Algoritmo de Boyer-Moore para comparar cadenas

Daniel Valverde Menasalvas

14 de marzo de 2019

1. Introducción

El algoritmo de Boyer-Moore para buscar cadenas de caracteres es un algoritmo que se usa como punto de referencia a la hora de resolver problemas prácticos de búsqueda de cadenas de caracteres en textos. Fue desarrollado por Robert S. Boyer y J Strother Moore en 1977.

El algoritmo preprocesa el patrón(cadena de caracteres) buscado, pero no el texto sobre el que se va a realizar la búsqueda. Por lo tanto, este algoritmo funciona bien en ocasiones donde o bien el texto es considerablemente más largo que el patrón o bien el patrón sea el mismo para varios textos. Como veremos más adelante, el algoritmo funcionará mejor conforme la longitud del patrón buscado incrementa. La información obtenida en el preprocesamiento es usada por el algoritmo para saltar secciones del texto, consiguiendo así una eficiencia mayor que otros algoritmos con el mismo fin.

1.1. Motivación para el estudio del algoritmo

Supongamos que x es un string de longitud m y queremos obtener la posición i donde empieza la primera aparición de x en un texto de longitud n que denotaremos por y.

El algoritmo obvio mira cada posición de y y comprueba si los siguientes m caracteres coinciden con los caracteres de x. Sin embargo, este algoritmo es cuadrático, es decir, en el peor de los casos el número de comparaciones es $O(n * m)$.Otra opción es preprocesar x en tiempo lineal a m para luego buscar en y inspeccionando cada carácter de y exactamente una vez. Lo que resultaría en un algoritmo lineal en n.

El algoritmo de Boyer-Moore es 'normalmente sublineal'. Con normalmente sublinear nos referimos a que el valor esperado de caracteres de y examinados es $c * n$, donde $c < 1$ y es más pequeño conforme el tamaño de m incrementa. Hay

patrones y textos para los que el algoritmo muestra peores comportamientos, pero es posible demostrar que el algoritmo es lineal en el caso peor.

A continuación se dará una descripción no formal del algoritmo a estudiar y se mostrará un ejemplo de su funcionamiento. Luego definiremos formalmente el algoritmo y daremos una implementación de este en C++. Después mostraremos los resultados de un test de ejecución del algoritmo, viendo como se comporta para diferentes alfabetos y longitudes de x. Finalmente, hablaremos del rendimiento de este algoritmo y de las variantes a las que ha dado lugar.

2. Descripción no formal

Para simplificar la explicación del algoritmo veremos x e y como vectores de caracteres. El primer carácter de x será $x[0]$ y el último $x[m-1]$.

El algoritmo comparará los caracteres del patrón de derecha a izquierda empezando por el que está más a la derecha. De esta manera el algoritmo empezará comparando el carácter $x[m-1]$ con el carácter $y[m-1]$. Si se encuentran dos carácteres diferentes, o ambas cadenas coinciden, se usarán dos funciones para mover el patrón a la derecha respecto al texto. Estas funciones son conocidas por los nombres **good-suffix shift** (movimiento del buen sufijo) y **bad-character shift** (movimiento del mal caracter).

Supongamos que $x[0]$ está alineado con $y[j]$, y encontramos el primer carácter diferente (buscando desde la derecha) en la posición $x[i] = a \neq y[j+i] = b$. Por lo tanto, $x[i+1 \ \ldots \ m-1] = y[j+i+1 \ \ldots \ j+m-1] = u$. El *good-suffix shift* consiste en alinear el segmento $y[j+i+1 \ldots j+m-1]$ con otra aparición suya en x que cumpla que el carácter que le precede es diferente a $x[i] = a$. Si hay varias, cogemos la que se encuentre más a la derecha (ver figura 1).

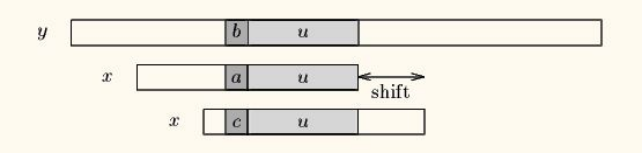


Figura 1: Good-suffix shift, u vuelve a aparecer en x precedido por un carácter $c \neq a$

Si no existe ese tipo de segmento, el movimiento consistirá en alinear el mayor sufijo v de $y[j+i+1 \ \ldots \ j+m-1]$ con un prefijo coincidente de x (ver figura 2).

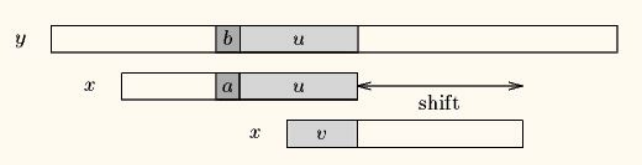


Figura 2: Good-suffix shift, un sufijo de u reaparece al principo de x

El *bad-character shift* consiste en alinear el carácter $y[i+j]$ con su aparición más a la derecha en $x[1 \ \ldots \ m-2]$ (ver figura 3).

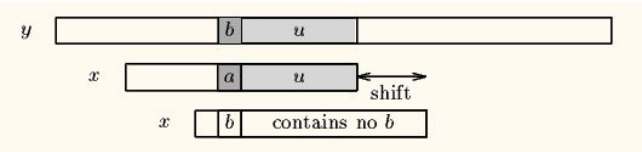


Figura 3: Bad-character shift, b aparece en x

Si $y[i+j]$ no aparece en x, no habrá alineación posible de x con $y[i+j]$, por lo que alinearemos $x[0]$ con $y[i+j+1]$(ver figura 4).

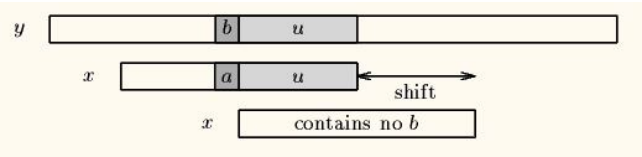


Figura 4: Bad-character shift, b no aparece en x

Es interesante darse cuenta de que el *bad-character shift* puede ser negativo, por lo tanto, el algoritmo elegirá el máximo valor arrojado por el *good-suffix shift* y el *bad-character shift* a la hora de hacer avanzar x respecto a y.

2.1. Ejemplo del algoritmo de Boyer-Moore

A continuación, se mostrará gráficamente el funcionamiento del algoritmo para un ejemplo concreto buscando un patrón en un texto en inglés. En cada paso, marcaremos con negrita el carácter en el que el algoritmo centra su atención.

$$
\begin{array}{ll}
x: & AT-THA\mathbf{T} \\
y: ... & WHICH-\mathbf{F}INALLY-HALTS.--AT-THAT-POINT...
\end{array}
$$

Como 'F' no aparece en x, usamos un *bad-character-shift* (ver figura 4) y desplazamos x:

$$
\begin{array}{ll}
x: & AT-THA\ \mathbf{T} \\
y: ... & WHICH-FINA\ LL\ Y\ \mathbf{-}HALTS.--AT-THAT-POINT\ ...
\end{array}
$$

Ahora usamos otro *bad-character-shift* (ver figura 3) y desplazamos x para alinear el espacio (-) de y con el primer espacio (desde la derecha) de x:

$$
\begin{array}{ll}
x: & AT-THA\mathbf{T} \\
y: ... & WHICH-FINALLY-HAL\mathbf{T}S.--AT-THAT-POINT...
\end{array}
$$

Observamos que los caracteres visitados coinciden, por lo que movemos nuestra atención un carácter hacia la izquierda:

$$
\begin{array}{ll}
x: & AT-TH\mathbf{A}T \\
y: ... & WHICH-FINALLY-HA\mathbf{L}TS.--AT-THAT-POINT...
\end{array}
$$

Hacemos un *bad-character-shift* (ver figura 4) y, como 'L' no aparece en x, avanzamos m posiciones:

$$
\begin{array}{ll}
x: & AT-THA\mathbf{T} \\
y: ... & WHICH-FINALLY-HALTS.\ \ --A\mathbf{T}-THAT-POINT...
\end{array}
$$

Ahora vemos que son los dos últimos caracteres de x los que coinciden con y, por lo que movemos nuestra atención 2 caracteres a la izquierda:

$$
\begin{array}{ll}
x: & \qquad\qquad\qquad\qquad AT - T\mathbf{H}AT \\
y: ... & WHICH - FINALLY - HALTS. \quad - \mathbf{-}AT - THAT - POINT...
\end{array}
$$

Finalmente, usamos un *good-suffix-shift* (ver figura 2) de forma que encontramos una aparición de x en y:

$$
\begin{array}{ll}
x: & \qquad\qquad\qquad\qquad\qquad\qquad AT - THA\mathbf{T} \\
y: ... & WHICH - FINALLY - HALTS. - - AT - THA\mathbf{T} - POINT...
\end{array}
$$

Hemos de notar que solo hemos referenciado y 14 veces, de las cuales 7 han sido para realizar la última comparación. Las otras 7 nos han permitido pasar los primeros 22 caracteres de y.

3. El algoritmo

A continuación concretaremos el algoritmo usando la notación de la sección 2. Para optimizar la implementación del algoritmo, se realiza una fase de preprocesamiento en la que calculamos el valor que devolvería un *bad-character-shift* y un *good-suffix-shift* en cualquier situación. Para ello se construyen dos tablas que concretaremos a continuación:

3.1. Tabla del buen sufijo (GST)

La primera será denominada la tabla del buen sufijo (GST). Para cada i menor a m, construiremos el patrón consistente en los últimos i caracteres de x precedido por un carácter no coincidente con x. Es decir, construimos el patrón $c + x[m-1-i \quad ... \quad m-1]$ donde $c \neq x[m-2-i]$ y $+$ representa la concatenación de caracteres. Ahora situaremos el patrón sobre x y anotaremos el número mínimo de posiciones que hemos de mover el patrón a la izquierda para encontrar una coincidencia con x.

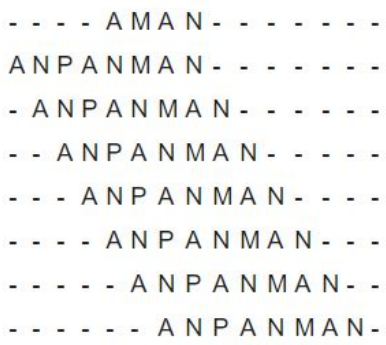


Figura 5: Tenemos $x[4] = N \neq A$, lo que nos obliga a desplazar x seis caracteres a la izquierda.

Para la búsqueda de la cadena 'ANPANMAN', la tabla sería la siguiente ('~~N~~MAN' denotará a una cadena compuesta por un carácter diferente de 'N' más los carácteres 'MAN'):

i	Patrón	Desp	Explicación
0	~~N~~	1	El penúltimo carácter de x es diferente a 'N', con lo que basta desplazar una posición.
1	~~A~~N	8	AN no es cadena en x por lo que desplazaremos $m = 8$.
2	~~M~~AN	3	~~M~~AN coincide con AN**PAN**MAN, por lo que desplazamos 3 posiciones.
3	~~N~~MAN	6	~~N~~MAN no es subcadena de **ANPA**NMAN, pero el final de ~~N~~MAN coincide con el inicio de x (~~NM~~**AN**PANMAN) por lo que desplazamos 6 posiciones.
4	~~A~~NMAN	6	Igual que $i = 3$
5	~~P~~ANMAN	6	Igual que $i = 3$
6	~~N~~PANMAN	6	Igual que $i = 3$
7	~~A~~NPANMAN	6	Igual que $i = 3$

El algoritmo originalmente publicado por Boyer y Moore (ver [1]) usa una tabla más simple en la que no se requiere una no-coincidencia para el carácter de más a la izquierda. Sin embargo, esto no es suficiente para conseguir que el algoritmo funcione el tiempo lineal en el peor caso.

3.2. Tabla del mal carácter (BCT)

Para calcular esta tabla recorreremos x de derecha a izquierda, empezando por el penúltimo carácter y pasando por todos. Si el carácter en el que nos encontramos no está aún en la tabla, lo añadimos con un desplazamiento asociado igual a su distancia hasta el último carácter. Es decir, si estamos en

el carácter $x[i]$, y este aún no figura en la tabla, su desplazamiento asociado será $m - 1 - i$. A los caracteres que no figuren en x se les asignará m. Para la cadena 'ANPANMAN', la tabla quedaría de esta manera (por claridad, las entradas están ordenadas por orden de inserción):

Carácter	**Desplazamiento**
A	1
M	2
N	3
P	5
Caracteres restantes	8

3.3. Pseudocódigo

A continuación se muestra mediante pseudocódigo el funcionamiento del algoritmo de Boyer-Moore (asumimos que tenemos las tablas GST y BCT ya rellenas):

Algorithm 1 Booyer-Moore Algorithm(x, y)

1: $j = 0$
2: **while** $j \leq n - m$ **do**
3: **if** $x[0 \ \ldots \ m-1] == y[j \ \ldots \ j+m-1]$ **then**
4: Hay una coincidencia en j, la tratamos como corresponda.
5: $j+ = GST[0]$
6: **else**
7: $i =$ Posición en x del primer carácter, contando desde la derecha, en el que difieran x e y ($x[i] \neq y[j+i]$).
8: $j+ = max(GST[i], BCT[y[i+j]] - (m-1-i))$
9: **end if**
10: **end while**

4. Implementación

Se muestra a continuación una implementación del algoritmo de Boyer-Moore en C++:

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

/*
```

```
/*
Funcion que relleana BCT.
Usamos mapa no ordenado para guardar la BCT,
pues un vector del tamanio del alfabeto es una perdida
de espacio para alfabetos grandes.
*/
unordered_map<char,int> preBCT(string const &x) {
        unordered_map<char, int> BCT;

        for (int i = x.size() - 2; i > 0; i--)
                if (BCT.count(x[i]) == 0)
                        BCT[x[i]] = x.size() - 1 - i;

        return BCT;
}

/*
Funcion auxiliar para calcular GST
suff[i] nos da la longitud del sufijo mas largo coincidente
entre el string x y el string x[0 ... i]
*/
vector<int> suffixes(string const &x) {
        int f, g;
        vector<int> suff(x.size());
        suff[x.size() - 1] = x.size();
        g = x.size() - 1;
        for (int i = x.size() - 2; i >= 0; --i) {
                if (i > g && suff[i + x.size() - 1 - f] < i - g)
                        suff[i] = suff[i + x.size() - 1 - f];
                else {
                        if (i < g)
                                g = i;
                        f = i;
                        while (g >= 0 && x[g] == x[g + x.size() - 1 - f])
                                --g;
                        suff[i] = f - g;
                }
        }
        return suff;
}

//Funcion que rellena GST usando como auxiliar suffixes
vector<int> preGST(string const & x) {
        int i, j;
        vector<int> suff = suffixes(x);
        vector<int> GST(x.size());
```

```
        for (i = 0; i < x.size(); ++i)
                GST[i] = x.size();

        j = 0;
        for (i = x.size() - 1; i >= 0; --i)
                //Si x[0 ... i] == x[x.size() - 1 - i ... x.size - 1]
                if (suff[i] == i + 1)
                        for (; j < x.size() - 1 - i; ++j)
                                if (GST[j] == x.size())
                                        GST[j] = x.size() - 1 - i;
        for (i = 0; i <= x.size() - 2; ++i)
                GST[x.size() - 1 - suff[i]] = x.size() - 1 - i;

        return GST;
}

//Algoritmo de Boyer-Moore usando las funciones de preprocesamiento
void BM(string const & x, string const & y) {

        //Preprocesamiento
        vector<int> GST = preGST(x);
        unordered_map<char,int> BCT = preBCT(x);

        //Busqueda
        int j = 0, i;
        while (j <= y.size() - x.size()) {
                for (i = x.size() - 1; i >= 0 && x[i] == y[i + j]; --i);
                if (i < 0) {
                        cout << j << " ";
                        j += GST[0];
                }
                else {
                        int bcs = x.size();
                        if (BCT.count(y[i + j]) != 0)
                                bcs = BCT[y[i + j]];
                        j += max(GST[i], int(bcs - x.size() + 1 + i));
                }
        }
}
```

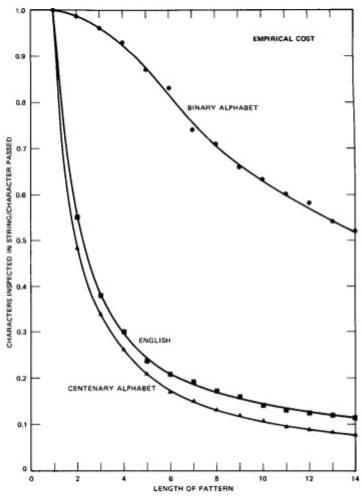
5. Evidencia empírica

En el artículo original publicado por Boyer y Moore (ver [1]) se presentaban los resultados de un test de eficiencia del algoritmo. Se ha considerado pertinente mostrar esos resultados aquí para comprobar de forma empírica la

eficiencia del algoritmo.

El test selecciona aleatoriamente subpatrones de una secuencia base de caracteres. Se seleccionaron 300 subpatrones para cada longitud de subpatrón entre 1 y 14. Después se utilizaba el algoritmo de Boyer-Moore para buscar coincidencias, empezando desde una posición aleatoria de la primera mitad de la secuencia base. Se midió la eficiencia dividiendo el número de referencias a y entre el número de caracteres de y pasados durante el tiempo de ejecución del algoritmo. Esta medida es independiente de la implementación del algoritmo. Después se realizaría una media aritmética entre las 300 muestras de cada patrón.

Como la eficacia del algoritmo dependía de las propiedades estadísticas de x e y, se realizó el test para tres textos distintos de 10000 caracteres cada uno. Uno era una secuencia aleatoria de 0's y 1's, otro era un texto en inglés obtenido de un manual online y el tercero era una secuencia aleatoria de caracteres de un diccionario de 100 caracteres.



Los resultados del test muestran que la eficacia del algoritmo incrementa, para los tres textos, conforme aumenta la longitud del patrón buscado. Vemos que, a partir de los seis caracteres de longitud en los textos no binarios, el algoritmo solo referencia de media uno de cada cinco caracteres de y. Además, el ratio caracteres inspeccionados/caracteres pasados siempre es menor a uno (si x consta de más de un carácter), es decir, el algoritmo mejora el coste lineal en estos casos.

6. Rendimiento del algoritmo

El problema del rendimiento del algoritmo no quedó zanjado en el artículo original publicado por Boyer y Moore (ver [1]). Ellos mostraron que el número de comparaciones no era mayor a $6n$, siendo n el tamaño del texto donde el patrón es buscado. Más tarde, en 1980, se demostró que no era más de $4n$. Finalmente, en septiembre de 1991 Cole publicó un artículo (ver [2]) en el que mostraba que, en el caso peor, el algoritmo necesita aproximadamente $3n$ comparaciones para encontrar todas las coincidencias, independientemente de si hay alguna o no.

Las demostraciones de estos resultados son largas y tediosas, por lo que se ha decidido no incluirlas aquí. Sin embargo, los artículos originales están referenciados al final del texto para que el lector pueda indagar más por su cuenta si así lo desea.

7. Variantes

A pesar de la eficiencia del algoritmo de Boyer-Moore, hay situaciones en las que es preferible utilizar variantes de este, ya sea para mejorar el rendimiento o para simplificar el código. A continuación se mostrarán dos de estas variantes:

- **Algoritmo de Boyer-Moore Turbo**: Esta variante toma una cantidad adicional constante de espacio para completar una búsqueda en $2n$ comparaciones, mejorando el $3n$ del algoritmo normal.
- **Algoritmo de Boter-Moore-Horspool**: Es una simplificación del algoritmo que omite la GST. Con esta simplificación se requieren, en el caso peor, nm comparaciones.

Referencias

[1] Boyer, R.S. y Moore, J.S. (1977). A Fast String Searching Algorithm. *Communications of the ACM,20*(10), 762-772.

[2] Cole, R. (1991). Tight bounds on the complexity of the Boyer-Moore algorithm. *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*. 224-233.

[3] Algoritmo de búsqueda de cadenas Boyer-Moore. Recuperado de: `https://es.wikipedia.org/wiki/Algoritmo_de_b\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{u\global\mathchardef\accent@spacefactor\spacefactor}\accent19u\egroup\spacefactor\accent@spacefactor\futurelet\@let@token\penalty\@M\hskip\z@skipsqueda_de_cadenas_Boyer-Moore#Rendimiento_`

del_algoritmo_de_b\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\
setbox\@tempboxa\hbox{u\global\mathchardef\accent@spacefactor\
spacefactor}\accent19u\egroup\spacefactor\accent@spacefactor\
futurelet\@let@token\penalty\@M\hskip\z@skipsqueda_de_cadenas_
Boyer-Moore

[4] Boyer-Moore algorithm. Recuperado de: http://www-igm.univ-mlv.fr/
~lecroq/string/node14.html