Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 20 (23.7.2014)

All-Pairs Shortest Paths, String Matching (Textsuche)

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Vorlesungsevaluation

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

UNI FREIBURG

- all pairs shortest paths problem

Berechne single-source shortest paths für alle Knoten

- Dijkstras Algorithmus mit allen Knoten:

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

  – Problem: funktioniert nur bei nichtnegativen Kantengewichten

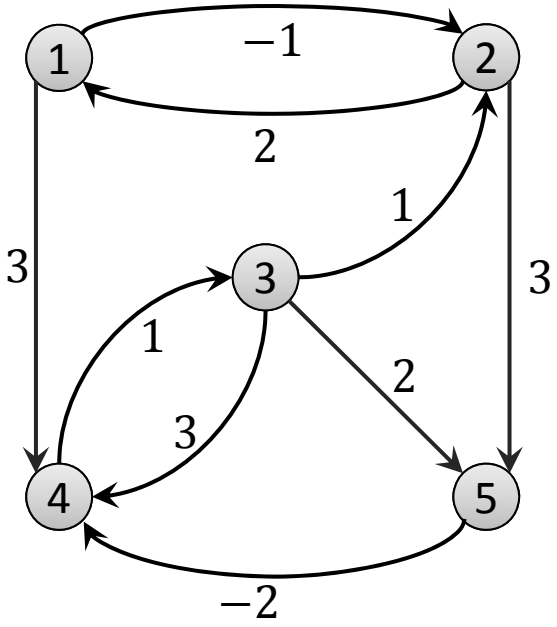
- Bellman-Ford Algorithmus mit allen Knoten:

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

  – 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:**



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

Distanzmatrix

UNI FREIBURG

Pfade aus $\leq t$ Kanten?

- Matrix L_t: Distanzen, falls nur Pfade aus ≤ 2 Kanten benutzt werden dürfen

Rekursive Berechnung:

Distanzmatrix und Matrixmultiplikation

UNI FREIBURG

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

Matrixmultiplikation von L_t und W:

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

- Matrixmultiplikation in der sogenannten Min-Plus-Algebra

Distanzmatrix berechnen

UNI FREIBURG

Algorithmus zum Berechnen der Distanzmatrix

$L_1 := W$

for $t := 2$ **to** $n-1$ **do**

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

oder ausgeschrieben...

$L_1 := W$

for $t := 2$ **to** $n-1$ **do**

$\quad$ **for** $i := 1$ **to** n **do**

$\quad\quad$ **for** $j := 1$ **to** n **do**

$\quad\quad\quad L_t(i,j) := L_{t-1}(i,j)$

$\quad\quad\quad$ **for** $k := 1$ **to** n **do**

$\quad\quad\quad\quad$ **if** $L_{t-1}(i,k) + W(k,j) < L_t(i,j)$ **then**

$\quad\quad\quad\quad\quad L_t(i,k) := L_{t-1}(i,k) + W(k,j)$

Distanzmatrix schneller berechnen

UNI FREIBURG

- Matrixmultiplikation in der Min-Plus-Algebra hat die gleichen Grundeigenschaften, wie die normale Matrixmultiplikation
- Insbesondere erfüllt sie das Distributivgesetz:
- Daher gilt $L_{x+y} = L_x \odot L_y$
- Und damit auch $L_{2t} = L_t \odot L_t$

Distanzmatrix schneller berechnen

UNI FREIBURG

Algorithmus zum Berechnen der Distanzmatrix

$L \coloneqq W$

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

$L' \coloneqq L \odot L$

$L \coloneqq L'$

- Am Schluss gilt $L = L_{2^{\lceil \log_2 n \rceil}} = L_n$

- Laufzeit:

Distanzmatrix schneller berechnen

UNI FREIBURG

Algorithmus zum Berechnen der Distanzmatrix

$L \coloneqq W$

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

$L' \coloneqq L \odot L$

$L \coloneqq L'$

- 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?

UNI FREIBURG

- 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 warden.

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 $k > 0$:

Distanzen noch schneller berechnen?

UNI FREIBURG

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 warden.

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 $k > 0$:

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

Floyd-Warshall Algorithmus

UNI FREIBURG

// Initialization

for $i \coloneqq 1$ to n do

    for $j \coloneqq 1$ to n do

        if $(i,j) \in E$ then $d_0(i,j) \coloneqq w(i,j)$ else $d_0(i,j) \coloneqq \infty$

    $d_0(i,i) \coloneqq 0$

// Main Loop

for $k \coloneqq 1$ to n do

    for $i \coloneqq 1$ to n do

        for $j \coloneqq 1$ to n do

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

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

Schnellere Algorithmen

UNI FREIBURG

- Es gibt schnellere Algorithm 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
  - 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

Transitive Hülle

- 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 warden
  - 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

UNI FREIBURG

Gegeben:

- Zwei Zeichenketten (Strings)
- Text T (typischerweise lang)
- Muster P (engl. pattern, typischerweise kurz)

Ziel:

- Finde alle Vorkommen von P in T

Annahmen:

- Länge Text T : $\boldsymbol{n}$, Länge Muster P : $\boldsymbol{m}$

Beispiel:

Motivation

UNI FREIBURG

- 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: strong.find(pattern, from)

Naiver Algorithmus

UNI FREIBURG

- 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 $T[s, \dots, s+m-1] == P$

$t := 0$

while $t < m$ **and** $T[s+t] = P[t]$ **do**

$t := t + 1$

return $(t = m)$

Naiver Algorithmus

UNI FREIBURG

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

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

Naiver Algorithmus Analyse

UNI FREIBURG

Laufzeit von `TestPosition(`s`)`:

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

Laufzeit des Algorithmus:

- **Beispiel:** $T = \texttt{000010001010001}$ $P = \texttt{0001}$

- **Best case:**
- **Worst case:**

Rabin-Karp Algorithmus

UNI FREIBURG

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 $0, \dots, 9$ 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

Rabin-Karp Algorithms

UNI FREIBURG

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

UNI FREIBURG

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 $M = 5$**

Rabin-Karp Algorithmus: Pseudo-Code

UNI FREIBURG

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

$h := b^{m-1} \bmod M$

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

for $i := 0$ **to** $m-1$ **do**
    $p := (p \cdot b + P[i]) \bmod M$
    $t := (t \cdot b + T[i]) \bmod M$

$s := 0;$

while $s \leq n-m$ **do**
    if $p = t$ **then**
        `TestPosition(`s`)`
    $t := \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$

Rabin-Karp Algorithmus: Laufzeit

UNI FREIBURG

Vorberechnung:

Im schlechtesten Fall:

- 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:

- 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.