Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
Santiago

Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

Búsqueda en Texto

CC4102-1 Diseño y Análisis de Algoritmos

Integrantes: Edward Moreno
Felipe Rubilar
Profesor: Gonzalo Navarro
Fecha: 12 de octubre de 2015

Índice

1 Introducción

Con el fin de analizar el rendimiento de distintos algoritmos de búsqueda en texto, se realizan mediciones de tiempo y número de comparaciones realizadas sobre múltiples experimentos, aplicando conceptos y técnicas vistas en clases tanto en la construcción de dichos experimentos como en el análisis de los resultados.

Los algoritmos a analizar son Fuerza Bruta, Knuth-Morris-Pratt (KMP) y Boyer-Moore-Horspool (BMH), y su rendimiento será evaluado en una serie de experimentos (búsquedas) en textos de tamaño 1MB ($\sim$ 1 millón de caracteres) y patrones de largo variable en el rango $[2^2, 2^7]$.

Los textos varían tanto en el lenguaje/alfabeto que utilizan (Binario, ADN, Inglés) como en la forma en la que son generados (extractos de texto *real* o generado aleatoriamente a partir del alfabeto), de modo de analizar el comportamiento de los algoritmos en múltiples situaciones y evitar errores asociados a casos particulares.

2 Hipótesis

- Si las condiciones de ejecución para los tres algoritmos son iguales y no dependen de los textos ni el patrón a buscar, entonces los resultados tendrán forma similar en todas las situaciones, y las variaciones que se produzcan van a ser el resultado del desempeño específico de los algoritmos frente a las características del lenguaje/patrón utilizado.

- En cuanto al tamaño del alfabeto del texto, si la cantidad de caracteres diferentes es baja, entonces existen menos combinaciones posibles, lo que aumenta las posibilidades de encontrar substrings del texto con combinaciones similares (o iguales) a las del patrón buscado, lo que se traduce en una mayor cantidad de tiempo y comparaciones debido al aumento en el trabajo que el algoritmo debe realizar. Por el contrario, si existe gran cantidad de caracteres distintos las combinaciones posibles aumentan, lo que disminuye la probabilidad de encontrar patrones similares al buscado y permite descartar posibles ocurrencias en menor cantidad de tiempo y comparaciones.

- Si el tamaño del texto n es mucho mayor al tamaño del patrón buscado m, entonces el orden $O(n+m)$ de KMP tenderá a $O(n)$, lo que resultará en curvas de forma constante y de escasa variación frente al crecimiento del patrón, al menos en el rango analizado.

- A medida que el tamaño del alfabeto va aumentando, el orden $O\left(n\left(\frac{1}{m}+\frac{1}{2c}\right)\right)$ de BMH tenderá a ser $O(n/m)$, siendo c la cantidad de elementos del alfabeto utilizado, m y n ya nombrados.

3 Diseño e Implementación

3.1 Algoritmos de Búsqueda

Las tres implementaciones de algoritmos que se verán heredan de la clase *StringSearch* (que a su vez implementa de la interfaz *IStringSearch*) de la cual se explicarán sus métodos y atributos más importantes:

- Métodos:
  - **void searching(String pattern)**: Se encarga de la búsqueda en texto realizada por cada algoritmo, este método debe ser sobre-escrito obligatoriamente por los nuevos algoritmos de búsqueda.
  - **void search(String pattern)**: Setea el número de comparaciones a cero, ejecuta una búsqueda con ayuda del método *searching* y toma el tiempo que demora esta considerando lo realizado por la función *preComp*.
  - **void preComp(String pattern)**: Realiza cálculos que son necesarios para el funcionamiento de un algoritmo, por ejemplo, calculo de la función de fracaso de KMP. Este método está vacío para que los algoritmos de búsqueda lo sobre escriban en caso de necesitarlo.
- Atributos:
  - **String text**: Es el texto sobre el cual se harán las búsquedas.
  - **long time**: Encargada de guardar el tiempo que demoró el algoritmo.
  - **long comps**: Guarda la cantidad de comparaciones realizadas por el algoritmo.
  - **long matches**: Tiene la cantidad de veces que un patrón calzó completamente en un texto.

3.1.1 Fuerza Bruta

Descripción: Consiste en alinear el patrón a buscar con el texto e ir comparando de izquierda a derecha si los caracteres coinciden, si no lo hacen se desplaza el patrón una posición a la derecha.

Implementación: Básicamente son dos **for** que iteran en el patrón y en el texto como se muestra a continuación:

for $i = 0$ to $n - 1$ **do**
  $match = 0$
  for $j = 0$ to $m - 1$ **do**
    if $text[i] == pattern[i + j]$ **then**
      $match = match + 1$
    else

```
            break
        end if
        if Hay match completo then
            comps = comps + 1
        end if
    end for
end for
```

3.1.2 Knuth-Morris-Pratt (KMP)

Descripción: Este algoritmo alinea el texto con el patrón a encontrar, buscando de izquierda a derecha una ocurrencia de éste. Cuando el algoritmo se topa con un fallo, KMP hace uso de una *Tabla de Fallos* para saltar a la siguiente posición, a diferencia del algoritmo de fuerza bruta que siempre se desplaza un lugar a la derecha.

El objetivo de esta Tabla de Fallos es evitar que los caracteres de T sean examinados más de una vez, reutilizando lo que ya se ha visto en el texto para construir otra posible ocurrencia del patrón. Esto es posible gracias al proceso de construcción de la tabla, que busca en el mismo patrón ocurrencias de alguno de sus prefijos, con el fin de reutilizarlos como el inicio de un nuevo posible match.

Por ejemplo, si se tiene el texto T = "ABCABCABD" y el patrón P = "ABCABD", el algoritmo encontraría un fallo al comparar T[5] = C con P[5] = D, pero en lugar de descartar lo comparado y empezar a buscar nuevamente desde T[1], se reutiliza T[3] a T[5] ("ABC"), y se sigue comparando desde T[6] y P[4].

Implementación: En esta clase se agrega el atributo *fallos*, un arreglo de enteros que es utilizado como Tabla de Fallos, y se definen los siguientes métodos:

- **void searching(String pattern)**: Realiza una búsqueda del patrón dado siguiendo el algoritmo explicado en la descripción. Al encontrar un match, se incrementa el contador de ocurrencias (no nos interesa cuales son en nuestro análisis) y se continúa tal como si hubiera ocurrido un error (es decir, moviendo los punteros del texto y el patrón según la tabla de fallos), lo que permite reutilizar un sufijo del patrón como prefijo de una nueva posible ocurrencia, si el patrón lo permite.

- **void buildFailureFunction()**: Crea el arreglo que será utilizado como tabla de fallos. A medida que se recorre el patrón, el algoritmo va buscando ocurrencias de algún prefijo del mismo y guardando en el arreglo la longitud de la ocurrencia más larga hasta la posición anterior. Así, al acceder a la tabla se sabe la cantidad de caracteres que se pueden reutilizar, además de la posición en el patrón desde donde debe continuar la búsqueda.

3.1.3 Boyer-Moore-Horspool (BMH)

Descripción: Se alinea el patrón con el texto y se compara cada carácter de derecha a izquierda al contrario de los otros algoritmos, en caso de fallar se hace coincidir el último carácter del texto que está alineado con el patrón con el mismo presente en este. En caso de no existir tal, nos movemos el largo del patrón a la derecha, cabe destacar que este es el mejor caso de BMH.

Implementación: En esta clase agregamos el atributo *next* el cual es un hash que posee elementos del tipo $[Character\ key, Integer\ value]$. Los métodos creados y sobrescritos son aquí son los siguientes:

- **void searching(String pattern)**: Recorre un texto haciendo coincidir el patrón *pattern* con este tal cual se explico en la descripción. Cuando se encuentra un match completo (calzo todo el patrón en el texto) agrega uno a *matches*.
- **void getNextFunction()**: Obtiene la función que es utilizada para dar el avance del patrón en el texto en caso de no coincidir los caracteres, esta función se crea de la siguiente manera:

  Require: $next \leftarrow \emptyset$
  for $[c, i]$ in $pattern$, $i \in [0 \cdots m-2]$ **do** ▷ $c = pattern[i]$
  $next \leftarrow next \cup [c, m-(i+1)]$
  end for
  if $pattern[m-1] \notin next$ **then**
  $next \leftarrow next \cup [pattern[m-1], m]$
  end if

  Cuando este algoritmo termina ya tenemos lo necesario en el atributo *next*.
- **boolean next(char c)**: Comprueba si c se encuentra en el hash *next*, si esta retorna el valor respectivo, si no retorna m.

3.2 Patrones y Textos

Para los experimentos de búsqueda en texto se consideraron 3 alfabetos: el alfabeto binario {0, 1}, el alfabeto de cadenas de ADN {A, T, C, G} y el alfabeto del lenguaje natural (inglés) {a, ..., z, A, ..., Z}, incluyendo el delimitador espacio. Además, para cada alfabeto se tienen dos variantes: un texto *real* (cadenas reales de ADN y fragmentos de textos en inglés) y un texto *sintético* (cadenas de ADN y palabras generadas al azar a partir de sus respectivos alfabetos), con la excepción del alfabeto binario, que no tiene una contraparte real (pues cualquier cadena puede tener significado en alguna codificación).

Los patrones a ser buscados por los algoritmos descritos anteriormente son generados aleatoriamente en el caso binario y extraídos directamente del texto (real o

sintético, según el experimento) en el caso del ADN y el lenguaje natural.

En cuanto al tamaño, todos los textos (tanto reales como sintéticos) tienen un tamaño igual a 1MB ($\sim$ 1 millón de caracteres), mientras que los patrones pueden tener 4, 8, 16, 32, 64 o 128 caracteres de largo.

Por último con respecto a la generación de secuencias y obtención de substrings aleatóreos del texto se creo la clase *Azar* la cual cuenta con:

- Atributos:
  - **char[] alpha**: Arreglo de char del alfabeto que se utiliza para generar secuencias al azar.
  - **Random r**: Objeto tipo *Random* que es utilizado para generar números aleatorios.
  - **int m**: Tamaño del alfabeto *alpha*
- Constructor:
  - **Azar(String text)**: Se encarga de convertir *text* en un arreglo de *char* que es almacenado en *alpha*, obtener el largo de este y guardarlo en m e inicializar r como un nuevo objeto de la clase *Random*.
- Métodos:
  - **String generatePattern(int l)**: Genera un patrón de largo l seleccionando caracteres al azar desde *alpha*

Para generar patrones aleatorios se crearon las clases *AdnAzar*, *AlphaAzar* y *BinaryAzar* las cuales extienden de *Azar* y en su constructor configuran su alfabeto respectivo. Por ejemplo, tenemos para el constructor de la clase *AndAzar*:

```
public AdnAzar() {
    super("GCAT");
}
```

Por otro lado, para la obtención de substrings aleatorios del texto se creo la clase *SubStringAzar* que hereda de *Azar* la cual manda a llamar a *super* con su texto respectivo y sobre-escribe el método *generatePattern(int l)* para que con ayuda de números aleatorios poder ir a cierta parte del texto y extraer un patrón de largo *l*.

3.3 Experimentos

Con el fin de analizar el rendimiento de los algoritmos implementados, se realizaron una serie de experimentos para medir tanto el tiempo que tardan en terminar la

búsqueda como la cantidad de comparaciones de caracteres que realizan en el proceso.

Estos experimentos cubren todas las combinaciones posibles de algoritmo utilizado, tipo de texto y largo del patrón buscado, y son ejecutados n veces de modo de obtener múltiples valores para cada caso y así realizar un análisis estadístico de los resultados.

Cabe mencionar que las mediciones de tiempo y comparaciones se realizaron de forma separada, realizando primero n experimentos midiendo el tiempo y luego otros n experimentos contando el número de comparaciones realizadas. Esto con el fin de evitar las posibles alteraciones en la medición del tiempo que se podrían provocar al incluir las operaciones de conteo de comparaciones dentro del tiempo de ejecución del algoritmo, por muy mínimas que sean.

En concreto para realizar los experimentos se creo la clase *Experiment* la cual cuenta con lo siguiente:

- Atributos:
  - **StringSearch stringSearch**: Encargado de almacenar el algoritmo de búsqueda que se utilizará.
  - **Azar azar**: Almacena la forma de azar que se usará, ya sea subStrings del texto o generación de patrones.
  - **long time**: Acumula el tiempo que toma realizar un experimento. Solo considera el tiempo que tarda la realización del algoritmo *stringSearch* que se utiliza.
  - **long comps**: Acumula la cantidad de comparaciones que realiza un experimento durante la ejecución de una búsqueda en texto.
  - **long nExp**: Es el contador de experimentos que se han realizado.
  - **StringBuilder log**: Es el log de los experimentos realizados. Almacena pares de (tiempo, comparaciones) para cada experimento realizado, es decir, para cada ejecución del método *doOneExperiment*.
- Constructor:
  - **Experiment(StringSearch stringSearch, Azar azar)**: Inicializa todas las variables necesarias para poder realizar un experimento correcto.
- Métodos:
  - **void writeLog(long a, long b)**: Agrega a *log* el nuevo par (**a**, **b**).

- **void doOneExperiment(int sizePattern)**: Lleva a cabo una ejecución de búsqueda por *stringSearch* con un patrón de largo **sizePattern** y agrega los resultados de esto a *log*, además cambia el valor de los atributos *time*, *comps* y *nExp*.
- **void doExperiments(int nExp, int sizePattern)**: Realiza una cierta cantidad de experimentos con el método *doBasura* posterior a lo cual setea los atributos *time*, *comps* y *nExp* a cero para finalmente realizar **nExp** experimentos con el patrón de largo **sizePattern** con ayuda de *doOneExperiment*.
- **void doBasura(int nExp, int sizePattern)**: Realiza **nExp** experimentos con un patron de largo **sizePattern** para disminuir el error en mediciones producto a la carga recursos, de forma tal de poder realizar los experimentos efectivos en *warm state*.
- **long getMeanTime()**: Retorna el promedio de comparaciones dividiendo *time* en *nExp*.
- **long getMeanComps()**: Retorna el promedio de comparaciones dividiendo *comps* en *nExp*.
- **String getLog()**: Retorna el *log* correspondiente al experimento en forma de *String*.

Ya con esta clase implementada se creo el *main* que realiza todos los experimentos pedidos, este se encuentra en la clase *Main* del cual se procederá a describir sus métodos:

- **static void main(String[] args)**: Llama al método *doAllExperiments* dándole una ruta para guardar los archivos resultantes. Por defecto la cantidad de experimentos que se realizan está en 7000 pero se puede cambiar ya sea en el mismo *main* o al ejecutar desde consola agregando un argumento de la siguiente manera:

  $$javac\ Main\ nExp$$

  donde *nExp* debe ser un numero entero.

- **static void doAllExperiments(String path)**: Realiza todos los experimentos por tipo de alfabeto siguiendo el siguiente patrón concatenando **path** en cada llamada:

  procedure doAllExperiments(path)
  Inicializar objetos tipo Azar
  Obtener y generar textos
  for $i = 2^2$ to 2^7 **do**
  Llamar a *doStringSearch* con Alfabeto Binario
  Llamar a *doStringSearch* con ADN real
  Llamar a *doStringSearch* con ADN Sintético
  Llamar a *doStringSearch* con Alfabeto Real

Llamar a *doStringSearch* con Alfabeto Sintético
end for
end procedure

- **static void doStringSearches(String text, String preName, int sizePattern)**: Para cierto alfabeto realiza los experimentos para cada tipo de algoritmo. Básicamente se sigue el siguiente patrón concatenando **preName** en cada llamada:

  procedure doStringSearches(text, preName, sizePattern)
  Llamar a *doAzarExperiment* con Fuerza Bruta
  Llamar a *doAzarExperiment* con KMP
  Llamar a *doAzarExperiment* con BMH
  end procedure

- **static void doAzarExperiment(StringSearch stringSearch, Azar azar, String path, int sizePattern)**: Realiza la cantidad de experimentos dada en el *main* utilizando el algoritmo de busqueda en texto **stringSearch** y la forma de obtener patrones aleatorios **azar** y luego exporta el archivo resultante en formato CSV con *exportString*. Se sigue el siguiente patrón:

  procedure doAzarExperiment(stringSearch, azar, path, sizePattern)
  Realizar experimentos
  Exportar resultados con *exportString*
  end procedure

- **static void exportString(String str, String file)**: Guarda el *String* **str** en la ruta **file**.

4 Análisis y Resultados

A continuación se presentan los resultados obtenidos al realizar 7000 ejecuciones de los experimentos descritos anteriormente (ver detalle en anexo).

4.1 Análisis por Alfabeto

4.1.1 Alfabeto Binario

Para el caso del alfabeto Binario notemos que la cantidad de caracteres que posee es bastante reducida, solo son 2, lo cual hace que la probabilidad de repetición de estos sea bastante elevada con respecto a los otros dos lenguajes estudiados. Debido a este motivo aquí se presentan los mayores tiempos y comparaciones obtenidos durante los experimentos.

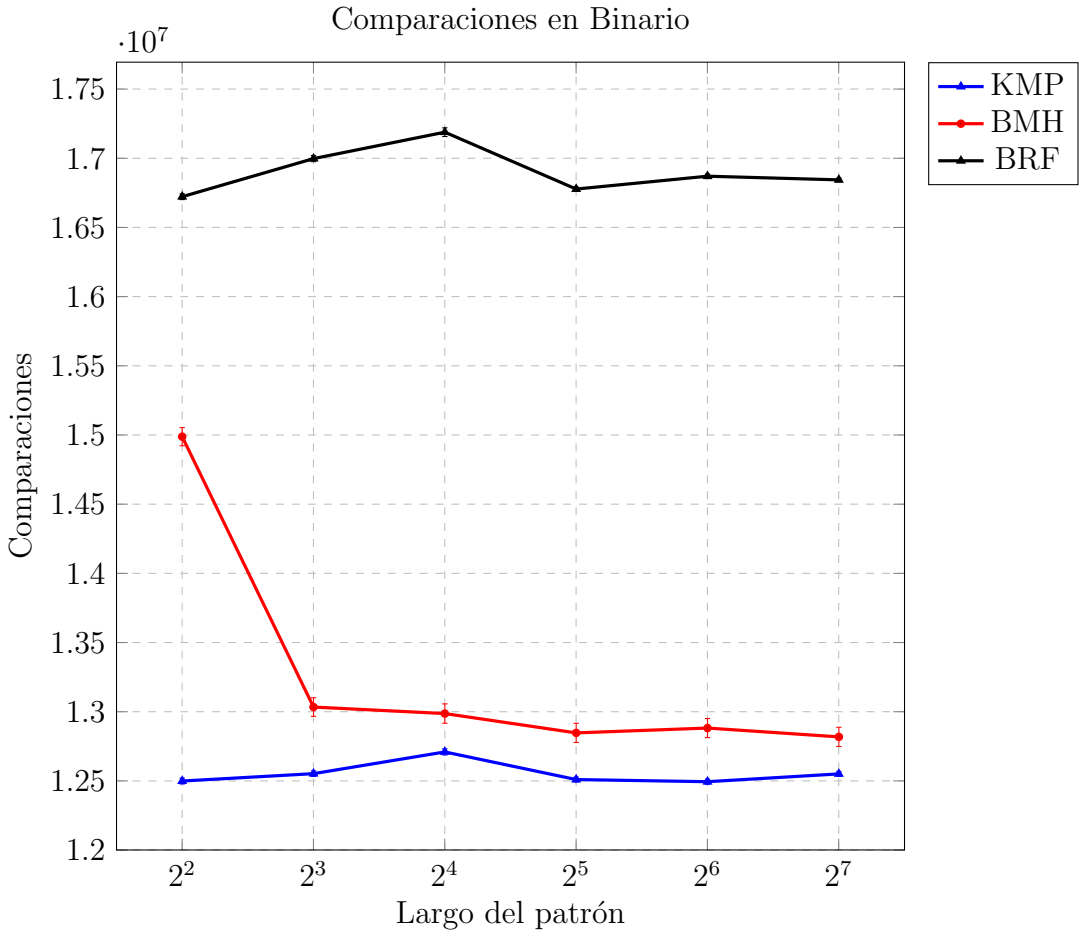

Figura 1: Gráfico de los distintos algoritmos de búsqueda aplicados sobre el alfabeto Binario. En el eje vertical podemos ver el promedio de comparaciones con su error mientras que en el horizontal los largos de patrón.

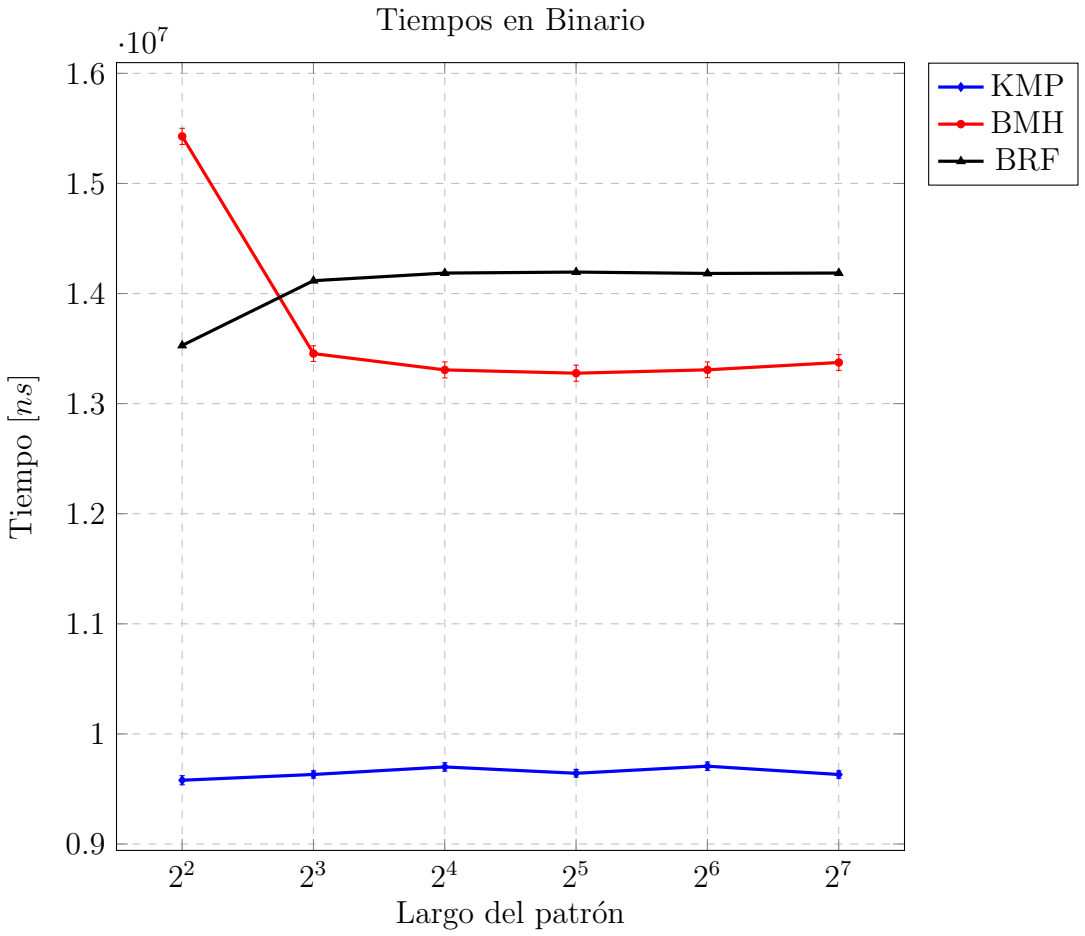

Figura 2: Gráfico que muestra los promedios de tiempo con su error tomados por los distintos algoritmo realizando búsquedas de patrones en textos con alfabeto Binario

4.1.2 ADN

En el caso del ADN se puede ver que tanto el tiempo como las comparaciones son bastante similares entre el caso real y sintético, pero este último tiende a ser mayor. Esto puede deberse a la forma en que se construye el ADN, con secuencias definidas que en ocasiones tienen posiciones determinadas en una cadena (como un marcador de inicio o fin), lo que hace poco probable que en una cadena real hayan segmentos que puedan ser "reutilizados" para formar otra posible ocurrencia, produciéndose así fallos más frecuentes y saltos más largos.

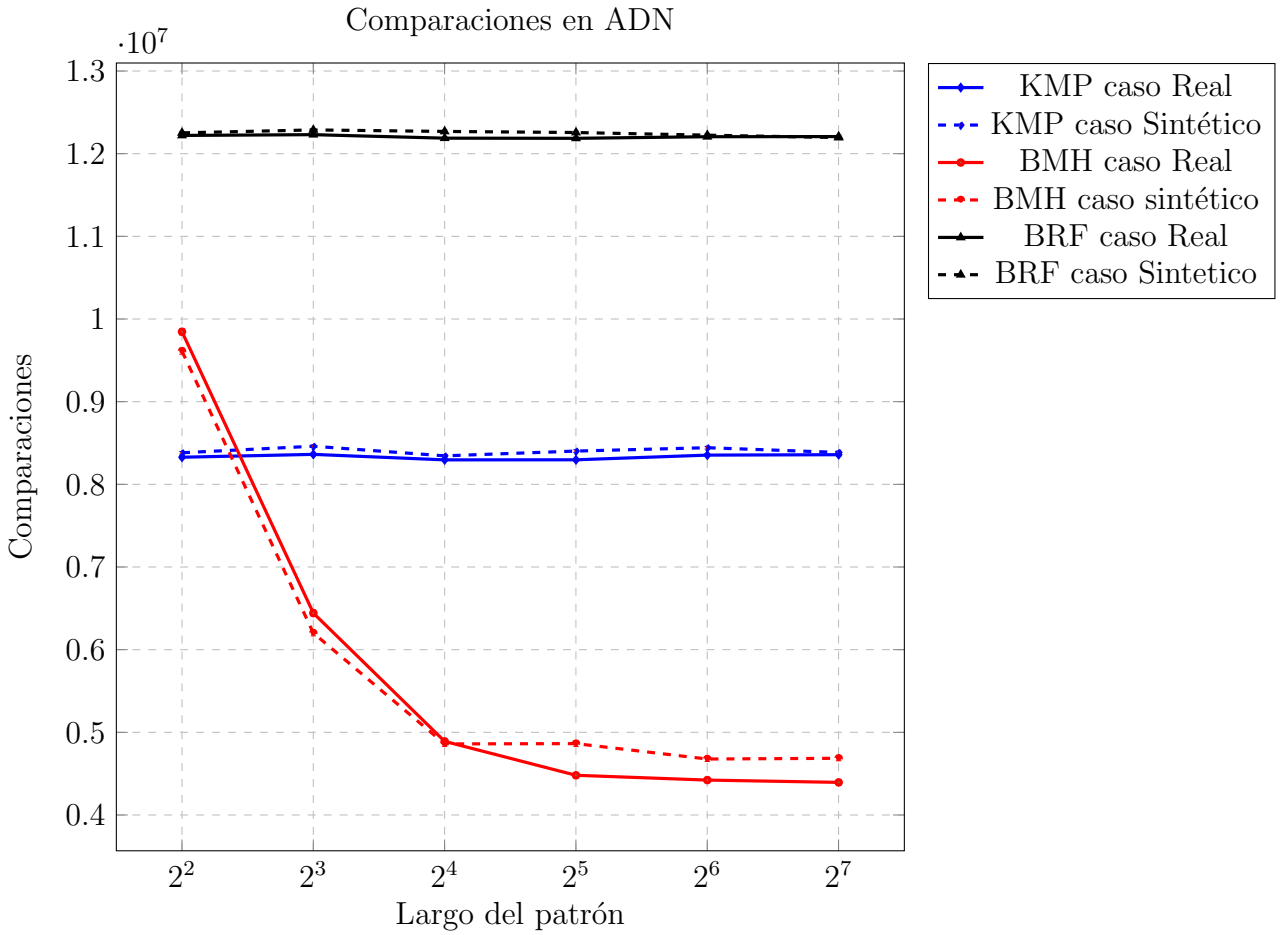

Figura 3: Gráfico que muestra los promedios de comparación con su error al aplicar los distintos algoritmos de búsqueda en ADN real y sintético.

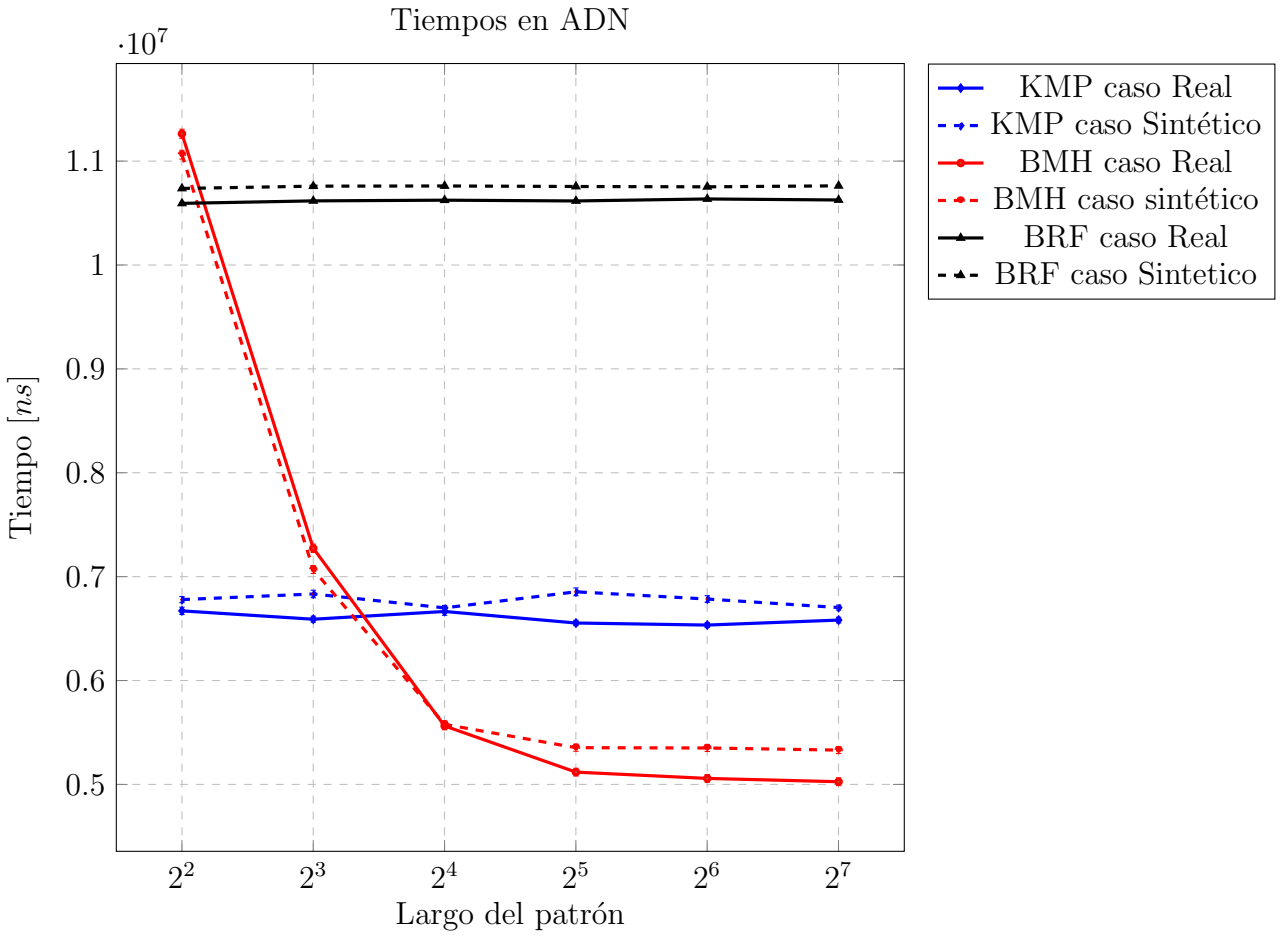

Figura 4: Gráfico que muestra el promedio y su error de las búsquedas en texto en el ADN real y sintético.

4.1.3 Lenguaje Natural

Notemos que para los 3 algoritmos se tiene que el número de comparaciones para el caso real es mayor que en el caso sintético.

Esto tiene sentido si consideramos que en el lenguaje real las palabras se escriben usando letras de forma particular, con vocales cada cierta frecuencia, sílabas comunes, etc., mientras que en el caso sintético la única regla es usar las letras del alfabeto. Esto significa que en el lenguaje sintético hay muchas menos repeticiones se substrings, lo que se traduce en encontrar fallos con mayor frecuencia, descartando posibles ocurrencias en menor tiempo/comparaciones y, en KMP y BMH, haciendo saltos más grandes.

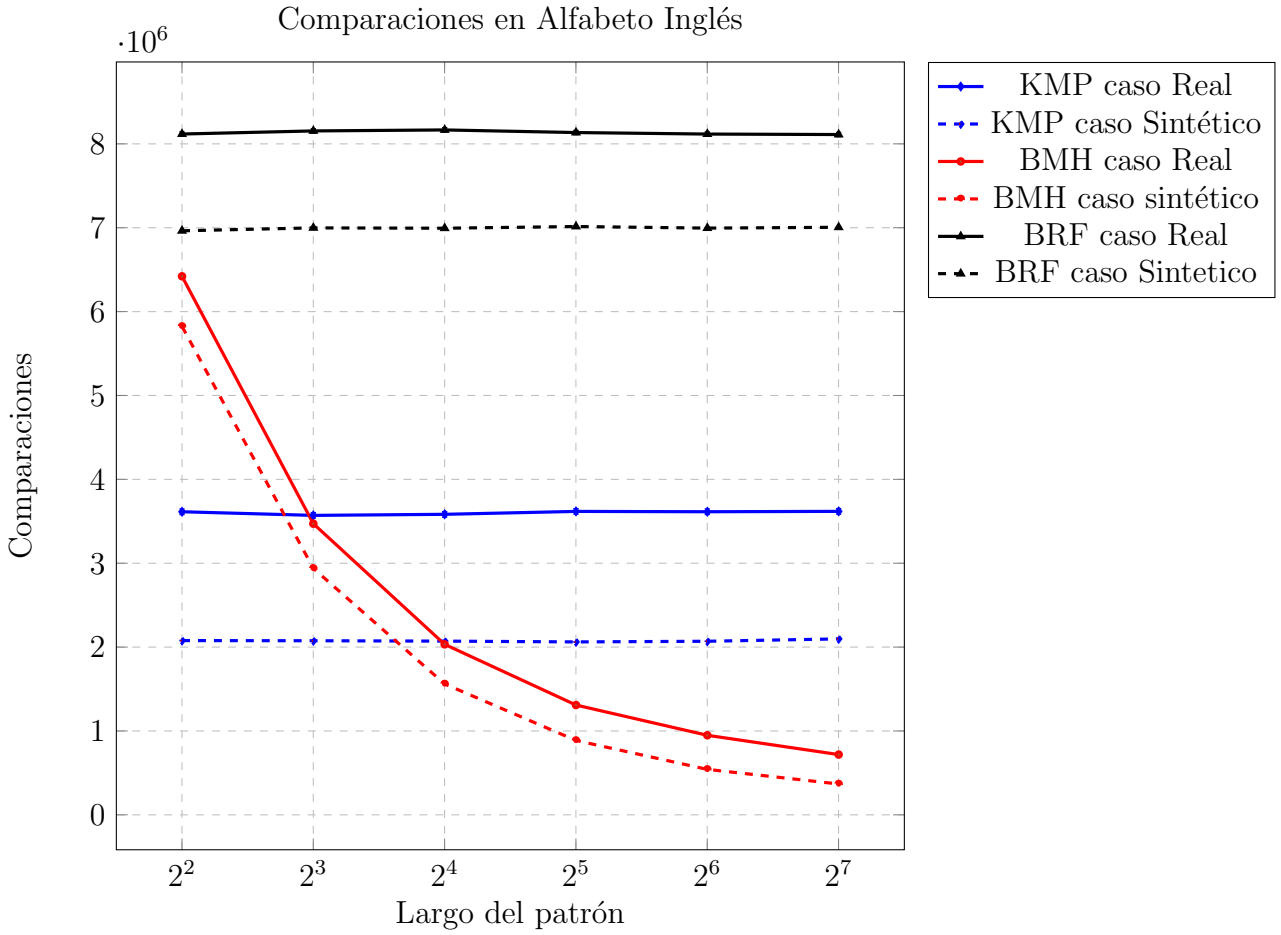

Figura 5: Gráfico del promedio y error de comparaciones realizadas en al alfabeto Inglés por los diferentes algoritmos utilizados.

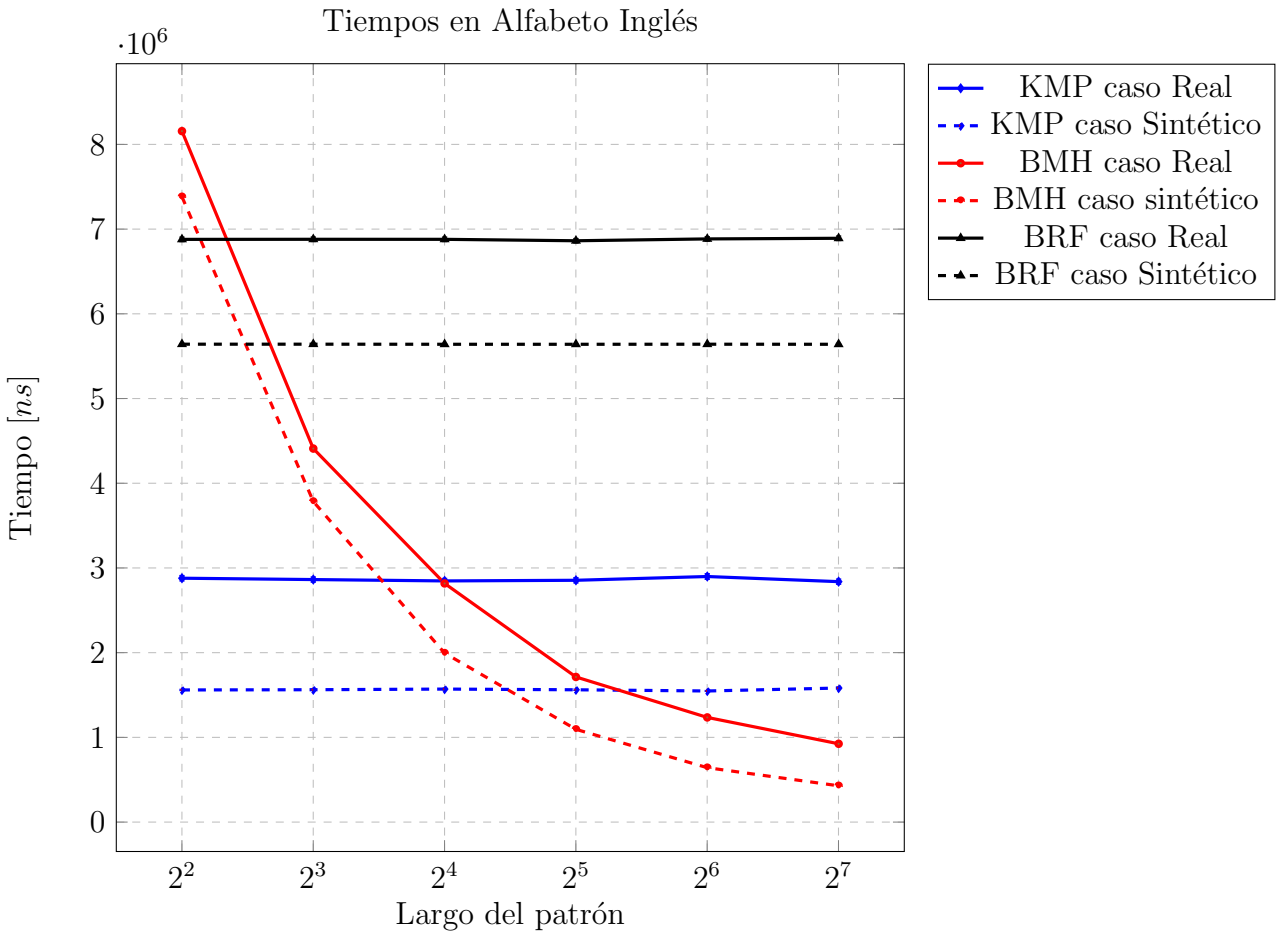

Figura 6: Gráfico que muestra los promedios y errores en estos que arrojan los distintos algoritmos al ser aplicados buscando patrones en el alfabeto Inglés.

4.2 Análisis por Algoritmo

4.2.1 Fuerza Bruta

Graficando los datos obtenidos por lenguaje para la cantidad promedio de comparaciones tenemos:

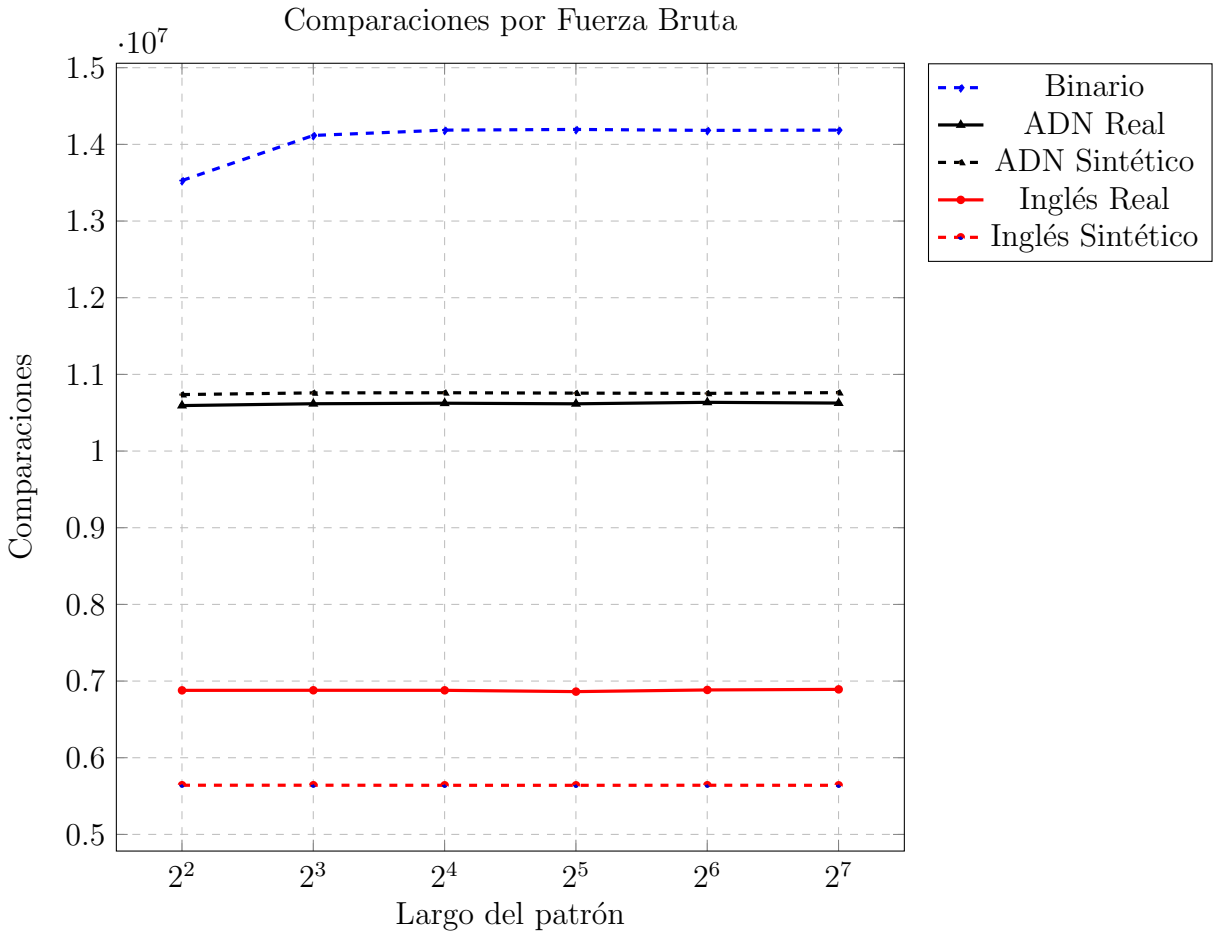

Figura 7: Resultados de comparaciones promedio al usar Fuerza Bruta para la búsqueda en texto en los alfabetos Binario, ADN (real y sintético) e Inglés (real y sintético) para distintos tamaños de patrones.

En cuanto a la cantidad de comparaciones promedio realizadas podemos notar que:

- A medida que el patrón aumenta de tamaño las comparaciones tienen tendencia lineal la cual no es muy visible debido a la escala presente en el gráfico.
- La diferencia sustancial radica en el lenguaje en si, se aprecia que para alfabetos con mayor variación se tiene menor cantidad de comparaciones debido menor probabilidad de tener letras similares en el texto/patrón lo cual hace que al

estar comparando haya algún tipo de discrepancia mas pronto que haga al patrón moverse a la derecha.

- En cuanto a la comparación entre el mismo lenguaje con sus variantes real/sintético podemos notar una discrepancia notable en el Inglés pues a pesar de tener la misma cantidad de caracteres la forma en la cual se distribuyen estos en un texto real no es uniforme tal como lo intenta hacer su contraparte sintética. Debido a lo cual en este ultimo existe mucho menor probabilidad de que se repitan letras lo cual resulta favorable por lo dicho en el punto anterior.

4.2.2 Knuth-Morris-Pratt (KMP)

Graficando los resultados obtenidos usando KMP en los distintos lenguajes tenemos que:

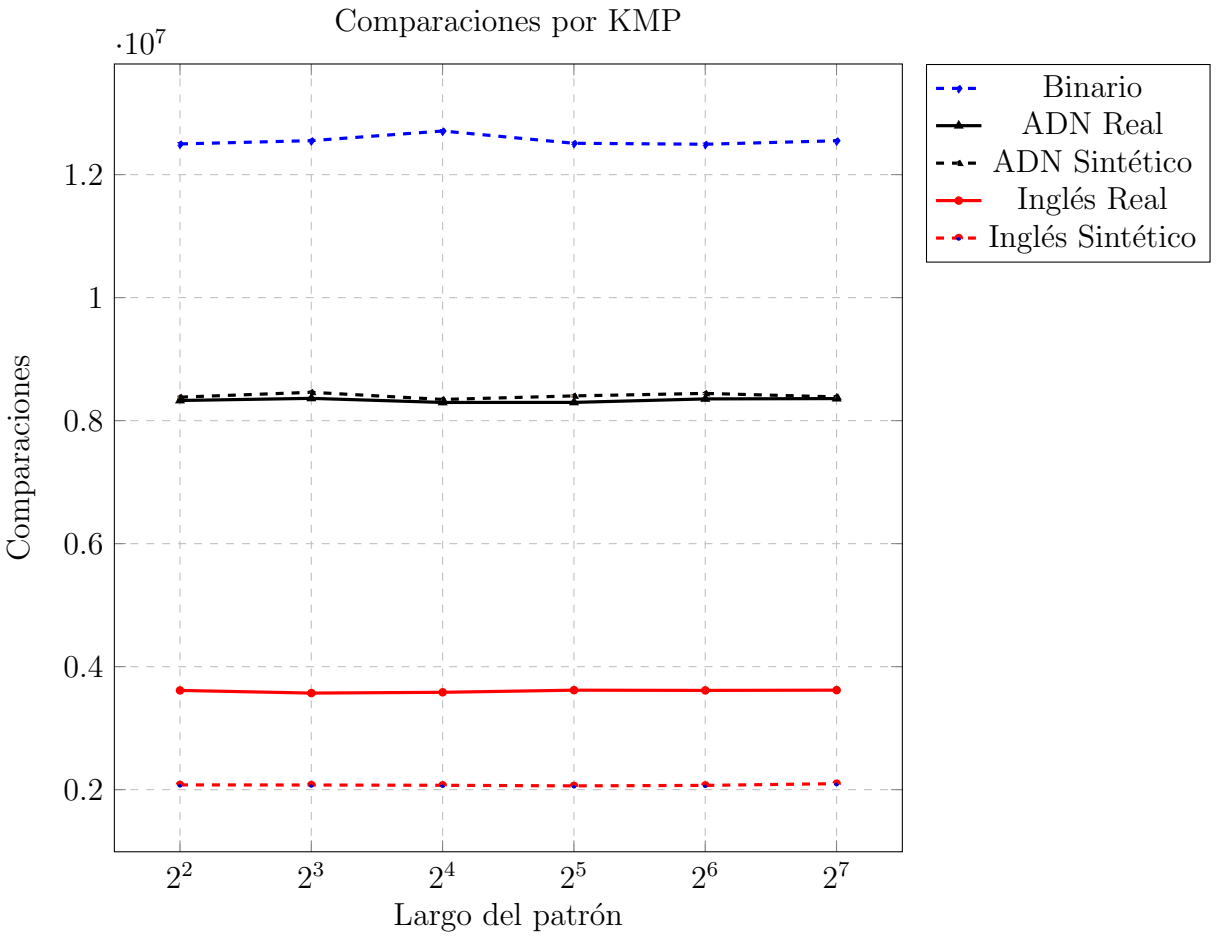

Figura 8: Resultados de las comparaciones promedio al usar KMP para buscar texto en los distintos lenguajes utilizados.

- Podemos notar que el algoritmo presenta un comportamiento similar en los tres lenguajes utilizados, con valores relativamente constantes dentro de sus órdenes de magnitud.
- No se aprecian grandes variaciones en función del largo del patrón, pero sí hay diferencias de gran tamaño entre los distintos lenguajes. Notar que esta diferencia es atribuible a la cantidad de caracteres de cada alfabeto y no a la forma en la que se construyen los textos, pues ambas variantes de ADN e Inglés tienen comportamientos similares a sus contrapartes, con diferencias dentro de lo esperado.

- También es posible apreciar la diferencia entre la versión real y sintética de ADN e Inglés, que como ya se explicó se debe a la frecuencia con la que se repiten ciertos substrings en las variantes reales. Esta diferencia es mucho menor en el caso del ADN, pues la cantidad de substrings "significativos" que posee es mucho menor a la que posee el Inglés.

4.2.3 Boyer-Moore-Horspool (BMH)

Graficando los tres lenguajes utilizados para distintos largos de patron tenemos lo siguiente:

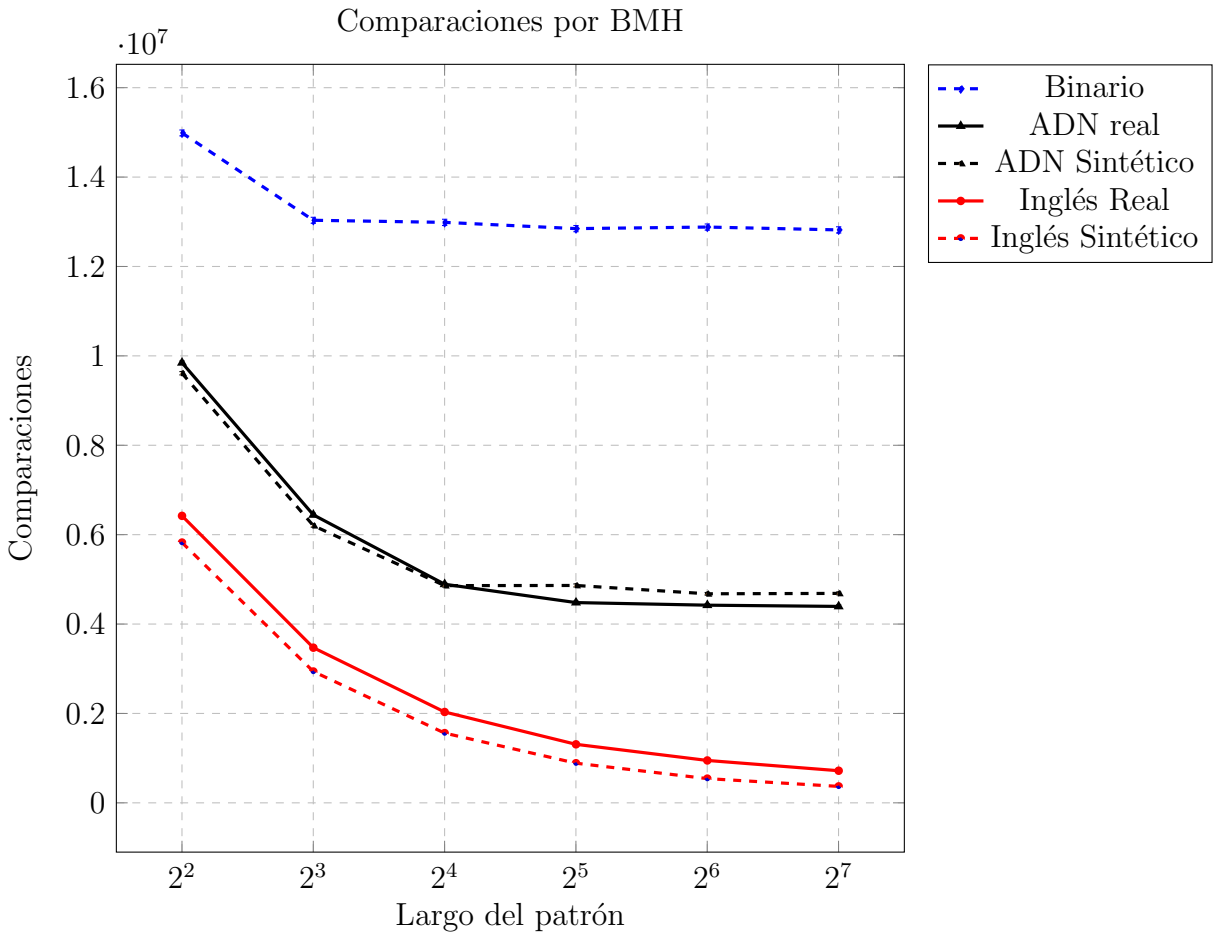

Figura 9: Resultados de comparaciones promedio al usar BMH para la búsqueda en texto en los alfabetos Binario, ADN (real y sintético) e Inglés (real y sintético) para distintos tamaños de patrones.

Del gráfico anterior podemos observar que en cuanto a la cantidad de comparaciones realizadas para cada lenguaje tenemos que:

- El alfabeto binario posee la mayor cantidad, lo cual es esperable puesto a que BMH se ve influenciado por la cantidad de caracteres del lenguaje, en este caso 2. Motivo por el cual ADN con 4 se sitúa después de éste seguido por el Inglés con 53. Esto se puede ver reflejado en el sumando $1/(2c)$ presente en el orden del caso promedio del algoritmo, mostrado en la Tabla 2, puesto que éste toma mayor importancia con valor bajo.

- Tenemos que el peor caso del algoritmo esta cuando no hay mucha variación en los caracteres del alfabeto, es decir, cuando tienden a haber muchas repeticiones tanto en el texto como en el patrón. Este caso se ve reflejado en el alfabeto binario donde se puede apreciar que BMH deja de tener su comportamiento "asintotico" como en el caso promedio y toma una forma lineal mucho mas rápido.

- Por cuanto a un mismo lenguaje se refiere se puede apreciar que a medida que va aumentando el largo del patrón la cantidad de comparaciones comienza a disminuir. Esto se ve mejor reflejado en el Inglés puesto a que como tiene una mayor cantidad de caracteres hay menos probabilidad de que estos se repitan en el patrón y por lo tanto los saltos que dará BMH serán mayores.

- Es posible notar que para cada lenguaje en particular la cantidad promedio de comparaciones realizada por el algoritmo parece estabilizarse conforme aumenta el largo del patrón. Esto Podemos apreciarlo en el orden promedio del algoritmo puesto a que cuando m se hace muy grande nos queda un orden del tipo $O(n/(2c))$ (mayor para lenguajes con menos caracteres) que resulta más importante que el $O(n/m)$.

- Por cuando a la variación entre lenguajes reales v/s sintéticos podemos notar que tenemos diferencias prácticamente constantes en cuanto al Inglés lo cual se puede deber al hecho de que al generar un patrón al azar todas las letras tienen la misma probabilidad de estar presentes es este, no así en su forma real puesto a que sabemos que hay letras mucho mas utilizadas que otras y por lo tanto debido a la mayor variación por la parte sintética tendremos que esta realizara menos comparaciones que su contraparte real.

5 Conclusiones

Comportamiento General: Los resultados obtenidos muestran que los algoritmos analizados siguen un comportamiento característico reconocible en todos los casos de prueba. Esto se aprecia fácilmente en los gráficos, en donde se puede reconocer cada algoritmo por su desempeño en comparación a los otros.

Tamaño del alfabeto: Tal como se esperaba, los resultados de mayor magnitud se obtuvieron en los alfabetos más pequeños. A menor cantidad de caracteres en el alfabeto, menos patrones posibles existen para un largo determinado, lo que implica que la probabilidad de encontrar prefijos grandes del patrón buscado aumenta mientras menos letras tenga el abecedario. Esto hace que los algoritmos "pierdan tiempo" analizando substrings que difieren del patrón en los caracteres finales, además de aumentar la probabilidad de poder reutilizar lo ya leído en una nueva ocurrencia, disminuyendo el tamaño de los saltos al fallar.

Tamaño del Texto y KMP: Los resultados de los experimentos para KMP no muestran una relación entre los datos obtenidos y el largo del patrón, sino que más bien se concentran en una curva de forma relativamente constante, con magnitudes que cambian para cada tipo de lenguaje. Esto cobra más sentido si recordamos que el orden $O(n+m)$ de KMP depende del largo del texto y del patrón, donde el texto es entre 10^4 y 10^6 veces más grande que el patrón.

Tamaño del Alfabeto y BMH : De los resultados obtenidos se pudo comprobar que efectivamente el tamaño del alfabeto influyó en gran medida en el desempeño de BMH lo cual es fácilmente apreciable, por ejemplo, comparando los resultados obtenidos por el alfabeto Binario y el Inglés donde la diferencia de caracteres entre uno y otro es de 51.

6 Anexo

Alfabeto \ Largo Patrón	2^2	2^3	2^4	2^5	2^6	2^7
Binario	2^4	2^8	2^{16}	2^{32}	2^{64}	2^{128}
ADN	4^4	4^8	4^{16}	4^{32}	4^{64}	4^{128}
Normal	53^4	53^8	53^{16}	53^{32}	53^{64}	53^{128}

Tabla 1: Tabla que muestra la cantidad de palabras que es posible formar dado un cierto alfabeto y un largo

Algoritmo \ Caso	Mejor	Promedio	Peor
Fuerza Bruta	-	-	$O(n \cdot m)$
KMP	-	$O(n+m)$	$O(n+m)$
BMH	-	$O\left(n \cdot \left(\frac{1}{m} + \frac{1}{2c}\right)\right)$	$O(n \cdot m)$

Tabla 2: Tabla que muestra los ordenes para distintos casos de los algoritmos a estudiar. Con n el largo del texto, m el tamaño de patrón y c la cantidad de caracteres del alfabeto.

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	6578109.201	12869.475	25738.950	6552370.252	6603848.151
8.000	3517857.216	7873.298	15746.596	3502110.619	3533603.812
16.000	2101078.087	6084.092	12168.184	2088909.903	2113246.271
32.000	1318127.721	2910.307	5820.613	1312307.108	1323948.335
64.000	957184.543	2389.986	4779.971	952404.572	961964.515
128.000	731992.843	1834.616	3669.233	728323.610	735662.076

Tabla 3: Valores obtenidos para comparaciones en Lenguaje Natural Real usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	8737119.240	27916.646	55833.292	8681285.948	8792952.532
8.000	8706647.260	30486.726	60973.452	8645673.808	8767620.713
16.000	8210756.638	16469.720	32939.440	8177817.198	8243696.078
32.000	8243950.903	16464.527	32929.055	8211021.848	8276879.958
64.000	8241833.916	16465.788	32931.576	8208902.340	8274765.491
128.000	8180534.028	16246.905	32493.809	8148040.219	8213027.838

Tabla 4: Valores obtenidos para Comparaciones en Lenguaje Natural Real usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	3707297.535	20559.022	41118.045	3666179.490	3748415.579
8.000	3694803.116	21274.292	42548.585	3652254.531	3737351.701
16.000	3667393.489	20551.504	41103.008	3626290.481	3708496.497
32.000	3689669.484	20394.186	40788.373	3648881.111	3730457.857
64.000	3611994.620	20439.613	40879.226	3571115.393	3652873.846
128.000	3641048.509	20366.433	40732.866	3600315.644	3681781.375

Tabla 5: Valores obtenidos para comparaciones en Lenguaje Natural Real usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	5874394.402	5463.345	10926.691	5863467.712	5885321.093
8.000	2979512.906	4423.606	8847.213	2970665.694	2988360.119
16.000	1724489.162	4138.607	8277.214	1716211.948	1732766.376
32.000	891948.111	938.995	1877.989	890070.122	893826.101
64.000	541442.135	683.600	1367.199	540074.936	542809.334
128.000	359336.346	520.349	1040.698	358295.648	360377.044

Tabla 6: Valores obtenidos para comparaciones en Lenguaje Natural Sintético usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	7293763.464	8317.303	16634.607	7277128.857	7310398.071
8.000	7082737.016	5464.599	10929.198	7071807.817	7093666.214
16.000	7171909.653	10895.096	21790.193	7150119.461	7193699.846
32.000	7051702.680	4728.946	9457.892	7042244.788	7061160.572
64.000	7067555.475	5211.844	10423.687	7057131.788	7077979.162
128.000	7023817.428	3721.282	7442.565	7016374.863	7031259.992

Tabla 7: Valores obtenidos para comparaciones en Lenguaje Natural Sintético usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	2123831.665	3549.631	7099.261	2116732.404	2130930.926
8.000	2155102.865	4242.837	8485.674	2146617.191	2163588.538
16.000	2314514.898	7824.677	15649.354	2298865.544	2330164.252
32.000	2115892.371	3793.354	7586.709	2108305.663	2123479.080
64.000	2071248.532	2616.905	5233.811	2066014.721	2076482.343
128.000	2122369.854	4151.010	8302.019	2114067.834	2130671.873

Tabla 8: Valores obtenidos para comparaciones en Lenguaje Natural Sintético usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	15032244.563	39778.227	79556.454	14952688.108	15111801.017
8.000	13462910.113	42572.731	85145.462	13377764.651	13548055.575
16.000	13186310.350	43245.465	86490.930	13099819.420	13272801.280
32.000	13639066.980	50140.488	100280.976	13538786.004	13739347.956
64.000	13199746.269	43199.370	86398.739	13113347.530	13286145.009
128.000	13139913.738	44566.428	89132.856	13050780.881	13229046.594

Tabla 9: Valores obtenidos para comparaciones en alfabeto Binario usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	16682542.448	12004.085	24008.169	16658534.279	16706550.617
8.000	17194018.036	14257.875	28515.750	17165502.286	17222533.786
16.000	17067780.652	11313.400	22626.801	17045153.851	17090407.453
32.000	19033808.032	43292.017	86584.034	18947223.998	19120392.066
64.000	16938606.950	8455.528	16911.057	16921695.893	16955518.006
128.000	17224188.991	20456.814	40913.628	17183275.364	17265102.619

Tabla 10: Valores obtenidos para comparaciones en alfabeto Binario usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	12512106.136	12622.498	25244.995	12486861.140	12537351.131
8.000	12820423.354	14862.972	29725.944	12790697.410	12850149.298
16.000	12547353.243	10373.619	20747.239	12526606.004	12568100.482
32.000	13381928.290	34062.796	68125.592	13313802.698	13450053.881
64.000	12525426.021	10260.999	20521.998	12504904.024	12545948.019
128.000	12499130.753	10328.058	20656.115	12478474.638	12519786.868

Tabla 11: Valores obtenidos para comparaciones en alfabeto Binario usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	10492995.550	25399.368	50798.736	10442196.814	10543794.286
8.000	6556795.045	22965.564	45931.129	6510863.916	6602726.174
16.000	5728676.316	35101.348	70202.695	5658473.621	5798879.011
32.000	4484735.675	19606.328	39212.655	4445523.020	4523948.331
64.000	4480757.636	20071.649	40143.299	4440614.337	4520900.935
128.000	4449894.658	20257.676	40515.352	4409379.305	4490410.010

Tabla 12: Valores obtenidos para comparaciones en ADN real usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	13046764.421	12661.734	25323.468	13021440.953	13072087.889
8.000	12593483.900	15126.179	30252.358	12563231.542	12623736.259
16.000	12386751.585	11374.589	22749.177	12364002.408	12409500.762
32.000	12314797.451	10608.182	21216.364	12293581.086	12336013.815
64.000	12453692.589	15692.112	31384.225	12422308.364	12485076.814
128.000	12640267.928	31642.606	63285.212	12576982.715	12703553.140

Tabla 13: Valores obtenidos para comparaciones en ADN real usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	8481851.934	11621.097	23242.195	8458609.739	8505094.129
8.000	8429892.381	13448.102	26896.204	8402996.177	8456788.584
16.000	9147192.235	35096.172	70192.344	9076999.891	9217384.579
32.000	8331167.632	11477.869	22955.738	8308211.894	8354123.370
64.000	8342153.192	11252.201	22504.402	8319648.790	8364657.594
128.000	8509480.500	17604.622	35209.244	8474271.256	8544689.744

Tabla 14: Valores obtenidos para comparaciones en ADN real usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	10351440.237	29953.653	59907.306	10291532.932	10411347.543
8.000	6502875.853	23159.286	46318.573	6456557.280	6549194.425
16.000	4887234.324	17920.158	35840.316	4851394.008	4923074.640
32.000	4736186.044	18183.274	36366.547	4699819.496	4772552.591
64.000	4755012.463	18246.106	36492.213	4718520.250	4791504.676
128.000	4729959.273	18246.884	36493.767	4693465.506	4766453.040

Tabla 15: Valores obtenidos para comparaciones en ADN sintético usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	13002935.157	15479.258	30958.516	12971976.641	13033893.672
8.000	12855306.936	34503.929	69007.858	12786299.078	12924314.794
16.000	13520112.626	39142.451	78284.902	13441827.724	13598397.528
32.000	12352004.835	10569.202	21138.404	12330866.431	12373143.239
64.000	12323798.011	9311.426	18622.852	12305175.159	12342420.863
128.000	12284206.353	8642.969	17285.938	12266920.415	12301492.290

Tabla 16: Valores obtenidos para comparaciones en ADN sintético usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	8694804.459	16759.651	33519.301	8661285.158	8728323.761
8.000	8914373.438	25206.836	50413.671	8863959.766	8964787.109
16.000	8591057.806	17067.632	34135.265	8556922.541	8625193.070
32.000	8444128.864	9460.872	18921.744	8425207.119	8463050.608
64.000	8468170.410	10067.740	20135.481	8448034.929	8488305.890
128.000	10433218.799	11558.568	23117.137	10410101.662	10456335.935

Tabla 17: Valores obtenidos para comparaciones en ADN sintético usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	9152343.538	11297.456	22594.912	9129748.626	9174938.449
8.000	4957003.779	7752.639	15505.279	4941498.500	4972509.057
16.000	2573349.057	4842.821	9685.641	2563663.416	2583034.698
32.000	1827520.804	3212.193	6424.387	1821096.418	1833945.191
64.000	1255552.759	2721.588	5443.176	1250109.583	1260995.935
128.000	964508.112	2147.899	4295.797	960212.314	968803.909

Tabla 18: Valores obtenidos para tiempos en Lenguaje Natural Real usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	6882702.667	16096.156	32192.312	6850510.355	6914894.979
8.000	6863230.418	16145.543	32291.086	6830939.332	6895521.504
16.000	6894445.826	16118.377	32236.754	6862209.072	6926682.580
32.000	6881517.107	16243.111	32486.222	6849030.886	6914003.329
64.000	6881846.713	16316.614	32633.229	6849213.485	6914479.942
128.000	6883999.882	16220.628	32441.256	6851558.626	6916441.138

Tabla 19: Valores obtenidos para tiempos en Lenguaje Natural Real usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	2871488.513	18919.425	37838.850	2833649.662	2909327.363
8.000	2838863.468	17969.052	35938.105	2802925.363	2874801.572
16.000	2832206.921	18454.302	36908.604	2795298.317	2869115.524
32.000	2903822.383	20440.471	40880.943	2862941.440	2944703.325
64.000	2871347.213	19932.181	39864.363	2831482.850	2911211.576
128.000	2877290.982	20159.244	40318.487	2836972.495	2917609.469

Tabla 20: Valores obtenidos para tiempos en Lenguaje Natural Real usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	7427037.911	4312.915	8625.831	7418412.080	7435663.742
8.000	3696656.080	2664.694	5329.388	3691326.692	3701985.469
16.000	1991341.045	1500.694	3001.388	1988339.657	1994342.433
32.000	1097269.955	878.879	1757.757	1095512.198	1099027.712
64.000	634746.549	558.137	1116.274	633630.275	635862.823
128.000	429312.138	487.825	975.650	428336.488	430287.788

Tabla 21: Valores obtenidos para tiempos en Lenguaje Natural Sintético usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	5640382.136	464.270	928.541	5639453.595	5641310.676
8.000	5642397.606	1916.418	3832.835	5638564.771	5646230.441
16.000	5641323.734	501.984	1003.967	5640319.766	5642327.701
32.000	5640908.250	483.305	966.610	5639941.641	5641874.860
64.000	5641631.830	483.182	966.365	5640665.465	5642598.195
128.000	5640354.568	430.443	860.885	5639493.682	5641215.453

Tabla 22: Valores obtenidos para tiempos en Lenguaje Natural Sintético usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	1570416.499	5358.028	10716.056	1559700.443	1581132.555
8.000	1539502.829	3762.609	7525.218	1531977.610	1547028.047
16.000	1578280.798	5784.356	11568.711	1566712.087	1589849.509
32.000	1557271.016	4642.748	9285.495	1547985.521	1566556.512
64.000	1559269.061	4738.222	9476.443	1549792.618	1568745.504
128.000	1559776.792	4754.873	9509.746	1550267.047	1569286.538

Tabla 23: Valores obtenidos para tiempos en Lenguaje Natural Sintético usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	15390899.547	42839.285	85678.570	15305220.977	15476578.117
8.000	13453621.679	42011.333	84022.666	13369599.013	13537644.344
16.000	13314132.622	42514.966	85029.931	13229102.691	13399162.553
32.000	13307467.602	42855.266	85710.532	13221757.070	13393178.133
64.000	13293837.882	42940.458	85880.915	13207956.967	13379718.798
128.000	13254619.604	42611.053	85222.105	13169397.499	13339841.709

Tabla 24: Valores obtenidos para tiempos en alfabeto Binario usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	13530980.995	1877.825	3755.649	13527225.346	13534736.645
8.000	14109311.418	4002.912	8005.824	14101305.594	14117317.242
16.000	14193886.242	3958.657	7917.314	14185968.929	14201803.556
32.000	14200127.982	3966.009	7932.018	14192195.964	14208060.000
64.000	14191556.321	3933.397	7866.794	14183689.527	14199423.114
128.000	14197327.293	3980.313	7960.626	14189366.668	14205287.919

Tabla 25: Valores obtenidos para tiempos en alfabeto Binario usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	9584147.779	24882.505	49765.010	9534382.769	9633912.788
8.000	9571860.785	17187.194	34374.389	9537486.396	9606235.174
16.000	9604734.426	19907.824	39815.648	9564918.778	9644550.073
32.000	9674579.816	21795.447	43590.894	9630988.922	9718170.709
64.000	9602520.544	20289.122	40578.244	9561942.299	9643098.788
128.000	9618140.544	19715.022	39430.045	9578710.499	9657570.588

Tabla 26: Valores obtenidos para tiempos en alfabeto Binario usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	11260640.099	26322.145	52644.289	11207995.809	11313284.388
8.000	7259350.141	24855.488	49710.976	7209639.165	7309061.117
16.000	5578755.848	21575.539	43151.078	5535604.770	5621906.926
32.000	5105408.935	22233.096	44466.191	5060942.744	5149875.126
64.000	5099121.635	22349.657	44699.314	5054422.321	5143820.949
128.000	5004506.409	22422.499	44844.997	4959661.412	5049351.406

Tabla 27: Valores obtenidos para tiempos en ADN Real usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	10584383.319	5186.697	10373.394	10574009.926	10594756.713
8.000	10623347.843	5167.555	10335.110	10613012.734	10633682.953
16.000	10624166.561	5100.005	10200.010	10613966.551	10634366.572
32.000	10620953.099	5327.365	10654.729	10610298.370	10631607.828
64.000	10614849.679	5138.802	10277.603	10604572.075	10625127.282
128.000	10613481.783	5235.173	10470.346	10603011.437	10623952.129

Tabla 28: Valores obtenidos para tiempos en ADN Real usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	6625742.867	18030.593	36061.187	6589681.680	6661804.054
8.000	6662105.944	21307.765	42615.530	6619490.413	6704721.474
16.000	6608071.456	19016.226	38032.453	6570039.003	6646103.909
32.000	6615564.989	18403.548	36807.096	6578757.893	6652372.085
64.000	6661893.446	20864.520	41729.040	6620164.405	6703622.486
128.000	6623709.457	17934.738	35869.475	6587839.982	6659578.933

Tabla 29: Valores obtenidos para tiempos en ADN Real usando KMP

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	11044822.167	24652.153	49304.306	10995517.861	11094126.473
8.000	7041101.792	22083.681	44167.361	6996934.430	7085269.153
16.000	5605473.787	19859.933	39719.865	5565753.922	5645193.653
32.000	5354470.021	19959.196	39918.392	5314551.629	5394388.413
64.000	5343339.630	19906.430	39812.861	5303526.770	5383152.491
128.000	5351921.830	19901.446	39802.892	5312118.938	5391724.722

Tabla 30: Valores obtenidos para tiempos en ADN Sintético usando BMH

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	10768391.071	4045.461	8090.923	10760300.148	10776481.994
8.000	10762301.687	3389.295	6778.590	10755523.097	10769080.277
16.000	10761686.019	3402.589	6805.178	10754880.840	10768491.197
32.000	10760577.675	3445.142	6890.284	10753687.391	10767467.959
64.000	10759687.895	3868.298	7736.596	10751951.299	10767424.491
128.000	10753984.878	3418.770	6837.539	10747147.339	10760822.418

Tabla 31: Valores obtenidos para tiempos en ADN Sintético usando Fuerza Bruta

n	Promedio	Desviacion	ErrorMax	MinIntervalo	MaxIntervalo
4.000	6745999.971	15402.543	30805.087	6715194.884	6776805.057
8.000	6765059.531	17659.793	35319.586	6729739.945	6800379.117
16.000	6855851.082	22725.844	45451.689	6810399.393	6901302.771
32.000	6831776.735	21522.272	43044.543	6788732.191	6874821.278
64.000	6760180.346	17539.885	35079.770	6725100.576	6795260.116
128.000	6761921.741	17166.755	34333.511	6727588.230	6796255.252

Tabla 32: Valores obtenidos para tiempos en ADN Sintético usando KMP