Departamento de Informática

Estruturas de Dados e Algoritmos II

Ano Letivo de 2022/2023

Trabalho realizado por:

- Luís Gonçalo Carvalho Nº51817
- Pedro Emílio Nº52649
- Mooshak: g204

Docente: Vasco Pedro

índice

Introdução

Este relatório é referente ao segundo trabalho da disciplina Estrutura de Dados e Algoritmos II – *"Was it a Dream?"*.

O problema consiste em mover uma esfera, que dado um mapa inicial pelo utilizador, ela move-se, desde a sua posição inicial (também dada pelo utilizador) até chegar a um buraco, no menor número possível de movimentos.

A esfera só se pode mover na horizontal ou na vertical, dentro do mapa e nas posições livres. A esfera não pode continuar na direção de um obstáculo e para quando chegar ao buraco. A solução proposta para encontrar o caminho mais curto da posição até o buraco utiliza o algoritmo de busca em largura (BFS)

Análise do problema - Input

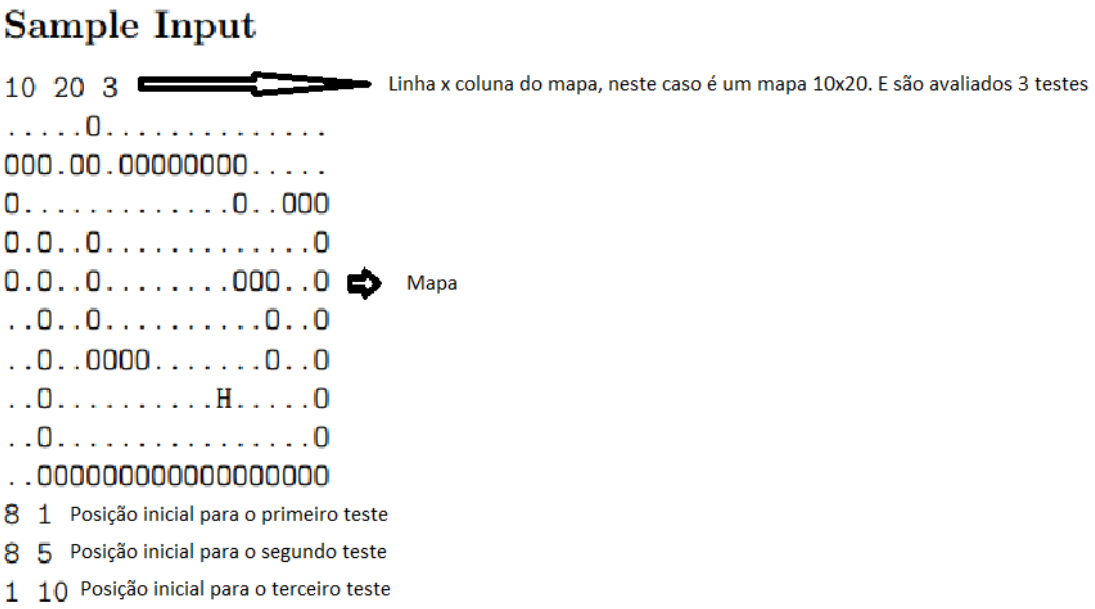

Figura 1: A nossa análise ao "Sample Input" dado pelo professor.

Estruturas Utilizadas

Foi implementada uma classe designada por ***Point***, responsável pela criação dos pontos. A classe possuí três variáveis de classe:

- Private int linha: Corresponde à linha da posição atual do ponto, onde a esfera está;
- Private int coluna: Corresponde à coluna da posição atual do ponto, onde a esfera está;
- Private int movimentos: Corresponde à quantidade de movimentos que foram realizados até atingir a posição atual do ponto, onde a esfera está.

A classe ***Point*** tem três métodos que facilitam no resto do programa:

- Public int getRow(): Retorna a linha onde a esfera está;
- Public int getCol(): Retorna a coluna onde a esfera está;
- Public int getMoves(): Retorna os movimentos que a esfera fez até chegar à posição em que está.

No método main:

- Int numeroDeLinhas: número de linhas do mapa dado pelo utilizador;
- Int numeroDeColunas: número de colunas do mapa dado pelo utilizador;
- Int numeroDeLinhas: número de casos a testar do mapa dado pelo utilizador;
- Char mapa[][]: Matriz com tamanho das linhas e colunas do mapa dado pelo utilizador, onde vai guardar o mapa (as posições livres - . ; os obstáculos – “O”; o buraco – “H”);
- String linhaMapa: Esta variável armazena cada linha do mapa a cada iteração do for. Foi criada de modo a facilitar o armazenamento do mapa dado pelo utilizador na variável acima apresentada, pois guarda carácter a carácter.
- Int caso: Variável auxiliar, para testar caso a caso.

- String[] linhaTeste: Variável que guarda a posição inicial de cada teste.
- Int coordLinha: Linha da posição inicial de cada teste.
- Int coordColuna: Coluna da posição inicial de cada teste.
- Int resultado: É a menor quantidade de movimentos possíveis, que a esfera realiza desde a posição inicial até o buraco. Se não chegar ao buraco, o resultado é -1 (Significa que dá "Stuck" como output). (Tem valor default -1).

No método bfs, recebe como argumentos o mapa do utilizador, e a linha e coluna das posições iniciais de cada teste.

- Criámos uma queue para armazenar os pontos (posições) onde a esfera para. No início, a fila contém apenas a posição inicial do teste. Em seguida, executamos o ciclo while que processa cada posição da fila, adicionando as posições adjacentes que ainda não foram visitadas à fila. Esse processo é repetido até que a fila esteja vazia (todas as posições alcançáveis tenham sido visitadas).
- Int nLinhas / nColunas: Variáveis auxiliares, dimensão do mapa do utilizador.
- Boolean[][] posicoesAvaliadas: Matriz com a mesma dimensão do mapa do utilizador que guardar valores true ou false sobre cada posição. Se a posição já foi avaliada guarda true, caso contrário guarda false.
- Point pontoEmQueEsta: É o ponto onde a esfera está, ou seja o ponto que é retirado da queue para ser avaliado nas quatro direções possíveis.
- Int linhaPos e colunaPos: Variáveis auxiliares, que armazenam a linha e a coluna da posição onde a esfera está.
- Int colunaDir: Variavel auxiliar para avaliar as posições à direita da posição atual da esfera (Avalia-se incrementando 1 na coluna).
- Int colunaEsq: Variavel auxiliar para avaliar as posições à esquerda da posição atual da esfera (Avalia-se ao subtrair 1 na coluna).

- Int linhaCima: Variavel auxiliar para avaliar as posições em cima da posição atual da esfera (Avalia-se ao subtrair 1 na linha).
- Int linhaBaixo: Variavel auxiliar para avaliar as posições em baixo da posição atual da esfera (Avalia-se ao incrementar 1 na linha).

Algoritmo

1. Criar a classe “Point” que representa o ponto no mapa, onde a esfera está parada, com os atributos das suas coordenadas e do menor número de movimentos até atingir essa posição.
2. No método main, criar um objeto BufferedReader (BufferedReader input) para ler o input (Resolução do mapa, o mapa, casos a testar, as coordenadas das posições iniciais de cada teste).
3. Criar o método bfs para encontrar o caminho mais curto da posição até o buraco. É criada a queue para guardar os pontos onde a esfera para, de modo a facilitar a avaliação de cada posição.
4. A posição inicial de cada teste é guardada inicialmente na queue, para ser o “ponto de partida”. Primeiro é verificado se a posição inicial é um objeto, se for então vai retornar -1, que vai ter como output “Stuck”.
5. Caso a condição anterior não se verifique, então o programa entra no ciclo while, que vai continuar a executar até que a queue esteja vazia. Dentro do while é retirado o primeiro elemento da queue, que corresponde à posição atual da esfera.
6. Foi criado quatro ciclos, que permite avaliar cada direção possível da esfera (para a direita, para baixo, para a esquerda, para cima).
7. Seja por exemplo, na avaliação para a direita, criámos a variável colunaDir, onde começa com a variável vizinha direita à da posição atual da esfera. O ciclo vai até ao limite direito, ou seja, avalia até à posição da última coluna. Durante a avaliação, se a posição da variável auxiliar (colunaDir) for um objeto e a posição anterior não foi avaliada, significa que a bola vai parar nessa posição porque não pode continuar, visto ter um objeto à sua frente. Então essa posição anterior é guardada e adicionada à queue, a posição passa a true (foi avaliada) e os moves da posição inicial até esta posição anterior, incrementa 1 nos movimentos. Se a bola chegou ao buraco, o ponto

é adicionado à queue, o movimento incrementado e dá break ao ciclo. Depois quando essa posição for avaliada (posição do ponto) vai entrar no primeiro if, onde o resultado vai ser os movimentos guardados no método getMoves() da classe "***Point***" e dá break ao ciclo.

Neste caso referimos como exemplo na direção direita, mas o algoritmo aplica-se nas outras direções mudando apenas a variável auxiliar, no caso das direções verticais, usa-se as linhas, no caso das horizontais usa-se as colunas.

Caso a bola nunca consiga chegar ao buraco, a queue ficará vazia, pois as posições vão se tornar avaliadas e por mais moves que faça, a bola pode nunca chegar ao buraco, então o resultado irá se manter -1.

8. Após o método bfs retornar o resultado, no método main, se o resultado for -1, então dá "Stuck", caso contrário, mostra o menor número de movimentos.

Análise do problema – Output

Para além dos testes que o professor deu, criámos um novo mapa 7x7, como mostra a figura seguinte e fizemos 3 testes (posições a,b,c):

Figura 2: Exemplo mapa 7x7

Para o teste a):

- Começamos na coordenada (2,2). A esfera não pode ir para cima nem para a esquerda, por causa dos objetos, logo para a direita vai parar na (2,6) e não pode continuar porque tem objetos. Para baixo vai para (5,2) e não pode continuar, logo fica "Stuck".

Para o teste b):

- Começamos na coordenada (6,4). A esfera basta subir que entra logo no buraco, que está na coordenada (4,4), logo o move é 1. Poderia ir também para a coordenada (6,6), (4,6) e depois chegar ao buraco, mas assim não seria no mínimo de moves, pois 3>1.

Para o teste c):

- Começamos na coordenada (6,3), a esfera só pode ir para a direita, e para em (6,6), depois ou sobe ou volta para a mesma posição, então sobe para (4,6) e depois vai para a esquerda, para o buraco (4,6), logo são 3 moves.

Complexidade Temporal

A classe "***Point***" tem complexidade O(1).

As linhas 38-42 têm complexidade O(1). 46-51 (ler e armazenar o mapa) tem complexidade O(numeroDeLinhas + numeroDecolunas). 54-70 (ler os casos de teste) tem complexidade O(numerorCasosATestar).

Como o algoritmos bfs é executado para cada caso de teste, a complexidade será O(numerorCasosATestar * (numeroDeLinhas + numeroDecolunas)).

Complexidade geral do código: O((numeroDeLinhas * numeroDeColunas) + numerorCasosATestar * (numeroDeLinhas + numeroDeColunas))

Complexidade Espacial

O mapa em si tem um tamanho de nLinhas * nColunas.
A matriz posicoesAvaliadas também tem um tamanho de nLinhas * nColunas.
A fila 'queue' armazena instâncias de Point, que podem chegar a um máximo de nLinhas * nColunas no pior caso, quando todas as células são visitadas.
A complexidade espacial total, portanto, é a soma do espaço ocupado por todas estas estruturas de dados:

O(nLinhas * nColunas) + O(nLinhas * nColunas) + O(nLinhas * nColunas)