RUB

Vorlesung Datenstrukturen

Kürzeste Wege

Maike Buchin

4. und 6.7.2017

Einführung

RUB

Motivation: Bestimmung von kürzesten Wegen ist in vielen Anwendungen, z.B. Routenplanung, ein wichtiges Problem.

Allgemeine Form: Gegeben ein Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}$. Die Kosten eines Pfades in G ist die Summe der Kosten seiner Kanten. Gesucht sind kürzeste Wege zwischen zwei (oder mehr) Knoten in V.

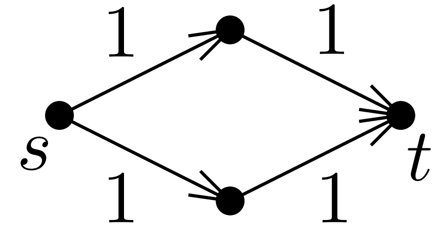
Effiziente Algorithmen für eingeschränkte Varianten: nicht-negative Kantenkosten, ganzzahlige Kantenkosten, azyklische Graphen.

Einen Spezialfall können wir schon effizient lösen: Kantengewichte $= 1$. Dann berechnet BFS in $O(m + n)$ Zeit alle kürzesten Wege von einem Startknoten aus.

10.1 Von Grundbegriffen zur allg. Methode

RUB

- Kosten eines Weges = Summe der Kosten seiner Kanten
- leerer Weg hat Kosten 0
- kürzeste Wege sind nicht unbedingt eindeutig



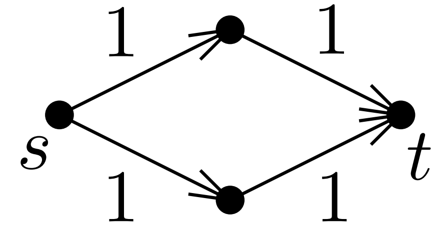
Beobachtung: Wenn G negative Kreise enthält (Kreise mit negativer Länge), existieren kürzeste Wege nicht unbedingt

Lemma: Wenn G keine negativen Kreise enthält, und v von s aus erreichbar ist, dann gibt es einen einfachen kürzesten Weg von v nach s.

Beweis: Wähle p_0 kürzesten Weg unter allen endlich vielen einfachen kürzesten Wegen von s nach v. Aus der Annahme, dass es einen kürzeren, nicht einfachen Weg gibt, lässt sich die Existenz eines negativen Kreises folgern.

RUB

- Kosten eines Weges = Summe der Kosten seiner Kanten
- leerer Weg hat Kosten 0
- kürzeste Wege sind nicht unbedingt eindeutig



Definition:

$$\mu(s,v) = \begin{cases} +\infty & \text{falls } v \text{ nicht von } s \text{ aus erreichbar} \\ c(P) & \text{falls kürzester Weg } P \text{ existiert} \\ -\infty & \text{sonst} \end{cases}$$

Für festen Startknoten s schreiben wir auch einfach $\mu(v)$.

Eigenschaften:

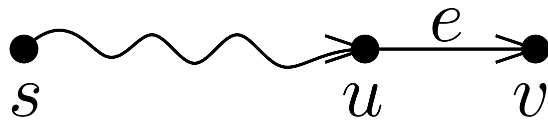
- Teilwege von kürzesten Wegen sind selbst kürzeste Wege.
- sind alle Knoten von s aus erreichbar, dann gibt es Baum mit Wurzel s und alle Wege im Baum sind kürzeste Wege.

10.1 Von Grundbegriffen zur allg. Methode

RUB

Strategie zum Finden kürzester Wege ist Verallgemeinerung der BFS. Zwei Arrays d und p speichern *Schätzdistanz* und *Vorgänger* nach aktuellem Wissen.
Initialisierung: $d[s] = 0, p[s] = s$, $d[v] = \infty, p[v] = \perp$ für $v \neq s$

Grundlegende Idee: Informationen von Kanten weitergeben.



Dann ist $d[v] \leq d[u] + c(e)$.
Dies nennt man eine *Kantenrelaxierung*.

Lemma: Wenn nach einer beliebigen Folge von Kantenrelaxierungen $d[v] < \infty$ gilt, dann gibt es einen Weg von s nach v der Länge $d[v]$.
Beweis: Induktion über die Anzahl der Relaxierungen.

Vorgehen: relaxiere wiederholt Kanten bis entweder ein negativer Kreis oder für jeden Knoten ein kürzester Weg gefunden.

10.1 Von Grundbegriffen zur allg. Methode

RUB

Beobachtung: Ist $P = \langle e_1, \ldots, e_k \rangle$ ein einfacher Weg von s nach v, und relaxieren wir die Kanten in dieser Reihenfolge, dann gilt $d[v] = c(P)$.

Lemma: Nach Ausführung einer Folge R von Relaxierungen die einen kürzesten Weg $P = \langle e_1, \ldots, e_k \rangle$ von s nach v als Teilfolge enthält, gilt $d[v] = \mu(v) = c(P)$. Weiter gilt, dass ein kürzester Weg durch die Parent-Zeiger gegeben wird.

Beweis:

$$R = \langle \ldots, \overset{t_1}{e_1}, \ldots, \overset{t_2}{e_2}, \ldots, \overset{t_k}{e_k}, \ldots \rangle$$

Da P kürzester Weg ist, gilt $c(P) = \mu(v)$.

Für $i = 1, \ldots, k$ sei v_i der Zielknoten von e_i. Sei $t_0 = 0$ und $v_0 = s$. Wir zeigen induktiv, dass ab der Relaxierung zum Zeitpunkt t_i gilt $d[v_i] = \sum_{j=1}^{i} c(e_j)$.

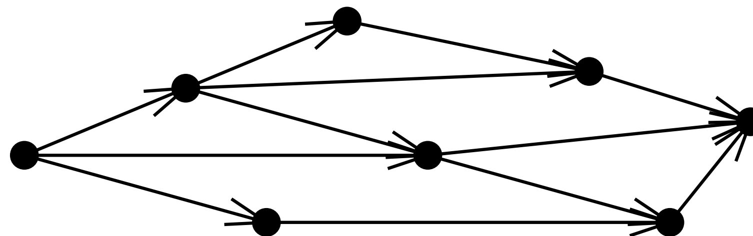
10.2 Gerichtete azyklische Graphen

RUB

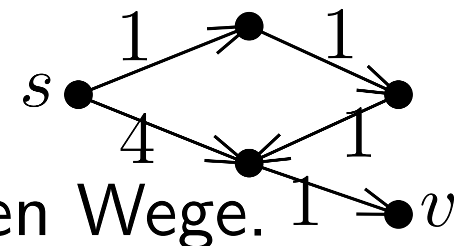
Vorgehen: Da es keine (insbesondere negativen) Kreise gibt, existieren kürzeste Wege. Wähle topologische Sortierung der Knoten. Dann gilt für $(v_i, v_j) \in E$ stets $i < j$, d.h. Knoten eines jeden Weges sind bzgl. der top. Sortierung aufsteigend. Also können wir nach vorherigem Lemma kürzeste Wege berechnen, indem wir erst alle Kanten ausgehend von v_1, dann v_2 usw. relaxieren.

Satz: Im azyklischen Graphen können kürzeste Wege in $O(m + n)$ Zeit berechnet werden.

Beispiel:



topologische Sortierung durch x-Koordinate geg.



Beobachtung: Relaxierung in BFS oder DFS Reihenfolge findet im Allgemeinen keine kürzesten Wege.

10.3 Nichtnegative Kantenkosten (Der Algorithmus von Dijkstra)

RUB

Wenn es keine negativen Kantenkosten gibt, gibt es auch keine negativen Kreise und für alle von s aus erreichbaren Knoten existieren kürzeste Wege.

Ziel: Bei geeigneter Reihenfolge der Relaxierung genügt es jede Kante einmal zu relaxieren.

Beobachtung: Entlang eines kürzesten Weges sind die kürzeste Wege Distanzen schwach monoton steigend.

Idee: Knoten in der Reihenfolge wachsender Distanzen zu s bearbeiten (d.h. ausgehende Kanten relaxieren).

Umsetzung: Im Algorithmus kennen wir die Distanzen noch nicht. Aber es genügt, den nächsten in der Reihenfolge zu kennen, und dieser ist Nachbar eines bekannten Knotens, und hat die kleinste Schätzdistanz.

10.3 Nichtnegative Kantenkosten
(Der Algorithmus von Dijkstra)

RUB

markiere alle Knoten als *unbearbeitet*;
initialisiere d und p;
while es gibt unbearbeiteten Knoten u mit $d[u] \leq \infty$ **do**
  $u :=$ ein solcher Knoten mit minimaler Schätzdistanz $d[u]$
  relaxiere alle von u ausgehenden Kanten (u, v)
  markiere u als *bearbeitet*

Satz: Der Algorithmus von Dijkstra löst das kürzeste Wege Problem mit einem Startknoten für Graphen mit nicht negativen Kosten.

Beweis: Wir zeigen Korrektheit in zwei Schritten:
1. alle Knoten werden bearbeitet
2. wenn ein Knoten bearbeitet wird, ist seine Schätzdistanz seine tatsächliche Distanz

10.3 Nichtnegative Kantenkosten
(Der Algorithmus von Dijkstra)

RUB

1. Schritt: alle von s aus erreichbaren Knoten werden bearbeitet

Wir sagen, dass ein Knoten *gefunden* wird, wenn seine Schätzdistanz auf $< \infty$ gesetzt wird. Der Algorithmus stellt sicher, dass jeder gefundene Knoten bearbeitet wird. Also g.z.z., dass jeder Knoten gefunden wird. Sei $P = \langle s = v_1, \ldots, v_k = v \rangle$ ein Weg von s nach v.

Induktion(i): v_i wird gefunden.
$s = v_1$ wird gefunden, da $d[s] = 0$ intialisiert wird.
Nach I.V. wird v_{i-1} gefunden, also $d[v_{i-1}] < \infty$.
Beim Bearbeiten von v_{i-1} wird die Kante (v_{i-1}, v_i) relaxiert, und dabei v_i gefunden.

10.3 Nichtnegative Kantenkosten (Der Algorithmus von Dijkstra)

RUB

2. Schritt: wenn ein Knoten u bearbeitet wird, gilt $d[u] = \mu(u)$

Durch **Induktion(t)** über die Schleifendurchläufe ("Runden") zeigen wir: wenn in Runde t Knoten u bearbeitet wird, dann gilt $d[u] \leq \mu(u)$. [Nach vorherigem Lemma gilt $d[u] \geq \mu(u)$.]

Behauptung gilt für $t = 1$, da $d[s] = 0 = \mu(s)$ initialisiert wird. Sei $t > 1$ und sei der aktuelle Knoten $u \neq s$. Wähle beliebigen Weg $P = \langle s = v_1, \ldots, v_k = u \rangle$ nach u. Sei v_i, $1 < i \leq k$, der erste Knoten auf P, der nicht vor Runde t bearbeitet wird. Dann wird v_{i-1} vor Runde t bearbeitet und dabei (v_{i-1}, v_i) relaxiert. Anschließend gilt:

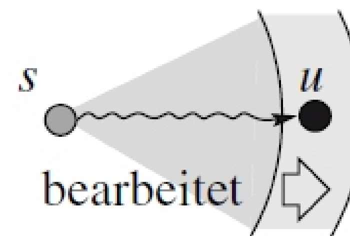
$$\begin{aligned} d[v_i] &\leq d[v_{i-1}] + c(v_{i-1}, v_i) \overset{\text{IV}}{\leq} \mu(v_{i-1}) + c(v_{i-1}, v_i) \\ &\leq c(\langle v_1, \ldots v_{i-1} \rangle) + c(v_{i-1}, v_i) = c(\langle v_1, \ldots v_i \rangle) \leq c(P) \end{aligned}$$

Da u in Runde t bearbeitet wird, gilt $d[u] \leq d[v_i] \leq c(P)$.
Da P beliebig, folgt schließlich $d[u] \leq \mu(u)$.

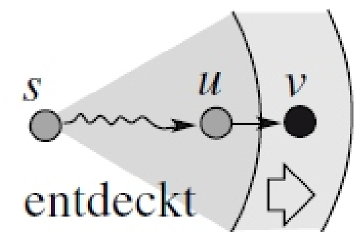
10.3 Nichtnegative Kantenkosten (Der Algorithmus von Dijkstra)

Umsetzung des Algorithmus: unbearbeitete Knoten werden in address. Prioritätswarteschlange mit Schätzdistanzen als Schlüssel gespeichert.

Function $Dijkstra(s : NodeId) : NodeArray \times NodeArray$ // gibt $(d, parent)$ zurück
  $d = \langle \infty, \ldots, \infty \rangle : NodeArray$ **of** $\mathbb{R} \cup \{\infty\}$ // Schätzdistanzen
  $parent = \langle \bot, \ldots, \bot \rangle : NodeArray$ **of** $NodeId$
  $parent[s] := s$ // Schleife signalisiert Wurzel
  $Q : NodePQ$ // unbearbeitete gefundene Knoten
  $d[s] := 0; \quad Q.insert(s)$
  while $Q \neq \emptyset$ **do**
    $u := Q.deleteMin$ // es gilt $d[u] = \mu(u)$



    foreach $edge\ e = (u, v) \in E$ **do**
      if $d[u] + c(e) < d[v]$ **then** // relaxiere
        $d[v] := d[u] + c(e)$
        $parent[v] := u$ // aktualisiere Baum
        if $v \in Q$ **then** $Q.decreaseKey(v)$
        else $Q.insert(v)$



  return $(d, parent)$

10.3 Nichtnegative Kantenkosten (Der Algorithmus von Dijkstra)

RUB

Laufzeit: $T_{Dijkstra} = \mathrm{O}(\underbrace{m \cdot T_{decreaseKey}(n)}_{\text{jede Kante wird einmal relaxiert}} + \underbrace{n \cdot (T_{deleteMin}(n) + T_{insert}(n))}_{\text{jeder Knoten wird einmal eingefügt und gelöscht}})$

Damit ist die Laufzeit bei Verwendung von

- Binärheaps $O((m+n)\log n)$
- Fibonacciheaps $O(m + n\log n)$

Weitere Möglichkeiten der Analyse:

- Durchschnittsanalyse ($\rightarrow$ 10.4 im Buch)
- Verwendung von monotonen Prioritätswarteschlangen für ganzzahlige Kantenkosten ($\rightarrow$ 10.5 im Buch)

10.5 Monotone Prioritätswarteschlangen

RUB

Beobachtung: Der Algorithmus von Dijkstra bearbeitet Knoten in der Reihenfolge nichtfallender (Schätz)distanzen. Daher genügt eine *monotone Prioritätswarteschlange*, bei der extrahierte Schlüssel schwach monoton steigen.

Nehmen wir zusätzlich an, dass die Kantenkosten ganze Zahlen aus $0 \ldots C$ sind, lässt sich dies effizient umsetzen.

Da kürzeste Wege einfach sind, haben sie Länge maximal $C(n-1)$. Zu jedem Zeitpunkt stehen nur Werte in $\min \ldots \min + C$ in der Prioriätswarteschlange, wobei $\min$ der letzte entnommen Wert. (Beweise dies induktiv nach Runde.)

10.5 Monotone Prioritätswarteschlangen

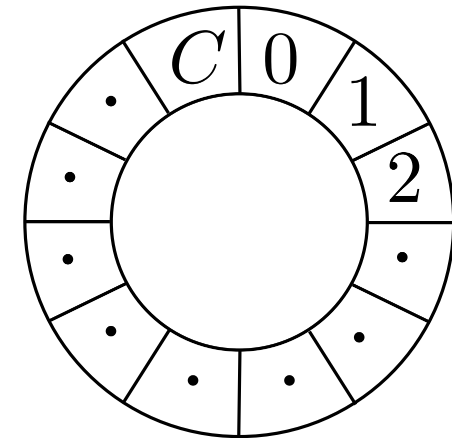
RUB

10.5.1 Behälterwarteschlange

Eine Behälterwarteschlange (engl. bucket queue) ist ein zirkuläres Array B von $C+1$ vielen doppelt-verketteten Listen. Alle Zahlen der Form $i+(C+1)j$ werden auf i abgebildet.

Ein Knoten v mit Schätzdistanz $d[v]$ wird in Behälter $B[d(v) \mod (C+1)]$ gespeichert.

Da alle Schätzdistanzen in $\min \ldots \min + C$ liegen, haben Knoten in gleichem Behälter auch tatsächlich gleiche Schätzdistanzen.



Implementierung:

- Initialisierung: erzeuge $C+1$ leere Listen, $\min = 0$
- insert(v): füge v in Behälter $B[d(v) \mod (C+1)]$ ein
- decreaseKey(v): entferne v aus Behälter und fügt es neu ein
- deleteMin: bearbeite alle Knoten in $\min$-Behälter; $\min{++}$

Laufzeit: $O(m + nC)$