

Datenstrukturen und Algorithmen

Vorlesung 17: Kürzeste Pfadalgorithmen

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsal10/>

2. Juli 2010



Übersicht

1 Kürzeste Pfade

2 Bellman-Ford

3 Dijkstra

1 Kürzeste Pfade

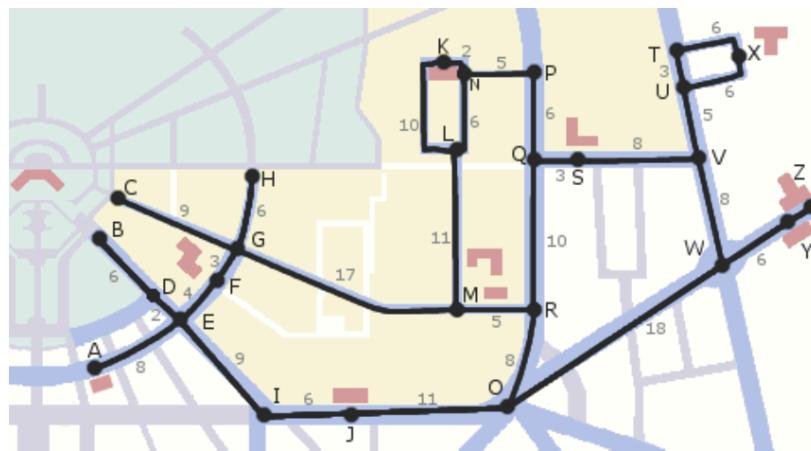
2 Bellman-Ford

3 Dijkstra



Andere Rechenprobleme: kürzester Weg

Andere Rechenprobleme: kürzester Weg



Kürzeste Pfade

Es gibt verschiedene Varianten:

- ▶ Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten:
[Single-Source Shortest Paths \(SSSP\)](#).
- ▶ Kürzeste Pfade von allen Knoten zu einem Zielknoten t .
Lässt sich auf SSSP zurückführen.
- ▶ Kürzeste Pfade für *ein* festes Knotenpaar u, v .
Es ist kein Algorithmus bekannt, der asymptotisch schneller als der beste SSSP-Algorithmus ist.
- ▶ Kürzeste Pfade für *alle* Knotenpaare.
[All-Pairs Shortest Paths](#) (nächste Vorlesung).

Andere Rechenprobleme: kürzester Weg

Beispiel (kürzester Weg)

- Eingabe:**
1. Eine Straßenkarte, auf der der Abstand zwischen jedem Paar benachbarter Kreuzungen eingezeichnet ist,
 2. eine Startkreuzung s , und
 3. eine Zielkreuzung t .

Ausgabe: Der kürzeste Weg von s nach t .

Single-Source Shortest Paths

Gegeben sei ein gewichteter (gerichteter oder ungerichteter) Graph G .

Das Gewicht eines Weges ist die [Summe](#) der Gewichte seiner Kanten.

Problem (Single-Source Shortest Path)

*Finde den Weg mit **minimalem Gewicht**, von Knoten s (Quelle / source) aus zu jedem anderen Knoten aus G .*

Übersicht

1 Kürzeste Pfade

2 Bellman-Ford

3 Dijkstra

Der Bellman-Ford Algorithmus

- ▶ Kürzeste Pfade bei einem **einzigem** Startknoten.
- ▶ Erlaubt **negative** Kantengewichte.
- ▶ Es zeigt an, ob es ein **Zyklus mit negativem Gewicht** gibt, der vom Startknoten aus erreichbar ist.
- ▶ Falls einen solchen Zyklus gefunden wird, gibt es **keine** Lösung.
 - ▶ da die Gewichte der kürzesten Pfade nicht mehr wohldefiniert sind.
- ▶ Sonst, bestimmt der Algorithmus die kürzeste Pfade mit ihren Gewichten.
- ▶ Es berechnet (iterativ) **Schätzungen** $d[v]$ für die Gewichte des kürzesten Pfades vom Startknoten nach v .

Bellman-Ford – Idee

- ▶ Wir wollen die Abstände aller Knoten $v \in V$ zum Startknoten $start$ bestimmen.
- ▶ Dazu initialisieren wir $d[v]$ mit ∞ , sowie $d[start]=0$.
- ▶ Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$:
 - ▶ Ist der bisher bekannte Abstand $d[w]$ größer als $d[v]+W(v, w)$, dann verbessere $d[w]$ auf diesen Wert (**Relaxierung**).
- ▶ Wiederhole vorigen Schritt, bis sich nichts mehr ändert, bzw. breche ab, falls es einen negativen Zyklus gibt.

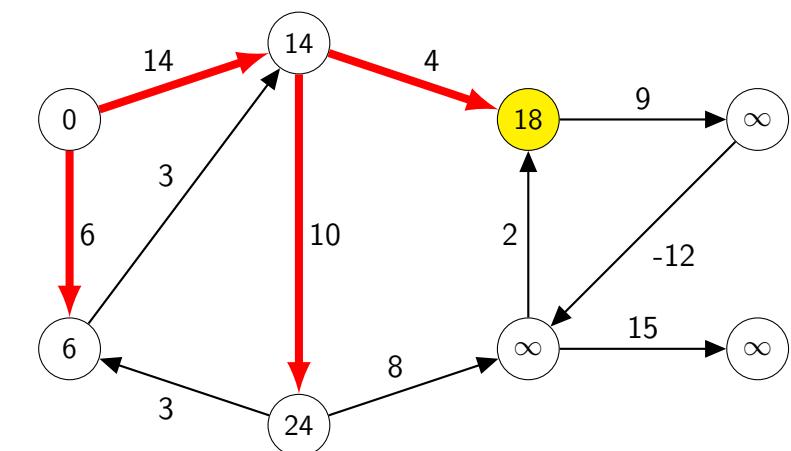
Theorem

Wenn nach $|V|-1$ Wiederholungen noch Verbesserungen möglich sind, dann gibt es einen negativen Zyklus. Andernfalls enthält $d[v]$ für alle v den kürzesten Abstand zum Startknoten.

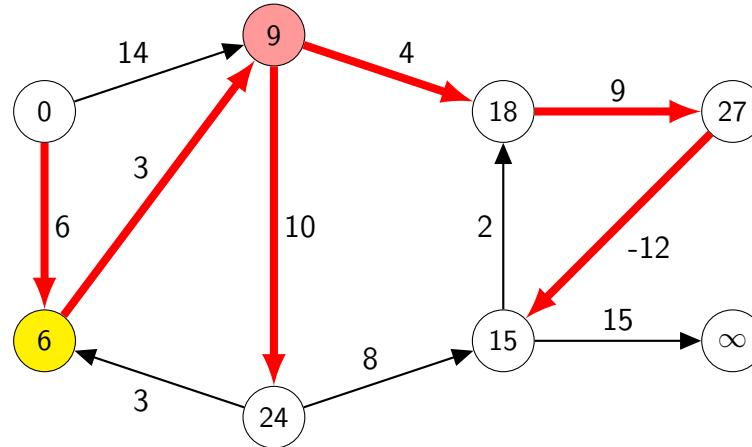
Beweisidee:

- ▶ Ein Pfad in (V, E) kann höchstens die Länge $|V|-1$ haben.

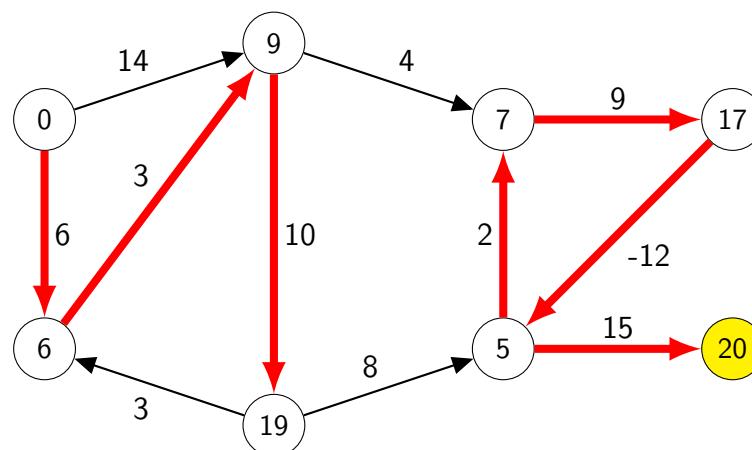
Bellman-Ford – Beispiel



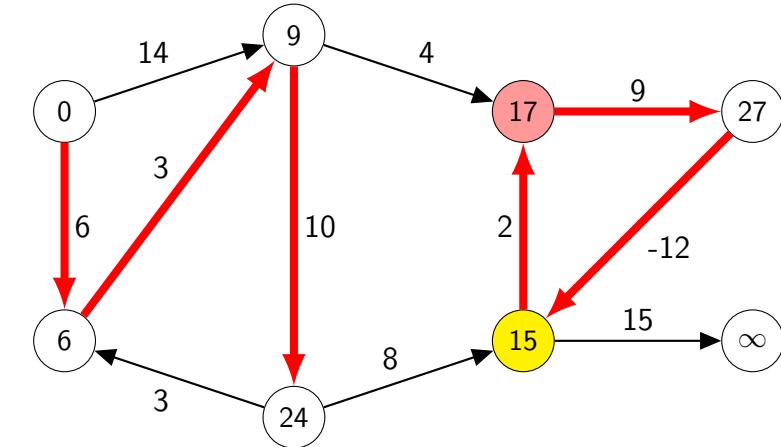
Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Bellman-Ford – Implementierung

```

1 // Keine Zyklen mit negativem Gewicht?
2 bool bellFord(List adjLst[], int n, int start) {
3     int d[n] = +inf;
4     d[start] = 0;
5     for (int i = 1; i < n; i++) // n-1 Durchläufe
6         for (int v = 0; v < n; v++) // alle Kanten
7             foreach (edge in adjLst[v])
8                 if (d[edge.w] > d[v] + edge.weight) {
9                     d[edge.w] = d[v] + edge.weight;
10                }
11    for (int v = 0; v < n; v++) // alle Kanten
12        foreach (edge in adjLst[v])
13            if (d[edge.w] > d[v] + edge.weight)
14                return false; // "noch kürzerer Weg"
15    return true;
16 }
```

- ▶ Erweiterbar durch Speichern des Vorgängers auf die Rückgabe der kürzesten Wege (Übung).
- ▶ Komplexität: $O(|V| \cdot |E|) = O(n \cdot m) \in O(n^3)$.

Übersicht

1 Kürzeste Pfade

2 Bellman-Ford

3 Dijkstra

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (BLACK, GRAY, WHITE) ein:

Baum-knoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- ▶ Fange mit einem Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.
- ▶ Finde die Kante mit kürzestem Abstand von s , die den bisherigen Baum verlässt.
- ▶ Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- ▶ Fahre fort, bis keine weiteren Randknoten mehr vorhanden sind.

Genauso wie der MST-Algorithmus von Prim handelt es sich um einen Greedy-Algorithmus.

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geq 0$.

- ▶ Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus basiert auf folgender Eigenschaft:

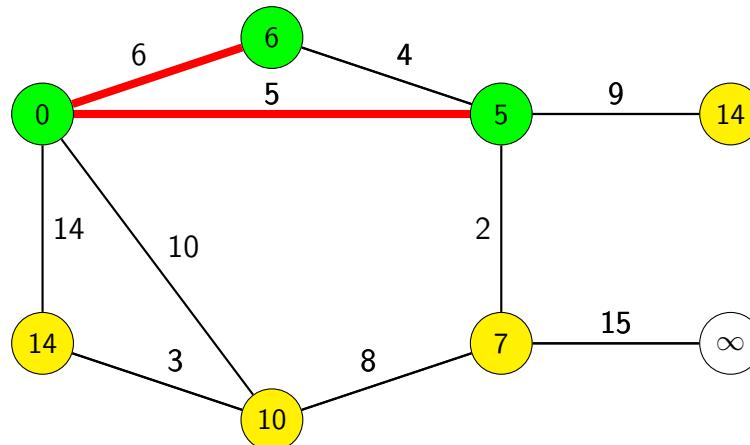
- ▶ Angenommen der kürzeste Pfad von x nach z geht über Knoten y .
- ⇒ Dann ist der Teilstieg von x nach y auch ein kürzester Pfad von x nach y .
- ⇒ Auch der Teilstieg von y nach z ist ein kürzester Pfad von y nach z .

Dijkstras Kürzeste-Wege-Algorithmus – Grundgerüst

```

1 // ungerichteter Graph G mit n Knoten
2 void dijkstraSP(Graph G, int n) {
3     initialisiere alle Knoten als ungesehen (WHITE);
4     markiere s als Baum (BLACK) und setze d(s,s) = 0;
5     reklassifiziere alle zu s adjazenten Knoten als Rand (GRAY);
6     while (es gibt Randknoten) {
7         wähle von allen Kanten zwischen einem Baumknoten t und
8             einem Randknoten v die mit minimalem d(s,t) + W(t,v);
9         reklassifiziere v als Baum (BLACK);
10        füge Kante tv zum Baum hinzu;
11        setze d(s,v) = d(s,t) + W(t,v);
12        reklassifiziere alle zu v adjazenten ungesesehenen Knoten
13            mit Rand (GRAY);
14    }
15 }
```

Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Beweis

Beweis.

Sei P der kürzeste Weg von s nach y , erweitert um die Kante (y, z) , die „am nahesten an s liegt“.

Sei nun $P' = s, z_1, \dots, z_k, \dots, z$ der kürzeste Weg von s nach z , wobei z_k der erste Knoten $\notin V'$ ist.

- Wegen der Wahl von (y, z) gilt:

$$W(P) = d(s, y) + W(y, z) \leq d(s, z_{k-1}) + W(z_{k-1}, z_k).$$

- Da s, z_1, \dots, z_k ein Prefix von P' ist und alle verbleibenden Kanten *nicht-negatives* Gewicht haben, gilt:

$$d(s, z_{k-1}) + W(z_{k-1}, z_k) \leq W(P')$$

⇒ Daher ist $W(P) \leq W(P')$, d. h. P ist der kürzeste Weg! □

Theorem

Sei $s \in V' \subseteq V$ mit kürzestem Abstand $d(s, y)$ von s nach $y \in V'$.

Theorem

Wenn (y, z) die Kante mit minimalem $d(s, y) + W(y, z)$ über alle Kanten mit $y \in V'$ und $z \in V \setminus V'$ ist, dann ist der zusammengesetzte Weg bestehend aus dem kürzesten Weg von s nach y gefolgt von der Kante (y, z) auch der kürzeste Weg von s nach z .

Korrektheit

Theorem (Korrektheit)

Dijkstras kürzeste-Wege-Algorithmus berechnet den kürzesten Abstand von Knoten s zu jedem von s erreichbaren Knoten in G .

Beweis.

Induktion nach der Sequenz von hinzugefügten Knoten im SSSP-Baum.
(Übung.) □

Dijkstras Kürzeste-Wege-Algorithmus – Implementierung (I)

```

1 // Wie Prim. Ergebnis als Vorgängerliste(-baum) in .parent
2 // curWeight[v] enthält gerade d(s,v)
3 VertexState[n] dijkSP(List adjLst[n], int n, int start) {
4     VertexState state[n] = // (eigentlich im Konstruktor von pq)
5         { color: WHITE, parent: -1, curWeight: +inf };
6     PriorityQueue pq = VS_PriorityQueue<&VS.curWeight>(&state);
7
8     pq.insert(start, {parent: -1, curWeight: 0});
9     while (!pq.isEmpty()) { // solange es Randknoten gibt
10         int v = pq.getMin(); // günstigste Kante, bzw. Randknoten
11         pq.delMin(); // setzt auch Farbe auf BLACK
12         updateFringe(pq, adjList, v); // update den Rand
13     }
14     return state;
15 }
```

Eigenschaften von Dijkstras Algorithmus

- ▶ Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.
- ▶ Implementierung: Ähnlich dem Algorithmus von Prim.
- ▶ Zeitkomplexität im Worst-Case: $\Theta(|V|^2)$.
- ▶ Untere Schranke der Komplexität: $\Omega(|E|)$.
 - ▶ da im schlimmsten Fall alle Kanten überprüft werden müssen.
- ▶ Platzkomplexität: $O(|V|)$.

Dijkstras Kürzeste-Wege-Algorithmus – Implementierung (II)

```

1 void updateFringe(PriorityQueue &pq, List adjLst[], int v) {
2     // kürzester Weg von s nach v
3     float ownWeight = pq.getWeight(v);
4     foreach (edge in adjLst[v]) {
5         // Distanz von s nach w über v
6         float newWeight = edge.weight + ownWeight;
7
8         if (pq.getColor(edge.w) == WHITE) { // -> GRAY
9             pq.insert(edge.w, {parent: v, curWeight: newWeight});
10        } else if (pq.getColor(edge.w) == GRAY) {
11            if (newWeight < pq.getWeight(edge.w)) {
12                // Randknoten-update: Weg über v ist besser
13                pq.decrKey(edge.w, {parent: v, curWeight: newWeight});
14            }
15        }
16    }
17 }
```

Proseminar Datenstrukturen und Algorithmen

Wir veranstalten ein **Proseminar Datenstrukturen und Algorithmen**.
Wann? im WS 2010/11.

Themen:

Splay Trees, Fibonacci Heaps, Google's Pagerank Algorithm, AVL-Bäume, Huffman Kodierung, Cocktail Sort, B-Bäume, Deutsch-Schorr-Waite Baumtraversierung, Bit-State Hashing, Binäre Entscheidungsdiagramme, k -Kürzeste Pfad Problem, Amortisierte Komplexität, Smoothed Analysis, String Matching, Spielbäume (Minimax-Verfahren, $\alpha - \beta$ Pruning), Lineare Programmierung.

Anmeldung: 26. Juni bis 10. Juli über die übliche Webseiten.