Trabalho Prático - Knight's Tour & Backtracking

Aluno: Mateus Augusto Gomes

Explicação Geral

O problema principal no projeto para o Passeio do Cavalo é encontrar uma forma de retroceder sempre que não houver mais jogadas possíveis para a peça cavalo. Como não há uma fórmula fechada para determinar o passeio corretamente, o método de "tentativa e erro", nesse caso chamado de Backtracking, é necessário. Seus passos são resumidos em:

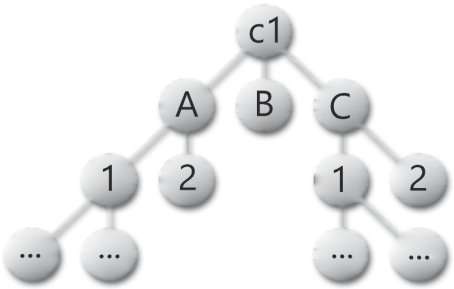
1 - Verificar, a cada movimento, se aquela jogada não irá fazer o cavalo sair do tabuleiro e se a casa de destino já foi visitada ou não.

- o passo 1 é coberto na função "int movimento_valido".

2 - Se não houver movimentos válidos, retroceder uma casa.

3 - Verificar, a cada movimento, se aquela jogada acaba de completar o tabuleiro, ou seja, a casa de número 64 do passeio.

Portanto, sempre que não houver jogadas possíveis o cavalo deve retroceder uma jogada feita antes e tentar outra jogada diferente daquela já feita. Tendo isso em mente, é possível inferir que o algoritmo deve se basear em um método recursivo, pois diferentes passeios serão possíveis para cada decisão de movimento. Assim, se uma casa tem n movimentos possíveis (0 <= n <= 8), isso implica em n chamadas recursivas. Ilustrando essa lógica, temos:



A casa c1 tem 3 movimentos possíveis (A, B e C) e cada um desses movimentos implicam em n movimentos possíveis partindo da nova posição. Esse árvore de possibilidades pode justamente ser entendida como a **Estrutura de Dados** que encontra a

solução do passeio do cavalo partindo de uma posição c1. Essa solução é encontrada no nível 64 dessa árvore.

Como o algoritmo se resume majoritariamente na função recursiva e em função auxiliares, a estrutura total do programa em um pseudo-código se resume em:

```
Procedure knight_tour (int board[M][M], int casa_atual, int pos_x, int pos_y, int casas, int back)

    board [pos_x][pos_y] = casa_atual
    casas_visitadas++

    if ( casa_atual == 64 ) then
        O tabuleiro esta preenchido, retorna 1 e encerra;
    end if

    //preenche movimentos_possiveis e retornas quantos existem
    contador = funcion movimentos_garantidos(movimentos_possiveis);
    //faz a euristica priorizando casas distantes do centro
    norma_euclideana(movimentos_possiveis);

        for ( int i = 0 ; i < contador ;  i++ )

            //chamo a funçao recursivamente para cada movimento possivel
            if (movimento_do_cavalo(++casa_atual, movimentos_possiveis)) then
                Return 1;
            else do
                Incremento o contador de casas retrocedidas
            end else

        end for

//desmarco a casa
board[pos_x][pos_y]=-1;
return 0;
end funcion
```

- *obs: "if (movimento_do_cavalo(++casa_atual, movimentos_possiveis))" esta simplificado por se tratar de um pseudocódigo. Existem outros parâmetros omitidos nesta e em outras funções, mas não afetam a lógica da explicação.*

Observe que o primeiro passo é marcar a casa atual como visitada. Isso pode ser feito pois essa função é chamada somente com movimentos possíveis. Eles foram obtidos anteriormente, quando usamos a função “void movimentos_garantidos”, que preenche um vetor e retorna um int.

- Fases da Implementação

Na implementação do programa sem o uso de uma heurística, os seguintes testes foram realizados:

Posição (4,8)

```
05 64 25 10 03 08 15 20
26 11 04 07 14 19 02 17
63 06 13 24 09 16 21 34
12 27 52 55 22 33 18 01
51 62 23 44 53 56 35 32
28 45 54 59 48 31 38 41
61 50 47 30 43 40 57 36
46 29 60 49 58 37 42 39
14318942 14318878
```

Posição (5,1)

```
07 64 25 14 05 10 19 12
26 15 06 09 18 13 04 21
63 08 17 24 03 20 11 44
16 27 02 51 30 43 22 37
01 62 29 58 23 38 45 42
28 57 50 31 52 41 36 39
61 32 55 48 59 34 53 46
56 49 60 33 54 47 40 35
59468789 59468725
```

Ou seja, sem nenhum critério o programa tentaria todas as casas aleatóriamente e, consequentemente, diversos retrocessos seriam necessários.

- Heurística Implementada -

Considerando que o algoritmo não faz uso de uma fórmula fechada para encontrar o passeio e sim um método como "Tentativa e Erro", é aconselhável encontrar alguma heurística que diminua a quantidade de retrocessos. Foi implementado a Heurística de Warnsdorff.

Considerando um tabuleiro 8x8 vazio, o cavalo pode ter uma quantidade diferente de movimentos dependendo de sua casa inicial. É visível que as casas centrais possuem mais possibilidades, pois não há possibilidades em que o cavalo saia do tabuleiro. Em raciocínio análogo, as casas laterais possuem poucas possibilidades de jogadas corretas. Por exemplo, as casas dos vértices do tabuleiro só possuem duas jogadas possíveis.

Portanto, seria útil usar primeiro as casas laterais durante o passeio do cavalo para evitar casos em que, por exemplo, a parte inferior do tabuleiro é preenchida, mas o cavalo se encontra sem movimentos possíveis (caso contrário, isso implicaria em diversos retrocessos). Para diferenciar uma casa lateral de uma central, é possível usar a norma euclidiana da sua coordenada, assim, a casa com maior norma euclidiana se encontra mais longe do centro. A norma euclidiana usada no algoritmo se diferencia da habitual somente pela subtração das coordenadas pelo valor 4 (ou seja,

dado coordenadas x e y, temos: $(x - 4)^2 + (y - 4)^2$). Essa subtraçãe foi feita como um tipo de tratamento para algumas excessões. Por conta da limitação estrutural do trabalho, não foi possível incluir a biblioteca math.h para usar a função sqrt, mas, neste caso, não há diferença em calcular a raiz ou não, pois o maior valor terá, consequentemente, a maior raiz.

Para calcular essa norma, foi implementado a função do tipo void norma_euclidiana_alt. Essa função utiliza o método de ordenação selection sort para ordenar de forma decrescente de norma euclidiana (porém, subtraindo 4 das coordenadas).

Comparação de tempo de execução e retrocessos

Com o intuito de comparar o tempo de execução do programa com e sem a heurística, foram realizados alguns testes no primeiro ambiente computacional especificado no leame.txt. Os seguintes resultados foram obtidos:

Sendo a posição inicial a casa (1,1) e sem o uso da heurística, o programa demorou cerca de 5 segundos e a quantidade de retrocessos foi 27.241.049.

01 10 23 64 07 04 13 18
24 63 08 03 12 17 06 15
09 02 11 22 05 14 19 32
62 25 40 43 20 31 16 51
39 44 21 58 41 50 33 30
26 61 42 47 36 29 52 55
45 38 59 28 57 54 49 34
60 27 46 37 48 35 56 53
27241113 27241049

Em contrapartida, partindo da mesma posição porém com o uso da heurística, o programa demorou cerca de 1 segundo e retrocedeu 23.746 vezes, ou seja, uma quantidade consideravelmente menor.

01 14 03 26 35 16 19 22
04 27 36 15 24 21 34 17
13 02 25 46 33 18 23 20
28 05 32 37 62 47 44 53
31 12 29 64 45 52 61 48
06 09 38 51 58 63 54 43
39 30 11 08 41 56 49 60
10 07 40 57 50 59 42 55
23810 23746

Em outro teste, sendo a posição inicial a casa (4,7) e com o uso da heurística, poucos retrocessos foram necessários para econtrar o passeio corretamente e o tempo de execução foi em torno de 1 segundo. Em contrapartida, sem o uso da heurística, o programa não encontrou solução mesmo após 12 minutos. Isso permite inferir que o tempo de execução pode variar drasticamente em posições diferentes e com uma implementação mais simples.

Solução para casa (4,7) encontrada com a otimização | Tempo de execução: menos de 1 segundo

08 05 10 33 30 03 38 35
11 32 07 04 37 34 29 02
06 09 54 31 52 49 36 39
55 12 51 48 57 64 01 28
46 19 56 53 50 59 40 63
13 16 47 58 43 62 27 24
20 45 18 15 22 25 60 41
17 14 21 44 61 42 23 26
2426 2362