Passeio do cavalo

Fases do projeto, análise da implementação final e comparação do desempenho.

Aluno: Luiz Fernando Gonçalves Rocha
Professor: Antônio Alfredo Ferreira Loureiro
MD - Turma TM1

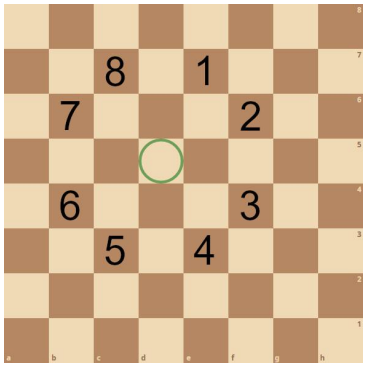
Fases do projeto:

Fase inicial:

Inicialmente, quis explorar o problema e, para isso, fiz uma base de funções para que eu pudesse abstrair a forma como procurava uma solução. Assim, fiz uma série de funções que checavam a disponibilidade de movimentos, realizavam-nos e desfaziam-nos. **Isso resultou em um formato simples que buscava uma solução de forma quase exaustiva, já que a alteração da prioridade demandava muitas alterações no código.**

Vou usar bastante esse termo, então vou definir exatamente a quê me refiro quando o utilizo: estou chamando de prioridade a forma com a qual o programa, dada a lista de movimentos que ele pode realizar a partir de cada posição, decide qual será realizado primeiro e quais serão realizados posteriormente em qual ordem, caso seja necessária a realização de um *backtracking*.

Nessa fase inicial, tudo que o programa fazia era, dada uma lista de movimentos possíveis para cada posição, tentar primeiro o para cima e para a direita, depois o para a direita e para cima, e assim em diante, até fechar o círculo de movimentos, como retrata a imagem:



Caso um caminho não fosse encontrado ou não houvesse mais opções de movimento, o cavalo voltava e tentava o próximo movimento da prioridade (nesse caso, primeiro o movimento 1, depois o 2, depois o 3 etc.).

Não é surpreendente que esse algoritmo era assustadoramente ineficiente. Por mais que ele funcionasse em uma quantidade considerável de casas, a quantidade de passos dados nunca ficava abaixo dos milhões, chegando até mesmo nos bilhões nos casos em que a busca era mais demorada. Há também várias casas para as quais não foi encontrado um caminho, mesmo depois de horas.

As únicas estruturas de dados que usei nessa primeira tentativa foram uma matriz de inteiros e um vetor de inteiros.

A matriz, que era 8 por 8, representava o tabuleiro, sendo que cada casa podia assumir um valor de 0 a 64: 0 se a casa ainda não tivesse sido visitada, um valor de 1 a 64 se ela já tivesse sido visitada, sendo esse valor correspondente à sua posição na sequência de casas que o cavalo já visitou.

O vetor, por sua vez, guardava os movimentos realizados anteriormente usando uma estratégia parecida com a retratada na imagem: 1 para o movimento para cima e para a direita, 2 para o movimento para a direita e para cima, e assim por diante.

Caso nenhuma casa do tabuleiro tivesse sido marcada com o número 64 e não houvesse nenhuma casa com o número 0 no alcance, era necessário realizar um *backtracking*. Nesse caso, a informação sobre o movimento que o cavalo realizou para chegar na casa era recuperada do vetor e utilizada para realizar o movimento inverso, desmarcando a casa e tentando o próximo movimento disponível da casa anterior, se houver. Isso era feito até que um passeio fosse encontrado (ou seja, até que uma casa tivesse o número 64, que indica o fim do caminho).

Aprimoramento da prioridade:

Observando que havia algumas casas para as quais o método anterior funcionava, **resolvi alterar a ordem da prioridade na esperança de encontrar alguma que encontrasse um passeio para todas as casas**.

Para isso, fiz um vetor "`int prioridade[8]`", que continha oito números inteiros, indicando a ordem com que as casas disponíveis deveriam ser verificadas. Por exemplo, o vetor `{8, 5, 3, 1, 4, 2, 6, 7}` fazia com que, tomando como referência a imagem na página anterior, primeiro fosse tentado o movimento 8, depois, se o cavalo voltasse, o movimento 5, assim em diante, até que se esgotassem as posições, caso no qual o cavalo voltaria mais uma casa.

Não foram feitas grandes modificações nas outras partes do programa. Elas apenas foram adaptadas para tomar esse vetor como referência para a tomada de decisões.

Ajustando a prioridade, **consegui melhorar bastante o desempenho se comparado à implementação anterior**. Consegui até mesmo implementar funções que alteravam a ordem de prioridade dependendo da posição geral da casa inicial. **Porém, ainda havia casas para as quais o programa não encontrava um caminho.**

O que eu percebi é que o caminho era encontrado mais facilmente se a posição inicial fosse próxima das bordas do tabuleiro e se o cavalo procurasse se manter próximo da borda mais próxima. Não entendi muito bem o motivo inicialmente, mas isso está relacionado com a estratégia utilizada na versão final do código.

Entendendo a Regra de Warnsdorff:

Nesse estágio, já que eu tinha um programa quase funcional e só precisava otimizar seu funcionamento, resolvi estudar um pouco sobre o passeio do cavalo na internet. **Conheci, então, uma heurística chamada de regra de Warnsdorff.** Ela diz que, dada uma lista de movimentos possíveis a partir de uma casa, deve ser escolhido o que leva para a casa mais inacessível, ou seja, a que tem menos movimentos disponíveis.

Essa abordagem faz muito sentido, já que, ao escolher a casa menos acessível, é menos provável que, ao se aproximar do final do passeio, uma casa tenha sido esquecida entre as que receberam os movimentos iniciais, cuja correção demandaria muitíssimos *backtrackings*, ao menos usando o método da versão inicial do meu programa. Debugando versões mais antigas do meu programa, percebi que era exatamente isso que acontecia nos casos em que o programa executava por horas sem encontrar um caminho.

Essa abordagem também parece se relacionar com o fato de eu ter obtido um desempenho melhor ao fazer o cavalo permanecer o máximo possível em um lado do tabuleiro: ao permanecer mais perto de um lado, os movimentos ficam mais condensados e

creio que seja mais provável que, acidentalmente, o cavalo escolha a casa menos acessível dentre as suas opções.

Aplicação da Regra de Warnsdorff:

Para aplicar a regra, praticamente tive que reescrever meu código. As funções que eu tinha não podiam ser facilmente adaptadas para estudar a acessibilidade das casas seguintes.

Aproveitei essa chance para alterar totalmente a minha abordagem.

A solução final e as estruturas de dados utilizadas:

Para facilitar o raciocínio e o processo de programação, fiz uma *struct* chamada "casa". Essa *struct* contém os seguintes campos:

```
int valor, maiorTentado, estudada;
coordenadas proxima[8], anterior, posicao;
```

Onde "coordenadas" é uma estrutura com dois campos: `int linha, coluna.`

O inteiro "`valor`" diz qual é a posição da casa, de 1 a 64. A primeira casa que o cavalo visita tem valor 1, a segunda tem valor 2, assim em diante.

O inteiro "`qtdProximosPossiveis`" diz quantas opções de movimento existem a partir dessa casa. Uma opção é considerada se a casa está no tabuleiro e ainda não foi visitada.

O inteiro "`maiorTentado`" diz qual foi o maior movimento já tentado dentre os disponíveis, que estão listados em ordem de qualidade no vetor "`proxima`". Quando um *backtracking* é feito, esse valor é incrementado, de modo indicar que, na próxima tentativa de movimento, outro movimento deve ser tentado.

O inteiro "`estudada`" pode assumir os valores 0 ou 1, para não estudada e estudada, respectivamente. Dizemos que uma casa está "`estudada`" quando seu vetor "`proxima`" foi preenchido e ordenado, sua "`qtdProximosPossiveis`" foi determinada e seu valor de "`maiorTentado`" foi zerado.

O vetor "`proxima[8]`" contém as casas acessíveis a partir dessa com um único movimento. Está ordenado em ordem de qualidade pela heurística do grau mínimo: as casas menos acessíveis (que têm menos movimentos disponíveis) vêm antes e, assim, são tentadas antes.

A coordenada "`anterior`" guarda a casa que o cavalo ocupou antes da atual. Ela guia o *backtracking*, uma vez que o cavalo volta para ela caso seja necessário.

A coordenada "`posicao`" guarda as coordenadas da posição da casa.

Definida a estrutura "casa", declarei uma matriz oito por oito formada por casas. Esse é o tabuleiro. Daí, **segui o seguinte procedimento na função `void passeio(int linhaInicial, int colunaInicial)` para achar o passeio:**

Primeiro, coloco o número 1 na casa inicial, que é o primeiro ponto pelo qual o cavalo passa.

Segundo, entro em um loop que tem como condição o número de movimentos com sucesso ser menor que 64, ou seja, ele roda até que um caminho seja encontrado.

Terceiro, se a casa atual ainda não foi estudada (ou seja, se o valor de seu campo `estudada` é 0), ela é estudada, isto é, seu valor `maiorTentado` é zerado, são encontrados os próximos movimentos possíveis, eles têm sua acessibilidade (`qtdProximosPossiveis`) determinada e o vetor formado por eles (`proxima`) é ordenado, sendo o critério a sua inacessibilidade.

Quarto, se o número total de tentativas na casa atual (`qtdProximosPossiveis`) é igual ao número de tentativas tentado (`maiorTentado`), então é realizado um *backtracking*, zerando o `valor` da casa atual, zerando o seu campo `estudada` (já que um *backtracking* requer um novo estudo, considerando o estado atual do tabuleiro) e alterando-a para a casa `anterior`, incrementado seu `maiorTentado` (já que uma das tentativas foi esgotada). Caso contrário, é realizado um passo para frente, tomando o melhor movimento disponível no vetor `proxima`, indicado pelo valor de `maiorTentado`, como posição atual e gravando o número do movimento atual no `valor` da casa.

Sexto, isso é repetido até que o número da posição atual chegue a 64 ao final de uma iteração, o que para o laço. Nesse caso, o programa prossegue para imprimir o resultado em “saida.txt”, como foi solicitado na especificação.

Obs: Resolvi não abordar muitos detalhes de implementação aqui, já que já gastei muito espaço, mas comentei o código em toda a sua extensão, explicando cada detalhe. Recomendo fortemente a leitura do arquivo “passeio.h” para uma compreensão aprofundada da implementação.

Análise do comportamento e do desempenho em cada técnica

Faremos a análise do desempenho do programa na posição (8, 8) usando as três técnicas aplicadas durante o desenvolvimento da solução. Essa casa foi escolhida porque os três algoritmos encontraram um caminho para ela.

Primeira versão: prioridade simples.

Caminho encontrado:

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

Passos para frente: 27241113
Backtrackings: 27241049
Tempo de execução: 1,732 s

Como é possível notar, o cavalo, no início do passeio, tende ao canto superior do tabuleiro, uma vez que as primeiras casas tentadas são as que estão ao nordeste do cavalo. Depois, ele tende a ficar no lado direito do tabuleiro, já que os movimentos do lado esquerdo são testados por último.

Lembrando que a prioridade utilizada nessa primeira implementação é simplesmente {1, 2, 3, 4, 5, 6, 7, 8}, tendo como referência o mapa na primeira página.

Note que o número de tentativas é muito elevado, o que indica que, provavelmente, algumas casas foram deixadas para trás e o cavalo precisou retornar várias vezes para visitá-las e poder completar o caminho.

Essa implementação tinha dificuldades para encontrar caminhos em muitas casas.

Segunda versão: prioridade por vetor (alterável com base na casa)

Caminho encontrado:

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

Passos para frente: 17739768
Backtrackings: 17739704
Tempo de execução: 1,176 s

Nessa segunda tentativa, a prioridade podia ser alterada para que ela fosse mais adequada à casa inicial. Apesar disso, ela era constante no decorrer da execução. Foram priorizados os movimentos para baixo, sendo testados por último os para cima.

Seguindo o formato usado anteriormente, o vetor prioridade foi `{4, 5, 3, 6, 2, 7, 1, 8}`, escolhido especificamente para manter o cavalo próximo da borda.

Note que, por isso, a parte inferior do tabuleiro recebeu os primeiros movimentos.

Essa estratégia reduziu consideravelmente as tentativas para algumas escolhas de casa inicial e vetor de prioridade, imagino que porque ela começa a se aproximar da adotada na versão final, já que, quando o cavalo se mantém de um lado, é mais provável que o movimento escolhido seja o menos acessível, dada a maior concentração de casas já visitadas na região.

Versão final: Regra de Warnsdorff.

Caminho encontrado:

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

Passos para frente: 64
Backtrackings: 0
Tempo de execução: 0,093 s

A versão final prioriza os movimentos para as casas menos acessíveis.

Claramente, o seu desempenho é muito superior ao dos outros métodos, já que nem mesmo foram necessários *backtrackings* no caso em questão. A análise das próximas casas altera a prioridade de forma que evita que uma casa não seja visitada no início do programa e, assim, fique inacessível no final, exigindo muitos *backtrackings*.

A diferença no tempo de execução é particularmente notável, caindo de 1,176 segundos no método anterior para 0,093 segundos nesse método. Há, porém, outras casas para as quais a diferença é ainda maior. Como disse, os outros métodos passam horas sem encontrar soluções em algumas casas.

Por mais que o método final seja mais complexo, ele é mais criterioso e preciso, de modo que a quantidade de iterações é muito menor.

Vale mencionar que a única casa que necessita de *backtracking* na versão final é a (6, 5), em que são feitos 46 *backtrackings*. Todas as outras têm seus caminhos encontrados sem *backtrackings*, em um tempo similar ao caso estudado.

Conclusões

O que mais chamou a minha atenção nesse projeto foi a forma como uma heurística relativamente simples foi capaz de causar impactos enormes na capacidade e na velocidade da obtenção de soluções.

Além disso, achei interessante o uso da estrutura “casa” na solução final, que elevou a abstração e permitiu que eu pensasse de forma mais clara durante a implementação, especialmente no que diz respeito ao *backtracking.*