A INTERPRETAÇÃO DE AUTÔMATOS PARA O ALGORITMO KMP

Arthur Botelho, UnB
Leonardo Riether, UnB

Resumo

As ideias aqui apresentadas são fruto de uma colaboração e já foram publicadas no Codeforces [1] e anteriormente no blog pessoal de Leonardo Riether [4]. O objetivo deste artigo é trazer o que acreditamos ser uma das formas mais naturais e intuitivas de se compreender o algoritmo Knuth-Morris-Pratt (KMP), utilizando o conceito de Autômatos Finitos, para a comunidade de programação competitiva brasileira.

1. Fundamentos e o Problema

Assumiremos que o leitor possui compreensão básica sobre a terminologia de grafos e cadeias (*strings*). Para entender o algoritmo, é necessário compreender conceitos fundamentais sobre Autômatos Finitos Não-Determinísticos (NFA) e Determinísticos (DFA).

De maneira simplificada, um autômato pode ser visto como um grafo direcionado onde:

1. Os vértices são chamados de **estados**. Há um estado inicial e um ou mais estados de *aceitação*.
2. As arestas representam **transições** e são rotuladas com caracteres de um alfabeto.

A simulação da leitura de uma cadeia S por um autômato pode ser visualizada através de *threads*, que trafegam de um estado a outro quando uma aresta compatível com o caractere atual de S é encontrada. Em um NFA, um estado pode ter múltiplas arestas saindo com o mesmo caractere (nesse caso, a *thread* se divide e segue vários caminhos simultaneamente), enquanto em um DFA, cada caractere leva a exatamente um estado.

Neste artigo, iremos mostrar o KMP como uma forma de resolver o problema clássico de Casamento de Cadeias: dadas duas cadeias S (texto) e P (padrão), encontre todas as subcadeias de S que correspondam exatamente a P.

2. Modelagem com Autômatos Não-Determinísticos

Existe uma construção simples de um NFA capaz de resolver esse problema. Para identificar ocorrências do padrão P, criamos um autômato de $|P|+1$ estados, organizados de forma sequencial e numerados de 0 a $|P|$. Cada estado i ($i < |P|$) possui uma aresta para o estado $i+1$ consumindo o caractere P_i. O estado 0 (inicial) possui uma transição (em laço) para si mesmo, consumindo qualquer caractere do alfabeto. O último estado ($|P|$) é o estado de aceitação.

Podemos visualizar, na Figura 1, como seria esse NFA para o padrão "abaa". O laço com o símbolo Σ no estado 0 indica o consumo de qualquer caractere do alfabeto.

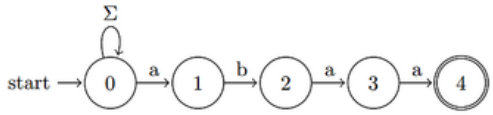

FIGURA 1: EXEMPLO DE UM NFA

Quando um texto é fornecido a este autômato, o laço no estado 0 gera uma nova thread a cada caractere lido, de forma que sempre pode-se iniciar um novo casamento. Ao mesmo tempo, threads geradas em passos anteriores avançam para o próximo estado se os caracteres corresponderem — caso contrário, a respectiva thread "morre". Se uma thread alcançar o estado de aceitação, um casamento foi encontrado. Note que isso já permite inclusive encontrar casamentos sobrepostos.

3. Exemplo Passo a Passo

Para facilitar a compreensão do comportamento simultâneo das threads, vamos simular a leitura do texto S = "abaa" pelo NFA do padrão P = "abaa". A evolução dos estados pode ser vista na Figura 2, onde os nós coloridos de vermelho representam as threads ativas.

Iniciamos com uma única thread no estado 0.

1. Lendo 'a': A thread no estado 0 se divide. Uma continua no estado 0 (devido ao laço Σ) e a outra avança para o estado 1. Temos threads em {0, 1}.

2. Lendo 'b': A thread no estado 1 avança para o estado 2. A thread no estado 0 permanece no 0. Temos threads em {0, 2}.

3. Lendo 'a': A thread no 2 avança para o 3. A thread no 0 se divide novamente, gerando uma no 1. Temos threads em {0, 1, 3}.

4. Lendo 'a': A thread no 3 avança para o 4, que é o estado de aceitação (portanto, encontramos um casamento). A thread que estava no 1 "morre" por não haver aresta com 'a', e o estado 0 gera uma nova no 1. Temos threads em {0, 1, 4}.

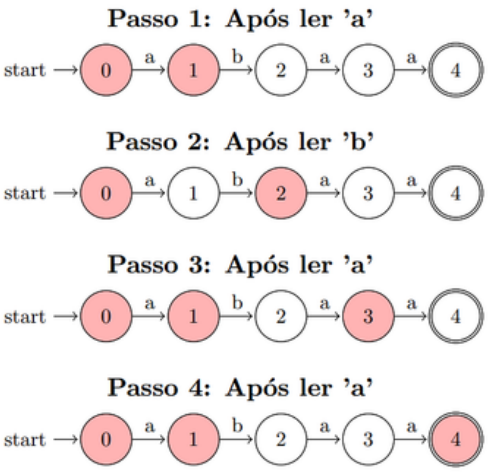

FIGURA 2: SIMULAÇÃO DO PADRÃO "abaa". OMITIMOS A ARESTA DE LAÇO NO ESTADO INICIAL PARA MAIOR CLAREZA.

Recomendamos o simulador de KMP de Leonardo Riether [5] para visualizar, de forma interativa, o processo de casamento com o autômato.

4. Custo da Simulação

A simulação ingênua deste NFA custaria tempo $\mathcal{O}(|S| \times |P|)$, já que múltiplas threads poderiam estar ativas e precisariam ser processadas a cada caractere do texto. No entanto, é possível otimizar esse processo observando a construção deste NFA.

5. Configurações Alcançáveis e a Thread Líder

Apesar da simulação não-determinística sugerir um número exponencial de configurações de threads simultaneamente ativas, pode-se provar que apenas $|P|+1$ configurações distintas são alcançáveis.

Denotaremos a thread mais avançada no autômato por "líder". Se o líder está no estado i, as posições de todas as outras threads (que estão em estados anteriores a i) são unicamente determinadas. Observando o Passo 3 do exemplo anterior, a thread líder está no estado 3. Note que as threads ativas à esquerda (nos estados 0 e 1) são uma consequência direta e previsível do caminho que o líder tomou.

Isso ocorre porque o caminho para alcançar o estado i exige que os últimos i caracteres lidos correspondam exatamente ao prefixo de tamanho i do padrão P. Logo, se mantivermos a posição da thread líder, sabemos que é possível inferir toda a informação necessária para o processamento do autômato, e que ocorrerá um casamento apenas quando o líder chegar ao estado $|P|$.

6. O Vizinho (Falha)

O desafio para simular o processamento do autômato surge quando a thread líder falha (o caractere lido não corresponde à próxima aresta e a thread morre). Nesse caso, precisamos saber quem assumiria a liderança.

A primeira candidata para novo líder será, naturalmente, a thread ativa mais avançada que não for o líder original. Chamaremos essa thread de "vizinho" — note que sempre há um vizinho desde que o líder esteja além do estado 0, pois este sempre possui uma thread ativa.

Assim que a thread líder falhar, verificamos se o seu vizinho consegue avançar com o caractere atual. Se sim, ele se torna o novo líder em sua respectiva próxima posição. Se também falhar, verificamos o vizinho do vizinho, e assim sucessivamente, até que ocorra um casamento ou a liderança volte ao estado 0.

Mesmo que precisemos retroceder a múltiplos vizinhos durante uma única falha, a soma total das quantidades de retrocessos durante o processamento de S no autômato que geramos baseado em P é $\mathcal{O}(|S|)$. O argumento é amortizado: no máximo uma thread é "criada" por caractere lido, logo, o número máximo de falhas (mortes de threads) está limitado ao tamanho de S.

7. Pré-cálculo e Implementação

Para simular o NFA em tempo $\mathcal{O}(|S|)$, precisamos pré-calcular um vetor vizinho, onde $vizinho[i]$ armazena o estado da thread vizinha caso o líder atual esteja no estado i.

O cálculo do vetor de vizinhos pode ser feito de forma construtiva iterando sobre o próprio padrão P. Trata-se essencialmente de fornecer P ao próprio NFA e verificar quem assumiria a liderança a cada passo.

Uma implementação do algoritmo descrito é apresentada no Código 1.

```cpp
struct KMP {
    string P;
    int n;  // n = |P|
    vector<int> vizinho;

    KMP(string& p) {
        P = p;
        n = (int)P.size();

        vizinho.resize(n + 1);
        vizinho[1] = 0;  // estado 0 sempre tem thread ativa

        for (int i = 1; i < n; i++)
            vizinho[i + 1] = proximo_lider(vizinho[i], P[i]);
    }

    bool avanca(int estado, char c) {
        return estado < n && P[estado] == c;
    }

    int proximo_lider(int lider, char entrada) {
        if (lider == 0)
            return P[0] == entrada;

        if (avanca(lider, entrada))
            return lider + 1;

        else
            return proximo_lider(vizinho[lider], entrada);
    }
};
```

CÓDIGO 1: IMPLEMENTAÇÃO DO ALGORITMO KMP ATRAVÉS DA ABORDAGEM DE AUTÔMATOS.

Esta estrutura pode ser usada para resolver o problema de Casamento de Cadeias em $\mathcal{O}(|S|+|P|)$. No Código 2, recebemos cadeias S e P e retornamos as posições (indexadas em 1) onde uma ocorrência de P em S termina.

```cpp
vector<int> ocorrencias(string P, string S) {
    KMP kmp(P);
    vector<int> res;
    int estado = 0;  // lider atual

    for (int i = 0; i < (int)S.size(); i++) {
        char c = S[i];
        estado = kmp.proximo_lider(estado, c);

        if (estado == (int)P.size())
            res.push_back(i + 1);
    }

    return res;
}
```

CÓDIGO 2: RESOLUÇÃO DO PROBLEMA DE CASAMENTO DE CADEIAS.

Para entender a complexidade de tempo deste algoritmo, podemos utilizar novamente análise amortizada. A cada caractere lido, a posição do líder é incrementada em no máximo uma unidade. Logo, ao processar uma cadeia de

tamanho N, ocorrem no máximo N incrementos. Por outro lado, cada chamada recursiva em proximo_lider (quando ocorre uma falha) substitui o líder atual pelo seu vizinho. Por definição, um vizinho de um estado é anterior a esse estado, então vizinho[lider] < lider. Assim, cada falha força a posição do líder a diminuir de valor. Como a posição não pode ser negativa (é limitada pelo estado inicial 0), o número total de diminuições em toda a execução não pode exceder o número total de incrementos. Portanto, o número total de transições ao longo do algoritmo é limitado por $2|S|$ na fase de busca e $2|P|$ no pré-cálculo, comprovando a complexidade de tempo $\mathcal{O}(|S| + |P|)$.

8. Relação com a Função de Prefixo

A interpretação mais comum para o algoritmo KMP em materiais de programação competitiva é através da Função de Prefixo (Prefix Function ou vetor π). Para cada prefixo p da cadeia, ela armazena o tamanho do maior prefixo próprio de p que também é sufixo de p. Isso é equivalente ao vetor $vizinho$ estudado nas últimas duas seções. Quando o líder está no estado i, a posição da segunda thread mais avançada é precisamente este maior prefixo-sufixo $\pi[i]$.

9. O Autômato Determinístico

O algoritmo mostrado até agora para o problema de Casamento de Cadeias tem complexidade $\mathcal{O}(|S|+|P|)$. No entanto, isso pode não ser o bastante para resolver alguns tipos de problemas.

A limitação da abordagem anterior surge do fato de que a complexidade de tempo depende de um argumento amortizado. Se estivermos processando um texto de forma linear, a quantidade de recuos nas *threads* é compensada pelos avanços. Porém, o argumento amortizado quebra quando precisamos ramificar o processamento. No contexto de programação competitiva, dois cenários comuns onde isso ocorre são:

1. Programação Dinâmica (PD) em cadeias: quando precisamos testar a adição de múltiplos caracteres diferentes ao final de uma cadeia, ramificando para próximos estados da PD.
2. Busca em Grafos onde uma cadeia é mantida: quando precisamos, para cada aresta do vértice atual da busca, ramificar para os vértices vizinhos com o caractere correspondente à aresta adicionado à cadeia mantida.

Nesses casos, calcular o próximo líder com o método amortizado pode resultar em uma complexidade ineficiente (causando Limite de Tempo Excedido). Para contornar isso, podemos pré-calcular, para todo líder e caractere do alfabeto, qual será o próximo líder, permitindo consultar as transições em $\mathcal{O}(1)$.

Isso nada mais é do que construir explicitamente o Autômato Determinístico (DFA) equivalente ao KMP. Em suma, iremos armazenar as transições diretas para todos os caracteres do alfabeto em qualquer estado. O Código 3 mostra uma extensão do Código 1 para gerar a matriz de transições, considerando caracteres de 'a' a 'z'.

```cpp
void constroi_dfa() {
    dfa = vector(n + 1, vector<int>(26));
    dfa[0][P[0] - 'a'] = 1;

    for (int k = 1; k <= n; k++) {
        for (int c = 0; c < 26; c++) {
            if (k < n && P[k] == 'a' + c)
                dfa[k][c] = k + 1;
            else
                dfa[k][c] = dfa[vizinho[k]][c];
        }
    }
}
```

CÓDIGO 3: CONSTRUÇÃO EXPLÍCITA DO AUTÔMATO DFA DO KMP.

Um excelente exemplo prático e recente da necessidade dessa estrutura ocorreu no problema B ("Busca Periódica") da Fase Zero da Maratona SBC de Programação de 2025 [3, 2]. O problema envolvia encontrar a periodicidade mínima de cadeias formadas por caminhos a partir da raiz em uma árvore [3].

Embora o problema em uma cadeia linear pudesse ser resolvido de forma eficiente apenas com o vetor $vizinho$, adaptar a abordagem clássica amortizada para uma DFS na árvore excede o tempo limite [2]. A solução oficial contorna essa ineficiência construindo exatamente o Autômato Determinístico do KMP [2]. Ao pré-calcular a matriz de transições para o alfabeto, cada passo da DFS avança o estado em $\mathcal{O}(1)$, garantindo que a solução execute no tempo esperado com complexidade $\mathcal{O}(26 \cdot N)$ [2].

10. Conclusão

Enxergar o algoritmo KMP não apenas como uma manipulação engenhosa de ponteiros de cadeias, mas como a simulação otimizada de um Autômato Finito, desmistifica profundamente o seu funcionamento e provas de corretude e complexidade. Além disso, essa visão também facilita o pensamento necessário para problemas avançados de PD e Grafos relacionados a cadeias onde o DFA pode ser aplicado.

Referências

[1] Arthur Botelho. The automaton interpretation for the kmp algorithm. Disponível em: https://codeforces.com/blog/entry/146191. Acesso em: 07 jul. 2026.
[2] Maratona SBC de Programação. Editorial da fase zero da maratona sbc de programação 2025. Disponível em: https://codeforces.com/gym/105925/attachments/download/31746/editorial-fase-zero-2025.pdf, 2025. Acesso em: 30 jul. 2026.
[3] Maratona SBC de Programação. Fase zero da maratona sbc de programação 2025 caderno de problemas. Disponível em: https://codeforces.com/gym/105925/attachments/download/31715/prova.br.pdf, 2025. Acesso em: 30 jul. 2026.
[4] Leonardo Riether. Kmp the automaton interpretation. Disponível em: https://leoriether.github.io/posts/kmp/. Acesso em: 07 jul. 2026.
[5] Leonardo Riether. Kmp simulator. Disponível em: https://leoriether.github.io/kmp-simulator/. Acesso em: 07 jul. 2026.