8-Puzzle

Edson Almeida*
edson.almeida.1435541@sga.pucminas.br
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte, Minas Gerais

ABSTRACT

Este estudo apresenta uma análise comparativa de métodos de busca clássicos e heurísticos para resolução do 8-Puzzle, problema fundamental em inteligência artificial. Foram implementados e avaliados três algoritmos: Busca em Largura (BFS), Busca em Profundidade (DFS) e A* com duas heurísticas (Peças Fora do Lugar e Distância de Manhattan). Utilizando uma configuração específica com solução em dois movimentos, os resultados revelaram diferenças significativas na eficiência computacional: enquanto o BFS visitou 13 nós em 0.07 ms, o DFS surpreendeu ao encontrar a solução ótima com apenas 3 nós visitados (0.01 ms), comportamento atípico explicado pela proximidade da solução e ordem de expansão. O algoritmo A* demonstrou superioridade geral, igualando a eficiência do DFS (3 nós, 0.04 ms) porém com garantia matemática de optimalidade. A comparação entre heurísticas mostrou equivalência prática neste cenário simples, porém a Distância de Manhattan destacou-se como mais informativa para problemas complexos. Conclui-se que métodos de busca informada (A*) são preferíveis para versões desafiadoras do puzzle, com a heurística de Manhattan oferecendo melhor equilíbrio entre custo computacional e precisão. Este trabalho reforça a importância da seleção de algoritmos baseada nas propriedades do problema, provendo insights para aplicações em planejamento automatizado e sistemas de recomendação.

ACM Reference Format:
Edson Almeida. 2025. 8-Puzzle. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Este relatório compara o desempenho de três algoritmos de busca (BFS, DFS e A*) na resolução do 8-Puzzle, utilizando duas heurísticas diferentes para o A*. Todos os testes foram realizados na seguinte configuração:

$$\text{Estado Inicial} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 7 & 8 & 6 \end{bmatrix}, \quad \text{Estado Objetivo} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

2 MÉTODOS IMPLEMENTADOS

Busca em Largura (BFS)

- **Funcionamento**: Expande nós nível por nível usando fila (FIFO)
- **Vantagens**:
  - Garante solução ótima
  - Completo para espaços de estado finitos
- **Desvantagens**:
  - Alto consumo de memória

Busca em Profundidade (DFS)

- **Funcionamento**: Explora caminhos até o fim usando pilha (LIFO)
- **Vantagens**:
  - Baixo uso de memória
- **Desvantagens**:
  - Não garante solução ótima
  - Pode entrar em loops infinitos

Algoritmo A*

- **Funcionamento**: Combina custo do caminho ($g(n)$) e heurística ($h(n)$)
- **Fórmula**: $f(n) = g(n) + h(n)$

3 HEURÍSTICAS

Peças Fora do Lugar

- Conta peças em posições incorretas (incluindo 0)
- **Exemplo**:

$$h\begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 7 & 8 & 6 \end{pmatrix} = 2$$

Distância de Manhattan

- Soma distâncias horizontais/verticais até posições corretas
- **Exemplo**:

$$h\begin{pmatrix}1 & 2 & 3\\ 4 & 0 & 5\\ 7 & 8 & 6\end{pmatrix} = 2$$

4 COMPARAÇÃO DE DESEMPENHO

Table 1: Resultados Comparativos

Algoritmo	Nós Visitados	Tempo (ms)	Passos
BFS	13	0.07	2
DFS	3	0.01	2
A* (Misplaced)	3	0.04	2
A* (Manhattan)	3	0.04	2

5 ANÁLISE DOS RESULTADOS

Comparação entre Algoritmos

- **BFS**: Garantiu solução ótima, mas visitou 4x mais nós que os outros métodos
- **DFS**: Surpreendeu com desempenho equivalente ao A* neste caso específico
- **A***: Mostrou eficiência superior com ambas heurísticas

Comparação entre Heurísticas

Table 2: Eficiência das Heurísticas

Métrica	Misplaced	Manhattan
Custo Computacional	Baixo	Médio
Precisão	Moderada	Alta
Nós Visitados	3	3

6 CONCLUSÃO

- **Melhor Método**: A* com Distância de Manhattan
  - Tempo: 0.04 ms
  - Nós Visitados: 3
  - Garantia de solução ótima
- **Recomendações**:
  - Usar Manhattan para problemas complexos
  - Evitar DFS em espaços de estado grandes

7 IMPLEMENTAÇÃO DO PROJETO

O código completo do projeto foi desenvolvido em Python 3 e está organizado em três componentes principais: algoritmos de busca, heurísticas e interface de teste. Segue a implementação detalhada:

Estruturas Básicas

Listing 1: Funções Auxiliares

```python
def get_blank_pos(state):
    """Encontra a posicao do espaco vazio (0)"""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)
    return None

def generate_moves(state):
    """Gera todos movimentos validos a partir de um estado"""
    blank_i, blank_j = get_blank_pos(state)
    moves = []
    dirs = [('up', -1, 0), ('down', 1, 0),
            ('left', 0, -1), ('right', 0, 1)
                ]

    for action, di, dj in dirs:
        new_i, new_j = blank_i + di, blank_j
             + dj
        if 0 <= new_i < 3 and 0 <= new_j <
            3:
            new_state = [list(row) for row
                in state]
            # Realiza a troca
            new_state[blank_i][blank_j],
                new_state[new_i][new_j] =
                new_state[new_i][new_j],
                    new_state[blank_i][
                    blank_j]
            new_state = tuple(map(tuple,
                new_state))
            moves.append((new_state, action)
                )
    return moves
```

Algoritmos de Busca

Listing 2: Implementacao BFS/DFS

```python
from collections import deque

def bfs(initial, goal):
    initial_tuple = tuple(map(tuple, initial
        ))
    goal_tuple = tuple(map(tuple, goal))
    visited = set()
    queue = deque([(initial_tuple, [])])

    while queue:
        current, path = queue.popleft()
        if current == goal_tuple:
```

```
            return path
        if current in visited:
            continue
        visited.add(current)
        for next_state, move in
            generate_moves(current):
            queue.append((next_state, path +
                [move]))
    return None

def dfs(initial, goal):
    initial_tuple = tuple(map(tuple, initial
        ))
    goal_tuple = tuple(map(tuple, goal))
    visited = set()
    stack = [(initial_tuple, [])]

    while stack:
        current, path = stack.pop()
        if current == goal_tuple:
            return path
        if current in visited:
            continue
        visited.add(current)
        for next_state, move in reversed(
            generate_moves(current)):
            stack.append((next_state, path +
                [move]))
    return None
```

Algoritmo A* e Heurísticas

Listing 3: Implementacao A*

```
import heapq

def a_star(initial, goal, heuristic):
    initial_tuple = tuple(map(tuple, initial
        ))
    goal_tuple = tuple(map(tuple, goal))
    open_heap = []
    heapq.heappush(open_heap, (0, 0,
        initial_tuple, []))
    closed = dict()

    while open_heap:
        current_f, current_g, current_state,
             path = heapq.heappop(open_heap)
        if current_state == goal_tuple:
            return path
        if current_state in closed and
            closed[current_state] <=
            current_g:
            continue
        closed[current_state] = current_g
        for next_state, move in
            generate_moves(current_state):
            next_g = current_g + 1
            next_h = heuristic(next_state,
                goal)
            next_f = next_g + next_h
            if next_state not in closed or
                next_g < closed.get(
                next_state, float('inf')):
                heapq.heappush(open_heap, (
                    next_f, next_g,
                    next_state, path + [move
                    ]))
    return None

# Heuristicas
def misplaced_tiles(state, goal):
    return sum(1 for i in range(3) for j in
        range(3)
            if state[i][j] != goal[i][j])

def manhattan_distance(state, goal):
    goal_pos = {goal[i][j]: (i,j) for i in
        range(3) for j in range(3)}
    return sum(abs(i - goal_pos[tile][0]) +
        abs(j - goal_pos[tile][1])
            for i in range(3) for j in range
                (3) if (tile := state[i][j])
                != 0)
```

Interface de Teste

Listing 4: Testes e Metricas

```
def run_algorithm(algorithm, heuristic=None)
    :
    initial = [[1,2,3],[4,0,5],[7,8,6]]
    goal = [[1,2,3],[4,5,6],[7,8,0]]

    start = time.time()
    path = None
    if algorithm == 'BFS':
        path = bfs(initial, goal)
    elif algorithm == 'DFS':
        path = dfs(initial, goal)
    elif algorithm == 'A*':
        path = a_star(initial, goal,
            heuristic)

    return {
        'time': time.time() - start,
        'path_length': len(path) if path
            else None,
        'path': path
    }
```

Notas de Implementação:

- O código usa tuplas para representar estados e garantir imutabilidade
- A conversão para tuplas permite usar estados como chaves de dicionário
- A interface de teste mede tempo de execução e comprimento do caminho

8 LINK PARA EXECUTÁVEL

https://drive.google.com/file/d/1J8dN8IK7b0A_W9EqKHmc0gKTjLwPBFV
view?usp=drive_link