UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Universidade Federal de Minas Gerais

Turma: Ciência da Computação **Prof.:** Loureiro

Nome: Rubia Alice Moreira de Souza

Passeio do Cavalo

Belo Horizonte
2022

1. Fases Projeto

Todo o projeto foi desenvolvido utilizando controle de versão. Você pode acompanhar o histórico de commits com as modificações relacionadas a cada um deles no repositório do projeto: Pesseio-Do-Cavalo.

Inicialmente, o projeto iria ser implementado de forma iterativa para realizar o passeio do cavalo. Neste caso, seria utilizada uma pilha para armazenar as posições visitadas pelo cavalo, mantendo as mais recentes acima, e que poderia ser desempilhada para realizar o Backtracking. Além disso, cada posição teria uma lista com as posições possíveis e válidas de serem visitadas a partir dela. No entanto, além do grande consumo de memória, essa abordagem teria grande dificuldade de manter o controle de estado das posições válidas. Ela foi revertida a partir do commit: 7f6f43c91f31e24351776600b5811f0ab84d9ea4.

Depois, iniciei uma abordagem recursiva, pois a descrição do problema passa a ser mais simples dessa forma. As chamadas continuam até o ponto em que o cavalo chega a uma posição sem saída ou finaliza o tabuleiro. Essa etapa iniciou no commit: 00393e869e206b1bde89f4e1ff0f6365496b1095.

Contudo, a forma de implementação não funcionava de forma eficiente para certas casas iniciais do tabuleiro. Enquanto que para casas como (1, 1) e (3, 2) o tempo de execução era curto, casas como a (2, 2) demoravam muito para finalizar. Então, foi necessário adicionar uma heurística. No caso, foi utilizada a heurística da regra de Warnsdorff, em que o cavalo passa a priorizar ir para as casas que terão menos possibilidades de progredir a partir delas. Essa heurística foi incluída no commit: 1b7594f2b0e4ec51b03840a600df23d2c5614866.

A partir da inclusão dessa heurística foram realizados testes com a posição inicial em cada casa do tabuleiro e nenhuma apresentou problema de desempenho para o caso de um tabuleiro 8x8.

2. Implementação

Como dito anteriormente, a implementação final da solução consiste em uma forma recursiva que utiliza uma fila de prioridades para ordenar as casas que devem ser testadas primeiro pelo cavalo durante o passeio.

O método recursivo consiste na seguinte estrutura. Inicialmente, verificamos a condição de parada, que consiste no cavalo ter percorrido uma quantidade de casas **diferentes** igual a quantidade de casas do tabuleiro.

Após isso, encontramos todas as posições válidas que o cavalo pode ir a partir da posição atual. Essa posição atual é passada junto do cavalo a cada chamada da função. Para cada posição válida, verificamos quantas posições conseguimos ir a partir dessa

próxima. A quantidade de posições possíveis irá definir a prioridade dessa casa. Por fim, adicionamos a casa, junto da sua prioridade, na fila de prioridade de casas possíveis.

A fila de prioridade ordena as casas de modo a deixar aquelas que possuem menos possibilidades primeiro.

Então, poderemos percorrer cada casa registrada na fila de prioridades. A partir da primeira, aquela que tem menos possibilidades que podem derivar a partir dela, chamamos novamente a função recursiva, passando essa primeira casa como a nova posição atual do cavalo. Assim, todo o processo será executado novamente, mas considerando a nova posição atual do cavalo. Desse modo, a cada chamada, o cavalo estará se movimentando para uma nova posição do tabuleiro.

A função recursiva tem um retorno booleano, sendo verdadeira para o caso em que o cavalo conseguiu atingir o fim do tabuleiro e falso para o caso que ele chegou a um ponto sem saída. Como ela é chamada novamente dentro de uma condicional (if) no laço (for) que verifica as posições na fila de prioridades, a função irá retornar falso apenas no momento que geramos a fila de prioridades está vazia, seja porque todas casas foram testadas ou pois não havia nenhum movimento válido. Desse modo, após o retorno falso na condicional, devemos retroceder o cavalo para a posição anterior, atualizar seus valores e testar a próxima posição. Se não houver mais posições, devemos retroceder novamente. Assim, o backtracking é realizado. Todo o registro da posição anterior e da fila de posições válidas é mantido pelo próprio contexto de cada chamada recursiva.

Para o caso em que a função recursiva seja chamada e o cavalo já tenha percorrido uma quantidade de casas diferentes igual ao tamanho do tabuleiro, a condição de parada foi atingida. A partir desse momento, essa última chamada retorna verdadeiro para as chamadas anteriores. Para cada chamada anterior, ela desaloca a fila de prioridades e retorna verdadeiro até encerrar todas as chamadas.

Dessa forma, o empilhamento de posições é feito pela própria estrutura da call stack da linguagem C, pois a cada chamada de função, a função atual guarda o contexto de suas variáveis locais.

3. Estrutura de Dados

Neste trabalho foi utilizada apenas uma estrutura de dados. No caso, foi uma fila de prioridades encadeada: “FilaPossibilidadePosicoes”. Ela consiste em uma célula inicial que marca a posição do começo da fila. Cada célula tem uma referência para a próxima e a última célula é identificada por ter a referência para a próxima como “NULL”.

As células são armazenadas em ordem, segundo uma prioridade crescente. Ou seja, aquelas que têm a menor prioridade são colocadas primeiro. Essa ordem foi estabelecida para cumprir a heurística da regra de Warnsdorff, que propõem que o cavalo priorize sempre se mover para a posição que contém menos possibilidades de movimentos.

Como o projeto foi implementado de forma recursiva, não houve necessidade de criar uma pilha para conter a sequência de passos do cavalo, pois como dito anteriormente, essa ordem pode ser representada pela ordem de chamadas e resoluções das funções recursivas.

4. Análise Resultados

4.1. Pré-Heurística

Antes da implementação da regra de Warnsdorff, haviam casos de teste que o programa executava por tempo muito longo. Além disso, a quantidade de casas visitadas e retrocedidas era muito maior. Abaixo, segue um resultado para a posição (1, 1):

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	**1**	38	59	36	43	48	57	52
2	60	35	2	49	58	51	44	47
3	39	32	37	42	3	46	53	56
4	34	61	40	27	50	55	4	45
5	31	10	33	62	41	26	23	54
6	18	63	28	11	24	21	14	5
7	9	30	19	16	7	12	25	22
8	64	17	8	29	20	15	6	13

Total casas visitadas: 8250733
Total casas retrocedidas: 8250669

4.2. Pós-Heurística

Após a implementação da heurística, todos os casos passaram a ser solucionados rapidamente e com uma quantidade de casas retrocedidas bem inferior.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	20	17	38	3	22	7	28	5
2	39	2	21	18	49	4	23	8
3	16	19	50	37	24	27	6	29
4	51	40	**1**	60	53	48	9	26
5	62	15	52	47	36	25	30	57

6	41	44	61	54	59	56	33	10
7	14	63	46	43	12	35	58	31
8	45	42	13	64	55	32	11	34

Total casas visitadas: 110
Total casas retrocedidas: 46

Observação: Entre todas as posições do tabuleiro, esta foi a única que houve necessidade de realizar backtracking.

4.3. Diferentes tamanhos tabuleiro

Além disso, foram feitos testes com diferentes tamanhos de tabuleiros. Para os tamanhos 6x6 e 8x8, sempre houveram soluções válidas. Já para um tabuleiro 5x5 haviam posições iniciais em que não era possível executar o passeio do cavalo. As posições estão representadas abaixo.

	1	**2**	**3**	**4**	**5**
1		Impossível		Impossível	
2	Impossível		Impossível		Impossível
3		Impossível		Impossível	
4	Impossível		Impossível		Impossível
5		Impossível		Impossível	