Árboles de sufijos

Algoritmia para problemas difíciles
25-11-14
Elvira Mayordomo

Contenido

- **El problema de string matching**
- Algoritmos de propósito general
- Árboles de sufijos
- Otras aplicaciones de árboles de sufijos
- Vectores de sufijos

Referencias

- 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*
- En bioinformática lo más frecuente es buscar un fragmento nuevo de DNA (un gen) en una colección de secuencias
- En este caso permitimos un cierto error, pero el pattern matching exacto es una subrutina

Enunciado del problema …

- Entrada: Dos strings t= t_1 … t_n, p= p_1 … p_m sobre Σ
- Salida: El conjunto de posiciones de t donde aparece p, es decir,

  $I \subseteq \{1,\ldots,\quad n\text{-}m+1\}$

  tales que $i \in I$ sii t_i … t_{i+m-1} =p

Contenido

- El problema de string matching
- **Algoritmos de propósito general**
- Árboles de sufijos
- Otras aplicaciones de árboles de sufijos
- Vectores de sufijos

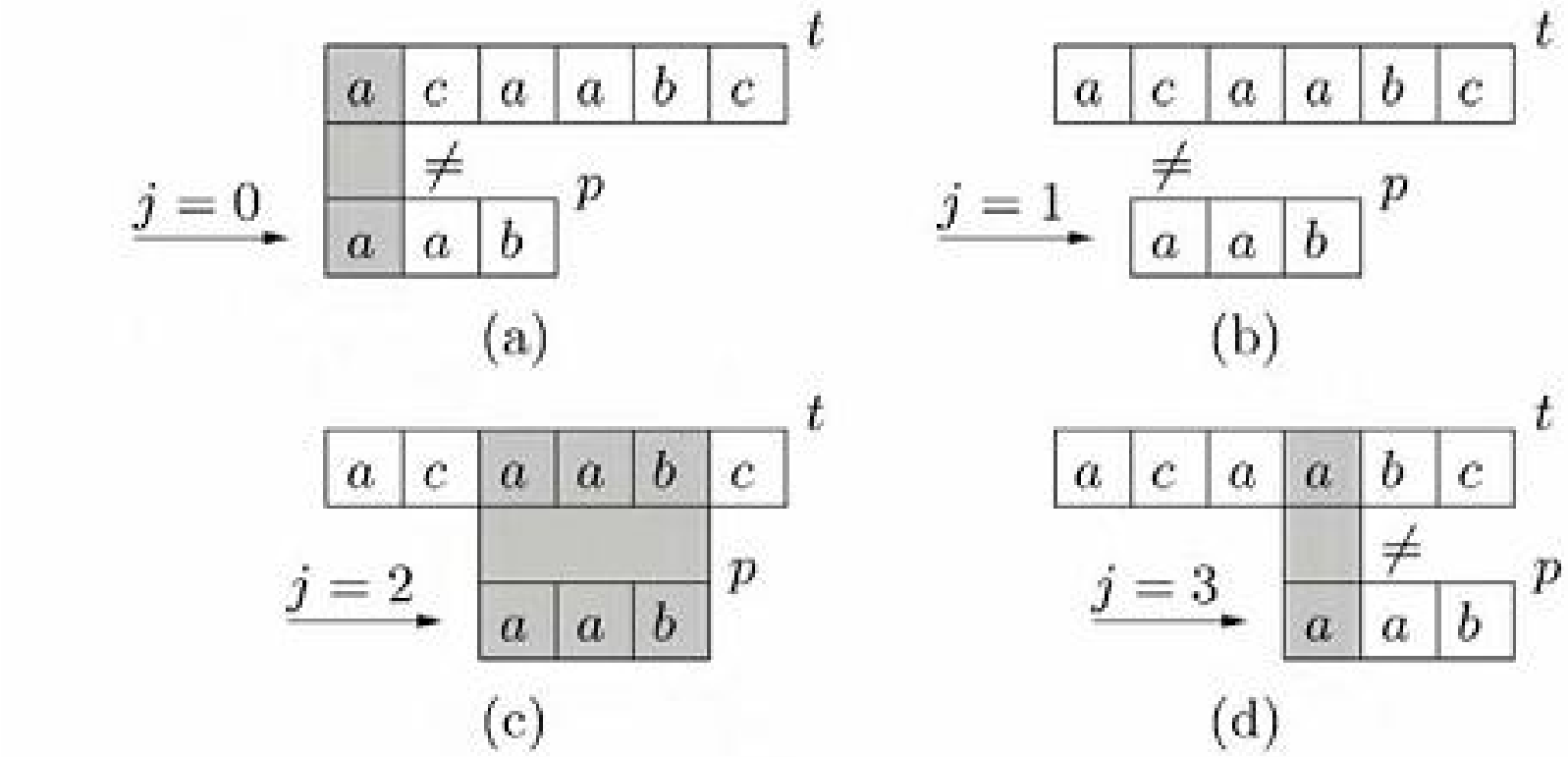
Algoritmo inocente …

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

I:=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 {$p_1 \ldots p_m = t_{j+1} \ldots t_{j+m}$}
        añadir(*I*,*j*+1}

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

Ejemplo ...


t
a c a a b c
≠
j = 0
p
a a b
(a)
t
a c a a b c
≠
j = 1
p
a a b
(b)
t
a c a a b c
j = 2
p
a a b
(c)
t
a c a a b c
≠
j = 3
p
a a b
(d)

Algoritmo inocente …

- ¿Complejidad?
- ¿Casos peores?

- Buscamos mejorar esto explotando la estructura del patrón

t= ababb p=abb

Resto de métodos

- Preprocesar el patrón
- Preprocesar el texto (árboles de sufijos)

Resto de métodos

- Vamos a ver por encima dos métodos clásicos: string matching automata y Boyer-Moore
- Los dos son algoritmos interesantes pero quedan fuera de la asignatura

Preprocesar el patrón: String matching automata

- Una solución en la que una vez preprocesado p en tiempo $O(|p|\cdot|\Sigma|)$ 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

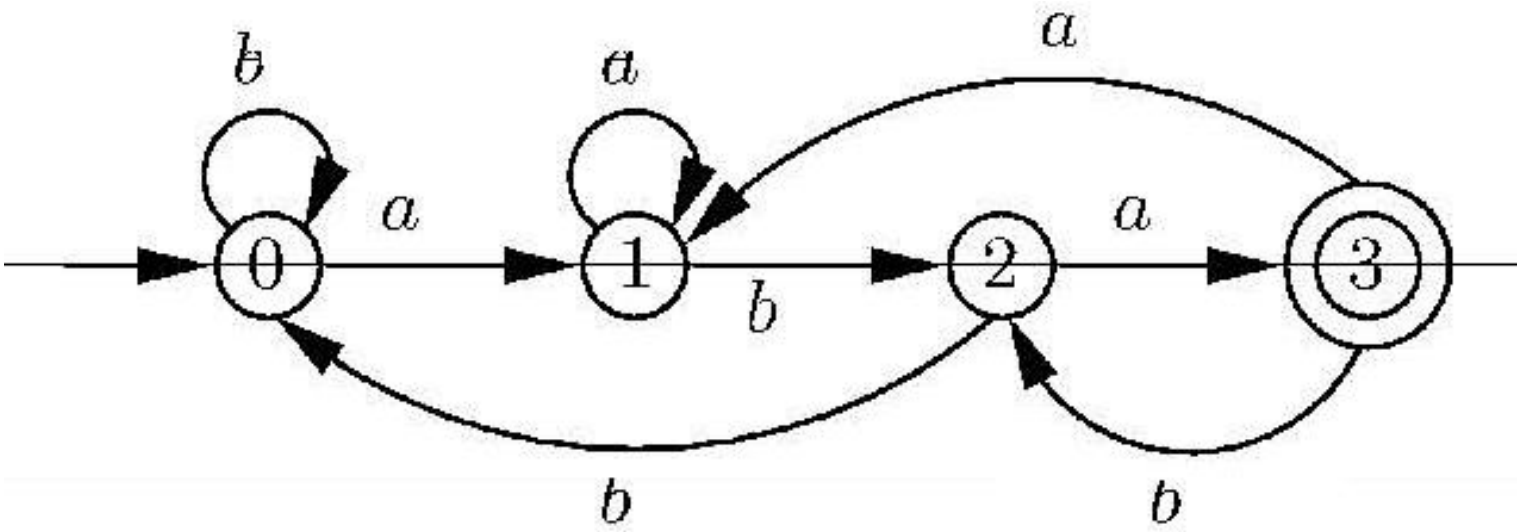
4) $q_0 \in Q$: estado inicial

3) δ : función de transición

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

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

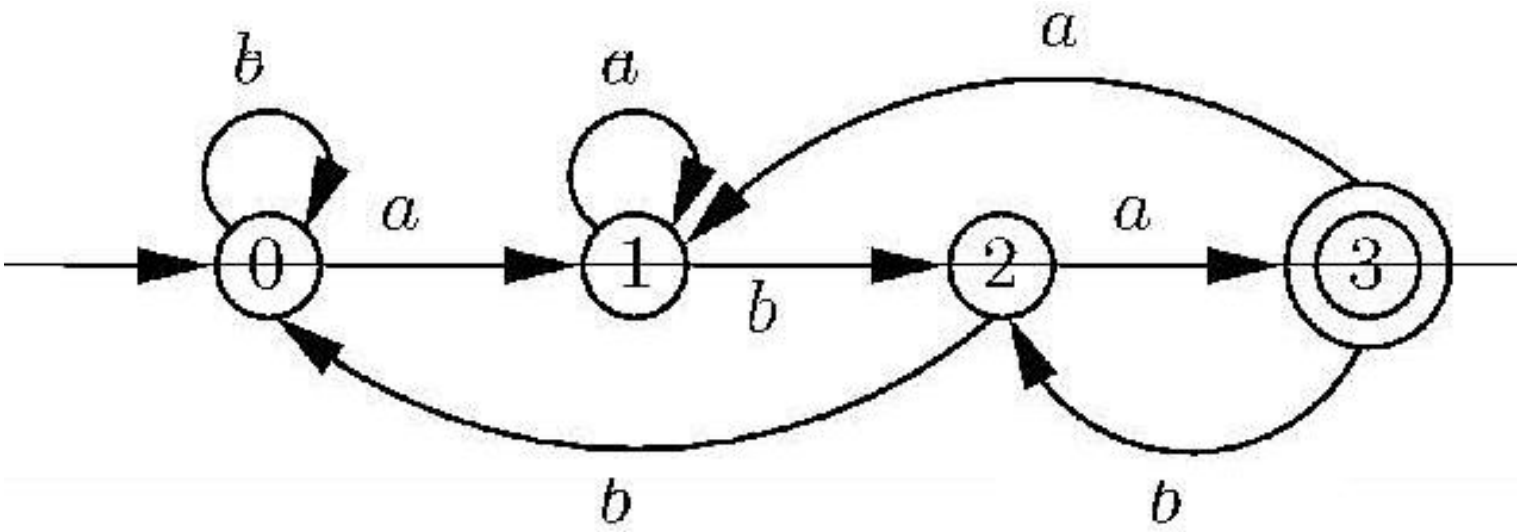
Ejemplo de autómata



String matching automata

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

Ejemplo p=aba



t= bababaa

Cada vez que encuentro aba llego al estado final 3

String matching con autómata

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

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

$q= q_0$; I=vacío;

For i:=0 to n do

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

If esFinal(q) then I:=añadir(q,I)

Resultado I

Complejidad

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

Enunciado del problema …

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

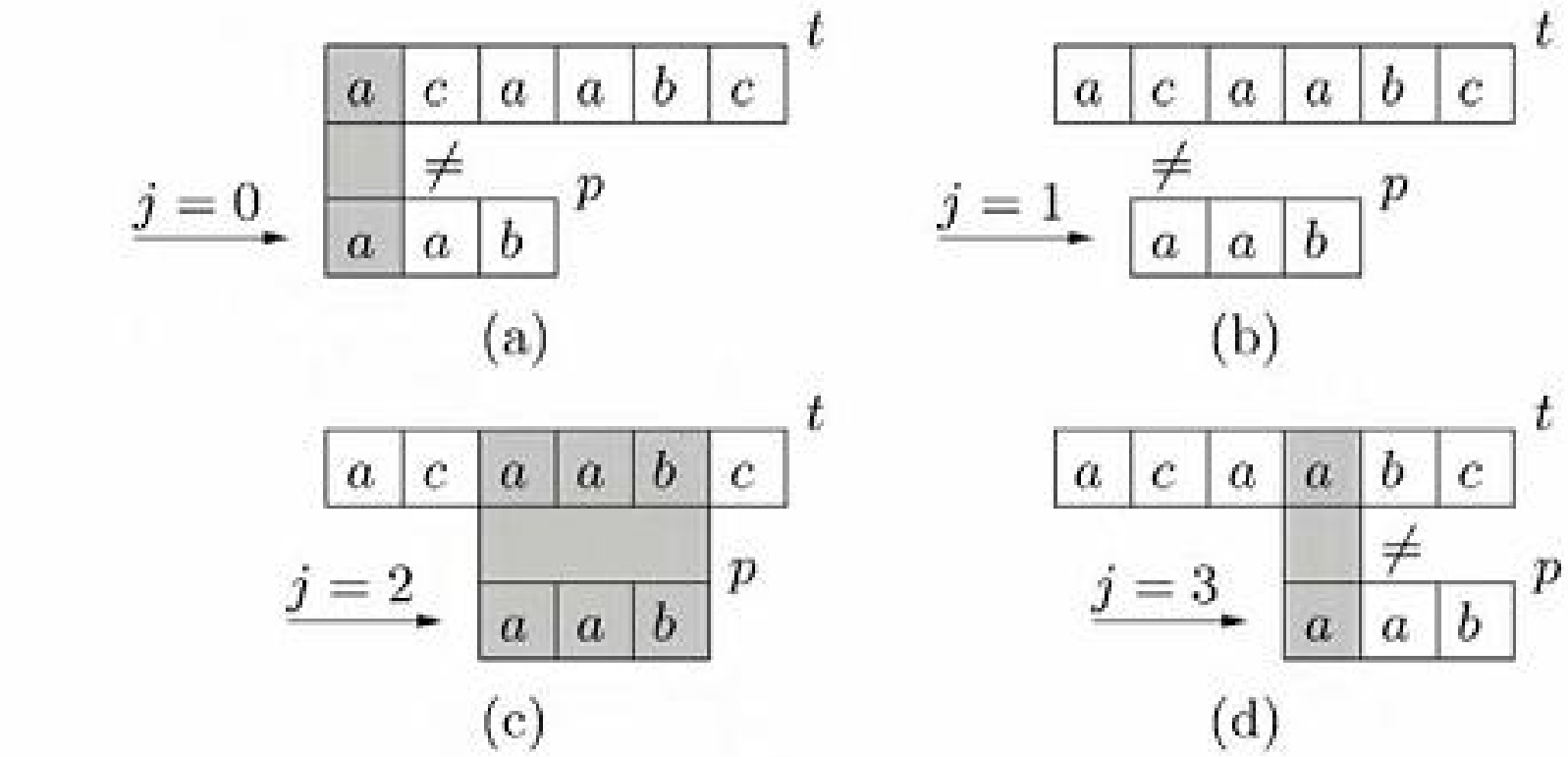
Vistos

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

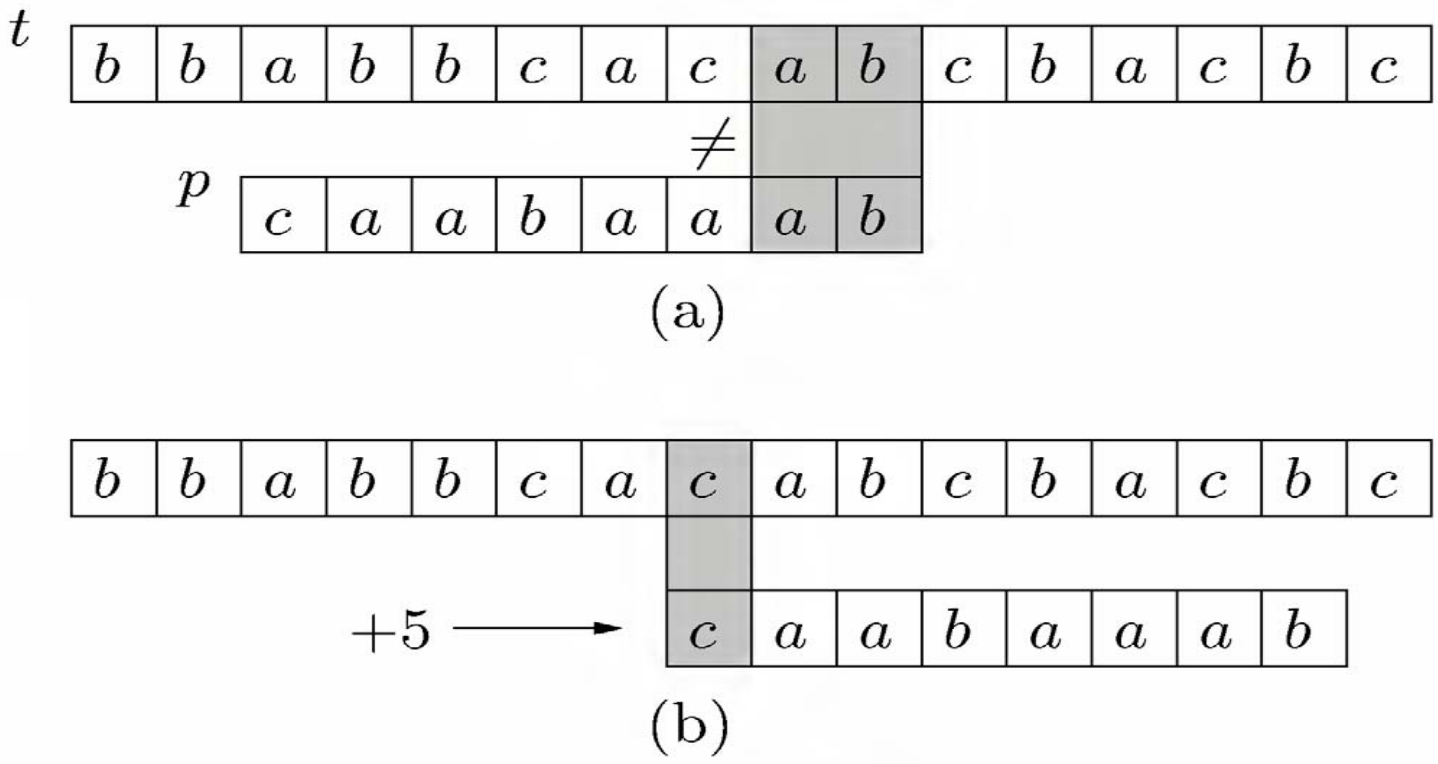
Algoritmo de Boyer-Moore

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

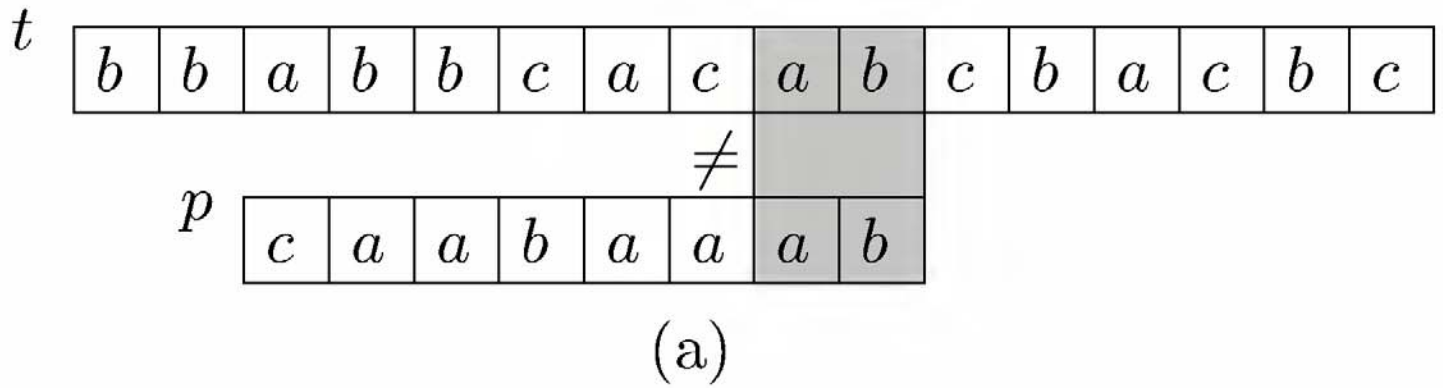
Ejemplo del inocente …



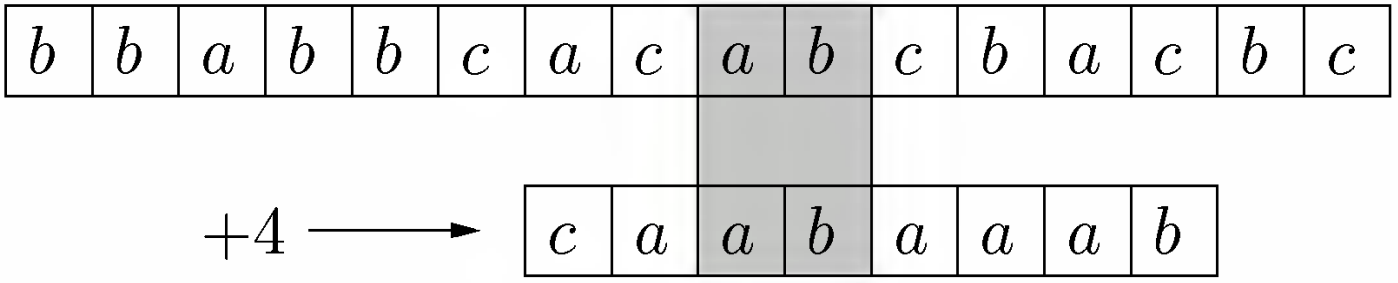
Boyer-Moore se comporta …



Boyer Moore se comporta ...



(a)



(c)

Complejidad de Boyer-Moore

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

Comparando complejidades

Caso peor …

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

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

Contenido

- El problema de string matching
- Algoritmos de propósito general
- **Árboles de sufijos**
- Otras aplicaciones de 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 ...

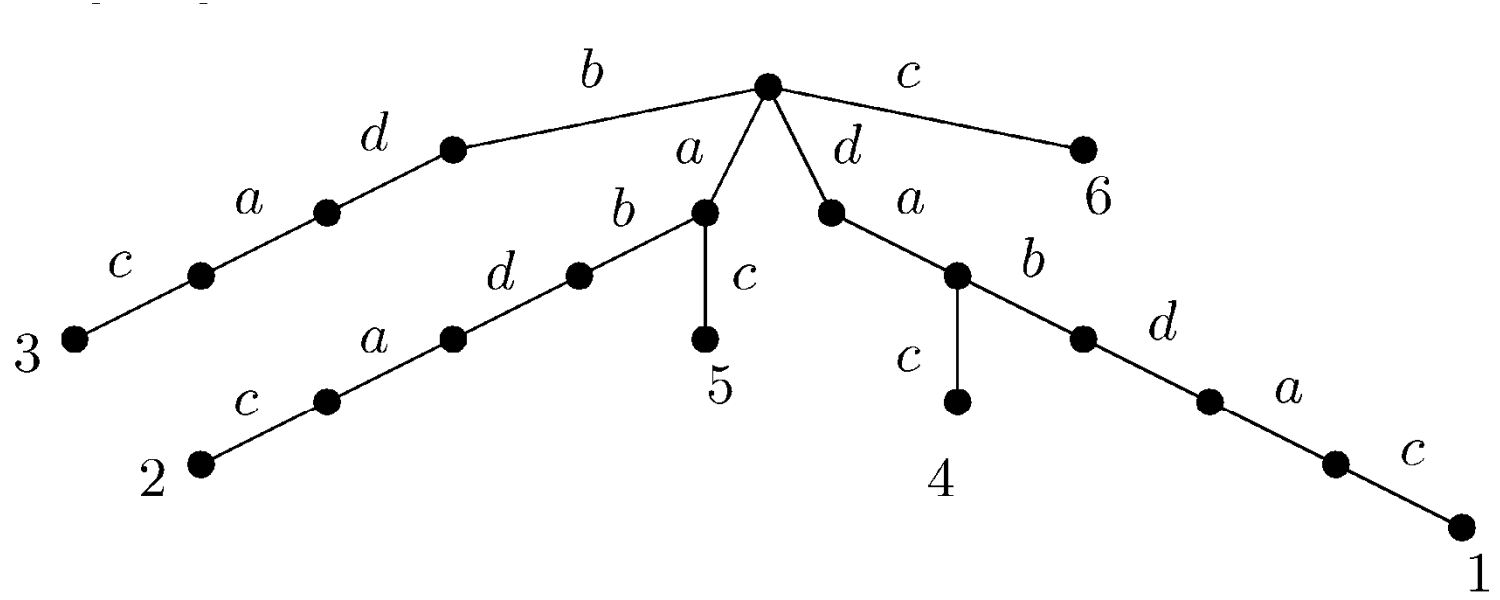


Fig. 4.5. A simple suffix tree for the text $t = dabdac$

Definición

- Dado un string $t=t_1 \ldots t_n$, un árbol sufijo simple es un árbol dirigido $T_t=(V,E)$ con una raiz r que cumple
  1. El árbol tiene n hojas con etiquetas 1, …, n. (La etiqueta i corresponde al sufijo $t_i \ldots t_n$)
  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 raiz a la hoja i tiene etiquetas $t_i \ldots t_n$

¿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($t = t_1 \ldots t_n$)

t’:= t\$
T:= árbol con raiz r
for i:=0 to n do
{insertar $t_i \ldots t_n$\$ en el árbol T}
Empezando en la raiz buscar un camino máximo $t_i \ldots t_j$
terminando en el nodo x
Añadir al árbol los nodos correspondientes a $t_{j+1} \ldots t_n$\$ como
camino a partir de x con etiquetas $t_{j+1}, \ldots, t_n$, \$,
etiquetando la hoja con i

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

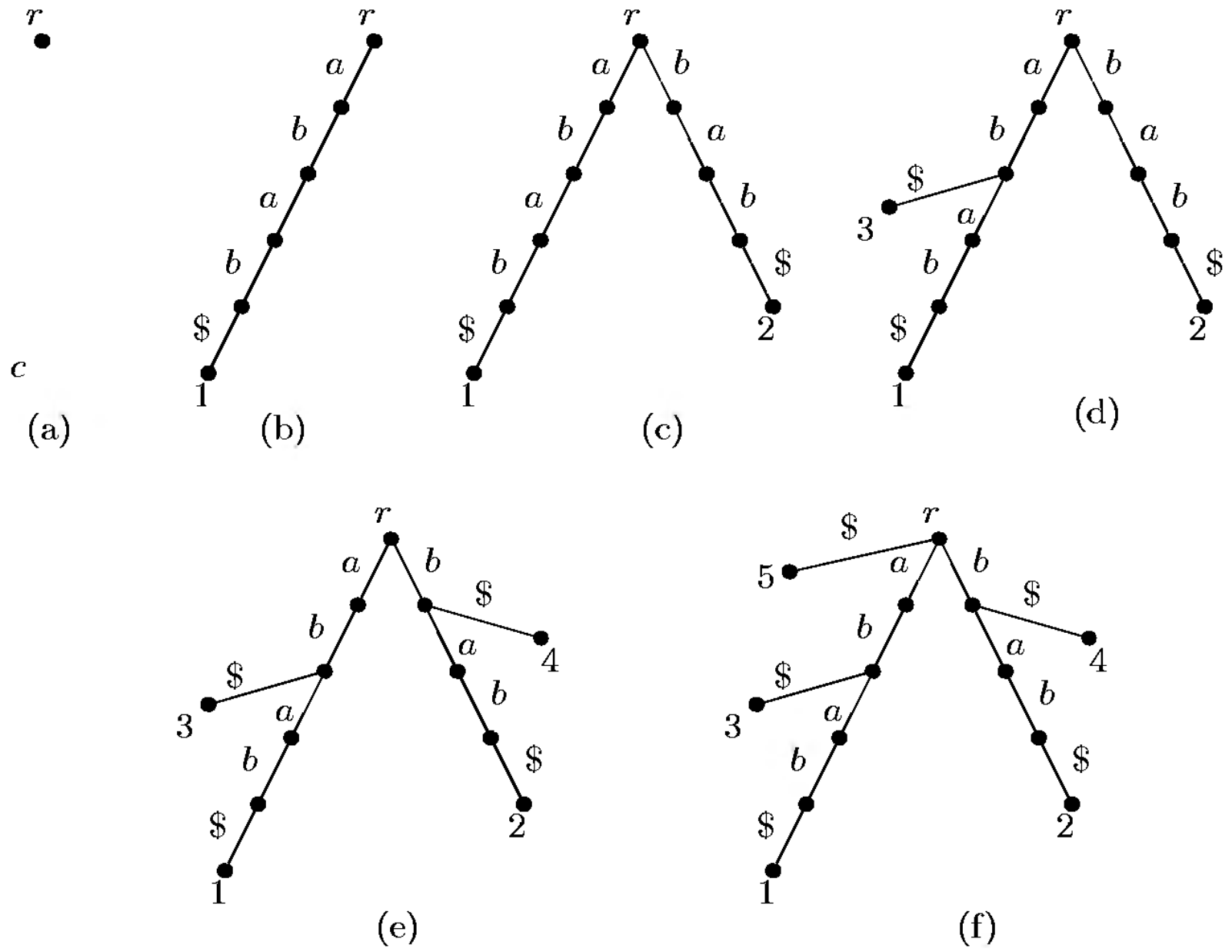


Fig. 4.6. The construction of a simple suffix tree for the text *abab*\$ using Algorithm 4.7

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

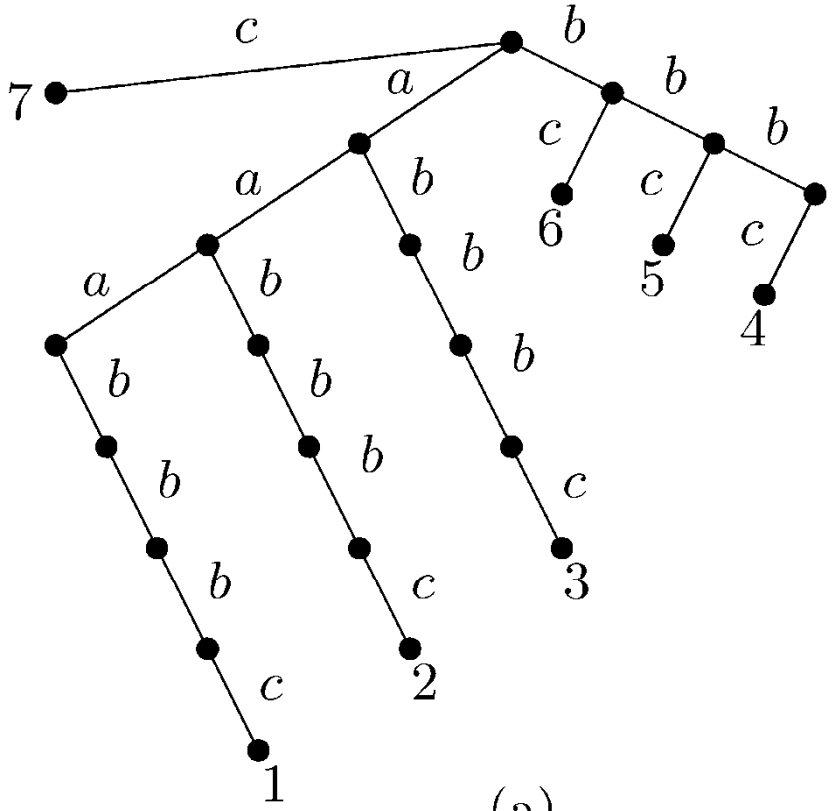
- Saber si p está en t cuesta tiempo O(|p|)
- Encontrar todas las ocurrencias de p en t puede ser muy largo dependiendo del tamaño del subárbol con raiz p

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

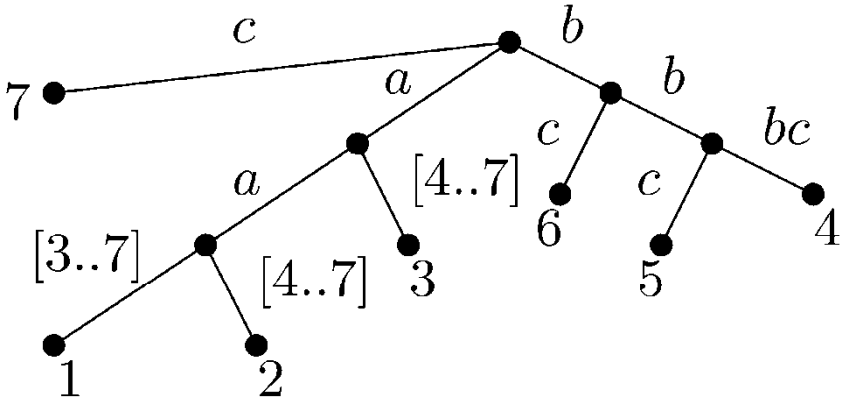
Árboles de sufijos compactos

- Podemos eliminar los nodos con un solo hijo … poniendo strings como etiquetas
- No tenemos que escribir las etiquetas

- En lugar de una etiqueta de k símbolos (ocupa $k \cdot \log(|\Sigma|)$) usamos los dos índices en el texto (ocupa $2 \cdot \log n$)


c
b
7
a
b
c
b
a
b
c
c
6
5
b
b
4
a
b
b
b
b
b
b
b
c
c
3
c
2
1

(a)


c
b
7
a
b
c
bc
a
[4..7]
c
[3..7]
6
[4..7]
3
5
4
1
2

(b)

Definición formal

- Dado un string $t=t_1 \ldots t_n$, un árbol sufijo compacto es un árbol dirigido $T_t=(V,E)$ con una raiz r que cumple
  1. El árbol tiene n hojas con etiquetas 1, …, n. (La etiqueta i corresponde al sufijo $t_i \ldots t_n$)
  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 raiz a la hoja i tiene etiquetas $t_i \ldots t_n$

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
- Codificar un nodo cuesta como máximo 2 logn bits
- En total O(n logn)

Notación en el algoritmo

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

Construcción compacta (1)

ÁrbolSufijosCompacto($t = t_1 . . . t_n$)

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

…

Construcción compacta (2)

ÁrbolSufijosCompacto($t = t_1 . . . t_n$)

…

{Comprimir las etiquetas largas}
For (x,y) arista do
    If label(x,y) >= log(n-|Σ|) then
        {Reemplazar la etiqueta por los números de posición}
        label'(y,z) = (Pos(y)+depth(x), Pos(y)+depth(x)+|label(x,y)|-1);

label=label'

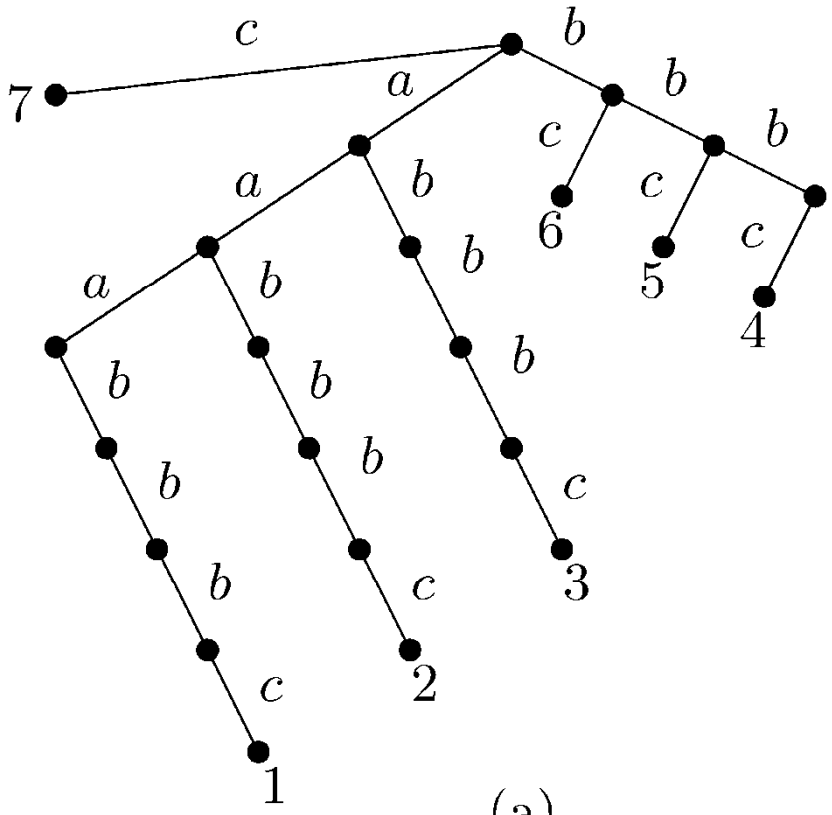
Resultado *T* (El árbol de sufijos compacto de $t_1 . . . t_n$\$)

Coste de la construcción

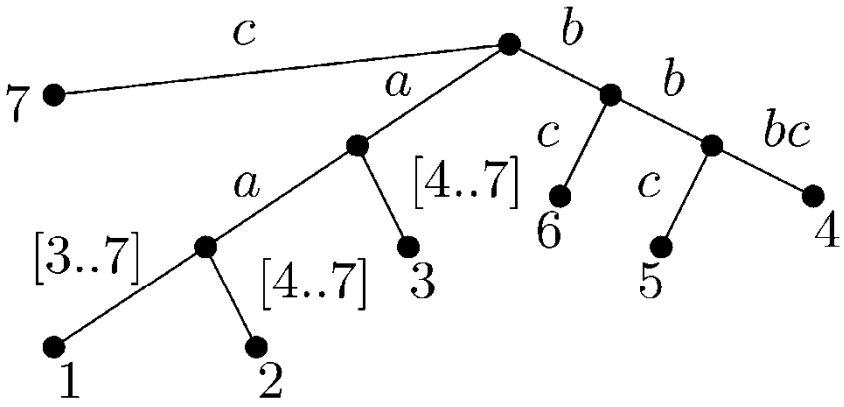
- Como el árbol sufijo inicial puede ser de tamaño $O(n^2)$ el tiempo es $O(n^2)$
- Hay métodos de construcción del árbol sufijo compacto sin pasar por el inicial que tardan O(n log n)
- Como el tamaño del árbol es O(n log n) esto es lo mínimo posible ...

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 raiz 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


c
b
7
a
b
c
b
a
b
c
c
6
5
4
a
b
b
b
b
b
b
b
b
c
b
b
c
c
3
2
1

(a)


c
b
7
a
b
c
bc
a
[4..7]
c
6
5
4
[3..7]
[4..7]
3
1
2

(b)

StringMatching(1)

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

(1) T:= ÁrbolSufijosCompacto (t);

(2) {Inicialización}
x:= raiz(T); {vértice actual}
i:=1; {posición actual en *p*}
found:=false;
possible:=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:= $t_k \ldots t_{l'}$

{leer la parte necesaria de la etiqueta}

borrar(v,U)

StringMatching(3)

While not found and possible

….

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

If (p_i … p_m no es prefijo de mylabel) and(mylabel no es prefijo de p_i … p_m) then

possible := false {*p* no aparece en *t*}

else if mylabel es prefijo de p_i … p_m then

x:=v

i:=i+|mylabel|

else {p_i … p_m es prefijo de mylabel}

x:=v

found := true

(4) if found then

Calcular el conjunto I de las etiquetas de hojas en el subárbol de raiz 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 $p=p_1 \ldots p_m$ 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 no se hubiera comprimido), a un fragmento de tamaño $\Omega(\log n)$. Luego eso sólo puede ocurrir $m/\log n$ veces
- Leer cada fragmento comprimido cuesta como mucho $|\Sigma|\log n$

Complejidad

- 3(b) cuesta tiempo m en total
- Atravesar el subárbol encontrado cuesta O(k) si p ocurre k veces en t

- El algoritmo completo cuesta

  $O(n \log n + m |\Sigma| + k)$

Resumen string matching

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

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

Resumen …

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

Resumen

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

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

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

Lo mejor: preprocesamiento de texto (árboles de sufijos)

Contenido

- El problema de string matching
- Algoritmos de propósito general
- Árboles de sufijos
- **Otras aplicaciones de árboles de sufijos**
- Vectores de sufijos

A continuación …

- Otras aplicaciones de árboles sufijos
  - El problema del substring
  - Substring común más largo
  - Solapes (Overlaps)
  - Repeticiones (exactas y maximales)

El problema del substring

- Entrada: Un patrón p= p_1 … p_m y N textos t_1,…, t_N
- Salida: El conjunto de textos en los que aparece el patrón (es decir, I $\subseteq$ {1,…, N} tal que i$\in$ I sii p es un substring de t_i)

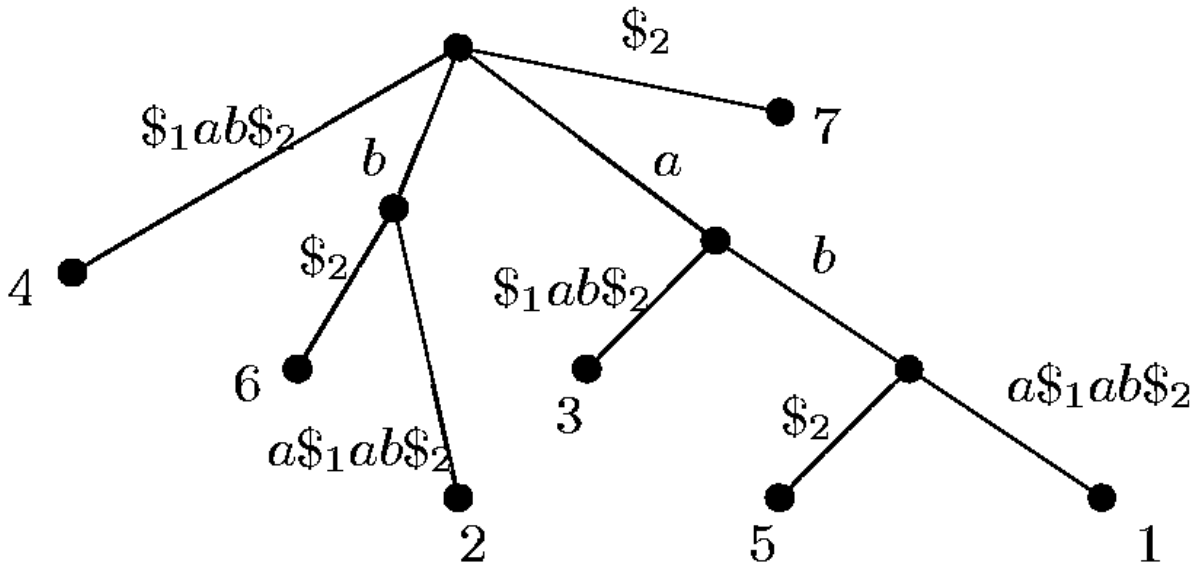
- Utilidad: para búsqueda en una base de datos de DNA

Una solución con árboles sufijos

- Construimos un árbol sufijo compacto para $t_1\$_1 \ldots t_N\$_N$, donde $\$_1, \ldots, \$_N$ son N símbolos distintos que no aparecen en los textos

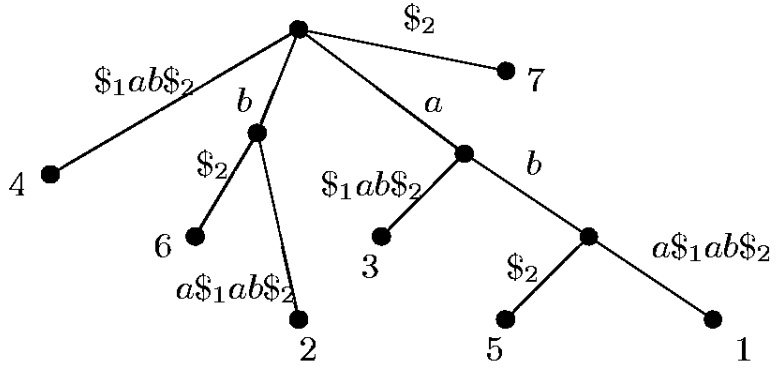
Ejemplo

t_1=aba t_2=ab



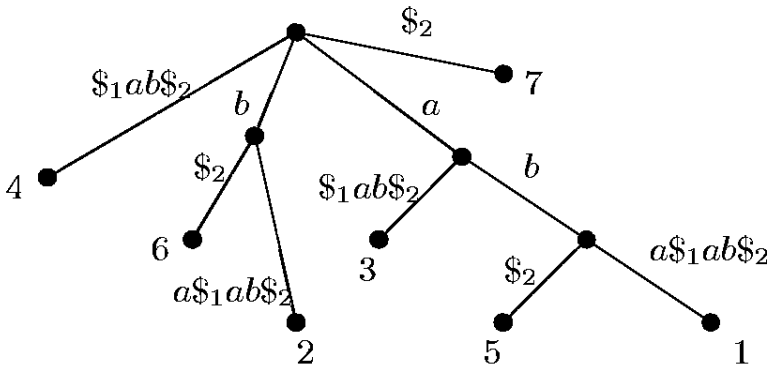
Pero

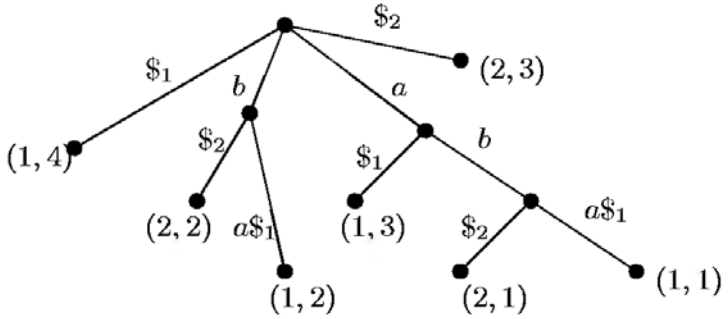
- $\$_1,\ldots,\$_N$ sólo aparecen en las etiquetas que llevan a una hoja … (porque cada $\$_i$ sólo aparece una vez en los textos)



- Podemos quitar todo lo que sigue al primer $ de cada etiqueta (es decir, cambiar cada $u\$_iv$ por $u\$_i$)

Ejemplo





También reetiquetamos la posición en $t_1\$_1 \ldots t_N\$_N$ con (i,j) para indicar “la posición j dentro de t_i“

Cuánto cuesta construir el árbol generalizado

- Sea n la longitud total de los textos (es decir, la longitud de $t_1\$_1 \ldots t_N\$_N$)
- Construir el árbol inicial cuesta O(n log n)
- La modificación de las etiquetas que van a las hojas y los índices de las hojas cuesta O(n)

Coste del problema del substring

- Hay que construir el árbol anterior
- Encontrar el nodo x del árbol que corresponde al patrón
- En el subárbol de x, encontrar todos los i para los que hay una hoja (i,j)
- $O(n \log n + m \cdot |\Sigma| + k)$

  k es el número total de ocurrencias

Substring común más largo

- Si tenemos una base de datos de DNA de organismos parecidos, encontrar strings comunes a todos ellos puede ser importante
- Si un string aparece en todas quiere decir que ha mutado muy poco y por tanto es importante para la supervivencia del organismo

Substring común más largo

Problema de optimización

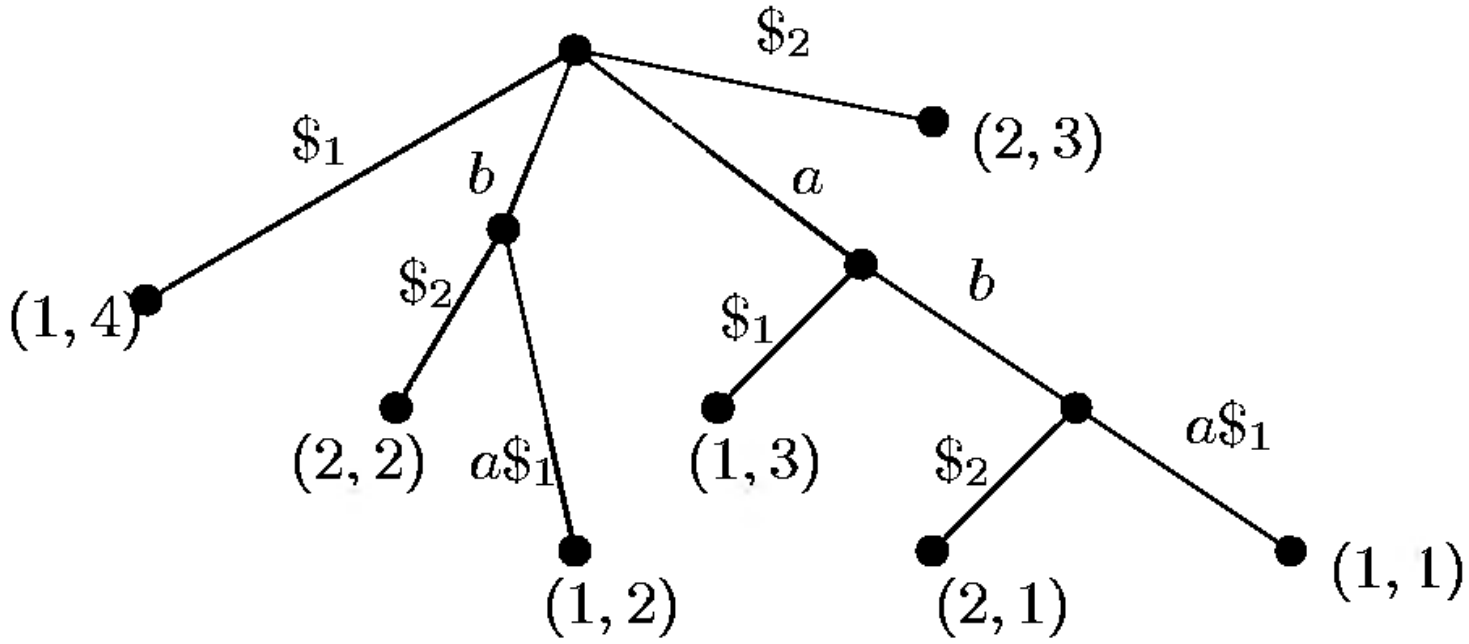
- Entrada: Un conjunto de textos $T = \{t_1, \ldots, t_N\}$
- Soluciones aceptables: Todos los strings x que son substrings de t_i para todo i
- Coste de una solución x: |x|
- Objetivo de optimización: Maximizar

Solución con árboles sufijos

- Construimos un árbol general para para $t_1,\ldots, t_N$ (es decir, el de $t_1\$_1\ldots t_N\$_N$)
- Cada vértice interior x lo etiquetamos con la lista de los i tal que hay una hoja (i,j) en el subárbol de x
- Buscamos las etiquetas {1,…, N} que corresponden a sufijos más largos
- Etiquetamos cada nodo interno con su profundidad (número de letras desde la raíz)

Recordatorio: árbol sufijo generalizado

t_1=aba t_2=ab

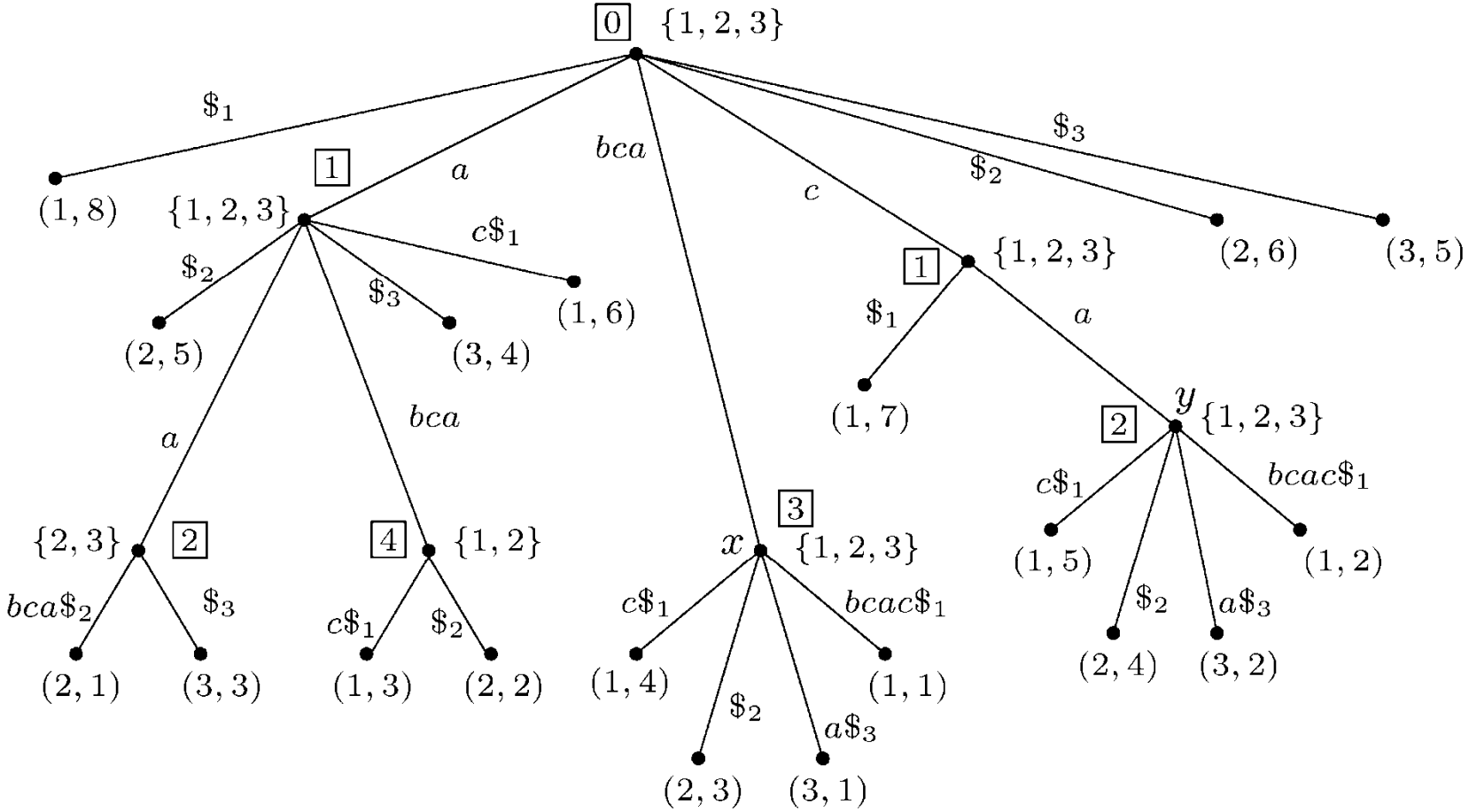


Solución con árboles sufijos

- Construimos un árbol general para para $t_1,\ldots, t_N$ (es decir, el de $t_1\$_1\ldots t_N\$_N$)
- Cada vértice interior x lo etiquetamos con la lista de los i tal que hay una hoja (i,j) en el subárbol de x
- Buscamos las etiquetas {1,…, N} que corresponden a sufijos más largos

Ejemplo

bcabcac, aabca, bcaa



Algorithm 4.10 Computation of the longest common substring

Input: A sequence $t_1, \ldots, t_N$ of N strings over an alphabet Σ.

1. Construct the generalized suffix tree T for $t_1, \ldots, t_N$.
2. Label every inner vertex x of the suffix tree with a subset $M(x) \subseteq \{1, \ldots, N\}$, such that $i \in M(x)$ if and only if there exists a leaf labeled (i, j) in the subtree rooted at x for some arbitrary j, i.e., if a suffix of t_i is contained in this subtree.
3. Between all inner vertices of T with label $\{1, \ldots, N\}$, find a vertex $x_{\max}$ with maximum string depth, i.e., with $\mathit{depth}(x_{\max}) = \max\{\mathit{depth}(x) \mid M(x) = \{1, \ldots, N\}\}$.
4. Compute $\alpha_{\max}$ as the label of the path from the root of the suffix tree to $x_{\max}$.

Output: The longest common substring $\alpha_{\max}$ of $t_1, \ldots, t_N$.

¿Por qué funciona?

- Cada nodo etiquetado {1, …, N} corresponde al prefijo de un sufijo (es decir, un substring) para todos los t_1,…, t_N
- Cuanto mayor sea la profundidad más largo es el substring

¿Cuánto tarda?

- Construir el árbol general

  O(n log n)
- Etiquetar cada x con los i tal que hay una hoja (i,j) en el subárbol de x

  $O(n(N+|\Sigma|)N + n \mid \Sigma \mid N)$
- Etiquetar cada nodo con la profundidad

  O(n log n)

TOTAL $O(n(\log n + N (N+|\Sigma|)))$

Generalizaciones

- Calcular el substring común más largo de al menos k de los N textos
- Calcular la posición del substring común más largo
- Todos los substrings comunes de una longitud dada l
  - Muy útil para testear la contaminación de DNA (N=2)

Cálculo de Overlaps

- El *solape* de s y t es el string y más largo que cumple s=xy, t=yz
- Se trata de calcular los solapes de cada par de textos de entre t_1,…, t_N
- Se usa mucho para secuenciación de ADN

Fuerza bruta

- Si la suma de las longitudes de $t_1,\ldots, t_N$ es n

- Tiempo $\Sigma_{i,j} \min(|t_i|,|t_j|)^2$
- Por ejemplo si todos los textos son casi iguales (de longitud alrededor de n/N) esto es $O(n^2)$

Usando árboles de sufijos

- Lo podemos hacer en tiempo

  $O(n(\log n + |\Sigma| + N))$

- Para secuenciación, estamos hablando de n’s enormes …

Idea del algoritmo

- Para buscar el overlap de t_i y t_j buscamos un sufijo de t_i que sea prefijo de t_j, es decir, seguimos el camino t_j en los sufijos de t_i
- Busco a la vez el overlap de todos los t_1,…, t_N con un t_j

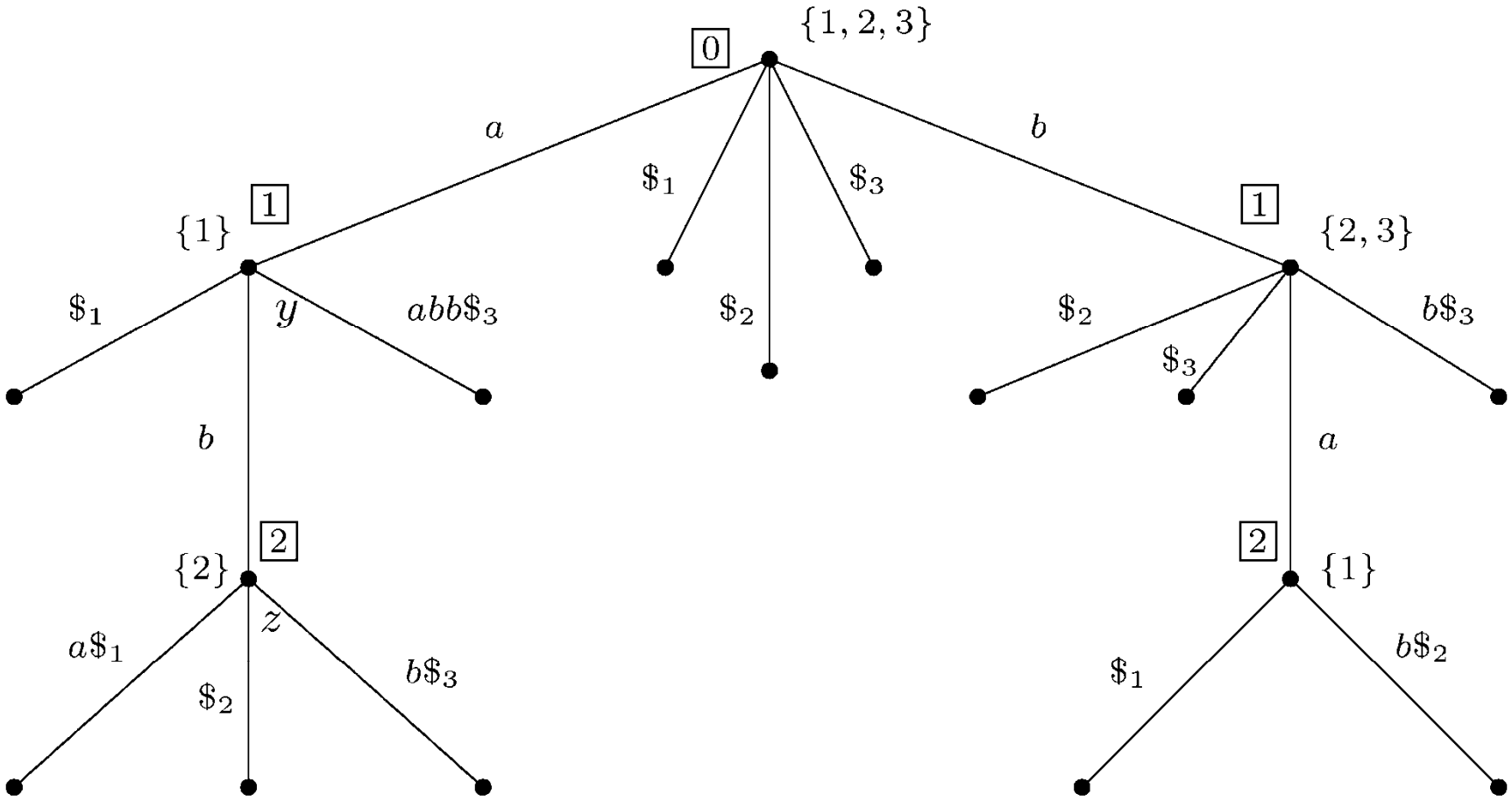


Fig. 4.10. The generalized suffix tree for computing all pairwise overlaps of the strings *aba*, *bab*, and *aabb* using Algorithm 4.11

Algorithm 4.11 Computation of all pairwise overlaps

Input: A sequence $t_1, \ldots, t_N$ of N strings over an alphabet Σ.

1. Construct the generalized suffix tree T for $t_1, \ldots, t_N$ with root r.
2. Label each inner vertex x of the suffix tree with a subset $L(x) \subseteq \{1, \ldots, N\}$, such that $i \in L(x)$ holds if and only if x is incident to an edge with label $\$_i$.
3. **for** $j := 1$ **to** N **do**
   $x := r$
   while x is no leaf **do**
   for all $i \in L(x)$ **do**
   if $depth(x) < \min\{|t_i|, |t_j|\}$ **then**
   $u(t_i, t_j) := depth(x)$
   $U(t_i, t_j) := pathlabel(x)$
   $x :=$ child of x on the path with label $t_j\$_j$

Output: The overlaps $Ov(t_i, t_j) = U(t_i, t_j)$ and their lengths $ov(t_i, t_j) = u(t_i, t_j)$, for all $i, j \in \{1, \ldots, N\}$.

Tiempo …

- Construir el árbol O(n log n)
- Calcular los L(x) O(n(N+|Σ|))
- Paso 3: en total todos los L(x) tienen N valores como mucho, y todos los caminos a seguir tienen longitud total n, luego

  O(n(N+|Σ|))

TOTAL: O(n(log n+ N+|Σ|))

Repeticiones

- Muy usadas para modelizar y tratar una secuencia de ADN
- Se usan repeticiones exactas y aproximadas, con solape o sin él
- Vamos a usar los árboles de sufijos para encontrar repeticiones exactas maximales (permitiendo solape)

Definición

- Sean t y p dos strings, $t=t_1 \ldots t_n$, $p=p_1 \ldots p_m$
- Añadimos principio y final a t, $t_0=\$$, $t_{n+1}=\$$

- p es una **repetición exacta en t** si existen i,j distintos que cumplen

  $p=t_{i+1} \ldots t_{i+m}=t_{j+1} \ldots t_{j+m}$

- p es una repetición exacta **maximal** en t si además $t_i \neq t_j$ y $t_{i+m+1} \neq t_{j+m+1}$

Hoy sobreentendemos **exacta**

Problema

- Entrada: Un string $t=t_1 \ldots t_n$
- Salida: El conjunto de todas las repeticiones exactas maximales en t

Observaciones

- Hay como máximo n-1 repeticiones maximales en $t=t_1 \ldots t_n$
- Si p es una repetición maximal en t, seguir el camino p en el árbol compacto de sufijos de t lleva a un nodo interior (es decir, un nodo con dos hijos)
- Puede haber varias repeticiones maximales empezando en el mismo sitio

  t= a**abc**b**ab**ac**abc**c

Observaciones …

- ¿Cómo distingo las repeticiones maximales de las no maximales en cabcab ?
- b, ab, cab llevan a nodos internos del árbol compacto de sufijos …

Caracterización

- El **símbolo izdo** de una hoja h es t_{i-1} tal que el camino de la raiz a la hoja es $t_i \ldots t_n$
- Un nodo x es "**left-diverse**" si dos hojas de x tienen símbolos izdos diferentes
- p es una repetición maximal sii el camino p lleva a un nodo left-diverse

Algoritmo

- Construir el árbol de sufijos compacto para t
- Encontrar el símbolo izdo de cada hoja
- Etiquetar de las hojas hacia la raiz los left-diverse

- Coste O(n log n), falta recorrer los caminos a left-diverse

- Una representación compacta de las repeticiones maximales: borrar los nodos que no son left-diverse (tamaño O(n log n), cuando puede haber $O(n^2)$ repeticiones maximales)

Contenido

- El problema de string matching
- Algoritmos de propósito general
- Árboles de sufijos
- Otras aplicaciones de 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=ababbabbb

Definición

- El vector de sufijos de un string s es

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

tal que el orden alfabético de los sufijos es $s[j_1,n] < s[j_2,n] < \ldots < s[j_n,n]$

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. $I := \{\mathcal{A}(t)[i], \mathcal{A}(t)[i+1], \ldots, \mathcal{A}(t)[j]\}$.

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

Coste: O(m log n +k) 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)

- Veamos las ideas prales …

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 $A = (A(1), \ldots, A(n))$ of integers from the range $\{0, \ldots, k\}$.

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 $B = (B(1), \ldots, B(n))$.

O(n+k)

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 $A = (a_1, \ldots, a_n)$ 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)

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, y si no es suficiente se cambia el alfabeto (cada tres letras viejas es una nueva) y se hace llamada recursiva
- Construye A^0 usando $A^{1,2}$
- Mezcla A^0 y $A^{1,2}$

- Coste O(n log n)

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 $0 \leqslant i \leqslant n+1$.
   c) Let $k_i = \max\{k \leqslant n+1 \mid k \equiv i \pmod 3\}$, for $i \in \{1, 2\}$;
   let $l_i := |\{k \leqslant n+1 \mid k \equiv i \pmod 3\}|$, 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 $0 \leqslant i \leqslant n+1$ such that $i \not\equiv 0 \pmod 3$.
   b) Sort the triples t_i, for all $i \leqslant \max\{k_1, k_2\}, i \not\equiv 0 \pmod 3$, 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 $1 \leqslant i \leqslant n$ 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 $\mathcal{A}(s)$.
