
# 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. Rabin - Karp

*Algoritmo Michael O. Rabin and Richard M. Karp 1987*

## 1.1. Funcionamiento

La idea del algoritmo es muy simple. Basándose en la estructura del algoritmo naïve, este agrega un paso previo que compara los strings por valores de hash. Para esto precisa una función de hash que se busca que compare entre valores lo mas rápido posible. Esto tiene el potencial beneficio de acortar los tiempos de comparación entre strings mientras que agrega la complejidad del calculo previo del valor de hash para cada string.

## 1.2. Implementación

La implementación es muy simple. Primero calcula el valor de hash para el patrón a buscar. Luego recorre el texto calculando el valor de hash para la palabra a buscar. Compara ambos valores y si dan iguales entonces compara si las palabras son realmente iguales o no. Para ganar mayor velocidad se utilizo la librería pyhash [1] que contiene implementaciones en C/C++ para mejor eficiencia de algoritmos no criptográficos. De estos se usaron (todos de 32 bits): FNV, Murmur Hash, City Hash, Spooky Hash.

## 1.3. Complejidad

En el peor de los casos, el algoritmo compara cada string del texto contra el patrón teniendo un orden de $O(nm)$ donde n es la longitud del texto y p la del patron. Esto ocurre en el caso en donde se use una función de hash muy mala. Con una función de hash relativamente buena se mejora el orden a $O(n + m)$.

## 1.4. Investigación y aplicaciones

Este algoritmo no es utilizado para Simple Matching ya que resulta poco eficiente. Esto se debe que el costo que tiene para calcular las claves entre algoritmos resulta mayor en relación al beneficio que se obtiene de la rapidez para comparar strings. Investigando sobre sus aplicaciones en el ámbito profesional se encuentra que este algoritmo resulta particularmente útil para el problema de múltiple string matching, mas es así en la búsqueda de plagios. Esto es, teniendo un texto A y un texto B, comparar que tan semejante resulta A contra B.

## 1.5. Conclusiones

Para simple matching este algoritmo resulta increíblemente ineficiente dando los peores tiempos ejecución. Sin embargo para múltiple matching es un muy buen algoritmo. Como optimización se siguiere sacar la parte en donde se verifica que la los valores de hashes que tuvieron match sean realmente iguales. Esto funcionaria sin problemas con una función de hashing perfecto (pero al entiza la ejecución), sin embargo si no lo es el algoritmo pasaría a ser randomizado ya que las funciones de hash utilizadas en este caso garantizan pocas colisiones pero no es imposible que ocurran.

# 2. Zhu-Takaoka

*Algoritmo Zhu Rui Feng - Tadao Takaoka 1987*

## 2.1. Funcionamiento

El algoritmo que esta siendo presentado es una variante del algoritmo de Booyer-Moore. Este algoritmo, al igual que el de BM, mantiene la regla de "good suffix" pero reemplaza la regla de "bad character" por la regla de "2-substrings". Lo que hace esta ultima regla es guardar en una matriz las apariciones mas a la derecha de cada par de caracteres (a,b) pertenecientes al patrón. Entonces el algoritmo va a comparar el texto con el patrón aplicando una de las 2 reglas en caso de encontrar un miss o un match.

## 2.2. Implementación

El algoritmo consta de 2 fases. La primera fase es la de pre-procesamiento en donde se calcula la matriz necesaria para aplicar la regla "2-substrings" y donde se crea el vector para la regla "good suffix" al igual que en el algoritmo de Booyer-Moore. En la segunda fase, el algoritmo alinea el texto con el patrón a izquierda y recorre de derecha a izquierda el patrón comparando carácter a carácter con el texto. En caso de encontrar un miss o de llegar a un

[1] https://github.com/flier/pyfasthash

match, el algoritmo calcula el máximo entre las 2 reglas antes mencionadas, y shiftea el patrón a derecha en esa cantidad. Esto se repite, hasta que el patrón llega al final del texto.

### 2.3. Complejidad

Este algoritmo tiene una complejidad de $O(m + a^2)$ para tiempo y espacio en la fase de pre-procesamiento, siendo a el tamaño del alfabeto. Pero para la fase de búsqueda el algoritmo tiene una complejidad temporal de $O(nm)$, siendo n y m el tamaño del patrón y del texto respectivamente.

### 2.4. Investigación y aplicaciones

Este algoritmo es utilizado con alfabetos pequeños, ya que es cuando resulta eficiente. Esto es debido a la dependencia del tamaño del alfabeto con la fase de pre-procesamiento. Además, este algoritmo resulta muy eficiente en multiple string matching en 2 dimensiones.

### 2.5. Conclusiones

El algoritmo anteriormente presentado, es uno de los que mejor tiempos tiene dentro de los algoritmos implementados en dicho trabajo. Además, se puede ver claramente que la fase de pre-procesamiento aumenta abruptamente a medida que aumentamos el tamaño del alfabeto, tanto en espacio como en tiempo. Se concluye, que este algoritmo funciona muy rápidamente cuando el alfabeto o el patrón son chicos. Como recomendación adicional, se aconseja utilizarlo para patrones chicos o multiple matching.

## 3. 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).

### 3.1. 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.

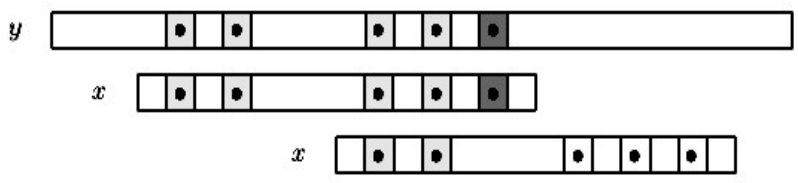
Cada intento del algoritmo consiste entonces de dos pasos:

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*.

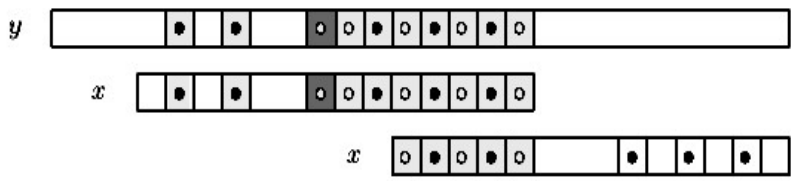
Esta estrategia tiene las siguientes ventajas:

- 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.

## 3.2. 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:

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

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:

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

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

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.

## 3.3. 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*.

## 3.4. 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.