Teoría de Algoritmos 2

Segundo Cuatrimestre 2017

Trabajo Práctico 1

Integrante	Padrón	Correo electrónico
Rodrigo De Rosa	97799	rodrigoderosa@outlook.com
Marcos Schapira	97934	schapiramarcos@gmail.com
Facundo Guerrero	97981	facundoiguerrero@gmail.com

Índice

1. Colussi

Algoritmo de Livio Colussi 1991

Este algoritmo surge como una optimización del algoritmo de Knuth, Morris y Pratt (que a su vez es una optimización del de Morris y Pratt, que a su vez es una optimización del algoritmo ingenuo).

1.1. Funcionamiento

La idea del algoritmo es identificar *holes* y *no-holes* en el patrón para poder comparar al mismo con el texto en busca de matches es dos pasos.

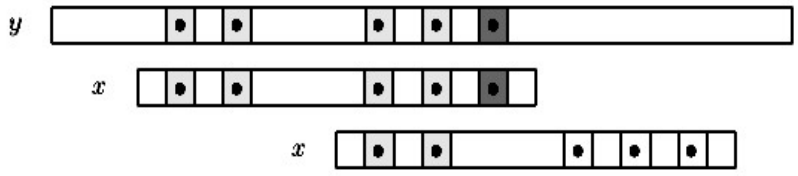
Los *holes* son aquellos cuyo valor de kmpNext (del algoritmo KMP) es -1 y los *no-holes* aquellos cuyo valor de kmpNext es distinto de -1.

Cada intento del algoritmo consiste entonces de dos pasos:

1. Se compara de izquierda a derecha, comparando sólo las posiciones que corresponden a 'no huecos' con los caracteres del texto que corresponden a sus respectivas posiciones.
2. Se compara de derecha a izquierda, comparando sóllo las posiciones que corresponden a *holes*.

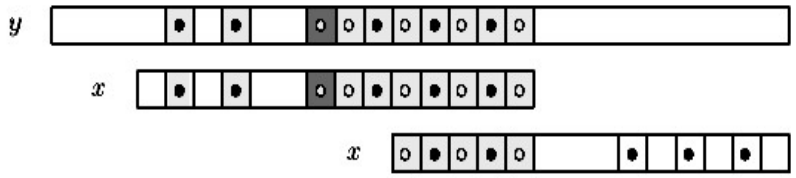
Esta estrategia tiene las siguientes ventajas:

- Si hay un mismatch en la primera fase, luego de un correcto desplazamiento, no será necesario comparar a los caracteres del patrón que son 'no huecos' con los caracteres del texto que están en el mismo lugar.



En este caso hay un mismatch en un *no-hole*. En esta situación, no es necesario comparar los dos primeros *no-hole* del patrón luego del desplazamiento

- Si hay un mismatch en la segunda fase, entonces hay un sufijo del patrón que es igual a un factor del texto y luego de desplazar correctamente estos seguirán coincidiendo y no es necesario volver a compararlos.



En este caso, luego del shift, no hace falta comparar el prefijo que coincidió.

1.2. Implementación

Para la implementación de este algoritmo, se utiliza una serie de *tablas* (implementadas como listas) para el preprocesamiento del patrón a buscar. Dichas tablas permiten realizar desplazamientos en el texto durante la comparación asegurándonos que no nos perderemos de nada y asegurándonos una mayor performance.

Dichas tablas son:

$$\text{Kmin}[i] = \begin{cases} d & sii \quad x[0...i-d-1] = x[d...i-1] \wedge x[i-d] = x[i] \\ 0 & sino \end{cases}$$

Esta tabla indica el shift que se debe realizar en caso de que la posición i pertenezca a un *no-hole*.

Rmin[i] = es el equivalente a Kmin pero para los *hole*.

Sea $ND+1$ la cantidad de *no-holes* en el patrón x, la tabla h contiene a todos los *no-holes* de menor a mayor y luego a los $m-ND-1$ *holes* en orden decreciente. Esto es para recorrer a los *no-holes* de izquierda a derecha y a los *holes* de derecha a izquierda.

$$\text{h}[i] = \begin{cases} h[i] < h[i+1](no-hole) & si \quad i \in [0, ND) \\ h[i] > h[i+1](hole) & si \quad i \in [ND, m) \end{cases}$$

first$[u] = v$, con v entero más pequeño tal que $u \leq h[v]$.

Para calcular el valor de Kmin, utilizamos la tabla hmax:

hmax$[i]$ es tal que: $\begin{cases} x[i...hmax[i]-1] = x[0...hmax[i]-i-1] \\ \\ x[hmax[i]] \neq x[hmax[i]-i] \end{cases}$

Finalmente, utilizamos la tabla

nhd0$[i]$ = cantidad de *no-holes* hasta la posición i.

Con estas tablas, armamos las dos tablas que realmente utilizamos en el algoritmo: *shift* y *next*. El valor de ambas depende de si la posición i contiene a un *hole* o un *no-hole* y se definen como:

$$\text{shift}[i] = \begin{cases} kmin[h[i]] \quad si \quad i \in [0, ND) \\ \\ rmin[h[i]] \quad si \quad i \in [0, ND) \end{cases}$$

$$\text{next}[i] = \begin{cases} ndh0[h[i] - kmin[h[i]]] \quad si \quad i \in [0, ND) \\ \\ ndh0[m - rmin[h[i]]] \quad si \quad i \in [0, ND) \end{cases}$$

Por lo tanto, si la ventana está ubicada en $T[j...j+m-1]$, cuando hay un mismatch entre $P[h[r]]$ y $T[j+h[r]]$, la ventana debe ser desplazada en *shift[r]* y las comparaciones iniciarán desde la posición *h[next[r]]* del patrón.

Por último, devolveremos un match sólo en dos casos:

- Si $i = m$.Arrancamos de $i = 0$ y llegamos al final sin errores.
- Si $j + m - 1 = j + h[i]$. Este es el caso en el que la comparación no empieza desde el inicio de P gracias a algún dato del preprocessing y h[i] = m-1, es decir, llegamos al final de las comparaciones.

1.3. Complejidad

La complejidad de este algoritmo es $O(n + m)$, siendo $O(m)$ la etapa de preprocesamiento y $O(n)$ la etapa de búsqueda. En el peor de los casos, realiza $\frac{3}{2}n$ comparaciones, con n la cantidad de caracteres del *Texto* y m la cantidad de caracteres del *Patrón*.

1.4. Características del algoritmo y casos de prueba

Este algoritmo tiene una característica importante y es que no es necesario conocer el alfabeto para buscar matches, pues en ningún momento es necesario conocer al mismo para ninguna de las tareas que se realizan en el mismo.

Es importante destacar que, si bien fue concebido como una mejora de KMP, en la mayoría de las pruebas que realizamos comparando el algoritmo ingenuo, MP, KMP y Colussi, este último fue el de peor performance. En el único caso en el que logramos que Colussi fuera el mejor fue un caso en el que teníamos un texto de la forma:

```
aaaaaaaaaaa#aaaaaaaaaaaaaa#aaaaaaaaaaa#....
123456789#123456789#123456789#123456789#.....
aaa..
123..
```

Con un patrón de la forma:

```
aaaaaaa#aaaaaa
```

En este caso, Colussi fue mucho mejor que los otros algoritmos previamente mencionados (aproximadamente un 50 % mejor). Pero en todo el resto (archivos de ADN, textos en español, textos en inglés, código en C) tanto con texto largo y patrón corto como con texto corto y patrón largo, encontramos que este algoritmo no tuvo la performance esperada, siendo superado incluso por el algoritmo ingenuo.