Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Diseño y Análisis de Algoritmos

Tarea II: Busqueda en Texto

Suffix Array y Automata

Integrantes: Tomas Perry
Juan Andrés Moreno
Profesor: Pablo Barcelo B.
Auxiliar: Ariel Cáceres Reyes
Ayudantes: Jaime Salas T.
Claudio Torres F.

Fecha de realización: 13 de junio de 2017
Fecha de entrega: 13 de junio de 2017
Santiago, Chile

Índice de Contenidos

Lista de Figuras

Lista de Tablas

1. Introducción

Los programas editores de texto se ven frecuentemente enfrentados a la tarea de encontrar una palabra en el archivo que se está editando. Debido a lo recurrente que se hace esta tarea es imperativo encontrar implementaciones eficientes de búsqueda en texto. Entre otras usos se encuentran el matching de secuencias de ADN y los motores de búsqueda en internet.

El problema de **Búsqueda en Texto** se formaliza de la siguiente manera: Se tiene un texto T de largo n, representado como un arreglo de caracteres en cierto alfabeto Σ: $T[1...n]$. Así mismo se tiene un patrón P de largo m que se desea buscar en el texto T: $P[1...m]$. Notamos que $m << n$.

Decimos entonces que el patrón P ocurre con **desplazamiento** s en el texto T si se tiene que $0 \leq s \leq n - m$ y $T[s + 1...s + m] = P[1...m]$. Entonces si P ocurre con desplazamiento s en T se dice que s es un desplazamiento válido. De esta manera el problema de **Búsqueda en Texto** corresponde al problema de encontrar todos los desplazamientos válidos de P sobre T.

En esta tarea se implementaron dos soluciones al problema de **Búsqueda en Texto**:

1. Un enfoque estático basado en **Suffix Array**.
2. Un enfoque dinámico basado en el **Algoritmo de Autómata**.

2. Algoritmo de Autómata

2.1. Hipótesis y Supuestos

Antes de realizar experimentos con la implementación del Autómata fue necesario hacer una limpieza del input. Estos fueron obtenidos de [1]. Se tomó el archivo de 200mb y se cortó para lograr los siguiente fragmentos:

Tabla 1: Archivos de prueba con cantidad de palabras.

Nombre de Archivo	Cantidad de Palabras
15.txt	34512
16.txt	69457
17.txt	138878
18.txt	269648
19.txt	531337
20.txt	1050374
21.txt	2098272

Dentro de estos existen caracteres acentuados, saltos de línea, puntuaciones y letras en mayúsculas. Se procedió a limpiar estos cambiando todas las mayúsculas por minúsculas, reemplazando las ocurrencias de puntuaciones y saltos de línea por espacios y eliminando los caracteres acentuados. De esta manera se definió el alfabeto valido para esta tarea como:

$$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, \ \}$$

Debido a cómo se implemento el contador de palabras y la función que obtiene las $N/10$ palabras de prueba fue conveniente reemplazar todo carácter no en Σ por un espacio.

Luego de esto se pide por enunciado comprobar que se tengan archivos de prueba de largo 2^i con $i \in \{15..,25\}$. La siguiente tabla muestra los largos obtenidos luego del pre procesamiento:

Tabla 2: Archivos de prueba pre procesados con cantidad de palabras.

Nombre de Archivo	Cantidad de Palabras
15.txtCleaned.txt	31798
16.txtCleaned.txt	64145
17.txtCleaned.txt	127455
18.txtCleaned.txt	250854
19.txtCleaned.txt	495318
20.txtCleaned.txt	980111
21.txtCleaned.txt	1956404

Se comprueba entonces que cada archivo de prueba obtenido luego del pre procesamiento tiene una cantidad de palabras $\mathcal{O}(2^i)$ donde i corresponde al numero en el nombre del archivo. La función *textCleaner* en el archivo *MainDFA.java* es la encargada de ejecutar el pre procesamiento.

2.2. Construcción del Autómata

En cátedras se vio un algoritmo para búsqueda en texto basado en el uso de un automata que solo depende del patrón de largo m P buscado. Este autómata queda definido por:

- Conjunto de estados $Q = \{0, 1, ..., m\}$ donde 0 es el estado inicial y m el final. Aquí el estado j está relacionado con el carácter en la posición j del patrón.
- Función de transición δ: $\Sigma \times Q \to Q$ que opera de la siguiente manera: $\delta : (q, a) \to \delta(q, a) = \sigma(P[1 : q]a)$. Donde esto último se traduce en el largo de sufijo más largo de $P[1 : q]a$ que es prefijo de P.

Luego de esto, si se ejecuta el autómata sobre un texto T la cantidad de ocurrencias del patrón P corresponde a la cantidad de veces que el autómata pasa por su estado final. Se puede ver entonces que la construcción del autómata se reduce a encontrar la función de transición, pues recordemos que el pre procesamiento de los textos nos permitió fijar el alfabeto.

El acercamiento *naive* a este problema genera un tiempo de construcción de la función de transición (y por ende del autómata) de $\mathcal{O}(m^3)$. Para esta tarea se implementó un algoritmo que permite construir la función de transición en $\mathcal{O}(m)$. El algoritmo construido corresponde a [2] *Knuth-Morris-Prat* y se basa en las siguientes observaciones:

Al calcular $\delta(q, a) = \sigma(P[1 : q]a)$ consideramos dos casos:

- Si $P[q + 1] = a$ es decir corresponde a un *match*, entonces $\delta(q, a) = q + 1$.
- Si no, entonces $\delta(q, a) = \sigma(P[1 : q]a) = \sigma(P[2 : q]a) \leq q$ y por lo tanto puede ser calculado corriendo el automata parcialmente construido sobre $P[2 : q]a$. Notamos además que esto último no es necesario si tiene una variable X en la cual se va guardando el resultado de correr el automata sobre $P[2 : q]$ a medida que se avanza.

La función *getTransitionFunction* dentro del archivo *MainDFA.java* recibe como parámetro un String que contiene el patron de interés. Aquí se define la función de transición como una matriz w de dimensiones $|\Sigma| \times (|m| + 1)$ de enteros. Notamos entonces que $w[i][j] = k$ significa que al ver carácter Σ_i en T y habiendo *matcheado* hasta el carácter $j - esimo$ de P avanzamos al estado k.

Dentro de la función *getTransitionFunction* se hila un poco mas fino y se distinguen cuatro casos distintos de llenado de matriz:

1. Condiciones iniciales: Se fijan todas las transiciones $w[i][0] = 0$ con $1 \leq i < |\Sigma|$.
2. Transiciones de *match*: Cuando se tiene que $P[q + 1] = a$. Esto en la matriz se traduce en $w[\Sigma_i][i] = i + 1$ con $0 \leq i < |m|$.
3. Transiciones del estado final al estado inicial. Estas permiten que el autómata continue su ejecución sobre el texto T una vez que se encuentra la primera ocurrencia de P.
4. Transiciones de *mismatch*. Cada vez que se tiene que $P[q + 1]! = a$, entonces se tiene que $w[i][j] = w[i][X]$ con $0 \leq i < |\Sigma|$ y $1 \leq j < |m|$.

El código asociado se puede encontrar en el anexo.

2.3. Muestreo de palabras de prueba y contador de palabras

2.3.1. Contador de Palabras

La funcion *getWordCount* en el archivo *MainDFA.java* recibe como parámetro un objeto del tipo *File* el cual contiene el archivo de texto ya pre procesado. Debido a que en el pre procesamiento se reemplazaron todos los caracteres no pertenecientes al alfabeto definido Σ por un espacio resulta conveniente utilizar un *Scanner* de Java. A este se le asigno como delimitador el carácter espacio. De esta manera utilizando el método asociado al *Scanner: next()* se obtienen las palabras una a una. Con esto se incrementa un contador y finalmente se retorna la cantidad de palabras. Cabe destacar que ambas tablas 1 y 2 muestran cantidades de palabras en archivos, las cuales fueron obtenidas utilizando la función *getWordCount*.

2.3.2. Muestreo de palabras de prueba

En el enunciado se pide, luego de efectuar el pre procesamiento, verificar que la cantidad de palabras N en el archivo sea adecuada y luego obtener $N/10$ palabras al azar del archivo. Para esto se consideró que, si bien el tamaño de los archivos no era excesivo y estos sí podían cargarse completamente en memoria, se optó una implementación que no asumiera esto. Para lograr esto entonces se utilizó un *Random Access File* el cual permite, como su nombre lo dice, acceder a distintas partes del archivo, indicando un offset.

Se generaron offsets de manera aleatoria entonces entre 0 y el largo del input en bytes. Una vez posicionado el cursor sobre este offset en el archivo se guardó la primera palabra completa que se encontrase. El criterio utilizado para definir una palabra fue: una palabra corresponde a una secuencia de caracteres entre dos espacios.

Para evitar problemas en el caso de que se indique un offset que apunte hacia la última palabra del texto, al pre procesar se añade un carácter de espacio al final del input. La función encargada de efectuar el muestreo es *getTestWords* y esta dentro del archivo *MainDFA.java*.

2.4. Contador de Apariciones

Por último, resta explicar como utilizando las funciones descritas anteriormente se lleva a cabo la *Búsqueda en Texto* utilizando el Algoritmo de Autómata. La función *countAppearances* recibe como parámetros el patrón a buscar y el texto donde se debe buscar el patrón. Mediante un *BufferedReader* se lee el texto carácter a carácter. Se tiene una variable que registra el estado del Autómata en cada paso y esta se va actualizando según el estado presente y el carácter que se este leyendo en ese momento. Cabe destacar que si se encuentra el carácter correspondiente al espacio, se debe volver al autómata al estadio inicial cero. Dado que el Autómata tiene un estado q_j para el carácter $j - esimo$ del patrón, se tiene que cada vez que el estado sea igual al largo del patron entonces llegamos al final de este y debemos incrementar la cantidad de ocurrencias en uno. Una vez que se termina el texto de entrada se retornan la cantidad de ocurrencias encontradas.

2.5. Pruebas

Para cada uno de los inputs pre procesados en la tabla 2 se obtienen las $N/10$ palabras de prueba. Para cada una de estas se mide el tiempo de construcción del autómata, el cual se define como el tiempo de construcción de la función de transición, ademas del tiempo que toma en encontrar todas las ocurrencias. Se escribe entonces en un archivo llamado *results.csv* una línea por palabra, la cual contiene:

- La palabra (patron) a buscar.
- Largo de la palabra.
- Tiempo de construcción de función de transición.
- Tiempo que toma encontrar todas las ocurrencias de esta en el texto.
- Largo del texto (Cantidad de palabras).

Cabe destacar que debido al tiempo se tomaron $N/100$ palabras de prueba por input.

Para el caso del Automata los tiempos de búsqueda sobre un texto T dependen principalmente del largo N de este además del largo m del patrón P a buscar. Debido a que los tamaños de los textos utilizados para probar van aumentando al doble su tamaño se espera que ocurra lo mismo con los tiempos de búsqueda+construcción.

El siguiente gráfico muestra los tiempos de búsqueda mas construcción para distintos tamaños de patrones y sobre todos los textos de prueba:

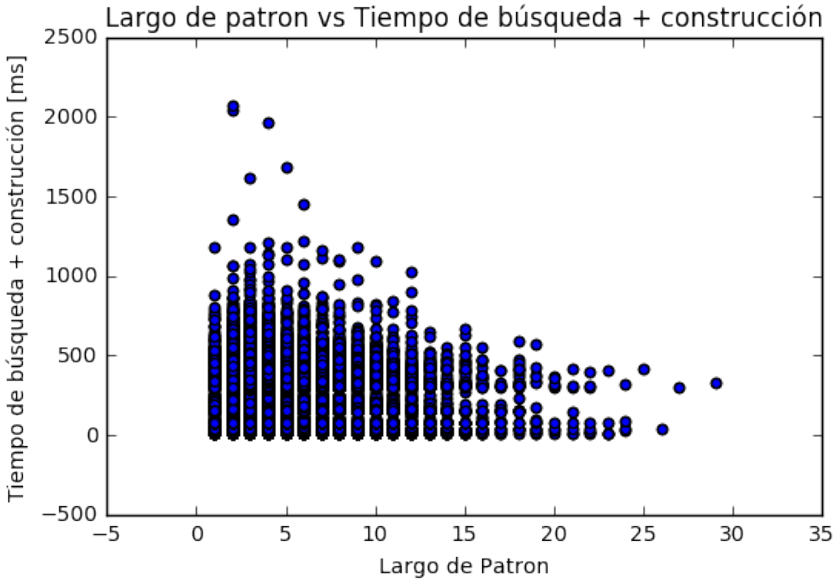

Figura 1: Largo de patrón vs tiempo de construcción + búsqueda.

Este ultimo gráfico confirma la noción de que para dos patrones del mismo largo pueden existir tiempos de búsqueda distintos. Esto sucede al buscar patrones del mismo largo, o el mismo patron en textos de largos distintos.

Ademas se debe verificar que efectivamente el tiempo de construcción del Autómata para un patrón P de largo m sea $\mathcal{O}(|\Sigma|m)$. En nuestro caso, como se definió el alfabeto $|\Sigma| = 27$. El siguiente gráfico muestra los tiempos de construcción de los distintos patrones buscados en las pruebas:

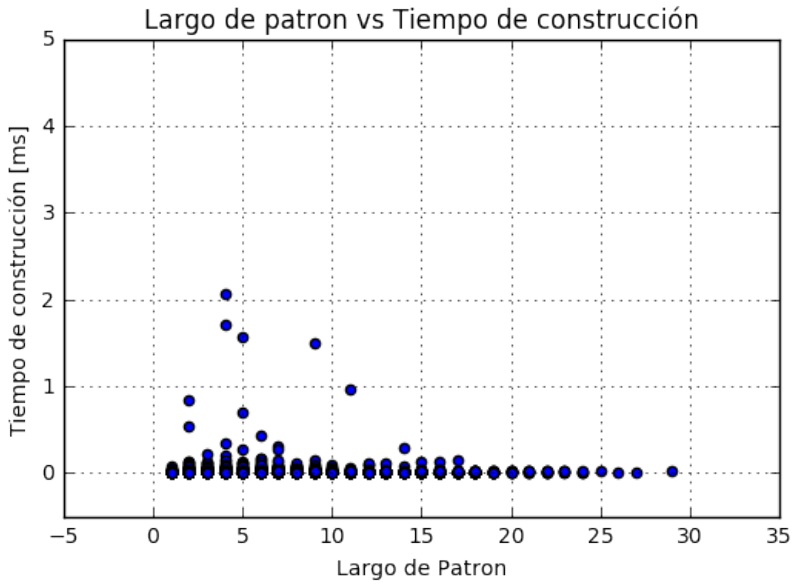

Figura 2: Largo de patron vs tiempo de construcción.

Se puede ver entonces de la figura 2 que no existen diferencias significativas en tiempos de construcción para patrones de un mismo largo. Los datos presentados sugieren que $\mathcal{O}(\Sigma m)$ parece ser una cota no muy ajustada. Las leves diferencias que se presentan para patrones de un mismo largo se le atribuyen a posibles anomalías en el funcionamiento del equipo al efectuar las pruebas o variaciones en la carga de este a lo largo de las pruebas. Por último cabe destacar que los datos recolectados y con los que se generaron los gráficos anteriores se adjuntan a este informe. (results.csv)

3. Suffix Array

Antes de desarrollar el método, es necesario entender qué es.

El Suffix Array, como su nombre lo dice, es un arreglo que, si bien no es de sufijos, es representativo de estos. Este se puede expresar como:

$$SA_T[i] = k \leftrightarrow T_k \text{ es el i-esimo substring en orden lexicográfico de T}$$

Donde T es el string y SA_T es el Suffix Array de este.

A partir de este arreglo uno puede buscar si un substring pertenece o no a un string que, en este caso, es una archivo entero.

3.1. Hipótesis y Supuestos

Al igual que en el caso del Autómata, es necesario preprocesar el texto y asegurarse de ser apto para la construcción del Suffix Array, removiendo caracteres fuera de Σ y reemplazandolos por espacios. Está incluído en el código

Para la construcción de este método, se ocupó C+. Decisión hecha a partir del autor principal teniendo mejor manejo de este lenguaje.

3.2. Construcción del Suffix Array

Para poder construir el arreglo se utiliza el algoritmo de Karkkainen y Sanders, el cuál utiliza triplas del string original y guiándonos de [3].

1. Rellenar el string con un caractér no ocupado para tener una cantidad múltiplo de 5.
2. Formar 3 subconjuntos donde se almacenan las triplas $T_iT_{i+1}T_{i+2}$, donde i es 1 mod 3, 2 mod 3 y 0 mod 3.
3. Asignar a cada tripla un caracter que las represente manteniendo el orden lexicográfico de estas[1]
4. Concatenar los caracteres de los arreglos para i = 1 mod 3, 2 mod 3 y obtener su Suffix Array. Esto se realiza de forma recursiva, repitiendo el método anterior, pero con substrings de largo 2 y ordenándolos.
5. Obtener el Suffix Array del arreglo para i = 0 mod 3, donde se podía ocupar el arreglo para i = 1 mod 3 para ahorrar una comparación.
6. Mezclar el Suffix Array de los arreglos concatenados y el del arreglo para i = 0 mod 3. Esto se lograba comparando el caractér más importante y los rangos obtenidos en el punto 3.

[1] Aquí se utilizó RadixSort de las triplas y luego se asignó un caractér o "rango".

Una vez construido, la búsqueda puede ser estática, como se verá ahora. Adjunto al informe se puede encontrar el código para la construcción y la búsqueda, con el nombre *SuffixArray.cpp* y su respectivo *Makefile*.

3.3. Búsqueda

Una vez construido el Suffix Array, si bíen los valores no están ordenados, representan substrings ordenados, por lo que se puede realizar búsqueda binaria al comparar el patrón a encontrar con los strings ordenados y si el patrón está contenido en un substring, significa que es un substring del texto. Teniendo un largo m el substring y el texto siendo de largo n, la búsqueda termina siendo de orden $\Theta(mlog(n))$.

Además, debido al orden del Suffix Array, si tenemos el primer y último substring en el que el patrón aparece, sabemos que todos los substrings entre estos dos contienen al patrón, por lo que la resta del último y el primer índice equivalen a la cantidad de apariciones.

3.4. Pruebas

Al igual con el autómata, existen múltiples archivos de tamaños variables. A partir de cada uno se construye un Suffix Array y se evalúa con N/10 substrings que existen en los textos, midiendo tiempo de construcción y de búsqueda. Se elaboró una interfaz para entregar el nombre del archivo, el número de tests y los patrones a evaluar o correr un test masivo de todos los archivos.

3.5. Resultados

Desafortunadamente, la construcción del Suffix Array está incompleta, por lo que no se pudo comparar su rendimiento con el del autómata. La construcción del Suffix Array de los arreglos para i = 1 mod 3, 2 mod 3 no se obtuvo, al existir problemas en los accesos de memoria que no se pudieron resolver en el tiempo estipulado.

4. Conclusiones

Al construir el Automata de búsqueda en texto para cada uno de los diferentes patrones encontrados de manera aleatoria en los textos de prueba generados se pudo comprobar que efectivamente el algoritmo de *Knuth-Morris-Prat* logra la construcción de este en tiempo $\mathcal{O}(m)$ considerando que en esta tarea se tomo un alfabeto Σ fijo. Esto queda expuesto en el gráfico de la figura 2 de la sección 2.5. Por otro lado se logro comprobar ademas que el tiempo de búsqueda para cualquier patron P es casi independiente del largo de este, y que efectivamente este es $\mathcal{O}(N)$ donde N es el largo del texto. Esto ultimo queda expuesto en la figura 1 de la sección 2.5.

5. Referencias

[1] PizzaChili Corpus. *Compressed Indexes and their Testbeds*
http://pizzachili.dcc.uchile.cl/texts/nlang/

[2] Robert Sedgewick, Kevin Wayne. *Knuth-Morris-Pratt Construction.*
1 de Diciembre de 2010.
http://www.cs.princeton.edu/courses/archive/spring11/cos226/demo/53KnuthMorrisPratt.pdf

[3] Graduate Level Algorithm Design and Analysis - Gusfield *Linear-time algorithm to build a suffix array for a string S*
http://web.cs.ucdavis.edu/~gusfield/cs222f07/lineartimesuffixarray.wmv