Boletín 3:
String matching

Nombre del estudiante: Dazhi Enrique Feng Zong
Número de matrícula: 2019435710

Introducción

Este tercer boletín se centra en strings, específicamente en la técnica de coincidencia de patrones (pattern matching). La coincidencia de patrones implica buscar un substring (patrón) dentro de un string original y contar cuántas veces aparece dicho patrón.

Para llevar a cabo la búsqueda de patrones, se emplearán dos enfoques: el uso de un 'suffix array' como estructura de datos y el algoritmo Knuth-Morris-Pratt (KMP).

Un 'suffix array' es una estructura de datos que contiene los índices de los sufijos de un string en orden lexicográfico. Permite realizar búsquedas de patrones de manera eficiente al aprovechar la propiedad de que los sufijos contiguos en el 'suffix array' comparten sufijos iniciales comunes.

El algoritmo Knuth-Morris-Pratt (KMP) es un método eficaz para buscar ocurrencias de un patrón en un string. Se basa en preprocesar el patrón para construir una tabla que indica cómo moverse en caso de que haya una falta de coincidencia entre el patrón y el string durante la búsqueda.

Finalmente, se realizará un análisis experimental. Este análisis tendrá dos partes: la primera, se variará el tamaño del string original manteniendo el tamaño del patrón fijo; la segunda, se mantendrá el tamaño del string original pero se variará el tamaño del patrón.

Ambas implementaciones se obtendrán de *Geek for Geeks*[1][2] y los conjuntos de datos para los experimentos serán extraídos de *Pizza&Chili*[3].

[1] `https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/`
[2] `https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/`
[3] `https://pizzachili.dcc.uchile.cl/texts/dna/`

Desarrollo

Análisis Teórico

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:** Para la búsqueda, dado que se utiliza búsqueda binaria, la complejidad es $O(m \log n)$, donde n es el tamaño del string original y m es el tamaño del patrón. Esto se debe a que se realiza una búsqueda binaria en el 'suffix array' para encontrar ocurrencias del patrón.
- **Espacio:** El espacio que ocupa el 'suffix array' es $O(n)$ para almacenar los índices de los sufijos del string original.

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:** La búsqueda en el algoritmo KMP es lineal, con una complejidad de $O(n)$, donde n es el tamaño del string, debido a que se desplaza el patrón a lo largo del string para encontrar ocurrencias, utilizando la tabla auxiliar previamente construida.
- **Espacio:** El espacio que ocupa el algoritmo KMP para la tabla auxiliar es $O(m)$, donde m es el tamaño del patrón.

Análisis Experimental

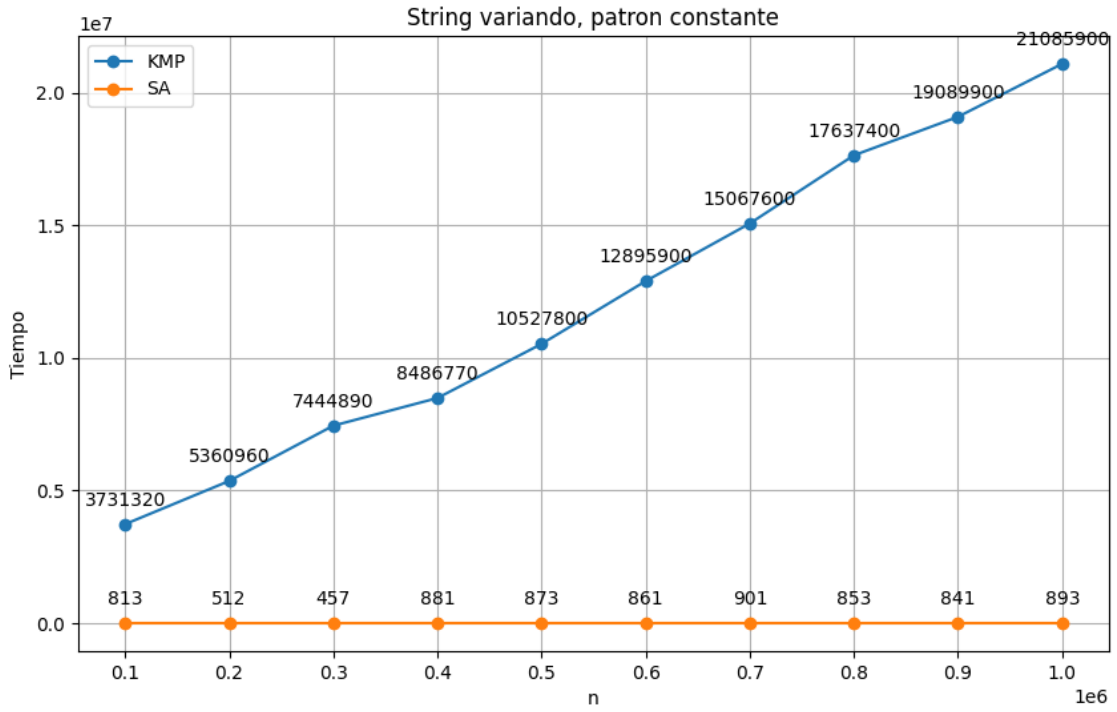
Se llevaron a cabo dos experimentos diferentes para analizar el rendimiento de la búsqueda de elementos en una secuencia. En el primer experimento, se investigó el efecto del tamaño del string en los tiempos de ejecución, manteniendo constante el patrón. El tamaño del string varió, comenzando con 100000 caracteres y sumando 100000 en cada iteración hasta llegar a 1000000. 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 1000000 caracteres del dataset, pero se varió el patrón a buscar. El patrón a buscar es los primeros 100000 caracteres del dataset y sumando 100000 caracteres en cada iteración hasta llegar a 1000000.

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.

Tiempo KMP vs SA



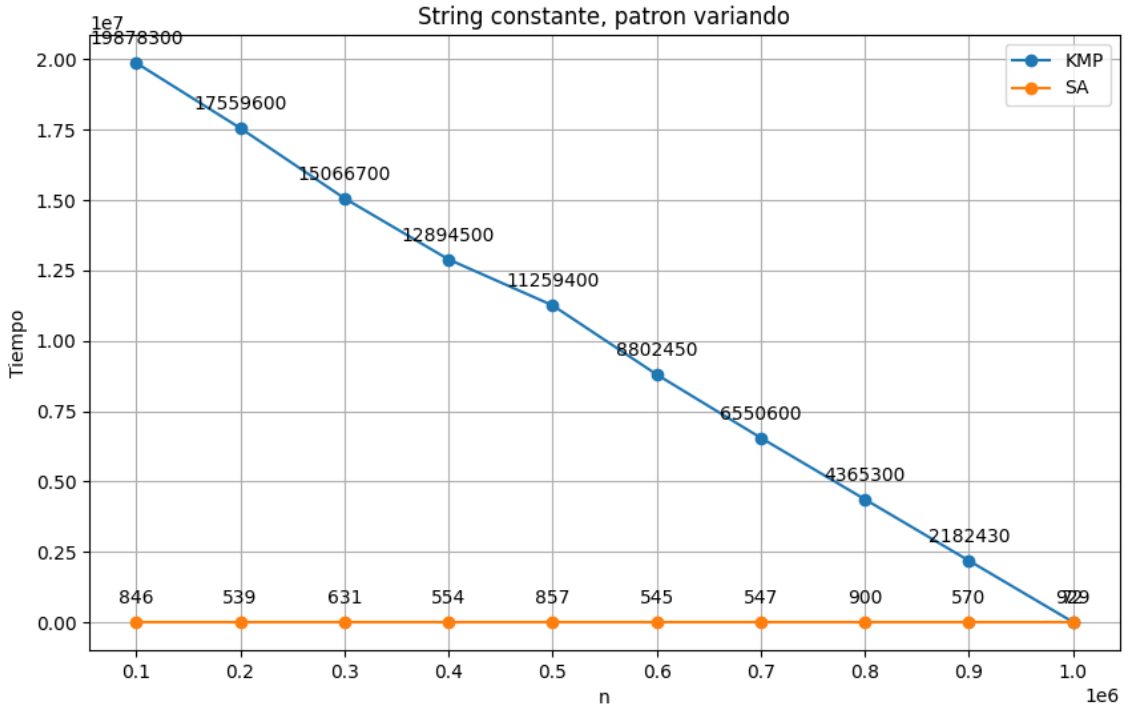


Tabla de los resultados, donde n representa el número de elementos, t representa el tiempo, y v representa la varianza

String de tamaño variado y patrón con tamaño fijo:

n	tsa	vsa
100000	813.067	2689.26
200000	511.9	24158
300000	457.067	5448.4
400000	881.167	2253.34
500000	873.333	1712.49
600000	860.767	7813.05
700000	900.567	430917
800000	853.3	29241.1
900000	841.1	29256
1000000	893.267	10688.5

Cuadro 1: Resultados de los tiempos de ejecución en nanosegundos para **Suffix Array**.

n	tkmp	vkmp
100000	3.73132e+06	2.30086e+13
200000	5.36096e+06	1.59091e+13
300000	7.44489e+06	1.605e+13
400000	8.48677e+06	1.85628e+09
500000	1.05278e+07	4.32889e+09
600000	1.28959e+07	4.49836e+09
700000	1.50676e+07	5.90653e+09
800000	1.76374e+07	8.26678e+12
900000	1.90899e+07	5.43158e+09
1000000	2.10859e+07	2.71857e+09

Cuadro 2: Resultados de los tiempos de ejecución en nanosegundos para **KMP**.

String de tamaño fijo y patrón con tamaño variado:

n	tsa	vsa
100000	845.5	38484.8
200000	538.567	114.179
300000	630.767	414.912
400000	553.5	130.05
500000	856.967	33181
600000	545.267	0.995556
700000	546.867	93.5822
800000	900.267	37471.3
900000	569.6	140.307
1000000	928.9	41546.7

Cuadro 3: Resultados de los tiempos de ejecución en nanosegundos para **Suffix Array**.

n	tkmp	vkmp
100000	1.98783e+07	5.88492e+09
200000	1.75596e+07	8.04745e+09
300000	1.50667e+07	9.01924e+09
400000	1.28945e+07	4.11724e+09
500000	1.12594e+07	2.36366e+09
600000	8.80245e+06	1.54916e+09
700000	6.5506e+06	2.237e+09
800000	4.3653e+06	2.06381e+09
900000	2.18243e+06	8.00066e+08
1000000	71.6	100.04

Cuadro 4: Resultados de los tiempos de ejecución en nanosegundos para **KMP**.

Discusión y Conclusión

Con el análisis experimental, se pudo validar la teoría de que el uso de un 'suffix array' con un algoritmo de búsqueda binaria para la coincidencia de strings tiene una complejidad menor que el algoritmo KMP (logarítmica vs. lineal).

Se observó claramente el comportamiento lineal del algoritmo KMP, en el que el tiempo de ejecución aumentaba de manera proporcional al incremento del tamaño del string original. Sin embargo, la complejidad logarítmica que se esperaba del 'suffix array' no fue tan evidente como se esperaba. Esto podría atribuirse, en parte, al tamaño de los conjuntos de datos utilizados en los experimentos o las optimizaciones del compilador u optimizaciones de algunas funciones de C++.

En el caso de un string con tamaño fijo y un patrón variando e incrementándose, este no cumplió con el análisis teórico para ambos algoritmos. Según la teoría, ambas búsquedas dependen de m, y si m se acerca a n, la complejidad debería aumentar. Sin embargo, este comportamiento no se reflejó en los resultados. De hecho, para KMP, el tiempo disminuía al aumentar el tamaño del patrón. Esto se debe un buble *while* de la implementación que depende de n y m y a medida que aumenta m, se disminuye el número de iteraciones. Para la búsqueda binaria del 'suffix array', esta se mantuvo casi constante (subiendo y bajando muy poco), al igual que el experimento anterior.

A pesar de que el 'suffix array' tiene una mejor complejidad teórica y experimental en este caso, no siempre es la mejor alternativa. Esto se debe a varias desventajas en comparación con el algoritmo KMP.

Una de las desventajas notables es la complejidad temporal de la construcción del 'suffix array'. En este caso, se utilizó una implementación medianamente costosa en términos de tiempo, con una complejidad de $O(n \log n \log n)$. Aunque existen implementaciones que mejoran esto a $O(n)$, nunca será más eficiente que la construcción de KMP, que simplemente implica la creación de la tabla o arreglo 'LPS' con un tiempo $O(m)$ (siendo m el tamaño del patrón).

Otra desventaja es el espacio utilizado por el 'suffix array'. Siempre será lineal, ya que necesita almacenar todos los índices de los sufijos, en comparación con el arreglo 'LPS' de KMP, que solo requiere el tamaño del patrón (m).

La elección entre uno u otro algoritmo dependerá de los recursos disponibles y de la eficiencia que se quiera lograr para un determinado contexto. Por ejemplo, si se van a realizar varias búsquedas en un mismo string, puede ser más eficiente usar el 'suffix array', ya que la construcción se realiza solo una vez, mientras que las búsquedas pueden realizarse varias veces. Por otro lado, si se tienen varios strings y se debe realizar la construcción para cada uno de ellos, es más eficiente utilizar KMP. También es importante considerar el espacio y la dificultad de implementación de cada algoritmo para tomar la desición.