Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC4102 Diseño y Análisis de Algoritmos
Prof: Gonzalo Navarro.

Búsqueda en Texto
Tarea 1
Informe

Nicolás Salas V.
Daniel Rojas C.

October 13, 2015

Contents

1 Introducción

El presente informe tiene como objetivo comparar algoritmos de búsqueda en texto y obtener conclusiones a partir de los resultados. Como es de esperar, se espera saber si hay un algoritmo que sea mejor que los otros en cualquier condición, o, si no es posible afirmar esto, saber qué algoritmo se comporta mejor bajo qué condiciones.

En particular, se analizarán tres algoritmos. Existen más, pero para esta tarea se evaluarán tres, por ser los más conocidos:

- Algoritmo de Fuerza Bruta
- Algoritmo de Knuth-Morris-Pratt
- Algoritmo de Boyer-Moore-Horspool

Para efectos de experimentación, las cantidades que interesa medir de los Algoritmos son:

1. Tiempo de Ejecución del Algoritmo
2. Número de Comparaciones

Dentro de la sección del diseño y discusión de los algoritmos se explica cómo funcionan éstos. Pero a grandes rasgos el Algoritmo de Fuerza Bruta es el más ingenuo de todos, el Algoritmo de Knuth Morris Pratt aprovecha las reocurrencias de *substrings* del patrón en él mismo y el Algoritmo de Boyer Moore Horspool analiza el patrón de atrás hacia delante cuidando de realinear el patrón con el texto en la última ocurrencia de la discrepancia en el patrón.

Sólo entendiendo cómo funcionan estos algoritmos es posible elaborar algunas hipótesis de cómo debieran comportarse estos algoritmos frente a distintos *input*s.

1.1 Hipótesis

En términos de comparaciones, es fácil predecir que el algoritmo que realizará la mayor cantidad de comparaciones es el algoritmo de fuerza bruta, porque no aprovecha nada de lo que "aprende" sobre las concordancias entre el patrón y el texto. Como el tiempo se parametriza en función de las comparaciones, se puede predecir que el Algoritmo de Fuerza Bruta será el que peor funcione de todos.

Por su parte, el Algoritmo de Knuth Morris Pratt sí que aprovecha el hecho de que a veces el texto y el patrón coinciden, y hace usufructo de este factor[1], esto significa que el Algoritmo hace **por lo menos** tantas comparaciones como carácteres existen en el texto, y adicionalmente puede o no haber algunas más. Con ello una predicción que se puede hacer es que este algoritmo debiese ser bastante bueno cuando el patrón se autocontiene en sí mismo una o más veces, de modo que en el alfabeto binario éste debiera ser el algoritmo que mejor lo haga.

[1]Revisar la parte de Diseño Experimental para una explicación más detallada de los Algoritmos

Finalmente, el Algoritmo de Boyer Moore Horspool, al analizar el patrón en dirección reversa y usar una función de salto para reindexar el texto, debería ser el mejor cuando constantemente hay discrepancias entre el texto y el patrón. Es interesante notar que al reindexar desde el último carácter examinado del texto, el Algoritmo de Boyer Moore Horspool realiza menos comparaciones que las palabras del texto, y su mejor caso es cuando el carácter examinado del texto no existe en el patrón. Cualquier caso donde es altamente probable este fallo es un buen candidato para experimentar con este algoritmo. En términos de los experimentos, se espera ver que este Algoritmo es mejor en casos donde las discrepancias son más probables (por ejemplo al buscar en un texto cualquiera usando Ctrl+F). Lo interesante de los experimentos a realizar es comparar los tiempos de ejecución en sus mejores y peores casos.

2 Diseño Experimental y Análisis de Algoritmos

2.1 Lenguaje de Preferencia

Para hacer el trabajo se ha escogido el lenguaje `C`, por ser un lenguaje que, en general, no estorba a la hora de medir el tiempo. Al escoger C, se desprecia el tiempo asociado a cualquier elemento extraño que no se quiera que influya en las mediciones, entre otros, se pueden destacar *Garbage Collectors*, *Estados internos de intérpretes*, *Procesos de Máquinas virtuales*. El objetivo es tener el mayor control posible sobre el tiempo y la memoria que usan los algoritmos.

2.2 Detalles de Implementación

En esta sección se mostrará los algoritmos en seudocódigo. Si es necesario, además se explicarán las partes críticas que no sean trivialmente entendibles. Además, en los detalles de implementación no se pondrá explícitamente dónde y cómo se almacenan los resultados que interesa medir, en esta parte sólo se explica cómo funcionan los algoritmos[2].

Algoritmo de Fuerza Bruta

El algoritmo en general es simple, por lo que aquí se expone el seudocódigo del mismo.

```
T <- texto
P <- patron
m <- length P
n <- length T
for k in 1..n
  j <- k
  for i in 1..m
    if T[j] = P[i] then j <- j+1
    else break
  if j-k = m then AddResultado(k)
```

Como se puede ver, el algoritmo simplemente compara el texto con el patrón letra por letra y al producirse una discrepancia se avanza el texto en 1 posición. Si, en cambio, se produjera concordancia entonces debe seguir comparándose el patrón con el texto hasta finalizar el patrón. Si existe un calce completo entonces simplemente se agrega como resultado la posición donde empezaba el calce.

Según la misma página del profesor, el tiempo que toma el algoritmo es $O(n \cdot m)$ para encontrar todas las ocurrencias posibles del patrón en el texto. Nótese que esto es un análisis pesimista en el sentido de que no siempre se recorren m caácteres del patrón por cada uno de los n carácteres del texto. No obstante, no hay condiciones favorables predecibles que ayuden a acotar este tiempo del algoritmo.

[2]En los apéndices se incluye una explicación más detallada respecto a dónde y cómo se guardan los resultados.

Algoritmo de Knuth-Morris-Pratt

Este algoritmo hace usufructo del hecho de que dentro del patrón puede haber subcadenas que se repitan al principio y al final. Si al comparar el texto con el patrón existen muchos calces antes de encontrar una discrepancia, se puede usar "lo que se ha aprendido" hasta ese momento para ver desde dónde se puede empezar a recomparar el texto. Al encontrarse una discrepancia se puede recalcular el índice del patrón para recalzar una subcadena más chica de él con el carácter del texto. Este proceso se vuelve a repetir hasta que haya un calce o bien se llegue al principio del patrón. Como se explicaba en la introducción, el hecho de que se haga recalzar el patrón con sus prefijos es un hecho que se aprovecha muy bien en este algoritmo.

Para encontrar los índices en caso de discrepancias se calcula una *función de fracaso*, que almacena para cada índice j, el largo del prefijo más largo de $P[1, j]$ que es también sufijo de $P[1, j]$. Si no existe dicho prefijo, entonces la función de fracaso de j vale 0. Entiendiendo esto, la función de fracaso sirve para el algoritmo KMP en el sentido de que almacena el índice donde se debe empezar a recomparar el patrón (que es justamente el mayor largo posible de una subcadena que es prefijo y sufijo a la vez de una subcadena del patrón).

La implementación que se ha usado aquí para los experimentos corresponde a la explicada en los Apuntes[3] del curso CC3001 - Algoritmos y Estructuras de Datos, elaborados por el profesor Benjamín Bustos. En este contexto se explica en seudocódigo cómo calcular la función de fracaso y el algoritmo propiamente tal.

```
# calculo de la funcion de fracaso
P <- patron
m <- length P
f <- array[m];
f[1] <- 0
j=1;
while j<m
  i <- f[j]
  while i>0 and P[i+1]!=P[j+1]) do i=f[i]
  if (patron[i+1]==patron[j+1]) then
    f[j+1]=i+1
  else
    f[j+1]=0
  endif
  j++
```

Después de esto la función de fracaso queda almacenada en el array `f`. Y con ello se puede implementar el algoritmo de Knuth-Morris-Pratt, que coincidentemente es muy parecido al cálculo de la función de fracaso.

```
# algoritmo de KMP
T <- texto
```

[3]Se pueden ver en http://users.dcc.uchile.cl/ bebustos/apuntes/cc3001/BusqTexto/#2

```
P <- patron
n <- length T
m <- length P
k <- 0;
j <- 0;
while k<n do
  while j>0 and T[k+1]!=P[j+1] do j <- f[j]
  if texto[k+1] == patron[j+1]
    j <- j+1
  k <- k+1
  #si hay calce simular falla para seguir calculando
  if j==m then
    AddResultado(k-j)
    j <- f[j]
end
```

En esta implementación, se ha considerado que los índices van desde 1 a n, por lo que la implementación real de este algoritmo en C que se ha provisto varía. En los apéndices se discute un poco más a fondo el asunto.

El algoritmo de Knuth-Morris-Pratt también toma tiempo $O(n \cdot m)$ con un análisis pesimista. Pero en este caso hay decisiones que toma el algoritmo que son claves para encontrar una cota *promedio* para él. Al producirse una discrepancia, este algoritmo **no vuelve al principio del patrón**, sino que recalcula alguna posible coincidencia entre el patrón y el texto, potencialmente saltándose carácteres del patrón por cada discrepancia de éste con el texto. Esto da una cota promedio para el algoritmo de $O(n)$.

Algoritmo de Boyer-Moore-Horspool

El Algoritmo de Boyer Moore Horspool analiza el **patrón** en forma inversa y el texto en forma "directa". Antes de empezar a calcular las ocurrencias del patrón en el texto se precalcula una *función de salto*, que asocia a cada letra del patrón la cantidad de carácteres que deben saltarse desde una posición del texto para reanalizar una futura ocurrencia del patrón en el texto. En otras palabras, la función de salto asocia a cada **letra del patrón** su última ocurrencia dentro de éste.

Es interesante explicar (antes de seguir), que existen distintos tipos de implementaciones para este mismo algoritmo. Para este trabajo se consideró, en general, el enfoque de los apuntes del Profesor Benjamín Bustos[4].

```
T <- array[256]
P <- patron
define salto(T,P)
  for i in 1..256
    T[i] = 0
  for k in 1..(length P) -1
    T[patron[k]] = k;
```

[4]Estos apuntes se pueden ver en http://users.dcc.uchile.cl/ bebustos/apuntes/cc3001/BusqTexto/#31

La función de salto hace usufructo de que los `char` pueden ser representados como enteros asumiendo que la longitud de un `char` es 1 byte. Con la definición antes dada, las letras que no pertenecen al patrón tienen `salto=0` y las letras que están antes de la última letra del patrón tienen *su última ocurrencia* asociada, en términos de posición. Nótese que la función de salto no calcula el salto para la última letra del patrón, porque precisamente esa es la letra que *siempre* se compara.

Lo medular es usar la función de salto para agilizar el proceso de la búsqueda. El objetivo de Boyer Moore Horspool es "deslizar" el patrón tanto como se pueda si es que los comparandos difieren, ése es el rol fundamental de la función de salto: si no hay ninguna letra del patrón en el pedazo de texto que estaba comparando con el patrón, entonces el patrón se puede deslizar m carácteres (donde m es la cantidad de carácteres en el patrón). Si, por el contrario, *hay* una letra del patrón en dicho pedazo de **texto**, entonces el patron calza la última letra del texto que se estaba comparando, con la última ocurrencia de dicha letra en el patrón. Esta última ocurrencia está dada justamente por la función de salto.

```
T <- texto
P <- patron
m <- length P
n <- length T
k <- m;
j <- m;
while k<=n and j>=1
  if T[k-(m-j)] = P[j] then j <- j-1
  else
    k <- k + m - salto[T[k]]
    j <- m
  if j = 0
    AddResultado(k - m)
    k <- k + m - salto[T[k]]
    j <- m
```

Nuevamente, esta exposición de seudocódigo es sólo eso, y su implementación real difiere de esta en el sentido de que los índices empiezan desde 0, pero el espíritu del algoritmo es el mismo. De todas maneras, se expone la implementación más real en los apéndices.

Este algoritmo, al igual que los otros dos, tiene tiempo de ejecución $O(nm)$, pero nótese, que aquí la función de salto produce -valga la redundancia- saltos sobre el *texto*, de manera que no todos los carácteres de éste son siempre comparados, el mejor caso produce un salto cada m carácteres del texto, y asumiendo que hay n carácteres esto da una cota $O(\frac{n}{m})$ para este algoritmo.

2.3 Casos de Prueba

Se harán pruebas sobre 5 casos:

- Alfabeto Binario

- ADN Real
- ADN Sintético
- Lenguaje Natural
- Lenguaje Sintético

Aquí el Alfabeto Binario corresponde a una cadena[5] compuesta sólo por los carácteres `0` y `1`. Los ADN son cadenas donde todo carácter de ésta está en el conjunto { `A, T, G, C` }. Finalmente, el caso de los Lenguajes corresponden a texto natural con todo carácter extraño extraído, es decir, son cadenas compuestas por letras de la A a la Z tanto mayúsculas como minúsculas y eventualmente puede haber un espacio.

Además, los casos de Alfabeto Binario, ADN Sintético y Lenguaje Sintético corresponden a cadenas generadas aleatoriamente según el caso, mientras que los casos de ADN y Lenguaje Real corresponden a ADN real (es decir, ADN de una célula) y textos reales, respectivamente. Para los casos reales este trabajo ha considerado los ADN y Textos provistos por Pizza & Chili Corpus[6], en ambos casos, se tomaron los archivos más pequeños que se pueden descargar, por asuntos de tiempo.

Respecto de los tamaños de los textos, en general se han considerado textos de al menos 1MB de largo, es decir, 10^6 carácteres. Para probar estos algoritmos, se han tomando patrones extraídos de un mismo texto de tamaño $2^k; k \in \{2, 3, ..., 7\}$, salvo en el caso del Alfabeto binario, donde el patrón se genera al azar.

Estos casos de prueba permitirán medir las distintas hipótesis que se han planteado en la Introducción de este informe, pero sólo para recordar, con los resultados de estos casos de prueba va a ser posible medir qué algoritmos son mejores para qué tamaños de alfabetos, se podrá medir el tiempo de ejecución en función del *input* del Algoritmo y esto se puede contrastar con lo que dice la teoría, también puede ser interesante ver si es que hay casos en que Knuth Morris Pratt y Boyer Moore Horspool no son mejores que Fuerza Bruta.

2.4 Medidas de Rendimiento

El rendimiento de los algoritmos se ha medido con dos parámetros que son los que siempre definen el tiempo de un algoritmo.

Experimentalmente, el tiempo es la variable de mayor interés, pues es lo que la teoría trata de aproximar. Teóricamente, el tiempo se aproxima comparando las operaciones núcleo que realiza algún algoritmo. En este caso, la operación que domina los algoritmos son las comparaciones. Por ello es que en este experimento se adoptará la convención de medir el tiempo de ejecución y el número de comparaciones. Se tratará de mostrar la correlación que existe entre estas variables, y también se podría ver si existe alguna otra variable que influya

[5]En el ámbito de las Ciencias de la Computación, *cadena* se suele usar como la traducción de String al español.

[6]http://pizzachili.dcc.uchile.cl/texts.html

en los tiempos de ejecución, aparte del input del problema.

Antes de seguir, es necesario recalcar que todos los resultados entregados y calculados cumplen las condiciones de que: miden el tiempo **absoluto** de ejecución, considerando todos los preprocesamientos, los algoritmos **no almacenan los resultados encontrados**, hacer ello mientras se ejecuta el algoritmo invocaría tiempos de ejecución asociados a propiedades internas de las estructuras de datos que los guardan, por último, todos los resultados entregados de tiempo y comparaciones están asegurados con un 95% de confianza, de modo que para la obtención de estos resultados, se itera varias veces, hasta disminuir la desviación estandar del promedio de ejecución e iteraciones.

3 Resultados

Esta sección muestra los resultados de los experimentos realizados. En particular se incluyen las comparaciones entre algoritmos para los distintos experimentos que se realizaron. Se entregan 5 gráficos para el tiempo de ejecución y 5 gráficos para el número de comparaciones. Aquí sólo se exponen los resultados y en la próxima sección se discuten ellos, para contrastas las hiótesis.

3.1 Tiempos de ejecución

Antes de comenzar a revisar los gráficos, se pide al lector que **mire la escala cuidadosamente**. Algunos resultados son increíblemente parecidos, y para mostrar la diferencia se ha escalado el eje y. En ese sentido, si no se tiene cuidado con los límites de este eje la mirada descuidada de los gráficos puede inducir a error.

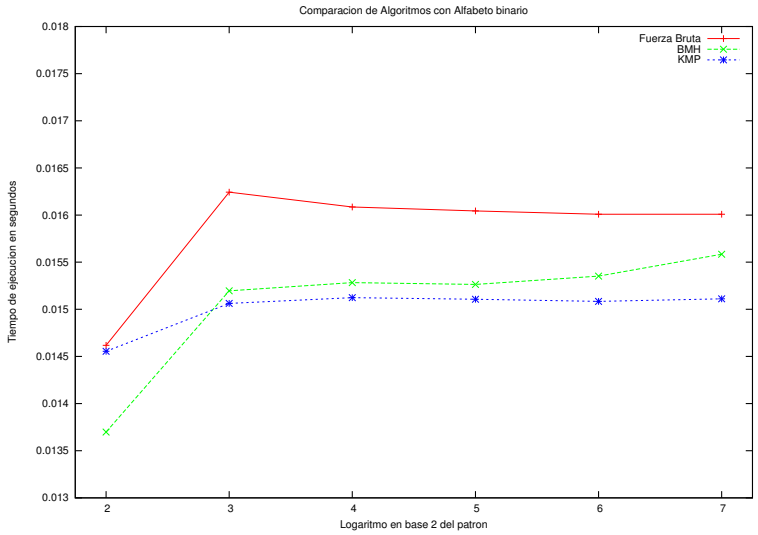

Figure 1: Tiempo de Ejecución en un Alfabeto Binario. El eje x está en escala logarítmica

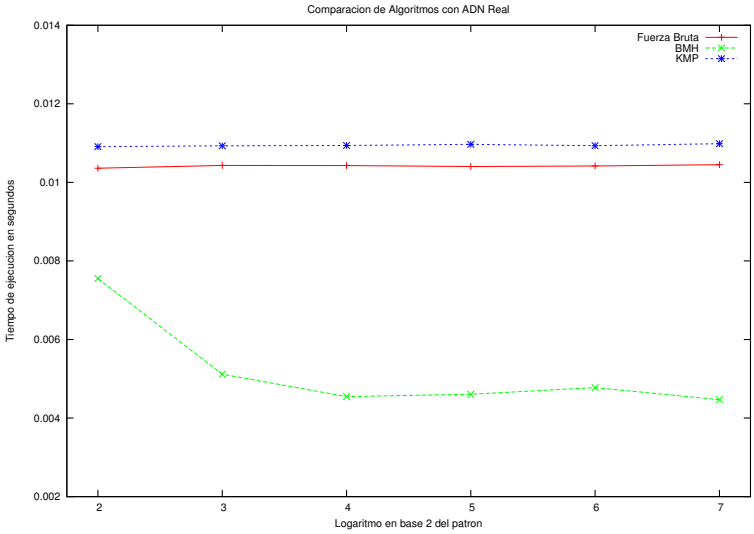

Figure 2: Tiempo de Ejecución en un Alfabeto que contiene ADN Real. El eje x está en escala logarítmica

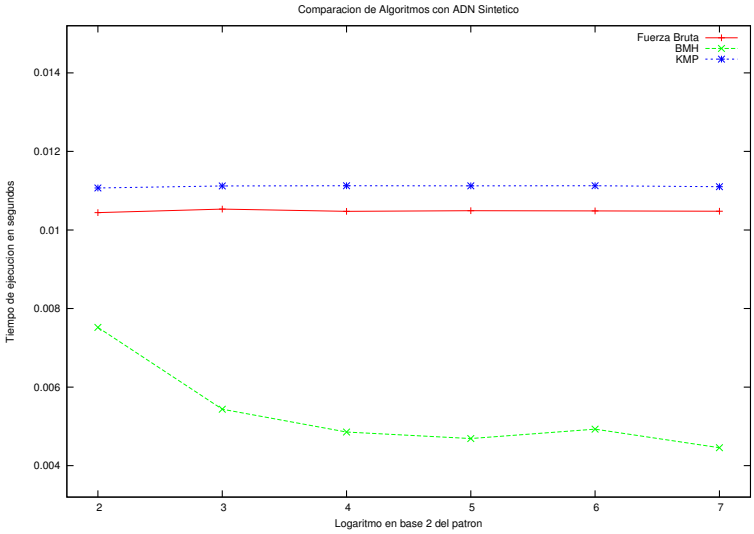

Figure 3: Tiempo de Ejecución en un Alfabeto que contiene ADN generado al azar. El eje x está en escala logarítmica

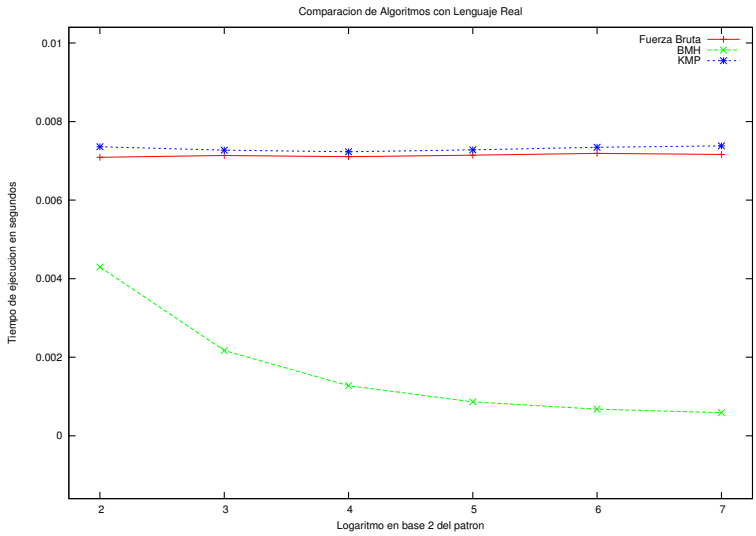

Figure 4: Tiempo de Ejecución en un Texto con Lenguaje real. El eje x está en escala logarítmica

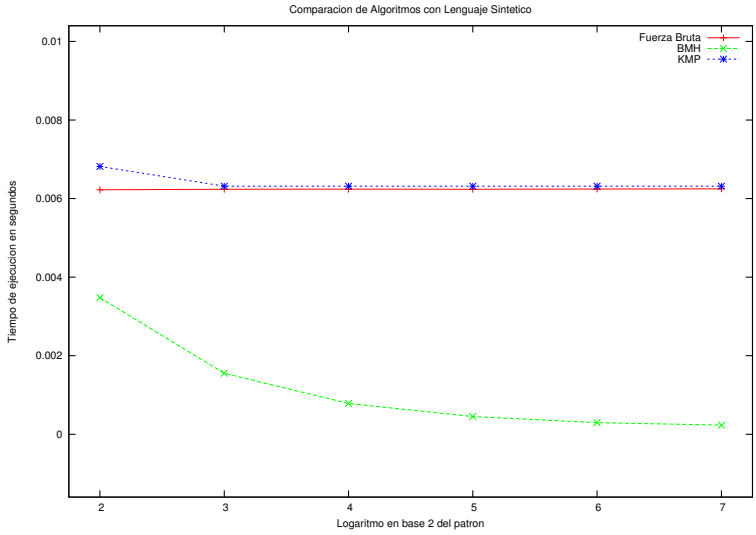

Figure 5: Tiempo de Ejecución en un Texto generado con caracteres del ingles al azar. El eje x está en escala logarítmica

3.2 Comparaciones

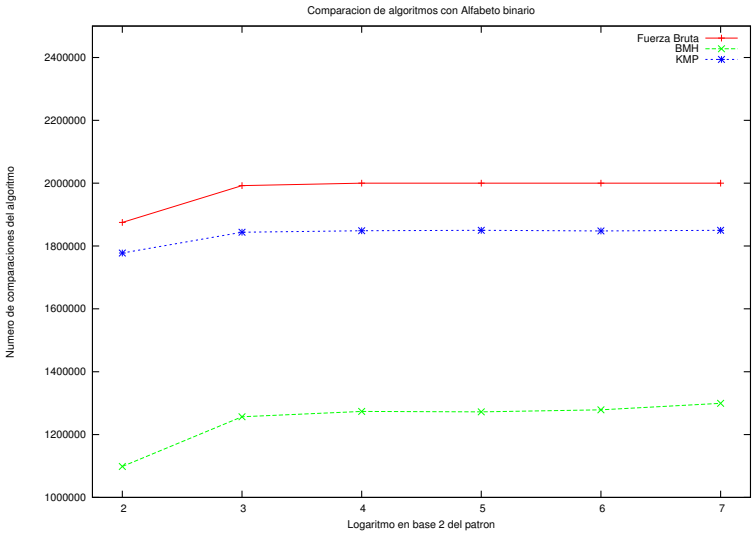

Figure 6: Cantidad de comparaciones en un Alfabeto Binario. El eje x está en escala logarítmica

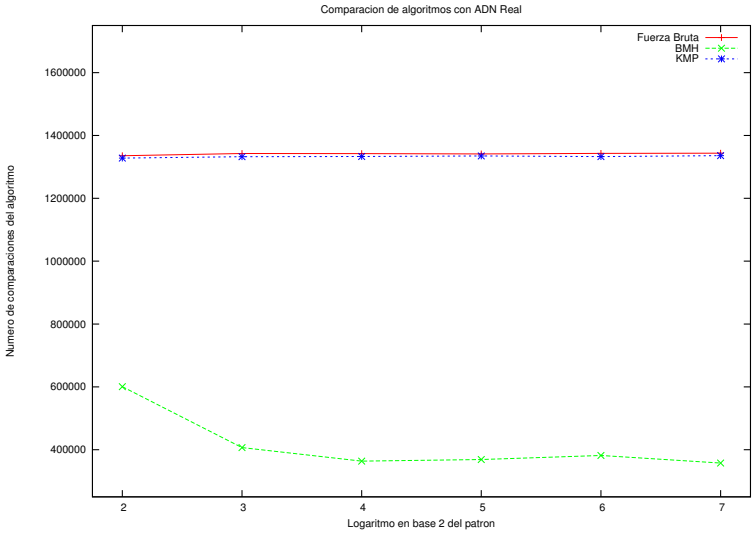

Figure 7: Cantidad de comparaciones en un Alfabeto que contiene ADN Real. El eje x está en escala logarítmica

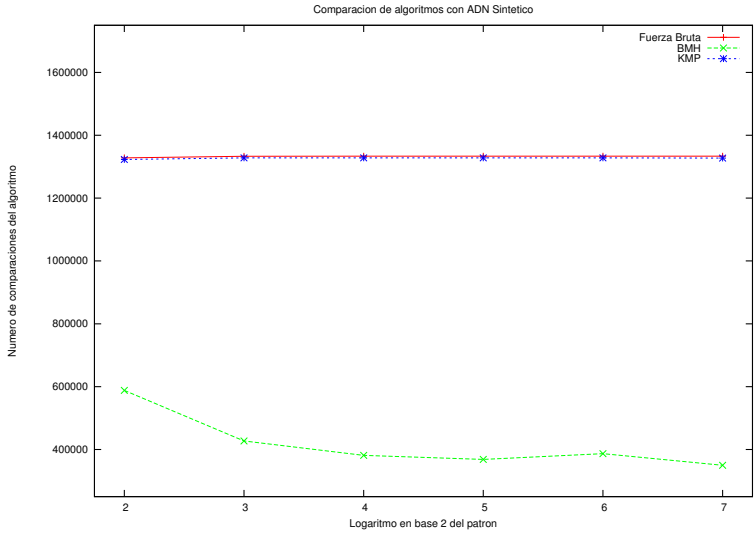

Figure 8: Cantidad de comparaciones en un Alfabeto que contiene ADN generado al azar. El eje x está en escala logarítmica

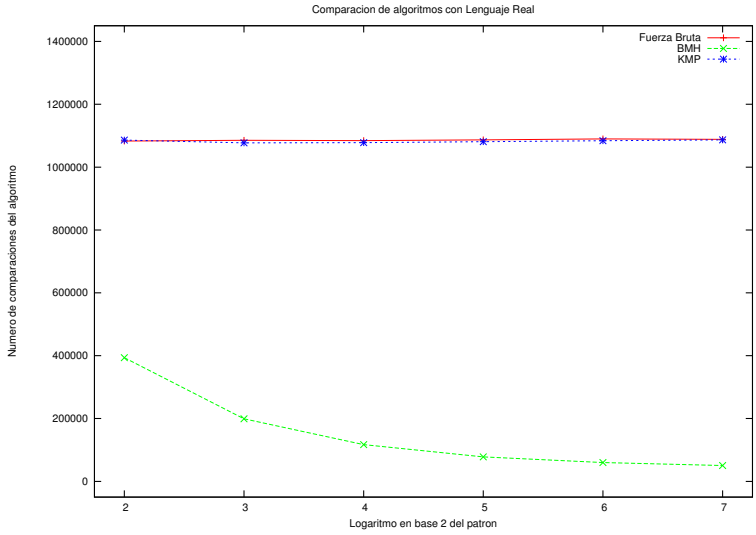

Figure 9: Cantidad de comparaciones en un Texto con Lenguaje real. El eje x está en escala logarítmica

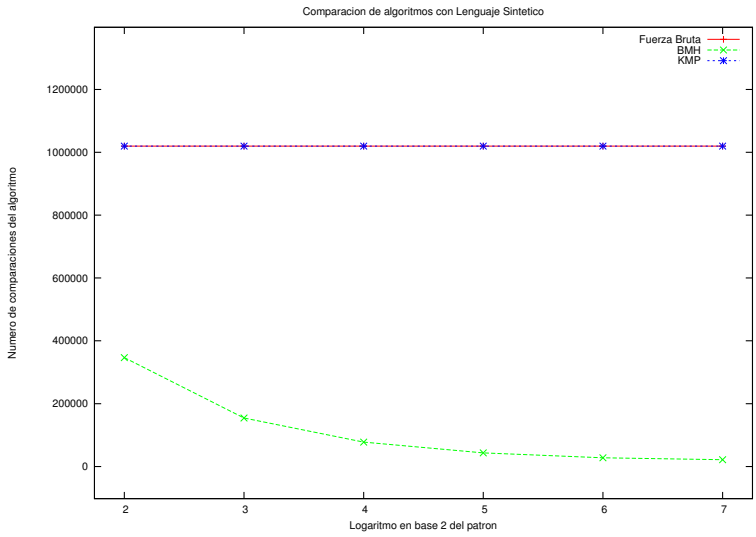

Figure 10: Cantidad de Comparaciones en un Texto generado con caracteres del ingles al azar. El eje x está en escala logarítmica

Nótese que en los últimos gráficos la cantidad de comparaciones es tan similar que pareciera que los gráficos se traslapan. En los apéndices se incluye una tabla que ilustra los parecidos entre los algoritmos KMP y Fuerza Bruta para cada experimento.

4 Interpretación de los Resultados

Los gráficos dan resultados sorprendentes, confirmando en algunos casos lo que se discutió en las hipótesis y rechazándolo en otros casos. En esta sección se discuten dichos tópicos.

4.1 Alfabeto Binario

Este es uno de los gráficos con los que hay que tener cuidado al leer, nótese que cada barra del gráfico aumenta en un factor de $5 \cdot 10^{-4}$, de manera que los resultados para este caso son **muy similares**. En todo caso, se ve que KMP es mejor que BMH, que a su vez es más rápido que Fuerza Bruta. Esto es esperado en el sentido de que era predecible, y fue discutido en las hipótesis.

En términos de comparaciones, Fuerza Bruta realiza más comparaciones que KMP, y KMP realiza más comparaciones que BMH. En este sentido, es raro el comportamiento que tiene BMH por sobre los otros dos algoritmos, pues aun teniendo menos comparaciones, se demora más. Esto -en parte- se debe a la tabla estándar de salto con la que se ha implementado el algoritmo. Un `char` tiene 256 posibles distinciones, pero un `bit` sólo tiene 2. El cálculo de saltos innecesarios respecto del alfabeto marca un tiempo de ejecución no variable al principio del algoritmo que influye de cierta manera en su tiempo de ejecución y no en el número de comparaciones.

4.2 ADN Real y Sintético

Los gráficos tanto de comparaciones como de tiempo de ejecución son muy similares para ambos casos. Esto da una medida de que hasta cierto punto, el ADN Real es de cierta manera generado al azar.

Y aunque sean similares, llama la atención el hecho de que KMP en este caso se demora más que el algoritmo de Fuerza Bruta, que ciertamente era algo que no se esperaba como hipótesis, sobre todo sabiendo que ambos realizan aproximadamente el mismo número de comparaciones.

No es difícil ver que para un número similar de comparaciones sobre un texto, fuerza bruta **siempre** será mejor que KMP, porque KMP antes de empezar sus cálculos hace un precálculo de la función de fracaso. Cuando finalmente se va a realizar el mismo número de comparaciones, en realidad es mucho mejor no precalcular la función de fracaso y proceder con un enfoque ingenuo que, de todas maneras, será más rápido y, por cierto, no ocupará memoria innecesaria para demorarse más.

Por otro lado, Boyer Moore Horspool gana formidablemente en esta competencia, incluso realizando menos comparaciones que los otros dos algoritmos y que el largo del texto. En este sentido el precálculo de una función de salto sí que ayuda a disminuir el tiempo de ejecución a costo de uso de memoria $O(c)$ donde c es el tamaño del número de carácteres permitidos en el texto y patrón.

4.3 Lenguajes Real y Sintético

En este caso vuelve a ocurrir la misma situación que con el ADN, pero la diferencia se hace más drástica. BMH realiza menos que la mitad de los carácteres del texto de comparaciones e incluso *disminuye* su tiempo de ejecución a medida que *aumenta* el tamaño del patrón. Esto es contraintuitivo, pero puede ser explicado de mejor manera en la siguiente subsección, donde se habla del tamaño del alfabeto.

Respecto de los otros dos algoritmos, se ve que la cantidad de comparaciones incluso disminuye, asunto que también es contraintuitivo y no se discutió sobre eso en la sección de Hipótesis. Lo que se puede extraer de esto es también tema de la siguiente subsección, donde se discute el tamaño del alfabeto.

4.4 Discusión sobre el tamaño del alfabeto

Como se venía ya introduciendo, el rol que juega el alfabeto en términos de el desempeño de los algoritmos no es despreciable. Ya se ve que -y sobre todo en los casos de ADN y Lenguaje real- los números de comparaciones disminuyen a medida que aumenta el tamaño del alfabeto. Recuérdese que para estos experimentos se ha dejado el tamaño del texto fijo e igual a 1MB de carácteres.

Es precisamente el tamaño del alfabeto lo que da resultados tan inesperados. Al tener un alfabeto de sólo dos carácteres la probabilidad de que un substring de un patrón (tanto al azar como real) se autocontenga en el patrón mismo es bastante más alta que en cualquier otro caso, y esta es la razón de por qué KMP es el mejor algoritmo cuando el alfabeto es Binario, pues el reaprovechamiento de los subpatrones autocontenidos es más probable mientras el alfabeto es chico.

En los casos de ADN, que tienen un alfabeto de tamaño 4, ya se empieza a ver la diferencia, en este caso la probabilidad de autocontención en los patrones es más bien chica y ello hace que la observación clave de KMP no sea de tanta ayuda como lo puede ser en un alfabeto binario. Por ello, Boyer Moore Horspool sale victorioso en este experimento, pues al aprovechar que **es más probable que la letra de la discrepancia no esté en el patrón**, se hace una menor cantidad de comparaciones, aprovechando los saltos del algoritmo. Esto se ve claramente en las comparaciones, cuando el patrón es más chico (el peor caso), el algoritmo aún así hace sólo 600.000 comparaciones.

Una situación más drástica es el caso de los lenguajes. Aquí el alfabeto aumenta drásticamente de tamaño, disminuyendo considerablemente la probabilidad de autocontención de substrings del patrón en él mismo, que a su vez implica aumento de la probabilidad de la no existencia de la letra discrepante en el patrón. Esto disminuye en número total de comparaciones en Boyer Moore y también en KMP y Fuerza Bruta, en estos casos, se ve que KMP (por sólo un poco) sigue siendo peor que fuerza bruta, pues el tiempo del precálculo de la función de fracaso sigue latente.

En resumidas cuentas, para el problema de la búsqueda en texto, el tamaño del alfabeto no es una cuestión inocente, y debe tomarse en cuenta al momento de decidir qué tipo de lenguaje usar.

4.5 Conclusiones

Se podría armar una tabla de decisión de uso de algoritmos, como la que sigue:

	Patrón Chico	Patrón Grande
Alfabeto Chico	BMH	KMP
Alfabeto Moderado	BMH	BMH
Alfabeto Grande	BMH	BMH

La tabla es certera, en general el mejor algoritmo en términos de patrón y alfabeto es BMH, que para el uso de la gente común es el caso con mayor frecuencia: búsqueda de texto en un alfabeto grande (incluso más grande que tamaño 256) y un patrón que oscila entre chico y mediano.

En todo caso, falta comparar contra el largo del texto, pero esa es una problemática que en este experimento no se aborda.

5 Apéndices

5.1 Detalles de Implementación

A continuación, se incluyen los códigos de las implementaciones de los cuatro algoritmos, con los cuidados necesarios de los índices, para la correcta ejecución de éstos y por completitud.

Fuerza Bruta

```
void fzabruta(char *patron, char *texto, struct resultado *result){
  int found=0;
  int start=0;

  char *p, *t, *t2;

  resultadoNew(result);

  for(t=texto; *t != 0; t=t+1, ++start){
    t2=t;
    for(p=patron; *p != 0; p=p+1){
      result->comparaciones++;
      if(*p == *t2){
        t2=t2+1;
          if(*(p+1) == 0){
          found = 1;
          break;
        }
      } else break;
    }
    if(found){
      found=0;
      //      resultadoAdd(result, start);
    }
  }
}
```

Knuth Morris Pratt

```
static void preprocesarPatron(char* patron, int* next){
  int j, i;

  char *p=patron;
  j=1;
  next[0] = 0;

  while(*(p+1)){
    i = next[j-1];
    while(i>0 && patron[i]!=patron[j]) i=next[i-1];
    if(patron[i] == patron[j]) next[j]=i+1;
    else next[j] = 0;
    j++;
    p++;
  }
}

void kmp(char *patron, char *texto, int textoSize, struct resultado *result){
  int k=0;
  int j=0;
  int m=strlen(patron);
  int *fracaso;

  resultadoNew(result);

  fracaso = (int*)malloc(m*sizeof(int));
  preprocesarPatron(patron, fracaso);

  while (k<textoSize){
    while (j>0 && texto[k]!=patron[j]){
      j=fracaso[j-1];
      result->comparaciones++; // cada parte del ciclo
    }
    if(j>0)result->comparaciones++; // la ultima iteracion del ciclo tb compara
    result->comparaciones++; //cada if -> 1 comparacion
    if (texto[k]==patron[j]) j++;
    k++;
    if(j==m){
      //      resultadoAdd(result, k-j);
      j=fracaso[j-1];
    }
  }

  free(fracaso);
}
```

Boyer Moore Horspool

```
void bmh(char *patron, char *texto, int textoLen, struct resultado *result){
  int patronlen, k, j, indice;
  int salto[TAMANO_ALFABETO];

  patronlen = strlen(patron);

  /* calculamos la funcion de salto */
  for(k=0; k<TAMANO_ALFABETO; k++) salto[k]=0;
  for(k=0; k < patronlen - 1; k++) salto[(int)patron[k]] = k;

  resultadoNew(result);

  if(patron == NULL || *patron == 0){
    printf("patron es null\n");
    return;
  }

  /* en adelante k recorre el patron y j el texto */
  for(k=patronlen-1, j=patronlen-1; j<textoLen;){
    indice = j + k + 1 - patronlen;
    result->comparaciones++;
    if(texto[indice] == patron[k]) k--;
    else {
      j += patronlen - salto[(int)texto[j]] - 1;
      k = patronlen - 1;
    }
    if(k==0){
      //      resultadoAdd(result, j - patronlen + 1);
      j += patronlen - salto[(int)texto[j]] - 1;
      k = patronlen - 1;
      //      return;
    }
  }
}
```

5.2 Tablas con los datos de experimentos con gráficos ambiguos

Esta tabla muestra los resultados obtenidos para comparaciones en el caso de Lenguaje Real

	Boyer Moore Horspool	Knuth Morris Pratt	Fuerza Bruta
4	393578	1086093	1082983
8	199027	1077395	1085570
16	117064	1077863	1084436
32	77822	1081104	1086665
64	60115	1083965	1089768
128	50552	1086644	1088409

Ahora, la misma tabla para Lenguaje Sintético:

	Boyer Moore Horspool	Knuth Morris Pratt	Fuerza Bruta
4	346587	1019563	1019569
8	154343	1019590	1019575
16	77663	1019661	1019662
32	43325	1019539	1019612
64	27786	1019672	1019581
128	21483	1019677	1019707

Como se puede ver, la cantidad de comparaciones entre Fuerza Bruta y Knuth Morris Pratt en ambos casos difieren en el orden de decenas, una cantidad prácticamente despreciable para los gráficos.

5.3 Cálculo estadístico de valores

Para asegurar que las mediciones sean verídicas, se ha propuesto en este experimento calcular todos los valores aquí presentados como el promedio de muchas mediciones tales que los valores entregados corresponden al promedio real con una confianza del 95%. Para ello lo que se hace es calcular los estimadores (insesgados) de promedio y desviación estándar de los tiempos (o comparaciones) obtenidos y calcular el error asociado al promedio. Se repite estos experimentos de tal modo que la desviación estándar del promedio sea tan baja como se requiera (en este caso se ha elegido el 5% del promedio).

$$
\begin{aligned}
\mu &= \frac{\sum_{i=1}^{n} X_i}{n} \\
\sigma &= \sqrt{\frac{\sum_{i=1}^{n} (X_i - \mu)^2}{n-1}} \\
\sigma &= \sqrt{\frac{\sum_{i=1}^{n} X_i^2 - n\mu^2}{n-1}} \\
\varepsilon &= \frac{\sigma}{\sqrt{n}}
\end{aligned}
\tag{1}
$$

Estas ecuaciones definen cómo se debe calcular el promedio μ, la desviación estándar σ y el error ε con las muestras X_i tomadas. En estos experimentos se ha definido como rango aceptable la condición

$$\varepsilon < 0.05 \cdot \mu$$

Para asegurar que los datos que se entregan sean verídicos y correspondan a un promedio aceptablemente real de la ejecución de los algoritmos.

5.4 Cálculo de error con valores de gran tamaño

El último apéndice explica cómo se ha calculado la varianza (y por lo tanto desviación estándar) cuando los valores entregados por las muestras son muy grandes. Por ejemplo, en el caso de las comparaciones es todo un problema, porque las comparaciones son números grandes, que en el caso de `C` hacen *overflow* con los datos, incluso con tipos extremos como lo son `unsigned long long`. Para ello se ha utilizado la definición normal de desviación estándar:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \mu)^2}{n-1}}$$

Para esto se ha creado una estructura de datos simple, que almacena los valores en un arreglo, y este arreglo se duplica según necesidad. Se puede consultar el archivo `resultados.c` para ver la implementación de este.

5.5 Ambiente de Ejecución de los Experimentos

Por cuestiones de medición de tiempo y de aseguramiento de la calidad de este informe, se incluye el ambiente donde fueron ejecutados los experimentos.

Sistema Operativo	Ubuntu 14.04 LTS
Procesador	Intel Core i5 CPU 750 2.67GHz x 4
Memoria Disponible	3.8 GiB
Tipo de Sistema	64bit.

Este tipo de sistema se encuentra en el laboratorio del Departamento de Ciencias de la Computación ubicado en el segundo piso del edificio poniente, Beauchef # 851.

5.6 Sobre la medición de comparaciones aparte

Como se puede ver en el apéndice de los detalles de implementación, en este experimento se ha combinado la búsqueda de las ocurrencias (que se ha comentado, para medir el tiempo efectivo) y la medición de comparaciones.

Un análisis purista hubiese separado la medición de estos dos parámetros, pues el almacenar un parámetro deseado genera *ruido* en la medición del otro en el sentido de que al medir las ocurrencias, si además vienen "gratis" las comparaciones, entonces la medición del tiempo de las ocurrencias tiene un error asociado con el tiempo de medición de comparaciones.

Con esto claro, en este experimento se ha decidido tomar el enfoque no purista por razones de tiempo, espacio de implementación, carga académica externa al ramo, y **sobre todo**, porque el tiempo promedio que arrojó el enfoque que se tomó, es demasiado bajo como para considerar ruido en las mediciones.