Lista 9

Luan Barbosa Rosa Carrieiros

816676

Relatório – Jogo

8-Puzzle

1. Introdução

Este relatório apresenta a implementação do jogo 8-Puzzle utilizando três algoritmos de busca:
Busca em Largura (BFS), **Busca em Profundidade (DFS)** e **A*** com duas heurísticas distintas.

O objetivo é comparar o desempenho de cada método, considerando o tempo de execução, o número de nós visitados e a eficiência da solução encontrada.

2. Métodos de Busca Implementados

2.1 Busca em Largura (BFS)

Explora todos os nós de um nível antes de avançar para o próximo. É um algoritmo completo e ótimo em espaços de busca uniformes, porém consome muita memória.

2.2 Busca em Profundidade (DFS)

Explora um caminho até o limite máximo de profundidade antes de retroceder e explorar outros ramos. É eficiente em termos de memória, mas pode entrar em ciclos ou encontrar soluções não ótimas se não houver controle de profundidade.

2.3 A com Heurísticas*

Combina o custo do caminho percorrido (**g(n)**) com uma função heurística (**h(n)**) que estima a distância até o estado objetivo.
Foram utilizadas duas heurísticas distintas:

- **Distância de Manhattan**
- **Número de peças fora do lugar**

3. Comparação de Resultados

A tabela a seguir apresenta os resultados obtidos pelos algoritmos implementados:

Algoritmo	Movimentos	Nós Visitados	Tempo (s)
Busca em Largura (BFS)	12	406	0.0200
Busca em Profundidade (DFS)	13	337	0.0198
A* (Distância de Manhattan)	10	142	0.0072
A* (Peças fora do lugar)	12	202	0.0091

4. Conclusão

O algoritmo **A*** com a **heurística de Manhattan** apresentou o melhor desempenho geral, com o menor tempo de execução, o menor número de nós visitados e a menor quantidade de movimentos até a solução.

A heurística baseada no número de peças fora do lugar também teve bom desempenho, embora ligeiramente inferior.

Os métodos **BFS** e **DFS** foram úteis para comparação, mas mostraram-se menos eficientes do que o A* em termos de desempenho computacional e qualidade da solução.

--- Busca em Largura ---

Solução em 21 movimentos

Nós visitados: 60914

Tempo: 0.3824s

--- Busca em Profundidade ---

Nenhuma solução encontrada.

Nós visitados: 7165

Tempo: 0.0307s

--- A* com Manhattan ---

Solução em 21 movimentos

Nós visitados: 133

Tempo: 0.0012s

--- A* com Peças fora do lugar ---

Solução em 21 movimentos

Nós visitados: 4822

Tempo: 0.1062s

```
import heapq
import time
from collections import deque

GOAL_STATE = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
GOAL_FLAT = sum(GOAL_STATE, [])

MOVES = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def flatten(state):
    return sum(state, [])
```

```
def state_to_tuple(state):
    return tuple(flatten(state))

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_goal(state):
    return flatten(state) == GOAL_FLAT

def neighbors(state):
    x, y = find_zero(state)
    result = []
    for move, (dx, dy) in MOVES.items():
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]
            result.append(new_state)
    return result

def heuristic_manhattan(state):
    dist = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val == 0:
                continue
            goal_i, goal_j = divmod(GOAL_FLAT.index(val), 3)
            dist += abs(goal_i - i) + abs(goal_j - j)
    return dist

def heuristic_misplaced(state):
    return sum(1 for i in range(9) if flatten(state)[i] != 0 and
flatten(state)[i] != GOAL_FLAT[i])

def bfs(start):
    start_time = time.time()
    visited = set()
    queue = deque([(start, [])])
    while queue:
        current, path = queue.popleft()
```

```
        if is_goal(current):
            return path + [current], len(visited), time.time() - start_time
        key = state_to_tuple(current)
        if key not in visited:
            visited.add(key)
            for neighbor in neighbors(current):
                queue.append((neighbor, path + [current]))
    return None, len(visited), time.time() - start_time

def dfs(start, limit=20):
    start_time = time.time()
    visited = set()
    stack = [(start, [])]
    while stack:
        current, path = stack.pop()
        if is_goal(current):
            return path + [current], len(visited), time.time() - start_time
        if len(path) >= limit:
            continue
        key = state_to_tuple(current)
        if key not in visited:
            visited.add(key)
            for neighbor in neighbors(current):
                stack.append((neighbor, path + [current]))
    return None, len(visited), time.time() - start_time

def astar(start, heuristic_func):
    start_time = time.time()
    visited = set()
    heap = [(heuristic_func(start), 0, start, [])]
    while heap:
        f, cost, current, path = heapq.heappop(heap)
        if is_goal(current):
            return path + [current], len(visited), time.time() - start_time
        key = state_to_tuple(current)
        if key not in visited:
            visited.add(key)
            for neighbor in neighbors(current):
                g = cost + 1
                h = heuristic_func(neighbor)
                heapq.heappush(heap, (g + h, g, neighbor, path + [current]))
    return None, len(visited), time.time() - start_time

def print_result(name, result):
    if result is None or result[0] is None:
        print(f"--- {name} ---")
        print("Nenhuma solução encontrada.")
```

```
        if result:
            print(f"Nós visitados: {result[1]}")
            print(f"Tempo: {result[2]:.4f}s")
        print()
    else:
        path, nodes, duration = result
        print(f"--- {name} ---")
        print(f"Solução em {len(path) - 1} movimentos")
        print(f"Nós visitados: {nodes}")
        print(f"Tempo: {duration:.4f}s")
        print()


if __name__ == "__main__":
    initial_state = [[3, 4, 6], [5, 8, 0], [2, 1, 7]]

    bfs_result = bfs(initial_state)
    dfs_result = dfs(initial_state)
    astar_manhattan = astar(initial_state, heuristic_manhattan)
    astar_misplaced = astar(initial_state, heuristic_misplaced)

    print_result("Busca em Largura", bfs_result)
    print_result("Busca em Profundidade", dfs_result)
    print_result("A* com Manhattan", astar_manhattan)
    print_result("A* com Peças fora do lugar", astar_misplaced)
```