Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Grundlagen der Sequenzanalyse
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 10.01.2023

Aufgabe 10.1 (4 Punkte) In einer früheren Übungsaufgabe haben Sie ein Python-Programm zur Berechnung der Edit-Distanz zweier Sequenzen geschrieben. Dieses wurde um eine Repräsentation der eingehenden minimierenden Kanten für jeden Knoten im Edit-Graphen erweitert. Ebenso wurde eine Klasse `Alignment` zur Repräsentation von Alignments implementiert. Die Musterlösungen zu den genannten Aufgaben finden Sie in den Dateien `editgraph.py` und `aligntype.py`, die Sie in dieser Aufgabe wiederverwenden dürfen.

Benennen Sie die Datei `alignment_template.py` in `alignment.py` um. Implementieren Sie in dieser Datei eine Funktion

```
traceback(al, dpmatrix, i, j)
```

für deren Parameter folgendes gilt:

- `al` ist eine Referenz auf eine Instanz der Klasse `Alignment` (siehe `aligntype.py`). Fügen Sie die beim Traceback erkannten Edit-Operationen durch die passende Methode hinzu.
- `dpmatrix` ist eine Referenz auf die DP-Matrix mit der Information über die eingehenden minimierenden Kanten für jeden Knoten des Edit-Graphen. Diese wird durch die Funktion `fillDPtable_minedges` aus `editgraph.py` berechnet.
- `i` und `j` geben die Längen der Präfixe der Sequenzen u und v an, für die das Alignment noch konstruiert werden muss.

Die Funktion `traceback` soll ausgehend vom Knoten (i, j) im Editgraphen die minimierenden Pfade rückwärts verfolgen, bis der Knoten $(0, 0)$ erreicht wird. Dabei wird das optimale Alignment der alignierten Sequenzen rückwärts konstruiert.

Es soll genau ein optimales Alignment berechnet werden. Dabei gelten die folgenden Präferenzregeln: Falls es bei der Auswahl minimierender Kanten mehrere Möglichkeiten gibt, hat die Ersetzungskante die höchste Präferenz und die Löschkante hat die zweithöchste Präferenz.

Implementieren Sie nun eine Funktion `alignment(u,v)` die für zwei Strings `u` und `v` ein optimales Alignment dieser Strings (bzgl. der Einheitskostenfunktion und gemäß obiger Präferenzregeln) als Instanz der Klasse `Alignment` zurückliefert. Dazu nutzen Sie die Funktion `fillDPtable_minedges` und `traceback`. Verwenden Sie in `alignment(u,v)` die `evaluate`-Methode der Klasse `Alignment`, um mit einer `assert`-Anweisung zu überprüfen, dass die Kosten des Alignments, welches Ihre `traceback`-Funktion berechnet, mit der Edit-Distanz (abzulesen aus der Matrix) übereinstimmt.

Im Material zu dieser Übung finden Sie ein Modul mit Unittests, sowie ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm die Unittests besteht. Zur Fehlersuche für einzelne Sequenzpaare können Sie `alignment.py` auch im Terminal aufrufen. `./alignment.py -h` liefert Informationen zur Benutzung.

Punkteverteilung:

- strukturierte Implementierung von `traceback`: 2.5 Punkte

- Implementierung der Funktion `alignment`: 0.5 Punkte
- erfolgreiche Tests: 1 Punkt

Aufgabe 10.2 (4 Punkte) Implementieren Sie in einer Datei `affinealign.py` eine Klasse `AffineAlignment` zur Berechnung global optimaler Alignments bzgl. des affinen Gap-Kosten-Modells. Die Klasse hat die folgenden Instanz-Variablen und Methoden:

- Die Instanz-Variablen `self.useq` und `self.vseq` speichern die beiden zu alignierenden Sequenzen.
- Die Instanz-Variablen `self._ulen` und `self._vlen` speichern die Länge der beiden zu alignierenden Sequenzen.
- Die Instanz-Variable `self._dptable` speichert die DP-Matrix, die in der Vorlesung durch A_{afn} bezeichnet wurde. Diese Matrix hat $(m+1) \times (n+1)$ Einträge (für $m =$`self._ulen` und $n =$`self._vlen`) mit jeweils einer Instanz der Klasse `AffineDPentry`, die wie folgt definiert ist:

  ```
  class AffineDPentry:
    def __init__(self,rdist,ddist,idist):
      self.Rcost = rcost
      self.Dcost = dcost
      self.Icost = icost
  ```

  Die Instanz-Variablen `self.Rcost` enthält für den Eintrag in Zeile i und Spalte j der Matrix den Kostenwert $A_{\text{afn}}(i, j, R)$. Entsprechendes gilt für `self.Dcost` und `self.Icost`.
- Die Klasse `AffineAlignment` hat eine Methode

  ```
  __init__(self,useq,vseq,mismatch_cost,gapopen_cost,gapextend_cost)
  ```

  an die die zu alignierenden Sequenzen sowie die Kosten für einen Mismatch, für das Öffnen und für das Erweitern eines Gaps übergeben werden. Die Kosten eines Matches, d.h. einer Ersetzungsoperation $a \to a$ für alle $a \in \mathcal{A}$ sind 0. Die Methode `__init__` initialisiert die genannten Instanz-Variablen und die DP-Matrix `self._dptable` und berechnet dann die Werte entsprechend der Rekurrenz aus der Vorlesung. Dieser letzte Teil muss nun von Ihnen in der Datei `affinealign.py`, die Sie durch Umbenennen von `affinealign_template.py` erhalten, implementiert werden.
- Ebenso müssen Sie die Methode `affine_edit_distance(self)` implementiren. Diese liefert mit einer **`return`**-Anweisung die Edit Distanz (nach dem affinen Gap-Kosten-Modell) für die Sequenzen und Kosten, die in den genannten Instanzvariablen gespeichert sind.

Hinweis: In der Rekurrenz für die Matrix A_{afn} wird das Symbol ∞ verwendet. Im Python kann man diesen Wert durch **`float`**`('inf')` darstellen. Dieser Wert wird in der Instanz-Variablen `self._infinity` gespeichert.

In den Materialien finden Sie ein Programm `affinealign_mn.py`, das die Klasse `AffineAlignment` aus `affinealign.py` importiert und die Edit-Distanz für alle Paare von Sequenzen der Eingabe-Datei formatiert ausgibt. Optional wird auch die DP-Matrix ausgegeben. Durch `make test` verifizieren Sie, das Ihre Implementierung für verschiedene Testdaten das richtige Ergebnis berechnet. Anhang der Testdaten zum Sequenzpaar aus der Vorlesung und der entsprechenden DP-Matrix können Sie systematisch mögliche Fehler in Ihrem Programm finden (`make test_lect`). Sie

finden in den Testdaten auch optimale Alignments, die aber von Ihnen in dieser Aufgabe nicht berechnet werden sollen.

Aufgabe 10.3 (6 Punkte) In der Vorlesung wurde gezeigt, wie man auf der Basis von Stichproben der Menge aller q-grams von zwei Sequenzen ein Distanzmaß für diese Sequenzen berechnen kann. Eine Stichprobe, genannt „Sketch“ der Größe s, erhält man durch die Anwendung einer Hash-Funktion auf die q-grams und die Auswahl der s kleinsten Hash-Werte. In der Vorlesung wurde argumentiert, dass ein Sketch damit eine Stichprobe bildet, aus der sich Eigenschaften der Menge aller q-grams ableiten lassen. In dieser Aufgabe wollen wir dieses Argument empirisch untersuchen.

Wir vereinfachen das Konzept, indem wir direkt mit q-grams anstatt mit Ihren Hash-Werten arbeiten. Daher verwenden wir auch eine Notation, die sich leicht von der in der Vorlesung unterscheidet.

Sei $G_{q,s}(u)$ die Menge der s lexikographisch kleinsten q-grams der Sequenz u. $G_{q,\infty}(u)$ bezeichnet die Menge *aller* q-grams aus u. Seien nun u und v Sequenzen und

$$J_s(u,v) = \frac{|G_{q,s}(u) \cap G_{q,s}(v)|}{|G_{q,s}(u) \cup G_{q,s}(v)|}.$$

$J_s(u,v)$ ist damit eine Form von Jaccard-Index, der sich aus den jeweils s lexikographisch kleinsten q-grams der beiden Sequenzen ergibt. Für $s = \infty$ erhält man den Jaccard-Index.

Sei $S = \{250, 500, 750, 1\,000, 1\,250, 1\,500\}$ die Menge der Stichproben-Größen. Wir wollen nun für alle $q \in \{15, 16\}$, alle $s \in S$ und alle Paare (u, v) von 31 Sequenzen von Ebola-Genomen die Werte $J_s(u,v)$ von $J_\infty(u,v)$ miteinander vergleichen und die Pearson-Korrelation (siehe unten) bestimmen. Sie müssen das entsprechende Python-Programm nicht komplett selbst schreiben, sondern nur die drei folgenden Funktionen implementieren:

- die Funktion `unique_qgram_list_get(sequence,q)` berechnet die Liste aller q-grams im String `sequence` in lexikographischer Reihenfolge und ohne Duplikate. Diese Liste wird durch eine **`return`**-Anweisung zurückgeliefert. Zum Sortieren können Sie die Python-Funktion `sort()` verwenden. Falls `ls` eine Liste von Strings ist, dann liegen nach der Anweisung `ls.sort()` die Elemente von `ls` in lexikographisch sortierter Reihenfolge vor. Die Funktion sortiert also *in place*. Beispiel:

  ```
  >>> ls = ['cc','ac','aa','cc']
  >>> ls.sort()
  >>> print(ls)
  ['aa', 'ac', 'cc', 'cc']
  ```

  Dabei ist >>> das Prompt der interaktiven Python3-Shell. 1 Punkt

- die Funktion `common_get(l1,p1,l2,p2)` erhält als Argument zwei Listen `l1` und `l2` von lexikographisch sortierten q-grams ohne Duplikate sowie zwei Präfixlängen-Parameter `p1` und `p2`. Unter den ersten `p1` q-qgrams aus `l1` und den ersten `p2` q-qgrams aus `l2` wird in $O(q(p1 + p2))$ Zeit die Anzahl gemeinsamer q-qgrams bestimmt und mit einer **`return`**-Anweisung zurückgeliefert. Diese Laufzeit wird z.B. dadurch erreicht, dass man $O(p1 + p2)$ Vergleiche von q-grams durchführt, wobei jeder Vergleich in $O(q)$ erfolgt. 2 Punkte

- die Funktion `pearson_correlation(x, y)` liefert durch eine **`return`**-Anweisung die Pearson-

Korrelation für Listen `x` und `y` von reellen Zahlen gleicher Länge. Sei

$$x = [x_0, \ldots, x_{n-1}]$$
$$y = [y_0, \ldots, y_{n-1}]$$

Dann ist die Pearson-Korrelation definiert durch 1 Punkt

$$Pearsoncorr(x, y) = \frac{\sum_{i=0}^{n-1}(x_i - \text{mean}(x)) \cdot (y_i - \text{mean}(y))}{\sqrt{\sum_{i=0}^{n-1}(x_i - \text{mean}(x))^2} \cdot \sqrt{\sum_{i=0}^{n-1}(y_i - \text{mean}(y))^2}}$$

$$\text{wobei mean}(x) = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

$$\text{und mean}(y) = \frac{\sum_{i=0}^{n-1} y_i}{n}$$

Die ersten beiden Funktionen müssen in der Datei `unique_qgram_list.py` implementiert werden. Die letzte Funktion muß in der Datei `pearson_corr.py` implementiert werden.

In der Datei `jaccard.py` (siehe Material) finden Sie (1) ein Hauptprogramm, das die genannten Funktionen aufruft, (2) eine Datei `multiseq.py` mit einer Klasse `Multiseq` zur Verwaltung der Sequenzen sowie (3) eine komprimierte Datei `ebola-genomes.fna.gz` mit den Sequenzen der Ebola-Genome. Durch `make test` wird der Unit-Test für die von Ihnen implementierten Funktionen durchgeführt. Zudem wird für alle Paare von Ebola-Genomen der Jaccard Index für alle q-grams bzw. für eine Stichprobe der q-grams ausgegeben sowie Korrelationskoeffizienten bestimmt.

Dokumentation der Funktion

Neben der Implementierung besteht Ihre Aufgabe darin, den Programmcode aus `jaccard.py` zu lesen und jede einzelne Funktion zu dokumentieren und zwar vor dem Funktionskopf (siehe Kommentar im Programmtext). Es reichen jeweils 2-3 Zeilen (mit max 80 Zeichen), die den Zweck der Funktionen beschreiben. Wenn dieser nicht direkt aus dem Programmcode ersichtlich ist, können Sie auch temporär `print`-Anweisungen einfügen, um Zwischenergebnisse anzuzeigen.In `jaccard.py` werden mehrfach List-Comprehensions genutzt. Hierzu finden Sie Informationen im Skript zur Vorlesung PfN1, siehe `doc/python-slides.pdf`. 1.5 Pkte

Beschreibung der berechneten Korrelationen

Beschreiben Sie die Ergebnisse der beiden Tests `correlation_15` und `correlation_16`. 0.5 Pkte

Bitte die Lösungen zu diesen Aufgaben bis zum 15.01.2023 um 22:00 Uhr an gsa@zbh.uni-hamburg.de schicken.