SuperSopa

Alex Herrero, Walter J. Troiani, Lluc Clavera, Pol Forner

Septiembre 2022

1 Introducción

El objetivo de este documento es exponer el problema de la SuperSopa y ofrecer distintas soluciones a través de diferentes estructuras de datos escritas en C++

1.1 El problema

El problema de la SuperSopa se basa en lo siguiente: Dado un tablero NxN, donde cada casilla es un carácter, tenemos que buscar todas las posibles palabras que aparezcan en nuestro diccionario. SuperSopa es muy parecido a la clásica *Sopa de Letras* salvo que en cualquier paso te puedes mover a cualquier casilla adyacente. Por ejemplo, las palabras "Cabra" y "Goku" se pueden encontrar juntando las casillas (1,1), (1,2), (1,3), (2,3), (3,2) para "Cabra" y (3,3), (4,3), (4,4) y (3,4) para "Goku".

c	a	b	p
o	n	r	v
l	a	g	u
w	d	o	k

Cabe destacar también que una misma palabra no puede repetir casillas para generarse. Por ejemplo, la palabra "Cono" no es válida, ya que para generarla solo lo podemos hacer a través de (1,1) , (2,1), (2,2) y (2,1).

Dejamos como ejercicio para el lector, para practicar, encontrar el nombre de un profesor que imparte la asignatura de algoritmia este cuatrimestre (2022-23 Q1) en la FIB en el ejemplo. Pista: **En el anexo ha sido citado.**

1.2 Nuestro trabajo

Nuestro trabajo consiste en buscar todas las posibles palabras de nuestro diccionario que aparezcan en la *SuperSopa*. Para ello implementaremos el diccionario con cuatro estructuras de datos diferentes y algoritmos muy similares para hacer las búsquedas en la Supersopa. Estas estructuras son: Vector Ordenado, *Ternary Search Tree* (abreviado TST), *Bloom Filter* y tabla con *Double Hashing*

2 Las Estructuras de Datos e hipótesis

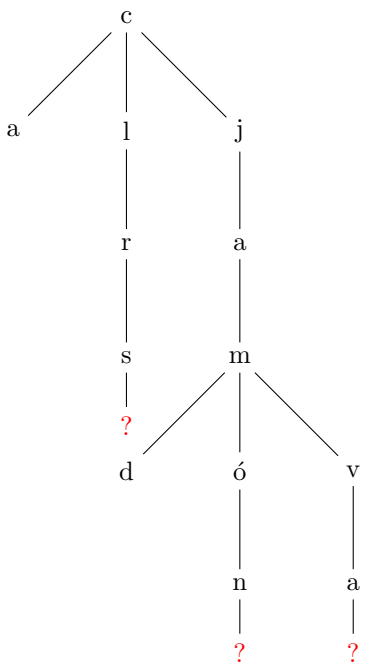
2.1 Vector Ordenado

Para implementar un vector ordenado usaremos la propia clase *Vector* de la STL de C++ y el método *Sort* de la librería *Algorithms* y hemos implementado unas funciones para buscar palabras y prefijos. La búsqueda de palabras se basa en el algoritmo de *Binary Search* que tiene un coste $O(lg\ n)$. Para encontrar los prefijos hacemos dos búsquedas binarias (una para encontrar la primera palabra que empiece por un prefijo determinado y otra para encontrar la primera palabra que empiece por el prefijo siguiente en orden lexicográfico) para encontrar el rango de todas esas palabras que empiezan por un mismo prefijo. Para ver más sobre nuestra implementación, consultar el archivo *sortedVector.cc*.

En esta sección no explicamos mucho más puesto a que consideramos que es un tipo de estructura de datos tan básica que el lector estará familiarizado con ella.

2.2 *Ternary Search Tree*

Un *Ternary Search Tree* (que llamaremos TST para abreviar) es un árbol de búsqueda para poder buscar palabras y sus prefijos. Cada nodo n tiene tres punteros a nodos que llamaremos $l(n), r(n), c(n)$ que tienen que cumplir lo siguiente: Sea n un nodo cualquier y sean $l(n), r(n)$ el nodo izquierdo y derecho respectivamente. Entonces tenemos que $n > l(n)$ y $n < r(n)$. $c(n)$ lo podemos definir como el hijo central que apunta al árbol donde se puede encontrar el siguiente carácter de la palabra después de añadir n a esta. Según la implementación, para indicar que se ha llegado a una palabra, se suele "pintar" el siguiente nodo del último o el propio último con el que acaba la palabra. Por ejemplo, el siguiente árbol es un TST:



En este TST podemos encontrar las palabras "clrs", "jamón" y "java". Ponemos el siguiente nodo de color rojo para indicar que ya hemos llegado al final de una palabra. El signo de interrogación es solo un símbolo elegido arbitrariamente que no tiene ningún valor y será cambiado cuando sea necesario.

Entonces, dada la definición anterior, sabemos que cada nodo tiene dos subárboles ordenados donde podemos realizar búsquedas en caso de que nos sea necesario. Si el lector está familiarizado con los árboles de búsqueda (que no se *autobalanceen*) sabrá que en caso medio las búsquedas tienen un coste $O(lg\ n)$. Es importante remarcar que la implementación que utilizaremos no se autobalancea (como bien pasa con otro tipo de árboles de búsqueda como por ejemplo los *Red Black Trees*) pues entonces se nos pueden dar casos en los que nuestro árbol sea similar a una lista y tengamos costes de búsqueda $O(n)$. Pues esto podría pasar, por ejemplo, insertando palabras en orden donde no tengan ningún prefijo en común.

Sin embargo, en un caso general nos encontraremos en una situación promedia, por lo que siempre que hablemos sobre TST's nos referiremos a ellos con costes $O(lg\ n)$ para las operaciones implementadas.

En la implementación dada hemos programado las operaciones de insertar elementos, buscar prefijos y saber si una palabra dada está en el TST o no.

Sobre la operación de insertar, parecida a la de buscar prefijos, nos basaremos en ir haciendo búsqueda de la palabra hasta el prefijo más grande que

podemos generar de ella y a partir de ahí insertar los nodos en las posiciones correspondientes para acabar de completar la palabra en el TST.

También comentar que las operaciones de buscar prefijos y de saber si una palabra está en el TST están muy relacionadas. Esto es ya que la operación de buscar el prefijo devuelve un TST con un trie donde todas las posibles combinaciones que contienen ya el prefijo pasado. Pues si queremos saber si este prefijo es también una propia palabra, en el árbol devuelto siempre tendrá, como mínimo, un nodo: El que nos indica que el prefijo que hemos estado formando ya es una palabra.

Para ver más detalladamente el uso de las funciones, consultar el fichero *Trie.cc* para ver la implementación.

2.3 Filtro de *Bloom*

Un filtro de *Bloom* es una estructura de datos eficiente en tamaño, pero probabilística con la cual podemos saber (con un margen de error) si un elemento está en la ED. El funcionamiento del filtro se basa en lo siguiente: Dado un array de **m** bits y definiendo **k** funciones de Hash (diferentes), cuando queramos insertar elementos en el filtro los pasaremos por las k funciones y activaremos los bits resultantes de estas a 1. En el caso de saber si un elemento está en el filtro, solo tendremos que pasarlo por las k funciones y ver si los k bits están a 1.

El problema de esta estructura de datos es que es probabilística puesto a que pueden existir elementos los cuales nunca han sido insertados, pero los k bits resultantes se encuentren activados en la ED. Dicho matemáticamente: Sean $e_1, e_2, ..., e_n$ diferentes elementos insertados en el filtro y sean $h_1(x), h_2(x)...., h_k(x)$ las k funciones de Hash donde las definimos como:

$$h_i : U \longrightarrow \mathbb{N} \tag{1}$$

Siendo U el universo de elementos.

Entonces hay una probabilidad no nula de que pase lo siguiente con un elemento e_j no insertado en el filtro:

$$h_1(e_{k_1}) = h_1(e_j), h_2(e_{k_2}) = h_2(e_j), ..., h_k(e_{k_i}) = h_k(e_j) \tag{2}$$

Donde $\forall i \in \{1, 2, ..., n\}$ $e_{k_i} \in \{e_1, e_2, ..., e_n\}$ (nótese que los distintos e_{k_i} pueden ser el mismo elemento).

Entonces si buscásemos e_j en la estructura nos diría que está, pero anteriormente hemos dicho que no está. Esto es porque el filtro "lo confunde". A estas "confusiones" se les llama *falsos positivos*.

Otra característica del filtro de Bloom es que una vez añadido un elemento, este no puede ser eliminado. Esto es debido a que para eliminarlo tendríamos que poner a 0 los bits activados por la palabra (o como mínimo 1 de ellos),

pero podría pasar que un bit sea usado por otra palabra insertada, causando la eliminación de esta.

Para inicializar los valores del filtro de Bloom, partimos de la siguiente fórmula de falso positivo **p**, para **m** grandes:

$$p = (1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{-k\frac{n}{m}})^k [3] \tag{3}$$

Donde **n** es el número de palabras insertadas, **m** el tamaño del bitArray y **k** el numero de funciones de hash

A partir de la formula anterior, el número de funciones de hash **k** que minimiza **p** es:

$$k = \frac{m \ln 2}{n} [3] \tag{4}$$

Para nuestro problema de la SuperSopa, el número de palabras insertadas **n** es conocido (es el tamaño del diccionario) y la probabilidad de falso positivo **p** es escojido previamente. Por tanto, usando la primera fórmula y la **k** óptima tenemos:

$$m = -\frac{n \ln p}{(\ln 2)^2} [3] \tag{5}$$

Para nuestro filtro de bloom necesitaremos **k** funciones de hash que sean independientes y uniformemente distribuidas. Por eso hemos optado por usar **Universal Hashing** porque nos permite generar las funciones de forma rápida y tienen un tiempo de ejecución bastante rápido.

Decimos que una clase **H** de funciones de hash es Universal si las funciones están uniformemente distribuidas y decimos que es fuertemente universal si además son independientes.

Hemos usado la construcción de Carter and Wegman para generar una clase fuertemente universal. Dado un Universo **U** y un primo **p** $\geq$ **U** la clase

$$H = \{h_{a,b}(x) | 0 < a < p, 0 \leq b < p\} [5] \tag{6}$$

es fuertemente universal con

$$h_{a,b}(x) = ((ax + b) \mod p) \mod M [5] \tag{7}$$

Para nuestro problema **U** $= 26^t$, siendo **t** el tamaño máximo de palabra. Para una $t = 13$ el siguiente primo más grande nos cabe en un registro de 64 bits que es como lo hemos implementado. Por tanto, para samples con un t $\leq$ 13 la clase será fuertemente universal y para samples con una t $\geq$ 13 seguiremos usando el mismo primo. A pesar de que ya no sea una clasae fuertemente universal, sigue funcionando de forma bastante correcta.

2.4 Tabla con *Double Hashing*

Double Hashing es una técnica utilizada en las tablas de Hash para resolver colisiones. Las colisiones se producen cuando dos elementos $e_!, e_2$ se les asigna la misma posición en la tabla (esto es que $h(e_1) = h(e_2)$ siendo $h(x)$ la función de hash que asigna elementos a posiciones de la tabla. Hay diferentes técnicas inventadas para solucionar este problema y una de ellas es la implementada en este proyecto: El Double Hashing.

Este consiste en, dentro del conjunto de claves sin usar (que llamemos B), iterar sobre ellas para conseguir una clave donde colocar nuestro elemento y que esa clave sea única. Lo ideal para este problema sería tener una función de hash $h(x)$ tal que cumpla las dos siguientes condiciones:

$$h\colon A \longrightarrow B \tag{8}$$

$$\forall y \in B \exists! x \in A\colon h(x) = y \tag{9}$$

Lo que se conoce en matemáticas como una función *biyectiva*. Para simular esta biyección en un espacio finito utilizaremos dos funciones de hash h_1, h_2 de la siguiente forma: Utilizamos h_1 para buscar una posición; Si está libre ponemos el elemento en esta, si no iremos sumando i veces la función h_2 hasta poder encontrar una posición. Esto es:

$$h(x) = (h_1(x) + i \cdot h_2(x)) \mod n \tag{10}$$

Donde $i \in \{1, 2, ..., n\}$

Para escoger la segunda función de hash es importante que cumpla: $(h_2(x) \neq 0, \forall x \in A)$ porque si no $h(x) = h_1(x)$ para ciertos x. También debería devolver valores co-primos al tamaño de la tabla, porque si no no se cumpliría el requisito de biyectividad. Para satisfacer la última condición nosotros hemos ideado una tabla que su tamaño sea siempre la potencia de 2 más cercana y la segunda función de hash que siempre devuelva números impares, entonces sea cual sea el número generado, siempre se cumplirá la condición de coprimalidad. Este último factor se basa en lo siguiente:

Lema 1. *Cualquier potencia de 2 y cualquier número impar serán siempre números co-primos.*

Proof. Sea m la potencia de 2 y sea n el número impar. Entonces m = 2^b y n = 2k + 1 para un $k \in \mathbb{N}$. Entonces, por el teorema fundamental de la aritmética, tenemos que m y n son productos de números primos. m es producto de b 2's (es su único factor primo) y n sea cual sea su producto de primos, el dos no puede estar entre ellos, ya que si no sería un número par. Entonces m y n no tienen factores primos en común. Entonces MCD(m,n) = 1. □

Otra ventaja de que la tabla sea la siguiente potencia de 2 más cercana respecto al número de elementos, con unos ajustes adicionales en el código, es que el load factor (del que hablaremos a continuación) nunca excederá ciertos

límites (nosotros hemos impuesto que nunca sea mayor de 0.75), eliminando la posibilidad de que el rendimiento de esta estructura colapse por la ocupación de la tabla.

En peor caso, las operaciones de esta tabla tiene un coste $O(n)$, pero será amortizado en $O(1)$ teniendo en cuenta este *load factor*.

El factor de carga α (también llamado *load factor*) que se define como $\alpha = n/m$ donde n es el número de elementos y m es la capacidad de la tabla de hash. Este factor, a primera vista irrelevante, es determinante en cuanto a la rapidez de las búsquedas, dándonos una aproximación de cuánto, en promedio, tardarán estas si tiene éxito o no, ya que en caso de búsqueda *unsuccesful* y *succesful* es:

$$Unsuccesful \approx \frac{1}{1-\alpha} \tag{11}$$

$$Succesful \approx \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha}) \tag{12}$$

Estas fórmulas son cada vez más precisas cuando el número de elementos tiende a infinito.

2.5 Hipótesis antes de experimentar

Después de haber analizado las estructuras de datos, pensamos que tanto el TST como el Vector Ordenado van a tener ventaja respecto al filtro de Bloom y al Double Hashing, ya que podemos usar propiedades de búsqueda de prefijos de forma eficiente y explotarlo para la búsqueda. En concreto, nos atrevemos a decir, por este orden, las ED's más eficientes para este problema

1. TST
2. Vector Ordenado
3. Filtro de Bloom
4. Double Hashing

Como hipótesis nos atrevemos a decir que el TST será el ganador puesto a que precisamente está especializado en la búsqueda eficiente de palabras y de prefijos. Un vector ordenado lo podremos adaptar a estas funcionalidades, así que por ello lo ponemos en segunda posición. Al final, como requiere más esfuerzo de adaptación para realizar las podas como con las otras dos ED's, ponemos al filtro de Bloom y al Double Hashing (en 3.º y 4.º puesto, ya que consideramos que el filtro de Bloom al final nos dará una respuesta de forma más eficiente (aunque pueda dar falso positivo) frente al Hashing que puede darnos un tiempo lineal en el peor caso)

3 Algoritmos de búsqueda según la estructura de datos

En todos los algoritmos descritos vamos a dar la idea sobre cómo funciona en pseudocódigo. El código en C++ utilizado para la búsqueda de palabras se adjunta en el archivo *SuperSopa.zip* junto a sus perspectivas especificaciones de implementación. El código común para todas las EDs es el siguiente:

```
procedure SOLVER(T, ED, S)
    for Pos in T do
        iSolver(T, ED, S , Pos)
    end for
```

Donde T es el tablero, ED es la estructura de datos y S es una palabra. La función iSolver es un algoritmo de backtracking que comprueba todas las combinaciones de palabras posibles. Se define de la siguiente forma:

```
procedure ISOLVER(T, ED, S, Pos)
    if not Pos.visited then
        S = S + T[Pos]
        Pos.visited = true
        if ED.exists(S) then
            insertar S en la solución
        end if
        //Comprueba si hay alguna palabra que contenga el prefijo S
        if ED.isPrefix(S) then
            for Ady in (T, Pos) do //los adyacentes a la posición
                iSolver(T, ED, S, Ady)
            end for
        end if
        Pos.visited = false
        S.pop() //Quitar la letra puesta al final
```

Hemos decidido implementar algoritmo de Backtracking con podas en función de los prefijos por las siguientes razones:

- En un principio la palabra puede estar en cualquier posición de cualquier forma, así que no tenemos una garantía de dejar de buscar en un lado en concreto o no
- Usando prefijos, tampoco aumentaremos potencialmente las llamadas de backtracking puesto a que cuando vea que un prefijo no puede ser posible no expandirá más.
- Hemos estado investigando y no encontramos otro algoritmo que se adapte bien al problema de la *SuperSopa*.

También consideramos que el hecho de usar un algoritmo de backtracking, en un caso medio, no nos va a suponer un coste realmente problemático. Pues nos basamos en lo siguiente:

Lema 2. *El coste esperado de las llamadas de Backtracking es* $O(1)$ *para buscar una sola palabra de tamaño indeferente.*

Proof. Sea Σ el alfabeto y supongamos que todas las letras son equiprobables. Entonces, dada una palabra $W = xV$ y un $i, j \in \{1, ..., n\}$ tenemos:

$\mathbb{E}$ [*runtime* de buscar xV a partir de la casilla $a_{i,j}$] $\leq \mathbb{P}$ $[a_{i,j} = x] \cdot (8 \cdot \mathbb{E}$ [Buscar V a partir de $a_{i,j}$]) $\implies$ Si definimos la siguiente función:

$$f(n) = \frac{1}{|\Sigma|} \cdot 8f(n-1) \tag{13}$$

Como lo anteriormente descrito entonces

$$f(n) = K \cdot (\frac{8}{|\Sigma|})^n = O((\frac{8}{|\Sigma|})^n) \tag{14}$$

Si sabemos que $|\Sigma| \geq 8$ entonces $\mathbb{E}$ [*runtime* de buscar xV a partir de la casilla $a_{i,j}$] $= O(1)$ Entonces sabemos que buscar una palabra tiene coste $O(1)$. □

Con esto en mente podemos suponer que el coste de buscar m palabras no tendrá un coste demasiado superior a $O(m)$.

El algoritmo anteriormente escrito tiene coste *esperado* de $O(n^2 m f(m))$ donde $f(m)$ es el coste de buscar en la estructura de datos: Para el vector y para el filtro de Bloom siempre serán $O(lg\ m)$ y $O(1)$ respectivamente. En el peor caso de la tabla de Hash y del TST pueden llegar a ser costes de $O(m)$, pero serán casos excepcionales, así que consideraremos que el TST siempre tendrá coste $O(lg\ m)$ (ya que supondremos que estará aproximadamente balanceado) y la tabla de Hash $O(1)$ pudiendo dar, a veces, costes $O(m)$.

A demás, también pensamos en iterar por nuestra estructura de datos (menos en el filtro de bloom) y buscar de una en una todas las palabras del diccionario en la sopa, este algoritmo también tendría coste $O(n^2 m f(m))$. Sin embargo, este método solo lo llegamos a implementar en el vector y vimos que tenía un desempeño peor en los pocos samples que probamos (aunque solo lo probamos con diccionarios de tamaño 25 mil). Creemos que podría ser una muy buena estrategia para diccionarios pequeños, pero por falta de tiempo no hemos analizado este método. Igualmente se incluye en el archivo *diccSortedVector.cc* la función que utiliza este algoritmo alternativo.

A continuación describiremos nuestro método concreto para encontrar prefijos en cada estructura de datos.

3.1 Utilizando Vector Ordenado

Para el vector ordenado simplemente usando una búsqueda dicotómica se puede encontrar si hay alguna palabra con cierto prefijo, ya que nuestra búsqueda dicotómica, en caso de no encontrar la palabra, te devuelve el índice del lugar

donde debería estar dicha palabra. Como cualquier palabra que empiece por un prefijo dado va a aparecer más tarde que el prefijo en un vector ordenado, si la palabra que se encuentra en la posición siguiente al índice devuelto por la búsqueda no empieza por el prefijo deseado, entonces no existe palabra alguna en todo el vector que empiece por tal prefijo.

A demás, para optimizar las búsquedas posteriores, usamos una búsqueda adicional para encontrar la primera palabra que empiece por el prefijo siguiente y así obtener un segundo índice para poder trabajar sobre un subvector más pequeño. Como en las siguientes llamadas de backtracking tendremos un prefijo fijado, buscar en el subvector formado por todas las palabras que empiecen por el prefijo actual nos dará el mismo resultado, pero ahorraremos bastante el tiempo de búsqueda.

3.2 Utilizando TST

Con el TST usamos una técnica parecida a la del vector: Cuando queremos buscar todas las palabras con un prefijo p dado lo que hacemos es devolver un TST donde todas las posibles combinaciones que se pueden hacer ya incluyen el prefijo p. Esto también nos servirá para poder saber si el prefijo p es una propia palabra del diccionario, entonces este TST tendrá en la raíz una marca que indicará que es una propia palabra.

3.3 Utilizando Filtro de Bloom y Double Hashing

Es cierto que no se pueden buscar prefijos directamente en estas EDs, pero si quisiéramos hacer un experimento con pruebas lo suficientemente grandes como para poder determinar resultados tenemos que adaptar estas EDs para poder aplicar la poda de los prefijos. De este modo, nosotros hemos adaptado las EDs haciendo que al insertar una palabra también insertemos todos los prefijos posibles de esa palabra (en la práctica tendremos dos EDs, llamaremos ED_1 y ED_2 donde ED_1 contendrá todas las palabras del diccionario y ED_2 contendrá todos los prefijos posibles). Sí que es cierto que a la hora de insertar perderemos más tiempo (pues estamos haciendo más inserciones por cada palabra) pero a la hora de hacer backtracking podremos hacer podas mucho más potentes. Al final, a la hora de insertar perderemos un tiempo polinómico frente a no expandir un tiempo exponencial.

4 Haciendo pruebas con las diferentes EDs, resultados experimentales

En esta sección hablaremos sobre la parte experimental de la *SuperSopa* que hemos realizado con los archivos adjuntados.

4.1 Comandos

Sobre la compilación: El compilador que usaremos en nuestro caso es el g++ (alias de GNU C++) en la versión 9.3.0. A esta compilación le añadiremos el flag de -O2 para que el compilador haga las optimizaciones necesarias a nuestro código (sin modificar nuestros algoritmos) ya que consideramos que, para esta práctica, tiempo añadido por la falta de optimización que nos puede ofrecer el compilador no es algo que deberíamos de tener tan en cuenta. De todas formas dejaremos un *MakeFile* para que se puedan ver los comandos de compilación.

Sobre la medida de tiempo: Hemos utilizado la librería *ctime* que nos proporciona C++ para poder medir el tiempo que ha estado el programa en la función de backtracking. En el código también hacemos diferentes medidas (por ejemplo el tiempo total del programa) que pueden ser consultadas compilando los códigos. En los resultados experimentales dejaremos solo el tiempo del algoritmo de backtracking. Destacar también que hemos usado el mismo equipo para todas las pruebas, para no contaminar nuestros resultados.

Para realizar las distintas pruebas de forma cómoda hemos usado un script de shell que ejecuta automáticamente diferentes juegos de pruebas para cada una de las estructuras, tanto el script como los resultados completos de los benchmarks se adjuntan en el archivo *SuperSopa.zip*.

4.2 Resultados experimentales

Para el diccionario del quijote:

Diccionario: Quijote (Unidad de tiempo: segundos)			
Tamaño tablero	100x100	1000x1000	5000x5000
Trie	0.022384	2.23701	55.7211
Vector	0.100528	10.2088	253.735
Bloom Filter	0.067499	6.32	159.836
Double Hashing	0.062073	4.05811	101.255

Sobre el Drácula:

Diccionario:Dracula (Unidad de tiempo: segundos)			
Tamaño tablero	100x100	1000x1000	5000x5000
Trie	0.021571	2.04581	52.2364
Vector	0.090115	8.82218	220.056
Bloom Filter	0.063098	6.11393	156.737
Double Hashing	0.048291	4.60232	119.267

Sobre el Mare Balena:

Diccionario: Mare Balena (Unidad de tiempo: segundos)			
Tamaño tablero	100x100	1000x1000	5000x5000
Trie	0.014278	1.24537	30.894
Vector	0.057172	5.49409	136.252
Bloom Filter	0.041871	3.80214	94.9325
Double Hashing	0.030846	2.89893	72.9687

Ahora nos gustaría comprobar cómo se comportan las diferentes EDs en función del número de palabras del diccionario. Por ello la siguiente tabla corresponde a un tablero 100x100 insertando el número de palabras especificado en esta:

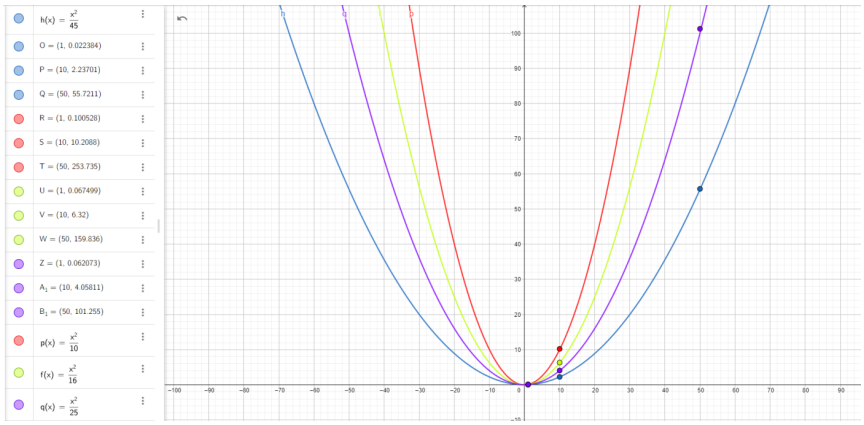
Diccionario: Random (Unidad de tiempo: segundos)			
Num palabras	100000	1000000	10000000
Trie	0.348029	2.00294	7.88243
Vector	0.865595	3.73435	13.4745
Bloom Filter	2.41236	34.4232	73.8413
Double Hashing	0.8283	4.82989	22.7875

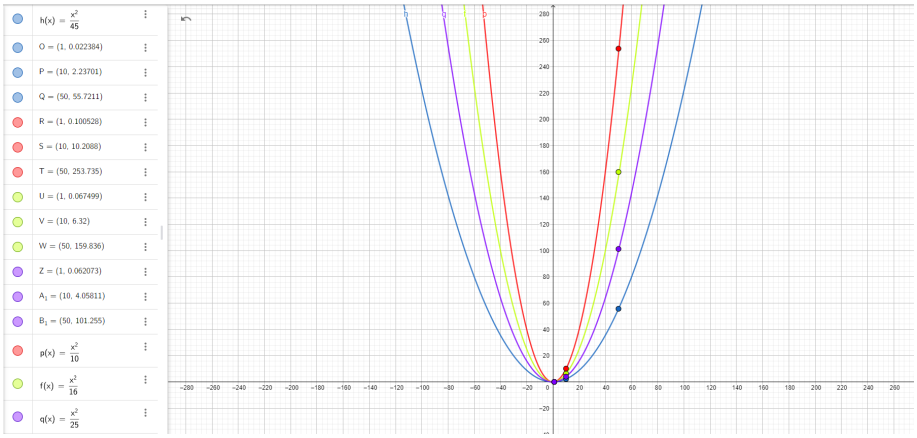
Ahora vamos a hacer pruebas aumentando los tamaños de palabra. El tablero es de siempre de 1000x1000 en este caso y el número de palabras es de 1000.

Diccionario: Random (Unidad de tiempo: segundos)			
Tamaño máximo de palabra	15	20	30
Trie	1.41058	1.45068	1.44588
Vector	4.98724	5.43706	5.77755
Bloom Filter	4.74018	5.02738	5.09197
Double Hashing	3.48647	4.32592	4.08563

Además de las pruebas mostradas hemos realizado otras pruebas adicionales para asegurarnos de la fiabilidad de nuestros resultados y para poder analizar de forma más acertada la relación entre el tiempo y nuestras variables. Podéis encontrar las demás pruebas en el archivo *SuperSopa.zip*.

A continuación dejamos gráficas que muestarn el comportamiento de las distintas estructuras al aumentar el tamaño del tablero, las pruebas usadas para hacer la gráfica han sido las realizadas con el diccionario del quijote variando el tamaño del tablero (hemos dividido entre 100 el valor del número de filas para que se pueda visualizar mejor).





- Vector
- Filtro de Bloom
- Double Hashing
- TST

5 Conclusiones

Analizando los resultados experimentales podemos ver los siguientes resultados:

- Sobre el tamaño de tablero vemos que todas las estructuras de datos se comportan de forma cuadrática, cosa que ya esperábamos desde un principio ya que hacemos $O(n^2)$ llamadas a la función para resolver
- Respecto al aumento del tamaño del diccionario vemos que, aunque el TST salga siempre como ganador, con tamaños muy grandes el vector se

comporta mejor que el *Double Hashing* y el filtro de Bloom empeora su rendimiento.

- En principio no vemos ninguna relación respecto al tamaño de las palabras (en este proyecto no consideraremos un tamaño de palabra mayor que 30 puesto a que consideramos que un lenguaje con palabras de tamaño de más de 30 sería muy poco práctico), por lo tanto asumimos que se comporta de forma constante en función del tamaño máximo de palabra

El tamaño de diccionario es el que produce más diferencias entre nuestras estructuras. Para empezar hemos notado que el comportamiento es sublineal (por lo tanto se confirma nuestra hipótesis inical de que las llamadas de backtracking serían $O(m)$, conultar apartado 3 para más información) en todas nuestras estructuras, aunque con bastantes diferencias.

Para empezar, el vector siempre va a hecer búsquedas, en el peor caso, logarítmicas en función del tamaño del diccionario, el trie, como no se autobalancea, puede llegar a hacer búsquedas no tan eficientes, pero hay tan poca diferencia con el vector que siempre sale en cabeza.

Por otro lado el filtro de bloom aumenta las probabilidades de falsos positivos al aumentar el número de palabras que tiene que comprobar, y cada vez que se produzca un falso positivo en el filtro de prefijos se va a realizar una llamada de backtracking adicional. Estas llamadas adicionales provocan que el tiempo de ejecución aumente mucho más que las otras EDs al aumentar mucho el tamaño del diccionario (cuando hay varios cientos de miles de palabras).

Respecto al Double Hashing, le ocurre algo similar, aunque menos extremo, al Filtro de Bloom. Al aumentar el tamaño de diccionario y al aumentar la comprobaciónes que tiene que hacer, aumenta el número de búsquedas lineales en el diccionario. La acumulación de búsquedas lineales provoca que se tarde más tiempo que el vector para diccionarios extremadamente grandes. Aunque cabe destacar que la diferencia entre el Double Hashing y el vector no es demasiado extrema para los tamaños que hemos comprobado nosotros (hasta 10 millones de elementos).

A demás hemos observado que nuestras estructuras tienen un mucho mejor desempeño en los diccionarios que se basan en el lenguaje natural (el Quijote, mare balena y Drácula), nuestra hipótesis para este fenómeno es que las letras en lenguaje natural no son equiprobables. Por ejemplo, las vocales son mucho más comunes que las consonantes, y hay ciertas consonantes que solo aparecen en determinadas construcciones (como la y, que solo aparece en la construcción ny en catalán). Esto provoca que, inintencionadamente, se realizen muchas más podas que en un diccionario completamente aleatorio (es la propia estructura del lenguaje que, de forma indirecta, nos hace las podas). Se deja, como se ha dicho anteriormente, las pruebas adicionales que hemos hecho en el ZIP entre-

gado (en concreto estas pruebas se pueden ver en la carpeta *samples Naturales*).

Dicho esto concluimos que:

- El TST es la estructura de datos que mejor se comporta en general. Por ello concluimos que es la mejor ED de las 4 presentadas para el problema de la *SuperSopa*.
- Si aumentamos solo el tamaño del diccionario el Vector se comporta mucho mejor que el *Double Hashing*
- Cuando aumentamos el tamaño de diccionario el filtro de Bloom se vuelve totalmente ineficiente respecto a las demás.
- Todas las EDs se comportan mejor con lenguaje natural que con uno generado aleatoriamente.

Así que si quiere resolver este problema le recomendamos que utilice un TST respecto a las demás. Si no quisiera utilizarlo, dependiendo del tamaño de su diccionario, le recomendamos que utilice el Vector ordenado o la tabla con *Double Hashing*. No vemos una utilidad clara al filtro de *Bloom* respecto a las demás EDs, a menos que se busque ahorrar memoria.

6 Anexo

En esta sección vamos a dejar diferentes *Miscellaneous* sobre el proyecto.

6.1 Pequeña guía sobre cómo funcionan los juegos de prueba dados

En el caso de que alguien quisiera replicar nuestro trabajo para hacer las pruebas dejamos una pequeña *guía* para que sea posible la réplica de este.

Nuestro *inputs* funcionan de esta forma: Para comenzar se lee, por este orden, el tamaño del tablero, el tamaño del diccionario, el número de palabras insertadas de forma segura en la sopa, el tamaño máximo de una palabra, el diccionario entero, las palabras insertadas en la sopa de forma segura y una sopa con esas palabras.

Así dejamos al lector la posibilidad de poder replicar nuestro proyecto si quisiera con nuestros propios juegos de prueba.

Se adjuntan todos los samples que hemos utilizado en el archivo *SuperSopa.zip*, junto a un script para ejecutarlos.

Como notación hemos utilizado n para el tamaño del tablero, m para el número de palabras y k para el tamaño máximo de palabras. Así que la mayoria

de juegos de prueba que dejamos su nombre es del estilo "sample-n-m-20-k.txt". El 20 es solo para indicar que en la sopa se tienen que colocar, al menos, 20 palabras. En los Samples de lenguaje natural (los del Quijote, Drácula y Mare Balena el número que sale en el nombre del archivo es solo el tamaño del tablero ya que las demás variables ya están definidas desde un principio ya que es el propio archivos que nos dan como ejemplo en la asignatura)

6.2 Referencias a las fuentes de información

1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. [Capítulo 11]

2 Dinesh P. Mehta and Sartaj Sahni. *Handbook of Data Structures and Applications, 1rst Edition* 2004 [Capítulos 9 y 28]

3 Wikipedia *Bloom Filter* https://en.wikipedia.org/wiki/Bloom$_f ilter$ [Visitado el día 26 de Septiembre de 2022]

4 Phillip G. Bradford and Michael N. Katehakis, *A probabilistic study on combinatorial expanders and hashing*
http://apdalab.org/mnk/papers/AProbStudyExpandersAndHashing.pdf

5 Conrado Martínez, Transparencias del MIRI sobre Hashing
https://www.cs.upc.edu/ conrado/docencia/ra-miri/10-RA-MIRI-Hashing.pdf