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

2.1. Main

En Main se crea un archivo de *logging* para registrar el tiempo de creación del Suffix Tree y los resultados de búsqueda, a partir de un texto leído desde el disco. En particular:

- El texto leído se preprocesa (se eliminan puntuaciones, espacios, saltos de línea y todo lo que no corresponda al *regex* [a-ZA-Z]).
- Se crea el Suffix Tree usando el algoritmo de Ukkonen.
- Se registra el tiempo de creación.
- Se generan palabras aleatorias del texto, para buscarlas usando el Suffix Tree.
- Se registran los resultados de búsqueda.

2.2. Ukkonen

Clase principal del algoritmo. Aquí se realiza la totalidad de la ejecución del algoritmo de Ukkonen.
Posee 5 métodos auxiliares:

- run(): Crea el Suffix Tree correspondiente, usando los siguientes métodos auxiliares.
- getPath(char s, Node n): Retorna el camino desde el nodo n, con el caracter s.
- search(String suffix, Node root): Busca el String *suffix* en el nodo *root*
- getSuffixes(Node root, String suffix, int count): Método auxiliar para search. Busca recursivamente en los nodos por el sufijo *suffix* usando el número *count* (usado para los caminos).
- getLeafPath(Node n, int count): Retorna el camino a partir del nodo n, recorriendo *count* nodos internos.

2.3. Last

Esta clase es usada para mostrar el final de un camino.

2.4. Node

Esta clase se usa para almacenar la información correspondiente a cada sufijo. Puede ser o no ser una hoja. Es clase padre de *InternalNode*.

2.5. InternalNode

Esta clase extiende de *Node* para usarse como nodo interno (i.e., explícitamente **no es** una hoja).

2.6. TextPreprocessor

Esta clase recibe un texto como String y devuelve solamente los caracteres que coinciden con el *regex* [a-zA-Z]. El resto de los caracteres se eliminan (entre los cuales están los saltos de línea, las puntuaciones y los espacios)

2.7. Logger

Esta clase recibe un nombre de archivo y escribe los datos que se le entregan a dicho archivo. Sirve para registrar la información necesaria para generar los gráficos del informe.

3. Presentación de los Resultados

3.1. Tiempo de Creación del Suffix Tree

Los resultados para los tiempos de construcción del Suffix Tree usando nuestra implementación son los siguientes:

Largo del texto	Tiempo de Construcción (mseg)
2^{15}	5
2^{16}	3
2^{17}	3
2^{18}	6
2^{19}	11
2^{20}	13
2^{21}	21
2^{22}	44
2^{23}	93
2^{24}	167
2^{25}	329

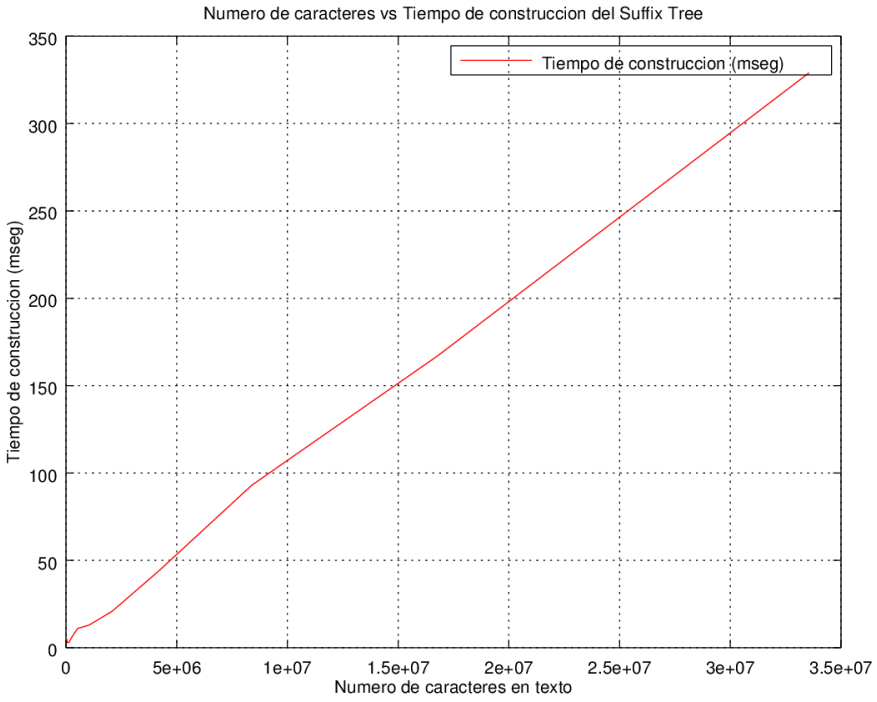

Figura 1: Tiempo de creación del Suffix Tree para N = $2^{15.,25}$

3.2. Desempeño de operación *Buscar*

Los resultados para los tiempos de búsqueda en el Suffix Tree son los siguientes:

Número de palabras	Largo promedio del patrón	Tiempo de búsqueda (nano-segundos)
2^{15}	5.36	2671.75
2^{16}	5.31	1109.40
2^{17}	5.52	259.94
2^{18}	5.39	296.97
2^{19}	5.35	254.40
2^{20}	4.76	268.31
2^{21}	4.50	246.81
2^{22}	4.43	245.39
2^{23}	4.37	254.88
2^{24}	4.42	253.84
2^{25}	4.47	261.40

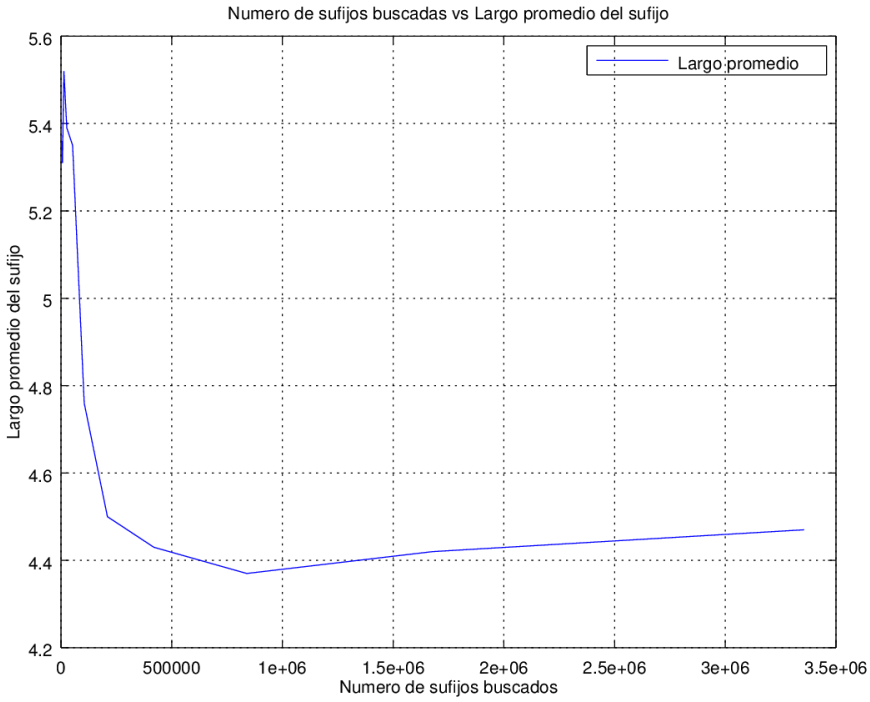

Figura 2: Largo promedio del patrón

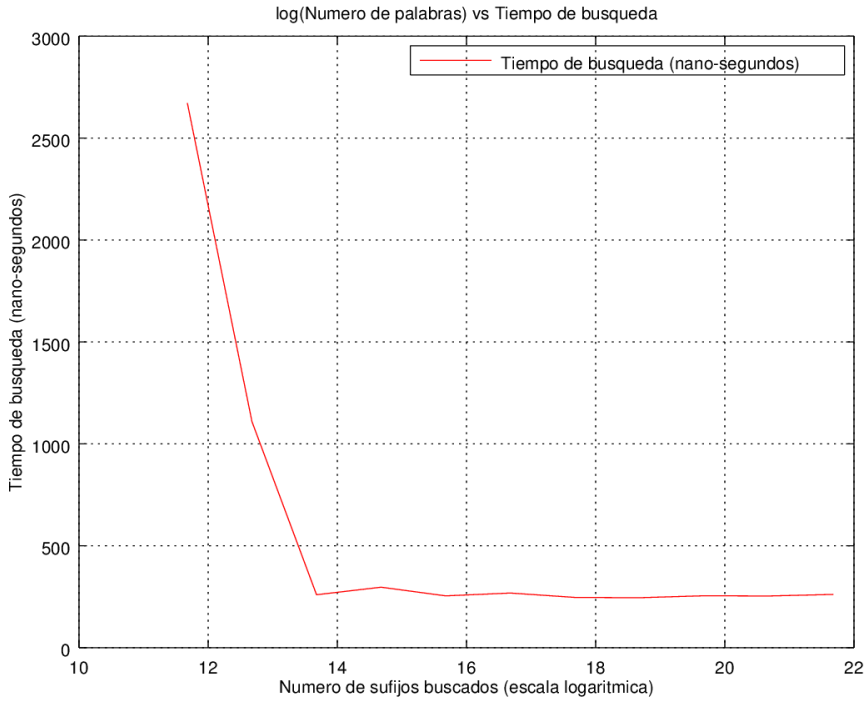

Figura 3: Tiempo de búsqueda

4. Análisis y Conclusiones

4.1. Construcción del Suffix Tree

Nuestra implementación funciona bien cuando se usan palabras cortas, dado que es posible construir el Suffix Tree en tiempo $O(n)$ (puesto que al duplicar el largo de la palabra, también se duplicaba el tiempo en nano-segundos para construirlo).
Nuestra hipótesis de que el *garbage collector* de Java interferiría con nuestros resultados es correcta, dado que cada vez que se corría el algoritmo para una cantidad fija de caracteres variaba notablemente. Aún así, su interferencia es muy baja como para considerarla válida, por lo que solo lo explicamos por motivos de completitud.
Tomando este caso, encontramos que el algoritmo de Ukkonen tarda (por ejemplo) 5 milisegundos para $N = 2^{15}$ caracteres, y se va duplicando (aproximadamente) cada vez que duplicamos la cantidad de caracteres, por lo que la implementación parece haber sido conseguida en tiempo $O(n)$.
A último momento, nos dimos cuenta que nuestra implementación tenía problemas al momento de ejecutarse con textos largos, dado que no creaba correctamente los nodos internos. De esta forma el arbol se creaba demasiado rápido como para ser coherente con la cantidad de enlaces que deberían existir en el Suffix Tree.

4.2. Búsqueda en el Suffix Tree

Encontramos que la búsqueda en el Suffix Tree (respecto a nuestra implementación) cae notablemente, lo que probablemente se debe a que mientras más sufijos se tienen, más sufijos se descartan al momento de buscar la respuesta final. Notar que en nuestra implementación, un **break** en el código *bypassea* el *loop*, cuando en el *loop* se debería salir solo cuando no quedan más sufijos por procesar.
En resumen, mientras más sufijos hay, más quedan sin procesar al momento de buscar, por lo que la cantidad de pasadas es menor.

4.3. Conclusiones

Creemos que el problema en la creación de nodos internos se debe a la creación errónea de los *suffix links* o al seguimiento incorrecto de éstos. Las funciones de conteo están correctas para Suffix Trees bien construídos (i.e. de tamaño pequeño en nuestra implementación), pero pueden mejorarse debido a que se usa recursión para su implementación. Si hubieramos construído correctamente el Suffix Tree con los textos extensos de prueba, probablemente lanzaría un *Stack Overflow*, por lo que en ese caso se cambiaría a una implementación iterativa.