Super Sopa

Grau A Q1 Curs 2022-2023

Miembros del grupo

Aranda Sánchez, Hugo

Caballero Castro, Joan

Miró López-Feliu, Oriol

Ruiz Vives, Albert

Introducción

En este proyecto se nos presenta un problema: la búsqueda de palabras en una sopa de letras estrambótica. Esta sopa de letras, al igual que cualquier otra, contiene palabras que deben ser buscadas para solucionar el problema. La diferencia principal con otras sopas es que las palabras contenidas en ella no se ven limitadas por las 8 direcciones principales de la cuadrícula, sino que pueden cambiar de sentido y zigzaguear. Por lo tanto, la búsqueda tendrá que tener en cuenta estas condiciones para poder encontrar todas las palabras de la sopa.

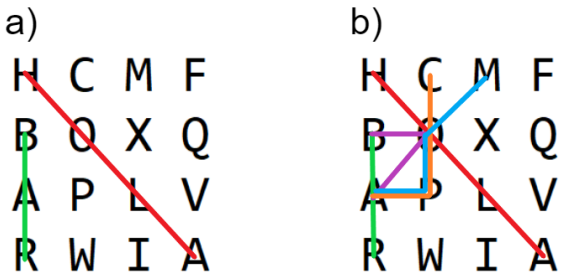

Figura 1: Diferencias entre una sopa de letras (a) y una sopa estrambótica (b)
La sopa (a) contiene solo las palabras: {hola, bar}
La sopa (b) contiene: {hola, bar, boa, copa, mopa}

Gracias a la figura 1, podemos ver que las sopas estrambóticas permiten encontrar muchas más palabras gracias a que las restricciones que presentan son más laxas y permisivas. Esto aumenta en cierta medida la complejidad del problema y de los algoritmos a emplear.

En este proyecto tendremos un diccionario *D* compuesto por *m* palabras. Estas serán las palabras que podremos encontrar en las sopas que usaremos. Del diccionario usaremos un subconjunto fijo *P* de palabras con $|P| = 20$. Este conjunto hará de auxiliar para la creación de sopas con el fin de garantizar que existen un mínimo de palabras en las sopas analizadas. Finalmente, usaremos 4 estructuras de datos distintas para guardar la información del diccionario *D* y poder compararlas entre ellas.

Objetivos

Nuestro objetivo será analizar y comparar la eficiencia de las siguientes 4 estructuras de datos: Vector ordenado, Trie, Filtro de Bloom y Tabla de Double Hashing.

Primeramente, queremos analizar de forma teórica el coste asintótico de las operaciones básicas de estas estructuras. Posteriormente, compararemos de forma experimental cómo se comportan en tiempo de ejecución, midiendo los tiempos de sus consultoras, así como el número de palabras que encuentran en la sopa (esto último es importante para garantizar que las estructuras se han implementado correctamente, así como para medir los falsos positivos introducidos por el Filtro de Bloom). Finalmente, mediremos la precisión del Filtro de Bloom, ya que este puede dar falsos positivos y no garantiza un resultado 100% correcto.

Para cumplir estos objetivos, hemos creado diversos algoritmos que se detallarán en los siguientes apartados:

- Un algoritmo creador de sopas estrambóticas, el cual es capaz de insertar las palabras de *P* en una sopa de tamaño *n * n* de forma totalmente aleatoria y no determinista.
- Un algoritmo buscador de palabras de acuerdo con las restricciones que estas presentan dentro de la sopa.
- Las 4 estructuras de datos mencionadas anteriormente para la comprobación de la existencia de las palabras encontradas en *D*.

Algoritmos

Algoritmo creador de sopas

Idea

La idea de nuestro algoritmo creador de sopas es hacer uso de funciones de tipo RNG (Random Number Generator) como rand() para crear un algoritmo no determinista. Primeramente, hacemos uso del RNG para colocar las 20 palabras del subconjunto *P* en la sopa. El RNG se usa tanto para decidir la posición inicial de forma pseudoaleatoria como para decidir las posiciones que van tomando los carácteres consecutivos de cada palabra. Finalmente, usamos el RNG para llenar el resto de la sopa con carácteres aleatorios con rango a-z.

Pseudocódigo

Sean una sopa vacía *S* y un subconjunto de palabras a insertar *P*:

1. Barajar aleatoriamente las palabras de *P* para que su orden original no influencie a la sopa
2. Para cada palabra *p* de *P*:
    a. Buscar una posición inicial válida (posición vacía o con la misma letra que la inicial de *p*) hasta encontrarla o llegar a un número de intentos predeterminado
    b. Colocar los demás carácteres de forma recursiva comprobando las posiciones directamente adyacentes (considerando por válidas aquellas que estén vacías o que tengan el mismo carácter que el que se quiere colocar) y que no hayan sido ocupadas ya por la palabra (para evitar duplicaciones). Para cada rama de recursión, en caso que se llegue a colocar la palabra entera se finaliza el algoritmo y corta el proceso de recursión. De lo contrario hace un proceso de backtracking en el árbol de recursión y sigue en la siguiente rama. Este proceso se repite hasta encontrar una rama que permita colocar la palabra o se llegue a un número de intentos predeterminado.
3. Si se han podido colocar todas las palabras correctamente, hacer un último recorrido por la sopa llenando todas las posiciones vacías de *S* con carácteres aleatorios. De lo contrario, se retorna una sopa vacía con un mensaje de error.

Correctitud

Al ser un algoritmo no determinista, no se puede garantizar la correctitud de éste. De hecho, se contempla que haya situaciones donde por combinatoria o por falta de espacio no se pueda completar la sopa. En cuyo caso se garantiza que se mandará un mensaje de error apropiado. Aún así, en caso de fallo, el algoritmo no garantiza que haya alguna posibilidad de obtener una sopa correctamente generada por más intentos que se hagan. De todos modos, se puede demostrar que si no hay mensaje de error la sopa ha sido generada correctamente:

Para demostrar la correctitud de las sopas, supondremos que hemos generado una sopa con una palabra mal insertada y veremos si existe alguna posibilidad de no haber recibido mensaje de error. Puesto que para que se mande el mensaje solo se necesita que una palabra haya estado insertada incorrectamente, estudiaremos sólo el caso individual.

Una palabra sólo ha estado mal insertada si su posición inicial era inválida o si una vez escogida su posición inicial no se ha podido insertar toda la palabra:

- Si la posición inicial era inválida, se seguirá intentando buscar una posición inicial de forma aleatoria hasta encontrar una, de lo contrario, se llegará al número máximo de intentos. Hecho que disparará el mensaje de error al no haber podido colocar la palabra.
- Una vez encontrada la posición inicial, se aplica un algoritmo similar a un DFS para hacer una búsqueda extensiva y probar todas las posibles combinaciones de inserción a fuerza bruta. Mientras no se encuentre una rama válida en el árbol de recursión, se sigue buscando. Una vez recorridas todas las combinaciones posibles, si no se ha encontrado una rama de recursión válida, sabemos con certeza que la palabra no se puede colocar y se dispara el mensaje de error automáticamente.

Por lo tanto, vemos que sólo se considerará una sopa como válida si todos los carácteres de todas las palabras en ella han sido colocados en posiciones válidas. Ya que hemos considerado por válidas las posiciones vacías o con carácteres iguales a los que se quieren colocar, podemos garantizar que ,siempre que no salte un mensaje de error, habremos obtenido una sopa válida.

Cabe destacar que lo contrario no es cierto. No siempre que salte un mensaje de error implica que no se puede encontrar una combinación de posiciones válida para *S* y un determinado conjunto *P*. Por lo tanto, solo podremos garantizar que hemos llegado a una respuesta final cuando generemos una sopa correctamente.

Algoritmo buscador de palabras

Idea

Nuestro algoritmo buscador de palabras implementa la estrategia de recorrido de un DFS con backtracking al que se le aplican podas en función del prefijo de las palabras. Es decir, va recorriendo toda la sopa en profundidad y para cada nueva posición, comprueba si la palabra que ha encontrado hasta entonces es prefijo de alguna palabra del diccionario *D.* En caso que no lo sea, poda esa rama de recursión puesto que ya no va a encontrar palabras por ahí.

Pseudocódigo

Sean una sopa *S* y un diccionario *D:*

1. Crear un set de soluciones *R*. Éste set nos permitirá almacenar los resultados encontrados en la sopa, ignorando repeticiones, de forma eficiente.
2. Para toda posición de la sopa:

a. Crear un string/array vacío donde iremos almacenando la palabra encontrada hasta el momento.
b. Si la palabra encontrada hasta el momento existe en *D*, insertarla en *R* para marcar que hemos encontrado una palabra válida.
c. Para cada una de las 8 posiciones adyacentes a la casilla actual (4 casillas directamente adyacentes + 4 diagonales):
    i. Si la posición es válida, no ha sido visitada y la palabra encontrada hasta el momento es prefijo de alguna palabra de *D*, marcar la nueva posición como visitada y volver recursivamente al paso 2.b
    ii. Si la posición no es válida, ha sido visitada o la palabra encontrada hasta el momento no es prefijo de alguna palabra de *D,* retorna sin proseguir la búsqueda.

Correctitud del algoritmo

Sean una sopa *S*, un diccionario *D*, un set de soluciones *R* y una palabra auxiliar *s*:

Invariante: Para cualquier paso *i*, *s[0] hasta s[i-1]* es un prefijo de alguna de las palabras contenidas en *D* y estamos en una posición válida y previamente no visitada de *S*.

Comprobación precondiciones: La función “padre” del algoritmo de recursión siempre hace la llamada con una de las posiciones de la sopa y un string con el carácter contenido en esa posición (s.length() == 1). Puesto que al empezar nos encontramos al paso 1, se verifica que s[0] hasta s[1-1=0] es prefijo de alguna palabra de D, pues la palabra vacía es prefijo de todas las palabras. Además, como llamamos posiciones de la sopa *x* e *y* tales que *x, y* >= 0 && *x, y* < n, sabemos que son posiciones válidas de la sopa. Finalmente, como iniciamos la llamada con el vector de visitados marcado como vacío, no podemos estar en una posición previamente visitada. Se cumplen todas las condiciones del invariante.

Comprobación recursión: Antes de cada llamada recursiva, se hacen varias comprobaciones: Primero, se mira que las nuevas posiciones *x* e *y* permanezcan dentro de los límites de *S*, además se comprueba que la nueva posición no haya sido visitada previamente. Finalmente, antes de hacer la llamada recursiva se comprueba que la palabra *s* obtenida hasta el momento (paso i-1 en la recursión) sea prefijo de alguna palabra de *D.* Por lo tanto, se cumplen todas las condiciones del invariante.

Comprobación finalidad: Puesto que a cada llamada recursiva incrementamos en 1 el número de posiciones visitadas y no se puede proseguir la búsqueda en una posición previamente visitada, a cada paso recursivo estamos eliminando una posición válida de la sopa. Por lo tanto, eventualmente la sopa se quedará sin posiciones válidas y nuestro algoritmo terminará de forma garantizada.

Comprobación resultados: Para cada nodo de la sopa donde empezamos el algoritmo, al estar usando un DFS como base, obtenemos el árbol de todas las palabras que se pueden formar a partir del nodo inicial escogido, sin importar su posición o dirección. Como vamos comprobando a cada nuevo carácter si la palabra se encuentra en *D* y aplicamos el algoritmo a todas las posibles posiciones iniciales, a efectos prácticos estamos aplicando un algoritmo de backtracking que comprueba todas las combinaciones posibles aplicando

ciertas podas. Por lo tanto, podemos asegurar que se encuentran todas las posibles palabras en *S*.

Anotación: Cabe destacar que, en nuestra implementación, el vector de visitados no ha sido implementado con booleanos sinó con enteros sin signo a modo de contador para evitar copias (ahorrar tiempo a cambio de memoria). Entonces, si estamos haciendo el DFS desde el nodo número *x*, marcaremos por visitados todas aquellas posiciones con valor == *x* y por no visitados aquellas con valor <= *x,* operando siempre sobre el mismo vector por referencia. Ya que el valor máximo de un *uint* es 4294967295, para una sopa de tamaño n*n >= 4294967295 nuestro algoritmo dejaría de funcionar.

Estructuras de datos

Vector ordenado

Introducción

Para esta implementación utilizaremos un vector de strings con todas las *n* palabras del diccionario *D* ordenadas crecientemente en orden alfabético.

Funcionamiento

El funcionamiento interno de un vector es punteros a memoria contigua

Implementación y coste

Constructora

Para la creación de esta estructura vamos añadiendo todas las palabras del diccionario *D* al vector, lo cual tiene un coste lineal al número de palabras añadidas, en este caso $\Theta(n)$.
Por otra parte, cuando hemos añadido todas las palabras al vector, debemos ordenarlas alfabéticamente utilizando algún algoritmo de ordenación eficiente, como por ejemplo el Quick Sort en tiempo $\mathrm{O}(n\ \log(n))$.
Por tanto, el coste total de estas dos operaciones resulta en $\mathrm{O}(n\ \log(n))$.

Algoritmo de búsqueda

A la hora de buscar cierto prefijo *p* dentro del vector hacemos una búsqueda dicotómica en la cual buscamos la primera palabra que no tenga un valor alfabéticamente menor al prefijo. Esta búsqueda tiene coste logarítmico al número de palabras del vector, $\mathrm{O}(\log(n))$.
Una vez encontrada esta palabra *s*, queremos comprobar si *p* es realmente su prefijo. Para eso comprobamos que todos los caracteres de *p* sean los mismos y estén en el mismo orden que en *s*. El coste en caso peor de esta comprobación es $\mathrm{O}(|p|)$, puesto que tenemos que recorrer todos sus caracteres.
Por tanto, el coste total de una búsqueda por prefijo *p* es $\mathrm{O}(\log(n) + |\mathrm{p}|)$.

Corrección del algoritmo de búsqueda

Queremos verificar que el algoritmo es correcto, es decir, que cuando buscamos una palabra de *D* que tenga como prefijo *p* la encuentre y que cuando la buscamos con un string que no sea prefijo de ninguna palabra, el algoritmo no la encuentre.
En primer paso comprobamos la búsqueda dicotómica: al encontrar una palabra *s* en la posición *i* del vector, como el vector está ordenado crecientemente por orden alfabético, sabemos que todas las palabras anteriores a *s* son menores lexicográficamente que *p* y, por tanto, *p* no puede ser prefijo de ellas. Podemos argumentar también que cualquier palabra siguiente a *s* puede tener como prefijo a *p* si y solo si *p* es prefijo de *s*, en caso contrario todas las palabras siguientes a *s* no tendrán a *p* como prefijo porque serán superiores lexicográficamente a *p*.

En segundo paso verificamos la comprobación del prefijo: dado un prefijo *p* de tamaño |*p*| y una palabra *s* de tamaño |*s*| donde se cumple que |*p*| <= |*s*|, vamos comprobando por toda posición *i* en el intervalo [0, |*p*|) que *p*[*i*] == *s*[*i*]. Este algoritmo es capaz de verificar que *p* es

prefijo de *s*, puesto que si todos los caracteres de *p* coinciden ordenadamente con los caracteres de *s* se considera que *p* es prefijo de *s*.
También detecta el caso donde *p* no es prefijo de *s*, puesto que si por alguna *i* donde *p*[i] != *s*[i], eso implica que *p* no es prefijo de *s*.

Aplicaciones

El vector ordenado es una estructura de datos muy utilizada al ser muy eficiente en espacio y tiempo, además, es una estructura muy generalizable. Cosa que permite que su número de aplicaciones sea muy grande.

El tamaño que ocupa en memoria es $\Theta(n)$, donde *n* es el número de elementos del vector.
Los elementos dentro de esta estructura se encuentran mediante una búsqueda binaria en tiempo $O(\log(n))$, y para su creación se necesita tiempo $\Theta(n)$ para añadir todos sus elementos y utilizando un algoritmo de ordenación eficiente, como por ejemplo el Quick Sort o el Merge Sort, se pueden ordenar sus elementos en tiempo $O(n\ \log(n))$, permitiendo además otros algoritmos de ordenación como el Counting Sort o el Radix Sort.

El vector funciona muy bien gracias a su simplicidad que permite a muchos algoritmos partir del vector como su estructura de datos. Además, su estructura poco restrictiva, permite que éste sea usado para problemas a los que otras estructuras de datos no pueden terminar de acatarse.

Trie

Introducción

Un Trie es una estructura de datos en forma de árbol n-ario de búsqueda. Por lo tanto, los árboles de búsqueda binarios, ternarios y árboles de Patricia quedarían bajo la clasificación de Tries.

La idea básica de cualquier Trie es montar un árbol de forma que cada nodo representa un prefijo de sus hijos. De esta forma se pueden almacenar de forma eficiente palabras (como strings) u otras codificaciones de tipo secuencial (como códigos binarios en el caso de los árboles Patricia).

Características de los Trie:

- Permiten operaciones eficientes por prefijos, a diferencia de otras estructuras de datos.
- Son flexibles, permitiendo añadir y borrar datos guardados.
- Pueden almacenar de forma muy eficiente en memoria y tiempo palabras o secuencias que comparten prefijos.
- Si los datos almacenados son demasiado disímiles entre sí, se pierde gran parte de la eficiencia.

Funcionamiento

Ya que en el proyecto hemos implementado un árbol de búsqueda ternario (o TST para abreviar), vamos a explicar el comportamiento de este. Aunque cabe destacar que su funcionamiento es parecido a otros tipos de trie y, por lo tanto, se podría generalizar la idea básica.

El árbol estará compuesto por distintos nodos. Cada uno con los siguientes atributos:

- El dato a almacenar (por ejemplo, un carácter de la palabra).
- Un puntero a cada uno de sus 3 nodos hijos (denominados *left*, *mid* y *right*).
- Un booleano que indica si el nodo marca el final de una palabra o secuencia.

La forma de guardar palabras dentro del árbol será mediante los nodos centrales o *mid*.
Así pues, solo estaremos adquiriendo información adicional sobre la secuencia cuando avancemos por el hijo central en el recorrido del árbol.

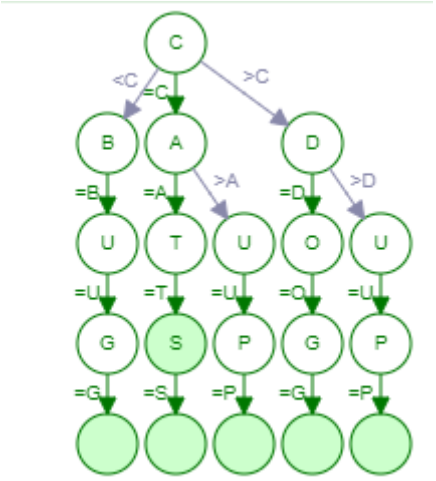

Figura 2: Representación de un árbol de búsqueda ternario

En la figura 2 podemos ver la representación de un TST. Miramos de entender su funcionamiento:

La letra que se observa dentro de cada nodo representa el carácter guardado por este. Los nodos de color verde indican final de palabra, mientras que los de color blanco no lo son (cabe destacar que en este TST se guarda también el carácter \0 que indica final de string). Además, solo cogeremos un carácter como candidato válido de una palabra cada vez que vayamos al hijo central de un nodo (representado como un símbolo “=” verde en su transición). Por ejemplo, si queremos encontrar la palabra *cup* se haría el siguiente recorrido:

1. Partimos de la raíz C, como el primer carácter de la palabra *cup* es igual a C vamos al hijo del medio y nos cogemos C como posible candidato.
2. Comprobamos el valor del siguiente nodo A con la segunda letra de la palabra. Puesto que U > A en orden lexicográfico, nos vamos al hijo de la derecha U. Al no haber escogido el hijo central no consideramos A como candidato.
3. Comprobamos el valor del siguiente nodo, puesto que este también es U nos vamos al hijo central y cogemos U como posible candidato.

4. Hacemos lo mismo con el carácter P y la tercera letra de la palabra, puesto que son iguales, vamos al hijo central y cogemos P como candidato.
5. Ya que hemos llegado al final de nuestra palabra, solo falta comprobar si el nodo actual marca final de palabra. Puesto que está en verde, hemos llegado al final de la palabra en el árbol y podemos concluir que *cup* se encuentra en este.

Como subcaso de recorrido, si en vez de buscar palabras concretas estamos buscando si existe un prefijo determinado, se pueden hacer los pasos del 1 al 4 ignorando el 5, ya que solo nos importa si la estructura de nodos está presente en el árbol, no si el último nodo marca final de palabra.

Una vez visto el funcionamiento del TST se puede observar fácilmente como se encuentran guardadas las palabras {bug, cat, cats, cup, dog, up}. Cabe destacar que aunque *cup* y *up* comparten sufijo, deben posicionarse en partes totalmente distintas del árbol. Esto es debido a que la discrepancia en prefijos fuerza que la U y la P de *cup* se guarden en el subárbol central de C (para poder tomar esta como primera letra de la palabra) mientras que las de *up* se deben guardar en un subárbol totalmente independiente de la letra C para no causar ambigüedad.

Coste

Puesto que el diccionario es fijo para cada juego de pruebas, solo se han implementado las funciones de crear, insertar y buscar. Aun y así, explicaremos también el coste y funcionamiento de la función de borrar un elemento, puesto que es una de las principales funciones de un TST:

Creación:
Crear un TST solamente implica crear un nodo raíz vacío. Por lo tanto, el coste de esta operación es $O(1)$.

Búsqueda:
Cuando buscamos en un árbol ternario podemos diferenciar dos operaciones: Primero, recorremos *k* (donde *k* es el tamaño en caracteres de nuestra palabra o secuencia) veces los hijos centrales de nodos del árbol para formar la palabra en cuestión. Segundo, hacemos búsquedas recursivas en los subárboles izquierdo y derecho de cada nodo (para encontrar el prefijo que necesitamos). Estas búsquedas recursivas suelen ser logarítmicas en función del número de palabras guardadas *n*, pero en caso que el árbol esté muy desequilibrado debido a la degeneración propia de los árboles de búsqueda o a un diccionario de entrada desequilibrado de inicio, el coste se puede volver lineal.

Por lo tanto, el coste de esta operación será $O(\log(n) + k)$ en árboles debidamente equilibrados y $O(n + k)$ en el peor de los casos.

Cabe destacar que es importante tener en cuenta *k* en el coste asintótico, ya que en los casos donde haya pocas palabras (*n* muy pequeña) pero de gran longitud (*k* muy grande) lo que regirá el coste de ejecución será el valor *k* y no el de *n*.

Inserción:
El proceso de inserción es muy parecido al de búsqueda, ya que simplemente se hace un recorrido del árbol como se ha explicado anteriormente y se asume que la palabra ya se encuentra dentro de este. Si todos los nodos ya existen de forma que se pueda hacer la palabra, se marca el último como final de palabra. De lo contrario, se crean nuevos nodos a medida que se vayan necesitando (con coste $O(1)$) hasta formar la estructura de nodos requerida para representar la palabra.

Por lo tanto, el coste será $O(\log(n))$ o $O(n)$ recorridos en subárboles y k creaciones de nodos con coste $O(1)$. Coste final: $O(\log(n) + k)$ en caso medio y $O(n + k)$ en caso peor.

Borrado:
De la misma forma que la inserción, el borrado también depende fuertemente de la operación de búsqueda, pues borrar un elemento del árbol supone buscar si está y su posición y luego procesar cualquiera de estos 4 posibles casos:

1. **La palabra no se encuentra en el árbol:** No se hacen cambios en este. Coste $O(1)$
2. **La palabra se encuentra en el árbol y es prefijo de otras palabras:** Simplemente se pone a falso el atributo que marca final de palabra. Coste $O(1)$
3. **La palabra se encuentra en el árbol y tiene otras palabras como prefijo:** Se eliminan recursivamente los nodos que forman la palabra, empezando por el final hasta llegar a un nodo final del prefijo o agotar el subárbol que la contiene. Coste $O(k)$
4. **La palabra se encuentra en el árbol y no es prefijo ni tiene prefijos:** Se eliminan recursivamente todos sus nodos, empezando por el final hasta agotar el subárbol que la contiene. Coste $O(k)$

Puesto que los casos a procesar tienen coste menor al de búsqueda y se hacen secuencialmente después de ejecutarla, el coste final es el de hacer una búsqueda: $O(\log(n) + k)$ en caso medio y $O(n + k)$ en caso peor.

Aplicaciones

Debido a la estructura de prefijos que presentan los tries, estos son especialmente útiles para ciertas aplicaciones en el mundo de la informática:

Reemplazar Tablas de Hash:
En ciertos casos, los Tries pueden reemplazar tablas de Hash, ya que permiten guardar palabras y comprobar su existencia con solo las primeras letras en vez de la palabra entera, ahorrando así tiempo y espacio.

Correctores ortográficos:
Puesto que las palabras con mismo prefijo se guardan muy cerca las unas de las otras, los tries son muy apropiados para comprobar la existencia de palabras parecidas, pudiendo usarse como correctores ortográficos entre otras cosas.

Buscadores y “Search Engines”:
De la misma forma, al implementar de forma muy eficiente la búsqueda por prefijos, los tries se pueden usar para hacer buscadores y recomendadores de búsqueda, permitiendo encontrar resultados apropiados con solo una pequeña parte de la información.

Filtro de Bloom

Introducción

El Filtro de Bloom es una estructura de datos probabilística, es decir, que da un resultado aproximado, pero no siempre cierto. Fue creado por Burton Howard Bloom en 1970 y se usa para comprobar si un elemento pertenece a un conjunto.

Varias características del Filtro de Bloom:

- Pueden darse falsos positivos (se indica que un elemento está cuando en realidad no está) pero nunca falsos negativos.
- No se pueden borrar elementos.
- Es muy eficiente en espacio y memoria.
- Con un tamaño fijo se pueden representar infinitos elementos.

Funcionamiento

En esta sección explicaremos el funcionamiento básico del Filtro de Bloom, y la parametrización de cada componente se estudiará en el apartado siguiente.

Esta estructura de datos permite solo las operaciones de inserción y comprobación de un elemento, y se realizan de la siguiente manera:

Inicialmente: Se requiere de un array de bits de tamaño m inicializado a cero y k funciones hash deterministas.

Inserción de un elemento: Se aplican las k funciones hash al elemento que se quiere insertar, y por cada resultado obtenido, ponemos el bit correspondiente del array a uno.

Comprobación de un elemento: Análogamente, para comprobar si un elemento está en nuestro conjunto, aplicamos las k funciones hash y comprobamos que, para cada una de ellas, haya un bit a uno en el array.

Ejemplo: En el ejemplo siguiente, tenemos un Filtro de Bloom parametrizado por un array de 18 bits y 3 funciones hash. Podemos ver como se rellenan las diferentes posiciones al insertar los elementos x, y, y z, y las comprobaciones al intentar consultar el elemento w.

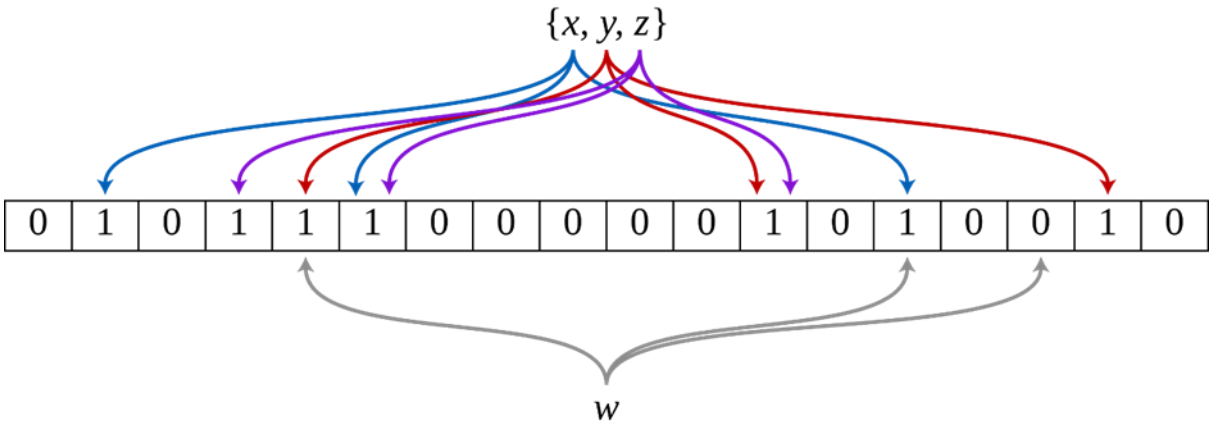

Figura 3: Ejemplo de inserción y consulta de elementos

Falsos positivos, pero no falsos negativos: Debido a la naturaleza del Filtro de Bloom, y como también se ilustra en el ejemplo anterior, se puede dar el caso que la inserción de dos elementos diferentes ponga a 1 la misma posición del array. Entonces, se podría dar el caso de que, al comprobar un elemento, todas las posiciones que lo representan están a uno a causa de otros elementos, por tanto, se devuelve un falso positivo. Pero la inversa nunca será cierta, ya que no se puede dar el caso que un elemento esté en el conjunto, pero no esté representado correcta y determinantemente.

Parametrización y Coste

En este apartado, primero se presentará la parametrización de las variables que actúan en el Filtro de Bloom y luego se justificará el coste en tiempo y memoria.

Al investigar para implementar la estructura de datos, nos dimos cuenta de que existían una serie de fórmulas con las que podíamos escoger los parámetros óptimos según nuestras necesidades.

Sea n el número de elementos del diccionario, X la cantidad de bits a 1, p la probabilidad de falso positivo, m el tamaño del array de bits y k la cantidad de funciones hash, las siguientes fórmulas relacionan los parámetros:

- Número de elementos aproximado $n \approx -\frac{m}{k} ln(1 - \frac{X}{m})$
- Probabilidad de falso positivo $p = (1 - [1 - \frac{1}{m}]^{kn})^k$
- Tamaño óptimo del array de bits $m = -\frac{n*ln(p)}{ln(2)^2}$
- Número óptimo de funciones hash $k = \frac{m}{n} * ln(2)$

Decidimos que lo más lógico era introducir nosotros el número de elementos del diccionario y la probabilidad de falso positivo que queremos arriesgar, por ende nos quedamos con las últimas dos fórmulas con tal de computar la m y la k.

El último punto clave era decidir cómo aplicar los diferentes hashes requeridos. Investigando, encontramos que una buena función hash para un Filtro de Bloom debía cumplir las siguientes características:

- Distribución independiente y uniforme, con tal de llenar balanceadamente el filtro y limitar colisiones
- Rápida, ya que es clave para cualquier operación.

Decidimos usar el hash *murmur* en su versión 3, puesto que era el que mejor reflejaba las características deseadas. El nombre del algoritmo viene de multiplicar (*multiply*) y rotar (*rotate*), dos operaciones básicas que usa. A la hora de aplicar varias veces este algoritmo, en cada iteración se le da una semilla diferente, así obtenemos los k hashes distintos.

En cuanto al coste en tiempo del algoritmo, analizaremos las tres operaciones que podemos realizar, crear, insertar y consultar.

Crear: Consiste en calcular los parámetros óptimos y luego crear un array de tamaño m inicializado todo a cero. Los cálculos tardan un tiempo constante, entonces, el coste lo domina la creación del array, en nuestra implementación un vector de C++. Coste: $\mathrm{O}(m)$

Insertar: Se aplican k funciones hash, cada una de las cuales tiene coste $\mathrm{O}(l)$, donde l es el tamaño de la palabra a la que aplicamos el hash. Por cada función hash ponemos a uno un bit del array, lo cual tiene tiempo constante, por tanto, el coste es $\mathrm{O}(k * l)$

Consultar: Consultar es lo mismo que insertar, excepto que consultamos los bits en vez de ponerlos a uno. Ambas operaciones son constantes, entonces, el coste es el mismo que al insertar, $\mathrm{O}(k * l)$.

Y finalmente el coste en memoria, también vendrá dominado por el array y será $\mathrm{O}(m)$.

Implementación específica

Después de realizar varias pruebas, nos dimos cuenta de que la implementación anterior era inviable con respecto al tiempo para resolver el problema. Esto se debe a que solo podemos comprobar si una palabra pertenece al conjunto al tenerla completa, y exigía una implementación que, desde cada posición de la sopa, se realizará un dfs de profundidad igual a la longitud de la palabra más larga.

Una solución que se nos ocurrió fué adaptar la estructura de datos con tal de poder buscar prefijos, y así podar mucho el espacio de búsqueda a cambio de sacrificar memoria.

Con tal de hacer esto y conservar aún toda la descripción hecha anteriormente, decidimos crear un segundo array de bits, que actúa a forma de Filtro de Bloom de prefijos. En este array auxiliar, insertamos todos los prefijos de todas las palabras, pero no la palabra completa. Es decir, si insertamos en la estructura de datos la palabra “hola”, en el array principal se guardaría igualmente “hola”, y en el auxiliar se guardaría “h”, “ho” y “hol”.

La parametrización para este array auxiliar es la misma que para la principal, aunque de esta solo se mantiene el valor de la p. La nueva n_2 es el número de carácteres totales menos el número de palabras, al no guardar la palabra entera.

A continuación estudiaremos los costes de las operaciones auxiliares. Estas son análogas a las anteriormente nombradas: Creación, Inserción y Consulta. Entiéndase que los parámetros m_2 y k_2 són los relativos a este array y operaciones auxiliares, con lo cual, son generalmente diferentes a los del apartado anterior.

Crear array auxiliar: Mismo coste que en el apartado anterior, $\mathrm{O}(m_2)$

Insertar prefijo: Sea l la longitud de la palabra que queremos insertar, se insertarán $l-1$ palabras (una por prefijo). Cada una de estas inserciones tendrá un coste de $\mathrm{O}(k_2 * l)$, como hemos visto en el apartado anterior, en consecuencia, el coste es
$\mathrm{O}(k_2 * l\ ) * (l-1) = \mathrm{O}\ (k_2 * l^2)$.

Consultar prefijo: La consulta se hace igual que en apartado anterior, pero en el array auxiliar, y tiene el mismo coste, $\mathrm{O}(k_2 * l)$

Finalmente, el coste en memoria será ahora dominado por el tamaño del array auxiliar, por tanto, será $\mathrm{O}(m_2)$

Aplicaciones

La naturaleza rápida y relativamente fiable de los Filtros de Bloom los hace ideales para varias aplicaciones. A continuación, listamos las que nos han parecido más interesantes, y cómo explotan las propiedades de la estructura de datos.

Caché de varios tipos: Cómo sabemos que no existen falsos negativos, podemos usar un Filtro de Bloom como caché. Esto es muy usado, por ejemplo, para comprobar si un nombre de usuario ya existe en un servicio de cuentas al crear una nueva cuenta. Si y solo si el Filtro de Bloom diera positivo, se comprobará de forma más exhaustiva (y cara) si realmente existe el nombre.

Lista de vistos en red social: En algunas redes que disponen de un sistema de contenido infinito, es decir, aquellas en las que se va sirviendo el contenido a medida que se navega, se usa muchas veces un Filtro de Bloom para evitar mostrar el contenido de forma repetida. Ejemplos famosos incluyen Quora, por ejemplo.

Correctores ortográficos: Se puede guardar un diccionario de las palabras correctas en un Filtro de Bloom, y comprobar las palabras escritas sobre este. Actualmente, se usan más otras técnicas, pero así se implementaron antes los correctores ortográficos.

Double Hashing

Introducción

Esta es una estructura de datos típica para los diccionarios y sets por basar la disposición de su información en esta según unas claves.

Idealmente, querríamos tener tantas posiciones en la estructura de datos como posibles claves, esto hace que la búsqueda en esta sea O(1), ya que usamos esta clave como índice directamente, desgraciadamente esto es muy costoso en memoria. Aquí entra el hashing, este consiste en establecer una pequeña traducción en tiempo razonable de esta clave a unos índices numéricos (que tienen una extensión mucho menor que toda la combinatoria de posibles claves) así simulamos tener tantas posiciones como claves aunque no sea así. Esta búsqueda en tiempo peor es O(n) aunque en caso medio sea similar a constante.

Por ende, los puntos fuertes de esta estructura son:

- Buen tiempo medio de todas sus operaciones
- Eficiencia en memoria en relación con su tiempo medio
- No hay probabilidad de fallo ni falsos positivos, etc.

Hay varias formas de implementar estos con distintos tipos de estructuras de datos según nuestros intereses, generalmente han de ser capaces de insertar elementos, borrarlos y consultarlos. En nuestra versión pero, habrá algunos detalles diferentes.

En primer lugar, no habrá función de borrar, pues los diccionarios que nos brindan son los que son y una vez los hemos cargado íntegros en la estructura para resolver la sopa no tenemos intención alguna de sacarlos de ahí o alterar el contenido. Entonces los cargaremos de una sentada en lugar de usar funciones de insertar individualmente o borrar elementos. Habrá una función de hash en la cual creamos la estructura entera desde 0 según un diccionario particular.

Funcionamiento

Como se menciona en la introducción, la base del hashing es la traducción de clave a índice entero mediante funciones de hash, ya que estos elementos son guardados en un array basado en un índice entero.

Tabla de Hash: Es un simple array que guarda las claves y sus informaciones asociadas. Estas están organizadas de una forma que es fácil encontrarlas si fuese necesario.

Función Hash: Es la función encargada de traducir la clave al índice entero. El objetivo de esta es lograr que cada clave reciba un índice único. Cada índice está mapeado a un índice particular, y en algunos casos esta función puede generar el mismo índice para dos claves

diferentes, llamaremos a esto colisión y deberemos evitarlas en la medida de lo posible y tratarlas cuando sucedan.

Una buena función de hash ha de cumplir estas propiedades:

- Ha de ser computable eficientemente.
- Deberá distribuir uniformemente las claves en índices (claves equiprobables).
- Minimizar colisiones
- Debería tener un factor de carga pequeño (relación entre el número de elementos inseridos en el array y el tamaño del mismo)

Tratado de colisiones: Como la función nos da un índice pequeño para una clave grande, cabe la posibilidad de que dos claves terminen en el mismo índice. Entonces, en este caso podemos proceder de dos maneras, o encadenando listas que cuelgan del índice donde hay colisión (Chaining) o calculando posiciones nuevas basándonos en la clave (Open Addressing).

Double hashing es una forma de Open Addressing que hace uso de una segunda función de hash para calcular las nuevas posibles posiciones de forma determinista a partir de la clave original.

Por tanto, la estructura realiza sus acciones de la siguiente manera:

Inicialmente, se genera una tabla de hash vacía la cual es un array en nuestro caso de tamaño 0, pues no podemos crearlo hasta saber el número de prefijos que introduciremos en el.

Hash se hace cuando se quiere introducir un diccionario, creamos el array de tamaño óptimo según la cantidad de los datos a meter y vamos haciendo inserts uno a uno hasta llenarlo a su punto óptimo.

Insertar se hace introduciendo la clave en la función de hash que devuelve un índice correspondiente a la posición en el array, si esta posición no contiene elementos lo introducimos. Si la posición está ocupada, usamos la clave de nuevo en la segunda función de hash para obtener un stride. Con este stride vamos saltando en la tabla hasta encontrar un espacio vacío. En nuestro caso, como guardamos tanto prefijos como palabras, metemos un valor adicional que es true si el string se corresponde a una palabra (puesto que prefijo lo será siempre, al menos de sí mismo).

Consultar sucede de una forma similar a insertar donde calculamos el índice y comprobamos si ese elemento se corresponde al que buscamos. Si no es así de la misma manera calculamos el stride y vamos saltando en la tabla según este hasta o bien encontrar el valor o bien encontrar un valor nulo en la tabla (lo cual significa que este valor no está presente dentro de la tabla).

Parametrización

Una vez entendida la estructura de datos y su funcionamiento, explicaremos la parametrización de los diferentes valores que la determinan.

Estudiando la estructura descubrimos que para ciertos parámetros hay valores que aportan una mayor eficiencia en la estructura de datos.

Primeramente, nos encontramos con el tamaño de la tabla, el cual es vital calcular correctamente para que esta funcione de forma adecuada.

Si es demasiado grande en relación con lo que vamos a almacenar, no estamos aprovechando la función de hash y nos queda una estructura de datos muy dispersa que ocupa más memoria de lo que debería.

Si es demasiado pequeña, por contra tenemos que la tabla tendría muchas colisiones e incluso peor aun podría llenarse por completo y necesitar un rehash (crear de nuevo la tabla de hash e inserir todos los elementos que hay en esta de nuevo teniendo un coste muy grande).

El punto medio en tamaño trata de tener un 75% de coeficiente de carga (elementos insertados/elementos totales) para asegurar que tenemos un poco de espacio para que la dispersión de elementos sea adecuada. De esta forma calculamos el número de elementos necesario haciendo alguna transformación aritmética y obtenemos que el tamaño ha de ser 1/0.75 = 1.333… que aproximamos a 1.3 .

Otro factor importante es que el tamaño del array ha de ser primo. Pues si este no lo fuese cuando hiciéramos el módulo por el tamaño del array con los índices, si un índice tiene algún factor común con este sería introducido en un índice que es múltiplo de ese factor generando muchas agrupaciones. Cosa que va muy en contra con la filosofía de dispersión de la tabla de hash.

Una vez definido el tamaño solo nos queda definir unas buenas funciones de hash que aseguren las condiciones necesarias para que el funcionamiento de la estructura de datos sea correcta.

Estas, en primer lugar, tienen un coste de cómputo razonable, ya que ambas hacen una operación de coste constante x veces donde x es el tamaño del string. Siendo, por tanto, O(x) para las dos.

Suponiendo que las palabras seleccionadas son aleatorias y no tienen ningún patrón en los caracteres que las forman, los procesos lingüísticos que las generan son suficientemente aleatorios, las claves se distribuyen de forma equiprobable, ya que la suma de los valores int en ascii será también aleatoria.

Después, esta minimiza colisiones con una justificación similar a la anterior. Además, evitamos que palíndromos o palabras formadas por las mismas letras reordenadas caigan

en la misma clave mediante la multiplicación de dos elevado a la posición, haciendo así que la posición de la letra altere su peso en el índice. Lo hacemos en potencias de dos, ya que estas son más eficientes de multiplicar a bajo nivel.

Operaciones, Justificación y Costes

Una vez vemos que el tamaño es adecuado y el coste asintótico y corrección de las funciones hash podemos terminar de justificar y calcular los costes de los algoritmos de las funciones a realizar.

Constructora

Consiste en calcular los parámetros óptimos para generar la estructura, en nuestro caso como creamos el array en la función hash solo inicializamos los algunos parámetros a cero en coste constante y pre calculamos un set con los números primos*.

* El tiempo es constante si tenemos en cuenta que pre-calculamos el bitmap mediante el *Sieve of Eratosthenes* con una complejidad temporal de $O(n*log(log(n)))$ donde n es el número de elementos en el bitmap. Esta es una demostración clásica, se puede ver en el siguiente enlace:
https://www.geeksforgeeks.org/how-is-the-time-complexity-of-sieve-of-eratosthenes-is-nloglogn/

Hash

Al haber de introducir todos los prefijos a partir de un solo input de palabras, utilizamos un vector auxiliar en la entrada de la función para poder introducirlo todo en una llamada. También tiene de parámetro de entrada int que contiene el número de prefijos que se pretende insertar en el hash.

Esta función auxiliar hace un rehash creando de nuevo el array según el número de prefijos (como hemos visto antes) y los inserta todos a partir de las palabras del vector. Este hace tantas llamadas a insertar como prefijos hay en él.

Por lo tanto, el coste es la creación del array (en nuestro caso el vector del stl de c++) más las n llamadas a insertar $O(m+n(m+x))$.

Insertar

La función siempre cumple la precondición de que el array no esté lleno, puesto que la función hash es la única que llama a insertar y esta lo hace habiendo calculado un size de array óptimo para insertar todos los elementos sin problemas (no solo sin que esté lleno sino siendo óptimo en complejidad temporal).

Calculamos el probe con la primera función hash. Esto como hemos visto en la explicación y demostración de la función de hash nos da un índice calculado en tiempo lineal en función del tamaño máximo del string ($O(x)$) que es equiprobable entre todos los posibles, asegurando que no se generan clusters.

Después calculamos su stride de una forma similar para tener diferentes strides equiprobables y vamos saltando hasta encontrar un lugar donde insertarlo. Este lugar existe, pues la tabla no está llena y terminamos llegando a él eventualmente, pues, al ser la tabla de tamaño primo a medida que hacemos módulos de este vamos iterando (potencialmente) todos los elementos de la tabla hasta dar con el hueco deseado.

El coste por tanto, es el de las funciones de hash más potencialmente un coste lineal de iterar por todo el array O($m + x$).

Consultar

La justificación y coste es idem al de insertar, ya que usan las mismas funciones de hash, la clave de que consulta sea idéntico a insertar es que el camino que sigue la inserción es totalmente determinista.

Es decir, el índice y stride calculados serán siempre el mismo, y haremos las mismas iteraciones que la inserción hasta dar con él, pues seguimos el mismo algoritmo determinista. Si el elemento ha sido insertado con anterioridad se encontrará y si no se llegara a un elemento vacío que nos dirá que el que buscamos no está en la tabla.

Cabe la posibilidad de que la tabla esté llena y el elemento no se encuentre en ella, para eso miramos también que no hayamos vuelto a la posición inicial (recorrido todo el array sin éxito).

El coste, por tanto, es el mismo que el de la inserción, siendo O($m + x$) donde m es el tamaño del array y x el tamaño máximo de string.

Coste en memoria

Para acabar, el coste en memoria, está determinado por el tamaño de los arrays y será $O(m + y)$ donde m es el tamaño del array de la hashtable e y es el tamaño del array del bitmap.

Una palabra en el coste medio

Si evitamos correctamente la generación de clusters obtenemos algo que asintóticamente se asemeja al uniform hashing el cual está demostrado que tiene un coste medio de O($1/(1-\alpha)$) donde alfa es el factor de carga del array calculado como $\alpha = n/m$ donde n es el número de elementos insertados en el array y m es el tamaño.

Aplicaciones

En general las aplicaciones de esta estructura de datos están muy relacionadas con el hecho de que sus accesos son rápidos, son muy útiles para definir si un elemento está en un conjunto.

Estructuras de datos: estos sirven para implementar diferentes estructuras de datos que representan conjuntos por su facilidad de comprobación.

Bases de datos: ya que nos permite generar índices muy prácticos a la hora de distribuir la información en discos.

Caches: pues nos pueden ayudar a acelerar el proceso en el que la información se guarda por la forma de gestionar las colisiones entre dos entradas diferentes de caché.

Experimentación

Recursos y herramientas usados

Para la experimentación hemos usado las librerías <sys/times.h> y <sys/resource.h> para medir las métricas de tiempo en las búsquedas de sopa.

El *hardware* usado para hacer las medidas experimentales es el siguiente:

- **Procesador:** AMD Ryzen 7 5800H
- **Tarjeta Gráfica:** Radeon Graphics 3.20 GHz (integrada del procesador)
- **Memoria:** 16GB RAM

Con una máquina virtual con las siguientes características:

- **OS:** Ubuntu 22.04.1 LTS de 64 bits
- **Núcleos de Procesador:** 4 núcleos
- **Memoria:** 4096MB RAM

Metodología y Resultados

Estudio de comparación de tiempos entre Estructuras de Datos

En este apartado medimos los tiempos de búsqueda de las distintas estructuras de datos y comparamos entre ellas para ver cuáles brindan mejor rendimiento.

Para hacer este experimento, hemos creado las estructuras de datos con el mismo diccionario *D*, el cual será constante para todas las medidas.

Además hemos creado una sopa de tamaño *n*n* especificado en el input y se le insertan 20 palabras subconjunto de *D.* Éstas palabras también serán constante durante todo el experimento para no introducir variabilidad adicional, pero el contenido de la sopa irá variando a cada medida por tal de estudiar el comportamiento más generalizado de las estructuras de datos a la hora de resolver el problema.

Finalmente, hemos puesto medidores chivatos antes y después de realizar el algoritmo de búsqueda con cada una de las estructuras de datos por tal de medir el tiempo que tardan en encontrar todas las palabras de la sopa.

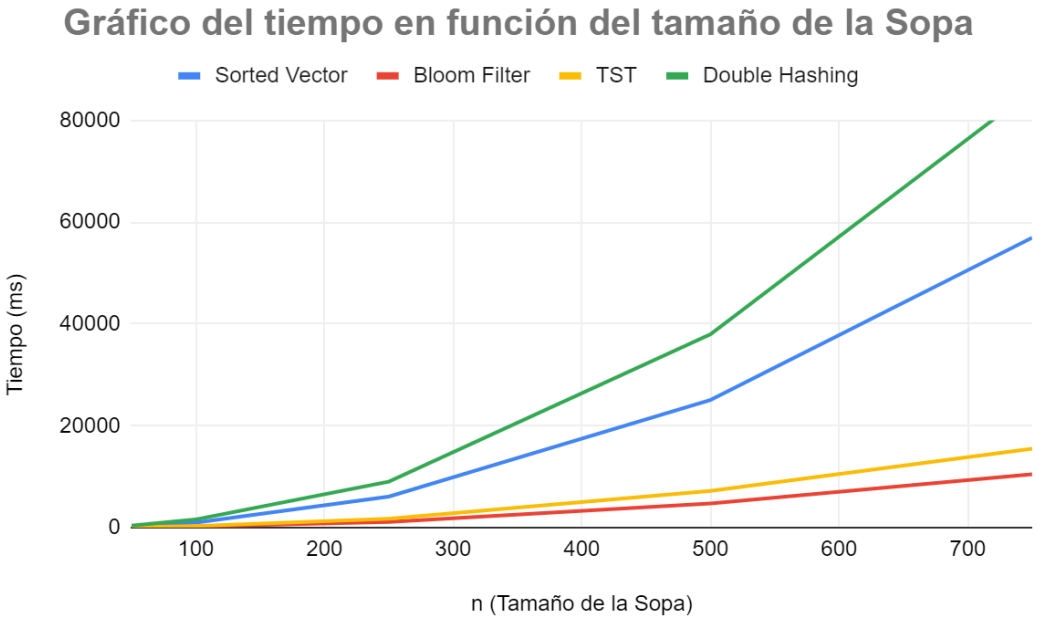

Figura 4: Resultados obtenidos para distintos tamaños de sopa:
Tamaños: {50, 100, 250, 500, 750}

En las figuras 4 y 5 podemos apreciar los resultados obtenidos en el experimento. Se puede observar que el Vector ordenado y la Tabla de Hash presentan crecimientos similares, siendo la tabla de hash la estructura más ineficiente en complejidad de todas.

De la misma forma, el Filtro de Bloom y el Trie también presentan un crecimiento similar, siendo éstos dos estructuras de datos, cuyas consultoras presentan un crecimiento casi negligible respecto al tamaño de la sopa (cabe recordar que en estas medidas no solo observamos el tiempo de consulta, sinó el tiempo de recorrido de la sopa en sí).

Este comportamiento tan eficiente se debe a que, por una parte, el Filtro de Bloom ya está creado con la idea de hacer un número de operaciones constantes para verificar si una palabra está contenida en él y, por otra parte, que el Trie es una estructura de datos específicamente creada para ser muy eficiente en la búsqueda por prefijos (como es el caso en el recorrido de una sopa de letras), pues ésta está integrada directamente en la implementación de la estructura en sí (los nodos ya están dispuestos a modo de prefijo).

n	Sorted Vector	Bloom Filter	TST	Double Hashing
50	244,381	45,458	74,424	405,621
100	960,310	190,420	296,598	1.576,064
250	6.089,870	1.109,214	1.719,596	9.017,092
500	25.061,740	4.724,262	7.204,132	37.977,560
750	56.920,060	10.461,704	15.503,880	85.965,260

Figura 5: Resultados en ms obtenidos en la experimentación

Estudio de precisión del Filtro de Bloom

En este apartado vamos a analizar la precisión de los Filtros de Bloom y cómo afecta esta al coste en tiempo y memoria.

Con este fin, medimos la cantidad de falsos positivos que encontramos y el tiempo que tardamos en hacerlo, variando el tamaño de la sopa y el parámetro p que se le daba al Filtro de Bloom en su creación. Se usan indistintamente los términos de precisión y probabilidad de falso positivo para referirnos a dicha p.

Para medir la cantidad de falsos positivos, simplemente medimos la diferencia entre resultados encontrados por el filtro de bloom y alguna otra estructura de datos, ya que las demás no dan nunca fallos. Para medir el tiempo, se hace como en los experimentos anteriores. Se han tomado estas medidas como la media de cinco iteraciones.

El uso de memoria y la cantidad de hashes usados se miden una vez por precisión, ya que son invariables y constantes para cada precisión independientemente del tamaño de la sopa.

Estos son los resultados para la cantidad de falsos positivos.

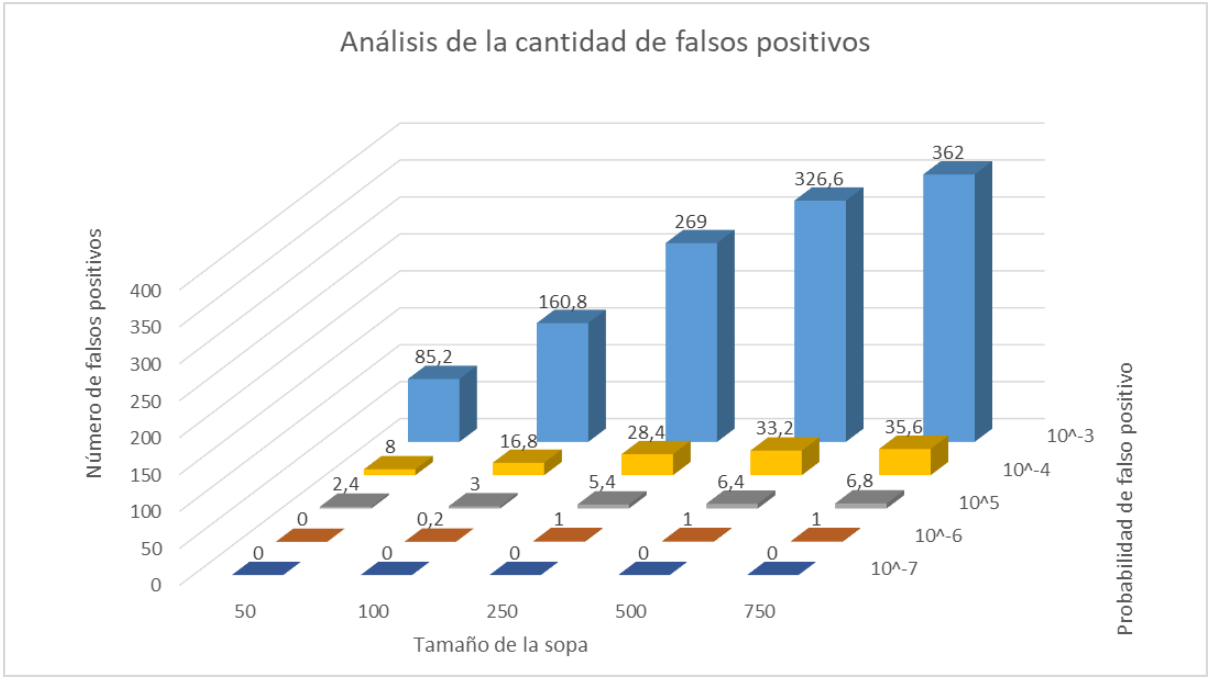

Figura 6: Número de fallos medidos experimentalmente respecto al tamaño de la Sopa y probabilidad teórica

Como podemos observar, la precisión teórica se ajusta a la práctica, ya que cada cambio de precisión se refleja como debería, en una potencia de 10, en la cantidad de falsos positivos encontrados. Por ejemplo, en la sopa de tamaño 750 el aumento de precisión de 10^{-3} a 10^{-4} reduce la cantidad de falsos positivos de 362 a 35,6, aproximadamente una décima parte.

Esta equivalencia entre precisión teórica y práctica se observa mejor en errores mayores, ya que con errores pequeños se necesitan muchas más mediciones para ajustarlo correctamente.

Estos son los resultados para el análisis del tiempo, realizado para las mismas iteraciones que en el estudio de la precisión.

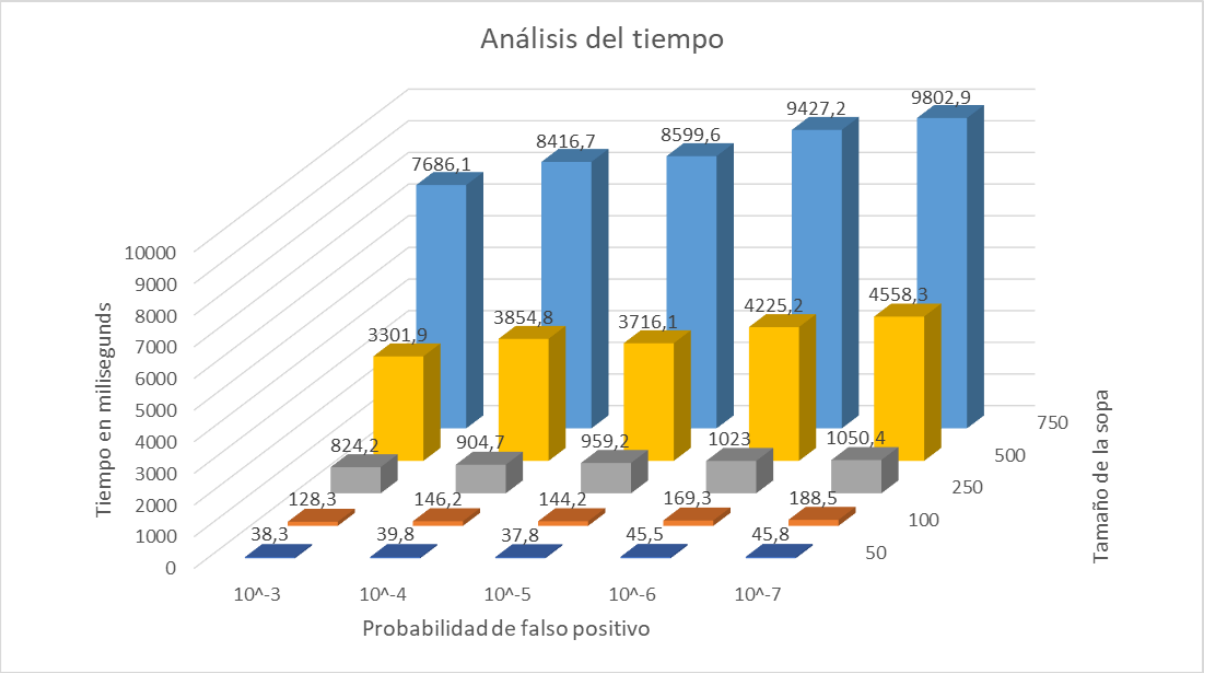

Figura 7: Tiempo obtenido experimentalmente respecto al tamaño de la Sopa

Hemos invertido los ejes con tal de poder visualizar más claramente los datos.
Como se puede observar, variar la precisión afecta ligeramente al tiempo de ejecución, aunque para nada aumenta acorde a lo que se reducen los fallos. El aumento es lineal, y es más pronunciado en tamaños de sopa más grandes.

El aumento del tiempo se debe a que se trabaja con un mayor número de hashes al aumentar la precisión, con lo que tardamos ligeramente más en comprobar cada elemento. En la siguiente gráfica se muestra dicho efecto.

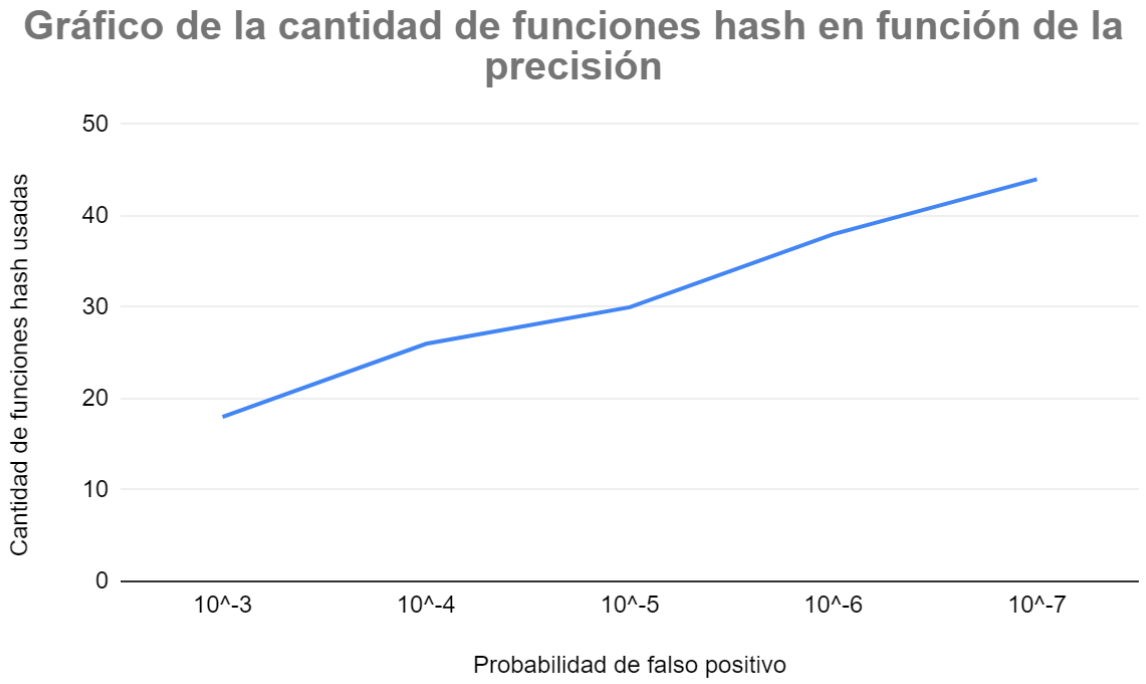

Figura 8: Relación entre la cantidad de funciones de hash y los falsos positivos

Como podemos ver, la cantidad de hashes usados aumenta de forma lineal, pero muy ligeramente, y es por eso que el tiempo aumenta también relativamente ligera y linealmente.

Finalmente, en el siguiente gráfico se muestra cómo evoluciona el uso de memoria y la cantidad de hashes usados según la precisión.

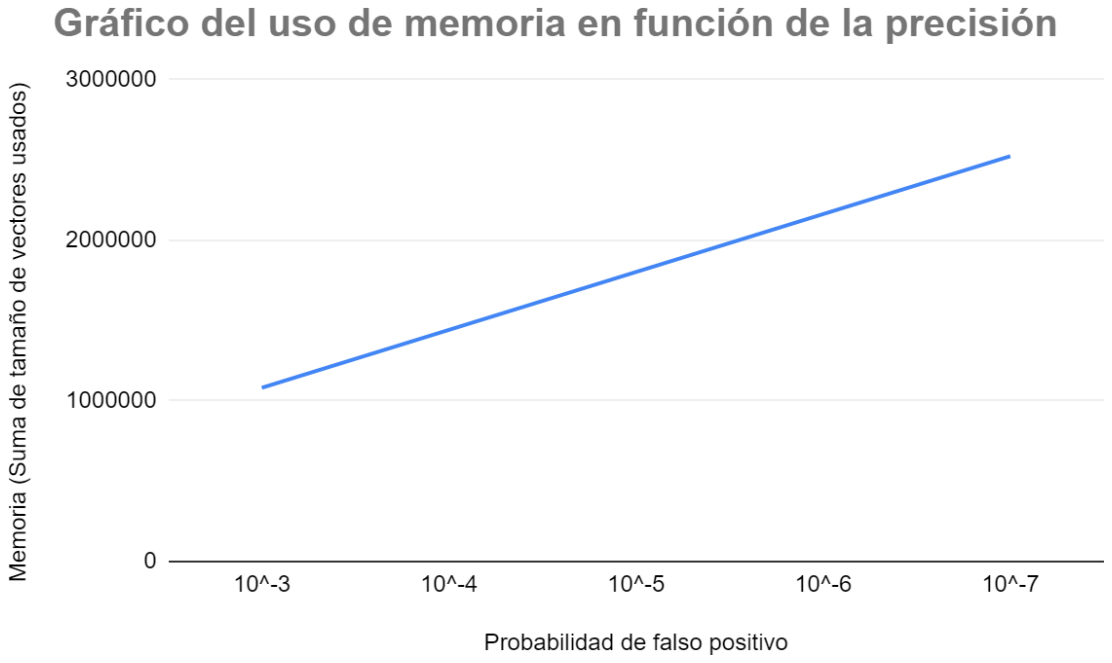

Figura 9: Gráfica representando la memoria usada en función de la probabilidad de falso positivo

Se puede observar que el uso de memoria aumenta mucho relativamente a la precisión, y de forma lineal.

En conclusión, el aumento en precisión supone un *trade-off* con tiempo y memoria, aunque no a partes iguales. Aumentar la precisión supone aumentar ligeramente el coste en tiempo de ejecución, y mucho en memoria. Además, experimentalmente hemos concluido que en las condiciones y características de este trabajo, una precisión de 10^{-7} era suficiente para no encontrar falsos positivos.

Conclusiones

Como hemos visto en los resultados, en el caso específico de este problema, las 2 estructuras de datos más eficientes con diferencia son el Trie y el Filtro de Boom, siendo este último el más óptimo. Aún y así, cabe destacar que sus rendimientos son muy parecidos y, por lo tanto, puede haber casos en los que sea mejor escoger uno frente al otro:

Como el Filtro de Bloom es probabilístico, es capaz de dar como resultado falsos positivos. Además, a más precisión se requiera, más se sacrificará en tiempo y memoria. Por este motivo, en los casos en los que obtener un resultado exacto y preciso sea de vital importancia, sería recomendable usar el Trie antes que pedir una precisión al Bloom que dispare su coste en memoria y tiempo. Por lo contrario, si la memoria o el tiempo son críticos pero se puede permitir cierto margen de error, el Filtro de Bloom prevalece por encima del Trie. Además, cabe destacar que este se puede aplicar a una gran diversidad de problemas de forma eficiente y no solo a búsquedas por prefijos como pasaría con el Trie.

Como podemos observar, la estructura de datos que peores resultados da es el Double Hashing. Esto nos ha sorprendido por ser esta una estructura relativamente sofisticada. Tras algo de análisis de la situación, vemos que esto se debe a que la estructura y sus puntos fuertes no se adecuan mucho a nuestro algoritmo de búsqueda. Puesto que su ventaja es en el acceso aleatorio y no en el secuencial por prefijos. Por lo tanto, no aprovecha la proximidad temporal ni espacial y hace que se dispare su coste y complejidad.

Finalmente, el vector ordenado tampoco brinda un resultado especialmente bueno. Puesto que, aunque técnicamente se pueda implementar una forma de búsqueda por prefijos, la estructura de datos no está precisamente preparada para ello, sufriendo así de los mismos contratiempos que la tabla de hash.

Bibliografia

Sorted Vector
https://cplusplus.com/reference/algorithm/lower_bound/
https://cplusplus.com/reference/algorithm/mismatch/

Funcionamiento general de Ternary Search Trees
https://en.wikipedia.org/
https://www.geeksforgeeks.org/ternary-search-tree/
https://www.drdobbs.com/database/ternary-search-trees/184410528

Costos de Ternary Search Trees
https://handwiki.org/wiki/Ternary_search_tree#Running_time

Funcionamiento Bloom Filter
https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/
https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf
https://en.wikipedia.org/wiki/Bloom_filter
https://octo.vmware.com/bloom-filter/

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

Función de hash para Bloom Filter

https://www.sderosiaux.com/articles/2017/08/26/the-murmur3-hash-function--hashtables-bloom-filters-hyperloglog/

Funcionamiento general de double hashing
https://www.geeksforgeeks.org/hashing-set-1-introduction/
https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/
https://www.geeksforgeeks.org/hashing-set-3-open-addressing/

Funciones de hash para strings
https://stackoverflow.com/questions/29759214/double-hashing-efficiency-with-word-dictionary

Parametrización tamaño de array
https://medium.com/swlh/why-should-the-length-of-your-hash-table-be-a-prime-number-760ec65a75d1
https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-8.html

Complejidad temporal de Double Hashing
https://core.ac.uk/download/pdf/82300919.pdf