Relatório 2º projecto ASA 2023/2024

Grupo: AL003

Aluno(s): Mafalda Szolnoky Ramos Pinto Dias (106494) e Francisco Lourenço Heleno (106970)

Descrição do Problema e da Solução

O problema baseia-se em, tendo os dados de uma rede social representativa das interações reais entre a população portuguesa, estudar o pior caso de propagação de uma dada infeção em Portugal, assumindo que indivíduos que se conhecem mutuamente de forma direta ou indireta, ficam infetados instantaneamente.

Para resolver o problema, definimos a rede social como um grafo dirigido de relações entre indivíduos (criando também o seu transposto). De seguida, aplicamos uma variação do algoritmo de Kosaraju (que realiza duas procuras em profundidade-primeiro iterativas, uma no grafo normal e outra no grafo transposto) de forma a que este, para além de encontrar os SCCs, calcule também o maior número de saltos que a doença pode fazer. Assim, **calculamos para cada vértice o número de arcos do maior caminho de um source component até ao SCC do vértice atual**: para cada vizinho do vértice atual, caso o mesmo pertença a um SCC diferente do vértice atual, o resultado do vértice atual será o máximo entre o resultado do vértice atual e o resultado do vizinho + 1. Caso os dois vértices pertençam ao mesmo SCC o resultado vai ser o máximo entre o resultado do vizinho e o resultado do vértice atual. Após a realização das duas DFS, a resposta ao problema será o maior dos resultados calculados.

Análise Teórica

Pseudo código da segunda DFS:

```
dfsT(start, visited, sccCounter)
    let s be a stack
    s.push(start)
    maxResult = 0
    visited[start] = true
    sccNumber[start] = sccCounter
    calcStart = true

    while !s.empty()
        current = s.top()
        hasUnvisitedNeighbor = false

        for neighbor in graphT[current]
            if !visited[neighbor]
                s.push(neighbor)
                visited[neighbor] = true
                hasUnvisitedNeighbor = true
                break;

        if !hasUnvisitedNeighbor
            if !calcStart
                s.push(start)
                calcStart = true
                current = s.top()

            s.pop()
            sccNumber[current] = sccCounter

            if !s.empty()
                for neighbor in graphT[current]
                    if sccNumber[current] != sccNumber[neighbor] &&
                       sccNumber[current] != 0 && sccNumber[neighbor] != 0
                        result[current] = max(result[current], result[neighbor] + 1)
                    else
                        result[current] = max(result[current], result[neighbor])
                    maxResult = max(maxResult, result[current])

    return maxResult
```

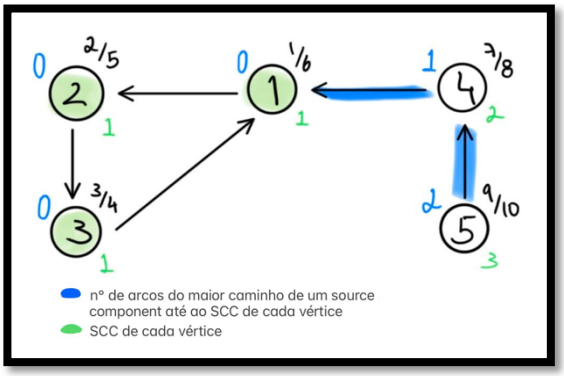
No total de todas as chamadas a dfsT, o nº de iterações do loop será linear com o número de vértices, n.

Ao todo, o corpo do loop é executado uma vez por cada arco do grafo.

Upper bound por nº de iterações:
m vezes

Não altera a complexidade da função porque só é executado 1x por vértice do grafo e cada um destes vértices tem um número constante de vizinhos.

Exemplo de funcionamento do programa:



Relatório 2º projecto ASA 2023/2024

Grupo: AL003

Aluno(s): Mafalda Szolnoky Ramos Pinto Dias (106494) e Francisco Lourenço Heleno (106970)

Observações: visited é um vetor inicializado com todas as entradas a false, result e sccNumber são dois vetores globais inicializados a 0. calcStart é uma flag para verificar se o resultado do vértice start, que é passado como argumento da função, já foi calculado ou não. graphT é uma variável global que representa o grafo transposto. Realçamos ainda que não foi colocado acima o pseudo código da primeira DFS iterativa uma vez que esta é bastante simples e apenas serve para preencher uma stack com os vértices na ordem topológica inversa.

- Leitura dos dados de entrada: simples leitura do input com ciclo a depender linearmente do número de relações entre indivíduos (m). Logo, Θ(m);
- Processamento da instância: criação de dois grafos (normal e transposto) e inserção dos arcos em tempo constante nos mesmos. Logo, O(1);
- Procura em profundidade-primeiro no grafo normal para encontrar ordem topológica inversa. Logo, O(n + m);
- Procura em profundidade-primeiro no grafo transposto para encontrar SCCs e calcular o número de saltos. Logo, O(n + m);
- Apresentação dos dados: O(1);

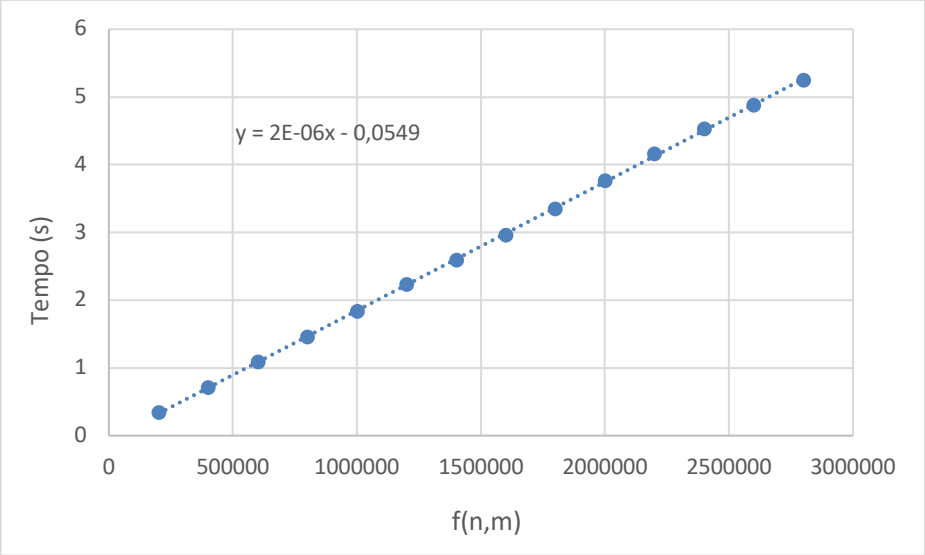
Complexidade global da solução: O(m + n + m) = O(n + 2m) ≈ **O(n + m)**

Avaliação Experimental dos Resultados

Para a realização de experiências, utilizámos o gerador de grafos aleatórios para perceber se o nosso programa estava de acordo com a análise teórica prevista.

Testámos assim o nosso código com 14 instâncias de tamanho incremental de modo a levar o nosso programa "ao limite".

O gráfico seguinte representa o tempo de execução do nosso programa em função da quantidade prevista pela análise teórica, ou seja, f(n, m) = n + m.



Analisando o gráfico acima, é evidente a relação linear entre a complexidade teórica prevista, O(f(n, m)), e os tempos de execução do nosso programa, o que confirma que a nossa implementação está de acordo com a análise teórica.