Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Matemática Discreta - 2022/2

Passeio do Cavalo

Aluno: José Eduardo Duarte Massucato
Matrícula: 2022043620 **Turma:** TM1

Problema

Este trabalho tem como objetivo elucidar a técnica da tentativa e erro (*backtracking*) por meio do problema do **Passeio do Cavalo**. A partir desta premissa, eu, a priori, procurei compreender o problema buscando potenciais padrões nas soluções, para inicializações arbitrárias. De maneira frustrada, não consegui chegar a uma conclusão de imediato, o que me levou ao desenvolvimento do meu primeiro algoritmo para solucionar esse problema.

Este meu primeiro *software* buscava, despretensiosamente, por movimentos possíveis[1] e, caso não houvesse nenhum, ele utilizava do *backtracking*, até que ele conseguisse novamente encontrar uma casa onde haja um movimento possível. Por "força bruta", esse programa tinha capacidade de encontrar uma solução para qualquer ponto de partida. Entretanto, note que o cavalo possui 8 possíveis movimentos em um tabuleiro de dimensões 8x8; isto significa que existem 8^{64} possíveis passeios desse cavalo. Com isso, a complexidade deste algoritmo ficava em $\mathbf{O(8^{64})}$, o que é muito custoso não só para a pessoa que utilizará dele, mas também para o computador.

Em vista disso, eu notei que estava faltando alguma outra métrica para que o meu algoritmo operasse mais satisfatoriamente. Então, eu testei a hipótese, que funcionou muito melhor do que o esperado, de levar o cavalo para uma casa onde ele possuiria uma quantidade pequena de movimentos possíveis[1]. A partir desse princípio, otimizei o meu algoritmo para o que ele se tornou no final.

Comentário: pesquisando mais sobre o porquê dessa otimização que eu implementei ser tão eficaz, descobri que já existe todo um estudo por trás disso. É a chamada "*Regra de Warnsdorff*". Coloquei nas referências um artigo que explica mais sobre isso.

[1] São os movimentos que não fazem o cavalo sair do tabuleiro, nem vão para uma casa já visitada.

Implementação

Meu algoritmo tem por **objetivo** imprimir no arquivo “saida.txt” uma possível resolução para o **Passeio do Cavalo**, a partir de um determinado ponto de partida, utilizando *backtracking*, caso necessário.

Variáveis

1) **int l e int c:** Diminuem em uma unidade as entradas da função **void passeio(int x, int y)**, a fim de facilitar a manipulação delas no array bidimensional tabuleiro[8][8]. A variável **l** representa as linhas, enquanto a variável **c** representa as colunas. Portanto, elas variam seus valores de 0 a 7.

2) **int cont:** Diz em qual movimento o cavalo está. Portanto, ela varia de 1 a 64.

3) **int tabuleiro[8][8]:** É o tabuleiro por onde o cavalo passeará.

4) **int salvamov[64]:** É um vetor que guarda qual dos possíveis movimentos que o cavalo realizou para cada instância de movimento. Assim, cada posição desse array possui valores que variam de 1 a 8.

5) **int contIdas e int contVoltas:** Guardam, respectivamente, a quantidade de idas que o cavalo deu e a quantidade de voltas que ele precisou dar. Dessa forma, temos que contIdas - contVoltas = 64.

6) **int restricao[8][8][8]:** É um array tridimensional que guarda quais movimentos, dentre os possíveis, o cavalo **não deve fazer** a partir de uma determinada posição. Assim, os valores de cada elemento variam de 1 a 8.

Observação: **int i**, **int j** e **int p** são variáveis auxiliares criadas com o intuito de assistir no transcorrimento dos arrays supracitados.

Funções

1) **void movimento (int n, int* l, int* c):** Realiza um dos 8 movimentos possíveis sobre as variáveis **l** e **c** anteriormente citadas.

2) **void antimovimento(int n, int* l, int* c):** Realiza o movimento contrário de cada um dos 8 movimentos possíveis sobre as variáveis **l** e **c**. Então, por exemplo, se o movimento 1 é andar 2 casas para cima e 1 para a direita, o anti movimento 1 será andar 2 casas para baixo e 1 para a esquerda.

3) **int movimento_possivel (int l, int c, int tab[8][8]):** Checa se um movimento é possível para o cavalo, isto é, se ele não o leva para uma casa já antes visitada no

tabuleiro, nem para uma casa fora do tabuleiro de dimensão 8x8. Assim, se for de fato possível, ela retorna o valor **1**, senão, retorna o valor **-1**.

4) **int MenorQuantidade (int l, int c, int tab[][8], int r[][8][8], int x, int y):** Retorna o número do movimento (de 1 a 8), que o cavalo deve fazer a fim de ir para uma casa onde exista a menor quantidade de movimentos possíveis. Em caso de empate entre dois movimentos, ele escolhe o cujo correspondente número possui menor módulo. Caso não haja nenhum movimento possível, ela retorna o valor **-1**. Note, neste caso, que, para um movimento ser possível, ele deve:
   - **A)** Não levar o cavalo para um lugar fora do tabuleiro.
   - **B)** Não levar o cavalo para uma casa já visitada.
   - **C)** Atender às restrições, isto é, não levar o cavalo a fazer um movimento que o levará a ter que fazer *backtracking*.

Funcionamento

Assim que a função **void passeio (int x, int y)** é chamada, o meu algoritmo atribui os correspondentes valores de **l** e **c**, inicializa o array tabuleiro[8][8] com o valor 0, o array salvamov[64] com o valor 1 e o array restrição[8][8][8] com o valor 0 (isto é, com nenhuma restrição), a fim de evitar erros. A partir disso, ele entra no laço for (cont = 1; cont <= 64; cont++), onde ele começa a preencher o tabuleiro e só termina quando todas as casas foram preenchidas.

1) **Ida:** ocorre quando salvamov[cont-1] > 0, isto é, quando, para uma posição, o movimento foi possível. Assim, ele atribui o valor de cont à casa correspondente do tabuleiro e incrementa em uma unidade a variável contIdas++.

Na **primeira instância** deste laço, portanto, ele realizará a “Ida” mencionada e pulará o else. Por conseguinte, temos que **salvamov[cont] = MenorQuantidade(l, c, tabuleiro, restricao, l, c)** e chamamos **movimento(salvamov[cont], &l, &c)**, ou seja, o array salvamov irá receber o número do movimento que levará o cavalo para uma posição onde ele terá a menor quantidade de movimentos possíveis e, em seguida, o cavalo tenta fazer esse movimento. Se foi possível, então a “Ida” supracitada será novamente satisfeita; senão, ocorre o *backtracking*.

2) ***Backtracking*:** ocorre quando salvamov[cont-1] <=0. Por definição do meu código, esse salvamov[cont-1] será igual a -1, o que significa que, para aquela posição, não há nenhum outro movimento possível. Diante desse contexto, o código atuará da seguinte forma:

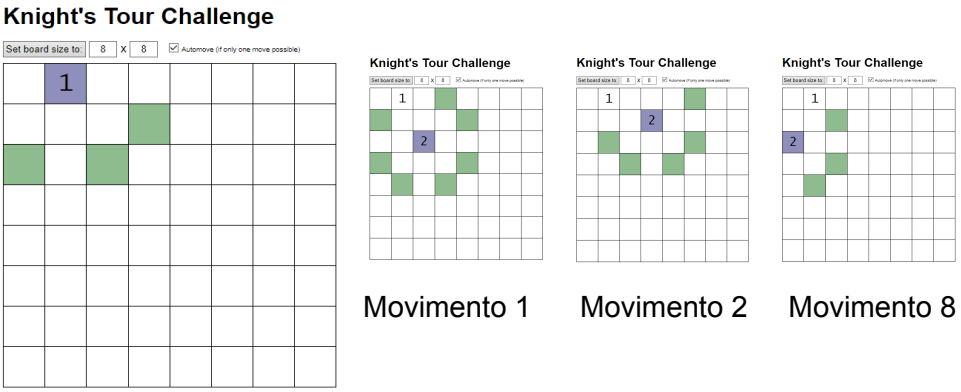
   - **A)** Zera todas as restrições da casa atual, pois ela não será mais a mesma.
   - **B)** Realiza o “antimovimento” correspondente, voltando para a casa anterior.
   - **C)** Uma vez na casa anterior, impomos sobre ela a restrição de que ela não deve realizar o movimento que a levou para a casa que a fez ter que fazer *backtracking*.

Assim, tentamos novamente fazer um movimento, diferente dos anteriores, para essa casa. Se houver outro movimento possível, então ele executa a "Ida", senão, ele executa o "*Backtracking*" novamente. Este processo é realizado repetidamente até que todas as casas tenham sido adequadamente preenchidas.

3) **Impressão:** Assim que o laço for mencionado tiver sido concluído, então o código imprime o tabuleiro resultante no arquivo "saida.txt".

Podemos ilustrar[2] esse funcionamento da seguinte forma:

Ida:



Movimento 1 Movimento 2 Movimento 8

Tomando como exemplo a inicialização na posição (1, 2), temos, seguindo a numeração de movimentos do meu código, que o cavalo só poderá realizar os movimentos 1, 2 ou 8. Nas imagens menores representadas à direita, podemos perceber que:

1) Fazendo o movimento 1, o cavalo irá para uma posição onde ele terá 7 outros possíveis movimentos.
2) Fazendo o movimento 2, o cavalo irá para uma posição onde ele terá 5 outros possíveis movimentos.
3) Fazendo o movimento 8, o cavalo irá para uma posição onde ele terá 3 outros possíveis movimentos.

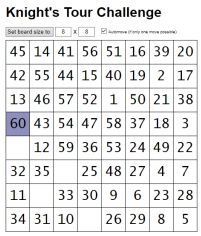
Ou seja, se chamarmos, nessa configuração, a função int MenorQuantidade (1, 2, tabuleiro, restricao, 1, 2), ela deverá retornar o valor 8, que é justamente o movimento que levará o cavalo para uma posição onde ele terá a menor quantidade de outros movimentos possíveis. Concomitantemente, salvamov[cont-1] receberá 8 e faremos o movimento 8.

***Backtracking*:** Tomemos como exemplo a inicialização na posição (3, 5), que utiliza *backtracking*. Note que ele irá, por tentativa e erro, ir e retroceder até que uma configuração de solução seja encontrada.

> **Comentário:** no meu código, a única inicialização que leva meu código a usar *backtracking* é, de fato, a (3, 5).

[2] Todas essas imagens foram feitas utilizando o site https://www.maths-resources.com/knights/.

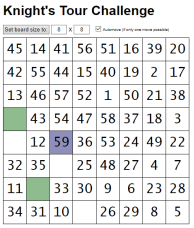
1)



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

Chegamos a um ponto em que não há mais possíveis movimentos e ainda não preenchemos todas as posições. Teremos, nesse caso, que salvamov[cont-1] será igual a **-1**.

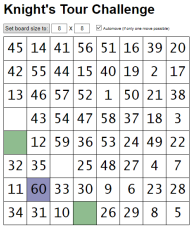
2)



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

A partir disso, meu código zera a posição onde estava o 60 e volta para a posição 59, colocando sobre ela a restrição de que ela não deverá fazer o movimento 6, que a levou a ter que fazer *backtracking,* como visto no passo anterior.

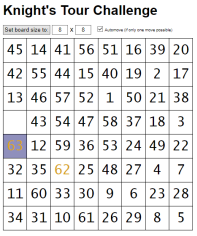
3)



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

Então, ele faz o outro movimento possível, o qual o faz entrar na configuração dessa imagem.

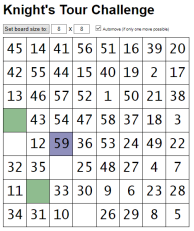
4)



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

Porém note que, a partir desse 60, ele também entra em uma outra situação de movimento impossível, para todos os casos.

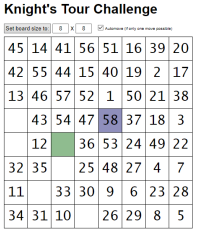
5)



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

Então, ele voltará de novo para a posição do 59, impondo então sobre ela a restrição do movimento 8. Note, entretanto, que, pelas restrições acumuladas para essa casa, o cavalo não poderá fazer nem o movimento 6, nem o movimento 8. Ou seja, não há movimentos possíveis.

6)



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

Então, novamente, ele realizará um *backtracking*, voltando para a posição do 58 e impondo sobre ela uma restrição de movimento. Como não haverá outro possível movimento, ele então voltará para a posição 57 e assim sucessivamente, por tentativa e erro, até encontrar uma solução

Referências

- https://www.codingninjas.com/codestudio/library/backtracking-the-knights-tour-problem
- https://www.maths-resources.com/knights/
- http://ws2.din.uem.br/~ademir/sbpo/sbpo2013/pdf/arq0328.pdf