Super Sopa

Projecte Algorísmia Tardor 2022/23

Grup PROJ.4

Rubén Aciego

Jofre Costa

Mariona Jaramillo

Francesc Pifarré

Contingut

1 Introducció

Aquest document actua com a memòria del projecte de l'assignatura d'algorísmia "Super Sopa". Aquest problema consisteix en, donada una "Super Sopa" de lletres, trobar en el seu interior totes les paraules d'un diccionari donat també. La particularitat que fa que la sopa no es comporti de forma tradicional, sinó estrambòticament, és que les lletres de les paraules en el seu interior no tenen perquè estar totes col·locades en una mateixa direcció.

h	q	m	s
p	**o**	h	n
s	y	**l**	x
p	u	z	**a**

i	**h**	m	s
p	**o**	**l**	n
s	y	r	**a**
p	u	z	w

Figura 1: Sopa tradicional i Super Sopa amb la paraula "hola" marcada

Considerarem doncs una Super Sopa un tauler de mida fixada quadrada de caràcters. Per simplificar, considerarem únicament l'alfabet bàsic (sense ç, ñ o altres lletres similars) en minúscula, ignorant accents, dièresis, o qualsevol altre tipus de caràcter no habitual. No considerarem tampoc la presència de dígits.

Sobre aquests caràcters, considerarem que una paraula del nostre diccionari pertany a una Super Sopa si aquesta es pot formar mitjançant la concatenació de caràcters adjacents, on s'admeten adjacències verticals, horitzontals i diagonals.

L'objectiu del projecte es resoldre aquest programa computacionalment, utilitzant diverses tècniques per emmagatzemar el diccionari i explorar la sopa buscant aquestes paraules. S'analitzarà el cost temporal de les diferents solucions proposades experimentalment mitjançant diferents paràmetres i finalment, es compararan tots aquests resultats per intentar determinar quin dels mètodes explorats resulta més eficient per tractar el problema.

2 Implementacions

Les diferents implementacions desenvolupades per tractar el problema estan sota una estructura de codi comuna pròpia de l'orientació orientada a objectes. Considerem una classe abstracta anomenada *SopaSolver* de la cual heretaran les diferents formes de solucionar el problema: *SortedVec*, *Trie*, *Bloom* i *HashMap*. Cada una d'aquestes classes modificaran únicament sobre la classe mare la forma en què es crea i s'emmagatzema el diccionari i com es busquen posteriorment les paraules en la sopa. Per fer-ho, és possible que les classes filles declarin mètodes propis o paràmetres diferents entre elles i no heretats de la classe original.
Els mètodes definits en *SopaSolver* que heretaran la resta de *solvers* tenen la finalitat d'inicialitzar una sopa omplint-la de caràcters aleatoris i incorporant-hi, si escau, alguna de les paraules del diccionari. Aquesta distribució aleatòria dels caràcters s'efectua mitjançant el següent algorisme voraç:
Donada una paraula que volem col·locar dins la sopa

1. Triem una posició aleatòria de la sopa
2. Si la posició no és buida, tornem a 1
3. Prenem la següent lletra de la paraula i la posem en la posició actual. Si la paraula queda buida hem acabat
4. Triem una de les 8 direccions aleatòriament
5. Mentre la direcció no sigui vàlida (o bé estigui fora de la sopa o no estigui buida) triem la següent
6. Si no trobem cap direcció vàlida, desfem els canvis introduïts i tornem a 1 (això ho fem un nombre finit de cops)
7. Tornem a 3

És possible que l'algorisme no sigui capaç de col·locar totes les paraules demanades dins la sopa (dependrà de la mida triada, de com s'han col·locat les anteriors paraules i de l'atzar) però per l'ús que en farem és suficient.
Utilitzem també, per executar el programa i experimentar amb les diferents solucions, un arxiu *main.cpp* que crea un *solver* que funcionarà amb el mètode indicat per després provar el seu funcionament sobre una sopa generada de forma aleatoria; mesurant, durant aquest procés, diversos indicadors de rendiment que compararem posteriorment.

2.1 Vector ordenat

Per la implementació del *SortedVecSolver* hem utilitzat, com indica el seu nom, un vector ordenat per emmagatzemar les paraules del diccionari. Aquestes poden ser donades en un ordre qualsevol i s'ordenen en el moment de creació del solver i emmagatzemen en un vector.
En la inicialització d'aquesta estructura de dades, es copien en un vector tots els mots del diccionari que son proporcionats en forma de llista, podent arribar aquests en un ordre qualsevol. Aprofitem que hem de recórrer la llista per emmagatzemar la máxima llargada de totes les paraules de la llista.
Un cop inicialitzat el vector, l'ordenem utilitzant la funció *sort* per defecte de la llibreria *algorithm* de C++. Aquesta funció es comporta, segons la documentació de C++, de forma quasi lineal O(n*log(n)) en mitjana (i cas pitjor) sobre el nombre de mots del diccionari.
Un cop inicialitzat el vector, podem utilitzar-lo per atacar el problema d'ubicar, en una Super Sopa qualsevol, tots els mots que estiguin en el diccionari. Per fer-ho, haurem de començar a cercar en la sopa a partir de cada posició de la quadrícula.
Un cop fixada una casella d'inici, preparem l'estructura amb la qual farem la cerca, que serà, essencialment, un recorregut en profunditat implementat de forma recursiva. A cada crida d'aquesta cerca disposarem de la següent informació: posició dins de la quadrícula de la Super Sopa (en forma de posició *i* referint-se a la fila que ocupa i *j* referint-se a la columna que ocupa), el seguit de caràcters que hem acumulat en el camí per arribar a aquesta posició (com a un string *word*), el nombre de passos en el camí (o nombre d'iteracions, que coincidirà per construcció amb la llargada del mot *word*), la matriu que guarda quines posicions de la sopa hem visitat ja, el conjunt de paraules que hem trobat en el diccionari i dos paràmetres *left* i *right* que explicarem posteriorment per a que utilitzem.
Per cada un d'aquests nodes, els tractem marcant-los com a visitats en la matriu que utilitzem com a referència per a fer la cerca (que és correcte degut a que se'ns diu que una paraula no es pot solapar amb ella mateixa per trobar-se present a la Super Sopa) i afegim a *word* el caràcter de la posició que estem visitant.
Cal, un cop afegit aquest caràcter, comprovar si aquesta paraula provisional *word* és realment una paraula del diccionari, i, en cas de que ho sigui, afegir-la al conjunt de paraules trobades en la sopa. Per comprovar-ho, ho fem mitjançant una cerca binària sobre el vector ordenar que conté tots els mots del diccionari. Aquesta cerca tindria complexitat logarítmica sobre la mida del diccionari.

Per intentar millorar aquesta complexitat, però, utilitzem els paràmetres *left* i *right* que actualitzem un cop visitem cada node. El paràmetre *left* correspón a la posició de la primera paraula del vector ordenat que és més gran o igual que la *word* que portem construïda fins aquell moment. El paràmetre *right* correspon, en canvi, a la paraula anterior a la primera paraula del vector que és estrictament més gran que *word*. Aquests paràmetres delimiten, per construcció, la zona del vector on, de ser-hi, es trobaria la paraula que hi estem cercant. En particular, el subvector *diccionari[left:right]* conté ordenades totes les paraules que tenen *word* com a prefix.

A cada pas doncs actualitzarem aquest subvector mitjançant crides a *std::lower_bound* per trobar les posicions desitjades. Observem també que podem realitzar una petita optimització, doncs per inducció (o precondició) tenim que al principi d'una nova iteració a *diccionari[left:right]* només tenim prefixes de *word[0:-1]* (word menys l'últim caràcter) i per tant a l'hora de buscar els nous valors només ens cal realitzar comparacions sobre aquest últim caràcter.

Un cop tenim calculats els valors de actualitzats de *left* i *right*, la paraula pertany al diccionari si i només si *word* = *diccionari[left]* doncs és la paraula més petita amb prefix *word*. Si obtenim *left* > *right* aleshores no hi ha cap paraula amb aquest prefix i podem aturar el recorregut.

Un cop buscada la paraula en el diccionari, trobada o no, cal decidir si seguir buscant entre els veïns de la posició actual. Ho decidim mitjançant diversos criteris: el veí existeix (és una posició vàlida a la quadrícula), no l'hem visitat ja en el recorregut actual (ja que una paraula no es pot solapar amb si mateixa), no portem més iteracions que la llargada de la paraula més llarga del diccionari (ja que de seguir mai no trobaríem una paraula que estigués al diccionari) i que la zona del vector on estem buscant contingui com a mínim un element (entés com existeixi algun element entre *left* i *right*).

Fent tot aquesta cerca començant per cada una de les posicions de la quadrícula, la recorrerem totalment i l'algoritme trobarà totes les paraules del diccionari que estiguin presents a la Super Sopa.

2.2 Trie

Existeixen moltíssimes tries diferents, i totes elles tenen característiques pròpies i els seus avantatges i inconvenients. Algunes d'elles són la trie bàsica, la compressed trie o radix tree, Patricia, suffix tree… Però nosaltres ens hem decantat per usar un Ternary Search Tree, que es tracta d'un arbre de cerca que emmagatzema un símbol i que té 3 descendents. Quan es compara un símbol amb el del node, si aquest és més petit la cerca continua cap al fill esquerra, si és igual cap el del mig i si és més gran cap al de la dreta.

El perquè de la tria de la TST en comptes d'algun altre tipus de trie és el següent: respecte la trie bàsica, el TST és més eficient en tema d'espai: cada node d'una trie estàndard ha de contenir un contenidor de la mida de l'abecedari per accedir als possibles fills, en canvi el TST només requereix de 3 fills. Passa quelcom similar amb la compressed trie: aquesta és més eficient en temes d'espai que la trie estàndard, però en general sol ser una mica pitjor que el TST (passa el mateix per variants de la compressed trie com el suffix tree). Pel que fa al temps d'execució, en general els TST són més lents que les tries o les compressed tries per fer cerques: el cost de cercar una paraula en un TST és logarítmic, i en canvi en les tries o compressed tries és proporcional a la llargada de la paraula que es busca. Però pel problema de la Super Sopa es pot aconseguir que les cerques també siguin proporcionals a la llargada de la paraula en el TST: el truc és simplement buscar lletra per lletra, i a cada pas quedar-se en el node on ho havíem deixat. Com que tenim un diccionari fix (de mida l = 26), si el TST està balancejat per a cada lletra visitem de mitjana log(26) nodes (i en general seran menys). Si això ho fem per a cada lletra de la paraula que busquem, el cost ens surt O(log(26)*k) = O(k) (on k és la longitud de la paraula), i per tant és proporcional a la mida de la paraula. Aquest justificació de cost és similar a l'estructura Patricia, ja que s'haurien de passar les paraules a cadenes de bits.

Ja per acabar, hem vist i s'ha justificat que tant en temes d'espai com en temps de cerques els TST són un bon tipus de trie en comparació de les altres possibles. A això cal afegir-hi també que els TST tenen una implementació més concisa, natural i lògica que molts dels altres tipus de tries mencionats.

Un cop justificat l'ús dels TST, n'explicarem la implementació feta. Per inicialitzar aquesta estructura de dades, es copien en un vector tots els mots del diccionari que són proporcionats en forma de llista, que no té perquè està ordenada. Aprofitem que hem de recórrer la llista per emmagatzemar la máxima llargada de totes les paraules de la llista. Un cop tenim aquest vector, s'ordenen en temps O(n*log(n)). Per què fa falta passar les paraules a un vector i

ordenar-lo? Doncs perquè gràcies a l'ordenació podem aconseguir que estigui el més balancejat possible: només cal crear una funció recursiva que inserti la mediana (element central) i que llavors inserti recursivament els elements més petits i més grans que aquesta.

La funció d'inserció s'assembla a la funció d'inserció en un Binary Search Tree, però hi ha alguna diferència important. Es tracta d'una funció recursiva a la que se li passa una string, un índex i un punter "doble" (punter que apunta a un altre punter) a un node de l'arbre. Si el punter apuntat pel punter doble és null, se'n crea un de nou amb la lletra situada a la posició de la paraula apuntada per l'índex. Llavors, es compara de nou aquesta lletra amb la del node que ens passen (que ens hem assegurat que ja no serà null). Si és més petita o més gran, es fa una crida recursiva al fill esquerre i al fill dret, respectivament (sense canviar els paràmetres). Si és igual, en general es fa la crida recursiva al fill del mig augmentant l'índex en 1 (i així passem a buscar la següent lletra), a menys que ens trobem a l'últim símbol de la paraula. Si això passa, marquem el node com a final de paraula i no fem cap més crida perquè ja hem acabat.

Un cop hem inicialitzat el TST i insertat tots els seus elements, cal procedir a buscar totes les paraules de la Super Sopa que estiguin al diccionari. La manera de fer-ho és molt semblant a l'explicada pel vector ordenat, i bàsicament consisteix en fer una cerca en profunditat a mode de backtracking des de cada cel·la de la sopa, per tal de trobar tots els possibles camins que comencen des d'aquella posició.

A cada crida d'aquesta cerca sabrem (mitjançant els arguments) la posició en què ens trobem, les posicions ja visitades de la sopa, el node del TST en què estem treballant, un string contenint els caràcters per arribar a aquesta posició i la longitud d'aquesta. Hi ha dues maneres bàsiques per tallar una cerca a un punt: si la posició on ens trobem ja ha sigut visitada anteriorment o la paraula que portem acumulada ja té la mida màxima. També es pot tallar però la cerca d'una altra manera, i que de fet és la que dona millors resultats i fa que el TST funcioni tan bé per resoldre aquest problema. El que es fa és, des del node del TST que ens han passat, començar una cerca al TST per trobar un node que contingui el símbol de la posició actual, i si no el trobem es talla la cerca. Tal com s'ha explicat prèviament, aquesta d'un cert símbol al TST té de mitjana cost $O(\log(26)) = O(1)$ (si tenim un TST totalment ple i balancejat, normalment és menys), i nosaltres la fem iterativament: mentre el node on ens trobem no sigui null, comparem el caràcter de la sopa amb el node actual. Si aquest últim és més petit passem al fill esquerra, si és més gran al fill dret, i si és igual retornem un punter al

node on ens trobem.

En cas que cap dels 3 mecanismes de poda s'hagi complert, es marca per fi la posició actual com a visitada, s'afegeix la lletra corresponent a la string d'acumulació i, en el cas que el nou node actual sigui final de paraula, s'afegeix aquesta al conjunt de paraules trobades.

Fet això, la cerca dins la sopa l'algorisme continua el seu recorregut per totes les 8 posicions adjacents (de forma recursiva en la nostra implementació, sempre i quan aquestes posicions siguin vàlides). Cal mencionar que quan es fa la crida recursiva s'augmenta el valor de la longitud de la string d'acumulació en 1 i se salta al fill del mig del node actual (ja que ja sabem que el seu símbol coincideix amb el de la posició de la qual marxem). Un cop acabades les crides recursives de les posicions veïnes, es completa l'esquema de backtracking desmarcant la posició actual com a visitada i s'esborra el símbol prèviament afegit a la paraula d'acumulació.

2.3 Filtre de Bloom

Abans de detallar la implementació mitjançant el filtre de Bloom, farem primer una breu explicació del funcionament d'aquesta estructura de dades. Un filtre de bloom és una estructura de dades probabilística que té com a principal objectiu comprovar la pertinença d'un determinat valor en un conjunt (potencialment gran) de manera eficient tant en memòria com en temps.

L'estructura funciona de la forma següent, es trien k funcions de hash independents (per reduir col·lisions) i construïm un vector de m bits inicialment tots a 0. Ara per cada paraula que es vol afegir al conjunt apliquem les k funcions de hash i posem el bit corresponent a la sortida de cada funció (mòdul m) a 1. Per comprovar la pertinença d'una paraula qualsevol, li apliquem les k funcions de hash i comprovem que els k bits valguin 1.

En la nostra implementació, les funcions de hash s'aplicaran sobre un rolling hash que es calcula incrementalment quan es visita una cel·la. D'aquesta manera cada cop que volem calcular un hash per l'*string* actual no cal recórrer-lo sencer de nou. Per tant les k funcions de hash s'aplicaran sobre aquest hash previ, les construïm de la següent forma:

$H_k(x) = \sum_{i=0}^{k-1} a_i^k x^i \mod p$ prenent p un nombre primer i els coeficients a_i^k de manera aleatòria a [0, p-1] imposant que per i > 0 algún d'ells sigui no nul.

L'avaluació d'aquests polinomis es pot fer de forma eficient mitjançant la regla de Horner, per tant cada càlcul d'una funció de hash té cost $\Theta(k)$ i aplicar les k funcions de hash té cost $\Theta(k^2)$. Per temes d'eficiència, també hem realitzat una implementació utilitzant funcions de hash *MurmurHash3*, l'aplicació de cadascuna és constant i per tant aplicar-les totes té cost $\Theta(k)$.

Crearem doncs dos filtres de Bloom diferents, un filtre construït amb les paraules del diccionari i un segon filtre que construïm amb tots els prefixos de cada paraula del diccionari. Un cop tenim inicialitzades les estructures de dades com s'ha explicat, realitzem un recorregut dins la sopa de lletres començant en cada casella d'aquesta. En cada punt podem tallar el recorregut si es compleix algun dels casos següents:

- La casella ja s'ha visitat en la construcció de la paraula actual

- La paraula que estem construint és més llarga que la paraula de màxima longitud dins el diccionari
- El filtre de Bloom de prefixes avalua a fals

Quan arribem a una nova casella

1. La marquem com a visitada
2. Actualitzem el rolling hash de la paraula
3. Afegim la lletra de la casella a la paraula que anem construint

Ara comprovem si la paraula actual pertany a l'estructura del filtre de Bloom aplicant les k funcions de hash, si és el cas l'afegim al conjunt de paraules trobades.

Seguidament l'algorisme continua el seu recorregut per totes les 8 caselles adjacents (de forma recursiva en la nostra implementació). Un cop acaba es desmarca la casella actual i s'esborra la lletra afegida a la paraula que anem construint.

Observem que la solució plantejada mitjançant un filtre de Bloom té com avantatge que comprovar la pertinença de cada paraula al diccionari té un cost fixat $\Theta(k^2)$ o $\Theta(k)$ depenent de les funcions de hash. No obstant, aquesta constant és alta comparada amb les altres estructures de dades utilitzades.

Com ja hem comentat, l'estructura de dades és probabilística i per tant és possible obtenir falsos positius, fet que no succeeix amb les altres opcions plantejades. La probabilitat d'obtenir falsos positius es pot ajustar però mitjançant la mida del vector m i la quantitat de funcions de hash k. Això també planteja un *trade-off* interessant doncs a mesura que obtenim més falsos positius en el filtre de Bloom de prefixos més recorreguts inútils farem.

Un fet clau aleshores són les tries de m i k, la mida del vector de bits i la quantitat de funcions de hash. Podem realitzar les tries per obtenir un millor rendiment sacrificant memòria prenent un vector de bits molt gran i una k petita, de manera que el vector quedi molt dispers.
No obstant, el filtre de Bloom està pensat principalment per estalviar memòria doncs un dels seus avantatges és que no cal guardar els valors que s'afegeixen. D'aquesta manera, fixades m i n (on n quantitat d'elements que volem introduir al vector) és $k = \frac{m}{n} ln(2)$,

donada la probabilitat p de col·lisió desitjada, el nombre òptim de bits és $m = -\frac{nln(p)}{(ln(2))^2}$ i per tant, obtenim $k = -\frac{ln(p)}{ln(2)}$. Amb aquests valors triats, observem que fixada una probabilitat p, el nombre de funcions de hash k és constant i per tant no depèn de la mida del diccionari.

2.4 Taula de Hash

La darrera implementació demanada per al diccionari és una taula de hash amb double hashing.

El hashing és una tècnica per poder fer *mapping* de claus i valors en una taula de hash mitjançant una funció de hash. L'eficiència del hashing dependrà d'aquesta funció de hash, que s'encarrega d'assignar a cada clau, una posició determinada en la taula de hash. És a dir, la funció de hash converteix la seva entrada, que pot tenir un nombre infinit d'elements, en una sortida amb un nombre finit d'elements i aquest nombre finit d'elements serà la mida de la taula de hash. Si suposem que cada clau dels valors que considerem s'associarà a una entrada de la taula de hash diferent, aleshores la cerca de valors en la taula de hash tindria temps constant O(1). No obstant, és molt fàcil que això no succeeixi i hi hagi col·lisions, és a dir, que la funció de hash assigni la mateixa entrada de la taula de hash a dues claus diferents. Per exemple, si considerem la funció de hash Hash1(clau) = clau%13 i les claus 5,18,4,13,9, ens resultaria la següent taula de hash (suposant que l'hem implementat amb un vector):

13				4	5				9			

però la clau 18, no l'hem pogut ubicar en la taula ja que Hash1(18) = 18%13 = 5, posició que ja està ocupada. Ens trobem doncs en el cas d'una col·lisió.
Per resoldre les col·lisions existeixen molts mètodes diferents, però nosaltres ens centrarem en l'opció demanada: double hashing.

El doble hashing consisteix en utilitzar una nova funció de hash Hash2, que s'apliqui sobre la clau quan hi ha una col·lisió i així successivament fins que la col·lisió es resolgui. A més, un dels avantatges d'utilitzar doble hashing per resoldre col·lisions és que no produeix clústers, és a dir, distribueix els valors d'una manera uniforme en cas de col·lisions.

Aquesta funció Hash2 ha de verificar que mai evalua a 0, perquè sinó, en cas d'haver-hi una col·lisió, faria *mapping* permanent de la clau a la mateixa posició, ja que al evaluar a 0, no canviaria de posició; i també ha de verificar que consultarà totes les posicions de la taula en cas que tot fossin col·lisions.
Una forma prou estandaritzada de fer doble hashing és mitjançant la fórmula:

$(Hash1(clau) + i * Hash2(clau)) \% midaTaulaHash$

on Hash1(clau) és una funció de hash

on Hash2(clau) = majorPrimer - (clau % majorPrimer);

on majorPrimer és el primer més gran dins l'interval [2, midaTaulaHash)

on i va de 0 al nombre de col·lisions que tingui la clau.

No obstant, com que en el nostre cas, les claus són strings, no podem fer com en l'exemple vist, on les claus i els valors eren el mateix (nombres naturals). Per tant, aplicarem el doble hashing sobre un rolling hash, com hem vist en la implementació del filtre de Bloom, el qual es calcularà incrementalment quan es visita una cel·la, aprofitant el hash anterior i reduint temps. Utilitzarem una funció de rolling hash polinòmica que assigna a cada string s un nombre natural, que serà la nostra clau:

$hash(s) = s[n-1] + s[n-2] * b + \ldots + s[0] * b^{n-1} \, mod \, p$

on cada posició de s visitada es traduirà a la seva representació ASCII (nombre natural), p és un primer gran i b pot ser qualsevol enter positiu, en el nostre cas hem triat b = 31 perquè és un nombre primer més gran que 27 (que és la mida de l'abecedari) i per tant evitem col·lisions. A més, per fer aquesta evaluació polinòmica de forma incremental, apliquem la regla de Horner, com s'explica en l'apartat del filtre de Bloom.

En el rolling hash de la segona funció de hash hem utilitzat el següent primer més gran que 31, és a dir, 37, perquè d'aquesta obtenim valors de la funció de hash diferents i, en cas que hi hagi dues o més col·lisions en la mateixa entrada de la taula, fem que la probabilitat que tornin a col·lisionar a l'aplicar el doble hashing sigui més baixa.

Per tant, en el nostre cas tindrem:

Hash1(s) = rollingHash1(s)

Hash2(s) = majorPrimer - (rollingHash2(s) % majorPrimer)

Per buscar paraules en la taula de hash, calculem la seva clau i li apliquem la primera funció de rolling hash per mirar on hauria d'estar ubicada en la taula. Si aquella posició de la taula està buida, significa que la paraula no està en el diccionari; si conté la paraula buscada, ja l'hem trobat i hem acabat; si conté una paraula però no és la buscada, apliquem la segona funció de hash i repetim el procés fins a trobar la paraula buscada o fins tornar a la primera

posició que havíem consultat, en aquest cas, hem consultat totes les entrades de la taula, significant que la paraula buscada no està en el diccionari.

Un altre factor a decidir en la implementació de la taula de hash és la seva mida (en relació a la quantitat de valors que volem inserir). Si està un 75% plena, ja ofereix bons resultats, per tant, una tria raonable és que la taula sigui 2 cops la mida del diccionari, la qual cosa no és excessiva i satisfà la "regla" del 75%.

Un cop tenim inicialitzada la taula amb les paraules del diccionari triat, fem un recorregut dins la sopa de lletres per cada casella que conté. Per optimitzar els recorreguts, els retallem en els mateixos casos que el filtre de bloom, si la casella ja s'ha visitat en la construcció de la paraula actual i si la paraula en construcció excedeix la longitud de la paraula més llarga del diccionari.

En el cas d'aquesta estructura de dades, no podem optimitzar molt més els recorreguts de forma natural (com en la trie), per tant, hem decidit crear una segona taula de hash que contingui tots els prefixos de cada paraula del diccionari. Tot i que és veritat que a nivell de memòria és pitjor, a nivell computacional es guanya molt (permet que acabin en temps raonable sopes que de l'altra manera no acabarien). Per tant, cada cop que estem construint una paraula, consultem si està en la taula de prefixos: en cas que hi sigui, seguim com abans, però en cas que no, podem acabar amb el recorregut d'aquesta paraula, fent que s'hagin de fer moltes menys cerques innecessàries en la taula de hash del diccionari.

El cost de cada iteració de l'algorisme és O(1) per calcular les funcions de hash (doncs es calculen incrementalment) i el cost de realitzar dues cerques a les taules de hash. Una cerca dins la taula de hash té cost O(m*ncol) on m és la mida de la paraula (doncs s'ha de realitzar una comparació en cas de trobar una coincidència) i ncol és el nombre esperat de col·lisions.

La resta de l'algoritme és idèntic al del filtre de Bloom, per tant, no es torna a detallar.

3 Anàlisi de la complexitat del problema

En aquesta secció descriurem la complexitat del problema per a cada tipus de diccionari usat. No només estudiarem la complexitat asimptòtica sinó que també es tindran en compte les constants, ja que també seran importants per poder analitzar correctament els resultats de la experimentació. Abans de començar l'anàlisi, direm **n** a la quantitat de files (i columnes) de la Super Sopar, **m** la mida del diccionari, **t** la mida de l'alfabet, **k** al nombre de funcions de hash independents pel filtre de Bloom i **L** a la llargada de la paraula més llarga del diccionari que es considerem.

Per les 4 implementacions estem resolent el problema de la següent manera: per a cada casella de la taula (n'hi ha $\mathbf{n}^2$), estem comprovant primer que si la paraula acumulada fins al moment es troba al diccionari, i després fem una crida a les 8 posicions (menys si es tracta d'una posició marginal) adjacents a l'actual per la primera lletra d'una paraula, i 7 per les següents (ja que sabem segur que una no l'hem de visitar). Per tant, la complexitat serà el nombre de cel·les de la sopa multiplicat pel nombre d'operacions que fem des de cada cel·la. Observem que aquest nombre d'operacions, com a màxim, serà el nombre de cel·les adjacents multiplicat pel nombre d'operacions necessàries a una cel·la, i tot això elevat a la potència **L**-èssima, perquè com a màxim des d'una certa posició arribarem a fer paraules de llargada **L**, i a cada nova cel·la que visitem com a màxim totes les cel·les adjacents. Llavors, la complexitat en el cas pitjor per les implementacions fetes són les següents:

- Vector ordenat: $O(\mathbf{n}^2 \cdot 8 \cdot 7^{\mathbf{L}-1} \cdot \log(\mathbf{m})^{\mathbf{L}})$
- Trie: $O(\mathbf{n}^2 \cdot 8 \cdot 7^{\mathbf{L}-1} \cdot \log(\mathbf{t})^{\mathbf{L}})$ (Considerant que està balancejada)
- Taula de Hash: $O(\mathbf{n}^2 \cdot 8 \cdot 7^{\mathbf{L}-1} \cdot C^{\mathbf{L}})$ (Considerant que la constant C és el cost de fer la cerca a la taula).
- Filtre de Bloom: $O(\mathbf{n}^2 \cdot 8 \cdot 7^{\mathbf{L}-1} \cdot \mathbf{k}^{\mathbf{L}})$ (Considerant que usem el *MurmurHash3*, sinó se substitueix la **k** per $\mathbf{k}^2$).

Però que succeeixi el cas pitjor és extremadament estrany, ja que hauríem de tenir una sopa de lletres on totes les posicions tinguessin la mateixa lletra i les paraules del diccionari foren totes de llargada **L** i formades únicament per aquesta lletra. És per això que és necessari i paga la pena fer l'anàlisi del cas mitjà, ja que serà més realista i s'aproparà més als casos aleatoritzats de la implementació. Farem primer l'anàlisi del cas mitjà per la Trie (TST), i llavors les complexitats de les altres implementacions en seran un "corol·lari".

Per simplificar una mica els càlculs, considerarem que cada lletra surt amb la mateixa probabilitat a cada posició de la Sopa ($1/\mathbf{t}$) i que de cada node de la Trie (TST) en poden

penjar des de 1 fins a **t** nodes diferents, tots amb la mateixa probabilitat (distribució uniforme). A més, suposem que la Trie està balancejada i que les paraules del diccionari estan ben "repartides" i uniformes al llarg l'alfabet. Amb aquestes suposicions, primer de tot calculem la probabilitat de, donada una lletra i trobant-nos a un cert node del TST, passar a algun node descendent (és a dir que algun dels descendents tingui com a valor la lletra donada). Com que tenim la mateixa probabilitat de tenir de 1 fins a **t** nodes descendents, la probabilitat de passar a un descendent serà la mitjana de les probabilitats de passar amb X nodes descendents (amb X = 1 … **t**):

$q = \frac{1}{t}\sum_{i=1}^{t}\frac{i}{t} = \frac{1}{t^2}\sum_{i=1}^{t} i = \frac{(t+1)t}{2t^2} = \frac{t+1}{2t}$ (recordem que **t** és la mida de l'alfabet)

Llavors considerem la variable aleatòria $X_{i,j}$ que des d'una posició (i,j) ens dona la llargada del camí que podem seguir a la sopa començant des d'aquella posició. Degut a la distribució uniforme de les lletres, podem considerar que aquesta variable aleatòria serà la mateixa per a qualsevol posició, és a dir, no dependrà de (i,j) (per posicions marginals hauria de ser menor, però negligirem aquest fet ja que assimptòticament acabarà essent el mateix). A partir d'ara doncs, denotarem $X_{i,j}$ per X. Observem que Im(X) = {0 … **L**}. La probabilitat que X prengui un valor **Y** ∈ Im(X) és $P(X = \mathbf{Y}) = q^{\mathbf{Y}}$, i la justificació d'això és la següent: ja hem vist que passar d'un cert node al següent encertat tenim una probabilitat de q. A més observem que passar d'un cert node **r** del TST al seu descendent i passar d'un node **s** al seu descendent són successos totalment independents, ja que el resultat que s'obtingui al node **r** no dependrà del resultat obtingut a **s**, i viceversa. Per tant, si hem de passar al node descendent **Y** vegades amb probabilitat q i cada passa és independent de les altres, llavors la probabilitat de totes passar serà $q^{\mathbf{Y}}$. Ara, l'únic que ens queda per calcular és el nombre esperat de passos des de (i,j) i ja ho tindrem tot. Per la definició d'esperança en variables aleatòries discretes, l'esperança de X és:

$$E[X] = \sum_{i=0}^{L} i \cdot P(X = i) = \sum_{i=1}^{L} i \cdot q^{i}$$

Per fitar aquesta suma, considerem la funció $f(z) = \frac{1}{1-z}$, que equival a la sèrie de potències $\sum_{n\geq 0} z^n$ per z's amb mòdul < 1 (Aquesta igualtat és certa ja que z < 1 la sèrie convergeix uniformament i *f(z)* és una funció holomorfa). Com que $q < 1$, la funció està ben definida en aquest punt. A més, la derivada també existeix i és holomorfa per z < 1, i obtenim

l'expressió $f'(z) = \frac{1}{(1-z)^2}$. Com que és holomorfa, és igual a la derivada de la sèrie de potències anterior (derivant terme a terme): $\sum_{n\geq 1} n \cdot z^n$. Dit això, tornem a l'expressió de l'esperança:

$E[X] \leq \sum_{n\geq 1} i \cdot q^i = f'(q) = \frac{1}{(1-q)^2} = \frac{4t^2}{(t-1)^2} \leq 5$ (per **t** > 9, com és el nostre cas)

Per tant, de mitjana la Trie farà com a molt 5 passos, en comptes dels **L** que havíem contemplat abans, i per tant la seva complexitat serà O(**n**^{2}·8·7^4·log(**t**)^5) (recordem que **t** = 26 en el nostre problema). Aquest valor segueix essent una fita superior, ja que hi ha moltes superposicions de camins que estem comptant i que a la realitat no passen, que els diccionaris que tenim no estan tan plens com hem suposat per fer els càlculs… Però és una fita molt millor que la donada pel cas pitjor.

Pel que fa al Filtre de Bloom i la Taula de Hash, com que s'hi han afegit les prefixos de les paraules el procés de tall de camins és el mateix que el de la Trie (amb costos diferents, evidentment). Per tant, el cost de la Taula de Hash serà O(**n**^{2}·8·7^4·C^5), i el del Filtre de Bloom serà O(**n**^{2}·8·7^4·**k**^{5}) (substituir **k** per **k**^{2} si no usem el *MurmurHash3*). En el cas del filtre de Bloom però, és possible que el nombre esperat de passos sigui major degut als falsos positius a l'hora de comprovar els prefixes.

Finalment, el Vector Ordenat té un cost superior ja que ha de fer cerques binàries. Per tant, la seva complexitat és O(**n**^{2}·8·7^4·log(**m**)^4).

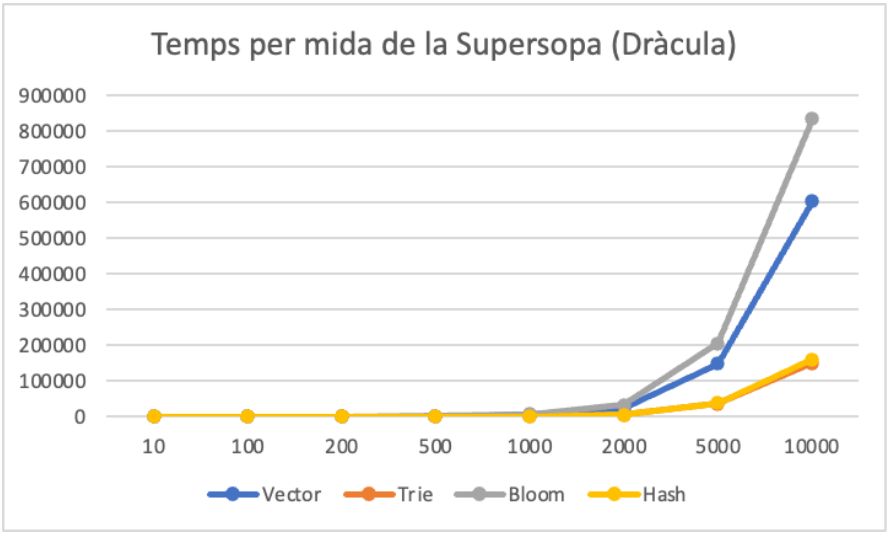
Cal esmentar que es podria fitar una mica millor la constant del Vector Ordenat amb la suposició de tenir un diccionari "ben repartit" i uniforme. A cada passa per una casella estaríem reduïnt la mida del vector a considerar en 1/**t**. Si considerem que hi ha la mateixa probabilitat de fer una cerca binària a qualsevol subinterval del vector, llavors el cost seria la mitjana de tenir-lo tot complet i anar-lo reduïnt cada pas per un factor de 26 fins arribara 1. D'aquesta manera estaríem acotant una mica millor el cost del Vector Ordenat, però el canvi no és molt substancial.

4 Experimentació

Per a l'experimentació, hem utilitzat tots els mètodes implementats per resoldre Super Sopes de diferents mides: 10, 100, 200, 500, 1000, 2000, 5000 i 10000. Per cada una d'aquestes mides, hem introduït 20 paraules per cada un dels diccionaris d'exemple (Dràcula, Balena i Quijote) i ho hem resolt 5 cops, per considerar com a temps que tarda en resoldre una Super Sopa d'aquesta mida la mitjana dels temps que ha tardat en aquestes 5 iteracions. Aquest procés, a més, s'ha efectuat per duplicat, canviant els paràmetres del filtre de Bloom per observar també com aquests afecten al seu rendiment.

El temps s'ha mesurat sempre en **milisegons**, i es mostrarà en aquestes unitats en l'eix OY de totes les gràfiques exposades en aquest informe. L'eix OX de totes les gràfiques correspon a la mida de la Super Sopa, mesurada com el nombre de files que té.

Un anàlisis dels resultats ens dona les següents gràfiques, inicialment amb els paràmetres del filtre de Bloom calculats de forma que la probabilitat d'obtenir col·lisions sigui de 10^{-7}, de manera que k = 23 la quantitat de funcions de hash i utilitzem la funció de hash *Murmur3Hash*.



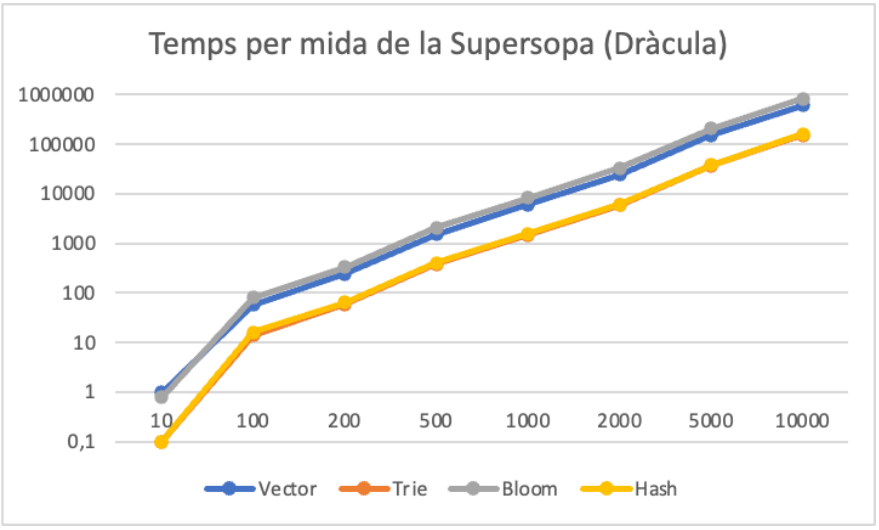
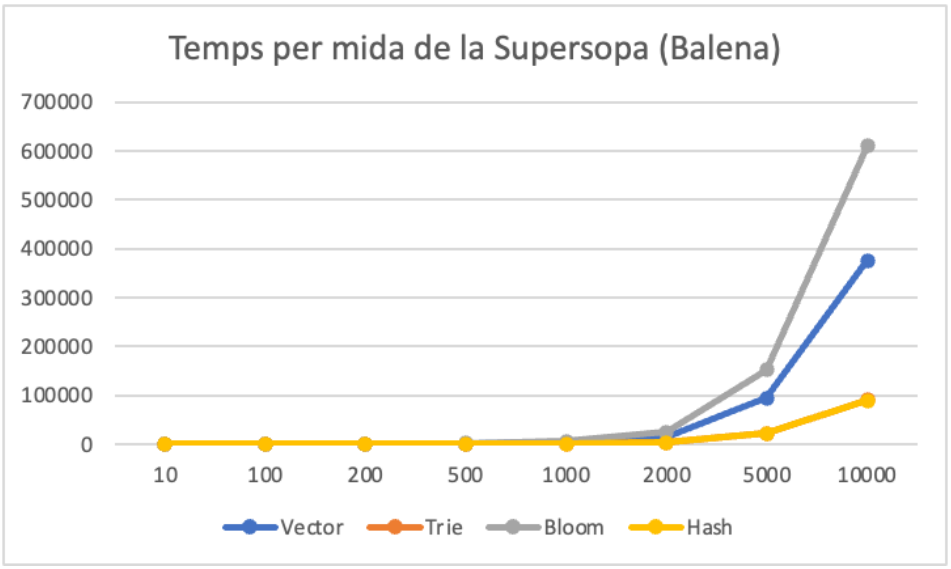


Figura 2: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Dràcula, en escala lineal i logarítmica.



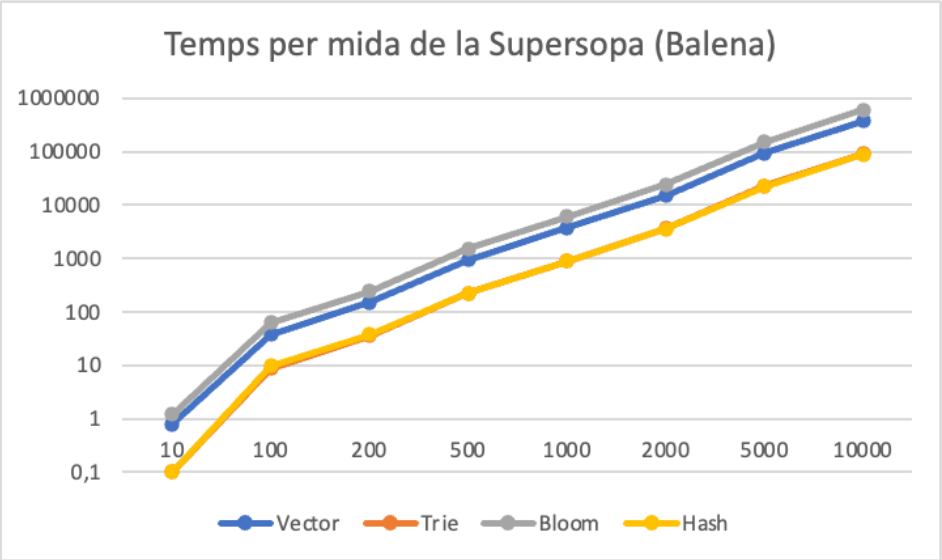


Figura 3: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Balena, en escala lineal i logarítmica.

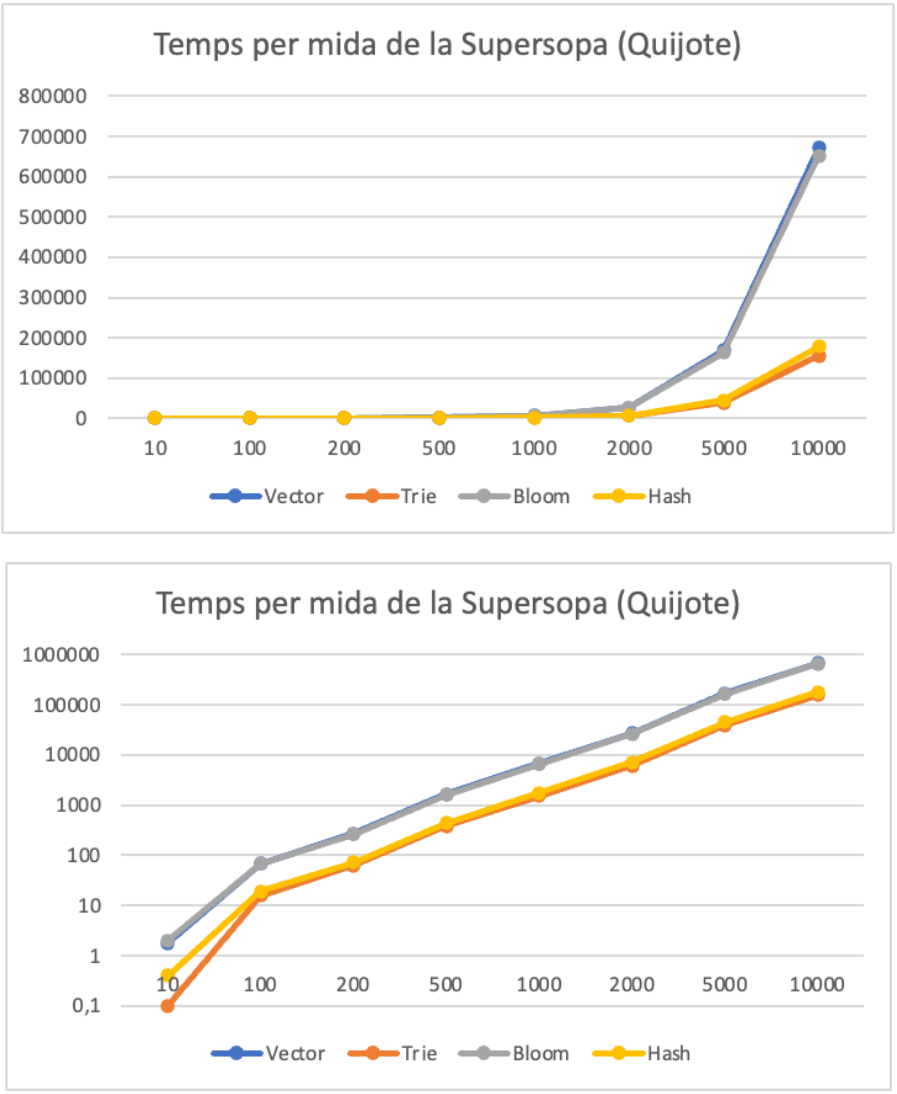


Figura 4: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Quijote, en escala lineal i logarítmica.

Podem observar que hi ha dos grups diferenciats: la Trie i la Taula de Hash, i el Vector i el Filtre de Bloom. Es pot observar que la Trie i la Taula tenen un comportament similar, essent la primera marginalment més ràpida. Això es deu que la constant per fer les cerques a la Taula de Hash és superior a la de la Trie, cosa que és més visible a mesura que la mida de la Sopa creix. Pel que fa al Vector i al Filtre de Bloom, en teoria el Filtre de Bloom és asimptòticament millor que el Vector, però en els casos mesurats la constant introduïda en fer els càlculs progressius de l'algorisme fa que sigui pràcticament igual i a vegades pitjor que el Vector en termes de temps (com a mínim amb les mides dels nostres experiments).

En el cas del filtre de Bloom, observem falsos positius en 10 ocasions, de les quals en una obtenim fins a 2 falsos positius i a la resta només un.

Efectuant el mateix anàlisis de nou mesurant, enlloc del temps que es tarda en resoldre la Super Sopa, el nombre de posicions de la sopa que es visiten. En aquest cas, els resultats son els següents:

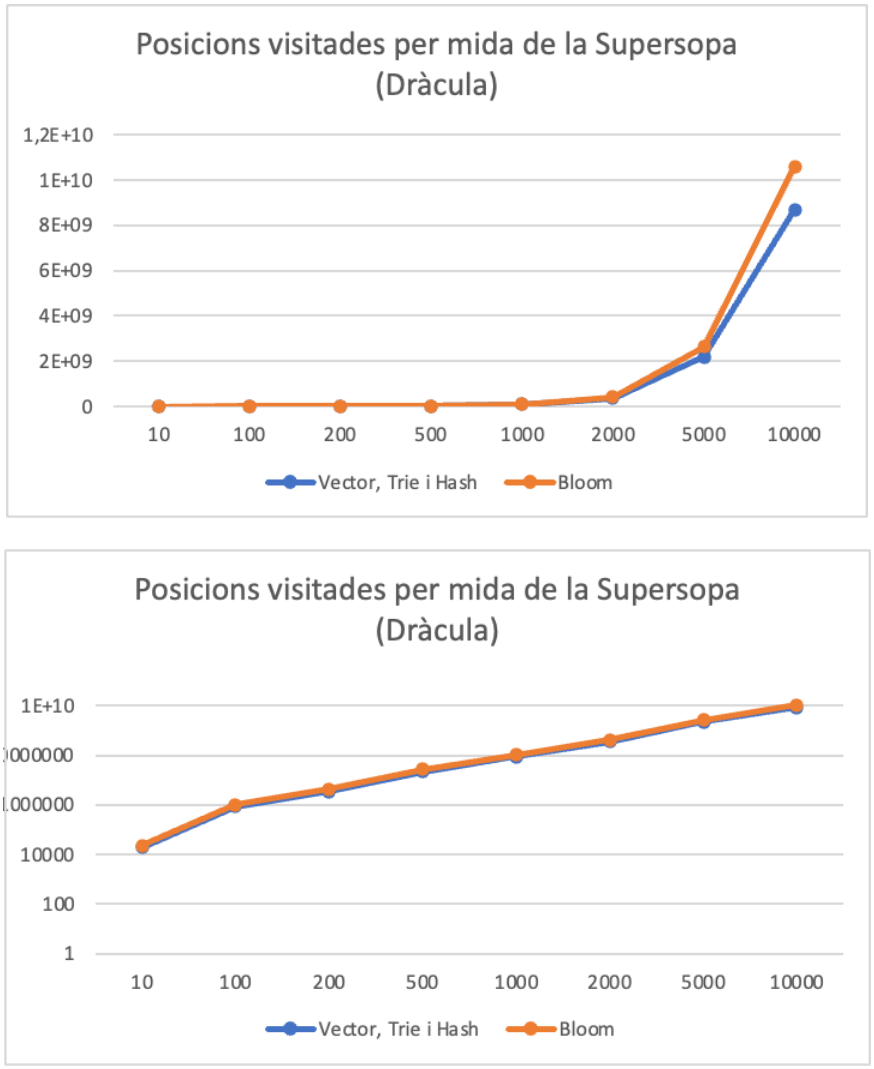
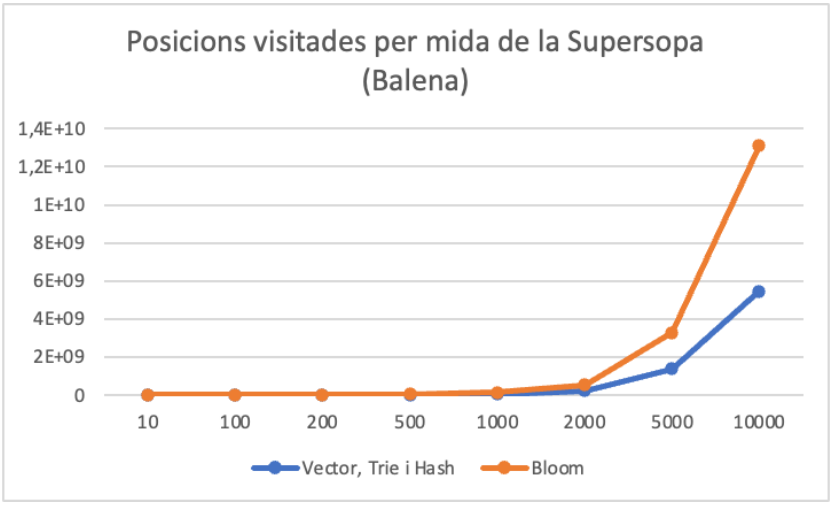


Figura 5: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Dràcula, en escala lineal i logarítmica.



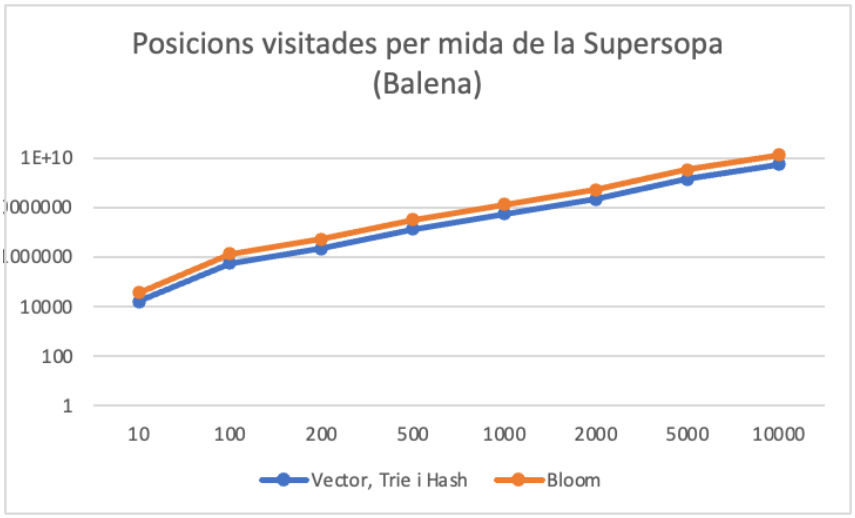
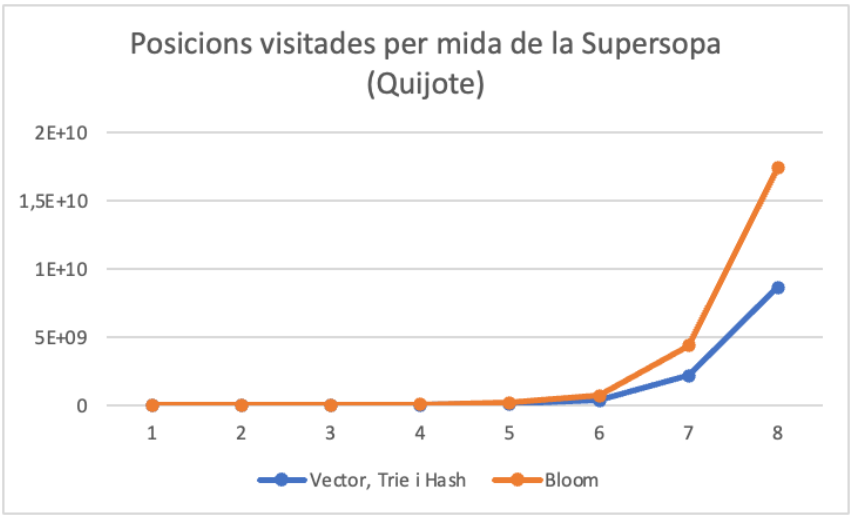


Figura 6: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Balena, en escala lineal i logarítmica.



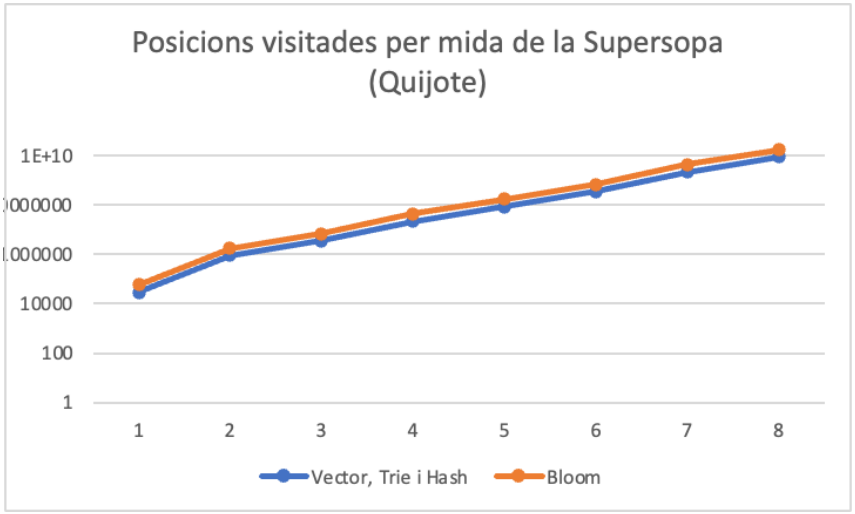
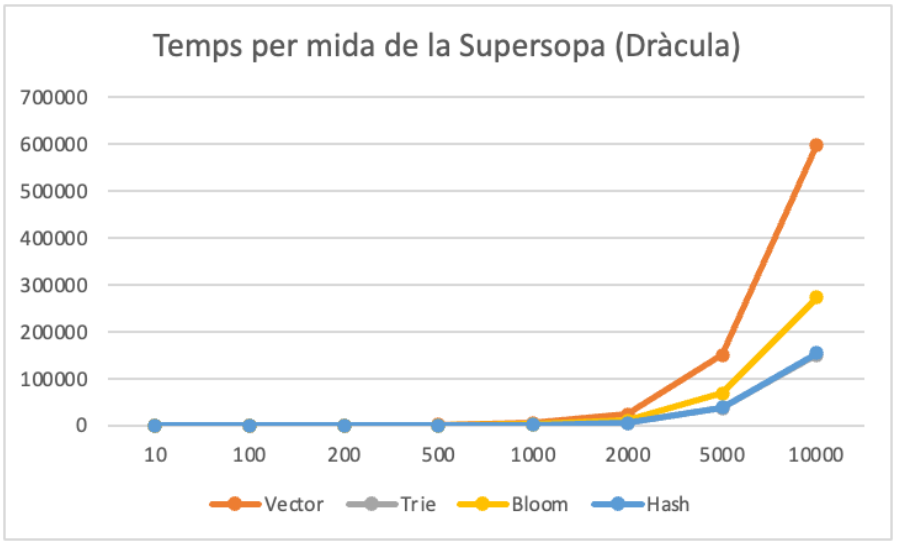


Figura 7: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Quijote, en escala lineal i logarítmica.

Observem que tant el Vector com la Trie com la Taula de Hash visiten el mateix nombre de posicions per a cada mida. El perquè d'això és molt simple, i és que tots estan usant la mateixa idea per podar els possibles camins. Les diferències de temps resultants es deuen que aplicar aquesta idea amb diferents estructures de dades no sempre té el mateix cost, sinó que depenent com s'adapti l'estructura de dades al problema el temps serà millor o pitjor, però en qualsevol cas la idea roman la mateixa.
El Filtre de Bloom aplica la mateixa idea que els altres, però el problema que apareix és que succeeixen falsos positius entre els prefixos, cosa que fa que continuï per camins els quals no hauria de continuar.

Pel segon conjunt de paràmetres del filtre de Bloom, hem imposat m = 100 * n on m és la mida de la taula de bits del filtre i n la mida del diccionari i k = 6 la quantitat de funcions de hash. Els gràfics que obtenim son els següents:



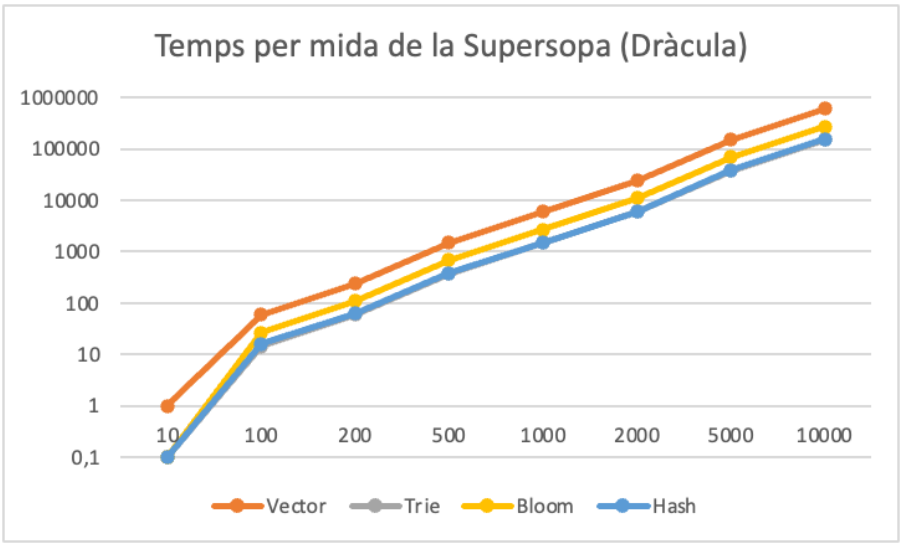


Figura 8: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Dràcula, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

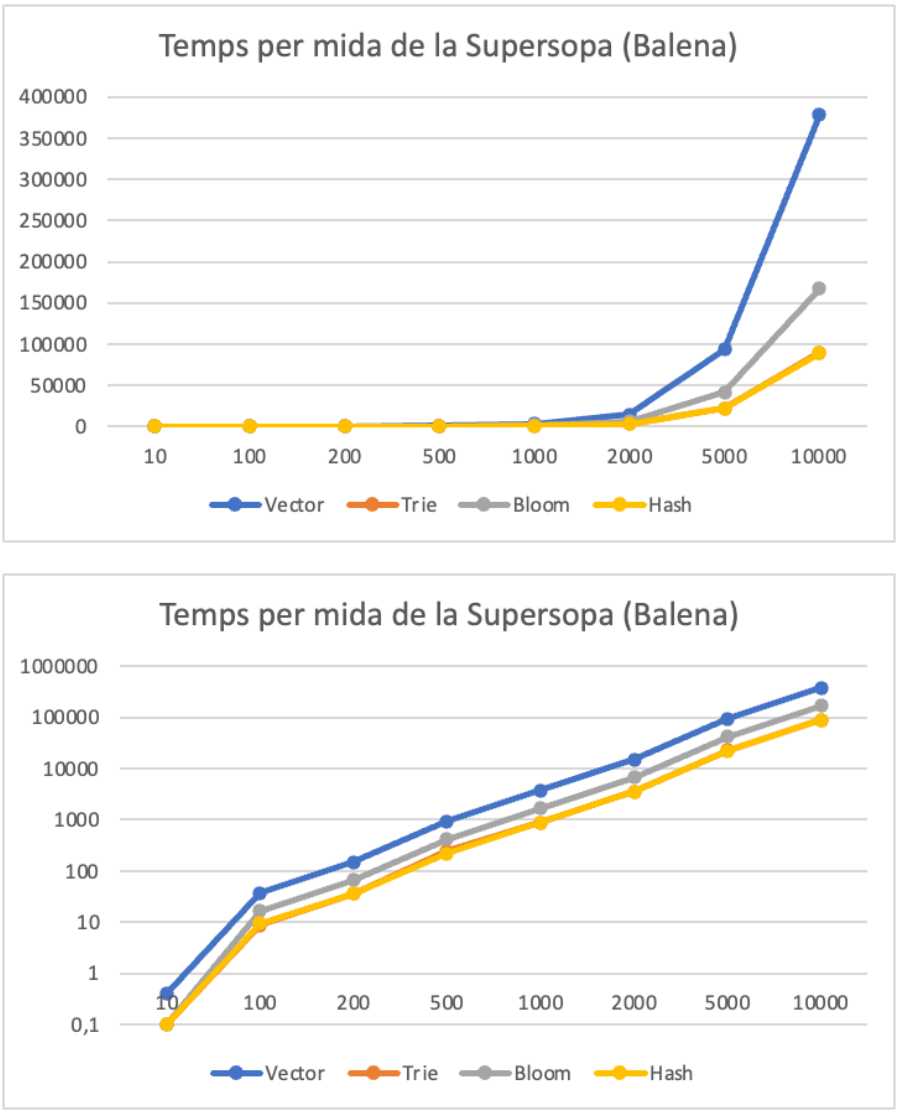


Figura 9: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Balena, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

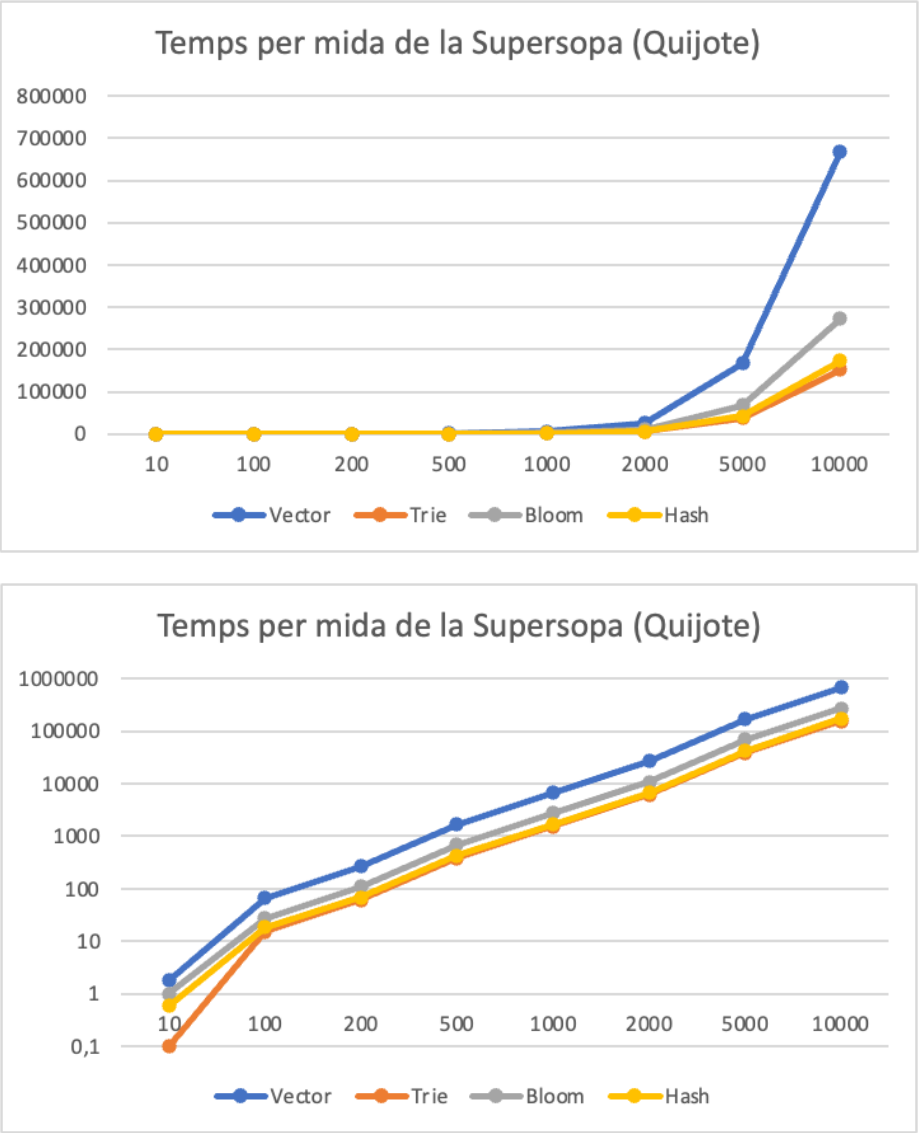
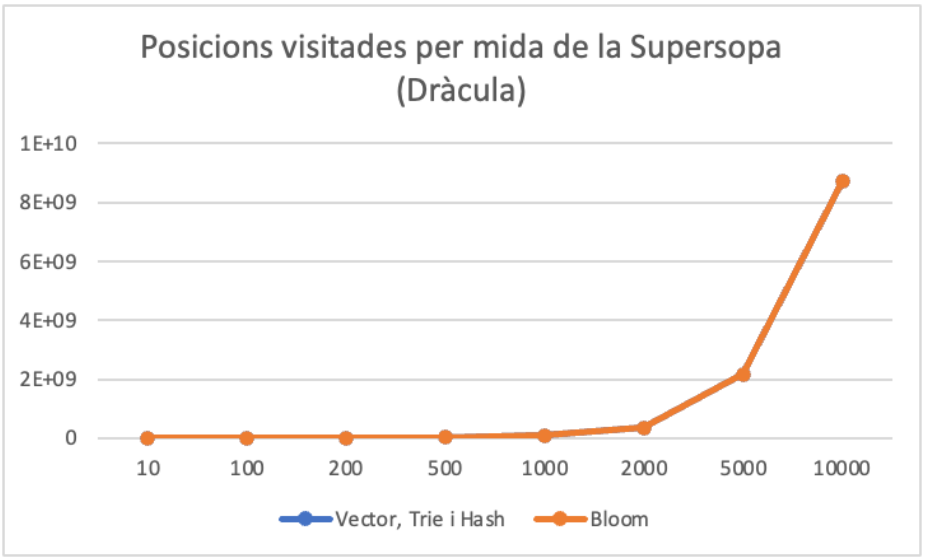


Figura 10: Temps que es tarda en resoldre una Super Sopa de mida donada sobre el diccionari Quijote, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

Com que només s'han variat els paràmetres del Filtre de Bloom, no s'observen canvis en la resta d'estructures de dades. En canvi, el temps d'execució del Filtre de Bloom sí que ha canviat substancialment, principalment degut a que hem reduït la quantitat de funcions de hash de 24 a 6, un factor de 4.



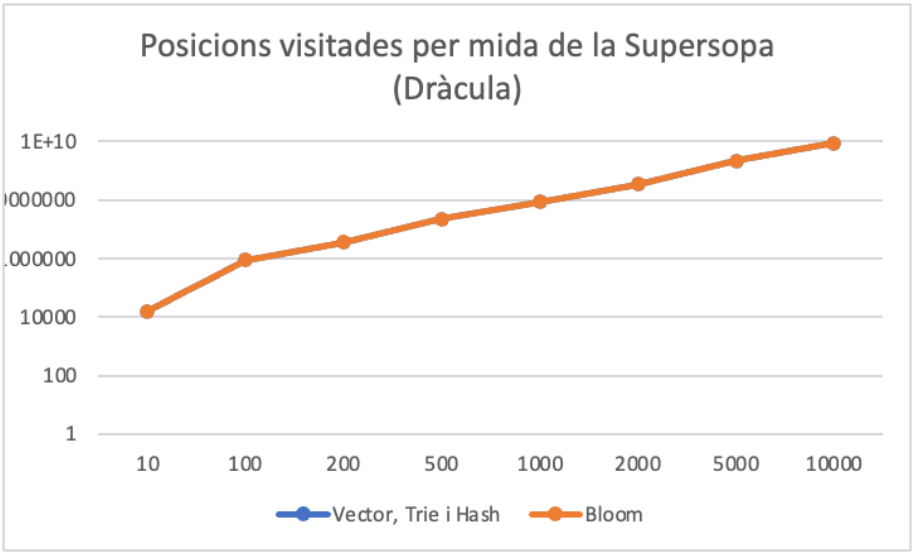


Figura 11: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Dràcula, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

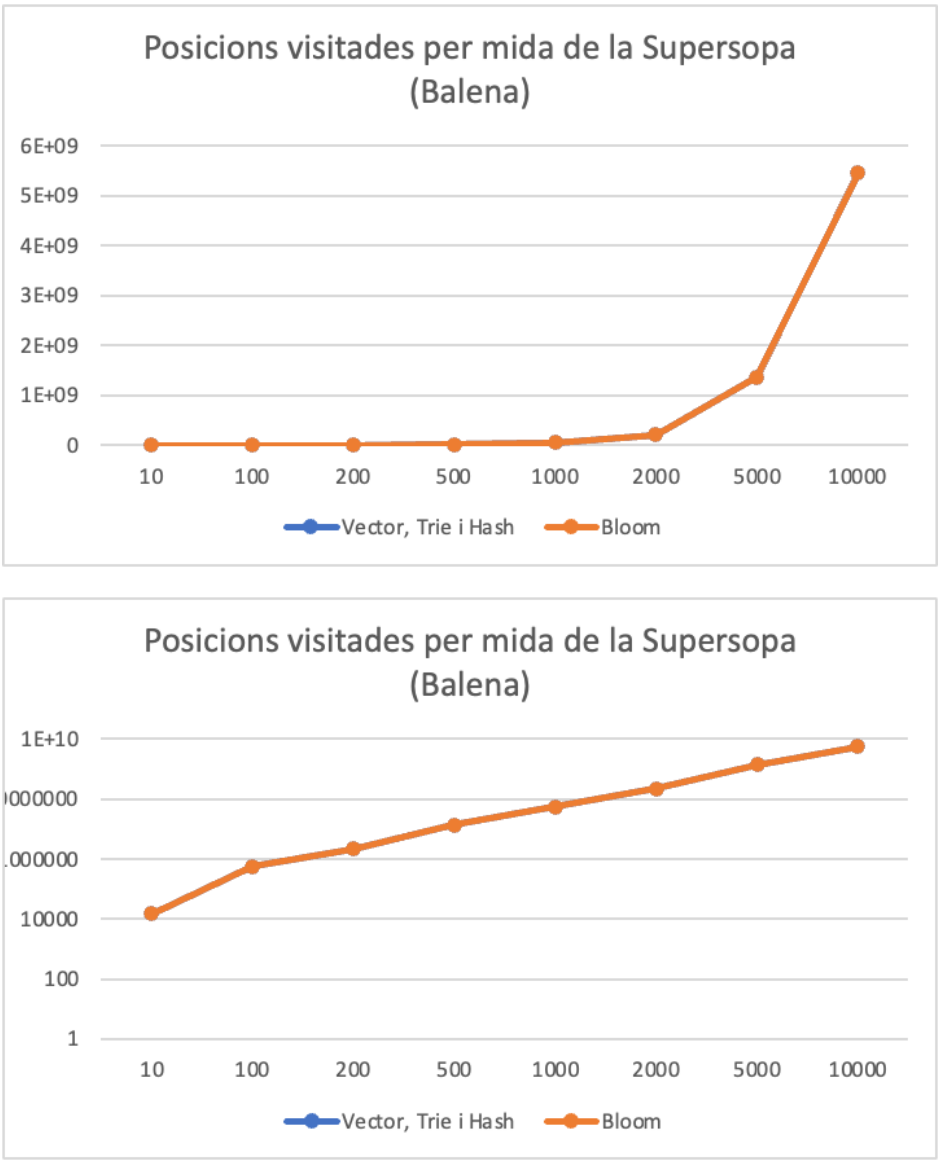


Figura 12: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Balena, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

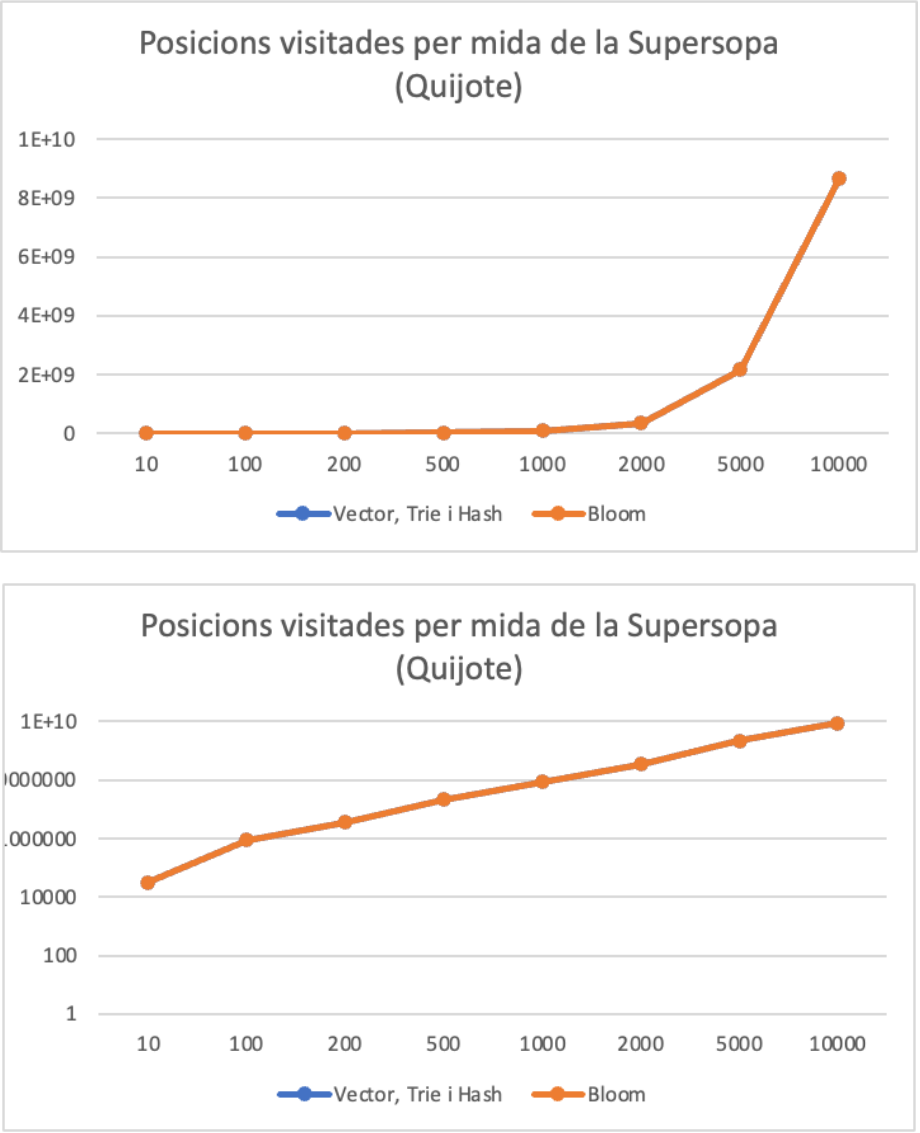


Figura 13: Posicions visitades per resoldre una Super Sopa de mida donada sobre el diccionari Quijote, en escala lineal i logarítmica amb els nous paràmetres del filtre de Bloom.

De nou no s'observen cap canvi substancial respecte les posicions visitades de la Trie, la Taula de Hash i el Vector, però clarament hi ha canvis respecte les posicions visitades pel Filtre de Bloom. Pel que fa a aquest últim, també observem menys falsos positius en els resultats finals, doncs en només 2 casos hem obtingut falsos positius i s'ha tractat només d'una paraula en cada cas. Per tant reduïm la quantitat de falsos positius en un factor de 5 respecte dels paràmetres anteriors.

Per consistència, tots aquests anàlisis s'han fet en el mateix ordinador portàtil, un Macbook Pro amb processador M1 Pro. Les dades han estat posteriorment tractades amb Excel per obtenir els gràfics anteriors.

5 Conclusions

Després de totes les dades obtingudes en l'experimentació, ens és fàcil concloure que, de les solucions explorades, resoldre una Super Sopa utilitzant un Trie és el més eficient, tot i que seguit d'aprop per el Doble Hash. De fet, en les gràfiques es poden solapar els temps del Trie i del Hash degut a la poca diferència en temps que hi ha entre ambdues implementacions. Els valors numèrics tractats per obtenir les gràfiques, però, mostren que el Trie és sempre més ràpid que el Hash, tot i que sigui de forma marginal.

També cal esmentar que el Doble Hash i el Filtre de Bloom funcionen així de bé perquè s'hi han afegit els prefixos dels diccionaris, ja que sinó no tindrien una manera clara de tallar els camins i tardarien molt més a executar-se.

En quant al vector i al Filtre de Bloom, es pot concloure que el seu rendiment dependrà enormement dels paràmetres del filtre. Si aquests estan ben escollits, el Filtre de Bloom tindrà millor que el vector ordenat, però si no ho estan, el vector ordenat resoldrà el problema amb més rapidesa. Tot i així, teòricament el Filtre de Bloom hauria de ser més ràpid asimptòticament que el Vector Ordenat, ja que aquest últim ha d'efectuar una operació logarítmica cada vegada.

Un anàlisis similar es pot fer observant el nombre de posicions visitades. En totes les proves efectuades, el Vector, el Trie i el Hash visiten exactament el mateix nombre de posicions de la sopa per arribar al resultat. Això és perquè, tot i que usen tècniques més o menys eficients, la poda de camins que s'efectua és la mateixa ja que es tracta de la mateixa idea però amb diferents estructures de dades. Amb la primera tria de paràmetres del Filtre de Bloom, aquest visita notablement més posicions que els altres mètodes. Això és degut a trobar prefixos falsos positius, cosa que provoca que continui per posicions per les que teòricament no hauria de continuar. Amb la segona tria de paràmetres, tot i que l'anàlisis numèric mostra que sempre visita més posicions que la resta de mètodes, aquesta diferència és prou petita per a que no sigui apreciable en els gràfics.

Cal tenir en compte que només hem mesurat experimentalment la complexitat en temps. No s'ha fet una experimentació directe per mesurar la memòria, però d'haver-ho fet hauríem notat que el Hash i el Filtre de Bloom tindrien una complexitat $O(m^2)$ (on m és la mida del

diccionari), ja que s'han d'afegir els prefixos de cada paraula del diccionari. En canvi, la complexitat en termes de memòria de la Trie i el Vector Ordenat és lineal en la mida del diccionari. D'aquí en podem concloure que la Trie (en particular, el TST) és la millor estructura de dades per resoldre la Super Sopa, tant en termes de temps com de memòria.

6 Fonts

https://en.cppreference.com/w/

https://en.cppreference.com/w/cpp/algorithm/lower_bound

https://en.cppreference.com/w/cpp/algorithm/sort

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

https://www.geeksforgeeks.org/double-hashing/

https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/

https://www.cs.upc.edu/~ps/downloads/tst/tst.html

https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/The%20Art%20of%20Computer%20Programming%20%28vol.%203_%20Sorting%20and%20Searching%29%20%282nd%20ed.%29%20%5BKnuth%201998-05-04%5D.pdf

https://doc.lagout.org/Others/Data%20Structures/Handbook%20of%20Data%20Structures%20and%20Applications%20%5BMehta%20%26%20Sahni%202004-10-28%5D.pdf