



String matching y Árboles de sufijos

(Estructuras de datos avanzadas)

Algoritmia para problemas difíciles

5-12-18

Elvira Mayordomo



Contenido

- **El problema de string matching**
- Árboles de sufijos
- Vectores de sufijos



Referencias

- El capítulo 8.1 (8.1.3) de Steven S. Skiena. *The Algorithm Design Manual*. Springer 2008.
- H.J. Böckenhauer, D. Bongartz: *Algorithmic aspects of bioinformatics*. Springer, 2007.
- D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.



El problema del string matching

- Consiste en encontrar un string (corto), el *patrón*, como substring de un string (largo), el *texto*



Dominios de aplicación

■ Innumerables:

- Spell Checkers,
- Spam Filters,
- Intrusion Detection System,
- Search Engines,
- Plagiarism Detection,
- Bioinformatics,
- Digital Forensics
- Information Retrieval Systems



Un buen resumen

- Importance of String Matching in Real World Problems. K. K. Soni, R. Vyas, A. Sinhal. International Journal Of Engineering And Computer Science (IJECS) 3(6):6371-6375 (2014)

https://www.researchgate.net/profile/Dr_Amit_Sinhal/publication/304305210_Importance_of_String_Matching_in_Real_World_Problems/links/576bb2f708aef2a864d3b881/Importance-of-String-Matching-in-Real-World-Problems.pdf?origin=publication_detail



Un buen resumen

- String Matching Algorithms and their Applicability in various Applications. N. Singla, D. Garg. International Journal of Soft Computing and Engineering (IJSCE) I(6):218-222 (2012)

<http://gdeepak.com/pubs/String%20Matching%20Algorithms%20and%20their%20Applicability%20in%20various%20Applications.pdf>



Enunciado del problema ...

- Entrada: Dos strings $t = t_1 \dots t_n$, $p = p_1 \dots p_m$ sobre Σ
- Salida: El conjunto de posiciones de t donde aparece p , es decir,
 $I \subseteq \{1, \dots, n-m+1\}$
tales que $i \in I$ sii $t_i \dots t_{i+m-1} = p$



Algoritmo inocente ...

StringMatching(patrón $p=p_1 \dots p_m$, texto $t=t_1 \dots t_n$)

$l := \text{vacío}$

for $j := 0$ to $n-m$ do

$i := 1$

 while $p_i = t_{j+i}$ and $i \leq m$ do

$i := i + 1$

 if $i = m + 1$ then $\{p_1 \dots p_m = t_{j+1} \dots t_{j+m}\}$

 añadir($l, j+1$)

Resultado l (El conjunto l de posiciones, donde empieza una ocurrencia de p como substring de t)



Algoritmo inocente ...

- ¿Complejidad?
- ¿Casos peores?

- Buscamos mejorar esto explotando la estructura del patrón (o la del texto)
t= ababb p=abb



Resto de métodos

- Preprocesar el patrón
- Preprocesar el texto (aquí entran los árboles de sufijos)



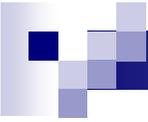
Distintos casos

- Dependiendo del caso se puede buscar el mismo patrón en muchos textos o distintos patrones en el mismo texto
- Por ejemplo buscar la misma mutación en muchos pacientes o todas las mutaciones que tiene un paciente
- Veremos primero algoritmos para **el mismo patrón en muchos textos**
- Luego árboles de sufijos para buscar **distintos patrones en el mismo texto**



Métodos clásicos ...

- Vamos a ver por encima dos métodos clásicos: string matching automata y Boyer-Moore
- Los dos son algoritmos clásicos interesantes que por lo menos hay que conocer ...



Preprocesar el patrón: String matching automata

- Una solución en la que una vez preprocesado p en tiempo $O(|p|(|\Sigma| + \log(|p|)))$ resolvemos el problema en una sola pasada de t
- Usa autómatas finitos ...



Autómata finito (DFA)

Un DFA es una 5-tupla $(Q, \Sigma, q_0, \delta, F)$ donde:

1) Q : conjunto de estados

2) Σ : alfabeto de entrada

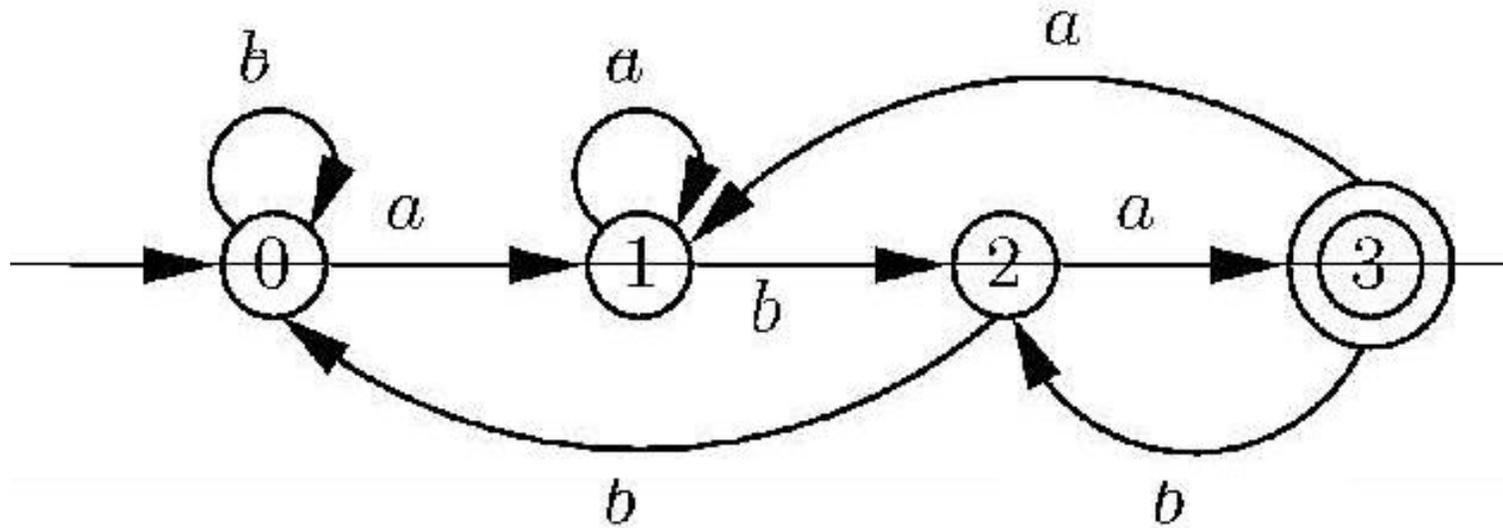
4) $q_0 \in Q$: estado inicial

3) δ : función de transición

$$\delta : Q \times \Sigma \rightarrow Q$$

5) $F \subseteq Q$: conjunto de estados finales
(o de aceptación)

Ejemplo de autómata

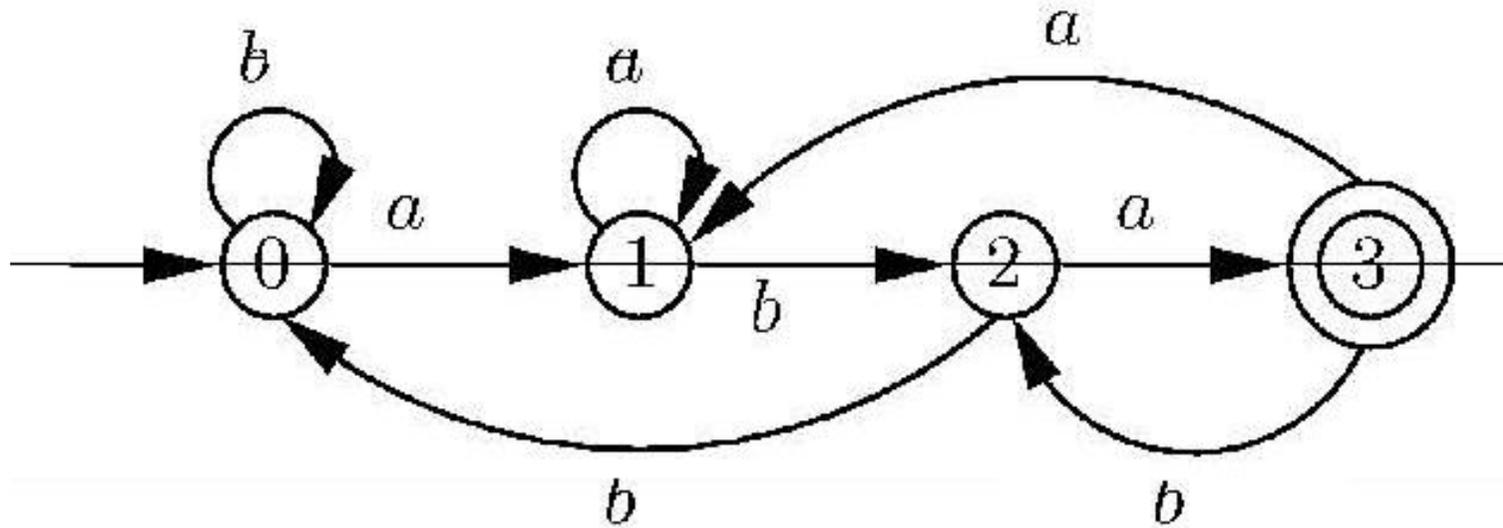




String matching automata

- Dado un patrón $p=p_1 \cdot \dots \cdot p_m$ construimos un autómata que acepte los textos con sufijo p (es decir, los textos que acaban en p)

Ejemplo $p=aba$



$t = \text{bababaa}$

Cada vez que encuentro aba llego al estado final 3



¿Cómo construir el autómata?

- Método inocente ($p = p_1 \dots p_m$)
 - Un estado por cada prefijo de p
 - Estado q_w quiere decir “ w es el final del texto leído que es prefijo de p ”
 - Para asignar $\delta(q_w, a)$ (w prefijo de p):
 - $i=1$; Si $y = w_1 \dots w_{|w|}a$ es prefijo de p
 - $\delta(q_w, a) = q_y$
 - Si no $i++$ y repetir



String matching con autómata

StringMatching($t = t_1 \dots t_n, p = p_1 \dots p_m$)

Calcular M_p el autómata para p ($M_p = (Q, \Sigma, q_0, \delta, F)$)

$q = q_0; l = \text{vacío};$

For $i := 0$ to n do

$q := \delta(q, t_i)$

 If esFinal(q) then $l := \text{añadir}(q, l)$

Resultado l



Complejidad

- Construir el autómata (no visto):
 $O(m(|\Sigma| + \log m))$
- String matching: $O(n \log n)$



Enunciado del problema ...

- Entrada: Dos strings $t = t_1 \dots t_n$, $p = p_1 \dots p_m$ sobre Σ
- Salida: El conjunto de posiciones de t donde aparece p , es decir, $I \subseteq \{1, \dots, n-m+1\}$ tales que $i \in I$ sii $t_i \dots t_{i+m-1} = p$



Vistos

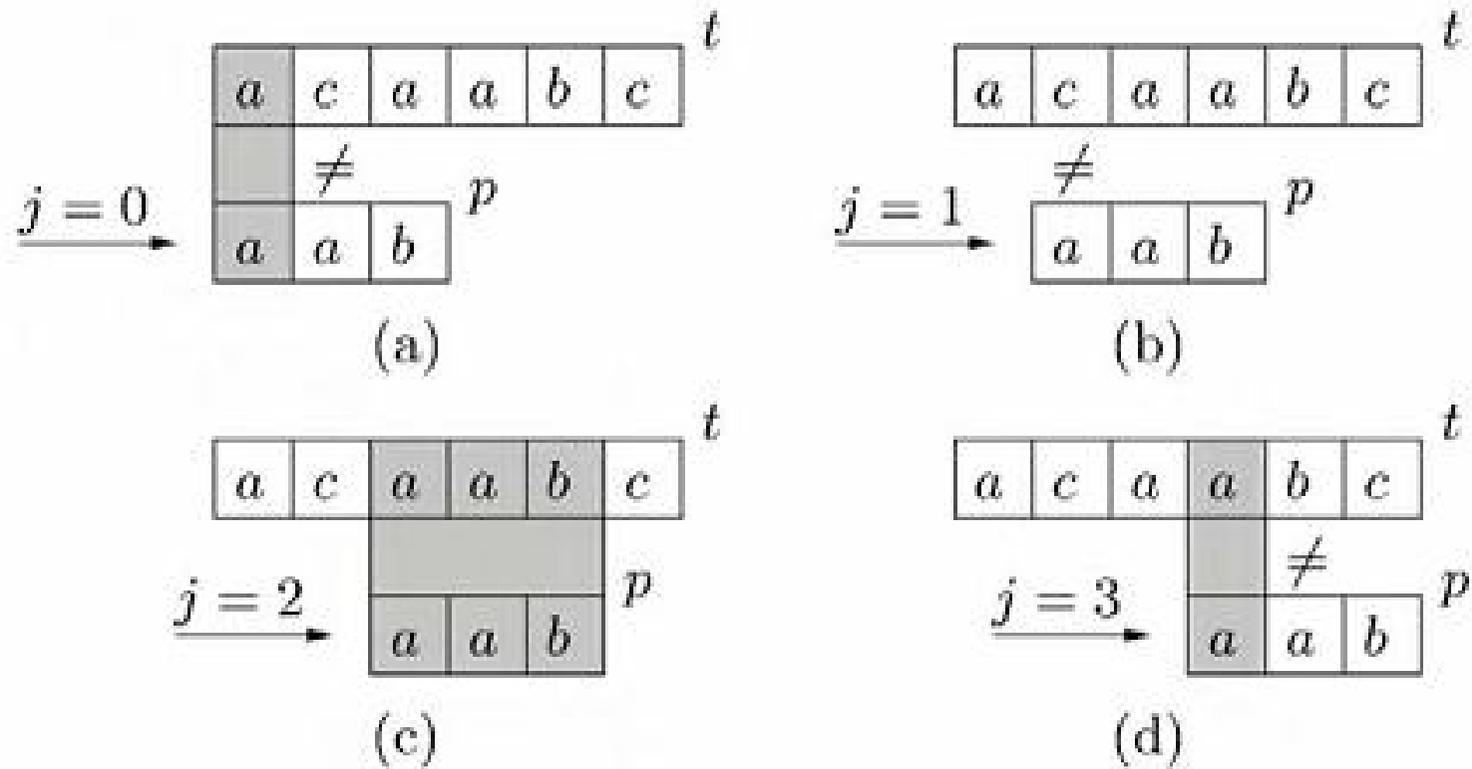
- Algoritmo inocente: $O(m \cdot (n-m) \log n)$
- String matching autómatas:
 - $O(m(|\Sigma| + \log m))$ preprocesamiento del patrón
 - $O(n \log n)$ string matching

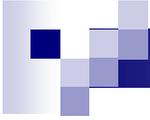


Algoritmo de Boyer-Moore

- Se trata de utilizar el algoritmo “inocente”:
ir moviendo patrón sobre el texto
- Pero ...
 - Cada comparación de $p_1 \dots p_m$ con $t_{j+1} \dots t_{j+m}$ la hacemos empezando por el final
 - Si podemos movemos más de una posición la j
- Usa preprocesamiento de p

Ejemplo del inocente ...

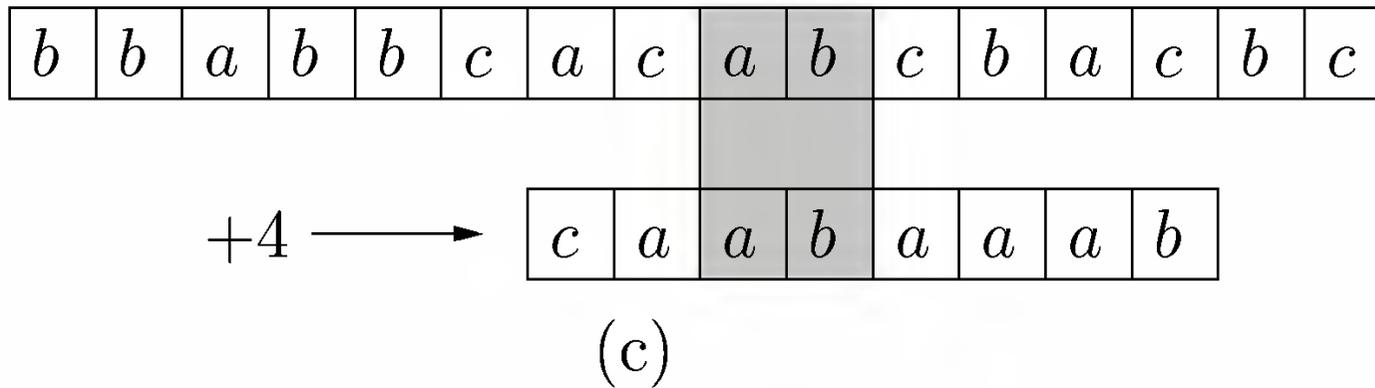
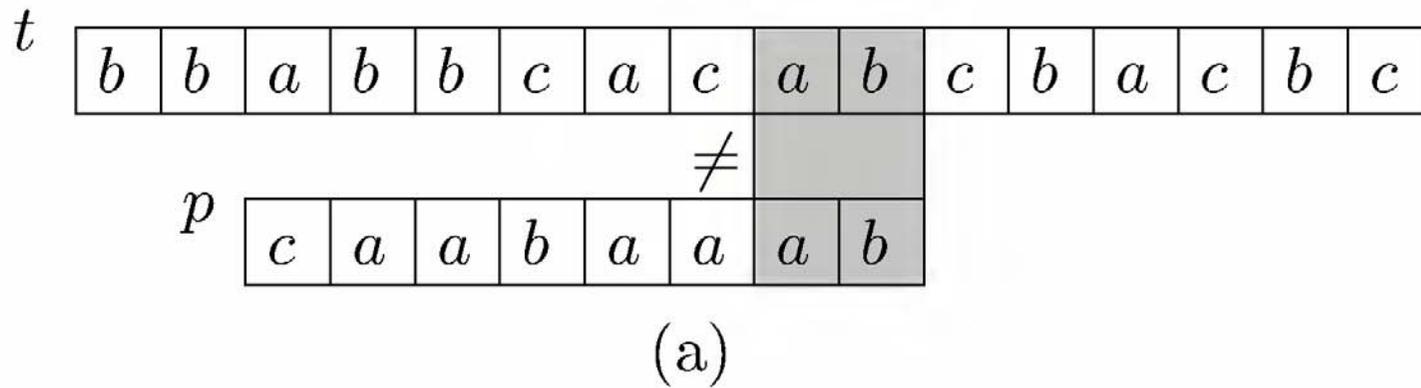


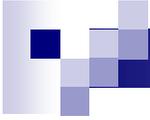


Regla del carácter raro

- Si c es la primera diferencia (carácter raro) entre t y p y está en la posición i de p
Busca la ocurrencia más cercana y anterior a i de c en p
- Tenemos ya precalculado $cr(i,c)$ para cada i de 1 a m y para carácter c

Boyer Moore se comporta ...





Regla del buen sufijo

- Si c es la primera diferencia (carácter raro) entre t y p y está en la posición i de p
 - Busca la ocurrencia de $p[i+1..m]$ más cercana y anterior en p
 - Si no existiera prueba con $p[j..m]$ para $j > i+1$
- Tenemos ya precalculado $bs(i)$ para cada i de 1 a $m-1$



Complejidad de Boyer-Moore

- Preprocesamiento $O(m(|\Sigma| + \log m))$
- Caso peor: igual que el inocente $O(|\Sigma| \cdot m \log m + n \cdot m \log n)$
- Bastante rápido en la práctica
- Se puede mejorar el preproceso para que el caso peor sea $O((n+m) \log n)$



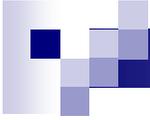
Comparando complejidades

Caso peor ...

- Algoritmo inocente $O(m \cdot (n-m) \log n)$
- String matching automata $O(n \log n + m(|\Sigma| + \log m))$
- Boyer-Moore $O(n \log n + (m + |\Sigma|) \log m)$

El caso peor del Boyer-Moore ocurre poco

Alternativa: el Knuth-Morris-Pratt (mejor en el caso peor, peor en la práctica)



Separando el preprocesamiento ...

- Algoritmo inocente: $O(m \cdot (n-m) \log n)$
- String matching autómatas:
 - $O(m(|\Sigma| + \log m))$ preprocesamiento del patrón
 - $O(n \log n)$ string matching
- Boyer-Moore
 - $O(m(|\Sigma| + \log m))$ preprocesamiento del patrón
 - $O(n \log n)$ string matching



Contenido

- El problema de string matching
- **Árboles de sufijos**
- Vectores de sufijos



Árboles de sufijos

- Se trata de preprocesar el texto
- Esto es útil cuando se quieren encontrar muchos patrones en el mismo texto (por ejemplo, muchos genes en el mismo DNA)



Árboles de sufijos

- El patrón p ocurre en t cuando p es el prefijo de un sufijo de t
- Se trata de calcular y almacenar eficientemente los sufijos de t

Ejemplo ...

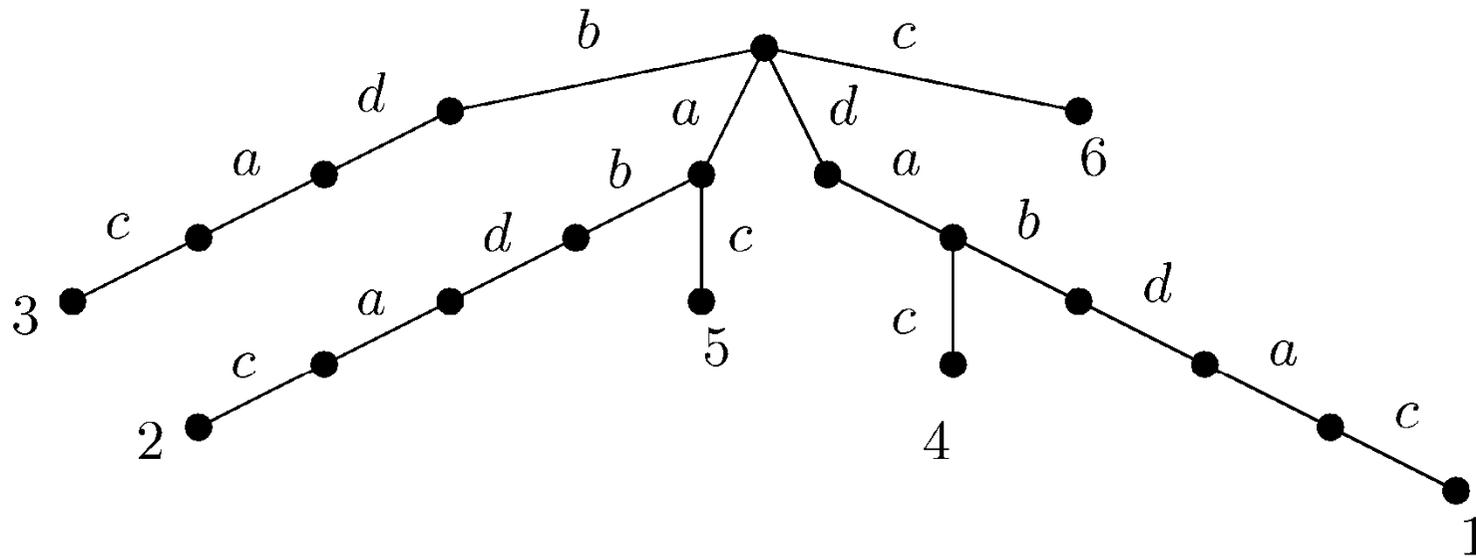


Fig. 4.5. A simple suffix tree for the text $t = dabdac$



Definición

- Dado un string $t=t_1\dots t_n$, un árbol sufijo simple es un árbol dirigido $T_t=(V,E)$ con una raíz r que cumple
 1. El árbol tiene n hojas con etiquetas $1, \dots, n$. (La etiqueta i corresponde al sufijo $t_i\dots t_n$)
 2. Las aristas del árbol están etiquetadas con letras del alfabeto
 3. Todas las aristas de salida de un nodo tienen letras diferentes
 4. El camino de la raíz a la hoja i tiene etiquetas $t_i\dots t_n$



¿Funciona siempre?

- ¿Qué pasa con cadada?
- Hay sufijos que son prefijos de otros sufijos ...
- Para evitarlo se construye el árbol de t\$

Construcción simple

ÁrbolSufijos($t = t_1 \dots t_n$)

$t' := t\$$

$T :=$ árbol con raíz r

for $i := 0$ to n do

 {insertar $t_i \dots t_n\$$ en el árbol T }

 Empezando en la raíz buscar un camino máximo $t_i \dots t_j$
 terminando en el nodo x

 Añadir al árbol los nodos correspondientes a $t_{j+1} \dots t_n\$$ como
 camino a partir de x con etiquetas $t_{j+1}, \dots, t_n, \$$,
 etiquetando la hoja con i

Resultado T (El árbol de sufijos de $t_1 \dots t_n\$$)



¿Cuánto cuesta construir el árbol?

- Con el algoritmo simple $O(n^2 \log n)$
- Se trata de añadir cada sufijo letra por letra
- Además ocupa bastante memoria, ¿es imprescindible?



String matching

- Si tenemos el árbol de sufijos de t ,
¿cuánto cuesta encontrar si p es substring
de t ?
- ¿y todas las apariciones de p como
substring de t ?



String matching

- Saber si p está en t cuesta tiempo $O(|p|\log|t|)$
- **Suponiendo que ya tenemos el árbol**
- Encontrar todas las ocurrencias de p en t puede ser muy largo dependiendo del tamaño del subárbol con raíz p

- Ejemplo: $t=a^n b^n c$



¿Podemos mejorar?

- Vamos a buscar una representación compacta del árbol
- Ahorraremos memoria y tiempo de string matching



Árboles de sufijos compactos

- Podemos eliminar los nodos con un solo hijo ... poniendo strings como etiquetas
- No tenemos que escribir las etiquetas largas
- En lugar de una etiqueta de k símbolos (ocupa $k \cdot \log(|\Sigma|)$) usamos los dos índices en el texto (ocupa $2 \cdot \log n$)
 - Cuando $k \cdot \log(|\Sigma|) > 2 \cdot \log n$



Definición formal

- Dado un string $t=t_1\dots t_n$, un árbol sufijo compacto es un árbol dirigido $T_t=(V,E)$ con una raíz r que cumple
 1. El árbol tiene n hojas con etiquetas $1, \dots, n$. (La etiqueta i corresponde al sufijo $t_i\dots t_n$)
 2. Cada vértice interior tiene al menos **dos hijos**
 3. Las aristas del árbol están etiquetadas con substrings de t (representadas tal cual o con los índices de inicio y final en el texto)
 4. Todas las aristas de salida de un nodo empiezan por letras diferentes
 5. El camino de la raíz a la hoja i tiene etiquetas $t_i\dots t_n$



Tamaño de un a. sufijo compacto

- Como tiene n hojas y todos los nodos interiores tienen al menos 2 hijos ...
 - Hay un máximo de $n-1$ nodos interiores
 - En total $2n-1$ nodos como máximo
- Codificar un nodo cuesta como máximo $2 \log n$ bits
- En total $O(n \log n)$ de memoria



Notación en el algoritmo

- Si x es un nodo del árbol sufijo compacto:
 - $\text{pathlabel}(x)$ es la concatenación de las etiquetas desde la raíz a x
 - $\text{depth}(x) = |\text{pathlabel}(x)|$
- Si x es un nodo del árbol sufijo:
 - $\text{Pos}(x)$ es la mínima etiqueta de una hoja del subárbol de raíz x



Construcción compacta (1)

ÁrbolSufijosCompacto($t = t_1 \dots t_n$)

T := ÁrbolSufijos(t);

{Eliminar los nodos de grado 2}

X := nodos de grado 2 de T;

While not EsVacío(X) do

 Elegir x en X con hijo z, padre y

 Reemplazar (y,x), (x,z) por (y,z) con etiqueta

 label(y,z) = label(y,x) label(x,z)

 Borrar x

End while

...



Construcción compacta (2)

ÁrbolSufijosCompacto($t = t_1 \dots t_n$)

...

{Comprimir las etiquetas largas}

For (x,y) arista do

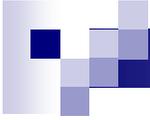
 If $|\text{label}(x,y)| \geq 2 \log(n)$ then

 {Reemplazar la etiqueta por los números de posición}

$\text{label}'(x,y) = (\text{Pos}(y)+\text{depth}(x), \text{Pos}(y)+\text{depth}(x)+|\text{label}(x,y)|-1);$

$\text{label} = \text{label}'$

Resultado T (El árbol de sufijos compacto de $t_1 \dots t_n$)



Coste de la construcción

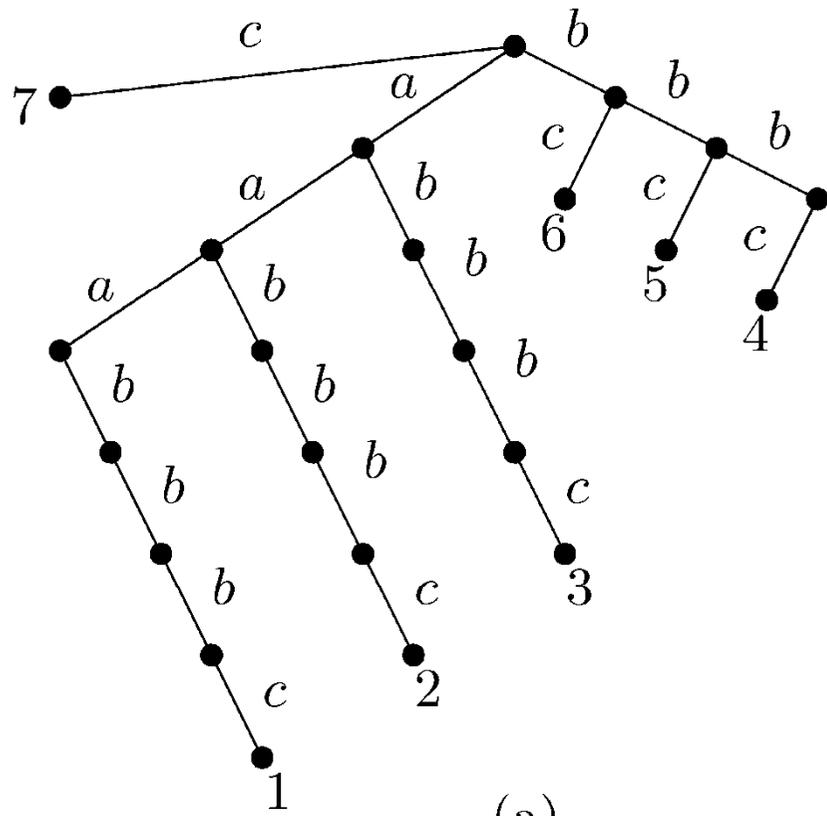
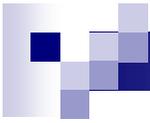
- Como el árbol sufijo inicial puede ser de tamaño $O(n^2 \log n)$ el tiempo es $O(n^2 \log n)$
- Hay métodos de construcción del árbol sufijo compacto sin pasar por el inicial que tardan $O(n \log n)$ (algoritmo de Ukkonen)
- Como el tamaño del árbol es $O(n \log n)$ esto es lo mínimo posible ...

Los textos que dicen que Ukkonen es $O(n)$ mienten

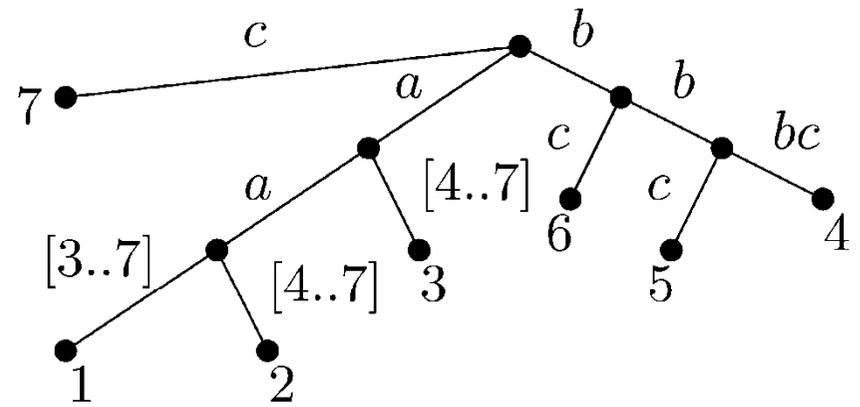


String matching usando árboles de sufijos compactos

- Buscamos un patrón p en un texto t
- Tenemos que encontrar un nodo x del árbol de forma que el camino de la raíz a x sea p ó bien tenga p como prefijo
- Las apariciones de p en t nos las dan las hojas del subárbol de x



(a)



(b)



StringMatching(1)

StringMatching($t = t_1 \dots t_n, p = p_1 \dots p_m$)

(1) $T := \text{ÁrbolSufijosCompacto}(t)$;

(2) {Inicialización}

$x := \text{raiz}(T)$; {vértice actual}

$i := 1$; {posición actual en p }

$\text{found} := \text{false}$;

$\text{possible} := \text{true}$;

...



StringMatching(2)

While not found and possible

(3a) {**buscar una arista desde x con label empezando en p_i** }

fitting:=false;

U:= hijos de x;

While not fitting and not esVacío(U) do

Elegir v en U

If label(x,v)= $p_i a$ then

fitting:=true

mylabel:=label(x,v)

else if label(x,v)=[k..l] and $t_k = p_i$ then

fitting:=true

$l' = \min(l, k+m-i)$

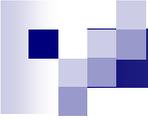
mylabel:= $t_k \dots t_{l'}$

{leer la parte necesaria de la etiqueta}

borrar(v,U)

end While

...



StringMatching(3)

While not found and possible

....

(3b) {**Comparar mylabel con la parte de p por encontrar**}

If ($p_i \dots p_m$ no es prefijo de mylabel) and (mylabel no es prefijo de $p_i \dots p_m$) then

 possible := false { p no aparece en t }

else if mylabel es prefijo de $p_i \dots p_m$ then

$x := v$

$i := i + |\text{mylabel}|$

else { $p_i \dots p_m$ es prefijo de mylabel}

$x := v$

 found := true

End while

(4) if found then

 Calcular el conjunto I de las etiquetas de hojas en el subárbol de raíz x (búsqueda en profundidad)

Resultado I (El conjunto I de posiciones, donde empieza una ocurrencia de p como substring de t)



Complejidad

3(a)

- Si sólo hay que atravesar etiquetas no comprimidas (abab) para encontrar un patrón $p=p_1\dots p_m$ basta tiempo $|\Sigma|m$
- Si hay que pasar por alguna etiqueta “comprimida” [2..5] eso quiere decir que la etiqueta corresponde a un fragmento largo (si no se hubiera comprimido), a un fragmento de tamaño $\Omega(\log n)$. Luego eso sólo puede ocurrir $m/\log n$ veces como mucho
- Leer todos los fragmentos comprimidos cuesta como mucho $m/\log n \cdot \log n$



Complejidad

- 3(b) cuesta tiempo $m \log m$ en total
- Atravesar el subárbol encontrado cuesta $O(k \log n)$ si p ocurre k veces en t
- El algoritmo completo cuesta $O(n \log n + m (|\Sigma| + \log m) + k \log n)$



Resumen string matching

Encontrar todas las ocurrencias de ...
un patrón de tamaño m en un texto de tamaño n

- Algoritmo inocente $O(m \cdot (n-m) \log n)$
- String matching automata $O(n \log n + m(|\Sigma| + \log m))$
- Boyer-Moore $O(n \log n + (m + |\Sigma|) \log m)$
- Árboles de sufijos compactos (k es el número de ocurrencias) $O(n \log n + m(|\Sigma| + \log m) + k \log n)$



Resumen ...

- Algoritmo inocente: $O(m \cdot (n-m) \log n)$
- String matching autómata:
 - $O(m(|\Sigma| + \log m))$ preprocesamiento del patrón
 - $O(n \log n)$ string matching
- Boyer-Moore
 - $O(m(|\Sigma| + \log m))$ preprocesamiento del patrón
 - $O(n \log n)$ string matching
- Árboles de sufijos
 - $O(n \log n)$ preprocesamiento del **texto**
 - $O(m(|\Sigma| + \log m) + k \log n)$ string matching



Resumen

Encontrar todas las ocurrencias de ...
un patrón de tamaño m en **N textos** de tamaño n

- Algoritmo inocente $O(N \cdot m \cdot (n-m) \log n)$
- String matching automata $O(N \cdot n \log n + m(|\Sigma| + \log m))$
- Boyer-Moore $O(n \cdot N \log n + m(|\Sigma| + \log m))$
- Árboles de sufijos compactos (k es el número máximo de ocurrencias)
 $O(N n \log n + N m (|\Sigma| + \log m) + N k \log n)$

Lo mejor: preprocesamiento de patrón (autómata o BM)



Resumen

Encontrar todas las ocurrencias de ...

M patrones de tamaño m en un texto de tamaño n

- Algoritmo inocente $O(M \cdot m \cdot (n-m) \log n)$
- String matching automata $O(M \cdot n \log n + Mm(|\Sigma| + \log m))$
- Boyer-Moore $O(n \cdot M \log n + mM(|\Sigma| + \log m))$
- Árboles de sufijos compactos (k es el número total de ocurrencias)
 $O(n \log n + M m (|\Sigma| + \log m) + k \log n)$

Lo mejor: preprocesamiento de texto (árboles de sufijos)



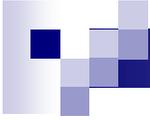
Contenido

- El problema de string matching
- Árboles de sufijos
- **Vectores de sufijos**



Vectores de sufijos

- Hemos visto la utilidad de los árboles de sufijos
- Ahora nos planteamos guardar los sufijos en un vector, ordenados alfabéticamente (lexicográficamente)
- Ejemplo: $s=ababbabb$



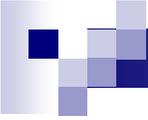
Definición

- El vector de sufijos de un string s es

$$A(s) = (j_1, \dots, j_n)$$

tal que el orden alfabético de los sufijos es

$$s[j_1, n] < s[j_2, n] < \dots < s[j_n, n]$$



Ejemplo

■ $s = \text{dabdac}$

$\text{abdac} < \text{ac} < \text{bdac} < \text{c} < \text{dabdac} < \text{dac}$

$[2..6] < [5..6] < [3..6] < [6..6] < [1..6] < [4..6]$

$A(s) = (2, 5, 3, 6, 1, 4)$



String matching con vectores de sufijos

- Para encontrar todas las ocurrencias de un patrón p en un texto t contando con el vector de sufijos de t
- Usar búsqueda dicotómica para encontrar el primer y último sufijo de t que empiezan por p



Algorithm 4.12 String matching using a suffix array

Input: A text t and a pattern p over an ordered alphabet Σ , and a suffix array $\mathcal{A}(t)$ for the text t .

1. Use binary search to find the first position i and the last position j in the suffix array such that p starts as a substring in t at positions $\mathcal{A}(t)[i]$ and $\mathcal{A}(t)[j]$.
2. $I := \{\mathcal{A}(t)[i], \mathcal{A}(t)[i + 1], \dots, \mathcal{A}(t)[j]\}$.

Output: The set I of all positions, where p starts as a substring in t .

Coste: $O(m \log n + k \log n)$ donde k es el número de ocurrencias



¿Cuánto cuesta construir el vector?

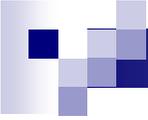
- Se usa el algoritmo “skew” o sesgo basado en ordenación
- Cuesta tiempo $O(n \log n)$
- Es bastante delicado ...

- Veamos las ideas para las ...



Radix sort para strings

- Podemos ordenar alfabéticamente n strings de longitud d sobre un alfabeto de k símbolos en tiempo $O((n+k)d)$
- Para ello ordenamos de forma estable según cada una de las posiciones de la 1 a la d



Ordenar n valores de 0 a k

Algorithm 4.13 Counting Sort

Input: An array $A = (A(1), \dots, A(n))$ of integers from the range $\{0, \dots, k\}$.

1. Counter initialization:

```
for  $i := 0$  to  $k$  do  
   $c(i) := 0$ 
```

2. Count the number of elements of each type:

```
for  $j := 1$  to  $n$  do  
   $c(A(j)) := c(A(j)) + 1$ 
```

3. Count the number of elements less or equal to i :

```
for  $i := 1$  to  $k$  do  
   $c(i) := c(i) + c(i - 1)$ 
```

4. Calculate the position of each element in the sorted array:

```
for  $j := n$  downto 1 do  
   $B(c(A(j))) := A(j)$   
   $c(A(j)) := c(A(j)) - 1$ 
```

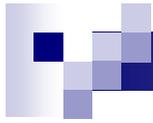
Output: The sorted array $B = (B(1), \dots, B(n))$.

$O((n+k) \log n)$



Para ordenar n strings ...

- El algoritmo anterior ordena **de forma estable** n datos de 0 a k
- Lo utilizamos para ordenar, según una posición, n strings sobre un alfabeto de k símbolos



Algorithm 4.14 Radix Sort

Input: An array $A = (a_1, \dots, a_n)$ of strings of length d in k -letter alphabet

for $i := 1$ **to** d **do**

Sort A according to the i -th letter using Algorithm 4.13.

Output: The sorted array.

$O((n+k)d \log n)$



El algoritmo para construir el vector

- Separa los sufijos que tienen longitud múltiplo de 3 (S^0) del resto ($S^{1,2}$)
- Construye primero $A^{1,2}$, el vector de $S^{1,2}$ ordenando sólo las 3 primeras letras de cada sufijo por radix-sort
- si no es suficiente se cambia el alfabeto (cada tres letras viejas es una nueva) y se hace llamada recursiva

Atención, si se usa recursión muchas veces hay que complicarlo para mantener el tiempo (crece el alfabeto)



El algoritmo para construir el vector

- Separa los sufijos que tienen longitud múltiplo de 3 (S^0) del resto ($S^{1,2}$)
- Construye primero $A^{1,2}$, el vector de $S^{1,2}$
- Construye A^0 usando $A^{1,2}$
- Mezcla A^0 y $A^{1,2}$
- Coste $O(n \log n)$
- Muchas referencias tienden a considerar este tiempo $O(n)$ contando las comparaciones entre números $O(1)$

Algorithm 4.15 Skew algorithm for constructing a suffix array

Input: A string $s = s_1 \dots s_n$ over the ordered alphabet $\{1, \dots, n\}$.

1. Initialization:

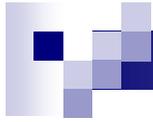
- a) Define $s_{n+1} := 0$, $s_{n+2} := 0$, and $s_{n+3} := 0$, and let s denote the string $s_1 \dots s_{n+3}$.
- b) Let $S_i := s_i \dots s_n$ be the i -th suffix of s for all $0 \leq i \leq n + 1$.
- c) Let $k_i = \max\{k \leq n + 1 \mid k \equiv i \pmod{3}\}$, for $i \in \{1, 2\}$;
let $l_i := |\{k \leq n + 1 \mid k \equiv i \pmod{3}\}|$, for $i \in \{1, 2\}$.

2. Construct the suffix array $\mathcal{A}^{1,2}$ for the set $\mathcal{S}^{1,2}$ of all suffixes S_i such that $i \not\equiv 0 \pmod{3}$:

- a) Let $t_i := s[i, i + 2]$ for all $0 \leq i \leq n + 1$ such that $i \not\equiv 0 \pmod{3}$.
- b) Sort the triples t_i , for all $i \leq \max\{k_1, k_2\}$, $i \not\equiv 0 \pmod{3}$, using radix sort (Algorithm 4.14).
- c) Assign lexicographical names $t'_i \in \{1, \dots, \lceil \frac{2}{3}(n + 1) \rceil\}$ to the triples, i.e., define the t'_i such that $t'_i = t'_j$ if $t_i = t_j$, and $t'_i < t'_j$ if $t_i \prec_{\text{lex}} t_j$.
- d) If all t'_i are distinct, construct $\mathcal{A}^{1,2}$ directly from the order of the t_i ; else, recursively compute the suffix array $\tilde{\mathcal{A}}$ for the string

$$\tilde{s} = \tilde{s}_1 \dots \tilde{s}_{l_1+l_2} := t'_1 t'_4 t'_7 \dots t'_{k_1} \cdot t'_2 t'_5 t'_8 \dots t'_{k_2}$$

and construct $\mathcal{A}^{1,2}$ from $\tilde{\mathcal{A}}$ by substituting the indices of the t'_i for the indices of the \tilde{s}_j .



3. Construct the suffix array \mathcal{A}^0 for the set \mathcal{S}^0 of all suffixes S_i such that $i \equiv 0 \pmod{3}$:
 - a) Represent S_i by the pair (s_i, S_{i+1}) for all $1 \leq i \leq n$ such that $i \equiv 0 \pmod{3}$.
 - b) Consider the order of \mathcal{S}^0 as given by the order of the second component of its elements in $\mathcal{A}^{1,2}$, and sort \mathcal{S}^0 by counting sort (Algorithm 4.13) with respect to the first components.
4. Merge the two suffix arrays $\mathcal{A}^{1,2}$ and \mathcal{A}^0 into a suffix array $\mathcal{A}(s)$.

Output: The constructed suffix array $\mathcal{A}(s)$.



Resumen

Encontrar todas las ocurrencias de ...

M patrones de tamaño m en un texto de tamaño n

- Algoritmo inocente $O(M \cdot m \cdot (n-m) \log n)$
- String matching automata $O(M \cdot n \log n + Mm(|\Sigma| + \log m))$
- Boyer-Moore $O(n \cdot M \log n + mM(|\Sigma| + \log m))$
- Árboles de sufijos compactos (k es el número total de ocurrencias)
 $O(n \log n + Mm(|\Sigma| + \log m) + k \log n)$
- Vectores de sufijos $O(n \log n + m \cdot \log n \cdot M + k \log n)$

Lo mejor: preprocesamiento de texto