

Super Sopa

Algoritmia (GEI)

Cuatrimestre de Primavera, curso 2022/23



Mario Fernández Simón
Pol Pérez Castillo
Joan Sales de Marcos
Victor Teixidó López

Índice

1. Introducción	3
2. Vector ordenado	4
3. Trie	6
3.1. Patricia	8
4. Filtro de Bloom	10
5. Doble Hashing	12
6. Experimentación	15
6.1. Vector ordenado	16
6.2. Trie	17
6.3. Filtro de Bloom	19
6.4. Doble Hashing	22
7. Conclusión	24
8. Bibliografía	25

1. Introducción

En este proyecto se nos pide implementar y realizar una validación experimental de la eficiencia y aplicabilidad de diferentes algoritmos y estructuras de datos para la búsqueda de palabras en una sopa de letras algo especial, llamada *Super Sopa*. Esta sopa de letras se caracteriza por, a diferencia de una normal, la posibilidad de que cada una de las palabras no esté ubicada en una misma dirección. Además, dentro de una misma palabra, cada una de las letras puede estar colocada respecto a la anterior en dirección horizontal, vertical o diagonal.

Además de la sopa de letras, tenemos una colección de palabras a la que llamamos diccionario. Todas estas palabras son cadenas de caracteres alfabéticos de cualquier longitud. Respecto a la implementación de la sopa de letras, se trata de un tablero de $n \times n$ casillas, representado por una matriz en el código. Inicialmente se seleccionan de forma aleatoria 20 palabras del diccionario proporcionado y se colocan en la sopa de letras, el resto de caracteres de la tabla son agregados aleatoriamente.

La implementación del diccionario es la parte interesante de este proyecto. La estructura que utilizaremos para representar este conjunto de palabras va variando entre las 4 posibilidades propuestas por el enunciado de la práctica:

- Vector ordenado
- Trie con implementación de Patricia
- Filtro de Bloom
- Tabla de hash con doble hashing

Como es de suponer, para cada estructura, un nuevo algoritmo y razonamiento lógico ha sido pensado e implementado, de acuerdo a las especificaciones de cada uno de los métodos. Tal y como veremos, cada una de estas estructuras trae consigo sus respectivas ventajas e inconvenientes.

El objetivo es encontrar, en el menor tiempo posible, todas las palabras válidas que se escondan en la Super Sopa. Recordemos que una palabra válida es cualquier palabra que se encuentre en el diccionario, aunque no fuera seleccionada al inicio para estar colocada. Partiendo de esta base podemos deducir que cualquier algoritmo correcto debería encontrar un mínimo de 20 palabras siempre.

2. Vector ordenado

En esta implementación de nuestro diccionario utilizaremos un vector de strings ordenado. Esto nos permitirá aplicar el algoritmo de búsqueda dicotómica que optimiza en gran medida la búsqueda de una posible palabra del diccionario presente en la sopa de letras.

En primer lugar crearemos un vector de strings de dimensión n , siendo n el tamaño de la entrada y, por tanto, tendremos coste espacial $O(n)$.

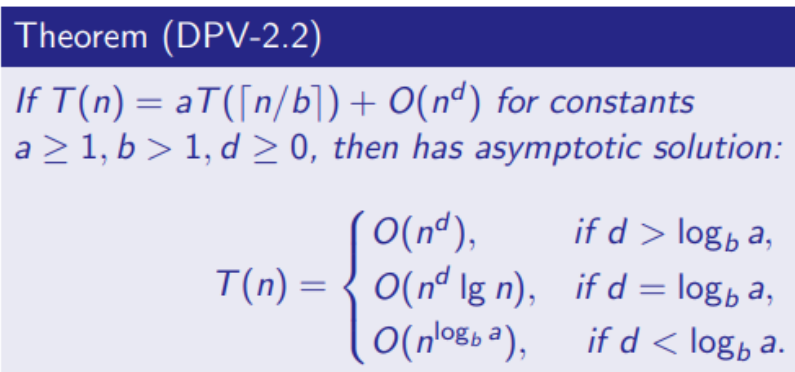
Cuando todas las palabras hayan sido insertadas en el vector realizaremos la llamada a la función `sort` de la librería de C++ "algorithm", la cuál nos permitirá ordenar el vector en orden creciente lexicográficamente con coste $O(n \cdot \log(n))$, ya que el algoritmo está basado en comparaciones. Esta acción es necesaria para posterior búsqueda.

Una vez tenemos nuestro diccionario creado se llama a la función `searchingWords` desde cada posición (i, j) de la sopa, la cual consiste en una búsqueda en profundidad de todas las posibles palabras que puedan empezar desde la posición (i, j) .

En la implementación de este DFS para comprobar si la potencial palabra almacenada en `currentWord` es una de las palabras del diccionario llamamos a la función `dichotomousSearch` que implementa una búsqueda binaria. `dichotomousSearch` es una función recursiva con 3 posibles casos:

- `currentWord` no forma parte del diccionario -> devuelve 0.
- `currentWord` es prefijo (haciendo uso de la función `find` de la librería string) -> devuelve 1.
- `currentWord` es una palabra del diccionario -> devuelve 2.

Para calcular el coste nos hemos basado en el Teorema Maestro de las Recurrencias Divisorias:



Theorem (DPV-2.2)

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for constants $a \geq 1, b > 1, d \geq 0$, then has asymptotic solution:

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a, \\ O(n^d \lg n), & \text{if } d = \log_b a, \\ O(n^{\log_b a}), & \text{if } d < \log_b a. \end{cases}$$

Figura 1: Teorema maestro de las recurrencias divisorias

Aplicándolo a nuestra función obtenemos que:

- $a = 1$
- $b = 2$
- $d = 0$

El valor de a está ligado al número de llamadas recursivas que como mucho será 1 por cada ejecución de la función.

El valor de b depende de en cuantas particiones se divida el vector para cada llamada recursiva siendo, en este caso, 2 ya que se parte el vector por la mitad.

El valor de d se obtiene del valor máximo de las operaciones ajenas a la recursión. En *dichotomousSearch* todas son $O(1)$, excepto la operación *find* que es $O(\text{currentWord.size}())$. A pesar de que *find* dependa de la entrada del tamaño de *currentWord* siempre será despreciable en comparación a n . Por tanto, no hay prácticamente diferencia entre $O(1)$ y $O(\text{currentWord.size}())$.

Finalmente, vemos que estamos en el caso donde $d = \log_b a$ y por eso nuestro coste es $O(\log(n))$.

3. Trie

Según *Wikipedia*, un trie es una estructura de datos de tipo árbol que permite la recuperación de información. A partir de esta breve definición y más concretamente, podemos describir un trie como un árbol con un número de hijos no delimitado previamente que se utiliza para almacenar un conjunto de palabras. Dicho árbol se estructura de modo que cada letra de la palabra en cuestión se encuentra en un nodo distinto, por tanto los hijos de un nodo definen las diversas posibilidades de palabras distintas que pueden continuar a partir del nodo padre en cuestión. Por ejemplo, en la Figura 2 podemos ver cómo resultaría la representación de un trie para las palabras *pot*, *past*, *pass* y *part*.

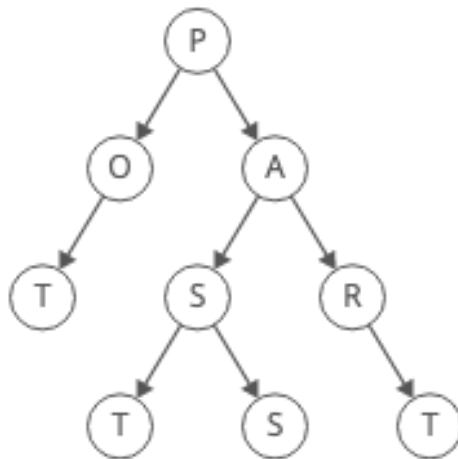


Figura 2: Representación del trie para las palabras *pot*, *past*, *pass* y *part*.

La búsqueda en la matriz que conforma la sopa de letras la hemos planteado a partir de una estrategia de backtracking. Desde cada una de las palabras de la sopa de letras, empezamos una búsqueda recursiva para encontrar palabras que se encuentren en el diccionario. A medida que vamos avanzando en la exploración se va acumulando la palabra hasta el momento y las posiciones de cada una de sus letras para que, en caso de que sea una palabra válida, retornar sus posiciones.

Para cada vuelta de la recursión, si la palabra hasta el momento existe como palabra final en el diccionario, añadimos la palabra y sus posiciones a la solución y retornamos. Por otra parte, si existe un hijo con la palabra acumulada como valor continuaremos con el backtracking pero nos moveremos en el trie al hijo en cuestión para continuar la búsqueda desde ahí. Además de esto, tendremos que resetear la palabra acumulada, ya que dentro de los hijos se encontrarán los posibles sufijos del vocablo formado hasta el momento. En el tercer caso, puede ser que la palabra acumulada sea el prefijo de alguno de los hijos del nodo en el que estamos dentro del diccionario, en este caso seguiremos el mismo proceso que antes, pero sin resetear la palabra hasta el momento ni desplazarnos dentro del diccionario. En cualquier otro caso, retornamos sin aumentar el proceso de backtracking.

Vamos a centrarnos ahora en nuestra implementación para realizar la búsqueda de palabras en la sopa de letras. Hemos creado una clase adicional *Trie* compuesta por todos los atributos y métodos necesarios para el correcto funcionamiento.

En cada uno de los nodos almacenamos la palabra que define dicho nodo, un *map* que muestra todos los nodos hijos que derivan de ese nodo padre y un booleano que define si dicha palabra es una palabra del diccionario o simplemente un prefijo de otra palabra. La necesidad de este booleano viene dada por la existencia de palabras completas que pueden llegar a ser prefijos de otras palabras más largas. Respecto a los métodos, tenemos un total de 6 funciones implementadas para los trie, además de la propia constructora de la clase.

La función *insert* permite insertar en el árbol una nueva palabra carácter a carácter. Si hay otras palabras que comparten el mismo prefijo, estas palabras comparten parte del camino a través del árbol tal y como hemos visto en el ejemplo de la Figura 2. Como veremos más adelante, con el algoritmo *Patricia* aplanamos el árbol y lo hacemos más eficiente siempre y cuando sea posible. Debido a esta inserción carácter a carácter, el coste asintótico de insertar una nueva palabra en la estructura de datos sería lineal al número de letras en dicha palabra.

$$O(\text{word_size}) \approx \Theta(1)$$

La función *search* tiene como objetivo indicar si existe o no una palabra completa dentro del trie igual al parámetro pasado. El procedimiento que sigue es el siguiente: El algoritmo de búsqueda va mirando letra a letra y acarreando si desde el nodo actual existe un nodo con ese mismo valor parcial o si no existe ningún nodo que contenga como prefijo el valor parcial calculado hasta el momento. En el primero de los casos, nos movemos al nodo hijo correspondiente, reiniciamos la palabra parcial y continuamos con la búsqueda desde ese hijo. En el segundo caso simplemente retornamos que no existe dicha palabra en el árbol, ya que si ningún nodo contiene ese valor como prefijo entonces no habrá ningún camino por el que continuar la búsqueda. Al final retornamos cierto si nos encontramos en un nodo marcado como palabra final, y falso en cualquier otro caso. Al igual que con la función de inserción, la búsqueda letra a letra provoca que tengamos un coste asintótico lineal al número de letras en la palabra.

$$O(\text{word_size}) \approx \Theta(1)$$

La función *existChildrenWithKey* retorna si existe, desde un nodo padre, un hijo cuyo valor sea igual al parámetro pasado. Para retornar este valor, únicamente hacemos uso de una expresión booleana por lo que el coste asintótico será igual a la búsqueda del hijo entre los hijos del nodo padre. Al estar implementado como un map, el coste es:

$$\Theta(\log \text{num_children})$$

La función *existChildrenWithKeyPrefix*, retorna si existe, desde un nodo padre, un hijo cuyo valor contenga como prefijo la palabra pasada por parámetro. En esta función recorremos también todos los nodos hijos del padre, por tanto el coste asintótico es el siguiente.

$$\Theta(\log \text{num_children})$$

La función *nodeWithKey*, devuelve el sub-Trie que tiene como valor raíz la clave pasada por parámetro a la función. Al tener guardados los hijos de un padre en una estructura map, la búsqueda tiene el coste siguiente:

$$\Theta(\log \text{num_children})$$

La función *Patricia*, aplica el algoritmo con ese mismo nombre para aplanar, si es posible, el trie instanciado. En el siguiente apartado vamos a ver en mayor profundidad la aplicación de dicho algoritmo.

3.1. Patricia

Una vez tenemos un trie instanciado, podemos aplicar el algoritmo de Patricia para optimizar el espacio del árbol de tal manera que cada nodo que sea hijo único se “fusiona” con su padre. De esta manera obtenemos ganancias tanto de espacio en memoria como en tiempo de ejecución, no sólo comprimimos la información sino que hacemos las búsquedas a través del árbol más rápidas y eficientes.

Esta función tiene una aplicación recursiva tal que se empieza en el nodo raíz y se va recorriendo cada uno de los nodos del trie. Si un nodo no tiene hijos simplemente no continuamos con la recursión. En caso de que tenga algún hijo, diferenciamos 2 distintos casos:

1 - El nodo actual no es una palabra final y tiene un único hijo, entonces se procede con los siguientes pasos:

- La palabra del nodo actual pasa a ser la suya misma más la del hijo.
- Si el hijo del nodo actual era palabra final, el nodo actual pasa a ser palabra final.
- Los hijos del padre ahora són los hijos a los que apuntaba su hijo.
- Se aplica patricia recursivamente al nodo actual.

2 - En cualquier otro caso, aplicaremos el algoritmo de patricia a cada uno de sus hijos de forma recursiva: El nombre de cada uno de los nodos inmediatamente hijos tomará como valor el resultado de la posible unión con los hijos del siguiente paso (que son nietos desde donde hemos hecho la llamada a patricia). Después de esto, se eliminan los punteros a los trie hijos antiguos y se añaden los nuevos hijos “*apatriciados*” como nuevos hijos del nodo padre.

En la imagen 3, podemos ver el resultado de un posible trie después de aplicar el algoritmo descrito anteriormente. Como podemos ver, hay caracteres que se han unido en un único nodo ya que entre ellos no habían bifurcaciones.

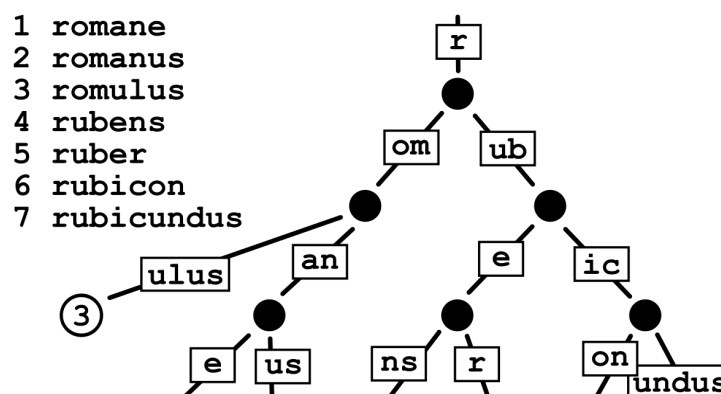


Figura 3: Trie después de aplicarle el algoritmo patricia

Tal y como está definido el problema de la Super Sopa, nunca hará falta aplicar el algoritmo de patricia más de una vez (después de crear el diccionario no se añaden más palabras). Es por esto que hemos optado por una aplicación a posteriori de la inserción de todos los elementos del trie, en lugar de utilizarlo de forma dinámica con la inserción de nuevas palabras.

Cada palabra aplicará patricia a cada uno de sus hijos, siempre que sea posible, por tanto, el número máximo de veces que llamaremos a la función patricia de forma recursiva es igual al número de palabras que existen en el trie. Siguiendo esta lógica, podemos afirmar que el tiempo asintótico de esta función es el que vemos a continuación.

$\Theta(\text{num_words})$

El principal problema que nos encontramos con esta implementación del diccionario fue el desarrollo de la propia clase trie y la previa fase de investigación antes de programar nada. Al principio planteamos el árbol de búsqueda como un conjunto de nodos en el que cada uno podría coger por valor únicamente un único carácter. Esta implementación incluso nos llegó a funcionar, el problema surgió a la hora de implementar Patricia. Este algoritmo conllevaba la suma de caracteres formando nuevas palabras lo que no era posible en nuestro concepto de árbol trie. Esto nos obligó a tener que implementar los atributos de nuestra clase y por ende, sus métodos. Es cierto, sin embargo, que teniendo ahora el producto final podemos afirmar que conceptualmente el espacio que se llega a ahorrar en un árbol trie con patricia es mucho mayor. Más aún si hablamos de diccionarios tan extensos como el del Quijote, en el que probablemente muchas de las palabras tienen caminos únicos en alguna sección.

Más adelante, en la sección de experimentación, comentaremos los tiempos de ejecución tanto de la inicialización como de la búsqueda de esta implementación del diccionario así como la complejidad temporal de ambos casos.

4. Filtro de Bloom

En esta implementación de nuestro diccionario utilizaremos una estructura de datos conocida como Filtro de Bloom. Es una estructura de datos probabilística que es usada para verificar si un elemento es miembro de un conjunto. Una característica importante de esta estructura es que permite la aparición de falsos positivos pero no de falsos negativos, cosa que habrá que tener en cuenta para los posibles resultados. Un falso positivo es un elemento que después de pasar el Filtro de Bloom nos afirma que el elemento pertenece al conjunto cuando realmente no está.

Este Filtro de Bloom se basa en que dado un conjunto de elementos n , una probabilidad de falsos positivos p , una máscara m de bits todos a 0 y un conjunto k de funciones de hash, para agregar un elemento se aplica cada una de las k funciones de hash para conseguir k bits que poner a 1 en la máscara de bits, y para la consecuente búsqueda de este si al aplicar todas las k funciones de hash sobre el elemento a buscar devuelven 1 quiere decir que ese elemento pertenece al conjunto o que es un falso positivo, si alguna devuelve un 0 se puede asegurar que 100% no pertenece al conjunto.

La gran ventaja que nos ofrece el Filtro de Bloom es que insertar y buscar un elemento en el filtro es $O(k)$, pero debido a que k respecto a la entrada n es despreciable, se puede considerar un coste de $\Theta(1)$ lo cual supone una gran eficiencia temporal.

En nuestro caso tenemos que n es el número de elementos del diccionario, p es una probabilidad que queremos dar nosotros de falsos positivos y los valores de k y m los conseguimos aplicando estas fórmulas.

```
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));  
k = round((m / n) * log(2));
```

Para nuestra implementación hemos decidido utilizar un vector *vBloomFilter* de Filtros de Bloom de tamaño 11. En este vector guardamos en *vBloomFilter[0]* la máscara de bits de los prefijos de tamaño 2 de las palabras del diccionario, en *vBloomFilter[1]* guardamos las de tamaño 3, y así sucesivamente hasta *vBloomFilter[8]* que guardamos las de tamaño 10. Entonces nos queda que *vBloomFilter[9]* guarda cualquier prefijo de tamaño 11 o superior. Por último en *vBloomFilter[10]* guardamos las palabras que no son prefijos, es decir las palabras reales del diccionario. Para tratar estos prefijos previamente hemos tenido que crear un `vector<set<string>> prefixes(10)` para separar todos los prefijos de cada palabra cuando se crea el diccionario.

Algorítmicamente el coste de preparar los prefijos para la creación de los Filtros de Bloom es de $O(m*n)$ siendo n el número de elementos en el diccionario y m el tamaño medio de las palabras del diccionario, pero como m es despreciable en comparación a n , el coste no se ve afectado por m . Así que finalmente tenemos que es $O(n)$.

Espacialmente el coste de preparar los prefijos para la creación de los Filtros de Bloom es de $\Theta(11*n)$, es decir $\Theta(n)$.

Para insertar las palabras o prefijos en sus respectivos Filtros de Bloom solo tenemos que recorrer su vector asociado elemento a elemento y como ya sabemos que el coste de insertar un elemento en un filtro es de $\Theta(1)$, tenemos que algorítmicamente insertar estos elementos nos implica un coste $O(11*n)$, es decir $O(n)$.

Una vez tenemos nuestro diccionario creado se llama a la función *searchingWords* desde cada posición (i, j) de la sopa, la cual consiste en una búsqueda en profundidad de todas las posibles palabras que puedan empezar desde la posición (i, j).

En la implementación de este DFS para comprobar si la potencial palabra almacenada en *currentWord* es una de las palabras del diccionario llamamos a la función *search* de nuestra clase *BloomFilter*, la cual con un coste de $\Theta(1)$ nos devuelve si el elemento a buscar pertenece a nuestro conjunto o no.

Como tenemos almacenados en Filtros de Bloom diferentes las palabras del diccionario y los prefijos, primero comprobamos si el elemento es una palabra del diccionario. Si pasa el filtro añadimos la palabra a la lista de resultados, si no volvemos a buscar el elemento pero en el Filtro de Bloom de prefijos correspondiente. Si otra vez no pasa el filtro podemos asegurar que con ese prefijo no existe ninguna palabra en el diccionario. Al ser esta búsqueda de coste $\Theta(1)$ el coste de buscar las palabras únicamente será el coste del DFS.

Hemos tenido varios problemas en la implementación del código con esta estructura de datos. Con el primero de ellos ya hemos descrito la solución en uno de los párrafos anteriores, y es la explicación del por qué del uso del vector *vBloomFilter* de Filtros de Bloom. Esta inicialización distinta al resto de estructuras anteriores es necesaria para comprobación en el DFS de la existencia de prefijos, ya que es importante saber de la existencia de estos para poder descartar posibles palabras más rápidamente.

El otro problema que hemos tenido ha sido la depuración de falsos positivos, ya que la solución al problema de los prefijos se nos ocurrió más tarde. Anteriormente, el problema de los prefijos lo tratábamos en el mismo Filtro de Bloom en el que se insertaban las palabras del diccionario. Esto provocaba que la máscara de bits del filtro se llenase descontroladamente, dando como resultado una cantidad inmensa de falsos positivos que no podíamos controlar aún modificando la probabilidad de falsos positivos ni aumentando el tamaño de la máscara ni modificando las funciones de hash. Todos estos problemas se arreglaron con la implementación del tratado de los prefijos.

5. Doble Hashing

La implementación del “Doble Hashing” es una técnica que consiste en tener dos funciones para asignar datos (en este caso palabras) a posiciones concretas de una lista (o en este caso, como sabemos el tamaño máximo del diccionario, un vector). Normalmente, de esta forma si se hacen bien las funciones de hashing, se tiene codificada para cada palabra un sitio predefinido el cuál se puede acceder en tiempo constante $\Theta(1)$, sin embargo el caso peor de una búsqueda tiene coste $O(n)$.

Concretando más su funcionamiento, la primera función de Hash sirve para elegir una posición en base al elemento a insertar o buscar, si esa posición resulta estar ya ocupada entonces se recurre a la segunda función de Hash, que determina cuántas posiciones saltar desde ahí para elegir la siguiente posición. Si esta última posición está también ocupada, se repite el desplazamiento que nos da la segunda función de hash hasta que la posición no esté cogida. La fórmula para asignar la posición final es la siguiente:

$$\text{FinalPosition} = (\text{hash1}(\text{key}) + i * \text{hash2}(\text{key})) \% \text{TABLE_SIZE}$$

Donde i es un número de 0 a TABLE_SIZE .

Si nos fijamos bien esta fórmula cumple lo que hemos dicho: Empezamos desde una posición inicial $\text{hash1}(\text{key})$ y hacemos una cantidad indefinida i de saltos de tamaño $\text{hash2}(\text{key})$ hasta encontrar un sitio que no esté ocupado sin pasarnos del máximo (de aquí viene el módulo del final).

En la siguiente figura podemos ver un esquema de cómo funciona lo recién explicado más claramente:

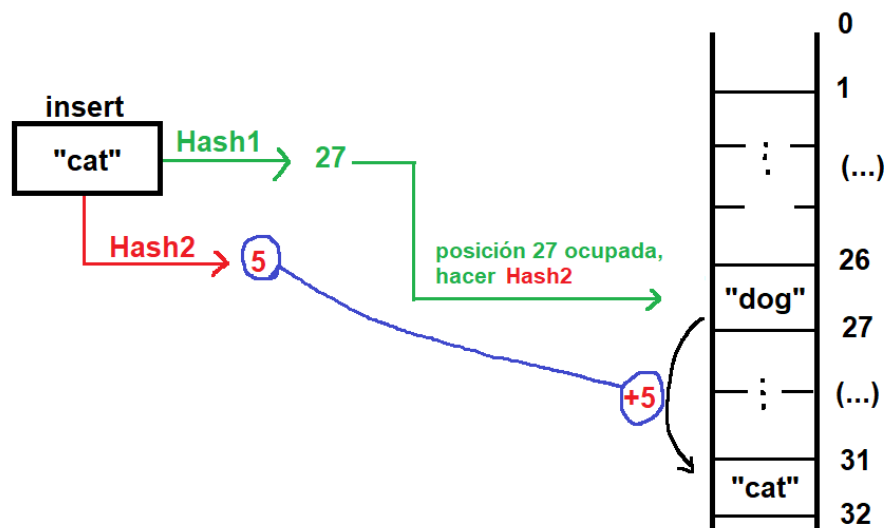


Figura 4: Esquema de funcionamiento de doble hashing

En el ejemplo de la Figura 4 se puede apreciar que se quiere insertar o buscar la palabra “cat” en el vector. Dicha palabra se envía a través de una función Hash1 que devuelve un entero indicando la posición a la que se le asignaría dicha palabra, pero resulta que ese hueco ya está ocupado por la palabra “dog”, así que se usa la función Hash2 para obtener

cuántas posiciones se van a saltar (en este caso 5). Si la posición siguiente hubiera estado ocupada por otra palabra se habría vuelto a hacer un salto de 5 posiciones desde aquí. Si intentamos acceder a una posición posterior al tamaño máximo del vector entonces se daría la vuelta y se seguiría dando el salto desde el inicio.

Tanto la primera como la segunda función de Hash están pensadas para que den números únicos o con propiedades especiales para lo que queremos obtener.

Hash1 en nuestro programa hace la suma de cada carácter de la palabra que se le pasa "key" en ASCII multiplicado por 37 (un número primo elegido arbitrariamente). En cuestiones matemáticas, este número no es importante para el resultado final, simplemente es para codificar la palabra. Durante cada paso de la suma se hace el módulo con el tamaño máximo del vector para no acceder a una posición que esté fuera de nuestro alcance. Tiene un coste de:

$$O(\text{key_size}) \approx \Theta(1)$$

Hash2 es una función un tanto más compleja, para que los saltos funcionen correctamente esta ha de cumplir varias condiciones:

- Nunca ha de retornar 0.
- El resultado ha de permitir circular por toda la tabla de Hash.
- Debe ser una función rápida.
- Debe ser independiente de Hash1.

Teniendo esto en cuenta, hemos implementado la función Hash2 siguiendo los criterios ya descritos:

- Evitamos que la función devuelva 0 haciendo al final $7 - (\text{valorObtenido}) \% 7$.
- Al devolver un número relativamente primo, no es divisor del tamaño de la tabla de Hash, por tanto el módulo siempre permitirá circular por todo el vector.
- La función recorre carácter a carácter, por tanto tiene un coste lineal respecto $|key|$.
- La codificación usa un número primo distinto a Hash1.

Coste de la función Hash2:

$$O(\text{key_size}) \approx \Theta(1)$$

Lo que hace la función *insert* es insertar una palabra a la posición que le toca de la tabla siguiendo la fórmula mencionada anteriormente (usa Hash1 y Hash2). El bucle recorre todas las posiciones posibles hasta que encuentra un hueco. Esta función tiene un coste de:

$$O(\text{key_size}) + O(\text{key_size}) + O(\text{hash_size}) \approx \mathbf{O(\text{hash_size})}$$

La función *searchWord* es una función de búsqueda que recorre la tabla siguiendo la fórmula (usa Hash1 y Hash2) hasta encontrar la palabra. Devuelve True si se ha encontrado key y False en caso contrario. Su coste es:

$$O(\text{key_size}) + O(\text{key_size}) + O(\text{hash_size}) \approx \mathbf{O(\text{hash_size})}$$

La función *getSize* devuelve el tamaño de la tabla de Hash. Su coste es:

$$\Theta(1)$$

Se puede apreciar en el código que hemos usado once tablas de doble Hashing, esto lo hemos hecho para poder buscar prefijos. Tenemos una tabla de Hash para las palabras completas y diez para los prefijos de tamaño 2,3,4... hasta 11 o más. El motivo de esto es porque si tuviéramos todas las palabras y prefijos en una sola tabla de hash entonces en el caso de buscar una palabra que no existe tendría que recorrerla entera, si tenemos los prefijos en diferentes tablas nos podemos ahorrar mucho coste. Esta decisión de implementación hace que el espacio que ocupa el diccionario en memoria sea mucho más grande, concretamente proporcional al número de letras que tiene cada palabra.

Una observación que hicimos al probar el programa con los diccionarios proporcionados es que las tablas de hash más llenas son las de los prefijos de tamaño 4, 5 y 6 mientras que las que menos palabras contenían eran las de prefijos de tamaño 2, 3 y 11 o más. Se deduce que esto ocurre porque los prefijos de 2 letras son limitados teniendo 26 letras en el abecedario (concretamente: $26 \text{ sobre } 2 = \frac{26!}{2! * (26-2)!} = 325$), los de 3 letras más de lo mismo y para los de tamaño muy grande, no existen muchas palabras con prefijos de más de 10 letras.

Como con las implementaciones de las demás estructuras de datos, primero se crea el diccionario con todas las palabras, pero esta vez insertamos todos los prefijos en la tabla que les toca.

Después empezamos la llamada recursiva para hacer backtracking en la sopa construyendo la posible palabra. Que tiene tres casos:

- La palabra parcialmente creada es una palabra existente en el diccionario, entonces se añade a las soluciones y se sigue con la recursión.
- La palabra parcialmente creada es un prefijo de una palabra existente, entonces se sigue con el backtracking para las ocho casillas adyacentes.
- La palabra parcialmente creada no está en la tabla de hash ni en los prefijos, se vuelve atrás en el backtracking.

Cada vez que se comprueba la primera o la segunda condición se ha de hacer una búsqueda en el vector de DHashing con la palabra parcial que nos pasan, que en el mejor de los casos tiene coste $\Theta(1)$ pero en el peor de los casos tiene coste asintótico $\Theta(n)$. Siendo la media $\Theta(n/2)$ que es aproximadamente $O(n)$, siendo n el número de palabras de la tabla.

Se sigue este patrón de decisiones hasta que se llega al final de la sopa de letras. Más adelante discutiremos el coste total temporal de este algoritmo. Pero adelantando los acontecimientos: no parece que esta sea la mejor estructura de datos para manejar el problema de la sopa de letras.

6. Experimentación

Para la experimentación de los tiempos de ejecución de las distintas estructuras de datos hemos utilizado la librería *chrono* de C++ y sus respectivas funciones para capturar los intervalos de tiempo en los que corrían las partes de código deseadas. Hemos dividido también el análisis en tres secciones: la inicialización, la búsqueda y el cómputo global. En la inicialización vemos los tiempos que necesita el programa para rellenar la estructura de árbol definida, por otro lado, en la búsqueda analizamos lo que tarda el programa en encontrar las palabras del diccionario que haya en la sopa de letras.

Los parámetros que hemos ido variando a la hora de medir nuestros datos son el tamaño de la matriz de la sopa de letras y las palabras que forman el diccionario. Cabe destacar que el número de filas en la sopa de letras es siempre igual al número de columnas. Hemos analizado tamaño de filas y columnas de 10, 20, 30, 40 y 50 caracteres y, respecto a los diccionarios, hemos utilizado los datos de *dracula-vocabulary*, *balena-vocabulary* y *quijote-vocabulary*. Los tres diccionarios proporcionados tienen una diferencia de número de palabras notable.

El procedimiento que hemos seguido para recoger los datos analizados es el siguiente: Para cada tamaño de fila/columna utilizado y para cada diccionario hemos tomado un total de 10 muestras de los respectivos tiempos de ejecución. A la hora de utilizar los valores en las gráficas, hemos hecho la media de los tiempos y hemos utilizado el resultado. Una vez hemos recogido toda la información, la hemos procesado a través de un script de Python para realizar las tres gráficas que veremos para cada una de las estructuras.

6.1. Vector ordenado

Tal y como hemos explicado en esta pequeña introducción vamos a analizar el vector ordenado gráficamente.

Primeramente y tal como hemos descrito en el apartado correspondiente, el coste de inicializar el diccionario solo depende del tamaño de este. Como podemos comprobar en la Figura 5 el tiempo se mantiene constante sin importar el tamaño de la sopa de letras, ya que para ordenar este vector solo nos importa cuantas palabras contenga el diccionario. Por esta razón cuantas más palabras tenga el diccionario más tarda la inicialización.

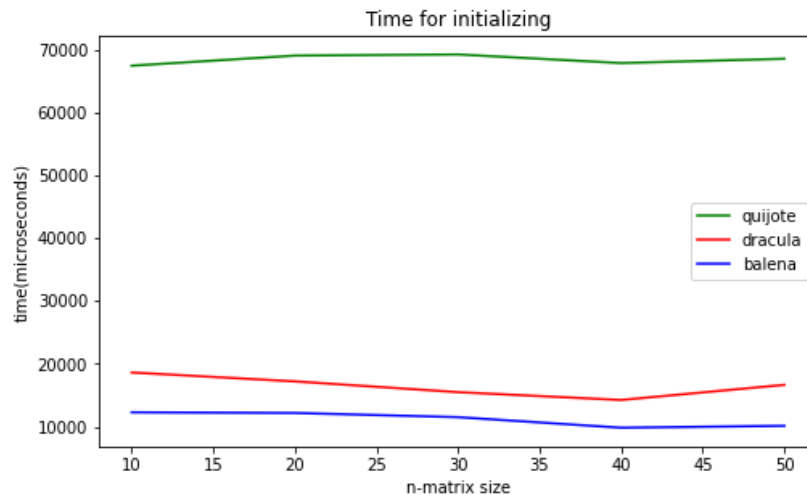


Figura 5: Gráfica de tiempos de ejecución en microsegundos para la inicialización del vector ordenado

El tiempo de búsqueda, al ser una búsqueda binaria también depende del número de elementos de entrada por eso podemos ver que la ejecución del programa con el diccionario verde siempre es mayor que el resto ya que contiene un número de elementos mayor. Además podemos observar una curva con tendencia ascendente dependiente ahora sí del tamaño de la sopa de letras, pues esto se debe que hacemos un dfs para cada posición (i,j) de la matriz que contiene la sopa y esto resulta en un coste $\Omega(n^2)$.

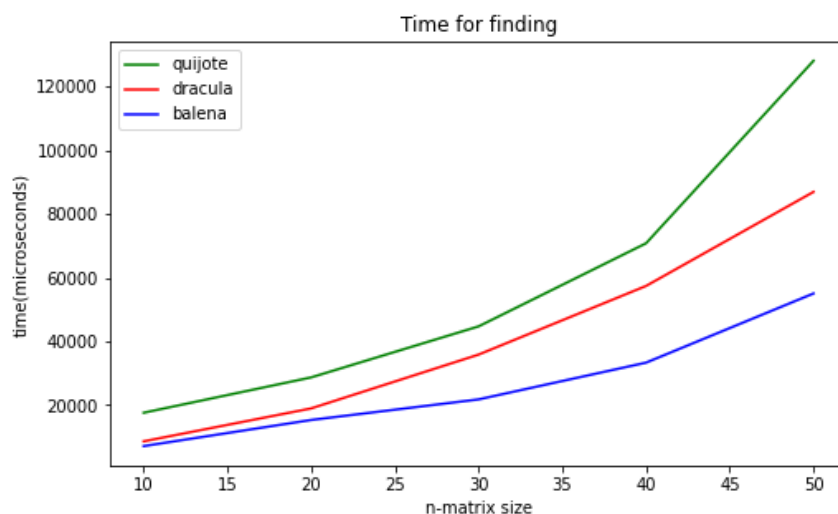


Figura 6: Gráfica de tiempos de ejecución en microsegundos para la búsqueda con el vector ordenado

En la Figura 7 computamos ambas gráficas anteriores y obtenemos esta, que sin mucha sorpresa podemos seguir viendo la curva ascendente de la Figura 6 y pero desde una altura en tiempo superior ya que el tiempo de inicialización se ha llevado un peso importante al inicio del programa.

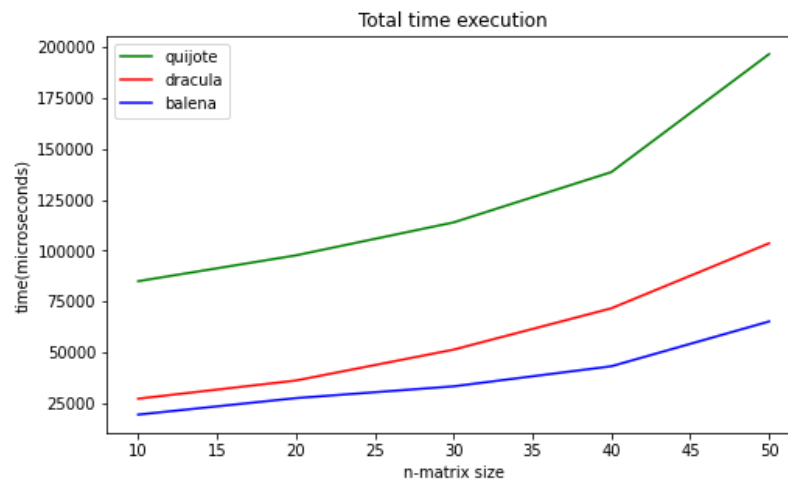


Figura 7: Gráfica de tiempos de ejecución en microsegundos para el programa con vector ordenado

6.2. Trie

Al igual que con las demás implementaciones, para experimentar los tiempos de ejecución del trie hemos utilizado la librería *chrono* de C++. Hemos estudiado los tiempos de ejecución en la inicialización, la búsqueda de las palabras y los tiempos conjuntos. Para cada uno de estos tiempos hemos hecho unas suposiciones y observaciones iniciales antes de realizar las ejecuciones.

Vamos a empezar analizando los tiempos de ejecución durante la inicialización del trie. Antes de tomar los resultados supusimos que, como resulta evidente, el número de filas y columnas en la sopa de letras no tendría ningún tipo de impacto a la hora de inicializar la estructura de datos. Por otra parte, el número de palabras en el diccionario iba a resultar decisivo a la hora de penalizar la creación del árbol.

Como podemos ver en la Figura 8, el factor decisivo para el coste temporal de la inicialización es el número de palabras a insertar en el diccionario. Los datos de quijote contienen un número de palabras muy elevado respecto a los otros dos, de ahí que tarde más del doble de tiempo en iniciarse. Tal y como habíamos supuesto, el tamaño de la matriz que conforma la sopa de letras no tiene ningún tipo de relevancia en el cálculo de estos tiempos de ejecución.

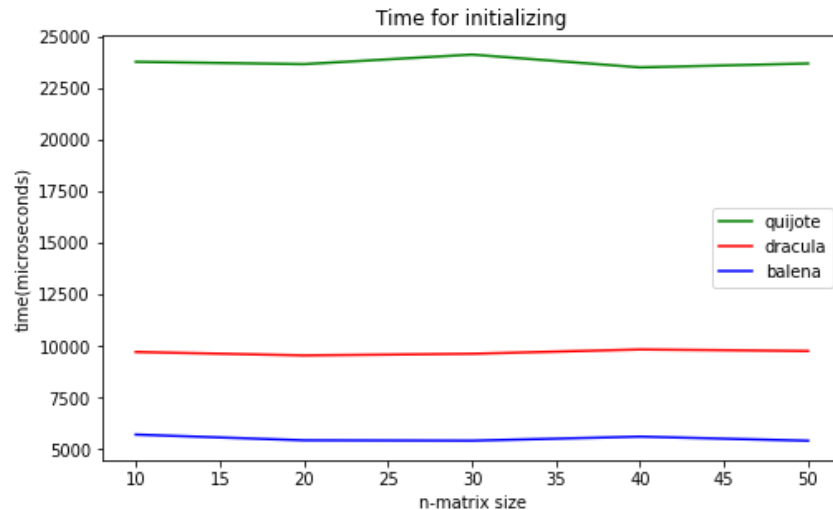


Figura 8: Gráfica de tiempos de ejecución en microsegundos para la inicialización del trie

En el siguiente punto vamos a analizar los tiempos de ejecución obtenidos durante la búsqueda de las palabras del diccionario en la sopa de letras. Igual que antes, previamente a la toma de resultados hicimos unas suposiciones de los resultados que íbamos a obtener. En este caso el tamaño de la matriz sí es un factor a tener muy en cuenta debido a la existencia de una estrategia de backtracking, además de eso pensamos que el número de palabras en el diccionario también afectaría en los tiempos, esto último veremos que no era así en realidad.

Como podemos ver en la Figura 9, a medida que el espacio de búsqueda aumenta, también lo hace el tiempo debido a que la recursividad se vuelve muy pesada. A pesar de eso, podemos fijarnos como los tres diccionarios, a pesar de tener un número de palabras muy distinto, tienen costes prácticamente idénticos. Esto quiere decir que tal y como se hacen las búsquedas en el trie, no importa cuántas palabras tengas almacenadas, ya que la búsqueda será igual de eficiente. Si pensamos en mayor profundidad nos damos cuenta de que esto tiene bastante sentido, al fin y al cabo solo nos desplazamos por las ramas del árbol que nos interesen, el resto nunca llegaremos a verlas por cómo funciona la estructura del árbol.

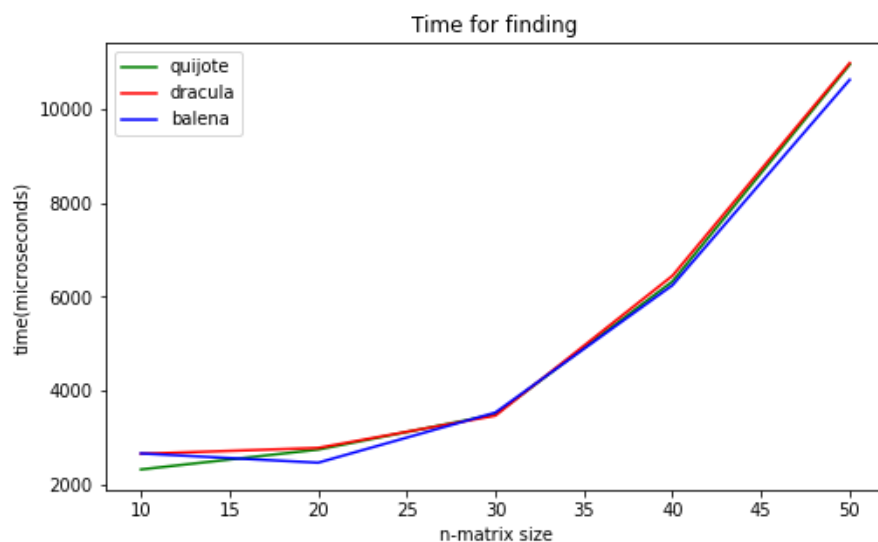


Figura 9: Gráfica de tiempos de ejecución en microsegundos para la búsqueda con trie

Por último, en la Figura 10, podemos ver el tiempo conjunto de la inicialización y la búsqueda en la sopa de letras con la estructura de trie. Se observa la influencia que tienen ambos procesos a la hora de calcular el tiempo total, pero si tuviéramos que recalcar uno de los dos, podríamos decir que el tiempo de inicialización acaba siendo el proceso que delimita más el tiempo de ejecución final. En otras palabras, podríamos afirmar que tratando con tries el número de palabras a insertar inicialmente puede ser más costoso que luego cualquier búsqueda que se quiera realizar. Las curvas de tiempo de la búsqueda respecto al tamaño de la matriz se mantienen, pero el punto de partida inicial depende totalmente de la inicialización del trie.

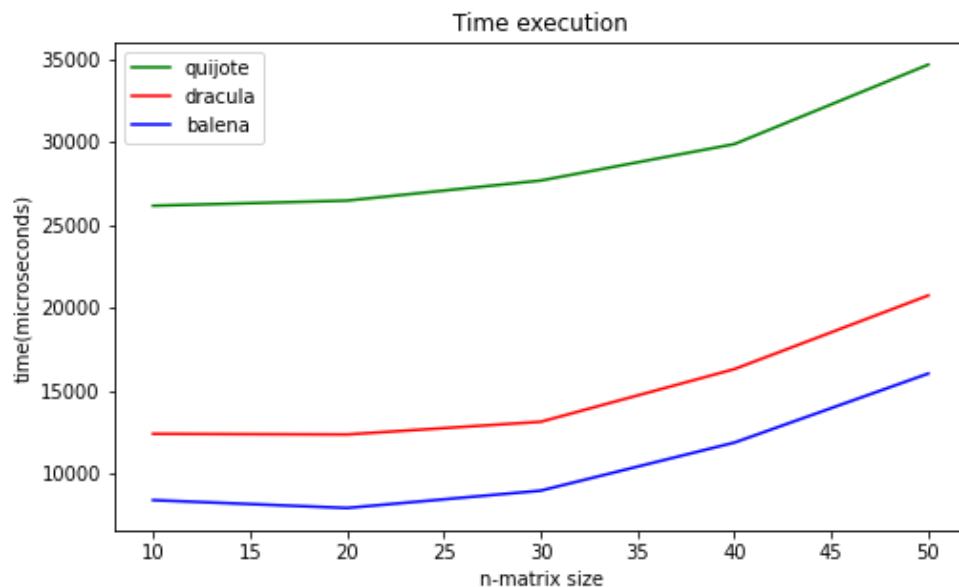


Figura 10: Gráfica de tiempos de ejecución en microsegundos para el programa con trie

Como conclusión individual de esta estructura de datos, podemos afirmar que un árbol con formato de trie y utilizando patricia es un método muy eficiente de almacenar claves alfabéticas que requieran de un gran número de accesos. La búsqueda de una palabra dentro del árbol siempre será rápida y ágil gracias al almacenamiento alfabético que mantiene la estructura. Además, obtener las palabras que empiezan por un prefijo determinado o hacer búsquedas basadas en prefijos, son también tareas idóneas para una estructura como el trie.

6.3. Filtro de Bloom

Este apartado va a ser un poco diferente a los anteriores ya que hemos partido el tiempo de inicialización en 2: construcción de los prefijos e inicialización(en este caso la inserción y llenado de la máscara del filtro). El caso del tiempo para la búsqueda sigue siendo el mismo.

Antes de todo hemos comprobado el porcentaje de aciertos de la lista de resultados obtenida con el diccionario original usando una búsqueda binaria de cada palabra de los resultados.

Sabemos que el diccionario del Quijote contiene 25583 elementos, el de Drácula 9421 y el de Mare balena 5053.

Hemos hecho el experimento para diferentes valores de p (única variable que permite modificar el valor de los falsos positivos y siendo n número de elementos del diccionario):

- $p = 1/n$ (1 error por cada n palabras)
 - Quijote -> 99.9% aciertos
 - Dracula -> 99.9% aciertos
 - Mare balena -> 99.9% aciertos
- $p = 1/(0.25*n)$ (4 errores por cada n palabras)
 - Quijote -> 99.8% aciertos
 - Dracula -> 98.5% aciertos
 - Mare balena -> 98% aciertos
- $p = 1/(0.05*n)$ (20 errores por cada n palabras)
 - Quijote -> 98.5% aciertos
 - Dracula -> 93.7% aciertos
 - Mare balena -> 84.1% aciertos
- $p = 1/(0.01*n)$ (100 errores por cada n palabras)
 - Quijote -> 95.8% aciertos
 - Dracula -> 63.1% aciertos
 - Mare balena -> 15% aciertos

Como podemos ver cuan más pequeño es el diccionario y más subimos la probabilidad de de falsos positivos, la diferencia en porcentaje de aciertos baja drásticamente.

En las Figuras 11 y 12 vemos que el tiempo de crear los prefijos y de inicializar no depende del tamaño de la sopa, ya que como hemos explicado estas dos operaciones van en función del número de elementos del diccionario y, por tanto, tiene sentido que el tiempo sea constante.

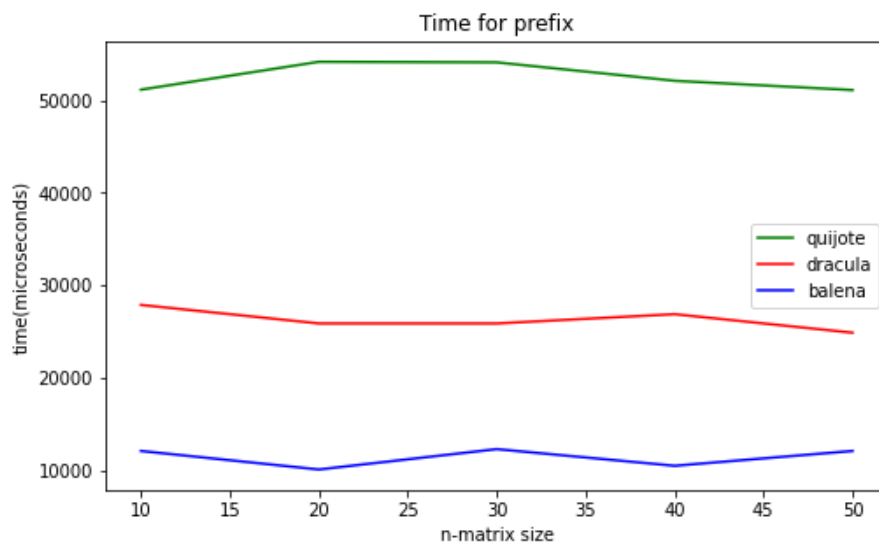


Figura 11: Gráfica de tiempos de ejecución en microsegundos para la inicialización del filtro de bloom

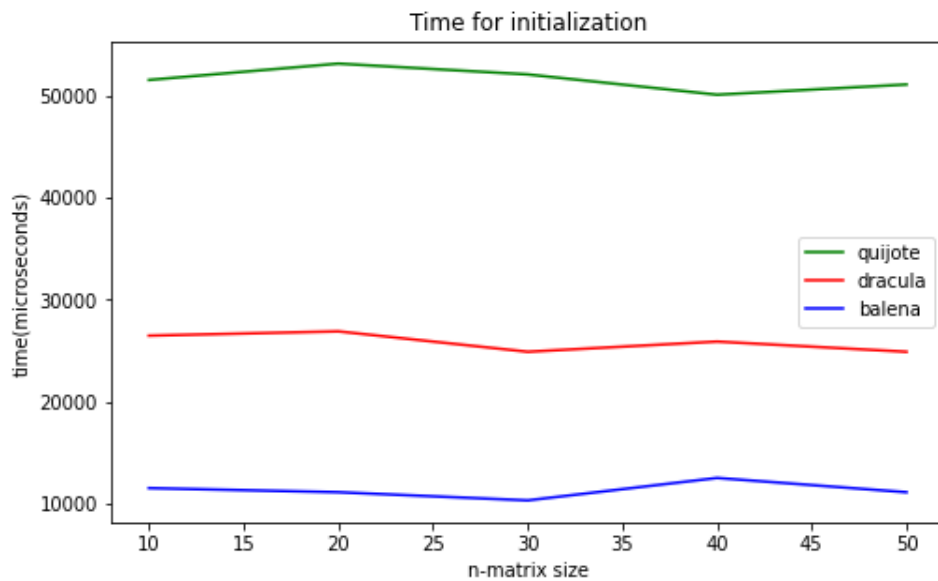


Figura 12: Gráfica de tiempos de ejecución en microsegundos para la inicialización del filtro de bloom

En la Figura 13 volvemos a ver la curva ascendente debido al coste $\Omega(n^2)$ que supone aplicar el DFS para cada posición y al haber más cantidad de 1 en los diccionarios más grandes se justifica el pequeño tiempo de más que puede tardar el DFS por posibles falsos positivos.

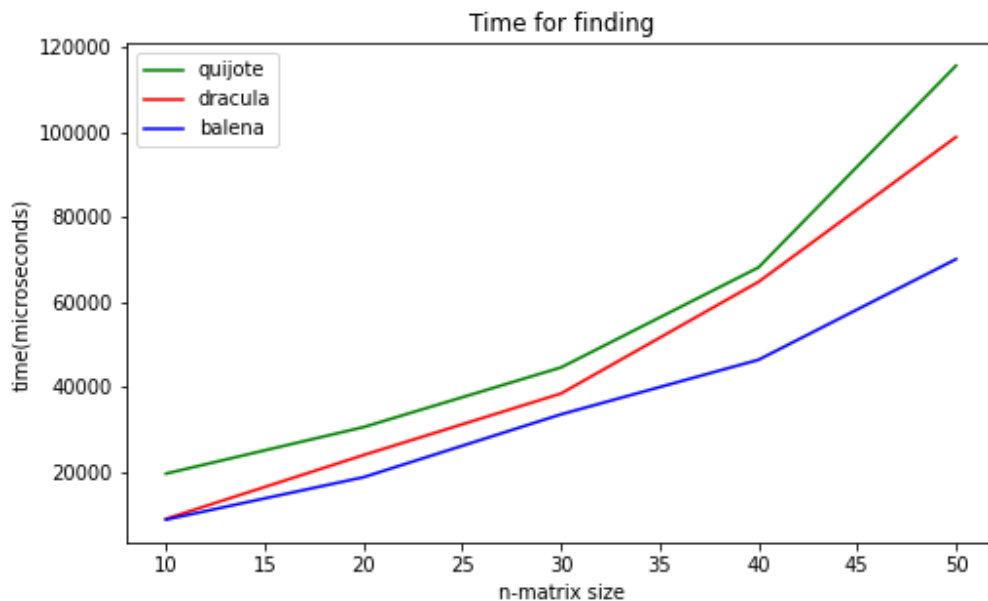


Figura 13: Gráfica de tiempos de ejecución en microsegundos para la búsqueda con filtro de bloom

Finalmente, al influir tanto la inicialización en nuestro programa podemos ver que gran parte del tiempo total lo dedica a esta función pero también vemos que a medida que aumentamos la n , este tiempo es el que gana protagonismo.

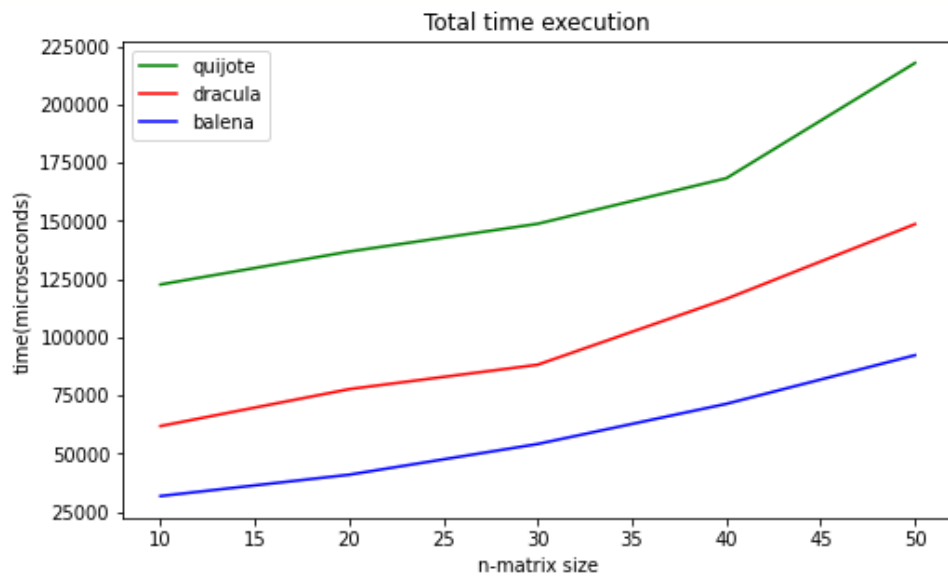


Figura 14: Gráfica de tiempos de ejecución en microsegundos para el programa con filtro de bloom

6.4. Doble Hashing

En este apartado estudiaremos el tiempo de ejecución de las diferentes partes del programa. Como se puede imaginar, lo hemos dividido en dos secciones: Inicialización de los datos y Búsqueda en la sopa de letras, y después el cómputo global del programa.

En la inicialización del diccionario se leen todas las palabras y se añaden todos los prefijos de las mismas, por tanto se ejecutará la función insert un total de $key_length * num_keys$ veces. El coste de la función insert es en el mejor de los casos instantáneo y en el peor de los casos $O(n)$, siendo la media $O(n/2)$ que es asintóticamente hablando $O(n)$.

Como podemos ver en la figura 15, el tiempo de inicialización, tanto de los prefijos como del propio *DHashing*, no varía según el tamaño de la sopa de letras, lo cuál es lógico. Además, se puede ver que estos tiempos son bastante similares a los de inicialización de las demás estructuras:

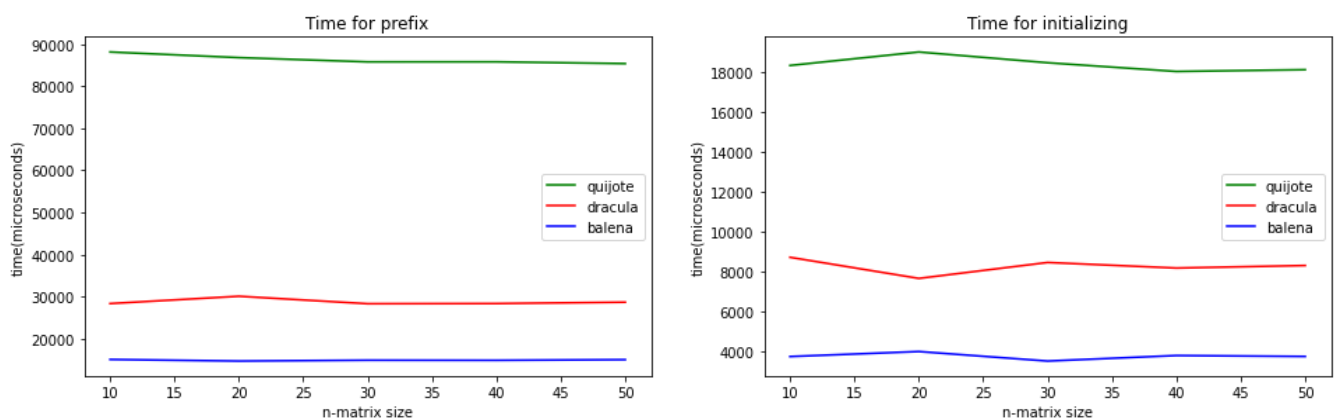


Figura 15: Gráfica de tiempos de ejecución en microsegundos para el programa con DHashing

En la búsqueda en la sopa de letras se ejecuta la función *searchWord* para cada paso del backtracking, además de hacerse para las n^2 casillas de la sopa. En la figura 16 se puede ver la evolución del tiempo de búsqueda en milisegundos según el tamaño de la matriz:

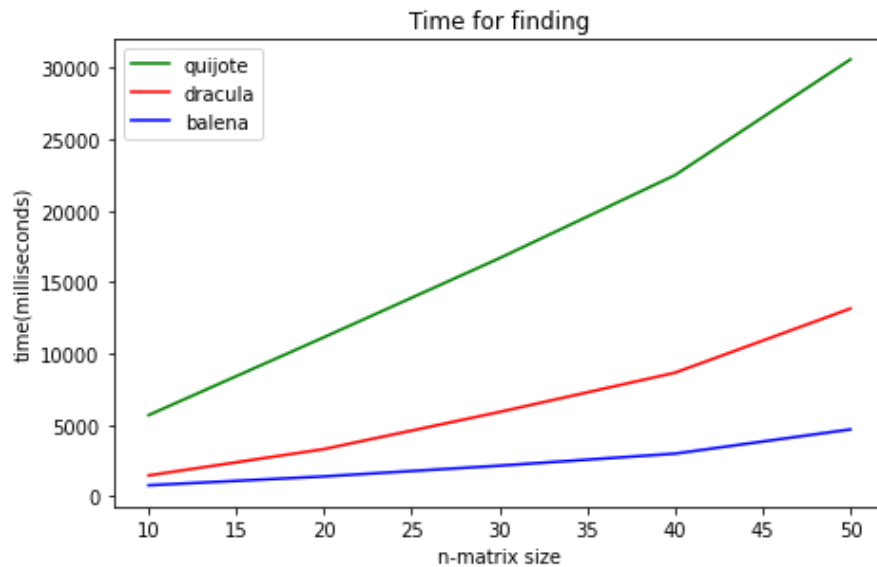


Figura 16: Gráfica de tiempos de ejecución en milisegundos para el programa con DHashing

A partir del tamaño de 40×40 se puede ver una leve subida del coste temporal que hasta el momento seguía un modelo lineal, a parte de que para las matrices más pequeñas sigue siendo mucho más lento que las demás estructuras.

También se ve que cuantas más palabras tiene el diccionario más empinada va a ser esta subida de coste ya que se han de añadir más prefijos a las tablas de hash y por ende las búsquedas serán mucho más pesadas.

Como el tiempo de la búsqueda es más grande que el tiempo de inserción no es necesario que se haga un análisis completo del programa. Se puede ver que el coste total de ejecución reside en la búsqueda. Suponemos que es porque para cada comparación de palabra que no exista en la tabla de Hash ésta se ha de recorrer entera, haciendo que como mínimo cada paso del final de un backtracking sea $O(\text{hash_size})$.

Con esto deducimos que la tabla de Hash no es la mejor forma de almacenar las palabras de un diccionario. Quizá sí lo sea para insertar palabras pero no para hacer una búsqueda parcial de las mismas.

7. Conclusión

Esta práctica ha sido de utilidad para aprender que la decisión de qué estructura de datos usar no es una decisión trivial. Dependiendo del problema y los algoritmos, unas estructuras de datos serán mejores que otras debido a la influencia en el coste final y la eficiencia que tienen sobre el programa. Además, hemos aprendido sobre tres estructuras de datos que no habíamos visto hasta la fecha: Trie, vector con doble hashing y vector con filtro de Bloom.

Entre las estructuras de datos vistas para buscar palabras en sopas de letras hemos llegado a la siguiente conclusión tras ver todos los resultados obtenidos: El Trie es la mejor para buscar prefijos, seguido del vector ordenado, el Filtro de Bloom y la peor la tabla de Hash con doble hashing.

El motivo de que el Trie sea tan bueno es que por cada paso, potencialmente, se descartan 25 ramas del árbol para llegar a la palabra final, así que la base del logaritmo de búsqueda es 25. Asintóticamente no hay diferencia con la base 2 del logaritmo del vector ordenado, pero al hacer la experimentación se nota ligeramente.

El motivo de que el doble hashing sea tan ineficiente para esta tarea es que al buscar en una sopa de letras siempre se llegará al caso en que una palabra no esté en la tabla, creando un escenario para el peor caso posible, que se recorre la tabla entera, disparando el coste.

En cuanto al filtro de Bloom, el coste de buscar es similar al del vector pero lo que hace que sea ineficiente es su coste para inicializarlo.

8. Bibliografía

OrderedVector:

<https://cplusplus.com/reference/string/string/find/>

Trie:

<https://es.wikipedia.org/wiki/Trie>

<https://www.javatpoint.com/trie-data-structure>

BloomFilter:

https://es.wikipedia.org/wiki/Filtro_de_Bloom

<https://hur.st/bloomfilter/?n=52&p=0.0001&m=&k=>

DHashing:

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

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