# Proyecto Algorítmia

# Hashing

Albert Lobo Cusidó

Enero 2016

# Índice

# 1. Introducción

Se nos pide implementar un sencillo programa de correción ortográfica, cuyo funcionamiento consiste en buscar las palabras de un texto en su diccionario.

El objetivo de este proyecto es la validación experimental de la efectividad de diferentes algoritmos de hash y estructuras de datos, con la finalidad de implementar un corrector de ortografía lo más eficiente posible.

Para simplificar un poco el contexto, asumiremos que tanto el diccionario como el texto son un conjunto de enteros positivos.

Las estructuras de datos que analizaremos son:

- *Array ordenado (con búsqueda binaria)*
- *Árbol AVL*
- *Tabla de hash (direccionamiento abierto y encadenamiento separado)*
- *Bloom filter (versión básica y Spectral)*

Y las funciones de hash:

- *Murmur*
- *FNV*
- *H3*

# 2. Búsqueda binaria

Compararemos la eficacia de aplicar la búsqueda binaria en un array ordenado, con la de una estructura de datos más compleja: el árbol AVL.

El coste en espacio, en ambos casos, es de *O(n)*, donde $n$ es el número de palabras del diccionario.

El coste temporal de inicialización será de *O(n log n)*, debido a la necesidad de ordenar las palabras.

Y el coste temporal de buscar un vocablo será *O(log n)*.

## 2.1. Array ordenado

Representamos el diccionario como un array de enteros, y la corrección consiste en una simple búsqueda binaria en ese array.

En la inicialización es necesario ordenar las palabras del diccionario.

## 2.2. Árbol AVL

El propio AVL representa el diccionario, y la corrección se hace con la función *find*.

La inicialización se realiza creando el árbol a partir del diccionario ordenado.

# 3. Hash Table

Estas estructuras de datos prometen un *coste amortizado constante* para las operaciones de *inserción* y *búsqueda*, por lo que nuestra inicialización del diccionario se hará en *O(n)*, y cada correción de vocablo en *O(1)*.

El coste en espacio será proporcional al número de palabras del diccionario -*O(n)*.

Para ver la eficacia de las tablas de hash, hemos probado con dos de las estrategias de resolución de colisiones más conocidas: el *direccionamiento abierto* y el *encadenamiento separado*.

Una colisión se produce cuando dos elementos diferentes se mapean, debido a la función de hash, a la misma posición dentro de la tabla.

## 3.1. Open Addressing (direccionamiento abierto)

En caso de colisión, se inserta el nuevo elemento en la siguiente posición libre (nuestra implementación utiliza linear probing con intervalo de 1).

## 3.2. Separate Chaining (encadenamiento separado)

Cada posición en la tabla contiene no un elemento sino varios. Para nuestra implementación, hemos utilizado el *set* de la STL de C++.

# 4. Bloom Filter

Un Bloom filter es una estructura de datos probabilística utilizada para comprobar si un elemento pertenece a un conjunto: es posible que se produzcan 'falsos positivos', aunque nunca negativos. A cambio de un pequeño márgen de error, el Bloom filter ofrece mucha eficiencia en espacio y tiempo.

El espacio requerido depende del número de elementos, y de la probabilidad de falso positivo *fp* (nuestra implementación establece ésta a *1.0/n*):

*O(m)*, donde *m=ceil(-(n*log(fp)/(ln2*ln2)))*, y se considera lineal.

La inserción y búsqueda de una palabra tiene coste:

*O(k)*, donde *k=ceil(ln2*m/n)*, y lo podemos considerar constante.

## 4.1. Básico

La implementación más básica consiste en un array de bits, y sólo dispone de métodos para insertar elementos, y consultar si están en el conjunto.

## 4.2. Spectral

Esta variación del Bloom filter permite además eliminar elementos del conjunto, y consultar el número de apariciones de un elemento -ambas operaciones de coste constante-, utilizando un array de enteros en vez de uno de bits.

Obviamente, dado que no vamos a utilizar estas operaciones en nuestro corrector, este filtro no ofrece mejoras respecto del básico.

# 5. Funciones de hash

Las funciones de hash que nos interesa utilizar deben ser *funciones independientes*, de *distribución uniforme*, y *muy rápidas*.

Hemos descartado las *funciones criptográficas de hash* (como md5 o sha-1) debido a su lentitud.

## 5.1. Murmur

Consiste en aplicar una serie de multiplicaciones (MU) y rotaciones (R) a los bytes de la entrada -de ahí su nombre MURMUR-, para obtener el valor del hash.

## 5.2. FNV

Esta funcion parte de un hash inicial *offset*. Éste se multiplica por un *número primo*, y hace *XOR* con los bytes de la entrada.

## 5.3. H3

El hash se calcula haciendo *XOR* de los bytes de una tabla precalculada. Los bytes de la entrada determinan qué elementos de la tabla hay que tomar para las operaciones.

# 6. Programa corrector de ortografía

Hemos creado un programa que corrige un texto dado también su diccionario (detecta qué palabras están bien escritas, y cuáles no).

Nuestro corrector contiene implementaciones de todas las estructuras de datos y funciones de hash mencionadas, y permite al usuario seleccionar qué estrategia quiere utilizar para la corrección.

El usuario puede también seleccionar el test que quiere ejecutar (indicando el nombre del diccionario y el texto), o ejecutar todos los tests disponibles uno detrás de otro.

La compilación del ejecutable se realiza mediante el comando **make**. Para cada test que se ejecute con nuestro programa, se imprimirá:

- El número de palabras del diccionario y del texto, y cuántas están bien escritas.
- El tiempo (en segundos) invertido en la inicialización del diccionario, en la corrección, y en total.

Con el comando **make ver** obtendremos la versión "verbose", que imprime, además, los siguientes datos relevantes:

- Para la búsqueda binaria y el AVL, el número de iteraciones (o llamadas recursivas) realizadas durante la corrección.
- Para el resto de estructuras de datos, el número de hashes que se hacen durante la fase de inicialización y la de corrección.

El nombre del ejecutable es **spellcheck**, y su uso:

```
spellcheck -binary (basic|avl) [-all|test]
spellcheck -hash (oaht|scht|bf|sbf) (mur|fnv1|h3) [-all|test]
```

La opción −**binary** nos permite usar búsqueda binaria; es necesario indicar la estrucura de datos: **basic** (array ordenado), o **avl** (árbol AVL).

−**hash** nos facilita el uso de las estructuras de datos que utilizan hashing:

- **oaht** | **open-addressing-hash-table**
- **scht** | **separate-chaining-hash-table**
- **bf** | **bloom-filter**
- **sbf** | **spectral-bloom-filter**

Y las funciones de hash:

- **mur** | **murmur**
- **fnv1**
- **h3**

Con −**all** mandaremos la corrección de todos los juegos de prueba que haya en la carpeta *tests*. Alternativamente, podemos corregir un test concreto si indicamos su nombre (**test**).

Por ejemplo, con el comando

```
spellcheck -hash scht mur test-1000000-0.8
```

utilizaremos una Separate chaining hash table con Murmur hash para corregir el fichero de texto '*test-1000000-0.8.txt*' con el diccionario '*test-1000000-0.8.dic*' (ambos ficheros deben estar en la carpeta '*tests*');

```
spellcheck -binary basic -all
```

utilizará un array ordenado y búsqueda binaria para corregir todos los pares de ficheros '*.dic*' y '*.txt*' que tengan el mismo nombre.

# 7. Generador de juegos de pruebas

Creamos un programa que automatiza la generación de diccionarios y textos (en la carpeta *gen*).

Se recomienda realizar la compilación con el comando **make**.

El programa se puede ejecutar con el comando **spellgen**.

Dicho programa pide al usuario:

- El número de palabras $n$ del diccionario.
- La proporción de palabras $p$ correctamente escritas en el texto.

Y a continuación genera:

- El diccionario de $n$ palabras diferentes en un fichero *test-n-p.dic*.
- El texto en un fichero *test-n-p.txt*, en el que *aproximadamente* el ($p$*100) % de las palabras están bien escritas.

Todas las palabras se generan al azar.

Una vez creado un juego de pruebas, podemos hacerlo accesible al programa corrector *spellcheck*, simplemente copiando los ficheros generados ('*.dic*' y '*.txt*') a la carpeta *tests*.

Todos los tests que hemos utilizado para los experimentos se crearon con *spellgen*.

# 8. Experimentación

Nuestra hipótesis es que el *Bloom Filter* resultará idóneo para el corrector de ortografía; éste aporta un ahorro de espacio muy importante, ya que no se guardan las palabras propiamente. Además tanto la inserción como la consulta toman tiempo constante.

Esto nos lleva a pensar que será la estructura de datos más eficiente en espacio y tiempo.

Además, queremos ver si la cantidad de palabras bien escritas en el texto influye en el rendimiento de las diferentes estructuras de datos. Para ello, hemos creado juegos de prueba con diferentes cantidades de errores -un *juego* de puebas o *test* consiste de un diccionario (un fichero .dic) y un texto a corregir (un .txt), que contienen palabras generadas aleatoriamente.

Vamos desde el 0 al 100 % de palabras correctas, a saltos de 10 %. Para cada porcentaje, ejecutamos 20 tests con diccionarios de $10^6$ palabras, y textos de $2*10^6$ palabras.

Todos los experimentos se han realizado con una versión de nuestro *spellcheck* modificada para combinar múltiples tests, y escribir los resultados en un fichero.

## 8.1. Función de hash

Hemos procedido a comparar los tiempos de ejecución de las diferentes estructuras de datos con las tres funciones de hash.

### 8.1.1. Hash tables

Por cuanto a la estrategia de Open addressing, vemos que tanto Murmur como H3 dan unos tiempos parecidos -siendo Murmur ligeramente más rápida -, mientras que FNV puede llegar a ser, en el peor de los casos, casi 4 veces más lenta que las otras. La explicación es que FNV, en combinación con esta estructura de datos, provoca muchas colisiones, y es la resolución de éstas quien causa el overhead.

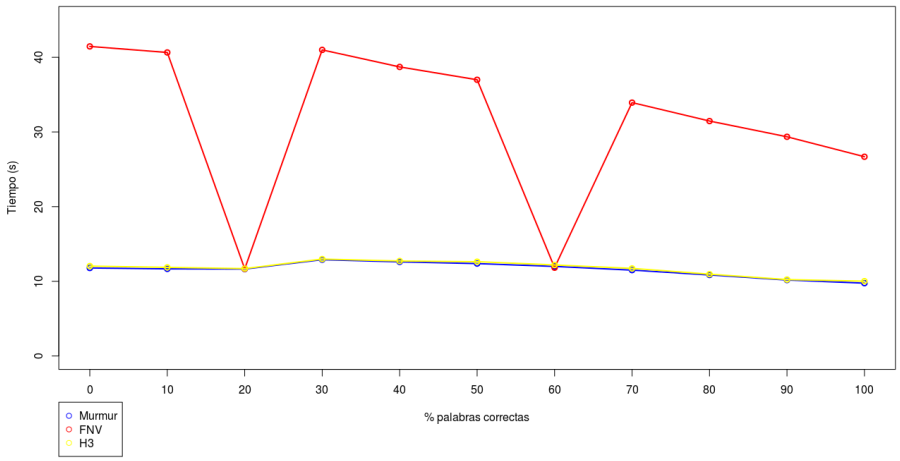


Figura 1: *Funciones de hash para Open addressing*

Con Separate chaining, los tiempos no varían tanto -siendo Murmur la más rápida con textos medianamente bien escritos -, pero son notablemente mayores que la primera estrategia de resolución de colisiones.

En vista de los resultados, optamos por la función Murmur como la mejor para ambas estrategias de resolución de colisiones.

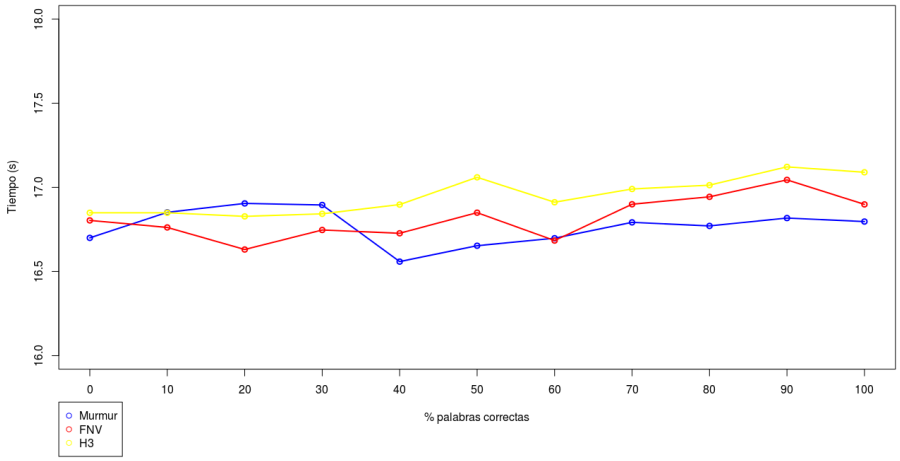


Figura 2: *Funciones de hash para Separate chaining*

### 8.1.2. Bloom filter

Queremos ver si aquí también resultará ser Murmur la función idónea. Veamos la Figura 3: Parece que FNV esta vez se comporta un poco más rápido que Murmur, mientras que H3 aquí queda claramente por detrás.

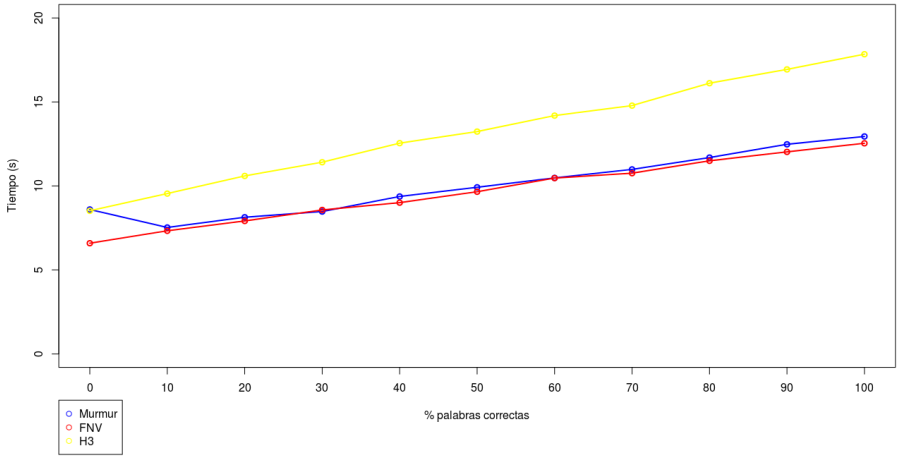


Figura 3: *Funciones de hash para el Bloom filter básico*

Queremos saber también qué función de hash produce menos falsos positivos.

| **Test** | | **Murmur** | | **FNV** | | **H3** | |
|---|---|---|---|---|---|---|---|
| **#** | **pc** | **pc** | $\varepsilon_{\mathbf{r}}$ | **pc** | $\varepsilon_{\mathbf{r}}$ | **pc** | $\varepsilon_{\mathbf{r}}$ |
| 0 | 399542 | 399566 | 0.006 % | 399595 | 0.013 % | 399577 | 0.009 % |
| 1 | 4400048 | 4400087 | $8{,}863 * 10^{-4}$ % | 4400087 | $8{,}863 * 10^{-4}$ % | 4400089 | $9{,}318 * 10^{-4}$ % |
| 2 | 8399527 | 8399562 | $4{,}167 * 10^{-4}$ % | 8399555 | $3{,}333 * 10^{-4}$ % | 8399551 | $2{,}857 * 10^{-4}$ % |
| 3 | 12405451 | 12405483 | $2{,}579 * 10^{-4}$ % | 12405479 | $2{,}257 * 10^{-4}$ % | 12405479 | $2{,}257 * 10^{-4}$ % |
| 4 | 16399817 | 16399846 | $1{,}768 * 10^{-4}$ % | 16399839 | $1{,}341 * 10^{-4}$ % | 16399847 | $1{,}829 * 10^{-4}$ % |
| 5 | 20397123 | 20397145 | $1{,}078 * 10^{-4}$ % | 20397148 | $1{,}226 * 10^{-4}$ % | 20397140 | $8{,}334 * 10^{-5}$ % |
| 6 | 24400001 | 24400022 | $8{,}606 * 10^{-5}$ % | 24400015 | $5{,}738 * 10^{-5}$ % | 24400016 | $6{,}147 * 10^{-5}$ % |
| 7 | 28398595 | 28398608 | $4{,}578 * 10^{-5}$ % | 28398604 | $3{,}169 * 10^{-5}$ % | 28398605 | $3{,}521 * 10^{-5}$ % |
| 8 | 32400618 | 32400622 | $1{,}234 * 10^{-5}$ % | 32400627 | $2{,}778 * 10^{-5}$ % | 32400626 | $2{,}469 * 10^{-5}$ % |
| 9 | 36400023 | 36400028 | $1{,}377 * 10^{-5}$ % | 36400025 | $5{,}494 * 10^{-5}$ % | 36400027 | $1{,}099 * 10^{-5}$ % |
| 10 | 40000000 | 40000000 | 0 % | 40000000 | 0 % | 40000000 | 0 % |

$pc$ = Número de palabras correctas

$\varepsilon_{\mathbf{r}}$ = Error relativo

Cuadro 1: *Errores relativos de las funciones de hash para los diferentes tests*

Lo primero que nos llama la atención es que la probabilidad de falso positivo es, en la práctica, mucho mayor que la que previmos. Basta con observar el *test 0*, en que todas las palabras están mal; si establecimos la $fp = 1{,}0/n = 10^{-6}$, con $2*10^6$ palabras deberíamos obtener sólo 2 falsos positivos. Atribuimos este hecho al comportamiento de las funciones de hash.

FNV es la función que más falsos positivos da.

Murmur y H3 muestran errores relativos bastante parecidos; pero teniendo en cuenta que Murmur es claramente más rápido, concluimos que es la función de hash más adecuada para el Bloom filter.

## 8.2. Búsqueda binaria

En las siguientes gráficas podemos ver los tiempos de ejecución de la búsqueda binaria sobre un array ordenado, y del árbol AVL. Como era de esperar, el AVL presenta tiempos mayores, pero proporcionales a los del array ordenado, debido a la implementación de la estructura de datos.

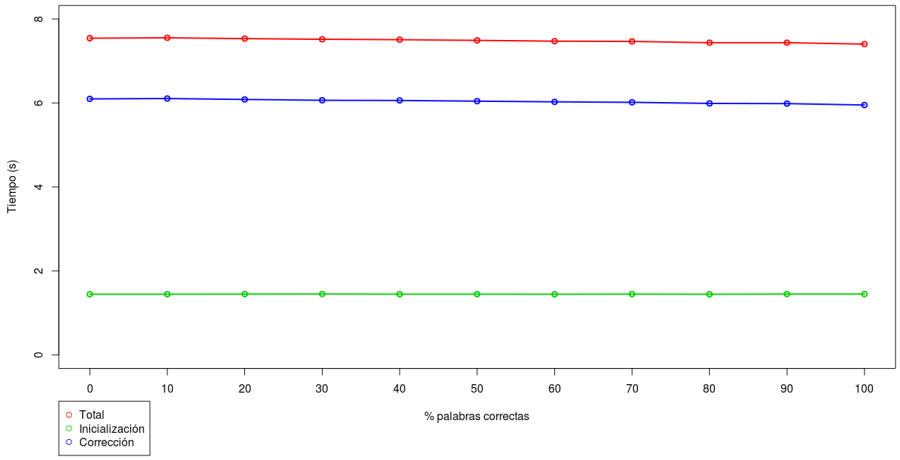


Figura 4: *Array ordenado + búsqueda binaria*

Podemos ver que la búsqueda binaria es ligeramente más eficiente cuanto más bien escrito está el texto -esto es debido a la forma del algoritmo, ya que hay que recorrer el árbol binario hasta una hoja para determinar que una palabra no está.

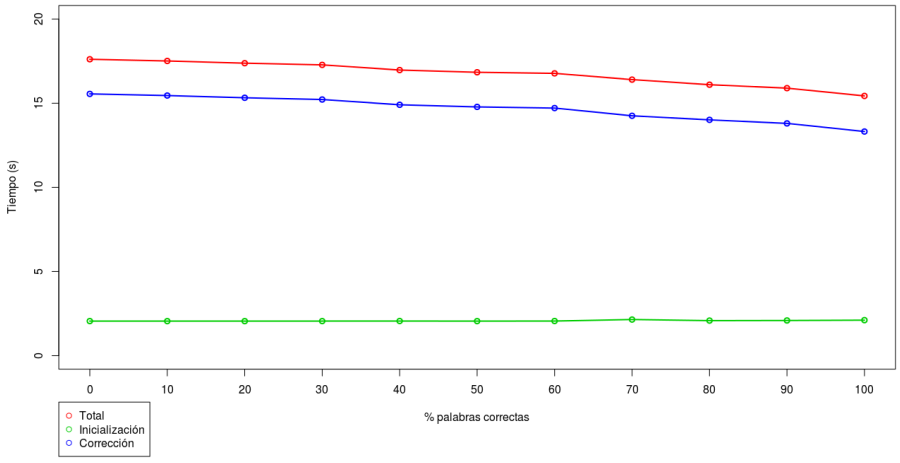


Figura 5: *AVL*

## 8.3. Hash Table

En las siguientes figuras podemos ver los tiempos de ejecución de los tests, utilizando tablas de hash, con las dos estrategias de resolución de colisiones anteriormente mencionadas, y la función de hash Murmur.

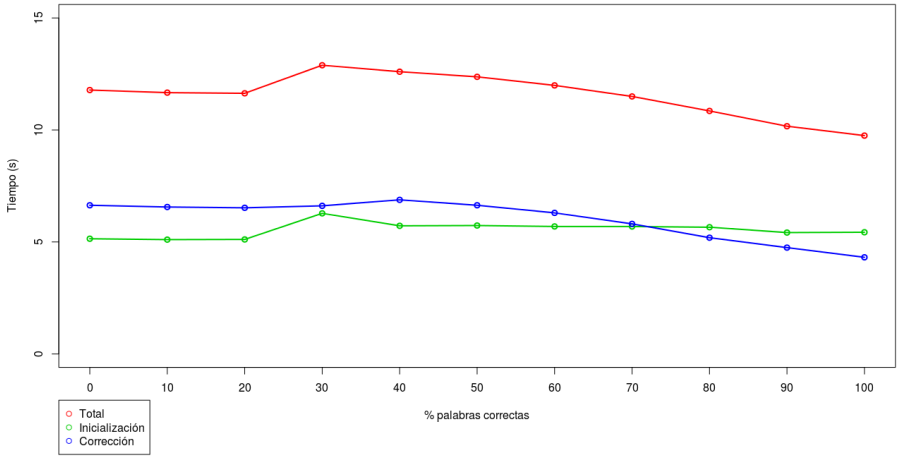


Figura 6: *Open addressing*

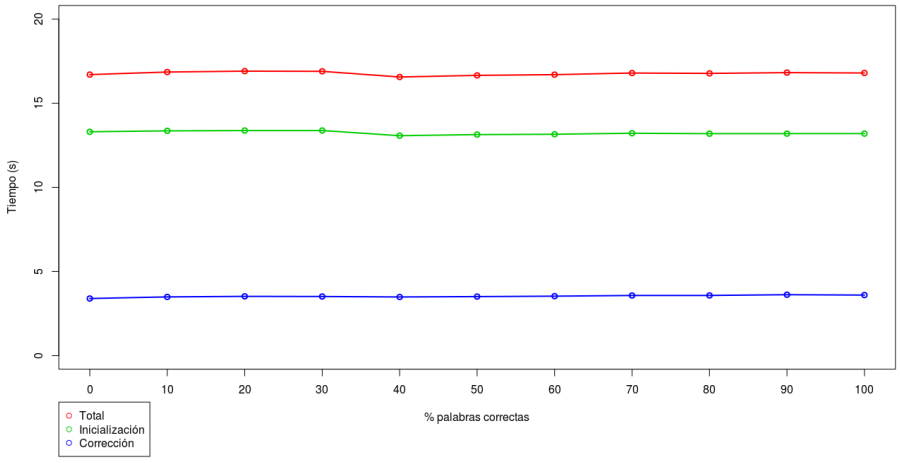


Figura 7: *Separate chaining*

Open addressing parece mejorar ligeramente su rendimiento cuanto mejor escrito está el texto, mientras que en Separate chaining se mantiene constante.

A juzgar por el tiempo total de ejecución, Open addressing parece mejor opción. Pero antes miremos los tiempos de inicialización y corrección: aunque Separate chaining tarda más en inicializarse, corrige siempre más rápido que la alternativa. Ya que suponemos que la cantidad de palabras de un texto puede ser mucho mayor que la que hemos utilizado para los tests, tomamos Separate chaining como la mejor de las dos opciones (en caso de decidirnos por una hash table).

## 8.4. Bloom Filter

Al igual que con la búsqueda binaria, tenemos dos estructuras de datos de igual funcionamiento: el Bloom filter básico, y el Spectral. Debido a la funcionalidad que aporta el Spectral (*delete* y *count*), su representación interna se basa en *int* en lugar de *bool* -caso del Bloom filter básico.

Hemos representado gráficamente los tiempos de ejecución de los tests, utilizando Bloom filters y la función de hash Murmur:

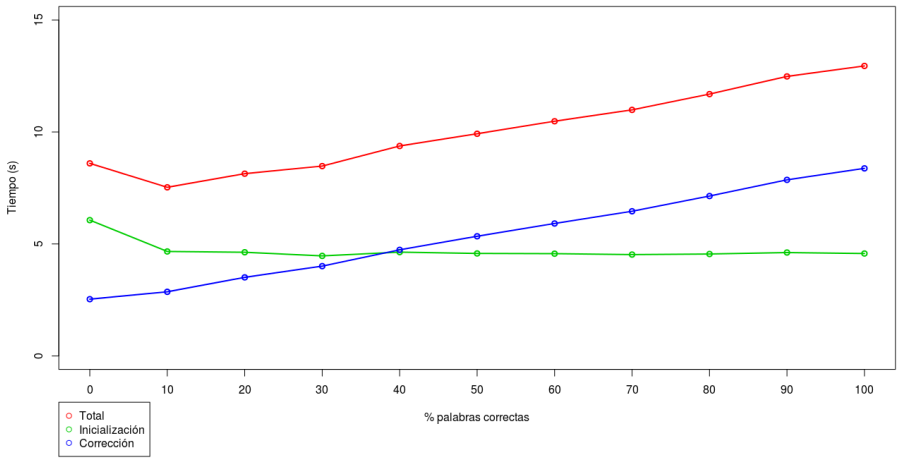


Figura 8: *Basic Bloom filter*

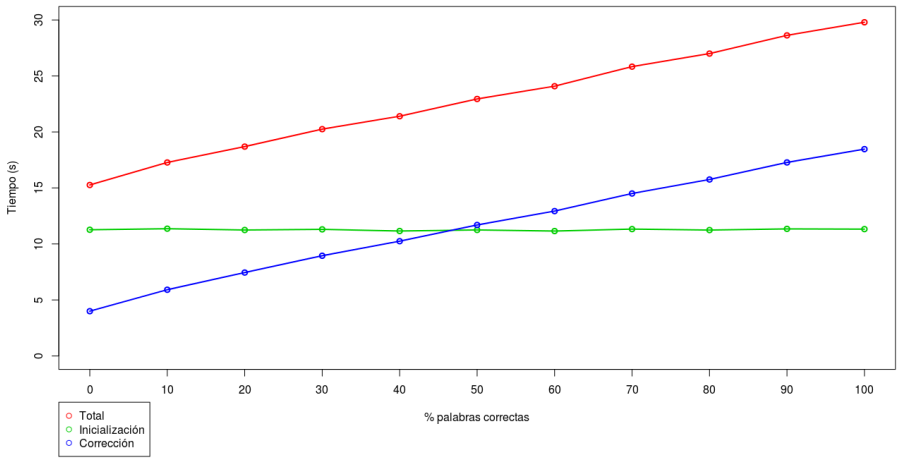


Figura 9: *Spectral Bloom Filter*

Tal como esperábamos, ambas gráficas muestran el mismo comportamiento, y el Spectral muestra un overhead debido a su implementación -¡es aproximadamente el doble de lento!

Podemos ver que el tiempo de corrección es directamente proporcional a la cantidad de palabras correctas. Ciertamente se debe a que el algoritmo de comprobación debe correr todas las funciones de hash antes de afirmar que un elemento pertenece al conjunto.

# 9. Resultados y conclusión

Con el fin de decidir la mejor estrategia para nuestro corrector de ortografía, comparamos los tiempos de corrección de texto de las estructuras que mejores resultados nos han dado:

- Array ordenado + búsqueda binaria
- Separate chaining hash table con función de hash Murmur
- Bloom filter básico con función de hash Murmur

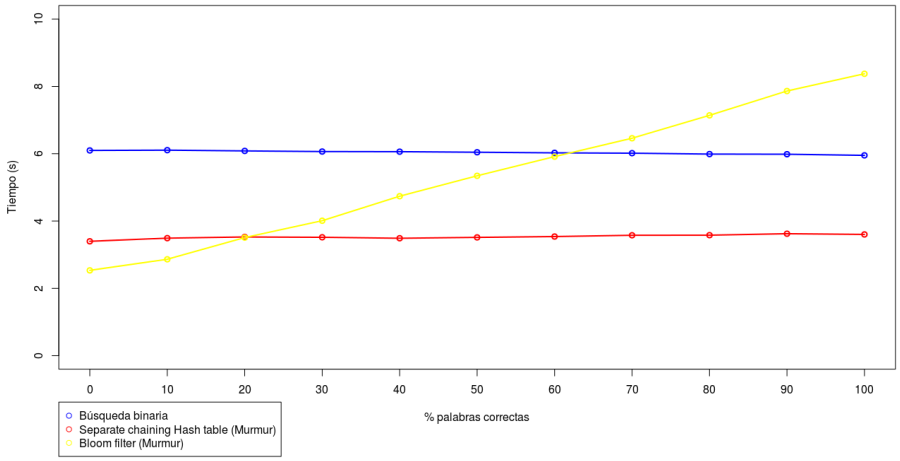


Figura 10: *Tiempos de corrección*

El *Bloom filter* no ha resultado ser la mejor estrategia; se vuelve lento cuando el texto está bien escrito, y la cantidad de falsos positivos (palabras inexistentes sin corregir) es inaceptable.

*Separate chaining hash table con función de hash Murmur* es nuestra primera opción por su presteza en la corrección del texto. Damos menos importancia al tiempo de inicialización ya que la carga del diccionario sólo se producirá una vez al arrancar el programa, mientras que el número de palabras a corregir dependerá del texto, que puede ser mucho mayor.

# Bibliografía


[1] National Institute of Standards y Technology. *Binary Search*. URL: https://xlinux.nist.gov/dads//HTML/binarySearch.html.

[2] Wikipedia. *Binary Search Algorithm*. URL: https://en.wikipedia.org/wiki/Binary_search_algorithm.

[3] Donald Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997. ISBN: 9780201896831.

[4] National Institute of Standards y Technology. *AVL Tree*. URL: https://xlinux.nist.gov/dads//HTML/avltree.html.

[5] Wikipedia. *AVL Tree*. URL: https://en.wikipedia.org/wiki/AVL_tree.

[6] Mark A. Weiss. *Data Structures and Algorithm Analysis in C++ (3rd ed.)* Addison-Wesley, 2006. ISBN: 0-321-44146-X.

[7] National Institute of Standards y Technology. *Hash Table*. URL: https://xlinux.nist.gov/dads//HTML/hashtab.html.

[8] Wikipedia. *Hash Table*. URL: https://en.wikipedia.org/wiki/Hash_table.

[9] National Institute of Standards y Technology. *Open Addressing*. URL: https://xlinux.nist.gov/dads//HTML/openAddressing.html.

[10] Wikipedia. *Open Addressing*. URL: https://en.wikipedia.org/wiki/Open_addressing.

[11] Robert Holte. *Open Addressing*. URL: https://webdocs.cs.ualberta.ca/~holte/T26/open-addr.html.

[12] National Institute of Standards y Technology. *Separate Chaining*. URL: https://xlinux.nist.gov/dads//HTML/separateChaining.html.

[13] Paul Kube. *Open addressing vs. separate chaining*. URL: http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-25.html.

[14] University of Washington. *Collision resolution*. URL: http://courses.cs.washington.edu/courses/cse326/06su/lectures/lecture11.pdf.

[15] National Institute of Standards y Technology. *Bloom Filter*. URL: https://xlinux.nist.gov/dads//HTML/bloomFilter.html.

[16] Pei Cao. *Bloom Filters - the math*. URL: http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html.

[17] Elliott Karpilovsky. *Bloom Filters and Applications*. URL: http://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/bloom_filters.pdf.

[18] Wikipedia. *Bloom Filter*. URL: https://en.wikipedia.org/wiki/Bloom_filter.

[19] Bill Mill. *Bloom Filters by Example*. URL: http://billmill.org/bloomfilter-tutorial.

[20] Perl.com. *Using Bloom Filters*. URL: http://www.perl.com/pub/2004/04/08/bloom_filters.html.

[21] J. Lawrence Carter y Mark N. Wegman. «Universal Classes of Hash Functions». En: *Journal of Computer and System Sciences* 18.2 (1979), págs. 143-154. DOI: https://www.cs.princeton.edu/courses/archive/fall09/cos521/Handouts/universalclasses.pdf.

[22] M.V. Ramakrishna, E. Fu y E. Bahcekapili. «A Performance Study of Hashing Functions for Hardware Applications». En: *In Proc. of Int. Conf. on Computing and Information*. 1994, págs. 1621-1636. DOI: http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.47.7985.

[23] Microsoft. *H3 Hash*. URL: https://msdn.microsoft.com/en-us/library/dd304662.aspx.

[24] Landon Curt Noll. *FNV Hash*. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[25] Kiem-Phong Vo Landon Curt Noll y Donald Eastlake. *The FNV Non-Cryptographic Hash Algorithm*. URL: https://tools.ietf.org/html/draft-eastlake-fnv-10.

[26] Wikipedia. *Fowler–Noll–Vo hash function*. URL: https://en.wikipedia.org/wiki/Fowler%C3%A2%C2%80%C2%93Noll%C3%A2%C2%80%C2%93Vo_hash_function.

[27] Boost.org. *FNV1 Example*. URL: http://www.boost.org/doc/libs/1_38_0/libs/unordered/examples/fnv1.hpp.

[28] Austin Appleby. *MurmurHash*. URL: https://sites.google.com/site/murmurhash/.

[29] Wikipedia. *MurmurHash*. URL: https://en.wikipedia.org/wiki/MurmurHash.

[30] Austin Appleby. *MurmurHash3*. URL: http://code.google.com/p/smhasher/wiki/MurmurHash3.