O Cavalo Perdido

Algoritmos e estrutura de dados II

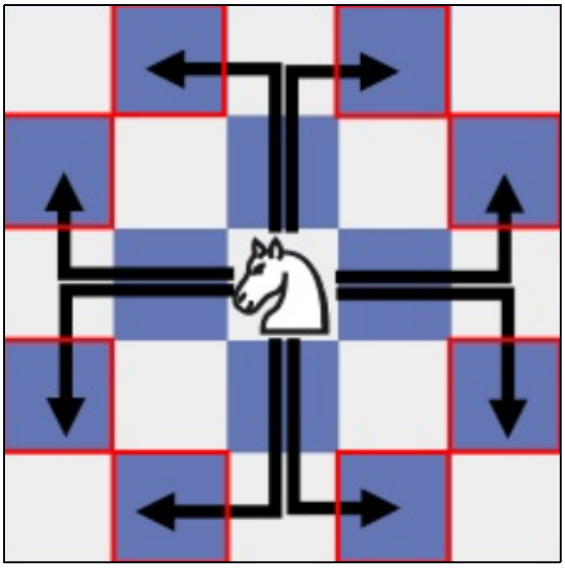
João Vitor Dall Agnol Fernandes - 19201612 9
Escola Politécnica - PUC-RS

1. Introdução

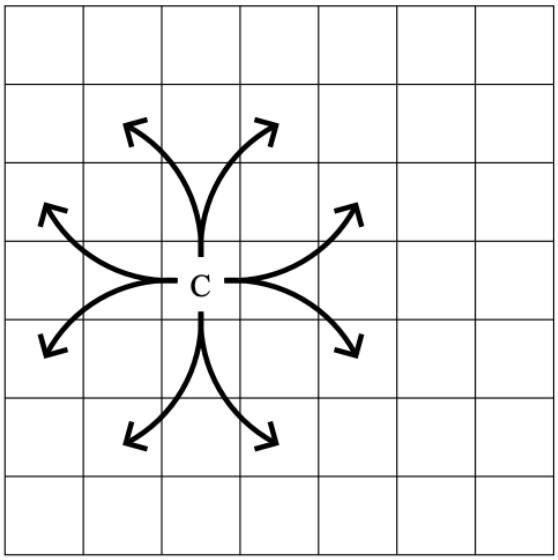
Esse relatório surge a partir da proposta de um desafio de xadrez espacial, onde uma peça semelhante a do cavalo no xadrez regular deve traçar sua jornada em direção à saída do labirinto.

2. O problema em questão

A saga do cavaleiro solitário pelo labirinto se da na forma de um tabuleiro.
No xadrez normal, a inusitada peça do cavalo anda pelo campo em L[1], executando o mesmo padrão de movimentação do nosso cavaleiro[2].



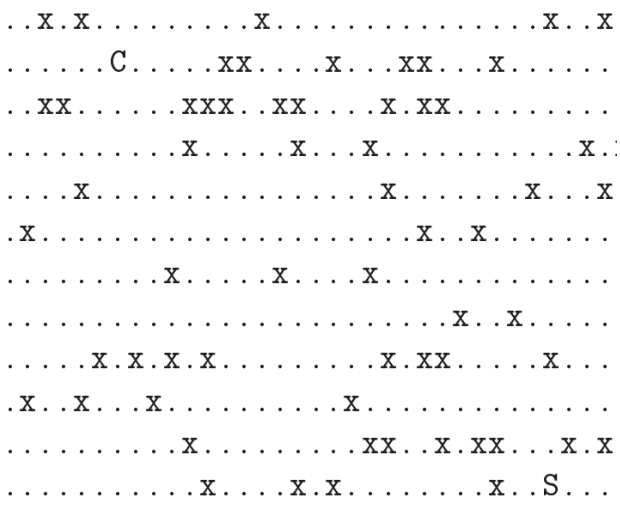
[1]



[2]

Dez tabuleiros[3] nos foram entregues e devemos garantir que em todos eles o cavaleiro, começando no ‘C', chegue na casa de saída, marcada por um ’S’, no menor tempo possível e somente executando o movimento em L.

O tabuleiro também esta repleto de armadilhas onde o cavaleiro não deve cair, estes campos estão marcados com um ‘X’. Uma característica singular do jogo é que o tabuleiro não possui limites, isso é, caso o cavalo faça um movimento que o leve



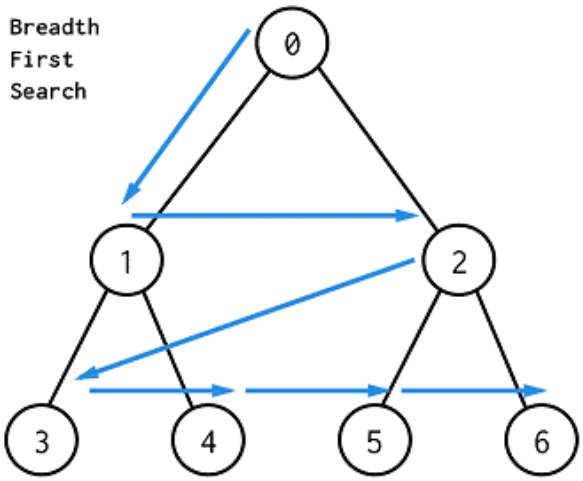
[3]

para fora do campo de batalha pela esquerda, ele imediatamente reaparece do lado direito. O mesmo ocorre se o limite superior for cruzado, caso onde o cavalo reaparece no lado debaixo do campo. Essencialmente o tabuleiro é infinito e, alem de tudo, pode não ter uma saída.

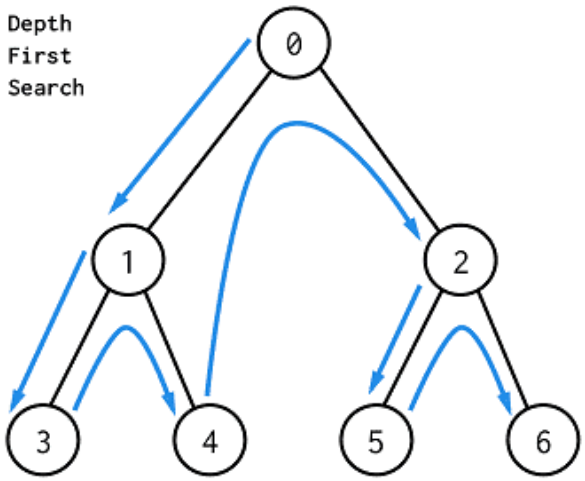
3. Modelação do problema

Para pensar em uma solução para o problema antes devemos descobrir como será feita a implementação do desafio. Para isso, foi decidido usar uma classe Terminal que fará toda a leitura do tabuleiro e a configuração do jogo. Um vetor bidimensional de strings foi escolhido para cumprir o papel do tabuleiro. A partir daqui podemos pensar em como encontraremos o caminho mais rápido até a saída. Conhecendo os movimentos do cavalo e sabendo que o tabuleiro é uma matriz bidimensional, podemos descobrir, através de um algoritmo de busca, a saída do labirinto. Para tal existem algumas opções: a busca em profundidade[4.1] e a busca em largura[4.2]. A primeira usa uma estrutura de pilha para encontrar a saída do tabuleiro mas, pelo seu comportamento, não é adequada para chegar na solução mais rápida. A busca em profundidade ja declara no próprio nome como funciona. Itera-se por cada caminho possível, até o próximo movimento não ser permitido, antes de realizar a busca no próximo caminho. Então, por esse motivo, foi decidido que a rota mais rápida até a saída será encontrada com a busca em largura (*bfs*). Sua forma de operação faz a busca por todo o nível antes de se aprofundar, isto é, busca para cada conjunto de movimentos seguintes se algum deles leva à saída e, caso negativo, busca no próximo conjunto de movimentos.

A *bfs* utiliza uma queue para processar a casa atual do cavaleiro baseado em quão longe ele esta do começo do labirinto (em qual profundidade ou, também, o numero do movimento). Ao adicionarmos uma fila contadora auxiliar, podemos inserir tanto o movimento do cavalo na fila da busca, quanto um inteiro na fila contadora. Quando a vitoria for atingida, guardamos o valor atual da fila contadora e assim saberemos o menor caminho possível até a saída do labirinto.

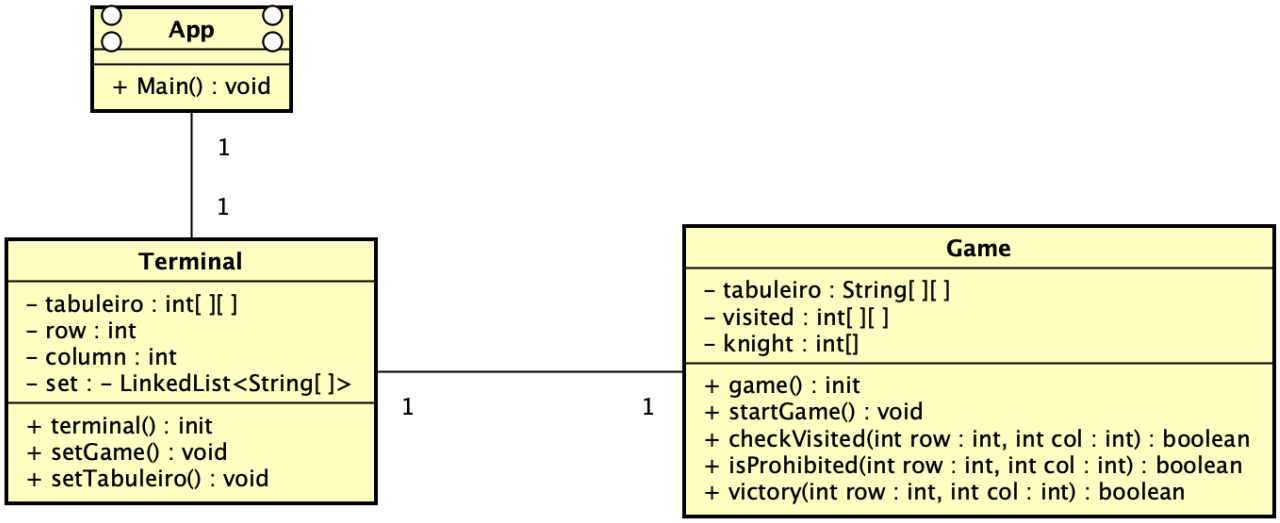


[4.1]



[4.2]

4. Solução e diagrama de classe



[5]

O programa começa no App, criando uma instancia de terminal. Essa instancia será encarregada da operação do sistema. Suas responsabilidades são:

A. Leitura do arquivo do caso de teste
B. Configurar o tabuleiro através do método setTabuleiro[5]
C. Inserir o cavaleiro no campo e começar o programa usando a função setGame[6]

```
//set tamanho do tabuleiro
row = set.size();
column = set.get(0).length;
tabuleiro = new String[column][row];

//preenche o tabuleiro
int j = 0;
for(String[] s : set) {
    tabuleiro[j] = s;
    ++j;
```

[6]

```
//busca pela casa contendo "C"
int[] knight = {};
for (int i = 0; i <row; ++i) {
    for(int j = 0; j < column; ++j) {
        if (tabuleiro[i][j].equals("C")) {
            knight = new int[] {i, j};
        }
    }
}
Game g = new Game(tabuleiro, knight);
```

[7]

Na sequencia, o terminal simplesmente cria uma instancia de Game que, por sua vez, chama o método startGame. Agora a classe Game entra em ação. É ela que executa a lógica por trás da solução. A classe é delegada por fazer:

A. A movimentação do cavaleiro pelas casas do tabuleiro. Definitivamente um desafio, foi solucionada analisando novamente o diagrama de movimento do cavalo [1] e [2] e mapeando a quantidade de linhas e colunas alteradas para cada uma das 8 possíveis direções. O resultado é observado nos vetores de movimentos[7.1], onde cada orientação possui um valor exato à ser adicionado ao index atual da posição do cavaleiro para obter o index dos próximos saltos.

```
//movement n        : 1, 2, 3, 4, 5, 6, 7, 8
static int[] mRow = {-2,-1,+1,+2,+2,+1,-1,-2};
static int[] mCol = {+1,+2,+2,+1,-1,-2,-2,-1};
```

[8.1]. Vetores de movimento.

```
public Game(String[][] tabuleiro, int[] knight) {
    this.tabuleiro = tabuleiro;
    visited = new int[tabuleiro.length][tabuleiro[0].length];
    this.knight = knight;
    startGame();
```

[8.2].Variáveis de Game.

B. A execução da interface Queue[7.3]. Assim que o método startGame for chamado, o elemento x da nossa solução entra no campo. A busca em largura foi aqui implementada através de um laço while que pega a próxima ordem da fila - ou o próximo movimento do cavaleiro - e verifica se o campo ja foi visitado[7.5] ou se é uma armadilha[7.6]. Se for o caso, encerra somente a execução atual do laço while.

Por fim, executa o método victory[7.7] para descobrir se a saída foi encontrada e, caso retorno seja true, imprime no terminal a saída com a posição do nosso cavaleiro e o valor da sua posição atual.

```
//queue interface
Queue<String> queue = new LinkedList<>();
```

[8.3]

```
//check boundaries
newRow = checkRowBoundaries(newRow);
newCol = checkColBoundaries(newCol);
//check if tile has been visited before
if (!checkVisited(newRow, newCol))
    continue;
//check if tile is prohibited
if (!isProhibited(newRow, newCol))
    continue;
//check if found exit
if (victory(newRow, newCol)) {
    System.out.println("Parabéns, o cava
    System.out.println(count);
    break;
}
```

[8.4]

```
//marca na cola de visitados
public boolean checkVisited(int newRow, int newCol) {
    if(visited[newRow][newCol] == 1) {
        return false;
    }
    visited[newRow][newCol] = 1;
    return true ;
```

[8.5]

```
//check if tile is marked with an x
public boolean isProhibited(int newRow, int newCol) {
    if (tabuleiro[newRow][newCol].equals("x")) {
        return false;
    }
    return true;
```

[8.6]

```
//check if won
public boolean victory(int newRow, int newCol) {
    if (tabuleiro[newRow][newCol].equals("S")) {
        return true;
    }
    return false;
```

[8.7]

4.1. Resultados

Os outputs de cada caso teste[7.8], contendo o numero de movimentos necessários para encontrar a saída, bem como o índex da célula final que o cavaleiro se encontra, pode ser observado abaixo.

```
caso100.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 92 ] [ 22 ] em 68 movimentos!
caso150.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 48 ] [ 98 ] em 64 movimentos!
caso200.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 105 ] [ 193 ] em 108 movimentos!
caso250.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 10 ] [ 185 ] em 156 movimentos!
caso300.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 179 ] [ 289 ] em 197 movimentos!
caso350.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 318 ] [ 241 ] em 170 movimentos!
caso400.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 359 ] [ 187 ] em 185 movimentos!
caso450.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 405 ] [ 59 ] em 186 movimentos!
caso500.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 374 ] [ 369 ] em 225 movimentos!
caso550.txt: O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> S [ 21 ] [ 103 ] em 233 movimentos!
```

[8.8]

Sua chamada se deu através de um print, dentro do if (victory), com o uso das seguintes variaveis:

```
System.out.println("O cavaleiro chegou no campo de saida S [ linha ][ coluna ] -> " +
tabuleiro[newRow][newCol] + " [ " + newRow + " ] [ " + newCol + " ]" +
"em " +  counter.remove() + " movimentos!");
```

5. Conclusão

Com base no nível de dificuldade da implementação, na movimentação do cavalo e nos detalhes dos limites do tabuleiro, pode-se dizer que o cavalo perdido representou um grande desafio. Com ajuda da dupla Busca em Largura e Queue, capazes de operar todos os conjuntos de movimentos possíveis, sempre armazenando a profundidade do conjunto, nosso cavaleiro conseguiu encontrar a saída mais rápida em todos os casos de teste.