O cavalo perdido

Elias Matos Garcia

23 de junho de 2021

Resumo

No decorrer deste trabalho será proposta uma alternativa de solução para um problema apresentado na disciplina de "Algoritmos e Estruturas de Dados II", que se trata de descobrir a quantidade mínima de movimentos necessários, à partir de um ponto que representa um cavalo, para chegar em um outro ponto, chamado de 'Saída', nos casos em que isso é possível, em uma série de tabuleiros toroidais e infinitos. Os movimentos que podem ser feitos são padronizados (são iguais os que um cavalo realiza no xadrez).

E, ao final do artigo, serão mostrados os resultados obtidos à partir do algoritmo aqui descrito para cada um dos casos fornecidos pelo professor.

1 Introdução

Em um tabuleiro de xadrez espacial, que é um tabuleiro de xadrez toroidal e infinito (se uma peça sair pelo lado esquerdo, entra pelo lado direito e vice-versa, a mesma coisa acontece para os lados de cima e de baixo do tabuleiro) existem vários pontos que aqui serão chamados de posições.

Foram recebidos vários arquivos de entrada, onde cada um simula um destes tabuleiros e neles se encontram algumas posições importantes, uma é um ponto que é chamado de 'Saída' e outra é o ponto onde se encontra um cavalo que deseja alcançar o ponto da 'Saída', além delas existem posições que são lugares para onde o cavalo pode caminhar e posições as quais ele não pode caminhar de forma alguma. Todas estas posições são representadas nos arquivos de entrada através de notações específicas. Cada uma é representada à partir de um caractere, sendo as posições que são acessíveis ao cavalo (para onde pode pisar) representadas por um '.', as inacessíveis por um 'x', a 'Saída' com um 'S' e o ponto inicial onde se encontra o cavalo por um 'C'. A Figura 1 mostra um exemplo de um destes arquivos.

```
.................x.....................x
.x......x..x..x..x..............x....x.
..........xx....x....x...........x...x
......x.x.........x..............x..x.x
..........C.....xx....x...xx...x........
.x....xx......xxx..xx....x.xx...........
.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.xx.....x.....
.....x..x...x..........x................
.x............x.........xx..x.xx...x.x..
x..............x....x.x........x..S.....
..............x...x.x.........x....x....
.....x....x.......xxx......x........x.x.
.............x.xx...xxx......x..........
........x..x........x.....x....x........
.....xx...x........x....x...x.....x.....
```

Figura 1: Exemplo de tabuleiro recebido

É preciso descobrir qual será a quantidade mínima de movimentos que o cavalo, que se encontra inicialmente na posição 'C', precisa fazer para chegar ao ponto 'S' de cada tabuleiro, caso isso seja possível. Os movimentos

que podem ser feitos pelo cavalo são padronizados, iguais os de um cavalo de xadrez (como podem ser observados na Figura 2). Para construir seu caminho, o cavalo pode caminhar para os pontos '.', mas não pelos representados com 'x'. Para o tabuleiro representado na Figura 1, o cavalo, que inicia no ponto 'C' pode chegar até a 'Saída', 'S', em 10 movimentos.

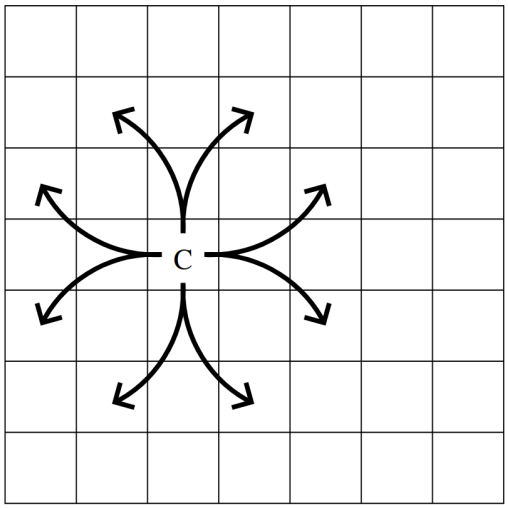

Figura 2: Movimentos os quais o cavalo pode realizar (um "movimento de cavalo").

2 Solução

Para a modelagem da solução foi pensado o uso de uma estrutura de grafo. Onde cada posição acessível (diferente de 'x' no arquivo de entrada) do tabuleiro seria representado por um vértice, que carrega consigo sua "cor"e "distância"(que serão importantes para o algoritmo que fará a busca pelo menor caminho até 'S') e sua posição no tabuleiro, sendo que o tabuleiro começa na tupla $(0, 0)$ e $(r - 1,\ c - 1)$ é a última posição(onde r é a última linha e c a última coluna do arquivo que descreve o tabuleiro). E, após isso, seriam criadas as arestas, representadas por listas de adjacência, para cada um dos vértices do grafo representando quais posições podem ser alcançados à partir de cada um fazendo um "movimento de cavalo"(Figura 2). E, por fim, usando o algoritmo de caminhamento em largura (BFS), que será explicado posteriormente, para descobrir o menor caminho entre o vértice que representa o ponto inicial do cavalo, 'C', e o que representa a 'Saída', 'S'.

Para a elaboração do algoritmo foi utilizada a linguagem de programação "Python"na sua versão 3.8.10.

Inicialmente foi construída uma classe para a representação dos vértices, tendo "cor", "distancia"e "coordenadas"como suas propriedades. Para todo novo vértice que é instanciado, sua "cor"é definida como sendo "branco", sua "distância"sendo $infinita$ e recebendo as coordenadas que ocupa no tabuleiro por parâmetro.

```
1 classe Vertice(coordenadas)
2    cor = "branco"
3    distancia = infinita
4    coordenadas = coordenadas
```

Para manter todos os vértices acessíveis com um bom desempenho, todos foram armazenados em um dicionário chamado "vertices", que utiliza como chave as coordenadas de cada vértice para acessá-los. Para computarmos as coordenadas de cada vértice duas variáveis foram criadas, "r"e "c"que representam a linha e coluna do tabuleiro respectivamente, a variável "r"é incrementada a cada nova linha lida do arquivo e a variável "c"incrementada após cada caractere ser lido de uma linha, e no momento em que o vértice é criado teremos em "r"a linha a qual ele ocupa no tabuleiro e em "c"a coluna.

Para se ter as dimensões totais do tabuleiro, o que será importante na criação das listas de adjacência, as variáveis "comprimentoC"e "alturaR"foram criadas, elas recebem os últimos valores armazenados nas variáveis "r"e "c"(já que são a última linha e a última coluna do tabuleiro).

Para todas as posições do tabuleiro que o caractere for diferente de 'x', que é um ponto inacessível, é adicionado um novo vértice ao dicionário "vertices"com chave sendo sua posição (a tupla de "r"e "c"atuais). E as posições que tenham os caractere 'S' ou 'C' também foram armazenadas em variáveis específicas ("posFim"e "posCav", respectivamente), já que são nossos pontos de interesse do tabuleiro, a 'Saída' e posição inicial do cavalo.

Ao final do método retornamos o dicionário "vertices", "posFim", "posCav"e as dimensões do tabuleiro "comprimentoC"e "alturaR"

O pseudo-código para a criação dos vértices foi o seguinte:

```
criaVertices(arquivoEntrada)
   r = 0
   vertices = dict()
   comprimentoC = 0
   alturaR = 0
      para cada linha do arquivoEntrada
         c = 0
         para cada caractere da linha
            novoVertice = Vertice((r,c))
            se caractere != 'x'
               vertices[(r,c)] = novoVertice
               se caractere == 'S'
                  posFim = novoVertice
               senao se caractere == 'C'
                  posCav = novoVertice
            c = c+1
         r = r+1
      comprimentoC = c
      alturaR = r
   retorna vertices, posFim, posCav, comprimentoC, alturaR
```

E à partir de cada um destes vértices, retornados do método anterior, foi criada uma lista de adjacência à eles (os vértices adjacentes à um outro vértice são aquelas que podem ser alcançadas através de um "movimento de cavalo", que é descrito na Figura 2).

Para construir a lista de adjacência chamada "listaAdj", que é um dicionário, é preciso iterar por todas as chaves no dicionário "vertices"e somar o "movimento de cavalo"(Figura 2) para cada "c"e cada "r"em todas as direções possíveis (isto é feito através da iteração por "movimentosR"e "movimentosC", que, juntos, têm as oito variações possíveis de "movimento de cavalo"). Com isto são gerados um "novoC"e um "novoR"que representam as coordenadas após o "movimento", caso os pontos "novoC"e "novoR"excedam os limites do tabuleiro (das colunas ou linhas), ele se conecta com o outro lado. Após isso é conferido no dicionário "vertices"se existe uma chave ("novoR", "novoC"), o que irá dizer se uma aresta pode ser criada, se não existir, é uma aresta inválida (já que não existe nenhum vértice neste lugar), caso exista, ela é adicionada à lista de adjacência do vértice atual da iteração e o vértice atual da iteração também é adicionado à lista de adjacência do vértice de chave ("novoR", "novoC") de "vertices".

O pseudo-código usado para a construção da lista de adjacência de cada vértice pode ser observado a seguir:

```
criaArestas(vertices, comprimentoC, alturaR)
   movimentosR = [-2, -1, 1, 2, -2, -1, 1, 2]
   movimentosC = [-1, -2, -2, -1, 1, 2, 2, 1]
   listaAdj = dict()
   para cada vertice em vertices
      tmpAdj = set()
      para i de 0 a 7
         novoR = vertice[0] + movimentosR[i]
         se novoR < 0
            novoR = novoR + alturaR
         se novoR > alturaR-1
            novoR = novoR - alturaR
         novoC = vertice[1] + movimentosC[i]
         se novoC < 0
            novoC = novoC + comprimentoC
         se novoC > comprimentoC-1
            novoC = novoC - comprimentoC
         se (novoR, novoC) existe em vertices
            tmpAdj.add(vertices[novoR, novoC])
         se (novoR, novoC) existe em listaAdj
            listaAdj[vertices[novoR, novoC]].adiciona(vertices[vertice])
         senao
            listaAdj[vertices[novoR, novoC]] = set(vertices[vertice])
      listaAdj[vertices[vertice]] = tmpAdj
   retorna listaAdj
```

Antes de prosseguir é preciso explicar as "cores"dos vértices para se compreender o algoritmo à seguir. Vértices cuja "cor"é "preta"são vértices que têm como adjacentes apenas outros vértices "preto"ou "cinza"(ou seja, todos os seus adjacentes são conhecidos), já os "cinza"podem ter como adjacente alguns vértices de "cor"igual a "branco"e são vértices que já foram "descobertos"e vértices que tem sua "cor"sendo "branco"são vértices ainda "desconhecidos.

E para encontrar o caminho mais curto do ponto inicial onde se encontra o cavalo 'C', que foi armazenado na variável "posCav", até o ponto de 'Saída', 'S', que está armazenado na variável "posFim", se fez uso do algoritmo de busca em largura (ou *Breadth First Search*, em inglês). O algoritmo recebe por parâmetro um vértice de origem, neste caso "posCav", e a lista de adjacência dos vértices, o dicionário "listaAdj". No começo do método a "cor"de "posCav"é definida como "cinza", já que ele foi "descoberto"e se atribui 0 à sua distância, já que ele é o vértice de origem. E então se enfileira a "posCav"na fila "Q", e é iniciada uma iteração que irá perdurar enquanto a fila "Q"não seja vazia. O primeiro elemento da fila "Q"é desenfileirado e atribuído à uma variável "u". E então se conferem todos os vértices "v"adjacentes à "u"(se diz que os vértices "v"são alcançáveis à partir de "u") e para cada um se atribui a "cor"igual a "cinza", já que passaram à ser conhecidos, eles passam a ter sua "distância"definida como a "distância"de "u"+1 e são enfileirados na fila "Q". Caso algum destes vértices "v"possuir as "coordenadas"iguais à da variável "posFim"(ou seja, "v"ser o vértice "posFim"), se pode parar a iteração e retornar a "distância"de "v", já que o ponto 'S' ("posFim") é acessível naquele momento, caso contrário "u"tem sua "cor"atribuída como "preto", já que todos vértices "v"adjacentes à ele já foram "descobertos"e se repete o processo até que a posição procurada seja alcançável ou que a fila "Q"esteja vazia e não se encontre tal posição.

O pseudo-código usado para a construção do algoritmo de busca em largura foi o seguinte:

```
1  BFS(posCav, listaAdj)
2     Q = list()
3     posCav.cor = "cinza"
4     posCav.distancia = 0
5     Q.enqueue(posCav)
6     enquanto Q != {}
7        u = Q.dequeue()
8        para cada v na listaAdj[u]
9           se v.cor == "branco"
10             v.distancia = u.distancia+1
11             v.cor = "cinza"
12             Q.enqueue(v)
13          se v.coordenadas == posFim.coordenadas:
14             retornar v.distancia
15       u.cor = "preto"
```

3 Análise de resultados

Resultados encontrados à partir dos arquivos fornecidos pelo professor, mostrando a quantidade de movimentos necessárias em cada caso para se chegar à 'Saída' do tabuleiro:

Caso	Movimentos necessários
caso100.txt	68
caso150.txt	64
caso200.txt	108
caso250.txt	156
caso300.txt	197
caso350.txt	170
caso400.txt	185
caso450.txt	186
caso500.txt	225
caso550.txt	233

Tabela 1: Resultados encontrados a partir dos arquivos fornecidos pelo professor

4 Conclusões

O entendimento do problema usando estrutura de grafo foi essencial para sua resolução, associando cada vértice do grafo à cada ponto acessível de um tabuleiro e estabelecendo as arestas como sendo os pontos acessíveis através de um "movimento de cavalo"à partir de cada vértice, a solução se mostrou não tão assustadora quanto parecia inicialmente.

Se acredita ter chego em uma solução confiável para o problema, já que foram testados para algumas outras entradas que não foram mostradas neste artigo e o programa respondeu bem chegando à resultados corretos para todos.

Apesar de não ser fundamental para a resolução deste problema, acredita-se que se chegou em um algoritmo de desempenho satisfatório, para o "caso550.txt"a construção do grafo demorou 0,74 segundos e a busca em profundidade levou, em média, 0.09 segundos para ser concluída.

Pensar a respeito do algoritmo desenvolvido neste trabalho fez com que o olhar do autor se expandisse sobre as possibilidades da computação e de resolução de problemas.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.