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*
Correlatore: *Rizzi Raffaella*

Relazione della prova finale di:
Federico Bergamini
Matricola 845646

Anno Accademico 2020 - 2021

Ringraziamenti

Al Prof. Della Vedova e alla professoressa Rizzi, per la fiducia riposta, la pazienza e l'aiuto fornitomi prima, dopo e durante l'esperienza di stage.
Alla mia famiglia, per avermi supportato (e sopportato) moralmente, psicologicamente ed economicamente in questi tre anni.
A tutti gli amici e colleghi, con i quali ho codiviso il mio percorso all'università, per averlo reso unico e speciale.
A chiunque mi abbia consigliato, chiesto aiuto e -soprattutto- criticato, troppi per essere qui elencati, per avermi ricordato che è fondamentale non smettere mai di imparare e migliorarsi.

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

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

1.2 Struttura della Tesi

Questa tesi è suddivisa in 5 capitoli.

- Introduzione : il capitolo che spiega lo stage svolto e le motivazioni che mi hanno portato a sceglierlo.
- Biologia Comutazionale : il capitolo che introduce il lettore alla biologia computazionale, fornendone una descrizione storica, storica, sulle metodologie utilizzate e sugli obiettivi perseguiti.
- Problema della Costruzione dello Splicing Graph : il capitolo che introduce al problema e formalizza lo Splicing Graph e descrive ad alcuni concetti teorici preliminari, quali allineamento semplice e multiplo di sequenze.
- Soluzione Proposta : il capitolo che spiega i vari passaggi che mi hanno portato all'elaborazione della soluzione proposta, presenta i concetti teorici alla base e descrive l'architettura del software creato.
- Conclusioni e sviluppi futuri : il capitolo che introduce allo splicing graph ciclico e descrive gli sviluppi futuri.

Capitolo 2

Biologia Computazionale

Con il termine *Biologia Computazionale* si intende la scienza che si prefigge l'obiettivo di sviluppare ed 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 ed 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 similari 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 strighe (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.

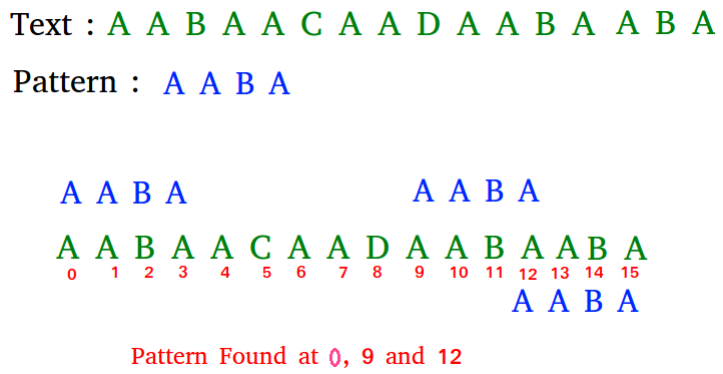


Figura 2.1: Esempio grafico di pattern matching

- 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 temine 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).

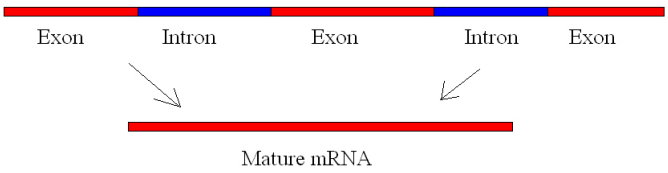


Figura 2.2: Alternanza di introni ed Esoni

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 sui 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.

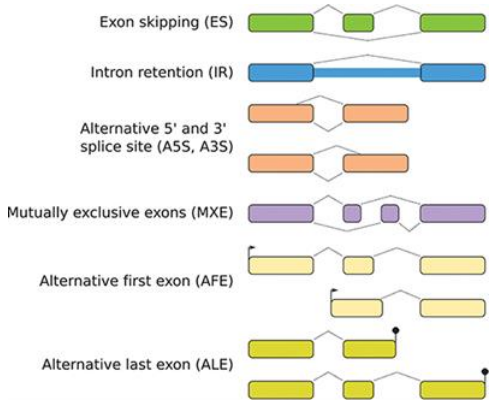


Figura 2.3: Esempi di Isoforme di Spliging

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

Capitolo 3

Problema della costruzione dello Splicing Graph

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 allineameto 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 strighe di lunghezza diversa.

La Distanza di Hamming è definita, solo su strighe 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 $d : \mathbb{S} \times \mathbb{S} \to \mathbb{R}_+$, 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'iserimento degli indel:

- Non possono esistere colonne di soli indel
- Le strinche 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

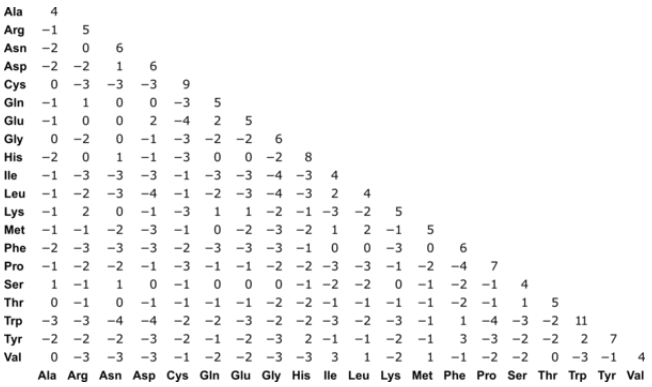
A	B	R	A	C	A	D	A	B	R	A
-	B	A	N	A	-	-	-	-	N	A

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'omologia, quindi la funzione obiettivo

Dove $score(s_1[i], s_2[j])$ è il valore del'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



	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Figura 3.1: La matrice di score BLOSUM62

questione. Ad esempio una matrice di score potrebbe essere BLOSUM62 e l'alfabeto $\Sigma = \{A, C, G, T\}$.

In breve un allineamento può essere interpretato come una misura di omologia tra due o più sequenze. I mismatches possono essere interpretati come punti di mutazione tra le sequenze, invece gli indel o le sequenze di indel (GAP) possono essere interpretate come punti di divergenza nelle sequenze genomiche

3.2 Algoritmi di Allineamento

Esistono due principali algoritmi che performano allineamenti: l'algoritmo di Needleman-Wunsch che performa un allineamento globale, cioè allinea le sequenze nella loro interezza e l'algoritmo di Smith-Waterman, che invece performa un allineamento locale, cioè evidenzia regioni simili nelle sequenze genomiche.

3.2.1 Algoritmo di Needleman-Wunsch

L'algoritmo di Needleman-Wunsch utilizza un approccio di programmazione dinamica per risolvere il problema e definisce una equazione di ricorrenza struttutata nel seguente modo :

$$M[i, j] = \text{allineamento ottimo su } s1[:i], s2[:j]$$

$$M[i,j] = \max \begin{cases} M[i\text{-}1, j\text{-}1] + d(s1[i], s2[j]) & \text{nessun indel} \\ M[i, j\text{-}1] + d(\text{-}, s2[j]) & \text{indel solo in } s_1 \\ M[i\text{-}1, j] + d(s1[i], \text{-}) & \text{indel solo in } s_2 \end{cases} \tag{3.1}$$

dove $d(s1[i], s2[j])$ è il punteggio ottenuto dalla matrice di score. Si ricoda che non è contemplato il caso di soli indel in colonna.

Le condizioni al contorno sono:

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

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 $M[0,0]$.

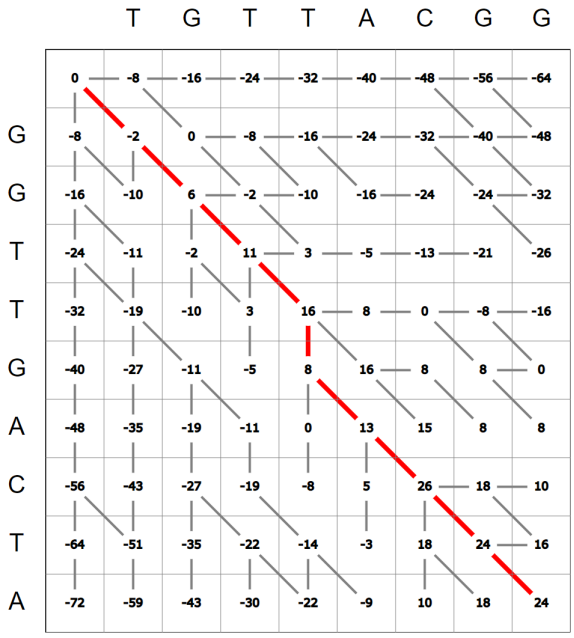


Figura 3.2: Esempio di esecuzione dell'algoritmo di Needleman-Wunsch su istanza : $s_1 = TGTTACGG$ e $s_2 = GGTTGACTA$

3.2.2 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:

$$M[i, j] = \text{allineamento ottimo su } s1[: i], s2[: j]$$

$$M[i, j] = \max \begin{cases} 0 & \\ M[i\text{-}1, j\text{-}1] + d(s1[i], s2[j]) & \text{nessun indel} \\ M[i, j\text{-}1] + d(\text{-}, s2[j]) & \text{indel solo in } s_1 \\ M[i\text{-}1, j] + d(s1[i], \text{-}) & \text{indel solo in } s_2 \end{cases} \tag{3.2}$$

dove $d(s1[i], s2[j])$, 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:

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

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.

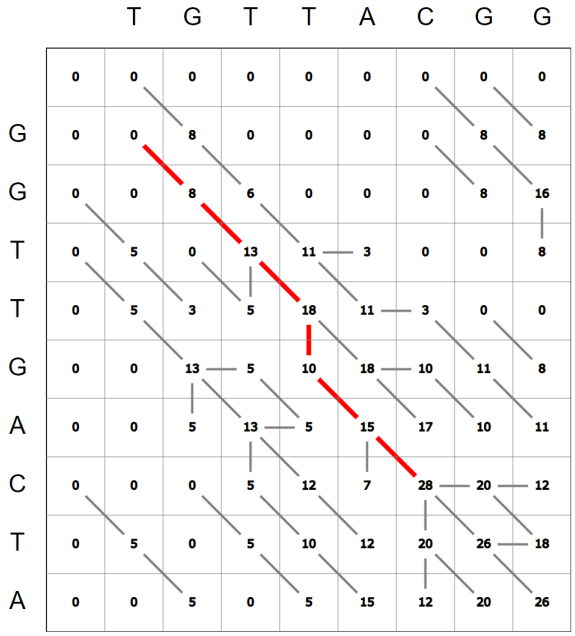


Figura 3.3: Esempio di esecuzione dell'algoritmo di Smith-Waterman sulla stessa istanza della figura 3.2

3.3 Allineamento multiplo di sequenze

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

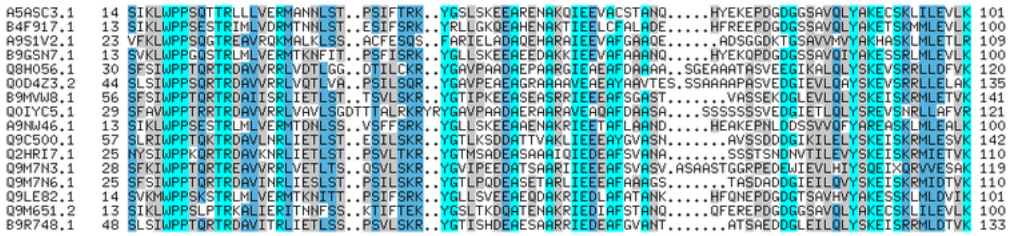


Figura 3.4: Esempio di allineamento multiplo tra sequenze

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 $k - 1$.

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

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 strighe, altrimenti è *NP-Completo*.

3.4 Splicing Graph

Come accennato nel capitolo riguardante la biologia molecolare, lo splicing alternativo è un fenomeno frequentemente osservato che sintetizza vari tracritti 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, accorpando, dove possibile, le parti comuni delle sequenze genomiche, dando una descrizione chiara e precisa dei trascritti in input.

3.4.1 Definizione Formale ed Esempio

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

- I VERTICI 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.

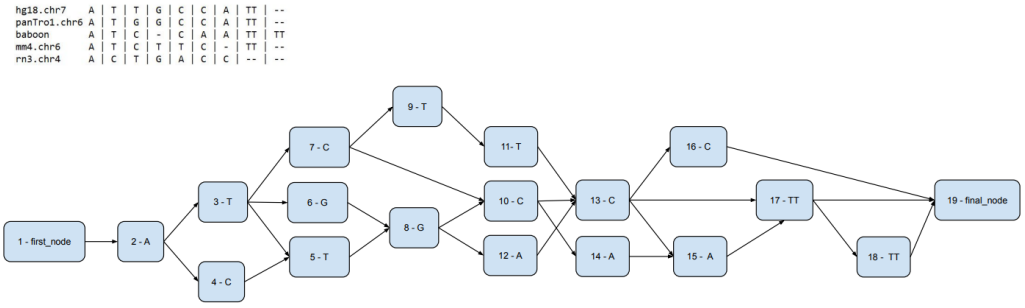


Figura 3.5: Esempio di Splicing graph, creato dal mio programma, con l'allineamento multiplo di riferimento

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

3.5 Costruzione dello Splicing Graph

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

In input è fornito un allineamento multiplo di 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 di 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 implememementato a partire da una libreria incompleta che lo eseguiva parzialmente.

Nel prossimo capitolo saranno dati i dettagli della soluzione proposta.

Capitolo 4

Soluzione proposta al Problema della costruzione dello Splicing Graph

Questo capitolo ha lo scopo di illustare 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 sui 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 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 patter 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

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

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 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 Soluzione Iniziale

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.

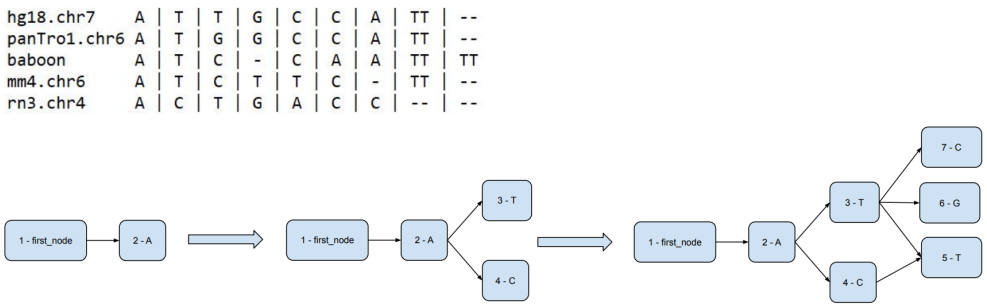


Figura 4.2: Le prime tre fasi della costruzione dello Splicing Graph. Vengono prese in considerazione le prime colonne dell'allineamento

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.

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, in modo da 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 percorsi $first\_node \rightarrow last\_node$. Per assolvere a questo compito, sono utilizzati due approcci, uno di *programmazione Dimanica* 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 $\mathbb{C} = \{C_1, C_2, ..., C_m\} \subseteq \Sigma^n$ un insieme di sequenze di lunghezza n su un alfabeto Σ detti *Ricombinanti*, ogni ricombinante è scritto come $C_i = c_{i1}, c_{i1}, ..., c_{in}$. Un insieme $\mathbb{F} = \{F_1, F_2, ..., F_k\} \subseteq (\Sigma \cup \{\text{-}\})^n$ 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 $C_i = f_{i1}f_{i2}...f_{ip_i}$ 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 α è uguale a quella di $|\alpha| = |f_{i1}...f_{ih-1}|$, mentre la lunghezza di β è uguale a quella di $|\beta| = |f_{ih+1}...f_{ip_i}|$.

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 un ricombinante C_i ha un *Punto di crossover* in j se il parse di C_i è $C_i = f_i...f_h f_{h+1}...f_{p_i}$ e $|f_1...f_h| = j - 1$.

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 λ_{min} e la lunghezza media dei frammenti λ_{ave}. Come è intuitivo pensare, questi sono due obiettivi contradditori.

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 $\mathbb{C}$ in termini 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 λ_{min} e/o λ_{ave} sotto la condizione che la cardialità 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 quato si è voluta tenere più bassa possibile la cardinalità di $\mathbb{F}$. Ciò ha portato 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 $c_{ij}...c_{ik}$ 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 minimimizza 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 $M(L), M(L+1)...M(n)$, nel seguente modo:

Caso base:

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

Passo Ricorsivo:

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

Il tempo di calcolo per questo approccio è $\mathcal{O}(n^2 m)$ 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 sottosequeza 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.

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.

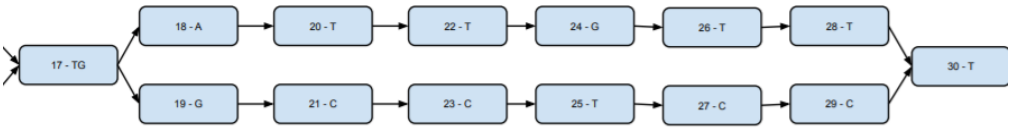


Figura 4.3: Primo Esempio di soluzione greedy. Per comodità, è riportato solo un frammento del grafo

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 soli 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.

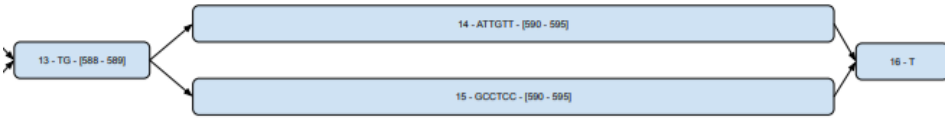


Figura 4.4: Esempio definitivo di soluzione greedy. Per comodità, è riportato solo un frammento del grafo e sono riportati gli indici dell'allineamento corrispondenti alle etichette dei nodi

4.4 Soluzione Definitiva

Tra il primo approccio e l'approccio definitivo si sono susseguiti numerosi tentativi. Ad esempio, nelle prime soluzioni, il calcolo nella cardinalità dei segmenti includeva anche le sequenze di soli indel, in seguito ciò è stato cambiato.

Per una maggiore chiarezza espositiva non verranno mostrati tutti i tentativi eseguiti, ma solo il primo e l'ultimo.

Avendo un partizionamento adeguato, costruito mediante soluzione greedy o di programmazione dinamica, la corstruzione 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.

Innanzitutto si itera sul partizionamento e per ogni segmento se ne estrae la cardinalità e l'insieme delle sottosequenze contenute, in seguito, per ognuna di queste, si crea un nodo che viene concatenato al precedente nodo correttamente, infine viene aggiunto al path corretto.

Per avere un'idea più chiara della costruzione del grafo, si può comunque fare riferimento alla figura 4.2, in quanto la costruzione avviene incrementalmente come mostrato. Verrà comunque fornita un'immagine contenente il codice Rust della procedura di costruzione del grafo (4.5).

I valori di soglia, M se si preferisce la soluzione greedy, o L se si preferisce quella di programmazione dinamica, sono impostabili dall'utente, ma è consigliato tenerli bassi in entrambi i casi, per fare in modo di costruire un grafo migliore. Ad esempio, nella figura 4.4, è stato scelto il valore $M = 1$, quindi l'algoritmo greedy di partizionamento.

```rust
fn build_vg(alignment : &Alignment, partition : &Partition) -> (HashGraph, Handle, Handle) {
    // Init
    let (mut vg, path, mut prev_handle, first_node) = VariationGraph::init(alignment);

    // Build
    for interval in partition.intervals() {
        let mut segment = Vec::new();

        for sequence in alignment.sequences() {
            let subsequence : Vec<u8> = sequence.seq[interval.begin..=interval.end].iter()
            .filter(|&&ch| ch != INDEL)
            .map(|&ch| ch)
            .collect();

            if !subsequence.is_empty() && !segment.contains(&subsequence) {
                segment.push(subsequence);
            }
        }

        for subsequence in segment {
            let handle = vg.append_handle(&subsequence[..]);
            for (i, sequence) in alignment.sequences().enumerate() {
                let clean_sequence : Vec<u8> = sequence.seq[interval.begin..=interval.end]
                .iter()
                .filter(|&&ch| ch != INDEL)
                .map(|&ch| ch)
                .collect();

                if clean_sequence == subsequence {
                    vg.create_edge(Edge(prev_handle[i], handle));
                    prev_handle[i] = handle;
                    vg.path_append_step(path[i], handle);
                }
            }
        }
    }

    // Epilogue
    let last_node = vg.append_handle(LAST_NODE_LABEL);

    for (i, handle) in prev_handle.into_iter().enumerate() {
        vg.create_edge(Edge(handle, last_node));
        vg.path_append_step(path[i], last_node);
    }

    (vg, first_node, last_node)
}
```

Figura 4.5: Procedura Rust di costruzione del Grafo di Splicing

4.5 Tempi di Calcolo

Fare un'analisi formale dei tempi di calcolo risulterebbe difficile, in quanto dipende da diversi fattori quali il tipo di partizionamento utilizzato, la lunghezza dei segmenti formati e il numero di sequenze in input.

La fase di preprocessamento delle sequenze, in cui viene estratto il partizionamento, ha tempi diversi a seconda del partizionamento utilizzato, in particolare:

- GREEDY : il tempo di esecuzione è $\mathcal{O}(nm)$.
- PROGRAMMAZIONE DINAMICA : il tempo di esecuzione è $\mathcal{O}(n^2m)$.

La vera e propria procedura di costruzione de grafo, è scomposta in tre fasi separate:

- FASE DI INIZIALIZZAZIONE che prende un tempo $\mathcal{O}(n)$ dove n è il numero delle sequenze, in quanto deve inizializzare i path per ogni sequenza e inserire $first\_node$ per ognuno di questi.
- FASE DI COSTRUZIONE che, come si può vedere in figura 4.5 prende un tempo che è proporzionale al numero di segmenti formati durante il pre-processamento, e al numero di sequenze in input. inoltre è proporzionale alla cardinalità del segmento preso in considerazione alla data iterazione del ciclo più esterno. In sistesi si potrebbe pensare ad un $\mathcal{O}(k(n + nm))$ dove n è il numero dei trascritti, k è il numero dei segmenti, m è la cardinalità del segmento.
- EPILOGO che prende un tempo $\mathcal{O}(n)$ dove n è il numero delle sequenze, in quanto deve finalizzare i path per ogni sequenza e inserire $last\_node$ per ognuno di questi.

In ogni caso viene complicato stabilire i tempi con accuratezza perchè, ad esempio la lunghezza dei segmenti diminuisce le iterazioni del ciclo più esterno, ma potrebbe allungare i tempi per il confronto delle sottostringhe.

4.6 Architettura del Software

Questa sezione è dedicata ad esporre l'architettura definitiva del software prodotto.

Verrà introdotta la struttura software del parser MAF e di quello Fasta, dopo di che verrà introdotta la struct *Partitioner* che si occupa del partizionamento e infine la struct *VariationGraph* che si occupa della costruzione e visualizzazione del grafo.

4.6.1 I parser MAF e FASTA

Figura 4.6: Esempio di allineamento memorizzato su file FASTA

```
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))

a score=23262.0
s hg18.chr7    27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon         116834 38 +   4622798 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
s mm4.chr6     53215344 38 + 151104725 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
s rn3.chr4     81344243 40 + 187371129 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG
```

Figura 4.7: Esempio di allineamento memorizzato su file MAF

Il parser Maf si avvale di una serie di struct ed enum, ognuna relativa ad un particolare elemento del file MAF.

Le prinipali sono:

- MAFItem: una enum che rappresenta un elemento del file MAF, quindi un commento od un blocco di allineamento.
- MAFBlock: una struct che rappresenta un blocco di allineamento che contiene un vettore di *MAFBlockEntry* e una mappa contenente i metadati di intestazione del blocco.

- MAFBlockAlignedEntry è una struct che rappresenta uno dei due tipi di *MAFBlockEntry*, questa struct contiene effettivamente la sequenza di basi allineate e l'identificativo della sequenza

Il paser Fasta si avvale di una libreria chiamata RustBio che performa il parser per noi.

Siccome i due parser producono oggetti differenti, è stato necessario "unificarli" in un unico oggetto che contiene solo le informazioni necessarie ai fini della costruzione dello splicing Graph.

Questo oggetto è stato chiamato *Alignment* e contiene :

- Le effettive sequenze allineate
- Per ogni sequenza viene memorizzato anche l'identificativo univoco
- Un metodo per stampare su file l'allineamento nel caso sia necessario.

Al fine di agevolare il parsing da ulteriori formati di file, nel caso ce ne fossero, è stato inserito un trait denominato *Parser*. Si può vedere un trait come un'interfaccia java che può fungere da linea guida, nel nostro caso, per lo sviluppo di ulteriori parser.

Inoltre è inserita una enum che incapsula gli errori che possono verificarsi al momento del parsing, ad esempio un file non trovato, oppure un file che non contiene nessun allineamento.

4.6.2 Partitioner

La struct *Partitioner* si occupa dei eseguire il partizionamento dell'allineamento multiplo. Per assolvere a questo compito si avvale di alcuni costrutti aggiuntivi.

```
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 4.8: Esempio di partizionamento su un'istanza semplice del problema

Come descritto nel capitolo precedente, il partizionamento viene eseguito da due algoritmi differenti che utilizzano due strategie diverse, perciò è sembrato opportuno definire un trait chiamato *Partitioner* che fungesse da linea guida per lo sviluppo delle struct effettive che effettuano il partizionamento.

Per la rappresentazione del partizionamento è stata scelta una classe che contiene un vettore di elementi di tipo *Inerval*. Interval è una struct congeniale allo scopo di una rappresentazione più chiara di un singolo segmento, si compone di un indice di inizio, chiamato *begin* e un indice di fine chiamato *end* che rappresentano rispettivamente l'inizio e la fine di ogni segmento. Gli intervalli sono sempre disgiunti e crescenti.

La struct che performa il partizionamento in prorammazione dinamica è denominata *DynProgPartitioner*. Questa struct si preoccupa di eseguire i controlli sul parametro *threshold* al fine di accertarne la validità, inoltre performa l'effettivo algoritmo di programmazione dinamica.

Per quanto riguarda quest'ultimo, per agevolare il traceback dell'array di programmazione dimanica, viene utilizzata una struct aggiuntiva, chiamata *Cell*, che contiene l'effettivo valore del partizionamento, ma anche un campo previous che è utile nella ricostruzione dei segmenti.

Al momento del traceback, vengono formati gli intervalli del partizionamento.

La struct che invece performa l'algoritmo greedy, è chiamata *GreedyPartitioner*. Questa struct performa il partizionamento nel modo descritto nel capitolo precedente, iterando sull'allineamento e verificando che i segmenti siano di cardinalità minore di *threshold*. Se la cardinalità diventa maggiore, cerca di allungare il segmento il più possibile fino a quando non si accorge

che si potrebbe formare un segmento di cardinalità minore o che potrebbe ampliarla uteriormente.

Per il calcolo della cardinalità di ogni segmento, viene utilizzato un metodo comune, dato che viene fatto nello stesso modo.

4.6.3 VariationGraph

La struct *Variation Graph* si occupa della costruzione dello splicing graph a partire da un partizionamento in input. Il partizionamento è ottenuto utilizzando il metodio greedy o di programmazione dinamica.

La struct *Variation Graph* contiene i riferimenti a first_node e last_node e un *HashGraph* che rappresenta l'effettivo grafo. HashGraph è un tipo di dato importato una una libreria esterna.

Il metodo *build_vg* si occupa della corstruzione progressiva del grafo a partire dal partizionamento in input, si avvale del metodo init per inizializzare tutti i *Path* relativi alle sequenze e l'array dei predecessori per ogni sequenza input.

Il metodo build_vg itera sulle partizione formate in precedenza per fare in modo di creare correttamente il grafo e i path contenenti i nodi correttamente etichettati e connessi. L'array dei predecessori è utile per tenere conto di quale nodo è presente nel path della sequenza associata, in modo da "Agganciare" correttamente il successivo.

La costruzione si conclude con un epilogo in cui ad ogni path è aggiunto il nodo last_node.

La struct VariationGraph contiene anche altri metodi quali :

- PRINT_PATH: che stampa a schermo il path associato all'identificativo della sequenza passato come parametro.
- GET_POSSIBLE_PATH: che restituisce il numero di percorsi $first\_node \rightarrow last\_node$. Questo metodo inizialmente è stato pensato come un metodo ricorsivo che si avvale di una funzione aggiuntiva per calcolare il numero di cammini possibili. Questo approccio però portava ad un tempo di esecuzione proporzionale al numero dei cammini presenti nel grafo, come si può vedere in figura 4.9.

  Di seguito, come mostrato in figura 4.10 è utilizzato un metodo di simil-programmazione-dinamica, che sfrutta il fatto di memorizzare il numero dei cammini che partono da un dato nodo e arrivano a last_node per rendere il tempo di esecuzione proporzionale al numero dei nodi, che è, in generale, molto inferiore a quello dei cammini.
- LABEL_LEN_SUM : che restituisce la somma delle lunghezze delle etichette dei nodi del grafo.

```rust
fn get_possible_paths_helper(&self, handle : Handle) {
    let outgoing_edges = self.graph.neighbors(handle, Direction::Right).count();

    if outgoing_edges == 0 {
        return 1;
    }

    let mut count = 0;
    for node in self.graph.neighbors(handle, Direction::Right) {
        count += get_possible_paths_helper(node);
    }

    count
}
```

Figura 4.9: versione del metodo $\mathcal{O}(|paths|)$

```rust
fn get_possible_paths_helper(&self, handle : Handle, occ : &mut Vec<usize>) {
    let outgoing_edges = self.graph.neighbors(handle, Direction::Right).count();

    if outgoing_edges == 0 {
        occ[u64::from(handle.id()) as usize] = 1;
        return;
    }

    let mut count = 0;
    for node in self.graph.neighbors(handle, Direction::Right) {
        if occ[u64::from(node.id()) as usize] == 0 {
            self.get_possible_paths_helper(node, occ);
        }
        count += occ[u64::from(node.id()) as usize];
    }

    occ[u64::from(handle.id()) as usize] = count;
}
```

Figura 4.10: versione del metodo $\mathcal{O}(|V|)$

- PRINT_GRAPH : che stampa il grafo in un modo opportuno. In particolare, per ogni nodo nel grafo stampa :
  - L'identificativo univoco
  - L'etichetta associata
  - I nodi entranti
  - I nodi uscenti

Nella figura 4.11 verrà fatto un esempio di costruzione di splicing graph per dare un'idea migliore di come funziona l'architettura software creata.

```rust
fn main() {
    let alignment = <MafParser as Parser>::get_alignment("./dataset/test_1.maf").unwrap();

    let output_file_name = "./dataset/test_1.aligned.txt";

    let partition = <GreedyPartitioner as Partitioner>::new(&alignment, 1).unwrap();

    alignment.dump_on_file(output_file_name, &partition);

    for elem in alignment.sequences() {
        println!("Seq : {}", elem);
    }

    let graph = VariationGraph::new(&alignment, &partition);

    for seq in alignment.sequences() {
        match graph.print_path(&seq.name) {
            Err(e) => println!("Error : {:?}", e),
            _ => {},
        }
    }

    graph.print_graph();
    println!("sum labels len : {}", graph.label_len_sum());
    println!("paths : {}", graph.get_possible_paths());
}
```

Figura 4.11: Esempio di costruzione del grafo di splicing

4.6.4 Test

Oltre alle struct sopra descritte, sono stati implementati una serie di test d'unità per verificare il corretto funzionamento dell'applicazione. Questi test si dividono in due parti:

- I test per verificare la correttezza dei Parser che testano i parser in condizioni normali e condizioni al contorno, ad esempio con un file inesistente, con un file che non contiene allineamento, con un file che contiene più allineamenti.
- I test per verificare la correttezza della costruzione dello Splicing Graph che fungono da esempio per l'utilizzo della libreria creata.

I test che verificano la correttezza del costruzione dello spliging graph non sono stati implementati a dovere perchè la libreria utilizzata non implementa il trait di uguaglianza tra elementi di tipo HashGraph, ciò non permette di verificare che il grafo sia formato corretamente

4.7 Considerazioni sulla qualità dell'allineamento in input

Dopo aver terminato la parte di costruzione del grafo di splicing, si è discusso della qualità dell'allineamento in input, che ovviamente è un fattore determinante per il corretto funzionamento del programma.

Il software utilizzato per ottenere l'allineamento in input è chiamato *Mafft*. Può performare un allineamento multiplo a partire da una serie di trascritti annotati su file Fasta con impostazioni di default che possono eventualmente essere personalizzate.

Fino a quel punto, mafft era stato utilizzato solo con parametri di default, perciò lo scopo dell'ultima parte dello stage è stato quello di trovare le corrette impostazioni di Mafft che permettessero di allineare correttamente i trascritti in input.

In questo software è possibile modificare, tra le altre cose, il peso di apertura ed estensione di GAP (sequenze di indel contigue), oltre che il costo dei mismatch (incolonnamento di due simboli differenti). La ricerca si è concentrata maggiormente su questi parametri sotto consiglio del tutor di stage.

Sono stati fatti innumerevoli tentativi di modifica dei parametri sopra descritti, ma nessuno di questi si è rivelato efficace. La configurazione "corretta" per una serie di trascritti si rivelava scorretta per un'altra, perciò non è stato possibile trovare una soluzione univoca al problema.

Nonostante ciò, un particolare degno di nota è che diminuendo il costo di apertura ed estensione dei GAP, il grafo di splicing si presenta, in generale, "migliore", nel senso che la somma della lunghezza delle etichette dei nodi è ridotta, rispetto alla lunghezza totale delle sequenze, di circa due terzi (in realtà dipende molto dalle sequenze in input) e il numero di percorsi $first\_node \rightarrow last\_node$ non esplode.

Capitolo 5

Conclusione e Sviluppi Futuri

Lo Splicing Graph aciclico, descritto in questa relazione, è un caso ristretto del problema di costruzione di uno splicing graph, infatti si potrebbe pensare ad introdurre cicli nel grafo, in modo da evidenziare ancora meglio le omologie tra i trascritti allineati.

L'introduzione dei cicli introduce una serie di problematiche che devono essere sviscerate e analizzate, tra tutte:

- Convenienza nella formazione di cicli: Nel momento in cui sono presenti sezioni duplicate nel grafo, si potrebbe pensare di inserire un ciclo, ma sezioni troppo corte produrrebbero un grafo con troppi cicli, che non evidenzierebbe al meglio le omologie tra le sequenze in input; si pensi ad esempio ad un grafo formato dai soli nodi corrispondenti ai simboli dell'alfabeto su cui sono costruite le sequenze, è chiaro che sarebbe poco utile e non congeniale agli scopi per cui è creato. D'altro canto, scegliere solo grossi frammenti per effettuare i cicli non evidenzierebbe correttamente le omologie presenti nei trascritti.
- Criteri di ottimalità del grafo: Quale criterio impostare per la "ricerca" di un buon grafo non è un problema semplice da risolvere, di certo non è possibile utilizzare i criteri di bontà descritti nei capitoli precedenti, in quanto, introducendo cicli, il numero di percorsi $first\_node \rightarrow last\_node$ sarebbero infiniti.

5.1 Sviluppi Futuri

In conclusione credo di aver preso parte ad un'esperienza molto formativa che mi ha insegnato ad affrontare le difficoltà che potrebbero presentarsi in questo lavoro.

Durante lo stage ho imparato ad utilizzare un nuovo linguaggio, all'inizio astruso, ma in corso d'opera piacevole da utilizzare. Ho imparato ad interfacciami con Git e Github per la manutenzione del progetto.

In generale ho affinato le conoscenze acquisite durante i precedenti tre anni di studio, ad esempio le conoscenze di analisi e progettazione del software, di algoritmi, è stato in generale uno stage formativo e consigliato.

Non sono certo di continuare il mio percorso nell'ambito della bioinformatica, certamente è stimolante, ma credo di conoscere ancora troppo poco di questo grande mondo.

Spero, alla fine della laure magistrale, di avere un'idea d'insieme più chiara di cosa è l'informatica e di cosa può offrire, in modo da scegliere correttamente cosa fare nella vita.

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. URL: http://www.sciencedirect.com/science/article/pii/S0167730608604585.

[7] Esko Ukkonen. «Finding Founder Sequences from a Set of Recombinants». In: (2002).