UNIVERSIDADE FEDERAL DE SANTA CATARINA
Departamento de Informática e Estatística - INE
Sistemas de Informação

INE5633 - Sistemas Inteligentes
Disciplina

Elder Rizzon Santos
Professor

Trabalho sobre Métodos de busca (2025/2)

Atividade Prática 1

Bruno Rafael Leal Machado
Diogo Henrique Fragoso de Oliveira
José Antonio de Oliveira
Alunos

24 de setembro de 2025

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A resolução de problemas por *busca* ocupa posição central na Inteligência Artificial (IA) simbólica: modela a navegação em um *espaço de estados* por meio de operadores, avaliando custos e selecionando expansões segundo políticas informadas ou não informadas, como discutem Russell e Norvig (2010) e Nilsson (1998). O interesse central está em compreender como diferentes algoritmos percorrem esse espaço, quais estruturas de dados utilizam e de que modo heurísticas afetam o desempenho.

Para fins experimentais, optou-se pelo *8-puzzle* apenas como domínio de teste. Trata-se de um tabuleiro 3×3 com oito peças móveis e um espaço vazio, no qual o objetivo é atingir uma configuração final a partir de um arranjo inicial. A escolha desse problema não se deve à sua complexidade prática, mas sim ao fato de ser computacionalmente leve, permitindo implementar e comparar variações de algoritmos de busca de forma controlada e reprodutível. Assim, o foco da análise permanece nos métodos de busca — custo uniforme e versões do algoritmo A^* com heurísticas de diferentes níveis de admissibilidade — e não no quebra-cabeça em si. Nesse sentido, o 8-puzzle funciona como um *laboratório didático* que viabiliza a avaliação sistemática de desempenho.

No contexto da disciplina INE5633—*Sistemas Inteligentes* (UFSC), esta Atividade Prática 1 (AP1) utiliza o 8-puzzle como laboratório para implementar e analisar o algoritmo A* e variações, em alinhamento ao conteúdo de *raciocínio e resolução de problemas* e à ênfase em técnicas de procura e informação heurística; a proposta didática privilegia implementação e análise prática, em consonância com a bibliografia básica adotada (RUSSELL; NORVIG, 2010; LUGER, 2009).

1.1 Escopo desta Atividade Prática teste

Serão estudadas quatro variantes: (i) busca de custo uniforme (sem heurística), (ii) A* com heurística *não admissível*, (iii) A* com heurística admissível simples e (iv) A* com a heurística admissível mais precisa desenvolvida pela equipe. A comparação considerará:

total de nós visitados, comprimento do caminho-solução, maior tamanho da fronteira (abertos), tempo de execução e um arquivo `.txt/.json` com fronteira e visitados ao término. Esses indicadores permitem discutir *admissibilidade* e *consistência* das heurísticas no A*, além de seus efeitos de eficiência (RUSSELL; NORVIG, 2010; LUGER, 2009).

■ 1.2 Objetivos formativos

Consolidar, por meio de implementação e experimentação reprodutível, a ponte entre teoria e prática em busca. Em particular, pretende-se:

(a) formalizar o problema como *espaço de estados* (definição de estados, operadores, teste de objetivo e função de custo);

(b) projetar e gerir a *fronteira* com checagem de dominância e de estados repetidos (políticas de inserção/remoção e estrutura de dados apropriada);

(c) definir e justificar heurísticas (admissíveis e não admissíveis), discutindo propriedades como admissibilidade e consistência;

(d) analisar comparativamente o desempenho das variantes (UCS e A*), com métricas reprodutíveis e interpretação crítica.

A fundamentação teórica apoia-se em Russell e Norvig (2010), Nilsson (1998) e obras complementares como Ertel (2017).

■ 1.3 Contribuições esperadas

O relatório apresentará:

(i) a modelagem do 8-puzzle e as estruturas de dados utilizadas;

(ii) o A* e o UCS com suas políticas de fronteira e critérios de expansão;

(iii) o desenho das heurísticas, com justificativa matemática e discussão sobre admissibilidade/consistência;

(iv) a avaliação experimental (casos fáceis, médios e difíceis), com comparação de nós visitados, comprimento do caminho-solução, maior tamanho da fronteira, tempo de execução e arquivo `.txt/.json` contendo fronteira e visitados ao término.

A análise será fundamentada em literatura clássica (RUSSELL; NORVIG, 2010; LUGER, 2009; NILSSON, 1998; ERTEL, 2017) e será reprodutível a partir dos artefatos entregues (código e logs).

2

MÉTODOS DE BUSCA

2.1 Definição e utilitários do tabuleiro

O *8-puzzle* é representado por uma matriz 3×3 cujas peças numeradas ocupam oito posições e um espaço vazio ocupa a nona. Usaremos o valor **9** para representar esse espaço. Para facilitar comparações e hashing em estruturas de dados (e.g., dicionários e conjuntos), convertemos a representação matricial em tupla linear e vice-versa.

```python
# Dimensões e estado objetivo (9 = espaço vazio)
LINHAS, COLS = 3, 3
TABULEIRO_OBJETIVO = [[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]]

def tabuleiro_para_tupla(tabuleiro):
    """Converte lista de listas (3x3) em tupla linear de tamanho 9."""
    return tuple(tabuleiro[i][j] for i in range(LINHAS) for j in range(COLS))

def tupla_para_tabuleiro(t):
    """Converte tupla linear (9) em lista de listas (3x3)."""
    return [list(t[i*COLS:(i+1)*COLS]) for i in range(LINHAS)]

def desenhar_tabuleiro(tabuleiro):
    """Imprime o tabuleiro linha a linha (útil para depuração)."""
    for linha in tabuleiro:
        print(linha)
```

Com isso, qualquer estado é uma tupla de nove inteiros, o que simplifica:

- *hashing* (armazenamento em `set`/`dict` para checagem de visitados);
- comparação por igualdade (detecção de estados repetidos);
- custo de cópia (tuplas são imutáveis e leves).

2.2 Funções de solubilidade

Nem toda permutação de peças é solucionável. Para o tabuleiro 3×3, uma configuração é solucionável se e somente se o número de *inversões* (pares fora de ordem, ignorando o vazio) é **par**. A seguir, implementamos a contagem de inversões e o teste de solubilidade.

```
def contagem_inversoes(tabuleiro):
    """Conta inversões (pares fora de ordem) ignorando o 9."""
    flat = [x for row in tabuleiro for x in row if x != 9]
    inv = 0
    for i in range(len(flat)):
        for j in range(i + 1, len(flat)):
            if flat[i] > flat[j]:
                inv += 1
    return inv

def eh_soluvel(tabuleiro):
    """Retorna True se o número de inversões é par (caso 3x3)."""
    return contagem_inversoes(tabuleiro) % 2 == 0
```

2.3 Gerador de tabuleiros aleatórios solucionáveis

Para gerar instâncias *seguramente solucionáveis*, partimos do objetivo e aplicamos uma sequência de movimentos válidos do espaço vazio. Assim, a solubilidade é preservada.

```
import random

def tabuleiro_aleatorio_soluvel(movimentos_embaralhar=40, seed=None):
    """Gera um tabuleiro aleatório solucionável a partir do objetivo."""
    if seed is not None:
        random.seed(seed)

    tabuleiro = [linha[:] for linha in TABULEIRO_OBJETIVO]
    t = tabuleiro_para_tupla(tabuleiro)

    def pos_branco(t):
        """Retorna (i, j) do espaço vazio (valor 9) em t."""
        idx = t.index(9)
        return divmod(idx, COLS)

    for _ in range(movimentos_embaralhar):
        i, j = pos_branco(t)
        movimentos = []
        if i > 0:          movimentos.append((-1, 0))
```

```
        if i < LINHAS - 1:  movimentos.append(( 1, 0))
        if j > 0:           movimentos.append(( 0,-1))
        if j < COLS - 1:    movimentos.append(( 0, 1))
        di, dj = random.choice(movimentos)
        ni, nj = i + di, j + dj

        lst = list(t)
        idx1, idx2 = i*COLS + j, ni*COLS + nj
        lst[idx1], lst[idx2] = lst[idx2], lst[idx1]
        t = tuple(lst)

    return tupla_para_tabuleiro(t)
```

2.4 Definição das estruturas de dados (nó)

Utilizamos uma classe `No` para encapsular: estado (tupla), ponteiro para o pai (para reconstrução de solução) e custo acumulado. O método `expandir` gera sucessores ao mover o vazio nas quatro direções válidas.

```
from dataclasses import dataclass, field

@dataclass(order=True)
class NoPriorizado:
    """Pacote (f, tie-breaker, nó) para a fila de prioridade."""
    prioridade: int
    contador: int
    no: object = field(compare=False)

@dataclass
class No:
    tabuleiro: tuple
    pai: object = None
    custo: int = 0

    def pos_branco(self):
        idx = self.tabuleiro.index(9)
        return divmod(idx, COLS)

    def expandir(self):
        i, j = self.pos_branco()
        movimentos = [(-1,0), (1,0), (0,-1), (0,1)]
        filhos = []
        for di, dj in movimentos:
            ni, nj = i + di, j + dj
            if 0 <= ni < LINHAS and 0 <= nj < COLS:
```

```
                nova_lista = list(self.tabuleiro)
                idx1 = i*COLS + j
                idx2 = ni*COLS + nj
                nova_lista[idx1], nova_lista[idx2] = nova_lista[idx2], nova_lista[idx1]
                filhos.append(No(tuple(nova_lista), pai=self, custo=self.custo + 1))
        return filhos
```

2.5 Heurísticas

Para o A^*, consideramos duas heurísticas clássicas admissíveis:

(a) **Peças fora do lugar** (*misplaced tiles*): número de peças que não estão em sua posição objetivo;

(b) **Distância de Manhattan**: soma, para cada peça, das distâncias de Manhattan entre a posição atual e a posição objetivo.

A seguir, implementações em tempo linear no tamanho do estado.

```
def objetivo_tupla():
    return tabuleiro_para_tupla(TABULEIRO_OBJETIVO)

_GOAL = objetivo_tupla()
# Mapa: valor -> (linha, coluna) no objetivo (ignora o 9)
GOAL_POS = {v: divmod(i, COLS) for i, v in enumerate(_GOAL) if v != 9}

def distancia_manhattan(t):
    """Soma das distâncias de Manhattan peça-a-peça (ignora o 9)."""
    dist = 0
    for idx, val in enumerate(t):
        if val == 9:
            continue
        i, j = divmod(idx, COLS)
        gi, gj = GOAL_POS[val]
        dist += abs(i - gi) + abs(j - gj)
    return dist

def pecas_erradas(t):
    """Conta peças fora da posição objetivo (ignora o 9)."""
    return sum(1 for i, val in enumerate(t) if val != 9 and val != _GOAL[i])
```

2.6 Reconstrução do caminho

```
def reconstruir_caminho(no):
```

```python
    """Caminho do nó inicial até o nó 'no' (inclusive)."""
    caminho = []
    while no:
        caminho.append(no.tabuleiro)
        no = no.pai
    return list(reversed(caminho))
```

2.7 Algoritmo de busca (UCS e A^*)

A busca de Custo Uniforme (UCS) é um caso particular do A^* com heurística nula ($h \equiv 0$). Em ambos, mantemos a *fronteira* em uma fila de prioridade ordenada por $f(n) = g(n) + h(n)$ e um mapa `custo_ate` para dominância. Estados visitados são marcados em `fechados`.

```python
import heapq

def busca(tabuleiro_inicial, algoritmo="ucs", heuristica="manhattan",
          limite_expansoes=None):
    """Executa UCS ou A* e retorna um dicionário com métricas e resultado."""
    t_inicial = tabuleiro_para_tupla(tabuleiro_inicial)
    t_objetivo = objetivo_tupla()

    if algoritmo not in ("ucs", "astar"):
        raise ValueError("algoritmo deve ser 'ucs' ou 'astar'")

    if algoritmo == "astar":
        if   heuristica == "manhattan":     hfun = distancia_manhattan
        elif heuristica == "pecas_erradas": hfun = pecas_erradas
        else: raise ValueError("heuristica deve ser 'manhattan' ou 'pecas_erradas'")
    else:
        hfun = lambda _: 0   # UCS

    fronteira = []
    contador = 0
    no_inicial = No(t_inicial, pai=None, custo=0)
    heapq.heappush(fronteira, (hfun(t_inicial), contador, no_inicial))
    contador += 1

    custo_ate = {t_inicial: 0}
    fechados = set()
    expandidos = 0

    while fronteira:
        _, _, atual = heapq.heappop(fronteira)

```

```
        if atual.tabuleiro == t_objetivo:
            caminho = reconstruir_caminho(atual)
            return {
                "encontrado": True,
                "movimentos": len(caminho) - 1,
                "custo": atual.custo,
                "caminho": caminho,
                "tamanho_fechados": len(fechados),
                "tamanho_fronteira": len(fronteira),
                "expandidos": expandidos,
                "algoritmo": algoritmo,
                "heuristica": heuristica if algoritmo == "astar" else None
            }

        if atual.tabuleiro in fechados:
            continue
        fechados.add(atual.tabuleiro)

        expandidos += 1
        if limite_expansoes is not None and expandidos >= limite_expansoes:
            return {
                "encontrado": False,
                "motivo": f"limite_expansoes={limite_expansoes} atingido",
                "tamanho_fechados": len(fechados),
                "tamanho_fronteira": len(fronteira),
                "expandidos": expandidos,
                "algoritmo": algoritmo,
                "heuristica": heuristica if algoritmo == "astar" else None
            }

        for filho in atual.expandir():
            g = filho.custo
            if (filho.tabuleiro not in custo_ate) or (g < custo_ate[filho.tabuleiro]):
                custo_ate[filho.tabuleiro] = g
                f = g + hfun(filho.tabuleiro)
                heapq.heappush(fronteira, (f, contador, filho))
                contador += 1
```

2.8 Impressão do caminho solução

```
def mostrar_caminho_solucao(caminho):
    """Imprime o caminho em formato 3x3 por passo."""
    for passo, t in enumerate(caminho):
        print(f"Passo {passo}:")
        for r in range(LINHAS):
            print(list(t[r*COLS:(r+1)*COLS]))
```

```
        print()
```

2.9 Exemplo de execução e saída

O trecho a seguir embaralha uma instância solucionável e executa UCS e A^* com Manhattan, reportando as métricas solicitadas.

```
if __name__ == "__main__":
    demo = tabuleiro_aleatorio_soluvel(movimentos_embaralhar=30, seed=39)
    print("Tabuleiro inicial:")
    desenhar_tabuleiro(demo)
    print("\nSolucionável?", eh_soluvel(demo))

    print("\n== UCS ==")
    r1 = busca(demo, algoritmo="ucs")
    print({k: v for k, v in r1.items() if k != "caminho"})

    print("\n== A* (Manhattan) ==")
    r2 = busca(demo, algoritmo="astar", heuristica="manhattan")
    print({k: v for k, v in r2.items() if k != "caminho"})

    if r2.get("encontrado"):
        print("\nPrimeiros 5 passos (A*):")
        mostrar_caminho_solucao(r2["caminho"][:5])
```

3 CONCLUSÃO

4 CONCLUSÃO

REFERÊNCIAS BIBLIOGRÁFICAS

ERTEL, W. *Introduction to Artificial Intelligence*. 2. ed. Cham: Springer, 2017.

LUGER, G. F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 6. ed. [S.l.]: Pearson, 2009.

NILSSON, N. J. *Artificial Intelligence: A New Synthesis*. San Francisco: Morgan Kaufmann, 1998.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. ed. [S.l.]: Prentice Hall, 2010.