Jogo Puzzle

Davi Caetano Tavares Ramos
1398113

Link do Github:
https://github.com/Caetano9/8-puzzle-Game

Código:

```
from ollections import deque
import heapq
import time

# Dimensão do tabuleiro
N = 3

# Vetores de deslocamento: ← → ↑ ↓
ROW = (0, 0, -1, 1)
COL = (-1, 1, 0, 0)

# Estrutura de estado
class PuzzleState:
    __slots__ = ("board", "x", "y", "depth", "cost")

    def __init__(self, board, x, y, depth, cost=0):
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth        # g(n)
        self.cost = cost          # f(n) para A*

    def __lt__(self, other):
        return self.cost < other.cost

# Funções utilitárias
GOAL_BOARD = ((1, 2, 3),
              (4, 5, 6),
              (7, 8, 0))

def is_goal_state(board):
```

```
    return tuple(map(tuple, board)) == GOAL_BOARD

def is_valid(x, y):
    return 0 <= x < N and 0 <= y < N

def board_to_tuple(board):
    return tuple(tuple(r) for r in board)

def print_board(board):
    for r in board:
        print(" ".join(" " if n == 0 else str(n) for n in r))
    print()

# Heurísticas para o A*
def h_manhattan(board):
    dist = 0
    for i in range(N):
        for j in range(N):
            v = board[i][j]
            if v:
                gi, gj = divmod(v - 1, N)
                dist += abs(i - gi) + abs(j - gj)
    return dist

def h_misplaced(board):
    return sum(1 for i in range(N) for j in range(N)
               if board[i][j] and board[i][j] != GOAL_BOARD[i][j])

# Busca em profundidade limitada (DFS)
def solve_dfs(start, x, y, depth_limit=50):
    stack = [PuzzleState(start, x, y, 0)]
    parent = {board_to_tuple(start): None}
    visited = {board_to_tuple(start)}
    expanded = 0

    while stack:
        curr = stack.pop()
        expanded += 1
        if is_goal_state(curr.board):
            return parent, curr, expanded
        if curr.depth >= depth_limit:
            continue
```

```
        for k in range(4):
            nx, ny = curr.x + ROW[k], curr.y + COL[k]
            if is_valid(nx, ny):
                new_board = [r[:] for r in curr.board]
                new_board[curr.x][curr.y], new_board[nx][ny] = \
                    new_board[nx][ny], new_board[curr.x][curr.y]
                key = board_to_tuple(new_board)
                if key not in visited:
                    visited.add(key)
                    ns = PuzzleState(new_board, nx, ny, curr.depth + 1)
                    parent[key] = curr
                    stack.append(ns)
    return None, None, expanded

# Busca em largura (BFS)
def solve_bfs(start, x, y):
    q = deque([PuzzleState(start, x, y, 0)])
    parent = {board_to_tuple(start): None}
    visited = {board_to_tuple(start)}
    expanded = 0

    while q:
        curr = q.popleft()
        expanded += 1
        if is_goal_state(curr.board):
            return parent, curr, expanded
        for k in range(4):
            nx, ny = curr.x + ROW[k], curr.y + COL[k]
            if is_valid(nx, ny):
                new_board = [r[:] for r in curr.board]
                new_board[curr.x][curr.y], new_board[nx][ny] = \
                    new_board[nx][ny], new_board[curr.x][curr.y]
                key = board_to_tuple(new_board)
                if key not in visited:
                    visited.add(key)
                    ns = PuzzleState(new_board, nx, ny, curr.depth + 1)
                    parent[key] = curr
                    q.append(ns)
    return None, None, expanded

# A* genérico
def solve_astar(start, x, y, h_func):
```

```
    open_heap = []
    parent = {board_to_tuple(start): None}
    g_cost = {board_to_tuple(start): 0}
    start_state = PuzzleState(start, x, y, 0, h_func(start))
    heapq.heappush(open_heap, start_state)
    expanded = 0

    while open_heap:
        curr = heapq.heappop(open_heap)
        expanded += 1
        if is_goal_state(curr.board):
            return parent, curr, expanded
        for k in range(4):
            nx, ny = curr.x + ROW[k], curr.y + COL[k]
            if is_valid(nx, ny):
                new_board = [r[:] for r in curr.board]
                new_board[curr.x][curr.y], new_board[nx][ny] = \
                    new_board[nx][ny], new_board[curr.x][curr.y]
                ntuple = board_to_tuple(new_board)
                tentative_g = curr.depth + 1
                if ntuple not in g_cost or tentative_g < g_cost[ntuple]:
                    g_cost[ntuple] = tentative_g
                    ns = PuzzleState(new_board, nx, ny,
                                     tentative_g,
                                     tentative_g + h_func(new_board))
                    parent[ntuple] = curr
                    heapq.heappush(open_heap, ns)
    return None, None, expanded

# Reconstrução e saída
def reconstruct(parent, node):
    path = []
    while node:
        path.append(node)
        node = parent[board_to_tuple(node.board)]
    return list(reversed(path))

def show_path(path):
    for step in path:
        print(f"Profundidade: {step.depth}")
        print_board(step.board)
```

```
def run_and_report(name, solver, *args, **kwargs):
    t0 = time.perf_counter()
    par, goal, expanded = solver(*args, **kwargs)
    dt = time.perf_counter() - t0
    if goal:
        print(f"{name:13} | tempo = {dt:7.4f} s | nós expandidos =
{expanded:6d} "
              f"| profundidade da solução = {goal.depth:2d}")
    else:
        print(f"{name:13} | não encontrou solução no limite dado.")

# Programa principal
if __name__ == "__main__":
    start_board = [[1, 2, 3],
                   [4, 0, 5],
                   [6, 7, 8]]
    x0, y0 = 1, 1  # posição inicial do branco

    print("Estado inicial:")
    print_board(start_board)

    run_and_report("DFS limit=50", solve_dfs, start_board, x0, y0,
depth_limit=50)
    run_and_report("BFS",          solve_bfs, start_board, x0, y0)
    run_and_report("A* Manhattan", solve_astar, start_board, x0, y0,
h_manhattan)
    run_and_report("A* Misplaced", solve_astar, start_board, x0, y0,
h_misplaced)

    par, goal, _ = solve_astar(start_board, x0, y0, h_manhattan)
    path = reconstruct(par, goal)
    show_path(path)

    # Mantém a janela do console aberta até o usuário confirmar
    input("Pressione Enter para sair...")
```

Relatório:

A implementação apresenta três algoritmos de busca: profundidade limitada (DFS-L), amplitude (BFS) e A*.

Esses algoritmos foram aplicados ao 8 puzzle, cujo objetivo é transformar o estado inicial em 1 2 3 4 5 6 7 8 0. O ambiente de teste fixou o tabuleiro inicial [[1, 2, 3], [4, 0, 5], [6, 7, 8]], atribuindo um custo unitário a cada deslocamento do espaço em branco. Todos os algoritmos partem da mesma representação de estado, armazenam o caminho percorrido e medem o número de nós expandidos, a profundidade da solução e o tempo decorrido com “time.perf_counter()” para permitir a comparação direta.

A busca em profundidade limitada foi configurada com um limite de 50, limite que é o suficiente para encontrar uma solução, mas não necessariamente para explorar todo o espaço. Sua operação sempre expande o sucessor mais profundo até atingir o teto de profundidade, o que produz margens de memória restritas, mas pode alongar o caminho até o objetivo.

A busca em largura visita nós em camadas equidistantes da raiz; a fila “deque” garante totalidade e uma optimização em ambientes de custo unitário, mas o algoritmo tende a consumir mais memória porque deve manter simultaneamente toda a fronteira do nível atual.

A* combina um custo cumulativo g(n) com uma estimativa h(n) do custo restante.

Para avaliar o impacto da qualidade heurística, implementamos a distância de Manhattan, que soma as distâncias horizontal e vertical de cada peça até sua posição objetivo, e a contagem de peças mal colocadas, ambas admissíveis de acordo com a definição clássica de heurística otimista e consistentes no sentido da desigualdade triangular.

Os experimentos produziram a seguinte saída:

Estado inicial:

1 2 3

4 5

6 7 8

DFS limit=50 | tempo = 0.9809 s | nós expandidos = 21473 | profundidade da solução = 46

BFS | tempo = 0.3537 s | nós expandidos = 4693 | profundidade da solução = 14

A* Manhattan | tempo = 0.0144 s | nós expandidos = 128 | profundidade da solução = 14

A* Misplaced | tempo = 0.0451 s | nós expandidos = 321 | profundidade da solução = 14

Profundidade: 0

1 2 3

4 5

6 7 8

Profundidade: 1

1 2 3

4 5

6 7 8

Profundidade: 2

1 2 3

4 5 8

6 7

Profundidade: 3

1 2 3

4 5 8

6 7

Profundidade: 4

1 2 3

4 5 8

 6 7

Profundidade: 5

1 2 3

 5 8

4 6 7

Profundidade: 6

1 2 3

5 8

4 6 7

Profundidade: 7

1 2 3

5 6 8

4 7

Profundidade: 8

1 2 3

5 6 8

4 7

Profundidade: 9

1 2 3

5 6

4 7 8

Profundidade: 10

1 2 3

5 6

4 7 8

Profundidade: 11

1 2 3

  5 6

4 7 8

Profundidade: 12

1 2 3

4 5 6

  7 8

Profundidade: 13

1 2 3

4 5 6

7   8

Profundidade: 14

1 2 3

4 5 6

7 8

DFS-L expandiu 21.473 nós, levou 0,981 s e retornou um caminho de profundidade 46; BFS expandiu 4.693 nós, levou 0,354 s e encontrou a solução ótima de profundidade 14; A* com Manhattan exigiu apenas 128 expansões e 0,014 s, enquanto A* com peças mal posicionadas exigiu 321 expansões e 0,045 s, ambas retornando profundidade 14. A diferença de desempenho aponta para dois aspectos.

Primeiro, heurísticas admissíveis e mais bem informadas reduzem drasticamente o fator de ramificação efetivo: Manhattan está mais próximo do custo real porque considera quantos

movimentos cada peça ainda requer, enquanto a contagem de peças fornece apenas um limite inferior aproximado, daí o motivo para cerca de 2,5 vezes mais expansões.

Segundo, embora o BFS também seja ótimo neste cenário de custo uniforme, ele paga o preço de visitar todos os nós nas primeiras treze camadas antes de atingir o objetivo, gerando sobrecarga temporal e espacial que A* evita ao priorizar nós com menor f(n) = g(n) + h(n).

A busca com profundidade limitada, mesmo encontrando uma solução, ilustra o risco de trajetórias longas: 46 movimentos contra o ótimo de 14. Esse alongamento se deve à estratégia de descer por um único ramo até forçar o backtracking, característica que, combinada à ausência de heurísticas e ao limite arbitrário, impede a garantia de otimalidade e pode causar explosões de custos quando o fator de ramificação é alto.

O método que “melhor desenvolveu o problema” foi, portanto, o A* com distância de Manhattan. Com o menor tempo, com a menor expansão de nós e produzindo a solução ótima. Em segundo lugar ficou o A* com peças mal posicionadas, demonstrando que mesmo uma heurística simples já traz ganhos substanciais em comparação ao BFS. O BFS, por sua vez, serve como referência para completude e otimalidade sem informações heurísticas, enquanto o DFS limitado se mostrou menos eficaz para esta instância.

Dado o mesmo ponto de partida e custo uniforme, A* expande, no máximo, o conjunto de nós igualmente promissores de acordo com a heurística escolhida, e nenhuma estratégia ótima e completa pode se sair melhor com a mesma estimativa. Conclui-se que, para o 8-Puzzle e as heurísticas admissíveis, a distância de Manhattan oferece o melhor compromisso entre qualidade da solução e custo computacional dentre os algoritmos selecionados.

Fontes: https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/ e pdfs disponibilizados no canvas.