Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Depto. de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Tarea 2

Integrantes : Rodrigo Delgado
Belisario Panay
Gabriel Sanhueza
Profesor : Gonzalo Navarro
Ayudantes : Sebastián Ferrada
Willy Maikowski
Auxiliar : Jorge Bahamondes

Índice

1. Introducción

Un *suffix tree* es un *trie* (Trie: Estructura ordenada en forma de árbol) comprimido, el cual contiene todos los sufijos de un texto dado como sus llaves, y sus posiciones en el texto como sus valores.
En el presente informe se muestra el diseño, implementación y experimentación del algoritmo de Ukkonen, el cual es un algoritmo de orden lineal para la creación de Suffix Trees, formada por Nodos que poseen información interna y referencias a sus hijos.
En particular el algoritmo de Ukkonen para un Suffix Tree almacena los sufijos de un string en forma de árbol, de modo que cada arco contiene los caracteres para formar una palabra, y agregando caracteres sucesivos hasta que el árbol está completo. Por tanto, al momento de buscar, si se llega a una hoja (nodo terminal) es porque se recorrieron los arcos necesarios para formar un sufijo.
La idea de la tarea es que el algoritmo a desarrollar es *on-line*, de orden lineal ($O(n)$) y con una implementación más sencilla con respecto a algoritmos similares (algoritmo de Weiner y algoritmo de McCreight).

1.1. Problema a resolver

Un algoritmo estándar de creación de Suffix Tree toma tiempo $O(n^3)$, mientras que el algoritmo de Ukkonen toma tiempo $O(n)$. El problema consiste en realizar los pasos necesarios para la buena implementación del algoritmo, ya que pequeños errores en código pueden producir un algoritmo de mayor orden de magnitud y con ello se falla en el objetivo.
Los pasos para realizarlos estan detallados en el enunciado de la tarea y en el *paper* de Esko Ukkonen, creador del algoritmo.

1.2. Hipótesis

Si bien es un algoritmo lineal, el algoritmo posee una constante que hace que el tiempo pueda crecer de manera rapida. Además, como el algoritmo será escrito en Java como lenguaje, la eficiencia del algoritmo se podría ver reducida, dado que el *garbage collector* usa la CPU para realizar sus funciones. Se espera demostrar que la constante del algoritmo varía con respecto al largo por motivos externos al algoritmo, lo cual se puede demostrar si para números pequeños el algoritmo sí demuestra un comportamiento lineal.

2. Diseño Teórico

// TODO: Mostrar las distintas clases creadas y para qué sirven (cada clase puede tener su propia subsection)
Para explicar el diseño teórico del algoritmo primero se deben definir algunos conceptos:

- Suffix Tree Implicit es un Suffix Tree al cual se le remueven sus simbolos terminales (en algunos casos se utiliza el signo $ como terminal)

2.1. Main

// TODO: Explicar que hicimos en main, i.e. hacer un logger, cargar texto, como procesamos el texto, etc

2.2. Ukkonen

// TODO: Explicar cómo lo hicimos para hacer que el algoritmo funcionara

2.3. Otras clases

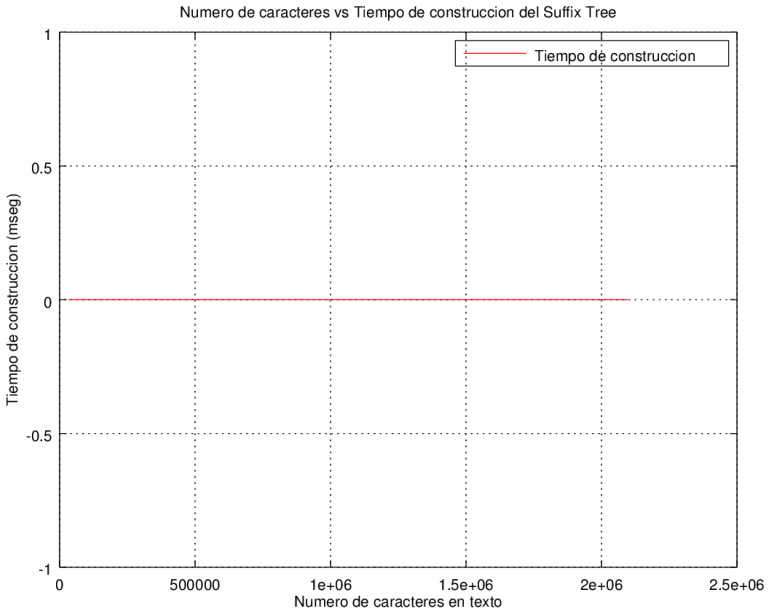
// TODO: Limpiar código de las clases que no se usan, para poner aquí las que sí usamos

3. Presentación de los Resultados

3.1. Tiempo de Creación del Suffix Tree

// TODO: Mostrar cuanto se tarda en crear un suffix tree, para N = 2 ** 15..21 palabras (aprox).

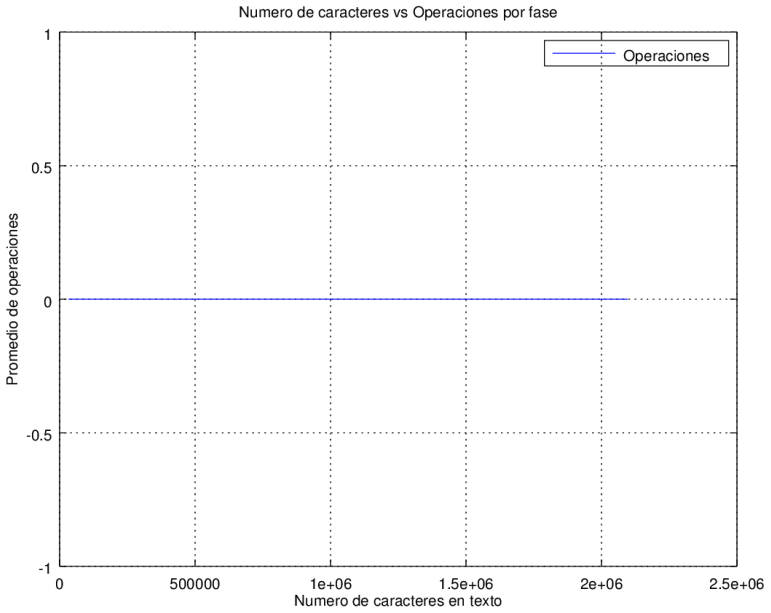
Largo del texto	Tiempo de Construcción	Operaciones por fase
2^{15}	X	X
2^{16}	X	X
2^{17}	X	X
2^{18}	X	X
2^{19}	X	X
2^{20}	X	X
2^{21}	X	X

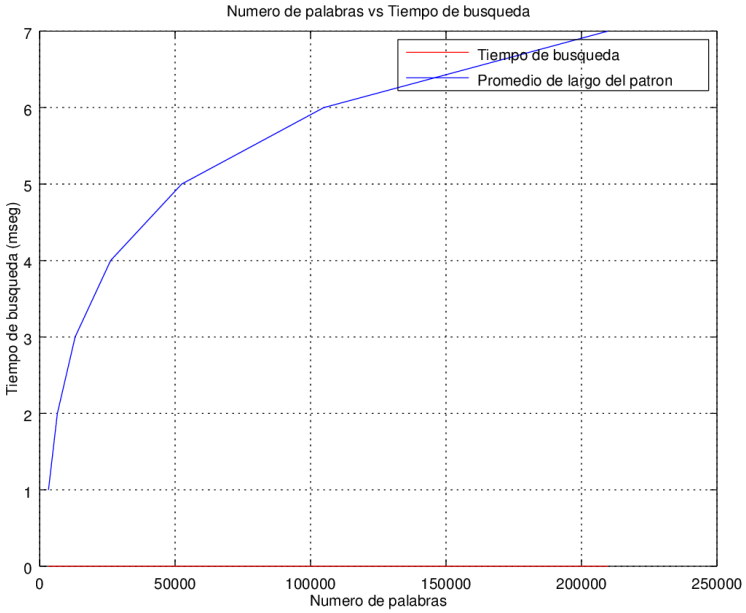


3.2. Desempeño de operación *Buscar*

// TODO: Mostrar cuanto se demora en buscar N = (2 ** 15..21) / 10 palabras (aprox).

Número de palabras	Largo promedio del patrón	Tiempo de búsqueda
2^{15}	X	X
2^{16}	X	X
2^{17}	X	X
2^{18}	X	X
2^{19}	X	X
2^{20}	X	X
2^{21}	X	X




Numero de palabras vs Tiempo de busqueda
Tiempo de busqueda
Promedio de largo del patron
Tiempo de busqueda (mseg)
Numero de palabras
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000

4. Análisis y Conclusiones

4.1. Construcción del Suffix Tree

// TODO: Verificar si el Suffix tree fue lineal o no

4.2. Búsqueda en el Suffix Tree

// TODO: Verificar si los tiempos están acotados o se fueron a la chu*** cada vez que aumentamos el número

4.3. Conclusiones

// TODO: Resumir lo “aprendido” (?)