Integrantes

Daniel Felipe Cadavid

Felipe Herrera Arteaga

Hugo Alejandro Hernández Cárdenas

**1.** Consultar alternativas para implementar la función de comparación de similaridad (considerar al menos dos alternativas). Para cada alternativa indicar:

• Seudocódigo del algoritmo

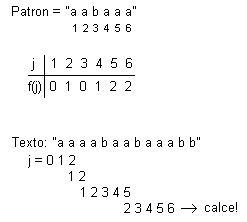
• Tiempo necesario (usar como modelo de costo la comparación de caracteres)

**Algoritmo Knuth-Morris-Pratt (KMP)**

El algoritmo KMP es un algoritmo de búsqueda de subcadenas simple y por lo tanto su objetivo es buscar la existencia de una subcadena dentro de una cadena. Para ello utiliza información basada en los fallos previos, aprovechando la información que la propia palabra a buscar contiene de sí (sobre ella se precalcula una tabla de valores), para determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca.

El algoritmo KMP trata de localizar la posición de comienzo de una cadena dentro de otra. Antes que nada con la cadena a localizar se precalcula una tabla de saltos (conocida como tabla de fallos) que después al examinar las cadenas se utiliza para hacer saltos cuando se localiza un fallo.

Ejemplo:



El tiempo de ejecución de este algoritmo no es difícil de analizar, pero es necesario ser cuidadoso al hacerlo. Dado que se tienen dos ciclos anidados, se puede acotar el tiempo de ejecución por el número de veces que se ejecuta el ciclo externo (menor o igual a n) por el número de veces que se ejecuta el ciclo interno (menor o igual a m), por lo que la cota es igual a , ¡que es igual a lo que demora el algoritmo de fuerza bruta!.

El análisis descrito es pesimista. Note que el número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede decrementar j, dado que f(j)<j. Pero j comienza desde cero y es siempre mayor o igual que cero, por lo que dicho número es menor o igual al número de veces que j es incrementado, el cual es menor que n. Por lo tanto, el tiempo total de ejecución es . Por otra parte, k nunca es decrementado, lo que implica que el algoritmo nunca se devuelve en el texto.

Codigo tomado de Algorithms, 4th edition (KMP.java).

public class KMP {

private final int R; *// the radix*

private int[][] dfa; *// the KMP automoton*

private char[] pattern; *// either the character array for the pattern*

private String pat; *// or the pattern string*

public **KMP**(String pat) {

this.R = 256;

this.pat = pat;

*// build DFA from pattern*

int m = pat.**length**();

dfa = new int[R][m];

dfa[pat.**charAt**(0)][0] = 1;

for (int x = 0, j = 1; j < m; j++) {

for (int c = 0; c < R; c++)

dfa[c][j] = dfa[c][x]; *// Copy mismatch cases.*

dfa[pat.**charAt**(j)][j] = j+1; *// Set match case.*

x = dfa[pat.**charAt**(j)][x]; *// Update restart state.*

}

}

.

.

.

public int **search**(String txt) {

*// simulate operation of DFA on text*

int m = pat.**length**();

int n = txt.**length**();

int i, j;

for (i = 0, j = 0; i < n && j < m; i++) {

j = dfa[txt.**charAt**(i)][j];

}

if (j == m) return i - m; *// found*

return n; *// not found*

}

}

**Algoritmo Boyer Moore (BM)**

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.

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.​ 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.

Codigo tomado de Algorithms, 4th edition (BoyerMoore.java).

public class BoyerMoore {

private final int R; *// the radix*

private int[] right; *// the bad-character skip array*

private char[] pattern; *// store the pattern as a character array*

private String pat; *// or as a string*

public **BoyerMoore**(String pat) {

this.R = 256;

this.pat = pat;

*// position of rightmost occurrence of c in the pattern*

right = new int[R];

for (int c = 0; c < R; c++)

right[c] = -1;

for (int j = 0; j < pat.**length**(); j++)

right[pat.**charAt**(j)] = j;

}

.

.

.

public int **search**(String txt) {

int m = pat.**length**();

int n = txt.**length**();

int skip;

for (int i = 0; i <= n - m; i += skip) {

skip = 0;

for (int j = m-1; j >= 0; j--) {

if (pat.**charAt**(j) != txt.**charAt**(i+j)) {

skip = Math.**max**(1, j - right[txt.**charAt**(i+j)]);

break;

}

}

if (skip == 0) return i; *// found*

}

return n; *// not found*

}

}

3. Estimar la complejidad algoritmica de la operación get, asumiendo una colección de N términos y uno de los algoritmos de comparación analizados en el punto 1.

**public** Bag<String> get(String termino) {

Bag<String> similares=**new** Bag<String>();

KMP patron = **new** KMP(termino.toLowerCase());

**for**(String N: terminos) {

**if**(patron.search(N)!=N.length()) similares.add(N);

}

**return** similares;

}

|  |  |  |
| --- | --- | --- |
| **Costo para una Búsqueda (hallado a partir de la consulta en el punto 1)** | **Tamaño de la Coleccion** | **Total (Complejidad Algoritmica)** |
| ~mn | ~N | ~mnN (orden de crecimiento cubico) |

**Otro algoritmo**

Bag<String> similares=**new** Bag<String>();

BoyerMoore patron = **new** BoyerMoore (termino.toLowerCase());

**for**(String N: terminos) {

**if**(patron.search(N)!=N.length()) similares.add(N);

}

**return** similares;

|  |  |  |
| --- | --- | --- |
| **Costo para una Búsqueda (hallado a partir de la consulta en el punto 1)** | **Tamaño de la Coleccion** | **Total (Complejidad Algoritmica)** |
| ~n | ~N | ~nN (orden de crecimiento cuadratico ) |

**Webgrafía**

<https://prezi.com/2avslcnt1slx/algoritmo-boyer-more/>

<https://algs4.cs.princeton.edu/53substring/BoyerMoore.java.html>

<https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/>

<https://es.wikipedia.org/wiki/Algoritmo_Knuth-Morris-Pratt>

<https://algs4.cs.princeton.edu/53substring/KMP.java.html>

<https://algs4.cs.princeton.edu/53substring/>