

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Gabriel Luís Mello Dalalio

Estudo de algoritmos e estruturas de dados para a resolução de
problemas relacionados a “string matching”

Trabalho de Graduação
2013

Computação

Gabriel Luís Mello Dalalio

**Estudo de algoritmos e estruturas de dados para a resolução de
problemas relacionados a “string matching”**

Orientador
Prof. Armando Ramos Gouveia

Divisão de Ciência da computação

SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

2013

Dados Internacionais de Catalogação-na-Publicação (CIP)

Divisão de Informação e Documentação

Dalalio, Gabriel Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching” / Gabriel Dalalio. São José dos Campos, 2013. 53f. Trabalho de Graduação – Divisão de Ciência da Computação – Instituto Tecnológico de Aeronáutica, 2013. Orientador: Prof. Armando Ramos Gouveia. . 1. Algoritmos. 2. Estruturas de dados. 3. String Matching. 4. Vetor de Sufixos. 5. Autômato de sufixos I. Instituto Tecnológico de Aeronáutica. II. Título
--

REFERÊNCIA BIBLIOGRÁFICA

DALALIO, Gabriel. **Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching”**. 2013. 53f. Trabalho de Conclusão de Curso. (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Gabriel Luís Mello Dalalio

TÍTULO DO TRABALHO: Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching”

TIPO DO TRABALHO/ANO: Graduação / 2013

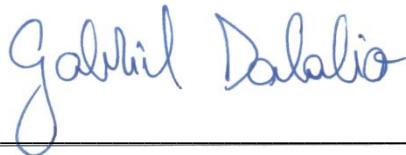
É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta monografia de graduação pode ser reproduzida sem a autorização do autor.



Gabriel Luís Mello Dalalio
Av. Doutor Eduardo Cury, 200, ap. 37
Jardim Colinas
São José dos Campos SP 12242-001
Brasil

Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching”

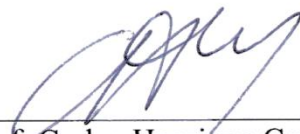
Essa publicação foi aceita como Relatório Final de Trabalho de Graduação



Gabriel Luís Mello Dalalio
Autor



Prof. Armando Ramos Gouveia
Orientador



Prof. Carlos Henrique Costa Ribeiro
Coordenador do Curso de Engenharia de Computação

São José dos Campos, 26 de Novembro de 2013

Chego aqui nesse momento
a partir de muita dedicação e sorte, onde pequenos fatos fizeram toda a diferença.

AGRADECIMENTOS

Agradeço aos meus colegas de equipe da maratona de programação, Diogo Holanda, Felipe Freitas e especialmente o Fernando Fonseca que competiu ao meu lado por dois anos. Foram eles que me ajudaram a participar duas vezes da competição mundial e também a conquistar a melhor colocação da América Latina em 2013. O tema deste trabalho surgiu a partir do estudo para essa competição.

Também preciso agradecer muito ao Prof. Armando Gouveia que me ajudou diversas vezes ao longo dessa jornada de cinco anos de graduação e que também é responsável direto pelo incentivo da participação dos alunos do ITA na competição da maratona de programação. São professores como ele que engrandecem o nome da instituição e que fazem diferença na vida dos alunos de computação que passam por ela.

RESUMO

A dissertação propõe-se a analisar algoritmos e estruturas de dados para resolver problemas relacionados a *string matching*. Inicialmente, este texto apresenta alguns algoritmos mais conhecidos para resolver o problema de *string matching*, que se caracteriza em achar uma sequência contínua de caracteres dentro de outra sequência maior de caracteres. Contudo, esses algoritmos não conseguem resolver eficientemente algumas variações do problema, como por exemplo, quando se deseja achar várias palavras dentro de uma mesma sequência de caracteres. Com isso, são apresentadas duas estruturas de dados capazes de lidar melhor com essas variações do problema, o vetor de sufixos e o autômato de sufixos. Após a apresentação das estruturas, são analisadas algumas variações do problema em questão, incluindo variações simples e também variações mais complexas baseadas em problemas de competições passadas de programação.

ABSTRACT

This work proposes to examine algorithms and data structures to solve problems related to string matching. Initially, this text shows some classical algorithms to solve the string matching problem. This problem consists in finding some string inside of another bigger string. However, these algorithms cannot solve efficiently some variations of the problem, for example, when the problem asks to find multiple strings inside of the same other string. Then the text describes two data structures that handle better these variations of the problem, the suffix array and suffix automaton. After this, some variations of the string matching problem are analyzed, including simple variations and some complex variations based on past programming competitions problems.

LISTA DE FIGURAS

1 Exemplo de vetor de sufixos	24
2 Exemplo dos passos de construção do vetor de sufixos	25
3 Exemplos de autômatos de sufixos.....	33
4 Exemplo de autômato de sufixos com representação de <i>links</i> e <i>length</i>	36

LISTA DE TABELAS

1 Tempos médios de execução utilizando texto e padrões aleatórios	40
2 Tempos médios de execução utilizando padrões com muitas ocorrências no texto	42
3 Tempos médios de pré-processamento de textos aleatórios	43
4 Tempos médios de pré-processamento de textos periódicos	43
5 Tempos médios de busca por padrão em texto escolhidos aleatoriamente	44
6 Tempos médios de busca por padrão em texto escolhidos com período	44

LISTA DE ABREVIATURAS E SIGLAS

KMP Algoritmo de Knuth-Morris-Pratt

LISTA DE LISTAGENS

1 Código C++ com algoritmo simples para <i>string matching</i>	19
2 Código C++ com o algoritmo KMP	21
3 Código C++ com construção do vetor de sufixos.....	27
4 Código C++ com opção alternativa de construção do vetor de sufixos	29
5 Código C++ com função de comparação entre padrão e um sufixo do texto.....	30
6 Código C++ com busca por padrão utilizando o vetor de sufixos	31
7 Código C++ com implementação de estados de autômatos utilizando <i>map</i>	35
8 Código C++ com variáveis globais do autômato de sufixos	37
9 Código C++ com função de inicialização do autômato de sufixos	37
10 Código C++ com função de extensão <i>online</i> do autômato de sufixos.....	38
11 Código C++ com busca por padrão utilizando o autômato de sufixos.....	39
12 Código C++ que sorteia padrões e textos periódicos	41
13 Código C++ com cálculo de contagem no autômato de sufixos	46
14 Código C++ com função de contagem de substrings do autômato de sufixos.....	48
15 Código C++ com cálculo de tamanho do maior padrão repetido	49
16 Código C++ com cálculo do maior tamanho de padrão presente em dois textos.....	50

SUMÁRIO

1 INTRODUÇÃO.....	15
1.1 Motivação	15
1.2 Objetivo	16
1.3 Estrutura do trabalho	16
2 O PROBLEMA.....	17
2.1 <i>String Matching</i>	17
3 ABORDAGENS CLÁSSICAS DE RESOLUÇÃO	19
3.1 Algoritmo simples	19
3.2 Algoritmo de Knuth-Morris-Pratt.....	20
3.3 Algoritmo de Aho-Corasick	22
3.4 Outros algoritmos	23
4 VETOR DE SUFIXOS	24
4.1 Definição	24
4.2 Construção.....	25
4.3 Busca por padrão utilizando a estrutura	30
5 AUTÔMATO DE SUFIXOS	32
5.1 Definição	32
5.2 Construção.....	34
5.2.1 Estrutura e atributos dos estados	34
5.2.2 Construção online.....	36
5.3 Busca por padrão utilizando a estrutura	39
6 ANÁLISE DO PROBLEMA DE <i>STRING MATCHING</i>	40
6.1 Busca de um padrão em um texto.....	40
6.2 Busca de múltiplos padrões em um texto	42
6.3 Número de ocorrências de um padrão em um texto	45

6.4 Posições das ocorrências de um padrão em um texto.....	46
7 PROBLEMAS RELACIONADOS	48
7.1 Número de subsequências diferentes em um texto.....	48
7.2 Maior padrão que ocorre múltiplas vezes em um texto.....	49
7.3 Maior padrão que ocorre em dois textos	49
8 COMENTÁRIOS FINAIS	51
8.1 Conclusão	51
8.2 Trabalhos Futuros	51
9 REFERÊNCIAS	52

1 INTRODUÇÃO

1.1 Motivação

Uma *string* pode ser definida como uma sequência de caracteres. Um dos problemas mais clássicos que surgiu a partir do estudo de *strings* é o problema denominado *string matching*, que consiste em buscar uma *string* menor dentro de uma *string maior*. Esse problema ocorre, por exemplo, quando é necessário achar uma palavra ou fragmento de palavra dentro de um texto ou até mesmo pode ser estendido para encontrar palavras dentro de um passatempo de caça-palavras.

Para resolver esse problema, surgiram vários algoritmos que são capazes de resolver o problema eficientemente, por exemplo, resolver o problema em complexidade linear ou ser eficiente pelo menos em casos médios. Dentre eles, podemos citar o algoritmo de Knuth-Morris-Pratt (KMP), algoritmo de Rabin-Karp, algoritmo de Boyer-Moore, entre outros.

Apesar da eficiência desses algoritmos para a resolução do problema de *string matching*, esses algoritmos parecem não ser tão eficientes quando modificamos e estendemos o problema em questão. Uma modificação possível para o problema, que será analisada nesse trabalho, é o problema de *string matching* com múltiplos padrões. O que acontece com a eficiência desses algoritmos quando é necessário encontrar várias *strings* dentro de um mesmo texto?

Para resolver o problema com múltiplos padrões, pode-se simplesmente repetir os algoritmos citados para todos os padrões. Porém, recentemente têm sido estudadas algumas estruturas de dados que podem nos ajudar nesse caso. Esse trabalho apresentará duas delas: o vetor de sufixos e o autômato de sufixos. Essas estruturas permitirão um pré-processamento dos textos para que possam ser feitas várias consultas buscando múltiplos padrões de forma mais eficiente que os algoritmos que resolvem o problema em sua forma mais simples.

Além da modificação do problema para múltiplos padrões, as estruturas citadas são capazes ainda de resolver extensões mais complexas do problema de *string matching*. Esse trabalho apresentará algumas delas, incluindo modificações que já foram alvos de competições de programação.

1.2 Objetivo

Este trabalho tem, por fim, analisar a resolução de problemas relacionados a *string matching* utilizando duas estruturas de dados, vetor de sufixos e autômato de sufixos, comparando os desempenhos com alguns algoritmos clássicos de busca de padrão em textos.

1.3 Estrutura do trabalho

A fim de proporcionar uma divisão natural do trabalho, esse trabalho foi dividido nos seguintes capítulos:

- O capítulo 2 irá definir o problema de *string matching* e tentar deixar claro qual é exatamente o problema que irá ser analisado.
- O capítulo 3 irá comentar sobre as abordagens clássicas que são usadas na resolução da versão mais simples do problema em questão, passando por exemplo por algoritmos como o de Knuth-Morris-Pratt (KMP).
- O capítulo 4 apresentará a primeira estrutura de dados alvo do estudo desse texto, o vetor de sufixos (*suffix array*), mostrando o processo de construção, algoritmos relacionados e complexidade de tempo envolvida.
- O capítulo 5 apresentará a segunda estrutura de dados alvo do estudo desse texto, o autômato de sufixos (*suffix automaton*), mostrando também o processo de construção, algoritmos relacionados e complexidade de tempo envolvida.
- O capítulo 6 mostrará a comparação entre os algoritmos clássicos e o uso das estruturas de dados para a resolução dos problemas similares ao *string matching*.
- O capítulo 7 ainda apresenta mais variações complexas do problema em questão e mostra como as estruturas de dados podem ser usadas nos diferentes casos.
- O capítulo 8 finaliza o trabalho apresentando as conclusões que puderam ser desenvolvidas ao longo do estudo desse trabalho.

2 O PROBLEMA

Antes de começar a análise proposta por esse trabalho, é necessário especificar com detalhes qual é o problema a ser resolvido. Nessa seção será definido o que é exatamente o problema denominado como *string matching*.

2.1 String Matching

Uma *string* pode ser definida como uma sequência finita de caracteres. O tamanho de uma *string* será definido como o número de caracteres na sequência. Se uma *string* S tem tamanho igual a N , será utilizada a notação $S[i]$ para representar o i -ésimo caractere da *string*. A indexação será feita a partir de zero, ou seja, o primeiro caractere será indicado por $S[0]$, o segundo por $S[1]$, assim por diante até o último caractere indicado por $S[N-1]$.

O conjunto de valores que os caracteres de uma *string* podem assumir será denominado de alfabeto. Durante a análise feita nesse trabalho, será discutida a influência do tamanho do alfabeto nos algoritmos propostos. Exemplificando, pode-se utilizar um alfabeto binário, com apenas dois valores distintos, um alfabeto consistindo no conjunto de todas as letras minúsculas ou até mesmo tomar todos os caracteres ASCII como alfabeto.

O problema denominado como *string matching* consiste em verificar se uma *string* pode ser encontrada dentro de uma *string* maior. Mais precisamente, dada uma *string* T de tamanho N e uma segunda *string* P de tamanho M , com $M \leq N$, deve-se determinar se existe ou não um inteiro não negativo pos tal que:

$$P[i] = T[pos + i] \text{ e } pos + i < N, \text{ para todo } i \in \{0, 1, \dots, M-1\}$$

A *string* P será denominada de padrão e *string* T será denominada de texto. Cada inteiro pos que satisfaz a condição acima é caracterizado como uma posição onde há uma ocorrência do padrão dentro do texto.

Um dos cuidados que se deve tomar ao encontrar as ocorrências do padrão no texto é que pode ocorrer intersecção entre as ocorrências. Por exemplo, se tomarmos como texto $T = \text{"aaabcbcabcc"}$ e $P = \text{"abcb"}$, há ocorrências nas posições 2 e 5 e elas possuem uma intersecção de duas letras, nas posições 5 e 6.

Da maneira que o problema foi caracterizado, podem-se utilizar os algoritmos de resolução de *string matching* para encontrar palavras em textos comuns, mas deve-se tomar

cuidado nesse caso, pois as ocorrências podem indicar palavras ou fragmentos de palavras. Exemplificando, pode-se encontrar a palavra "*dos*" dentro da palavra "*quadrados*". Também é possível encontrar expressões inteiras ao invés de apenas palavras avulsas.

Há muitas aplicações do problema analisado, que vão desde ferramentas de busca de texto até pesquisas relacionadas a processamento de cadeias de DNA. Essas aplicações podem ser vistas mais detalhadamente na referência (1).

3 ABORDAGENS CLÁSSICAS DE RESOLUÇÃO

Esta seção apresentará alguns algoritmos conhecidos para resolver o problema de *string matching*. A análise desses algoritmos não será feita de forma extensa, pois o que se busca aqui é só o conhecimento necessário para comparação posterior com os algoritmos baseados nas estruturas de sufixos.

3.1 Algoritmo simples

Um dos algoritmos mais simples para encontrar ocorrências de um padrão em um texto pode ser feito utilizando força bruta em todas as posições. Testam-se todas as posições do texto à procura do padrão. Esse algoritmo pode ser implementado como mostrado a seguir em um código feito utilizando a linguagem C++.

Listagem 1: Código C++ com algoritmo simples para *string matching*

```
// Esta função retorna o número de ocorrências do padrão p
// no texto t
int func_simples() {
    int i, j, ret=0;
    for( i=0 ; i<n ; i++ ) {
        for( j=0 ; j<m && i+j<n ; j++ ) {
            if( p[j]!=t[i+j] ) {
                break;
            }
        }
        if( j==m ) {
            ret++; // encontrou uma ocorrência na posição i
        }
    }
    return ret;
}
```

Assim como todas as implementações que serão apresentadas nesse trabalho, as variáveis n e m são utilizadas como o tamanho do texto e o tamanho do padrão respectivamente. Os vetores $p[]$ e $t[]$ sempre indicam também o padrão e o texto nessa ordem. As implementações utilizam letras minúsculas, porém essas quatro variáveis são indicadas com letras maiúsculas durante o texto.

Analisando o algoritmo simples proposto, obtém-se uma complexidade de tempo igual a $O(N \cdot M)$, pois o número de posições a serem testadas pode chegar perto de N e cada teste

pode levar M passos. Além da memória utilizada para guardar o texto e o padrão, a complexidade de memória adicional utilizada pelo algoritmo é $O(1)$, pois só há a necessidade de criar algumas variáveis auxiliares.

Apesar do pior caso de execução desse algoritmo ter complexidade de tempo $O(N \cdot M)$, o algoritmo é rápido caso não haja muitas ocorrências de prefixos do padrão dentro do texto. Isso acontece porque o teste feito em cada posição do texto pode levar poucos passos caso o programa detecte pelo menos um caractere errado rapidamente.

Com os caracteres do padrão e do texto sendo escolhidos de maneira totalmente aleatória dentro do alfabeto, o algoritmo tem uma complexidade de caso médio $\Theta(N)$, facilmente alcançada se o alfabeto for grande. Apesar disso, na prática a maioria das aplicações do problema não possuem padrão e texto escolhidos dessa forma.

3.2 Algoritmo de Knuth-Morris-Pratt

O algoritmo de Knuth-Morris-Pratt (KMP) foi publicado por Donald Knuth, Vaughan Pratt e James H. Morris em 1977. Esse algoritmo consegue resolver o problema de *string matching* na melhor complexidade possível, $O(N + M)$. Não há como conseguir complexidade melhor, pois apenas a leitura total do padrão e do texto já alcança a mesma complexidade.

Da mesma maneira que o algoritmo simples testa todas as posições possíveis de ocorrência no texto, o KMP também passa por todas elas, porém o teste de ocorrência não é feito de maneira independente. Dependendo de quais letras são encontradas no teste de certa posição de ocorrência, já é possível prever o que acontecerá no teste da próxima posição que será testada.

Antes de o algoritmo KMP começar a varrer as posições do texto à procura de ocorrências, há a execução de um pré-processamento do padrão. Esse processamento permite que o algoritmo consiga fazer cada teste de checagem de ocorrência em tempo constante. Dessa maneira, como o pré-processamento tem complexidade de tempo linear, o algoritmo alcança a complexidade de $O(N + M)$, tanto em tempo como em memória.

A listagem de código a seguir mostra uma possível implementação do algoritmo na linguagem C++. Ela será usada mais adiante no trabalho para comparar o algoritmo KMP com os outros algoritmos propostos.

Listagem 2: Código C++ com o algoritmo KMP

```
vector<int> pos; // função de transição

// calcula função de transição
void trans_kmp() {
    int i, j=-1;
    pos.resize(m);
    pos[0]=-1;
    for( i=1 ; i<m ; i++ ){
        while( j>=0 && p[j+1]!=p[i] )
            j=pos[j];
        if( p[j+1]==p[i] )
            j++;
        pos[i]=j;
    }
}

//conta o número de ocorrências
int kmp() {
    int i, j=-1, ret=0;
    for( i=0 ; i<n ; i++ ){
        while( j>=0 && t[i]!=p[j+1] )
            j=pos[j];
        if( t[i]==p[j+1] )
            j++;
        if( j==m-1 ){
            //ocorrência do padrão em i-m+1
            ret++;
            j=pos[j];
        }
    }
    return ret;
}

int func_kmp() {
    trans_kmp();
    return kmp();
}
```

A implementação mostrada apresenta a função *func_kmp()* que chama a função de pré-processamento do padrão e retorna o valor encontrado pelo método que conta o número de ocorrências. A referência (2) aponta para o trabalho original do algoritmo KMP, que contém explicação detalhada do funcionamento do algoritmo e a prova de sua complexidade linear.

3.3 Algoritmo de Aho-Corasick

O algoritmo de Aho-Corasick tem como objetivo encontrar múltiplos padrões dentro de um texto. Esse algoritmo foi publicado por Alfred V. Aho e Margaret J. Corasick em 1975. O procedimento funciona passando somente uma vez no texto e procurando todos os padrões visados ao mesmo tempo.

Dado inicialmente o conjunto de padrões a serem buscados no texto, o algoritmo de Aho-Corasick realiza um pré-processamento com todos os padrões e gera a partir disso um autômato finito que auxilia na busca de ocorrências. Esse autômato é gerado em complexidade de tempo proporcional à soma dos tamanhos dos padrões. A complexidade de memória pode ser linear também, mas dependendo da implementação pode haver influência do tamanho do alfabeto sobre a quantidade de memória necessária.

Após esse pré-processamento, o algoritmo passa por todo o texto realizando as transições presentes no autômato de acordo com os caracteres lidos. Ao final, a busca pode ser feita em complexidade de tempo linear em relação ao tamanho do texto somado ao número de ocorrências encontradas.

A complexidade alcançada por esse algoritmo até será equiparada à complexidade dos procedimentos que utilizam o vetor de sufixo e o autômato de sufixo, porém com restrições. Para utilizar o Aho-Corasick é necessário o conhecimento prévio de todos os padrões antes de se realizar a busca, ou seja, de maneira *offline*, o que não é necessário com as estruturas de sufixo.

Outra restrição da complexidade que deve ser destacada é a influência do número de ocorrências dos padrões no texto. As estruturas de sufixo não tem essa mesma influência na complexidade se não houver necessidade de retornar todas as posições de ocorrência.

Devido a essas restrições, a execução desse algoritmo não será comparada com as implementações dos algoritmos referentes às estruturas de sufixo, principalmente porque será feita a análise de busca de padrões de modo *online*, ou seja, cada padrão será buscado separadamente. Mais informações sobre o algoritmo de Aho-Corasick podem ser encontradas na referência (3).

3.4 Outros algoritmos

Além dos algoritmos já apresentados, a literatura sobre *strings* ainda apresenta mais opções de algoritmos para a resolução do problema. Dentre eles encontram-se o algoritmo de Rabin-Karp (4), a busca baseada em autômato finito (5) e o algoritmo de Boyer-Moore (6).

Todos os algoritmos citados até agora possuem uma característica em comum: o pré-processamento é baseado nos padrões a serem buscados. Antes de processar a leitura do texto, os algoritmos tentam descobrir informações sobre os padrões para que com poucas leituras do texto seja possível descobrir onde há ocorrências.

Nas próximas seções serão exploradas estruturas de dados que trabalham com o pré-processamento do texto ao invés do padrão. A análise das estruturas mostrarão quais as vantagens em termos de complexidade que podem ser obtidas com essa diferença no procedimento inicial tomado pelo algoritmo.


4 VETOR DE SUFIXOS

4.1 Definição

Um texto de tamanho N possui N sufixos não vazios diferentes. Cada sufixo é caracterizado por uma *substring* que começa em uma posição $i \in \{0, 1, \dots, N-1\}$ e que termina na última posição do texto, a posição $N-1$. Cada sufixo pode ser representado unicamente pela sua posição de início no texto.

O vetor de sufixos de um texto é definido como um vetor de inteiros representando os sufixos do texto, de maneira que os sufixos estejam ordenados em ordem alfabética. A figura a seguir possui um texto de exemplo e exibe a lista de sufixos e o vetor de sufixos referentes ao texto de exemplo.

Figura 1: Exemplo de vetor de sufixos

Texto: $T = "abcbcab"$				
Lista de sufixos			Vetor de sufixos	
Id	Sufixo		Id	Sufixo
0	abcbcab		5	ab
1	bcbcab		0	abcbcab
2	cbcab		6	b
3	bcab		3	bcab
4	cab		1	bcbcab
5	ab		4	cab
6	b		2	cbcab

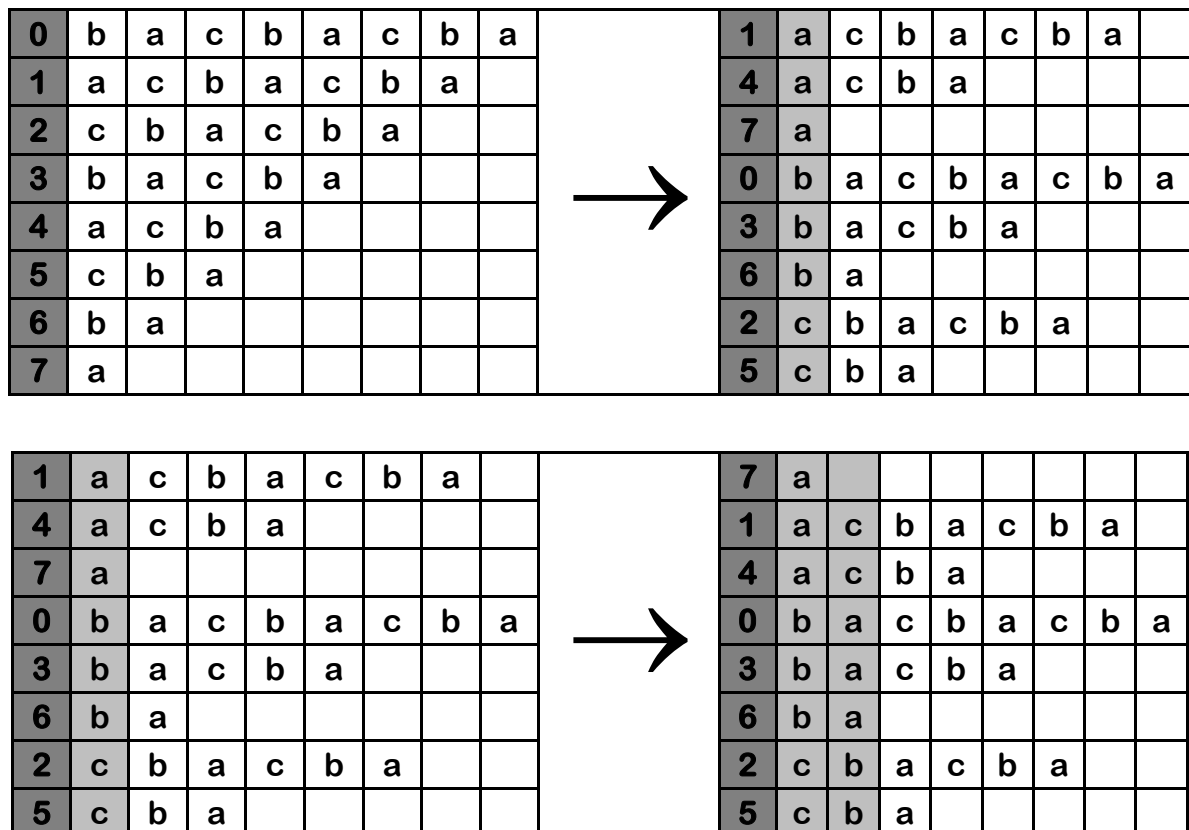
Além do vetor de inteiros representando os sufixos, a figura mostra os sufixos ao lado dos inteiros. Porém, isso é apenas para facilitar a visualização do leitor. Uma implementação eficiente em tempo e memória mantém apenas o vetor que contém os inteiros, pois a soma dos tamanhos dos sufixos é proporcional ao quadrado do tamanho do texto.

4.2 Construção

A construção do vetor de sufixos pode ser feita utilizando algum algoritmo de ordenação comum, *merge sort* por exemplo. Porém, apesar dos algoritmos mais conhecidos de ordenação de inteiros possuírem complexidade de tempo igual a $O(N \log N)$, no caso do vetor de sufixos a comparação entre dois inteiros é feita a partir da comparação de duas *strings*. Como a comparação comum entre dois sufixos possui complexidade $O(N)$, a ordenação do vetor de sufixos teria complexidade $O(N^2 \log N)$.

Para conseguir uma complexidade melhor, é necessário tentar se aproveitar do fato de que as *strings* que precisam ser ordenadas são os sufixos de um mesmo texto. A seguir, será mostrado um algoritmo de construção com complexidade $O(N \log N)$. A ideia básica do algoritmo é começar a construir um vetor de sufixos olhando apenas as primeiras letras de cada sufixo. A partir disso, o segundo passo irá olhar as duas primeiras letras, o terceiro passo irá olhar as quatro primeiras letras, assim por diante, dobrando o número de caracteres analisados até que se alcance N caracteres. A figura a seguir mostra os passos de construção exemplificados para o texto "*bacbacba*".

Figura 2: Exemplo dos passos de construção do vetor de sufixos



7	a									7	a								
1	a	c	b	a	c	b	a			1	a	c	b	a	c	b	a		
4	a	c	b	a						4	a	c	b	a					
0	b	a	c	b	a	c	b	a		6	b	a							
3	b	a	c	b	a					0	b	a	c	b	a	c	b	a	
6	b	a								3	b	a	c	b	a				
2	c	b	a	c	b	a				5	c	b	a						
5	c	b	a							2	c	b	a	c	b	a			

7	a									7	a								
1	a	c	b	a	c	b	a			4	a	c	b	a					
4	a	c	b	a						1	a	c	b	a	c	b	a		
6	b	a								6	b	a							
0	b	a	c	b	a	c	b	a		3	b	a	c	b	a				
3	b	a	c	b	a					0	b	a	c	b	a	c	b	a	
5	c	b	a							5	c	b	a						
2	c	b	a	c	b	a				2	c	b	a	c	b	a			

O número de passos de reordenações desse algoritmo proposto será no máximo $\lceil \log_2 N \rceil$, pois a cada passo se dobra o número de caracteres analisados até que se chegue em N . Cada reordenação será feita com complexidade linear utilizando o *counting sort*, fazendo a complexidade de tempo de construção ser igual a $O(N \log N)$.

Inicialmente deve-se realizar a ordenação dos sufixos pela primeira letra. Esse passo pode ser feito de maneira linear utilizando o *counting sort*, considerando que o tamanho do alfabeto seja menor que o tamanho do texto.

Após o passo inicial, não é tão simples preparar o algoritmo para realizar os próximos passos também utilizando o *counting sort*. Os sufixos devem ser separados em classes numeradas sequencialmente. Dois sufixos devem estar na mesma classe se os mesmos possuírem o prefixo analisado igual. Por exemplo, no primeiro passo os dois sufixos ficam na mesma classe se possuem a primeira letra igual. No terceiro passo, os sufixos possuem a mesma classe se possuem o prefixo de tamanho 4 igual, e assim por diante, seguindo os tamanhos iguais as potências de 2.

O vetor de sufixos estará pronto quando os sufixos forem separados em N classes diferentes, pois nenhum sufixo pode ser igual a outro. A implementação do algoritmo explicado pode ser vista na listagem a seguir.

Listagem 3: Código C++ com construção do vetor de sufixos

```
int n;           // tamanho do texto
int t[MAXN];     // texto
int vs[MAXN];    // vetor de sufixos
int ord[MAXN];   // classes de ordenação dos sufixos
int chave[MAXN]; // chave de comparação para o counting sort
int cnt[MAXN];   // vetor de contagem do counting sort
int aux[MAXN];   // vetor para auxiliar update de vetores
int classes;     // contador de classes de sufixos

void count_sort(){ // counting sort estável
    int i;
    memset(cnt,0,(classes+1)*sizeof(int));
    for( i=0 ; i<n ; i++ ){
        cnt[chave[i]]++;
    }
    for( i=1 ; i<=classes ; i++ ){
        cnt[i]+=cnt[i-1];
    }
    for( i=n-1 ; i>=0 ; i-- ){
        int id=vs[i];
        aux[--cnt[chave[id]]] = id;
    }
    memcpy(vs,aux,n*sizeof(int));
}

void constroi_vetor(){
    int i, j;
    classes=0;
    for( i=0 ; i<n ; i++ ){
        vs[i]=i;
        chave[i]=t[i];
        classes=max(classes,t[i]);
    }
    count_sort();
    ord[vs[0]] = 1;
    classes = 1;
    for( i=1 ; i<n ; i++ ){
        if( t[vs[i]] != t[vs[i-1]] ){
            ord[vs[i]] = ++classes;
        }
        else{
            ord[vs[i]] = ord[vs[i-1]];
        }
    }
}
```

```

for( i=0 ; classes<n ; i++ ){
    for( j=0 ; j<n ; j++ ){
        chave[j] = (j+(1<<i))<n ? ord[j+(1<<i)] : 0;
    }
    count_sort();
    for( j=0 ; j<n ; j++ ){
        chave[j] = ord[j];
    }
    count_sort();
    aux[vs[0]] = 1;
    classes = 1;
    for( j=1 ; j<n ; j++ ){
        int second1 = vs[j]+(1<<i)<n ?
            ord[vs[j]+(1<<i)] : 0;
        int second2 = vs[j-1]+(1<<i)<n ?
            ord[vs[j-1]+(1<<i)] : 0;
        if( ord[vs[j]] == ord[vs[j-1]] &&
            second1==second2 ){
            aux[vs[j]] = aux[vs[j-1]];
        }
        else{
            aux[vs[j]] = ++classes;
        }
    }
    memcpy(ord,aux,n*sizeof(int));
}
}

```

A implementação mostrada na listagem possui a complexidade de tempo igual a $O(N \log N)$, porém possui uma constante de tempo alta, pois são necessários muitos passos lineares durante a execução do algoritmo. A referência (7) mostra uma construção do vetor de sufixos nos mesmos moldes da ideia apresentada, mas apresenta uma constante um pouco mais baixa.

Um fato importante da implementação é que o *counting sort* foi implementado de maneira estável, ou seja, se houver empate entre dois sufixos, o algoritmo mantém a mesma ordem relativa entre eles. Isso é importante, pois em todos os passos exceto no primeiro, é necessária a execução da ordenação duas vezes, e a primeira execução serve justamente para desempatar os sufixos que estão com prefixos analisados iguais até certo passo.

Outra característica interessante da implementação consiste no fato de que o algoritmo percebe quando o vetor de sufixos já está correto analisando o número de classes diferentes de sufixos. Em casos médios, podem ser necessárias menos ordenações do que o pior caso prevê.

Por exemplo, se não houver letras repetidas no texto, só haverá um único passo de ordenação e o vetor de sufixos estará pronto.

Uma segunda opção de implementação possível de construção do vetor de sufixos pode ser feita sem o uso do *counting sort*. Ao invés disso pode-se utilizar um método de ordenação comum, já implementado nas bibliotecas da linguagem C++. O tamanho do código resultante pode ficar menor, porém isso fará com que a complexidade do algoritmo aumente para $O(N \log^2 N)$, já que as ordenações não serão mais feitas de modo linear. A listagem abaixo mostra a implementação desse segundo método.

Listagem 4: Código C++ com opção alternativa de construção do vetor de sufixos

```
bool comp( int a, int b ){
    return chave[a]<chave[b];
}

void constroi_vetor(){
    int i, j;
    for( i=0 ; i<n ; i++ ){
        vs[i] = i;
        chave[i] = t[i];
    }
    sort(vs,vs+n,comp);
    for( i=0 ; ; i++ ){
        classes = 0;
        for( j=0 ; j<n ; j++ ){
            ord[vs[j]] = j>0 && chave[vs[j]]==chave[vs[j-1]]
                ? ord[vs[j-1]] : ++classes;
        }
        if( classes==n ){
            break;
        }
        for( j=0 ; j<n ; j++ ){
            chave[j] = ord[j]*(classes+1LL);
            chave[j] += j+(1<<i)<n ? ord[j+(1<<i)] : 0;
        }
        sort(vs,vs+n,comp);
    }
}
```

Mais informações sobre a construção do vetor de sufixos em complexidade $O(N \log N)$ podem ser encontradas na referência (8). Há também trabalhos bem recentes que mostram como construir o vetor de sufixos em complexidade linear (9) utilizando algoritmos mais complexos.

4.3 Busca por padrão utilizando a estrutura

Após a construção do vetor de sufixos, é possível utilizar a estrutura para encontrar rapidamente um padrão dentro do texto. Uma ocorrência do padrão sempre pode ser considerada como um prefixo de um dos sufixos do texto, ou seja, devem-se encontrar os sufixos que possuem o padrão como um prefixo.

Os sufixos estão ordenados alfabeticamente no vetor de sufixos. Isso pode ser usado para encontrar o padrão no texto utilizando uma busca binária. Comparando o padrão com um sufixo, pode-se determinar se uma possível ocorrência do padrão estaria antes ou depois da posição do sufixo dentro do vetor de sufixos. Essa comparação pode ser feita da mesma maneira que são feitos os testes de ocorrência do algoritmo simples, possuindo complexidade $O(M)$. A listagem a seguir exibe uma função de comparação que compara o padrão com o sufixo de posição *pos*.

Listagem 5: Código C++ com função de comparação entre padrão e um sufixo do texto

```
int compara_vetor( int pos ){
    for( int i=0 ; i<m ; i++ ){
        if( i+pos>=n ){
            return 1;
        }
        else if( p[i]!=t[i+pos] ){
            return p[i]-t[i+pos];
        }
    }
    return 0;
}
```

Essa função de comparação funciona da mesma maneira que a função *strcmp*, da biblioteca da linguagem C. Os três tipos de retorno são:

- Retorno nulo: o padrão ocorre no sufixo analisado.
- Retorno positivo: o padrão é lexicograficamente maior que o sufixo.
- Retorno negativo: o padrão é lexicograficamente menor que o sufixo.

Utilizando a função de comparação, é possível realizar agora uma busca binária para determinar se há uma ocorrência ou não do padrão no texto. A implementação a seguir mostra uma função booleana que retorna se existe pelo menos uma ocorrência.

Listagem 6: Código C++ com busca por padrão utilizando o vetor de sufixos

```
bool busca_vetor() {  
    int ini=0, fim=n-1, meio, aux;  
    while( ini<=fim ) {  
        meio=(fim+ini)/2;  
        aux=compara_vetor(vs[meio]);  
        if( aux==0 ) {  
            return true;  
        }  
        else if( aux>0 ) {  
            ini=meio+1;  
        }  
        else {  
            fim=meio-1;  
        }  
    }  
    return false;  
}
```

Multiplicando a complexidade de tempo da busca pela complexidade de tempo da comparação de sufixo com o padrão, obtém-se a complexidade final do algoritmo de busca, igual a $O(M \log N)$.

Apesar da construção do vetor de sufixos não ser tão simples, o produto final gerado pela construção é apenas um vetor de inteiros. Com isso, o uso da estrutura pode ser feito de maneira simples. Os capítulos 6 e 7 ainda irão especificar mais modos de uso dessa estrutura além da verificação de ocorrência de padrão.

5 AUTÔMATO DE SUFIXOS

5.1 Definição

Assim como o vetor de sufixos, o autômato de sufixos é uma estrutura de dados que pode ser utilizada para pré-processar o texto em que será buscado o padrão. A definição da estrutura é a seguinte:

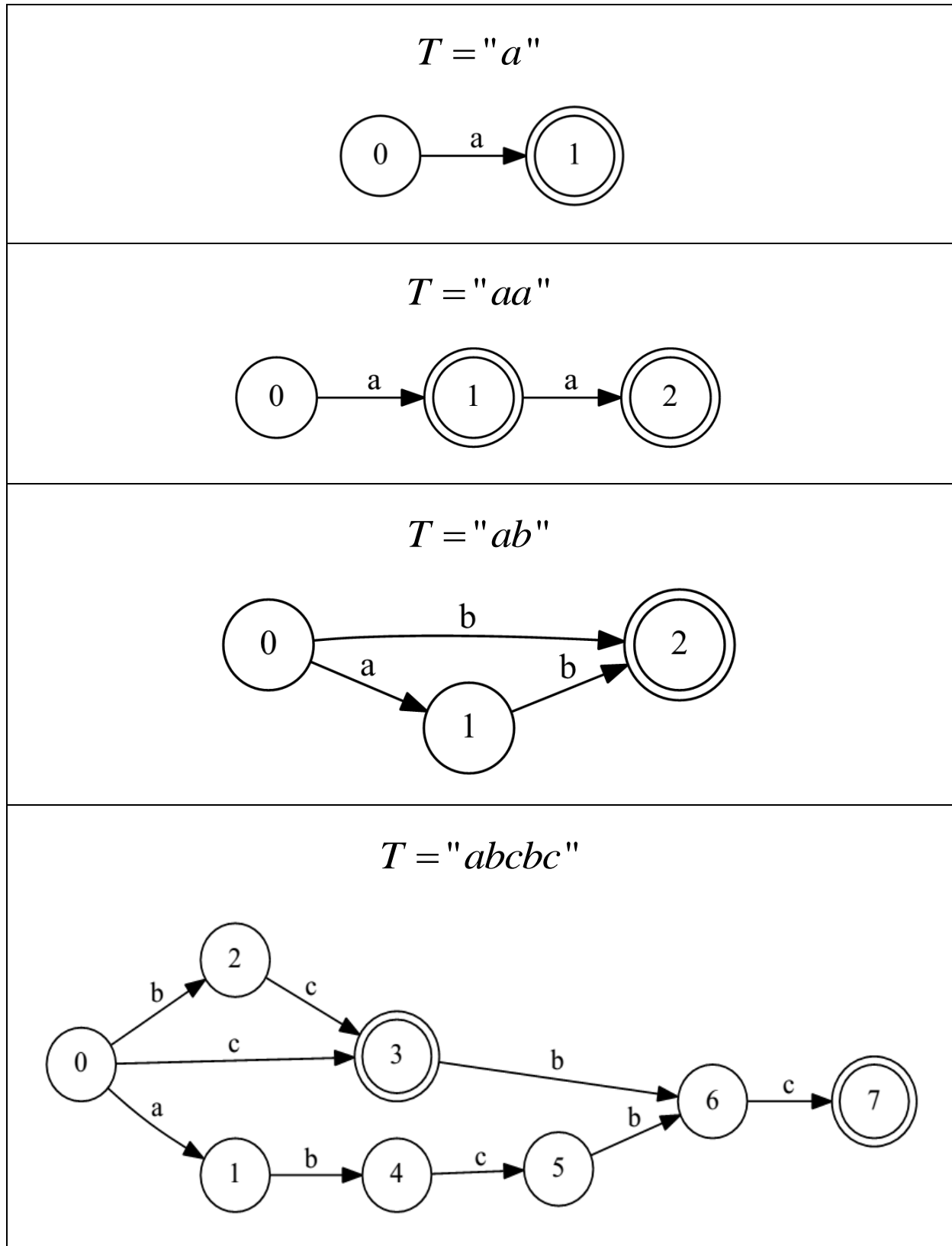
- O autômato de sufixos de um texto é o menor autômato finito determinístico que aceita todos os sufixos do texto.

A definição da estrutura traz uma série de implicações, dentre elas:

- O autômato de sufixos é um grafo direcionado acíclico que possui vértices, que são os estados, e arestas entre os estados, que são as transições.
- Um dos estados desse grafo é o estado inicial. A partir desse estado é possível alcançar todos os outros estados.
- Toda transição desse autômato possui algum símbolo. Todas as transições que se originam de um mesmo estado devem possuir símbolos diferentes.
- Um ou mais estados são marcados como estados finais. Qualquer caminho partindo do estado inicial até um dos estados finais possui a sua sequência de símbolos formando um dos sufixos do texto.
- O autômato de sufixos possui o número mínimo de vértices entre todos os autômatos que satisfazem as condições citadas.

Uma das propriedades mais importantes do autômato de sufixos é a seguinte: uma *string* s é *substring* do texto se e somente se existe um caminho partindo do estado inicial que possua a sequência de símbolos das transições igual a s . Essa propriedade será muito útil para encontrar ocorrências do padrão no texto.

A figura a seguir mostra alguns exemplos do autômato de sufixos para alguns textos pequenos. As ilustrações contêm os estados, as transições com seus símbolos e possui também a marcação dos estados finais indicados por um círculo duplo. O estado inicial do autômato sempre será indicado pelo número zero, esse estado é o único que não contém transições que o alcancem.

Figura 3: Exemplos de autômatos de sufixos

5.2 Construção

Nessa seção será apresentada uma implementação da construção do autômato de sufixos com complexidade linear tanto em tempo como em memória, considerando o tamanho do alfabeto fixo. Apesar da teoria por trás da construção ser complexa, o código não é tão extenso. Esta seção se baseia principalmente na referência (10), além dos primeiros trabalhos publicados referentes a essa estrutura (11, 12).

5.2.1 Estrutura e atributos dos estados

Os estados do autômato serão indicados sequencialmente por $st[i]$, sendo $st[0]$ o estado inicial. Para toda *substring* s que ocorre no texto, existe um caminho no autômato de sufixos que possui a sequência de símbolos igual a s . Sendo $st[i]$ o estado final desse caminho, será dito que a *substring* s corresponde ao estado $st[i]$.

Cada estado então pode corresponder a uma ou mais *strings*, pois cada estado pode ser alcançado a partir do estado inicial por um ou mais caminhos. Será definido o $length(i)$ de cada estado $st[i]$ como o tamanho da maior *string* que corresponde a $st[i]$. Essa maior *string* que corresponde ao estado $st[i]$ será indicada por $longest(i)$. Durante a implementação, o $length(i)$ será indicado por $st[i].len$.

Também será definido o $minlen(i)$ de cada estado $st[i]$ como o tamanho da menor *string* que corresponde a $st[i]$. A partir disso, a construção do autômato de sufixos respeitará as seguintes propriedades:

- Cada estado $st[i]$ corresponde a uma e apenas uma *string* de tamanho L para todo L inteiro pertencente ao intervalo $[minlen(i); length(i)]$.
- Se as *strings* A e B correspondem a um mesmo estado, e o tamanho de A é menor que o tamanho de B , então A é sufixo de B .

Essas propriedades implicam na seguinte afirmação: as *strings* que correspondem ao estado $st[i]$ são iguais aos sufixos de $longest(i)$ com tamanho maior ou igual a $minlen(i)$. Porém, os sufixos de $longest(i)$ com tamanho menor que $minlen(i)$ também devem corresponder a algum estado, pois todo sufixo de $longest(i)$ ocorre no texto. Com isso, será definido o $link(i)$ da seguinte maneira:

- $link(0) = -1$.
- $link(i)$, para $i > 0$, é igual ao número do estado que corresponde ao sufixo de tamanho $minlen(i) - 1$ da *string* $longest(i)$.

O $link(i)$ será representado por $st[i].link$ durante a implementação. Durante a construção do autômato, também será respeitada a seguinte condição: $minlen(i) = length(link(i)) + 1$ para $i > 0$. Isso é equivalente a dizer que $longest(link(i))$ é igual ao sufixo de tamanho $minlen(i) - 1$ da *string* $longest(i)$ para $i > 0$.

Adiante no texto será visto como esses atributos contribuirão com o algoritmo de construção. Portanto, a estrutura de dados precisa guardar os atributos $st[i].len$ e $st[i].link$, além das transições entre os estados. Considerando o alfabeto composto por inteiros, uma das opções possíveis de *struct* para cada estado é apresentada na listagem a seguir.

Listagem 7: Código C++ com implementação de estados de autômatos utilizando *map*

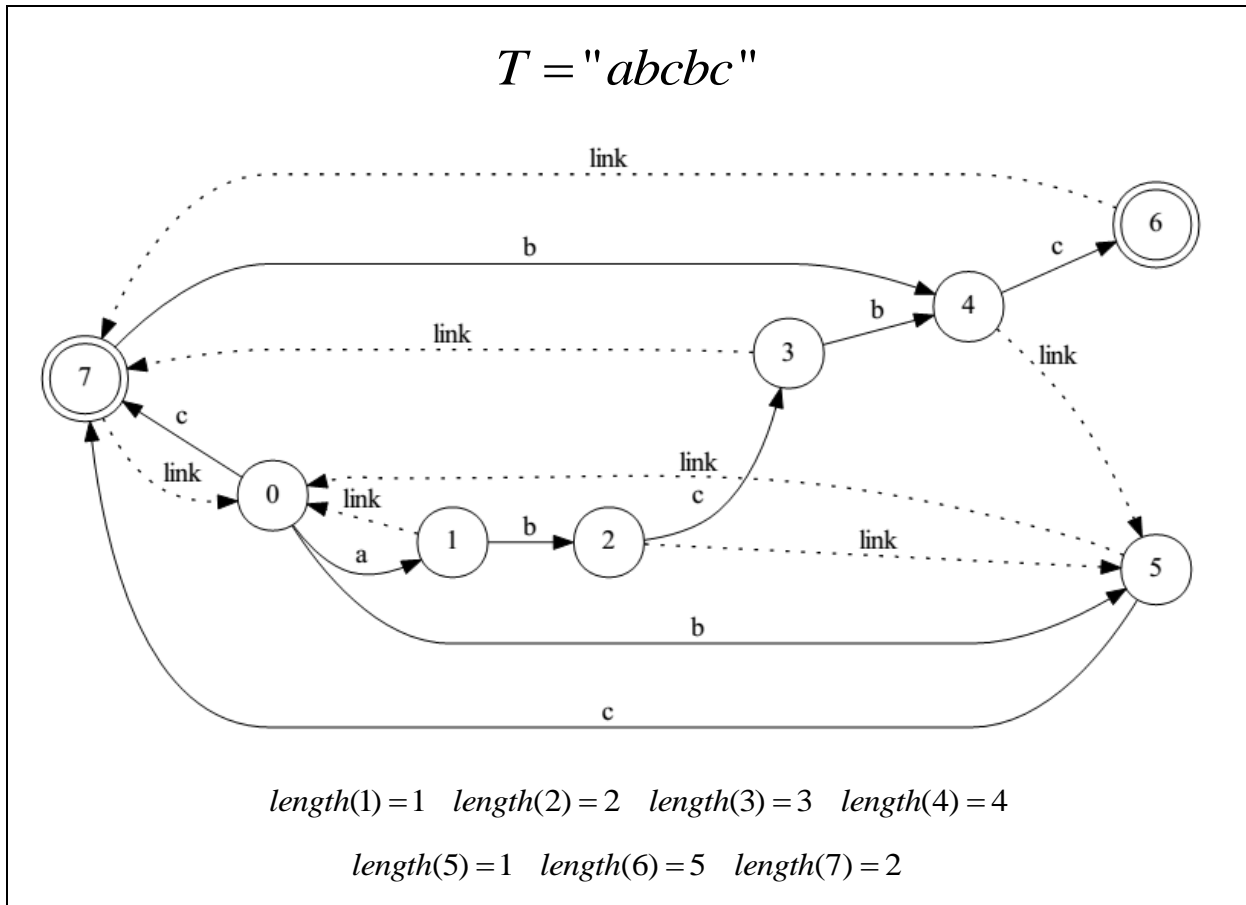
```
struct state{
    int len, link;
    map<int, int> next;
};
```

Com essa opção de *struct*, as transições serão guardadas em um *map*, estrutura presente na biblioteca padrão de C++. Sendo assim, $st[i].next[j]$ indicará o número do estado correspondente à transição do símbolo j saindo do estado $st[i]$. O acesso de $st[i].next[j]$ tem complexidade igual a $O(\log K)$ onde K é o tamanho do alfabeto.

Outra opção de implementação é utilizar um vetor comum ao invés de um *map*. Se essa escolha fosse utilizada, todo estado possuiria uma complexidade de memória $O(K)$, porém o acesso de $st[i].next[j]$ seria realizado em complexidade de tempo $O(1)$.

Perceba que os estados não salvam um atributo para indicar se o estado é terminal ou não. Como essa informação não é tão relevante durante o algoritmo de construção, essa propriedade não será salva no *struct*. Porém, não é difícil determinar quais são os estados terminais a partir dos *links* salvos. Sendo o estado de número i correspondente à *string* composta pelo texto todo, os números dos estados terminais serão iguais a i , $link(i)$, $link(link(i))$, $link(link(link(i)))$, assim por diante até que se chegue no estado inicial.

A figura seguir exemplifica um autômato de sufixos com as representações dos *links* e listagem dos valores de *length*.

Figura 4: Exemplo de autômato de sufixos com representação de *links* e *length*

5.2.2 Construção online

A construção que será proposta aqui será feita de modo *online*. A partir da estrutura pronta para certo texto, será desenvolvido um método que consegue atualizar a estrutura caso seja adicionado ao final do texto um caractere.

Para cada caractere adicionado ao final do texto, a implementação mostrará que serão adicionados um ou dois estados a mais no autômato de sufixos. Isso implicará que o número de estados do autômato de sufixos é no máximo $2N - 1$ (pois o primeiro caractere sempre adiciona apenas um estado). Com isso, a próxima listagem apresenta as variáveis globais que serão utilizadas na implementação.

Listagem 8: Código C++ com variáveis globais do autômato de sufixos

```
state st[2*MAXN]; // vetor que armazena os estados
int sz; // contador do número de estados
int last; // número do estado que corresponde ao texto todo
```

Antes da execução do algoritmo *online*, deve-se inicializar o autômato de forma que o mesmo represente um texto vazio. Para um texto vazio, só é necessário o estado inicial sem transições, como mostra a função de inicialização apresentada a seguir.

Listagem 9: Código C++ com função de inicialização do autômato de sufixos

```
void sa_init() {
    sz = 1;
    last = 0;
    st[0].len = 0;
    st[0].link = -1;
    st[0].next.clear(); // limpa o mapeamento de transições
}
```

Agora é necessário criar a função de extensão do autômato de sufixos caso um caractere C seja adicionado ao final do texto. Pelo menos um novo estado E deve ser criado. Todo sufixo agora possui o último caractere igual a C . Se forem criadas transições do símbolo C partindo do estado inicial e dos estados terminais e destinadas ao novo estado E , o autômato irá aceitar todos os novos sufixos e o estado E passaria a ser o único estado terminal. Porém, o autômato poderia deixar de ser determinístico, ou seja, poderiam ocorrer transições partindo de um mesmo estado e com o mesmo símbolo C .

O algoritmo então começa iterando pelos estados terminais até que se encontre um estado que já possua a transição de símbolo C . A iteração começa por $last$ e vai indo por $link(last)$, $link(link(last))$, assim por diante até que se chegue em -1 ou em algum estado que possua a transição do símbolo C . Caso não se chegue em -1 , seja p o estado em que a iteração acaba. O estado p possui uma transição de símbolo C que vai para um estado q . A partir disso podem ocorrer duas situações diferentes:

- Caso $st[q].len = st[p].len + 1$, isso indica que os novos sufixos iguais aos sufixos de $longest(p)$ com a adição do símbolo C são iguais as *strings* que correspondem a q . Nessa situação, basta fazer com que $link(E)$ seja igual a q .

- Caso contrário, será necessária a criação de um novo estado denominado de clone, pois o estado será quase uma cópia do estado q . Esse estado clone será modificado de tal modo que respeite $st[clone].len = st[p].len + 1$ e partir disso $link(E)$ e $link(q)$ passarão a ser iguais ao clone.

A implementação a seguir mostra com detalhes como realizar as operações necessárias para a extensão do autômato com o símbolo C .

Listagem 10: Código C++ com função de extensão *online* do autômato de sufixos

```
void sa_extend (int c) {
    int cur = sz++; // novo estado a ser criado
    st[cur].len = st[last].len + 1;
    st[cur].next.clear();
    int p; // variável que itera sobre os estados terminais
    for (p=last; p!=-1 && !st[p].next.count(c) ; p=st[p].link) {
        st[p].next[c] = cur;
    }
    if (p == -1) {
        // não ocorreu transição c nos estados terminais
        st[cur].link = 0;
    }
    else { // ocorreu transição c no estado p
        int q = st[p].next[c];
        if (st[p].len + 1 == st[q].len)
            st[cur].link = q;
        else {
            int clone = sz++; // criação do vértice clone de q
            st[clone].len = st[p].len + 1;
            st[clone].next = st[q].next;
            st[clone].link = st[q].link;
            for (; p!=-1 && st[p].next[c]==q; p=st[p].link) {
                // atualização das transições c
                st[p].next[c] = clone;
            }
            st[q].link = st[cur].link = clone;
        }
    }
    // atualização do estado que corresponde ao texto
    last = cur;
}
```

Apesar da teoria por trás do autômato de sufixos ser complicada, a função de extensão do autômato não possui um código tão grande. Mais informações sobre corretude e prova de complexidade do algoritmo podem ser encontradas em (10). Não foi discutido aqui como o

algoritmo garante a propriedade de tamanho mínimo do autômato, entretanto isso pode ser encontrado consultando a referência (13).

5.3 Busca por padrão utilizando a estrutura

Um padrão ocorre no texto se o padrão corresponde a algum estado dentro do autômato de sufixos do texto. Isso já nos dá um algoritmo simples de verificação de ocorrência do padrão, basta seguir um possível caminho partindo do estado inicial e utilizando na ordem os símbolos do padrão, resultando em uma complexidade de tempo $O(M)$ considerando alfabeto fixo ou implementação com vetor ao invés de *map* para salvar as transições. A listagem seguinte mostra como percorrer o caminho para realizar a verificação.

Listagem 11: Código C++ com busca por padrão utilizando o autômato de sufixos

```
bool busca_automato() {  
    int i, pos=0;  
    for( i=0 ; i<m ; i++ ) {  
        if( st[pos].count(p[i])==0 ) {  
            return false;  
        }  
        else {  
            pos=st[pos].next[p[i]];  
        }  
    }  
    return true;  
}
```

Os próximos capítulos irão ainda detalhar outras maneiras de usar o autômato de sufixos para não só descobrir se ocorre ou não o padrão, mas também descobrir outras informações relevantes como, por exemplo, o número de ocorrências, as posições de ocorrências, entre outras aplicações.

6 ANÁLISE DO PROBLEMA DE *STRING MATCHING*

6.1 Busca de um padrão em um texto

Essa seção apresentará os tempos de execução obtidos a partir da realização de vários testes buscando-se um padrão dentro de um texto utilizando quatro métodos já discutidos no trabalho: algoritmo simples, KMP, vetor de sufixos, autômato de sufixos.

As implementações utilizaram os códigos expostos no trabalho, exceto pelo autômato de sufixos, que foi implementado com o uso de vetor ao invés de *map*, assim como explicado na seção 5.2.1. A implementação com vetor apresentou tempos melhores por apresentar tempos menores de acesso às transições dos estados. Em compensação, utiliza mais memória.

A primeira bateria de testes foi realizada com padrões e textos escolhidos de forma totalmente aleatória dentro do alfabeto. Apesar de na prática isso normalmente não ocorrer, esse teste foi realizado para analisar se a teoria prevê corretamente o que ocorre com o tempo de execução. Para a análise, foram colhidos os tempos médios das execuções realizadas durante os testes.

Tabela 1: Tempos médios de execução utilizando texto e padrões aleatórios

Tamanho do texto (N)	100000								
Tamanho do padrão (M)	10			100			1000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Algoritmo	Tempos médios de execução (milissegundos)								
Algoritmo Simples	12.76	7.68	6.14	12.32	7.70	6.18	12.27	7.86	6.16
KMP	13.15	8.29	6.10	12.94	8.42	6.08	13.13	8.84	6.16
Vetor de sufixos	66.65	51.03	47.42	64.41	53.44	47.39	64.17	54.84	46.65
Autômato de sufixos	40.97	40.47	64.02	39.14	42.17	63.69	38.98	43.38	62.74

A partir da observação dos tempos da tabela, é possível fazer as seguintes análises:

- O algoritmo simples e o KMP foram os algoritmos mais rápidos. Apesar da complexidade do algoritmo simples ser maior, o padrão escolhido aleatoriamente possui baixa probabilidade de ocorrer no texto, fazendo com que o teste das posições falhe rapidamente.
- Com o aumento do tamanho do alfabeto, há maior chance de falha do padrão e de sufixos sem prefixos grandes em comum, deixando todos os algoritmos mais rápidos exceto o autômato de sufixos.
- O aumento do alfabeto deixa o autômato de sufixos mais devagar, pois precisa de mais memória por estado e precisa limpar essa quantidade maior de memória.
- As estruturas de sufixos apresentaram os maiores tempos devido à maior constante dos algoritmos.
- Apesar da complexidade de tempo do vetor de sufixos ser maior que a complexidade do autômato de sufixos, os tempos das duas estruturas ficaram próximos, pois o texto aleatório contribui para que não haja muitos passos de reordenação durante a construção do vetor de sufixos.

A próxima tabela mostra os tempos médios obtidos com padrões que ocorrem muitas vezes em um texto. Isso também implica que o texto contém muitos sufixos com grandes prefixos iguais.

Para forçar a ocorrências de padrões, foram utilizados padrões e textos periódicos. Com isso, tenta-se forçar com que os tempos de execução tenham maior proximidade com a complexidade esperada para os algoritmos. A listagem a seguir mostra uma função utilizada nos testes para sortear textos e padrões com períodos que se encaixam.

Listagem 12: Código C++ que sorteia padrões e textos periódicos

```
void sorteio_de_texto_e_padrao_periodicos() {
    int i, period;
    period=a+(rand()%(4*a)); // a = tamanho do alfabeto
    for( i=0 ; i<m ; i++ ) {
        if( i<period )
            p[i]=rand()%a;
        else
            p[i]=p[i%period];
    }
    for( i=0 ; i<n ; i++ )
        t[i]=p[i%period];
}
```

Tabela 2: Tempos médios de execução utilizando padrões com muitas ocorrências no texto

Tamanho do texto (N)	100000								
Tamanho do padrão (M)	10			100			1000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Algoritmo	Tempos médios de execução (milissegundos)								
Algoritmo Simples	24.13	11.85	6.48	153.63	50.19	10.31	2051.68	468.81	55.36
KMP	12.64	11.16	6.52	12.31	11.93	9.84	12.59	12.03	11.83
Vetor de sufixos	110.97	112.95	127.49	104.55	115.81	131.77	102.33	113.16	130.67
Autômato de sufixos	10.19	13.12	37.35	9.68	13.51	37.67	9.59	13.40	37.76

Nessa tabela é possível perceber que os tempos de execução são mais condizentes com as complexidades de tempo analisadas na teoria.

6.2 Busca de múltiplos padrões em um texto

Essa seção analisará o problema de se buscar muitos padrões em um mesmo texto. Nesse caso, as estruturas de sufixos irão levar muita vantagem sobre os outros algoritmos. Se houver muitos padrões e o texto possuir um grande tamanho, pode-se fazer com que os tempos das estruturas de sufixos fiquem muito menores que os tempos de execução dos outros algoritmos. Por isso, os tempos do algoritmo simples e do KMP não serão expostos nessa seção.

Nessa seção será interessante analisar separadamente os tempos de pré-processamento do texto e o tempo de busca do padrão. As tabelas 3 e 4 apresentam os tempos de pré-processamento das duas estruturas, variando-se o tamanho do texto e o tamanho do alfabeto. A tabela 3 contém os tempos de pré-processamento de textos aleatórios, e na tabela 4 foram utilizados textos periódicos.

Tabela 3: Tempos médios de pré-processamento de textos aleatórios

Tamanho do texto (N)	10000			100000			500000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Estrutura	Tempos médios de pré-processamento (milissegundos)								
Vetor de sufixos	6.52	4.47	4.13	101.31	88.37	47.68	885.80	889.69	667.55
Autômato de sufixos	3.65	2.49	5.41	58.75	62.07	61.88	118.68	253.60	353.37

A partir da tabela percebe-se que normalmente a construção do autômato de sufixos é mais rápida que a construção do vetor de sufixos. O aumento do alfabeto normalmente piora o tempo de execução do autômato, enquanto que no vetor de sufixos ocorre o contrário. A próxima tabela apresenta os tempos para textos periódicos.

Tabela 4: Tempos médios de pré-processamento de textos periódicos

Tamanho do texto (N)	10000			100000			500000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Estrutura	Tempos médios de pré-processamento (milissegundos)								
Vetor de sufixos	9.10	9.00	9.50	168.20	148.50	163.70	837.60	1251.70	2611.80
Autômato de sufixos	1.30	1.20	3.70	17.20	15.70	41.40	72.10	72.50	205.30

Na última tabela é possível perceber um aumento do tempo de construção do vetor de sufixos. Esse aumento de tempo é devido ao aumento do número de reordenações necessárias até se garantir o vetor de sufixos ordenado corretamente.

As tabelas 5 e 6 apresentam os tempos de busca por padrão utilizando as estruturas. A tabela 5 utiliza os textos e padrões aleatórios e a tabela 6 utiliza textos e padrões periódicos.

Tabela 5: Tempos médios de busca com padrões e textos escolhidos aleatoriamente

Tamanho do texto (N)	500000								
Tamanho do padrão (M)	1000			10000			50000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Estrutura	Tempos médios de busca por padrão (milissegundos)								
Vetor de sufixos	0.05	0.01	0.00	0.01	0.00	0.00	0.03	0.00	0.02
Autômato de sufixos	0.02	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00

Tabela 6: Tempos médios de busca com padrões e textos escolhidos com período

Tamanho do texto (N)	500000								
Tamanho do padrão (M)	1000			10000			50000		
Tamanho do alfabeto (K)	2	5	50	2	5	50	2	5	50
Estrutura	Tempos médios de busca por padrão (milissegundos)								
Vetor de sufixos	0.05	0.04	0.06	0.57	0.65	0.64	5.84	6.33	6.63
Autômato de sufixos	0.08	0.06	0.06	0.85	0.83	1.24	14.53	14.42	15.07

Para os padrões aleatórios, é possível ver que a checagem de ocorrência é muito rápida, independente do tamanho do padrão, pois o teste de verificação falha rápido nos dois casos. Para os padrões que ocorrem muitas vezes no texto, já é possível perceber uma demora maior na execução dos algoritmos. Como nesse caso o padrão ocorre muitas vezes no texto, o vetor de sufixos acabou ficando mais rápido que o autômato, pois a busca binária encontra uma das ocorrências do padrão em poucos passos.

6.3 Número de ocorrências de um padrão em um texto

Nessa seção será visto que o número de ocorrências do padrão no texto pode ser encontrado utilizando as estruturas de sufixo. Além disso, esse número pode ser encontrado na mesma complexidade que o teste de verificação da ocorrência do padrão.

Para o vetor de sufixos basta encontrar o primeiro e o último sufixo no vetor que possuem o padrão como prefixo. A partir disso, todas as ocorrências acontecem nos sufixos do vetor que estão entre esses dois sufixos. Isso pode ser feito utilizando duas buscas binárias, resultando na complexidade de tempo $O(M \log N)$.

Para o autômato de sufixos é um pouco mais complexo. É necessário adicionar alguns passos na construção da estrutura. Para cada estado $st[i]$ deve-se calcular um inteiro $st[i].cnt$. Esse inteiro informará justamente o número de ocorrências das *substrings* que correspondem ao estado $st[i]$.

Todo estado que não foi originado a partir de clonagem representa uma posição no texto. Com isso, inicialmente pode ser definido $st[i].cnt$ como 0 para os estados que são clones, e definido como 1 para os outros estados comuns. Se uma *substring* corresponde a um estado $st[i]$, ela ocorre pelo menos uma vez no texto, e a posição de $st[i]$ no texto representa a última posição dessa ocorrência.

Para encontrar mais ocorrências, deve-se lembrar de que se $st[j]$ é um estado comum e se $link(j) = i$, todas as *substrings* que correspondem a $st[i]$ possuem uma ocorrência que acaba na posição do texto referente ao estado $st[j]$. Com isso, o procedimento necessário a ser executado é somar o valor $st[j].cnt$ à variável $st[st[j].link].cnt$ para todo $j > 0$.

Esse procedimento de soma deve ser feito começando pelos estados $st[j]$ com os maiores valores de $st[j].len$ e iterando-se até os estados com os menores valores. Após isso, se uma *substring* corresponde ao estado $st[i]$, então o número de ocorrências dessa *substring* no texto é igual a $st[i].cnt$.

O cálculo dos valores de $st[i].cnt$ pode ser feito a partir da ordem topológica baseada nos *links* dos estados. A implementação a seguir mostra como realizar uma busca em largura para realizar os cálculos necessários. Esse procedimento possui complexidade linear baseada no tamanho do texto.

Listagem 13: Código C++ com cálculo de contagem no autômato de sufixos

```

int grau[2*MAXN]; // contagem de links
queue<int> fila; // fila para busca em largura

void calcula_contagem() {
    memset(grau, 0, sz*sizeof(int));
    while( !fila.empty() )
        fila.pop();
    for( int i=1 ; i<sz ; i++ )
        grau[st[i].link]++;
    for( int i=1 ; i<sz ; i++ ){
        if( grau[i]==0 )
            fila.push(i);
    }
    while( !fila.empty() ){
        int k=fila.front();
        if( k>0 ){
            st[st[k].link].cnt+=st[k].cnt;
            if( (--grau[st[k].link])==0 )
                fila.push(st[k].link);
        }
        fila.pop();
    }
}

```

6.4 Posições das ocorrências de um padrão em um texto

Nessa seção será visto que as posições das ocorrências do padrão no texto podem ser encontradas utilizando as estruturas de sufixo. Além disso, as posições podem ser encontradas na mesma complexidade que o teste de verificação da ocorrência do padrão, porém com o acréscimo de complexidade linear baseada no número de ocorrências do padrão no texto.

Na última seção foi explicado que para o vetor de sufixos pode-se encontrar no vetor o primeiro e o último sufixo que possuem o padrão como prefixo. A partir disso, todas as ocorrências acontecem nos sufixos do vetor que estão entre esses dois sufixos. Para encontrar todas as ocorrências, basta iterar sobre todas as posições do vetor de sufixos entre os sufixos encontrados. As buscas binárias resultam na complexidade de tempo $O(M \log N)$, e a iteração sobre as posições possui complexidade de tempo linear baseada no número de ocorrências do padrão.

Assim como para encontrar o número de ocorrências do padrão utilizando autômato de sufixos, será necessário adição de dados aos estados para encontrar as posições de ocorrência.

Pode-se definir para os estados não clonados um valor de $st[i].pos$ igual à posição do caractere referente à criação do estado $st[i]$. Se o padrão corresponde ao estado $st[i]$, então há uma ocorrência do padrão na posição $st[i].pos - M + 1$.

Para encontrar as outras ocorrências, é necessário salvar os *links* do autômato de forma invertida, ou seja, para cada estado $st[i]$ é necessário ter a lista de todos os estados $st[j]$ tal que $st[j].link$ é igual a i . A partir disso, uma busca em profundidade a partir do estado $st[i]$ usando esses *links* invertidos passará por todas as ocorrências do padrão no texto. Se a busca alcançou um estado $st[j]$ não clonado, então o padrão ocorre na posição $st[j].pos - M + 1$.

Para se alcançar o estado que corresponde ao padrão, é necessária uma complexidade de tempo $O(M)$. A busca em profundidade possui complexidade de tempo linear baseada no número de ocorrências do padrão.

7 PROBLEMAS RELACIONADOS

Esse capítulo explora o uso das estruturas de sufixo para a resolução de problemas mais complexos envolvendo *string matching*. Essas estruturas são alvo de questões pertencentes a competições de programação como a ACM International Collegiate Programming Contest (14) e competições *online* em sites como o *Codeforces* (15). Esse capítulo contém três desses problemas para análise.

Os problemas apresentados aqui serão resolvidos utilizando o autômato de sufixos. O vetor de sufixos também é capaz de resolver os problemas desse capítulo eficientemente, porém isso envolve o cálculo rápido do maior prefixo comum entre dois sufixos (*longest common prefix*). Esse cálculo não foi abordado nesse trabalho, mas pode ser visto com detalhes na referência (16).

7.1 Número de subsequências diferentes em um texto

Dado um texto de tamanho N , deseja-se saber quantos padrões diferentes ocorrem nesse texto. O número de *substrings* de um texto é $O(N^2)$, portanto um algoritmo simples que teste todas as *substrings* possuiria no mínimo complexidade de tempo quadrática.

Utilizando o autômato de sufixos do texto, é possível resolver o problema em tempo linear. Todo padrão que ocorre no texto corresponde a algum estado do autômato. O conjunto de *strings* que correspondem a um estado $st[i]$ são os sufixos de $longest(i)$ com tamanho maior ou igual a $minlen(i)$. Com isso, cada estado corresponde a um número de *strings* igual a $length(i) - minlen(i) + 1$. Isso permite que a implementação mostrada a seguir resolva o problema.

Listagem 14: Código C++ com função de contagem de substrings do autômato de sufixos

```
long long conta_substrings() {
    long long ret=0;
    for( int i=1 ; i<sz ; i++ ){
        ret+=st[i].len-st[st[i].link].len;
    }
    return ret;
}
```


7.2 Maior padrão que ocorre múltiplas vezes em um texto

Dado um texto de tamanho N e um inteiro K , determinar o tamanho do maior padrão que ocorre pelo menos K vezes no texto. Assim como no problema anterior, um algoritmo simples que testa todos os padrões possuiria no mínimo complexidade $O(N^2)$, podendo chegar a uma complexidade de até $O(N^3 \log N)$ se fosse utilizado um *map* para detectar os padrões que ocorrem mais de uma vez.

Com o autômato de sufixos, é possível resolver o problema em tempo linear utilizando o cálculo de $st[i].cnt$ que foi mostrado na seção 6.3. Se certo estado $st[i]$ possui o valor de $st[i].cnt$ maior ou igual a K , então a *string* $longest(i)$ ocorre pelo menos K vezes no texto. Nessa situação, uma possível resposta para o problema poderia ser $st[i].len$. Para encontrar o maior tamanho, basta iterar por todos os estados à procura do maior valor de $st[i].len$, respeitando a condição $st[i].cnt \geq K$, como mostra o código a seguir.

Listagem 15: Código C++ com cálculo de tamanho do maior padrão repetido

```
int maior_tamanho_repetido( int vezes ){
    int ret=0;
    for( int i=1 ; i<sz ; i++ ){
        if( st[i].cnt>=vezes ){
            ret=max( ret, st[i].len );
        }
    }
    return ret;
}
```

7.3 Maior padrão que ocorre em dois textos

Dados dois textos, determinar o tamanho do maior padrão que ocorre nos dois textos. Assim como os outros dois problemas dessa seção, um algoritmo simples necessita testar todas as *substrings* de um texto. Além disso, verificar a existência das mesmas no outro texto. Se os textos têm tamanhos N e M , a complexidade necessária para isso seria $O(N^2M)$ se, por exemplo, fosse utilizado KMP para a verificação da *substring* no segundo texto.

Uma das estratégias para resolver esse problema consiste em construir o autômato de sufixos do primeiro texto e tentar encontrar a sequência de caracteres do segundo texto nas transições do autômato construído.

Após a construção do autômato do primeiro texto, pode-se realizar uma iteração por todos os caracteres do segundo texto. Durante esse processo, pode-se manter apontado o estado $st[i]$ correspondente ao maior sufixo dos caracteres lidos que pertence também ao primeiro texto. A cada caractere C lido, deve-se procurar a transição do símbolo C no estado $st[i]$. Caso não exista a transição, deve-se ir para o estado $link(i)$ até que se encontre a transição.

Durante o procedimento descrito, deve-se manter qual é o tamanho do sufixo encontrado. Quando existe a transição do caractere lido C , o tamanho deve ser incrementado em uma unidade. Caso contrário, o tamanho passa a ser igual a $length(link(i))$. O código a seguir apresenta a implementação desse procedimento. A função mostrada recebe como parâmetros os dois textos utilizando a estrutura de *string* da biblioteca padrão de C++. A função possui complexidade de tempo linear baseada na soma dos tamanhos dos textos.

Listagem 16: Código C++ com cálculo do maior tamanho de padrão presente em dois textos.

```
int maior_tamanho_em_comum( string s, string t ){
    // Constrói o autômato com o primeiro texto
    sa_init();
    for (int i=0; i<(int)s.length(); i++)
        sa_extend (s[i]);

    int estado = 0, tamanho = 0, maior = 0;

    // Passando pelos caracteres do segundo texto
    for (int i=0; i<(int)t.length(); ++i) {
        while( estado && ! st[estado].next.count(t[i]) ){
            estado = st[estado].link;
            tamanho = st[estado].len;
        }
        if (st[estado].next.count(t[i])) {
            estado = st[estado].next[t[i]];
            tamanho++;
        }
        if (tamanho > maior)
            maior = tamanho;
    }
    return maior;
}
```

8 COMENTÁRIOS FINAIS

8.1 Conclusão

Nesse trabalho, foi analisado o problema de *string matching* e diversas estratégias de solução. As principais técnicas estudadas se baseavam em algoritmos envolvendo as estruturas relativas aos sufixos do texto, o vetor de sufixos e o autômato de sufixos. As soluções envolvendo as estruturas de sufixos foram comparadas a algoritmos clássicos usados na resolução desse tipo de problema.

O estudo da teoria mostrou que as estruturas de sufixo conseguem resolver de maneira eficiente o problema de *string matching*, principalmente quando é necessária a busca de múltiplos padrões em um mesmo texto. Quando há muitos padrões a serem buscados em um texto, ficou comprovado que as estruturas de sufixos são muito mais eficientes do que abordagens usando algoritmos como o KMP e o algoritmo simples de busca.

Foi mostrado também que as estruturas de sufixos são capazes de resolver variações complexas do problema de *string matching*. Alguns problemas envolvendo essas variações foram apresentados e resolvidos utilizando implementações com poucas linhas de código. Além disso, as soluções mostradas apresentam complexidade de tempo muito menor do que as soluções mais simples que não utilizam as estruturas de dados apresentadas no trabalho.

8.2 Trabalhos Futuros

Esse trabalho abre portas para que sejam feitos aprofundamentos do estudo das estruturas de sufixos apresentadas. É possível encontrar muitos outros problemas difíceis relacionados a *string* que poderiam ser resolvidos utilizando essas estruturas. Portanto, uma recomendação possível seria analisar mais variações complexas de *string matching*, incluindo a resolução desses problemas utilizando as duas estruturas estudadas.

Além dessa recomendação, sugere-se um trabalho envolvendo também uma terceira estrutura de sufixos, a árvore de sufixos (*suffix tree*). Poderia ser analisada a relação entre as três estruturas e um estudo comparativo para determinar as melhores possibilidades de uso de cada uma das estruturas.

9 REFERÊNCIAS

- 1 SINGLA, Nimisha; GARG, Deepak. **String Matching Algorithms and their applicability in various applications**. International Journal of Soft Computing and Engineering (IJSCE). ISSN: 2231-2307, Volume-I, Issue-6, January 2012.
- 2 KNUTH, Donald; MORRIS, James H., JR; PRATT, Vaughan. **Fast pattern matching in strings**. SIAM Journal on Computing 6 (2): 323-350 (1977).
- 3 AHO, Alfred V.; CORASICK, Margaret J. **Efficient string matching: An aid to bibliographic search**. Communications of the ACM 18 (6): 333-340 (1975).
- 4 KARP, Richard M.; RABIN, Michael O. **Efficient randomized pattern-matching algorithms**. IBM J. Res. Develop. Vol. 31 No. 2 March 1987.
- 5 ICS 161: **Design and Analysis of Algorithms**. Lectures notes for February 22, 1996. Disponível em: <<http://www.ics.uci.edu/~eppstein/161/960222.html>>. Acesso em: 19 out. 2013.
- 6 BOYER, Robert S.; MOORE, J Strother. **A fast string searching algorithm**. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762-772. ISSN: 0001-0782.
- 7 IVANOV, Maxim. **Suffix array, construction in $O(N \log N)$ and applications**. Disponível em: <http://e-maxx.ru/algo/suffix_array>. Acesso em: 20 out. 2013.
- 8 MANBER, Udi; MYERS, Gene. **Suffix arrays: a new method for on-line string searches**. In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms 90 (319): 327 (1990).
- 9 NONG, Ge; ZHANG, Sen; CHAN, Wai Hong. **Linear suffix array construction by almost pure induced-sorting**. 2009 Data Compression Conference, p. 193. ISBN 978-0-7695-3592-0.
- 10 IVANOV, Maxim. **Suffix automata, construction in $O(N)$ and applications**. Disponível em: <http://e-maxx.ru/algo/suffix_automata>. Acesso em: 20 out. 2013.
- 11 BLUMER, Anselm; BLUMER, J.; EHRENFEUCHT, Andrzej; HAUSSLER, David; MCCONELL, Ross M. **Linear size finite automata for the set of all subwords of a word – an outline of results**. Bulletin of the EATCS 01/1983; 21:12-20 (1983).
- 12 BLUMER, Anselm; BLUMER, J.; EHRENFEUCHT, Andrzej; HAUSSLER, David; CHEN, M.J.; SEIFERAS, J. **The smallest automaton recognizing the subwords text of A**. Theoretical Computer Science, Volume 40, 1985, Pages 31-55, ISSN 0304-3975.
- 13 NERODE, A. **Linear automaton transformations**. Proceedings of the American Mathematical Society Vol. 9, No. 4 (Aug., 1958), pp. 541-544.

14 **ACM International Collegiate Programming Contest.** *The ACM-ICPC International Collegiate Programming Contest Web Site sponsored by IBM.* Disponível em: <<http://www.acmicpc.org/>>. Acesso em: 22 out. 2013.

15 **Codeforces.** Codeforces (c) Copyright 2010-2013 Mike Mirzayanov. Disponível em: <<http://codeforces.com/>>. Acesso em: 22 out. 2013.

16 KASAI, Toru; LEE, Gunho; ARIMURA, Hiroki; ARIKAWA, Setsuo; PARK, Kunsoo. **Linear-time longest-common-prefix computation in suffix arrays and its applications.** A. Amir and G.M. Landau (Eds.): CPM 2001, LNCS 2089, pp. 181-192, 2001.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO TC	2. DATA 26 de novembro de 2013	3. REGISTRO N° DCTA/ITA/TC-103/2013	4. N° DE PÁGINAS 53
5. TÍTULO E SUBTÍTULO: Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching”			
6. AUTOR(ES): Gabriel Luís Mello Dalalio			
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica - ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Algoritmos, Estruturas de dados, String matching, Vetor de Sufixos, Autômato de sufixos, Busca de texto, Competições de programação			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Algoritmos; Estruturas(processamento de dados); Vetores(Matemática); Teoria de autômatos; Computação; Engenharia de software.			
10. APRESENTAÇÃO: <div style="float: right; text-align: right;"> X Nacional Internacional </div> ITA, São José dos Campos. Curso de Graduação em Engenharia de Computação. Orientador: Prof. Armando Ramos Gouveia. Publicado em 2013			
11. RESUMO: <p>A dissertação propõe-se a analisar algoritmos e estruturas de dados para resolver problemas relacionados a <i>string matching</i>. Inicialmente, este texto apresenta alguns algoritmos mais conhecidos para resolver o problema de <i>string matching</i>, que se caracteriza em achar uma sequência contínua de caracteres dentro de outra sequência maior de caracteres. Contudo, esses algoritmos não conseguem resolver eficientemente algumas variações do problema, como por exemplo, quando se deseja achar várias palavras dentro de uma mesma sequência de caracteres. Com isso, são apresentadas duas estruturas de dados capazes de lidar melhor com essas variações do problema, o vetor de sufixos e o autômato de sufixos. Após a apresentação das estruturas, são analisadas algumas variações do problema em questão, incluindo variações simples e também variações mais complexas baseadas em problema de competições passadas de programação.</p>			
12. GRAU DE SIGILO: <div style="display: flex; justify-content: space-between;"> (X) OSTENSIVO () RESERVADO () CONFIDENCIAL () SECRETO </div>			