Torre de Hanoi (artigo)

By: Higor Ferreira Alves Santos
v1.0

Proposta

Usar o algorítmo DFS (Depth First Search. Busca em profundidade), para solucionar o jogo da torre de Hanoi

Resumo

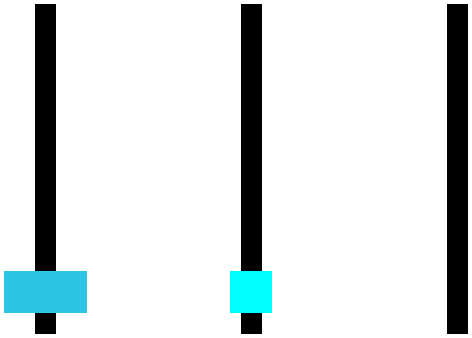
Para propósitos de desafio, optou-se por não utilizar nenhuma bilbioteca externa ao python afim de se tentar aplicar o conceito puramente nos recursos oferecidos pela linguagem no prazo definido pelo orientador (23/05/2025). O algorítmo retorna ao final um caminho solução para o problema. Até a data de 25/05/2025, o algorítmo ainda não está retornando uma resposta satisfatória para o jogo com três blocos, mas apenas para 2. Acredita-se que o problema está em como os nós filhos são gerados, a combinação de diferentes estados a partir de um estado pai não está gerando passos considerados corretos dentro das normas do jogo. Para desenhar a sequência de passos foi escolhido o padrão SVG (Scalable Vector Graphics). Padrão este de fácil compreenção tanto para seres humanos, quanto facilmente lido por qualquer navegador Web ou aplicativo de visualização de fotos.

Solução para dois blocos:

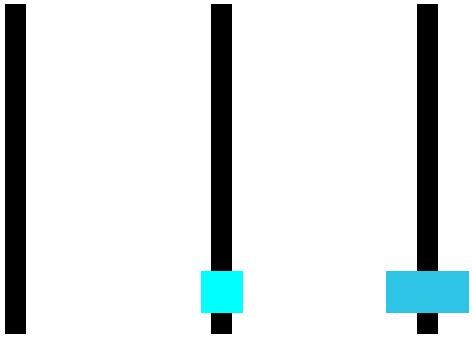
Passo 1

Passo 2

Passo 3



Passo 4

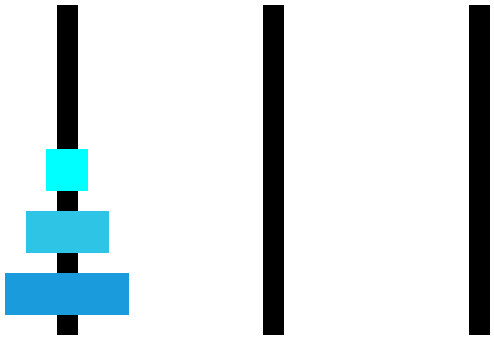


Passo 5

Nota-se que a solução se dá em 5 passos, com o passo 2 parecendo-se desnecessário.

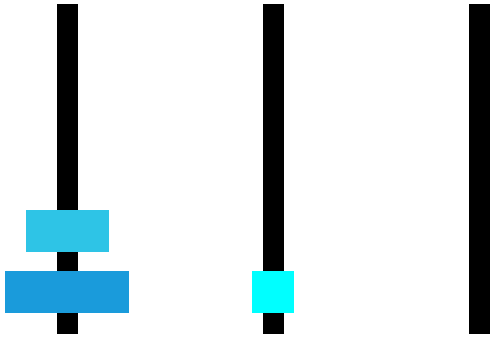
Solução para três blocos:

Passo 1

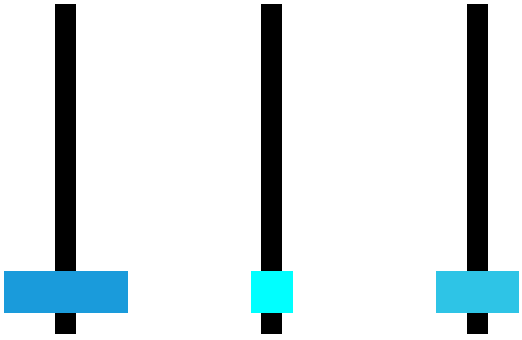


Passo 2

Passo 3



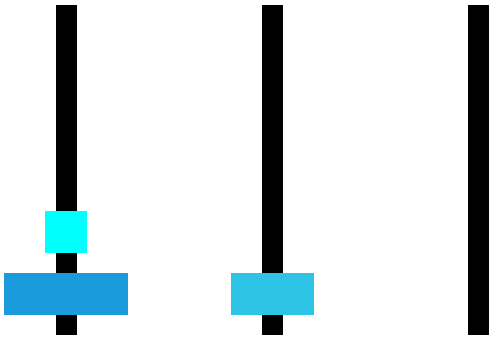
Passo 4



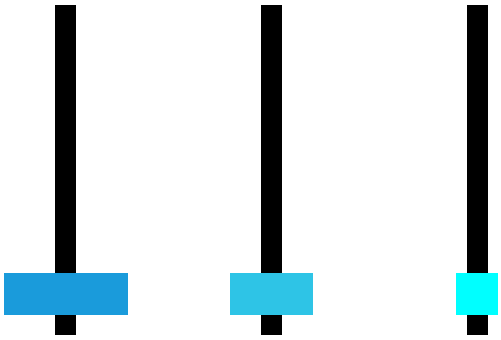
Passo 5

Passo 6

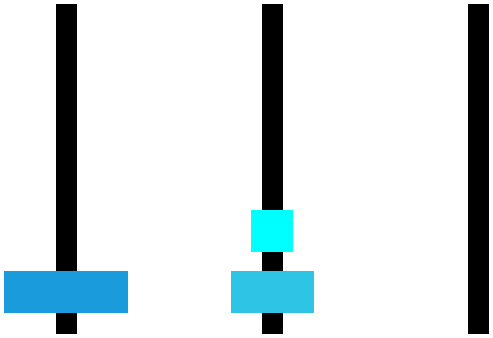
Passo 7



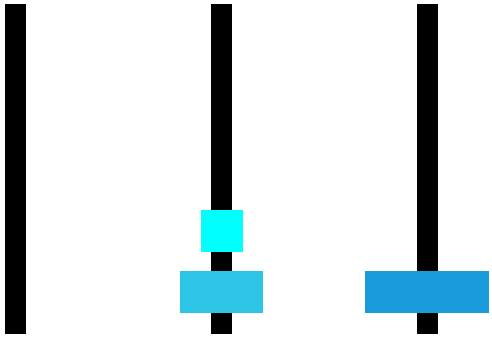
Passo 8



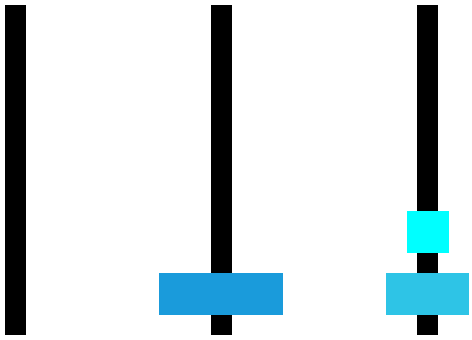
Passo 9



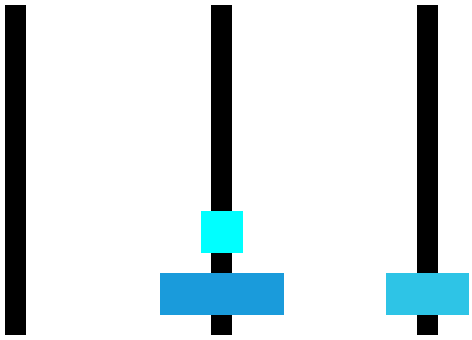
Passo 10



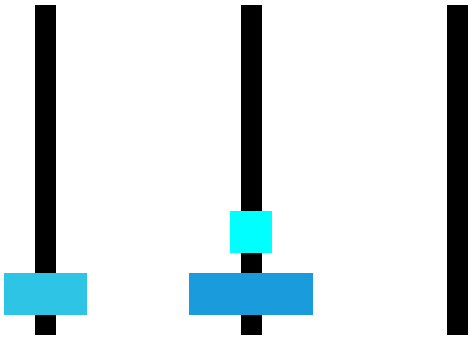
Passo 11



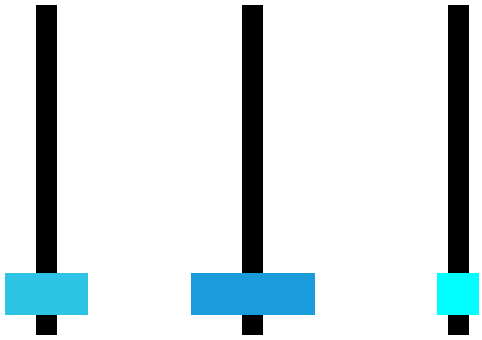
Passo 12



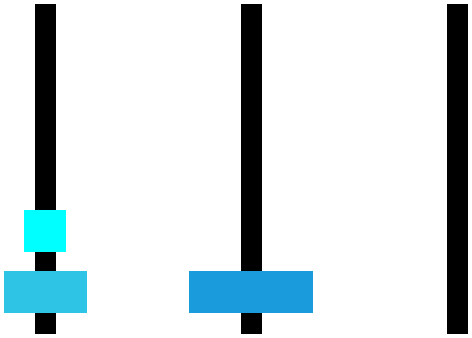
Passo 13



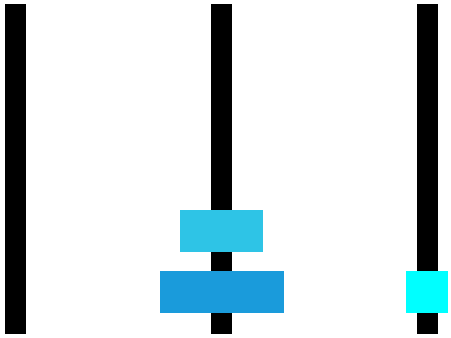
Passo 14



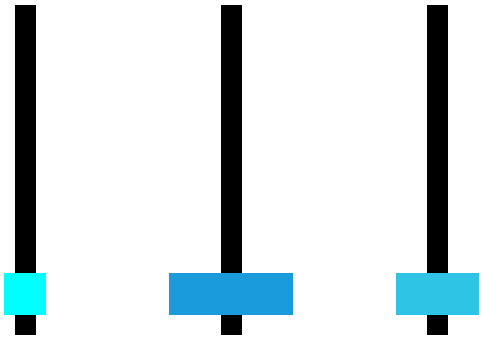
Passo 15



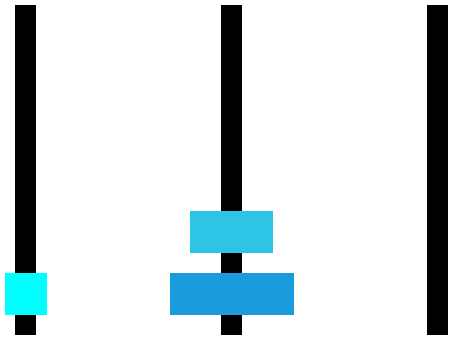
Passo 16



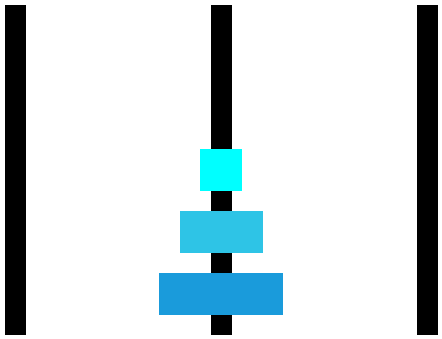
Passo 17



Passo 18



Passo 19



Aqui podemos notar claramente onde o algorítimo se perde. Do passo 10 para o passo 11, as torres 2 e 3 simplesmente trocam de lugar. Algo semelhante também ocorre do passo 15 para o 16 e 17.

Implementação

O ponto de entrada do código é o arquivo **Hanoi.py**. Todo o código fonte pode ser encontrado em https://github.com/HigorFerreira/PosGraduacao/tree/master/InteligenciaArtificial, **commit c5e65541e7300aa641a64794e87edfed644991f4**.

> Hanoi.py

```
from hanoi_lib import HanoiState, Tree, Node
import os

# Criando a árvore
tree = Tree(Node(HanoiState([ 3, 2, 1 ])))

# Montando a árvore
print("Mounting tree...")
tree.mountTree(tree.root)
print("Tree mountted")


# Função Objetivo
def goalFunction(node: Node):
    towerA = node.get_state().towers[0]
    towerB = node.get_state().towers[1]
```

```
    towerC = node.get_state().towers[2]

    return len(towerA) == 0 and len(towerB) == 0 or len(towerA) == 0 and len(tow

# Utilizando DFS para procurar a solução
print("Finding a path...")
sucess, path = tree.dfs(tree.root, lambda node: goalFunction(node))
if sucess: print("Path found")
else: print("No path found")

# Desenhando os passos
if not os.path.exists(os.path.join(os.getcwd(), 'path')):
    os.mkdir("path", 0o777)

for i, step in enumerate(path):
    with open(f"path/step-{i+1}.svg", "w") as f:
        f.write(step.get_state().generateImage(label=f"Passo {i+1}"))
```

Nota-se que o DFS utiliza uma **lambda function** para testar se a condição desejada foi alcançada:

```
sucess, path = tree.dfs(tree.root, lambda node: goalFunction(node))
```

A condição desejada é quando há duas torres vazias, e uma delas é a torre inicial:

```
def goalFunction(node: Node):
    towerA = node.get_state().towers[0]
    towerB = node.get_state().towers[1]
    towerC = node.get_state().towers[2]

    return len(towerA) == 0 and len(towerB) == 0 or len(towerA) == 0 and len(tow
```

Modelagem

Para modelar o jogo, optou-se por uma lista de três listas de **blocos**, cada um representando uma torre. Os blocos são simplesmente número que variam de 1 a 4. Naturalmente, 4 é o maior bloco possível e 1 o menor:

> hanoi_lib/HanoiState.py

```python
from .types import Block
from image_build import plot_game
import copy

class HanoiState:
    towers: list[list[Block]]
    def __init__(self,
            towerA: list[Block] = [],
            towerB: list[Block] = [],
            towerC: list[Block] = []
        ):
        self.towers = [
            copy.deepcopy(towerA),
            copy.deepcopy(towerB),
            copy.deepcopy(towerC),
            # towerA,
            # towerB,
            # towerC,
        ]

    def isValid(self) -> bool:
        for tower in self.towers:
            for i, block in enumerate(tower):
                if i == 0: continue
                if block > tower[i-1]: return False
        return True

    def generatePossibleWays(self) -> list['HanoiState']:
        states: list['HanoiState'] = []

        for i in range(len(self.towers)):
            if len(self.towers[i]) != 0:
                indexes = set(range(len(self.towers)))
                indexes.remove(i)
                for j in indexes:
                    towers = copy.deepcopy(self.towers)
                    block = towers[i].pop()
                    towers[j].append(block)
                    states.append(HanoiState(
                        towers[0],
                        towers[1],
                        towers[2],
                    ))

        return states
```

```
    def generateImage(self, x=0, y=0, label=""):
        return plot_game(self.towers, label=label)

    def serialize(self):
        return "\n".join(
            list(map(
                lambda blocks: "".join(list(map(lambda block: f"{block}", blocks
                self.towers
            ))
        )

    def __eq__(self, value):
        if type(value) == HanoiState: return self.serialize() == value.serialize
        elif type(value) == str: return self.serialize() == value
        else: return False
```

Cada instância **HanoiState** recebe uma cópia das torres que são passadas pois o estado deve ser **imutável**. Isto é bastante útil para a função **generatePossibleWays**, pois a partir de um estado pai, diversas manipulações são feitas de forma a gerar os estados filhos, de forma que o estado pai continue mantendo seu estado após operações de **desempilhamento** em suas respectivas torres.
É a função **generatePossibleWays** em que acredita-se estar gerando os estados problemáticos da solução de três blocos discutida anteriormente.
A função **isValid** checa se não há blocos maiores por cima de blocos menores. Mais tarde, esta funcão ajudará o dfs no percorrimento.
A função **generateImage** gera a imagem do estado atual.
as função **serialize** e **__eq__** trabalham em conjunto para testar se dois estados são iguais. Graças a ela, a saída do seguinte código é **True**:

```
a = HanoiState([3, 2, 1])
b = HanoiState([3, 2, 1])
print(a == b)
```

Mesmo que sejam objetos diferentes, seus estados representam a mesma coisa, e tal igualdade será avaliada como verdadeira.

Nó

Um nó consiste basicamente de seu estado e seus estados filhos.

> hanoi_lib/Node.py

```
from .HanoiState import HanoiState

class Node:
    state: HanoiState
    children: list['Node'] = []
    def __init__(self, state: HanoiState):
        self.state = state

    def set_children(self, children: list['Node']):
        self.children = children

    def get_children(self): return self.children

    def get_state(self):
        return self.state
```

Árvore (tree)

A montagem da árvore ocorre nesta parte. Em **mountTree** pode-se notar o uso do **generatePossibleWays** nos nós. A montagem é feita até o décimo nó de profundidade, pois a partir disso, o consumo de memória começa a se tornar insustentável. Uma possível melhor abordagem para solucionar este problema é montar a árvore dinamicamente a medida que o DFS vai percorrendo-a:

```
from hanoi_lib import Node
from typing import Callable

class Tree:
    root: Node
    def __init__(self, root: Node) -> None:
        self.root = root

    def mountTree(self, base: Node, level=1, maxLevel=10):
        if level == maxLevel: return
        ways = base.get_state().generatePossibleWays()
        ways = list(map(lambda x: Node(x), ways))
        base.set_children(ways)
        for child in base.get_children():
            self.mountTree(child, level+1, maxLevel)

    def dfs(
                    self,
                    start_node: Node,
```

```
                    check: Callable[['Node'], bool]
            ) -> tuple[bool, list[Node]]:

        counter = 0
        visited: set[str] = set()
        stack: list[Node] = []
        processed_stack: list[Node] = []

        stack.append(start_node)

        while True:
            if len(stack) == 0: return False, processed_stack
            node = stack.pop()
            if node.get_state().serialize() in visited: continue
            if not node.get_state().isValid(): continue

            # print(stack)
            visited.add(node.get_state().serialize())
            processed_stack.append(node)
            # print(processed_stack)
            stack = [ *stack, *node.get_children() ]
            # print(stack)
            # return processed_stack
            counter += 1
            if(check(node)): return True, processed_stack
```

A busca em profundidade é implementada na funcão **dfs**. Nota-se que o percorrimento é iniciado com um nó de partida. Utiliza-se um conjunto denominado ***visited*** para não entrar em nós já visitados. A função **check** que é recebida como parâmetro é utilizada como condição de parada do algorítimo. Caso a árvore seja percorrida em sucesso, o retorno é dado em: `if len(stack) == 0: return False, processed_stack`.

Conclusão

O objetivo de encontrar uma solução para três blocos não foi alcançado, uma vez que a saída gera passos incorretos. Para futuro trabalho e revisão deste artigo, além de possíveis melhorias, é necessário:

- a) Melhorar a função **generatePossibleWays** para gerar filhos corretamente
- b) Montar a árvore dinamicamente para prevenir estouros de memória