Resolução de Quebra-Cabeça 3x3 usando BFS

Esta apresentação aborda a resolução de quebra-cabeças 3x3 utilizando o algoritmo de Busca em Largura (BFS).

BFS é uma técnica de busca de grafos que explora todos os nós em um nível de profundidade antes de passar para o próximo. Em termos de quebra-cabeças, este método garante que a solução de menor número de movimentos será encontrada, caso exista. A implementação do jogo e do algoritmo é realizada em Python, utilizando a biblioteca Tkinter para a interface gráfica.

Introdução

O objetivo desta apresentação é apresentar o desenvolvimento de um jogo de quebra-cabeça 3x3, com foco na aplicação do algoritmo BFS para encontrar a solução. Além disso, será demonstrada a implementação utilizando a biblioteca Tkinter em Python. O quebra-cabeça 3x3, também conhecido como "Slide Puzzle", ou "Puzzle 8" consiste em oito peças numéricas, e um espaço vazio, e o objetivo é organizar as peças em ordem numérica através de movimentos válidos.

1 Objetivo Principal

Demonstrar a aplicação prática do algoritmo BFS na resolução de um quebra-cabeça 3x3.

2 Benefícios

Compreensão do algoritmo BFS, sua aplicação em problemas práticos e desenvolvimento de uma aplicação gráfica interativa.

Conceitos Básicos

Para entender a resolução do quebra-cabeça através do BFS, é preciso definir alguns conceitos básicos. Em essência, o quebra-cabeça é um problema de busca em espaço de estados.

Estado	Uma representação do quebra-cabeça em um dado momento, definindo a posição de cada peça e do espaço vazio. É um conjunto de 8 números e uma posição vazia.
Estado Inicial	O arranjo aleatório inicial das peças, a partir do qual se busca a solução.
Ações	Os movimentos possíveis do espaço vazio, que são: esquerda, direita, cima e baixo. Cada movimento resulta em um novo estado.

Modelo de Transição	A função que recebe um estado e uma ação, retornando o novo estado resultante da ação aplicada.
Teste de Objetivo	Verifica se o estado atual é o estado desejado, onde as peças estão em ordem numérica e o espaço vazio está na posição correta.
Custo de Caminho	Representa o número de movimentos (ações) necessários para atingir um determinado estado. Cada movimento tem custo 1.

Código - Busca em Largura (BFS)

A implementação do BFS para solucionar o quebra-cabeça é realizada em Python. O código abaixo ilustra a função 'bfs_solve', que recebe o estado inicial e retorna o caminho de ações (movimentos) para alcançar o estado objetivo. A implementação utiliza uma fila (queue) para armazenar os estados a serem explorados, e um conjunto (set) para armazenar os estados já visitados, evitando a revisitação de estados redundantes. A função retorna o caminho de ações caso a solução seja encontrada, ou retorna None caso a solução não exista.

```python
def solve_puzzle(self):
        # Tenta resolver o quebra-cabeça usando busca em largura (BFS)
        print("solve_puzzle() foi chamada")
        solution = self.bfs_solve()
        if solution:
            print("Solução encontrada:", solution)
            self.animate_solution(solution)
        else:
            print("Nenhuma solução encontrada")

    def bfs_solve(self):
        # Algoritmo de busca em largura (BFS) para encontrar a solução do quebra-cabeça
        print("bfs_solve() foi chamada")
        start = tuple(tuple(row) for row in self.board)
        goal = (1, 2, 3, 4, 5, 6, 7, 8, None)
        queue = deque([(start, [])])  # Fila para estados a serem explorados
        visited = set()  # Conjunto para estados já visitados
        directions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

        def get_next_state(state, blank_i, blank_j, di, dj):
            # Gera o próximo estado movendo o espaço em branco na direção especificada
            ni, nj = blank_i + di, blank_j + dj
            if 0 <= ni < 3 and 0 <= nj < 3:
                new_state = [list(row) for row in state]
                new_state[blank_i][blank_j], new_state[ni][nj] = new_state[ni][nj], new_state[blank_i][blank_j]
                return tuple(tuple(row) for row in new_state), ni, nj
            return None, blank_i, blank_j
```

```python
def solve_puzzle(self):
        # Tenta resolver o quebra-cabeça usando busca em largura (BFS)
        print("solve_puzzle() foi chamada")
        solution = self.bfs_solve()
        if solution:
            print("Solução encontrada:", solution)
            self.animate_solution(solution)
        else:
            print("Nenhuma solução encontrada")

    def bfs_solve(self):
        # Algoritmo de busca em largura (BFS) para encontrar a solução do quebra-cabeça
        print("bfs_solve() foi chamada")
        start = tuple(tuple(row) for row in self.board)
        goal = (1, 2, 3, 4, 5, 6, 7, 8, None)
        queue = deque([(start, [])])  # Fila para estados a serem explorados
        visited = set()  # Conjunto para estados já visitados
        directions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

        def get_next_state(state, blank_i, blank_j, di, dj):
            # Gera o próximo estado movendo o espaço em branco na direção especificada
            ni, nj = blank_i + di, blank_j + dj
            if 0 <= ni < 3 and 0 <= nj < 3:
                new_state = [list(row) for row in state]
                new_state[blank_i][blank_j], new_state[ni][nj] = new_state[ni][nj], new_state[blank_i][blank_j]
                return tuple(tuple(row) for row in new_state), ni, nj
            return None, blank_i, blank_j
```

Equipe e links

Tayane Cibely Batista Rodrigues

Leonardo Nunes Barros

Link repositório: https://github.com/leonardonb/ia_puzzle8.git

Github Tayane: https://github.com/TayaneCibely

Github Leonardo: https://github.com/leonardonb