

Università degli Studi di Padova

INGEGNERIA INFORMATICA

Tesina di ALGORITMI PER LA BIOINFORMATICA

Analisi di Tool per la ricerca di K-mer

Professore:

MATTEO COMIN

Studenti:

DAVIDE MARTINI

SIMONE NIGRO

Sommario

Una delle basi più importanti nell’analisi di sequenze genomiche è il conteggio dei k-mer (stringa di lunghezza k) contenuti nelle letture del DNA.

Data una o più sequenze, contare quante volte è presente un dato k-mer, è alla base di molti algoritmi di allineamento, ma non solo. Purtroppo la dimensione dei dati, ed i valori k di interesse tipicamente elevati ($k \geq 30$), rendono questo semplice problema computazionalmente oneroso. Sono stati sviluppati negli anni diversi tool per il conteggio di k-mer, alcuni operano in RAM (in memory) altri utilizzano anche il disco (out of memory). Lo scopo della tesina è di testare alcuni dei principali tool per il conteggio di k-mer, valutarne le performance su un laptop e confrontarle con quelle ottenute su computer molto più potenti.

I software utilizzati per lo sviluppo di questa tesina sono Visual Studio 2017, Gnuplot 5.2, Turtle 0.3, DSK 2.1.0, KMC3, Microsoft Excel 2016, Sublime Text 3, e MikTex 2.9.

Il codice di questo progetto è disponibile al seguente link:

‘<https://github.com/davidemartini/Bioinformatics-Algorithms.git>’.

Indice

Elenco delle figure	vi
Elenco delle tabelle	ix
1 Tool	1
1.1 Introduzione al problema	1
1.2 Bloom filter	2
1.3 DSK	5
1.4 Turtle	7
1.5 KMC	9
2 Lavoro Svolto	11
2.1 Descrizione preliminare	11
2.2 Raccolta ed elaborazione dei dati	12
2.3 Dati	13
3 Conclusioni	23
Appendice A Gnuplot	27
Riferimenti	31

Elenco delle figure

1.1	Comportamento di un Bloom filter	2
1.2	Inserimento di due elementi in un Bloom filter	3
1.3	Verifica della presenza di un elemento in un Bloom filter	3
1.4	Esempio di un Bloom filter	4
1.5	Algoritmo DSK	6
1.6	Algoritmo Turtle	7
1.7	Memorizzazione k-mer Turtle	8
2.1	Esecuzione temporale degli algoritmi per $k = 5$	14
2.2	Esecuzione temporale degli algoritmi per $k = 5$	14
2.3	Esecuzione temporale degli algoritmi per $k = 5$	15
2.4	Esecuzione temporale degli algoritmi per $k = 10$	15
2.5	Esecuzione temporale degli algoritmi per $k = 15$	16
2.6	Esecuzione temporale degli algoritmi per $k = 20$	16
2.7	Esecuzione temporale degli algoritmi per $k = 25$	16
2.8	Esecuzione temporale degli algoritmi per $k = 30$	17
2.9	Resident set occupato dagli algoritmi al variare di k	18
2.10	Confronto temporale degli algoritmi per $k = 50, 55, 60$	20
2.11	Resident set richiesto dagli algoritmi al variare di k	20
3.1	Confronto temporale riportato nell'articolo [12]	24
3.2	Confronto della memoria riportato nell'articolo [12]	25
A.1	Rappresentazione gnuplot del $\sin(x)$	27
A.2	Prima rappresentazione gnuplot personalizzata del $\sin(x)$	28

Elenco delle tabelle

1.1	Probabilità falso positivo in un Bloom filter	5
2.1	Tempo di esecuzione al variare di k	19

1

Tool

1.1 Introduzione al problema

Un k-mer è una sequenza genomica di lunghezza k ricorrente in una sequenza di DNA. La ricerca dei k-mer è uno degli step fondamentali per altre applicazioni nel campo della bioinformatica. Oltre ad essere un passaggio molto importante, purtroppo è anche un problema computazionalmente molto oneroso, che richiede delle risorse ed una potenza di calcolo molto elevata. L'obiettivo di questo progetto è legato alla ricerca di k-mer all'interno di sequenze di DNA. Questo lavoro è stato svolto tramite la comparazione prestazionale su laptop dei vari tool, con confronto di analisi effettuate su computer molto potenti presenti in articoli di ricerca.

I problemi principali che affliggono la ricerca dei k-mer sono il tempo impiegato per la loro ricerca e l'abbondante uso della memoria che questo compito richiede. Esistono vari tool per la ricerca e per la memorizzazione di k-mer da stringhe di DNA. Alcuni di questi hanno come obiettivo la riduzione dello spazio richiesto per la memorizzazione dei k-mer trovati, altri quello di diminuire il tempo impiegato per effettuare la scansione delle sequenze di DNA. Entrambi questi aspetti sono molto influenti quando si deve ricercare k-mer di lunghezza elevata in dataset molto vasti.

1.2 Bloom filter

Una tecnica efficiente utilizzata spesso nella ricerca dei k-mer nelle sequenze di DNA è il Bloom filter. Questa tecnica viene usata per testare l'appartenenza o meno di un elemento ad un dato insieme. Il Bloom filter viene usato per velocizzare la risposta ad una richiesta in un sistema di archiviazione basato su chiavi. Il Bloom filter essendo efficiente, riesce ad evitare degli accessi alla memoria che aumenterebbero il tempo di risposta. A volte però alcuni accessi alla memoria non possono essere evitati anche se potrebbero esserci dei falsi positivi.

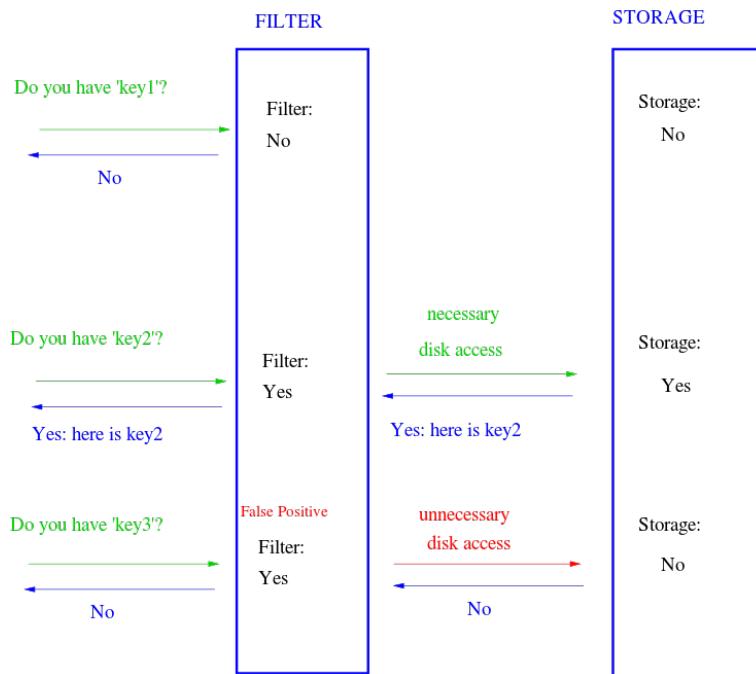


Figura 1.1: Comportamento di un Bloom filter

Un Bloom filter è un array B di m bit, i quali vengono inizialmente settati a 0. Vengono inoltre definite k funzioni hash differenti, ognuna delle quali mappa un elemento in una delle m posizioni dell'array.

Per aggiungere un elemento all'insieme, vengono calcolate tutte le funzioni hash, in modo

da ottenere k posizioni dell'array, le quali vengono tutte settate a 1. Il tempo necessario per l'aggiunta di una nuova unità è $O(k)$ ed è indipendente dagli elementi già presenti nell'insieme.

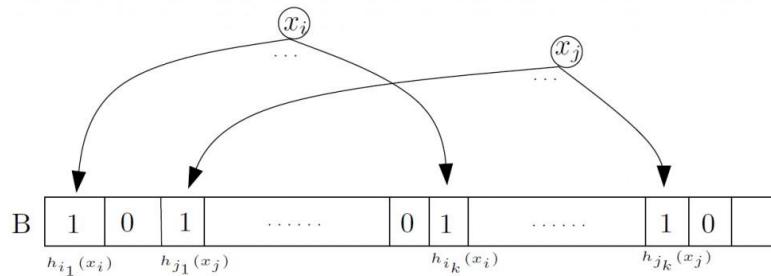


Figura 1.2: Inserimento di due elementi in un Bloom filter

Per testare invece la presenza di un elemento nell'insieme considerato, vengono calcolate le k funzioni hash e vengono controllate le rispettive k posizioni all'interno dell'array. Se anche solamente una delle posizioni contiene il valore 0, allora questo certamente non è presente nell'insieme, se invece tutte le posizioni presentano il valore 1, allora l'elemento è effettivamente contenuto nell'insieme, oppure, siamo nel caso di un falso positivo e le posizioni sono state settate a 1 non per quello specifico elemento, ma per la presenza di altri componenti nell'insieme.

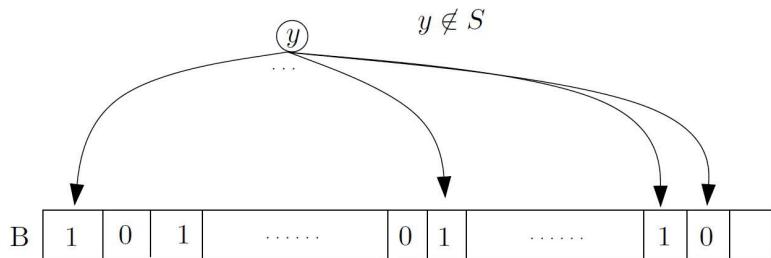


Figura 1.3: Verifica della presenza di un elemento in un Bloom filter

Uno dei problemi di questo sistema è la rimozione di un elemento dall'insieme. Si dovrebbero teoricamente settare tutte le k posizioni generate da questo elemento a 0, ma nel caso una o più di queste siano in comune con altri elementi, se venisse generata una richiesta

per uno degli elementi non eliminati questo verrebbe classificato come non appartenente all'insieme.

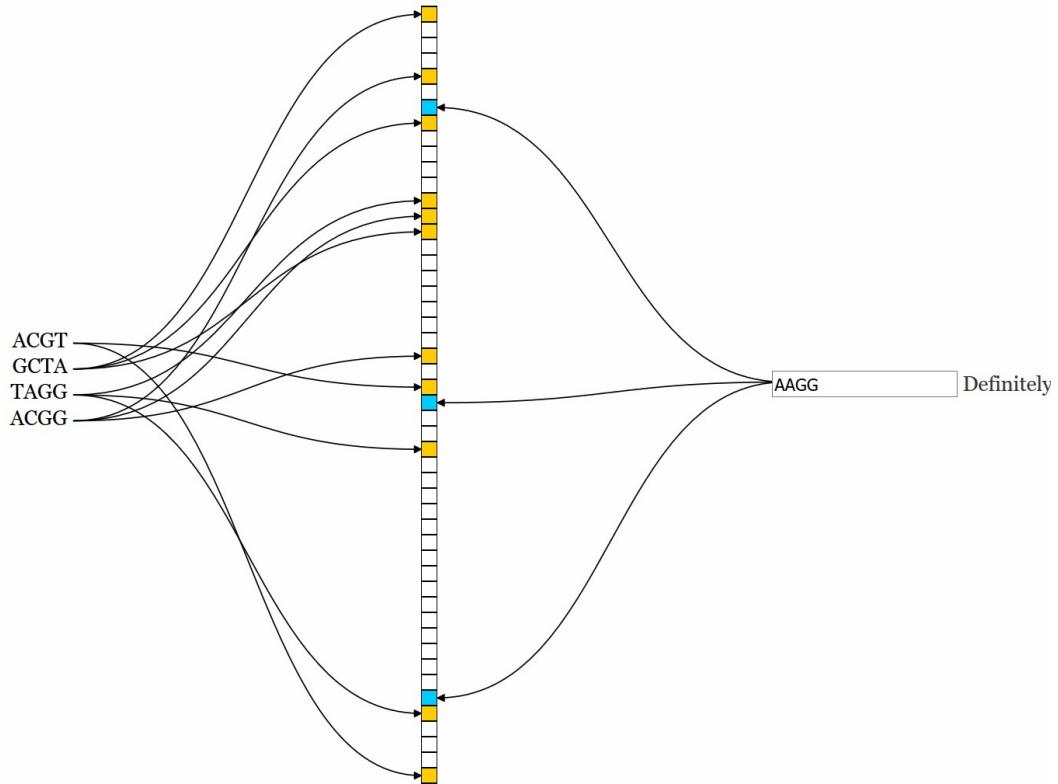


Figura 1.4: Esempio di un Bloom filter

Per ovviare invece al problema della presenza dei falsi positivi, ogni volta che il sistema verifica l'esistenza del valore 1 nelle k posizioni, viene calcolata la probabilità che un elemento sia un falso positivo o sia effettivamente un elemento dell'insieme. Assunto che le funzioni di hash distribuiscano gli elementi nelle varie posizioni in modo equiprobabile, la probabilità che una posizione assuma il valore 0 è:

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} = p$$

Quindi la probabilità di un falso positivo è:

$$\varepsilon = \left(1 - e^{-kn/m}\right)^k = (1 - p)^k$$

Questa probabilità decremente all'aumentare di m (il numero di bit dell'array B) mentre incrementa al decrescere di n (numero degli elementi inseriti nel Bloom filter). Fissati m e n , possiamo ora trovare il valore ottimale di k (il numero delle funzioni hash utilizzate dal filtro) che minimizza la probabilità di falsi positivi.

Dall'equazione precedente ricaviamo dunque:

$$k = \ln 2 \left(\frac{m}{n} \right)$$

In generale viene considerata una buona scelta prendere valori per m multipli elevati di n .

m	ε
n	0.61
$2n$	0.38
$5n$	0.09
$10n$	0.008

Tabella 1.1: Probabilità falso positivo in un Bloom filter

In un Bloom filter, che rappresenti l'intero insieme degli elementi, tutte le posizioni dell'array sono settate al valore 1, quindi, ogni elemento è contenuto nell'insieme e l'uso del Bloom filter risulta inutile non essendo più in grado di differenziare gli input.

1.3 DSK

DSK (Disk Streaming of K-mer) è un tool sviluppato da Rizk, Lavenier e Chikhi, ha come obiettivo la ricerca dei k-mer contenuti in una sequenza di DNA usando una quantità di memoria e una quantità del disco fissa, definita dall'utente prima della sua esecuzione. La determinazione della quantità di ogni singolo k-mer in un set di sequenze di DNA è concettualmente semplice ma allo stesso tempo fondamentale. Lo stato dell'arte si basa sull'uso di tabelle hash e applicazione di Bloom filter per il conteggio dei k-mer. Questi

metodi però, se applicati al conteggio di k-mer per un dataset di una persona, richiedono decine di gigabyte di memoria.

In Figura 1.5 viene riportato lo pseudocodice alla base del tool DSK. Viene usata una funzione di hash $h(\cdot)$ che mappa un k-mer in un valore numerico compreso tra $[0, H]$, dove H è generalmente 264. Sia d il numero totale di k-mer distinti trovato nell'input, allora i k-mer assumono valore di hash $x \in [0; H]$ con limite massimo $\lceil d/H \rceil$, in questo modo i k-mer vengono partizionati in modo uniforme. Ogni k-mer viene inoltre codificato tramite l'uso della rappresentazione binaria. L'algoritmo dovrebbe essere eseguito in un tempo $O(v^2b/D)$, ha una complessità lineare rispetto a v nel caso in cui $D = \Theta(v)$, ove D è la quota del disco definita dall'utente e v il numero di k-mer previsti.

```

1: Input: The set  $S$  of sequences, k-mer length  $k$ , target memory usage  $M$ 
   (bits), target disk space  $D$  (bits) and hash function  $h(\cdot)$ 
2:  $v \leftarrow \sum_{s \in S} (|s| - k + 1)$                                      (Number of k-mers)
3:  $n_{\text{iters}} \leftarrow \lceil v \cdot 2^{\lceil \log_2(2k) \rceil} / D \rceil$            (Number of iterations)
4:  $n_p \leftarrow \left\lceil \frac{v(2^{\lceil \log_2(2k) \rceil} + 32)}{0.7n_{\text{iters}}M} \right\rceil$       (Number of partitions)
5: for each iteration  $i = 0..n_{\text{iters}}$  do
6:   Initialize a set of empty lists  $\{d_0, \dots, d_{n_p}\}$  stored on disk
7:   for each sequence  $s$  in  $S$  do
8:     for each k-mer  $m$  in  $s$  do
9:       if  $(h(m) \bmod n_{\text{iters}}) = i$  then
10:         $j \leftarrow h(m)/n_{\text{iters}} \bmod n_p$ 
11:        Write  $m$  to disk in  $d_j$ 
12:   for each index  $j = 0..n_p$  do
13:     Initialize a hash table  $T$  with  $M$  bits of memory
14:     for each k-mer  $m$  in  $d_j$  do
15:        $T[m] \leftarrow \begin{cases} T[m] + 1, & \text{if } m \text{ is present in } T \\ 1, & \text{otherwise} \end{cases}$ 
16:     output  $(m, T[m])$  for each  $m$  in  $T$ 
17:     Delete  $T$ 
18:   Delete  $\{d_0, \dots, d_{n_p}\}$ 
```

Figura 1.5: Algoritmo DSK

Questo tool non permette un accesso casuale nel conteggio dei k-mer, ma permette di ottenere dei vantaggi:

- Ridotto uso della memoria: solamente un sottoinsieme ridotto di k-mer viene caricato in memoria in un istante temporale (se questo non venisse fatto, per leggere un dataset di una persona sarebbe richiesto uno spazio ≥ 1 TB per le letture usando una tabella di hash di circa 5 GB).
- I parametri vengono automaticamente dedotti: l'unico parametro richiesto è la lunghezza dei k-mer, e in maniera opzionale può essere specificato il limite del disco utilizzabile.
- Supporta valori elevati di k per la lunghezza dei k-mer.

1.4 Turtle

Un altro tool che ottimizza l'utilizzo della memoria durante la ricerca dei k-mer è Turtle, sviluppato da Roy, Bhattacharya e Schliep. Questo tool è stato sviluppato per minimizzare i cache miss per ottenere una gestione efficiente della memoria cache. Questo viene effettuato tramite l'uso di un pattern-blocked Bloom filter che rimuove i k-mer meno frequenti, assieme ad uno schema compatto ed ordinato che sostituisce le tabelle di hash. Questo approccio incrementa la complessità a livello teorico, ma evitando i cache miss viene ridotto in maniera empirica il tempo di esecuzione.

In un genoma di dimensione g , è prevista la presenza di al massimo g k-mer differenti. Questo numero può essere ridotto dalle regioni ripetute (che producono gli stessi k-mer) e da valori di k più piccoli, in quanto k-mer di dimensioni ridotte sono presenti in maggior quantità. Questo tool può ricercare i 31-mer più frequenti e quante volte questi vengono identificati (con un rapporto molto basso di falsi positivi) da un set di letture umane con 135.3 GB usando 109 GB di memoria, impiegando un tempo $< 2 h$ con 19 thread.

1. Let S be the stream of k -mers coming from the read library, BF be the Bloom filter, A be the array to store k -mers with counts and t be the threshold when we apply sorting and compaction.
2. **for all** k -mer $\in S$ **do**
3. **if** k -mer present in BF **then**
4. Add k -mer to A
5. **end if**
6. **if** $|A| \geq t$ **then**
7. Apply sorting and compaction on A
8. **end if**
9. **end for**
10. Apply sorting and compaction on A .
11. Report all k -mers in A with their counts as frequent k -mers and their counts.

Figura 1.6: Algoritmo Turtle

Questo approccio usa un Bloom filter per scremare i k-mer con una sola presenza. La numerosità dei rimanenti k-mer viene conteggiata con un nuovo algoritmo che si basa sulla compressione e l'ordinamento degli stessi. La complessità dell'ordinamento durante l'azione di compressione è $O(n \log n)$, questo passo ha un accesso sequenziale e localizzato nella memoria che aiuta ad evitare cache miss ed empiricamente viene eseguito più

velocemente di un algoritmo con complessità $O(n)$ con cache miss proporzionali alla complessità. Per dataset di dimensioni elevate (dove lo spazio non è di dimensione $O(n)$), questo metodo fallisce.

Per l'identificazione dei k-mer viene usato un Bloom filter, per contare la loro numerosità viene usato invece un array di elementi contenenti il k-mer e la sua numerosità. Quando il conteggio è stato effettuato è possibile ottenere, come output dell'algoritmo, i k-mer con una frequenza maggiore di una data soglia. Per ottenere un controllo della cache efficiente, il Bloom filter viene implementato come un pattern-blocked Bloom filter. Questo riesce a localizzare il set di bit di un elemento e i suoi blocchi di memoria consecutivi, riducendo i cache miss. Quando il numero di elementi contenuti nell'array supera una certa soglia, vengono compressi in modo che gli stessi k-mer vengano memorizzati come un unico elemento.

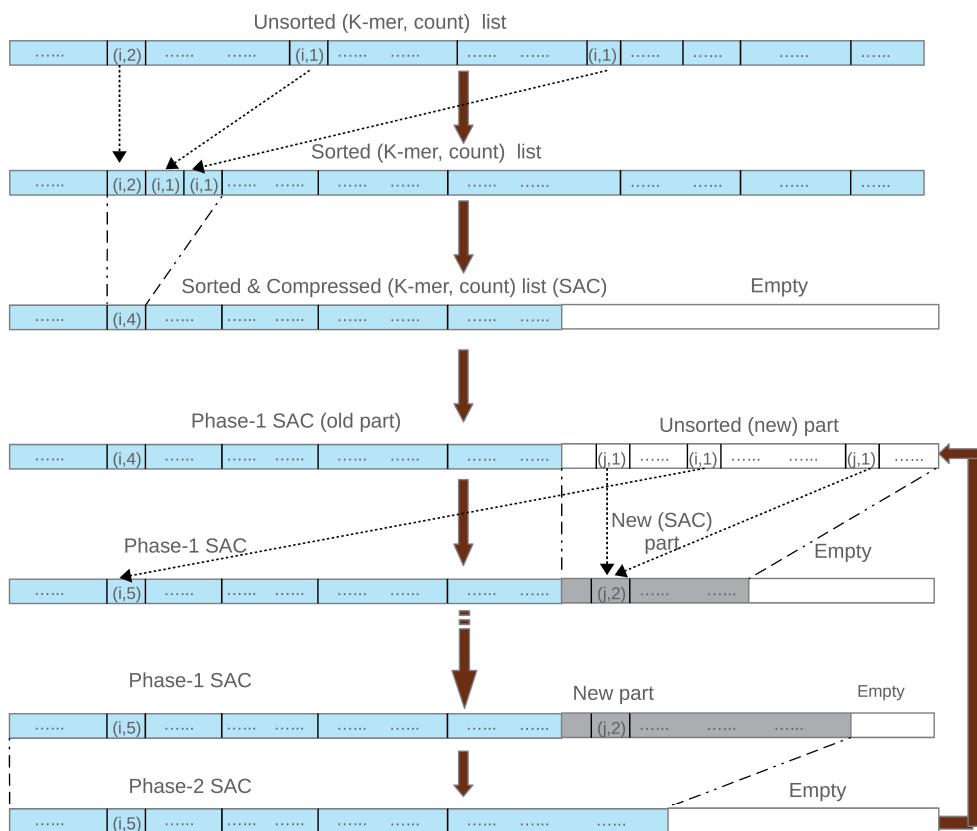


Figura 1.7: Memorizzazione k-mer Turtle

Per ottimizzare lo spazio, i k-mer vengono memorizzati tramite una codifica a bit, in cui 2 bit rappresentano un nucleotide. Questo è possibile perché i k-mer vengono estratti da dataset che contengono sequenze di DNA formate soltanto dalle lettere A , C , G e T . In Figura 1.7 è rappresentata l’idea base per la memorizzazione dei k-mer più frequenti. Si vuole quindi memorizzare i k-mer in un array A di dimensione $> n$, dove n è il numero degli elementi ripetuti. Vengono quindi memorizzati i k-mer trovati nelle prime posizioni dell’array senza tenere conto dell’ordinamento e se quel k-mer è già presente all’interno dell’array. Viene impostata una soglia t , $n < t < |A|$, e quando il numero di elementi memorizzati nell’array supera il valore t , gli elementi ripetuti nell’array vengono compattati in un solo elemento, viene aggiunta la sua numerosità e tutti gli elementi vengono ordinati. Alla fase i quindi la prima parte dell’array è ordinata e compressa. I nuovi k-mer trovati nel dataset vengono messi nella parte vuota dell’array, in coda agli altri elementi. Se il k-mer inserito è già presente nell’array si avranno elementi ripetuti, che verranno poi compressi nella fase successiva. Sia m il numero totale di elementi presenti nell’array. Quando $m > t$, vengono ordinati e compressi gli elementi inseriti nell’ultima fase. Questa procedura ha complessità $O(m)$.

1.5 KMC

KMC, K-mer Counter, è un’utility sviluppata per il conteggio dei k-mer da Kokot, M., Dugosz, M., Deorowicz, S. Questa soluzione scandisce le righe del dataset e produce una rappresentazione compatta di tutti i k-mer non unici, riportandone il numero di volte che questi compaiono. L’algoritmo implementato in KMC usa molto più spazio sul disco piuttosto che sfruttare in maniera pesante la RAM. Questa caratteristica permette a questa utility di essere eseguita anche su un laptop.

KMC è composto principalmente da due fasi. Nella prima, le lettura del dataset vengono splittate in parecchie centinaia di bin (file su disco) con una firma, contenenti i k-mer. I bin, vengono poi ordinati uno ad uno, in modo da rimuovere eventuali duplicati nella seconda fase. Nella seconda fase i vari k-mer vengono ordinati tramite un algoritmo sviluppato da Kokot, che rimpiazza l’algoritmo radix sort. Questo algoritmo si basa sul radix sort, ma usando la parallelizzazione permette di ridurre i cache miss e riesce a sfruttare al meglio la potenza di calcolo del processore. Questo algoritmo viene chiamato RADULS ed è dipendente dal tipo di architettura della macchina, infatti non ha prestazioni costanti su tutti i tipi di sistemi. Radix sort ha una complessità computazionale $O(nk)$, dove

n è la lunghezza dell'array e k è la media del numero di cifre degli n numeri, RADULS, invece, riesce ad ottenere, dove è nota l'architettura del sistema, una complessità pari a $O(n)$, ove n è il numero degli elementi dell'array.

2

Lavoro Svolto

2.1 Descrizione preliminare

Il lavoro svolto è stato complessivamente suddiviso in varie fasi. La prima fase è stata quella di ricercare dei tool per il conteggio dei k-mer in grado di essere testati su un dataset di tipo FASTA. Nella fase successiva, il lavoro è avanzato con lo studio di questi tool, rendendo i test prodotti più omogenei possibili, ottenendo un confronto equo e senza il vincolo di introdurre troppe ipotesi sui dati ottenuti.

Per avere vari dataset, il più omogenei possibili e di dimensioni variabili, è stato prodotto un programma per la generazione di dataset di tipo FASTA, in modo da ottenere dati con probabilità uniforme tra i 4 nucleotidi trattati (A, C, G, T) e con una probabilità variabile per ognuno di questi. È possibile infatti generare un dataset con probabilità discrezionale per ogni nucleotide presente. Si è in grado quindi di ottenere dataset di dimensione variabile, inserendo come parametri il numero di righe e il numero di elementi per ogni riga. Il formato FASTA consente un numero di elementi per ogni riga non superiore a 120, anche se è consigliato non superare il valore 80, sul numero di righe, invece, non si hanno restrizioni.

È stato sviluppato inoltre un algoritmo di tipo brute force, per il conteggio dei k-mer negli stessi dataset utilizzati nei tool definiti in precedenza. Questo algoritmo è stato sviluppato per evidenziare se, e nel caso fino a che dimensioni di dataset e lunghezza dei

k-mer, fosse possibile ottenere delle prestazioni confrontabili a quelle prodotte dai tool presi in considerazione.

La fase di test si è suddivisa principalmente in due fasi, la prima, nella quale il gruppo di algoritmi è stato testato su dataset di dimensioni ridotte, per scoprire se l'algoritmo brute force fosse concorrenziale fino ad una certa dimensione dell'input e per verificare se le prestazioni riportate negli articoli di ricerca, su istante di dimensioni maggiori, assumessero in proporzione lo stesso comportamento. La seconda fase invece, riguarda solamente lo studio dei tre tool su istanze di dimensione maggiore, in quanto l'algoritmo brute force non avrebbe fornito dati utili, ma avrebbe solamente richiesto un tempo di calcolo molto elevato.

Per la creazione dei dataset utilizzati nella prima fase di test è stata sviluppata una funzione in linguaggio *C* per istanze di dimensioni ridotte, invece per dataset più pesanti, è stata sviluppata una funzione con la stessa complessità computazionale ma che permettesse di creare dati con un numero di righe pari a 10^{10} , valore che supera il valore massimo permesso da una variabile di tipo *int*. Essendo però, per motivi di tempo, un'operazione molto onerosa, il numero di righe del dataset è stato limitato a 10^8 .

Per valutare le prestazioni, sia per quanto riguarda il tempo di calcolo, sia per l'utilizzo della memoria, è stato utilizzato l'eseguibile *time* presente nella cartella *usr/bin* disponibile per il sistema operativo *Ubuntu*. L'esecuzione di questo programma permette di ottenere informazioni sul tempo di calcolo richiesto per un processo (da specificare nel comando), la quantità di memoria utilizzata ed in generale l'utilizzo complessivo delle risorse di sistema per eseguire il programma specificato quando *time* viene invocato. Dei valori restituiti da *time*, sono stati analizzati:

- Il tempo percepito dall'utente per l'esecuzione del programma.
- La percentuale di CPU richiesta dal processo.
- La dimensione massima del resident set occupata dal programma in esecuzione.

2.2 Raccolta ed elaborazione dei dati

Questi dati sono stati utilizzati per la creazione di tabelle e grafici per la valutazione delle performance dei vari algoritmi presi in considerazione. I test sui vari programmi

sono stati eseguiti su un computer con un processore Intel 3.1GHz i5-2400 e 8 GB di RAM.

Per evitare di ottenere dati dipendenti dal tipo dell’istanza su cui sono stati ottenuti, sono state effettuate 10 analisi per gli stessi algoritmi. In questo modo possiamo vedere se il comportamento in generale abbia una modifica, più o meno ampia, sulle prestazioni in base all’istanza. I quattro algoritmi sono stati testati su dataset di dimensione variabile. I dataset di dimensione minore sono composti da un numero di elementi per riga che variano da 40 fino ad arrivare ad un valore massimo di 80, che quando raggiunto resta costante. I numeri di righe, invece, variano da un minimo di 40 fino ad un massimo di 200. Sia il numero di righe, che il numero di elementi per riga è stato aumentato di 5 per ogni iterazione. I dataset analizzati contenevano nucleotidi equiprobabili, ma l’analisi può essere estesa anche su dataset con probabilità differente tra i quattro componenti delle righe del dataset, anche se la composizione del dataset non dovrebbe influire sulle prestazioni di ricerca in quanto il tipo di nucleotidi presenti e la loro quantità influisce solamente nella numerosità del k-mer che formano.

Per le istanze di dimensione maggiore, il numero di elementi per ogni riga è stato mantenuto costante ad 80, mentre il numero di righe è variato da un minimo di 10^4 , ad un massimo di 10^8 , con un aumento di un fattore di moltiplicazione 10. I dati forniti in un file di testo, dal programma *time*, sono stati poi elaborati da una funzione, in modo da ottenere i dati delle analisi in un unico foglio di calcolo, per poi essere elaborato, così da ottenere grafici e altri valori utili per la valutazione degli algoritmi.

2.3 Dati

Dopo aver raccolto i dati tramite le varie esecuzioni degli algoritmi, si è provveduto ad analizzarli, in modo da mostrare le caratteristiche in maniera più immediata del loro comportamento. In primo luogo, si è cercato di analizzare il tempo di esecuzione dei vari tool, così da vedere quale potesse restituire una risposta nel minor periodo di esecuzione possibile.

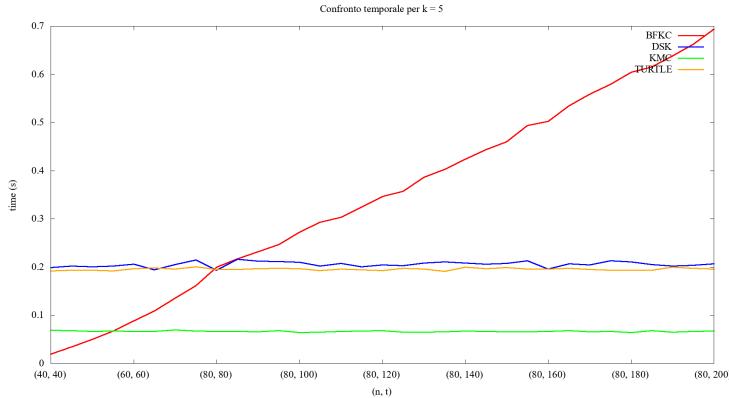


Figura 2.1: Esecuzione temporale degli algoritmi per $k = 5$

In Figura 2.1 lungo l’asse delle ascisse viene rappresentata la coppia (elementi per riga, numero di righe), che evidenzia la dimensione dell’istanza analizzata. Lungo l’asse delle ordinate, viene riportato il tempo impiegato per l’esecuzione del conteggio dei k -mer in secondi. Si può notare come sia evidente che le prestazioni dell’algoritmo *BFKC* siano molto peggiori rispetto a quelle degli altri algoritmi (in questo caso di un fattore poco più grande di 3, per $k = 30$, invece, si supera ampiamente il fattore 10). Questo risultato non sorprende in quanto l’algoritmo *BFKC* ha complessità computazionale di tipo esponenziale.

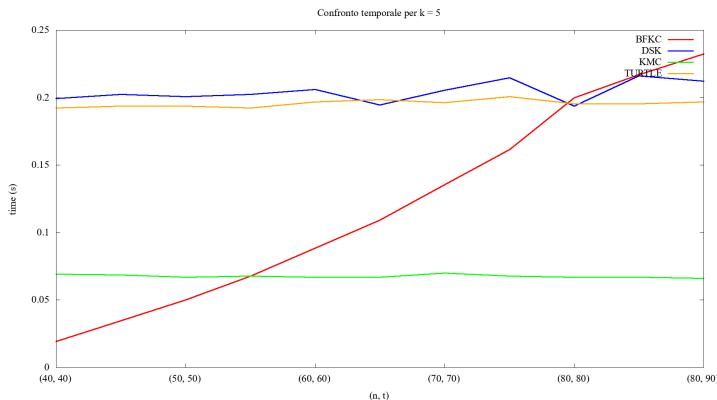


Figura 2.2: Esecuzione temporale degli algoritmi per $k = 5$

Come evidenziato in Figura 2.2, l’algoritmo brute force riesce ad essere competitivo rispetto agli altri tre, solamente per istanze con meno di 55 righe e altrettanti elementi per ognuna di queste. Una dimensione molto modesta considerato che dataset di sequenze di

DNA posso arrivare a numeri di righe di svariati ordini di grandezza maggiori. Nemmeno un uso della memoria $O(1)$ potrebbe compensare il tempo di calcolo richiesto da questo algoritmo, in quanto per istanze di dimensioni notevoli, per arrivare alla fine di un run richiederebbe giorni di calcolo a piena potenza, in quanto per dataset con 80 elementi per riga e 80 righe l'algoritmo comincia ad avere prestazioni peggiori rispetto a quelle degli altri.

Eliminando quindi il contributo temporale richiesto dall'algoritmo *BKFC*, vengono analizzate le prestazioni degli altri tre algoritmi considerati su istanze di dimensioni minori, in primo luogo, per poi passare a dataset più impegnativi. Ovviamente, questo lavoro non si è concentrato sull'analisi di una dimensione k fissata, ma questa viene fatta variare dal valore minimo 5, fino ad un massimo di 30 per i dataset ridotti, invece nel secondo caso, questo valore varia da 50 fino a 60.

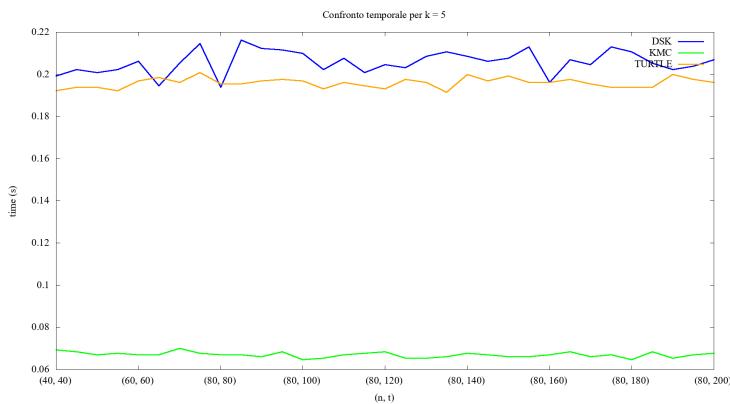


Figura 2.3: Esecuzione temporale degli algoritmi per $k = 5$

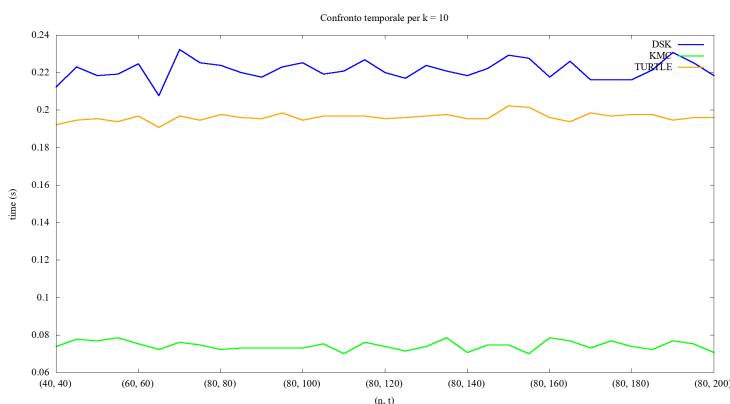


Figura 2.4: Esecuzione temporale degli algoritmi per $k = 10$

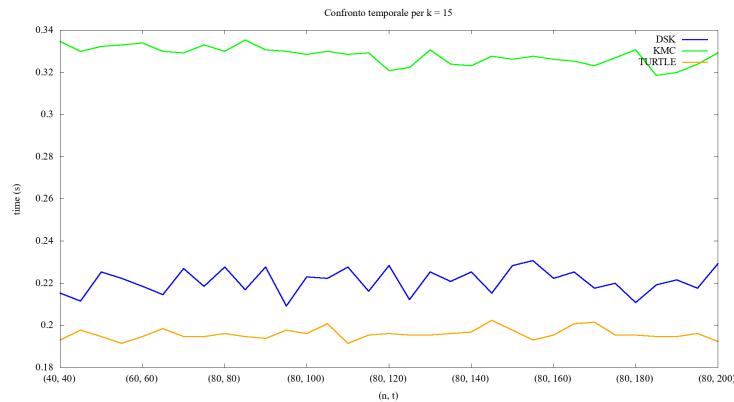


Figura 2.5: Esecuzione temporale degli algoritmi per $k = 15$

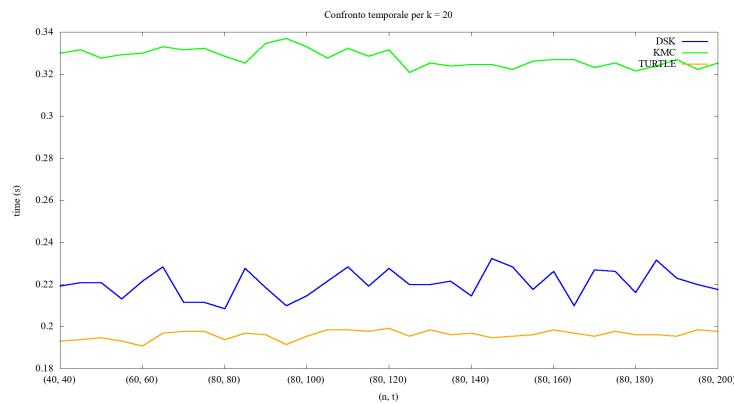


Figura 2.6: Esecuzione temporale degli algoritmi per $k = 20$

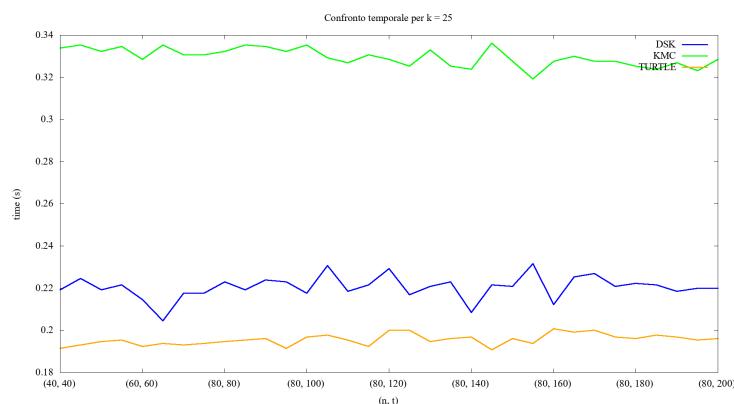


Figura 2.7: Esecuzione temporale degli algoritmi per $k = 25$

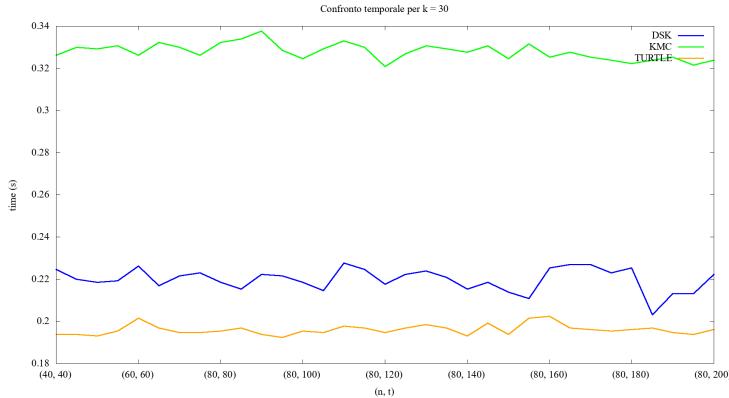


Figura 2.8: Esecuzione temporale degli algoritmi per $k = 30$

Dai grafici riportati, si possono trarre alcune informazioni sui vari tool considerati. Per comodità e chiarezza di esposizione, queste verranno riportate di seguito in un elenco.

- Per valori di k bassi, in questo caso 5 e 10, il tool *KMC* sembra essere il migliore in termini di prestazione, risultando circa 3 volte più veloce a parità di dataset. All'aumentare di k , il tempo di calcolo richiesto da *KMC* risulta essere però maggiore rispetto agli altri due tool (poco meno del doppio).
- Si può dedurre, inserendo alcuni vincoli e restrizioni, una relazione tra il tool *DSK* e *Turtle*. Per i valori di k considerati, al variare della dimensione dell'istanza del problema, il tool *DSK* impiega sempre tempo maggiore per la risoluzione del problema rispetto a *Turtle*. Da questa analisi preliminare si può definire la seguente relazione:

$$DSK = O(Turtle)$$

in termini di tempo di calcolo richiesto per dataset con un numero di righe $t < 200$, e $k < 30$.

- In questa analisi si può notare che la dimensione del dataset usato per la ricerca è quasi ininfluente, in quanto la differenza di tempo richiesta per l'analisi di un dataset di dimensioni minime e quello di dimensioni massimo considerato è quasi nulla, o molto bassa. Il tempo di calcolo richiesto aumenta, invece, con l'aumentare della lunghezza dei k-mer da ricercare all'interno delle sequenze di DNA.

Dopo aver eseguito l'analisi temporale sull'esecuzione dei vari algoritmi, si è provveduto al confronto della quantità di memoria richiesta da ogni singola esecuzione, messa in rapporto con la lunghezza dei k-mer da ricercare all'interno dei dataset di dimensione ridotta, riportata in Figura 2.9. Per comodità di rappresentazione, la quantità di memoria

è stata rappresentata in scala logaritmica, vista l'evidente disparità tra i vari algoritmi. L'algoritmo *BFKC* richiede circa 1 MB di memoria per eseguire la ricerca dei k-mer, ma anche se lo spazio occupato durante l'esecuzione è ridotto, il tempo impiegato non giustifica l'utilizzo di questo algoritmo in campo applicativo. Tra i restanti tre tool rimasti, è evidente la propensione di *KMC* al ridotto uso della memoria, di quasi un ordine di grandezza rispetto agli altri due. Vedendo che le prestazioni per dataset di dimensioni ridotte sono ottime per valori di k bassi, mentre meno performanti per valori più elevati, il consistente risparmio di memoria fornito durante l'esecuzione, potrebbe favorire, in questa prima fase delle analisi, l'utilizzo del tool *KMC* rispetto a *DSK* e *Turtle*. *KMC* arriva ad un utilizzo di circa 15 MB per la sua esecuzione, contro i 101 MB di *Turtle*, fino ad arrivare a quasi 1 GB da parte del tool *DSK*, contando che siamo ancora su istanze di dimensione modesta.

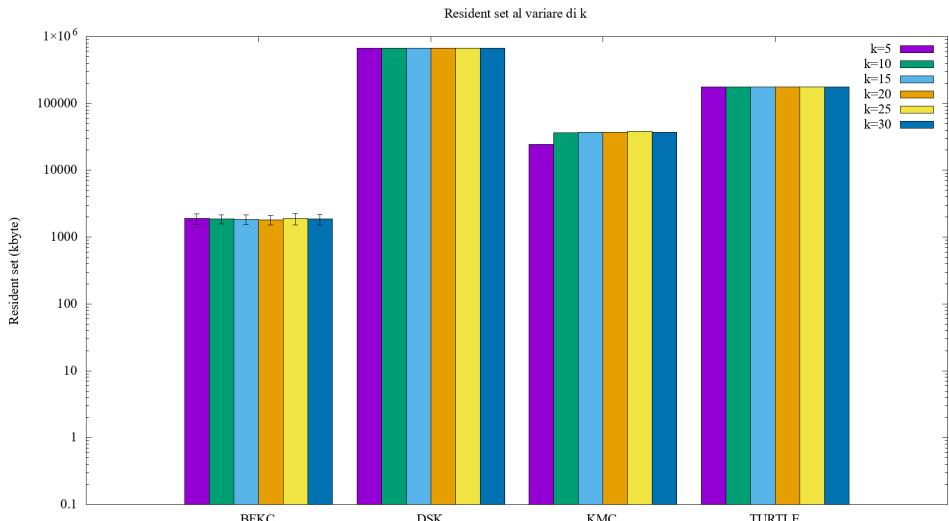


Figura 2.9: Resident set occupato dagli algoritmi al variare di k

Nella seconda fase dell'analisi delle prestazioni dei vari tool, le dimensioni dei dataset sono state aumentate. Il numero di elementi per riga è rimasto fisso ad 80, mentre il numero delle righe è variato da 10^4 a 10^8 aumentando di un fattore 10 ad ogni iterazione. Su queste dimensioni sono stati testati solo i tre tool *DSK*, *KMC*, e *Turtle*. Per il tool *Turtle* però occorre precisare che sono forniti due tipologie di programmi, una per l'analisi di k-mer di lunghezza massima 32, ed uno per poter far aumentare la lunghezza fino al valore di 64. In questa seconda fase si è utilizzato il secondo programma, in quanto il primo, vedendo la dimensione troppo elevata non procedeva all'analisi del dataset, ma

restituiva un errore sui dati di input.

Nella seconda fase di analisi, il tool *Turtle* si è rivelato inadatto, in quanto, per ogni dataset considerato non procedeva alla ricerca dei k-mer, ma restituiva un errore di core dump, per tutti i dataset di dimensioni elevate. L’analisi è quindi proseguita solamente con i due tool rimasti.

Il tool *DSK* riesce a ricercare k-mer per tutti i dataset creati, tranne per quello con numero di righe pari a 10^8 , per il quale restituisce un errore di core dump. Solamente *KMC* riesce a fornire i risultati dell’analisi per tutti i dataset considerati. Si può quindi confrontare solamente *KMC* con *DSK* per dataset con un numero di righe fino a 10^7 .

	t	<i>Turtle(s)</i>	<i>DSK(s)</i>	<i>KMC(s)</i>
$k = 50$	10^8	Core dumped	Core dumped	893.967
	10^7	Core dumped	191.181	91.981
	10^6	Core dumped	18.905	9.564
	10^5	Core dumped	2.147	1.175
	10^4	Core dumped	0.524	0.406
$k = 55$	10^8	Core dumped	Core dumped	869.395
	10^7	Core dumped	191.056	90.451
	10^6	Core dumped	18.866	9.271
	10^5	Core dumped	2.155	1.172
	10^4	Core dumped	0.516	0.414
$k = 60$	10^8	Core dumped	Core dumped	815.078
	10^7	Core dumped	189.827	88.514
	10^6	Core dumped	18.755	9.141
	10^5	Core dumped	2.156	1.166
	10^4	Core dumped	0.509	0.409

Tabella 2.1: Tempo di esecuzione al variare di k

Vengono qui riportati i grafici che rappresentano il tempo di esecuzione richiesto per i tool al variare di k .

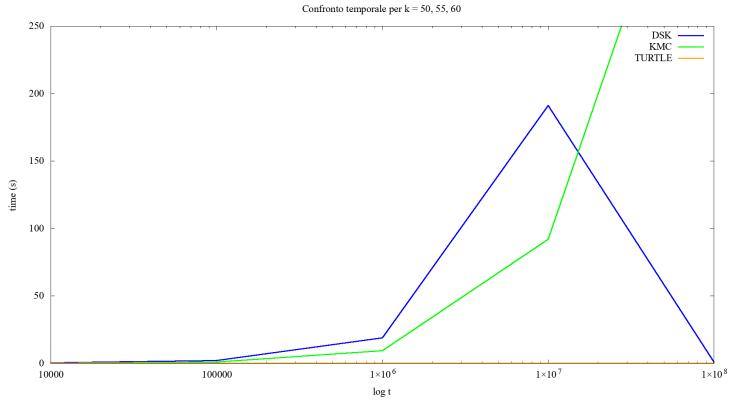


Figura 2.10: Confronto temporale degli algoritmi per $k = 50, 55, 60$

Si nota, quindi, che *KMC* riesce ad analizzare tutti i dataset presi in analisi, e per i dataset di dimensioni elevate, oltre a questo, richiede un tempo di esecuzione sempre minore a quello richiesto dal tool *DSK*, a volte anche per un fattore maggiore di 2. Inoltre, l'esecuzione di *KMC* vediamo che richiede un tempo circa 10 volte maggiore, ogni volta che il numero di righe aumenta di un fattore 10. Quindi sembra che il tempo di esecuzione, anche per valori di k differenti, sia direttamente proporzionale alla dimensione del dataset.

Per quanto riguarda la memoria richiesta dai vari tool per l'esecuzione, o il tentativo, viene rappresentato il livello di resident set richiesto tramite un istogramma, con barre di errore, in modo da vedere in modo più evidenti le variazioni di memoria richieste nelle varie esecuzioni.

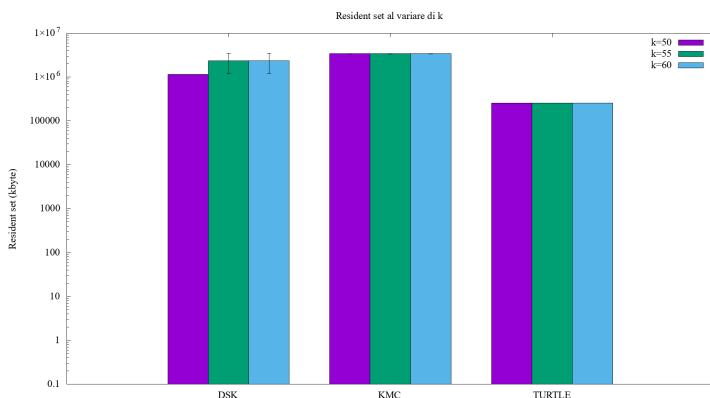


Figura 2.11: Resident set richiesto dagli algoritmi al variare di k

Come riportato in Figura 2.11, il tool *Turtle*, anche non riuscendo a compiere un’analisi completa, ma dando subito un errore di esecuzione, riesce ad occupare comunque, anche se per pochi istanti, una quantità di memoria pari a circa 130 MB, mentre *KMC* resta quasi sempre fisso ad una quota di memoria intorno ai 3 GB, mentre il tool *DSK*, varia da più di 1 GB fino a quasi 3 GB.

3

Conclusioni

Dopo aver svolto le nostre analisi, possiamo dedurre che dei tool presi in considerazione, non ne esiste uno che abbia in assoluto prestazioni migliori rispetto ai restanti. Per migliorare e trarre il massimo dai vari algoritmi presi in considerazione, possiamo pensare ad un uso di un approccio ibrido, che riesca a mettere in relazione le caratteristiche migliori di ogni tool, in modo da massimizzare le prestazioni nella ricerca dei k-mer, in relazione anche al tipo di input.

Possiamo quindi pensare, di utilizzare l'algoritmo *BFKC* per istanze molto piccole, con un numero di elementi per riga < 50 e altrettante righe, in modo da minimizzare sia l'uso della memoria richiesta per la ricerca, sia il tempo di calcolo necessario perché questa venga eseguita per qualsiasi valore di k . Se poi consideriamo valori di k inferiori a 15, possiamo considerare come scelta migliore l'utilizzo del solo tool *KMC*, in quanto richiede un minor tempo di calcolo per eseguire l'analisi del dataset. Inoltre, questo tool richiede una quota di memoria inferiore rispetto a *DSK* e *Turtle*.

Se invece consideriamo valori di $k \geq 15$, allora se si è disposti ad impiegare una quantità superiore di memoria, è possibile ottenere un tempo di calcolo minore rispetto all'utilizzo del tool *KMC*. Queste prestazioni possono essere ottenute da *Turtle*, per $k \leq 30$, in quanto oltre ad impiegare un tempo minore, rispetto sia a *KMC* che a *DSK*, richiede una quantità di memoria inferiore in confronto a quest'ultimo.

Se invece concentriamo la nostra attenzione verso dataset con dimensioni maggiori, cioè con un numero di elementi per riga pari ad 80, e un numero di righe $t \geq 10^4$, e valori di

$k \geq 50$, il tool *Turtle* risulta inadeguato, in quanto non riesce ad adempiere nemmeno ad una ricerca, ma il programma restituisce immediatamente un errore di core dump. Tra gli altri due tool rimasti, oltre al tempo di calcolo che cresce in maniera lineare rispetto al numero di righe t , notiamo che il tool *DSK* richiede un tempo di calcolo sempre maggiore rispetto a *KMC*, quindi si potrebbe tendere verso quest'ultimo. A sfavore, però, troviamo una quota di memoria maggiore richiesta da *KMC*, cosa che se si è disposti ad accettare un tempo di calcolo circa pari al doppio, *DSK* potrebbe fornire un risparmio di memoria maggiore.

Come già affermato, non esiste, tra i tool analizzati, uno preferibile confronto agli altri, ma dipende tutto dal tipo di problema che ci si trova a dover risolvere e dai vincoli, temporali o di memoria, che si devono rispettare.

Nell'articolo [12], vengono riportati i confronti effettuati su vari tool per istanze molto elevate. Qui di seguito verranno riportati due grafici inseriti nell'articolo, per confrontare gli andamenti rilevati in questa tesi, con quelli rilevati nell'articolo.

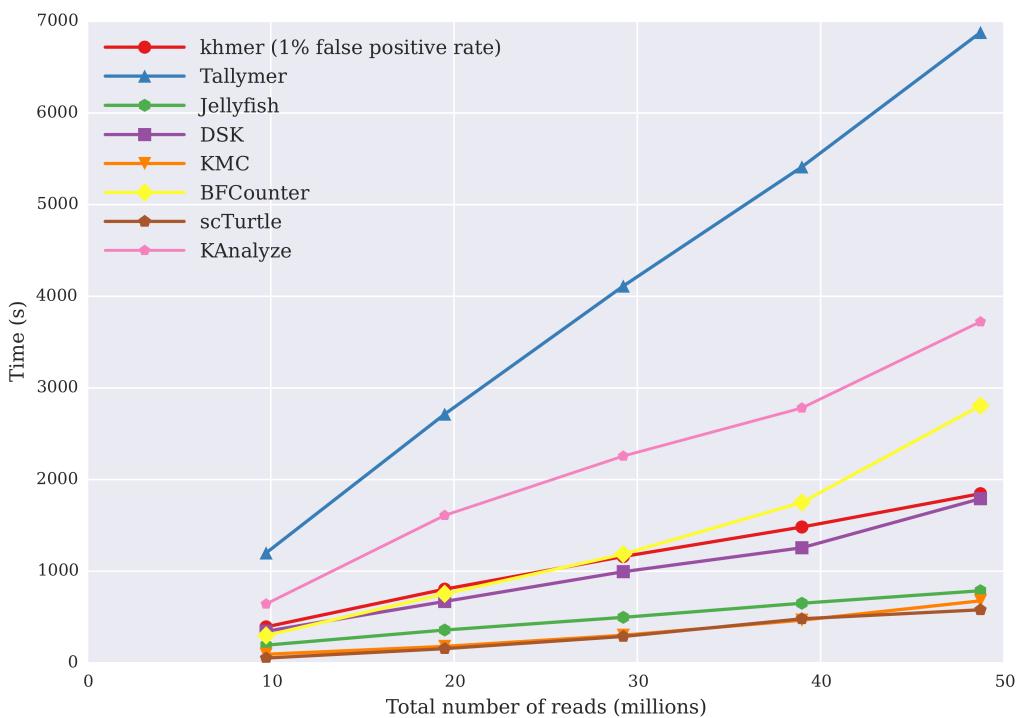


Figura 3.1: Confronto temporale riportato nell'articolo [12]

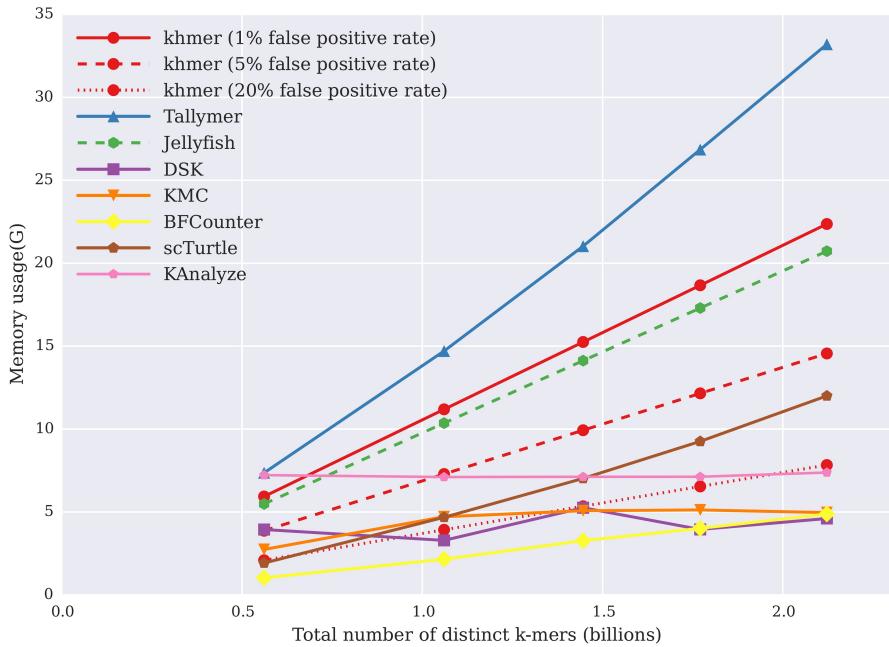


Figura 3.2: Confronto della memoria riportato nell’articolo [12]

Possiamo notare dalla Figura 3.1 che tra i tre tool, *DSK* si rivela il peggiore in termini di tempo di calcolo per istanze molto grandi, come riflesso anche nelle nostre istanze di dimensioni maggiori. *Turtle* si rivela, su macchine più potenti di un laptop, concorrenziale in termini di tempo rispetto a *KMC*, che nel nostro caso rappresentava la soluzione migliore per istanze di dimensione molto grande.

Per quanto riguarda la quantità di memoria, evidenziata in Figura 3.2, notiamo che *DSK* occupa una quantità minore, o in alcuni punti uguale, a *KMC*, elemento risultante anche dalle nostre rilevazioni. Il tool *Turtle*, che nelle istanze di dimensioni minori occupava una quantità di memoria compresa tra *DSK* e *KMC*, e per istanze elevate una quota inferiore ad entrambi (forse perché restituiva un errore di esecuzione), nel grafico occupa una quota di memoria molto maggiore rispetto agli altri due.

Tenendo conto delle considerazioni qui ottenute, se si dovesse scegliere uno tra i tre tool, *KMC* sarebbe quello da prediligere, essendo migliore o vicino ad esserlo, per istanze minori e medio-grandi. *Turtle* sarebbe consigliato da utilizzare per istanze medio-piccole, mentre per quelle elevate, escludendo *KMC*, si dovrebbe prediligere *DSK*.

A

Gnuplot

Gnuplot è un programma che permette la rappresentazione di funzioni matematiche in due o tre dimensioni. E' disponibile per diversi sistemi operativi e possiede un'interfaccia a riga di comando. Il suo utilizzo permette di esportare grafici in differenti formati, nel nostro caso utilizzeremo il PNG. Una volta scaricato e installato il programma tramite il sito <http://www.gnuplot.info> è possibile da subito utilizzare il software aprendo il terminale e digitando il comando gnuplot. Una prima rappresentazione basilare può essere ottenuta tramite il comando *plot sin(x)* ottenendo il risultato riportato in Figura A.1.

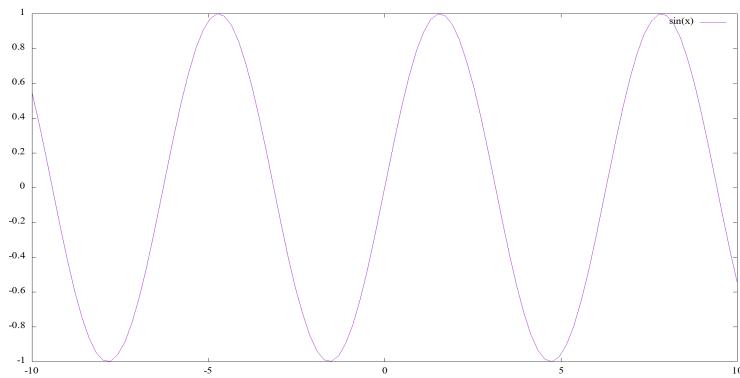


Figura A.1: Rappresentazione gnuplot del $\sin(x)$

Attraverso l'help o tramite diverse guide o documentazioni disponibili online è possibile personalizzare il grafico che verrà stampato tramite la funzione *plot*.

Per esempio, tramite delle impostazioni di gnuplot è possibile decidere se rappresentare il grafico tramite una linea di un determinato spessore, tipo e colore. Questo viene fatto usando un comando del tipo *plot sin(x) with linespoints linecolor rgb 'blue' linewidth 2 pointtype 7 pointsize 1* ottenendo il seguente risultato

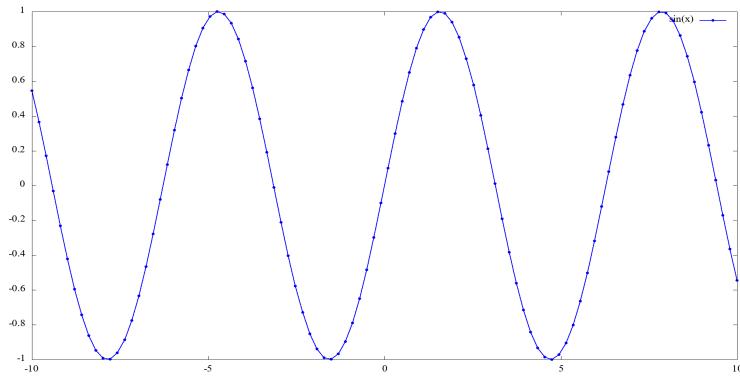


Figura A.2: Prima rappresentazione gnuplot personalizzata del $\sin(x)$

Lo stesso risultato viene raggiunto con il comando più sintetico *plot sin(x) with linespoints lc rgb 'blue' lw 2 pt 7 ps 1*. Se richiesto dall'utente, le impostazioni possono essere settate all'apertura del programma rendendole valide per tutte le rappresentazioni che verranno effettuate in quella sessione. Un'esempio di istruzione utile riguarda la scelta del terminale da voler utilizzare, impostando così il formato di output del plot.

Nel caso specifico di nostro interesse abbiamo deciso, per comodità, di creare un file in formato *.gp* che, fornito in input a gnuplot, permette al software di seguire in ordine le istruzioni scritte. Così facendo l'unica istruzione necessaria per la rappresentazione di uno o più grafici è *gnuplot nome_file.gp*. Questo metodo permette di creare veri e propri programmi script con gnuplot.

Di seguito vengono riportati i codici dei file *.gp* utilizzati per la creazione dei grafici per i dataset di dimensione ridotta.

```

set terminal png size 1920, 1080 font "Times, 20"
set output "SMALLtimek5.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
set xlabel "(n, t)"
5 set ylabel "time (s)"
set title "Confronto temporale per k = 5"
plot './BFKC/BFKCtimek5.txt' w 1 lc rgb 'red' lw 3 title "BFKC", './DSK/DSKtimek5.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCTimek5.txt' w 1 lc rgb 'green' lw 3 title "KMC", './TURTLE/TURTLEtimek5.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"
```

```

set output "SMALLtimek5.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
set xlabel "(n, t)"
set ylabel "time (s)"
set title "Confronto temporale per k = 5"
plot './DSK/DSKtimek5.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek5.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek5.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"
15
set output "SMALLtimek10.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
set xlabel "(n, t)"
set ylabel "time (s)"
20 set title "Confronto temporale per k = 10"
plot './DSK/DSKtimek10.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek10.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek10.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"

set output "SMALLtimek15.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
25 set xlabel "(n, t)"
set ylabel "time (s)"
set title "Confronto temporale per k = 15"
plot './DSK/DSKtimek15.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek15.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek15.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"

30 set output "SMALLtimek20.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
set xlabel "(n, t)"
set ylabel "time (s)"
set title "Confronto temporale per k = 20"
35 plot './DSK/DSKtimek20.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek20.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek20.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"

set output "SMALLtimek25.png"
set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
40 set xlabel "(n, t)"
set ylabel "time (s)"
set title "Confronto temporale per k = 25"
plot './DSK/DSKtimek25.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek25.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek25.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"

set output "SMALLtimek30.png"
45 set xtics ("(40, 40)" 40,"(60, 60)" 60,"(80, 80)" 80,"(80, 100)" 100,"(80, 120)" 120,"(80, 140)" 140,
"(80, 160)" 160,"(80, 180)" 180,"(80, 200)" 200)
set xlabel "(n, t)"
set ylabel "time (s)"
set title "Confronto temporale per k = 30"
plot './DSK/DSKtimek30.txt' w 1 lc rgb 'blue' lw 3 title "DSK", './KMC/KMCtimek30.txt' w 1 lc rgb 'green'
' lw 3 title "KMC", './TURTLE/TURTLEtimek30.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"
50
set output "Bestmiddlek5.png"
set xtics ("(40, 40)" 40,"(50, 50)" 50,"(60, 60)" 60,"(70, 70)" 70,"(80, 80)" 80,"(80, 90)" 90)
set xlabel "(n, t)"
set ylabel "time (s)"
55 set xrange[40:90]
set title "Confronto temporale per k = 5"
plot './BFKC/BFKCtimek5.txt' w 1 lc rgb 'red' lw 3 title "BFKC", './DSK/DSKtimek5.txt' w 1 lc rgb 'blue'
' lw 3 title "DSK", './KMC/KMCtimek5.txt' w 1 lc rgb 'green' lw 3 title "KMC", './TURTLE/
TURTLEtimek5.txt' w 1 lc rgb 'orange' lw 3 title "TURTLE"

```

Qui, invece, viene allegato il codice per la creazione degli histogrammi con le barre di errore, per i dataset di dimensione ridotta, estendibili per quelli di dimensione più elevata.

```
set terminal png size 1920, 1080 font "Times, 20"
set output "SMALLresidentsetall.png"
set style data histogram
set style histogram cluster gap 1 errorbars
5 set style fill solid border rgb "black"
set auto x
set yrange [0:*]
set datafile separator ","
set ylabel "Resident set (kbyte)"
10 set title "Resident set al variare di k"
set logscale y 10
plot for [i=2:17:3] 'SMALLresidentsetall.dat' using i:i+1:xtic(1) title col(i)
```

Riferimenti

- [1] KOKOT, M., DUGOSZ, M., DEOROWICZ, S. “*KMC 3: counting and manipulating k-mer statistics*”, 27 Jan 2017.
- [2] KMC3, <https://github.com/refresh-bio/KMC>.
- [3] KMC3, <http://sun.aei.polsl.pl/REFRESH/index.php>.
- [4] KOKOT, M., DEOROWICZ, S., DEBUDAJ-GRABYSZ, A., “*Sorting Data on Ultra-Large Scale with RADULS New Incarnation of Radix Sort*”, 8 Dec 2016.
- [5] RADULS, <https://arxiv.org/abs/1612.02557>.
- [6] RIZK, G., LAVENIER, D., CHIKHI, R., “*DSK: k-mer counting with very low memory usage*”, 16 Jan 2013.
- [7] DSK, <http://minia.genouest.org/dsk/>.
- [8] ROY, R. S., BHATTACHARYA, D., SCHLIEP, A., “*Turtle: Identifying frequent k-mers with cache-efficient algorithms*”, 10 Mar 2014.
- [9] TURTLE, <http://bioinformatics.rutgers.edu/Software/Turtle/>.
- [10] BLOOM FILTER, <https://www.jasondavies.com/bloomfilter/>.
- [11] BLOOM FILTER, https://en.wikipedia.org/wikni/Bloom_filter.
- [12] ZHANG, Q., PELL, J., CANINO-KONING, R., HOWE, A. C., BROWN C. T., “*These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure*”, 25 July 2014.
- [13] GNUPLOT, http://www.gnuplot.info/docs_5.2/Gnuplot_5.2.pdf.