PROJECTE “SUPERSOPA”

Bernat Borràs Civil, Alexandre Ros i Roger, Oscar Ramos Núñez, Raúl Medina Martínez

Índex

1 Introducció

En aquesta pràctica d'algorísmia, se'ns demana implementar diverses Estructures de Dades i Algorismes per a tal de cercar paraules d'un diccionari en una sopa de lletres. La sopa de lletres, anomenada "SuperSopa", té una mida d' *NxN* caràcters en l'alfabet anglès, en majúscules.

Formalment, és una matriu *NxN* on *N* és un valor definit en temps de compilació, possiblement variable, i on cada entrada de la *SuperSopa* és un caràcter representat per 1 Byte.

La *SuperSopa* serà generada pel programa en temps d'execució, i s'hi amagaran aleatòriament i exactament vint paraules d'un diccionari. El diccionari, de llargada desconeguda, serà llegit per l'entrada estàndard (stdin) del programari. És imperatiu per tal que tot funcioni correctament que per l'entrada estàndard es llegeixin cadenes de caràcters en el rang [a..z-A..Z], separades per espais en blanc, LF o CRLF.

Hem pres la decisió que, per a aquelles caselles on no s'hi amagui cap paraula de les vint preseleccionades, estiguin ocupades per lletres escollides a l'atzar.

Les paraules es poden amagar en qualsevol direcció, i aquesta direcció pot variar. És a dir, una paraula és dins de la *SuperSopa* si es pot ubicar dins del tauler de manera que les lletres consecutives de la paraula estiguin en posicions adjacents horitzontalment, verticalment o en diagonal. Tot i així, no és possible que dues lletres repetides d'una mateixa paraula es trobin a una mateixa posició. Però, sí que és possible que dues paraules diferents "col·lideixin".

Després de llegir l'entrada estàndard, el nostre programari buscarà, per a cada implementació proposada, totes les paraules del diccionari que s'hi puguin trobar a la *SuperSopa*, estiguin al subconjunt de vint paraules o no. Per a cada implementació es generarà un fitxer amb extensió **.out*, que contindrà les paraules trobades en ordre alfabètic i una primera línia amb el temps estimat que ha trigat l'algorisme. Les Estructures de Dades que farem servir seran les següents:

I. *Vector Ordenat*
II. *Filtres de Bloom*
III. *Taula Hash amb doble hash*
IV. *Ternary Search Tree*

Els fitxers font que implementen cada estructura de dades i algorisme són, respectivament:

I. *diccSortedVector.hh* i *diccSortedVector.cc*
II. *BloomFilter.hh* i *BloomFilter.cc*
III. *diccDHashing.cc* i *diccDHashing.hh*
IV. *diccTernary.hh, diccTernary.cc, Ternary.hh* i *Ternary.cc*

També es troben dos diccionaris de prova que hem emprat, dues llistes de paraules:

I. *EFFwordlist.txt*, de la *Electronic Frontiers Foundation* (7.776 paraules)
(https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases)
II. *CORNCOBwordlist.txt*, de *Mieliestronk* (58.108 paraules)
(http://www.mieliestronk.com/wordlist.html)

Per defecte els dos estan ordenats alfabèticament. Hem randomitzat les files de cada fitxer i generat els seus corresponents amb sufix *-random.txt.* A més a més, hem proporcionat un *Makefile* per a compilar convenientment amb el compilador de C / C++ de GNU (g++). Simplement, la comanda *make* compilarà tot i generarà un executable *program.x*, i la comanda *make clean* eliminarà els executables, fitxers objecte i altres generats durant la compilació.

Finalment, hem adjuntat un petit programa escrit en Python, *histogram.py*, que podrà generar histogrames i poder comprovar la uniformitat dels diferents hashes que hem emprat.

2 Vector Ordenat i Cerca Dicotòmica

Aquesta primera implementació consta d'un sol vector amb totes les paraules ordenades ascendentment per ordre lexicogràfic. S'inicialitza el vector amb totes les paraules i, finalment, s'ordena el vector.

Per a ordenar el vector hem utilitzat la comanda *sort* de la llibreria *algorithmics*, que fa servir l'*Introsort* (o *Introspective Sort*), un algorisme d'ordenació híbrid per comparació, degut a que es tracta de la combinació del *Quicksort, Heapsort* i *Insertion Sort*.

L'*Introsort* utilitza el *Quicksort* fins a arribar a un nivell de profunditat recursiu límit basat en el logaritme dels elements que s'estan ordenant, canviant al *Heapsort*. Es passa a utilitzar *l'Insertion Sort* quan el nombre d'elements baixa d'un determinat límit.

Per tant, l'*Introsort* combina les parts bones dels tres algoritmes, fent que el rendiment sigui pràcticament igual al del *Quicksort* però amb un cost en el cas pitjor de *O(n log n)* gràcies al *Heapsort*.

A més a més, per tal d'accelerar i optimitzar la cerca, s'ha fet ús d'un vector auxiliar de vint-i-sis posicions, anomenat *Index*, que indicarà per a cada lletra (índex del vector) on es troba la primera paraula començant per la lletra al vector principal (o -1 si no hi ha cap paraula que comenci per la lletra al diccionari). Això només afegeix un cost *O(log n)* a la inicialització, fent que es mantingui el cost asimptòtic.

Per a trobar les paraules dins la sopa es recorreran les caselles i, primerament, es comprovarà que hi hagi paraules al diccionari que comencin per la lletra de la casella que s'està observant. Si n'hi ha, es comprovarà que la pròpia lletra formi part del diccionari.

Com que hi ha paraules que comencen per la lletra (prefix) observada, es formarà un *string* afegint lletres adjacents i es farà una cerca dicotòmica, començant per la primera posició en la que es troben paraules que comencen pel mateix prefix, i fins la posició anterior a la primera en la que es troben paraules que comencen per la següent lletra que es trobi al diccionari. Quan no es trobi el prefix al diccionari, es deixarà d'afegir lletres de caselles adjacents i es passarà al següent. Una vegada s'acabin els prefixos, es passarà a la següent casella.

A més a més, per tal d'accelerar i optimitzar la cerca, s'ha fet ús d'un vector auxiliar de vint-i-sis posicions, anomenat *Index*, que indicarà per cada lletra (índex del vector) on es troba la primera paraula començant per la lletra al vector principal.

On *n* és el nombre de paraules del diccionari:

- El cost espacial serà de *O(n)*, on *n* és el nombre de paraules
- El cost temporal d'inicialització serà de *O(n log n)*, doncs caldrà ordenar tots els elements
- El cost de cercar un element dicotòmicament serà de *O(log n)*

3 Filtres de Bloom

Un filtre de Bloom és una estructura de dades probabilística que busca ésser eficient en espai, usat per a intentar determinar si un element (en el nostre cas, paraules) és d'un conjunt (en el nostre cas, el diccionari) o no. És possible que, amb els filtres de Bloom, arribem a *falsos positius,* és a dir, el filtre ens digui que un element està al conjunt quan en veritat no hi és; però no és possible trobar-nos amb *falsos negatius*. Per tant, si el filtre ens diu que un element no hi és, aquell element segur que no és en el conjunt inicial.

L'estructura en sí es compon d'un vector de bits d'una certa llargada *m*, i a més a més necessita *k* funcions de hash que converteixin el tipus d'element (en el nostre cas, una *string*), a una posició entre [0..*m*]. El vector serà implementat amb un *vector<bool>,* doncs les versions més recents de C++ ho implementen eficientment amb un bit per a representar un booleà.

Inicialment, quan no hi hagi cap element al filtre, tot bit del vector serà inicialitzat a zero (*false*). Per a inserir una paraula, s'evaluarà cada funció de hash amb la paraula i les posicions que retornin cada funció de hash seran canviades per un u (*true*).

Per a probar una paraula, s'evaluarà cada funció de hash amb la paraula i només si totes les posicions que retornen cada funció de hash estan a 1, llavors la paraula **pot** estar al conjunt, amb una probabilitat de *fals positiu* que podrem refinar amb *k* (nombre de funcions hash) i *m* (llargada del vector). Cal recordar que, si alguna de les funcions retorna una posició on hi ha un zero, llavors la paraula **no pot** estar al vector.

És imperatiu que per tal que el filtre funcioni correctament les funcions de hash seguiexin distribucions uniformes. Per aquesta raó hem realitzat un estudi de cadascuna de les sis funcions de hash que han estat implementades pel filtre de Bloom.

Anàlisi de la taxa de falsos positius i optimalitat d' m

Nosaltres hem optat per a tenir el nombre de funcions de hash constant a sis ($k = 6$). Per tant, l'única variable que podrem modificar per tal d'aconseguir una taxa de falsos positius (que representarem amb la variable *p*) serà la llargada del nostre filtre, *m.*

Deduirem l'optimalitat d' *m* a partir de *k, p* i *n,* on *n* és el nombre d'elements al nostre diccionari:

Primer, la probabilitat que *una* funció de hash posi a 1 una determinada posició del vector és de $1/m$ donat que podem suposar que és uniforme, i doncs la probabilitat que no posi una determinada posició a 1 és de $1-1/m$. Donat que tenim *k = 6* funcions de hash, la probabilitat de que cap funció de hash posi a 1 una determinada posició del vector és de $(1-1/m)^6$.

Segueix que per a una *m* prou gran, podem aplicar la definició per la constant d'Euler i veure que, de forma aproximada, $(1-1/m)^6=((1-1/m)^m)^{6/m}\simeq e^{-6/m}$. Havent inserit *n* elements, la probabilitat de que una determinada posició encara estigui a zero és de $e^{-6n/m}$, i anàlogament la probabilitat de que estigui a 1 és de $1-e^{-6n/m}$.

Amb aquesta última probabilitat, ja podem aproximar *p*. *p* és la probabilitat de que un element que no està al conjunt doni positiu. Ergo, és la probabilitat que sis posicions aleatòries estiguin a 1. És a dir, que podem aproximar *p* amb la següent fórmula: $p\simeq(1-e^{-6n/m})^k$.

Llavors, donada la *p* que volem aconseguir, i donada la llargada del diccionari que sabrem abans de construir el filtre, la llargada del filtre òptima serà $m=\frac{-6n}{\ln(1-e^{\ln(p)/6})}$.

Aquesta és la funció que s'empra a la constructora per tal d'optimitzar els falsos positius. El valor *pFalsePositive* és, per defecte, de 0.00001 (0.01 ‰).

Com s'ha plantejat l'algorisme i anàlisi dels costs

És evident que necessitem un filtre de Bloom per al diccionari de paraules. Tot i així, al fer una cerca per la *SuperSopa*, com ens ho farem per a cercar les paraules que s'hi troben de forma eficient? Com ens ho fem per a saber si un cert prefix és un prefix d'alguna paraula existent al diccionari?

Nosaltres hem decidit implementar un segon filtre de Bloom, on hi inserirem els prefixos de cada paraula del diccionari. El fitxer *main.cc* calcula el nombre de prefixos totals i passa aquest paràmetre a la constructora de *Bloom*, amb la taxa de falsos positius desitjada.

El cost en espai del filtre primari serà, evidentment, $O(m) = O(n)$ doncs m depèn d' n linialment. Concretament, l'espai que ocuparà el filtre a memòria és d'exactament m bits. Tot i així tenim un segon filtre pels prefixos, que ocuparà en espai $O(nv)$, on n és el nombre de paraules del diccionari i v la longitud mitjana de cada paraula.

- El cost espacial serà de $O(nv)$, n és el nombre de paraules i v la longitud mitjana d'una paraula
- El cost temporal d'inicialització serà de $O(nv)$, per a inicialitzar els dos filtres amb zeros.
- El cost de cercar o afegir una paraula sempre serà lineal respecte la paraula, $O(kv)$, doncs fa falta per a cada funció de hash iterar per cada caràcter de la paraula.

Anàlisi de les funcions de hash

Una de les propietats més importants a l'hora d'escollir una funció de hash és la seva uniformitat. Una funció de hash no uniforme pot portar a moltes col·lisions i, en el cas dels filtres de Bloom, molts més falsos positius dels esperats.

En aquesta implementació de dos filtres de bloom utilitza sis funcions de hash, i hem pogut analitzar la seva uniformitat i dispersió amb histogrames.

Com s'ha comentat en la introducció d'aquest document, a més de les classes i fitxers relacionats amb el programa en sí, s'ha fet entrega d'un programa escrit en *Python*, *histogram.py,* que genera sis histogrames amb la llibreria *matplotlib* de cadascuna de les funcions de hash implementades.

Per tal de fer-ho, hi ha diverses línies comentades en els fitxers *BloomFiler.cc* (línies 12-14, 179-184) i a *BloomFilter.hh* (línia 22) que, si es descomenten, generaran sis diferents fitxers *h1, h2, ..., h6*.

Per a cada paraula del diccionari llegit, la funció *hash1* evaluarà el primer hash i, a més de posar la posició del filtre a 1, imprimirà aquesta posició al fitxer *h1*. El mateix per a cadascuna de les sis funcions, de manera que el recompte de línies de cada fitxer *h{1..6}* és el nombre d'elements al diccionari.

Després, si s'executa *histogram.py* amb *Python* (cal haver instal·lat *matplotlib* amb *pip install*), es generaran sis histogrames, un per cada hash, que ajuden a valorar l'uniformitat de cadascun.

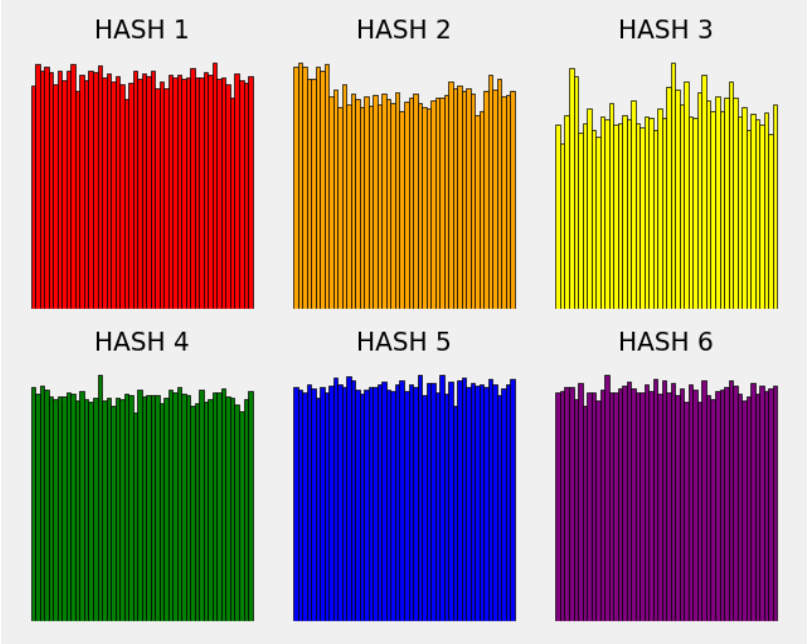

Figura 1: Histogrames de les funcions de hash amb el diccionari CORNCOB

Per simplificar-ho, s'han omès els eixos *X* i *Y* en les gràfiques. L'eix *X* és el valor de la funció de hash (mòdul amb el nombre d'elements del diccionari) i l'eix *Y* seria el nombre de paraules (ocurrències) que s'evaluen al valor *X*. A més a més, tots els histogrames es dibuixen amb *n_bins* rangs, on *n_bins* és una variable que es troba al principi de *histogram.py*. També es poden editar els colors, si no agraden.

Gràcies a aquesta fantàstica llibreria de *Python*, hem pogut, per exemple, observar perquè en una primera implementació hi havia un nombre exessiu de falsos positius:

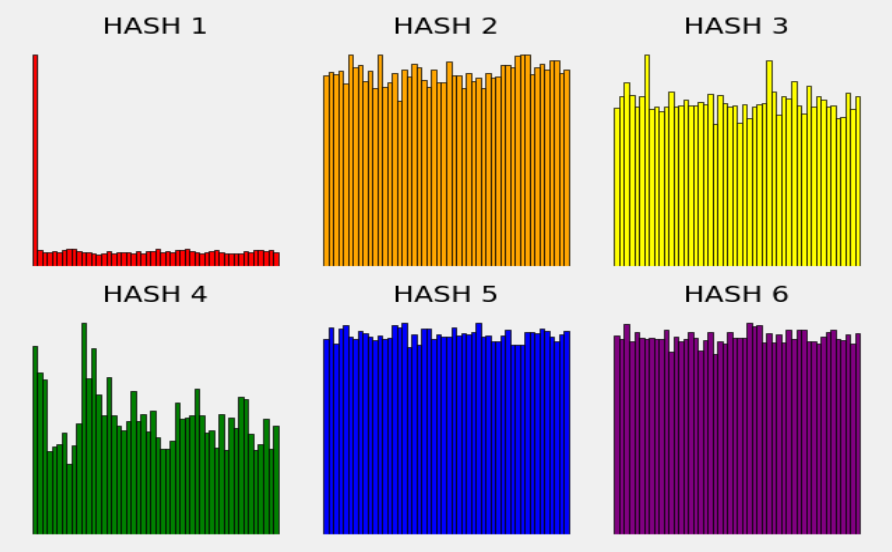

Figura 2: Histograma de les funcions de hash implementades incorrectament

Com es pot observar, la primera funció de hash que es feia servir era completament desigual i poc uniforme. El mateix passava amb la funció de hash número 4, que no presentava una bona uniformitat. Gràcies a aquests gràfics vam ser capaços d'arreglar els errors i de corregir les dues funcions de hash que donaven problemes.

Les funcions de hash definitives inclouen *Fowler-Noll-Vo (FNV), djb2, SDBM, Jenkin's One-at-a-time* i *Murmur3*. Totes estan implementades a *BloomFilter.cc*.

4 Taula Hash amb doble hash

Per a la tercera estructura de dades s'ha implementat una *Taula de Hash*, mitjançant un vector de cadenes de caràcters, amb una estratègia de *double hashing* a l'hora d'assignar les paraules. Una *taula de hash* ens permet emmagatzemar i cercar elements mitjançant claus o *key*s generades a partir de les funcions de hash.

Donat un element (que volem emmagatzemar o cercar) aquest, després de passar per la funció de hash primària, tindrà associat una *key* que el permetrà distingir-lo de quasi tots els altres elements. Tot i això, la clau no és única i la possibilitat de que dos elements en tinguin la mateixa ve donada per diversos factors:

- El *tamany de la Taula de Hash*: Evidentment, a més grandària de la taula, menys col·lisions es produiran amb la primera funció de hash doncs hi haurà més dispersió. Nosaltres hem pres la decisió de generar la taula principal de mida *p(2N)*, on:

  - *p(x)* retorna el número primer mínim d'aquells més grans que *x*.
  - *N* és el nombre d'elements que tindrà la taula.

  Hem implementat des de zero la funció es*Primer(x)*, amb 300 tests de primalitat de Fermat.

  D'aquesta manera, utilitzant nombres primers sempre asseguren menys regularitat i més uniformitat a l'hora de fer, per exemple, el residu amb ells.

- *La "qualitat" de les funcions de Hash*: Com també s'ha descrit a l'apartat dels *filtres de Bloom*, una funció de Hash ha de preservar diverses propietats per tal de ser eficient. Una d'elles és assegurar uniformitat i per tant, que per a qualsevol paraula *a* la probabilitat de prendre un determinat valor sigui $1/M$ on *M* és la mida de la taula.

El *double hashing* té un sistema per evitar les col·lisions formades per elements que comparteixen la clau generada per la primera funció de hash. Per aquesta raó, fem ús d'una segona funció de hash.

Sigui *a* una paraula tals que *h1(a)* ja està ocupat a la taula , on *h1: String → [0 .. N-1]* i *N* és la llargada de la taula, en una implementació estàndard d'una taula de hash acceptariem la col·lisió i crearíem una llista de dos paraules.

Amb el *double hashing*, s'aplica una segona funció de hash *h2 : String → [1 .. N]*. Després, la nova posició assignada per a la paraula *a* serà *h1(a) + h2(a).* Si aquesta posició segueix essent ocupada, tornem a sumar-li *h2(a)* fins que deixi de ser-ho. Per aquesta raó, és imperatiu que *h2(x) > 0* per a qualsevol *x*, o *h1(x) + 0 + 0 + 0.... = h1(x)* i sempre estariem col·lidint.

Les dues funcions de hash escogides han estat per *Multiplicació per nombre primer* i *djb2*.

1. *Multiplicació amb Nombre Primer:* Es basa en anar sumant el valor de cada lletra de la paraula multiplicat per un nombre primer. Aquesta multiplicació per un nombre primer és el que fa que sigui considerablement difícil arribar a un mateix valor per paraules diferents, gràcies a les propietats d'aquests.
2. *djb2:* La funció de Hash de Dan Bernstein és una coneguda funció de hash utilitzada sovint per la seva uniformitat contrastada. També ha estat implementada en els filtres de *Bloom.*

Pel que fa als costos d'aquesta estructura de dades en la *SuperSopa*, hem de tenir en compte que a més d'un vector per guardar les paraules del diccionari, hem utilitzat un segon vector auxiliar per a emmagatzemar els prefixos de les paraules i així fer la cerca més eficient. Sigui *n* la longitud del nostre diccionari, *v* la longitud mitjana d'una paraula:

- El cost en espai serà *O(2n + 2nv) = O(nv)*
- El cost temporal d'inicialització serà de *O(nv)*, per a inicialitzar els dos filtres amb ".".
- El cost de cercar o afegir una paraula és, en el cas mitjà, aproximadament *O(v).* Tot i així si arribem a una col·lisió el cas pitjor seria linial respecte *n* per la longitud de la paraula, *O(n*v*).*

5 Ternary Search Tree

Un *Ternary Search Tree* és una estructura de dades utilitzada per a determinar si un element (en el nostre cas, paraules) és d'un conjunt (en el nostre cas, el diccionari) o no, on l'element pot ser concatenat per diversos elements. Per tant, és una estructura ideal per a cadenes de caràcters.

Aquesta estructura de dades és una variant del *Trie*. Hem decidit utilitzar aquesta variant ja que un *Ternary Search Tree* utilitza menys memòria que altres variants de *Trie*.

Cada node del nostre *Ternary Search Tree* es compon d'un caràcter que correspon a la clau. També tenim tres apuntadors (left, right, center) que corresponen als tres fills possibles d'aquell node. En cas que un (o varis) fills no existeixin, l'apuntador prendrà el valor de *NULL*. Per a indicar si el node correspon a una paraula, utilitzem un booleà.

Inicialment el nostre diccionari estarà compost per a un node on la clau serà buida, i els tres apuntadors prendran el valor de *NULL*. Per a la primera paraula accedirem al node arrel i la clau d'aquest prendrà el valor de la primera lletra de la paraula. Per a afegir una paraula accedirem al node arrel i comprovem si la primera lletra de la paraula correspon amb la clau del node.

En cas afirmatiu, accedirem al node central. En cas contrari, farem una cerca binària utilitzant els fills de la dreta i l'esquerra. Si el valor de la lletra que busquem és menor a la clau del node farem la cerca pel fill de la dreta, altrament pel fill de l'esquerra.

Una vegada trobat el node amb la clau equivalent a la lletra de la paraula seguirem el procediment anterior. Per a totes les lletres de la paraula farem el mateix procediment. Si en algun moment el fill a visitar no existeix, és a dir, que pren valor *NULL*, crearem un node amb la clau desitjada, i l'apuntador del node fill farà referència en aquest creat. Una vegada arribem al node que correspon a la última lletra de la paraula a inserir, marcarem el nostre booleà a cert, indicant que el node és final d'una paraula.

Per a fer la cerca de les paraules de la *SuperSopa* farem un *DFS* a la vegada que recorrem el nostre *Ternary Search Tree*. Per a cada lletra de la sopa farem una cerca binària pels nodes de la dreta i l'esquerra de l'arbre.

Comencem la cerca pel node arrel. Si el valor de la lletra que busquem és menor a la clau del node farem la cerca pel fill de la dreta, altrament pel fill de l'esquerra. Si el node arrel o el node visitat tenen la clau equivalent a la lletra de la sopa accedirem al fill central d'aquest i és d'on seguirem fent l'algorisme *DFS*. Si al fer la cerca binària pel *Ternary Search Tree* no hem trobat cap node amb la clau desitjada, significarà que el prefix cercat a la sopa no s'inclou al nostre arbre, i per tant no hi haurà paraules al nostre diccionari que continguin aquest prefix.

Si durant el nostre recorregut visitem un node fill que és fill central i el booleà que ens indica si el prefix és una paraula pren el valor de cert, significarà que la paraula obtinguda de la sopa està inclosa al nostre diccionari.

Sigui n la longitud del nostre diccionari, v la longitud mitjana d'una paraula:

- El cost en espai serà de mitjana $O(log(nv))$, i en el cas pitjor $O(nv)$.
- El cost de cercar o afegir una paraula en el cas mitjà és $O(log(n + v))$, i en el cas pitjor $O(n + v)$.

6 Experimentació i Conclusions

En aquesta pràctica hem après molt sobre diferents estructures de dades que es poden utilitzar per a emmagatzemar i cercar cadenes de caràcters, una tasca fonamental en moltes aplicacions de la computació. Hem vist alguns molt eficients en espai, com els *Bloom Filters*, i alguns molt eficients en rapidesa, com la tècnica del *Double Hashing*.

Gràcies als dos diccionaris, de l'*EFF* i de *Mieliestronk*, i els diccionaris de prova proporcionats pel professorat d'Algorísmia, hem pogut fer un parell d'experiments i hem pogut avaluar la rapidesa de cada implementació amb diferents diccionaris. Els primers resultats venen expressats en la següent taula, en milisegons.

La CPU emprada pels experiments és un *Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz*

N = 50	# de paraules	Sorted Vector	Bloom Filter	Ternary S.T.	Double Hashing
Tres Paraules †	*3*	5.0	5.1	0.4	6.0
Mare Balena	*5 051*	62.7	88.1	18.4	120.0
EFF	*7 776*	77.6	106.4	25.6	154.3
Dracula	*9 421*	86.8	125.1^{2E}	31.1	194.0
Quijote	*25 546*	104.3	173.3^{2E}	47.4	215.9
Mieliestronk	*58 108*	170.9	339.4^{3E}	101.6	423.9

Aquests experiments han estat executats tres vegades i s'ha pres la mitjana aritmètica dels tres, en el mateix computador. El diccionari "*Tres Paraules"* és un diccionari que conté només tres paraules i amaga exactament tres paraules.

† S'ha hagut de reduir el nombre de paraules amagades de $P = 20$ a $P = 3$.

Sembla ser que els *Ternary Search Tree* són, amb diferència, l'estructura de dades més òptima; seguits dels vectors ordenats i en últimes posicions els *Bloom Filters* i les taules amb *Double Hashing*. A més a més només es produeixen fins a tres falsos positius pels *filtres de Bloom* (3E) en els diccionaris més llargs, gràcies als càlculs presentats anteriorment. Amb els diccionaris amb menys de ~8000 paraules els filtres no presenten falsos positius.

Seguidament farem experiments amb el diccionari més llarg de tots, el de *Mieliestronk*, canviant aquesta vegada el tamany de la sopa de lletres. Aquesta vegada els resultats seran presentats en *segons.*

Mieliestronk	*N²*	Sorted Vector	Bloom Filter	Ternary S.T.	Double Hashing
N = 20	*400*	0.060	0.217^{1E}	0.068	0.205
N = 50	*2 500*	0.179	0.342^{2E}	0.104	0.405
N = 100	*10 000*	0.658	0.871^{3E}	0.233	1.287
N = 200	*40 000*	3.010	2.841^{6E}	0.751	4.604
N = 500	*250 000*	47.39	50.89^{14E}	4.696	59.26
N = 1 000	*1 000 000*	298.2	289.9^{24E}	17.87	358.0

Evidentment, a més grandària de la *SuperSopa*, més temps requereixen les dierents implementacions. Així i tot, els *Ternary Search Tree* segueixen al capdavant i són els més veloços, amb temps increïbles de 18 segons en una sopa de *1000x1000*. Això no és cap sorpresa, doncs els *Trie* en general són estructures de dades dissenyades especialment per a aquest tipus de cerques amb prefixos. Els que no ho són són els *Bloom Filters* i taules amb *Double Hashing*, doncs no estan dissenyades per a emmagatzemar prefixos.

Sembla a ser, tot i així, que si continuem ampliant *N*, el cost de cerca d'un element als *filtres de Bloom* és menor que el cost de cerca d'un element a un vector ordenat o el cost de cerca d'un element a una taula amb *Double Hashing*, si comencen a haver-hi moltes col·lisions. Per tant, els *Bloom Filters* serien una bona opció si tenim molts elements, no hem de fer recorreguts i si podem admetre falsos positius.

7 Referències

Introduction to Bloom Filter. (2022). Retrieved 13 October 2022, from https://coderscat.com/bloom-filter/

Bloom filter - Wikipedia. (2022). Retrieved 13 October 2022, from https://en.wikipedia.org/wiki/Bloom_filter

Paul E. Black, "ternary search tree", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed. 14 February 2022. (accessed 14 October 2022) Available from: https://www.nist.gov/dads/HTML/ternarySearchTree.html

Introsort - Wikipedia. (2022). Retrieved 14 October 2022, from https://en.wikipedia.org/wiki/Introsort

Hash Functions - York University. (2022). Retrieved 10 October 2022, from http://www.cse.yorku.ca/~oz/hash.html

A Hash Function for Hash Table Lookup. (2022). Retrieved 11 October 2022, from http://www.burtleburtle.net/bob/hash/doobs.html

Bonneau, J. (2022). EFF's New Wordlists for Random Passphrases. Retrieved 5 October 2022, from https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases

MurmurHash - Wikipedia. (2022). Retrieved 15 October 2022, from https://en.wikipedia.org/wiki/MurmurHash

Ternary Search Trees – Dr. Dobb’s Journal. (1998). Retreived 13 October 2022, from

https://drdobbs.com/database/ternary-search-trees/184410528

Ternary search tree – Wikipedia. (2022). Retreived 13 October 2022, form

https://en.wikipedia.org/wiki/Ternary_search_tree

Ternary Search Tree – University of San Francisco. (2011). Retreived 9 October 2022, from https://www.cs.usfca.edu/~galles/visualization/TST.html

Trie – Wikipedia. (2022). Retreived 13 October 2022, from https://en.wikipedia.org/wiki/Trie

Introduction to Tries – Kansas State University. (2022). Retreived 2 October 2022, from https://cis300.cs.ksu.edu/trees/tries/intro/