UNIVERSITÀ DEGLI STUDI DI VERONA

Laurea Magistrale in Ingegneria Informatica (LM-32)
Dipartimento di informatica

ALLINEAMENTO SEMIGLOBALE UN NUOVO APPROCCIO

Un algoritmo per l'allineamento semiglobale
su stringhe solide, D-strings e ED-strings

LAUREANDO
Vincenzo Minolfi
VR457811

RELATRICE
Zsuzsanna Lipták

ANNO ACCADEMICO 2024 - 2025

Indice

Intertwine the lines
That swim beneath the dark
Realize the pain we live in
Demonize the need we reel in, no
In my memories I'll dig deep enough to know
Centuries of dreams unending
Another me that yielded tears when someone had betrayed

Storm the sorrow - Epica

Abstract (IT)

L'allineamento tra stringhe è una tecnica fondamentale della bioinformatica, utile per confrontare sequenze di DNA, RNA o proteine al fine di scoprire similarità e differenze significative. In questa tesi si propone un nuovo algoritmo per l'allineamento *semiglobale* tra le sequenze, ottimizzato per l'uso di memoria ed applicabile anche in caso di incertezze o varianti in una delle due stringhe.

L'algoritmo, concepito per stringhe ordinarie, è stato esteso per supportare anche il confronto tra una *solid string* e una *elastic degenerate string (ED-string)*, ovvero una sequenza nella quale certe regioni possono contenente diverse possibili sottosequenze di lunghezza variabile. A differenza degli approcci tradizionali basati sull'algoritmo di Needleman-Wunsch, la soluzione proposta adotta una strategia di memorizzazione differente che, pur avendo nel caso peggiore una complessità spaziale di $O(m^2)$ (dove m è la lunghezza della stringa più corta), in scenari reali si avvicina ad una complessità di spazio lineare; con la possibilità di ridurla fino a $O(m)$ mediante la suddivisione dell'allineamento in sotto-allineamenti di dimensione minore, da ricalcolare separatamente. L'implementazione è stata realizzata in linguaggio Rust. I capitoli tecnici illustrano nel dettaglio il funzionamento dell'algoritmo dalla versione iniziale, più semplice, fino alla versione più ottimizzata e all'estensione per le *ED-strings*.

Abstract (EN)

String alignment is a fundamental technique in bioinformatics, used to compare DNA, RNA, or protein sequences to identify significant similarities and differences. This thesis proposes a novel algorithm for *semiglobal alignment*, optimized for memory usage and applicable even when one of the sequences contains uncertainties or variations.

The algorithm, initially designed for ordinary strings, has been extended to handle the alignment between a *solid string* and an *elastic degenerate string (ED-string)*, a sequence where certain regions can have different possible subsequences of varying length. Unlike traditional approaches based on the Needleman-Wunsch algorithm, the proposed solution follows a different memory management strategy which, while maintaining a worst-case space complexity of $O(m^2)$ (where m is the length of the shorter string), in practice approaches linear space complexity. Furthermore, it can be reduced to $O(m)$ space usage by splitting the alignment into smaller sub-alignments that can be recomputed separately. The implementation was carried out in the Rust programming language. The technical chapters describe in detail the operation of the algorithm, from the initial, simpler version to the optimized version and the extension for ED-strings.

Capitolo 1

Introduzione

L'allineamento tra sequenze genomiche è una tecnica fondamentale nel campo della bioinformatica e consiste nel confrontare sequenze di DNA, RNA o proteine. L'obiettivo è trovare la corrispondenza ottimale tra due stringhe inserendo eventuali spazi, chiamati *gap*, in modo da massimizzare il punteggio di similarità tra esse. Tale punteggio si basa sull'associazione tra le lettere della sequenza e gli eventuali spazi. Una lista non esaustiva delle possibilità date dall'allineamento comprende lo studio delle relazioni evolutive tramite il confronto tra sequenze di specie differenti, un aiuto alla comprensione di alcune funzioni biologiche tramite l'associazione di sequenze simili ma non identiche e l'individuazione di diversi tipi di mutazioni. Esistono diversi tipi di allineamento. Con l'allineamento globale si cerca di allineare completamente le due sequenze oggetto del confronto. Con l'allineamento locale si cercano le regioni più simili tra loro all'interno delle due sequenze. L'allineamento semiglobale, ampiamente utilizzato nella pratica bioinformatica [5], costituisce un approccio intermedio tra questi due permettendo di allineare parte di una stringa con parte dell'altra ed ignorando, se utile ai fini del punteggio, le parti terminali di una o di entrambe le sequenze. L'allineamento semiglobale è utile quando si desidera confrontare due sequenze senza penalizzare eventuali basi non allineate alle estremità di una o di entrambe. Questa caratteristica permette di ottenere un allineamento ottimale anche quando una sequenza è contenuta solo parzialmente nell'altra, evitando che regioni extra all'inizio o alla fine distorcano il punteggio complessivo. È impiegato, per esempio, nel mapping dove le letture interessano solo una parte del genoma di riferimento [26]. È inoltre utile in situazioni pratiche di sequenziamento genomico, dato che il processo genera frammenti la cui posizione e lunghezza non sono note a priori e l'assemblaggio o il mapping richiedono gaps terminali non penalizzati [29] [6] [27].

Nel tempo sono stati sviluppati diversi algoritmi per l'allineamento di sequenze genomiche. Un algoritmo molto conosciuto, usato ed insegnato per l'allineamento tra due stringhe è l'algoritmo Needleman-Wunsch [25]. Questo algoritmo ha una complessità di spazio $O(nm)$, dove m ed n sono le lunghezze delle due sequenze interessate. Nel tempo sono state ideate diverse versioni ottimizzanti, ad esempio tramite un approccio divide&conquer si è riusciti a raggiungere una complessità lineare pari $O(n+m)$ [7]. Un esempio di questo è l'algoritmo

di Hirschberg [12].

Questi algoritmi si occupano, almeno nelle loro versioni originali, di stringhe solide, ovvero stringhe in cui ogni carattere è conosciuto con certezza. È tuttavia frequente che le stringhe su cui eseguire l'allineamento abbiano dei caratteri, o addirittura delle sequenze di caratteri, incerti. Nel caso la lunghezza rimanga fissa ci si riferisce a queste sequenze con il termine di *degenerate strings (D-strings)* mentre nel caso la lunghezza sia variabile tra le diverse alternative possibili le si chiamano *elastic degenerate strings (ED-strings)*. L'introduzione delle ED-strings è motivata dalla necessità di rappresentare insiemi di sequenze correlate. In genomica, lo studio della variazione genetica all'interno di una specie ha mostrato i limiti di una rappresentazione lineare del genoma di riferimento in quanto regioni biologicamente rilevanti possono presentare una diversità elevata [19]. Per questo motivo, la ricerca più recente si è orientata verso rappresentazioni capaci di includere le varianti all'interno della sequenza di riferimento. Le ED-strings offrono un modello naturale per questa esigenza, permettendo di esprimere in modo compatto l'insieme di più sequenze tra loro correlate [13] [3] [14].

Per quanto riguarda D-strings e ED-strings sono stati studiati alcuni approcci sempre derivanti fondamentalmente dall'algoritmo Needleman-Wunsch ma non c'è ancora un utilizzo consolidato di specifiche varianti ed è un campo in cui la ricerca è ancora molto attiva.

In questa tesi verrà presentato un nuovo algoritmo basato su un approccio differente da quelli finora utilizzati per ottimizzare l'occupazione di spazio dell'algoritmo Needleman-Wunsch. Considerando $m \ll n$, l'algoritmo nella sua versione finale fornisce la soluzione con complessità di spazio $O(m^2)$ nel caso pessimo, questo è già vantaggioso nel caso in cui ci sia una grande disparità nella lunghezza delle due stringhe da allineare. Il numero di nodi che vengono mantenuti in memoria dipende però fortemente dal grado di somiglianza delle stringhe confrontate in quanto l'occupazione è proporzionale al numero di aperture e chiusure di sequenze di gap; per questo motivo se usato su stringhe che hanno una ragionevole somiglianza, com'è nel caso di confronto tra varianti diverse di un gene, ad esempio, oppure nel caso l'allineamento contenga lunghe sequenze di gap, l'occupazione di spazio può essere molto bassa, in alcuni casi persino minore di $n + m$.

Per l'algoritmo verranno presentate diverse versioni. Partendo da un'intuizione iniziale, progressivamente sviluppata e raffinata, verranno presentate le ottimizzazioni che portano alla versione finale per l'allineamento tra stringhe solide. Nell'ultima parte di questa tesi verranno mostrate le ulteriori modifiche apportate per gestire anche l'allineamento tra una stringa solida ed una ED-string.

L'implementazione è stata effettuata in Rust [24], un linguaggio moderno che enfatizza le prestazioni e la sicurezza nella gestione della memoria.

1.0.1 Struttura della tesi

Nel Capitolo 2 verranno introdotti tutti i concetti coinvolti nei capitoli successivi. Nel Capitolo 3 verrà mostrato il funzionamento dell'algoritmo partendo dalla sua versione iniziale

ed aggiungendo, una alla volta, le ottimizzazioni che hanno portato alla versione attuale. Nel Capitolo 4 verranno mostrate le modifiche che rendono l'algoritmo funzionante anche con le ED-strings. In ultimo, nel Capitolo 5, tracceremo le conclusioni del lavoro svolto, con un confronto riassuntivo tra i diversi algoritmi e le diverse versioni esposte.

Capitolo 2

Nozioni preliminari

2.1 L'allineamento tra sequenze

Prima di vedere la definizione di allineamento è necessario definire un alfabeto, solitamente indicato con Σ, come l'insieme dei caratteri che possono appartenere alle stringhe di cui si vuole effettuare l'allineamento. Nel caso di sequenze di DNA, l'alfabeto comprende le quattro lettere A,C,G,T rappresentanti rispettivamente le basi azotate Adenina, Citosina, Guanina, Timina. Nel caso di sequenze proteiche ci sono venti amminoacidi standard, rappresentati da altrettante lettere dell'alfabeto latino, o 23 nel caso di alcuni organismi particolari. Gli esempi contenuti in questa tesi si basano sull'alfabeto del DNA, composto dalle quattro lettere A,C,G,T sopracitate.

Una *stringa* su Σ è una sequenza finita di caratteri appartenenti a Σ. In questa tesi verranno usati indifferentemente i termini *stringa* e *sequenza* per indicare questo concetto. Per indicare la lunghezza delle due stringhe soggette all'allineamento verranno usate le lettere n ed m.

Un allineamento tra due stringhe s e t è definito come una matrice con due righe tali che:

- i caratteri appartengono all'insieme $\Sigma \cup \{-\}$. Il carattere "$-$" indica un *gap*
- rimuovendo tutti i gap dalla prima riga si ottiene la stringa s
- rimuovendo tutti i gap dalla seconda riga si ottiene la stringa t
- nessuna colonna contiene due gap

Per computare il punteggio di un allineamento si sommano i punteggi di tutte le colonne dell'allineamento. Per trovare il punteggio della singola colonna si definisce una *scoring function* $f : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \backslash \{(-,-)\} \longrightarrow R$ che assegna un punteggio a seconda che i due caratteri coincidano (*match*), non coincidano (*mismatch*) o uno dei due sia un *gap*.

Ad esempio, una scoring function può assegnare 1 punto ad un *match*, -1 punto ad un *mismatch* e -1 punto ad un gap. Questa è la funzione di punteggio utilizzata in tutti gli esempi contenuti in questa tesi, e viene a volte utilizzato il termine *penalità* per indicare il punteggio dei gap o dei mismatch.

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ C & A & G & C & G & T & A & C & A & C & T \\ C & - & - & C & - & T & - & - & A & - & - \end{array} \tag{2.1}$$

Esempio 2.1: allineamento con punteggio -3 usando la scoring function 1 (m), -1 (mm), -1 (gap)

Utilizzando differenti scoring function è possibile raggiungere obiettivi differenti. Ad esempio, assegnando 1 ad ogni match, 0 ad ogni gap, ed eliminando il caso del mismatch assegnando ad essi $-\infty$, è possibile ottenere la LCS (Longest Common Subsequence) [15], come nell'esempio che segue.

$$\begin{array}{cccccccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ A & C & C & T & T & C & C & T & A & A & A & C & A & G & T & C & A & - & - & - \\ - & C & C & T & - & - & - & - & A & - & - & C & A & - & T & - & - & T & T & T \end{array} \tag{2.2}$$

Esempio 2.2: allineamento con punteggio 7 usando la scoring function 1 (m), 0 (mm), $-\infty$ (gap). Considerando le sole colonne prive di gap si ottiene la LCS tra le due stringhe.

Date due stringhe s e t e dato un allineamento A tra esse, questo è detto *ottimale* se non esistono allineamenti tra tali stringhe con un punteggio superiore ad esso.

$score(A) = max\{score(A') \mid A' \; is \; an \; alignment \; of \; s \; and \; t\}$

L'allineamento ottimale può non essere unico: possono esistere due o più allineamenti col medesimo punteggio massimo. L'algoritmo esposto in questa tesi fornisce, nel caso esistano più allineamenti ottimali, uno di essi. Quale di questi viene fornito è determinato da una scelta progettuale spiegata nel Paragrafo 3.1.2.

2.2 L'algoritmo Needleman-Wunsch

L'algoritmo Needleman-Wunsch [25] è un algoritmo di programmazione dinamica utilizzata per effettuare l'allineamento tra due stringhe. L'obiettivo è trovare un allineamento ottimale.

Questo algoritmo si basa sul principio di sottostruttura ottimale[7][10]: se un allineamento α è ottimale, allora β, lo stesso allineamento senza l'ultima colonna, è anch'esso ottimale. È possibile quindi calcolare un allineamento ottimale tra le stringhe s e t basandosi sugli allineamenti dei loro prefissi. Un allineamento ottimale tra s e t può terminare in tre modi possibili:

- l'ultima colonna è $\binom{s[n]}{-}$
- l'ultima colonna è $\binom{-}{t[m]}$
- l'ultima colonna è $\binom{s[n]}{t[m]}$

Come viene spiegato in [28], se l'allineamento α tra s e t è ottimale anche l'allineamento β (α senza l'ultima colonna) è ottimale. A seconda dei tre casi poco sopra esposti abbiamo quindi, rispettivamente, che:

- β è un allineamento ottimale tra $s[1...n-1]$ e $t[1...m]$
- β è un allineamento ottimale tra $s[1...n]$ e $t[1...m-1]$
- β è un allineamento ottimale tra $s[1...n-1]$ e $t[1...m-1]$

Sommando il punteggio dell'ultima colonna, a seconda del caso, abbiamo i 3 punteggi (ed allineamenti) possibili tra cui scegliere il migliore.

L'algoritmo Needleman-Wunsch si basa sulla costruzione di una tabella di valori. L'obiettivo è allineare due sequenze s e t di lunghezza rispettivamente m ed n. Si prepara innanzitutto una tabella di dimensioni $(m+1) \times (n+1)$ e si procede quindi ad inizializzare la prima riga e la prima colonna. La prima riga viene compilata con i punteggi dell'allineamento ottimale, l'unico allineamento possibile, tra la stringa *s[0..i]* e la stringa vuota ϵ; la prima colonna con i punteggi dell'allineamento ottimale, anch'esso unico, tra la stringa *t[0..j]* e la stringa vuota ϵ. L'unicità di questi due allineamenti è la conseguenza diretta dell'esclusione di colonne di tipo $(-,-)$ dalla definizione di allineamento e dal fatto che una delle due stringhe è una stringa vuota.

$$M(i,0) = i \cdot gap\ penalty$$

$$M(0,j) = j \cdot gap\ penalty$$

Si parte ora dalla prima cella libera in alto a sinistra e si procede con il calcolo del valore per ciascuna cella secondo la seguente regola:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + \begin{cases} \text{match score} & \text{se } s[i] = t[j] \quad \text{(match)} \\ \text{mismatch penalty} & \text{se } s[i] \neq t[j] \quad \text{(mismatch)} \end{cases} \\ M(i-1,j) + \text{gap penalty} \quad \text{(gap in } t\text{)} \\ M(i,j-1) + \text{gap penalty} \quad \text{(gap in } s\text{)} \end{cases}$$

Si può osservare che ogni valore viene calcolato basandosi sui valori di tre altre celle della tabella: quella sopra, quella a sinistra, quella sopra a quella sinistra (in diagonale, quindi). Il significato di questa operazione è il seguente. Il valore della cella $M(i,j)$ è il punteggio massimo che possiamo avere allineando i prefissi $s[0...i]$ e $t[0...j]$. Se i valori $s[i]$ e $t[j]$ sono allineati tra loro (cioè appaiono nella stessa colonna dell'allineamento ottimale che si sta calcolando tra $s[0..i]$ e $t[0..j]$), il punteggio dell'allineamento è pari al punteggio della cella $M(i-1, j-1)$, a cui va sommato il punteggio dato dal *match* o dal *mismatch* tra i due nuovi caratteri aggiunti all'allineamento. Se invece estendiamo l'allineamento con $s[i]$ ed un gap (oppure con un gap e $t[j]$) il punteggio dell'allineamento è pari all'allineamento della cella $M(i-1, j)$ (oppure $M(i, j-1)$) a cui va sommato il punteggio associato al *gap*. Si sceglie

tra queste tre possibilità il valore maggiore in quanto lo scopo è estendere l'allineamento nel modo migliore possibile, ottenendo un allineamento ottimale.

Una volta completato il calcolo di tutti i valori della tabella si trova nella cella in basso a destra, quindi alle coordinate (m, n), il punteggio di un allineamento ottimale tra s e t.

Per ricostruire un allineamento ottimale si procede a ritroso, si parte dall'ultimo valore calcolato, quello di coordinate m, n, ed a ogni passo:

- Se $M(i, j)$ deriva dall'elemento $M(i-1, j-1)$ si inserisce $\binom{s[i]}{t[j]}$ all'inizio dell'allineamento
- Se $M(i, j)$ deriva dall'elemento $M(i, j-1)$ si inserisce $\binom{-}{t[j]}$ all'inizio dell'allineamento
- Se $M(i, j)$ deriva dall'elemento $M(i-1, j)$ si inserisce $\binom{s[i]}{-}$ all'inizio dell'allineamento

La ricostruzione termina quando è stata raggiunta la posizione $(0, 0)$.

L'ordine del calcolo dei valori della tabella non influenza il risultato finale purché ogni valore venga calcolato quando sono già stati calcolati i tre valori necessari per il suo calcolo. Solitamente si procede in ordine per riga o per colonna. Il nuovo algoritmo spiegato in questa tesi implementa un calcolo riga per riga ma è possibile implementare ordini differenti, ad esempio colonna per colonna.

Tornando alla formula esposta poco sopra, si deve porre l'attenzione sul fatto che ogni valore si basa su tre altri valori a lui vicini nella tabella. Da ciascun valore alle coordinate (i, j) derivano al massimo tre valori: quello alla sua destra di coordinate $(i + 1, j)$, quello sotto di coordinate $(i, j + 1)$, quello sotto a destra di coordinate $(i + 1, j + 1)$. Una volta calcolato il valore in posizione $(i+1, j+1)$ il valore in posizione (i, j) non viene più utilizzato in fase di calcolo ma solo, eventualmente, in fase di ricostruzione dell'allineamento. La sua utilità in fase di ricostruzione dipende dal "passaggio" o meno dell'allineamento ottimale da tali coordinate all'interno della matrice: in altre parole il valore in posizione (i, j) è utile alla ricostruzione solo se esiste un allineamento ottimale che è estensione di un allineamento ottimale tra s[..i] e t[..j].

2.2.1 Complessità di spazio

L'algoritmo Needleman-Wunsch prevede il mantenimento di tutti i valori della tabella, le cui dimensioni sono nell'ordine di $O(m \cdot n)$, fino alla ricostruzione dell'allineamento. Esistono varianti dell'algoritmo che riducono significativamente l'occupazione di memoria: ad esempio, l'approccio di Hirschberg [12] consente di scendere fino a $O(\min(m, n))$, al prezzo però di un aumento del numero di calcoli necessari.

2.2.2 Complessità di tempo

La complessità temporale dell'algoritmo Needleman-Wunsch è di $O(m \cdot n)$ in quanto calcola tutti i valori di una tabella di dimensioni $(n + 1) \times (m + 1)$ e per il calcolo di ciascun valore

si utilizza un tempo costante legato al calcolo e confronto tra tre valori. Per la ricostruzione dell'allineamento la complessità è $O(m+n)$, cioè la lunghezza massima di un percorso che dalla cella di coordinate (m,n) alla cella $(0,0)$ potendosi muovere solo verso l'alto, verso sinistra oppure in diagonale di un passo.

2.3 L'allineamento semiglobale

L'allineamento spiegato nella sezione precedente viene chiamato *allineamento globale*, per contrapposizione con altri tipi di allineamento che, pur riprendendo la stessa definizione e basandosi sugli stessi concetti, presentano delle differenze che li rendono più adatti a scopi specifici. Uno di questi è l'*allineamento semiglobale*. Mentre l'allineamento globale forza l'allineamento tra le due sequenze prese interamente, l'allineamento semiglobale è un tipo di allineamento che può ignorare i gap iniziali e finali delle sequenze eliminando la penalità nelle regioni periferiche [28]. A seconda delle necessità è possibile scegliere di penalizzare o non penalizzare ciascuna delle estremità delle due stringhe: combinando tutte le possibilità si ottengono 16 varianti. Siccome una di queste, quella che penalizza i gap sia all'inizio che alla fine sia della prima stringa che della seconda stringa, coincide con l'allineamento globale si può dire che l'allineamento semiglobale ha 15 varianti totali. In questa tesi si espone l'allineamento semiglobale nella variante che non penalizza alcun gap all'inizio o alla fine di ognuna delle due stringhe, per ogni altra variante è comunque applicabile l'algoritmo con modifiche minime che verranno mostrate alla fine del Capitolo 3.

$$\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ C & A & G & C & G & T & A & C & A & C & T \\ - & - & - & C & C & T & A & - & - & - & - \end{array} \tag{2.3}$$

Esempio 2.3: allineamento semiglobale con punteggio 2 (3 *match*, 1 *mismatch*, 0 *gap* interni)

2.4 NW modificato per l'allineamento semiglobale

Per l'allineamento semiglobale, a seconda della variante desiderata, si può non voler penalizzare i gap all'inizio ed alla fine delle due sequenze. Per quanto spiegato nella Sezione 2.2, le penalità dei gap all'inizio delle due sequenze sono indicati dalle celle della prima riga e della prima colonna. Non penalizzare i gap all'inizio delle due stringhe si traduce nel valorizzare tali celle con 0. Le penalità dei gap alla fine delle due sequenze sono indicate dai movimenti lungo l'ultima riga e lungo l'ultima colonna. Per non considerare penalità per i gap alla fine delle due sequenze, quindi, il risultato dell'allineamento è il valore maggiore tra le celle dell'ultima riga e dell'ultima colonna, non necessariamente l'ultima cella in basso a destra (quella che identifica la fine di entrambe le sequenze nello stesso punto dell'allineamento) come per l'allineamento globale.

2.4.1 Complessità

La complessità è uguale a quella dell'algoritmo Needleman-Wunsch non modificato sia in termini di tempo che di spazio. Il processo di costruzione della tabella, al netto della valorizzazione della prima riga e della prima colonna, è infatti il medesimo e le differenze non comportano una differenza di complessità. L'unica modifica nelle operazioni da effettuare consiste nel tenere traccia del valore massimo incontrato lungo l'ultima colonna e lungo l'ultima riga durante la costruzione della tabella oppure, in alternativa, la ricerca del valore massimo sull'ultima riga e sull'ultima colonna prima di partire con la ricostruzione dell'allineamento. In entrambi i casi, non si tratta di un aumento apprezzabile della complessità totale che rimane quadratica in termini sia di calcolo che di spazio.

2.5 Un algoritmo per l'allineamento in spazio lineare

Un algoritmo usato per effettuare l'allineamento in spazio lineare è spiegato nel libro *Biological Sequence Analysis*, di Durbin, Eddy, Krogh, Mitchison, nel capitolo 2.6 (vedere [7]), in questa tesi verrà chiamato *algoritmo divide&conquer* perchè utilizza, appunto, la tecnica *divide and conquer*.

La strategia consiste nel dividere le sequenze s e t in due parti, trovare un allineamento ottimale per i due prefissi e quello per i due suffissi, infine concatenare i due allineamenti. tuttavia, non è possibile dividere le due stringhe in una posizione qualsiasi: perché venga trovato un allineamento ottimale complessivo è necessario che le stringhe vengano divise in determinati punti, come nel seguente esempio:

$$\begin{pmatrix} \text{GA} \\ \text{G-} \end{pmatrix}\begin{pmatrix} \text{-C} \\ \text{AC} \end{pmatrix} \longrightarrow \begin{pmatrix} \text{GA-C} \\ \text{G-AC} \end{pmatrix} \qquad \begin{pmatrix} \text{GA} \\ \text{GA} \end{pmatrix}\begin{pmatrix} \text{C} \\ \text{C} \end{pmatrix} \longrightarrow \begin{pmatrix} \text{GAC} \\ \text{GAC} \end{pmatrix}$$

In questo esempio, il taglio delle stringhe s e t effettuato a sinistra non porta ad un allineamento ottimale dopo la concatenazione dei due allineamenti ottimali, il taglio delle stringhe effettuato a destra sì.

È necessario trovare, ad ogni passo, il punto adatto per effettuare un buon taglio (*good cut* in inglese) e dividere ogni stringa in un prefisso ed un suffisso. Questo può essere fatto tramite il seguente algoritmo.

Sia $n' = \lceil \frac{n}{2} \rceil$, ovvero l'intero superiore della metà di n. Esisterà una m' tale che la cella (n', m') si trovi su un allineamento ottimale: in altre parole, m' sarà l'indice della riga che insieme alla colonna di indice n' individua una cella di passaggio di un allineamento ottimale nella matrice. Possiamo quindi dividere il problema del trovare un allineamento ottimale in due diverse parti: quello da $(0, 0)$ a (n', m') e quello da (n', m') a (n, m).

Chiamiamo la matrice principale D. Per trovare m', si deve creare una seconda matrice di supporto, chiamata M, da compilare come segue:

- Per $i = n'$ e $j = 1..m$: $M(n', j) = j$

- Per $i > n'$ e $0 \leq j \leq m$: $M(i, j) = M(i\prime, j\prime)$ dove $D(i, j)$ deriva dalla cella $D(i\prime, j\prime)$. D è la matrice principale.

con valori da 1 a m.

Conclusi i calcoli, troviamo in $M(n, m) = r$ il valore r tale che $M(\lceil \frac{n}{2} \rceil, r)$ sia una cella da cui passa un allineamento ottimale. tra s e t. Effettuata la divisione delle due stringhe in tali punti si procede ricorsivamente per ognuna delle parti ottenute finché si ottengono delle parti con lunghezza minore o uguale a 2 e si applica su queste l'algoritmo Needleman-Wunsch.

2.5.1 Complessità di spazio

Questo algoritmo ha una complessità di spazio lineare. Più nel dettaglio, necessita di memorizzare $2m$ valori per la matrice D, un massimo di $2m$ valori per il calcolo della matrice M, infine vanno memorizzati tutti gli allineamenti parziali per un massimo di $n + m$ valori. Quando la matrice M ha dimensioni maggiori gli allineamenti parziali già trovati sono pochi, con il progredire dei calcoli ed il trovare nuovi allineamenti parziali la matrice M diminuisce in dimensione. In totale, quindi, si può stimare di avere in memoria circa $4m$ valori nel momento di massimo utilizzo della memoria. La complessità di spazio è quindi $O(m)$.

2.5.2 Complessità di tempo

Durante la prima iterazione il numero di calcoli da effettuare per la matrice D è pari a $(n + 1)(m + 1)$ mentre per la matrice M è pari a $(\lceil \frac{n}{2} \rceil)(m + 1)$. La complessità di queste due componenti combinate è $O(nm)$. Ad ogni nuova iterazione il numero di calcoli da effettuare è dimezzato in quanto si dimezza la stringa s, per un numero totale di calcoli pari a $2nm$, risultato di $nm + \frac{1}{2}nm + \frac{1}{4}nm + \frac{1}{8}nm\ldots$. La complessità di tempo è quindi $O(nm)$, come per l'algoritmo Needleman-Wunsch spiegato nella Sezione 2.2, anche se il numero di calcoli effettivi è raddoppiato.

2.6 Le stringhe degeneri e le stringhe degeneri elastiche

Quanto visto fino adesso è applicabile per sequenze che sono conosciute con certezza, nella pratica questo non è necessariamente vero. A causa di limiti tecnici nella procedura di sequenziamento possono verificarsi ambiguità, con porzioni di sequenza per le quali il macchinario può fornire due diverse possibili interpretazioni. Può anche essere che queste alternative rappresentino non un'incertezza nella procedura di sequenziamento ma l'esistenza di varietà differenti [8]. Si può quindi introdurre due definizioni per definire dei tipi particolari di stringhe.

Una *stringa degenere*[1][14] (in inglese *degenerate strings*, abbreviato in *D-strings*) è una sequenza di n insiemi di stringhe, dove ogni insieme contiene stringhe della stessa lunghezza ma la lunghezza può essere differente tra insiemi diversi.

Rimuovendo il vincolo della medesima lunghezza delle stringhe all'interno di ogni insieme, si ottiene la definizione di *stringa degenere elastica*[14] (in inglese *elastic degenerate string*, abbreviato in *ED-string*). In questa tesi verrà spesso usato il termine *ED-string*. Le ED-string possono essere usate, ad esempio, per rappresentare in maniera compatta due o più varianti alleliche di uno stesso gene.

$$\text{A}\left\{\begin{array}{c}\text{TT}\\ \text{C}\end{array}\right\}\text{GATTACAAT}\left\{\begin{array}{c}\text{A}\\ \text{C}\end{array}\right\}\text{GAAAGGT}$$

In questo esempio, le due alternative "TT" e "C" rappresentano un blocco elastico di alternative mentre "A" e "C" rappresentano un blocco non elastico. Le quattro stringhe rappresentate sono le seguenti:

- *ATTGATTACAATAGAAAGGT*
- *ACGATTACAATAGAAAGGT*
- *ATTGATTACAATCGAAAGGT*
- *ACGATTACAATCGAAAGGT*

Un altro caso di utilizzo delle *ED-strings* è l'analisi di pangenomi, ovvero sequenze rappresentanti una specie in tutte la varietà genetica espressa dai suoi individui.

Le *ED-strings* possono essere utilizzate anche per la ricerca efficiente di pattern all'interno di insiemi di sequenze tra loro simili.

Le stringhe caratterizzate da un unico simbolo per ogni posizione, prive quindi di alternative, sono chiamate *stringhe solide*, *stringhe standard* o *stringhe deterministiche* [14]. Quando non meglio specificato, in questa tesi, il termine stringa identifica una stringa solida.

Esistono versioni modificate dell'algoritmo Needleman-Wunsch adatte ad effettuare l'allineamento tra una stringa solida ed una D-string. Per gestire la presenza di più alternative nella *D-string*, si può memorizzare in ogni cella un vettore di punteggi, uno per ciascuna alternativa [20]. La complessità rimane quella dell'algoritmo originale, $O(nm)$, in tempo e spazio, e l'approccio risulta di facile comprensione ed implementazione.

L'algoritmo divide&conquer per l'allineamento globale, pur garantendo complessità spaziale lineare, risulta applicabile unicamente a stringhe solide. Nel caso di *D-strings* e *ED-strings*, la suddivisione ricorsiva delle sequenze non si estende direttamente: la presenza di insiemi di blocchi di alternative potrebbe rendere problematica la definizione di un *good cut*, su cui si basa l'intera strategia. Al momento, non risulta nella letteratura consultata alcun approccio di questo tipo per stringhe non solide, né è stata individuata una soluzione nell'ambito di questo lavoro di tesi.

2.6.1 Rappresentazione

Per rappresentare una *ED-string* si possono utilizzare diversi metodi e non c'è uniformità in letteratura. I formati come FASTA [17] e FASTQ [4] possono rappresentare stringhe solide,

prive di ambiguità, oppure D-strings, tramite l'utilizzo di simboli che indicano incertezza a livello del singolo carattere (ad esempio, nella rappresentazione di un acido nucleico "R" indica un'ambiguità tra adenina e guanina). Non sono formati adatti a rappresentare ambiguità tra insiemi di amminoacidi di lunghezza variabile e non sono quindi adatti a rappresentare ED-strings.

Un metodo utilizzato spesso per rappresentare le alternative quando esse possono avere lunghezze diverse é quello denominato *insiemi espliciti per regioni variabili* che consiste nell'elencare le alternative separate da virgola e delineare il loro insieme tramite graffe, ad esempio $ACG\ \{CC,\ TTT,\ AGG\}\ CGGCT$ [9][16].

In altri articoli è possibile trovare rappresentazioni strutturate costruite ad hoc per l'algoritmo implementato, ad esempio nella forma di un array di array di stringhe, ricalcando la definizione di ED-string [2].

Per l'algoritmo implementato e per gli esempi inseriti in questa tesi, si è scelto di separare le alternative tramite il simbolo di *pipeline* e delineare il loro insieme tramite parentesi quadre, ad esempio $ACG[CC|TTT|AGG]CGGCT$. Questa notazione dovrebbe risultare di immediata comprensione a chi ha familiarità con l'utilizzo delle espressioni regolari nell'ambito della programmazione.

Capitolo 3

Allineamento di stringhe

3.1 Il nuovo algoritmo

In ogni capitolo di questa tesi verranno utilizzati n ed m per indicare rispettivamente la lunghezza della sequenza s, che costituisce le righe della matrice, e la lunghezza di t, la stringa che costituisce le colonne della matrice. Siccome nella matrice ciascuna sequenza é preceduta da un carattere vuoto, è importante notare che la larghezza di una riga della matrice è pari a $n + 1$ mentre l'altezza di una colonna della stessa è pari a $m + 1$.

Nell'algoritmo Needleman-Wunsch abbiamo visto come da un elemento della matrice possono discendere da zero a tre elementi. Nel caso da un elemento non sia disceso alcun nuovo elemento esso può essere rimosso dalla memoria senza conseguenze per i calcoli che seguono e per la ricostruzione dell'allineamento ottimale trovato. La prima ottimizzazione che è possibile fare è pertanto quella di rimuovere dalla memoria gli elementi da cui non discendono altri elementi, in quanto costituiscono rami morti nell'albero che viene costruito. Quando un elemento viene eliminato il suo nodo genitore potrebbe a sua volta diventare un elemento privo di figli, in tal caso anche esso può essere eliminato senza conseguenze per il medesimo motivo.

3.1.1 Strutture dati

I dati ottenuti nel corso del calcolo vengono memorizzati in una *HashMap*, le cui chiavi sono le posizioni delle celle della tabella. Le celle sono numerate a partire da 0, seguendo l'ordine per riga. Ad ogni chiave è associata una struttura contenente la posizione del genitore, le posizioni degli eventuali figli ed il punteggio della cella in questione. Ciascun nodo è quindi identificato da un numero compreso tra 0 e $((n + 1) \cdot (m + 1)) - 1$: sulla prima riga sono presenti i nodi con identificativo compreso tra 0 e m, mentre i nodi che si trovano nella prima colonna hanno identificativo x tale che $0 < x < ((n+1) \cdot (m+1)) - 1$ e $x \pmod{(m+1)} \equiv 0$.

3.1.2 Costruzione di base

La costruzione nella prima versione avviene calcolando la matrice di valori nello stesso modo in cui viene calcolata con l'algoritmo Needleman-Wunsch spiegato nella Sezione 2.2. La memorizzazione dei valori calcolati non viene però gestita in una matrice ma in una struttura del tipo spiegato nel paragrafo 3.1.1. L'occupazione di memoria a questo stadio è pertanto data dalla dimensione del singolo nodo moltiplicata per $(n + 1) \cdot (m + 1)$ nodi. In Figura 3.1 è mostrata l'espansione dell'albero con i nodi delle ultime tre righe della matrice.

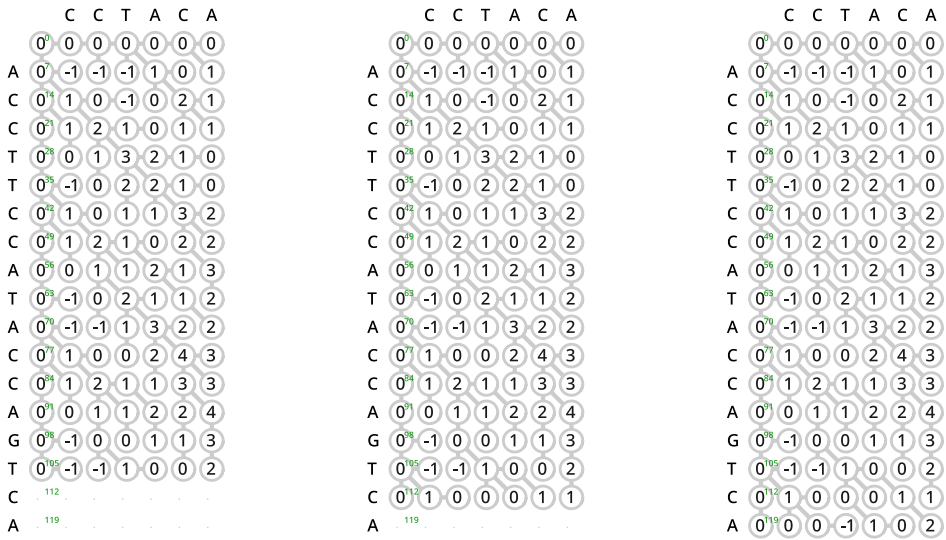


Figura 3.1: Calcolo delle ultime tre righe della matrice di valori ed estensione dell'albero

Nel caso in cui il valore di una cella possa derivare indifferentemente da due o da tre valori, viene utilizzata la seguente priorità: sinistra, diagonale, verticale. Questa scelta si fonda sulla seguente osservazione: i nodi della prima colonna derivano necessariamente dal nodo che si trova in verticale, e questo garantisce sempre l'esistenza di un percorso di derivazione che percorre l'intera prima colonna. L'algoritmo tende quindi, ove possibile, a sfruttare la presenza di questo percorso per ridurre la presenza di più percorsi che percorrono verticalmente colonne tra loro affiancate.

3.1.3 Prima ottimizzazione: scarto dei valori non più necessari

Tra tutti i nodi dell'albero ce ne sono molti che non hanno alcuna funzione per l'ottenimento finale di un allineamento ottimale delle due sequenze. Osservando attentamente la Figura 3.1 è possibile notare come il miglior punteggio considerando quelli appartenenti all'ultima riga ed all'ultima colonna sia il 4 presente in posizione 97, esso è infatti stato evidenziato in blu nella Figura 3.2. Prendiamo in prestito dall'ambito degli algoritmi di ricerca nell'intelligenza artificiale il termine di *frontiera*, definendolo come l'insieme dei nodi che possono ancora

essere coinvolti nel calcolo di altri nodi dell'albero, il che corrisponde a quegli elementi in posizioni tali per cui devono ancora essere calcolati i tre elementi che possono derivare da ciascuno di essi. Questi sono le foglie colorate in rosso nella Figura 3.2. Mentre si calcola un nodo in posizione w i nodi di frontiera sono tutti quelli inclusi tra $w - (n + 1) - 1$, ovvero l'elemento nella riga e nella colonna precedenti all'elemento che si sta calcolando w, e $w - 1$, ovvero l'elemento immediatamente a sinistra di w. In un qualunque momento della costruzione dell'albero vanno mantenuti in memoria il miglior nodo trovato fino a tale momento, i nodi costituenti la frontiera e tutti i nodi che portano dalla radice dell'albero, ovvero l'elemento più in alto a sinistra della matrice, a ciascuno dei nodi appena detti. Tutti gli altri nodi sono foglie che non fanno parte della frontiera e che non sono quindi necessari ad alcun nuovo calcolo, oppure sono nodi che hanno solo questo tipo di foglie nel proprio sottoalbero e non sono quindi usati né per ricostruire l'allineamento ottimale trovato fino a quel momento né eventuali nuovi allineamenti ottimali che potrebbero dover essere ancora trovati. Questi sono tutti nodi che non è utile tenere e che possono essere eliminati per ridurre l'occupazione di memoria da parte dell'algoritmo.

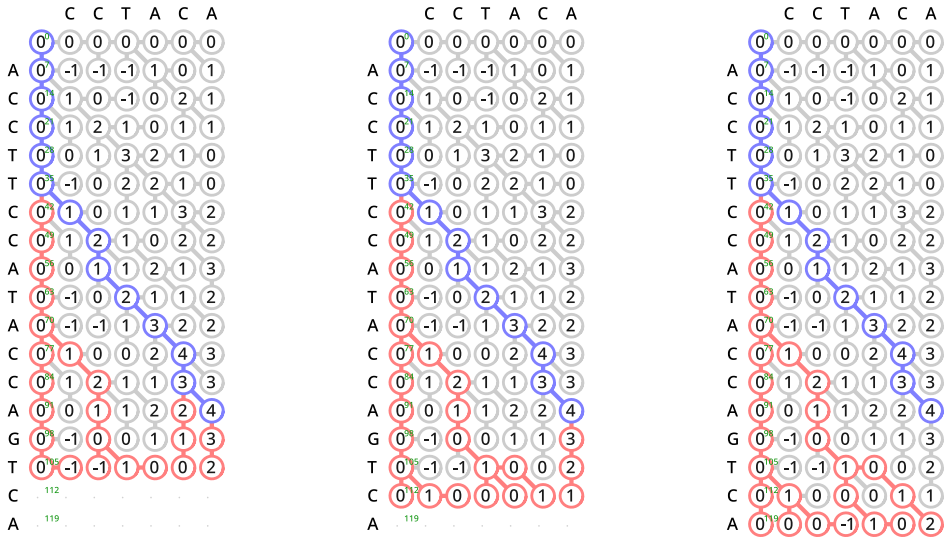


Figura 3.2: Calcolo delle ultime tre righe della matrice di valori ed estensione dell'albero come nella figura precedente. Sono qui evidenziati in blu i nodi che compongono un allineamento ottimale ed in rosso i nodi che compongono gli allineamenti tra $s[1..i]$ per ciascun i e $t[1..j]$, cioè l'ultima riga calcolata

L'Algoritmo 1 mostra la creazione dell'albero, la scelta dell'elemento col maggiore punteggio ed il primo sistema di ottimizzazione della memoria spiegato nel Paragrafo 3.1.3.

La funzione $tree\_prune$ presente nell'Algoritmo 1 viene eseguita ogni volta che l'algoritmo completa il calcolo di un valore w e viene chiamata con l'indice di quell'elemento che non verrà più preso in considerazione nei calcoli, corrispondente a $w - (n + 1) - 1$, come

Algoritmo 1 Costruzione dell'albero

```
1: procedure BUILD_TREE(seq_s, seq_t)
2:     tree ← empty_dictionary
3:     max_points ← 0
4:     max_pos ← 0
5:     n ← len(seq_s)
6:     m ← len(seq_t)
7:     create_node(0, 0, tree)
8:     for i ← 1 to n do
9:         create_node(i, i-1, 0, tree)
10:    end for
11:    for j ← 1 to m do
12:        points ← max(0, tree[(j − 1) ∗ (n + 1)].points + gap)
13:        create_node(j ∗ (n + 1), (j − 1) ∗ (n + 1), points, tree)
14:        if j > 1 then
15:            tree_prune((j − 1) ∗ (n + 1), tree, max_pos)
16:        end if
17:        for i = 1 to n do
18:            w ← i + (j ∗ (n + 1))
19:            wdiag ← w − (n + 1) − 1
20:            wup ← w − (n + 1)
21:            match_mismatch ← tree[wdiag].points + (seq_s[i − 1] ≡ seq_t[j −
    1] ? match_score : mismatch_score)
22:            del_score ← tree[wup].points + gap
23:            ins_score ← tree[w − 1].points + gap
24:            if match_mismatch > ins_score and match_mismatch > del_score then
25:                create_node(w, wdiag, match_mismatch, tree)
26:            else if del_score > ins_score then
27:                create_node(w, wup, del_score, tree)
28:            else
29:                create_node(w, w-1, ins_score, tree)
30:            end if
31:            tree_prune(wdiag, tree, max_pos)
32:        end for
33:        last_idx ← (j + 1) ∗ (n + 1) − 1
34:        if tree[last_idx] > max_score then
35:            if max_pos > 0 ∧ max_pos < (j − 1) ∗ (n + 1) ∧ max_pos is not leaf then
36:                tree_prune(max_pos, tree, 0)
37:            end if
38:            max_score ← tree[last_idx].score
39:            max_pos ← last_idx
40:        end if
41:    end for
42:    for i = m ∗ (n + 1) to (m + 1) ∗ (n + 1) do
43:        if tree[i] > max_score then
44:            max_score ← tree[i].score
45:            max_pos ← i
46:        end if
47:    end for
48:    return tree, max_points, max_points_idx
49: end procedure
```

Algoritmo 2 Procedure necessarie per l'algoritmo precedente

```
1: procedure CREATE_NODE(w, parent, points, tree)
2:     parent_node ← tree[parent]
3:     push w into parent_node.children
4:     n ← TreeNode{pos : w, parent : parent, children : empty_array, points :
   points}
5:     tree[w] = n
6: end procedure
7: procedure TREE_PRUNE(w, tree, protected)
8:     current_id ← w
9:     while current_id ≠ protected and current_id is not root do
10:        current_node ← tree[current_id]
11:        if current_node.children is not empty then
12:            break
13:        end if
14:        parent_node ← tree[current_node.parent]
15:        remove current_id from parent_node.children
16:        remove current_node from tree
17:        current_id ← current_node.parent
18:    end while
19: end procedure
```

input. Siccome da tale nodo non possono derivare nuovi elementi si controlla quanti elementi ne sono derivati. Se il numero è zero l'elemento può essere eliminato dall'albero. In tal caso si procede con l'effettuare la stessa valutazione sul relativo genitore, ricorsivamente finché si raggiunge un elemento con altri figli.

3.1.4 Seconda ottimizzazione: compattazione dei percorsi

Identificando i nodi tramite la loro posizione ed essendo l'obiettivo un percorso che unisce il nodo con punteggio maggiore alla radice dell'albero, è possibile evitare la memorizzazione di ogni nodo facente parte del percorso. Non vi è alcuna perdita di informazione, infatti, memorizzando solo i nodi che comportano un "cambio di direzione" del percorso, cioè i nodi in cui la direzione della dipendenza rispetto al nodo genitore è differente da quella rispetto al nodo figlio. Data una serie di nodi $[w_1, w_2 ... w_k]$ tale che ogni nodo w_k derivi dal nodo $w_{(k-1)}$ seguendo la stessa direzione, è sufficiente memorizzare i soli nodi w_1 e w_k. Durante la fase di ricostruzione del percorso si verifica una delle tre situazioni seguenti:

- $w_1 \pmod{n+1} \equiv w_k \pmod{n+1}$ ovvero w_1 e w_k si trovano sulla stessa colonna: tutti gli elementi tra essi in tale colonna fanno parte del percorso
- $\lfloor w_1/(n+1) \rfloor \equiv \lfloor w_k/(n+1) \rfloor$ ovvero w_1 e w_k si trovano sulla stessa riga: tutti gli elementi tra essi in tale riga fanno parte del percorso

- $(w_1 \pmod{n+1} - w_k \pmod{n+1}) \equiv (\lfloor w_1/(n+1) \rfloor - \lfloor w_k/(n+1) \rfloor)$ ovvero w_1 e w_k si trovano sulla stessa diagonale: tutti gli elementi tra essi in tale diagonale fanno parte del percorso

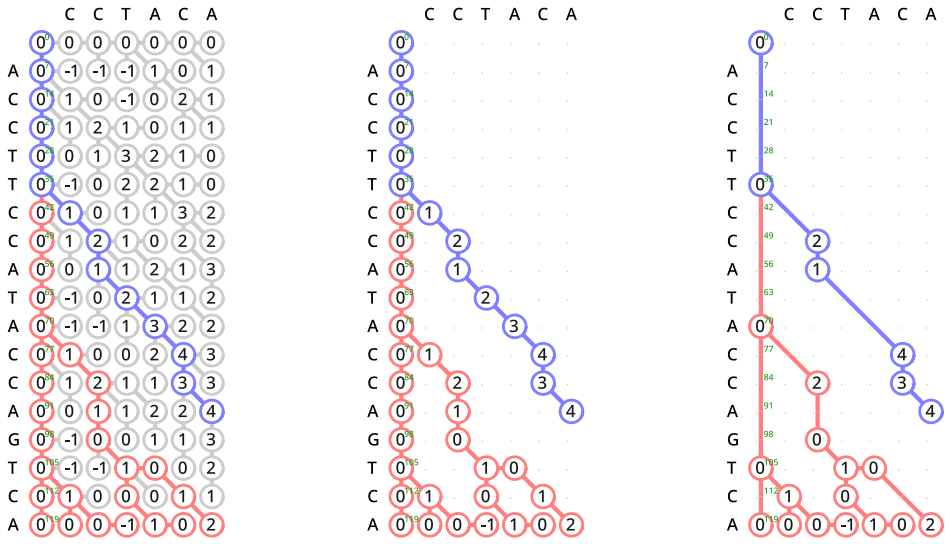


Figura 3.3: Confronto tra l'albero completo, l'albero con pruning effettuato e compattazione dei percorsi

Per effettuare questa compattazione è necessario modificare la procedura di pruning aggiungendo un ulteriore passaggio: dopo aver eliminato un ramo dell'albero si verifica se il nodo al quale il ramo era collegato è ancora un branching node ed in caso contrario lo si rimuove, collegando direttamente il suo genitore al suo unico figlio, a patto che tale operazione non introduca un cambio di direzione a livello di quel nodo. Nell'Algoritmo 3, il codice aggiuntivo che implementa questo nuovo passaggio è evidenziato in verde.

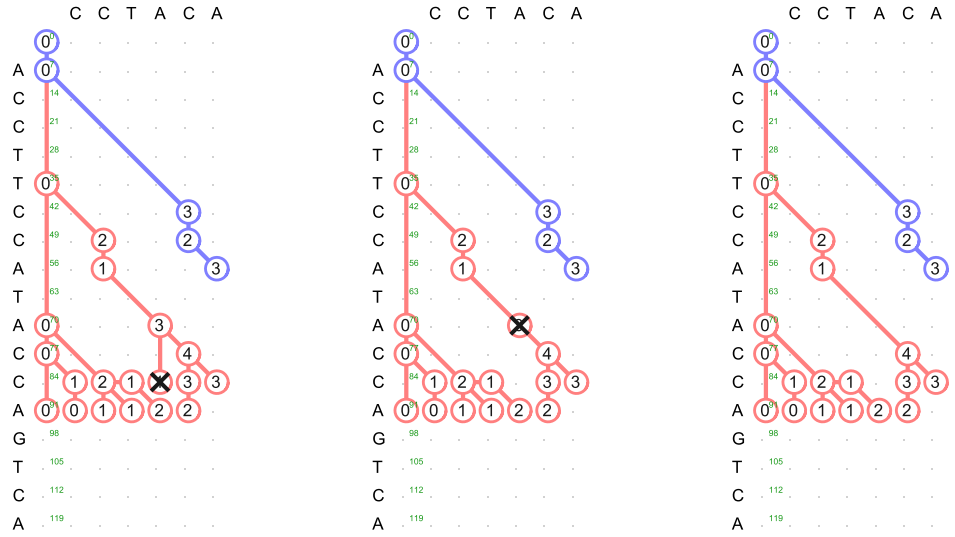


Figura 3.4: Procedura di pruning dopo il calcolo del nodo in posizione 96: il nodo 88 è un nodo terminale e non più utile, dopo la sua eliminazione il nodo 74 è un nodo senza ramificazioni e senza cambi di direzione e conseguentemente non più utile

Algoritmo 3 Nuova tree_prune per la compattazione dell'albero

```
1:  procedure TREE_PRUNE(w, tree, protected)
2:      current_id ← w
3:      children ← 0
4:      while current_id ≠ protected do
5:          current_node ← tree[current_id]
6:          children ← current_node.children.len()
7:          if current_node.children is not empty then
8:              break
9:          end if
10:         parent_node ← tree[current_node.parent]
11:         remove current_id from parent_node.children
12:         remove current_node from tree
13:         current_id ← n.parent
14:     end while
15:     if current_id ≠ protected ∧ current_id is not root ∧ children is 1 then
16:         child_id ← id of the only child of current_node
17:         parent_id ← id of the parent of current_node
18:         if parent_id, current_id and child_id are on the same row, column or diagonal
    then
19:             skip_node(current_id, tree)
20:         end if
21:     end if
22: end procedure
```

Nella nuova versione di *tree_prune* abbiamo un riferimento alla nuova funzione *skip_node*

il cui compito è di rimuovere un nodo dall'albero modificando al contempo i riferimenti del suo genitore e del suo unico figlio.

Algoritmo 4 skip_node per la rimozione di un nodo dall'albero

```
1:  procedure SKIP_NODE(current_id, tree)
2:      current_node ← tree[current_id]
3:      parent_id ← current_node.parent
4:      child_id ← current_node.children[0]
5:      parent_node ← tree[parent_id]
6:      child_node ← tree[parent_id]
7:      remove current_id from parent_node.children
8:      insert child_id from parent_node.children
9:      sets parent_id as child_node.parent
10:     remove current_node from tree
11: end procedure
```

3.1.5 Complessità di spazio

Memorizzando solo i cambi di direzione, ossia i nodi di apertura e chiusura delle sequenze di gap, ogni percorso richiede al massimo due nodi per colonna: uno corrispondente all'ingresso nella colonna (ovvero l'apertura di una sequenza di gap nella sequenza s, con cambio di direzione verso il basso), l'altro corrisponde all'uscita dalla colonna (cioè la chiusura della sequenza di gap, con spostamento verso una colonna più a destra). Poiché la frontiera ha dimensione n+1, ed aggiungendo un eventuale percorso aggiuntivo per l'elemento massimo trovato lungo l'ultima colonna, il numero massimo di percorsi da memorizzare risulta $n + 2$. Per ciascuno di essi, il numero massimo di nodi è pari a $2(n + 1)$, portando il caso pessimo a $O(n^2)$. Una situazione simile è tuttavia estremamente improbabile e test sperimentali mostrano che l'occupazione di memoria è decisamente inferiore. Basti considerare che le sequenze confrontate presentano solitamente delle lunghe sottosequenze in comune e che per ciascuna di esse vengono memorizzati solamente due nodi: quello di inizio e quello di fine della sottosequenza, collegati da un lungo percorso in diagonale nella matrice di calcolo. Nel caso di lunghe sezioni differenti avremo lunghe sequenze di gap, in questo caso il percorso all'interno della nostra matrice sarà orizzontale o verticale e verrà memorizzato comunque tramite i soli nodi di inizio e fine sequenza. Tra due sottosequenze simili ci possono essere anche regioni ricche di nodi, in questo caso è possibile applicare un'ulteriore ottimizzazione come spiegato nella Sezione 3.3.

Per effettuare una verifica in un contesto realistico è stato scelto l'organismo modello *Caenorhabditis elegans* e più in particolare *let-60*, un gene che codifica per una proteina Ras coinvolta nella segnalazione cellulare e che si trova nel cromosoma IV (coordinate da 11.688.227 a 11.691.088) dell'organismo in questione [11]. Si tratta di un gene di circa 2.8 kilobasi nella versione *unsliced* e presenta polimorfismo tra le varianti N2 [21] e CB4856 [22]. Sono quindi state estratte dalle due varianti 2.862 basi e queste sono state allineate tramite l'algoritmo. La matrice di allineamento possiede quindi 2.863 righe ed altrettante colonne,

per un numero totale di celle pari a 8.196.769. Nel corso dell'esecuzione dell'algoritmo è stato costruito e mantenuto in memoria un albero contenente, in fase finale, 31.650 nodi, pari a circa lo 0.39% della dimensione della matrice. Eseguendo l'allineamento dopo aver scambiato le due sequenze, ottenendo quindi una trasposizione della matrice dei punteggi, l'albero in memoria raggiunge un massimo di 29.659 nodi, pari allo 0.36% della dimensione della matrice. L'allineamento ottimale trovato è il medesimo, cambia tuttavia l'albero in quanto l'algoritmo lavora con una direzione preferenziale (dall'alto verso il basso). Nel caso una delle due sequenze sia molto più breve dell'altra conviene tenere questa sull'asse orizzontale in quanto il numero di foglie dell'albero è legato a tale lunghezza. In questo esempio, le due sequenze hanno la medesima lunghezza e lo scambio tra le stringhe porta a differenze solo del numero e della posizione del nodi intermedi.

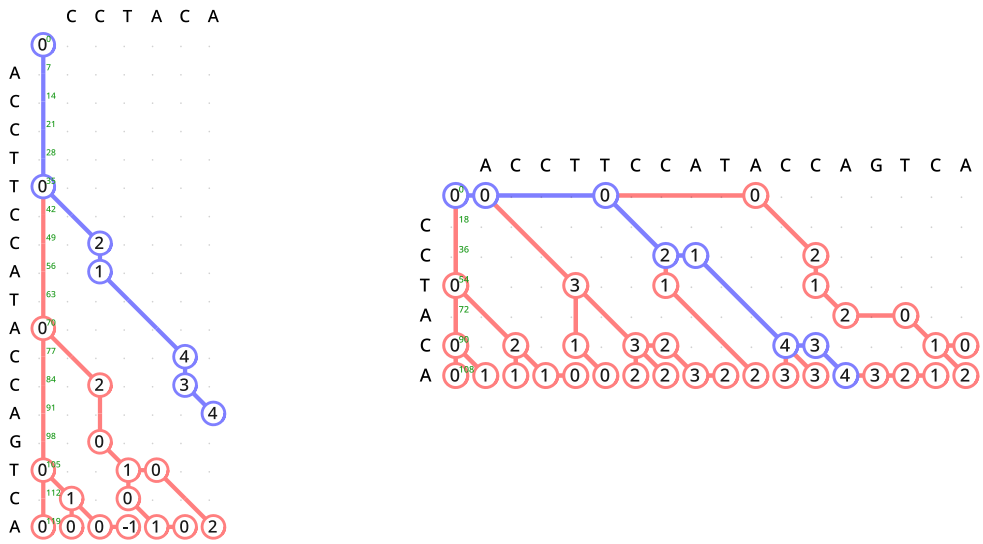


Figura 3.5: Confronto tra l'albero dell'esempio precedente e quello ottenuto dal medesimo allineamento ma scambiando le due sequenze. E evidente come la soluzione migliore sia la disposizione di sinistra, con la sequenza più lunga disposta per colonna e la più corta per riga in modo da ridurre il numero di foglie.

3.1.6 Complessità di tempo

Durante l'esecuzione l'algoritmo deve calcolare un numero di valori pari alle dimensioni della matrice, ovvero $(n + 1) \times (m + 1)$. La complessità per questa fase è quindi $O(nm)$. Dopo il calcolo di ciascun valore, l'algoritmo tenta di eliminare eventuali nodi non più utili, partendo dal nodo che si trova in diagonale. Poiché per ogni nodo valutato si procede con la valutazione del relativo genitore solo nel caso il nodo valutato venga eliminato, si può affermare che il numero di valutazioni ed eliminazioni è anch'esso nell'ordine di $n \cdot m$ considerando l'intera esecuzione. Ogni nodo può infatti essere eliminato al massimo una

volta e, prima di ciò, valutato da una a tre volte (una per ciascun figlio). La complessità di tempo finale è quindi nell'ordine di $O(nm)$.

3.1.7 Ricostruzione dell'allineamento

Per ricostruire l'allineamento ottimale trovato dall'algoritmo individuato sono necessari: le due sequenze (s e t), l'albero calcolato ($tree$), la posizione della cella col massimo punteggio (max_points_pos) tra quelli appartenenti all'ultima riga o colonna della matrice, la larghezza della matrice (pari a $|s| + 1$).

Poiché max_points_pos si trova sull'ultima riga o sull'ultima colonna, ma non necessariamente su entrambe, è necessario distinguere due casi:

- se si trova nella cella all'intersezione tra ultima riga ed ultima colonna, si può partire direttamente dal nodo già presente in $tree$ corrispondente a tale cella
- altrimenti, si introduce un nuovo nodo associato a tale cella e figlio del nodo in indicato da max_points_pos

Partendo dal nodo così individuato o creato, la direzione di avanzamento viene determinata confrontando l'id del nodo corrente con quello del suo nodo genitore e stabilendo così se si deve procedere in direzione verticale (verso l'alto), orizzontale (verso sinistra) o obliqua (verso l'alto a sinistra).

Stabilita la direzione, si procede lungo essa muovendosi di cella in cella finché viene raggiunto il nodo genitore dell'ultimo nodo incontrato. A quel punto, questo nodo genitore diventa il nuovo nodo corrente e si ripete il processo di calcolo della direzione e di movimento. Il ciclo termina quando viene raggiunto il nodo in posizione 0, corrispondente alla cella all'intersezione tra la prima riga e la prima colonna della matrice.

Va ricordato che le posizioni sono rappresentate non da una coppia di indici (di riga e di colonna) ma tramite un indice unico che esegue una scansione della matrice per righe. il Dato un indice w, il numero di riga i è quindi calcolabile come $\lfloor pos/(|s| + 1) \rfloor$, il numero di colonna j è calcolabile come $pos \pmod{|s| + 1}$. Ad ogni spostamento su una nuova cella dobbiamo aggiungere una coppia di valori all'allineamento. Dipendentemente dalla direzione che stiamo seguendo i caratteri da aggiungere sono i seguenti:

- diagonale: aggiungiamo all'inizio dell'allineamento $\binom{s[i]}{t[j]}$
- verticale: aggiungiamo all'inizio dell'allineamento $\binom{-}{t[j]}$
- orizzontale: aggiungiamo all'inizio dell'allineamento $\binom{s[i]}{-}$

È importante notare come la ricostruzione dell'allineamento proceda a ritroso: questo è dovuto al fatto che si parte dalla posizione finale e si risale verso la radice dell'albero e rispecchia la ricostruzione già vista per l'algoritmo Needleman-Wunsch.

Quando il ciclo ha raggiunto la posizione 0 la ricostruzione termina ed il risultato ottenuto rappresenta l'allineamento ottimale trovato dall'algoritmo.

Algoritmo 5 Funzione per la ricostruzione dell'allineamento

```
1:  procedure RECOSTRUCT_ALIGNMENT(max_points_pos, tree, seq_s, seq_t)
2:      n ← len(seq_s)
3:      a ← empty_string
4:      b ← empty_string
5:      end_pos ← len(seq_s) ∗ len(seq_t)
6:      if end_pos ≡ max_points_pos then
7:          cnode ← map[end_pos]
8:      else
9:          cnode = TreeNode{pos : end_pos, parent : max_points_pos}
10:     end if
11:     hmov ← cnode.pos (mod n + 1) ≠ cnode.parent (mod (n + 1))
12:     vmov ← ⌊cnode.pos/(n + 1)⌋ ≠ ⌊cnode.parent/(n + 1)⌋
13:     p ← cnode.pos
14:     while p > 0 do
15:         if vmov then
16:             b.prepend(seq_t[(p/(n + 1) − 1)])
17:         else
18:             b.prepend('−')
19:         end if
20:         if hmov then
21:             a.prepend(seq_s[p (mod (n + 1)) − 1])
22:         else
23:             a.prepend('−')
24:         end if
25:         if vmov then
26:             p ← p − (n + 1)
27:         end if
28:         if hmov then
29:             p ← p − 1
30:         end if
31:         if p ≡ cnode.parent then
32:             cnode ← tree[p]
33:             hmov ← cnode.pos (mod n + 1) ≠ cnode.parent (mod n + 1)
34:             vmov ← ⌊cnode.pos/(n + 1)⌋ ≠ ⌊cnode.parent/(n + 1)⌋
35:         end if
36:     end while
37:     return a, b
38: end procedure
```

3.1.8 Varianti

L'algoritmo può essere facilmente adattato per supportare anche ciascuna delle altre 14 varianti dell'allineamento semiglobale, nonché l'allineamento globale, combinando le seguenti modifiche in base alle necessità:

- Penalizzazione gap ad inizio di una o di entrambe le stringhe: invece di inizializzare la prima riga (o colonna) con zeri si utilizzano valori tra 0 e -n (o -m) come per l'allineamento globale
- Penalizzazione gap alla fine della stringa t: durante la costruzione dell'albero non si memorizza il nodo con punteggio massimo tra quelli a fine riga
- Penalizzazione gap alla fine della stringa s: non si effettua la ricerca del nodo di punteggio massimo tra quelli presenti dell'ultima riga

Ad esempio, se si desidera non penalizzare i gap all'inizio delle due stringhe ma penalizzare quelli al termine di entrambe, la prima riga e la prima colonna vanno inizializzate entrambe con degli zero e l'allineamento deve essere ricostruito partendo dall'ultimo elemento della matrice.

Se si desidera, invece, penalizzare i gap all'inizio delle due stringhe ma non quelli al termine di entrambe, vanno compilate la prima riga con valori da 0 a $-n$ e la prima colonna con valori tra 0 e $-m$. Durante l'esecuzione va tenuta traccia del nodo con massimo punteggio tra quelli dell'ultima colonna e durante il calcolo dell'ultima riga va verificato se c'è un nodo con punteggio maggiore a quello già trovato.

3.2 Variante algoritmo: divide&conquer

È stato finora spiegato come effettuare il pruning in modo da ridurre enormemente il numero di nodi da tenere in memoria durante la costruzione dell'albero. È possibile, con una modifica puntuale all'algoritmo, ridurre ulteriormente il numero di nodi rimuovendo anche quei nodi che, pur non essendo branching, sono stati mantenuti perché fulcro di un cambio di direzione. Questo causa una perdita di informazione in quanto non si è più in grado di ricostruire il percorso tra due branching nodes dell'albero. Siccome in un albero il numero di branching nodes è obbligatoriamente minore del numero di foglie, e quindi in questo algoritmo è minore del numero di nodi della frontiera ($m + 2$ al massimo come spiegato in precedenza), questa modifica garantisce l'abbassamento della complessità di spazio dell'algoritmo a $O(m)$. Rende però necessario rieseguire l'allineamento su ciascuna delle sottomatrici individuate dalle coordinate di ciascuna coppia di branching nodes consecutivi, ad eccezione dei casi in cui essi si trovano sulla stessa colonna o sulla stessa riga e sia quindi banale il percorso tra essi - risultante cioè in un allineamento tra una sottosequenza ed una serie di gap.

Questa variante può risultare utile nel caso di sequenze estremamente lunghe e per le quali non si riesce ad eseguire algoritmi che forniscono la soluzione completa ma con una richiesta di spazio maggiore.

Nell'esempio di *C. elegans* col gene let-60 si ottiene un albero di 4.583 nodi ed un percorso migliore che passa da una lista di nodi cruciali: l'elemento di partenza, sei branching nodes, il nodo finale. Le coordinate di questi nodi cruciali individuano alcune sottomatrici che andranno ricalcolate con un algoritmo che tenga traccia dell'allineamento completo oppure ulteriormente divisi in sottoproblemi a seconda della loro complessità, le dimensioni di queste sottomatrici nell'esempio in questione sono di 2.951.424, 442.240, 15.312, 8.712 ed infine 26.600 elementi, a fronte di una matrice di partenza che era di 8.133.782 elementi. La dimensione delle sottomatrici è strettamente legata alla relazione tra i vari branching nodes ed alla loro posizione.

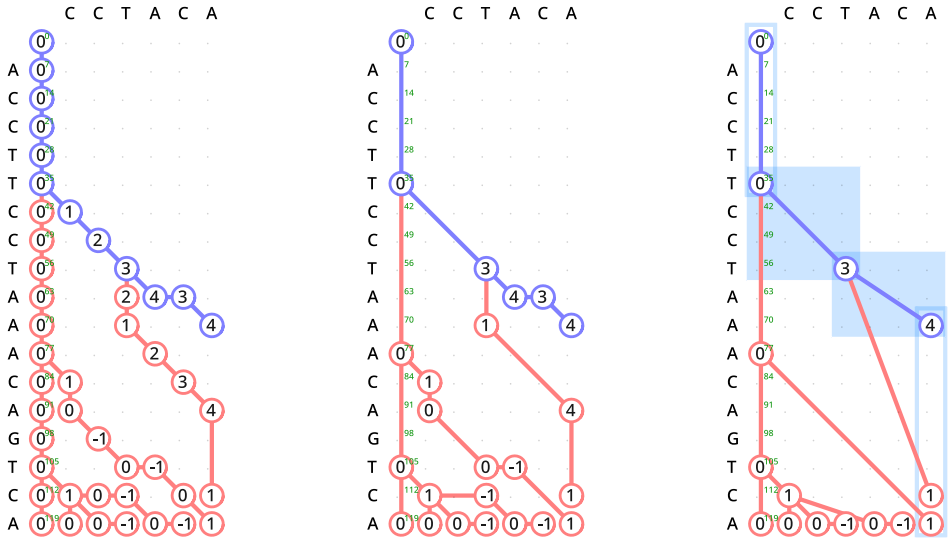


Figura 3.6: Confronto tra l'albero senza compattazione, con compattazione normale e con compattazione estrema dei percorsi. I rettangoli con bordo azzurro individuano le sottomatrici banali mentre i rettangoli con sfondo azzurro individuano le sottomatrici che richiedono un ricalcolo

L'allineamento rappresentato dalla Figura 3.6 è il seguente, nelle due versioni con soluzione completa e divide&conquer. In marrone sono indicati i nodi mantenuti in memoria come nodi di passaggio dell'allineamento ottimale, in verde sono segnate le parti di allineamento che vanno ricalcolate. La prima riga contiene gli id dei nodi in memoria.

$$
\begin{array}{c}
0 \cdot\cdot\cdot\cdot\cdot 35 \cdot\cdot\cdot 59 \cdot 67 \cdot 68 \cdot 76 \cdot\cdot\cdot\cdot\cdot\cdot\cdot \\
|-----|\ CCT\ |\ A\ |\ C\ |\ A\ |\ -------- \\
|ACCTT\ |\ CCT\ |\ A\ |\ -\ |\ A\ |\ ACAGTCA
\end{array}
$$

Esempio di allineamento semiglobale con punteggio 4 (5 *match*, 0 *mismatch*, 1 *gap*). Il nodo 59 rimane in memoria perchè un secondo allineamento, non ottimale, si divide partendo da tale nodo.

$$
\begin{array}{c}
0 \cdot\cdot\cdot\cdot\cdot 35 \quad \cdot \quad 59 \quad \cdot \quad 76 \cdot\cdot\cdot\cdot\cdot\cdot\cdot \\
|-----|\ [CCT]\ |\ [ACA]\ |\ -------- \\
|ACCTT\ |\ [CCT]\ |\ \ [AA]\ \ |\ ACAGTCA
\end{array}
$$

Esempio di allineamento semiglobale nella variante divide&conquer.

3.2.1 Costruzione

In questa variante, l'algoritmo prevede di rimuovere un non-branching node anche se è il fulcro di un cambio di direzione. È quindi sufficiente rimuovere il controllo su tale condizione e chiamare *skip_node* in qualunque caso il nodo abbia un unico figlio, indipendentemente dalla posizione di tale figlio e del genitore.

Algoritmo 6 Nuova tree_prune per la compattazione estrema dell'albero

```
1:  procedure TREE_PRUNE(w, tree, protected)
2:      current_id ← w
3:      children ← 0
4:      while current_id ≠ protected do
5:          current_node ← tree[current_id]
6:          children ← current_node.children.len()
7:          if current_node.children is not empty then
8:              break
9:          end if
10:         parent_node ← tree[current_node.parent]
11:         remove current_id from parent_node.children
12:         remove current_node from tree
13:         current_id ← n.parent
14:     end while
15:     if current_id ≠ protected ∧ current_id is not root ∧ children is 1 then
16:         skip_node(current_id, tree)          ▷ Eseguito senza l'If prima presente
17:     end if
18: end procedure
```

3.2.2 Ricostruzione dei sotto-problemi

Questo metodo calcola e restituisce una lista di coppie di coordinate all'interno della matrice. Muovendosi lungo la lista con una finestra di dimensione 2, cioè prendendo coppie interlacciate di coordinate, si ottengono gli estremi delle sottomatrici all'interno delle quali va ricalcolato l'allineamento parziale. Nell'esempio in Figura 3.6 si ottengono le coppie (0,35), (39,59), (59,76), (76,125). Siccome i nodi 0 e 35 si trovano sulla medesima colonna, come i nodi 76 e 125, queste coppie individuano sottoproblemi banali e l'unico percorso disponibile in questi sottoproblemi è verticale, in altre parole l'allineamento in questi tratti vede solo gap per la stringa s e solo non-gap per la stringa t. Le due coppie restanti individuano invece delle sottomatrici su cui si dovrà effettuare nuovamente i calcoli necessari per trovare nuovamente il percorso. Trattandosi di sottomatrici, la loro dimensione è inferiore a quella della matrice iniziale e possono risultare quindi affrontabili con questo stesso algoritmo ma nella variante che fornisce la soluzione completa, oppure con altri algoritmi. Le soluzioni trovate per ciascuna sottomatrice andranno concatenate per ottenere l'allineamento completo.

3.2.3 Complessità di spazio

La modifica appena descritta permette di ottenere un albero contenente i soli *branching nodes* e le foglie. Il numero di foglie di questo albero corrisponde alla larghezza della matrice, pari a sua volta alla lunghezza di una delle due sequenze incrementata di uno, a cui va aggiunta una eventuale foglia data dall'allineamento ottimale nel caso in cui questo non si trovi nell'ultima riga della matrice. Scegliendo opportunamente di disporre orizzontalmente la stringa più corta tra le due, la lunghezza di una riga della matrice corrisponde a $min(n, m)+1$. Poiché ogni nodo che non sia foglia è un *branching node* ed il numero massimo di foglie è $min(n, m) + 2$,

Algoritmo 7 Funzione per la ricostruzione dei sottoproblemi

```
1:  procedure RECOSTRUCT_SUBPROBLEMS(max_points_pos, tree, seq_s, seq_t)
2:      n ← len(seq_s)
3:      end_pos ← len(seq_s) * len(seq_t)
4:      if end_pos ≡ max_points_pos then
5:          cnode ← tree[end_pos]
6:      else
7:          cnode = TreeNode{pos : end_pos, parent : max_points_pos}
8:      end if
9:      pos ← cnode.pos
10:     parent ← cnode.parent
11:     couples ← empty_array
12:     while pos > 0 do
13:         couples.append(parent%(n + 1), ⌊parent/(n + 1)⌋)
14:         cnode ← tree[parent]
15:         pos ← cnode.pos
16:         parent ← cnode.parent
17:     end while
18:     couples.reverse()
19:     return couples.windows(2)
20: end procedure
```

si può dimostrare che il numero di nodi totali è minore o uguale a $2 \cdot (min(n, m) + 2) - 1$ o, riscritto in maniera più compatta, $2 \cdot min(n, m) + 3$.

3.2.4 Complessità di tempo

L'analisi della complessità di tempo di questa versione dell'algoritmo è la medesima di quanto spiegato nel Paragrafo 3.1.6. L'unica differenza in termini di elaborazione è infatti la rimozione del nodo finale durante la procedura di *tree_pruning* senza il controllo prima presente. Si deve tuttavia considerare che in questa modalità l'algoritmo non fornisce una soluzione completa per l'allineamento ma obbliga a rieffettuare l'allineamento su alcuni sottoproblemi, la complessità di tempo di questi ricalcoli dipende dall'algoritmo scelto per effettuarli.

3.2.5 Riduzione della dimensione dei sottoproblemi

All'inizio di questa sezione è stato riportato un esempio riguardante il gene let-60 di *C. elegans* ed è stato detto che la dimensione delle sottomatrici su cui rieffettuare i calcoli è strettamente legata alla relazione tra i vari branching nodes ed alla loro posizione. In particolare, nell'esempio riportato una delle sottomatrici ha una dimensione di 2.951.424 elementi, una dimensione molto maggiore delle altre sottomatrici e non trascurabile rispetto alla dimensione della matrice totale. Ciò è dovuto al fatto che i due branching nodes che la individuano sono molto distanti tra loro. Questo problema può essere tenuto sotto controllo mantenendo in memoria un nodo, anche nel caso non sia un branching node, se la distanza

dal nodo precedente supera una soglia definita. Questa modifica aumenta in piccola misura il numero di nodi mantenuti in memoria ma definisce ed assicura una dimensione massima delle sottomatrici su cui è necessario effettuare nuovamente i calcoli. Si tratta di un compromesso: la scelta della soglia consente di spostare l'equilibrio tra consumo di memoria ed consumo di potenza di calcolo, in modo da adattarsi alle esigenze specifiche caso per caso. Per questione di sintesi non viene riportato, in questa tesi, il codice dettagliato; si fornisce comunque una descrizione sommaria dell'implementazione:

- Per ogni nodo si tiene traccia della profondità nell'albero, pari normalmente a quella del suo nodo genitore incrementata di uno
- Se il genitore è un branching node, la profondità è impostata a 1
- Se la profondità supera la soglia, il nodo non viene eliminato dalla procedura $tree\_pruning$ pur non trattandosi di un branching node
- I nodi figli di un nodo che a causa del superamento della soglia è stato mantenuto hanno una profondità pari a 1

Questa soluzione porta il numero di nodi da mantenere in memoria a crescere: minore è la soglia impostata, maggiore è il numero di nodi da memorizzare. Se la soglia è ridotta il numero di nodi da memorizzare si avvicina al numero memorizzato nella versione con salvataggio completo del percorso; se la soglia è elevata il numero di nodi si avvicina a quello memorizzato dalla versione divide&conquer.

3.3 Rappresentazione alternativa per i percorsi

Viene qui proposta un'ulteriore ottimizzazione per il salvataggio dei nodi, di cui è stato sviluppato un prototipo preliminare che non verrà presentato in questa tesi.

Un approccio alternativo per salvare i percorsi è utilizzare delle maschere di bit. Questo approccio consiste nel memorizzare come nodi dell'albero soltanto i branching nodes ed utilizzare due sequenze di bit per rappresentare all'interno di un nodo il percorso tra esso ed il relativo nodo genitore. Il salvataggio consiste nell'impostare ad 1 il bit corrispondente sul primo numero se il primo movimento ha una componente verticale ed il bit corrispondente sul secondo numero se il primo movimento ha una componente orizzontale. In questo modo è possibile memorizzare nell'albero i soli branching nodes e le foglie, mantenendo comunque le informazioni necessarie alla ricostruzione completa dell'allineamento. I due numeri interi devono avere un numero di bit sufficiente a coprire l'intero percorso tra un nodo ed il relativo genitore, non sono quindi limitati a 64bit né ad altri valori fissi. In Python 3.x questo aspetto è gestito automaticamente[23], in altri linguaggi è possibile realizzare un'implementazione ad hoc. È importante, comunque, osservare che con questo metodo non si salvano tutti i dati in $O(m)$, la complessità data dal numero di nodi dell'albero in sé, ma in $O(m*x)$ dove x dipende linearmente dalla lunghezza media di un percorso che unisce due nodi dell'albero. Si deve tenere conto che un calcolatore moderno rappresenta solitamente un numero in 64bit quindi si potrebbe scrivere $O(m*\lceil\frac{x}{64}\rceil)$, anche se nel dettaglio dipende dall'esatta implementazione della memorizzazione dei valori.

Facendo riferimento alla Figura 3.6, il percorso ottimale viene così memorizzato con quattro nodi, ognuno correlato da due numeri indicanti gli spostamenti verticali ed orizzontali che separano un nodo dal proprio nodo genitore, nel modo visibile in Tabella 3.1.

nodo	genitore	movimento verticale	movimento orizzontale	
0				
35	0	31 (11111)	0 (00000)	(3.1)
59	35	7 (111)	7 (111)	
76	59	5 (101)	7 (111)	

Tabella 3.1: Memorizzazione con maschere di bit del percorso ottimale per l'esempio in Figura 3.6

Nei nodi 35 e 59 si può, come forma di ottimizzazione, eliminare le maschere di bit in quanto i percorsi da esse rappresentante non hanno cambi di direzione ed è quindi possibile procedere, durante la ricostruzione dell'allineamento, calcolando la direzione da seguire in base alla posizione del nodo genitore. Lo svantaggio di questo metodo di memorizzazione è che l'occupazione in memoria è linearmente dipendente dalla lunghezza del percorso, indipendentemente dalla quantità di aperture e di chiusure di *gap*. Il metodo spiegato in 3.1, invece, permette di risparmiare molto spazio nel caso che il percorso abbia pochi cambi di direzione. Un sistema di memorizzazione di questo tipo può risultare però vantaggioso se usato in combinazione con il sistema visto in precedenza: nel caso di lunghi tratti senza

cambi di direzione si memorizzano i nodi di inizio e fine del tratto, nel caso di una parte di percorso densa di cambi di direzione si memorizzano soltanto i nodi di inizio e fine e si utilizza il metodo appena spiegato per i nodi interni a questa parte.

Facendo di nuovo riferimento alla Figura 3.6 ed immaginando di avere un computer a 2 bit (viste le dimensioni ridotte dell'esempio), l'algoritmo con questo sistema ibrido dovrebbe memorizzare i nodi 0, 35, 59, 68 e 76, come visibile in Tabella 3.2.

nodo	genitore	movimento verticale	movimento orizzontale
0			
35	0		
59	35		
68	59	2 (10)	3 (11)
76	68		

(3.2)

Tabella 3.2: Memorizzazione ibrida del percorso ottimale per l'esempio in Figura 3.6

Il nodo 68 ora compare perché avendo introdotto il limite di 2 bit non è possibile memorizzare interamente il percorso tra il nodo 59 ed il nodo 76 in quanto si tratta di un percorso di lunghezza 3. Anche la distanza tra il nodo 0 ed il nodo 35 è di lunghezza superiore a 2 ma in questo caso non è necessario salvarlo in quanto privo di cambi di direzione e quindi deducibile al momento della ricostruzione dell'allineamento.

Anche se è realizzato un prototipo del sistema di memorizzazione tramite bit si è scelto di non portare in questa sede l'implementazione del sistema ibrido appena spiegato per non complicare ulteriormente il codice.

È stata effettuato un calcolo sull'albero generato dall'allineamento visto per il gene let-60 dell'organismo *C. elegans* e si è visto che su una macchina a 32bit verrebbero utilizzati 6.094 nodi, su una a 64bit 5.275 nodi, su una a 128bit 4.883 nodi.

Capitolo 4

Allineamento di D-strings e ED-strings

Applicando alcune modifiche all'algoritmo descritto nel precedente capitolo è possibile adattarlo per lavorare con le ED-strings. Nell'implementazione di seguito descritta, la stringa identificata negli scorsi capitoli come t può essere una ED-string mentre l'altra, quella identificata come s, deve restare una stringa solida. Questo non rappresenta un limite concettuale dell'algoritmo ma solo una scelta per la sua attuale implementazione: è possibile estendere l'algoritmo per effettuare un allineamento tra due ED-strings, applicando anche alle righe le stesse accortezze che verranno descritte nel presente capitolo per le colonne. Questo renderebbe però il procedimento ed il relativo codice più complessi da visualizzare e comprendere, si è quindi preferito mantenere più semplice la descrizione.

Una ED-string è composta da alcuni blocchi fissi ed alcuni blocchi alternativi tra loro. Il principio di base è quello di poter effettuare una sola volta i calcoli sui blocchi fissi. Procedendo riga per riga, ogni volta che si incontra un gruppo di alternative va tenuto in considerazione che tutte rappresentano la prosecuzione dell'allineamento calcolato fino al carattere precedente all'inizio delle alternative.

Se si rimuovono dalla ED-string in input i caratteri non alfabetici, ovvero quelli che delimitano le alternative, si ottiene una stringa a cui è possibile applicare l'algoritmo descritto nel Capitolo 3. Durante il calcolo bisogna però considerare che i valori presenti nella prima riga di ciascuna alternativa non devono derivare dai valori nella riga immediatamente precedente nella matrice, bensì da quelli presenti nella riga corrispondente al carattere immediatamente precedente all'inizio del gruppo di alternative nella ED-string originale. Quando l'ultima riga di ciascuna delle alternative è stata compilata, si può calcolare la riga associata al primo carattere successivo al termine del gruppo. Tale calcolo dovrà essere effettuato scegliendo, cella per cella, il miglior valore tra quelli in posizione corrispondente nelle righe finali di tutte le alternative.

$$
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
A & A & A & [T & T & T & | & C & C] & A & A & A & T & G & G & A & A & A \\
A & A & A & T & T & T & & & & A & A & A & T & G & G & A & A & A \\
A & A & A & & & & & C & C & A & A & A & T & G & G & A & A & A
\end{array}
\tag{4.1}
$$

L'esempio 4.1 mostra una ED-string e le due varianti identificate dalle alternative possibili, rappresentate separatamente

Come è possibile vedere, nella prima variante della ED-string di esempio il carattere identificato dal numero di colonna 4 segue il carattere identificato dal numero di colonna 3 mentre il carattere in colonna 9 segue il carattere in colonna 6. In modo simile, nella seconda variante il carattere in colonna 7 segue il carattere in colonna 3 mentre il carattere in colonna 9 segue il carattere in colonna 8. Utilizzando la stringa solida ottenuta concatenando le alternative una all'altra si deve tenere conto di questi salti. Ricordando che la ED-string costituisce la stringa t, è possibile seguire l'algoritmo esposto nel Capitolo 3 ed effettuare il calcolo dei valori della riga 4 basandosi sui valori della riga 3, i valori della riga 7 basandosi ancora sui valori della riga 3 ed arrivare così alla riga 9. Arrivati a questo punto le righe $1, 2, 3, 4, 5, 6$ contengono i valori come in un normale allineamento tra due stringhe solide considerando in t la stringa solida contenente la prima alternativa mentre le righe $1, 2, 3, 7, 8$ contengono i valori come in un normale allineamento tra due stringhe solide considerando in t la stringa solida contenente la seconda alternativa. Arrivati al calcolo dei valori della riga 9 ci si trova nella seguente situazione:

- Nella riga 6 sono presenti i valori dei migliori allineamenti utilizzando la prima alternativa
- Nella riga 8 sono presenti i valori dei migliori allineamenti utilizzando la seconda alternativa

L'obiettivo dell'algoritmo è trovare l'allineamento col punteggio maggiore possibile, a prescindere dalla variante. Per farlo, si deve basare quindi il calcolo della riga 9 su una riga fittizia costituita a partire, posizione per posizione, dal miglior valore tra quello della riga 6 e quello della riga 8. Una volta calcolata la riga 9 l'algoritmo può proseguire riga dopo riga come descritto per le stringhe solide, senza altre particolarità.

4.1 Implementazione

Le modifiche interessano diversi punti dell'algoritmo ma riguardano tutte la gestione delle righe particolari appena descritta. Per quanto riguarda il calcolo dei valori gli interventi riguardano l'apertura di ciascuna nuova alternativa e la ripresa dell'allineamento su una parte di stringa solida. Anche durante la procedura di *tree_pruning* e la fase di ricostruzione della soluzione è necessario tenere conto dei salti effettuati nei calcoli.

4.1.1 Derivazione dei valori di una riga

Nel Capitolo 3 è stato descritto come i valori di ogni riga vengono calcolati a partire dai valori della riga immediatamente precedente. Come appena visto, con le *D-strings* e le *ED-strings* il calcolo deve essere più raffinato per alcune righe. Per chiarire il funzionamento è utile identificare alcune righe come nell'esempio seguente:

$$\text{AAA}\left\{\begin{array}{c}\text{TTT}\\ \text{CC}\\ \text{AGGG}\end{array}\right\}\text{GGAAA}$$

1	2	3	4	5	6		7	8		9	10	11	12	13	14	15	16	17
A	A	A	$[T$	T	T	$\|$	C	C	$\|$	A	G	G	$G]$	G	G	A	A	A
		s_t	s_1		e_1		s_2	e_2		s_3			e_3	e_t				

Si definisce quindi s_t l'ultima riga prima dell'inizio delle alternative, s_i la riga di inizio dell'alternativa numero i; e_i la riga finale dell'alternativa numero i; ed infine e_t la riga successiva alla conclusione delle alternative.

Il calcolo dei valori di una riga segue ora i seguenti criteri, a seconda della riga da calcolare:

- s_i: i valori dipendono dalla riga s_t
- e_t: i valori dipendono dal massimo, colonna per colonna, tra i valori di tutte le righe e_i
- in tutti gli altri casi: i valori dipendono dalla riga immediatamente precedente

Va quindi modificata la funzione $BUILD\_TREE$ descritta nel capitolo precedente in modo da calcolare la riga che deve essere usata per i calcoli mentre si procede di riga in riga, la nuova funzione è indicata in Algorithm 5. Come si può osservare confrontando gli algoritmi 1 e 2, quest'ultimo è una rielaborazione del primo, principalmente introducendo tre aspetti:

- trasformazione della ED-string in input in stringa solida
- geestione più complessa della scelta dei nodi da mantenere in memoria in alcune righe particolari
- calcolo della riga di riferimento per il calcolo della riga corrente

4.1.2 Tree pruning

Nella funzione $BUILD\_TREE$ nell'Algoritmo 2 vengono passati due parametri aggiuntivi alla funzione $TREE\_PRUNE$. Questi due parametri servono a inibire l'azione della funzione in due situazioni:

Algoritmo 2 Costruzione dell'albero per seq_s stringa solida e seq_t ED-string

```
1: procedure BUILD_TREE(x, y)
2:     (y, dependences) ← create_concatenated_alternatives_string(y)
3:     (lines_to_keep, dont_skip) ← analyze_dependences(dependences)
4:     tree ← empty_dictionary
5:     max_points ← 0
6:     max_pos ← 0
7:     n ← len(seq_s)
8:     m ← len(seq_t)
9:     create_node(0, 0, tree)
10:    for i ← 1 to n do
11:        create_node(i, i-1, 0, tree)
12:    end for
13:    for j ← 1 to m do
14:        points ← max(0, tree[(j − 1) ∗ (n + 1)].points + gap)
15:        create_node(j ∗ (n + 1), (j − 1) ∗ (n + 1), points, tree)
16:        if j > 1 then
17:            tree_prune((j − 1) ∗ (n + 1), tree, max_pos, lines_to_keep, dont_skip)
18:        end if
19:        uprow ← get_uprow(tree, dependences, j)
20:        for i = 1 to n do
21:            w ← i + (j ∗ (n + 1)); wdiag ← uprow[i − 1]; wup ← uprow[i]
22:            match_mismatch ← tree[wdiag].points + (seq_s[i − 1] ≡ seq_t[j −
    1]?match_score : mismatch_score)
23:            del_score ← tree[wup].points + gap
24:            ins_score ← tree[w − 1].points + gap
25:            if match_mismatch > ins_score and match_mismatch > del_score then
26:                create_node(w, wdiag, match_mismatch, tree)
27:            else if del_score > ins_score then
28:                create_node(w, wup, del_score, tree)
29:            else
30:                create_node(w, w-1, ins_score, tree)
31:            end if
32:            tree_prune(wdiag, tree, max_pos, lines_to_keep, dont_skip)
33:        end for
34:        last_idx ← (j + 1) ∗ (n + 1) − 1
35:        if tree[last_idx] > max_score then
36:            if max_pos > 0 ∧ max_pos < (j − 1) ∗ (n + 1) ∧ max_pos is not leaf then
37:                tree_prune(max_pos, tree, 0)
38:            end if
39:            max_score ← tree[last_idx].score; max_pos ← last_idx
40:        end if
41:        clean_dependences(dependences, j, dont_skip, lines_to_keep, max_pos, (n+
    1))
42:    end for
43:    for i = m ∗ (n + 1) to (m + 1) ∗ (n + 1) do
44:        if tree[i] > max_score then
45:            max_score ← tree[i].score; max_pos ← i
46:        end if
47:    end for
48:    return tree, max_points, max_points_idx
49: end procedure
```

Algoritmo 3 Pulizia di dependences dopo la chiusura di una sequenza di alternative

```
1: procedure clean_dependences(dependences, j, dont_skip, lines_to_keep, max_pos, n)
2:     if j in dependences.keys() then
3:         if dependences[j].len() > 2 then  ▷ This is a closing-alternative node, we can clean up all the previously blocked rows!
4:             remove dependences[j][0] from dont_skip
5:             remove all dependences[j] values from lines_to_keep
6:             for d in dependences[j] do
7:                 for i ← 0 to (n + 1) − 1 do
8:                     w ← d ∗ (n + 1) + i
9:                     tree_prune(w, tree, max_pos, n, lines_to_keep, dont_skip)
10:                end for
11:            end for
12:        end if
13:    end if
14: end procedure
```

Algoritmo 4 Trasformazione della ED-string in stringa solida ed elaborazione dipendenze tra righe

```
1: procedure CREATE_CONCATENATED_ALTERNATIVES_STRING(seq)
2:     faked ← empty string
3:     dependences ← empty dictionary
4:     start ← 0
5:     derivates ← [ ]
6:     for c in seq do
7:         if c in ['A','C','G','T'] then
8:             faked.append(c)
9:         else if c =='[' then
10:            building_alternative ← 1
11:            start ← faked.len() − 1
12:            dependences[faked.len() + 1] ← [start + 1]
13:        else if c ==']' then
14:            derivates.append(faked.len())
15:            derivates.prepend(start + 1)
16:            dependences[end] ← derivates
17:            derivates ← [ ]
18:        else if c =='|' then
19:            derivates.append(faked.len())
20:            dependences[faked.len() + 1] ← [start + 1]
21:        end if
22:    end for
23:    return (faked, dependences)
24: end procedure
```

Algoritmo 5 Calcolo della riga da cui derivare i valori per una nuova riga

```
1:  procedure GET_UPROW(tree, dependences, j, n)
2:      uprow ← [(j − 1) ∗ (n + 1)..j ∗ (n + 1)]
3:      if j in dependences.keys() then
4:          deps ← dependences[i]
5:          if deps.len() == 1 then                    ▷ Caso riga s_i, deps è [s_t]
6:              uprow = [deps[0] ∗ (n + 1)..(deps[0] + 1) ∗ (n + 1)]
7:          else                                       ▷ Caso riga e_t, deps è [s_t, e_1, ..., e_i]
8:              for i ← 0 to (n + 1) do
9:                  max ← 1
10:                 points_max ← tree[deps[max] ∗ (n + 1) + i].points
11:                 for d in 2..deps.len() do
12:                     if tree[deps[d] ∗ (n + 1) + i].points > points_max then
13:                         max ← d
14:                         points_max ← tree[deps[max] ∗ (n + 1) + i].points
15:                     end if
16:                 end for
17:                 uprow.append(deps[max] ∗ (n + 1) + i)
18:             end for
19:         end if
20:     else
21:         uprow ← [(j − 1) ∗ (n + 1)..j ∗ (n + 1)]
22:     end if
23:     return uprow
24: end procedure
```

Algoritmo 6 Creazione di lines_to_keep e dont_skip

```
1:  procedure ANALYZE_DEPENDENCES(dependences)
2:      lines_to_keep ← [ ]
3:      dont_skip ← [ ]
4:      for dep in dependences do
5:          for d in dep do
6:              if d not in lines_to_keep then
7:                  lines_to_keep.append(d)
8:              end if
9:              if d not in dont_skip then
10:                 dont_skip.append(d)
11:             end if
12:         end for
13:     end for
14:     return lines_to_keep, dont_skip
15: end procedure
```

- *lines_to_keep*: viene usato per fermare la risalita del ciclo di pruning dal nodo di partenza verso la radice dell'albero quando incontra un nodo posizionato in una delle righe specificate

- *don_skip*: viene usato per inibire l'eliminazione del nodo su cui il ciclo si è interrotto nel caso in cui questo appartenga a determinate righe

Nella spiegazione seguente si fa riferimento al caso di una stringa con un un solo gruppo di alternative, come nell'esempio, per semplicità. In presenza di più gruppi la medesima logica si applica ad ogni gruppo in maniera indipendente.

All'inizio dell'esecuzione dell'algoritmo, questi parametri contengono entrambi liste di numeri di riga corrispondenti alle righe di tipo s_t e e_i. Dopo il calcolo della riga e_t viene rimosso da $dont\_skip$ solo il numero di riga s_t mentre da $lines\_to\_keep$ vengono rimossi tutti i numeri di riga s_t ed e_i. Lo scopo di questa distinzione tra le due liste è la seguente: la prima parte della procedura di pruning si occupa di risalire verso la radice eliminando, se possibile, l'intero ramo su cui sta agendo fino al primo branching node incontrato; in questa fase è importante poter attraversare anche un gruppo di alternative per rimuovere tutti i nodi non più utili. La seconda parte della procedura, invece, valuta la rimozione di un nodo nel caso questo non comporti un cambio di direzione (ovvero quando non è necessario per ricostruire correttamente l'allineamento). In questa fase, mantenendo tutti gli e_i in $dont\_skip$, si evita l'eliminazione di un nodo posizionato nell'ultima riga di un'alternativa. Considerato che un nodo che si trova in una riga di tipo e_t può avere come genitore un nodo che si trova in una qualunque delle righe di tipo e_i, questa accortezza garantisce che si possa ricostruire l'allineamento nella maniera corretta e senza ambiguità.

Queste modifiche alla funzione $TREE\_PRUNE$ comportano il mantenimento di alcuni nodi aggiuntivi che potrebbero diventare non più necessari dopo aver calcolato la prima riga successiva ad un gruppo di alternative, cioè una riga di tipo e_t. Per questo motivo si può vedere in $BUILD\_TREE$ la chiamata alla funzione aggiuntiva $CLEAN\_DEPENDENCES$, evidenziata in verde. Questa nuova funzione si occupa non solo di effettuare le già descritte modifiche a $line\_to\_keep$ e $dont\_skip$, ma anche di rieseguire la procedura $tree\_prune$ per quei nodi che erano stati mantenuti in quanto appartenenti alle righe di tipo e_i.

Oltre alle modifiche effettuate per la gestione delle ED-strings, indicate in verde, in questa versione di $TREE\_PRUNE$ è stata integrata la possibilità di utilizzare o meno la compattazione estrema descritta nell'Algoritmo 6, in marrone, tramite la costante `TREE_MODE`. Impostando tale costante a `true` si procede con la rimozione del nodo indipendentemente dall'allineamento tra i tre nodi (nodo corrente, nodo genitore, nodo figlio). Impostanto invece tale costante a `false` l'allineamento spaziale tra i tre viene verificato per decidere se procedere o meno con la rimozione del nodo corrente, come nella versione vista in precedenza.

Algoritmo 7 Nuova tree_prune per la compattazione dell'albero con ED-string

```
1:  procedure TREE_PRUNE(w, tree, protected, n, lines_to_keep, dont_skip)
2:      current_id ← w
3:      children ← 0
4:      row ← ⌊current_id/(n + 1)⌋
5:      while current_id ≠ protected ∧ row not in lines_to_keep do
6:          current_node ← tree[current_id]
7:          children ← current_node.children.len()
8:          if current_node.children is not empty then
9:              break
10:         end if
11:         parent_node ← tree[current_node.parent]
12:         remove current_id from parent_node.children
13:         remove current_node from tree
14:         current_id ← current_node.parent
15:         row ← ⌊current_id/(n + 1)⌋
16:         if children > 0 then
17:             break
18:         end if
19:     end while
20:     current_node ← tree[current_id]
21:     row_parent ← ⌊current_node.parent/(n + 1)⌋
22:     if row_parent in dont_skip ∨ row in dont_skip then
23:         return
24:     end if
25:     if current_id ≠ protected ∧ current_id is not root ∧ children is 1 then
26:         if TREE_MODE then
27:             skip_node(current_id, tree)
28:         else
29:             child_id ← id of the only child of current_node
30:             parent_id ← id of the parent of current_node
31:             if parent_id, current_id and child_id are on the same row, column or diagonal
    then
32:                 skip_node(current_id, tree)
33:             end if
34:         end if
35:     end if
36: end procedure
```

4.1.3 Ricostruzione allineamento

La ricostruzione dell'allineamento segue quanto già visto in 3.1.7 per la ricostruzione dell'allineamento tra stringhe solide, ad eccezione delle righe di tipo e_t. In questo caso, l'algoritmo di ricostruzione non può proseguire seguendo una direzione tra verticale, orizzontale o diagonale ma deve effettuare un salto alla riga di tipo e_i corrispondente all'alternativa che fornisce un allineamento ottimale. Questo può essere facilmente effettuato in quanto durante la costruzione dell'albero l'algoritmo ha mantenuto in memoria per ciascun percorso i nodi posti sulla riga di rientro dalla serie di alternative, e_t, ed il relativo nodo genitore posto sulla riga di tipo e_i dell'alternativa interessata. L'algoritmo di ricostruzione, nel momento in cui si trova su una riga di rientro, deve quindi passare al genitore del nodo corrente senza applicare le logiche per gli altri spostamenti. Questa unica modifica all'algoritmo di ricostruzione garantisce un funzionamento corretto per la versione dell'algoritmo di allineamento vista nel presente capitolo; questa versione modificata può essere usata anche per l'allineamento tra stringhe solide visto nel capitolo precedente impostando come parametro $dependences$ un dizionario vuoto. È importante ricordare che nella variante con $TREE\_MODE$ a `true` non è possibile ricostruire l'allineamento completo e si ottiene invece una suddivisione dell'allineamento in sotto-allineamenti da dovere ricalcolare, come descritto nella Sezione 3.2 relativamente all'allineamento tra stringhe solide.

Algoritmo 8 Ricostruzione allineamento per seq_s stringa solida e seq_t ED-string

```
1: procedure RECOSTRUCT_ALIGNMENT(max_points_pos, tree, seq_s, seq_t, dependences)
2:     n ← len(seq_s)
3:     a ← empty_string
4:     b ← empty_string
5:     if end_pos ≡ max_points_pos then
6:         cnode ← map[end_pos]
7:     else
8:         cnode = TreeNode{pos : end_pos, parent : max_points_pos}
9:     end if
10:    hmov ← cnode.pos (mod n + 1) ≠ cnode.parent (mod n + 1)
11:    vmov ← ⌊cnode.pos/(n + 1)⌋ ≠ ⌊cnode.parent/(n + 1)⌋
12:    p ← cnode.pos
13:    while p > 0 do
14:        if vmov then b.prepend(seq_t[(⌊p/(n + 1)⌋ − 1)])
15:        else b.prepend('−')
16:        end if
17:        if hmov then a.prepend(seq_s[p (mod (n + 1)) − 1])
18:        else a.prepend('−')
19:        end if
20:        if row_number in dependences and dependences[row_number].len() > 2
   then
21:            p ← cnode.parent
22:        else
23:            if vmov then p ← p − (n + 1)
24:            end if
25:            if hmov then p ← p − 1
26:            end if
27:        end if
28:        if p ≡ cnode.parent then
29:            cnode ← tree[p]
30:            hmov ← cnode.pos (mod n + 1) ≠ cnode.parent (mod n + 1)
31:            vmov ← ⌊cnode.pos/(n + 1)⌋ ≠ ⌊cnode.parent/(n + 1)⌋
32:        end if
33:    end while
34:    return a, b
35: end procedure
```

4.1.4 Ricostruzione dei sottoproblemi

Nel caso l'algoritmo venga eseguito con TREE_MODE a `true`, cioè nella variante divide&conquer, per ottenere le sottomatrici su cui effettuare nuovamente l'allineamento bisogna modificare leggermente l'Algoritmo 7 descritto nella Sezione 3.2. L'unica modifica necessaria riguarda la gestione delle coppie di nodi in cui uno in una riga di tipo e_t ed il relativo nodo genitore in una riga di tipo e_i. In questo caso, infatti, la coppia formata dai due nodi rappresenta un salto di riga e non una sottomatrice su cui è necessario effettuare nuovamente dei calcoli per ottenere un segmento di allineamento e va quindi escluda dalla lista dei sottoproblemi.

Poiché l'algoritmo mantiene in memoria i nodi che si trovano su righe di tipo e_i (Par. 4.1.2), e la procedura di ricostruzione esclude le alternative non facenti parte della soluzione dalla lista dei sottoproblemi, questi ultimi possono essere ricalcolati anche con algoritmi che non gestiscono le ED-strings ed operano solo su stringhe solide. Ciò consente di utilizzare, ad esempio, la variante lineare dell'algoritmo di Needleman-Wunsch (Sez. 2.5).

Algoritmo 9 Funzione per la ricostruzione dei sottoproblemi

```
1:  procedure RECOSTRUCT_SUBPROBLEMS(max_points_pos, tree, seq_s, seq_t, dependences)
2:      n ← len(seq_s)
3:      end_pos ← len(seq_s) * len(seq_t)
4:      if end_pos ≡ max_points_pos then
5:          cnode ← tree[end_pos]
6:      else
7:          cnode = TreeNode{pos : end_pos, parent : max_points_pos}
8:      end if
9:      pos ← cnode.pos
10:     parent ← cnode.parent
11:     couples ← empty_array
12:     while pos > 0 do
13:         couples.append(parent%(n + 1), ⌊parent/(n + 1)⌋)
14:         cnode ← tree[parent]
15:         pos ← cnode.pos
16:         parent ← cnode.parent
17:     end while
18:     couples.reverse()
19:     return couples.windows(2).filter((parent, node) => parent.row not in dependences.keys())
20: end procedure
```

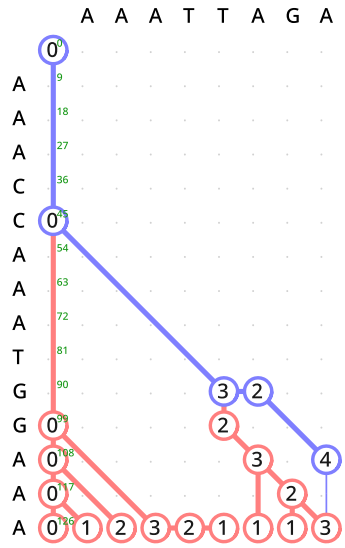


Figura 4.1: AAACCAAATGGAAA

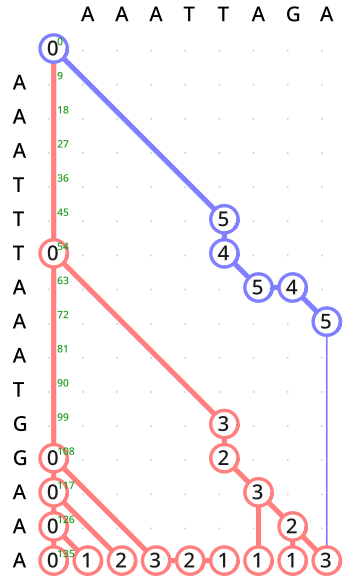
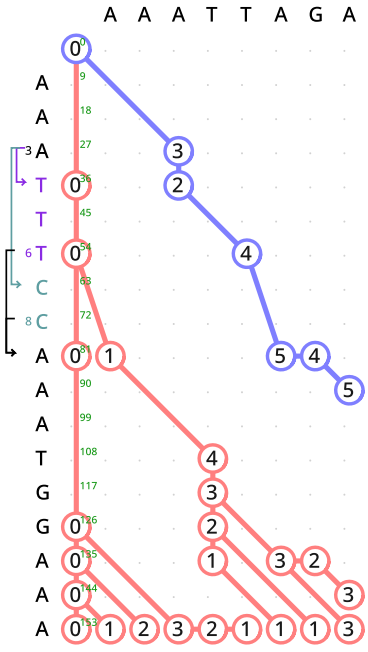


Figura 4.2: AAATTTAAATGGAAA



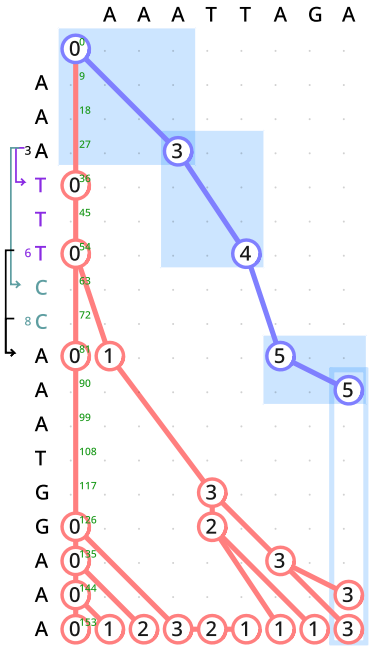


Figura 4.3: Calcolo effettuato sulla ED-string AAA[TTT|CC]AAATGGAAA, data dalle due varianti delle figure 4.1 e 4.2. A sinistra nella variante soluzione completa, a destra nella variante divide&conquer

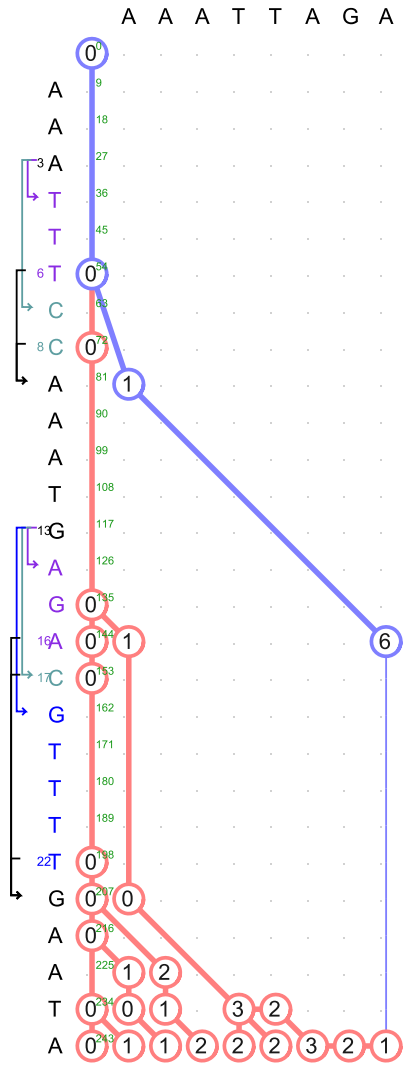


Figura 4.4: Esempio con varianti in più punti, anche di lunghezze molto differenti

4.2 Complessità di spazio

Nella valutazione della complessità di spazio si devono seguire gli stessi ragionamenti e calcoli visti nel paragrafo 3.1.5, tenendo conto di una importante differenza. Oltre ai branching nodes, l'algoritmo mantiene in memoria un nodo di passaggio situato nella riga di rientro da una serie di alternative. Questo nodo funge infatti da collegamento tra un branching node appartenente a una riga precedente e uno appartenente a una riga successiva rispetto alla riga considerata. Il numero di righe di tipo e_t è naturalmente ridotto, è presente una riga di questo tipo per ogni gruppo di alternative all'interno della de-string. Il numero di nodi che vengono

quindi memorizzati è legato al numero di percorsi che attraversano il gruppo di alternative ed è quindi potenzialmente pari al numero di elementi presenti su una riga, $m + 1$, moltiplicata per il numero di gruppi di alternative presenti. Tuttavia, siccome la maggior parte delle foglie dell'albero appartiene a rami che si uniscono nelle ultime righe che sono state calcolate, può essere verificato sperimentalmente che è improbabile che un elevato numero di nodi debba essere salvato a causa di questo. Nell'esempio in Figura 4.4, solo due nodi appartengono alla riga 23 e soltanto un nodo salvato appartiene alla riga 9.

4.3 Complessità di tempo

Anche nella valutazione della complessità di spazio si devono seguire gli stessi ragionamenti e calcoli visti nel paragrafo 3.1.6. Come visto, infatti, l'algoritmo modificato per le ED-strings è molto simile a quello descritto per le stringhe solide, le differenze introdotte non modificano l'ordine di grandezza del numero di calcoli da effettuare. La differenza più consistente tra le due versioni riguarda la trasformazione iniziale della ED-string in una stringa solida e la contestuale elaborazione delle dipendenze tra righe, operazioni di complessità $O(n)$ in quanto prodotto di un unico ciclo sui caratteri della rappresentazione della ED-string.

Capitolo 5

Conclusione

In questo lavoro, è stato descritto un nuovo approccio di memorizzazione dei dati per il calcolo dell'allineamento semiglobale, a partire dall'idea iniziale e seguendone l'evoluzione fino alla versione più completa e raffinata. Come visto, l'algoritmo può essere eseguito in due varianti: una che riduce meno l'uso della memoria ma non richiede calcoli aggiuntivi, e una che ottiene una complessità di spazio lineare al prezzo di dover ricalcolare alcune parti della matrice. La seguente tabella riepiloga i dati di occupazione di spazio per l'allineamento di 2.862 basi prese dal gene let-60 di *C. elegans* nelle diverse varianti dell'algoritmo di questa tesi e degli altri algoritmi in essa citati:

Versione algoritmo	Nodi	Ricalcolo necessario
matrice totale	8.196.769	no
tree_pruning (prima ottimizzazione, Par. 3.1.3)	78.589	no
tree_pruning (seconda ottimizzazione, Par. 3.1.4)	31.650	no
divide&conquer (Sez. 3.2)	4.583	parziale
rappresentazione alternativa 64bit (Sez. 3.3)	5.275	no
Algoritmo lineare (Sez. 2.5)	*11.442	no

(5.1)

Tabella 5.1: comparazione tra le diverse versioni dell'algoritmo

Eseguendo gli algoritmi, si vede che l'allineamento risultante, uno degli allineamenti ottimali possibili, è composto da 3368 colonne. Tra tutte le versioni riportate, la versione *divide&conquer* è l'unica a fornire una garanzia di complessità spaziale linearmente dipendente da m. Tuttavia, le versioni più ottimizzate dell'algoritmo si avvicinano a tale complessità di spazio nell'utilizzo pratico, pur mantenendo una complessità teorica quadratica.

Nonostante esistano algoritmi capaci di effettuare l'allineamento di stringhe solide in spazio lineare (come visto nel Paragrafo 2.5), il nuovo algoritmo riesce ad occupare uno spazio ad esso confrontabile senza la necessità di effettuare i calcoli aggiuntivi richiesti dall'algoritmo visto in 2.5. In particolare, rileggendo l'analisi relativa alla sua complessità di spazio e considerando due stringhe di 2.862 caratteri ciascuna, si può stimare un'occupazione di spazio di circa 11.442 elementi.

Inoltre, poiché la procedura di ricostruzione esclude le alternative non facenti parte della soluzione dalla lista dei sottoproblemi (come visto nel Paragrafo 4.1.2), tali sottoproblemi possono essere ricalcolati anche con algoritmi che operano solo su stringhe solide. Ciò consente di integrare il nuovo approccio con la variante lineare dell'algoritmo di Needleman-Wunsch, vista nella Sezione 2.5, ottenendo un processo complessivo di allineamento con complessità lineare.

Il punto più interessante, tuttavia, è come questo nuovo algoritmo sia in grado di effettuare l'allineamento tra una stringa solida ed una stringa degenerata elastica mantenendo lo stesso livello di complessità di spazio e di tempo di quanto visto per questo esempio su stringhe solide e, con ulteriori modifiche, tra due stringhe degenerate elastiche.

Nella letteratura consultata, non risultano esistere algoritmi per effettuare un allineamento tra una stringa solida ed una stringa degenerata con una complessità inferiore a $O(n \cdot m)$ anche nel caso migliore.

L'algoritmo ideato e spiegato in questa tesi presenta un buon grado di adattabilità alle esigenze di occupazione di spazio, tarando in fase di implementazione la complessità di gestione delle logiche di eliminazione dei nodi per raggiungere un compromesso tra semplicità di implementazione delle logiche, esigenze di spazio ed infine necessità di evitare o limitare un ricalcolo parziale.

L'implementazione completa dell'algoritmo, effettuata in linguaggio Rust, è disponibile pubblicamente al seguente indirizzo:

`https://github.com/vncnz/thesis_rust`

Per garantire l'individuazione della versione del codice utilizzata per questa tesi anche in caso di modifiche ed evoluzioni future del repository, verrà creato un `tag v1.0.0`.

Bibliografia

[1] M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Degenerate String Comparison and Applications. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics (WABI 2018)*, volume 113 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[2] G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Elastic-degenerate string matching via fast matrix multiplication, 2021.

[3] D. M. Church, V. A. Schneider, K. M. Steinberg, M. C. Schatz, A. R. Quinlan, C.-S. Chin, P. A. Kitts, B. Aken, G. T. Marth, M. M. Hoffman, J. Herrero, M. L. Z. Mendoza, R. Durbin, and P. Flicek. Extending reference assembly models. *Genome Biology*, 16(1):13, 2015.

[4] P. Cock, C. Fields, N. Goto, M. Heuer, and P. Rice. The sanger fastq format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38:1767–71, 12 2009.

[5] J. Daily. Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):81, 2016.

[6] DNASTAR, Inc. Choosing a pairwise alignment method — megalign pro guide. `https://www.dnastar.com/manuals/MegAlignPro/17.4/en/topic/choosing-a-pairwise-alignment-method`. Accessed 2025-09-20.

[7] R. Durbin, S. Eddy. A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[8] E. Gabory, M. N. Mwaniki, N. Pisanti, S. P. Pissis, J. Radoszewski, M. Sweering, and W. Zuba M. Pangenome comparison via ed strings. *Frontiers in bioinformatics*, 4, 2024.

[9] E. Gabory, M. N. Mwaniki, N. Pisanti, S. P. Pissis, J. Radoszewski, M. Sweering, and W. Zuba. Elastic-degenerate string comparison. arXiv preprint arXiv:2411.07782, 2024.

[10] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[11] M. Han and P. W. Sternberg. let-60, a gene that specifies cell fates during *C. elegans* vulval induction, encodes a ras protein. *Cell*, 63(5), 1990.

[12] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[13] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 06 2013.

[14] C. S. Iliopoulos, R. Kundu, and S. P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Information and Computation*, 279:104616, 2021. Selected Papers of the 11th International Conference on Language and Automata Theory and Applications, LATA 2017.

[15] N. C. Jones and P. A. Pevzner. *An introduction to bioinformatics algorithms*. MIT Press, 2004.

[16] R. Kundu. Eldes: Elastic-degenerate string comparison tool. `https://github.com/Ritu-Kundu/ElDeS`, 2024.

[17] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[18] Zs. Lipták. Slide del corso "fundamental algorithms for bioinformatics". Materiale didattico disponibile sulla pagina Moodle relativa al corso, 2021. Università degli Studi di Verona, A.A. 2020/2021 (e seguenti).

[19] Y. Liu, M. Koyutürk, S. Maxwell, M. Xiang, M. Veigl, R. S. Cooper, B. O. Tayo, Li Li, T. LaFramboise, Z. Wang, X. Zhu, and M. R. Chance. Discovery of common sequences absent in the human reference genome using pooled samples from next generation sequencing. *BMC Genomics*, 15(1):685, 2014.

[20] N. M. Mwaniki, E. Garrison, and N. Pisanti. Fast exact string to d-texts alignments. `https://arxiv.org/abs/2206.03242`, 2022. Preprint; contiene dettagli algoritmici utili e codice dsa disponibile su GitHub.

[21] NCBI Assembly. *C. elegans* strain n2 genome assembly (gca_000002985.3). `https://www.ncbi.nlm.nih.gov/assembly/GCA_000002985.3`, 2017.

[22] NCBI Assembly. *C. elegans* strain cb4856 genome assembly (gca_003402415.1). `https://www.ncbi.nlm.nih.gov/assembly/GCA_003402415.1`, 2019.

[23] Python Software Foundation. Python 3 standard library – numeric types: int. `https://docs.python.org/3/library/stdtypes.html#int-max-str-digits`, 2025.

[24] Rust Project. The Rust Programming Language. https://www.rust-lang.org/, 2025.

[25] C. D. Wunsch S. B. Needleman. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[26] K. Sahlin, T. Baudeau, B. Cazaux, and C. Marchet. A survey of mapping algorithms in the long-reads era. *Genome Biology*, 24(1):133, 2023.

[27] SeqAn developers. Pairwise alignment configuration (global, semi-global, local) — seqan3 documentation. https://docs.seqan.de/seqan3/3.0.3/group__alignment.html. Accessed 2025-09-20.

[28] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Computer Science Series. PWS Pub., 1997.

[29] H. Suzuki and M. Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19(S1):45, 2018.