

# Trabalho 1 - Autômatos de Sufixos

Bruno Aurélio Rôzza de Moura Campos<sup>1</sup> (202103807)

Lucas Henrique Fonseca<sup>1</sup> (202103876)

Pedro Alexandre Barradas da Côrte<sup>1</sup> (202103680)

<sup>1</sup>Departamento de Informática e Estatísticas

Universidade Federal de Santa Catarina (UFSC) - Florianópolis, SC - Brasil

## 1. Autômatos de Sufixos

### 1.1. O que são?

De acordo com [CP-Algorithms ], os autômatos de sufixos são estruturas de dados que permitem resolver muitos problemas relacionados a *strings*, e podem ser definidos como Grafos Acíclicos Dirigidos. De maneira geral, um autômato de sufixo para uma *string*  $S$  qualquer é um AFD (Autômato Finito Determinístico) que aceita todos os sufixos da *string*  $S$ . Essa é a característica mais importante do autômato de sufixo, isso é, ele contém informações de todas as *substrings* que representam sufixos da *string*  $S$ . Caracterizado por possuir estados, transições, nó inicial e conjunto de nós finais.

### 1.2. Para que servem?

De acordo com [Wikipedia ] (e assumindo que  $T$  é dado como *input* no autômato de sufixo de  $S$  (ou seja, tal autômato já foi construído)), um autômato de sufixo de uma *string*  $S$  pode ser utilizado para contar o número de *substrings* distintas de  $S$  em  $O(|S|)$ , para encontrar a *substring* mais longa de  $S$  que aparece ao menos duas vezes em  $O(|S|)$ , para encontrar a *substring* mais longa e comum entre  $S$  e  $T$  em  $O(|T|)$ , para contar o número de ocorrências de  $T$  em  $S$  em  $O(|T|)$ , ou para encontrar todas as ocorrências de  $T$  em  $S$  em  $O(|T| + k)$  (onde  $k$  é o número de ocorrências). Esses autômatos também podem ser utilizados, de maneira geral, em compressão de dados, em recuperação de música ou em correspondência de sequências de genoma.

### 1.3. Como construí-los?

De acordo com [Wikipedia ], a construção de um autômato de sufixo de uma *string*  $S$  começa por um AFD simples que possui apenas um estado (que corresponde à palavra vazia). Na sequência, o AFD será incrementado a cada carácter da *string*  $S$ , sendo re-construído de forma incremental conforme necessário.

A cada novo carácter que é anexado à *string*, algumas classes de equivalência são alteradas. Considerando que  $[\alpha]_{R_S}$  é o contexto correto de  $\alpha$  com respeito à linguagem dos sufixos de  $S$ , a transição de  $[\alpha]_{R_S}$  para  $[\alpha]_{R_{Sx}}$  após anexar um carácter  $x$  a  $S$  pode ser definido por um lema que diz que existe uma correspondência entre  $[\alpha]_{R_S}$  e  $[\alpha]_{R_{Sx}}$  onde:

- $[\alpha]_{R_{Sx}} = [\alpha]_{R_S}x \cup \{\epsilon\}$  caso  $\alpha$  seja um sufixo de  $Sx$
- $[\alpha]_{R_{Sx}} = [\alpha]_{R_S}x$  caso  $\alpha$  não seja um sufixo de  $Sx$

[Wikipedia ] apresenta um algoritmo teórico de construção do autômato de sufixo. Para adicionar um carácter  $x$  e reconstruir o autômato de sufixo de  $S$  para que este passe a ser o autômato de sufixo de  $Sx$ , deve-se:

1. Salvar uma referência *last* ao último estado da palavra *S*;
2. Após anexar o caráter *x*, guarda-se *last* numa variável *p* e *last* é reatribuído ao novo estado final de *Sx*;
3. Os estados correspondentes aos sufixos de *S* são atualizados com as transições para *last*. Para fazer isso, deve-se passar por *p*, *link(p)*, *link<sup>2</sup>(p)*, etc, até que haja um estado que já tenha uma transição de *x*;
4. Assim que o loop mencionado acima terminar, existem 3 casos:
  - (a) Se nenhum dos estados no caminho de sufixo tiver uma transição de *x*, então *x* nunca ocorreu em *S* antes e o link de sufixo de *last* deve levar a  $q-\{0\}$ ;
  - (b) Se a transição por *x* for encontrada e levar do estado *p* para o estado *q*, tal que  $len(p) + 1 = len(q)$ , então *q* não precisa ser dividido e é um link de sufixo de *last*;
  - (c) Se a transição for encontrada, mas  $len(q) > len(p) + 1$ , então as palavras de *q* tendo comprimento no máximo  $len(p) + 1$  devem ser segregadas em um novo estado “clone” *cl*;
5. Se a etapa anterior foi concluída com a criação do *cl*, as transições dele e de seu link de sufixo devem copiar as de *q*, ao mesmo tempo que *cl* é designado como link de sufixo comum de *q* e *last*;
6. As transições que levaram a *q* antes, mas corresponderam a palavras de comprimento no máximo  $len(p) + 1$ , são redirecionadas para *cl*. Para fazer isso, continua-se percorrendo o caminho do sufixo de *p* até que o estado seja encontrado de forma que a transição de *x* a partir dele não leve a *q*.

#### 1.4. Que tipos de problemas eles podem resolver?

De acordo com [CP-Algorithms ], existem algumas tarefas que podem ser resolvidas utilizando autômatos de sufixo. Alguns dos exemplos mencionados são os seguintes:

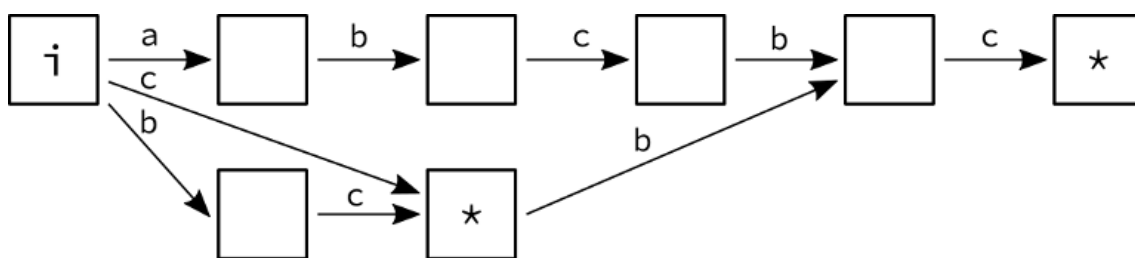
1. Checar ocorrência
  - (a) Dado um texto *T* e vários padrões *P*, pode-se checar se as *strings* de *P* são *substrings* de *T*.
2. Número de *substrings* diferentes
  - (a) Dada uma *string* *S*, pode-se computar o total de *substrings* diferentes contidas em *S*.
3. Tamanho total de todas as *substrings* diferentes
  - (a) Dada uma *string* *S*, pode-se computar o tamanho total de todas as *substrings* diferentes contidas em *S*.
4. *Substring* *n* lexicograficamente
  - (a) Dada uma *string* *S*, para cada número *K* fornecido, pode-se encontrar a *string* de número *K* na lista de *substrings* organizadas em ordem lexicográfica.
5. Número de ocorrências
  - (a) Dado um texto *T*, para cada padrão *P* fornecido, pode-se encontrar quantas vezes a *string* *P* está contida na *string* *S*.
6. Posição de primeira ocorrência
  - (a) Dado um texto *T*, para cada padrão *P* fornecido, pode-se encontrar a primeira posição onde a *string* *P* aparece na *string* *T*.

7. Todas as posições de ocorrência
  - (a) Dado um texto  $T$ , para cada padrão  $P$  fornecido, pode-se encontrar todas as posições diferentes onde a *string*  $P$  aparece na *string*  $T$ .
8. *String* mais curta não aparente
  - (a) Dada uma *string*  $S$ , pode-se encontrar uma *string* do menor tamanho possível que não aparece em  $S$ .
9. A *substring* comum mais longa de duas *strings*
  - (a) Dadas duas *strings*  $S$  e  $T$ , pode-se encontrar uma *string*  $X$  que é a maior *substring* que aparece simultaneamente em  $S$  e  $T$ .
10. Maior *substring* comum de várias *strings*
  - (a) Dado uma quantidade  $k$  de *strings*  $S_i$ , pode-se encontrar uma *string*  $X$  que aparece como uma *substring* de cada *string*  $S$ .

### 1.5. Outros detalhes interessantes

Para a construção desses autômatos, existem algumas premissas básicas que validam essa estrutura, dentre elas, o número de estados ou nós de um autômato de sufixos é igual ao número de classes *endpos* [Saisumit]. A classe *endpos* refere-se ao conjuntos de todas as posições onde  $T$  termina em  $S$ , para um  $T$  não vazio. Considerando um autômato para a *string* *abcbc* (Figura 1).

Figura 1. Autômato para a *string* *abcbc*



Fonte: [Codeforces]

Uma possível abordagem, é mapear a cada estado toda a localização de uma *string*  $S$ , baseando-se pelas letras anteriores, e a cada nova letra de entrada, checar a localização correta e das possíveis subsequentes. Para o exemplo da Figura 1, considerando uma *string* fornecida *bcbc*, as possíveis tentativas de localização seriam: *aBcBc*, *abCbC*, *abcBc*, *abcbC*. Desse modo, todos os nós de um autômato de sufixos devem corresponder as possíveis localizações que podem ocorrer durante a checagem (classes *endpos*).

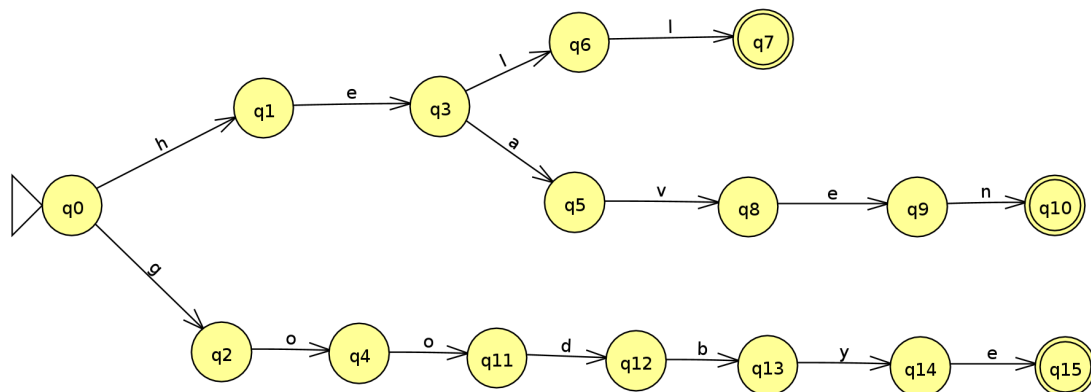
## 2. Análise informal do problema “Digitando no Telefone Celular”

O problema em questão tem como objetivo otimizar tempo do usuário através da inserção de letras quando ele estiver digitando alguma palavra.

### 3. Estratégia adotada pelo grupo

Primeiramente mapeamos o problema utilizando um autômato para montar um árvore de sufixo com o primeiro exemplo dado (Figura 2).

**Figura 2. Autômato contendo três palavras**



**Fonte: Desenvolvido pelos autores**

Para implementar os nós e inserir novas palavras no autômato foi utilizado a estrutura de dados de árvore de prefixo, também conhecida como Trie. Conforme descrito por [Hovhannisyan ] esta árvore é constituída de nós de prefixo ramificados que, quando seguidos na sequência correta, nos levam a uma palavra completa. Abaixo há trechos do código com a implementação dos nós e da função de inserção:

```
class TrieNode:
    """Um nó na estrutura trie"""

    def __init__(self, char: str):
        # a letra armazenada neste nó
        self.char = char
        # pode ser o fim de uma palavra
        self.is_end = False
        # um dicionário de nós filhos,
        # onde chaves são caracteres e os valores são nós
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode("")

    def insert(self, word: str) -> None:
        node = self.root

        for char in word:
```

```

# Verifica se o nó filho contem o caractere da word
if char in node.children:
    node = node.children[char]
else:
    # se não encontrado, cria o novo nó na trie
    new_node = TrieNode(char)
    node.children[char] = new_node
    node = new_node

node.is_end = True

```

Para o problema de calcular o número médio de pressionamentos de teclas no dicionário, dividimos em três partes. Na primeira, implementamos uma função que retorna o total de teclas digitadas a partir de uma palavra. Abaixo há o código da função:

```

def get_total_keystrokes_by_word(self, word: str) -> int:
    """Obtém o total de teclas digitadas"""
    node = self.root
    total_keystrokes = 0

    for char in word:
        total_child = len(node.children.keys())

        # Se o nó da trie é final
        # ou se total_child é maior do q 1,
        # então conta mais uma tecla
        if total_child > 1 or node.is_end is True:
            # print(f'typing {node.children.keys()}')
            total_keystrokes += 1

        # próximo node da trie
        node = node.children[char]

    return total_keystrokes

```

Na segunda parte, implementamos na função principal uma iteração de todas as palavras de um dicionário, onde dentro desta é feita a invocação do método desenvolvido na primeira etapa. Por fim, realizamos a divisão do total de teclas digitadas sobre o total de palavras. Abaixo há o trecho de código citado:

```

# Obtém uma quantidade de teclas digitadas
# a partir de um dicionário de palavras
ans = sum(
    tree.get_total_keystrokes_by_word(word=word)
    for word in list_words
)

# Média de pressionamentos de tecla por dicionário
print(round(ans / len(list_words), 2))


```

### 3.1. Dificuldades encontradas pelo grupo

O melhor resultado alcançado dentro do Juri online foi de 65% de acerto. Contudo, ao executar o código desenvolvido com as amostras de entradas, todas as saídas retornam o valor correto.

### 3.2. Screenshot da resposta do juri online

Figura 3. Screenshot da resposta do juri online

 **SOURCE CODE** EDIT & SUBMIT

VISUALIZE THE SOURCE CODE OF YOUR SUBMISSION, PLUS SOME EXTRA DETAILS.

**SUBMISSION # 24292433**

PROBLEM:	1284 - Cellphone Typing
ANSWER:	<b>Wrong answer (35%)</b>
LANGUAGE:	Python 3.8 (Python 3.8.2) [+1s]
RUNTIME:	0.028s
FILE SIZE:	3.11 KB
MEMORY:	-
SUBMISSION:	9/2/21, 12:31:41 PM

## Referências

Codeforces. <https://codeforces.com/blog/entry/20861>.

CP-Algorithms. <https://cp-algorithms.com/string/suffix-automaton.html>.

Hovhannisyan, A. <https://www.aleksandrhovhannisyan.com/blog/trie-data-structure-implementation-in-python/>.

Saisumit. <https://saisumit.wordpress.com/2016/01/26/suffix-automaton/>.

Wikipedia. [https://en.wikipedia.org/wiki/Suffix\\_automaton](https://en.wikipedia.org/wiki/Suffix_automaton).