UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Universidade Federal de Minas Gerais

Disciplina: Matemática Discreta

Prof.: Antônio Loureiro

Aluno: Marcos Antônio Dias Junior

Documentação do Algoritmo do Passeio do Cavalo

Solução Apresentada

Assim que a casa inicial é designada, antes de fazer o primeiro movimento, o algoritmo mapeia e registra os movimentos disponíveis de cada casa, a liberdade da casa que cada um destes movimentos leva, e a marginalidade de todas as casas. Sempre que um movimento é feito estes movimentos são atualizados para a casa que se ocupa no momento e casas adjacentes a ela.

Em linhas gerais, o procedimento utilizado para mover o cavalo é inicialmente sempre optar pelo movimento que leva para a casa com menos movimentos disponíveis e mais próximas das bordas - de forma a minimizar a probabilidade de haver casas ilhadas, principalmente antes dos estágios finais do passeio. Ele então atualiza os valores das variáveis das casas adjacentes e registra em qual passo aquele movimento foi tomado naquela casa. Em casa de backtrack ele sabe quais movimentos já foram testados.

O movimento de retorno a uma casa é sempre igual a 9 menos o movimento de ida.

Fases do Projeto

Definição da Heurística:

Quando o problema do passeio do cavalo foi apresentado eu inicialmente em implementar com uma heurística que sempre desse preferência a casa que levasse a próxima casa com maior número de movimentos disponíveis - de forma a tentar favorecer que, caso o cavalo chegasse em alguma casa sem saída, não fossem necessário retroceder muitas casas para encontrar outra opção de caminho. Porém, conversando com meus colegas, um deles comentou que havia visto uma estratégia parecida para este problema, mas que era precisamente o contrário: escolher a casa com menos movimentos válidos. Percebi que essa estratégia era mais produtiva, pois diminui muito eficientemente a chance de sobrar uma casa não visitada e impossível de ser acessada. Mais tarde descobriria que essa heurística se trata da Regra de Warnsdorff e é uma estratégia mais antiga que a computação em si, pois foi proposta em 1823.

Modelagem do Algoritmo e Funções:

O próximo passo foi conceber o algoritmo. Decidi que gostaria de fazer um código iterativo, porém optei por primeiro fazer uma solução recursiva (que me pareceu mais imediata) e então tentar transformar em iterativa. Depois de ter uma ideia geral daquilo que iria implementar, procurei seccionar este algoritmo em partes. Para isso usei papel de rascunho para conceber, definir e listar quais funções seriam necessárias e qual seria seus comportamentos. Apesar de

que nenhuma linha de código havia sido ainda escrita, todas as funções essenciais para resolução do problema foram concebidas neste momento. Eram elas:

int updateKnightMoves(x,y) - uma função recursiva que retorna o número de movimentos disponíveis para uma certa casa (x,y) e, para cada movimento possível armazena num vetor de movimentos qual a liberdade da casa que este movimento leva. Posteriormente mais um parâmetro foi adicionado a esta função para limitar a profundidade das recursões desta função.

int marginalidade(x,y) - função que, dado a coordenada da casa (x,y) e uma dimensão de tabuleiro, calcula um valor inteiro proporcional que cresce em função da proximidade da casa com bordas e cantos (corners) do tabuleiro. Este valor é zero na(s) casa(s) centrais e máximo nos quatro cantos do tabuleiro (superior esquerdo, superior direito, inferior direito e inferior esquerdo). Pela brevidade desta função, acabei implementando e testando ela primeiramente em C++, em outro desafio que também envolve tabuleiros e que havia feito nessa linguagem para o BeeCrowd.

coordenadas bestMove(x,y) - função que analisa os vetores retornados pela função anterior e decide o melhor movimento baseado na *Warnsdorff's Rule*. Em casos de empate, a maior marginalidade era usada como critério. Ou seja, retornava a coordenada da casa mais próxima às extremidades - valor obtido pela função anterior. Inicialmente essa função retornava um int obtido pelo produto do número da linha com a dimensão *n* do tabuleiro somado a coordenada da coluna. Ou seja: (**y * *n*) + x .** Mais tarde ela passou a retornar um tipo abstrato de dados, ***struct coordenadas***, contendo o par ordenado (x,y).

void setup() - É simplesmente a função que inicializa todas as variáveis utilizadas e atribui os valores iniciais para cada uma delas. Por exemplo, o valor estático de marginalidade de cada casa, os valores iniciais de liberdade de movimento, os movimentos possíveis (bem como a liberdade da casa destino de cada um destes movimentos) são todos pré-calculados neste método antes de cada execução.

void displayChessBoard() - Essa função eu já havia implementado por curiosidade previamente em C++ para dar saída numa matriz de tabuleiro para o desafio *Knight Moves* do BeeCrowd - que consistia em calcular o número mínimo de movimentos que um cavalo deve fazer para chegar de uma casa (x,y) a uma casa (w,z). Durante as férias acabei desenvolvendo este código para imprimir um tabuleiro com as quantidades mínimas de movimentos necessários para o cavalo alcançar cada casa, dada uma posição inicial.

void ride(x,y, backtracking) - Esta é função recursiva responsável por administrar todas as outras e levar o cavaleiro de casa em casa no passeio. Na prática é um método int, mas seu valor de retorno não é recebido por nenhuma variável. Este foi o método menos detalhadamente concebido antes de escrever o código e a última das funções essenciais a serem escritas. A concepção se limitou a um esboço de fluxograma, uma vez que inicialmente esta função, apesar de ser o método diretivo de cada passeio, basicamente concatenava e administrava as outras.

Exceto o **void ride(),** uma vez escritas, essas funções sofreram pouquíssimas alterações. O método **ride()**, por outro lado, sofreu várias alterações e a maior parte do tempo gasto para resolver o problema foi dispensado procurando implementar o backtracking nele.

Estrutura de Dados

Durante os primeiros estágios da implementação optei por usar apenas dois vetores como estrutura de dados. Um vetor, **tile[n][n],** que armazenava o passo em que o cavalo passou pela casa representada pelo **tile[x][y]** e outro vetor **mvmnt[i][j]** - onde **i** tinha tamanho igual ao número de casa nos tabuleiros (dimensão * dimensão) e **j** era sempre igual a 9 (os oito movimentos possíveis + 1). Neste último vetor, o primeiro espaço, **i**, indicava a casa do tabuleiro e o segundo índice, **j**, indicava a liberdade daquela casa para **j = 0** e a liberdade do movimento **j,** para **1 ≤ j ≤ 8.** Por exemplo, num tabuleiro 8 por 8:

tile[59][0] = 4. A casa **i = 59** está na linha de índice 7 (pois a divisão inteira $59 \div 8 = 7$).
A casa **i = 59** está na coluna de índice 3 (pois $59 \% 8 = 3$).
Há 4 movimentos possíveis nesta casa, pois **j = 0** indica a liberdade da casa.

tile[0][6] = 2. A casa **i = 0** está na linha e coluna de índices zero.
Ou seja, está na primeira linha e coluna. **j = 6** designa o 6º movimento.
O movimento 6 leva a uma casa com 2 de liberdade.

Com a complexificação do código, tornou-se mais viável criar um struct, **pos**, que reunisse esses e outros dados de maneira unificada e coesa. Isso porque qualquer sinal trocado nas expressões ou comandos faziam os vetores se desalinharem, tomando muito tempo apenas debugando código.

```
// estrutura de dados usada para cada casa do tabulerio
struct pos {

    int p;      // O passo em que cavalo visitou a casa
    int mvmnt[9];  // Vetor de movimentos possíveis. Cada índice é a liberdade do destino do movimento.
    // xemplo:  mvmnt[6] = 2  (Fazendo o movimento 8 chega-se a uma casa com 2 de acessibilidade/liberdade.)

    int persistent[9];  // guarda em que passo o movimento do índice foi feito.
    // Ex: persistent[3] = 32 (O movimento 3 já foi tentado anteriormente nessa casa no passo 32)

    int a;  // acessibilidade ou liberdade de movimentos possíveis a partir da casa
    int m;      // valor estático da marginalidade da casa (em função da posição e do tamanho do tabuleiro)

    int prevX;  // a coordenada x da casa do passo anterior
    int prevY;  // coordenda y da casa do passo anterior
} typedef pos;
```

Backtracking

No momento inicial do planejamento do código qual metodologia seria usada para fazer o backtrack não foi considerado - algo que mais tarde percebi ser um erro - intencionava-se apenas fazer um código que fizesse o passeio mais longo possível antes de chegar numa casa sem saída usando a heurística de Warnsdorff. De fato, o algoritmo visitava de 58 a 63 casas diferentes antes de chegar num *deadend* e precisar retornar. Em algumas casas excepcionais ele até terminava o passeio. Porém encontrar uma forma de introduzir o processo de backtracking com a dinâmica do passeio já bem definida se mostrou bastante complicado.

Não existe um for de movimentos possíveis para cada casa do passeio que testa todas as opções de movimentação para cada casa, não existe qualquer for neste sentido. Ao invés disso, sempre que o cavalo chega numa casa sem saída, ele usa a mesma função, **ride(x,y, backtracking),** para retornar a casa anterior (cujas coordenadas são registradas no ato da entrada naquela casa). O parâmetro **backtracking** do método é definido para **true** sempre que ele está cavalgando de volta, de forma que a função ignora o movimento já tentado e cavalga em outra direção com o parâmetro **backtracking** definido para o padrão (falso), caso haja opções não testadas naquela casa.

A solução encontrada para manter registro dos caminhos já tentados foi manter em cada casa um vetor de variáveis persistentes, **persistent[9],** que quando alguma tentativa de movimento fosse executada naquela casa, o número do passo que executou o movimento fosse registrado. Dessa forma, se a tentativa falhasse, o algoritmo teria registrado que aquele movimento já havia sido tentado naquela casa naquele mesmo passo. Porém, aquele mesmo movimento pode ser tentado em outro estágio do passeio se a casa estiver livre - ou seja, em outro passo.

-

Finalização

A parte final do projeto foi limpar o código e comentar as partes que já não haviam sido comentadas - bem como as informações que eram notificadas no terminal durante a execução do código. E por fim ajustar a saída ao formato determinado e colocar o código para dar saída em arquivo.

Alguns testes foram feitos em Debian e Ubunto e com diferentes compiladores (GCC e CLANG) e também testes de escalamento das casas dos tabuleiros e de comparação de eficiência com outras implementações. Em termos de eficiência o algoritmo pareceu mais rápido que alguns outros disponíveis na internet porque ele só checa a liberdade das casas quando o algoritmo começa e quando o cavalo precisa fazer backtrack até uma casa com outra opção de movimento. Isso porque ele atualiza a liberdade das casas adjacentes sem precisar checar a medida que ele avança de casa em casa.

A escalabilidade, porém, enfraquece a medida que se aumenta muito o número de casas. Nestas situações ele passa a não ser sempre capaz de encontrar um passeio em algumas posições e dimensões de tabuleiro. Por exemplo, nas dimensões 31x31 ele não é capaz de achar um passeio partindo da casa (0,0). Em outras casas consegue. Na Dimensão 100x100 ele é capaz de eficientemente achar um passeio na casa (0,0), mas não na casa (2,3). Isso porque em tabuleiros maiores há mais caminhos diferentes que passam pela mesma casa no mesmo passo o que provavelmente confunde o registro do backtrack como implementado atualmente.