Notizen zu Algorithmen II

Jens Ochsenmeier

4. Februar 2018

Inhaltsverzeichnis

1 Stringology

Inhalt dieses Kapitels:

- Strings sortieren
- Patterns suchen
- Datenkompression

1.1 Strings sortieren

Naive Sortierverfahren, wie sie aus der Vorlesung "Algorithmen 1" bekannt sind, sind beim Sortieren von Strings ineffizient, deswegen gibt es für das Sortieren von Strings andere Algorithmen. Ein solcher ist der **Multikey Quicksort**-Algorithmus:

```
MKQSORT (S: String Seq, l : ℕ): String Seq
assert ∀e, e' ∈ S : e[1...l−1] = e'[1...l−1]
if |S| ≤ 1 then return S
pick p ∈ S randomly
return concatenation of
   MKQSORT (⟨e ∈ S : e[l] < p[l]⟩, l),
   MKQSORT (⟨e ∈ S : e[l] = p[l]⟩, l + 1),
   MKQSORT (⟨e ∈ S : e[l] > p[l]⟩, l)
```

Abbildung 1.1. Pseudocode-Implementierung des Multikey-Quicksort-Algorithmus.

Pattern
Text

Dieser Algorithmus sortiert eine String-Sequenz und nimmt an, dass die ersten $l - 1$ Buchstaben bereits sortiert wurden.
Zuerst wird ein zufälliges Pivotelement gewählt. Danach wird die übergebene Sequenz an Strings wird in drei Teilsequenzen geteilt:

1. Sequenz an Strings, deren l-ter Buchstabe kleiner ist als der l-te Buchstabe des Pivotelements.
2. Sequenz an Strings, deren l-ter Buchstabe derselbe ist wie der l-te Buchstabe des Pivotelements.
3. Sequenz an Strings, deren l-ter Buchstabe größer ist als der l-te Buchstabe des Pivotelements.

Auf die erste und dritte Teilsequenz wird der Algorithmus nun rekursiv mit dem selben Parameter l ausgeführt, da die Buchstaben an der l-ten Position nicht übereinstimmen (müssen) — auf die zweite Teilsequenz wird der Algorithmus rekursiv mit dem Parameter $l + 1$ ausgeführt, weil hier die l-ten Buchstaben aller Wörter in der Sequenz gleich sind.
Die Laufzeit des Algorithmus ist in $O(|S| \log |S| + d)$, wobei d die Summe der eindeutigen Präfixe der Strings in S ist.

1.2 Pattern Matching

Hinweis: In diesem Abschnitt sind Arrays 1-basiert.

In diesem Abschnitt wird es darum gehen, alle oder zumindest ein Vorkommen eines **Patterns** $P = p_1 \dots p_m$ in einem gegebenen **Text** $T = t_1 \dots t_n$ zu finden. Im Allgemeinen ist $n \gg m$, also der Text wesentlich länger als das Pattern, das wir in ihm suchen.

Naives Pattern Matching

Das naive Vorgehen ist, an jeder Position von T zu schauen, ob an dieser das gesuchte Pattern vorkommt. Offensichtlich ist dieser Algorithmus in $O(nm)$, da im schlimmsten Fall für jede Position des Textes das gesamte Pattern durchlaufen werden muss. Dieser Algorithmus kann folgendermaßen implementiert werden:

Knuth-Morris-Pratt-Algorithmus

```
naivePatternMatch (P, T)
i, j := 1
while i ≤ n − m + 1
   while j ≤ m ∧ t_{i+j−1} = p_j do j++
   if j > m then return "P occurs at pos i in T"
   i++
   j := 1
```

Abbildung 1.2. Pseudocode-Implementierung des naiven Pattern-Matching-Algorithmus.

Knuth-Morris-Pratt

Ein anderer Algorithmus zum Finden von Patterns in einem gegebenen Text ist der **Knuth-Morris-Pratt-Algorithmus**. Dieser hat sogar optimale Laufzeit, nämlich $O(n + m)$.
Idee dieses Algorithmus ist es, das Pattern eleganter nach vorne zu verschieben, wenn es einen Mismatch zwischen Text und Pattern gibt. Hierfür brauchen wir ein Hilfswerkzeug:
Für einen String S mit Länge k sei $\alpha(S)$ die Länge des Längsten Präfixes von $S_{1\ldots k-1}$, das auch Suffix von $S_{2\ldots k}$ ist.[1]

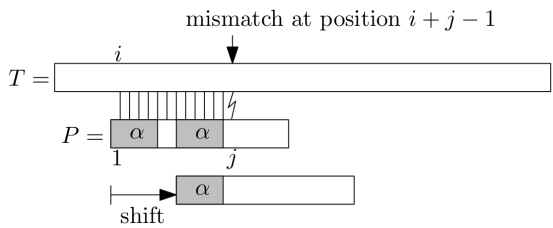

Abbildung 1.3. Idee beim Verschieben des Patterns: α wurde bereits gematcht. Früher als mit dem bereits gematchten Suffix kann das nächste Vorkommen von P nicht auftauchen, also kann mann P direkt um $j - 1 - \alpha$ verschieben.

Der Algorithmus besteht aus zwei Teilen:

1. **Border-Array berechnen** ($O(m)$). Damit die oben erläuterten Verschiebungen nachher effizient durchgeführt werden können, berechnen wir für das leere Wort

[1] Wir lassen absichtlich bei Betrachtung des Präfixes den letzten und bei Betrachtung des Suffixes den ersten Buchstaben weg, damit $\alpha(S) = 0$ ist, wenn $|S| = k = 1$ ist.

Border-Array
String
Suffix

und jeden Buchstaben in P einen α-Wert. Diese Werte ergeben das **Border-Array**:

$$\text{border}[j] = \begin{cases} -1, & \text{falls } j = 1 \\ \alpha(P_{1\ldots j-1}), & \text{sonst} \end{cases}.$$

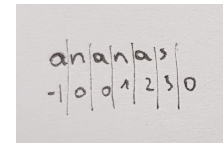

Abbildung 1.4. Beispiel für das Border-Array eines Patterns.

2. **Pattern matchen** ($O(n)$). Nun verwenden wir das erstellte Border-Array, um Vorkommnisse von P in T zu finden. Wir starten sowohl im Text als auch im Pattern an Position 1 und fangen an zu matchen. Kommt es an Position $1 \leq j \leq m$ des Patterns zu einem Mismatch, so können wir P direkt um $j - \text{border}[j] - 1$ verschieben. In Pseudocode sieht das so aus:

```
KMPMatch (P,T)
i, j := 1
while i ≤ n − m + 1
   while j ≤ m ∧ t_{i+j−1} = p_j do j++
   if j > m then return "P occurs at pos i in T"
   i += j − border[j] + 1
   j := max{1, border[j] + 1}
```

Eine Ausführung des Algorithmus kann also folgendermaßen aussehen:

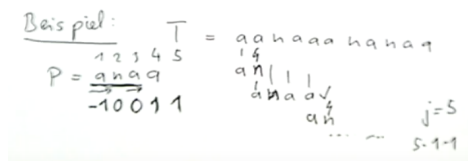

Abbildung 1.5. Beispiel für das Verwenden des Knuth-Morris-Pratt-Algorithmus.

Suffix-Arrays

Im Folgenden werden Arrays wieder mit Position 0 beginnen. Wir verwenden desweiteren folgende Festlegungen:

- Ein **String** ist ein Array von Buchstaben,

$$S[0\ldots n) := S[0\ldots n-1] := [S[0], \ldots, S[n-1]].$$

- Das **Suffix** S_i sei der Substring $S[i\ldots n)$ von S.

- Wir setzen an das Ende jedes Strings ausreichend viele **Endmarkierungen**: $S[n] := S[n+1] := \cdots := 0$. 0 sei per Definition kleiner als alle anderen vorkommenden Zeichen.

Endmarkierung
Suffix-Array

Das **Suffix-Array** eines Strings lässt sich nun folgendermaßen konstruieren:

1. Bilde die Menge aller Suffixe S_i ($i = 0, \ldots, n-1$) des Strings.
2. Sortiere die Menge aller Suffixe des Strings (z.B. mit Multikey Quicksort).

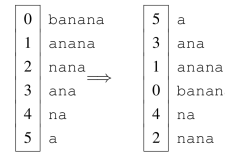

Abbildung 1.6. Beispiel für die Konstruktion des Suffix-Arrays des Strings *"banana"*.

Mithilfe dieses Suffix-Arrays lassen sich später viele Suchprobleme in Linearzeit lösen. Beispielsweise ist die Suche nach dem längsten Substring, der (eventuell mit Überschneidung) zweimal im Text vorkommt, linear — dafür muss nach Berechnung des Suffix-Arrays der längste String gefunden werden, der Präfix von zwei Strings im Suffix-Array ist (im Beispiel oben wäre das *"ana"*).

Berechnung des Suffix-Arrays in Linearzeit

Das Suffix-Array eines Strings lässt sich in Linearzeit berechnen.[2] Hier soll lediglich das Prinzip erläutert werden, genauere Angaben gibt es im Paper.
Wir betrachten den String

$$T[0,n) = \underset{0}{\mathtt{x}}\,\underset{1}{\mathtt{a}}\,\underset{2}{\mathtt{b}}\,\underset{3}{\mathtt{b}}\,\underset{4}{\mathtt{a}}\,\underset{5}{\mathtt{d}}\,\underset{6}{\mathtt{a}}\,\underset{7}{\mathtt{b}}\,\underset{8}{\mathtt{b}}\,\underset{9}{\mathtt{a}}\,\underset{10}{\mathtt{d}}\,\underset{11}{\mathtt{o}}.$$

Unser Ziel ist das Suffix-Array

$$\mathrm{SA} = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0).$$

Wir gehen wie folgt vor:

0. **Suffixe wählen.** Sei

$$B_k = \{i \in [0,n] : i \mod 3 = k\}$$

und $C = B_1 \cup B_2$ sowie S_C die Menge der entsprechenden Suffixe. C ist also die Menge aller Positionen in T, an denen Suffixe mit einer nicht durch 3 teilbaren Länge beginnen. Hier ist $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

[2] Kärkkäinen, Sanders, Burkhardt: Linear Work Suffix Array Construction

1. **Gewählte Suffixe sortieren**. Wir fügen am Ende von T beliebig viele $\mathtt{0}$ hinzu und bilden zuerst für $k = 1, 2$ die Strings

   $$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}].$$

   Der Charaktere von R_k sind also Tripel. Das letzte Tripel ist immer eindeutig, weil $t_{\max B_k+2} = 0$. Sei $R = R_1 \odot R_2$.
   Hier ist

   $$R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}].$$

   Die Ordnung der Suffixe von R stimmt mit der Ordnung der Suffixe S_i überein, deswegen genügt es, die Suffixe von R zu sortieren.
   Wir sortieren R nun, indem wir die einzelnen Charaktere von R sortieren und durch ihren Rang in R ersetzen:

   $$\mathrm{SA}_R = (8, 0, 1, 6, 4, 2, 5, 3, 7).$$

   Nun weisen wir jedem Suffix einen Rang zu. Dazu sei $\mathrm{rank}(S_i)$ der Rang von S_i in C. Für $i \in B_0$ sei $\mathrm{rank}(S_i)$ nicht definiert.
   Hier ist $\mathrm{rank}(S_i) = \bot\ 1\,4\ \bot\ 2\,6\ \bot\ 5\,3\ \bot\ 7\,8\ \bot\ 0\,0$.
2. **Restliche Suffixe sortieren**. Jeder Suffix $S_i \in S_{B_0}$ sei dargestellt durch $(t_i, \mathrm{rank}(S_{i+1}))$. Da wir alle anderen Suffixe oben schon sortiert haben ist $\mathrm{rank}(S_{i+1})$ hier stets definiert.
   Offensichtlich ist

   $$S_i \le S_j \iff (t_i, \mathrm{rank}(S_{i+1})) \le \big(t_j, \mathrm{rank}(S_{j+1})\big),$$

   also lassen sich die Paare Radix-sortieren.
   Hier ist

   $$S_{12} < S_6 < S_9 < S_3 < S_0, \quad \text{weil} \quad (\mathtt{0}, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{x}, 1).$$

3. **Zusammenführen**. Das Zusammenführen erfolgt vergleichsbasiert. Beim Vergleichen on $S_i \in S_C$ mit $S_j \in S_{B_0}$ unterscheiden wir zwei Fälle:

   $$i \in B_1: \quad S_i \le S_j \iff (t_i, \mathrm{rank}(S_{i+1})) \le \big(t_j, \mathrm{rank}(S_{j+1})\big)$$
   $$i \in B_2: \quad S_i \le S_j \iff (t_i, t_{i+1}, \mathrm{rank}(S_{i+2})) \le \big(t_j, t_{j+1}, \mathrm{rank}(S_{j+2})\big)$$

   Hier ist z.B. $S_1 < S_6$ weil $(\mathtt{a}, 4) < (\mathtt{a}, 5)$ und $S_3 < S_8$ weil $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Suchen in Suffix-Arrays

LCP-Array
Suffix-Array!invertiert
Suffix-Baum

Um ein Pattern in einem String zu finden, zu dem man das Suffix-Array konstruiert hat, muss man lediglich ein Suffix finden, dass das gesuchte Pattern als Präfix hat. Man kann so beispielsweise mit binärer Suche in $O(m \log n)$ ein Vorkommen von P in T finden.

Nutzen wir eine zusätzliche Struktur, das **LCP-Array** — dieses speichert in $\mathrm{LCP}[i]$ die Länge des längsten gemeinsamen Präfixes von $\mathrm{SA}[i]$ und $\mathrm{SA}[i-1]$ — so können wir die Suchzeit auf $O(m + \log n)$ reduzieren.

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

0	a
1	**a**na
3	**ana**na
0	banana
0	na
2	**na**na

Abbildung 1.7. Suffixe, Suffix-Array und LCP-Array des Strings *"banana"*.

Um das LCP-Array berechnen zu können brauchen wir das **invertierte Suffix-Array**. Dieses gibt Aufschluss darüber, wo im Suffix-Array ein bestimmter Suffix steht. Offensichtlich ist $\mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$.

Der Algorithmus sieht folgendermaßen aus ($O(n)$):

```
CALCULATELCPARRAY (SA^-1, SA)
h := 0, LCP[1] := 0
for i = 1, ..., n do
   if SA^-1[i] ≠ 1 then
      while t_{i+h} = t_{SA[SA^-1[i]-1]+h} do h++
      LCP[SA^-1[i]] := h
      h := max(0, h − 1)
```

Suffix-Bäume

Noch anschaulicher, allerdings wesentlich platzverbrauchender, sind **Suffix-Bäume** von Strings. Sie sind formal der *kompaktierte Trie der Suffixe* und lassen sich (wenn auch sehr kompliziert) in $O(n)$ berechnen.

Datenkompression

Bevor wir den Suffix-Baum eines Strings bilden hängen wir hinten an den String noch einen Charakter dran, der nicht im Alphabet des Strings vorkommt. Das hat den Vorteil, dass anschließend alle Suffixe in einem Blatt des Baums enden.

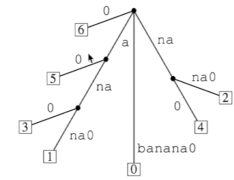

Abbildung 1.8. Beispiel für den Suffix-Baum des Strings *"banana"*.

"Naiv" ist die Erstellung des Suffixbaums in $O(n^2)$. Man kann ihn aber auch aus Suffix-Array und LCP-Array in Linearzeit konstruieren. Dazu hängt man die Suffixe sukzessive in der Tiefe ein, die ihr LCP-Wert angibt:

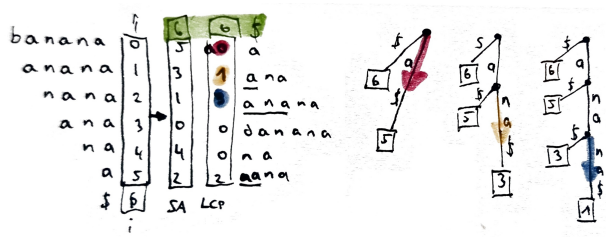

Abbildung 1.9. Sukzessive Konstruktion des Suffix-Baums aus Suffix- und LCP-Array. Zuerst hängt man $ und das erste Suffix (dessen LCP-Wert immer 0 ist) an die Wurzel. Anschließend nutzt man den LCP-Wert des darauffolgenden Suffixes (hier 1), um festzulegen, wo der Suffix zum Baum hinzugefügt werden muss (durch Pfeile gekennzeichnet).

Die Suche in einem Suffix-Baum ist relativ simpel — man muss lediglich den entsprechenden Kanten entlanglaufen, alle Vorkommen des Patterns liegen im entsprechenden Teilbaum.
Zur Angabe der Komplexitäten sind zwei Fälle zu unterscheiden:

1. Die ausgehenden Kanten sind als Arrays der Größe $|\Sigma|$ gespeichert. Dann ist die Suchzeit in $O(m)$ und der Gesamtplatzbedarf in $O(n\,|\Sigma|)$.
2. Die ausgehenden Kanten sind als Arrays gespeichert, deren Größe proportional zur Anzahl der Kinderknoten ist. Dann ist die Suchzeit in $O(m \log |\Sigma|)$ und der Gesamtplatzbedarf in $O(n)$.

1.3 Datenkompression

Eine Anwendung der Suffix-Arrays und -Trees ist die **Datenkompression**.

Index