Algorísmia

Mètodes de cerca en diccionaris

Carles Barreiro Gili
Daniel Roca Lopez
Ferran Coma Rosell

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Q1 2015–2016

Índex

1. Descripció de la pràcitca

La pràctica consisteix en l'anàlisi de diferents tipus d'estructures de dades per a funcionar com a diccionaris de cerca de paraules, en les que mitjançant una key (paraula) podem retornar un valor. Per fer això el que hem fet és utilitzar quatre estructures de dades i comparar els resultats amb diferents entrades. Les estructures de dades utilitzades són:

- taula ordenada i posterior cerca binària
- taules de hash
- filtres de bloom
- tries

Per a cada un dels algorismes hem executat un seguit de jocs de proves generats aleatòriament i hem comparat els resultats.

2. Algorismes

Cada algorisme el dividirem en dos apartats a evaluar, la inserció de les paraules i la comprovació de si una paraula es troba en el diccionari.

2.1 Taula ordenada + cerca binària

El mètode de cerca binària és molt bàsic: consisteix a mantenir una taula(vector) ordenada amb les N paraules del diccionari i realitzar una cerca dicotòmica per a cada paraula que vulguem cercar. El cost en memòria és **O(N)**.

2.1.1 Inserció

El procediment d'inserció amb la taula ordenada és trivial ja que sabem el nombre de paraules que contindrà el diccionari. Es llegeix un N primer, i posteriorment es llegeixen les N paraules i s'insereixen a un vector d'enters. Per últim, s'ordena el vector. Això es pot fer així amb cost **O(N logN)** ja que el nombre de paraules del diccionari és conegut d'avantmà. Altrament, caldria utilitzar algun algorisme que permetés mantenir la taula ordenada a mesura que es van inserint paraules al diccionari.

2.1.2 Comprovació

Quant a la comprovació de l'existència de les paraules del text en el diccionari, simplement cal aplicar la cerca dicotòmica ja coneguda, que ens permet comprovar l'existència de cada paraula gràcies a tenir el diccionari ordenat amb cost **O(log N)**, on N és el nombre de paraules del diccionari. Per tant, veiem que en aquest mètode, el tamany del diccionari influeix a l'hora de determinar el cost de cercar una paraula.

2.2 Taula de Hash

En aquest algorisme ens basem en una taula de tamany igual al doble del nombre d'entrades que tindrem, aleshores mitjançant una funció de hash (mòdul N) on N és el tamany del diccionari, col·loquem la paraula a la posició que mapeja la funció de hash. En cas de col·lisió en una cel·la de la taula, hem decidit resoldre-ho amb llistes encadenades, és a dir, enllistar les paraules que mapejen a la mateixa posició de la taula de hash. La raó d'utilitzar la funció de hash mòdul, tot i ser bastant fàcil aconseguir una entrada que la torni molt ineficient, per a números random funciona força bé i és especialment ràpida.

En la primera versió de la taula que hem programat, el tamany era igual al nombre d'entrades que tindrem però això provocava que el temps en fer les comprovacions fos considerablement més gran i a més, el número de comprovacions que feia de mitjana per trobar si un element hi era a la taula, era aproximadament del doble. Per tant, hem decidit que el tamany òptim o adequat era el doble del número de claus del diccionari. A més, tenim en consideració el factor de càrrega, és a dir, com de plena està la taula de hash, per tal que si el factor és molt alt, fem un resize de la taula i tornem a inserir els elements. Així, en el cas que no sapiguessim el número d'elements que ens entraran, també funcionaria.

2.2.1 Inserció

Per a la inserció simplement agafem la key que volem col·locar, hi apliquem la funció de hash i obtenim la posició en què l'hem de col·locar i l'afegim al final de la llista. Cost assimptòtic constant en cas pitjor **O(1)**.

2.2.2 Comprovació

Per a la comprovació seguim el mateix procediment que per a la inserció, escollim la key que volem buscar, li apliquem la funció de hash i accedim a la posició de la taula que ens ha retornat. Un cop en aquella posició recorrem la llista fins a trobar el valor, o en cas contrari

dir que no hi és. Cost assimptòtic esperat : **O(N/m)**. Cost assimptòtic en cas pitjor: **O(N)**. On N és el tamany del diccionari i m el tamany de la taula de hash.

2.3 Filtre de Bloom

El filtre de bloom consisteix en una taula de bits en la que anem marcant a cert els que ajuden a identificar un dels elements que hem inserit. És a dir, per inserir el número 30, segons els paràmetres òptims que haguem calculat, suposant que haguem de fer 10 funcions de hash amb el 30, calcularíem el valor resultant de cada funció amb el número 30 i aquestes 10 caselles les posaríem a 1. D'aquesta manera, si posteriorment volem comprovar si el 1 hi és, farem el mateix procés sense modificar la taula. En el cas que totes les caselles que ens donen les funcions de hash estiguin a 1, indicarà que és possible que el número que consultem hi sigui. Diem que és possible ja que el filtre de bloom té una probabilitat de donar falsos positius.

En la primera versió del filtre que hem programat, el tamany de la taula de bits i el número de cops que feiem hash per cada inserció/comprovació ho calculavem mitjançant fórmules que hem trobat als apunts i a internet, per calcular els paràmetres òptims. Posteriorment, hem fet proves reduïnt el número de hashs que fem, per tal d'estalviar temps en cada inserció i comprovació, i hem observat que el número de falsos positius es manté. Per tant, hem decidit que el número òptim o adequat de cops que fem hash és una mica menor al que ens calcula la fórmula. En funció del valor que ens surt, restem 2, 3 o 4 al número de hash que fem.

2.3.1 Inserció

Per a la inserció simplement agafem la key que volem col·locar, hi apliquem el número de funcions de hash que hem decidit com a òptimes i obtenim les posicions en què hem de col·locar un 1 a la taula en cas que no hi sigui. Cost assimptòtic constant en cas pitjor **O(1)**.

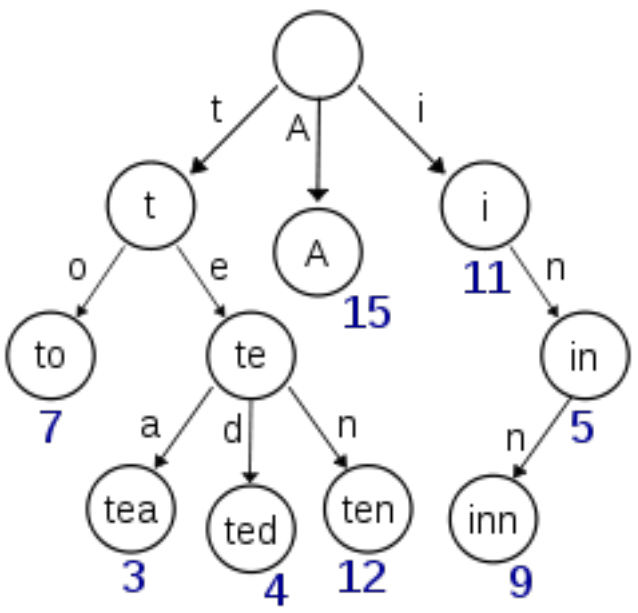
2.3.2 Comprovació

Per a la comprovació seguim el mateix procediment que per a la inserció, escollim la key que volem buscar, li apliquem les funcions de hash i comprovem les posicions de la taula que ens han tornat les funcions. Com hem dit abans, això pot donar un fals positiu, és a dir, podria ser

que coincideixi que un altre número que si que hi és, es codifiqui amb les mateixes posicions de la taula.

2.4 Trie

Un trie, o arbre de prefixes, és una estructura de dades que s'utilitza per implementar arrays associatius, es basa en un arbre com el de la imatge



on (de manera molt simplificada) cada clau s'emmagatzema a partir dels seus prefixes mitjançant fills en l'arbre. Només es construeixen els prefixes de les paraules que volem emmagatzemar, que en el cas de l'exemple són: to, tea, ted, ten, a, i, in, inn. En cada node, a més, es guarda un valor que ens indica si és final de paraula o no. Això ens serveix per identificar que el node "in" també ha estat emmagatzemat, i no confondre'l amb un prefixe del node "inn" i així evitar falsos positius. El tamany d'un trie pot ser un inconvenient si no tenim molts prefixos similars, ja que aleshores haurem d'emmagatzemar de l'ordre de **N*k** nodes diferents. En el nostre cas, com que l'alfabet son els dígits de 0 a 9, és un alfabet relativament petit i és molt més probable que hi hagi prefixes comuns. Per contra, en un alfabet de caràcters tenim de l'ordre de 26 caràcters, que fa que la probabilitat de prefixos comuns sigui més alta.

2.4.1 Inserció

Per a inserir una paraula s[0..k-1] en el trie, simplement agafem cada lletra, o en el nostre cas dígit, en ordre i creem un fill per a cadascun(si no hi és ja a l'arbre). Per últim, marquem

l'últim node k-1 com a final de paraula. Això té un cost en el cas pitjor del tamany de la paraula que inserim **O(k)**, i no depèn per res del tamany del diccionari.

2.4.2 Comprovació

El trie ens permet cercar paraules també en cost lineal respecte del tamany de la paraula que cerquem **O(k)**. La cerca consisteix en un recorregut per l'arbre de prefixes comprovant si existeix un recorregut en l'arbre tal que cada node és una lletra de la paraula i l'últim node té el booleà de final de paraula a cert.

3. Tests

Per tal de demostrar i comprovar quin és l'algorisme més eficient donant-li aquest ús, hem fet un seguit d'experiments. Els experiments consisteixen a inserir en una estructura de dades (propia de cada algorisme), un seguit de claus que anomenem diccionari. Posteriorment, agafem un seguit de valors (sempre més valors que claus en els nostres experiments), i comprovem un a un si hi són en el nostre diccionari. Per comparar els algorismes hem mesurat per separat el temps d'inserir i el de consultar. A més, en els temps no hem tingut en compte el procés de llegir de fitxer les dades, és a dir, primer les llegim, les fiquem a un vector i posteriorment comencem a calcular el temps de processat de les dades (inserir i consultar).

A l'hora de crear les gràfiques hem fet una comparació de tots 4 algorismes: Hash, Bloom, Trie i Binary Search. Podem trobar algunes gràfiques comparatives a l'**Anex**: Comparació del temps d'inserció i de consulta dels 4 algorismes esmentats.

Després de realitzar experiments comprovant els temps hem decidit comparar també el nombre de comprovacions que fa cada algorisme, és a dir, per exemple l'algorisme de hash té un cost estimat **(1)** per consultar un element però, en realitat això depen de l'experiment i de cada moment, per tant, cada cop que consultavem un element, sumavem 1 a cada comprovació que feia, en el cas del Hash, si no ho troba a la primera posició de la llista corresponent a la posició de la taula que li ha donat la funció de hash, recorrerà la llista fins a trobar l'element i anirà sumant 1 a cada consulta.

3.1 Resum Tests

En aquest apartat resumirem els test que hem realitzat. En primer lloc, com hem comentat, a l'Anex trobarem gràfiques comparatives del temps d'inserció i consulta dels algorismes escollits. A més, també podeu trobar totes les dades dels experiments a la carpeta **outputs** del projecte.

Per fer els tests hem generat diferents fitxers, que podreu trobar en la carpeta de **jocdeprobes**, aquests van de més simple a més complexe, en l'anex trobarem les dades que hi ha a cada test i també podem trobar-ho dins la carpeta **jocdeprobes** en l'arxiu **Log_tests**.

Un cop executat tots els testos, els hem passat a una taula per poder fer-ne un estudi.

Amb els resultats extrets podem dir que l'algorisme més eficient per a implementar un diccionari d'aquest tipus són les taules de hash, ja que efectivament donen un cost constant tant en la inserció com en la consulta, no queda molt lluny l'algorisme de cerca binaria, que en cas de la inserció és pràcticament idèntic, amb un petit increment a l'hora de realitzar les comprovacions, per això amb els nostres tests podríem deduir que l'algorisme de cerca binaria es es segon millor dels quatre implementats.

L'algorisme de cerca binaria, ens dona uns resultats força correctes, i amb un volum de dades força elevat, però matemàticament, podem dir que acavaria tenint un cost superior al filtre de bloom.

Tot seguit, els filtres de bloom que a nivell d'eficiencia també ens genera un cost constant en la inserció i en la consulta, té diversos inconvenients. En primer lloc el filtre de bloom no ens permet recuperar un valor, és a dir, simplement ens diu si trobem l'element en el diccionari o no, però si aquest tingues un valor, no el podríem extreure. I també trobem el problema dels falsos positius. Encara que després de buscar la manera de reduir al màxim aquest problema, és impossible dir amb certesa que el resultat es correcte.

Al filtre de bloom el segueix el trie, aquest algorisme funciona segons el temany de l'entrada, i ens dona un cost superior a tots els altres algoritmes, tot i així, treballan amb números força grans, el seu temps no es dispara, i continua constant segons l'entrada.

També hem volgut realitzar un estudi sobre les comprovacions que fa cada un dels algorismes, pot sobtar que el que més comprovacions fagi és el de cerca binaria i que a l'hora tingui un temps d'execució millor que filtres de bloom i trie. Això és degut a que encara que fa moltes comprovacions, són directes, es a dir, és una sola operació mentres que els altres, fan més operacions per comprovació.

En primer lloc el que menys comprovacions fa es la taula de hash, axiò es coherent, ja que en principi l'accés a un valor es directe, o com a molt, ha de recorrer una petita llista, com podem veure le nombre de comprovacions no s'allunya molt del nombre d'entrada. El seguiex en número de comprovacions el trie, no molt lluny del filtre de bloom, aquest cost

dependra de la profunditat de l'arbre, que aquesta depen de l'abecedari que utilitzem i de quans prefixos communs hi ha.

El filtre de bloom, te un comportament semblant al de la taula de hash, ja que realitza un seguit de operacions i va directe al valor, però utilitza més funcions de hash, per tant, si ens hi fixem, el número de comprovacions que realitza el filtre de bloom, es el mateix que la de les taules de hash, multiplcat per el número de funcions de hash que tenim en el filtre.

Finalment trobem les comprovacions que realitza la cerca binaria. Com ja s'ha comentat anteriorment, és un nombre molt elevat, però són comprovacions molt sencilles que fan que sigui un algorisme eficient, essent dons log(n) comprovacions per a cada entrada, en el cas mitja.

3.2 Conclusió

Per a implementar un diccionari tenim diferents funcions, cada una te unes qualitats diferents. La cerca binaria sembla un bon sistema de diccionari, ja que en eficiencia de temps és correcte (amb el volum de dades provat), i el temany per guardar dades és N, el millor junt amb el trie. Però sabem que si fem proves amb dades molt més gran tenim dos algorismes amb cost constant que l'acaben superan, que són el filtre de bloom i les taules de hash.

El filtre de bloom, no consumeix tanta memoria com les taules de hash, però tenim el problema dels falsos positius i que no podem recuperar un valor, el qual limita molt la seva utilitat com a diccionari, però té moltes altres aplicacions. La taula de hash, té un cost espaial de 2N, per poder mantenir uns costos temporals pràcticament constants, també és un molt bon algorisme ja que es comportara igual encara que augmentem el volum de dades, però si ens posem en el cas de tenir un enemic que ens entre les dades, podria fer que totes les keys vagin a la mateixa casella, és a dir, la taula de hash en el cas pitjor podria tenir cost N, i passar a ser el que te pitjors temps dels quatre algoritmes.

Finalment trobem el trie, que és una forma eficient en memoria de guardar un diccionari, ja que tenim un cost N però a nivell temporal és el pitjor de tots, i el seu comportament dependrà sempre de les entrades i de la longitud d'aquestes.

3.3 Test Aïllat

Test Aïllat per mesurar temps amb una prova considerablement gran:
500.000 claus al diccionari, 3.000.000 paraules al text.

```
-------------------------
Tamany Dicc: 500000
Tamany Text: 3000000
-------------------------
Test amb 500000 dades en el diccionari

Insercio de hash...
Temps d'insercio: 0,060956

Insercio de la taula de bits del filtre de bloom...
Numero de funcions de hash: 16
Tamany de taula de bits: 13656294
Temps d'insercio: 0,18626

Insercio del vector de cerca binaria...
Temps d'insercio: 0.041012

Insercio del trie...
Temps d'insercio: 0,197762
 ;-------------------------------------;
 ;          Prova de contains          ;
 ;-------------------------------------;
Comprovacio de hash...
----------------------------------------
Temps d'acces: 0,182263
Encerts: 590262
Fallats: 2409738
Comprovacions relitzades totals: 1339551
----------------------------------------
Comprovacio del filtre de bloom...
----------------------------------------
Temps d'acces: 0,643501
Encerts: 590267
Fallats: 2409733
Comprovacions relitzades totals: 48000000
----------------------------------------
```

Comprovacio de trie...

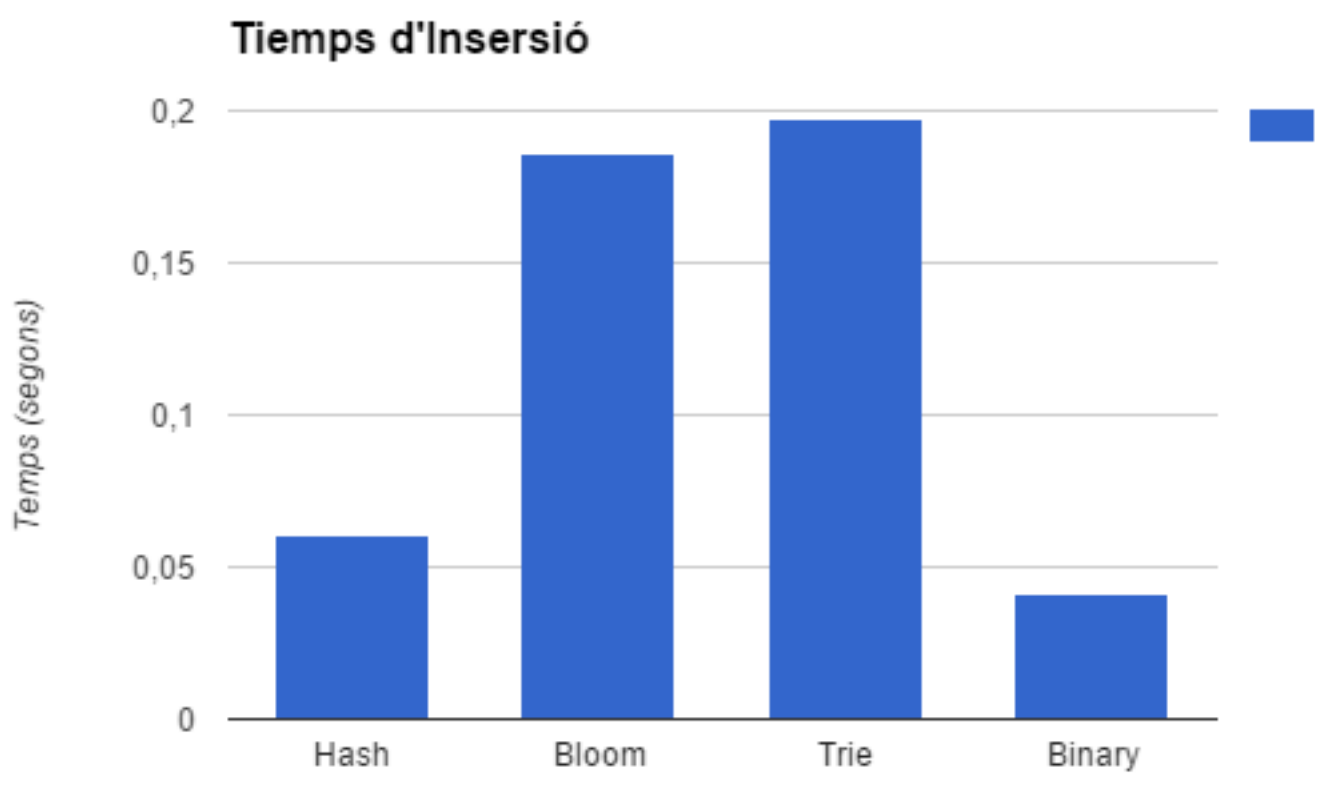
Temps d'acces: 1,17298
Encerts: 590262
Fallats: 2409738
Comprovacions relitzades totals: 1339551

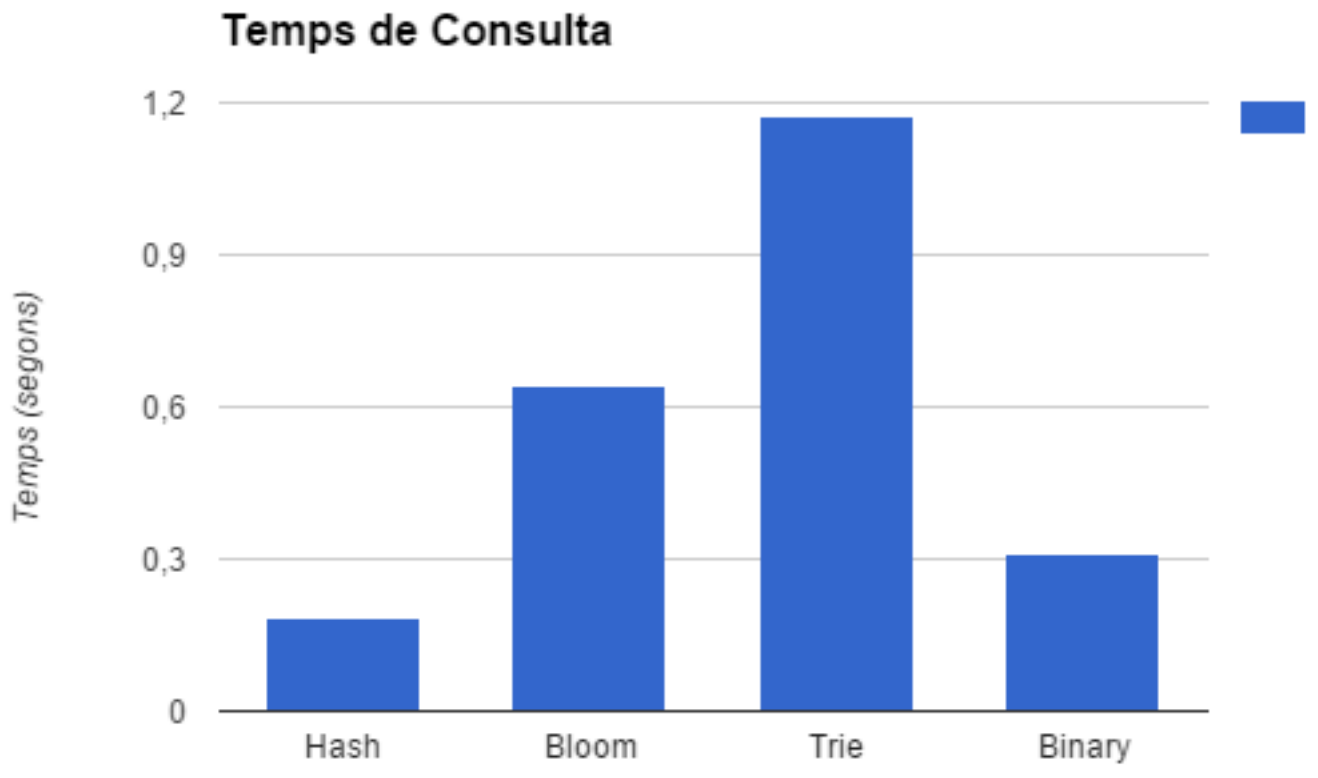
Comprovacio de cerca binaria...

Temps d'acces: 0,309786
Encerts: 590262
Fallats: 2409738
Comprovacions relitzades totals: 58597246

Com podem observar aquí, el filtre de bloom ens ha donat 5 fals positius. Ho sabem ja que podem comparar quants encerts i fallats hauria d'haver donat, amb els altres algorismes.

Per comparar els temps d'inserció i de consulta podem veure les següents gràfiques:





Com observem a les gràfiques, a més de ser el 2n més lent el filtre de Bloom, ens pot donar falsos positius, la qual cosa el converteix en una mala opció per a utilitzar en aquest àmbit.

4. Entrega

En l'entrega de la pràctica podreu trobar la carpeta **projecte**. Aquesta conté en carpetes separades els quatre algorismes implementats, cada un amb una classe separada. Per adjuntar-los, hem generat un mainGeneral, aquest el que fa es crear una estructura de dades de cada tipus de diccionari i llegir els jocs de proba. Com ja s'ha comentat anteriorment, el main primer llegeix el fitxer d'entrada, guarda les dades en un vector i aleshores comencem a fer els incerts a cada estructura de dades, una a una, per poder calcular el temps exacte i exclusiu de cada un. Un cop realitzat això fem el mateix amb les comprovacions. Un cop acabat el test, es genera un fitxer de sortida a la carpeta output, amb els resultats, i comença un nou test.

4.1 Com executar el main

Hem generat un Makefile que complica tot el projecte, per a l'execució del projecte simplement hem d'executar la comanda Make i tot seguit la següent linea de codi:

./test1 < cinpruebas

Això executarà el nostre projecte amb els fitxers de prova que es trobin a dins de la carpeta **jocdeprobes**, aquests fitxers han de tenir el següent format:

- **dicX.txt**
- **textX.txt**

on X és el número de test que realitzem.

En el fitxer **cinpruebas** hem d'introduir el número de tests a realitzar, seguit dels tests que volem fer, és a dir, si volem realitzar 5 tests, i aquests són el test 2,7,9,21,22, el que farem serà escriure en el fitxer **cinpruebas**

- **5 2 7 9 21 22**

Amb això al executar el test amb el fitxer cinpruebas com entrada, el main realitzarà un estudi dels testos 2,7,9,21,22, per a cada algorisme, i n'extraura els resultats en la carpeta output.

4.2 Generador de tests

Per a la realització de les proves hem creat un generador random de fitxers d'entrada, és a dir, ens crean els fitxers dicX.txt i textX.txt.

Per executar el generador simplement s'ha de compilar i executar:

g++ main.cpp

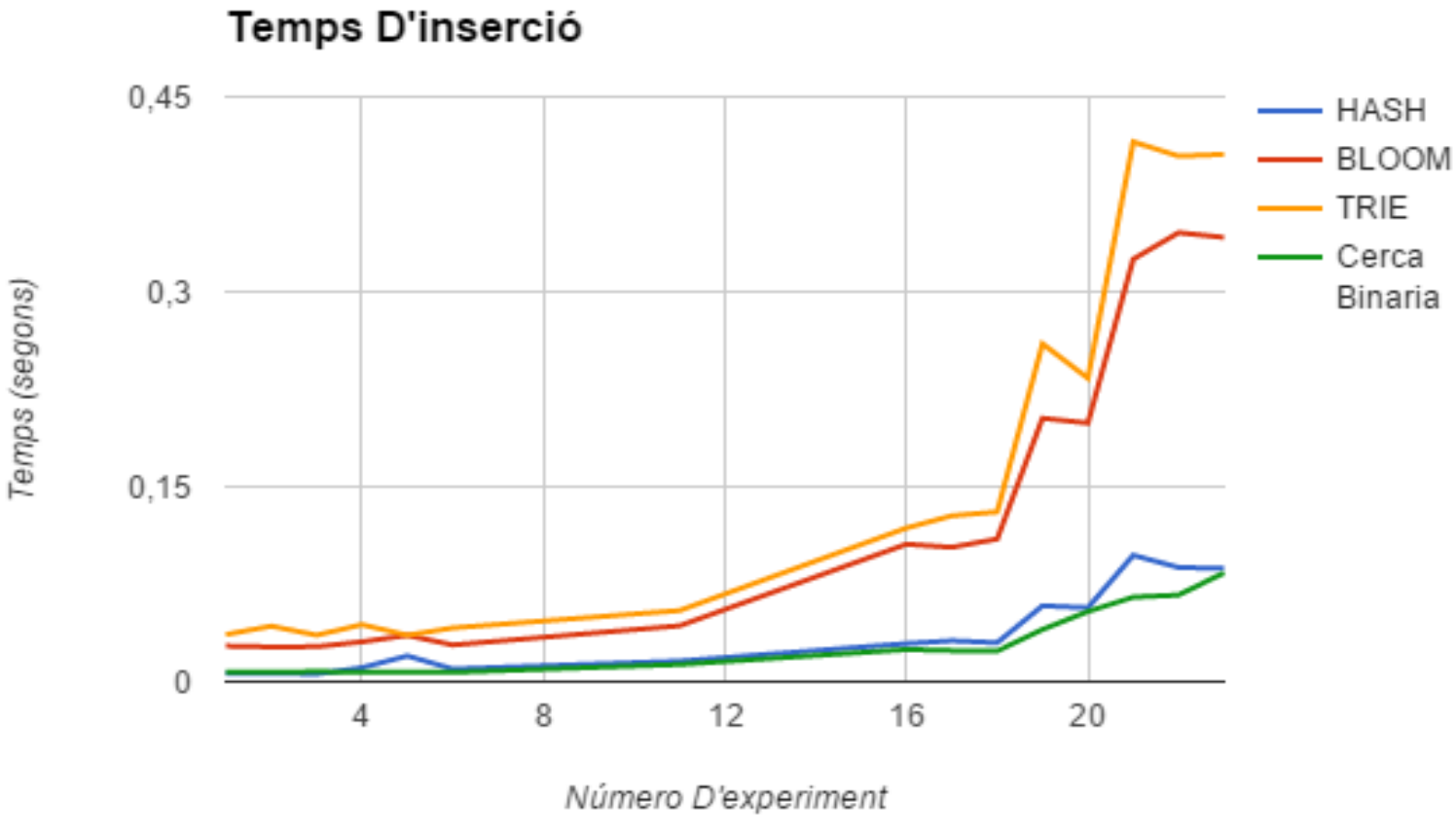
./a.out

El generador ens preguntarà quants tests volem crear, i a continuació per cada un dels tests, quantes entrades volem en el diccionari i a continuació quants texts volem consultar. Això es repetirá fins que hagem creat els N tests, informats al principi.

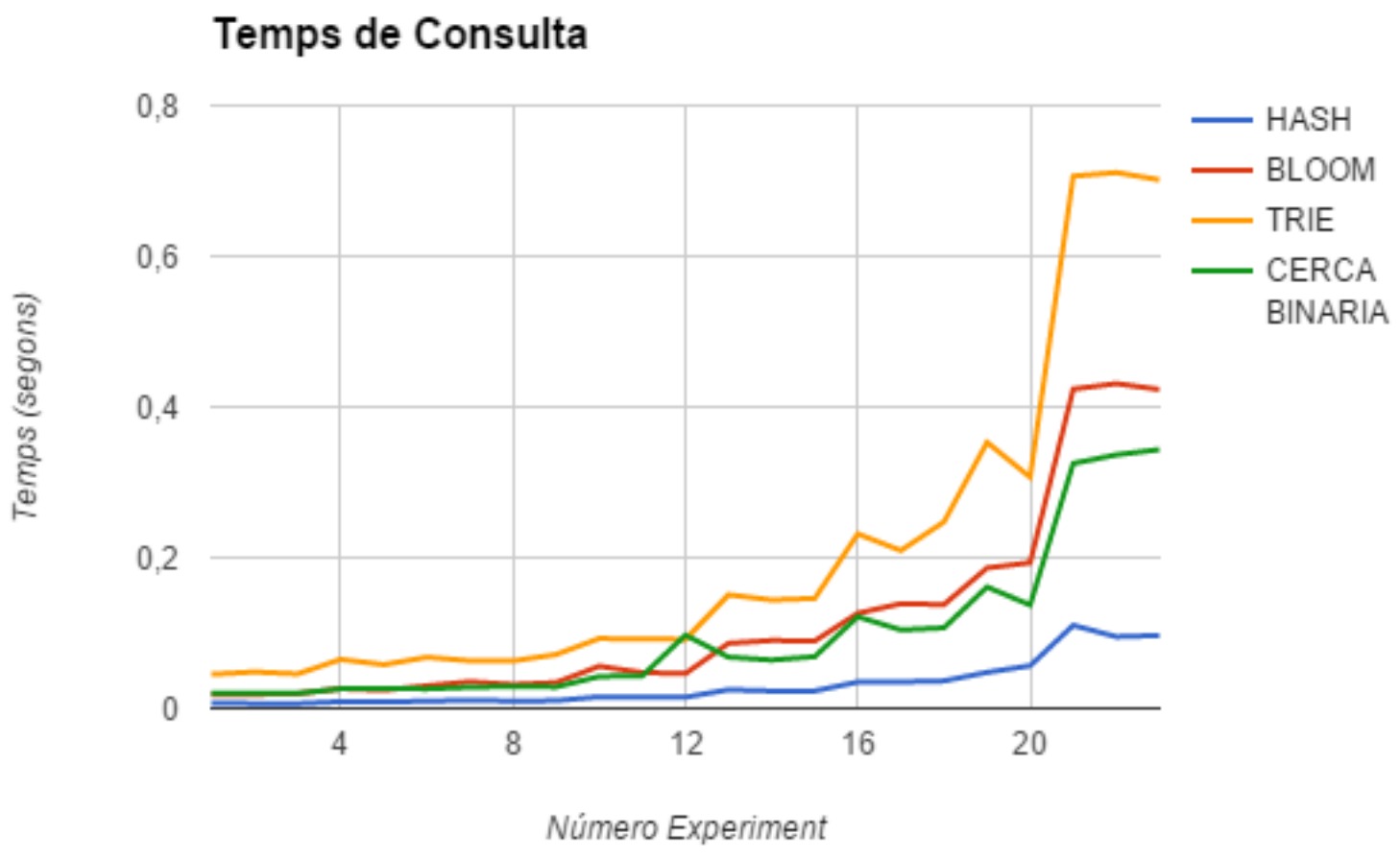
ANEX

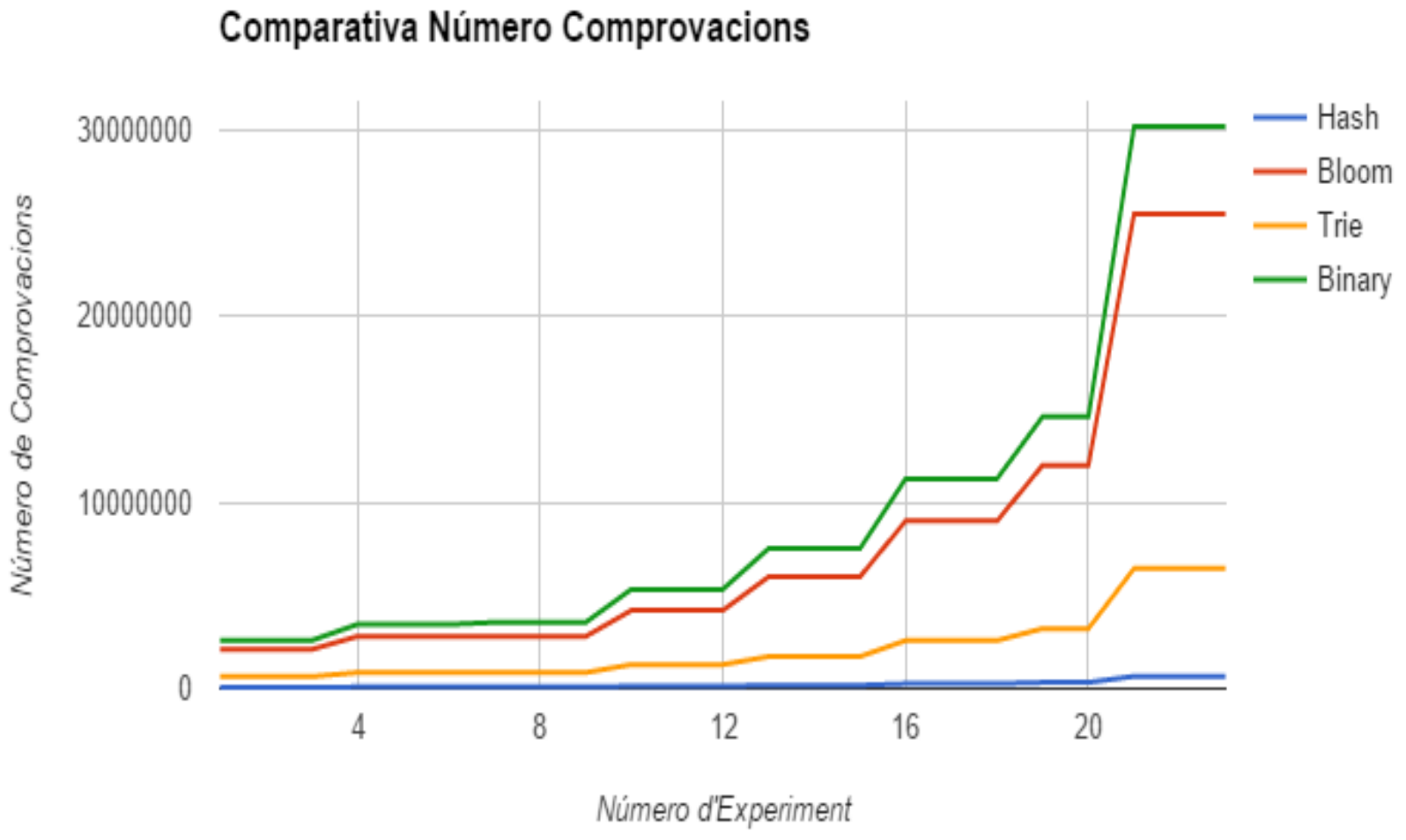
Gràfiques dels temps amb els 23 experiments

inserció:



cerca:




Comparativa Número Comprovacions
30000000
20000000
10000000
0
Número de Comprovacions
4
8
12
16
20
Número d'Experiment
Hash
Bloom
Trie
Binary


Datos para los Experimentos
2000
1500
1000
500
0
Número de Valors (x1000)
Quantitat de claus (x1000)
Quantitat de valors (x1000)
4
8
12
16
20
numTest