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

Genominformatik
Sommersemester 2017
Übungen zur Vorlesung: Ausgabe am 27.06.2017

Punkteverteilung: Aufgabe 6.1: 4 Punkte, Aufgabe 6.2: 3 Punkte, Aufgabe 6.4: 6 Punkte

Abgabe bis zum 6.7.2017, 23:59

Aufgabe 6.1 Implementieren Sie in C auf Basis der Software-Bibliothek `GtSuffixtree` (siehe Übungsaufgabe 4.3.) in der Datei `stree-mum.c` eine Funktion

```
int gt_stree_mum(const char *indexname,
                 GtUword minlength,
                 bool withsequence,
                 GtError *err);
```

die den auf Suffixbäumen basierten Algorithmus zur Berechnung von Maximum Unique Matches (MUMs) (siehe Vorlesungsskript) implementiert. Dabei ist `indexname` der Name des Suffixbaumindex, der genau zwei Sequenzen enthält und `minlength` die minimale Länge der auszugebenden MUMs. Falls der Parameter `withsequence` den Wert `true` hat, wird die Sequenz des MUMs jeweils mit ausgegeben. `err` ist das Fehlerobjekt, dass für die relevante Funktionen von `GtSuffixtree` benötigt wird, um Fehlermeldungen zu speichern. Siehe dazu auch die Hinweise in Übungsaufgabe 5.2. Beachten Sie, dass für ein MUM, bei dem die Sequenz mit der kleineren Position nicht an Position 0 beginnen die linken Kontexte a und b verglichen werden müssen. Es muß entweder $a \neq b$ sein oder a ist ein Sonderzeichen. Letzteres wird mit dem Makro `ISSPECIAL` aus `core/chardef.h` überprüft.

Das Hauptprogramm, das Sie in den Materialien zu diesem Übungsblatt finden (siehe `stree-mum-mn.c`) hat zwei Argumente, nämlich den Namen des Suffixbaumindex sowie die minimale Länge ℓ der MUMs. Für jeden MUM werden die Startpositionen in den beiden Sequenzen S und S' sowie die Länge tabulator-separiert in einer eigenen Zeile ausgegeben. Wenn zusätzlich noch die Option `-s` vor den beiden Argumenten des Programms angegeben ist, dann hat der Parameter `withsequence` den Wert `true`.

Damit Sie obiges Programm erfolgreich kompilieren und Testen können, müssen Sie die gleichen Schritte durchführen wie in Aufgabe 4.3..

Als Beispielsequenz zum Testen Ihres Programms verwenden Sie die Genome von *Ecoli K12* und *Ecoli O127 H6* in den Dateien `Ecoli_K12.fna` und `Ecoli_O127_H6.fna`, die Sie durch den Aufruf von `./download.sh 1 2` erhalten. Durch den Aufruf

```
./index.sh Ecoli_K12_O127 Ecoli_K12.fna Ecoli_O127_H6.fna
```

erhalten Sie den Suffixbaumindex `Ecoli_K12_O127` aus beiden Genomen. Die genannten Aufrufe finden Sie auch im `Makefile` beim Ziel `test-mum`. Die erwartete Ausgabe für die

Mindestlänge 400 finden Sie in der Datei `Ecoli_K12_O127_MUM.csv`. Durch Aufruf von `make test-mum` wird die Ausgabe Ihres Programms mit dieser Datei verglichen.

Bitte löschen Sie vor Abgabe der Lösungen die beiden Dateien mit den Genomen sowie den Index durch `rm -f *.fna Ecoli_K12_O127.*`.

Aufgabe 6.2 Ein lcp-Intervall $[\ell, r]$ mit lcp-Wert h ist ein lokales Maximum, wenn für alle q, $\ell + 1 \leq q \leq r$ die Eigenschaft $\mathsf{LCP}[q] = h$ gilt. Geben Sie in Form von Pseudocode einen Algorithmus an, der für eine gegebene LCP-Tabelle jedes lokalen Maximum und dessen LCP-Wert ausgibt. Sie können nicht davon ausgehen, dass die lcp-Intervalle bereits bestimmt sind. Für eine LCP-Tabelle einer Sequenz der Länge n soll die Laufzeit Ihres Algorithmus $O(n)$ sein. Verifizieren und dokumentieren Sie, dass Ihr Algorithmus für die beiden Sequenzen $taaaaga\$$ und $ccttcgt\#ctgtcgt\$$ die korrekten Ergebnisse liefert. Hier sind die beiden Suffix Arrays sowie die lokalen Maxima:

i	SUF	LCP
0	1	
1	2	3
2	3	2
3	4	1
4	6	1
5	5	0
6	0	0
7	7	0

lok. Max.	LCP-Wert
[0,1]	3

i	SUF	LCP
0	0	
1	4	1
2	12	3
3	8	1
4	1	2
5	10	0
6	5	2
7	13	2
8	3	0
9	11	4
10	9	1
11	2	1
12	6	1
13	14	1
14	7	0
15	15	0

lok. Max.	LCP-Wert
[1,2]	3
[3,4]	2
[5,7]	2
[8,9]	4

Aufgabe 6.3 Implementieren Sie den Greedy-Algorithmus zur Fragmentassemblierung aus der Vorlesung und zwar so, dass auch die assemblierte Sequenz aus dem Hamilton-Pfad rekonstruiert wird. Zum Sortieren der Fragmente (absteigend nach dem Gewicht, bei gleichem Gewicht, aufsteigend nach Nummer des Start-Fragmentes) können Sie eine Standard-Sortierfunktion verwenden. Eine C- und eine Ruby-Implementierung der *Union-Find*-Datenstruktur und eines Fasta-Iterators finden Sie in der Materialien.

Beachten Sie in Ihrer Implementierung, dass im Überlappungs-Graphen eventuell implizite Kanten mit dem Gewicht 0 berücksichtigt werden müssen.

Verifizieren Sie die Korrektheit Ihrer Implementierung durch folgende Tests:

- Überprüfen Sie, dass alle Fragmente Substrings der assemblierten Sequenzen sind. Falls Sie Ruby verwenden, können Sie hierfür den Operator `match` (möglicherweise mit dem

Modifikator `i`) verwenden. Entsprechendes gilt für andere Programmiersprachen.

- Überprüfen Sie, dass die assemblierte Sequenz der Länge ℓ ein Substring der Original-Genomsequenz der Länge g ist und falls ja, berechnen Sie $100 \cdot \frac{\ell}{g}$ und geben den Wert entsprechend aus, siehe unten. Die Originalsequenz ist natürlich in der Realität nicht bekannt, aber soll hier zu Testzwecken zur Verfügung stehen.

Falls einer der beiden Tests scheitert, soll Ihr Programm mit einer sinnvollen Fehlermeldung abbrechen.

Zum Testen Ihres Programm nutzen Sie die Materialien zu dieser Übungsaufgabe. Darin finden Sie die folgenden Dateien:

- `phage-lambda.fna` ist eine Fasta-Datei mit der Genomsequenz der Phage λ (Genbank Zugriffsnummer JO2459, Länge 48102 bp).
- `lambda-reads.fna` ist eine Multi-Fasta-Datei mit der Menge $\mathcal{F}$ von 667 Fragmenten, jeweils der Länge 650. Jedes Fragment ist ein exakter Substring der Genomsequenz der Phage λ. Diese Fragmentmenge entspricht einer 9-fachen Überdeckung des Genoms.
- Die Kanten des Überlappungsgraphs $\mathcal{OG}(\mathcal{F})$ für eine Fragmentmenge $\mathcal{F}$ der Grösse $k = 667$ sind in der Datei `lambda-overlaps.txt` angegeben. Darin definiert jede Zeile der Form

  $i + j + t$

  eine Kante $f \xrightarrow{t} f$ mit $t \geq 20$, wobei $0 \leq i \leq k-1$ die Nummer des Fragments f und $0 \leq j \leq k-1$ die Nummer des Fragments f' ist.

Wenn Ihr Programm korrekt funktioniert, sollte die Ausgabe für diese Daten wie folgt aussehen:

```
verified assembly (47862 bp) vs. 667 fragments of avg length 650.0
all 667 fragments occur in assembled string
coverage of genome by fragments: 9.01X
verified assembly (47862 bp) vs. genome (48102 bp)
assembly is substring of genome sequence, coverage=99.50% of genome
```

Die Lösungen zu diesen Aufgaben werden am 11.07.2017 besprochen.