Implementação de uma solução para do jogo do 8 (tabuleiro 3x3)

Enzo L. de Aragão[1] , Jordana Bezerra França[1] , Luis Felipe do N. Moura[1], Lucas Jesus S. Silva[1]

[1]Departamento de Computação
Universidade Federal do Piauí (UFPI) – Teresina, PI – Brazil

{enzo.lda,jordanafranca,luquinhajssilva,felipemoura1407}@ufpi.edu.br

1. Introdução

Este relatório apresenta a implementação e a análise de quatro estratégias de busca aplicadas ao jogo do 8(8-puzzle), um quebra-cabeça clássico de movimentação de peças em um tabuleiro 3x3. As estratégias de busca incluem Força Bruta com busca cega (largura e profundidade), Busca Heurística utilizando Busca Gulosa com heurística euclidiana, e o algoritmo A* com a heurística de Manhattan.

O objetivo principal é oferecer uma análise do desempenho de diferentes métodos de busca aplicados ao jogo do 8. Ao longo do relatório, serão discutidos detalhes sobre a implementação das estratégias de busca, os resultados obtidos para obter a solução do jogo.

2. Implementação

2.1. Classe

Primeiramente, decidimos criar uma classe para cada nó da árvore com a finalidade de ter informações detalhadas sobre cada estado do jogo.

```
lass No:
    def __init__(self, estado, no_pai=None, acao=None, custoCaminho=0, profundidade=0, funcaoAvaliacao=0):
        self.estado = estado
        self.no_pai = no_pai
        self.acao = acao
        self.custoCaminho = custoCaminho
        self.profundidade = profundidade
        self.funcaoAvaliacao = funcaoAvaliacao
```

2.2. Funções utilitárias

2.2.1. Ações possíveis

Função utilizada para descobrir possíveis estados a partir de um estado inicial

```python
def acoesPossiveis(estado):
    # Retorna as ações possíveis no estado (movimentos da peça vazia)
    linhas, colunas = 3, 3
    posicao_vazia = estado.index(0)
    linha_vazia, coluna_vazia = divmod(posicao_vazia, colunas)

    acoes_possiveis = []

    # Verificar se é possível mover a peça para cima
    if linha_vazia > 0:
        acoes_possiveis.append(('cima', (linha_vazia - 1, coluna_vazia)))

    # Verificar se é possível mover a peça para baixo
    if linha_vazia < linhas - 1:
        acoes_possiveis.append(('baixo', (linha_vazia + 1, coluna_vazia)))

    # Verificar se é possível mover a peça para a esquerda
    if coluna_vazia > 0:
        acoes_possiveis.append(('esquerda', (linha_vazia, coluna_vazia - 1)))

    # Verificar se é possível mover a peça para a direita
    if coluna_vazia < colunas - 1:
        acoes_possiveis.append(('direita', (linha_vazia, coluna_vazia + 1)))

    return acoes_possiveis
```

2.2.2. Gerar filho

Função utilizada para a geração de novos estados a partir de ações de movimentação em um estado inicial

```python
def gerarFilho(estado, acao):
    novo_estado = estado[:]
    posicao_vazia = novo_estado.index(0)
    nova_posicao = acao[1][0] * 3 + acao[1][1]
    novo_estado[posicao_vazia], novo_estado[nova_posicao] = novo_estado[nova_posicao], novo_estado[posicao_vazia]
    return novo_estado
```

2.2.3. Teste de Objetivo

Função utilizada para verificar se o estado objetivo foi alcançado

```python
def teste_De_Objetivo(EstadoAtual, EstadoObjetivo):
    return EstadoAtual == EstadoObjetivo
```

2.2.4. Imprimir caminho da solução

Função utilizada para percorrer o caminho necessário para chegar do estado inicial até o estado objetivo

```python
def imprimirCaminhoSolucao(no_solucao):
    caminho = []
    while no_solucao:
        caminho.append(no_solucao.estado)
        no_solucao = no_solucao.no_pai
    caminho.reverse()
    for estado in caminho:
        print("Estado:")
        print(estado[0], estado[1], estado[2])
        print(estado[3], estado[4], estado[5])
        print(estado[6], estado[7], estado[8])
        print()
```

2.2.5. Imprimir estados na borda

Função utilizada para gerar um arquivo de saída(output.txt) com todos os estados que passaram pela borda

```python
def imprimirEstadosNaBorda(borda, arquivo_saida):
    with open(arquivo_saida, 'a') as arquivo:
        arquivo.write("Borda:\n")
        for elemento in borda:
            arquivo.write(str(elemento.estado) + "\n\n")
```

2.2.6. Quantidade de estados na borda

Função utilizada para retornar a quantidade de estados na borda

```python
def quantidadeDeEstadosNaBorda(borda):
    return len(borda)
```

2.2.7. Pegar índice randômico

Função utilizada para pegar um índice randômico em um intervalo pré-definido

```
def pegarNoRandomico(borda):
    return random.randint(0, 10)
```

2.3. Heurísticas

2.3.1. Distância de Manhathan

Heurística que utiliza a equação

$$h^2 = |x1-x2| + |y1-y2|$$

para calcular a distância entre a posição atual e a posição objetivo de um elemento do tabuleiro utilizando um mapeamento no tabuleiro em que cada posição possui um valor x e um valor y

```
def heuristicaDistanciaManhathan(elemento, estadoAtual, estadoObjetivo):
    # h² = |x1-x2| + |y1-y2|
    # x = coluna
    # y = linha

    # x0 x1 x2
    # 0  1  2     --> y = 0
    # 3  4  5     --> y = 1
    # 6  7  8     --> y = 2

    posicaoObjetivo = estadoObjetivo.index(elemento)
    posicaoAtual = estadoAtual.index(elemento)

    x1_mapping = {8: 2, 5: 2, 2: 2, 1: 1, 4: 1, 7: 1}
    y1_mapping = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2}

    x2_mapping = {8: 2, 5: 2, 2: 2, 1: 1, 4: 1, 7: 1}
    y2_mapping = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2}

    x1, y1 = x1_mapping.get(posicaoObjetivo, 0), y1_mapping.get(posicaoObjetivo, 0)
    x2, y2 = x2_mapping.get(posicaoAtual, 0), y2_mapping.get(posicaoAtual, 0)

    resultado = math.fabs(x1 - x2) + math.fabs(y1 - y2)
    return resultado
```

2.3.2. Distância Euclidiana

Heurística que utiliza a equação

$$h^2 = (x1 - x2)^2 + (y1 - y2)^2$$

para calcular a distância entre a posição atual e a posição objetivo de um elemento do tabuleiro utilizando um mapeamento no tabuleiro em que cada posição possui um valor x e um valor y

```python
def heuristicaDistanciaEuclidiana(elemento, estadoAtual, estadoObjetivo):
    # h² = (x1 - x2)² + (y1 - y2)²
    # x = coluna
    # y = linha

    # x0 x1 x2
    # 0  5  2     --> y = 0
    # 3  4  1     --> y = 1
    # 6  7  8     --> y = 2

    posicaoObjetivo = estadoObjetivo.index(elemento)
    posicaoAtual = estadoAtual.index(elemento)

    x1_mapping = {8: 2, 5: 2, 2: 2, 1: 1, 4: 1, 7: 1}
    y1_mapping = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2}

    x2_mapping = {8: 2, 5: 2, 2: 2, 1: 1, 4: 1, 7: 1}
    y2_mapping = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2}

    x1, y1 = x1_mapping.get(posicaoObjetivo, 0), y1_mapping.get(posicaoObjetivo, 0)
    x2, y2 = x2_mapping.get(posicaoAtual, 0), y2_mapping.get(posicaoAtual, 0)

    resultado = math.sqrt(math.pow(x1 - x2, 2) + math.pow(y1 - y2, 2))
    return resultado

def heuristicaQuantidadeDeElementosForaDoLugar(estadoAtual, estadoObjetivo):
    foraDoLugar = 0
    for elemento in estadoAtual:
        if(estadoAtual.index(elemento) != estadoObjetivo.index(elemento)):
            foraDoLugar += 1

    return foraDoLugar
```

2.3.3. Quantidades de elementos fora do lugar

Heurística que verifica quais elementos estão fora da posição objetivo

```python
def heuristicaQuantidadeDeElementosForaDoLugar(estadoAtual, estadoObjetivo):
    foraDoLugar = 0
    for elemento in estadoAtual:
        if(estadoAtual.index(elemento) != estadoObjetivo.index(elemento)):
            foraDoLugar += 1

    return foraDoLugar
```

2.4. Buscas Cegas

2.4.1. Em largura

```python
def buscaEmLargura(estado_inicial, estado_objetivo):
    borda = deque([No(estado_inicial)])
    explorados = set()
    passo = 0

    while borda:
        passo += 1
        #print(f"Passo {passo}:")

        no = borda.popleft()
        explorados.add(tuple(no.estado))

        if teste_De_Objetivo(no.estado, estado_objetivo):
            return no, passo, quantidadeDeEstadosNaBorda(borda)  # Retorna o nó solução

        for acao, posicao in acoesPossiveis(no.estado):
            filho_estado = gerarFilho(no.estado, (acao, posicao))
            if tuple(filho_estado) not in explorados:
                filho = No(filho_estado, no, acao)
                borda.append(filho)
                explorados.add(tuple(filho_estado))

        imprimirEstadosNaBorda(borda, 'output.txt')

    return None, None, None  # Se não encontrou solução
```

A função acima é utilizada para encontrar o caminho mais curto entre dois pontos. O algoritmo explora todos os nós, a partir do estado inicial, expandindo sistematicamente para os estados adjacentes em níveis sucessivos. Os nós a serem explorados são mantidos em uma fila e o nós já explorados são mantidos em um conjunto para evitar repetições, o algoritmo expande as opções de modo uniforme, de modo a garantir que os nós mais próximos do estado inicial sejam explorados primeiro.

2.4.2. Em profundidade

Tal método utiliza a ideia de percorrer um ramo até o limite de profundidade definido(33), ao chegar no limite é feita uma randomização para pegar um estado gerado em um intervalo de índice na variável borda que é definido pela função “pegarRandomico(borda)” para, assim, tentar evitar execuções muito longas a partir do investimento em um ramo que não chegue à solução em uma profundidade razoável

```python
def buscaEmProfundidade(estado_inicial, estado_objetivo, limite_profundidade=33):
    borda = [No(estado_inicial)]
    explorados = set()
    passo = 0
    nos_expandidos = 0

    while borda:
        nos_expandidos += 1
        passo += 1
        no_atual = borda.pop()
        explorados.add(tuple(no_atual.estado))

        if( no_atual.profundidade == limite_profundidade):
            indice_aleatorio = pegarNoRandomico(borda)
            no_aleatorio = borda[indice_aleatorio]
            no_aleatorio.profundidade = 0
            borda.clear()
            borda.append(no_aleatorio)
            explorados.clear()
            passo = 0
            print(no_aleatorio.estado)
            continue

        if teste_De_Objetivo(no_atual.estado, estado_objetivo):
            return no_atual, passo, quantidadeDeEstadosNaBorda(borda), nos_expandidos  # Retorna o nó solução

        for acao, posicao in acoesPossiveis(no_atual.estado):
            filho_estado = gerarFilho(no_atual.estado, (acao, posicao))
            if tuple(filho_estado) not in explorados:
                filho = No(filho_estado, no_atual, acao, profundidade=no_atual.profundidade + 1)
                borda.append(filho)
                explorados.add(tuple(filho_estado))

        imprimirEstadosNaBorda(borda, 'output.txt')

    return None, None, None, None  # Se não encontrou solução
```

2.5. Busca Gulosa

2.5.1. Escolha da heurística

Essa função é crucial, pois calcula o valor heurístico total para um estado atual em relação ao estado objetivo.

```python
def heuristicaGulosa(estadoAtual, estadoObjetivo):
    valor_heuristico_total = 0
    for elemento in estadoAtual:
        if elemento != 0:
           #valor_heuristico_total += heuristicaQuantidadeDeElementosForaDoLugar(estadoAtual, estadoObjetivo)
           #valor_heuristico_total += heuristicaDistanciaEuclidiana(elemento, estadoAtual, estadoObjetivo)
           valor_heuristico_total += heuristicaDistanciaManhathan(elemento, estadoAtual, estadoObjetivo)
    return valor_heuristico_total
```

2.5.2. Algoritmo

O algoritmo mantém uma lista de nós a serem explorados (borda) e um conjunto de estados já visitados. Em cada iteração, o algoritmo prioriza o nó com menor valor heurístico, determinado pela função ***heuristicaGulosa***, para ser explorado em seguida. Caso esse nó escolhido corresponda ao estado objetivo, o algoritmo retorna o nó solução, a profundidade alcançada e a quantidade de estados restantes na borda. Caso contrário, expande o nó selecionado, adicionando seus filhos válidos à borda para futura exploração.

```python
def buscaGulosa_algoritmo(estado_inicial, estado_objetivo):
    borda = [No(estado_inicial)]
    explorados = set()
    passo = 0

    while borda:
        passo += 1
        #print(f"Passo {passo}:")

        borda.sort(key=lambda x: heuristicaGulosa(x.estado, estado_objetivo))
# Ordena a borda pela heurística pelo menor custoTotal

        no = borda.pop(0)
        explorados.add(tuple(no.estado))

        if teste_De_Objetivo(no.estado, estado_objetivo):
            return no, passo, quantidadeDeEstadosNaBorda(borda)  # Retorna o nó solução

        for acao, posicao in acoesPossiveis(no.estado):
            filho_estado = gerarFilho(no.estado, (acao, posicao))
            if tuple(filho_estado) not in explorados:
                filho = No(filho_estado, no, acao)
                borda.append(filho)
                explorados.add(tuple(filho_estado))

        imprimirEstadosNaBorda(borda, 'output.txt')



    return None, None, None  # Se não encontrou solução
```

2.6. Busca A*

2.6.1. Função Avaliação

Função responsável por calcular o valor da heurística (custo do caminho(profundidade do nó) + distância de manhattan)

```python
def funcaoAvaliacao(estado, custoCaminho, estadoAtual, estadoObjetivo):
    # f(n)= g(n) + h(n)
    # A* = profundidade do nó + heuristica
    h = 0
    g = custoCaminho
    for indice in estado:
        h += heuristicaDistanciaManhathan(indice, estadoAtual, estadoObjetivo)
    return (g + h)
```

2.6.2. Algoritmo

O funcionamento da busca A* é semelhante ao da busca gulosa. A função *a_estrela(estado_inicial, estado_objetivo)* mantém uma lista de nós a serem explorados - será a borda, e um conjunto de estados explorados. Em cada iteração, o algoritmo ordena os nós da borda e seleciona aquele que tiver o menor valor da heurística. Fará um teste para saber se é o estado objetivo e caso não seja, ele expandirá esse nó, inserindo seus filhos na borda.

```python
def a_estrela(estado_inicial, estado_objetivo):
    borda = [No(estado_inicial)]
    explorados = set()
    passo = 0

    while borda:
        passo += 1
        #print(f"Passo {passo}:")
        borda.sort(key=lambda x: x.funcaoAvaliacao)  # Ordena a borda pela heurística pelo menor custoTotal
        no = borda.pop(0)
        explorados.add(tuple(no.estado))

        if teste_De_Objetivo(no.estado, estado_objetivo):
            return no, passo, quantidadeDeEstadosNaBorda(borda)  # Retorna o nó solução

        for acao, posicao in acoesPossiveis(no.estado):
            filho_estado = gerarFilho(no.estado, (acao, posicao))
            if tuple(filho_estado) not in explorados:
                filho = No(filho_estado, no, acao, custoCaminho=passo, profundidade=passo, funcaoAvaliacao = funcaoAvaliacao(
estado= filho_estado, custoCaminho= passo, estadoAtual= filho_estado, estadoObjetivo= estado_objetivo))
                borda.append(filho)
                explorados.add(tuple(filho_estado))

        imprimirEstadosNaBorda(borda, 'output.txt')

    return None, None, None  # Se não encontrou solução
```

3. Resultados

Para comparar os resultados entre os quatro métodos de buscas aplicados para a implementação do jogo foram aplicados dois estados iniciais diferentes:
Estado inicial 1 = [1, 2, 3,7, 4, 6, 0,5, 8] e Estado inicial 2 = [1, 3, 0, 4, 2, 5, 7,8, 6]

3.1 Estado inicial 1

Algoritmo	Profundidade	Quantidade de estados nas bordas	Quantidade de nós expandidos
Busca em Largura	21	17	21
Busca em Profundidade	31	25	31
Busca gulosa	5	5	5
Busca A*	5	5	5

3.2 Estado inicial 2

Algoritmo	Profundidade	Quantidade de estados nas bordas	Quantidade de nós expandidos
Busca em Largura	29	17	21
Busca em Profundidade	3	4	129308
Busca gulosa	5	5	5
Busca A*	7	6	7

3.3 Comparação entre as heurística

Após a comparação do custo de processamento a heurística escolhida foi a Manhattan, ela é uma heurística admissível, pois ela nunca superestimou o custo para atingir o estado final. A admissibilidade dessa heurística garante que o algoritmo A* encontrará a solução com o menor custo possível, desde que os movimentos tenham custos uniformes.

Heurística	Tempo de execução	Quantidade de estados	Utilização de memória (Pico)
Euclidiana	0.47s	25	52624 bytes
Manhattan	0.34s	4	50136 bytes
Quantidade de elementos fora do lugar	35.1s	5	440576 byte bytes

4. GUI

Para o projeto utilizamos a biblioteca tkinter que é uma biblioteca padrão do Python para criar interfaces gráficas do usuário (GUI). Seu uso foi de grande ajuda para a visualização do funcionamento do algoritmo e para a facilitar a interação com o código