PROYECTO ALGORISMIA

Octavi Pascual
Patricia Sampedro Garcia

1.¿CÓMO EJECUTARLO? 2

2. EXPLICACIÓN DEL GENERADOR 3

3.EXPLICACIÓN DE ALGORITMOS 4

3.1. Búsqueda dicotómica: 4

3.2. Búsqueda total: 4

3.3. Hash Table: 4

3.5. Bloom Filter: 5

4. EXPLICACIÓN DE LAS GRÁFICAS “Pdfenfunciondelamidadiccionario.pdf” 6

4.1. Gráfico1: Mida diccionari vs tiempo de creación 6

4.2. Gráfico2: Mida diccionari vs tiempo hit 7

4.3. Gráfico3: Mida diccionario vs tiempo miss 8

4.4. Gráfico4: Mida diccionario vs tiempo consulta 8

4.5. Gráfico5: Mida diccionario vs numero comparaciones hit 8

4.6. Gráfico6: Mida diccionario vs comparaciones miss 9

4.7. Gráfico7: Mida diccionario vs comparaciones consulta 9

4.8. Gráfico8: Tiempo que tarda en comparar 9

4.9. Gráfico9: Mida diccionario vs tiempo de hash 9

4.10. Gráfico10:Tiempo función de hash de Bloomer Filter 10

4.11. Gráfico11:Aciertos/Fallos del Bloom Filter 10

1.¿CÓMO EJECUTARLO?

Antes de empezar es necesario hacer make en la carpeta principal donde hay el archivo metodes.cc. Para ejecutar cada algoritmo primero se debe entrar en la carpeta correspondiente del archivo que se quiere ejecutar y seguir las instrucciones que pondremos a continuación para cada algoritmo:

- CercaTotal y CercaDicotomica:
    1. Primero se debe hacer make
    2. Después de debe ejecutar el archivo con los siguientes parámetros:
        a. Nombre del archivo donde hay el diccionario
        b. Nombre del archivo donde hay el texto
        c. 1 si es quicksort y 2 si es radixsort

    Ambos se pueden ejecutar de forma interactiva (se explica por pantalla qué se tiene que poner) ejecutando ./cercaDicotomica.exe o ./cercaTotal.exe
    También existe la posibilidad de ejecutarlos con los parámetros:
    ./cercaDicotomica.exe NomFitxerDiccionari NomFitxerText SortingAlgorithm
    ./cercaTotal.exe NomFitxerDiccionari NomFitxerText SortingAlgorithm

También existe la posibilidad de volver a generar las pruebas que se han realizado. Para ello se dispone de un script, ./prueba.sh. Solo hay que asegurarse que en la carpeta del Generador se han creado todos los ficheros de prueba.

- HashTable y BloomFilter

Tanto en HashTable como en BloomFilter hay dos ficheros. Uno que se llama nombrealgoritmosensem y otro que que es solo el nombre del algoritmo. El makefile esta hecho para el del nombrealgorimtosensem, pero canviarlo es trivial, solo se tiene de cambiar el nombre del archivo en el makefile.

Para el archivo nombrealgoritmosensem

1. Primero se debe hacer make
2. Después de debe ejecutar el archivo con los siguientes parámetros:
    a. Nombre del archivo donde hay el diccionario
    b. Nombre del archivo donde hay el texto
    c. tamaño del diccionario
    d. factor por el que multiplicamos la mida del diccionario para obtener el texto
    e. proporción de palabras del texto que estan en el diccionario

Para el archivo nombrealgoritmo:

1. Primero se debe hacer make
2. Despues de debe ejecutar el archivo con los siguientes parámetros:
    a. Nombre del archivo donde hay el diccionario
    b. Nombre del archivo donde hay el texto

Es importante destacar que los tres últimos parámetros de cada ejecución de cada algoritmo, solo se han puesto para la parte estadística.

Para tal de que podéis comprobar los datos, hemos introducido también los scripts que generan los .txt, que hemos usado, así como los archivos en python para la generación de las gráficas. Los scripts los podéis encontrar en la propia carpeta del algoritmo y los archivos en python en la carpeta en python, así como la recopilación de todos los csv

2. EXPLICACIÓN DEL GENERADOR

A continuación explicaremos como se ha creado el generador y su funcionamiento.
En primer lugar hemos considerado que íbamos a trabajar con números enteros naturales tanto en el diccionario como en el texto. También hemos decidido trabajar con enteros de 32 bits, por lo que todos los números que aparecerán el los archivos generados estarán dentro del rango $[0, 2^{31}-1]$. No sería complicado generar números de 64 bits o números arbitrariamente grandes, pero hemos considerado que este rango era suficiente para realizar nuestro estudio.

Una vez decidido el formato de los números, el siguiente paso ha sido generar el diccionario y el texto. En el diccionario nos hemos asegurado de que todos los números sean diferentes y en el texto puede haber números que no pertenezcan al diccionario.

Para realizar el generador hemos pensado que cuanto más se pudiera parametrizar mejor. Los valores que se pueden parametrizar son los siguientes:

- n (entero): tamaño del diccionario (número de elementos que tendrá)
- v (real): factor por el que multiplicaremos n para obtener el tamaño del texto (el tamaño del texto será n*v)
- p (real): proporción esperada de números del texto que pertenecen al diccionario
- file1 (string): nombre del diccionario (sin ninguna extensión, se añade al generar el archivo)
- file2 (string): nombre del texto (sin ninguna extensión, se añade al generar el archivo)
- seed (entero): semilla que se utilizará para generar los números aleatorios. Es útil si queremos volver a generar los mismos números

3.EXPLICACIÓN DE ALGORITMOS

3.1. Búsqueda dicotómica:

La idea de este algoritmo es ordenar todos los números del diccionario. De esta forma, cuando buscamos si una palabra está o no en el diccionario podemos utilizar el algoritmo de búsqueda dicotómica. Este algoritmo nos indica si un número cualquiera está o no en el diccionario y la posición en que se encuentra dicho número. Nosotros hemos adaptado el algoritmo para que nos retorne si se ha encontrado el número buscado y cuantas comparaciones de claves se han realizado en el proceso. También medimos el tiempo que dura una búsqueda. Para que sea el menor posible, hemos implementado la búsqueda dicotómica de forma iterativa en vez de recursiva, para asi evitar el overhead de las llamadas recursivas.

Para ordenar los números también necesitamos un algoritmo. Hemos implementado dos: quicksort y radix sort. El primero es un algoritmo universal basado en comparaciones mientras que el segundo es un algoritmo que saca partido de ciertas propiedades de los elementos a ordenar. Hemos pensado que podría ser interesante comparar el tiempo que se tarda en ordenar una serie de números con ambos algoritmos.

3.2. Búsqueda total:

Este algoritmo ha sido pensado por nosotros y el nombre que le hemos puesto intenta reflejar la idea de este. Es parecido al algoritmo anterior, búsqueda dicotómica, pero aquí no buscamos los elementos uno a uno sino que realizamos una búsqueda total: buscamos todos los elementos del texto en el diccionario.
Vamos a explicar el proceso que se sigue para que quede clara la idea: ordenamos el diccionario pero también el texto. Ordenar el texto puede ser muy costoso ya que puede ser varias veces más grande que el diccionario, pero si se quieren buscar muchos elementos del texto en el diccionario puede ser útil. En efecto, podemos recorrer el texto y el diccionario a la vez. Tenemos un apuntador para el texto y otro para el diccionario y en función de los números que apuntan incrementamos uno u otro. Este proceso es lineal respecto al tamaño del mínimo entre el tamaño del diccionario y del texto.
Hemos pensado que este algoritmo podría ser interesante, especialmente compararlo con el de búsqueda lineal, así que lo hemos implementado.

3.3. Hash Table:

La idea de este algoritmo es tener una función de hash que relacione todos los números con una entrada de nuestra tabla.

Hemos querido comprobar cómo variaba el tiempo en función de la medida de la tabla. Para hacerlo para cada fichero que nos entraba mediamos su tiempo de creación,tiempo de búsqueda si acierta, tiempo de búsqueda si falla, comparación de llaves si acierta y comparación de llaves si falla en función de M. Las gráficas están en el pdf "PlotsHashTable.pdf". Queremos destacar que aunque en algunas gráficas parece que hay muchas diferencias, por ejemplo en el tiempo de creación de la HashTable cuando la mida del diccionario es 100, eso es porque estamos tratando con números pequeños y cualquier variación en el ordenador hace que parezca mucho.Como podemos observar en todas las mesuras que hemos hecho tanto el tiempo como el número de comparaciones es inversamente proporcional a la mida de la tabla.

Para la comparación que haremos con los otros algoritmos, hemos hecho que la tabla sea lo más grande posible y le hemos sumado un número primero, concretamente el 1049, ya que buscar el número primero que era más adecuado era muy costoso a nivel de tiempo, y como ya vimos que de esta forma no había muchas colisiones decidimos hacerlo así..Como en este algoritmo el tiempo principal, no es solo las comparaciones que hace sino cuánto tarda la función de hash, hemos calculado cuánto se tarda en función de la entrada en hacer una función de hash. En principio esperábamos que el tiempo en todas las medidas fuera constante, pero cuando la mida del diccionario es muy pequeño, no hemos sabido porque el tiempo es un muy grande. Nuestra explicación es que algo interno del ordenador, pero lo hemos repetido varias veces y siempre ocurre.

3.5. Bloom Filter:

La idea de este algoritmo es tener 3 funciones de hash, que relacionen los números con una entrada de nuestra tabla de booleanos. De tal forma que si una de las entradas de nuestro numero convertido con la función de hash es falso, eso significa que el numero no esta en el diccionario. Si es cierto, eso significa que hay muchas probabilidades que sea cierto.

De forma análoga a la tabla de hash,hemos querido observar cómo variaba el tiempo de creación, el tiempo de búsqueda, los fallos y los aciertos en función de la M. Las gráficas de este análisis las podéis encontrar en el pdf "PlotsBloomFilter.pdf". Lo hemos hecho con dos archivos, cuando la mida del diccionario es 100 y cuando es 1000, en ambos casos podéis ver que el tiempo de creación y de búsqueda es muy bajo, excepto un pico puntual cuando la M es 200. No hemos sabido averiguar el motivo, y siempre nos sale en el mismo punto aunque lo repitamos varias veces. El numero de acierto es proporcional a la M, mientras que el número de fallos es inversamente proporcional a ella.

Para la comparación que haremos con los otros algoritmos, hemos hecho que la tabla sea lo más grande posible y le hemos sumado un número primero, concretamente el 1049, de la

misma forma que HashTable.Como en este algoritmo el tiempo principal, no es solo las comparaciones que hace sino cuanto tarda las funciones de hash, hemos calculado cuánto se tarda en función de la entrada en hacer cada función de hash. En principio esperábamos que el tiempo en todas las medidas fuera constante, pero cuando la mida del diccionario es muy pequeño,ocurre lo mismo que en la HashTable y es que es muy grande y las tres funciones de hash tardan lo mismo. Ocurre lo mismo con el tiempo de comparaciones. Hemos supuesto que también era algo propio del ordenador.

Para la comparación de aciertos y fallos podemos ver que los aciertos son mucho más grandes que los fallos, en media, por lo que suponemos que las tres funciones de hash funcionan bien.

Queremos destacar que todas las pruebas se han hecho con máquina virtual, por lo que los tiempos pueden ser peores que sino fueran en maquina virtual.

4. EXPLICACIÓN DE LAS GRÁFICAS “Pdfenfunciondelamidadiccionario.pdf”

La principal idea de este trabajo era estudiar lo efectivas que eran las técnicas de hashing que hemos implementado. Para ello hemos realizado experimentos y hemos medido algunos parámetros. Hemos pensado que se podría medir el número de comparaciones que se realizan en cada búsqueda. Cuantas menos comparaciones se necesiten para encontrar un elemento mejor. También hemos medido el tiempo necesario para realizar una consulta. En teoría cuantas menos comparaciones más rápida será una consulta pero podría haber otros factores que influyan en el tiempo así que hemos considerado adecuado medir esta variable.
Finalmente, es muy importante medir el coste que significa crear las diversas estructuras que se utilizan el los algoritmos. Por ejemplo, para la búsqueda dicotómica se tiene que ordenar el diccionario o en hashing calcular el número de hash de cada número de diccionario e insertarlo donde corresponda. Hemos querido separar los costes de creación de los costes de consulta.

Ahora vamos a analizar las gráficas que hemos obtenido de las diversos experimentos que se han realizado.

4.1. Gráfico1: Mida diccionari vs tiempo de creación

En primer lugar, hemos querido ver como evoluciona el tiempo de creación de la estructura en función del tamaño del diccionario. Antes de analizar los resultados podemos intentar calcular el coste teórico de los algoritmos de ordenación.

El algoritmo quicksort tiene coste:

$$O(n \cdot \log(n))$$

El algoritmo radix sort tiene coste:

$$O((n+b) \cdot \log_b(k)) = O((n+10) \cdot \log_{10}(2^{31}-1))= O(n \cdot 9.33) = O(n)$$

donde b es la base en la que trabajamos y k es el valor máximo que se puede obtener, $2^{31}-1$ en nuestro caso.

Para los algoritmos de hashing es más complicado dar un coste ya que depende del tamaño de la tabla y de la bondad de la función de hashing. Recordando que el valor que hemos escogido para el tamaño de la tabla (M = n + 1049, siendo M el tamaño del la tabla y n el tamaño del diccionario) y considerando que la función de hash es adecuada, insertar un elemento tiene coste constante. Por lo tanto, el coste es

$$O(n)$$

Lo primero que se observa en los resultados es que aparentemente quicksort ordena los elementos más rápidamente que radix sort. Esto no concuerda con los costes teóricos que hemos expresado anteriormente. Hemos intentado explicar el motivo de este hecho y puede ser debido a que la constante de radix sort es muy alta. Es decir, nuestra n no es suficientemente grande. También puede ser debido a la implementación del algoritmo radix sort: hemos usado dos vectores auxiliares y esto puede provocar muchos fallos en la cache cuando se realiza la ordenación. En cambio en quicksort siempre trabajamos con un único vector. Como el objetivo de este estudio no era sobre este aspecto, tampoco hemos querido ir más allá con este tema y simplemente hemos explicado superficialmente este resultado.

Volviendo al análisis general de la gráfica, observamos que el tiempo de creación es proporcional al tamaño del diccionario. Cuanto más grande es el diccionario más se tarda en crear la estructura de datos deseada. El algoritmo de búsqueda dicotómica solo ordena el diccionario mientras que el de búsqueda total ordena texto y diccionario, por lo que siempre es más lento que el primero. Esta observación se ve reflejada en la gráfica.

4.2. Gráfico2: Mida diccionari vs tiempo hit

Hemos estudiado el tiempo que se tarda en encontrar un elemento que está en el diccionario. En la gráfica que se ha obtenido observamos que el tiempo se mantiene constante respecto al tamaño del diccionario. Se podría extender el experimento con tamaños mayores para ver si realmente es constante o aumenta de forma muy lenta.

Intuitivamente esta hipótesis es la más razonable ya que, por ejemplo, el coste de encontrar un elemento en una tabla ordenada es O(log(n)) por lo que al menos en este sí existe una relación entre n y el tiempo de búsqueda.
Es interesante ver que en las funciones de hash se produce algún pico. Seguramente es debido a que para aquellos valores en concreto la función de hash no ha funcionado correctamente y se han producido muchas colisiones. Esto implica que al buscar un elemento hemos tenido que recorrer toda la lista en la que podría estar, y en dicha lista se habían producido las colisiones.

4.3. Gráfico3: Mida diccionario vs tiempo miss

Hemos estudiado el tiempo que se tarda en buscar un elemento que no está en el diccionario. En este caso observamos que los resultados son prácticamente los mismos para cada algoritmo.

4.4. Gráfico4: Mida diccionario vs tiempo consulta

En este experimento hemos analizado el tiempo medio que se tarda en realizar una consulta. Por lo tanto no hemos distinguido entre acierto o fallo como sí lo habíamos hecho en los dos experimentos anteriores. Aquí vemos que el algoritmo de hashing vuelve a ser un poco más lento que los otros, lo que tiene sentido ya que como hemos visto anteriormente era más lento que los otros en encontrar un elemento que sí está en el diccionario.

4.5. Gráfico5: Mida diccionario vs numero comparaciones hit

En este experimento hemos estudiado el número medio de comparaciones de clave que se realizan cuando se encuentra el elemento que buscamos en el diccionario.
La primera observación es que hay una superposición de algunas curvas. CercaTotal quicksort y CercaTotal radix realizan exactamente el mismo número de comparaciones ya que se usa exactamente el mismo algoritmo para buscar elementos. Recordemos que la única diferencia entre estos dos algoritmos estaba en el paso anterior, el de la ordenación de los elementos. De forma análoga esto ocurre en CercaDicotómica quicksort i CercaDicotómica radix, que también están superpuestas. Es interesante observar que en este caso CercaTotal se comporta mejor que CercaDicotómica. Esto es así porque en la última buscamos los elementos uno a uno mientras que en la total lo hacemos de forma más inteligente. El número de comparaciones del algoritmo de hash es ligeramente menor al de CercaTotal. Esto se debe a que el número de comparaciones de hash es aproximadamente igual a la mitad de su factor de carga.

4.6. Gráfico6: Mida diccionario vs comparaciones miss

En este experimento hemos estudiado el número medio de comparaciones de clave que se realizan cuando no se encuentra el elemento que buscamos en el diccionario.
Volvemos a observar que se produce el fenómeno de la superposición, como en el experimento anterior. Observamos que tanto CercaTotal como hashing aumentan muy ligeramente respecto al tamaño del diccionario mientras que CercaDicotómica lo hace de forma mucho más clara. Buscar un elemento que no está en una tabla ordenada tiene coste O(log(n)) mientras que hacerlo en una tabla de hash tiene coste su factor de carga.

4.7. Gráfico7: Mida diccionario vs comparaciones consulta

En este experimento hemos estudiado el número medio de comparaciones de clave que se realizan independientemente si se encuentra el elemento que buscamos o no.

De forma analoga al Gráfico6 y al Gráfico5, hay una superposición de algunas curvas. CercaTotal quicksort y CercaTotal radix realizan exactamente el mismo número de comparaciones e igual sucede con CercaDicotómica quicksort y CercaDicotómica radix. Observamos que los algoritmos CercaTotal i hash table son muy similares mientras que CercaDicotómica es claramente más lento que los otros dos. Este resultado era esperado visto los resultados de las dos gráficas anteriores.

4.8. Gráfico8: Tiempo que tarda en comparar

Quiero destacar que esta grafica igual que la 9,10 y 11, las hemos comentado durante el documento, pero ahora las explicaremos más detalladamente.

En este experimento hemos estudiado el tiempo que tarda el algoritmo de HashTable y el de bloom en comparar. Teniendo en cuento que lo estamos haciendo en la misma maquina virtual el resultado deberia ser el mismo, aunque no es asi, ya que en la HashTable comparamos el valor de un puntero con un número, es decir, primero debemso obtener el valor del puntero y despues compararlo, mientras que en el bloomFilter comparamos dos números. En este caso el tiempo medio de una comparación es 0.15 nanosegundos

4.9. Gráfico9: Mida diccionario vs tiempo de hash

Como ya hemos dicho antes, esta gráfica ya la hemos comentado durante el documento, pero ahora las explicaremos más detalladamente.

En bloom tenemos tres funciones de hash pero en este caso hemos optado por utilizar la función que hace el modulo, asi es la misma función que en la HashTable. Así pues, como podemos observar tiene un tiempo quasi constante y las dos funciones funcionan de igual. En este caso es alrededor de 0.25 nanosegundos, por tanto, podemos afirmar que es más costos comparar que ejecutar una función de hash basada en módulos.

4.10. Gráfico10:Tiempo función de hash de Bloomer Filter

Como ya hemos dicho antes, esta gráfica ya la hemos comentado durante el documento, pero ahora las explicaremos más detalladamente.

En la gráficas anteriores hemos visto que las funciones de hash basada en módulos son más baratas que las comparaciones. Ahora queremos ver si nuestras otras dos funciones basadas en el método de la multiplicación, tiene tiempos parecidos que las de basadas en módulos y en consecuencia funciona mejor que las comparaciones.

Así pues,tal y como observamos en el gráfico, las tres líneas están sobrepuestas por lo cual podemos deducir que tienen los mismos tiempos.

4.11. Gráfico11:Aciertos/Fallos del Bloom Filter

Como ya hemos dicho antes, esta gráfica ya la hemos comentado durante el documento, pero ahora las explicaremos más detalladamente.

En esta gráfica queremos ver si nuestro bloom filter funciona adecuadamente. Tal y como se observa en la gráfica el número de aciertos siempre es mayor que el número de fallos, lo que a primera instancia nos dice que como mínimo no és pésimo.

Nuestra tabla siempre tiene el mismo tamaño que n más 1049, pero que en n grandes es un factor constante. Así pues como n/m = 1, la probabilidad que dos números caigan en la misma posición es de 1-(1/(e)^3)^3 = 0.8 si el elemento está. Así pues, como en los aciertos también hemos contado cuando un elemento no está (que eso siempre es acierto), podemos observar que funciona dentro de lo esperado