Super Sopa

Projecte d'Algorísmia

Grup 5

Agnès Felip i Díaz

Joan Casahuga Altimiras

Marta Granero i Martí

Miquel Torner Viñals

Departament de Ciències de la Computació

Universitat Politècnica de Catalunya

Octubre, 2022

Índex

1.Introducció

Al llarg de la nostra vida, molts de nosaltres ens hem vist immersos en situacions en què, volent passar una estona entretinguda o menys tediosa hem volgut posar a prova el nostre enginy amb un passatemps.

I és que de passatemps n'existeixen una bona pila, entre els més coneguts tenim el dòmino, els mots encreuats, el sudoku, la sopa de lletres o bé els jeroglífics, entre molts altres. Cal no perdre de vista, que també n'han sorgit de nous, com és el cas del WordleCAT, que té l'objectiu de trobar un mot diari en un seguit d'intents, o bé el famós passatemps nascut recentment com és el cas del Paraulògic.

En aquest projecte, però, el nostre propòsit no serà passar el temps amb cap dels jocs mencionats anteriorment, sinó que estenent el passatemps de cercar paraules en una sopa de lletres, ara ho voldrem fer sobre una sopa de lletres **estrambòtica**, anomenada *Super Sopa*.

Per poder estudiar aquest joc de cercar les paraules que s'amaguen dins de la *Super Sopa,* el nostre projecte dissenyarà, implementarà i analitzarà diverses aproximacions que ens permetran cercar totes les paraules que s'hi troben ocultes.

Algunes tècniques ho faran amb més o menys enginy i encert, i és per això, que el nostre objectiu consistirà en: donades diferents tècniques basades en estructures de dades i algorismes **realitzar una validació experimental de l'eficiència i l'aplicabilitat de cada tècnica** a l'hora de cercar les paraules a la *Super Sopa*. Les tècniques emprades han set: el vector ordenat, la *Trie*, el filtre de *Bloom* i el *double hashing*.

En els següents capítols, donarem la descripció del joc de la *Super Sopa* i explicarem com hem dissenyat el generador de la sopa de lletres **estrambòtica** que ens ha permès crear el tauler sobre el que aplicarem cada tècnica emprada. Seguidament, s'explicarà el mode d'exploració de la sopa que comarteixen tres de les tècniques emprades.

Tot seguit, continuarem amb l'explicació en detall de cada tècnica i n'explicarem la seva descripció, implementació, correctesa, complexitat i, depenent de la tècnica, se'n farà una

experimentació parcial per valorar com les diferents optimitzacions realitzades sobre aquestes han permès millorar significativament el temps de cerca de paraules a la *Super Sopa.*

El penúltim capítol previ a les conclusions consistirà en l'anàlisis i la validació experimental de l'eficiència i aplicabilitat dels diferents algorismes i ED. Aquest anàlisis ens permetrà extreure tot el suc discutint l'efectivitat de les diferents tècniques mitjançant els temps d'execució obtinguts pels diferents estudis realitzats.

Finalment, al capítol de les conclusions donarem les idees claus de l'estudi realitzat, sintetitzarem els diferents resultats obtinguts i parlarem de millores a tenir en compte de cara a futurs projectes.

2.Descripció del problema

Disposem d'una col·lecció de paraules la qual denominem *Diccionari*. Les paraules d'un *Diccionari* són successions de caràcters alfabètics. Una *paraula* no es pot trobar repetida dins d'aquest, per tant són úniques, i considerarem com a *paraula vàlida* a aquelles paraules que es troben dins del *Diccionari*.

La *Super Sopa* és un *tauler* de *n x n* caselles de lletres. Considerem com a caselles *adjacents* totes aquelles caselles que es troben contigües a una casella en qualsevol de les vuit direccions.

Les característiques especials de la *Super Sopa* i el *Diccionari* són les següents:

- Una *paraula* es considerarà dins el *taulell* si la successió de caràcters que la componen es troben de forma consecutiva en caselles *adjacents*. No és necessari que les paraules prenguin una sola direcció. Aquesta característica fa que la sopa de lletres prengui el nom **d'estrambòtica**.
- Les *paraules* d'una *Super Sopa* mai poden creuar-se amb sí mateixes i, per tant, causar cicles. És a dir, si tenim *n* i *m* on *n* > *m* no es pot formar la paraula de longitud *n* amb *m* caràcters.
- Es permet la superposició de *paraules*. Si tenim una paraula formada per una *paraula* prefix llavors s'haurien de poder trobar les dues.

L'objectiu del problema serà "experimentar amb diferents estructures de dades" per trobar de la forma més eficient possibles, totes les paraules vàlides que es troben ocultes a la Super Sopa mitjançant les tècniques que es presenten als capítols següents.

3. Generació i exploració de la *Super Sopa*

El generador de la *Super Sopa* és un programa creat amb l'objectiu de generar l'input per poder donar a cada tècnica l'entrada per poder-la fer córrer en cadascuna dels nostres diccionaris.

Les variables controlables del generador són la mida de la Super Sopa (quin valor prendrà N en el taulell NxN) i, el nombre de paraules del diccionari que es plantaran a la sopa, aquestes es seleccionaran de forma aleatoria del diccionari, i seran col·locades en posicions aleatòries amb orientacions aleatòries.

Als subcapítols que es troben a continuació, farem un repàs de les funcions auxiliars que ens permetran generar la *Super Sopa*.

3.1. Selecció del subset de paraules plantades

Per assegurar-nos del correcte funcionament dels algorismes de cerca crearem un subset de paraules que sabrem del cert que podem trobar a la *Super Sopa*, ja que són les primeres que col·locarem. A la pràctica veurem que el nombre de paraules que trobarem a la sopa serà molt més elevat que les paraules inicials que plantem, però gràcies a la 'inserció de les paraules plantades, podrem comprovar que els nostres algorismes efectuen la cerca de la forma correcte trobant com a mínim sempre el subset en qüestió.

Degut a això, si volem que una paraula es pugui trobar a la sopa aquesta ha de ser vàlida i, per tant, trobar-se al *Diccionari*. Per tant les paraules del subset s'extreuran del diccionari d'entrada i es farà un subset de mida n on n = |P|, sent P el subset de paraules resultants. Notem que la mida del subconjunt de paraules plantades P la llegirem com a argument de la terminal de comandes per validar que s'han trobat totes les que s'inserten a la sopa, podent escollir en una execució qualsevol el nombre de paraules que considerarem plantar a la *Super Sopa*.

```
void selectWordSet(int numberOfWords) {

    copyOfDic = copia del dictionary
    random_shuffle(copyOfDic)
    for (int i = 0; selectedWords.size() < numberOfWords; ++i) {
        insert.copyOfDic[i]
    }
}
```

Figura 3.1. *Pseudocodi algorisme Select Subset*

Suposant el cost del shuffle rand lineal a la mida del conjunt copyOfDic, la inserció de la paraula al subset, i l'obtenció d'una paraula del diccionari negligible el cost recau sobre el nombre d'iteracions del while que s'executaran o el shuffle rand. Per tant el cost serà $O(mida\ diccionari)$

3.2. Inserció d'una única paraula a la *Super Sopa*

La inserció única d'una paraula a la *Super Sopa* es fa mitjançant la funció `addWordToSoup(..)`. Aquesta té com a objectiu situar les paraules d'una forma totalment aleatoria dins la sopa. En primer lloc construim un vector de totes les posicions del taulell i a aquest vector hi apliquem un random_shuffle que ens servirà per desordenar els elements del vector que hi passem. Seguidament iterem sobre les posicions desordenades intentant fer que la paraula comenci des de la posició en qüestió aplicant `placeToSoup(..)` a cada posició. En el cas que retorni true acabem, en el cas contrari busquem una nova posició.

```
bool addWordToSoup(string word)
{
      vector<pair<int, int>> traceback
      vector<vector<bool>>   visitat(N, vector<bool>(N, false))
      vector<pair<int, int>> positions

      ∀i∈A  in range (0, soupSize.size()):
          ∀i∈A  in range (0, soupSize.size()):
              positions.push_back(make_pair(i, j))

      random_shuffle(positions)

      ∀{i,j}∈A  in range (0, positions.size()):
            traceback.push_back(make_pair(i, j));
            visitat[i][j] = true;
```

```
            if(placeToSoup(word, 0, i, j, traceback, visitat))
                  return true;

            visitat[i][j] = false;
            traceback.pop_back();

      return false;
}
```

Figura 3.2. *Pseudocodi algorisme addWordToSoup*

En cridar a `placeToSoup(..)` es comença a intentar col·locar la paraula work a la posició i i j, per tal d'intentar maximitzar l'aleatorietat s'obté un vector de desplaçament randomitzat i s'intenta continuar expandint a l'estil BFS en cadascuna d'aquestes direccions, cosa que només accepta en cas que no s'hagi col·locat una lletra de la mateixa paraula abans o bé que la següent posició sigui buida o tingui la mateixa lletra que li toca col·locar a la paraula. Un cop la longitud del camí comprovat és igual a la mida de la paraula es col·loca la paraula i es retorna true, si no és possible es retorna false.

```
bool placeToSoup(string word, int currentLetter, int i, int j,
vector<posicions>& traceback, Matriu bool visitat)

    if (mida de la paraula == currentLetter)
        ∀k∈ in range (0, currentLetter.size())
            soup[traceback[k].first][traceback[k].second] = word[k]
        return true;

    bool foundSpot = false;

    if (posicio buida o hi ha la mateixa lletra que volem afegir) {

        ∀i∈A  in getRandomDirections()
            next_pos = pair(i+dir.first, j+dir.second)

            if (posicio valida i no utilitzada) {
                traceback.push_back(next_pos);
                visitat[next_pos.first][next_pos.second] = true;

                foundSpot = foundSpot or placeToSoup(word,
currentLetter+1, i+dir.first, j+dir.second, traceback, visitat);

                visitat[next_pos.first][next_pos.second] = false;
                traceback.pop_back();
            }
            if (foundSpot) break;
```

```
        }
    }
    return foundSpot;
}
```

Figura 3.3. *Pseudocodi algorisme placeToSoup*

Notem que el cost del BFS és $O(E + V)$

3.3. Emplenament de lletres buides

Finalment acabarem emplenant tots els espais en què no s'han plantat paraules amb lletres aleatòries utilitzant la funció `fillSoupEmptySpaces()`. El cost d'això serà O(n²).

```
void fillSoupEmptySpaces() {
    ∀i∈ in range (0, soup.size()):
        ∀j∈ in range(0, soup[i].size()):
            if (soup[i][j] == '.')  soup[i][j] = rand()%26 + 'a';
    }
}
```

Figura 3.4. *Pseudocodi algorisme Select Subset*

3.4. *Exploració de la Super Sopa*

```
void exploreSoup() {
      creem una Matriu de booleans used(false)
      string s = "";
      ∀i∈ range (0, soupSize.size())
            ∀j∈ range (0, soupSize.size())
                        exploreSoupDeep(s, i, j, used, 1);
}
```

Figura 3.4.1. *Pseudocodi algorisme exploreSoup*

```
void exploreSoupDeep(string& s, int x, int y, Matriu<bool>& used, const
int total) {
      used[x][y] = true;
      s.push_back(soup[x][y]);

      if (total >= maxWordSize) {
          used[x][y] = false;
          s.pop_back();
          return ;
      }

      if (total >= minWordSize and search(s)) foundWords.insert(s);

      // Loops to All Directions
      ∀i∈ range (0, 8):
          updateX, updateY;
          if (min(x, y) >= 0 and max(x, y) < soupSize) {
                  if (!used[x][y]) {
                      used[x][y] = true;
                      exploreSoupDeep(s, x, y, used, total+1);
                      used[x][y] = false;
                  }
           }
           updateX, updateY;
      }
      s.pop_back();
     used[x][y] = false;
}
```

Figura 3.4.2. *Pseudocodi algorisme exploreSoupDeep.*

Per tal de recórrer tota la sopa i trobar les paraules que s'hi amaguen s'utilitza un BFS (Breadth-first search). Per començar aquesta cerca el que es fa és cridar al mètode

`exploreSoup()` el qual fa una crida a `exploreSoupDeep()` per cada posició de la sopa. Aquesta segona funció comprova si la paraula és dins del diccionari, i en cas afirmatiu s'afegeix a un `set` de paraules trobades. Seguidament expandeix la cerca en totes les direccions possibles. Notem que utilitza una matriu de booleans *used* la qual serveix per saber si s'ha visitat una cel·la de la matriu anteriorment i així evitar tornar-hi a passar i com a conseqüent no caure en bucles infinits.

3.4.1. Correctesa

Per demostrar la correctesa d'aquest algorisme voldrem:

- Si hi ha una paraula a la *Super Sopa* trobar-la sempre mitjançant la cerca en amplada. Com que visitarem cada posició de la Sopa, per cada posició mirarem si la paraula és dins del diccionari implementat per cada tècnica.
- Si ja hem trobat una paraula, el que voldrem és continuar buscant totes les que s'hi puguin amagar després d'haver fet la cerca anterior. Per tant, iniciarem el casos recursius, expandint la cerca en totes les direccions possibles, fins arribar altre cop al cas base, ja que voldrà dir que hem trobat un prefix que es troba al diccionari.
- Si no hi ha una paraula, d'igual forma recorrem totes les posicions del tauler per assegurar-nos que no ens queda cap cel·la de la matriu sense visitar, ens n'assegurarem fent servir la matriu *used*

Notem que el nostre algorisme sempre acabarà gràcies al fet que fem servir la matriu de booleans. Mai podrem tornar a una cel·la si ja s'ha visitat anteriorment durant la cerca.

A més a més no trobarà duplicats ja que les paraules trobades s'afegeixen en un `set`, però aquest no conté duplicats.

3.4.2 Cost

Complexitat temporal: $O(V + E)$ i complexitat espacial: $O(V^2)$

4. Vector ordenat

4.1. Descripció de l'algorisme

4.1.1. Comprovació de l'existència d'una paraula en un vector ordenat.

La comprovació d'existència es basa en un algorisme de cerca comú; el Binary Search. Binary Search és un algorisme de cerca que aprofita la característica que la estructura de dades es troba ordenada i reduir la complexitat temporal a $O(log\ n)$. El que fa l'algorisme és dividir l'interval de cerca per la meitat fins que es trobi la paraula o que els extrems de l'interval es creuin i per tant es tingui clar que la paraula no existeix.

4.1.2. Cerca d'una paraula en funció d'una posició de la *Super Sopa*.

Per la cerca d'una sola paraula s'utilitza l'algorisme d'exploració explicat a l'apartat 3.4.

4.3. Correctesa

La correctesa és la mateixa que a l'apartat 3.4.1, l'única cosa que varia és la implementació de la cerca d'una paraula. La correctesa del binary search recau en el fet que el vector es troba ordenat, al anar comparant índexs, si n < m es trobi en una posició.

4.4. Cost

La complexitat temporal és de $O((V\ +\ E)\ *\ log\ n)$ ja que es fa una crida a binary search per cada iteració.

4.2. Implementació d'un diccionari en un vector ordenat

Els vectors són contenidors seqüencials que poden canviar en mida. Un vector ordenat d'strings és una seqüència de contenidors ordenats alfabèticament. Com que considerem que els diccionaris que se'ns proporcionen ja es trobaran ordenats el cost de la creació del diccionari es correspondrà amb el cost de lectura i inserció d'un diccionari. Per una mostra d'un diccionari de mida *n* s'hauran de fer *n* insercions. A C++ el mètode d'inserció push_back és de temps constant pel que el cost de la creació del vector serà $O(n)$. En el

cas que els diccionaris que se'ns proporcionen no es trobessin ordenats el cost de la creació de estructura seria la suma de la creació del vector i l'ordenació mitjançant algun dels algorismes corresponents d'ordenació, per exemple el MSD Radix sort, que té cost $O(n * m)$ on *m* és la mitja de la longitud de les paraules del diccionari.

4.3. Experimentació

La cerca de paraules amb l'algorisme actual dona pas a uns temps desorbitats, això és degut a que es fa una cerca a cegues sense saber del cert si trobarà la paraula en la següent posició. Hem fet proves d'un una cerca completa d'un diccionari de mida 100 i variat la mida de la sopa:

Mida de la sopa	Temps d'execució sense pruning [ms]	Temps d'execució amb pruning [ms]
3x3	7,91 ms	4,28 ms
4x4	1424 ms	7,7344 ms
5x5	2650543 ms	12,62 ms

Figura 4.1. Mostra d'experimentació per vector ordenat sense poda

Degut a la baixa eficiència del Vector Ordenat vam decidir afegir una millora al codi, la tècnica *pruning* o *poda.* Podarem en funció de la possiblitat de que la nostra paraula sigui prefix. Això significa que la cerca exhaustiva podrà ser tallada i el temps d'execució millorarà considerablement.

En concret, creem i inicialitzem un vector de prefixos, i cada vegada que vulguem aplicar search des d'una posició en concret mirarem si és prefix d'alguna paraula o si directament és paraula.

Consequentment a l'experimentació final només utilitzarem el vectorOrdenat amb pruning.

5.Trie

La segona tècnica que hem utilitzat per resoldre el joc de la *Super Sopa* és mitjançant l'estructura de dades anomenada *Trie.* La *Trie* ens permetrà implementar el nostre diccionari en forma d'arbre on hi podrem fer insercions i consultes de forma senzilla i ràpida. Això és així, ja que la forma en com s'emmagatzema la informació ens permet fer cerques eficients de cadenes que comparteixen prefixos. És per això que a la *Trie*, també se la sol anomenar, *Prefix Tree.*

Abans d'explicar com es realitza la implementació del diccionari a la *Trie*, donarem una definició formal d'aquesta per posteriorment entendre com funciona i com guardar la informació del diccionari.

Definim la *Trie,* com un autòmat finit i determinista formada amb 5-tupla de la forma

$$T = (Q, \Sigma, \delta, q, F)$$

la qual ens permetrà guardar un conjunt de cadenes D.

Els components de la qual podem definir a continuació:

- Q, és el nostre conjunt d'estats, on cada estat ens representa unívocament un prefix de D
- Σ, és el nostre alfabet sobre el qual estan definides les cadenes. En el nostre cas és l'alfabet llatí de 26 símbols.
- δ, és la funció de transició, que la definim com una aplicació de la forma: $\delta: Q \times \Sigma \rightarrow Q$ tal que donat un estat qualsevol de Q i un símbol de l'alfabet Σ ens permetrà anar al següent estat donat aquest símbol. Per tant, podrem anar al següent node de l'arbre a partir del node anterior mitjançant l'aplicació de la funció de transició sobre un node.
- $q \in Q$, és l'estat inicial és la cadena buida," ", i.e el node arrel del *Trie*.
- $F \subseteq Q$, és el conjunt d'estats acceptadors, que en el cas de la *Trie* és igual a D

5.1 Descripció de l'algorisme

Una vegada ja hem vist com podem emmagatzemar les paraules del diccionari a la Trie, ens falta veure la forma de cercar paraules dins del tauler i veure si són cadenes del *Diccionari* que es troben dins l'estructura de dades.

Per poder fer la cerca ens ajudarem d'un algorisme que es compon de tres funcions. Les explicarem a continuació i per cadascuna d'elles donarem el seu pseudocodi.

La primera funció que tractarem és la funció de `cercaParaules(Diccionari, SuperSopa)`. Aquesta funció té un rol important dins l'algorisme, ja que, en primer lloc, ens permetrà crear la *Trie* i encabir-hi totes les paraules a l'ED.

Seguidament, per cada posició `(i,j)` de la Super Sopa ens permetrà invocar la funció de `backtracking(arrel,i,j,paraula,SuperSopa,paraules)` i poder saber si a partir d'aquella posició on es troba un caràcter podem trobar un camí de prefixos, sent el primer prefix el caràcter de la posició, que duen a trobar a la *Trie* un node fulla que representi una cadena.

Un cop hàgim iterat per tota posició de la *Super Sopa* retornarem el conjunt de paraules trobades dins la sopa. Les paraules que retornem formaran part d'un conjunt que s'ha implementat amb un `set` d'strings, ja que en cap moment volem retornar una paraula duplicada. Això és així, perquè podria donar-se el cas en què a la *Super Sopa* s'hi amagui una mateixa paraula construïda des de posicions distintes, per tant, volem un conjunt de paraules no duplicades.

```
funció cercaParaules(Diccionari, SuperSopa):
      creem una Trie t(Diccionari)
      obtenim el node arrel de la Trie t
      paraules ← ∅

      ∀{i,j}∈ in range (0, |SuperSopa|)
            paraula = ' '
            backtracking(arrel, i, j, paraula, SuperSopa, paraules)
```

```
    retorna paraules
```

Figura 5.1.1. *Pseudocodi de la funció cercaParaules*

La segona funció que ja ha aparegut al pseudocodi anterior és la funció anomenada `backtracking(...)`. Aquesta funció té el rol més important a l'algorisme, ja que s'encarregarà de comprovar el fet que desitgem; saber si una paraula de la *Super Sopa* és dins de la nostre estructura arborescent fent una cerca en profunditat o *DFS*.

Tal i com hem vist al pseudocodi anterior, cridarem aquesta funció per cada posició de la Super Sopa, amb el node arrel de la Trie, amb una paraula buida i amb el conjunt de paraules que hem trobat fins al moment a la *Trie*.

La funció inicialment comprovarà que la posició que li arriba no està fora de *bounds* de la *Super Sopa,* i també que el caràcter no sigui fill d'un node a la *Trie.*

Seguidament comprovarà que l'arrel apunti a un node que no sigui **`nullptr`** i que per tant existeix pel caràcter de la *Super Sopa* que es troba a la posició `(i,j)`. Si és compleix que existeix el node, actualitzarem la paraula que tenim fins al moment i amb la funció de transició definida per la Trie, δ: $Q \times \Sigma \rightarrow Q$ anirem al nou node de la Trie que representa el prefix trobat fins al moment. Per tant, mourem el punter al nou node.

Si al node al qual arribem és un node fulla, hem arribat al nostre cas base i és un estat acceptador. El que significarà, és que tenim un camí de l'arrel al node fulla on hem arribat i el prefix trobat dins la *Super Sopa*, és a la vegada una paraula del *Diccionari.* Si és així, inserirem el prefix dins del conjunt de paraules trobades fins al moment.

Un cop després, si el node al que hem arribat no és una fulla, estarem en el cas recursiu. Per tant, agafarem el caràcter que es troba a la posició `SuperSopa[i][j]`, definirem la posició amb el caràcter buit i farem una crida recursiva per cadascuna de les caselles adjacents, 8 en total, des de la posició on ens trobem. Amb aquestes crides, el que pretenem és arribar fins al cas base altre cop, on tinguem un prefix que sigui un node fulla i per tant sigui una paraula del *Diccionari.*

```
funció backtracking(arrel, i, j, paraula, SuperSopa, paraules):
      if (esFora(i,j,SuperSopa) o SuperSopa[i][j] == ' ') retorna
```

```
    if (∃ node apuntat per arrel amb el caràcter SuperSopa[i][j]-'a'):
        paraula <- paraula+SuperSopa[i][j]
        arrel <- apuntarà al node amb el caràcter SuperSopa[i][j]-'a'

    if (arrel.fulla):
        paraules <- paraula

    caràcter = SuperSopa[i][j]
    SuperSopa[i][j] = ' '

    backtracking(i+1,j, paraula, SuperSopa, paraules)
    backtracking(i-1,j, paraula, SuperSopa, paraules)
    backtracking(i,j+1, paraula, SuperSopa, paraules)
    backtracking(i,j-1, paraula, SuperSopa, paraules)
    backtracking(i+1,j+1, paraula, SuperSopa, paraules)
    backtracking(i+1,j-1, paraula, SuperSopa, paraules)
    backtracking(i-1,j+1, paraula, SuperSopa, paraules)
    backtracking(i-1,j-1, paraula, SuperSopa, paraules)

    SuperSopa[i][j] = caràcter
```

Figura 5.1.2. *Pseudocodi de la funció de backtracking*

Aquesta segona funció ens permet introduir una tercera funció de l'algorisme, que simplement serà una funció auxiliar que ens comprobarà si donada una certa posició es troba es troba fora de la *Super Sopa.*

```
funció esFora(i, j, SuperSopa):
    retorna (i < 0 o i > |SuperSopa| o j < 0 o j > |SuperSopa|)
```

Figura 5.1.3. *Pseudocodi de la funció esFora*

5.2 Implementació del diccionari

Per dur a terme la implementació del diccionari amb una *Trie* hem realitzat la implementació de dues EDs, anomenades `NodeTrie` i `Trie`.

5.2.1 NodeTrie

La primera ED que hem emprat per implementar el diccionari amb aquesta tècnica ha set `NodeTrie`. Aquesta estructura ens permet representar de forma abstracta un node o estat (ja que una *Trie* és DFA). L'estructura de dades tindrà una forma ben senzilla ja que únicament un node haurà d'emmagatzemar dos atributs. Un atribut per saber si aquell node és una fulla i per tant és un node que emmagatzema una paraula que es troba al diccionari i un altre, per emmagatzemar un vector de punters de tipus `NodeTrie`, un per cada caràcter del nostre alfabet llatí, Σ. Per tant, la mida del vector de punters serà de $|\Sigma| = 26$.

No hem de perdre de vista que cada node es correspon amb un prefix d'una cadena de D que anirà des del camí traçat començant per l'arrel fins el node en qüestió.

Inicialment doncs, definirem els atributs considerats dins la la constructora de la classe `NodeTrie` on direm que tot `NodeTrie` no serà una fulla, ja que per ser fulla ha de ser una paraula que pertany al diccionari i ho farem posant el booleà que representa aquest atribut `fulla` a fals.

Així mateix, inicialitzarem el vector de punters de tipus `NodeTrie` que tindrà tot node de la *Trie,* dient-li que la seva mida sigui de $|\Sigma| = 26$, però posant-li a cada punter un valor de **`nullptr`**. Això ens serà molt útil ja que ens servirà per saber si un caràcter en concret existeix o no al *Trie.*

```
classe NodeTrie
      constructora NodeTrie:
            node <- |{NodeTrie*}| = (26,nullptr)
            fulla = FALS
```

Figura 5.2.1.1. *Pseudocodi la classe de l'estructura NodeTrie*

A continuació també podem veure un exemple de diferents nodes de la Trie:

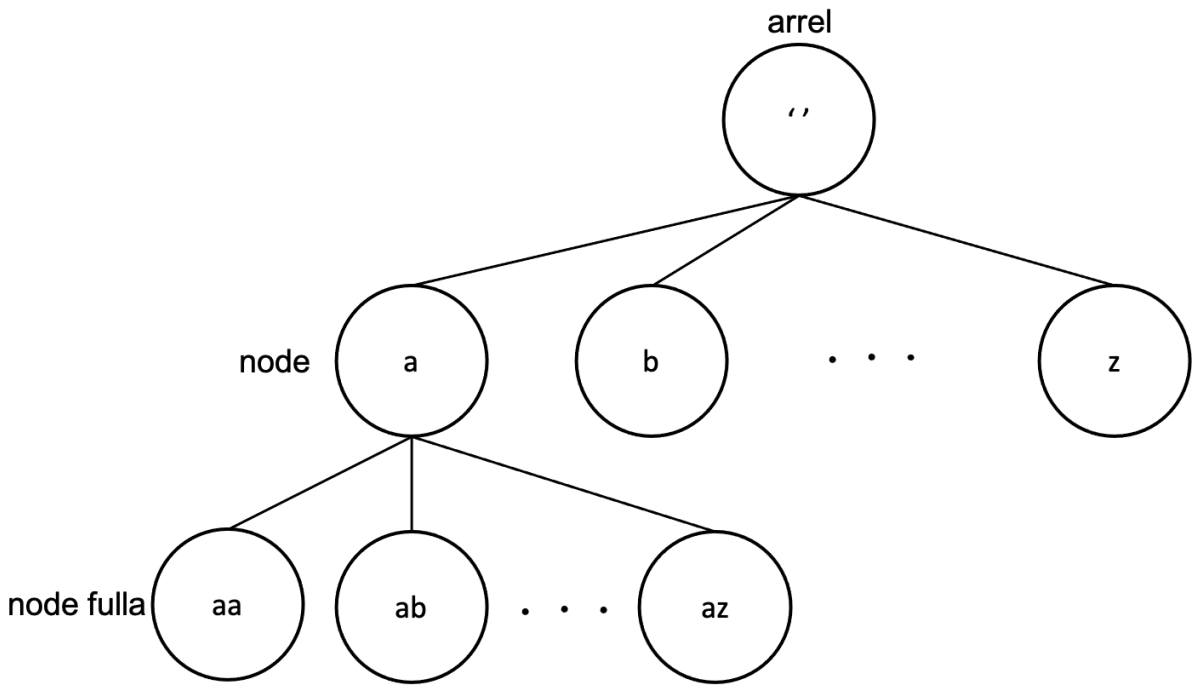

Figura 5.2.1.2. *Representació dels nodes d'una Trie*

Si el node que representa el prefix "aa" d'una cadena el considerem node fulla, significa que la cadena "aa" és una paraula que està continguda al *Diccionari*.

Notem que cada node potencialment pot arribar a tenir 26 nodes successors, però podria ser que en tingués menys, ja que col·loquem únicament els nodes a la *Trie* que són prefixos d'una cadena. Si una prefix d'una cadena no existeix, no el podrem tenir com a camí des de l'arrel fins a un node que representi aquell estat.

5.2.2 Trie

La segona ED que ens ha ajudat a construir el diccionari amb una *Trie* ha set la classe pública anomenada `Trie`. Aquesta classe com bé el nom indica, ens representa pròpiament l'arbre *Trie* on emmagatzemarem el conjunt de cadenes D, o *Diccionari*.

La nostre estructura estarà composada per un únic atribut privat que serà el node arrel de la *Trie.* Aquest node podrà ser accedit mitjançant la funció `getArrel()`.

De la mateixa forma que hem fet a l'ED anterior també ens tocarà inicialitzar la `Trie`. Per una banda assignarem i inicilitzarem l'arrel utilitzant l'operador `new` amb el tipus que li pertoca, que en aquest és de `NodeTrie()`. Un cop creat el node arrel aquest ja tindrà

associat un booleà indicant que no és un node fulla i també tindrà associat un vector de punters `NodeTrie` amb mida |Σ| inicializats a un valor de **`nullptr`** .

Per altra banda, ens quedarà col·locar a la *Trie* el *Diccionari*. Per fer-ho passarem a la constructora de la *Trie* un `vector` de `strings`, on cada element del vector, serà una paraula del *Diccionari* en qüestió. Seguidament, mitjançant la funció pública `afegirParaula()` farem les insercions a la *Trie* passant-li com a paràmetre cada paraula del vector, perquè l'afegeixi a la *Trie*.

Aquesta funció que farà les insercions a la *Trie*, recorrerà cada caràcter que compon la paraula que volem inserir, i per cada caràcter, obtindrem un nombre enter que representarà la posició que ocupa el caràcter al vector de punters del node. Representat com: `i = c - 'a'`, sent `c` el caràcter i `i` la posició dins del vector.

Una vegada obtinguda la posició del caràcter al vector, voldrem assegurar-nos abans de crear un nou node successor que pengi del node actual(on hi haurà el caràcter previ al que tractarem) que no existeix ja un node successor que tingui aquest mateix caràcter. Per això voldrem comprobar la condició de `actual->node[i] == nullptr`.

Si aquesta condició es compleix, voldrà dir que pel caràcter de la paraula que estem tractant encara no existeix un node successor a l'actual. Si és així l'afegirem, de la mateixa forma que hem fet instanciant l'arrel a la constructora de la *Trie*.

Si realment ja existeix aquest node successor a l'actual, i per tant ja hi ha un node amb aquell caràcter, ens hi voldrem moure i per tant desplaçarem el punter a la posició on es troba el caràcter, fet com `actual=actual->node[i]`

Repetirem per cada caràcter que compon la paraula el mateix procés.

Finalment després de llegir el darrer caràcter, marcarem el node com a fulla modificant-ne l'atribut, ja que voldrem marcar en efecte que és el node que representa el prefix de mida igual a la cadena que es volia inserir.

```
class Trie:
      arrel com a membre privat

      constructora Trie(Diccionari):
            arrel <- creem node arrel de tipus NodeTrie
            ∀paraula∈ Diccionari:
                  Trie <- afegirParaula(paraula)

      funció afegirParaula(paraula):
            node actual <- arrel
            ∀c∈ paraula:
                  i <- c - 'a'
                  if (actual no té node successor[i] pel c):
                      actual <- afegim nou node successor[i] pel c
                  actual <- node successor[i] amb caràcter c
            actual.fulla = TRUE
```

Figura 5.2.2.1. *Pseudocodi dels components principals de la classe Trie*

Per poder clarificar una mica com s'estan guardant els nodes a la Trie i com s'entrellacen per formar finalment una estructura arborescent. Veurem una a continuació una imatge d'una Trie que únicament conté 5 paraules. Cada paraula està composada per tants estats com $|Prefixos(paraula)|$. Notem que diverses paraules poden compartir un mateix prefix, com és el cas de **torn, tos i trie**, **torn i tos**, i **uix i ull**.

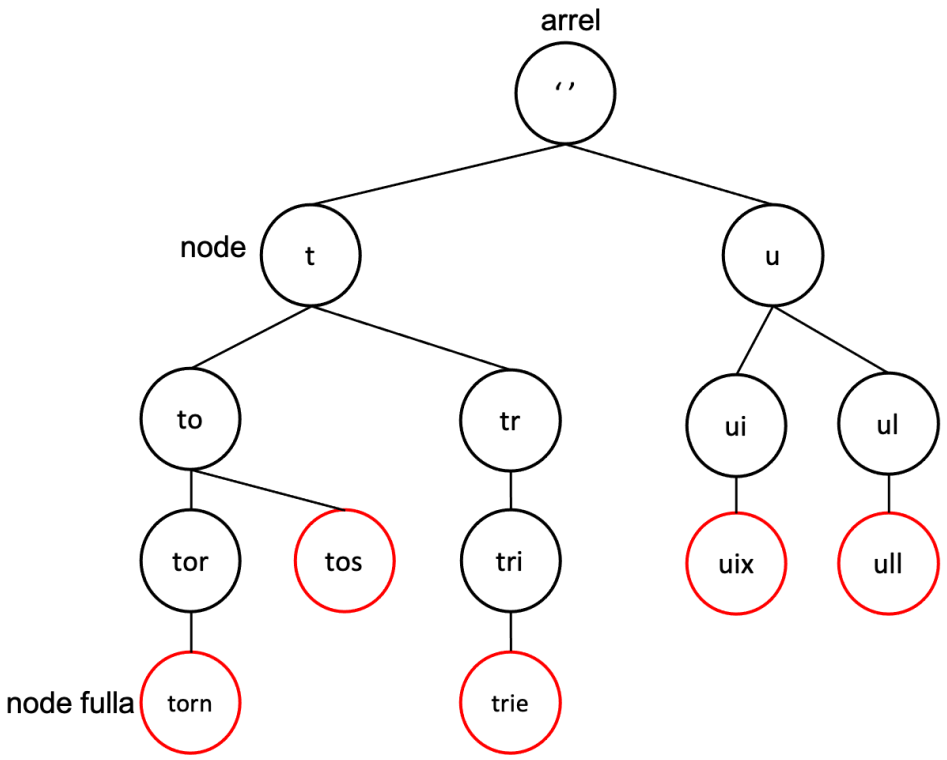

Figura 5.2.2.2. *Representació de la Trie que conté les paraules:* ***torn, tos, trie, uix*** *i* ***ull.***

En vermell podem veure els estats acceptadors de la Trie, és a dir, els estats que contenen la paraula sencera com a prefix. També podem notar a partir de la figura anterior que el conjunt d'estats de la *Trie* seguirà un ordre lexicogràfic.

5.3 Correctesa

Per demostrar la correctesa del nostre algoritme volem:

- Si hi ha una paraula a la Super Sopa, trobar-la sempre mirant si hi ha el camí de prefixos de la paraula des del node arrel fins al node fulla de la *Trie*. Si és així, com que arribem a un node fulla i aquest és el cas base, inserirem la paraula al conjunt de paraules trobades.
- Si hi ha una paraula a la Super Sopa, després de trobar-la el que volem és continuar cercant-ne totes les possibles que hi pugui haver un cop haver fet la cerca d'una paraula anterior. Per tant, miraríem tots els camins de la Trie i mirarem totes les direccions adjacents a una posició de la *Super Sopa* fent servir una cerca en profunditat. Durant la realització de les crides recursives, si trobem un prefix que sigui un node fulla acabarem i inserirem la paraula al `set`.

- Si no hi ha cap paraula a la *Super Sopa* de la mateixa manera comprovarem per cada posició del tauler si el caràcter forma part d'una camí de prefixos d'una paraula que es troba a la Trie.

Una vegada trobem una paraula a la Super Sopa i, per tant, acabem en un node fulla de la Trie, afegim la paraula en el conjunt de paraules trobades a la sopa. El conjunt de paraules trobades no tindrà duplicats, ja que si tornem a trobar una paraula dins la *Super Sopa* que ja es troba en el conjunt, al tractar-se d'un `set` no es tornarà a inserir la paraula en el conjunt.

5.4 Cost

A continuació, detallarem el cost en el pitjor dels casos tant en espai com en temps que tenim a l'hora d'implementar el diccionari amb una Trie i també el cost de consultar si les paraules cercades al tauler són prefixos correctes que es troben dins aquesta ED.

5.4.1 Cost implementació diccionari

El cost que tenim en espai en el cas pitjor per poder guardar les paraules del *Diccionari* a la Trie és de $O(|\Sigma|\cdot|Q|)$. Aquest cost es justifica així ja que amb una *Trie* de $|Q|$ nodes necessitarem un espai en el pitjor dels casos de $|\Sigma|\cdot|Q|$ degut al fet que comptem amb punters per cada node.

El cost temporal en el cas pitjor serà el cost que tardem en inserir les D paraules del *Diccionari*. Per tant la complexitat temporal serà $O(|D|\cdot\frac{1}{D}\sum_{i=0}^{D}|paraula_i|)$ on $\frac{1}{D}\sum_{i=0}^{D}|paraula_i|$és la longitud mitjana de la cadena que cal inserir. També cal tenir en compte que aquesta funció de complexitat és en el cas pitjor, on tot paraula inserida no compartiria cap prefix sinó la complexitat no seria exactament tenir en compte la longitud mitjana de la paraula..

5.4.2 Cost algorisme

La complexitat algorísmica del procediment que cerca prefixos a la *Trie* és de l'ordre de:

$$O(|D|\cdot\frac{1}{D}\sum_{i=0}^{D}|paraula_i|) \; + \; O(N^2\cdot\frac{1}{D}\sum_{i=0}^{D}|paraula_i|\cdot 8^{\frac{1}{D}\sum_{i=0}^{D}|paraula_i|})$$

Aquesta funció ve donada per les tres funcions que componen el nostre algoritme i també per la complexitat de les estructures de dades.

El cost d'aquest algorisme ve donat pel cost de crear la Trie per al *Diccionari* donat i pel cost de fer la consulta de les paraules a la *Trie*. El primer cost és el que hem vist anteriorment i és el cost de la implementació del diccionari a la *Trie*. El segons cost sorgeix de cercar N^2 caselles de la *Super Sopa* per a la longitud mitjana de cada paraula, a més a més, per a cada lletra de la paraula, cercarem en les 8 possibles direccions adjacents, provocant-ne un factor de ramificació de $8^{\frac{1}{D}\sum_{i=0}^{D}|paraula_i|}$. Si ajuntem aquesta imbricació de costos, ens resulta el segon cost: $O(N^2 \cdot \frac{1}{D}\sum_{i=0}^{D}|paraula_i| \cdot 8^{\frac{1}{D}\sum_{i=0}^{D}|paraula_i|})$.

6. Filtre de Bloom

6.1 Descripció de l'algorisme

Un filtre de Bloom és una estructura de dades probabilística que s'utilitza per comprovar si un element és membre d'un conjunt. Les coincidències falses positives són possibles, però els falsos negatius no, és a dir, una consulta retorna cert si la paraula és possiblement al conjunt o fals si definitivament no és al conjunt. Com més elements s'afegeixin, més gran és la probabilitat de falsos positius.

Aquesta tècnica està destinada per a aplicacions on la quantitat de dades d'origen requereix una quantitat de memòria gran si s'apliquen tècniques de hashing "convencionals" sense errors.
Un filtre de bloom buit consisteix en un vector de booleans (bits) on tots els elements són establerts a *false*. Per afegir-hi un element cal realitzar *k* funcions de hash per obtenir *k* posicions del vector, establirem aquestes posicions a *true.*

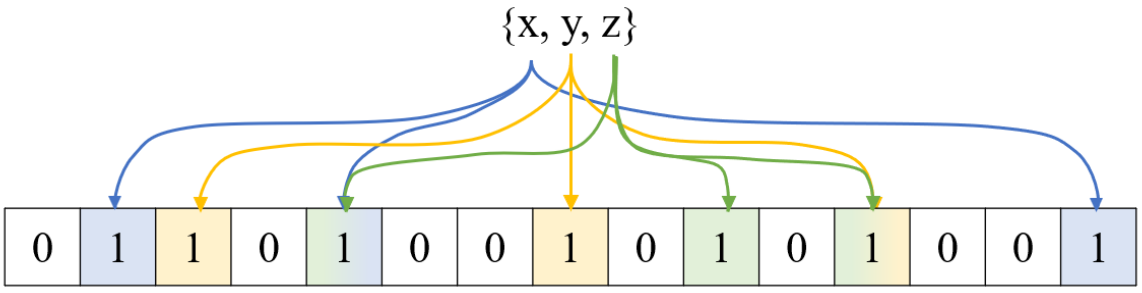

Figura 6.1.1. *Activació dels bits del filtre de Bloom amb 3 elements i k=3.*

Per comprovar si un element es troba en el filtre cal aplicar cadascuna de les *k* funcions de hash i si es dona que tots els bits estan activats, és a dir a *true*, llavors voldrà dir que possiblement l'element buscat és un dels que hi hem afegit anteriorment. La figura 6.1.2 ens mostra una cerca del valor y.

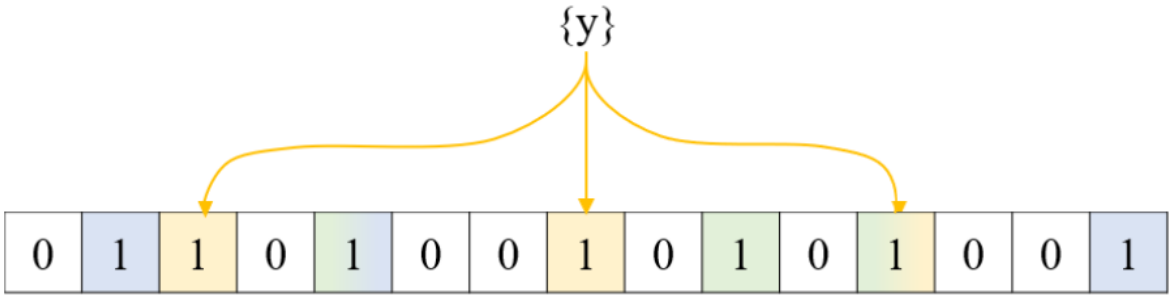

Figura 6.1.2. *Cerca de l'element y en el filtre de Bloom (resultat positiu).*

Si ara cerquem un dels elements que no em afegit poden succeir dues coses en primer lloc que no tots els bits de les posicions obtingudes de les *k* funcions de hash estiguin activades, cosa que implicaria que definitivament l'element no és un dels afegits (com podem observar a la figura 6.1.3) o bé que es doni que tots els bits estan activats i per tant esdevingui un fals positiu (com podem observar a la figura 6.1.4).

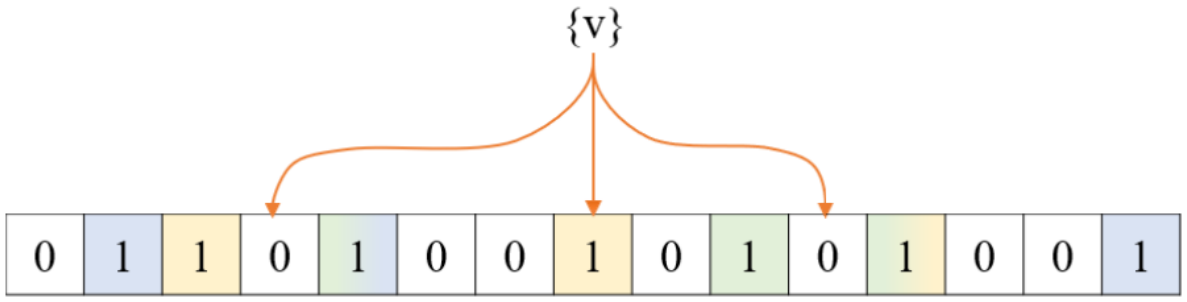

Figura 6.1.3. *Cerca de l'element v en el filtre de Bloom (resultat negatiu).*

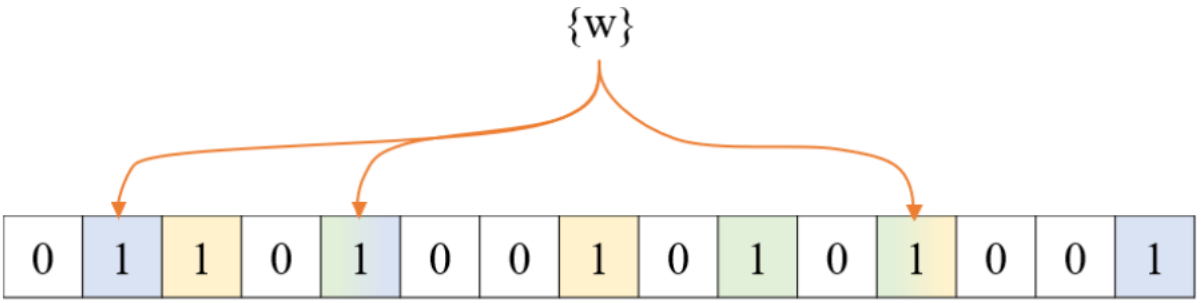

Figura 6.1.4. *Cerca de l'element w en el filtre de Bloom (resultat fals positiu).*

La quantitat de falsos positius dependrà en gran mesura de la quantitat del nombre de bits que utilitzem en el nostre filtre de Bloom i quantes funcions de hash apliquem a cada element. En el cas de la mida del vector, aquesta pot ser aproximada utilitzant la següent fórmula:

$$m = \frac{-n \cdot ln(p)}{(ln(2)^2)}$$

on: m és la mida del vector

n és el nombre de paraules a afegir

p és la probabilitat de falsos positius

En quan el nombre de funcions de hash aquest es pot calcular una bona aproximació al nombre òptim mitjançant la següent fórmula:

$$k = -\log_2(p)$$

on: k és el nombre de funcions de hash

p és la probabilitat de falsos positius

Aquestes fórmules s'utilitzen per calcular la mida i nombre de funcions de hash en la implementació del filtre de Bloom.

6.2 Implementació del diccionari

```
unsigned int stringToInt(string s) {
    const int p = 31;
    const unsigned int m = INT64_C(1) << (sizeof(int)*8 - 1);
    unsigned int hash_value = 0;
    int p_pow = 1;
    ∀i∈ in range (0, s.size()):
        hash_value = (hash_value + (s[i] - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Figura 6.2.1. *Pseudocodi de la funció stringToInt*

La funció `stringToInt(string word)` s'utilitza per obtenir un valor numèric donat un string per poder utilitzar-lo a posteriori per calcular la funció de hash d'una paraula en concret.

```
void buildFilter() {
    c = totalWords;
    filterSize = -(c*log(probability))/(pow(log(2), 2));
    k = -log2(probability);
```

```
    bloomFilter = vector<bool>(filterSize, false);

    ∀i∈ in range (0, k)
        hashFunctions.push_back(randomPrimeNumber());
    }
    ∀word∈ words:
        int wordValue = stringToInt(word);
        ∀hash∈ in hashFunctions:
            bloomFilter[abs(hash*wordValue)%filterSize] = true;
    }
        if(word.size() > maxWordSize) maxWordSize = word.size();
        if(word.size() < minWordSize) minWordSize = word.size();
    }
}
```

Figura 6.2.2. *Pseudocodi de la funció buildFilter*

La funció `buildFilter()` es fa servir per crear el filtre de bloom, en primer lloc s'obtenen els valors de la mida del vector de booleans i el nombre de funcions de hash òptim amb les fórmules que s'ha comentat anteriorment. A continuació es generen nombre primers de forma aleatòria que s'empreran posteriorment per multiplicar en el loop els valors de les paraules en format numèric i així poder obtenir diferents posicions per cadascuna de les funcions de hash. A més a més també s'aprofita per calcular la paraula més petita i més gran dins del loop.

```
bool search(string word) {
    int value = stringToInt(word);
    bool found = true;
    ∀hash∈ in hashFunctions:
        found = found and bloomFilter[abs(value*hash)%filterSize];
    }
    return found;
}
```

Figura 6.2.3. *Pseudocodi de la funció search*

La funció `search(string word)` retorna *true* si la paraula és possiblement una de les paraules que es trobava al diccionari emprat per crear el filtre, o *false* si definitivament no s'ha col·locat al crear el filtre de Bloom.

6.3 Correctesa

Per demostrar la correctesa del nostre algoritme volem:

- Si hi ha una paraula a la Super Sopa, trobar-la sempre mirant si totes les posicions del bits corresponents a la paraula estan activats en el filtre de Bloom.
- Si hi ha una paraula a la Super Sopa, després de trobar-la el que volem és continuar cercant-ne totes les possibles que hi pugui haver un cop haver fet la cerca d'una paraula anterior. Per tant, utilitzant el mètode d'exploració de la sopa comentat a l'apartat 3.4 continuarem la cerca per tots els camins, mirant totes les direccions adjacents a una posició actual de la *Super Sopa*.
- Si no hi ha cap paraula a la *Super Sopa* de la mateixa manera comprovarem per cada possible paraula si aquesta no té tots els corresponents bits activats en el filtre de Bloom.

Una vegada trobem una paraula a la Super Sopa i, per tant, tots el bits del filtre de Bloom estan activitats, afegim la paraula en el conjunt de paraules trobades a la sopa. Tot i que el conjunt de paraules trobades no tindrà duplicats donada la naturalesa del conjunt, no obstant pot donar-se, que donada la naturalesa del filtre de Bloom, podriem trobar-hi paraules no pertanyin al diccionari inicial, és a dir, falsos positius, doncs es tracta d'una estructura de dades probabilística.

6.4 Cost

El cost d'utilitzar un filtre de Bloom es pot dividir en la creació d'aquest i la utilització, és a dir, en la cerca d'un valor dins del filtre.

En primer lloc la creació dependrà del nombre de paraules i el nombre de funcions de hash el qual dependrà de la probabilitat de falsos positius que es defineixi. Donat que el nombre de paraules serà superior al nombre de funcions de hash, podem considerar només el loop en que es s'activen els bits del filtre, dins d'aquest hi trobem dos loops, el que transforma les paraules en valor numèric que té un cost igual al nombre de lletres de la paraula, és a dir $O(w)$, on w és la mida de la paraula, i el que activa els bits com a tal en el que s'itera tantes vegades com nombre de funcions de hash hi ha, és a dir $O(k)$, on k és el nombre de funcions de hash.
Per tant el cost de crear el filtre de bloom serà $O(n * max(w, k))$.

Per altre banda la verificació de si una paraula tindrà una complexitat de $O(k)$, on k és el nombre de funcions de hash, i donat que aquestes sempre seran les mateixes el temps de `search(string word)` serà constant segons k.
En quant a cost espacial, aquest dependrà del nombre de bits del filtre, aquest es defineix segons les fórmules de l'apartat 6.1.

6.5 Experimentació

Ens el temps resultants de l'execució que es pot observar a la figura 6.2.1, es prova el filtre de Bloom amb diferents inputs, en concret s'utilitza un diccionari de 100 paraules i se n'intenten col·locar un subconjunt 50 paraules dins de diversos taulells de mida nxn (eix horitzontal).

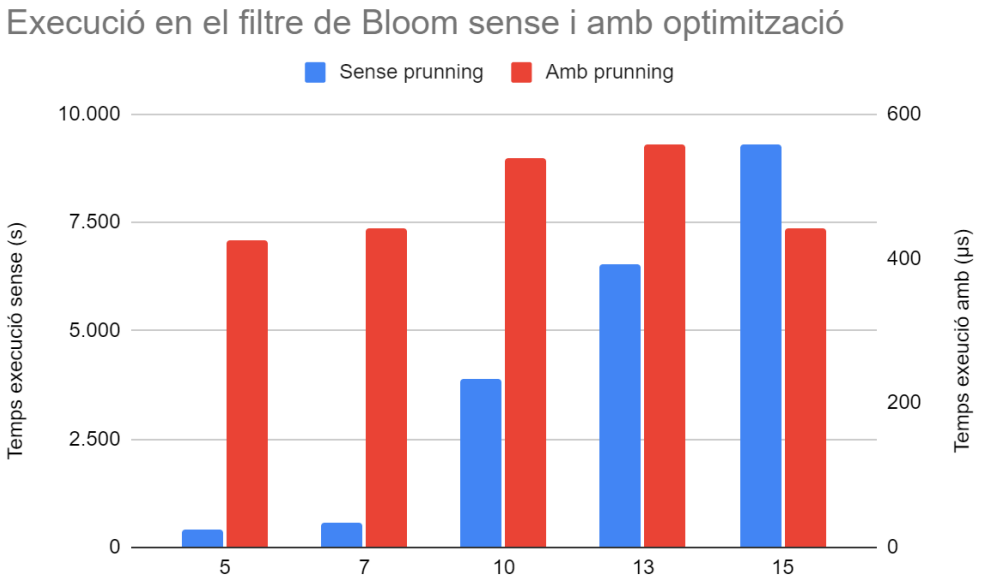

Figura 6.2.1. *Mostra de temps d'execució abans i després de pruning, taula 11.5.*

Podem veure que en comparar els temps obtinguts utilitzant la versió del filtre de Bloom sense pruning aquests estan escalats en gairebé un factor de 10e6 de vegades respecte de la versió que utilitza un cerca amb pruning. Això es deu a que cada vegada que es comença a explorar des de qualsevol posició de la sopa el BFS s'ha d'expandir en totes les direccions possibles i trobar totes les possibles paraules per comprovar si corresponen al diccionari. Com a resultat, això provoca que a mesura que s'incrementa el taulell, les potencials paraules augmentin de forma més que exponencial i conseqüentment ho fa el temps d'execució. A més a més, hem de tenir en compte que el filtre de Bloom és una estructura de dades probabilística i això provoca que es donin una gran quantitat de grans positius (8953891 de 4 paraules reals en el taulell de 5x5 establint la probabilitat d'aquests a un 1%),

cosa ens ha suposat problemes amb la memòria, doncs en arribar a aquesta escala no és suficient amb la memòria d'un ordinador convencional per poder guardar les paraules trobades dins del mateix programa. Per tal de millorar la velocitat d'execució hem utilitzat una tècnica d'optimització anomenada *pruning* empra d'un filtre de Bloom auxiliar en el qual guardem tots el prefixos de les paraules del diccionari, això permet millorar l'eficiència donat que cada vegada que trobem una possible paraula la comprovem primer al filtre de prefixos, i si no hi és sabem que no caldrà continuar expandint en la direcció en que anàvem, doncs no podrà donar-se que hi ha hagi una paraula amb un prefix que no hi ha al filtre ja que els hem insertat tots previament.

7. Double Hashing

7.1 Descripció de l'algorisme

La última estructura de dades que hem implementat és el *Double Hashing*, aquest ens permet indexar les diferents paraules completes en un vector anomenat *Hash Table*, a partir d'aquesta propietat podem crear funcions d'inserció i consulta les quals son essencials per resoldre SuperSopa.

La *Hash Table* ens emmagatzema les paraules del diccionari, aquesta ens indexa qualsevol string a partir de l'aplicació de 3 funcions, una que ens passarà el valor d' alfanumèric a un valor numèric i les dos següents s'apliquen sobre el valor numèric i s'anomenen funcions de hash, d'aquí ve el nom de double hashing, aquestes serveixen per determinar a quina posició es troba un cert string en la taula.

alphanumericToNumeric: Retorna el valor alphanumeric en numèric.

$$\forall\, c \in alphanumeric \rightarrow numeric \;=\; numeric \;+\; (c \;-\; 'a' \;+\; 1) \cdot offset \;\; mod\; tableSize$$

$$offset \;=\; offset \cdot prime \; mod\; tableSize$$

Anomenem la primera funció de hash H1, i la segona H2.

H1: Retorna el residu del valor entre la mida de la taula.

$$hash1 \;=\; V \; mod\; tableSize$$

H2: Retorna la resta entre un primer i el residu del valor entre aquest mateix primer.

$$hash2 \;=\; prime \;-\; (V \; mod\; prime)$$

$$prime \;<=\; tableSize$$

Consulta: Per trobar la paraula necesitem primer transformar-la en un índex, per tant apliquem la funció alphaToNumeric, que retorna el valor numèric *V*, sobre aquest apliquem H1, que utilitzarem com a valor inicial(*indexInicial*), en cas de que la taula en la posició del *indexInicial* tingui el valor "buit" retornem fals, en cas de ser un valor que no és igual a *V*, anirem augmentarem *indexInicial* per el resultat obtingut en H2(*offset*) i consultant si trobem el valor, i si en algun moment tornem al *indexInicial* o és buida retornem fals.

$$index = H1(V) + k\,H2(V)\ mod\ tableSize$$

$$k = \#\ de\ vegades\ que\ H2\ s'ha\ aplicat$$

Inserció: Igual que la consulta utilitza la primera funció per obtenir el valor numèric *V*, sobre aquesta H1 calcula el *indexInicial*, el H2 calcula el valor *offset*, si en *indexInicial* posició la taula té el valor "buit" podem guarda en aquesta V i retornem cert, en cas contrari, que la posició ja està ocupada, afegim sobre el index el *offset* fins a trobar una posició on el valor sigui "buit", en el cas que tornem al valor inicial en alguna d'aquestes operacions, retornem fals.

$$index = H1(V) + k\,H2(V)\ mod\ tableSize$$

$$k = \#\ de\ vegades\ que\ H2\ s'ha\ aplicat$$

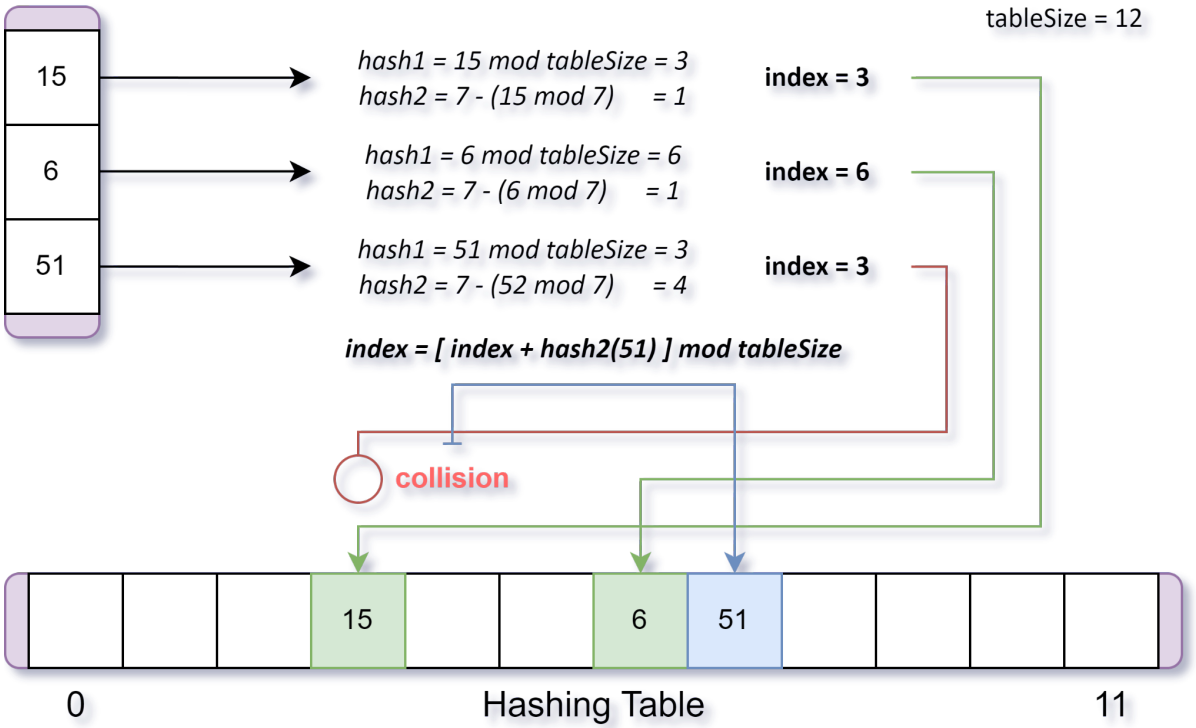

Figura 7.1.1. Taula de hashing

Com es pot veure en aquest exemple del funcionament, en cas de col·lisió augmentarà l'índex a partir del valor retornat amb H2 i en cas que aquest sigui el mateix que l'inicial retorna que no ha estat possible insertar l'element.

7.2 Implementació de l'algorisme

L'implementació de les funcions és la següent:

```
funció stringToInt(string s):
      v = 0
      p = número primer
      ∀c∈ in s:
          v = [v + (c - 'a' + 1) · p ] mod midaInt
          p = (p · p) mod midaInt
      return v
```

Figura 7.2.1. *Pseudocodi de la funció stringToInt*

Per convertir les cadenes de caràcters anirem caracter a caracter, aquest li restarem la seva base ('a') i ho multipliquem per un *factorPrimer*, finalment fem el mòdul de la mida de un enter per verificar que no superi el int, i el guardem en v.
El *factorPrimer* va incrementant per tal de verificar que un mateix carácter no ocupi els bits de l'altre.

```
funció hash1(int v):
      retorna v % tableSize
```

Figura 7.2.2. *Pseudocodi de la funció hash1*

```
funció hash2(int v):
      retorna prime - (v % prime)
```

Figura 7.2.3. *Pseudocodi de la funció hash2*

Aquestes dues funcions són essencials i han de ser tan eficients com potents alhora d'evitar col·lisions, mentres hash1 permet indexar un valor en una posició inicial verificant que no superi la mida de la taula, hash2 ens permetrà trobar posicions posteriors a partir d'un offset(preferiblement un número primer).

```
funció insert(int v):
      h1 = hash1(v)
```

```
      h2 = hash1(v)
      index = h1
      first = true
      while hashTable[index] != -1:
            si index == h1 && !first
                  return false
           index = (index + h2) mod tableSize
      hashTable[index]= v
      return true
```

Figura 7.2.4. *Pseudocodi de la funció insert*

Insert: recorre la taula a partir de h1 i h2, si troba una posició buida en la taula, hi assigna el valor, en cas de retornar a la posició inicial retorna fals.

```
funció search(int v):
      h1 = hash1(v)
      h2 = hash1(v)
      index = h1
      first = true
      while cert:
            si hashTable[index] == v
                  return true
            si index == h1 && !first
                  return false
            si hashTable[index] == -1
                  return false
           index = (index + h2) mod tableSize
```

Figura 7.2.5. *Pseudocodi de la funció search*

Search: recorre la taula a partir de h1 i h2, si troba una posició buida a la taula, retorna fals, en cas de retornar a la posició inicial retorna fals i en cas de trobar una posició la qual conte el valor buscat retorna cert.

Com es pot observar search i insert, utilitzen el mateix procediment per indexar a dins del vector les paraules.

7.3 Correctesa

Cada valor, donades les funcions H1 i H2, es mapeja a una posició de la taula amb el seu valor, per tant, si en algun moment quan estem recorrent la taula trobem que aquella posició no és buida, i el valor guardat coincideix amb el de la paraula, podem verificar que hem trobat una paraula del diccionari dins la sopa, això passarà sempre ja que les funcions de hash donades un cert valor, sempre retornaran el mateix.

En cas de que es busqui una paraula, tot i que aquesta quedi indexada a la mateixa posició si el valor de la taula en aquell index no és el mateix retornarà fals.

7.4 Cost

La complexitat de l'ús de Taules de Hashing la podem dividir en dos parts: generació taula i consultes.

La generació de la taula requereix definir una mida de taula, per definir aquesta mida serà exactament:

$$\#\, paraulesDiccionari\,*$$

Si utilitzessim aquesta mida, cada paraula hauria d'anar mapejada en un index diferent, per prevenir aquestes col·lisions s'ha d'ampliar per un factor, en aquest cas el factor f que ha funcionat millor i no requería d'increments de la taula era 4:

$$*\ \ [log\ (\#\, paraulesDiccionari)\ /\ log(2)] \cdot f$$

Un cop creada la taula, inserim en aquesta les n paraules amb la funció *insert(int v)*, en cas d'haver-hi una col·lisió augmentarem la mida de la taula(la probabilitat de que existeixin no és nul·la, per tant és preferible sobreestimar la mida).

Un cop les paraules inserides ja tenim la estructura de dades llesta!

Repassem els costs, en cas de que encertem a la primera la mida de la taula el cost serà:

$$\#\, paraulesDiccionari \cdot O(insert(int\ v))$$

En millor dels casos quan inserim la paraula i indexar a la primera el cost és O(1), en el pitjor haurem de recórrer tota la taula, O(n), però per la gran majoria de casos el cost d'accés és constant per tant, O(1), si observem per inserir també requerim calcular hash1(v) i hash2(v) que utilitzen les operacions mod i resta, que tenen cost constant, per tant per norma general obtindrem un cost de O(1) en les insercions.

Les consultes funcionen de forma similar, calculen hash1 i hash2 de cost O(1) i iterant sobre el vector fins a trobar el valor, en el millor dels casos el valor es troba en el index calculat del vector, O(1), en el pitjor haurem de recorre tota la taula, O(n), però per norma general les paraules queden freqüentment distribuïdes uniformement per la taula de hashing, per tant podem assumir un cost de O(1).

$$Cost\ Creació\ HT\ = \#\ paraulesDiccionari \cdot O(1)$$
$$Cost\ consulta\ = O(1)$$
$$Complexitat\ Espacial\ = O(midaHT)$$

7.5 Experimentació

Un cop ens hem posat a fer les proves, hem observat que els resultats de les estructures en forma de vector no utilitzen cap mena de poda alhora d'explorar, per tant són extremadament lents ja que requereixen de viatjar cegament per la sopa sense saber si un cert prefix serà part d'una paraula o no, per tant, introduïm la creació de prefixos en la generació d'estructures de dades, per posteriorment fer consultes sobre aquestes.

Posant un exemple pràctic de com és d'útil, si existeix una única paraula en el diccionari com pot ser "carretera", i està generant "carr", on en la següent expansió genera: *carre, carrq, carrt, carrl*, podem descartar aquestes tres últimes, i reduir l'expansió.

Aquest és útil en l'algoritme ja que el temps d'accés és constant, i per tal de comparar la diferencia hem fet aquestes proves:

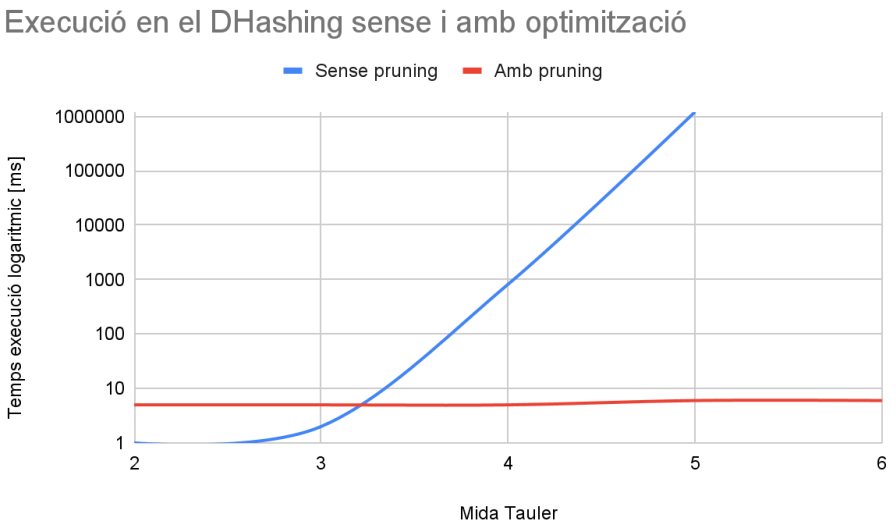

Figura 7.5.5. *Execució amb i sense pruning, taula 11.4*

D'aquesta gràfica podem extreure que quan utilitzem diccionaris grans, recorre la sopa buscant paraules és inviable degut a la llargada de les paraules i la gran quantitat de combinacions que existeixen, a més podem veure, que un cop aplicat l'ús de prefixos obtenim una millora considerable, que a casos petits és de cost constant degut al temps que tarda a crear els prefixos.

La utilització de prefixos és la següent, es crea una taula de hashing per cada mida de prefixos seguint el mateix procediment que per la inserció de paraules del diccionari, en cas de la consulta, per exemple si estem explorant una paraula de mida 3, verificarà si existeix algun prefix de mida 3 amb aquell valor que aparegui en la taula, sino apareix, podem descartar l'expansió, en cas contrari, seguim fent el mateix, per les consecutives mides.

8. Anàlisi experimental de l'eficiència i aplicabilitat dels diferents algorismes i ED a la *Super Sopa*

Un cop realitzat l'estudi de la correctesa dels diferents algorismes, farem la validació experimental de l'eficiència i aplicabilitat d'aquests a l'hora d'implementar el diccionari i cercar les paraules a la *Super Sopa*.

En l'experimentació ens interessarà saber com es comporten els diferents algorismes a l'hora de fer la cerca de paraules i si aquests tenen un cost d'execució acceptable.

Per tal d'extreure resultats fiables totes les execucions es faran en el mateix entorn, en un ordinador té un procesador AMD Ryzen 2700X, 32GB de RAM, sistema operatiu Ubuntu 22.04 i la versió del gcc és la 11.2.0. Cada execució es farà cinc vegades i els resultats mostrats en es gràfics seran el resultat de la seva mitjana.

Per fer aquesta experimentació variarem diferents paràmetres que intervenen en la cerca, com són:

- La mida de la *Super Sopa*
- La mida del subset de paraules que plantem a la *Super Sopa*
- La mida del *Diccionari*

En concret, farem diversos estudis variant i fixant els paràmetres anteriors. Aquests seran:

- Estudi en funció de la mida de la *Super Sopa* variant la mida del subset de paraules plantades per tots els *Diccionaris*
- Estudi en funció de la mida del subset que plantem, fixant la mida de Taulell en **300x300** amb el ***Diccionari*** de **quijote-vocabulary**
- Estudi en funció de la mida del *Diccionari*, amb una mida de subset de **500** paraules i fixant la mida del taulell en **300x300** amb el ***Diccionari*** de **quijote-vocabulary**
- Estudi del nombre de paraules extres que s'han generat aleatòriament amb una mida de subset de 500 paraules amb el *Diccionari* de **quijote-vocabulary**.
- Estudi de l'ús de memòria en funció de la mida del *Diccionari* **quijote-vocabulary** fixant la mida del subset a 500 paraules i amb una *Super Sopa* a **500x500**

Notem primer de tot que disposem de tres *Diccionaris* distints, on cadascun d'ells té un nombre diferent de paraules:

- **mare-balena-vocabulary.txt *[5053 paraules]***
- **dracula-vocabulary.txt *[9421 paraules]***
- **quijote-vocabulary.txt *[25583 paraules]***

8.1 Estudi en funció de la mida de la *Super Sopa* variant la mida del subset de paraules plantades

8.1.1 Presentació de l'experiment

En aquest primer estudi, ens focalitzarem en estudiar quin efecte té variar la mida del Tauler de la Super Sopa variant-ne també la mida del subset plantat a la sopa. Això és així ja que a mesura que la mida de la sopa augmenti ens interessarà també augmentar la mida de subset ja que sinó el ratio entre la mida de la sopa i la quantitat de paraules plantades no mantindria una mateixa proporció, on l'estudi a realitzar no seria significatiu

Per aquest estudi farem servir tots els *Diccionaris* i observarem com es comporten els diferents algorismes per a cada tipus de diccionari, on cadascun d'ells, tal i com s'ha mencionat abans té una mida diferent.

8.1.2 Hipòtesi

Quant més gran sigui la mida del tauler i més paraules tingui el diccionari utilitzat més temps trigarà a executar-se. A més a més, la *Trie* serà la estructura més ràpida de totes, tot i les optimitzacions fetes a la resta per aconseguir el mateix efecte de pruning d'aquesta estructura, mentre que el vector serà el més lent doncs cada cerca en aquest té un cost logarítmic.

8.1.3 Gràfiques

A continuació podem veure les gràfiques, que representen les taules de dades que podem trobar al annex 11.1. Cal destacar que degut als temps d'execució elevats de vector Ordenat s'ha decidit representar-lo escalat per un factor de 1/10, i en alguns casos no s'ha arribat a executar ja que ja es pot observar la seva tendència o els temps d'execució eren extremadament alts.

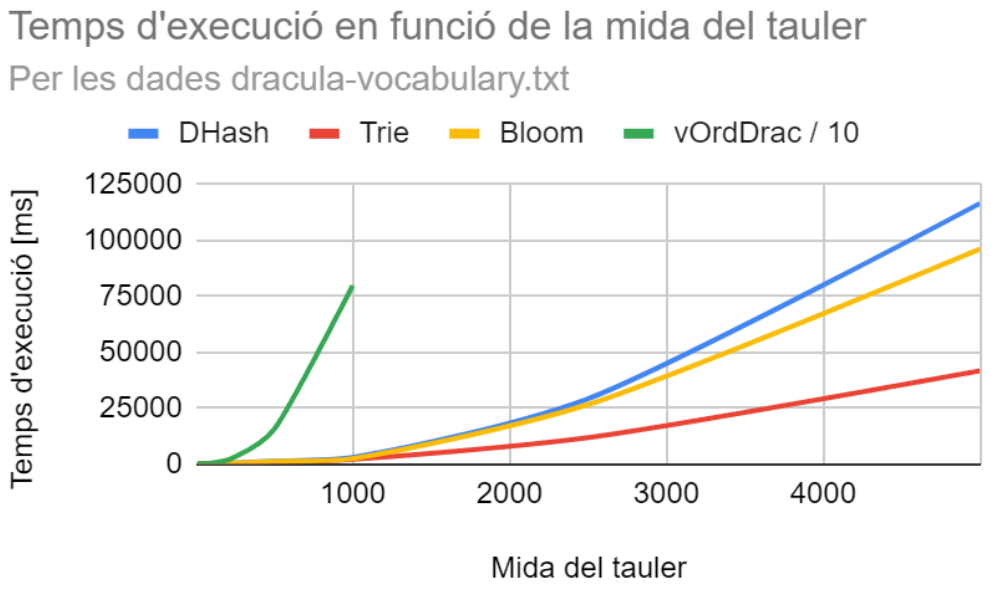

Figura 8.1.3.1. Gràfic de temps d'execució en funció de la mida del tauler per les dades Dracula.

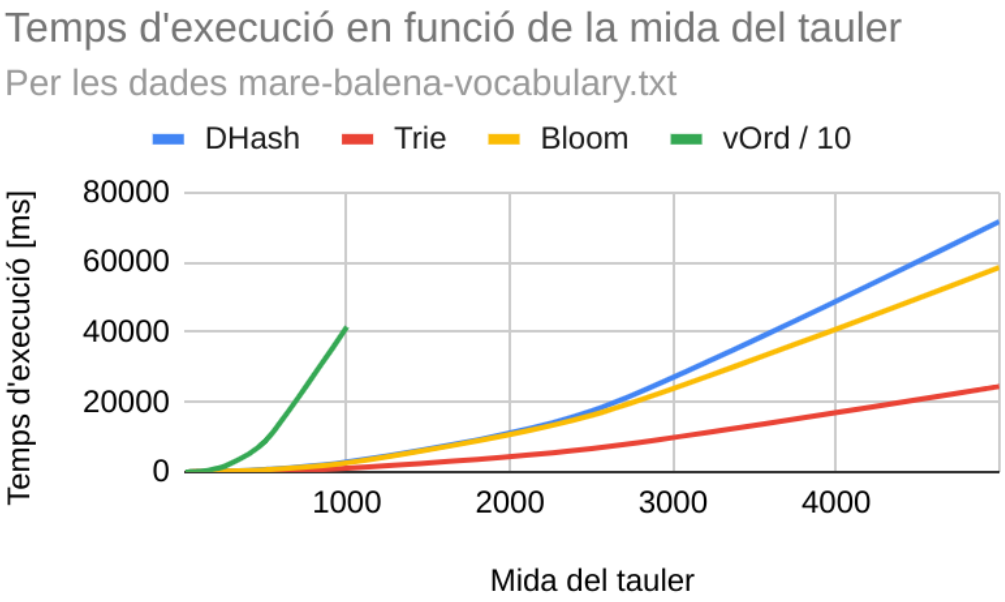

Figura 8.1.3.2. *Gràfic de temps d'execució en funció de la mida del tauler per les dades Mare Balena.*

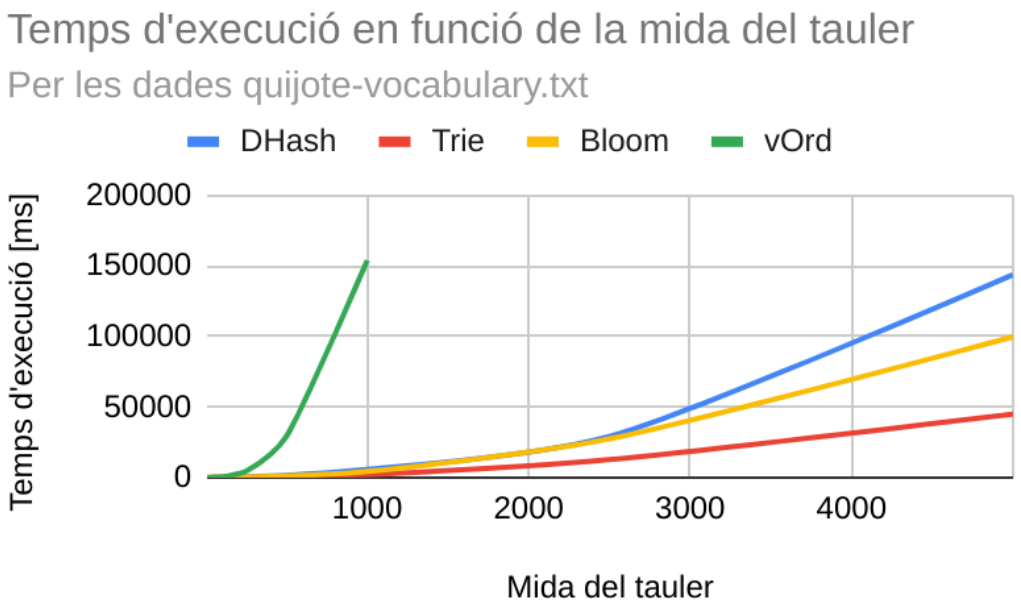

Figura 8.1.3.3. *Gràfic de temps d'execució en funció de la mida del tauler per les dades Quijote.*

8.1.4. Resultats

A partir de les gràfiques podem observar que el vector ordenat és el més lent de tots amb diferència, doncs cada cerca té un temps logarítmic a diferència dels altre els qual resolen la cerca amb una complexitat constant.
A més a més, també podem observar que per totes les mides de tauler i per cadascuna de les estructures el temps d'execució augmenta de forma directament proporcional respecte la mida del diccionari, per això mateix vam decidir fer un experiment on de forma gradual incrementem la mida del diccionari.

Per tant, tal i com hem vist a partir dels resultats, podem afirmar la nostra hipòtesis de partida.

8.2. Estudi en funció de la mida del subset que plantem fixant la mida de Taulell en 300x300 amb el *Diccionari* de quijote-vocabulary

8.2.1 Presentació de l'experiment

En aquest segon estudi, ens centrarem en estudiar quin efecte té variar la mida del subset plantat a la *Super Sopa* amb una mida de Taulell establerta de **300x300**[1] i amb el *Diccionari*

[1] Si la mida del taulell fos més petita, les variacions entre tests no es veurien reflexades en els resultats, en canvi si la mida fos més gran els experiments encara els esteriem fent ara.

quijote-vocabulary. Aquest és el diccionari amb més paraules i per tant és el que ens interessa més a l'hora de fer variar la mida del subset.

8.2.2 Hipòtesi

Quant més gran sigui la mida del subset més temps trigarà a executar-se ja que al augmentar el nombre de paraules a la Super Sopa augmentarà el nombre de prefixos plantats.

8.2.3 Gràfiques

A continuació podem veure les gràfiques, que representen les taules de dades que podem trobar al annex 11.2.. Cal destacar que degut als temps d'execució elevats de vector Ordenat s'ha decidit representar-lo escalat per un factor de 1/100.

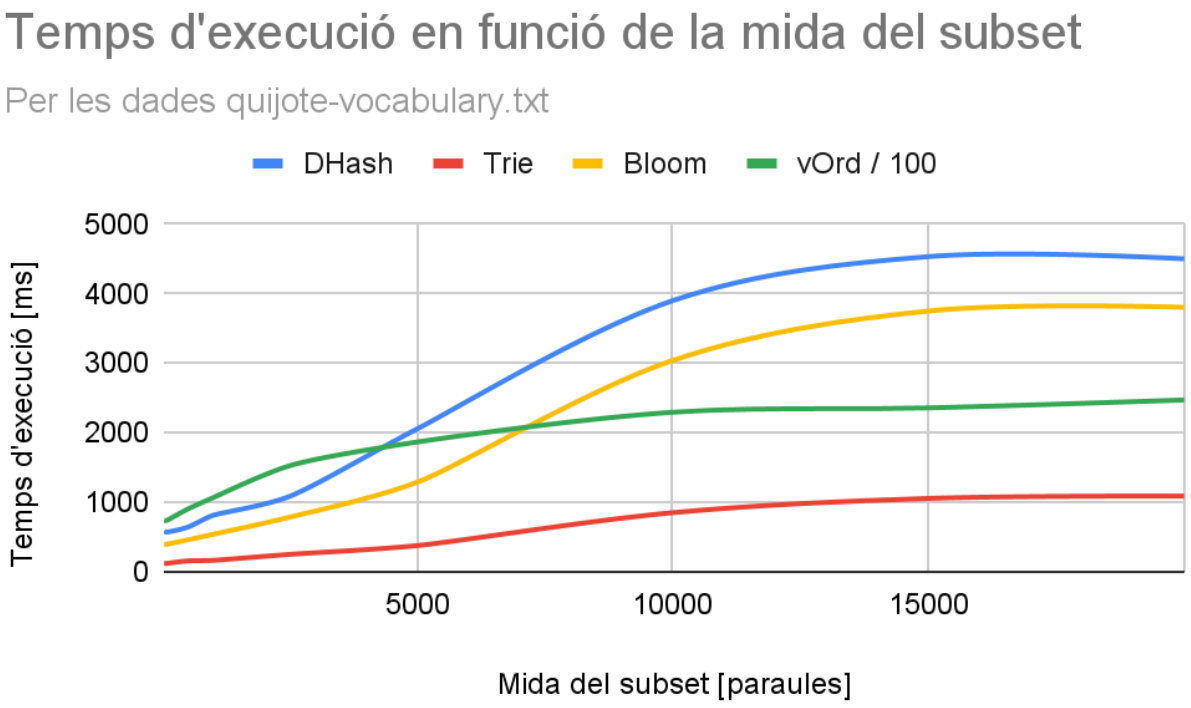

Figura 8.2.3.1 *Gràfic de temps d'execució en funció de la mida del subset per les dades Quijote*

8.2.4 Resultats

A partir de la gràfica anterior podem observar que al anar incrementant la mida de subset, la sopa s'hi troben més paraules del subconjunt, degut això les consultes de prefixos deixen de tenir el mateix efecte ja que al augmentar el nombre de prefixos a la *Super Sopa*, la poda de branques no es fa fins al final. Degut a això el temps de cerca a la sopa no es pot "retallar" amb el *pruning*.

Per tant, altre cop, a partir dels resultats, podem afirmar la nostra hipòtesis de partida.

8.3 Estudi en funció de la mida del diccionari, amb un subset de 500 paraules i fixant la mida del taulell en 300x300 amb el *Diccionari* de quijote-vocabulary

8.3.1 Presentació de l'experiment

En aquest tercer estudi, ens enfocarem en estudiar quin efecte té variar la mida del *Diccionari*, **quijote-vocabulary**, amb una mida de subset de 500[2] paraules plantades i fixant la mida de la *Super Sopa* en **300x300**. Hem decidir altre cop agafar altre cop aquest diccionari perquè al contenir moltes més paraules, podem escollir de partida un subset de paraules molt més gran i anar-lo augmentant per a cada mida de diccionari.

8.3.2 Hipòtesi

El temps d'execució augmenta a mesura que ho fa la mida del diccionari.

8.3.3 Gràfiques

A continuació podem veure les gràfiques, que representen les taules de dades que podem trobar al annex 11.3. Cal destacar que degut als temps d'execució elevats de vector ordenat s'ha decidit representar-lo escalat per un factor de 1/100.

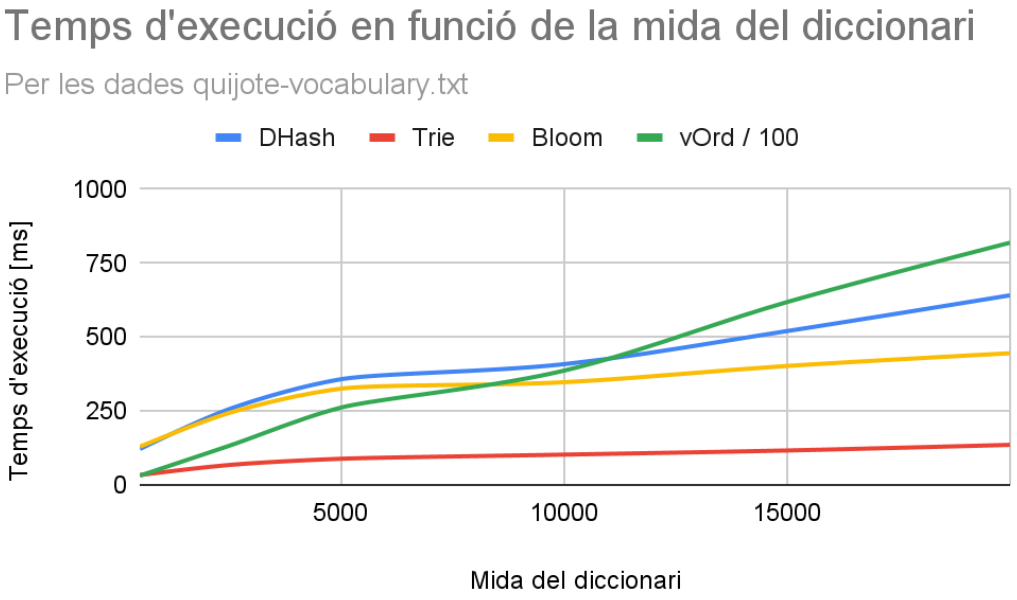

Figura 8.3.3.1. *Gràfic de temps d'execució en funció de la mida del diccionari per les dades Quijote*

[2] Si la mida del subset fos més petita, les variacions entre tests no es veurien reflexades en els resultats, en canvi si la mida fos més gran, limitarà el mínim de paraules que pot tenir del diccionari.

8.3.4 Resultats

A partir de les gràfiques podem observar com la creació de les estructures amb diccionaris més grans és més elevada, i com el cost de consulta per vector és més elevat que la resta augmenta de forma més accentuada, on bloom i Double Hashing, els hi suposa un impacte gran al principi degut a la creació d'estructures força més grans, però en canvi a la llarga aquestes acaben contribuint una part menor a l'execució total, el cost del Trie acaba sent el menys afectat, ja que no s'hi adhereixen paraules gaire diferents i de longitud creixent, que són els que realment generen un impacte, sabent també que en els diccionaris existeixen paraules que són prefixes d'altres i per tant no cal afegir més nodes.
Consegüentment a partir dels resultats, podem afirmar la nostra hipòtesis de partida.

8.4 Estudi del nombre de paraules extres que s'han generat aleatòriament amb una mida de subset de 500 paraules amb el *Diccionari* de quijote-vocabulary

8.4.1 Presentació de l'experiment

En aquest estudi, el nostre marc d'anàlisi serà veure quin efecte té variar la mida del taulell envers al nombre de paraules que es troben del diccionari fixant el nombre de paraules del subconjunt a plantar a 500.

8.4.2 Hipòtesi

Com més gran sigui el taulell més paraules podrem trobar del diccionari.

8.4.3 Gràfiques

A continuació podem veure la gràfica que representa la taula de dades que podem trobar al annex 11.7.

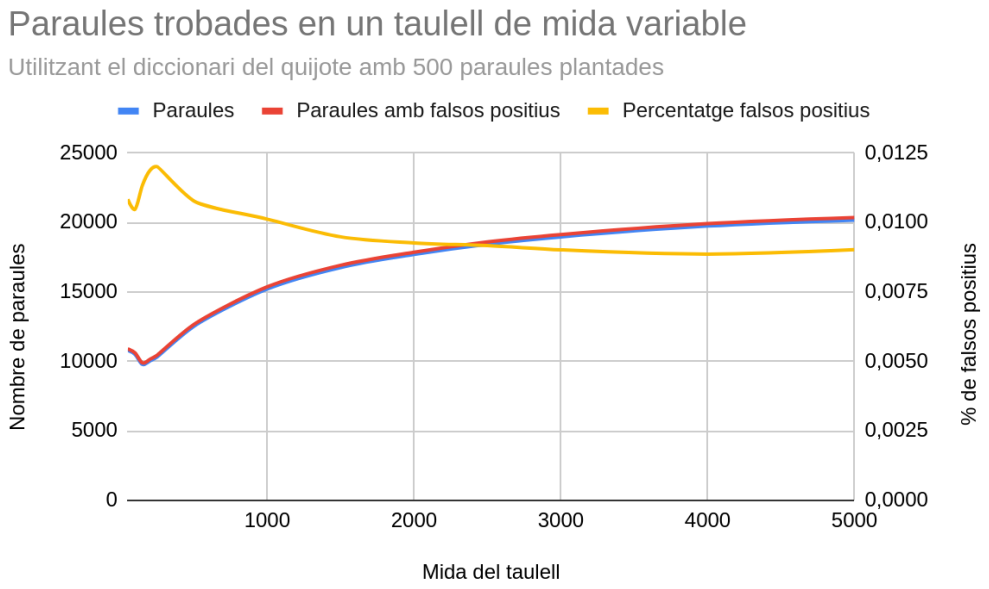

Figura 8.4.3.1. *Gràfic del nombre de paraules trobades segons la mida del taulell.*

8.4.4 Resultats

A partir de la gràfica anterior podem observar que el nombre de paraules trobades incrementa de forma logarítmica segons la mida del taulell, això es deu a que no es tenen en compte les repeticions de les paraules i per tant el subconjunt de paraules que falta per trobar és cada vegada més petit i per tant més difícil que aparegui de forma aleatoria al taulell. Addicionalment, podem veure un mínim local quan la mida és 150, cosa que es pot explicar en base a que a partir d'aquesta mida les paraules comencen a estar més distanciades unes de les altres, i donat que llavors emplenem els espais buits amb lletres aleatòries aquestes no estan distribuïdes amb la mateixa freqüència en què apareixen a les paraules (fet que si que es dona en mides inferiors).

Tanmateix podem veure com el filtre de Bloom amb optimització no genera gaires falsos positius, de fet és sempre pròxim al 1% que s'estableix en les fórmules de creació d'aquest. Podem observar que té una tendència inversament proporcional al nombre de paraules reals, i això es dona ja que com més paraules hi ha més *pruning* es fa, cosa que ajuda a reduir aquest percentatge.

Així doncs, podem afirmar la hipòtesi de partida de l'estudi.

8.5. Estudi de l'ús de memòria en funció de la mida del Diccionari quijote-vocabulary fixant la mida del subset a 500 i la Super Sopa a 500x500.

8.5.1 Presentació de l'experiment

En aquest estudi, el nostre marc d'anàlisi serà veure quin efecte té en l'ús de la memòria variar la mida del *Diccionari* **quijote-vocabulary**, per a un subset de **500** paraules i una Super Sopa de **500x500.**

8.5.2 Hipòtesi

Les estructures de dades que ocupen més memòria serà la Trie, mentre que el Vector Ordenat, el Filtre de Bloom i el Double Hashing n'ocupen molta menys.

8.5.3 Gràfiques

A continuació podem veure les gràfiques, que representen les taules de dades que podem trobar al annex 11.4.

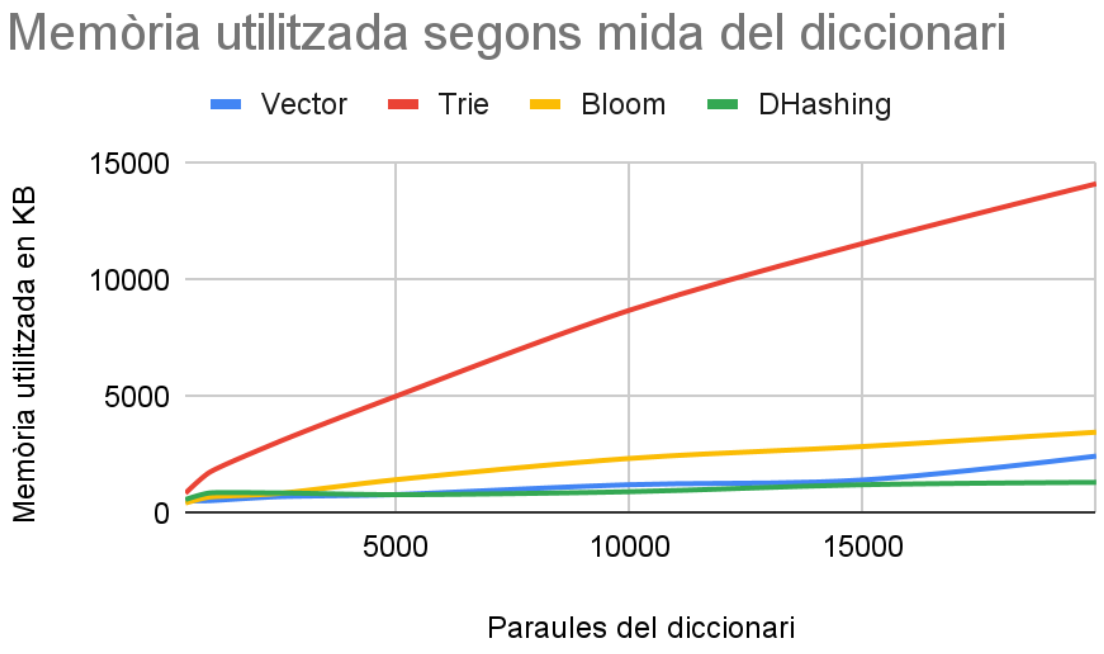

Figura 8.5.3.1. *Gràfica que representa l'ús de memòria en de les estructures de dades*

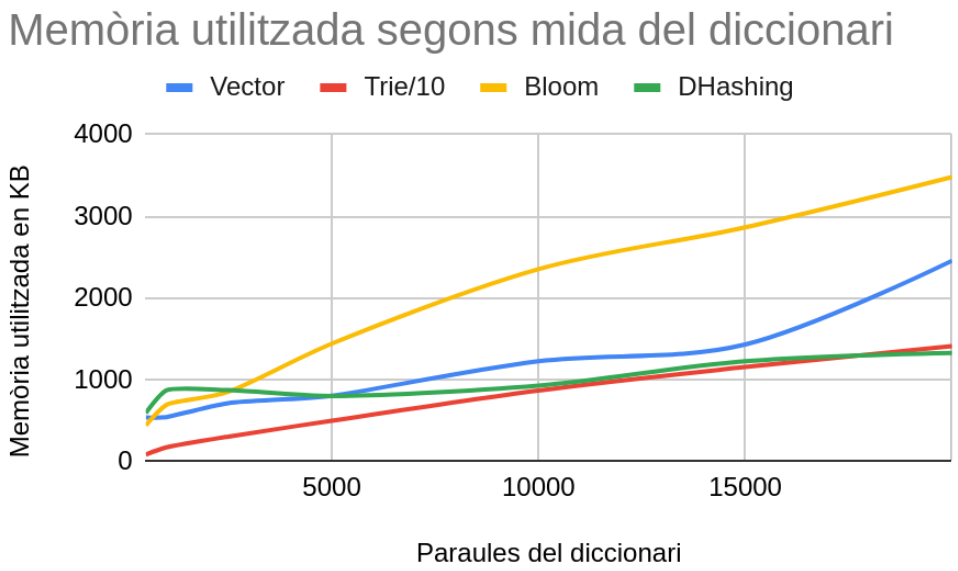

Figura 8.5.3.2. *Gràfica que representa l'ús de memòria en de les estructures de dades dividint la trie entre 10 per poder observar les tendències de Vector, Bloom i Double Hashing.*

8.5.4 Resultats

A partir de les gràfiques podem observar que la Tire clarament és la implementació que ocupa més memòria, això és degut a el seu funcionament, ja que es tracta d'un arbre de prefixos. Seguidament ens trobem amb la implementació del diccionari de Bloom, Vector Ordenat i finalment Hashing.

Consegüentment a partir dels resultats, podem afirmar la nostra hipòtesis de partida.

9. Conclusions

Després de realitzar la validació experimental vista a l'apartat anterior, ja podem donar les idees claus i conclusions pel que fa a l'anàlisis que s'ha dut a terme per cada estudi..

Primerament, pel primer estudi, podem afirmar que quan més gran sigui la mida del tauler i més paraules tingui el diccionari utilitzat més temps trigarà a executar-se.Tanmateix, la *Trie* serà la estructura més ràpida de totes, tot i les optimitzacions fetes a la resta per aconseguir el mateix efecte de pruning d'aquesta estructura, mentre que el vector serà l'estructura més lenta.

El segon estudi ens ha mostrat que si ens trobem amb una sopa plena de paraules on aquestes són el diccionari, les consultes a prefixos deixen de ser tan útils, ja que freqüentment estem explorant el prefix d'una paraula que es troba en la sopa i per tant no la podem ometre la branca.

En el tercer estudi hem vist com la creació de les estructures amb diccionaris més grans és més elevada, i com el cost de consulta per vector és més elevat que la resta d'estructures. Les tècniques de *Bloom* i *Double Hashing*, els hi suposa un impacte gran al principi degut a la creació d'estructures força més grans, però a la llarga aquestes acaben contribuint una part menor a l'execució total. Per contra, el cost del *Trie* acaba sent el menys afectat.

El quart estudi ens ha mostrat la relació entre el nombre de paraules que es poden trobar al diccionari segons la mida del taulell que s'utilitza, hem pogut observar que la mida d'aquest és directament proporcional al nombre de paraules, doncs podem veure que en incrementar-ne el tamany més paraules poden aparèixer de forma aleatòria, però donat que el diccionari és limitat cada vegada costen més que apareixin espontàniament.

I finalment, el cinquè estudi ens mostra que de cara a eficiència en l'ús de memòria la Trie és la pitjor implementació degut al seu funcionament, al crear un arbre de prefixos. El Filtre de Bloom també és més costós però molt més eficient que Trie. Vector Ordenat també té un cost bastant semblant a Bloom però una mica millor i finalment el Double Hashing és la millor de les implementacions.

En termes d'aplicabilitat observem com el vector ordenat no és una forma raonable de resoldre *Super Sopes* gaire grans i/o amb diccionaris gaire grans tot i que té un ús de memòria no massa elevat. Per altra banda, tant *Double Hashing* com *Bloom Filter*, ja són aplicables a la gran majoria de *Super Sopes* sense recaure gaire importància amb els paràmetres que hem variat amb els estudis anteriors i els dos tenen costos de memòria semblants als de Vector Ordenat, on . Finalment, si el que volem és la solució més eficient temporalment concluïm que seria el *Trie* degut a la seva naturalesa de poda. En el cas que volguessim uns resultats més equilibrats seria millor utilitzar Filtre de Bloom, si no ens afecta massa l'aparició de falsos positius, o de Double Hashing, que és més costós temporalment que el Filtre de Bloom però també ens assegurarem de treure uns resultats exactes i farem servir menys memòria que aquest altre.

A partir de la validació experimental realitzada, i partir de les conclusions extretes, hem arribat a la conclusió que seria útil realizar futurs estudis com un estudi més específic de la memòria, ja que actualment estem fent l'estudi de l'espai utilitzat com a un conjunt, comptant-hi l'espai de la Super Sopa, que és el mateix per a totes les estructures. És a dir fer un estudi on no es té a compte l'espai que ocupa el tauler, per poder observar millor les diferències de creixement en les estructures corresponents.
També seria interessant fer un estudi sobre la Super Sopa en sí, la generació de paraules aleatòries

A mesura que augmentem la mida del taulell la memoria que fem servir totes creixen igual, perquè tots guardem la sopa a la mateixa estructura, per tant fem servir una memòria semblant. Llavors es veu que

10. Referències

[1] - Swamidass, S. Joshua; Baldi, Pierre (2007), "Mathematical correction for fingerprint similarity measures to improve chemical retrieval", *Journal of Chemical Information and Modeling*, **47** (3): 952–964, doi:10.1021/ci600526a, PMID 17444629

[2] - Wikipedia contributors. (2022a, agosto 16). *Bloom filter*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bloom_filter

[3] Bloom, B. H. (1970, julio). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422-426. https://doi.org/10.1145/362686.362692

[4] - Wikipedia contributors. (2022, septiembre 11). *Trie*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Trie

[5] - *Trie*. (s/f). Nist.gov. https://xlinux.nist.gov/dads/HTML/trie.htm

[6] - GeeksforGeeks. (2022, 1 julio). *Double Hashing*. https://www.geeksforgeeks.org/double-hashing/

[7] - Ken Christensen,Allen Roginsky,Miguel Jimeno*: "*A new analysis of the false positive rate of a Bloom filter". (15 October 2010). Elsevier

[8] -

11. Annex

11.1. Dades de “Estudi en funció de la mida de la *Super Sopa* variant la mida del subset de paraules plantades”

	Dades Dracula vocabulary [9421]			
Mida Tauler	vOrd	DHash	Trie	Bloom
10	22	13	6	10
25	88	17	8	14
50	490	28	13	27
75	1459	48	20	41
100	2814	68	31	58
150	8276	124	52	112
200	16229	207	85	186
250	30636	312	125	285
500	154375	1181	480	1068
1000	798142	2840	1964	2319
2500	-	29087	11685	26499
5000	-	116684	41641	96196

	Dades Mare Balena vocabulary [5053 paraules]			
Mida Tauler	vOrd	DHash	Trie	Bloom
10	15	6	3	5
25	52	9	4	8
50	267	17	7	15
75	885	32	13	28
100	1654	42	17	37
150	4768	78	35	69
200	9728	128	49	115
250	16093	191	73	184
500	88229	724	270	658
1000	414937	2840	1052	2616
2500	-	17405	6668	16190
5000	-	71634	24463	58533

	Dades Quijote Vocabulary [25583]			
Mida Tauler	vOrd	DHash	Trie	Bloom
10	67	81	15	25
25	163	89	18	30
50	747	105	24	45
75	2536	131	36	70
100	4514	160	43	82
150	13034	226	73	140
200	26611	329	101	204
250	44379	452	149	324
500	291931	1559	527	1097
1000	1539257	5806	1988	4297
2500	-	29087	12695	27406
5000	-	143803	44958	99730

11.2. Dades de Estudi en funció de la mida del subset que plantem fixant la mida de Taulell en 300x300 amb el *Diccionari* de quijote-vocabulary

	Estudi en funció de paraules subset 300x300				
	Dades Quijote Vocabulary [25583]				
Mida Subset	vOrd	DHash	Trie	Bloom	vOrd / 100
50	73893	581	117	393	738
100	73663	569	118	394	736
500	89678	637	154	455	896
1000	106274	809	162	536	1062
2500	152331	1082	248	783	1523
5000	186694	2055	375	1287	1866
10000	229438	3901	849	3040	2294
15000	235759	4534	1054	3751	2357
20000	247138	4502	1087	3804	2471

11.3. Dades de Estudi en funció de la mida del diccionari, variant la mida del subset de paraules plantades i fixant la mida del taulell en 300x300 amb el Diccionari de quijote-vocabulary

	Estudi en funció de paraules diccionari 300x300 (500 subset)				
	Dades Quijote Vocabulary [25583]				
Mida Diccionari	vOrd	DHash	Trie	Bloom	vOrd / 100
500	3108	121	33	129	31
2500	13217	256	67	243	132
5000	26150	357	88	325	261
10000	38679	408	102	347	386
15000	61835	520	116	402	618
20000	81900	641	135	445	819

11.4. Dades de l'estudi número 6, estudi sobre l'ús de memòria.

Variant Mida Taulell				
En kB	Vector	Trie	Bloom	Double Hashing
500	540,7	864,4	442,4	593,9
1000	544,8	1740,8	696,3	872,4
2500	716,8	3072	856,1	876,5
5000	806,9	5017,6	1443,6	802,8
10000	1228,8	8704	2355,2	929,8
15000	1433	12185,6	2867,2	1228,8
20000	2457,6	14131,2	3481,6	1331,2

La següent taula és la mateixa amb Trie/10.

Variant Mida Taulell				
En kB	Vector	Trie/10	Bloom	Double Hashing
500	540,7	86,4	442,4	593,9
1000	544,8	174,8	696,3	872,4
2500	716,8	307	856,1	876,5
5000	806,9	501	1443,6	802,8
10000	1228,8	870	2355,2	929,8
15000	1433	1157	2867,2	1228,8
20000	2457,6	1413	3481,6	1331,2

11.5. Dades de Double Hashing amb i sense optimització

Dades Mare Balena Vocabulary (ms)		
Mida Sopa	Original	Pruning
2	1	5
3	2	5
4	824	6
5	1216751	6
6	-	9

11.6. Dades de filtre de Bloom amb i sense optimització

	Optimització del filtre de Bloom	
Mida Tauler	Sense pruning	Amb pruning
5	409	426
7	584	442
10	3901	538
13	6543	558
15	308	441

11.7. Dades del generador de paraules

	Estudi nombre de paraules en funció de la mida del taulell	
Mida taulell	Paraules	Paraules amb falsos positius
50	10797	10914
100	10495	10605
150	9785	9896
200	10028	10147
250	10325	10449
500	12527	12662
1000	15217	15371
1500	16764	16923
2000	17700	17864
2500	18433	18602
3000	18964	19135
4000	19745	19920
5000	20176	20358