UNIVERSITÀ
DEGLI STUDI
FIRENZE

Bloom Filter

Gabriele Boddi

UNIVERSITÀ DEGLI STUDI FIRENZE

Indice

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduzione

- **Bloom Filter**: Struttura dati basata sulle *probabilità* che, attraverso l'uso di *Funzioni Hash*, specifica se un elemento si trova *probabilmente* all'interno di un insieme.
  - *Ricerca rapida*: il suo utilizzo consente di ricercare degli elementi con costi approssimabili a valori *lineari*.
  - *Rischio di falsi positivi*: una ricerca molto vasta, senza i dovuti accorgimenti, può comportare la presenza di test che ammettono la presenza di un elemento senza che questo lo sia davvero.

UNIVERSITÀ DEGLI STUDI FIRENZE

Descrizione della struttura

- *Array* di **m** bit con valore iniziale a 0.
- **k** *Funzioni Hash* che permettono la mappatura degli indici dell'array rispetto agli elementi dell'insieme.

Funzionamento della struttura

- ***Aggiunta di un elemento***: dato un elemento *e* in input, si passa sulle funzioni hash per generare degli indici del Bloom Filter con la seguente formula:

$$j = ((h(e) \cdot (i + 1)) + n) \mod m \qquad (1)$$

dove *h(e)* è la funzione hash sull'elemento *e*, *i* indica l'*i*-esima funzione che si sta utilizzando ed *n* la grandezza dell'insieme di studio.
Si imposta quindi la posizione *j*-esima dell'array ad 1.

UNIVERSITÀ DEGLI STUDI FIRENZE

Funzionamento della struttura

- ***Controllo della presenza di un elemento***: Attraverso la medesima formula, si va a controllare tutte le posizioni dell'array che corrispondono agli indici ottenuti, passando l'elemento che si vuole controllare in input, sulle funzioni hash.
- Se tutti i bit sono a 1, *probabilmente* l'elemento è presente nell'insieme.
- Se anche solo un bit è a 0, *certamente* l'elemento non si trova nell'insieme.

Possibili ottimizzazioni

- La tolleranza dei Falsi Positivi deve essere tenuta strettamente sotto la soglia dell'*1%*.
- Se si ha a che fare con un insieme di n elementi, il numero ottimale di bit del Bloom Filter è il seguente:

$$m\_ottimo = \frac{|n \cdot ln(0.01)|}{2 \cdot ln(2)} \tag{2}$$

- Da tale risultato si può quindi definire anche il numero *ottimale* delle funzioni hash:

$$k\_ottimo = \frac{m\_ottimo}{n} \cdot ln(2) \tag{3}$$

UNIVERSITÀ DEGLI STUDI FIRENZE

Descrizione dell'Algoritmo

- Il seguente programma è stato implementato in ***C++***, sfruttando le direttive di ***OpenMP*** per definire la sua versione parallela.
- La classe principale con le sue funzioni corrispondenti è stata definita in modo tale da poter inserire qualsiasi *Tipo* di insieme di dati, in modo da rendere il programma molto flessibile in termini di necessità.
- Al suo interno l'insieme di dati studiato viene rappresentato tramite una *deque*, ossia una struttura dati dinamica che utilizza le funzionalità di una coda e può essere usata anche come array.

UNIVERSITÀ DEGLI STUDI FIRENZE

Costruttore di entrambi i programmi

```
this->my_set = my_set;
this->n = this->my_set.size();
this->m = abs((n*log(0.01))) / (log(2)*2);
this->k = (m/n)*log(2);
initializeBloomFilter();
add(my_set);
```

Inizializzazione

Versione ***Sequenziale***:

```
template <typename T>
void BloomFilter<T>::initializeBloomFilter(){
    bloom_filter = new bool[m];
    for(size_t i=0; i<m; i++){
        bloom_filter[i] = false;
    }

}
```

Inizializzazione

Versione ***Parallela***:

```
template <typename T>
void BloomFilter_parallel<T>::initializeBloomFilter(){
    bloom_filter = new bool[m];

    #pragma omp parallel for num_threads(omp_get_max_threads())
    for(size_t i=0; i<m; i++){
        bloom_filter[i] = false;
    }

}
```

Aggiunta

Versione ***Sequenziale***:

```
template <typename T>
void BloomFilter<T>::add(std::deque<T> my_set){
    std::hash<T> h;
    while(!my_set.empty()){
        for(size_t j=0; j<k; j++)
        {
            size_t index = ((h(my_set.front())*(j+1))+n)%m;
            if(!bloom_filter[index]) bloom_filter[index] = true;
        }
        my_set.pop_front();
    }

}
```

Aggiunta

Versione ***Parallela***:

```
template <typename T>
void BloomFilter_parallel<T>::add(std::deque<T> my_set){

    #pragma omp parallel for num_threads(omp_get_max_threads())
    for(size_t i=0; i<my_set.size(); i++)
    {
        insert_element(my_set[i]);
    }
}
```

Controllo

Versione ***Sequenziale***:

```
template <typename T>
bool* BloomFilter<T>::probably_contains(std::deque<T> elements){

    bool *results = new bool[elements.size()];

    for(size_t i=0; i<elements.size(); i++){
        results[i] = check(elements[i]);
    }

    return results;
}
```

Controllo

Versione ***Parallela***:

```
template <typename T>
bool*
BloomFilter_parallel<T>::probably_contains(std::deque<T>elements){

    bool *results = new bool[elements.size()];

    #pragma omp parallel for num_threads(omp_get_max_threads())
    for(size_t i=0; i<elements.size(); i++){
        results[i] = check(elements[i]);
    }

    return results;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Risultati di Esecuzione

- Nella seguente sezione è possibile vedere i risultati di due test distinti: uno sui caratteri presenti in un testo di *1235195* caratteri di grandi dimensioni ed uno fatto su un insieme di interi variabile.
- Per entrambi si possono vedere i risultati del tempo di esecuzione dell'algoritmo *Sequenziale* e di quello *Parallelo*.
- Si può notare, nel secondo test, come lo **Speedup** incrementi al crescere degli insiemi di dati studiati.
- Architettura usata: *CPU Intel(R) Core(TM) i3-3217U, RAM 4 GB ddr3, Processori Logici: 4, Processori Fisici: 2*.

UNIVERSITÀ DEGLI STUDI FIRENZE

Esecuzione su Testo

Tempo Sequenziale	Tempo Parallelo	Speedup
0.421866	0.249991	1.687

Esecuzione su Interi

Valori	Tempo Seq.	Tempo Par.	Speedup
10^3	0.000999	0.000998	1.001
10^4	0.004001	0.003996	1.001
10^5	0.038977	0.028984	1.345
10^6	0.042374	0.026642	1.345
10^7	4.27939	2.55342	1.676

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni

Attraverso la versione implementata su OpenMP è possibile ottenere ampi margini di miglioramento, nelle prestazioni dell'algoritmo, quando si vogliono studiare enormi quantità di dati.
In questo modo, il programma che sfrutta la struttura dati del Bloom Filter, viene reso ancora più efficiente oltre che dal semplice controllo dei parametri di inizializzazione.