

**Proseminar**

Algorithmen und Datenstrukturen

Thema:

**Kürzeste Wege (Single Source Shortest Paths)**

bei

Prof. Dr. Franz J. Brandenburg

Lehrstuhl für theoretische Informatik

Referent: Sebastian Heglmeier

SS 01

I. Gliederung	2
II. Vorstellung der einzelnen Algorithmen	3
1. Grundlegendes über kürzeste Wege	3
2. Dijkstra – Algorithmus	3
2.1 Funktionsschema	3
2.2 Edge Relaxation	4
2.3 PseudoCode	5
2.4 Komplexität	6
2.5 Beweis auf Korrektheit	6
3. Bellman - Ford – Algorithmus	7
3.1 Problem und Motivation	7
3.2 Lösungsidee	7
3.3 PseudoCode	7
3.4 Komplexität	8
3.5 Beweis auf Korrektheit	8
4. größte gemeinsame Teilfolge (LCS)	9
4.1 Vorteile und Motivation	9
4.2 Lösungsidee und PseudoCode	9
4.3 Komplexität	10
III. Literaturnachweis	11

## II Vorstellung der einzelnen Algorithmen

### 1. Grundlegendes über kürzeste Wege

Single Source Shortest Path Algorithmen berechnen die jeweils kürzesten Wege in einem Graphen  $G$  mit einer Menge von Knoten  $V$  und einer Menge von Kanten  $E$ , ausgehend vom Startknoten  $s$  zu allen anderen Knoten.

Dies ist für Straßenkarten und Schienennetze wichtig, aber auch für sehr praxisbezogene Anwendungen, beispielsweise die Navigation von Robotern (Graphen-Wegplanung für bekannte Umgebungen).

Der Dijkstra – Algorithmus, kombiniert mit einem Heap, berechnet kürzeste Wege in Distanzgraphen, liefert jedoch falsche Ergebnisse bei negativen Kantenlängen. Der Bellman-Ford-Algorithmus dagegen besitzt eine Prüfroutine um auch solche negativen Kantenlängen (z.B. bei Kosten) korrekt behandeln zu können.

### 2. Dijkstra – Algorithmus

- $G = (V, E)$ , ( $E = V \times V$ ),  $G$  wird durch Adjazenzlisten dargestellt.
- $E = \{(u, v) \mid u \neq v, u, v \in V\}$
- $w: E \rightarrow \mathbb{R}^+ \cup \{+\infty\}$
- $d(u, v) = \begin{cases} \min \{w(p) \mid p \text{ ist Weg von } u \text{ nach } v\} & \text{(Gewicht des kürzesten Pfades)} \\ \infty & \text{wenn kein Weg existiert} \end{cases}$
- $w(P) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$   
(das Gewicht eines Pfades ist die Summe der Gewichte der einzelnen Kanten von  $P$ )

#### 2.1 Funktionsschema

Die Idee des Dijkstra-Algorithmus basiert auf 3 Mengen:

Die Menge der gewählten Knoten ( $C$ ) beinhaltet alle bereits abgearbeiteten Knoten (Innerhalb der Wolke).

Die Menge der Randknoten  $R$  enthält alle Knoten, die direkt adjazent zu einem Knoten in  $C$  sind. (In der PriorityQueue, die als Heap implementiert ist)

Die restlichen Knoten sind in der Menge der unerreichten Knoten, sie haben Gewicht unendlich und sind aus  $(V-(C+R))$ .

#### Start:

Sei  $D[v]$  der aktuell kürzeste Abstand zum Startknoten  $s$ . Alle Knoten sind unerreicht also ist ihr Label  $D[v] = \infty$ .

#### Initialisierung:

$C = \{s\}$ ,  $D[s]=0$ , alle direkt mit  $s$  verbundenen Knoten  $v$  werden in  $R$  aufgenommen und ihr Label wird neu berechnet:  $D[v]=d(s, v)$ , ihr Vaterverweis zeigt auf  $s$ .

#### Solange $R \neq \emptyset$ :

(Es gibt noch Knoten deren Abstand nicht endgültig berechnet wurde): der Knoten  $v$  aus  $R$  (Heap) mit der kürzesten Distanz zu einem Knoten  $u \in C$  wird in die Wolke  $C$  aufgenommen. Der Vaterzeiger von  $v$  wird auf  $u$  gesetzt.

Füge alle zu  $v$  adjazenten Knoten aus der Menge der unerreichten Knoten zu  $R$  hinzu, führe dabei bei allen Knoten die Edge-Relaxation durch.

Alle Knoten haben nun einen Wert, den kürzesten Weg vom Startknoten  $s$  aus, aufgrund der Vaterverweise kann man nachvollziehen wie dieser zustande gekommen ist. Bedingt durch die Initialisierung haben unerreichbare Knoten noch immer den Wert  $\infty$ .

## **2.2 Edge Relaxation**

Ein Knoten  $z \in R$  mit minimalem Abstand  $d(u, z)$  wird in die Wolke der bisher abgearbeiteten Knoten hinzugefügt.

Ist das Label des aktuellen Knotens  $z$  größer als die Summe der Abstände  $d(s,u) + d(u,z)$  (der neu berechnete Weg ist kürzer als der bisherige Wert), wird das Label  $D[z]$  auf den neu errechneten Minimalwert gesetzt.

- if  $(D[u] + w((u,z)) < D[z])$   
     $D[z] = D[u] + w((u,z));$

## 2.3 PseudoCode

```
class Vertex {
    List edgesIn, edgesOut;
    boolean marked = false;
    int label = infinity;
    Vertex father;
}

class Edge {
    Vertex start, end;
    int weight;
}

class Algorithm {
    List vertices;
    List edges;

    void dijkstra(Vertex start) {
        // init
        List inCloud;
        PriorityQueue inBoundary;

        start.label = 0;
        inBoundary.add(start);

        while(inBoundary.notEmpty()) {
            Vertex nearest = inBoundary.removeMin();
            nearest.marked = true;
            inCloud.add(nearest);

            // Neuberechnung der direkten Nachbarn von nearest
            while(nearest.edgesOut.hasNext()) { // nutze einen Iterator
                Vertex relaxCandidate = nearest.edgesOut.next();
                if(relaxCandidate.marked)
                    continue;
                if(relaxCandidate.label != infinity)
                    inBoundary.remove(relaxCandidate);
                relax(nearest, relaxCandidate);
                inBoundary.add(relaxCandidate);
            }
        }
    }

    void relax(Vertex father, Vertex child) {
        Edge e = father.getEdgeTo(child); // die getEdgeTo() evtl. durch hashing beschleunigen
        int distance = father.label + e.weight;
        if(distance < child.label) {
            child.father = father;
            child.label = distance;
        }
    }
}
```

## 2.4 Komplexität

Anmerkung:  $m$  ist die Anzahl der Kanten,  $n = |V|$  die der Knoten.

Berechnungen in der while-Schleife:

- Suchen und löschen des ausgewählten Knotens  $v$  aus der Queue:  $O(\log n)$
- Relaxation-Procedure auf alle zu  $v$  adjazenten Knoten (die außerhalb von  $C$  liegen):  $O(\text{degree}(v)\log n)$   
(Löschen + wieder in Queue einfügen eines Knotens:  $O(\log n)$ )

à Gesamtlaufzeit des Algorithmus (worst case):

$$\sum_{v \in G} (\log n + \text{degree}(v) \log n) = \sum_{v \in G} (1 + \text{degree}(v)) \log n = O((n+m)\log n),$$

## 2.5 Beweis auf Korrektheit

Behauptung: Wenn ein  $v \in V \setminus \{s\}$  als nächst einzufügender Knoten gewählt wird, ist der Wert des Labels  $D[v]$  identisch mit der Länge des kürzesten Pfades von  $s$  nach  $v$ .

Beweis durch Widerspruch:

Sei  $v \in V \setminus \{s\}$  der erste Knoten, für den diese Behauptung nicht stimmt und sei

$$s \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_r \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_q \rightarrow v$$

ein kürzester Pfad von  $s$  nach  $v$ , dessen Länge kleiner ist als  $D[v]$ .

Es gilt  $s_1, \dots, s_r \in C$ ,  $v_1 \notin C$ .

Betrachtet man den Pfad  $s \rightarrow s_1 \rightarrow \dots \rightarrow s_r \rightarrow v_1$ , so ist dessen Länge kleiner als  $D[v]$ . Daraus folgt nun aber, dass für  $q \geq 0$   $D[v_1] < D[v]$ , was ein Widerspruch zur Wahl von  $v$  ist.

### 3. Bellman – Ford – Algorithmus

#### 3.1 Problem und Motivation

Wo der Dijkstra-Algorithmus bei negativen Kantenlängen falsche Ergebnisse liefert, hat dieser Algorithmus, basierend auf dynamic programming eine Routine zur Erkennung negativer Zyklen und handelt diese korrekt ab (falls ein vom Startknoten erreichbarer Zyklus gefunden wird, liefert das Programm false zurück – es existiert keine Lösung).

$\exists w: E \rightarrow \mathbb{R}$

#### 3.2 Lösungsidee

Die  $n$  Kanten werden (z.B. lexikographisch) geordnet und  $n-1$  mal durchlaufen und in jedem Schritt die Relaxation-Procedure aufgerufen.

(Wäre nämlich der Graph mit dem Startknoten  $s$  wie folgt aufgebaut, würde ein Fortschritt jeweils erst am Ende eines Durchlaufs erzielt werden:  $a \rightarrow b \rightarrow c \rightarrow \dots \rightarrow r \rightarrow s$ ).

Nach dieser Schleife sind alle Knoten abgearbeitet, nun wird bei jeder Kante  $(u,v) \in G$  das Label  $D[v]$  des Zielknotens mit dem des Ausgangsknotens  $D[u] + d(u,v)$  verglichen. Ist  $D[v]$  größer als das Label von  $u$  addiert mit der Distanz von  $u$  nach  $v$ , liegt ein vom Startknoten aus erreichbarer negativer Zyklus vor.

#### 3.3 PseudoCode

```
//Initialisierung
for (int i=0; i<n-1; i++)
    for (int j=0; j < lexEdges.length; j++)
        relax(u,v);
for (int i=0; i < lexEdges.length; i++)
    if (D[v] > (D[u] + w((u,v))))
        return false;
return true;
```

### 3.4 Komplexität

- 1. for-Schleife: Durchlaufen des Graphen und zuordnen der Labels:  $O(nm)$
- 2. for-Schleife: Prüfen der Werte:  $O(m)$
- ⇒ Gesamtkomplexität:  $O(nm) + O(m) = O(nm)$

### 3.5 Beweis auf Korrektheit

Sei  $v$  ein vom Startknoten  $s$  aus erreichbarer Knoten, sei  $p = \{v_0, v_1, \dots, v_k\}$  ein kürzester Pfad von  $s$  nach  $v$ , in dem  $v_0 = s$  und  $v_k = v$  gilt. Der Pfad ist kein Zyklus, also  $k \leq |V| - 1$ .

#### Beweis durch Induktion:

- Behauptung: (gültig, da  $|V| - 1$  Durchläufe)  
 $i = 0, 1, \dots, k$ :  $D[v_i] = d(s, v_i)$  nach dem  $i$ -ten Durchlauf über die Kanten von  $G$
- Induktionsanfang:  $i=0$ :  
 $D[v_0] = d(s, v_0) = 0$  nach der Initialisierung
- Induktionsschritt:  $i \Rightarrow i+1$ :  
 $D[v_{i-1}] = d(s, v_{i-1})$  nach dem  $(i-1)$ ten Durchlauf  
Kante  $(v_{i-1}, v_i)$  wird im  $i$ -ten Durchlauf relaxed,  
⇒  $D[v_i] = d(s, v_i)$  nach dem  $i$ -ten und folgenden Durchläufen

## 4. größte gemeinsame Teilfolge (LCS)

### 4.1 Vorteile und Motivation

Der longest-common-subsequence Algorithmus basiert wie der Bellman-Ford-Algorithmus auf dynamic programming und besitzt – im Gegensatz zu weniger effizienten ähnlichen Algorithmen - keine exponentiale Laufzeit, da die Sequenzen geschickt abgearbeitet werden. Der Motivation besteht darin, aus zwei Strings (repräsentiert durch eine Matrix) die Länge der längsten Sequenz aufeinanderfolgender (nicht unbedingt zusammenhängender) übereinstimmender Characters möglichst effizient zu bestimmen.

### 4.2 Lösungsidee und Pseudocode

```
//Initialisierung (-1. Spalte/Zeile der Matrix werden mit 0 initialisiert)
String str1, str2;
int n=str1.length();
int m = str2.length();

for (int i=-1;i<n-1;i++)
    L[i,-1]=0;
for (int j=0;j<m-1;j++)
    L[-1,j]=0;

//sequentielles Abarbeiten der Matrix
for (i=0;i<=n;++i)
    for (j=0;j<=m;++j)
        if (str1.charAt(i) == str2.charAt(j))
            L [i] [j] = L [i-1] [j-1] + 1;
        else
            L [i] [j] = max (L [i-1] [j], L [i] [j-1]);
```

Stimmen zwei Characters an den Stellen  $i$  und  $j$  überein, nimmt das aktuelle Feld  $L[i][j]$  in der Matrix  $L$  den Wert des schräg links darüber stehenden Feldes  $L[i-1][j-1]$  um eins erhöht an (+1, da die Übereinstimmung mitgezählt werden muß). Dort ist die Länge der bisherigen größten gemeinsamen Teilfolge gespeichert, die die Strings beim Vergleich ihrer jeweils letzten Characters hatten.

Im Fall, dass die beiden Stellen nicht übereinstimmen, wird die Anzahl der bis dahin übereinstimmenden Characters der letzten verglichenen Positionen des einen Strings mit der des anderen verglichen und die höhere der beiden Zahlen übernommen.

Im Beispiel wird das Maximum aus  $L[i-1][j]$  und  $L[i][j-1]$  gewählt.

Dadurch werden bisher aufgetretene Teilfolgen nicht „vergessen“, im zuletzt bearbeiteten Feld, dass die jeweils letzten Stellen der Folgen vergleicht, kann die Länge der größten Teilfolge abgelesen werden.

Nun kann man zusätzlich noch durch Backtracking die Characters der Teilfolge bestimmen, indem man von  $i=n$  und  $j=m$  aus folgende Abfrage startet:

```
//Backtracking
String lcs=null;
int i=n, j=m;
while (i>=0 || j>=0) {
    if (str1.charAt(i)==str2.charAt(j)) {
        lcs=toString(str1.charAt(i))+lcs;
        i--; j--;
    } else
        j--;
}
return lcs;
```

### 4.3 Komplexität

Ohne Backtracking liegt die Komplexität bei  $O(n) + O(m) + O(n*m) = O(n*m)$ , bedingt durch die geschachtelten for-Schleifen - die Laufzeit der Initialisierungsschritte kann wieder vernachlässigt werden. Dies ist bei weitem schneller als einfache Algorithmen, deren Laufzeit sogar exponential ansteigen kann.

### III Literaturnachweis

A. Aho; J. E. Hopcroft und J. D. Ullman:  
Data Structures and Algorithms,  
Addison-Wesley Publishing Company, Reading Massachusetts, 1983.  
80/ST 270 A286.983

A. Aho; J. D. Ullman:  
Informatik. Datenstrukturen und Konzepte der Abstraktion,  
International Thomson Publishing, Bonn [u.a.], 1996. 80/ST 110 A286 I4

T. H. Cormen; C. E. Leiserson; R. L. Rivest:  
Introduction to Algorithms  
The MIT Press, Cambridge, Mass. u.a., 1990. 80/ST 130 C811

M. T. Goodrich, R. Tamassia:  
Data Structures and Algorithms in Java,  
John Wiley & Sons, Inc., 1998. 17/ST 270 G655

T. Ottmann; P. Widmayer:  
Algorithmen und Datenstrukturen,  
Spektrum Verlag Heidelberg, 1990. 17/ST 270 O91+2

[www.tu-muenchen.de](http://www.tu-muenchen.de)  
Effiziente Algorithmen & Datenstrukturen