

Dynamic Shortest Path

VO Algorithm Engineering

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

9. VO

22.11.2005

Literatur für diese VO

- C. Demetrescu und G.F. Italiano: A new approach to dynamic all pairs shortest paths, Proc. 35th Annual ACM Symposium on Theory of Computing (STOC '03), San Diego, 2003, 159-166.
- Zeitschriftenversion: Journal of the Association for Computing Machinery (JACM), vol. 51 (6), 2004, 968-992.

Überblick

Einführung Dynamische Algorithmen

Einführung Dynamischer APSP

Algorithmus-Idee

Eigenschaften von LSPs

Algorithmus Increase-Only

Analyse: Korrektheit + Laufzeit

Dynamische Algorithmen

- Ein dynamischer Algorithmus erhält eine geg. Eigenschaft P eines gewichteten Graphen während **dynamischer Änderungen des Graphen**, z.B.
 - Einfügen neuer Kanten,
 - Entfernen von Kanten und
 - Kosten-Änderungen der Kanten.
- Ein dynamischer Algorithmus sollte **Anfragen** auf Eigenschaft P **schnell beantworten** können, und
- **Update-Operationen schneller** bearbeiten als ein statischer Algorithmus, der jedesmal alles von vorn berechnen muss.

Dynamische Algorithmen

- Wir betrachten o.E. nur Kosten-Änderungen der Kanten.
- Ein dynamischer Algorithmus heißt **voll-dynamisch (full dynamic)** wenn er für Erhöhungen als auch für Verminderungen von Kantenkosten geeignet ist.
- Sonst heißt er **teil-dynamisch (partially dynamic)**.

Dynamische APSP

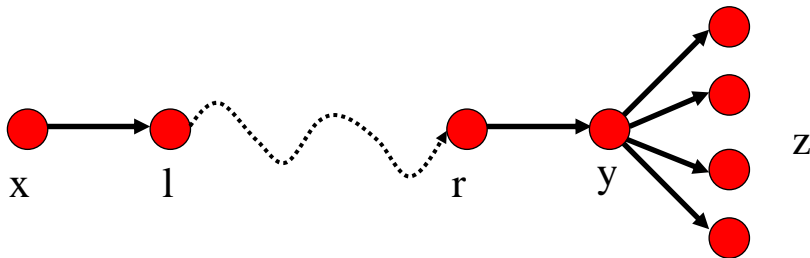
- Geg.: Gerichteter Graph $G=(V,E)$ mit nicht-negativen Kantenkosten $w(e)\in\mathbb{R}$, sowie eine Sequenz der folgenden Operationen:
 - **update(v,w')**: ändere die Kosten aller zu Knoten v inzidenten Kanten zur neuen Kostenfunktion w' (verallgemeinerte Version von dyn. APSP)
 - **distance(x,y)**: gib die Distanz zwischen Knoten x und y zurück
 - **path(x,y)**: gib den kürzesten Weg von x nach y aus, falls einer existiert.

Historisches

- Seit 1967 studiert:
 - „Highway Research“, „Transport Network Theory“
- King 1999:
 - erster voll-dynamischer Algorithmus, der im worst case schneller als der statische Algorithmus war;
 - allerdings nur für ganzzahlige Kosten $\leq C$;
 - Laufzeit: $O(n^{2.5}(C \log n)^{0.5})$ per update & optimal per Query
- Demetrescu und Italiano 2003:
 - erster voll-dynamischer Algorithmus ohne Einschränkungen, der im worst case schneller als der statische Algorithmus ist
 - Laufzeit: $O(n^2 \log^3 n)$ amort. per update & optimal per Query

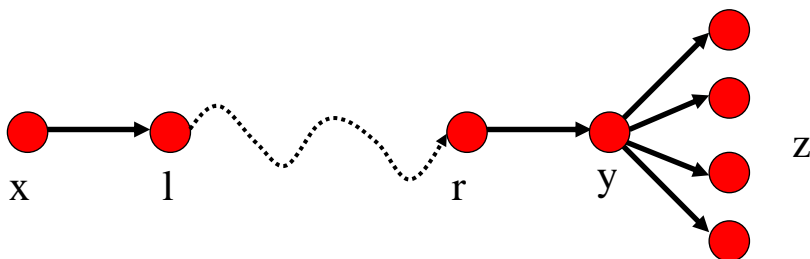
Idee: mittels LSP

Locally Shortest Paths



- Ein Weg $P(x \rightarrow z)$ heißt „Lokal Kürzester Weg“ („Locally Shortest Path“, LSP) in G , falls entweder
 - $P(x \rightarrow z)$ aus einem einzigen Knoten besteht, oder
 - jeder echte Teilweg von $P(x \rightarrow z)$ ein kürzester Pfad in G ist.

Locally Shortest Paths



Nach einem Update(v, w') sind

- einige kürzeste Wege vor dem Update nicht mehr die kürzesten nach dem Update
- andere Wege, die vorher nicht kürzeste Wege waren, werden jetzt die kürzesten

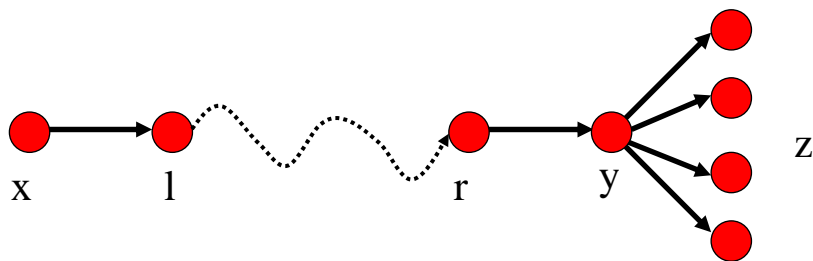
• ZIEL: Finde die neuen kürzesten Wege

• Natürliche Kandidaten hierfür: LSP

Algorithmus-Idee

- Halte alle LSPs in einer Datenstruktur DS, so dass die Ersatzpfade schnell gefunden werden können.
- Frage: Wie teuer ist die Verwaltung einer solchen DS?
 - Wieviele LSPs können per Update neu dazu kommen?
 - Wieviele LSPs können per Update herausfallen?

Definitionen



- Notation für einen Weg: $P(x \rightarrow z)$
- $l(P(x \rightarrow z))$: „linker“ Teilweg, d.h. $P(x \rightarrow z)$ ohne Kante (y, z)
- $r(P(x \rightarrow z))$: „rechter“ Teilweg, d.h. $P(x \rightarrow z)$ ohne Kante (x, l)
- LSP: Locally shortest path
- SP: Shortest path

Eigenschaften von LSPs

- Beobachtung 1: Falls $l(P(x \rightarrow y))$ und $r(P(x \rightarrow y))$ SP sind, dann ist $P(x \rightarrow y)$ ein LSP
- Beobachtung 2: Bezeichnen \mathcal{SP} bzw. \mathcal{LSP} die Menge aller SPs bzw. LSPs in G . Dann ist $\mathcal{SP} \subseteq \mathcal{LSP}$.
- Lemma 2: Falls SP eindeutig in G sind, dann: Für alle Knotenpaare (x, y) gilt: die LSP-Wege von x nach y sind kontendisjunkt (außer Anfangs- und Endknoten x, y).
- Beweis Indirekt: Annahme: Es existieren 2 LSP Pfade P^1 und P^2 durch den gleichen Knoten $v \neq x, y$.
- Da jeder echte Teilweg von $P(x, y)$ ist SP, also auch $P^1(x \rightarrow v)$ und $P^2(x \rightarrow v)$
- wegen Eindeutigkeit: $P^1(x \rightarrow v) = P^2(x \rightarrow v)$.

Eigenschaften von LSPs

- Lemma 3: Falls SP eindeutig in G sind, dann kann es höchstens mn LSPs in G geben.
- Lemma 4: Sei G Graph, der eine Sequenz Σ von Knoten Updates durchlaufen hat. Falls SP eindeutig in G sind, dann können anlässlich eines „increase“ updates höchstens $O(n^2)$ Wege aus der LSP Menge herausfallen („stop being LSP“)
- Beweis: Ein Weg kann aus LSP herausfallen, nur wenn durch den Update einer seiner Teilwege nicht mehr in SP ist.
- Dies kann nur passieren, wenn dieser Teilweg den Knoten v enthält.
- Es gibt aber höchstens $O(n^2)$ LSPs, die v als internen Knoten enthalten und höchstens $O(n^2)$ LSPs mit v als Endknoten.

Eigenschaften von LSP's

- Theorem 1: Sei G Graph mit einer Sequenz Σ von increase-only update Operationen und sei m größte Kantenanzahl in G unter Σ . Falls SP eindeutig sind, dann ist die Anzahl der Pfade, die durch eine increase-Operation neue LSPs werden:
 - $O(mn)$ im Worst Case
 - $O(n^2)$ amortisiert über $\Omega(m/n)$ Update Operationen.
- Beweis: $O(mn)$ gilt, weil $O(mn)$ die maximale Anzahl in G ist.
- Amortisierte Analyse (Sketch): zu jedem Zeitpunkt können zwar theoretisch $O(mn)$ LSPs neu entstehen; andererseits können höchstens $O(n^2)$ LSPs herausfallen; insgesamt können zu jedem Zeitpunkt höchstens $O(mn)$ LSPs existieren. Nach m/n Operationen können also höchstens per Operation $O(mn/(m/n))=O(n^2)$ LSPs neu entstanden sein.

Algorithmus Idee für Increase-Only

- Halte alle LSPs $P(x \rightarrow y)$ in Datenstruktur $DS(x,y)$
- Wg. Theorem 1 hat man $O(n^2)$ Änderungen per Update in DS bei $\Omega(m/n)$ Operationen
- Als DS wähle Prioritätsschlange; Priorität= $\text{dist}()$
- $O(\log n)$ Kosten pro Weg-Änderung
- Insgesamt Laufzeit: $O(n^2 \log n)$

Speicherung der Wege:

- $P(x \rightarrow y)$ Zeiger zu $l(x,y)$ und $r(x,y)$ (konstant viel Speicherplatz)
- Das geht, wegen
 - Eindeutigkeit von SP und
 - optimaler Teilweg Eigenschaften

Datenstrukturen

- $\text{dist}[x,y]$: Matrix: Distanzen zwischen x und y
- $P(x,y)$: Menge aller LSPs von x nach y (in PrioQ)
- $P^*(x,y)$: Menge aller SPs von x nach y (in PrioQ)
- $L(P(x,y))$: Liste der möglichen $P(x,y)$ path extension Wege nach vorn (*left*), die LSP sind
- $L^*(P(x,y))$: Liste der möglichen $P(x,y)$ path extension Wege nach vorn (*left*), die SP sind
- $R(P(x,y))$: Liste der möglichen $P(x,y)$ path extension Wege nach hinten (*right*), die LSP sind
- $R^*(P(x,y))$: Liste der möglichen $P(x,y)$ path extension Wege nach hinten (*right*), die SP sind

Algorithmus Update()

Algorithmus Update(v, w'):

1. Cleanup(v): entfernt alle Wege, die v enthalten
2. Fixup(v, w'): Addiere alle neuen SP's und LSP's

Nach einer Operation Update(v, w') heißt ein SP (bzw. LSP) **NEU**, wenn er vorher nicht SP (bzw. LSP) war oder er den Knoten v enthält.

Prozedur Cleanup(v)

Prozedur Cleanup(v):

Entfernt alle Wege aus DS, die in $G \setminus \{v\}$ nicht mehr LSP wären

=== Entfernt alle Wege $P(x,y)$, die v enthalten aus

$P(x,y), P^*(x,y), L(r(P(x,y))), L^*(r(P(x,y))), R(l(P(x,y))), R^*(l(P(x,y)))$.

Cleanup(v) kann realisiert werden durch:

1. Entferne alle Wege der Form $P(u \rightarrow v)$ und $P(v \rightarrow u)$
2. Entferne rekursiv die dazugehörigen $L()$ und $R()$ -Wege

Prozedur Cleanup(v)

- $\text{Insert}Q \leftarrow \{(v)\}$
- Solange $Q \neq \emptyset$:
 - $P \leftarrow \text{ExtractElem}(Q)$
 - Für jeden Weg $P(x \rightarrow y) \in L(P) \cup R(P)$
 - $\text{Insert}Q \leftarrow P(x \rightarrow y)$
 - Entferne $P(x \rightarrow y)$ aus $P(x,y)$, $L(r(P(x,y)))$ und $R(l(P(x,y)))$
 - Falls $P(x \rightarrow y) \in P^*(x,y)$, dann
 - Entferne $P(x \rightarrow y)$ aus $P^*(x,y)$, $L^*(r(P(x,y)))$ und $R^*(l(P(x,y)))$.

Prozedur Fixup(v, w')

Phase 1:

Für alle $u \neq v$:

- $w(u,v) \leftarrow w'(u,v); w(v,u) \leftarrow w'(v,u)$
- Falls $w(u,v) < M$ dann:
 - setze $w((u,v)), l((u,v)) \leftarrow (u); r((u,v)) \leftarrow (v)$
 - addiere (u,v) zu $P(u,v), L((v)), R((u))$
- Falls $w(v,u) < M$ dann:
 - setze $w((v,u)), l((v,u)) \leftarrow (v); r((v,u)) \leftarrow (u)$
 - addiere (v,u) zu $P(v,u), L((u)), R((v))$

Alle LSP's der Länge 1 werden zu DS hinzuaddiert

Prozedur Fixup(v, w')

Phase 2:

$H \leftarrow \emptyset$

Für jedes Knotenpaar (x, y) :

- addiere den Weg $P(x \rightarrow y) \in P(x, y)$ mit den kleinsten Kosten zu H

Phase 3:

Solange $H \neq \emptyset$: ExtractMinH \leftarrow P(x→y)

Falls P(x→y) der erste extrahierte Weg für das Paar (x,y) ist:

Falls $P(x \rightarrow y) \notin P^*(x,y)$

– Addiere P(x→y) zu $P^*(x,y)$, $L^*(r(P(x,y)))$ und $R^*(l(P(x,y)))$

– Für jeden Weg $P(x' \rightarrow b) \in L^*(l(P(x \rightarrow y)))$ //kombiniere alle LSPs

- $P(x' \rightarrow y) \leftarrow (x',x)P(x \rightarrow y)$
- $w(P(x' \rightarrow y)) \leftarrow w(x',x) + w(P(x \rightarrow y))$
- $l(P(x' \rightarrow y)) \leftarrow P(x' \rightarrow b)$; $r(P(x' \rightarrow y)) \leftarrow P(x \rightarrow y)$;
- Addiere $P(x' \rightarrow y)$ zu $P(x',y)$, $L(P(x,y))$, $R(P(x',b))$ und H

– Für jeden Weg $P(a \rightarrow y') \in R^*(r(P(x \rightarrow y)))$

- $P(x \rightarrow y') \leftarrow P(x \rightarrow y)(y,y')$
- $w(P(x \rightarrow y')) \leftarrow w(P(x \rightarrow y)) + w(y,y')$
- $l(P(x \rightarrow y')) \leftarrow P(x \rightarrow y)$; $r(P(x \rightarrow y')) \leftarrow P(a \rightarrow y')$;
- Addiere $P(x \rightarrow y')$ zu $P(x,y')$, $L(P(a,y'))$, $R(P(x,y))$ und H

“

Analyse Korrektheit

- Invariante: Falls die kürzesten Wege eindeutig sind, dann ist für jedes Knotenpaar x und y in G der erste (x,y) -Weg, der aus H in Phase 3 von $\text{Fixup}()$ extrahiert wird ein kürzester Weg.
- Annahme: Invariante ist (erstes Mal) verletzt bei Pfad $P'(x \rightarrow y)$. Sei $P(x \rightarrow y)$ SP zwischen x und y .
- Offensichtlich $P(x \rightarrow y) \notin H$ und $\notin P(x,y)$ (sonst Phase 2 in H)
- Also ist $P(x \rightarrow y)$ ein neuer LSP mit Länge >1 (wg. Phase 1)
- $l(P(x \rightarrow y))$ und $r(P(x \rightarrow y))$ müssen neue kürzeste Wege sein; einer davon war zu Beginn von $\text{fixup}()$ nicht in P^*
- $l(P(x \rightarrow y))$ oder $r(P(x \rightarrow y))$ wurden vor der falschen Extraktion (s.o.) aus H extrahiert (weil geringere Kosten)
- dort wurde dann auch $P(x \rightarrow y)$ gebildet und zu H addiert

Analyse Korrektheit

- Falls die Operation ein increase war und die kürzeste Wege eindeutig sind, dann werden die Mengen $P(x,y)$ und $P^*(x,y)$ für jedes Knotenpaar x und y korrekt aktualisiert.
- Cleanup(): o.k.
- Wird jeder neue LSP gebildet? Länge 1: o.k. Länge > 1:
- $P(x \rightarrow y)$ muss neu gebildet werden, wenn beide, $l(P(x \rightarrow y))$ und $r(P(x \rightarrow y))$ SP sind aber mind. einer davon nicht im alten P^* war.
- Irgendwann wird (o.E.) $l(P(x \rightarrow y))$ extrahiert; dann wird $l(P(x \rightarrow y))$ kombiniert mit $r(P(x \rightarrow y)) \in R^*(r(l(P(x \rightarrow y))))$ um $P(x \rightarrow y)$ zu bilden und zu H zu addieren.

Analyse Laufzeit

- In einer Increase-only Sequenz von $\Omega(m/n)$ Operationen wird jede Update Operation in $O(n^2 \log n)$ amortisierter Zeit, und jede Distanz und Weg-Anfrage in optimaler Zeit durchgeführt.
- Da $P(x,y)$ in Priority Queue gehalten wird: Abfrage: $O(1)$
- Cleanup(v): maximal $O(n^2)$ LSPs können durch v laufen; pro Iteration $O(\log n)$: $O(n^2 \log n)$
- Fixup():
 - Phase 1: $O(n \log n)$
 - Phase 2: $O(n^2 \log n)$
 - Phase 3: $O(n^2 \log n)$

Ende der VO

