Hashing/Aho Corasick

Marcos Kolodny

FAMAF-UNC

July 23, 2019

Hashing

Llamamos función de hash a aquella que mapea un elemento de su dominio (de cualquier tamaño) a un elemento de tamaño específico contenido en la imagen.

Hashing

Llamamos función de hash a aquella que mapea un elemento de su dominio (de cualquier tamaño) a un elemento de tamaño específico contenido en la imagen.

Sería super lindo tener una función de hashing que produjera un resultado distinto siempre al evaluarla con datos de entrada distintos.

Hashing

Llamamos función de hash a aquella que mapea un elemento de su dominio (de cualquier tamaño) a un elemento de tamaño específico contenido en la imagen.

Sería super lindo tener una función de hashing que produjera un resultado distinto siempre al evaluarla con datos de entrada distintos.

Spoiler: eso no pasa.

Hashing

En esta clase vamos a enfocarnos en aplicar este tipo de funciones a strings.

Hashing

En esta clase vamos a enfocarnos en aplicar este tipo de funciones a strings.

La idea es definir una función de hash para un string S determinado mediante un polinomio de grado $|S| - 1$ que tiene la forma:

$$\begin{aligned} \text{hash}(s) &= s_0 + s_1 \cdot p + s_2 \cdot p^2 + ... + s_{n-1} \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s_i \cdot p^i \mod m \end{aligned}$$

Hashing

Usar un polinomio como función de hash tiene un detalle super importante:

Nos permite ver el hash correspondiente a un substring de S como:

$$\text{hash}(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \mod m$$

Hashing

Al hashear strings, se pueden hacer cosas super interesantes con muy poca complejidad computacional:

- Comparar substrings de una palabra

Hashing

Al hashear strings, se pueden hacer cosas super interesantes con muy poca complejidad computacional:

- Comparar substrings de una palabra
- Queries de palindromos

Hashing

Al hashear strings, se pueden hacer cosas super interesantes con muy poca complejidad computacional:

- Comparar substrings de una palabra
- Queries de palindromos
- "Borrar" un pedazo de un string

Hashing

Al hashear strings, se pueden hacer cosas super interesantes con muy poca complejidad computacional:

- Comparar substrings de una palabra
- Queries de palindromos
- "Borrar" un pedazo de un string
- Comparar substrings de una palabra (version hardcore)

Hashing

Ademas de ser útiles para operaciones sobre strings, las funciones de hash pueden aplicarse sobre otros dominios, por ejemplo caminos en un arbol.

Hashing

Ademas de ser útiles para operaciones sobre strings, las funciones de hash pueden aplicarse sobre otros dominios, por ejemplo caminos en un arbol.

Otra utilidad importante de hashing:

Resolver el problema Moonwalk Challenge :O

Trie

Para comenzar a hablar de Aho Corasick, primero es necesario tener bien en claro qué es y cómo se construye un trie.

Cuando hablamos de un trie nos referimos a un arbol que representa a un conjunto de strings. Las aristas se encuentran etiquetadas con un caracter, y los caminos que parten desde la raiz son prefijos de al menos una de las palabras del conjunto.

Trie

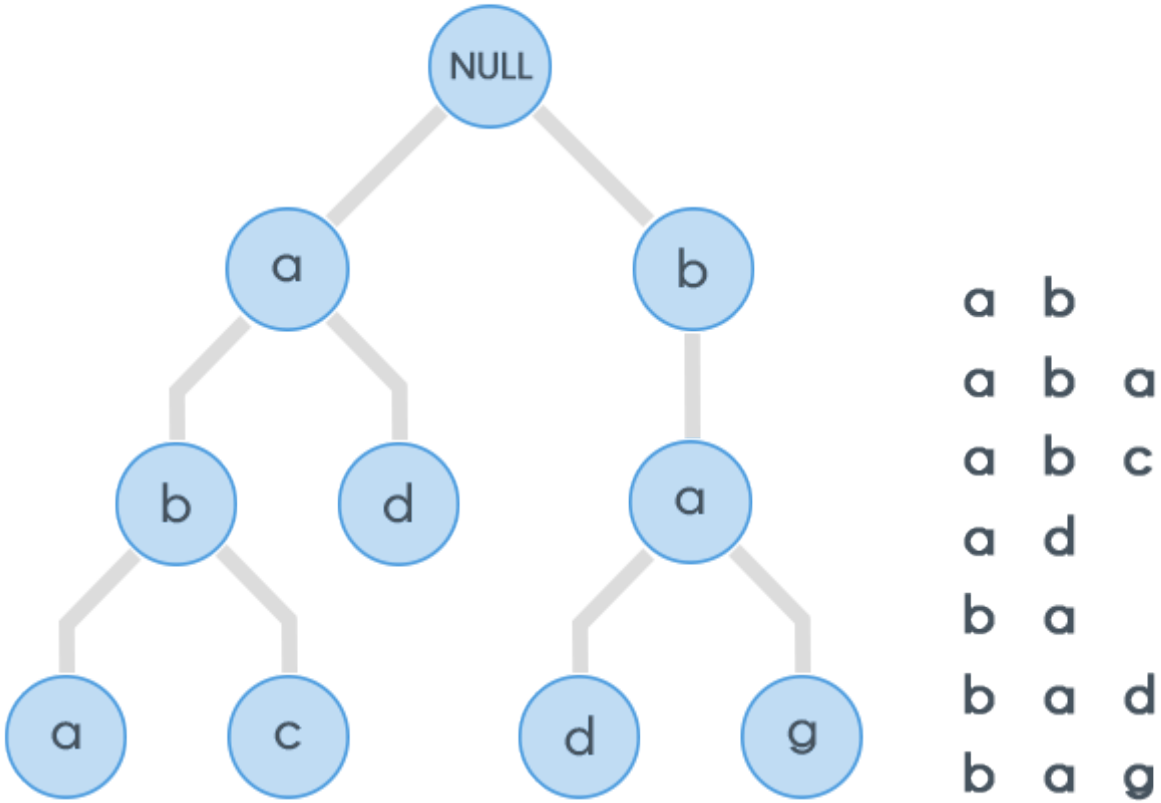


Fig. 1

Trie

```
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[c];
    }
    trie[v].leaf = true;
}
```

Aho Corasick

Un nodo del trie determina el estar parado en una posición de al menos un string del conjunto representado por el arbol. Estos nodos pueden verse como un autómata finito determinista (las transiciones son entre posiciones del conjunto de strings mediante letras).

Aho Corasick

Un nodo del trie determina el estar parado en una posición de al menos un string del conjunto representado por el arbol. Estos nodos pueden verse como un autómata finito determinista (las transiciones son entre posiciones del conjunto de strings mediante letras).

Problema:

Los autómatas tienen que tener transiciones definidas para todo caracter.

Solución:

En caso de estar en un nodo X y no poder transicionar hacia un hijo mediante un caracter C, intentar hacerlo desde el nodo correspondiente al sufijo mas grande del camino actual hasta X que exista en el trie. Luego, intentar realizar la transición desde ahí.

El único caso particular es la raiz del arbol, que vuelve a si misma en caso de no poder transicionar.

Aho Corasick

Vamos a llamar "suffix links" a estas nuevas transiciones. Si logramos computar de forma eficiente los suffix links, ganamos.

Aho Corasick

Vamos a llamar "suffix links" a estas nuevas transiciones. Si logramos computar de forma eficiente los suffix links, ganamos.

Cabe notar que un suffix link siempre nos va a llevar mas cerca de la raiz (ojo, no necesariamente a un ancestro del nodo actual).

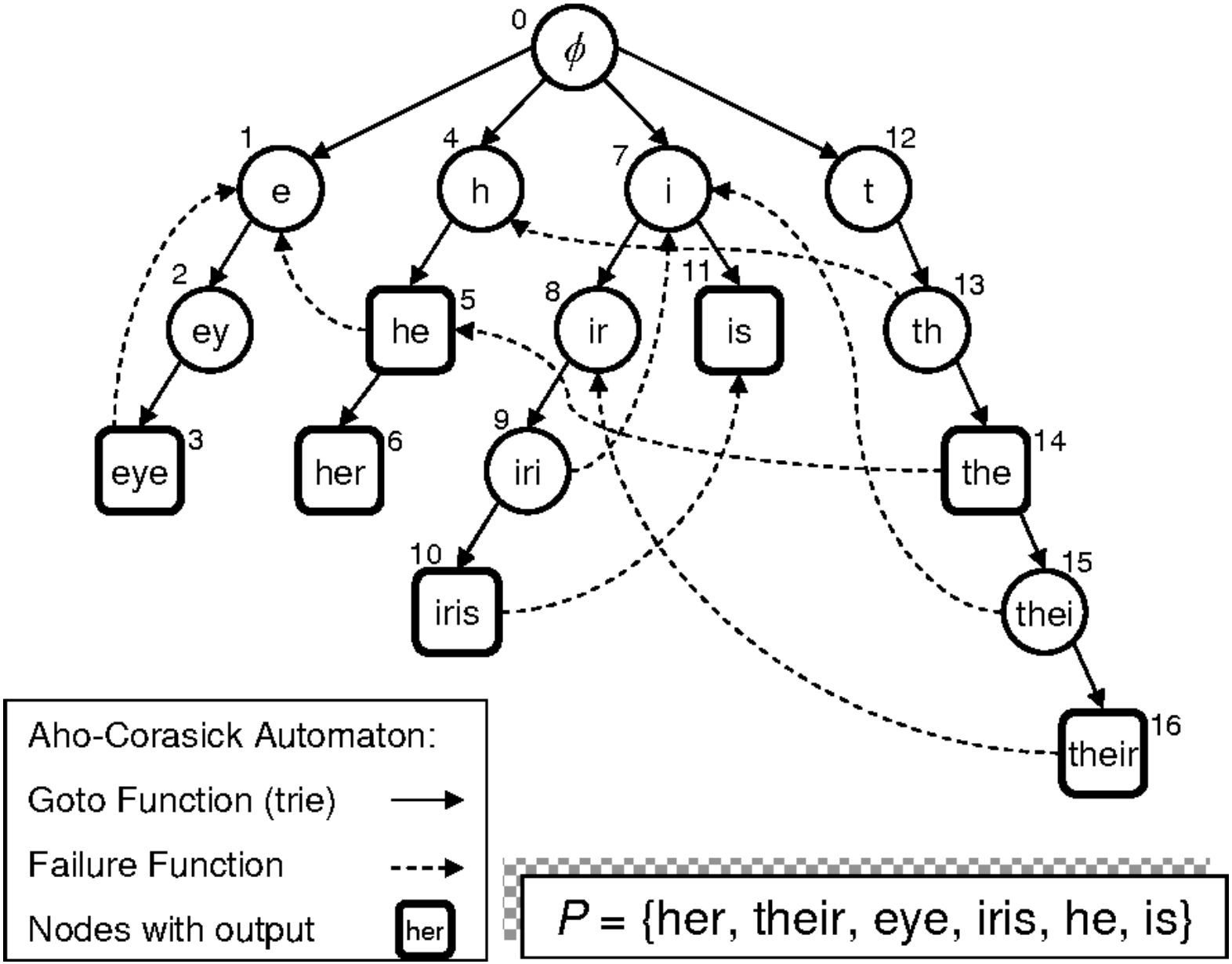
Una forma de obtener el suffix link para el nodo X, seria buscar el suffix link del padre de X (llamémoslo P_x), y luego transicionar por el mismo con la letra que utilizo P_x para ir hasta X.

Aho Corasick

Es facil ver que la idea propuesta para el cómputo de los suffix link tiene pinta de ser recursiva. Como procesamos cada nodo del trie a lo sumo una vez para obtener su suffix link, el cómputo total es lineal en la cantidad de nodos del autómata.

En la implementación, es conveniente guardar quién es el padre de cada nodo y la letra con la que éste transicionó hasta el vértice actual.

Aho Corasick



Aho Corasick

```
int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

Aho Corasick

A simple vista, esta estructura no parece ser muy util, pero en realidad es super poderosa. Algunas aplicaciones de Aho Corasick son:

- Encontrar todas las ocurrencias de un conjunto de strings en un texto

Aho Corasick

A simple vista, esta estructura no parece ser muy util, pero en realidad es super poderosa. Algunas aplicaciones de Aho Corasick son:

- Encontrar todas las ocurrencias de un conjunto de strings en un texto
- Minimo string que contenga un conjunto de strings como substrings

Aho Corasick

A simple vista, esta estructura no parece ser muy util, pero en realidad es super poderosa. Algunas aplicaciones de Aho Corasick son:

- Encontrar todas las ocurrencias de un conjunto de strings en un texto
- Minimo string que contenga un conjunto de strings como substrings
- Minimo string que contenga un conjunto de strings como substrings, pero version hardcore: la palabra generada no puede contener ningun string de otro conjunto que te dan.

Aho Corasick

A simple vista, esta estructura no parece ser muy util, pero en realidad es super poderosa. Algunas aplicaciones de Aho Corasick son:

- Encontrar todas las ocurrencias de un conjunto de strings en un texto
- Minimo string que contenga un conjunto de strings como substrings
- Minimo string que contenga un conjunto de strings como substrings, pero version hardcore: la palabra generada no puede contener ningun string de otro conjunto que te dan.
- Cantidad de palabras de largo $A \leq len \leq B$ que no contengan strings prohibidos

Aho Corasick

No siempre es trivial darse cuenta de que la solución a un problema involucra Aho Corasick, por lo que es un buen ejercicio codear varios de estos para familiarizarse con la estructura del código y los distintos trucos que se pueden hacer.

Dudas/preguntas

¿Preguntas?

Dudas/preguntas

¿Preguntas?

¡Gracias!