

Kapitel 6: Dynamic Shortest Path

6.1 Einführung

6.2 SSSP von Ramalingam & Reps

VO Algorithm Engineering

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

12./13. VO

15./22. Mai 2007

Organisatorisches zur Übung

wichtig:

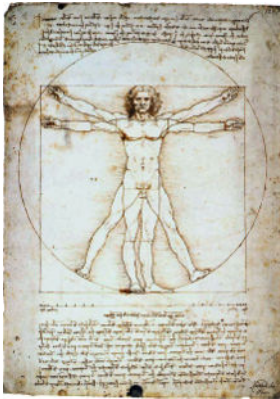
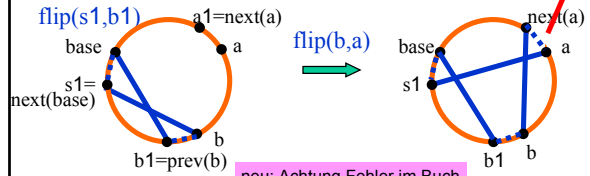
- **Dienstags**-Übung vom 29. Mai wird verschoben auf 5. Juni
- **Donnerstags**-Übung: Einteilung der Aufgabenbearbeitung für Übungsblatt 3:
 - Aufgabe 1: Gruppe 16
 - Aufgabe 2: Gruppen 12 und 14
 - Aufgabe 3: Gruppen 9, 11 und 15

Nachlese Oberwolfach Seminar on Algorithm Engineering

- TSP-Buch hier!
- Achtung: falsche Flipfolge für Lin-Kernighan im Buch, Fig. 15.5 passt nicht mit Flipfolge
- korrekt (Bill Cook):
 - Fall 1.1: $\text{flip}(a1, \text{base}), \text{flip}(\text{base}, b), \text{flip}(b, s1)$,
 - Fall 1.2: $\text{flip}(s1, b1), \text{flip}(b, a)$

Dritte flip()-Sequenz: 3er Austausch

- Wähle einen Nachbarn b von $\text{next}(a)$ entfernt $(a, \text{next}(a))$
- **1. Fall:** b liegt zwischen $\text{next}(\text{base})$ und a :
 - Betrachte 2 Alternativen für $\text{flip}()$ -Sequenzen:
 - **1.1:** $\langle \text{flip}(a1, \text{base}), \text{flip}(\text{base}, b), \text{flip}(b, s1) \rangle \rightarrow$ entfernt zusätzlich $(b, \text{next}(b))$
 - **1.2:** $\langle \text{flip}(s1, b1), \text{flip}(b, a) \rangle$, wobei $b1 = \text{prev}(b)$ zu Beginn \rightarrow entfernt zusätzlich $(\text{prev}(b), b)$



Pino Italiano (Rom):

- A Theoretician knows: Theory and Practice are quite the same!
- A Practitioner knows: Theory and Practice are quite different!

Leonardo da Vinci: [Der vitruvianische Mensch](#), 1492

Nachlese Oberwolfach Seminar on Algorithm Engineering

Rajeev Raman (London):

- In my theoretical papers I have always written: „it is easy to do this and that...“. Now I have tried to implement it, and it was not clear how to do this at all.“

- Jetzt HIER: 3. Übungsblatt über Suffix Arrays
- Aufgabenzuordnung für die Donnerstag-Gruppe: s. Web
- für Dienstags-Gruppe: heute

Überblick

5.1 Einführung

- Dynamische Graphalgorithmen
- Dynamische Komplexitätsmaße
- Dijkstra's SSSP-Algorithmus
- Analyse: Korrektheit und Laufzeit

5.2 SSSP von Ramalingam & Reps

- Einführung SP-Teilgraph für SSSP
- Algorithmus Dynamic SSSP
- Laufzeit-Analyse

7

Literatur für diese VO

- G. Ramalingam und T. Reps: On the computational complexity of dynamic graph problems. Theoretical Computer Science 158, 1996, 233-277

8

Dynamische Graphalgorithmen

Ein **dynamischer Algorithmus** erhält eine gegebene Eigenschaft P eines gewichteten Graphen während **dynamischer Änderungen des Graphen**, z.B.

- Einfügen neuer Kanten,
- Entfernen von Kanten und
- Kosten-Änderungen der Kanten

Anforderungen an einen dynamischen Algorithmus:

- **schnelle Beantwortung von Anfragen** auf Eigenschaft P
- **schnelle** Bearbeitung von **Update-Operationen**, d.h. schneller als ein statischer Algorithmus, der jedesmal alles von vorn berechnen muss

9

Dynamische Graphalgorithmen

- 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)**.

10

Dynamische Komplexitätsmaße

- **Amortisiert:** Worst-Case in Größe des Inputs, wobei der Durchschnitt über eine Folge von Operationen genommen wird.
- **Alternativ:** Worst-Case in Größe der Änderungen bzgl. des Inputs und des Outputs

11

Worst Case bzgl. der I/O-Änderungen

- Ein Knoten heißt **modifiziert (modified)**, wenn er oder eine inzidente Kante einen neuen Input-Wert bekommen hat oder eingefügt oder entfernt wurde.
- Ein Knoten heißt **betroffen (affected)**, wenn er entweder neu eingefügt wurde oder er durch die Änderung einen neuen Output-Wert bekommen hat.
- $\text{CHANGED} := \{\text{Menge aller modifizierten oder betroffenen Knoten}\}$
- Sei $|\delta| = |\text{CHANGED}|$
- Sei $\|\delta\| = |\delta| + \text{Anzahl der zu CHANGED inzidenten Kanten}$
- So viele Änderungen sind mindestens notwendig
- \rightarrow notwendige Laufzeit

12

Worst Case bzgl. der I/O-Änderungen

- Ein dynamischer Algorithmus heißt **beschränkt (bounded)**, wenn die benötigte Zeit für einen Update Schritt durch eine Funktion in $||\delta||$ beschränkt ist.
- Sonst heißt er **unbeschränkt (unbounded)**.
- Ein dynamisches Problem heißt **unbeschränkt (unbounded)**, wenn kein dynamischer beschränkter Algorithmus existiert.

- **HIER: ein beschränkter dynamischer SSSP Algorithmus**

13

Kürzeste Wege Probleme

Single-Source Shortest Path (SSSP)

- **Geg.:** Graph $G=(V,E)$ mit Kantenkosten $w \in \mathbb{R}$, keine negativen Kreise, $s \in V$
- **Gesucht:** Kürzeste (s,v) -Wege in G für alle $v \in V$

All-Pairs Shortest Path (APSP)

- **Geg.:** Graph $G=(V,E)$ mit Kantenkosten $w \in \mathbb{R}$, keine negativen Kreise
- **Gesucht:** Kürzeste Wege zwischen allen Knotenpaaren (Distanzmatrix)

4

Dijkstra's Algorithmus

- Graph G gerichtet mit Kantenkosten $w \geq 0$
- Sei $M =$ „große Zahl“
- Sei $dist(v)$ die Distanzmatrix
- Q sei Prioritätswarteschlange mit Operationen
 - $InsertPrioQ(v, dist(v))$: fügt v ein mit Priorität $dist(v)$
 - $ExtractMinQ()$: Gibt das Minimum zurück und entfernt es aus Q
 - $DecreasePrioQ(v, dist(v))$: Aktualisiert die Priorität von v in Q auf $dist(v)$

15

Dijkstra's Algorithmus

- $Q \leftarrow \emptyset$; $dist[s]=0$; $InsertPrioQ(s,0)$ // $S \leftarrow \{s\}$
- **Für** alle Knoten $x \neq s$:
 - Setze $dist[x]=M$; $InsertPrioQ(x,M)$
- **Solange** $Q \neq \emptyset$:
 - $x \leftarrow ExtractMinQ()$; // Addiere x zu S ;
 - **Für** alle Kanten (x,y) , die x verlassen: ★
 - **Falls** $dist[y] > dist[x]+w[x,y]$:
 - Setze $dist[y] \leftarrow dist[x]+w[x,y]$;
 - $DecreasePrioQ(y, dist[y])$
- **Return** $dist[]$

★ edge relaxation. edge scanning ..

Analyse: Korrektheit

- **Induktion:** Nach jeder Iteration ist V in 2 Mengen aufgeteilt: S und $T := V \setminus S$
- **Ind.-Ann.:**
 - (1) Für $x \in S$ ist $dist(x)$ =Länge des kürzesten (s,x) -Weges
 - (2) für $x \in T$ ist $dist(x)$ =Länge des kürzesten (s,x) -Weges bei dem jeder Knoten außer x zu S gehört.
- **Ind.-Schritt:**
 - Knoten x mit kleinstem $dist$ -Wert wird in S aufgenommen. Ist korrekt, denn: Falls ein kürzerer Weg existieren würde, dann würde dieser einen ersten Knoten in T benutzen; aber dieser müßte weiter weg von s sein, denn sein $dist$ -Wert ist größer als der von x . Die „edge relaxation“ sorgt dafür, dass (2) erfüllt ist.

17

Analyse: Laufzeit

- Sei $n=|V|$ und $m=|E|$
- $T = n * T(InsertPrioQ()) + n * T(ExtractMinQ()) + m * T(DecreasePrioQ())$
- **Binärer Heap** für Q : $O((n+m) \log n)$
 - $T(InsertPrioQ()) = O(\log n)$
 - $T(ExtractMinQ()) = O(\log n)$
 - $T(DecreasePrioQ()) = O(\log n)$
- **Fibonacci Heap** für Q : $O(m+n \log n)$
 - $T(InsertPrioQ()) = O(1)$
 - $T(ExtractMinQ()) = O(\log n)$ amortisiert
 - $T(DecreasePrioQ()) = O(1)$ amortisiert

18

5.2 Algorithmus von Ramalingam & Reps

Betrachte folgendes kürzestes Wegeproblem:

Single-Source Shortest Path (SSSP)

- **Geg.:** Graph $G=(V,E)$ mit Kantenkosten $c > 0$, keine negativen Kreise, $t \in V$
- **Gesucht:** Kürzeste (v,t) -Wege in G für alle $v \in V$

19

5.2 Algorithmus von Ramalingam & Reps

Geg.: Gerichteter Graph $G=(V,E)$ mit **positiven** Kantenkosten $c(e) \in \mathbb{R}$, $t \in V$, sowie Sequenz der folgenden Operationen:

- **delete_edge(v,w):** entferne die Kante (v,w) aus G
- **insert_edge(v,w,c):** füge die Kante (v,w) in G ein
- **dist(v):** gib die Distanz zwischen Knoten v und t zurück
- **path(v):** gib den kürzesten Weg von v nach t aus, falls einer existiert.

Damit sind auch die folgenden Operationen simulierbar:

- **increase_weight(e,c'):** erhöhe die Kosten von Kante e auf c'
- **decrease_weight(e,c'):** reduziere Kosten der Kante e auf c' (durch delete_edge und insert_edge)

20

Definitionen

- Ein Teilgraph T von G heißt **kürzester-Wege-Baum** für G mit Senke t , falls
 - T ist ein gerichteter Baum mit Wurzel t
 - $V(T)$ ist die Menge aller Knoten, die t erreichen können, und
 - für jede Kante in T gilt: $\text{dist}(u) = \text{dist}(v) + c(u,v)$

21

Definitionen

- Eine Kante heißt **SP-Kante**, wenn sie auf einem kürzesten Weg von v nach t liegt für ein $v \in V$.
- Eine Kante ist also genau dann SP-Kante, wenn gilt: $\text{dist}(u) = \text{dist}(v) + c(u,v)$

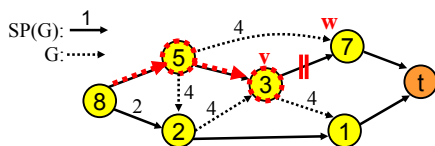
- Sei **SP(G)** der durch die Menge aller SP-Kanten induzierte Teilgraph von G (der kürzeste-Wege Teilgraph).
- Jeder kürzeste Weg in G ist in **SP(G)** enthalten und umgekehrt:
- Jeder Weg in **SP(G)** ist ein kürzester Weg in G .

- Da die Kantengewichte alle positiv sind, ist **SP(G)** ein gerichteter, azyklischer Graph.

22

Delete_edge(v,w): $G \rightarrow G'$

- Ein Knoten in G' heißt **betroffen**, wenn der Wert $\text{dist}_{G'}(v) \neq \text{dist}_G(v)$ ist.
- Eine SP-Kante (x,y) heißt **betroffen** durch die delete_edge(v,w) Operation, wenn kein Pfad in G' von x nach t existiert, der die Kante (x,y) benutzt und Länge $\text{dist}_{\text{alt}}(x)$ besitzt.



23

Delete_edge(v,w): $G \rightarrow G'$

- Ein Knoten in G' heißt **betroffen**, wenn der Wert $\text{dist}_{G'}(v) \neq \text{dist}_G(v)$ ist.
- Eine SP-Kante (x,y) heißt **betroffen** durch die delete_edge(v,w) Operation, wenn kein Pfad in G' von x nach t existiert, der die Kante (x,y) benutzt und Länge $\text{dist}_{\text{alt}}(x)$ besitzt.

Beobachtungen in SP(G)

- (x,y) ist betroffen $\Leftrightarrow y$ ist betroffen
- Knoten $x \neq v$ ist betroffen \Leftrightarrow alle ausgehenden SP-Kanten von x sind betroffen
- Knoten v selbst ist betroffen $\Leftrightarrow (v,w)$ die einzige ausgehende SP-Kante ist

24

Algorithmus Delete_edge(v,w)

Phase 1:

- Bestimme die Menge aller betroffenen Knoten und Kanten
- Entferne die betroffenen Kanten von SP(G)

Phase 2:

- Berechne neue dist-Werte aller betroffenen Knoten
- Aktualisiere SP(G)

25

Phase 1 von Delete_edge(v,w)

- Menge der betroffenen Knoten = Menge der Knoten, die im Graphen SP(G)-(v,w) keine Verbindung zur Senke t besitzen.

Algorithmus-Idee:

- Aufruf von Top-Sort, wenn outdegree von v in SP(G') = 0.
- Speichere SP(G) durch Adjazenzlisten: in(v)+out(v), oder
- nicht explizite Speicherung, sondern durch Test:
 $\text{dist}(x) = \text{dist}(y) + c(x,y)$

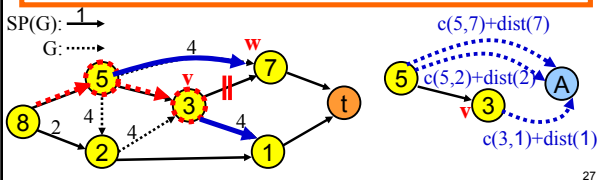
26

Phase 2 von Delete_edge(v,w)

- Sei A die Menge der nicht-betroffenen Knoten
- Ziel: Berechne $\text{dist}(x)$ für alle Knoten in $B = V \setminus A$

Algorithmus-Idee:

- Kontrahiere A zu einem Knoten t' und ersetze die Kanten (x,y) durch (x,t') mit Kosten $c(x,y) + \text{dist}(y)$
- Aufruf von Dijkstra für den neuen Graphen



27

Algorithmus Delete_edge(v,w)

1. **Falls** $(v,w) \in \text{SP}(G)$ **dann:**
2. Entferne (v,w) aus SP(G) und E; dekrementiere $\text{outdegSP}(v)$
3. **Falls** $\text{outdegSP}(v) == 0$ **dann:** // Phase 1
4. $\text{WorkSet} = \{v\}$; $\text{AffectVert} = \emptyset$
5. **Solange** $\text{WorkSet} \neq \emptyset$ **do:**
6. Wähle und entferne Knoten u aus WorkSet
7. $\text{AffectVert} = \text{AffectVert} \cup \{u\}$
8. **Für jeden** Knoten x mit $(x,u) \in \text{SP}(G)$:
9. Entferne (x,u) aus SP(G); dekrement. $\text{outdegSP}(x)$
10. **Falls** $\text{outdegSP}(x) == 0$ **dann:**
11. $\text{WorkSet} = \text{WorkSet} \cup \{x\}$

28

Algorithmus Delete_edge(v,w) ff

12. $Q = \emptyset$ // Phase 2
13. **Für jeden** Knoten $a \in \text{AffectVert}$ **do:**
14. $\text{dist}(a) = \min(\{c(a,b) + \text{dist}(b) \mid (a,b) \in E \text{ und } b \notin \text{AffectVert}\})$ UM
15. **Falls** $\text{dist}(a) \neq M$ **dann:**
16. $\text{InsertQ}(a, \text{dist}(a))$
17. **Solange** $Q \neq \emptyset$ **do:**
18. $a \leftarrow \text{ExtractMin}(Q)$
19. **Für jeden** Knoten $b \in \text{Succ}(a)$ mit $c(a,b) + \text{dist}(b) == \text{dist}(a)$
20. Füge (a,b) in SP(G) ein; inkrementiere $\text{outdegSP}(a)$
21. **Für jeden** Knoten $b \in \text{Pred}(a)$ mit $c(b,a) + \text{dist}(a) < \text{dist}(b)$
22. $\text{dist}(b) = c(b,a) + \text{dist}(a)$
23. AktualisiereHeapQ(b, dist(b))

29

Laufzeitanalyse

- Phase 1:
 - Anzahl der Iterationen: $|\delta|$
 - Eine Iteration für Knoten u benötigt $O(|\text{Pred}(u)|)$
 - Gesamt: $O(|\delta|)$
 - Phase 2: mit Fibonacci Heaps
 - Insert + Decrease_key: $O(1)$ amortisiert
 - ExtractMin $O(\log p)$ mit $p = \text{Anzahl der Elemente im Heap}$
 - Anzahl der Iterationen: höchstens $|\delta|$
 - Gesamt: $O(|\delta| + |\delta| \log |\delta|)$
- Laufzeit: $O(|\delta| + |\delta| \log |\delta|)$

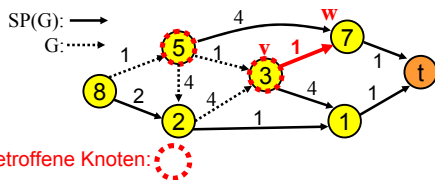
30

Insert_edge(v,w): $G \rightarrow G'$

Beobachtung:

- Wenn u betroffen ist, dann sehen alle kürzesten Wege in G' von u nach t folgendermaßen aus:
- (kürzester (u,v)-Weg, Kante (v,w), kürzester (w,s)-Weg)

u ist betroffen $\Leftrightarrow \text{dist}_G(u,v) + c(v,w) + \text{dist}_G(w) < \text{dist}_G(u)$

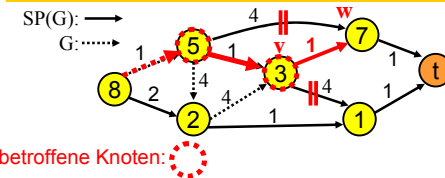


31

Idee: Benutze Dijkstra's Algorithmus für Menge der betroffenen Knoten mit Key: $\text{dist}(x) - \text{dist}(v)$ für alle x

Betrachte kürzesten-Wege Baum T_v für Knoten v (Wurzel):

- Sei x beliebiger Knoten, u: Elter von x in T_v
- Wenn x betroffen ist, dann muß auch u betroffen sein, denn sonst: ex. kürzester Weg P von u nach t ohne Kante (v,w); dann ist Weg ((x,u),P) auch kürzester Weg; x wäre nicht betroffen.
- \Rightarrow Menge der betroffenen Knoten bilden zusammenhängenden Teilbaum von T_v mit Wurzel v



32

Algorithmus Insert_edge((v,w),c)

1. Insert edge (v,w) in G, setze Kosten $c(v,w)$
2. $Q = \emptyset$
3. **Falls** $c(v,w) + \text{dist}(w) < \text{dist}(v)$ **dann:**
4. $\text{dist}(v) = c(v,w) + \text{dist}(w)$
5. InsertHeapQ(v,0)
6. **Sonst Falls** $c(v,w) + \text{dist}(w) == \text{dist}(v)$ **dann:**
7. Füge (v,w) in SP(G) ein
8. $\text{outdeg}(v)++$

33

Algorithmus Insert_edge((v,w),c) ff

9. **Solange** $Q \neq \emptyset$:
10. $u \leftarrow \text{ExtractMin}(Q)$
11. Entferne alle ausgehenden Kanten von u in SP(G) und update $\text{outdeg}(u)$
12. **Für jeden** Knoten $x \in \text{Succ}(u)$:
13. **Falls** $c(u,x) + \text{dist}(x) == \text{dist}(u)$ **dann:**
14. Füge (u,x) in SP(G) ein und aktualisiere $\text{outdeg}(u)$
15. **Für jeden** Knoten $x \in \text{Pred}(u)$:
16. **Falls** $c(x,u) + \text{dist}(u) < \text{dist}(x)$ **dann:**
17. $\text{dist}(x) = c(x,u) + \text{dist}(u)$
18. Aktualisiere/InsertHeapQ(x, $\text{dist}(x) - \text{dist}(v)$)
19. **Sonst Falls** $c(x,u) + \text{dist}(u) == \text{dist}(x)$
20. Füge (x,u) in SP(G) ein und aktualisiere $\text{outdeg}(x)$

34

Laufzeitanalyse

- **mit Fibonacci Heaps**
 - Insert + Decrease_key: $O(1)$ amortisiert
 - ExtractMin: $O(\log p)$ mit $p = \text{Anzahl der Elemente im Heap}$
 - Anzahl der Iterationen: höchstens $|\delta|$
 - Gesamt: $O(|\delta| + |\delta| \log |\delta|)$

– Laufzeit: $O(|\delta| + |\delta| \log |\delta|)$

35

Zusammenfassung

- **Theorem:** Es gibt einen **beschränkten** dynamischen Algorithmus für das SSSP-Problem mit Laufzeit $O(|\delta| + |\delta| \log |\delta|)$ für jede Update-Operation $\text{delete_edge}(v,w)$ und $\text{insert_edge}(v,w,c)$.

36