EL PROBLEMA DEL DICCIONARIO: ASSOCIATIVE ARRAYS Y MEDIDAS DE SIMILITUD.

Mario Garrido

Universidad de Chile
Santiago, Chile

ABSTRACT

Se estudiaron 3 implementaciones de *Associative arrays* para *multisets*: *Patricia Tree*, *Ternary Search Tree* y *Dynamic Hash Table* con *linear probing*. Los resultados muestran que la tabla de hash, manteniendo el *load factor* bajo el 40 %, es órdenes de magnitud más rápida que las otras estructuras para las operaciones *insert* y *search*. Si bien, en general, los *Patricia trees* son la estructura más lenta, sus costos de espacio escalan bien con el aumento del tamaño del alfabeto, a diferencia del *Ternary Search Tree*, que se ve obligado a agregar más *branches* y aumentar su altura con la inclusión de nuevos caracteres. Una ventaja de los árboles, que podrían hacerlos preferibles a la tabla, es que pueden soportar operaciones como recorrido o sucesor, por lo que su aplicación puede ser más amplia. Otro aspecto redentor del árbol *Patricia* es que se puede almacenar el diccionario en memoria externa, ya que no es necesario consultarlo para recorrer el árbol (*Blind trie*), y solo es menester consultar cuando se ha llegado a una hoja, muy en el espíritu del algoritmo *PATRICIA* original.

1. INTRODUCCIÓN

Los objetivos perseguidos en el presente trabajo son los siguientes:

- Realizar una implementación de *Patricia Tree*, *Ternary Search Tree* y *Hash Table con linear probing*.
- Realizar experimentos que permitan medir el comportamiento de las estructuras implementadas frente a distintas fuentes de texto.
- Los experimentos deben ayudar a dilucidar los siguientes puntos:
  1. ¿Qué tanto afecta la permutación de la entrada a los tiempos de inserción y búsqueda?
  2. ¿Qué estructura tiene mejores tiempos de inserción y búsqueda?
  3. ¿Aquellas estructuras que son sub-óptimas en tiempo tienen algún otro atractivo confirmable a través de la experimentación?
- Concluir respecto a las hipótesis.

Las hipótesis a estudiar son las siguientes:

- La *Hash Table* debe presentar los tiempos de operación más bajos, no solo porque su estructura es ideal para el *set* reducido de operaciones consideradas (*insert* y *search*), sino que los tamaños del problema considerado no presentan problema para mantener un *load factor* bajo.
- *Ternary Search Tree* es vulnerable a malas permutaciones de la entrada, ya que los primeros patrones insertados afectan fuertemente el recorrido de todos los subsiguientes.
- El costo en espacio del *Patricia Tree* debería ser inferior al del *Ternary Search Tree* y ser competitivo con la *Hash Table*.

2. IMPLEMENTACIÓN

Para realizar la implementación se utilizó el lenguaje C++, y está pensada para el *standard* de C++11.

Las 3 estructuras descritas a continuación implementan las siguientes operaciones, como interfaz común:

- insert: Dada una *key*, un valor y un entero 0 o 1, inserta el par $(key, valor)$ en la estructura. El entero se utiliza para codificar el texto de procedencia del par, por lo que cada estructura puede guardar los valores correspondientes a 2 textos distintos.
- search: Dada una *key*, inspecciona si esta se encuentra contenida en la estructura y devuelve el elemento que da acceso a su información (nodo en caso de los árboles y *slot* en caso de la tabla).
- search_report: Dada una *key* y un entero que codifica el texto de procedencia, retorna 1 si se encuentra en la estructura y 0 si no.
- get_name: Retorna el nombre de la estructura (i.e: *PATR*, *TERN* o *HASH*).
- structure_size: Retorna el tamaño total, en *bytes*, de la estructura.

- occurences: Dada una *key*, retorna el número de ocurrencias del patrón en cada texto, por separado.

- extra_measurement: Retorna un dato relevante de la estructura, para los árboles corresponde a la altura del árbol y para la tabla corresponde al *load factor* o *fill rate*.

- delete_data: Borra completamente los elementos de la estructura y libera la memoria.

Es a través de esta interfaz común que se llevan a cabo los experimentos, en el archivo *experiments.cpp*.

2.1. Patricia Tree

Esta sección describe la implementación correspondiente a los archivos patricia.cpp y patricia.hpp adjuntos.

Un *Patricia Tree* es, esencialmente, un *Radix Tree* de aridad 2, esto implica que las *key* se comparan *bit* a *bit*, hasta encontrar un punto en que se difiera. Esto entrega una mayor granularidad en la cantidad de *branches* que se pueden crear, a diferencia del *Ternary Search Tree*, que compara de 1 caracter a la vez (8 *bits*). Vale la pena notar que, tanto en esta implementación como en la de *Ternary Search Tree*, se le agrega un carácter menor que todos los utilizados por los patrones a cada patrón. El carácter utilizado para este fin corresponde al *(char)1*.

El primer problema que es necesario resolver es el de la representación binaria de las llaves. Si bien, un *char* es, normalmente, 1 *byte* en C/C++, y es, por tanto, un número con una representación binaria en el computador, lo que implica que puede ser operado con operadores binarios, esto no significa que dichas operaciones sean necesariamente eficientes. Era necesario tener acceso a una representación sencilla y rápida de las *key* en formato binario, para poder navegar el árbol, por lo que se decidió utilizar *boost::dynamic_bitset*, que es un contenedor que permite operar *bits*. Hay que notar que el almacenamiento de los *bits* se hace en bloques de *longs*, por lo que sigue existiendo un *overhead* al acceder a cualquier posición del *dynamic_bitset*. Es por este *overhead* y por el costo adicional de tener que transformar los *strings* de las llaves en *dynamic_bitsets* que se implementó, además, un *Patricia Tree* con operadores binarios sobre *strings*, pero resultados preliminares demostraron que su comportamiento era peor. Para entender por qué, notamos que el costo adicional de tener que acceder los *bits* del *string* se paga por cada *query*, mientras que el costo de transformar a *dynamic_bitset* se paga solo una vez, y experimentalmente se demostró que el costo de acceder a las posiciones es bajo.

Un nodo cuenta con los siguientes campos:

- left_child: Puntero al hijo izquierdo del nodo. El hijo izquierdo es aquel al que se debe avanzar si el *bit* comparado es 0.

- right_child: Puntero al hijo derecho del nodo. El hijo derecho es aquel al que se debe avanzar si el *bit* comparado es 1.

- leaf: Puntero a cualquier hoja que se encuentre hacia abajo del nodo. Este puntero se mantiene para agilizar las operaciones de inserción, ya que permite no tener que estar recorriendo el árbol hasta encontrar una hoja.

- left_offset: Entero correspondiente al salto en el patrón si es que el *bit* comparado es igual a 0 y se avanza, por tanto, por la rama izquierda.

- right_offset: Entero correspondiente al salto en el patrón si es que el *bit* comparado es igual a 1 y se avanza, por tanto, por la rama derecha.

- text_position: Vector que contiene los valores asociados a la *key* que representa el nodo. Esto solo es importante para los nodos hoja.

El procedimiento para recorrer el árbol es el siguiente:

1. Si la raíz no es *nullptr* se entra a ella y se consulta el primer *bit* del patrón. Si es 1 se avanza al hijo derecho y si es 0 al hijo izquierdo.

2. Si se avanzó hacia la derecha, entonces la siguiente posición a consultar en el patrón es la posición anterior + *right_offset*, en caso contrario se avanza *left_offset*.

3. Se consulta la siguiente posición y se avanza de acuerdo al resultado.

4. Si se alcanza una hoja se compara el patrón con la *key* codificada por la hoja, y si son iguales se ha encontrado el patrón.

Para la inserción se sigue, en esencia, la lógica descrita en el enunciado, pero en la implementación actual existen 4 posibilidades al momento de una inserción. Las funciones involucradas en el proceso de inserción son las siguientes:

- insert: Primero, revisa si la raíz es *nullptr* y, si lo es, la crea con un hijo que contiene al nuevo patrón. Si la raíz no es nula, entonces se busca en el árbol una hoja que pueda contener el patrón. Cuando se tiene el candidato se revisa si es la hoja que codifica el patrón, buscando el punto en que difieren, y si lo es entonces solo se le agrega un valor adicional al vector *text_position*, en caso contrario se reinserta desde esa hoja, utilizando la información del punto donde difieren. En caso de que el candidato no sea una hoja, buscamos el punto donde el patrón difiere de alguna hoja del nodo (utilizaremos, para ahorrar tiempo, la hoja apuntada por el nodo a través del puntero *leaf*) y reinsertamos desde esa hoja.

- node_search: Este método recorre el árbol leyendo un *bit* del nuevo patrón a la vez, hasta un límite determinado o hasta que se alcanza una hoja. Cuando *insert* llama a *node_search* se busca hasta que se salta a un punto más largo que el patrón, o se alcanza una hoja, pero cuando es llamado por *reinsert_from_leaf* se busca hasta que se alcanza el punto en que el patrón difiere de la *key* de la hoja, o se alcanza una hoja. Notamos que puede pasar que el punto punto del patrón hasta el que se saltó podría ser antes, después o igual al punto en que difiere de la *key* de la hoja. Esta diferencia será utilizada para realizar la reinserción.
- find_max_prefix: Calcula la primera posición en la que el nuevo patrón difiere de la *key* codificada por la hoja.
- reinsert_from_leaf: Llama nuevamente a *node_search*, pero solo permite explorar hasta el primer punto en que el patrón difiere de la *key* de la hoja. Si el punto al que se llegó fue exactamente el punto de diferencia y el nodo candidato encontrado corresponde a una hoja, entonces se invoca *split_leaf*. Si el candidato no es una hoja, entonces solo puede corresponder a un caso: se trata de la raíz que tiene 1 solo hijo, por lo que se inserta el hijo restante. Si saltamos a un punto posterior a la primera diferencia del patrón con la *key* del nodo hoja, entonces es necesario cortar el arco y agregar un nodo que tenga como hijos a un nuevo nodo que codificará al patrón agregado y al nodo que originalmente recibía ese arco. Si saltamos a un punto anterior a la primera diferencia, entonces es necesario extender el arco y separar el camino.
- split_leaf: Se crea, en el punto de la hoja, un nuevo nodo con 2 arcos: uno hacia la hoja original y uno hacia el nuevo nodo que contiene al patrón agregado. El nuevo nodo padre tendrá, además, un puntero al nuevo hijo en *leaf*, así nos evitamos tener que recorrer el árbol para encontrar una hoja si es que es necesario reinsertar desde una hoja de este nodo.
- insert_missing_son: Se inserta el hijo restante al nodo raíz. Ya que no hay borrados, este es el único momento en que existe un nodo con un solo hijo.
- split_arc: Es equivalente a *split_leaf*, pero es necesario modificar el *offset* del padre, para consultar el *bit* relevante, y el offset del nuevo nodo que tiene como hijo al nodo del patrón agregado y al nodo original, para que el camino original conserve sus saltos.
- extend_arc_and_split_leaf: Análogo al caso anterior, pero se extiende el salto dado por el *offset* del padre.

Un punto interesante del *Patricia Tree* es que, independientemente del órden con que se inserten las *keys*, el árbol resultante será siempre el mismo, por lo que el costo en espacio para un determinado *set* de *keys* es fijo.

2.2. Ternary Search Tree

Esta sección describe la implementación correspondiente a los archivos ternary.cpp y ternary.hpp adjuntos.

La estructura del *Ternary Search Tree* es, conceptualmente, más simplista que el *Patricia Tree*. Recorremos los nodos comparando 1 carácter a la vez y navegamos al hijo correspondiente en base al resultado de la comparacíón. Un nodo *ternary_node* cuenta con los siguientes campos:

- lesser_child: Puntero al nodo menor. Este nodo es aquel en el que se continúa la búsqueda si el carácter comparado es menor al carácter almacenado en el nodo.
- equal_child: Puntero al nodo del centro. Este nodo es aquel en el que se continúa la búsqueda si el carácter comparado es igual al carácter almacenado en el nodo.
- greater_child: Puntero al nodo mayor. Este nodo es aquel en el que se continúa la búsqueda si el carácter comparado es mayor al carácter almacenado en el nodo.
- character: Corresponde al carácter almacenado en el nodo.
- text_position: Vector que contiene los valores asociados a una llave.
- counts_as_lesser: *Boolean* que indica si el nodo cuenta como el carácter más pequeño.

La lógica para recorrer el árbol es la siguiente:

1. Si la raíz no es vacía, comparamos el primer carácter del patrón con el patrón almacenado en la raíz y continuamos la búsqueda en el hijo correspondiente.
2. En caso de que continuemos la búsqueda por el *lesser* o *greater child* utilizaremos la totalidad del patrón nuevamente, esto implica que volveremos a consultar el mismo carácter al nodo al que lleguemos. Los carácteres solo se consumen cuando se desciende por *equal child*.
3. Cuando se llega a un nodo que tiene el mismo carácter que el que se está consultando y todo lo que resta del patrón corresponde al carácter más pequeño (*(char)1*), entonces hemos terminado la búsqueda si es que el *equal child* contiene ese patrón.

Notaremos que la inserción corresponde a la búsqueda, pero en caso de que un nodo del camino no exista se va creando. Un detalle importante, y la razón de por qué los nodos contienen el campo *counts_as_lesser*, es que se podría querer insertar un nodo que contenga el carácter más pequeño como *equal child* de un nodo que ya contiene un *equal child* con un carácter distinto. En ese caso se marca ese *equal child* con *counts_as_lesser = 1*, lo que permite al algoritmo de búsqueda considerarlo como ambos nodos. Todos los nodos que tienen activo ese campo son nodos que guardan valores asociados a *keys*.

2.3. Dynamic Hash Table, Linear Probing

Esta sección describe la implementación correspondiente a los archivos linear_hash.cpp y linear_hash.hpp.

La tabla de hash implementada utiliza la estrategia de *linear probing*, que consiste en aplicar una función de hash sobre la *key* y explorar la tabla desde el valor obtenido en adelante, hasta encontrar la *key*. Si se encuentra un espacio vacío, lo que se codifica con el *string* vacío en la implementación, se detiene la búsqueda, ya que no se encuentra en la tabla. En el caso de la inserción, si se encuentra un espacio vacío entonces se procede a insertar la *key* en ese espacio. Como función de hash[1] se utiliza *std::hash<std::string>* h, la que cumple $\forall x, y; x \neq y \Rightarrow P(h(x) = h(y)) < \frac{1}{std::numeric\_limits<std::size\_t>::max()}$, lo que se considera satisfactorio para los fines de la implementación actual.

La tabla es, además, dinámica, y dobla su tamaño cuando se supera un *fill rate* α de 0.4. El costo amortizado de inserción es, por lo tanto, $\mathcal{O}(1)$, para una función potencial $\Phi = 5T.num - T.size$, y esto se nota bastante en los resultados obtenidos por la implementación.

3. EXPERIMENTOS

Se realizaron 3 experimentos distintos, con el fin de estudiar el comportamiento de las estructuras frente a la mayor cantidad de variables posible. Vale la pena notar que el estado final de *Ternary Search Tree* y de *Hash Table* depende del orden con que se insertan las llaves, tomando esto en consideración es que se permuta la entrada para la inserción. Los resultados para cada permutación se promedian y esto constituye el resultado de una iteración. El primer experimento realiza un número de iteraciones (donde se crea un texto aleatorio), dentro de las cuales hay un número dado de permutaciones (donde se permuta el texto creado).

3.1. Random experiment

El objetivo de este experimento es analizar el impacto del tamaño del alfabeto $|\Sigma|$ sobre los tiempos de construcción y el costo en espacio de las estructuras. Con este fin se generan palabras de forma aleatoria para distintos $|\Sigma|$ y se insertan a las estructuras. La secuencia de pasos es la siguiente:

1. Genera una muestra aleatoria de números desde una distribución binomial centrada en el parámetro *average_word_length*. Esta muestra corresponderá al largo de las palabras a generar.
2. Genera, para cada número de la muestra anterior, una palabra que tiene justamente ese largo. Toma los caracteres desde una distribución uniforme cuyo rango es el tamaño del alfabeto. Llamaremos a estos elementos la lista de palabras.
3. Genera una muestra de $\frac{n}{10}$ palabras no contenidas en la lista de palabras, donde n es el número de palabras en la lista de palabras. Los largos de cada palabra genera se escogen de manera uniforme desde 1 al máximo largo de palabra de la lista de palabras.
4. Se crea la estructura y se inserta una permutación aleatoria de toda la lista de palabras.
5. Se consulta a la estructura por la totalidad de la lista de palabras (por lo que sirve como prueba, además), debiendo encontrarlas todas.
6. Se consulta a la estructura por las $\frac{n}{10}$ palabras no insertadas. Debe fallar en encontrarlas todas.

Este procedimiento se repite para todos los tamaños 2^i con $i \in [10, 20]$, y $|\Sigma| \in \{2, 4, 6, 8, 10, 20, 26, 40, 60, 80, 94\}$, donde 94 corresponde al límite de la implementación actual, ya que esta se basa en la codificación *ASCII*. Se miden los tiempos de inserción y de búsqueda, tanto para búsquedas exitosas como *search misses*, y se separan los tiempos de las búsquedas respecto al largo de los patrones. Lamentablemente la implementación actual no genera palabras mucho más largas que aquellas contenidas en la muestra original, por lo que se considera que el parámetro m no está suficientemente estudiado en esta etapa.

3.2. Single book experiment

Corresponde a una versión modificada del experimento descrito en el enunciado, que consiste en:

1. Se transforma cada libro en un vector de *strings* que contiene las palabras del texto, considerando solo caracteres alfanuméricos y apóstrofos, los que se llevan a *lowercase*.
2. Para cada libro se toma una muestra de las $n = 2^i$ primeras palabras.
3. Se construye una muestra escogida aleatoriamente de $\frac{n}{10}$ palabras presentes en el texto y una muestra de $\frac{n}{10}$ palabras generadas aleatoriamente no presentes en el texto.
4. Se insertan una permutación de las n palabras y se consulta por las 2 listas de palabras del paso 2.

El principal objetivo de este experimento es estudiar el comportamiento de las estructuras frente a textos de lenguaje natural, los cuales tienen una estructura, a diferencia de las palabras armadas aleatoriamente, y presentan un alto nivel de repetición en las palabras utilizadas. Cuando se mencionan las primeras 2^i palabras, lo que sucede realmente es que se acorta el texto si es que $2^i < N$, donde N es el número de palabras totales del texto, y se expande el texto (se insertan al final del texto las palabras del principio del texto) si es que

N resulta ser menor. Al igual que en el experimento anterior, los tiempos de búsqueda se separan en el largo del patrón m. Para la realización de estos experimentos se utilizaron 16 textos, los cuales se incluyen en la carpeta *./text/single_books/* y se encuentran descritos en los archivos adicionales.

3.3. Similarity experiment

El último experimento calcula la medida de similitud entre los textos adjuntos, pero es fácil modificar el código de *experiments.cpp* para que calcule sobre otros textos, solo es necesario indicar el nombre y la carpeta de los textos de destino en el tercer apartado del *main*. Los pasos realizados son los siguientes:

1. Se transforma cada libro en un vector de *strings* que contiene las palabras del texto en *lowercase*, ignorando todo caracter que no sea alfanumérico o apóstrofos.

2. Para cada libro se toma una muestra de las $n = 2^i$ primeras palabras.

3. Se calcula la similitud entre el texto de tamaño $n = 2^i$ y todos los otros textos de tamaño 2^i.

Se miden los tiempos de inserción de las permutaciones de la entrada y del cálculo de la medida de similitud, lo que implica un recorrido sobre el diccionario combinado de ambos textos, sin repeticiones. Esta parte se enfoca, principalmente, en el aspecto de *multiset* de las estructuras, ya que se hace uso de su habilidad para almacenar los valores asociados a las llaves provenientes de 2 textos distintos. Esto, dada la implementación con vectores, es fácilmente extensible a n textos.

4. RESULTADOS

Los experimentos fueron realizados en un computador con procesador Phenom II 955, 32Gb de ram corriendo a 1333 Mhz y un disco SSD Crucial BX100 de 250Gb. El sistema operativo utilizado fue *Ubuntu* 14.04.3 y el compilador fue *g++* 4.8.4, con *boost* 1.54.

Los experimentos mostraron que la estructura más lenta es, por lejos, el árbol Patricia. Ternary Search Tree logra ser competitivo con la tabla de Hash, pero paga caro esto en espacio total utilizado, el cual incrementa demasiado con el tamaño del alfabeto.

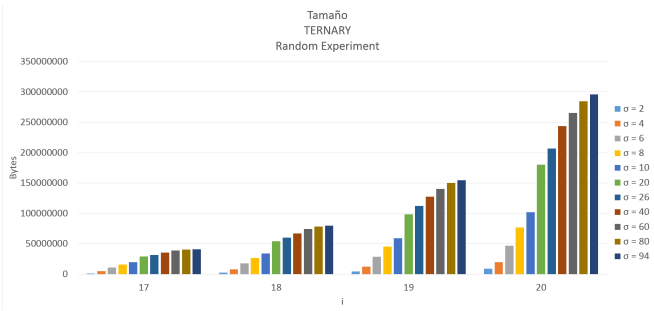

Fig. 1. Tamaño de estructura Ternary Search Tree.

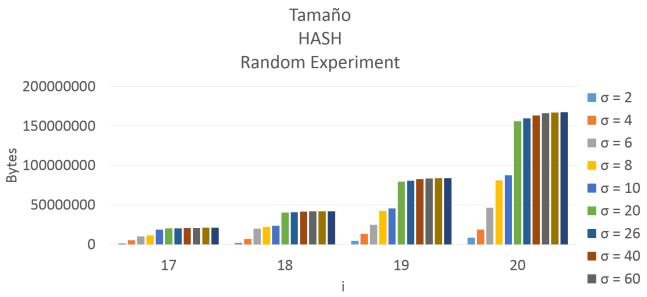

Fig. 2. Tamaño de estructura Hash Table.

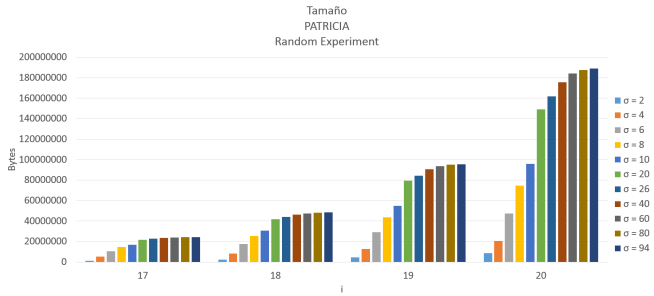

Fig. 3. Tamaño de estructura Patricia Tree.

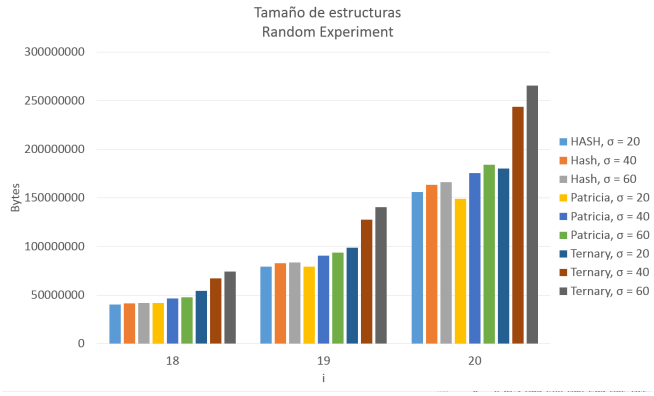

Fig. 4. Comparación de tamaños de las estructuras.

Podría pensarse que, si el árbol Patricia no es competitivo en tiempo, y tampoco logra superar en costos de espacio a la tabla de hash, entonces nunca sería preferible. Sin embargo, ambos árboles soportan operaciones que la tabla de Hash no puede soportar, como el cálculo de prefijos, y de sucesor o antecesor, por lo que el uso del árbol Patricia es una decisión que dependerá mucho de la aplicación para la que se utilice.

Los experimentos siguientes solo re-afirmaron las conclusiones que nos permite sacar la generación aleatoria de textos, con la salvedad de que los tamaños de todas las estructuras disminuyen gracias a la alta cantidad de repeticiones de las palabras. Las medidas de similitud para los textos escogidos son sorprendentemente altas. Algunos libros de A Song of Ice and Fire alcanzan hasta cerca de 0.71 de similitud con los libros de Malazan. Mientras mayor es el tamaño utilizado para comparar los textos, mayor es su similitud. Es interesante preguntarse, de los textos en su tamaño sin adulterar, ¿cuales textos de diferentes autores se parecen más según esta medida?

- ASOIAF 5 y Malazan 8
- ASOIAF 5 y Malazan 9
- ASOIAF 5 y Malazan 7
- ASOIAF 3 y Malazan 10
- ASOIAF 3 y Malazan 8

Todos los pares de textos anteriores alcanzan más de 0.7 de similitud. Los más parecidos a la Biblia son Malazan 7 y Malazan 5, que alcanzan un poco más de 0.5. Pero fuera de esta trivia, este experimento no aporta mucho más que reforzar la idea de que reinsertar los mismos patrones es más barato en espacio, aunque demuestra que se puede ahorrar cerca de un 30 % de espacio, lo que es bastante bueno por si mismo. El tiempo, si bien disminuye un poco, no es considerablemente menos: una diferencia cercana al 5 % para inserción y del 10 % para las búsquedas.

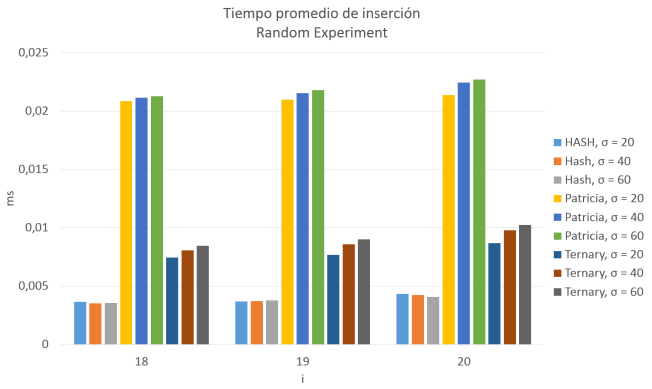

Fig. 5. Tiempo promedio de inserción.

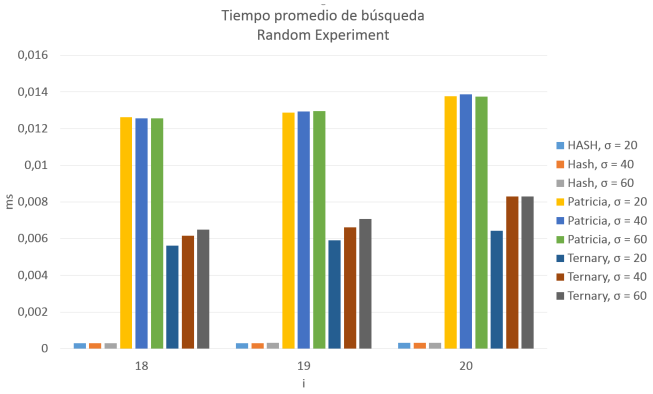

Fig. 6. Tiempo promedio de búsqueda de patrones contenidos en la estructura.

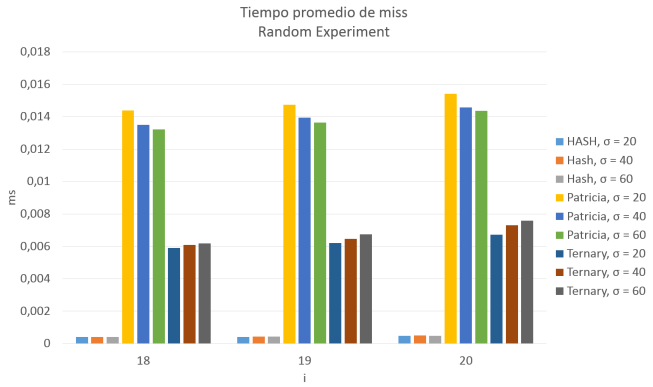

Fig. 7. Tiempo promedio de búsqueda para patrones no contenidos en la estructura.

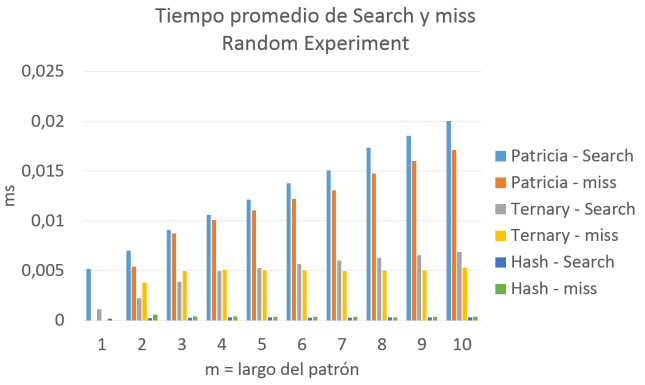

Fig. 8. Tiempos promedio de búsqueda para patrones contenidos y no contenidos en la estructura, separados por largo del patrón.

Es claro el costo en el largo del patrón para búsquedas, en el caso del Patricia Tree. Un comportamiento interesante es el que exhibe Ternary Search Tree, quien toma más tiem-

po en buscar patrones más largos, pero cuyo miss time se ve poco alterado por el largo del patrón, gracias a la aridad de los nodos puedes decidir rápidamente si una clave está o no en el árbol. Sigue siendo, sin embargo, la tabla de Hash la estructura más rápida bajo todos los casos.

5. REFERENCES

[1] http://en.cppreference.com/w/cpp/concept/Hash