Introdução

O seguinte relatório aborda a solução e respetivos testes e análise ao problema proposto como primeira parte do projeto de Análise e Síntese de Algoritmos do primeiro semester do ano escolar 2013/2014.

O problema trata a partilha de informação entre pessoas e como é que aquela navega entre vários grupos. Assim, o objetivo é classificar as pessoas em grupos de partilha de tal forma que quando uma das pessoas do grupo recebe algo, todas as outras também irão receber.

Através de um *input* com o número de pessoas e os canais de partilha existentes entre elas seguidos da identificação das ligações individuais entre elas, o programa resultante tem de apresentar como output o tamanho do maior grupo de partilha, ou seja, o número de pessoas desse grupo, a quantidade de grupos de partilha e a quantidade de grupos máximos de pessoas que partilham informação apenas dentro do grupo.

Descrição da Solução

Primeiramente, tivemos de optar pela escolha da linguagem de programação na qual iríamos escrever o código. Optámos por C.

Para conceber o programa, criámos um mapeamento entre os conceitos pessoa e partilha para vértice e arco, respetivamente, de um grapo dirigido. Assim, cada grupo de pessoas é um Componente Fortemente Ligado.

Para navegar, explorar o gráfico e descobrir os CFL, implementámos uma variação do Algoritmo de Tarjan.

Após definir a estratégia a utilizar, decidimos que a estrutura que representaria o grafo é um *array* de estruturas, cada uma das quais com um ponteiro para um vértice e um inteiro a representar se esse vértice está ou não na *stack*. Cada vértice é também uma estrutura, cada qual com uma lista de arcos e três inteiros: dois deles necessários para o Algoritmo de Tarjan (*d* e *low*) e um terceiro para o identificar (*id*). Igualmente, cada arco tem um inteiro para identificar o seu alvo e um ponteiro para o próximo arco que o vértice tem.

A razão de escolha das estruturas anteriormente mencionadas prende-se com o facto de ser possível navegar de forma praticamente imediata entre vértices.

Em baixo, a representação da estrutura do grafo com N vértices e E arcos.

index = 1	*index = 2*	*index = 3*		*index = N*
int stack	*int stack*	*int stack*	…	*int stack*
*struct vertex *vertex*	*struct vertex *vertex*	*struct vertex *vertex*	…	*struct vertex *vertex*

Figura 1 – Esquema da variável *stack* denominada no programa.

*struct edge *connect*
int d, low, id

Figura 2 – Esquema da estrutura *vertex*.

*struct edge * next*
int id

Figura 3 – Esquema da estrutura edge.

75541 – Tomás Alves
75551 – Miguel Santos

A estrutura duma pilha é uma lista simplesmente ligada e é denominada por *list*. A mesma foi escolhida devido às características temporais e espaciais de acesso e adição de informação que aquela estrutura proporciona. Cada nó da pilha tem um inteiro que corresponde ao *id* do vértice que foi alvo da função *push*. Existe também uma estrutura que aponta para o primeiro membro da stack, denominada *stack_head*. A função *pop* retorna o inteiro que está na cabeça da pilha para poder comparar posteriormente. Abaixo, a representação da mesma com três elementos.

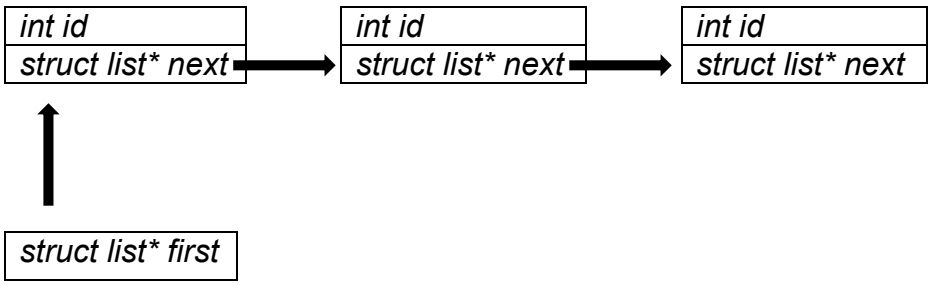

Figura 3 – A estrutura da pilha que é usada no programa exemplificada com 3 elementos e a estrutura que aponta para a cabeça da pilha.

Para a apresentação dos três outputs, queriámos três variáveis, as quais representam o maior CFL, o número de CFL do gráfico e o número de CFL sendo que nenhum dos vértices constituintes tem uma ligação com outro CFL. Estas variáveis são atualizadas na resolução do Algoritmo de Tarjan aquando de uma ou múltiplas execuções da função *pop*, visto que aquele algoritmo trata de descobrir os CFL um a um, ou seja, permite um melhor controlo na manutenção e alteração das variáveis destinadas ao output.

Análise Teórica

Designarei arco por E e vértice por V nesta parte do relatório.

A função `Stack* create_stack(int n)` recebe como argumento o número de vértices que são necessários criar e depois entra num ciclo *for* para criar cada um individualmente, logo, a sua execução é um O(V).

Após a criação da estrutura que armazena os vértices, é criado um ciclo *for* que lê do *input* todas as ligações entre os vértices, ou seja, os arcos e cria-os individualmente, sendo que a este ciclo tem a sua execução em O(E). A criação de um arco é realizada em O(1), uma vez que quando é adicionado um novo arco na lista de arvos de um vértice, aquele é adicionado no início da lista.

Operações de *push* e *pop* na pilha são ambas executadas em tempo constante, ou seja, O(1).

O Algoritmo de Tarjan é executado em O(V+E), uma vez que são visitados todos os vértices e os respetivos arcos. A função `void scc_tarjan(Stack *stack, StackHead *head, int number_vertices)` tem um ciclo *for* que é executado uma vez por cada vértice e, sendo assim, o seu termo de execução é O(V). Por outro lado, consideramos que, no geral, um arco apenas é visitado (cada arco de cada vértice com a função `void tarjan_visit(Stack *stack, Vertex *vertex, StackHead *head)`) no

máximo duas vezes e, assim, o tempo de execução do algoritmo é linear no grafo em O(V+E).

Avaliação Experimental dos Resultados

No teste do programa que concebemos, submetê-mo-lo aos seguintes estes apresentados na tabela indicada abaixo. O programa passou a todos os testes com sucesso. Cada teste implica um problema diferente de identificação de CFL. É possível observer o número de vértices (pessoas), arcos (partilhas) e o tempo com que o programa resolveu os ditos testes.

Teste	Vértices	Arcos	Tempo (s)
1	8	10	< 0.01
2	7	5	< 0.01
3	100 000	5 000	0.02
4	100 000	8 000	0.03
5	100 000	80 000	0.14
6	100 000	5	0.01
7	100 000	199 997	0.35
8	100 000	99 999	0.21
9	100 000	99 999	0.25
10	100 000	199 997	0.66

Tabela 1 – Testes realizados sobre o programa com o *input* (Vértices,Arcos) e o respetivo tempo de execução.

Através da análise de resultados, podemos concluir que, tal como esperado, em grafos com menor número de vértices, o programa é mais rápido. O Crescente aumento de arcos faz igualmente com que o programa demore mais tempo. No entanto, ao analisar e comparar os testes 7 e 10, podemos verificar que para o mesmo *input* (V,E), a diferença do tempo de execução é enorme (0,31 segundos). Isto deve-se à diferença de CFL presentes em ambos os arcos, sendo que, para um maior número de CFL, o programa tem de executar mais funções.

Desta forma, concluímos que o programa que desenvolvemos é eficiente. No que toca à eficácia, reconhecemos que falha a alguns testes porn ão abordar todos os possíveis casos de identificação de CFL.