Teoria da Computação - Autômatos de Sufixos

Israel Efraim de Oliveira
israel.eo.cs@outlook.com
Universidade Federal de Santa Catarina
Navegantes, SC, Brasil

José Carlos Zancanaro
jose.zancanaro@hotmail.com
Universidade Federal de Santa Catarina
Brusque, SC, Brasil

ABSTRACT

Este relatório apresenta uma solução para o problema indicado no 'Trabalho I – Autômatos de Sufixos' em Teoria da Computação (*INE410113-41000025*). Com base no conceito de autômatos de sufixo, foi solucionado o problema Beecrowd 1284 por meio de uma *Trie* implementada na linguagem de programação C++17. Houve aceite do juiz online com tempo de execução de 0.876 segundos.

KEYWORDS

Teoria da computação, Autômatos de Sufixos, Trie.

1 INTRODUÇÃO

O autômato de sufixo de uma determinada palavra w é definido como o autômato determinístico mínimo que reconhece todo o conjunto finito de sufixos de w [4]. Sabendo-se que $w = xyz$ para as palavras $x, y, z \subset \Sigma*$, tem-se x como prefixo, y como subpalavra e z como sufixo [2].

Nesse sentido, o autômato de sufixo atua como uma estrutura de dados que permite resolver um amplo espaço de problemas relacionados ao processamento de textos [2]. Especialmente àqueles interessados em determinar se uma palavra p é uma subpalavra de w, isto é, se p ocorre em w.

A construção de um autômato para o reconhecimento das subpalavras de w serve como um índice auxiliar de busca que permite a localização de p em tempo proporcional à p e não à w [8, 9], sendo eficiente, por exemplo, para buscas em textos estáticos em bancos de dados textuais [11]. Outras aplicações, descritas em Gusfield [6], envolvem encontrar a maior sequência entre duas cadeias (eficiente em tempo e espaço), identificar sequências comuns em mais de dois textos, buscar subpalavras em bases previamente reconhecidas e tratadas, e realizar a pesquisa e o reconhecimento de trechos e contaminação genética em bases genômicas de DNA, cuja correspondência exata não se faz possível. Karpischek [7] complementa a utilidade de autômatos de sufixos na biologia molecular pelo cálculo eficiente de uma série de estatísticas sobre uma palavra, muitas das quais computáveis em tempo linear.

A Figura 1 demonstra um autômato simples (A) que aceita três palavras que finalizam em sufixos distintos (ac, acab e acba). O autômato determinístico (B) representa a estrutura de índice completa que aceita o conjunto de todos os sufixos de A [9]. Se todos os estados de B fossem de aceitação, o autômato aceitaria também todas as subpalavras de A. Nesse caso, a estrutura poderia ser vista como um grafo direcionado acíclico, chamado de Grafo Orientado Acíclico de Palavras (em inglês: *Directed Acyclic Word Graph - DAWG*) [2].

O algoritmo para construção de um autômato de sufixo em tempo linear foi desenvolvido inicialmente por Blumer et al. [2] e é disposto, de forma simples, em Algorithms for Competitive Programming [1]. No algoritmo, cada estado possuí uma propriedade $pai(\alpha)$, que identifica o ancestral de α, e uma propriedade $profundidade(\alpha)$, que representa a distância de α para o estado inicial (caso seguisse $pai(\alpha)$ recursivamente, por exemplo). O estado inicial possuí $profundidade = 0$ e $pai = indefinido$. Para cada produção b, adiciona-se um novo estado com $profundidade(\alpha) = profundidade(pai(\alpha)) + 1$. Procura-se saber se antecessores de α já possuem transição em b, adicionando-se quando negativo. Caso positivo num estado p com produção b para q e $profundidade(p)+1 \neq profundidade(q)$, redefine-se as transições destinadas a q para um novo estado, cópia de q, com $profundidade(p) + 1$. Nas condições em que antecessores de α não possuem transição com produção b, somente atualiza-se seu ancestral para o estado inicial. Da mesma forma, se q for predecessor de p, define-se $pai(\alpha) = q$.

2 PROBLEMA

Nesse trabalho, é requerido atingir dois objetivos: (1) a confecção deste relatório, contextualizando o problema, método e solução; e (2) a resolução de um desafio apresentado na plataforma online Beecrowd, utilizando na solução o conceito de autômatos de sufixo.

O problema indicado para ser resolvido neste trabalho é o desafio de número 1284, disponível na plataforma Beecrowd[1]. O desafio, intitulado de 'Digitando no Telefone Celular' insere-se no contexto de *autocompletar* palavras disponíveis dentro de um dicionário. O objetivo do desafio é determinar a quantidade média de letras que o usuário precisa informar para que o sistema complete automaticamente a palavra.

Supondo que em um dicionário composto por três palavras, 'ride', 'riders' e 'ridiculous', o usuário queira digitar a palavra 'ridiculous'. A primeira letra é sempre necessária, e então o usuário digita 'r', que em seguida é autocompletada com as letras 'i' e 'd', pois essas duas letras se repetem para as outras palavras expostas no dicionário. Adiante, é necessário que o usuário digite uma outra letra, visto que há 'd' e 'i' como alternativas disponíveis. Para atingir a palavra 'ridiculous', o usuário então tecla a letra 'i'. Como não existe outra ramificação no dicionário a partir deste ponto, o sistema determina que o restante (sufixo) da palavra é 'culous'. Dessa forma, a média de teclas necessárias para completar as palavras ride, riders e ridiculous é, respectivamente: $2 + 3 + 2 = 7/3 = 2.33$.

O critério de aceitação do desafio é definido por inúmeros cenários de testes diversificados quanto ao tamanho e às palavras do dicionário. A problematização define como intervalo do tamanho do dicionário $|X| = 1 \leq N \leq 10^5$. As palavras variam até 80 caracteres $[a - z]$ e a soma do tamanho de todas as palavras de um dicionário varia até 10^6. A entrada do programa é fornecida por meio do *stdin* e para cada dicionário se deve imprimir uma linha no *stdout* com a média de letras necessárias para determinar exclusivamente cada palavra [3], com arredondamento de duas casas decimais. Ainda,

[1] disponível em beecrowd.com.br/judge/pt/problems/view/1284

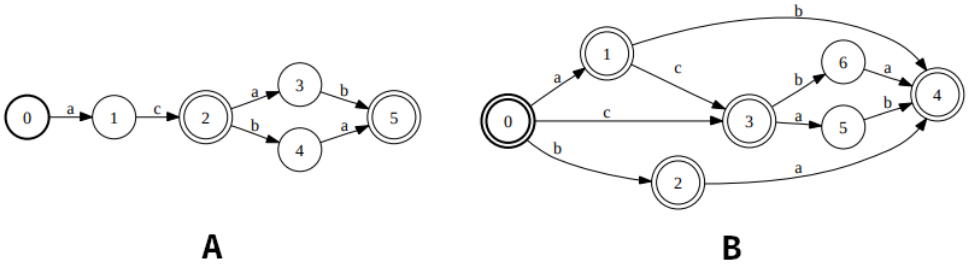

Figura 1: Autômato de Sufixo (B) para o autômato (A) que aceita ac, acab, acba [9]

a submissão na plataforma online aceita somente soluções com tempo de execução menor que três segundos.

3 SOLUÇÃO

A solução desenvolvida nesse trabalho utiliza amplamente do conceito de Autômatos de Sufixo apresentada na Seção 1. A versão final do algoritmo, desenvolvido na linguagem de programação C++17 e **aceita** pela plataforma Beecrowd, é uma *Trie* (árvore digital ou árvore de prefixo - *trie* vem de *retrieval* [5]). Embora a presente solução tem como base uma estratégia de busca de prefixo, a tentativa inicial consistia na implementação de um autômato de sufixo para cada dicionário, que representava todas as suas palavras.

A construção do autômato de sufixo para as palavras de um dicionário se dá pelo algoritmo descrito em [1], com poucas modificações. Dado que a inserção deve tratar de mais de uma palavra, a extensão do autômato começa pelo primeiro estado após o maior prefixo entre o autômato e a palavra a ser inserida. Dessa forma, ao inserir 'hell', a palavra 'hello' começa não pelo estado inicial, mas no estado ao fim da palavra 'hell'.

Ao passo que a solução aparenta gerar um autômato de sufixo correto (não apresentamos a prova de corretude, ver [10]), não é trivial atender ao objetivo de mensurar a quantidade de letras necessárias para determinar a exclusividade de uma palavra. Determina-se o supracitado, pois, a inserção dos sufixos na árvore dificulta a decisão de quais estados representam ramificações de sufixo para as palavras do dicionário. Considere as palavras 'ride' e 'structure' (presentes nos exemplos de entrada do desafio): ao digitar 'r', deveria ser possível determinar que ride é a única palavra disponível, entretanto, a inclusão dos sufixos faz com que a próxima letra possa ser 'u' de 'ructure' ou 'e' de 're', ambos sufixos de 'structure'.

Consequentemente, optou-se por não inserir os sufixos da palavra, somente as possíveis variações de prefixo, tendo como resultado efetivo uma árvore de prefixos. Desse modo, pode-se facilmente destacar quais são os estados que possuem ramificações: aqueles que são de aceitação e que possuem predecessores e aqueles que possuem mais de um predecessor imediato.

Portanto, a implementação da solução é composta pela estrutura de Nó (*Node*), pelo algoritmo de inserção (ilustrado na Figura 2) e pelo procedimento da quantificação das ramificações presentes no dicionário (Figura 3). Para cada conjunto de palavras, o programa instancia uma trie (*Node*), lê cada entrada do *stdin* e anexa a palavra na trie. Ao fim da última palavra, invoca-se a função *count* com a trie para obter a resposta para o conjunto em questão.

O *Node* é uma estrutura alocada em *heap* que armazena um vetor e a quantidade de outros nós (*children, count*), bem como a contagem de ocorrências (*visit*) e um indicador de aceitação (*is_final*). Todos os estados, inclusive o inicial, são do tipo *Node*. Nesse sentido, a estrutura da trie é o nó inicial.

```cpp
/* struct Node */
template <typename Str>
auto append(Str && word) -> void {
  auto state = 0ul;
  auto node = this;

  for (auto i = 0ul; i < std::size(word); ++i) {
    auto nth = word[i] - 'a';

    if (node->children[nth] == nullptr) {
      node->children[nth] = std::make_unique<Node>();
      node->count++;
    }

    node = node->children[nth].get();
    node->visit++;
  }

  node->is_final = true;
}
```

Figura 2: Inserção de uma palavra na trie

Na Figura 2, verifica-se o procedimento responsável por inserir uma nova palavra à trie. Essa operação busca o prefixo da palavra na trie e, as letras não encontradas resultam na criação de novos estados e na associação de uma transição com a determinada produção a partir do nó de referência. Simultaneamente, o procedimento registra as métricas de *count* e *visit*, com objetivo de facilitar o cômputo do valor solicitado no problema.

```cpp
auto count(Node* const node) -> u64 {
  auto value = 0ull;

  for (auto i = 0ul; i < std::size(node->children); ++i) {
    if (node->children[i]) {
      value += count(node->children[i].get());
    }
  }

  value += (node->count > 1 || node->is_final)
           * node->visit - node->is_final;

  return value;
}
```

Figura 3: Quantificação das ramificações presentes na trie

Uma vez gerada a trie por meio de sucessivas chamadas de *Node::append* para o conjunto de palavras do dicionário, pode-se conhecer a quantidade de visitas totais (teclas digitadas) por meio do procedimento *count* (Figura 3), que recebe o estado inicial da trie. A função caminha recursivamente na árvore somando as visitas quando o estado é final ou possui ramificação. A partir do valor de retorno v para a chamada de *count* com o nó inicial, a média de teclas necessárias para determinar uma palavra é dada por $(n+v)/n$, sendo n a quantidade de palavras no dicionário. Ressalta-se que a adição de n ao dividendo equivale à premissa de que o usuário sempre precisa digitar a primeira letra.

4 CONSIDERAÇÕES FINAIS

O algoritmo criado pelos autores[2] para a resolução do desafio 'Digitando no Telefone Celular' foi escrito na linguagem de programação C++17. Conforme exibido na Figura 4, a submissão da solução na plataforma foi totalmente aceitada pelo juiz online, apresentando um tempo de execução de 0.876 segundos e sendo um arquivo de tamanho 1.94 KB.

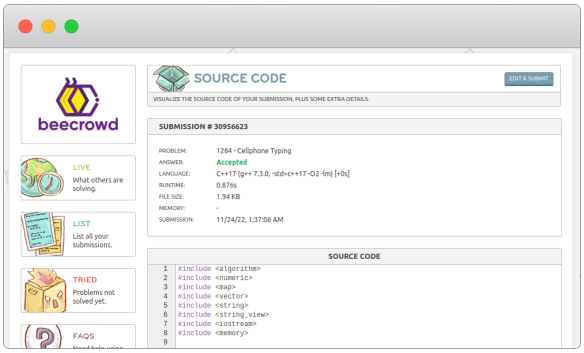

Figura 4: Resultado obtido pela submissão da solução.

A solução consistiu na implementação de uma trie para permitir o cálculo eficiente da situação problematizada. Embora a estrutura adotada para determinar a média de letras necessárias no *autocomplete* corresponda ao caráter de reconhecimento de prefixos, o automato de sufixos permitiria um suporte melhor de *autocomplete* e sugestão de palavras, visto que é possível determinar a ocorrência de uma subpalavra independentemente de sua posição, que nem sempre é conhecida, de modo a oferecer mais usabilidade.

Quanto à restrição de performance, a implementação sofreu dois principais ajustes. O primeiro, justificado previamente por outro motivo, foi o abandono da inserção dos sufixos das palavras, pois o intervalo do tamanho do conjunto de palavras foi suficientemente grande para que o limite de tempo fosse atingido. O segundo foi a antecipação do cálculo das métricas para a rotina de confecção do autômato, visto que anteriormente a entrada era copiada para um vetor de palavras para que, após a construção do autômato, fosse utilizada na verificação das ramificações. A utilização da trie singularmente sem esse mecanismo de metrificação simultânea não havia resolvido o problema de performance (tempo máximo excedido). Outra alteração, de menor impacto, foi a utilização de vetores de tamanho fixo ao invés de *maps* para armazenar as transições de cada estado.

[2]em https://github.com/IsraelEOliveira/code-problems/tree/main/beecrowd1284

REFERÊNCIAS

[1] Algorithms for Competitive Programming. 2022. Suffix automaton. https://cp-algorithms.com/string/suffix-automaton.html

[2] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. 1985. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science* 40 (1985), 31–55. https://doi.org/10.1016/0304-3975(85)90157-4

[3] cppreference. 2022. stdin, stdout, stderr. https://en.cppreference.com/w/cpp/io/c/std_streams

[4] Maxime Crochemore and Christophe Hancart. 1997. Automata for Matching Patterns. In *Handbook of Formal Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 399–462. https://doi.org/10.1007/978-3-662-07675-0_9

[5] Jairo Francisco de Souza. 2009. *Busca Digital (Trie e Árvore Patrícia)*. Universidade Federal de Juiz de Fora. https://www.ufjf.br/jairo_souza/files/2009/12/6-Strings-Pesquisa-Digital.pdf

[6] Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, England. https://doi.org/10.1017/cbo9780511574931

[7] Ricardo Ueda Karpischek. 1993. *O Autômato dos Sufixos*. Master's thesis. Universidade de São Paulo, São Paulo. https://doi.org/10.11606/d.45.1993.tde-20210728-235623

[8] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (June 1977), 323–350. https://doi.org/10.1137/0206024

[9] Mehryar Mohri, Pedro Moreno, and Eugene Weinstein. 2009. General suffix automaton construction algorithm and space bounds. *Theoretical Computer Science* 410, 37 (Sept. 2009), 3553–3562. https://doi.org/10.1016/j.tcs.2009.03.034

[10] Israel Efraim Oliveira and José Carlos Zancanaro. 2022. Autômatos de sufixo: Exemplos de aceitação. https://github.com/IsraelEOliveira/code-problems/tree/main/beecrowd1284/automata#readme

[11] Marinella Sciortino and Luca Q. Zamboni. 2007. Suffix Automata and Standard Sturmian Words. In *Developments in Language Theory*. Springer Berlin Heidelberg, Berlin, Heidelberg, 382–398. https://doi.org/10.1007/978-3-540-73208-2_36