Trabalho II - O Labirinto do Horror I

Bianca Camargo Machado*
Escola Politécnica - PUCRS

7 de novembro de 2018

Resumo

Este artigo descreve uma solução para o segundo problema proposto na disciplina Algoritmos e Estruturas de Dados II no semestre 2018/2, trata-se da determinação da entrada, saída e distância percorrida entre elas no labirinto de Teseu, Ariadne e Minotauro. Este, que teve seu segredo descoberto em uma escavação em Creta. Foram disponibilizados 10 casos de teste para resolução do problema: cada um contém uma matriz de hexadecimais, utilizada para representar o labirinto, e dois números que especificam o número de linhas e colunas existentes nesta matriz. Após descrever a solução do problema, são apresentados os resultados obtidos para os casos de teste, juntamente com seus tempos de execução.

Introdução

No contexto da disciplina de Algoritmos e Estruturas de Dados II, o enunciado do segundo trabalho proposto pode ser resumido da seguinte forma: após descoberto o segredo do labirinto de Teseu, Ariadne e Minotauro, sendo estes descritos por m x n células, sendo $m \geq 20$ e $n \leq 500$. E ainda, cada célula do labirinto contém informações sobre suas conexões com as células conectadas a ela diretamente (acima, abaixo, à esquerda e à direita) e estas informações representam 4 bits na numeração hexadecimal, em que cada bit corresponde à definição de existência ou não de parede superior, direita, inferior e à esquerda da célula.

Um exemplo de valor de célula arbitrária do labirinto é E, que em representação binária é 1110. A partir disto observe a imagem a seguir, que demonstra o que podemos abstrair de informações através deste código.

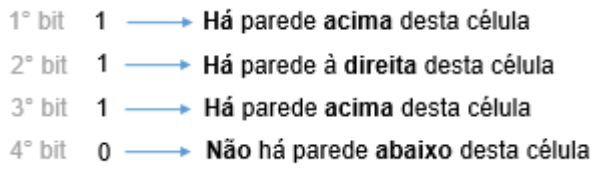

Figura 1: Demonstração do significado de cada bit para uma célula do labirinto.

A partir destas informações sobre o segredo deste labirinto, tem-se como objetivo localizar as células de entrada e saída do mesmo, junto com o comprimento do caminho percorrido de um ponto ao outro. Ao observar as informações prestadas no enunciado e os casos de teste disponibilizados, podemos chegar a conclusões inerentes à proposta que podem auxiliar na resolução do problema, são elas:

- Existem somente dois pontos que possibilitam ir para fora do labirinto (uma entrada e uma saída);
- Não há definição clara sobre qual dos dois pontos de saída do labirinto é de fato a entrada e vice-versa;
- As possíveis saídas e entradas do labirinto localizam-se nas bordas do mesmo.

*bianca.camargo@acad.pucrs.br

Para ilustrarmos o problema e trabalharmos na sua resolução, vamos utilizar o exemplo fornecido no enunciado do trabalho, este que contém um modelo de caso de teste, ilustração do labirinto deste caso e os valores dele convertidos em binário, como é possível observar na imagem abaixo.

Figura 2: Exemplo de caso de teste disponibilizado, representação binária do caso de teste e ilustração do labirinto. Este exemplo será abordado ao decorrer do artigo.

Ao observar este exemplo podemos abstrair mais algumas informações úteis para o entendimento da proposta e análise de como a solução deve ser implementada. São elas:

- A primeira linha do arquivo de caso de teste, sempre apresentará as dimensões do labirinto;
- As informações sobre as células do caso de teste estão dispostas em forma de matriz;
- É necessário converter o valor de cada célula para que tenhamos as informações sobre as paredes da mesma;
- Uma célula está ligada a outra quando existe uma direção (superior, direita, inferior ou esquerda) com o valor 0 e em seguida uma célula existente.
- Como identificar entrada/saída do labirinto? Sabemos que elas possuem ao menos uma das direções com valor = 0, mas para se caracterizar como tal, esta direção deve estar nas bordas do labirinto, ou seja, estas são as possibilidades para saída/entrada do labirinto:
  - Lado **superior** da célula: linha = 0;
  - Lado **esquerdo** da célula: coluna = 0;
  - Lado **inferior** da célula: linha = tamanho da matriz - 1;
  - Lado **direito** da célula: coluna = tamanho da matriz - 1;

Os valores apresentados acima, de linha e coluna, são baseados no fato de que a matriz é composta por **n** linhas e **m** colunas, sendo que $\mathbf{n} \geq 0$ e $\mathbf{m} \leq$ tamanho da matriz.

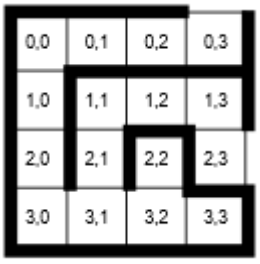

Figura 3: Valores de n,m para cada célula da matriz.

Análise do problema

Ao analisarmos o labirinto (Figura 3) em uma estrutura de grafos podemos observar que cada célula do labirinto corresponde a um vértice do grafo a ligação entre estes vértices depende dos valores armazenas no mesmo. Portanto, assim teríamos um grafo não direcionado - pois o caminho que se faz de um vértice ao outro pode servir tanto para ida quanto para volta.

Como não existe nenhum critério definido para especificar qual é a entrada e qual é a saída, portanto, consideraremos que a entrada será a primeira célula para fora do grafo que o algoritmo localizar e a saída será a segunda que o algoritmo identificar como tal.

Devido à necessidade de armazenar no vértice informações sobre as paredes do mesmo, cabe considerar que cada vértice do grafo deve direcionar para tais informações e portanto, será considerada a seguinte estrutura:

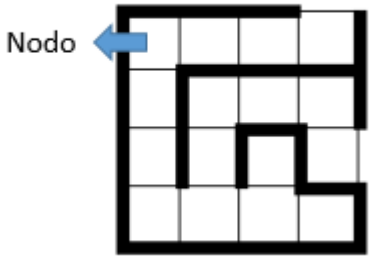

Figura 4: Cada vértice do grafo referencia um Nodo.

Com base nestas informações, considerando que já temos informações suficientes para determinar qual é o vértice de entrada e qual é o de saída - a partir de sua posição na matriz -, o próximo passo é localizar o caminho de um ponto ao outro. Mas como fazer isto? Um algoritmo que resolve este problema é o algoritmo de busca por profundidade no grafo, que recebe uma entrada específica e uma saída para percorrer o grafo da seguinte forma: parte de um nodo e busca seus adjacentes. Vale lembrar que o caminho que precisamos encontrar nada mais é que o conjunto de nodos adjacentes necessário para ir de um ponto ao outro.

Segundo [1], a estratégia seguida pela busca em profundidade é, como seu nome implica, procurar "mais fundo" no grafo sempre que possível. Na busca por profundidade, as arestas são exploradas a partir do vértice **v** mais recentemente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas de **v** são exploradas, a busca "regressa" para explorar as arestas que deixam o vértice a partir do qual **v** foi descoberto. Esse processo continua até descobrirmos todos os vértices acessíveis a partir do vértice de origem inicial.

A busca em profundidade, neste caso, vai ser feita até que o vértice de saída seja encontrado e durante esta busca, é possível contar o número de vértices percorridos, o resultado que o problema nos pede.

Solução

O algoritmo para o problema em questão foi implementado na linguagem Java e segue a lógica descrita anteriormente. Os casos de testes estavam disponíveis em um arquivo de texto, portanto o algoritmo que localiza os pontos de entrada e saída do labirinto e a distância percorrida foi implementado após a leitura do arquivo e armazenamento dos dados em uma matriz, na qual cada célula armazena um nodo (Figura 3).

O nodo é representado por uma estrutura com os seguintes atributos: superior, direita, inferior e esquerda - sendo que cada um deles armazena um bit -, marca - para controlar se o nodo já foi visitado na busca por profundidade - e uma lista de adjacentes do nodo.

A imagem a seguir também é relacionada ao labirinto apresentado na introdução, está disposta afim de ilustrar as informações que serão utilizadas no algoritmo que será descrito a seguir.

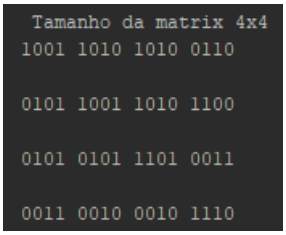

Figura 5: Imagem dos bits armazenado em cada nodo da matriz.

O algoritmo a seguir localiza a entrada e saída do labirinto (Figura 2, 3 e 4) e armazena as coordenadas x e y destas em uma lista chamada **pontos**.

```
para i de 0 ate tamanho da matriz {
    //testa paredes superiores dos nodos da linha 0 da matriz
    se (matriz[0][i].superior == 0)
        adiciona 0 em ponto;//coordenada da linha armazenada
        adiciona i em ponto;//coordenada da coluna armazenada
    //testa paredes esquerdas dos nodos da coluna 0
    se (matriz[i][0].esquerda == 0)
        adiciona i em ponto;
        adiciona 0 em ponto;
    //testa paredes inferiores dos nodos da ultima linha da matriz (tamanho da matriz -1)
    se (matriz[tamanhoDaMatriz-1][i].inferior == 0)
        adiciona (tamanhoDaMatriz-1) em ponto;
        adiciona i em ponto;
    //testa paredes direitas dos nodos da ultima coluna da matriz (tamanho da matriz -1)
    se (matriz[i][tamanhoDaMatriz-1].direita == 0)
        adiciona i em ponto;
        adiciona (tamanhoDaMatriz-1) em ponto;
}
```

Ao final da execução do algoritmo acima, **pontos** terá 4 coordenadas, sendo as duas primeiras x e y da célula em que se encontra a entrada do labirinto e as duas últimas referentes à saída do labirinto. E com isso podemos chamar o método **buscaProf** que recebe por parâmetro o nodo de entrada do labirinto e um contador, inicialmente com o valor 0. Este contador armazenará o número de nodos percorridos para chegar na saída do labirinto, a partir da entrada.

```
buscaProf(Nodo atual, contador) {
    //critério de parada: se chegou no objetivo
    se (atual == saida)
        retorna contador
    //marca o nodo como visitado
    nodo.marca = 1
    para cada nodo n adjacente de nodo atual
        se n não foi visitado
            contador++
            //busca o próximo adjacente ao nodo atual
            contador = buscaProf(n,contador)
retorna contador
}
```

Observe que, para que o algoritmo acima funcione corretamente, é necessário que o atributo contendo a lista de adjacentes do nodo esteja completa. O algoritmo a seguir (setaAdjacentes) insere todos os nodos adjacentes (diretamente ligados ao nodo em questão) na lista correspondente, possibilitando que o algoritmo **buscaProf** retorne o número de tamanho do caminho percorrido da entrada até a saída do labirinto.

```
setaAdjacentes() {
//critério de parada: se chegou no objetivo
    para i de 0 até (tamanhDaMatriz-1)
        para j de 0 até (tamanhDaMatriz-1)
            se (matriz[i][j].superior == 0) e i maior que 0

              adiciona m[i-1][j] na lista de adjacentes de m[i][j]
            se (matriz[i][j].direita == 0) e j menor que (tamanho da matriz - 1)

              adiciona m[i][j+1] na lista de adjacentes de m[i][j]
            se (matriz[i][j].inferior == 0) e i menor que (tamanho da matriz - 1)

              adiciona m[i+1][j] na lista de adjacentes de m[i][j]
            se (matriz[i][j].esquerda == 0) e j maior que 0

              adiciona m[i][j-1] na lista de adjacentes de m[i][j]
}
```

Utilizando as coordenadas dos nodos na matriz, como mostrado na Figura 3, foi possível obter os adjacentes de cada nodo de acordo com a direção em que não havia parede (valor = 0). Com este algoritmo foi possível localizar os nodos adjacentes e então realizar a busca por profundidade, para no fim obtermos os resultados esperados.

Para o algoritmo abordado como exemplo ao longo deste artigo, obteve-se os seguintes resultados:

- Entrada: c2,3 (célula na linha 2 e coluna 3)
- Saída: c0,3 (célula na linha 0 e coluna 3)
- Distância: 15 células percorridas.

Na imagem a seguir, encontra-se a ilustração destes resultados.

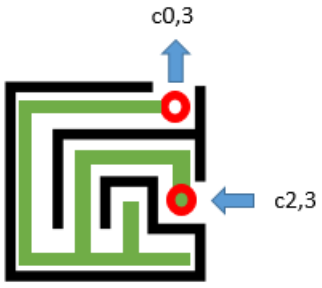

Figura 6: Caminho (em verde) percorrido no labirinto de exemplo. Célula de entrada: c2,3 e célula de saída: c0,3

Resultados

Os resultados obtidos com os casos de testes, disponibilizados junto ao problema, exigiram um tempo de execução do algoritmo que variou entre 78 e 6.542 milissegundos e esta diferença de tempo deve-se à complexidade das estruturadas das matrizes contidas em cada caso.

Na tabela a seguir são apresentados os resultados obtidos para cada caso de teste, contendo o tempo de execução desde a leitura do arquivo até o retorno do resultado, além do retorno sobre a entrada e saída do labirinto e a distância (quantidade de nodos percorridos no caminho da entrada até o fim do labirinto).

Caso de teste	Entrada: c(linha,coluna)	Saída: c(linha,coluna)	Distância (células)	Tempo (milissegundos)
caso25c	c0,3	c24,12	624	78
caso50c	c17,49	c49,43	2.496	268
caso75c	c7,0	c0,60	5.624	303
caso100c	c11,0	c0,85	621	8.418
caso150c	c51,0	c112,149	22.492	809
caso200c	c0,91	c157,0	39.999	1.395
caso250c	c53,0	c100,0	62.499	1.911
caso300c	c274,299	c279,0	89.760	2.590
caso400c	c9,399	c38,0	159.997	4.134
caso500c	c0,212	c0,381	221.421	6.105

Tabela 1: Resultados dos casos de teste aplicados ao algoritmo desenvolvido.

Conclusões

A solução desenvolvida e apresentada ao longo deste artigo obteve resultados satisfatórios em relação às saídas esperadas e o tempo necessário para que o algoritmo resolvesse os problemas disponíveis nos casos de teste. Tendo em vista que algumas decisões a nível lógico e estrutural do código podem afetar de forma significativa os resultados e eficiência do algoritmo, foi de extrema importância o entendimento do problema como um todo e detalhamento das informações inerentes à estrutura do labirinto. O que possibilitou chegar-se a resultados satisfatórios e o objetivo principal concluído.

Referências

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Cliford: "Algoritmos, Teoria e Prática". 2° ed. Elsevier, 2002, 429 p.