

Alle kürzesten Wege in einem
Graphen $G = (V,E)$ -
Zusammenfassung

Inhaltsangabe

1 Alle kürzesten Wege in einem Graphen G

2 Verfahren für kürzesten Wege in G

2.1 Dijkstra – Algorithmus

- Funktionsweise und Laufzeit des Algorithmus
- Verhalten bei negativer Kantenlänge

2.2 Ford - Algorithmus

- Unterschied zum Dijkstra-Algorithmus
- Funktionsweise und Laufzeit

3 Direkte Verfahren zur Bestimmung aller kürzesten Wege in G

3.1 Ein auf Matrixmultiplikation basiertes Verfahren

- Idee des Verfahrens
- Zusammenhang mit Matrixmultiplikation
- Pseudocode und Laufzeit
- Verbesserung der Laufzeit

3.2 Floyd - Algorithmus

- Idee des Verfahrens
- Pseudocode
- Laufzeit
- Erkennen negativer Zyklen

4 Warshall - Algorithmus

- Idee des Algorithmus
- Pseudocode und Laufzeit

5 Literaturverzeichnis

1 Alle kürzesten Wege in einem Graphen G

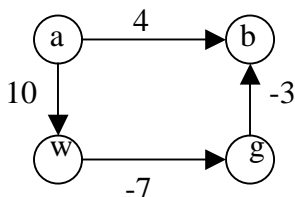
Dieses Problem ist verwandt mit der Suche nach kürzesten Wegen in einem Graphen G. Der Unterschied besteht darin, dass nicht nur Wege von einem bestimmten Knoten aus zu den restlichen ermittelt werden, sondern die kürzesten Entfernungen zwischen jeweils zwei Knoten in einem Graphen G bestimmt werden. Der Graph muss natürlich gewichtet sein, wobei es keine Rolle spielt, ob er gerichtet oder ungerichtet ist. In ungerichteten Graphen werden die Kanten als Doppelpfeile in beide Richtungen angesehen. In der Praxis finden die Algorithmen zur Bestimmung aller kürzesten Wege ihre Anwendung z. B. für Straßennetze, Flugrouten oder Bahnnetze, um die kürzeste Entfernung zwischen zwei bestimmten Orten anzugeben.

2 Algorithmen für kürzeste Wege in G

2.1 Dijkstra - Algorithmus

Man kann sich leicht vorstellen, dass die Verfahren zur Bestimmung der kürzesten Entfernungen von einem Knoten aus zu allen anderen für jeden Knoten in G angewandt werden können, so dass schließlich alle kürzesten Wege in diesem Graphen bekannt sind. Der Dijkstra-Algorithmus zählt zu diesen Verfahren. Der Verlauf des Algorithmus sieht folgendermaßen aus: alle Knoten werden in 3 Klassen eingeteilt, und zwar in gewählte Knoten, Randknoten und unerreichte Knoten. Gewählt wird ein Knoten dann, wenn die Länge des kürzesten Wegs vom Ausgangsknoten s zu ihm bekannt ist. Zu den Randknoten existiert eine Verbindung von s aus und zu den unerreichten Knoten ist noch keine Verbindung hergestellt worden. Aus der Menge der Randknoten wird nun derjenige v gewählt, der am nächsten zum aktuell gewählten Knoten liegt. Nachdem ein Knoten gewählt ist, werden alle zu ihm adjazenten Knoten in den Randbereich eingetragen. Falls dieser in einem Heap gespeichert ist beträgt die Laufzeit des Algorithmus $O(n \log n)$ mit $n := |V|$ und $m := |E|$ (für Bestimmung aller kürzesten Wege).

Nachdem zwei Knoten s und v gewählt sind, wird allerdings die Entfernung $c(s,v)$ als die kürzeste zwischen s und v gespeichert und kann nicht mehr verändert werden. Aus dieser Tatsache folgt, dass der Dijkstra-Algorithmus nur für Distanzgraphen ($c: E \rightarrow \mathbf{R}_0^+$; c : Kostenfunktion des Graphen G) bestimmt ist, da er darauf basiert, dass durch Hinzunahme weiterer Kanten die Entfernung zum Ausgangspunkt s nur länger werden kann. Für Graphen mit negativen Kantenlängen liefert der Algorithmus in den meisten Fällen ein falsches Ergebnis:



Der Dijkstra-Algorithmus würde an dieser Stelle für $c((a,b))$ folgendes berechnen: a wird gewählt, dann wird der Knoten b aus dem Randbereich genommen, weil er am nächsten zu a liegt. Danach ist auch b gewählt und die kürzeste Entfernung von a nach b mit $c(d(a,b)) = 4$

wird als die Endgültige gespeichert (d : der kürzeste Weg). Der richtige kürzeste Pfad (a,b) geht jedoch über w und g, so dass die entsprechende Entfernung 0 Längeneinheiten beträgt.

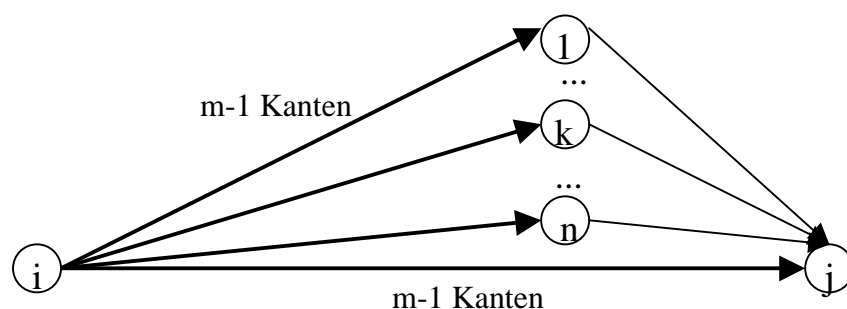
2.2 Ford – Algorithmus

Dieser Algorithmus basiert genauso wie der Dijkstra-Algorithmus auf Adjazenzlisten, erlaubt aber, die kürzesten Wege in beliebigen, bewerteten Graphen zu bestimmen. Mittels n -facher Wiederholungen können natürlich alle kürzesten Wege in diesen Graphen bestimmt werden. Für den Ford-Algorithmus sind nur Randknoten von Bedeutung, zwischen gewählten und unerreichten Knoten wird nicht unterschieden. Die als erste berechnete Entfernung zwischen zwei Knoten s und v wird jetzt nicht als die Entgültige angesehen – sie wird mit Längen aller existierenden Pfade (s,v) verglichen, und die minimale unter diesen schließlich in $c(d(s,v))$ gespeichert. Für den oben erwähnten Beispiel würde der Ford-Algorithmus zuerst den Weg (a,b) mit 4 Längeneinheiten finden, dann aber (weil a und b nicht mehr gewählt werden) den Pfad a->w->g->b mit 0 LE, so dass $c(d(a,b)) = 0$ als das entgültige Ergebnis geliefert wird. Die Laufzeit des Ford-Algorithmus mit n -facher Wiederholung für alle kürzesten Wege beträgt $O(n^3)$. Genauso wie der Dijkstra-Algorithmus hält er an, nachdem der Randbereich leer ist.

3 Direkte Verfahren zur Bestimmung aller kürzesten Wege in G

3.1 Ein auf Matrixmultiplikation basiertes Verfahren

Natürlich gibt es auch direkte Verfahren, um alle kürzesten Wege in beliebigen, bewerteten Graphen zu bestimmen. Eines davon hängt mit der Matrixmultiplikation zusammen und arbeitet deshalb mit Adjazenzmatrizen. Die Idee des Verfahrens ist in der folgenden Skizze dargestellt:



$$d_{ij}^{(m)} = \begin{cases} 0, & \text{falls } i = j \\ \min(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} (d_{ik}^{(m-1)} + c_{kj})) , & \text{falls } i \neq j \end{cases}$$

wobei $d_{ij}^{(m)}$ den Weg von i nach j mit höchstens m Kanten bedeutet und $m \in \{0, 1, 2, \dots, n-1\}$ ist. Dieses Verfahren vergleicht für alle Knotenpaare die Pfadlänge $c((i,j))$ mit höchstens $m-1$ Kanten mit den „Umwegen“ über jeden Knoten in G . Der „Umweg“ besteht jeweils aus

höchstens $m-1$ Kanten von i zum Zwischenknoten k und nur einer Kante von k nach j . Die minimale Entfernung wird dann in $d_{ij}^{(m)}$ gespeichert. Da es auf einem kürzesten Weg von i nach j maximal $n-1$ Kanten geben kann (vorausgesetzt der Graph besitzt keine negative Zyklen), hält der Algorithmus nach $n-1$ Schritten an.

Um auf die allgemein bekannte Matrixmultiplikation zu schließen, muss der oben definierte Term umgeformt werden:

$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n} (d_{ik}^{(m-1)} + c_{kj}))$$

$$\text{wegen } d_{ij}^{(m-1)} = d_{ij}^{(m-1)} + c_{jj} (=0)$$

$$\Rightarrow d_{ij}^{(m)} = \min_{1 \leq k \leq n} (d_{ik}^{(m-1)} + c_{kj})$$

Seien $A = (a_{ij})_{i,j \in \{1, \dots, n\}}$ und $B = (b_{ij})_{i,j \in \{1, \dots, n\}}$ $n \times n$ -Matrizen. Ihr Produkt hat die Form:

$$A * B = (\sum_{k=1}^n a_{ik} * b_{kj})_{i,j \in \{1, \dots, n\}}$$

Unter Berücksichtigung des vorher gewonnenen rekursiven Terms

$d_{ij}^{(m)} = \min_{1 \leq k \leq n} (d_{ij}^{(m-1)} + c_{kj})$ definiert man eine neue Operation auf den Matrizen A und B so, dass „+“ durch „min“ und „*“ durch „+“ ersetzt wird. Demzufolge sieht die „Matrixmultiplikation“ jetzt so aus:

$$A \bullet B = (\min_{1 \leq k \leq n} (a_{ik} + b_{kj}))_{i,j \in \{1, \dots, n\}}$$

Mit dieser Definition kann man die Lösung des Problems aller kürzesten Wege in einem Graphen auf eine veränderte Form der Matrixmultiplikation zurückführen.

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} (d_{ik}^{(m-1)} + c_{kj}) \quad \Rightarrow \quad D^{(m)} = D^{(m-1)} \bullet C \quad \text{für } m \in \{0, 1, 2, \dots, n-1\}$$

C ist die vorher bestimmte Matrix der Kostenfunktion $c: E \rightarrow \mathbf{R}$

Folgender Algorithmus beschreibt die Vorgehensweise des Verfahrens:

Algorithmus (Pseudocode): *alle kürzesten Wege in $G = (V, E)$ mit C als Matrix der Kostenfunktion $c: E \rightarrow \mathbf{R}$*

```

n: = |V|;
D(1): = C;
for m: = 2 to n-1 do
    D(m): = einzelne kürzeste Wege ( D(m-1), C );
return D(n-1);

```

Algorithmus (Pseudocode): *einzelne kürzeste Wege (D, C)*

```

n: = |V|;
let D' = (d'_{ij}) be an n x n-matrix
for i: = 1 to n do
    for j: = 1 to n do
        d'_{ij}: <- ∞;
        for k: = 1 to n do
            d'_{ij}: = min( d'_{ij}, d_{ik} + c_{kj} );
return D';

```

Die 4 for-Schleifen in beiden Teilalgorithmen liefern eine Gesamtlaufzeit von $O(n^4)$, was für breite Graphen von großem Nachteil ist.

Es ist jedoch möglich, diese Laufzeit zu verbessern. Wenn man bedenkt, dass nur die letzte Matrix $D^{(n-1)}$ von Bedeutung ist, so lässt sich der Zeitaufwand auf $O(n^3 \log n)$ verkürzen. Die Berechnung erfolgt dann nur für Matrizen mit 2er Potenzen als die höchste Kantenzahl:

$$\begin{aligned} D^{(1)} &= C \\ D^{(2)} &= C^2 = C * C \\ D^{(4)} &= C^4 = C^2 * C^2 \end{aligned}$$

$$\dots \\ D^{(2^{\lceil \lg(n-1) \rceil})} = C^{(2^{\lceil \lg(n-1) \rceil})}$$

Für alle $m \geq n-1$ gilt $D^{(m)} = D^{(n-1)}$, da die kürzesten Entfernungen bereits in $D^{(n-1)}$ gespeichert sind und weitere Hinzunahme von Kanten den Weg nur verlängern kann. Folglich gilt auch für das Endprodukt $D^{(2^{\lceil \lg(n-1) \rceil})}$: $D^{(2^{\lceil \lg(n-1) \rceil})} = D^{(n-1)}$. Der Algorithmus stellt den genaueren Ablauf dar:

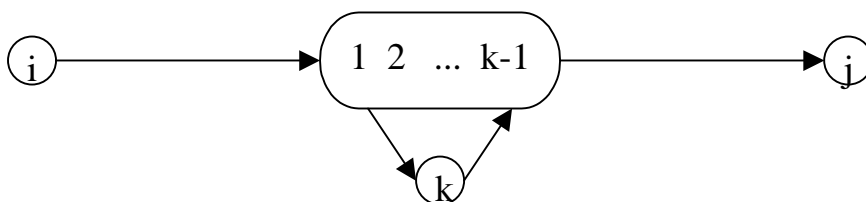
Algorithmus (Pseudocode): alle kürzesten Wege in $G = (V;E)$ mit C als Matrix der Kostenfunktion $c: E \rightarrow \mathbf{R}$ in $O(n^3 \log n)$

```
n := |V|;
D(1) := C;
m := 1;
while n-1 > m do
    D(2m) := einzelne kürzeste Wege (D(m), D(m));
    m := 2m;
return D(m);
```

3.2 Floyd - Algorithmus

Dieses Verfahren basiert genauso wie die Matrixmultiplikation auf Adjazenzmatrizen, ist jedoch etwas schneller.

Das Prinzip des Algorithmus besteht darin, dass man alle Knoten nacheinander zu einem sogenannten Pivot-Element wählt:



Mit

$$d_{ij}^{(k)} = \begin{cases} 0, & \text{falls } i = j \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{falls } i \neq j \end{cases}$$

für $k \in \{0, 1, 2, \dots, n\}$. Hier wird der direkte Weg von i nach j mit höchstens $k-1$ Knoten mit dem „Umweg“ über das Pivot-Element k verglichen, wobei die Pfade (i,k) und (k,j) jeweils

höchstens $k-1$ Knoten enthalten dürfen. Die minimale Entfernung $c(d(i,j))$ wird in $d_{ij}^{(k)}$ gespeichert. Nachdem jeder Knoten zum Pivot-Element gewählt wurde, hält der Algorithmus an. Der folgende Pseudocode stellt den genauen Ablauf dar:

Algorithmus : *Floyd-Algorithmus* (C)

```

n := |V|;
D(1) := C;
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
       $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} )$ ;
return D(n);

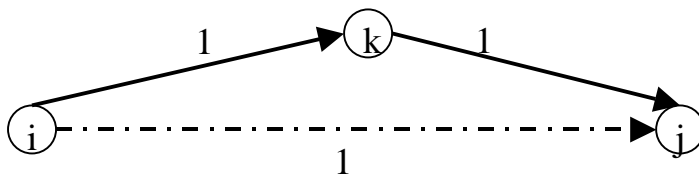
```

Da jeder Knoten als Anfangs- und Endpunkt eines Pfades und zusätzlich als Pivot-Element im Algorithmus verwendet wird (vgl. 3 for-Schleifen), beträgt die Laufzeit für den Floyd-Algorithmus $O(n^3)$.

Genauso wie das Verfahren auf der Basis von Matrixmultiplikation, kann der Floyd-Algorithmus feststellen, ob ein Graph negative Zyklen enthält, und zwar, ist es dann der Fall, wenn in der Endmatrix $D^{(n)}$ auf der Diagonale mindestens ein negativer Eintrag steht. D.h. von dem Knoten zu sich selbst gibt es einen Pfad mit Länge, die kleiner als 0 ist. Diese Tatsache lässt darauf schließen, dass in G Zyklen negativer Länge vorhanden sein müssen.

4 Warshall – Algorithmus

Es existiert eine etwas veränderte Form des Floyd-Algorithmus, die der Bestimmung der transitiven Hülle eines Graphen G dient – der Warshall-Algorithmus. Die transitive Hülle G' von G ist folgender Maßen definiert:



Falls in G keine Kante von i nach j existiert, jedoch aber ein Pfad über k (entspricht dem Pivot-Element von vorher), wird die Kante (i,j) hinzugefügt. Nachdem alle Kanten in dieser Weise geprüft sind, erhält man die transitive Hülle G' .

Algorithmus : *Warshall-Algorithmus* (W)

```

n := |V|;
T(1) := W;
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
       $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee ( t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} )$ ;
return T(n);

```

Die Matrix W enthält als Einträge 0, falls die Kante nicht existiert, und 1, falls sie vorhanden ist.

Da der Algorithmus die gleiche Struktur wie der Floyd Algorithmus besitzt, beträgt auch hier die Laufzeit $O(n^3)$.

5 Literaturverzeichnis

- [1] : “Introduction to Algorithms“ ; T. H. Cormen, C. E. Leiserson, R. L Rivest;
The MIT Press, Cambridge; 1990
- [2] : „Algorithmen und Datenstrukturen“ ; T. Ottman; P. Widmayer; Spektrum Verlag
Heidelberg; 1990
- [3] : „Informatik. Datenstrukturen und Konzepte der Abstraktion“ ; International Thomson
Publishing, Bonn; 1996
- [4] : „Algorithmen“ ; R. Sedgewick ; Addison-Wesley, Bonn ; 1991
- [5] : „Graphen und Algorithmen“ ; Teubner, Stuttgart ; 1994