Universidad Francisco de Paula Santander
Vigilada Mineducación

Análisis de Algoritmo

Programa de Ingeniería de Sistemas
Acreditado de Alta Calidad
"Educación y Tecnología con Compromiso Social"

COSTO ALGORÍTMICO TUNED BOYER MOORE

AUTORES

Juan David Ortiz Cano
CÓDIGO: 1152298

Alessandro Umberto Daniele Saltarin
CÓDIGO: 1152194

Cúcuta, 29 de Junio de 2024

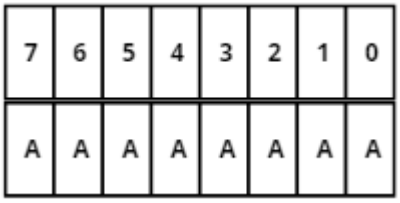
COSTO ALGORÍTMICO

Para determinar la complejidad algorítmica temporal del método de búsqueda de Strings Tuned Boyer Moore lo vamos a realizar con ayuda de la notación Big O considerando su peor caso durante el procesamiento de los patrones con respecto al texto, además, analizaremos brevemente su caso promedio y mejor caso ya que en la práctica son los que más suelen ocurrir. Cabe aclarar que al tratarse de una variación del método Tuned Boyer Moore comparte muchas características con este incluyendo su complejidad algorítmica en ciertos aspectos.

Peor Caso:

El peor caso para el algoritmo ocurre cuando todos los caracteres de nuestro texto T son iguales y coinciden con todos los caracteres del patrón P, ya que las comparaciones y los saltos son a partir de las posiciones asignadas por la regla del mal carácter. Por las reglas establecidas, los saltos para este único carácter son desplazamientos muy pequeños de solo una 1 posición ya que necesita comparar el patrón en cada desplazamiento para comprobar la exactitud y registrarlo.

Podemos ver más fácilmente esta situación con la siguiente imagen:



Por lo tanto la variable auxiliar tendría el valor de 1 y los desplazamientos máximos tras cada comparación es de solo una posición. Esto ocasiona que el ciclo más externo itere n (tamaño del texto) veces y el interno con los saltos ciegos sea constante y solo se haga 1 vez por iteración ocasionando que los desplazamientos del puntero sean mínimos y necesite de más ciclos, además como todos los caracteres son iguales el ciclo encargado de la comparación entre el patrón y el texto será llamado en cada iteración y con una complejidad de m (el tamaño del patrón).

Antes de verificar la complejidad del método principal es necesario verificar la complejidad de cada uno de los métodos que lo componen durante la inicialización y comparación.

Cálculo del T(n) y Big(O):

preBmBc: Método encargado de definir e inicializar un arreglo de enteros para determinar los desplazamientos de cada uno de los caracteres que componen el alfabeto admitido, para ello se hacen dos iteraciones, una de complejidad σ con el tamaño del alfabeto y otra de complejidad m con el tamaño del patrón que se va a buscar. Al principio se le asigna para todos los valores el tamaño del patrón m, luego para los caracteres que contiene el patrón los valores son reemplazados con la posición de su primera aparición de derecha a izquierda del patrón.

Line	Code	Operations Elementals	
1	function preBmBc(x, m, bmBc) {		
2			
3	for (let i = 0; i < ASIZE; ++i) {	*5*	*σ*
4	bmBc[i] = m;	*1*	
5	}		
6			
7	for (let i = 0; i < m - 1; ++i) {	*6*	*m-1*
8	bmBc[x.charCodeAt(i)] = m - i - 1;	*k+2*	
9	}		
10			
11	}		

$$T(n) = 2+1+\sum_{i=0}^{(\sigma-0)/1}(1+2+1)+2+1+2+2+\sum_{j=0}^{(m-1-0)/1}(k+2+2+2)+2+2$$

$$T(n) = 3+\sum_{i=0}^{\sigma}(4)+7+\sum_{j=0}^{m-1}(k+6)+4$$

$$T(n) = 14+(4\sigma)+(k(m-1)+6(m-1))$$

$$T(n) = 14+4\sigma+(km-k+6m-6)$$

$$T(n) = 8+4\sigma+km-k+6m$$

$$T(n) = 8-k+4\sigma+(m(k+6))$$

$$T(n) \to (c+c\sigma+cm) \to O(\sigma+m)$$

memset: Método encargado de añadir al texto un centinela de tamaño m con solo los caracteres finales del patrón, esto se hace con la finalidad de evitar cualquier posible desbordamiento por lo saltos en las comparaciones sin afectar el proceso de búsqueda, lo hace por medio de una iteración de tamaño m y modifica el texto en el retorno.

Line	Code	Operations Elementals	
1	function memset(y, character, m)		
2	{		
3			
4	let aux = '';	*2*	
5			
6	for(let i = 0;i<m;i++)	*2+1+2*	*m*
7	{		
8			
9	aux += character;	*2*	
10			
11	}		
12			
13	return y + aux;	*2*	
14			
15	}		

$$T(n) = 2 + 2 + 1 + \sum_{i=0}^{(m-0)/1} (2 + 1 + 2) + 2 + 1 + 2$$

$$T(n) = 5 + \sum_{i=0}^{m} (5) + 5$$

$$T(n) = 10 + (5m)$$

$$T(n) \to (c + cm) \to O(m)$$

memcmp: Método encargado de comparar los caracteres del fragmento del texto correspondiente en el apuntador con los caracteres del patrón uno a uno, si se encuentra una coincidencia exacta retorna un valor booleano, en caso de tratarse de una coincidencia parcial retorna otro valor booleano, realiza un ciclo con m iteraciones para comparar todos los caracteres que contiene el patrón.

Line	Code	Operations Elementals	
1	function memcmp(y, j, m, x)		
2	{		
3			
4	for(let i = 0;i<m;i++)	*2+1+2*	*m*
5	{		
6			
7	if(y[j + i] !== x[i])	*4*	
8	{		
9			
10	return false;	*1*	
11			
12	}		
13			
14	}		
15			
16	return true;	*1*	
17			
18	}		

$$T(n) = 2 + 1 + \sum_{i=0}^{(m-0)/1} (4 + 1 + 2) + 2 + 1 + 1$$

$$T(n) = 3 + \sum_{i=0}^{m} (7) + 4$$

$$T(n) = 7 + (7m)$$

$$T(n) \rightarrow (c + cm) \rightarrow O(m)$$

TunedBM: Es un algoritmo de búsqueda de cadenas de texto que funciona como una variación del algoritmo Boyer Moore, siendo más rápido para casos promedio y más sencillo de implementar. Este algoritmo realiza comparaciones entre los caracteres y al no encontrar coincidencias realiza los desplazamientos indicados por los valores almacenados en la regla del mal carácter hasta recorrer toda la cadena de texto.

Line	Code	Operations Elementals	
1	const ASIZE = 256;	*2*	
2			
3	function TUNEDBM(x, m, y, n)		
4	{		
5			
6	let j, k, shift;	*3*	
7	const bmBc = new Array(ASIZE);	*k+2*	
8			
9	preBmBc(x, m, bmBc);	*3+8-k+4σ+(m(k+6))*	
10	shift = bmBc[x.charCodeAt(m-1)];	*k+3*	
11	bmBc[x.charCodeAt(m-1)] = 0;	*k+2*	
12			
13	y = memset(y,x[m-1],m);	*6+10+5m*	
14			

15	j = 0;	*1*	
16			
17	while(j<n)	*1*	*n*
18	{		
19			
20	k = bmBc[y.charCodeAt(j + m - 1)];	*k+3*	
21			
22	while(k !== 0)	*1*	*1*
23	{		
24			
25	j += k;	*2*	
26	k = bmBc[y.charCodeAt(j + m - 1)];	*k+3*	
27	j += k;	*2*	
28	k = bmBc[y.charCodeAt(j + m - 1)];	*k+3*	
29	j += k;	*2*	
30	k = bmBc[y.charCodeAt(j + m - 1)];	*k+3*	
31			
32	}		
33			
34	if(j<n && memcmp(y, j, m, x))	*7+7+7m*	
35	{		
36			
37	console.log("Conincidencia encontrada en la posicion: " + j);	*k+1*	
38			
39	}		
40			

41	j+=shift;	*2*	
42			
43	}		
44			
45	}		

$$T(n) = 2+3+k+2+3+8-k+4\sigma+(m(k+6))+k+3+k+2+6+10+$$
$$5m+1+1+\sum_{i=0}^{(n-0)/1}(k+3+1+\sum_{j=0}^{1}(2+k+3+2+k+3+2+k+3+1)+$$
$$1+7+7+7m+k+1+2+1)+1$$
$$T(n) = 41+2k+4\sigma+(km+6m)+5m+\sum_{i=0}^{(n)}(k+4+(16+3k)+19+7m+k)+1$$
$$T(n) = 42+2k+4\sigma+km+11m+\sum_{i=0}^{(n)}(5k+39+7m)$$
$$T(n) = 42+2k+4\sigma+km+11m+(5kn+39n+7nm)$$
$$T(n) = 42+2k+4\sigma+km+11m+5kn+39n+7nm$$
$$T(n) \to c+ck+c\sigma+cm+ckn+cn+nm \to O(\sigma+nm)||O(nm)$$

Cabe hacer una aclaración acerca de la complejidad del algoritmo, al tener que realizar una cantidad de iteraciones σ correspondiente al tamaño del alfabeto para conocer los desplazamientos, podemos decir que la complejidad más precisa del algoritmo es de **O(σ + nm)** en donde n es el tamaño del texto y m es el tamaño del patrón, no obstante, en la práctica el tamaño del alfabeto suele mantenerse como un valor constante al usar el algoritmo y solo cambia para alfabetos muy extensos y complejos, además de que la tasa de σ es mucho más pequeña que nm en la búsqueda en textos promedio.

Por lo tanto, ciertos artículos y páginas que lo referencian determinan que su complejidad en su peor caso de manera simplificada como **O(nm)** considerando el tamaño del alfabeto como un valor constante, para términos de nuestro análisis consideramos importante hacer la aclaración con ambos valores, pero en nuestro caso consideramos que la complejidad de **O(σ + nm)** es mucho más precisa y apropiada para la práctica.

Mejor Caso:

El mejor caso para el algoritmo es cuando todos los caracteres del texto son diferentes a los del patrón o se encuentra una única coincidencia de los caracteres al final del texto lo que permite en cada comparación realizar desplazamientos de m posiciones recorriendo muy rápidamente el texto en orden **O(n/m)** o **O(σ+n/m)** si consideramos la inicialización del alfabeto**.**

Caso Promedio:

En el caso promedio para los alfabetos más extensos se dice que su complejidad es de **O(n+m)** si se consideran alfabetos más pequeños se dice que su complejidad es aproximadamente de **O(nlog(m))** de manera muy general por lo que en la práctica esto puede variar y no ser tan preciso.