Mini-Proyecto II: Búsqueda en Texto

Diego Seco[1], Meraioth Ulloa Salazar[2], Cristóbal Donoso Oliva[3],
Matías Medina Silva[4],

[1]Docente a cargo de la asignatura[2−3−4]Estudiantes Pre-grado
[1−2]Dpto. de Ingeniería Civil Informática y Ciencias de la Computación
Universidad de Concepción, Concepción, Chile.

Octubre de 2016

Introducción

Uno de los problemas más frecuentes en la cotidianidad, dice relación con el análisis de secuencia de caracteres con el objetivo de identificar ocurrencias de un patrón previamente definido. Así mismo, se han desarrollado algunos algoritmos que van más allá de la fuerza bruta. Diversas técnicas que tienen su origen en el pre-procesamiento de patrones para realizar menos comparaciones, o en su defecto, comparaciones mas eficientes para disminuir la complejidad temporal.

A continuación, revisaremos cuatro algoritmos que resuelven el problema de Patter-Mattching. Haremos experimentos con fines comparativos y analizaremos los resultados.

Descripción del Problema

El problema básico de búsqueda en texto consiste en encontrar todas las ocurrencias de un patrón P de largo m en un texto T de largo n. Existen diferentes soluciones a este problema que tratan de mejorar el algoritmo de fuerza bruta (complejidad $O(nm)$) asintóticamente o en la práctica. En este proyecto se implementará como baseline el algoritmo de fuerza bruta. Además, se implementarán algunos de los algoritmos más conocidos para resolver el problema (Boyer-Moore, Knuth-Morris-Prat, Rabin-Karp, etc.).

Algoritmos

Fuerza Bruta

Este algoritmo resulta ser el más intuitivo pero menos eficiente en términos de complejidad asintótica. Consiste en comparar el patrón en el texto, caracter por caracter, hasta que no queden caracteres por comparar.

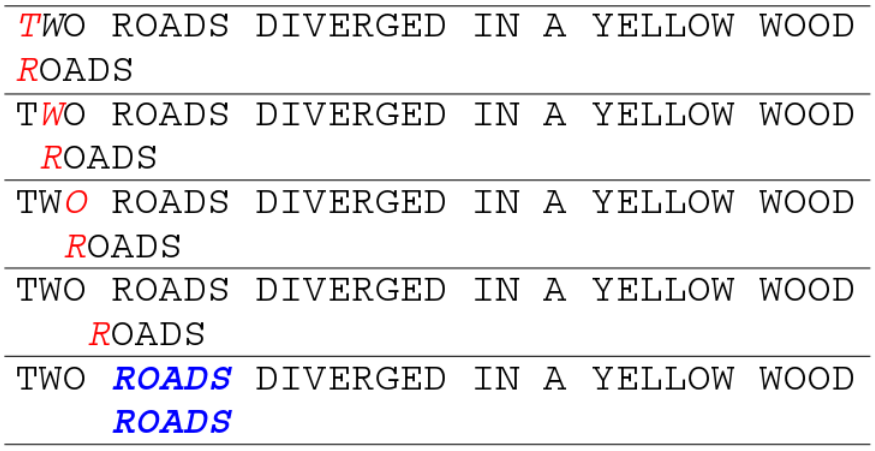

Figure 1: búsqueda de un patrón utilizando fuerza bruta

Algoritmo 1: Fuerza Bruta

```
1 for i=0 hasta i < n do
2     for j=0 hasta j < m do
3         if caracter de texto != caracter del patron then  break
4         else if caracter de texto == caracter del patron then
5             Agregar Ocurrencia del Patrón
6         end
7         ;
8     end
9 end
```

Los algoritmos de fuerza bruta no se preocupan del rendimiento. Podemos ver que en el **peor caso** se debe comparar n veces cada caracter del patrón de tamaño m, lo cual genera una complejidad temporal $O(n \bullet m)$.
En síntesis puede resultar útil el uso de este algoritmo para dominios pequeños donde el primer caracter del patrón no aparezca tan frecuente en el texto y con esto disminuir el recorrido en el patrón de tamaño m.

Boyer&Moore

El algoritmo de Boyer&Moore es un método que basa su funcionamiento en dos principios:

1. Examina el patrón de derecha a izquierda
2. Evita realizar comparaciones de caracteres que no se encuentran en el patrón. Esto quiere decir, que si encuentra un fallo, busca en lo que resta del patrón si se encuentra el caracter, en ausencia de éste se desplaza hasta m posiciones.

En la práctica *Horspool* implementa una simplificación de manera más sencilla, manteniendo la idea original de B&M. Sin embargo, en este trabajo hemos implementado este algoritmo bajo la *heurística del caracter malo*. Consiste en generar una tabla que guarde el índice de los caracteres del patrón. Cuando un caracter del texto produce fallo, se desliza el patrón de tal forma que coincida con la ultima ocurrencia del caracter malo.

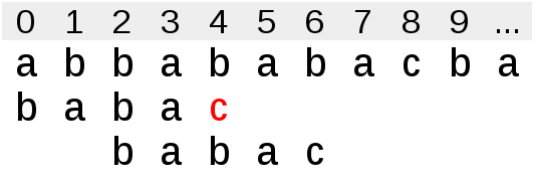

Figure 2: Desplazamiento según heurística del caracter malo

Algoritmo 2: Boyer&Moore

```
1 while i<n-m do
2     j=m-1
3     while caracter del texto == caracter del patrón do
4         j−−
5         if j<0 then Agregar Ocurrencia del Patrón
6         else i+=max.tabla_caracter_malo;
7     end
8 end
```

En general, son necesarias $O(n \bullet m)$ comparaciones si el patrón es a^m y el texto a^n. Sin embargo, si el alfabeto es de gran tamaño comparado con el patrón el algoritmo realiza $O(\frac{n}{m})$ comparaciones en promedio. Esto se debe a que la probabilidad de que un carácter del texto esté en el patrón son reducidas, y por lo tanto es altamente probable el desplazamiento de m espacios.

Knuth Morris Pratt

Este algoritmo trata de encontrar la ocurrencia de un patrón haciendo uso de la información obtenida de las comparaciones previas. Para determinar el desplazamiento a realizar, el algoritmo genera una tabla de fallos que impide procesar aquellos caracteres del texto que ya hayan sido procesados.

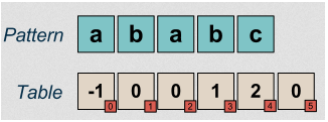

Figure 3: Tabla de Fallo

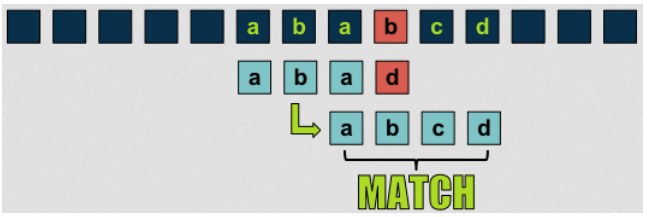

Figure 4: ejemplo de match

Algoritmo 3: Knuth Morris Pratt

```
1  k←0 puntero a T
2  i←0 puntero a P
3  if n>=m then
4      calcular tabla
5      while k + i es menor que la longitud de T do
6          if Patrón[i] = Texto[k + i] then
7              if i es igual a la longitud de P - 1 then
8                  Devolver k
9              end
10             i ← i + 1
11             else k ← k + i - tfallos[i]
12             if i es mayor que 0 entonces then
13                 i ← tfallos[i]
14             end
15             ;
16         end
17     end
18 end
```

Ya que el algoritmo precisa de 2 partes donde se analiza una cadena en cada parte, la complejidad resultante es O(k) y O(n), cuya suma resulta ser O(n + k).

Rabin Karp

Calcula un valor hash para el patrón, y para cada subsecuencia de m caracteres de texto. Si los valores hash son diferentes, se calcula una valor para la siguiente secuencia. Si los valores hash son iguales se usa una comparación de Fuerza Bruta.

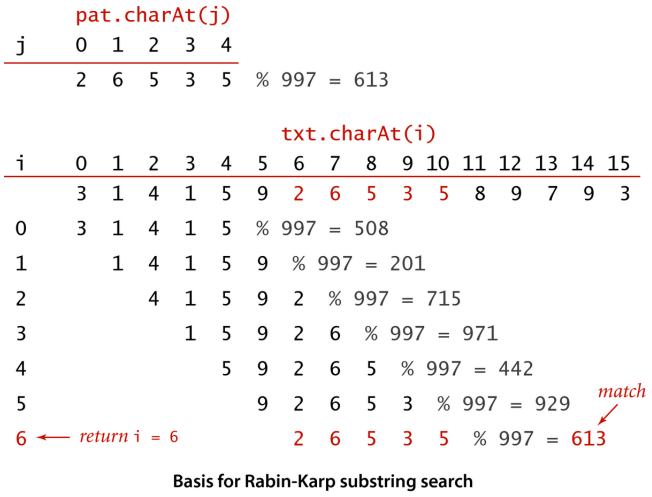

Figure 5: Ejemplo de ejecución Rabin&Karp

Algoritmo 4: Knuth Morris Pratt

```
1 for s = 0 hasta s ≤ n-m; s++ do
2     if hash.patron == hash.texto then
3         if caracter texto == caracter patrón then
4             Ocurrencia
5         end
6     end
7 end
```

R&K es útil para multipatrones, vale decir, busqueda de varias secuencias de caracteres en un texto. La complejidad asociada a este algoritmo es O(n•m)

Experimentos

Para los experimentos se usaron 4 datasets (http://pizzachili.dcc.uchile.cl/index.html), dos de ellos contenían palabras en inglés y el resto secuencias de adn. Para los dos tipos de archivos se utilizó un archivo de 50 Mb y otro de 100Mb.
Se hicieron dos tipos de experimentos sobre los 4 data set, uno generando patrones al azar y otro tomando substrings contenidos en posiciones aleatorias del texto.
Para todos los experimentos se hizo crecer el tamaño de los patrones desde 100 a 10000 caracteres.

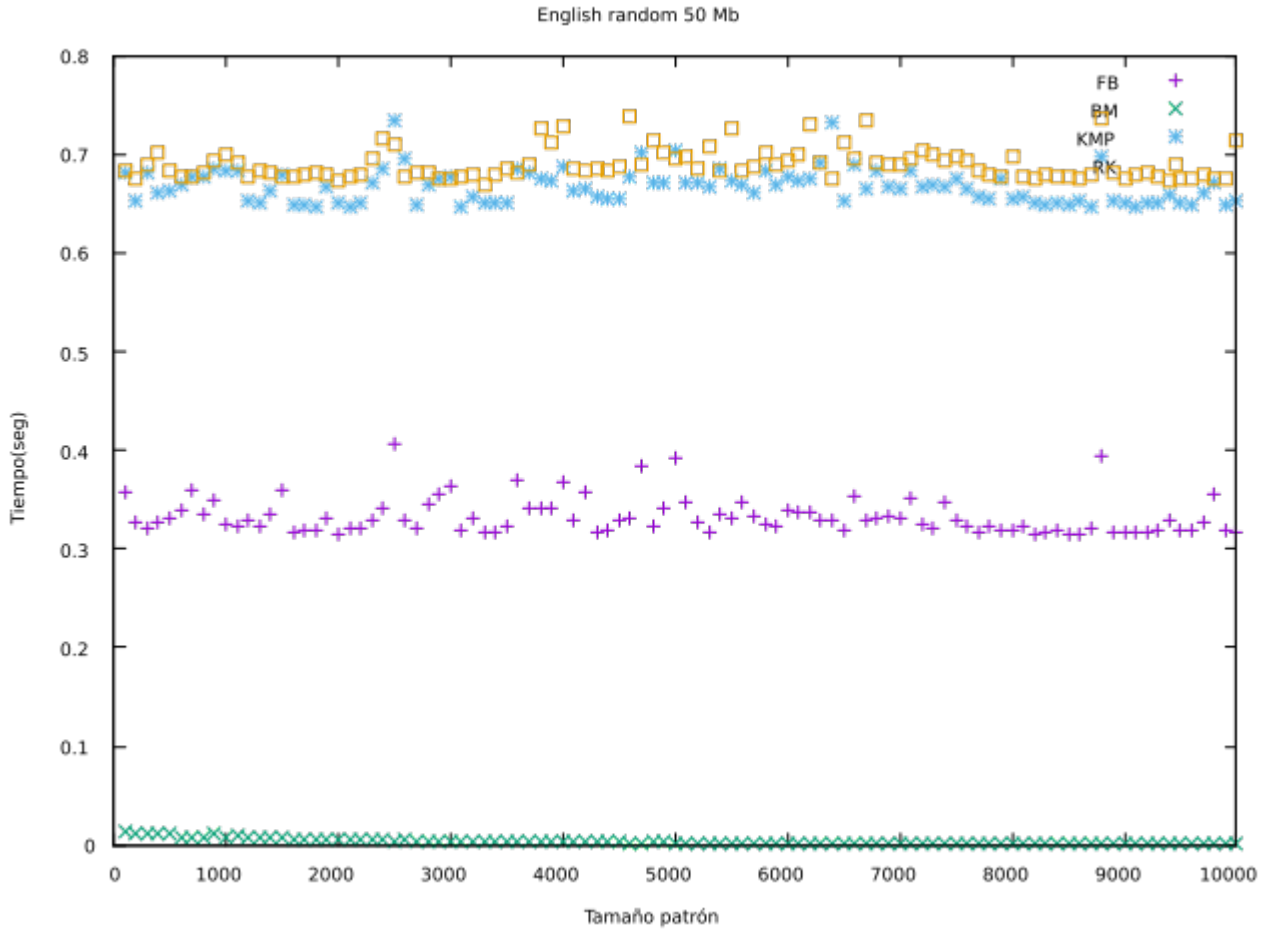

Figure 6: Gráfico tamaño patrón vs tiempo en lenguaje natural

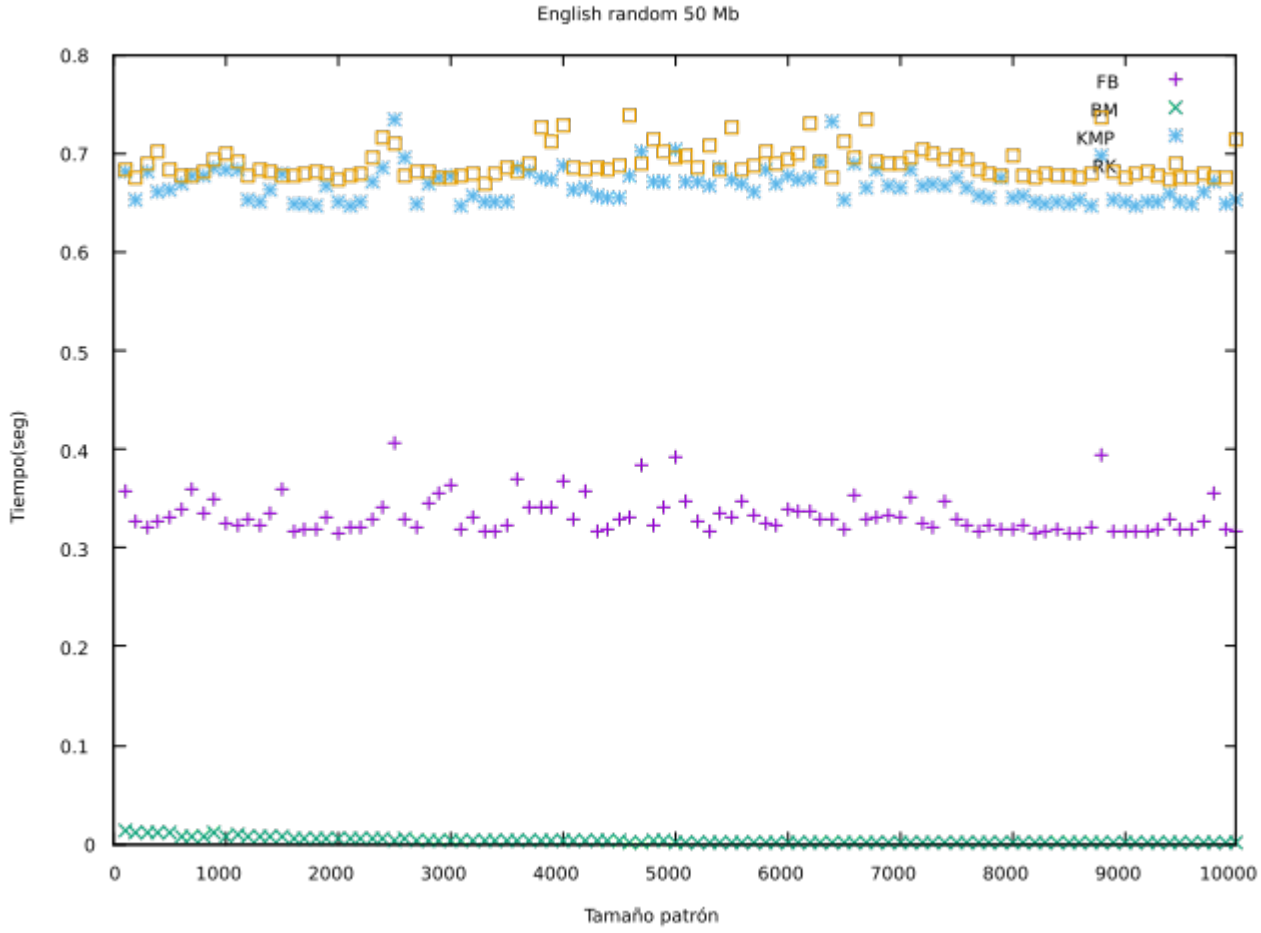

Figure 7: Gráfico tamaño patrón vs tiempo en lenguaje natural

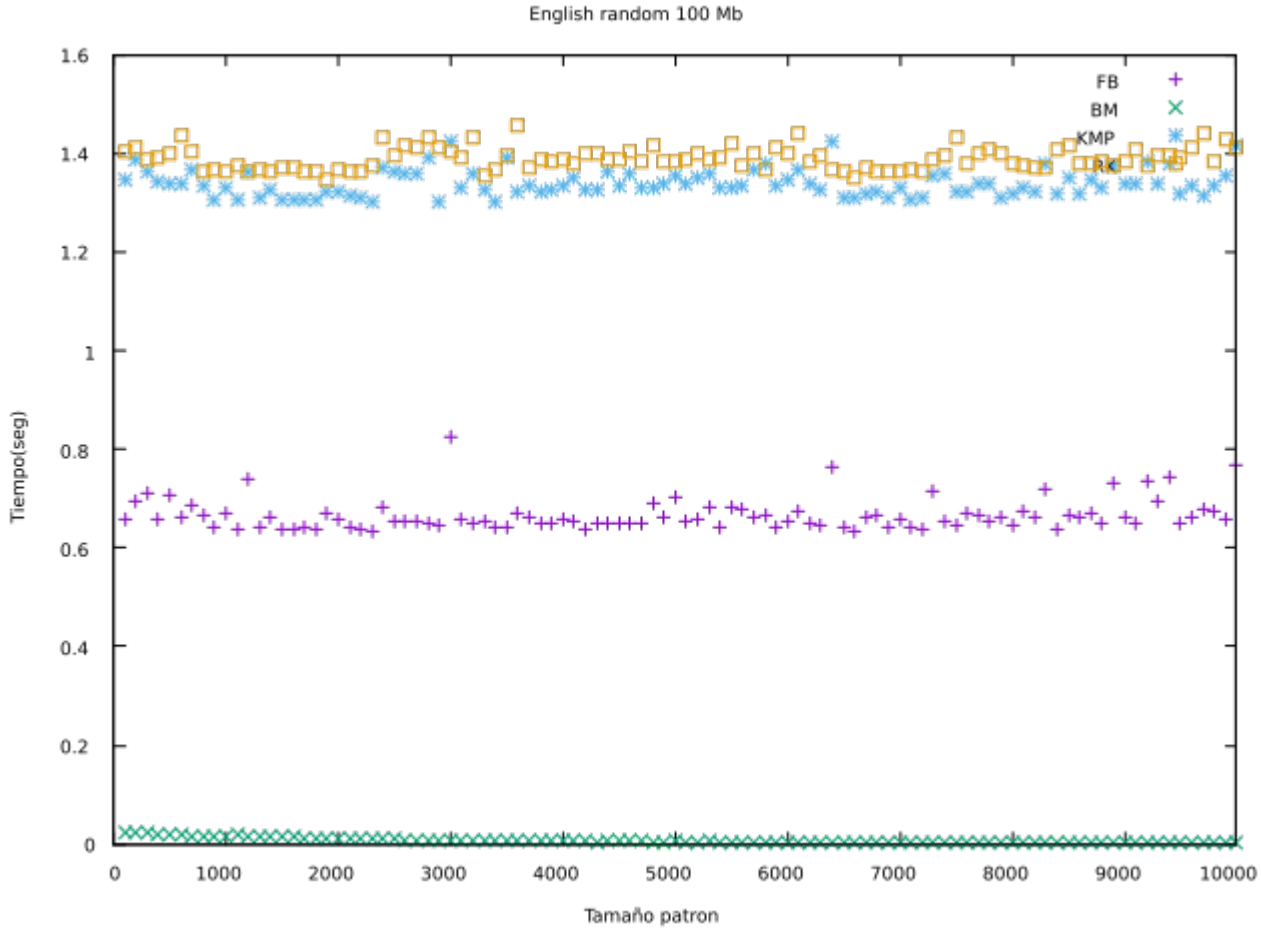

Figure 8: Gráfico tamaño patrón vs tiempo en lenguaje natural

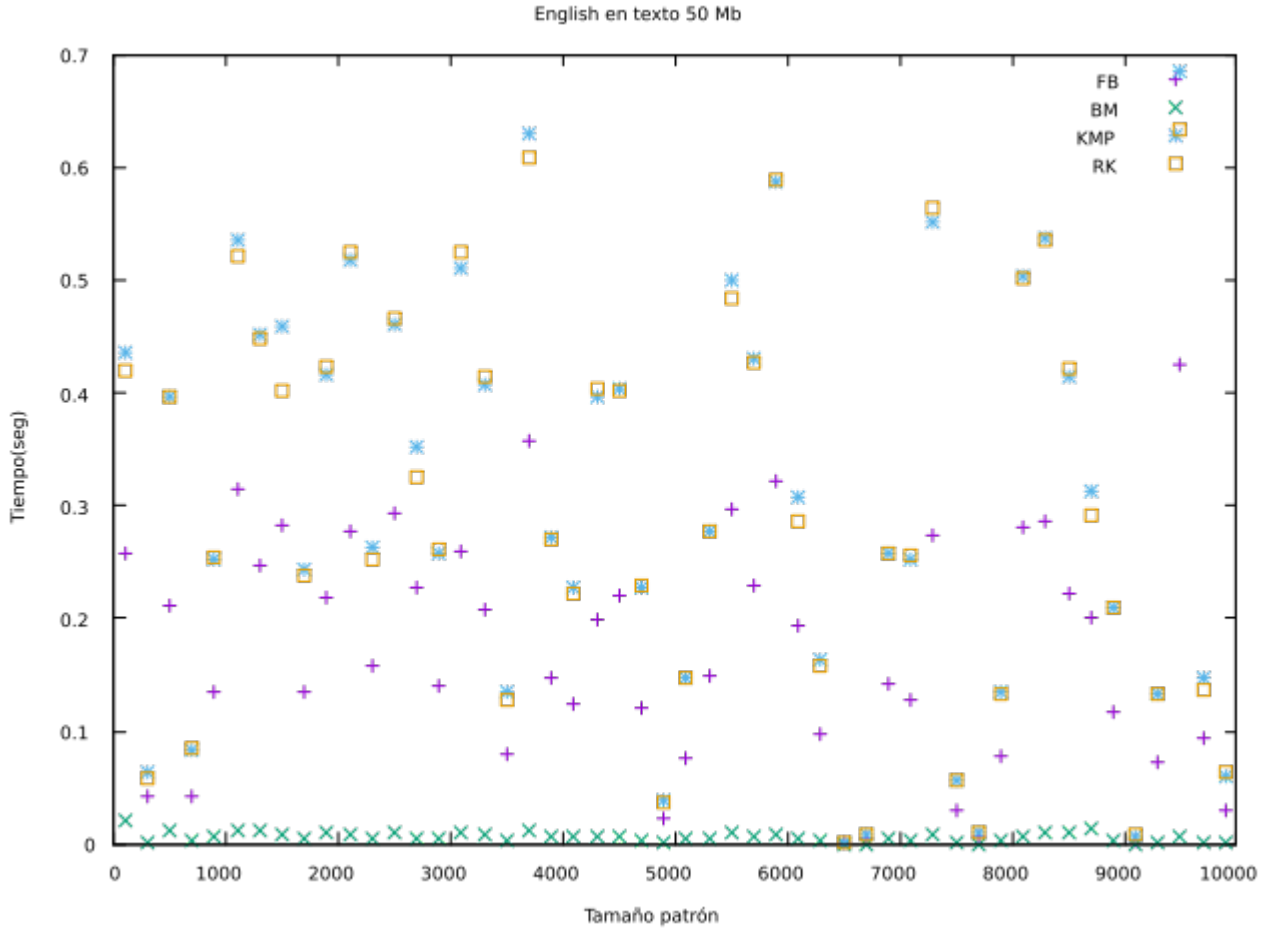

Figure 9: Gráfico tamaño patrón vs tiempo en lenguaje natural

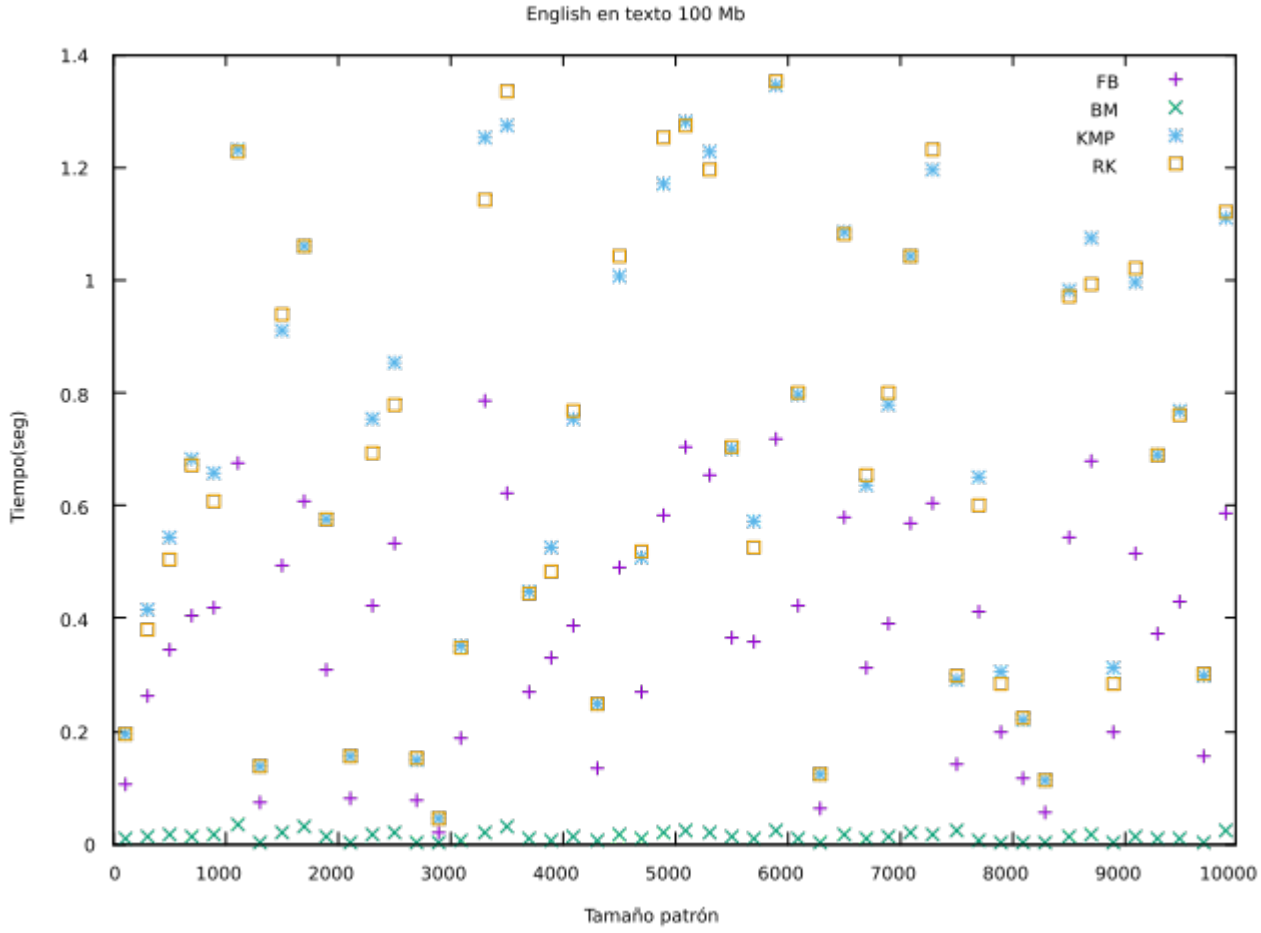

Figure 10: Gráfico tamaño patrón vs tiempo en lenguaje natural

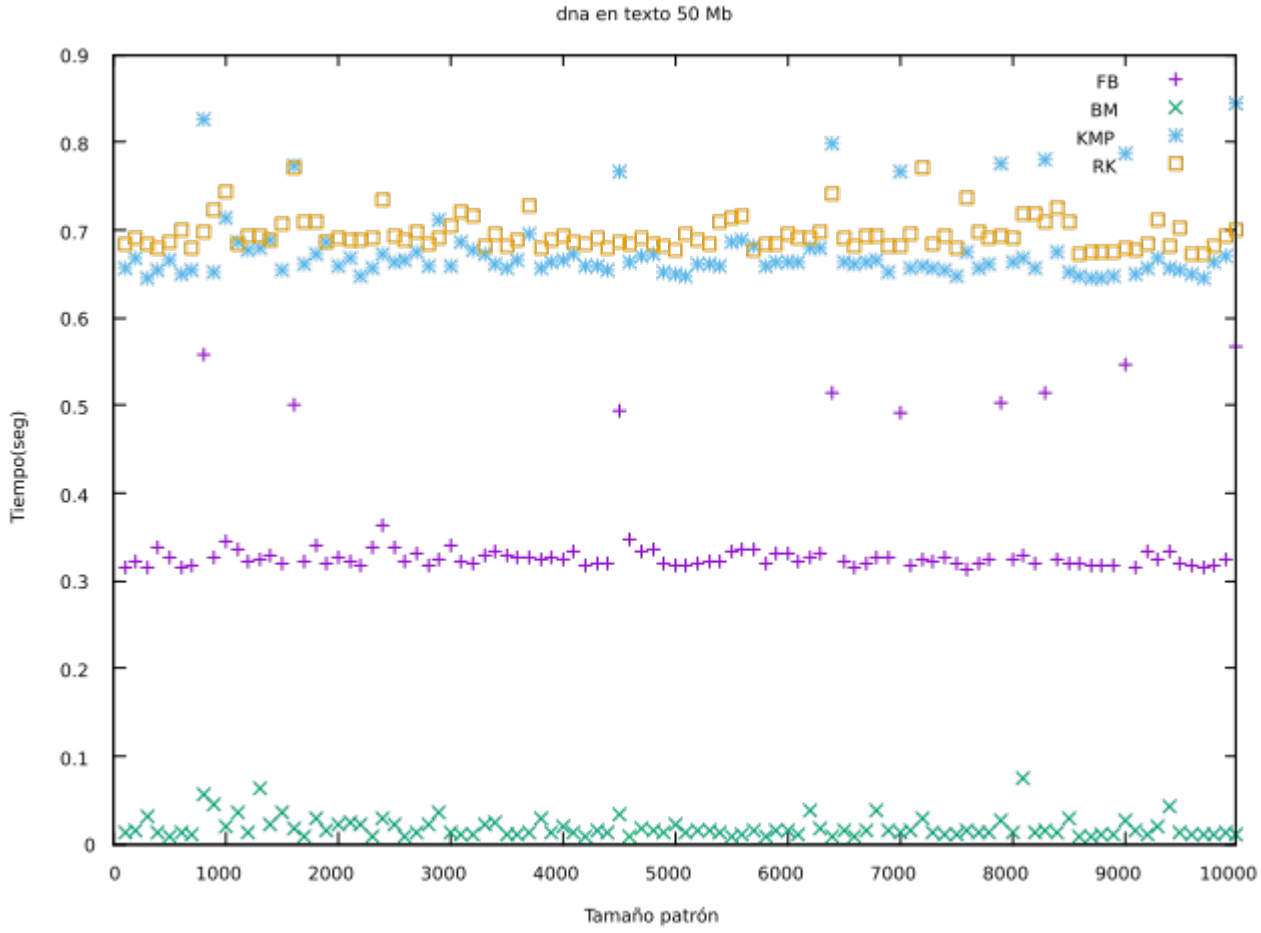

Figure 11: Gráfico tamaño patrón vs tiempo en DNA

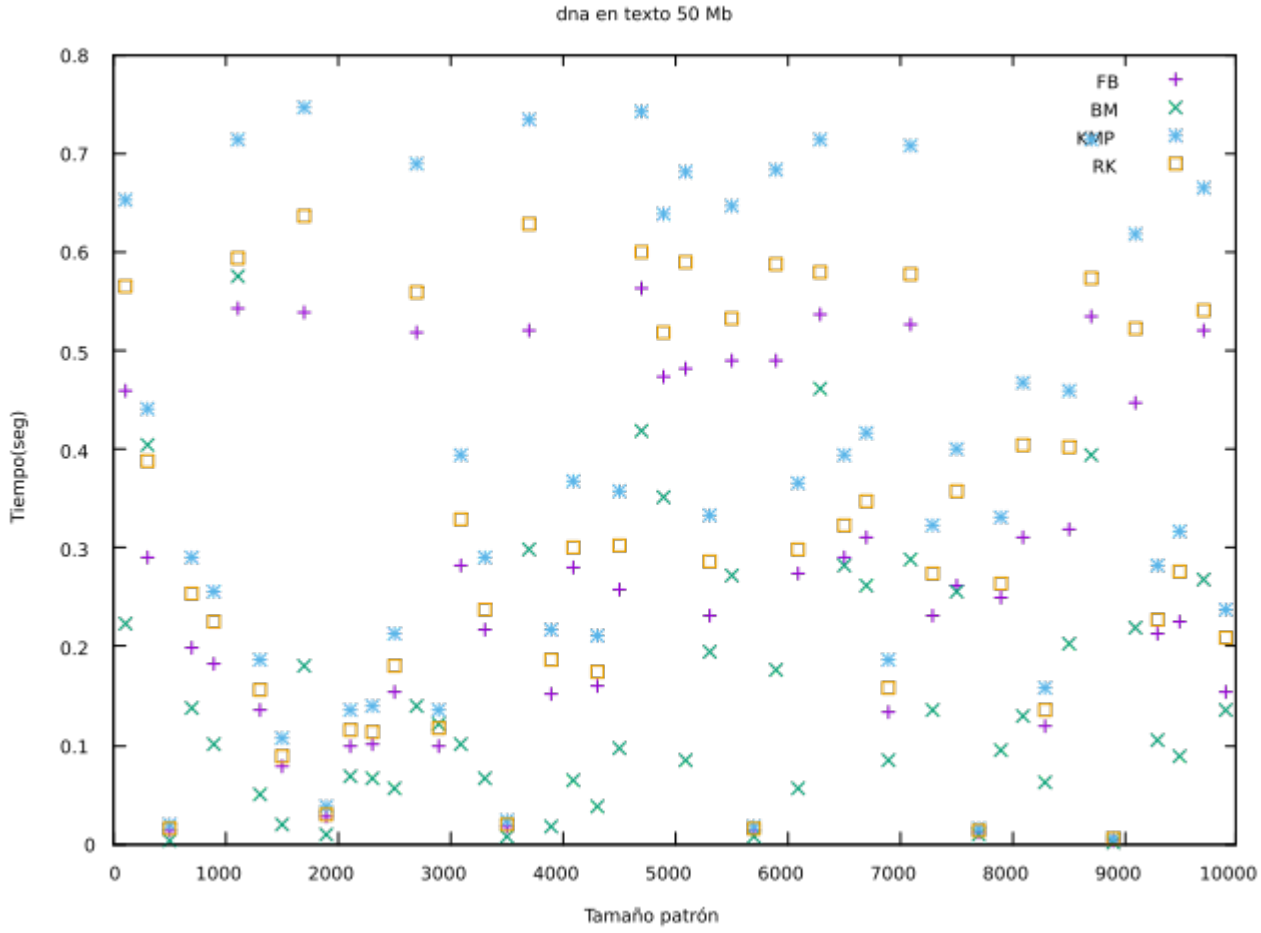

Figure 12: Gráfico tamaño patrón vs tiempo en DNA

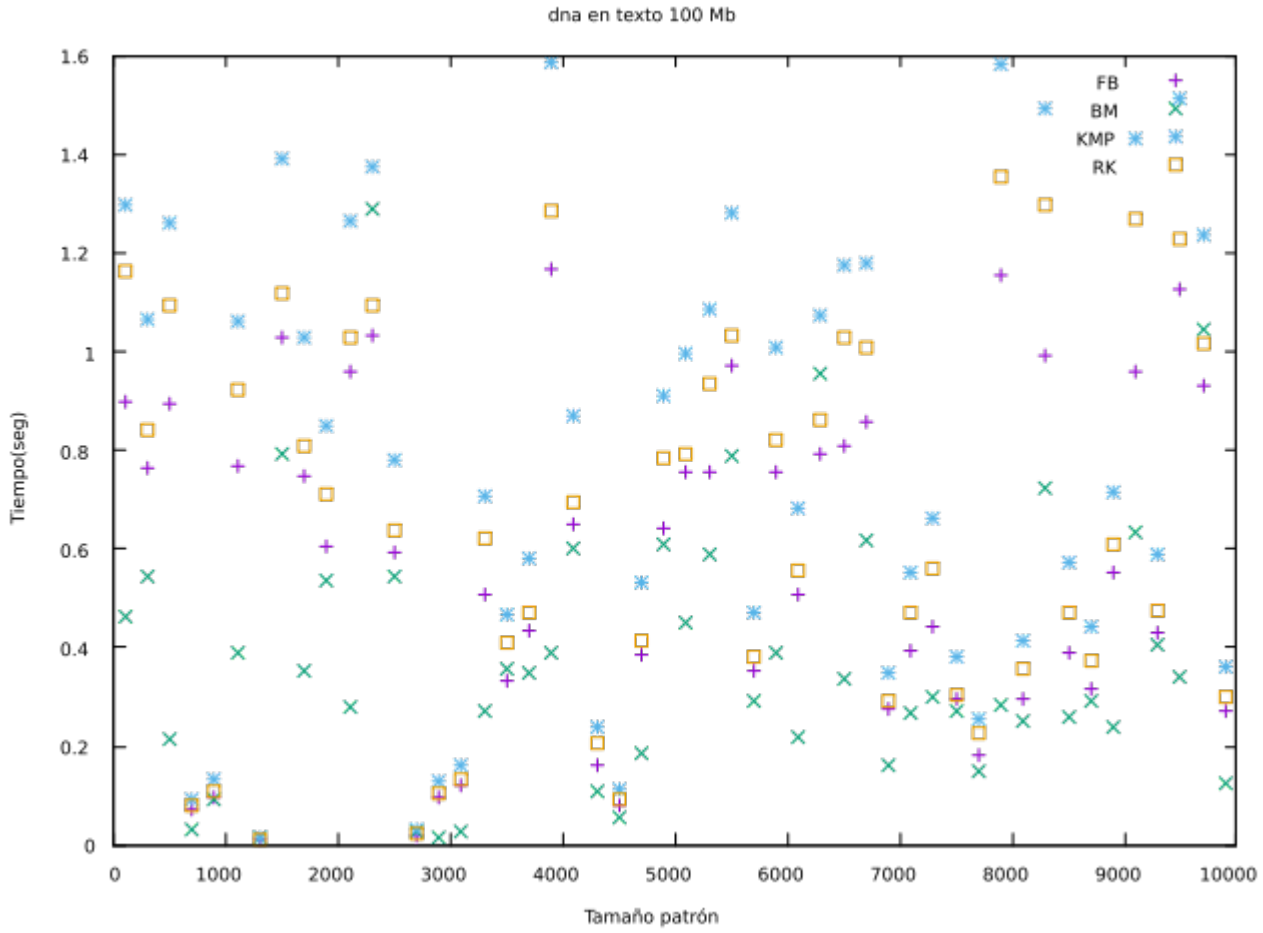

Figure 13: Gráfico tamaño patrón vs tiempo en DNA

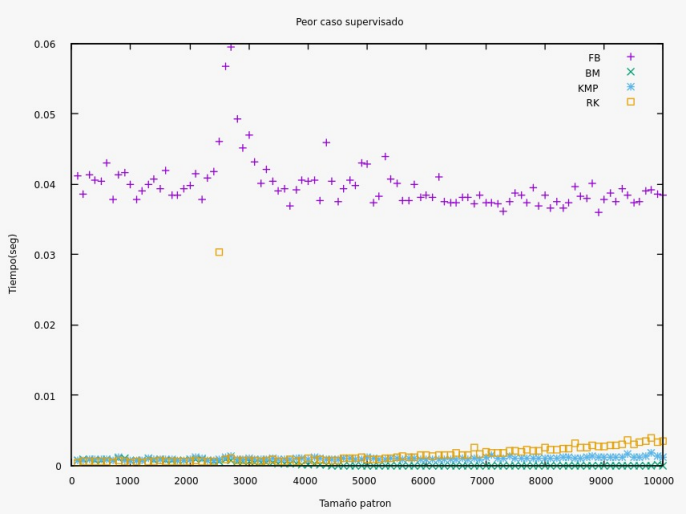

Figure 14: Fuerza Bruta vs Otros Algoritmos

Conclusión

Queda demostrado que el rendimiento de los distintos algoritmos revisados están en función del tamaño del texto y el patrón. El fundamento de los algoritmos recae en la técnica que se aplica para un dominio particular con el fin de establecer condiciones que hacen correr mejor una implementación de otra.
Por otro lado, existen estructuras de datos basadas en tries que optimizan más aun este problema. Hoy se puede realizar búsquedas de manera rápida y utilizando pequeñas porciones de memoria gracias a la compresión y estructuras de datos compactas. Pese a ello los algoritmos son una buena aproximación para situaciones controladas y de bajo volumen de datos.

Bibliografia

https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf

http://www.cs.uku.fi/ kilpelai/BSA05/lectures/slides03.pdf