PROJEKT 1: PATTERNMATCHING

Dang Quynh Tram Nguyen
Studiengang: Bioinformatik
Matrikelnummer: 5311561

Inhaltverzeichnis

1. Einleitung

In Molekularbiologie beschäftigt man sich regelmäßig mit dem zentralen Dogma, das im Allgemeinen den Erbinformationsfluss vom DNA zu Proteinen besagt. Um das genauer zu verstehen werden 3 Hauptsynthesen dabei untersucht: Synthese von DNA zu DNA (Replikation), von DNA zu RNA (Transkription) und von RNA zu Proteinen (Translation). Bevor solche biologischen Prozesse starten, müssen bestimmten DNA- oder RNA-Abschnitte erkannt werden, an den bspw. die synthesefunktionellen Enzyme, Faktoren, usw. binden können. Die Frage wird gestellt, wie diese Abschnitte auf einem langen Erbinformationsstrang gefunden werden. Da folgt das Thema dieses Projekts: „Mustererkennung“, welche in diesem Fall eine Methode bzw. ein Algorithmus ist, einen bestimmten Pattern auf einer Sequenz zu suchen.

In diesem Projekt wird zunächst der Ablauf zweier Algorithmen „SimpleSearch“ und „Horspool“ mit Hilfe der Programmiersprache Python grundsätzlich dargestellt. Danach folgt die Anwendung eines der zwei Algorithmen auf einer großen Datei als Beispiel. Demnächst werden die zwei Algorithmen verglichen. Und zum Schluss kommt eine Zusammenfassung dieses ganzen Projekts.

2. Algorithmen

1. SimpleSearch Algorithmus

Wie oben erwähnt, geht es in diesem Algorithmus darum, Pattern auf einem bekannten Sequenz herauszufinden. Deutlicher ergibt er die Anzahl aller Treffer und die genauen Positionen, wo sie auf Sequenz liegen. Im ersten Blick kann man einen einfachen Modell der Methode vorstellen, wobei das Pattern auf jeder Position auf der Sequenz in einer Richtung bis zum Ende verschoben und mit dem auf der gleichen Position liegenden Teilsequenz verglichen wird. Das heißt, dass der Algorithmus alle Fälle versucht und am Ende die möglichste Anzahl der Hits zurückgibt. Dies erklärt den „SimpleSearch“ Algorithmus.

Demnächst wird die Implementierung dieses Algorithmus in Detail beschrieben. Die Funktion SimpleSearch nutzt eine for-Schleife für die zu vergleichende Position pos, die in einer bestimmten Grenze liegt, damit Teilsequenz teilstring die Länge der Sequenz nicht überschreitet. Wir fangen an der ersten Postion der Sequenz an. Da wird die Funktion Match_SS angewendet, wobei es geprüft wird, ob die Charakter in derselben Stelle der Teilsequenz und des Patterns gleich sind. Wenn sie miteinander übereinstimmen (Match_SS ergibt True), wird die Position in der vordeklarierten Liste match gespeichert und die Anzahl der Vergleiche count werden gezählt. Danach wird die Position pos auf eine verschoben, an dieser Position nimmt die Teilsequenz einen neuen Wert und wird mit dem Pattern wieder verglichen. Der Algorithmus läuft weiter wieder so, bis die letzte Position der Teilsequenz dieselbe der Sequenz ist. Am Ende der Funktion wird die Liste der Treffer zurückgegeben.

```
count = 0 #Anzahl der Vergleiche

def SimpleSearch(pattern,string):
    m = len(pattern)
    n = len(string)
    match = list()

    for pos in range(n-m+1):
        teilstring = string[pos:pos+m]
        #pos+m statt pos+m-1, weil die letzte Postion nicht einschließlich ist.

        if Match_SS(pattern,teilstring): #Wenn diese Funktion abgeruft wird, wird count trotz false verändert
            match.append(pos)

    return match

def Match_SS(pattern,teilstring):
    m = len(pattern)
    global count #Die globale Variable count wird innerhalb der Funktion verändert
    for i in range(m):
        count += 1
        if (pattern[i] != teilstring[i]):
            return False
    return True
```

Im Folgenden ist ein Beispiel dazu. Seien Pattern p als „tram“ und Sequenz s als „dangquynhtramnguyen“. Wenden wir das implementierten Skript an, bekommen wir folgendes Erbgebnis:

```
p = "tram"
s = "dangquynhtramnguyen"
pattern = list(p)
sequence = list(s)
```

```
match = SimpleSearch(pattern,sequence)
treffer = len(match)
print("Es gibt",treffer,"Treffer an der/den Position(-en)",match)
print("Insgesamt braucht der Algorithmus",count,"Vergleiche")
```

```
Es gibt 1 Treffer an der/den Position(-en) [9]
Insgesamt braucht der Algorithmus 19 Vergleiche
```

Der Pattern und die Sequenz werden zu Listen umgewandelt, dann hat jeder Charakter ein Index und diese werden in der Funktion Match_SS geprüft. Die Trefferanzahl treffer entspricht die Länge der Liste der Positionen.

2. Horspool Algorithmus

Die Laufzeit des “SimpleSearch” Algorithmus mit der großen Sequenz ist aber ziemlich lang. Deswegen wird dieser Algorithmus später optimiert, damit sein Ablauf vielmals beschleunigt wird. Es ist dann als “Horspool” Algorithmus genannt. Bei diesem Algorithmus wird die zu vergleichende Position auf der Sequenz, wo die Teilsequenz gefunden ist, auf bestimmten Stellen verschoben, anstatt auf allen Stellen wie in “SimpleSearch” Algorithmus. Die Schritten zum Vergleich bleiben behalten.

Zunächst beschreibe ich, wie dieser Algorithmus implementiert wird. Um die Verschiebung der zu vergleichenden Position zu verdeutlichen, wird eine Shift-Tabelle (Verschiebung-Tabelle) unten einem Dictionary shift_table erzeugt. Die Schlüssel von diesem Dictionary sind alle Charakter im angegebenen Pattern, außer dem letzten. Ihre entsprechenden Werte betragen die Differenz der Länge des Patterns und seine Position im Pattern. Welcher Charakter (als Schlüssel), der vielmals im Pattern getroffen ist, erhält als Wert das Ergebnis von der Subtraktion zwischen den Patternlänge und die Position im Pattern, die zuletzt gefunden wird.Wenn der Algorithmus läuft, beginnt die zu vergleichende Position an der ersten Position der Sequenz. Nach jedem Vergleich wird die Position auf den Betrag, was das Wert des letzten Charakters der Teilsequenz in der Shift-Tabelle entspricht, verschoben. Ist der letzter Charakter kein Schlüssel in der Shift-Tabelle, nimmt der Betrag die Länge des Pattern.

```
count = 0 #Anzahl der Vergleiche

def Horspool(pattern,sequence):
    shift_table = dict()
    m = len(pattern)
    n = len(sequence)
    match_position = list() #list alle Positionen des Patterns in der Sequenz (äq. einer Zeile des Textes)

    #bildet die Shift-Tabelle mit nur Charaktere (Buchstaben) im Pattern, außer den letzten
    for i in range(m-1): #nimmt alle Indexe außer den letzten
        shift_table[pattern[i]] = m-i-1 #weil i von 0...m-2 ist

    pos = 0
    while pos < n-m+1:
        teilstring = sequence[pos:pos+m]
        #alle Stellen von pos bis pos+m-1, weil den letzte Index pos+m ungültig ist.
        if(Match_H(pattern,teilstring)):
            match_position.append(pos)

        if(teilstring[m-1]) in shift_table.keys():
            pos += shift_table[teilstring[m-1]]
        else: pos += m

    return match_position


def Match_H(pattern,teilstring):
    m = len(pattern)
    global count
    for i in range(m-1,-1,-1): #i von m...1. Die Vergleichrichtung startet von der letzten Buchstabe
        count += 1
        if(pattern[i] != teilstring[i]):
            return False
    return True
```

Die Funktion Match_H läuft ähnlich wie die Match_SS vom oberen Algorithmus. Der Laufrichtung des Vergleichs startet aber an dem letzten Charakter und wird rückwärts fortgesetzt. Bei der Funktion werden die getätigten Vergleiche auch gezählt.

Ein Beispiel der Durchführung des Algorithmus wird (mit denselben Angaben wie “SimpleSearch” Algorithmus) wie folgt gegeben:

```
match = Horspool(pattern,sequence)
treffer = len(match)
print("Es gibt",treffer,"Treffer an der/den Position(-en)",match)
print("Insgesamt braucht der Algorithmus",count,"Vergleiche")
```

```
Es gibt 1 Treffer an der/den Position(-en) [9]
Insgesamt braucht der Algorithmus 8 Vergleiche
```

3. Beispiele

2 folgende Beispiele erklären genauer, wie die Algorithmen bei den großen Dateien abläuft.

1. Text „Schneewittchen“

Bei dem ersten Beispiel wird das Pattern „zwerg“ im Text „Schneewittchen“ gefunden und am Ende des Algorithmus sollten alle Positionen sowie die Anzahl der Treffer ergeben werden.

Zuerst wird der Text geöffnet und eine Liste, in der die Werte alle Zeilen im Text enthalten sind, wird erzeugt. Jeder dieser Werte ist eine Liste der Charakter derselben Zeile. Diese Charakterlisten gelten in den Algorithmen als die Sequenzen,worauf der angegebene Pattern gesucht wird.

```
file = open("Schneewittchen_(1850).txt","r")
file_read = list()
file_read_sequence = list()
for x in file:
    file_read.append(x)
    file_read_sequence.append(list(x.casefold()))
```

Danach werden alle Charakter in jeder Liste geprüft, damit sich alle besondere Charakter wie Umlaut oder die Anführungszeichen in einer Python passenden Format umsetzen lassen. Dieser Schritt ist in der Funktion aendert_umlaut_und_zeichen implementiert.

```
def aendert_umlaut_und_zeichen(file_read):
    for l in file_read:
        for letter in l:
            if letter in ["ä","ö","ü"]:
                index = l.index(letter)
                l.pop(index)
                if letter == "ä": l.insert(index,"a")
                if letter == "o": l.insert(index,"o")
                if letter == "ü": l.insert(index,"u")
                l.insert(index+1,"e")
            if letter in [",,","“"]:
                index = l.index(letter)
                l.pop(index)
                l.insert(index,"\"")
```

```
aendert_umlaut_und_zeichen(file_read_sequence)
```

Weil jede Zeile als eine Sequenz gilt und die aufgerufene Funktion des Algorithmus endlich nur die Liste der gefundenen Positionen eine Zeile zurückgibt, ist ein Dictionary benötigt, um alle Listen zu sammeln. Dies wird in der Funktion dict_positionen beschreibt. Die Zeilen, wo die Treffer liegen, sind die Schlüssel vom Dictionary und die Listen der genauen Position sind die Werte. Endlich wird der Dictionary zurückgegeben.

Für die Liste match_position passt entweder die Funktion SimpleSearch oder Horspool. Hier wird Horspool benutzt.

```
def dict_positionen(text,pattern):
    dict_position_pattern = dict()
    treffer = 0

    for line in text:
        match_position = Horspool(pattern,line)
        treffer += len(match_position)
        if(len(match_position) > 0):
            dict_position_pattern[text.index(line)] = match_position

    print(treffer,"Treffer sind im Text gefunden")
    return dict_position_pattern
```

Der Pattern „zwerg“ wird im Text gefunden. Die Positionen, die Zeilen der Treffer und die Anzahl der Vergleiche werden ausgegeben. Das Ergebnis wird im Quellcode angezeigt.

```
pattern1 = list("zwerg")
count = 0
print("Pattern \"zwerg\":")
dict_positionen_zwerg = dict_positionen(file_read_sequence,pattern1)

for line in dict_positionen_zwerg.keys():
    #Position des Pattern (in welcher Zeile und wo genau)
    if len(dict_positionen_zwerg[line]) == 1:
        print("in der Zeile",line,"an der Position",dict_positionen_zwerg[line])
    else:
        print("in der Zeile",line,"an den Positionen",dict_positionen_zwerg[line])
    #Zeile, die das Pattern enthält
    print(file_read[line])

vergleiche_zwerg = count
print("Insgesamt sind",vergleiche_zwerg,"Vergleiche benötigt")
```

Es gibt noch ein Problem, dass alle Vorkommen von „zwerg“ im Text nicht zum Wort „zwerge“ gehören.

Die Idee dafür ist, dass alle Positionen von „zwerge“ im Dictionary von Treffer „zwerg“ gelöscht werden. Dafür wird ein Dictionary von Treffer „zwerge“, nämlich dict_positionen_zwerge auch erzeugt.

```
pattern2 = "zwerge"
count = 0

print("Pattern \"zwerge\":")
dict_positionen_zwerge = dict_positionen(file_read_sequence,pattern2)
vergleiche_zwerge = count
print("Insgesamt sind",vergleiche_zwerge,"Vergleiche benötigt")

dict_pos_zwerg_not_in_zwerge = dict()
treffer = 0
for line in dict_positionen_zwerg.keys():
    list_pos_zwerg_not_in_zwerge = dict_positionen_zwerg[line].copy()

    #Prüft, ob der Schlüssel line vom "zwerg"-Dictionary auch den vom "zwerge"-Dictionary ist
    if line in dict_positionen_zwerge.keys():
        #Elimination der Positionen von "zwerge" in der "zwerg"-Liste
        for pos in dict_positionen_zwerge[line]:
            list_pos_zwerg_not_in_zwerge.remove(pos)

    #Prüft die noch gültige Positionen. Wenn es nichts ergibt, wird die Zeile vernachlässigt
    if len(list_pos_zwerg_not_in_zwerge) > 0:
        dict_pos_zwerg_not_in_zwerge[line] = list_pos_zwerg_not_in_zwerge
        treffer += len(list_pos_zwerg_not_in_zwerge)

print(treffer,"Treffer \"zwerg\", die nicht zum Wort \"zwerge\" gehören, im Text gefunden")
for line in dict_pos_zwerg_not_in_zwerge.keys():
    #Position des Pattern (in welcher Zeile und wo genau)
    if len(dict_pos_zwerg_not_in_zwerge[line]) == 1:
        print("in der Zeile",line,"an der Position",dict_pos_zwerg_not_in_zwerge[line])
    else:
        print("in der Zeile",line,"an den Positionen",dict_pos_zwerg_not_in_zwerge[line])
```

Ergebnis:

```
Pattern "zwerge":
17 Treffer sind im Text gefunden
Insgesamt sind 4230 Vergleiche benötigt
5 Treffer "zwerg", die nicht zum Wort "zwerge" gehören, im Text gefunden
in der Zeile 71 an der Position [1362]
in der Zeile 73 an der Position [976]
in der Zeile 119 an der Position [1221]
in der Zeile 157 an der Position [4]
in der Zeile 159 an der Position [791]
```

2. Text „ABAB“

Das zweiter Beispiel zeigt die Suche nach 2 Pattern „ABA“ und „CCC“ im Text „ABAB“. Der Hauptpunkt in diesem Beispiel ist die gesamte Anzahl der Vergleiche in jedem Algorithmus.

Der Text wird zuerst geöffnet und eine Liste des Textes wie im ersten Beispiel mit dem Text „Schneewittchen“ erzeugt. Wichtig ist, dass alle Zeilenumbrüche „\n“ in der Liste entfernt werden, damit mehrere Vergleiche und Treffer gefunden werden. Damit kann man zwei Algorithmen vergleichen.

```
file2 = open("ABAB.txt","r")
sequence = list()

for line in file2:
    l = list(line)
    for w in l:
        if(w == "\n"): l.remove(w)
    sequence.extend(l)
```

Mit dem Pattern „ABA“:

```
pattern1 = list("ABA")
print("Mit Pattern \"ABA\"")

count = 0
matchSS_aba = SimpleSearch(pattern1,sequence)
print("sind",count,"Vergleiche benötigt beim SimpleSearch Algorithmus")

count = 0
matchH_aba = Horspool(pattern1,sequence)
print("sind",count,"Vergleiche benötigt beim Horspool Algorithmus")

#Prüfen ob die Ergebnisse von 2 Methoden ähnlich sind
if matchSS_aba == matchH_aba:
    print("Insgesamt sind",len(matchH_aba),"Treffer gefunden")
```

```
Mit Pattern "ABA"
sind 7196 Vergleiche benötigt beim SimpleSearch Algorithmus
sind 5397 Vergleiche benötigt beim Horspool Algorithmus
Insgesamt sind 1799 Treffer gefunden
```

Mit dem Pattern „CCC“:

```
pattern2 = list("CCC")
print("Mit Pattern \"CCC\"")

count = 0
matchSS_ccc = SimpleSearch(pattern2,sequence)
print("sind",count,"Vergleiche benötigt beim SimpleSearch Algorithmus")

count = 0
matchH_ccc = Horspool(pattern2,sequence)
print("sind",count,"Vergleiche benötigt beim Horspool Algorithmus")

#Prüfen ob die Ergebnisse von 2 Methode ähnlich sind
if matchSS_ccc == matchH_ccc:
    if(len(matchH_ccc) == 0):
        print("Keiner Treffer")
    else: print(len(matchH_ccc),"Treffer sind gefunden")
```

```
Mit Pattern "CCC"
sind 3598 Vergleiche benötigt beim SimpleSearch Algorithmus
sind 1200 Vergleiche benötigt beim Horspool Algorithmus
Keiner Treffer
```

In zwei Versuche mit 2 Pattern kann man sehen, dass die Anzahl der Vergleiche vom *SimpleSearch* viel mehr als die von *Horspool* Algorithmus ist.

4. Diskussion

Im zweiten Beispiel gibt es einen großen Unterschied zwischen der Anzahl der Vergleiche zweier Algorithmen. Das spiegelt die Laufzeit der Algorithmen ab. Der “Horspool” Algorithmus läuft deutlich schneller als der “SimpleSearch”, weil die zu vergleichenden Positionen im “Horspool” besser gewählt werden.