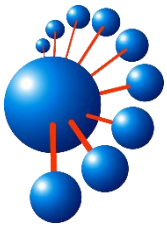

Universidad de Concepción

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 del preprocesamiento: $O(m + \sigma)$

Complejidad de búsqueda: $O(n)$

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 del preprocesamiento: $O(m)$

Complejidad de búsqueda: $O(n)$

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 comparar el patrón consigo mismo a partir del segundo carácter porque como dijimos antes el LPS nos servirá para ver si se puede volver a empezar a comparar desde una posición que no sea la inicial 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
LPS	0	0	1	2	3	1	2

Luego repetiremos el proceso que ocupamos para encontrar el LPS pero desde el primer carácter del texto que en caso de verse interrumpido, ocupara el vector LPS que guarda los índices de regreso para volver a empezar. En caso de que la búsqueda no se vea interrumpida y el índice llegue a 6 guardamos a posición y volvemos al siguiente inicio posible usando el vector de regreso.

2.1.3 Rabin-Karp

Complejidad: $O(m)$

Complejidad de búsqueda: $O(n)$

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 11va 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.

Algoritmo	*100 patrones (ms)*	*200 patrones (ms)*	*300 patrones (ms)*	*1000 patrones (ms)*	*2000 patrones (ms)*	*Tiempo promedio general (ms)*
Rabin-Karp	1.0	13.5	16.0	25.4	23.5	15.88
KMP	0.0	4.0	2.0	4.0	6.0	3.2
Boyer-Moore	1.0	8.5	6.5	55.5	38.0	21.9
Suffix Array	0.0	0.0	0.0	0.0	0.0	0.0

Tabla 2. Tiempos de ejecución

Ahora en la tabla 3 se presenta el uso de memoria de cada algoritmo, con patrones que en caso de no existir dentro del texto presentan 0 bytes de memoria usados.

Patrón	*Rabin-Karp (bytes)*	*KMP (bytes)*	*Boyer-Moore (bytes)*	*Suffix Array (bytes)*
but	24	-	-	-
GAT	92	92	92	26432
PEOPLE	0	-	-	-
Rusia	0	-	-	-
two	8	8	8	26432
young	4	4	4	18360
elrubius	0	0	0	18360
Manoel	36	36	36	13664
En español?	0	0	0	13664
GTGAGAAAATTCTTTCCCTCACTTTCTTCTAA	4	4	4	1292
colores	0	0	0	1292
No	-	-	-	13664
conejo	-	-	-	13664
aula	0	-	-	-
a	-	-	-	13664

Tabla 3. Memoria usada

5 Análisis y discusión de resultados

Los resultados experimentales permiten observar claramente cómo varía el comportamiento de los algoritmos frente al aumento en la cantidad de patrones y al tipo de datos procesados.

Primero se mostrará el análisis temporal de los algoritmos, comparando los rendimientos experimentales con sus rendimientos teóricos:

Rabin-Karp: Demuestra un comportamiento estable y predecible consistente con su complejidad teórica de $O(n+m)$, con un crecimiento moderado. Su consistencia lo convierte en una opción equilibrada para aplicaciones que requieren un balance entre rendimiento y simplicidad de implementación.

Knuth-Morris-Pratt: Presenta la mejor escalabilidad entre los algoritmos de búsqueda directa, manteniendo tiempos bajos y crecimiento lineal. Este comportamiento confirma su eficiencia teórica $O(n/m)$ y lo posiciona como la opción más confiable para búsquedas frecuentes debido a sus tiempos de respuesta consistente**.**

Boyer-Moore: A pesar de su complejidad teórica $O(n)$ presenta el comportamiento más irregular, con picos significativos en 1000 patrones y 2000 patrones. Esta variabilidad sugiere una fuerte dependencia del tipo de patrón y texto, característica típica de este algoritmo que puede ser muy eficiente o problemático según el caso específico.

Suffix Array: Demuestra un rendimiento excepcional que concuerda con su complejidad $O(n \log n)$ con tiempos de ejecución constantes de 0 ms incluso en el caso más extremo, lo cual se explica por su naturaleza de estructura de datos preconstruida que permite búsquedas en tiempo constante una vez indexado el texto.

A continuación se presenta el análisis espacial de cada uno de los algoritmos el cual presenta grandes diferencias dependiendo del tipo de solución:

En lo que respecta a las soluciones algorítmicas implementadas Rabin-Karp, Knuth-Morris-Pratt y podemos ver que las tres presentan un consumo moderado y similar entre sí. Esto se debe a la naturaleza de estas soluciones, que esta ligada directamente al tamaño del patrón y el número de coincidencias encontradas, lo que los hace eficientes en eficientes para búsquedas rápidas y sencillas, ya que no tienen la necesidad de estructuras de datos complejas.

El Suffix Array es un caso particular, ya que al ser una solución en base a estructuras de datos presenta un consumo mucho más elevado, que en el caso de nuestras pruebas podemos ver que pudo llegar a ser más de mil veces mayor que en los otros algoritmos. Esto se debe a la necesidad de almacenar todos los sufijos posibles en el texto y ordenarlos, lo que aunque costoso permite una búsqueda binaria extremadamente rápida y eficiente. Además que el consumo no depende del patrón, sino del tamaño del texto.

6 Conclusión

En base a los resultados experimentales podemos deducir que para aplicaciones con búsquedas ocasionales y recursos limitados o equipos de no tan alta potencia, KMP representa la mejor opción por su eficiencia y bajo consumo de memoria. En sistemas que realizan búsquedas intensivas sobre los mismos textos.
Suffix Array justifica su alto consumo de memoria con una velocidad de búsqueda excepcional, debido a esto es por lo que generalmente es usado en bases de datos de gran volumen que poseen grandes capacidades de procesamiento.
Rabin-Karp es recomendable para aplicaciones que requieren simplicidad de implementación con rendimiento aceptable por lo que se puede considerar una opción intermedia en comparación a las otras dos soluciones algorítmicas implementadas.
Finalmente tenemos el algoritmo Boyer-Moore el cual debe usarse con cautela, evaluando previamente su comportamiento con los patrones específicos de la aplicación ya que puede presentar grandes irregularidades a la hora de ponerlo en práctica.

Podemos concluir que las soluciones algorítmicas tienen ventajas cuando se trata de facilidad de implementación y bajos costos en el procesamiento. Por otro lado las soluciones en base a estructuras de datos tienen un alto consumo espacial lo que podría limitar su uso en ciertos dispositivos, pero a cambio entregan una increíble velocidad de búsqueda superando con creces a la de las soluciones algorítmicas.

Referencias