Disciplina: Inteligência Artificial

Professora: Cristiane Neri Nobre

Data de entrega: 04/05

Valor: 5 pontos

Aluno: Lucas Henrique Rocha Hauck

Github com o código: https://github.com/o-hauck/IA

Relatório: Implementação de Jogo Puzzle com Algoritmos de Busca

Introdução

Este relatório descreve a implementação do jogo de puzzle 8 (ou 3x3) utilizando três métodos de busca diferentes: **Busca em Largura (BFS)**, **Busca Gulosa** e **Algoritmo A* (com duas heurísticas diferentes)**. O objetivo do trabalho foi comparar o desempenho desses algoritmos na solução do problema, com foco no número de movimentos, tempo de execução e número de nós visitados. A heurística A* foi experimentada com duas variantes: **Manhattan** e **Mal Pos**.

Descrição dos Métodos de Busca

1. Busca em Largura (BFS)

A busca em largura explora os nós do jogo de forma iterativa, começando pelas camadas mais rasas e expandindo até atingir a solução. Este algoritmo é completo, ou seja, sempre encontrará a solução, se ela existir, mas pode ser ineficiente em termos de tempo e memória, pois expande todos os nós à medida que percorre as camadas.

- **Complexidade**: A BFS tem uma complexidade de tempo e espaço proporcional ao número de nós no estado do espaço de busca.
- **Vantagem**: Garante a solução ótima em termos de número de movimentos.
- **Desvantagem**: Consome muita memória e tempo, especialmente em estados com muitas possibilidades.

2. Busca Gulosa

A busca gulosa utiliza uma heurística para escolher qual nó expandir com base na estimativa de quão próximo o nó está da solução. No entanto, não leva em consideração o custo acumulado, o que pode resultar em soluções subótimas. Este método é muito rápido, mas nem sempre encontra o caminho mais curto.

- **Complexidade**: Pode ser mais rápida que a BFS, mas depende fortemente da qualidade da heurística utilizada.
- **Vantagem**: Rápido em termos de tempo, especialmente quando a heurística é boa.
- **Desvantagem**: Pode ser ineficiente, pois não garante que encontrará a solução ótima.

3. Algoritmo A* (com Heurísticas Manhattan e Mal Pos)

O A* é um algoritmo de busca ótimo que combina o melhor dos dois mundos: ele leva em consideração tanto o custo acumulado quanto a estimativa de distância até o objetivo, usando uma função de avaliação f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n), onde g(n)g(n)g(n) é o custo do caminho até o nó e h(n)h(n)h(n) é a heurística.

- **Heurística Manhattan**: A heurística de Manhattan calcula a distância em termos do número de movimentos horizontais e verticais necessários para mover de um ponto a outro.
- **Heurística Mal Pos**: A heurística Mal Pos calcula o número de peças fora de lugar, ou seja, o número de peças que não estão na posição correta.

Descrição dos Resultados

A seguir, são apresentados os resultados dos testes realizados para cada algoritmo, com as respectivas heurísticas:

Algoritmo: A* Manhattan

- **Estado inicial**: [2, 6, 4, 3, 5, 1, 8, 0, 7]
- **Movimentos**: 27
- **Tempo**: 0.0376 segundos
- **Caminho**: Up -> Up -> Right -> Down -> Left -> Left -> Down -> Right -> Right -> Up -> Left -> Left -> Down -> Right -> Up -> Left -> Up -> Right -> Right -> Down -> Left -> Left -> Up -> Right -> Down -> Right -> Down

Algoritmo: A* Mal Pos

- **Estado inicial**: [2, 5, 4, 1, 0, 3, 8, 6, 7]
- **Movimentos**: 20
- **Tempo**: 0.0198 segundos
- **Caminho**: Right -> Up -> Left -> Down -> Right -> Down -> Left -> Up -> Right -> Up -> Left -> Left -> Down -> Right -> Down -> Left -> Up -> Right -> Right -> Down

Algoritmo: BFS

- **Estado inicial**: [1, 4, 5, 2, 6, 8, 7, 3, 0]
- **Movimentos**: 22
- **Tempo**: 0.2782 segundos
- **Caminho**: Up -> Left -> Up -> Left -> Down -> Right -> Down -> Left -> Up -> Up -> Right -> Down -> Left -> Down -> Right -> Right -> Up -> Up -> Left -> Down -> Right -> Down

Algoritmo: Busca Gulosa

- **Estado inicial**: [0, 8, 7, 5, 4, 1, 3, 6, 2]
- **Movimentos**: 56
- **Tempo**: 0.0030 segundos
- **Caminho**: Down -> Right -> Up -> Right -> Down -> Down -> Left -> Up -> Up -> Right -> Down -> Down -> Left -> Up -> Up -> Left -> Down -> Right -> Up -> Left -> Down -> Down -> Right -> Right -> Up -> Left -> Up -> Left -> Down -> Right -> Up -> Left -> Down -> Right -> Right ->

Up -> Left -> Left -> Down -> Right -> Right -> Up -> Left -> Down -> Left -> Up -> Right -> Right -> Down -> Left -> Up -> Left -> Down -> Right -> Right -> Down

Análise e Comparação dos Algoritmos

- **Desempenho em Tempo**:
    - O algoritmo A* com a heurística Mal Pos apresentou o melhor desempenho em termos de tempo (0.0198 segundos), seguido pelo A* Manhattan (0.0376 segundos), a Busca em Largura (0.2782 segundos) e, por fim, a Busca Gulosa (0.0030 segundos).
    - A Busca Gulosa, apesar de ser extremamente rápida, não garantiu uma solução ótima e resultou em muitos movimentos.
- **Desempenho em Movimentos**:
    - O algoritmo A* Mal Pos foi o que obteve a menor quantidade de movimentos (20), seguido pelo A* Manhattan (27), BFS (22) e a Busca Gulosa (56).
    - O A* Mal Pos se destacou ao fornecer uma solução mais eficiente em termos de número de movimentos.
- **Exploração de Nós**:
    - O número de nós visitados não foi diretamente reportado, mas podemos observar que a BFS, sendo uma busca cega, tende a explorar mais nós do que os algoritmos heurísticos, como A* e Busca Gulosa.

Conclusão

O algoritmo **A* com a heurística Mal Pos** se mostrou superior tanto em termos de tempo quanto de número de movimentos, proporcionando uma solução ótima mais rapidamente. A heurística Manhattan, apesar de ser eficaz, não foi tão eficiente quanto a Mal Pos para este caso específico. A **Busca em Largura**, embora garantisse uma solução ótima, foi significativamente mais lenta e explorou mais nós. A **Busca Gulosa**, embora muito rápida, não encontrou a solução ótima e resultou em um número de movimentos muito maior.

Este trabalho demonstra a importância da escolha da heurística no desempenho de algoritmos de busca, especialmente no contexto de jogos de puzzle, onde a eficiência e a otimização são cruciais.

Código em python:

```
import tkinter as tk

from tkinter import messagebox

import random

import heapq

from collections import deque

import time
```

```
# --------- Classe do Jogo ---------
class Puzzle8:
    def __init__(self):
        self.estado_objetivo = list(range(1, 9)) + [0]
        self.estado_atual = self.embaralhar_tabuleiro()

    def embaralhar_tabuleiro(self):
        estado = self.estado_objetivo[:]
        while True:
            random.shuffle(estado)
            if self.eh_solucionavel(estado) and estado != self.estado_objetivo:
                return estado

    def eh_solucionavel(self, estado):
        inversoes = 0
        for i in range(8):
            for j in range(i + 1, 9):
                if estado[i] and estado[j] and estado[i] > estado[j]:
                    inversoes += 1
        return inversoes % 2 == 0

    def mover(self, direcao):
        indice = self.estado_atual.index(0)
        linha, coluna = divmod(indice, 3)
        if direcao == "Up" and linha > 0:
            self.trocar(indice, indice - 3)
        elif direcao == "Down" and linha < 2:
            self.trocar(indice, indice + 3)
        elif direcao == "Left" and coluna > 0:
            self.trocar(indice, indice - 1)
        elif direcao == "Right" and coluna < 2:
            self.trocar(indice, indice + 1)
```

```
    def trocar(self, i, j):
        self.estado_atual[i], self.estado_atual[j] = self.estado_atual[j], self.estado_atual[i]

    def resolvido(self):
        return self.estado_atual == self.estado_objetivo

# --------- Heurísticas ---------

def heuristica_manhattan(estado):
    distancia = 0
    for i, val in enumerate(estado):
        if val == 0: continue
        linha_destino, col_destino = divmod(val - 1, 3)
        linha_atual, col_atual = divmod(i, 3)
        distancia += abs(linha_destino - linha_atual) + abs(col_destino - col_atual)
    return distancia

# Nova heurística: número de peças fora do lugar

def heuristica_mal_colocado(estado):
    return sum(1 for i in range(9) if estado[i] != 0 and estado[i] != i + 1)

# --------- Funções Auxiliares ---------

def gerar_vizinhos(estado):
    vizinhos = []
    indice = estado.index(0)
    linha, coluna = divmod(indice, 3)
    direcoes = [("Up", -3), ("Down", 3), ("Left", -1), ("Right", 1)]

    for direcao, delta in direcoes:
        novo_indice = indice + delta
        if direcao == "Up" and linha == 0: continue
```

```
        if direcao == "Down" and linha == 2: continue
        if direcao == "Left" and coluna == 0: continue
        if direcao == "Right" and coluna == 2: continue
        novo_estado = list(estado)
        novo_estado[indice], novo_estado[novo_indice] = novo_estado[novo_indice],
novo_estado[indice]
        vizinhos.append((novo_estado, direcao))
    return vizinhos


# --------- Algoritmos ---------


def busca_a_estrela(estado_inicial, heuristica):
    objetivo = list(range(1, 9)) + [0]
    fila = [(heuristica(estado_inicial), 0, estado_inicial, [])]
    visitados = set()
    while fila:
        _, custo, estado, caminho = heapq.heappop(fila)
        if tuple(estado) in visitados:
            continue
        visitados.add(tuple(estado))
        if estado == objetivo:
            return caminho
        for vizinho, direcao in gerar_vizinhos(estado):
            if tuple(vizinho) not in visitados:
                novo_custo = custo + 1
                estimativa = novo_custo + heuristica(vizinho)
                heapq.heappush(fila, (estimativa, novo_custo, vizinho, caminho +
[direcao]))
    return []


def busca_em_largura(estado_inicial):
    objetivo = list(range(1, 9)) + [0]
    fila = deque([(estado_inicial, [])])
    visitados = set()
```

```
    while fila:
        estado, caminho = fila.popleft()
        if estado == objetivo:
            return caminho
        visitados.add(tuple(estado))
        for vizinho, direcao in gerar_vizinhos(estado):
            if tuple(vizinho) not in visitados:
                fila.append((vizinho, caminho + [direcao]))
    return []

def busca_em_profundidade(estado_inicial, max_depth=50):
    objetivo = list(range(1, 9)) + [0]
    pilha = [(estado_inicial, [], 0)]  # A pilha agora inclui a profundidade
    visitados = set()

    while pilha:
        estado, caminho, profundidade = pilha.pop()

        # Limite de profundidade para evitar loops infinitos
        if profundidade > max_depth:
            continue

        if tuple(estado) in visitados:
            continue

        visitados.add(tuple(estado))

        if estado == objetivo:
            return caminho

        # Adiciona vizinhos à pilha com a profundidade aumentada
        vizinhos = gerar_vizinhos(estado)
        for vizinho, direcao in reversed(vizinhos):
```

```
            if tuple(vizinho) not in visitados:
                pilha.append((vizinho, caminho + [direcao], profundidade + 1))

    return []

def busca_gulosa(estado_inicial, heuristica):
    objetivo = list(range(1, 9)) + [0]
    fila = [(heuristica(estado_inicial), estado_inicial, [])]
    visitados = set()
    while fila:
        _, estado, caminho = heapq.heappop(fila)
        if tuple(estado) in visitados:
            continue
        visitados.add(tuple(estado))
        if estado == objetivo:
            return caminho
        for vizinho, direcao in gerar_vizinhos(estado):
            if tuple(vizinho) not in visitados:
                estimativa = heuristica(vizinho)
                heapq.heappush(fila, (estimativa, vizinho, caminho + [direcao]))
    return []

# --------- Interface Tkinter ---------
class InterfacePuzzle:
    def __init__(self, root):
        self.jogo = Puzzle8()
        self.root = root
        self.root.title("8-Puzzle com Inteligência Artificial")

        self.frame_tabuleiro = tk.Frame(root, padx=20, pady=20)
        self.frame_tabuleiro.grid(row=0, column=0, columnspan=3)

        self.botoes = []
```

```
        for i in range(9):
            botao = tk.Button(self.frame_tabuleiro, text="", font=("Helvetica", 24,
"bold"), width=4, height=2,
                              command=lambda i=i: self.clique_bloco(i))
            botao.grid(row=i//3, column=i%3, padx=5, pady=5)
            self.botoes.append(botao)

        self.frame_controles = tk.Frame(root, pady=10)
        self.frame_controles.grid(row=1, column=0, columnspan=3)

        self.btn_shuffle = tk.Button(root, text="Embaralhar", command=self.embaralhar)
        self.btn_shuffle.grid(row=2, column=0, pady=10)

        self.btn_astar = tk.Button(root, text="A* Manhattan",
command=self.resolver_astar_manhattan)
        self.btn_astar.grid(row=2, column=1, pady=10)

        self.btn_astar2 = tk.Button(root, text="A* Mal Pos",
command=self.resolver_astar_mal_colocado)
        self.btn_astar2.grid(row=2, column=2, pady=10)

        self.btn_bfs = tk.Button(root, text="BFS", command=self.resolver_bfs)
        self.btn_bfs.grid(row=3, column=0, pady=10)

        self.btn_gulosa = tk.Button(root, text="Gulosa", command=self.resolver_gulosa)
        self.btn_gulosa.grid(row=3, column=1, pady=10)

        self.atualizar_tabuleiro()

    def atualizar_tabuleiro(self):
        for i in range(9):
            num = self.jogo.estado_atual[i]
            self.botoes[i].config(text="" if num == 0 else str(num), bg="#e0e0e0" if
num == 0 else "#ffffff")
```

```
def clique_bloco(self, indice):
    branco = self.jogo.estado_atual.index(0)
    linha1, col1 = divmod(branco, 3)
    linha2, col2 = divmod(indice, 3)
    if abs(linha1 - linha2) + abs(col1 - col2) == 1:
        self.jogo.trocar(branco, indice)
        self.atualizar_tabuleiro()

def embaralhar(self):
    self.jogo.estado_atual = self.jogo.embaralhar_tabuleiro()
    self.atualizar_tabuleiro()

def salvar_resultado(self, nome_algoritmo, caminho, duracao):
    with open("resultado_resolucao.txt", "w") as f:
        f.write(f"Algoritmo: {nome_algoritmo}\n")
        f.write(f"Estado inicial: {self.jogo.estado_atual}\n")
        f.write(f"Movimentos: {len(caminho)}\n")
        f.write(f"Tempo: {duracao:.4f} segundos\n")
        f.write(f"Caminho: {' -> '.join(caminho)}\n")

def animar_solucao(self, solucao, duracao, nome_algoritmo):
    if not solucao:
        messagebox.showinfo("Erro", "Nenhuma solução encontrada.")
        return

    total = len(solucao)
    self.salvar_resultado(nome_algoritmo, solucao.copy(), duracao)

    def passo():
        if solucao:
            self.jogo.mover(solucao.pop(0))
            self.atualizar_tabuleiro()
            self.root.after(300, passo)
```

```
            else:
                messagebox.showinfo("Resolvido!", f"Movimentos: {total}\nTempo:
{duracao:.4f}s")

        passo()

    def resolver_astar_manhattan(self):
        inicio = time.time()
        caminho = busca_a_estrela(self.jogo.estado_atual[:], heuristica_manhattan)
        fim = time.time()
        self.animar_solucao(caminho, fim - inicio, "A* Manhattan")

    def resolver_astar_mal_colocado(self):
        inicio = time.time()
        caminho = busca_a_estrela(self.jogo.estado_atual[:], heuristica_mal_colocado)
        fim = time.time()
        self.animar_solucao(caminho, fim - inicio, "A* Mal Pos")

    def resolver_bfs(self):
        inicio = time.time()
        caminho = busca_em_largura(self.jogo.estado_atual[:])
        fim = time.time()
        self.animar_solucao(caminho, fim - inicio, "BFS")

    def resolver_gulosa(self):
        inicio = time.time()
        caminho = busca_gulosa(self.jogo.estado_atual[:], heuristica_manhattan)  #
Usando a heurística de Manhattan
        fim = time.time()
        self.animar_solucao(caminho, fim - inicio, "Busca Gulosa")

# --------- Executar Interface ---------
if __name__ == "__main__":
    root = tk.Tk()
```

```
app = InterfacePuzzle(root)
root.mainloop()
```