FACULTAT D'INFORMÀTICA DE BARCELONA

ALGORÍSMIA

Pràctica de *Hashing*

Josep Ciurana Herrera (josep.ciurana)
Oriol Closa Márquez (oriol.closa)

6 de maig de 2018

Sumari

1 Introducció

En aquesta pràctica comparem diversos algorismes i implementacions dels mateixos per tal de comparar i experimentar amb les diferències entre els temps dels diferents mètodes, així com els avantatges i els inconvenients de cadascun d'ells. Les tècniques a implementar i comparar són les següents.

- Cerca binària
- *Hash* amb taula
- Filtre de Bloom

Per tal de poder realitzar una comparació justa entre els tres, se'ns proposa un domini concret sobre el que treballar. Volem que aquests tres mètodes compleixin la funció d'un diccionari, el qual s'omplirà primerament amb un arxiu i posteriorment buscarà si conté o no les paraules d'un segon. Per poder-ho delimitar encara més, definim les paraules del llenguatge com a nombres enters no negatius.

En les pàgines següents es concretaran i explicaran els algorismes implementats per aquesta pràctica. Tot seguit s'exposaran els resultats obtinguts de l'experimentació per, finalment, desenvolupar les conclusions a les que hem arribat.

2 Cerca binària

Dels tres algorismes, aquest és l'únic que no era desconegut per a nosaltres, l'hem vist, analitzat i usat en moltes assignatures de la carrera. Tot i això, ens vam plantejar moltes implementacions diferents, el nostre objectiu era trobar la millor de totes. Finalment vam seleccionar dues opcions, condicionades per l'estructura de dades que allotja el diccionari[1], les quals s'exposen a continuació.

2.1 Cerca binària amb *array*

La primera implementació volíem que fos amb un vector ordenat ja que és l'exemple que s'utilitza per il·lustrar la cerca binària.

El programa obté un vector amb tots els elements del diccionari tal i com apareixen a l'entrada (les claus estan desordenades) i utilitza l'algorisme *merge sort* per ordenar-lo amb cost $O(n \log n)$, usem aquest i no el *quick sort* per assegurar-nos que en cap cas el cost augmentarà a $O(n^2)$.

Un cop finalitzada aquesta primera part va sorgir un problema; durant el *merge sort*, per a entrades significatives, la pila s'omplia. Així que vam haver de redissenyar parcialment l'algorisme i canviar l'estructura de dades a un *array* per treballar fàcilment sobre punters i així solucionar aquest inconvenient.

Finalment, l'algorisme de cerca binària implementat és l'estàndard amb cost $O(\log n)$ per element.

2.2 Cerca binària amb arbre AVL

En algun punt, també vam considerar implementar la mateixa estructura que usa std::map en C++, un arbre vermell-negre. Al final, per poder destinar més temps als nous algorismes vam acabar implementant un arbre AVL, el qual pertany a la mateixa família d'arbres binaris de cerca equilibrats, té costos asimptòtics iguals i ja l'hem estudiat amb anterioritat.

Així, vam crear la classe AVL amb mètode d'inserció de cost $O(\log n)$ per element, en total $O(n \log n)$, i mètode de cerca amb cost $O(\log n)$ per element.

3 *Hash* amb taula

D'acord amb l'article sobre tècniques de *hashing* [3], s'han elaborat diversos algorismes segons les tècniques d'*open hashing* i de *separate chaining*.

3.1 *Open hashing*

Segons les indicacions de l'article anteriorment mencionat, en aquesta secció es troben les tècniques més eficients. Quan hi ha una col·lisió, la funció de *probing* és cridada fins que s'ha trobat la clau que es cercava o s'ha trobat una posició buida. S'han implementat les següents funcions.

3.1.1 *Linear probing*

Partim d'una funció de *hash* h i d'un valor de passos i. Per tal de trobar la posició la qual pertoca la clau k, només cal aplicar la funció anterior $h(k)$. En cas de col·lisió, sumarem tantes vegades el nombre de passos com siguin necessàries per tal de trobar una posició lliure [6]. Així doncs, la funció resultant que haurem d'aplicar serà la següent.

$$h(k, i) = (h(k) + i) \mod m$$

Per posar d'exemple un cas específic, si i tingués com a valor 1, estaríem aplicant la funció de *hash* i, en cas de coincidir amb una posició ja ocupada, miraríem l'adjacent, i així fins que en trobéssim una de lliure.

3.1.2 *Quadratic probing*

En aquest cas, també tenim una funció de *hash* h i un valor de passos i, però afegim dos valors nous $c1$ i $c2$. Altra vegada afegirem posteriorment i tantes vegades com siguin necessàries per trobar un espai que no estigui ocupat. La funció a aplicar serà la següent.

$$h(k, j) = (h(k) + c_1 i + c_2 i^2) \mod m$$

3.1.3 *Double hashing*

Introduïm una segona funció de *hash* h_2 i reanomenem a la primera h_1. El que farem doncs és calcular el nombre de passos a partir d'una segona funció cosa que ens farà variar el valor segons la clau k que provem en cas de col·lisió.

$$h(k, j) = (h_1(k) + i h_2(k)) \mod m$$

3.1.4 *Cuckoo hashing*

Finalment trobem una versió una mica més complexa. Tot i que en l'article anteriorment mencionat utilitzaven una sola taula, nosaltres hem decidit implementar-ho amb dues[5]. Així doncs, tenim dues funcions h_1 i h_2. Procedim de la següent manera a l'hora de trobar la posició de k. Apliquem la funció h_1, en cas que la posició a la primera taula estigui lliure, ja haurem acabat. Si la posició no està lliure, substituïm la clau que anomenarem k_2 per k. Ara, aplicarem la funció h_2 a k_2 i la col·locarem a la segona taula. Si la posició resultant està buida, ja haurem acabat, en cas contrari, aplicarem la funció h_1 a l'element k_3 que hi havia on ara posarem k_2 a la segona taula. Repetim aquest procediment fins que haguem col·locat totes les claus.

Aquest procediment pot ser perillós, i és que mitjançant aquesta tècnica podem trobar bucles. Per exemple, comencem substituint una clau n per una altra m i després de fer l substitucions, ens trobem que tornem a substituir la clau n per m en la mateixa taula. Per detectar això, podem utilitzar un comptador d'iteracions màximes o vigilar expressament que no es repeteixi el primer cas, que és com nosaltres ho hem implementat. Una vegada passa això i per tal de poder afegir l'element, només podem fer un *rehash* amb dues funcions noves. En el nostre cas però, no afegim la clau i ho indiquem. Una possible manera d'evitar això últim, seria utilitzar x funcions de *hash* i no només dues.

3.2 *Separate chaining*

Per aquesta part hem implementat dues tècniques molt similars entre elles d'acord amb Grill[3].

3.2.1 *Move to front*

Utilitzarem una taula i una funció de *hash* h. Cada posició de la taula no contindrà només una clau, sinó que consistirà en una llista de claus. Cada vegada que volguem afegir-ne una, només caldrà aplicar la funció h i afegir-la a la llista.

Una de les parts interessants d'aquesta tècnica és que a l'hora de consultar una clau, aquesta es mou al davant de tot de la llista. És una bona manera d'accedir més ràpidament a claus que són utilitzades més freqüentment. Altres maneres serien mantenir la llista ordenada i inserir les claus mantenint aquest ordre, tot i que això només tindria sentit amb un nombre important de col·lisions i realitzant una cerca binària posteriorment, cosa que va en contra de la idea d'utilitzar funcions de *hash* amb taula.

3.2.2 *Exact fit*

Aquest cas és molt similar a l'anterior però no utilitza cap llista sinó un *array*. La idea és mantenir totes les claus amb col·lisió en una mateixa regió de la memòria. Així, quan s'hagi de recórrer aquest *array*, no anirem saltant per diferents posicions de memòria sinó que accedirem a la posició adjacent. Això ho podem assegurar donat que la funció per afegir un element a un vector en C++ ens assegura que si els vector creix més de la capacitat reservada, aquest es copia en una altra regió de la memòria mantenint tots els seus elements en aquesta mateixa regió, i a més, adjacents evitant l'accés aleatori i realitzant-lo de manera seqüencial.

4 Filtre de Bloom

Aquest algorisme utilitza un mètode molt enginyós i eficient a nivell espaial per indicar si donat un element qualsevol, aquest pertany a un conjunt fixat anteriorment o no.

La idea és la següent. Donat un conjunt d'n elements, generem un *array* d'm bits i k funcions de *hash* aleatòries per a una distribució homogènia amb $kn < m$, inicialment els m bits són a 0 i per cada element del conjunt col·loquem a 1 els bits del vector de les posicions que obtenim aplicant les k funcions de *hash* a l'element. D'aquesta manera, per saber si un element pot pertànyer al conjunt o no, només cal comprovar que els bits de les posicions obtingudes d'aplicar l'element a les funcions de *hash* siguin uns.

Amb aquest mètode però, apareix el concepte de fals positiu. Si algun dels bits és 0 podem saber segur que l'element no pertany al conjunt, ara bé, si tots són 1s pot ser que efectivament l'element pertanyi al conjunt o bé que un altre element, o parts d'altres elements, és *mapegin* igual que l'element que estem comprovant i per tant obtenim que l'element és al conjunt quan en realitat no hi és. La majoria d'aplicacions reals de filtres de Bloom tenen tolerància als falsos positius sempre que la proporció sigui la suficientment petita[4]. Aquest cas es veu clarament a la figura 1.

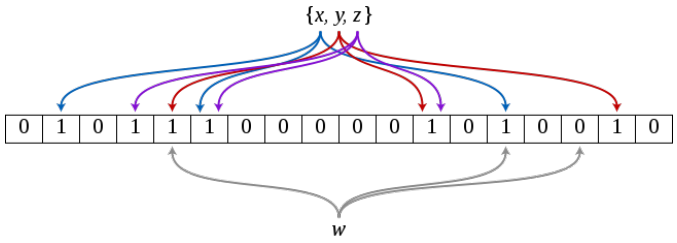

Figura 1: Exemple de filtre de Bloom amb el conjunt {x,y,z} per $k = 3$ i $m = 18$. Podríem tenir un fals positiu si preguntem per l'element v i aquest es *mapeja* en les posicions 1, 5 i 16.

I parlant d'implementacions reals, és una llàstima però per com hem fixat aquesta pràctica és difícil veure com d'útil pot arribar a ser aquesta tècnica que sobretot té una amplia utilització en xarxes, poder enviar informació el més petita possible, o en ordinadors antics on la memòria s'exhauria ràpidament i no podia ser menyspreada.

D'acord amb l'article sobre aplicacions de xarxa de filtres de Bloom[2], s'han elaborat un parell d'implementacions de l'algorisme amb diversos valors de k i m/n.

4.1 Primera versió

Amb $k = 4$, és important mantenir el nombre de funcions de *hash* reduït per evitar falsos positius, i $m/n = 16$, no vam voler excedir-nos en aquesta relació perquè aquesta versió usa un vector de *bools* en comptes d'un *array* de bits que és més fàcil d'implementar en C++ però realment per aquest llenguatge un *bool* té mida de *byte* per poder ser adreçat via *hardware* i encara que en certes circumstàncies es produeixen optimitzacions, en el nostre cas no en podem estar-ne segurs.

4.2 Segona versió

Amb $k = 4$ i $m/n = 28$, igual que un exemple que trobem a la documentació abans esmentada i que ens assegura una molt baixa proporció de falsos positius. Vam intentar utilitzar *setbit* per representar un *array*

de bits i posteriorment *setbit_dynamic* per poder fer-ho en temps d'execució, no ens en vam ensortir i finalment hem tornat a implementar un vector de *bools* suposant que un *bool* ocupa un bit.

5 Experiments

Per a realitzar els experiments s'ha utilitzat diversos jocs de proves que podeu consultar a l'annex B al final d'aquesta documentació. S'indicarà per a cada prova quin número s'ha utilitzat. Les taules detallades dels resultats obtinguts es poden trobar a l'annex A.

S'han realitzat experiments de comparació de salts (*miss* de la clau en la posició calculada), nombre de comparacions, temps d'inserció i temps de cerca, diferenciant també entre els temps dels elements trobats i els no trobats donat que ho hem cregut adient i interessant.

Així doncs, a continuació es detallen els experiments realitzats separats per categories. Una vegada realitzats, s'ha seleccionat el millor de cadascuna i s'han fet les comparacions adients entre categories. El temps es mostra en mil·lisegons en tots els gràfics.

Per tal de procedir amb l'experimentació hem establert un seguit de variables que no es canviaran durant el procés. Una d'elles és la mida de la taula, fixada en 1,5 vegades la mida del diccionari d'entrada, així com el valor de passos fixat en 13 després de realitzar el primer experiment i els valors de c_1 i c_2 fixats en 11 i 17.

5.1 Cerca binària

Una vegada implementats els algorismes vam procedir a fer la comparació entre els dos. Tot i que durant l'experimentació vam veure que s'havia de modificar per resultats erronis, els resultats que es presenten a continuació han estat realitzats amb els fitxers finals entregats juntament amb aquesta documentació.

S'han realitzat 5 execucions per tal de fer una mitjana del temps d'execució per a tots els jocs de prova, 1, 2, 3, 4 i 5. Veient la figura 1 podem determinar clarament que la primera versió, la que utilitza un *array* utilitza moltes menys comparacions que la versió d'arbre AVL. Quan l'entrada és molt petita, aquesta diferència és molt baixa, veiem els dos punts propers a 0 que corresponen a la primera entrada, i els dos punts corresponents a l'entrada 5 que ni tant sols hi caben al gràfic. Aquests valors es poden comprovar a la taula 1 i 2 de l'annex A. Pel que fa però a les comparacions de cerca, veiem que utilitzant un arbre AVL se'n fan moltes menys, de l'ordre de 10.000. Així doncs, si veiem la suma de comparacions totals que es fan, es veu clarament que la segona versió en fa menys de mitjana. El solapament que s'observa és degut a l'entrada més petita per la primera versió i la més gran per la segona.

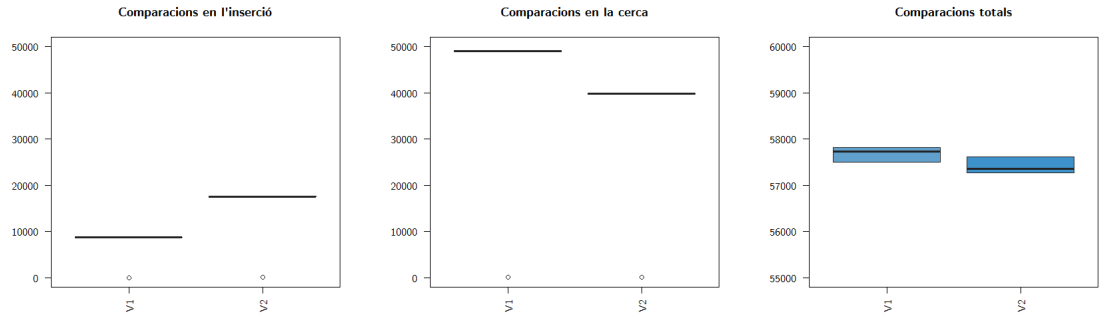

Figura 2: D'esquerra a dreta, la distribució del nombre de comparacions necessàries per inserir els elements, per cercar-los i el total.

Pel que fa als temps d'execució la cosa canvia. En ambdós casos, la versió amb AVL és té un cost superior a la de l'*array*, ho veiem en el temps total on el resultat de la segona duplica el de la primera. Tot i això, hem cregut convenient considerar el segon com una millor opció degut al menor nombre de comparacions.

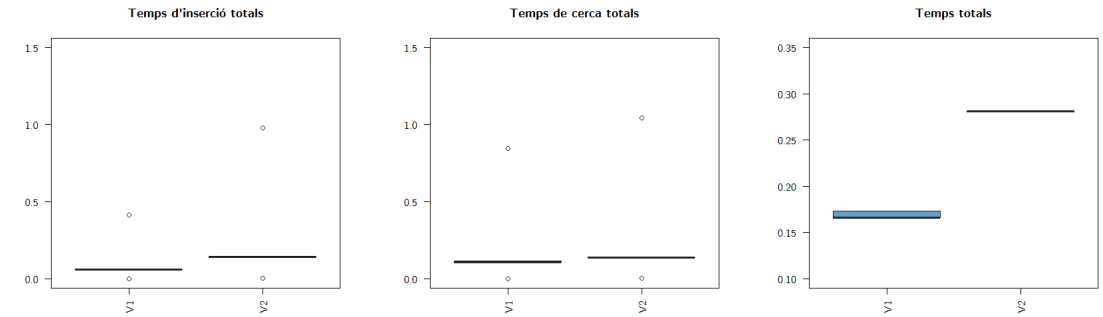

Figura 3: D'esquerra a dreta, la distribució del temps d'execució d'inserir els elements, de cercar-los i el total.

5.2 *Hash* amb taula

Per tal de comparar els algorismes de *hash* amb taula implementats, hem realitzat 5 execucions diferents per a cada joc de proves i tècnica. A continuació es detallen els resultats obtinguts amb la mitjana dels conjunts de dades 2, 3 i 4 amb mida de diccionari 10.000, mida de text 20.000 i d'un 20% aproximadament de la mida del conjunt unió entre els dos respecte al diccionari.

Per començar, hem realitzat experiments variant el nombre de passos de la funció de *hash* per tal d'escollir un valor i procedir amb la resta.

Podem observar a partir dels resultats, que *linear* i *double* són pràcticament idèntics (així doncs, els hem unit en un sol gràfic). Veiem que en el cas del *double* hi ha pics per a certs valors, que evidentment, hem d'evitar. El cas de Cuckoo era d'esperar donat que la funció de *hash* que hem implementat no inclou el nombre de passos en la seva fórmula, així doncs, variar aquest paràmetre no farà que el resultat canviï.

Una vegada vists els resultats de les figures 4 i 5, sobretot més en concret d'aquesta última, hem procedit a escollir un valor per la variable de passos. Finalment, comparant el nombre de salts i també alguns valors de temps, hem acordat utilitzar el nombre 13 pels resultats obtinguts i també per ser un nombre primer.

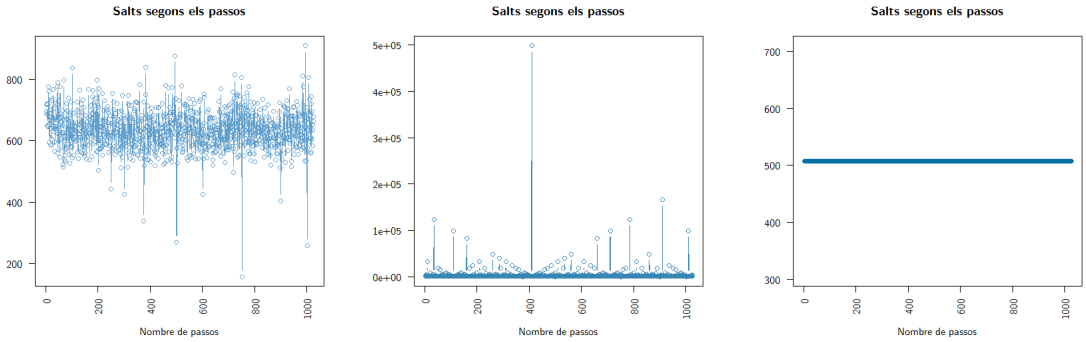

Figura 4: D'esquerra a dreta, la distribució del nombre de salts segons el nombre de passos (de 0 a 1024) per *linear* i *quadratic*, *double* i Cuckoo.

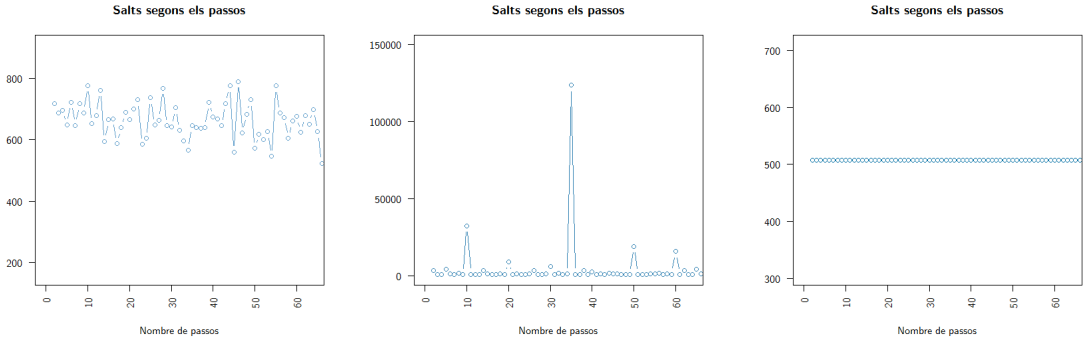

Figura 5: D'esquerra a dreta, la distribució del nombre de salts segons el nombre de passos (de 0 a 64) per *linear* i *quadratic*, *double* i Cuckoo.

D'acord amb el que es diu al final de la secció d'experiments i abans d'entrar en cap subapartat, els valors de c_1 i c_2 s'han fixat a 11 i 17 després de plantejar si era necessari realitzar el mateix experiment per aquests dos valors. Com que també es poden tenir en compte com a nombre de passos sent c_2 el valor de c_1^2, hem cregut convenient definir-los en els valors esmentats anteriorment comparant salts, temps i tenint en compte que també són primers.

Una vegada fixat el valor del nombre de passos a 13, continuem experimentant, aquesta vegada comparant el nombre de salts. Els resultats obtinguts a partir d'aquí també són de la mitjana de 5 execucions, però no només amb els conjunts de dades 2, 3 i 4, sinó que també inclouen l'1 i el 5. En el primer gràfic de la figura 6 podem observar com la tècnica que genera menys salts de totes és el Cuckoo, seguida de ben a prop pel *linear* i el *quadratic* que tenen pràcticament el mateix valor. Veiem que les dues tècniques de *chaining* han obts una mica tingut uns resultapitjors a les d'*open hashing* generalment.

De cara a experimentar amb temps d'execució, cal recordar que s'ha realitzat una mitjana de cinc execucions per a cada joc de proves separat per obtenir uns resultats més fiables.

Pel que fa als temps d'inserció, podem veure clarament com tot i que abans el *quadratic* tingués el mateix nombre de salts que el *linear*, té un temps d'inserció lleugerament inferior, tot i que això podria ser degut a la *caché* de la màquina i per tant, tampoc podem assegurar que sigui així. Si mirem Cuckoo, veurem que efectivament té un temps inferior d'inserció. Això ja era d'esperar veient el menor nombre de salts que feia mirant el gràfic anterior. És així donat que generalment una col·lisió la pot solucionar només amb un salt donat l'ús d'una segona taula de *hash*.

Finalment, veiem com les dues tècniques de *separate chaining* tenen un cost superior. En el cas de la tècnica amb llista (*move front*), veiem que el temps és inferior al del vector (*exact fit*), cosa que és degut a la relocalització d'aquest últim quan les col·lisions superen la seva capacitat, i per tant, tenim el temps afegit de moure'l sencer a un altre lloc de la memòria. Cal recordar també que el cas del Cuckoo no contempla el *rehash*, cosa que provocaria un augment considerable en el temps d'inserció total.

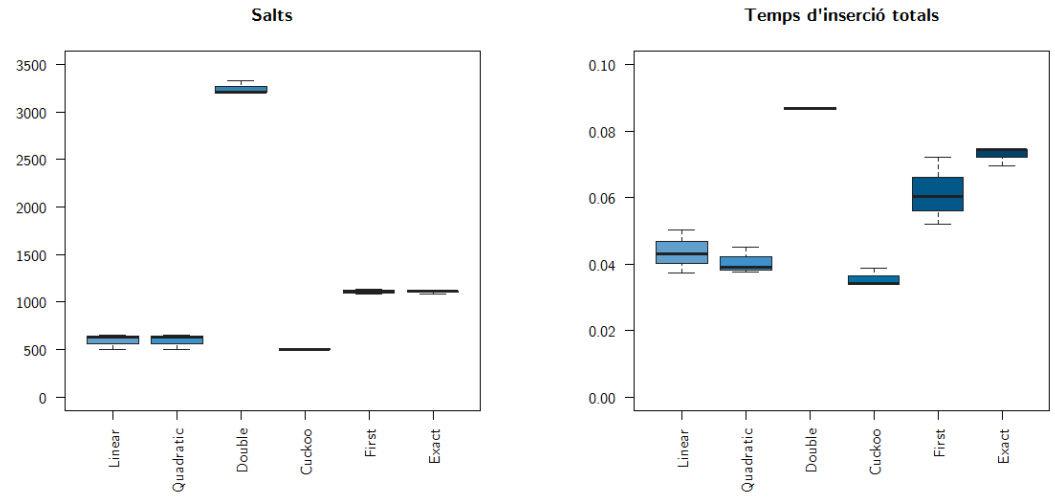

Figura 6: D'esquerra a dreta, la distribució del nombre de salts i el temps d'inserció total.

Pel que fa als temps de cerca d'elements al diccionari, i com s'ha comentat anteriorment, hem separat els temps de cerca dels elements que en formen part i dels que no.

Veient els temps de cerca podem trobar dues regions molt definides, tal i com es veu en el primer gràfic de la figura 7. Les tres primeres tècniques d'*open hashing* provoquen un augment molt important en el temps de cerca total. Aquesta diferència es veu molt clarament que prové dels elements que no formen part del diccionari tal i com es veu al gràfic de la dreta de la mateixa figura. Això és així donat que en aquests tres primers casos, anem fent salts per la taula buscant l'element indicat, així doncs, ens recorrem la taula (sencera o no) fins a trobar un cicle, és a dir, que tornem a mirar la posició que hem mirat la primera vegada. Com que l'element que cerquem no és a la taula, el temps de fer la cerca serà el del cost pitjor, justament per aquest recorregut.

El fet anterior canvia però en el cas de Cuckoo i de *separate chaining*. En el primer cas, només hem de comprovar dues posicions, així doncs, el cas pitjor només comprendrà un sol salt. Pel que fa al *move front* i *exact fit*, si la posició on hem anat a parar després d'aplicar la funció de *hash* és buida, no farem cap salt donat que no hi és, si té algun continut, només haurem de recórrer la llista o vector que tot i que podria contenir tots els elements, no passa en el cas mitjà (tenint una funció de *hash* mínimament correcta, un mòdul per exemple, ja ho evitem en la majoria de casos).

Per acabar, si ens fixem en els temps de cerca d'elements compresos al diccionari, veurem que efectivament i tal com hem dit abans, el temps de cerca d'elements a Cuckoo és inferior que als de les tècniques de *separate chaining* donat que farem com a molt un salt, és a dir, que limitem a que la llista o vector (per fer una comparació amb els altres dos) tingui mida 2. Recordem però que el temps d'inserció total és més baix degut a la no implementació del *rehash*.

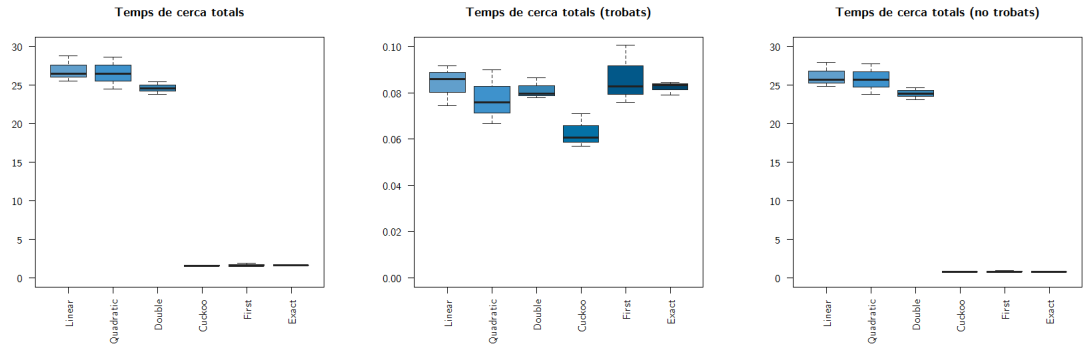

Figura 7: D'esquerra a dreta, la distribució del temps de cerca total, cerca d'elements que pertanyen al diccionari i cerca d'elements que no hi pertanyen.

A part, també podem veure l'evolució dels salts i dels errors en l'inserció per cada joc de proves. Ho visualizem molt bé als gràfics de la figura 8. En l'eix inferior de cada gràfic hi ha la indicació de quina entrada correspon a cada valor. Donat que les entrades 2, 3 i 4 tenen la mateixa mida, s'ha triat la 3 donat que s'aproxima molt a la mitjana de les tres.

La gràfica però no ens ha de fer veure que el nombre de salts és exponencial en relació a la mida del text. Cal tenir en compte que la mida de l'entrada 1 és de 16, la de l'entrada 3 de 2.000 i la de l'entrada 5 de 20.000. Observem però que en el cas del gràfic d'errors d'inserció, només n'hi ha un que augmenta mentre que la resta romanen a 0. Aquests valors corresponen a Cuckoo i es produeixen quan troba un cicle.

Pel que fa al gràfic de l'esquerra, de baix a dalt de l'entrada 5, les dades corresponen a *linear*, *quadratic*, Cuckoo, *move first*, *exact fit* i *double*.

Si ens fixem però en els mateixos valors però relatius a la mida d'entrada i tenint en compte tal i com s'ha dit

anteriorment que la mida del conjunt unió entre el diccionari i el text és d'un 20% aproximadament respecte al segon, podem veure que efectivament no hi ha cap creixement lineal. Tot al contrari, podem observar que a mesura que l'entrada creix, el nombre de salts i d'errors es redueix. Segurament s'estabilitzarà i ho entenem com una manera d'aproximar un percentatge de salts i d'errors per qualsevol entrada aleatòria.

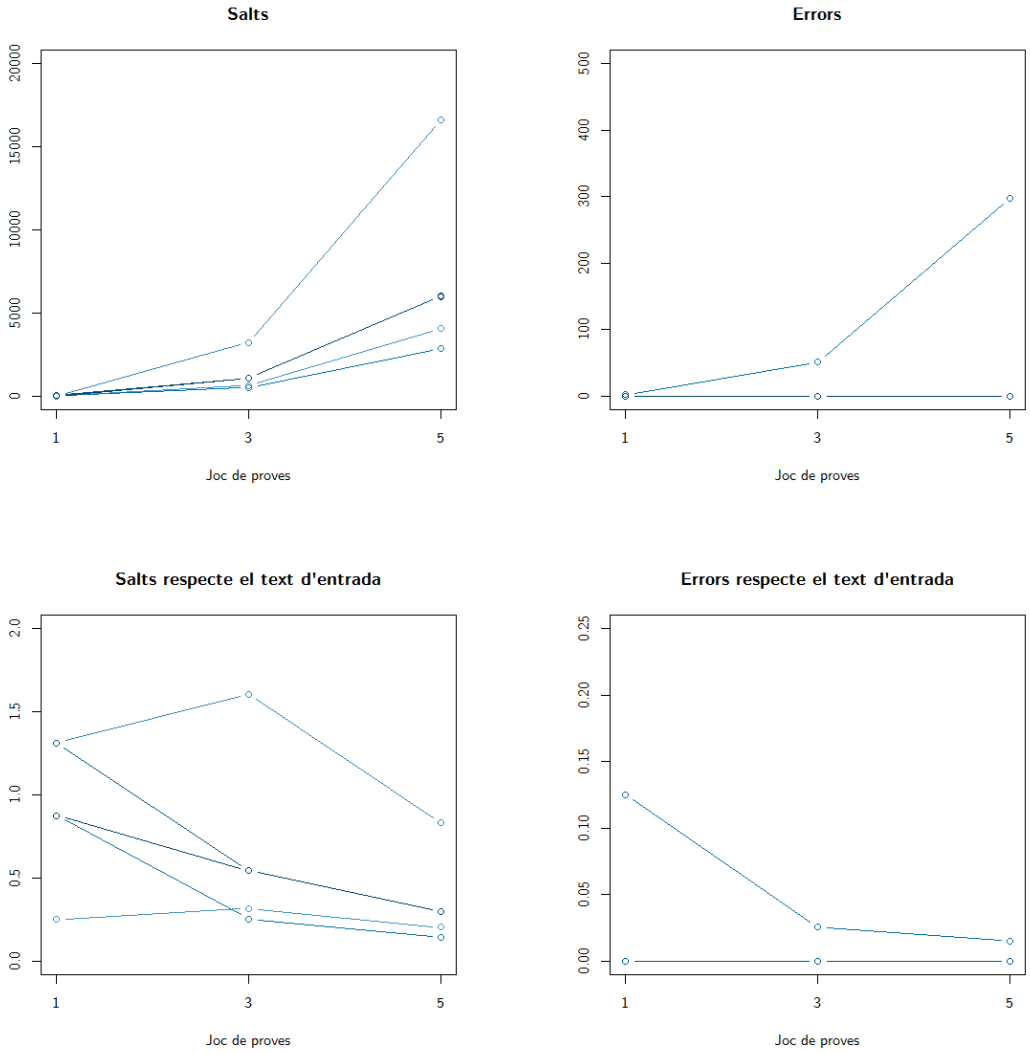

Figura 8: D'esquerra a dreta, l'evolució del nombre de salts i del nombre d'errors en l'inserció.

5.3 Filtre de Bloom

Sobre els filtres de Bloom, també hem realitzat 5 execucions per a cada joc de proves per tal d'extreure'n les mitjanes corresponents. Una vegada fets, ens n'adonem de la quantitat de falsos positius que genera la primera versió tal i com es veu al gràfic de l'esquerra de la figura 9. Si ens fixem en el valor de la taula 13 i 14 per a l'entrada 5, veurem que en el cas de la primera versió es dispara a 128 mentre que la segona queda reduït a 9, un 0,00045% respecte les dades d'entrada. Veiem doncs que la reducció és molt important i que pot afectar de cara a decidir quin dels dos és millor.

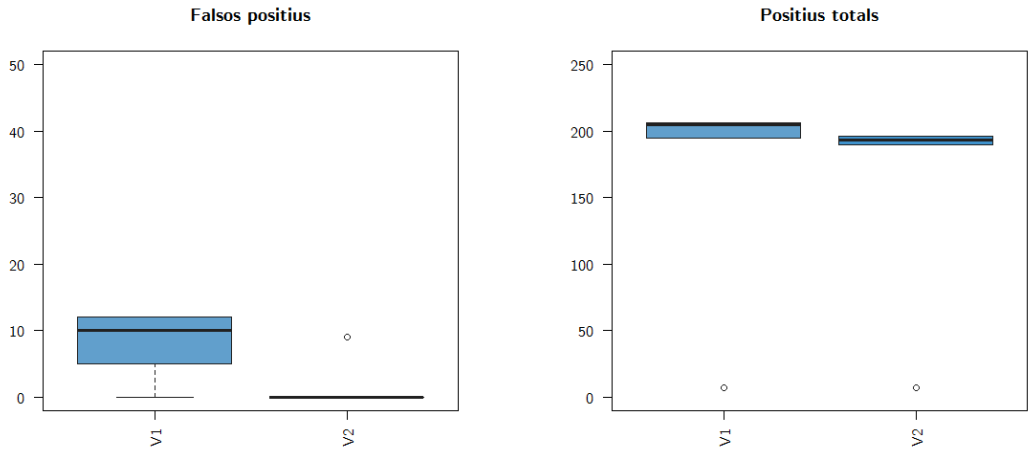

Figura 9: D'esquerra a dreta, el nombre de falsos positius i el nombre total de positius.

Pel que fa als temps, veiem que són molt similars, mentre que la primera versió és una mica més lenta inserint després és més ràpida cercant, i al revés. Finalment però, observem que la que té menys temps de mitjana és la segona, tot i que sembla que per a entrades grans el tems podria ser bastant més superior que el de la primera com es veu a la taula 14.

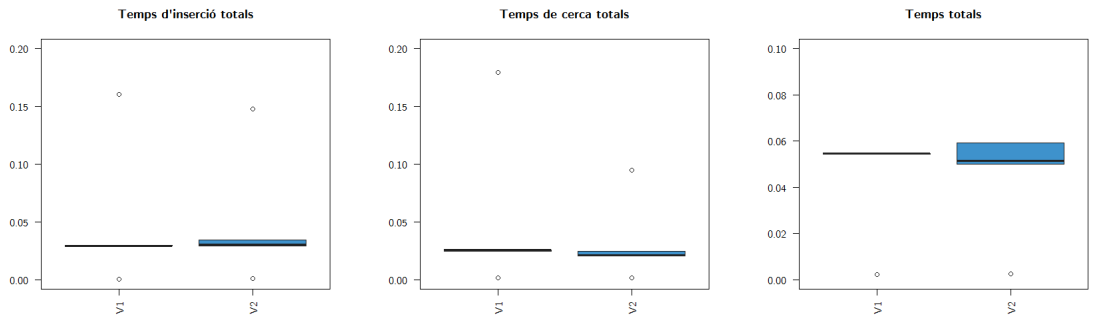

Figura 10: D'esquerra a dreta, la distribució del temps d'execució d'inserir els elements, de cercar-los i el total entre els dos.

5.4 Comparació final

Finalment i per tal de determinar quin dels mètodes implementats en principi seria millor, hem fet la comparativa entre els millors de cada subapartat anterior. Hem utilitzat també 5 execucions per a tots els jocs de proves disponibles.

Podem veure clarament els resultats de la comparació en la figura 11. Pel que fa a temps d'inserció, veiem que *binary* és el que té un temps superior, seguit de *double*, Cuckoo i Bloom. Cal recordar, tal i com s'ha dit abans, que Cuckoo no inclou el temps de *rehash* en cas que fos necessari (que ho és), i per tant, també hem inclòs el *double*.

Si ens fixem en els temps de cerca totals, la cosa canvia. La tècnica que triga més ara a realitzar aquest procés és la *double* mentre que el *binary*, que abans era la més costosa, ara és una de les més ràpides.

Finalment, si ens fixem de més a prop en els temps totals d'inserció i cerca, observem que la segona versió del filtre de Bloom és la més ràpida de totes, tot i això la cerca binària s'hi apropa bastant. En concret, Bloom triga poc més de 0,05ms mentre que la cerca binària ho fa en menys de 0,3ms. Si ens fixem però en entrades molt grans, podem veure que efectivament tot i que el filtre de Bloom només triga 0,25ms, *binary* ho fa en 2ms, 8 vegades més.

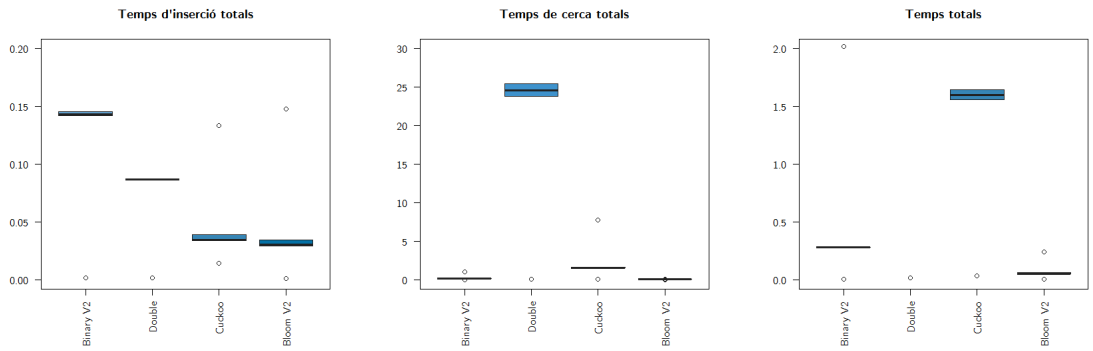

Figura 11: D'esquerra a dreta, la distribució del temps d'execució d'inserir els elements, de cercar-los i el total entre els dos.

6 Conclusions

Després d'implementar tots els algorismes indicats en la secció 1 i de realitzar tots els experiments d'acord amb una bona metodologia i disseny tal i com es pot veure a la secció 5, procedim a extreure'n conclusions.

D'acord amb l'experiment final on comparem les millors tècniques de cada subapartat, podem afirmar que *a priori*, la versió de filtres de Bloom amb una k i una relació n/m seleccionades de manera no arbitrària, experimentant amb diversos valors abans de fixar una valors, ens resulta l'opció més adient sempre i quan ens trobem davant d'un sistema que admeti una proporció molt reduïda de falsos positius.

En cas que el sistema no suporti incerteses i hagi de tenir un rigor absolut, segons el mateix experiment final, la segona millor opció que se'ns planteja sembla ser la cerca binària implementada amb arbres AVL. Tot i això, i si només féssim comparacions de temps d'execució, tal i com s'ha vist en el primer apartat de l'experimentació, la *binary* amb *arrays* té un cost inferior tot i realitzar més comparacions.

Tot i que la nostra sensació incial era que les tècniques de *hash* amb taula serien millors que la resta, hem pogut comprovar que no sempre és així. Tot i això, en el cas que *hash* fos millor, creiem que tant Cuckoo com el *move first* serien una de les dues millors implementacions possibles. Considerant el temps possible de *rehash* que podria comportar Cuckoo, la versió de *separate chaining* podria arribar a ser millor. Per a valors arbitraris dels seus paràmetres, *chaining* no hauria de realitzar moltes comprovacions, sobretot en el cas del *move first* a diferència de les dues que sempre realitza Cuckoo segons la nostra implementació.

Bibliografia

[1] Balogun, B. G.; Sadiku, J. S. «Simulating Binary Search Technique Using Different Sorting Algorithms». *International Journal of Applied Science and Technology*. Ilorin, Nigèria: Yunifásítì ìlú Ilorin, Agost 2013, vol. III, núm. 6, p. 67-75.

[2] Broder, Andrei; Mitzenmacher, Michael. «Network Applications of Bloom Filters: A Survey». *Internet Mathematics*. Massachusetts, Estats Units d'Amèrica: A K Peters, Ltd., 2004, vol. I, núm. 4, p. 485-509.

[3] Grill, Bernhard. *A Survey on Efficient Hashing Techniques in Software Configuration Management*. Viena, Àustria: Technische Universität Wien, 2014.

[4] Mitzenmacher, Michael; Upfal, Eli. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge, Anglaterra: Cambridge University Press, 2005, p. 107-112. ISBN 9780521835404.

[5] Pagh, Rasmus; Rodler, Flemming Frichie. «Cuckoo Hashing». *Journal of Algorithms*. Aarhus, Dinamarca: Aarhus Universitet, 2004, vol. LI, núm. 2, p. 122-144.

[6] «Hash Tables». *Further Programming in Java*. [En línia]. Melbourne, Austràlia: Royal Melbourne Institute of Technology, 2006. <http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html>. [Consulta: 28/04/2018].

A Taules de resultats

El següent annex és un extracte dels resultats obtinguts en l'apartat 5, els arxius originals es poden trobar a la carpeta de taules dels arxius lliurats juntament amb aquesta documentació.

A.1 Cerca binària

Dades	Comp. d'inserció	Comp. de cerca	Temps d'inserció	Temps de cerca	Temps total
1	16	105	0.0006	0	0.0006
2	8689	48813	0.0582	0.1078	0.166
3	8702	49108	0.0578	0.1084	0.1662
4	8721	49005	0.0582	0.115	0.1732
5	55208	304652	0.4144	0.8424	1.2568

Taula 1: Versió 1 de la cerca binària amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Comp. d'inserció	Comp. de cerca	Temps d'inserció	Temps de cerca	Temps total
1	29	97	0.0014	0.0008	0.0022
2	17676	39943	0.1452	0.1366	0.2818
3	17561	39702	0.1422	0.1382	0.2804
4	17572	39780	0.143	0.1382	0.2812
5	112202	248536	0.9772	1.0436	2.0208

Taula 2: Versió 2 de la cerca binària amb 5 conjunts de dades i la mitjana de les 5 execucions.

A.2 *Hash* amb taula

A.2.1 Nombre de passos

Passos	Salts	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	717.00	0.027	22.909	0.078	22.238
2	688.33	0.05	11.727	0.075	11.067
3	695.67	0.034	8.131	0.069	7.512
4	648.67	0.024	6.254	0.069	5.644
5	722.00	0.049	5.176	0.062	4.583

Taula 3: *Linear probing* amb la mitjana de 5 conjunts de dades i diferents valors de passos. Gran part de la taula s'ha omès per llargada, la resta de la taula la podeu trobar en un fitxer a la carpeta de taules dels arxius lliurats amb aquesta documentació.

Passos	Salts	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	717.00	0.024	22.476	0.068	21.823
2	688.33	0.033	14.126	0.086	13.41
3	695.67	0.046	9.527	0.072	8.779
4	648.67	0.027	7.375	0.071	6.647
5	722.00	0.037	6.325	0.057	5.571

Taula 4: *Quadratic probing* amb la mitjana de 5 conjunts de dades i diferents valors de passos. Gran part de la taula s'ha omès per llargada, la resta de la taula la podeu trobar en un fitxer a la carpeta de taules dels arxius lliurats amb aquesta documentació.

Passos	Salts	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	3218.00	0.064	24.839	0.077	24.12
2	732.00	0.054	12.785	0.075	12.095
3	713.67	0.041	9.165	0.07	8.442
4	4094.67	0.074	6.947	0.072	6.26
5	1088.67	0.065	5.87	0.078	5.18

Taula 5: *Double hashing* amb la mitjana de 5 conjunts de dades i diferents valors de passos. Gran part de la taula s'ha omès per llargada, la resta de la taula la podeu trobar en un fitxer a la carpeta de taules dels arxius lliurats amb aquesta documentació.

Passos	Salts	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	507.67	0.016	1.31	0.054	0.637
2	507.67	0.023	1.126	0.055	0.544
3	507.67	0.013	1.16	0.042	0.546
4	507.67	0.012	1.209	0.061	0.561
5	507.67	0.015	1.126	0.052	0.527

Taula 6: Cuckoo *hashing* amb la mitjana de 5 conjunts de dades i diferents valors de passos. Gran part de la taula s'ha omès per llargada, la resta de la taula la podeu trobar en un fitxer a la carpeta de taules dels arxius lliurats amb aquesta documentació.

A.2.2 Salts i temps

Dades	Salts	Errors	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	4	0	0,0016	0,0156	0,0024	0,0056
2	655	0	0,0504	28,7912	0,086	27,9294
3	633	0	0,0374	25,549	0,0744	24,8094
4	496	0	0,043	26,4556	0,0918	25,6618
5	4051	0	0,2224	587,4336	0,1726	584,0428

Taula 7: *Linear probing* amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Salts	Errors	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	4	0	0,0016	0,0184	0,0034	0,0078
2	655	0	0,039	26,4956	0,0758	25,7054
3	633	0	0,0452	28,6108	0,09	27,761
4	496	0	0,0376	24,4654	0,0666	23,7584
5	4051	0	0,1986	591,6906	0,1816	588,2954

Taula 8: *Quadratic probing* amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Salts	Errors	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	21	0	0,0018	0,0162	0,0036	0,0052
2	3334	0	0,0872	24,615	0,078	23,907
3	3209	0	0,0864	25,4416	0,0866	24,6828
4	3214	0	0,0868	23,8168	0,0796	23,1344
5	16639	0	0,4556	584,0416	0,1856	580,6692

Taula 9: *Double hashing* amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Salts	Errors	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	14	2	0,014	0,0162	0,0016	0,007
2	513	50	0,0342	1,5242	0,0606	0,7142
3	505	51	0,0388	1,604	0,071	0,7518
4	505	49	0,0338	1,565	0,0568	0,7496
5	2870	297	0,1336	7,7546	0,1492	3,8344

Taula 10: Cuckoo *hashing* amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Salts	Errors	T. d'inserció	T. de cerca	T. de cerca trobats	T. de cerca no trobats
1	21	0	0,003	0,0172	0,0034	0,0054
2	1132	0	0,0602	1,5478	0,0828	0,7214
3	1086	0	0,072	1,879	0,1006	0,8844
4	1109	0	0,052	1,5516	0,0758	0,7202
5	5984	0	0,2474	7,0186	0,1954	3,4688

Taula 11: *Move front* amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Fals positius	Positius totals	Temps d'inserció	Temps de cerca	Temps total
1	0	7	0.0006	0.0016	0.0022
2	12	205	0.029	0.0256	0.0546
3	10	206	0.0296	0.0248	0.0544
4	5	195	0.029	0.0256	0.0546
5	128	636	0.1602	0.1794	0.3396

Taula 12: *Exact fit* amb 5 conjunts de dades i la mitjana de les 5 execucions.

A.3 Filtre de Bloom

Taula 13: Versió 1 del filtre de Bloom amb 5 conjunts de dades i la mitjana de les 5 execucions.

Dades	Fals positius	Positius totals	Temps d'inserció	Temps de cerca	Temps total
1	0	7	0.0008	0.0016	0.0024
2	0	193	0.0344	0.0248	0.0592
3	0	196	0.0306	0.0208	0.0514
4	0	190	0.029	0.021	0.05
5	9	517	0.1476	0.0948	0.2424

Taula 14: Versió 2 del filtre de Bloom amb 5 conjunts de dades i la mitjana de les 5 execucions.

B Dades d'entrada

Les entrades estan ordenades de menor a major mida. L'entrada 1 té una mida reduïda per proves ràpides, mentre que la 5 consisteix en un text de 10.000 caràcters. Pel que fa a l'entrada 2, 3 i 4 tenen totes la mateixa mida de text de 2.000 caràcters i són les utilitzades principalment pels experiments. Els nombres de totes les entrades han estat generats aleatòriament assegurant un conjunt mínim resultant de la unió del diccionari amb el text en cada cas.
En aquest annex només hem inclòs la primera entrada per la seva mida reduïda. La resta estan disponibles dins de la carpeta de dades dels fitxers entregats juntament amb aquesta documentació.

B.1 Entrada 1

Diccionari
8
9611 7255 5176 3196 6664 9744 9540 820
Text
517682 7255 5176 3196 6664 9744 9540 820 4937 9195 1875 1686 1091 1084 2376 3021