-Introdução:

A resolução do Passeio do Cavalo por backtracking pelo “método ingênuo” se baseia em tentativas, uma a uma, por busca de movimentos soluções possíveis. Isso demanda numerosos casos-bases, que precisam ser recomputados para cada possível casa inicial considerada (o que os torna altamente custoso para casas iniciais arbitrárias). Para a maioria das casas iniciais arbitrárias, esse alto custo leva o programa a demorar mais do que 5 minutos (tempo máximo solicitado para saída). Portanto, empreguei uma estratégia mais eficiente junto ao backtracking para reduzir o custo do programa, a chamada regra de Warnsdorff.

A **regra de Warnsdorff** para o Passeio do Cavalo consiste na seleção de uma casa candidata que minimize o número de próximos movimentos possíveis. Assim, a cada passo, posicionamos o cavalo nas casas em que os movimentos são válidos e comparamos quantos passos possíveis teríamos nessas novas posições. Ou seja, para dar o passo seguinte, teremos como critério buscar uma casa com menor número de casas candidatas posteriores. Assim, reduzimos a quantidade de casas visitadas e, por conseguinte, diminuímos o custo do programa.

*__***Observação***__: O programa ainda conterá backtracking, pois, enquanto o método ingênuo tem como critério de retorno (volta) a falha no encontro de uma solução, a regra de Warnsdorff terá como critério para retornar os casos em que as possibilidades de movimentos futuros aumentarão e, portanto, gerarão uma compilação de alto custo. Com isso, essa regra permite reduzir o número de ramificações para encontrar soluções. Portanto, o número de casas visitadas serão todas as casas candidatas.

Implementação: Descrição dos elementos básicos que compõem o programa.

-Elementos e funções básicas para o Passeio do Cavalo

- Vetores de movimentos do cavalo

```
int movimentoLin[8] = {1, 1, 2, 2, -1, -1, -2, -2};
int movimentoCol[8] = {2, -2, 1, -1, -2, 2, -1, 1};
```

- Função de impressão do tabuleiro:

```
      void Impressor(int tabuleiro[])
```

Explicação da lógica do programa:

-**Função passeio**:

- Recebe como parâmetros x e y, respectivamente, linha e coluna da primeira posição do cavalo;
- Cria um vetor que funcionará como um tabuleiro (tabuleiro[64]);
- Chama a função “AvaliadorDeMovimentos” dentro de um laço (while), tendo como argumento o vetor tabuleiro, as posições x e y. Esse laço chama a função “AvaliadorDeMovimentos” até obtermos uma solução válida.

-Função AvaliadorDeMovimentos:

```
int AvaliadorDeMovimentos(int tabuleiro[], int Xinicial, int Yinicial)
```

- Recebe como parâmetros um vetor de 64 posições (o tabuleiro) e as posições iniciais do cavalo (respectivamente, Xinicial e Yinicial);
- Chamamos a função “ConfiguradorDeBase”, a qual preenche o tabuleiro com valores -1 que representam os espaços vazios;
- Criamos os inteiros x e y que valem, respectivamente, Xinicial -1 e Yinicial -1 para que possamos usar como entrada valores no intervalo [1,8];
    - Usamos x e y para criar uma posição inicial para o cavalo.
- Criamos um laço que vai de 0 a 63:
    - Chamamos a função “Movimentador”, que altera os valores de x e y e busca valores válidos para elas, e, assim, busca movimentos válidos para o cavalo;
    - Caso não encontremos um movimento válido, retornamos 0, ou seja, sinalizamos que o movimento feito não é possível.
- Chamamos a função “CavaloValido”, a qual avalia se o movimento obtido no passo anterior (novos valores de x e y) obedecem à lógica de movimento do cavalo.

```
int CavaloValido(int x, int y, int start_x, int start_y)
```

    - Caso não seja um movimento de L, o movimento do cavalo, retornamos 0, ou seja, sinalizamos que o movimento feito não é possível

-Função “ConfiguradorDeBase”:

```
void ConfiguradorDeBase(int tabuleiro[])
```

- Recebe um vetor (tabuleiro como parâmetro);
- Preenchemos o vetor com valor -1.

-Função “Movimentador”:

```
int Movimentador(int tabuleiro[], int *x, int *y)
```

- Criamos um inteiro “seed” que recebe um valor aleatório no intervalo fechado entre 0 e 7. Esse valor servirá para escolhermos um candidato inicial entre os 8 movimentos possíveis do cavalo presentes nos vetores de movimento do cavalo.
- Dentro de um laço for, criamos outro inteiro, “i”, que corresponde à soma de “seed” com “count”, sendo este último a variável que é incrementada pelo laço e permite percorrer os 2 vetores de movimentos candidatos do cavalo.
- Criamos também 2 inteiros, nx e ny, sendo que cada um deles receberão 8 novas posições possíveis para o cavalo, por meio de uma soma das posições originais (*x e *y) com as posições candidatas (movimentoLin[i] e movimentoCol[i]).

- Agora aplicamos o critério da Regra de Warnsdorff e faremos backtracking, avaliando qual das posições candidatas atende ao critério de ter menor quantidade de posições seguintes (ou seja, menor quantidade de possibilidades de movimentos posteriores).

    - Para isso, passamos cada uma das posições candidatas por uma função que checa se o movimento candidato leva a uma nova posição vazia (função “ChecadorDeVazio”).

    - Posicionamos o cavalo nas casas candidatas e avaliamos a quantidade de movimentos que cada uma das posições permite fazer (função “ContaVazios”), sendo esse valor guardado, a cada vez, na variável inteira “Vazio” e comparado a um valor mínimo temporário, chamado “min”, que recebe a cada movimento o valor de "vazio" se for maior que ele.

    - Assim, ao final do laço, teremos encontrado um movimento do cavalo que terá o mínimo de casas vazias posteriores para o cavalo. Então, o índice desse movimento é recebido pelo inteiro “flag”.

- A cada passagem pelo laço, caso a casa candidata atenda às condições de pertencerem ao tabuleiro e serem vazias, podemos considerá-las casas visitadas pelo cavalo, então incrementaremos 1 à variável global “visitadas”.

- Armazenamos em uma variável “nx” e “ny” a posição seguinte para a qual o cavalo moverá em relação à casa original. Além disso, marcamos no vetor (tabuleiro) o índice (*y*8 + *x) do movimento que foi feito na posição (nx + ny*8) do vetor.

- Ao final da função, atualizamos *x e *y (posições originais) para receberem as novas posições “nx” e “ny”. Essas novas posições são passadas de volta para a função “AvaliadorDeMovimentos” e avalia se o movimento do cavalo será válido.

-Função “LimitadorDoTabuleiro”:

```
bool LimitadorDoTabuleiro(int x, int y)
```

- Essa função avalia se o movimento candidato do cavalo está dentro dos limites do tabuleiro

-Função “ChecadorDeVazio”:

```
bool ChecadorDeVazio(int tabuleiro[], int x, int y)
```

- Essa função checa 2 critérios para o movimento do cavalo: se ele estará dentro dos limites do tabuleiro e se a casa candidata do cavalo está vazia.

-Função “ContaVazios”:

```
int ContaVazios(int tabuleiro[], int x, int y)
```

- Dada uma posição candidata (x,y), passada como parâmetro, essa função conta a quantidade de vazios dessa posição e conta a quantidade de vazios que ela terá.

-Função “Impressor”

```
void Impressor(int tabuleiro[])
```

Caso as condições de movimentos sejam todas válidas, chamamos a função “Impressor”, que imprimirá no arquivo “saida.txt” no modo append (a) tanto o tabuleiro preenchido quanto o valor das casas visitadas e retrocedidas.