

Archivo de la etiqueta: string matching

Boyer-Moore-Horspool

julio 23, 2013 Algoritmos boyer-moore, string, string matching

Un problema muy común es, dado un texto, encontrar la posición de una palabra dada en el texto. Existen varios algoritmos que resuelven este problema de una forma eficiente, aunque la eficiencia de cada algoritmo esta condicionada por varios factores del texto, como el tamaño del alfabeto, el largo del texto, y del patrón, etc. Uno de los algoritmos mas famosos para resolver este problema es el algoritmo de Boyer-Moore. Este algoritmo fue modificado después por **Nigel Horspool**, con el objetivo de crear un algoritmo que fuera rápido al buscar en textos de alfabetos naturales (lenguajes humanos) y a la vez sencillo de entender e implementar. El algoritmo fue publicado en 1980 con el nombre de Boyer-Moore-Horspool.

El algoritmo esta basado en dos ideas fundamentales:

1. Buscar de derecha a izquierda
2. Calcular los saltos que se pueden hacer para avanzar mas en la búsqueda

Buscar la aguja en el pajar


Tenemos el texto 'TEXTOGENERADOALEATORIAMENTE', y queremos encontrar la posicion de 'EATOR' dentro del texto. A continuacion, nos referiremos a 'EATOR' como el *patron*, y a 'TEXTOGENERADOALEATORIAMENTE' como el *texto*.

1. Alineamos el patrón con el texto

TEXT**O**GENERADOALEATORIAMENTE
EATOR**R**
▲


2. Comparamos de derecha a izquierda los caracteres del patrón con los del texto. En el patrón tenemos 'R' y en el texto 'O'. En este caso, el algoritmo común movería a ciegas una posición a la derecha el patrón, en cambio, nuestro algoritmo trata de alinear el caracter del texto que esta alineado *con el ultimo caracter del patron*, con su posición mas a la derecha en el patrón.

TEXT**O**GENERADOALEATORIAMENTE
EAT**O**R



3. De esta forma se alinean las dos **O**, y se vuelve a comparar comenzando por la derecha. Al comparar '**R**' con '**G**', el algoritmo trata de alinear '**G**' con su posición mas a la derecha en el patrón, pero dado que '**G**' no aparece en el patrón, podemos correctamente alinear el patrón hasta despues de la '**G**' en el texto, ya que el patron no puede aparecer antes de esa posicion, porque no contiene la '**G**'.

TEXT**O**GENER**A**DOALEATORIAMENTE
G**E**AT**O**R



4. Ahora al comparar '**R**' con '**A**', alineamos la '**A**' del texto con la del patrón, y se vuelve a comparar de derecha a izquierda

TEXT**O**GENER**A**DO**A**LEATORIAMENTE
G**E**AT**O**R



5. Sucede lo mismo que en el caso anterior, volvemos a alinear la '**A**' con su posición en el patrón

TEXT**O**GENERADO**A**LE**A**ATORIAMENTE
G**E**AT**O**R



6. Idem

TEXT**O**GENERADOALE**A**T**O**R**I**AMENTE
G**E**AT**O**R



7. En este caso, el algoritmo continua comparando de derecha a izquierda hasta encontrar el patrón completo en el texto. Aquí tenemos una aparición.

TEXTOGENERADOALEATORIAMENTE
EATOR
▲

8. En este punto el algoritmo puede terminar concluyendo que se encontró el texto en la posición dada, o continuar a buscar todas las posiciones donde aparece el patrón dentro del texto. En este caso, alineáramos la 'R' del texto a su próxima posición en el patrón, como no aparece mas, podemos mover el patrón pasada la 'R' del texto.

TEXTOGENERADOALEATORIAMENTE
REATOR
▲

9. En este punto, al comparar 'N' con 'R' notamos que 'N' no aparece en el patrón, y al mover el patrón pasado 'N', el algoritmo termina, ya que no hay suficientes caracteres en el texto para seguir comparando.

TEXTOGENERADOALEATORIAMENTE
NEATOR
▲

Precalcular la aguja

Para ejecutar el algoritmo, necesitamos definir la función R , que dado un caracter c del texto, $R(c)$ nos indica la cantidad de posiciones que debemos mover el patrón para alinear la posición mas a la derecha de c en el patrón con la posición actual de c en el texto. Esta función se puede precalcular para cada uno de los caracteres del texto, y luego usarla en el algoritmo de forma constante.

Con esta función definida, el algoritmo alinea a cada paso el patrón con la posición actual en el texto. En caso de una coincidencia, se continua explorando el patrón tratando de encontrar mas coincidencias, en caso de encontrar dos caracteres diferentes, el algoritmo usa la función $R(c)$ en el caracter del texto que esta siendo alineado con el ultimo caracter del patrón para calcular la nueva alineación del patrón. La implementacion quedaria asi

```

1 public int[] preBoyerMooreHorspool(char[] P, char[] T, int size)
2 {
3     int pLength = P.length;
4     int last = P.length - 1;
5
6     // precalcular R
7     int[] R = new int[size];
8     for(int i = 0; i < size; i++) R[i] = pLength;
9     for(int i = 0; i < last; i++) R[P[i]] = last - i;
10

```

```

11 |     return R;
12 | }

```

Esta función inicialmente asigna a cada letra del alfabeto (*size* es el tamaño del alfabeto) un salto del tamaño del patrón. Esto sucede en el caso en que la letra no aparece en el patrón. Luego, a las letras que están en el patrón (excepto la última) se le asigna un salto del tamaño de su posición más a la derecha.

El algoritmo luego va a usar la tabla para ir alineando el patrón con el texto

```

1 | public int boyerMooreHorspool(String pattern, String text)
2 | {
3 |     char[] P = pattern.toCharArray();
4 |     char[] T = text.toCharArray();
5 |     int pLength = P.length;
6 |     int offset = 0;
7 |     int scan = 0;
8 |     int last = P.length - 1;
9 |     int maxoffset = T.length - P.length;
10 |
11 |     // precalcular R
12 |     int[] R = preBoyerMooreHorspool(P, T, 256);
13 |
14 |     // aun hay suficientes caracteres para comparar
15 |     while(offset <= maxoffset)
16 |     {
17 |         // comparar de derecha a izquierda
18 |         for(scan = last; P[scan] == T[scan+offset]; scan--)
19 |             if(scan == 0) return offset; // encontrado
20 |
21 |         offset += R[T[offset + last]]; // alinear el patron
22 |     }
23 |
24 |     return -1; // no encontrado
25 | }

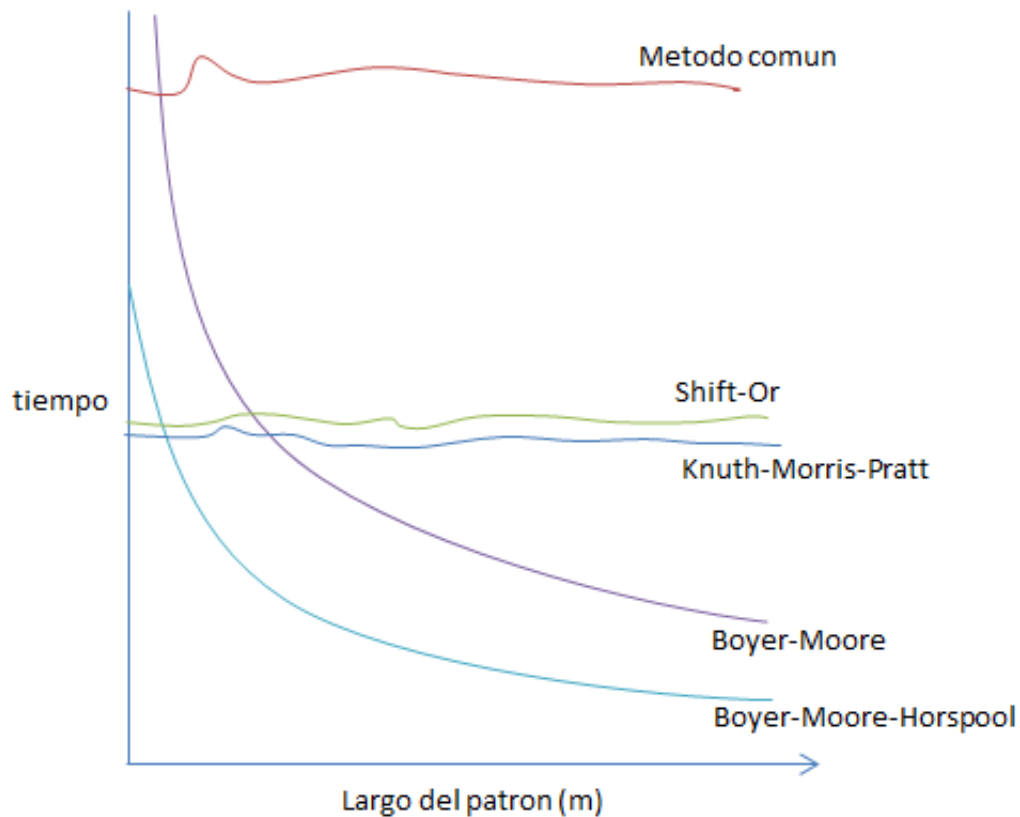
```

En esta implementación se usa 256 para el tamaño del alfabeto. Esto se puede modificar para tamaños menores del texto, pero con la observación de que este algoritmo es particularmente eficiente para alfabetos grandes y patrones largos. Para alfabetos pequeños (como el de ADN o el binario) es mas eficiente usar la versión completa de Boyer-Moore.

Análisis

El algoritmo tiene un caso peor acotado por $O(nm)$ donde n es el tamaño del texto y m del patrón. En la practica sin embargo – y especialmente con alfabetos de tamaño grande – este caso nunca ocurre. De hecho, el algoritmo se comporta en orden sublineal, ya que en la mayoría de las ocasiones los saltos que se producen en el algoritmo conllevan a que se produzcan pocas comparaciones. El análisis de este algoritmo es sumamente complejo, pero no es difícil notar que en el mejor caso, se realizan solamente $O(n/m)$ comparaciones.

La siguiente gráfica muestra como se comporta el algoritmo comparado con otros algoritmos conocidos que resuelven el mismo problema:



Referencias

http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm

<http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/Matching-Boyer-Moore2.html>

<http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/horsen.htm>

<http://www-igm.univ-mlv.fr/~lecroq/string/node18.html>

<http://www.cs.ucdavis.edu/~gusfield/cs224f09/bnotes.pdf>

Problemas

<http://codeforces.com/problemset/problem/318/B>