DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Implementazione e Analisi di uno Hierarchical Overlap Graph

Relatore: Prof.ssa Cinzia Pizzi

Laureando: Davide Buttolo
1224265

ANNO ACCADEMICO 2023 – 2024

Data di laurea 16/11/2023

Abstract

L'avanzamento delle tecnologie nel campo del sequenziamento del materiale genetico ha reso possibile un accesso sempre più economico a tali risorse. Di conseguenza, la mole di materiale sequenziato e di dati da analizzare, è in costante aumento. Ciò ha posto la necessità di sviluppare algoritmi e strutture dati efficienti per elaborare in modo ottimale questo crescente volume di informazioni. In particolare, tutte le moderne tecniche di sequenziamento richiedono successivamente una fase cruciale detta "assemblaggio", in cui le cosiddette "read" (brevi frammenti di materiale sequenziato) devono essere elaborate al fine di ricostruire la sequenza completa del materiale genetico di partenza. Attualmente, esistono diverse metodologie per effettuare questo processo di assemblaggio, ma le due più utilizzate si basano su due tipi di grafi: i "De Bruijn Graph" e gli "Overlap Graph". Recentemente, in letteratura è stata presentata una nuova struttura dati chiamata "Hierarchical Overlap Graph" che offre un significativo miglioramento in termini di efficienza rispetto ad esse. L'obiettivo principale di questa tesi è l'implementazione di questa nuova struttura, nonché la conduzione di analisi sperimentali mirate a verificarne l'efficienza.

Indice

1 Introduzione 1

2 Hierarchical Overlap Graph 4
2.1 Notazione 4
2.2 Struttura dati Trie 5
2.3 Algoritmo di Aho-Corasick 6
2.4 Extended Hierarchical Overlap Graph (EHOG) 6
2.5 Hierarchical Overlap Graph (HOG) 7
2.6 Algoritmo di Cazaux-Rivals 7
2.7 Algoritmo di Khan 9

3 Implementazione 11
3.1 Organizzazione della Struttura Dati 12
3.1.1 Prima Versione della Struttura Dati 12
3.1.2 Seconda Versione della Struttura Dati 15
3.1.3 Terza Versione della Struttura Dati 16
3.1.4 Inserimento delle Read nel Trie 17
3.2 Implementazione dell'Automa di Aho-Corasick 17
3.3 Implementazione dell'Extended Hierarchical Overlap Graph 17
3.4 Implementazione dello Hierarchical Overlap Graph 18
3.4.1 L'algoritmo di Cazaux-Rivals 18
3.4.2 L'algoritmo di Khan 18

4 Analisi delle Prestazioni 20
4.1 Descrizione dei dataset 20
4.2 Confronto dei tempi di esecuzione 21
4.3 Confronto dell'occupazione di memoria 23
4.4 Prestazioni con il Dataset GAGE-B 24

5 Conclusioni 26
5.1 Miglioramenti Proposti 26
5.2 Direzioni Future 27

Bibliografia 28

Capitolo 1
Introduzione

Nel campo della bioinformatica, l'assemblaggio è il processo mediante il quale si ricostruiscono sequenze genomiche complete partendo da frammenti chiamati "read". Questo procedimento è essenziale perché le tecniche di sequenziamento del materiale genetico (DNA o RNA) generano frammenti che coprono solo una piccola porzione del genoma in analisi. È fondamentale notare che l'assemblaggio è associabile al problema dello Shortest Common Superstring, noto per essere un problema di difficile risoluzione (NP-hard). Questo significa che significa che le metodologie utilizzate non possono garantire l'individuazione di una soluzione ottimale, ma si sforzano di fornire soluzioni di alta qualità entro limiti di tempo ragionevoli. Poiché non esiste un algoritmo efficiente per l'assemblaggio che possa affrontare tutte le possibili complicazioni derivanti dalla fase di sequenziamento, la ricerca in questo ambito rimane estremamente competitiva. L'assemblaggio di materiale sequenziato presenta varie sfide sia dal punto di vista della procedura di sequenziamento sia da quello computazionale e algoritmico.

Le problematiche associate alla fase di sequenziamento possono essere classificate in due categorie: la lunghezza delle read sequenziate e gli errori di sequenziamento. La lunghezza delle read sequenziate è principalmente influenzata dalla tecnologia di sequenziamento utilizzata, con due principali categorie: le "short-read", che producono read di lunghezze che variano dalle decine alle centinaia di basi azotate, e le "long-read", che possono superare alcune migliaia di basi. L'altro aspetto critico nella fase di sequenziamento riguarda il tasso di errori nella determinazione delle basi azotate e la tipologia di errori. Nelle short-read, l'errore più comune è la lettura errata di una base azotata, mentre nelle long-read, gli errori sono principalmente di

tipo "indel", ovvero l'identificazione di basi non presenti o l'omissione di basi presenti. Questo secondo tipo di errore è più problematico poiché comporta disallineamenti nelle sovrapposizioni tra le read.

Per quanto riguarda l'aspetto computazionale, le tecniche di assemblaggio sfruttano le sovrapposizioni parziali tra le read con l'obiettivo di unire tali sovrapposizioni per ricostruire la sequenza completa. Dato che tale problema rientra nella categoria NP-hard, spesso ci si concentra sulla soluzione di un problema intermedio meno complesso, ovvero la ricostruzione di sequenze denominate "contig", che rappresentano una sottosequenza della sequenza finale da trovare. Le "contig" possono successivamente essere sovrapposte con delle reference per migliorare il risultato finale. La complessità di questo problema non riguarda solamente i tempi di elaborazione, ma è anche legata a una notevole richiesta di memoria, poiché la quantità di basi analizzate può facilmente superare le decine di miliardi (10^{10}).

Le sovrapposizioni tra le read sono comunemente rappresentate tramite l'uso di grafi detti di assemblaggio. Le due tecniche principali in questo ambito sono i "De Bruijn Graph", in breve DBG, e gli "Overlap Graph", in breve OG. Nei DBG, ciascun nodo rappresenta un frammento di lunghezza fissa delle read sequenziate, denominato "k-mer", mentre gli archi collegano i nodi che condividono sovrapposizioni di lunghezza k-1. La scelta del valore di k (la lunghezza dei k-mer) influisce sia sul requisito di memoria di questo metodo che sulla qualità dei risultati, rendendo l'analisi molto sensibile a tale parametro. Gli OG, al contrario, rappresentano le sovrapposizioni tra le read in modo più diretto, in quanto i nodi contengono le read fornite in input, e gli archi rappresentano le sovrapposizioni tra il suffisso del nodo di partenza e il prefisso del nodo di arrivo. Gli OG sono in grado di rappresentare direttamente sovrapposizioni di lunghezza variabile tra due read, a differenza dei DBG, in cui le read vengono suddivise in k-mer. Gli OG rappresentano le sovrapposizioni in modo più naturale e comprensibile e non dipendono da un parametro fisso k, ma richiedono una maggiore complessità computazionale nella fase di creazione e occupano uno spazio che, nel peggiore dei casi, è quadratico rispetto al numero di read in ingresso.

In questa tesi verrà analizzata una nuova tecnica utilizzata per rappresentare le sovrapposizioni tra le read che è stata recentemente proposta in letteratura [1] e richiede spazio lineare rispetto alla dimensione dell'input. Questa tecnica sfrutta una struttura dati chiamata "Hierarchical Overlapping Graph" della quale in questa tesi saranno proposte delle possibili implementazioni ed eseguite delle analisi sperimentali delle loro prestazioni dal punto di vista computazionale.

La tesi è strutturata in 3 capitoli principali, il primo in cui vengono spiegati gli algoritmi e le strutture dati necessarie per la creazione di uno Hierarchical Overlapping Graph, il secondo nella quale vengono proposte delle possibili implementazioni di essa ed il terzo in cui vengono eseguiti dei test sperimentali per verificarne l'efficienza. Alcune idee per possibili sviluppi futuri vengono descritti nelle Conclusioni.

Capitolo 2

Hierarchical Overlap Graph

Lo Hierarchical Overlap Graph (HOG) è una struttura dati presentata inizialmente in [1] che permette di rappresentare in maniera efficiente la massima sovrapposizione di ogni stringa in un insieme detto dizionario con ogni altra stringa presente nello stesso dizionario, permettendo di migliorare l'efficienza del processo di assemblamento delle read. Sono state presentate in letteratura diverse tecniche di costruzione di questa struttura in maniera via via più efficiente [2] [3]. Prima di definire formalmente gli HOG e spiegare la loro struttura, nonché le tecniche di costruzione, è necessario fornire le nozioni fondamentali di algoritmi e strutture dati che sono utilizzati per la sua costruzione, in particolare la struttura dati Trie e l'algoritmo di multi-pattern matching di Aho-Corasik. Successivamente verrà introdotta la struttura dati Extended Hierarchical Overlap Graph (EHOG) che rappresenta una versione meno restrittiva dello HOG e per finire verrà introdotto lo HOG stesso.

2.1 Notazione

Definiamo di seguito una notazione che sarà comoda per le definizioni formali delle strutture e degli algoritmi. Le read vengono rappresentate con delle stringhe formate da simboli (di seguito indicati anche come caratteri) di un alfabeto. Nel campo dell'analisi di materiale genetico in bioinformatica l'alfabeto base più utilizzato è formato dai caratteri A, C, G e T anche se ne esistono delle varianti estese che comprendono appositi simboli per rappresentare l'incertezza tra 2 o più simboli. La lunghezza di una stringa s viene indicata con $|s|$ e rappresenta il numero di caratteri di cui è composta. Definiamo la sottostringa di una stringa s come la stringa formata dai caratteri di s tra gli indici i e j con $i \leq j$ e i,j in $[1, |s|]$. Definiamo inoltre il prefisso di s come una sottostringa con i=0, in modo analogo definiamo il suffisso di s come la

sottostringa di s con $j=|s|$. Se un prefisso o un suffisso di una stringa s differiscono da s, allora sono definiti prefisso proprio o suffisso proprio. Di seguito se non specificato altrimenti, prefissi e suffissi si riferiranno a prefissi propri e suffissi propri.

Definiamo anche sovrapposizione tra una stringa s ed un'altra stringa t (potrebbe anche essere $s=t$) un suffisso di s coincidente con un prefisso di t. La sovrapposizione massima tra s e t è la sottostringa di lunghezza maggiore che è suffisso di s e prefisso di t. Un insieme di read verrà chiamato anche dizionario e solitamente indicato con P. Il numero di read presenti in un dato dizionario è indicato con $|P|$ e solitamente verrà indicato con il simbolo n, se non diversamente specificato. La somma delle lunghezze di tutte le stringhe in un dato dizionario sarà infine indicata con $||P||$.

2.2 Struttura dati Trie

Un Trie o Prefix Tree è una struttura ad albero che serve a rappresentare un dizionario di stringhe in maniera ordinata. Questa struttura è essenziale poiché fornisce la base attorno a cui verrà costruito lo HOG. I nodi del Trie rappresentano tutti i prefissi di ogni stringa di un dato dizionario. La radice rappresenta la stringa vuota, che è prefisso di ogni stringa. Le foglie, invece, rappresentano le stringhe del dizionario che non sono prefissi propri di un'altra stringa. Definiamo ora in maniera formale il legame tra i nodi di un Trie:
dato un nodo p figlio di un nodo q, data una parola s nel dizionario, data la scomposizione di s in $x•a•y$ con "•" che indica la concatenazione, x e y stringhe (compresa la stringa vuota), a simbolo dell'alfabeto, x stringa rappresentata dal nodo q, allora p rappresenta la stringa $x•a$ prefisso di s, e tra q e p vi è un arco che rappresenta a.

Alternativamente alla struttura ad albero, è possibile considerare un Trie come un grafo con gli archi diretti originati da quelli che considereremo nodi "genitore" in un albero e che puntano ai nodi "figli".

Per quanto concerne la rappresentazione di un Trie, si potrebbe fare una breve considerazione utile per la progettazione della struttura da mantenere in memoria in un elaboratore: è possibile ricavare la stringa rappresentata da un nodo concatenando tutti i simboli rappresentati dagli archi da padre in figlio a partire dalla radice fino ad arrivare al nodo di interesse; viceversa è possibile ricavare il simbolo rappresentato da un arco a partire dalle stringhe rappresentate dai nodi, in quanto esso rappresenta l'ultimo carattere della stringa rappresentata del nodo puntato

dall'arco. La prima delle due modalità è evidentemente quella che permette di risparmiare più memoria, occupando al più $||P||$ caratteri in $O(||P||)$ spazio.

2.3 Algoritmo di Aho-Corasick

Il primo passo essenziale nella trasformazione del Trie in HOG, è quello di creare l'automa di Aho-Corasick [4]. Questo ricalca la struttura di un Trie aggiungendo però dei collegamenti ulteriori chiamati failure link. Un failure link è un arco diretto che connette un dato nodo ad un altro che rappresenta il più lungo suffisso proprio della stringa rappresentata dal nodo di partenza del collegamento. Si noti che che i failure link connettono un dato nodo ad un altro che si trova ad una profondità strettamente inferiore dato che i suffissi propri di una parola hanno lunghezza strettamente inferiore alla parola stessa.

Per inserire i failure link nel Trie, operazione necessaria per effettuare la trasformazione da Trie ad HOG, si utilizza l'algoritmo di Aho-Corasick che consiste in un attraversamento in ampiezza del Trie stesso, durante la quale vengono assegnati i failure link ai nodi. Ogni nodo ha un failure link eccetto la radice, che ne è sprovvista, dovendo rappresentare la stringa vuota che non possiede suffissi propri.

Per trovare il failure link di un nodo, consideriamo che dato un nodo v il cui failure link è da trovare ed s il simbolo rappresentato dal collegamento dal suo genitore a v, per trovare il nodo failure è necessario risalire l'albero considerando solo i failure link a partire dal genitore di v, fino a trovare un nodo che ha tra i suoi figli un nodo raggiungibile con il simbolo s.

2.4 Extended Hierarchical Overlap Graph (EHOG)

L'Extended Hierarchical Overlap Graph è una struttura dati a grafo che permette di rappresentare tutte le sovrapposizioni di ogni stringa di un dato dizionario con ogni altra stringa del dizionario e sé stessa.

Questa struttura può essere ricavata in maniera efficiente partendo dall'automa di Aho-Corasick, eliminando i nodi superflui e comprimendo i collegamenti dei nodi da eliminare in modo da non perdere la loro informazione. In questo modo gli archi che connettono un nodo genitore ad uno figlio potrebbero rappresentare non più solo un simbolo di una stringa, ma una sua sottostringa. Per effettuare la marcatura dei nodi da mantenere, è sufficiente effettuare un attraversamento del grafo durante la quale, a partire dalle foglie, vengono visitati tutti i nodi raggiungibili con i soli failure link. La fase di visita dell'attraversamento consiste nella sola marcatura dei nodi visitati. Una volta marcati i nodi, è sufficiente eliminare i nodi non marcati, facendo attenzione a non eliminare l'informazione delle parti di stringa da essi rappresentate.

2.5 Hierarchical Overlap Graph (HOG)

Lo Hierarchical Overlap Graph è una struttura dati a grafo che riprende la definizione di EHOG, imponendo però che vengano mantenuti i soli nodi che rappresentano la massima sovrapposizione tra una stringa s di un dizionario e tutte le stringhe presenti in esso, compresa s stessa. Il fatto di mantenere solo l'informazione della massima sovrapposizione tra le stringhe permette, in teoria, a questa struttura di avere un vantaggio che, all'aumentare della dimensione n del dizionario P utilizzato:

$$\lim_{n\to\infty} \frac{\#\ di\ nodi\ di\ HOG(P)}{\#\ di\ nodi\ di\ EHOG(P)} = 0$$

Lo HOG permette quindi un vantaggio in termini di spazio occupato che tende all'infinito all'aumentare di n. Considerazioni su questo aspetto verranno effettuate nel Capitolo 4.

Per la creazione di questa struttura sono stati proposti vari algoritmi, in questa tesi ne verranno analizzati 2: il primo proposto in letteratura ed una delle varianti proposte successivamente che ne eliminano la principale criticità.

(a) (b) (c)

Figura 2.1. Dato il dizionario P = {aabaa, aadbd, dbdaa} nella figura sono riportati da sinistra a destra il grafo dell'Automa di Aho-Corasick (a), quello dell'EHOG (b) e quello dello HOG (c) di P. Le frecce tratteggiate rosse rappresentano i failure link. Fonte [3].

2.6 Algoritmo di Cazaux-Rivals

Il primo algoritmo che andiamo ad analizzare, presentato in [2], propone di creare uno HOG a partire da un EHOG in tre fasi: la prima fase consiste nella creazione delle liste R_l, la seconda fase è quella della marcatura dei nodi, infine avviene la contrazione del grafo lasciando i soli nodi marcati. La terza fase, quella della contrazione avviene in maniera simile alla contrazione già discussa nella creazione dell'EHOG discussa nella sezione precedente, quindi non verrà

trattata in questa. La prima fase consiste nella creazione di una lista, che indicheremo con R_l, per ogni nodo del grafo EHOG in ingresso, contenente i riferimenti di tutte le foglie a partire dalle quali, seguendo i soli failure link, è possibile raggiungere tale nodo. Questa fase avviene in O(||*P*||) dato che bisogna creare al più ||*P*|| liste e risalire l'albero tante volte quante sono le foglie, cioè *n*, fino a raggiungere la radice, effettuando una visita che aggiunge un riferimento alla volta alle liste in O(1). Quindi questa fase viene eseguita in tempo

$$O(||P|| + \sum_{i=0}^{n} |s_i|) = O(||P||).$$

La fase successiva serve per trovare le massime sovrapposizioni tra coppie di stringhe in *P*. Per fare questo si noti che dato un nodo interno *u* puntato da un failure link che parte da una foglia nella lista R_l di *u*, indicata con $R_l(u)$, che rappresenta la stringa *s*, allora tra tutte le stringhe rappresentate nel sottoalbero di *u* ed *s* ci sarà una sovrapposizione lunga almeno pari alla stringa rappresentata da *u*. Tale sovrapposizione sarà massima se e solo se nessuno dei nodi presenti nel sottoalbero di *u* rappresenta una sovrapposizione con *s*.
Viene quindi effettuato un attraversamento in profondità in maniera ricorsiva durante la quale l'informazione della presenza di una sovrapposizione massima viene passata da figlio in genitore con l'ausilio di un array di booleani, di modo da marcare i soli nodi che rappresentano una sovrapposizione massima.
Il caso base si verifica nelle foglie che sono marcate per definizione e passano un array contenente solo *False*, infatti le foglie rappresentando una stringa completa di *P*, non possono essere delle sovrapposizioni proprie. Nella fase ricorsiva, ogni nodo interno riceve dai figli l'array contenente i riferimenti delle foglie di cui sono massima sovrapposizione, viene fatto un AND logico tra gli array, ottenendo quindi un array contenente *True* per i soli riferimenti delle stringhe per la quale è stata trovata una sovrapposizione maggiore nel sottoalbero di *u*, *False* altrimenti. Viene quindi scansionata la lista $R_l(u)$ e verificato se corrispondono alla condizione *False* nell'array booleano per trovare le stringhe che hanno sovrapposizioni con *u* ma che non ne hanno con i suoi figli. Se presente almeno una volta questa condizione *u* viene marcato. La fase ricorsiva termina consegnando al nodo genitore una copia dell'array booleano modificato per indicare le stringhe della quale è massima sovrapposizione.
Per quanto riguarda la nostra analisi, si nota che questa fase risulta quadratica in ||*P*||, considerando che le read in *P* hanno lunghezza costante. In questo caso |*P*|=Θ(||*P*||). In questa fase è richiesto l'attraversamento del grafo in O(||*P*||) e la creazione degli array che richiedono la memorizzazione dei riferimenti alle foglie, che sono lunghi al più |*P*| elementi in O(|*P*|) e ne

vengono creati uno per foglia. In totale avremmo quindi O(|P|) operazioni che richiedono O(|P|) tempo; quindi, in totale l'algoritmo richiede per l'attraversamento e la marcatura tempo

$$O(||P||+|P|^2) = O(||P|| + ||P||^2) = O(||P||^2).$$

In totale lo spazio richiesto per la trasformazione da EHOG a HOG, rimane uguale all' spazio necessario per creare l'EHOG, cioè O(||P||).

2.7 Algoritmo di Khan

Il secondo algoritmo che andiamo ad analizzare, quello proposto in [4], differisce da quello precedentemente illustrato nella sola fase di marcatura, quella che rendeva l'algoritmo di creazione dello HOG quadratico. La nuova fase di marcatura utilizza come quella precedentemente descritta le liste R_l associate ai nodi. Queste liste contenenti i riferimenti alle foglie dalle quali è possibile raggiungere un nodo *u* utilizzando i soli failure link, possono essere alternativamente viste come i riferimenti delle stringhe che hanno come suffisso la stringa rappresentata da *u*. In questo modo sappiamo che la sovrapposizione maggiore tra due stringhe *x* e *y*, sarà rappresentata dal nodo di profondità maggiore tra gli antenati di *x* che conterrà nella sua lista R_l un riferimento a *y*. Quindi l'algoritmo di Khan prevede per marcare tutti i nodi necessari per la creazione dello HOG, basterà effettuare un attraversamento in profondità dell'albero e:

1. durante la discesa verso una foglia memorizzare il riferimento al nodo interno *u,* considerato attualmente nella visita, nelle foglie appartenenti ad $R_l(u)$ con l'ausilio di uno stack e inserire un riferimento a tali foglie in una lista *S*;
2. arrivati ad una foglia, marcare tutti i nodi che sono l'ultimo inserito per ogni stack associato ad ogni foglia memorizzata nella lista *S* citata al punto precedente e svuotare tale lista;
3. durante la fase di risalita, eliminare il nodo *u* visitato da ogni stack in cui era stato inserito.

La complessità temporale di questa variante può essere calcolata considerandola come la somma di due contributi, la visita dei nodi interni dell'albero e la visita delle foglie. La visita dei nodi interni complessivamente fa un lavoro che equivale ad iterare due volte (una durante la discesa ed una durante la risalita) su tutte le liste R_l, quindi è O(||P||) come già verificato precedentemente; nella visita delle foglie vengono analizzate tutte e sole le foglie che sono nella lista *S,* che se sommate per tutte le foglie sono al più O($\sum |R_l|$) e quindi O(||P||).

In totale abbiamo che la visita impiega tempo O($\|P\|$) per i nodi interni ed altrettanto per le foglie; quindi, in totale la marcatura impiega tempo O($\|P\|$).

Capitolo 3
Implementazione

Il principale aspetto di studio svolto in questo scritto riguarda l'implementazione pratica della struttura HOG descritta nel capitolo precedente, analizzandone ogni aspetto e scelta implementativa durante l'intera fase di sviluppo, comprese le scelte intraprese inizialmente e modificate durante la fase di sviluppo. Per scegliere il linguaggio di programmazione sono stati considerati aspetti come la possibile quantità elevata di dati da analizzare durante l'utilizzo di questo software, la presenza di pacchetti nativi o di terze parti compatibili con tale linguaggio che facilitassero lo sviluppo e la facilità di implementazione dovute ad esperienze pregresse. La scelta è ricaduta sul linguaggio C++ che permette l'utilizzo efficiente di memoria se correttamente sviluppato, offre strutture dati basilari utili direttamente nella libreria standard, nonché librerie open source di libero utilizzo ottimizzate per svariati casi d'uso.

A seguire verranno presentati i vari aspetti dello sviluppo e creazione dello HOG, suddividendoli in macro categorie come nel capitolo precedente spiegando anche le differenti versioni della struttura di appoggio dello HOG stesso.

3.1 Organizzazione della Struttura Dati

Per l'implementazione della struttura dati sono stati testati tre approcci differenti nel tentativo di migliorare soprattutto l'occupazione di memoria della struttura dati stessa.

3.1.1 Prima Versione della Struttura Dati

L'implementazione di una struttura ad albero o grafo in letteratura viene spesso implementata con entità separate, i nodi, allocate in maniera sparsa in memoria e connesse con l'utilizzo di puntatori che rappresentano gli archi. Questa implementazione, che ricorda quella di una linked list, comporta un'alta percentuale di cache miss durante le fasi di attraversamento del grafo necessarie durante la costruzione ed analisi del grafo stesso, dovute al collocamento sparso in memoria dei nodi.

Per ovviare all'inconveniente sopra menzionato, nella prima implementazione sono state create delle entità rappresentanti i nodi (in C++ implementate attraverso l'uso di una struct) self-contained, cioè contenenti tutte le informazioni necessarie: informazioni di stato ed i collegamenti verso gli altri nodi. Dopodiché al posto di allocare dinamicamente la memoria occupata dai nodi uno alla volta, questi sono stati inseriti in un vettore che li contenesse in uno spazio contiguo.
Come riportato nel Capitolo 2, per rappresentare un Trie, non è necessario che ogni nodo contenga la stringa che rappresenta, bensì è sufficiente che sia presente l'informazione dei simboli rappresentati dai collegamenti tra i nodi di "padre" in "figlio". Intuitivamente, si può anche pensare che ogni nodo avendo un solo nodo "padre" (eccetto la radice), potrebbe contenere le informazioni dell'arco che lo collegano al nodo padre, non richiedendo così la presenza di un'entità esplicita separata che rappresenta i collegamenti tra i nodi che conterrebbe solo l'informazione del simbolo rappresentato. Ogni nodo ha quindi necessità di mantenere le seguenti informazioni: riferimento al padre, riferimenti ai figli di cui successivamente è presente un commento, failure link (utilizzato solo dopo aver trasformato il Trie in un automa di Aho-Corasick), un flag per evidenziare nelle fasi successive se un nodo è marcato o meno ed infine il simbolo (o la stringa nelle fasi successive) che rappresenta il collegamento con il nodo padre. Il collegamento con i figli in un Trie, potrebbe essere ricreato utilizzando un array di lunghezza fissa, nella quale ad ogni indice si assegna per convenzione un simbolo dell'alfabeto. In questo modo si evita l'utilizzo dello heap. Durante lo sviluppo per raggiungere l'obiettivo di costruire uno HOG, ci si accorge però che durante le trasformazioni successive, un nodo

potrebbe arricchirsi di figli ulteriori dovuti alle cancellazioni di altri nodi. Di conseguenza per mantenere la struttura come descritta fino ad ora, è stato utilizzato un `std::vector` per mantenere i riferimenti ai nodi figli. Durante la fase di inizializzazione di un nodo sono stati assegnati vettori con già 4 elementi che rappresentano un collegamento mancante, in modo da velocizzare la fase di inserimento delle read. Successivamente i collegamenti rimasti vuoti venivano rimossi.

```
struct Node
{
    string label;                               // 24 bytes
    int parent;                                 // 4 bytes
    vector<int> children{-1, -1, -1, -1};       // 24 bytes
    int failure_link{-1};                       // 4 bytes
    bool marked{false};                         // 1 byte

    Node(string lbl, int prnt) : label{lbl}, parent{prnt} {}
    ...
};

class Trie
{
private:
    vector<Node> nodes{}; // struttura che contiene i nodi dell'albero
    ...

public:
    Trie(const vector<string> &reads);
    ...
};
```

Figura 3.1. Porzione di codice dove viene definita la struttura `Node` *con accanto ad ogni variabile membro la dimensione in byte occupata nella struttura (senza considerare l'utilizzo dello heap per la stringa* `label` *ed il vettore* `children`*) e della classe* `Trie` *che ha come variabile membro un* `std::vector<Node>` *contenente i nodi.*

I nodi, come detto precedentemente, sono stati memorizzati in un vettore, in modo da essere localizzati in memoria in uno spazio contiguo. Per i collegamenti tra i nodi, rispetto all'utilizzo di puntatori che nelle architetture moderne occupano 64 bit, sono stati utilizzati degli interi da 32 bit che contengono l'indice della posizione nel vettore dei nodi, del nodo puntato. L'utilizzo di un semplice `int` ha permesso la diminuzione di memoria occupata dai collegamenti, a discapito del limite del numero massimo di nodi puntabili, cioè $2^{32} - 1$. Da notare che lo spazio di nodi non è di dimensione 2^{32} poiché il numero $2^{32} - 1$ che è il massimo numero rappresentabile con 32 bit è stato utilizzato per la rappresentazione di un collegamento non esistente, lasciando lo spazio utilizzabile $[0,\ 2^{32} - 2]$ estremi compresi. Le implementazioni proposte consentono di modificare facilmente il codice di modo da poter utilizzare riferimenti

a 64 bit per ospitare un numero di nodi superiore al limite imposto dall'utilizzo dei 32 bit di circa 4 miliardi di nodi, ma trattandosi di un primo progetto di implementazione ed analisi, si è ritenuto che questo limite non fosse troppo restrittivo, permettendo d'altro canto di risparmiare della memoria. La specifica sull'utilizzo di riferimenti a 32 bit è quindi rimasta invariata in tutte le versioni sviluppate.

Analizzando questa prima versione, sono emerse tre principali criticità, la prima dovuta all'allineamento in memoria delle variabili membro dei nodi, la seconda dovuta all'utilizzo di una struttura dinamica per i link con i nodi figli, la terza lo spreco di bit per la marcatura. La prima era causata dalla presenza nelle variabili membro di elementi con posizione nella struttura non multipla della propria dimensione, causando dello spreco di memoria per effetto del *padding*, ovvero dello spazio vuoto tra la memoria allocata per una variabile membro ed un'altra. Nell'esempio specifico, se compatta, la struttura dovrebbe occupare 57 byte, mentre nella pratica ne occupa 72. La seconda comporta dei problemi simili a quelli criticati in precedenza, ovvero l'utilizzo di memoria allocata nello heap in maniera non contigua non consente un efficiente utilizzo della cache, inoltre si rende necessaria una doppia dereferenziazione per raggiungere un nodo figlio, una per raggiungere il contenuto del vettore ed una per raggiungere il nodo cercato. La terza similmente alla prima, comportava uno spreco di memoria per indicare se un nodo era marcato o meno (necessario per le fasi successive), in quanto basterebbe 1 bit per memorizzare questa informazione, ma nella realtà viene allocato un byte intero.

type	*name*
std::string	label
int	parent
std::vector<int>	children
int	failure_link

padding

Nodo 0

type	*name*
std::string	label
int	parent
std::vector<int>	children
int	failure_link

padding

Nodo 1

...

Figura 3.2. Rappresentazione della disposizione in memoria dei nodi inseriti nel Trie nella prima versione

Durante lo sviluppo di questo progetto, lo sviluppo della prima versione della struttura dati è stato interrotto prima del suo completamento totale per sostituirla con la seconda, che è priva delle problematiche sopra menzionate.

3.1.2 Seconda Versione della Struttura Dati

La seconda versione della struttura sviluppata, tenta di eliminare tutte e tre le criticità evidenziate in quella precedente.

Per eliminare il *padding* presente nella lista di nodi che provocava uno spreco di memoria, è stato eliminato il concetto di identità esplicita contenente tutte le informazioni di un nodo, raggruppando le varie proprietà di tutti i nodi in un'unica struttura collettiva, nello specifico sono stati utilizzati un std::vector per ogni variabile membro dei nodi. Un nodo è quindi rappresentato da una ennupla formata da tutti gli elementi presenti alla stessa posizione dei vettori.

Questa modifica elimina anche il terzo dei tre problemi evidenziati nella prima versione, poiché la struttura std::vector<bool>, in automatico occupa un solo bit per elemento, eseguendo in automatico tutte le operazioni necessarie (bitmask, shift, …) per inserire o estrarre un elemento.

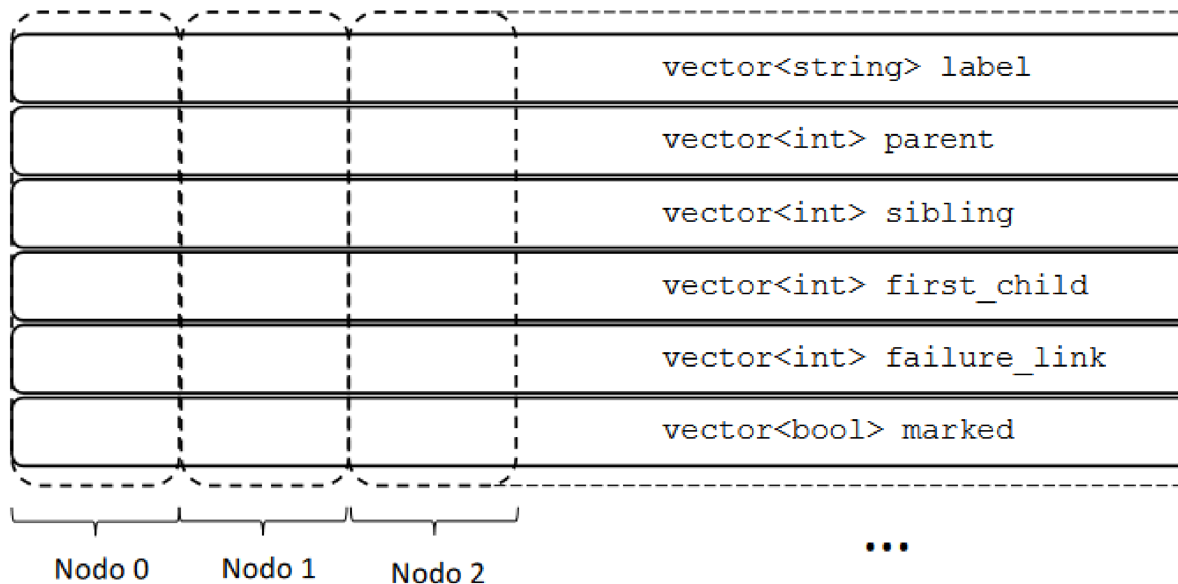

Figura 3.3. Rappresentazione della disposizione in memoria dei nodi inseriti nel Trie nella seconda versione. I nodi non sono degli oggetti memorizzati in memoria singolarmente bensì ogni nodo è formato dai valori presenti allo stesso indice dei vettori label, parent,… .

Eliminare il secondo dei problemi evidenziati nella prima versione, richiedeva l'eliminazione del vettore che memorizzava i riferimenti ai figli di un nodo. La prima idea per sostituirlo è stata quella di utilizzare in ogni nodo un riferimento ad un nodo figlio definito "primo nodo figlio" ed un riferimento ad un eventuale nodo fratello. In questo modo per visitare i figli di un

nodo è sufficiente visitare il primo figlio e poi tutti i fratelli seguendo quella che potrebbe essere considerata una linked list a singola direzione.
La seconda idea, ispirata dallo studio del codice proposto come materiale aggiuntivo in [5], è stata quella di utilizzare l'idea appena menzionata senza l'utilizzo del riferimento al primo figlio, che diventa un'informazione implicita. Per ricavare tale informazione, i nodi vengono posizionati in modo che il primo figlio di un nodo sia il successivo del nodo padre. Questa modifica, come vedremo, rende più complicate le operazioni di modifica dell'albero (inserimento ed eliminazione dei nodi), poiché è necessario mantenere la caratteristica che se un nodo ha almeno un figlio, questo sia posizionato all'indice successivo. Ad esempio, prima di inserire le read nel Trie, queste devono venir ordinate in ordine lessicografico per fare in modo che l'inserzione di una read r_1 che è prefisso di un'altra read r_2 avvengano una di seguito all'altra senza che nessun'altra read sia inserita nel frattempo. Da notare che in questo caso la creazione dello HOG rimane $O(||P||)$, mentre l'intera procedura richiede tempo $O(||P|| \log(||P||))$. Per poter sapere se presente o meno un collegamento padre-figlio tra un nodo ed il successivo, viene utilizzata una struttura di supporto (un vettore di booleani) che indica se il nodo in un dato indice è foglia o nodo interno. In questo modo è possibile verificare se un nodo ha figli ed in caso positivo sapere che è il primo inserito nell'indice successivo.

3.1.3 Terza Versione della Struttura Dati

La terra ed ultima iterazione della struttura, porta una sola modifica, anche se sostanziale e che può comportare, non sempre, un elevato risparmio di memoria. La modifica consiste nell'utilizzare le read in ingresso per memorizzare le stringhe inserite nell'albero senza dover tenere delle copie di tutte le sottostringhe di interesse all'interno del Trie. Questa modifica comporta un minor uso della memoria nei casi in cui l'overhead dovuto al grande numero di `std::string` utilizzato è elevato ed elimina la frammentazione della memoria in cui sono memorizzate le stringhe. Per poter indirizzare una specifica sottostringa, sono state introdotte due nuove variabili nei nodi, una che indica la read specifica di interesse ed una che indica quanto è lunga la sottostringa.
Da notare che questa modifica non comporta sempre una diminuzione della memoria occupata, poiché nel caso di forte ridondanza, ad esempio con input artificiali costruiti appositamente, la memoria allocata con questa versione è superiore a quella della versione precedente. Nei casi reali di interesse però questo non avviene come mostrato nel capitolo successivo nelle analisi.

3.1.4 Inserimento delle Read nel Trie

L'inserimento delle read in usa struttura come il Trie, avviene considerando le stringhe da inserire una ad una ed analizzandone in sequenza i caratteri che la compongono confrontandoli con i simboli rappresentati dai nodi presenti nell'albero, a partire dalla radice, e scendendo di un livello ad ogni carattere analizzato, scegliendo il prossimo nodo tra i figli, se presenti, tale per cui rappresenti lo stesso carattere del prossimo da analizzare nella stringa. Se ad un certo punto la stringa analizzata continua con una sequenza che non è stata precedentemente inserita nel Trie, vengono creati dei nuovi nodi che rappresentano la parte restante della stringa.

3.2 Implementazione dell'Automa di Aho-Corasick

Per la costruzione dei failure link necessari per le fasi successive della costruzione dello HOG, è stato utilizzato l'algoritmo di Aho-Corasick come specificato nel Capitolo 2. Per effettuare la visita dei nodi in ampiezza è stata utilizzata una tecnica iterativa con una coda che mantenesse i riferimenti dei nodi ancora da visitare. La visita consiste nel risalire l'albero attraverso i failure link dei livelli superiori fino a trovare il nodo che rappresenta il suffisso maggiore del nodo analizzato. La radice rappresenta un caso speciale in quanto il suo failure link non è propriamente definito, quindi secondo la convenzione sopra menzionata è stato utilizzato un valore di riferimento per contrassegnare l'assenza di tale collegamento.

3.3 Implementazione dell'Extended Hierarchical Overlap Graph

La costruzione del EHOG avviene in due passaggi, il primo è necessario a marcare i nodi che devono essere preservati, il secondo contrae l'albero eliminando i nodi non più necessari ed accorpando ai nodi non eliminati l'informazione rappresentata dagli archi in entrata ed uscita dai nodi eliminati.

La prima fase, quella della marcatura, consiste quindi in una visita dell'albero a partire da ogni foglia risalendo l'albero esclusivamente utilizzando i failure link e marcando tutti i nodi che si visitano nel frattempo. La risalita avviene fino ad arrivare alla radice oltre la quale non è possibile proseguire, o fino ad arrivare ad un nodo già marcato, dato che il precorso che si andrebbe a marcare sarebbe già stato sicuramente marcato in precedenza.

La seconda fase, quella della contrazione, consiste in un attraversamento dell'albero durante il quale i nodi non marcati nella fase precedente vengono eliminati. L'eliminazione di un nodo deve avvenire in modo da non alterare le sottostringhe rappresentate, altrimenti il rischio è

quello di eliminare delle sottostringhe di read inserite nell'albero. Quindi, se un nodo viene eliminato, bisogna effettuare un merge delle informazioni riguardanti gli archi in entrata ad esso (quello che arriva dal nodo genitore e l'eventuale failure link) con i nodi figli o quelli antenati del nodo da eliminare.

3.4 Implementazione dello Hierarchical Overlap Graph

Entrambi gli algoritmi illustrati nel capitolo precedente sono stati implementati, in primo luogo è stato implementato l'algoritmo quadratico, successivamente durante la fase di testing, è emersa la difficoltà nell'analizzare sample che simulassero casi d'uso realistici, data la sua natura non lineare. A questo punto è stata implementata anche la versione ottimizzata per poter effettuare un confronto diretto tra i due algoritmi.

3.4.1 L'algoritmo di Cazaux-Rivals

L'implementazione dello HOG con l'algoritmo di Cazaux-Rivals rispecchia lo pseudocodice proposto dagli autori originali, e segue quanto spiegato al capitolo precedente. Si ritiene opportuno sottolineare un aspetto piuttosto importante, riguardante l'algoritmo di marcatura dei nodi. Come spiegato nel capitolo precedente, questa fase dell'algoritmo è ricorsiva, e nella visita dei nodi interni è necessario calcolare l'AND logico di un vettore di booleani lungo $|P|$. Per effettuare questa operazione in modo efficiente, è stata utilizzata la struttura dati `dynamic_bitset<>` presente nella libreria open source boost [6]. Questa permette la vettorizzazione delle operazioni eseguendo in parallelo 128, 256 o 512 AND logici (in base all'architettura della CPU), grazie alle istruzioni SIMD (Single Instruction Multiple Data) presenti nelle moderne architetture dei processori.

3.4.2 L'algoritmo di Khan

L'implementazione dello HOG con l'algoritmo di Khan, è stato sviluppato seguendo la maggior parte delle specifiche proposte dall'autore, mentre altre sono state modificate per agevolare lo sviluppo. L'unica modifica all'algoritmo è l'utilizzo di una HashMap al posto di una lista concatenata, per contenente gli stack associati alle foglie. La HashMap permette tempi di accesso costanti nel caso medio, quindi non peggiora la complessità dell'algoritmo originariamente proposto. Come spiegato nel capitolo precedente, l'algoritmo di Khan differisce da quello di Cazaux-Rivals nella fase di marcatura che prevede l'attraversamento dell'albero. Una delle scelte implementative non presenti nel metodo proposto, poiché non modifica la

complessità del metodo stesso, riguarda la modalità di attraversamento dell'albero, se iterativa o ricorsiva. In questo caso è stata scelta la modalità iterativa perché ritenuta più vantaggiosa.

Capitolo 4

Analisi delle Prestazioni

In questo capitolo verranno analizzate le prestazioni della struttura dati nelle varie versioni sviluppate nel Capitolo 3 e confrontate tra loro.
Per analizzare le prestazioni durante l'esecuzione, è stato utilizzato un PC equipaggiato con processore Intel i7-8750H, 32GB di RAM e sistema operativo Windows 10. Per misurare il picco di memoria durante l'esecuzione, è stato utilizzato il programma "Process Explorer" della suite "Sysinternals" di Microsoft.

4.1 Descrizione dei dataset

Per eseguire i test di funzionamento sono stati utilizzati gli esempi giocattolo presenti in [2] e [3]. Successivamente per verificare le prestazioni con esempi più rappresentativi di casi di utilizzo reale, sono stati usati i dati utilizzati nell'analisi condotta dagli autori dello uno studio GAGE-B [7], nella quale viene fatta una comparazione prestazionale tra i metodi più utilizzati all'epoca dello scritto per eseguire l'assemblaggio di read. I dati da loro utilizzati sono liberamente disponibili in [8]. Il numero di read contenute nei singoli campioni è disponibile nella Tabella 4.1.
Gli algoritmi sviluppati per questa tesi non contemplano l'utilizzo di read con l'alfabeto esteso {A, C, G, T, N}, nella quale N indica un carattere che non è stato possibile identificare durante la fase di sequenziamento. Quindi prima di utilizzare i dati, questi sono stati modificati per poterli utilizzare, modificando le occorrenze di N in maniera casuale con un altro dei simboli dell'alfabeto {A, C, G, T}. Per rendere queste modifiche replicabili, prima della fase di modifica è stata utilizzata la funzione `srand` della libreria `random` in modo da poter ottenere le stesse modifiche ad ogni esecuzione.

Campione	Numero di read
A_hydrophila_HiSeq	13086404
B_cereus_HiSeq	24057408
B_fragilis_HiSeq	12322890
M_abscessus_HiSeq	5527852
R_sphaeroides_HiSeq	7731148
S_aureus_HiSeq	7609378
V_cholerae_HiSeq	3857294
X_axonopodis_HiSeq	11693464
B_cereus_MiSeq	2079628
M_abscessus_MiSeq	1989222
R_sphaeroides_MiSeq	1511694
V_cholerae_MiSeq	1579366

Tabella 4.1. Numero di read presenti nei campioni del dataset GAGE-B.

4.2 Confronto dei tempi di esecuzione

Per poter confrontare i due algoritmi, sono stati utilizzati 10 sottocampioni del sample *Vibrio cholerae CO 1032(5) - HiSeq* del dataset GAGE-B, organizzati in modo da avere il 10% delle read nel primo sottocampione, 20% nel secondo e così via fino all'ultimo che conteneva tutte le read del sample. Inoltre le read contenute in ogni sottocampione erano contenute in quello successivo di dimensioni maggiori.

I dati dei tempi di esecuzione sono presenti nella Tabella 4.2 in funzione di $||P||$ che rappresenta la somma delle lunghezze delle read in ingresso. Il dato che rappresenta l'analisi dell'intero campione con l'algoritmo di Cazaux-Rivals non è presente poiché l'esecuzione è stata interrotta dopo più di 1 ora e mezza. Nonostante la mancanza di questo dato, l'andamento asintotico dei restanti dati raccolti è sufficiente a fornire le giuste indicazioni.

Nella Figura 4.1, è riportato il grafico rappresentante i dati raccolti con anche il best fit per le due serie di dati, nel caso dell'algoritmo di Cazaux-Rivals, l'approssimazione è quadratica, mentre per l'algoritmo di Khan è lineare, come ci saremmo dovuti aspettare dall'analisi teorica presente nel Capitolo 2. Per effettuare il confronto diretto tra i due algoritmi, non sono stati considerate le procedure di inizializzazione dell'input come ad esempio la fase di ordinamento presente nella versione che utilizza l'algoritmo di Khan.

$\|P\|$	Algoritmo di Cazaux-Rivals [s]	Algoritmo di Khan [s]
35387093	12	**10**
70793265	41	**25**
106211348	95	**42**
141611804	239	**61**
177002660	411	**84**
212385500	956	**104**
247780622	1959	**128**
283181934	2846	**150**
318578443	3733	**173**
353980652	-	**195**

Tabella 4.2. Dati di confronto tra il tempo di esecuzione dell'algoritmo di Cazaux-Rivals e quello di Khan. Tempo riportato in secondi.

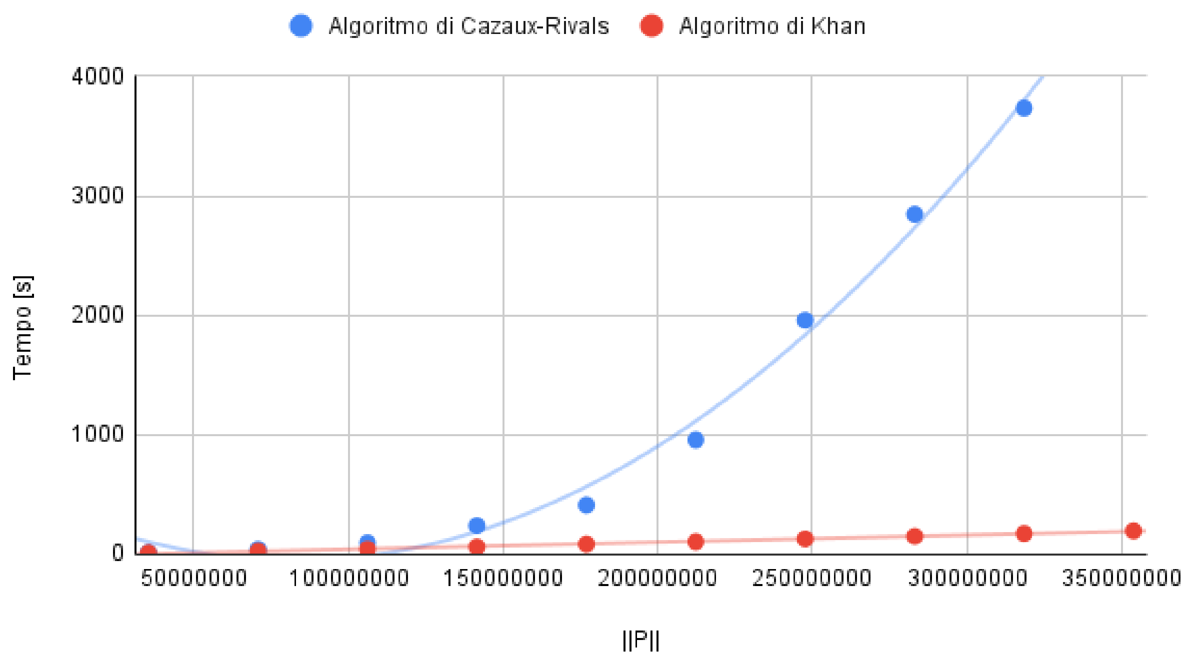

Figura 4.1. Grafico dei dati di confronto dei tempi di esecuzione dei due algoritmi, quello di Cazaux-Rivals e quello di Khan. Dati della Tabella 4.2.

4.3 Confronto dell'occupazione di memoria

Per poter confrontare la memoria occupata dalle due versioni implementate, quella descritta nella Sottosezione 3.1.2 e quella descritta nella Sottosezione 3.1.3, sono stati utilizzati 10 sottocampioni del sample *Staphylococcus aureus M0927 - HiSeq* del dataset GAGE-B, organizzati nello stesso modo descritto nel paragrafo precedente.

I picchi di memoria utilizzata per i sottocampioni sono presenti nella Tabella 4.3 in funzione di $||P||$ e sono riportati nel grafico presente nella Figura 4.2. Come possiamo notare l'andamento del consumo della memoria è lineare come previsto, quello che differisce è la pendenza che rappresenta la costante moltiplicativa nell'analisi asintotica. Si nota, inoltre, che l'occupazione della memoria per le due istanze di dimensione inferiore sono a vantaggio della versione descritta nella Sottosezione 3.1.2, a dimostrazione che le migliorie apportate dalla versione descritta nella Sottosezione 3.1.3 non sono efficaci con istanze "piccole".

$\|\|P\|\|$	Versione 2 [MB]	Versione 3 [MB]
58369010	**1996**	2217
116754480	**3337**	3487
175125699	4833	**4717**
233464242	6397	**5900**
291857569	8001	**7040**
350231931	9617	**8134**
408563454	11249	**9461**
466892922	12829	**10731**
525278457	14376	**11977**
583675928	15885	**13188**

.

Tabella 4.3. Dati di confronto del picco di occupazione di memoria delle due versioni della struttura dati implementate. La memoria è riportata in Megabyte.

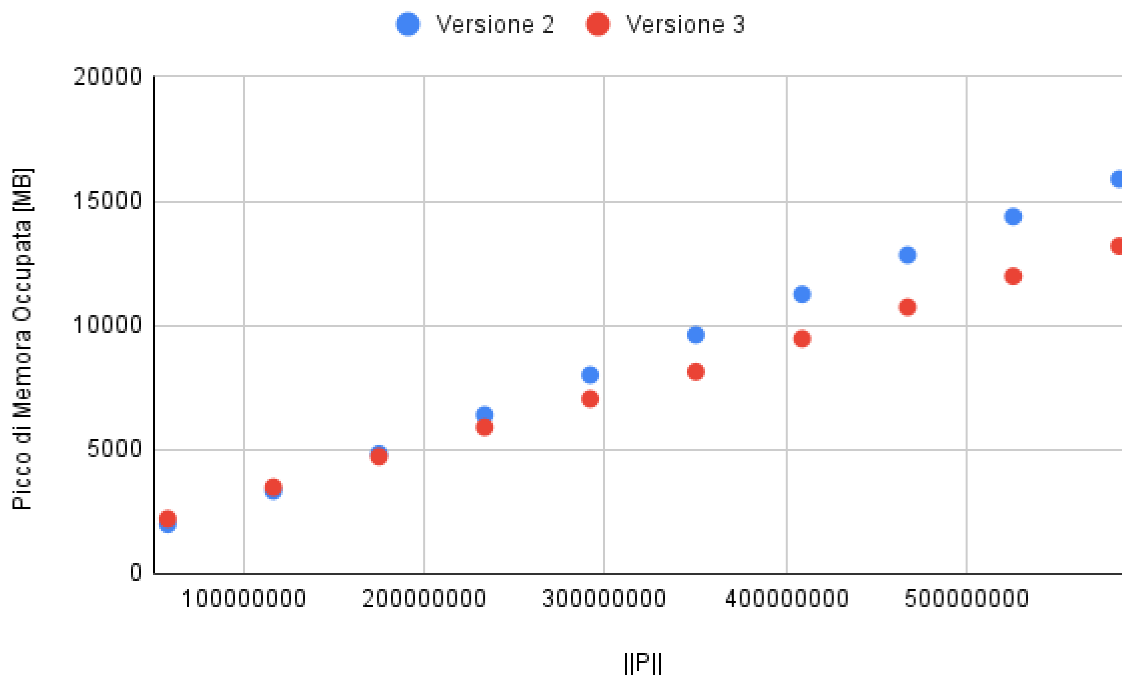

Figura 4.2. Grafico che mostra il confronto di occupazione della memoria tra le due versioni della struttura dati. Dati della Tabella 4.3.

4.4 Prestazioni con il Dataset GAGE-B

Per poter effettuare un'analisi delle prestazioni generali della struttura dati HOG con il dataset GAGE-B, è stata utilizzata la struttura dati basata sulla versione descritta nella Sottosezione 3.1.3 che utilizza l'algoritmo di Khan, poiché il suo inferiore utilizzo della memoria permette di eseguire i test su sample di dimensioni maggiori rispetto alle altre versioni. Per eseguire questo test sono stati utilizzati i seguenti sample: *Aeromonas hydrophila SSU - HiSeq, Mycobacterium abscessus 6G-0125-R - HiSeq, Rhodobacter sphaeroides 2.4.1 - MiSeq, Staphylococcus aureus M0927 - HiSeq* e *Xanthomonas axonopodis pv. Manihotis UA 323 - HiSeq*. Questi sample sono stati scelti poiché rappresentano input con dimensioni differenti come mostrato nella Tabella 4.1.

I dati riportati nella Tabella 4.4 confermano che l'occupazione di memoria ed il tempo di esecuzione sono lineari rispetto alla dimensione dell'input. L'aspetto da analizzare, invece, riguarda la dimensione della struttura dati intesa come numero di nodi del grafo. Nella fase iniziale, dopo aver inserito le read nel Trie, il numero di nodi è circa la metà di $||P||$ (il campione *R_sphaeroides_MiSeq* è quello che rispetta di meno questa tendenza). Successivamente, con la

trasformazione in EHOG, il grafo viene nuovamente notevolmente ridotto, comportando un notevole vantaggio in termini di memoria occupata per informazione rappresentata, che in questo caso sono le sovrapposizioni. La trasformazione in HOG, però, non prosegue questo trend, eliminando solo una manciata di nodi. Ricordiamo che nel Capitolo 2 è stato trattato il fatto che al tendere della dimensione dell'input all'infinito, lo HOG porta un vantaggio "infinito" rispetto all'EHOG. Nei casi testati, però, questo vantaggio è molto limitato.

	$\|\|P\|\|$	Numero nodi Aho-Corasick Automaton	Numero nodi EHOG	Numero nodi HOG	Tempo [s]	Picco uso memoria [MB]
R_sphaeroides_MiSeq	208260403	173587821	17953613	17951843	124	5626
M_abscessus_HiSeq	494140940	258623092	50503562	50502735	252	8607
S_aureus_HiSeq	742566976	363567603	204185434	204184582	737	18168
X_axonopodis_HiSeq	1098768274	616105720	268877767	268876714	1187	26108
A_hydrophila_HiSeq	1235961918	643132239	358477565	358476027	1572	27812

Tabella 4.4. Dati per l’analisi delle prestazioni dello HOG con dati reali

Capitolo 5
Conclusioni

In questa tesi è stata studiata la struttura dati Hierarchical Overlap Graph, proponendo due diverse implementazioni. I risultati sperimentali hanno confermato le prestazioni ricavate dalla teoria. Nello svolgimento della tesi sono emerse anche alcune idee che potrebbero utilmente essere esplorate per migliorare ulteriormente le prestazioni.

5.1 Miglioramenti Proposti

I miglioramenti alla struttura dati che potrebbero essere implementati riguardano principalmente la diminuzione di memoria utilizzata. La prima proposta riguarda la memorizzazione delle stringhe, in quanto trattandosi di materiale genetico composto da 4 tipi di nucleotidi rappresentati dai caratteri A, C, G e T, sarebbe possibile utilizzando una struttura dati apposita memorizzare ogni carattere con soli 2 bit al posto degli 8 utilizzati dai `char` o `string`. Questo permetterebbe di ridurre di circa un fattore 4 la memoria richiesta dalla memorizzazione delle read. La seconda proposta riguarda l'utilizzo di una nuova modalità di contrazione nel passaggio da automa di Aho-Corasick a EHOG e da EHOG a HOG, eliminando i nodi in-place senza la necessità di copiare i nodi marcati in una nuova struttura. La terza proposta riguarda una proposta per limitare l'uso della memoria occupata dalle liste R_l. Queste potrebbero, infatti, non essere memorizzate in maniera esplicita, bensì ricorrere ad una rappresentazione implicita considerando i failure link rovesciati. In questo modo si avrebbe un risparmio di memoria a discapito del tempo di computazione. Rimarrebbe da verificare la complessità temporale di questa implementazione. L'ultima proposta riguarda l'eventuale

possibilità di creare una struttura HOG efficiente che permetta l'aggiunta di read una volta terminata la sua costruzione.

5.2 Direzioni Future

La struttura dati sviluppata in questa tesi è limitata dal punto di vista delle applicazioni reali. Il primo passo per poter utilizzare questa tecnica in modo efficace, sarebbe quello di includere il suo utilizzo in una pipeline di elaborazione di materiale sequenziato. A questo punto si potrebbero verificare anche le sue prestazioni a confronto con le tecniche correntemente utilizzate. Un aspetto molto importante che nelle implementazioni che trovano utilizzi reali è la capacità di poter elaborare anche gli alfabeti estesi per la rappresentazione dei nucleotidi inseriti nella struttura dati come ad esempio quello formato da {A, C, G, T, N}.
Infine bisogna considerare l'aspetto delle read complementari invertite che vengono sequenziate ed inserite nel Trie. Queste sono strettamente collegate alle omologhe dirette, quindi sarebbe possibile sfruttare anche questa informazione aggiuntiva per migliorare il risultato finale.

Bibliografia

[1] Bastien Cazaux, Rodrigo Cánovas, Eric Rivals. Shortest DNA cyclic cover in compressed space. DCC: Data Compression Conference, Mar 2016, Snowbird, UT, United States. pp.536-545, ff10.1109/DCC.2016.79ff. fflirmm-01314330f

[2] Bastien Cazaux, Eric Rivals. Hierarchical Overlap Graph. Information Processing Letters, 2020, 155, pp.#105862. ff10.1016/j.ipl.2019.105862ff. fflirmm-01674319v2f

[3] Khan, S. (2021). Optimal Construction of Hierarchical Overlap Graphs. In P. Gawrychowski, & T. Starikovskaya (Eds.), 32nd Annual Symposium on Combinatorial Pattern Matching: CPM 2021 (pp. 17:1-17:11). (Leibniz International Proceedings in Informatics (LIPIcs); Vol. 191). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CPM.2021.17

[4] A Aho and M Corasick. Efficient string matching: an aid to bibliographic search. Comm ACM, 18:333–340, 1975.

[5] Rodrigo Canovas, Bastien Cazaux, and Eric Rivals. The Compressed Overlap Index. ´ CoRR, abs/1707.05613, 2017.

[6] https://www.boost.org/

[7] Magoc T, Pabinger S, Canzar S, Liu X, Su Q, Puiu D, Tallon LJ, Salzberg SL. GAGE-B: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics* 2013 Jul 15;29(14):1718-25. doi: 10.1093/bioinformatics/btt273. Epub 2013 May 10.

[8] http://ccb.jhu.edu/gage_b/datasets/index.html