Algoritmo de búsqueda de cadenas Boyer-Moore

De Wikipedia, la enciclopedia libre

El **algoritmo de búsqueda de cadenas Boyer-Moore** es un particularmente eficiente algoritmo de búsqueda de cadenas, y ha sido el punto de referencia estándar para la literatura de búsqueda de cadenas práctica.[1] Fue desarrollado por Bob Boyer y J Strother Moore en 1977. El algoritmo preprocesa la cadena objetivo (clave) *que* está siendo buscada, pero no *en* la cadena en que se busca (no como algunos algoritmos que procesan la cadena en que se busca y pueden entonces amortizar el coste del preprocesamiento mediante búsqueda repetida). El tiempo de ejecución del algoritmo Boyer-Moore, aunque es lineal en el tamaño de la cadena siendo buscada, puede tener un factor significativamente más bajo que muchos otros algoritmos de búsqueda: no necesita comprobar cada carácter de la cadena que es buscada, puesto que salta algunos de ellos. Generalmente el algoritmo es más rápido cuanto más grande es la clave que es buscada, usa la información conseguida desde un intento para descartar tantas posiciones del texto como sean posibles en donde la cadena no coincida.

Índice

Cómo funciona el algoritmo

A la gente frecuentemente le sorprende el algoritmo de Boyer-Moore, cuando lo conoce, porque en su verificación intenta comprobar si hay una coincidencia en una posición particular marchando hacia atrás. Comienza una búsqueda al principio de un texto para la palabra "ANPANMAN", por ejemplo, comprueba que la posición octava del texto en proceso contenga una "N". Si encuentra la "N", se mueve a la séptima posición para ver si contiene la última "A" de la palabra, y así sucesivamente hasta que comprueba la primera posición del texto para una "A".

```
- - - - - - - X - - - - - - -
A N P A N M A N - - - - - - -
- A N P A N M A N - - - - - -
- - A N P A N M A N - - - - -
- - - A N P A N M A N - - - -
- - - - A N P A N M A N - - -
- - - - - A N P A N M A N - -
- - - - - - A N P A N M A N -
- - - - - - - A N P A N M A N
```

La X en la posición 8 excluye todas la 8 posibles posiciones de comienzo mostradas.

La razón por la que Boyer-Moore elige este enfoque está más clara cuando consideramos que pasa si la verficación falla-por ejemplo, si en lugar de una "N" en la octava posición, encontramos una "X". La "X" no aparece en "ANPANMAN", y esto significa que no hay coincidencia para la cadena buscada en el inicio del texto-o en las siguientes siete posiciones, puesto que todas fallarían también con la "X". Después de comprobar los ocho caracteres de la palabra "ANPANMAN" para tan sólo un carácter "X", seremos capaces de saltar hacia delante y comenzar buscando una coincidencia en el final en la 16.ª posición del texto.

Esto explica por qué el rendimiento del caso promedio del algoritmo, para un texto de longitud **n** y patrón fijo de

Esto explica por qué el rendimiento del caso promedio del algoritmo, para un texto de longitud n y patrón fijo de longitud m, es n/m: en el mejor caso, solo uno en m caracteres necesita ser comprobado. Esto también explica el resultado algo contra-intuitivo de que cuanto más largo es el patrón que estamos buscando, el algoritmo suele ser más rápido para encontrarlo.

El algoritmo precalcula dos tablas para procesar la información que obtiene en cada verficiación fallada: una tabla calcula cuantas posiciones hay por delante en la siguiente búsqueda basada en el valor del carácter que no coincide; la otra hace un cálculo similar basado en cuantos caracteres coincidieron satisfactoriamente antes del intento de coincidencia fallado. (Puesto que estas dos tablas devuelven resultados indicando cuán lejos "saltar" hacia delante, son llamada en ocasiones "tablas de salto", que no deberían ser confundidas con el significado más común de tabla de saltos en ciencia de la computación.) El algoritmo se desplazará con el valor más grande de los dos valores de salto cuando no ocurra una coincidencia.

Tabla primera

Rellénese la primera tabla como sigue. Para cada *i* menor que la longitud de la cadena de búsqueda, constrúyase el patrón consistente en los últimos *i* caracteres de la cadena precedida por un carácter *no*-coincidente, alinéense a la derecha el patrón y la cadena, **y anótese el menor número de caracteres para que el patrón tenga que desplazarse a la izquierda para una coincidencia**.

Por ejemplo, para la búsqueda de la cadena ANPANMAN, la tabla sería como sigue:
(~~N~~MAN significa una subcadena en ANPANMAN consistente en un carácter que no es 'N' más los caracteres 'MAN'.)

i	Patrón	Desplazamiento a la izquierda
0	~~N~~	Es cierto que la letra siguiente a la izquierda en 'ANPANMA**N'** no es N (es A), de aquí que el patrón ~~N~~ debe desplazarse una posición a la izquierda para una coincidencia; por tanto = **1**
1	~~A~~N	~~A~~N no es una cadena en ANPANMAN, por tanto : el desplazamiento izquierdo es el número de letras en 'ANPANMAN' = **8**
2	~~M~~AN	Subcadena ~~**M**~~**AN** coincide con AN**PA**NMAN tres posiciones a la izquierda. Por tanto desplazamiento a la izquierda = **3**
3	~~N~~MAN	Vemos que '~~N~~MAN' no es una subcadena de '**ANPA**NMAN' pero '~~NM~~AN' es una posible subcadena 6 posiciones más a la izquierda : ('~~NM~~**AN**PANMAN'); por tanto = **6**
4	~~A~~NMAN	**6**
5	~~P~~ANMAN	**6**
6	~~N~~PANMAN	**6**
7	~~A~~NPANMAN	**6**

- - - - A M A N - - - - - - -
A N P A N M A N - - - - - - -
- A N P A N M A N - - - - - -
- - A N P A N M A N - - - - -
- - - A N P A N M A N - - - -
- - - - A N P A N M A N - - -
- - - - - A N P A N M A N - -
- - - - - - A N P A N M A N -

La "A" no coincidente en la posición 5 (3 atrás desde la última letra de la aguja) excluye las primeras 6 de las posibles posiciones iniciales mostradas.

La cantidad de desplazamiento calculada por la primera tabla es a veces llamada "desplazamiento de sufijo

bueno"[2] o "regla de sufijo bueno (fuerte)". El algoritmo original Boyer-Moore publicado[3] usa una más simple, más débil, versión de la regla de sufijo bueno en que cada entrada en tabla de arriba no requiere una *no*-coincidencia para el carácter de más a la izquierda. Esto es a veces llamado "regla del sufijo bueno débil" y no es suficiente para conseguir que Boyer-Moore funcione en tiempo lineal en el peor caso.

Tabla segunda

La segunda tabla es fácil de calcular: iniciése en el último carácter de la cadena vista y muévase hacia el primer carácter. Cada vez que usted se mueve a la izquierda, si el carácter sobre el que está no está ya en la tabla, añádalo; su valor de desplazamiento es la distancia desde el carácter más a la derecha. Todos los otros caracteres reciben un valor igual a la longitud de la cadena de búsqueda.

***Ejemplo**: Para la cadena ANPANMAN, la segunda tabla sería como se muestra (por claridad, las entradas son mostradas en el orden que serían añadidas a la tabla):* (La N que se supuestamente sería cero está basada en la segunda N desde la derecha porque solo anotamos el cálculo para las primeras $m - 1$ letras)

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

La cantidad de desplazamiento calculada por la segunda tabla es a veces llamada "desplazamiento de carácter malo".[2]

Rendimiento del algoritmo de búsqueda de cadenas Boyer-Moore

El caso peor para encontrar todas las coincidencias en un texto necesita aproximadamente $3n$ comparaciones, de aquí que la complejidad sea $O(n)$, a pesar de que el texto contenga una coincidencia o no.[4] Esta prueba llevó algunos años para desarrollarse. En el año en que se ideó el algoritmo, 1977, se mostró que el número máximo de comparaciones no era más de $6n$; en 1980 se demostró que no era más de $4n$, hasta el resultado de Cole en Sep 1991.

Implementación

C

```
# include <limits.h>
# include <string.h>

# define ALPHABET_SIZE (1 << CHAR_BIT)

static void compute_prefix(const char* str, size_t size, int result[size]) {
 size_t q;
 int k;
 result[0] = 0;

 k = 0;
 for (q = 1; q < size; q++) {
     while (k > 0 && str[k] != str[q])
         k = result[k-1];

     if (str[k] == str[q])
         k++;
     result[q] = k;
 }
}
```

```
static void prepare_badcharacter_heuristic(const char *str, size_t size,
 int result[ALPHABET_SIZE]) {

 size_t i;

 for (i = 0; i < ALPHABET_SIZE; i++)
     result[i] = -1;

 for (i = 0; i < size; i++)
     result[(size_t) str[i]] = i;
}

void prepare_goodsuffix_heuristic(const char *normal, size_t size,
 int result[size + 1]) {

 char *left = (char *) normal;
 char *right = left + size;
 char reversed[size+1];
 char *tmp = reversed + size;
 size_t i;

 /* reverse string */
 *tmp = 0;
 while (left < right)
     *(--tmp) = *(left++);

 int prefix_normal[size];
 int prefix_reversed[size];

 compute_prefix(normal, size, prefix_normal);
 compute_prefix(reversed, size, prefix_reversed);

 for (i = 0; i <= size; i++) {
     result[i] = size - prefix_normal[size-1];
 }

 for (i = 0; i < size; i++) {
     const int j = size - prefix_reversed[i];
     const int k = i - prefix_reversed[i]+1;

     if (result[j] > k)
         result[j] = k;
 }
}
/*
 * Boyer-Moore search algorithm
 */
const char *boyermoore_search(const char *haystack, const char *needle) {
 /*
 * Calc string sizes
 */
 size_t needle_len, haystack_len;
 needle_len = strlen(needle);
 haystack_len = strlen(haystack);

 /*
 * Simple checks
 */
 if(haystack_len == 0)
     return NULL;
 if(needle_len == 0)
     return NULL;
 if(needle_len > haystack_len)
     return NULL;
 /*
 * Initialize heuristics
 */
 int badcharacter[ALPHABET_SIZE];
 int goodsuffix[needle_len+1];

 prepare_badcharacter_heuristic(needle, needle_len, badcharacter);
 prepare_goodsuffix_heuristic(needle, needle_len, goodsuffix);

 /*
 * Boyer-Moore search
 */
 size_t s = 0;
 while(s <= (haystack_len - needle_len))
 {
     size_t j = needle_len;
     while(j > 0 && needle[j-1] == haystack[s+j-1])
         j--;
```

```
    if(j > 0)
    {
        int k = badcharacter[(size_t) haystack[s+j-1]];
        int m;
        if(k < (int)j && (m = j-k-1) > goodsuffix[j])
            s+= m;
        else
            s+= goodsuffix[j];
    }
    else
    {
        return haystack + s;
    }
}

/* not found */
return NULL;
}
```

Python

```
"""
Returns the index of the given character in the English alphabet, counting from 0.
"""
def alphabet_index(c):
    return ord(c.lower()) - 97 # 'a' is ASCII character 97

"""
Returns the length of the match of the substrings of S beginning at idx1 and idx2.
"""
def match_length(S, idx1, idx2):
    if idx1 == idx2:
        return len(S) - idx1
    match_count = 0
    while idx1 < len(S) and idx2 < len(S) and S[idx1] == S[idx2]:
        match_count += 1
        idx1 += 1
        idx2 += 1
    return match_count

"""
Returns Z, the Fundamental Preprocessing of S. Z[i] is the length of the substring
beginning at i which is also a prefix of S. This pre-processing is done in O(n) time,
where n is the length of S.
"""
def fundamental_preprocess(S):
    if len(S) == 0: # Handles case of empty string
        return []
    if len(S) == 1: # Handles case of single-character string
        return [1]
    z = [0 for x in S]
    z[0] = len(S)
    z[1] = match_length(S, 0, 1)
    for i in range(2, 1+z[1]): # Optimization from exercise 1-5
        z[i] = z[1]-i+1
    # Defines lower and upper limits of z-box
    l = 0
    r = 0
    for i in range(2+z[1], len(S)):
        if i <= r: # i falls within existing z-box
            k = i-l
            b = z[k]
            a = r-i+1
            if b < a: # b ends within existing z-box
                z[i] = b
            elif b > a: # Optimization from exercise 1-6
                z[i] = min(b, len(S)-i)
                l = i
                r = i+z[i]-1
            else: # b ends exactly at end of existing z-box
                z[i] = b+match_length(S, a, r+1)
                l = i
                r = i+z[i]-1
        else: # i does not reside within existing z-box
            z[i] = match_length(S, 0, i)
            if z[i] > 0:
                l = i
                r = i+z[i]-1
```

```
                r = i+z[i]-1
    return z

"""
Generates R for S, which is an array indexed by the position of some character c in the
English alphabet. At that index in R is an array of length |S|+1, specifying for each
index i in S (plus the index after S) the next location of character c encountered when
traversing S from right to left starting at i. This is used for a constant-time lookup
for the bad character rule in the Boyer-Moore string search algorithm, although it has
a much larger size than non-constant-time solutions.
"""
def bad_character_table(S):
    if len(S) == 0:
        return [[] for a in range(26)]
    R = [[-1] for a in range(26)]
    alpha = [-1 for a in range(26)]
    for i, c in enumerate(S):
        alpha[alphabet_index(c)] = i
        for j, a in enumerate(alpha):
            R[j].append(a)
    return R

"""
Generates L for S, an array used in the implementation of the strong good suffix rule.
L[i] = k, the largest position in S such that S[i:] (the suffix of S starting at i) matches
a suffix of S[:k] (a substring in S ending at k). Used in Boyer-Moore, L gives an amount to
shift P relative to T such that no instances of P in T are skipped and a suffix of P[:L[i]]
matches the substring of T matched by a suffix of P in the previous match attempt.
Specifically, if the mismatch took place at position i-1 in P, the shift magnitude is given
by the equation len(P) - L[i]. In the case that L[i] = -1, the full shift table is used.
Since only proper suffixes matter, L[0] = -1.
"""
def good_suffix_table(S):
    L = [-1 for c in S]
    N = fundamental_preprocess(S[::-1]) # S[::-1] reverses S
    N.reverse()
    for j in range(0, len(S)-1):
        i = len(S) - N[j]
        if i != len(S):
            L[i] = j
    return L

"""
Generates F for S, an array used in a special case of the good suffix rule in the Boyer-Moore
string search algorithm. F[i] is the length of the longest suffix of S[i:] that is also a
prefix of S. In the cases it is used, the shift magnitude of the pattern P relative to the
text T is len(P) - F[i] for a mismatch occurring at i-1.
"""
def full_shift_table(S):
    F = [0 for c in S]
    Z = fundamental_preprocess(S)
    longest = 0
    for i, zv in enumerate(reversed(Z)):
        longest = max(zv, longest) if zv == i+1 else longest
        F[-i-1] = longest
    return F

"""
Implementation of the Boyer-Moore string search algorithm. This finds all occurrences of P
in T, and incorporates numerous ways of pre-processing the pattern to determine the optimal
amount to shift the string and skip comparisons. In practice it runs in O(m) (and even
sublinear) time, where m is the length of T.
"""
def string_search(P, T):
    if len(P) == 0 or len(T) == 0 or len(T) < len(P):
        return []

    matches = []

    # Preprocessing
    R = bad_character_table(P)
    L = good_suffix_table(P)
    F = full_shift_table(P)

    k = len(P) - 1      # Represents alignment of end of P relative to T
    previous_k = -1     # Represents alignment in previous phase (Galil's rule)
    while k < len(T):
        i = len(P) - 1  # Character to compare in P
        h = k           # Character to compare in T
        while i >= 0 and h > previous_k and P[i] == T[h]:   # Matches starting from end of P
            i -= 1
            h -= 1
```

```
        if i == -1 or h == previous_k:  # Match has been found (Galil's rule)
            matches.append(k - len(P) + 1)
            k += len(P)-F[1] if len(P) > 1 else 1
        else:   # No match, shift by max of bad character and good suffix rules
            char_shift = i - R[alphabet_index(T[h])][i]
            if i+1 == len(P):   # Mismatch happened on first attempt
                suffix_shift = 1
            elif L[i+1] == -1:   # Matched suffix does not appear anywhere in P
                suffix_shift = len(P) - F[i+1]
            else:               # Matched suffix appears in P
                suffix_shift = len(P) - L[i+1]
            shift = max(char_shift, suffix_shift)
            previous_k = k if shift >= i+1 else previous_k  # Galil's rule
            k += shift
    return matches
```

Variantes

El **Algoritmo Boyer-Moore Turbo** toma una cantidad constante adicional de espacio para completar una búsqueda en $2n$ comparaciones (en contra de $3n$ para Boyer-Moore), donde n es el número de caracteres en el texto para ser buscado.[5]

El algoritmo Boyer-Moore-Horspool es una simplificación del algoritmo que omite la "tabla primera". El algoritmo Boyer-Moore-Horspool requiere (en el peor caso) mn comparaciones, mientras que el algoritmo Boyer-Moore requiere (en el peor caso) tan solo $3n$ comparaciones.

Véase también

- Algoritmo Knuth-Morris-Pratt
- Algoritmo Rabin-Karp

Referencias

1. Hume and Sunday (1991) *[Fast String Searching]* SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(11), 1221–1248 (NOVEMBER 1991)
2. http://www.movsd.com/bm.htm
3. R. S. Boyer; J. S. Moore (1977). «A fast string searching algorithm». *Comm. ACM* **20**: 762-772. doi:10.1145/359842.359859 (http://dx.doi.org/10.1145%2F359842.359859).
4. Richard Cole (1991). «Tight bounds on the complexity of the Boyer-Moore algorithm (http://portal.acm.org/citation.cfm?id=127830)». *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*. pp. 224–233.
5. http://www-igm.univ-mlv.fr/~lecroq/string/node15.html

Enlaces externos

- Applet de animación de búsqueda de cadenas (http://www.cs.pitt.edu/~kirk/cs1501/animations/String.html)
- Artículo original (http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf)
- Un ejemplo del algoritmo de Boyer-Moore (http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html) de la página hogar de J Strother Moore, co-inventor del algoritmo
- Una explicación del algoritmo (con código C de ejemplo) (http://www-igm.univ-mlv.fr/%7Elecroq/string/node14.html)
- Cole et al., Límites inferiores más estrechos sobre la complejidad exacta de la coincidencia de cadenas (http://www.cs.nyu.edu/cs/faculty/cole/papers/CHPZ95.ps)
- Una implementación del algoritmo en Ruby (http://github.com/jashmenn/boyermoore)

Categorías: Algoritmos de búsqueda | Algoritmos epónimos en matemáticas