**Nome: Yury Vieira Marques Oliveira   
Matricula: 202520065A**

**Relatório Técnico: Jogo das Fichas (Régua Puzzle)**

Este documento detalha a implementação de uma solução para o "Régua Puzzle", um jogo de quebra-cabeça que envolve a reorganização de fichas em um tabuleiro linear. O projeto inclui um modo de jogo interativo e um solucionador automático que utiliza diversos algoritmos de busca de Inteligência Artificial para encontrar a solução do quebra-cabeça, além de comparar suas performances.

**1. Descrição do Problema**

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. O estado objetivo é ter todas as fichas de uma cor agrupadas de um lado e todas as fichas da outra cor do outro, separadas por um único espaço vazio.

Exemplo de estado objetivo para n=3: B B B \_ A A A

As regras de movimento são:

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

O desafio computacional reside em encontrar a sequência de movimentos ótima (ou qualquer sequência válida) que leva do estado inicial ao estado objetivo. Isso se enquadra em um problema clássico de busca em espaço de estados, ideal para ser resolvido com algoritmos de IA.

**2. Descrição da Implementação**

A solução foi implementada de forma modular, separando a lógica do jogo, a interface do usuário e os algoritmos de busca em diferentes arquivos e classes.

**2.1. Linguagem(ns) Utilizada(s)**

* **C++:** A linguagem principal utilizada para toda a implementação, escolhida por seu desempenho e controle sobre o gerenciamento de memória, fatores importantes para algoritmos de busca intensivos.
* **Standard Template Library (STL):** Utilizada extensivamente para estruturas de dados (vector, string, queue, priority\_queue, set, unordered\_set) e algoritmos.

**2.2. Formato de Entrada**

O programa interage com o usuário através de um menu de console. As principais entradas são:

* **Número de fichas:** O usuário define o valor de n (o número de fichas de cada cor).
* **Seleção de modo:** O usuário escolhe entre jogar manualmente, ver as regras, usar o solucionador automático ou comparar os algoritmos.
* **Seleção de algoritmo:** No modo de solução automática, o usuário escolhe qual algoritmo de busca deseja utilizar.
* **Seleção de heurística:** Para algoritmos que as utilizam (Gulosa, A\*, IDA\*), o usuário escolhe entre "Manhattan" e "Fichas Fora do Lugar".
* **Movimentos no jogo:** No modo manual, o usuário digita o índice da ficha que deseja mover.

**2.3. Principais Estruturas/Classes**

O código é organizado em torno de três componentes principais:

1. **ReguaPuzzle (Classe):**
   * **Descrição:** Gerencia o estado do jogo, a interação com o jogador e a lógica do tabuleiro. É responsável por inicializar, exibir, embaralhar (se necessário) e validar os movimentos no modo manual.
   * **Atributos Principais:**
     + tabuleiro: Um std::vector<char> que representa o estado atual da régua.
     + num\_fichas: O número n de fichas de cada cor.
     + tamanho: O tamanho total do tabuleiro (2n + 1).
     + movimentos: Contador de jogadas no modo manual.
2. **Solver (Classe):**
   * **Descrição:** Contém toda a lógica dos algoritmos de busca. É uma classe estática, funcionando como um namespace para agrupar as funções de resolução. Ela recebe um estado inicial do tabuleiro e retorna as estatísticas da solução encontrada.
   * **Estruturas Aninhadas:**
     + **Estado (struct):** Representa um nó no espaço de busca. Armazena o estado do tabuleiro, o caminho de movimentos até ele, a profundidade, o custo acumulado (g) e o valor heurístico (h).
     + **SolverStats (struct):** Armazena as métricas de desempenho de um algoritmo após a execução, como caminho da solução, tempo de execução, número de nós expandidos/visitados, etc.
3. **main.cpp (Arquivo Principal):**
   * **Descrição:** Contém a função main 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.

**2.4. Principais Funções/Operações/Métodos**

**Na classe ReguaPuzzle:**

* iniciarJogo(): Loop principal do modo de jogo manual.
* inicializarTabuleiro(): Prepara o tabuleiro no estado inicial.
* podeMover(int posicao): Verifica se uma ficha em uma dada posição 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(tabuleiro, algoritmo, heuristica): Ponto de entrada principal que seleciona e executa o algoritmo de busca escolhido.
* resolverBFS(), resolverBacktracking(), resolverDFS(), resolverOrdenada (), resolverGulosa(), resolverAStar(), resolverIDAStar(): Implementações individuais de cada algoritmo de busca.
* encontrarMovimentosPossiveis(tabuleiro): Retorna uma lista de movimentos válidos a partir de um estado.
* aplicarMovimento(tabuleiro, movimento): Retorna um novo estado do tabuleiro após um movimento.
* heuristicaManhattan(tabuleiro) e heuristicaFichasForaDoLugar(tabuleiro): Calculam o valor heurístico de um estado.
* mostrarSolucao(tabuleiro, stats): Exibe o caminho da solução passo a passo e as estatísticas finais.

**3. Heuristicas**

**3.1. Heurística da Distância de Manhattan (heuristicaManhattan)**

* 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 seu Código:
* Estado Objetivo: Primeiro, vamos lembrar qual é o estado ideal. Para n=3, é B B B \_ A A A. As "casas" das fichas 'B' são as posições 0, 1 e 2. As "casas" das fichas 'A' são as posições 4, 5 e 6.
* Cálculo para Fichas 'B': O código percorre o tabuleiro. Toda vez que encontra uma ficha 'B', ele calcula a distância absoluta (abs()) entre a posição atual da ficha e a próxima posição ideal disponível para uma 'B'.
* Exemplo: No tabuleiro A B \_ B A A, a primeira ficha 'B' está na posição 1. Sua casa ideal é a posição 0. Distância = abs(1 - 0) = 1. A segunda 'B' está na posição 3. Sua casa ideal é a posição 1. Distância = abs(3 - 1) = 2.

**3.2. Heurística de Inversões (heuristicaInversoes)**

* 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 seu Código:
* código usa dois loops aninhados para comparar cada ficha com todas as fichas que vêm depois dela.
* loop externo procura por uma ficha 'A' (if (tabuleiro[i] != 'A') continue;).
  + Uma vez que encontra uma 'A', o loop interno varre o resto do tabuleiro à direita dela.
  + Toda vez que o loop interno encontra uma ficha 'B' (if (tabuleiro[j] == 'B')), significa que temos um par (A, B) que está "invertido" em relação à ordem final (B antes de A).
  + A variável inversoes é incrementada para cada par invertido encontrado.
* **Exemplo: No tabuleiro A A B \_ B, vamos contar as inversões:**
  + A primeira 'A' (posição 0) está antes de duas 'B's (posições 2 e 4). +2 inversões.
  + A segunda 'A' (posição 1) está antes de duas 'B's (posições 2 e 4). +2 inversões.
  + Total: 4 inversões.

**4. Exemplos de Execuções e Estatísticas**

Para demonstrar o funcionamento e comparar os algoritmos, foi executado o modo de comparação para n=4 fichas e tabuleiro padrão(escalonado: exemplo ABA\_BAB)

**Texto

O conteúdo gerado por IA pode estar incorreto.Tabela de Comparação de Desempenho (n=4) com manhattan:**

**Uma imagem contendo Texto

O conteúdo gerado por IA pode estar incorreto.Tabela de Comparação de Desempenho (n=4) com Inversões:**

**Texto

O conteúdo gerado por IA pode estar incorreto.Tabela de Comparação de Desempenho (n=10) com manhattan:**

**Uma imagem contendo Texto

O conteúdo gerado por IA pode estar incorreto.Tabela de Comparação de Desempenho (n=10) com Inversões:**

**Análise das Estatísticas:**

* **Buscas Cegas (Largura, Profundidade, Backtracking, Ordenada):** Encontraram a solução ótima, mas exploraram um número muito grande de nós, resultando em maior tempo de execução.
* *Buscas Heurísticas (Gulosa, A\*, IDA\**): Foram drasticamente mais eficientes. A **Busca Gulosa** foi a mais rápida, pois a heurística guiou a busca de forma muito eficaz. **A**\* e **IDA**\* também foram extremamente rápidos, contudo a heuristica de Manhattan se mostrou mais eficiente.

**5. Principais Dificuldades Encontradas**

1. **Gerenciamento de Estados Visitados:** Uma dificuldade inicial em algoritmos de busca em grafos é evitar ciclos e trabalho redundante. A solução foi usar um std::unordered\_set<string> (ou std::set) para armazenar uma representação em string de cada tabuleiro já visitado, garantindo que cada estado seja processado apenas uma vez.
2. **Complexidade dos Algoritmos:** Implementar corretamente algoritmos como A\* e IDA\* exige atenção aos detalhes, especialmente no gerenciamento da fila de prioridade (para A\*) e no controle do limite de busca iterativo (para IDA\*).
3. **Otimização de Desempenho:** Para um número maior de fichas (n > 4), o espaço de busca cresce exponencialmente. Algoritmos de busca cega, como a Busca em Largura, rapidamente se tornam inviáveis. Isso reforçou a necessidade de implementar buscas heurísticas eficientes.
4. **Garantia de Solução Ótima:** Algoritmos como a Busca Gulosa são rápidos, mas não garantem a solução ótima. Foi importante implementar também algoritmos como A\* e Busca em Largura, que garantem encontrar o menor caminho, para fins de comparação.

**6. Divisão de Tarefas**

N/A