* **Descrição do Problema e da Solução**

Ao processar o input, criamos o nosso grafo , representado por um vetor de vetores de adjacências de cada vértice. Ou seja, em cada posição do vetor, temos um vetor composto por todos os vértices tal que e se encontram ligados por uma aresta em . Passando agora para o algoritmo em si, a ideia é a de encontrar as componentes fortemente ligadas, SCC, e calcular o maior caminho.

Ao executarmos uma DFS no grafo, a ordem de tempos de fim corresponde a uma “ordem topológica” (a menos de ciclos). Portanto, para encontrarmos o número máximo de saltos possíveis num grafo, aproveitamos essa mesma ordem e ignoramos eventuais SCC. Deste modo, escusamos de calcular o grafo das SCC. Temos então duas chamadas da DFS, a primeira clássica, em que nos interessa apenas armazenar a ordem de tempos de fim, e a segunda, em que percorremos o grafo para dar conta de saltos de vizinhos e unificar SCC, aproveitando evidentemente o maior dos possíveis saltos para a solução final.

* **Análise Teórica**

Pseudocódigo:

(é uma variável global.

A variável é importante, permite-nos percorrer o grafo na ordem pretendida)

DFSVisit(, , ,):

= 0

= empty stack

.push()

while is not empty:

j = .top()

.pop()

if [j] is :

[j] =

.push(j)

size = size of g[j]

for k from 0 to size - 1:

if [[j][k]] is :

.push([j][k])

else if [[j][k]] is and < [[j][k]] + 1:

= [[j][k]] + 1

else if [j] is :

If [j] > :

= [j]

[j] =

.append(j)

[j] =

return

(O valor de retorno só interessa na segunda chamada da DFS. Esta função é chamada várias vezes, com a ordem pretendida.)

parseDimensions():

read , from input

= initialize 2 vector of size ( + 1)

= initialize 2 vector of size (+ 1)

parseEdges():

for from 1 to :

read , j from input

add j to g[]

add to gT[j]

1. Leitura dos dados de entrada: Simples leitura do input, com um ciclo a depender linearmente de (número de arestas). Logo, O().
2. Processamento da instância: Inserir o vértice no vetor de vetor de adjacências. Logo, O(1).
3. Aplicação da DFS para o grafo: A versão da DFS que utilizámos é iterativa. Para tal, o algoritmo utiliza uma pilha para acompanhar os vértices a serem visitados. Ele explora o máximo possível ao longo de cada vértice antes de retroceder. A complexidade é então determinada pelo número de vértices e arestas no grafo, dado que cada aresta e cada vértice são visitados uma vez. Logo, O(+).
4. Apresentar o resultado final: É feito um print do valor que se encontra armazenado na variável . Corresponde a uma complexidade O(1).
5. Complexidade global da solução: O() + O(1) + 2O(+) + O(1) = O(+).

* **Avaliação Experimental dos Resultados**

Neste gráfico, apresentamos o tempo de execução do algoritmo em função da quantidade prevista pela análise teórica O((, )) (onde . Para tal, utilizámos 12 instâncias espaçadas igualmente entre si.

A graph with blue dots

Description automatically generated

Observamos uma relação linear com os tempos no eixo dos YY, confirmando que a nossa implementação está de acordo com a análise teórica O((, )).