Universidade do Estado de Santa Catarina
Centro de Ciências Tecnológicas
Bacharelado em Ciência da Computação
Disciplina: Linguagens Formais e Autômatos
Professor: Ricardo Ferreira Martins
Alunos: Camile Neves e Rian Carlos Valcanaia
8 de julho de 2025

Implementação de um Autômato com Pilha para Resolução do Problema Torre de Hanoi

1 Introdução

O problema da Torre de Hanoi é um quebra-cabeça matemático clássico que serve como um excelente exemplo para o ensino de conceitos de recursão em ciência da computação. Tradicionalmente resolvido através de um algoritmo recursivo direto, este trabalho propõe uma abordagem alternativa e mais formal, utilizando os conceitos de Teoria das Linguagens Formais e Autômatos. O objetivo principal é demonstrar como um problema algorítmico pode ser modelado e resolvido por um Autômato com Pilha (PDA - Pushdown Automaton), tratando-o não como um reconhecedor de linguagens, mas como um mecanismo de evolução que gera a sequência de passos para a solução.

Este projeto consiste na implementação de um simulador em Python que resolve o quebra-cabeça da Torre de Hanoi para um número N de discos. A lógica central do simulador não será um algoritmo recursivo convencional, mas sim a emulação passo a passo de um autômato com pilha determinístico, conforme inspirado pelo modelo teórico proposto por Krasimir Yordzhev. O autômato utiliza sua pilha como uma "lista de tarefas" ou um "plano de ação", que é dinamicamente expandido e consumido para gerar a sequência ótima de movimentos.

A implementação completa, incluindo o código-fonte e instruções de uso, está disponível no repositório público do GitHub: `https://github.com/RianValcanaia/Trabalho_Final_LFA`.

Através desta simulação, busca-se ilustrar equivalência entre a descrição declarativa de uma solução (representada por uma gramática livre de contexto, que fundamenta o autômato) e sua execução procedural (a operação do autômato com pilha), fornecendo uma visão prática sobre a aplicação de modelos computacionais teóricos na resolução de problemas concretos.

2 Modelagem do problema

A solução para o problema da Torre de Hanoi pode ser descrita utilizando formalismos da teoria das linguagens. Nesta seção, detalhamos a Gramática Livre de Contexto (GLC) que gera a sequência de movimentos ótima e, na sequência, o Autômato com Pilha que "executa"essa gramática. Mas primeiramente vamos entender a dedução que será utilizada para a GLC.

2.1 Entendendo a resolução da Torre de Hanoi

Antes de introduzirmos formalmente a Gramática Livre de Contexto (GLC) e a construção do Autômato com Pilha (PDA) para modelar a Torre de Hanoi, é importante revisitar a abordagem tradicional que historicamente resolve esse problema: o algoritmo recursivo.

A versão recursiva é amplamente conhecida e representa uma solução eficiente e elegante, baseada na divisão sistemática do problema em subproblemas menores. Sua lógica está fundamentada em três etapas principais, aplicadas de forma repetitiva para N discos:

- **Passo 1:** mover os $N-1$ discos superiores do pino de origem para o pino auxiliar;

- **Passo 2:** transferir o disco N (o maior) diretamente do pino de origem para o pino de destino;
- **Passo 3:** mover os $N - 1$ discos do pino auxiliar para o pino de destino, sobrepondo-os ao disco maior.

Essa ideia pode ser traduzida diretamente em código, como mostra o exemplo abaixo:

Listing 1: Algoritmo recursivo tradicional para resolução da Torre de Hanoi

```
def hanoi(n, origem, destino, auxiliar):
    if n == 1:
        print(f"\nMova o disco 1 do pino {origem} para o pino {destino}")
    else:
        hanoi(n - 1, origem, auxiliar, destino)
        print(f"\nMova o disco {n} do pino {origem} para o pino {destino}")
        hanoi(n - 1, auxiliar, destino, origem)

def main():
    num = int(input("Digite o numero de discos:\n"))
    hanoi(num, 'A', 'C', 'B')

if __name__ == "__main__":
    main()
```

Analisando a lógica: A função `hanoi()` é definida com quatro parâmetros:

- `n`: número de discos;
- `origem`: pino de onde os discos devem ser movidos;
- `destino`: pino de destino final;
- `auxiliar`: pino intermediário usado para ajudar na movimentação.

Quando $n = 1$, temos o caso base da recursão, que representa a movimentação de um único disco, operação simples e direta.

Listing 2: Caso base da recursão

```
if n == 1:
    print(f"\nMova o disco 1 do pino {origem} para o pino {destino}")
```

Esse comando imprime na tela a instrução correspondente ao movimento do único disco presente, como ilustrado nas figuras 1 e 2.

Figura 1: Com apenas 1 disco

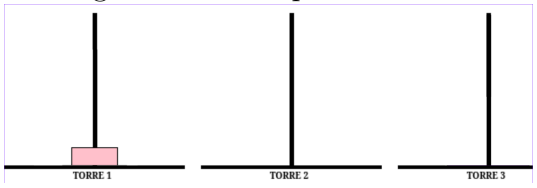
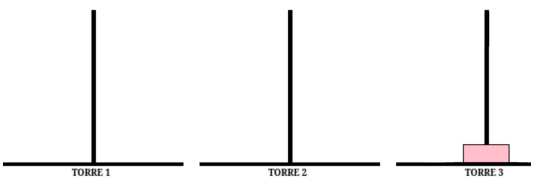


Figura 2: Torre finalizada



Para $n > 1$, o algoritmo realiza chamadas recursivas, quebrando o problema em partes menores:

Listing 3: Parte recursiva do algoritmo

```
else:
    hanoi(n - 1, origem, auxiliar, destino)
    print(f"\nMova o disco {n} do pino {origem} para o pino {destino}")
    hanoi(n - 1, auxiliar, destino, origem)
```

- `Primeiro:` move os $n-1$ discos superiores para o pino auxiliar;
- `Depois:` move o disco n diretamente para o destino;
- `Por fim:` move os $n-1$ discos do pino auxiliar para o destino, agora usando o pino de origem como apoio.

Figura 3: Primeiro movimento

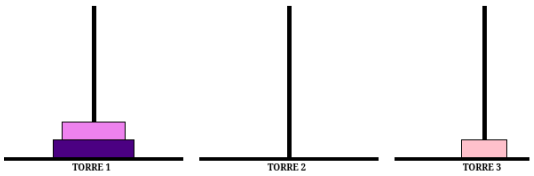


Figura 4: Segundo movimento

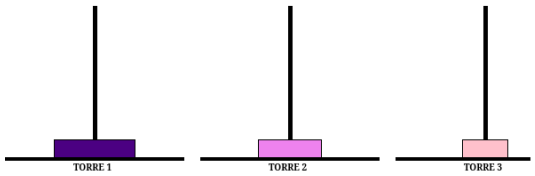


Figura 5: Terceiro movimento

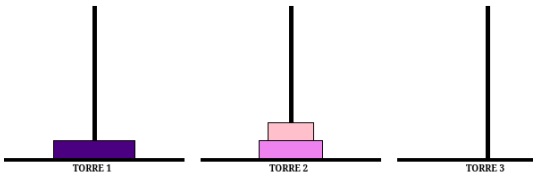


Figura 6: Quarto movimento

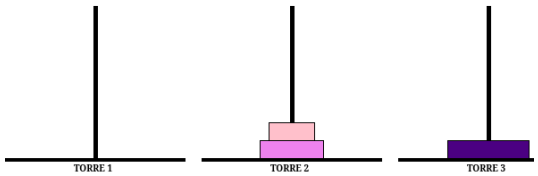


Figura 7: Quinto movimento

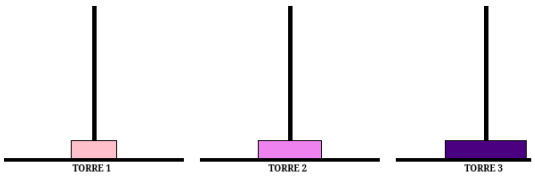
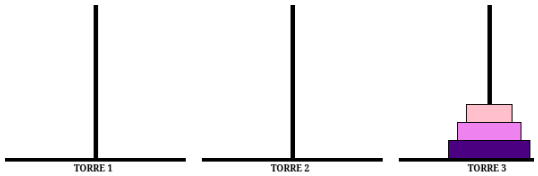


Figura 8: Sexto e sétimo movimento



Esse processo recursivo resulta em uma estrutura em forma de árvore, na qual cada chamada divide o problema em subproblemas menores até atingir o caso base ($n = 1$). Cada nó dessa árvore representa uma operação (ou grupo de operações) que depende da conclusão de etapas anteriores — refletindo um comportamento sequencial e hierárquico típico de algoritmos recursivos.

Um aspecto fundamental dessa abordagem é sua *repetitividade estruturada*. À medida que o número de discos aumenta, a lógica do algoritmo permanece a mesma: mover $n-1$ discos para uma torre auxiliar, mover o disco maior, e depois reposicionar os $n-1$ discos restantes. A recursão repete esse padrão indefinidamente, até alcançar a menor unidade do problema — o movimento de um único disco. Assim, mesmo com parâmetros diferentes, o algoritmo segue uma estrutura cíclica previsível.

Ao observar esse comportamento, nota-se que o fluxo do algoritmo pode ser interpretado como uma sequência de instruções armazenadas em uma pilha. Isso permite substituir as chamadas recursivas por uma **manipulação explícita** da pilha (ou seja, um controle direto sobre a estrutura) onde o programador empilha e desempilha instruções manualmente.

Essa analogia é o ponto de partida para transformar o algoritmo recursivo em um autômato com pilha determinístico. Nesse modelo, o problema é resolvido sem recursão, mas com base em uma gramática formal e uma pilha de controle que simula o mesmo comportamento por meio de regras sintáticas e operações sobre a memória auxiliar.

2.2 GLC para a Resolução do Problema

Para gerar a sequência de passos que resolve a Torre de Hanoi, utilizamos uma Gramática Livre de Contexto, conforme proposto por Yordzhev. A genialidade deste modelo está em mapear a natureza recursiva do problema diretamente para as regras de produção da gramática. Em vez de gerar uma linguagem com múltiplas palavras, esta gramática é projetada para gerar uma única palavra: a sequência exata de movimentos para mover N discos da torre de origem para a de destino.

A gramática G_N para um problema com N discos pode ser formalmente definida pela 4-tupla:

$$G = (V, T, P, S)$$

Onde:

- **V (Conjunto de Variáveis / Símbolos Não-Terminais)**:
  As variáveis representam os objetivos ou subproblemas a serem resolvidos. Uma variável $h(n, O, D)$ representa a tarefa abstrata de "mover uma torre de n discos da torre de origem O para a torre de destino D".
  Formalmente:

$$V = \{h(n, O, D) \mid 1 \leq n \leq N,\ O, D \in \{A, B, C\},\ O \neq D\}$$

- **T (Conjunto de Terminais)**:
  Os terminais representam as ações atômicas do problema, ou seja, os movimentos de um único disco. Um terminal $p(O, D)$ representa a instrução "mova o disco do topo da torre O para o topo da torre D".
  Formalmente:

$$T = \{p(O, D) \mid O, D \in \{A, B, C\},\ O \neq D\}$$

- **S (Símbolo Inicial)**:
  O símbolo inicial representa o objetivo principal: mover N discos da torre A para

a torre C.
Formalmente:

$$S = h(N, A, C)$$

- **P (Conjunto de Regras de Produção)**:
  As regras de produção definem como um objetivo pode ser decomposto em subobjetivos menores. Para quaisquer torres distintas O (origem), D (destino) e X (auxiliar), temos:

  - **Regra Recursiva (para $n > 1$)**:

    $$h(n, O, D) \to h(n{-}1, O, X)\ p(O, D)\ h(n{-}1, X, D)$$

  - **Regra Base (para $n = 1$)**:

    $$h(1, O, D) \to p(O, D)$$

  Forma geral das regras da gramática:

$$\begin{aligned} S &\to h(N, A, C) \\ h(n, O, D) &\to h(n{-}1, O, X)\ p(O, D)\ h(n{-}1, X, D) \quad \text{para } n > 1 \\ h(1, O, D) &\to p(O, D) \end{aligned}$$

Exemplo de Derivação para $N = 2$

Para ilustrar o funcionamento da gramática, vamos derivar a solução para mover 2 discos da torre A para a torre C, usando a torre B como auxiliar.

$$\begin{aligned} S &\Rightarrow h(2, A, C) \\ &\Rightarrow h(1, A, B)\ p(A, C)\ h(1, B, C) \\ &\Rightarrow p(A, B)\ p(A, C)\ h(1, B, C) \\ &\Rightarrow p(A, B)\ p(A, C)\ p(B, C) \end{aligned}$$

A palavra final gerada, $p(A, B)\ p(A, C)\ p(B, C)$, representa exatamente a sequência correta de movimentos. O autômato com pilha que será apresentado a seguir executa essa derivação de forma procedural.

2.3 Construindo um Autômato com Pilha

Enquanto a Gramática Livre de Contexto descreve *o que* é a solução da Torre de Hanoi de forma declarativa, o Autômato com Pilha (PDA) nos fornece um modelo de máquina que descreve *como* construir essa solução passo a passo. O PDA implementado neste trabalho é uma adaptação direta do modelo de Yordzhev, projetado para simular as derivações da gramática G_N de forma procedural.

Uma característica fundamental deste autômato é que ele opera de maneira não convencional: ele não lê uma fita de entrada. Em vez disso, ele usa transições vazias (ε-transições) para manipular sua pilha. A pilha não serve para validar uma entrada, mas sim como um plano de ação dinâmico. O autômato inicia com o objetivo principal na pilha e, autonomamente, o refina até gerar a sequência completa de movimentos.

Definição Formal do PDA

O autômato M é definido como uma 7-tupla:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Onde:

- Q (Conjunto de estados):
  Como toda a lógica está na pilha, utilizamos apenas um único estado:

  $$Q = \{q_0\}$$

- Σ (Alfabeto de entrada):
  Não há entrada a ser lida:

  $$\Sigma = \emptyset$$

- Γ (Alfabeto da pilha):
  Inclui variáveis e terminais da gramática, além do símbolo inicial da pilha:

  $$\Gamma = \{h(n, O, D) \mid 1 \leq n \leq N,\ O, D \in \{A, B, C\},\ O \neq D\} \cup \{p(O, D) \mid O \neq D\} \cup \{Z_0\}$$

- δ (Função de transição):
  Define como a pilha é manipulada. Para quaisquer torres distintas O, D e X, temos:

  - **Transição de Inicialização:**

    $$\delta(q_0, \varepsilon, Z_0) = \{(q_0, h(N, A, C))\}$$

  - **Transição de Expansão (para $n > 1$):**

    $$\delta(q_0, \varepsilon, h(n, O, D)) = \{(q_0, h(n{-}1, O, X)\ p(O, D)\ h(n{-}1, X, D))\}$$

  - **Transição de Simplificação (caso base):**

    $$\delta(q_0, \varepsilon, h(1, O, D)) = \{(q_0, p(O, D))\}$$

  - **Transição de Execução:**

    $$\delta(q_0, \varepsilon, p(O, D)) = \{(q_0, \varepsilon)\}$$

- q_0 (Estado inicial):
  Estado inicial e único:

  $$q_0$$

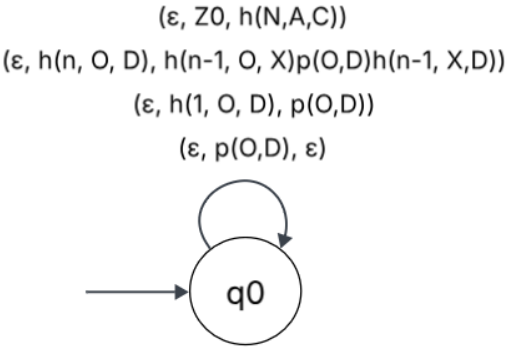
- Z_0 (Símbolo inicial da pilha):
  Símbolo usado para marcar o fundo da pilha:

  $$Z_0$$

- F (Estados finais):
  Como o autômato não aceita por estado, mas sim por pilha vazia:

$$F = \emptyset$$

Figura 9: Autômato com Pilha



Fluxo de Execução do Autômato

1. **Inicialização:** O autômato começa no estado q_0 com Z_0 na pilha. A transição $\delta(q_0, \varepsilon, Z_0)$ substitui Z_0 por $h(N, A, C)$, o objetivo principal.
2. **Loop de Processamento:** Enquanto houver símbolos na pilha, o autômato executa transições com base no topo:
   - Se for $h(n, O, D)$, ele empilha a sequência recursiva.
   - Se for $h(1, O, D)$, substitui por $p(O, D)$.
   - Se for $p(O, D)$, consome o símbolo.
3. **Término:** A computação termina quando a pilha está vazia.

3 Implementação do Simulador

A simulação do Autômato com Pilha foi desenvolvida na linguagem Python, aproveitando sua clareza e flexibilidade para modelar os componentes teóricos de forma eficiente. A implementação foi estruturada em torno de duas classes principais: `Pilha`, que fornece a funcionalidade básica de uma estrutura de dados LIFO (Last-In, First-Out), e `AutomatoPilha`, que encapsula a lógica e o estado do Autômato com Pilha para resolver o problema da Torre de Hanoi.

3.1 Estrutura das Classes

Classe `Pilha`

Esta classe implementa uma pilha genérica. Um detalhe importante é a presença do símbolo `#`, que é adicionado no momento da inicialização para representar o marcador de fundo da pilha, Z_0, do modelo formal. O método `mostrar()` foi customizado para formatar os símbolos da pilha (tuplas) de maneira legível, facilitando a análise da saída do programa.

Classe `AutomatoPilha` (Autômato com Pilha)

Esta é a classe central do simulador. Suas instâncias representam um autômato configurado para resolver o problema para um número específico de discos.

- **Construtor `__init__(self, n_discos, origem='A', destino='C')`:**
  - `self.estado_atual`: Mantido como `"q0"`, representando o único estado q_0.
  - `self.torres`: Um dicionário onde as chaves são os nomes dos pinos (`'A'`, `'B'`, `'C'`) e os valores são listas que representam os discos em cada pino.
  - `self.pilha`: Uma instância da classe `Pilha`.
  - **Simulação da Transição Inicial**: O construtor também é responsável por simular a primeira transição do autômato. Com base no número de discos n, ele empilha a sequência inicial de tarefas, espelhando a transição $\delta(q_0, \varepsilon, Z_0)$. Esta abordagem pré-carrega o plano de ação, preparando o autômato para iniciar seu ciclo de execução.
- **Método `_obter_torre_auxiliar(self, origem, destino)`:**
  Uma função utilitária que calcula a torre auxiliar a partir da torre de origem e torre de destino, usando operações de conjunto. Isso torna o código mais semântico e legível do que usar cálculos numéricos.
- **Método `_imprimir_passo(...)`:**
  Este método é crucial para a geração da saída detalhada. A cada passo do autômato, ele imprime o estado completo da simulação, incluindo o número do passo, a transição formal que foi executada, uma descrição da ação, o estado visual dos pinos e o conteúdo atual da pilha. Essa formatação foi projetada para conectar explicitamente a execução do código ao modelo teórico.

3.2 Lógica de Execução

O coração do simulador é o método `executar()`. Ele implementa o loop de processamento do autômato, que continua enquanto a pilha não estiver vazia (contendo apenas o marcador `#`).

- **Loop Principal**: `while not self.pilha.vazia()`
- **Consumo de Símbolo**: Em cada iteração, `self.pilha.desempilhar()` consome o símbolo do topo da pilha.
- **Lógica de Transição (Simulação de δ)**: Uma estrutura condicional `if/elif` analisa o símbolo desempilhado e executa a lógica correspondente, simulando a função de transição δ:
  - Se o símbolo é um objetivo (`('h', ...)`): O código verifica se $n = 1$ (caso base) ou $n > 1$ (caso recursivo).
    - Para $n = 1$, aplica a transição de simplificação, empilhando a ação `('p', ...)` correspondente.

    - Para $n > 1$, aplica a transição de expansão, empilhando as três subtarefas (`h`, `p`, `h`) na ordem correta.
  - Se o símbolo é uma ação (`('p', ...)`): O código executa a transição de execução. Isso envolve a manipulação das listas em `self.torres` para simular o movimento do disco. Esta é a única ação que altera o estado físico do quebra-cabeça.

- **Término**: O loop se encerra quando o último símbolo, o marcador `#`, é desempilhado. Ao final, uma mensagem de conclusão e o número total de movimentos são exibidos.

Exemplo de Execução com 2 Discos

A seguir, mostramos a saída do terminal Linux para a simulação do Autômato com Pilha com entrada de 2 discos:

Listing 4: Saída da simulação para 2 discos

```
--- Iniciando Simulacao com Automato com Pilha ---

Digite o numero de discos: 2

--- Passo 0 ---
Transicao: δ(q0, ε, #) → ⟨q0, h(2,A→C)⟩
Pilha (topo na esquerda): ['h(2,A→C)', '#']
Estado das Torres:
  Pino A: | 2 1
  Pino B: |
  Pino C: |

--- Passo 1 ---
Transicao: δ(q0, ε, h(2,A→C)) → ⟨q0, h(1,A→B)p(A→C)h(1,B→C)⟩
Pilha (topo na esquerda): ['h(1,A→B)', 'p(A→C)', 'h(1,B→C)', '#']
Estado das Torres:
  Pino A: | 2 1
  Pino B: |
  Pino C: |

--- Passo 2 ---
Transicao: δ(q0, ε, h(1,A→B) → ⟨q0, p(A→B)⟩
Pilha (topo na esquerda): ['p(A→B)', 'p(A→C)', 'h(1,B→C)', '#']
Estado das Torres:
  Pino A: | 2 1
  Pino B: |
  Pino C: |

--- Passo 3 ---
Transicao: δ(q0, ε, p(A→B)) → ⟨q0, ε⟩
Acao: Mover disco 1 de A para B
Pilha (topo na esquerda): ['p(A→C)', 'h(1,B→C)', '#']
Estado das Torres:
```

```
  Pino A: | 2
  Pino B: | 1
  Pino C: |

--- Passo 4 ---
Transicao: δ(q0, ε, p(A→C)) → ⟨q0, ε⟩
Acao: Mover disco 2 de A para C
Pilha (topo na esquerda): ['h(1,B→C)', '#']
Estado das Torres:
  Pino A: |
  Pino B: | 1
  Pino C: | 2

--- Passo 5 ---
Transicao: δ(q0, ε, h(1,B→C) → ⟨q0, p(B→C)⟩
Pilha (topo na esquerda): ['p(B→C)', '#']
Estado das Torres:
  Pino A: |
  Pino B: | 1
  Pino C: | 2

--- Passo 6 ---
Transicao: δ(q0, ε, p(B→C)) → ⟨q0, ε⟩
Acao: Mover disco 1 de B para C
Pilha (topo na esquerda): ['#']
Estado das Torres:
  Pino A: |
  Pino B: |
  Pino C: | 2 1

--- Simulacao Concluida ---
Numero total de movimentos de disco: 3
```

4 Conclusão

Este trabalho se propôs a resolver o clássico problema da Torre de Hanoi através de uma abordagem formal, implementando um simulador baseado no modelo teórico de um Autômato com Pilha (PDA). Ao final do projeto, foi possível desenvolver com sucesso um programa em Python que não apenas gera a solução ótima para um número N de discos, mas também fornece um rastreamento detalhado de sua execução, conectando cada passo do processo à teoria de autômatos.

A implementação demonstrou de forma prática a equivalência entre a descrição declarativa de um algoritmo, representada pela Gramática Livre de Contexto, e sua execução procedural, modelada pelo Autômato com Pilha. O simulador validou o modelo teórico ao gerar corretamente a sequência de $2^N - 1$ movimentos para todas as entradas testadas, alcançando o objetivo final de mover todos os discos para a torre de destino seguindo as regras do problema. A saída detalhada, que referencia explicitamente as transições formais (δ), serviu como uma ferramenta eficaz para visualizar e

compreender como a máquina teórica opera para resolver o quebra-cabeça.

No entanto, durante o desenvolvimento e a análise do projeto, foram identificadas nuances e desafios importantes que merecem destaque:

- **Reinterpretação do Papel do Autômato**: A principal reflexão teórica deste trabalho reside na adaptação do papel do PDA. Formalmente, um autômato é um reconhecedor de linguagens. No entanto, para resolver a Torre de Hanoi, o modelo foi empregado como um gerador de processo. As instruções $p(O, D)$ foram reinterpretadas como gatilhos para ações externas, destacando uma flexibilidade no uso de modelos formais para além de sua definição estrita.

- **Limitações do Modelo e Estruturas Adicionais**: O modelo de um Autômato com Pilha, por si só, é insuficiente para simular completamente o problema. O PDA pode gerar a sequência de instruções, mas não pode gerenciar o estado físico do problema. Para superar essa limitação, foi necessário acoplar o modelo formal a uma estrutura de dados adicional — uma lista de listas representando os pinos (`self.torres`), que gerencia o estado do ambiente.

- **Desafios de Escalabilidade e Limites Práticos**: A natureza exponencial do problema da Torre de Hanoi ($2^N - 1$ movimentos) impõe severas limitações práticas à execução para um número elevado de discos. Durante a fase de testes, foram realizadas execuções com um número crescente de discos para avaliar a robustez do simulador. Em um teste com $N = 35$ discos, o tempo de execução se tornou excessivo, exigindo a interrupção manual do processo. Devido à duração extrema, não foi possível determinar o limite máximo de discos suportado pela implementação ou se ocorreria uma falha de "estouro de pilha"(Stack Overflow) devido à profundidade da recursão simulada na pilha do autômato. Isso demonstra que, embora o modelo teórico seja válido para qualquer N, sua simulação prática é limitada pelo tempo de processamento e pelo consumo de memória.

Apesar desses desafios, o projeto foi um sucesso em seu objetivo principal. Ele não só entregou uma solução funcional para a Torre de Hanoi, mas o fez de uma maneira que demonstra a utilidade dos autômatos e das linguagens formais. A implementação serviu como uma ponte tangível entre a teoria abstrata estudada em sala de aula e sua aplicação prática na resolução de um problema algorítmico, demonstrando que os modelos formais são mais do que construções teóricas — são ferramentas poderosas para o pensamento computacional.

Referências

[1] YORDZHEV, Krasimir. *An entertaining example of using the concepts of context-free grammar and pushdown automation*. Open Journal of Discrete Mathematics, Blagoevgrad, v. 2, n. 3, p. 1–4, 2012. DOI: 10.4236/ojdm.2012.23020. Disponível em: https://www.scirp.org/html/6-1200073_21127.htm. Acesso em: 7 jul. 2025.