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

# Grundlagen der Sequenzanalyse
# Wintersemester 2022/2023
# Übungen zur Vorlesung: Ausgabe am 20.12.2022

Hier sind die URLs bzw. die QRcodes zur Lehrevaluation:

Vorlesung:
`https://evasys-online.uni-hamburg.de/evasys/online.php?pswd=G1QMU`

Übung:
`https://evasys-online.uni-hamburg.de/evasys/online.php?pswd=C3FKL`

**Aufgabe 9.1** (5 Punkte) Laut Definition ist ein Alignment eine Folge von Editoperationen. In dieser Aufgabe geht es darum, Alignments speichereffizient durch Multi-Editoperationen (Multi-Eops) zu repräsentieren. Es gibt dabei wie gewohnt drei verschiedene Typen von Editoperationen. Zu jeder Editoperation gehört noch eine positive ganze Zahl $i$. Diese spezifiziert die $i$-malige Anwendung der entsprechenden Editoperation.

In der Liste der Multi-Eops gibt es niemals zwei direkt aufeinanderfolgenden Multi-Eops des gleichen Typs. Das folgende Alignment lässt sich durch die rechts gezeigte Liste und die beiden Sequenzen darstellen:

```
acgtaga--tatata-gat
|   |||  ||  | | |            [R 7,I 2,R 2,D 1,R 3,I 1,R 3]      (1)
agaaagaggta-agaggga
```

Dabei wird jede Editoperation durch ihren ersten Buchstaben abgekürzt (Deletion $\mapsto$ `D`, Insertion $\mapsto$ `I`, Replacement $\mapsto$ `R`). Beachten Sie, dass

```
[R 7,I 2,R 2,D 1,R 2,R 1,I 1,R 3]
```

keine korrekte Darstellung ist, da `R 2` und `R 1` direkt aufeinander folgen.

Eine Liste von Multi-Editoperationen kann man am einfachsten durch zwei gleich lange Listen repräsentieren, einer Liste mit den Zeichen für die Editoperation und eine Liste mit den Anzahlen. Zu einer Alignmentrepräsentation gehören noch die beiden alignierten Sequenzen.

Implementieren Sie nun in Python in einer Datei `aligntype.py` eine Klasse `Alignment` zur Speicherung eines Alignments, wie oben beschrieben. Die Studierenden, die noch nicht genau wissen, wie man in Python eine Klasse implementiert, sollten sich die entsprechenden Folien in der Datei `python-slides.pdf` ansehen (Abschnitt *Abstract date types and classes, frame 4-24*). Dieser Teil der Folien ist relativ unabhängig von den vorherigen Teilen und kann daher auch „vorgezogen“ werden.

Die Klasse `Alignment` implementiert mindestens die folgenden Methoden:

- `__init__(self,u,v)` speichert in entsprechenden Instanzvariablen die beiden Strings $u$ und $v$ sowie deren Länge. Außerdem werden zwei Instanzvariablen `_eops` und `_eopnumbers` jeweils als leere Liste initialisiert. Hierin werden die Multi-Editoperationen gespeichert.
- `add_replacement(self,nof_eops = 1)`, `add_deletion(self,nof_eops = 1)`, `add_insertion(self,nof_eops = 1)` fügt jeweils eine entsprechende Multi-Editoperation zur Alignment-Repräsentation durch den Instanz-Variablen hinzu. Dabei ist `nof_eops` die Anzahl die der Operationen, die im default-Fall 1 ist.
- `reverse(self)` kehrt die Reihenfolge der Elemente in den beiden genannten Listen um.
- `pretty_u_line(self)`: liefert den ersten String der drei Zeilen des Alignments, also den String, der nach Löschen der Gap-Symbole die Sequenz $u$ darstellt.
- `pretty_m_line(self)`: liefert den zweiten String der drei Zeilen des Alignments, also den String, der nur aus Leerzeichen und dem Zeichen `|` besteht, das für die Spalten mit identischen Zeichen verwendet wird.
- `pretty_v_line(self)`: liefert den dritten String der drei Zeilen des Alignments, also den String, der nach Löschen der Gap-Symbole die Sequenz $v$ darstellt.
- Die folgende Methode liefert die String-Repräsentation des Alignments. Für unser Beispiel ist das der String, der oben links in (1) dargestellt ist.

```
def __str__(self):
  return '\n'.join([self.pretty_u_line(),
                    self.pretty_m_line(),
                    self.pretty_v_line()])
```

- `evaluate(self,mismatch_cost=1, indel_cost=1)` liefert die Kosten eines Alignments für zwei als Parameter übergebene Kostenwerte:
  - `mismatch_cost` sind die Kosten eines einzelnen Mismatches (d.h. Replacement mit zwei verschiedenen Buchstaben)
  - `indel_cost` sind die Kosten eines Indels, also einer einzelnen Deletion oder einer einzelnen Insertion

  Die Kosten eines Matches (d.h. Replacement mit gleichen Buchstaben) sind immer 0 und müssen daher nicht angegeben werden.

In einigen der oben genannten Methoden wird der **default**-Wert 1 angegeben. Dieser Wert wird verwendet, wenn beim Aufruf der Methode am Ende der Parameterliste Werte nicht explizit angegeben werden.

Fügen Sie nun das folgende Programmfragment aus der Datei `aligntype_main.py` an das Ende Ihrer Datei `aligntype.py` und machen Sie `aligntype.py` ausführbar.

```
if __name__ == '__main__':
  # Beispielhafte Benutzung dieser Klasse
  u = 'acgtagatatatagat'
  v = 'agaaagaggtaagaggga'
  a = Alignment(u, v)
  a.add_replacement(7)
  a.add_insertion(2)
  a.add_replacement(2)
  a.add_deletion()
  a.add_replacement(3)
```

```
a.add_insertion()
a.add_replacement(3)
print(a)
print('Gesamtkosten: {}'.format(a.evaluate()))
```

In den Materialien finden Sie Testdaten und ein Makefile. Durch `make test_short` wird verifiziert, dass `./aligntype.py` die Darstellung des Alignments in (1) ausgibt. Durch `make test` werden insgesamt vier verschiedene Test nacheinander durchgeführt.

Punkteverteilung:

- 2 Punkte insgesamt für die Funktionen `__init__`, `add_replacement`, `add_deletion` und `add_insertion`.
- 0.5 Punkte für `reverse`
- 1 Punkt insgesamt für die drei `pretty_`-Funktionen
- 1 Punkt für `evaluate`
- 0.5 Punkte für den bestandenen Test.

**Aufgabe 9.2** (3 Punkte) Im Folgenden sei $\delta$ die Einheitskostenfunktion. Dann liefert $edist_\delta$ die sog. *Einheitskostendistanz* von zwei Sequenzen.

(1) Seien $u = acgacgtag$ und $v = ggacgtgcag$ zwei Sequenzen über dem DNA-Alphabet $\mathcal{A} = \{a, c, g, t\}$. Bestimmen Sie die $q$-Wort Distanz von $u$ und $v$ für $q = 2$ und $q = 3$.

(2) Bestimmen Sie $edist_\delta(u, v)$ und ein optimales Alignment von $u$ und $v$.

(3) Ersetzen Sie in $u$ und $v$ die Zeichen $a$ und $g$ jeweils durch $r$ und die Zeichen $c$ und $t$ jeweils durch $y$. Welche Werte erhalten Sie nun für die $q$-Wort Distanz (für $q = 2$ oder $q = 3$) und für die Einheitskostendistanz?

(4) Geben Sie ein Beispiel für zwei Sequenzen $u_1$ und $v_1$ und ein $q_1 \geq 2$ an mit $qgdist_{q_1}(u_1, v_1) > edist_\delta(u_1, v_1)$.

(5) Geben Sie ein Beispiel für zwei Sequenzen $u_2$ und $v_2$ und ein $q_2 \geq 2$ an mit $qgdist_{q_2}(u_2, v_2) < edist_\delta(u_2, v_2)$.

(6) Geben Sie ein Beispiel für zwei Sequenzen $u_3$ und $v_3$ und ein $q_3 \geq 2$ an mit $qgdist_{q_3}(u_3, v_3) = edist_\delta(u_3, v_3)$.

Hinweis: Zur Lösung dieser Aufgabe ist es nicht notwendig, ein Programm zu schreiben. Ggf. ist es sinnvoll vorhandene Software aus früheren Aufgaben zur Berechnung der Edit-Distanz zu nutzen.

Punkteverteilung: je 0.5 Punkte für die 6 Teilaufgaben.

**Aufgabe 9.3** (5 Punkte) Implementieren Sie in einer Datei `qgram.py` eine Python-Klasse `Qgram` mit den folgenden Methoden:

1. Die Methode `__init__(self, q, alphabet)` erhält als Parameter die positive ganze Zahl $q$ (die Länge der $q$-grams) und eine Liste `alphabet` der sortierten Zeichen des Alphabets ohne Duplikate. Aus diesen beiden Werte werden die für die Codierung von $q$-grams und die Dekodierung der entsprechenden Integercodes benötigten Werte, Dictionaries und Listen initialisiert, insbesondere:

- ein Dictionary `self._rank`, das für das $i$-te Zeichen $a$ in `alphabet` (Indizierung ab 0) das Schlüssel-/Wertpaar $a : i$ enthält.
- ein Dictionary `self._rank_weight_first`, das für das $i$-te Zeichen $a$ in `alphabet` (Indizierung ab 0) das Schlüssel-/Wertpaar $a : i \cdot r^{q-1}$ enthält. Dabei ist $r$ die Alphabetgröße.
- eine Liste `self._alphasize_exp_rev` der Länge $q$, dessen $i$-ter Wert (Indizierung ab 0) $r^{q-1-i}$ ist.
- eine Konstante `self._num_of_qgrams`, die den Wert $r^q$ enthält.

Selbstverständlich sollen diese Member-Variablen in den folgenden Methoden verwendet werden. 1 Punkt

2. Die Methode `encode(self,seq)` berechnet für einen String der Mindestlänge `q` mit Zeichen aus der Liste `alphabet` den Integercode des Präfixes der Länge `q` von `seq` und liefert diesen mit einer **`return`**-Anweisung zurück. 1 Punkt
3. Die Methode `each_code(self,seq)` ist ein Generator, der nacheinander mit `yield` die Integercodes der $q$-grams aus `seq` aufzählt. Das soll mit der in der Vorlesung vorgestellten inkrementellen Methode in linearer Zeit erfolgen. 1 Punkt
4. Die Methode `distance(self,seq1, seq2)` berechnet die $q$-gram Distanz der beiden Strings `seq1` und `seq2` nach dem in der Vorlesung vorgestellten Algorithmus und unter der Verwendung der Methode `each_code`. 2 Punkte

Im Material zu dieser Übung finden Sie ein Programm mit Unit-Tests und ein Makefile. Durch `make test` verifizieren Sie, dass Ihre Implementierung für die Testdaten die korrekten Ergebnisse liefert.

**Aufgabe 9.4** (5 Punkte) Dies ist eine Aufgabe außerhalb der Wertung, d.h. die Punkte zählen nicht bei der Bestimmung der Gesamtzahl der Punkte für die 50%-Schranke zur Erfüllung der Studienleistungen.

Zeigen Sie durch vollständige Induktion über $q$, dass für alle $q \geq 0$ und alle $u, v \in \mathcal{A}^q$ gilt:

$$\overline{u} = \overline{v} \Rightarrow u = v \tag{1}$$

Dabei bezeichnet $\overline{u}$ die Integer-Codierung eines $q$-grams $u \in \mathcal{A}^q$, siehe Vorlesung zum Thema *The q-gram sequence comparison model*.

Der Induktionsbeweis besteht aus drei Schritten.

**Induktionsanfang** Man zeigt, dass die obige Aussage (1) für $q = 0$ gilt.

**Induktionsannahme** Man nimmt an, dass Aussage (1) für ein beliebiges $q$ gilt.

**Induktionsschritt** Man zeigt unter Verwendung der Induktionsannahme, dass Aussage (1) für $q + 1$ gilt.

**Bitte die Lösungen zu diesen Aufgaben bis zum 08.01.2023 um 22:00 Uhr an gsa@zbh.uni-hamburg.de schicken.**