Departamento de
Ingeniería Mecánica
Universidad de Concepción

Proyecto Semestral: Detección de patrones en múltiples archivos

	José Alarcón A.
INTEGRANTES	Gabriel Garrido Baeza
	Nicolás Ricciardi R.
PROFESOR	José Sebastián S.
AYUDANTES	Luciano Argomedo S.
	Alex Blanchard O.
	Rodrigo Coronado V.
	Iván Montti D.
	Pedro Muñoz S.

08 de Julio 2025

1 Introducción

Este proyecto busca que los estudiantes del curso demuestren su capacidad de aprender nuevas estructuras de datos y algoritmos, no estudiadas en profundidad en clases teóricas ni laboratorios. En este proyecto en particular tendrán una pequeña introducción a un área de investigación en Ciencias de la Computación que tiene décadas de resultados.

En la realización de este proyecto se contó con 3 objetivos:

1. Implementar 4 soluciones que tengan como objetivo el reconocimiento de patrones dentro del texto, con 3 siendo soluciones algorítmicas y 1 basada en estructuras de datos.
2. Analizar las soluciones y explicarlas de manera precisa y concisa, dando a entender el conocimiento que se tiene de estas.
3. Medir los tiempos de ejecución promedio con documentos de distintos tamaños y distintos casos considerando cuando el patrón buscado existe dentro del texto como cuando no existe, y finalmente analizar la complejidad de las soluciones.

2 Descripción de las soluciones

2.1 Soluciones algorítmicas

2.1.1 Boyer-Moore

Complejidad: $O(n)$

Primero se entrega el patrón a buscar y se preprocesa, almacenando todos los caracteres y los saltos necesarios desde el ultimo carácter del patrón para llegar a al siguiente.

Por ejemplo:

Patrón	c	a	b	c	a	a	b
Índice	1	2	3	4	5	6	7
Saltos	6	5	4	3	2	1	

Después del procesado la tabla quedaría como:

carácter	a	b	c	Otro
Saltos	1	4	3	7

Posteriormente se “alinea” al inicio del texto y se comprueba si coincide el patrón, de no ser así el patrón avanza la cantidad de saltos que se le asigno al último carácter de la parte de texto con la que se alineó el patrón para evitar búsquedas redundantes, que significa esto, significa que en caso de que una letra del patrón no coincida con el texto a mitad de la comparación, no tengamos que necesariamente empezar la comparación desde el inicio del patrón, sino desde un punto medio que pueda coincidir.

2.1.2 Knuth-Morris-Pratt

Complejidad: $O(\frac{n}{m})$

Al igual que en la solución anterior el patrón pasa por un preprocesamiento antes de ser comparado en el texto, donde se le asigna un valor a cada carácter del texto. Este preprocesamiento es para conseguir algo conocido como *Longest Preffix which is also Suffix*(LPS) lo que nos evita búsquedas redundantes en caso de que un carácter no coincida en mitad de la comparación.

El proceso consiste en empezar a comparar el patrón consigo mismo a partir del segundo carácter porque como dijimos antes el LPS nos servirá para ver si se puede

empezar a compara desde la mitad de patrón en caso de que se vea interrumpido el proceso.

Un ejemplo del proceso sería:

Posición	0	1	2	3	4	5	6
Patrón	x	o	x	o	x	x	o
Regreso	0	0	1	2	3		
					1		
					0	1	2

Luego procedemos a hacer el mismo proceso hicimos para encontrar el LPS pero desde el primer carácter del texto, y cuando llegue a 6 guardamos el índice y volvemos al siguiente inicio posible usando el vector de regreso.

2.1.3 Rabin-Karp

Complejidad: $O(n+m)$

Este algoritmo consiste en genera un hash por medio de nuestro patrón y verificar si el hash del texto coincide con el de nuestro patrón.

forma de crear el hash es:

$$hash(patrón) = (S_1 * d^{m-1} + S_2 * d^{m-2} + S_3 * d^{m-3} \dots S_m * d^0) \; mod \; q$$

m: longitud del patrón

d: tamaño del alfabeto ASCII

n: longitud del texto

q: número primo arbitrario

El d elevado se ocupa para evitar que se tenga el mismo número en caso de que un patrón tenga las misma letras pero en diferente orden.

Entonces si tenemos un q=1, un d=26, n=3 y el patrón aab el hash del patrón quedaría como:

$$hash(aab) = (1 * 26^{3-1} + 1 * 26^{3-2} + 2 * 26^{3-3}) \; mod \; 1 = 704$$

Y luego empezamos comparando del inicio del texto tomando de n caracteres a la vez para transformarlos con la función hash, avanzando un espacio a la vez y en caso de encontrar una coincidencia se guarda el índice.

2.2 Soluciones basadas en estructuras de datos

2.2.1 Suffix Array

Complejidad: $O(n\ log\ n)$

En esta solución si tenemos un texto con n caracteres, entonces, crearemos n sufijos descontando una letra del inicio del texto en cada uno de la siguiente manera y luego ordenando dichos sufijos lexicográficamente.

Índice	Texto
0	Banana
1	anana
2	nana
3	ana
4	na
5	a

→

Nuevo índice	
5	a
3	ana
1	anana
0	Banana
4	na
2	nana

Luego guardamos como vector el nuevo orden del índice

Ahora supongamos que queremos buscar "ana" en "banana", hacemos búsqueda binaria en el suffix_array. En cada paso, tomamos el sufijo que comienza en esa posición, y comparamos sus primeros caracteres con el patrón. Si hay coincidencia, revisamos si hay múltiples apariciones en posiciones vecinas.

3 Descripción de equipos

Hay que tomar en cuenta que el equipo utilizado para las pruebas experimentales fue un HP VICTUS 16, que posee las siguientes especificaciones:

Tabla 1. Especificaciones del computador

Componente	Modelo
CPU	Intel Core I5 8va gen
RAM	8GB

4 Resultados experimentales

En las tablas que se verán a continuación se presentarán los tiempos de ejecución de las distintas soluciones llevando a cabo el reconocimiento de patrones en los textos con

varios archivos a la vez además de tener casos donde el patrón se encuentra dentro del texto y otros casos donde no existe.

CANT. DE ARCHIVOS.	PATRÓN	EXISTENCIA	ALGORITMO	TIEMPO DE EJECUCIÓN(MS)	CONCURRENCIAS
10	Descarga	Existe	Rabin-Karp	28	1
10	Descarga	Existe	KMP	7	1
10	Descarga	Existe	Boyer-Moore	30	1
10	Descarga	Existe	Suffix-Array	1	1
20	Televisor	No Existe	Rabin-Karp	207170	0
20	Televisor	No Existe	KMP	28130	0
20	Televisor	No Existe	Boyer-Moore	46245	0
20	Televisor	No Existe	Suffix-Array		
30	Logica	Existe	Rabin-Karp	422480	434
30	Logica	Existe	KMP	71226	434
30	Logica	Existe	Boyer-Moore	345660	434
30	Logica	Existe	Suffix-Array		
100	Patron que no quiero que encuentre	No Existe	Rabin-Karp	2686158	0
100	(El mismo anterior)		KMP	283051	0
100	(El mismo anterior)		Boyer-Moore	511801	0
100	(El mismo anterior)		Suffix-Array		0

Tabla 2. Resultados experimentales

5 Análisis y discusión de resultados

Se evaluaron los cuatro algoritmos de búsqueda de patrones a conjuntos de archivos de distintos tamaños (10, 20, 30 y 100). Se consideran tanto casos donde el patrón está presente en los textos como donde no lo está. Los indicadores principales son el tiempo de ejecución (en milisegundos) y el número de concurrencias (coincidencias del patrón).

En el caso donde se trabajó con 10 archivos y se ocupó el patrón **Descarga:**

Rabin-Karp	KMP	Boyer-Moore	Suffix-Array
28	7	30	1

Tabla 3. Tiempos de ejecución 10 archivos

Al trabajar con 10 archivos y un patrón existente el algoritmo Suffix-Array mostró un rendimiento sobresaliente, resolviendo la tarea en tan solo 1 milisegundo, seguido por KMP con 7 ms. Rabin-Karp y Boyer-Moore fueron considerablemente más lentos, registrando 28 ms y 30 ms respectivamente.

En el caso donde se trabajó con 20 archivos y se ocupó el patrón **Televisor:**

Rabin-Karp	KMP	Boyer-Moore	Suffix-Array
207,170	28,130	46,245	(Sin dato)

Tabla 3. Tiempos de ejecución 20 archivos

En el conjunto de 20 archivos con el patrón Televisor, KMP superó a Boyer-Moore y Rabin-Karp con una ventaja significativa.

En el caso donde se trabajó con 30 archivos y se ocupó el patrón **Logica:**

Rabin-Karp	KMP	Boyer-Moore	Suffix-Array
422,480	71,226	345,660	(Sin dato)

Tabla 3. Tiempos de ejecución 30 archivos

En el escenario con 30 archivos y el patrón Logica, donde KMP nuevamente lideró en eficiencia, muy por delante de Rabin-Karp y Boyer-Moore. Lamentablemente, no se obtuvieron datos para Suffix-Array en este caso. En cuanto a los casos donde el patrón no está presente en los archivos, la diferencia entre algoritmos se vuelve aún más evidente.

En el caso donde se trabajó con 100 archivos y se ocupó el patrón **Patron largo que no existe**:

Rabin-Karp	KMP	Boyer-Moore	Suffix-Array
2,686,158	283,051	511801	(Sin dato)

Tabla 3. Tiempos de ejecución 10 archivos

Como podemos ver a lo largo de los 4 casos distintos la tendencia en el tiempo de ejecución se ha mantenido constante teniendo un podio claro con respecto a la eficiencia en el reconocimiento de patrones. Siendo en primer lugar el algoritmo de Knuth-Morris-Pratt, seguido de Boyer-Moore y en último lugar y por tanto el menos eficiente el algoritmo de Rabin-Karp.

6 Conclusión

Los resultados obtenidos a partir del análisis de los distintos algoritmos de búsqueda de patrones en texto permiten concluir que el algoritmo Knuth-Morris-Pratt es, en la práctica, el más eficiente frente a una variedad de escenarios, tanto cuando el patrón existe como cuando no, y sin importar el tamaño del conjunto de archivos. Su bajo tiempo de ejecución y buen escalamiento lo posicionan como una opción robusta para sistemas que requieren búsquedas rápidas y fiables.
Por el contrario, Rabin-Karp demostró un rendimiento muy inferior, especialmente en casos donde el patrón no existe, debido al alto costo de sus verificaciones adicionales, lo que lo vuelve una opción poco recomendable en contextos de gran volumen.
Boyer-Moore presentó un desempeño intermedio, sin destacar significativamente. Suffix-Array, si bien mostró un tiempo de ejecución extremadamente bajo en el único caso registrado, no cuenta con suficientes datos en escenarios de mayor complejidad para ser evaluado en profundidad; no obstante, su rendimiento puntual sugiere que, con una implementación adecuada, podría ser incluso superior a KMP en ciertos contextos, especialmente cuando se requieren múltiples búsquedas sobre un mismo conjunto de datos. Por lo tanto, se recomienda el uso de KMP como solución principal, dejando abierta la posibilidad de explorar Suffix-Array en aplicaciones donde su estructura compleja pueda aprovecharse al máximo.

Referencias