Bloom Filters

Filippo Todeschini

April 22, 2016

Materia: Laboratory on Algorithms for Big Data
Professore: Rossano Venturini

Contents

1 Introduzione

Questo progetto ha l'obiettivo di analizzare il comportamento di una particolare struttura dati e di alcune sue varianti. la struttura dai in questione è il Bloom Filter e le versioni implementate sono: Standard Bloom Filter(SBF), Counting Bloom Filter(CBF), Compressed Standard Bloom Filter(SBF_CM), Compressed Counting Bloom Filter(CBF_CM), Dynamic Standard Bloom Filter(SBF_DY), Dynamic Counting Bloom Filter(CBF_DY) e Spectral Bloom Filter(CBF_SP).

In questa relazione verrá prima di tutto descritta in termini teorici, nella sezione 2, la struttura dati in questione e poi verranno presentate nel dettaglio tutte le versioni implementate per questo progetto. Nella sezione 3 queste verranno confrontate, in particolare verr analizzata la quantitá di memoria occupata e il tempo necessario per le operazioni di add(), lookup(), frequency() e delete() al variare del numero di elementi rappresentati.

Infine, nella sezione 4 verrá presentata una struttura dati alternativa al Bloom Filter che permette sempre di rappresentare un insieme in maniera succinta. Questa verrá analizzata, sempre in termini di quantitá di memoria occupata e di tempo necessario per l'operazione di lookup(), confrontandola con i risultati ottenuti con la versione base dei Bloom Filter.

2 Bloom Filter

The Bloom Filter principle

Wherever a list or set is used, and space is at a premium, consider using a Bloom Filter if the effect of false positive can be mitigated.[1]

2.1 Definizione

Il Bloom Filter è una particolare struttura dati probabilistica che permette di rappresentare un insieme di dati. A differenza di strutture dati quali le liste, questa permette ridurre lo spazio di memoria occupato dallinsieme al costo della presenza di alcuni Falsi Positivi. Questa struttura dati viene utilizzata sopratutto per effettuare delle query di appartenenza di un elemento in un insieme. Tuttavia, essendo una struttura dati probabilistica, c'è una certa probabilitá di ottenere un falso positivo (FP), ovvero che loperazione effettuata ha portato un risultato positivo, quando in realtá sarebbe negativo. Se gli FP non generano significanti problemi nellapplicazione in cui viene utilizzata questa struttura dati, allora lefficienza che si ottiene in termini di spazio tale da preferire questa struttura dati rispetto alle altre.

Questa struttura dati stata teorizzata ed introdotta negli anni '70 da Burton Bloom ed inizialmente è stata utilizzata prevalentemente in applicazioni per la

gestione ed interrogazione di database. Recentemente, è stata utilizzata in altre tipologie di applicazioni, problemi spesso implementandone nuove varianti per superarne i limiti iniziali come ad esempio laggiunta di nuove operazioni. Alcuni esempi di problematiche in cui sono stati applicati i bloom filter verranno analizzati nella sezione relativa alle applicazioni. Ció che unisce gli utilizzi in queste diverse applicazioni che i Bloom FIlter permettono di ottenere una rappresentazione succinta di una lista di elementi; ad esempio nei casi in cui le liste sono molto grandi e debbano essere inviate, condivise allinterno di una rete, se fossero rappresentate attraverso strutture dati come le liste questo comporterebbe una grossa inefficienza in termini di spazio. Quindi, il Bloom filter una struttura dati che riduce significativamente lo spazio occupato da una lista, al costo di ottenere con una certa probabilitá dei Falsi Positivi.

2.2 Problema

Dato un insieme S=$\{x_1, x_2, x_3, x_4, ..., x_n\}$ di dimensione n, l'obiettivo dei Bloom Filter è quello di rappresentare questo insieme S in un array di m bits utilizzando k hash function h() indipendenti tra loro, ognuna delle quali mappa ogni elemento di S in un range {1, ..., m}. Per rappresentare S nel Bloom Filter, l'idea è che $\forall$ x $\in$ S si vadano a calcolare i k valori delle funzioni hash $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ e si vanno a modificare opportunamente tutti i bit di m con indice $h_k(x)$, $\forall$ k=1, ..., k.
Questo permette di eseguire, per quasi tutte le varianti, operazioni di add() e lookup() in tempo costante O(k); tuttavia, a seconda della variante che si utilizza, puó variare lo spazio di memoria occupato e a volte si possono implementare operazioni in altre versioni non sono possibili, come ad esempio quella di delete().

In generale, i valori ottimali di m e k dipendono dalla probabilitá di ottnere un falso positivo(FP) e dal numero di elementi che ci si aspetta vengano inseriti nel Bloom Filter. Assumendo n, la dimensione dell'insime da rappresentare S, come il numero atteso di elementi da inserire nel Bloom Filter e p la probabilitá di ottenere un falso positivo desiderata si ha che:

$$m = \frac{-\ln p}{\ln^2 2} n \qquad\qquad k = \frac{m}{n} \ln 2$$

Questi valori ottimali di m e h in funzione di p e n sono stati ricavati dalla formula relativa alla probabilitá di ottenere un falso positivo. Sappiamo che la probabilitá che un bit sia uguale a 1 per una certa funzione hash è $\frac{1}{m}$, quindi probabilitá che un bit sia uguale a 0 per una certa funzione hash è $(1-\frac{1}{m})$, invece considerando le k funzioni hash è $(1-\frac{1}{m})^k$. Dopo aver effettuato n inserimenti, n operazioni add(), la probabilitá che un certo bit sia settato ancora a 0 è $(1-\frac{1}{m})^{kn}$, mentre per quelli settati ad 1 è $1-(1-\frac{1}{m})^{kn}$. Quindi la probabilitá di ottenere un falso positivo è

$$fp = (1-(1-\tfrac{1}{m})^{kn})^k = (1-e^{-k\frac{n}{m}})^k$$

Nelle sezioni successive verranno descritte nel dettaglio tutte la varie versioni implementate, concentrandosi sopratutto sulle particolaritá di ognuna di esse. Alle descrizioni verranno affiancati alcune porzioni di codice per mostrare pi nel dettaglio l'implementazione effettuata.

2.3 Funzioni Hash

La scelta di quale funzione hash utilizzare è stata fatta seguendo alcune indicazioni trovate online e su articoli riguardanti i Bloom Filter in generale oppure il problema di come definire un insieme di funzioni hash indipendenti tra loro, ma tutte con lo stesso range di valori.

Il primo step è stato quello di scegliere quale famiglia di funzioni hash utilizzare. Per quanto riguarda i Bloom Filter, alcune librerie utilizzano funzioni quali MD5 oppure SHA, tuttavia sono stati trovati alucuni articoli e blog che consigliano l'utilizzo delle funzioni MurmurHash; queste risultano essere piú performanti e piú adatte a questa struttura dati. Per ció sono state inserite nel progetto delle classi, trovate nelle librerie online, nelle quali sono implementate tali funzioni.

Il secondo step è quello di combinare le k funzioni hash, in maniera da mantenerle indipendenti; per fare ció si è utilizzata la tecnica del Double Hashing

$$h(i,k) = (h_1(k) + i * h_2(k)) mod |T|$$

Questo è stato implementato nella classe BFHashing in una serie di metodi statici; i principali sono quelli indicati qua sotto, dove a partire da un elemento, il range m e il tipo di funzione hash utilizzare restituisce il valore ottenuto dalla funzione hash; i valori di seedHash1 e seedHash2 sono due valori consecutivi in modo da rendere indipendenti tra loro le due funzioni hash.

```
1  public static int getHashing(byte[] element, int i, int m, String
       type){
2    int hash=(hash(element,type,seedhash1)+ hash(element,type,
         seedhash2) * i) % m;
3    if( hash < 0 ) hash += m;
4    return hash;
5  }
6  private static int hash( byte[] key, String type, int seed){
7    switch(type){
8      case "Murmur3":return MurmurHash3.murmurhash3_x86_32(key, 0,
           key.length, seed);
9      case "Murmur2":return MurmurHash2.hash32(key, key.length, seed)
           ;
10     case "Murmur":return MurmurHash.hash(key, seed);
11   }
12 }
```

2.4 Standard Bloom Filter

Lo Standard Bloom Filter(SBF)[1] è la versione base di questa famiglia di strutture dati; questo è stato implementato nella classe chiamata StandardBloomFilter. Le istanze di questa classe vengono costruite a partire da una data probabilitá di ottenere FP (fp_obj), dal numero di elementi che devono essere rappresentati(n) e dalla tipologia di funzione hash da utilizzare.

```
1 StandardBloomFilter sbf=new StandardBloomFilter(0.001, 10000,"
    Murmur3");
```

Inoltre, per lasciare una maggiore flessibilitá sono stati implementati altri costruttori che permettono, ad esempio, di costruire lo SBF a partire da una lista di elementi.

Tutti i costruttori a loro volta richiamano il metodo initialize() nel quale vengono inizializzate i parametri m e k secondo quanto descritto nella sezione precedente; una volta definito il valore di m viene costruito un oggetto BitSet, un array di bit di dimensione m, impostati inizialmente tutti a false(0).

```
1 private boolean initialize(double aFP,int expectedNElement, String
    typeHashing){
2     n=expectedNElement;
3     fp_obj=aFP;
4     m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2),
          2));
5     k=(int)Math.round((double)m/(double)n* Math.log(2));
6     this.typeHashing=typeHashing;
7     this.set= new BitSet(m);
8     return true;
9   }
```

Per quanto riguarda le operazioni ammissibili, lo SBF ammette solamente queste due:

- **add(x)** Questa operazione permette di inserire l'elemento x nel SBF e viene eseguita in tempo O(k), andando a settare ad 1 il valore di ogni bit di m con indice $h_k(x)$ $\forall$ k=1, ..., k; se tale bit è giá settato ad 1 allora abbiamo trovato una collisione che non comporta alcun problema.
  Nella classe implementata ci sono una serie di metodi add() che permettono di inserire varie tipologie differenti di elementi; c'è anche un metodo addAll che permette di inserire tutti gli elementi di una lista.

```
1 public void add(byte[] element){if(checkAdd()) addLocal(
    element);}
2 public void add(int c){if(checkAdd()) this.addLocal(BigInteger
    .valueOf(c).toByteArray());}
3 public void add(String c){if(checkAdd()) this.addLocal(c.
    getBytes(charset));}
4 public void addAll(List c){if(checkAdd()) for(Object el : c){
    this.addLocal(el.toString().getBytes(charset));}}
```

Ognuno di questi metodi converte i singoli elementi da inserire in un array di byte e va a richiamare il metodo addLocal() che va a eseguire l'operazione di add() descritta, andando a modificare a true(1) il bit di m di indice $h_k(x)$ $\forall$ k=1, ..., k; viene inoltre incrementato un contatore numberOfAddedElement che tiene traccia del numero di elementi rappresentati nel SBF.

```
1 private void addLocal(byte[] element){
2   for(int j=0;j<k;j++){
3     set.set(BFHashing.getHashing(element, j, m, this.
          typeHashing), true);
4   }
5   numberOfAddedElement++;
6 }
```

- **lookup(x)** Questa operazione permette di valutare se un elemento x è rappresentato all'interno del SBF in tempo O(k). L'idea è quella di andare a valutare se esiste almeno un bit di m di indice $h_k(x)$ $\forall$ k=1, ..., k che è uguale a 0, l'elemento x non è rappresentato all'interno del SBF; altrimenti, se sono tutti uguali a 1, l'elemento c'.
  Nella classe sono stati implementati, in maniera simile a quanto fatto per l'operazione di add(), una serie di metodi lookup() che permettono di ricercare tipologie differenti di elementi.

```
1 public boolean lookup(String c){return this.lookup(c.getBytes(
      charset));}
2 public boolean lookup(int c){return this.lookup(BigInteger.
      valueOf(c).toByteArray());}
```

  Entrambi i metodi convertono l'elemento in un array di byte e ritornano il valore booleano true o false che indica la presenza o meno dell'elemento nell SBF. Questo valore è ottenuto invocando il metodo lookup(byte[] element) che va a effettuare l'operazione di lookup descritta precedentemente.

```
1 public boolean lookup(byte[] element){
2   for(int j=0;j<k;j++)
3   if(set.get(BFHashing.getHashing(element, j, m, this.
        typeHashing))==false)return false;
4   return true;
5 }
```

Infine, sono stati implementati un paio di metodi che calcolano il valore della probabilitá di ottenere un falso positvo: il primo lo calcola considerando il numero atteso di elementi, mentre il secondo considerando il numero di elementi rappresentati; la formula utilizzata è quella descritta nella sezione precedente. Questi due metodi sono stati implementati poichè successivamente verrá analizzato l'andamento della probabilitá di ottenere un FP mano a mano che si aggiungono gli elementi, e anche per verificare che tale probabilitá sia vicina quella desiderata.

```
public double getProbabilityFP(){
    return Math.pow((1 - Math.exp(-k * (double) n / (double) m)), k
        );
  }
public double getProbabilityFPReal(){
    return Math.pow((1 - Math.exp(-k * (double)
        numberOfAddedElement/ (double) m)), k);
  }
```

2.5 Counting Bloom Filter

Il Counting Bloom Filter(CBF)[1] è una delle prime varianti dello SBF. In particolare, questa struttura dati utilizza non piú un vettore di bit, ma un vettore di contatori per rappresentare gli elementi. I contatori devono essere numeri interi, e come si vedrá nella sezione successiva, lo spazio occupato dalle istanze della classe CountingBloomFilter è superiore rispetto a quello occupato dal SBF e varia a seconda del tipo di contatore che si usa (int, short, byte). Il vantaggio principale è che il CBF permette di effettuare una nuova operazione, delete(x), la quale permette di cancellare un elemento x; questa operazione è particolamente utile nei casi in cui è necessario tenere aggiornati i dati inseriti, per poter eliminare quelli non piú necessari e senza dover ricostruire tutte le volte uno SBF simile a quello di partenza.

L'implementazione della classe CountingBloomFilter è stata fatta a partire dalla classe relativa allo SBF facendo le opportune modifiche. La modifica essenziale è stata quella di sostituire il BitSet con un array di contatori i cui valori sono stati impostati di default a 0; poichè lo spazio occupato dai contatori dipende dal loro tipo (int 32bit, short 16bit, byte 8 bit), e quindi dal range di valori che possono assumere, ho voluto lasciare la possibilitá di far specificare in fase di costruzione dell'oggetto CBF il tipo di contatore che si vuole utilizzare.

```
CountingBloomFilter cbf_b=new CountingBloomFilter(0.001, 10000,"
    Murmur3",Byte.class);
CountingBloomFilter cbf_s=new CountingBloomFilter(0.001, 10000,"
    Murmur3",Short.class);
CountingBloomFilter cbf_i=new CountingBloomFilter(0.001, 10000,"
    Murmur3",Integer.class);
```

Il metodo per l'inizializzazione dei parametri è stato cosí modificato:

```
private void initialize(double aFP,int expectedNElement,String
    typeHashing,Class<?> cl){
  n=expectedNElement;
  fp_obj=aFP;
  m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2), 2))
      ;
  k=(int)Math.round((double)m/(double)n* Math.log(2));
  this.typeHashing=typeHashing;
  switch(cl.getSimpleName()){
    case "Byte":{setByte=new byte[m];setShort=null;setInt=null;
        break;}
```

```
9       case "Short":{setShort=new short[m];setByte=null;setInt=null;
            break;}
10      case "Integer":{setInt=new int[m];setShort=null;setByte=null;
            break;}
11    }
12    cl=null;
13  }
```

Invece, per quanto riguarda le operazioni ammesse:

- **add(x)** Questa operazione è stata leggermente modificata rispetto lo SBF, a causa della sostituzione del vettore di bit con il vettore di contatori. L'idea di fondo è rimasta la stessa, ma invece che modificare il valore di ogni bit di m di indice $h_k(x)$ $\forall$ k=1, ..., k da 0 a 1 (da false a true), questi per ogni contatore m[$h_k(x)$] $\forall$ k=1, ..., k, questo viene incrementato di 1; il codice qua sotto fa riferimento al caso in cui il vettore di contatori contenga oggetti di tipo int. Il costo di tale operazione sempre O(k).

```
1   private void addLocalInt(byte[] element){
2       for(int j=0;j<k;j++){
3         setInt[BFHashing.getHashing(element, j, m, this.
              typeHashing)]=(int)setInt[BFHashing.getHashing(
              element, j, m, this.typeHashing)]+1;
4       }
5       numberOfAddedElement++;
6     }
```

- **lookup(x)** Non sono state apportate modifiche per questa operazione rispetto alla versione implementata per lo SBF. Il costo di tale operazione sempre O(k).

- **delete(x)** Questa nuova operazione permette di cancellare gli elementi dal CBF e l'idea è quella di andare decrementare di 1 ogni contatore m[$h_k(x)$] $\forall$ k=1, ..., k , sapendo che x è rappresentato all'interno del CBF; il codice qua sotto fa riferimento al caso in cui il vettore di contatori contenga oggetti di tipo int

```
1   private void deleteLocalInt(byte[] element){
2       for(int j=0;j<k;j++){
3         setInt[BFHashing.getHashing(element, j, m, this.
              typeHashing)]=(int)setInt[BFHashing.getHashing(
              element, j, m, this.typeHashing)]-1;
4       }
5       numberOfAddedElement--;
6     }
```

Infine, anche per questa classe sono stati implementati i metodi per rendere pubblici i parametri che definiscono le caratteristiche principale del CBF, e per quanto riguarda la i metodi relativi alla probabilitá di ottenere dei falsi postivi sono stati implementati allo stesso modo di quelli dello SBF poichè l'utilizzo di un vettore di contatori non incide su questo valore.

2.6 Compressed Bloom Filter

Il Compressed Bloom Filter[2] è una variante introdotta per ottimizzare la trasmissione nella rete dei BoomFilter e delle informazioni che rappresentano; l'obiettivo è quello di ridurre lo spazio occupato dal Bloom Filter, mantenendo peró la probabilitá di ottenere un falso positvo richiesta.
Se consideriamo uno SBF di partenza con un BitSet di dimensione m e k funzioni hash, vogliamo comprimere questo oggetto in un Bloom Filter che utilizza un BitSet di dimensione z. La classe CompressedStandard_BF, a differenza delle due precedenti, contiene un costruttore che costruisce il SBF_CMP a partire da uno SBF.

```
1 StandardBloomFilter sbf=new StandardBloomFilter(0.001, 10000,"
      Murmur3");
2 Compressed_SBF sbf_cmp=new Compressed_SBF(sbf);
```

A sua volta il metodo initialize, è stato opportunamente modificato per calcolare il valore ottimale di z, a partire dal valore di m del SBF da comprimere, mantenendo lo stesso valore di fp_obj:

$$z = m \ln 2$$

questo valore permette di risparmiare circa il 30% dello spazio di memoria, tuttavia, come vedremo meglio nella sezione successiva, i tempi delle operazioni sono piú alti poichè è aumentato il numero k di funzioni hash; il metodo initialize è stato cosí modificato:

```
1 private void initialize(double aFP,int expectedNElement, String
      typeHashing){
2   n=expectedNElement;
3   fp_obj=aFP;
4   m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2), 2))
        ;
5   m = z = (int) Math.round(m*Math.log(2));
6   k=(int)Math.round((double)-z/(double)n * Math.log(fp_obj));
7   this.typeHashing=typeHashing;
8   set=new BitSet(m);
9   }
```

Le operazioni ammesse sono quelle possibili nello SBF; queste sono uguali, sia per quanto riguarda l'implementazione che per le caratteristiche: add(x) in tempo O(k) e lookup(x) in tempo O(k).

Infine, per quanto riguarda i valori della probabilitá di ottenere i falsi positivi, anche qua i metodi utilizzati sono gli stessi dello SBF.

In maniera analoga è stata implementata anche la soluzione del Compressed Counting Bloom Filter, dove l'unica differenza da quella appena presentata è considerare la versione CBF invece che quella SBF.

2.7 Dynamic Bloom Filter

La versione Dynamic Bloom Filter(BF_DY)[3] è la variante dinamica dei Bloom Filter. Considerando uno SBF, il problema principale è che superato il numero di elementi massimo da rappresentare e data una certa probabilitá di falsi positivi, se si aggiungono ulteriori elementi la probabilitá degli FP aumenta in maniera molto rapida fino ad 1. Per ovviare a questo problema ci possono essere due strategie: la prima è quella di bloccare il numero di inserimenti fino al numero massimo di elementi che è possibile rappresentare, l'altro è quello di utilizzare un SBF_DY.

Questa struttura dati è costituita da un inseme dinamico di SBF tutti con le stesse caratteristiche; l'idea è che ogni qual vota un SBF è saturo di elmenti viene creato un nuovo SBF vuoto e aggiunto all'insieme per permettere ulteriori inserimenti.
La classe Dynamic_BF implementa la soluzione descritta a partire dalla definizione dei costruttori che, in generale, richiedono gli stessi parametri dello SBF, ma dove il numero atteso di elementi non è piú il numero massimo di elementi rappresentabili, ma la capacitá massima di ogni singolo SBF.

```
1 Dynamic_SBF sbf_dy=new Dynamic_SBF(0.001,1000);
```

Per quanto riguarda le operazioni possibili, si ha che quelle possibili sono le stesse dello SBF, ma con una implementazione un po' differente:

- **add(x)** Questa operazione va ad aggiungere l'elemento x nel primo SBF non saturo; il primo SBF non saturo è sempre l'ultimo aggiunto all'insieme. Ogni qual volta si deve inserire un elemento x si va a verificare che effettivamente non sia pieno: se è pieno creo un altro SBF e aggiungo x a quello appena creato, altrimenti aggiungo x; il costo di questa operazione rimane O(k)

```
1  private void addLocal(byte[] element){
2    if(bfList.get(this.indexActiveBF).isFull()==false){
3      bfList.get(this.indexActiveBF).add(element);
4      numberOfAddedElement++;
5    }else{
6      boolean b=addBloomFilter();
7      addLocal(element);
8    }
9  }
10 private boolean addBloomFilter(){
11   bfList.add(new StandardBloomFilter(fp_obj,c));
12   indexActiveBF=bfList.size()-1;
13   return true;
14 }
```

- **lookup(x)** Questa operazione permette di cercare la presenza di x in almeno uno degli SBF disponibili; l'idea è quella di cercare iterativamente la presenza di x in ogni SBF e se trovato ritornare il valore true, altrimenti

false. Il costo di questa operazione non è piú O(k), ma dipende dal numero s di SBF presenti per cui è O(k*s).

```
1  public boolean lookup(byte[] element){
2    for(int j=0;j<this.bfList.size();j++){
3      if(bfList.get(j).lookup(element)==true)return true;
4    }
5    return false;
6  }
7  public double getProbabilityFPReal(){
8    double fpProb=0;
9    if(this.c<=this.numberOfAddedElement)fpProb=1-Math.pow(1-
         Math.pow(1-Math.exp(((double)-1*this.getK()*this.
         getCapacity())/(double)this.getM()), this.getK()), this.
         getNumberOfBloomFilter());
10   else fpProb=bfList.get(0).getProbabilityFPReal();
11   return fpProb;
12 }
```

Infine, anche il valore della probabilitá degli FP è diversa: finchè c'è un solo SBF disponibile, questa è uguale a quella dello SBF, altrimenti tale valore è espresso dalla seguente formula:

$$fp = \begin{cases} 1-(1-e^{-k\frac{n}{m}})^k & s=1 \\ 1-(1-(1-e^{-k\frac{n}{m}})^k)^{\lfloor \frac{nAddedElement}{c} \rfloor}(1-(1-e^{-k\frac{nAddedElement-c\lfloor \frac{nAddedElement}{c} \rfloor}{m}})^k) & s>1 \end{cases}$$

dove:

- c è la capacitá dei singoli SBF;
- nAddedElement è il numero di elementi aggiunti nello SBF_DY;
- s è il numero di SBF utilizzati

$$s = \lceil \frac{nAddedElement}{c} \rceil$$

```
1  public double getProbabilityFPReal(){
2    if(this.c<=this.numberOfAddedElement)
3      return 1 -
4        (Math.pow(1-Math.pow(1-Math.exp(((double)(-1*this.getK()*this
             .c))/(double)this.getM()), this.getK()), Math.ceil(
             numberOfAddedElement/c)))*
5        (1-Math.pow(1-Math.exp(-1.0*this.getK()*((double)(
             numberOfAddedElement-c*Math.ceil(numberOfAddedElement/c))
             /((double)this.getM()))), this.getK()));
6    else return bfList.get(0).getProbabilityFPReal();
7  }
```

L'andamento di tale probabilitá rispetto a quella dello SBF è molto diversa: come giá detto finchè abbiamo che s=1, i due andamenti sono uguali, ma dopo questo valore aumenta sempre in maniera esponenziale, ma molto piú lentamente rispetto alla curva dello SBF; nella prossima sezione verrá mostrato nel dettaglio tale confronto.

Analogamente per quanto fatto con il Compressed Bloom Filter, è stata implementata sia la versione Dynamic anche per il CBF; questa peró non supporta l'operazione di cancellazione poichè, per la versione implementata, non è possibile sapere esattamente su quale degli s CBF un elemento è stato inserito per poterlo cancellare.

2.8 Spectral Bloom Filter

Gli Spectral Bloom Filter[4] sono una particolare variante dei CBF in cui è possibile effettuare una nuova operazione: frequency(x). L'implementazione di tale struttura dati puó essere fatta utilizzando due algoritmi differenti: Minimum Insertion(MI) e Recurring Minimum(RM).

Le operazioni possibili su questa struttura dati dipendono molto dal tipo di algoritmo utlizzato, ad eccezione di quella di lookup() che è uguale per entrambi ed è la stessa utlizzata nel CBF. Anche per quanto riguarda il valore della probabilitá di ottenere un falso positivo, questo è lo stesso di quello visto nel CBF e nel SBF.

2.8.1 Minimum Insertion

L'idea di questo algoritmo è quello di andare a rappresentare gli elementi dentro il il Bloom Filter, migliorando l'operazione di add() per poter recuperare la frequenza di ogni singolo elemento. Per cui, ci che distingue questo tipo di Bloom Filter dal CBF :

- **add(x)** questa operazione, a differenza di quanto accade con i CBF, non va a incrementare di 1 ogni contatore m[h_k(x)] $\forall$ k=1, ..., k, ma solamente quelli il cui valore è il minimo. Il codice seguente mostra come è stata implementata questa operazione:

```
1  private void addLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      ArrayList<Integer> a= map.get(setInt[key]);
6      try{
7        a.add(key);
8      }catch(Exception e){
9        a=new ArrayList<Integer>();
10       a.add(key);
11     }
12     map.put(setInt[key], a);
13   }
14   int min= Collections.min(map.keySet());
15   ArrayList<Integer> indexList=map.get(min);
16   for(int j : indexList) setInt[j]=(int)(min+1);
17   numberOfAddedElement++;
18   }
```

Inizialmente viene costruito una tabella Hash che ha per chiave i valori assunti dagli m[h_k(x)] contatori e come valore un vettore contenente gli indici h_k(x). Una volta riempito, si trova il minimo valore tra la lista delle chiave e si vanno ad incrementare solamente gli indici associati a questo valore minimo. Tuttavia, nonostante le modifiche, l'operazione viene sempre eseguita in tempo costante O(k)

- **frequency(x)** questa nuova operazione ritorna, come giá detto, la frequenza dell'elemento x. La frequenza di x è data dal valore minimo tra i contatori m[h_k(x)] $\forall$ k=1, ..., k. Il codice che implementa questa operazione è il seguente:

```
1 private int frequencyLocalInt(byte[] element){
2   int minValue=0;
3   for(int j=0;j<k;j++){
4     int hash=setInt[BFHashing.getHashing(element, j, m, this.
          typeHashing)];
5     if(hash==0) return 0;
6     else if((minValue>hash || j==0 ) && hash!=0)minValue=hash;
7   }
8   return minValue;
```

Anche questa operazione viene fatta in tempo costante e il suo costo è O(k).

- **delete(x)** questa operazione non è supportata, poichè andare a decrementare di 1 solamente il minimo tra i contatori m[h_k(x)] $\forall$ k=1, ..., k porterebbe alla possibilitá di ottenere dei Falsi Negativi(FN).

2.8.2 Recurring Minimum

Questo altro algoritmo, invece, permette di costruire una struttura dati che, a differenza del MI, ammette l'operazione di delete(x) e utilizza un secondo vettore in cui verranno salvati quegli elementi che hanno un recurring minimum. Per cui, la fase di inizializzazione cambia leggermente, aggiungendo questo secondo vettore di contatori che conterrá al massimo il 20% degli elementi rappresentabili(n) nell'altro vettore.

```
1  private void initialize(double aFP,int expectedNElement,String
       typeHashing,Class<?> cl){
2    n=expectedNElement;
3    n2 = (int)(n*(20.0f/100.0f));
4    fp_obj=aFP;
5    m=(int)Math.round(-n * Math.log(fp_obj)/Math.pow(Math.log(2), 2))
         ;
6    k=(int)Math.round((double)m/(double)n* Math.log(2));
7    m2=(int)Math.round(-n2 * Math.log(fp_obj)/Math.pow(Math.log(2),
       2));
8    k2=(int)Math.round((double)m2/(double)n2* Math.log(2));
9    this.typeHashing=typeHashing;
10   switch(cl.getSimpleName()){
11     case "Byte":{setByte=new byte[m];setShort=null;setInt=null;
         setByte2=new byte[m2];setShort2=null;setInt2=null;break;}
```

```
12      case "Short":{setShort=new short[m];setByte=null;setInt=null;
            setShort2=new short[m2];setByte2=null;setInt2=null;break;}
13      case "Integer":{setInt=new int[m];setShort=null;setByte=null;
            setInt2=new int[m2];setShort2=null;setByte2=null;break;}
14    }
15    cl=null;
16    }
```

Questo algoritmo inoltre comporta alcune modifiche per le seguenti operazioni:

- **add(x)** questa operazione, a differenza del precedente algoritmo, non va piú ad incrementare di 1 ogni contatore m[h_k(x)] $\forall$ k=1, ..., k e poi valutare se tale valore minimo è ricorrente o meno. Se è unico va ad aggiungere x al secondo vettore incrementando ogni contatore m_2[h_2k2(x)] $\forall$ k_2=1, ..., k_2 con il valore minimo tra i contatori m[h_k(x)] $\forall$ k=1, ..., k; se l'elemento è giá rappresentato in questo secondo array, allora si incrementano i contatori m_2[h_2k2(x)] di 1.

```
1  private void addLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      setInt[key]=setInt[key]+1;
6      ArrayList<Integer> a= map.get(setInt[key]);
7      try{
8        a.add(key);
9      }catch(Exception e){
10       a=new ArrayList<Integer>();
11       a.add(key);
12     }
13     map.put(setInt[key], a);
14   }
15   int min= Collections.min(map.keySet());
16   ArrayList<Integer> indexList=map.get(min);
17   if(indexList.size()==1){
18     int increment=min;
19     for(int j=0;j<k2;j++){
20       if(setInt2[BFHashing.getHashing(element, j, m2,
             typeHashing)]==0){
21         increment=1;
22         break;
23       }
24     }
25     for(int j=0;j<k2;j++){
26       int key=BFHashing.getHashing(element, j, m2, typeHashing
             );
27       setInt2[key]=(setInt2[key]+increment);
28     }
29     uniqueItems++;
30   }
31   numberOfAddedElement++;
32 }
```

- **frequency(x)** questa nuova operazione va a valutare la frequenza dell'elemento x. Questa è data dal valore minimo (min) dei contatori contatore m[h_k(x)]

$\forall$ k=1, ..., k, se il valore trovato non è unico viene restituito tale valore; altrimenti si va a controllare la presenza di x nel secondo vettore: se è presente viene restituito il minimo valore tra i $m_2[h_2k2(x)]$ $\forall$ k_2=1, ..., k_2 contatori, altrimenti il valore min trovato precedentemente. Il costo di questa operazione è O(k).

```
1  private int frequencyLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      if(setInt[key]==0) return 0;
6      ArrayList<Integer> a= map.get(setInt[key]);
7      try{
8        a.add(key);
9      }catch(Exception e){
10       a=new ArrayList<Integer>();
11       a.add(key);
12     }
13     map.put(setInt[key], a);
14   }
15   int min= Collections.min(map.keySet());
16   ArrayList<Integer> indexList=map.get(min);
17   if(indexList.size()==1){
18     int min2=0;
19     for(int j=0;j<k2;j++){
20       int val=setInt2[BFHashing.getHashing(element, j, m2,
             typeHashing)];
21       if(val==0) return min;
22       else if(val<min2 || j==0) min2=val;
23     }
24     return min2;
25   }else{
26     return min;
27   }
28 }
```

- **delete(x)** questa operazione effettua gli stessi step dell'operazione di add, ma invece di incrementare i vari contatori questi vengono decrementati.

```
1  private void deleteLocalInt(byte[] element){
2    HashMap<Integer,ArrayList<Integer>> map=new HashMap<Integer,
         ArrayList<Integer>>();
3    for(int j=0;j<k;j++){
4      int key=BFHashing.getHashing(element, j, m, typeHashing);
5      ArrayList<Integer> a= map.get(setInt[key]);
6      try{
7        a.add(key);
8      }catch(Exception e){
9        a=new ArrayList<Integer>();
10       a.add(key);
11     }
12     map.put(setInt[key], a);
13   }
14   int min= Collections.min(map.keySet());
15   ArrayList<Integer> indexList=map.get(min);
```

```
16    for(int j : indexList) setInt[j]=(int)(min-1);
17    if(indexList.size()==1){
18      int increment=min;
19      for(int j=0;j<k2;j++){
20        if(setInt2[BFHashing.getHashing(element, j, m2,
              typeHashing)]==0){
21          increment=1;
22          break;
23        }
24      }
25      for(int j=0;j<k2;j++){
26        int key=BFHashing.getHashing(element, j, m2, typeHashing
              );
27        setInt2[key]=(setInt2[key]-increment);
28      }
29      this.uniqueItems--;
30    }
31    numberOfAddedElement--;
32  }
```

3 Analisi

In questa sezione verranno descritti i risultati ottenuti da quanto implementato in questo progetto: nella prima parte, verrá analizzato e confrontato lo spazio di memoria occupato; mentre nella seconda verranno confrontati distintamente i tempi medi per le operazioni di add(), lookup(), frequency() e delete().

3.1 Spazio

Per questa analisi si è voluto andare a valutare lo spazio occupato da ogni singolo Bloom Filter fissando il valore della probabilitá di ottenere falsi positivi(fp_obj) ed incrementando iterativamente la capacitá(n); questo test è implementato nella classe Test_Space.
Per prima cosa sono stati fissati alcuni valori quali la capacitá minima, quella massima e la probabilitá desiderata. Successivamente, dentro ad un ciclo for che itera dalla capacitá minima a quella massima incrementando tale variabile(size) di un certo valore, vengono inizializzati i Bloom Filter, e anche due array (uno di stringhe e uno di byte). Questi ultimi verranno utilizzati per valutare l'ammontare guadagno, in termini di memoria, ottenuto con i Bloom Filter.
Per calcolare lo spazio occupato dai singoli oggetti è stata utlizzata una libreria in java che si chiama SizeOf[7].

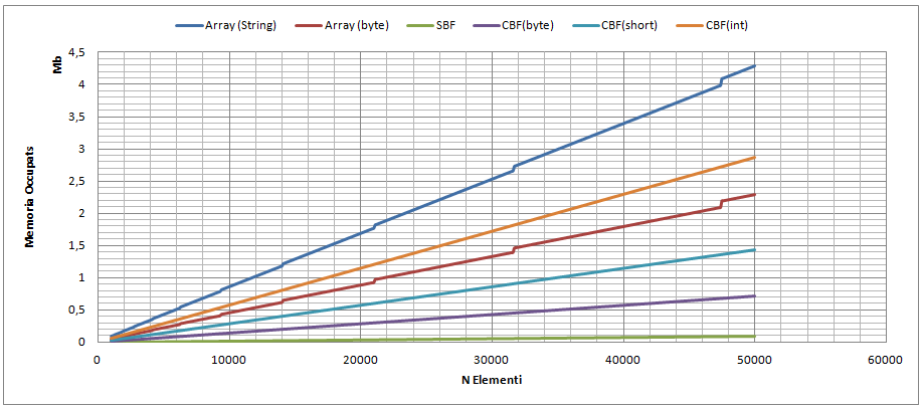


Figure 1: Arrays vs Bloom Filters

Il grafico in figura 1 mostra gli andamenti della memoria occupata dai signoli oggetti al variare del numero di elementi rappresentati; tutti gli andamenti sono lineari, ma con pendenze molto diverse l'una dall'altra. Come ci si poteva aspettare, la struttura dati che costa di piú è quella dell'array di stringhe, seguita dal CBF di interi, mentre quella che costa meno è lo SBF. Quest'ultimo costa, in termini di spazio, solamente m bits, mentre le versioni CBF, che anche loro

hanno un vettore di dimensione m, sono piú costose perchè ogni elemento necessita di 8, 16, 32 bits a seconda del tipo di contatore che si usa (byte, short, int).

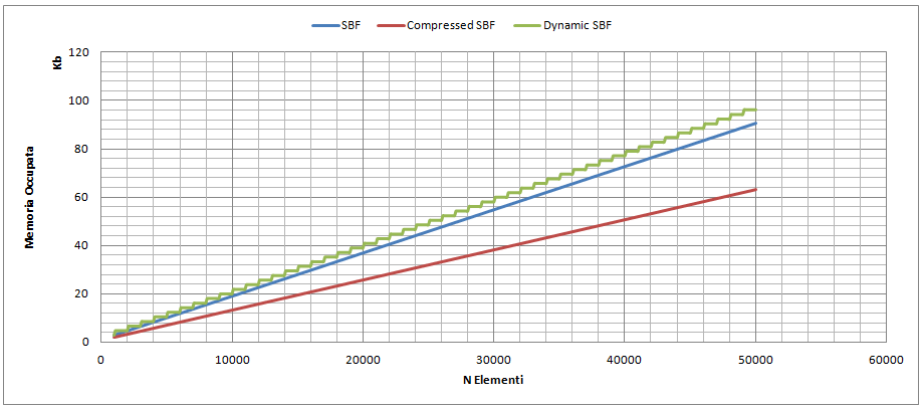


Figure 2: Standard Bloom Filters

Considerando lo SBF, il grafico in figura 2 permette di confrontare lo spazio utilizzato dallo SBF con la sua versione dinamica e quella compressa. Come ci si poteva aspettare la versione compressa riduce lo spazio, e il gap con la versione standard aumenta all'aumentare degli elementi rappresentabili; questo perchè il vettore di bit non è piú di dimensione m, ma z con $z = m \ln 2$.

Invece per quanto riguarda la versione dinamica, lo spazio occupato è molto simile a quello della versione standard, ma con l'aumentare del numero di elementi tende ad essere piú elevato. L'andamento è un po' diverso, a scalini perchè quando si raggiunge la saturazione dell'ultimo SBF ne viene aggiunto un altro di dimensione m, quindi lo spazio totale costa s * m bit, dove s è il numero di SBF utilizzati.

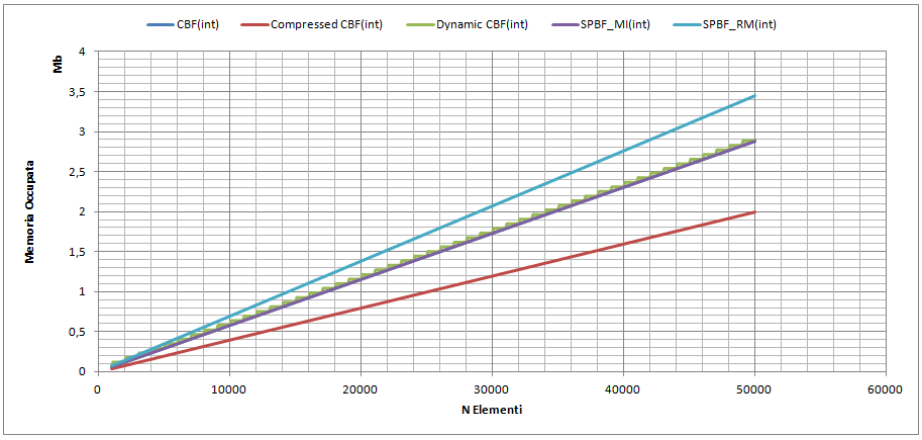


Figure 3: Counting Bloom Filters

Analogamente a quanto appena detto per lo SBF le sue due varianti, si puó fare lo stesso ragionamento considerando il CBF e le sue estensioni. Guardando il grafico in figura 3 si puó vedere come la versione compressa e quella dinamica abbiano lo stesso andamento, rispetto alla versione standard, osservato per il SBF.
Infine, si puó confrontare il CBF anche con le altre due versioni relative allo Spectral Bloom Filter. Dal grafico si vede bene come la versione che utlizza l'algoritmi Minimum Insertion (SPBF_MI) occupo lo stesso spazio della versione CBF, poichè entrabe utlizzano un vettore di contatori di dimensione m. Tuttavia, gli andamenti sono diversi se consideriamo l'altra versione, quella che utlizza l'algoritmo del Recurring Minimum. Quest'ultima risulta essere piú costosa, e all'aumentare del numero di elementi il gap con la versione standard cresce costantemente, poichè al suo interno c'è un secondo vettore di contatori di dimensione r; per cui il costo totale è m + r.

3.2 Tempo

Questa serie di esperimenti vuole andare a confrontare i tempi medi delle operazioni che possono essere effettuate in questa particolare struttura dati.

3.2.1 add()

La prima operazione ad essere analizzata è quella di add(x); questa è presente in tutte le versioni dei Bloom Filter, ma con alcune variazioni tra una versione e l'altra come giá descritto precedentemente.

Questo test è stato implementato nella classe Test_Add, dove iterativamente vengono inizializzate le istanze dei varie versioni dei Bloom Filter e anche un semplice array. Dopo ogni inizializzazione, viene aggiunto un blocco di n elementi e viene monitorato il tempo necessario per il completamento di tale operazione; questo tempo viene diviso per il numero di elementi n, in modo da ottenere un tempo medio dell'operazione di add(x). Ad ogni iterazione n viene incrementato fino ad un massimo di 50000 elementi.

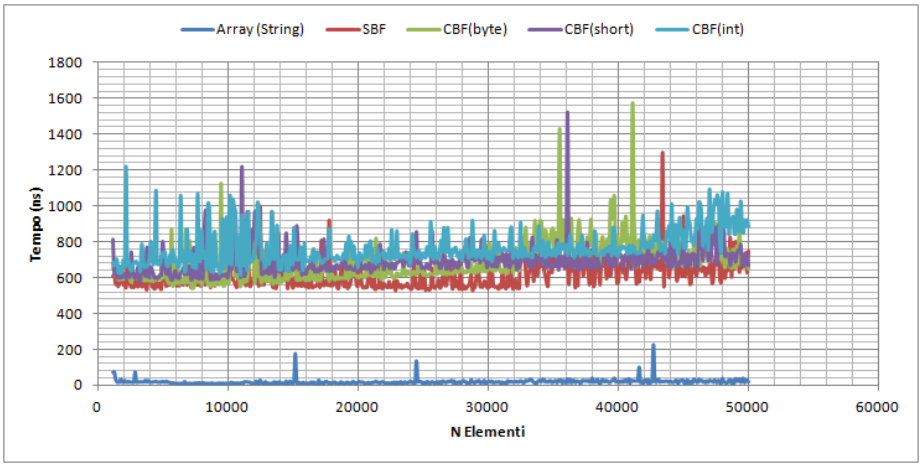


Figure 4: Add() Bloom Filters

Il grafico in figura 4 mostra i tempi medi (nanosecondi) delle operazioni di add() e si puó notare come, per tutte le strutture dati, il tempo sia costante. Nel caso dell'array il tempo è il piú veloce e costa O(1), mentre per quanto riguarda i SBF e CBF è un po' piú alto poichè dipende dal numero di funzioni hash k; il costo è di O(k).

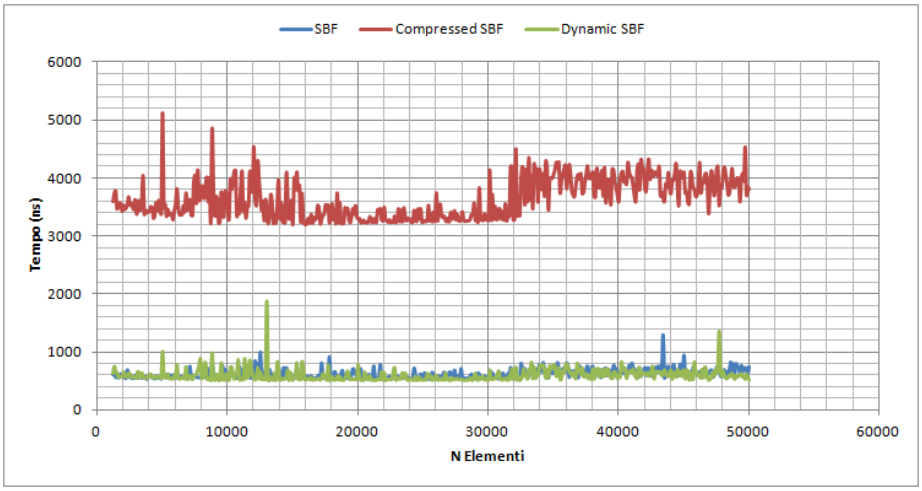


Figure 5: Add() Standard Bloom Filters

Invece, dal grafico in figura 5 si possono confrontare i tempi medi per le varie implementazioni dello SBF, quella standard, quella dinamica e quella compressa. Poichè come detto prima i tempi dipendono dal numero k di funzioni hash, la versione dimanica e quella standard viaggiano sugli stessi tempi, mentre quella compressa che ha un numero piú elevato di funzioni hash ha un tempo medio molto piú alto, ma sempre costante.

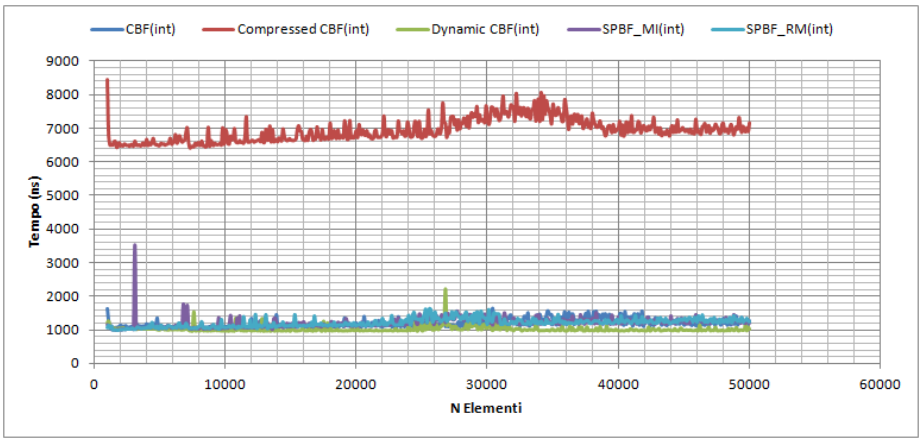


Figure 6: Add() Counting Bloom Filters

Lo stesso discorso lo si puó fare considerando il CBF e le sue estensioni, e la figura 6 mostra tali andamenti. I tempi sono costanti in tutte le versioni, ma sono nettamente superiori nella versione compressa. Tuttavia, una analisi interessante è data dal confronto tra lo SPBF_MI e lo SPBF_RM. Come detto in

precedenza, il primo dovrebbe avere un algoritmo di inserimento piú performante rispetto al secondo.

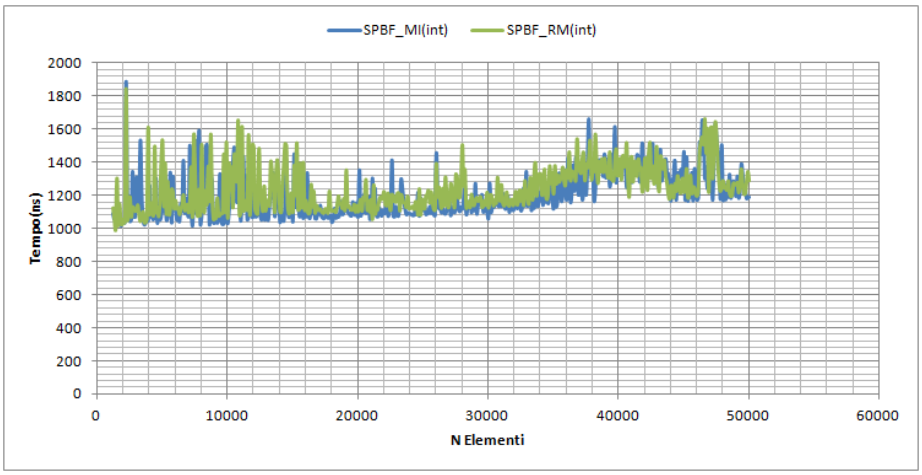
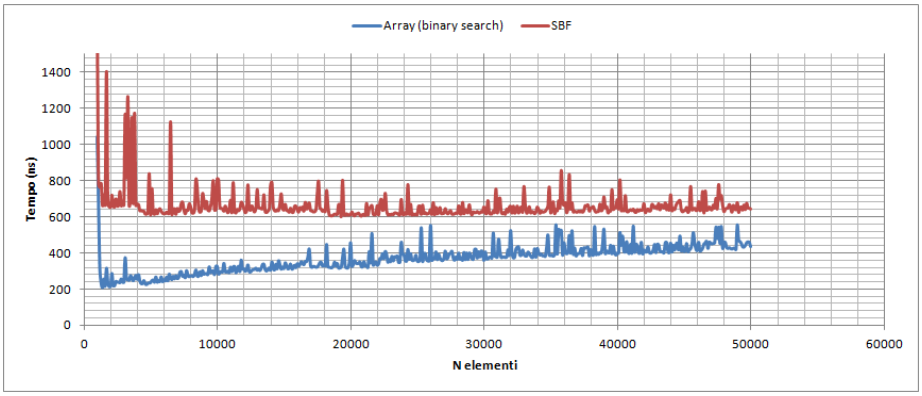


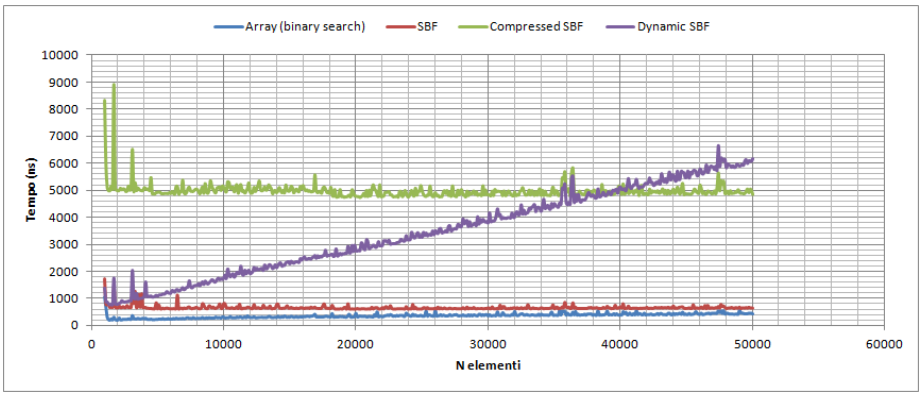
Figure 7: Add() Counting Bloom Filters

Il grafico della figura 7 mostra come i tempi siano molto simili l'uno rispetto l'altro, ma quasi sempre quelli ottenuti utilizzando l'algoritmo del Minimum Insertion sono leggermente migliori.

3.2.2 lookup()

L'operazione di lookup(x) è presente in tutte le versioni dei Bloom Filter implementate; ad eccezione dei Dynamic Bloom Filter, questa operazione ha sempre la stessa implementazione.
Il test per valutare i tempi medi di tali operazioni è stato implementato nella classe Test_Space, dove iterativamente vengono inizializzate le varie tipologie di Bloom Filter, un array che contiene gli stessi elementi dei vari BF e un array di elementi da ricercare. A quest'ultimo vengono aggiunti in maniera casuale alcuni degli elementi contenuti nei BF e altri n elementi, in modo da ottenere qualche risultato positivo e qualche negativo dall'operazione in esame. L'operazione di lookup(x) sull'array di confronto è stata fatta utilizzando l'algoritmo della ricerca binaria, ordinando opportunamente l'array. I grafici seguenti mostrano i tempi medi registrati nel test.





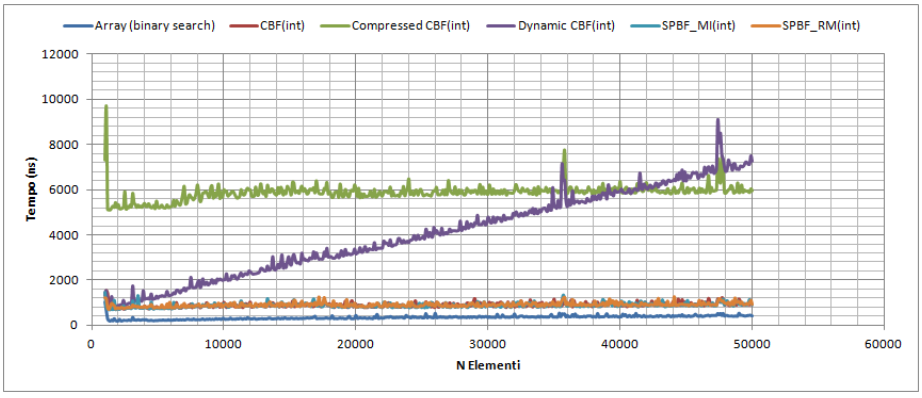


Figure 8: Lookup() Bloom Filters

Il costo dell'operazione sull'array è di $O(\log n)$, infatti, dal grafico 8 si puó osservare una curva che all'aumentare del numero di elementi si stabilizza sullo

stesso valore, mentre per lo SBF l'andamento è piú lineare. Osservando meglio gli altri due grafici, e non considerando i Dynamic Bloom Filter, l'andamento dei tempi è simile: i tempo sono costanti, e non dipendono dal numero di elementi ma dal numero di funzioni hash k; il costo di tale operazione è quindi O(k).
La differenza che si nota tra le versioni standard e quelle compresse è data dal numero di funzioni hash; questo andamento è simile a quanto giá osservato nell'operazione di add().
Un comportamento diverso ce l'ha la il Dynamic Bloom Filter, poichè il costo di tale operazione dipende dal numero k di funzioni hash, ma anche dal numero s di SBF presenti all'interno. Questo valore s,a sua volta, dipende dal numero di elementi e dalla capacitá dei singoli SBF; quindi fissata la capacitá, all'aumentare del numero di elementi aumenta il valore di s e il costo di tale operazione è O(s*k).

3.2.3 frequency()

Questa particolare operazione è possibile solo nello Spectral Bloom Filter; essa ritrorna un numero intero che indica la frequenza dell'elemento x. Il test è stato eseguito in parallelo a quello fatto per l'operazione di lookup, monitorando i tempi di tale operazione.

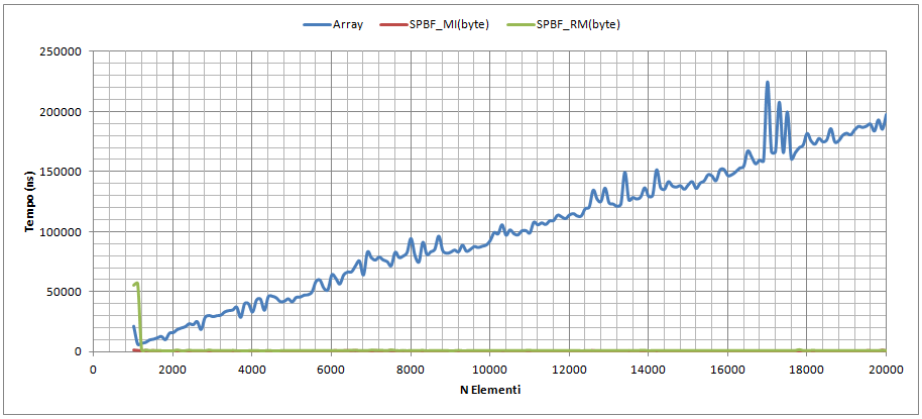


Figure 9: Frequency() Bloom Filters

Nel grafico di figura 9 possiamo veder i tempi medi relativi di questa operazione, su un semplice array e le due versioni di Spectral Bloom Filter. L'andamento dei tempi relativi all'array sono molto superiori rispetto a quelli registrati per le altre due strutture dati; i primi risultano avere un andamento crescente all'aumentare del numero di elementi, mentre i secondi uno costante.

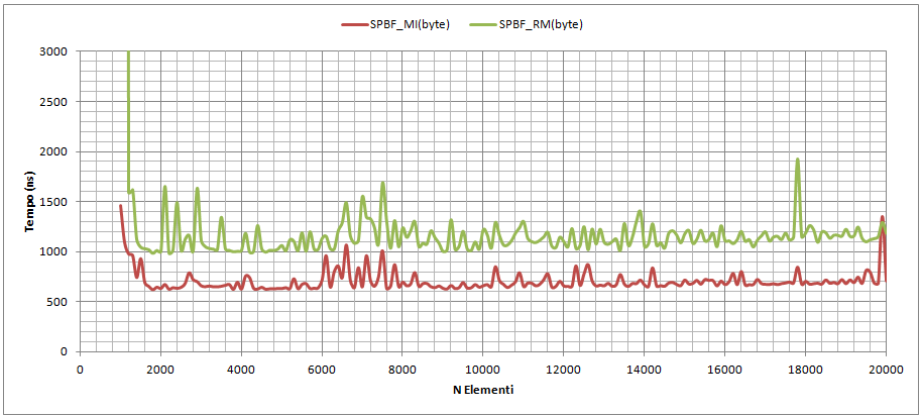


Figure 10: Frequency() Spectral Bloom Filters

La figura 10 evidenzia gli andamenti costanti delle due Spectral Bloom Filter considerati. Come giá detto entrambi hanno un andamento costante, ma quello che utlizza l'algoritmo Recurring Minimun impiega poco piú tempo dell'altro; questo perchè le due implementazioni dei metodi sono diverse e, come spiegato nella sezione precedente, quello relativo alla versione SPBF_RM deve andare a valutare se l'elemento ha un minimo unico oppure no.

3.2.4 delete()

L'ultima operazione da analizzare è quella della delete(x); questa operazione è ammessa solo su alcune tipologie di Bloom Filter: CBF e le sue varianti Compresse e Spectral. Anche per fare questo test, come per quanto accaduto in quelli precedenti, è stata implementata una classe Test_Delete che, all'aumentare del numero di elementi rappresentabili nei vari Bloom Filter, monitora e registra i tempi medi relativi alla cancellazione di un n di elementi.

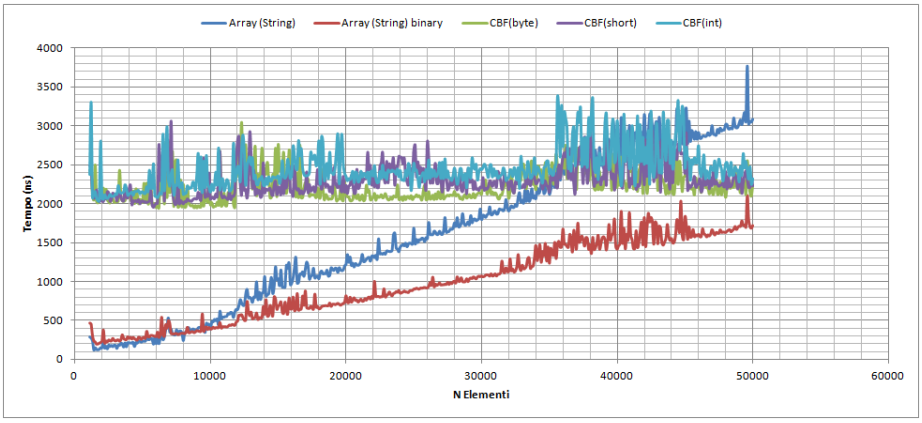


Figure 11: Delete() Bloom Filters

Il grafico della figura 11 mostra i risultati ottenuti dal test. Per quanto riguarda la cancellazione degli elementi dagli array, questa è stata fatta in due diversi modi: il primo è semplicemente l'operazione di remove(x), mentre la seconda considera una versione ordinata dell'array, tramite la ricerca binaria trova l'indice dell'elemento da eliminare(index) e effettua l'operazione di remove(index). Entrambi hanno un andamento crescente all'aumentare della grandezza dell'array, ma il secondo cresce in maniera piú lenta.

Invece, se guardiamo gli andamenti ottenuti sui CBF, questi sono costanti e il costo di tali operazioni è O(k). Confrontando le due strutture dati, la cancellazione da un array risulta essere piú performante fino ad una certa soglia poichè il tempo aumenta all'aumentare del numero di elementi, mentre l'operazione sui CBF risulta essere meno performante sotto una certa soglia, ma costante nel tempo perchè non dipende dal numero di elementi.

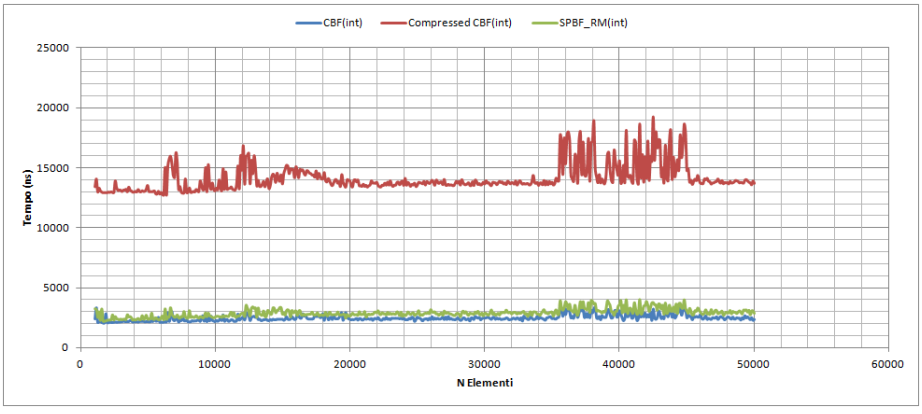


Figure 12: Delete() Counting Bloom Filters

Il grafico 12 ci permette di confrontare gli andamenti dell'operazione sulle 3 diverse tipologie di Counting Bloom Filter: standard, compressa e spectral. In tutti e tre i casi il costo dell'operazione è costante, O(k), e come giá osservato nelle operazioni di lookup() e add(), la versione compressa impiega piú tempo a causa del numero maggiore di k; la versione standard è leggermente pi rapida rispetto a quella Spectral.

3.3 FP probability

In questa sezione verranno confrontati i valori della probabilitá di ottenere un falso positivo(FPP) all'aumentare del numero di elementi rappresentati nei Bloom Filter in questione. Non verranno confrontati tutte le implementazioni dei Bloom Filter fatte, ma veranno considerato solamente lo StandardBloomFilter(SBF), la sua versione dinamica(SBF_DY) e la sua versione compressa (SBF_CMP); questo perch in tutte le altre versioni tale probabilit uguale a quella dello SBF.

Questo esperiemnto vuole mostrare i diversi andamenti della FPP ed è stato implementato nella classe Test_FP. Per prima cosa sono stati definiti gli oggetti relativi alle varie implementazioni dei Bloom Filter; ognuno di esso ha un numero atteso di elementi pari a 1000 e una fp_obj di 0.001.
Una volta inizializzati, questi vengono riempiti iterativamente con 100 elementi alla volta finchè il numero totale di elementi contenuti è 100000. Ad ogni iterazione vengono salvati i valori della FPP relativa ad ogni singolo Bloom Filter su un file csv.

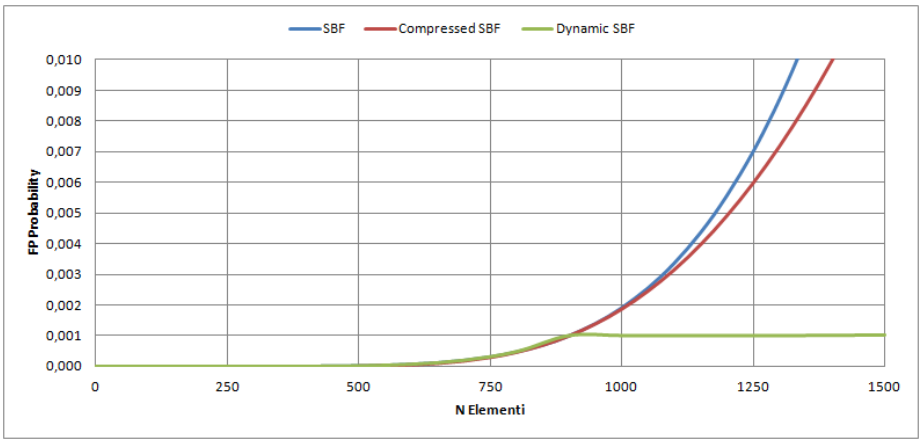


Figure 13: False positive probability

La figura 13 mostra il grafico realativo all'andamento della probabilitá fino al valore desiderato $fp\_obj * 10$, e si puó notare come la versione SBF e quella compressa abbiano circa lo stesso andamento, mentre quella dinamica ha un andamento diverso; inizialmente coincide perchè il numero di SBF dentro la versione dinamica è uguale ad 1, ma poi l'andamento tende ad essere piú lineare quando il numero degli SBF aumenta.

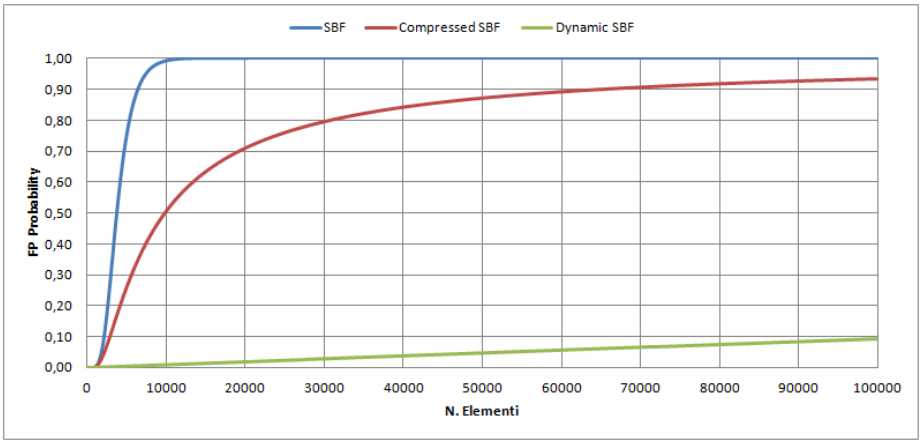


Figure 14: False positive probability

Invece, nel grafico in figura 14 si puó notare l'andamento fino all'inserimento di tutti i 100000 elementi: qua la differenza degli andamenti tra tutte e tre le versioni è piú accentuata:

- SBF la crescita è esponenziale e molto veloce; infatti giá alla rappresentazione di 10000 elementi si raggiunge un valore di FPP = 1;
- SBF_CMP l'andamento è esponenziale ma con una crescita piú lenta rispetto lo SBF; se consideriamo 10000 elementi qua il valore di FPP è di circa 0.5, e alla fine non si raggiunge un valore poco superiore allo 0.9; molto probabilmente, ció dipende dal fatto di avere piú funzioni hash che mappano i vari elementi.
- SBF_DY l'andamento qua invece non è piú esponenziale, ma abbastanza lineare e si vede come questa struttura dati riesca a garantire una bassa FPP, nonostante il grande numero di elementi rappresentati; con 100000 elementi si ha un FPP di circa 0.01. Questo valore rimane comunque superiore a quello desiderato, ma è nettamente inferiore da quelli raggiunti dalle altre due versioni.

4 Minimal Perfect Hash vs Bloom Filter

4.1 Definizione

In questa sezione verrá mostrata una soluzione alternativa ai Bloom Filter, che permette di rappresentare un insieme in una maniera succinta e di effettuare operazioni di lookup in tempo costante.[5]

Questa struttura dati, MPHS, utilizza una funzione hash minimale perfetta per mappare un insieme di n elementi in un range [0,n-1]. Inoltre, viene costruito un array di signature, di dimensione n, dove ogni elemento occupa 2^n_bits bit. Il valore di n_bits dipende dalla probabilitá di ottenere un falso positivo(fp) desiderata:

$$n\_bits = \log_2 \frac{1}{fp}$$

L'implementazione di ció è stata fatta nella classe MPHS. Per l'inizializzazione delle istanze di tale classe, a differenza dei costruttori dei Bloom Filter, sono necessari solamente due parametri: la lista degli oggetti da rappresentare e il valore della probabilitá FP desiderata.

```
1  public MPHS(List<String> aS,double fp){
2    List<String> list=(List<String>)Arrays.asList(new HashSet<String
         >(aS).toArray(new String[0])); //eliminare i duplicati dalla
         lista
3    this.fp_prob=fp;
4    n=list.size();
5    n_bits=Math.log(1.0/this.fp_prob) / Math.log(2);
6    range=(int)Math.ceil(Math.pow(2,n_bits));
7    signature=new int[n];
8    try {
9      mph= new GOVMinimalPerfectHashFunction.Builder<String>().keys(
           list).transform( TransformationStrategies.utf32() ).signed(
            32).build();
10     for (int i=0;i<n;i++) signature[(int)mph.getLong(list.get(i))]=
           getSignature(list.get(i));
11   } catch (IOException e) {
12     e.printStackTrace();
13   }
14 }
```

La funzione minimale perfetta, GOVMinimalPerfectHashFunction, la quale fa parte di una libreria java (Sux4j) sviluppata dal Sebastiano Vigna [6],viene costruita a partire da una lista di chiavi, che in questo caso corrisponde con la lista degli elementi da rappresentare.
Successivamente viene riempito l'array di signature, andando ad inserire la signature dell'elemento x nella cella che ha per indice il valore della funzione hash minmale perfetta associata alla chiave x. La signature viene creata utilizzando una qualsiasi funzione hash con range 2^{n_bits}.

```
1 private int getSignature(Object c){
2   return (int) (c.hashCode() % range);
3 }
```

Come nella versione dello Standard Bloom Filter, su questa struttura dati è possibile effettuare solo operazioni di add e lookup. Tuttavia l'operazione di add è ammessa solamente in fase di costruzione, secondo gli step descritti precedentemente, mentre l'operazione di lookup(x) va a cercare se l'elemento x è rappresentato dalla funzione mimale perfetta. Se non lo è l'elemento non è presente viene restituito false, altrimenti vengono confrontata la signature di x con signature[index] dove index è il valore ottenuto dalla funzione minimale perfetta. Se queste signature sono uguali allora restitusce il valore true, altrimenti false.

4.2 Analisi

In questa sezione verrá messa a confronto la struttura dati MPHS con la versione SBF dei Bloom Filter.

4.2.1 Spazio

Figure 15: Space: MPHS vs SBF

Per quanto riguarda lo spazio occupato, si puó veder dal grafico 15 come l'andamento sia crescente all'aumentare del numero di elementi. Questo perchè lo spazio occupato dalla funzion hash e dall'rray di signature dipende dal numero di elementi da rappresentare. Confrontandola con lo SBF, occupa un po' piú di memoria , ma considerando lo spazio occupato dall'array, il guadagno che si ottiene puó essere paragonato a quello ottenuto dal SFB.

4.2.2 lookup()

L'altro confronto riguarda i tempi medi relativi all'operazione di lookup. Per come è stata implementata a struttura, ci si aspetta che il tempo sia costante all'aumentare del numero di elementi da ricercare.

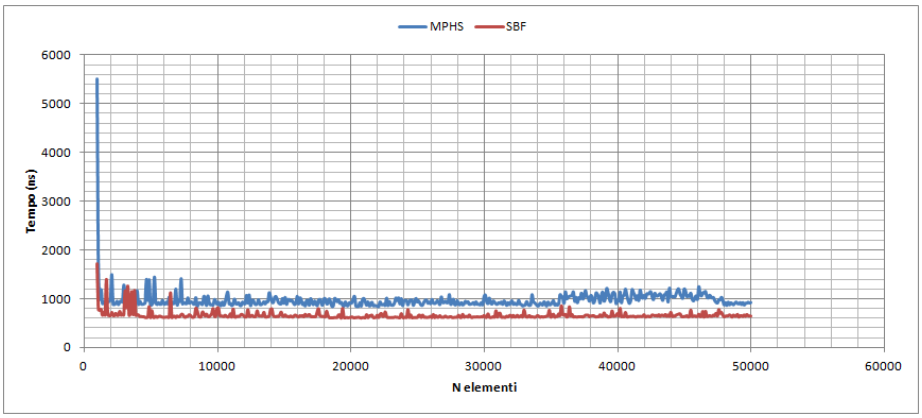


Figure 16: Space: MPHS vs SBF

Il grafico 16, mostra come tale andamento sia costante e di poco superiore a quello ottenuto con lo SBF.

5 Conclusioni

In questo progetto sono state analizzate le performance di una particolare struttura dati che permette, a costo di alcuni falsi positivi, di rappresentare un array in maniera succinta. Dai risultati ottenuti si evince che, se il problema ammette la presenza di alcuni FP, i Bloom Filter sono una buona soluzione per risparmiare spazio e garantire alcune operazione in tempo costante. Inoltre, questa struttura dati pu essere adatattata a seconda delle esigenze del problema, modificandone opportunamente alcuni aspetti per permettere l'implementazione di nuove operazioni.

Infine, si è realizzata una alternativa ai Bloom Filter che permette, anch'essa di rappresentare un array in maniera succinta garantendo l'operazione di lookup() in tempo costante.

Durante la realizzazione del progetto, la difficoltá principale riscontrata è stata nel trovare i valori ottimi dei parametri dei vari Bloom Filter perch cercando tra i vari articoli e blog le implementazioni sono molto variegate e spesso specifiche per i singoli problemi.

References

[1] Broder, Andrei, and Michael Mitzenmacher, *Network applications of bloom filters: A survey*, Internet mathematics 1.4 (2004): 485-509.

[2] Mitzenmacher, Michael, *Compressed bloom filters*, IEEE/ACM Transactions on Networking (TON) 10.5 (2002): 604-612.

[3] Guo, Deke, et al., *The dynamic bloom filters.*, Knowledge and Data Engineering, IEEE Transactions on 22.1 (2010): 120-133.

[4] Cohen, Saar, and Yossi Matias, *Spectral bloom filters*, Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003.

[5] Belazzougui, Djamal, and Rossano Venturini, *Compressed static functions with applications*, Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2013.

[6] Sebastiano Vigna, *Sux4J*, https://github.com/vigna/Sux4J

[7] Nicolas Fafchamps, *SizeOf*, http://sizeof.sourceforge.net/