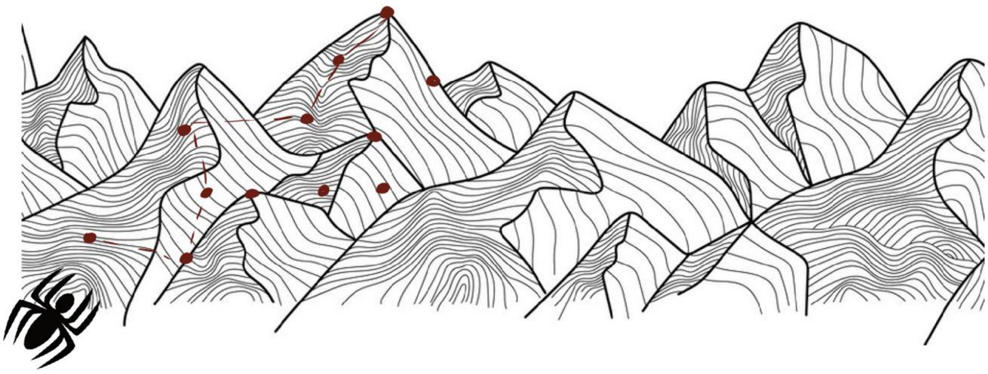
UNIVERSIDADE
DE ÉVORA

Problem:

Hill the Climber

Relatório do Trabalho Prático



Curso: Engenharia Informática

Disciplina: Estrutura de Dados 2

Docentes: Vasco Pedro

Entregue Abril 2022

Alunos:

Joana Carrasqueira nº48566

João Condeço nº48976

g202

Índice

1. Descrição do Algoritmo

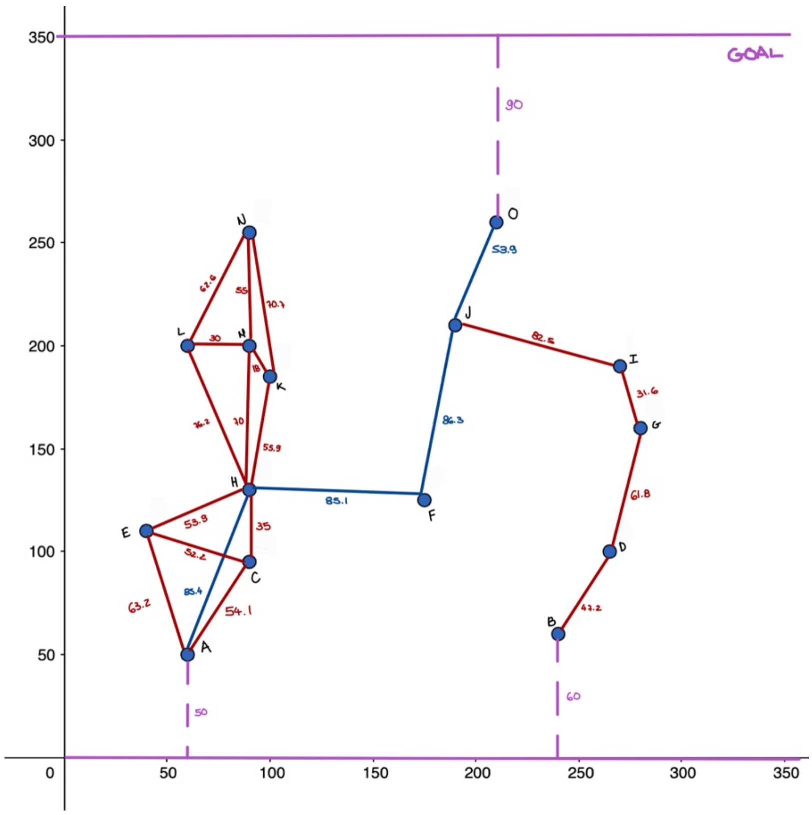
De modo, a calcular o trajeto menor para o Hill alcançar o topo da montanha, tendo em conta o alcance (reach) do salto deste, com o número mínimo de pontos necessários para alcançar o topo foi concebido um algoritmo que obtém o valor pretendido com uma complexidade constante. Para tal foi feita a representação dos grafos construídos para aplicar este algoritmo no problema em causa.

O programa desenvolvido começa por construir o grafo com base nas distâncias entre os pontos recolhidos e no alcance (range) máximo do problema (de forma a não ter que construir mais do que um grafo). Tendo o grafo construído irão ser feitos os cálculos para os diferentes alcances onde primeiramente é avaliado se o salto é possivel diretamente do chão até ao topo (neste caso o resultado será zero pois não necessita de pontos). De seguida é verificado se é possivel efetuar o percurso com apenas um ponto, se isto se verificar não serão necessários mais cálculos e a resposta é 1, caso contrário serão recolhidos os pontos iniciais possíveis (aqueles aos quais a distância entre o chão e os mesmos é menor ou igual ao alcance de salto) e, seguidamente, aplicado o algoritmo de percurso em largura ao grafo.

No algoritmo que percorre o grafo são adicionados à Queue os pontos iniciais e marcados como encontrados, mas não processados (cor Grey). De seguida é feito o processo de cada ponto onde são adicionados à Queue os pontos adjacentes do ponto a ser processado, que ainda não foram visitados (cor White) e o peso do ramo (ou seja, a distância entre os pontos) é menor ou igual ao alcance. Se, a partir de algum destes pontos, for possivel chegar ao objetivo, é retornado o número de pontos percorridos. Se não for encontrado nenhum caminho viável e todos os pontos tiverem sido percorridos, é retornado -1.

2. Descrição dos grafos utilizados

- No grafo construído para representar o exemplo dado pelo professor no enunciado foram feitas todas as ligações com o alcance 90 e considerados os pontos iniciais A e B.



Input:

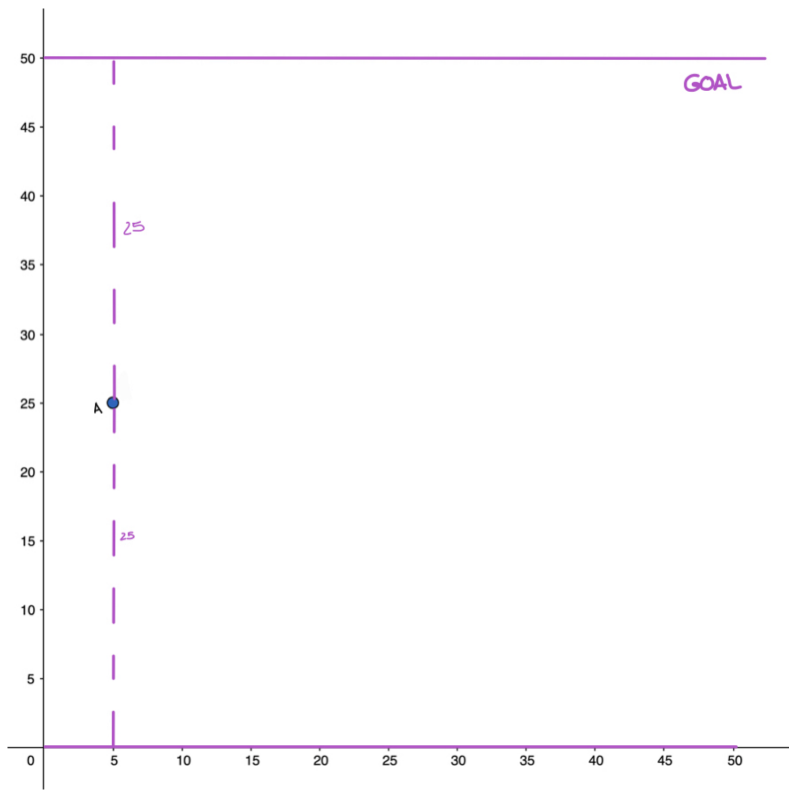
15 350 2
60 50
240 60
90 95
265 100
40 110
175 125
280 160
90 130
270 190
190 210
100 185
60 200

90 200
90 255
210 260
90
89

Output:
5
unreachable

Fig. 1 – Grafo referente ao exemplo 1

- Também foi construído o grafo correspondente a outro exemplo indicado pelo professor (Dúvida 3 do moodle). Onde, com um dos alcances, é possivel efetuar o salto sem recorrer a nenhum ponto.



Input:
1 50 2
5 25
30
60

Output:
1
0

Fig. 1 – Grafo referente ao exemplo 2

3. Análise da complexidade

- Complexidade espacial

  Sendo a complexidade espacial o espaço exigido pelo algoritmo para executar até ao fim em função do input, tendo em conta os 3 valores escalares (colour, d e heights) por cada vértice do grafo ($\Theta(V)$) e a fila que, no pior dos casos, terá $|V| - 1$ vértices ($O(V)$) então a complexidade espacial será $\Theta(V)$.

- Complexidade temporal

  Complexidade temporal consiste na porção de tempo que o algoritmo demora a ser executado em função do input. Assim, a função com a maior complexidade temporal no pior caso será a complexidade temporal do programa. Relativamente ao BFS (percurso em largura), no pior dos casos as operações de ENQUEUE e DEQUEUE, tem custo $\Theta(1)$, visto que estas, se repetem tantas vezes quanto o número de vértices do grafo, cujo o custo do primeiro ciclo é $\Theta(V)$, sendo V o número de vértices. Quanto ao segundo ciclo, este irá se repetir tantas vezes quanto o número de pontos iniciais tendo um custo de $\Theta(SP)$. Nos últimos dois ciclos, considerando que todas as operações, incluindo a criação de uma fila vazia, ENQUEUE e DEQUEUE, têm custo $\Theta(1)$, e considerando E, como o conjunto de vértices adjacentes a um ponto no grafo podemos concluir que o custo destes será $O(E)$, no pior dos casos. Assim, a complexidade temporal do algoritmo será $O(V + E + SP)$.

  Relativamente à função main (ignorando a recolha do input), uma vez que o ciclo, que tem maior custo, no pior dos casos, é percorrido tantas vezes quanto o número de casos, e o ciclo no seu interior corre tantas vezes quanto o número de holding points, então a complexidade temporal será de $O(Ncases \times HP)$. Quanto à construção do grafo, uma vez que este é quase linear podemos concluir que a sua complexidade temporal será de $O(HP^2)$. Concluímos assim que a complexidade temporal da função main é de $O(HP^2)$.

Pseudo-código

```
BFS(STARTINGPOINTS)

let colour[0..g.nodes] be a new array;
let d[0..g.nodes] be a new array;
let h[0..g.nodes] be a new array;
for u <- 0 to g.nodes do
    colour[u] <- WHITE;
    d[u] <- INFINITY;
for each vertex u in startingPoints do
    d[u] <- 0;
    colour[u] <- GREY;
Q <- EMPTY       //Fila (FIFO)
for each vertex i in startingPoints do
    ENQUEUE(Q, i);
while Q != EMPTY do
    u <- DEQUEUE(Q)
    for each vertex v in g.adjacents[u] do
        if colour[v] = WHITE and v.weight <= range then
            colour[v] <- GREY
            d[v] <- d[u] + 1
            heights[v] <- v.y
            ENQUEUE(Q, v);
            if v.y + range >= goal
                return d[v] + 1
    colour[u] <- BLACK
return -1
```

Código em Java:

```java
public int bfs(List<Integer> startingPoints){
            Colour[] colour = new Colour[this.g.nodes];
            int[] d = new int[this.g.nodes];  // => Distância de cada ponto à origem
            int[] heights = new int[this.g.nodes];

            for (int u = 0; u < this.g.nodes; u++){
                  colour[u] = Colour.WHITE;
                  d[u] = INFINITY;
            }

            for (int i : startingPoints) {
                  d[i] = 0;
                  colour[i] = Colour.GREY;
            }

            Queue<Integer> Q = new LinkedList<>();
            for (Integer i : startingPoints) {
                  Q.add(i);
            }
            while (!Q.isEmpty()){
                  int u = Q.remove();
                  for (Edge v : this.g.adjacents[u]){
                        if (colour[v.dest()] == Colour.WHITE &&
                           (v.weight() <= this.range)){

                              colour[v.dest()] = Colour.GREY;
                              d[v.dest()] = d[u] + 1;
                              heights[v.dest()] = v.y();
                              Q.add(v.dest());

                              if(v.y() + this.range >= goal){
                                    return d[v.dest()] + 1;
                              }
                        }
                  }
                  colour[u] = Colour.BLACK;
            }
            return -1;                        // => Se não for encontrado nenhum ponto
      }
```

4. Decisões e dificuldades encontradas

Na elaboração deste trabalho surgiram alguns obstáculos, dos quais se destacam a criação de um grafo para cada alcance de salto, o que levava a um excesso de tempo, mas foi corrigido, fazendo apenas um grafo com o alcance máximo, e posteriormente no BFS verificarmos se o salto é possivel.

Outro problema que surgiu, foi relativamente ao tipo de Queue que foi usado no BFS, que inicialmente era uma Priority Queue, no entanto como a ordem de remoção por prioridade não correspondia ao pretendido, foi optado por implementar uma Queue com LinkedList.