colloquium exactarum

ANÁLISE E APLICAÇÃO DE ESTRUTURAS DE SUFIXOS NA RESOLUÇÃO DO *STRING MATCHING*

ANALYSIS AND APPLICATION OF SUFFIX STRUCTURES IN THE RESOLUTION OF STRING MATCHING

Guilherme Henrique Santos Miranda[1]; Leandro Luiz de Almeida[2]; Danillo Roberto Pereira[3]; Mario Augusto Pazoti[4]; Francisco Assis da Silva[5]

[1]Universidade do Oeste Paulista – UNOESTE, Faculdade de Informática – FIPP, Presidente Prudente, SP.
E-mail: chico@unoeste.br

RESUMO – *String Matching* é o problema que busca responder a seguinte pergunta: "É possível encontrar determinado padrão dentro de um texto?". É um problema amplamente estudado na Ciência da Computação e também na Biologia Computacional, devido à existência de suas diferentes modificações em ferramentas de pesquisa e também no processamento de cadeias de DNA. Já existem algoritmos que alcançaram a solução ótima para responder a pergunta do problema, entretanto tais soluções não possuem a mesma eficiência nas extensões e variações do problema. Dessa forma, diversas pesquisas tem estudado estruturas de dados relativas aos sufixos do texto para alcançar soluções que sejam capazes de resolver variações complexas do string matching. O presente trabalho realiza um estudo e análise aprofundada sobre a eficiência de dessas estruturas: a árvore de sufixos e o autômato de sufixos. Algoritmos clássicos também são abordados e comparados às estruturas enquanto o trabalho é discorrido. As análises seguem critérios estatísticos, tempos de execução e complexidade de algoritmos para obter maior grau de confiança nos resultados.
Palavras-chave: string matching; árvore de sufixos; autômato de sufixos; estrutura de dados; pesquisa em textos.

ABSTRACT – String Matching is the problem that seeks to answer the following question: "Is it possible to find a certain pattern within a text?". It is a problem widely studied in Computer Science and also in Computational Biology, due to the existence of its different modifications in research tools and in the processing of DNA chains. Already there are algorithms that have reached the optimal solution to answer the question of the problem, however such solutions do not have the same efficiency when the problem is extended or variated. Thus, several studies have used approaches with data structures related to text suffixes to reach solutions that are capable of solving complex string matching variations. The present work makes a study and efficiency analysis of these structures: the suffix tree and the suffix automaton. Classical algorithms are also approached and compared with the structures while work is discussed. The analyzes follow statistical criteria, execution times and complexity of algorithms to obtain a greater confidence degree in the results.
***Keywords*:** string matching; suffix tree; suffix automaton; text processing; pattern search.

Recebido em: 09/05/2017
Revisado em: 30/08/2017
Aprovado em: 10/10/2017

1. INTRODUÇÃO

String Matching é um dos problemas de *strings* – definida como cadeia de caracteres – mais explorados da Ciência da Computação. Tal problema consiste em encontrar uma dada *string* (padrão) dentro de outra *string* (texto) de tamanho superior ou igual ao tamanho do padrão. Essa temática é voltada a várias outras aplicações que necessitam encontrar um ou vários padrões em um dado texto (SINGLA; GARG, 2012). São alguns exemplos: (i) Alinhamento de sequências de DNA; (ii) Compressão de Textos; (iii) Consultas em Banco de Dados e (iv) Ferramentas de Pesquisa.

É possível alcançar uma solução para o problema com um algoritmo simples, que se baseia em força bruta. Nessa solução é tentado combinar todos os M caracteres do padrão a partir de cada um dos N caracteres do texto. Como ao fazer isso existe a possibilidade de encontrar um padrão não desejado de tamanho próximo a M para cada um dos N caracteres do texto, o algoritmo resulta em uma complexidade de ordem quadrática, o que não é desejável para a maior parte das aplicações.

Algoritmos com complexidade linear de tempo já foram alcançados, como é o caso do algoritmo Knuth-Morris-Pratt (KMP) (KNUTH; MORRIS; PRATT, 1977). A solução alcançada pelo KMP é ótima, visto que apenas o tempo de leitura do padrão e do texto demandam tempo O(N+M). Outros algoritmos (BOYER; MOORE, 1997; KARP; RABIN, 1987) também alcançaram soluções satisfatórias. Embora tais algoritmos sejam eficazes para a busca de um padrão, não mantém a mesma eficiência quando o problema é estendido para, por exemplo, encontrar a posição em que diferentes padrões ocorrem em um mesmo texto. Ou, ainda, não são capazes de resolver variações do problema, como quando se tem a necessidade de encontrar o maior padrão comum a múltiplos genomas (FARIAS, 2010), onde a clássica solução com programação dinâmica não é eficiente.

Desta forma, abordagens com o uso de estruturas de dados relativas aos sufixos do texto (estruturas de sufixos) têm sido cada vez mais utilizadas para alcançar soluções eficientes mesmo quando o problema de *string matching* é modificado.

Este trabalho propõe a realização de uma análise aprofundada sobre duas estruturas de sufixos, com intuito de identificar melhores possibilidades de uso para as mesmas. As estruturas abordadas são denominadas árvore de sufixos e autômato de sufixos. Ambas são apresentadas junto suas respectivas teorias e algoritmos. Após, são submetidas a uma bateria de testes sobre diferentes problemas de *string matching*. O algoritmo KMP também é submetido aos mesmos testes. Posteriormente, são utilizados cinco critérios para comparação dos resultados: (i) tempo médio de execução; (ii) desvio padrão; (iii) teste estatístico de Wilcoxon; (iv) complexidade de tempo no pior caso; (v) complexidade de memória no pior caso. Depois da comparação, alguns outros problemas relacionados à temática são apresentados, junto a como solucioná-los com o uso das estruturas.

O detalhamento do trabalho está estruturado em outras 7 Seções. A Seção 2 apresenta trabalhos relacionados à temática. A Seção 3 apresenta os aspectos da árvore de sufixos. A Seção 4, de semelhante modo, apresenta o autômato de sufixos. A Seção 5 detalha os critérios de avaliação utilizados sobre os resultados dos testes. Os resultados dos testes, junto à análise, são expostos na Seção 6. Outros problemas relacionados são apresentados junto a suas respectivas soluções com o uso das estruturas na Seção 7. Enfim, a Seção 8 evidência a conclusão final do trabalho e sugestões de trabalhos futuros.

2. TRABALHOS RELACIONADOS

Serão apresentadas nesta seção as referências bibliográficas que proporcionaram embasamento teórico e inspiração à realização desse trabalho.

2.1. Árvore de sufixos

Os primeiros algoritmos eficientes para a construção da árvore de sufixos foram dados por Weiner (1973) e posteriormente por McCreight (1976). Desde então, a estrutura de dados tem sido estudada extensivamente por décadas (GUSFIELD, 1997; DELCHER et al., 1999; SETUBAL; BRAUNING, 2006).

Diversos autores (WEINER, 1973; UKKONEN, 1995; GIEGERICH; KUTZ, 1997; FARACH, 1997) definem a árvore de sufixos como *tries* (árvores ordenadas utilizadas geralmente para armazenar caracteres) compactas ou árvores PATRICIA *(Practical Algorithm To Retrieve Information Coded In Alphanumeric)* para um conjunto não vazio de sufixos de um determinado texto.

Árvores de sufixos foram aplicadas em importantes problemas de strings, tais como encontrar a maior repetição de substrings (WEINER, 1973), computar estatísticas de substrings (APOSTOLICO; PREPARATA, 1985), encontrar todas as repetições em uma string (APOSTOLICO; PREPARATA, 1983) e comparação de strings (EHRENFEUCHT; HAUSSLER, 1985).

A estrutura também tem sido utilizada na compressão de textos (RODEH; PRATT; SHIMON, 1981), inclusive uma pesquisa realizada por Fraser (1984) utilizou a árvore de sufixos na compressão de códigos Assembly.

Em alinhamento de sequências de DNA, variação do *string matching* na biologia computacional, é possível utilizar a árvore de sufixos para obtenção de uma abordagem rápida na combinação de sequências que contém milhões de nucleotídeos (DELCHER et al., 1999). Outros métodos de se resolver o problema, como programação dinâmica, algoritmo BLAST (ALTSCHUL et al., 1990) (ALTSCHUL et al., 1997) ou mesmo o MegaBLAST (modificação do BLAST para grandes sequências) por exemplo, são inviáveis para sequências de tamanho exorbitante, devido ao gasto de memória (SETUBAL; BRAUNING, 2006).

Para soluções que envolvem casamento exato de caracteres (desconsidera sufixos e prefixos) é mais adequada a utilização de árvores esparsas de sufixos. Esta estrutura é detalhada na dissertação do Sacomoto (2011), juntamente com a implementação ótima da estrutura.

Outros trabalhos mais contemporâneos que utilizam a árvore de sufixos em diferentes variações do *string matching* estão constantemente sendo realizados. No trabalho do Apostolo e Cunial (2014), o vetor de sufixos e a árvore de sufixos são destacados como estruturas poderosas para a realização de várias aplicações. Ainda, o uso das estruturas é dito como incontável e que ainda outros usos estão sendo descobertos. Em um outro trabalho realizado por Dalalio (2013), a árvore de sufixos é recomendada como estudo para trabalhos futuros, junto ao autômato de sufixos e o vetor de sufixos, com a sugestão de se analisar a relação entre as estruturas e as melhores possibilidades de uso de cada uma delas.

Um artigo publicado recentemente, realizado por Apostolico et al. (2016), é intitulado “40 anos da árvore de sufixos”. O artigo trata sobre as variantes da árvore de sufixos e as, não só úteis, como também surpreendentes, características que a estrutura carrega. Na publicação são dados diferentes exemplos de diversas áreas onde a árvore de sufixos foi o centro das pesquisas por anos. E ainda, o tamanho do impacto dos estudos da estrutura na Ciência da Computação para a pesquisa de textos e também na bioinformática são destaques na publicação, sendo afirmado que tais áreas não seriam as mesmas sem a existência da estrutura. Estruturas como o autômato de sufixos e o vetor de sufixos também são mencionadas.

2.2. Autômato de sufixos

Os primeiros trabalhos publicados sobre Autômato de Sufixos foram realizados por Blumer et al. (1983, 1985). A partir de então diversos outros pesquisadores

utilizaram esta estrutura para a resolução de variantes do *string matching.*

Um algoritmo *online* para a combinação exata de *strings* é tratado por Faro e Lecroq (2012). A importância de encontrar todas as ocorrências de um padrão no texto é novamente destacada e uma abordagem para a solução é realizada com o uso do autômato de sufixos. O novo algoritmo apresentado mostrou a possibilidade de ser muito rápido em casos mais práticos. O resultado das experiências e mais detalhes de como o autômato de sufixos foi abordado nesse trabalho pode ser visto na referência (FARO; LECROQ, 2012).

O trabalho de Dalalio (2013) realizou um estudo teórico sobre a resolução do *string matching.* Nele, foram implementados os seguintes algoritmos: (i) solução com força bruta; (ii) KMP; (iii) vetor de sufixos e (iv) autômato de sufixos para a resolução do problema. Todas as implementações foram comparadas conforme o tempo de execução obtido por cada um dos algoritmos. Foi concluído por ele que os algoritmos KMP e com força bruta tiveram melhor desempenho que as estruturas na busca por um padrão. Contudo, na busca por múltiplos padrões, as estruturas de sufixos tiveram um desempenho significativamente superior. Detalhes sobre o vetor de sufixos podem ser encontrados tanto em (DALALIO, 2013) quanto em (MANBER; MYERS, 1990), um dos trabalhos pioneiros no assunto.

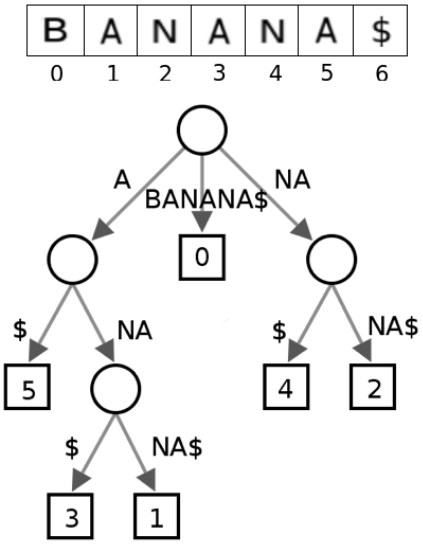
Na pesquisa realizada por Giaquinta (2015), o tema *string matching* é novamente abordado, mas desta vez direcionado a grandes textos, sobre pequenos alfabetos. O método utilizado para a solução do problema no trabalho da Giaquinta (2015) utiliza uma variação não-determinística do algoritmo KMP e o autômato de sufixos.

Ainda, em um artigo realizado por Badkobeh et al. (2016), o autômato de sufixos é utilizado para alcançar uma solução eficiente no problema do máximo anti-expoente de fatores em um palíndromo. A solução obtida alcança complexidade de tempo linear sobre o tamanho do texto para alfabeto fixo e, sendo A o tamanho do alfabeto, a solução possui O (N log(A)) em alfabetos largos, se mostrando bastante superior a solução ingênua para o problema, que possuía complexidade cúbica de tempo.

3. ÁRVORE DE SUFIXOS

Uma árvore de sufixos é uma estrutura de dados que armazena todos os sufixos de uma *string* em memória. Cada sufixo é representado por uma aresta na árvore e as folhas indicam a posição de início de cada sufixo na *string* armazenada. A representação gráfica da estrutura é mostrada na Figura 1.

Figura 1. Árvore de Sufixos para *string* BANANA.



Formalmente, uma árvore de sufixos para um texto S de tamanho N é uma árvore que segue as seguintes definições:

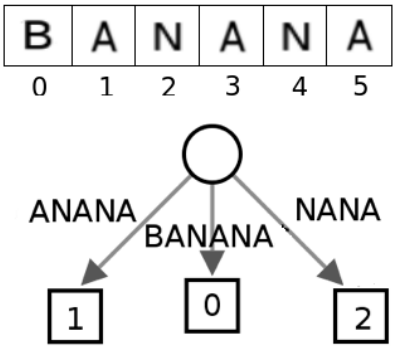
- Há exatamente N folhas, numeradas de 1 até N;
- Com exceção da raiz, cada nó interno possui no mínimo dois filhos;
- Cada aresta é rotulada com uma subsequência contígua de caracteres não vazia de S;
- Não podem coexistir dois rótulos que partam do mesmo nó e iniciam com o mesmo caractere;
- A concatenação dos rótulos encontrados no caminho da raiz a folha *i* resulta no sufixo $s_i \ldots s_n$, para $1 \leq i \leq N$.

Um caractere especial (como o $ no caso da Figura 1) é concatenado a S para assegurar que a quantidade de folhas presentes na árvore seja igual a N. Isso também garante que nunca existirá um sufixo que é prefixo de outro na árvore. Apesar de graficamente representada com uma *string* em cada aresta, na implementação da estrutura, apenas os índices de início e fim de cada sufixo de S são alocados em memória, reduzindo a complexidade de espaço para O(N).

3.1. Árvore de Sufixos Implícita

O conceito de árvore de sufixos implícita é necessário para o entendimento da seção seguinte. Assim sendo, uma árvore de sufixos para um dado texto S é definida como uma árvore obtida de uma árvore de sufixos comum, na qual inicialmente é retirado o símbolo terminal (o caractere $ da árvore mostrada na Figura 1). Após, são removidas as arestas que possuem rótulos vazios e em seguida todo vértice que não possui pelo menos dois filhos é removido e, se houver aresta que parte dele, esta tem seu rótulo concatenado ao rótulo da aresta que leva até ele. Não é garantido que a árvore implícita terá menos folhas que a comum. Tal afirmação é verdade se, e somente se, existe em S algum sufixo que é prefixo de outro sufixo. A Figura 2 mostra a árvore de sufixos implícita referente ao mesmo texto da Figura 1.

Figura 2. Árvore de Sufixos Implícita para o texto BANANA.

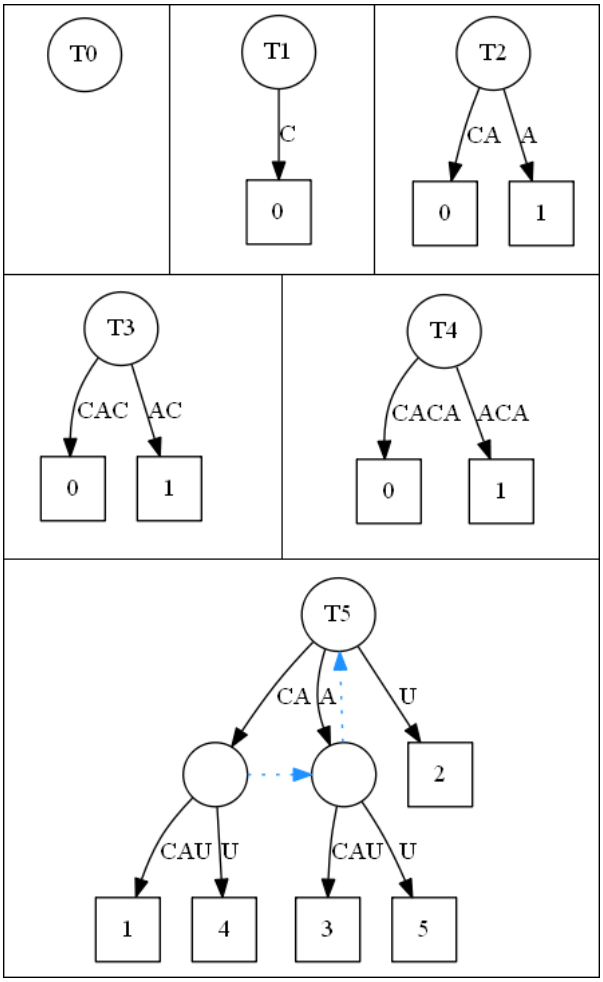


3.2. Algoritmo de Ukkonen

O algoritmo de Ukkonen (1995), dentre os algoritmos pioneiros para a construção de uma árvore de sufixos em tempo linear (WEINER, 1973; MCCREIGHT, 1976; UKKONEN, 1995; FARACH, 1997), é o único também *online* e assim terá sua teoria abordada nesta seção, juntamente ao código de sua implementação.

O algoritmo processa o texto da esquerda para a direita e constrói uma árvore de sufixos implícita T_i para cada fase *i* ($1 \leq i \leq N$). Uma árvore T_i aloca todos os prefixos $s_1 \ldots s_i$ e é construída a partir de T_{i-1}. Assim, T_0 é definida como uma árvore vazia, T_1 será a árvore construída para o 1º caractere, T_2 para o 2º, até a árvore T_n ser construída para o enésimo caractere. A Figura 3, logo abaixo, ilustra o processo descrito para o texto CACAU.

Figura 3. Fases da construção de uma Árvore de Sufixos proposta por Ukkonen. Os *suffix links* estão representados na cor azul.



Em cada fase *i* do algoritmo há *i* expansões a serem realizadas. Para cada expansão *j* de uma fase *i*, um algoritmo ingênuo realizaria a travessia do rótulo s*j* . . . s_i combinando caractere a caractere a partir da raiz, resultando numa complexidade de tempo $O(N^3)$. Para que seja possível alcançar complexidade linear de tempo é necessário observar com atenção as regras da estrutura e também realizar a implementação de alguns truques. Por se tratar de uma teoria muito extensa, esses detalhes não serão abordados aqui, mas podem ser vistos no próprio trabalho de Ukkonen (1995) e também no trabalho de Gusfield (1997). O código implementado no trabalho foi desenvolvido seguindo a teoria de ambas as referências.

4. AUTÔMATO DE SUFIXOS

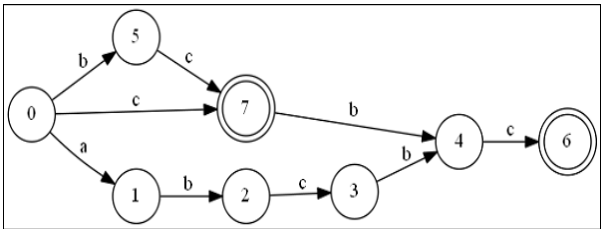
Nesta seção serão discutidos os aspectos teóricos necessários para a construção e entendimento do autômato de sufixos. O texto discorrido a seguir possui embasamento teórico principalmente no trabalho de Inanov (2008). Em nenhum momento será abordada a prova de complexidade e corretude do algoritmo, mas é possível encontrar tais informações no mesmo trabalho do Inanov (2008). Outros dois trabalhos que foram os primeiros referentes à estrutura (BLUMER et al., 1983, 1985) também forneceram conhecimento para o assunto discorrido nesta seção.

4.1. Definições e funcionamento

Assim como uma árvore de sufixos, o autômato de sufixos é uma estrutura que realiza o pré-processamento de um texto T e armazena todos os seus sufixos em memória.

A estrutura é definida como o menor autômato finito determinístico que aceita todos os sufixos de T. Ainda, um autômato finito é um grafo acíclico dirigido onde seus vértices são chamados de estados – terminais ou não – e suas arestas são chamadas de transições. É chamado de estado inicial o primeiro estado criado para a construção do autômato e é o único estado que não recebe transições de nenhum outro. É possível alcançar todos os estados do autômato com a realização de transições entre os estados a partir do estado inicial. Um padrão faz parte de T se, e somente se, é possível transitar a partir do estado inicial até um estado final combinando os caracteres do padrão com os caracteres dos rótulos das transições. A Figura 4 apresenta a representação gráfica de um autômato de sufixos.

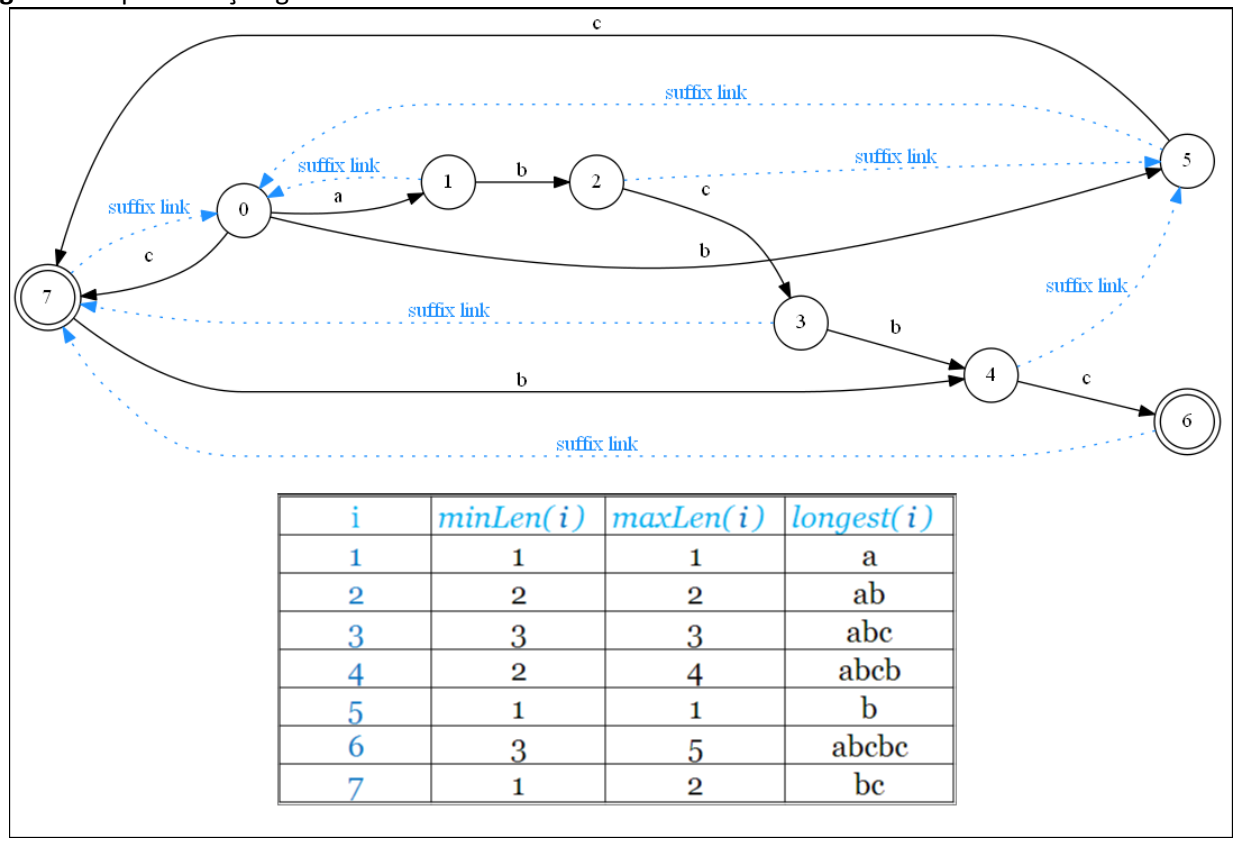
Figura 4. Representação gráfica de um autômato de sufixos



Por conveniência, os estados do autômato serão definidos como st_i ($1 \leq i \leq K$), sendo st_0 o estado inicial e st_k o último estado criado no autômato. Cada estado do autômato, com exceção do st_0, é alcançado por, pelo menos, um caminho. Assim, cada estado pode representar o término de um ou vários padrões. Será dito que *maxLen(i)* representa o tamanho do maior padrão que corresponde a st_i; *longest(i)* é o padrão correspondente a *maxLen(i)* e que *minLen(i)* representa o tamanho do menor padrão do mesmo st_i. Todos os padrões que correspondem a um estado st_i possuem tamanhos distintos dentro do intervalo *maxLen(i)* e *minLen(i)* e essa propriedade deve ser seguida durante a construção do autômato. Essa propriedade implica que os padrões correspondentes a st_i são todos sufixos de *longest(i),* mas é interessante notar que nem todos os sufixos de *longest(i)* correspondem ao mesmo st_i, já que, como dito anteriormente, apenas estão presentes padrões de tamanhos distintos entre *maxLen(i)* e *minLen(i)*. Para que todos os sufixos de *longest(i)* estejam de fato presentes no autômato, é necessário a criação de uma nova transição, que será definida como *suffixLink(i)*. Com essa nova transição, outras três propriedades devem ser seguidas:

1. *suffixLink(0)* é igual a -1;
2. *suffixLink(i)* é igual ao índice do estado correspondente ao sufixo de tamanho *minLen(i)*-1;
3. *minLen(i)* ($i \neq 0$) é igual a *maxLen(suffixLink(i))*.

Figura 5. Representação gráfica detalhada de um autômato de sufixos



i	*minLen(i)*	*maxLen(i)*	*longest(i)*
1	1	1	a
2	2	2	ab
3	3	3	abc
4	2	4	abcb
5	1	1	b
6	3	5	abcbc
7	1	2	bc

A Figura 5 apresenta um autômato de sufixos e seus respectivos *maxLen(i), minLen(i)* e *suffixLink(i)*.

A implementação do algoritmo segue de acordo com o texto discorrido nessa seção. O código considera a utilização de alfabetos de tamanho fixo, com um total de A símbolos. Assim, é possível alocar as transições de um estado com um vetor estático de A posições, uma para cada símbolo distinto. Dessa forma, cada estado do autômato possui complexidade de espaço O(A), mas o acesso a cada transição é realizado em tempo O(1).

Para alfabetos de tamanhos variáveis, é possível utilizar, ao invés do vetor, a classe map existente na biblioteca Standard Template Library na linguagem C++. Essa segunda abordagem reduz a complexidade de memória, já que é realizada a alocação somente para os símbolos que apareceram no texto. Apesar disso, o acesso a uma transição em um map possui complexidade de tempo logarítmica em relação à quantidade de transições armazenadas no estado, tendendo a aumentar o tempo de execução do algoritmo.

5. METODOLOGIA DE TESTES SOBRE O *STRING MATCHING*

Essa seção evidência a metodologia utilizada na criação da base de dados utilizadas nos testes e também análise dos algoritmos utilizando as estruturas de dados e também do KMP para a realização dos seguintes problemas:

1. Construção das estruturas;
2. Pesquisa de um padrão dentro de um texto;
3. Posição de todas as ocorrências de múltiplos padrões dentro de um mesmo padrão.

5.1. Critérios para geração da base de dados

A base de dados é composta por diferentes casos de testes gerados para cada um dos tipos de problema citados. Os casos de teste possuem alfabetos, textos e padrões de tamanhos distintos. Os textos e padrões são classificados em aleatórios e periódicos

dentro do alfabeto. Gerar textos e padrões de forma aleatória torna possível averiguar se os tempos de execução dos algoritmos estão de acordo com a teoria. Quando gerados de forma periódica, é possível realizar a análise dos algoritmos no caso da existência de ocorrências múltiplas de um mesmo padrão no texto, fazendo com que o tempo de execução seja mais próximo da complexidade dos algoritmos.

5.2. Critérios para avaliação dos algoritmos

5.2.1 Complexidade de Algoritmos

A análise de algoritmos a partir de sua complexidade fornece uma medida de desempenho objetiva que, diferente da medida por tempo de execução, não depende de fatores como a qualidade do compilador, processador da máquina ou habilidade do programador. Continuará sendo levado em consideração a complexidade de pior caso dos algoritmos, denotada por O(K), sendo K: a quantidade de operações realizadas pelo algoritmo, para o caso da complexidade de tempo; a quantidade de espaço em memória para o caso da complexidade de espaço.

5.2.2 Tempo médio de execução

A fim de se obter uma análise com maior precisão, o tempo de execução também é considerado. Com isso, é possível analisar quais fatores levam ao pior caso do algoritmo, ou seja, quais fatores levam a complexidade do algoritmo.

5.2.3 Desvio Padrão

5.2.4 Teste Estatístico de Wilcoxon Pareado

O teste de Wilcoxon Pareado é um teste estatístico que possibilita verificar se existem diferenças significantes entre os resultados (tempos médios) obtidos na realização dos testes dos algoritmos. A descrição do funcionamento do teste, logo abaixo, é embasada no trabalho de Wilcoxon (1945) e também em Demsar (2006). O trabalho de Triola (2008) também proporcionou conhecimento ao conteúdo apresentado e não pode deixar de ser citado.

Funcionamento:

Sejam X e Y duas amostras mutuamente independentes, de tamanho N, com seus elementos sendo representados por x_i e y_i ($1 \leq i \leq N$), respectivamente. Um terceiro conjunto de diferenças $D = \{d_1, d_2, ..., d_n\}$ é definido. Em D, quaisquer valores $d_i = 0$ são removidos e, assim, o valor de N é redefinido para a quantidade restante de elementos em D. A partir de tais definições, o teste segue com os seguintes passos:

- Ordenar o conjunto D em ordem ascendente, de acordo com o valor absoluto dos elementos;
- Substituir os valores de cada d_i para os valores de seus respectivos postos (posto $d_i = i$, para $d_i > 0$, ou posto $d_i = -i$, para $d_i < 0$), entretanto para todo posto d_i com valores iguais, um único posto deve ser associado, com valor igual à média dos postos de cada d_i envolvido no empate;
- Calcular W+, que é a soma de todos os postos positivos;
- Calcular W-, que é a soma do valor absoluto de todos os postos negativos;
- Seja T = *min*(W+, W-);
- Determinar a estatística de teste, que é:

$$Z = \frac{T - \frac{N(N+1)}{4}}{\sqrt{\frac{N(N+1)(2N+1)}{24}}}$$

- Obter o p-valor, encontrado na tabela da distribuição normal padrão;
- Considerar duas hipóteses:

  H_0: ambas as amostras levam a mesma eficiência;

  H_1: as amostras possuem eficiências distintas;
- Se o p-valor for menor ou igual ao nível de confiança, considerado 0,05, então H_0 deve ser tida como verdade. Caso contrário, H_0 é descartada e a conclusão é que as amostras possuem eficiências distintas.

6. ANÁLISE DE EFICIÊNCIA SOBRE O *STRING MATCHING*

A seção apresenta os resultados obtidos nos testes realizados com o uso de algoritmos da árvore de sufixos, do autômato e também do algoritmo KMP. Em cada subseção os resultados são expostos e a análise é realizada em seguida.

6.1. Construtor das estruturas de sufixos

É interessante abordar o tempo que é gasto com o pré-processamento do texto realizado pelas estruturas separadamente e notar as variações que ocorrem conforme o aumento do tamanho do texto e alfabeto. As tabelas abaixo apresentam o tempo médio gasto na execução dos construtores das estruturas de sufixos, com diferentes tamanhos de textos.

Tabela 1. Execução do construtor das estruturas com textos aleatórios

Tamanho Texto	**10000**				
Tamanho Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	7.92 (8.63)	2.96 (6.26)	4.95 (7.51)	5.10 (7.80)	5.25 (7.61)
Autômato de Sufixos	8.79 (8.08)	8.03 (8.71)	5.20 (7.59)	3.92 (7.28)	4.49 (7.10)
Tamanho Texto	**100000**				
Tamanho do Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	96.79 (7.54)	71.46 (8.12)	69.40 (9.46)	64.38 (6.35)	76.09 (5.43)
Autômato de Sufixos	116.32 (7.77)	87.13 (7.99)	76.36 (10.41)	59.71 (6.24)	52.56 (7.53)
Tamanho Texto	**500000**				
Tamanho Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	560.56 (29.25)	455.89 (17.68)	420.58 (17.37)	425.30 (17.59)	474.92 (16.09)
Autômato de Sufixos	319.50 (11.81)	521.27 (7.77)	442.37 (7.71)	401.37 (8.69)	354.35 (7.99)

A partir dos dados na Tabela 1 e 2, é possível perceber que:

- Uma vez que, quanto maior o alfabeto, maior a quantidade de memória alocada e

maior tempo despendido para limpar esse espaço de memória, esperava-se um aumento considerável no tempo de execução das estruturas, o que não ocorreu;

Tabela 2. Execução do construtor das estruturas com textos periódicos

Tamanho Texto	**10000**				
Tamanho Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	5.21 (7.49)	11.46 (8.88)	5.47 (8.00)	6.37 (8.02)	8.89 (8.45)
Autômato de Sufixos	4.36 (7.01)	1.36 (4.59)	4.34 (7.03)	4.37 (7.14)	3.73 (6.78)
Tamanho Texto	**100000**				
Tamanho do Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	70.78 (8.47)	75.15 (6.53)	88.91 (10.85)	86.80 (8.43)	109.49 (4.80)
Autômato de Sufixos	38.88 (7.84)	37.02 (7.59)	43.72 (8.60)	38.01 (7.75)	38.50 (7.81)
Tamanho Texto	**500000**				
Tamanho Alfabeto	2	5	10	20	50
Estrutura	**Tempo Médio em Milissegundos (Desvio Padrão)**				
Árvore de Sufixos	399.04 (33.45)	447.39 (47.30)	448.13 (25.74)	497.44 (27.30)	626.01 (24.91)
Autômato de Sufixos	192.00 (8.39)	210.07 (22.92)	199.37 (7.24)	199.94 (7.94)	202.60 (6.66)

- Com textos periódicos, o tempo de execução do autômato de sufixos diminuiu, enquanto o tempo de execução da árvore de sufixos aumentou;
- Quanto maior o tamanho do texto, mais homogênea é a amostra tanto dos tempos de execução do autômato, quanto dos tempos de execução da árvore;
- O tempo médio de execução das estruturas vai mais de acordo com suas respectivas complexidades de tempo;
- Apesar de ambas possuírem complexidade linear de tempo, há uma divergência nos tempos de execução das estruturas, que pode ser explicada por fatores como o tempo gasto para alocação de memória no *heap* na implementação da árvore de sufixos e também as constantes de suas complexidades.
- A hipótese de que tanto os tempos de execução da árvore de sufixos, quanto o do autômato de sufixos possuem a mesma eficiência foi tida como verdade no teste

estatístico de Wilcoxon para todos os testes realizados com alfabetos de 50 caracteres, em textos periódicos e aleatórios. A hipótese foi rejeitada apenas para textos aleatórios, com alfabeto de 50 caracteres e textos com tamanho 100000, onde o *p-value* obtido foi 0.5101, acima do nível de confiança.

6.2. Posição das ocorrências de um padrão em um texto

Essa seção analisa o problema de encontrar todas as posições que um padrão ocorre em um texto. Nesse caso o KMP possui complexidade de tempo O(N+M) e as estruturas de sufixos resolvem o problema em tempo O(N+M+OCC), sendo OCC a quantidade de ocorrências do padrão pesquisado. A partir da realização de testes com diferentes tamanhos de padrões é possível averiguar o quão as três soluções divergem nos tempos de execução. As figuras 6 e 7 apresentam os resultados de tais testes.

Figura 6. Gráfico com o tempo médio de execução (milissegundos) na busca de todas as posições em que um padrão aparece no texto periódico de 500000 caracteres [a-z, A-X]

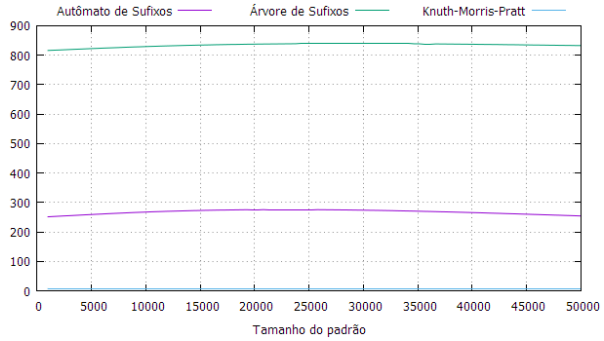
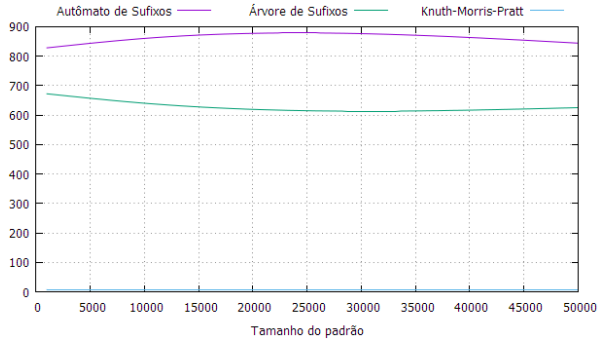


Figura 7. Gráfico com o tempo médio de execução (milissegundos) na busca de todas as posições em que um padrão aparece em um texto aleatório de 500000 caracteres [a-z, A-X]



Ao analisar o gráfico, percebe-se que o algoritmo KMP teve desempenho superior as estruturas de sufixos tanto em textos aleatórios, quanto em textos periódicos, por conta da constante OCC de suas complexidades. É possível ver também que a variação nos tempos de execução de acordo com o aumento do tamanho dos padrões é pequena, principalmente com textos periódicos. Com texto e padrão aleatório, há menos chance de combinações entre os caracteres, reduzindo o tempo da árvore de sufixos, mas aumentando o tempo de execução do autômato. Esse aumento no tempo de execução do autômato com textos aleatórios se deve principalmente ao tempo de construção da estrutura, discutido na seção anterior.

6.3. Posição das ocorrências de múltiplos padrões em um texto

O mesmo problema abordado na seção anterior é tratado, entretanto agora são utilizados múltiplos padrões e um mesmo texto. Nesse caso, seja K a quantidade de padrões, as estruturas de sufixos despendem de tempo O (N + (M+OCC)*K) para encontrar todas as posições de cada um dos padrões, enquanto o KMP leva tempo O((N+M)*K). Nos testes, o valor de K é igual a 1000. Os gráficos para análise são expostos nas Figuras 8 e 9.

Figura 8. Gráfico com o tempo médio de execução (milissegundos) na busca de todas as posições de múltiplos padrões em um texto periódico de 500000 caracteres [a-z, A-X]

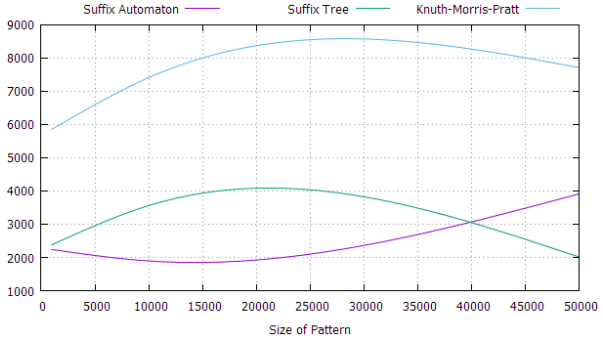
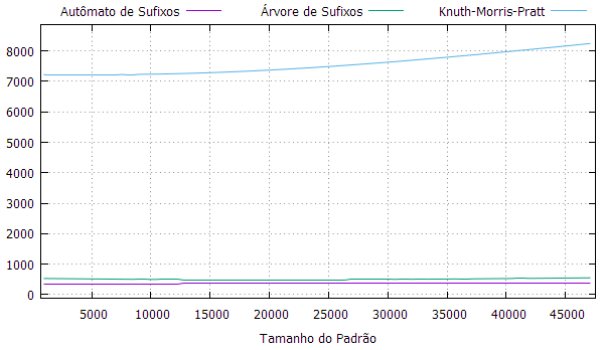


Figura 9. Gráfico com o tempo médio de execução (milissegundos) na busca de todas as posições de múltiplos padrões em um texto aleatório de 500000 caracteres [a-z, A-X]



É possível visualizar que o tempo médio de execução das estruturas de sufixos tende a ser menor que o tempo gasto pelo KMP. Isso se deve ao fato ter que percorrer novamente o texto no algoritmo KMP para cada padrão pesquisado. Quanto maior a quantidade de padrões, mais as estruturas de sufixos se sobressaem em questões de eficiência, quando comparadas aos algoritmos clássicos que necessitam processar o texto sempre que buscam por um padrão.

7. RESOLUÇÃO DE PROBLEMAS RELACIONADOS

Essa seção apresenta as modificações e variações do problema de *string matching* que foram exploradas no trabalho, explicando como resolver cada um dos problemas com o uso das estruturas.

7.1 Busca por um padrão

O algoritmo de busca por um padrão em um texto funciona de modo semelhante em ambas às estruturas. Um padrão ocorre no autômato de sufixos e na árvore de sufixos se é possível seguir um caminho que parte do estado inicial do autômato ou na raiz da árvore e utiliza todos os símbolos do padrão, na ordem em que eles aparecem, para travessia entre as transições ou rótulos das arestas.

7.2 Posições das ocorrências de um padrão

Nesse problema deseja-se não apenas saber se um determinado padrão existe, mas também todas as posições em que ele

apareceu no texto. Nesse caso, os algoritmos das estruturas funcionam de forma diferente:

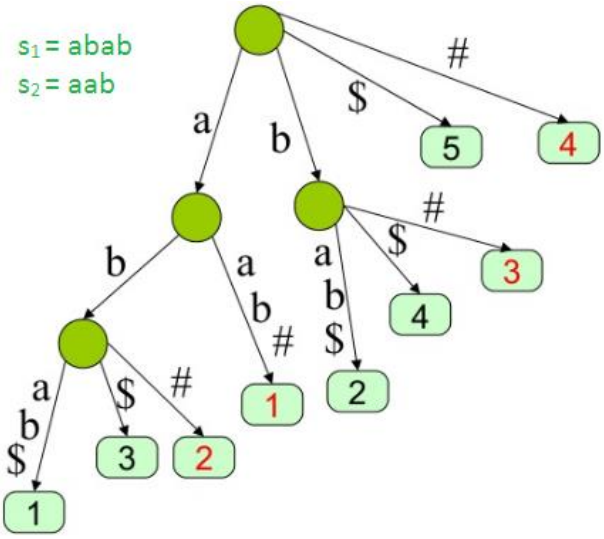
- Árvore de Sufixos: inicialmente deve-se verificar a existência do padrão. Caso este exista, é necessário realizar uma busca em profundidade a partir do nó correspondente ao padrão encontrado e o sufixo de cada folha encontrada indicará uma ocorrência do padrão.
- Autômato de Sufixos: para encontrar as posições de ocorrências do padrão com o uso do autômato é necessário armazenar algumas informações adicionais nos estados. Assim, define-se o campo *pos* na estrutura e para todo estado st_i não clonado esse campo terá valor igual à posição do caractere referente à criação do st_i. Há uma ocorrência do padrão na posição *pos – M + 1*, onde *pos* refere-se ao st_i correspondente à busca do padrão. Para que seja possível encontrar as demais ocorrências, outro campo *reverseLink* deve ser adicionado a estrutura dos estados. Esse campo nada mais é que do que uma lista de *suffixLinks* salvos de forma invertida em todo st_i, ou seja, uma lista com todos os estados st_j tal que o campo *suffixLink* aponta para o estado st_i. Após adicionadas as informações, é necessário a realização de uma busca em profundidade a partir do estado st_i correspondente ao padrão utilizando-se os *reverseLinks.* Cada estado não clonado que for visitado corresponde a uma ocorrência do padrão na posição *pos – M + 1*.

7.3 Maior *substring* comum entre duas *strings*

O problema da maior *substrings* comum é um problema bastante conhecido e explorado na computação e bioinformática. No contexto do problema, deseja-se encontrar a maior sequencia subsequente de caracteres que existe entre um conjunto de *strings*. Esse problema é tratado no projeto de *primers* de Farias (2016), também com o uso de uma árvore de sufixos. Farias (2016) comenta que os algoritmos utilizados no trabalho não suportaram grandes cadeias, por se tratar de uma biblioteca já implementada.

Para resolver tal problema com o uso da árvore de sufixos, primeiramente é necessário entender o conceito de árvore de sufixos generalizada, que é o nome dado para uma árvore de sufixos construída para duas ou mais *strings.* Considere duas *strings,* S_1 e S_2. Uma árvore de sufixos generalizada para tais *strings* é o resultado da construção de uma árvore de sufixos para $S_1\$S_2\#$ (S_1 concatenado a S_2, separados por caracteres especiais). O mesmo algoritmo pode ser utilizado, com a adição de apenas uma alteração, referente aos índices que indicam o término dos rótulos das folhas. Essa alteração se faz necessária para que não haja rótulos com trechos de ambas as *strings*. Para isso, basta verificar se o caractere especial $ ocorre em algum rótulo da folha e, nesse caso, o índice de termino é atualizado para o índice de $ na *string* concatenada. A Figura 10 ilustra uma árvore de sufixos generalizada.

Figura 10. Árvore de Sufixos Generalizada para duas *strings*



Fonte: Wang, Jun. Disponível em: < http://www.slideshare.net/hustwj/suffix-trees >

Um caminho da raiz a um nó interno resulta em uma *substring* de S_1, de S_2 ou de ambos. O caminho da raiz até o nó mais profundo que possui *substring* de S_1 e S_2 corresponde a maior *substring* comum entre ambas as *strings*. Uma variação de um algoritmo de busca em profundidade na árvore que segue a teoria descrita pode ser codificado em poucas linhas. O problema pode ser estendido para encontrar a maior *substring* comum entre conjuntos maiores de *strings* apenas associando um caractere especial único a cada uma delas, concatenando-as e seguindo o mesmo conceito do algoritmo para duas *strings*.

8. CONCLUSÃO

Pôde-se perceber que os algoritmos para construção das estruturas de dados respondem de maneira diferente quando são utilizados textos periódicos, caso em que o autômato de sufixos demonstrou ser mais eficiente.

A partir do teste de Wilcoxon foi apontado que o tempo médio de execução gasto para a construção das estruturas pode ser considerado igual para a maior parte dos testes realizados.

Também foi perceptível que tanto o autômato de sufixos quanto a árvore de sufixos se sobressaem sobre o algoritmo KMP quando o *string matching* é estendido para encontrar a posição no texto em que múltiplos padrões ocorreram. O aumento do tamanho do alfabeto não demonstrou aumento considerável no tempo de execução das estruturas de sufixos para os casos de testes realizados, embora esse fator possa ter maior impacto com o aumento do tamanho do texto.

Ainda, o trabalho demonstrou como resolver as extensões abordadas nos testes e também como solucionar o problema de encontrar a maior *substring* comum entre duas *strings* sem o uso de bibliotecas de terceiros.

REFERÊNCIAS

ALTSCHUL, S. et al. Basic local alignment search tool. Journal of Molecular Biology, v.215, n.3, p.403-410, 1990. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0022283605803602> Acesso em: 14 Maio 2016.

ALTSCHUL, S. et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Oxford Journals, v.25, n.17, p.3389-3402, 1997. Disponível em: <https://www.ncbi.nlm.nih.gov/pubmed/9254694>. Acesso em: 2 Abr. 2016.

APOSTOLICO, A.; PREPARATA, F. P. Optimal off-line detection of repetitions in a string.

Theoretical Computer Science, v.22, n.3, p.297-315, 1983. Disponível em: <http://www.sciencedirect.com/science/article/pii/0304397583901093>. Acesso em: 2 Maio 2016.

APOSTOLICO A.; PREPARATA, F. P. Structural Properties of the Sring statistics problem. Journal of Computer and System Sciences, v.31, n.3, p.394-411, 1985. https://doi.org/10.1016/0022-0000(85)90060-1

APOSTOLICO, A. et al. 40 Years of Suffix Trees. Communications of the ACM, v.59, n.4, p.66-73, 2016. Disponível em: <http://dl.acm.org/citation.cfm?id=2810036>. Acesso em: 23 Jun. 2016.

BADKOBEH, G. et al. Efficient computation of maximal anti-exponent in palindrome-free strings. Theoretical Computer Science, 2016. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0304397516001365>. Acesso em: 25 Jun. 2016.

BLUMER, A. et al. Linear size finite automata for the set of all subwords of a word na outline of results. Bulletin of The European Association for Theoretical Computer Science, v.21, p.12-20, 1983. Disponível em: <http://65.54.113.26/Publication/816517/linear-size-finite-automata-for-the-set-of-all-subwords-of-a-word-an-outline-of-results>. Acesso em: 3 Abr. 2016.

BLUMER, A. et. al. The smallest automaton recognizing the subwords of a text. Theoretical Compuer Science, v.40, p.31-55, 1985. Disponível em: <https://pdfs.semanticscholar.org/d909/8f385860608e3b1e7dace4cd6a8e88d85c77.pdf> Acesso em: 2 Maio 2016.

BOYER, R. S.; MOORE, J. S. A fast string searching algorithm. Communications of the ACM, v.20, n.10, p.762-772, 1977. Disponível em: <http://www.cs.utexas.edu/users/moore/publications/fstrpos.pdf>. Acesso em: 4 Abr. 2016.

DALALIO, G. Estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a “string matching”. 2013. Disponível em: <http://www.bdita.bibl.ita.br/TGsDigitais/lista_resumo.php?num_tg=65317>. Acesso em: 12 Out. 2015.

DELCHER, A. L. et al. Alignment of whole genomes. Oxford Journals, Nucleic Acids Research, v.27, n.11, p.2369-2376, 1999. Disponível em: <http://nar.oxfordjournals.org/content/27/11/2369.short>. Acesso em: 5 Abr. 2016.

DEMSAR, J. Statistical Comparisons of Classifiers over Multiple Data Sets. Journal of Machine Learning Research, v.7, p.1-30, 2006. Disponível em: <http://jmlr.csail.mit.edu/papers/volume7/demsar06a/demsar06a.pdf>. Acesso em: 9 Abr. 2016.

EHRENFEUCHT, A.; HAUSSLER, D. A new distance metric on strings computable in linear time. Discrete Applied Mathematics, v. 20, n. 3, p.191-203, 1988. Disponível em: <http://www.sciencedirect.com/science/article/pii/0166218X88900765>. Acesso em: 7 Abr. 2016.

FARACH, M. Optimal suffix tree construction with large alphabets. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 38. Proceedings... Washington: IEEE Computer Society, p.137. 1997. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.4336>. Acesso em: 9 Abr. 2016.

FARIAS, C. N. Projeto de Primers via Árvore de Sufixos. Trabalho (Conclusão de Curso) – Universidade Federal de Mato Grosso do Sul. 2010. Disponível em:

<http://facom.sites.ufms.br/files/2015/11/Projeto-de-Primers.pdf>. Acesso em: 6 Set. 2016.

FARO, S.; LECROQ, T. A Fast Suffix Automata Based Algorithm for Exact Online String Matching. Implementation and Application of Automata, v.7381, p.149-158. Disponível em: <http://link.springer.com/chapter/10.1007%2F978-3-642-31606-7_13>. Acesso em: 6 Maio 2016.

FRASER, C. W.; MYERS, E. W.; WENDT, A. L. Analyzing and Compressing Assembly Code. In: ACM SIGPLAN '84 SYMPOSIUM ON COMPILER CONSTRUCTION SIGPLAN NOTICES. Proceedings... v.19, n.6, p.117-121, 1984. https://doi.org/10.1145/502874.502886

GIAQUINTA, E. Run-Length Encoded Nondeterministic KMP and Suffix Automata. Department of Computer Science and Engineering, Alto University, 2015. Disponível em: <https://arxiv.org/abs/1412.3688v2>. Acesso em: 1 Jul. 2016.

GIEGERICH, R.; KUTZ, S. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. Algorithmica, v.19, p.331-353, 1997. Disponível em: <http://www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf>. Acesso em: 23 Abr. 2016.

GUSFIELD, D. Book: Algorithms on Strings, Trees and Sequences: computer science and computational biology. 1997. Disponível em: <http://dl.acm.org/citation.cfm?id=262228>. Acesso em: 20 Abr. 2016.

INANOV, M. Suffix automata, construction in O(N) and applications. 2008. Disponível em: <http://e-maxx.ru/algo/suffix_automata>. Acesso em: 12 Out. 2015.

KARP, R.M.; RABIN, M.O. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, v.31, n.2, p.249-260, 1987.

KNUTH, D. E.; MORRIS, J. H. J.; PRATT, V. R. Fast Pattern Matching in strings. SIAM Journal on Computing, v.6, n.2, 1977. Disponível em: <https://pdfs.semanticscholar.org/4479/9559a1067e06b5a6bf052f8f10637707928f.pdf> Acesso em: 12 Maio 2016.

MANBER, U.; MYERS, G. Suffix Arrays: a new method for on-line string searches. SIAM Journal on Computing, v.22, n.5, p.935-948, 1990. Disponível em: <http://epubs.siam.org/doi/abs/10.1137/0222058> Acesso em: 12 Dez. 2015.

MCCREIGHT, E.M. A space-economical suffix tree construction algorithm. Journal of Association for Computing Machinery, v.23, n.2, p.262-272, 1976. Disponível em: <http://libeccio.di.unisa.it/TdP/suffix.pdf>. Acesso em: 27 Abr. 2016.

RODEH, M.; PRATT, V. R.; SHIMON, E. Linear Algorithm for Data Compression via String Matching. Journal of the ACM, v.28, n.1, p.16-24, 1981. Disponível em: <http://dl.acm.org/citation.cfm?id=322237>. Acesso em: 2 Maio 2016.

SACOMOTO, G. A. T. Árvores de Ukkonen: caracterização combinatória e aplicações. 2011. Dissertação (Mestrado) – Instituto de Matemática e Estatística, São Paulo, 2011. Disponível em: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-21042011-092209/pt-br.php>. Acesso em: 2 Abr. 2016.

SETUBAL, G. C.; BRAUNING, R. Book: Bioinformatics in Tropical Disease Research: A Practical and Case-Study Approach. 2006. Chapter 5, Similarity Search. Disponível em: <https://www.ncbi.nlm.nih.gov/books/NBK6831/> Acesso em: 11 Maio 2016.

SINGLA, N.; GARG, D. String Matching Algorithms and their applicability in various applications. International Journal of Soft Computing and Engineering, v.1, n.6, 2012. Disponível em: <http://gdeepak.com/pubs/String%20Matching%20Algorithms%20and%20their%20Applicability%20in%20various%20Applications.pdf>. Acesso em: 14 Maio 2016.

TRIOLA, M. F. Livro: Introdução a Estatística. 10. ed. 2008. Disponível para download em: <http://br.librosintinta.in/triola,-mario-f-introdu%C3%A7%C3%A3o-%C3%A0-estat%C3%ADstica-pdf.html>. Acesso em: 6 Maio 2016.

UKKONEN, E. On-line construction of suffix trees. Algorithmica, v.14, p.249, 1995. Disponível em: <http://link.springer.com/article/10.1007/BF01206331>. Acesso em: 20 Maio 2016.

WEINER, P. Linear pattern matching algorithms. In: SWAT '73, ANNUAL SYMPOSIUM ON SWITCHING AND AUTOMATA THEORY, 14. Proceedings… 1973, p. 1-11. Disponível em: <http://dl.acm.org/citation.cfm?id=1441766>. Acesso em: 2 Maio 2016.

WILCOXON, F. Individual Comparisons by Ranking Methods. Biometrics Bulletin, v.1, n.6, p.80-83, 1945. Disponível em: <http://sci2s.ugr.es/keel/pdf/algorithm/articulo/wilcoxon1945.pdf>. Acesso em: 3 Ago. 2016.