FILTRES DE BLOOM

**Introducció**

Com hem vist en els apartats anteriors, el diccionari es pot implementar amb taules de hash amb separate chaining i amb open addressing i ens hem fixat sobretot en el nombre d’accessos necessaris a diverses posicions de la taula tant per guardar com per veure si hi és present un element. De totes maneres, hi ha un altre aspecte que, tot i que avui en dia no és tan rellevant com ho havia estat, és important considerar per a certes aplicacions. Parlem de la memòria.  
 En les taules de hash, a grans trets, guardem el valor dels elements per després comprovar si hi és amb una comparació d’igualtat. Els filtres de Bloom, però, no fan això, sinó que utilitzar una funció de hashing per transformar el nostre element, que podríem dir-li la “clau”, en l’índex d’una posició en un array de bits (en la versió original dels filtres) incialitzats tots a 0. Així, posem a 1 el bit en qüestió i per comprovar si l’element hi és, només hem de repetir el procés i, en lloc de canviar el valor del bit, comprovar si està a 1, cosa que ens confirmaria que l’element està guardat al filtre. Però si només apliquem una sola funció de hash, és relativament fàcil que a dos elements els correspongui el mateix bit, cosa que causaria el que anomenarem “fals positiu”, és a dir, que el filtre de Bloom ens indicaria que un element que no hem afegit en cap moment hi és més present. A diferència de les taules de hash, doncs, pot donar resultats erronis, cosa que el restringeix a aplicacions on un “fals positiu” es pot corregir o no ocasiona problemes majors.

Per a disminuir la probabilitat de “falsos positius” apliquem k > 1 funcions de hashing que han de ser diferents, és a dir, que el seu resultat no es relacioni de cap manera amb les altres i una mesura que ens serà molt útil és el ratio de falsos positius, que es defineix com el quocient entre els falsos positius i el nombre total de negatius (amb falsos positius inclosos).

|  |
| --- |
|  |
| **Imatge 1.** Aquesta imatge ens mostra un exemple de filtre de Bloom, on tenim n = 3 elements que són afegits, k = 3 funcions de hashing que envien cada element a 3 bits i 1 element que mira si hi és present, però dona negatiu. De totes maneres, si hi hagués un element que les funcions portéssin el seus bits a les posicions quarta, cinquena i sisena, donaria positiu, tot i que cap dels 3 elements que hem afegit corresponen a tots aquests 3 bits, i, per tant, es donaria un fals positiu.  *(Imatge extreta de Wikipedia)* |

**Paràmetres òptims**

L’inconvenient dels filtres de Bloom és l’existència de falsos positius, per la qual cosa la optimització dels paràmetres d’aquesta estructura de dades anirà dirigida a reduir el més possible el ratio de falsos positius sense que acabi perdent l’aventage d’ocupar poca memòria o que tingui un cost computacional massa alt.

A l’article [Broder et al. 03], a l’apartat 2.2 *A Lower Bound* es demostra analíticament que la cota inferior de *m* bits necessaris és

*,*

sent *n* el nombre d’elements que hi volem poder guardar, ε la fracció màxima dels elements que pot representar el filtre de Bloom que poden donar fals positiu. Si fem aquest càlcul amb *n = 200* i *ε = 1%*, veiem que la *m* òptima (és a dir, la més petita possible perquè la probabilitat de fals positiu sigui igual o menor a ε) és *m = 1917*. Així doncs, veiem el gràfic següent:

|  |
| --- |
|  |
| **Gràfic 1 (***‘plot\_N.png’***).** Veiem el comportament del ratio de falsos positius al Bloom Filter quan varia la seva mida. (Cal dir que la *p* del títol és ε)  S’ha generat executant “*./main N* “ i al GNUPLOT “ *load 'gnuplot\_commands\_N.plt'*  ” |

Observem que quan arriva a un lloc proper a la m òptima, el *RFP* està molt proper a 0 i si anem al fitxer de dades (‘*plot\_F.dat*’), podem veure com efectivament el *RFP* és aproximadament 1%.



Ara ens falta saber quantes funcions de hashing *k* són les adients (és a dir, les menys possibles perquè la probabilitat de fals positiu sigui igual o menor a ε). A l’article [Kirsch et al. 06], a l’apartat *2. Standard Bloom Filters* se’ns presenta la deducció del valor de *k* pel qual es minimitza la probabilitat d’un fals positiu:



Si fem el càlcul amb els valor d’abans de n i el valor òptim que hem trobat de m, veiem que *k = 6,64* , que per la restricció d’enteresa, agafem el valor enter per sobre (la funció sostre), així asegurem que tenim igual o menys posibilitat de fals positiu que la que ens interessa.

|  |
| --- |
|  |
| **Gràfic 2 (***‘plot\_F.png’***).** Veiem el comportament del ratio de falsos positius al Bloom Filter quan varia el nombre de funcions de hashing i, equivalentment, el nombre de bits que representen cada element. (Cal dir que la *p* del títol és ε)  S’ha generat executant “*./main F* “ i al GNUPLOT “ *load 'gnuplot\_commands\_F.plt'*  ” |

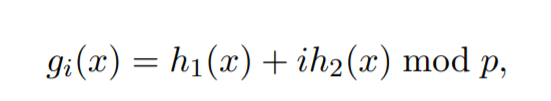
De la mateixa manera que el Gràfic 1, en el Gràfic 2 també sembla que dona el resultat correcte, i si mirem el fitxer de dades (‘*plot\_F.dat*’), veiem que, efectivament dona un valor molt proper a l’1% de *RFP*.



Tot i això, per ser del tot correcte no hauria de donar més de l’1%, però amb quasi tota seguretat en un altre generació de dades seria inferior o si augmentéssim el nombre d’iteracions del programa s’estabilitzaria cap a <1%.

**Funcions de hashing**

Per a implementar el filtre de Bloom necessitem *k* funcions de hashing diferents, cosa que a primera vista sembla una cosa difícil de trobar per nombres molt grans i molt costosa a l’hora d’executar. Una solució, proposada al paper *Less Hashing, Same Performance: Building a Better Bloom Filter*, és la Optimització de Kirsch-Mitzenmacher, que, a grans trets, per tot i amb *0 <= i < k* ens permet obtenir les *k* funcions de hashing diferents, *gi(x)*, que volíem avaluant només dues funcions de hashing diferents, *h1(x)* i *h2(x)*, tal i com mostra la següent fórmula:



*(Aquí no entrarem en la demostració d’aquesta proposició, però està tot recollit a l’article mencionat anteriorment i present a la bibliografia.)*

Aleshores, hem de trobar dos funcions de hashing per a generar les *k* que són necessàries. Així doncs, s’han de triar funcions de hashing molt robustes, que assegurin la uniformitat del hashing per tal de limitar el màxim possible les col·lisions. La cerca per la xarxa va portar-nos a escollir:

* El MurmurHash3, el més recent de la família dels MurmurHash, i en particular la versió de 32 bits, doncs no necessitàvem tant com la versió de 128 bits, que a més té diferentes implementacions depenent de l’arquitectura del processador de l’ordinador que l’executa. Aquest algorisme destaca per ambdues altes uniformitat i velocitat, encara que no és criptogràfic, és a dir, amb relativament poc esforç es pot deduir l’entrada a partir de la sortida si es tenen prous dades, però això no ens afecta. Necessita una *seed* que normalment es generaria (pseudo)aleatòriament per cada instància de Bloom Filter. De totes maneres, hem decidit fixar la *seed* a *26* per a facilitar tot el possible la reproductibilitat dels resultats (tot i que reconeixem que hi ha prous elements (pseudo)aleatoris com perquè això sigui pràcticament impossible). Cal comentar, també, que al fer mòdul d’un nombre no que no sigui potència de 2 fa que el valor resultat no sigui uniforme, tot i que la gran part del temps a la pràctica serà suficientment proper a uniforme perquè no afecti al funcionament del filtre de Bloom\*.

\*La informació està ampliada aquí:

<https://cs.stackexchange.com/questions/53072/understanding-murmur3>

* El SHA-256, de la família SHA, és un algorisme criptogràfic molt famós que encara no està compromès i és computat amb paraules de 32 bits. L’únic problema que presenta és que retorna strings, i per tant s’ha hagut de fer una funció que les converteixi en enters.

*(El codi dels algorismes de hash mencionats ha estat extret de fonts externes, per això estan a la carpeta ‘hash\_functions’ amb les seves llicències corresponents)*

**Counting Bloom Filters**

Amb el que s’ha presentat fins ara, podem afegir i mirar si un element és al filtre, però què passa si volem eliminar un element del filtre de Bloom? La resposta intuïtiva seria fer el mateix procediment que fem per afegir o mirar si hi és i posar a 0 els bits corresponents, però hi ha una probabilitat molt significativa que alguns dels bits que representa aquest element també s’utilitzen per a representar-ne d’altres. Això provoca que eliminant un element potencialment n’eliminéssim també alguns altres. La solució és implementar el filtre de Bloom amb enters enlloc de bits, de manera que quan s’afegeixi un element es sumi 1 a l’enter de les posicions que el representen i per eliminar només caldria restar 1 al mateix.

|  |
| --- |
|  |
| **Imatge 2.** Aquesta imatge ens mostra un exemple de CBF, on tenim n = 4 elements que són afegits i k = 3 funcions de hashing que envien cada element a 3 paraules de varis bits. Fixem-nos com els enters compten quantes vegades una funció de hash envia un element cap a ell.  *(Imatge extreta del blog de FlinkIt, link:* [*http://blog.flink.it/2010/05/flow-analysis-time-based-bloom-filters.html*](http://blog.flink.it/2010/05/flow-analysis-time-based-bloom-filters.html)*)* |

Tal i com podríem imaginar, com mostra el Gràfic 3, el fet de posar comptadorsno afecta a la probabilitat de fals positiu, però òbviament sí que ocupa més memòria, ja que substituïm un bit per un enter. De totes maneres segons l’anàlisi de [Fan et al. 00] amb 4 bits per comptador n’hi hauria prou per la majoria d’aplicacions, per la qual cosa es quadruplica el cost en memòria, a canvi de poder utilitzar els filtres de Bloom en aplicacions que requereixin d’eliminacions.

|  |
| --- |
|  |
| **Gràfic 3 (***‘plot\_C.png’***).** Primer cal fixar-se en l’escala de l’eix y. A partir d’aquí, al gràfic es veu com els RFP dels dos filtres de Bloom van al voltant de l’1% i pugen i baixen aleatòriament.  S’ha generat executant “*./main C* “ i al GNUPLOT “ *load 'gnuplot\_commands\_C.plt'*  ” |

**Complexitat de les operacions**

Les operacions en sí, tant d’afegir un element, com mirar si hi és, com eliminar-lo són lineals respecte *k* si no comptem el cost de evaluar la funció de hash, que és on recau pràcticament tot el cost computacional i més amb les funcions que hem escollit que, tot i ser ràpides per assegurar la uniformitat de les sortides. Així doncs el cost de les operacions seria el més gran entre *Θ(k)*, la complexitat del MurmurHash3 i la complexitat del SHA-256. Cal afegir, que a diferència de les taules de hash, el load factor o com de carregat està el filtre no afecte en el cost d’afegir o comprovar si hi ha elements, si no que afecta en la quantitat de falsos positius que passaran quan comprovem la presència d’un element.

**Implementació**

El programa es basa en 3 parts, que a la vegada estan dividits en 3 arxius diferents:

* ‘*creaArixus.cc*’: Correspon a la part de creació dels conjunts de dades i és pràcticament igual que els que s’han fet servir a la resta del projecte, amb la diferència que està implementat en forma de classe estàtica (almenys tots els seu mètodes són estàtics) per a poder cridar-ho més còmodement des d’un altre fitxer. A grans tres, genera l’*arxiu1* amb *n* nombres enters aleatoris no repetits i l’*arxiu2* amb *2n* nombres enters aleatoris no repetits més entre 0 i *n* nombres del primer arxiu, que de mitja serà *n/2* per com esta implementat. A més, a diferència de la resta del projecte, retorna una parella d’enters que es corresponen al nombre de nombres de l’arxiu1 a l’arxiu2 i el nombre total de nombres a l’arxiu2.
* ‘*BloomFilter.cc*’: Correspon a la implementació de l’estructura de dades del filtre de Bloom en forma de classe, pel que també hi ha els mètodes que calen que s’executin sobre aquest. Per a no repetir codi innecessàriament a l’hora d’implementar el Counting Bloom Filter, s’ha fet una superclasse que comprèn tots dos BloomFilters, amb les funcions de hash i passar a enter sense signe un cadena de caràcters que treu l’algorisme de SHA-256 (aquests dos estan explicats amb un mica més de profunditat al codi) i la declaració de les operacions bàsiques que comparteixen els dos tipus de filtres de Bloom. La única funció diferent és la de “*del*” que elimina elements en el CBF. Les operacions bàsiques no estan comentades pel fet de que el codi és prou evident. Només cal comentar, que per a cada operació només s’executen les funcions de hash una vegada cada una, doncs la funció “*hash*” retorna un vector amb els *k* valors calculats.
* ‘main.cc’: Tal i com el nom indica, conté la “*main”* que s’encarrega d’acceptar paràmetres de l’usuaris i sinó posar-los ell per defecte, calcular els valors dels paràmetres òptims i treure’ls per pantalla i iterar sobre la funció “*graficar*” que genera els fitxers que després és plottejaràn amb el GNUPLOT. S’itera canviant alguns valors segons el que vulgui l’usuari. També va informant per pantalla del progrés a l’usuari.

Per últim, també hi ha els ‘*gnuplot\_commands\_X.plt*’, sent X, o bé F o bé C o bé N.

Serveixen perquè els gràfics quedin exactament com estan adjuntats aquí.

Al ‘*README*’ hi ha les instruccions necessàries per a compilar el programa, executar-lo amb diversos paràmetres si es vol (tot i que depenent del primer paràmetre es poden ignorar alguns d’aquests) i finalment generar els gràfics.

**Comentaris respecte Bloom Filters**

És important subratllar que el ratio de falsos positius (RFP) és el quocient entre els falsos positius i el nombre total de negatius. Vaig gastar molt el temps veient en què fallava la meva implementació perquè no em donava resultats ajustats als càlculs teòrics i era perquè calculava aquest ratio com el quocient entre els falsos positius i el nombre total de positius. Llavors diria que he après que s’han de tenir clares totes les definicions, en especial les d’una mesura tan important com aquesta, que és equivalent a la probabilitat de que passi un fals positiu.

Fora d’això, ha estat interessant investigar una estructura de dades tan peculiar, que avui en dia s’utilitza en coses molt concretes i la documentació mitjanament escassa, tot i que hi ha un parell de papers (estan a la bibliografia) que han resultat molt enriquidors, i tot i que no s’ha posat a la documentació cada explicació en detall, qualsevol que llegeixi el projecte pot anar-hi i veure que tot el que s’ha dit tenia base matemàtica estricta. En aquest aspecte, he intentat millorar i centrar-me en sintetitzar la informació de les diverses fonts, enlloc d’escriure el mateix amb diferents paraules.

Per últim, comentaré el meu aprenentatge en funcions de hash, que tot i que en el funcionament intern no he pogut aprofundir, sí que he hagut de buscar diverses funcions i determinar si depenent de les seves característiques (rapidesa, uniformitat, cost de memòria fins i tot) eren adients pel que es necessitava pel programa.

**Bibliografia/ Webgrafia**

[Fan et al. 00] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol.” IEEE/ACM Transactions on Networking 8:3 (2000), 281—293.

[Kirsch et al. 06] Kirsch, Adam & Mitzenmacher, Michael. (2006). Less Hashing, Same Performance: Building a Better Bloom Filter.. 4168. 456-467.

Pot trobar-se a <https://www.eecs.harvard.edu/~michaelm/postscripts/tr-02-05.pdf>

[Broder et al. 03] Broder, Andrei & Mitzenmacher, Michael. (2003). Survey: Network Applications of Bloom Filters: A Survey.. Internet Mathematics. 1. 10.1080/15427951.2004.10129096.

<http://www.zedwood.com/article/cpp-sha256-function>:

Pàgina d’on s’ha tret el codi per la hashing function SHA256.

<https://stackoverflow.com/>:

Fòrum on s’ha trobat informació vària molt valuosa, per guiar a altres aspectes per aprofundir.