Comparación de Algoritmos de Búsqueda de Cadenas para la Detección de Patrones en Genomas (Junio 2020)

María Inés Quihillalt, Estudiante, Instituto Tecnológico de Buenos Aires.

***Index Terms*—String Matching Algorithms, Maquinas de Estado Finitas, Complejidad Computacional, Rabin-Karp, Aho-Corasik, Boyer-Moore.**

***Abstracto*—**
Las extensas secuencias moleculares como el ADN tienen una serie de características únicas, cuyo análisis los hacen demandar muchos recursos computacionalmente, requiriendo un gran soporte. En la actualidad, la búsqueda de patrones en genomas es una de las claves del campo de la bioingeniería y los algoritmos de búsqueda están presentes en ella para alivianar la tarea del estudio de información molecular. En este ensayo nos proponemos comparar el tiempo de ejecución de diferentes algoritmos de búsqueda de cadenas en la secuencia de ADN del SARS-CoV2 obtenido el 30 de Marzo de 2020 en la cuidad de Wuhan, China.

A. NOMENCLATURA

BM: Boyer Moore
RK: Rabin Karp
AC: Aho Corasick

I. INTRODUCCIÓN

La búsqueda es sin lugar a duda la operación informática más frecuente, y en particular, su aplicación es vital en diferentes campos de la ciencia. Podríamos argumentar que en gran medida el desarrollo de conocimiento científico es una constante búsqueda de patrones para la explicación de los fenómenos que nos rodean, y siguiendo este razonamiento la informática puede ser un elemento esencial para conseguirlo.

En la actual carrera que la comunidad científica está transitando hacia una vacuna contra el *coronavirus,* la búsqueda de coincidencias entre enfermedades ya conocidas y el COVID-19 es diferencial.

Una manera de lograrlo es mediante la comparación de genomas resultantes de la secuenciación del ADN. Esta secuenciación se basa en determinar el orden de los cuatro componentes básicos químicos, Adenina (A), timina (T), citosina (C) y guanina (G), que forman la molécula de ADN. La secuenciación del genoma humano marcó un hito en la biología, y a partir de entonces los investigadores pueden comparar largos tramos de ADN de distintos individuos relativamente rápido. Tales comparaciones pueden generar una enorme cantidad de información acerca de la función de la herencia en la propensión a enfermedades. (Genome.gov, 2019)

De todas formas, las comparaciones entre millones de genomas existentes demandan muchos recursos computacionales y aún más importante, tiempo, por lo que la informática es imprescindible para acelerar el proceso mediante algoritmos de búsqueda apropiados.

En este ensayo, inspirados en el momento único en la historia que la especie humana se encuentra transitando, analizaremos tres algoritmos de búsqueda de cadenas y su desempeño frente a la búsqueda de patrones en el genoma del Síndrome Respiratorio Agudo Severo aislado del Coronavirus 2, Wuhan-Hu-1.

A. Conceptos Útiles

1) Genoma: El genoma es el conjunto de instrucciones genéticas que se encuentra en una célula. *(Collins, s.f.)* Posee la capacidad para determinar total o parcialmente el funcionamiento del organismo.
2) Los *Algoritmos de Búsqueda de Cadenas* son secuencias de instrucciones bien definidas que intentan encontrar lugares donde una o varias cadenas (también llamados patrones) son encontrados en una cadena aún más grande (texto). Formalmente ambas cadenas son la

concatenación de elementos finitos de un alfabeto. En la aplicación del ADN el conjunto finito será $\sum = \{a, c, g, t\}$.

3) Coincidencia de cadenas (String Matching): Encuentran una o más ocurrencias de un patrón *(Gimel'farb)*.

4) Autómata: Máquina automática programable capaz de realizar determinadas operaciones de manera autónoma y sustituir a los seres humanos en algunas tareas.

II. Algoritmos

Los algoritmos de coincidencia de cadenas se dividen en cuatro tipos según la forma acercarse a los datos dados: 1) algoritmos clásicos; 2) algoritmos de paralelismo de bits; 3) algoritmos de *sufix automata*; 4) algoritmos de *hash*. *(Bekakos, Bowring, Coddington, & Cao, 2008)*

1) *Enfoque clásico:* Estos algoritmos están basados en la estricta comparación entre caracteres del texto con caracteres del patrón.
2) *Enfoque de paralelismo de bits*: Utiliza el paralelismo intrínseco en la manipulación de bits de la computadora para llevar a cabo muchas operaciones de comparación en paralelo. Se caracteriza por su simplicidad, flexibilidad y el hecho de que no utiliza un buffer.
3) *Enfoque de Suffix Automata:* Estos algoritmos utilizan una poderosa estructura de datos llamada *suffix automaton* que permite resolver muchos problemas relacionados con cadenas. Se podría entender a *suffix automaton* como una forma comprimida de todas las subcadenas de una cadena mayor dada.
4) Enfoque de *hashing*: Estos algoritmos utilizan *hashing* para obtener un método simple de evitar un número cuadrático de comparaciones de caracteres, el peor caso práctico en la mayoría de las situaciones.

Para esta investigación elegimos tres algoritmos con diferentes enfoques. En primer lugar, el algoritmo con enfoque clásico de Boyer Moore, en segundo lugar, el algoritmo con enfoque de *hashing* Rabin-Karp y en tercer lugar el algoritmo con enfoque de *suffix automata* Aho-Corasick.

A) Boyer – Moore

El algoritmo Boyer Moore fue desarrollado en 1977 por Rober S. Boyer y Strother Moore. Este algoritmo realiza comparaciones con dos métodos: El desplazamiento ante un **carácter incorrecto** (*bad character heuristic*) o el desplazamiento ante un **sufijo correcto** (*good suffix heuristic)*. (Storer, 2001)

El método de la heurística del "carácter incorrecto" se basa en una regla que determina que, al encontrar una discordancia entre un carácter del texto y uno del patrón en cierta posición, a ese carácter del texto se lo llama "carácter erróneo" y se desplaza el patrón para que ese carácter erróneo se alinee con un carácter del patrón que sí sea igual a él.

De todas formas, algunas veces el método anterior puede fallar y no logra provocar un desplazamiento o provoca un desplazamiento negativo. En estos casos la heurística del "sufijo correcto" usa los caracteres del patrón que coinciden con el texto y nos dice cuando hay que desplazar el patrón hacia adelante.

La heurística del carácter erróneo es más sencilla de implementar y más rápida que la del sufijo correcto (Heung) por lo que en este trabajo utilizamos un programa con el algoritmo de Boyer Moore que utiliza el método basado en desplazamiento ante un sufijo incorrecto.

- Procesamiento de los caracteres erróneos

Como ya mencionamos, el algoritmo compara carácter a carácter el texto con el último carácter a la derecha del patrón y mueve el patrón al encontrar una discordancia. Veamos un ejemplo en una pequeña porción del genoma analizado y el patrón "gcc". Señalamos con verde las coincidencias y con rojo las discordancias:

0	1	2	3	4	5	6	7
t	g	c	a	g	g	c	c
g	c	c					

Al encontrar una coincidencia se prosigue a comparar los caracteres en la posición 1. Luego encuentra que hay una discordancia y busca en el patrón alguna letra que sí coincida, en este caso la g en violeta. Se desplaza al patrón una posición a la derecha:

0	1	2	3	4	5	6	7
a	g	c	a	g	g	c	c
	g	c	c				

Luego al desplazar el patrón, el último carácter a la derecha no coincide, y al no haber carácter "a" en el patrón, se lo desplaza por completo, es decir 3 posiciones:

0	1	2	3	4	5	6	7
a	c	a	a	g	g	c	c
				g	c	c	

Con el desplazamiento se encuentra una discordancia en la posición 5 y se busca la letra "g" en el patrón, al encontrarla se desplaza "gcc" una posición:

0	1	2	3	4	5	6	7
a	c	a	a	g	g	c	c
					g	c	c

Finalmente, no hay ninguna discordancia por lo que el patrón se encuentra en el texto y se guarda este resultado.

En la figura 1 se encuentra disponible un diagrama de flujo que refleja el procedimiento del algoritmo.

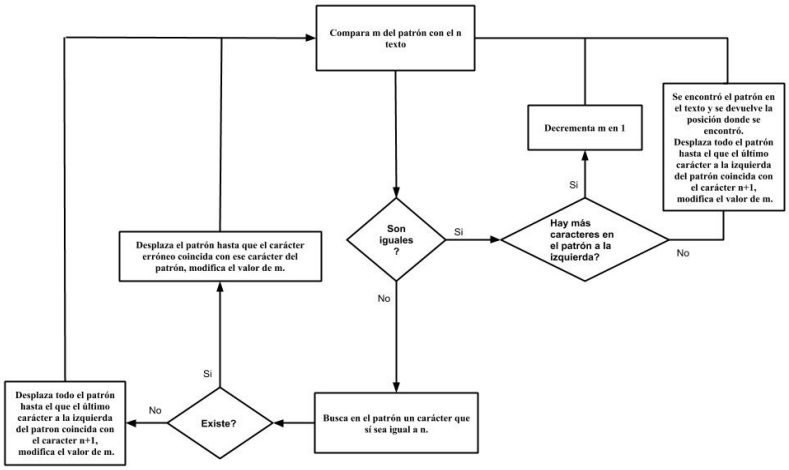


Figura1. Flujo del algoritmo de Boyer Moore con heurística del carácter erróneo.

B) Rabin – Karp

El siguiente algoritmo fue desarrollado por Richard M. Karp y Michael O. Rabin en 1987. Se caracteriza por acelerar el proceso de búsqueda mediante una función *hash* para calcular valores *hash* para todos los caracteres del texto y del patrón. Así el valor *hash* del patrón se compara con valores hash de subcadenas del texto. En caso de no coincidir los valores hash, el algoritmo desplaza el texto y se elige una nueva subcadena para realizarle el hash para comparar y se repite el proceso. Cuando los valores hash coinciden entonces se realiza una comparación *Brute Force* para verificar la coincidencia. De esta forma se logra acelerar el proceso ya que hay solo una

comparación por secuencia de texto y se requiere de menos análisis carácter por carácter (Cormen, 2001).

- Función hash

En la adaptación al C++ utilizada, para obtener los valores hash utilizamos el operador módulo y un número primo predefinido (q) para calcular lo que llamaremos el "patrón modular" (P). Luego se evalúan los primeros m (m=P) caracteres del texto T para computar el patrón numérico del texto. Luego se dividen ambos patrones numéricos por q y si los restos son iguales proseguimos a compararlos. (Ashish Prosad & Rabi Narayan, 2014).
La función hash utilizada para obtener los patrones modulares es la siguiente:

```
for ( i = 0; i < M ; i++)
        {
            p = (d * p + pat[i]) % q;
            t = (d * p + txt[i]) % q;
        }
```

Figura 2. Función hash de Rabin-Karp. Siendo:
pat: patrón de caracteres a buscar.
txt: texto donde buscaremos el patrón (en nuestro caso el genoma).
p: valor hash (patrón modular numérico) de pat.
t: valor hash (patrón modular numérico) de txt.
q: número primo.

Veamos un ejemplo de cómo se realizan las comparaciones de los valores hash. Sea el patrón "gcc" cuyo valor hash computado es 14, proseguimos a obtener los valores hash de las subcadenas del texto que serán del tamaño m=3. En cada ciclo se arma el valor hash de la subcadena, se lo compara con el valor hash del patrón, y de no coincidir se vuelve a calcular un valor *hash* pero de otros m caracteres del texto. En la figura 3 podemos ver los patrones numéricos obtenidos con la función hash, donde el valor de los últimos tres caracteres coincide con el valor hash del patrón que buscábamos.

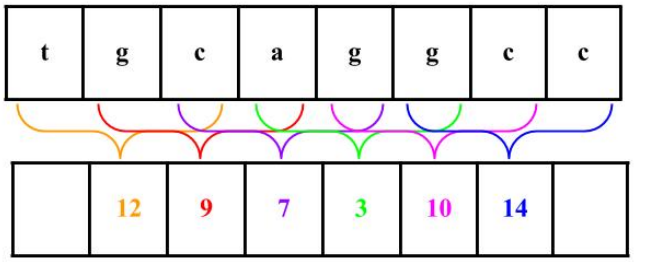


Figura 3. Representación de los valores hash calculados en el texto.

Luego se prosigue a comparar carácter por carácter al subconjunto "gcc" del texto con el patrón y habremos encontrado una coincidencia que se devolverá como resultado.

C) Aho – Corasick

El algoritmo desarrollado por Margaret J. Corasick y Alfred V. Aho en los Laboratorios Bell en 1975 propone un algoritmo eficiente combinando ideas del algoritmo Knuth-Morris-Pratt (Knuth, Morris, & Pratt, 1974) con el uso de una máquina de estados finita. Este algoritmo fue usado para mejorar la velocidad de un programa de búsqueda bibliográfica, logrando una mejora por un factor de 5 a 10 (Aho & Corasick).
El algoritmo consiste en dos etapas principales:

1. La construcción de una máquina de estados a partir de las palabras claves a buscar, es decir los patrones que queremos encontrar en el texto.
2. Aplicación del texto como el valor de entrada de la máquina de estados, que señalará cuando encuentre una coincidencia.

- *Pattern Matching Machine*

Sea K un conjunto de patrones a los que llamaremos **palabras claves** y sea X el texto donde buscaremos las palabras en K. Una máquina de estados para K es un programa cuya entrada es el texto X y produce como salida las posiciones x donde se encuentran las palabras de K. Los estados son definidos como números y se forman a partir de las palabras. La figura 4 muestra un ejemplo de máquina de estados de un conjunto K = {hers, his, she }

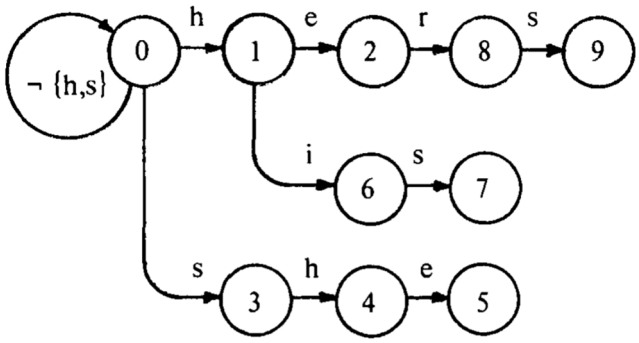


Figura 4. Pattern matching machine para el conjunto K.

Los estados se señalan con números y los eventos son los caracteres que se toman del texto. Una función *goto g* une un par formado por un carácter y un estado con otro estado o el mensaje de *fail.* En el ejemplo de la figura uno la función la función funcionaria en el estado cero de la siguiente forma, al recibir un carácter "h" g(0,h) = 1, indicando que con dicho input se pasara al estado 1. En el caso que reciba por ejemplo una "s" g(0,s) = *fail.* Los estados que indican que una palabra clave ha sido encontrada son denominados *output states* y poseen una función una función (*o*) que formaliza la coincidencia.
En la figura 5 resumimos el funcionamiento de la maquina con un algoritmo.

Algorithm 1. Pattern matching machine.
Input. A text string $x = a_1 a_2 \cdots a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function *output*, as described above.
Output. Locations at which keywords occur in x.
Method.

```
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, a_i) = fail do state ← f(state)
            state ← g(state, a_i)
            if output(state) ≠ empty then
                begin
                    print i
                    print output(state)
                end
        end
end
```

Figura 5. Cada iteración a través del ciclo *for* representa un ciclo operativo de la máquina (Aho & Corasick).

Sea s un estado y a un carácter, un ciclo de la máquina transcurre de la siguiente forma:

1. *Si g(s,a) =s' la maquina hace una transición con la función goto. Luego entra al estado s´ y el siguiente carácter, sea r,*

se convierte en el nuevo carácter a evaluar. Si el output de un estado es vacio entonces la máquina emite el evento de que se halló el patrón. Indica la posición y se termina el ciclo, vuelve al estado 0 hasta que llegue un nuevo carácter.

2. *Si g(s,a) = fail la maquina consulta a la función de error f. Si f(s) = s' la máquina repite el ciclo con s' como el estado actual y a como el carácter a evaluar.*

- Estructuras

Un Trie es una estructura de datos especial para guardar cadenas que puede visualizarse como grafo. AC utiliza la estructura trie para representar los estados como nodos, donde cada rama del árbol está señalada por una letra, como se muestra en la figura 4. Todas las ramas salientes de un nodo deben ser diferentes y existen un máximo de 26 hijos por nodo[1]. Es importante notar que este algoritmo no funciona con números, pero esto no es un inconveniente para nuestro análisis del genoma.

Los vértices del trie pueden ser interpretados como estados en un autómata finito determinista, donde para cada estado en que se encuentre el autómata, y con cualquier símbolo leído no existe más de una transición posible desde ese estado y con un determinado símbolo[2].

III. Complejidad Computacional

En este ensayo tomaremos al tiempo de ejecución como medida de eficiencia ya que la complejidad computacional está directamente relacionada con el mismo. Para esto mediremos el desempeño de los tres algoritmos frente a secuencias de ADN determinadas por los componentes adenina (a), timina (t), citosina (c) y guanina (g). Las complejidades computacionales de búsqueda de los algoritmos utilizados se encuentran resumidos en la tabla 1. Nótese que no tomaremos en consideración la complejidad computacional de pre procesado.

Algoritmo	Caso Promedio	Peor Caso
Boyer Moore	O(m)	O(m*n) / O(m+n)
Rabin Karp	O(n+m)	O((n-m+1)m)
Aho Corasick	O(n+m+k)	O(n+km)

Tabla1. Complejidad de algoritmos
n: largo del texto
k: cantidad de coincidencias
m: largo del patrón

El algoritmo Boyer Moore es efectivo al lograr un caso promedio de complejidad lineal. En cuanto al peor caso a pesar de ser O(m*n) en algunos casos si determinadas condiciones su pero caso es capaz de ocurrir con complejidad O(m+n) (Ashish Prosad & Rabi Narayan, 2014).

En el caso del algoritmo RK si se utiliza un numero primo lo suficientemente grande para la función hash, dos valores de hash para patrones diferentes van a ser distintos. El peor caso es posible en caso de usar un numero primo muy pequeño para el *hashing* o en el caso de que el patrón y el texto sean exactamente iguales por lo que se realizaran todas las comparaciones. (Cormen, 2001)

El algoritmo AC es de los más eficientes a la hora de buscar muchas palabras en un mismo texto, mientras que con otros algoritmos deberíamos llevarlos a cabo una vez por cada patrón que quisiéramos buscar, es decir modificando la complejidad de O(X) a O(X*N + L), siendo X la complejidad original, N la cantidad de patrones y L la longitud promedio de los patrones. Con AC más patrones no significan un cambio sustancial en el tiempo de ejecución del código, y su complejidad resulta O(n+m+k) [2].

IV. Análisis

Utilizaremos la secuencia del genoma completo del Coronavirus 2 disponible en el National Center for Biotechnology Information[3]. Las variables independientes del análisis serán los patrones de texto y el tiempo será la variable dependiente que utilizaremos para comparar los algoritmos. Se medirá el factor del tiempo de ejecución con funciones de la biblioteca chrono de C++, midiendo sólo el tiempo de ejecución de la siguiente forma:

```
/********* START TIMER **********/
auto start = high_resolution_clock::now();

/********* ALGORITMO **********/
search(pat, txt, q);

/********* END TIMER **********/
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);

cout << endl << "COUNT = " << duration.count() << endl;
```

Siendo "COUNT" el resultado que recibiremos por consola del tiempo en microsegundos transcurrido durante la búsqueda.

El análisis se realizará con dos variaciones. En la primera nos enfocaremos en el rendimiento de los algoritmos ante el incremento de tamaño de patrones. Los patrones que se evaluarán se encuentran detallados en la tabla 2.

Patrón	Largo (Caracteres)	Ocurrencias
cg	2	439
agt	3	507
gcgt	4	37
agtgt	5	51
ataaaa	6	14
ccataac	7	1

Tabla 2. Patrones analizados en la parte 1 del análisis.

La segunda parte consiste en la comparación de patrones del mismo largo, cuatro caracteres, con diferente número de ocurrencias, especificados en la tabla 3.

Patrón	Ocurrencias
gggg	15
tgag	90
agtg	146
gtta	179
atga	187

[1] https://www.hackerearth.com/practice/data-structures/advanced-data-structures/trie-keyword-tree/tutorial/

[2] https://cp-algorithms.com/string/aho_corasick.html

[3] https://www.ncbi.nlm.nih.gov/

aaaa	281
tttt	299

Tabla 3. Patrones analizados en la parte 2 del análisis.

El repositorio de Github con el proyecto de se encuentra disponible en el apéndice 1. Las adaptaciones de los algoritmos a C++ se obtuvieron del Sanfoundry's Global Learning Project[4].

V. Resultados

A) Parte I: Búsqueda de patrones con longitud creciente

En una primera instancia al comparar los tiempos de ejecución de los diferentes algoritmos, sin tomar en consideración la cantidad de coincidencias de los patrones en el genoma, se obtuvieron los siguientes resultados:

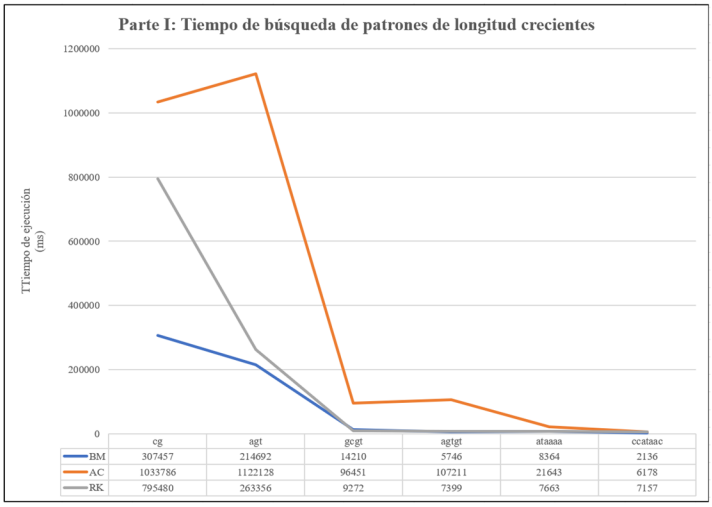


Figura 6. Resultados parte I sin considerar número de ocurrencias.

Es evidente una tendencia en cuanto a los rendimientos de los algoritmos en los patrones más cortos, con el algoritmo de BM llevando una clara delantera en la cadena "cg" y el algoritmo AC completando la búsqueda en 20,000 microsegundos más que el algoritmo RK. A partir del patrón de 4 caracteres de largo, la diferencia en los tiempos de ejecución no es muy significativa entre los algoritmos, en especial entre el RK y BM.

El salto en tiempo de búsqueda del algoritmo AC es interesante, ya que no cumple con la tendencia decreciente observada en los tres algoritmos, esto indicó que habría que considerar el número de ocurrencias de cada patrón para comprender correctamente los comportamientos de los algoritmos.

En la figura 7 presentamos los mismos resultados que en la figura 6 pero esta vez dividiendo el tiempo de búsqueda por la cantidad de ocurrencias de los patrones en el genoma.

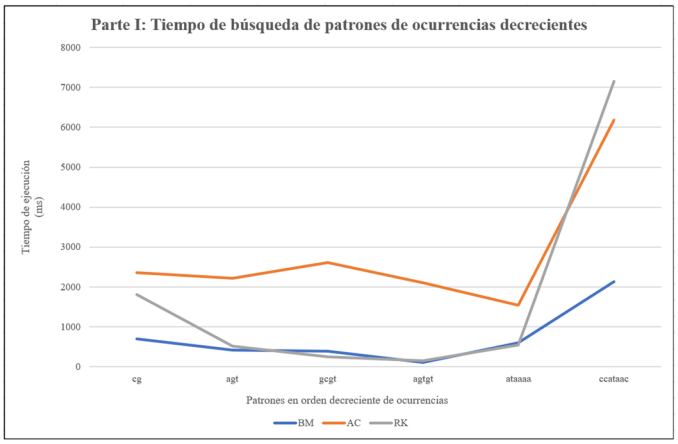


Figura 7. Resultados de parte I considerando número de ocurrencias.

A mayor cantidad de ocurrencias, como lo es en el caso del patrón "cg" con un total de 439 ocurrencias, la tendencia de rendimientos presente en la figura 6 sigue presente, con el algoritmo BM siendo el más eficiente. Pero a medida que disminuye el número de ocurrencias de los patrones notamos un cambio en la eficiencia de los algoritmos, terminando con el BM como el menos eficiente para la cadena de 7 caracteres de longitud y tan solo 1 ocurrencia. El algoritmo RK resultó ser el más eficiente en términos de tiempo de búsqueda por número de ocurrencias seguido por el AC, aunque la diferencia de tiempo entre ellos dos resulta ser poco significativa.

B) Parte II:

Luego de observar los resultados de la parte I era de esperar que el algoritmo AC fuera el que demande un mayor tiempo de ejecución. Aun así, es interesante notar que en los resultados de la parte II en un comienzo los algoritmos AC y RK requirieron casi la misma cantidad de tiempo, pero a medida que incrementaban las ocurrencias, la tasa de cambio del tiempo requerido por AC fue mucho mayor que la de los otros dos algoritmos. En la figura 8 podemos observar como en los tres algoritmos hay una tendencia creciente a medida que aumentan las ocurrencias, pero la pendiente del algoritmo AC es mucho mayor a aquellas del algoritmo BM y RK. En la línea de tendencia del algoritmo BM podemos observar un valor atípico para el patrón "aaaa" con 281 ocurrencias.

En la sección de discusión desarrollaremos en profundidad las explicaciones de los resultados obtenidos en la parte I y II.

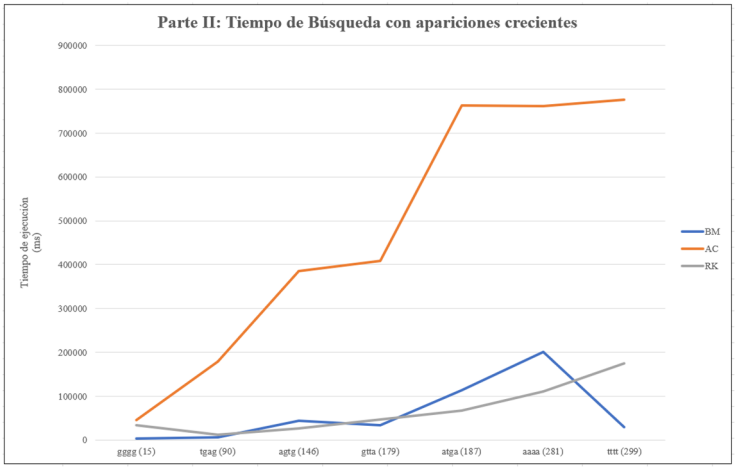


Figura 8. Resultados de parte II.

[4] https://www.sanfoundry.com/

VI. Discusión

En ambas variaciones de las pruebas fue notable como los algoritmos RK y BM superaron en eficiencia significativamente a AC. El BM obtuvo mejores resultados que el RK en casi todas las búsquedas por lo que podríamos argumentar que es el más eficiente para el análisis de patrones en genomas. De todas formas, cabe hacer una importante distinción sobre el algoritmo AC. Era de esperar que demandara más tiempo que los otros algoritmos dado a que necesita de más pasos y acciones para funcionar. Debemos tomar en cuenta que el algoritmo AC se utilizó para comparar una sola palabra clave en el patrón, cuando la verdadera ventaja de este algoritmo se evidencia en los casos donde debemos comparar muchas palabras claves en una misma pasada. Por lo tanto, para analizar una sola palabra resulta ser una mala elección. El hecho de que el algoritmo RK no lograra superar al algoritmo BM podemos adjudicarlo al hecho de que los patrones, al ser relativamente pequeños, demandaron muchas comparaciones y no se pudo sacar provecho del *hashing*. Es interesante de todas formas cómo el algoritmo RK logró superar a los tres algoritmos en tiempo de búsqueda por ocurrencia de un patrón en la parte I. Esto se debe a que al haber una sola ocurrencia del patrón casi todas las comparaciones de valores hash fueron falsas y no se prosiguió a comparar carácter a carácter la secuencia. Esto no se logra en el BM ya que al utilizar un enfoque clásico el BM compara carácter a carácter y a pesar de que pueda desplazar el patrón al encontrar caracteres erróneos esto no se compara a la eficiencia de desplazar el texto con una función hash en el algoritmo RK.

A) Tiempo de búsqueda para patrones de largo creciente

En la figura 7 podemos observar un salto en el tiempo de búsqueda por patrón en los algoritmos. Esto es esperable ya que hay una sola ocurrencia del patrón “ccataac” en el genoma, por lo que en los tres casos nos acercamos mucho al peor caso posible de complejidad computacional. En el algoritmo AC podemos justificar este salto con el hecho de que el comienzo del patrón “ccat” es más común a lo largo del genoma, por lo que podemos adjudicar el incremento en el tiempo con una mayor cantidad de llamadas a las funciones *goto (g), fail (f) y output (o)* del algoritmo. Siguiendo esta misma lógica podemos pensar lo sucedido en el algoritmo RK y BM ya que ambos presentaron un incremento significativo. Es de esperar que el incremento RK sea mayor que el de BM, pues en RK se requieren más operaciones antes de cada comparación fallida dado al cálculo de los valores hash de subcadenas del texto. El incremento en BM sigue una tendencia creciente sin un cambio significativo, tomando en consideración que el patrón “ataaaa” tuvo 14 ocurrencias mientras que “ccataac” solo una. Con estos resultados podemos observar como la teoría se corresponde con la práctica, dado a que las complejidades computacionales para los peores casos de RK y AC predicen un tiempo requerido en ejecución similar. Es llamativo que el algoritmo BM no haya tenido un tiempo tan alto como RK y AC dado que, en el caso que evaluamos, la complejidad de BM es similar a la de RK. Es por esto por lo que adjudicamos el caso que probamos con uno de los escenarios particulares que mencionamos en la sección III, donde la complejidad de BM puede llegar a ser de O(m+n). Esta es la única explicación factible, pero de todas formas no explica del todo la diferencia de más de 4000 microsegundos respecto a RK y AC.

B) Tiempo de búsqueda para patrones con ocurrencias creciente

En la parte II del experimento fue evidente que todos los algoritmos incrementaron su tiempo de ejecución a medida que hay más ocurrencias de una misma cadena, pero podemos justificar esta diferencia en los tiempos con la cantidad de operaciones requeridas por cada algoritmo a medida que incrementan las ocurrencias. Mientras que en BM se comparan las palabras y hay más desplazamientos, como es de esperar, el algoritmo AC realiza más llamadas a las funciones g, f y o con lo que podemos explicar un mayor incremento con este algoritmo. Por el otro lado el algoritmo RK se encuentra en una situación similar a la de BM, a pesar de que en teoría hubiéramos esperado que RK superara a BM, dado a que el uso de las funciones hash pareciera acelerar el proceso de búsqueda, pero parece ser que con patrones tan pequeños de 4 caracteres la ventaja de las funciones hash no poseen un efecto significativo en el tiempo de búsqueda.

V. CONCLUSIÓN

Volviendo a nuestra temática de búsqueda de patrones en un genoma, las pruebas realizadas nos brindaron una nueva herramienta para utilizar en la evaluación de algoritmos antes de elegir qué algoritmo usar.
El algoritmo Boyer Moore demostró ser el más efectivo para cadenas de diferente largo y para cadenas con mayor número de ocurrencias. Esto es razonable dado a la simplicidad del enfoque de este algoritmo contra por ejemplo el Aho Corasick,
donde para buscar cadenas relativamente cortas no resulta ser eficiente un enfoque de suffix automata. El algoritmo AC demostró ser el peor de los tres algoritmos, ya que no solo su tiempo de búsqueda fue mayor, sino que el incremento en el tiempo de búsqueda a medida que se incrementaba el largo de cadena o incrementaban sus apariciones era mayor que el incremento entre cadenas de los otros dos algoritmos. Estos dos resultados son comprensibles dado a que la maquina de estado finita diseñada por el algoritmo para la búsqueda es mayor a medida que crece el largo de la cadena, demandando un mayor uso de funciones de *goto, fail* y *output* a medida que se van encontrando cadenas similares. En el otro caso, cuando se incrementan el numero de coincidencias también es esperable que se incremente el tiempo de búsqueda dado a que caemos en la misma situación descrita anteriormente con el incremento del largo de cadena, pero además se utiliza con mayor frecuencia la función de output. Debemos recalcar que el tiempo medido en el algoritmo de AC excluyó el tiempo de preparación, es decir el armado de las máquinas de estado. Finalmente, le algoritmo Rabin Karp fue una sorpresa dado a que no logró superar al BM, que parece ser razonable dado a que los sub conjuntos que tomaba el algoritmo RK para calcular los valores hash eran de a 4 caracteres, con lo cual la ventaja que se podría haber obtenido de comparar valores hash y “saltar” aquellos valores que no coincidan se pierde dado a que no son muchas las comparaciones que se ahorran, incluso en el mejor caso posible.
Finalmente, la respuesta a problemática de la elección de un algoritmo para la búsqueda de coincidencias de patrones en genomas es que dependiendo de lo que se quiera comparar debemos elegir o el BM o el AC para realizar la búsqueda en el menor tiempo posible. Por ejemplo, en el caso de la búsqueda de patrones singulares en un texto, es decir tan solo una cadena de caracteres que represente una porción del ADN, el algoritmo BM es la mejor opción dado a que mostró ser el más rápido y mostro un incremento lineal a medida que aumentaban las ocurrencias o el

largo del patrón a buscar. En definitiva, el algoritmo Boyer Moore es la mejor opción de las tres a la hora de manejar patrones largos, ya que demanda menos tiempo de ejecución encontrar el patrón a medida que crece el largo del mismo.

Pero por el otro lado en el caso de querer buscar muchos patrones diferentes en un mismo genoma, la mejor opción para hacerlo es el AC dado su escalabilidad. En este algoritmo para agregar una palabra es tan solo agregar una palabra clave en el conjunto K de palabras a buscar, donde se armará para cada una máquina de estado finita.

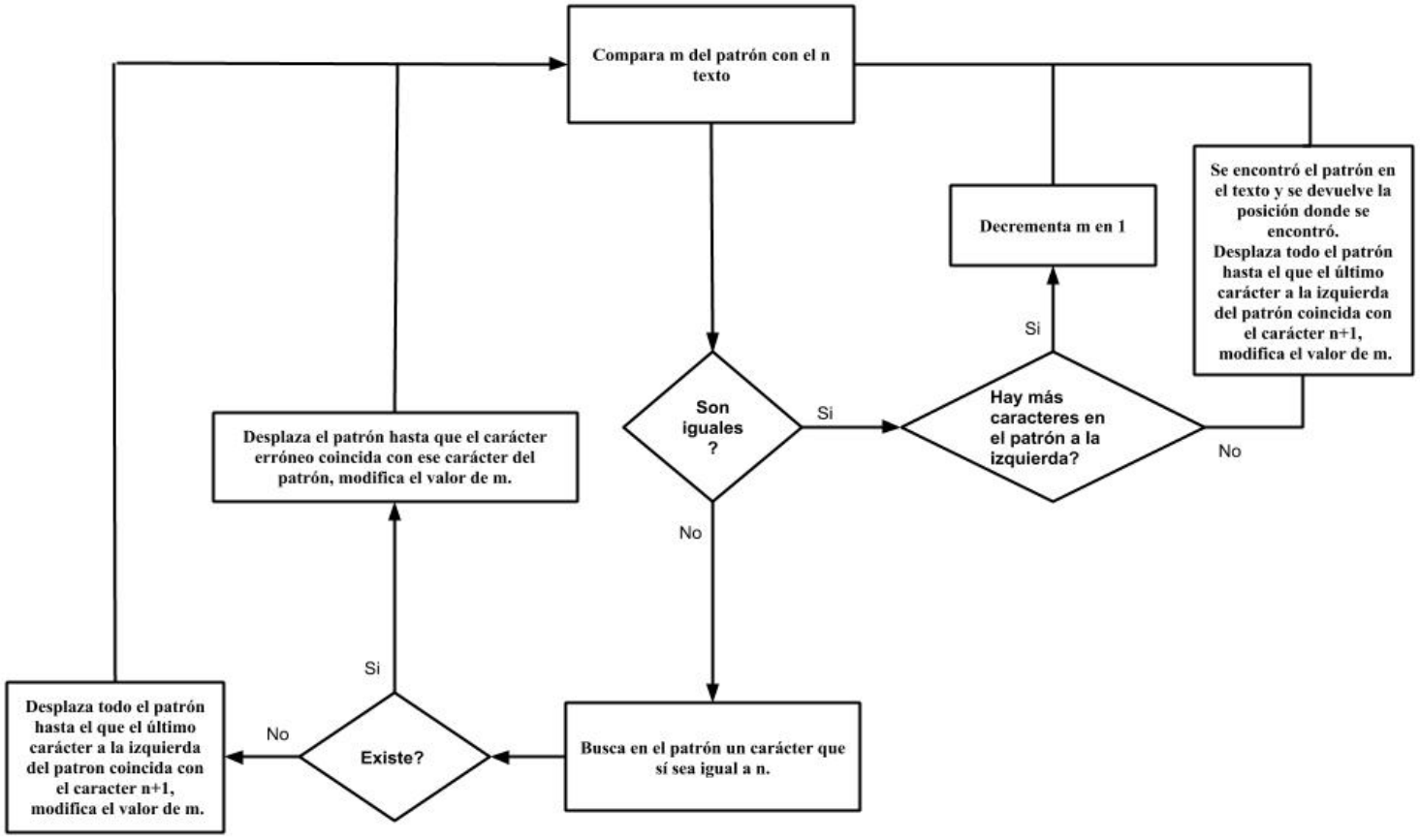
El algoritmo Rabin Karp no se destacó como una buena opción para la detección de pequeños patrones de ADN en genomas, no obstante, este algoritmo es usado en varios programas de verificación de plagio donde las cadenas son más largas y el *hashing* realmente puede ser aprovechado. Por lo tanto, podemos concluir que en caso de querer comparar un genoma completo con otro genoma completo entonces en esa situación el RK podría ser una de las mejores opciones.

VII. Mejoras y Futuras Investigaciones

Algunas modificaciones podrían haberse llevado a cabo para lograr un análisis más completo, como por ejemplo no limitarse al genoma de una de las secuenciaciones del SARS-CoV y evaluar los algoritmos en diferentes secuencias. De todas formas, no creemos que esto no hubiera impactado significativamente en nuestras conclusiones. Otra mejora podría ser comparar con un cuarto algoritmo que posea un enfoque de paralelismo de bits ya que fue el cuarto tipo de algoritmo de coincidencia de datos que no consideramos. Además, sabiendo los resultados de la investigación podrían evaluarse diferentes algoritmos con enfoque clásico como lo es el algoritmo Boyer Moore, y así poder decidir mejor cual algoritmo es más eficiente para la búsqueda de relativamente pequeños patrones en un genoma. Por el otro lado, si se quisieran comparar muchos patrones se podría realizar otra investigación que evalúe el rendimiento de diferentes algoritmos con enfoque de suffix automata.

APÉNDICE

FIGURA 1.



APENDICE 1. LINK AL REPOSITORIO DE GITHUB CON EL PROYECTO

https://github.com/inequihi/EDA-Monografia

APÉNDICE 2. RESULTADOS BRUTOS PARTE I

TABLA III
RESULTADOS PARA RABIN-KARP

Patrón	Largo (Caracteres)	Ocurrencias	Tiempo de ejecución (ms)
cg	2	439	795480
agt	3	507	263356
gcgt	4	37	9272
agtgt	5	51	7399
ataaaa	6	14	7663
ccataac	7	1	7157

TABLA IV
RESULTADOS PARA AHO-CORASIK

Patrón	Largo (Caracteres)	Ocurrencias	Tiempo de ejecución (ms)
cg	2	439	1033786
agt	3	507	1122128
gcgt	4	37	96451
agtgt	5	51	107211
ataaaa	6	14	21643
ccataac	7	1	6178

TABLA V
RESULTADOS PARA BOYER-MOORE

Patrón	Largo (Caracteres)	Ocurrencias	Tiempo de ejecución (ms)
cg	2	439	307457
agt	3	507	214692
gcgt	4	37	14210
agtgt	5	51	5746
ataaaa	6	14	8364
ccataac	7	1	2136

APPÉNDICE 3. RESULTADOS BRUTOS PARTE II

TABLA VI
RESULTADOS PARA AHO-CORASIK

Patrón	Ocurrencias	Tiempo de ejecución (ms)
agtg	146	655302
tgag	90	179763
atga	187	385769
gtta	179	409066
aaaa	281	604048
tttt	299	1636094
gggg	15	58549

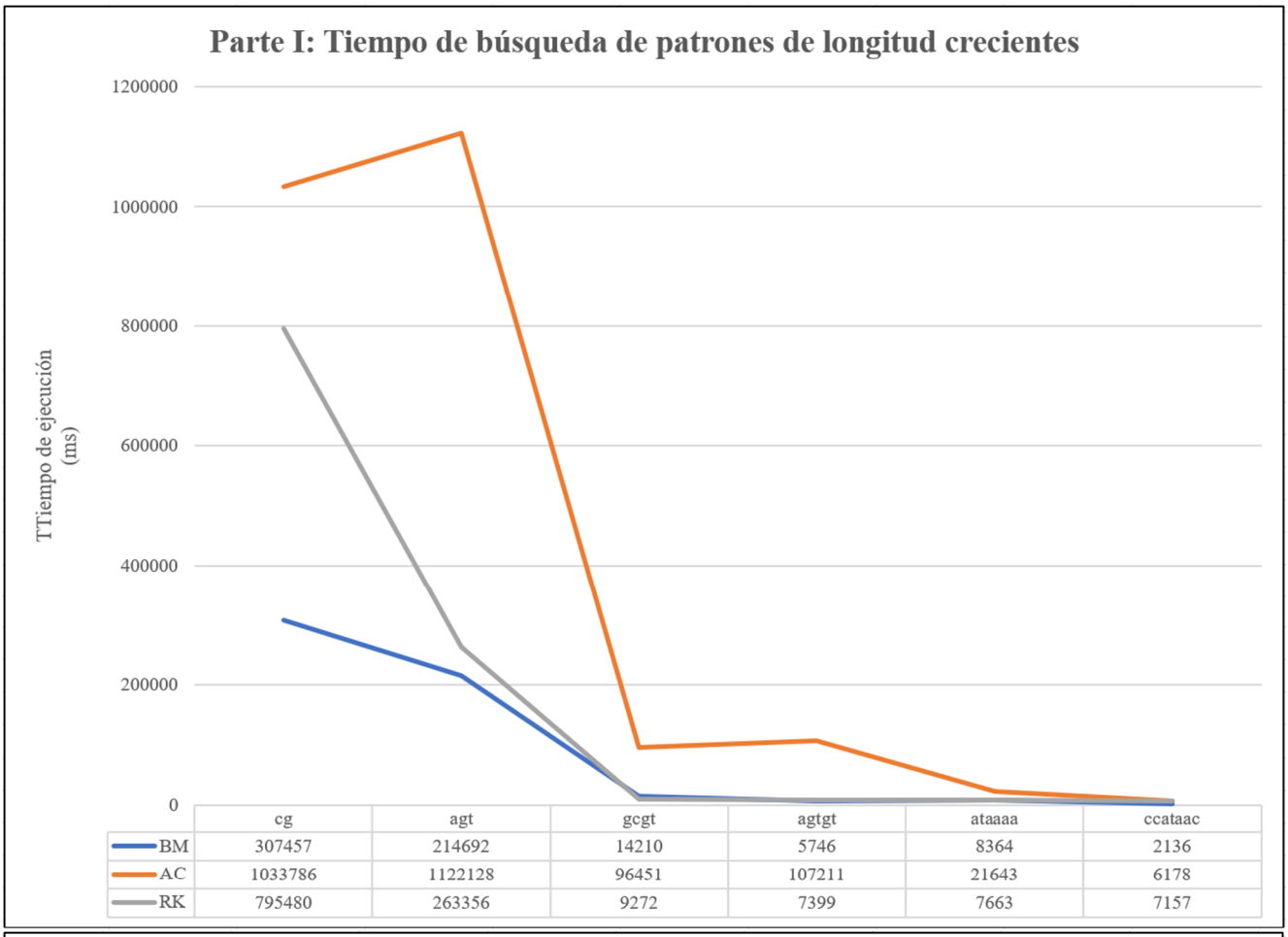
TABLA VII
RESULTADOS PARA BOYER-MOORE

Patrón	Ocurrencias	Tiempo de ejecución (ms)
agtg	146	2592
tgag	90	11406
atga	187	71590
gtta	179	37123
aaaa	281	115452
tttt	299	199552
gggg	15	13195

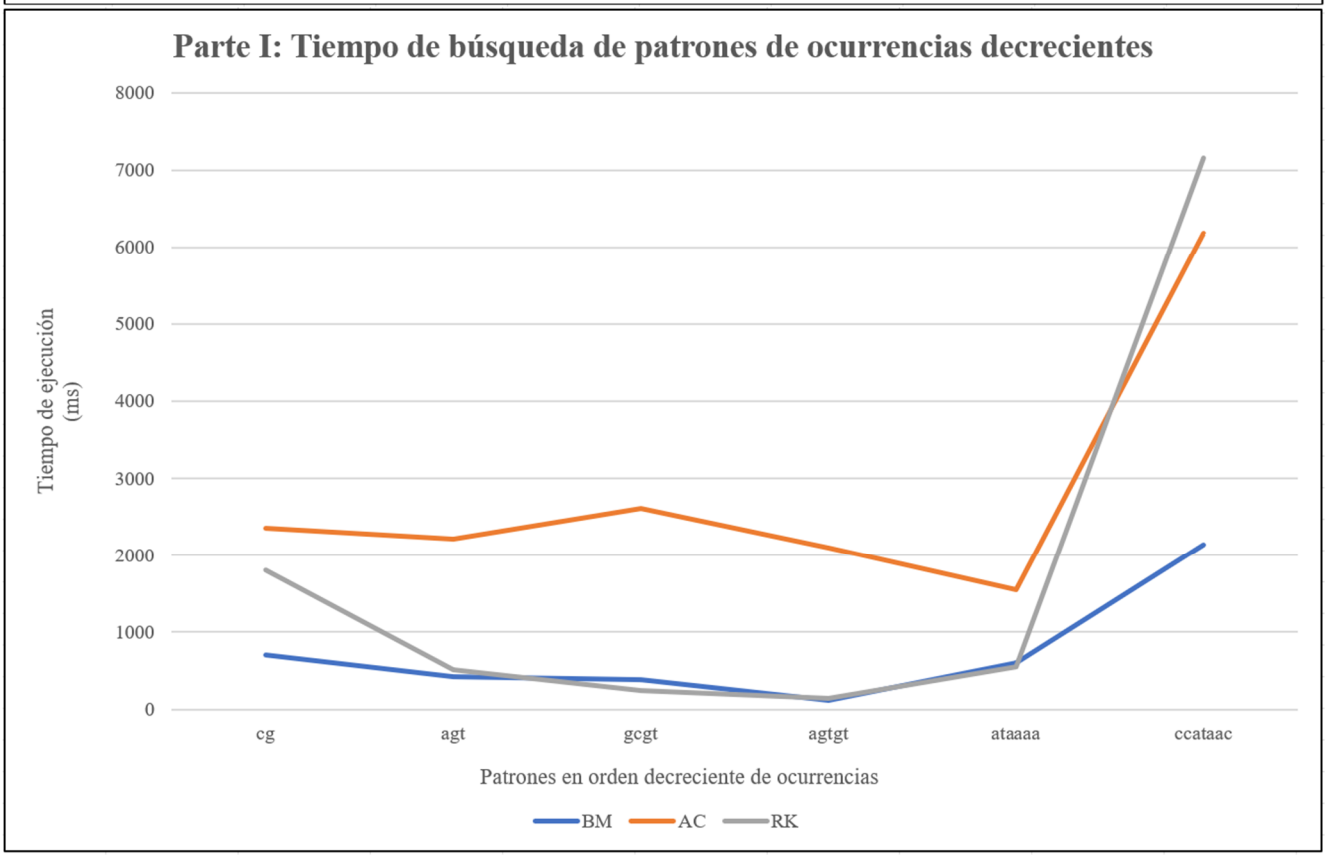
TABLA VIII
RESULTADOS PARA RABIN-KARP

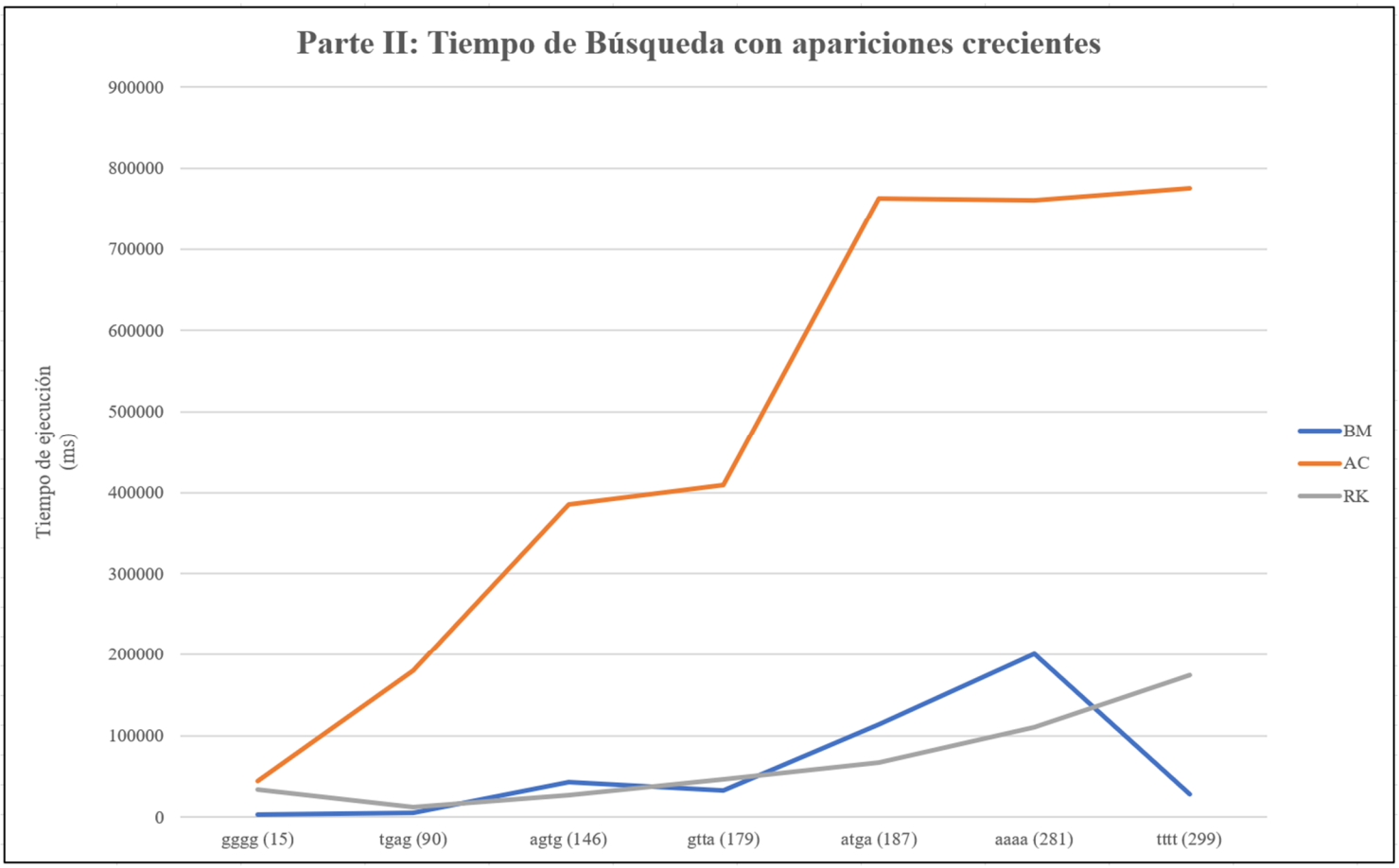
Patrón	Ocurrencias	Tiempo de ejecución (ms)
agtg	146	27052
tgag	90	12691
atga	187	67366
gtta	179	46504
aaaa	281	426114
tttt	299	286942
gggg	15	34404

APPÉNDICE 4. GRÁFICOS



	cg	agt	gcgt	agtgt	ataaaa	ccataac
BM	307457	214692	14210	5746	8364	2136
AC	1033786	1122128	96451	107211	21643	6178
RK	795480	263356	9272	7399	7663	7157





VIII. Referencias

Aho, A., & Corasick, M. (s.f.). *Efficient String Matching: An Aid to Bibliographic Search.* Murray Hill, N.J.: Association for Computing Machinery, Inc.

Ashish Prosad, G., & Rabi Narayan, B. (2014). *Novel Pattern Matching Algorithm in Genoma Sequence Analysis.* Salt Lake, Kolata, India.

Bekakos, M. P., Bowring, N., Coddington, P. D., & Cao, J. (2008). *Supercomputing Research Advances.* New York: Nova Science Publishers, Inc.

Bhojasia, M. (s.f.). *Sanfoundry.* Obtenido de Cpp Program Implement Aho Corasick Algorithm for String Matching: https://www.sanfoundry.com/cpp-program-implement-aho-corasick-algorithm-string-matching/

Bhojasia, M. (s.f.). *Sanfoundry.* Obtenido de Cpp Program Implement Boyer Moore Algorithm for String Matching: https://www.sanfoundry.com/cpp-program-implement-rabinkarp-algorithm/

Bhojasia, M. (s.f.). *Sanfoundry.* Obtenido de Cpp Program Implement Rabin Karp Algorithm: https://www.sanfoundry.com/cpp-program-implement-rabinkarp-algorithm/

Collins, F. S. (s.f.). *Genome*. Obtenido de Genetics Glossary: https://www.genome.gov/es/genetics-glossary/Genoma

Cormen, T. H. (2001). *Introduction to Algorithms.* Cambridge, Massachusetts: McGraw-Hill Book Company.

Genome.gov. (27 de Septiembre de 2019). Obtenido de Secuenciación del ADN: https://www.genome.gov/es/about-genomics/fact-sheets/Secuenciacion-del-ADN

Gimel'farb, G. (s.f.). String Martching Algorithms. *COMPSCI 369 Computational Science.* Auckland.

Heung, C. (s.f.). *Text MAtching Boyer Moore.* Atlanta, Georgia: Emory University Math/CS Department.

Knuth, D., Morris, J., & Pratt, V. (1974). *Fast Pattern matching in strings.* California: Stanford University.

Storer, J. (2001). *Introduction to Data Structures and Algorithms.* Waltham,MA: Sprincer Science+Business Media, LCC.