Methode extractMinimum():

```
extractMinimum(){
   finde den Knoten u mit minimaler Distanz dist aus der Menge
      aller Knoten, bei denen qs=="q" ist ;
   setzt qs von u auf "s" (aka markiere u als settled);
   return u;
}
```

Methode relaxNeighbors(u):

```
relaxNeighbors(knoten u){
   foreach Knoten v, qs von v ist nicht "s"  &&
                  v ist nachbar von u{
      if (distanz von v > distanz von u + abstand zwischen u und v ){
         distanz von v = distanz von u + abstand zwischen u und v ;
         vorgänger von v = u;
         qs von v = "q";
      }
   }
}
```

2 Programmierung

2.1 Link-States-Matrix

Als erstes ist eine Datenstruktur entwerfen, die alle Link-States sammelt.

Diese Link-States-Matrix besteht aus einer Hash-Tabelle (`dict`), deren Schlüssel die Knotennamen bilden (String) und deren zugeordnete Werte wieder eine Hash-Tabelle sind (Zeilen Tab. 1).

Die Schlüssel-Werte-Paare der Hash-Tabellen, die die Zeilen der folgenden Tabelle (Tab. 1) bilden, enthalten einen Zielknoten (Schlüssel) mit seiner Entfernung (Wert).

Die beiden Abbildungen in Tabelle 1 sind gleichwertig: Die inneren Hash-Tabellens wurden zunächst als Tabellen (links) und dann vereinfacht als Menge von Wertepaaren dargestellt (rechts).

String	Hash-Tabelle	
a	b	4
	c	2
b	a	4
	c	3
	d	1
c	a	2
	b	1
	d	5
d	b	1
	c	5

String	Hash-Tabelle			
a		(b,4)	(c,2)	
b	(a,4)		(c,3)	(d,1)
c	(a,2)	(b,1)		(d,5)
d		(b,1)	(c,5)	

Tabelle 1: Link-States-Matrix mit Hash-Tabellen

In Python soll die Matrix des Beispielnetzes so aussehen:

```
{
  'a': {'b': 4, 'c': 2},
  'b': {'a': 4, 'c': 3, 'd': 1},
  'c': {'a': 2, 'b': 1, 'd': 5},
  'd': {'b': 1, 'c': 5}
}
```

2.1.1 Testdaten

Mit folgenden Funktionsaufrufen kann mann das Testnetzwerk einspielen:

```
ls.add_link_states(knoten="a", nachbar="b", distanz=4)
ls.add_link_states(knoten="a", nachbar="c", distanz=2)

ls.add_link_states(knoten="b", nachbar="a", distanz=4)
ls.add_link_states(knoten="b", nachbar="c", distanz=3)
ls.add_link_states(knoten="b", nachbar="d", distanz=1)

ls.add_link_states(knoten="c", nachbar="a", distanz=2)
ls.add_link_states(knoten="c", nachbar="b", distanz=1)
ls.add_link_states(knoten="c", nachbar="d", distanz=5)

ls.add_link_states(knoten="d", nachbar="b", distanz=1)
ls.add_link_states(knoten="d", nachbar="c", distanz=5)
```

2.2 Aufteilung in Module

2.2.1 adjacencies

Die Link-States-Matrix (aka *adjacencies*) soll in einem eigenen Modul angelegt werden. Dieses Modul soll folgende Funktionen enthalten:

```
add_link_states(knoten, nachbar, distanz)]
get_nachbarn(knoten)
get_alle_knoten()
get_distanz(u,v)
```

2.2.2 dijkstra-Algorithmus

Der eigentliche Algorithmus soll im Modul `dijkstra` angelegt werden. Die Zustände der Knoten sollen in einer weiteren Hashtabelle gespeichert werden. Key ist hier ebenfalls der Knotenname, Value ist eine Hashtabelle mit den Keys:

- `"dist"`: Distanz zum Startknoten
- `"pre"`: Vorgängerknoten
- `"qs"` nicht besucht, besucht, settled

Nach dem Aufruf von `init_zustaende("a")`, soll die Hashtabelle mit den Knotenzuständen so aussehen:

```
{
  'a': {'dist': 0, 'pre': '', 'qs': 'q'},
  'b': {'dist': 1000000, 'pre': '', 'qs': '-'},
  'c': {'dist': 1000000, 'pre': '', 'qs': '-'},
  'd': {'dist': 1000000, 'pre': '', 'qs': '-'}
}
```

Funktionen:

```
init_zustaende(knoten)  #knoten ist der startknoten
nicht_alle_knoten_settled() #True/False
extract_minimum()
relax_neighbors(knoten)
run_dijkstra()
```

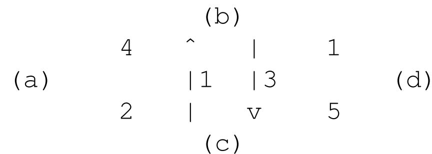
2.2.3 Start- und Testmodul

Um den Algorithmus zu starten, soll ein Modul geschrieben werden, das die Ajacencies-Matrix für das Beispiel aus Abb. 1.1 aufbaut, indem es wiederholt die Funktion `add_link_states(knoten, nac` aufruft und dann die besten Pfade von einem Startknoten aus ermittelt.

Um das Ergebnis zu sehen, soll ein Druckfunktion geschrieben werden, die die Zustandsmatrix nach dem Initialisieren und am Ende des Programmlaufs ausgibt:

```
13:07:43|micha@msv:~/development/python/dijkstra$ python3 start.py

 datenstruktur, die alle LSAs sammelt und in einem dict von
  dict speichert;
  beispiel:  (knoten) und distanzen
```




```
  dict von dict

       ----------> U
         | a   b   c   d
       -----------------
  |    a | 0   4   2
  |    b | 4   0   3   1
  v    c | 2   1   0   5
  V    d |     1   5   0

so sieht die nachbarschaftsmatrix aus:
a| {'b': 4, 'c': 2}
b| {'a': 4, 'c': 3, 'd': 1}
c| {'a': 2, 'b': 1, 'd': 5}
d| {'b': 1, 'c': 5}

a| {'dist': 0, 'pre': '', 'qs': 'q'}
b| {'dist': 1000000, 'pre': '', 'qs': '-'}
c| {'dist': 1000000, 'pre': '', 'qs': '-'}
d| {'dist': 1000000, 'pre': '', 'qs': '-'}

a| {'dist': 0, 'pre': '', 'qs': 's'}
b| {'dist': 3, 'pre': 'c', 'qs': 's'}
c| {'dist': 2, 'pre': 'a', 'qs': 's'}
d| {'dist': 4, 'pre': 'b', 'qs': 's'}
```