Proyecto 2: Conteo de patrones

DAZHI FENG, 2019435710, Universidad de Concepción, Chile

Este segundo proyecto del curso de Estructura de Datos y Algoritmos Avanzados se centra en el concepto de búsqueda de patrones en una secuencia de texto. En este proyecto, se han implementado tres algoritmos distintos: el algoritmo de Knuth-Morris-Pratt, Suffix Array y FM-Index. Este informe proporciona una descripción detallada de los tres algoritmos, incluyendo los aspectos de su implementación. Además, se presenta un análisis teórico de su rendimiento, seguido de un análisis experimental diseñado para validar las conclusiones teóricas. Por último, se incluye una discusión y conclusiones basadas en los resultados obtenidos en el proyecto.

ACM Reference Format:
Dazhi Feng. 2023. Proyecto 2: Conteo de patrones. 1, 1 (October 2023), 14 pages.

Author's address: Dazhi Feng2019435710,, Universidad de Concepción, Concepción, Chile.

XXXX-XXXX/2023/10-ART $15.00
https://doi.org/

1. INTRODUCCIÓN

La coincidencia de cadenas, comúnmente conocida como "string matching", es un problema fundamental en la informática y la ciencia de la computación. Consiste en encontrar todas las apariciones de un patrón específico en una secuencia de texto, lo que tiene aplicaciones en campos tan diversos como la búsqueda de texto en motores de búsqueda, la bioinformática y el análisis de datos.

En este contexto, este proyecto se enfoca en la implementación y análisis de tres algoritmos para abordar el problema de coincidencia de cadenas: el algoritmo de Knuth-Morris-Pratt (KMP), Suffix Array y FM-Index. Cada uno de estos algoritmos se ha diseñado para determinar la cantidad de veces que un patrón dado, denotado como p, aparece en un texto, representado como T.

El algoritmo KMP se basa en el preprocesamiento del patrón para identificar los saltos que se deben realizar cuando se encuentra una falta de coincidencia. Esto permite minimizar el número de comparaciones, mejorando así el rendimiento en comparación con enfoques naive.

Suffix Array es una estructura de datos que almacena todos los sufijos de un texto dado en un orden lexicográfico. Este enfoque proporciona una forma eficiente de buscar patrones en el texto, ya que se pueden utilizar búsquedas binarias en el Suffix Array para identificar las coincidencias de manera rápida.

FM-Index es una estructura de datos compacta que se deriva a partir de la transformación de Burrows-Wheeler (BWT) y se utiliza para la búsqueda de patrones. FM-Index se caracteriza por su eficiencia en la compresión de datos y su capacidad para realizar búsquedas rápidas en el texto original.

En el transcurso de este informe, se detallará la implementación de estos tres algoritmos, se presentará un análisis de su funcionamiento y rendimiento, y se expondrán los resultados obtenidos a través de experimentos diseñados para evaluar su eficacia en la resolución del problema de coincidencia de cadenas.

2. SOLUCIONES PROPUESTAS

Knuth–Morris–Pratt (KMP)

El algoritmo KMP se basó en el pseudocódigo y la información proporcionada en la referencia [4]. A diferencia de los métodos de fuerza bruta, que comparan el patrón con cada subcadena de la cadena de texto, KMP realiza una búsqueda más inteligente y evita comparaciones innecesarias. Este algoritmo consta de dos partes:

1. **Preprocesamiento:** Se crea una estructura llamada tabla LPS (Longest Prefix Suffix) utilizando el patrón. La tabla LPS se utiliza para determinar cuántos caracteres se pueden retroceder en la cadena de búsqueda cuando se encuentra un mismatch entre el patrón y la cadena.
   La idea principal en la construcción de la tabla LPS es identificar el prefijo más largo del patrón que también es un sufijo válido. Por ejemplo, para el patrón `"ABAB"`, el prefijo más largo que es también un sufijo válido es `"AB"`. La tabla LPS se crea de manera iterativa y almacena esta información para cada posición del patrón. Esto permite al algoritmo saltar en caso de una no coincidencia parcial entre el patrón y la cadena.
2. **Algoritmo de búsqueda:** El proceso de búsqueda comienza comparando el primer carácter del patrón con el primer carácter del string. Si hay coincidencia, la búsqueda continúa con el siguiente carácter en ambos. Si se produce una discrepancia, en lugar de reiniciar la búsqueda desde el principio, el algoritmo utiliza la información de la tabla LPS para determinar cuántos caracteres retroceder en la cadena de búsqueda.

Suffix Array

Inicialmente, se implementó una versión simple del Suffix Array que tenía una complejidad de $O(n^2 \log n)$. Sin embargo, dado que el enfoque principal del proyecto era la implementación de la búsqueda en lugar de la estructura de datos en sí, se optó por obtener y utilizar una implementación más eficiente obtenida de GeekforGeeks[1]. Esta implementación es un poco más compleja, pero tiene una menor complejidad temporal y permitió llevar a cabo experimentos con conjuntos de datos más grandes.

La idea detrás de esta implementación se basa en que los strings a ordenar son sufijos de una única cadena. Inicialmente, se ordenan los sufijos según el primer carácter, luego según los primeros dos caracteres, seguido por los primeros cuatro caracteres, y así sucesivamente. Si ya se ordenó los sufijos según los primeros 2^i caracteres, se puede ordenar los sufijos según los primeros 2^{i+1} con complejidad $O(n \log n)$, ya que solo es necesario comparar dos valores.

Una vez generado el Suffix Array, se implementó el algoritmo de búsqueda basado en la información proporcionada en la referencia [5]. Este algoritmo utiliza una doble búsqueda binaria para encontrar el intervalo en el Suffix Array donde se encuentran las ocurrencias del patrón.

FM-Index

En la construcción del FM-Index, el proceso comienza con la obtención de la Transformada Burrows-Wheeler (BWT) del texto de entrada[2]. La BWT se deriva a partir de todas las rotaciones posibles del texto, y se selecciona el último carácter de cada rotación.

A partir de la BWT, se crean dos tablas: la tabla C (Tabla de conteo de caracteres) y la tabla occ (Tabla de ocurrencias). La tabla C almacena información sobre el alfabeto y cuántas veces caracteres más pequeños que c aparecen en el texto. Por otro lado, la tabla occ registra cuántas veces aparece cada carácter en diferentes posiciones del texto BWT.

Finalmente, con estas dos tablas, se puede realizar la operación $count()$, que permite contar las ocurrencias del patrón en el texto. La operación $count()$ avanza desde el final hasta el inicio del patrón, realizando en cada iteración una suma (similar a la operación LF). Esto implica la adición de valores de la tabla C y la tabla occ. En cada iteración, se determina el rango del intervalo que contiene las posibles ubicaciones del patrón en el texto. Al terminar, se obtiene el rango en donde esta el patrón[3].

3. DETALLES DE IMPLEMENTACIÓN

Cada algoritmo se implementó mediante una clase que almacena sus respectivas estructuras de datos. La clase para el algoritmo KMP guarda el arreglo LPS, la clase para Suffix Array guarda el arreglo de sufijos y la clase para el FM-Index almacena las tablas C y occ. Cada una de estas clases incluye una función que devuelve el espacio de memoria utilizado, space() y la función count(), que cuenta los patrones encontrados.

Knuth–Morris–Pratt

Para realizar el preprocesamiento, se crea la tabla KMP (arreglo LPS), como un vector de enteros. Luego, se recorre el patrón carácter por carácter. Si se encuentra un sufijo que coincide con el prefijo en la posición actual, se registra su longitud en la tabla. Si no hay coincidencia, se guarda la longitud actual del sufijo en la tabla y se retrocede a través de la tabla hasta encontrar un sufijo que coincide. Esto permite determinar cuántos caracteres se puede saltar si no hay una coincidencia completa.

La búsqueda es similar a la búsqueda por fuerza bruta, donde se compara carácter por carácter. La diferencia es que, en caso de coincidencia con el patrón o no coincidencia al comparar los caracteres, se consulta la tabla para evitar comparaciones innecesarias de algunos caracteres.

KMP depende de la tabla LPS, lo que permite realizar operaciones de count() con diferentes strings de entrada sin necesidad de recrear la tabla LPS. Sin embargo, no es posible realizar la operación count() con diferentes patrones, ya que esta tabla está vinculada al patrón específico.

Suffix Array

La creación del Suffix Array comienza con una estructura para sufijos. Cada sufijo se etiqueta con un índice y se le asigna un rank. Inicialmente, el rank se obtiene restando 'a' al primer carácter del sufijo. Cada sufijo tiene dos ranks: uno con el carácter actual (primer caracter del sufijo) y otro con el siguiente. Se recorre el string y para cada sufijo se va asignando el índice y los dos ranks.

Luego, se ordenan los sufijos, comparando los ranks (si hay empate, desempatar por el segundo rank). Terminando eso, los sufijos están ordenados por los primeros dos caracteres. Ahora, hay que seguir con los primeros 4, 8, 16 caracteres, y así sucesivamente, hasta 2 veces la longitud original n.

En cada iteración, se asigna nuevos ranks basados en los sufijos ordenados previamente. Esto se hace comparando los ranks de los sufijos con los sufijos adyacentes en el arreglo. Los sufijos con ranks iguales y caracteres siguientes idénticos reciben el mismo rank. Si no coinciden, se asigna un nuevo rank, aumentando el contador de ranks. Esto asegura que los sufijos con caracteres iniciales diferentes reciban diferentes ranks. También, se actualiza un arreglo ind que almacena las posiciones correctas de los sufijos. Luego se calcula el segundo rank (rank[1]) para cada sufijo, comparándolo con el sufijo k/2 posiciones adelante.

Finalmente, nuevamente se reordenan los sufijos según los primeros k caracteres, y este proceso se repite hasta que k alcance 2n.

Para la operación count(p), se realiza una doble búsqueda binaria. En la primera búsqueda binaria, se determina la posición inicial en la que comienzan las ocurrencias del patrón en el Suffix Array. La segunda búsqueda binaria se utiliza para identificar la posición donde finalizan estas ocurrencias. Luego, al restar la posición final de la posición inicial, se obtiene el número de ocurrencias del patrón en el string.

FM-index

Para implementar el autoíndice, se siguieron los siguientes pasos:

Ok, pero ojo

1. En primer lugar, se obtuvo la Transformación de Burrows-Wheeler (BWT) del string original. Esto se logró creando un vector de strings que almacenaba todas las rotaciones del string original, generadas mediante la función `substr()`. Luego, se ordenaron estas rotaciones y se construyó la BWT tomando el último carácter de cada rotación.
2. Luego, se implementó la tabla C como un mapa. Se ordenó la BWT y se recorrió caracter por caracter. En cada iteración, se verificaba si el carácter era distinto al anterior. En caso afirmativo, se actualizaba la tabla C y se incrementaba el contador.
3. Con la BWT y el alfabeto del string (obtenido a partir de la tabla C), se creó un mapa que relaciona cada carácter con un vector de enteros para almacenar las ocurrencias. Para cada carácter, se recorrió todo el string original y se registraron las ocurrencias.

Una vez que se tenían estas dos tablas (o mapas), se pudieron implementar las funciones `LF(i)` y `count(p)`:

1. La función LF(i) devuelve el mapeo del carácter i de la BWT con el carácter correspondiente en el string original. Esto se logra simplemente sumando el valor en la tabla C con el valor en la tabla de ocurrencias.
2. Para buscar patrones, se utiliza un bucle que itera a lo largo de todo el patrón. En cada iteración, se realiza dos veces una operación similar a LF, que suma un valor en la tabla C con otro valor en la tabla de ocurrencias. Al finalizar este proceso, se obtiene el rango en el que se encuentra el patrón. Basta con restar y luego sumar 1 para obtener el número de coincidencias.

En general bien, pero una recomendación: Usa unidades para indicar la complejidad es

4. ANÁLISIS TEÓRICO

Knuth–Morris–Pratt

El algoritmo KMP tiene una complejidad de $O(n + m)$, debido al preprocesado que se tiene que hacer $O(m)$ y luego el matching $O(n)$.

- **Construcción:** La construcción del algoritmo KMP toma $O(m)$, donde m es el tamaño del patrón, ya que implica la construcción de una tabla LPS auxiliar basada en el patrón.
- **Búsqueda:** $O(n)$, donde n es el tamaño del string, debido a que se desplaza el patrón a lo largo del string para encontrar ocurrencias.
- **Espacio:** El espacio que ocupa el algoritmo KMP para la tabla auxiliar es $O(m)$.

Suffix Array

La complejidad del análisis del suffix array depende de la implementación utilizada, y existen varias implementaciones con diferentes niveles de eficiencia. En este caso, se utilizó implementación en la que la construcción del suffix array tiene un tiempo de $O(n \log n \log n)$, mientras que el matching tiene una complejidad de $O(m \log n)$ debido a la búsqueda binaria.

- **Construcción:** $O(n \log n \log n)$ en esta implementación, debido a que usa ranks y va comparando los primeros 2 caracteres, luego 4, 8, y así sucesivamente. Esta comparación inicial toma $O(n \log n)$ debido a que, una vez que se han comparado los caracteres anteriores, la comparación de 2 sufijos se reduce a una operación de tiempo constante (comparación de solo 2 valores). Este proceso se repite $O(\log n)$ veces, lo que resulta en una complejidad total de $O(n \log n \log n)$.
- **Búsqueda:** Dado que se utiliza búsqueda binaria, la complejidad es $O(m \log n)$, donde n es el tamaño del string y m es el tamaño del patrón. Esto se debe a que se realiza una doble búsqueda binaria en el suffix array para encontrar ocurrencias del patrón y un sufijo debe compararse con m caracteres.
- **Espacio:** $O(n)$ para almacenar los índices de los sufijos del string original.

FM-index

La complejidad del auto-índice depende del tiempo de construcción de las tablas auxiliares y la creación de la BWT. La implementación utilizada para la BWT es bastante sencilla: genera todas las rotaciones del texto y luego las ordena. Esto resulta en una complejidad de $O(n^2 + n \log n)$.

La creación de la tabla C implica ordenar el texto, lo que tiene una complejidad de $O(n \log n)$. Para la tabla *occ*, se recorre el alfabeto del texto, y en cada iteración, se trabaja con todos los caracteres, lo que resulta en una complejidad de $O(\sigma \cdot n)$, donde σ representa el tamaño del alfabeto.

Finalmente, en la fase de búsqueda, se itera sobre el patrón completo, lo que implica una complejidad de $O(m)$, donde m es la longitud del patrón.

- **Construcción:** La fase de construcción de FM-Index implica la generación del BWT y las tablas auxiliares, donde el BWT tiene una complejidad de $O(n^2)$, la tabla C se crea mediante una ordenación, lo que da $O(n \log n)$, y la tabla *occ* se genera recorriendo el alfabeto y el texto, lo que resulta en $O(\sigma \cdot n)$, donde σ representa el tamaño del alfabeto. Entonces el de mayor complejidad es generar el BWT, $O(n^2)$.
- **Búsqueda:** Se itera sobre el patrón completo, lo que implica una complejidad de $O(m)$, donde m es la longitud del patrón.
- **Espacio:** El espacio utilizado por el FM-Index consiste en dos tablas auxiliares. La tabla C tiene un tamaño igual al tamaño del alfabeto, $O(\sigma)$, y la tabla *occ* tiene un tamaño de $O(\sigma \cdot n)$, donde σ es el tamaño del alfabeto y n es la longitud del texto.

5. ANÁLISIS EXPERIMENTAL

Ok, me parece c

Se llevaron a cabo dos experimentos diferentes para analizar el rendimiento de la búsqueda de elementos en una secuencia. En el primer experimento, se investigó el efecto del tamaño del string en los tiempos de ejecución, manteniendo constante el patrón. El tamaño del string varió, comenzando con 10000 caracteres y sumando 10000 en cada iteración hasta llegar a 100000. El patrón a buscar son los primeros 1000 caracteres del dataset.

En el segundo experimento, se mantuvo el tamaño del string constante con los primeros 100000 caracteres del dataset, pero se varió el patrón a buscar. El patrón a buscar es los primeros 10000 caracteres del dataset y sumando 10000 caracteres en cada iteración hasta llegar a 100000.

En ambos experimentos, se hizo lo siguiente: primero, se tomó el tiempo exclusivamente a la búsqueda del algoritmo, seguido por el tiempo requerido para la construcción y la búsqueda del algoritmo. Finalmente, se utilizó una función correspondiente a cada algoritmo, la cual devuelve el espacio utilizado para cada tamaño de entrada específico.

Cada experimento se repitió 40 veces. Las primeras 10 repeticiones se consideraron “cold runs” y no se incluyeron en el análisis final. Las 30 repeticiones restantes fueron consideradas en el análisis.

Las pruebas se llevaron a cabo utilizando el servidor *chome* de la Universidad de Concepción y se utilizó el dataset "dna"de Pizza&Chili.

Hipótesis

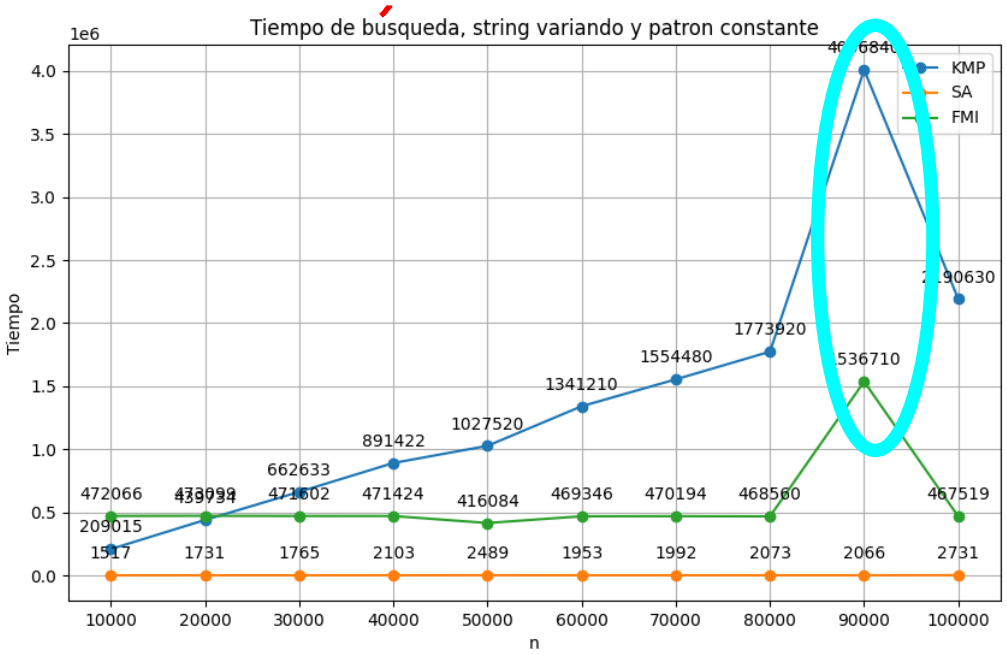
1. Considerando únicamente la búsqueda (sin construcción) con un patrón pequeño, se espera que el algoritmo KMP sea el más lento debido a su complejidad $O(n)$. Le seguiría la búsqueda en el Suffix Array con una complejidad de $O(m \log n)$ y, finalmente, el autoíndice FM-Index debería ser el más rápido y constante, ya que su rendimiento depende solo del tamaño del patrón, es decir, $O(m)$.

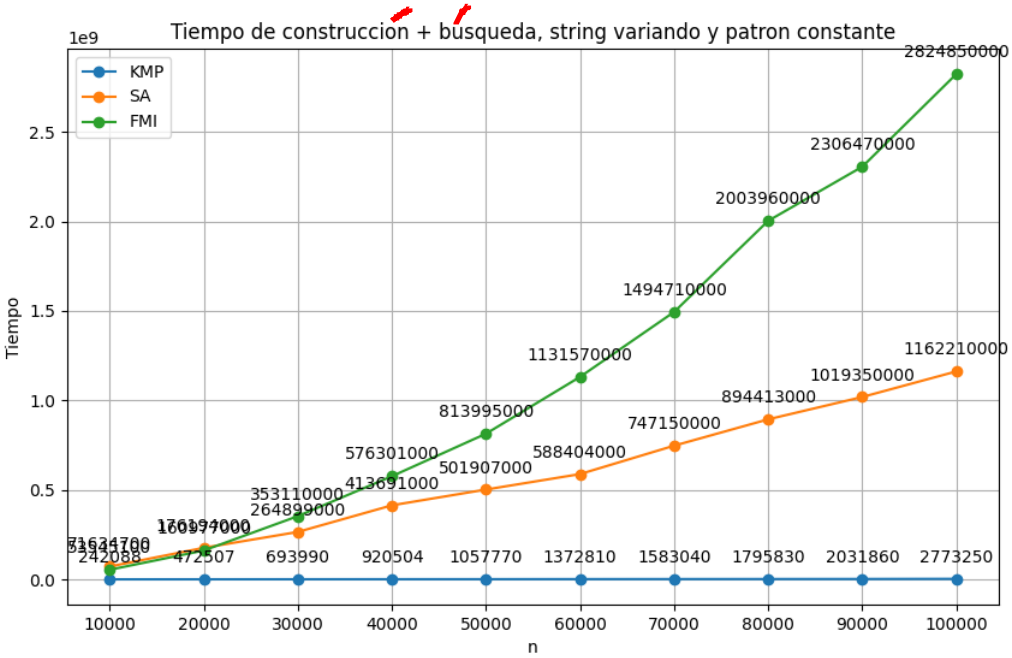
2. En el escenario de búsqueda (sin construcción) con un patrón creciente en tamaño, se espera que el tiempo de ejecución del algoritmo KMP no varíe, mientras que los algoritmos basados en Suffix Array y FM-Index incrementen su tiempo de búsqueda. Cuando el patrón se acerca al tamaño del string original, se espera que los tiempos de búsqueda de Suffix Array y FM-Index sean iguales o mayores a KMP.

3. El tiempo del algoritmo completo (construcción de la estructura de datos + búsqueda), independientemente del tamaño del patrón, el algoritmo KMP debería ser el más rápido, seguido por Suffix Array y, finalmente, FM-Index, que se espera que sea el más lento y costoso.

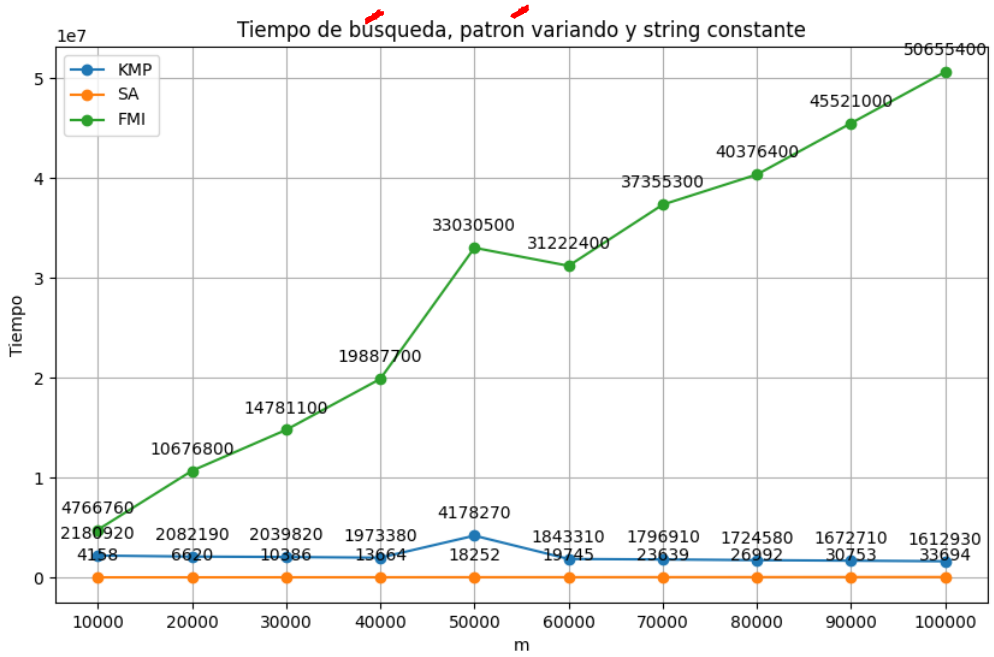
4. En cuanto al uso de espacio, se prevé que KMP sea el que requiera menos memoria, seguido por Suffix Array, y en última instancia FM-Index, sin importar el tamaño del patrón.

Gráficos de tiempo para string aumentando en cada iteración y patrón constante (m=1000)

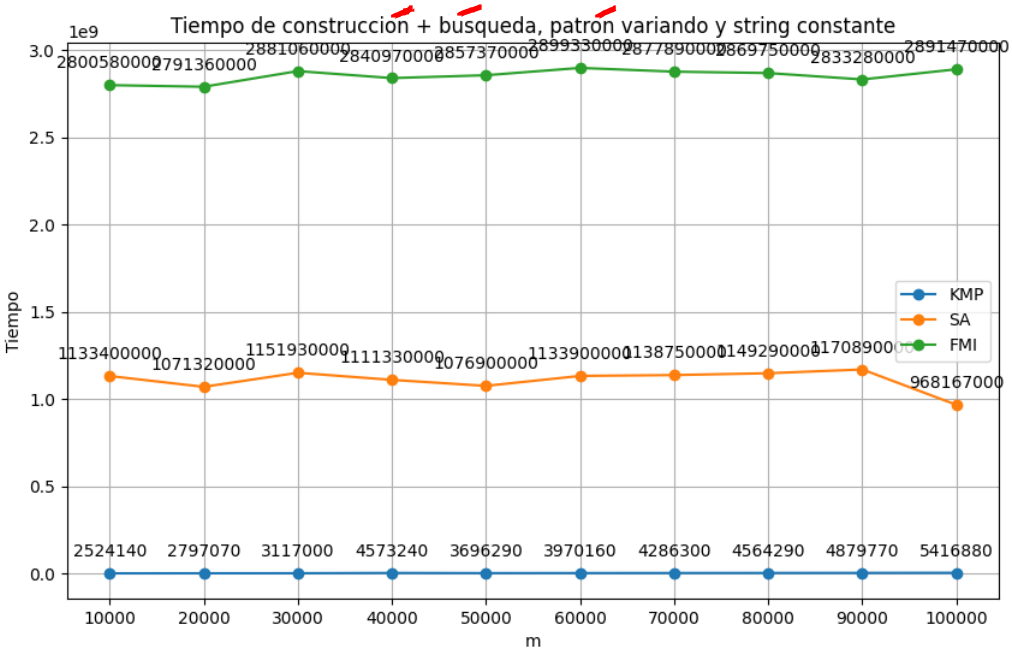




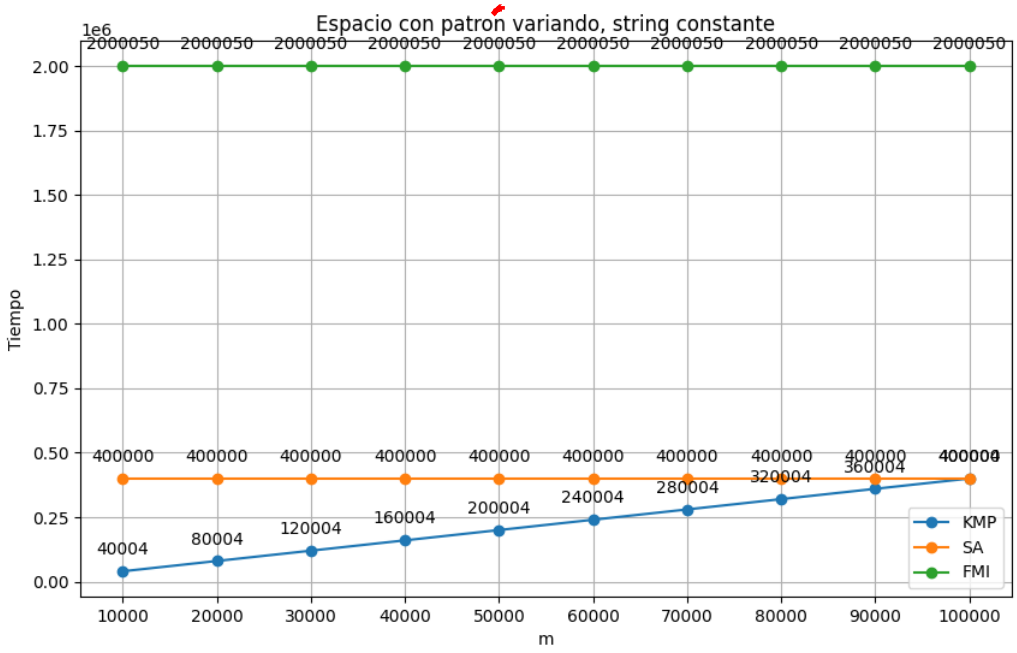
Gráficos de tiempo para patrón aumentando en cada iteración y string constante (n=100000)



Si veo bien, KMP dismi



Gráficos de espacio



Tablas string incrementándose, patrón constante

n representa el número de elementos, t representa el tiempo, y v representa la varianza. $t1$ es el tiempo de búsqueda solo, mientras que $t2$ es el tiempo de búsqueda y construcción.

Cuadro 1. Tiempo solo de búsqueda

n	t1kmp	v1kmp	t1sa	v1sa	t1fmi	v1fmi
10000	2.09015e+05	1.47782e+07	1.51747e+03	6.11632e+04	4.72066e+05	7.47799e+07
20000	4.39734e+05	4.55610e+07	1.73057e+03	9.76441e+04	4.73099e+05	7.58479e+07
30000	6.62633e+05	1.27976e+08	1.76477e+03	6.04686e+04	4.71602e+05	8.68062e+07
40000	8.91422e+05	2.14465e+08	2.10300e+03	3.68580e+03	4.71424e+05	6.76720e+07
50000	1.02752e+06	3.23144e+08	2.48910e+03	5.35448e+04	4.16084e+05	6.16153e+07
60000	1.34121e+06	3.81311e+08	1.95267e+03	3.72442e+03	4.69346e+05	7.84661e+07
70000	1.55448e+06	5.55600e+08	1.99190e+03	4.30316e+03	4.70194e+05	9.01483e+07
80000	1.77392e+06	6.90399e+08	2.07307e+03	5.60266e+03	4.68560e+05	8.43302e+07
90000	4.00684e+06	3.91736e+13	2.06560e+03	4.57611e+03	1.53671e+06	1.59302e+13
100000	2.19063e+06	1.35530e+09	2.73087e+03	6.24730e+05	4.67519e+05	1.83426e+08

Cuadro 2. Tiempo construcción + búsqueda

n	t2kmp	v2kmp	t2sa	v2sa	t2fmi	v2fmi
10000	2.42088e+05	2.47636e+07	7.16347e+07	2.81775e+11	5.35451e+07	8.08908e+13
20000	4.72507e+05	7.05384e+07	1.76194e+08	3.79283e+14	1.60977e+08	2.06163e+14
30000	6.93990e+05	1.48179e+08	2.64899e+08	1.41965e+15	3.53110e+08	2.88149e+15
40000	9.20504e+05	2.91537e+08	4.13691e+08	2.30408e+15	5.76301e+08	8.08706e+15
50000	1.05777e+06	3.51816e+08	5.01907e+08	4.12131e+15	8.13995e+08	8.07977e+15
60000	1.37281e+06	4.84982e+08	5.88404e+08	4.68860e+15	1.13157e+09	2.50189e+16
70000	1.58304e+06	5.74271e+08	7.47150e+08	7.18366e+15	1.49471e+09	2.88645e+16
80000	1.79583e+06	7.87699e+08	8.94413e+08	7.29315e+15	2.00396e+09	5.96211e+16
90000	2.03186e+06	8.79369e+08	1.01935e+09	9.17064e+15	2.30647e+09	3.46885e+16
100000	2.77325e+06	8.21945e+12	1.16221e+09	1.79362e+16	2.82485e+09	3.62740e+16

Tablas patrón incrementándose, string constante

Cuadro 3. Tiempo solo de búsqueda

n	t1kmp	v1kmp	t1sa	v1sa	t1fmi	v1fmi
10000	2.18092e+06	8.03062e+08	4.15750e+03	3.41187e+04	4.76676e+06	1.01827e+09
20000	2.08219e+06	9.77232e+08	6.61973e+03	1.29266e+06	1.06768e+07	3.16496e+13
30000	2.03982e+06	8.22823e+08	1.03865e+04	3.02069e+05	1.47811e+07	3.95208e+09
40000	1.97338e+06	8.59936e+08	1.36637e+04	1.04674e+06	1.98877e+07	1.56373e+10
50000	4.17827e+06	3.38226e+13	1.82521e+04	2.56104e+06	3.30305e+07	9.54416e+13
60000	1.84331e+06	7.83919e+08	1.97454e+04	4.32013e+06	3.12224e+07	2.76569e+13
70000	1.79691e+06	5.08118e+08	2.36390e+04	2.94812e+06	3.73553e+07	6.95374e+13
80000	1.72458e+06	7.39637e+08	2.69924e+04	4.44725e+06	4.03764e+07	4.44344e+10
90000	1.67271e+06	5.70006e+08	3.07527e+04	7.94845e+06	4.55210e+07	4.20953e+10
100000	1.61293e+06	6.73315e+08	3.36938e+04	6.12846e+06	5.06554e+07	5.14441e+10

Cuadro 4. Tiempo construcción + búsqueda

n	t2kmp	v2kmp	t2sa	v2sa	t2fmi	v2fmi
10000	2.52414e+06	1.01343e+09	1.13340e+09	1.18482e+16	2.80058e+09	1.92538e+16
20000	2.79707e+06	1.19303e+09	1.07132e+09	2.55035e+16	2.79136e+09	2.09255e+16
30000	3.11700e+06	1.34987e+09	1.15193e+09	1.53223e+16	2.88106e+09	5.31974e+16
40000	4.57324e+06	3.91970e+13	1.11133e+09	1.12602e+16	2.84097e+09	5.45938e+16
50000	3.69629e+06	1.44196e+09	1.07690e+09	1.59908e+16	2.85737e+09	3.30804e+16
60000	3.97016e+06	1.90213e+09	1.13390e+09	8.46938e+15	2.89933e+09	4.17692e+16
70000	4.28630e+06	1.94477e+09	1.13875e+09	1.08582e+16	2.87789e+09	3.17991e+16
80000	4.56429e+06	1.65778e+09	1.14929e+09	9.63168e+15	2.86975e+09	3.16862e+16
90000	4.87977e+06	1.01615e+09	1.17089e+09	1.54450e+16	2.83328e+09	2.01885e+16
100000	5.41688e+06	1.86222e+12	9.68167e+08	2.51033e+16	2.89147e+09	3.20372e+16

Tablas de espacios

Fig. 1. Espacio string incrementándose, patrón constante

n	skmp	ssa	sfmi
10000	4004	40000	200050
20000	4004	80000	400050
30000	4004	120000	600050
40000	4004	160000	800050
50000	4004	200000	1000050
60000	4004	240000	1200050
70000	4004	280000	1400050
80000	4004	320000	1600050
90000	4004	360000	1800050
100000	4004	400000	2000050

Fig. 2. Espacio patrón incrementándose, string constante

n	skmp	ssa	sfmi
10000	40004	400000	2000050
20000	80004	400000	2000050
30000	120004	400000	2000050
40000	160004	400000	2000050
50000	200004	400000	2000050
60000	240004	400000	2000050
70000	280004	400000	2000050
80000	320004	400000	2000050
90000	360004	400000	2000050
100000	400004	400000	2000050

6. DISCUSIÓN Y CONCLUSIÓN

De los resultados obtenidos en el análisis experimental, se puede validar parcialmente la primera hipótesis. Se observa que el algoritmo KMP es, efectivamente, el más lento, con una complejidad temporal lineal. En el caso del FM-index, se evidencia que el tiempo se mantiene relativamente constante, dado que el tamaño del patrón permanece invariable. Sin embargo, posiblemente debido a las constantes ocultas en este algoritmo, requirió más tiempo que el Suffix Array, a pesar de tener una complejidad menor. Finalmente, en el caso del Suffix Array, se aprecia que su tiempo de ejecución aumenta gradualmente, aunque no se observan diferencias muy significativas.

En el caso de la segunda hipótesis, con string constante y tamaño del patrón aumentando, se validó el análisis teórico, el algoritmo KMP mantiene un tiempo prácticamente constante. Por otro lado, se puede notar que el tiempo de búsqueda del FM-index se vuelve lineal a medida que el patrón se incrementa. En el caso del Suffix Array, también se experimenta un aumento lineal, aunque no tan pronunciado como en el caso del autoíndice.

Al evaluar el rendimiento del algoritmo completo (construcción y búsqueda), se confirmó que KMP es el más eficiente de todos, mostrando un aumento de tiempo lineal. También se observó que Suffix Array requiere menos tiempo que FM-index, con una complejidad similar al tiempo lineal en el caso de un string con tamaño en aumento y constante en el caso de un patrón con tamaño en aumento. En el FM-index, se pudo notar una complejidad cuadrática en el proceso de construcción cuando el tamaño del string aumentaba y en caso de patrón en aumento, dado que el tamaño del string se mantiene constante, no se apreció un cambio significativo en el tiempo.

En términos de uso de espacio, se confirmó que KMP es más eficiente, ya que ocupa solo el tamaño del patrón. Le sigue el Suffix Array, que utiliza un espacio igual al del string, y el FM-index, que utiliza un espacio del tamaño del string multiplicado por el alfabeto. Se observó una curva lineal en el caso del string en aumento y una curva constante cuando no se modificaba el tamaño del string para estos dos últimos algoritmos.

Con estos resultados, podría parecer que la implementación más efectiva es KMP, mientras que FM-index se muestra como la menos eficiente. Sin embargo, esta conclusión no es del todo precisa. Si bien es cierto que KMP tiene la ventaja de utilizar el menor espacio y de construir su estructura en el menor tiempo posible, esta eficiencia se traduce en un costo en términos de búsqueda, que depende del tamaño del string original. Además, una vez que se construye la estructura KMP, solo admite un solo patrón (aunque puede utilizarse con varios strings). Esto limita su utilidad cuando se busca una variedad de patrones en un solo string, ya que requeriría la construcción de una nueva estructura KMP para cada patrón.

Por otro lado, Suffix Array ofrece un buen tiempo de búsqueda, lo que lo convierte en una excelente opción para patrones pequeños. También es eficiente en el uso de espacio, que depende solo del tamaño del string, y el tiempo de construcción se puede mejorar mediante otros métodos y optimizaciones.

Finalmente, la implementación que mostró los resultados menos favorables, FM-index, es la más costosa en términos de espacio y construcción. No obstante, tiene potencial para ofrecer tiempos de búsqueda más rápidos, ya que depende únicamente del tamaño del patrón. El costo de construcción también es mejorable, ya que existen métodos más rápidos y eficientes para obtener la BWT y las tablas.

En resumen, no existe un método claramente superior a los demás, y la elección del algoritmo depende en gran medida de los recursos disponibles y de los objetivos específicos. Cada algoritmo presenta un equilibrio particular entre rendimiento y recursos, y no existe una solución única que sea la más rápida y económica al mismo tiempo. La elección del algoritmo adecuado debe considerar cuidadosamente el trade-off de cada opción.

En el caso de estos tres algoritmos, si se busca la coincidencia de distintos patrones en un mismo string, es recomendable utilizar un Suffix Array o un FM-index. Luego, la elección dependerá de si se prioriza la velocidad (lo cual implicará un mayor consumo de espacio) o si se da prioridad al uso eficiente del espacio (lo que podría resultar en tiempos de búsqueda más prolongados). Por otro lado, si se busca el mismo patrón en distintos strings, el algoritmo KMP puede ser la elección más conveniente. En última instancia, la elección del algoritmo óptimo dependerá de los recursos disponibles y de las necesidades específicas de cada aplicación.

REFERENCIAS

[1] GeeksforGeeks. Accedido en 2023. *Suffix Array - Set 2 (nLognLogn Algorithm).* https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/
[2] Wikipedia. Accedido en 2023. *Burrows–Wheeler transform.* https://en.wikipedia.org/wiki/Burrows-Wheeler_transform
[3] Wikipedia. Accedido en 2023. *FM-index.* https://en.wikipedia.org/wiki/FM-index
[4] Wikipedia. Accedido en 2023. *Knuth–Morris–Pratt algorithm.* https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm
[5] Wikipedia. Accedido en 2023. *Suffix array - Applications.* https://en.wikipedia.org/wiki/Suffix_array#Applications

- Código [2]: 2 pts (Buen código, pero evita hacer include de archivos .cpp)- Evaluación experimental [