

Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 20 (23.7.2014)

All-Pairs Shortest Paths,
String Matching (Textsuche)

Fabian Kuhn

Algorithmen und Komplexität



**UNI
FREIBURG**

- Sie sollten alle eine E-Mail mit den Details zur Evaluation der Vorlesung erhalten haben.
- Bitte die Evaluation bis Ende Woche ausfüllen
- Am interessantesten sind für uns die allgemeinen Kommentare
- Auf dem letzten Übungsblatt bekommen Sie 15 Punkte für die Abgabe der Evaluation
 - falls Sie bei der Abgabe in der Datei erfahrungen.txt angeben, dass Sie die Evaluation gemacht haben
 - sollte auch helfen, die Prüfungszulassung zu bekommen, wenn's mit den 50% der Punkte sonst knapp wird...
- Ich werde die Resultate nächste Woche in der Vorlesung besprechen

Kürzeste Wege zw. allen Knotenpaaren

- all pairs shortest paths problem

Berechne single-source shortest paths für alle Knoten

- Dijkstras Algorithmus mit allen Knoten:

Laufzeit: $n \cdot O(\text{Laufzeit Dijkstra}) \in O(\underline{mn + n^2 \log n})$

- Problem: funktioniert nur bei nichtnegativen Kantengewichten

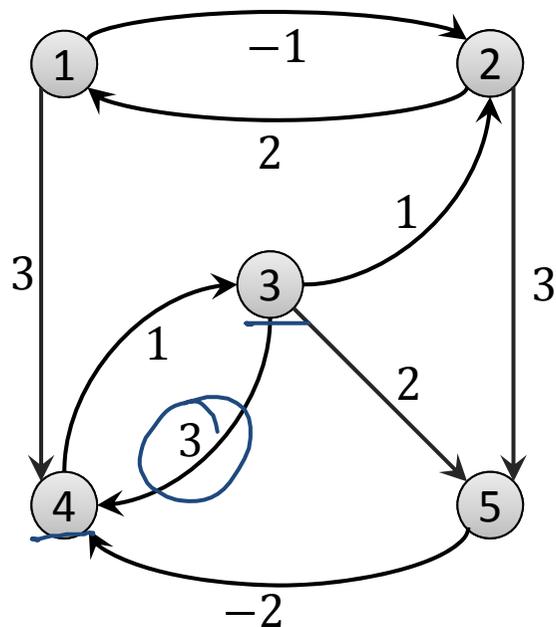
- Bellman-Ford Algorithmus mit allen Knoten:

Laufzeit: $\underline{n} \cdot O(\underline{\text{Laufzeit BF}}) \in O(mn^2) \in \underline{O(n^4)}$

- Problem: langsam...

Distanzen zw. allen Knotenpaaren

- Wir beschränken uns zur Einfachheit auf Distanzen
- Anstatt mit Adjazenzlisten werden wir dieses Mal mit der Adjazenzmatrix arbeiten
- Oder etwas genauer, mit einer Distanzmatrix
- **Initialisierung:**



$$W = \underline{L_1} = \begin{pmatrix} 0 & -1 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty & 5 \\ \infty & 1 & 0 & \underline{3} & 2 \\ \infty & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & -2 & 0 \end{pmatrix}$$

Distanzmatrix

L_{n-1} : Distanzmatrix
(falls kein neg. kreise)

Pfade aus $\leq t$ Kanten?

- Matrix L_t : Distanzen, falls nur Pfade aus $\leq t$ Kanten benutzt werden dürfen

Rekursive Berechnung:

$$L_t(i, j) = \min_{1 \leq k \leq n} \{ L_{t-1}(i, k) + L_1(k, j) \}$$

Distanzmatrix und Matrixmultiplikation

Berechnung von L_t (aus L_{t-1} und W):

$$L_t(i,j) = \min_{1 \leq k \leq n} \{ L_{t-1}(i,k) + L_1(k,j) \}$$

Matrixmultiplikation von L_{t-1} und W : $L' = L_{t-1} \cdot W$

$$L'(i,j) = \sum_{k=1}^n L_{t-1}(i,k) \cdot L_1(k,j)$$

Definition: $L_t = L_{t-1} \odot W$

- Matrixmultiplikation in der sogenannten Min-Plus-Algebra

Distanzmatrix berechnen

Algorithmus zum Berechnen der Distanzmatrix

```
 $L_1 := W$   
for  $t := 2$  to  $n - 1$  do  
     $L_t := L_{t-1} \odot W$ 
```

oder ausgeschrieben...

```
 $L_1 := W$   
for  $t := 2$  to  $n - 1$  do  
    for  $i := 1$  to  $n$  do  
        for  $j := 1$  to  $n$  do  
             $L_t(i, j) := L_{t-1}(i, j)$   
            for  $k := 1$  to  $n$  do  
                if  $L_{t-1}(i, k) + W(k, j) < L_t(i, j)$  then  
                     $L_t(i, k) := L_{t-1}(i, k) + W(k, j)$ 
```

Laufzeit $O(n^4)$

$L_t(i, j) := \min_k \{ \dots + \dots \}$

Distanzmatrix schneller berechnen

Algorithmus zum Berechnen der Distanzmatrix

$L := W$ $L_1, L_2, L_4, L_8, \dots$

for $t := 1$ **to** $\lceil \log_2 n \rceil$ **do**

$L' := L \odot L$

$L := L'$ $O(n^3)$

- Am Schluss gilt $L = L_{2^{\lceil \log_2 n \rceil}} = L_{n-1}$ $t \geq n-1 \quad L_t = L_{n-1}$

- Laufzeit: $O(n^3 \cdot \log n)$

Distanzmatrix schneller berechnen

Algorithmus zum Berechnen der Distanzmatrix

$L := W$

for $t := 1$ **to** $\lceil \log_2 n \rceil$ **do**

$L' := L \odot L$

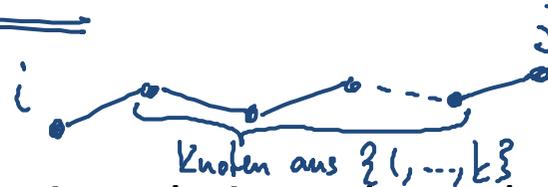
$L := L'$

- Alg. ist ein klassisches Beispiel für dynamische Programmierung
 - Opt. Pfad der Länge $\leq t$ besteht aus optimalen Pfaden der Länge $\leq t/2$
 - L_t wird rekursiv berechnet
 - Zwischenresultate werden wiederverwendet

Distanzen noch schneller berechnen?

- Ein anderer Ansatz, wieder durch dynamische Programmierung...
- Annahme: Knoten sind von 1 bis n nummeriert

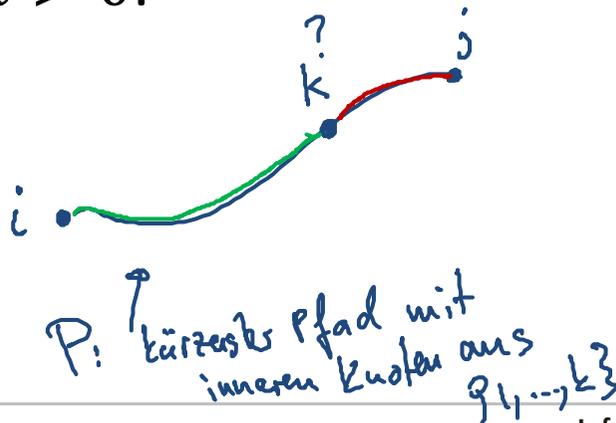
Definition $d_k(i, j)$



- Länge des kürzesten Pfades von i nach j , so dass als innere Knoten nur die Knoten $1, \dots, k$ verwendet werden.

Rekursive Definition von $d_k(i, j)$

- $d_0(i, i) = 0$, $d_0(i, j) = w(i, j)$ falls $(i, j) \in E$, $d_0(i, j) = \infty$ sonst
- Für $k > 0$:



Fall 1: k ist nicht innerer Knoten von P

$$d_k(i, j) = d_{k-1}(i, j)$$

Fall 2: k ist innerer Knoten von P

$$d_k(i, j) = d_{k-1}(i, k) + d_{k-1}(k, j)$$

Distanzen noch schneller berechnen?

Definition $d_k(i, j)$

- Länge des kürzesten Pfades von i nach j , so dass als innere Knoten nur die Knoten $1, \dots, k$ verwendet werden.

Rekursive Definition von $d_k(i, j)$

- $d_0(i, i) = 0$, $d_0(i, j) = w(i, j)$ falls $(i, j) \in E$, $d_0(i, j) = \infty$ sonst
- Für $k > 0$:

$$\underline{d_k(i, j)} := \min\{\underline{d_{k-1}(i, j)}, \underline{d_{k-1}(i, k) + d_{k-1}(k, j)}\}$$

falls k nicht
innerer Knoten
von P ist

falls k innerer
Knoten von P ist

Floyd-Warshall Algorithmus

// Initialization

```
for  $i := 1$  to  $n$  do
  for  $j := 1$  to  $n$  do
    if  $(i, j) \in E$  then  $d_0(i, j) := \underline{w(i, j)}$  else  $d_0(i, j) := \underline{\infty}$ 
     $\underline{d_0(i, i) := 0}$ 
```

// Main Loop

```
for  $\underline{k := 1}$  to  $n$  do
  for  $\underline{i := 1}$  to  $n$  do
    for  $\underline{j := 1}$  to  $n$  do
       $\underline{d_k(i, j) := \min\{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\}}$ 
```

- Korrektheit folgt, da $\underline{d_G(i, j)} = \underline{d_n(i, j)}$
- Laufzeit: $\underline{O(n^3)}$

Schnellere Algorithmen

bestes Matrix-Mult.-Alg.
zur Zeit hat Laufzeit $O(n^{2.38})$

- Es gibt schnellere Algorithmen zum Multiplizieren von Matrizen
 - Die Techniken können zum Teil verwendet werden
 - Allerdings nicht direkt
 - Und insbesondere bei gewichteten Graphen nicht ohne zusätzliche Kosten

Dünn besetzte Graphen

$$m \in o(n^2)$$

Dijkstra: $O(m + n \log n)$

- Johnsons Algorithmus (Laufzeit $O(n^2 \log n + mn)$)
- Idee: Falls die Kantengewichte nichtnegativ sind, kann man einfach n Mal Dijkstra ausführen
- Falls der Graph negative Kantengewichte hat, werden zuerst nichtnegative Gewichte berechnet, welche die gleichen kürzesten Pfade ergeben
- Das kann man in Zeit $O(mn)$ tun

- Gegeben ein gerichteter Graph $G = (V, E)$
- Die transitive Hülle von G ist ein Graph $H = (V, E')$ für welchen gilt:
$$(u, v) \in E' \Leftrightarrow \text{gerichteter Pfad von } u \text{ nach } v \text{ in } G$$
- Kann genauso mit Floyd-Warshall in Zeit $O(n^3)$ berechnet werden
 - Man kann etwas optimieren (da man nur eine 1/0-Antwort benötigt)
- Hier kann man die Matrix-Multiplikationstechniken anwenden
 - Beste bekannte Komplexität: $O(n^{2.38})$

Textsuche / String Matching

Gegeben:

- Zwei Zeichenketten (Strings) $m \ll n$
- Text T (typischerweise lang) Länge: n
- Muster P (engl. pattern, typischerweise kurz) Länge: m

Ziel:

- Finde alle Vorkommen von P in T

Annahmen:

- Länge Text T : n , Länge Muster P : m

Beispiel:

Text T : DUBI DUBAD DUBI DUBADU

Pattern P : DUBI

- Ist offensichtlich wichtig...
- Wird in jedem Texteditor gebraucht
 - jeder Editor hat eine find-Funktion
- Wird von Programmiersprachen unterstützt:
 - Java: `String.indexOf(String pattern, int fromThisPosition)`
 - C++: `std::string.find(std::string str, size_t fromThisPosition)`
 - Python: `string.find(pattern, from)`

Naiver Algorithmus

- Gehe den Text von links nach rechts durch
- Das Muster kann an jeder der Stellen $s = 0, \dots, n - m$ vorkommen



- Prüfe an jeder dieser Stellen ob das Muster passt
 - indem das Muster Buchstabe für Buchstabe mit dem Text an der Stelle verglichen wird
 - Werden wir gleich noch etwas genauer anschauen...

Naiver Algorithmus

```
TestPosition(s):           // tests if  $T[s, \dots, s + m - 1] == P$   
     $t := 0$                   $t=0, \dots, m-1$   
    while  $t < m$  and  $T[s + t] = P[t]$  do  
         $t := t + 1$   
    return  $(t = m)$  and  $T[s + m - 1] = P[m - 1]$ 
```

Naiver Algorithmus

```
TestPosition(s):           // tests if  $T[s, \dots, s + m - 1] == P$   
     $t := 0$   
    while  $t < m$  and  $T[s + t] = P[t]$  do  
         $t := t + 1$   
    return  $(t = m)$  Geld...
```

String-Matching:

```
for  $s := 0$  to  $n - m$  do  
    if TestPosition(s) then  
        report found match at position s
```

Naiver Algorithmus Analyse

Laufzeit von TestPosition(s):

- Annahme: erster Mismatch ist an Position x des Patterns P
 - $x = m$ falls das Pattern gefunden wird

$$O(x+1)$$

Laufzeit des Algorithmus:

- Beispiel: $T = 000010001010001$ $P = 0001$

- Best case: $O(n)$ Bsp: $T = KLZYVPURSTSABQZBL$
 $P = AB$
- Worst case: $O(n \cdot m)$ Bsp: $T = AAA \dots A$
 $P = A \dots A$

Rabin-Karp Algorithmus

Grundidee



- Wir schieben wieder ein Fenster der Grösse m über den Text und schauen an jeder Stelle, ob das Muster passt
- Zur Einfachheit nehmen wir an, dass der Text nur aus den Ziffern 0, ..., 9 besteht
 - dann können wir das Muster und das Fenster als Zahl verstehen
- Wenn wir das Fenster eins nach rechts schieben, kann die neue Zahl einfach aus der alten berechnet werden

Text: 572839402137928
Pattern: 283

$$\begin{array}{l} 572 \downarrow -5 \cdot 100, \cdot 10, +8 \\ 728 \downarrow -7 \cdot 100, \cdot 10, +3 \\ 283 \downarrow -2 \cdot 100, \cdot 10, +9 \\ \vdots \end{array}$$

Beobachtungen:

- In jedem Schritt müssen wir einfach zwei Zahlen vergleichen
- Falls die Zahlen gleich sind, kommt das Muster an der Stelle vor
- Wenn man das Fenster um eins weiter schiebt, lässt sich die neue Zahl in $O(1)$ Zeit berechnen
- Falls wir zwei Zahlen in $O(1)$ vergleichen können, dann hat der Algorithmus Laufzeit $O(n)$
- **Problem:** Die Zahlen können sehr gross sein ($\Theta(m)$ bits)
 - Zwei $\Theta(m)$ -bit Zahlen vergleichen benötigt Laufzeit $\Theta(m)$
 - Nicht besser als mit dem naiven Algorithmus
- **Idee:** Benutze Hashing und vergleiche Hashwerte
 - Wenn man das Fenster eins weiter schiebt, sollte sich der neue Hashwert wieder in $O(1)$ Zeit aus dem alten Hashwert berechnen lassen

Rabin-Karp Algorithmus

Pattern $P = \overbrace{\quad\quad\quad}^{\text{m Ziffern}}$
 \Rightarrow Zahl x
 $\hookrightarrow x \bmod M$

Lösung von Rabin und Karp:

- Wir rechnen alles mit den Zahlen modulo M
 - M sollte möglichst gross sein, allerdings klein genug, damit die Zahlen $0, \dots, M - 1$ in einer Speicherzelle (z.B. 32 Bit) Platz haben
- Muster und Textfenster sind dann beides Zahlen aus dem Bereich $\{0, \dots, M - 1\}$
- Beim Schieben des Fensters um eine Stelle, lässt sich die neue Zahl wieder in $O(1)$ Zeit berechnen
 - Falls das nicht klar ist, siehe spätere Folie...
- Falls das Muster gefunden wird, sind die zwei Zahlen gleich, falls nicht, können sie trotzdem gleich sein
 - Falls die Zahlen gleich sind, dann überprüfen wir nochmals wie beim naiven Algorithmus Buchstabe für Buchstabe

Rabin-Karp Algorithmus: Beispiel

Text: 572830354826

Muster: 283

Modulus $M = 5$

$283 \bmod 5 = 3$
(M)

↳ sollte in der Praxis gross sein

$572 \bmod M = 2 \Rightarrow 2 \neq 3 \Rightarrow$ kein Match

$728 \bmod M = 3 \Rightarrow \underline{3=3} \Rightarrow$ Match möglich

↳ vergleiche 728 und 283
(Zifferweise) \Rightarrow kein Match } $O(m)$

$283 \bmod M = 3 \Rightarrow 3=3 \Rightarrow$ Match möglich

↳ vgl. 283 & 283
 \Rightarrow Match } $O(m)$

⋮
↓

Rabin-Karp Algorithmus: Pseudo-Code

Text $T[0 \dots n - 1]$, Muster $P[0 \dots m - 1]$, Basis b , Modulus M

$h := b^m \text{ mod } M$

$p := 0; t := 0;$

for $i := 0$ **to** $m - 1$ **do**

$p := (p \cdot b + P[i]) \text{ mod } M$

$t := (t \cdot b + T[i]) \text{ mod } M$

$s := 0;$

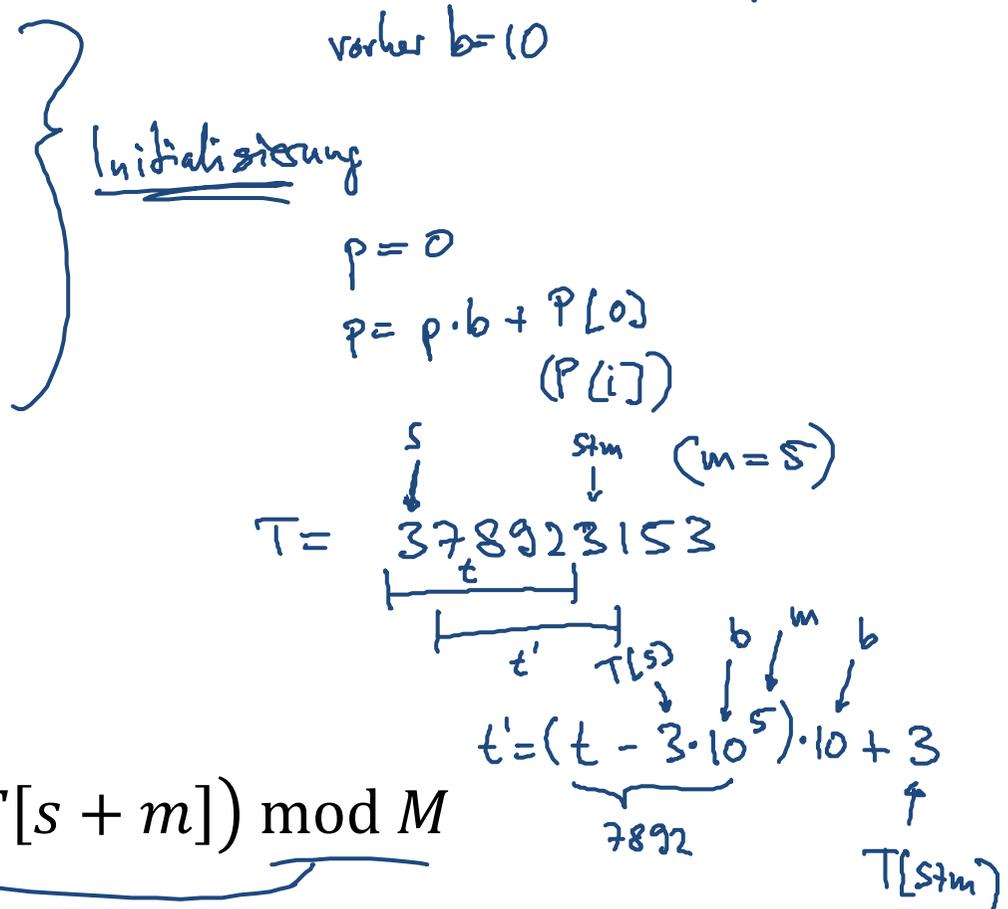
while $s \leq n - m$ **do**

if $p = t$ **then**

TestPosition(s)

$t := ((t - T[s] \cdot h) \cdot b + T[s + m]) \text{ mod } M$

Fenster verschieben



Rabin-Karp Algorithmus: Laufzeit

Vorberechnung: (Initialisierung) $O(m)$

Im schlechtesten Fall: $O(n \cdot m)$

$T = AAA \dots A$

$P = AAA.A$

- Der schlechteste Fall tritt ein, falls die Zahlen in jedem Schritt übereinstimmen. Dann muss man in jedem Schritt Buchstabe für Buchstabe überprüfen, ob man das Muster wirklich gefunden hat.
 - Sollte bei guter Wahl von M nicht allzu oft geschehen...
 - ausser, wenn das Muster tatsächlich sehr oft ($\Theta(n)$ mal) vorkommt...

Im besten Fall: $O(n)$

- Im besten Fall sind die Zahlen nur gleich, falls das Muster auch wirklich gefunden wird. Die Kosten sind dann $O(n + k \cdot m)$, falls das Muster im Text k Mal vorkommt.