Exakte Suche/ Pattern Matching

K. Dührkop, M. Fleischauer

Lehrstuhl für Bioinformatik
Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena

Exakte Suche

- Suche nach einem String in einem anderen.
- Definition:
  Gegeben sei eine Sequenz $a = a_1 \dots a_n$ und ein Muster $b = b_1 \dots b_m$. Ein Vorkommen von b in a (an Position i) ist eine Subsequenz $a_i \dots a_{i+m-1}$ von a, die exakt gleich b ist. Exakte Suche ist das Problem die/ein Vorkommen von b in a zu finden.

Anwendungen in der Bioinformatik

- direkt: Suche nach exakten Teilsequenzen
- deutlich schneller als nicht-exakte Suche, deshalb für Approximation nicht-exakter Suche.
- Sequenzen können Nukleotid- oder AS-Sequenzen sein, aber durchaus auch “lesbare” Texte

Das Naive Verfahren

- Prinzip: das Muster wird an allen Positionen der Sequenz von Neuem mit der Sequenz verglichen.

```
          1         2         3
12345678901234567890123456789012
anna mag banane lieber als ananas
ananas
 ananas
  ananas ...
          ananas
           ananas ...
                              ananas
                               ananas ...
```

- Komplexitt: Worst-Case $O(n \cdot m)$. Wird das Muster jeweils nur soweit wie ntig verglichen, ergibt sich in der Praxis oft $O(n + m)$ im Mittel.

Knuth-Morris-Pratt Algorithmus

- Idee
  - Vermeide Teile der Sequenz mehrfach zu lesen.
  - Verschiebe das Muster soweit wie mglich, statt wie im naiven Verfahren immer nur um eine Position.
- Beispiel

```
          1         2         3
1234567890123456789012345678901234
anna mag banane lieber als ananas
ananas
  ananas ...
         ananas
           ananas...
                                ananas ...
```

- Verschiebung lsst sich aus Muster berechnen, dann $O(n)$

Knuth-Morris-Pratt Algorithmus

- Berechnung der nchsten Position im Muster
  - `next[j] = 1` + Lnge des lngsten echten Suffixes von $b_1 \ldots b_{j-1}$, das zugleich Prfix von b ist
  - `next[1] = 0` (Stopper-Wert)
- Konstruktion des next-Arrays durch „Suche des Musters in sich selbst.“
- Verfahren zur Suche von b in a (Lngen m und n)

```
int i=1, j=1; // Zeiger in Sequenz(i) und Muster(j)
while (i<=n && j<=m)
   if (j==0 || a[i] == b[j])
      i++; j++;
   else
      j = next[j];
```

`if (i<=n)` gefunden an Position i-m `else` nicht gefunden

Boyer-Moore Algorithmus

- Idee
  - viele Textzeichen kommen im Muster nicht oder nur selten vor
  - das lsst sich ausnutzen, um nicht mehr alle Textzeichen zu betrachten
- Prinzip
  - das Muster wird von rechts beginnend mit der Sequenz verglichen, aber wie bisher von links nach rechts ber die Sequenz geschoben
  - falls das Zeichen der Sequenz gar nicht im Muster auftaucht, kann das Muster komplett um eine MusterInnge verschoben werden.(Bad-Character)
  - sonst lsst sich immer noch berechnen wie weit das Muster bei einem Mismatch verschoben werden kann.(Good-Suffix)

Bad-Character-Heuristik

```
Hoola-Hoola girls like Hooligans
Hooligan
     Hooligan
       Hooligan
              Hooligan
                     Hooligan
```

(Strong) Good-Suffix-Heuristik

```
reinesupersauersupesupersupe
supersupe
      supersupe
                supersupe
                     supersupe
```

Boyer-Moore Algorithmus

- $delta_{BCH}$-Tabelle

$$delta_{BCH}(c) = \begin{cases} m - j & \text{falls } b_j \text{ rechtestes Vorkommen von } c \text{ in } b \text{ ist} \\ m & \text{falls } c \text{ nicht in } b \end{cases}$$

- $delta_{GSH}$-Tabelle

$$delta_{GSH}(j) = \begin{cases} j - k & \text{falls } b_{k+1} \dots b_{k+m-j} \text{ mit } b_k \neq \\ & b_j \text{ rechtestes Vorkommen von } b_{j+1} \dots b_m \text{ ist} \\ m & \text{falls } b_{k+1} \dots b_{k+m-j} \text{ mit } b_k \neq b_j \text{ nicht in b vorkommt} \end{cases}$$

Boyer-Moore Algorithmus

- Algorithmus zur Suche von b in a (Lngen m und n)

```
int i=m, j=m; // Zeiger in Sequenz und Muster
while (j>=1 && i<=n)
   if (a[i] == b[j])
      i-; j-;
   else
      i+=max(m-j+1, delta_BCH(a[i]), delta_GSH(j))
      j=m
return i+1;
```