Christopher Goeritz

Stringmatching-Algorithmen in der Bioinformatik

Bachelorarbeit

Institut für Genomik und Bioinformatik
Technische Universität Graz
Petersgasse 14, 8010 Graz

Betreuer: Dipl.-Ing. Dr. techn. Hubert Hackl

Graz, September 2008

1. Inhaltsverzeichnis

1) Inhaltsverzeichnis.. 1

2) Abstract...2

3) Hintergrund...2

4) Ziele... 3

5) Exaktes Stringmatching... 3

5.1) Naiver Algorithmus... 3

5.2) Z-Box Algorithmus..5

5.3) Boyer-Moore Algorithmus.. 7

5.4) Knuth-Morris-Pratt.. 9

6) Approximatives Stringmatching…………………………………………………...11

6.1) Dotplot…………………………………………………………………… 11

6.2) Edit-Skript…….. 13

6.3) Globales Alignment……………………………………………………… 14

6.4) Scoringfunktion………………………………………………………... 16

6.5) Lokales Alignment……... 16

6.6) Smith-Waterman Algorithmus... 17

6.7) Multiples Sequenz Alignment... 18

6.8) Profil Alignment.. 20

7) Diskussion.. 22

8) Referenzen... 23

9) Literaturverzeichnis... 23

10) Anhang..24

2. Abstract

Die folgende Arbeit konzentriert sich auf das Thema Stringmatching mit Fokus auf den Bereich Algorithmische Bioinformatik. Hierfür werden einige der wichtigsten Algorithmen vorgestellt, die zum Auffinden eines Pattern innerhalb eines (enorm großen) Template benutzt werden. Dabei wird für jede dieser Methoden aufgezeigt welche Vorbereitungen zu treffen sind, wie der Algorithmus an sich abläuft, welche Tricks er benutzt um hohe Effizienz zu erreichen und mit was für einem zeitlichen Aufwand die Aufgabe erfüllt wird. Außerdem werden auch alle gezeigten Beispiele für exaktes Stringmatching implementiert.

3. Hintergrund

Im Jahre 2003 wurde das humane Genom mit einer Genauigkeit von 99,99% zu 99% sequenziert [1] – es umfasst (grob geschätzt) 3.000.000.000 Zeichen [2]. Um in diesem Datensatz einzelne Pattern zu finden, z.B. für die Suche nach bestimmten Genen, bedarf es hinreichend leistungsstarker Algorithmen, die in angemessener Zeit eine Lösung finden können, denn es gibt schätzungsweise 30.000-40.000 proteinkodierende Gene im menschlichen Genom [1], die aber nur 2-3% der gesamten DNA ausmachen [2]. Die Rolle der Informatik geht hier sogar noch weiter, nämlich u.a. im Aufbauen so genannter phylogenetischer Bäume, bei denen es um weiter verzweigte Verwandtschaftsgrade verschiedener Spezies geht.

Hinter dem Gedanken, einen Stammbaum der Spezies aufzustellen, steckt die Grundidee der Evolution, das Ziel, alle Arten des Lebens auf eine Urform zurückzuführen. Man benutzt dazu die Annahme, dass ähnliche Stämme auch ähnliche Sequenzen besitzen, ähnliche Sequenzen wiederum ähnliche Funktionen der Proteine bedingen und somit wahrscheinlich auch eine ähnliche Wirkung haben. Es kann sich dann auch herauslesen lassen, an welchen Stellen und zu welcher Phase der Evolution Mutationen aufgetreten sind.

Es gibt jedoch noch weitere Anwendungen der Stringmatching-Algorithmen, denen ein viel näheres Ziel zu Grunde liegt. Soll zum Beispiel ein Genom sequenziert werden, so kann dies nur in sehr kurzen Stücken passieren. Hierzu werden die Chromosomen in viele Einzelteile zerlegt (Clonierung), diese dann einzeln sequenziert, und müssen anschließend wieder zusammengesetzt werden (Assembly), d.h. aus der Sequenz der Bruchstücke muss die Sequenz des ursprünglichen Chromosoms berechnet werden. Dies erweist sich oft als komplizierter als zunächst gedacht, da sich aufgrund der zufälligen Schnitte durch das Chromosom die Teilstücke überlappen. Auf der einen Seite ist diese Redundanz von Vorteil, da man so eine höhere Gewissheit über den Inhalt hat, andererseits führt sie beim Zusammenfügen zu Konflikten und Mehrdeutigkeiten. Mit exakten Stringmatching-Algorithmen ist hier meistens keine eindeutige Lösung zu finden, weswegen das approximative Stringmatching, das Heuristiken und Näherungsverfahren nutzt, ein sehr viel wichtigeres Gebiet darstellt.

Ein anderes Beispiel der Anwendung der Algorithmen ist das EST Clustering zum Auffinden von Genen. Eine ansequenzierte cDNA erzeugt so genannte ESTs (Expressed Sequence Tags) [3], welche in Datenbanken gelagert werden. Von diesen ESTs wird zunächst die Überlappung berechnet, und sehr ähnliche Sequenzen zu Sequenzclustern zusammengefasst. Mit erneuten heuristischen Verfahren kann dann die Anzahl der Gene durch Clustering aller EST Sequenzen berechnet werden. Des Weiteren kann man aus der Sequenz auch auf die Funktion schließen, wodurch sich eine Annotationspipeline ergibt: Suche nach ähnlichen Gensequenzen, Suche nach ähnlichen Proteinen, Vorhersage neuer Gene durch Programme (trainiert auf bekannten Gensequenzen).

Diese Datenmengen zeiteffizient und speicherplatzsparend durchzuführen hat sich die algorithmische Bioinformatik zur Aufgabe gemacht, als Unterkategorie der Bioinformatik. Als in den 1960ern GenBank eingerichtet wurde, wurde schnell klar, dass man Methoden brauchte, diese Datensätze zu handhaben. So entstand aus der Biologie und der Informatik der neue Wissenschaftszweig der Bioinformatik [2]. Es wurden verschiedene Algorithmen entwickelt, Pattern in langen Templates exakt oder approximativ zu finden, um die oben genannten Probleme mit akzeptabler Komplexität lösen zu können.

4. Ziele

Ziel dieser Arbeit ist es, die wichtigsten Stringmatchingalgorithmen mit Fokus auf die algorithmische Bioinformatik zu erläutern. Um die Grundlagen der verschiedenen Stringmatching Methoden zu besprechen, wird im Nachfolgenden zunächst auf exaktes und danach auf approximatives Matching eingegangen, so dass zuletzt einige umfangreichere Anwendungen besprochen werden können. Es soll dabei stets auf Komplexität und Effizienz der Verfahren geachtet werden, und deren Bedeutung für andere Aufgaben der Bioinformatik hervorgehoben werden.

5. Exaktes Stringmatching

Bei exaktem Stringmatching wird ein Pattern P in einem Template T gesucht, wobei P normalerweise (sehr) viel kleiner als T ist. Es sollen immer alle exakten Vorkommen von P in T gefunden werden, hierzu wollen wir ab jetzt annehmen, dass die Längen von T (m) und von P (n) größer Null sind, und dass das zu Grunde liegende Alphabet endlich ist. Die nachfolgenden Komplexitätsanalysen werden die Anzahl an Stringvergleichen bei einem Algorithmus zählen, wofür wir annehmen, dass die Kosten für Vergleiche zweier Zeichen stets 1 betragen.

Wir können uns das exakte Stringmatching anhand eines für die Bioinformatik sehr wichtigen Bereiches vor Augen führen: der Transkription. Der Startpunkt dieses Vorgangs wird durch einen so genannten Promotor gekennzeichnet, der aus verschiedenen regulatorischen Sequenzen besteht, einer AT-basenpaar reichen Region, und unter anderem der „-35-Region“ (bei bakteriellen Promotoren zum Beispiel TTGACA) und der „Pribnow-Box“ (bei Prokaryoten zum Beispiel TATAAT), benannt nach ihrem Entdecker David Pribnow [4]. Da diese Bereiche explizit bekannt sind, kann man Algorithmen exakt nach ihnen suchen lassen, und so zum Beispiel die Positionen von Promotoren oder sogar ganzer Gene abschätzen. Die in diesem Fall benutzten Sequenzen des Promotors sind zwar nicht exakt konserviert und somit nur Consensus Sequenzen, doch damit die Beispiele überschaubar bleiben, werden wir uns auf die aus den oben genannten Regionen zusammen gesetzte Sequenz ATATTTGACATATAAT beschränken, und in den folgenden Beispielen als Pattern P verwenden.

5.1 Naiver Algorithmus

Beginnen wir zunächst mit dem naiven Ansatz. Wir wollen P in T finden, also beginnen wir in beiden Strings an der ersten Position und vergleichen dann Zeichen für Zeichen. Solange die Zeichen matchen rücken wir in beiden Strings eine Position vor, doch wenn wir auf ein Mismatch stoßen, dann wissen wir, dass P nicht an der aktuellen Startposition in T beginnt. Also beginnen wir diesmal in T eine Position weiter rechts und in P wieder am Anfang und matchen wieder jedes Zeichen der Reihe nach. Jedes Mal wenn wir das letzte Zeichen in P erreichen ohne ein Mismatch zu produzieren, können wir ein exaktes

Vorkommen von P in T an der aktuellen Startposition ausgeben, müssen dann allerdings trotzdem weiter machen, da es noch weitere geben könnte. Erst wenn wir die letzte Position in T abgearbeitet haben, können wir die Prozedur beenden. Man kann allerdings hier schon einige Schritte sparen, wenn man nicht bis zur letzten Position in T arbeitet, sondern nur bis zur Position m – n, weil die Suffixe von T hinter dieser Stelle zu kurz sind um P überhaupt enthalten zu können.

Die Komplexität dieses Verfahrens lässt sich relativ leicht bestimmen. Wir betrachten den Worst-Case dieses Algorithmus, nämlich dass in T an jeder Stelle das gleiche Zeichen steht, und in P auch an jeder Stelle das Zeichen aus T steht bis auf die letzte Position, an der ein beliebiges anderes Zeichen steht. Somit müssen wir an jeder einzelnen Startposition bis zum Ende von P matchen nur um dann jedes Mal einen Mismatch zu produzieren. Wir führen somit insgesamt rund n * (m-n+1) Vergleiche durch, was uns zu der extrem hohen Komplexitätsklasse O(m*n) führt.

Kehren wir nun zu unserem Beispiel des Auffindens der Promotersequenz zurück. Wir haben eine DNA-Sequenz T, in der unsere Sequenz P zu finden ist.

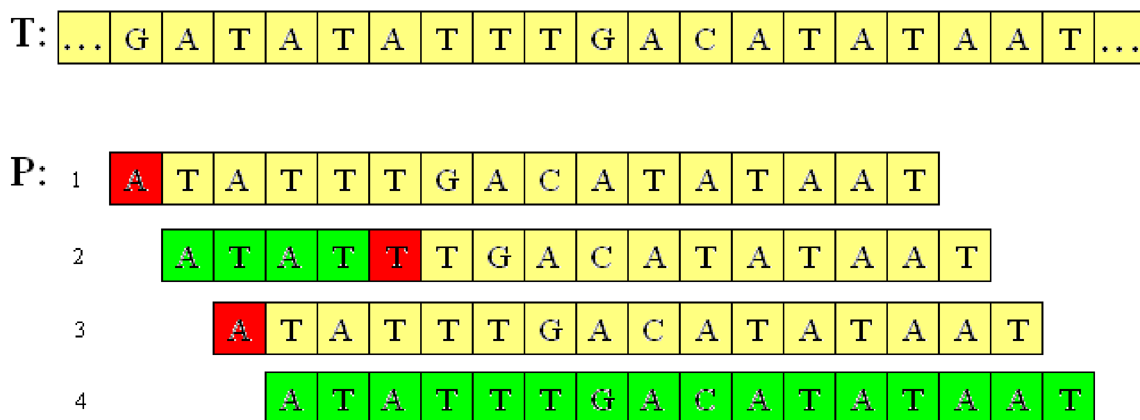


Fig.1: Naive Suche nach Matches

Im ersten Durchlauf mismatcht das erste Zeichen sofort, d.h. wir schieben P eine Stelle nach rechts und beginnen erneut. Im zweiten Durchlauf müssen immerhin schon 5 Vergleiche bearbeitet werden, bevor ein Mismatch auftritt. Im vierten Durchlauf finden wir dann ein komplettes Vorkommen, müssten dann allerdings, wenn T noch länger wäre, noch weitere Durchläufe starten, da es multiple Vorkommen geben könnte.

Dass der naive Ansatz nicht der beste ist, haben wir damit gezeigt, doch es können noch einige Optimierungsmöglichkeiten gefunden werden. Auf jeden Fall muss die Anzahl der Vergleiche reduziert werden, und das schaffen wir indem wir P nicht an jeder einzelnen Position in T suchen, sondern solche überspringen, von denen wir sicher wissen, dass P dort nicht matcht. Allerdings darf dabei natürlich kein Match übersehen werden, wir kommen also zu dem Ansatz, dass, wenn wir bereits ein Paar Zeichen aus P erfolgreich matchen konnten, wir bereits die Zeichen aus T bis vor der Stelle des Mismatches kennen. Somit wissen wir auch, ob das Anfangszeichen von P bis zu jener Stelle nochmals in T vorkommt, und falls ja, dann können wir gleich ab der ersten Position dieses Auftretens weiter suchen, ansonsten brauchen wir diesen Substring von T gar nicht mehr zu betrachten und können direkt an der Stelle des Mismatches fortfahren.

Eine andere Vorgehensweise zum Optimieren der Vergleichsanzahl wäre, sich zuvor P genauer anzuschauen und eventuelle Muster zu lernen. Wenn zum Beispiel der Präfix von P auch als echter Substring von P erneut vorkommt, dann kann uns das ebenfalls helfen Vergleiche zu sparen. Denn wenn wir einmal erfolgreich den Substring aus P in T matchen

konnten, dann wissen wir, wo dieser Substring auch in T nochmals vorkommt. Wenn wir dann P in T sozusagen weiter schieben, bis der Präfix von P mit dem Substring in T (die ja identisch sind) matcht, können wir gleich von dort aus weitersuchen, und sogar hinter dem Präfix in P anfangen zu vergleichen, da wir P ja extra so gesetzt haben dass der Anfang stimmt.

5.2 Z-Box Algorithmus

Der naive Algorithmus benötigt also quadratische Laufzeit, was auf den enorm großen Datenmengen, die für die Bioinformatik interessant sind, viel zu langsam und zu teuer ist. Doch erste Ideen zur Verbesserung der Performanz scheinen zu zeigen, dass etliche Vergleiche gespart werden können. Dies führt uns zum nächsten Algorithmus in der Betrachtung des exakten Stringmatchings, der so genannte Z-Algorithmus. Dieses Verfahren läuft in zwei Phasen ab, zuerst ein Preprocessing auf den beiden Strings, und danach erst die eigentliche Suche. Das Preprocessing soll uns Aufschluss über die Struktur von P und T geben, und alle Eigenheiten lernen, die uns bei der Suche von Vorteil sein können. Somit wollen wir erreichen, dass wir beim Matching P um mehr als eine Position in T verschieben können und gleichzeitig nicht jedes Mal an der ersten Stelle in P anfangen müssen zu vergleichen.

Das Preprocessing des Z-Algorithmus soll alle „Z-Boxen“ unseres Suchproblems finden, das heißt alle längsten Substrings, die auch gleichzeitig Präfix sind. Hierzu bauen wir einen neuen String S, indem wir P und T konkatenieren und durch ein Sonderzeichen, dass nicht Teil unseres Alphabets ist, trennen (z.B. S = P || ‚$’ || T). Dann durchlaufen wir S ab dem Sonderzeichen, und merken uns für jede Position die Länge des längsten Substrings ab dieser Stelle, der auch Präfix von S ist. Wir wissen also für jede Stelle von S die Größe der Z-Box (kann auch Null sein). Das Suchen geht dann ganz schnell, wir gehen nämlich S nur einmal komplett durch, schauen uns jede Z-Box an, und jedes Mal wenn die Größe gleich n ist, haben wir an der aktuellen Position ein Match gefunden (weil dort dann ein Substring steht, der auch Präfix von S ist, und n Zeichen lang ist, und die ersten n Zeichen von S sind genau P).

Die Suche läuft nun also schon in O(m) ab, da wir für jede Position nur einen Vergleich durchführen müssen, nämlich nur einen Zahlenvergleich. Doch wie findet man nun eigentlich alle Z-Boxen? Wenn wir einfach naiv ab jeder Position mit dem Präfix matchen, dann haben wir für die Komplexität keinen Vorteil erreicht, denn wir würden zwar in O(m) suchen können, aber wir bräuchten wieder $O((m+n)^2)$ für das Finden der Z-Boxen, und kämen somit wieder auf eine quadratische Gesamtkomplexität $O(m^2)$. Wir müssen also geschickter Vorgehen, und werden zunächst einige Definitionen einführen: Wir bezeichnen mit r(i) den maximalen Endpunkt aller Z-Boxen, die bei oder vor der Position i beginnen, und mit l(i) die Startposition der längsten Z-Box, die bei r(i) endet. Somit haben wir für jedes i einen Substring S[l(i)..r(i)], der Z-Box ist, die Position i enthält, am weitesten nach rechts reicht und am längsten ist. Mit Hilfe dieser Werte können wir die Größen von neuen Z-Boxen aus bereits bekannten mit konstantem Aufwand berechnen. Der Beweis läuft induktiv ab:

Wir beginnen an Position k = 2, berechnen den Wert Z(2) der Z-Box an dieser Stelle, und wenn Z(2) > 0, dann setzen wir eine Merkvariable r = r(2) = 2+Z(2)-1 und l = l(2) = 2, ansonsten gilt r = l = 0. Für den Induktionsschritt k > 2 kennen wir bereits die Werte von r und l, und außerdem kennen wir alle Werte der Z-Boxen vor der momentanen Position. Wir unterscheiden nun zwei Fälle: Wenn unser k größer als r ist, dann wissen wir, dass es keine Z-Box gibt die k enthält (da k weiter rechts liegt als der rechteste Rand einer uns bekannten Z-Box). Somit wissen wir auch nichts über die Zeichen in S an dieser Stelle und müssen erstmal primitiv vorgehen, indem wir jedes Zeichen mit dem Präfix matchen, dann r = r(k) und l = l(k) setzen und mit dem nächsten k fortfahren. Wenn unser k allerdings kleiner oder

gleich r ist, dann liegt es innerhalb einer bekannten Z-Box. Die Zeichen in einer Z-Box sind aber gleich dem Präfix, also kommt auch k im Präfix vor (nennen wir es dort k' = k-l+1). Der Substring b = S[k..r] steht also auch an Position k', außerdem kennen wir Z(k'). Wenn jetzt Z(k') kleiner ist als |b|, dann gab es bei der Berechnung von Z(k') einen Mismatch bei k'+Z(k'), somit wird es auch bei S[k+Z(k')] den gleichen Mismatch geben (da wir noch innerhalb der bekannten Z-Box sind, und dort alle Zeichen gleich wie der Präfix sind). Dann können wir vorhersagen, dass Z(k)=Z(k'), und wir brauchen r und l nicht zu verändern. Sollte allerdings Z(k') größer oder gleich |b| sein, dann wissen wir schon dass b ein Präfix von S ist, und brauchen nur mehr ab S[r+1] mit S[|b|+1] zu matchen, bis wir einen Mismatch finden. Sei dieser Mismatch an Position q in S, dann wissen wir sofort Z(k) = q-k (Länge der Z-Box von k bis q), r = q-1 und l = k (wenn q ungleich r+1, also wenn wir mindestens einen richtigen Match finden konnten).

Somit können wir die meisten Z-Boxen aus bereits bekannten berechnen, und sparen wiederum eine Menge Vergleiche. Wir benötigen mindestens O(|S|), da wir mit k jede Stelle von S einmal betrachten. Wir machen pro k maximal einen Mismatch bzw. keinen Mismatch wenn wir im zweiten Fall keine Zeichen vergleichen müssen. Im Induktionsanfang (bzw. in Fall 1) machen wir eine bestimmte Anzahl Matches, dann einen Mismatch, dann setzen wir r und l. Sofern nicht der erste schon Mismatch war, wird jetzt Fall 2 eintreten, und wir matchen entweder gar nicht oder nur rechts von r. Somit wird jedes Zeichen aus S nur ein einziges Mal mit dem Präfix verglichen. Wir brauchen also |S| Matches oder weniger und |S| Mismatches oder weniger, wodurch wir für das gesamte Preprocessing in O(|S|) bleiben. Die Suche brauchte O(m), was für den kompletten Z-Algorithmus bedeutet, dass er in O(|S|)=O(m+n) alle exakten Vorkommen von P in T finden kann.

Für unser Promoterbeispiel betrachten wir ein Paar einzelne Berechnungen für Z-Boxen während des Vorgangs.

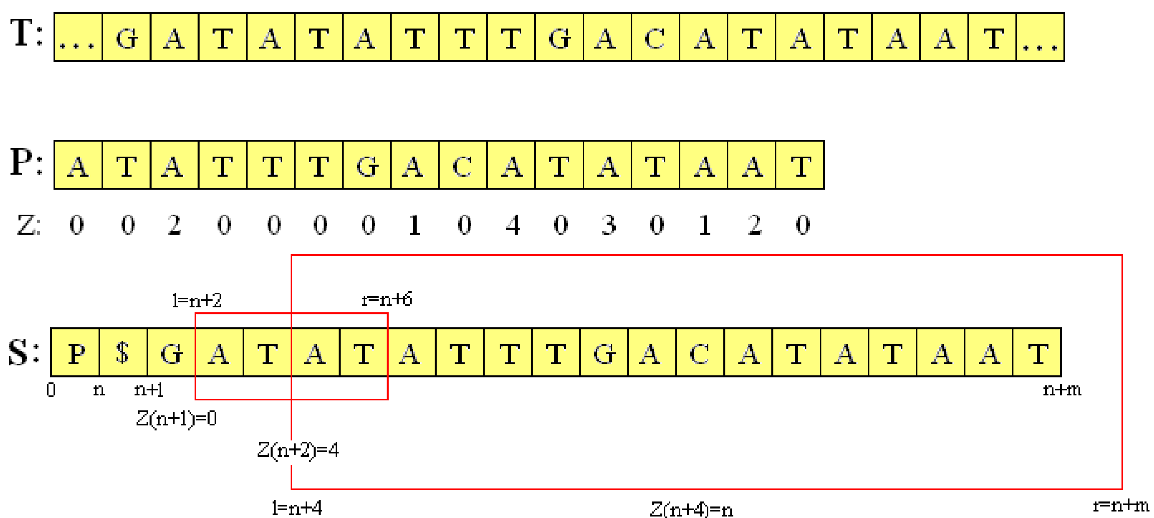


Fig.2: Z-Boxen bei n+2 und n+4

In dieser Situation haben wir schon alle Z-Boxen bis $ berechnet, und beginnen nun bei n+1. Für k=n+1 befinden wir uns in keiner bekannten Z-Box, somit gehen wir primitiv vor, bekommen ein Mismatch und setzen Z(n+1)=0. Das gleiche für k=n+2, jedoch finden wir hier eine Z-Box der Länge 4, und wir setzen l=n+2 und r=n+6. k=n+3 ist nun kleiner als r, wir setzen k'=k-l+1=(n+3)-(n+2)+1=2 und |b|=3. Z(k') ist allerdings 0 und somit kleiner als b, daher wissen wir Z(k)=Z(k')=0. Für k=n+4 kriegen wir k'=3 und Z(k')=b, somit matchen wir ab S[n+7]=S[3] weiter, und finden l=n+4 und r=n+m und Z(n+4)=n, was bedeutet, dass P an

dieser Stelle komplett gematcht wurde. Da r jetzt schon ganz rechts am Ende liegt, werden alle weiteren Untersuchungen im ersten Fall enden, z.B. für $k=n+13$ ist es natürlich kleiner als r, $k'=(n+13)-(n+4)+1=10$, $Z(k')=4 < b=7$ wodurch $Z(k)=4$ ohne weitere Vergleiche vorhergesagt werden kann.

5.3 Boyer-Moore Algorithmus

Wir kennen nun bereits einen Algorithmus, der unser Problem in linearer Zeit lösen kann, doch auch das ist in den meisten Fällen immer noch nicht optimal. Wir suchen also nach einer Methode, wie wir exaktes Stringmatching in sublinearer Zeit durchführen können. Hierfür ist der Algorithmus von Boyer und Moore geeignet [5], welcher zwar im Worst-Case immer noch linear ist, dafür aber in durchschnittlichen Fällen Sublinearität erreicht (in „normalen“ Texten, denen ein großes Alphabet zu Grunde liegt, z.B. Chinesisch, aber auch für Proteinsequenzen). Dafür werden P und T wieder so wie im naiven Ansatz ausgerichtet, der Unterschied ist allerdings, dass P von rechts nach links gematcht wird, also von P[n] mit T[n] zu P[1] mit T[1]. Außerdem wird versucht die Sprungweite noch stärker zu vergrößern mit Hilfe der Bad Character Rule und der Good Suffix Rule, auf die im Nachfolgenden genauer eingegangen werden soll.

Für die Betrachtung der Bad Character Rule legen wir zunächst folgenden Sachverhalt zu Grunde: P und T sind beliebig aligniert, die Positionen seien P[n] und T[j], wobei j wahrscheinlich sehr viel größer (nur ganz am Anfang gleich) als n ist. Weiters nehmen wir an, dass bei P[i] ein Mismatch auftrat, also nach n-i Matches, und das Zeichen in T, dass den Mismatch ausgelöst hat, liegt damit an Position j-n+i und heiße x. Wir überlegen uns nun, welches Zeichen in P stattdessen mit x matchen könnte – kommt x in P überhaupt nicht vor, können wir P gleich an die Position hinter x weiter schieben (i Schritte nach rechts). Gibt es x allerdings an anderer Stelle in P, und sei das am weitesten rechts in P liegende x an Stelle l, dann können wir P um i-l Zeichen verschieben, wenn l kleiner ist als i (d.h. x ist in P nur links vom Mismatch). Alle Vorkommen von x in P rechts vom Mismatch sind uninteressant, da wir dort schon verglichen haben, wir legen uns also eine Funktion R(x) an, die für jedes Zeichen unseres Alphabetes entweder Null (kommt nicht in P vor) oder l (Position des am weitesten rechts in P aber links von i liegenden Vorkommens) ist. Die Berechnung dieser Werte lässt sich leicht innerhalb eines Durchlaufs über P durchführen (in O(n) Zeit), und die Sprungweite lässt sich nun mittels $\max(1, i - R(x))$ bestimmen. Diese Vorgehensweise ist somit sehr einfach und leicht anzuwenden, und auch sehr hilfreich bei größeren Alphabeten (wenn einfach die Wahrscheinlichkeit für viele gleiche x in P gering ist), jedoch ist der (angepasste) Worst-Case des naiven Algorithmus immer noch genauso komplex.

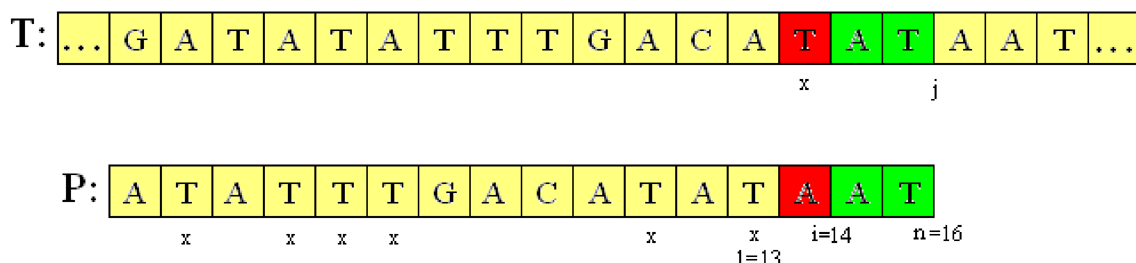


Fig.3: Bad Character Rule

Aus diesem Grund verwendet der Algorithmus nicht nur die Bad Character Rule, sondern gleichzeitig auch die Good Suffix Rule, die unabhängig von einander arbeiten und beide eine mögliche Sprungweite berechnen, so dass wir dann P um das Maximum der beiden Werte schieben können. Die Good Suffix Rule macht sich zu nutze, dass wir bis zu jenem ersten Mismatch bereits einen größeren Match gefunden haben, und falls dieser Suffix in P mehrmals auftritt, können wir den am weitesten rechts liegenden mit dem gerade in T gematchten Substring alignieren. Sollte der Suffix nicht mehrmals in P vorkommen, so können wir P bis an die Position nach dem Mismatch in T schieben. Sei k die letzte Position des am weitesten rechts liegenden Vorkommens des Suffix t in P (k < n), und sei das Zeichen in P, das den Mismatch ausgelöst hat (an Stelle P(n-|t|)) ungleich dem Zeichen vor t (an Stelle P(k-|t|)), dann verschiebe P um n-k (wir wollen den gleichen Mismatch nicht zweimal produzieren). Falls kein weiteres t mit obigen Bedingungen existiert, dann schiebe P um n-|t|+1. Mit Hilfe dieser Regel läuft der bisher quadratische Worst-Case in linearer Zeit, allerdings benötigen wir wiederum ein Preprocessing von P, das uns mehrmals auftretende Suffixe in P berechnet. Wir suchen also zu jedem Suffix t = P[i..n] den am weitesten rechts liegenden Endpunkt L'(i) eines identischen echten Substrings von P. Dieses Verfahren ähnelt sehr dem oben beschriebenen Z-Box Algorithmus, nur dass wir jetzt anstatt Präfixen Suffixe suchen, und wir nur mehr eines brauchen. Um also den Algorithmus trotzdem verwenden zu können, definieren wir uns N(j) als die Länge des längsten Suffix von P[1..j], das auch gleichzeitig Suffix von ganz P ist, welches wir dann symmetrisch zu den Z(i)-Werten mit Z-Boxen auf einem „umgekehrten" P berechnen können. Da wir aber nur ein bestimmtes N(j) brauchen, suchen wir hinterher einen Suffix der am weitesten rechts liegt und n-i+1 lang ist, und nennen ihn L'(i) (Es ist der größte Wert von j, für den gilt: N(j) = n-i+1). Zwar stehen alle j mit diesem N(j) für ein nicht weiter verlängerbares Suffix der passenden Länge, aber wir benötigen nur das rechteste, also in diesem Fall das größte j.

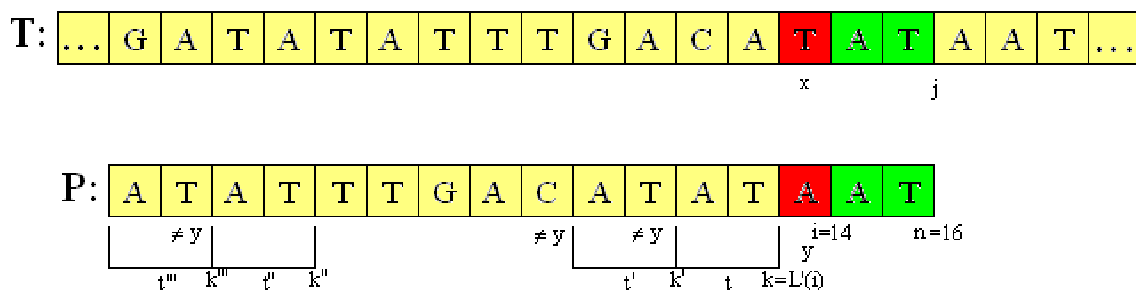


Fig.4: Good Suffix Rule

Das Preprocessing berechnet also zuerst die N-Werte mittels Z-Boxen, was in O(n) Zeit abläuft (siehe oben), und danach noch die L'-Werte in einem einzigen Durchlauf, wobei einfach verglichen wird welches größer ist – also auch in O(n) Zeit (da wir nur auf P und nie auf T arbeiten ist O(n) sehr niedrig). Die Suche wird im Normalfall dann ziemlich viele Sprünge machen, und T nur ein einziges Mal durchlaufen, wodurch viele Zeichen überhaupt nie gematcht werden müssen, und der Algorithmus sublineare Zeit benötigt. Um jetzt auch die Worst-Case Komplexität linear zu bekommen, wollen wir sicherstellen, dass wir jedes Zeichen in T maximal einmal matchen, denn dann hätten wir maximal m Matches und maximal m Mismatches, da wir nach jedem mindestens um 1 schieben – also zusammen O(m). Dies können wir mit Hilfe eines Arrays M[m] erreichen, den wir zunächst mit -1 initialisieren. Wenn der Algorithmus jetzt beim matchen auf einen Mismatch an Stelle i in P stößt, dann stimmen P[i+1..n] und T[k-n+i..k] überein (k sei die Position von P in T). Damit der Algorithmus sich dies „merkt", setzen wir einen bestimmten Wert im Array an die Stelle M[k].

Somit kennen wir nach dem Schieben von P bereits alle Werte M[i] mit i < k. Sei nun i die Laufvariable in P und h die Laufvariable in T, dann können wir fünf Fälle unterscheiden:

1) Wenn M[h] = -1 oder M[h]=N(i)=0 , dann wissen wir, dass wir einen vollständigen Match haben bei T[h]=P[i] und i=1, dass wir weitermachen müssen bei T[h]=P[i] und i>1, oder dass M[k]=k-h bei T[h]=/=P[i] .
2) Wenn M[h]<N(i), dann wissen wir bereits, dass die Zeichen hier übereinstimmen, und wir können sofort bei h=h-M[h] und i=i-M[h] weitermatchen.
3) Wenn M[h]>=N(i) und N(i)=i > 0, dann können wir sofort einen kompletten Match von P in T melden, denn in T liegt ein Substring, der bei h endet, länger als N(i) ist, und dessen Suffix gleich N(i) ist. Außerdem matcht der Teil von P vor dieser Stelle mit T (sonst würden wir gar nicht so weit kommen), und N(i) beschreibt den gesamten Rest von P, der auch mit T matcht, wodurch P an dieser Stelle komplett in T vorkommt (M[k] wird dann auf n gesetzt).
4) Wenn M[h]>N(i) und N(i)<i, dann können wir sofort einen Mismatch melden, denn N(i) ist kürzer als M[h] gerade weil dort ein anderes Zeichen als im Suffix von P auftritt (d.h. wäre dort kein Mismatch, dann wäre N(i) größer), und wir können M[k] = k-h+N(i) setzen.
5) Wenn M[h]=N(i) und N(i)<i, dann wissen wir zwar, dass der folgende Teilstring auf jeden Fall mit T matcht, was danach kommt wissen wir aber nicht. Also können wir direkt zu der Stelle hinter dem Substring springen mit h=h-M[h] und i=i-M[h] und von dort weitermatchen.

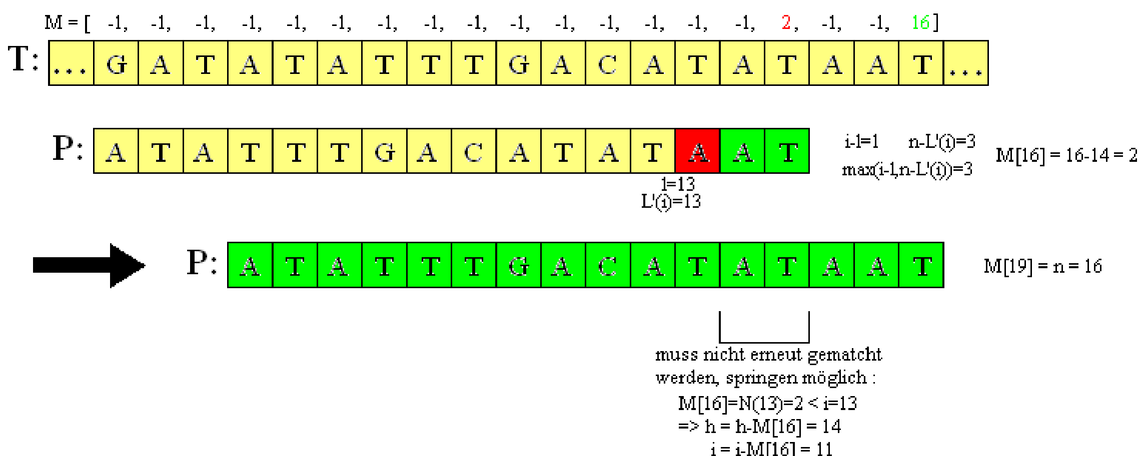


Fig.5: Veranschaulichung der Arbeitsweise des Boyer-Moore Algorithmus

Mit Hilfe dieser Änderung und der Einführung des Hilfs-Arrays M wird also jedes bereits gematchte Zeichen von T in M „gespeichert“, und bei erneuten Vergleichen werden diese Zeichen dann einfach übersprungen. Somit wird jedes Zeichen von T nur maximal einmal gematcht, und jeder Mismatch führt zu einer Verschiebung von P, d.h. wir erreichen lineare Komplexität im Worst-Case.

5.4 Knuth-Morris-Pratt

Wir kennen nun einen Algorithmus, der sich für die meisten Arten von Texten als der praxistauglichste erwiesen hat, doch es soll trotzdem noch ein weiteres Verfahren für exaktes Stringmatching vorgestellt werden, da dieses einen entscheidenden Vorteil hat: Es lässt sich leicht auf mehrere Pattern erweitern. Der Knuth-Morris-Pratt Algorithmus geht ähnlich vor wie der naive Algorithmus, führt jedoch wiederum ein Preprocessing auf P durch um unnötige Vergleiche zu überspringen. Da wir wie zu Anfang von links nach rechts matchen, suchen wir

jetzt echte Suffixe für jedes P[1..i], die mit einem Präfix von P matchen. Die Länge des längsten Suffix sei sp(i), und wir definieren ein sp(i)' = sp(i), wenn das Zeichen nach dem Präfix ungleich dem Zeichen nach dem Suffix ist (P[i+1]=/=P[sp(i)+1]), ansonsten ist sp(i)'=0. Diese Werte können wir dann zum Verschieben benutzen: Wenn das erste Zeichen sofort mismatcht, schieben wir P nur um 1, wenn der erste Mismatch bei i+1 auftritt, können wir P um i-sp(i)' Stellen nach rechts schieben und ab P[sp(i)'+1] weiter vergleichen. Sollte kein einziger Mismatch auftreten (wir haben P komplett in T gefunden), dann können wir P um n-sp(n)' nach rechts schieben und ab P[sp(n)'+1] weiter matchen.

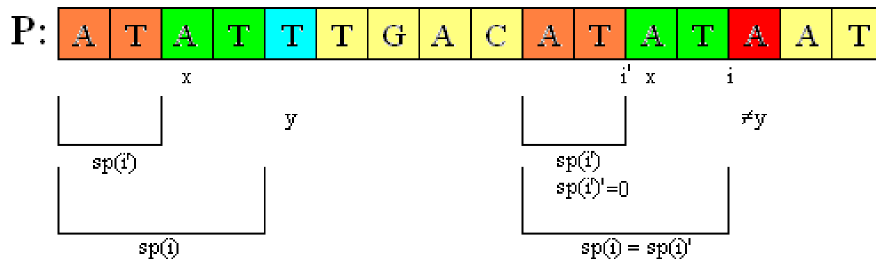


Fig.6: Bestimmung der sp(i)'-Werte

Dass diese Regel nie soweit schiebt, dass ein Vorkommen von P in T übersehen wird, beweisen wir durch Annahme des Gegenteils. Angenommen wir finden einen Mismatch bei i+1 in P und bei k in T, dann endet bei i+1 ein Substring b, der auch Präfix von P ist und wir schieben P bis an die Anfangsposition von b in T. Nehmen wir weiters an, dass wir bei diesem Shift einen Substring a (|a|>0) direkt vor b übersehen hätten, der zu einem Auffinden von P in T geführt hätte, und wir nennen die Position von P vor dem Shift P1, nach dem Shift P2 und das übersehene P sei P3. Laut Definition ist |b| = sp(i)', P1 matcht mit T bis k-1, und P3 matcht ebenfalls bis k-1, da a auch Präfix wäre, danach b kommen müsste, dann allerdings kein Mismatch, da wir von einem übersehenen kompletten P ausgehen. Somit ist [ab] Suffix von P1[1..i], aber auch Präfix von P3, und das Zeichen nach [ab] ist ein Mismatch (P[1+i]=/=P[|a|+1]). Diese drei Beobachtungen zusammen erfüllen alle Voraussetzungen für sp(i)', aber |ab| > sp(i)' (sp(i)'=|b|), somit haben wir einen Widerspruch zur Definition von sp(i)', denn es müsste gelten |ab| = sp(i)'. Also können wir bei oben beschriebenen Shifts nie aus Versehen ein Vorkommen von P in T übersehen.

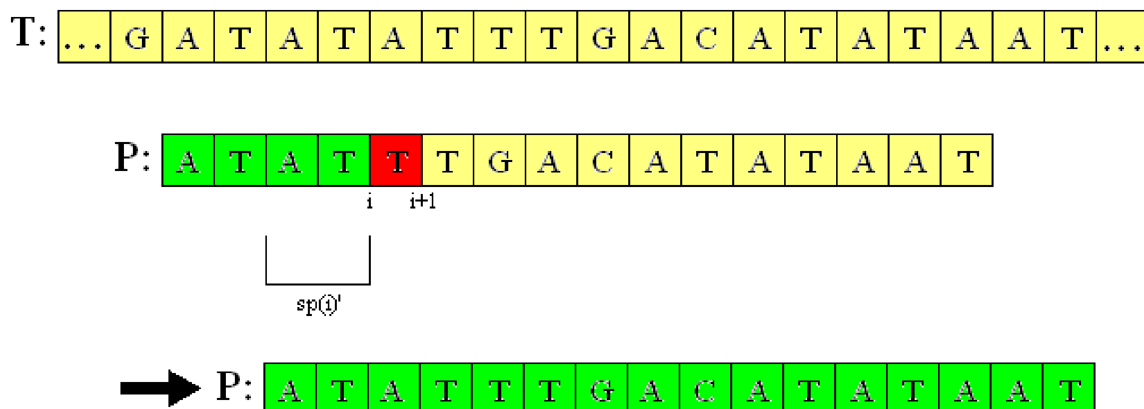


Fig.7: Schrittweise des Knuth-Morris-Pratt Algorithmus

Die Komplexität des Algorithmus ist wieder linear, da wir pro Phase maximal einen Mismatch bekommen, und um mindestens 1 schieben und somit höchstens m Mismatches finden können. Bei Matches beginnt jeder Vergleich in T entweder am letzten Zeichen des letzten Vergleichs, wenn es einen Mismatch gab, oder am Zeichen rechts davon, wenn es ein vollständiger Match war. Somit wird kein Zeichen in T mehr als einmal positiv gematcht, und wir erreichen O(m) Zeit für die Suche. Das Preprocessing lässt sich erneut auf Z-Boxen zurückführen, die den längsten Substring für jede Position i beschreiben, der auch Präfix ist. Da wir Suffixe von Teilstrings von P suchen, die auch Präfix sind, müssen dies die Z-Boxen an der Stelle $j = i\text{-}sp(i)'+1$ sein. Also ist j die am weitesten links von i stehende Position mit $i=j+Z(j)-1$, und $sp(i)'=Z(j)$ wenn j existiert, sonst 0, denn Z(j) ist gerade das längste Suffix von P[1..i] das auch Präfix ist laut Definition von Z-Boxen (und wenn kein solches j existiert matcht auch kein Suffix mit einem Präfix). Die Berechnung von Z-Boxen erfolgt in O(n), ebenso wie die der sp(i)'-Werte, und zusammen mit der Komplexität der Suche ergibt sich für den gesamten Algorithmus eine Komplexität von O(m+n).

6. Approximatives Stringmatching

Das exakte Stringmatching ist für die Bioinformatik zwar sehr wichtig, doch nicht immer will man ein genaues Vorkommen eines Pattern finden. Es kann durchaus interessanter sein, nach Unterschieden zu suchen, wenn man zum Beispiel zu einem gegebenen Exon ähnliche Abschnitte in einer Datenbank finden will, oder für ein Gen eine Sequenz sucht, die einen Abschnitt enthält der einem Abschnitt des Gens ähnelt. Hierbei sprechen wir von lokalem Alignment, im Gegensatz zu einem globalen Alignment, bei dem zwei Strings komplett miteinander verglichen und auf Ähnlichkeit getestet werden. Die meist genutzten Implementationen von approximativen Stringmatching sind sicher BLAST (Basic Local Alignment Search Tool) [7] und FASTA (Fast-All) [8], die beide auf der Grundannahme der Bioinformatik beruhen, dass hohe Ähnlichkeit der Sequenzen in der Regel ähnliche Funktion bzw. Struktur bedeutet. Bei Proteinen wird die biochemische Aktivität hauptsächlich durch die dreidimensionale Faltung und das Vorkommen von bestimmten Aminosäuren an bestimmten Stellen bestimmt. Aber nicht alle Aminosäuren sind gleichwichtig für die Struktur, und Abweichungen von der üblichen Sequenz verändern die Funktion oftmals nicht. Wenn allerdings doch eine veränderte Funktion auftritt, so geschieht diese Evolution meist in sehr kleinen Schritten, und wir können diese über Sequenzähnlichkeiten zurückverfolgen (z.B. für die Stammbaumberechnung). Im Zentrum des Interesses der Bioinformatik stehen also die Zusammenhänge der biologischen Funktionen, welche eng mit der jeweiligen Sequenz zusammen hängen. Da die Bestimmung der Funktion allerdings extrem aufwendig oder gar unmöglich ist, konzentrieren wir uns auf Sequenzähnlichkeiten, für die im Nachfolgenden verschieden leistungsstarke Algorithmen vorgestellt werden.

6.1 Dotplot

Zunächst müssen wir uns allerdings darüber klar werden, was „ähnlich“ eigentlich bedeutet, welche Matches besser sind als andere und wie wir die Güte von Matches messen können. Eine erste Idee wäre zu untersuchen, wie sehr man die eine Sequenz verändern muss um die andere zu erzeugen. Hierzu müssten wir zuerst herausfinden, wo die beiden Sequenzen gleiche Teilstrings enthalten, und wo wir am wenigsten Veränderungen vornehmen müssen. Wir werden uns also für zwei „ähnliche“ Strings einen Dotplot anlegen, mit dessen Hilfe wir leicht gleiche Teilstrings herausfiltern können. Ein Dotplot ist eine Matrix D, deren Spalten den Zeichen von String A und Zeilen den Zeichen von String B entsprechen. Wenn nun $A[a]=B[b]$ gilt, so ist $D[a,b]=1$, ansonsten 0. Gleiche Teilstrings sind so sehr schnell zu

erkennen, sie treten nämlich als diagonalen von links oben nach rechts unten im Dotplot hervor.

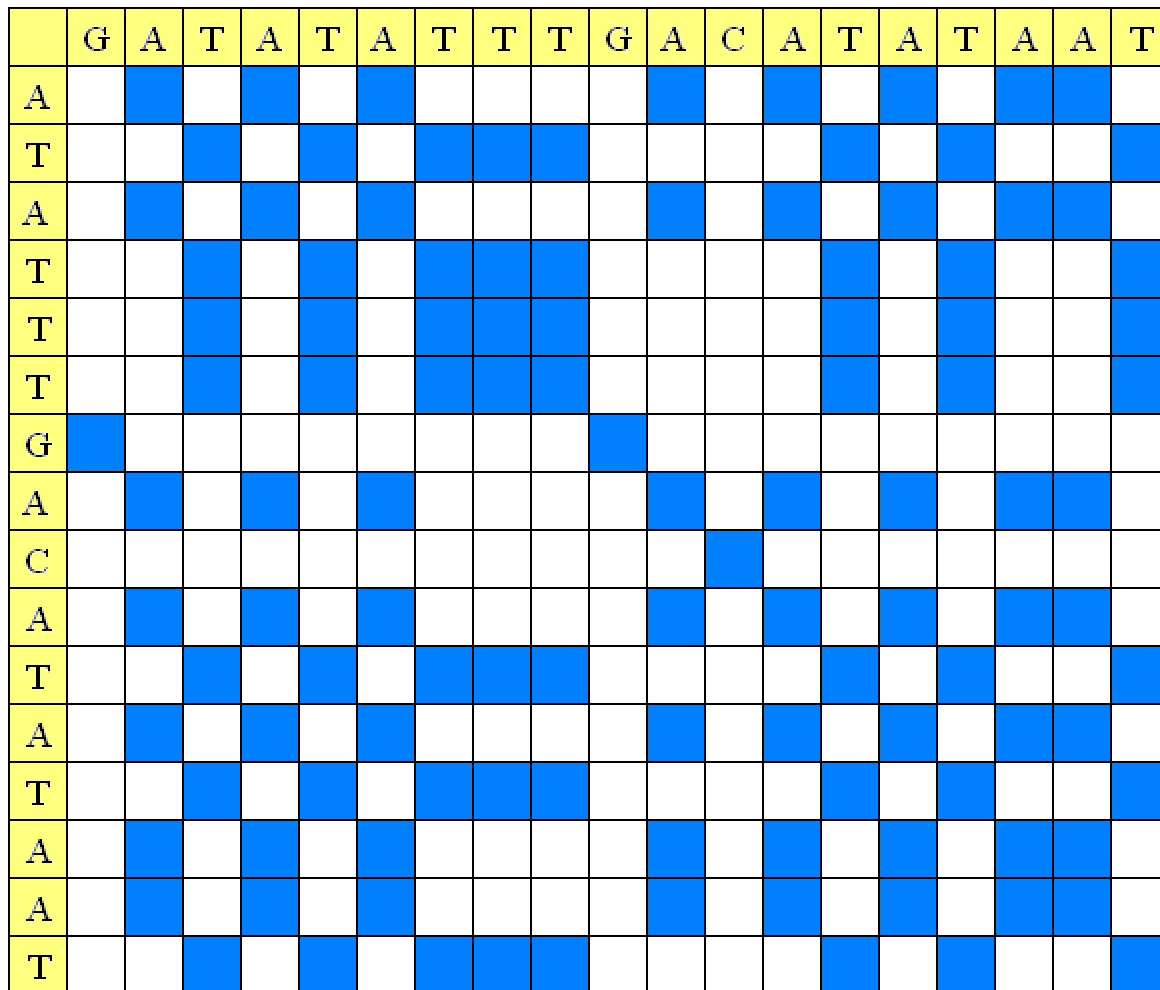


Fig.8: Dotplot

Liegt uns allerdings nur ein sehr kurzes Alphabet zu Grunde (wie es bei DNA-Betrachtungen der Fall ist), so bekommen wir ein starkes Rauschen in unserer Matrix, d.h. wir finden viel zu viele kurze Matches, die für unsere Betrachtungen keineswegs interessant sind. Um dies zu vermeiden, benutzt man ein Sliding Window der Länge l mit einem Schwellenwert z, um so nicht Zeichen für Zeichen zu vergleichen, sondern ganze Substrings zu betrachten. Wir setzen also nur eine 1 in die Matrix, wenn innerhalb des Fensters, welches wir über die beiden Strings schieben, mindestens z Zeichen übereinstimmen. So werden in der Visualisierung der Matrix kurze Matches verschwinden, und wichtige Regionen deutlicher sichtbar.

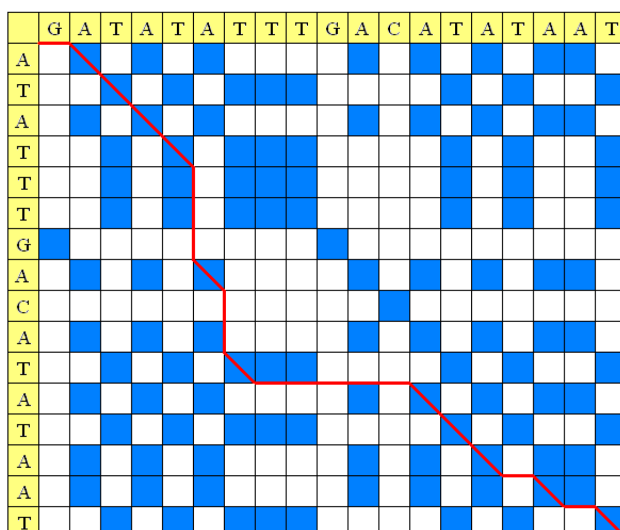
Wir wollen jedoch nicht nur gleiche Teilstrings finden, sondern wissen, wie ähnlich bzw. unterschiedlich sich die beiden Strings sind. Deshalb benötigen wir für die Regionen zwischen zwei Teilstrings ein geeignetes Abstandsmaß. Ein bekanntes Maß ist zum Beispiel der Hammingabstand, der den Unterschied zweier gleich langer Strings misst. Hierbei werden die Strings einfach miteinander verglichen, und die Anzahl der Mismatches ergibt das Ähnlichkeitsmaß. Für die Bioinformatik ist diese einfache Vorgehensweise jedoch unzulänglich, da wir hauptsächlich Strings vergleichen wollen, die sich durch evolutionäre Prozesse voneinander entfernt haben, sprich durch Basenaustausch, Baseneinfügung, Basenlöschung. Es kann also sein, dass sich die Strings nur durch eine einzige eingefügte

Base am Anfang unterscheiden, sie sind sich also sehr ähnlich, doch der Hammingabstand würde dies nicht reflektieren (z.B. A = GCGTAT... und B = AGCGTA... haben Hammingabstand 6). Um genauere und aussagekräftigere Ergebnisse erzielen zu können, werden wir für dieses Problem Edit-Skripte verwenden.

6.2 Edit-Skript

Ein Edit-Skript e ist eine Sequenz von Editieroperationen, die einen gegebenen String A in einen anderen String B überführt (e(a) = B). Dabei können vier Operationen verwendet werden: Einfügen („I“, Einfügen eines Zeichens, das in B auftaucht, in A dargestellt als Lücke), Löschen („D“, Löschen eines Zeichens in A, in B dargestellt als Lücke), Ersetzen („R“, ersetze ein Zeichen in A durch ein anderes Zeichen in B), Match („M“, die Zeichen in A und B an dieser Stelle stimmen überein). Mit Hilfe dieser vier Operationen lässt sich jeder String in jeden anderen umwandeln, ungeachtet deren Länge, jedoch gibt es natürlich für zwei feste Strings A und B beliebig viele Edit-Skripte (Bsp.: A = ATGTA und B = AGTGTC, e1 = MIMMMR (A’=A_TGTA und B’=AGTGTC), e2 = IRMMMDI (A’=_ATGTA_ und B’=AGTGT_C), etc...). Damit wir uns für eines der vielen möglichen Edit-Skripte entscheiden können, definieren wir uns die „Länge“ eines Skriptes als die Anzahl der Operationen I, R und D (Matches sind uninteressant, da wir die Unterschiede wissen wollen) und den „Editabstand“ als die Länge des kürzesten Edit-Skriptes für A und B (Dieses Maß ist auch als Levenshtein-Distanz bekannt, benannt nach dem russischen Mathematiker Wladimir I. Lewenstein).

Die Kombination aus Dotplot und Edit-Skript vereinfachen wir uns mit Pfaden durch den Plot. String A habe die Länge n und String B die Länge m, wir wollen also einen zusammenhängenden Pfad finden, der bei Position (1,1) startet und bei (n,m) endet und dabei nur Schritte nach rechts, nach unten oder diagonal nach rechts-unten durchführt. Wenn A die Zeilen und B die Spalten repräsentieren, dann bedeutet ein Schritt nach rechts das nächste Zeichen aus A zu schreiben und in B eine Leerstelle setzen, ein Schritt nach unten das nächste Zeichen aus B zu schreiben und in A eine Leerstelle setzen, und ein diagonaler Schritt aus beiden Strings das nächste Zeichen zu schreiben. Somit können wir jetzt die Güte eines Pfades als die Anzahl der diagonal durchquerten Felder, die eine „1“ enthalten, bestimmen, d.h. die maximale Güte die ein Pfad erzielen kann, ist min(n,m). Aufgrund der eben genannten Umformungen, kann man diesen besten Pfad dann auch sofort in ein Alignment mit optimalen Editabstand übersetzen.



GATAT___A__TTTGACATATAAT
_ATATTTGACAT__ __ _ATA _A_ T

Pfadgüte : 11

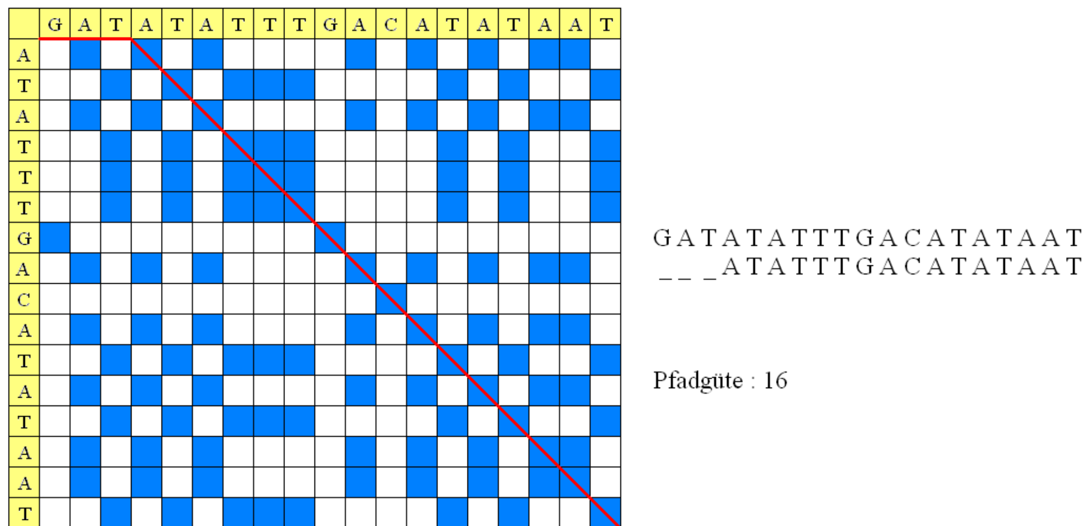


Fig.9: Zusammenhang von Edit-Skript und Dotplot

6.3 Globales Alignment

Die Anzahl der möglichen Pfade in einem Dotplot ist allerdings exponentiell hoch, weswegen wir eine bessere Methode benutzen müssen, als einfach alle Varianten durchzuprobieren. Hierfür werden wir ein rekursives Verfahren aus der dynamischen Programmierung anwenden, wobei Bottom-Up anstatt Top-Down vorgegangen wird. Historisch gesehen war der Needleman-Wunsch-Algorithmus der erste zu diesem Thema [6], doch aufgrund der sehr hohen Komplexität von $O(n^3)$ wird er nicht mehr angewendet, und wir betrachten daher eine effizientere Methode für globales Alignment. Die Funktion, die den Editabstand berechnen soll, sei dist(A,B), und benutze die Unterfunktion d(i,j), die den Editabstand von A[1..i] und B[1..j] berechnet (mit $0 \leq i \leq n$ und $0 \leq j \leq m$ und d(n,m)=dist(A,B)). Nehmen wir nun an, wir hätten schon ein optimales Edit-Skript für A[1..i'] und B[1..j'] mit $i' \leq i$, $j' \leq j$, aber nicht i'=i und j'=j gleichzeitig, dann können wir für den nächsten Schritt 4 Fälle unterscheiden: Bei einer Einfügung in A benutzen wir ein Zeichen mehr aus B, d.h. d(i,j-1) ist der bekannte Editabstand, und d(i,j)= d(i,j-1)+1, bei einer Einfügung in B benutzen wir ein Zeichen mehr aus A, d.h. d(i-1,j) ist der bekannte Editabstand, und d(i,j)=d(i-1,j)+1, bei einem Match ist d(i,j) =d(i-1,j-1), da ein Match keine Editierung ist und somit nichts kostet, und bei einem Mismatch benutzen wir aus beiden Strings das nächste Zeichen, wodurch d(i,j)=d(i-1,j-1)+1. In der Rekursion suchen wir das kürzeste Skript, das sich aus bereits bekannten ergibt, sprich das neue d(i,j) ist das Minimum der vier Fälle, mit Hilfe der Bedingungen für die Randbereiche d(i,0)=i und d(0,j)=j, also

$$d(i,j) = \min \begin{cases} \mathrm{d}(\mathrm{i},\mathrm{j}-1)+1 \\ \mathrm{d}(\mathrm{i}-1,\mathrm{j})+1 \\ \mathrm{d}(\mathrm{i}-1,\mathrm{j}-1)+s(i,j), s(i,j) = \begin{cases} 1, Mismatch \\ 0, Match \end{cases} \end{cases}$$

Jeder Rekursionsschritt löst also 3 Aufrufe aus (ein Match braucht keine Berechnungen), weshalb der Algorithmus eine Komplexität von mindestens O(3^min(n,m)) besitzt – also durchaus Optimierungspotenzial. Denn aufgrund der Rekursionsgleichung werden viele der Zwischenlösungen mehrmals berechnet, d.h. es gibt nur (n+1)*(m+1) verschiedene Aufrufe. Die restlichen Berechnungen sind alle redundant, und wir können sie uns sparen indem wir die Teillösungen in einer Tabelle speichern und bei Bedarf wieder

verwenden. Diese Tabelle wird ähnlich aufgebaut wie ein Dotplot, allerdings mit einer extra Zeile und extra Spalte am Anfang für die leeren Strings, die mit festen Werten initialisiert werden. So lässt sich Schritt für Schritt jeder Wert aus denen in der Zeile darüber, in der Spalte daneben und diagonal darüber bestimmen, und das für jedes Feld einmal, mit Komplexität O(m*n). Damit ist jetzt aber nur der Abstand von A und B berechnet in Feld (n,m), um auch das Alignment ablesen zu können müssen wir uns Pointer auf das minimale Vorgängerfeld behalten. Jeder zusammenhängende Pfad von (n,m) nach (1,1) ist damit ein optimales Alignment mit den oben genannten Übersetzungen. Die Komplexität des Traceback hat eine Worst-Case Größe von O(m+n), da es mindestens einen Pfad geben muss, jede Zelle mindestens einen Pointer hat (sonst wäre sie nicht berechnet worden), und keiner aus der Tabelle hinaus zeigt. Beide Schritte zusammen (Aufbau der Tabelle und Traceback) haben somit eine Komplexität von O(m*n) (wenn m*n > m+n).

		G	A	T	A	T	A	T	T	T	G	A	C	A	T	A	T	A	A	T
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	2	2	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	3	3	2	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	4	4	3	2	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	5	5	4	3	3	2	2	2	3	4	5	6	7	8	9	10	11	12	13	14
T	6	6	5	4	4	3	3	2	2	3	4	5	6	7	8	9	10	11	12	13
G	7	6	6	5	5	4	4	3	3	3	3	4	5	6	7	8	9	10	11	12
A	8	7	6	6	5	5	4	4	4	4	4	3	4	5	6	7	8	9	10	11
C	9	8	7	7	6	6	5	5	5	5	5	4	3	4	5	6	7	8	9	10
A	10	9	8	8	7	7	6	6	6	6	6	5	4	3	4	5	6	7	8	9
T	11	10	9	8	8	7	7	6	6	6	7	6	5	4	3	4	5	6	7	8
A	12	11	10	9	8	8	7	7	7	7	7	7	6	5	4	3	4	5	6	7
T	13	12	11	10	9	8	8	7	7	7	8	8	7	6	5	4	3	4	5	6
A	14	13	12	11	10	9	8	8	8	8	8	8	8	7	6	5	4	3	4	5
A	15	14	13	12	11	10	9	9	9	9	9	8	9	8	7	6	5	4	3	4
T	16	15	14	13	12	11	10	9	9	9	10	9	9	9	8	7	6	5	4	3

Fig.10: Globales Alignment

Diesem Modell liegt allerdings die Annahme zu Grunde, dass Matches umsonst sind, und alle Editieroperationen gleich teuer sind. Doch solch eine Vereinfachung lässt sich nur schwer in der Praxis der Bioinformatik wieder finden, denn manche Aminosäuren sind sich ähnlicher als andere, so dass Vertauschungen unterschiedlich starke Wirkungen haben. Auch ist es meistens der Fall, dass ein Einfügen/Löschen sehr viel größere Auswirkungen mit sich bringt als ein simples Ersetzen, denn wenn plötzlich eine Base mehr oder weniger vorhanden ist, verschiebt sich bei der Transkription das gesamte Leseraster – es kommt zu einem

Frameshift wodurch auch alle nachkommenden Reads verschieben und somit andere Ergebnisse liefern könnten. Daher benötigt man in der Berechnung der Gewichte flexiblere Bewertungsschemata, die man gegebenenfalls an die jeweilige Aufgabe anpassen kann. Jede der vier Operationen benötigt also ein eigenes Gewicht, c(i) für Einfügen, c(d) für Löschen, m für Match und r(x,y) für Replace, mit x=A[i] und y=B[j]. Die Funktion r kann man am besten einer Substitutionsmatrix entnehmen, bekannteste Beispiele hierfür sind die PAM (Point Accepted Mutation) [9] und BLOSUM (BLOcks of Amino Acid SUbstitution Matrix) Matrizen [10], man sollte nur darauf achten, dass $r(x,y) < c(i)+c(d)$, da Einfügen und Löschen hintereinander ein Replace darstellen. Integriert in den Algorithmus werden diese Verfeinerungen ganz leicht, für die Initialisierung fügen wir $d(i,0) = i * c(d)$ und $d(0,j)=j*c(i)$ hinzu, bei der Berechnung der Felder jeweils $d(i,j-1)+c(i)$ bzw. $d(i-1,j)+c(d)$ oder $d(i-1,j-1)+r(i,j)$, um so die unterschiedlichen Gewichte zu berücksichtigen.

6.4 Scoringfunktion

Wir sind nun in der Lage, für zwei Strings Alignments mit minimalem Abstand zu finden, doch eigentlich suchten wir nach der Ähnlichkeit. Logisch gesehen sind diese beiden Größen äquivalent, denn wenn wir den Abstand minimieren, dann maximieren wir die Ähnlichkeit. Betrachten wir das Problem nun von der anderen Seite, und maximieren direkt die Ähnlichkeit, können wir also äquivalent vorgehen, und bewerten ähnliche Zeichen mit positiven Werten und unähnliche mit negativen Werten. Diese Scoringfunktion s können wir wiederum einer Tabelle entnehmen (siehe Bsp. links), die sozusagen die Gewichte implizit enthält, und formulieren dann „Ähnlichkeit“ als $sim(A,B)=\sum s(A[i],B[i])$. Die Initialisierung wird wieder leicht verändert mit $d(i,0)=\sum s(A[i],_)$ und $d(0,j)= \sum s(_,B[j])$, und in der Berechnung der Felder werden die Gewichte ersetzt durch ihre entsprechenden Einträge in der Tabelle, nämlich $c(i)= s(_,B[j])$, $c(d)= s(A[i],_)$ und $r(i,j)= s(A[i],B[j])$. Zu beachten ist hierbei nur, dass wir dieses Mal nicht den minimalen Wert suchen, sondern den maximalen Wert der Berechnung auswählen.

	A	C	G	T	_
A	4	-2	-2	-2	0
C	-2	4	-2	-2	-2
G	-2	-2	4	-1	0
T	-2	-2	-1	4	-2
_	0	-2	0	-2	

6.5 Lokales Alignment

Globale Alignments mittels dynamischer Programmierung zu finden benötigt also quadratischer Komplexität, da zuerst immer eine Tabelle aufgebaut wird, und dann der optimale Pfad ausgelesen wird. Zudem können wichtige biologische Ereignisse bei der Betrachtung der Ähnlichkeit berücksichtigt werden, wie zum Beispiel Mutationswahrscheinlichkeiten, Aminosäureähnlichkeiten oder das Evolutionsmodell. Allerdings benötigt das Anlegen dieser Tabelle gleichsam auch quadratischen Speicherplatz, was bei den Mengen an Daten, die für die Anwendungen in der Bioinformatik anfallen können, von großem Nachteil sein kann. Es sei an dieser Stelle darauf verwiesen, dass es Algorithmen für die gleiche Aufgabe mit linearem Platzbedarf gibt, aus Komplexitätsgründen aber darauf verzichtet wird, diese ebenfalls vorzustellen. Denn nicht immer will man ein komplett globales Alignment finden, liegen zum Beispiel verschiedene Strings als Ergebnis einer Shotgun-Sequenzierung vor, so reicht es lediglich die optimalsten lokalen Alignments zu suchen. Hierzu betrachtet man die End-Free Ähnlichkeiten, das heißt Leerzeichen am Anfang und am Ende der Strings sind umsonst. Wir bauen also wiederum eine Tabelle auf, mit den Initialisierungen $d(i,0)=d(0,j)=0$ (wir können also am Anfang kostenlos beliebig viele

Leerzeichen setzen), und verfolgen den Pfad nicht von (n,m) aus zurück, sondern vom maximalen Wert in der letzten Zeile und Spalte (denn von dort aus bis (n,m) können auch kostenlos beliebig viele Leerzeichen folgen). Dadurch kann man ein Alignment finden, das lokal (zum Beispiel in der Mitte) optimal matcht, dafür am Anfang und Ende nur Leerzeichen hat.

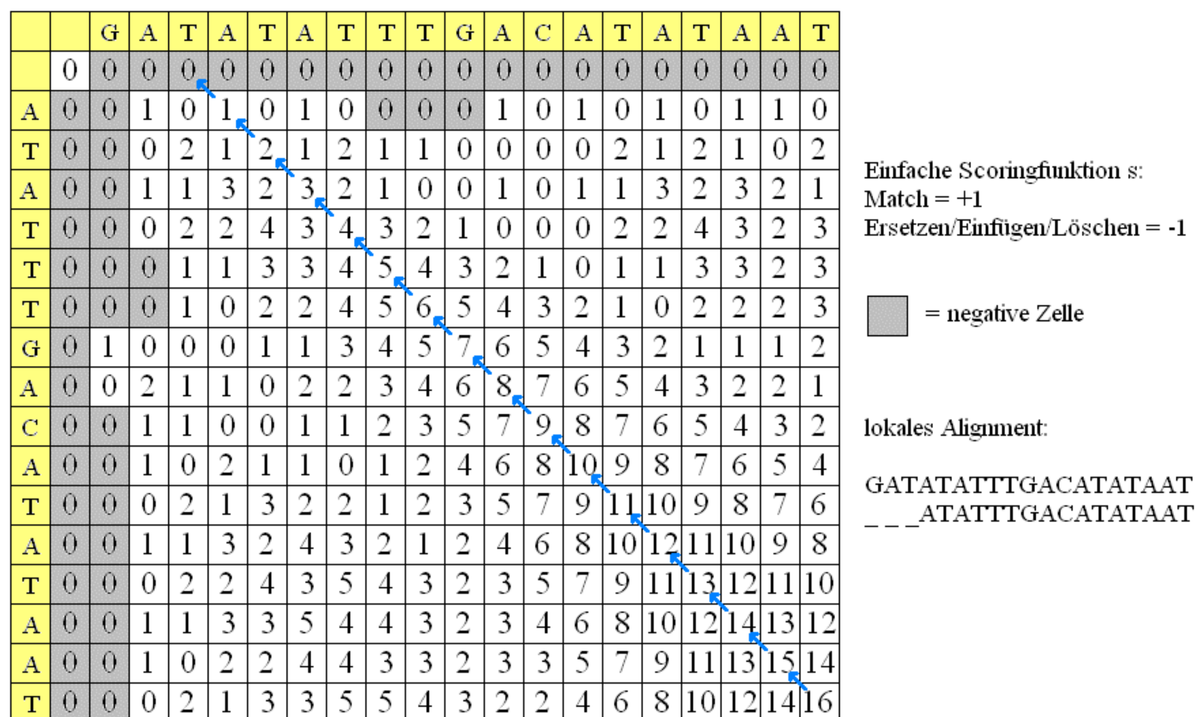
Doch lokales Alignment findet nicht nur bei der Shotgun-Sequenzierung seine Anwendung, auch in anderen Gebieten der Biologie ist ein globales Alignment oft unzureichend genau, zum Beispiel wenn durch die Evolution ganze Blöcke von Teilsequenzen verschoben werden, und diese Blöcke bestimmte wichtige Funktionen haben (Gene, Exons, etc). Das Ziel hierbei ist also zu bemerken, dass zwei Teilstrings an verschiedenen Positionen exakt übereinstimmen, auch wenn die sie umgebenen Teile nur sehr ungenau matchen (was ein globales Alignment also übersehen würde), da hier die biologisch wichtige Information liegt. Wir formulieren uns also ein lokales Alignment als sim*(A,B)=max(sim(a,b)) über allen Teilstrings a aus A und b aus B. Würde man allerdings alle möglichen Teilstrings ($O(n^2)$) beider Strings in Paare aufteilen ($O(n^2)$) und dann immer das Alignment berechnen ($O(n^2)$) um das Maximum zu finden, wäre man bei einer Komplexität von O(n^6). Das können wir natürlich verbessern mit Hilfe des Smith-Waterman Algorithmus. [11]

6.6 Smith-Waterman Algorithmus

Dieser Algorithmus wird das Substringpaar mit maximaler Ähnlichkeit finden und benutzt dafür eine Scoringfunktion mit positiven Werten für Matches und negativen Werten für alles andere. Jedes Mal, wenn wir in einem Pfad viele Matches finden, wird der Score also größer, ansonsten wieder kleiner, aber solange er noch positiv ist suchen wir weiter. Pfade, deren Score negativ wird, können sofort als unbrauchbar abgetan werden. Seien a und b echte Präfixe von A und B mit a=A[1..i] und b=B[1..j], dann suchen wir das maximale sim(a',b'), wobei a' und b' Suffixe von a und b sind. Um dies etwas abzukürzen bezeichnen wir den maximalen Score als v(i,j)=max(sim(a',b')). Das bedeutet aber, dass sim*(A,B)=max(v(i,j)), somit können wir das lokale Alignment berechnen, wenn wir das lokale Suffixalignmentproblem lösen können. Dieses Problem lässt sich wieder ähnlich berechnen wie bisher, wir benutzen unsere Scoringfunktion um für jedes Feld der Tabelle das Maximum der drei Berechnungen zu finden, mit dem Unterschied, dass wir eine 0 setzen, wenn alle drei Werte negativ sind. So bleibt $v(i,j)\geq 0$ für alle i und j erhalten, und für ein bestimmtes Feld ergibt sich

$$v(i,j) = \max \begin{cases} 0 \\ v(i,j-1) + s(_,B[j]) \\ v(i-1,j) + s(A[i],_) \\ v(i-1,j-1) + s(A[i],B[j] \end{cases}$$

analog zum Vorgehen beim globalen Alignment. Der Traceback jedoch verläuft nun anders, er startet beim maximalen Wert der gesamten Matrix, also nicht zwingend am Rand, und verfolgt den Weg nur bis zu einer Zelle mit dem Wert 0. Er kann also mitten in der Matrix anfangen und wieder aufhören, denn genau dieses Teilstück ist dann das optimal Ähnliche lokale Alignment, und arbeitet trotzdem wie globales Alignment, sprich mit quadratischer Komplexität O(n*m).



		G	A	T	A	T	A	T	T	T	G	A	C	A	T	A	T	A	A	T
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	1	1	0
T	0	0	0	2	1	2	1	2	1	1	0	0	0	0	2	1	2	1	0	2
A	0	0	1	1	3	2	3	2	1	0	0	1	0	1	1	3	2	3	2	1
T	0	0	0	2	2	4	3	4	3	2	1	0	0	0	2	2	4	3	2	3
T	0	0	0	1	1	3	3	4	5	4	3	2	1	0	1	1	3	3	2	3
T	0	0	0	1	0	2	2	4	5	6	5	4	3	2	1	0	2	2	2	3
G	0	1	0	0	0	1	1	3	4	5	7	6	5	4	3	2	1	1	1	2
A	0	0	2	1	1	0	2	2	3	4	6	8	7	6	5	4	3	2	2	1
C	0	0	1	1	0	0	1	1	2	3	5	7	9	8	7	6	5	4	3	2
A	0	0	1	0	2	1	1	0	1	2	4	6	8	10	9	8	7	6	5	4
T	0	0	0	2	1	3	2	2	1	2	3	5	7	9	11	10	9	8	7	6
A	0	0	1	1	3	2	4	3	2	1	2	4	6	8	10	12	11	10	9	8
T	0	0	0	2	2	4	3	5	4	3	2	3	5	7	9	11	13	12	11	10
A	0	0	1	1	3	3	5	4	4	3	2	3	4	6	8	10	12	14	13	12
A	0	0	1	0	2	2	4	4	3	3	2	3	3	5	7	9	11	13	15	14
T	0	0	0	2	1	3	3	5	5	4	3	2	2	4	6	8	10	12	14	16

Fig.11: Smith-Waterman Algorithmus

Während globales Alignment beide Sequenzen komplett erfasst und somit für die Charakterisierung von Proteinfamilien oder später für Multiples Sequenz Alignment benötigt wird, findet lokales Alignment optimale Teilsequenzen und kann somit biologisch interessante Regionen wie Exons oder funktionale Subsequenzen aufspüren, oder dient als End-Free Alignment zur Assembly von Shotgun-Sequenzen. Auch gewichtete Editabstände können zur Auffindung von Alignments benutzt werden, somit ist die Wahl des Modells stets abhängig von der jeweiligen Aufgabe.

6.7 Multiples Sequenz Alignment

Bisher haben wir aber immer noch immer nur zwei Strings mit einander verglichen, doch wenn man sehr viele Proben optimal alignieren will, würde es undenkbar aufwendig werden jede einzelne Paarung zu optimieren und dabei gleichzeitig über der gesamten Menge optimal zu bleiben. Betrachten wir nun also, wie man multiple Stringvergleiche mit k>2 Strings durchführen kann. Solche Multiplen Sequenz Alignments (MSA) mit k Strings sind Tabellen mit k Zeilen, in denen jeweils einer der Strings mit beliebig eingefügten Leerzeichen steht in der Art, dass in jedem Feld nur ein Zeichen vorkommt, und keine Spalte nur aus Leerzeichen besteht. Das optimale Ziel wird hierbei schnell klar, nämlich die Anzahl an Leerzeichen und somit an Spalten zu minimieren, und dabei trotzdem eine hohe Übereinstimmung der Sequenzen zu finden, d.h. wir suchen die größten Gemeinsamkeiten verschiedener Sequenzen. Die biologische Motivation ist hierbei genau invertiert zum Alignment von nur zwei Strings, wo wir herausfinden wollten, welche zwei Sequenzen sich am ähnlichsten sind, um so auf ähnliche Struktur und gleiche Funktion schließen zu können. Beim MSA hingegen geben wir viele Sequenzen hinein, von denen wir bereits aufgrund gleicher Funktion ähnliche Struktur vermuten und wollen wissen, was die gemeinsamen Elemente aller Proben sind, um so die Ursache für die Funktion finden zu können und diese auch in anderen Sequenzen vorhersagen zu können. Würde man sich für diese Suche auf paarweise Vergleiche verlassen, könnte man diese funktionalen Blöcke meistens nicht von

zufälligen Übereinstimmungen oder vom „Rauschen" unterscheiden, doch wenn sie in vielen Sequenzen an der gleichen Stelle auftreten kann man den Zufall relativ sicher ausschließen, denn unter der Annahme, dass alle betrachteten Sequenzen den gleichen genetischen Ursprung haben und sich diese Blöcke trotz verschiedener evolutionärer Vorgänge in jeder Sequenz nicht verändert haben, kann man davon ausgehen, dass sie für die Funktion von entscheidender Bedeutung sind.

Das Vorgehen bei MSA unterscheidet sich zu normalem Alignment dadurch, dass wir nicht mehr so leicht für jedes Alignment einen Score vergeben können. Bei nur zwei Strings ist klar, ob es ein I, ein D oder ein R ist, aber bei mehreren Zeichen untereinander lässt sich das nicht mehr sagen. Auch eine Substitutionsmatrix ist nicht mehr effizient genug, denn bei k Sequenzen über dem Alphabet ∑ würde sie O(|∑|^(k+1)) Einträge besitzen. Zwei Auswege hierfür sind zum Beispiel der Sum-of-Pairs Score, der die Summe aller paarweisen Sequenzen maximiert, und der Center-Star Score, der die Summe der Alignments jeder Sequenz zu einer Consensussequenz maximiert. Widmen wir uns zunächst dem Sum-of-Pairs Score (SP-Score), und definieren das induzierte Alignment für zwei der Sequenzen S(i) und S(j) aus dem MSA M als eine zweizeilige Tabelle M', in der alle Spalten aus M entfernt wurden, die in Zeile i und j ein Leerzeichen enthalten. Der SP-Score ist also die Summe aller Alignmentscores der durch M induzierten paarweisen Alignments, und das SP-Alignment Problem sucht das MSA mit minimalem SP-Score. Die Berechnung eines solchen SP-Score für ein gegebenes MSA M ist sicher einfach, und kann mit einer Komplexität von $O(l*k^2)$ gelöst werden, wobei k die Zeilen und l die Spalten des MSA sind. Doch wir wollen das MSA mit minimalem SP-Score finden, und das erfordert noch einige zusätzliche Berechnungen.

Wir erinnern uns an die Funktionen dist und d, und erweitern diese nun um weitere Dimensionen. Gezeigt wird dies für 3 Strings S1, S2 und S3 mit Laufvariablen i, j und k, bei mehr als drei erfolgt die Erweiterung analog. Wir benötigen drei Kostenvariablen c(i,j), c(i,k) und c(j,k), die jeweils 0 sind bei einem Match bei S1[i]=S2[j] bzw. S1[i]=S3[k] oder S2[j]=S3[k], ansonsten den Kosten für Einfügen, Löschen, Ersetzen entsprechen. Damit können wir d(i,j,k) berechnen als das Minimum von d(i-1,j,k)+2, da wir S1 zweimal mit Leerzeichen alignieren, bzw. d(i,j-1,k)+2 oder d(i,j,k-1)+2, und d(i-1,j-1,k)+2+c(i,j), da wir S1 und S2 mit einem Leerzeichen alignieren, bzw. d(i-1,j,k-1)+2+c(i,k) oder d(i,j-1,k-1)+2+c(j,k), und d(i-1,j-1,k-1)+c(i,j)+ c(i,k)+ c(j,k). Initialisiert wird der Vorgang hierbei ebenfalls in drei Dimensionen mit d(0,0,0) = 0, d(i,j,0)=d1,2(i,j)+(i+j), d(i,0,k)=d1,3(i,k)+(i+k) und d(0,j,k)=d2,3(j,k)+(j+k). In der gleichen Weise lässt sich die Rechnung auf beliebig viele Dimensionen bzw. Strings erweitern, und der Beweis der Korrektheit erfolgt stets analog zum paarweisen Alignment. Die Komplexität des SP-Score ist allerdings unpraktisch hoch, denn für k Sequenzen der Länge n hat der Hyperwürfel n^k Zellen, für jede Zelle sind (2^k)-1 Berechnungen notwendig, nämlich alle Ecken des k-dimensionalen Würfels minus die eine, die gerade berechnet wird, und das ergibt zusammen die Klasse O(2^k * n^k), also ist das allgemeine SP-Score Problem NP-Vollständig.

Um das MSA für mehr als nur eine handvoll Sequenzen doch noch berechnen zu können, betrachten wir jetzt eine andere Zielfunktion, das Center-Star Verfahren. Es wird zunächst eine Konsensussequenz S(c) gesucht, die den kleinsten durchschnittlichen Abstand zu allen anderen Sequenzen hat. S(c) wird im Normalfall aus einer der k Sequenzen ausgewählt, um sich so die Konstruktion einer neuen Sequenz zu sparen. Mit S(c) als Kern des MSA M (das Zentrum des Sterns) wählen wir nun der Reihe nach eine noch nicht alignierte Sequenz T, und alignieren M und T bis alle k Sequenzen in M enthalten sind. Für die Fehlergrenze dieses Verfahrens gilt $d/d^* \leq 2 - 1/k < 2$, mit d als SP-Score des fertigen MSA M, und d* als der optimale SP-Score. [12]

S1 = A T G G C
S2 = A G C C
S3 = T G C G A T
S4 = G C A T G
S5 = T G C C T A
S6 = C A A C TA

	S1	S2	S3	S4	S5	S6
S1	0	2	4	4	4	5
S2	2	0	4	4	3	4
S3	4	4	0	3	3	5
S4	4	4	3	0	3	4
S5	4	3	3	3	0	3
S6	5	4	5	4	3	0
Durchschnitt	3,8	3,4	3,8	3,6	3,2	4,2

- Wähle S5 als Kern
- Aligniere S3 mit Kern
  TGCC_TA
  TGCGAT_
- Aligniere S2 mit Kern
  TGCC_TA
  TGCGAT_
  AGCC_ _ _
- Aligniere S1 mit Kern
  _ TGCC_TA
  _ TGCGAT_
  _ AGCC_ _ _
  ATGGC_ _ _
- Aligniere S4 mit Kern
  _ TGCC_TA
  _ TGCGAT_
  _ AGCC_ _ _
  ATGGC_ _ _
  _ _ GC_ATG
- Aligniere S6 mit Kern
  _ TGCC_TA
  _ TGCGAT_
  _ AGCC_ _ _
  ATGGC_ _ _
  _ _ GC_ATG
  CA AC__TA

6.8 Profil Alignment

Jetzt kennen wir also ein komplettes MSA für k ähnliche Sequenzen, nun wollen wir es auch dazu benutzen, nach neuen ähnlichen Sequenzen zu suchen. Wir müssen also entscheiden können, wie gut eine andere Sequenz zu unserem MSA passt. Dazu benutzen wir ein Profil P als eine Tabelle der Größe n*|∑'|, wobei n die Anzahl der Spalten von M ist, und ∑' das zugrunde liegende Alphabet ∑ erweitert um das Leerzeichen. In der Zelle (i,j) soll dann die relative Häufigkeit des Zeichens j in der Spalte i aus dem MSA stehen, um so die Ähnlichkeit einer Sequenz zu dem MSA bewerten zu können. Im Grunde läuft diese Idee wieder auf ein Alignment hinaus, nämlich welche Stelle der neuen Sequenz mit welcher Spalte des MSA verglichen werden soll, und dafür benötigen wir eine Methode zur Bewertung eines Alignments mit dem Profil, und eine Methode zum Finden des besten Alignments. Im Prinzip können wir ähnlich vorgehen wie beim Alignment von zwei Strings, wir können also in der Sequenz S wieder Leerzeichen einsetzen, und im Profil P entsprechend leere Spalten. Den Score des Alignments A berechnen wir aus

$$\sum_{i=1}^{|A|} \sum_{c(k)\in\Sigma'} P'\left[c(k),i\right] * m[c(k),S'[i]]$$

wenn i keine leere Spalte ist, sonst m[_,S'[i]], wobei m unsere Substitutionsmatrix ist, und P' bzw. S' sich aus P und S mit zusätzlichen Leerstellen ergeben. Das optimale Alignment finden wir wie gehabt mittels dynamischer Programmierung, sei also c(x,j) der Alignmentscore des Zeichens x mit Spalte j aus P, dann gilt c(x,j)= ∑c(k) aus ∑' über P[c(k),j]*m[c(k),x]. Wir berechnen den Score des optimalen Alignments v(i,j) aus dem Maximum von v(i-1,j)+c(_,i), v(i,j-1)+m[_,j] und v(i-1,j-1)+c(S[j],i), wobei v der Score von den ersten i Spalten von P mit dem Präfix S[1..j] ist.

$$v(i,j) = \max \begin{cases} v(i-1,j) + c(_,i) \\ v(i,j-1) + m[_,j] \\ v(i-1,j-1) + c(S[j],i) \end{cases}$$

Beispiel

S_1:	A	G	C	_	A
S_2:	A	G	A	G	A
S_3:	A	C	C	G	_
S_4:	C	G	_	G	C

Substitutionsmatrix m

	A	G	C	_
A	2	-1	-3	-1
G		2	-1	-1
C			2	-1

Profil

	1	2	3	4	5
A	0.75	0	0.25	0	0.50
G	0	0.75	0	0.75	0
C	0.25	0.25	0.50	0	0.25
T	0	0	0	0	0
_	0	0	0.25	0.25	0.25

S_5: AAGGC

Alignment

P	1	_	2	3	4	5
S_5	A	A	G	_	G	C

S(A) = (2*0.75 + -1*0 + -3*0.25 + -1*0) +
(-1) +
(-1*0 + 2*0.75 + -1*0.25 + -1*0) +
(-1*0.25 + -1*0 + -1*0.50 + -1*0.25) +
(-1*0 + 2*0.75 + -1*0 + -1*0.25) +
(-3*0.50 + -1*0 + 2*0.25 + -1*0.25)
= 0

Fig.12: Alignment eines MSA und eines Strings mittels Profil

7. Diskussion

Es wurden nun mehrere Verfahren für exaktes und approximatives Stringmatching vorgestellt, von denen jedes mit unterschiedlicher Komplexität in Zeit und Platzbedarf abläuft. Ein Vergleich der einzelnen Methoden soll deren Tauglichkeit in bestimmten Situationen aufzeigen:

Exaktes Stringmatching	Naiver Algorithmus	Z-Box Algorithmus	Boyer-Moore Algorithmus	Knuth-Morris-Pratt Alg.
Preprocessing Suche Gesamt	- - O(m*n)	O(m+n) O(m) O(m+n)	O(n) O(m) O(m+n)	O(n) O(m) O(m+n)
Größe Alphabet	Praktisch unabhängig von Alphabetgröße	Praktisch unabhängig von Alphabetgröße	Je größer desto besser (BCR führt zu großen Sprüngen, greift selten bei kleinen Alphabeten)	Praktisch unabhängig von Alphabetgröße
Bemerkung	Schlechteste Komplexität	Average und Worstcase Komplexität gleich	Best Case ist O(m/n)	Average und Worstcase Komplexität gleich, erweiterbar auf mehrere Pattern

Die Erweiterung auf mehrere Pattern wird durch Keywordtrees und Suffixtrees mittels des Algorithmus von Ukkonen gelöst, und die Optimierung des Speicherplatzes durch Suffixarrays, was den Umfang dieser Arbeit allerdings überschritten hätte.

Approximatives Stringmatching	Globales Alignment	Lokales Alignment (Smith-Waterman)	MSA
Komplexität	O(m*n)	O(m*n)	SP: O(2^k * n^k) CS: O(m*n)
Platzbedarf	O(m*n)	O(m*n)	O(m*n)
Bemerkung	- Verwendet z.B. für Charakterisierung von Proteinfamilien - Grundlage für Multiples Sequenz Alignment	- findet optimale Teilsequenzen - kann biologisch interessante Regionen aufspüren - End-Free Alignment zur Assembly von Shotgun-Sequenzen	- Vergleich mehrere Strings - soll Ursache für Funktion finden und in anderen Sequenzen vorhersagen

Außer den hier vorgestellten Algorithmen gibt es noch weitere, die Lücken in Sequenzen extra behandeln (längere Lücken müssen nicht mehrere Evolutionsschritte bedeuten), den Platzbedarf linear halten (mittels Teilpfaden und K-Band Algorithmus), oder andere heuristische Verfahren zum Auffinden von wichtigen Regionen, wie zum Beispiel Genen (unter Verwendung von Hidden Markov Modellen).

Aus dieser Aufstellung ist also zu entnehmen, dass die verschiedenen Methoden für exaktes Stringmatching die gleiche Aufgabe unter unterschiedlichen Bedingungen (Alphabeten) optimal lösen können, die Methoden für approximatives Stringmatching allerdings für unterschiedliche Aufgaben herangezogen werden.

Auch in der Praxis finden die gezeigten Algorithmen immer noch Anwendung. So arbeitet ClustalW zum Beispiel mit MSAs [13], HMMer mit Hidden Markov Modellen [14] und BLAST sucht Pattern in mehreren Templates innerhalb großer Sequenzdatenbanken. Selbst wenn der Anwender meist nicht mitbekommt, wie die Programme zu ihren Ergebnissen gelangen, findet ClustalW den besten Match eines MSA und ein aussagekräftiges Alignment mit den hier vorgestellten Methoden. BLAST hingegen ist in der Bioinformatik schon fast alltäglich geworden wenn es darum geht, lokale Alignments zu finden und diese mit Datenbanken von bereits vorhandenen Sequenzen zu vergleichen. Selbst andere Anwendungen wie der UCSC Genome Browser benutzen Abwandlungen davon wie BLAT (BLAST-Like Alignment Tool) [15], das Ähnlichkeiten zwischen DNA-Sequenzen und Proteinsequenzen finden kann, wobei es durch die Berechnung überlappender K-Mere viel schneller als BLAST ist.

8. Referenzen:

[1] Initial sequencing and analysis of the human genome; *Nature* 2001,409, 0028-0836
[2] Toscano,W.A.;Oehlke,K.P.; *Int.J.Environ.Res.PublicHealth* 2005,2,4–9
[3] Quackenbush,J.;Liang,F.;Holt,I.;Pertea,G.;Upton,J.; *Nucl.AcidsRes.* 2000,28,141-145
[4] Pribnow,D.; *Proc.Natl.Acad.Sci.U.S.A.* 1975, 72,784-788.
[5] Boyer,R.S.;Moore,J.S. *A Fast String Searching Algorithm;* Communications of the ACM, 1977
[6] Needleman,S.B.;Wunsch,C.D.; *J. of Molecular Biology* 1970, 48
[7] Altschul,S.F.;Gish,W;Miller,W.; *Journal of Molecular Biology* 1990,215,403-410.
[8] Lipman,D.J.;Pearson,W.R.; *Science* 1985,227,1435-1441.
[9] Dayhoff,M.O.;Schwartz,R.;Orcutt,B.C.; *Nat. Biomed. Res. Found.*1978, 345-358
[10] Henikoff,S.; *Proc.Natl.Acad.Sci.US.A.* 1992,89,10915–10919
[11] Smith,T.F.;Waterman,M.S.; *J. Mol. Bio* 1981, 147
[12] Gusfield,D.; *Algorithms on Strings, Trees, and Sequences;* Cambridge University Press, 1997
[13] Larkin,M.A.; *Bioinformatics* 2007,23,2947-2948.
[14] *Biosequence analysis using profile hidden Markov models;* Zu finden unter <http://hmmer.janelia.org/> [30.8.2008]
[15] Kent,W.J.; *Genome research* 2002,12,656–64

9. Literaturverzeichnis:

- Leser,U. *Algorithmische Bioinformatik;* Zu finden unter <http://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/archive/ws0708/hk_algbio> [30.8.2008]
- Lesk,A.M. *Bioinformatik - Eine Einführung;* Spektrum Akademischer Verlag,2002
- Böckenhauer,H.J.;Bongartz,D. *Algorithmische Grundlagen der Bioinformatik;* Teubner,2003
- Koolman,J.;Röhm,K.H.;Wirth,J. *Taschenatlas der Biochemie;* Thieme Georg Verlag,2002
- Mount,D.W. *Bioinformatics: Sequence and Genome Analysis;* Cold Spring Harbor Laboratory Press,2004

- Merkl,R.;Waack,S. *Bioinformatik Interaktiv - Algorithmen und Praxis;* Wiley-VCH,2002
- Attwood,T.;Parry-Smith,D. *Introduction to Bioinformatics;* Pearson,1999

10. Anhang

Java-Code: Naiv

```
for (int i = 0; i<=(template.length() - pattern.length() + 1);i++) {
   match = true;
   j = 0;
   while (match && (j < pattern.length())) {
      if (template.charAt(i + j) != pattern.charAt(j))
         match = false;
      else
         j = j + 1;
      }
   if (match) System.out.print(i-1+", ");
}
```

Java-Code: Z-Box

```
while (match && (j<str.length())) {
   if (str.charAt(j) != str.charAt(j-1))
      match = false;
   else
      j = j + 1;
}
Z[1]=j-1;
if (Z[1]>0) {r=1+Z[1]-1;l=1;}
else {r=l=0;}
for (int k = 2 ;k<str.length();k++) {
   if (k>=r) {
      match =true;
      j=k;
      while (match && (j<str.length())) {
         if (str.charAt(j) != str.charAt(j-k))
            match = false;
         else
            j = j + 1;
         }
      Z[k]=j-k;
      if (Z[k]>0) {r=k+Z[k]-1;l=k;}
   }
   else {
      int kneu = k-l;
```

```
            int b = r-k+1;
            if (Z[kneu]<b) {
               Z[k] = Z[kneu];
            }
            else {
               match =true;
               j=r+1;
               int i=b+1;
               while (match && (j<str.length())) {
                  if (str.charAt(j) != str.charAt(i))
                     match = false;
                  else {
                     j = j + 1;
                     i=i+1;
                  }
               }
               Z[k] = j-k;
               r = j-1;
               if (j!=r+1) l = k;
            }
         }
      }

      Z = getBoxes(str);
      for (int i=pattern.length()+1;i<str.length();i++){
         if (Z[i]==pattern.length()) System.out.print(i-pattern.length()-1+", ");
      }
```

Java-Code: Boyer-Moore

```
      //invert pattern
      for (int i=pattern.length()-1;i>=0;i--) {
         invert[j] = pattern.charAt(i);
         j++;
      }
      String pneu = new String(invert);
      String str = pneu;
      //get Z-Boxes
      int[] Z = new int[str.length()];
      Z = zbox.getBoxes(str);
      //get N
      int[] N = new int[str.length()];
      j=0;
      for (int i=str.length()-1;i>=0;i--) {
         N[j] = Z[i];
         j++;
      }
      //find L'
      int[] L = new int[str.length()];
      for (int i=0;i<str.length();i++) L[i]=0;
      for (int i=0;i<str.length()-1;i++) {
```

```
    j = str.length()-N[i];
    if (j<str.length()) L[j]=i;
}
//compute R(x)
int[] R = new int[4];
for (int i=0;i<4;i++) R[i]=0;
for (int i=0;i<pattern.length();i++) {
    if (pattern.charAt(i)=='A') R[0]=i;
    if (pattern.charAt(i)=='C') R[1]=i;
    if (pattern.charAt(i)=='G') R[2]=i;
    if (pattern.charAt(i)=='T') R[3]=i;
}
int k=pattern.length()-1;
while(k<template.length()) {
    match = true;
    j=k;
    int i = pattern.length()-1;
    while (i>=0 && match) {
        if (pattern.charAt(i) != template.charAt(j))
            match=false;
        else {
            j--;
            i--;
        }
    }
    if (match) {
        System.out.print(k-(pattern.length()-1)+", ");
        k++;
    } else {
        //bcr
        int ch=0;
        if (template.charAt(j)=='A') ch = 0;
        if (template.charAt(j)=='C') ch = 1;
        if (template.charAt(j)=='G') ch = 2;
        if (template.charAt(j)=='T') ch = 3;
        int s1;
        if (i-R[ch]>1) s1 = i-R[ch];
        else s1 = 1;
        //gsr
        int suffix=0;
        if (i<pattern.length()-1) suffix=L[i+1];
        int s2=0;
        if (i!=pattern.length()-1) {
            if (suffix!=0) s2=pattern.length()-1-suffix;
            if (suffix==0 && i!=0) s2=i+1;
            if (suffix==0 && i==0) s2=1;
        }
        if (s1>s2) k=k+s1;
        else k=k+s2;
    }
}
```

Java-Code: Knuth-Morris-Pratt

```
//get Z-Boxes
Z = zbox.getBoxes(str);
//compute sp'
int[] sp = new int[pattern.length()];
for (int i=0;i<pattern.length();i++) sp[i]=0;
for (int i=pattern.length()-1;i>0;i--) {
   j = i+Z[i]-1;
   sp[j]=Z[i];
}
int h=0;
int i=0;
while (h+(pattern.length()-i)<=template.length()) {
   while (i<pattern.length() && pattern.charAt(i)==template.charAt(h)) {
      i++; h++;
   }
   if (i==pattern.length()) System.out.print(h-pattern.length()+", ");
   else {
      if (i==0) {
         i++; h++;
      }
   }
   i=sp[i-1];
}
```