Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 12a, Dienstag, 18. Juli 2017
(String Matching, Teil 1)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen ÜB11 — Edi-Tier
- Inhalt
  - String Matching — Definition + Beispiel
  - Naiver Algorithmus — Beispiel + Code
  - Knuth-Morris-Pratt Algorithmus — Beispiel + Code
  - Karp-Rabin Algorithmus — kommt morgen dran
  - ÜB12: Implementieren Sie den Karp-Rabin Algorithmus und benutzen Sie ihn zur automatischen **Plagiat**serkennung

Erfahrungen ÜB11 1/6

- Zusammenfassung / Auszüge
  - Am Montagnachmittag hatte noch kaum jemand etwas abgegeben und es gab genau eine erfahrungen.txt
  - Am Montagabend dann ein paar mehr Abgaben
  - Aufgabe 1 durch die ausführlichen Hinweise gut machbar

    Einige waren sich unsicher, ob Beweis ausreichend, weil man das Sieht-man-Doch Theorem nicht verwenden durfte

    Das ist ein Zeichen, dass man Beweise noch üben muss!
  - Bei Aufgabe 2 haben einige keinen guten Ansatz gefunden

    Bei vielen laut eigener Aussage auch aus Zeitmangel

Erfahrungen ÜB11 2/6

- Lösungsskizze Aufgabe 1
  - Erstmal betrachten wir alle Fälle von **zwei benachbarten** Operationen σ_1 und σ_2 in nicht monotoner Reihenfolge

    4 A 4 B nicht monoton
  - **Fall 1:** insert(i_1, c_1) , insert(i_2, c_2) mit $i_1 \geq i_2$

    4 B 5 A monoton

    Äquivalent: insert(i_2, c_2) , insert($i_1 + 1$, c_1) ... $\mathbf{i_2 < i_1 + 1}$

    Geht analog, wenn erste Operation replace oder delete ist

    5 4 nicht monoton
  - **Fall 2:** delete(i_1) , delete(i_2) mit $i_1 > i_2$

    4 4 monoton

    Äquivalent: delete(i_2) , delete($i_1 - 1$) ... $\mathbf{i_2 \leq i_1 - 1}$

    Hier braucht man, dass bei delete Positionsgleichheit erlaubt
  - So bekommt man für alle **neun** Kombination von insert / replace / delete eine äquivalente monotone Folge

Erfahrungen ÜB11   3/6

- Lösungsskizze Aufgabe 1, Fortsetzung
  - Betrachten wir jetzt eine optimale nicht monotone Folge **beliebiger** Länge: $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_k$
  - Wenn Sie nicht monoton ist, gibt es mindestens eine Stelle von benachbarten Operationen in "falscher" Reihenfolge
  - Die können wir umdrehen, wie auf der Folie vorher skizziert
  - Damit ist der Beweis aber noch nicht fertig

    Wenn wir so argumentieren wollen, müssen mir noch zeigen, dass dieser Prozess auch irgendwann aufhört

    Das ist nicht so einfach, deswegen argumentieren wir lieber etwas anders, siehe nächste Folie

Erfahrungen ÜB11 4/6

- Lösungsskizze Aufgabe 1, Fortfortsetzung
  - Betrachten wir jetzt eine optimale nicht monotone Folge **beliebiger** Länge: $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_k$
  - Betrachten wir die Operation mit der **kleinsten** Position i_{min}

    Bei mehreren inserts an Position i_{min} nehmen wir das rechteste, bei mehreren deletes mit Position i_{min} das linkeste

    Mehrere replaces mit i_{min} oder Mischung kann es nicht geben
  - Diese Operation können wir jetzt mit einer Folge von < k Nachbar-Vertauschungen and die erste Stelle bringen
  - Für die Folge $\sigma_2, \sigma_3, \ldots, \sigma_k$ verfahren wir jetzt genau so, usw.
  - Damit erhalten wir nach einer endlichen Anzahl von Vertauschungen eine monotone Folge

Erfahrungen ÜB11 5/6

- Lösungsskizze Aufgabe 2, Variante 1
  - Wir benutzen den "δ-Algorithmus" aus Vorlesung 11b

    Der berechnet für ein gegebenes δ genau $\min\{\delta, ED(x, y)\}$
  - Variante 1: der Reihe nach $\delta = 1, 2, 3, 4, \ldots$ durchprobieren, bis der berechnete Wert **kleiner als** δ ist
  - Das ist korrekt: Für $i = 1, \ldots, ED(x, y)$ wird in Runde i der Wert i berechnet, in Runde $ED(x, y) + 1$ der Wert $ED(x, y)$
  - Die Laufzeit ist $\mathbf{O(\min\{|x|, |y|\} \cdot D)}$, wobei

    $D = 1 + 2 + \ldots + (ED(x, y) + 1) = \Theta(\, ED(x, y)^2 \,)$

    Also **quadratisch** in der tatsächlichen Editierdistanz

Erfahrungen ÜB11 6/6

- Lösungsskizze Aufgabe 2, Variante 2
  - Variante 2: der Reihe nach $\delta = 1, 2, 4, 8, \ldots$ durchprobieren, bis der berechnete Wert **kleiner als** δ ist

    Also gerade alle **Zweierpotenzen**
  - Das ist korrekt: sobald $\delta > ED(x, y)$ wird der richtige Wert berechnet und in den Runden i davor der Wert i

    Das ist dasselbe Argument wie bei Variante 1
  - Das δ_{max} bei dem der Algorithmus abbricht ist höchstens doppelt so groß wie ED(x, y) … also $\delta_{max} \leq 2 \cdot ED(x, y)$
  - Also ist die Laufzeit $\mathbf{O(\min\{|x|, |y|\} \cdot D)}$, wobei

    $D = 1 + 2 + 4 + \ldots + \delta_{max} \leq 2 \cdot \delta_{max} = O(\ ED(x, y)\ )$

String Matching 1/3

- Definition
  - Gegeben zwei Zeichenketten / strings:

    Ein Text (engl. text) typischerweise **lang**

    Ein Muster (engl. pattern) typischerweise **kurz**
  - Finde alle Vorkommen des Musters im Text

    Es werden die Anfangspositionen zurückgegeben
  - Notation: wir benutzen durchgängig n für die Länge des Textes und m für die Länge des Musters

0 1 2 3 ...

TEXT: DUBIDUBIDUBADUBIDU

PATTERN: DUBI

AUSGABE: 0, 4, 12

String Matching 2/3

- Motivation
  - Jeder Editor hat eine "Find" Funktion (Strg+F)
  - Jede Programmiersprache hat Methoden dafür

    Python: str.find(pattern, start, end)

    Java: String.indexOf(pattern, start)

    C++: std::string.find(pattern, start)
  - Damit bekommt man das nächste Vorkommen ab einer bestimmten Position (das start)
  - Durch wiederholtes Aufrufen dann alle Vorkommen

String Matching 3/3

- Mehr Motivation
  - Auch zentral für die Plagiatserkennung
  - Da hat man allerdings typischerweise nicht nur ein Pattern, sondern eine Menge davon, die man wiedererkennen will

    Dazu mehr in der Vorlesung morgen

Naiver Algorithmus 1/2

- Prinzip + Beispiel
  - Gehe den Text von links nach rechts durch
  - Prüfe an jeder Stelle ob das Muster passt, indem man es Buchstabe für Buchstabe mit dem Text dort vergleicht
  - Den jeweiligen Ausschnitt (der Größe m) aus dem Text nennen wir Fenster (engl. window)
  - Das implementieren wir jetzt zusammen !

0 1 2 3 4 5 ..... 10 11 12 13

TEXT: DUBIDUBIDUBADU

PATTERN: DUBI

Letzter möglicher Match. Danach ist nicht mehr genug übrig vom Text.

Naiver Algorithmus 2/2

$n >> m$

- Laufzeit
  - Sei wie gehabt n = Länge Text, m = Länge Pattern
  - Laufzeit im worst case + Beispiele: $O(n \cdot m)$

    text = AAAAAAAA… pattern = AAA

    text = DUDADUDIDUDI… pattern = DUDA

    Man muss (oft) durch das ganze Pattern "durchgehen"
  - Laufzeit im best case + Beispiele: $O(n)$

    text = AACTAACCTAAGC pattern = XACAG

    Man merkt bei den ersten Buchstaben schon, dass das Pattern nicht passt

Knuth-Morris-Pratt Algorithmus 1/9

- Motivation für den Algorithmus
  - Wenn man die ersten k Zeichen des Musters mit den ersten k Zeichen eines Fensters im Text verglichen hat

  ... und jetzt das Fenster im Text um eins weiter schiebt

  ... dann hat man k – 1 Zeichen dieses Fensters schon mal mit dem Muster verglichen

  - Man möchte gerne vermeiden, die nochmal anzuschauen
  - Wie das gehen könnte, sieht man am besten an ein paar Beispielen … siehe nächste Folien

Knuth-Morris-Pratt Algorithmus 2/9

- Beispiel nicht-repetitives Muster
  - Im besten Fall kann man die Suche im Text da fort-setzen, wo der letzte "Mismatch" mit dem Muster war
  - Nehmen wir an, dass Muster ist **ABCDEFG**
  - Und nehmen wir an, an der aktuellen Textstelle passt es bis vor das **E** und dann nicht mehr **ABCDEFG**

TEXT: ... ABCD ....

PATTERN: ABCDEFG

ABCDEFG

Es reicht jetzt, hier im Text weiter zu suchen

Hier kann es gar nicht matchen (das weiß man aufgrund des Patterns)

Knuth-Morris-Pratt Algorithmus 3/9

- Beispiel repetitives Muster
  - Es kann aber auch sein, dass es davor auch noch einen Treffer gibt
  - Nehmen wir an, dass Muster ist **DUBIDUBADU**
  - Und nehmen wir an, an der aktuellen Textstelle passt es bis vor das **A** und dann nicht mehr **DUBIDUBADU**

TEXT: ...DUBIDUB...

PATTERN: DUBIDUBADU

Es wäre ein Fehler hier erst weiter zu suchen

Es reicht aber, ab hier weiter zu suchen

Hier müssen wir gar nicht mehr probieren

Knuth-Morris-Pratt Algorithmus 4/9

- Vorverarbeitung des Musters 1/3
  - Wir berechnen für jede Stelle des Musters vor, um wie viel links von der Stelle des letzten "Mismatches" man die Suche fortsetzen kann, ohne einen Treffer zu verpassen

    Dabei wollen wir so weit nach rechts gehen wie möglich
  - Erst mal ein paar Beispiele (für Muster)

	0	1	2	3	4	5	6	7	8	9
	D	U	B	I	D	U	B	A	D	U
SHIFT	0	0	0	0	1	2	3	0	1	2

m = 10

Diese 3 heißt, dass der Teilstring DUB (Länge 3) den Anfang vom Pattern matched.

	0	1	2	3	4	5
	A	A	A	A	A	A
SHIFT	0	1	2	3	4	5

m = 6

Heißt: der Teilstring AAAAA (Länge 5) matched Anfang vom Pattern

	0	1	2	3	4	5	6
	A	B	C	D	E	F	G
SHIFT	0	0	0	0	0	0	0

Knuth-Morris-Pratt Algorithmus 5/9

- Vorverarbeitung des Musters 2/3
  - Genauer gesagt, berechnen wir für jedes $j \in \{0, ..., m - 1\}$

    $$\text{shift}[j] = \max \{ k \leq j : \underbrace{\overbrace{P[j - k + 1 .. j]}^{\text{Länge } k}}_{\text{bis Position } j} = \underbrace{\overbrace{P[0 .. k - 1]}^{\text{Länge } k}}_{\text{am Anfang}} \}$$

    In Worten: die Länge des längsten Teilstückes bis Stelle j (< alles bis j), die gleich dem Anfang des Musters ist
  - Man beachte, dass per Definition $\text{shift}[j] \leq j$

Knuth-Morris-Pratt Algorithmus 6/9

- Vorverarbeitung des Musters 3/3
  - Das Feld shift lässt sich einfach iterativ in Zeit O(m) von links nach rechts berechnen:
  - Entweder **shift[j]** ist gleich **shift[j - 1] + 1**

    Wenn das Teilstück, dass zu shift[j – 1] > 0 geführt hat auch noch bei pattern[j] passt
  - Oder **shift[j]** ist gleich **0** bzw. gleich **1**

    Je nachdem ob pattern[j] != pattern[0] oder nicht

	0	1	2	3	4
PATTERN:	M	I	M	M	I
SHIFT	0	0	1	1	2

Knuth-Morris-Pratt Algorithmus 7/9

- Beschreibung des Algorithmus
  - Vorberechnung des shift Feldes wie gerade erklärt
  - Genau wie beim naiven Algorithmus:

    Fenster der Größe m über den Text schieben

    An Stelle i prüfen ob das Muster passt
  - Einziger Unterschied zum naiven Algorithmus:

    Falls erster Mismatch an Stelle j in P, dann im Text weiter an Stelle $i + j - \text{shift}[j - 1]$ bzw. bei $i + 1$ falls $j = 0$

    Treffer dabei wie Mismatch bei $j = |P|$ behandeln

    Zum Vergleich: naiver Algo. macht immer bei $i + 1$ weiter

Knuth-Morris-Pratt Algorithmus 8a/9

0 1 2 3 4 5 6 7 8 9
PATTERN: D U B I D U B A D U m = 10
SHIFT 0 0 0 0 1 2 3 0 1 2

0 1 2 3 4 . . . . 10 11 12 13 14 . . . 20 21 22 29
TEXT D U B I D U B I D U B A D U B I D U B A D U B I D U B I D U n = 30

= = = = = = = #
0 1 2 3 4 5 6 7
D U B I D U B A D U

7 = Pos. vom Mismatch
6 = 7 − 1

← 7 − SHIFT[6] = 4

= = = = = = = = = = # ← MATCH nur Mismatch am Ende
0 1 2 3 4 5 6 7 8 9
D U B I D U B A D U

← 14 − SHIFT[9] = 12

= = = = = = = = = = #
D U B I D U B A D U

← 22 − SHIFT[9] = 20

diese drei Vergleiche brauchen wir nicht noch mal zu machen

Dito

= = = = = = = #
D U B I D U B A D U

letzte Position, wo es noch matchen kann (danach nicht mehr genug Text übrig)

Knuth-Morris-Pratt Algorithmus 8b/9

0 1 2 3 4 5
A A A A A A
SHIFT 0 1 2 3 4 5

A A A X
" " " ≠
A A A A A A

Jetzt würde es sogar reichen, als Dir weiter zu rücken

Die SHIFT Werte sind nicht optimal bei KMP
reichen aber für Laufzeit $O(n)$

Knuth-Morris-Pratt Algorithmus 9/9

- Laufzeit
  - Die Laufzeit ist proportional zur Anzahl der Vergleiche

    eines Zeichens des Textes mit einem Zeichen des Patterns
  - Für jeden Vergleich gilt (siehe Bild auf Folie 21):

    1. Man schaut sich ein neues Zeichen im Text an

    Eins rechts von dem, dass man zuletzt verglichen hat

    2. Man schaut sich dasselbe Zeichen nochmal an, aber hat das Muster mindestens eins weiter "nach rechts geschoben"

    Die Verschiebung ist gerade $j - \text{shift}[j - 1] > 0$
  - Da man im Bild höchstens n mal "nach rechts" gehen kann, gibt es also höchstens 2n Vergleiche → Laufzeit O(n)

Literatur / Links

- String Matching
  - Mehlhorn/Sanders: gar nichts zu dem Thema!
- Wikipedia
  - http://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus
  - http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm
- Originalarbeit
  - Donald Knuth und James Morris und Vaughan Pratt

    Fast Pattern Matching in Strings

    1977 SIAM Journal on Computing