Inhaltsverzeichnis

1 Single String Matching

Sei T ein Text mit Alphabet Σ und sei P (oder p) ein Pattern über dem selben Alphabet. Bei exaktem String-Matching durchsuchen wir T auf alle Vorkommnisse von P. Die Länge von T bezeichnen wir mit n und die Länge von P mit m.

1.1 KMP-Algorithmus

Der KMP-Algorithmus hat eine Worst-Case-Laufzeit von $O(n+m)$, aber ist im Average-Case schlechter als der Naive Algorithmus.

Im Preprocessing wird jedem Buchstaben des Patterns p_i ein Zahl $\pi[i]$ zugewiesen. $\pi[j]$ ist die Anzahl der Buchstaben vor p_j, die auch Prefix des Patterns sind. Trifft beim vergleichen des Patterns mit dem Text ein Mismatch an der Stelle j auf, so kann man das Pattern sicher um $j - \pi[j]$ Stellen shiften.

Beispiel für Preprocessing des Patterns "accacgacct" ($\sigma = \{a, c, g, t\}$:

j	0	1	2	3	4	5	6	7	8	9	10
p_j	a	c	c	a	c	g	a	c	c	t	
$\pi[j]$	-1	0	0	0	1	2	0	1	2	3	0

Die Durchführung des Algorithmus besteht nur darin von links nach rechts das Patern anzulegen, zu vergleichen und sicher zu shiften.

1.2 Shift-And/Or

Da der Shitf-Or eine Operation weniger benötigt, wird nur dieser hier erläutert.

Im Preprocessing weisen wir jedem Buchstaben des Alphabets eine für das Pattern spezifische Bitmaske zu. Diese Maske gibt an an welchen Stellen im Pattern ein bestimmer Buchstabe zu finden ist, jedoch von rechts nach links und negiert (für den ShiftOr). Für unser o.g. Beispielpattern sieht das so aus (die letzte Spalte ist die korrekt Bitmaske für den ShiftOr, die vorletzte für ShiftAnd):

Buchstabe	kommt vor	umgekehrt	negiert
a	1001001000	0001001001	1110110110
c	0110100110	0110010110	1001101001
g	0000010000	0000100000	1111011111
t	0000000001	1000000000	0111111111

Die Durchführung des Algorithmus ist relativ simpel: Wie legen eine Bitmaske D der Länge m an (initialisiert mit 1en) und führen nacheinander für jeden Buchstaben t_i des Textes folgenden Operation durch

$$D = (D << 1) \mid B[t_i]$$

D.h. die Bitmaske wird um eine Stelle nach "links geshiftet" (eine Null wird von rechts "reingeschoben") und dann mit der Bitmaske des gerade akutellen Buchstaben "verodert".

Wird dabei das höchstwertige Bit (das linkeste) 0, so wurde für die Position $i-m+1$ (aktuelle Position - Länge des Paterns + 1) ein Match gefunden.

Wenn die beschriebene Operation in konstanter Zeit möglich ist, ist die Laufzeit linear (für jeden Buchstaben des Textes eine Operation).

1.3 Horspool

Der Horspool-Algorithmus ist eine Vereinfachung des Boyer-Moore-Algorithmus und im Gegensatz zu dem KMP suffix-basiert, d.h. das Pattern wird beim "anlegen" an den Text mit diesem "rückwärts" verglichen. Bei einem Mismatch shiften wir wieder. Der Wert der geshiftet wird ist jeweils für jeden Buchstaben gleich und abhängig von der letzten Position an der dieser im Pattern auftritt, genauer:

Länge des Patterns - letzte-Stelle-des-Auftretens - 1

wobei das Zählen der Stellen bei 0 beginnt (ansonsten kein -1). Bei Buchstaben die im Pattern garnicht auftreten und bei dem Buchstaben, der an der letzten Stelle im Pattern vorkommt wird um die komplette Länge des Patterns geshiftet. Für das Beispielpattern von oben ergibt sich folgende Tabelle:

j	0	1	2	3	4	5	6	7	8	9
p_j	a	c	c	a	c	g	a	c	c	t
$d[p_j]$	3	1	1	3	1	4	3	1	1	10

oder kurz:

$s \in \Sigma$	a	c	g	*
$d[s]$	3	1	4	10

Der Horspool-Algorithmus ist im Worst-Case langsamer als der KMP, im Average-Case jedoch wesentlich besser ($\in O(n/m)$).

1.4 Suffix-Bäume

Aus jedem Text über einem finiten Alphabet, lässt sich ein Sufixbaum generieren. In diesem Baum hat jeder Knoten mindestens zwei Kinder und auf jeder Kante stehen bestimmte Sequenzen aus T. Genauergesagt ergibt der Text über den Kanten von der Wurzel bis zu jeweils einem Blatt genau ein Suffix von T.

Suffixbäume lassen sich eigentlicj linear anlegen, wir müssen sie aber nur naiv (d.h. in $O(n^2)$) anlegen können: Man nimmt jeden Suffix des Texts ("immer einen Buchstaben mehr weglassen") und fügt ihn in den Baum ein, dabei besucht man zuerst die Wurzel und folgt solange exisiterenden Kanten, wie der besuchte Text mit dem des aktuellen Suffix übereinstimmt. Ist dies nicht mehr möglich "zweigt" man an dieser Stelle ab, d.h. wenn dort ein Knoten schon existiert, fügt man eine neue Kanten an; ansonsten wird vorher ein neuer Knoten an der Stelle angelegt.Außerdem wird ein dem Alphabet fremder Buchstabe, aus Konvention "$", vorher ans Ende des Texts angefügt, damit die Endpunkte von Suffixen innerhalb des Baumes markiert werden können. Die Suffixe werden üblicherweise durchnummeriert und diese Indeces an die Endpunkte, d.h. die Blätter geschrieben.

Suchen in diesem Suffixbaum ist relativ trivial, man "geht den Text des Patterns von der Wurzel herab" ($\in O(m)$), alle darunterliegenden Suffixe (die man in $O(j)$ findet), enthalten das Pattern, sind Ergebnisse. $\Longrightarrow$ Gesamtlaufzeit $\epsilon O(m+j)$

Um Speicherplatz zu sparen schreibt man die Texte nicht tatsächlich an die Kanten, sondern speichert dort nur die Indizes. Trotzdem ist der Baum sehr Speicheraufwendig.

1.5 Suffix-Arrays

Statt Suffix-Bäumen, kann man auch in Suffixarrays Stringmatching betreiben. Dafür konstruieren wir die Liste aller Suffixe und sortieren sie alphabetisch (dies geht tatsächlich in O(m+j)). Dann machen wir eine Binärsuche auf der sortierten Liste, wodurch wir den ersten und letzten Match mit dem Pattern finden (das geht in O(log n)). Alle Suffixe dazwischen sind auch Matches.

Der Algorithmus ist mit eine Laufzeit von $O(m+j+logn)$ etwas langsamer, aber verbraucht wesentlich weniger Speicherplatz (durch "Speichertricks" kann man es fast in gleicher Laufzeit wie Suffixbaum-Matching hinbekommen).

2 Multiple String Matching

Beim Multiple String Matching geht es darum eine Menge von r Patterns gleichzeitig in einem gegebenen Text zu suchen.

TODO

3 Pairwise Sequence Alignment

Beim Alignment geht es um das nicht-exakte finden von Übereinstimmungen. Dies ist besonders beim Vergleich von Nukleotid- und Aminosäuresequenzen essentiell, da (kleine) Veränderungen aus allerlei Gründen auftreten können.

Dabei spielt das sog. Scoring eine zentrale Rolle. Scoring gibt für zwie Buchstaben aus einem Alphabet an wie gut oder schlecht sie zusammen passen (es ordnet dem Paar ihre sog. Kosten zu). Scoringmatrizen oder Funktionen hängen von dem Problem z.B. der biologischen Fragestellung ab. Meistens können Buchstaben auch mit einem Gap (d.h. *nichts*) aliniert werden. Dies simuliert z.B. Einfügungen oder Deletationen in Genen.
Beim Scoring werden normalerweisen nur die Fälle Buchstabe 1 mit Buchstabe 2 und jeweils ein Buchstabe mit einem Gap betrachtet (die Reihenfolge bestimmer Buchstaben(-kombinationen) ist irrelevant). Für Gap-Kosten macht man jedoch manchmal eine Ausnahme, da es oft sinnvoll ist eine Reihe von Gaps auf einer Seite

nicht gleichschwehr wie ganau so viele einzelne Gaps zu bewerten. Dies nennt man affine Gap-kosten. Affine Gap-kosten können also das Ergebnis verbessern, erhöhren jedoch oft die Laufzeit der Algorithmen.

3.1 Global Alignemt

Beim Global Alignment betrachten wir die Frage: Wie passt genau diese SequenzA zu dieser SequenzB? (dabei werden Indels - also das Einfügen von Gaps gestattet)

Dies lässt sich mit dem Needleman-Wunsch-Algorithmus berechnen. Bei diesem Algoritmus wird eine Tabelle mit beiden Sqeuenzen als "Beschriftung" angelegt. Die ersten Spalte und zeile wird mit den Gap-Kosten gefüllt (jeweils aufaddiert), danach werden die anderen Felder berechnet. Dabei ergibt sich der Wert an der Stelle so:

1. "von linksoben zu kommen" (Wert links oben + score der beiden Buchstaben)
2. "von oben zu kommen" (Wert oben + score des Buchstabens der vertikalen Sequenz mit Gap)
3. "von links zu kommen" (Wert links score des Buchstabens der horizontalen Frequenz mit Gap)

Von den drei Möglichkeiten wird dann das Maximum genommen und ein Backtracepointer angelegt um zu markieren "von wo man gekommen ist". Ist man unten rechts angekommen, verfolgt man die Backtracepointer zurück und erhällt so Alignement und Gesamtscore (das im letzten Feld).

Es gibt insgesamt $(|SequenzA| + 1)(|SequenzB| + 1)$ Felder, für jedes werden konstant viele Operation durchgeführt, Laufzeit und Speicherverbrauch sind also $\in O(|SequenzA| + |SequenzB|)$.

Beispiel aus dem Skript für die Sequenzen AS-Sequenzen HEAGAWGHEE und PAWHAEAE:

To score the alignment we will use the BLOSUM50 matrix and a gap cost of $d = 8$.

BLOSUM50 values:

	H	E	A	G	A	W	G	H	E	E
P	-2	-1	-1	-2	-1	-4	-2	-2	-1	-1
A	-2	-1	5	0	5	-3	0	-2	-1	-1
W	-3	-3	-3	-3	-3	15	-3	-3	-3	-3
H	10	0	-2	-2	-2	-3	-3	10	0	0
E	0	6	-1	-3	-1	-3	-3	0	6	6
A	-2	-1	5	0	5	-3	0	-2	-1	-1
E	0	6	-1	-3	-1	-3	-3	0	6	6

DP matrix:

	H	E	A	G	A	W	G	H	E	E
	0 ←	**-8** ←	**-16** ←	-24 ←	-32 ←	-40 ←	-48 ←	-56 ←	-64 ←	-72 ←	-80
P	-8 ↖	-2 ↖	-9 ↖	**-17** ←	**-25** ↖	-33 ←	-42 ←	-49 ←	-57 ↖	-65 ↖	-73
A	-16 ↑	-10 ↑	-3 ↖	-4 ←	-12 ↖	**-20** ↖	-28 ←	-36 ←	-44 ←	-52 ←	-60
W	-24 ↑	-18 ↑	-11 ↖	-6 ↖	-7 ↖	-15 ↖	**-5** ←	**-13** ←	-21 ←	-29 ←	-37
H	-32 ↑	-14 ↑	-18 ↖	-13 ↖	-8 ↖	-9 ↖	-13 ↖	-7 ↖	**-3** ←	-11 ←	-19
E	-40 ↑	-22 ↑	-8 ←	-16 ↖	-16 ↖	-9 ↖	-12 ↖	-15 ↖	-7 ↖	**3** ↖	-5
A	-48 ↑	-30 ↑	-16 ↑	-3 ←	-11 ↖	-11 ↖	-12 ↖	-12 ↖	-15 ↖	-5 ↖	2
E	-56 ↑	-38 ↑	-24 ↑	-11 ↖	-6 ↖	-12 ↖	-14 ↖	-15 ↖	-12 ↖	-9 ↖	**1**

HEAGAWGHE-E
--P-AW-HEAE

Durbin et al. (1998)

3.2 Local Alignment

Oft ist es jedoch nicht sinnvoll, zu überprüfen wie zwei Sequenzen genau zueinander passen, sondern zu überprüfen welche Teilsequenzen der beiden ein für diese Kombination optimales Alignment bilden. Das nennt man lokales Alignment und kann man mit dem Smith Waterman Algorithmus berechnen.

Der unterscheidet sich nur in wenigen Punkten von Needleman-Wunsch: Man initialisiert die erste Zeile und Spalte mit Nullen; Man lässt keine negativen Scores zu (stattdessen schreibt man 0); außerdem werden für Nullfelder keine Backtracepointer benötigt.

Das beste Lokale Alignment ist endet am größten Score in der Tabelle, von da aus kann man die Pointer zurückverfolgen bis sie auf ein Nullfeld zeigen. Für das obige Beipsiel ergibt sich folgenden Tabelle:

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	**5**	0	0	0	0	0
W	0	0	0	0	2	0	**20**	**12**	4	0	0
H	0	10	2	0	0	0	12	18	**22**	14	6
E	0	2	16	8	0	0	4	10	18	**28**	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

Und folgendes Alignment (das zwischen den Balken):

```
HEAG|AWGHE|E
   P|AW_HE|AE
```

4 Multiple Sequence Alignment

TODO

5 Automaten & Formale Sprachen

Sei Σ eine endliche Menge von Symbolen, genannt **Alphabet**, dann ist eine Konkatenation von Symbolen dieses Alphabets ein **Wort**. Auch das **leere Wort** bezeichnet durch ε ist ein Wort. Außerdem ist Σ^* die Menge aller Wörter über diesem Alphabet und Σ^+ die Menge aller Wörter ohne das leere Wort.

Eine **formale Sprache** über einem Alphabet Σ ist eine Teilmenge von Σ^*, die **leere Sprache** enthält keine Wörter (auch nicht das leere Wort).

Eine **Grammatik** produziert alle Wörter der für sie spezifischen Sprache.

Ein **Automat** testet ein Wort darauf, ob es Teil seiner akzeptierenden Sprache ist.

5.1 Grammatiken Allgemein

Eine Grammatik G ist gegeben durch

- V Eine endliche Menge von *Variablen Symbolen* oder einfach *Variablen* oder Nicht-terminalen
- Σ Das Alphabet der Sprache, die wir beschreiben wollen; eine Menge von *Terminalen (Symbolen)*; $V \cap \Sigma = \emptyset$
- P Eine endliche Menge von sog. Produktionen. Das sind Regeln die eine Menge von Symbolen (variable oder termina
- S Ein Element von V, dass als Startvariable bezeichnet wird.

Aus Konvention verwendet man für Variablen Großbuchstaben, für Terminale Kleinbuchstaben und für einen String aus beidem griechische Kleinbuchstaben.

Es gibt 4 Arten von Grammatiken:

1. Regular grammars. Only production rules of the form $W \to aW$ or $W \to a$ are allowed.
2. Context-free grammars. Any production of the form $W \to a$ is allowed.
3. Context-sensitive grammars. Productions of the form $\alpha_1 W \alpha_2 \to \alpha_1 \beta \alpha_2$ are allowed.
4. Unrestricted grammars. Any production rule of the form $\alpha_1 W \alpha_2 \to \gamma$ is allowed.

Wir interessieren uns nur für die ersten Beiden.

5.2 Finite Automaten Allgemein

Finite Automaten werden im Allgemeinen als Graphen dargestellt, wobei die Zustände Knoten sind, und die Kanten die Übergange, wobei diese mit den jeweiligen Buchstaben des Alphabets gekennzeichnet werden. Akzeptierende Zustände werden mit zwei Kreisen markiert; Startzustände haben einen Pfeil der auf sie zeigt. Es korrelieren ungefähr: Knoten <-> Variable; Buchstabe <-> Terminale(Buchstaben); Produktion <-> Transition; Startvariable <-> Startzustand;

Aus Konvention werden für "Buchstaben" Kleinbuchstaben verwendet, für Knoten auch :S, insbesondere v,u,z (meist mit Indezes).

5.3 Reguläre Grammatiken

Außer den bereits benannten Eigenschaften gilt:

- Eine Sprache ist genau dann regulär wenn sie durch einen Regulären Ausdruck beschrieben wird.
- Eine Sprache ist genau dann regulär wenn sie von einem Deterministischen Finiten Automaten akzeptiert wird.
- Eine Sprache ist genau dann regulär wenn sie von einem Nichtdeterministischen Finiten Automaten akzeptiert wird (da DFAs und NFAs equivalent sind, s.u.).

Die formalern Kriterien für reguläre Ausdrücke werden hier nicht erörtert.

5.3.1 NFAs

A nondeterministic finite automaton (NFA) is a 5-tuple $M = (Z, \Sigma, \delta, U_0, E)$ satisfying the following conditions.

- Z is a finite set of states.
- Σ is the alphabet.
- $\delta : Zx\Sigma \to P(Z)$ is the transition function. Here P(Z) is the power set of Z, i.e. the set of all subsets of Z.
- U_0 is the set of initial states.
- E is the set of accepting states.

Ein NFA ist nicht determiniert, was bedeutet, dass es von einem Knoten mehrere Übergänge mit demselben Buchstaben geben kann oder garkeine. Er akzeptiert ein Wort, wenn es mindestens einen gültigen Pfad dafür gibt.

Wir können aus einer regulären Grammatik einen NFA erzeugen(das heißt wir zeigen, dass $L(G) := A = L(M)$):

G	M	
V	$Z = V \cup \{X\}$	Wobei X ein neuer Zustand ist, der für die letzte Transition benötitgt wird.
Σ	Σ	sind identisch
P	δ	für jede Produktion $Variable \to terminalVariable$ eine entspr. Transition
		für jede Produktion $Variable \to terminal$ eine Transition nach X
S	$U_0 = z_0 = \{S\}$	
	$E := \{X\}$	Der Endknoten. Wenn $\varepsilon \in L(G)$, dann ist $E := \{X, S\}$

5.3.2 DFAs

Ein DFA unterscheidet sich von einem NFA in folgenden Punkten:

1. Es gibt nur einen Startknoten
2. Es gibt von jedem Knoten genau eine Transition für jeden Buchstaben des Alphabets

Abhängig von den gewälten Buchstaben ist der "Weg durch einen DFA" also eindeutig, ein Wort wird akzeptiert, wenn man nahc dem Lesen bei einem Endzustand landet.

Jeder NFS lässt sich in einen DFA umwandeln:

1. Man legt eine Tabelle an, die die Zustandsmengen enthält von jeweils einem Zustand mit einem Buchstaben erreichbar sind.
2. Dann legt man eine zweite Tabelle an, die jeder dieser Zustandsmengen wiederum die Zustandsmenge zuordnet, die mit einem Buchstaben erreichbar ist.
3. Jede dieser Zustandsmengen wird als ein Zustand des neuen DFAs definiert, die Transitionen sind nun determiniert.
4. Dabei sind alle Zustandsmengen, die den ENdzustand des NFAs enthalten neue Endzustände.

Ein DFA lässt sich wiederum in eine Grammatik überführen:

M	G	
Z	$V := Z$	
Σ	Σ	
δ	P	für jede $z_i \in \delta(z_j, a)$ eine entspr. Produktion $Z_i \rightarrow aZ_j$
		wenn $z_i \in E$ dann auch eine entspr. Produktion $Z_i \rightarrow a$
S	$S := z_0$	
E		

5.4 Kontexfreie Grammatiken

Wie oben beschrieben, erlauben kontextrfreie Grammatiken auch Produktionen, die Kombinationen von Variablen und Terminalen enthalten. Dies ist besonders für Palindromartige Sprachen nützlich.

5.4.1 PDA

Der Push-Down-Automat ist ein spezieller Automat für das Testen auf kontextfreie Grammatiken.

[TODO]