Stefan Kurtz **Universität Hamburg**
Christina de Bruyn Kops **Zentrum für Bioinformatik**

Grundlagen der Sequenzanalyse
Wintersemester 2016/2017
Übungen zur Vorlesung: Ausgabe am 08.11.2016

Punkteverteilung: Aufgabe 4.1: 4 Punkte, A. 4.2: 3 Punkte, A. 4.3: 3 Punkte

Abgabe bis zum 14.11.

Aufgabe 4.1 Gegeben seien zwei Sequenzen u und v jeweils der Länge n und ein zusätzlicher Parameter k. In beiden Sequenzen gibt es jeweils $n-k+1$ Substrings der Länge k (im folgenden k-mere genannt). Folglich gibt es $O(n^2)$ Paare von k-meren von u und v. Für jedes Paar von k-meren definieren wir den *match-count* als die Anzahl von gegenüberliegenden Zeichen, die gleich sind, wenn man die k-mere zeichenweise vergleicht. So ist zum Beispiel für $k=5$ und die beiden k-mere `acata` und `agaaa` der *match-count* gleich 3.

Wir betrachten nun das Problem der Berechnung von *match-count* für jedes der $O(n^2)$ Paare von k-meren in u und v. Beispiel: Gegeben seien die Sequenzen `aaaba` und `baaba` sowie $k=3$, d.h. wir betrachten die Substrings `aaa`, `aab` und `aba` sowie `baa`, `aab` und `aba`. Damit ergeben sich die folgenden Substring-Paare mit den angegebenen *match-count* (`mc`) Werten:

```
mc(aba, baa)= 1
mc(aab, baa)= 1
mc(aba, aab)= 1
mc(aaa, baa)= 2
mc(aab, aab)= 3
mc(aba, aba)= 3
mc(aaa, aab)= 2
mc(aab, aba)= 1
mc(aaa, aba)= 2
```

Dieses beschriebene Problem kann leicht in $O(kn^2)$ Zeit gelöst werden. Entwickeln Sie einen Algorithmus, der das Problem in $O(n^2)$ Zeit löst. Beschreiben Sie diesen Algorithmus z.B. durch Pseudocode bzw. implementieren Sie ihn.

Hinweis: Um das Problem in $O(n^2)$ Zeit zu lösen, müssen Sie sich überlegen, welchen Zusammenhang es zwischen zwei aufeinander folgenden Paaren

$$mc(u[i..i+k-1],v[j..j+k-1])$$
$$mc(u[i-1..i+k-2],v[j-1..j+k-2])$$

von k-meren gibt. Offensichtlich haben sich hier nur höchstens zwei von k Zeichenpaaren geändert: $(u[i-1],v[j-1])$ fällt weg, und $(u[i+k-1],v[j+k-1])$ kommt hinzu. Damit ist der *match-count* Beitrag des um eine Position nach rechts verschobenen Paares von k-meren abhängig von dem Vorgängerpaar und lässt sich in konstanter Zeit (unabhängig von k) berechnen.

Damit Sie diesen Zusammenhang nutzen können, müssen Sie bei der Aufzählung aller Paare eine geschickte Reihenfolge wählen.

Aufgabe 4.2 Das exakte String Matching Problem ist ein klassisches Informatik-Problem. Es besteht darin, alle Vorkommen eines Musters p der Länge m in einem Text t der Länge n zu finden, d.h. alle Positionen j, $1 \leq j \leq n - m + 1$, so dass $t[j \ldots j + m - 1] = p$ ist. Dieses Problem kann man z.B. dadurch lösen, dass man an allen möglichen Positionen in t jeweils testet, ob p zum entsprechenden Substring in t passt, siehe Alg. 1. Dieser naive Ansatz benötigt $O(mn)$ Zeit.

Algorithm 1 Ein naiver Algorithmus für das exakte String Matching Problem.

Require: Text t der Länge n, Pattern p der Länge m

```
1: for j ← 1 to n − m + 1 do
2:     i ← 1
3:     while i ≤ m and t[j + i − 1] = p[i] do
4:         i ← i + 1
5:     end while
6:     if i = m + 1 then
7:         print "match at position " j
8:     end if
9: end for
```

Beschreiben Sie einen Algorithmus, der das String Matching Problem effizienter, d.h. durch weniger Buchstabenvergleiche, löst. Es ist nicht notwendig, dass ihr Algorithmus auch im worst case weniger Buchstabenvergleiche durchführt als der naive Algorithmus.

Hinweis: Ihr Algorithmus könnte z.B. ein „Fenster“ der Länge m von links nach rechts über den Text schieben und das Muster p mit dem Fensterinhalt vergleichen.

Bestimmte Teilworte aus t der Länge m, bei denen klar ist, dass sie nicht zu Treffern führen, könnten übersprungen werden, und zwar dadurch, dass man das „Fenster“ mehr als eine Position nach rechts verschiebt. Die Verschiebespanne könnte man z.B. auf der Basis des letzten Buchstabens im aktuellen Fenster bestimmen. Eine Vorverarbeitung von p unabhängig vom Text ist dabei notwendig, um diese „Verschiebespanne“ effizient bestimmen zu können.

Beispiel: Sei $t =$ `aatgacgacatacgtga` und $p =$ `acgat`. Wir geben für die folgenden Fälle an, um wieviele Positionen man das Fenster nach rechts verschieben kann, ohne einen Treffer zu verlieren. Dabei ist das aktuelle Fenster in t jeweils unterstrichen.

aktuelles Fenster	Verschiebespanne
aatga<u>cgacatacgtga</u>	1
aa<u>tgacgacatacgtga</u>	3
aatg<u>acgacatacgtga</u>	3
aatgacg<u>acata</u>cgtga	1
aatgacga<u>catac</u>gtga	3
aatgacgacat<u>acgtga</u>	2

Versuchen Sie nun aus diesem Beispiel eine Regel abzuleiten, aus der Sie dann einen Algorithmus entwickeln könnten.

Aufgabe 4.3 Überlegen Sie sich, wie man den Algorithmus zur Berechnung der Editdistanz erweitern könnte, dass eine Transposition, d.h. Vertauschung zweier benachbarter Basen kostenlos ist.

Formal bedeutet dies, dass es für ein Alphabet $\mathcal{A}$ mit $a, b \in \mathcal{A}$ eine weitere Editoperation $ab \to ba$ gibt. Um diese zu bewerten, wird neben der aus dem Vorlesungsskript bekannten Kostenfunktion δ eine Kostenfunktion $\delta' : \mathcal{A}^2 \times \mathcal{A}^2 \to \mathbb{R}$ definiert, die folgende Eigenschaften besitzt:

- $\delta'(ab \to ba) = 0$ für alle $a, b \in \mathcal{A}$
- $\delta'(ab \to cd) > 0$ für alle $a, b, c, d \in \mathcal{A}$ mit $a \neq d$ oder $b \neq c$

Skizzieren Sie, welche zusätzlichen Kanten im Editgraph benötigt werden, um die Vertauschung zu berücksichtigen. Geben Sie schließlich die Rekurrenzgleichung zur Berechnung der DP-Matrix für das erweiterte Modell an.

Bitte die Lösungen zu diesen Aufgaben bis zum 14.11.2016 abgeben. Die Besprechung der Lösungen erfolgt am 15.11.2016.

Lösung zu Aufgabe 4.1:

Für alle i, j $k \leq i \leq m - k + 1$ und $k \leq j \leq n - k + 1$ sei $MC(i, j) = mc(u[1 \ldots i], v[1 \ldots j])$. Falls $i = k$ oder $j = k$, dann berechnet man $MC(i, j)$ durch paarweise Zeichenvergleiche jeweils in $O(k)$ Zeit. Falls $i = k$ ist, dann gibt es $n - k$ Fälle, in denen diese direkte Berechnung erfolgen muss. Falls $j = k$, dann dann gibt es $m - k$ Fälle, in denen diese direkte Berechnung erfolgen muss. Also ist der Aufwand für diesen Randfall insgesamt $(m + n - 2k)k \in O(k(m + n)$. Für den Fall $i > k$, $j > k$ und $i + k - 1 \leq m$ sowie $j + k - 1 \leq n$ gilt

$$\begin{aligned} MC(i, j) = MC(i - 1, j - 1) &- (\text{if } u[i - 1] = v[j - 1] \text{ then } 1 \text{ else } 0) \\ &+ (\text{if } u[i + k - 1] = v[j + k - 1] \text{ then } 1 \text{ else } 0) \end{aligned}$$

Damit kann man die Matrix MC diagonalweise berechnen und zwar jede Diagonale mit wachsenden Spaltern- und Zeilenindices. Insgesamt berechnet man so alle $O(mn)$ Werte jeweils in konstanter Zeit. Da $k \leq m$ und $k \leq n$ ist, gilt $km \in O(mn)$ und $kn \in O(mn)$ so dass inklusive des Aufwandes für die Randfälle $O(mn)$ Zeit benötigt wird.

Ein Implementierung (ohne eine explizite Matrix) findet man im folgenden Programm.

```ruby
#!/usr/bin/env ruby

# we use this function as a mixin for Strings
class String
  def matchcount(string, k)
    (-(self.length-k)).upto(string.length-k) do |d|
      # determine start positions
      (i,j) = case
        when d < 0
          [-d, 0]
        when d == 0
          [0, 0]
        else  # diagonal > 0
          [0, d]
      end

      # shift a window along the diagonals
      mc = 0
      while i < self.length and j < string.length do
        # window not full yet, count matches
        if i < k or j < k then
```

```
          mc += 1 if self[i] == string[j]
        else
          # window full, update matchcount
          mc -= 1 if (self[i-k] == string[j-k])
          mc += 1 if (self[i] == string[j])
        end
        # return matchcount to callback function
        if i >= k-1 and j >= k-1 then
          yield self[i-(k-1)..i], string[j-(k-1)..j], mc
        end
        # advance along the diagonal
        i += 1
        j += 1
      end
    end
  end
end

if ARGV.length != 3
  STDERR.puts "Usage: #$0 sequence1 sequence2 k"
  exit 1
end
begin
  ARGV[0].matchcount(ARGV[1], Integer(ARGV[2])) do |s1, s2, mc|
    puts "mc(#{s1}, #{s2}) \t= #{mc}"
  end
rescue ArgumentError => e
  STDERR.puts "#{ARGV[2]} is not a number:", e
end
```

Lösung zu Aufgabe 4.2:

Siehe Algorithmus von Boyer-Moore-Horspool:

Horspool RN. Practical fast searching in strings. *Software – Practice & Experience*, 10(6):501–506 (1980).

Eine Beispielimplementation findet sich in den *GenomeTools* (siehe *http://genometools.org*, in Datei `extended/string_matching.c`):

```
#include <limits.h>

/* Gets called for every match. If it returns true, the matching is stopped. */
typedef bool (*ProcessMatch)(unsigned long pos, void *data);

void gt_string_matching_bmh(const char *s, unsigned long n,
                            const char *p, unsigned long m,
                            ProcessMatch process_match, void *data)
{
  unsigned long i, j, pos, d[UCHAR_MAX];
  if (m > n || !m || !n) /* no match possible */
    return;
  /* preprocessing */
  for (i = 0; i < UCHAR_MAX; i++)
    d[i] = m;
  for (j = 0; j < m-1; j++)
    d[(unsigned) p[j]] = m-(j+1);
  /* searching */
```

```
  pos = 0;
  while (pos <= n-m) {
    j = m;
    while (j > 0 && s[pos+j-1] == p[j-1])
      j--;
    if (j == 0 && process_match) {
      if (process_match(pos, data))
        return;
    }
    pos += d[(unsigned) s[pos+m-1]];
  }
}
```

Lösung zu Aufgabe 4.3: Seien $u, v \in A^*$ zwei Sequenzen und $u[i]$ bzw. $v[j]$ mit $i, j > 1$ das jeweils letzte Zeichen einer Transposition in einem optimalen Alignment.

$$\begin{array}{lcccc} & u[i-1] & u[i] & \\ & \downarrow & \downarrow & \\ u: \ldots & a & b & \ldots \\ v: \ldots & b & a & \ldots \\ & \uparrow & \uparrow & \\ & v[j-1] & v[j] & \end{array}$$

Offensichtlich gilt also in diesem Fall:

$$u[i-1] = v[j] \text{ und } u[i] = v[j-1]$$

Um diesen Fall in einer Rekurrenzgleichung zu behandeln, wird die Editdistanz bis direkt vor der Transposition (d.h. $E_\delta(i-2, j-2)$) zu Hilfe genommen. Wird eine Transposition erkannt, so wird dieser Wert (zuzüglich der Transpositionskosten) mit in die Minimierung, die den aktuellen Editdistanz-Wert liefert, einbezogen:

$$E_\delta(i,j) = \min \left\{ \begin{array}{ll} \left.\begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \epsilon) \\ E_\delta(i,j-1) + \delta(\epsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array}\right\} & \text{für } i > 0 \text{ und } j > 0 \\ E_\delta(i-2,j-2) + \delta'(u[i-1 \ldots i] \to v[j-1 \ldots j]) & \text{für } i > 1 \text{ und } j > 1 \end{array} \right.$$

wobei die Rekurrenzen für $E_\delta(0,0)$, $E_\delta(0,j)$ und $E_\delta(i,0)$ identisch mit der Version im Skript sind und nicht gesondert angegeben wurden.

Ist die Transposition die günstigste Operation an dieser Stelle, so wird eine minimierende Kante im Editgraph von $E_\delta(i-2, j-2)$ zu $E_\delta(i,j)$ angelegt, siehe Abbildung 1.

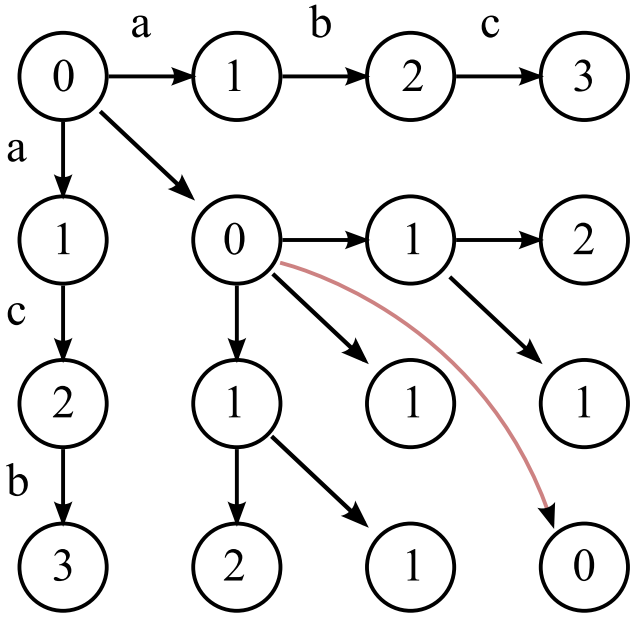

Abbildung 1: Beispielhafter Editgraph für $u = acb, c = abc$. Als Kostenfunktion gilt die Einheitskostenfunktion und $\delta'(cb \to bc) = 0$. Alle Pfeile markieren minimierende Kanten. Die rot ausgezeichnete Kante von $E_\delta(1,1)$ nach $E_\delta(3,3)$ entspricht der an dieser Stelle günstigsten Transpositionsoperation.