Cadenas

Isabel Pita.

Facultad de Informática - UCM

11 de noviembre de 2022

Cadenas

- Una *cadena* es una secuencia de caracteres. Las cadenas pueden ser muy largas...
- Tipos de datos para leer las cadenas:
  - `string`,
  - `char t[MAX]`. Carácter de finalización `'\0'`.
- Para transformar un `string` al tipo `char*` utilizar el método `c_str()`.
- Importante conocer las funciones de la librería para el tratamiento de cadenas. Estas funciones suelen tener complejidad del orden de la longitud de la cadena.

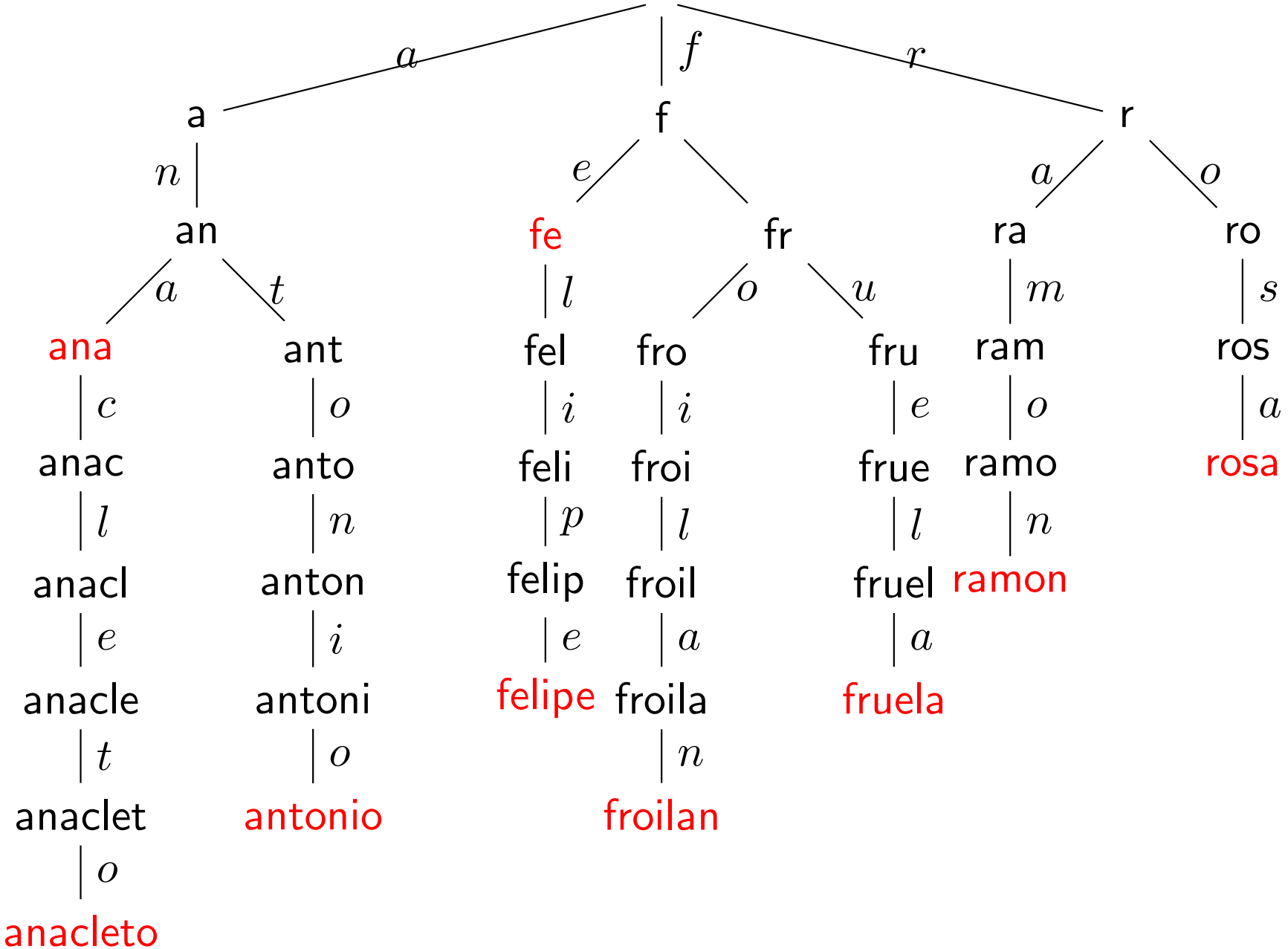
- Tratamiento de cadenas con programación dinámica.
- Diccionarios: Tries.
- *String matching*. Algoritmo KMP, o Suffix Array.
- Longest Repeated String, Longest Common Substring: Suffix Array.

Trie

- Estructura de datos eficiente como diccionario.
- Permite determinar si una palabra P, de longitud m pertenece a un diccionario ya creado en $\mathcal{O}(m)$.

- Cada nodo del árbol tiene tres atributos:
  - Número de prefijos de los que forma parte este nodo, `prefixes`.
  - Número de cadenas que terminan en este nodo, `words`.
  - Vector de punteros a nodos con tantas posiciones como letras del alfabeto, `child`.

Trie


a
f
r
n
e
a
o
an
fe
fr
ra
ro
a
t
l
o
u
m
s
ana
ant
fel
fro
fru
ram
ros
c
o
i
i
e
o
a
anac
anto
feli
froi
frue
ramo
rosa
l
n
p
l
l
n
anacl
anton
felip
froil
fruel
ramon
e
i
e
a
a
anacle
antoni
felipe
froila
fruela
t
o
n
anaclet
antonio
froilan
o
anacleto

Trie

Se proporciona el método `add` que permite crear el Trie. Los métodos para recorrer el Trie son particulares de cada problema y se deben añadir en cada caso a la estructura.

```
const int MAXN = 26;

class Trie {
    int prefixes;
    int words;
    std::vector<Trie *> child;

public:
  Trie():prefixes(0),words(0),child(MAXN, nullptr) {}

  ~Trie() {
    for (int i = 0; i < MAXN; ++i)
      delete child[i];
  }
```

Trie

```
  void add(const char *s) {
    if (*s == '\0') ++words;
    else {
      Trie * t;
      if (child[*s - 'a'] == nullptr) {
        t = child[*s - 'a'] = new Trie();
        t->prefixes = 1;
      } else {
        t = child[*s - 'a'];
        t->prefixes++;
      }
      t->add(s+1);
    }
  }
};
```

String matching

Problema: Dada una cadena T de longitud n y un patrón P de longitud m, encontrar las apariciones del patrón en el texto.

P = ab
T = aaabcaabacbaabbbcaabca

El algoritmo de *fuerza bruta* comprueba para cada posición de T si encaja el patrón. Tiene complejidad en el caso peor $\mathcal{O}(n * m)$.
Si el tamaño del patrón y del texto no son muy *grandes* el coste es aceptable.

P = aaaaaaaaab
T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Algoritmo KMP (Knuth-Morris-Pratt)

El algoritmo utiliza la información que se obtiene al comparar el patrón con el texto, para no comparar dos veces un carácter en T que haya unificado con un carácter en P.

P = abcabd
0 1 2 3 4 5

T = babcabcabc
0 1 2 3 4 5 6 7 8 9

Tabla que nos indica la posición del patrón a la que debemos retroceder. (El texto no retrocede nunca)

P	=	a	b	c	a	b	d	
b	=	−1	0	0	0	1	2	0
		0	1	2	3	4	5	6

Algoritmo KMP (Knuth-Morris-Pratt)

```
std::string T,P;
std::vector<int> b; //  back table
int n, m;   // n = length of T, m = length of P

void kmpPreprocess() { // before calling kmpSearch
    int i = 0, j = -1; b[0] = -1;
    while (i < m) {
        while (j >= 0 && P[i] != P[j]) j = b[j];
        ++i; ++j;
        b[i] = j;
    }
}
```

Complejidad $\mathcal{O}(m)$

P	=		a	b	c	a	b	d	
b	=	-1	0	0	0	1	2	0	
		0	1	2	3	4	5	6	

Algoritmo KMP (Knuth-Morris-Pratt)

```
void kmpSearch() {
    int i = 0, j = 0;
    while (i < n) {
        while (j >= 0 && T[i] != P[j])
            j = b[j]; // different reset j using b
        ++i; ++j;    // same, advance both pointers
        if (j == m) {
            printf("P is found at index %d in T\n",i-j);
            j = b[j];
        }
    }
}
```

Complejidad $\mathcal{O}(n)$ ($+ \mathcal{O}(m)$ del preprocesado)

P	=	a	b	c	a	b	d			
		0	1	2	3	4	5	6	7	8
T	=	a	b	c	a	b	c	a	b	d

Otras aplicaciones del algoritmo KMP

Dada una cadena encontrar la longitud mínima de una subcadena que se repite N veces para formar la cadena.

Ejemplos:

abababab - Longitud mínima 2

aaaa - Longitud mínima 1

abcde - Longitud mínima 5.

Algoritmo:

- Calcular el vector $\mathtt{b}$ del preprocesado de la cadena.
- Si alguna cadena se repite su longitud debe ser:

$$l = \textit{longitud de la cadena - último valor de la tabla}$$

- Si ($\textit{longitud de la cadena}\ \%l == 0$) es la cadena que queremos, en otro caso la longitud mínima es la longitud de la cadena.

Suffix Array

- Estructura de datos que permite resolver eficientemente los problemas :
  - *String matching*
  - *Longest Repeated Substring*. Subcadena más larga que se repite en una cadena.
  - *Longest Common Substring*. Subcadena más larga que se repite en varias cadenas.
- Un *Suffix Array* es un vector de enteros que almacena los índices de los sufijos de una cadena ordenados lexicográficamente.
- Debe añadirse un carácter al final de la cadena cuyo valor en el código ASCII sea menor que el de las letras de la cadena.

Suffix Array

i	Suffix
0	moviesemos$
1	oviesemos$
2	viesemos$
3	iesemos$
4	esemos$
5	semos$
6	emos$
7	mos$
8	os$
9	s$
10	$

i	SA[i]	Suffix
0	10	$
1	6	emos$
2	4	esemos$
3	3	iesemos$
4	7	mos$
5	0	moviesemos$
6	8	os$
7	1	oviesemos$
8	9	s$
9	5	semos$
10	2	viesemos$

Suffix Array

- Ordenar el vector con las funciones de la librería (`sort`) es muy costoso, porque cada comparación de cadenas tiene coste lineal respecto a la longitud n de la cadena. El coste de ordenar esta en $\mathcal{O}(n^2 \log n)$.
- Para mejorar el coste realizamos $\log n$ veces la ordenación, pero cada vez ordenamos únicamente según el prefijo de longitud $k = 1, 2, 4, 8, \ldots$ de cada sufijo.
- Cómo el rango de los valores a ordenar no es *muy grande*, utilizamos el método de ordenación *Radix sort*, que llama al método `CountingSort` para ordenar en tiempo $\mathcal{O}(n)$.
- El tiempo total de realizar las $\log n$ ordenaciones es $\mathcal{O}(n \log n)$.

Suffix Array

```
#define MAX_N 100010
std::string T;
int n;
int RA[MAX_N], tempRA[MAX_N];
int SA[MAX_N], tempSA[MAX_N];
int c[MAX_N];

void countingSort(int k) {
 int i, sum, maxi = std::max(300,n); // up to 255 ASCII
 memset(c,0,sizeof c);
 for (i = 0; i < n; ++i)
   ++c[i+k<n ? RA[i+k] : 0];
 for (i = sum = 0; i < maxi; ++i) {
   int t = c[i]; c[i] = sum; sum += t;}
 for (i = 0; i < n; ++i)
   tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
 for (i = 0; i < n; ++i)
   SA[i] = tempSA[i];
}
```

Suffix Array

```
// Construccion del suffix array
void constructSA() {
 int i, k, r;
 for (i = 0; i < n; ++i) RA[i] = T[i];
 for (i = 0; i < n; ++i) SA[i] = i;
 for (k = 1; k < n; k <<= 1) {
   countingSort(k);
   countingSort(0);
    tempRA[SA[0]] = r = 0;
   for (i = 1; i < n; ++i)
     tempRA[SA[i]] =
        (RA[SA[i]] == RA[SA[i-1]] &&
        RA[SA[i]+k] == RA[SA[i-1]+k]) ?
        r : ++r;
   for (i = 0; i < n; ++i)
     RA[i] = tempRA[i];
   if (RA[SA[n-1]] == n-1) break;
  }
}
```

Suffix Array. Aplicaciones

String matching. Encontrar las veces que se repite un patrón P de longitud m en un texto T de longitud n.

- Utilizar búsqueda binaria para localizar la aparición más a la izquierda de P en el vector SA.
- Utilizar búsqueda binaria para localizar la aparición más a la derecha de P en el vector SA.
- El número de veces que se repite el patrón es la diferencia entre los índices obtenidos en las búsquedas.
- Complejidad una vez construido SA: $\mathcal{O}(m \log n)$.

Suffix Array. Aplicaciones

Longest Common Prefix. Calcular la máxima longitud de un prefijo común a dos sufijos del vector SA.

- El prefijo común debe estar en posiciones consecutivas del vector SA.
- Comparar directamente los prefijos tiene un coste $\mathcal{O}(n^2)$, siendo n la longitud de la palabra.
- Se puede calcular en $\mathcal{O}(n)$, utilizando la posición inicial de los sufijos en lugar del vector SA.

Longest Common Prefix

```
int LCP[MAX_N];

void computeLCP() {
 int Phi[MAX_N];
 int PLCP[MAX_N];
 int i, L;
 Phi[SA[0]] = -1;
 for (i = 1; i < n; ++i)
  Phi[SA[i]] = SA[i-1];
 for (i = L = 0; i < n; ++i) {
  if (Phi[i] == -1) {PLCP[i] = 0; continue;}
  while (T[i+L] == T[Phi[i] + L]) ++L;
  PLCP[i] = L;
  L = std::max(L-1,0);
 }
  for (i = 0; i < n; ++i)
    LCP[i] = PLCP[SA[i]];
}
```

Suffix Array. Aplicaciones

Longest Repeated Substring. Subcadena más larga que aparece al menos dos veces en la cadena

- El valor de LRP es el máximo del vector LCP.

i	SA[i]	LCP	Suffix
0	10	0	$
1	6	0	emos$
2	4	1	esemos$
3	3	0	iesemos$
4	7	0	mos$
5	0	2	moviesemos$
6	8	0	os$
7	1	1	oviesemos$
8	9	0	s$
9	5	1	semos$
10	2	0	viesemos$

Suffix Array. Aplicaciones

Longest Common Substring. Subcadena más larga que aparece en varias cadenas.
Para dos cadenas:

- Concatenar las dos subcadenas, utilizando diferentes caracteres para controlar el final de cada una. Los caracteres utilizados deben ser anteriores a los caracteres de la cadena en el código ASCII.
- Calcular el SA y LCP de la cadena concatenada.
- Calcular el LCP máximo de los sufijos tales que un sufijo pertenece a una cadena y el siguiente a la otra.
- Para comprobar si un sufijo pertenece a una cadena se comprueba si su SA[i] esta en el rango que corresponde a la cadena en la cadena concatenada. Por ejemplo un sufijo está en la primera cadena si su SA[i] es menor que la longitud de la cadena