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

Grundlagen der Sequenzanalyse
Wintersemester 2016/2017
Übungen zur Vorlesung: Ausgabe am 20.12.2016

Punkteverteilung: Aufgabe 10.1: 3 Punke, Aufgabe 10.2: 2 Punkte, Aufgabe 10.3: 5 Punkte

Wir wünschen Ihnen allen frohe Festtage, geruhsame Ferien und einen guten Start ins Jahr 2017.

Abgabe bis zum 9.1.2017

Aufgabe 10.1 Implementieren Sie die rekursive Funktion *evaluatecrosspoints* des Linear-Space-Alignment-Algorithmus. Das entsprechende Programm soll die Tabelle C der Kreuzungspunkte als Array der Länge $n + 1$ berechnen und ausgeben. Sie können sich auf das Einheitskostenmodell beschränken und sollen die Funktion *evaluateallcolumns*, die Sie im Rahmen einer früheren Übungsaufgabe entwickelt haben, wiederverwenden.

Der Rückgabewert der Funktion *evaluatecrosspoints* soll die Edit-Distanz der beiden zu vergleichenden Sequenzen sein. Im Material zu dieser Aufgabe finden Sie in der Datei `sequences.txt` einige Sequenzen und jeweils die entsprechenden Kreuzungspunkte in einem einfachen textbasierten Format. Verwenden Sie das gleiche Format und verifizieren Sie, dass Ihr Programm die gleichen Kreuzungspunkte berechnet, indem Sie

```
./Verifycrosspoints.sh ./linear-space.x sequences.txt | diff - crosspoints.txt
```

aufrufen. Sie müssen ggf. noch das zweite Argument des Shell-Skriptes (den Pfad Ihres Programms) beim obigen Aufruf anpassen.

Aufgabe 10.2 Gegeben seien die Sequenzen $u = agtgcacacatc$ und $v = atcacacttagc$.

1. Berechnen Sie die links-rechts Zerlegung von u im Bezug auf v.
2. Berechnen Sie die rechts-links Zerlegung von u im Bezug auf v.
3. Berechnen Sie die links-rechts Zerlegung von v im Bezug auf u.
4. Berechnen Sie die rechts-links Zerlegung von v im Bezug auf u.

Aufgabe 10.3 Implementieren Sie eine Funktion, die für eine gegebene Sequenz der Länge n über einem Alphabet $\mathcal{A}$ der Größe r die in der Vorlesung behandelten Integer-Codierungen aller Teilworte der Länge q aufzählt. Stellen Sie dabei durch effiziente Implementierung sicher, dass dies in einer Laufzeit von $O(n + q + r)$ geschieht. Die einzelnen Buchstaben des Alphabets sollen dabei mit Hilfe einer alphabetspezifischen Tabelle `translate` in Zahlen zwischen 0 und $r - 1$ transformiert werden.

Implementieren Sie mit Hilfe dieser Funktion den in der Vorlesung vorgestellten Algorithmus zur Berechnung der q-Wort-Distanz für zwei Sequenzen. Das lauffähige Programm soll drei Argumente haben, nämlich die beiden Sequenzen, deren Distanz bestimmt werden soll, sowie den Wert von q. Das Alphabet ergibt sich als die Menge der Zeichen, die in den beiden Sequenzen vorkommen. Die Ausgabe Ihres Programms für zwei Sequenzen u und v sollte von der Form

`qgdist(u,v,q)=<qgram-distance of u and v>`

sein.

Im Material zu dieser Aufgabe finden ein Makefile, in dem Sie unter `myprogram=` den Pfad Ihres Programms eintragen können. Durch `make test` wird dann verifiziert, dass Ihr Programm die Ergebnisse aus der Referenzdatei `results.txt` liefert.

Bitte die Lösungen zu diesen Aufgaben bis zum 09.01.2017 abgeben. Die Besprechung der Lösungen erfolgt am 10.01.2017.

Lösung zu Aufgabe 10.1:

Zweiter Teil der Lösung. Der Rest des Codes findet sich auf einem vorherigen und einem folgenden Übungsblatt.

```
static unsigned long evaluatecrosspoints(const unsigned char *useq,
                                         const unsigned char *vseq,
                                         unsigned long ulen,
                                         unsigned long vlen,
                                         unsigned long *EDtabcolumn,
                                         unsigned long *Rtabcolumn,
                                         unsigned long *Ctab,
                                         unsigned long rowoffset)
{
  if (vlen >= 2)
  {
    unsigned long distance, midrow, midcol = vlen/2;

    distance = evaluateallcolumns(EDtabcolumn, Rtabcolumn, midcol, useq, vseq,
                                  ulen, vlen);
    midrow = Rtabcolumn[ulen];
    Ctab[midcol] = rowoffset + midrow;
    (void) evaluatecrosspoints(useq,
                               vseq,
                               midrow,
                               midcol,
                               EDtabcolumn,
                               Rtabcolumn,
                               Ctab,
                               rowoffset);
    (void) evaluatecrosspoints(useq + midrow,
                               vseq + midcol,
                               ulen - midrow,
                               vlen - midcol,
                               EDtabcolumn,
                               Rtabcolumn,
                               Ctab + midcol, /*! this is important to fill Ctab
                                                 correctly! */
                               rowoffset + midrow);
    return distance;
  }
  return 0; /* value not used, as only value at recursion level 0 is used. */
}

static void showcrosspoints(const unsigned long *Ctab,unsigned long vlen)
{
```

```
  unsigned long idx;

  for (idx = 0; idx <= vlen; idx++)
  {
    printf("%lu\n",Ctab[idx]);
  }
}

static void usage(const char *progname)
{
  fprintf(stderr, "%s: wrong number of arguments\n"
          "Usage: %s ali|crp  sequence_u sequence_v\n",progname,progname);
  exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
  char *useq = NULL,
       *vseq = NULL;
  bool withalignment = true;
  unsigned char *ali1 = NULL, *ali2 = NULL;
  unsigned long ulen, vlen,
                distance,
                maxalilen,
                alilen,
                *Ctab;

  if (argc != 4)
  {
    usage(argv[0]);
  }
  if (strcmp(argv[1],"ali") == 0)
  {
    withalignment = true;
  } else
  {
    if (strcmp(argv[1],"crp") == 0)
    {
      withalignment = false;
    } else
    {
      usage(argv[0]);
    }
  }
  useq = argv[2];
  vseq = argv[3];

  ulen = (unsigned long) strlen(useq);
  vlen = (unsigned long) strlen(vseq);

  printf("sequence u (of length m=%2lu): %s\n", ulen, useq);
  printf("sequence v (of length n=%2lu): %s\n", vlen, vseq);

  maxalilen = ulen + vlen;
  if (withalignment)
  {
    ali1 = malloc(maxalilen * sizeof *ali1);
```

```
    assert(ali1 != NULL);
    ali2 = malloc(maxalilen * sizeof *ali2);
    assert(ali2 != NULL);
  }
  Ctab = malloc((vlen+1) * sizeof *Ctab);
  distance = linearspace_alignment((unsigned char *) useq,
                                   (unsigned char *) vseq,
                                   ulen, vlen,
                                   Ctab,
                                   ali1, ali2,
                                   &alilen);

  printf("distance=%lu\n", distance);
  if (withalignment)
  {
    printf("alignment:\n");
    showalignment(ali1, ali2, alilen);
    free(ali1);
    free(ali2);
  } else
  {
    printf("crosspoints:\n");
    showcrosspoints(Ctab,vlen);
  }
  free(Ctab);
  return EXIT_SUCCESS;
}
```

Lösung zu Aufgabe 10.2: Sei $u = agtgcacacatc$ und $v = atcacacttagc$.

$\psi_{lr}(u, v) = (ag, t, gc, a, caca, t, c)$

$\psi_{rl}(u, v) = (a, g, t, g, caca, c, atc)$

$\psi_{lr}(v, u) = (atc, a, cac, t, t, a, gc)$

$\psi_{rl}(v, u) = (a, t, cacac, t, t, a, gc)$

Lösung zu Aufgabe 10.3:

```
class Qgram
  def initialize(q, alphabet)
    @q = q
    @alphasize = alphabet.length
    @alphasize_exp_q_minus_1 = @alphasize**(@q-1)
    @alphabet = alphabet
    @translate = Hash.new
    @translate_mult = Hash.new
    @num_of_possible_qgrams = @alphasize**@q
    alphabet.each_with_index do |char, i|
      @translate[char] = i
      @translate_mult[char] = i * @alphasize_exp_q_minus_1
    end
  end

  def encode(seq)
    qgram_code = 0
    1.upto(@q) do |idx|
      qgram_code += @translate[seq[idx-1]] * @alphasize**(@q-idx)
```

```
    end
    return qgram_code
  end

  def decode(qgram_code)
    char_array = Array.new(@q)
    @q.downto(1) do |i|
      char_code = qgram_code % @alphasize
      qgram_code /= @alphasize
      char_array[q - i] = @alphabet[char_code]
    end
    return char_array.join
  end

  def each_code(seq)
    current_code = self.encode(seq)
    yield current_code
    @q.upto(seq.length - 1) do |idx|
      current_code = (current_code -
                      @translate_mult[seq[idx-@q]]) * @alphasize +
                     @translate[seq[idx]]
      yield current_code
    end
  end

  def each_code_with_pos(seq)
    pos = 0
    self.each_code(seq) do |code|
      yield code, pos
      pos += 1
    end
  end

  def code_profile(seq)
    profile = Array.new(@num_of_possible_qgrams, 0)
    self.each_code(seq) do |code|
      profile[code] += 1
    end
    return profile
  end

  def distance(seq_a, seq_b)
    # enumerate qgrams and count them to get profiles
    profile_a = self.code_profile(seq_a)
    profile_b = self.code_profile(seq_b)
    # calculate distance from profiles
    dist = 0
    0.upto(@num_of_possible_qgrams - 1) do |code|
      dist += (profile_a[code] - profile_b[code]).abs
    end
    return dist
  end

end

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

```ruby
$:.unshift(File.join(File.dirname(__FILE__), "."))

require 'qgram'

if ARGV.length != 3
  STDERR.puts "Usage: #{$0} <seq1> <seq2> <q>"
  STDERR.puts "       (e.g. #{$0} ACGACGTAG GGACGTGCAG 3)"
  exit(1)
end
seq1 = ARGV[0]
seq2 = ARGV[1]
alpha = (seq1 + seq2).split(//).uniq
q = ARGV[2].to_i
qgram = Qgram.new(q, alpha)
# calculate and output q-gram distance
puts qgram.distance(seq1,seq2)
```