Estudio de algoritmos de Hashing

Àlex Osés Laza, Albert Segarra Roca

Algorismia

FIB • UPC

11 de enero de 2016

Resumen

En este documento analizamos el coste de varios algoritmos de búsqueda de palabras en un diccionario. Concretamente, hemos implementado y analizado 5 algoritmos, entre los que se incluyen 3 algoritmos de hashing, un algoritmo de búsqueda en árbol tipo trie, y un algoritmo de búsqueda binaria en tabla ordenada.

Índice

1. Introducción **3**

2. Descripción de los algoritmos **3**

2.1. Búsqueda Binaria . . . 3

2.2. Tabla de hash con encadenamiento . . . 4

2.3. Tabla de hash tipo Hopscotch . . . 4

2.4. Filtro de Bloom . . . 5

2.5. Búsqueda en árbol Trie . . . 5

3. Documentación de los experimentos **6**

3.1. Algoritmo de generación . . . 6

3.2. Ejecución de los algoritmos . . . 6

4. Análisis de resultados **7**

4.1. Tiempo de ejecución vs Tamaño del diccionario . . . 7

4.2. Tiempo de ejecución vs Proporción de elementos del diccionario . . . 9

4.3. Gráfico del tiempo de ejecución vs tamaño del texto en función del tamaño del diccionario 10

4.4. Memoria máxima utilizada . . . 11

4.5. Tiempo medio de búsqueda . . . 11

4.6. Tiempo medio de inserción . . . 12

4.7. Otros datos importantes . . . 12

5. Conclusiones **13**

1. Introducción

Los algoritmos de búsqueda de palabras de un texto en un diccionario son de importante relevancia en muchas áreas de la informática. Muchas veces son la base de algoritmos más complejos, o de sistemas software muy utilizados y también complejos como bases de datos o sistemas operativos. Es por eso y por la gran variedad de algoritmos de búsqueda que podemos encontrar, que es importante analizar cuál es el algoritmo que mejor se adecua a las demandas del sistema al que está destinado, ofreciendo el mayor rendimiento posible.

Nosotros hemos analizado un total de cinco algoritmos de búsqueda en diccionario, todos ellos con costes asimptóticos muy bajos en el caso medio. Son en concreto los siguientes:

- **Búsqueda binaria**
- **Tabla de hash con encadenamiento**
- **Tabla de hash tipo Hopscotch**
- **Filtro de Bloom**
- **Búsqueda en árbol Trie**

2. Descripción de los algoritmos

2.1. Búsqueda Binaria

Este algoritmo consiste de 2 partes: ordenación de los elementos del diccionario y búsqueda binaria o dicotómica en la tabla de elementos ordenada:

- **Ordenación**: Para poder descartar elementos en la búsqueda binaria la tabla debe estar ordenada. Para ordenarla hemos optado por utilizar la función `sort` de la librería estándar de `C++`. Dado que la compilación de los programas se realiza con el estándar `C++11`, podemos asegurar [3] que esta función tiene un coste asimptótico de comparaciones en caso peor de $\mathcal{O}(n \log n)$, donde n es el tamaño del diccionario. El algoritmo no ordena de forma estable, pero en este caso no nos importa dado que no diferenciamos entre dos enteros iguales. Notemos también que se realiza una copia del diccionario para no modificarlo directamente, pero su coste $\Theta(n)$ se ve enmascarado por el coste de ordenar.
- **Búsqueda**: La parte clave del algoritmo, realiza una búsqueda por la tabla a modo dividir y vencer, descartando la mitad del vector en cada paso. Hemos implementado la versión iterativa para reducir el overhead de las llamadas recursivas. El coste asimptótico temporal en caso peor de esta parte del algoritmo es $\mathcal{O}(m \log n)$, dado que se realizan m búsquedas de coste $\mathcal{O}(\log n)$ n cada una, donde n es el tamaño del diccionario y m es el tamaño del texto.

Con esto concluimos, sumando el coste de cada una de las partes, que el coste total del algoritmo es $\mathcal{O}(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}(m \log n)$ dado que $n = \Theta(m)$.

Cabe recalcar que el coste no se expresa en función del número de operaciones o tiempo empleado, sino en función del número de comparaciones. El estándar de `C++11` no especifica nada respecto este aspecto, y solo restringe el coste en número de comparaciones.

2.2. Tabla de hash con encadenamiento

En este caso, el algoritmo consiste en una tabla de hash con resolución de colisiones mediante encadenamiento. El conjunto de elementos se insertan uno a uno en la tabla mediante el cálculo de la `función de hash`, que nos da la posición a la que tiene que ir el elemento. La lista encadenada que habrá en cada posición tendrá tantos elementos como llaves del diccionario hasheen en esa posición.

Para la función de hash, hemos optado por la más sencilla y eficaz para un conjunto de enteros distribuido uniformemente en el universo de valores como es nuestro caso: la función identidad. Esta función de hash es perfecta, ya que todo elemento diferente del conjunto del diccionario va a tener un hash distinto. Finalmente sólo hay que aplicar la operación de módulo al hash para adaptarlo al espacio de posiciones de la tabla de hash. No nos importa si el tamaño de la tabla es primo o no, ya que el conjunto de entrada son enteros aleatorios y por ende ya están bien distribuidos.

El tamaño de la tabla de hash lo hemos escogido de forma que podamos asegurar que no van a ser necesarias operaciones de rehash. Hemos decidido que $2n$, donde n es el tamaño del diccionario era un tamaño adecuado, ni demasiado grande ni demasiado pequeño.

- **Inserción**: En este caso, la inserción tiene un coste temporal en caso medio $\mathcal{O}(1)$, suponiendo que la función de hash es buena y el nombre de elementos que hashean en la misma posición va a ser asimptóticamente de $\mathcal{O}(n/(2n)) = \mathcal{O}(1)$ elementos. En caso peor, si la función de hash es mala o la carga es alta (no aplica en este caso) tendremos un coste $\mathcal{O}(n)$ en caso peor, donde n es el tamaño del diccionario. Como se insertan n elementos el coste temporal total de esta parte es $\mathcal{O}(n)$ en mediana, y $\mathcal{O}(n^2)$ en caso peor.
- **Búsqueda**: La búsqueda, del mismo modo que la inserción, va a tener un coste constante en mediana, y lineal en caso peor, razonando del mismo modo. A diferencia de la inserción, vamos a realizar la búsqueda para m elementos, con lo que el coste temporal será $\mathcal{O}(m)$ en mediana, donde m es el tamaño del texto, y $\mathcal{O}(n * m)$ en caso peor.

Sumando los costes, tenemos $\mathcal{O}(n + m) = \mathcal{O}(n)$ es el coste temporal en caso medio del algoritmo, dado que m = $\Theta(n)$, y $\mathcal{O}(n^2)$ en caso peor. El coste espacial del algoritmo es proporcional al tamaño de la tabla, que tiene tamaño $2n$, con lo cual el coste espacial es $\Theta(n)$.

2.3. Tabla de hash tipo Hopscotch

Este algoritmo es muy similar al de hashing con lista, pero el método de resolución de colisiones es distinto. La resolución de colisiones es con direccionamiento abierto.

Concretamente, cada posición de la tabla tiene asociado un vecindario de elementos cercanos, de tamaño H, donde H es normalmente el número de bits de la arquitectura, comunmente 32 o 64. Las inserciones se realizan del siguiente modo:

Primero se calcula la posición p del elemento en la tabla con la función de hash, tal y como se haría en el algoritmo de tabla de hash con lista. Si p está libre simplemente se coloca el elemento en p. En caso de que p ya esté ocupada, se busca una posición p' libre entre las del vecindario de tamaño H de p, es decir, las siguientes $H - 1$ posiciones $p + 1, p + 2, \ldots, p + H - 1$, y se coloca el elemento en p'. Si no hay posiciones libres en el vecindario, se procede a rehashear la tabla y se vuelve a intentar insertar el elemento.

El caso es que esta búsqueda es rápida, porque normalmente las posiciones cercanas se encuentran en la

misma línea de caché que la original, y la probabilidad de tener que rehashear es extremadamente baja siempre que la carga de la tabla se mantenga normal/medianamente alta.

Para una búsqueda, el procedimiento es similar. Se busca el elemento en el vecindario de la posición que le tocaría, hasta encontrarlo. Si no se encuentra significa que el elemento no está en la tabla.

Una forma de implementar eficientemente el algoritmo para no tener que visitar todo el vecindario es mantener en cada posición p un mapa M de H bits (se puede guardar en un tipo *long* por ejemplo en `C++`) donde el $i - esimo$ bit de menor a mayor peso de M indica si hay algún elemento que hashee en p que haya ido a parar a la posición $p + i$. La particularidad de esta implementación es que la mayoría de computadores ofrecen instrucciones muy eficientes para poder saltar los bits que estén a 0 o a 1 del mapa, mirando sólo aquellos que sean estrictamente necesarios.

En este caso, el coste de inserción es el mismo, dado que pueden haber rehashes durante las inserciones que podrían degradar el coste a $\mathcal{O}(n^2)$ en total, donde n es el tamaño del diccionario. Para las búsquedas, en cambio, el coste es constante y es $\mathcal{O}(H)$, dado que sólo visitamos las posiciones del vecindario.

Cabe recabar que en nuestra implementación y testeo hemos usado un tamaño $H = 64$, lo que nos da una probabilidad de rehash prácticamente nula.

2.4. Filtro de Bloom

Nuestra implementacion del filtro de Bloom consta de un vector de booleanos de tamaño m y un vector que contiene k semillas aleatorias para funciones de hash. Nuestro algoritmo se divide en 2 fases:

- **Inserción**: Primero codificamos con las k funciones de hash los enteros del diccionario. Cada funcion de hash nos devolvera un numero modulo m, la codificacion consiste en poner a `true` esas posiciones del vector de booleanos. Esto tiene un coste de: $\mathcal{O}(nk)$ donde n es el tamaño del diccionario y k el numero de funciones de hash.
- **Busqueda**: Empezamos simulando la misma codificación que para la inserción y comprobamos si las posiciones del vector son ciertas. En caso que todas las posiciones codificadas sean ciertas diremos que el entero estaba en el diccionario. Esto representa un coste de : $\mathcal{O}(zk)$ donde z es el tamaño de nuestro texto y k, una vez más, el numero de funciones de hash

El problema de este algoritmo es que, dada la codificación es posible encontrarnos con **falsos positivos**. Para reducir la probabilidad de que esto pase podemos aumentar el tamaño del vector de booleanos o codificar los enteros con más funciones de hash perjudicando la eficiencia del algoritmo. En nuestras pruebas hemos fijado una probabilidad del 10^-6. Para las funciones de hash hemos adaptado la conocida como `murmurhash` [4] pasando como parametro la semilla de la funcion. El coste total es: $\mathcal{O}(nk + zk)$ donde, como hemos apuntado anteriormente, n corresponde al tamaño del diccionario, z al tamaño del texto y k al numero de funciones de hash. Como el tamaño del texto esta expresado como una proporcion lineal con el tamaño del diccionario, podriamos decir que el coste total seria, asintóticamente : $\mathcal{O}(nk)$ siendo k constante dado que es proporcional a m/n, por tanto: $\mathcal{O}(n)$

2.5. Búsqueda en árbol Trie

Este algoritmo se basa en guardar los carácteres o dígitos de las llaves formando un árbol de búsqueda tipo radix. Cada nodo del árbol representa un dígito o carácter de una llave perteneciente al conjunto, y tiene tantos hijos como elementos tenga el alfabeto del conjunto, en este caso 10, los 10 posibles dígitos.

De esta forma. para buscar un elemento en el árbol, tenemos que dividirlo en trozos (dígitos en este caso) y irnos guiando por las ramas del árbol en función del dígito actual, accediendo al hijo que corresponda, y igual para la inserción. Podemos deducir entonces, que tanto el coste de inserción como el coste de búsqueda es constante $\Theta(1)$, ya que sólo depende del tamaño en dígitos del número de la entrada, que siempre va a ser menor que $2^(32-1)-1$. Deducimos también que el coste total de inserción y búsqueda es $\mathcal{O}(n+m)=\mathcal{O}(n)$, donde n es el tamaño del diccionario y m es el tamaño del texto.

Para implementar este algoritmo de la forma más eficiente posible hemos escogido que el dígito esté en una base potencia de 2, en este caso base 16, porque es la que nos ha dado mejores resultados. El hecho de que la base sea potencia de dos nos permite dividir el número rápidamente mediante las operaciones and bit a bit (para realizar el módulo y cojer el último dígito) y división por shifteo de bits.

3. Documentación de los experimentos

3.1. Algoritmo de generación

Nuestro algoritmo de generación del diccionario y el texto recibe 3 `parámetros`: tamaño del diccionario, proporción de palabras del diccionario en el texto y tamaño del texto en función del tamaño del diccionario. Con estos argumentos, primero crea un diccionario del tamaño indicado con enteros aleatorios positivos y lo guarda en `./archivos/arxiu1.txt`. A continuación, crea archivos de texto con enteros positivos aleatorios con la proporción deseada de elementos del diccionario y lo guarda en `./archivos/arxiu2.txt`.

En el diccionario puede haber enteros repetidos, no obstante, esto no afecta a los resultados de los algoritmos ya que no tienen ningun mecanismo para saber si ya han buscado la palabra y , en el caso de encontrarse una palabra repetida, la tratan como una más. Hemos realizado `7 pruebas`. Para realizarlas de forma representativa hemos hecho un *script* que ejecuta los programas con distintos parametros y obtiene los resultados. Hemos adjuntado el script con nombre `generador-datos.py`

3.2. Ejecución de los algoritmos

Para dar resultados precisos sobre los algoritmos, las ejecuciones para calcular las estadisticas son distintas a las ejecuciones para calcular el tiempo. Hemos usado un sistema que utilitza directivas de compilador para poder escoger sencillamente si queremos que calcule las estadisticas o no. Todos los algoritmos se dividen en dos fases:Inserción y busqueda. `Nota:` El tiempo calculado es el tiempo a partir que se llame el algoritmo, sin contar la lectura de los datos. A continuación listamos las pruebas realizadas y entre claudators los valores que varian sobre las variables de las pruebas, en caso de no variar, los valores fijos són: tamaño del diccionario: 5e6, proporcion de palabras del diccionario en el texto: 0.5, relación diccionario/texto: 2.5

- **Prueba 1: Tiempo en función del tamaño del diccionario[2e6, 4e6, 6e6, 8e6, 10e6, 12e6, 14e6]**
- **Prueba 2: Tiempo en función de la proporción de palabras del diccionario en el texto[0.1, 0.2, 0.4, 0.6, 0.8, 0.9, 1]**
- **Prueba 3: Tiempo en funcion del tamaño de la relación diccionario/texto[2, 2.5, 3, 3.5, 4, 4.5, 5]**

- **Prueba 4: Tiempo medio de Inserción**
- **Prueba 5: Tiempo medio de Búsqueda**
- **Prueba 6: Uso de Memoria**
- **Prueba 7: Comparativa de iteraciones y otros datos**

4. Análisis de resultados

4.1. Tiempo de ejecución vs Tamaño del diccionario

En este gráfico hemos analizado el tiempo de ejecución de cada algoritmo en función del tamaño del diccionario de entrada.

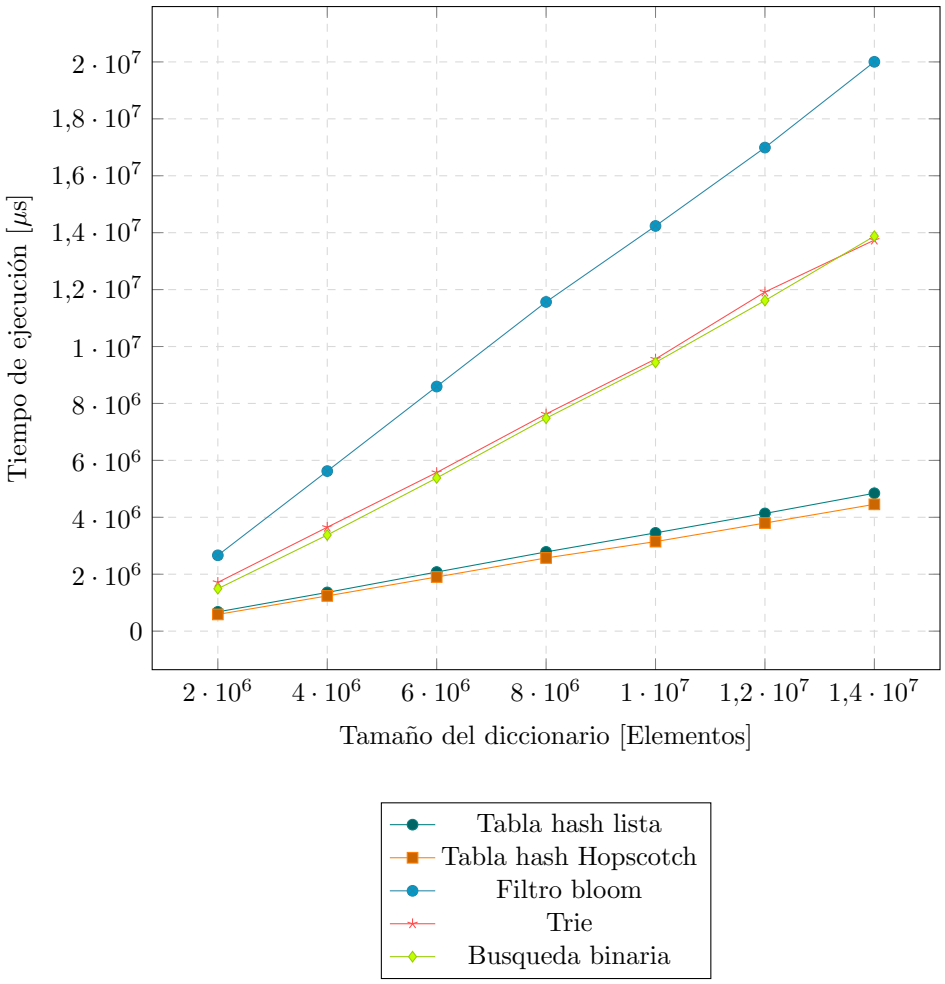


Figura 1: Gráfico del tiempo de ejecución en función del tamaño del diccionario

Tal y como podemos ver en la gráfica, se pueden identificar 3 grupos claramente difereciables por las

pendientes de las rectas y su valor inicial, aunque todas los algoritmos presentan un comportamiento claramente lineal en este rango de valores estudiados.

En el grupo más rápido podemos encontrar los dos algoritmos de hash con tabla, con una pendiente mucho más reducida que el resto de algoritmos, y con unos valores muy similares entre sí, si bien el Hopscotch parece ofrecer un mejor rendimiento cuando el tamaño del diccionario augmenta significativamente.

En el grupo intermedio encontramos el algoritmo de búsqueda binaria y el algoritmo de búsqueda en trie. Claramente su pendiente es mucho mayor a la de los dos algoritmos de hash con tabla, y nuevamente es muy similar entre ellos. Nos ha sorprendido el hecho que el algoritmo de búsqueda en trie (que tiene coste $\mathcal{O}(n)$ en total) tarde más tiempo que un algoritmo como la búsqueda binaria (el cual es $\mathcal{O}(n \log n)$).

Finalmente, en el último grupo encontramos el algoritmo de filtro de Bloom, que tiene un comportamiento también lineal pero mucho peor dadas las pequeñas constantes escondidas en los costes de la inserción y sobretodo en la búsqueda, ya que se tienen que calcular repetidas veces las funciones de hash sólo para una búsqueda.

4.2. Tiempo de ejecución vs Proporción de elementos del diccionario

En este tipo de gráfico, podemos detectar qué algoritmos se ven perjudicados o beneficiados cuando la proporción búsquedas satisfactorias/búsquedas fallidas varía. Cuando la proporción es alta augmentamos la proporción de búsquedas satisfactorias, mientras que cuando ocurre lo contrario incrementamos la proporción de búsquedas fallidas.

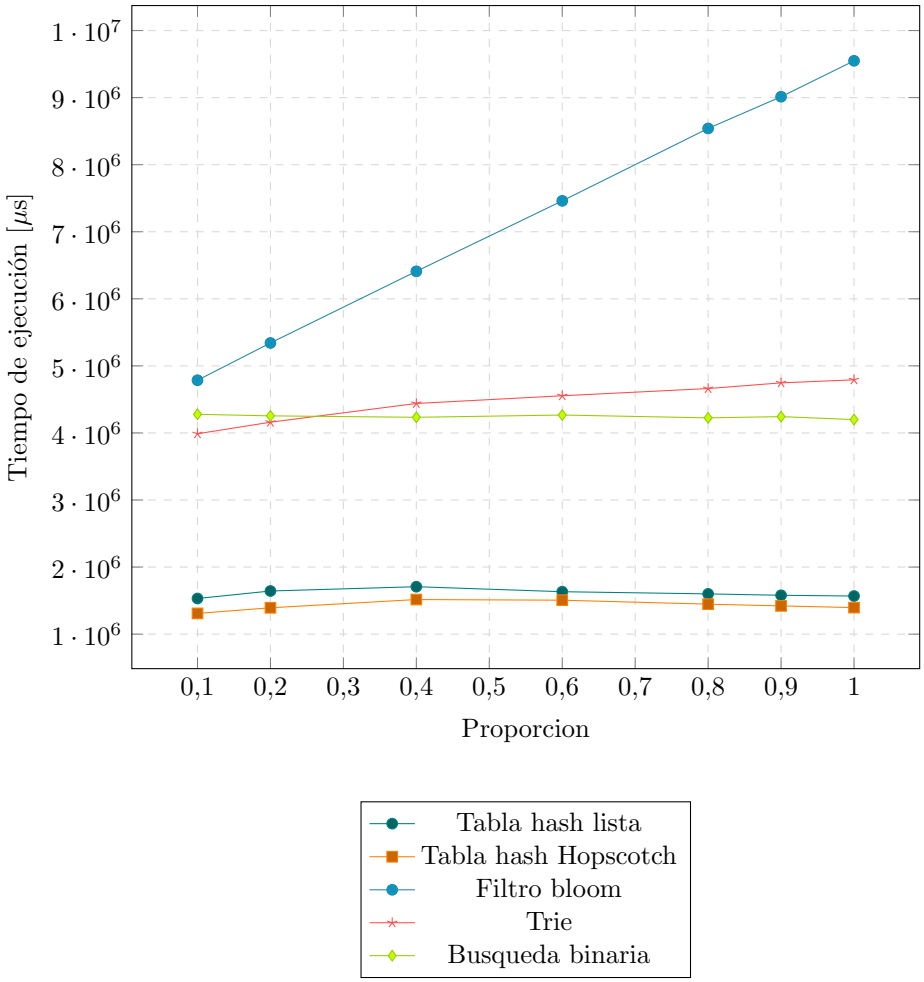


Figura 2: Gráfico del tiempo de ejecución en función de la proporción de elementos del texto que también pertenecen al diccionario

Por lo que podemos ver en el gráfico, la mayoría de algoritmos no se ven afectados por esta proporción, ya que el tiempo de ejecución se mantiene más o menos constante. Hay dos excepciones, el algoritmo de búsqueda en árbol trie y en mayor medida el algoritmo de filtro de Bloom.

Los dos algoritmos se ven perjudicados cuando la proporción augmenta, es decir, cuando hay más búsquedas satisfactorias que búsquedas fallidas. La razón la podemos encontrar en el hecho de que los dos algoritmos tienen un coste de búsqueda relativamente elevado, y, en el caso de Bloom, el algoritmo se ve forzado a calcular más funciones de hash al realizar una búsqueda satisfactoria, al contrario que cuando el elemento no está, dado que entonces sólo calcula funciones hasta que encuentra un bit a 0. En el caso

de Trie occurre algo similar.

4.3. Gráfico del tiempo de ejecución vs tamaño del texto en función del tamaño del diccionario

Este grafico es muy parecido al de la sección `4.1` sólo que en esta ocasión variamos el tamaño del texto, y no el del diccionario.

Lo que varía entonces en este gráfico es la cantidad de búsquedas comparado con la cantidad de inserciones, augmentando cuando el tamaño del texto augmenta y disminuyendo cuando el tamaño del texto disminuye.

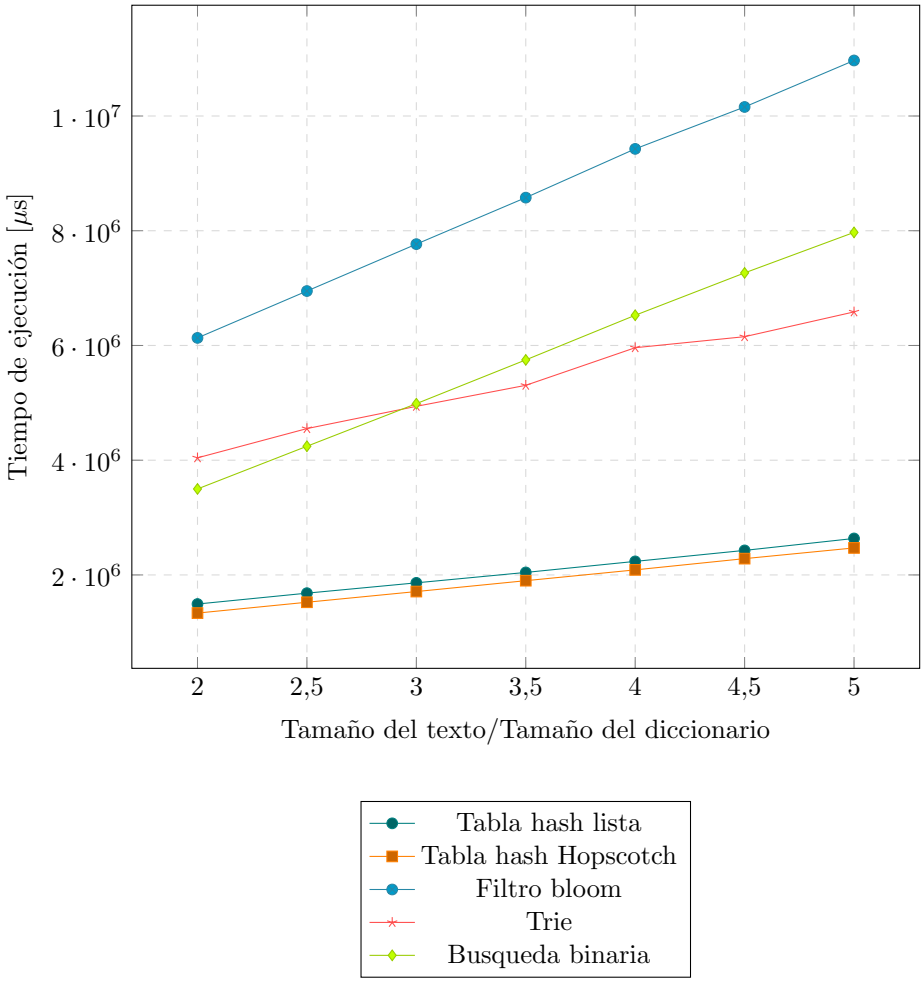
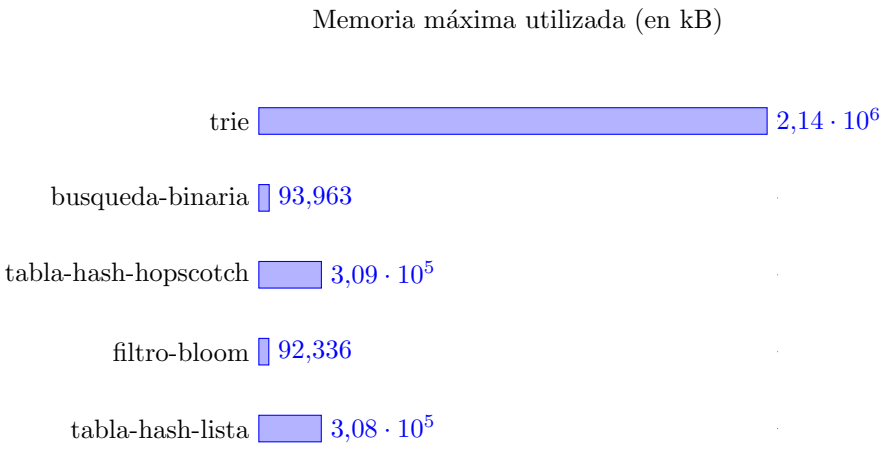


Figura 3: Gráfico del tiempo de ejecución en función del tamaño del diccionario

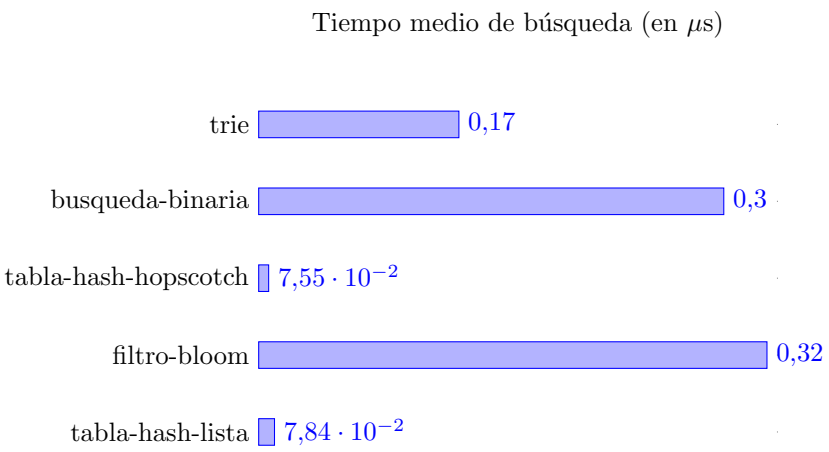
Tal y como vemos en los gráficos, ahora las pendientes no son tan marcadas, como antes, y el algoritmo de trie se ve claramente beneficiado por el cambio de tamaño de texto. Esto es debido a que su coste de inserción es mucho mayor al coste de búsqueda, dado que hay que reservar memoria para los nuevos punteros del árbol.

4.4. Memoria máxima utilizada



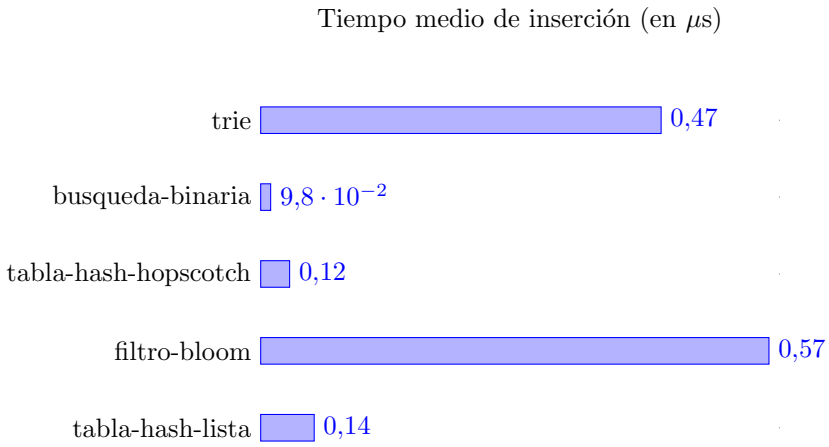
Destacar que el filtro de Bloom usa muy poca memoria, al contrario de lo que al principio pueda hacer pensar. Aunque necesitemos crear un vector de m bits al ser este un vector de booleanos, conseguimos un bajo coste de memoria.Trie tiene un coste en memoria tan elevado por la utilización de punteros(10 por nodo) y porque cada dígito del numero se almacena como un nodo del arbol.

4.5. Tiempo medio de búsqueda



En este gráfico observamos que los algoritmos basados en tablas de hash tienen un comportamiento notablemente mejor y concuerda con las gráficas de los apartados anteriores donde són las más rápidas.

4.6. Tiempo medio de inserción



En la inserción tenemos al filtro de Bloom como más lento, una vez más, dado que tiene que codificar las entradas en el vector de bits calculando las funciones de hash. La búsqueda binaria gana en este caso a los algoritmos basados en hash porque sólo tiene que realizar una ordenación, que, si bien tiene un coste asimptótico más elevado $\mathcal{O}(nlogn)$ vs $\mathcal{O}(n)$, la constante escondida es mucho menor en el caso de la ordenación.

También corroboramos lo que hemos visto en el gráfico 4.3, que el algoritmo de trie tiene un coste de inserción muy elevado en relación al tiempo de búsqueda (4 veces mayor).

4.7. Otros datos importantes

En esta sección discutiremos los contadores que reflejan la cantidad de trabajo hecha por cada algorismo

Filtro de Bloom Para corroborar que el filtro de bloom falla más cuanta menos proporción de palabras del diccionario haya en el texto `m`, hemos calculado el numero de hashes que hace de media por busqueda fallida. La hemos calculado dividiendo el numero de busquedas fallidas por `m`. El resultado ha sido: 1.929 hashes por busqueda fallida. De forma análoga hemos calculado la media de hashes por busqueda exitosa el qual ha sido 19.09 que concuerda con el numero total de hashes(19).

Hash lista vs hash hopscotch Estos dos algorismo son iguales en cuanto a carga de trabajo realizada. No obstante vemos una clara mejora en el algorismo de hopscotch. La razón reside en la chaché

Hash hopscotch y sus 0 rehashes Otra de las principales razones por las que hopscotch tiene mejor rendimiento que hash lista es porque en la version que hemos programado el algorismo hace 0 rehashes dado que la probabilidad que en arquitecturas de 64 bits la probabilidad es bajísima

5. Conclusiones

`Hopscotch` ha demostrado ser el algorismo más eficiente de los que hemos probado, seguido muy de cerca por el `hash lista`. No obstante no significa que este sea la mejor opción. Si tenemos un entorno donde la memoria es muy reducida, podriamos decantarnos por opciones menos eficientes pero que la memória utilizada sea más baja como podrían ser: `Búsqueda binaria` y `Filtro de bloom`.

Referencias

[1] Wikipedia: Hopscotch Hashing *[en línia]*. [Última modificación 11 Diciembre 2015]. [Consulta 10 de Enero de 2015]. Disponible en:
https://en.wikipedia.org/wiki/Hopscotch_hashing

[2] Jakubiuk V., Popovic S., Implementation and Performance Analysis of Hash Functions and Collision Resolutions *[en línea]* Disponible en:
http://www.mit.edu/~victorj/hash_tables_cache_performance.pdf

[3] C++ Reference: Sort *[en línea]* Disponible en:
http://en.cppreference.com/w/cpp/algorithm/sort

[4] Wikipedia: MurmurHash *[en línea]* [Última modificación 17 Diciembre 2015] [Consulta 23 de Diciembre de 2015] Disponible en:
https://en.wikipedia.org/wiki/MurmurHash

[5] luison9999, TopCoder: Using Tries *[en línea]* [Consulta 23 de Diciembre de 2015] Disponible en:
https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/