UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA EN TEXTO MEDIANTE UN ÍNDICE COMPRIMIDO DE Q-GRAMAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

HERNÁN ENRIQUE ARROYO GARCÍA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CARDENAS
MAURICIO MARÍN CAIHUAN

SANTIAGO DE CHILE
ABRIL 2010

ESTE TRABAJO HA SIDO FINANCIADO EN PARTE POR EL PROYECTO
FONDECYT 1-080019

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: HERNÁN ARROYO G.
FECHA: 20/04/2010
PROF. GUIA: GONZALO NAVARRO B.

BÚSQUEDA EN TEXTO MEDIANTE UN ÍNDICE COMPRIMIDO DE Q-GRAMAS

La cantidad de datos disponibles crece de forma dramática cada día. Esto trae consigo la necesidad de poder manejar estos datos de forma adecuada, de manera de poder acceder a estos de forma eficiente y al mismo tiempo ahorrar espacio de almacenamiento. En particular, para manejar grandes cantidades de texto una herramienta clave son los índices de texto, y en el contexto de este trabajo los índices comprimidos, los cuales no sólo responden consultas de forma rápida sino que también almacenan sus datos y el texto en forma eficiente.

El objetivo general del presente trabajo es desarrollar un índice comprimido basado en listas de ocurrencias de los q-gramas del texto y comprimir este último. Se desea comparar la eficacia de este índice con los auto-índices ya desarrollados en el sitio Pizza&Chili.

Un índice invertido de q-gramas permite encontrar patrones en un texto. Para tal efecto las consultas se dividen en dos etapas. En la primera etapa se seleccionan las regiones del texto (llamadas bloques) donde ocurren todos los q-gramas del patrón y por lo tanto éste podría encontrarse. En la segunda etapa se verifica si efectivamente el patrón se encuentra en los bloques que fueron seleccionados.

Además es necesario almacenar el texto de forma independiente. En la implementación realizada se mantiene el texto dividido en bloques comprimidos, los cuales se almacenan en memoria secundaria. Esto permite utilizar menos espacio y acceder a los bloques individualmente.

Se implementaron diversos algoritmos para comprimir el índice y realizar consultas. Además se diseñaron y ejecutaron experimentos para medir el rendimiento de las distintas variantes obtenidas al combinar los diferentes algoritmos. En base a los resultados obtenidos se seleccionaron los algoritmos que presentaron mejor rendimiento tanto en velocidad como en niveles de compresión alcanzados.

De la misma forma se implementaron y midieron experimentalmente alternativas para comprimir y buscar en el texto.

Finalmente se comparó el rendimiento de las variantes seleccionadas del índice frente a los índices competitivos presentes en el sitio Pizza&Chili.

Los resultados indican que el índice tiene un rendimiento competitivo para búsquedas de patrones pequeños.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Gracias a los avances en la computación, la cantidad de información que nuestra sociedad produce y almacena crece de forma exponencial. Día a día se producen vastas cantidades de información de múltiples fuentes como transacciones financieras, revistas, imágenes y otras. Toda esta información se almacena de forma digital en computadores. Esto no sólo permite reducir el espacio físico necesario para almacenamiento, sino que también permite reproducir los datos con mucha facilidad.

El desafío que se presenta al tener grandes volúmenes de información, es poder manejarlos de forma eficiente y efectiva, de manera de poder encontrar y recuperar los datos que sean relevantes en un momento determinado.

Si se desea buscar o acceder a datos particulares dentro de una colección de gran tamaño, una búsqueda secuencial podría tardar tiempos muy largos. Por ejemplo el contenido de la web es de miles de terabytes. Una búsqueda secuencial tardaría días. La solución a este problema se encuentra en los índices.

Un índice se compone de estructuras y datos adicionales a la información original, y permiten realizar búsquedas en tiempos mucho menores que una búsqueda secuencial. Por ejemplo, para la Web se utilizan índices invertidos de palabras, los cuales almacenan listas que señalan las ocurrencias de cada palabra. De esta forma la búsqueda en un índice de este tipo se reduce a intersectar las listas de los términos buscados.

En el ejemplo anterior sería posible crear un índice invertido de q-gramas, en este índice las listas de ocurrencias no almacenan la ocurrencia de palabras, en cambio se señalan ocurrencias de sub secuencias de largo fijo del texto. Estas sub secuencias se conocen como q-gramas del texto.

Los índices no están limitados a listas de ocurrencias. Existen índices como los suffix array que se basan en sufijos. Un suffix array es un arreglo que contiene la posición donde comienza cada uno de los sufijos del texto al ordenarlos lexicográficamente. El suffix array

permite buscar de forma rápida en el texto mediante búsquedas binarias sobre las posiciones de los sufijos en el arreglo.

Debido a los grandes volúmenes de información que se manejan en ciertas áreas, es usual que una parte significativa de los datos se almacenen en memoria secundaria. La memoria primaria es más rápida que la secundaria por órdenes de magnitud. Peor aún, esta diferencia de velocidad ha aumentado en el tiempo. De ahí surge el interés de los índices basados en estructuras de datos compactas, las cuales permiten realizar las mismas operaciones que una estructura tradicional, pero utilizando menos espacio. Gracias a estas estructuras es posible realizar búsquedas de forma más rápida, y en ciertos casos prescindir del uso de memoria secundaria.

En el área de indexación de texto, existen índices compactos que permiten funcionalidades como búsqueda de patrones o acceso aleatorio a segmentos del texto en forma rápida. En algunos casos estas estructuras prescinden del texto original, por lo que se llaman auto-índices. Las tres familias principales de auto-índices son el FM-Index, el Compressed Suffix Array y el LZ-Index.

En el caso de los textos no separables en unidades, por ejemplo secuencias de ADN, los suffix arrays son el único tipo de índice que ofrecen un rendimiento aceptable. Cuando se trabaja con un texto muy grande no es posible utilizar un suffix array debido a sus requerimientos de espacio. Por otra parte, un suffix array comprimido permite obtener rápidamente la cantidad de ocurrencias de un patrón en el texto, pero no es competitivo frente a la versión tradicional para encontrar estas ocurrencias. Esto motiva la búsqueda de índices que utilicen menos espacio que un suffix array, pero que ofrezcan un rendimiento similar a los índices más rápidos.

Un tipo de índice que podría cumplir con estas condiciones es el índice invertido de q-gramas basado en bloques, ya que puede ser utilizado en lenguajes donde no existe una separación clara entre una palabra y otra. Para utilizar menos espacio que un suffix array el índice puede comprimirse. Actualmente existen índices de q-gramas comprimidos que compiten con suffix arrays comprimidos, pero no comprimen el texto original, por lo que ni su rendimiento ni su aplicación son comparables.

La implementación de un índice de q-gramas comprimido con texto comprimido no es trivial. Para que sea útil, debe reducir significativamente el tamaño del texto y encontrar patrones en el texto con suficiente rapidez. La implementación de un índice de este tipo plantea diversos problemas, los cuales requieren explorar múltiples soluciones y tomar decisiones de diseño. Por ejemplo, qué algoritmo utilizar para comprimir las listas de ocurrencias, o cómo resolver el problema de patrones que se encuentran entre dos bloques contiguos de forma eficiente. Otra dificultad que surge de la necesidad de almacenar el

texto comprimido es el acceso aleatorio a bloques. Para buscar de forma eficiente es necesario descomprimir cada bloque de forma independiente, pues resultaría impráctico tener que descomprimir el texto desde el comienzo. Sin embargo, comprimir cada bloque independientemente obtendría muy poca reducción de espacio.

1.1. Trabajo Realizado

Este trabajo se centró en la implementación de un índice de q-gramas basado en bloques, implementando diversas alternativas para cada componente, y en la comparación de su rendimiento tanto en tiempo para responder consultas como espacio requerido para almacenar sus estructuras.

La primera parte del trabajo consistió en implementar el índice, entregando distintas alternativas para codificar el índice y buscar en éste.

La segunda parte consistió en explorar e implementar distintas alternativas para almacenar el texto en forma de bloques comprimidos. Se implementó una versión modificada de $Re-Pair$ aproximado en disco, el cual permite comprimir el texto primero y luego generar los bloques. Además se exploró generar bloques para luego comprimirlos de forma independiente con *Gzip* o *Compress*.

Para buscar dentro del texto en el caso de $Re-Pair$ se descomprime el bloque completo y luego se utiliza Boyer-Moore-Horspool para buscar dentro del bloque. En el caso de *Gzip* y *Compress* se utiliza la herramienta *LZgrep*, la que permite buscar dentro de texto comprimido sin necesidad de descomprimirlo completamente.

Finalmente se realizaron experimentos con diferentes colecciones de texto obtenidas del sitio Pizza&Chili. Debido a la gran cantidad de parámetros y algoritmos que pueden utilizarse en el índice (y que afectan su rendimiento), se seleccionaron los mejores parámetros y algoritmos por etapas.

En la primera etapa se seleccionaron los algoritmos y parámetros que afectan solamente la búsqueda al interior del índice, de esta forma se lograron fijar algunos algoritmos y parámetros relacionados con la codificación y algoritmos de búsqueda dentro del índice.

En la segunda etapa se comparó el rendimiento de buscar dentro de bloques comprimidos con $Re-Pair$, *Compress* y *Gzip*.

En la tercera etapa se realizaron pruebas sobre distintas combinaciones de parámetros que afectan al índice en forma global, midiendo el rendimiento de éste para patrones de diferente largo.

Finalmente se comparó el rendimiento de las mejores alternativas obtenidas de las pruebas anteriores frente a otros índices comprimidos.

1.2. Estructura de la Memoria

En el capítulo 2 se tratan los conceptos necesarios para leer la memoria y se presenta el marco teórico en el que se desarrolla el trabajo realizado. Se detallan algoritmos y conceptos utilizados en el resto de la memoria, tratando temas como compresión de datos, búsqueda en texto e índices de texto.

En el capítulo 3 explica cómo se construye el índice implementado a partir de un texto, explicando las diferentes alternativas implementadas y detallando cada una de las decisiones de diseño tomadas en la implementación.

En el capítulo 4 se detalla el proceso de búsqueda hasta la selección de bloques de texto a revisar, explicando los algoritmos involucrados en cada uno de los tipos de búsqueda que se pueden presentar.

En el capítulo 5 se explica cómo se comprime el texto y se almacena en bloques, además se muestra cómo se busca en los bloques dependiendo de la estrategia utilizada para comprimir el texto.

En el capítulo 6 se presentan los experimentos realizados para seleccionar las versiones más competitivas del índice, y medir el rendimiento de éstas frente a índices de otros tipos.

Finalmente en el capítulo 7 se analizan los resultados obtenidos, presentando direcciones que podría tomar el trabajo futuro.

Capítulo 2

Conceptos Básicos

2.1. Compresión de Datos

La compresión de datos es el proceso de codificar información permitiendo representar los mismos datos utilizando menos bits que los utilizados originalmente.

En este trabajo se trata sólo con algoritmos de compresión sin pérdida de información, es decir que permiten recuperar los datos originales íntegramente a partir de los datos comprimidos.

Para recuperar los datos es necesario descomprimirlos, si se desea conocer el valor de un bit que se encuentra en la posición p de los datos, dependiendo del algoritmo de compresión utilizado, puede ser necesario tener que descomprimir primero los bits entre 0 y $p-1$.

También existen algoritmos que permiten comprimir localmente, lo que posibilita descomprimir parte de los datos sin tener que procesar el documento desde el comienzo.

El nivel de compresión que pueden alcanzar los datos depende de las propiedades de éste. Frecuentemente se utiliza la entropía empírica como medida de compresibilidad. Para un texto T de largo n, cuyos símbolos provienen de un alfabeto $\sum$ de tamaño σ, se define la entropía empírica de orden cero como:

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

La entropía empírica de orden cero indica el número mínimo de bits necesarios para representar T, si al comprimir cada símbolo del texto no se consideran los símbolos que le anteceden o preceden.

Frecuentemente los textos a comprimir presentan regularidades. Es posible explotar algunas de estas regularidades si consideramos el contexto en el que aparecen los símbolos. Por ejemplo, en un texto en español si aparecen los símbolos "*qu*" en este contexto es muy

probable que el símbolo siguiente sea "e" o "i". La entropía empírica de orden k es una medida de compresibilidad que considera estos contextos y se define como:

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{n} H_0(T^s)$$

Donde T^s es la subsecuencia de T formada por todos los símbolos que ocurren a continuación del contexto s en T.

La gran mayoría de los métodos de compresión se pueden clasificar en dos categorías: métodos estadísticos y métodos basados en diccionarios.

2.1.1. Métodos estadísticos

Los métodos estadísticos se basan en dos componentes: un modelador y un codificador. El modelador genera una distribución de probabilidades de los símbolos presentes en el texto. El codificador recibe estas probabilidades y las utiliza para asignar códigos a cada símbolo. De esta forma un algoritmo de codificación no está limitado a un modelo en particular. Los modelos se pueden clasificar en tres categorías: estáticos, semiadaptativos y adaptativos:

Los modelos estáticos fijan la distribución de probabilidades antes de procesar el texto. Éste es el modelo utilizado en las comunicaciones telegráficas, donde tanto el emisor como el receptor deben conocer de antemano el modelo y la codificación que se utilizan.

Al igual que en los modelos estáticos, los modelos semiadaptativos fijan la distribución de probabilidades antes de codificar el texto, pero a diferencia de ellos la distribución se genera procesando el texto a comprimir. Esto permite tener un modelo que se ajuste mejor al texto, su desventaja radica en la necesidad de almacenar la distribución de probabilidades utilizada junto al texto para permitir su decodificación.

Los modelos adaptativos cambian la distribución de probabilidades a medida que se codifica el texto, de esta forma el modelo se ajusta al texto a medida que se codifica lo que elimina la necesidad de almacenar la distribución de probabilidades utilizada.

Modelos de orden k Los modelos de orden k consideran un contexto de tamaño k para codificar cada símbolo. Por ejemplo un modelo de orden 1 consideraría como contexto el último símbolo codificado para estimar la probabilidad del siguiente símbolo. Si no se considera el contexto se considera que el modelo es de orden cero.

Huffman Coding Huffman Coding es un algoritmo de codificación basado en símbolos. A continuación se presenta su descripción:

1. Listar todos los símbolos posibles y sus probabilidades.

2. Encontrar los dos símbolos con las probabilidades de ocurrencia menor.

3. Reemplazar estos dos símbolos por un conjunto que los contenga a ambos, la probabilidad del conjunto será igual a la suma de las probabilidades de los símbolos que contiene.

4. Repetir hasta que sólo quede un conjunto.

Este procedimiento genera una estructura de conjuntos donde cada conjunto contiene dos elementos. Para realizar la codificación la estructura generada será vista como un árbol binario. La codificación de los mensajes se obtiene recorriendo el árbol desde la raíz hasta el mensaje, generando un 0 si se sigue una rama izquierda y un 1 si se sigue una rama derecha. La compresión alcanzada por Huffman coding es menos de $n(H_0(T) + 1)$ bits.

Arithmetic Coding Arithmetic coding es un algoritmo de codificación que transforma un texto en un número real, utilizando la precisión necesaria como para recuperar el texto sin pérdida de información. Antes de comenzar a codificar el mensaje tiene como rango el intervalo $[0, 1)$. Cada símbolo del mensaje tiene un intervalo asociado en cada paso de la codificación. El orden de los intervalos está asociado al orden lexicográfico de los símbolos. El tamaño del intervalo está asociado a la probabilidad de cada símbolo.

Por ejemplo si el menor símbolo (lexicográficamente) de un texto tiene probabilidad $\frac{1}{2}$, inicialmente el intervalo asociado a este símbolo sería $[0, 0{,}5)$, y el intervalo $[0{,}5, 1)$ sería asignado al resto de los símbolos del texto.

En cada iteración cambia el intervalo y se vuelven a asignar intervalos dentro del nuevo intervalo dependiendo de las probabilidades del modelo, tal como al comenzar.

El resultado final de este algoritmo puede ser cualquier número dentro del intervalo obtenido al codificar el último símbolo del texto.

Este algoritmo logra una compresión de $nH_0(T) + 2$ bits

Prediction by Partial Matching Prediction by partial matching (PPM) utiliza modelos de contextos finitos. En vez de utilizar un modelo de orden k, con k fijo, PPM utiliza diferentes órdenes de contexto, dependiendo de qué secuencias de símbolos se hayan codificado previamente. Cuando codifica un símbolo, se busca el contexto más grande dentro del modelo, si no existían ocurrencias previas del contexto, se intenta con un orden menor. Una vez que se ha encontrado un contexto de un orden adecuado que sí ha ocurrido previamente, se revisa el valor del símbolo a codificar. El caso general es idéntico al de otros modelos de codificación basados en símbolos. Cuando el símbolo obtenido no ha sido identificado previamente en este contexto, se guarda un caracter especial que indica que el símbolo será codificado utilizando

un contexto de orden inferior. Este proceso continúa hasta que se cuente con un modelo de orden adecuado o se llegue a un modelo de orden 0, donde se codificará el símbolo de la misma forma que en un modelo de orden 0.

2.1.2. Métodos basados en diccionarios

Los métodos basados en diccionarios trabajan reemplazando secuencias de texto (por ejemplo palabras) por referencias a un diccionario. Las referencias que se encuentran en el diccionario también pueden contener recursivamente referencias a otras unidades de texto del diccionario.

Ziv-Lempel La codificación de Ziv-Lempel es un método de compresión adaptivo basado en diccionario. La esencia del método es la de reemplazar ocurrencias de secuencias de caracteres por referencias a una ocurrencia previa de la misma secuencia. No existe una descripción única en forma de algoritmo de esta codificación. Esto se debe en parte importante a que la descripción original es teórica, y no especifica un límite con respecto a cuánta distancia puede haber entre la secuencia que se desea reemplazar y la referencia que tomará su lugar. Otro factor importante es si se imponen limitaciones sobre el conjunto de secuencias que pueden ser apuntadas o si se imponen restricciones sobre la cantidad de secuencias a las que se tienen referencias (tamaño del diccionario). Estos factores afectan la velocidad de codificación y la compresión que se obtiene, lo que ha dado paso a una familia de algoritmos basados en esta codificación. Los métodos de este tipo reducen el espacio que utiliza un texto T de largo n a $nH_k(T) + O(\frac{k \log |\Sigma|}{\log_{|\Sigma|} n})$.

LZ77 En este esquema las subsecuencias se reemplazan por punteros a ocurrencias previas siempre que la distancia entre el texto y el puntero no sea mayor a un parámetro fijo. Las subsecuencias a reemplazar tienen un tamaño fijo, dado por un parámetro F. A medida que el algoritmo codifica el texto, la parte del texto que puede ser usada para apuntar secuencias que se repiten va cambiando. Se denota el tamaño de la secuencia junto con el tamaño máximo de sub secuencia como N. De esta forma, el diccionario para buscar ocurrencias tiene un tamaño fijo $N - F$, y la distancia entre el puntero y la secuencia que apunta también queda acotada por este valor.

Durante la codificación, el algoritmo considera todos los substrings que se comienzan dentro de los anteriores $N - F$ caracteres codificados. Intenta codificar los siguientes F caracteres considerando el diccionario anteriormente descrito. La sub secuencia puede contener parte de los caracteres a codificar, pero no puede ser la misma secuencia.

Los punteros almacenan 3 valores $< i, j, a >$: i denota la distancia desde el inicio de la secuencia a codificar hasta el comienzo de la sub secuencia apuntada, j es el largo de la sub secuencia que coincide y a corresponde al primer carácter del texto que no coincide con la secuencia a reemplazar. Una vez que se codifica una secuencia, el algoritmo avanza $j + 1$ caracteres y continúa hasta llegar al fin del texto.

LZ78 Este algoritmo en cada paso busca la secuencia codificada más larga que coincida con la parte del texto que se está codificando actualmente y agrega al diccionario una referencia a esta secuencia junto con el carácter siguiente del texto que no coincidió con la secuencia que se está reemplazando.

Debido a que cada vez que se codifica una secuencia ésta se agrega al diccionario, el diccionario se compone de todas las secuencias previamente codificadas.

Para poder realizar la búsqueda de secuencias de forma eficiente, se utiliza un trie. Esto no solo facilita la búsqueda de secuencias, sino que también la inserción, ya que al encontrar la secuencia en el trie, también se ha encontrado el lugar donde se debe insertar la nueva secuencia en el diccionario.

2.1.3. Métodos basados en gramáticas

En algunos casos la gramática que genera un texto resulta significativamente más pequeña que el texto mismo. Mientras más simple es la gramática, mejor es la compresión que se podría obtener. El problema radica en que encontrar la gramática óptima es un problema NP-Completo. Por esta razón se utilizan gramáticas aproximadas, las cuales pueden ser usadas para comprimir de forma aceptable.

Re-Pair Re-Pair es un algoritmo de compresión basado en frases (secuencias de caracteres) que permite descomprimir rápidamente y de forma local. Funciona identificando los pares de caracteres más frecuentes en el texto y reemplazando cada ocurrencia de este par por un nuevo carácter que no pertenezca a los caracteres del texto. Cuando se identifica el par ab de mayor frecuencia, se guarda en un diccionario s, a, b donde s indica el nuevo carácter que reemplazará todos los pares ab. Una vez que se ha realizado el reemplazo, se vuelve a buscar el par más frecuente hasta que todos los pares aparezcan sólo 1 vez en el texto.

Para realizar la compresión de forma más rápida, es posible realizar varios reemplazos de pares al mismo tiempo. Para hacer esto sólo se debe tener cuidado de no hacer un reemplazo que impida realizar otro reemplazo de un par que tenga mayor frecuencia. Por ejemplo si se tiene el texto "$aacaa$", donde aa tiene mayor frecuencia que ac, si se reemplaza primero el

par aa por b, se obtendría "bcb", mientras que si se reemplaza ac por b se obtendría el texto "$abaa$".

Re-Pair aproximado en disco Este algoritmo diseñado por Francisco Claude es una variante de Re-Pair que mantiene el texto en memoria secundaria, lo que permite comprimir textos de gran tamaño.

El algoritmo de compresión recorre el texto sucesivas veces para comprimirlo. En cada recorrido se identifican k pares a reemplazar y luego se realizan estos reemplazos. De esta forma k es un parámetro que aumenta o disminuye el tiempo y el nivel de compresión. Cada recorrido se divide en las siguientes etapas:

Contar frecuencias Se recorre el texto secuencialmente y se insertan los pares encontrados en una tabla de hash. Si la tabla de hash sobrepasa un factor de carga aceptable, no se insertan más pares, pero sí se continúa aumentando la frecuencia de los pares ya insertados en cada ocurrencia futura.

Selección de k pares Se extraen los k pares que presentan mayor frecuencia en la tabla de hash.

Reemplazo de pares En esta etapa se reemplazan simultáneamente los k pares identificados con mayor frecuencia. Es importante considerar que algunos reemplazos son excluyentes entre sí ya que se invalidan entre ellos. Debido a esto algunos pares podrían tener muchas ocurrencias invalidadas, haciendo innecesario su reemplazo.

Teniendo en cuenta las consideraciones anteriores, cuando se encuentra un par en el texto éste no se reemplaza inmediatamente.

Al leer secuencialmente el texto cada par se busca en la tabla de hash. Si el par no se encuentra, entonces ésta es la primera ocurrencia del par. Se verifica si la página de disco ya se encontraba en memoria, de no ser así se guarda la página en una tabla de hash para las páginas de disco (si el par se encuentra en dos páginas, se almacenan ambas), además se asocia en la tabla de hash de pares la posición de la primera ocurrencia del par.

Si la posición de la primera ocurrencia del par es válida y el par se encuentra en la tabla de hash, se verifica la página en la tabla de hash de páginas cacheadas. Existen dos posibilidades:

Si la ocurrencia previa es válida se reemplazan ambas ocurrencias. Si no existen otros pares asociados a la página cacheada ésta se escribe en disco. En la tabla de hash de pares se cambia el valor de la posición de la primera ocurrencia por un valor especial *proceed* para indicar que las próximas ocurrencias deben reemplazarse inmediatamente.

Si la ocurrencia previa es inválida, se verifica si existen otros pares asociados a la página cacheada. De no ser así se verifica si la página ha sido modificada para guardarla en disco. Finalmente se guarda la posición y pagina del par actual como primera ocurrencia del par en las tablas de hash de páginas y de pares.

Si la posición de la primera ocurrencia del par tiene el valor especial *proceed*, entonces el par actual se reemplaza inmediatamente.

Cuando se realiza un reemplazo se cambia el primer símbolo del par por un nuevo símbolo y el segundo símbolo del par por un símbolo especial que representa un espacio vacío.

En todo momento no se tienen más de $2k$ páginas de disco en memoria, además se tienen como máximo $2k$ pares en la tabla de hash de pares.

Al final de la etapa se guardan en disco todas las páginas que aún se encuentren en memoria y han sido modificadas.

Compactación Se lee el texto secuencialmente y se eliminan los símbolos vacios, de esta forma disminuye el tamaño del texto luego de realizar los reemplazos.

2.1.4. Compresión de secuencias

Cuando los datos que se desean comprimir consisten en secuencias de números no acotados, resulta útil utilizar una técnica de codificación para alfabetos infinitos. Estas técnicas transforman números en secuencias de bits de largo variable. A diferencia de otras codificaciones como la de Huffman, donde el largo de cada código está dado por la probabilidad de ocurrencia del símbolo a codificar, en estas técnicas el largo de cada código está relacionado con el tamaño del número a codificar. Es decir generan códigos pequeños para valores pequeños y códigos más largos para valores más grandes.

En este trabajo se utilizan las codificaciones Gamma, Delta, Rice, Simple9 y Simple16.

Codificación unaria La codificación unaria representa un número n como $1^{n-1}0$. El 0 final sirve para identificar el término de la codificación. Por ejemplo $n = 4$ tiene como codificación unaria 1110. Esta codificación es utilizada en otras codificaciones como Gamma.

Gamma La codificación Gamma de un número entero n es la concatenación del largo de su representación binaria en unario, seguido por su representación binaria omitiendo su bit más significativo. Por ejemplo el número 7, cuya representación binaria está dada por 111, tiene como codificación Gamma 11011. Representar un entero n utiliza $2\lfloor \log n \rfloor + 1$ bits, $\lfloor \log n \rfloor + 1$ son usados para representar n en unario seguido de $\lfloor \log n \rfloor$ bits para representar el número en binario menos su bit más significativo.

Delta Se aplica la codificación Gamma, y luego se vuelve a aplicar codificación Gamma sobre la parte unaria. Esta codificación utiliza $1 + 2\lfloor \log \log n \rfloor + \lfloor \log n \rfloor$ bits

Rice Esta codificación requiere de un parámetro adicional b, potencia de 2, el cual se obtiene generalmente redondeando hacia abajo a una potencia de 2 el valor promedio de los elementos de la secuencia a codificar. Para codificar n se calcula $q = \lfloor (n-1)/b \rfloor$ y luego $r = n - qb - 1$, finalmente se escribe q en unario seguido de r en binario utilizando $\log_2 b$ bits .

Simple9 Esta codificación tiene como propiedad ser alineada en bytes. Funciona empaquetando la mayor cantidad de números posibles en grupos de 32 bits, de estos 32 bits se usan 28 bits para almacenar los datos y 4 bits para describir cómo están agrupados de acuerdo a los siguientes casos:

- 1 número de 28 bits
- 2 números de 14 bits
- 3 números de 9 bits (se desperdicia 1 bit)
- 4 números de 7 bits
- 5 números de 5 bits (se desperdician 3 bits)
- 7 números de 4 bits
- 9 números de 3 bits (se desperdicia 1 bit)
- 14 números de 2 bits
- 28 números de 1 bit

Simple16 Esta codificación es muy similar a Simple9. La diferencia radica en la forma en que se agrupan los números. Esta agrupación permite aprovechar algunos bits que se pierden en algunos casos en Simple9. Esto permite utilizar ligeramente menos espacio a cambio de una decodificación ligeramente menos rápida.

n	$Gamma$	$Delta$	$Rice(b=4)$
1	0	0	000
2	100	1000	001
3	101	1001	010
4	11000	10100	011
5	11001	10101	1000

Cuadro 2.1: Codificaciones Gamma, Delta y Rice para enteros entre 1 y 5.

2.2. Rolling Hash

Rolling hash es una familia de funciones, donde los valores sobre los que se calcula la función se obtienen desde una ventana que se mueve a lo largo de una entrada.

El propósito de estas funciones es obtener el valor del hash para la ventana siguiente de forma rápida. Para esto se necesita la última entrada hasheada, el valor de su hash y la nueva entrada a hashear.

Una de estas funciones de hash es utilizada en el algoritmo de búsqueda en texto de Rabin-Karp. Esta función se define para una constante a, y una entrada de caracteres $c_1, c_2, \ldots, c_q$ como:

$H(c_1, c_2, .., c_q) = c_1 a^{q-1} + c_2 a^{q-2} + \cdots + c_q a^0 = h$

Es posible calcular $H(c_2, c_3, \ldots, c_{q+1})$ en tiempo constante si se tiene el resultado de $H(c_1, \ldots c_q)$. Esto se hace restando el primer término, multiplicando el resto de la expresión por a y finalmente agregando el término c_{q+1}:

$H(c_1, c_2, .., c_q) - c_1 a^{q-1} = c_1 a^{q-1} + c_2 a^{q-2} + \cdots + c_q a^0 - c_1 a^{q-1}$

$(H(c_1, c_2, .., c_q) - c_1 a^{q-1})a = (c_2 a^{q-2} + \cdots + c_q a^0) * a = c_2 a^{q-1} + \cdots + c_q a^1$

$(H(c_1, c_2, .., c_q) - c_1 a^{q-1})a + c_{q+1} = c_2 a^{q-1} + \cdots + c_q a^1 + c_{q+1} =$

$c_2 a^{q-1} + \cdots + c_q a^1 + c_{q+1} a^0 = H(c_2, ..., c_{q+1})$

Para calcular el valor de la función sobre la nueva entrada se necesita realizar una resta, una multiplicación y una suma. Esto es independiente del tamaño de la entrada sobre la que se debe calcular la función. Luego una vez que se ha calculado el valor de la función de hash para la primera entrada, obtener los valores de las siguientes entradas sólo costará $O(1)$.

Calcular el valor para la primera entrada cuesta tiempo $O(q)$ ya que se realizan 2 multiplicaciones por cada término. Esto se hace calculando los valores sucesivos de $a^0, a^1, \ldots . a^{q-1}$. Cada uno de estos términos se multiplicará por un carácter de la entrada y finalmente se suma el total.

2.3. Búsqueda en Texto

El problema de búsqueda en texto o string matching consiste en encontrar las ocurrencias de un patrón dentro de un texto.

Formalmente, dado un patrón $P = p_1 \ldots p_m$ y un texto $T = t_1..t_u$, ambos secuencias de símbolos sobre un alfabeto Σ de tamaño σ, encontrar todas las ocurrencias de P en T consiste en encontrar el conjunto $\{|x|\,, T = xPy\}$. Existen diversos algoritmos que resuelven este problema. Algunos de estos algoritmos no necesitan inspeccionar todos los caracteres del texto dependiendo de las características del patrón a buscar. Un ejemplo es el algoritmo

Boyer-Moore.

Boyer-Moore Este algoritmo compara un patrón P de largo m con un texto T de largo n recorriendo el texto de forma secuencial y el patrón de derecha a izquierda. Este algoritmo preprocesa el patrón a buscar, lo que en algunos casos permite evitar comparar algunos caracteres del texto. Por ejemplo si se compara el ultimo símbolo del patrón con el texto y el símbolo encontrado en el texto no está presente en el patrón, entonces es posible avanzar m caracteres en el texto. Si por ejemplo se tiene el texto "*abbadabacba*" y el patron "*babac*", el algoritmo comienza comparando P_m, que corresponde al último símbolo del patrón con el símbolo $T_m = d$, como d no se encuentra en P la próxima comparación se realiza entre P_my T_{m+m}, la figura 2.1 ilustra esta situación.

1	2	3	4	5	6	7	8	9	10	11
a	b	b	a	**d**	a	b	a	c	b	a
b	a	b	a	**c**						
					b	a	b	a	c	

Figura 2.1: Ejemplo de búsqueda usando Boyer-Moore.

Para determinar cuánto se debe avanzar en el texto se utilizan 2 heurísticas:

Si el símbolo del texto no coincide con el símbolo con el símbolo del patrón con el que está siendo comparado, pero sí se encuentra en otro lugar dentro del patrón, es posible avanzar en el texto hasta que la ocurrencia de este símbolo en el patrón se encuentre alineada a la ocurrencia del símbolo en el texto. La figura 2.2 muestra este caso

1	2	3	4	5	6	7	8	9	10	11
a	b	b	a	**b**	a	b	a	c	b	a
b	a	b	a	**c**						
		b	a	b	a	c				

Figura 2.2: Ejemplo de heurística de búsqueda utilizada por Boyer-Moore.

La heurística anterior no puede aplicarse si todas las ocurrencias del símbolo del texto se encuentran a continuación del símbolo que se compara actualmente en el patrón. Alinear el patrón con el texto en este símbolo implicaría mover el patrón hacia la izquierda. En esta situación podría moverse el patrón en una posición a la derecha. Sin embargo, una mejor alternativa consiste en desplazar el patrón lo máximo posible dada la porción de texto que coincide actualmente con el patrón. Esta heurística se basa en encontrar lugares en el patrón donde se repita el sufijo del patrón que coincide actualmente con el texto, de no repetirse este sufijo dentro del patrón es posible avanzar m símbolos. La figura 2.3 presenta un ejemplo de esta heurística.

1	2	3	4	5	6	7	8	9	10	11
a	**b**	b	a	b	a	b	a	c	b	a
c	**a**	b	a	b						
	c	a	b	a	b	c				

Figura 2.3: Ejemplo de heurística de sufijos utilizada por Boyer-Moore.

Utilizando estas heurísticas se construye una tabla t en base a la primera heurística, y una tabla s en base a la heurística de sufijos. Estas tablas indican el desplazamiento que puede realizar el patrón en el texto de diferir en algún símbolo.

Durante una búsqueda si el patrón coincide completamente con el texto, entonces se reporta la ocurrencia y se revisa en la tabla s cuanto se debe avanzar en el texto. Si el patrón difiere del texto en algún símbolo se revisan las dos tablas y se produce un desplazamiento igual al mayor valor reportado por ellas.

Para un texto de largo n en el peor caso este algoritmo toma tiempo $O(n)$ al realizar búsquedas.

Boyer-Moore-Horspool Este algoritmo es una versión simplificada del algoritmo Boyer-Moore utilizando solamenta la tabla t. Para un patrón de largo m y un texto de largo n y alfabeto de tamaño σ tiene una complejidad de tiempo promedio de $O(min(m, \sigma))$ y peor caso $O(mn)$.

2.3.1. Búsqueda en texto comprimido

Un problema frecuente es el de buscar un patrón en un texto que se encuentra comprimido. El problema de realizar string matching en un texto comprimido podría resolverse descomprimiendo el texto y luego utilizando un algoritmo de string matching tradicional. Resulta claro que si se pudiera realizar la búsqueda sin tener que descomprimir el texto, o tener que descomprimir sólo una parte de éste, la búsqueda sería más rápida comparada con el enfoque anterior.

Por ejemplo, cuando el texto se comprime utilizando alguna variante de Ziv-Lempel como LZ77, algunos caracteres del texto se encuentran explícitos en el texto comprimido, es decir no son referencias a ocurrencias pasadas de secuencias de caracteres. Si un algoritmo como Boyer-Moore comienza comparando el patrón con caracteres explícitos del texto, cuando estos caracteres no coinciden con el patrón, en algunas ocasiones al avanzar en el texto podrían omitirse comparaciones con algunos caracteres del texto que no se encuentren explícitos, ahorrando de esta forma el trabajo de descomprimir.

2.4. Índices

Un índice es una estructura de datos que permite acceder de forma rápida a las ocurrencias de un patrón cualquiera en un texto. Es importante tener en cuenta que los índices se obtienen preprocesando el texto antes de realizar la búsqueda. Esto impone como requisito contar con el texto y tiempo suficiente para procesarlo.

Algunos índices permiten buscar patrones aproximados, es decir permiten realizar búsquedas aún si existen errores en el patrón a buscar o en el texto.

Los mecanismos de indexación se pueden dividir en dos grandes categorías: índices de recuperación de palabras e índices de recuperación de secuencias. Los índices de recuperación de palabras están orientados a los lenguajes naturales. En estos índices los términos a consultar son palabras. En el caso de que permitan realizar búsquedas aproximadas, permiten recuperar cada palabra cuya distancia de edición con respecto al patrón sea a lo más k. La distancia de edición entre dos strings A, B denominada $d(A, B)$ se define como el número mínimo de reemplazos, eliminaciones o adiciones de caracteres necesarios para transformar A en B o viceversa. Esto presenta una gran limitación, pues si se busca un patrón que está contenido dentro de una palabra que se encuentra en el índice, esta palabra no siempre se recuperará. Por ejemplo, si se desea utilizar un índice que permita un error para un corrector ortográfico, al revisar el término "aller" ofrecerá sugerencias como "taller", pero la palabra "ayer" no será sugerida. Los índices de recuperación de secuencias están orientados a aplicaciones donde no se maneja un lenguaje natural, estos lenguajes no son separables en palabras o unidades, por ejemplo secuencias de ADN. También son útiles cuando el concepto de palabra es difícil de definir (como en lenguajes orientales como el chino, o lenguajes aglutinantes como el alemán).

2.4.0.1. Q-grama

Un q-grama es una secuencia de caracteres de largo q.

En la figura 2.4 se muestran los q-gramas, con $q = 3$, para el texto "textext".

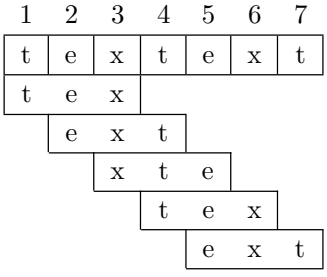

Figura 2.4: Q-gramas de un texto.

Como se aprecia en la figura, para el ejemplo existen 3 q-gramas diferentes: "*tex*", "*ext*" y "*xte*". Además el q-grama "*tex*" se encuentra en las posiciones 1 y 4, y el q-grama "*ext*" se encuentra en las posiciones 2 y 5.

2.4.1. Índices invertidos

Un índice invertido es una estructura de datos muy similar al índice de un libro. Los índices invertidos manejan un vocabulario, el vocabulario corresponde a las unidades de texto que contiene el índice. Dependiendo del vocabulario permiten buscar palabras o secuencias. El vocabulario puede corresponder por ejemplo a palabras del texto, o los q-gramas del texto. Dentro del índice se agregan la unidad de texto junto con todas las posiciones donde ocurre ésta en el texto. Existen muchas variaciones, por ejemplo si se desea utilizar menos espacio, se puede dividir el texto en bloques y guardar en el índice referencias a los bloques donde se encuentran estas unidades en vez de la posición exacta. Esto permite tener un índice de menor tamaño a cambio de encarecer la búsqueda de las ocurrencias.

									1	1	1	1	1	1	1	1	1	1	2
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
a	l	a	b	a	r		a		l	a		a	l	a	b	a	r	d	a

Figura 2.5: Texto de ejemplo.

El índice invertido de palabras del texto de la figura 2.1 contendría las palabras "alabar", "a", "la", "alabarda". Cada uno de esos términos tiene asociado las ocurrencias 1, 8, 10, 13 respectivamente.

2.4.1.1. Índice invertido de q-gramas

Es posible generar un índice invertido donde las unidades de texto utilizadas sean q-gramas. Para generar este índice a partir de un texto T se debe fijar el tamaño q de los q-gramas. Luego se agregan todos los distintos q-gramas del texto en el índice (el cual se mantiene ordenado lexicográficamente). A cada q-grama se le asocia una lista que contiene en forma ascendente las posiciones en el texto donde aparece.

Por ejemplo, en el texto de la figura 2.4, el q-grama "*ext*" tendría asociada la lista de ocurrencias $\{2, 5\}$.

Con la estructura básica de índice descrita anteriormente, es posible encontrar todas las ocurrencias de un q-grama en un texto buscando el q-grama en el índice y recuperando la lista de ocurrencias asociada.

Si se desea buscar un patrón P cuyo largo m sea mayor a q es necesario obtener una cobertura del patrón. La cobertura del patrón corresponde a un conjunto de q-gramas contenidos en el patrón, que lo cubren completamente. Cada ocurrencia de un q-grama del patrón en el índice es candidato a ocurrencia del patrón en el texto. Para verificar o descartar tal ocurrencia se pueden intersectar las posiciones de las ocurrencias de los diferentes q-gramas de la cobertura, o verificar con un algoritmo online si el patrón se encuentra en el texto. Es muy importante notar que no es necesario buscar todos los q-gramas del patrón, ya que algunos de estos son redundantes. Por ejemplo si $q = 2$ y $P = abcd$, bastaría con buscar los q-gramas ab y cd, resultando innecesario buscar el q-grama bc en el índice.

En general, las listas de ocurrencias del índice se mantienen comprimidas. Según el tipo de compresión utilizado se puede perder la capacidad de acceder en forma aleatoria a los elementos de las listas. Algunos algoritmos de intersección no acceden a los elementos de las listas en forma secuencial. Para que estos algoritmos puedan intersectar las listas de ocurrencias comprimidas se agregan muestras a la lista, cada muestra apunta a un elemento en la lista y contiene información que permite descomprimir elementos a partir de la posición que apuntada. De esta forma si se desea acceder a un elemento en la lista, se busca la muestra más cercana y se descomprime secuencialmente a partir de ella hasta llegar al elemento deseado.

2.4.1.2. Índice invertido de q-gramas basado en bloques

Si se desea utilizar menos espacio, es posible dividir el texto en bloques y considerar que todos los q-gramas que comienzan en un bloque se encuentran en éste. Dentro del índice se almacenan los bloques donde comienza cada q-grama. Esto permite ahorrar espacio ya que sólo se guarda una ocurrencia de un q-grama dentro de un bloque, además se necesita menos espacio para indicar en qué bloque se encuentra el q-grama en comparación al espacio necesario para almacenar su posición exacta dentro del texto.

Que un bloque contenga todos los q-gramas de la cobertura de un patrón no garantiza que el patrón se encuentre en el bloque, pues no se cuenta con información adicional sobre la posición que tienen estos q-gramas dentro del bloque. Es posible que se encuentren en un orden distinto al del patrón. Adicionalmente, es posible que al buscar un patrón en el índice, se encuentren algunos q-gramas de la cobertura del patrón en un bloque y otra parte de estos

en un bloque consecutivo, esta situación tampoco garantiza una ocurrencia del patrón, pero ocurrirá si un patrón comienza en un bloque y termina en otro. Se vuelve necesario buscar el patrón dentro de cada bloque candidato.

2.4.1.3. Índice invertido de q-samples

Los q-samples son muy similares a los q-gramas. Corresponden a substrings disjuntos de un texto, de tamaño fijo q y separados por intervalos regulares, dicho de otra forma, corresponden a muestras de los q-gramas de un texto. Los q-samples son utilizados de la misma forma que los q-gramas para crear índices invertidos.

Es claro que como los q-samples son muestras de los q-gramas, la ventaja de un índice de q-samples es su tamaño. Las desventajas se relacionan con la búsqueda de patrones. Para realizar una búsqueda es necesario que el patrón tenga un largo tal que algún q-grama del patrón se encuentre en el índice.

2.4.2. Auto-índices

Un auto-índice es un índice que permite recuperar el texto original íntegramente. La ventaja de los auto-índices es que permiten descartar el texto original, reduciendo significativamente los requerimientos de espacio para la estructura.

2.4.2.1. Auto-índices de sufijos

Suffix Trie Un suffix trie es una estructura de datos que contiene todos los sufijos de un texto T a partir del que fue construido. Internamente es un trie, que contiene caracteres de los sufijos de T en sus aristas. Los suffix tries están construidos de forma tal que para cada sufijo en T existe un único camino que va desde la raíz del suffix trie hasta una hoja de éste.

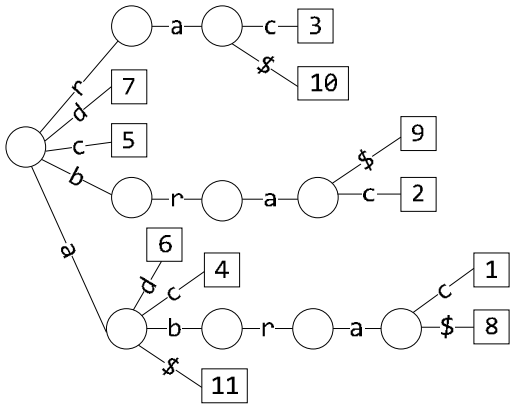

Figura 2.6: Suffix Trie de “abracadabra$”.

Dado un suffix trie de un texto T de largo n y un patrón P de largo m, cada ocurrencia de P en T es un substring de T, es decir un prefijo de un sufijo de T. Si se siguen los caracteres de P en el suffix trie, se llega a un nodo que tiene como hijos todas las ocurrencias de P en T (si no se llega a un nodo entonces P no está en T). Esto permite contar las *occ* ocurrencias de P en T en tiempo $O(m)$ si se almacena el número de hojas que descienden del nodo encontrado. También permite encontrar todas las *occ* ocurrencias de P en T en tiempo $O(m + occ)$ al recorrer el subárbol del nodo encontrado.

El suffix trie tiene $O(n^2)$ nodos, esto significa que su tamaño es muy grande en comparación al tamaño del texto, debido a esto el suffix trie no resulta práctico para textos de gran volumen. El suffix tree es una estructura similar que asegura un tamaño lineal con respecto al texto.

Suffix Tree El suffix tree de un texto T es un suffix trie de T, donde cada camino unario ha sido convertido en una única arista. Las etiquetas de estas aristas se obtienen al concatenar las etiquetas de las aristas que las formaron.

Es posible construir un suffix tree utilizando$O(n \log n)$ bits. Para tal efecto se representan las etiquetas de las aristas con punteros al texto.

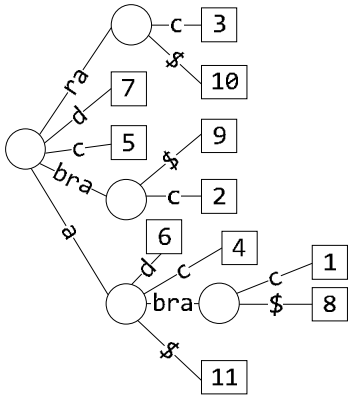

Figura 2.7: Suffix Tree de “abracadabra$”.

La búsqueda de patrones es casi idéntica a la búsqueda en un suffix trie. En este caso se puede recorrer más de un carácter de P por arista. Para pasar de una arista a la siguiente todos los caracteres de la etiqueta de la arista deben encontrarse en el patrón. La búsqueda puede terminar de tres formas:

- Se llega a un nodo a partir del cual no existe ninguna arista con cuyos caracteres comience lo que resta por recorrer en el patrón. En este caso P no se encuentra en T.
- Se termina de recorrer el patrón y se encuentra en medio de una arista o en un nodo.

En este caso todo el sub árbol que desciende del nodo corresponde a ocurrencias de P en T. este caso ocurre al buscar "a" o "abr" en el ejemplo de la figura 2.4.

- Se termina de recorrer el árbol y se llega a una hoja, pero aún no termina de recorrerse el patrón. Este caso requiere verificar en el texto (en la posición que indica la hoja) si P se encuentra en T. Este caso ocurre si buscamos "$dabra$" en el ejemplo de la figura 2.4.

En los tres casos la búsqueda toma tiempo $O(m)$. El conteo de ocurrencias de un patrón se realiza de la misma forma que en un suffix trie.

Una gran desventaja de los suffix trees es que utilizan al menos entre 9 y 12 veces el tamaño del texto.

Suffix Array Un suffix array es un arreglo de números que señalan las posiciones iniciales de los sufijos de un texto. Este arreglo está ordenado lexicográficamente de acuerdo a los sufijos. Por ejemplo, si se tienen los sufijos "b" y "eb" el sufijo "b" siempre se encontrará en una posición menor dentro del suffix array que "eb".

Texto

1	2	3	4	5	6	7	8	9	10	11
a	b	r	a	c	a	d	a	b	r	a

Suffix Array

11	8	1	4	6	2	9	5	7	10	3

Figura 2.8: Suffix Array de "abracadabra".

En la figura 2.5 es posible apreciar un suffix array, el cual contiene punteros al texto, los cuales indican todos los sufijos del texto en orden lexicográfico.

Para buscar un patrón en el índice se realizan dos búsquedas binarias del patrón en el suffix array. La primera búsqueda ubica la posición sp del primer sufijo que es lexicográficamente mayor o igual a P. La segunda búsqueda ubica la posición ep del último sufijo que comienza con P. Las ocurrencias de P en T se encuentran en el intervalo $A[sp, ep]$. El costo de realizar las dos búsquedas binarias es de $O(m \log n)$. Es posible reducir el costo de la búsqueda de los sufijos a $O(m + \log n)$. Esto se logra almacenando información adicional en el índice que permite reducir el costo de la comparación del patrón con los sufijos. Para reportar el número de ocurrencias basta con calcular $ep - sp + 1$.

Si bien esta estructura es un poco más lenta que un suffix tree, requiere menos espacio, aproximadamente 4 veces el tamaño del texto.

2.4.2.2. Auto-Índices comprimidos

Los auto-índices comprimidos son índices que además de permitir recuperar el texto original, utilizan menos espacio para almacenar sus estructuras. Para describir estos auto-índices es necesario explicar los conceptos y algoritmos que utilizan.

Transformada de Burrows-Wheeler La transformada de Burrows-Wheeler, también conocida como BWT (Burrows-Wheeler transform), es un algoritmo que se utiliza para transformar un texto en una versión usualmente más compresible que la original. Al aplicar esta transformada sobre un texto, se obtiene una permutación reversible de éste. Esta permutación tiene la propiedad de agrupar caracteres iguales cuando el texto contiene secuencias repetidas. Gracias a esta propiedad resulta más fácil comprimir la permutación.

BWT funciona de la siguiente forma:

Dado un texto T, de tamaño m:

1. Se crea una matriz M cuyas filas son shifts cíclicos de T.
2. Se ordenan las filas lexicográficamente
3. Se retorna la última columna de M como la transformada BWT de T.

Es importante señalar que para que esta transformación sea reversible, antes de aplicar la transformada se le agrega a T un carácter final, el cual es lexicográficamente menor a todos los caracteres de T y es único.

Ψ Sea $A[1, \ldots, n]$ el suffix array de T, $\Psi(i)$ se define como la posición i' de A donde$A[i'] = A[i] + 1$, salvo en el caso del último sufijo ($A[i] = n$) donde $A[i'] = 1$. En otras palabras, si un sufijo comienza en la posición i, Ψ indica en que parte del suffix array se encuentra el sufijo que comienza en la posición $i + 1$ del texto.

FM-Index FM-Index corresponde a una familia de auto-índices compactos. Contienen el texto en una representación comprimida de la transformada de Burrows-Wheeler, junto con estructuras auxiliares que permiten recuperar con rapidez la cantidad y la posición de las ocurrencias de los patrones a buscar en el texto original. Estas estructuras se hacen necesarias, ya que para recuperar el texto original se debe aplicar la transformada inversa sobre el texto. Si se comienza a aplicar la anti transformada en un lugar arbitrario del texto, no es posible saber cuántos caracteres hay antes del texto que se está recuperando. Por esta razón se deben guardar en intervalos regulares estos valores.

Compressed Suffix Array Es posible convertir un suffix array en un auto-índice comprimido, este índice se conoce como Compressed Suffix Array. Este tipo de índice se basa en la función Ψ.

Gracias a Ψ es posible recorrer el texto de izquierda a derecha en forma secuencial. Además de Ψ , se utilizan estructuras adicionales para eliminar la necesidad del texto. Por ejemplo, en la implementación de Sadakane se agrega un vector de bits que permite conocer los lugares en el suffix array donde los sufijos comienzan con un carácter distinto. En este caso el índice se compone de un arreglo con todos los valores de Ψ , el vector de bits llamado $newF$ y finalmente un arreglo que contiene los caracteres presentes en el texto en orden lexicográfico.

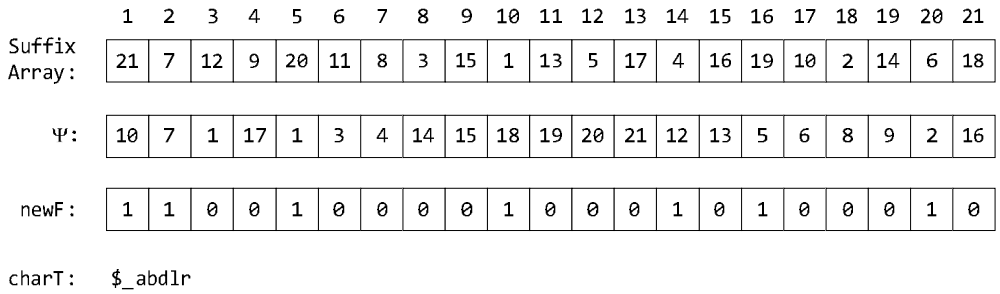

Figura 2.9: Suffix Array y compressed suffix array de sadakane, para T="alabar_a_la_alabarda$".

Para conocer las posiciones en el texto de las ocurrencias basta con tomar muestras de las posiciones a intervalos regulares. Para recuperar estos valores se calcula sucesivamente $\Psi(i)$ hasta que se encuentre un valor muestreado.

Si bien los arreglos $newF$ y Ψ son de tamaño n, es posible comprimirlos y operar sobre ellos en su forma comprimida, esto permite reducir el tamaño del índice de forma considerable.

2.5. Algoritmos de Intersección

Dados dos conjuntos A, B se define la intersección de estos como el conjunto que contiene los elementos que se encuentran tanto en A como en B. Existen muchas formas de obtener este conjunto, a continuación se describen algoritmos que permiten intersectar conjuntos representados como listas ordenadas.

Merge Este algoritmo recorre en forma secuencial ambas listas, selecciona el elemento más grande de ambas listas y recorre secuencialmente la lista de la que no proviene el elemento hasta encontrar un elemento de igual o mayor tamaño o se termine la lista, si el elemento encontrado es igual éste se agrega al resultado, si es mayor entonces se repite el proceso

pero buscando el elemento mayor en la otra lista, el algoritmo finaliza cuando se recorre completamente alguna de las listas.

Bys Se busca la mediana de la lista más pequeña en la lista más grande. Si se encuentra el elemento en la lista más grande, entonces se agrega este elemento al resultado. En el caso contrario se obtiene la posición del elemento inmediatamente más grande que el elemento buscado en la lista más grande. De esta forma se dividen ambas listas en dos. Como la búsqueda de la mediana encuentra la posición de este elemento o al inmediatamente más grande en la lista más grande, ningún elemento que se encuentre a la izquierda de esa posición puede estar a la derecha de la posición de la mediana de la lista más pequeña. Los elementos de la izquierda de la mediana sólo pueden encontrarse a la izquierda de la posición encontrada en la lista más grande. Esto permite continuar intersectando de forma recursiva la mitad izquierda y la mitad derecha.

Set versus Set El algoritmo Set versus Set (SvS) recorre secuencialmente la lista más pequeña buscando los elementos de esta en la lista más grande.

A diferencia de Merge, para encontrar los elementos se utiliza un algoritmo de búsqueda binaria, o un algoritmo de búsqueda exponencial, los cuales reportan la posición en la lista donde se encuentra el elemento buscado, de no encontrarse un elemento en la lista más grande los algoritmos reportan la posición del elemento más pequeño de la lista que es mayor al elemento buscado. Cuando se busca un nuevo elemento se restringe el intervalo de búsqueda en la lista grande desde la última posición reportada hasta el final de ésta.

Capítulo 3

Construcción del Índice

En este capítulo se detalla la construccion del indice comprimido de q-gramas basado en bloques descrito en la sección 2.4.1.2.

Para construir un índice a partir de un texto T de tamaño n, se utiliza una tabla de hash, en la cual se almacenarán los q-gramas encontrados en el texto.

Al momento de construir el índice se fija un parámetro q, el cual define el largo de los q-gramas del índice.

El texto se recorre de forma secuencial agregando cada uno de los q-gramas encontrados a la tabla de hash. Se leen los primeros q caracteres $T[1, .., q]$ y se forma un q-grama, a continuación se lee el carácter $q+1$ y se forma un q-grama con los caracteres entre 2 y $q+1$, es decir $T[2, ..., q+1]$, este proceso se repite hasta llegar al final del texto.

En el caso del índice donde se señalan las posiciones exactas de cada ocurrencia, cada vez que se lee un q-grama se busca este q-grama dentro de la tabla de hash, si el q-grama no se encuentra en la tabla, este se inserta en ella junto con su posición en el texto, si el q-grama se encuentra en la tabla se recupera la lista de ocurrencias asociada a este y se almacena en la lista la posición de la nueva ocurrencia.

El caso de que el índice no guarde la posición exacta sino que utilice bloques es muy similar al anterior pero tiene las siguientes diferencias: Se busca el q-grama en la tabla de hash, si no se encuentra se guarda el q-grama en la tabla de hash junto con el número de bloque donde comienza el q-grama. Si se encuentra el q-grama en la tabla de hash, entonces se verifica cuál fue el último número de bloque agregado a la lista del q-grama. El número de bloque se agrega sólo cuando el valor de la última ocurrencia agregada a la lista de ocurrencias del q-grama es diferente al bloque actual. Al igual que en el caso anterior, no se almacenan los valores de las posiciones de las ocurrencias, sino que las diferencias entre las posiciones.

Las ocurrencias asociadas a los q-gramas se almacenan en un arreglo. Como no es posible conocer de antemano la cantidad de ocurrencias asociadas a cada q-grama, al momento de

agregar la primera ocurrencia se crea un arreglo de un tamaño fijo. Si se agrega una ocurrencia cuando el arreglo está completo, se duplica el tamaño del arreglo y luego se agrega el elemento. Una vez que se han agregado todas las ocurrencias se modifica el tamaño de cada uno de los arreglos.

Para facilitar la búsqueda dentro de bloques de texto el procesamiento del texto no sea realiza de forma completamente secuencial, sino que cada bloque tiene al final parte del principio del bloque siguiente. La cantidad de traslape está dada por un parámetro fijo *overlap*. Este traslape permite asegurar que al buscar dentro del bloque no será necesario buscar en el siguiente bloque si al final del texto se encuentra un prefijo del patrón a buscar.

Si los bloques se indexaran sin traslape es posible que un patrón se encuentre entre dos bloques, quedando el prefijo de un patrón en un bloque y su sufijo en el siguiente bloque. Esto obligaría a revisar ambos bloques.

Al indexar con traslape, siempre que los patrones a buscar sean de tamaño *overlap* o menores, sólo se buscara el patrón en los bloques que contengan todos los q-gramas del patrón. Si se desea buscar un patrón de un tamaño mayor a *overlap*, se corta este patrón y se busca uno de tamaño *overlap*. Luego cada vez que se encuentra este patrón recortado, se busca el patrón original en cada bloque. Sólo en los casos en los que el patrón recortado se encuentre al final del bloque será necesario buscar el resto del patrón en el bloque siguiente.

Una vez que se ha terminado de procesar el texto original, se extraen todos los q-gramas de la tabla de hash y se ordenan lexicográficamente, obteniendo de esta forma el índice invertido de q-gramas.

Debido a que cada vez que se lee un carácter (salvo los primeros $q-1$) se hace una búsqueda en la tabla de hash, en total se realizan $n-(q-1)$ búsquedas. Esto implica calcular la función de hash $n-(q-1)$ veces. Normalmente esto tardaría tiempo $O(q(n-(q-1)))$, es posible mejorar este tiempo utilizando una función de hash especial para este propósito, la cual permite reducir este tiempo a $O(n)$. Esta función se conoce como Rolling Hash. Para este propósito se utiliza el algoritmo de hash de Rabin-Karp, descrito en la sección 2.2.

3.1. Compresión del Índice

La compresión del índice consiste en comprimir cada una de las listas de ocurrencias. En vez de escribir la posición de la ocurrencia, se guarda la resta entre la posición actual y la posición de la ocurrencia anterior. Si tenemos la lista $< p_1, p_2, p_3 \ldots, p_l >$, se almacena $< p_1, p_2 - p_1, p_3 - p_2, \ldots . p_l - p_{l-1} >$. Como las listas de ocurrencias están ordenadas de forma creciente, la lista de diferencias sólo tiene números positivos. Guardar las diferencias no utiliza menos espacio, pero sí permite que las listas tengan valores mucho más pequeños

en comparación a la lista con los valores absolutos. Esta propiedad es explotada utilizando una codificación de bits variable que permita utilizar menos bits para valores pequeños. En particular se utilizan Gamma, Delta, Rice, Simple9 y Simple16 descritos en la sección 2.1.4.

3.1.1. Sampling de las listas de ocurrencias

El problema de codificar las listas de restas es que ya no se puede acceder a cualquier elemento de la lista de forma aleatoria, sino que se debe recorrer la lista de forma secuencial para saber dónde comienza cada valor codificado. Además se debe conocer el valor absoluto de la posición de la ocurrencia anterior y sumárselo al último valor leído para conocer la posición real de esta ocurrencia. Dependiendo de los algoritmos utilizados en la etapa de filtrado, será necesario acceder a la lista de forma no secuencial. Esto se logra extrayendo muestras de cada lista, guardando su valor absoluto e indicando en qué parte de la lista se encuentran. Esto no sólo permite obtener los elementos que están en la muestra, sino también extraer los elementos que se encuentran a continuación de los elementos muestreados. Las muestras se toman cada $k \log l$ elementos, siendo k un parámetro del índice y l el largo de la lista. Como no es posible conocer el largo de la lista en la etapa de construcción, las muestras se extraen una vez que se ha terminado de almacenar las ocurrencias de q-gramas en el índice. Con esta técnica de sampling, acceder a un elemento cualquiera requiere decodificar a lo más $k \log l$ elementos

Gamma, Delta, Rice Para estos algoritmos de codificación se guarda el valor absoluto del elemento muestreado, la posición del byte en la que se encuentra primer bit del elemento dentro de la lista y el bit donde comienza el elemento dentro de ese byte

Para decodificar los elementos que se encuentran a continuación del elemento muestreado, se lee el elemento muestreado y luego al decodificar el siguiente elemento se le suma el valor del elemento anterior.

Simple9, Simple16 El caso de Simple9 y Simple16 es especial con respecto a los demas códigos ya que estas codificaciónes comprimen y extraen los elementos en grupos. Para no extraer múltiples veces cada elemento, se mantiene un pequeño buffer en cada lista que contiene los últimos elementos extraídos, y cada vez que se quiera obtener la siguiente ocurrencia se busca el elemento en el buffer. Si el buffer se encuentra vacío se extrae el siguiente grupo de elementos. Otra diferencia es que las muestras no se encuentran a la misma distancia, sino que varían en una cantidad acotada de elementos (máximo 27). De no ser así habría que cortar los grupos con el propósito de tener la muestra al comienzo del

grupo, o almacenar la posición del elemento muestreado dentro del grupo codificado en el que se encuentra.

Parámetros del Índice El cuadro 3.1 contiene un resumen de los parámetros utilizados por el índice.

Parámetro	Nombre	Si crece...
Tamaño de q-grama	q	Mejora el tiempo en la etapa de filtrado. Aumenta el tamaño del índice en $O(\sigma^q)$, siendo σ el tamaño del alfabeto del texto.
Constante de sampling	k	Aumenta el tiempo de intersección de algoritmos no secuenciales. Disminuye el tamaño del índice.
Tamaño de bloque	b	Disminuye el tamaño del índice. Aumenta el tiempo de verificación en el texto.
Traslape entre bloques	$overlap$	Permite buscar patrones más grandes directamente en el índice. Aumenta el tamaño del texto.
Finalización de la etapa de filtrado	f	Disminuye el tiempo en la etapa de verificación de bloques. Aumenta el tiempo en la etapa de filtrado

Cuadro 3.1: Parámetros del índice.

3.1.2. Espacio del índice

Al comprimir el índice, se codifican las listas de ocurrencias, pero los q-gramas en el índice permanecen intactos. Suponiendo un texto aleatorio, podemos calcular el espacio utilizado por los q-gramas del índice, suponemos que existen σ^q distintas "urnas" (q-gramas) y n "bolas" (q-gramas en el texto). La probabilidad de que un q-grama sea seleccionado en una prueba de Bernoulli es de $1/\sigma^q$. Por lo tanto la probabilidad de que un q-grama no sea seleccionado en n pruebas es $(1 - 1/\sigma^q)^n$. Por lo tanto el número promedio de q-gramas seleccionados en n pruebas es $V = \sigma^q(1 - (1 - 1/\sigma^q)^n) = \Theta(\sigma^q(1 - e^{-n/\sigma^q})) = \Theta(min(n, \sigma^q))$. El espacio utilizado por las listas de ocurrencias se mide experimentalmente.

Capítulo 4

Búsqueda en el Índice de Bloques

La búsqueda de patrones en un índice invertido de q-gramas de bloques se realiza en tres etapas.

Primero se buscan los bloques candidatos donde se encuentran los q-gramas del patrón. Luego se intersectan estas listas de bloques. De la intersección se obtiene una lista de bloques donde podría encontrarse el patrón. Finalmente se busca el patrón dentro de cada uno de esos bloques.

4.1. Búsqueda de Q-gramas

Para buscar un q-grama en el índice se realiza una búsqueda binaria . Si se tienen V q-gramas diferentes en el índice, la búsqueda tarda $O(log(V))$. Si el q-grama se encuentra en el texto se retornará la lista de ocurrencias. Si el q-grama no se encuentra en el índice, entonces tampoco se encuentra en el texto. El resultado de la búsqueda es la lista de bloques donde se encuentra el q-grama buscado.

4.2. Búsqueda General

Si se desea buscar un patrón más pequeño que un q-grama se deben buscar todos los q-gramas que contengan el patrón buscado en su interior. Para esto se podrían generan todos los q-gramas posibles que contengan al patrón y buscar cada uno de ellos. Existe una mejor alternativa, buscar solamente los q-gramas que comienzan con el patrón. Como los q-gramas se encuentran ordenados lexicográficamente dentro del índice, los q-gramas que comienzan con el patrón se encuentran en una región continua del índice. El resultado de la búsqueda es la unión de las listas de ocurrencias asociadas a los q-gramas en la región.

La búsqueda de patrones P de largo m más grande que un q-grama, es decir $m > q$ se

realiza en tres etapas, cálculo de cobertura, filtrado de ocurrencias y verificación de bloques candidatos. Se exploran dos alternativas para generar la cobertura, la cobertura máxima consiste en generar todos los substrings de largo q contenidos en el patrón, también se puede generar una cobertura mínima mediante un algoritmo greedy, tomando los primeros q caracteres para formanr el primer q-grama $P[1,..,q]$, desde el carácter $P[q+1]$ hasta $P[2q]$ para formar el segundo q-grama $P[q+1,...,2q]$ y así sucesivamente hasta llegar al final del patrón. Si la división no es exacta entonces el último q-grama contiene los caracteres $P[m-q, Pm-q+1, \ldots m]$. Si todos los q-gramas de la cobertura son distintos entre sí, para la cobertura máxima se deben buscar $m-q+1$ q-gramas, mientras que con la cobertura mínima se deben buscar $\lceil m/q \rceil$ q-gramas en el índice.

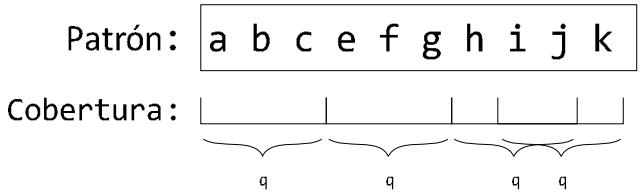

Figura 4.1: Ejemplo de cobertura mínima del patrón "abcefghijk" para $q = 3$.

Una vez generada la cobertura se ordenan los q-gramas extraídos del patrón en orden lexicográfico. Se busca el q-grama más pequeño en la lista, si no se encuentra entonces el patrón no está en el texto. En el caso contrario se busca el siguiente q-grama pero la búsqueda binaria se realiza desde la posición del primer q-grama encontrado y el final del índice. Se repite este procedimiento hasta recuperar las listas de ocurrencias asociadas a todos los q-gramas extraídos del patrón. La figura 4.2 ilustra este proceso.

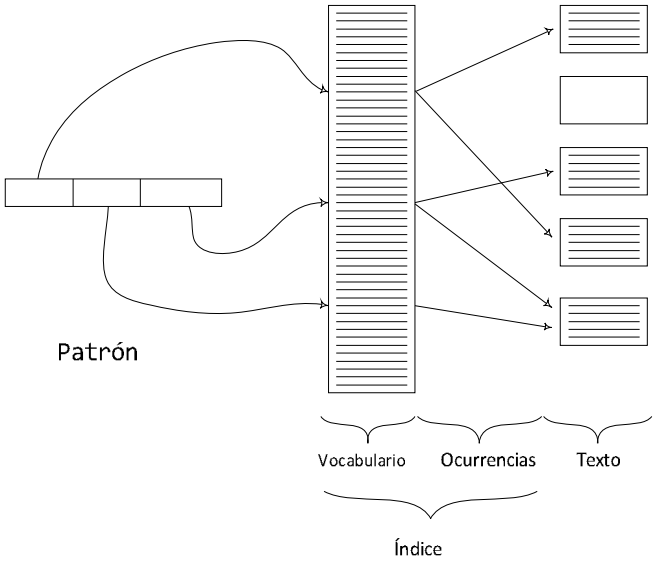

Figura 4.2: Proceso de búsqueda.

4.2.1. Filtrado de resultados

El último paso en el índice consiste en intersectar las listas de ocurrencias. Sólo se deben seleccionar los bloques que contengan todos los q-gramas de la cobertura del patrón.

Para realizar las intersecciones se ordenan las listas de ocurrencias de forma creciente. Luego se intersectan la lista más pequeña con la segunda más pequeña. El resultado de esta intersección se intersecta con la tercera lista más pequeña, este proceso se repite iterativamente hasta que se obtenga una lista de tamaño f, o se intersecten todas las listas.

4.2.1.1. Algoritmos de intersección

Mientras que la intersección de las listas puede realizarse recorriendo en forma sincronizada las listas de forma iterativa, es posible que la diferencia en el tamaño de las listas a intersectar sea muy grande. Existen algoritmos que explotan esta condición, intersectando las listas de forma más rápida. Por este motivo se implementaron los algoritmos de intersección descritos en la sección 2.5 adaptándolos para intersectar las listas de ocurrencias comprimidas. A continuación se detalla cómo se implemento cada algoritmo.

Merge Ademas de la versión original, se implemento una versión alternativa (llamada Merge2) donde se busca en forma secuencial cada uno de los elementos de la lista más pequeña en la lista de ocurrencias más grande. Si se encuentra el elemento se agrega al resultado. Si no se encuentra se busca el siguiente elemento en la lista grande comenzando desde el último elemento comparado en la búsqueda del elemento anterior.

Bys Se implementó la versión original.

Set versus Set A diferencia de su versión original, el algoritmo implementado busca de forma binaria o exponencial sobre la muestra de elementos de la lista más larga y con esto se encuentra la primera muestra que es más grande que el elemento buscado. Si el elemento buscado se encuentra en la lista más larga, entonces debe estar entre la muestra encontrada y la muestra anterior. Para verificar si el elemento se encuentra en ese intervalo se decodifican todos los elementos entre ambas muestras, y se busca el elemento dentro del intervalo mediante una búsqueda binaria o una búsqueda exponencial. Si el elemento se encuentra, se agrega a la lista de candidatos. Luego se determina si es que el siguiente elemento de la lista pequeña puede encontrarse dentro del intervalo de muestras que acaba de ser decodificado. De ser así se busca de la misma forma, pero la búsqueda se realiza entre la posición del elemento encontrado anteriormente y el fin del intervalo recuperado. Si el siguiente elemento no se encuentra en este intervalo, se busca un nuevo intervalo candidato de la misma forma en la que se buscó el primero, pero acotando el intervalo de búsqueda en las muestras.

Capítulo 5

Construcción de Bloques y Búsqueda en Texto

5.1. Construcción de Bloques

Para construir los bloques se exploraron dos alternativas: generar los bloques y comprimirlos de forma independiente y comprimir el texto con un algoritmo que permita descompresión local para luego generar los bloques.

Para generar los bloques se lee el texto de forma secuencial repitiendo al final de cada bloque una porción de texto del bloque siguiente, de tamaño *overlap*. Para comprimir los bloques se utilizaron 2 programas: *compress* y *gzip*. *compress* implementa una versión mejorada de LZ78 (ver sección 2.1.2) conocida como LZW. *gzip* implementa el algoritmo DEFLATE, el cual combina Huffman Coding (ver sección 2.1.1) y LZ77 (ver sección 2.1.2).

Para comprimir y luego generar los bloques se implementó una versión modificada de Re-Pair aproximado en disco (sección 2.1.3). En primer lugar se combinaron las etapas de reemplazo de pares y compactación. Además se agrego un símbolo especial que no debe ser reemplazado. Al agregar este símbolo al final de cada bloque se puede comprimir el texto completo utilizando un solo diccionario, permitiendo descomprimir los bloques de forma independiente.

5.2. Búsquedas en Texto Comprimido

Una vez que el índice genera la lista de bloques en donde se puede encontrar el patrón, es necesario verificar si realmente existen ocurrencias del patrón en cada bloque. Si se tratase de bloques de texto no comprimidos basta con utilizar un algoritmo lineal. Un algoritmo de tiempo lineal resulta aceptable ya que el tamaño de los bloques es significativamente inferior

al tamaño del texto.

La búsqueda de patrones dentro de un texto comprimido se puede realizar ya sea buscando dentro del texto comprimido o descomprimiendo el texto y luego utilizando un algoritmo de búsqueda en texto plano.

5.2.1. LZgrep

LZgrep es una herramienta que permite buscar patrones dentro de textos comprimidos. Para tal efecto el texto debe estar comprimido con el algoritmo DEFLATE o el agoritmo LZW, el cual es una implementación mejorada de LZ78.

La búsqueda se realiza de forma similar a la del algoritmo Boyer Moore. La diferencia radica en que en algunos casos es posible descomprimir el texto sólo de forma parcial.

5.2.2. Re-pair

Para buscar en bloques comprimidos con Re-Pair se descomprime el bloque completamente y luego se busca el patrón en el texto plano utilizando Boyer-Moore-Horspool, la descripción de este algoritmo se encuentra en la sección 2.3

Capítulo 6

Resultados Experimentales

Con el fin de determinar los mejores parámetros y algoritmos utilizados en el índice, los experimentos se dividieron en cuatro etapas.

En la primera etapa se midió experimentalmente el tiempo y la utilización de memoria del índice en la etapa de filtrado, con el fin de determinar los algoritmos de intersección y compresión de listas que presentan mejor rendimiento.

En la segunda etapa se midió experimentalmente el tiempo de búsqueda de patrones en los bloques de texto sin considerar el tiempo que tarda el índice en generar las listas de bloques.

En la tercera etapa se midió el tiempo y la memoria utilizada por el índice al buscar patrones de distinto largo para determinar los parámetros que afectan al índice de forma global.

Finalmente se comparó el rendimiento de las mejores versiones del índice con auto-índices competitivos del sitio Pizza&Chili.

6.1. Procedimiento

Todos los experimentos se ejecutaron en un computador con 2 procesadores Core 2 Duo de 3Ghz 6MB cache, 8GB RAM, 2TB de disco. El sistema operativo utilizado es Ubuntu 8.04 (64bits) con kernel 2.6.24-27-server. El compilador utilizado fue g++ (gcc versión 4.2.4) con la opción de optimización –O3.

Para realizar los experimentos se utilizaron los textos de prueba *english*[1] (colección de textos en inglés) y *dna*[2] (secuencias de adn) en sus versiones de 200MB del sitio Pizza&Chili.

[1] http://pizzachili.dcc.uchile.cl/texts/nlang/
[2] http://pizzachili.dcc.uchile.cl/texts/dna/

6.2. Búsqueda y Filtrado en el Índice

Tanto la compresión del índice, como el tiempo de búsqueda de q-gramas y filtrado de ocurrencias, son independientes de los métodos y técnicas utilizadas para buscar en los bloques de texto. Gracias a esto, es posible determinar de forma independiente qué parámetros y algoritmos presentan mejores resultados en estas etapas. El Cuadro 6.1 presenta los parámetros y algoritmos que se miden en este experimento.

Para realizar el experimento se buscaron 5000 patrones de cada largo en cada índice, cada patrón se generó extrayendo desde una posición aleatoria una porción del texto original. Todas las búsquedas se realizaron generando una cobertura mínima del patrón a buscar.

Parámetro / Algoritmo	Valores
Algoritmo de compresión de listas	*gamma*, *delta*, *rice*, *Simple*9, *Simple*16
Algoritmo de intersección de listas	*merge*, *merge*2, *SvS*, *bys*
Tamaño de q-grama q	$8, 10$ para *dna*; $2, 5$ para *english*
constante de sampling k	$1, 4, 10$
Tamaño de bloque b	4KB
Tamaño de traslape *overlap*	100
Tamaño de la lista para terminar la etapa de filtrado f	∞
Largo de patrón m	$2q$, $5q$

Cuadro 6.1: Parámetros y algoritmos utilizados en el experimento.

A continuación se presenta parte de los resultados obtenidos.

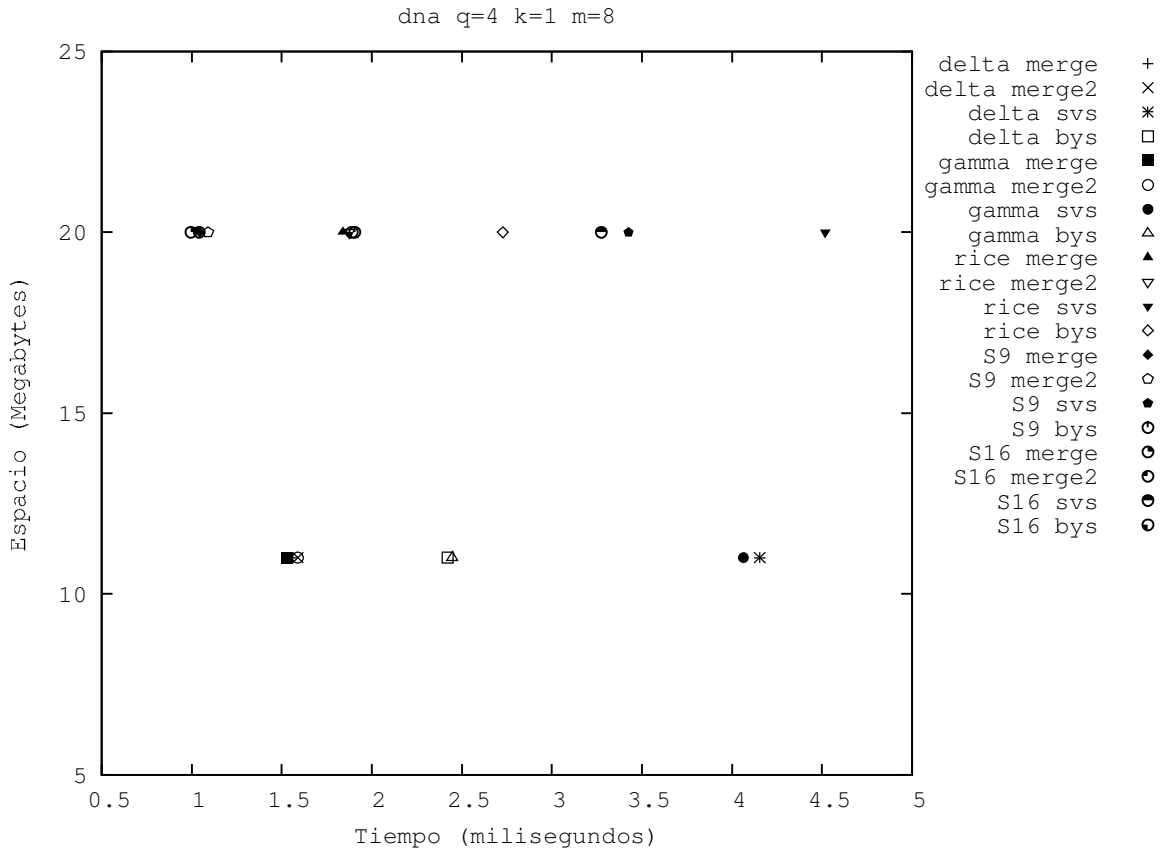

Figura 6.1: Tiempo y espacio de índices para *dna*, q=4, k=1 y m=8.

En la figura 6.1 se aprecia que tanto Gamma como Delta logran compresiones similares, entre los índices que utilizan menos espacio Gamma junto a Merge y Merge2 crean los índices más rápidos. Por otra parte S9 y S16 junto a Merge y Merge2 generan los índices más rápidos en la prueba.

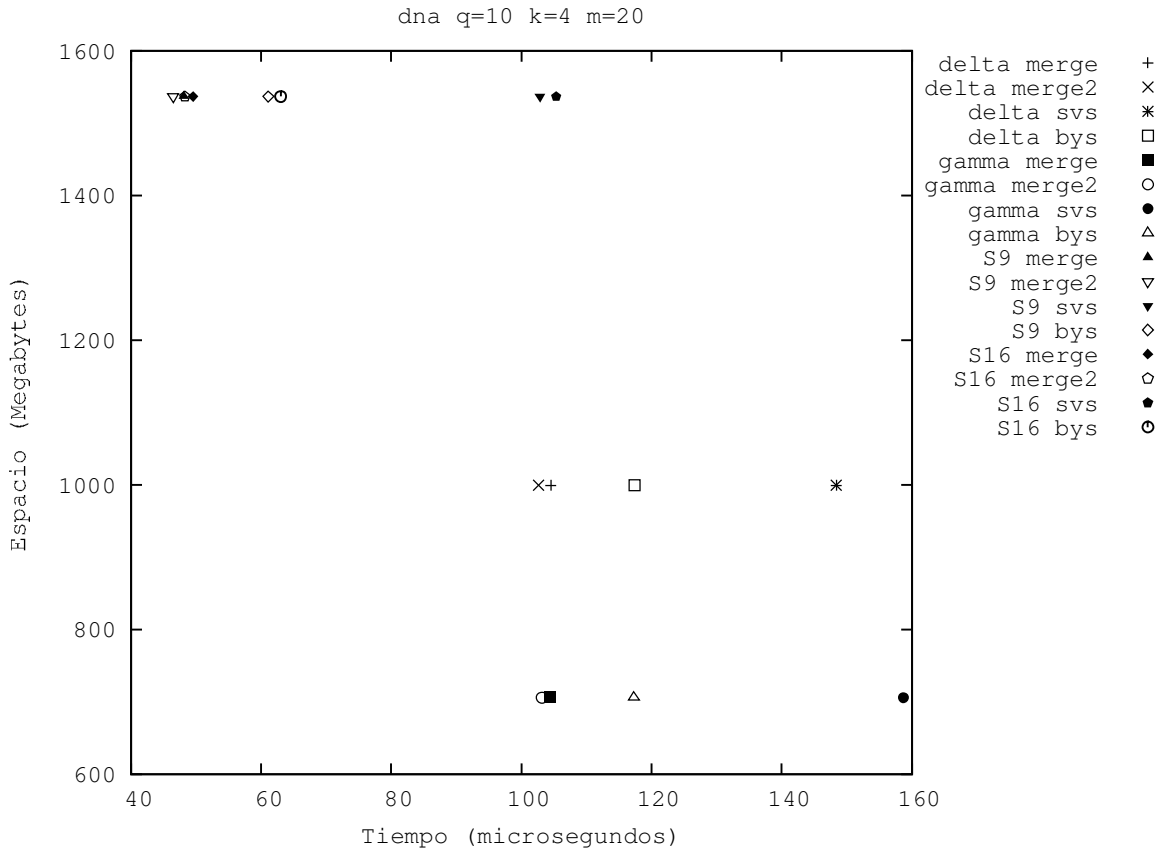

Figura 6.2: Tiempo y espacio de índices para *dna*, q=10, k=4 y m=20.

En la figura 6.2 se aprecia que para valores grandes de q el índice se vuelve demasiado grande. En esta prueba Gamma utiliza menos espacio que Delta, pero tienen rendimientos similares. Nuevamente Merge y Merge2 generan los índices más rápidos, siendo S9 y S16 los más rápidos entre ellos.

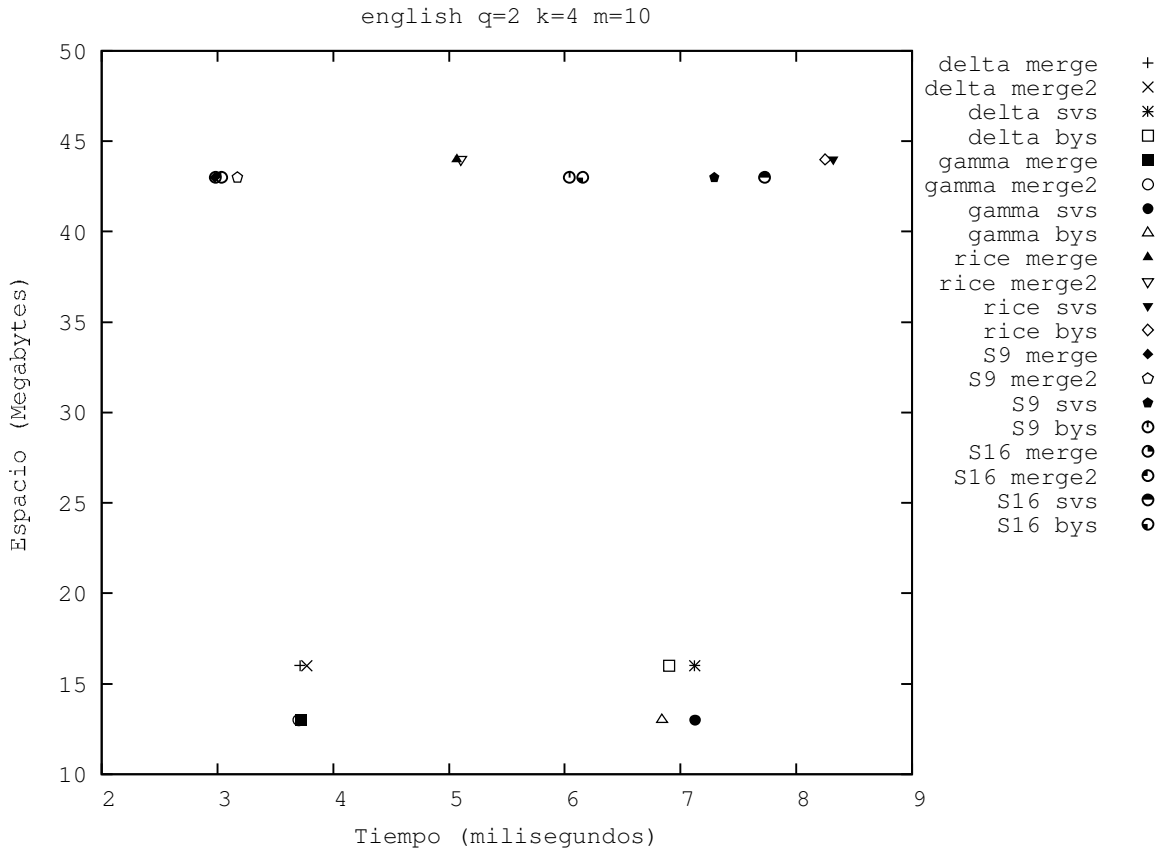

Figura 6.3: Tiempo y espacio de índices para *english*, q=2, k=4 y m=10.

La figura 6.3 presenta resultados muy similares a los obtenidos para *dna*. En esta prueba los algoritmos de compresión Delta y Gamma logran una compresión mucho mayor que la del resto de los algoritmos de compresión.

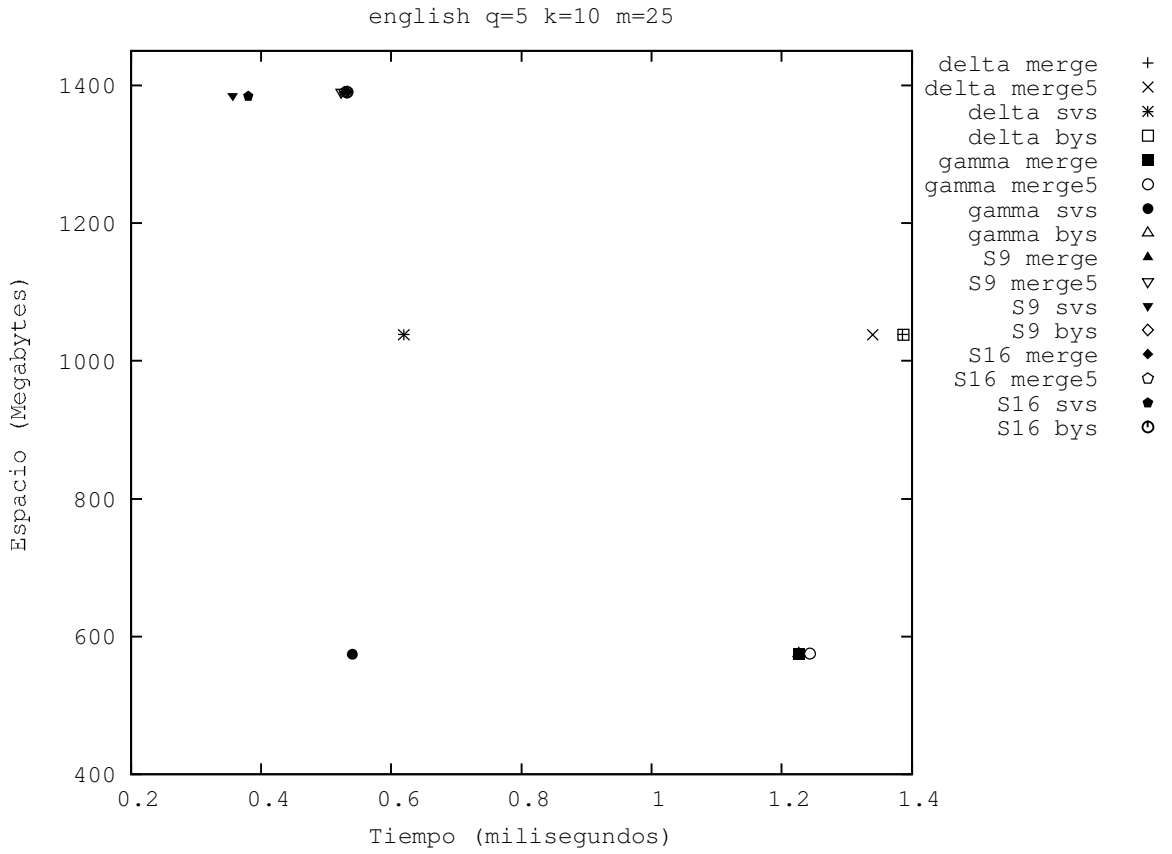

Figura 6.4: Tiempo y espacio de índices para *english*, q=5, k=10 y m=25.

La figura 6.4 muestra que para patrones suficientemente largos SvS es más rápido que Merge. Además se aprecia que el tamaño del índice es demasiado grande, lo que acota el valor de q para *english* en las pruebas siguientes.

Los resultados experimentales fueron consistentes en todas las pruebas. La codificación Gamma obtuvo la mejor compresión, Simple9 y Simple16 la mejor velocidad. Para patrones cortos Merge y Merge2 presentan el mejor rendimiento. Mientras que para patrones largos SvS presenta un rendimiento superior, especialmente para $k = 4$.

6.3. Búsqueda en Texto Comprimido

Con el objetivo de comparar las diferentes técnicas de compresión y búsqueda en texto se midió tanto el nivel de compresión, como el tiempo de búsqueda de patrones para las técnicas y algoritmos descritos en el capítulo 5.

Compresor / b	8KB	128KB	512KB
Re-Pair	0,5	0,492	0,492
compress	0,52	0,415	0,391
Gzip	0,51	0,415	0,286

Cuadro 6.2: Compresión de *english*.

Compresor / b	8KB	128KB	512KB
Re-Pair	0,265	0,264	0,264
compress	0,286	0,257	0,248
Gzip	0,491	0,296	0,286

Cuadro 6.3: Compresión de *dna*.

Los cuadros 6.2 y 6.3 muestran el nivel de compresión que logra el texto con cada algoritmo de compresión para distintos tamaños de bloque, el tamaño se expresa como una fracción del texto original. El nivel de compresión se ve muy impactado tanto en *Compress* como en *Gzip* por el tamaño de bloque, la razón principal radica en que al comprimir cada bloque de forma independiente, los traslapes no tendrán un nivel de compresión muy alto.

Para medir la velocidad de búsqueda se genero un índice utilizando el algoritmo de compresión Gamma y el algoritmo de intersección Merge, fijando $q = 2$ para *english*, $q = 4$ para *dna* y *overlap* en 100. Se buscaron 5000 patrones de largo $2q$ generando una cobertura mínima y se midió el tiempo de búsqueda en el texto, sin considerar el tiempo de intersección en el índice.

Inicialmente se consideró probar con los tamaños de bloques de las figuras 6.3 y 6.2, pero los resultados obtenidos para 8KB indicaron que Re-Pair es entre 40 y 60 veces más rápido. Como los bloques más grandes tienen mayor compresión LZgrep demoraría aún más en buscar para estos casos que Re-Pair.

6.4. Parámetros Globales del Índice

Gracias a los experimentos anteriores se fijaron los algoritmos de compresión y búsqueda del índice, además se fijó la constante de sampling y se acotó el tamaño de q para cada texto.

En esta sección se presentan los resultados de los experimentos realizados para fijar los parámetros que afectan al índice tanto en la etapa de filtrado como en la búsqueda en bloques.

Para medir el tiempo de búsqueda se generaron patrones extrayendo porciones del texto original desde posiciones aleatorias. Al igual que en las pruebas anteriores se generaron 5000 patrones de cada longitud.

Texto / b	1KB	4KB	8KB	128KB	512KB
english	0,578	0,507	0,500	0,492	0,492
dna	0,310	0,269	0,265	0,264	0,264

Cuadro 6.4: Compresión del texto para los valores medidos de b.

El Cuadro 6.4 muestra el nivel de compresión alcanzado por Re-Pair para distintos tamaños de bloques, expresado como una fracción del texto original.

q	b	algoritmo de compresión	tamaño del índice	tamaño total
3	1024	Gamma	0,537	1,115
3	1024	S9	1,039	1,617
3	4096	Gamma	0,299	0,805
3	4096	S9	1,008	1,514
3	8192	Gamma	0,230	0,730
3	8192	S9	1,062	1,561
3	131072	Gamma	0,116	0,609
3	131072	S9	0,439	0,931
3	524288	Gamma	0,098	0,589
3	524288	S9	0,261	0,753
4	1024	Gamma	1,438	2,016
4	1024	S9	3,352	3,930
4	4096	Gamma	1,036	1,543
4	4096	S9	3,157	3,664
4	8192	Gamma	0,881	1,381
4	8192	S9	3,038	3,537
4	131072	Gamma	0,594	1,086
4	131072	S9	1,736	2,228
4	524288	Gamma	0,532	1,024
4	524288	S9	1,264	1,756

Cuadro 6.5: Tamaños de los índices generados en las pruebas del texto *english*.

q	b	algoritmo de compresión	tamaño del índice	tamaño total
6	1024	Gamma	0,576	0,885
6	1024	S9	0,934	1,244
6	4096	Gamma	0,220	0,489
6	4096	S9	1,075	1,344
6	8192	Gamma	0,135	0,399
6	8192	S9	1,149	1,413
6	131072	Gamma	0,024	0,289
6	131072	S9	0,191	0,455
6	524288	Gamma	0,017	0,282
6	524288	S9	0,062	0,327
8	1024	Gamma	1,882	2,192
8	1024	S9	5,884	6,194
8	4096	Gamma	1,159	1,428
8	4096	S9	5,228	5,497
8	8192	Gamma	0,886	1,151
8	8192	S9	4,889	5,154
8	131072	Gamma	0,200	0,464
8	131072	S9	2,045	2,309
8	524288	Gamma	0,094	0,359
8	524288	S9	0,719	0,985

Cuadro 6.6: Tamaños de los índices generados en las pruebas del texto *dna*.

Los Cuadros 6.5 y 6.6 muestran el espacio utilizado por el índice en memoria junto al tamaño total que incluye el espacio utilizado por el texto en memoria secundaria.

En general valor de q es el que tiene mayor influencia sobre el tamaño del texto. El tamaño de los bloques también influye sobre el tamaño del índice, pero como el tamaño del texto no disminuye significativamente al aumentar b el impacto de este parámetro es menor sobre el tamaño total.

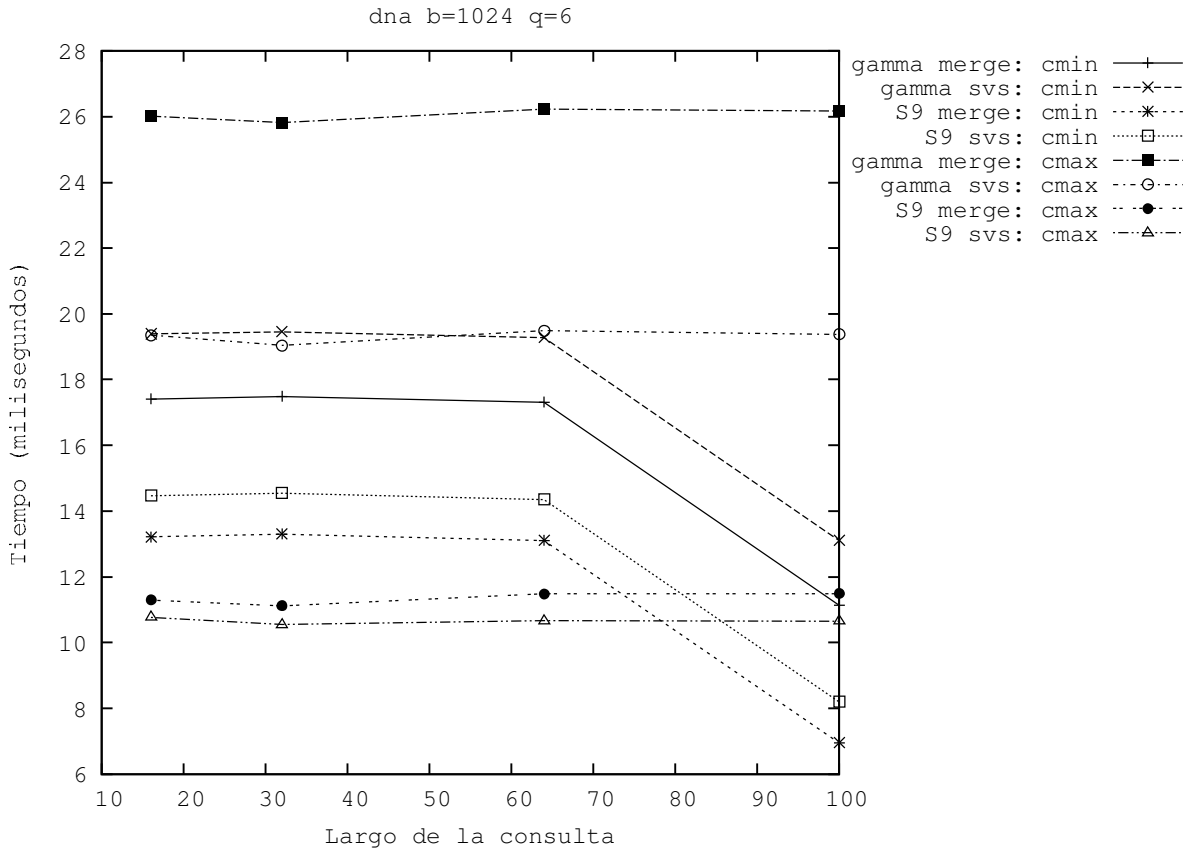

Figura 6.5: Tiempo de búsqueda para *dna*, b=1KB, q=6.

La figura 6.5 muestra el tiempo de búsqueda para patrones de diferentes longitudes en un índice para *dna*, con $b = 1$KB y $q = 6$, *cmin* indica la utilización de una cobertura mínima, mientras que *cmax* indica la utilización de una cobertura máxima. Se aprecia que para bloques pequeños los algoritmos de intersección y de compresion utilizados influyen significativamente sobre los resultados. El tiempo de búsqueda no cambia con respecto al tamaño del patrón buscado, salvo en los casos de coberturas mínimas para patrones de tamaño 100. Esto se debe en gran parte a que mientras largo el patrón, más q-gramas contiene su cobertura y más listas deben intersectarse. Si el patrón es lo suficientemente grande la cobertura mínima permite filtrar una cantidad suficiente de bloques de texto.

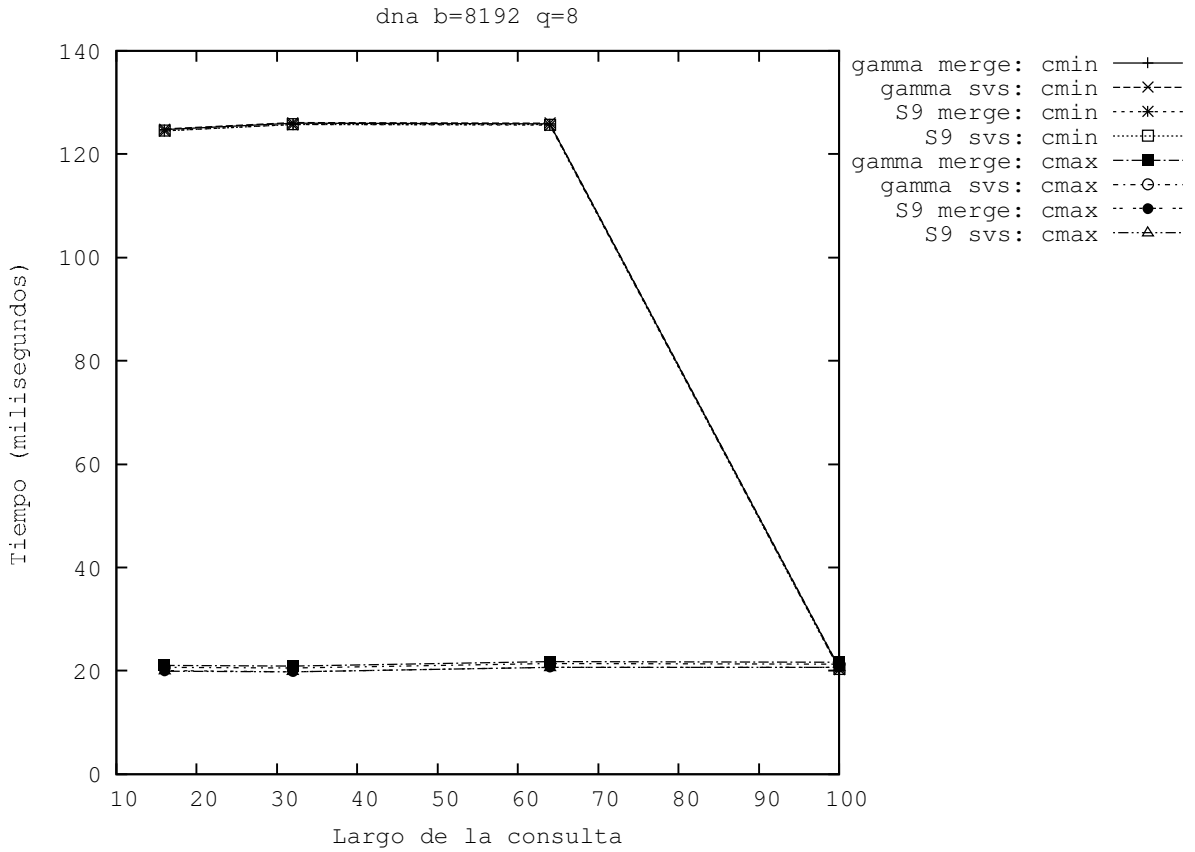

Figura 6.6: Tiempo de búsqueda para *dna*, b=8KB, q=8.

La figura 6.6 muestra un escenario muy diferente a la figura 6.5. En este caso las búsquedas con cobertura mínima son significativamente más lentas que las búsquedas con cobertura máxima. Esto ocurre ya que al utilizar$b$=8KB, cada bloque contiene más q-gramas. Al tener listas de ocurrencias más llenas, se filtran menos bloques durante una intersección. Esto que explica la gran diferencia en tiempos entre las búsquedas con cobertura mínima y máxima.

La diferencia de tiempo de búsqueda al utilizar las distintas coberturas se mantuvo consistente en las pruebas para b mayor a 1KB, independiente del texto y el tamaño de q-grama utilizado.

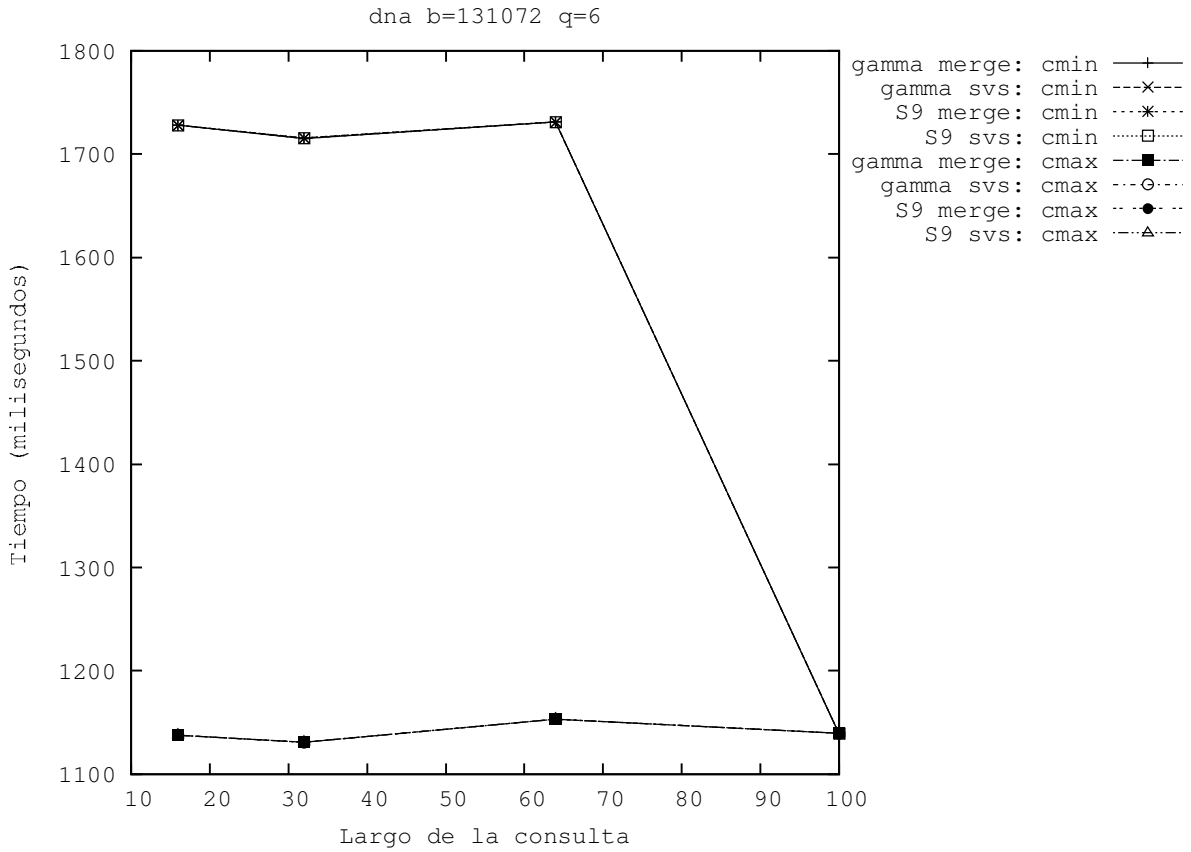

Figura 6.7: Tiempo de búsqueda para *dna*, b=128KB, q=6.

En la figura 6.7 muestra que para bloques grandes sigue existiendo una gran diferencia en los tiempos según el tipo de cobertura utilizado. Aún cuando se utiliza un q mayor, este tamaño de bloque ya produce índices competitivos. Los tiempos de búsqueda son muy grandes con respecto a otras alternativas.

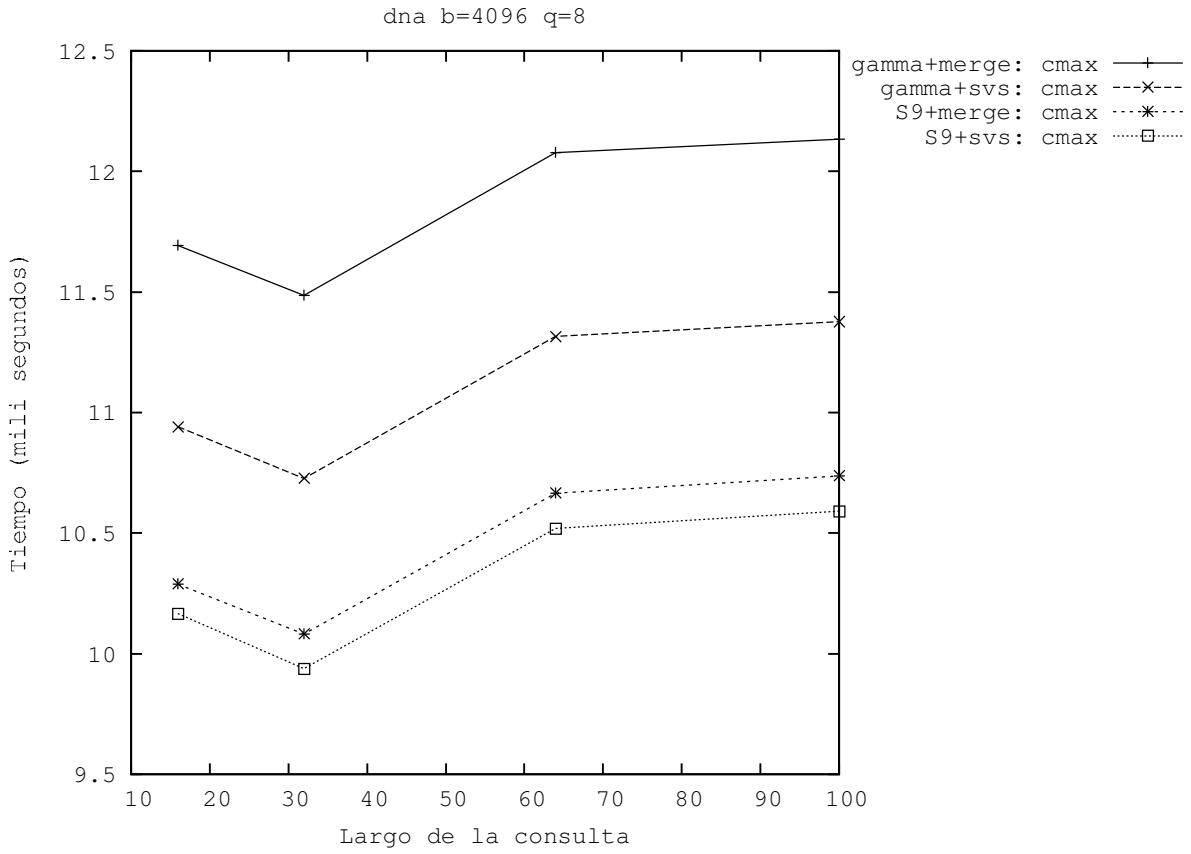

Figura 6.8: Tiempo de búsqueda para *dna*, b=4KB, q=8.

La figura 6.8 muestra tiempos de búsqueda utilizando cobertura máxima con un q mayor. En este caso se obtiene un rendimiento superior a los índices con q=6 y b=1KB. Se aprecia que para todos los algoritmos de compresión e intersección utilizados se obtienen tiempos mejores a los presentados en al figura 6.5.

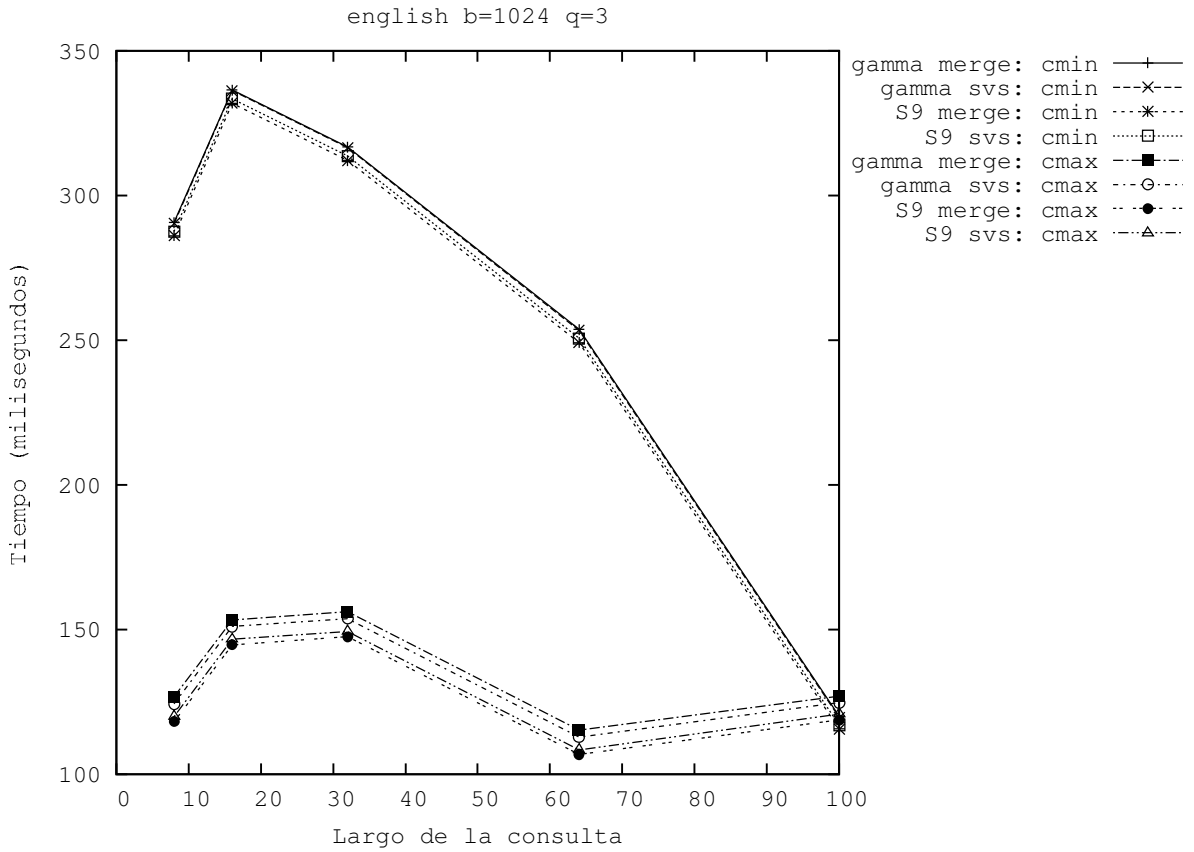

Figura 6.9: Tiempo de búsqueda para *english*, b=1KB, q=3.

El índice presenta un rendimiento muy inferior para búsquedas con *english*, los resultados indican que la búsqueda en bloques de texto es hasta 100 veces más lenta que la etapa de filtrado. Bajo estas condiciones el tipo de cobertura utilizado tiene un gran impacto sobre el rendimiento, tal como se aprecia en la figura 6.9.

Salvo la diferencia entre los tiempos de búsqueda para *english* y *dna*, los resultados fueron muy similares para todas las pruebas.

Para la selección de los índices finales se descartan todas las alternativas con b mayor a 8KB. En el caso de *dna*, para q=8 se descarta b=1KB por el espacio utilizado.

En las pruebas con el texto *english*, para tamaños de bloques pequeños se tienen mejores tiempos que para bloques más grandes, esto ocurre aún si se comparan distintos valores de q. Esto se debe al tiempo de búsqueda en disco. La diferencia entre los tiempos de filtrado y búsqueda disminuyen la relevancia de los algoritmos utilizados para filtrar las ocurrencias. Se descartó S9 por utilizar más espacio que Gamma.

Se seleccionaron tres variantes del índice para comparar las búsquedas en *dna*:

q=8, b=4KB, algoritmo de compresión Gamma, algoritmo de intersección SvS, cobertura máxima. Por tener los tiempos de búsqueda más bajos.

q=6, b=8KB, algoritmo de compresión Gamma, algoritmo de intersección Merge, cobertura máxima. Ya que presenta la mejor compresión.

q=6, b=1KB, algoritmo de compresión Gamma, algoritmo de intersección Merge. Por ofrecer un equilibro entre compresión y tiempo de búsqueda.

Para las pruebas con el texto *english*, se seleccionó q=3, b=1024, algoritmo de compresión Gamma, algoritmo de intersección SvS, cobertura máxima. Por presentar los mejores tiempos de búsqueda.

6.5. Rendimiento del índice

En esta sección se presentan los resultados obtenidos al comparar el índice desarrollado con los índices: Succinct Suffix Array (SSA) y Alphabet-Friendly FM-Index (AF-Index) . Cada uno de los índices se probó con distintos parámetros que afectan el nivel de compresión y la velocidad al responder las consultas.

texto / S_A	16	64	256
english	1,229	0,854	0,761
dna	0,920	0,545	0,451

Cuadro 6.7: Compresión alcanzada por SSA.

texto / S_A	16	64	256
english	1,230	0,855	0,761
dna	1,04	0,668	0,574

Cuadro 6.8: Compresión alcanzada por AF-Index.

Los cuadros 6.7 y 6.8 muestran los niveles de compresión de los índices medidos al utilizar distintos parámetros.

En cuanto al nivel de compresión, los índices invertidos son comparables a los auto-índices comprimidos con los parámetros utilizados (ver tabla 6.6).

A continuación se presenta el tiempo de búsqueda de patrones para todos los índices.

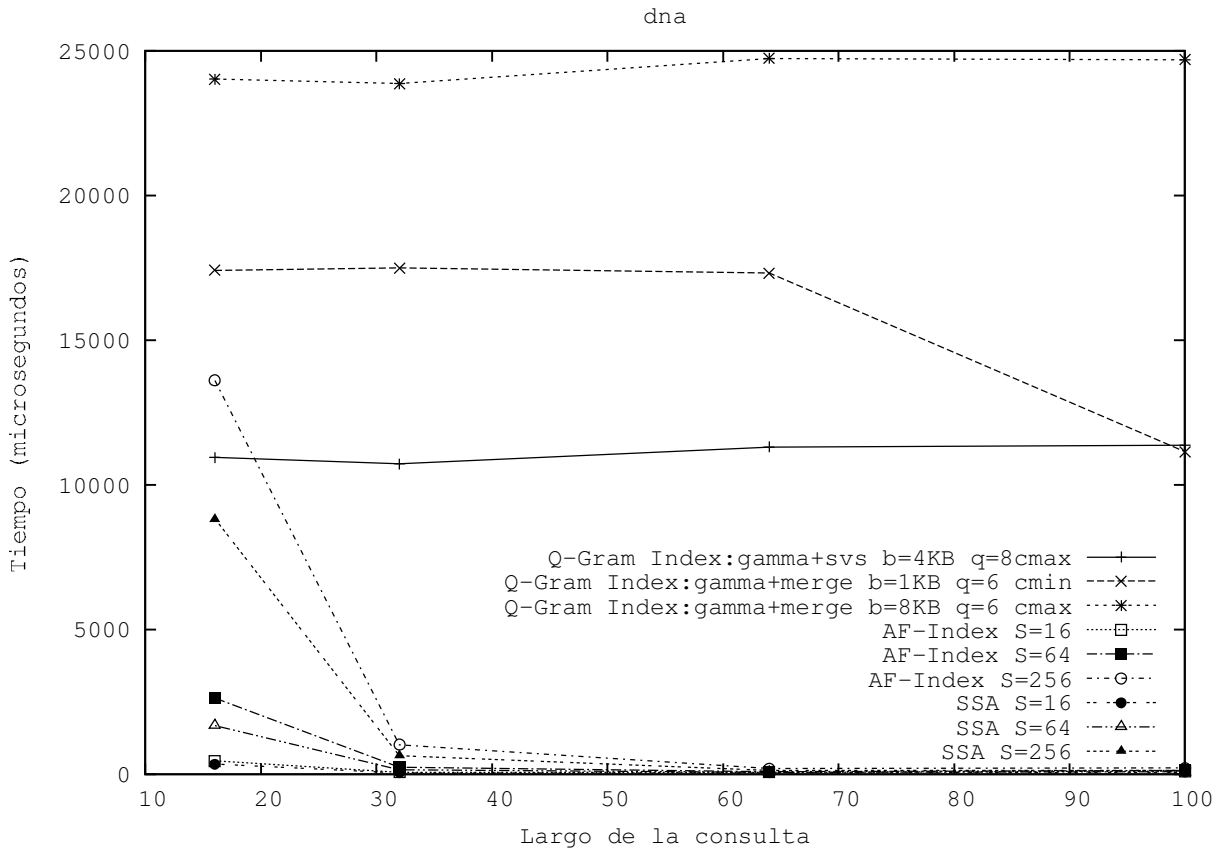

Figura 6.10: Tiempo de búsqueda para *dna*.

La figura 6.10 muestra que el indice de q-gramas no presenta un rendimiento competitivo para patrones de tamaño 16 y mayores. A medida que disminuye el tamaño de los patrones aumenta el tiempo tanto en el AF-Index como en el SSA.

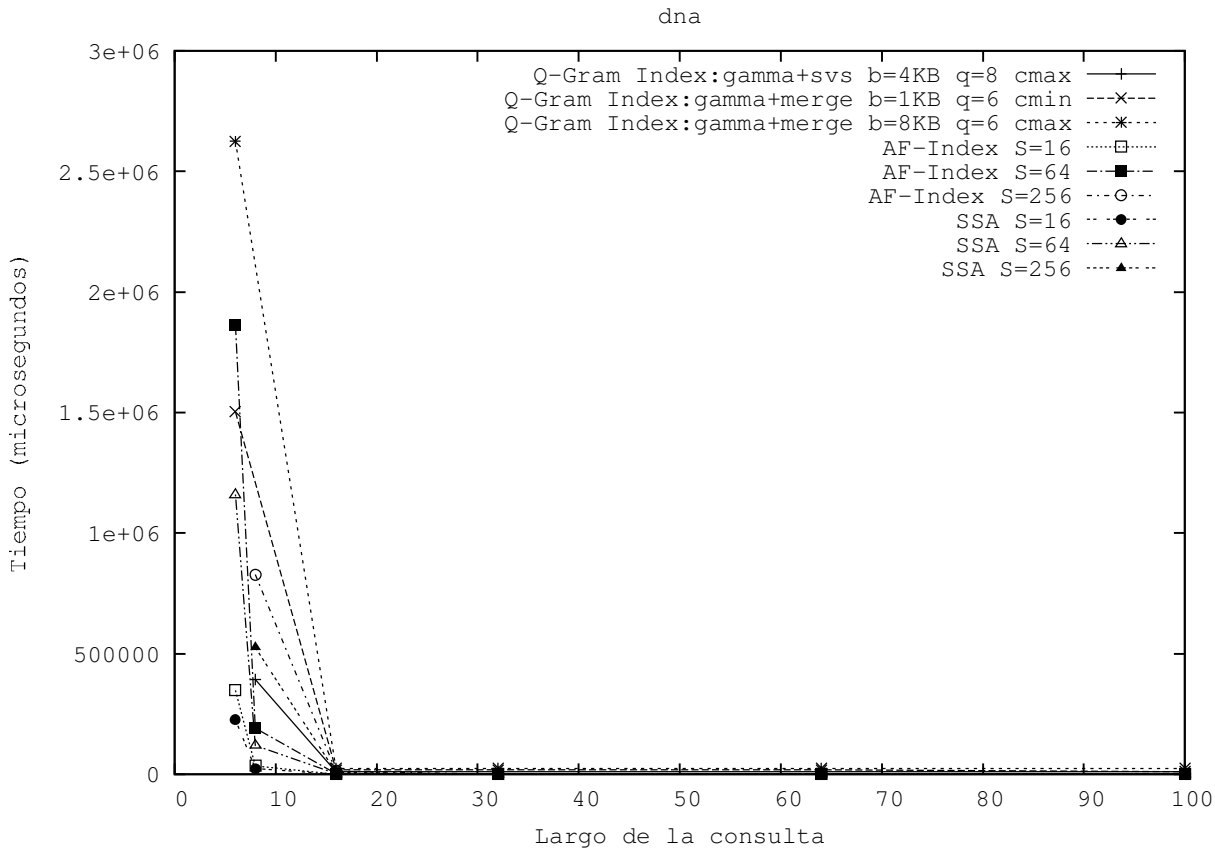

Figura 6.11: Tiempo de búsqueda con m pequeño para *dna*.

En la figura 6.11 se comparan patrones de menor tamaño. En este escenario algunos índices de q-gramas responden más rápido a consultas que los demás índices cuando se utiliza S=64 y S=256.

Mientras más pequeño el patrón, es más probable que aumente el número de ocurrencias en el texto. Esto facilita la búsqueda en el índice de q-gramas. Si se busca un patrón de largo q, entonces prácticamente todos los bloques indicados por la lista asociada al q-grama contiene una o más ocurrencias del patrón, la única posibilidad para que esto no ocurra es que el q-grama se encuentre completamente dentro del traslape del bloque.

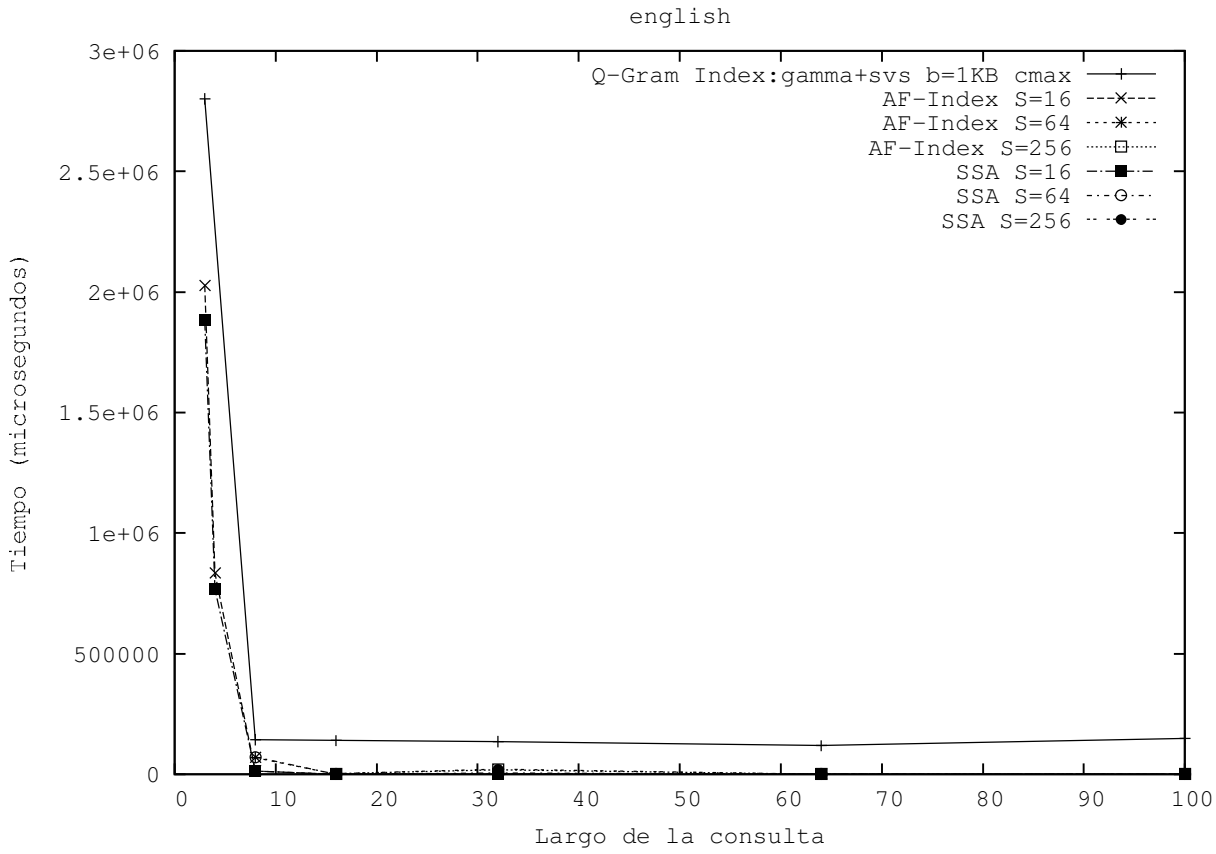

Figura 6.12: Tiempo de búsqueda para *english*.

En el caso de búsquedas en ingles, el índice implementado no ofrece un rendimiento competitivo. Aun para patrones pequeños.

Capítulo 7

Conclusiones

En este trabajo se presentó la implementación de un índice comprimido de q-gramas basado en bloques y se evaluó su desempeño en términos del espacio requerido y el tiempo que tarda al buscar patrones.

Dentro de las variantes evaluadas, aquellas que utilizaban tamaños de bloques más pequeños presentaban el mejor rendimiento. En particular el índice permite buscar patrones de diferentes longitudes en 10 milisegundos en textos de adn utilizando $1,2$ veces el espacio requerido por el texto plano y 20 milisegundos utilizando 0,399 veces el espacio del texto (en ambos casos se incluye el tamaño del texto comprimido) tiempos que se mantienen tanto para patrones grandes como patrones medianos.

En cuanto a la compresión de texto, se concluyó que la mejor alternativa es comprimir el texto completo. Esto permite reducir el tamaño de los traslapes de bloques de forma significativa. Al comprimir mejor los traslapes, es posible utilizar bloques pequeños, donde el tamaño del traslape es significativo en relación al tamaño del bloque.

De acuerdo a los resultados obtenidos, la mayor parte del tiempo que el índice dedica a una búsqueda transcurre en la verificación de ocurrencias de bloques. Esto es esperable ya que leer el texto de la memoria secundaria (aunque se encuentre comprimido) es mucho más lento que recuperarlo de la memoria primaria.

El rendimiento del índice es superado por los índices competitivos de Pizza&Chili, salvo en los casos donde el patrón buscado es de menor tamaño. Para estos casos se mostró experimentalmente que existen variantes del índice que presentan rendimientos comparables al resto de los índices. Por lo tanto, si se necesita buscar patrones pequeños, el índice implementado es una alternativa valida, especialmente si la memoria primaria es limitada.

7.1. Trabajo Futuro

Si bien los resultados indican que para patrones largos el índice tiene un rendimiento inferior a otras implementaciones, es posible mejorar la búsqueda de bloques de distintas formas.

En primer lugar se podría almacenar el texto comprimido en la memoria primaria. Esto aumentaría el rendimiento del índice, aunque en la práctica no sería tan útil ya que aumentaría los requerimientos de memoria primaria, limitando el tamaño de los textos sobre los que se puede construir el índice.

También se puede mejorar el algoritmo de búsqueda. Si se almacena información adicional en el texto comprimido con Re-Pair, podría implementarse un algoritmo para búsqueda en texto comprimido utilizando una técnica similar a la de LZgrep.

Finalmente queda la posibilidad de explorar otros algoritmos de compresión de texto. si se obtienen mejores tazas de compresión, sería posible generar índices con bloques de menor tamaño, o utilizar los algoritmos de filtrado y compresión de mayor velocidad ya que se cuenta con más espacio .

Además de las mejoras, es posible extender el índice e implementar nuevos tipos de consultas. Es posible extender el índice para permitir búsquedas aproximadas. En la búsqueda aproximada se busca un patrón y se especifica la distancia de edición máxima entre el patrón buscado y las ocurrencias que debe recuperar el índice.

Bibliografía

[1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[2] F. Claude. *Estructuras Comprimidas para Grafos de la Web.* Tesis Magister, Universidad de Chile, 2008.

[3] F. Claude, A. Farina, and G. Navarro. Re-pair compression of inverted lists. Nov 2009.

[4] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[5] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.

[6] G. Navarro, E. Sutinen, and J. Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms (JDA)*, 3(2–4):157–175, 2005.

[7] S. J. Puglisi, W. F. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. *13th Symposium on String Processing and Information Retrieval (SPIRE)*, 4209:122–133, 2006.

[8] Sitio web Pizza&Chili. `http://pizzachili.dcc.uchile.cl/`.

[9] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann, 1999.