

Strings

Tries – Definição e Construção

Prof. Edson Alves - UnB/FGA

2019

1. Definição
2. Construção *naive* $O(N^2)$ da *trie*

Definição

Árvores de sufixos

- Árvores de sufixos são estruturas de dados que representam o conjunto $B(s)$ de todas as substrings de uma string s dada
- A relação de pertinência ($r \in B(s)$?) é o mais básico problema associado a esta estrutura
- Uma “boa” árvore de sufixos tem três características fundamentais:
 1. pode ser construída com tamanho linear
 2. pode ser construída em tempo linear
 3. pode responder questão de pertinência em complexidade linear em relação ao tamanho de s

Conceitos elementares

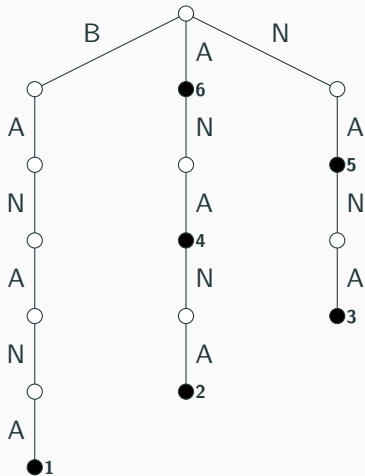
- Seja G um grafo acíclico direcionado, com raiz, cujas arestas recebem, como rótulos, caracteres ou palavras de um alfabeto A de tamanho constante
- Seja $label(e)$ o rótulo da aresta e
- O rótulo de um caminho p é a concatenação dos rótulos de todas as arestas do caminho
- Tal grafo representa um conjunto de strings, que são definidas pelos rótulos de todos os caminhos possíveis em G
- Seja

$$\mathcal{L}(G) = \{label(p) \mid p \text{ é caminho em } G \text{ com início na raiz}\}$$

- G representa todas as substrings de s se $\mathcal{L}(G) = B(s)$
- Um nó n cujo caminho da raiz até n tem como rótulo um sufixo de s é denominado nó essencial

- Uma *trie* de substrings de s , ou simplesmente *trie*, é o grafo G que representa todas as substrings de s , cujos rótulos consistem apenas de um único caractere
- O nome foi cunhado em 1961 por Edward Fredkin, a partir da sílaba central da palavra *retrieval*
- A pronúncia é idêntica a palavra *tree*, mas a grafia é diferente para diferenciar esta estrutura das árvores em geral
- A próxima figura ilustra a *trie* da palavra "BANANA"
- Os nós pretos são nós essenciais
- Os números ao lado dos nós essenciais são os índices do caractere inicial do sufixo

Visualização da *trie* da palavra 'BANANA'



Construção *naive* $O(N^2)$ **da** *trie*

Construção *naive* da *trie*

- Seja s uma string de tamanho N
- Cada nó da *trie* de s pode ter até $|A|$ filhos, onde A é o alfabeto
- Assim, cada nó pode ser implementado como um vetor de pares ou como um mapa, onde o par (c, n) , indicando que há uma aresta de rótulo c que aponta para o nó n
- A raiz da árvore será o nó identificado por $n = 0$
- Para cada caractere c do sufixo $s[i..N]$, e iniciando na raiz, verifica-se se existe a aresta (c, n)
- Em caso, afirmativo, segue-se esta aresta e se processa o caractere que sucede c
- Caso não exista, cria-se um novo nó m , adiciona-se ao nó atual a aresta (c, m) , e segue o processamento para m e para o próximo caractere
- Esta construção tem complexidade $O(N^2)$
- A memória necessária também é $O(N^2)$

Implementação *naive* da *trie*

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using Node = map<char, int>;
5 using Trie = vector<Node>;
6
7 Trie build_naive(const string& s)
8 {
9     int root = 0, next = 0;
10
11     Trie trie(1);      // Instancia o nó raiz vazio
12
13     for (int i = s.size() - 1; i >= 0; --i)
14     {
15         string suffix = s.substr(i);
16         int v = root;
17
18         for (auto c : suffix)
19         {
20             auto it = trie[v].find(c);
21
```

Implementação *naive* da *trie*

```
22         if (it != trie[v].end())
23         {
24             v = it->second;
25         } else
26         {
27             trie.push_back({ });
28             trie[v][c] = ++next;
29             v = next;
30         }
31     }
32 }
33
34 return trie;
35 }
```

Busca de substring em um *trie*

- A *trie* da string s pode ser utilizada para identificar se uma string p é ou não substring de s
- O algoritmo é semelhante à busca binária, e tem complexidade $O(m)$, onde $m = |p|$
- Por exemplo, se $s = \text{"BANANA"}$ e $p = \text{"NAN"}$, partindo da raiz, tem-se "N" na aresta à direita, "A" na única aresta e "N" na aresta seguinte: logo p é substring de s
- Como o nó de chegada é branco, p não é sufixo de s
- Para $p = \text{"NAS"}$, o último caractere ("S") não seria encontrado
- O mesmo vale para $p = \text{"MAS"}$, porém a falha acontece logo no primeiro caractere
- Para $p = \text{"NANAN"}$ a busca se encerraria ao atingir um nó nulo

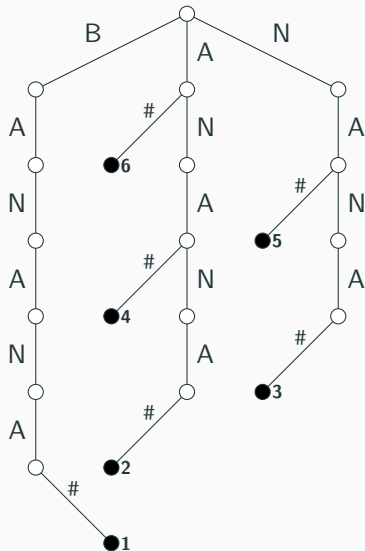
Implementação da busca em $O(m)$ em uma *trie*

```
66 bool search(const Trie& trie, const string& s)
67 {
68     int v = 0;
69
70     for (auto c : s)
71     {
72         auto it = trie[v].find(c);
73
74         if (it == trie[v].end())
75             return false;
76
77         v = it->second;
78     }
79
80     return true;
81 }
```

Trie com marcadores

- A construção proposta para a *trie* permite ao algoritmo de busca descrito apenas determinar se a substring p ocorre ou não em s
- Se for preciso determinar a posição (ou posições) desta ocorrência, é preciso modificar a construção da *trie*, de modo que seja possível discriminar os nós essenciais dos demais
- Uma maneira de fazê-lo é adicionar um caractere terminador (em geral, o caractere '#'), que não pertença a string original
- A este caractere estará associado o índice i da string tal que o sufixo terminado no marcador é igual a $S[i..N]$
- Importante notar que o segundo elemento do par terá dois significados distintos: ou será um ponteiro para o próximo nó, ou o índice do sufixo caso o rótulo da aresta seja o terminador
- É preciso atentar a esta diferença na implementação das rotinas de construção e busca

Visualização da *trie* da palavra 'BANANA' com terminador



Construção da *trie* com marcador

```
83 Trie build_naive_with_marker(const string& s)
84 {
85     int root = 0, next = 0;
86
87     Trie trie(1);          // Instancia o nó raiz vazio
88
89     for (int i = s.size() - 1; i >= 0; --i)
90     {
91         string suffix = s.substr(i) + '#';
92         int v = root;
93
94         for (auto c : suffix)
95         {
96             if (c == '#')
97             {
98                 trie[v][c] = i;
99                 break;
100             }
101
102             auto it = trie[v].find(c);
103
```


Construção da *trie* com marcador

```
104         if (it != trie[v].end())
105         {
106             v = it->second;
107         } else
108         {
109             trie.push_back({ });
110             trie[v][c] = ++next;
111             v = next;
112         }
113     }
114 }
115
116 return trie;
117 }
```

1. CP Algorithms. [Suffix Tree](#). [Ukkonen's Algorithm](#), acesso em 02/10/2019.
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **ROY, TUSHAR**. [Trie](#), acesso em 02/10/2019.
4. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
5. Wikipédia. [Trie](#), acesso em 02/10/2019.