Relatório 2º projeto ASA 2023/2024

Grupo: AL008
Alunos: Vasco Conceição (106481), Henrique Luz (99417)

❖ Descrição do Problema e da Solução

Ao processar o input, criamos o nosso grafo g, representado por um vetor de vetores de adjacências de cada vértice. Ou seja, em cada posição u do vetor, temos um vetor composto por todos os vértices v tal que u e v se encontram ligados por uma aresta em g. 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:

(*order* é uma variável global.
A variável *i* é importante, permite-nos percorrer o grafo na ordem pretendida)

```
DFSVisit(g, colors, jumps, i):
  max= 0
  s = empty stack
  s.push(i)
  while s is not empty:
    j = s.top()
    s.pop()
    if colors[j] is WHITE:
      colors[j] = GREY
      s.push(j)
      size = size of g[j]
      for k from 0 to size - 1:
        if colors[g[j][k]] is WHITE:
          s.push(g[j][k])
        else if colors[g[j][k]] is BLACK and max< jumps[g[j][k]] + 1:
          max= jumps[g[j][k]] + 1
    else if colors[j] is GREY:
      If jumps [j] > max:
        max = jumps[j]
      jumps[j] = max
      order.append(j)
      colors[j] = BLACK
  return max
```

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

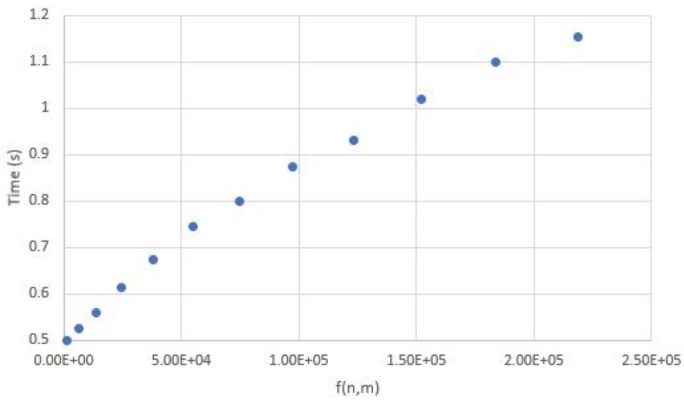
Grupo: AL008
Alunos: Vasco Conceição (106481), Henrique Luz (99417)

```
parseDimensions():
|  read V, E from input
|  g = initialize 2D vector of size (v + 1)
|  gT = initialize 2D vector of size (v + 1)
parseEdges():
|  for n from 1 to E:
|  |  read i, j from input
|  |  add j to g[i]
|  |  add i to gT[j]
```

1. Leitura dos dados de entrada: Simples leitura do input, com um ciclo a depender linearmente de E (número de arestas). Logo, O(E).
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(V+E).
4. Apresentar o resultado final: É feito um print do valor que se encontra armazenado na variável res. Corresponde a uma complexidade O(1).
5. Complexidade global da solução: O(E) + O(1) + 2O(V+E) + O(1) = O(V+E).

❖ Avaliação Experimental dos Resultados

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



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