Notizen zu Algorithmen II

Jens Ochsenmeier

3. 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]

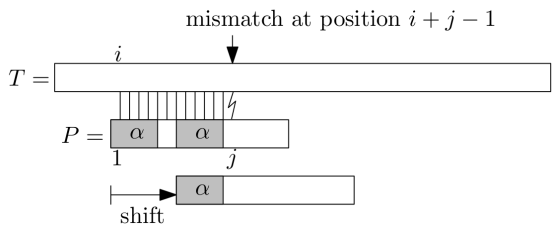

Abbildung 1.3. Idee beim Verschieben des Patterns: α wurde bereits gematcht. Früher als mit dem bereits gematchten Suffix kann das nächste Vorkommen von P nicht auftauchen, also kann mann P direkt um $j-1-\alpha$ verschieben.

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**:

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

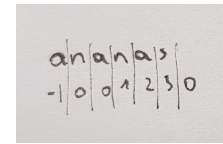

Abbildung 1.4. Beispiel für das Border-Array eines Patterns.

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:

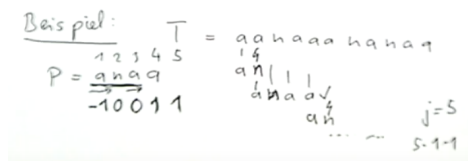

Abbildung 1.5. Beispiel für das Verwenden des Knuth-Morris-Pratt-Algorithmus.

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
Suffix-Baum

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).

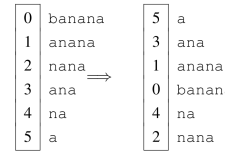

Abbildung 1.6. Beispiel für die Konstruktion des Suffix-Arrays des Strings *"banana"*.

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"*).

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.

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.

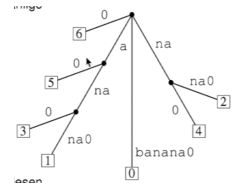

Abbildung 1.7. Beispiel für den Suffix-Baum des Strings *"banana"*.