PROJEKT 2: SEKUNDÄRSTRUKTUREN VON RNAS

Dang Quynh Tram Nguyen
Studiengang: Bioinformatik
Matrikelnummer: 5311561

Inhaltsverzeichnis

1. Einleitung

RNA ist eine einsträngige Kette, die aus vielen Nukleotiden besteht. RNA spielt eine wichtige Rolle in Zellen der Lebewesen. Es gibt verschiedene Arten von RNA und jede hat die bestimmten Funktionen in Zellen.

In diesem Projekt geht es um eine Variante von RNA, nämlich MicroRNA (Abkürzung: miRNA). miRNA kann mit sich selbst eine Sekundärstruktur bilden, die aus Stems besteht.

2. MicroRNA

MiRNAs sind die kleinen nicht-codierenden RNA. MiRNAs sind wichtig in vielen grundlegenden biologischen Prozesse wie:

- Regulation von Proteinsynthese
- Zellproliferation und -differenzierung
- Steuerung der Apoptose

MiRNA kann mit sich selbst eine Sekundärstruktur bzw. Stems-Struktur bilden. Die sogenannten Stems sind jeweils zwei zueinander komplementären Abschnitte auf derselben miRNA. Sie sind ähnlich wie die Doppelhelix-Struktur von DNA aber nutzt die Bausteine von RNA.

Um diesen Abschnitten mit der Rechnersprache zu bestimmen, kommt es zu einer Idee: Der reverse Komplement von gegebener RNA wird erzeugt und danach mit RNA verglichen. Die Position des Übereinstimmungsabschnittes auf dem reversen Komplement gibt später seine entsprechende Position auf RNA zurück.

Zuerst wird das File, das miRNA enthält, gelesen und nur die Sequenz als ein String gespeichert.

```
ff = open('miRNA.fasta','r')
fasta_file = ff.readlines()
mi_rna = fasta_file[1]
```

Mit der Funktion *Reverse_komplement* wird eine komplementäre Sequenz anhand des miRNA erzeugt und zu dem reversen Komplement umgekehrt.

```python
def Reverse_komplement(rna_sequenz):
    kom_sequenz = str()

    for nu in rna_sequenz:
        if nu == "A": kom_sequenz += "U"
        elif nu == "U": kom_sequenz += "A"
        elif nu == "G": kom_sequenz += "C"
        elif nu == "C": kom_sequenz += "G"
        else: #Falls die Sequenz nicht passende Base enthält
            print(nu, "ist kein Nukleotid für RNA")
            break

    #die komplementäre Sequenz wird umgekehrt
    rev_kom_seq = kom_sequenz[::-1]

    return rev_kom_seq
```

```python
rev_kom_sequenz = Reverse_komplement(mi_rna)
print(mi_rna)
print(rev_kom_sequenz)
```

```
GGAGCUUAUCAGAAUCUCCAGGGGUACUUUAUAAUUUCAAAAAGUCCCCCAGGUGUGAUUCUGAUUUGCUU
C
GAAGCAAAUCAGAAUCACACCUGGGGGACUUUUUGAAAUUAUAAAGUACCCCUGGAGAUUCUGAUAAGCUC
C
```

3. Maximale Repeats

Im nächsten Schritt kommt der Vergleich zweier RNA-Sequenzen, um die Stems zu bestimmen. Man kann in diesem Fall den Algorithmus für die Suche nach maximale Repeats anwenden. Ein maximales Repeat bezeichnet ein Paar zweier zueinander entsprechenden Substrings auf einer Sequenz und wird durch seine Indizes bestimmt. Die Substrings sollten möglichst maximale Länge erreichen. Die Bedeutung von maximalen Repeats passt der Definition vom Stem und dem Vergleich von RNA und ihrer reversen Komplement nicht genau, trotzdem gibt der Algorithmus von maximalen Repeats in diesem Fall doch das gewünschte Ergebnis zurück.

Der ganze Algorithmus beschäftigt vielmals mit Suchen und Vergleichen, deswegen ist die Suchmethode mit Suffix Array gewählt, die gute Laufzeit von *log(n)* hat. Durch diesen Algorithmus werden aber nur die möglichen Suffixe, was das Pattern enthält können, zurückgegeben. Dann braucht man noch eine *Match*-Methode zur Sicherheit, dass das Pattern wirklich in den Suffixen ist. Die Patterns sind alle Suffixe des reversen Komplements gemeint und sie werden jeweils mit Suffix-Array-Suchmethode in Suffix-Liste von RNA gesucht.

Nach dem Suchen ergibt eine Liste von alle gefundene ‚Repeats' bzw. Stems, was jeweils unten einer Liste mit den Positionen zweien zueinander komplementären Abschnitten auf RNA und der Länge der Abschnitte gespeichert werden. Das Ergebnis sieht aber nicht gut aus, weil es noch die nicht-maximale Repeats enthält. Da werden sie aussortiert und am Ende ergibt es nur eine Liste von maximalen Repeats.

Die gesamte Idee wird durch das folgende Pseudocode genauer beschreibt.

D ist ein Dictionary von Suffixen, seine Werte ist die Position auf RNA.

L ist ein sortiertes ArrayList von Suffixen.

Algorithm Suffix (s [1…n], D, L[1…n])

for i <- 1…n

suffix <- s[i…n]

L.add(suffix)

```
            D[suffix] = i
    L.sort()

Algorithm SASearch(p[1…m],L[1…n])
    l <- 1
    r <- n
    while (r – l > 1)
            m = (l+r)/2
            if p < L[m]:
                    r = m
            else:
                    l = m
    return r

Algorithm Match(p[1…m],s[1…n])
    m = 0
    if m <= n:
            for i <- 1…m
                    if p[i] != s[i]:
                            m = i-1
                            break
                    else:
                            m = i
    else:
            for i <- 1…n
                    if p[i] != s[i]:
                            m = i-1
                            break
                    else:
                            m = i
    return m
```

Algorithm RepeatOfAPattern(p[1…m],L[1…n])

D1 ist Suffix-Dictionary von RNA, D2 ist Suffix-Dictionary von Komplement

R ist Liste von allen Repeats. R1 ist temporale Liste von allen Repeats. R1 ist eine Kopie von R

Re ist Liste von einem Repeat

pos <- SASearch(p[1…m],L[1…n])

len <- Match(p[1…m],R1[pos])

while len > 3:

Speichert in Re die ersten Positionen von zwei Abschnitten des Repeats und die Länge

R.add(Re)

R1.remove(R1[pos])

pos <- SASearch(p[1…m],R1[1…n-1])

len <- Match(p[1…m],R1[pos])

Diese sind vier wichtige Algorithmen für die Suche der Stems auf RNA. Da hat jeder die Laufzeit:

Suffix: O(n)

SASearch: O(log n)

Match: O(n) / O(m)

RepeatOfAPattern: O(k.n) ,k ist beliebig

Es gibt insgesamt m Suffixe von Komplement, was als Pattern sind. Das heißt, dass m-mal *RepeatOfAPattern* durchgeführt werden.

Das ergibt die Laufzeit O(k.n.m).

Dann kommen noch die Algorithmen zum Filtern, die nach jedem Aufruf die Laufzeit O(a) hat. Dabei ist a die Anzahl der gefundenen Repeats. Es gibt insgesamt sechs-mal Algorithmen zum Filtern. Das ergibt die Laufzeit O(6a) bzw. O(a).

Fazit: Die gesamte Laufzeit ist O(k.n.m + a) oder O(k.n.m)

Der genauer Python-Code und die Durchführung sowie die Ergebnisse werden im Quellecode gezeigt.

4. RNA Sekundärstrukturen

Es gibt jetzt eine Liste von alle maximale Repeats, die Stem-Struktur bzw. miRNA-Sekundärstruktur bilden können. Darin gibt noch die überlappten Stems, die entfernt werden müssen.

```
#length = stop - start + 1 (bp)
#gap = start2 - stop1 - 1

#Dictionary des Abstands jedes maximalen Repeatpaares in RNA
list_gap_repeats = []

for repeat in list_max_repeats_filter:

    gap_repeat = repeat[1] - repeat[0] - 1
    list_gap_repeats.append(gap_repeat)

print(list_gap_repeats)
print(len(list_gap_repeats))
```

Ein Vorschlag für die Stem-Struktur auf der RNAfold Webserver wird gezeigt. Diese Struktur enthält 3 Stems, die auch vorhanden in der oben gefundenen Liste sind. Durch das Code kann man die Abstand zwischen zweier Abschnitten des Stems bestimmen. Es ist klar zu sein, dass die Webseite 3 Stems, die gute Abstände zueinander haben.

5. Abschätzung mit längeren Sequenzen