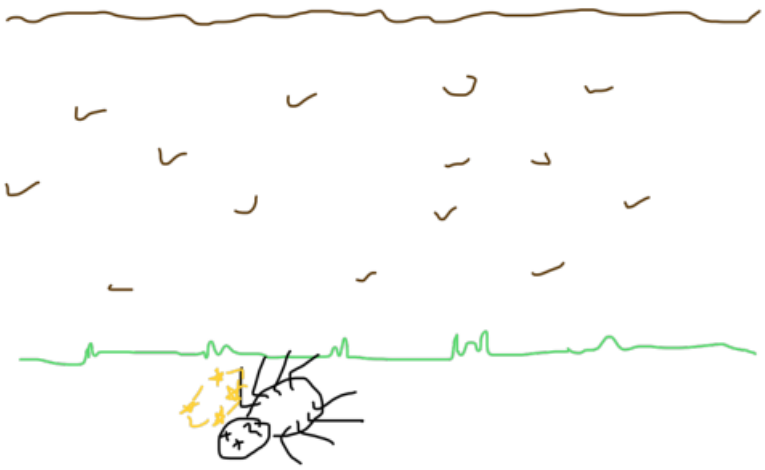
Relatório do Trabalho

Disciplina de Estruturas de Dados

e Algoritmos II

–Problem: Hill, the Climber –

29/05/2022



Membros do grupo (g227):

-Diogo Carreiro nº48729

-Rodrigo Alves nº 48681

Introdução/Descrição

Este trabalho tem como objetivo, dado um certo número de pontos, assim como as coordenadas que definem cada um deles, encontrar se possível o melhor caminho até uma certa altura, tendo em conta que o caminho será definido com base no alcance de Hill.

Assim, a partir da primeira linha do input, que contem três inteiros, número de pontos(N), altura (H) e número de diferentes casos (C), sendo que cada caso contem um alcance. As linhas seguintes irão conter as coordenadas x e y de cada ponto. Quando forem introduzidos todos os pontos, será inserido o alcance para cada caso. Estes dados são introduzidos no input, e a partir dos mesmos podemos calcular o caminho que contem menos pontos. O output esperado poderá ser o número mínimo de pontos que definem o melhor caminho, ou *unreachable*, no caso de não existir um caminho possível tendo em conta um certo alcance definido num caso.

Descrição do Programa

Como foi mencionado anteriormente, a primeira linha do input terá três inteiros (número de pontos, altura e número de casos, respetivamente), sendo que estes vão ser lidos com o auxílio do *BufferedReader*. De seguida, serão introduzidas nas linhas seguintes as coordenadas x e y dos N pontos presentes no local em que Hill pretende escalar. Ao serem introduzidos as posições dos N pontos, serão colocados os alcances até ao número de casos C mencionados na primeira linha do *input*.

Neste programa, foram criadas duas classes, classe *Point* e a classe *Climb*, sendo que a primeira é usada para definir um ponto com coordenadas que o representam, como também serve para obtermos as informações acerca de um determinado ponto, ou seja o x e o y correspondentes, a partir dos métodos *value_x* e *value_y* respetivamente. Nesta classe *Point* temos um método *compareTo* que está definido na interface *Comparable*, este método é usado para comparar as ordenadas de dois pontos distintos. A segunda classe mencionada anteriormente é a classe *Climb* que serve para criar o nosso grafo pesado não orientado, desta forma definimos o método *create_graph*. A classe também é usada para atualizar a nossa lista de pontos ordenados por ordem crescente pela ordenada utilizando o método *order*. Nesta classe é possível adicionar um arco ao grafo utilizando o método *addEdge*, um ponto à nossa lista de pontos através do método *addList*, calcular a distância entre dois pontos com o método *dist* e por último a pesquisa em largura do nosso grafo criado a partir dos alcances inseridos no input através do método *BFS*.

Ao ser introduzido o *input* que já foi mencionado anteriormente, com base nas sugestões dadas em aula pelo professor responsável pela cadeira, o grupo decidiu que o conjunto de pontos inseridos no terminal iriam ser adicionadas a uma lista definida na classe *Climb* e quando estes fossem inseridos à lista de

pontos P, da classe *Climb*, seria posteriormente ordenada por ordem crescente relativamente às ordenadas de cada ponto. Esta ordenação escolhida permite diminuir o número de vezes que é calculado a distância entre dois pontos, isto é, os pontos ao estarem ordenados pelos y, e a diferença entre os mesmos for menor ou igual ao alcance, e de seguida a distância for também menor ou igual ao alcance, estão criadas as condições de criar um arco no nosso grafo entre estes pontos. Caso contrário, não é criado um arco e consequentemente não são avaliadas mais distâncias, porque se um dado ponto da lista com outro ponto não respeitarem a condição de a diferença dos seus y ser menor ou igual ao alcance, então não é necessário averiguar as posições seguintes da lista pois também não irão verificar a condição anteriormente mencionada, devido ao facto de a lista estar ordenada pelos y de cada ponto.

A criação do grafo é gerada com base no maior dos alcances, esta abordagem permite que não seja feito um grafo para cada caso, o que levaria tempo e ocupação de uma boa parte de memória, pois o maior dos alcances inclui todos os alcances menores. Assim, é criado um único grafo que posteriormente será utilizado para todos os casos introduzidos no terminal. O grafo é construído utilizando a classe *Graph*, e cada arco do grafo com a classe *Edge*, sendo que a primeira classe mencionada possui um método add_edge que cria os arcos do grafo, sendo que o mesmo é um grafo não orientado, neste método é chamada a classe *Edge* para gerar o arco, que vai ser definido com um peso, que representa a distância entre dois pontos, um início e fim que são representados pelos índices em que os pontos se encontram na lista ordenada de pontos (P) isto num dado sentido, o mesmo acontece para o sentido inverso.

A criação de todos os arcos bidirecionais, permite obter o grafo que vamos utilizar para encontrar o caminho mais curto, ou seja, o que possui o menor número de pontos. Para identificar este caminho, utilizamos um dos percursos básicos em grafos, percurso em largura (*Breadth-first search*), que permite descobrir a distância de um determinado ponto relativamente ao ponto onde foi iniciado a contagem, ou seja, neste caso o/s ponto/s que se encontram a uma distância menor ou igual ao alcance relativamente ao chão, são considerados pontos inicias. A partir dos mesmos, serão iniciados os possíveis percursos até ao ponto mais próximo ou igual à altura que se pretende alcançar, em certas situações poderá acontecer não existir nenhum caminho possível e será retornado como *unreachable*, caso não seja necessário utilizar nenhum ponto desde base até à altura pretendida, pois o alcance é maior ou igual à altura, é retornado zero “0”, caso contrário será retornado um número representativo do menor percurso encontrado no grafo gerado.

Métodos e Classes

- **Classe *Point***

Point (construtor) - define as coordenadas de um determinado ponto;

value_x - retorna a abcissa de um ponto;

value_y - retorna a ordenada de um ponto;

compareTo - neste método faz a comparação entre dois pontos tendo em conta a sua ordenada;

- **Classe *Climb***

Climb (construtor) - define o número de nós que o grafo irá possuir como também define o tamanho da nossa lista;

add_to_graph - adiciona o arco que contém os índices dos pontos *p1* e *p2* nas suas extremidades, com um peso igual à distância entre os mesmos;

addList - adiciona o ponto *p* à lista de pontos *P* definida na classe;

order - ordena a nossa lista de pontos P com base nas ordenadas dos pontos, esta ordenação é feita com o auxílio do método *Collections.sort()*;

create_graph - com base no maior alcance e no número de pontos, são definidos os arcos do grafo, que verificam a condição da distância entre os mesmos (utilizamos o método *dist*) ser menor ou igual ao maior alcance, caso isso se verifique adicionamos o arco ao grafo, com a ajuda do método *add_to_graph*;

dist - calcula a distância entre dois pontos;

BFS – inicialmente todos os pontos vão ser identificados com a cor “WHITE”, esta cor indica que o ponto não foi visitado, e também lhes vão ser atribuídos uma distância com um valor máximo positivo. Estas duas informações acerca de cada ponto são armazenadas em dois arrays, *color* e *dist_array*. De seguida, é inicializada uma *Queue S* para receber todos os pontos em que a sua ordenada tem um valor que é menor ou igual ao alcance do parâmetro do método, para estes casos estes pontos serão identificados com a cor “GREY” no array *color*, e também são definidos com uma distância igual a um “1” no dist_array nas respetivas posições que os representam. Depois de ser terminada a adição dos pontos à *Queue S*, irá ocorrer o seguinte processo: enquanto S não estiver vazia, ou seja, sem nenhum índice de um ponto contido na mesma, será retirado um elemento, e este será utilizado para averiguar os índices dos seus vértices adjacentes. Desta forma, será feita uma avaliação se os mesmos já foram visitados e se o peso do arco que os define é menor ou igual ao alcance que está definido no parâmetro. Se o vértice adjacente estiver definido de cor “WHITE” e o peso do arco formado é menor ou igual ao alcance, este vértice será definido com a cor “GREY” e a sua distância será atualizada usando a distância do último ponto retirado da *Queue S* adicionando um “1” a este valor, e por último inserimos o índice do vértice adjacente na lista de pontos à *Queue S*. No final de cada iteração do ciclo while, o ponto retirado da Queue será definido com a cor “BLACK”. Finalmente, é criada uma variável (*mindist*) que irá ser inicializada com um valor máximo positivo e que serve para guardar o resultado final, ou seja, irá ser percorrido o *dist_array* de forma

decrescente, por isso começamos a averiguar a última posição do array até à primeira posição cuja ordenada do ponto mais o alcance seja menor do que a altura. Caso a variável *mindist* que contem o menor número de pontos percorridos num dado percurso seja maior do que a distância de um dos pontos, esta vai ser atualizada. No fim do método será retornado o valor guardado na variável *mindist*.

- **Classe *Graph***

Graph (construtor) - define o número de nós de um determinado grafo, cria um array de arcos com um tamanho igual ao número de nós, representando este a lista de adjacências;

addEdge - adiciona à lista de adjacências dos dois nós correspondentes o arco cujas as extremidades são os índices de p1 e de p2 assim como o peso que representa a distância entre eles;

- **Classe *Edge***

Edge (construtor) – define as extremidades de um determinado arco assim como o seu peso, que é a distâncias entre os nós do mesmo;

Complexidade Temporal e Espacial

A complexidade espacial na *main* será linear com o número de pontos inseridos no input, ou seja, O(N), sendo *N* o número de pontos inseridos. A justificação para a escolha desta complexidade deve-se ao facto de o número de pontos na maior parte dos casos ser maior do que o número de casos introduzidos no *input*, para além de que o número máximo de casos que podem ser introduzidos é menor do que o número máximo de pontos inseridos no *input*. A complexidade temporal será igual O(N) sendo *N* o número de pontos inseridos, porque na *main* definida o primeiro ciclo for será executado mais vezes do que o segundo for, pois como o número de pontos tende a ser maior do que o número de casos introduzidos no *input,* logo o número de vezes do primeiro ciclo for será maior.

A complexidade temporal e espacial da classe *Graph* será linear ao número de nós que o grafo conterá, O(N), sendo *N* o número de nós. Relativamente à classe *Climb,* no método create_graph, a complexidade temporal será igual O(N*M), sendo N o número de pontos menos um “1”, e M o número de iterações do ciclo while. No método BFS, a complexidade espacial será O(N), sendo *N* o número de pontos inseridos no input, esta complexidade deve-se ao facto de reservarmos memória para guardar as informações acerca de todos os pontos. A complexidade temporal irá ser O(N*A), sendo *N* o número de pontos do grafo e A o máximo número de vértices adjacentes de cada ponto, esta atribuição justifica-se pelo número de vezes que o ciclo *while* deste método é executado, isto é, enquanto a *Queue* não estiver vazia e enquanto um determinado vértice contiver vértices adjacentes.