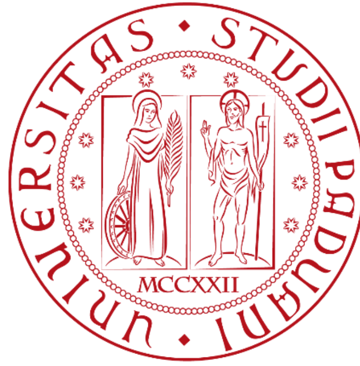


UNIVERSITÀ DEGLI STUDI DI PADOVA

Tesi di Laurea Magistrale in Ingegneria Informatica

COSTRUZIONE EFFICIENTE DI OVERLAP GRAPH MEDIANTE FM-INDEX PER IL METAGENOMIC BINNING

Laureando

ALBERTO RIZZARDI

Relatore

PROF. MATTEO COMIN

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

ANNO ACCADEMICO 2017 – 2018

Padova, 9 luglio 2018

Sommario

Il solo genoma umano conta circa 3,2 miliardi di paia di basi azotate. Benché tale numero appaia enorme, può essere solo una frazione delle basi che si trovano in un campione metagenomico, ovverosia un campione contenente più specie, come, ad esempio, un campione ambientale. Nel momento dunque in cui si va a trattare la metagenomica non è più possibile impiegare tecniche adatte singoli genomi per la ricomposizione del DNA, ma vanno ricercati algoritmi che raggruppino i frammenti delle diverse specie in cluster prima di poter ricomporre le varie sequenze genomiche. Tale operazione di clusterizzazzione è chiamata metagenomic binning, e per effettuarla è stato sviluppato MetaProb, un software molto efficiente dal punto di vista temporale, ma non altrettanto da quello spaziale. Nel progetto trattato in questa tesi si tenterà di ridurre la memoria RAM impiegata da MetaProb, cercando di mantenere un limite temporale non eccessivamente maggiore della versione originale. Per fare ciò verranno impiegate strutture dati avanzate, come il suffix array, il longest common prefixes, la Burrows Wheeler Transform e il FM-index.

Ringraziamenti

Dedico questa tesi ai miei meravigliosi genitori che mi hanno guidato in ogni istante della mia vita e supportato in ogni momento della mia carriera universitaria. Ringrazio il mio relatore, il Professor Comin, per avermi introdotto all'appassionante campo della Bioinformatica e per avermi aiutato nel corso del progetto. Ringrazio inoltre tutti i miei amici per aver sempre reso unica ogni giornata spesa insieme.

Indice

Introduzione **1**

Capitolo 1 Assembly e MetaProb **3**

1.1 Cosa è l'assembly 3

1.1.1 Le difficoltà dell'assembly 4

1.1.2 Risoluzioni dell'assembly: grafi di De Brujin e overlap-graphs 5

1.2 Cosa è il Metagenomic Binning 7

1.3 Cosa è MetaProb 8

1.3.1 Come funziona MetaProb 8

1.3.2 Generazione del grafo in MetaProb 10

1.3.3 Analisi del processo di generazione del grafo 12

1.3.4 Gestione paired-end read ed errori 13

Capitolo 2 Suffix-array, lcp, BWT e FM-index **15**

2.1 Suffix-array 15

2.1.1 Analisi del suffix array 17

2.1.2 Suffix array generalizzati 17

2.1.2 Costruzione del suffix array generalizzato 18

2.2 Longest common prefixes 19

2.3 BWT 19

2.3.1 Calcolo della BWT 21

2.3.2 La trasformata inversa 21

2.3.3 BWT su stringhe multiple 23

2.4 FM-index 24

2.4.1 Ricostruzione mediante FM-index 25

2.4.2 Ricerca di pattern mediante FM-index 25

2.4.3 Riduzione dello spazio impiegato 27

Capitolo 3 Il nuovo MetaProb **29**

3.1 Generazione del suffix array generalizzato 29

3.1.1 Inizializzazione 30

3.1.2 Induzione 31

3.1.3 Durata del processo 34

3.2 Generazione del longest common prefixes 35

3.3 Generazione BWT e FM-index 37

3.4 Ottimizzazione delle strutture dati 37

3.5 Gestione errori 39

3.6 Tecnica 1: grafo prefisso-suffisso ottenuto mediante FM-index ... 39
3.7 Tecnica 2: grafo prefisso-suffisso ottenuto mediante LCP ... 45
3.8 Tecnica 3: grafo matching massimali ottenuto mediante LCP ... 48
3.9 Tecnica 4: grafo di MetaProb (m q-mer) ottenuto mediante LCP ... 50
Capitolo 4 Risultati ... 51
4.1 Dataset impiegati negli esperimenti ... 51
4.2 Tempo e spazio di inizializzazione ... 52
4.3 Tempo e spazio per la costruzione del grafo ... 53
4.4 Tempo e spazio a fine processo ... 57
4.5 Precision, Recall e F-measure ... 59
4.6 Costruzione del suffix array per read corte e lunghe ... 62
4.7 Tempo e spazio per un dataset reale ... 62
Capitolo 5 Conclusioni ... 65
5.1 Considerazioni sui risultati e sul progetto ... 65
5.2 Eventuali lavori futuri ... 65
Appendice A Ottimizzazione Strutture dati ... 67
A.1 Array numerici ... 67
A.2 Stringhe ... 70
A.3 Suffix array ... 72
A.4 Array di conteggio ... 74
Bibliografia ... 77

Introduzione

L'assembly, ovverosia la ricomposizione del DNA a partire dalle sue read (letture frammentate) è sempre stato uno dei problemi fondamentali nella trattazione della bioinformatica. Benché si siano trovati strumenti estremamente efficienti per eseguirlo su campioni contenenti read appartenenti a una singola specie, questi perdono efficacia quando si prova ad applicarli direttamente a campioni metagenomici, ovverosia contenenti read provenienti da specie diverse. Si è dunque rivelato utile effettuare un raggruppamento delle read in cluster a seconda della specie di appartenenza prima della ricomposizione dei vari genomi. Tale processo, chiamato metagenomic binning, è stato implementato in MetaProb, un programma, sviluppato in C++ da Samuele Girotto presso l'Università degli Studi di Padova. Benché tale software operi in maniera estremamente veloce, dal punto di vista della memoria presenta dei consumi eccessivi, il che, per dataset di dimensione considerevole, lo rende inutilizzabile su dispositivi con poca memoria a disposizione. Lo scopo di questa tesi sarà dunque ridurre lo spazio impiegato per la memorizzazione dei vari dati durante il processo. Come vedremo, buona parte dell'eccessivo consumo di RAM nella versione originale di MetaProb è dovuta al calcolo delle strutture dati che verranno utilizzate per la costruzione di un grafo di adiacenza tra le read. Poiché le adiacenze verranno calcolate in base alla condivisione dei q-mer (segmenti lunghi q) tra le read, si è scelto di sfruttare delle strutture dati estremamente efficienti dal punto di vista del consumo della memoria della memoria e della rapidità. Queste strutture sono: il suffix array, il longest common prefixes (che nel nostro caso si è rivelato lo strumento più utile), la Burrows Wheeler Transform e, infine, il FM-index. Mediante queste strutture dati si è ottenuto un buon miglioramento della memoria, nonostante ciò sia stato accompagnato da un incremento del tempo impiegato (tenendosi comunque entro un limite accettabile).
Nel Capitolo §1 viene trattata una base di assembly, metagenomica, metagenomic binning e il funzionamento di MetaProb. Nel Capitolo §2 saranno descritte approfonditamente le nuove strutture dati impiegate nel nostro progetto. Nel Capitolo §3 saranno riportati gli algoritmi implementati per il calcolo dei grafi di adiacenza nella nuova versione di MetaProb. Nel Capitolo §4 si tratteranno i risultati ottenuti dalle nuove tecniche sviluppate, e verranno confrontati con quelli ottenuti dalla versione originale. Nel Capitolo §5 si ricaveranno delle conclusioni, proponendo eventuali progetti da sviluppare in futuro. In Appendice A è riportata una parte del codice implementato.

Capitolo 1

Assembly e MetaProb

Il problema dell'assembly è uno dei più trattati nell'ambito della Bioinformatica, e sotto alcuni aspetti esso può essere considerato il quesito antecedente a tutti gli altri: infatti, il concetto base dietro l'assembly è la ricostruzione del DNA originario partendo dalle sue *read* (letture), ovverosia il riassemblamento del DNA mediante le letture "spezzate". Senza questo riassemblamento del DNA tutti gli altri problemi derivati non potrebbero essere nemmeno considerati, il che pone così l'assembly al top per priorità.
Un'altra questione legata all'assembly è il binning di reads metagenomiche, che consiste nel raggruppare in cluster le specie presenti in un campione partendo dalle sue read, senza previo riassemblamento, consentendo dunque di effettuare la ricostruzione del DNA operando sui singoli cluster. Per questo problema è stato sviluppato *MetaProb*, un software che usa il conteggio dei *k*-mers (sottostringhe lunghe *k*) per il raggruppamento delle read in clusters. Benché MetaProb sia un software che presenta ottime prestazioni dal punto di vista temporale, non è invece ottimizzato dal punto di vista dell'uso nella memoria. Lo scopo del progetto trattato in questa tesi sarà dunque modificare MetaProb per ottenere una riduzione del consumo di RAM, tendando però di mantenere un buon tempo di esecuzione e la qualità dei risultati della versione originale. Nei seguenti paragrafi approfondiremo meglio l'assembly e le caratteristiche di MetaProb.
Essendo tutti questi problemi basati su un insieme di read, che altro non sono che stringhe su un alfabeto finito, è possibile trattarli non come quesiti biologici ma informatici, risolvendoli dunque mediante algoritmi e strutture dati ottimizzate.

1.1 Cosa è l'assembly

Quando avviene la lettura del DNA, che sia da una singola cellula o da un campione ambientale (metagenomica), la sequenza non viene letta interamente come un'unica stringa, bensì essa viene "spezzettata" in frammenti chiamati *read*, di lunghezza assai inferiore rispetto al genoma di partenza. Questa lunghezza, normalmente tra le 50 e le 1000 basi, varia a seconda della tecnologia impiegata; ad esempio le read ottenute dalla attuale tecnologia Illumina hanno lunghezza tra le 50 e le 150 basi [1]. Partendo da un singolo genoma e dividendolo in read sarebbe impossibile riassemblare la sequenza originale, e per tanto si sfrutta il fatto che una sequenza di DNA sia spesso presente più volte in un campione genetico (questo si indica con il concetto di *coverage*, che indica proprio quante volte una determinata sequenza sia presente in

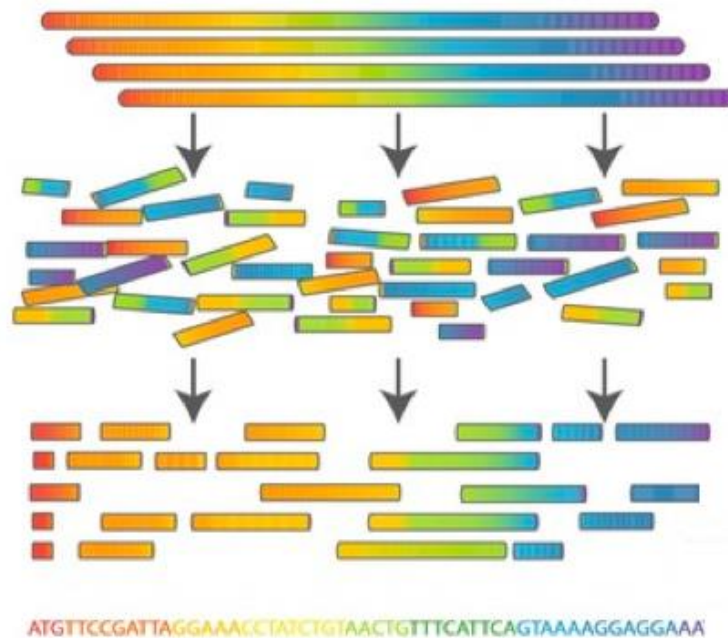


Figure 1.1. *Esempio del processo di ricostruzione dell'assembly*

un campione). Avendo più sequenze identiche spezzate in posizioni diverse si può osservare come read provenienti da sequenze diverse abbiano dei prefissi e dei suffissi in comune (Figura 1.1). Queste sezioni comuni sono dette *overlap*. Osservando dunque gli overlap tra i vari read si possono ricostruire le sequenze di partenza, ma non è facile trovare algoritmi efficienti per effettuare questo riassemblamento. La ricerca di tali algoritmi costituisce il problema dell'*assembly*.

1.1.1 Le difficoltà dell'assembly

Nonostante la breve introduzione precedente faccia apparire intuitive le strategie che sarebbe possibile utilizzare per la risoluzione dell'assembly, nella pratica non è un problema così banale, a causa di varie complicazioni:

- Frequentemente è possibile trovare mutazioni all'interno del DNA, ma non solo: possono essere presenti anche cancellazioni, inserimenti, e inversioni. Non è dunque sufficiente cercare overlap esatti, tuttavia la ricerca di overlap approssimanti aumenterebbe considerevolmente la durata del riassemblamento.
- Anche la tecnologia adottata per la lettura delle read non sempre funziona perfettamente: spesso si producono in output read che potrebbero essere state lette in maniera errata, o in alcuni casi viene associata a ogni base letta una misura di qualità (formato fastq).
- Un filamento di DNA è sempre composto da due sequenze complementari, e dunque, quando questo viene aperto, all'interno del campione saranno presenti read provenienti sia da uno strand che dall'altro. Queste letture sono dette *reverse-complement*, ovverosia

non solo sono il complementare dal punto di vista delle basi, ma sono anche “rovesciate” nel verso di lettura.

- Per ricostruire il DNA originario è necessario avere una coverage sufficiente, altrimenti il problema non è risolvibile.
- Il fatto che si osservi un overlap tra due read non assicura che queste siano effettivamente sovrapposte: spesso, infatti, sono presenti svariate ripetizioni nel DNA, le quali dunque non consentono di distinguere precisamente overlap esatti da overlap non esatti.

1.1.2 Risoluzioni dell’assembly: grafi di De Brujin e overlap-graphs

Partiamo dalla ricostruzione del DNA dato un campione contenente read provenienti da un solo individuo o da una sola specie. In generale per risolvere il problema dell’assembly vengono usate tecniche classificabili principalmente in due categorie: *Comparative assembly*, che sfrutta la presenza di genomi di riferimento, contenuti in un database, interrogabile mediante query e *De-Novo assembly*, che non usa sequenze di riferimento. La prima categoria può funzionare solo se sono presenti tutte le specie a cui appartengono le read nel database di riferimento, cosa che non sempre è vera, dato l’elevatissimo numero di specie sconosciute. Il Comparative assembly, inoltre, diventa problematico in presenza di errori nelle read e di ripetizioni, per non parlare del fatto che una stessa read può essere mappata su genomi diversi. L’altra categoria è basata sul cosiddetto De-Novo assembly: questa tecnica non assume nessuna conoscenza previa, ma tenta di ricostruire il DNA originario partendo solo dalle read a disposizione. Da adesso tratteremo solo il De-Novo assembly.

Collegare read tra di loro mediante l’overlap che condividono consente di generare un grafo. L’uso e lo studio di grafi per la risoluzione dell’assembly è dunque un campo molto studiato, che ha dato vita a due tecniche principali: tecniche basate su *overlap graph* [2], a cui è strettamente legato il concetto di *string graph*, e tecniche basate sui grafi di *De Brujin*. L’overlap graph usa come nodi le read e genera tra due nodi un arco solo se le read corrispondenti condividono un overlap maggiore di una certa lunghezza prefissata. Date dunque due read, r ed s, tali che $r = \alpha\beta$ e $s = \beta\gamma$, se l’overlap β è lungo a sufficienza, si può generare un arco tra le read r ed s. In questo caso β viene chiamato l’overlap dell’arco e α l‘estensione dell’arco. A questo punto, una volta costruito l’intero overlap graph, è possibile trovare il genoma originario individuando un percorso, attraverso i rami del grafo, che passi per tutti i nodi una e una sola volta, ovverosia un cammino hamiltoniano.

Una comoda esemplificazione dell’overlap graph è data dallo string graph [2]: questo viene ottenuto dall’overlap graph, ma eliminando alcuni archi detti *transitivi* (o *riducibili*): un arco $e_1 = (r, s)$ con estensione α è detto riducibile se esiste un altro arco $e_2 = (t, s)$ con estensione ω tale che ω è un suffisso di α. Alternativamente si può usare la definizione seguente, più intuitiva: date due read $r = \alpha\beta\gamma$ e $t = \gamma\delta\omega$, l’arco con overlap γ che connette le read r e t è

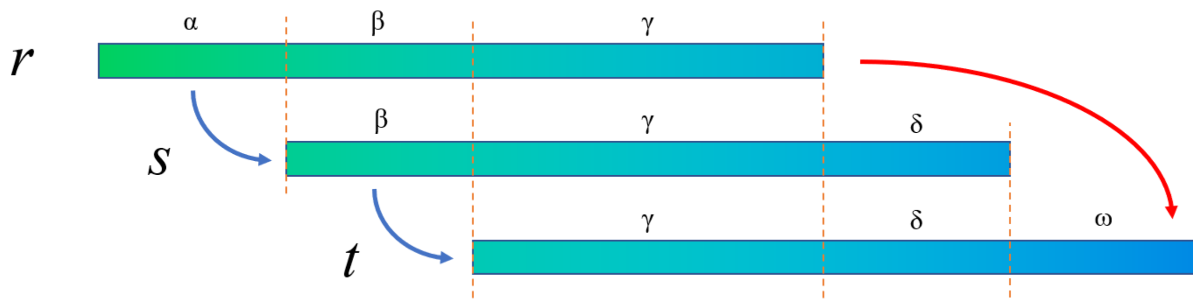


Figure 1.2. *Esempio di rimozione di archi riducibili (in rosso)*

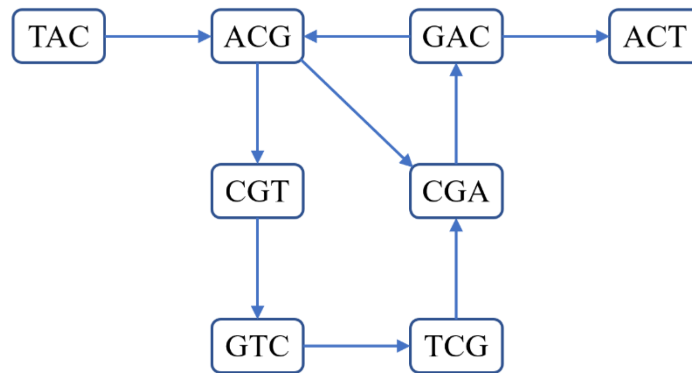


Figure 1.3. *Esempio di grafo di Brujin generato dalle read TACGA, CGTCG, CGACT*

detto riducibile se esiste una read *s* tale che $s = \beta\gamma\delta$. L'individuazione degli archi riducibili è chiamata *transitive detection*. Un esempio di tale riducibilità è riportato in Figura 1.2. La caratteristica principale di questi archi transitivi è che possono essere eliminati senza che venga perduta l'informazione necessaria alla risoluzione dell'assembly, rendendo dunque così più semplice il calcolo del cammino hamiltoniano.

Nonostante l'esemplificazione introdotta dallo string graph, trovare un cammino hamiltoniano resta un problema NP-hard, e per tanto si sono cercate soluzioni alternative. Una tecnica che dunque è stata adottata è quella dei grafi di De Brujin (o *k*-mer graphs). In questi grafi le read sono scomposte in tutti i loro *k*-mer (segmenti lunghi *k*) che saranno i nodi del grafo, e vengono usati come overlap soltanto i segmenti di lunghezza k - 1 di ogni nodo (che saranno sempre solo due). Dal grafo ottenuto si può ottenere la sequenza originaria individuando un percorso tra i nodi che passi una e una sola volta per ogni ramo, ovverosia un percorso Euleriano, la cui individuazione, diversamente dal percorso hamiltoniano, non è un problema NP-hard. Tuttavia, questo approccio genera alcuni problemi: il numero di nodi del grafo viene considerevolmente aumentato, e *k*-mer che appartengono a una stessa read sono ora collegati come fossero *k*-mer di read qualsiasi. Un esempio di grafi di De Brujin è riportato in Figura 1.3

Benché in MetaProb si tratti la Metagenomica, discussa nel paragrafo successivo, inizialmente verrà comunque generato un grafo di adiacenza tra le read. Abbiamo modificato il metodo precedente per la generazione di tale grafo (spiegato nel Paragrafo §1.3.2), utilizzando come

strumenti il *suffix array* [3], l'*FM-index* [4] e il *longest common prefix*, che saranno definiti nel capitolo §2. L'approccio usato nel nostro progetto si basa sull'overlap graph, in quanto non è necessario, per il nostro scopo, eliminare archi riducibili. Abbiamo inoltre sviluppato una tecnica che produce il medesimo grafo generato da MetaProb, usando cioè il numero di q-mer comuni, ma impiegando meno memoria. Infatti, lo scopo principale del nostro progetto è la diminuzione dell'uso di memoria RAM, cercando comunque di ottenere risultati simili al software precedente e di non estenderne troppo la durata temporale. La trattazione del progetto è rimandata al capitolo §3.

1.2 Cosa è il Metagenomic Binning

Un campo sempre più di interesse è la *metagenomica*. Con essa si intende lo studio di sequenze genomiche utilizzando non un insieme di read appartenenti a uno stesso individuo o ad una stessa specie, bensì usando un campione ambientale, contenente dunque un elevato numero di specie. Se, per esempio, lo scopo è il riassemblamento delle sequenze originali, allora una prima soluzione potrebbe essere l'utilizzo delle tecniche adottate per l'assembly di campioni contenenti singole specie, ottenendo in output le sequenze separate. Tuttavia, questo approccio effettuerà un elevato numero di confronti tra read non appartenenti a una stessa specie, deteriorando così le performance in termini di tempo. Le tecniche invece più frequentemente utilizzate sono basate su un previo raggruppamento delle read in cluster: ogni cluster contiene solo read appartenenti alla stessa specie, consentendo così un successivo riassemblamento nelle sequenze originali evitando confronti tra specie diverse. Questa classificazione di read è un processo chiamato *binning*, che consente di determinare quali e quante specie sono presenti in un determinato campione ambientale.

Svariate soluzioni sono state sviluppate per effettuare tale classificazione. Chiaramente tecniche brute-force in questa situazione non sarebbero applicabili, dato l'elevato numero di read che possono essere presenti in un campione. Le tecniche adottate principalmente si suddividono in due categorie [5]: tecniche *reference-based* (supervised), dove viene interrogato un database per scoprire l'origine di ogni read (come nel Comparative assembly) e tecniche *reference-free* (unsupervised). Per le tecniche *reference-based* dunque necessaria la presenza di database molto vasti contenenti tutte le specie presenti nel campione, cosa che nella pratica non succede mai, dato l'elevatissimo numero di specie ancora sconosciute, e pertanto questo metodo non può essere sempre usato. Esempi di software che usano questa tecnica sono: Mega illustrata in [6], Kraken in [7], Clark in [8] e MetaPhlan in [9]. L'altra tecnica utilizzata, basata su metodi *reference-free*, non usa conoscenze precedenti sulle read, ed è stata implementata in software come BiMeta descritta in [10], MetaCluster in [11] e [12], AbundanceBin [13] e CompostBin [14]. Tipicamente in questi software si cerca di raggruppare le read in clusters mediante misure di similarità.

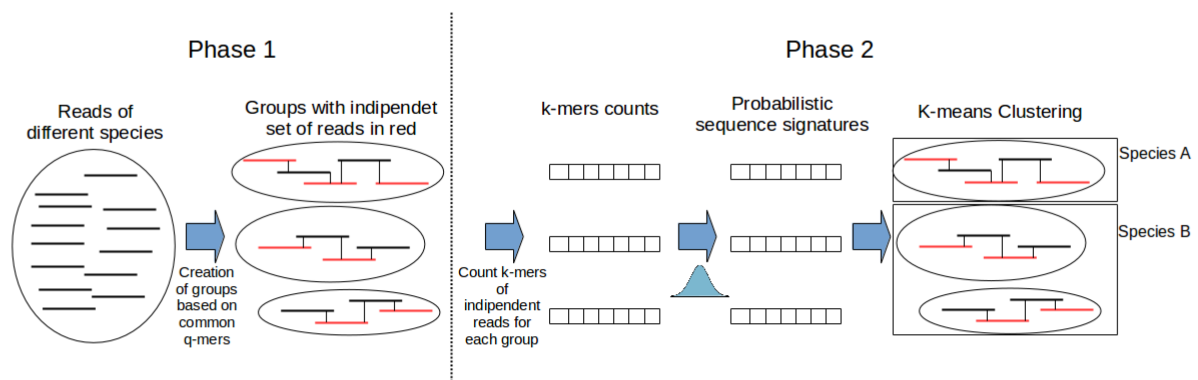


Figure 1.4. *Fasi operative di MetaProb [5]: nella prima si osserva la generazione delle adiacenze tra read sulla base dei q-mer comuni e la formazione dei gruppi, nella seconda il raggruppamento in cluster sulla base della probabilistic sequence signature ottenuta dai k-mer*

1.3 Cosa è MetaProb

MetaProb [5] è software creato da Samuele Girotto presso l'Università degli studi di Padova per il metagenomic binning. Metaprob raggruppa le read sulla base sia dei q-mer comuni che sulla distribuzione dei k-mer (q e k sono due parametri che vedremo meglio nel paragrafo §1.3.1). Differentemente da altri software per il binning, MetaProb mantiene buone performance anche in presenza di campioni contenenti specie presenti in maniera impari. Inoltre, MetaProb gestisce agevolmente anche read provenienti da diverse tecnologie di sequenziamento, sia corte (100 bp) che lunghe (700 bp).

1.3.1 Come funziona MetaProb

In MetaProb il processo di raggruppamento è diviso in due fasi, come riportato in Figura 1.4. Nella prima fase, si generano le adiacenze tra le read in base al numero di q-mer condivisi. Se le read condividono almeno m q-mer, allora è possibile considerarle adiacenti, e all'adiacenza viene assegnato uno score. È possibile settare i parametri m e q quando viene avviato il programma. Chiaramente questa operazione genera un grafo, ed il suo calcolo è la parte più dispendiosa nel programma dal punto di vista della memoria, e sarà pertanto la modifica di tale processo l'elemento focale del nostro progetto. Il processo dettagliato di questo calcolo verrà descritto nel paragrafo §1.3.2. A questo punto inizia l'effettivo raggruppamento nella prima fase, in cui si unificano le read sulla base dello score di lati condivisi: partendo dalla prima read, i suoi lati sono inseriti all'interno della frontiera dell'insieme, e viene selezionata la read con adiacenza maggiore sulla frontiera. Questa read sarà dunque aggiunta al gruppo e le sue adiacenze dirette verso read non appartenenti a nessun altro gruppo andranno aggiunte alla frontiera (se sulla frontiera è presente già un'adiacenza diretta verso una stessa read, allora il suo score verrà sommato allo score della nuova adiacenza). Le adiacenze sulla frontiera che vanno dalla read appena aggiunta verso read già contenute nel gruppo vengono ora eliminate dalla frontiera. Il processo viene poi ripetuto finché la frontiera non è vuota o non si si raggiunge

un numero di basi nel gruppo maggiore di *T*. Questo parametro è stato aggiunto per limitare il danno prodotto dagli errori presenti nel sequenziamento in quanto potrebbe portare al raggruppamento di read provenienti da organismi differenti. Anche il parametro *T* è settabile all'avvio del programma. Poiché successivamente i gruppi saranno clusterizzati sulla base del conteggio dei *k*-mer, durante la l'espansione di un gruppo viene selezionato un sottoinsieme di read dal quale prelevare i *k*-mer. Infatti, se fossero usate tutte le read presenti all'interno del gruppo, si avrebbe una considerevole sovrastima dei *k*-mer, in quanto le read contengono molte sezioni comuni. Pertanto, MetaProb seleziona un sottoinsieme di read indipendenti per il calcolo dei *k*-mer. Siccome il calcolo del *maximum indipendent set* di un grafo è un problema NP-hard, la generazione dell'indipendent set è stata implementata secondo la seguente modalità: quando viene selezionata una read *x* da aggiungere al gruppo *G*, si osserva se questa condivide adiacenze con le read presenti nell'indipendent set $I(G)$; se non sono presenti adiacenze allora *x* viene aggiunta a $I(G)$.
Durante la seconda fase i gruppi vengono accorpati secondo il seguente criterio *alignment-free*: dato un set *X* di read e data la i-esima read X^i, dove $i = 1..|M|$ ed M è la dimensione dell'input set, si definisce X_w^i il numero di volte che il *k*-mer *w* appare all'interno di X^i. Poiché le read provengono da entrambi gli strand del genoma, in X_w^i viene tenuto conto anche del contributo derivato dal reverse complement di w. Se la lunghezza del *k*-mer è piccola rispetto alla lunghezza delle read (qui verificato poiché MetaProb usa $k = 4$), allora si può considerare X_w^i come una variabile di Bernoulli. Dato l'indipendent set $I(G)$ si può definire X_w^G come la frequenza con cui il *k*-mer *w* appare nel gruppo $I(G)$, e dunque $X_w^G = \sum_{i=1}^{g} X_w^i$, dove g è la dimensione di $I(G)$. Usando queste variabili MetaProb calcola la media μ_w^G e la varianza σ_w^G, con le quali viene standardizzata la variabile X_w^G mediante la formula:

$$\tilde{X}_w^G = \frac{X_w^G - \mu_w^G}{\sigma_w^G} \quad . \tag{1.1}$$

Infine, si usa $\tilde{X}_w^G$ per derivare la *probabilistic sequence signature* f_w^G:

$$f_w^G = \frac{\tilde{X}_w^G}{\sqrt[2]{\sum_{w \in \Sigma^k} (\tilde{X}_w^G)^2}} \quad , \tag{1.2}$$

HASH TABLE

		Q-MER	SEQUENZE
Seq. 1	AGTCGGTG	AGTC	Seq. 1
		GTCG	Seq. 1 | Seq. 2
		TCGG	Seq. 1 | Seq. 2
		CGGT	Seq. 1
Seq. 2	GGTCGGAG	GGTG	Seq. 1
		GGTC	Seq. 2
		CGGA	Seq. 2
		GGAG	Seq. 2

Figure 1.5. *Esempio di Hash Table ottenuta mediante i q-mer*

che sarà il vettore utilizzato come misura per calcolare la distanza tra i gruppi.

È possibile a questo punto usare la metrica impostata per calcolare la distanza tra i gruppi: per esempio, per read corte converrà usare **-feature 1** (NORM_D2star_All_Read_Prob_Kmer + Euclidian norm), mentre per read lunghe useremo **-feature 2** (NORM_D2star_Group_Prob_Bernulli_Kmer + Euclidian norm). I dettagli sulle variabili e le metriche usate sono riportati nel paper originale [5]. Questa metrica sarà dunque usata da k-means per calcolare i cluster nel caso sia ne sia fornito il numero, altrimenti viene utilizzata una variante di G-means che, anziché usare la *Anderson-Darling statistic*, usa il *two-sample Kolmogorov-Smirnov test*. Rimandiamo sempre alla lettura del paper originale [5] per i dettagli sul funzionamento di G-means in MetaProb, mentre analizzeremo ora la parte dedicata alla costruzione iniziale del grafo, che verrà successivamente modificata per il nostro progetto.

1.3.2 Generazione del grafo in MetaProb

La costruzione del grafo di adiacenza delle read in Metaprob è divisa in due parti: la prima parte scorre linearmente tutte le read, generando per ogni q-mer che incontra un elemento da inserire nella mappa non ordinata (in C++ unordered_map); questo elemento è costituito da un *hash* rappresentante il q-mer, e da una lista di *nodi* contenenti le sequenze che contengono il q-mer. Dunque, quando si preleva un q-mer da una read, se l'elemento contenente il corrispettivo hash, non è ancora stato creato, allora viene generato un nuovo elemento con hash e un unico nodo, altrimenti, viene semplicemente annesso nuovo nodo contenente la sequenza alla lista di nodi dell'elemento. Se poi sono già presenti sia l'hash che la sequenza, significa che il q-mer è presente più volte nella stessa read, e sarà incrementato il campo del nodo che indica il numero di volte che il q-mer appare in una determinata sequenza. Un esempio di tale processo è riportato in Figura 1.5. Come valore di default per i q-mer MetaProb usa $q = 30$. Tuttavia, è possibile impostarlo fino a un massimo di 32: infatti, il tipo **hash_type** utilizzato per memorizzare l'hash

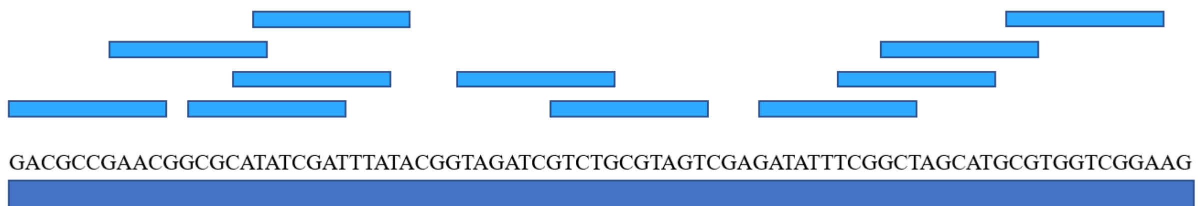
Algoritmo 1.1

```
01. unordered_map<hash_type, nodes> MapHash ← insieme delle coppie hash-nodi
02. set reads ← set contenente le read
03. for each r ∈ reads
04. {
05.     set hashes ← r.getHashes();
06.     unordered_map<seq, score> mapAdj;
07.     for each h ∈ hashes
08.     {
09.         nodes Nodi ← MapHash[h];
10.         num h_in_r ← N.get(r).getAppearences();
11.         for each n ∈ Nodi
12.         {
13.             seq s = n.getSeq();
14.             if r != s
15.                 mapAdj[s] ← mapAdj[s] + (h_in_r * n.getAppearences());
16.         }
17.     }
18.     for each t ∈ mapAdj
19.         if t.getValue() >= m
20.             r.addAdjVertex(t.getKey());
21. }
```

in un elemento è un unsigned long di 64 bit, e, codificando ogni base con 2 bit, si ottiene una lunghezza dei q-mer al massimo di 32.

Nella seconda fase si scorrono le read, dalle quali si riottengono i q-mer, e quindi gli hash che contiene, e per ognuna si osservano i nodi adiacenti nella mappa, verso i quali viene dunque generata un'adiacenza temporanea, memorizzata in *mapAdj,* in base al numero di volte che questi q-mer sono presenti nelle read. Se sono presenti almeno m q-mer comuni, allora viene generata l'adiacenza definitiva come riportato nell'Algoritmo 1.1 (riga 19). Il tipo **unordered_map** contiene coppie (*key*, *value*): o si ottiene il valore di *value* accedendo mediante la *key* (ad esempio *mappa*[*key*]), o, ciclando su tutti gli elementi, per ognuno dei quali si possono ottenere *key* e *value* mediante i metodi ***getKey***() e ***getValue***(). *MapHash* rappresenta la mappa ottenuta nella fase precedente, contenente come chiavi gli hash dei q-mer e come valori i nodi che contengono le sequenze in cui il q-mer compare e il numero di volte che compaiono. La variabile *reads* di tipo **set** contiene la lista delle read. **Nodes** è il tipo di *Nodi*, che rappresenta i nodi contenenti le sequenze, a cui si accede mediante ***getSeq***(), e il numero di volte che presentano in essi un determinato q-mer (la chiave hash corrispondente), a cui si accede con ***getAppearences***(). Alla riga 10 si può osservare che è possibile ottenere un nodo mediante ***get***(**seq s**) su *Nodi* (in particolare, in questo caso si fa per ottenere il numero di volte che l'hash h appare nella read r di partenza). La mappa *mapAdj* contiene la mappa di adiacenze temporanee, usando come chiave le sequenze (nel programma contiene gli ID delle sequenze, in questa spiegazione è stata semplificata), e come valori il numero di q-mer condivisi con la

Single-end read



Paired-end read

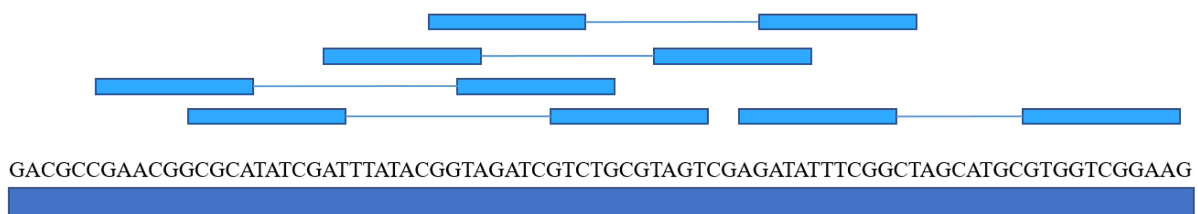


Figure 1.6. *Differenza tra single-end read e paired-end read*

sequenza identificata dalla chiave. Se alla fine una read t ha un'adiacenza temporanea maggiore di m, allora alla read r viene aggiunta come adiacenza t.

1.3.3 Analisi del processo di generazione del grafo

Dal punto di vista temporale, per la generazione del grafo, sono poche le tecniche efficienti come quella impiegata in MetaProb: inizialmente vengono salvati in *MapHash* gli hash q-mer ottenuti scorrendo linearmente l'insieme delle read, e pertanto, chiamando N il numero di read ed M la taglia di ognuna di esse (assumeremo per l'analisi che tutte le stringhe abbiano la stessa lunghezza), il tempo impiegato è $O(N(M - q + 1))$, poiché ogni read contiene $M - q + 1$ q-mer. Questo può essere arrotondato a $O(NM)$, dato che, come già detto nel paragrafo precedente, q è sempre minore di 32. Vengono poi generate le adiacenze ripercorrendo linearmente le read in input e osservando i relativi q-mer in *MapHash*. Poiché ogni q-mer può generare un numero di adiacenze temporanee diverse (queste corrispondo alla condivisione di un solo q-mer, mentre le adiacenze definitive vengono generate solo quando sono condivisi almeno m q-mer), chiameremo a_p il numero totale di lati parziali generati, e possiamo dunque definire il tempo impiegato in questa fase come $O(NM + a_p)$. Includendo anche il tempo della parte precedente ($O(NM)$), otteniamo comunque un tempo totale che è $O(NM + a_p)$.
A livello di memoria, invece, il discorso diventa più complesso: l'impiego di strutture dati come **unordered_map** non consente di stabilire esattamente quanti bit vengono occupati; approssimativamente possiamo assumere che ogni q-mer venga salvato separatamente tramite

hash in *MapHash*, la quale dunque contiene *NM* hash (ipotesi ovviamente estrema, dato che equivale ad assumere che non esistano q-mer condivisi), e che siano presenti in tutto anche *NM* nodi, contenenti ognuno un indice di sequenza e un riferimento al nodo successivo. Ricordiamo che gli *NM* hash occupano ognuno 64 bit, che gli indici delle sequenze occupano ognuno 32 bit (anche se sarebbe stato meglio impiegarne 64, nel caso fossero presenti più di 2^{32} sequenze) e che ogni riferimento al nodo successivo, che per l'analisi assumiamo essere un indirizzo di memoria da 64 bit, consuma 64 bit; dunque la memoria impiegata dagli hash è NM*64, mentre quella impiegata dai nodi è $NM*(32 + 64) = NM*96$ bit. In tutto dunque sono usati almeno $NM*(64 + 96) = NM*160$ bit, che equivalgono a NM*20 byte, impiegando così circa 20 byte per ogni base (un po' meno in realtà considerando anche che abbiamo approssimato $N(M - q + 1)$ a NM). A questo si aggiunge il fatto che le strutture impiegate (come l'**unordered_map**) usano a loro volta una certa quantità di memoria non facilmente descrivibile. Nel nostro progetto si tenterà dunque di migliorare l'impiego di RAM, usando soprattutto strutture dati più semplici (spesso array) in modo da ottenere un migliore controllo sullo spazio che consumano e che poi andranno a liberare quando non saranno più impiegate.

1.3.4 Gestione paired-end read ed errori

Le read generate dall'analisi di un campione metagenomico possono essere memorizzate in un singolo file fasta o fastq, nel caso siano di tipo single-end, oppure in due file fasta o fastq se vengono usate tecnologie di tipo paired-end. In quest' ultimo caso, chiamando l'insieme di read del primo file S e del secondo file T, sappiamo che le read $s_i \in S$ e $t_i \in T$ provengono dalla stessa sequenza di DNA, come esemplificato in Figura 1.6. Benché questo non ci dia informazioni sula distanza tra le due read o su quali siano le read consecutive, ci permette di stabilire, nel caso del metagenomic binnig, che, se le read appartengono a due gruppi distinti dopo la prima fase, possiamo unificare i gruppi dato che contengono due read certamente provenienti dallo stesso genoma.

In MetaProb viene usata la classe **clsSequencePE** per rappresentare una sequenza, e le sue istanze (non le singole read) verranno usate per costituire i gruppi.

MetaProb mette a disposizione tre modalità operative nel caso di read paired-end, impostabili all'avvio del programma nella variabile **TypeGraph**:

- *Single*: ogni read è salvata in un singolo oggetto **clsSequencePE**, non tenendo così conto dell'accoppiamento tra read paired-end, trattando ogni read come una read single-end.
- *Paired*: le istanze di **clsSequencePE** contengono ognuna due read paired-end, e, nella costruzione dei gruppi, durante la seconda fase, si tiene conto si tiene conto degli hash provenienti da entrambe le read paired. Poiché gli elementi dei gruppi sono formati dalle singole istanze di **clsSequencePE**, due read paired-end saranno sempre nello stesso gruppo.

- *SingleUnion*: ogni read è salvata in un singolo oggetto clsSequencePE (come nella modalità *single*), però dopo la creazione dei gruppi, questi vengono uniti sulla base delle read paired-end che contengono.

Per la gestione degli errori Metaprob opera in modo molto semplice: se un q-mer proveniente da una read contenente lettere diverse da A, C, G e T, allora il suo hash viene semplicemente ignorato.

Capitolo 2

Suffix-array, lcp, BWT e FM-index

Il sistema di generazione del grafo in MetaProb è efficiente dal punto di vista temporale, in quanto tutti i cicli per trovare i lati e generare le adiacenze sono lineari nella dimensione dell'input, ma dal punto di vista temporale rischia di occupare troppo spazio memorizzando da un lato gli hash e dall'altro un riferimento alle read che presentano un determinato hash. Lo scopo del nostro progetto è la riduzione della memoria impiegata, anche sacrificando un po' di tempo. I metodi che impiegheremo sono stati derivati mediante l'uso di strutture dati favorevoli alla ricerca di sezioni comuni tra stringhe [15] [16]. In questo capitolo verranno dunque introdotte le strutture dati impiegate e la loro funzione all'interno del nostro progetto, in particolare vedremo come sono state adattate nel caso della ricerca su read metagenomiche.

2.1 Suffix-array

Le strutture dati per la ricerca su stringhe sono uno dei problemi più analizzati sia dal punto di vista storico che dal punto di vista funzionale. Quando, per esempio, si effettua la ricerca di una determinata parola all'interno di una pagina web, è molto probabile non venga effettuato il confronto con ogni singolo vocabolo presente nel testo, ma che prima sia stata generata una struttura dati sul testo in cui effettuare la ricerca. Una delle prime strutture dati che si studia (benché non venga poi usata nella pratica) è il *suffix trie*: questo si ottiene generando un albero con radice, dove vengono riportati tutti i suffissi, partendo dalla radice, e dove ogni nodo indica una lettera di un suffisso. Se più suffissi condividono i loro prefissi, allora ognuno di questi sarà riportato sullo stesso ramo, e quando questi si differenzieranno sarà presente una biforcazione. La ricerca di un pattern lungo m su un testo lungo n (su cui è stato costruito il suffix trie) può essere effettuata in tempo $O(m)$. Tuttavia, il numero di nodi del suffix trie è quadratico nella dimensione dell'input, e dunque $O(n^2)$. Ciò rende l'uso del suffix trie impraticabile nella realtà. Un esempio di suffix trie è riportato in Figura 2.1. Alla fine di ogni parola viene posto inoltre il carattere *$*, che rappresenta un carattere piccolo di tutti gli altri (oltre che la fine della stringa).

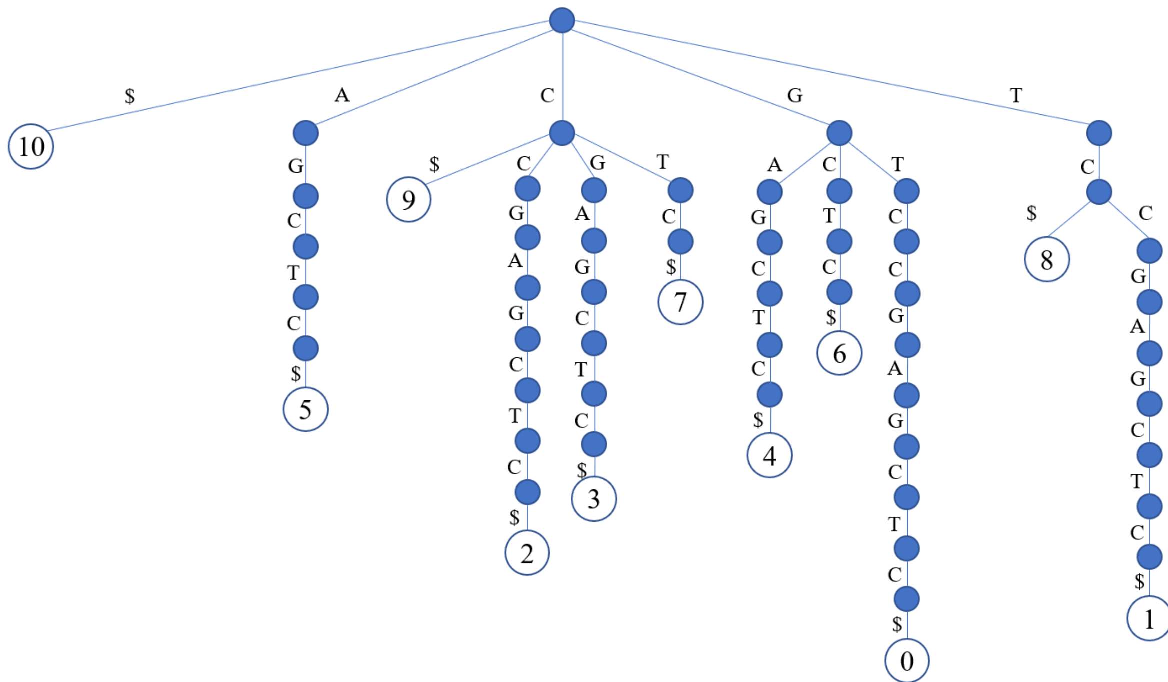


Figure 2.1. *Esempio di suffix trie calcolato sulla stringa GTCCGAGCTC$*

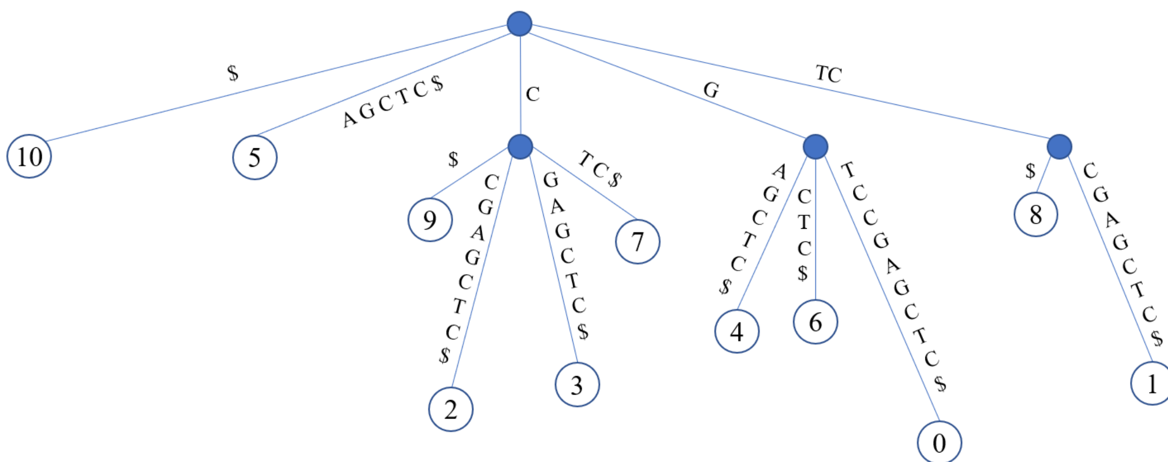


Figure 2.2. *Esempio di suffix tree calcolato sulla stringa GTCCGAGCTC$*

in quanto i nodi consecutivi che non presentano biforcazioni sono unificati in un unico nodo, dove la stringa che lo rappresenta può essere dedotta semplicemente osservando l'indice di inizio del prefisso riportato nelle foglie dell'albero. Un esempio di suffix tree è riportato in Figura 2.2. La ricerca di un pattern lungo m su un testo lungo n (su cui è stato precedentemente costruito il suffix tree) può essere effettuata nuovamente in tempo $O(m)$. Lo spazio impiegato invece è notevolmente ridotto: si può dimostrare che è lineare nella grandezza del testo in input, e dunque $O(n)$. Tuttavia, ciò che la notazione O-grande nasconde, è la costante: nel caso dei suffix tree lo spazio impiegato infatti è circa $20{*}n$, e dunque sarebbe inutilizzabile nella pratica (specialmente per l'analisi di campioni metagenomici contenenti un gran numero di read). Pertanto, è stata introdotta una nuova struttura dati molto efficiente, sia in termini di tempo che di spazio: il *suffix array*.

	INDICE	SUFFISSO		SUFFIX ARRAY	SUFFISSO
	0	ACGCCGA$		7	$
	1	CGCCGA$		6	A$
	2	GCCGA$		0	ACGCCGA$
ACGCCGA$ →	3	CCGA$	→	3	CCGA$
	4	CGA$		4	CGA$
	5	GA$		1	CGCCGA$
	6	A$		5	GA$
	7	$		2	GCCGA$

Figure 2.3. *Esempio di suffix array calcolato sulla stringa ACGCCGA$*

2.1.1 Analisi del suffix array

Il suffix array [3] è uno strumento concettualmente semplice da capire: data una stringa t, il suo suffix array si ottiene prima generando tutti i suoi possibili suffissi, e poi ordinandoli in odine alfabetico. Ciò che viene memorizzato però non sono i suffissi (che impiegherebbero spazio $O(n^2)$), ma l'indice di inizio del suffisso (e dunque $O(n)$). Un esempio di suffix array è riportato in Figura 2.3.

Per ottimizzare la ricerca all'interno del testo da cui viene derivato il suffix array, sarà necessario affiancarlo a un'altra struttura dati, il *longest comon prefixes* (*lcp*). Una versione di questa struttura che utilizzeremo nel nostro progetto, diversa da quella originariamente descritta [3], sarà illustrata nel paragrafo §2.2. Il tempo richiesto per la ricerca è dunque $O(m + \log n)$, simile a come avviene in generale con qualunque ricerca in un dizionario ordinato. È possibile ottimizzare la ricerca all'interno del suffix array [17] per ottenere tempo di ricerca $O(m)$. In particolare, vedremo nel capitolo §2.4 un'altra struttura dati derivata dal suffix array che effettuerà in tempo lineare la ricerca di un pattern in un testo [4], e che sarà impiegata nel nostro progetto.

Il suffix array può essere costruito in tempo $O(n \log n)$. Benché non sia tempo lineare, vedremo che, per la funzione che ne faremo, e per l'uso dei *suffix array generalizzati*, la componente logaritmica potrà essere quasi considerata una costante.

2.1.2 Suffix array generalizzati

Si parla di *suffix array generalizzati* quando ci si riferisce a suffix array costruiti non su una sola stringa, ma su un set di k stringhe $S = \{S^0, S^1, \ldots, S^{k-1}\}$ (che saranno le read nel caso della metagenomica), su un alfabeto Σ, ognuna delle quali finisce con il carattere *$*. La differenza principale con il suffix array tradizionale è la possibilità che due stringhe presentino due suffissi

AGGA$
ACGA$
GAGT$

→

INDICE (STRINGA, SUFFISSO)	SUFFISSO
(0, 0)	AGGA$
(0, 1)	GGA$
(0, 2)	GA$
(0, 3)	A$
(0, 4)	$
(1, 0)	ACGA$
(1, 1)	CGA$
(1, 2)	GA$
(1, 3)	A$
(1, 4)	$
(2, 0)	GAGT$
(2, 1)	AGT$
(2, 2)	GT$
(2, 3)	T$
(2, 4)	$

→

SUFFIX ARRAY (STRINGA, SUFFISSO)	SUFFISSO
(0, 4)	$
(1, 4)	$
(2, 4)	$
(0, 3)	A$
(1, 3)	A$
(1, 0)	ACGA$
(0, 0)	AGGA$
(2, 1)	AGT$
(1, 1)	CGA$
(0, 2)	GA$
(1, 2)	GA$
(2, 0)	GAGT$
(0, 1)	GGA$
(2, 2)	GT$
(2, 3)	T$

Figure 2.4. *Esempio di suffix array calcolato sulle stringhe AGGA$, ACGA$ e GAGT$*

uguali. Per risolvere questa ambiguità osserveremo che l'algoritmo proposto per la costruzione del suffix array generalizzato ordinerà automaticamente i suffissi uguali in base all'ordine dalle stringhe di appartenenza. Dunque, se definiamo i caratteri $\$^0$, $\$^1$, …, $\$^{k-1}$ posti ognuno rispettivamente al termine delle stringhe $S^0, S^1, \ldots, S^{k-1}$, è come se alfabeticamente $\$^0$ precedesse $\$^1$, $\$^1$ precedesse $\$^2$ e così via fino a $\$^{k-1}$, e dunque (da adesso verrà usato il simbolo < per indicare il concetto di precedenza) $\$^0 < \$^1 < \ldots < \$^{k-1}$. In più non è sufficiente utilizzare l'indice di inizio per indicare un suffisso, ma si userà la combinazione dell'indice i della stringa S^i e dell'indice di inizio suffisso. Un esempio di suffix array generalizzato è riportato in Figura 2.4.

2.1.2 Costruzione del suffix array generalizzato

La costruzione del suffix array generalizzato non è un problema banale: per riuscire a costruirlo in tempo ottimo vengono utilizzate più strutture dati, che verranno definite nel capitolo §3. La cosa rilevante di questa costruzione è che impiegherà tempo $O(N \log n)$, dove N è il totale dei caratteri dell'input set (se l'input è composto da t stringhe, tutte della stessa lunghezza n, allora $N = t*n$) ed n rappresenta la stringa di lunghezza massima. Se dunque consideriamo che le read hanno tutte lunghezze simili, e, al variare delle tecnologie, comprese tra 50 bp e 1000 bp, il termine log n diventa quasi una costante, e sarà minore in presenza di read corte. A livello di spazio, per la creazione del suffix array generalizzato, sarà invece impiegato spazio lineare $3*N$ in generale (i dettagli si vedranno nel capitolo §3). Infine, poiché come avviene nel caso del suffix array singolo, in cui vengono memorizzati solo gli indici, anche qui si memorizzano solo

LCP	SUFFISSO
0	$
1	$
1	$
0	A$
2	A$
1	ACGA$
1	AGGA$
2	AGT$
0	CGA$
0	GA$
3	GA$
2	GAGT$
1	GGA$
1	GT$
0	T$

Figure 2.5. *Esempio lcp derivato dal suffix array calcolato sulle stringhe AGGA$, ACGA$ e GAGT$*

le coppie (indice suffisso, indice stringa), e si può dunque considerare nuovamente lo spazio di memoria occupato, per la memorizzazione del solo suffix array generalizzato, uguale a N.

2.2 Longest common prefixes

Il concetto di *longest common prefixes* introdotto originariamente [3] era adattato alla ricerca su una lista ordinata di stringhe (il suffix array); il *lcp* conteneva informazioni riguardanti il prefisso comune con il limite superiore e il limite inferiore della ricerca a ogni passo della bisezione.

Il *lcp* usato invece in questo progetto è una versione più moderna, e in parte più semplice: data una serie di stringhe ordinate $X = \{X^0, X^1, \dots, X^{k-1}\}$, si definisce *lcp* l'array dove ogni elemento contiene il prefisso più lungo condiviso tra le stringhe consecutive. Dunque, *lcp*[0] è posto uguale a 0, mentre *lcp*[*i*] è posto uguale alla massima lunghezza del prefisso condiviso tra X^{i-1} e X^i, con $i = 1, \dots, k-1$. Tale informazione risulterà la più fruttuosa nel nostro progetto, come sarà visibile nel Capitolo §4. Un esempio è riportato in Figura 2.5. L'algoritmo di calcolo sarà presentato nel capitolo §3.

2.3 BWT

La *Burrows-Wheeler Transform* (*BWT*) è una struttura dati inizialmente utilizzata per la compressione di testi, il cui funzionamento è il seguente: dato un testo, vengono generate tutte le sue rotazioni, le quali saranno poi ordinate alfabeticamente, ottenendo così la *Burrows-*

final char (*L*)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Figure 2.6. *Estratto dal paper originale [18] che evidenzia la compressibilità della BWT*

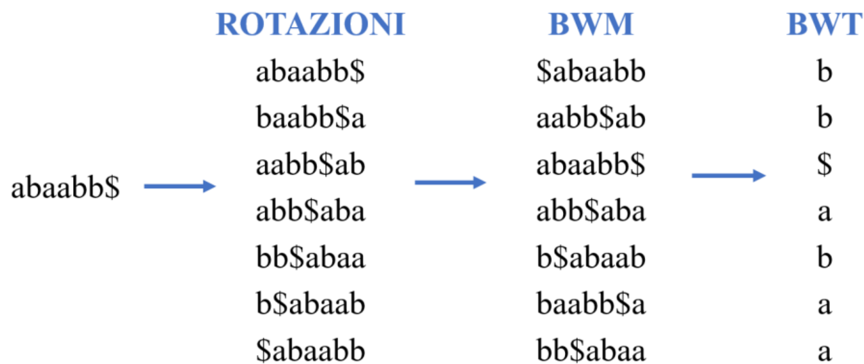


Figure 2.7. *Calcolo della BWT sulla stringa abaabb$*

Wheeler Matrix (*BWM*), dalla quale verrà prelevata l'ultima colonna. Un esempio, estratto dal paper originale [18], dove viene calcolata la *BWT* proprio sul testo del paper stesso, è riportato in Figura 2.6. Come si può osservare, l'ultima colonna presenta consecutivamente caratteri uguali, il che rende la *BWT* estremamente efficiente per la compressione di testi. Ciò è dovuto al fatto che spesso un determinato carattere è preceduto da caratteri simili (in Figura 2.6 il carattere *n* è spesso preceduto dalle vocali). Un altro esempio, che verrà usato anche per mostrare la reversibilità della BWT, è visibile in figura 2.7, dove viene calcolata la *BWT* della stringa *abaabb$*. Continueremo ad assumere che le stringhe terminino con il carattere *$*, che sarà utile per l'inversione della *BWT*.

BWT	SA SUFFISSI
$abaabb	$
aabb$ab	aabb$
abaabb$	abaabb$
abb$aba	abb$
b$abaab	b$
baabb$a	baabb$
bb$abaa	bb$

Figure 2.8. *Analogia presente tra BWT e suffix array*

2.3.1 Calcolo della BWT

La *BWT* può essere ottenuta, come già detto, generando tutte le possibili rotazioni, ordinandole alfabeticamente, e infine prelevando l'ultima colonna. Chiaramente, questo metodo brute-force è estremamente inefficiente. Una tecnica invece più utilizzata sfrutta il suffix array: si può infatti osservare che i prefissi dell'ordinamento delle rotazioni, fino al carattere *$*, sono equivalenti ai suffissi derivati dal suffix array, come evidenziato in Figura 2.8. Pertanto, data una stringa *r* (qui trattata come un array di caratteri *r*) su cui viene calcolato il suffix array SA, e poiché l'ultima colonna è composta semplicemente dai caratteri precedenti gli elementi della prima colonna, si può ottenere ogni singolo carattere della BWT nel seguente modo: $BWT[i] = r[k - 1]$, dove $SA[i] = k$, se $k > 0$, o $BWT[i] = \$$ se $k = 0$.

2.3.2 La trasformata inversa

Una proprietà fondamentale della *BWT* è la sua invertibilità. Per calcolare la stringa originale, si utilizza la proprietà L-F (*last-first*) della *BWT*: i caratteri uguali presenti nell'ultima colonna sono ordinati allo stesso modo dei corrispondenti caratteri nella prima colonna, come è possibile osservare nella Figura 2.9*a*. Associando a ogni carattere dell'ultima colonna un indice che indica quanti caratteri uguali sono presenti precedentemente (questa notazione sarà detta *ranking*), e riportandolo anche su tutti gli altri elementi della *BWM,* è possibile evidenziare questa corrispondenza tra caratteri della prima e dell'ultima colonna. Definiremo ora il motivo per cui hanno lo stesso ordinamento, usando come esempio il carattere *a* della Figura 2.9*a*: nella prima colonna, le *a* sono ordinate ognuna in base alle stringhe che le seguono; dunque a_0 è ordinato in base a *abb$ab*, a_1 in base a *baabb$* e a_2 in base a *bb$aba.* Nell'ultima colonna è possibile osservare che le colonne sono ordinate allo stesso modo; infatti, in ordine rotazionale, i caratteri a_0, a_1, a_2 e a_3 sono seguite dalle stesse stringhe che seguono i rispettivi caratteri nella prima colonna.

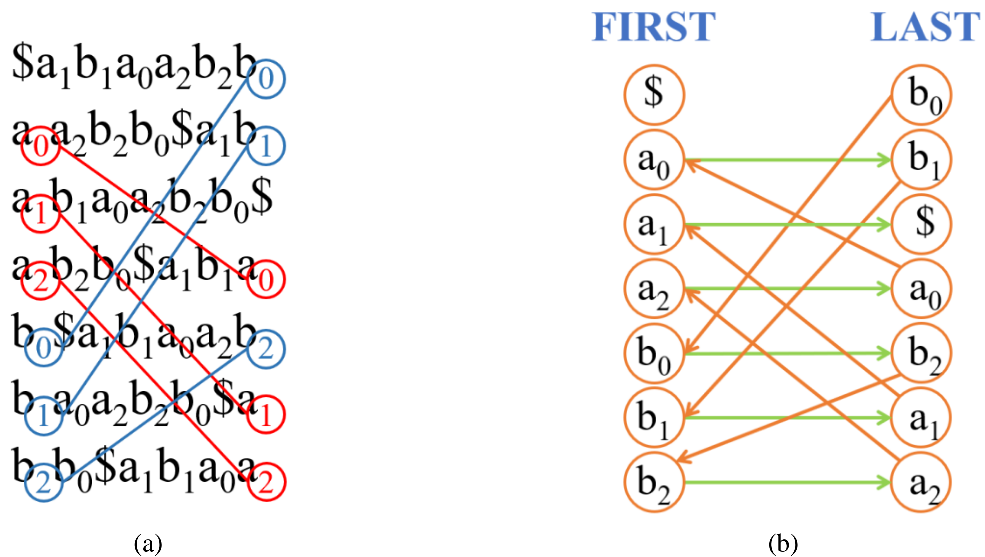


Figure 2.9. *In* (a) *è evidenziata la relazione che intercorre tra i ranking della prima e dell'ultima colonna, in* (b) *è rappresentato il processo di ricostruzione della stringa originale partendo dalla sua BWT*

Definita la proprietà *L-F* passiamo ora a illustrare le strutture richieste per la ricostruzione della stringa originaria: sono necessarie la *BWT*, ovverosia l'ultima colonna delle *BWM*, e la prima colonna. Per ottenere quest'ultima è sufficiente osservare che la *BWT* presenta tutti e soli i caratteri presenti nella prima colonna, e che dunque è sufficiente ordinare alfabeticamente la *BWT* per ottenere la prima colonna. Un'altra struttura necessaria è il ranking dei caratteri. Per ora possiamo immaginare di memorizzarlo in due array, uno per la prima colonna e uno per l'ultima, ma vedremo che, mediante l'FM-index, lo spazio richiesto per memorizzare quest'informazione e la prima colonna sarà estremamente ridotto.

Il procedimento è effettuato a ritroso, ovverosia si parte dall'ultimo carattere della stringa, che è il primo della *BWT*, b_0 in Figura 2.9*b*. Ovviamente sarà preceduto da *\$*, che viene dunque già annesso alla stringa da ricostruire, e a seguire si annette b_0. Si sa che il carattere nella stringa che lo precede sarà collocato nella *BWT* nella stessa posizione in cui il carattere di partenza è collocato nella prima colonna, in quanto la *BWT* è formata dai caratteri che precedono gli elementi della prima colonna. In questo caso, il carattere che precede b_0 sarà b_2, che viene annesso a sua volta alla stringa da ricostruire. Il procedimento viene ripetuto per b_2: si cerca la sua posizione nella prima colonna e si trova il carattere nella corrispondente posizione nella BWT, che sarà l'elemento che precederà b_2. In generale, dato l'elemento c_k in posizione *i* della prima colonna, $FC[i] = c_k$ (FC indica First Column), e l'elemento che lo precede nella stringa originaria sarà $BWT[i] = d_j$. Il processo termina quando si trova il carattere *\$* (non si include nella ricostruzione), e la stringa trovata sarà l'inverso della stringa originaria, nell'esempio \$*bbaaba*. È sufficiente invertirla per ottenere la stringa originaria *abaabb\$*.

AGGA$
ACGA$
GAGT$

SUFFISSO	SA	SUFFISSO	BWT
$AGGA	(4, 0)	$	A
$ACGA	(4, 1)	$	A
$GAGT	(4, 2)	$	T
A$AGG	(3, 0)	A$	G
A$ACG	(3, 1)	A$	G
ACGA$	(0, 1)	ACGA$	$
AGGA$	(0, 0)	AGGA$	$
AGT$G	(1, 2)	AGT$	G
CGA$A	(1, 1)	CGA$	A
GA$AG	(2, 0)	GA$	G
GA$AC	(2, 1)	GA$	C
GAGT$	(0, 2)	GAGT$	$
GGA$A	(1, 0)	GGA$	A
GT$GA	(2, 2)	GT$	A
T$GAG	(3, 2)	T$	G

Figure 2.10. *Esempio di calcolo della BWT sulle stringhe AGGA$, ACGA$ e GAGT$ e analogia con il suffix array*

2.3.3 BWT su stringhe multiple

La prima differenza che si può notare e che le stringhe della *BWM* non sono più ordinate in modo strettamente alfabetico: infatti, sono ordinate fino al carattere *$*, dopodiché seguono l'ordine del set con cui sono state create. Ciò garantisce comunque il corretto funzionamento della *BWT*, ed è integrabile con il concetto espresso precedentemente di generalized suffix array. Infatti, il suffix array può essere usato per derivare la *BWT* (come evidenziato in Figura 2.10) allo stesso modo del caso di *BWT* su una singola stringa, e dunque si può ricavare con il seguente metodo: definendo il suffix array *SA* come l'array composto dalle coppie di elementi (indice suffisso, indice stringa) calcolato sul set di stringhe $\{r_0, r_1, \ldots, r_{n-1}\}$, possiamo definire *BWT* come l'array dove ogni elemento $BWT[i] = r_j[k - 1]$, dove $SA[i] = (j, k)$, se $k > 0$, o $BWT[i] = \$$ se $k = 0$.
Una proprietà fondamentale che viene mantenuta è la reversibiltà della *BWT*: assumendo che la *BWT* sia ottenuta da un set di *n-1* stringhe, è possibile, è possibile ricostruire la *i*-esima stringa partendo dall'i-esimo elemento della BWT, considerando anche, come nel caso di BWT su stringa singola, il suo ranking. L'unico elemento per cui non resta valida la proprietà di ricostruzione è l'ultimo carattere nella costruzione, che sarà sempre il carattere $, ma non con il ranking della stringa originale: in Figura 2.11 è evidenziato il processo di ricostruzione della seconda stringa (dunque con ID 1), e si può osservare che ogni carattere lungo il percorso appartiene alla stringa originale, a parte l'ultimo *$*, con ranking 0 anziché 1. Ciò comunque non influenza il processo di ricostruzione, in quanto viene interrotto in ogni caso appena si osserva

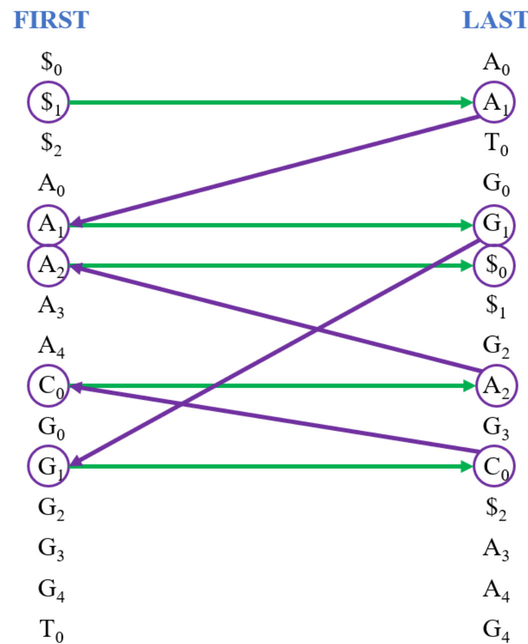


Figure 2.11. *Esempio di ricostruzione della seconda stringa (ACGA$) dalla BWT calcolata su stringhe multiple*

il carattere *$*, indipendentemente dal suo ranking (ricordiamo poi che questo non sarà incluso nella ricostruzione). Alla fine, si ottiene l'inverso della stringa originale, quindi è sufficiente invertirla per ottenere la stringa corretta.

2.4 FM-index

L'ultima struttura dati che analizzeremo è l'*FM-index*; la definizione che daremo è un po' diversa da quella proposta originariamente [4], poiché è stata adattata sia alla metagenomica che al nostro progetto. L'FM-index non è altro che la *BWT* e il suffix array affiancati ad altre due strutture: la prima è composta da una variabile per ogni lettera dell'alfabeto di riferimento *Σ* (in questo caso *$*, *A*, *C*, *G* e *T*, dunque 5) ognuna delle quali riporta il conteggio di tutti i caratteri precedenti o uguali alla lettera di riferimento. In Figura 2.12 è riportato un esempio di tali variabili (*totS* indica *tot$*, ma per la nomenclatura delle variabili in C++ il *$* è stato sostituito con *S*). L'altra struttura necessaria è composta da un array per ogni lettera dell'alfabeto di riferimento *Σ*. Questi array, che chiameremo array di conteggio, indicano, per ogni posizione, quanti caratteri uguali al carattere dell'array di riferimento sono stati trovati fino a quella posizione nella *BWT*, ovverosia il ranking di ogni carattere nel relativo array. Queste strutture dati aggiunti potevano essere racchiuse in delle matrici anziché in delle singole variabili, tuttavia, per evidenziare il legame con le singole variabili, abbiamo preferito lasciare gli array con nomi significativi.

Benché possa sembrare più convenente memorizzare il ranking in un singolo vettore anziché in cinque variabili e cinque vettori diversi, vedremo che è possibile ottimizzare tali strutture per ridurre notevolmente lo spazio impiegato. Poiché le operazioni sull'FM-index calcolato su

totS = 3 (3*S*)
totSA = 8 (3*S*+5*A*)
totSAC = 9 (3*S*+5*A*+1*C*)
totSACG = 14 (3*S*+5*A*+1*C*+5*G*)
totSACGT = 15 (3*S*+5*A*+1*C*+5*G*+1*T*)

SA	BWT	CountS	CountA	CountC	CountG	CountT
(4, 0)	A	0	1	0	0	0
(4, 1)	A	0	2	0	0	0
(4, 2)	T	0	2	0	0	1
(3, 0)	G	0	2	0	1	1
(3, 1)	G	0	2	0	2	1
(0, 1)	$	1	2	0	2	1
(0, 0)	$	2	2	0	2	1
(1, 2)	G	2	2	0	3	1
(1, 1)	A	2	3	0	3	1
(2, 0)	G	2	3	0	4	1
(2, 1)	C	2	3	1	4	1
(0, 2)	$	3	3	1	4	1
(1, 0)	A	3	4	1	4	1
(2, 2)	A	3	5	1	4	1
(3, 2)	G	3	5	1	5	1

Figure 2.12. *Esempio di FM-index calcolato sulle stringhe AGGA$, ACGA$ e GAGT$*

singola stringa e su stringhe multiple seguono lo stesso procedimento, li descriveremo entrambi riferendoci genericamente all'FM-index.

2.4.1 Ricostruzione mediante FM-index

Avendo a disposizione gli array di conteggio, è facile determinare il ranking di un determinato carattere: è sufficiente osservare il valore dell'elemento nella stessa posizione al relativo array di conteggio. Dunque, se per esempio $BWT[i] = C$, allora il suo ranking sarà $rank = countC[i]$. Poiché quest'informazione viene utilizzata per trovare la posizione di tale carattere nella prima colonna della Burrows-Wheeler Matrix, per completare tale ricerca è sufficiente sommare al ranking la somma delle lettere alfabeticamente precedenti. Dunque, continuando l'esempio precedente, la formula completa per ottenere la posizione è $j = countC[i] + totSA$.

2.4.2 Ricerca di pattern mediante FM-index

L'applicazione più interessante dell'FM-index è il tempo impiegato per la ricerca di un pattern all'interno del testo: il tempo impiegato ottenere in output il numero di ricorrenze di un pattern è $O(|pattern|)$, e a seguire la listatura di tali ricorrenze richiede tempo $O(|numero\ ricorrenze|)$. La ricerca è effettuata nel seguente modo: a ogni passaggio, si tengono memorizzati la posizione iniziale (*start*) e finale (*end*) all'interno della Burrows Wheeler Matrix, nella quale, siccome le stringhe sono presenti in ordine alfabetico, le ricorrenze del pattern saranno consecutive. All'inizio si pongono $start = 0$ e $end = totSACGT - 1$ (massimo indice possibile). Dato un pattern $P = p_0p_1...p_{m-1}$, si parte dall'ultimo carattere, e si procede a ritroso: a ogni passaggio si

INDICE	SUFFISSO	CountG
0	$\$_0AGGA_0$	0
1	$\$_1ACGA_1$	0
2	$\$_2GAGT_0$	0
3	$A_0\$AGG_0$	1
4	$A_1\$ACG_1$	2
5	$A_2CGA\$_0$	2
6	$A_3GGA\$_1$	2
7	$A_4GT\$G_2$	3
8	$C_0GA\$A_2$	3
9	$G_0A\$AG_3$	4
10	$G_1A\$AC_0$	4
11	$G_2AGT\$_2$	4
12	$G_3GA\$A_3$	4
13	$G_4T\$GA_4$	4
14	$T_0\$GAG_4$	5

(a)

INDICE	SUFFISSO	CountA
0	$\$_0AGGA_0$	1
1	$\$_1ACGA_1$	2
2	$\$_2GAGT_0$	2
3	$A_0\$AGG_0$	2
4	$A_1\$ACG_1$	2
5	$A_2CGA\$_0$	2
6	$A_3GGA\$_1$	2
7	$A_4GT\$G_2$	2
8	$C_0GA\$A_2$	3
9	$G_0A\$AG_3$	3
10	$G_1A\$AC_0$	3
11	$G_2AGT\$_2$	3
12	$G_3GA\$A_3$	4
13	$G_4T\$GA_4$	5
14	$T_0\$GAG_4$	5

(b)

SA	INDICE	SUFFISSO
(4, 0)	0	$\$_0AGGA_0$
(4, 1)	1	$\$_1ACGA_1$
(4, 2)	2	$\$_2GAGT_0$
(3, 0)	3	$A_0\$AGG_0$
(3, 1)	4	$A_1\$ACG_1$
(0, 1)	5	$A_2CGA\$_0$
(0, 0)	6	$A_3GGA\$_1$
(1, 2)	7	$A_4GT\$G_2$
(1, 1)	8	$C_0GA\$A_2$
(2, 0)	9	$G_0A\$AG_3$
(2, 1)	10	$G_1A\$AC_0$
(0, 2)	11	$G_2AGT\$_2$
(1, 0)	12	$G_3GA\$A_3$
(2, 2)	13	$G_4T\$GA_4$
(3, 2)	14	$T_0\$GAG_4$

(c)

Figure 2.13. *Esempio di ricerca del pattern AG sul FM-index*

cercano le ricorrenze del carattere p_i tra le posizioni *start* ed *end* nella BWT, ma poiché questi saranno consecutivi, si possono ottenere semplicemente sottraendo l'elemento in posizione *start-1* della matrice di conteggio dello stesso carattere all'elemento in posizione *end* della medesima matrice. Ciò fornisce il numero di ricorrenze del suffisso di P che comincia in p_i all'interno della BWT. Per ottenere i nuovi start ed end è sufficiente usare nuovamente la matrice di conteggio nel seguente modo:

$$\begin{cases} start = tot < caratteri\ precedenti\ p_i > + count < p_i > [start - 1] & \text{se } start > 0 \\ start = tot < caratteri\ precedenti\ p_i > & \text{altrimenti} \end{cases}$$

$$end = tot < caratteri\ precedenti\ p_i > + count < p_i > [end] - 1$$

Poiché ognuna di queste operazioni richiede tempo $O(1)$, e vengono effettuate |*pattern*| volte, il tempo totale impiegato è $O(|pattern|)$. Un esempio è riportato in Figura 2.13. Gli elementi utilizzati, ed effettivamente memorizzati, sono evidenziati in verde, gli altri servono a far comprendere meglio il funzionamento del processo. I valori iniziali sono:

$$pattern = "AG"$$

$$start = 0$$

$$end = totSACGT - 1 = 14$$

La prima fase (Figura 2.13*a*) inizia cercando il carattere *G*, individuando le nuove posizioni *start* ed *end* come riportate di seguito:

$$start = totSAC = 9$$

$$end = totSAC + countG[end] - 1 = 9 + countG[14] - 1 = 9 + 5 - 1 = 13$$

La procedura viene ripetuta identica (Figura 2.13*b*) per il secondo (e ultimo) carattere, *A*:

$$start = totS + countA[start - 1] = 3 + countA[9 - 1] = 3 + countA[8] = 3 + 3 = 6$$

$$end = totS + countA[end] - 1 = 3 + countA[13] - 1 = 3 + 5 - 1 = 7$$

Infine, per il recupero dell'indice delle stringhe contenenti il pattern e per la sua posizione all'interno di tali stringhe, è sufficiente ciclare dalla posizione *start* a *end* nel suffix array, motivo per cui è richiesto solo *O*(|*numero ricorrenze*|) tempo; nell'esempio in figura 2.13*c* il pattern *AG* si trova nella stringa 0 in posizione 0 e nella stringa 2 in posizione 1.
Nel caso fosse stata cercato un pattern non presente nelle stringhe di riferimento, l'algoritmo restituisce un valore di *end* minore del valore di *start*, Permettendo così di identificare l'assenza di tale pattern.

2.4.3 Riduzione dello spazio impiegato

Gli array di conteggio possono sembrare un'addizione considerevole allo spazio impiegato; tuttavia, è possibile memorizzare solo una frazione dell'array: partendo dal primo elemento, e memorizzando un valore ogni *k*, è possibile frazionare di *k* lo spazio di memoria impiegato dai vettori di conteggio. Un esempio di ciò è riportato in Figura 2.14, dove *k* = 5. Gli elementi memorizzati nell'indice *i* di un array di conteggio ridotto corrisponderà all'elemento in posizione *i*k* dell'array di conteggio completo. Inversamente, un elemento dell'array completo in posizione *j* sarà l'elemento presente in posizione *j/k* solo se *j* è divisibile per *k*. Quando viene richiesto un elemento in posizione *j* dell'array di conteggio completo di un certo carattere *E* che non è presente nell'array ridotto, è sufficiente partire dal primo elemento inferiore disponibile (ottenuto tenendo la parte intera di *j/k*), e sommare tutte le ricorrenze (n_E) del carattere trovate nella BWT nelle posizioni dal primo intero *i* divisibile per *k* (escluso) fino a *j* (incluso). L'aggiunta di tale spazio mantiene comunque la linearità del tempo di ricerca di un pattern, poiché le operazioni per recuperare il ranking di un elemento sono comunque effettuate in tempo costante.
In generale dunque, chiamando c_E l'array di conteggio completo riferito alla lettera *E* e <u>countE</u> la sua versione ridotta, si ottiene un generico valore in posizione *j* mediante la formula:

$$c_E[j] = countE\left[\left\lfloor\frac{j}{k}\right\rfloor\right] + n_E \quad . \tag{2.1}$$

ID	SA	BWT	CountS		CountA		CountC		CountG		CountT	
0	(4, 0)	A	0	0	1	1	0	0	0	0	0	0
1	(4, 1)	A	0	1	2	2	0	0	0	2	0	1
2	(4, 2)	T	0	2	2	3	0	1	0	4	1	1
3	(3, 0)	G	0		2		0		1		1	
4	(3, 1)	G	0		2		0		2		1	
5	(0, 1)	$	1		2		0		2		1	
6	(0, 0)	$	2		2		0		2		1	
7	(1, 2)	G	2		2		0		3		1	
8	(1, 1)	A	2		3		0		3		1	
9	(2, 0)	G	2		3		0		4		1	
10	(2, 1)	C	2		3		1		4		1	
11	(0, 2)	$	3		3		1		4		1	
12	(1, 0)	A	3		4		1		4		1	
13	(2, 2)	A	3		5		1		4		1	
14	(3, 2)	G	3		5		1		5		1	

Figure 2.14. *Esempio di riduzione dello spazio degli array di conteggio*

Per esempio (sempre seguendo la Figura 2.14), se richiedo l'elemento in posizione $j = 8$ dell'array completo c_G, poiché j non è divisibile per $k = 5$, questo valore non sarà immediatamente disponibile. Allora si ottiene il valore inferiore immediatamente prima di 8 che sia divisibile per 5, cioè $i=5$, e si conta quante G sono presenti nella BWT da i (escluso) a j (incluso). In questo caso è presente una G in posizione 7, dunque il valore dell'array completo in posizione 8, definendo come *countG* la versione ridotta, è

$$c_G[8] = countG\left[\left\lfloor\frac{8}{5}\right\rfloor\right] + 1 = 2 + 1 = 3 \quad . \tag{2.2}$$

Capitolo 3

Il nuovo MetaProb

Come già accennato, il nostro progetto consiste nel modificare la parte iniziale di MetaProb, dedicata alla generazione del grafo che verrà utilizzato per l'assemblamento delle read in gruppi. I grafi che utilizzeremo conterranno adiacenze classificabili secondo tre categorie diverse: nella prima categorie i lati del grafo sono determinati in base agli overlap massimi delle coppie suffisso-prefisso delle read, nella seconda si ottengono le adiacenze mediante matching massimali, che sono semplicemente la più lunga sottostringa condivisa tra due stringhe (senza dunque il vincolo di essere prefissi-suffissi), e la terza categoria corrisponde allo stesso identico grafo presente in MetaProb (condivisione di m q-mer). Queste tre tipologie di grafi saranno ottenute tramite le quattro nuove tecniche che abbiamo implementato: la prima (paragrafo §3.6) e la seconda (paragrafo §3.7) otterranno la prima categoria di grafo mediante FM-index e *lcp*, la terza (paragrafo §3.8) otterrà la seconda categoria mediante *lcp*, e la quarta produrrà la terza tipologia di grafo, sempre mediante *lcp*. Tutti questi metodi per la generazione di tale grafo useranno una parte delle stesse strutture dati (il suffix array generalizzato), mentre successivamente percorreranno strade diverse. Pertanto, per cominciare, illustreremo gli algoritmi implementati per ottenere tali strutture dati.

3.1 Generazione del suffix array generalizzato

La parte più importante della nostra implementazione è il suffix array generalizzato, non solo perché è la struttura dati che verrà utilizzata per generare tutte le altre, ma anche perché sarà il punto di picco di consumo della memoria nel nostro progetto. L'algoritmo utilizzato, per nulla banale, è un riadattamento dell'algoritmo proposto originariamente [19] per la generazione di suffix array generalizzati: tale algoritmo si divide in due fasi: una di inizializzazione e una di induzione. Solo la prima fase leggerà effettivamente le read, mentre l'altra sfrutterà l'informazione precedentemente acquisita (a parte l'uso dell'informazione riguardante la lunghezza di ogni read) nel seguente modo: con l'inizializzazione verranno prelevati tutti i suffissi delle read che saranno ordinati in base al primo carattere (di essi sarà salvato solo l'indice di read e l'indice della posizione di inizio suffisso). Tutti questi suffissi possono essere dunque visti come se fossero raggruppati in bucket, ognuno contenente i suffissi ordinati in base ai prefissi di lunghezza 1, che chiameremo 1-bucket. Durante la fase di induzione, si otterranno in output i $2H$-bucket, ordinati in base ai precedenti H-bucket, usando la seguente proprietà [3] [19]: dati due suffissi $A(i_1, j_1)$ e $A(i_2, j_2)$ (usiamo $A(i, j)$ per indicare il j-esimo suffisso della i-esima read) che appartengono allo stesso H-bucket, per i quali dunque $A(i_1, j_1)$

$=_H A(i_2, j_2)$ ($=_H$ indica che due stringhe condividono lo stesso prefisso lungo H), questi saranno posti nei successivi $2H$-bucket in base agli H simboli successivi, da $j_1 + H + 1$ (e $j_2 + H+1$) fino a $j_1 + 2H$ (e $j_2 + 2H$). Questi sono i primi H simboli dei prefissi $A(i_1, j_1 + H)$ e $A(i_2, j_2 + H)$, dei quali sappiamo già il relativo ordine in base ai loro primi H simboli: infatti se i due suffissi appartengono a due H-bucket diversi, allora un H-bucket procede l'altro, e dunque, se per esempio l'H-bucket contenente $A(i_1, j_1 + H)$ precede l'H-bucket contenente $A(i_2, j_2 + H)$, allora sappiamo che $A(i_1, j_1 + H) <_H A(i_2, j_2 + H)$ ($<_H$ indica che una stringa precede l'altra in base ai primi H simboli), e di conseguenza $A(i_1, j_1) <_{2H} A(i_2, j_2)$, e si andranno a inserire i due suffissi $A(i_1, j_1)$ e $A(i_2, j_2)$ in due $2H$-bucket diversi, dove il $2H$-bucket rappresentato dal prefisso lungo $2H$ di $A(i_1, j_1)$ precede il $2H$-bucket rappresentato dal prefisso lungo $2H$ di $A(i_2, j_2)$. Se i due suffissi $A(i_1, j_1 + H)$ e $A(i_2, j_2 + H)$ appartengono invece allo stesso H-Bucket, allora anche i suffissi $A(i_1, j_1)$ e $A(i_2, j_2)$ apparterranno allo stesso $2H$-bucket (rappresentato dal prefisso lungo $2H$ di $A(i_1, j_1)$ e di $A(i_2, j_2)$) poiché condividono gli stessi $2H$ simboli. Come questa proprietà verrà usata sarà meglio specificato nel paragrafo §3.1.2. Procedendo dunque con questa induzione si continua finché H è minore della lunghezza massima delle stringhe, ottenendo infine una lista di bucket ordinati contenenti ognuno un suffisso (a meno che non siano presenti suffissi uguali), dai quali si estrae dunque la lista dei suffissi ordinati.

3.1.1 Inizializzazione

Come algoritmo per l'inizializzazione si è scelto un semplice counting-sort, riportato in Algoritmo 3.1; le strutture dati impiegate sono le seguenti: la matrice *prm* di taglia totale uguale al numero totale di basi (*total_size*) delle read, che corrisponde alla funzione inversa del suffix array (il suffix array riceve in input la posizione i e restituisce il suffisso (s, p) in posizione i-esima alfabeticamente, mentre *prm* riceve in input il (s, p) e restituisce la sua posizione all'interno del suffix array), il vettore *count_char* di taglia $|\Sigma|$ (Σ comprende in questo caso i cinque elementi *$*, *A*, *C*, *G* e *T*), e la matrice *reads* contenente le read. La funzione ***c_to_i*(char** *T*) restituisce semplicemente l'intero corrispondente al carattere *T* (0 per *$*, 1 per *A*, 2 per *C*, 3 per *G* e 4 per *T*). Nell'effettivo codice C++, al fine di usare solo valori unsigned, gli indici usati sono leggermente diversi.

Le righe 4 e 5 semplicemente inizializzano a zero il vettore *count_char*; le righe dalla 6 alla 13 contano il numero di volte che un determinato carattere T è presente, salvando tale conteggio nella corrispettiva posizione (data dalla funzione ***c_to_i*(char** *T*)) di *count_char*; le righe 14 e 15 calcolano la somma parziale di tali caratteri, in modo che ogni elemento di *count_char*, corrispondente al carattere *T,* contenga l'ultima posizione in cui inserire le stringhe che cominciano per *T*. Il codice che verrà riportato nella riga 16 sarà spiegato nel paragrafo §3.1.2. Dalle righe 17 alla 25 l'algoritmo procede a ritroso (dall'ultimo suffisso dell'ultima stringa) inserendo gli elementi nella posizione corretta e decrementando ogni volta il valore del rispettivo elemento in *count_char.* Alla fine, dunque, *prm* conterrà le posizioni dei suffissi in base all'ordine alfabetico del primo carattere di ogni suffisso.

Algoritmo 3.1

```
01.  count_char ← array di taglia |Σ|;
02.  prm ← array di taglia total_size;
03.  reads ← matrice contenente le reads (taglia totale total_size);
04.  for k from 0 to |Σ| - 1
05.     count_char[k] ← 0;
06.  for is from 0 to reads_number – 1 do
07.  {
08.     for ic from 0 to reads[is].size() – 1 do
09.     {
10.        num c ← c_to_i(reads[is][ic]);
11.        count_char[c]++;
12.     }
13.  }
14.  for k from 1 to |Σ| - 1 do
15.     count_char[k] ← count_char[k] + count_char[k - 1];
16.  //calcolo bh
17.  for is from reads_number - 1 to 0 do
18.  {
19.     for ic from reads[is].size() – 1 to 0 do
20.     {
21.        num c ← c_to_i(reads[is][ic]);
22.        count_char[c]--;
23.        prm[is][ic] ← count_char[c];
24.     }
25.  }
```

3.1.2 Induzione

La seconda fase procede per induzione: partendo dagli *H*-bucket calcola i 2*H*-bucket. Per fare ciò serviranno delle strutture dati aggiuntive: due array booleani *bh* e *b2h* di taglia *total_size* + 1, dove *bh* è utilizzato per indicare il confine tra i bucket (di default sono inizializzati a **false**), e *b2h* indica le posizioni dei suffissi spostati durante l'induzione; l'array *sa* di taglia *total_size* per indicare il suffix array, che riceverà in input la posizione *i* e restituisce in output la coppia (*s*, *p*) rappresentante il suffisso in tale posizione; l'array *count* di taglia *total_size*, usato per indicare il numero di elementi spostati nel H-bucket durante ogni iterazione. Gli array *bh*, *b2h* e *count* sono temporanei, e verranno eliminati alla fine del processo, mentre l'array *prm* sarà usato per la generazione del longest common prefixes (se usato nel calcolo del grafo) e poi eliminato, ed *sa* verrà mantenuto fino alla fine del calcolo del grafo.
Per inizializzare *bh* si inserisce il codice dell'algoritmo 3.2 nella riga 16 dell'algoritmo 3.1; in tal modo saranno posti a **true** solo le posizioni successive alla fine di ogni 1-bucket, delimitandone così il confine. L'array *sa* invece viene inizializzato usando *prm*, che rappresenta

Algoritmo 3.2

```
01.   bh ← array booleano inizializzato a false di taglia total_size + 1;
02.   for k from 0 to |Σ| - 1 do
03.      bh[count_char[k]] ← true;
```

Algoritmo 3.3

```
01.   sa ← array di taglia total_size;
02.   for is from 0 to reads_number - 1 do
03.      for ic from 0 to reads[is].size() – 1 do
04.         sa[prm[is][ic]] ← (is, ic);
05.
06.   count ← array di taglia total_size;
07.   for k from 0 to total_size – 1 do
08.      count[k] ← 0;
09.
10.   b2h ← array booleano inizializzato a false di taglia total_size + 1;
```

il suo inverso, come indicato dalle righe 2 alla 4 dell'algoritmo 3.4. il vettore *count* inizializza ogni suo elemento a 0 nelle righe da 7 e 8.

Il processo di calcolo del suffix array segue il codice riportato nell'algoritmo 3.3. il ciclo principale (riga 3) viene ripetuto finché *h* non è maggiore di *size_max*. La prima sezione (dalla riga 7 alla riga 18) resetta tutti i valori di *prm* in modo che puntino al primo elemento del *H*-bucket di appartenenza, piuttosto che all'effettiva posizione nel *H*-bucket (a meno che non siano più corti della taglia della relativa stringa diminuita di *h*, in tal caso non vengono spostati perché sono effettivamente i primi elementi del 2*H*-bucket). Inoltre, si resettano a zero tutti gli elementi di *count*. La seconda sezione del codice, (dalla riga 21 alla riga 57) scorre l'array *sa*, un *H*-bucket alla volta. Chiamiamo *r* ed *l* i limiti destro e sinistro del *H*-bucket che si sta analizzando, e T_i il suffisso *sa*[*i*] – (0, *h*). Per ogni i, con $l <= i <= r$, incrementiamo *count*[*prm*[T_i]], settiamo *prm*[T_i] uguale a *prm*[T_i] + *count*[*prm*[T_i]] – 1, e impostiamo *b2h*[*prm*[T_i]] a **true**. Così facendo, tutti i suffissi i cui elementi dal (H + 1)-esimo al 2*H*-esimo sono uguali al prefisso lungo *H* che rappresenta il bucket attualmente analizzato vengono mossi in testa al loro *H*-bucket, ovviamente subito dopo i suffissi in quel *H*-bucket già mossi durante i cicli (riga 25) precedenti. L'array *b2h* indica le posizioni dei suffissi spostati nel bucket. Prima di ricominciare il ciclo si modifica *b2h*, settando a **false** le posizioni che non rappresentano il confine tra i nuovi 2*H*-bucket (riga 49). Infine, la terza sezione di codice (dalla riga 59 in poi) aggiorna *sa* usando il suo inverso, *prm*, e si salvano i nuovi 2*H*-buchet, salvati in *b2h*, nell'array *bh*, contenente ancora gli *H*-bucket. Alla fine di tutto il processo si ottiene in output il suffix array generalizzato *sa*, e si possono eliminare le altre strutture, a parte *prm*, che verrà utilizzato per la costruzione del longest common prefixes (se usato nel calcolo del grafo).

Algoritmo 3.4

```
01.  num h ← 1;
02.  while h <= size_max + 1 do
03.  {
04.     num i ← 0;
05.     num j ← 0;
06.     num k ← 0;
07.     while i < total_size do
08.     {
09.        j ← i;
10.        count[j] ← 0;
11.        do
12.        {
13.           (num is, num ic) ← sa[i];
14.           if ic < reads[is].size() – h then
15.              prm[is, ic] ← j;
16.           i++;
17.        } while !bh[i];
18.     }
19.     i ← 0;
20.     j ← 0;
21.     while i < total_size do
22.        {
23.        j ← k;
24.        i ← j;
25.        do
26.        {
27.           (num is, num ic) ← sa[i] – (0, h);
28.           if ic >= 0 then
29.           {
30.              count[prm[is, ic]] ← count[prm[is, ic]] + 1;
31.              prm[is, ic] ← prm[is, ic] + count[prm[is, ic]] – 1;
32.              b2h[prm[is, ic]] ← true;
33.           }
34.           i++;
35.        } while !bh[i];
36.
37.        k ← i;
38.        i ← j;
39.        do
40.        {
41.           (num is, num ic) ← sa[i] – (0, h);
42.           if ic >= 0 then
43.           {
44.              if b2h[prm[is, ic]] then
45.              {
```

```
46.                 num  e ← prm[is, ic] + 1;
47.                 while !bh[e] and b2h[e] do
48.                 {
49.                     b2h[e] ← false;
50.                     e++;
51.                 }
52.             }
53.         }
54.         i++;
55.     } while i != k;
56.     i ← k;
57. }
58.
59. for is from 0 to reads_number - 1 do
60.     for ic from 0 to reads[is].size() – 1 do
61.         sa[prm[is][ic]] ← (is, ic);
62.
63. for t from 0 to total_size - 1 do
64.     if b2h[t] and !bh.[t] then
65.         bh[t] ← b2h[t];
66.
67. h ← h*2;
68. }
```

3.1.3 Durata del processo

Il processo di inizializzazione riportato nell'algoritmo 3.1 effettua quattro cicli indipendenti, di cui i due più lunghi effettuano *total_size* iterazioni. Dunque, il tempo impiegato nella prima fase è $O(total_size)$, e quindi lineare nel numero di basi in input. La seconda fase ha un ciclo principale, che, poiché dura finché *h* non è maggiore di *size_max*, e poiché *h* viene raddoppiato a ogni iterazione, effettua $\log_2(size_max)$ iterazioni. All'interno di tale ciclo sono presenti a loro volta altri due cicli, i quali sono effettuati per *total_size* iterazioni (il secondo contiene al suo interno un altro ciclo che effettua, sommando ognuna delle sue iterazioni del ciclo precedente, *total_size* operazioni, e dunque la durata del secondo ciclo resta $O(total_size)$). La durata di tale processo è dunque $O(total_size \log size_max)$. Poiché nel processo sono stati usati 5 vettori (*bh*, *b2h*, *count*, *prm* e *sa*), tutti di dimensione *total_size*, lo spazio totale impiegato risulta essere 5* *total_size*, e dunque lineare. Tuttavia, è possibile salvare i due array booleani, *bh* e *b2h*, nei bit di segno degli altri array, riducendo così lo spazio a 3**total_size*. Nel nostro progetto sono state impiegate classi apposite per la riduzione dello spazio, che verranno illustrate nel paragrafo §3.4 e approfondite con il relativo codice nell'Appendice A.

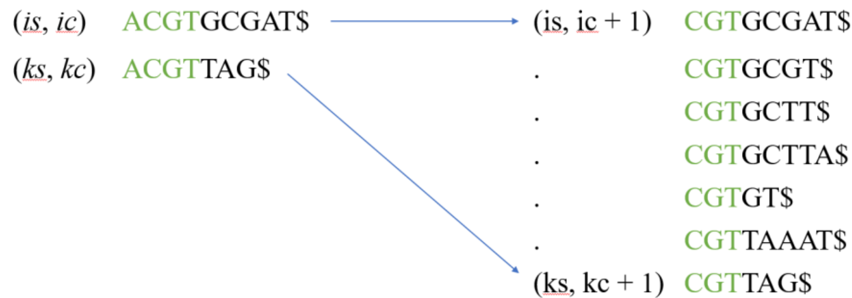


Figure 3.1. *Esempio di prefissi comuni nel suffix array: poiché CGTGCGAT$ e CGTTAG$ condividono il prefisso CGT, questo sarà presente anche in tutti gli elementi tra questi due nel suffix array*

3.2 Generazione del longest common prefixes

Ottenuto il suffix array delle read in input, si passa ad ottenere il longest common prefixes (*lcp*). Ricordiamo che questo è rappresentato dall'array *lcp*, dove ogni elemento $lcp[i]$ rappresenta la lunghezza del prefisso comune tra $sa[i - 1]$ e $sa[i]$ ($lcp[0]$ è 0). Una tecnica naive potrebbe essere andare a scorrere i suffissi del suffix array a uno a uno, contando ogni volta il numero di caratteri nel prefisso comune. Esiste una tecnica tuttavia assai più efficiente: si calcola il prefisso comune (che chiamiamo p), per esempio, dei primi due elementi del suffix array, dunque $sa[0]$ e $sa[1]$, ma, all'iterazione successiva, si considera non il suffisso successivo nel suffix array, ma il suffisso successivo rispetto $sa[0]$ (se, per esempio, $sa[0] = (i, j)$ corrisponde al suffisso ACGTGCGAT$, allora il suffisso successivo sarà CGTGCGAT$, rappresentato dalle posizioni $(i, j + 1)$), che chiamiamo $sa[k]$, e rispetto a $sa[1]$, che chiamiamo $sa[k + n]$. Il fatto che $sa[0]$ precede $sa[1]$, se questi hanno un prefisso comune maggiore di 0, implica che anche $sa[k]$ precederà $sa[k + n]$. Tali suffissi, anche se non è detto che siano immediatamente uno successivo all'altro, avranno in comune un prefisso di lunghezza $|p| - 1$, e forniscono inoltre informazioni sui prefissi degli elementi nel mezzo, e dunque $sa[k]$, $sa[k + 1]$, …, $sa[k + n]$. Come esemplificato in Figura 3.1, tutti i prefissi di tali elementi avranno in comune gli stessi caratteri del prefisso comune ai due estremi, $sa[k]$ e $sa[k + n]$, ovverosia i primi $|p| - 1$ caratteri. In particolare, si considera l'elemento immediatamente successivo, $sa[k + 1]$, e, per calcolare il longest common prefix con l'elemento precedente ($sa[k]$), sapendo già che i primi $|p| - 1$ caratteri del prefisso sono condivisi, questi vengono saltati, passando ai confronti dei caratteri successivi. Tale ciclo si ripete considerando il suffisso successivo alla stringa di partenza, finchè questa non finisce, e ripetendo il processo per le stringhe seguenti. Per fare ciò è sufficiente scorrere linearmente l'array *prm*, ottenuto nel paragrafo §3.1. L'algoritmo 3.5 illustra il procedimento completo. In particolare, le righe 16 e 18 mostrano come il confronto tra due sequenze si fermi quando si trova un carattere nella prima che preceda il corrispondente carattere nella seconda, o quando i suffissi sono uguali fino all'ultimo carattere (essendo un suffix array generalizzato potrebbero essere presenti suffissi uguali).

Algoritmo 3.5

```
01.  lcp ← array di taglia total_size;
02.  for k from 0 to total_size - 1 do
03.     lcp[0] ← 0;
04.
05.  num s ← 0;
06.  for is from 0 to reads_number – 1 do
07.  {
08.     for ic from 0 to  reads[is].size() – 1 do
09.     {
10.        k ← prm[is][ic];
11.        if k > 0 then
12.        {
13.           (num js, num jc) ← sa[k - 1];
14.           while true do
15.           {
16.              if jc + s == reads[js].size() and ic + s == reads[is].size() then
17.                 break;
18.              else if c_to_i(reads[is][ic + s]) < c_to_i(reads[is][ic + s]) then
19.                 break;
20.              else if c_to_i(reads[js][jc + s]) == c_to_i(reads[is][ic + s]) then
21.                 s = s + 1;
22.           }
23.           lcp[k] ← s;
24.           if s > 0 then
25.              s = s - 1;
26.        }
27.     }
28.  }
```

Algoritmo 3.6

```
01.  position ← position for the BWT value to get
02.  (num is, num ic) ← sa[position];
03.  if ic > 0 then
04.     return reads[is][ic - 1];
05.  else if ic == 0 then
06.     return '$';
```

In tutto vi sono *total_size* iterazione, ma poiché a ogni iterazione si può ottenere un nuovo valore, dato il precedente $|p|$, che varia tra $|p|$ - 1 e *size_max*, il numero di confronti che si può effettuare resta comunque $O(total_size)$, rendendo dunque l'algoritmo estremamente efficiente.

3.3 Generazione BWT e FM-index

Il processo di generazione della BWT è estremamente semplice: come già detto, ogni elemento della BWT può essere ottenuto mediante il suffix array, e dunque $BWT[i] = r_j[k - 1]$, dove $SA[i] = (j, k)$, se $k > 0$, o $BWT[i] = \$$ se $k = 0$. Poiché tale operazione è $O(1)$, e comunque vengono tenuti in memoria il suffix array e l'array *reads* contenente le sequenze di partenza, si è deciso di non memorizzare la *BWT*, ma di ottenere il valore di ogni suo elemento dinamicamente quando questo viene richiesto, mediante la formula descritta. Il codice implementato per ottenere tale valore è riportato nell'algoritmo 3.6, e sarà implementato nella funzione della classe da cui ottenere la *BWT*. Per l'FM-index vengono invece memorizzati gli array count compressi come descritto nel paragrafo §2.4. Tale memorizzazione viene fatta semplicemente scorrendo la BWT e incrementando le variabili di conteggio dei caratteri (*totS*, *totA, totC*, *totG* e *totT*), e salvando tali valori nei relativi array di conteggio compressi quando la posizione è divisibile per lo spazio impiegato tra gli elementi degli array di conteggio completo. Il codice completo per ottenere gli array di conteggio compressi è riportato nell'algoritmo 3.7. Una volta ottenuti gli array di conteggio si salvano anche le variabili contenenti la somma del numero di caratteri successivi (*totSA*, *totSAC*, *totSACG* e *totSACGT*), che saranno usate nel processo di ricerca della *BWT*.
Il ciclo presente nell'algoritmo 3.7 scorre semplicemente il suffix array, lungo *total_size*, e poiché le operazioni presenti al suo interno sono $O(1)$ la durata del processo di generazione degli array di conteggio compressi è dunque $O(total_size)$. Per la generazione delle strutture dati, dunque, l'operazione più dispendiosa a livello temporale (e spaziale) resta il calcolo del suffix array descritto nel paragrafo §3.1.

3.4 Ottimizzazione delle strutture dati

Molti degli array numerici impiegati hanno la peculiarità di contenere elementi i cui valori massimo e minimo sono noti: per esempio, nel suffix array, le coppie di elementi (ID stringa, ID suffisso) sono comprese tra (0, 0) e (*reads_number*, *size_max*). Usando tale informazione è possibile rappresentare i valori mediante un numero di bit inferiore al numero che sarebbe richiesto usando i tipi di default del C++ (**long**, **int**, **short**, **char**). Per fare ciò è stata creata la classe **clsSizeArray**, che genera un array di **unsigned long** (composto da 64 bit), ma di cui usa solo i bit necessari per la rappresentazione di ogni cifra e le pone consecutivamente in ogni elemento dell'array. Un approccio simile è stato impiegato per l'array *reads*, contenente stringhe che rappresentano read costituite da 5 possibili valori (*$*, *A*, *C*, *G* e *T*), ma, poiché il carattere *$* è presente solo alla fine della stringa, e ogni stringa contiene anche la sua lunghezza *size_string*, è sufficiente salvare gli elementi da 0 fino a *size_string* – 2 (contenenti solo i caratteri *A*, *C*, *G* e *T*), e restituire il carattere *$* solo quando viene richiesto l'ultimo carattere, in posizione *size_string* – 1. È stata dunque sviluppata la classe **clsString** per memorizzare una read, che utilizza un array di **char**, (composto da 8 bit), per salvare l'informazione rappresentata

Algoritmo 3.7

```
01.  countA ← array di taglia total_size;
02.  countC ← array di taglia total_size;
03.  countG ← array di taglia total_size;
04.  countT ← array di taglia total_size;
05.  countS ← array di taglia total_size;
06.
07.  num totS ← 0
08.  num totA ← 0
09.  num totC ← 0
10.  num totG ← 0
11.  num totT ← 0
12.
13.  num totSA ← 0
14.  num totSAC ← 0
15.  num totSACG ← 0
16.  num totSACGT ← 0
17.
18.  for i from 0 to total_size - 1 do
19.  {
20.     (num is, num ic) ← sa[i];
21.     if ic > 0 then
22.     {
23.        if reads[is][ic - 1] == 'A' then
24.           totA++;
25.        else if reads[is][ic - 1] == 'C' then
26.           totC++;
27.        else if reads[is][ic - 1] == 'G' then
28.           totG++;
29.        else if reads[is][ic - 1] == 'T' then
30.           totT++;
31.     }
32.     else if ic == 0 then
33.     {
34.        totS++;
35.     }
36.
37.     if i % spazio == 0 then
38.     {
39.        countA[i] ← totA;
40.        countC[i] ← totC;
41.        countG[i] ← totG;
42.        countT[i] ← totT;
43.        countS[i] ← totS;
44.     }
45.  }
46.
```

```
47.  totSA ← totS + totA;
48.  totSAC ← totS + totA + totC;
49.  totSACG ← totS + totA + totC + totG;
50.  totSACGT ← totS + totA + totC + totG + totT;
```

da due bit per ogni carattere (dunque quattro caratteri consecutivi per ogni **char**). Infine, usando una tecnica simile a **clsString**, per gli array booleani è stata scritta la classe **clsArrayBool**, dove l'informazione è salvata in un array di **char**, contenente 8 valori booleani consecutivi (un bit per ognuno). Il codice delle classi **clsSizeArray**, **clsString**, **clsSuffixArray** (utilizzato per memorizzare il suffix array) e **clsCountChar** (utilizzato per contenere gli array di conteggio) e la loro spiegazione sono riportati in Appendice A. Sono state tralasciate le classi **clsArrayBool** (poiché molto simile a **clsString**) e la classe **clsBWTSeq**, usata per ottenere i valori della BWT (poiché semplicemente riporta l'algoritmo 3.6).

3.5 Gestione errori

Come già detto, MetaProb ignora i q-mer contenenti caratteri errati (diversi da A, C, G e T). Per il nostro progetto abbiamo utilizzato un approccio simile: è stata generata la matrice di valori booleani *correct_array*, dove ogni elemento in posizione (*is*, *ic*) specifica se il q-mer che inizia dall'*ic*-esimo carattere della *is*-esima read contiene caratteri errati (gli elementi che iniziano in $ic > reads[is].size() - q$ potrebbero essere ignorati perché la loro taglia è minore di q, ma saranno comunque posti a **true** o **false** sulla base della correttezza dei caratteri da *ic* a *reads*[*is*].***size***() - 1). Poiché alcune delle tecniche di costruzione del grafo richiedono di controllare la correttezza di sottostringhe di taglia maggiore di q, è stata implementata la funzione ***check_correct***, riportata nell'algoritmo 3.8: le righe dalla 7 alla 12 controllano, la correttezza dei q-mer saltando di q caratteri ogni volta, dunque, supponendo controllino i caratteri da *is* a $is + size_check - 1$, la prima iterazione controlla la correttezza dei caratteri da *is* a $is + q - 1$, la seconda da $is + q$ a $is + 2*q - 1$, finché il q-mer non finisce oltre $is + size_check - 1$ (escluso quest'ultimo q-mer). Poiché alla fine della sezione della stringa di cui verificare la correttezza potrebbero essere rimasti dei caratteri ancora non controllati, si osserva la correttezza del q-mer che finisce in $ic + size_check - 1$, e dunque che inizia in $ic + size_check - q$ (riga 14 e 15). Si assume che *size_check* sia sempre maggiore o uguale a q.

3.6 Tecnica 1: grafo prefisso-suffisso ottenuto mediante FM-index

La prima tecnica, il cui codice è riportato nell'algoritmo 3.9, usata per la generazione del grafo è la stessa impiegata in [15] e [16], e si basa sull'uso dell'FM-index: questa genera il grafo che ha come overlap prefissi e suffissi, dai quali, supponendo abbiano taglia *size_overlap* e che questa sia maggiore o uguale a $q + m - 1$, viene generato un lato con score $size_overlap - q + 1$.

Algoritmo 3.8

```
01.   q ← lunghezza dei q-mer;
02.
03.   bool check_correct(num k, num size_check)
04.   {
05.      bool correct ← true;
06.      (num is, num ic) ← sa[k];
07.      num j ← 0;
08.
09.      while j < size_check – q do
10.      {
11.         if not correct_array[is][ic + j] then
12.            correct ← false;
13.         j ← j + q;
14.      }
15.
16.      if not correct_array[is][ic + size_check - q] then
17.         correct ← false;
18.
19.      return correct;
20.   }
```

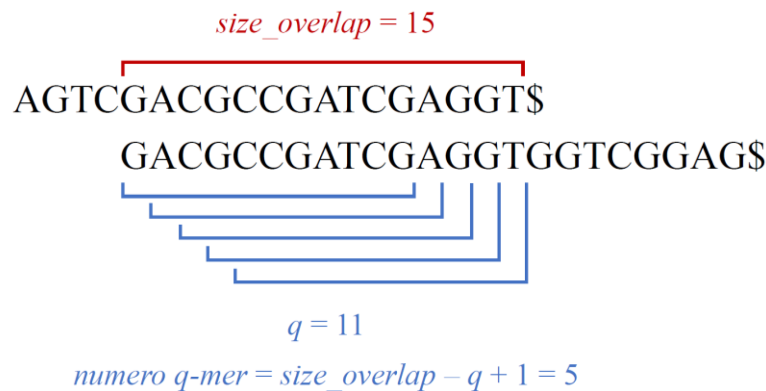


Figure 3.2. *Esempio che evidenzia il numero di q-mer generati da un overlap di taglia maggiore di q*

Il motivo di questa scelta è riportato in Figura 3.2: per mantenere una similarità rispetto agli score che vengono utilizzati in MetaProb, si è osservato il numero di q-mer generato da un overlap lungo *size_overlap* è *size_overlap* - $q + 1$, e che la condizione che siano presenti almeno m q-mer equivale ad avere *size_overlap* >= $q + m - 1$. La ricerca di overlap funziona nel seguente modo: supponendo di aver trovato tutti gli elementi che iniziano con la stringa ω, questi sono presenti sequenzialmente nel suffix array, e possono dunque essere rappresentati dagli indici della prima (b_ω) e dall'ultima (e_ω) posizione in cui ω compare, ovverosia dall'intervallo $[b_\omega, e_\omega]$. Tutti questi sono potenziali overlap prefisso-suffisso: per controllare quali di questi sono suffissi, oltre a tenere gli indici b_ω e e_ω, vengono conservati anche gli indici relativi la stringa $\omega\$$, usati alla riga 28, che indicheremo con $b_{\omega\$}$ e $e_{\omega\$}$ (sarebbe possibile tenere

Algoritmo 3.9

```
01.  m ← numero minimo di presenze dei q-mer per generare adiacenza
02.
03.  void ex_ss(num sp_sub, num ep_sub, num sp_suf, num ep_suf, num size_overlap)
04.  {
05.     if size_overlap >= q + m - 1 then
06.     {
07.        num sp_pref;
08.        num ep_pref;
09.        find_occorrenze('$', sp_sub, ep_sub, sp_pref, ep_pref);
10.        num n_pref ← ep_pref - sp_pref + 1;
11.        if n_pref > 0 then
12.        {
13.           vector prefissi_id ← vettore di id_seq_type;
14.           vector suffissi_id ← vettore di id_seq_type;
15.           for i from sp_sub to ep_sub do
16.           {
17.              (num is, num ic) ← sa[i];
18.              if ic == 0 then
19.              {
20.                 bool correct ← check_correct(i, size_overlap);
21.                 if correct then
22.                 {
23.                    id_seq_type ID ← getIDsSeq(i);
24.                    prefissi_id.push_back(ID);
25.                 }
26.              }
27.           }
28.           for i from sp_suf to ep_suf do
29.           {
30.              bool correct ← check_correct(i, size_overlap);
31.              if correct then
32.              {
33.                 id_seq_type ID ← getIDsSeq(i);
34.                 suffissi_id.push_back(ID);
35.              }
36.           }
37.           for i from 0 to prefissi_id.size() – 1 do
38.           {
39.              seq prefisso ← getSeqFromID(prefissi_id[i]);
40.              for j from 0 to suffissi_id.size() - 1 do
41.              {
42.                 if prefissi_id[i] != suffissi_id[j] then
43.                 {
44.                    seq suffisso ← getSeqFromID(suffissi_id[j]);
45.                    prefisso.AggiornaAdjVertice(suffisso, size_overlap - q + 1);
46.                    suffisso.AggiornaAdjVertice(prefisso, size_overlap - q + 1);
47.                 }
```

```
48.                 }
49.             }
50.         }
51.     }
52.     num new_sp_sub;
53.     num new_ep_sub;
54.     num new_sp_suf;
55.     num new_ep_suf;
56.     // num n_sub;
57.     num n_suf;
58.
59.     find_occorrenze('A', sp_sub, ep_sub, new_sp_sub, new_ep_sub);
60.     find_occorrenze('A', sp_suf, ep_suf, new_sp_suf, new_ep_suf);
61.
62.     // n_sub ← new_ep_sub - new_sp_sub + 1;
63.     n_suf ← new_ep_suf - new_sp_suf + 1;
64.
65.     if n_suf > 0 then // and n_sub > n_suf
66.         ex_ss(new_sp_sub, new_ep_sub, new_sp_suf, new_ep_suf, size_overlap + 1);
67.
68.     find_occorrenze('C', sp_sub, ep_sub, new_sp_sub, new_ep_sub);
69.     find_occorrenze('C', sp_suf, ep_suf, new_sp_suf, new_ep_suf);
70.
71.     // n_sub ← new_ep_sub - new_sp_sub + 1;
72.     n_suf ← new_ep_suf - new_sp_suf + 1;
73.
74.     if n_suf > 0 then // and n_sub > n_suf
75.         ex_ss(new_sp_sub, new_ep_sub, new_sp_suf, new_ep_suf, size_overlap + 1);
76.
77.     find_occorrenze('G', sp_sub, ep_sub, new_sp_sub, new_ep_sub);
78.     find_occorrenze('G', sp_suf, ep_suf, new_sp_suf, new_ep_suf);
79.
80.     // n_sub ← new_ep_sub - new_sp_sub + 1;
81.     n_suf ← new_ep_suf - new_sp_suf + 1;
82.
83.     if n_suf > 0 then // and n_sub > n_suf
84.         ex_ss(new_sp_sub, new_ep_sub, new_sp_suf, new_ep_suf, size_overlap + 1);
85.
86.     find_occorrenze('T', sp_sub, ep_sub, new_sp_sub, new_ep_sub);
87.     find_occorrenze('T', sp_suf, ep_suf, new_sp_suf, new_ep_suf);
88.
89.     // n_sub ← new_ep_sub - new_sp_sub + 1;
90.     n_suf ← new_ep_suf - new_sp_suf + 1;
91.
92.     if n_suf > 0 then // and n_sub > n_suf
93.         ex_ss(new_sp_sub, new_ep_sub, new_sp_suf, new_ep_suf, size_overlap + 1);
94. }
```

Algoritmo 3.10

```
01.  void find_occorrenze(char carat, num sp, num ep, num& new_sp, num& new_ep)
02.  {
03.     if carat == '$' then
04.     {
01.        if sp == 0 then
02.           new_sp ← 0;
03.        else
04.           new_sp ← countS[sp - 1];
05.        new_ep ← countS[ep] - 1;
06.     }
07.     else if carat == 'A' then
08.     {
09.        if sp == 0 then
10.           new_sp ← totS;
11.        else
12.           new_sp ← totS + countA[sp - 1];
13.        new_ep ← totS + countA[ep] - 1;
14.     }
15.     else if carat == 'C' then
16.     {
17.        if sp == 0 then
18.           new_sp ← totSA;
19.        else
20.           new_sp ← totSA + countC[sp - 1];
21.        new_ep ← totSA + countC[ep] - 1;
22.     }
23.     else if carat == 'G' then
24.     {
25.        if sp == 0 then
26.           new_sp ← totSAC;
27.        else
28.           new_sp ← totSAC + countG[sp - 1];
29.        new_ep ← totSAC + countG[ep] - 1;
30.     }
31.     else if carat == 'T' then
32.     {
33.        if sp == 0 then
34.           new_sp ← totSACG;
35.        else
36.           new_sp ← totSACG + countT[sp - 1];
37.        new_ep ← totSACG + countT[ep] - 1;
38.     }
39.  }
```

solo gli indici b_ω, e_ω e $e_{\omega\$}$ poiché b_ω è sempre uguale a $b_{\omega\$}$, ma poiché questi non occupano una quantità rilevante di memoria e poiché la loro unione non avrebbe portato a un miglioramento del tempo di calcolo significativo, si è scelto di tenerli separati), mentre per controllare quali sono prefissi, è sufficiente controllare quali iniziano in posizione 0 della loro stringa mediante il suffix array, come riportato alla riga 18. Ottenuti i suffissi e i prefissi vengono generate le adiacenze, come indicato dalla riga 37 alla 49. La funzione ***AggiornaAdjVertice***(**seq** *sequenza*, **num** *overlap*) aggiunge alla sequenza chiamante un'adiacenza con score *overlap* verso *sequenza*. Se è già presente un'adiacenza verso *sequenza*, e se questa ha score minore di *overlap* allora lo score è posto uguale a *overlap*, salvando dunque così ad ogni chiamata di ***AggiornaAdjVertice*** l'adiacenza di score maggiore dalla sequenza chiamante a *sequenza*.
Vengono poi cercati i nuovi overlap partendo dalla stringa ω, i quali sono $A\omega$, $C\omega$, $G\omega$ e $T\omega$, insieme ai relativi suffissi $A\omega\$$, $C\omega\$$, $G\omega\$$ e $T\omega\$$. Come visto nel paragrafo §2.4.2, dato un carattere c, e l'intervallo $[b_\omega, e_\omega]$ del suffix array nel quale compare ω, per cercare nell'FM-index le stringhe che iniziano con $c\omega$ viene impiegato tempo $O(1)$ (lo stesso ovviamente vale per ogni stringa, quindi anche per l'intervallo $[b_{\omega\$}, e_{\omega\$}]$ relativo la stringa $\omega\$$); questo processo è chiamato backward c-extension, ed implementato dal codice dell'algoritmo 3.10 (funzione ***find_occorrenze***), ed è dunque usato poi nell'algoritmo 3.9. Da questi nuovi overlap trovati verrà ripetuto il processo chiamando ricorsivamente l'algoritmo 3.9.
Per avviare l'algoritmo è sufficiente chiamare la funzione dandogli la stringa vuota in input, dunque $[b_\omega, e_\omega] = [0, totSACGT - 1]$ (ricordiamo che *totSACGT* è il numero di caratteri presenti nella BWT, o alternativamente si poteva utilizzare $[0, total_size - 1]$) e $size_overlap = 0$, mentre per gli indici dei suffissi si ha la concatenazione nella stringa vuota con $\$$, che è semplicemente la stringa $\$$, e dunque $[b_{\omega\$}, e_{\omega\$}] = [0, totS - 1]$ (*totS* è il numero di volte che è presente il carattere $\$$ nella BWT).
Per velocizzare il processo si osserva se è presente almeno un prefisso nella lista dei prefissi trovati (riga 11), mentre la ricorsione viene effettuata solo se il numero di suffissi presenti è maggiore di 0 (righe 65, 74, 83 e 92). Nella versione originaria ([15] e [16]) dell'algoritmo si osservava anche se potevano essere presenti potenziali prefissi, guardando se il numero di sottostringhe contenenti ω era maggiore del numero di suffissi contenenti ω (righe commentate dalla 58 alla 88) tuttavia ciò valeva sotto l'assunzione che nessun prefisso fosse anche suffisso, ipotesi da noi rimossa per ottenere lo stesso grafo che sarà ottenuto nella tecnica 2, illustrata nel paragrafo seguente.
Calcoliamo ora il tempo impiegato per la costruzione del grafo: poiché ogni potenziale overlap è un suffisso dell'input, vi possono essere al più NM possibili suffissi (useremo la stessa notazione impiegata nel paragrafo §1.3.3, dove N sono il numero di read e M la loro lunghezza, assumendo per l'analisi che abbiano tutte lunghezze uguali), e, poiché sono presenti all'interno del codice dei cicli per la generazione dei lati, si ha anche un tempo addizionale di n_A (chiamiamo n_A il numero di lati generati). Ricordiamo inoltre che viene introdotto un ritardo dovuto allo spazio, che indicheremo con la variabile k, tra gli elementi all'interno degli array di conteggio, che possiamo considerare come se fosse presente in ogni iterazione della funzione

ex_ss dell'algoritmo 3.9, eseguito per ognuno dei possibili *NM* suffissi. Il tempo totale può essere dunque considerato approssimativamente $O(NMk + n_A)$.
Usando poi le formule dello spazio delle classi impiegate nel progetto, che vengono analizzate in appendice A, possiamo dedurre lo spazio occupato dalle strutture dati per la costruzione del grafo: sappiamo che ognuno dei cinque array di conteggi impiega

$$\left\lceil \frac{NM}{k} \right\rceil \lceil \log_2(NM + 1) \rceil \quad \text{bit} \quad , \tag{3.1}$$

che il suffix array impiega

$$NM\lceil \log_2(N) \rceil \quad \text{bit} \quad , \tag{3.2}$$

$$NM\lceil \log_2(M) \rceil \quad \text{bit} \quad , \tag{3.3}$$

e che le read memorizzate impiegano

$$2NM + 16N \qquad \text{bit} \quad . \tag{3.4}$$

Dunque, il consumo totale di memoria può essere considerato

$$NM\lceil \log_2(N) \rceil + NM\lceil \log_2(M) \rceil + 5\left\lceil \frac{NM}{k} \right\rceil \lceil \log_2(NM + 1) \rceil + 2NM + 16N \quad \text{bit} \quad . \tag{3.5}$$

Ricordiamo che sono inoltre a loro volta presenti degli indirizzi di memoria all'interno degli array che puntano alle classi, come l'array contenente le read, ma questi occupano relativamente meno spazio rispetto al resto delle strutture, e dunque la memoria indicata nella formula precedente può essere considerata una buona approssimazione.

3.7 Tecnica 2: grafo prefisso-suffisso ottenuto mediante LCP

Un'altra tecnica assai più semplice e veloce è risultata essere l'uso del *lcp*, che verrà usato per tutte le prossime tecniche, ed in particolare per la seconda tecnica è utilizzato per la generazione del grafo prefisso-suffisso (lo stesso della tecnica precedente), il cui codice è riportato nell'algoritmo 3.11. Come mostrato in Figura 3.3, dato un certo suffisso in posizione *i* del suffix array, e dunque tale che $sa[i] = (is, ic)$, le sottostringhe comuni con la sottostringa di *is* che inizia dal carattere *ic* sono presenti in successione nel suffix array, e dunque è possibile individuare tutti gli overlap comuni scorrendo linearmente il suffix array, e per ogni suo

Algoritmo 3.11

```
01.  for k from 0 to total_size – 2 do
02.  {
03.     (num is, num ic) ← sa[k];
04.     num n ← k + 1;
05.     num size_prefix ← lcp[n];
06.     if ic + size_prefix == reads[is].size() then
07.        size_prefix ← size_prefix - 1;
08.     if size_prefix >= q + m - 1 then
09.     {
10.        id_seq_type ID1 ← getIDsSeq(k);
11.        seq sottostringa1 ← getSeqFromID(ID1);
12.        bool correct1 ← check_correct(k, size_prefix);
13.        if correct1 then
14.        {
15.           while size_prefix >= q + m - 1 do
16.                 and n < total_size
17.                 and (ic == 0 or ic + size_prefix == reads[is].size() - 1)
18.           {
19.              (num js, num jc) ← sa[n];
20.              id_seq_type ID2 ← getIDsSeq(n);
21.              bool correct2 ← check_correct(n, size_prefix);
22.              if ID1 != ID2 and correct2
23.                          and ((ic == 0 and jc + size_prefix == reads[js].size() - 1)
24.                             or (jc == 0 and ic + size_prefix == reads[is].size() - 1))
25.                             then
26.              {
27.                 seq sottostringa2 ← getSeqFromID(ID2);
28.                 sottostringa1.AggiornaAdjVertice(sottostringa2, size_prefix - q + 1);
29.                 sottostringa2.AggiornaAdjVertice(sottostringa1, size_prefix - q + 1);
30.              }
31.              n ← n + 1;
32.              if n < total_size and lcp[n] < size_prefix then
33.                 size_prefix ← lcp[n];
34.           }
35.        }
36.     }
37.  }
```

elemento i si determina l'overlap con i suffissi successivi nel seguente modo: si salva nella variabile *size_prefix* il valore di $lcp[i + 1]$, ovverosia il massimo overlap condiviso con il suffisso in posizione $i + 1$, che verrà usato per aggiornare l'adiacenza tra la sequenza in posizione i ed $i + 1$ mediante la funzione ***AggiornaAdjVertice*** (righe 28 e 29). Per continuare a calcolare le adiacenze alla sequenza in posizione i, si considera adesso la sequenza in posizione $i + 2$: in particolare, se il valore di $lcp[i + 2]$ è inferiore a *size_prefix*, allora questo valore sarà la nuova grandezza dell'overlap tra gli elementi in posizione i e $i + 2$, e dunque si porrà

SUFFIX ARRAY (STRINGA, SUFFISSO)	OVERLAP	LCP	SUFFISSO
.	.	.	.
.	.	.	.
.	.	.	.
(825, 25)	.	.	GCCGACCGAGAT$
(147, 28)	8	8	GCCGACCGT$
(9587, 28)	7	7	GCCGACCTT$
(54, 31)	5	5	GCCGAG$
(578, 31)	5	7	GCCGAG$
(429, 27)	5	6	GCCGAGAAGT$
(368, 29)	5	7	GCCGAGAT$
(525, 33)	4	4	GCCGT$
(84, 26)	2	2	GCGATTCGAGT$
(525, 28)	1	1	GGTATGCCGT$
.	.	.	.
.	.	.	.
.	.	.	.

Figure 3.3. *In figura è evidenziato come sia possibile ottenere le sottostringhe uguali al prefisso di un elemento del suffix array mediante lcp; in questo caso l'elemento è il suffisso 25 della stringa 825, GCCGATCGAGAT$, e le sottostringhe comuni di taglia uguale o maggiore della soglia (che, in questo caso, supponiamo essere 4) sono evidenziate in verde, mentre quelle di taglia inferiore sono evidenziate in rosso*

size_prefix = *lcp*[*i* + 2], altrimenti l'overlap avrà ancora dimensione uguale a *size_prefix*. Questo valore di overlap verrà usato per aggiornare l'adiacenza tra l'adiacenza tra la sequenza in posizione *i* ed *i* + 2 mediante la funzione ***AggiornaAdjVertice*** e il processo verrà nuovamente ripetuto con la sequenza in posizione *i* + 3, poi *i* + 4 e così via, finché *size_prefix* non diventa minore della soglia $q + m - 1$ (la stessa soglia della tecnica 1), che sarà dunque la taglia minima di ogni overlap, o finché non si raggiunge la fine del suffix array (riga 32). Per ottenere tutte le adiacenze è dunque sufficiente scorrere linearmente l'intero *lcp*. Poiché il grafo generato è il grafo suffisso-prefisso, un'adiacenza tra due sequenze viene generata solo se una ha come overlap un suffisso (rispettivamente prefisso), e l'altra ha come overlap un prefisso (rispettivamente suffisso) (righe 23 e 24, ed è stato aggiunto anche un controllo alla riga 17 sulla sequenza di riferimento, poiché resta la stessa per tutto il ciclo alla riga 15, interrompendolo così se questa non può essere né il suffisso né il prefisso di un overlap). Alla riga 6 viene semplicemente controllato se la taglia dell'overlap non includa anche il carattere *$*, e in tal caso è ridotta di uno (per le proprietà del carattere *$*, questo non farà più parte di nessuno degli overlap successivi con la stessa sequenza di riferimento).

Analizzando ora il tempo di costruzione del grafo, possiamo osservare che, benché il *lcp*, che ricordiamo essere un array lungo *total_size* (o anche *NM*), viene percorso linearmente, in esso una stringa comune può essere ripetuta per diversi elementi del suffix array: ad esempio, sempre considerando la Figura 3.3, il suffisso GCCGATCGAGAT$ e l'elemento che lo segue all'interno del suffix array, cioè GCCGATCGT$, presenteranno ognuno i due suffissi seguenti (nelle rispettive stringhe) da qualche parte nel suffix array. In questo caso i suffissi seguenti sono CCGATCGAGAT$ e CCGATCGT$. Questi a loro volta condividono il prefisso iniziale (CCGATCG) lungo 7, il quale, anche se non è un suffisso o un prefisso in nessuno dei due casi, potrebbe essere comunque analizzato. Dunque, ogni overlap trovato lungo il *lcp* viene considerato, sia che nel caso in cui è del tipo prefisso-suffisso, sia nel caso in cui non lo è, e, chiamando *t* la soglia sopra cui un overlap è accettabile, un segmento comune lungo *s* che rappresenta l'adiacenza più grande tra due read, genera altri *s* – *t* overlap inferiori accettabili, quindi $s - t + 1$ in tutto. Viene dunque aggiunto alla percorrenza lineare (*NM*) un tempo di calcolo derivato dalle adiacenze e dalle loro lunghezze; in particolare, ricordando che la soglia *t* in questo caso è $q + m - 1$, e chiamando A l'insieme delle adiacenze finali generate e |*a*| la taglia del più lungo overlap prefisso-suffisso usato per generare un'adiacenza, il tempo impiegato per la generazione del grafo mediante la tecnica 1 è $O(NM + \sum_{a \in A}(|a| - (q + m - 1) + 1))$, cioè $O(NM + \sum_{a \in A}(|a| - q - m + 2))$.
A livello di spazio, per la costruzione del grafo sono impiegati il suffix array, le read e il *lcp*. La memoria impiegata nei primi due è già stata illustrata nelle Formule 3.2, 3.3 e 3.4, mentre la memoria usata dal *lcp*, usando il ragionamento illustrato in appendice A, è

$$NM\lceil \log_2(M+1) \rceil \quad \text{bit} \quad , \qquad (3.)$$

E dunque la memoria totale impiegata è

$$NM\lceil \log_2(N) \rceil + NM\lceil \log_2(M) \rceil + NM\lceil \log_2(M+1) \rceil + 2NM + 16N \quad \text{bit} \quad . \qquad (3.7)$$

3.8 Tecnica 3: grafo matching massimali ottenuto mediante LCP

La supposizione fatta inizialmente allo sviluppo del progetto era che MataProb cercasse implicitamente, attraverso i q-mer, sottostringhe comuni di taglia maggiore di q (una sottostringa di taglia $k >= q$ contiene $k - q + 1$ q-mer, come è stato esposto nella Figura 3.2). Pertanto, si è provato a usare non solo gli overlap tra prefissi e suffissi delle sequenze, ma anche matching massimali tra sequenze, ovverosia generando un'adiacenza tra due read in base alla più lunga sottostringa che entrambe condividono, e lo score assegnato a tale adiacenza, sempre denominando *size_prefix* la taglia di tale overlap, sarà *size_prefix* - q + 1 (come nella tecnica precedente). Il codice è riportato nell'algoritmo 3.12, che è quasi uguale all'algoritmo 3.11, al

Algoritmo 3.12

```
01.  for k from 0 to total_size – 2 do
02.  {
03.     (num is, num ic) ← sa[k];
04.     num n ← k + 1;
05.     num size_prefix ← lcp[n];
06.     if ic + size_ prefix == reads[is].size() then
07.        size_prefix ← size_ prefix - 1;
08.     if size_prefix >= q + m - 1 then
09.     {
10.        id_seq_type ID1 ← getIDsSeq(k);
11.        seq sottostringa1 ← getSeqFromID(ID1);
12.        bool correct1 ← check_correct(k, size_prefix);
13.        if correct1 then
14.        {
15.           while size_prefix >= q + m - 1 and n < total_size do
16.           {
17.              id_seq_type ID2 ← getIDsSeq(n);
18.              bool correct2 ← check_correct(n, size_prefix);
19.              if ID1 != ID2 and correct2 then
20.              {
21.                 seq sottostringa2 ← getSeqFromID(ID2);
22.                 sottostringa1.AggiornaAdjVertice(sottostringa2, size_prefix - q + 1);
23.                 sottostringa2.AggiornaAdjVertice(sottostringa1, size_prefix - q + 1);
24.              }
25.              n ← n + 1;
26.              if n < total_size and lcp[n] < size_prefix then
27.                  size_prefix ← lcp[n];
28.           }
29.        }
30.     }
31.  }
```

quale però sono state rimosse le parti che controllano se gli overlap sono composti da prefissi e suffissi. Come vedremo questa tecnica ottiene spesso risultati simili a MetaProb, confermando così la supposizione fatta inizialmente al progetto. A livello temporale, il calcolo del grafo richiede lo stesso tempo riportato nella tecnica 2, dunque $O(NM + \sum_{a \in A}(|a| - q - m + 2))$, dove adesso A indica un insieme di lati più vasto, visto che la funzione ***AggiornaAdjVertice*** viene chiamata, diversamente dalla tecnica 2, non solo per le alle coppie prefisso-suffisso, ma in generale per tutti i matching maggiori o uguali a $q + m - 1$, risultando così in un tempo maggiore. A livello di spazio, invece, le strutture dati impiegate solo le stesse usate nella tecnica 2, e impiegheranno quindi lo stesso numero di bit, riportato nella formula 3.7.

Algoritmo 3.13

```
01.  for k from 0 to total_size – 2 do
02.  {
03.     (num is, num ic) ← sa[k];
04.     num n ← k + 1;
05.     num size_prefix ← lcp[n];
06.     id_seq_type ID1 ← getIDsSeq(k);
07.     seq seq1 ← getSeqFromID(ID1);
08.     if ic + q < reads[is].size() and correct_array[is][ic] then
09.     {
10.        while size_prefix >= q and n < total_size do
11.        {
12.           (num js, num jc) ← sa[n];
13.           id_seq_type ID2 ← getIDsSeq(n);
14.           if ID1 != ID2 and jc + q < reads[js].size() and correct_array[js][jc] then
15.           {
16.              seq seq2 ← getSeqFromID(ID2);
17.              seq1.AddAdjVertice(seq2, 1);
18.              seq2.AddAdjVertice(seq1, 1);
19.           }
20.           n ← n + 1;
21.           if n < total_size and lcp[n] < size_prefix then
22.              size_prefix ← lcp[n];
23.        }
24.     }
25.  }
```

3.9 Tecnica 4: grafo di MetaProb (m q-mer) ottenuto mediante LCP

La tecnica 4 (algoritmo 3.13) genera lo stesso grafo generato da MetaProb: sempre scorrendo linearmente il *lcp*, si osserva ora se l'overlap di taglia *size_prefix* è maggiore o uguale a q (riga 10); in tal caso, diversamente dalle tecniche precedenti, viene usata la funzione ***AddAdjVertice*** (**seq** *sequenza*, **num** *overlap*) che genera un'adiacenza di score 1 tra le due sequenze considerate, e, se questa adiacenza è già presente, incrementa il suo score di 1. Per rilevare la correttezza del q-mer che inizia nella posizione *ic* della sequenza *is* è sufficiente osservare il valore di *correct_array*[*is*][*ic*], senza invocare la funzione ***check_correct***. Poiché ogni lato *a* che viene generato contiene uno score equivalente al numero di q-mer condivisi, i quali sono percorsi uno ad uno, spesso più volte, il calcolo del grafo richiede ora, oltre alla solita componente lineare *NM* per scorrere il *lcp*, un tempo additivo $\sum_{a \in A} score_a$, dove $score_a$ indica lo score di un certo lato *a*. Il tempo totale risulta dunque essere $O(NM + \sum_{a \in A} score_a)$, dove adesso *A* indica l'insieme di lati ottenuto in questa tecnica (spesso più vasto degli insiemi ottenuti dalle tecniche precedenti). Lo spazio impiegato dalle strutture dati resta sempre lo stesso delle tecniche 2 e 3, richiedendo quindi il numero di bit riportato nella formula 3.7.

Capitolo 4

Risultati

Illustreremo ora i risultati derivati dall'utilizzo delle quattro nuove tecniche introdotte nel capitolo precedente, e li confronteremo con quelli ottenuti dalla vecchia versione di MetaProb dal punto di vista del tempo, del consumo di memoria e della qualità dei risultati (precision, recall e F-measure).
Successivamente evidenzieremo come si abbia un consumo di memoria minore per read corte rispetto alle read lunghe nelle nuove tecniche impiegate. Infine, illustreremo come variano il tempo e il consumo di memoria su un dataset reale a seconda delle sue dimensioni.

4.1 Dataset impiegati negli esperimenti

Per gli esperimenti sono stati impiegati 28 dataset (gli stessi usati in [5]): 25 di questi sono già stati utilizzati anche in [10], 2 contengono genomi sintetici basati su read reali, e uno è un campione metagenomico reale appartenente al Human Microbiome Project. I primi 25 dataset includono svariati metagenomi batterici simulati mediante MetaSim ([20]) e possono essere divisi in tre categorie: *S*, *L* ed *R*. *S* ed *L* sono composti da read paired-end corte, dove ognuna di queste è lunga al massimo 81 basi, e sono state generate secondo le statistiche di errore del software Illumina, usando un rate di errore del 1%. *L* è stato costruito utilizzando i genomi appartenenti a due specie, *Eubacterium eligens* e *Lactobacillus amylovorus*, mentre *S* contiene una differenziazione maggiore a livello di specie (fino a 30) e bilanciamento tra queste (il bilanciamento indica quante volte è più spesso presente una specie rispetto a un'altra). *R* contiene read Roche 454 single-end, lunghe da 873 basi a 972, con rate di errore di sequenziamento del 1%. I due dataset contenenti genomi sintetici derivati da read reali sono MiSeq_a1 e MiSeq_a2, e sono costituiti da read corte (massima lunghezza di 102 basi) usate anche in Kraken ([7]). L'ultimo dataset, SRR1804065, generato usando tecnologia Illumina, contiene read paired-end di lunghezza massima di 101 basi. Nel paper originale ([5]) si era usato solo un sottoinsieme di questo set, ottenuto mediante BLAST, poiché la versione iniziale era troppo grande. Nell'esperimento attuale, invece, utilizzeremo varie dimensioni di SRR1804065, per confrontare le versioni vecchie e nuove di MetaProb in relazione a dataset reali. Illustreremo ora i risultati sui primi 27 dataset (il dataset SRR1804065 sarà considerato nell'ultimo paragrafo).

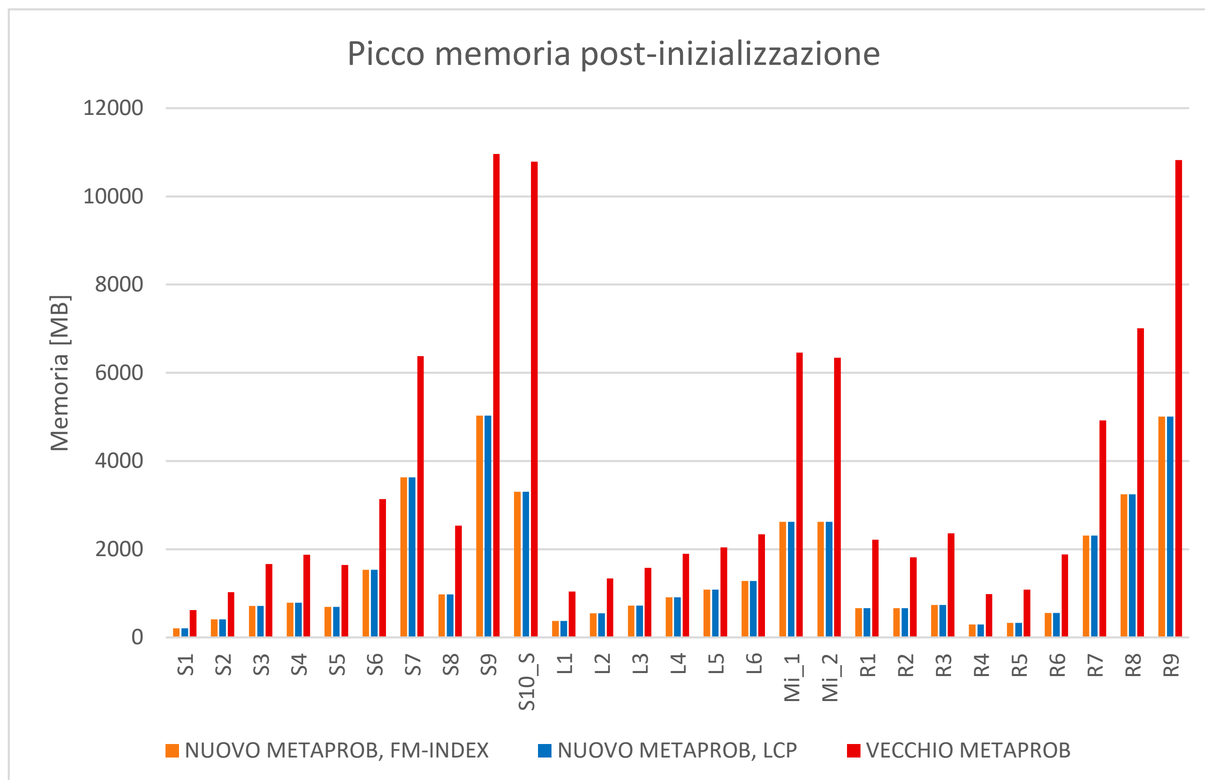


Figure 4.1. *Picco della memoria dopo la generazione delle strutture dati per la costruzione del grafo*

4.2 Tempo e spazio di inizializzazione

Con inizializzazione si intende la fase che ottiene le strutture dati che verranno utilizzate per la costruzione del grafo. Nel caso della vecchia versione di MetaProb queste consistono solo nella **unordered_map** *MapHash*, descritta nel Capitolo §1. Nel caso nella nuova tecnica 1 le strutture rilevanti sono l'insieme delle sequenze *reads*, il suffix array, gli array di conteggio compressi e la *BWT*. Infine, per le tecniche 2, 3 e 4 sono impiegati l'insieme delle sequenze *reads*, il suffix array, e il *lcp*. Per la tecnica 1 si è scelto di usare un fattore di compressione 5 per gli array di conteggio. Dal punto di vista spaziale, come evidenziato in Figura 4.1 (dove le tecniche 2, 3 e 4 sono state raggruppate in un solo risultato, la colonna in blu, poiché, nell'inizializzazione, ottengono le stesse strutture dati eseguendo le stesse operazioni), i picchi di memoria raggiunti per le costruzioni delle strutture dati in tutte e 4 le nuove tecniche sono uguali: ciò è dovuto al fatto che i picchi raggiunti nelle nuove tecniche sono ottenuti durante la costruzione nel suffix array, a causa di tutti gli altri array (booleani e numerici) che vengono impiegati in questa fase, descritti nel capitolo §2. Il picco resta comunque assai inferiore (approssimativamente la metà) rispetto alla versione vecchia di Metaprob.
Una volta eliminate le strutture dati superflue, restano solo quelle che saranno impiegate per la costruzione del grafo nella fase successiva, e la memoria che queste impiegano è evidenziata in Figura 4.2. Come è possibile osservare, queste strutture impiegano ancora meno spazio rispetto

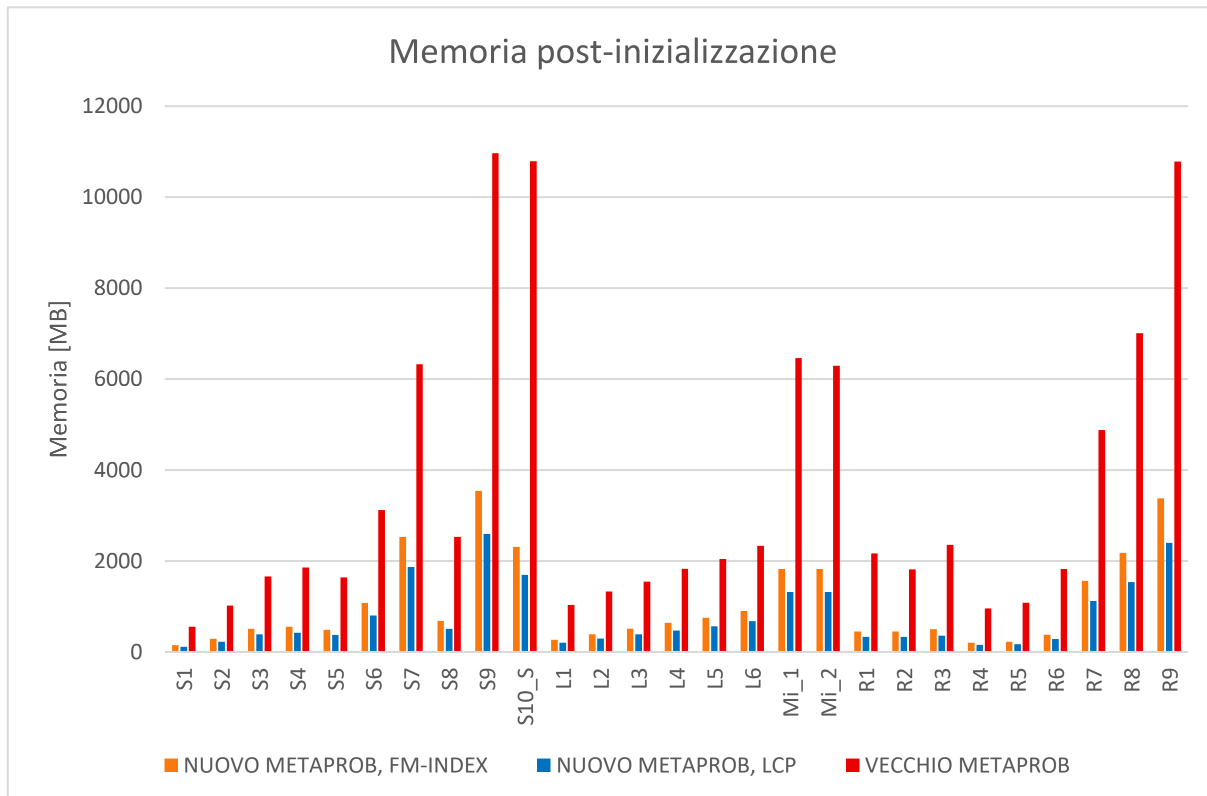


Figure 4.2. *Memoria occupata dopo la generazione delle strutture dati per la costruzione del grafo*

a quelle impiegate nella versione precedente; facciamo notare tuttavia che nella versione precedente di MetaProb vengono usate spesso strutture complesse (come le **unordered_map**) la cui cancellazione spesso non liberava tutta la memoria che impiegavano (sarà evidente in uno dei grafi successivi). Come già detto, il fattore di compressione impiegato nella tecnica 1 per gli array di conteggio è 5. Se fosse impiegato un fattore maggiore, si potrebbe ottenere un impiego di memoria da parte delle strutture dati minore rispetto a tutte le altre tecniche impiegate (ciò è stato sfruttato in [15] e [16]), ma questo aumenterebbe il tempo di costruzione del grafo nella fase successiva, che già risente del fattore di compressione attuale.
Il tempo impiegato in questa fase è riportato in Figura 4.3. Come spiegato in precedenza, purtroppo l'algoritmo impiegato per la costruzione del suffix aray non è puramente lineare, ma presenta un fattore moltiplicativo logaritmico che dipende dalla taglia massima delle read impiegate. Essendo però tale fattore spesso limitato, si può considerare il tempo quasi lineare, ma con un fattore moltiplicativo maggiore rispetto alla versione precedente di MetaProb.

4.3 Tempo e spazio per la costruzione del grafo

Il picco di memoria impiegato per la costruzione del grafo resta quasi lo stesso della fase precedente (i picchi non sono resettati a ogni fase, si considerano sempre partendo dalla fase precedente). Questi sono riportati nel grafico in Figura 4.4. Solo in alcuni casi si ha una memoria maggiore, dovuta a una dimensione del grafo non trascurabile.

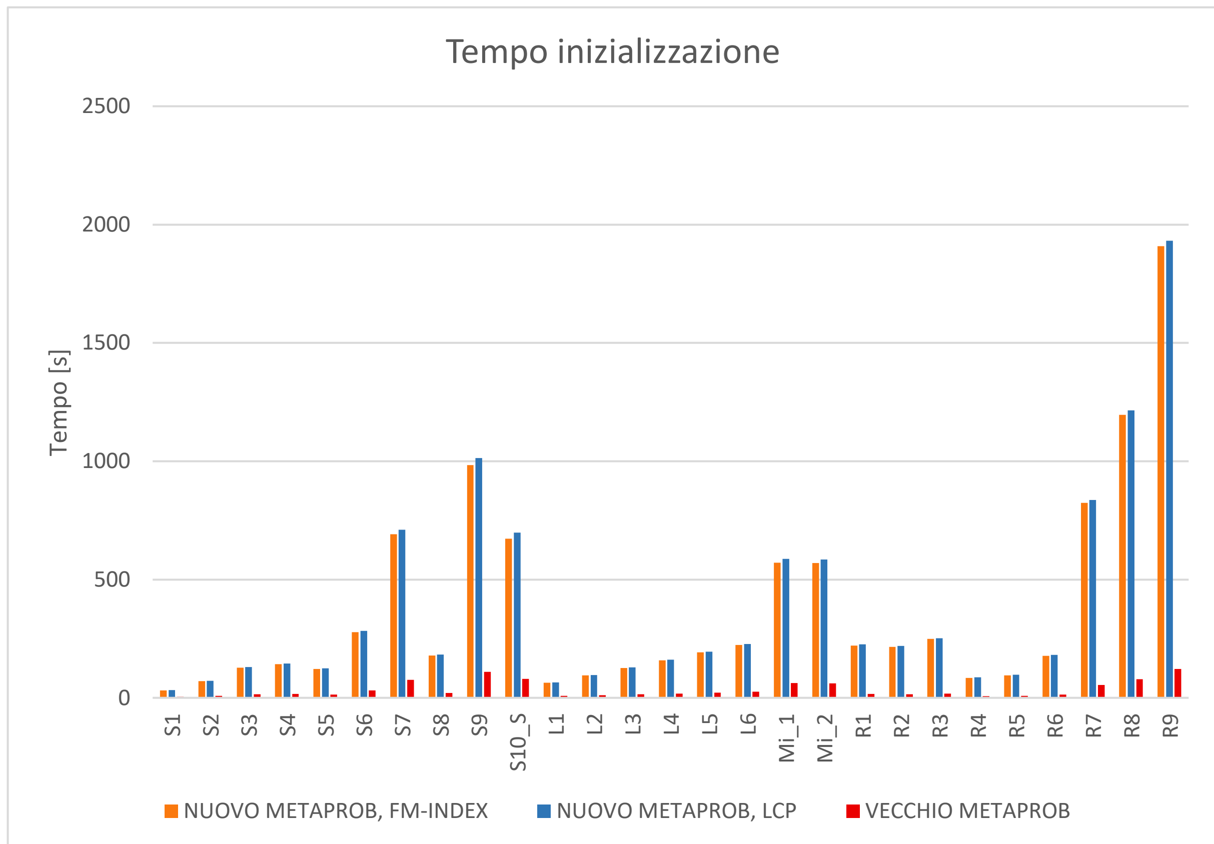


Figure 4.3. *Tempo impiegato per la generazione delle strutture dati per la costruzione del grafo*

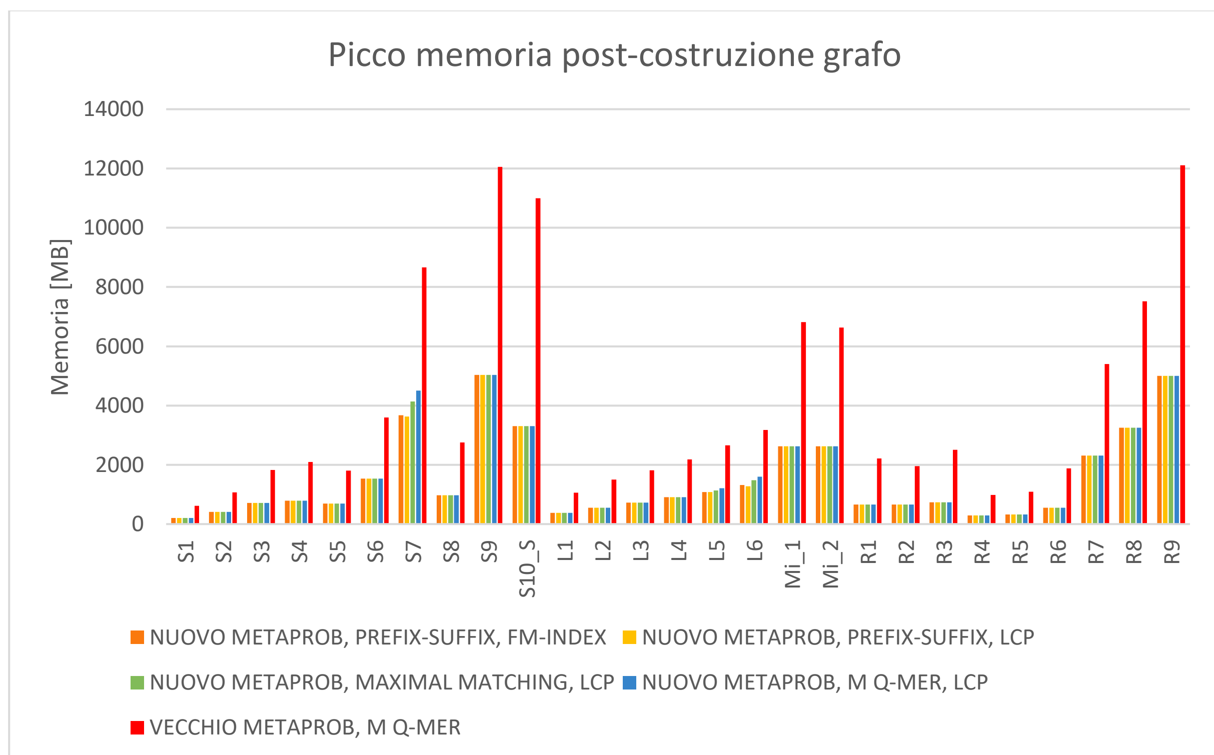


Figure 4.4. *Picco della memoria dopo la costruzione del grafo*

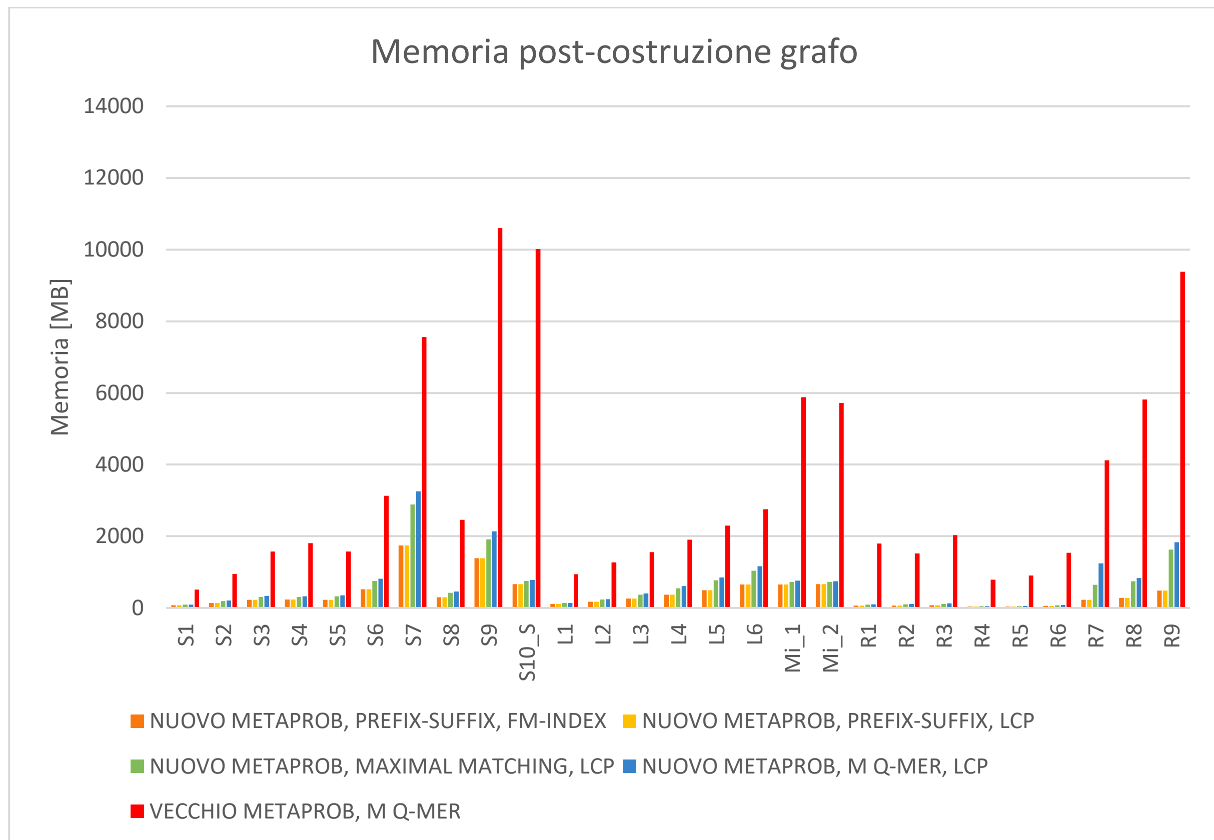


Figure 4.5. *Memoria occupata dopo la costruzione del grafo*

Una volta costruito il grafo, vengono eliminate tutte le altre strutture dati impiegate (suffix array, *lcp*, array di conteggio…), lasciando solo la memoria impiegata dal grafo. Come si osserva dalla Figura 4.5, le tecniche 1 e 2 (in arancione e in giallo) impiegano la stessa memoria poiché generano lo stesso grafo. Lo stesso ci si aspetterebbe dalla tecnica 4 (in blu) e dalla vecchia versione di MetaProb (in rosso), dato che entrambe costruiscono il medesimo grafo (*m q*-mer), ma poiché, come già osservato in precedenza, quest'ultima non riesce a liberare in maniera ottimale lo spazio impiegato dalle sue strutture dati, non si ha la stessa memoria impiegata risultante dalla tecnica 4.
Il tempo impiegato, riportato in Figura 4.6, varia ovviamente a seconda del tipo di grafo e delle strutture dati impiegate per ottenerlo. In particolare, si può osservare come il tempo impiegato per la tecnica 1 (in arancione) sia maggiore degli altri soprattutto a causa del fattore di compressione 5 impiegato. Se fosse stato maggiore, nonostante un miglioramento della memoria (ma non dei picchi) si sarebbe incrementato notevolmente il tempo di calcolo del grafo.
Nelle Tabelle 4.1, 4.2, 4.3 e 4.4 sono riportati i guadagni in termini di spazio e tempo ottenuti alla fine della generazione del grafo dalle quattro nuove tecniche introdotte rispetto alla vecchia versione di MetaProb. È possibile osservare come la media dello spazio impiegato sia da read corte che lunghe sia inferiore al 40% della memoria occupata dalla versione originale del software, mentre il tempo purtroppo arriva a essere fino 5 volte superiore nelle tecniche 2, 3 e 4, e quasi 6 volte superiore nella tecnica 1.

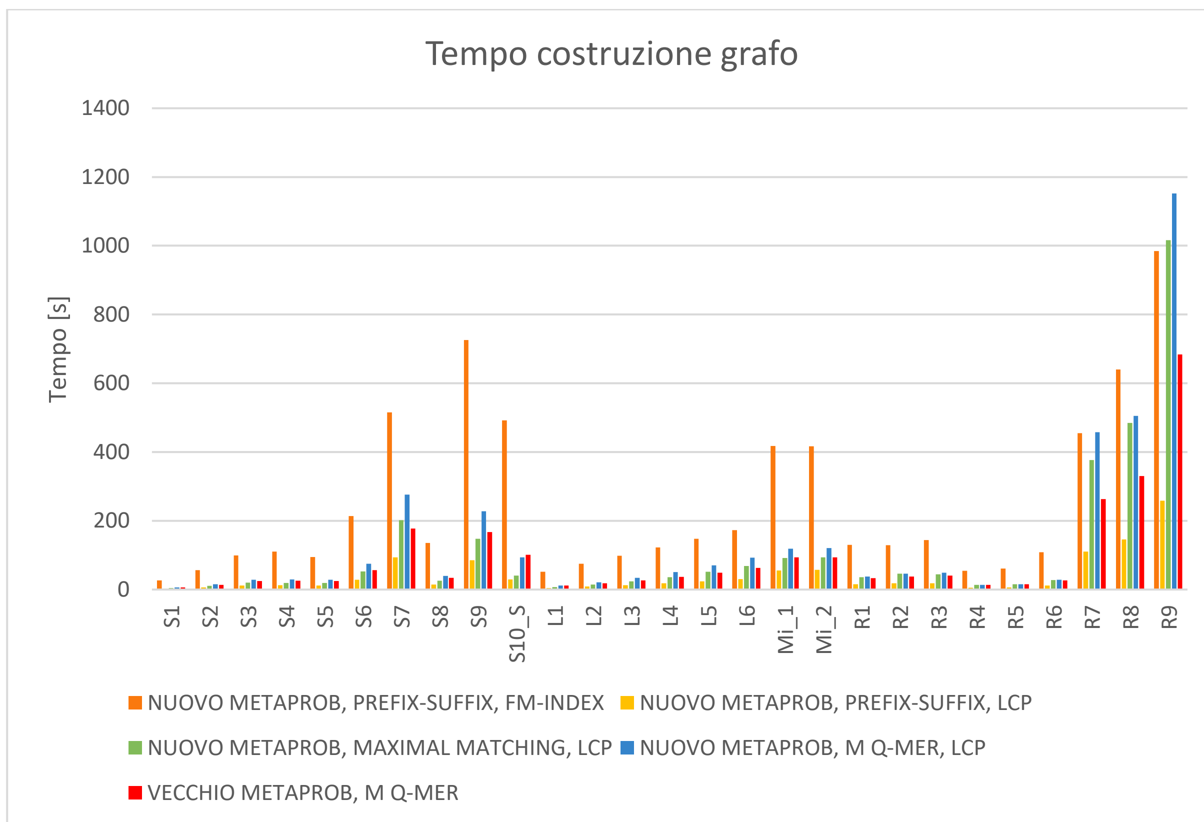


Figure 4.6. *Tempo impiegato per la costruzione del grafo*

Guadagno spazio picco post-grafo, read corte [%]	Tecnica 1	Tecnica 2	Tecnica 3	Tecnica 4
	38.46968	38.37667	39.15932	39.7796

Tabella 4.1. *Guadagno di memoria delle nuove tecniche, su read corte, rispetto alla versione originale di MetaProb*

Guadagno spazio picco post-grafo, read lunghe [%]	Tecnica 1	Tecnica 2	Tecnica 3	Tecnica 4
	34.4539	34.4539	34.4539	34.4539

Tabella 4.2. *Guadagno di memoria delle nuove tecniche, su read lunghe, rispetto alla versione originale di MetaProb*

Guadagno tempo iniz. + grafo, read corte [%]	Tecnica 1	Tecnica 2	Tecnica 3	Tecnica 4
	564.1652	356.57	380.7742	404.2107

Tabella 4.3. *Guadagno della somma dei tempi di inizializzazione e costruzione del grafo, su read corte, rispetto alla versione originale di MetaProb*

Guadagno tempo iniz. + grafo, read corte [%]	Tecnica 1	Tecnica 2	Tecnica 3	Tecnica 4
	585.8545	406.3218	464.9024	470.9947

Tabella 4.4. *Guadagno della somma dei tempi di inizializzazione e costruzione del grafo, su read lunghe, rispetto alla versione originale di MetaProb*

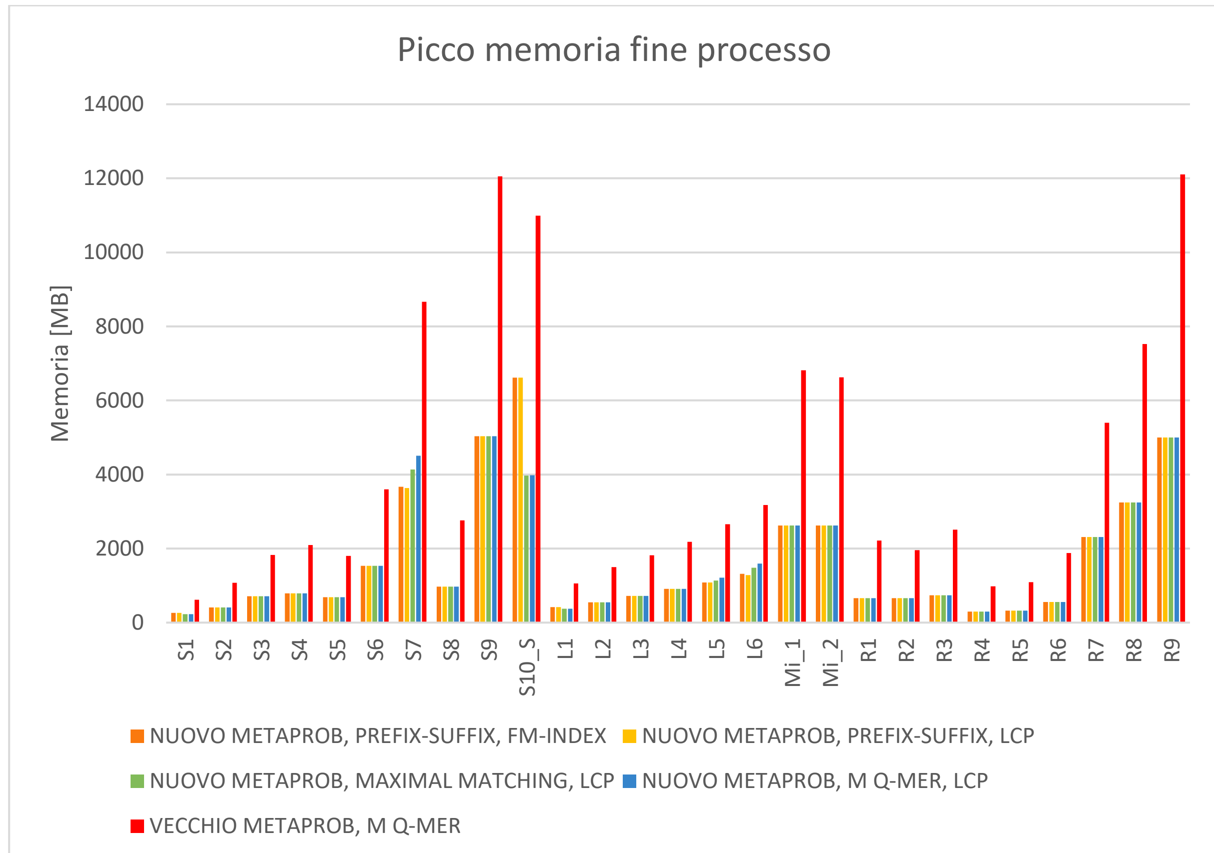


Figure 4.7. *Picco della memoria alla fine del processo*

4.4 Tempo e spazio a fine processo

Una volta ottenuto il grafo, MetaProb prosegue nell'esecuzione fino a completamento del processo. Ricordiamo che non sono state modificate le fasi successive alla costruzione del grafo. Il picco di memoria impiegata a fine processo è riportato in Figura 4.7. nel caso delle 4 nuove tecniche introdotte in MetaProb, questo resta spesso simile al picco raggiunto nella costruzione del suffix array, che, come già detto, rappresenta frequentemente la parte più dispendiosa del processo a livello di memoria. Tale picco si mantiene comunque bel al di sotto della soglia raggiunta dalla vecchia versione di MetaProb.
A livello di tempo purtroppo il processo totale dura più della versione originale di MetaProb, ma si mantiene comunque su una soglia non troppo distante dalla vecchia versione, come è possibile osservare dalla Figura 4.8.

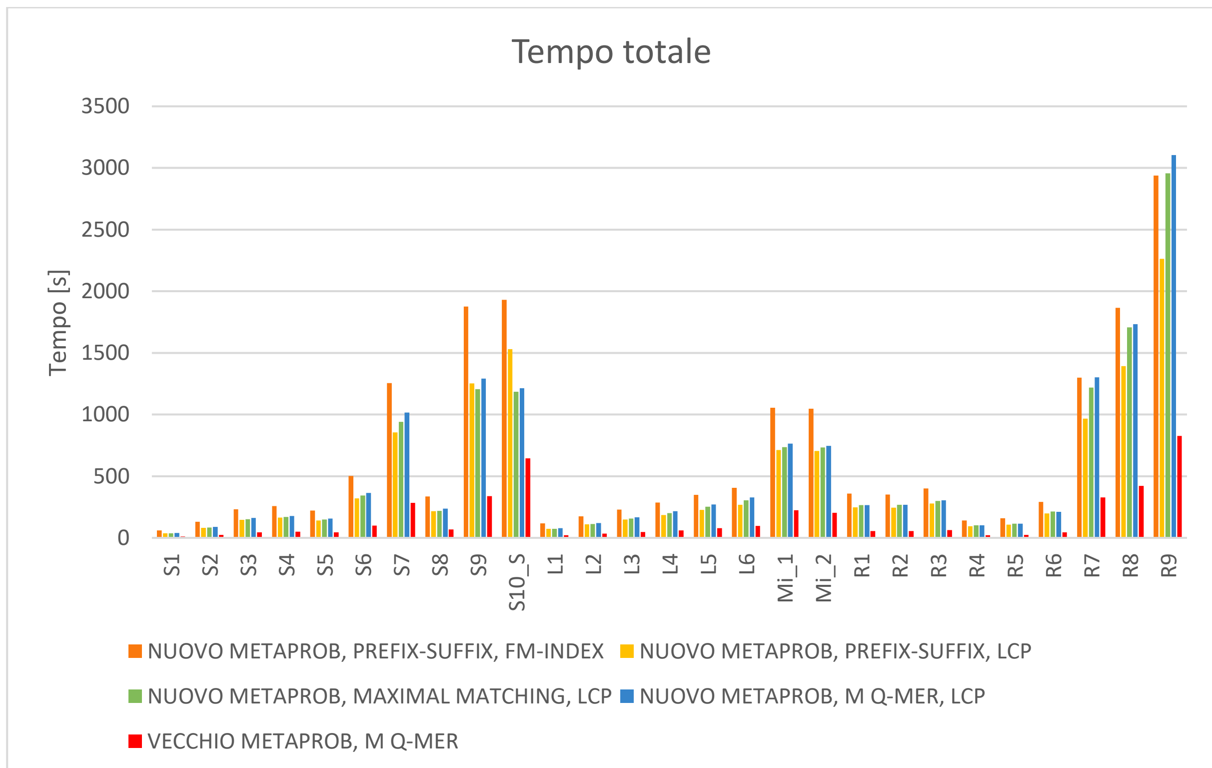


Figure 4.8. *Tempo impiegato per la conclusione del processo*

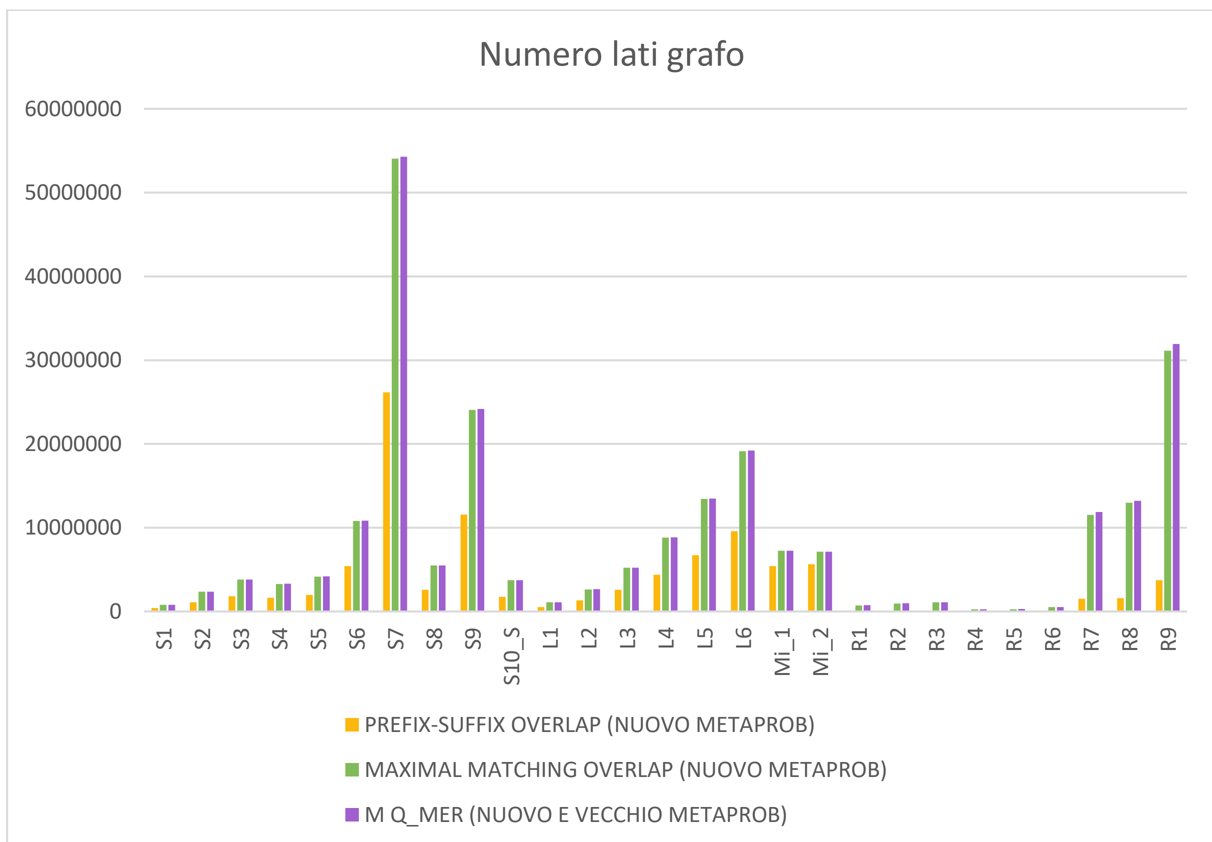


Figure 4.9. *Lati ottenuti dai diversi tipi di grafo*

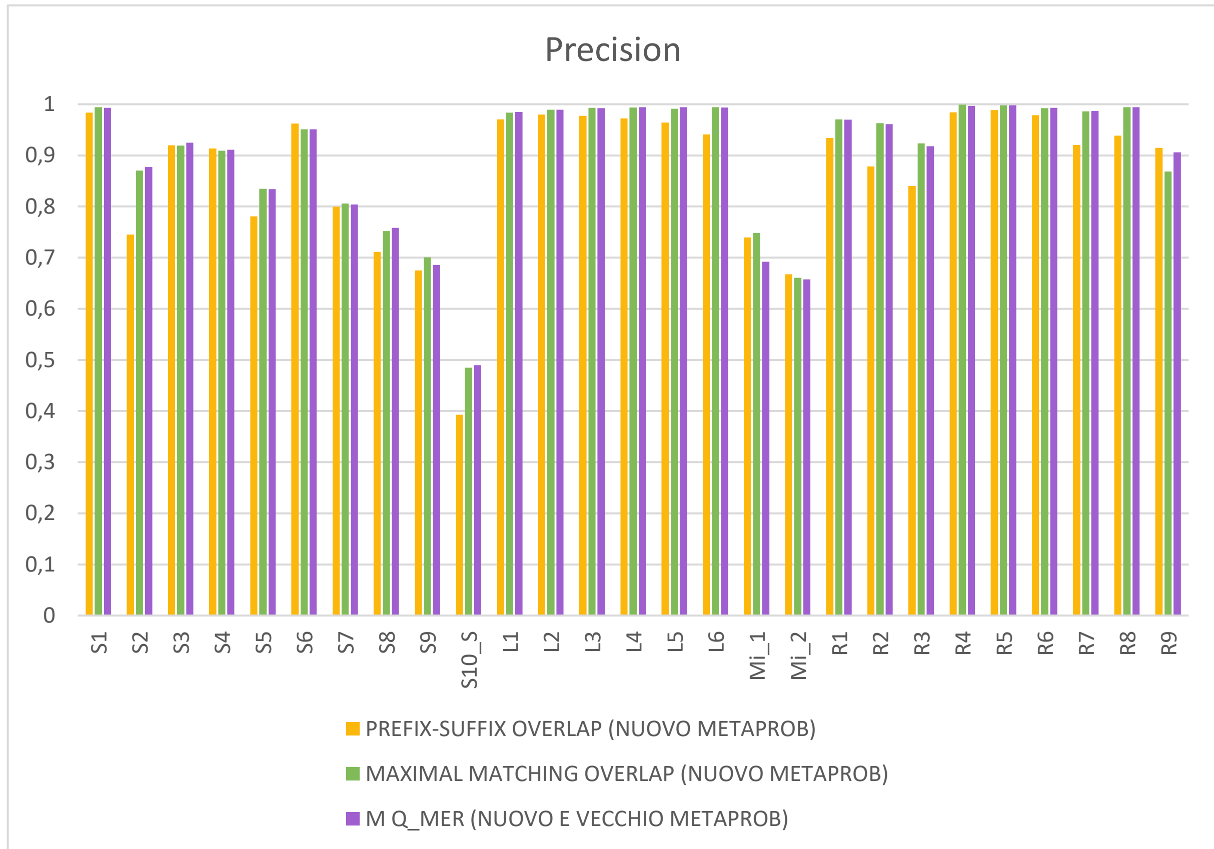


Figure 4.10. *Precision ottenuta dai diversi tipi di grafo*

4.5 Precision, Recall e F-measure

A questo punto la qualità dei risultati ottenuti dipende strettamente dal tipo di grafo impiegato (suffisso-prefisso, matching massimale o *m q*-mer): è interessante osservare, nella Figura 4.9, la similarità tra il numero di lati presenti nel grafo ottenuto con i matching massimali (tecnica 3) e con gli *m q*-mer (tecnica 4 e versione originale di MetaProb). Questo conferma l'ipotesi fatta inizialmente, cioè che in realtà Metaprob, mediante la ricerca di *q*-mer condivisi, spesso cerchi in realtà la più grande stringa contivisa tra due read.
Per la precision e la recall sono state usate le seguenti formule (le stesse usate in [5] e in [10]):

$$Precision = \frac{\sum_{i=1}^{C} \max_j A_{ij}}{\sum_{i=1}^{C} \sum_{j=1}^{n} A_{ij}} \quad , \qquad (4.1)$$

$$Recall = \frac{\sum_{j=1}^{n} \max_i A_{ij}}{\sum_{i=1}^{C} \sum_{j=1}^{n} A_{ij} + \#read_non_assegnate} \quad , \qquad (4.2)$$

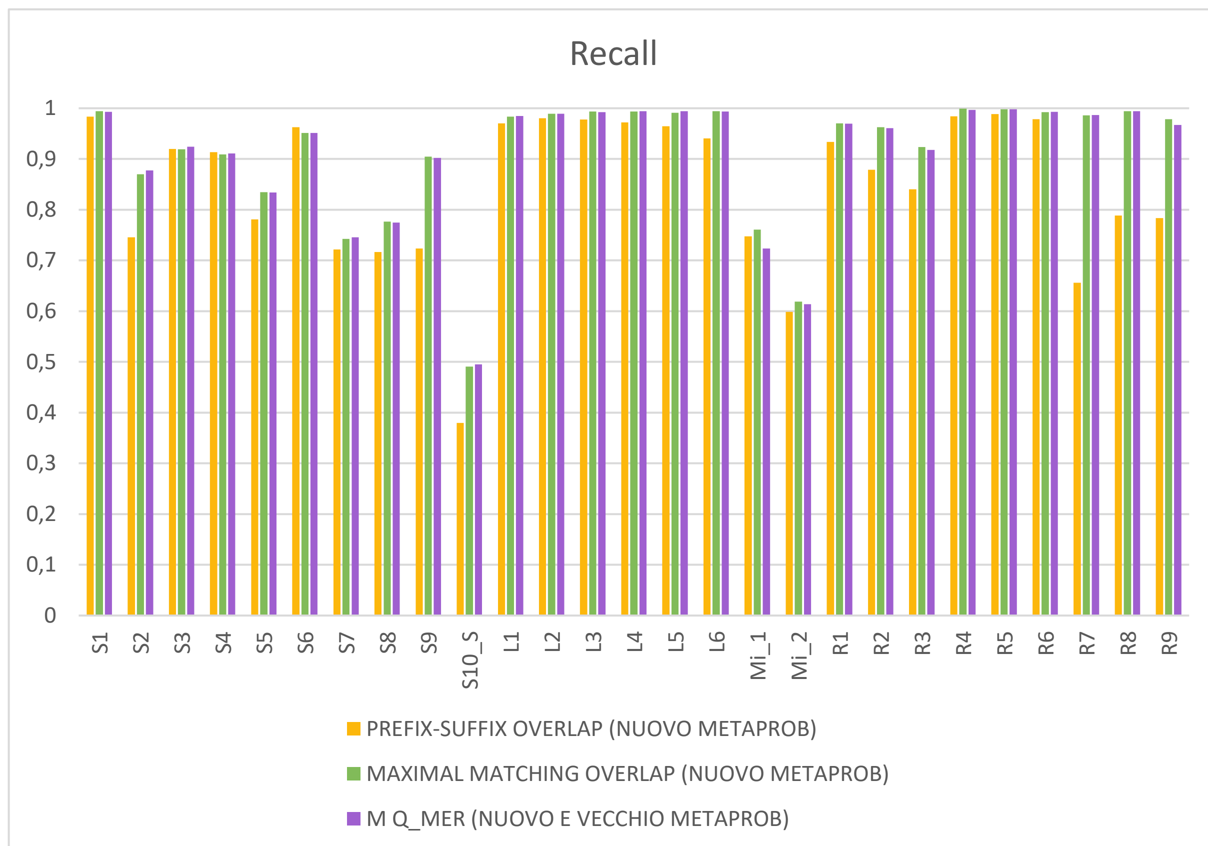


Figure 4.11. *Recall ottenuta dai diversi tipi di grafo*

dove A_{ij} indica il numero di read della specie j assegnate al cluster i, C è il numero di cluster ottenuti, e n è il numero di specie nel campione metagenomico.
L'F-measure è invece ottenuta mediante la seguente formula

$$F - measure = \frac{2 * precision * recall}{precision + recall} \quad , \qquad (4.3)$$

Come si può osservare dalle Figure 4.10, 4.11 e 4.12, i grafi basato su matching massimali e *m q*-mer ottengono dei valori molto simili, mentre i grafi basati su lati ottenuti da overlap di prefissi-suffissi, riducendo il numero di archi validi rilevanti, riducono anche la qualità dei risultati.

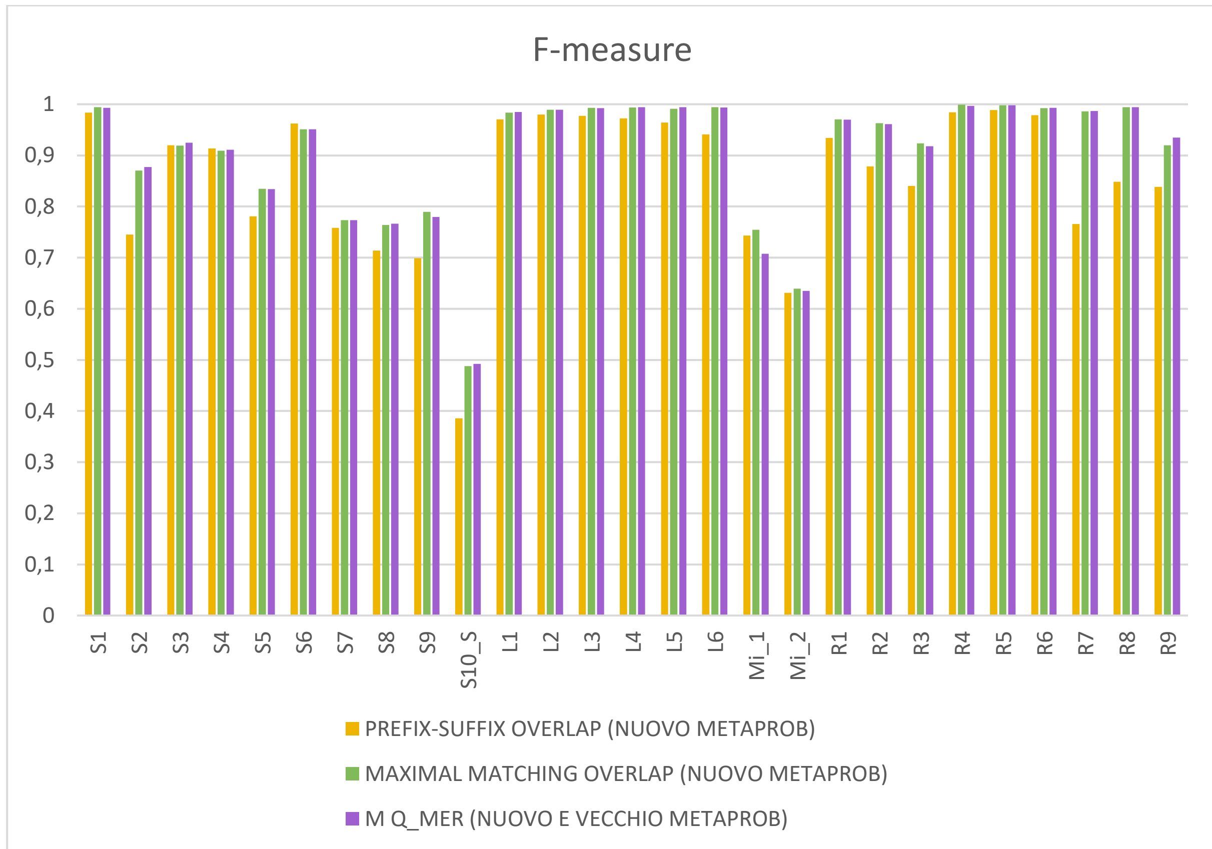


Figure 4.12. *F-measure ottenuta dai diversi tipi di grafo*

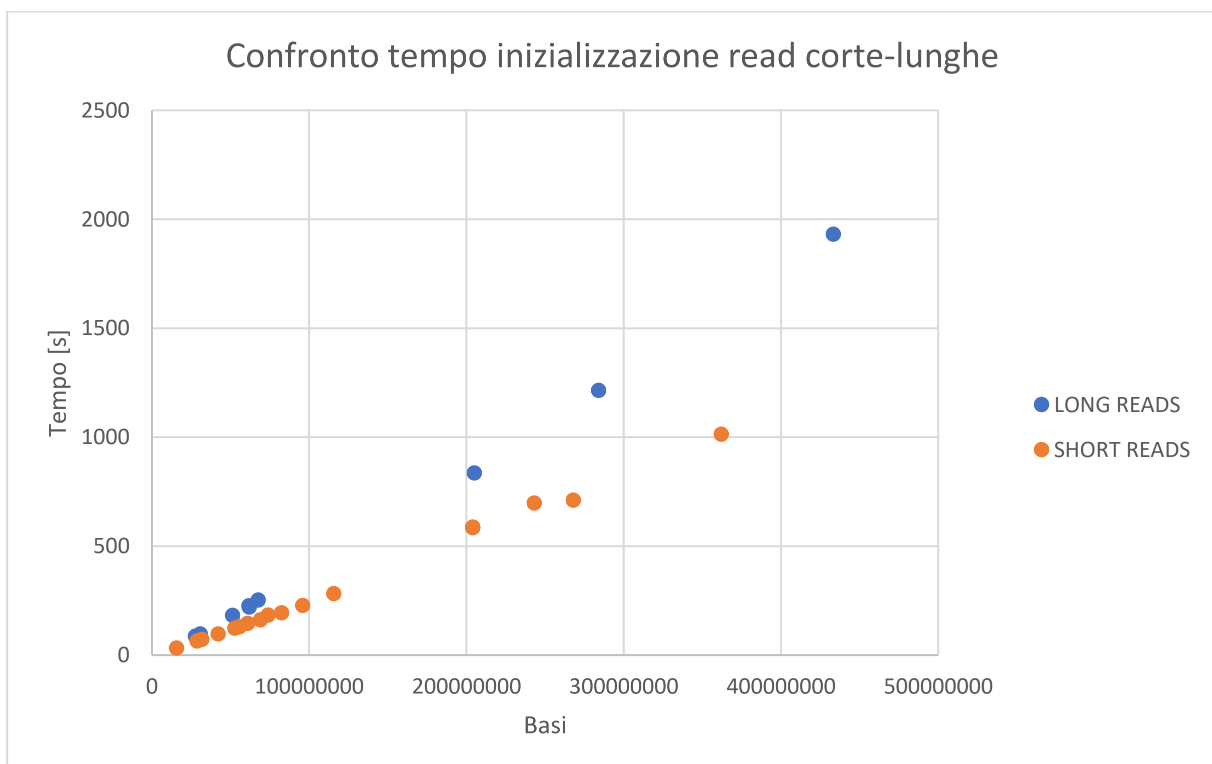


Figure 4.13. *Confronto dei tempi di inizializzazione (calcolo delle strutture dati per la costruzione del grafo) tra read corte e lunghe*

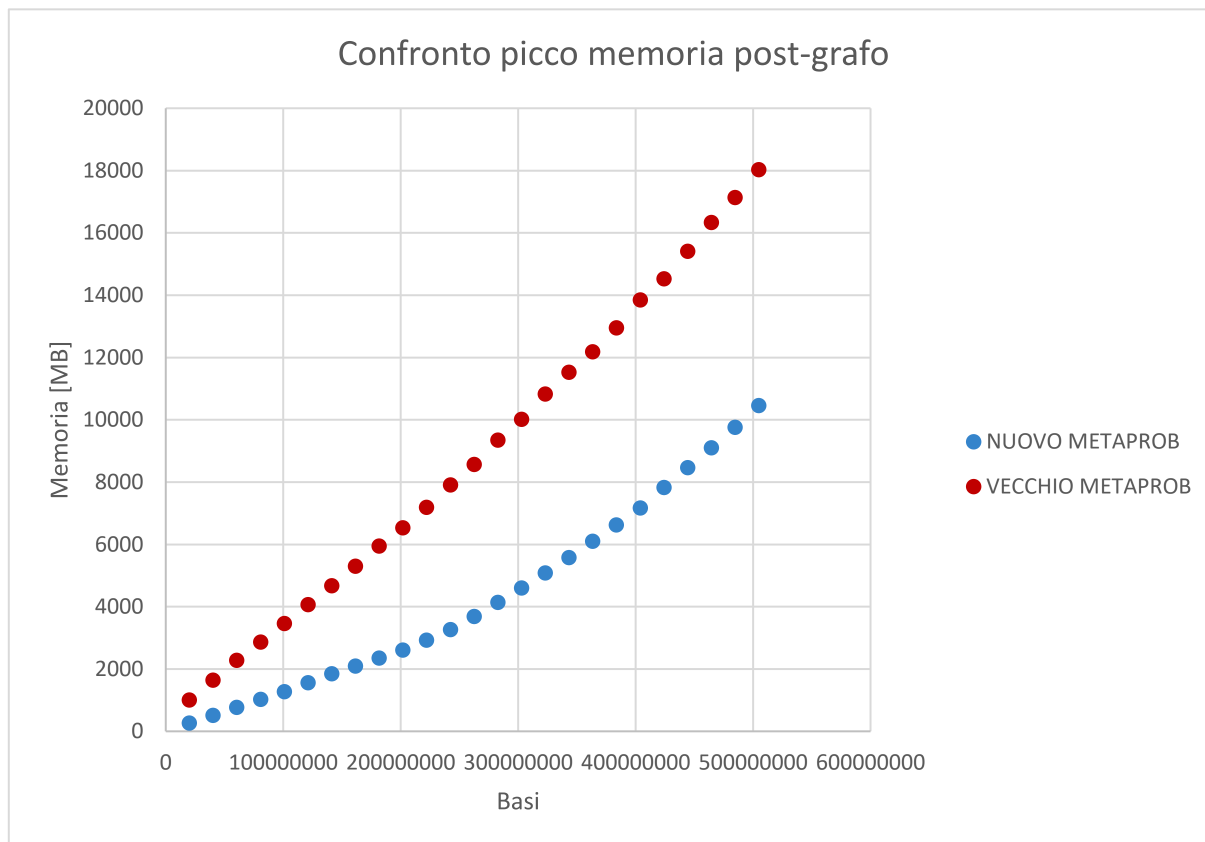


Figure 4.14. *Confronto dei picchi di memoria raggiunti, dopo la costruzione del grafo, grafo su un dataset reale tra la versione originale di MetaProb e la quarta tecnica della nuova versione*

4.6 Costruzione del suffix array per read corte e lunghe

È interessante osservare come il tempo impiegato per la costruzione del suffix array migliori per read corte: ricordiamo infatti che il tempo impiegato per la costruzione del suffix array è $O(total_size \log size_max)$, e tale regola è osservabile in Figura 4.13 (in blu sono le read lunghe, cioè le read appartenenti al dataset R, in arancione tutte le altre).

4.7 Tempo e spazio per un dataset reale

Per testare l'efficienza delle nuove tecniche di MetaProb, per il calcolo del grafo di adiacenza, si è utilizzato il dataset metagenomico reale SRR1804065. Questo è un campione considerevolmente grande (più di 43 milioni di read), che è stato campionato in vari sottoinsiemi di grandezza ridotta per poter verificare l'efficienza temporale e spaziale al crescere della dimensione del dataset. Essendo questo un dataset reale, presenta dimensioni del grafo assai maggiori dei casi simulati, ma, come si può osservare dalla Figura 4.14, le nuove tecniche di MetaProb riescono comunque a mantenere un consumo di memoria ridotto rispetto alla versione originale. Nella Figura 4.15, invece, si nota l'incremento di tempo rispetto alla vecchia versione, restando comunque entro un limite quasi lineare (con fattore moltiplicativo logaritmico quasi costante, come visto precedentemente), e dunque mantenendo una relativa

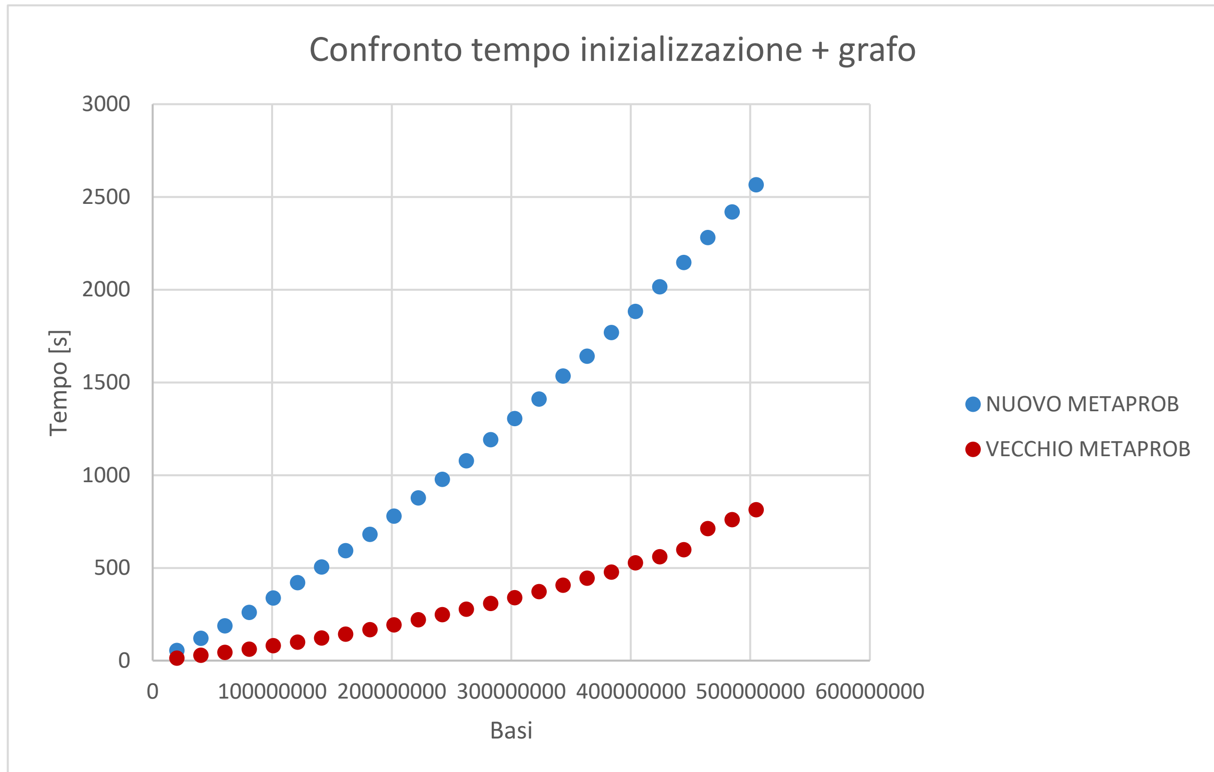


Figure 4.15. *Confronto della somma dei tempi di inizializzazione e costruzione del grafo su un dataset reale tra la versione originale di MetaProb e la quarta tecnica della nuova versione*

efficienza temporale. Essendo un campione metagenomico reale, non si hanno a disposizione i dati relativi alle effettive specie di appartenenza, e non è quindi possibile calcolare la precision, la recall e F-measure.

In Figura 4.16 e 4.17 è possibile osservare i guadagni di spazio e tempo, ottenuti come rapporto tra i valori della nuova versione (usando la tecnica 4) e della versione originale di MetaProb riportate nei grafi precedenti. Come si può vedere, nei casi analizzati, è impiegato meno del 60% della memoria della versione precedente. In media è usato il 44,1% della memoria di MetaProb originale. Benché il consumo di RAM aumenti al crescere della dimensione del dataset, questo aumento tenderà a diventare meno considerevole, poiché è dovuto soprattutto all'incremento del numero di bit impiegati per rappresentare i numeri nelle strutture illustrate in Appendice A. Più questo numero di bit aumenta, dunque, più aumenta anche la quantità di numeri rappresentabili, diminuendo così la necessità di aggiungere bit rispetto al dataset precedente. il tempo impiegato, purtroppo, aumenta anche di 4 volte rispetto alla versione originale (in media il 385,85%). Tuttavia, questo aumento sembra calare al crescere del dataset, forse a causa del differente incremento di tempo dovuto, da un lato, all'inizializzazione delle strutture dati impiegate per il calcolo del grafo, e dall'altro, al calcolo del grafo stesso.

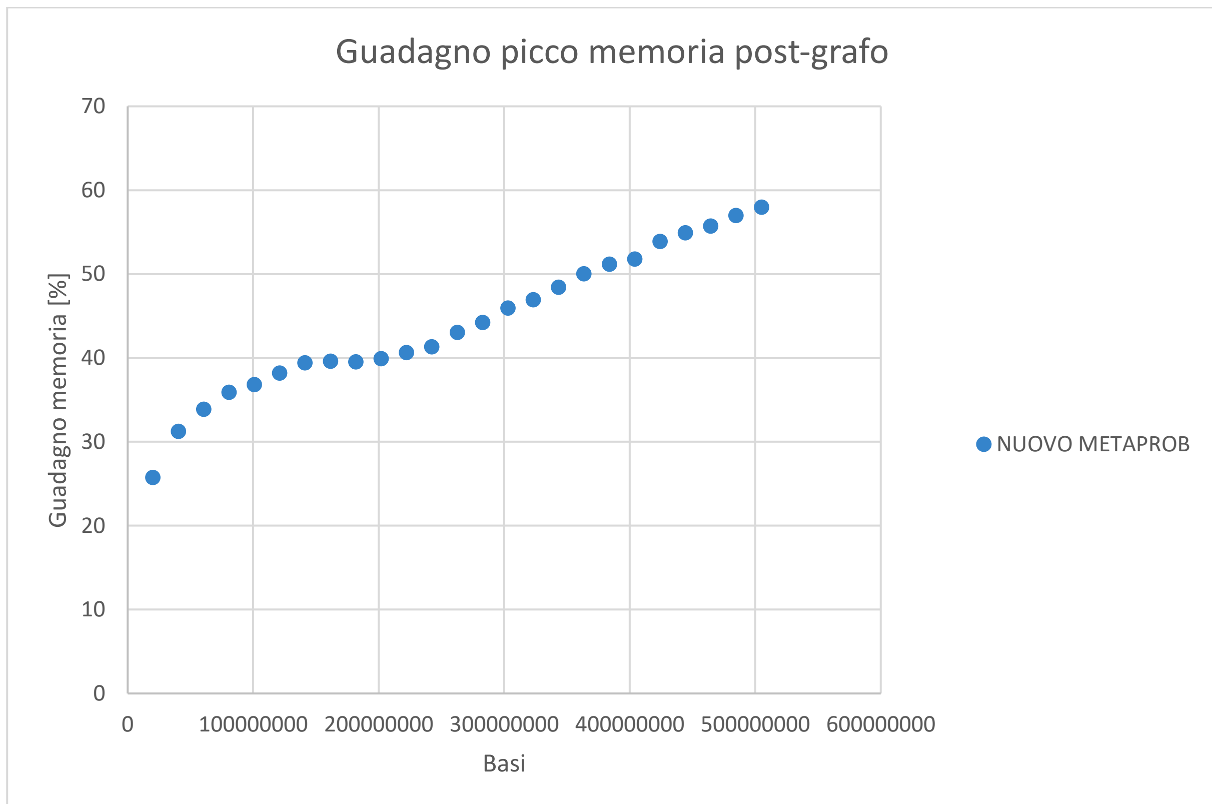


Figure 4.16. *Guadagno del picco di memoria, dopo la costruzione del grafo, ottenuto dalla quarta tecnica della nuova versione rispetto alla versione originale di MetaProb*

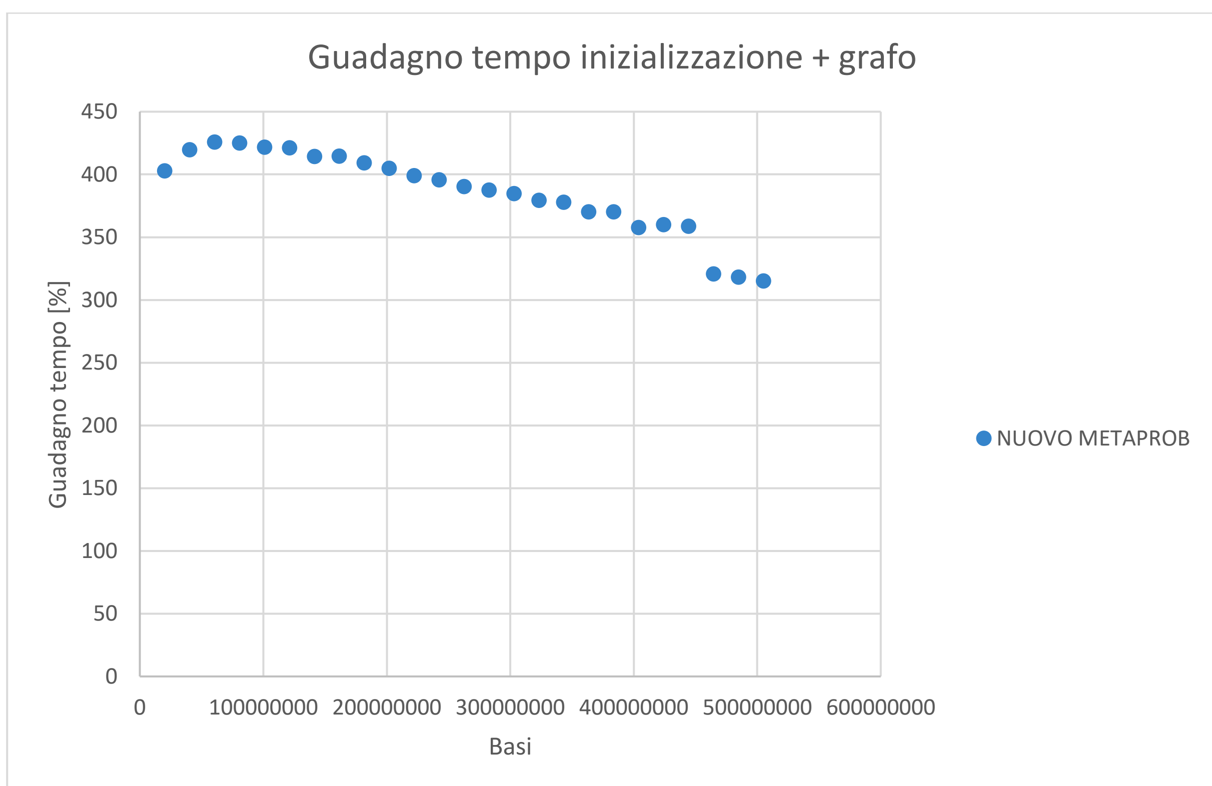


Figure 4.17. *Guadagno della somma dei tempi di inizializzazione e di costruzione del grafo, ottenuto dalla quarta tecnica della nuova versione rispetto alla versione originale di MetaProb*

Capitolo 5

Conclusioni

Effettueremo ora delle considerazioni sui risultati ottenuti e illustreremo eventuali lavori futuri.

5.1 Considerazioni sui risultati e sul progetto

È stato mostrato, nei capitoli precedenti, come in tutte le quattro nuove tecniche introdotte si sia raggiunta un'efficienza spaziale superiore rispetto alla vecchia versione di MetaProb, aumentando però il consumo temporale, ma restando sotto una soglia accettabile in tale incremento (difficilmente si può trovare un algoritmo migliore della versione originale per quanto riguarda il tempo di esecuzione).

L'idea da cui è partito il progetto, riportata in [15] e [16] (e in paper pubblicati precedentemente come [21] e [22]), sfruttava le potenzialità del FM-index nel calcolare gli overlap suffisso-prefisso. In particolare, dati il FM-index, l'intervallo $[b_{\omega}, e_{\omega}]$ della stringa ω e l'intervallo $[b_{\omega\$}, e_{\omega\$}]$ della stringa $\omega\$$ (spiegati nel capitolo §3), questi consentono di calcolare in tempo $O(1)$ il calcolo di dati come: il numero di ricorrenze della sottostringa ω, il numero di ricorrenze del suffisso ω, e il numero di ricorrenze dei prefissi ω. Questa rapidità delle operazioni, unita allo spazio estremamente ridotto impiegato dal FM-index (dato che sono da rappresentare 5 simboli, cioè *$*, *A*, *C*, *G*, e *T*, sono sufficienti 3 bit per ogni base, e quindi in tutto *total_size**3 bit), fornisce un ottimo punto di partenza per creare algoritmi di ricerca su stringhe, adattandoli alla metagenomica.

5.2 Eventuali lavori futuri

L'efficienza spaziale del FM-index descritta nel Paragrafo precedente, però, si basa sul fatto che si ha già a disposizione il FM-index, che normalmente viene ottenuto mediante il suffix array, ma per evitare il picco di memoria raggiunto in questo calcolo, in [15] e [16] vengono impiegati algoritmi per il calcolo del FM-index in memoria secondaria, e lo stesso string graph che ne viene calcolato non è tenuto in memoria principale ([23] e [24]). Tali algoritmi puntato a limitare il numero di accessi alla memoria secondaria effettuati in questi calcoli per velocizzare il processo. Si potrebbe pensare, per eventuali lavori futuri, di implementare algoritmi che usino memoria secondaria, o, più semplicemente, poiché nella maggior parte dei casi il picco di memoria raggiunta è dovuto all'algoritmo per la costruzione del suffix array, per eventuali lavori futuri si potrebbe pensare di sostituire tale algoritmo con uno più efficiente dal punto di vista spaziale.

SUFFIX ARRAY (STRINGA, SUFFISSO)	LCP	SUFFISSO
.	.	.
.	.	.
.	.	.
(825, 25)	.	GCCGACCGAGAT$
(147, 28)	8	GCCGACCGT$
(9587, 28)	7	GCCGACCTT$
(54, 31)	5	GCCGAG$
(578, 31)	7	GCCGAG$
(429, 27)	6	GCCGAGAAGT$
(368, 29)	7	GCCGAGAT$
(525, 33)	4	GCCGT$
(84, 26)	2	GCGATTCGAGT$
(525, 28)	1	GGTATGCCGT$
.	.	.
.	.	.
.	.	.

→

SUFFIX ARRAY (STRINGA, SUFFISSO)	LCP	SUFFISSO
.	.	.
.	.	.
.	.	.
(825, 25)	.	GCCGACCGAGAT$
(147, 28)	8	GCCGACCGT$
(9587, 28)	7	GCCGACCTT$
(429, 27)	5	GCCGAGAAGT$
(368, 29)	7	GCCGAGAT$
(84, 26)	2	GCGATTCGAGT$
(525, 28)	1	GGTATGCCGT$
.	.	.
.	.	.
.	.	.

Figure 5.1. *Esempio di riduzione del suffix array e del lcp*

Un'altra tecnica che potrebbe ottimizzare sia il tempo che lo spazio sarebbe l'impiego di un suffix array parziale: per esempio, prendendo la tecnica 4, all'interno del suffix array i suffissi di lunghezza minore di q (o anche uguali a q, se consideriamo anche il simbolo *$* alla fine della read) non influenzano il risultato. Purtroppo, durante la sua costruzione, vengono comunque richiesti tutti i suoi suffissi, altrimenti l'algoritmo di costruzione del suffix array non funziona. Successivamente, però, si potrebbe modificare il suffix array in modo da rimuovere questi suffissi corti, considerando solo, data la read r, gli elementi che hanno come indice di inizio un valore compreso tra 0 e $|r| - q - 1$, anziché tra 0 e $|r| - 1$. Tale riduzione potrebbe essere applicata alle tecniche 2, 3 e 4, ma purtroppo non alla 1, la quale richiede l'intero suffix array per costruire la *BWT*. Poiché vengono ridotte le lunghezze delle stringhe analizzate, tale tecnica dovrebbe ottimizzare (a parità di numero di basi) più le stringhe corte rispetto le stringhe lunghe (ad esempio, nel caso delle stringhe corte, se queste sono lunghe ognuna 80 basi, su 80 se ne eliminano 30, se q è 30). Inoltre, a seconda delle varie tecniche, possono essere trascurate stringhe diverse: infatti, mentre nella tecnica 4 si trascurano i suffissi di lunghezza minore (o uguale, considerando il *$*) di q, nelle tecniche 2 e 3, poiché si considerano solo gli overlap di lunghezza maggiore o uguale a $q + m - 1$, i suffissi di lunghezza minore di $q + m - 1$ (o anche uguali a $q + m - 1$, sempre se consideriamo anche il simbolo *$* alla fine) possono essere trascurati. In particolare, considerando la similarità tra i risultati (precision, recall e F-measure) ottenuti dalle tecniche 3 e 4, si potrebbe pensare di impiegare di più la tecnica 3, poiché questa trascura un numero di suffissi ($q + m - 1$) maggiore di quelli trascurati dalla tecnica 4 (q). Un esempio di tale riduzione è riportato in Figura 5.1. Questa riduzione, dunque, libererebbe parte della memoria usata dalle strutture dati per la costruzione del grafo (quindi la memoria post-inizializzazione), ma non il picco raggiunto, in quanto si usa lo stesso algoritmo per costruire il suffix array e il *lcp*. Eventualmente si potrebbe provare a esplorare nuovi algoritmi per la costruzione di versioni parziali di queste strutture.

Appendice A

Ottimizzazione strutture dati

Sono ora riportate quattro delle classi impiegate nel nostro progetto: **clsSizeArray**, **clsString**, **clsSuffixArray** e **clsCountChar**. Altre due classi, **clsBWTSeq** e **clsArrayBool**, non sono riportate per la loro similarità con le classi precedenti. Il codice che invece contiene tutta la sezione del calcolo del grafo di adiacenza e che fa uso delle classi precedenti si trova nel file **BWT.cpp**, anch'esso non riportato per la sua notevole dimensione e per il fatto che il suo contenuto, e in particolare i suoi algoritmi, sono stati descritti in dettaglio nei capitoli precedenti.

A.1 Array numerici

clsSizeArray.h

```
01.    #ifndef __CLSSIZEARRAY_H__
02.    #define __CLSSIZEARRAY_H__
03.
04.    #include <math.h>
05.
06.    class clsSizeArray
07.    {
08.    public:
09.      clsSizeArray();
10.      void inizializza(unsigned long numero_elementi, unsigned long max_value);
11.      unsigned long getAt(unsigned long position);
12.      void setAt(unsigned long position, unsigned long value_string);
13.      void release();
14.    private:
15.      unsigned long numero_elementi;
16.      unsigned long* arrai;
17.      unsigned short bit_size_element;
18.    };
19.    #endif
```

clsSizeArray.cpp

```
01.    #include "clsSizeArray.h"
02.
03.    const unsigned long sm[64] = {  //start_mask
04.      0xffffffffffffffff, 0xfffffffffffffffe, 0xfffffffffffffffc, 0xfffffffffffffff8,
05.      0xfffffffffffffff0, 0xffffffffffffffe0, 0xffffffffffffffc0, 0xffffffffffffff80,
06.      0xffffffffffffff00, 0xfffffffffffffe00, 0xfffffffffffffc00, 0xfffffffffffff800,
07.      0xfffffffffffff000, 0xffffffffffffe000, 0xffffffffffffc000, 0xffffffffffff8000,
08.      0xffffffffffff0000, 0xfffffffffffe0000, 0xfffffffffffc0000, 0xfffffffffff80000,
```

```
09.      0xfffffffffff00000, 0xffffffffffe00000, 0xffffffffffc00000, 0xffffffffff800000,
10.      0xffffffffff000000, 0xfffffffffe000000, 0xfffffffffc000000, 0xfffffffff8000000,
11.      0xfffffffff0000000, 0xffffffffe0000000, 0xffffffffc0000000, 0xffffffff80000000,
12.      0xffffffff00000000, 0xfffffffe00000000, 0xfffffffc00000000, 0xfffffff800000000,
13.      0xfffffff000000000, 0xffffffe000000000, 0xffffffc000000000, 0xffffff8000000000,
14.      0xffffff0000000000, 0xfffffe0000000000, 0xfffffc0000000000, 0xfffff80000000000,
15.      0xfffff00000000000, 0xffffe00000000000, 0xffffc00000000000, 0xffff800000000000,
16.      0xffff000000000000, 0xfffe000000000000, 0xfffc000000000000, 0xfff8000000000000,
17.      0xfff0000000000000, 0xffe0000000000000, 0xffc0000000000000, 0xff80000000000000,
18.      0xff00000000000000, 0xfe00000000000000, 0xfc00000000000000, 0xf800000000000000,
19.      0xf000000000000000, 0xe000000000000000, 0xc000000000000000, 0x8000000000000000,
20.    };
21.
22.    const unsigned long em[64] = {  //end_mask
23.      0x0000000000000001, 0x0000000000000003, 0x0000000000000007, 0x000000000000000f,
24.      0x000000000000001f, 0x000000000000003f, 0x000000000000007f, 0x00000000000000ff,
25.      0x00000000000001ff, 0x00000000000003ff, 0x00000000000007ff, 0x0000000000000fff,
26.      0x0000000000001fff, 0x0000000000003fff, 0x0000000000007fff, 0x000000000000ffff,
27.      0x000000000001ffff, 0x000000000003ffff, 0x000000000007ffff, 0x00000000000fffff,
28.      0x00000000001fffff, 0x00000000003fffff, 0x00000000007fffff, 0x0000000000ffffff,
29.      0x0000000001ffffff, 0x0000000003ffffff, 0x0000000007ffffff, 0x000000000fffffff,
30.      0x000000001fffffff, 0x000000003fffffff, 0x000000007fffffff, 0x00000000ffffffff,
31.      0x00000001ffffffff, 0x00000003ffffffff, 0x00000007ffffffff, 0x0000000fffffffff,
32.      0x0000001fffffffff, 0x0000003fffffffff, 0x0000007fffffffff, 0x000000ffffffffff,
33.      0x000001ffffffffff, 0x000003ffffffffff, 0x000007ffffffffff, 0x00000fffffffffff,
34.      0x00001fffffffffff, 0x00003fffffffffff, 0x00007fffffffffff, 0x0000ffffffffffff,
35.      0x0001ffffffffffff, 0x0003ffffffffffff, 0x0007ffffffffffff, 0x000fffffffffffff,
36.      0x001fffffffffffff, 0x003fffffffffffff, 0x007fffffffffffff, 0x00ffffffffffffff,
37.      0x01ffffffffffffff, 0x03ffffffffffffff, 0x07ffffffffffffff, 0x0fffffffffffffff,
38.      0x1fffffffffffffff, 0x3fffffffffffffff, 0x7fffffffffffffff, 0xffffffffffffffff,
39.    };
40.
41.    clsSizeArray::clsSizeArray() {
42.
43.    }
44.
45.    void clsSizeArray::inizializza(unsigned long numero_elementi,
46.                                   unsigned long max_value) {
47.      this->numero_elementi = numero_elementi;
48.      this->bit_size_element = (unsigned short)(ceil(log2(max_value + 1)));
49.      unsigned long numero_bit_totali = bit_size_element*numero_elementi;
50.      unsigned long numero_long_totali = numero_bit_totali/64;
51.      if(numero_bit_totali%64!=0)
52.        numero_long_totali++;
53.
54.      arrai = new unsigned long[numero_long_totali];
55.    }
56.
57.    unsigned long clsSizeArray::getAt(unsigned long position) {
58.      unsigned long start_bit = bit_size_element*position;
59.      unsigned long end_bit = (bit_size_element*(position+1))-1;
60.      unsigned long si = start_bit/64;
61.      unsigned long so = start_bit%64;
62.      unsigned long ei = end_bit/64;
63.      unsigned long eo = end_bit%64;
64.      unsigned long return_value = 0;
65.      if(si == ei)
66.        return_value = (arrai[si] & sm[so] & em[eo]) >> so;
67.      else
```

```
68.        return_value = ((arrai[ei] & em[eo]) << (64-so)) | ((arrai[si] & sm[so]) >> so);
69.
70.      return return_value;
71.    }
72.
73.    void clsSizeArray::setAt(unsigned long position, unsigned long value) {
74.      unsigned long start_bit = bit_size_element*position;
75.      unsigned long end_bit = (bit_size_element*(position+1))-1;
76.      unsigned long si = start_bit/64;     //start_index
77.      unsigned long so = start_bit%64;     //start_offset
78.      unsigned long ei = end_bit/64;       //end_index
79.      unsigned long eo = end_bit%64;       //end_offset
80.      if(si == ei) {
81.        arrai[si] = (arrai[si] & ~(sm[so] & em[eo])) | ((value << so) & sm[so] & em[eo]);
82.      }
83.      else {
84.        arrai[si] = (arrai[si] & (~sm[so])) | ((value << so) & sm[so]);
85.        arrai[ei] = (arrai[ei] & (~em[eo])) | ((value >> (64-so)) & em[eo]);
86.      }
87.    }
88.
89.    void clsSizeArray::release() {
90.      delete[] arrai;
91.    }
```

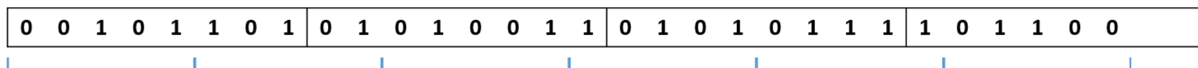


Figure A.1. *Grafico*

Come già detto, la peculiarità degli array numerici usati nel nostro progetto è che spesso se ne conoscono già i limiti inferiore e superiore dei valori contenuti. Se, per esempio, un array contiene solo i numeri da 0 a 10, allora è sufficiente usare 4 bit per rappresentare ogni elemento, mentre normalmente si userebbero un numero assai più elevato di bit usando, per esempio, il tipo standard **int**. È stata dunque scritta la classe **clsSizeArray**, il cui codice è riportato qui sopra, per ottenere una rappresentazione più efficiente degli array numerici; si assume che il valore minimo sia sempre 0 (quindi si avrà un array di valori senza segno). In input si riceve il numero di valori da salvare *numero_elementi*, e il valore massimo rappresentabile *max_value*. Per prima cosa si calcola quanti bit occuperà un elemento (riga 48 di **clsSizeArray.cpp**). Questo numero si trova mediante la formula $\lceil \log_2(max_value + 1) \rceil$. Si calcola ora il numero di bit necessari a rappresentare tutti gli elementi, moltiplicando il valore appena ottenuto per *numero_elementi*. Questi elementi sono poi salvati in un array con valori a 64 bit, dove vengono posti consecutivamente. Un elemento può quindi trovarsi o a cavallo tra due valori a 64 bit o all'interno di un solo valore a 64 bit. Nell'esempio in Figura A.1 si è usato un array di elementi da 8 bit contenente valori composti da 5 bit. Mentre il primo, il terzo e il sesto valore sono contenuti ognuno in un singolo elemento dell'array a 8 bit, il secondo, il quarto e il quinto valore sono ciascuno a cavallo tra due elementi. Alla fine potrebbero esserci dei bit inutilizzati (due in questo caso), che sono comunque di poca rilevanza se l'array è molto più lungo di un singolo elemento, come spesso avviene. Per ottenere in valori in output o per salvarli in input si sono

generate le maschere riportare alla riga 3 e 22 di **clsSizeArray.cpp**, che vengono utilizzate insieme a una serie di operazioni binarie (dalla riga 57 alla riga 87 di **clsSizeArray.cpp**). Tutte queste operazioni sono effettuate in tempo costante, e dunque non degradano in modo rilevante le performance temporali degli algoritmi che usano la classe **clsSizeArray** anziché gli array numerici con i tipi standard del C++.

A.2 Stringhe

clsString.h

```
01.    #ifndef __CLSSTRING_H__
02.    #define __CLSSTRING_H__
03.
04.    #include <iostream>
05.    #include <string>
06.    #include "clsSuffixArray.h"
07.
08.    class clsString
09.    {
10.    public:
11.      clsString();
12.      void assegna(std::string& originale);
13.      char getAt(unsigned short position);
14.      unsigned short size();
15.      std::string substr(unsigned short position);
16.      std::string toString();
17.      void release();
18.    private:
19.      unsigned short basis_number;
20.      char* sequenza;
21.    };
22.    #endif
```

clsString.cpp

```
01.    #include "clsString.h"
02.
03.    clsString::clsString() {
04.
05.    }
06.
07.    void clsString::assegna(std::string& originale)
08.    {
09.      basis_number = originale.size();
10.      unsigned short char_number = basis_number/4;
11.      if(basis_number%4!=0)
12.        char_number++;
13.      this->sequenza = new char[char_number];
14.      for(unsigned short i = 0; i < basis_number; i++) {
15.        unsigned short index = i/4;
16.        unsigned short resto = i%4;
17.        if(originale[i]=='A') {
18.          this->sequenza[index] &= ~(1UL << (0+(resto*2)));
```

```
19.         this->sequenza[index] &= ~(1UL << (1+(resto*2)));
20.       }
21.       else if(originale[i]=='C') {
22.         this->sequenza[index] &= ~(1UL << (0+(resto*2)));
23.         this->sequenza[index] |=   1UL << (1+(resto*2));
24.       }
25.       else if(originale[i]=='G') {
26.         this->sequenza[index] |=   1UL << (0+(resto*2));
27.         this->sequenza[index] &= ~(1UL << (1+(resto*2)));
28.       }
29.       else if(originale[i]=='T') {
30.         this->sequenza[index] |=   1UL << (0+(resto*2));
31.         this->sequenza[index] |=   1UL << (1+(resto*2));
32.       }
33.       else {
34.         std::cout << "Errore, carattere non è A, C, G o T" << std::endl;
35.       }
36.     }
37.   }
38.
39.   std::string clsString::toString() {
40.     std::string bwtre = "";
41.     for(unsigned short i = 0; i<basis_number; i++) {
42.       unsigned short index = i/4;
43.       unsigned short resto = i%4;
44.       int bit0 = (sequenza[index] >> (0+(resto*2))) & 1U;
45.       int bit1 = (sequenza[index] >> (1+(resto*2))) & 1U;
46.       if(bit0==0 && bit1==0)
47.         bwtre = bwtre + 'A';
48.       else if(bit0==0 && bit1==1)
49.         bwtre = bwtre + 'C';
50.       else if(bit0==1 && bit1==0)
51.         bwtre = bwtre + 'G';
52.       else if(bit0==1 && bit1==1)
53.         bwtre = bwtre + 'T';
54.     }
55.     return bwtre;
56.   }
57.
58.   char clsString::getAt(unsigned short position) {
59.     if(position==basis_number)
60.       return '$';
61.     else if (position>basis_number) {
62.       std::cout << "Errore: " << position << " > " << basis_number << std::endl;
63.       return '1';
64.     }
65.     else {
66.       unsigned short index = position/4;
67.       unsigned short resto = position%4;
68.       int bit0 = (sequenza[index] >> (0+(resto*2))) & 1U;
69.       int bit1 = (sequenza[index] >> (1+(resto*2))) & 1U;
70.       if(bit0==0 && bit1==0)
71.         return 'A';
72.       else if(bit0==0 && bit1==1)
73.         return 'C';
74.       else if(bit0==1 && bit1==0)
75.         return 'G';
76.       else if(bit0==1 && bit1==1)
77.         return 'T';
```

```
 78.      }
 79.      std::cout << "Errore sconosciuto" << std::endl;
 80.      return '1';
 81.    }
 82.
 83.    unsigned short clsString::size() {
 84.      return basis_number + 1;
 85.    }
 86.
 87.    std::string clsString::substr(unsigned short position) {
 88.      std::string substring = "";
 89.      for(unsigned short i = position; i<basis_number; i++) {
 90.        unsigned short index = i/4;
 91.        unsigned short resto = i%4;
 92.        int bit0 = (sequenza[index] >> (0+(resto*2))) & 1U;
 93.        int bit1 = (sequenza[index] >> (1+(resto*2))) & 1U;
 94.        if(bit0==0 && bit1==0)
 95.          substring = substring + 'A';
 96.        else if(bit0==0 && bit1==1)
 97.          substring = substring + 'C';
 98.        else if(bit0==1 && bit1==0)
 99.          substring = substring + 'G';
100.        else if(bit0==1 && bit1==1)
101.          substring = substring + 'T';
102.      }
103.      substring = substring + '$';
104.      return substring;
105.    }
106.
107.    void clsString::release() {
108.      delete[] sequenza;
109.    }
```

Per rappresentare le singole read si è usata la classe **clsString**, contenente un array di elementi da 16 bit, ognuno contenente 8 valori rappresentati da 2 bit (00 per *A*, 01 per *C*, 10 per *G* e 11 per *T*). Se viene richiesto l'elemento in ultima posizione è sufficiente restituire *$*. Si sono usati elementi da 16 bit anziché di 64 per evitare che rimanessero alla fine troppi bit inutilizzati.

A.3 Suffix array

clsSuffixArray.h

```
01.    #ifndef __CLSSUFFIXARRAY_H__
02.    #define __CLSSUFFIXARRAY_H__
03.
04.    #include "clsSizeArray.h"
05.
06.    class clsSuffixArray
07.    {
08.    public:
09.      clsSuffixArray(unsigned long sizetotale,
10.                     unsigned long reads_number,
11.                     unsigned long max_size_read);
```

```
12.     void setStringCharAt(unsigned long  position,
13.                          unsigned int value_string,
14.                          unsigned short value_char);
15.     unsigned long  getStringAt(unsigned long  position);
16.     unsigned short getCharAt(unsigned long  position);
17.     unsigned long size();
18.     void releaseString();
19.     void releaseChar();
20.   private:
21.     unsigned long sizetotale;
22.     clsSizeArray* pos1;
23.     clsSizeArray* pos2;
24.   };
25.   #endif
```

clsSuffixArray.cpp

```
01.   #include "clsSuffixArray.h"
02.
03.   clsSuffixArray::clsSuffixArray(unsigned long sizetotale,
04.                                  unsigned long reads_number,
05.                                  unsigned long max_size_read) {
06.     this->sizetotale = sizetotale;
07.     pos1 = new clsSizeArray();
08.     pos1->inizializza(sizetotale, reads_number-1);
09.     pos2 = new clsSizeArray();
10.     pos2->inizializza(sizetotale, max_size_read-1);
11.   }
12.
13.   void clsSuffixArray::setStringCharAt(unsigned long position,
14.                                        unsigned int value_string,
15.                                        unsigned short value_char) {
16.     pos1->setAt(position, value_string);
17.     pos2->setAt(position, value_char);
18.   }
19.
20.   unsigned long clsSuffixArray::getStringAt(unsigned long position) {
21.     return pos1->getAt(position);
22.   }
23.
24.   unsigned short clsSuffixArray::getCharAt(unsigned long position) {
25.     return pos2->getAt(position);
26.   }
27.
28.   void clsSuffixArray::releaseString() {
29.     pos1->release();
30.   }
31.
32.   void clsSuffixArray::releaseChar() {
33.     pos2->release();
34.   }
35.
36.   unsigned long clsSuffixArray::size() {
37.     return sizetotale;
38.   }
```

Il suffix array è salvato mediante la classe **clsSuffixArray**. In questa vengono usati due array numerici della classe **clsSizeArray** spiegata in precedenza, uno che ha come valore massimo la massima lunghezza delle read meno 1 (riga 10 di **clsSuffixArray.cpp**), usato per salvare gli indici di suffisso, e uno che ha come valore massimo il numero di read meno 1 (riga 8 di **clsSuffixArray.cpp**), usato per memorizzare gli indici che identificano le read.

A.4 Array di conteggio

clsCountChar.h

```
01.    #ifndef __CLSCOUNTCHAR_H__
02.    #define __CLSCOUNTCHAR_H__
03.
04.    #include "clsSizeArray.h"
05.    #include <iostream>
06.    #include "clsBWTSeq.h"
07.
08.    class clsCountChar
09.    {
10.    public:
11.      clsCountChar(unsigned long basis_number,
12.                   unsigned short spazio,
13.                   clsBWTSeq* BWT,
14.                   char char_value);
15.      void setAt(unsigned long position, unsigned long value);
16.      unsigned long getAt(unsigned long position);
17.      unsigned long size();
18.      void release();
19.    private:
20.      char char_value;
21.      clsSuffixArray* sa;
22.      clsBWTSeq* BWT;
23.      unsigned long basis_number;
24.      unsigned short spazio;
25.      clsSizeArray* countChar;
26.    };
27.    #endif
```

clsCountChar.cpp

```
01.    #include "clsCountChar.h"
02.
03.    clsCountChar::clsCountChar(unsigned long basis_number,
04.                               unsigned short spazio,
05.                               clsBWTSeq* BWT,
06.                               char char_value) {
07.      this->char_value = char_value;
08.      this->basis_number = basis_number;
09.      this->spazio = spazio;
10.      this->BWT = BWT;
11.      unsigned long size_divided = basis_number/spazio;
12.      if(basis_number%spazio!=0)
13.        size_divided++;
```

```
14.     countChar = new clsSizeArray();
15.     countChar->inizializza(size_divided, basis_number);
16.   }
17.
18.   void clsCountChar::setAt(unsigned long position, unsigned long value) {
19.     if (position>=basis_number)
20.       std::cout << "Errore: " << position << " >= " << basis_number << std::endl;
21.     else if (position%spazio!=0)
22.       std::cout << "Errore: " << position << " % " << spazio << " != 0" << std::endl;
23.     else
24.       countChar->setAt(position/spazio, value);
25.   }
26.
27.   unsigned long clsCountChar::getAt(unsigned long position) {
28.     unsigned short count_spazio_indice = 0;
29.     while(position%spazio != 0) {
30.       if(BWT->getAt(position)==char_value)
31.         count_spazio_indice++;
32.       position--;
33.     }
34.     unsigned long value = countChar->getAt(position/spazio)+count_spazio_indice;
35.     return value;
36.   }
37.
38.   void clsCountChar::release() {
39.     countChar->release();
40.   }
41.
42.   unsigned long clsCountChar::size() {
43.     return basis_number;
44.   }
```

Come riportato nel codice qui sopra, è possibile osservare il funzionamento degli array di conteggio compressi: un valore viene salvato nell'array solo se la sua posizione è divisibile per lo spazio (riga 24 di **clsCountChar.cpp**), altrimenti viene restituito un errore (riga 20 e riga 22 di **clsCountChar.cpp**). Quando viene richiesto un valore non salvato, e quindi il cui indice non sia divisibile per lo spazio impiegato, si parte più vicino all'elemento di indice minore, e da esso viene contato il numero di volte che è presente il carattere considerato (dalla riga 28 alla riga 34 di **clsCountChar.cpp**).

Bibliografia

[1] S. L. Salzberg e e. al., «GAGE: a critical evaluation of genome assemblies and assembly algorithms,» *Genome Res.,* vol. 22, n. 3, pp. 557-567, 2012.

[2] E. W. Myers, «The fragment assembly string graph,» *Bioinformatics,* vol. 21, n. s2, pp. 79-85, 2005.

[3] U. Manber e G. Myers, «Suffix arrays: a new method for on-line string searches,» *SIAM J. Comput.,* vol. 22, n. 5, pp. 935-948, 1993.

[4] P. Ferragina e G. Manzini, «Opportunistic Data Structures with Applications,» *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science,* pp. 390-398, 2000.

[5] S. Girotto, C. Pizzi e M. Comin, «MetaProb: Accurate Metagenomic Reads Binning based on Probabilistic Sequence Signatures,» *Bioinformatics,* vol. 32, n. 17, pp. i567-i575, 2016.

[6] D. H. Huson, A. F. Auch, J. Qi e S. S. C., «MEGAN analysis of metagenomic data,» *Genome Research,* vol. 17, n. 3, pp. 377-386, 2007.

[7] E. D. Wood e L. S. Salzberg, «Kraken: ultrafast metagenomic sequence classification using exact alignments,» *Genome Biology,* vol. 15, n. 3, p. R46, 2014.

[8] R. Ounit, S. Wanamaker, T. J. Close e S. Lonardi, «CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers,» *BMC Genomics,* vol. 16, n. 1, pp. 1-13, 2015.

[9] N. Segata, L. Waldron, A. Ballarini, V. Narasimhan, O. Jousson e C. Huttenhower, «Metagenomic microbial community profiling using unique clade-specific marker genes,» *Nat Methods,* vol. 9, n. 8, pp. 811-814, 2012.

[10] L. V. Vinh, T. V. Lang, L. T. Binh e T. V. Hoai, «A two-phase binning algorithm using l-mer frequency on groups of non-overlapping reads,» *Algorithms for Molecular Biology,* vol. 10, n. 1, pp. 1-12, 2015.

[11] Y. Wang, H. C. Leung, S. M. Yiu e F. Y. Chin, «MetaCluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample,» *Bioinformatics,* vol. 28, n. 18, pp. i356-i362, 2012.

[12] B. Yang, Y. Peng, H. C.-M. Leung, S.-M. Yiu, J. Qin, R. Li e F. Y. Chin, «MetaCluster: Unsupervised Binning of Environmental Genomic Fragments and Taxonomic Annotation,» *BMC Bioinformatics,* 2010.

[13] Y. W. Wu e Y. Ye, «A novel abundance-based algorithm for binning metagenomic sequences using l-tuples,» *Journal of Computational Biology,* vol. 18, n. 3, pp. 523-534, 2011.

[14] S. Chatterji, I. Yamazaki, Z. Bai e J. A. Eisen, «CompostBin: A DNA Composition-Based Algorithm for Binning Environmental Shotgun Reads,» in *RECOMB 2008: Research in Computational Molecular Biology*, Berlin, Heidelberg, Springer, 2008, p. 17–28.

[15] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali e R. Rizzi, «FSG: Fast String Graph Construction for De Novo Assembly of Reads Data,» in *12th International Symposium on Bioinformatics Research and Applications, ISBRA 2016; Minsk; Belarus; 5 June 2016 through 8 June 2016*, Belarus, Springer Verlag, 2016, pp. 27-39.

[16] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali e R. Rizzi, «FSG: Fast String Graph Construction for de Novo Assembly,» *Journal of Computational Biology,* vol. 24, n. 10, pp. 953-968, 2017.

[17] M. I. Abouelhoda, S. Kurtz e E. Ohlebusch, «Replacing suffix trees with enhanced suffix arrays,» *Journal of Discrete Algorithms,* vol. 2, n. 1, p. 53–86, 2004.

[18] M. Burrows e D. Wheeler, «A Block-sorting Lossless Data Compression Algorithm,» *Technical report, Digital Systems Research Center,* 1994.

[19] F. Shi, «Suffix arrays for multiple strings: A method for on-line multiple string searches,» in *ASIAN 1996: Concurrency and Parallelism, Programming, Networking, and Security*, pp. pp 11-22.

[20] D. C. Richter, F. Ott, A. F. Auch, R. Schmid e D. H. Huson, «MetaSim: a sequencing simulator for genomics and metagenomics,» *PloS One,* vol. 3, n. 10, p. e3373, 2008.

[21] J. T. Simpson e R. Durbin, «Efficient de novo assembly of large genomes using compressed data structures,» *Genome Research,* vol. 22, n. 3, pp. 549-556, 2012.

[22] J. T. Simpson e D. Richard, «Efficient construction of an assembly string graph using the FM-index,» *Bioinformatics,* vol. 26, n. 14, pp. i367-i373, 2010.

[23] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali e R. Rizzi, «An External-Memory Algorithm for String Graph Construction,» *Algorithmica,* vol. 78, n. 2, pp. 394-424, 2017.

[24] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali e R. Rizzi, «LSG: An External-Memory Tool to Compute String Graphs for Next-Generation Sequencing Data Assembly,» *Journal of Computational Biology,* vol. 23, n. 3, pp. 137-149, 2016.