JOGO:

REGUA-PUZZLE

Yury Vieira Marques Oliveira

Inteligência Artificial - 2025

SOBRE O JOGO

O "Régua Puzzle" consiste em um tabuleiro unidimensional (uma "régua") com 2n + 1 posições, onde n é o número de fichas de cada uma das duas cores (Azul e Vermelha). O objetivo do jogo é mover as fichas de um estado inicial, geralmente desorganizado, para um estado final ordenado.

OBJETIVO

O estado objetivo é ter todas as fichas de uma cor agrupadas de um lado e todas as fichas da outra cor do outro

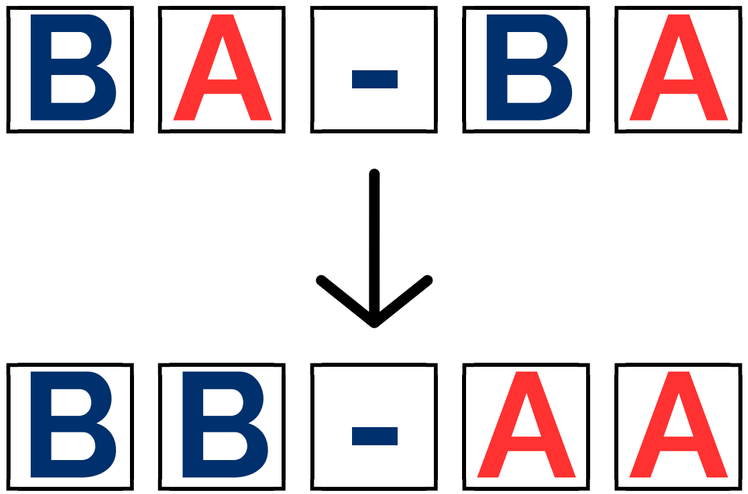
REGRAS DE MOVIMENTO

Movimento Adjacente: Uma ficha pode ser movida para um espaço vazio adjacente a ela.
Salto: Uma ficha pode pular sobre uma ficha adjacente (de qualquer cor) para aterrissar em um espaço vazio.

DESAFIO COMPUTACIONAL:

Encontrar sequencia otima de movimentos para chegar do estado inicial ao final (problema de busca em espaço de estados

ESTADO INICIAL



ESTADO FINAL

IMPLEMENTAÇÃO - VISÃO GERAL

IMPLEMENTAÇÃO MODULAR

Separação clara entre a lógica do jogo , interface do usuário e algoritmo de busca em diferentes arquivos e classes.

C++ COMO LINGUAGEM PRINCIPAL

- Escolhida por seu desempenho superior
- Controle preciso sobre o gerenciamento de memória
- Ideal para algoritmos de busca intensivos

STANDART TEMPLATE LIBRARY (STL)

Utilizada extensivamente para estruturas de dados (vector, string, queue, priority_queue, set, unordered_set, stack)

CMAKE

Para facilitar a compilação e execução do projeto, foi criado um Makefile customizado. Ele automatiza o processo de geração do executável

ESTRUTURAS PRINCIPAIS

REGUA PUZZLE

Gerencia o estado do jogo, a interação com o jogador e a lógica do tabuleiro.
Atributos principais: tabuleiro (vector<char>), num_fichas, tamanho, movimentos

SOLVER

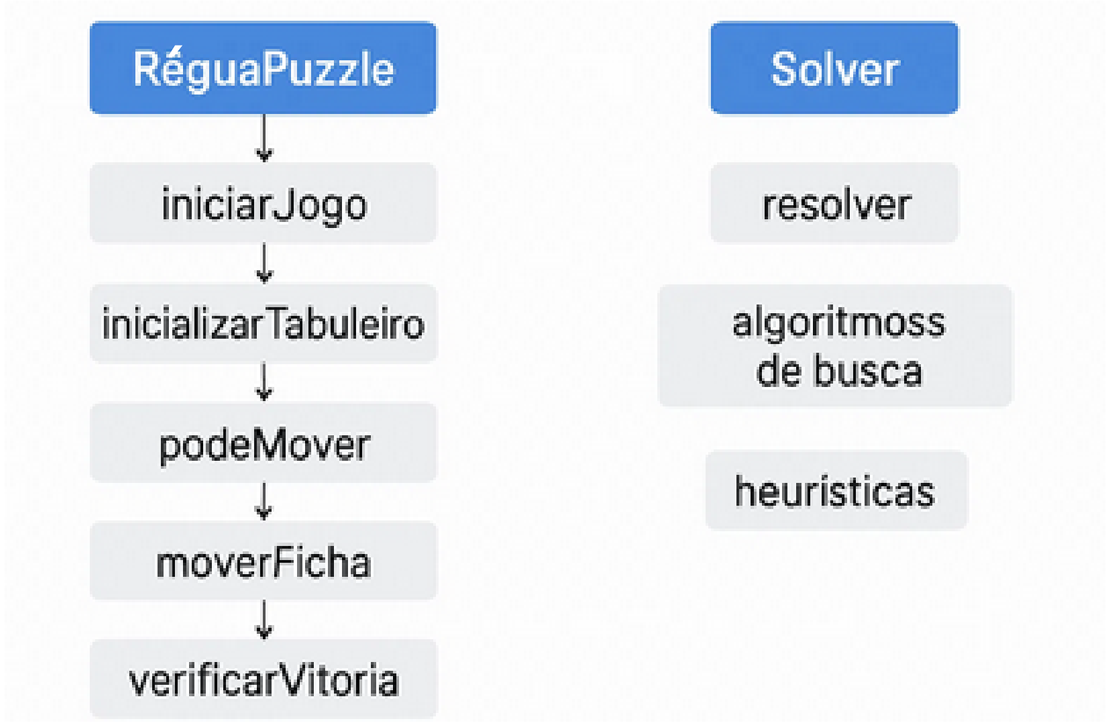
Contém toda a lógica dos algoritmos de busca.
Estruturas Aninhadas:

- Estado: Nó no espaço de busca
- SolverStats: Métricas de desempenho

MAIN

Contém a função principal e as funções de interface do usuário (menus, regras, etc.). Ele atua como o controlador principal, instanciando ReguaPuzzle e chamando o Solver conforme as escolhas do usuário.

FUNÇÕES PRINCIPAIS



NA CLASSE REGUAPUZZLE

- iniciarJogo(): Loop principal do modo manual
- inicializarTabuleiro(): Prepara o tabuleiro no estado inicial
- podeMover(int posicao): Verifica se uma ficha pode ser movida
- moverFicha(int posicao): Executa o movimento de uma ficha
- verificarVitoria(): Checa se o tabuleiro está no estado objetivo

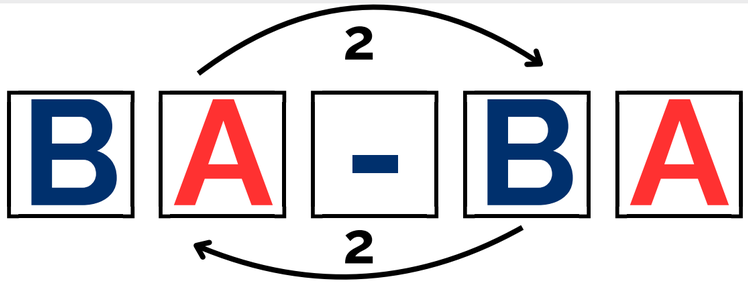
NA CLASSE SOLVER

- resolver(): Ponto de entrada para algoritmos de busca
- Algoritmos: BFS, DFS, Backtracking, Ordenada, Gulosa, A*, IDA*
- encontrarMovimentosPossiveis(): Lista movimentos válidos
- heuristicaManhattan(), heuristicaFichasForaDoLugar(): Cálculo heurístico
- mostrarSolucao(): Exibe o caminho da solução e estatísticas

HEURÍSTICAS

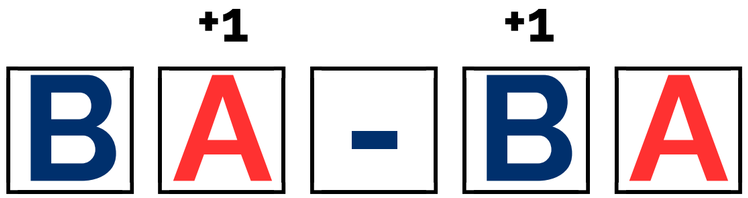
MANHATTAN

- **Conceito Principal**:
- Calcular a soma das distâncias que cada peça precisa percorrer para chegar à sua "casa" ideal, ignorando as outras peças no caminho.
- **Como Funciona no Código:**
- Estado Objetivo: Para n=3, é B B B _ A A A
- Casas das fichas: 'B' nas posições 0, 1, 2; 'A' nas posições 4, 5, 6
- Cálculo: Distância absoluta entre posição atual e posição ideal
- **Exemplo de Cálculo:**
- No tabuleiro B A _ B A:
- Primeira 'B' (pos 1): esta na posição , não entra = 0
- Segunda 'A' (pos 2): dist = abs(3 - 1) = 2
- Terceira '_' (pos 3): ignora
- Quarta 'B' (pos 4): dist = abs(3 - 1) = 2
- Quinta 'A' (pos 4): esta na posição , não entra = 0
- Total: 4



FICHAS FORA DO LUGAR

- **Conceito Principal**:
- Contar o número de "pares desordenados". Um par desordenado ocorre quando uma ficha 'A' aparece antes de uma ficha 'B', o que é o oposto do estado final.
- **Como Funciona no Código:**
- Usa dois loops aninhados para comparar cada ficha com todas as que vêm depois dela
- •Loop externo procura por uma ficha 'A'
- Loop interno varre o resto do tabuleiro à direita
- Incrementa contador de inversões quando encontra um par (A, B)
- **Exemplo de Cálculo:**
- No tabuleiro B A _ B A:
- Primeira 'B' (pos 1): esta na posição , não entra = 0
- Segunda 'A' (pos 2): fora logo = 1
- Quarta 'B' (pos 4): Fora logo = 1
- Total: 2



COMPARAÇÃO N=3

MANHATTAN

Algoritmo	Tamanho do Caminho	Profundidade da Solucao	Nos Expandidos	Nos Visitados	Tempo de Execucao (s)	Fator de Ramificacao
Busca em Largura	14	14	134	134	0.0011	3.14
Backtracking	76	76	415	414	0.0000	2.94
Busca em Profundidade (Limitada)	76	76	108	109	0.0010	3.19
Busca Ordenada	14	14	134	134	0.0000	3.14
Busca Gulosa	14	14	20	33	0.0000	3.16
Busca A*	18	18	38	58	0.0010	3.16
Busca IDA*	20	20	44	43	0.0000	3.21

FICHAS FORA DO LUGAR

Algoritmo	Tamanho do Caminho	Profundidade da Solucao	Nos Expandidos	Nos Visitados	Tempo de Execucao (s)	Fator de Ramificacao
Busca em Largura	14	14	134	134	0.0011	3.14
Backtracking	76	76	415	414	0.0010	2.94
Busca em Profundidade (Limitada)	76	76	108	109	0.0010	3.19
Busca Ordenada	14	14	134	134	0.0000	3.14
Busca Gulosa	19	19	38	65	0.0000	3.27
Busca A*	14	14	123	133	0.0011	3.16
Busca IDA*	14	14	14550	14540	0.0149	3.31

Testes para n=3 e tabuleiro no padrão intercalado (A B A _ B A B)

COMPARAÇÃO N=6

MANHATTAN

Algoritmo	Tamanho do Caminho	Profundidade da Solucao	Nos Expandidos	Nos Visitados	Tempo de Execucao (s)	Fator de Ramificacao
Busca em Largura	47	47	11991	12000	0.0317	3.54
Backtracking	N/A	N/A	5850826	5857031	10.0075	N/A
Busca em Profundidade (Limitada)	5751	5751	8063	8064	0.2306	3.57
Busca Ordenada	47	47	11987	11999	0.0217	3.54
Busca Gulosa	71	71	136	231	0.0010	3.71
Busca A*	66	66	230	396	0.0010	3.71
Busca IDA*	80	80	10465	10464	0.0117	3.79

FICHAS FORA DO LUGAR

Algoritmo	Tamanho do Caminho	Profundidade da Solucao	Nos Expandidos	Nos Visitados	Tempo de Execucao (s)	Fator de Ramificacao
Busca em Largura	47	47	11991	12000	0.0314	3.54
Backtracking	N/A	N/A	5354950	5361149	10.0091	N/A
Busca em Profundidade (Limitada)	5751	5751	8063	8064	0.2326	3.57
Busca Ordenada	47	47	11987	11999	0.0242	3.54
Busca Gulosa	124	124	772	1242	0.0020	3.63
Busca A*	47	47	11939	11983	0.0248	3.54
Busca IDA*	N/A	N/A	12813986	12813971	10.0004	N/A

Testes para n=6 e tabuleiro no padrão intercalado (A B A B A B _A B A B A B)

ANÁLISES

N=3

- Algoritmos Eficientes: Busca em Largura, Busca Ordenada, Busca Gulosa, Busca A* e Busca IDA* apresentaram tempos de execução muito baixos. soluções ótimas ou próximas do ótimo.
- •Backtracking e DFS: Embora tenham encontrado soluções, foram significativamente menos eficientes.
- •Impacto da Heurística: A escolha da heurística (Manhattan vs. Fichas Fora do Lugar) teve um impacto mínimo nos algoritmos informados (A*, IDA*, Gulosa), que já eram muito eficientes.

CONCLUSÕES GERAIS

- Algoritmos Informados (A*, IDA*): São geralmente mais eficientes, especialmente para problemas maiores, quando combinados com heurísticas adequadas.
- •Heurísticas: A heurística Manhattan mostrou-se mais eficiente para problemas maiores, mas sem alcançar o caminho ótimo.
- •Backtracking e Busca em Profundidade: Não são adequados para problemas de maior complexidade devido à sua alta demanda computacional e falha em encontrar soluções.
- •Trade-offs:Há um claro trade-off entre a completude/otimalidade e a eficiência computaciona. Algoritmos que garantem otimalidade (A*, IDA*) podem exigir mais recursos, mas são mais confiáveis para encontrar as melhores soluções.

N=6

- Escalabilidade Crítica: A complexidade do problema (N=6) expôs as limitações de alguns algoritmos.
- •Backtracking e DFS: Não conseguiram encontrar solução para N=6, indicando que não são escaláveis para problemas maiores ou excederam limites de recursos/tempo.
- •Busca IDA* (Fichas Fora do Lugar): Embora tenha encontrado uma solução ótima, o número de nós expandidos/visitados e o tempo de execução foram extremamente altos (mais de 11 milhões de nós e 10 segundos), mostrando uma explosão combinatória para essa heurística.
- Algoritmos Robustos em N=6: Busca em Largura, Busca Ordenada, Busca Gulosa e Busca A* demonstraram melhor escalabilidade, encontrando soluções em tempos razoáveis.
- •Busca Gulosa (Fichas Fora do Lugar): Encontrou uma solução com custo mais alto (125) em N=6, confirmando que, embora rápida, não garante otimalidade.
- •Busca A e Busca IDA (Manhattan): Mantiveram boa performance em N=6, com tempos de execução baixos e soluções ótimas ou próximas, destacando a eficácia da heurística Manhattan para problemas maiores.

PRINCIPAIS DIFICULDADES

1. Gerenciamento de Estados Visitados:
Evitar ciclos e trabalho redundante. Solução: uso de std::unordered_set<string> para armazenar estados já visitados.

2. Complexidade dos Algoritmos:
Implementação correta de A* e IDA* exige atenção aos detalhes, especialmente no gerenciamento da fila de prioridade e no controle do limite de busca iterativo.

3. Otimização de Desempenho:
Para n > 4, o espaço de busca cresce exponencialmente. Buscas cegas rapidamente se tornam inviáveis, reforçando a necessidade de heurísticas eficientes.

4. Garantia de Solução Ótima:
Algoritmos de busca informada nem sempre alcançaram o caminho mínimo, exigindo ajustes nas heurísticas e na implementação.

5. Falta de heurística em para todos os casos de vitória:
Dificuldade para desenvolver uma heurística que funcionasse tanto para a vitória com os Bs na primeira metade do tabuleiro tanto quanto os As. Em diversas tentativas as buscas não informadas alcançaram um caminho menor

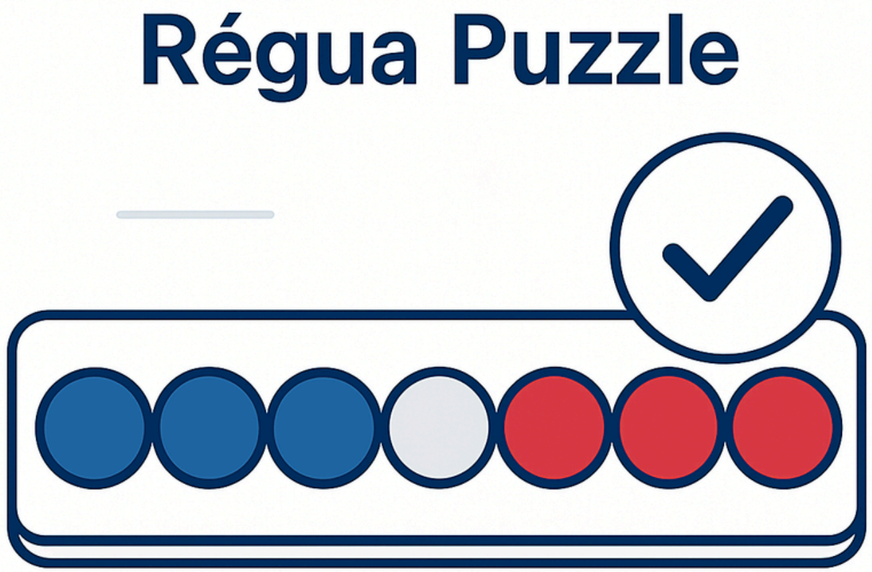
CONCLUSÃO

Pontos-Chave do Projeto

- Implementação modular e eficiente em C++
- Múltiplos algoritmos de busca implementados
- Heurísticas eficazes para guiar a busca

Resultados Obtidos

- Buscas heurísticas superaram significativamente as buscas cegas
- Ampla coleta de resultados para diferentes análises



Considerações Futuras

- Otimização adicional para tabuleiros maiores (n > 10)
- Desenvolvimento de novas heurísticas mais precisas
- Implementação de interface gráfica mais elaborada

FIM

Yury Vieira Marques Oliveira
Matricula: 202565200A
Link do projeto: GITHUB