UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS DE RUSSAS

LÓGICA PARA COMPUTAÇÃO

PROF: ALEXANDRE M. ARRUDA

TRABALHO DA DISCIPLINA: PROBLEMA 8-PUZZLE

Douglas Lima Menezes – 552590

Francisco Wandallis Pereira de Sousa – 553083

Milena de Santiago Maia Costa – 552981

Russas, CE

2025

1. DESCRIÇÃO DO PROBLEMA

O 8-puzzle é um quebra-cabeça deslizante que consiste em uma grade 3x3 com 8 peças numeradas (de 1 a 8) e um espaço vazio (representado por 0). O objetivo é reorganizar as peças, movendo-as para o espaço vazio, até que a ordem numérica seja alcançada na grade. As peças só podem ser movidas na horizontal ou vertical para o espaço vazio.

2. DESENVOLVIMENTO

Este trabalho apresenta uma solução para o problema descrito, cujo foi desenvolvido em Python e utilizado a biblioteca *py-sat* para modelar e resolver o puzzle em uma matriz 3x3 com números padrões [0, 1, 2, 3, 4, 5, 6, 7, 8].

O solver python-sat é específico para realização de problemas booleanos. Neste caso, queremos verificar se o movimento das peças gera um resultado satisfazível, ou seja, o problema pode ser resolvido. O núcleo do programa utiliza o solver *Glucose3* (da biblioteca *py-sat*), responsável por resolver as fórmulas transformadas em FNC para encontrar uma sequência de ações que transforme o estado inicial das peças no estado final desejado.

1	2	3
8	0	4
7	6	5

Estado Inicial

0	1	2
3	4	5
6	7	8

Estado Final

2.1. Estado Inicial

A cada inicialização o programa gera um novo Estado Inicial aleatório. Para garantir que o Estado Inicial seja sempre solucionável pegamos o Estado Final desejado e realizamos movimentos aleatórios sobre ele (embaralhamos), cujo número de repetições da ação é especificado no arquivo de orientação do problema – para gerar puzzles mais ou menos difíceis basta alterar o parâmetro em num_embaralhamentos, na última seção do código.

Para isso, foi-se, antes, necessário realizar o mapeamento das direções:

```
deltas_acao = {"C": (-1, 0), "B": (1, 0), "E": (0, -1), "D": (0, 1)} # Cima, Baixo, Esquerda, Direita.
```

O estado do tabuleiro e as ações possíveis em cada passo são mapeados para variáveis proposicionais, que por sua vez são convertidas em inteiros para uso com o solver SAT.

Sendo 0 o espaço vazio, todos os movimentos são realizados em troca desse espaço com peças adjacentes, assim, garante-se que o embaralhamento seja eficiente, trocando as peças certas e com uma variável armazenando as ações, a fim de evitar que ocorra um movimento que anule o anterior (exemplo: Cima -> Baixo).

2.2. Lógica de programação

Com o Estado Inicial definido, o primeiro passo foi gerar símbolos proposicionais através de variáveis na classe Puzzle8_SAT.

Nela temos o formato *t_P_r_c_v*, ou seja, para cada passo de tempo *'t'*, será representado uma peça *'v'* que está na posição *(r,c)*. Assim como os símbolos de ação: *(t_A_acao)* onde as ações são aquelas definidas anteriormente (C, B, E, D). Para mapeá-los foi usado símbolos (armazenamento) e *int_para_simbolo* (conversão para inteiro) cujos estão associados.

```
def gerar_variaveis_proposicionais(self, max_passos):
    contador_var = 1

    for t in range(max_passos + 1):
        for r in range(self.tamanho_grade):
            for c in range(self.tamanho_grade):
                for v in range(9):
                    simbolo = f"{t}_P_{r}_{c}_{v}"
                    self.simbolo_para_int[simbolo] = contador_var
                    self.int_para_simbolo[contador_var] = simbolo
                    contador_var += 1

    for t in range(max_passos):
        for acao in self.acoes:
            simbolo = f"{t}_A_{acao}"
            self.simbolo_para_int[simbolo] = contador_var
            self.int_para_simbolo[contador_var] = simbolo
            contador_var += 1

    return self.simbolo_para_int, self.int_para_simbolo
```

Com isso, finalmente podemos gerar cláusulas completas através das funções:

- _gerar_clausulas_estado_inicial (t = 0)
- _gerar_clausulas_estado_final (t = *max_passos*)
- _gerar_clausulas_regras_basicas
- _gerar_clausulas_regra_acao

As duas últimas definições servem, respectivamente, para garantir que a cada tempo seja movido no mínimo e no máximo uma peça e para que cada peça, exceto o 0, apareça uma vez em cada passo de tempo. Ou seja, uma única posição poderá ser fornecida para uma única peça durante o mesmo passo de tempo. E, garantir a validade das ações

Só para deixar claro, é considerado como ação válida movimentar o espaço vazio (0) para uma posição adjacente, trocando de posição com a peça correspondente (as demais peças permanecem inalteradas) e inválida movimentar o espaço vazio (0) para uma coordenada fora da matriz 3x3.

Caso uma ação inválida ocorra, a última considerada válida será repetida neste passo.

- _gerar_clausulas_transicao

Garante que o estado do tabuleiro irá mudar de um passo para o outro, dessa maneira, a cada novo passo uma nova ação seja executada.

2.3. Resolvendo o Problema

O objetivo é montar uma fórmula SAT completa e chamar o solver *Glucose3* para encontrar uma solução válida.

```
def resolver(self, max_passos):
    self.gerar_variaveis_proposicionais(max_passos)
    vars_map = self.simbolo_para_int
    int_para_simbolo = self.int_para_simbolo

    with Glucose3() as solver:
        solver.append_formula(self.gerar_clausula_estado_inicial(vars_map))
        solver.append_formula(self.gerar_clausula_estado_final(vars_map, max_passos))
        solver.append_formula(self.gerar_clausulas_regras_basicas(vars_map, max_passos))
        solver.append_formula(self.gerar_clausulas_regras_acao(vars_map, max_passos))
        solver.append_formula(self.gerar_clausulas_transicao(vars_map, max_passos))

        if solver.solve():
            modelo = solver.get_model()
            return self.extrair_solucao(modelo, int_para_simbolo, max_passos)
        else:
            return None, None
```

Verificado se a fórmula passada é satisfazível, se uma solução for encontrada, ele irá extrair e retornar o caminho de ações; se não houver solução, retorna vazio.

```
def extrair_solucao(self, modelo, int_para_simbolo, max_passos):
    caminho_tabuleiros = []
    variaveis_verdadeiras = {p for p in modelo if p > 0}

    for t in range(max_passos + 1):
        tabuleiro_passo_t = [[-1] * self.tamanho_grade for _ in range(self.tamanho_grade)]
        for var_int in variaveis_verdadeiras:
            simbolo = int_para_simbolo.get(var_int)
            if simbolo and simbolo.startswith(f"{t}_P_"):
                _, _, r, c, v = simbolo.split("_")
                tabuleiro_passo_t[int(r)][int(c)] = int(v)
        caminho_tabuleiros.append(tabuleiro_passo_t)

    lista_acoes = [""] * max_passos
    for var_int in variaveis_verdadeiras:
        simbolo = int_para_simbolo.get(var_int)
        if simbolo and "_A_" in simbolo:
            t, _, nome_acao = simbolo.split("_")
            if int(t) < len(lista_acoes):
                lista_acoes[int(t)] = nome_acao

    return caminho_tabuleiros, lista_acoes
```

O programa busca a solução com o menor número de passos possível. Ele faz isso de forma iterativa, testando soluções para 1 passo, 2 passos, 3 passos, e assim por diante, até que solução mais curta seja encontrada.

Para converter o modelo dado pelo solver em uma sequência de ações, o tabuleiro é reconstruído para cada passo de tempo, baseado nas variáveis dadas como verdade.

2.4. Visualização

As funções *imprimir_tabuleiro* e *solucao_completa* exibem o tabuleiro e a solução aplicada de forma legível através do próprio terminal.

```python
def imprimir_tabuleiro(self, tabuleiro_matriz):
    for linha in tabuleiro_matriz:
        print(" ".join(str(p) for p in linha))

def solucao_completa(self, caminho_solucao, acoes):
    for i, tabuleiro in enumerate(caminho_solucao):
        print(f"\nPASSO {i}:")
        self.imprimir_tabuleiro(tabuleiro)
        if i < len(acoes) and acoes[i]:
            print(f"Acao a ser executada: Mover para {acoes[i]}")

            print("mapeamento do estado atual:")
            #adicionanado o mapeamento do estado atual
            for r in range(self.tamanho_grade):
                for c in range(self.tamanho_grade):
                    valor = tabuleiro[r][c]
                    simbolo = f"{i}_P_{r}_{c}_{valor}"
                    if simbolo in self.simbolo_para_int:
                        print(f"{simbolo} -> {self.simbolo_para_int[simbolo]}")
```

3. COMPLEMENTO

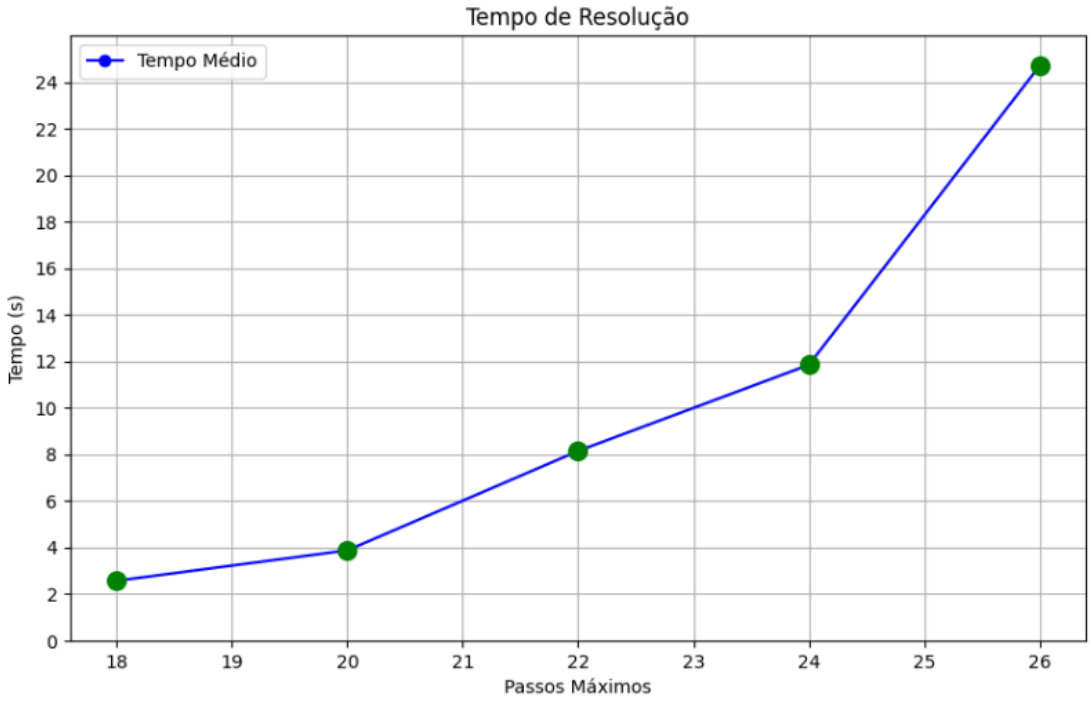
Com o intuito de enriquecer o compreendimento da nossa solução, foi desenvolvido um gráfico visual que analisa o tempo médio de execução do programa.

O primeiro passo foi coletar os dados de desempenho do programa:

```python
TEMPO_DAS_EXECUCOES = [
    {"passos": 18, "tempos": [2.69, 2.46], "solucao": True},
    {"passos": 20, "tempos": [3.98, 3.78], "solucao": True},
    {"passos": 22, "tempos": [8.15], "solucao": True},
    {"passos": 24, "tempos": [13.75, 12.41, 9.39], "solucao": True},
    {"passos": 26, "tempos": [30.65, 18.75], "solucao": True},
]
```

Em seguida, extraímos os principais valores, considerando o tempo da resolução máxima de cada caso, e calculamos a média do tempo de cada conjunto de passos.

Com a biblioteca matplotlib.pyplot importada, podemos finalmente gerar o gráfico:



4. CONCLUSÃO

A abordagem garante que o estado inicial seja sempre solucionável e busca soluções ótimas varrendo o número de passos. A modularidade do código e a clareza das regras lógicas facilitam a compreensão e a extensão do sistema, permitindo

A abordagem utilizada garante que o estado inicial seja sempre solucionável e que as regras do puzzle sejam respeitadas. A clareza das regras lógicas facilita o desenvolvimento do sistema, permitindo que uma solução ótima seja encontrada de forma eficiente, de forma que o passo a passo é visível para melhor compreensão da solução.

```
Estado Inicial Gerado:
1 4 0
7 2 8
3 6 5

Objetivo do Estado Final:
0 1 2
3 4 5
6 7 8
```

```
==========================================
-> SOLUCAO ENCONTRADA EM 20 PASSOS! <-
-> Tempo total de busca: 7.64 segundos.

==========================================
```

```
Passo 1:
1 4 8
7 2 0
3 6 5
Acao a ser executada: Mover para ESQUERDA

mapeamento do estado atual:
1_P_0_0_1 -> 83
1_P_0_1_4 -> 95
1_P_0_2_8 -> 108
1_P_1_0_7 -> 116
1_P_1_1_2 -> 120
1_P_1_2_0 -> 127
1_P_2_0_3 -> 139
1_P_2_1_6 -> 151
1_P_2_2_5 -> 159
```

3.1. Análise de Conformidade

- A implementação atual atende a todos os requisitos e sugestões detalhados no documento do trabalho:
- Resolver o 8-Puzzle: Sim. O programa recebe um estado e retorna a sequência de passos até a solução.
- Uso da biblioteca *python-sat*: Sim. Utiliza o *Glucose3* para toda a resolução lógica.
- Criação de Símbolos Proposicionais: Sim. Símbolos como *t_P_r_c_v* (posição) e *t_A_acao* (ação) são gerados.
- Regras de Unicidade: Sim. Garante que cada peça/posição/ação seja única em cada passo de tempo.
- Regras de Transição (Efeitos): Sim. A função *_obter_clausulas_transicao* modela corretamente como as peças trocam de lugar.
- Tratamento de Movimentos Impossíveis: Sim. Implementa a regra de "repetir o estado" quando uma ação inválida é selecionada.

- Busca por Solução Ótima (Varredura): Sim. O laço principal testa N=1, 2, 3... até encontrar a solução.
- Garantia de Estado Solucionável: Sim. A função *gerar_estado_solucionavel* cria um puzzle a partir do estado final.