Anàlisi de quatre mètodes de resolució de la Super Sopa

Aina Gomez, Paula Muñoz, Jordi Palomera, Neus Mayol

Octubre 2022

Índex

1 Introducció

Moltes vegades relacionem els algorismes amb la modernitat, l'ús de les noves tecnologies o les xarxes socials. I sí, és veritat, l'algorísmia juga un paper important en aquests camps, però si retrocedim temporalment veurem que ja fa centenars i milers d'anys que fem servir aquests procediments. Podem dir que els algorismes sempre ens han donat el camí per trobar la solució a tot tipus de problemes quotidians d'ordenació, de càlcul, de lògica etc.

És més, independentment de la seva dificultat, cost temporal o de memòria, tots tenen una cosa en comú: mecanitzar i agilitzar tasques. És just en aquest moment —quan les tasques es poden mecanitzar— que podem deixar l'execució de l'algorisme a mans d'un ordinador, que resoldrà problemes com el que tracta aquest escrit de manera fàcil i ràpida.

2 Formulació del problema

El problema presenta una sopa de lletres de mida $n \times n$, les paraules de la qual poden disposar-se en qualsevol de les vuit direccions. És a dir: amunt, avall, esquerra, dreta, i les quatre diagonals. A més, una paraula pot tenir canvis de direcció entremig i sobreposar-se amb una altra paraula, però mai es sobreposarà amb ella mateixa.

Com el lector ja haurà pogut deduir, el principal problema plantejat és la cerca de paraules dins la sopa de lletres. Les paraules que s'han de buscar a la sopa venen determinades per un diccionari donat. Per assegurar-se que es trobaran paraules a la sopa, s'hi introduiran alguns mots del diccionari de forma aleatòria al moment de crear-la. Ara bé, al moment de resoldre-la, s'hauran de trobar tots els mots que hi hagi al diccionari, no només els que s'han introduït expressament.

3 Metodologia

Per comoditat i adequació amb el que es demana a l'enunciat, s'ha decidit programar els algorismes en C++. Durant el treball s'aniran ensenyant els algorismes segons aquest llenguatge, la resta del codi serà visible dins el repositori de *GitHub* enllaçat a l'Annex del final de l'escrit.

Per tal de resoldre la sopa de lletres, l'enunciat planteja quatre maneres d'emmagatzemar el diccionari de paraules donat. Per tal d'organitzar millor les tasques a l'hora de programar, s'ha decidit crear una classe per a cada una de les maneres següents i que podreu trobar explicades amb detall més endavant.

- Amb un vector ordenat (o *sorted vector*).
- Amb un *trie*.
- Amb un filtre de Bloom.
- I amb una taula de *hash* amb *double hashing*.

També, hem creat una cinquena classe anomenada *SuperSopa*, on hi guardem les funcions principals de la sopa de lletres. A més, hem dissenyat una sèrie d'experiments per comparar l'efectivitat de les diferents estructures de dades a l'hora de resoldre la Super Sopa.

4 Vector ordenat (o *Sorted Vector*)

L'estructura de dades més simple i coneguda. En aquest cas s'emmagatzemarà cada paraula dins una posició del vector. Per tant, i mirant-ho des de termes del llenguatge C++, sempre es parlarà d'un vector de paraules (o *string*).

El vector ha de ser ordenat, les paraules estaran organitzades alfabèticament tal i com els diccionaris. L'enunciat no garanteix que les paraules del diccionari donat estiguin en ordre alfabètic. És per això que, abans que res, s'haurà de procurar que estiguin ben ordenades. D'aquesta manera, el vector s'estructurarà tal i com representa la *Figure 1*, on s'han fet servir paraules d'exemple ordenades alfabèticament.

...	any	bou	plor	rosa	urna	...

Figure 1: Exemple de vector ordenat.

I és que de fet, el vector és el principal atribut de la classe *SortedVector*, definida al fitxer *diccSortedVector.cc* i al seu fitxer de capçaleres corresponent: *diccSortedVector.hh*. L'altre atribut de la classe és un *map* anomenat *trobades*, la clau del qual és una paraula que és solució de la sopa i s'aparella amb un enter corresponent al nombre de vegades que hi apareix.

4.1 Funcionament de la classe i anàlisi del seu cost

Tal com s'ha explicat anteriorment, abans de començar a solucionar la sopa, hem de construir l'estructura de dades adequada per emmagatzemar el diccionari, en aquest cas un vector de paraules ordenades alfabèticament. Tenint l'estructura creada, ja es podrà procedir a la resolució de la sopa.

4.1.1 Construcció de l'estructura

En primer lloc i des del fitxer *experiment_vector.cc*, es crearà una instància de la classe *SortedVector*. Seguidament s'afegirà el diccionari de paraules al vector cridant la funció *afegir* de la classe *SortedVector*. L'únic paràmetre de la funció *afegir* és el diccionari que, al ser també un vector de paraules, només caldrà ordenar i emmagatzemar a l'atribut de la classe. Per ordenar el vector s'han considerat diversos algorismes, finalment s'ha decidit utilitzar l'ordenació *Mergesort*.

Mergesort Aquest procediment és un dels mètodes més eficients d'ordenació. Pertany al grup dels algorismes de dividir i vèncer (o *divide and conquer*, en anglès), que agrupa tots aquelles funcions que divideixen el problema en subproblemes fàcils de resoldre per tal d'acabar resolent el problema en general.

És per això que el *Mergesort* comença dividint el vector de *string* en dos subvectors, i així recursivament fins a obtenir vectors de mida 1. Quan els subvectors tenen mida 1 es sap del cert que estaran ordenats, ja que són vectors d'un sol element. S'ha arribat a un problema fàcil de resoldre i per tant, l'algorisme podrà començar a solucionar la totalitat del problema.
Tot aquest procés es pot veure dins el procediment *mergesort*, que pertany a la classe *SortedVector*.

```
void SortedVector::mergesort(int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergesort(l,m);
        mergesort(m+1,r);
        merge(l,r,m);
    }
}
```

Listing 1: Funció *mergesort*.

Tal com mostra el codi, les crides a *mergesort* s'encarreguen de la divisió, la funció *merge* s'ocupa de l'ordenació dels subvectors i la fusió d'aquests. *merge* sempre rebrà un vector on cada meitat està ordenada independentment de l'altra. Es pot assegurar que sempre serà així perquè la ordenació comença en subvectors de mida 1 —ordenats de per sí— i va fusionant subvectors ja ordenats i repetint el mateix procediment fins a obtenir el vector inicial ordenat.
El fragment de codi a continuació realitza la tasca comentada: En primer lloc es creen dos vectors corresponents als subvectors que s'han d'organitzar. Per ordenar el vector es recorren ambdós subvectors linealment amb dos iteradors. A cada iteració es compararen els valors dels subvectors en la posició de cada iterador i el més petit dels dos valors es guarda al vector v, que contindrà el vector ja ordenat. Seguidament s'augmenta el valor de l'iterador del subvector que contenia el valor més petit i torna a començar el bucle.

Com que no podem assegurar que els dos subvectors tenen la mateixa mida, en acabar l'ordenació hi ha dos bucles que acaben de posar els valors del subvector de mida superior.

```
void SortedVector::merge(int l, int r, int m) {
    int n1 = m - l + 1, n2 = r - m;

    // Dos vectors amb els valors de cada subvector:
    vector< string> v1(n1), v2(n2);
    for (int i = 0; i < n1; i++)
        v1[i] = v[l + i];
    for (int j = 0; j < n2; j++)
        v2[j] = v[m + 1 + j];

    // Ordenacio:
    int i = 0, j = 0, k = l;
    while (i < n1 and j < n2) {
        if (v1[i] <= v2[j]) {
            v[k] = v1[i];
            ++i;
        } else {
            v[k] = v2[j];
            ++j;
        }
        ++k;
```

```
    }

    // Ordenacio de les parts restants
    while (i < n1) {
        v[k] = v1[i];
        ++i; ++k;
    }
    while (j < n2) {
        v[k] = v2[j];
        ++j; ++k;
    }

}
```

Ja per acabar, és ben sabut que el cost de l'algorisme *Mergesort* és $\Theta(n \log n)$ en tots els seus casos, ja que fa servir dues crides recursives que divideixen el vector per la meitat, i la part no recursiva té cost $\Theta(n)$.

4.1.2 Resolució de la sopa

Al tenir el vector ordenat alfabèticament, s'ha cregut que l'opció més eficient és utilitzar la cerca dicotòmica per resoldre la sopa. Tot i així s'ha decidit retocar la manera com s'implementa aquesta per adaptar-se al problema, i que no s'hagi de cercar dins de tot el diccionari cada vegada que es llegeix una nova lletra de la sopa.

Per començar la resolució, el procediment *resoldre* de la classe *SuperSopa* recorrerà tota la sopa posició per posició. A cada posició es buscarà totes les solucions que comencen des d'allà. Per fer-ho s'entrarà a l'algorisme *buscarParaula*.

buscarParaula La idea de l'algorisme *buscarParaula* pot semblar complicada d'entrada, però s'ha considerat que és un mètode eficient. S'ha vist que la cerca d'una paraula de la sopa al diccionari pot donar lloc a diferents casos:

1. La paraula no és al diccionari.
2. La paraula és al diccionari
   (a) i és també prefix de la següent (o següents) paraules, i podem trobar una d'aquestes paraules seguint la cerca des d'aquella posició.
   (b) i és també prefix de la següent (o següents) paraules, però cap de les paraules que la contenen de prefix es poden trobar començant per aquella posició.
   (c) i no és prefix de cap paraula.

Per tant, una vegada s'ha trobat una paraula que es prefix s'ha de continuar buscant per assegurar-se de trobar totes les paraules més llargues que la tenen com a prefix, en cas que hi siguin. En altres paraules, la cerca començant per aquella posició acaba quan al diccionari no hi ha cap de les paraules que s'hi generen en qualsevol direcció i sentit, i incloent canvis de direcció en el transcurs de la paraula.

En C++, les cadenes de text o *string*, són definides com a vectors de caràcters. Per tant, podem veure el vector ordenat com un vector de vectors de caràcters o en altres paraules: com una matriu de caràcters. A la *Figure 2* es mostra un exemple visual d'un possible fragment del vector.

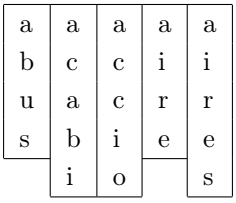

Figure 2: vector vist com una matriu de caràcters.

Per recórrer aquesta "matriu" de caràcters utilitzarem tres variables. Les dos primeres: l i r, serveixen per buscar la primera i l'última ocurrència al vector de les paraules amb el prefix llegit a la sopa. La tercera variable s'anomena *iterador* i serveix per saber en quina posició de la paraula s'ha de trobar cada nova lletra que llegim de la sopa. En cas que la paraula del *Sorted Vector* tingui l'última lletra llegida de la sopa a la posició *iterador*, podrem dir que la paraula tindrà almenys un prefix de mida *iterador* igual que la concatenació de lletres llegides de la sopa. Aquestes són les paraules que haurem de procurar incloure dins el subvector [l,r].

Aquesta explicació és molt més senzilla si l'acompanya l'exemple gràfic *Figure 3*, que mostra com els valors de l i r es mantenen a l'inici i fi del vector, ja que totes les paraules compleixen el prefix llegit de la sopa. En canvi, quan es llegeix un nou caràcter de la sopa, només hi ha dos paraules que tinguin la lletra buscada (en aquest cas, una i) a la posició *iterador*. Per això canvien els valors de l i r.

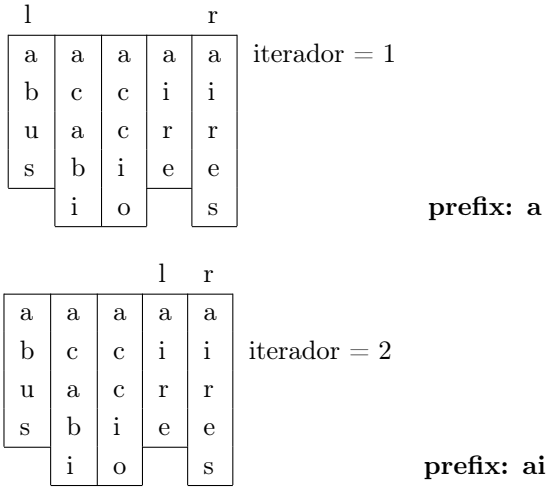

Figure 3: l i r limiten les paraules del vector que tenen com a prefix la cadena de caràcters es va llegint.

El procediment segueix i es van llegint nous caràcters. Imaginem que en aquest cas els caràcters llegits ens porten a una paraula que és dins del diccionari. Aquest podria ser, per

exemple, el cas de la *Figure 4*. Veiem llavors que una paraula serà solució del diccionari si i només si al moment que es troba a la posició l, el valor de *iterador* és igual a la seva mida.

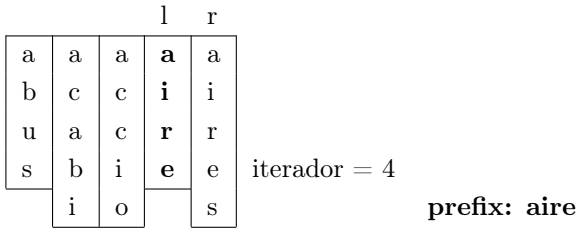

Figure 4: Hem arribat a una solució.

De totes maneres, tot i haver arribat a una solució, encara tenim la paraula "aires" inclosa al segment del vector [l,r]. Per tant, hem de continuar llegint caràcters de la sopa per veure si trobem la lletra que ens falta. En cas que la trobem, l es mourà una posició. Així doncs, l marca la posició de la paraula "aires" quan la variable *iterador* és igual a la mida de la paraula.

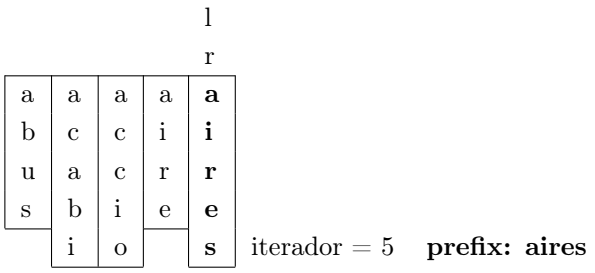

Figure 5: De nou, la condició per trobar una paraula es compleix.

Com que ja hem afegit l'única paraula que ens quedava dins el segment [l,r], és hora de tirar enrere la cerca, llegir altres caràcters de la sopa, recuperar el prefix i les variables l i r anteriors. També s'ha de decrementar la variable *iterador*. Una vegada s'ha retrocedit, ja podrem repetir el mateix procés cap a altres direccions fins no poder trobar més paraules.

Per tant, s'està encarant un problema de cerca exhaustiva que hem de fer satisfent una restricció molt simple: que les paraules generades per la cerca pertanyin al diccionari o siguin prefix d'una o més paraules que hi pertanyin. Per aquesta raó s'ha decidit que el procediment *buscarParaula* es programi amb un algorisme *backtracking* recursiu, el codi del qual es mostra al següent fragment.

```
void SortedVector::buscarParaula(int i , int j,
                                 vector<vector<bool>>& pos,
                                 int l, int r, int iterador,
                                 Sopa& s) {

    // Busquem la primera i ultima ocurrencia
    int nl = firstOcurrence(l,r,s[i][j],iterador);
    int nr = lastOcurrence(max(l,nl),r,s[i][j],iterador);
```

```
    // Cert si hem trobat paraules amb el prefix demanat
    if ((nl <= nr and nl != -1 and nr != -1)) {
        ++iterador;

        // Comprovem la condicio de solucio
        if (iterador == v[nl].size()) {
            ++trobades[v[nl]]; // Afegim la paraula a la solucio
            // Incrementem nl (no necessitarem mes la paraula)
            ++nl;
        }
        // Extenem la cerca cap a totes les direccions.
        int direccions_provades = 0, ni, nj;
        pos[i][j] = true;
        while (direccions_provades < 8) {
            ni = DIR[direccions_provades].first + i;
            nj = DIR[direccions_provades].second + j;
            if (compleixLimits(s,ni,nj,s.size())
                                and not pos[ni][nj]) {
                buscarParaula(ni,nj,pos,nl,nr,iterador,s);
            }
                ++direccions_provades;
        }
        --iterador;
        pos[i][j] = false;
    }
}
```

On, abans que res, es fa la cerca de la primera i última ocurrència de les paraules del diccionari que compleixen la condició i s'emmagatzemen a les variables *nl* i *nr* respectivament. Si *nl* i *nr* formen un interval correcte dins el vector, es busquen noves lletres de la sopa i es torna a cridar *buscarParaula* perquè cerqui dins l'interval [*nl*,*nr*].

El principal avantatge d'aquest tipus de cerca és que, per cada lletra nova que llegim, no caldrà cercar dins de tot el vector ordenat: podrem partir d'un subvector que ja sabem que conté totes les paraules amb prefix igual a les lletres que hem llegit anteriorment i des d'aquell punt, buscar l'interval que contingui les paraules el prefix de les quals són iguals a l'anterior, afegint-hi l'última lletra que hem llegit de la sopa.

Podem dir que aquest algorisme és correcte partint dels casos base que, tal com hem vist ho són: Quan una paraula no es troba al diccionari el segment del vector [*l*,*r*] no serà vàlid, i per tant es deixarà de buscar. En canvi sempre que arribem a una solució, la trobarem a la primera posició del segment (*l*) quan el valor d'*iterador* és igual a la mida de la paraula. Si trobem una paraula començarem la següent cerca partint del subvector [*l+1*,*r*], ja que sabem que la paraula a la posició *l* ja ha estat trobada, i que de moment no tornarà a ser solució de la sopa perquè les següents paraules que buscarem són de mida superior.

Pel que fa la resta de l'algorisme, podem justificar l'adequació a la seva tasca perquè per cada nova posició llegida a la sopa i mentre hi hagi paraules al diccionari que compleixin el prefix llegit, continuem buscant paraules en les vuit direccions. A més excloem de la cerca aquelles posicions que ja s'han visitat en aquella paraula i que ja formen part del prefix.

Per acabar d'entendre bé l'algorisme, mirarem com funcionen les cerques de la primera i l'última ocurrència.

firstOcurrence Com el lector ja haurà pogut deduir, es farà servir la cerca dicotòmica per dissenyar aquests algorismes. Cal entendre però, que una cerca dicotòmica normal busca només si l'element donat existeix al vector. En aquest cas però, volem cercar la primera ocurrència de l'element —un prefix— dins la sopa. Recordem que la cerca dicotòmica descarta un segment del vector a través del càlcul del punt mig entre el principi i el final del mateix. En aquest problema però, ens trobarem en els següents casos.

1. No hi ha cap paraula amb el prefix demanat: S'acabarà la funció sense èxit.
2. El punt mig conté el prefix buscat, però no és la primera ocurrència: Es tornarà a cridar la funció perquè cerqui la primera ocurrència dins el subvector que comença a l'inici del vector i acaba a la posició anterior al punt mig.
3. El punt mig conté el prefix buscat i és la primera ocurrència: Es retorna el punt mig.
4. El punt mig conté un prefix posterior al buscat: Es repeteix la cerca descartant la segona meitat del vector.
5. El punt mig conté un prefix anterior al buscat: Es repeteix la cerca descartant la primera meitat.
6. El punt mig conté una paraula de mida més petita que el prefix: Significa que la paraula no existeix o bé que està abans del punt mig. No pot estar situada posteriorment perquè les paraules després d'una paraula de mida inferior tindran un prefix diferent. Per tant la resolució del cas serà igual a la de l'apartat 2.

També ens podem trobar als casos 4 i 5, i que finalment el prefix buscat no existeixi. Això no és problema ja que programem l'algorisme recursivament, i repetirem la cerca fins a veure que el prefix demanat no existeix. Això és el que podem veure reflectit al codi en C++ de la funció *firstOcurrence*.

```
int SortedVector::firstOcurrence(int l, int r, const char& c,
                                                    int iterador) {
    // Cas 1
    if (l > r) return -1;
    int m = (l + r) / 2;

    if (v[m].size() > iterador) {
        if (v[m][iterador] == c) {
            if (l != m and (m != 0 and v[m-1].size() > iterador
                                    and v[m-1][iterador] == c)) {
```

```
                // Cas 2
                return firstOcurrence(l,m-1,c,iterador);
            }
            else // Cas 3
                return m;
        }
        else if (v[m][iterador] < c) // Cas 4
            return firstOcurrence(m+1,r,c,iterador);

        else // Cas 5
            return firstOcurrence(l,m-1,c,iterador);
    }
    else // Cas 6
        return firstOcurrence(l,m-1,c,iterador);

}
```

Com a nota final, cal recordar que dins el codi de la funció hi ha condicions addicionals perquè no totes les paraules del diccionari tenen la mateixa mida. Per això hem de comprovar que en accedir a una posició d'una paraula aquesta sigui vàlida.

lastOcurrence Té una implementació semblant a *firstOcurrence*, l'únic que ara necessitarem trobar l'última ocurrència d'un prefix enlloc de la primera. Dins el procés ens podem trobar amb els següents casos, que són molt semblants als de la funció anterior:

1. No hi ha cap paraula amb el prefix demanat: S'acabarà la funció sense èxit.
2. El punt mig conté el prefix buscat, però no és l'última ocurrència: Es tornarà a cridar a la funció perquè cerqui l'última ocurrència dins el subvector que comença al punt mig i acaba al final del vector.
3. El punt mig conté el prefix buscat i és l'última ocurrència: Es retorna el punt mig.
4. El punt mig conté un prefix posterior al buscat: Es repeteix la cerca descartant la segona meitat del vector.
5. El punt mig conté un prefix anterior al buscat: Es repeteix la cerca descartant la primera meitat.
6. El punt mig conté una paraula de mida més petita que el prefix: Significa que la paraula no existeix o bé que està abans del punt mig. No pot estar situada posteriorment perquè les paraules després d'una paraula de mida inferior tindran un prefix diferent. Per tant la resolució serà la mateixa que en el cas 2.

Implementat, i afegint-hi condicions addicionals per comprovar que no accedim a posicions d'un caràcter que no existeixen, queda així:

```
int SortedVector::lastOcurrence(int l, int r, const char& c,
                                                    int iterador) {
```

```
    // Cas 1
    if (l > r) return -1;
    int m = (l + r) / 2;

    if (v[m].size() > iterador) {
        if (v[m][iterador] == c) {
            if (r == m or (m == (v.size() - 1) )
                        or (c != v[m+1][iterador]) ) // Cas 3
                return m;
            else // Cas 2
                return lastOcurrence(m+1,r,c,iterador);
        }
        else if (v[m][iterador] < c) // Cas 4
            return lastOcurrence(m+1,r,c,iterador);
        else // Cas 5
            return lastOcurrence(l,m-1,c,iterador);
    }
    else // Cas 6
        return lastOcurrence(l,m-1,c,iterador);
}
```

Sabem que els algorismes de cerca dicotòmica de la primera i última ocurrència són correctes perquè la cerca dicotòmica general ho és, i perquè els casos especials que ens fan trobar la primera i última ocurrència sempre criden a una funció que segueix la cerca dins el subvector que conté la primera i última ocurrència respectivament. Aquestes crides sempre ens duran a un dels casos base iguals que els de la cerca dicotòmica general.

És fàcil veure que el cost de les cerques dicotòmiques *firstOcurrence* i *lastOcurrence* és $\mathcal{O}(\log n)$, sent n la mida del vector ordenat. Sabem que és així perquè es tracta d'una recurrència divisora que divideix l'espai de cerca per la meitat a cadascuna de les crides recursives, mentre que la part no recursiva té cost constant.
Pel que fa l'algorisme buscar paraula, no sabem quantes vegades l'haurem d'executar, ja que depèn de la sopa. De totes maneres sabem que la cerca acabarà sempre quan ja no hi hagi cap paraula del diccionari que compleixi la condició prefix. Per tant podem dir que cada cerca no s'executarà més de s vegades, sent s la mida de la paraula amb més caràcters del diccionari. Pel que fa la part no recursiva, tenim les dues crides a les cerques dicotòmiques i la inserció al mapa, totes amb cost logarítmic.

A partir d'això, i recordant que recorrem totes les posicions de la sopa al principi, al procediment *resoldre* de la classe *SuperSopa*, podem concloure que el cost de resoldre la sopa de lletres es calcula així:

$$Cost = \Theta(m)\mathcal{O}(s)\mathcal{O}(\log n) = \mathcal{O}(ms \log n)$$

Sent m el nombre de caràcters a una fila de la sopa de lletres.

5 *Trie*

Una de les quatre maneres que tenim per resoldre la *superSopa* és guardant el diccionari en una estructura en forma de *trie*. Aquesta estructura de dades presenta forma d'arbre i té diverses variants.

Una d'aquestes variants, i la que fem servir en aquest projecte, és la *ternary search tree*. Aquesta manera de guardar les dades és molt semblant a un *binary search tree*, però en aquest cas comptem amb un tercer fill. S'utilitza sobretot quan es tracta amb dades de tipus *string*.

Per acabar d'entendre el concepte de *ternary search tree*, en mirem un exemple: tenim les següents paraules i volem guardar-les en un arbre d'aquest tipus.

```
dicc = {nata, nas, nau}
```

Comencem per la primera paraula "nata". Com que l'arbre es troba buit, col·locarem el primer caràcter a l'arrel, el següent com a fill central, el següent com a fill central del anterior i així fins tenir-los tots col·locats. Ens queda un arbre com el de la figura 6.

Figura 6: arbre ternari amb paraula "nata".

Seguim amb la següent paraula "nas". Com que l'arbre no és buit i segueix un determinat ordre, haurem de col·locar la paraula on li correspon. Per això, compararem el primer caràcter amb l'arrel. Tenim que tots dos caràcters són iguals, per tant, saltarem al fill central. Farem el mateix procediment, però enlloc d'observar el primer caràcter de "nas", observarem el segon. Tornem a obtenir el mateix resultat (a == a), per tant, saltem al fill central.

Ara estem en el node 't', farem el mateix procediment: observem el tercer caràcter de "nas", i ens troben en que són diferents. Concretament, 't' < 's', per tant, passarem al fill esquerre. Com aquest està buit, escriurem allà el caràcter 's' acabant així d'afegir la paraula.

Anem ara amb la última paraula "nau". Si ens fixem, seguirà el mateix procediment que la paraula anterior, excepte que al arribar al node 't' ens trobem en que 'u' < 't'. Per tant, anirem cap al fill dret i escriurem allà l'últim caràcter. De manera que obtindríem un arbre com el de la figura 7.

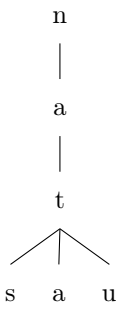

Figura 7: arbre ternari amb totes les paraules de *dicc*.

En conclusió, aquest tipus d'estructura segueix un ordre amb l'objectiu de facilitar la cerca de paraules. Aquest ordre és (1) el fill esquerre ha de ser més petit que el pare (2) el fill dret ha de ser més gran que el pare i (3) el fill central ha de ser la continuació del pare.

5.1 Especificació de la classe *diccTrie*

Per fer la implementació d'un diccionari en forma de *trie*, hem dissenyat una serie de funcions i procediments que es poden trobar a la classe *diccTrie.hh*. A continuació en comentarem els atributs i les tres funcions que utilitzarem tant per emmagatzemar les paraules com per resoldre la sopa de lletres.

5.1.1 Atributs privats

Abans de començar amb l'explicació de les funcionalitats, és important conèixer de dintre l'estructura amb la qual estem treballant.

```
struct node_trie {
    string info;

    node_trie* esq;
    node_trie* dre;
    node_trie* cnt;

    bool finalparaula;

};
```

Com hem comentat abans, cada node té tres fills. Per tant, tindrem tres nodes que corresponen a cadascun d'ells. A més, també hi tenim un *string* on hi guardarem un tros de paraula i un *bool* que ens indicarà si aquest és l'últim tros que forma una paraula.

Per acabar, també guardarem el primer node de l'arbre, l'anomenarem *arrel*, i un *int* per mantenir un registre de les paraules trobades. Aquest *int* l'anomenarem *paraulesTrobades*.

5.1.2 Funcionalitat *afegir*

Abans, quan hem explicat en que consistia un *ternary search tree*, hem vist un exemple d'inserció, així que hem pogut veure quina és la idea a seguir. Com que treballem amb punters, la funció afegir cridarà una altra funció privada que recorrerà l'arbre de forma rescursiva.

```
void afegirRec(node_trie* &n, const string& p, int i);

void TrieDictionary::afegir(const string& p) {
    afegirRec(arrel, p, 0)
}
```

A la funció recursiva li passarem tres paràmetres: la paraula que volem afegir al diccionari, el primer node i un enter i que ens indicarà quin caràcter falta col·locar, és a dir, totes les lletres de la paraula p que es trobin per davant de i hauran estat afegides a l'arbre.

Quan tractem amb funcions recursives hem de tenir clar amb quins casos ens podem trobar i, sobretot, quins d'ells finalitzen amb la recursivitat. Estudiem aquests casos:

- **Cas base.**

  1. Es compleix que $i == p.mida()$.

     Quan es compleix que $i == p.mida()$ significa que hem pogut col·locar totes les lletres que formen part de la paraula p, per tant, hem acabat.

- **Casos recursius.**

  1. L'element del node és més gran que el caràcter i de la paraula p.

     Per tant, seguirem amb la crida recursiva passant com a node $n->dre$ i amb el mateix valor i (ja que no n'hem col·locat cap lletra).

  2. L'element del node és més petit que el caràcter i de la paraula p.

     Molt semblant al punt 1. Seguirem amb la crida recursiva passant com a node $n->esq$ i amb el mateix valor i.

  3. L'element del node és igual al caràcter i de la paraula p.

     En aquest cas, seguirem amb la crida recursiva passant com a node $n->cnt$, però incrementarem una posició el nombre i, ja que, encara que aquest caràcter no l'haguem afegit, el reutilitzarem per la nova paraula.

  4. El node n és *null*.

     Aquest cas és molt semblant al cas base, però encara ens manca més d'un caràcter per afegir. És per això que la crida recursiva haurà de seguir fins arribar a $i == p.mida$, és a dir, el cas base. Farem la crida recursiva passant el node $n->cnt$ i incrementant una posició i.

De manera que la funció per afegir una paraula a l'arbre queda de la següent manera:

```
void afegirRec(node_trie* &n, const string& p, int i) {

    string s(1, p[i]);

    if (n == nullptr) {
        node_trie* aux;
        aux = new node_trie;

        aux->info = s;
        aux->dre = nullptr;
        aux->esq = nullptr;
        aux->cnt = nullptr;
        aux->finalparaula = false;

        n = aux;
        ++i;

        if (i < p.size()) afegirRec(n->cnt, p, i);     // cas 4
        else n->finalparaula = true;                   // cas base
    }
    else if (n->info == s) {
        ++i;
        if (i < p.size()) afegirRec(n->cnt, p, i);     // cas 3
        else finalparaula = true;                      // cas base
    }
    else if (n->info > s) afegirRec(n->dre, p, i);     // cas 1
    else if (n->info < s) afegirRec(n->esq, p, i);     // cas 2
}
```

Correctesa. La justificació informal d'aquesta funció recursiva és la següent.

- **Cas base.**

  Si tenim que i és igual al nombre de lletres que conté p, significa que ja hem col·locat totes les lletres a l'arbre. Per tant, hem acabat.

- **Cas recursiu.**

  Si l'arbre és buit o el node amb què treballem és buit, l'inicialitzarem amb la lletra i de la paraula p. Llavors, com que ja no serà buit, podrem accedir sense error als seus subarbres.

  Si el node amb què treballem no és buit, compararem la lletra que volem afegir amb el valor del node i, segons el seu valor, ens mourem cap a la dreta o esquerra.

- **Decreixement.**

  Si ens fixem en el codi, només incrementem la i en aquelles crides en què ens desplacem al subarbre central. Això és perquè en aquests casos disminueix el nombre de lletres a afegir.

  També, a cada crida recursiva l'arbre es fa cada vegada més petit.

Cost. Afegir una paraula de longitud k en un *ternary search tree* té un cost mitjà de $O(\log n + k)$. Això depèn de si l'arbre es troba del tot equilibrat. És a dir, que té un nombre de nodes semblant al subarbre dret respecte al subarbre esquerra i el subarbre central. Si no és així, el cas pitjor ens trobem amb un cost de $O(n + k)$

5.1.3 Funcionalitat *simplificar*

Aquesta funció està pensada per ser cridada després de tenir l'arbre inicialitzat amb totes les paraules. Consisteix en agrupar tots aquells nodes que no presenten cap fill dret ni esquerre amb els seus fills centrals. És a dir, si tenim l'arbre de la *Figura 7* i cridem a la funció *simplificar*, ens retorni el següent arbre.

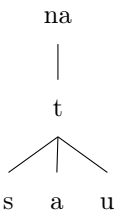

Figura 8: arbre ternari amb totes les paraules de *dicc* simplificat.

Hem de tenir en compte, tal com hem dissenyat la nostra funció *afegir*, que un cop el nostre arbre sigui simplificat, si volem afegir alguna paraula més, pot donar-se el cas que el resultat sigui erroni. Això és perquè en afegir no tenim en compte que un node pugui tenir més d'un caràcter.

Igualment, afegir una paraula després de simplificar l'arbre podria implicar refer els canvis fets per simplificar. Per exemple, si a l'arbre de la *figura 4* hi volem afegir la paraula nen, hauríem de tornar a separar el node *"na"*. És per això que hem decidit que la funció *simplificar* hauria d'utilitzar-se després d'afegir totes les paraules.

Tornem a treballar amb punters, per tant tindrem un mètode public que cridarà un altre mètode privat.

```
void simplificaRec(node_trie* &n);

void simplificaArbre() {
    simplificaRec(arrel);
}
```

A la funció recursiva li passarem el node per on volem començar a recórrer l'arbre. Com que el volem simplificar tot, començarem per l'arrel (primer node).

Igual que amb el mètode anterior, estudiarem els casos que acaben la recursivitat.

- **Cas base.**

  1. El node n és *null*

     Quan ens trobem que un node és *null*, vol dir que ja hem visitat tots els nodes d'aquell subarbre. Per tant, ja haurem ajuntat tots aquells que podem ajuntar.

- **Casos recursius.**

  1. El node n no presenta fills laterals, el seu fill central tampoc i n no és final de paraula.
     Si totes aquestes condicions es compleixen, ajuntarem la informació del node n afegint la informació del node central. També haurem de substituir els fills de n pels fills de *n->cnt*. Farem la crida recursiva sobre el mateix node després d'actualitzar-lo.
  2. No es compleix la condició anterior.
     Per tant, avançarem la recursivitat cap als seus tres fills: el dret, el central i l'esquerre.

Per tant, tenim el següent codi.

```
void afegirRec(node_trie* &n) {

    if (n != nullptr) {

        node_trie* seg = n->cnt;

        if (n->dre == nullptr and n->esq == nullptr and
            seg != nullptr and seg->dre == nullptr and
            seg->esq == nullptr and not n->finalparaula) {

                n->info += seg->info;
                n->finalparaula = seg->finalparaula;
                n->cnt = seg->cnt;
                simplificaRec(n);
        }
        simplificaRec(n->dre);
        simplificaRec(n->cnt);
        simplificaRec(n->esq);
    }
}
```

Correctesa. La justificació d'aquesta funció és la següent.

- **Cas base.**

  Si ens trobem amb que n és igual a *null*, significa que hem arribat al final de l'arbre. Per tant, hem acabat.

- **Cas recursiu.**

  Si el node n i el seu fill central no presenten fills laterals i n no és el final de paraula, ajuntarem els dos nodes i farem la crida recursiva al mateix node n, per si es pot tornar ajuntar amb el següent fill central. Si no, farem la crida per recórrer tots els subarbres de n.

- **Decreixement.**

  A cada crida recursiva l'arbre es fa cada vegada més petit. Fins i tot al primer cas recursiu, ja que en ajuntar dos nodes també estem escurçant la seva mida.

Cost. Com hem vist, per simplificar l'arbre necessitem recórrer tots els subarbres i veure quins nodes compleixen les condicions i poden ser agrupats. És per això que el cost de la nostra funció és lineal.

Una altra manera de simplificar Una altra manera de simplificar l'arbre podria ser ajuntar tots aquells nodes en que n no presenta fills centrals sense tenir en compte si n->cnt en presenta o no. D'aquesta manera, l'arbre de la *figura 7* quedaria de la següent manera.

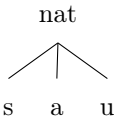

Figura 9: arbre *figura 3* simplificat amb diferents condicions.

Nosaltres hem decidit simplificar el nostre arbre com a la *figura 4* per poder fer les cerques amb més facilitat. Ja que en l'arbre de la *figura 9* hauríem de comparar amb l'últim caràcter de la paraula per saber cap a quin subarbre anar.

Com veurem al següent apartat, on expliquem com hem plantejat la funcionalitat *comprova*, per buscar una paraula mirem tota la paraula que guarda el nostre node. Per tant, si volguéssim utilitzar la implementació de la *figura 5*, hauríem de modificar també la funcionalitat *comprovar*.

5.1.4 Funcionalitat *comprovar*

Finalment, tenim la funció *comprovar*. Aquesta rep una paraula com a paràmetre i mira si existeix dintre del arbre. És en aquesta funció on la variable *finalparaula* té utilitat, ja que si trobem una paraula dins del nostre arbre, i coincideix en que l'ultim caràcter té com a cert aquest atribut, incrementarem la variable de *totalTrobat*.

Com sempre, tenim un mètode públic que crida a una funció privada.

```
void existeixParaula(const string& par, node_trie* n, int i,
                                                    bool& r);
bool comprovar(const string& par) {
    bool r = false;
    existeixParaula(par, arrel, 0, r);
    return r;

}
```

A la funció privada *existeixParaula* li passem l'*string* c (la paraula a cercar), el primer node de l'arbre, un índex i que ens indica el nombre de lletres trobades, i un *bool* on guardarem el resultat.

Abans de veure el codi de *comprova*, hem d'entendre una altra funció de la classe, anomenada *inclouParaula*. Aquesta rep dos *strings* com a paràmetres i mira si algun dels dos inclou l'altra.

```
bool inclouParaula(const string& a, const string b, int i, int& k);
```

Com veiem, rep les dues paraules *a* i *b* i dos enter *i* i *k*. L'enter *i* ens indica a partir de quin caràcter hem de buscar a la paraula *a*. En canvi, l'enter *k* té dues utilitats, primer l'utilitzem com a índex de la paraula *b* i, en acabar, l'utilitzem com un booleà: si *k* és 1 significa que la paraula *a* conté la paraula *b*, altrament si és 0.

Mirem els casos en què haurem de continuar la recursivitat i amb aquells que han d'aturar-la.

- **Casos base.**

  1. El node *n* és *null*.

     Si arribem a un node buit, significa que no hem pogut trobar la paraula *par*. El resultat serà *false* i haurem acabat.

  2. Si es compleix que *i* és igual al nombre de lletres de la paraula *par*.

     En aquest cas, significa que hem sigut capaços de trobar totes les lletres que formen la paraula *par*. Per tant, el resultat serà cert. A més, si l'ultim caràcter de *par* és final de paraula, incrementarem el nombre de paraules trobades a la sopa.

  3. La paraula *n->info* inclou a la paraula *par*.

     Si es dona el cas que la paraula que es troba al node *n* inclou la paraula *par* voldrà dir que hem acabat. Ja que *inclouParaula* mira totes les lletres després del index *i*. És a dir, que tota la paraula o les lletres que falten per cercar es trobaran incloses en el node *n*. Per tant, existeix la paraula *par* dins l'arbre. El resultat serà cert.

- **Casos recursius.**

  1. La paraula *par* inclou la paraula *n->info*.

     Aquest cas és bastant semblant al tercer cas base, però en aquest cas haurem de seguir amb la recursivitat. Si *par* inclou la paraula *n->info*, significa que encara faltaran lletres per ser trobades. Per tant, farem la crida cap al fill central i incrementarem *i* com tantes lletres conté la paraula del node *n* (ja que són lletres trobades).

  2. El caràcter *i* de *par* és més gran que el primer caràcter de *n->info*.

     El caràcter *i* de la paraula *par* és més gran que el que trobem a *n*, per tant, haurem de seguir la cerca pel subarbre esquerre sense incrementar el valor de *i*, ja que no hem trobat cap paraula.

  3. El caràcter *i* de *par* és més petit que el primer caràcter de *n->info*.

     Si el caràcter *i* de *par* és menor que el que guarda el node *n*, haurem de fer la crida recursiva cap al subarbre dret sense incrementar el valor de *i*.

El codi de *comprova* queda de la següent manera:

```
void existeixParaula(const string& par, node_trie* n, int i,
                                                    bool& r) {
    int j = 0;
    if (n == nullptr) {          // cas base 1
        r = false;
        return;
```

```
    }
    else if (i == par.size()) {                    // cas base 2
        if (n->finalparaula) ++totalTrobat;
        r = true;
        return;
    }
    else if (n->info[0] == par[i] and
                        inclouParaula(par, n->info, i, j)) {
        if (j == 0) {
            i += n->info.size();                   // cas base 3
            r = true;
            if (n->finalparaula and i == par.size()) {
                ++totatTrobat;
            }
            return;
        }
        else {                                     // cas rec 1
            int salt = n->info.size();
            existeixParaula(par, n->cnt, i+salt, r);
        }
    }
    else if (n->info[0] < par[i]) {                // cas rec 2
        existeixParaula(par, n->esq, i, r);
    }
    else if (n->info[0] > par[i]) {                // cas rec 3
        existeixParaula(par, n->dre, i, r);
    }
}
```

Correctesa. La justificació informal de la funció *comprova* és la següent.

- **Cas base.**

  Si ens trobem amb què i és igual al nombre de lletres que conté, *par* o bé amb què el node n és *null* haurem acabat amb la recursivitat. En el primer cas haurem trobat la paraula *par* i en el segon no.

- **Cas recursiu.**

  Si el node n no és buit, podrem accedir als seus subarbres sense produir-se cap error. Per cada node visitat realitzarem comparacions i, segons el resultat, seguirem el camí per un fill o per altre.

- **Decreixement.**

  Per cada lletra trobada incrementem la i una posició. Abans de fer la crida pel fill central actualitzem el valor de i, per tant hi haurà menys lletres a buscar a l'arbre.

  També, a cada crida recursiva l'arbre és fa cada vegada més petit.

Cost. Cercar una paraula de longitud k en un *ternary search tree* té un cost mitjà de $O(\log n + k)$. Això es pot veure condicionat segons l'equilibri de l'arbre. Per un arbre equilibrat entenem que té un nombre de nodes semblants a cadascun dels seus fills. Si es dona el cas que té més nodes concentrats en un subarbre pot presentar un cost de $O(n + k)$.

6 Filtre de Bloom

6.1 Introducció

Un filtre de Bloom és una estructura de dades eficient en l'espai que permet comprovar si un element forma part d'un conjunt. No emmagatzema els elements en sí, només si en formen part, i no se'n poden eliminar elements.

Està compost per un vector d'm bits, posats a 0, i una sèrie de k funcions de *hash* que generen distribucions aleatòries uniformes. Té dues funcionalitats: afegir un element i comprovar un element n'és part.

Per afegir-hi un element, es passa per les funcions de *hash*, que retornen una sèrie d'índexs del vector. Els bits d'aquestes posicions del vector s'han d'activar (posar a 1).

Per comprovar si un element pertany al filtre, també es passa per les funcions de *hash*, es comproven els bits dels índexs obtinguts i, si tots estan activats, indica que l'element forma part del conjunt.

Com a desavantatge, es poden produir falsos positius. Si, per casualitat, els bits d'un element han estat activats per altres elements, o dos elements comparteixen el mateix resultat a les funcions de *hash*, el resultat serà que aquest element forma part del conjunt, cosa que no és certa. En un filtre de Bloom simple, com el que hem implementat, no es poden detectar, tot i que es pot dissenyar perquè el percentatge en sigui molt reduït.

6.2 Característiques

El nombre de bits del vector i la quantitat de funcions de *hash* necessàries es poden calcular amb dues senzilles fórmules matemàtiques.

Suposant que volem que funcioni correctament per un diccionari de 5000 elements (n = 5000) i tingui una ràtio de falsos positius de l'1% (p = 0.01), tenim que:

$$m = \frac{-n * \ln(p)}{\ln(2)} \approx 47926$$

$$k = \ln(s) * \frac{n}{m} = 6.64 \approx 7$$

, on m és el nombre de bits del vector i k el nombre de funcions. Així doncs, tindríem un vector de 47926 bits i 7 funcions de *hash* diferents.

6.3 Funcionament de la classe *BloomFilterDictionary*

El filtre de Bloom està implementat dins de la classe *BloomFilterDictionary*, al fitxer *diccBloomFilter.cc*.

La classe té dos atributs privats: un vector de booleans i la seva mida, un enter.

Tenim 7 funcions de *hash* privades, de les quals 4 estan basades en un polinomi (en varia el coeficient), i 3 estan basades en una funció *hash* d'enllaçament de fitxers objecte força comuna en C++.

En algun cas, degut a l'*overflow*, el resultat de les funcions és negatiu. Això provocaria un error, car els índexs dels vectors no poden ser negatius. Per solucionar-ho, l'hem passat per la funció de valor absolut.

```
int BloomFilterDictionary::h1 (const string& s) {
    long long int hash = 0;

    for (int i = 0; i < s.size(); ++i) {
        int valor = (int)s[i];
        hash = 11*hash + valor;
    }

    return abs(hash % mida);
}

int BloomFilterDictionary::h5 (const string& s) {
    long long int hash = 0;

    for (int i = 0; i < s.size(); ++i) {
        int valor = (int)s[i];
        hash = 16*hash + valor;

        unsigned long int g = hash & 0xF0000000L;
                if (g != 0) hash = pow(hash, g >> 24);
                hash = hash & (~g);
    }

    return abs(hash % mida);
}
```

A més a més, disposem de dues funcions públiques per interactuar amb el filtre de Bloom des de l'exterior de la classe: *afegir* i *comprovar*.

La primera afegeix un element al filtre i la segona en comprova la pertinença.

Finalment, disposem de dos constructors. El constructor per defecte crea un filtre per un diccionari d'unes 5000 paraules. L'altre constructor rep per paràmetre la quantitat d'elements a guardar-hi, i calcula la mida del vector necessària emprant la fórmula de l'apartat anterior.

6.4 Anàlisi de cost temporal i espacial

En aquest apartat fem un anàlisi teòric del cost espacial i temporal del filtre de Bloom.

6.4.1 Cost espacial

El principal avantatge del filtre de Bloom és l'estalvi espacial. Només necessita una matriu de booleans i, per tant, el seu cost és d'm bits.

De fet, un filtre amb una ràtio de falsos positius de l'1% i un nombre òptim de funcions de *hash* només necessita 9.6 bits per element, a diferència d'altres estructures de dades, que necessiten guardar l'element en sí i, en cas de ser gran, necessita un nombre més elevat de bits.

6.4.2 Cost temporal

Tant el cost d'afegir com el de comprovar són constants i no depenen del nombre d'elements que ja formen part del conjunt. Aquest cost serà el de la funció de *hash* més extensa. Totes les funcions depenen de la quantitat de caràcters de l'*string*.

Per tant el cost temporal del filtre depèn de la mida de l'element que es vol afegir/comprovar.

6.5 Anàlisi del ràtio de falsos positius

Hem dissenyat un programa (`experiment_falsos_positius.cc`) per testar que el filtre funciona correctament. Aquest experiment no s'ha de prendre com un apartat del tot rigorós, sinó més aviat orientatiu.

Tenim tres fitxers que emmagatzemen la llista completa de paraules de tres llibres: Mare Balena (en català), Dracula (en anglès) i El Quijote (en castellà). Tots tres són de diferent mida i s'avaluen per separat. Per cada paraula del fitxer, s'afegeix al filtre i es comprova si totes les paraules del fitxer en formen part o no.

Tot i que, teòricament, els falsos negatius no són possibles en un filtre de Bloom, també els comptabilitzem.

Els resultats que hem obtingut són:

Nom	Mida	Falsos negatius	Falsos positius	Comprovacions	%
Mare Balena	5051	0	17391	25512601	0,068
Dràcula	9421	0	399676	88755241	0,450
El Quijote	22444	0	28500774	503733136	5,657

Figure 10: taula falsos positius.

Gràcies a aquests resultats, que ens semblen prou desitjables pel nostre projecte, hem pogut comprovar que no es produeixen falsos negatius i que el ràtio de falsos positius és admissible. Per tant, podem fer servir aquesta implementació del filtre per resoldre la *Supersopa*. Un anàlisi més precís sobre aquest tema es pot trobar a l'apartat de Resultats.

7 Taula Hash

7.1 Introducció

Una taula *hash* és una estructura de dades que mapeja claus amb valors. Utilitza una funció, normalment anomenada *hash function*, que calcula un índex per cada clau. A partir d'aquest índex podem emmagatzemar el valor a a ubicació adequada de la taula *hash*. Es diu que s'ha produït una col·lisió si dues claus diferents obtenen el mateix índex. Per resoldre les col·lisions fem servir el mètode de *double hashing* La tècnica de *double hashing* consisteix en aplicar una segona funció *hash* a la clau per obtenir un segon índex, aleshores sumem el primer índex amb el segon multiplicat per i i fem el mòdul d'aquest resultat amb la mida de la taula hash.

(hash1(key) + i * hash2(key)) tableSize)

Podem anar incrementant el multiplicador del segon índex i fins que no es produeixi una col·lisió i així sigui possible emmagatzemar el valor adequadament.

Tot el benefici d'utilitzar una taula *hash* es deu al seu temps d'accés molt ràpid. Tot i que pot haver-hi alguna col·lisió, si triem una bona funció *hash* aquesta possibilitat és gairebé nul·la. Així, de mitjana, la complexitat del temps és un temps d'accés constant O(1).

7.2 Funcionament de la classe

La Taula de Hash amb *double hashing* està implementada dins de la classe *HashTableDictionary*, al fitxer *diccDHashing.cc*.

La classe té quatre atributs privats, tres enters, la mida de la taula, el primer nombre primer més petit que la mida i el número de col·lisions. També té un iterador d'un *array* de *strings*.

Tenim 5 funcions privades, dues de les quals són funcions de hash. La primera consisteix. La segona només l'utilitzem en cas de col·lisió. Utilitza el nombre primer nombre primer més petit que la mida de la taula i fa el mòdul de la suma de caràcters i aquest primer i li resta a aquest mateix primer. Les altres tres funcions són per calcular primers. *esPrimer* retorna un booleà dient si el paràmetre és un nombre primer, *seguentPrimer* va incrementant un número fins que troba el següent primer i *abansPrimer* va disminuint el número fins que troba l'últim primer.

```
int HashTableDictionary::hash1 (string key) {
    int length = key.length();
    int sum = 0;
    for(int i = 1; i <= length; i++) {
        sum += (i*(int)key[i]);
    }
    return (sum % tableSize);
}


int HashTableDictionary::hash2 (string key) {
```

```
    int length = key.length();
    int sum = 0;
    for(int i = 0; i < length; i++) {
        sum += (int)key[i];
    }
    return (primerPetit - (sum % primerPetit));
}
```

A la part pública hem creat dues funcions, *afegir* i *comprovar*, que ens permeten relacionar la classes amb la resta.

La funció *afegir* té com a paràmetre la paraula que volem afegir a la taula de *hash*. Li apliquem la primera funció *hash* i si no hi ha col·lisió afegim la paraula a la taula a la posició que indica l'índex. Si hi ha col·lisió utilitzem *double hashing*, que ja l'hem explicat abans, fins que trobem una posició per poder emmagatzemar la paraula. Durant el procés de *double hashing* comptem el nombre de col·lisions produïdes i ens guardem el seu valor màxim, que ens serà molt útil a la funció de *comprovar*.

```
void HashTableDictionary::afegir (string key) {
    int index = hash1(key);
    if (hashTable[index] != "nnnn") {
        int index2 = hash2(key);
        int i = 1;
        bool found = false;
        while (!found) {
            int newIndex = (index + i * index2) % tableSize;
            if (hashTable[newIndex] == "nnnn") {
                hashTable[newIndex] = key;
                if(i > maxcolision) maxcolision = i;
                found = true;
            }
            i++;
        }
    }
    else hashTable[index] = key;
}
```

La funció *comprovar* té com a paràmetre la paraula que volem comprovar si existeix a la taula de *hash*, aleshores comencem aplicant-li la primera funció hash i amb l'índex que ens torna anem a la ubicació exacta de la taula, comprovem si el valor que hi ha emmagatzemat a aquella ubicació és el mateix que el que ens passen com a paràmetre, si ho és la funció retorna que ho ha trobat. Si no ho és, apliquem *double hashing* fins que el valor emmagatzemat sigui el mateix que el paràmetre o i sigui el nombre màxim de col·lisions, que hem calculat anteriorment.

Si apliquem *double hashing* tantes vegades com el nombre màxim de col·lisions i segueix sense coincidir el valor emmagatzemat amb el paràmetre, aleshores retornem que no l'hem trobat, és a dir, la paraula no existeix.

```
bool HashTableDictionary::comprovar (string s) {
    int index = hash1(s);
    if (hashTable[index] != s) {
        int index2 = hash2(s);
        int i = 1;
        while (i <= maxcolision) {
            int newIndex = (index + i * index2) % tableSize;
            if (hashTable[newIndex] == s) return true;
            i++;
        }
        return false;
    }
    return true;
}
```

Per últim, disposem de dues constructores. La constructora per defecte crea una taula de *hash* per un diccionari d'unes 100 paraules, que només s'utilitza quan es crea la sopa i l'altra constructora rep per paràmetre la quantitat d'elements a guardar-hi, i calcula la mida necessària de la taula.

7.3 Nombres Primers com a mida de la taula

Generalment, les funcions *hash* calculen un valor enter a partir de la clau. Per assegurar que aquest valor enter es troba dins de la longitud de la taula de hash, el més comú és calcular el mòdul de l'enter i la longitud de la taula.

```
int HashTableDictionary::hash1 (string key) {
    tableSize = 15;
    int length = key.length();
    int sum = 0;
    for(int i = 1; i <= length; i++) {
        sum += (i*(int)key[i]);
    }
    return (sum % tableSize);
}
```

En aquest exemple, on hem definit la longitud per defecte a 15, els enters que són múltiples de 3 o múltiples de 5, però no múltiples de 15, es dividiran en índexs de 3 i 5, respectivament. Per exemple, les claus que produeixen enters de 0, 15, 30, ... se'ls assignarà índex 0, les claus que produeixen enters de 3, 18, 33, ... se'ls assignarà índex 3, i les claus que produeixen enters de 5, 20, 35, ... se'ls assignarà l'índex 5.

En altres paraules, cada enter que comparteixi un factor comú amb la longitud serà resumit en un índex que és un múltiple d'aquest factor.

Una taula de *hash* amb una longitud de nombre primer produirà la distribució més àmplia dels enters als índexs. En l'exemple podem veure que surten patrons per cada factor de la longitud, 15. Així que ens beneficia triar una longitud que té el menor nombre de factors. Els nombres primers només són divisibles per 1 i per si mateixos, per tant, si definim la longitud de la taula de *hash* a un nombre primer gran, es reduirà molt el nombre de col·lisions.

7.4 Cost

Quan parlem d'una taula de *hash*, normalment mesurem el seu rendiment amb el cost de buscar un element a la taula. En la majoria de configuracions de *hashing*, podem demostrar que és O(1), ja que amb la funció de *hash* trobem l'índex per anar a la ubicació exacta. Així que el temps d'execució esperat d'una consulta de taula de *hash* és O(1), sense importar el número d'elements que hi hagi a la taula.

Però això només és així si no es produeix cap col·lisió. En canvi, en casos particulars tenim algunes col·lisions, cosa que fa que tinguem pitjors temps de cerca, el qual com a màxim és O(n), ja que hauria de recórrer linealment tota la taula, sent n el número d'elements a la taula.

En conclusió, el cost en el pitjor cas és O(n) i en el millor cas el es O(1).

8 Classe *SuperSopa*

A la classe *Supersopa* s'implementen els mètodes per generar i manipular les super sopes, que es criden des dels programes dels experiments.

8.1 Mètode per generar sopes

Per crear sopes de paraules utilitzem la seguent funció:

Listing 1: Funció per crear sopes

```
void superSopa::generarSopa (const vector<string>& dicc, Sopa& sopa)
```

A aquesta funció li passem dos paràmetres per referència, un vector d'*strings* amb les paraules que volem col·locar a la sopa, i la sopa. Un cop dins la funció el primer que fem és crear una matriu de booleans de la mateixa mida que la sopa, per poder saber quines posicions ja hem visitat, i fem un primer bucle per intentar col·locar les 20 paraules.

Per cada paraula, recorrem la sopa fins a trobar una posició buida (o amb la mateixa lletra que la primera lletra de la paraula) on puguem començar a col·locar la primera lletra. Quan la trobem, marquem aquesta posició com a visitada al vector que hem creat al principi i passem la paraula, un nou enter anomenat l que serà 1 (explicarem mes tard perquè serveix), la posició i la sopa a la funció *colocarParaula* que acabarà de posar la resta de les lletres la paraula.

Si aquesta nova funció ens retorna que s'ha pogut acabar de col·locar la paraula correctament, passem a la següent paraula. Però si, per al contrari, ens retorna que no s'ha pogut acabar de col·locar correctament, tornem a buscar una primera posició on començar a col·locar la paraula, repetim aquest procés fins que es pugui col·locar tota la paraula correctament o fins que no quedin possibles posicions inicials. Finalment, quan ja hem col·locat totes les 20 paraules omplirem els buits que ens queden a la sopa, passant-li la sopa a la funció *omplebuits* que genera caràcter aleatòriament i els col·loca als buits de la sopa.

Listing 2: Funció per col·locar paraules a la sopa

```
bool superSopa::colocarParaulaRec(const string& p, int l, int i, int j,
                                  Sopa& sopa)
```

Aquesta funció és recursiva, tenim com a cas base que si l, el paràmetre que ens passen equivalent al número de lletres que ja hem col·locat de la paraula, és igual a la mida, vol dir que hem col·locat totes les lletres de la paraula i retornem que la paraula s'ha pogut col·locar correctament.

Si no és el cas base vol dir que encara ens queden lletres de la paraula per col·locar. Aleshores a partir de la posició que ens han passat com a paràmetre, que és la posició de la lletra anterior, generem aleatòriament una nova posició contigua a la qual ens han passat, si és una posició buida (o amb la mateixa lletra que volem col·locar), marquem la posició com a visitada, col·loque'm la lletra i cridem recursivament a la funció passant-li com paràmetres l+1, la paraula, la nova posició i la sopa. Però si, per al contrari, si no és una posició buida tornem a generar una de nova. Si hem comprovat les 8 posicions contigües possibles i cap està buida retornarem que no és possible col·locar la paraula.

8.2 Mètodes de resolució de sopes

8.2.1 Vector ordenat

El mètode per resoldre sopes del vector ordenat, el qual es menciona dins el capítol corresponent, és un simple bucle doble que recorre totes les posicions de la sopa. A partir d'allà ja s'entra a les funcions de la classe *SortedVector* explicades anteriorment.

```
void superSopa::resoldre (SortedVector& d, Sopa& sopa) {
    d.iniciarResults();
    int l = 0 , r = d.getSize() - 1;
    int n = sopa.size();
    vector<vector<bool>> pos(n, vector<bool>(n,false));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int l = 0, r = d.getSize()-1, iterador = 0;
            d.buscarParaula(i,j,pos,l,r,iterador,sopa);
        }
    }
}
```

8.2.2 Taula Hash i Filtre de Bloom

Ambdues estructures de dades utilitzen el mateix mètode (sobreescrit) per resoldre la super sopa. Aquest mètodes és:

Listing 3: Funció de resolució de la Taula de Hash

```
int resoldre (HashTableDictionary& d, HashTableDictionary& pre, Sopa& so) {
    nTrobades = 0;
    n = so.size();
    sopa = so;
    d_hash = d;
    pre_hash = pre;
    matbool visitats;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            string p = "";
            p.push_back(sopa[i][j]);
            visitats = vector<vector<bool>>(n, vector<bool>(n, false));
            visitats[i][j] = true;

            resoldreRecursiuHash(visitats, p, i, j);
        }
    }

    return nTrobades;
}
```

L'equivalent pel filtre de Bloom és idèntic, però substituint la classe per la corresponent.

Tenen tres paràmetres: el diccionari de paraules correctes, el diccionari de prefixos i la sopa, una matriu d'n x n caràcters. Retornen el nombre de paraules trobades.

En primer lloc, es creen i inicialitzen les variables i estructures de dades necessàries: el nombre de paraules trobades, la mida de la sopa, la sopa, els dos diccionaris i una matriu de booleans, necessària pel mètode recursiu que expliquem més endavant.

Per cada casella, s'executa un mètode recursiu que aplica *backtracking* per resoldre la sopa, que es pot veure a continuació:

Listing 4: Funció recursiva per comprovar les paraules

```
void resoldreRecursiuHash (matbool& v, string paraula, int i, int j) {
    for (auto dir : DIR) {
        int i2 = i + dir.first;
        int j2 = j + dir.second;

        if (comprovarPosicio(i2, j2, v)) {
            string paraula2 = paraula;
            paraula2.push_back(sopa[i2][j2]);

            if (d_hash.comprovar(paraula2)) {
                ++nTrobades;
            }
            if (pre_hash.comprovar(paraula2)) {
                v[i2][j2] = true;
                resoldreRecursiuHash(v, paraula2, i2, j2);
                v[i2][j2] = false;
            }
        }
    }
}
```

Aquesta funció s'encarrega de comprovar totes les paraules existents a la sopa. Rep una possible paraula, la posició de la casella de l'última lletra i una matriu de booleans on es guarda la informació de quines caselles (lletres) de la sopa ja s'han fet servir i, per tant, no es poden tornar a afegir a la paraula. A partir d'això, genera les paraules possibles del seu voltant i torna a cridar-se a ella mateixa de forma recursiva.

Per cada una de les 8 possibles direccions en què pot avançar la paraula (amunt, avall, dreta, esquerra i les 8 diagonals), s'aplica el mateix algorisme. En primer lloc, es genera la nova paraula (*paraula2*), afegint-li a la rebuda per paràmetre la lletra de la casella nova que es vol comprovar. A continuació, es comprova si aquesta nova paraula és correcta, i es comptabilitza en cas cert.

Finalment, es comprova si és prefix d'alguna paraula correcta. Si ho és, es marca la nova casella com a visitada, es torna a cridar la funció per avançar en la cerca, i després es desmarca la casella. En cas contrari, significa que la seqüència de lletres generada fins ara no forma part de cap paraula correcta i, per tant, no cal continuar en aquesta direcció. Es descarta i es torna enrere en el *backtracking*.

En definitiva, es tracta d'una cerca en profunditat.

8.2.3 Trie

El mètode de resolució pel *Trie* és, essencialment, el mateix que el de l'apartat anterior. L'única diferència destacable és que no es requereix un diccionari addicional per emmagatzemar els prefixos: la pròpia implementació del *trie* en permet la comprovació conjuntament amb les paraules correctes.

9 Experimentació

Nosaltres hem decidit separar els experiments amb la creació de les sopes. És a dir, tenim un programa anomenat *generar.cc* que genera 100 sopes de mides des de 10x10 a 55x55 caràcters i les guarda a un fitxer extern.

Pel que fa a la part de l'experimentació, tenim quatre programes, estructurats de la mateixa manera, que rebran aquestes sopes com a *inputs* i cridaran a les funcions necessàries per resoldre-les.

9.1 Funcionament *generar.cc*

El programa *generar* selecciona un subconjunt de 20 paraules del diccionari i les guarda en una estructura de tipus vector (anomenat p). Després de seleccionar les 20 paraules que col·locarem a les sopes, passem a generar-les.

```
for (int n = 10; n <= 55; n += 5) {
    for (int j = 0; j < 10; ++j) {

        Sopa matriu = Sopa(n, vector<char>(n, '#'));
        super_sopa.generarSopa(p, matriu);
        escriure_fitxer(matriu, fw);
    }
}
```

Com veiem en la figura en el fragment de codi anterior, generarem 100 super sopes de mides entre 10x10 i 55x55. Per cada mida generarem 10 sopes i augmentarem de cinc en cinc, és a dir, tindrem sopes de mida 10, 15, 20,...

Podeu trobar explicat el funcionament de *generarSopa* a l'apartat anterior, on parlem de la classe *SuperSopa*. Per poder utilitzar correctament el programa, només caldrà executar '`make`' a la consola. Finalment, trobareu les sopes utilitzades per els tres diccionaris a la carpeta *sopes*.

9.2 Funcionament de la part experimental

Com hem comentat abans, hem decidit realitzar quatre mètodes diferents, un per cada mètode. Per poder-los utilitzar heu d'executar '`make experiment_x.exe`' a la consola, on x és el nom del mètode a utilitzar.

El programa imprimirà (en un fitxer anomenat *resultatsX.txt*) el temps en guardar el diccionari a la estructura de dades i per cada sopa: el nombre de la sopa (1, 2, ..., 100), la seva mida (10, 15, ..., 55), el temps (en μs) i el nombre de paraules trobades.

Tots quatre programes segueixen la següent estructura:

1. Llegeix el diccionari i el guarda a la estructura corresponent del mètode.
2. Llegeix la super sopa i la guarda en una matriu de caràcters.
3. Resolem la sopa llegida 10 vegades i guardem el nombre de paraules trobades i el temps emparat en cada resolució.
4. Després de resoldre-la 10 vegades, imprimim el nombre de paraules trobades i la mitjana del temps.
5. Repetim des de el punt dos amb la següent sopa fins resoldre les 100.

10 Resultats

Al *sorted vector* hem pogut comprovar que el temps dedicat a trobar les paraules es manté pels tres diccionaris diferents, tal i com es pot observar a la *Figure 11*.

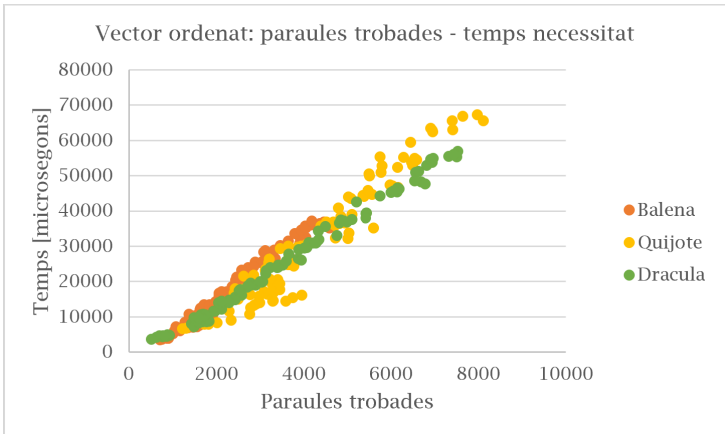

Figure 11: Relació entre les paraules trobades a cada sopa i el temps dedicat a resoldre-la al mètode *Sorted Vector*.

Això sí, el nombre de paraules trobades varia segons la quantitat de paraules del diccionari usat. Aquest resultat té sentit perquè si el diccionari té més paraules hi haurà més probabilitat que una combinació de paraules aleatòria de la sopa sigui dins el diccionari.
La resolució amb *Trie* presenta el mateix esquema, el temps per trobar les paraules de la sopa es semblant en cada diccionari. Es pot veure a la *Figure 12* .

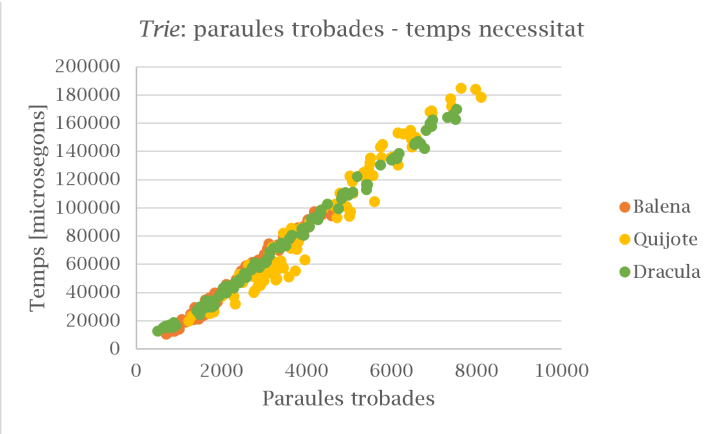

Figure 12: Relació entre les paraules trobades a cada sopa i el temps dedicat a resoldre-la al mètode *Trie*.

En canvi, als mètodes del filtre Bloom i el doble *hash* trobem més variació entre els temps emprats per resoldre cada diccionari i el nombre de paraules trobades. És fàcilment deduïble mirant els gràfics de la *Figure 13* i la *Figure 14*.

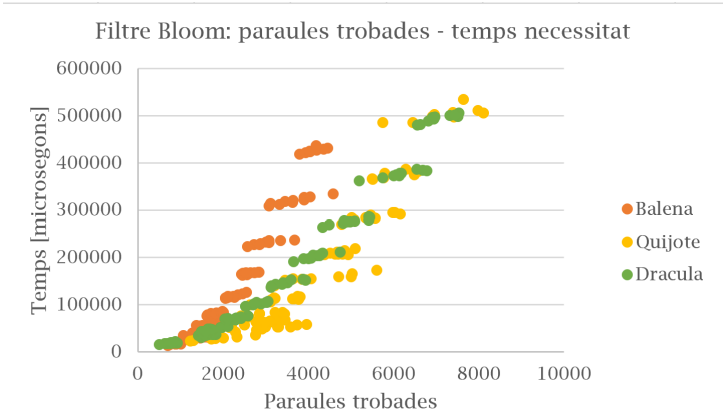

Figure 13: Relació entre les paraules trobades a cada sopa i el temps dedicat a resoldre-la al mètode Filtre Bloom.

A més, al mètode doble *hash* trobem un gran increment de temps al diccionari en castellà. Aquest fet podria tenir a veure amb que és el que conté més paraules, més del doble que els altres dos.

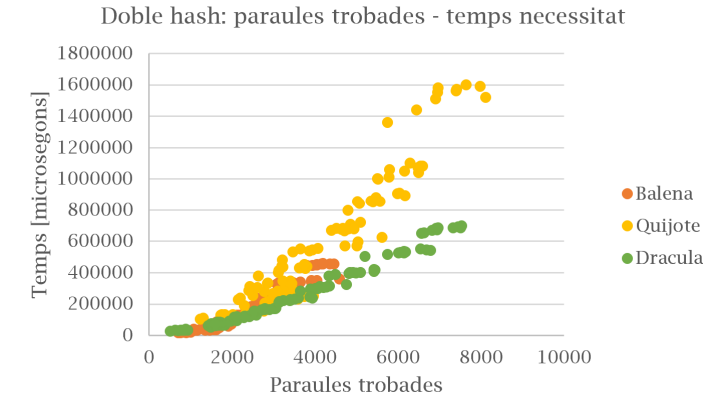

Figure 14: Relació entre les paraules trobades a cada sopa i el temps dedicat a resoldre-la al mètode *Double Hash*.

També hem fet comparacions entre els quatre mètodes d'emmagatzemar el diccionari, el resultat dels quals es pot veure a la *Figure 15*. Veiem una clara diferència de la tendència que prenen el filtre Bloom i el doble *hash* respecte el vector ordenat i el *Trie*. Aquest esquema es repeteix també als altres dos diccionaris.

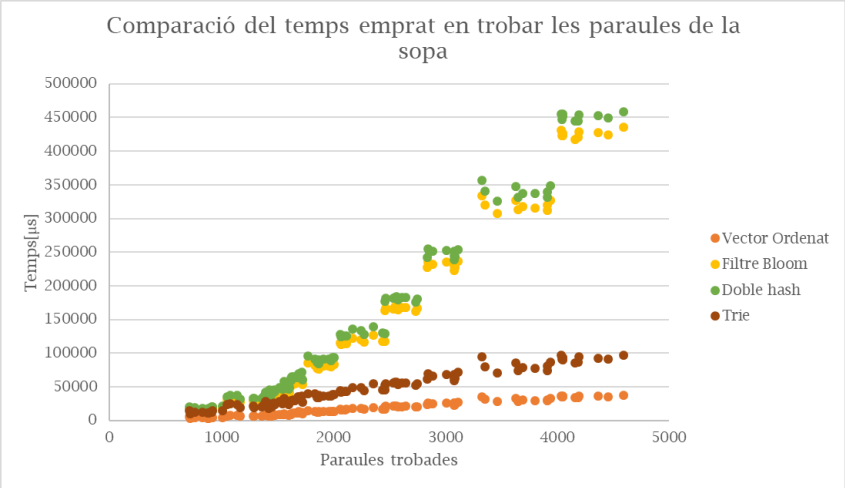

Figure 15: Comparació dels quatre mètodes entre el temps per trobar un determinat nombre de paraules al diccionari *mare-balena-vocabulary-3.txt*.

Els resultats estructurats en taules es troben a l'Annex desglossats.

10.1 Filtre de Bloom: Falsos positius

A part d'analitzar el temps d'execució de la resolució de la *Supersopa* amb el filtre de Bloom, també n'hem analitzat la quantitat de falsos positius, característica important d'aquesta estructura de dades.

Per cada sopa resolta, n'hem comparat la quantitat de paraules trobades amb el mètode del vector ordenat i amb el mètode del filtre de Bloom. Els resultats es poden observar al gràfic següent:

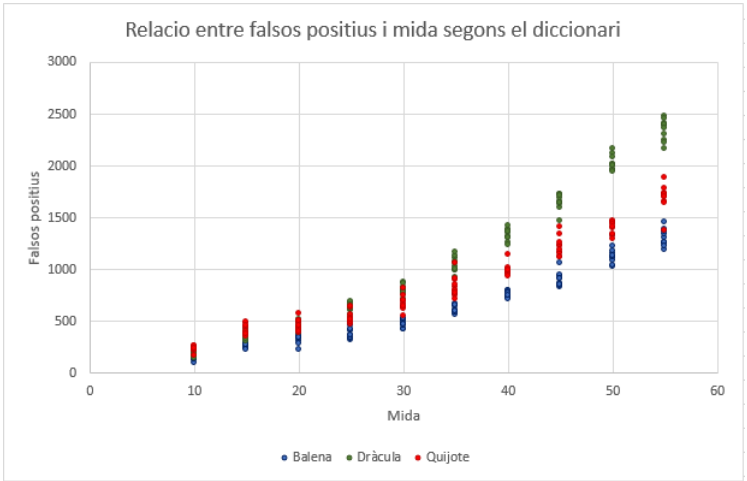

Figure 16: Relació entre els falsos positius i la mida de la sopa, segons el diccionari emprat.

Com es pot observar el gràfic, el nombre de falsos positius és directament proporcional a la mida de la sopa. Té sentit pel fet que, com més gran la sopa, més possibles paraules conté i, per tant, més comprovacions s'han de fer.

També se'n pot extreure que depèn del diccionari emprat. Com més paraules conté el conjunt de paraules correctes, més possible és que, de casualitat, dues paraules comparteixin bits al filtre, o que un conjunt de paraules activin els bits necessaris perquè una altra es presenti com a positiva.

Com a resultat sorprenent, el diccionari Dràcula produeix més falsos positius que el Quijote, tot i contenir menys de la meitat de paraules. Això pot ser degut a la composició dels dos diccionaris.

10.2 Conclusió sobre els resultats

Sorprenentment, el vector ordenat ha donat els resultats més ràpids de l'experiment. La principal conclusió que en traiem és que aquest mètode és eficient, però què un factor clau de la rapidesa ha sigut l'adequació de l'algorisme creat a l'estructura del problema.

Sovint només pensem en una simple cerca dicotòmica per resoldre aquest tipus de problemes amb un vector. Aquesta vegada anar més enllà i buscar una manera d'implementar-la amb modificacions ha donat bons resultats.

Pel que fa l'espai, resoldre la sopa amb el vector ordenat també ha donat resultats positius, ja que no hem necessitat cap estructura de dades auxiliar per executar els algorismes amb èxit. L'única estructura de dades creada ha sigut un *map*, del qual podríem prescindir perfectament ja que la seva única funció és saber quines eren les paraules solució de la sopa, i l'enunciat no demana saber-ho.

Com a conclusió final del vector ordenat, creiem que aquesta estructura d'emmagatzemar el diccionari és molt bàsica i no està pensada per resoldre problemes d'aquest tipus però si s'implementa un algorisme que l'adapti a les condicions del problema podrem obtenir resultats molt eficients.

Pel que fa la implementació de la classe *diccTrie*, encara que és una estructura bastant adient pel tipus de problema esmentat, ha sigut el segon mètode més ràpid i no pas el primer. Ens esperàvem més bons resultats respecte a les cerques de paraules. Això es pot deure a la gran dimensió de l'arbre. Un *ternary search tree* és molt efectius quan es tracta amb paraules que comparteixen molts prefixos comuns i amb profunditats no gaire grans.

Una altra manera d'implementar aquest mètode podria ser tenint un *array* amb mida 26 (un arbre per lletra del abecedari) on hi guardaríem *ternary search trees*. Podríem aplicar les mateixes funcions *afegir, simplificar* i *comprovar*, ja que podríem accedir a l'índex de l'arbre sabent la lletra per la qual comença la paraula a afegir/comprovar. D'aquesta manera podríem obtenir arbres no tant profunds.

Una altra raó podria ser la manera en què llegim les paraules, ja que comencem a llegir per l'element del mig cap a l'últim element, i després llegim l'altra meitat. D'aquesta manera obtenim un arbre on l'arrel tindrà (possiblement) tres subarbres, però el seu subarbre dret només presentarà els subarbres dret i central i el mateix passarà amb el subarbre esquerre. És a dir, que els subarbres no estaran equilibrats. Aquest problema podria solucionar-se amb la implementació

del vector que hem comentat abans, però no neccessàriament, ja que depèn molt de les paraules que afegim.

Respecte al filtre de Bloom, aquesta estructura és molt eficient en espai i temps. Malgrat això, té la particularitat de produir falsos positius. Com podem veure en el segon apartat de resultats, l'estudi dels falsos positius, veiem que aquests creixen a mesura que la sopa també creix i, per tant, el nombre de paraules trobades també creix. Això es deu a que el marge d'error és més gran.

Finalment, hem calculat quin és el temps mitjà per trobar una paraula segons el diccionari usat. Veiem que els mètodes del vector ordenat i el *Trie* no pateixen variacions en el temps de modificació, independentment de la mida del diccionari. En canvi, el Filtre Bloom sembla tenir la tendència de ser inversament proporcional a la mida del diccionari. Per acabar, el temps de resolució d'una paraula amb el doble *hash* és directament proporcional a la mida del diccionari.

Temps mitjà per trobar una paraula [microsegons]	Català	Castellà	Anglès
Vector ordenat	7.07	6.93	6.70
Trie	19.56	20.18	20.78
Filtre Bloom	53.28	38.02	41.7
Doble *hash*	58.17	129.62	63.87

Figure 17: Temps per trobar una paraula a la sopa segons cada tipus de diccionari.

11 Conclusions

Finalment, ens agradaria concloure el nostre projecte dient que ens sentim satisfets amb les implementacions dels quatre mètodes. Aquest projecte ens ha servit per aprendre com aplicar teoremes teòrics a la practica i resoldre els problemes que poden sorgir.

Per exemple, un dels problemes trobats al realitzar aquest projecte va ser al realitzar la cerca en un *ternary search tree*. Ja que no sabíem quants caràcters podia presentar un node (treballàvem amb un arbre simplificat). Primer vam pensar en passar dos índexs, l'index i, que ja passem, i un segon índex j que indiqués el caràcter que observem de la paraula que guarda el node n. D'aquesta manera, a cada crida que canviéssim de fill, l'iniciaríem a 0, i a les crides al mateix fill l'incrementaríem una posició.

Però no ens va acabar d'agradar ja que fèiem moltes voltes. Vam pensar també d'incorporar l'index j com a atribut d'un node de l'arbre. Però cada cop que visitaves aquell node i avançaves per aquest, hauries de tornar a deixar-lo a zero per la següent cerca. No ens va semblar gaire bona idea, ja que no tenia gaire sentit, de manera que també la vam descartar.

Finalment, vam veure que el problema es resolia veient si una paraula contenia l'altre. És per això, que vam implementar la funció *inclouParaula*.

Un altre problema que ens vam trobar va ser amb les implementacions del *filtre de Bloom* i amb la *Taula Hash*. Originalment, pel filtre de Bloom i la taula de *hash*, el mètode de resolució comprovava totes les paraules possibles existents. Quant vam començar a experimentar amb les classes, ens vam adonar que no era viable, ja que quan creixia la mida de la sopa el temps de resolució era massa elevat.

Per optimitzar-ho, vam fer servir una altra estructura de dades on es guardaven tots els prefixos de totes les paraules del diccionari. Així, quan es comprovava una seqüència de lletres de la sopa, si no era prefix de cap paraula correcta es descartava aquesta possible solució i no seguia buscant per aquest camí.

Per acabar, creiem que aquest treball ens ha servit tant per conèixer noves estructures de dades com per enfrontar-nos a nous reptes i a noves maneres d'implementar problemes. Ja que no és el mateix la explicació d'un algorisme a fer-lo funcionar.

12 Bibliografia

1. Campos, Javier (s.f.) *2.4 Arboles digitales, tries y Patricia* [Diapositives].http://webdiis.unizar.es/asignaturas/TAP/material/2.4.digitales.pdf
2. *Implementación Trie en C - Insertar, Buscar y Eliminar.* (s. f.). Recuperat 12 d'octubre de 2022, de https://www.techiedelight.com/es/trie-implementation-insert-search-delete/
3. *Implementación en C++ de la estructura de datos Trie.* (s. f.). Recuperat 12 d'octubre de 2022, de https://www.techiedelight.com/es/cpp-implementation-trie-data-structure/
4. *10 formas de convertir un carácter en una cadena en C++.* (s. f.). Recuperat 13 d'octubre de 2022, de https://www.techiedelight.com/es/convert-char-to-string-cpp/
5. *Ternary Search Trees.* (s. f.). Dr. Dobb's. Recuperat 10 d'octubre de 2022, de https://www.drdobbs.com/database/ternary-search-trees/184410528#
6. Wikipedia contributors. (2022, 12 septiembre). *Trie.* Wikipedia. Recuperado 15 de octubre de 2022, de https://en.wikipedia.org/wiki/Trie
7. *Merge Sort (With Code in Python/C++/Java/C).* (s. f.). Recuperat 13 d'octubre de 2022, de https://www.programiz.com/dsa/merge-sort
8. *std::map - cppreference.com.* (s. f.). Recuperado 15 de octubre de 2022, de https://en.cppreference.com/w/cpp/container/map
9. *Create LaTeX tables online – TablesGenerator.com.* (s. f.). Recuperat 17 d'octubre de 2022, de https://www.tablesgenerator.com/
10. *Jutge.org - P84219_en - First occurrence.* (s. f.). Recuperat 17 d'octubre de 2022, de https://jutge.org/problems/P84219_en
11. *Hash Table* https://stackoverflow.com/questions/2771368/can-hash-tables-really-be-o1
12. Departament de Ciències de la Computació. (2017, octubre). *Correctesa de Programes Iteratius i Programació Recursiva.* [Document]. https://www.cs.upc.edu/pro2/data/uploads/it_rec_corr.pdf
13. *Double Hashing* http://courses.washington.edu/css502/zander/Notes/09hash.pdf
14. Roura, Salvador (s. f.). *Eficiència d'algorismes.* [Document]. https://www.cs.upc.edu/eda/data/uploads/eficiencia.pdf
15. Rodrígez, Enric (s. f.). *Correctesa i Anàlisi del Cost de l'Algorisme d'Euclides.* [Document]. https://www.cs.upc.edu/eda/data/uploads/gcd.cat.pdf
16. Implementing a simple, high-performance Bloom filter in C++ https://daankolthof.com/2019/05/06/implementing-a-simple-high-performance-bloom-filter-in-c/
17. Hash function https://en.wikipedia.org/wiki/Hash_function
18. Bloom Filter https://en.wikipedia.org/wiki/Bloom_filter
19. Bloom Filters - Introduction and Implementation https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

20. *Hash Table* https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-2.html

21. How many hash functions does my Bloom filter need? https://stackoverflow.com/questions/658439/how-many-hash-functions-does-my-bloom-filter-need

22. *Prime size* https://stackoverflow.com/questions/3980117/hash-table-why-size-should-be-prime

23. *Prime size* https://stackoverflow.com/questions/3980117/hash-table-why-size-should-be-prime

24. University of California San Diego *Lecture 16 - Hashing* https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16.html

25. Bloom Filter https://brilliant.org/wiki/bloom-filter/

26. Wikipedia contributors. (2022, 5 junio). *Cerca binària*. Viquipèdia, l'enciclopèdia lliure. Recuperado 21 de octubre de 2022, de https://ca.wikipedia.org/wiki/Cerca_bin%C3%A0ria

27. *Hash Functions* https://www.digitalocean.com/community/tutorials/hash-table-in-c-plus-plus

13 ANNEX: Informació addicional

Podreu trobar les taules resultants de l'experimentació al fitxer *Resultats.xlsx* adjunt al document. Pel que fa el codi complet i les instruccions per executar-lo, el trobareu al repositori de *Github* del següent enllaç: `https://github.com/nmayol/Super-Sopa`. Les instruccions per executar el codi es troben al fitxer *README.md*.