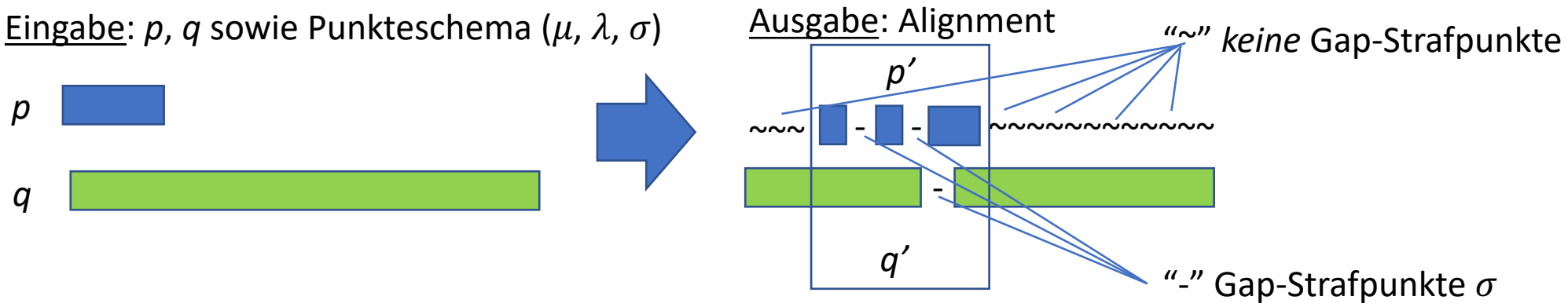
Algorithmen der Sequenzanalyse
- Projektthemen -

WS2024/2025

Prof. Sammeth

(1) Mustersuche

Paarweise Alignments können auch dazu eingesetzt werden, den besten Treffer eines verrauschten Musters *p* in einer Sequenz *q* zu finden, wobei naturgemäß $|p| << |q|$ gilt. Daher sollen bei der Mustersuche wie beim lokalen Alignment sowohl das Präfix als auch das Suffix des mit *p* alignierten Substrings *q'* keine Strafpunkte für gaps (Symbol "~") erhalten, allerdings soll *q'* (der alignierte Substring von *q*) global mit *p* (inkl. gaps "-") aligniert werden. Das Ergebnis der Mustersuche ist dann die Position des Substrings *q'*, der das Alignment-Score über alle *q'* in *q* optimiert:



Entwerfen und implementieren Sie einen Algorithmus basierend auf Dynamischer Programmierung, der ein optimales Mustersuche-Alignment von *p* und *q* nach einem gegebenen Punkteschema (z.B. Match μ = +1, Mismatch λ = -1, Gap σ = -1) ausgibt. Wie im o.a. Beispiel sollen im Alignment-Layout der Ausgabe nicht bestrafte Gaps mit dem Symbol " ~ " gekennzeichnet werden, um diese von "normalen" Gaps "-" unterscheiden zu können. Wenden Sie Ihr Programm dann auf den Datensatz "subtiles Motiv" an, indem Sie in jeder der Sequenzen *q* das Motiv *p*= AAAAAAGAGGGGGGT alignieren. Vergleichen Sie Ihre Ergebnisse mit entsprechenden globalen und lokalen Alignments von p mit den Sequenzen aus "subtiles Motiv".

(2) Anker im Alignment

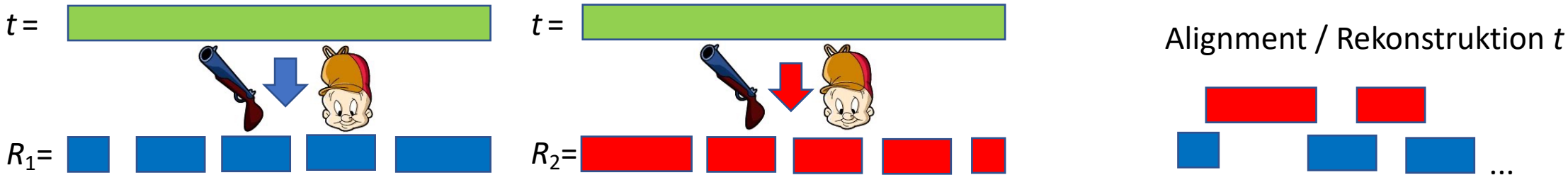
Wie im Seminar besprochen, entschlüsselte Marahiel den nicht-ribosomalen Code basierend auf u.a. Alignment der A-Domänen, das 19 Matches (markiert in rot) aufwies. Nehmen wir an, dass jede A-Domäne diese 19 Positionen unverändert ausweist, und die Aminosäuren AFDLLGGKIPLLDGNYGTE daher als "Anker" für A-Domänen Alignments benutzt werden können. Allerdings – abhängig vom Scoring Schema und den zu alignierenden A-Domänen – muss ein "optimales", globales Alignment nicht umbedingt diese Positionen alignieren.

Asp:
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
Orn:
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
Val:
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

Basierend auf dynamischer Programmierung, skizzieren Sie einen Algorithmus für ein paarweises Anker-Alignment, welches neben den üblichen Eingaben für das globale Alignment zweier Seqeuenzen *s* und *t* auch miteinander zu alignierende "Anker"-Positionen (a_i,b_j), mit $1 \leq a_i \leq |s|$ und $1 \leq b_j \leq |t|$, akzeptiert. Unter der Voraussetzung, dass alle Anker (a_i,b_j) in einem globalen Alignment realisierbar sind, sollen entsprechende Positionen (a_i,b_j) dann unabhängig von den restlichen Parametern (Scoring Schema, Gap Penalties, etc.) immer miteinander aligniert sein. Implementieren Sie Ihren Algorithmus und wenden Sie ihn auf alle 3 Paare des o.a. multiplen Alignments an. Generalisieren Sie Ihren Algorithmus dann für die Vorgabe der "Längsten GemeinsamenSubsequenz" (LCS, hier: AFDLLGGKIPLLDGNYGTE), die beim Alignment erzeugt werden soll.

(3) Sequenzrekonstruktion

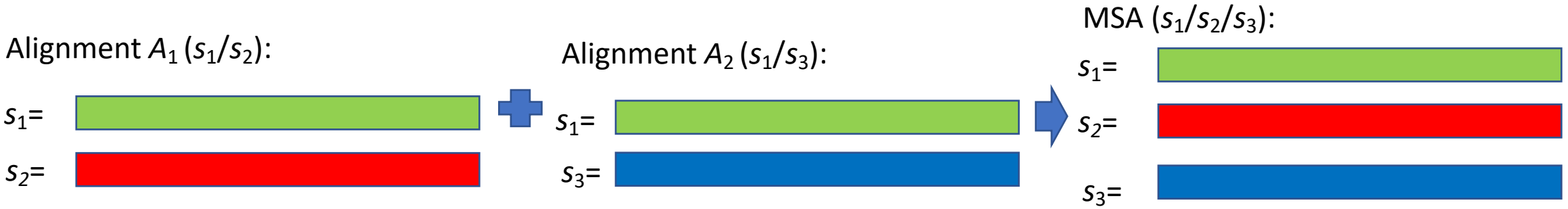
Ein anderes Andwendungs-Szenario für Alignments findet sich bei der Rekonstruktion längerer Nukelotidsequenzen aus verrauschten Substrings. Aus Kosten- und Zeit-Gründen werden die Sequenzen langer Nukleotidketten *t*, wie z.B. Genome, heute nicht mehr am Stück sondern in kleinen, fehlerbehafteten Abschnitten bestimmt. Bei diesem Vorgang (sog. “Shotgun-Sequenzierung”) werden viele Kopien der zu bestimmenden Sequenz an randomisierten Punkten wie mit einer Schrotflinte (“shotgun”) in ungefähr gleichlange Subsequenzen “zerschossen” und dann parallel “ausgelesen”, was zu möglicherweise fehlerbehafteten Substrings *R* (“reads”) der Original-Sequenz führt, die somit in ihren Präfixen bzw. Suffixen gegenseitig überlappen.



Entwerfen Sie einen Algorithmus basierend auf dynamischer Programmierung, der zunächst die Präfixe bzw. Suffixe *zweier* Read-Sequenzen *r* und *s* miteinander aligniert, nach dem Scoring Schema +1 für Matches und (-1) sowohl für Mismatches und Gaps. Basierend auf den Präfix-Suffix Alignment-Scores soll der Algorithmus den Eingabesequenzen dann eine Reihenfolge $\prec$ zuordnen*: s* $\prec$ *r* wenn das Score für das Alignment des Suffixes von *s* mit dem Präfix von *r* größer ist als das Alignment-Score des Präfixes von *s* mit dem Suffix von *r* , und *r* $\prec$ *s* für den umgekehrten Fall. Diskutieren Sie die Eingabegrößen, von denen die Zuordnung der Reihenfolge abhängt, und generalisieren Sie Ihren Algorithmus dann für *N* Eingabe-Reads. Zum Testen können Sie sich selbst randomisierte Substrings einer Beispiel-Sequenz *t* aus dem Seminar (z.B. OriC- oder Genom-Sequenzen) generieren, indem Sie Sie verschiedene Kopien von *t* jeweils an *N* randomisierten Stellen in Substrings unterteilen, die dann noch durch zufällige Mutationsereignisse (Substitutionen und/oder Indels) modifiziert werden.

(4) Stern-Alignments

Wie im Seminar besprochen, wird für die Lösung eines multiplen Alignment-Problems oft auf “Greedy”-Heuristiken zurückgegriffen. Ein Greedy Ansatz ist das sog. “Stern-Alignment”, bei dem eine Pivot-Sequenz (z.B. s_1) gewählt wird gegen die dann alle anderen Sequenzen (stern-förmig von s_1 aus) paarweise aligniert werden, um letztendlich das MSA aus den einzelnen paarweisen Alignments zusammenzusetzen:



Skizzieren und implementieren Sie einen Algorithmus, der als Eingabe zwei Alignments (A_1 und A_2) mit einer in beiden Alignments vorkommende Sequenz (z.B. s_1) zu einem MSA konkateniert, so dass im MSA dann alle in A_1 oder A_2 paarweise alignierten Positionen ($s_{i,a}$, $s_{j,b}$), $1 \leq a \leq |s_i|$ und $1 \leq b \leq |s_j|$, realisiert werden. Testen Sie Ihren Algorithmus für die u.a. Alignments mit der gemeinsamen Sequenz *Orn*. Implementieren Sie dann das Stern-Alignment, wobei Sie die Pivot-Sequenz nach dem besten Score im Alignment gegen alle anderen Sequenzen ausgewählt wird. Zeigen und diskutieren Sie den Einfluss der Pivot-Sequenz auf das Ergebnis des Stern-Alignments.

A_1

Orn:
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
Val:
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA---PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS

A_2

Asp:
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
Orn:
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS

(5) Branch-and-Bound

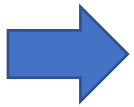
Neben Heuristiken, die nicht garantieren können eine beste Lösung finden, gibt es auch “exakte” Heuristiken, die ein nach der Zielfunktion ein optimales Ergebnis produzieren. Eine recht natürliche Heuristik ist die Reduktion des Suchraums durch Abschneiden (“*pruning*”) von Teilräumen, die keine optimale Lösung mehr produzieren können und somit nicht weiter abgesucht werden müssen. Das kann z.B. durch einen sog. “*branch-and-bound*” Schritt bei der iterativen Zusammensetzung der Lösung bewirkt werden, welcher entscheidet (“*branch”*), ob nach einer vorgegebenen Schranke die intermediär berechnete Teillösung weiter verfolgt oder alle darauf basierenden Ergebnisse als suboptimale Lösungen verworfen werden können (“*bound*”).

s= A T G T T A T A
t= A T C G T C C

Eingabesequenzen

A T – G T T A T A
A T C G T – C – C

Gesamtlösung
Kosten = 5



A T G T T A ...
– A T C G T ...

Teillösung
Kosten = 6 → verworfen

Entwerfen und implementieren Sie einen branch-and-bound Algorithmus basierend auf dynamischer Programmierung, der zwei Sequenzen *s* und *t* paarweise miteinander aligniert, und dabei nur so viele Schritte wie nötig berechnet. Benutzen Sie für Ihren Algorithmus die Kosten-Matrix (Match= 0, Mismatch= +1, Gaps= +1 Kosten) und wählen Sie zunächst eine triviale Schranke, z.B. max(|*s*|,|*t*|). Erklären Sie die Bedeutung der Schranke und schlagen Sie geeignetere Möglichkeiten vor, um generell bessere Schranken für das Alignment beliebiger Sequenzen *s* und *t* zu bestimmen. Beachten Sie, dass der Aufwand zur Bestimmung dieser Schranke nicht über dem Aufwand der anschliessend durchgeführten Rekursion für die Dynamischen Programmierung liegen sollte. Diskutieren Sie den Anteil der vom branch-and-bound tatsächlich berechneten Schritte in Abhängigkeit der Sequenzähnlichkeit ausgedrückt durch z.B. HAMMINGDISTANZ(*s*,*t*). Skizzieren Sie einen “best case” sowie einen “worst case” für Ihren Algorithmus, um zu beschreiben, welchen Aufwand Sie im “mittleren Anwendungsfall” von Ihrem Algorithmus erwarten.

(6) Progressives Alignment

In der Biologie ist das progressive Alignment eine der populärsten Greedy-Heuristiken für das MSA-Problem, da die zu alignierenden Sequenzen sich meist natürlich nach ihrer Evolution ordnen lassen. In einer rudimentären Form werden beim progressiven Alignment alle Eingabesequenzen s_i sukzessive zu einer Teillösung, d.h. (abgesehen vom ersten Schritt) einer Matrix von bereits alignierten Sequenzen "dazualigniert".

s_1= AGTC
s_2= GGGATTCC
s_3= CCTGACTTAC
s_4= TCGGATTCAT
s_5= TAGGGGAACC
s_6= TCGGGGTTTTT
s_7= ACGGGATTTTC
s_8= TTGGGACTTTT
s_9= TTGGGGACTTCC
s_{10}= TCGGGTATAACC

Progressives Alignment, Reihenfolge:

$< s_9, s_8, s_4, s_5, s_6, s_{10}, s_3, s_7, s_1, s_2 >$

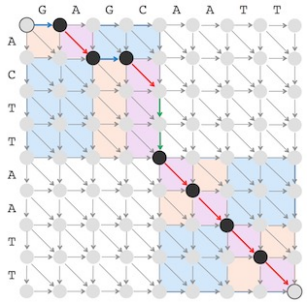
Alignment / Sequenz
+
i-te Sequenz

Implementieren Sie ein progressives Alignment, das die o.a. Eingabesequenzen nach ihrer Länge (von der kürzesten bis zur längsten, bei gleichlangen Sequenzen gilt die Eingabereihenfolge) aligniert. Verwenden Sie für Ihr Alignment das Scoring-Schema Match= +1, Mismatch= -1, Gap = -2, und mitteln Sie diese Punkte entsprechend für Profile bereits alignierter Sequenzen. Diskutieren Sie den Einfluß der Reihenfolge auf das Alignment und skizzieren Sie einen "worst case" für *jedes* progressive Alignment.

Verbessern Sie dann ihr progressives Alignment, indem Sie die Ähnlichkeiten zwischen allen Sequenzen (bzw. Alignments bereits alignierter Sequenzen) durch paarweise (Profil-)Alignments (s_i x s_j) vor jedem progressiven Schritt bestimmen, und das jeweils ähnlichste Paar (*i* , *j*) als nächstes zur Teillösung alignieren. Schlagen Sie eine geeignete Strategie vor, die Punkte des Scoring-Schemas im Alignment zweier Alignments zu anzuwenden.

(7) Divide-and-Conquer Alignment

Wie im Seminar besprochen, wird die Divide-and-Conquer Strategie bei paarweisen Alignments insbesondere zur Reduzierung des Speicheraufwandes angewendet. Implementieren Sie einen Divide-and-Conquer Algorithmus, der zwei Eingabesequenzen *v* und *w* nach einem beliebigen Scoring Schema in $O(\min(|v|,|t|)$ optimal aligniert. Die Ausgabe Ihres Algorithmus soll sowohl das/ein optimales Alignment und die Länge dieses, nach dem Scoring Schema längsten, Pfades zur Verfügung stellen.



Alignment:

A T – G T T A T A
A T C G T – C – C

Score = -1

Diskutieren Sie, welche Lösung der verwendete Algorithmus im Falle von mehreren, gleichwertig optimalen Lösungen findet, und belegen Sie Ihre Argumente mit Beispiel-Läufen Ihres Programmes. Skizzieren Sie dann einen Algorithmus, der mit der Divide-and-Conquer Strategie ebenfalls in $O(\min(|v|,|t|)$ Speicher alle optimalen Alignments der Sequenzen *v* und *t* ausgibt. Evaluieren Sie Ihr Programm zunächst am Minimalbeispiel *v*= ATGTTATA, *w*= ATCGTCC mit Match= +1, Mismatch= -1 und Gap= -1, und wenden Sie Ihr Programm dann auch auf die A-Domänen für *Asp* und *Orn* an:

Asp:
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
Orn:
AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS