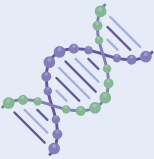


Analisi di BFCounter per il conteggio efficiente dei k-mer nel DNA tramite Bloom Filter



Rosalia Fortino
Matricola 0522502023



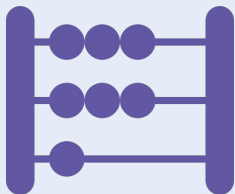
Contesto



**Evoluzione delle tecnologie
di sequenziamento**



Sfide nella gestione dei dati



Conteggio dei k-mer



Obiettivi



**Principi e limiti dei
Bloom Filter**



**Confronto tra
BFCounter e Jellyfish**





01

Background



Definizione di k-mer

Tutte le possibili **sottosequenze di lunghezza k** che possono essere estratte da una sequenza genomica più lunga

Alfabeto $\Sigma = \{A, C, G, T\}$

Sequenza S di lunghezza n

Sottostringa contigua di S di lunghezza k , dove $k \leq n$





Proprietà

Numero di k-mer sovrapposti che possono essere estratti da una sequenza di lunghezza n

$$N_{k\text{-mer}} = n - k + 1$$

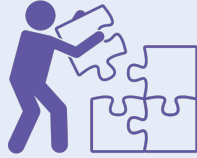
Numero totale di possibili k-mer distinti per un dato valore di k

Crescita
esponenziale

$$N_{possibili} = |\Sigma|^k$$

k	k-mer
1	G, T, A, C
2	GT, TA, AG, GA, AG, GC, CT, TG
3	GTA, TAG, AGA, GAG, AGC, GCT, CTG, TGT
4	GTAG, TAGA, AGAG, GAGC, AGCT, GCTG, CTGT
5	GTAGA, TAGAG, AGAGG, GAGCT, AGCTG, GCTGT
6	GTAGAG, TAGAGC, AGAGCT, GAGCTG, AGCTGT
7	GTAGAGC, TAGAGCT, AGAGCTG, GAGCTGT
8	GTAGAGCT, TAGAGCTG, AGAGCTGT
9	GTAGAGCTG, TAGAGCTGT
10	GTAGAGCTGT

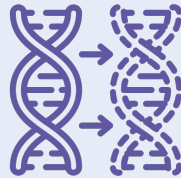
Ruolo nell'analisi genomica



**Assemblaggio
genomico**



Metagenomica



**Identificazione di
varianti genomiche**



**Studi evolutivi
e filogenetici**



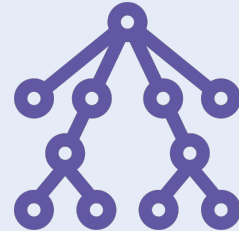
Tecniche di conteggio

Ordinamento

Tabelle hash



**Strutture dati
avanzate**





Bloom Filter

Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM

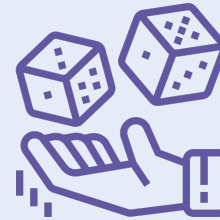
Computer Usage Company, Newton Upper Falls, Mass.

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications, in particular, applications in which a large amount of data is involved and a core resident hash area is consequently not feasible using conventional methods.

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

Analysis of the paradigm problem demonstrates that allowing a small number of test messages to be falsely identified as members of the given set will permit a much smaller hash area to be used without increasing reject time.



**Struttura dati
probabilistica**





Definizione di Bloom Filter

Dato un insieme $S = \{x_1, x_2, \dots, x_n\}$ con $|S| = n$, la struttura di base del Bloom Filter può essere descritta come:

- Un **array** B di dimensione m , con $B[i] \in \{0, 1\}$ per $i \in [0, m - 1]$
- Un insieme di k **funzioni hash** indipendenti $\{h_1, h_2, \dots, h_k\}$
dove ogni $h_i : U \rightarrow [0, m - 1]$, con U universo degli elementi




Riduzione della memoria

Progettati per rappresentare insiemi di grandi dimensioni utilizzando una **quantità ridotta di memoria**

Non è necessario memorizzare gli elementi stessi, ma solo i bit calcolati dalle funzioni hash

La memoria è determinata principalmente dalla dimensione m dell'array di bit


$$m \approx - \frac{n \ln(P(fp))}{(\ln 2)^2}$$



Falsi positivi

I **falsi positivi** si verificano quando il Bloom Filter indica erroneamente che un elemento è presente, anche se in realtà non lo è

Causati dalle collisioni tra gli hash di diversi elementi, che possono impostare gli stessi bit a 1

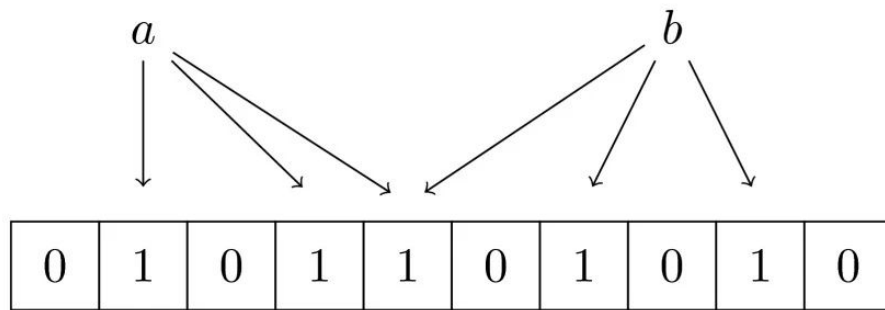
$$P(fp) = \left(1 - \left(1 - \frac{1}{m} \right)^{kn} \right)^k \quad k_{opt} = \frac{m}{n} \ln(2)$$



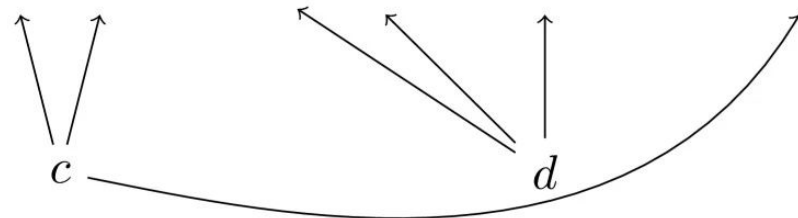


Esempio

inserted



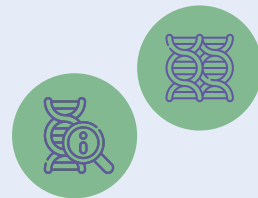
not inserted





02

Analisi dei tool



BFCOUNTER



Efficient counting of k -mers in DNA sequences using a bloom filter

Páll Melsted^{1*} and Jonathan K Pritchard^{1,2*}

Abstract

Background: Counting k -mers (substrings of length k in DNA sequence data) is an essential component of many methods in bioinformatics, including for genome and transcriptome assembly, for metagenomic sequencing, and for error correction of sequence reads. Although simple in principle, counting k -mers in large modern sequence data sets can easily overwhelm the memory capacity of standard computers. In current data sets, a large fraction—often more than 50%—of the storage capacity may be spent on storing k -mers that contain sequencing errors and which are typically observed only a single time in the data. These singleton k -mers are uninformative for many algorithms without some kind of error correction.

Results: We present a new method that identifies all the k -mers that occur more than once in a DNA sequence data set. Our method does this using a Bloom filter, a probabilistic data structure that stores all the observed k -mers implicitly in memory with greatly reduced memory requirements. We then make a second sweep through the data to provide exact counts of all nonunique k -mers. For example data sets, we report up to 50% savings in memory usage compared to current software, with modest costs in computational speed. This approach may reduce memory requirements for any algorithm that starts by counting k -mers in sequence data with errors.

Conclusions: A reference implementation for this methodology, BFCOUNTER, is written in C++ and is GPL licensed. It is available for free download at <http://pritch.bsd.uchicago.edu/bfcounter.html>



Pseudocode



Algorithm 1 Bloom filter k -mer counting algorithm

```
1:  $B \leftarrow$  empty Bloom filter of size  $m$ 
2:  $T \leftarrow$  hash table
3: for all reads  $s$  do
4:   for all  $k$ -mers  $x$  in  $s$  do
5:      $x_{rep} \leftarrow \min(x, \text{revcomp}(x))$  //  $x_{rep}$  is the canonical  $k$ -mer for  $x$ 
6:     if  $x_{rep} \in B$  then
7:       if  $x_{rep} \notin T$  then
8:          $T[x_{rep}] \leftarrow 0$ 
9:       else
10:        add  $x_{rep}$  to  $B$ 
11: for all reads  $s$  do
12:   for all  $k$ -mers  $x$  in  $s$  do
13:      $x_{rep} \leftarrow \min(x, \text{revcomp}(x))$ 
14:     if  $x_{rep} \in T$  then
15:        $T[x_{rep}] \leftarrow T[x_{rep}] + 1$ 
16: for all  $x \in T$  do
17:   if  $T[x] = 1$  then
18:     remove  $x$  from  $T$ 
```





Gestione dei falsi positivi in BFCounter

Fase 1

I k-mer rilevati come già osservati vengono inseriti in una tabella hash per ulteriori verifiche

Fase 2

La tabella hash viene utilizzata per calcolare con precisione il conteggio effettivo dei k-mer

Questo permette di correggere gli errori dovuti ai falsi positivi generati dal Bloom Filter

< 2%



Implementazione



Libreria Bloom Filter

Libreria SparseHash



- Inserimento e verifica rapidi dei k-mer
- Allocazione dinamica della memoria
- Personalizzazione dei parametri di hash
- Riduzione dell'overhead computazionale





Hashing

```
bloom_type hash_ap(const unsigned char* begin, std::size_t remaining_length, bloom_type hash) const
{
    const unsigned char* it = begin;
    while(remaining_length >= 2)
    {
        hash ^= (hash << 7) ^ (*it++) * (hash >> 3);
        hash ^= (~((hash << 11) + ((*it++) ^ (hash >> 5))));
        remaining_length -= 2;
    }
    if (remaining_length)
    {
        hash ^= (hash << 7) ^ (*it) * (hash >> 3);
    }
    return hash;
}
```



Inserimento

```
inline void insert(const unsigned char* key_begin, const std::size_t length)
{
    std::size_t bit_index = 0;
    std::size_t bit = 0;
    for(std::vector<bloom_type>::iterator it = salt_.begin(); it != salt_.end(); ++it)
    {
        compute_indices(hash_ap(key_begin,length,(*it)),bit_index,bit);
        bit_table_[bit_index / bits_per_char] |= bit_mask[bit];
    }
    ++inserted_element_count_;
}
```



Jellyfish



A fast, lock-free approach for efficient parallel counting of occurrences of k -mers

Guillaume Marçais^{1,*} and Carl Kingsford²

¹Program in Applied Mathematics, Statistics and Scientific Computation and ²Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

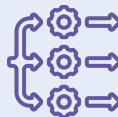
Associate Editor: Alex Bateman



Gestione dinamica



Filtraggio dei dati



Parallelismo e scalabilità





03

Replica dei test



Metodologia



SEQUENCE READ ARCHIVE

Genome sequencing of SARS-CoV-2 isolates after 1 passage in Vero E6 cells (SB2)

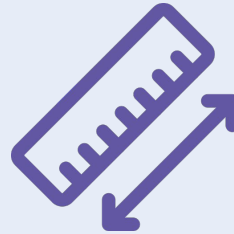
Severe acute respiratory syndrome coronavirus 2

Genome sequencing of two clinical SARS-CoV-2 isolates after one passage in Vero E6 cells

Illumina MiniSeq

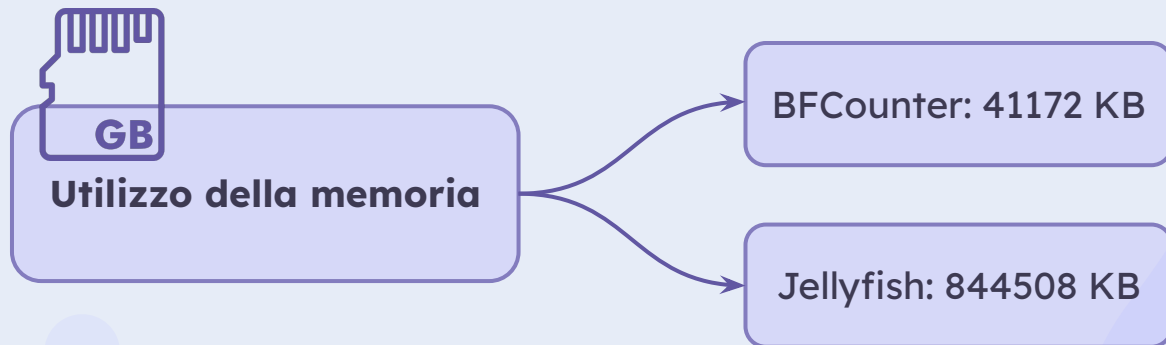
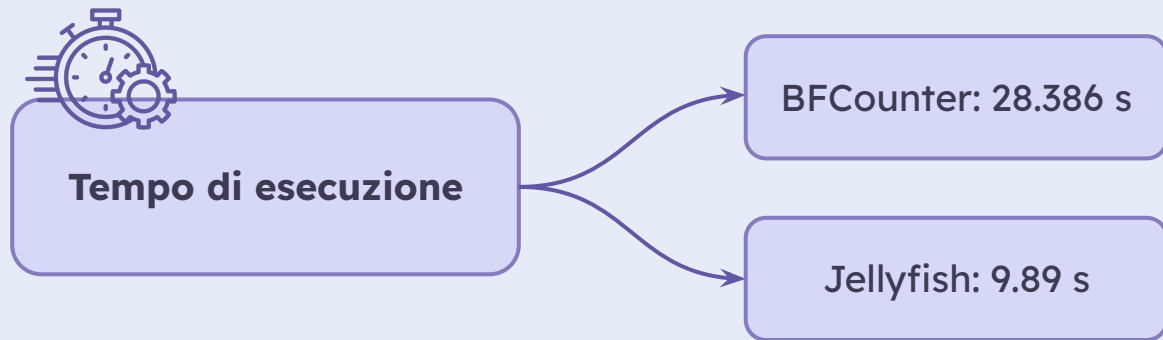
[SRR11528307](#)

k = 31





Risultati



04

Conclusione





Analisi comparativa

BFCOUNTER

- Minor consumo di memoria
- Buone prestazioni in contesti con risorse limitate
- Probabilità di falso positivo inferiore al 2%



Jellyfish

- Risultati esatti
- Prestazioni ottimali su hardware potenti
- Maggior consumo di memoria





**Grazie per
l'attenzione!**

