Resolução do 8-Puzzle com Métodos de Busca: BFS, DFS e A*

Luan Barbosa Rosa Carrieiros
PUC Minas
Belo Horizonte, MG, Brasil
luan.rosa@sga.pucminas.br

ABSTRACT

Este artigo apresenta a implementação do jogo 8-Puzzle utilizando três métodos de busca: busca em largura (BFS), busca em profundidade (DFS) e o algoritmo A* com duas heurísticas diferentes. A interface gráfica em Tkinter foi adicionada para visualização interativa. O objetivo é comparar o desempenho entre os algoritmos quanto à eficiência, tempo de execução, profundidade da solução e número de nós visitados.

KEYWORDS

Inteligência Artificial, 8-puzzle, Busca em Largura, Busca em Profundidade, A*, Heurísticas, Tkinter

ACM Reference Format:
Luan Barbosa Rosa Carrieiros. 2025. Resolução do 8-Puzzle com Métodos de Busca: BFS, DFS e A*. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

O 8-Puzzle é um clássico problema de busca em Inteligência Artificial. Neste trabalho, implementamos três algoritmos de solução (BFS, DFS e A*) e adicionamos uma interface em Tkinter para visualização e controle interativos.

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

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

2 MÉTODOS DE BUSCA E FUNÇÕES AUXILIARES

2.1 Geração de Vizinhos

```python
def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)
    moves = [
        ("Cima", index-3, row>0),
        ("Baixo", index+3, row<2),
        ("Esquerda", index-1, col>0),
        ("Direita", index+1, col<2)
    ]
    for direction,new_idx,valid in moves:
        if valid:
            new_state = list(state)
            new_state[index], new_state[new_idx] =
                new_state[new_idx], new_state[index]
            neighbors.append((new_state,direction))
    return neighbors
```

Listing 1: Função get_neighbors

2.2 Heurísticas

```python
def manhattan(state):
    dist = 0
    for i,val in enumerate(state):
        if val==0: continue
        goal_r,goal_c = divmod(val-1,3)
        r,c = divmod(i,3)
        dist += abs(goal_r-r)+abs(goal_c-c)
    return dist
```

Listing 2: Heurística Manhattan

```python
def heuristic_misplaced(state):
    goal = list(range(1,9))+[0]
    return sum(1 for i,val in enumerate(state)
               if val!=0 and val!=goal[i])
```

Listing 3: Heurística Peças Fora do Lugar

2.3 Busca em Largura (BFS)

```python
def bfs(start):
    goal=list(range(1,9))+[0]
    queue=deque([(start,[])])
    visited=set()
    while queue:
        state,path=queue.popleft()
        if state==goal:
            return path
        visited.add(tuple(state))
        for nbr,dir in get_neighbors(state):
            if tuple(nbr) not in visited:
                queue.append((nbr,path+[dir]))
    return []
```

Listing 4: Implementação BFS

2.4 Busca em Profundidade Limitada (DFS)

```python
def dfs_limited(start,limit=30):
    goal=list(range(1,9))+[0]; visited=set()
    def rec(s,p,d):
        if s==goal: return p
        if d>=limit: return None
        visited.add(tuple(s))
        for nbr,dir in get_neighbors(s):
            t=tuple(nbr)
            if t not in visited:
                res=rec(nbr,p+[dir],d+1)
                if res is not None: return res
        return None
    return rec(start,[],0) or []
def dfs(start): return dfs_limited(start)
```

Listing 5: Implementação DFS com limite

2.5 Busca A*

```python
def astar_manhattan(start):
    goal=list(range(1,9))+[0]
    heap=[(manhattan(start),0,start,[])]
    visited=set()
    while heap:
        _,g,s,path=heapq.heappop(heap)
        if s==goal: return path
        visited.add(tuple(s))
        for nbr,dir in get_neighbors(s):
            if tuple(nbr) not in visited:
                heapq.heappush(heap,(g+1+manhattan(nbr),g
                    +1,nbr,path+[dir]))
    return []
```

Listing 6: Implementação A* (Manhattan)

```python
def astar_misplaced(start):
    goal=list(range(1,9))+[0]
    heap=[(heuristic_misplaced(start),0,start,[])]
    visited=set()
    while heap:
        _,g,s,path=heapq.heappop(heap)
        if s==goal: return path
        visited.add(tuple(s))
        for nbr,dir in get_neighbors(s):
            if tuple(nbr) not in visited:
                heapq.heappush(heap,(g+1+
                    heuristic_misplaced(nbr),g+1,nbr,
                    path+[dir]))
    return []
```

Listing 7: Implementação A* (Peças Fora)

3 INTERFACE GRÁFICA (TKINTER)

```python
# (ver arquivo puzzle.py)
if __name__=='__main__':
    root=tk.Tk()
    app=PuzzleApp(root)
    root.mainloop()
```

Listing 8: Código da interface

Interface completa disponível em `puzzle.py`.

4 COMPARATIVO DE DESEMPENHO

Table 1: Desempenho dos Algoritmos

Método	Tempo (s)	Nós Visitados	Movimentos	Heurística
BFS	0.38	60 914	21	—
DFS limitado	—	—	—	—
A* (Manhattan)	0.0012	133	21	Manhattan
A* (Peças Fora)	0.1062	4 822	21	Fora do lugar

5 CONCLUSÃO

A análise comparativa demonstrou que o algoritmo A* com heurística de Manhattan apresentou o melhor desempenho geral, com menor tempo e menor número de nós visitados. A heurística de peças fora do lugar também teve desempenho satisfatório. A BFS,

embora completa, foi significativamente mais lenta. A DFS não conseguiu encontrar solução no limite de profundidade adotado. Conclui-se que heurísticas eficazes tornam o A* a melhor abordagem para resolver o 8-Puzzle.

6 CÓDIGO DESENVOLVIDO

Link para o código e executável: https://github.com/LuanCarrieiros/IA/tree/main/Lista%20Extra