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

Tarea 3

Integrantes : Americo Ferrada
Belisario Panay
Profesor : Pablo Barcelo
Ayudantes : Claudio Torres
Jaime Salas
Auxiliar : Ariel Cáceres

Índice

1. Introducción

En el presente informe se muestra el diseño, implementación y exvaluación y comparación en la práctica de tres enfoques para encontrar una solución aproximada dle problema de Cubrimiento de Vértices Mínimo, el primero la 2-aproximación vista en clases, el segundo la heurística vista en clase auxiliar y por ultimo un tercer enfoque que utiliza ambos enfoques.

Para esta tarea se comprarán el tamaño de los resultados entregados por las tres aproximaciones antes explicados. Para esto se correran los experimentos en grafos de distintos tamaños, creando grafos de manera aleatoria con una probabilidad de que dos vertices esten conectados, para así obtener promedios confiables.

1.1. Problema a resolver

Encontrar el Cubrimiento por Vértices Mínimo es un problema np-Completo por lo que no puede ser usado en la práctica, para esto se ocupan los algoritmos aproximados que entregan una solucion a este problema con ciertas garantias (distancia máxima del optimo).

1.2. Hipótesis

En general se espera que la cantidad de vertices encontrados por los tres algoritmos, aumente en función del valor de p, esto es, mientras exista una probabilidad mayor de que una arista pertenezca al grafo, mayor será el tamaño de las soluciones.

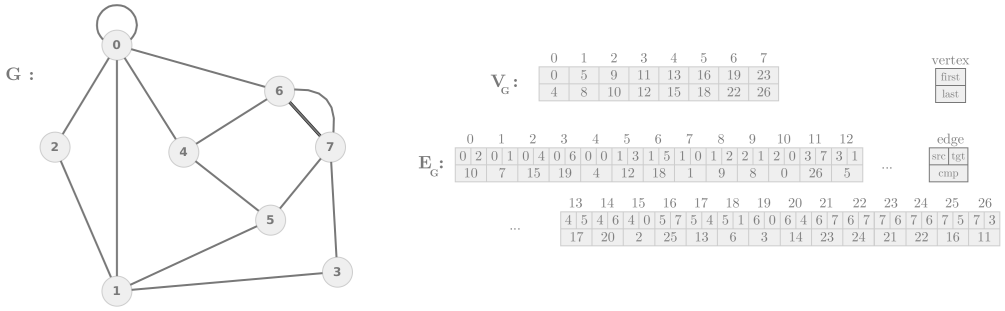
La 2-aproximación tomará menor tiempo de ejecución que las otras dos aproximaciones, pero entregará mayor cantidad de vertices que la solución de la 2-Aproximacion mejorada.

Para el caso de la Heurística, se espera que la cantidad de vertices entregados sean dependientes de la entrada.

Por último para la 2-Aproximación mejorada, se cree que ocupando parte de los dos enfoques anteriores, entregará mejores resultados que las otras aproximaciones, aunque se espera que no se diferencie mucho de la 2-Aproximación.

2. Diseño Teórico

En esta sección se mostrará la estructura utilizada para representar el grafo, además de la descripción de las clases utilizadas en la implementación.
Para representar el grafo, se dicidió utilizar la estructura presentada por el profesor José Fuentes, el cual representa el grafo con dos arreglos, un arreglo de aristas E_G (una lista de adyacencia), las cuales poseen una arista y otra complementaria (ej: (u,v) y (v,u)), cada arista posee un *source* o src (vertice del que proviene) y un *target* o tgt (vertice al que va), además de que cada una posee el indice de su arista complementaria (cmp). Tambien posee un arreglo de vertices V_G, donde cada vertice posee el inicio y final de donde se encuentra en la lista de aristas. Para una mayor comprensión se muestra la siguiente figura.



Para los experimentos se pidió utiliar cinco distintas probabilidades para la probabilidad de que exista una arista, se utilizaron los siguientes valores

p
0,00005
0,0001
0,0005
0,001
0,05

Que para todo $n = 2^i$ con $i \in 10, 11, ..., 20$, p se encuentra en el intervalo $[1/n, 1]$ como se pidió.

2.1. Main

En Main se crean los archivos donde se registrán los tiempos de creación y la búsqueda del patron en el suffix array y el automata, a partir de textos de distintos tamaños leidos desde disco. En particular:

- El texto leído se preprocesa (se eliminan puntuaciones, caracteres especiales, espacios, saltos de línea y todo lo que no corresponda, lo que resulta en un alfabeto que va desde **a** hasta **z**.
- Durante el preproceso se toman al azar distintas palabras pertenecientes al texto.
- Se crea el suffix array sobre un texto, y se buscan las apariciones de las palabras random que se obtuvieron el *TextCleaner*, midiendo los tiempos y registrandolos en los archivos correspondientes.
- Se crea un automata por cada una de las palabras random obtenidas de la preproceso del texto y se hace correr el automata sobre el texto, registrando el tiempo y escribiendolo en el archivo correspondiente.
- Esto es repetido para textos de distintos tamaños.

2.2. Graph

Preprocesa el texto para eliminar todos los caracteres que no serán utilizados en los experimentos, además elige aleatoriamente palabras pertenecientes al texto.
Posee 3 métodos auxiliares:

- clean(): Elimina del texto todos los caracteres que no serán utilizados.
- getRandomWord(ArrayList¡String¿words): Dato una lista elige un elemento de manera aleatoria.
- getNextRandom(): Entrega palabras del texto, como una *queue*.

2.3. Vertex

Crea el suffix array y también realiza la busqueda de un patrón sobre este.

- constructArray(int[] text, int n, int alphabet): Recive un texto representado con enteros y el tamaño del alfabeto, retorna el arreglo de sufijos (como arreglo de enteros).
- sort(int[] a, int[] array, int n, int buckets) : Radix Sort de una lista de triplas de caracteres en un alfabeto, usando counting sort.
- getNextRandom(): Entrega palabras del texto, como una *queue*.
- findPattern(String pattern): Esta funcion encuentra las posiciones en el texto donde se puede encontrar el patron, haciendo busqueda binaria en el suffixArray y en cada busqueda binaria comparando el patron con el sufijo, caracter a caracter.

2.4. Edge

Esta clase crea un automata recibiendo un *String* y crea la tabla de transiciones.

- getCantRepeticiones(): Entrega la cantidad de repeticiones del patrón en el texto.

2.5. GraphGenerator

Esta clase crea un automata recibiendo un *String* y crea la tabla de transiciones.

- getCantRepeticiones(): Entrega la cantidad de repeticiones del patrón en el texto.

2.6. TwoAproximation

Esta clase crea un automata recibiendo un *String* y crea la tabla de transiciones.

- getCantRepeticiones(): Entrega la cantidad de repeticiones del patrón en el texto.

2.7. MaximumDegreeHeuristic

Esta clase crea un automata recibiendo un *String* y crea la tabla de transiciones.

- getCantRepeticiones(): Entrega la cantidad de repeticiones del patrón en el texto.

2.8. ImprovedTwoAproximation

Esta clase crea un automata recibiendo un *String* y crea la tabla de transiciones.

- getCantRepeticiones(): Entrega la cantidad de repeticiones del patrón en el texto.

3. Presentación de los Resultados

3.1. Tiempo de Creación y Búsqueda del Suffix Array

Los resultados para los tiempos de construcción del Suffix Array, fueron los siguientes.

Tamaño del texto (Kb) (	Tiempo de Construcción (ms)
250	210,398
500	384,365
1000	562,266
2000	2345,070
4000	9040,519
8000	25547,724
16000	66096,574
32000	96510,139
64000	209359,431

Debido al mayor uso de memoria del Suffix Array en comparaión al Automata, no se pudieron realizar experimentos para textos más grandes.

Para el caso de la búsqueda de una palabra el texto, los resultados fueron los siguientes.

Tamaño del texto (Kb) (	Tiempo de Búsqueda (ms)
250	3,426
500	7,713
1000	16,271
2000	37,243
4000	104,214
8000	284,076
16000	513,799
32000	964,349
64000	1951,790

Se gráficaron estás dos tablas, los resultados son los siguientes.

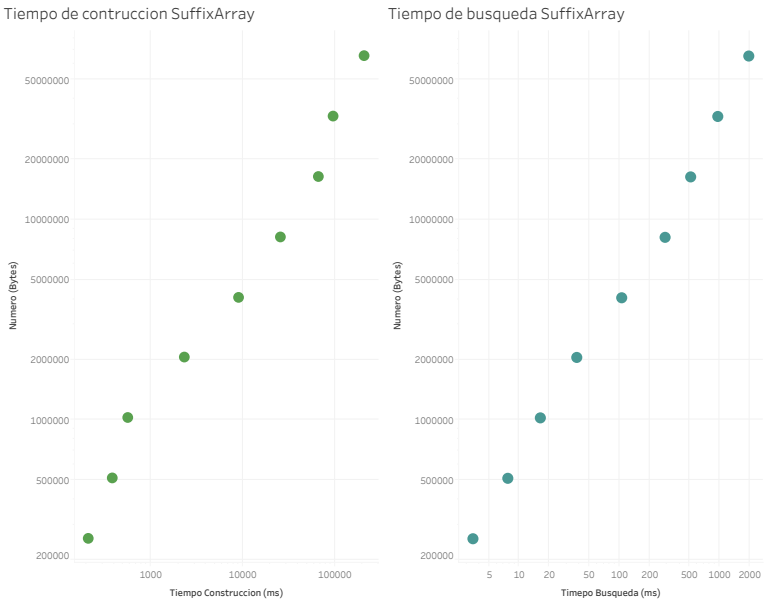


Figura 1: Gráficos del tiempo de creación y búsqueda en el Suffix Array.

3.2. Tiempo de Creación y Búsqueda del Automata

Los resultados para los tiempos de construcción del Automata, fueron los siguientes.

Tamaño del texto (Kb) (	Tiempo de Construcción (ms)
250	0,014
500	0,011
1000	0,011
2000	0.012
4000	0.102
8000	0.029
16000	0.013
32000	0.031
64000	0.013
128000	0.092
256000	0.011

Para el caso de la búsqueda de una palabra el texto, los resultados fueron los siguientes.

Tamaño del texto (Kb) (	Tiempo de Búsqueda (ms)
250	3,199
500	5,909
1000	12,521
2000	25,595
4000	45,360
8000	117,171
16000	215,554
32000	776,584
64000	664,720
128000	1150,971
256000	2774,850

Se gráficaron estás dos tablas, los resultados son los siguientes.

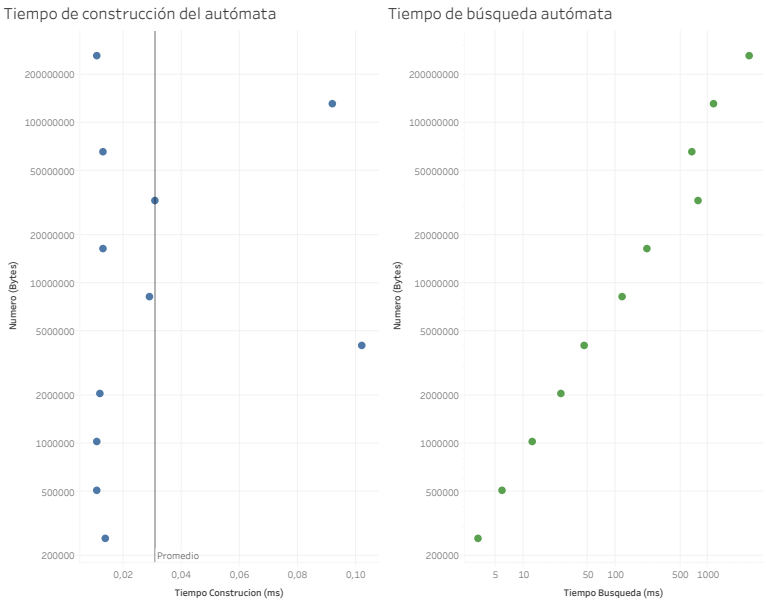


Figura 2: Gráficos del tiempo de creación y búsqueda en el Automata.

4. Análisis y Conclusiones

4.1. Construcción del Suffix Array

Se observa del gráfico obtenido para la construcció que la implementación de la construcción del Suffix Array crece linealmente con el tamaño del texto, lo cual coincide con la hipótesis presentada al inicio del informe.

4.2. Búsqueda en el Suffix Array

Se observa del gráfico de la búsqueda del Suffix Array que el tiempo crece linealmente con el tamaño del texto, lo cual no coincide con la hipótesis que se habia presentado, revisando el código se realizó la tranformación de cada sufijo a un arreglo de *char* lo que toma $O(n)$, es por esto que se obtuvieron estos resultados.

4.3. Construcción del Automata

El grafico de la construcción del Automata no puede ser analisado debido a que no fueron registrados los largos del patrón, solo puede observarse que los tiempos de ejecución son muy pequeños.

4.4. Búsqueda en el Automata

Se observa del gráfico que la búsqueda de una palabra en el texto usando el algoritmo del Automata, que el tiempo crece linealmente con el tamaño del texto, lo cual coincide con la hipótesis que se habia presentado.

4.5. Conclusiones

La construcción del Suffix Array resultó igual al orden teórico que se conocia, para el caso del la búsqueda en él, se obtuvieron malos resultados debido a una mala implementación, debido a que en un paso del algoritmo se realizó un paso innecesario de la transformación de un *String* a un arreglo de *char* el cual hace que el timepo aumnete linealmente con la entrada, se corrigió este problema y se corrieron un número pequeños de experimentos (Anexo 1), en los cuales se podía observar la mejora del tiempo de búsqueda del algoritmo, pero al ser tan pocos experimentos no es concluyente.
Para la construcción del Automata se realizaron de manera incorrecta los experimentos debido a que no se midió el largo del patron, se deberian haber repetido los experimentos, pero debido que el momento en que se detecto esta falla se estaba muy cerca de la fecha de entrega ya no quedaba tiempo para realizar un número suficiente de experimentos para obtener resultados convincentes. Aunque no se puede comparar el orden del algoritmo de construcción del Automata, si se puede observar que los tiempos de construcción en general fueron muy pequeños. La búsqueda en el Automata resultó del mismo orden del algoritmo teórico.

Estas dos implementaciones funcionan para distintos propositos, el enfoque del Suffix Array tiene un orden lineal para construirse, pero posee un tiempo menor para la busqueda en el arreglo, por lo que debería ser usado en el caso de hacer muchas consultas sobre un texto. En el caso del Automata, la construcción toma muy poco tiempo pero correrlo sobre el texto es lineal al tamaño del texto, por lo que este enfoque puede ser usado cuando se quiere hacer pocas consultas sobre el texto.

5. Anexos

5.1. Anexo 1

	Suffix Array		Automata	
Largo palabra	Tiempo Consturccion	Tiempo Busqueda	Tiempo Consturccion	Tiempo Busqueda
1	147,933033	2,047722667	0,003044333	5,557406667
2	147,933033	2,073409152	0,002631939	5,01866103
3	147,933033	2,206835654	0,003310538	5,069784423
4	147,933033	1,932976625	0,003208	4,919958688
5	147,933033	2,083488833	0,003791167	4,839957167
6	147,933033	2,295601	0,004499556	4,829205889
7	147,933033	2,141490154	0,004902385	4,841323154
8	147,933033	2,271789571	0,004376	4,766590143
9	147,933033	2,489606333	0,004869667	4,921996333
10	147,933033	2,076573444	0,005906889	4,681657667

Figura 3: Experimento con corrección de busqueda en Suffix Array texto de 1MB, con largo de palabra de 1 a 10.