



Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

# Costruzione Efficiente di Grafi di Splicing

**Relatore:** *Della Vedova Gianluca*

**Relazione della prova finale di:**

*Federico Bergamini*

*Matricola 845646*

**Anno Accademico 2020 - 2021**

## **Sommario**

Le nuove tecnologie di sequenziamento producono dati che hanno caratteristiche diverse da quelle che hanno dominato gli ultimi 10-15 anni. In particolare ad oggi vengono prodotte read molto più lunghe (da 10 a 500 volte più lunghe) ma con un tasso di errore più elevato. Ciò richiede di rivedere gli approcci sviluppati in precedenza. Questo stage affronta un importante problema in pangenomica computazionale: ottenere un grafo di variazioni a partire da un allineamento multiplo di long read. Più precisamente, lo stage studierà il problema nel caso ristretto di grafo ottenuto a partire da un insieme di trascritti (grafo di splicing).

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Perchè uno stage in Bioinformatica . . . . .	2
<b>2</b>	<b>Biologia Computazionale</b>	<b>3</b>
2.1	Obiettivi . . . . .	3
2.2	Bioinformatica . . . . .	4
2.3	Introduzione alla Biologia e Terminologia . . . . .	5
<b>3</b>	<b>Descrizione del Problema Affrontato</b>	<b>7</b>
3.1	Allineamento di Sequenze . . . . .	7
3.1.1	Algoritmi Utilizzati per performare un Allineamento . . . . .	9
3.2	Allineamento multiplo di sequenze . . . . .	12
3.3	Splicing Graph . . . . .	13
3.3.1	Definizione Formale ed Esempio . . . . .	13
3.4	Lavoro Richiesto . . . . .	14
<b>4</b>	<b>Soluzione Proposta</b>	<b>15</b>
4.1	Perchè il Linguaggio Rust . . . . .	15
4.2	Primo Approccio . . . . .	16
4.3	Metodologie di partizionamento . . . . .	17
4.3.1	Concetti Teorici Preliminari . . . . .	18
4.3.2	Soluzione di Programmazione Dinamica . . . . .	19
4.3.3	Soluzione Greedy . . . . .	20
4.4	Approccio Definitivo . . . . .	22
4.5	Tempi di Calcolo . . . . .	22

# Capitolo 1

## Introduzione

Questa tesi si prefigge l'obiettivo di illustrare ed argomentare l'esperienza di stage maturata durante il terzo anno di laurea triennale nella sua completezza. Verrà posta particolare enfasi sulle motivazioni che mi hanno portato a scegliere questo stage, le tecnologie utilizzate, i concetti teorici alla base, i problemi affrontati con le relative soluzioni e le nozioni da me apprese, oltre che un'introduzione ai concetti di biologia necessari al fine di comprendere al meglio questo elaborato.

### 1.1 Perchè uno stage in Bioinformatica

Al terzo anno della laurea in Scienze Informatiche dell'università di Milano Bicocca, è chiesto agli studenti di affrontare uno stage su argomenti da loro scelti, al fine di approfondire le conoscenze in materia e acquisire competenze professionalizzanti.

Avendo frequentato l'insegnamento facoltativo di Bioinformatica durante il primo semestre del terzo anno ho avuto la possibilità di appassionarmi alla materia. Nonostante all'inizio la trovassi astrusa, si è rivelata affascinante e mi ha fornito la conoscenza necessaria per portare a termine al meglio l'esperienza di tirocinio.

Ciò mi ha portato a scegliere uno stage difficoltoso, ma stimolante e molto formativo, che affronta un importante problema della pangenomica computazionale :

La costruzione, in linguaggio Rust, di uno splicing graph a partire da un allineamento multiplo di trascritti in input.

## Capitolo 2

# Biologia Computazionale

Con il termine *Biologia Computazionale* si intende la scienza che si prefigge l'obiettivo di sviluppare e applicare algoritmi, modelli e metodi matematici al fine di studiare sistemi biologici [6]. E' una scienza interdisciplinare che attinge a conoscenze di altri campi come matematica, biologia, statistica, biochimica al fine di perseguire i propri obiettivi [1].

Lo sviluppo della biologia computazionale iniziò negli anni 70 del secolo scorso, da allora una sempre più grande mole di dati e una modesta quantità di pubblicazioni in questo campo, hanno contribuito a rendere questa scienza quella che è oggi.

### 2.1 Obiettivi

Approcci studiati dalla biologia computazionale sono stati utilizzati per portare a termine vari compiti, tra cui il sequenziamento del genoma umano, creare modelli accurati del cervello umano e dei sistemi biologici.

Alcune sfide che si prefigge la biologia computazionale sono [6]:

- ANALISI STRUTTURA DELLE PROTEINE. Capire quali sono le proteine presenti in un campione biologico.
- RICERCA DI OMOLOGIE NELLE SEQUENZE GENOMICHE
- ALLINEAMENTO MULTIPLO DI SEQUENZE E RICOSTRUZIONE DI FILOGENESI. Analizzare la storia evolutiva di più specie.
- ANALISI DI SEQUENZE. Comparazione di sequenze genomiche di specie simili o distinte al fine di evidenziare omologie e differenze. Ricostru-

zione di lunghe sequenze genomiche a partire da frammenti. Correzione di errori da parte delle macchine sequenziatrici.

## 2.2 Bioinformatica

In particolare la *Bioinformatica* si focalizza sullo studio di tecniche ed algoritmi per l'analisi di stringhe (sequenze), applicando e studiando metodi tipici delle scienze informatiche, creando quindi interdisciplinarietà tra le due materie.

Alcuni esempi delle tecniche studiate in Bioinformatica sono:

- PATTERN RECOGNITION. Identificare una sottostringa all'interno di una più estesa.

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A A B A								A A B A							
A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
												A A B A			

Pattern Found at 0, 9 and 12

Figura 2.1: Esempio grafico di pattern matching

- DATA MINING
- MACHINE LEARNING ALGORITHMS

E' da notare che, in realtà, il termine *Bioinformatica* è usato per evidenziare una più ampia gamma di studi inerenti alla biologia che utilizzano approcci informatici per portare a termine i propri obiettivi.

## 2.3 Introduzione alla Biologia e Terminologia

Questa Relazione ha lo scopo di esporre il problema affrontato durante lo stage del terzo anno da un punto di vista informatico. Nonostante ciò verrà fatta un'introduzione ai concetti necessari di biologia molecolare, in modo da facilitare la comprensione di quanto segue.

In biologia Molecolare, la *Trascrizione* è un processo mediante il quale le informazioni contenute nel Dna vengo trascritte su una molecola di Rna. Un *gene* è una regione del genoma (detta *locus*) che esprime una proteina e viene identificato attraverso il suo *Hugo name* che è un acronimo della sua descrizione.

Si può pensare ad un gene come un'alternanza di regioni codificanti (esoni) e regioni non codificanti (introni).

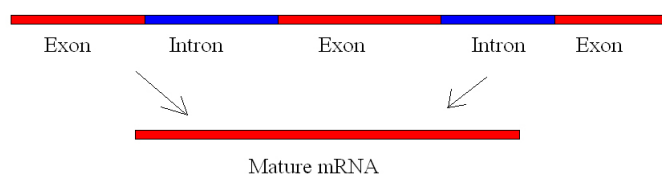


Figura 2.2: Alternanza di introni ed Esoni

Il confine tra esone ed introne è detto *Splicing site al 5'*, mentre il confine tra introne ed esone è detto *Splicing site al 3'*.

Quando avviene una *Trascrizione*, il locus del gene viene copiato, compiendo una sostituzione tra le molecole di *Timina* e *Uracile*. Dal punto di vista informatico, il locus è una stringa su alfabeto **A, C, G, T**, dopo la trascrizione, l'alfabeto diventa **A, C, G, U** e la molecola formata prende il nome di *pre - mRNA*. Successivamente vengono eliminate le regioni non codificanti e viene formata la molecola di *mRNA* (trascritto).

I geni umani sono circa 25000, invece le proteine sono centinaia di migliaia. La corrispondenza 1:1 non è rispettata in quanto un gene può ricombinare i suoi esoni in modi diversi al fine di esprimere una molteplicità di trascritti (quindi proteine). Tutti i trascritti del gene vengono dette *Isoforme di Splicing* del gene.

Studi recenti asseriscono che lo splicing alternativo avvenga sul 90% dei geni di Homo Sapiens.

Uno dei motivi per cui è utile studiare le isoforme di splicing di un gene è quello che sono fortemente legate alle malattie.

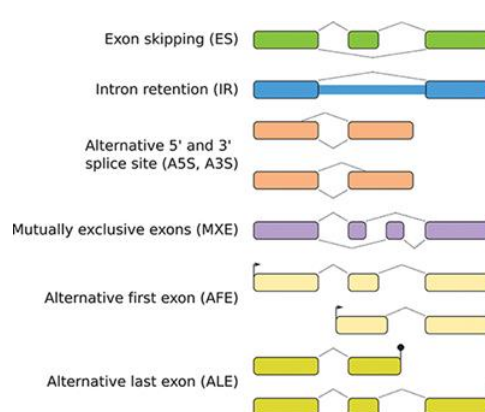


Figura 2.3: Esempi di Isoforme di Splining

Patten frequenti dell'alternative splicing sono Exon skipping, Intron Retention, 5' Competing Site, 3' Competing Site, Multiple promoters.



## Capitolo 3

# Descrizione del Problema Affrontato

La sfida a cui ho accettato di sottopormi, consiste in un importante problema di pangenomica computazionale, ovvero quello di costruire, in linguaggio Rust, uno splicing graph a partire da un allineamento multiplo di trascritti di un gene fornito in input.

Prima di esporre la soluzione al problema saranno necessari alcuni concetti preliminari, quali quelli di *Allineamento Multiplo* e *Splicing Graph*. Nozioni come *Hugo Name* o *Trascritto* sono già stati forniti nel capitolo 2.

### 3.1 Allineamento di Sequenze

In generale, un allineamento di sequenze è una procedura bioinformatica con cui vengono messe a confronto ed allineate due o più sequenze primarie di aminoacidi, DNA o RNA. L'allineamento permette di individuare regioni identiche o simili che possono avere relazioni funzionali, strutturali o filogenetiche (evolutive). Spesso l'allineamento viene utilizzato per verificare se una sequenza di interesse sia presente all'interno di un database di sequenze conosciute oppure se ne esista una simile.

Nel caso ristretto dell'allineamento tra due sequenze, quest'ultimo estende il concetto di *Distanza di Hamming* a stringhe di lunghezza diversa.

La Distanza di Hamming è definita, solo su stringhe di uguale lunghezza, come il numero di posizioni nelle quali i simboli corrispondenti in due stringhe  $s_1$  ed  $s_2$  divergono.

In simboli, per distanza, si intende una funzione  $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}_+$ , dove  $\mathbb{S}$  è l'insieme delle stringhe, che abbia tre proprietà :

- RIFLESSIVITÀ:  $d(x, y) = 0 \iff x = y \forall x, y \in \mathbb{S}$
- SIMMETRIA:  $d(x, y) = d(y, x) \forall x, y \in \mathbb{S}$
- DISUGUAGLIANZA TRIANGOLARE:  $d(x, y) + d(y, z) \geq d(x, z), \forall x, y, z \in \mathbb{S}$

Per fare in modo di poter allineare due stringhe con lunghezze diverse, vengono introdotti gli *Indel* (-), che hanno la funzione di "Riempire i buchi". Ci sono due restrizioni nell'inserimento degli indel:

- Non possono esistere colonne di soli indel
- Le stringhe estese di Indel devono avere la stessa lunghezza

**Example 1.**

$s_1 = ABRACADABRA, s_2 = BANANA$

Alcuni possibili allineamenti sono:

A	B	R	A	C	A	D	A	B	R	A
-	B	-	A	N	A	-	-	-	N	A
A	B	R	A	C	A	D	A	B	R	A
-	-	-	B	-	A	N	A	-	N	A
A	B	R	A	C	A	D	A	B	R	A
-	B	A	N	A	-	-	-	-	N	A

Ciò non significa che siano tutti ugualmente *buoni*

Per saggiare la bontà di un allineamento, è necessario formalizzarlo come un problema di ottimizzazione, quindi definendo:

- INSIEME DELLE STANZE: Un elemento di  $\mathbb{S} \times \mathbb{S}$
- INSIEME DELLE SOLUZIONI AMMISSIBILI: gli allineamenti che rispettano le condizioni elencate precedentemente
- FUNZIONE OBIETTIVO:  $\sum_{i=1}^l score(s_1[i], s_2[i])$
- SOLUZIONE: Allineamento che massimizza l'obiettivo, quindi la funzione obiettivo

Dove  $score(s_1[i], s_2[j])$  è il valore dell'incolonnamento della colonna presa in esame. Questo valore può essere prelevato da opportune *Matrici di Score* che contengono una funzione  $score(c_1, c_2)$  per ogni carattere dell'alfabeto in



$$M[i, j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) & \text{nessun indel} \\ M[i, j-1] + d(-, s_2[j]) & \text{indel solo in } s_1 \\ M[i-1, j] + d(s_1[i], -) & \text{indel solo in } s_2 \end{cases} \quad (3.1)$$

dove  $d(s_1[i], s_2[j])$  è il punteggio ottenuto dalla matrice di score. Si ricorda che non è contemplato il caso di soli indel in colonna.

Le condizioni al contorno sono:

- $M[0, 0] = 0$
- $M[i, 0] = M[i - 1, 0] + d(s_1[i], -)$
- $M[0, j] = M[0, j - 1] + d(-, s_2[j])$

Il tempo di esecuzione è  $\mathcal{O}(nm)$  dove  $|s_1| = n$  e  $|s_2| = m$ .

L'allineamento può essere ottenuto tramite l'usuale *TraceBack* della matrice di programmazione dinamica, partendo da  $M[n, m]$  retrocedendo fino a  $M[0, 0]$ .

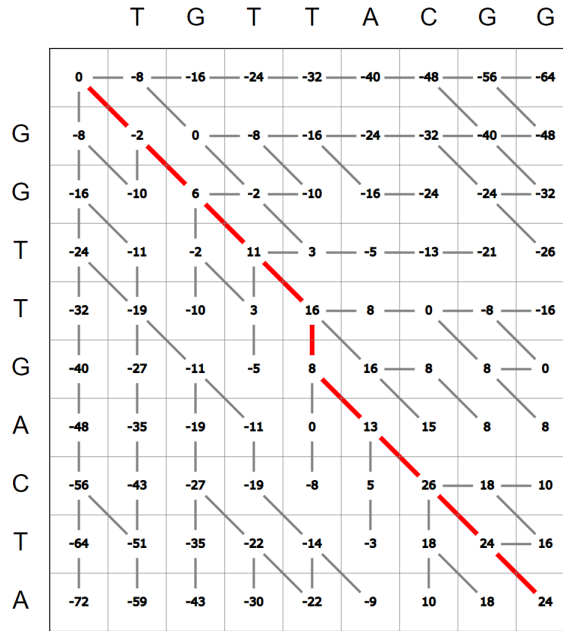


Figura 3.2: Esempio di esecuzione dell'algoritmo di Needleman-Wunsch su istanza :  $s_1 = TGTTACGG$  e  $s_2 = GGTGACTA$

## Algoritmo di Smith-Waterman

L'algoritmo di Smith-Waterman, come quello di Needleman-Wunsch, utilizza un approccio di programmazione dinamica, ma a differenza di quest'ultimo, evidenzia le sottostringhe di  $s_1$  ed  $s_2$  che massimizzano il punteggio di allineamento. In altre parole cerca di trovare  $t_1 = s_1[h : i]$  e  $t_2 = s_2[k : j]$  tale che il punteggio di allineamento di  $t_1$  e  $t_2$  sia massimo.

Per raggiungere questo obiettivo pone un limite inferiore al punteggio di allineamento, che non può essere negativo, aggiungendo un nuovo caso all'equazione di ricorrenza dell'algoritmo di Needleman-Wunsch, che si modifica nel seguente modo:

$$M[i, j] = \text{allineamento ottimo su } s_1[:i], s_2[:j]$$

$$M[i, j] = \max \begin{cases} 0 & \\ M[i-1, j-1] + d(s_1[i], s_2[j]) & \text{nessun indel} \\ M[i, j-1] + d(-, s_2[j]) & \text{indel solo in } s_1 \\ M[i-1, j] + d(s_1[i], -) & \text{indel solo in } s_2 \end{cases} \quad (3.2)$$

dove  $d(s_1[i], s_2[j])$ , come in Needleman-Wunsch, è il punteggio ottenuto dalla matrice di score.

Le condizioni al contorno sono differenti da quelle dell'algoritmo di Needleman-Wunsch, in quanto un allineamento non può avere punteggio negativo, e si modificano come segue:

- $M[0, 0] = 0$
- $M[i, 0] = 0 \ \forall i \mid 1 \leq i \leq n$
- $M[0, j] = 0 \ \forall j \mid 1 \leq j \leq m$

Il tempo di calcolo, come in Needleman-Wunsch, è  $\mathcal{O}(nm)$  dove  $|s_1| = n$  e  $|s_2| = m$ .

Il Traceback diventa però più complicato, in quanto deve essere effettuato dal valore massimo della matrice di programmazione dinamica, retrocedendo fino al primo 0.

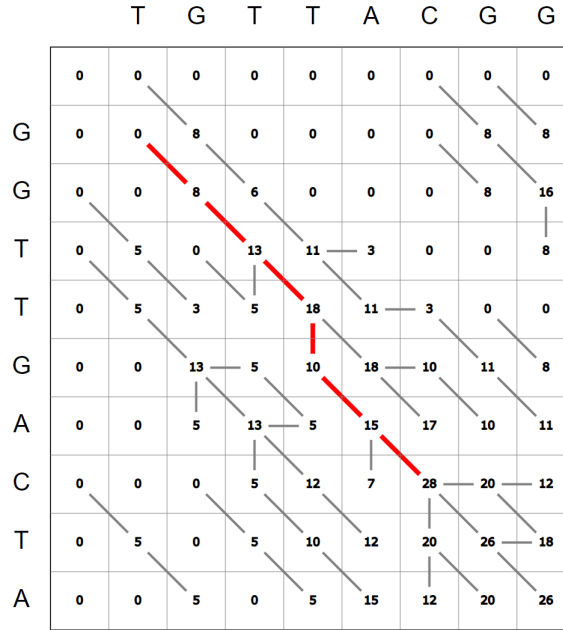


Figura 3.3: Esempio di esecuzione dell'algoritmo di Smith-Waterman sulla stessa istanza della figura 3.2

### 3.2 Allineamento multiplo di sequenze

L'allineamento multiplo è un'estensione del problema dell'allineamento di due sequenze, infatti, l'istanza del problema non sono più due sole stringhe, ma un insieme di  $k$  stringhe  $S = \{s_1, s_2, s_3, \dots, s_k\}$ .

Ciò però introduce alcuni problemi, ad esempio come posizionare gli indel nelle colonne dell'allineamento. Sicuramente ci potranno essere più indel in una singola colonna, ma non solamente indel, infatti è possibile accettarne fino a  $k - 1$ .

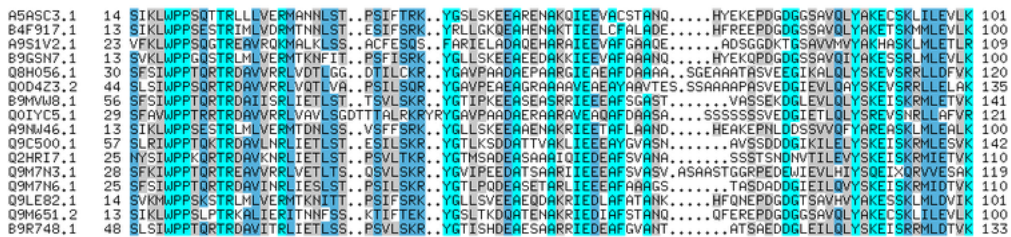


Figura 3.4: Esempio di allineamento multiplo tra sequenze

Un altro problema è la funzione codificata nella matrice di score. Infatti essa è una funzione definita  $score : \Sigma \times \Sigma \rightarrow \mathbb{R}_+$ , dove  $\Sigma$  è l'alfabeto dei simboli delle stringhe, perciò non può restituire un valore per un numero arbitrario di stringhe.

Questo problema è risolto utilizzando la funzione *Sum of Pairs* che somma il punteggio di tutte le righe, prese a coppie, in una colonna di allineamento e ne fa la somma.

Una possibile soluzione al problema dell'allineamento multiplo, è un algoritmo di programmazione dinamica. Questa volta però, a differenza dei due precedenti, l'ultima componente dell'allineamento può trovarsi in  $2^k - 1$  casi (non è possibile il caso di soli indel).

Il tempo di calcolo è  $\mathcal{O}(n^k)$  per un numero contenuto di stringhe, altrimenti è *NP-Completo*.

### 3.3 Splicing Graph

Come accennato nel capitolo riguardante la biologia molecolare, lo splicing alternativo è un fenomeno frequentemente osservato che sintetizza vari trascritti dello stesso gene. Queste varianti possono essere molte e può risultare difficoltoso, specie per i geni più grandi, descrivere le varie isoforme di splicing, con una struttura formale e conveniente che evidenzi le omologie e differenze dei trascritti in input.

Uno splicing graph cerca di assolvere a questo compito, accorpendo, dove possibile, le parti comuni delle sequenze genomiche, dando una descrizione chiara e precisa dei trascritti in input.

#### 3.3.1 Definizione Formale ed Esempio

Uno Splicing Graph è un Grafo Diretto Aciclico (**DAG**) tale che [4] [2]:

- I VERICI rappresentano i siti di splicing per un gene dato. Sono numerati da 1 a n con un identificatore univoco ed etichettati da una sottostringa dei trascritti in input.
- Gli ARCHI rappresentano gli esoni e gli introni tra i siti di splicing. Seguono un ordine crescente degli identificatori univoci dei nodi.

Sono aggiunti due nodi "Artificiali", etichettati con "first\_node" e "last\_node", che hanno come identificatori univoci rispettivamente 1 e n, che segnano i punti di inizio e di fine nella lettura dei trascritti.

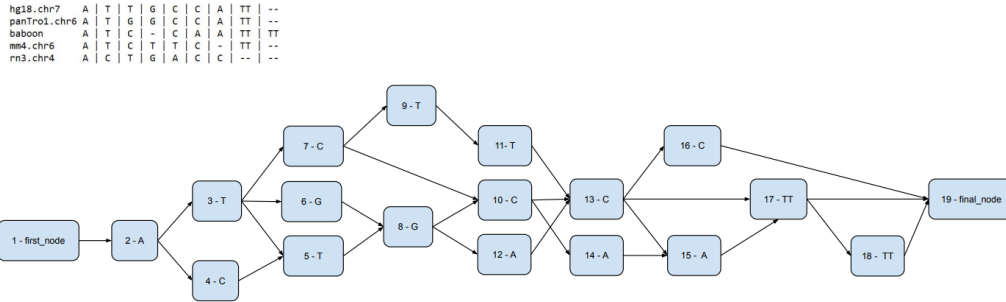


Figura 3.5: Esempio di Splicing graph, creato dal mio programma, con l'allineamento multiplo di riferimento

Non tutti i percorsi  $First\_node \rightarrow Last\_node$  corrispondono agli effettivi trascritti del gene. Sarebbe necessario limitare il numero di questi falsi trascritti.

### 3.4 Lavoro Richiesto

Mi è stato richiesto di implementare una libreria che permetta di creare un grafo di splicing.

In input è fornito un allineamento multiplo dei trascritti per fare in modo di evidenziare le omologie tra le isoforme di splicing al fine di rendere più agevole la creazione del grafo. Sono state utilizzate due librerie esterne che permettono costruire un *Variation Graph* (una generalizzazione dello Splicing Graph) ed eseguire il parsing di file FASTA, uno dei formati standard in bioinformatica, per ottenere l'allineamento multiplo.

Il parsing di file in formato MAF (Multiple Alignment Format), un altro formato standard in bioinformatica, in assenza di un'alternativa migliore, è stato da me implementato a partire da una libreria incompleta che lo eseguiva parzialmente.

Nel prossimo capitolo saranno dati i dettagli della soluzione proposta.



## Capitolo 4

# Soluzione Proposta

Questo capitolo ha lo scopo di illustrare la soluzione proposta, i passaggi intermedi ed i ragionamenti logici che mi hanno portato ad elaborare la soluzione finale.

Sarà anzitutto chiarito il perchè del linguaggio *Rust*, definendo le sue particolarità ed i suoi punti critici, poi saranno illustrate le pubblicazioni utilizzate per migliorare le soluzioni iniziali, infine verrà eseguita un'analisi dei tempi di calcolo della soluzione definitiva.

### 4.1 Perchè il Linguaggio Rust

Il linguaggio Rust è un linguaggio di programmazione sviluppato da Mozilla insieme alla comunità open source per la programmazione di sistemi.

Rust persegue obiettivi di efficienza e sicurezza ed è idoneo allo sviluppo di software concorrente. Sintatticamente simile a *C++*, si differenzia da quest'ultimo a causa del suo meccanismo di salvaguardia della memoria chiamato *Borrow Checker*, che si avvale del design pattern RAII (Resource Acquisition Is Initialization) per deallocare le risorse nel caso il loro *Owner* esca di scope [3].

Rust non permette l'utilizzo di *null pointer* e *Dangling Reference* che devono essere, in alcuni casi, esplicitamente validati tramite *lifetime*, ciò lo rende un linguaggio sicuro, infatti è stato utilizzato per scrivere alcuni moduli del browser mozilla e parte di alcuni sistemi operativi, tra cui Android.

Rust è usato in bioinformatica, ma in generale in ambito scientifico, perchè oltre alle caratteristiche sopra elencate, è un linguaggio con una buona



Figura 4.1: Logo del Linguaggio Rust affiancato a quello di Mozilla

espressività, simile a quella di un linguaggio a più alto livello, ma più veloce. Ciò lo rende adeguato allo sviluppo di applicativi che necessitano di una buona velocità d'esecuzione e per utenti che non vogliono perdere tempo con gli errori tipici, ad esempio, del C++. Una pecca di rust è che, in generale, è difficoltoso da imparare [5].

## 4.2 Primo Approccio

Prima di poter lavorare sul problema effettivo, c'è stato un preambolo di circa tre settimane in cui è avvenuta una grossa fase di apprendimento. Dapprima ho imparato il linguaggio Rust, poi ho appreso i costrutti resi disponibili dalle varie librerie per adempiere ai miei compiti. In questa fase ho anche creato il parser di file in formato MAF, che è stato un buon esercizio per apprendere ancora meglio Rust.

Terminato questo preambolo, l'approccio che ho deciso di utilizzare per la costruzione dello splicing graph, è stato quello di scansionare l'allineamento colonna per colonna e costruire il grafo di conseguenza.

In pratica, si assiste ad una fase di inizializzazione, in cui viene creato il nodo 1 con etichetta *first\_node* e aggiunto al *Path* di ogni trascritto nell'allineamento.

Il costrutto *Path* è utile in quanto permette di memorizzare i cammini associati ai vari trascritti in termini di sequenza di nodi, quindi di poter associare ad ogni trascritto un percorso attraverso i vertici che corrisponde alla sua sequenza genomica.

hg18.chr7	A	T	T	G	C	C	A	TT	--
panTro1.chr6	A	T	G	G	C	C	A	TT	--
baboon	A	T	C	-	C	A	A	TT	TT
mm4.chr6	A	T	C	T	T	C	-	TT	--
rn3.chr4	A	C	T	G	A	C	C	--	--

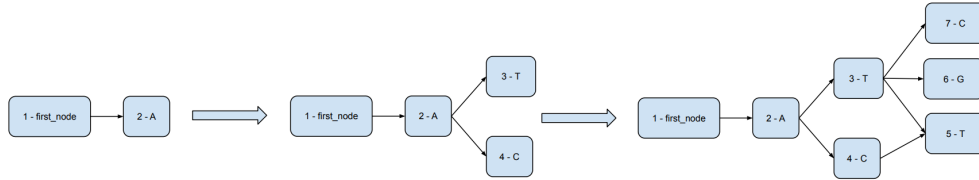


Figura 4.2: Le prime tre fasi della costruzione dello Splicing Graph. Vengono prese in considerazione le prime colonne dell'allineamento

In seguito, per ogni colonna dell'allineamento, si costruisce l'insieme dei simboli contenuti, scartando eventuali indel. Per ogni simbolo si costruisce un nodo la cui etichetta è costituita da simbolo stesso e lo si "Aggancia" correttamente al nodo precedente. Dopo aver portato a termine questa operazione, si aggiunge ogni nodo al o ai Path corretti (vedi figura 4.2).

Infine si assiste ad una fase di epilogo in cui tutti i nodi formati nell'ultima iterazione della fase precedente vengono agganciati ad un ultimo nodo etichettato *last\_node* che corrisponde al nodo di fine e unico nodo foglia del grafo.

Come si può intuire, questo metodo di creazione dello splicing graph, porta alla creazione di un grafo con un'eccessivo numero di nodi ed un eccessivo numero di percorsi *first\_node*  $\rightarrow$  *last\_node*. Ricordo che, dove possibile, è necessario limitare il numero di questi percorsi.

Una soluzione potrebbe essere quella di seguire lo stesso approccio per la costruzione del grafo, ma invece di considerare l'allineamento colonna per colonna, procedere con un partizionamento opportuno per fare sì di creare dei nodi con etichetta di lunghezza maggiore di 1, facendo sì di diminuire i nodi già presenti.

La prossima sezione spiegherà le metodologie utilizzate per l'ottenimento di un partizionamento opportuno.

## 4.3 Metodologie di partizionamento

Un buon partizionamento, dovrebbe fare in modo di accorpare il più possibile le colonne dell'allineamento, così da diminuire le quantità di nodi e di percor-

si  $first\_node \rightarrow last\_node$ . Per assolvere a questo compito, sono utilizzati due approcci, uno di *programmazione Dinamica* e uno *Greedy*, che originariamente sono stati pensati per risolvere il *Founder Sequence Reconstruction Problem*. Entrambi gli approcci sono descritti in [7].

### 4.3.1 Concetti Teorici Preliminari

Ancora una volta saranno presentati alcuni concetti teorici necessari alla comprensione degli metodi utilizzati per la risoluzione del problema.

Sia  $\mathbb{C} = \{C_1, C_2, \dots, C_m\} \subseteq \Sigma^n$  un insieme di sequenze di lunghezza  $n$  su un alfabeto  $\Sigma$  detti *Ricombinanti*, ogni ricombinante è scritto come  $C_i = c_{i1}, c_{i1}, \dots, c_{in}$ . Un insieme  $\mathbb{F} = \{F_1, F_2, \dots, F_k\} \subseteq (\Sigma \cup \{-\})^n$  è chiamato *Insieme dei Fondatori* per l'insieme  $\mathbb{C}$  e ogni  $F_i$  è detta sequenza fondatrice se  $\mathbb{C}$  ha un *Parsing* in termini di  $\mathbb{F}$ .

Ciò significa che ogni  $C_i \in \mathbb{C}$  può essere decomposto in una serie di *Frammenti* non vuoti  $f_{ih}$  tale che  $C_i = f_{i1}f_{i2}\dots f_{ip_i}$  e ogni  $f_{ih}$  occorre un qualche  $F \in \mathbb{F}$  nella stessa posizione in cui è presente anche in  $C_i$ .

In altre parole, si può riscrivere  $F = \alpha f_{ih} \beta$  dove la lunghezza di  $\alpha$  è uguale a quella di  $|\alpha| = |f_{i1}\dots f_{ih-1}|$ , mentre la lunghezza di  $\beta$  è uguale a quella di  $|\beta| = |f_{ih+1}\dots f_{ip_i}|$ .

Si assume che il parsing sia *ridotto* nel senso che due successivi frammenti di  $f_{ih}$  e  $f_{ih+1}$  siano sempre estratti da elementi diversi di  $\mathbb{F}$ .

In un parsing (ridotto) di  $\mathbb{C}$ , si dice che un ricombinante  $C_i$  ha un *Punto di crossover* in  $j$  se il parse di  $C_i$  è  $C_i = f_{i1}\dots f_{jh}f_{jh+1}\dots f_{ip_i}$  e  $|f_{i1}\dots f_{jh}| = j - 1$ .

#### Example 1.

L'insieme dei ricombinanti:

0	0	1	0	0	0	0	1	0	0	1	1
1	1	1	1	1	1	1	0	0	1	1	0
0	0	1	0	1	1	1	1	0	1	1	0
1	1	1	1	0	0	1	0	0	0	1	1

L'insieme dei fondatori:

0	0	1	0	1	1	1	0	0	0	1	1
1	1	1	1	0	0	0	1	0	1	1	0

A questo punto, il nostro problema è quello di trovare il set di fondatori  $\mathbb{F}$  di cardinalità minima, nel contempo massimizzare la lunghezza del frammento più corto  $\lambda_{min}$  e la lunghezza media dei frammenti  $\lambda_{ave}$ . Come è intuitivo pensare, questi sono due obiettivi contraddittori.

Ciò ci porta alla formalizzazione di due problemi:

- Il primo è il MINIMUM FOUNDER SET PROBLEM che consiste nel trovare il set di fondatori  $\mathbb{F}$  di cardinalità minima tale che il parse di  $\mathbb{F}$  abbia  $\lambda_{min} > L$  oppure  $\lambda_{ave} > L$  per un  $L$  dato chiamato *Threshold*.
- Il secondo è il MAXIMUM FRAGMENT LENGTH PROBLEM che consiste nel massimizzare  $\lambda_{min}$  e/o  $\lambda_{ave}$  sotto la condizione che la cardinalità di  $\mathbb{F} \leq M$  per un  $M$  dato.

Al fine di costruire uno Splicing Graph, è sembrato più congeniale, modellare il problema a partire dal MINIMUM FOUNDER SET PROBLEM, in quanto si è voluta tenere più bassa possibile la cardinalità di  $\mathbb{F}$ . Ciò ha porta ad un buon grafo, a cui possono essere fatte ulteriori migliorie.

### 4.3.2 Soluzione di Programmazione Dinamica

Un algoritmo risolutivo del minimum founder set problem è basato sull'idea di segmentare l'insieme dei ricombinanti  $\mathbb{C}$  in segmenti disgiunti, in modo da partizionare interamente  $\mathbb{C}$ . Un segmento  $C[j, k]$  è definito come l'insieme di tutte le sottosequenze  $c_{ij} \dots c_{ik}$  dell'insieme dei Ricombinanti.

Il nostro obiettivo diventa quindi quello di dare un parsing di  $\mathbb{C}$  con  $\lambda_{min} > L$  per un  $L$  dato, tale che la cardinalità massima dei segmenti sia minimizzata, ciò minimizza anche la cardinalità dell'insieme  $\mathbb{F}$  dato che  $|\mathbb{F}| = |\max\{C[j, k]\}|$

Si vuole quindi computare:

$$M(n) = \min_{s \in S_L} \{|C[j, k]| : |C[j, k]| \in s\}$$

Dove  $s$  è l'insieme dei partizionamenti possibili su  $\mathbb{C}$ .

Quindi  $M(n)$  può essere ottenuto valutando  $M(L), M(L + 1) \dots M(n)$ , nel seguente modo:

Caso base:

$$M(L) = |C[1, L]|$$

Passo Ricorsivo:

$$M(j) = \min_{h \leq j-L} \max_{j=L+1, \dots, n} \{M(h), |C[h+1, j]|\}$$

Il tempo di calcolo per questo approccio è  $\mathcal{O}(n^2m)$  dove  $n$  è la lunghezza dell'allineamento e  $m$  è il numero delle sequenze allineate.

Con l'usuale traceback dell'array di programmazione dinamica, è possibile giungere al partizionamento corretto.

Questa soluzione non tiene però conto degli indel presenti nelle sequenze.

E' opportuno o meno contare sottosequenze che presentano solo indel nella cardinalità dei segmenti? o ancora; è opportuno contare le sottosequenze formate solo parzialmente da indel?

Come conseguenza della soluzione greedy presentata nella prossima sezione, sono arrivato alla conclusione che non ha senso il conteggio di una sottosequenza di soli indel nella cardinalità di un segmento. Invece ha senso il conteggio di una sottosequenza formata solo parzialmente da indel.

In generale questa soluzione di programmazione dinamica non restituisce risultati ottimali.

In particolare con allineamenti di sequenze troppo diverse tra di loro, si assiste ad un partizionamento praticamente inesistente, nel senso che, in un caso limite, si verrà a formare un grafo con  $n+2$  vertici, con etichette corrispondenti alle intere sequenze in input, che corrispondono alle  $n$  sequenze in aggiunta al nodo etichettato *first\_node* e a quello etichettato *last\_node*.

La soluzione greedy cerca di ovviare a questo problema.

### 4.3.3 Soluzione Greedy

La soluzione greedy, anch'essa descritta in [7], ha come obiettivo quello di trovare il partizionamento con il minor numero di segmenti tale che ogni segmento abbia cardinalità massima  $M$ .

Tale obiettivo è raggiunto considerando inizialmente  $|C[1, k]| \leq M$  poi progressivamente  $|C[k+1, k']| \leq M$  fino al termine dell'allineamento, tale che i partizionamenti siano più lunghi possibile.

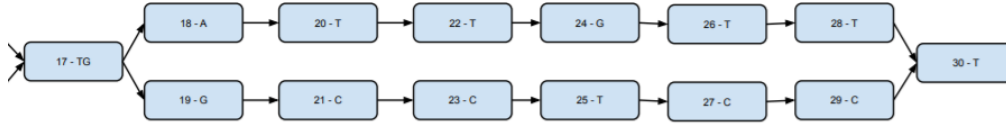


Figura 4.3: Primo Esempio di soluzione greedy. Per comodità, è riportato solo un frammento del grafo

Questa soluzione è però fin troppo semplicistica, infatti non contempla il fatto che un segmento di cardinalità 1 potrebbe già superare la soglia  $M$ .

Che fare in questo caso?

Si potrebbe pensare di scegliere  $M$  opportunamente, in modo da non permettere a questo caso di presentarsi. Dovrebbe essere sufficiente inserire un valore di  $M \geq \min\{|\Sigma|, n\}$  dove  $n$  è il numero delle sequenze e  $|\Sigma|$  è la cardinalità dell'alfabeto su cui sono costruite le stringhe.

Questo tipo di soluzione porterebbe però, in alcuni casi, a segmenti troppo lunghi, quindi a non accorpare al meglio l'allineamento.

Al fine di poter mantenere un valore di  $M > 0$ , inizialmente ho pensato di interrompere il segmento nel caso  $|C[k, k + 1]| > M$ . Però ciò portava ad un grafo come quello mostrato in figura 4.3.

Logicamente si vorrebbero accorpare i nodi 18, 20, 22, 24, 26, 28 in un unico nodo, così come i vertici 19, 21, 23, 25, 27, 29.

Ciò è possibile se si permette all'algoritmo di partizionamento, in certi casi, di non attenersi al vincolo di cardinalità dei segmenti. I casi in questione sono proprio quelli menzionati in precedenza, infatti se il segmento di cardinalità minima già eccede  $M$ , allora gli è permesso accorpare più colonne dell'allineamento fino a quando non aumenta ulteriormente di cardinalità, o quando non è possibile formare un nuovo frammento di cardinalità minore. Allora si inizia un nuovo frammento.

Nel conteggio di cardinalità dei segmenti sono omesse le sottosequenze di sonli indel, esattamente come nell'algoritmo di programmazione dinamica menzionato nella sezione precedente.

Questo "Nuovo" algoritmo greedy permette di formare un grafo come quello in figura 4.4.

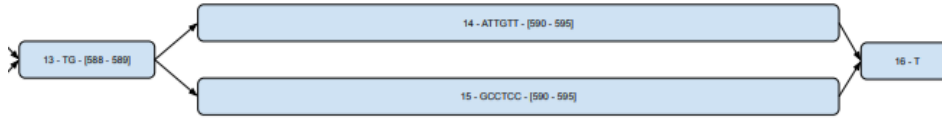


Figura 4.4: Esempio definitivo di soluzione greedy, Per comodità, è riportato solo un frammento del grafo e sono riportati gli indici dell'allineamento corrispondenti alle etichette dei nodi

## 4.4 Approccio Definitivo

Tra il primo approccio e l'approccio definitivo si sono susseguiti numerosi tentativi. Per chiarezza espositiva, verranno esposti solo il primo e il definitivo, in modo da non essere troppo prolissi nell'esposizione.

Avendo un partizionamento adeguato, la fase di costruzione del grafo risulta molto simile a quella del primo approccio.

La fase di inizializzazione è identica a quella precedentemente descritta, come la fase di terminazione, mentre quella di costruzione vera e propria deve tenere conto che non si ragiona più in termini di simboli, ma in termini di sottostringhe dell'allineamento. A parte questo, il grafo è costruito incrementalmente come mostrato nella figura 4.2.

Il partizionamento è ottenuto tramite la segmentazione greedy. in ogni caso l'utente può utilizzare l'algoritmo di programmazione dinamica.

Il valore di soglia  $M$  è impostabile dall'utente, ma è consigliato tenerlo basso per fare in modo di costruire un grafo migliore. Ad esempio, nella figura 4.4, è stato scelto il valore  $M = 1$ .

## 4.5 Tempi di Calcolo

Fare un'analisi formale dei tempi di calcolo risulterebbe difficile. La fase di preprocessamento delle sequenze, in cui viene estratto il partizionamento, come detto in precedenza, ha tempo  $\mathcal{O}(nm)$ .

La vera e propria fase di costruzione prende un tempo che è proporzionale al numero dei segmenti nella partizione creata, alla cardinalità dei segmenti e al numero delle sequenze allineate.



# Bibliografia

- [1] Center for Computational Molecular Biology. *Center for Computational Molecular Biology*. URL: <https://ccmb.brown.edu/home> (visitato il 20/05/2021).
- [2] Alekseyev M Heber S. «Splicing graphs and EST assembly problem. Bioinformatics.» In: (2002). DOI: 10.1093/bioinformatics/18.suppl\_1.s181.
- [3] Steve Klabnik e Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.
- [4] Sammeth M. «Complete alternative splicing events are bubbles in splicing graphs.» In: (2009). DOI: 10.1089/cmb.2009.0108.
- [5] Jeffrey M. Perkel. *Why scientists are turning to Rust*. URL: <https://www.nature.com/articles/d41586-020-03382-2> (visitato il 02/12/2020).
- [6] David B. Searls. «"Chapter 1 - Grand challenges in computational biology"». In: *Computational Methods in Molecular Biology*. A cura di David B. Searls Steven L. Salzberg e Simon Kasif. Vol. 32. New Comprehensive Biochemistry. Elsevier, 1998, pp. 3–10. DOI: [https://doi.org/10.1016/S0167-7306\(08\)60458-5](https://doi.org/10.1016/S0167-7306(08)60458-5). URL: <http://www.sciencedirect.com/science/article/pii/S0167730608604585>.
- [7] Esko Ukkonen. «Finding Founder Sequences from a Set of Recombinants». In: (2002).