Engenharia Informática

Professor Vasco Pedro

Estrutura de dados e algoritmos II

Relatório trabalho prático

“Was it a dream?”

Grupo nº219

Diogo Ferreira, n.º43075
Rúben Farinha, n.º48329

1. Introdução

O problema "Was it a Dream?" trata de encontrar o caminho mínimo para chegar em casa depois de se encontrar dentro de uma esfera em movimento. Dado um mapa retangular com obstáculos e um buraco que leva para casa, o objetivo é determinar quantos movimentos são necessários para chegar ao buraco a partir de uma posição inicial dada. A esfera só pode se mover em locais vazios, em direções paralelas aos lados do mapa, e uma vez em movimento, só pode ser interrompida por obstáculos ou pelo buraco. Se a esfera cair de um dos lados do mapa, não há como voltar ao mapa novamente. O problema apresenta múltiplos casos de teste, onde cada caso começa com o mapa e a posição inicial da esfera.

Foi-nos assim proposto que realizássemos um trabalho que inclui um programa que recebe como input 3 inteiros (sendo eles número de linhas, número de colunas do mapa e casos de teste), um mapa e as posições iniciais da esfera.

O output do mesmo deveria ser o número mínimo de movimentos que a esfera pode fazer desde a sua posição inicial até “home” (posição simbolizada por H no mapa).

Caso seja impossível chegar a H a partir da posição inicial o resultado deve ser a string “stuck”.

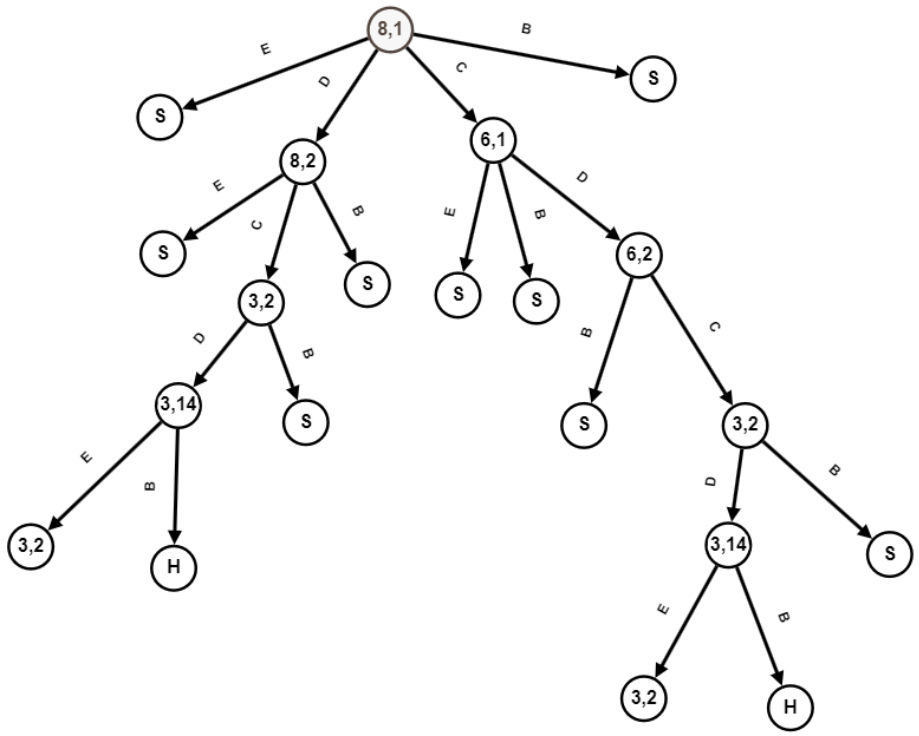
2. Descrição do algoritmo

Considerámos a utilização do algoritmo BFS (Breadth First Search), o qual consiste em explorar todos os nós adjacentes a um nó atual antes de prosseguir para os próximos nós. Neste problema em particular, podemos empregar o algoritmo BFS para examinar todas as posições acessíveis a partir da posição inicial da esfera, permitindo-nos assim descobrir o caminho mais curto para o objetivo final - o buraco (H).

Para implementar o BFS, utilizámos uma fila para armazenar os nós (posições) a serem explorados. Começando por adicionar a posição inicial dada à fila. Repetimos então o seguinte processo até encontrar o buraco ou que a fila fique vazia:

1. Retiramos o nó/posição da fila.
2. Verificamos se este é o buraco ou se o buraco se encontra em uma das 4 direções possíveis (cima, baixo, direita, esquerda).
3. Se encontrarmos o buraco, então encontramos o caminho mais curto e finalizamos o algoritmo.
4. Caso contrário, verificamos todas as direções possíveis.
5. Se a direção leva a um obstáculo, adicionamos a posição anterior ao obstáculo (onde há um "bump") na fila e atualizamos o valor na variável "moves" para indicar que essa posição foi visitada e evitar possíveis loops.
6. Se a direção leva para fora do mapa, descartamos essa posição.

Podemos representar como exemplo do primeiro caso de teste de posição (8, 1) e a sua execução no algoritmo com este grafo:



No final do algoritmo, se não encontrarmos o buraco, significa que a esfera está presa e não é possível chegar ao destino, nesse caso, a resposta é "Stuck". Caso contrário, a resposta será o número de movimentos necessários para chegar ao buraco.

O programa começa por ler três valores do input: o número de linhas e colunas do labirinto e o número de testes que serão realizados. Em seguida, lê-se a representação do labirinto, em forma de matriz de caracteres, e as coordenadas dos buracos para os testes.

Para cada teste, é chamada a função minimumMoves com as coordenadas da posição inicial e com a matriz que representa o mapa. É inicializada outra matriz moves com os valores a “Integer.Max_Value” que irá servir para guardar se a posições já foram visitadas ou não com o número de movimentos.
Em seguida, são definidas as quatro possíveis direções de movimento da esfera: para cima, para baixo, para a esquerda e para a direita. Também é criada uma fila de inteiros para realizar uma busca em largura. A posição inicial é adicionada à fila com o número de movimentos igual a zero.
O algoritmo entra em um loop while que fica em execução enquanto houver posições na fila. A posição atual é removida da fila e suas coordenadas são salvas em currRow e currCol. Em seguida, para cada direção possível, são calculadas as coordenadas da próxima posição. Se a próxima posição estiver dentro dos limites do labirinto e não for uma obstáculo (representado pelo carácter 'O'), o algoritmo entra em um loop while que continua enquanto a esfera não atingir uma parede ou sair dos limites do labirinto.

Dentro desse loop, o algoritmo verifica se a esfera atingiu o buraco (representado pelo carácter 'H'). Se sim, a função retorna o número de movimentos necessários para atingi-lo. Caso contrário, o algoritmo verifica se a posição atual já foi visitada anteriormente e, se não foi, atualiza a matriz moves com o número de movimentos necessários para atingi-la.
Em seguida, o algoritmo calcula as coordenadas da próxima posição na direção atual e verifica se ela está dentro dos limites do labirinto. Se estiver, o algoritmo verifica se ela é um obstáculo. Se for um obstáculo, a posição atual é adicionada à fila. Se não for um obstáculo, o algoritmo continua atualizando as coordenadas da próxima posição na direção atual e verificando se ela é uma parede.
Se o loop terminar sem ter encontrado o buraco, a função retorna -1, indicando que a esfera está presa no labirinto e não pode chegar ao buraco.

Por fim, o programa imprime "Stuck" se a função minimumMoves retornar -1, ou o número de movimentos necessário para chegar ao buraco, caso contrário.

3. Complexidade espacial e temporal

A complexidade espacial do nosso programa é O(R*C) em que R é o número de linhas do mapa e C o número de colunas do mapa pois o array “moves” é inicializado com o valor máximo no início do programa. A quantidade de espaço que a queue e as outras variáveis ocupam é constante e por essa razão estes não contribuem para a complexidade espacial do programa.

Já a complexidade temporal do programa será O(RCT) em que R é o número de linhas do mapa, C o número de colunas do mapa e T o número de casos de teste. A complexidade temporal é esta pois para cada teste a função “minimumMoves” itera sobre todas as células da matriz no seu pior caso. Assim, o tempo de execução da função é proporcional ao número de células na matriz, que é RC e como o número de casos de teste é T então, novamente, a complexidade temporal deste programa é O(RCT).

4. Conclusão e comentários extra

Em tom de conclusão, o trabalho ensinou-nos mais aprofundadamente como funcionava o algoritmo de pesquisa em largura que já tínhamos abordado previamente nas aulas, sendo que aí tínhamos adquirido as noções teóricas que precisávamos para poder completar e complementar com este trabalho prático. A parte em que sentimos mais dificuldade foi tentar tornar o programa mais eficiente e correto, de modo a evitar os erros de “Time Limit” e “Runtime Error” que obtivemos, depois de alguma experiência de tentativa e erro e debug do código conseguimos identificar onde existiam erros nossos e daí podermos corrigi-los de modo que fosse aceite na plataforma Mooshak.