Unidade Curricular de Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

Hill the Climber

Grupo:
g216

Discentes:
Helder Godinho nº42741
Guilherme Grilo nº48921

abril 2022

Docente:
Vasco Pedro

Índice

Introdução

O presente trabalho consiste na criação de um algoritmo que tem como principal objetivo descobrir o melhor caminho desde um ponto inicial, cujo y terá valor 0, até ao ponto final, cujo y terá a altura máxima dada como input pelo utilizador.

Para a realização deste trabalho tivemos que recorrer ao uso de grafos. Um grafo pode ser definido como uma estrutura de dados que consiste essencialmente em um número finito de vértices também chamados de nós, e ainda um número finito de conjuntos na forma (u, v) que são chamados de arcos, que indicam que existe uma ligação entre o vértice u e o vértice v. Quando essa mesma ligação é mútua, isto é, se ambos os vértices se conectam um ao outro, podemos denominar um grafo desse género como grafo não orientado, caso contrário designa-se como grafo orientado. O arco pode ainda conter outra propriedade, o peso, que, quando presente nos arcos de um grafo, o mesmo passa a ser apelidado de grafo pesado. Existem 2 formas de percorrer um grafo, isto é, de visitar todos os nós de um determinado grafo e são eles: *Breadth First Search* (BFS), em português percurso em largura, consiste em começar num certo nó, visitar os seus adjacentes, de seguida visitar os adjacentes destes e por aí adiante até todos os nós terem sido visitados; de salientar que este foi o método escolhido para o nosso trabalho. Temos ainda o *Depth First Traversal* (DFS), em português percurso em profundidade, que consiste em começar num determinado nó, visitar um nó adjacente deste e fazer isto até chegar a um nó que não possua qualquer nó adjacente a si ou, se possuir, que não tenha sido ainda visitado.

Algoritmo

O nosso algoritmo divide-se essencialmente entre três classes, sendo as três bastante importantes para o sucesso deste trabalho.

Climb

A classe Climb pode ser considerada como a classe principal de entre as três, pois é esta que recebe todos os dados importantes para a resolução do problema, tais como o número de pontos disponíveis, a altura máxima que o Hill terá que alcançar ou até mesmo as coordenadas dos vértices, sendo que passará estas mesmas coordenadas para a classe Vertex durante a execução do algoritmo, através da função.

Para além disso, a *Climb* é ainda responsável pela construção do grafo (que será explicada a seguir) e pelo cálculo final do melhor caminho desde o ponto inicial ao ponto final. Após a contrução do grafo, de forma a se calcular a melhor distância do vértice inicial ao vértice final, percorre-se os nós do grafo recorrendo ao BFS com algumas alterações, de forma a adaptar esta forma de pesquisa ao nosso objetivo. Inicialmente, começamos por inicializar a variável *d*, que guarda o número de vértices que foi necessário percorrer desde um certo vértice até ao vértice em questão. As alterações referidas foram feitas dentro do ciclo que serve para percorrer o nó adjacente de determinado nó, isto é, foram introduzidas duas condições que permitem verificar se a distância entre o vértice e o seu adjacente é menor ou igual ao salto de teste e, em caso afirmativo, calcula o número de vértices que foi necessário percorrer para chegar ao adjacente, que é sempre o vértice proveniente + 1, e se este for inferior ao valor guardado na variável *d* do adjacente, então troca-se o valor da mesma variável *d* pelo novo valor calculado. Após terminar de percorrer todos os nós até ao final, devolve-se então o valor guardado na variável *d* do vértice final e se este ainda possuir o valor MAX_VALUE significa que não é possível chegar a esse mesmo ponto com o salto atual, então declara-se como *"unreachable"*, caso contrário imprime-se o valor obtido.

Vertex

A classe Vertex é responsável por alojar as coordenadas de todos os vértices dados, de forma a poder calcular a distância entre os y's de dois vértices com

o salto que o Hill pode realizar. Em caso afirmativo, ou seja, se a distância de salto for inferior ou igual à distância dos y's, então será necessário calcular a distância entre os dois vértices, através da função *calcDistance()*, e se a mesma for inferior ou igual ao salto, então significa que este é um caminho que Hill poderá percorrer, sendo que ambos se podem então conectar um ao outro, de forma não orientada. Esta ligação é possível com recurso a uma lista do tipo da classe *Edge* com o nome de *destination*, cujo principal papel é guardar todos os vértices aos quais um determinado vértice se pode conectar.

Esta classe alberga ainda outra função,*compareTo()*, que é implementada pela interface *Comparable*, que serve para indicar à *priority Queue*, que se encontra na classe *Climb*, que deve ordenar os vértices do y menor para o y maior, o que facilita bastante na construção do grafo.

Edge

Esta classe serve essencialmente para a lista *destination*, que está presente na classe *Vertex*, pois a mesma possui apenas duas instâncias, uma do tipo *Vertex*, *destiny*, de forma a guardar o vértice destino de um determinado vértice, e outra do tipo *int*, *distance*, que guarda a distâcia entre os dois vértices.

Grafo

Este grafo, ao contrário de outros que tínhamos implementado na resoluação de probelmas anteriores, difere um pouco na sua implementação, pois a forma mais comum de implementar um grafo é fazer um *array* de listas, tudo na mesma classe, neste caso o nosso grafo é implementado de maneira um pouco diferente, pois o *array* do tipo *Vertex* está declarado na classe *Climb*, já a lista do tipo *Edge* está declarada na classe *Vertex*. Apesar desta ligeira diferença, na prática, o grafo funciona da mesma forma que os mais comuns.

Inicialmente começamos por passar as coordenadas de cada ponto para a classe *Climb* de forma a podermos inicializar a classe *Vertex* de cada vez que passamos um ponto e colocá-los todos numa *priority Queue*, que os ordena do menor y para o maior. De salientar que criamos ainda mais dois pontos para além dos dados pelo *input*, o ponto inicial de coordenadas (0, 0) e o ponto final de coordenadas (0, altura máxima),sendo a altura máxima dada também como *input*.

De seguida, retiram-se os vértices da *Queue* um por um e colocam-se num *array* e aqui sim começa a verdadeira construção do grafo. Após todos os vértices estarem colocados no *array*, pegamos no primeiro vértice do *array* (*index* 0) e começamos a comparar com os restantes vértices da seguinte forma: calculamos a distância entre o y do vértice em questão e do vértice com o qual estamos a comparar, se essa mesma distância for inferior ou igual ao salto máximo que o Hill poderá fazer, então comparamos a distância entre os dois vértices e se essa também for inferior ou igual ao salto máximo significa que podemos adicionar o vértice com o qual estamos a comparar à lista *destination* do vértice em questão; caso contrário, quando encontramos um vértice que, calculando a distância dos y's, essa distância é superior ao salto máximo, então quebramos o ciclo, não sendo necessário verificar os restantes vértices, então pegamos noutro vértice e repetimos o processo até todos os pontos terem sido utilizados.

Complexidade

Temporal

A função main() chama dois métodos que afetam a complexidade temporal do algoritmo, o método *addToArray()* na classe *Climb*, que serve para passar os vértices ordenados da *Queue* para o *array*, então a complexidade pode ser considerada O(n*log(n)), sendo n o número de vértices presentes na *Queue*. O segundo método acima mencionado é o método *build()* que também se encontra na classe *Climb*, que tem como função construir o grafo, então na pior das hipóteses a complexidade temporal é O(n*log(n)), pois n, tal como anteriormente referido, é o número de vértices.

Por fim, no método que é responsável por calcular o valor final desejado, que é o *bestPath()*, na pior das hipóteses é necessário visitar todos os nós presentes no grafo duas vezes, logo, no pior dos casos, a complexidade temporal é de $O(n^2)$. Como este método pode ser chamado mais do que uma vez, consoante o número de testes a serem executados, então na verdade a complexidade temporal é $p*O(n^2)$, sendo p o número de testes a serem feitos.

Assim, é possível concluir que a complexidade temporal final do algoritmo é de:

$$O(n * log(n)) + O(n * log(n)) + p * O(n) = O(2(n * log(n)) + p * n^2)$$

Espacial

A inicialização do *array* do tipo *Vertex v*, presente na classe *Climb*, apresenta como complexidade temporal O(v), sendo que v é o número de vértices disponibilizados no *input*.

Outro fator que afeta a complexidade espacial do algoritmo é a inicialização da lista *destination*, de tipo *Edge*, de cada um dos vértices, ou seja, podemos considerar a sua complexidade espacial como O(e), sendo e o número de listas *destination*.

Desta forma, é possível concluir que a complexidade espacial final do nosso programa é:

$$O(v) + O(e) = O(v + e)$$

Comentários adicionais

Ao longo da realização deste trabalho sentimos algumas dificuldades que conseguimos ultrapassar com sucesso tais como, entender como a implementação da interface *Comparable* nos poderia ajudar a organizar os vértices pela altura, recorrendo a uma *priority Queue*; o facto de não necessitarmos de comparar a distância de todos os nós, pois isso levaria a que o nosso algoritmo levaria muito tempo a correr, o que levaria a " *Time Limit Exceeded*"no *Mooshak*. Mas a maior dificuldade sentida foi em relação ao cálculo da distância entre dois vértices pois, após várias " *Wrong Answer*"no *Mooshak*, conseguimos descobrir que o nosso erro seria no arredondamento de *double* para *int*, pois era necessário arredondar sempre para cima, sendo então o método a utilizar neste caso o *Math.ceil()*.

Conclusão

Após o término deste trabalho, concluímos que, apesar das dificuldades que fomos enfrentando ao longo da realização do mesmo, fomos capazes de concluir esta tarefa com sucesso.

De referir ainda que este trabalho revelou-se bastante importante para a consolidação de conhecimentos na matéria dos grafos, pois permitiu-nos alterar a maneira de implementar um grafo e o resultado ser o expectável na mesma.

Bibliografia

Pedro, Vasco in "Aulas de Estruturas de Dados e Algoritmos II". 2022 in Universidade de Évora.

Pedro, Vasco in eda2-t2-climber.pdf. Universidade de Évora's MOODLE.

https://www.geeksforgeeks.org/graph-and-its-representations/

https://techvidvan.com/tutorials/breadth-first-search/

https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm