Tries, Suffix Tries & Suffix Arrays

Vicente Errázuriz

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Santiago, Chile

Motivación

Definiciones

Un conjunto finito de símbolos Σ se conoce como un **alfabeto**

Una concatenación de símbolos de Σ se conoce como una **palabra**

El conjunto de todas las posibles palabras que se pueden formar con Σ se escribe como Σ^*

Teniendo un conjunto finito de n palabras finitas $\mathbf{S} \subset \Sigma^*$

Queremos poder responder la pregunta:

$$\text{Dado } w \in \Sigma^*, \text{ ¿} w \in \mathbf{S}\text{?}$$

Dado $w \in \Sigma^*$, ¿$w \in \mathbf{S}$?

Una forma sencilla sería comparar w con cada elemento de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(n)$

Podríamos construir un ABB con los elementos de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(n \log n)$
Y luego consultar el ABB por w.
$\rightarrow$ Esto demoraría $O(\log n)$
$\Rightarrow$ En total $O(n \log n)$

¿No estamos olvidando nada?

La comparación de `Strings` no es $O(1)$

Podemos comparar w con cada elemento de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(k \cdot n)$

Podemos construir un ABB con los elementos de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(k \cdot n \log n)$
Y luego consultar el ABB por w.
$\rightarrow$ Esto demoraría $O(k \cdot \log n)$
$\Rightarrow$ En total $O(k \cdot n \log n)$

Con k el largo de la palabra más larga de $\mathbf{S}$

Extendamos el problema

¿Qué pasa si queremos saber si más de una palabra pertenece a $\mathbf{S}$?

Es decir:

Dado un conjunto $\mathbf{W} \subset \Sigma^*$ de m palabras ($m < n$)

$$\mathbf{W} = \{w_1, \ldots, w_m\}$$

Para cada $1 \leq i \leq m$, ¿$w_i \in \mathbf{S}$?

Para cada $1 \leq i \leq m$, ¿$w_i \in \mathbf{S}$?

Podemos comparar cada $w \in \mathbf{W}$ con cada elemento de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(k \cdot m \cdot n)$

Podemos construir un ABB con los elementos de $\mathbf{S}$
$\rightarrow$ Esto demoraría $O(k \cdot n \log n)$
Y luego consultar el ABB por cada $w \in \mathbf{W}$.
$\rightarrow$ Esto demoraría $O(k \cdot m \cdot \log n)$
$\Rightarrow$ En total $O(k \cdot n \log n)$

¿Se podrá hacer de otra forma en que el tiempo de consulta no dependa de n?

Ya hemos visto... Radix Sort

Radix Sort era un algoritmo que se abstraía de los valores de los números y en lugar de eso trabajaba con los dígitos en orden.

Al hacer esto, el tiempo necesario para ordenar n elementos dependía menos de n y más de el tamaño de los distintos elementos.

Intentemos usar este enfoque para construir una estructura que nos permita trabajar en tiempos independientes del tamaño de $\mathbf{S}$

Ejemplo

$$\Sigma = \{\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\ \}$$

$$\mathbf{S} = \{\ 1529,\ 2519,\ 2591,\ 2915,\ 5192,\ 5291,\ 9215\ \}$$

[Solución en la pizarra]

Forma General

Dado un alfabeto Σ y un conjunto finito de palabras finitas $\mathbf{S} \in \Sigma^*$

Podemos construir un árbol tal que si tomamos cualquier palabra $s \in \mathbf{S}$, y elegimos un i $(1 \leq i < |s|)$ existe siempre un nodo en el nivel i con el símbolo $s[\ i\ ]$ que es padre de un nodo en el nivel $i+1$ con el símbolo $s[\ i+1\ ]$.

Cada nodo tiene un único padre, y en el nivel 0 existe un único nodo sin símbolo que es padre de todos los nodos del nivel 1.

Éste árbol se conoce como el **Radix Tree** o **Trie** de $\mathbf{S}$

Inserción en un Trie

A continuación el pseudocódigo de la inserción de una palabra w en un Trie T

```
1  procedure Trie-Insert(T, w)
2      n ← raiz[T]
3      foreach símbolo i in w do
4          if existe un x ∈ hijos[n] tal que simbolo[x] = i then
5              n ← x
6          else
7              x ← nuevo nodo con símbolo i
8              agregar x a hijos[n]
9              n ← x
10     marcar n como nodo de término
```

Búsqueda en un Trie

A continuación el pseudocódigo de la búsqueda de una palabra w en un Trie T

```
1  procedure Trie-Search(T, w)
2      n ← raiz[T]
3      foreach símbolo i in w do
4          if existe un x ∈ hijos[n] tal que simbolo[x] = i then
5              n ← x
6          else
7              return false
8      if n está marcado como nodo de término then
9          return true
10     return false
```

Complejidad

En la práctica, en un **Trie** $\mathbf{T}$ cada nodo es un arreglo de $|\Sigma|$ punteros a los nodos inferiores, por lo que revisar si un nodo es hijo de otro es $O(1)$

Esto significa que tanto la busqueda como la inserción de una palabra w son $O(|w|)$!!

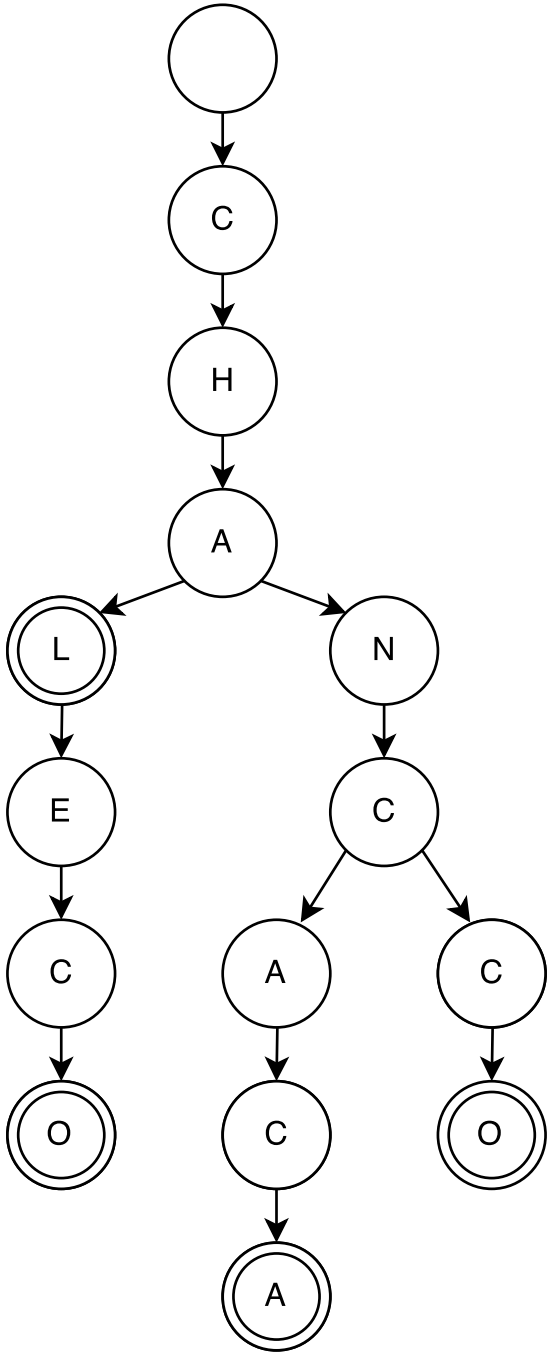
Y también significa que la cantidad de espacio que ocupan es...

no nos preocupemos de eso por ahora :s

Ejemplo

$\Sigma = \{$ A, C, H, L, N, O $\}$

$\mathbf{S} = \{$ CHANCHO, CHAL, CHANCACA, CHALECO $\}$


C
H
A
L
N
E
C
C
A
C
O
C
O
A

Volviendo al problema

Eso significa que para resolver nuestro problema, podemos construir un **Trie** sobre $\mathbf{S}$.
$\rightarrow$ Esto demoraría $O(n \cdot k)$
Y luego consultarlo por cada una de las palabras $w \in \mathbf{W}$.
$\rightarrow$ Esto demoraría $O(m \cdot k)$
$\Rightarrow$ En total $O(n \cdot k)$

Ok, ahora, ¿para qué nos interesa resolver este problema?

Un nuevo desafío

Queremos encontrar el substring **más largo** que aparece **más de una vez** dentro de una palabra $w \in \Sigma^*$ de n símbolos

Podríamos generar todos los substrings de w y luego...

Espera, eso es una locura: w tiene

$$\sum_{i=1}^{n} i = \frac{n^2 + n}{2}$$

substrings.

Seamos más astutos

Definición

Para una palabra $w \in \Sigma^*$ de n símbolos

$$w = \{ \sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n \}$$

$$\sigma_i \in \Sigma, \ \forall \ 1 \leq i \leq n$$

El i-ésimo **sufijo** de w es el substring que s comienza en σ_i y termina en σ_n. Se dice que s es **sufijo** de w.

El i-ésimo **prefijo** de w es el substring p que comienza en σ_1 y termina en σ_i. Se dice que p es **prefijo** de w.

Substrings

Teorema

Cada substring de una palabra $w \in \Sigma^*$ de n símbolos es **prefijo** de *algún* **sufijo** de w.

Demostración

Sea φ el substring de w que comienza en σ_i y termina en σ_j

Sea s el i-ésimo sufijo de w: este empieza en σ_i y termina en σ_n

φ es el $(j - i + 1)$-ésimo prefijo de s.

Podemos aprovechar esto!

Suffix Trie

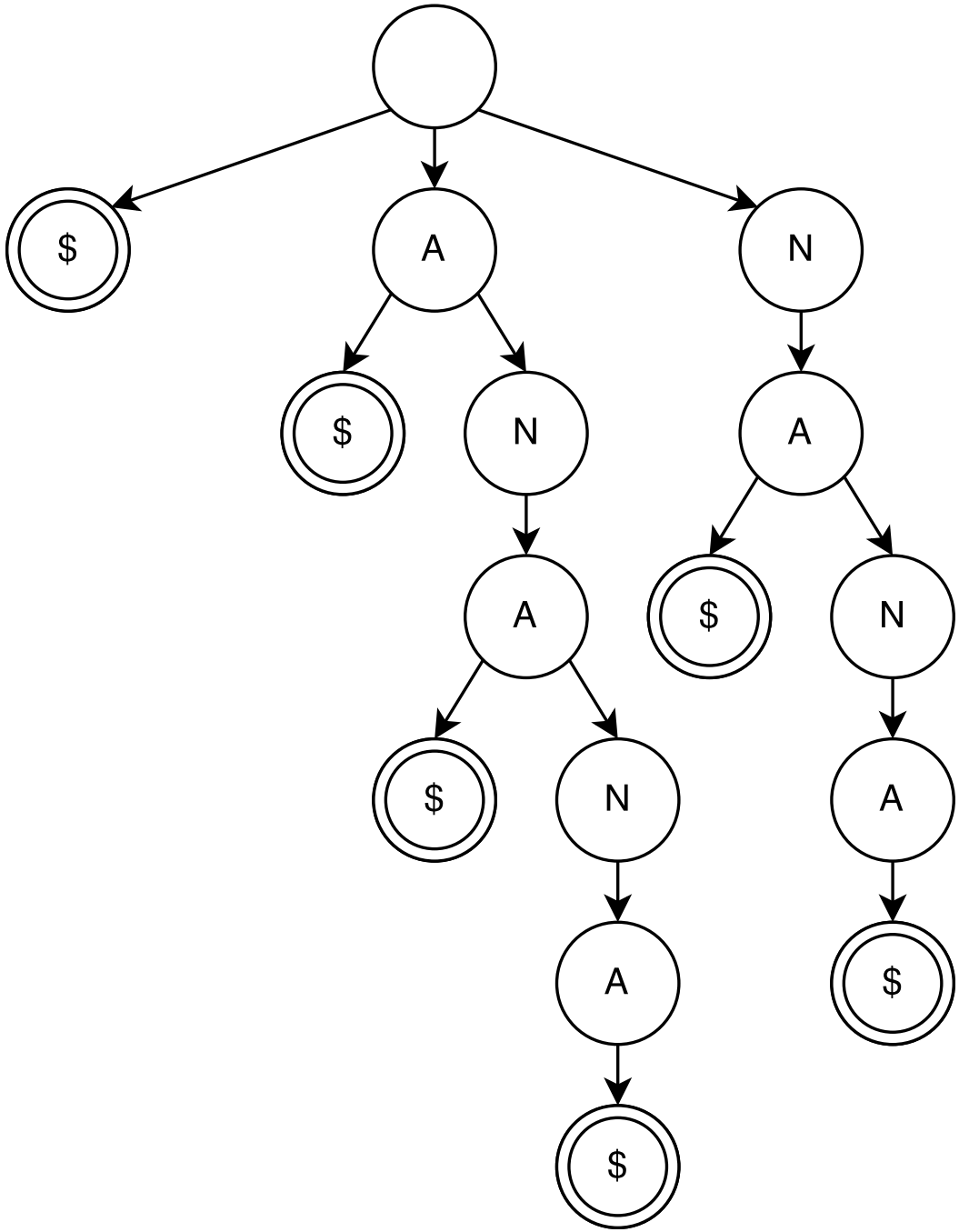
Vamos a construir un **Trie** con los sufijos de w.

$$w = \text{ANANA}$$

Esto se conoce como el **Suffix Trie** de w

El estándar al trabajar con sufijos de una palabra es agregarle al final el símbolo $\$ \notin \Sigma$ el cual además cumple que

$$\$ < \sigma, \ \forall \ \sigma \in \Sigma$$


$
A
N
$
N
A
$
N
$
N
A
$
A
$

Suffix Trie

Podemos hacer una pequeña modificación a la rutina de inserción sin modificar su complejidad para facilitar la búsqueda en el Trie del substring que más se repite en w

[Solución en la Pizarra]

Suffix Trie

Con esto, tenemos que el substring **más largo** y que aparece **más de una vez** en w es $s =$ ANA

Y esto nos tomó solamente

$\sim O(|w|)$ la búsqueda dentro del árbol, y

$O(|w|^2)$ la construcción...

¿Cómo podemos arreglar esto?
$\rightarrow$ existe una estructura llamada **Suffix Tree** que soluciona todos estos problemas. Pero también podemos hacer esto de otra forma

Otro punto de vista

Teorema

En una lista ordenada $\mathbf{S}$ de palabras distintas, por cada **prefijo** p que exista en $\mathbf{S}$ hay un único intervalo de $\mathbf{S}$ que contiene **todas** las palabras que comienzan con p y **solamente** palabras que comienzan con p

Otro punto de vista

Demostración

Para el prefijo p, sea w_1 la menor palabra comienza con p, y w_2 la mayor. Sea x una palabra que esté entre w_1 y w_2. Esto significa que x es tal que

$$w_1 < x < w_2$$

x debe necesariamente comenzar con p, dado que sino sería $x < w_1$ ó $w_2 < x$

Por lo tanto, toda palabra que no comience con p está fuera del intervalo comprendido entre w_1 y w_2

Suffix Array

¿Qué pasa si $\mathbf{S}$ son los sufijos de una palabra w de n símbolos?

$$w = \text{ANANA}$$

ANANA$
NANA$
ANA$
NA$
A$
$

$\rightarrow$

$
A$
ANA$
ANANA$
NA$
NANA$

Esto es lo que se conoce como el **Suffix Array** de $\mathbf{S}$

LCP : Longest Common Prefix

Podemos usar un arreglo auxiliar al que llamaremos LCP para buscar el substring **más largo** que aparece **más de una vez** en w en tiempo $O(n)$

[Solución en la pizarra]

El prefijo que comparten las palabras de los indices 2 y 3 del arreglo es el más largo de todos, por lo que es ese prefijo el substring que buscamos: ANA

Esto es imposible

Generar el **Suffix Array** de manera explícita en memoria es $O(n^2)$, por lo que queremos poder hacerlo de manera implícita. En lugar de tener el arreglo con todos los sufijos, tenemos un arreglo con todos los índices de los sufijos y ordenamos ese arreglo de acuerdo a los sufijos mismos.

Suffix Array

0		0	ANANA$		5	$		5
1		1	NANA$		4	A$		4
2	$\Leftrightarrow$	2	ANA$	$\rightarrow$	2	ANA$	$\Leftrightarrow$	2
3		3	NA$		0	ANANA$		0
4		4	A$		3	NA$		3
5		5	$		1	NANA$		1

Complejidad

Tener el **Suffix Array** de manera implícita en memoria significa que usa solamente $O(n)$ de memoria.

Construirlo toma $O(n^2 \log n)$ dado que el largo del sufijo más largo es n

Construir el LCP toma $O(n^2)$ por lo mismo

Consultar el LCP toma $O(n)$

Tanto la construcción del **Suffix Array** como de su LCP pueden hacerse ambos en un solo proceso que toma $O(n \log n)$

Pero en la práctica, el aporte del largo n del sufijo más largo no afecta tanto en el rendimiento de la estructura.