Exaktes Stringmatching

Theorie (L)

Version vom 7. November 2024

1 Einleitung

Hier sollen hier Algorithmen betrachtet werden, die nach *exakten* Übereinstimmungen (*matches*) suchen.

Zeichenketten und Muster werden als Listen repräsentiert, deren Elemente die einzelnen Zeichen sind.

Anwendungen

- Textverarbeitungsprogramme
- Untersuchung von DNA- und Proteinsequenzen in der Bioinformatik
- Erkennung von Plagiaten
- Virenscanner

2 Naive Methode (Brute Force)

Beispiel 2.1

Suche das Textmuster `ABBA` in der Zeichenkette `ABABBCABBACB`

A	B	A	B	B	C	A	B	B	A	C	B	Vergl.
A	B	B										3
	A											1
		A	B	B	A							4
			A									1
				A								1
					A							1
						A	B	B	A			4

Analyse (Worst Case)

Text: `aaaaaaa` ($n = 7$ Zeichen)

Pattern: `aab` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	a	Vergleiche
a	a	b					3
	a	a	b				3
		a	a	b			3
			a	a	b		3
				a	a	b	3
							(7-3+1)*3=15

Allgemein:

$O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Implementierung in Python

```
1  def matcher_naive(pat, text):
2      matches = []
3      n = len(text)
4      m = len(pat)
5      for i in range(0, n-m+1):
6          j = 0
7          while j < m and text[i+j] == pat[j]:
8              j += 1
9          if j==m:
10             matches.append(i)
11     return matches
```

3 Der Boyer-Moore-Horspool-Algorithmus

Schnelle Verschiebungen

Stimmt das Muster an irgend einer Stelle nicht mit dem entsprechenden Teilstring des Textes überein, soll es so weit wie möglich nach rechts verschoben werden, ohne einen Treffer zu verpassen.

```
B B A A A B C B B B A A B B A B A A B B B A
A B B A
→ → → A B B A
        → → → → A B B A
                → → → A B B A
                        → A B B A
```

Bei Nichtübereinstimmung (rot), kann ich das Muster so weit nach rechts verschieben, bis das letzte Zeichen im Text (blau) mit dem nächsten Zeichen im Muster übereinstimmt. Kommt das Zeichen im Muster nicht vor, kann man sogar um die Länge des Musters verschieben.

Beispiel 3.2

Suche das Muster `ABBA` (Pattern p) in der Zeichenkette `ABABBCABBACB` (Text t).

Alphabet: $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ (jedes Symbol in $t \cup p$)

Bad Character Table (BCT): Um wie viele Positionen darf man das Muster nach rechts verschieben, wenn über seinem rechten Ende im Text das Symbol $\sigma \in \Sigma$ steht und man keinen Treffer verpassen möchte?

```
? ? ? A
A B B A    ⇒   3 Zeichen
-----------------------
? ? ? B
A B B A    ⇒   1 Zeichen
-----------------------
? ? ? C
A B B A    ⇒   4 Zeichen
```

Implementierung der BCT in Python

```
1 def bad_character_table(pat, alph):
2     m = len(pat)
3     D = dict()
4     for i in range(0, len(alph)):
5         D[alph[i]] = m
6     for i in range(0, m-1):
7         D[pat[i]] = m-i-1
8     return D
```

Zeilen 4–5: Jedem Symbol wird provisorisch die Länge des Musters $m = |p|$ zugeordnet.

Zeile 6–7: Jedem Zeichen im Muster (ausser dem Letzten) wird sein kürzester Abstand vom rechten Ende zugewiesen.

Aufwand: $O(|\Sigma| + m)$

Beispiel 3.2 (Fortsetzung)

Bad Chraracter Table:

A	B	C
3	1	4

A	B	A	B	B	C	A	B	B	A	C	B	Vergl.
A	B	B	A									1
	A	B	B	A								1
		A	B	B	A							1
						A	B	B	A			4

Implementierung in Python

```
1  def matcher_bmh(pat, text, alph):
2      matches = []
3      bct = bad_character_table(pat, alph)
4      n = len(text)
5      m = len(pat)
6      i = 0 # Position im Text
7      while (i < n-m+1):
8          j = m-1 # letzte Position im Muster
9          while (j>-1 and pat[j] == text[i+j]):
10             j = j-1
11         if j == -1: # alle Zeichen matchen
12             matches.append(i)
13         i = i + bct[text[i+m-1]] # shift aufgrund BCT
14     return matches
```

Worst Case-Analyse

Text: `aaaaaa` ($n = 6$ Zeichen)

Pattern: `baa` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	a	a				3
	b	a	a			3
		b	a	a		3
			b	a	a	3
						(6-3+1)*3=12

Allgemein: $O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Solche Text-Muster-Strukturen sind jedoch eher die Ausnahme!

Best Case-Analyse

Text: `aaaaaaa` ($n = 6$ Zeichen)

Pattern: `bbb` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	b	b				1
			b	b	b	1
						(6//3)*1=2

Allgemein: $O(n/m)$

Auch solche Text-Muster-Strukturen sind eher die Ausnahme.

Average Case-Analyse

Ricardo Baeza-Yates und Mireill Régnier haben in ihrem Artikel in der Fachzeitschrift *Theoretical Computer Science* 1992 gezeigt, dass die Komplexität des Boyer-Moore-Horspool-Algorithmus im Mittel $O(n)$ ist.

Bemerkung

Der hier vorgestellten Boyer-Moore-Horspool-Algorithmus (BMH) ist eine Vereinfachung des Boyer-Moore-Algorithmus' (BM), der zusätzlich zur Bad Character Table allfällige Übereinstimmungen am Ende des Musters einbezieht, um es beim ersten Mismatch eventuell noch weiter nach rechts zu verschieben.

Wenn man den Fachartikeln im Internet Glauben schenkt, so ist für natürliche Sprachen der BMH-Algorithmus dem BM-Algorithmus überlegen. Dies liegt offenbar daran, dass der BM-Algorithmus mehr Aufwand zur Verschiebung des Suchmusters betreibt, was sich in einer grösseren Anzahl von Anweisungen niederschlägt. Siehe z. B.:

`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61442/`

4 Textsuche mittels Automaten

Die Idee

Wir werden das zu suchende Muster p als deterministischen endlichen Automaten darstellen, um in linearer Zeit alle akzeptierenden Zustände (Treffer) im Text t zu finden.

Der Schlüssel zum Verständnis ist die folgende, trivial anmutende Beobachtung:

Ein Muster p ist Teilstring eines Textes t, wenn p Präfix eines Suffixes von t ist.

Beispiel: Text: $t =$ `GCTATCTATGG`, Muster: $p =$ `TAT`

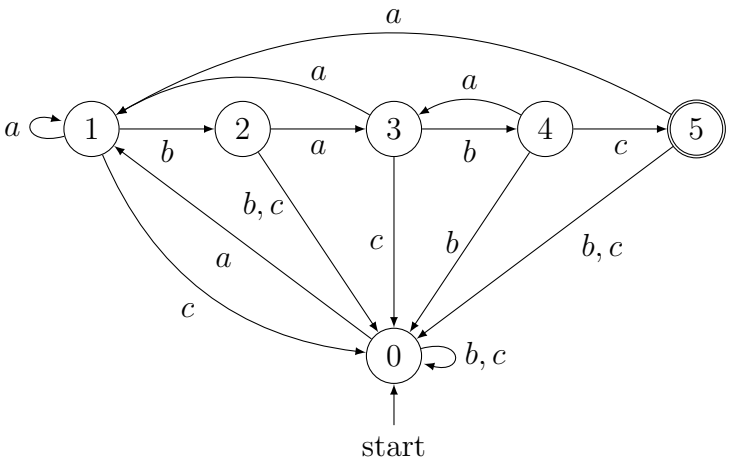
- $p =$ `TAT` ist Präfix des Suffixes `TATCTATGG` von t
- $p =$ `TAT` ist Präfix des Suffixes `TATGG` von t

Der Bauplan für den Automaten

- Die Anzahl der Zeichen dar, die bereits korrekt erkannt wurden, bilden die Zustände $q_0, q_1, \ldots, q_m$ des Automaten.
- Für jedes Zeichen σ des Alphabets Σ wird ein Übergang $\delta(q_i, \sigma) = q_j$ definiert.
- Der Automat akzeptiert genau dann eine Folge von Eingabezeichen, wenn er den Zustand q_m erreicht hat.
- Befindet sich der Automat im Zustand q_i mit $0 < i < m$ (d. h. er hat bereits einen Teil des Musters korrekt erkannt) und liegt beim nächsten gelesenen Zeichen keine Übereinstimmung mehr vor, dann geht der Automat nur so weit wie nötig zurück. Dazu müssen wir das Muster mit sich selbst vergleichen.

Beispiel 4.3

Muster $p = ababc$



Algorithmische Bestimmung des Automaten

Für die systematische Konstruktion des Automaten müssen wir zunächst alle möglichen Präfixe des Musters p (auch das leere) um jedes Zeichen $x \in \Sigma$ erweitern.

Dann bestimmen wir das längste Präfix von p, das gleichzeitig Suffix des um x erweiterten Präfixes ist.

Die Anzahl der Zeichen in diesem längsten Präfix entspricht dann gerade dem Zustand, den der Automat beim Lesen des Symbols x erreicht.

Beispiel 4.4 (systematische Konstruktion)

Muster: $p = ababc$

$$
\begin{array}{rcl@{\qquad}rcl}
a \sqsupset \varepsilon a & \to & 1 & a \sqsupset aba a & \to & 1 \\
\varepsilon \sqsupset \varepsilon b & \to & 0 & aba b \sqsupset aba b & \to & 4 \\
\varepsilon \sqsupset \varepsilon c & \to & 0 & \varepsilon \sqsupset aba c & \to & 0 \\
a \sqsupset a a & \to & 1 & aba \sqsupset aba b a & \to & 3 \\
ab \sqsupset a b & \to & 2 & \varepsilon \sqsupset aba b b & \to & 0 \\
\varepsilon \sqsupset a c & \to & 0 & ababc \sqsupset aba b c & \to & 5 \\
aba \sqsupset a b a & \to & 3 & a \sqsupset aba b c a & \to & 1 \\
\varepsilon \sqsupset a b b & \to & 0 & \varepsilon \sqsupset aba b c b & \to & 0 \\
\varepsilon \sqsupset a b c & \to & 0 & \varepsilon \sqsupset aba b c c & \to & 0
\end{array}
$$

Funktionsweise des Automaten

Wenn einmal der DFA aus dem Muster p erzeugt wurde, ist es eine einfache Aufgabe, nach den Vokommen von p in einem Text t zu suchen: Man liest den Text t zeichenweise ein und „füttert“ damit den DFA, der zuvor in den Anfangszustand q_0 gesetzt wurde. Abhängig vom gelesenen Zeichen führt der DFA einen Zustandswechsel durch oder verbleibt im gleichen Zustand. Gerät der DFA nach dem i-ten gelesenen Zeichen in den akzeptierenden Zustand, so hat er das Ende des Musters erreicht und „weiss“, dass das Muster an der Position $i - |p| + 1$ im Text t vorkommt, wobei $|p|$ für die Länge des Musters p steht.

Beispiel 4.5

Wir suchen im Text $t =$ `aaababcababcc` nach dem Muster $p =$ `ababc`, wobei wir den in Beispiel 4.3 bzw. Beispiel 4.4 konstruierten DFA verwenden.

											1 0	1 1	1 2
Index	-	0	1	2	3	4	5	6	7	8	9			
Symbol:	ε	a	a	a	b	a	b	c	a	b	a	b	c	c
State:	0	1	1	1	2	3	4	5	1	2	3	4	5	0
Match:	-	-	-	-	-	-	-	2	-	-	-	-	7	-

Generierung des DFAs in Python

```
def build_dfa(pat, alph):
    '''Gibt einen DFA aus Muster und Alphabet zurück.'''
    m = len(pat)
    delta = [dict() for i in range(0, m+1)]

    for q in range(0, m+1):
        for x in alph:
            s1 = pat[:q] + x
            for k in range(m, -1, -1):
                s2 = pat[:k]
                if s1.endswith(s2):
                    delta[q][x] = k
                    break

    return delta
```

Mustersuche mit einem DFA in Python

```
def matcher_dfa(pat, text, alph):
    '''Gibt eine Liste mit den Anfangspositionen
    der Matches zurück.'''
    matches = []
    delta = build_dfa(pat, alph)
    q = 0
    m = len(pat)
    for i, c in enumerate(text):
        q = delta[q][c]
        if q == m:
            matches.append(i-m+1)
    return matches
```