O Cavalo Perdido

Gabriela Panta Zorzo

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 – 90.619-900 – Porto Alegre – RS – Brasil

gabriela.zorzo@edu.pucrs.br

Resumo. *Este artigo apresenta uma solução para calcular o menor número de movimentos possíveis de um cavalo em um tabuleiro de xadrez, para que ele chegue em uma posição específica. Ele traz a descrição do problema bem como a sua análise, uma opção de solução, e o pseudo-código dos algoritmos implementados usando busca em largura em uma matriz. São testados 10 casos fornecidos e é feita a análise dos resultados obtidos para os mesmos. Este artigo compõe a disciplina de Algoritmos e Estruturas de Dados II, da Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.*

Introdução

Um tabuleiro de xadrez (Figura 1) pode ser facilmente representado por uma matriz, onde as posições verticais são representadas por colunas e as posições horizontais são representadas por linhas. Assim, o cavalo neste tabuleiro de xadrez está posicionado na linha c, coluna 3, sendo possível definir a sua posição "P" como P(c,3).

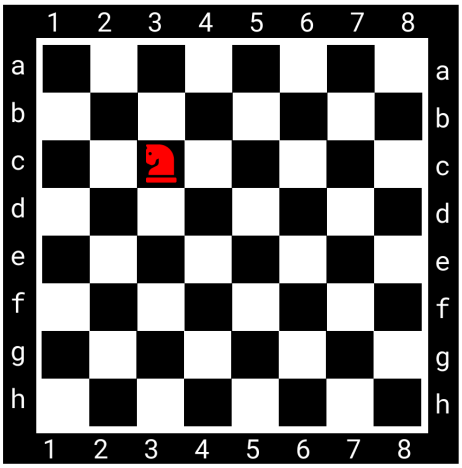

Figura 1. Tabuleiro de xadrez

Este artigo irá tratar um problema de xadrez espacial com uma única peça de xadrez: o cavalo. Em cada tabuleiro o cavalo inicia em uma posição diferente e deve chegar em uma posição específica, denominada saída, com o menor número de movimentos possível. Para solucionar este problema foi usada a estrutura de uma matriz bidimensional, realizando os caminhamentos do cavalo até a saída através da busca em largura [1].

O Problema

O problema [2] consiste em um desafio de xadrez espacial: foram fornecidos 10 tabuleiros de tamanhos distintos com um único cavalo posicionado em cada um deles. O desafio está em mover o cavalo até uma posição denominada saída com o menor número de movimentos possíveis. Existem algumas casas onde o cavalo não pode pisar de forma alguma, essas casas estão marcadas com a letra "x". A forma como os tabuleiros são representados pode ser observada na Figura 2.

```
x x x x x x x x x x
x C x x x x x x x x
x x x . x x x x x x
x x x x x . x x x x
x . x x x x x x x x
x x x x . x x x x x
. x S x x x x x x x
x x . x x x x x x x
. x x x x x x x x x
x x . x x x x x x x
```

Figura 2. Caso de teste

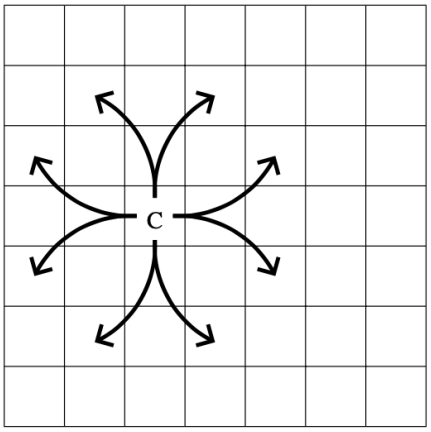

Figura 3. Movimentação do cavalo

No xadrez espacial o tabuleiro é toroidal e infinito, ou seja, os lados do tabuleiro se encostam de forma que se o cavalo sair pelo lado direito ele entra pelo lado esquerdo, e vice-versa. O mesmo acontece para a borda superior e inferior.

Sabendo que a posição inicial do cavalo é marcada pela letra "C", que as casas onde ele pode pisar estão marcadas com ".", e que o cavalo se move de acordo com a Figura 3, é necessário descobrir o menor número de movimentos necessários para que ele encontre a saída "S"em cada um dos casos de teste.

Ao todo, são 10 casos de teste fornecidos com tamanhos de tabuleiro diferentes: 100x100, 150x150, 200x200, 250x250, 300x300, 350x350, 400x400, 450x450, 500x500, 550x550. Estes casos de teste são fornecidos em um arquivo de texto como o caso de teste de teste da Figura 2, representado abaixo.

```
xxxxxxxxxx
xCxxxxxxxx
xxx.xxxxxx
xxxxx.xxxx
x.xxxxxxxx
xxxx.xxxxx
.xSxxxxxxx
xx.xxxxxxx
.xxxxxxxxx
xx.xxxxxxx
```

Análise e modelagem do problema

Ao olhar a forma como os tabuleiros de teste são representados (por exemplo, Figura 2), é possível ver a semelhança do modelo com uma matriz (por exemplo, Figura 4). Desta forma,

cada posição do tabuleiro pode ser encontrada através de um valor de linha e um valor de coluna. A escolha da representação do problema através de uma matriz se deu pela semelhança ao tabuleiro e pela facilidade de calcular quais são os próximos movimentos que o cavalo pode fazer através da posição em que ele se encontra.

	0	1	2	3	4	5	6	7	8	9	
0	x	x	x	x	x	x	x	x	x	x	0
1	x	C	x	x	x	x	x	x	x	x	1
2	x	x	x	.	x	x	x	x	x	x	2
3	x	x	x	x	x	.	x	x	x	x	3
4	x	.	x	x	x	x	x	x	x	x	4
5	x	x	x	x	.	x	x	x	x	x	5
6	.	x	S	x	x	x	x	x	x	x	6
7	x	x	.	x	x	x	x	x	x	x	7
8	.	x	x	x	x	x	x	x	x	x	8
9	x	x	.	x	x	x	x	x	x	x	9
	0	1	2	3	4	5	6	7	8	9	

Figura 4. Tabuleiro em uma matriz

Além disso, outra informação que o problema nos traz é que o tabuleiro é toroidal e infinito (Figura 5), então ao sair por um lado do tabuleiro, o cavalo entraria novamente no tabuleiro pelo lado oposto. Para melhor visualização deste cenário, vamos começar a pensar nos movimentos do cavalo. Assim como no xadrez tradicional, essa peça caminha em "L", então ilustrando os possíveis movimentos obtém-se a Figura 6.

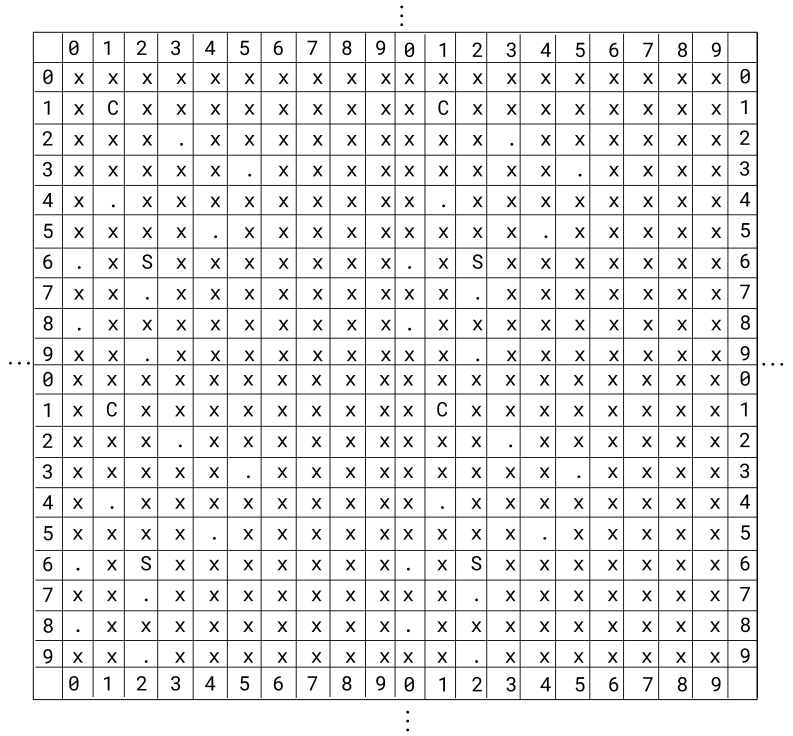

Figura 5. Matriz da Figura 4 representada como toroidal infinita

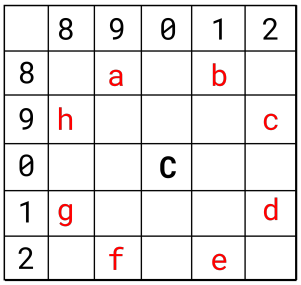

Figura 6. Movimentos possíveis para o cavalo na posição P(0,0)

O movimento necessário para chegar em cada uma das posições mostradas na Figura 6, é calculado conforme mostrado abaixo, com “linha” sendo a linha onde o cavalo está posicionado, e “coluna” sendo a coluna onde ele está posicionado.

```
a = (linha-2, coluna-1)
b = (linha-2, coluna+1)
c = (linha-1, coluna+2)
d = (linha+1, coluna+2)
e = (linha+2, coluna+1)
f = (linha+2, coluna-1)
g = (linha+1, coluna-2)
h = (linha-1, coluna-2)
```

Porém, como o tabuleiro é toroidal, é necessário tratar os casos das bordas. Quando o valor calculado for menor do que zero, o novo valor é o tamanho do tabuleiro mais o valor calculado. Já quando o valor exceder o tamanho do tabuleiro, o novo valor é o tamanho calculado menos o tamanho do tabuleiro. Este tratamento é feito através do trecho de código abaixo:

```
if (i < 0){
        i = tamanho + i;
} else if (i > tamanho - 1){
        i = i - tamanho;
}
```

Com base nesses cálculos, para um cavalo posicionado no canto superior esquerdo do tabuleiro P(0,0), obtêm-se as posições demonstradas na Figura 6. Neste caso é possível entender melhor como o tabuleiro toroidal funciona, pois para os casos “a”, “h”, “g” e “f” o cavalo precisa entrar novamente pelo lado oposto do tabuleiro. Ainda é necessário fazer a conferência de cada movimento para saber se a casa onde o cavalo irá pisar é permitida, conferindo se a casa possui um ponto “.” e não um “x”.

Como o objetivo principal é encontrar a saída “S” com o menor número de movimentos possíveis, é preciso realizar uma busca pelo tabuleiro. Existem algumas formas de fazer uma busca em uma matriz, como por exemplo, busca em profundidade e busca em largura.

Para realizar a busca em profundidade, para cada caminho possível a partir de um movimento do cavalo, ele deve seguir até o fim daquele caminho para tentar encontrar a saída, voltando para analisar os próximos caminhos somente quando o anterior for todo percorrido. Neste tipo de busca, seria preciso guardar o número de movimentos feitos em cada caminho no qual a saída foi encontrada e comparar no final para saber qual foi o menor.

Já a busca em largura funciona diferente, pois para cada posição que o cavalo se encontra, são analisadas todas as próximas posições para as quais ele pode ir imediatamente. Caso a

saída não seja encontrada em nenhuma delas, são analisadas todas as próximas posições a partir das já encontradas. Assim, a busca vai se espalhando de forma radial até que a saída seja encontrada. No momento em que ela for encontrada, só é necessário saber quantos movimentos o cavalo precisou para andar da posição inicial até ela, pois este já será o menor caminho possível.

Pensando nessas duas possibilidades de busca e na complexidade de implementar cada uma delas, para resolver este desafio foi escolhida a busca em largura [1]. Os métodos implementados para encontrar a solução para cada um dos tabuleiros fornecidos serão descritos a seguir.

Solução

O primeiro passo para começar a resolver o desafio é realizar a leitura dos casos de teste colocando-os em uma matriz. Para isso, foi determinado o tamanho do tabuleiro (*tamanho*) através do número de linhas lidas (*linhas*) no arquivo. Posteriormente cada posição do tabuleiro recebeu um caractere das linhas lidas conforme o algoritmo abaixo.

```
tabuleiro = new char[tamanho][tamanho];
for (int i = 0; i<tamanho; i++){
    for (int j = 0; j<tamanho; j++){
        String linha = linhas.get(i);
        tabuleiro[i][j] = linha.charAt(j);
    }
}
```

Para evitar que o cavalo ande em círculos voltando para posições que ele já conhece, foi inicializado um tabuleiro de booleanos com o mesmo tamanho da matriz do tabuleiro (*tamanho*) para atualizar quais posições ele já visitou (*true*) e conferir quais ele ainda não conhece (*false*). Estas posições serão atualizadas conforme o cavalo for pisando em casas novas ao longo da busca em largura que será realizada mais a frente.

Depois de já ter os dois tabuleiros inicializados, é a hora de encontrar o cavalo na sua posição inicial. Isto é feito percorrendo todas as linhas e colunas da matriz, até que seja encontrado "C" em uma das posições do tabuleiro, retornando qual é esta posição. Será preciso também um método para calcular as próximas posições que o cavalo pode visitar. Este método é chamado de *posicoesParaVisitar*, e recebe a posição atual do cavalo como parâmetro, retornando um array com as 8 próximas posições calculadas conforme o trecho de código a seguir. O método *valorPosicao* é chamado para realizar os ajustes de bordas caso sejo necessário, conforme visto na seção anterior.

```
int[] p1 = {valorPosicao(posicao[0]-1), valorPosicao(posicao[1]+2)};
int[] p2 = {valorPosicao(posicao[0]-1), valorPosicao(posicao[1]-2)};
int[] p3 = {valorPosicao(posicao[0]-2), valorPosicao(posicao[1]-1)};
int[] p4 = {valorPosicao(posicao[0]-2), valorPosicao(posicao[1]+1)};
int[] p5 = {valorPosicao(posicao[0]+2), valorPosicao(posicao[1]-1)};
int[] p6 = {valorPosicao(posicao[0]+2), valorPosicao(posicao[1]+1)};
int[] p7 = {valorPosicao(posicao[0]+1), valorPosicao(posicao[1]+2)};
int[] p8 = {valorPosicao(posicao[0]+1), valorPosicao(posicao[1]-2)};
```

Com a posição inicial do cavalo encontrada e o método para calcular as próximas posições que ele pode visitar definido, a busca em largura pode ser iniciada. A busca em largura irá retornar um *array* com três elementos: a posição da saída composta por linha e coluna, e o número de movimentos que o cavalo realizou até aquele certo ponto. São inicializadas duas listas para auxiliar na busca: uma para encontrar as próximas posições para serem

visitadas (*proximasPosicoes*) a partir da posição que o cavalo se encontra calculadas através do método *posicoesParaVisitar*, e outra para armazenar todas as posições que já foram visitadas com o número de movimentos que o cavalo precisou para chegar até elas (*jaVisitou*). Também são usados dois vetores: um para pegar a posição inicial do cavalo e inicializar o número de movimentos do mesmo (*x*), e outro para armazenar o resultado.

```
List<int[]> proximasPosicoes;
List<int[]> jaVisitou = new ArrayList<int[]>();
int x[] = {posicao[0], posicao[1], 0}; // posição inicial do cavalo
int resultado[] = {-1, -1, -1}; // resultado caso não encontre a saída
```

O método começa então adicionando a posição do cavalo na lista de posições visitadas, marcando ela na matriz de booleanos como *true*.

```
jaVisitou.add(x); // adiciona a posição inicial na lista de visitadas
visitado[x[0]][x[1]] = true; // marca a posição inicial como visitada
```

Posteriormente todos os elementos da lista *jaVisitou* serão analisados para tentar encontrar a saída. Isto será feito retirando sempre a primeira posição da lista e calculando quais as próximas posições que o cavalo pode ir a partir dela. Para cada uma das próximas posições possíveis é feita a conferência para ver se o cavalo pode pisar nela e se ela ainda não foi visitada, ou se a saída foi encontrada. Se a posição for válida e ainda não foi conhecida, ela vai ser adicionada no fim da lista *jaVisitou*. Isso acontece repetidamente enquanto a lista ainda tiver elementos e a saída não for encontrada. Este processo descrito pode ser melhor compreendido analisando o código abaixo.

```
while (!jaVisitou.isEmpty()){
    int p[] = jaVisitou.get(0); // pega a primeira posição da lista

    jaVisitou.remove(p); // remove a primeira posição da lista

    proximasPosicoes = posicoesParaVisitar(p); // posições para visitar

    // para as 8 posições possíveis a partir de "p"
    for(int j = 0; j < 8; j++){
        // pega as próximas posições para aquela posição
        int p2[] = proximasPosicoes.get(j);

        // confere se é uma posição válida e não conhecida
        if (tabuleiro[p2[0]][p2[1]] == '.' && !visitado[p2[0]][p2[1]]){
            // p[2]+1 indica que foi necessário +1 movimento do cavalo
            int y[] = {p2[0], p2[1], p[2] + 1};
            jaVisitou.add(y); // adiciona a posição para ser visitada
            visitado[p2[0]][p2[1]] = true; // marca como visitada
        } else if (saida(p2)){ // se encontrou a saída
            resultado[0] = p2[0];
            resultado[1] = p2[1];
            resultado[2] = p[2] + 1;
            return resultado; // retorna a saída e o número de movimentos
        }
    }
}
```

Como uma opção de teste, o exemplo da Figura 2 foi usado, pois é um caso de tabuleiro mais simples que os 10 casos de teste fornecidos. O resultado obtido para este exemplo está de acordo com o esperado, e pode ser observado a seguir.

```
O cavalo está em (1,1)
A saída foi encontrada na posição (6,2)
Em 4 movimentos.
```

Resultados

Para cada um dos casos de teste, foram impressas a posição inicial do cavalo "C" e a posição onde a saída "S" foi encontrada. Isso foi feito para que fosse possível analisar nos tabuleiros fornecidos e conferir se os valores estão de acordo com o esperado. Com todos os valores de acordo, o menor número de movimentos para que o cavalo encontra a saída em cada um dos 10 tabuleiros pode ser visto na Tabela 1.

Caso	100	150	200	250	300	350	400	450	500	550
Posição inicial do cavalo	(30,66)	(119,87)	(14,118)	(121,20)	(124,291)	(174,65)	(120,377)	(223,325)	(477,117)	(298,267)
Posição da saída	(99,22)	(48,98)	(105,193)	(10,185)	(179,289)	(318,241)	(359,187)	(405,59)	(374,369)	(21,103)
Movimentos necessários	68	64	108	156	197	170	185	186	225	233

Tabela 1. Movimentos necessários para encontrar a saída

Em todos os casos de teste foi possível movimentar o cavalo até a saída. O número de movimentos necessários para o cavalo encontrar a saída depende muito da posição inicial do cavalo, da posição da saída e das possibilidades de caminho do cavalo até ela. Devido a isso, em algumas situações foram necessários menos movimentos em um tabuleiro maior do que em um menor, como é o caso dos tabuleiros 100 e 150, e dos tabuleiros 250 e 300.

Como o número de movimentos necessários para encontrar a saída não depende do tamanho do tabuleiro e sim dos fatores ditos acima, o mesmo serve para o tempo de execução do algoritmo. Assim, o tempo para cada caso é irrelevante, mas em nenhum dos tabuleiros demorou mais de 62000 nanosegundos para realizar a busca em largura.

Conclusão

Ao analisar os resultados obtidos e o tempo máximo para obtê-los, a busca por largura foi uma solução adequada para resolver o desafio. Embora o número de movimentos necessários para encontrar a saída não dependa exclusivamente do tamanho do tabuleiro, caso tivesse sido optado pelo algoritmo de busca em profundidade é bem provável que o número de caminhos possíveis para serem percorridos aumentasse junto com os tabuleiros. O mesmo não acontece com a busca em largura, uma vez que a cada posição nova descoberta são abertas somente mais 8 posições de cada vez, até que a saída seja encontrada. Caso o cavalo estivesse a um mesmo número de movimentos da saída, o custo para que o algoritmo de busca em largura encontre este caminho é o mesmo, indepentente do tamanho do tabuleiro.

Referências

[1] Kelvin Jose. *Graph Theory — BFS Shortest Path on a Grid*, 2021. Disponível em: https://towardsdatascience.com/graph-theory-bfs-shortest-path-problem-on-a-grid-1437d5cb4023 Acessado em: 15.06.2021.

[2] João Batista Souza de Oliveira. *O Cavalo Perdido*, 2021. Disponibilizado para os alunos da turma de Algoritmos e Estrutura de Dados II da PUCRS em Maio de 2021.