Relatório do 1º Projeto de ASA

Grupo al065

Filipe Azevedo – nº82468

Martim Zanatti – nº82517

Introdução

O objetivo deste projeto é encontrar um algoritmo eficiente para descobrir as pessoas fundamentais numa rede. Uma pessoa p é considerada fundamental se o único caminho para a partilha de informação entre outras duas pessoas r e s passa necessariamente por p (onde p diferente de r e p diferente de s).

Uma rede pode ser encarada como um grafo não dirigido, em que cada arco representa a partilha de informação e cada vértice representa uma pessoa. O algoritmo terá então que encontrar os vértices que correspondem a "pessoas fundamentais", que na teoria dos grafos se chamam "pontos de articulação".

Para realizar este algoritmo é necessário guardar os dados fornecidos à entrada numa estrutura. Estes dados contêm informações sobre o número de pessoas na rede, o número ligações entre pessoas (partilha de informação) e as pessoas entre as quais existe uma ligação.

De seguida apresentaremos a nossa solução para o problema assim como a sua complexidade, e também as dificuldades que tivemos ao implementar a nossa solução. O código do nosso grupo foi desenvolvido na linguagem de programação C.

Descrição da Solução

Para este projeto foram criadas três estruturas, a estrutura vértice, que representa um vértice e contém os atributos fundamentais para a resolução do algoritmo, uma estrutura chamada struct LISTAnode que guarda um vértice e

ponteiro para outro elemento dessa estrutura, e a estrutura do tipo lista de adjacências, que guarda a informação da rede e permite uma gestão eficiente da mesma.

No início, o programa lê do standard input dois inteiros que representam o número de pessoas e a as suas ligações, respetivamente. É criada uma lista de struct LISTAnode alocada dinamicamente com a dimensão do número de pessoas. À medida que lê do standard input as ligações entre as pessoas, vai guardando-as na lista de adjacências.

O algoritmo utilizado para a resolução eficiente deste problema foi o algoritmo de John Hopcroft, Robert Tarjan. Este algoritmo é baseado no algoritmo depth-first search.

Vai percorrendo o grafo vértice a vértice e à medida que passa em cada vértice V, vai atualizando os seus atributos: assinala que V já foi visitado, atualiza o valor da profundidade, atualiza o low (que corresponde à profundidade mínima de qualquer vértice W tal que exista uma ligação entre W e outro vértice U tal que U é V ou sucessor imediato ou não de V), põe o número de filhos a zero tal como o atributo que diz se é fundamental ou não.

Percorre os vértices adjacentes W, um a um e verifica se esses já foram visitados. Caso não tenham sido, o pai de W é igual a V e chama a função recursivamente em W. Quando a função retorna, aumenta o número de filhos de V, se o low de W for maior ou igual que a profundidade de V significa que possivelmente é fundamental, porque se V for a raiz, mesmo sendo o low de W maior ou igual que a profundidade de V o vértice V não é fundamental. De seguida verifica se o low de V é menor ou igual que o low de W, caso seja o low de V mantém-se inalterado, caso não seja o low de V fica com o valor do low de W, para garantir que o low de V é o mais baixo ou igual ao de W

Caso já tenha sido visitado verifica se W é diferente do pai de V, caso seja verifica se o low de V é menor ou igual que a profundidade de W. Se sim o low de V mantém-se inalterado, caso contrário o low de V toma o valor da profundidade de W.

Quando forem percorridos todos os vértices adjacentes a V, existe ainda uma condição final para excluir os casos em que a raiz do vértice é fundamental mas apenas tem um filho.

No final do algoritmo os atributos dos vértices devem estar atualizados nas posições “principais” da lista. Assim os vértices fundamentais ficam marcados como tal e basta produzir o output desejado: o número de pessoas fundamentais (pontos de articulação) na primeira linha e na segunda a pessoa fundamental com menor identificação e a pessoa fundamental com maior identificação.

Análise Teórica

A complexidade estará intrinsecamente ligada á complexidade da função *main* do programa. Relembrar que: *V* é o número de vértices e *E* o número de arcos entre estes.

A função que cria a lista de adjacências tem complexidade O(1). Depois recorremos a um ciclo *for* para criar os vertices que vamos lendo do input e as respetivas ligações. Dentro deste ciclo temos duas chamadas à função criaVertice(), que tem complexidade O(1), duas chamadas à função addVertice(), com complexidade O(1), e uma chamada à função criaLigação() também com complexidade O(1). Assim dentro do loop *for* a complexidade é de O(1) e como o loop é realizado E vezes, a complexidade total do loop *for* é O(E), apenas o número de vezes que o loop é executado.

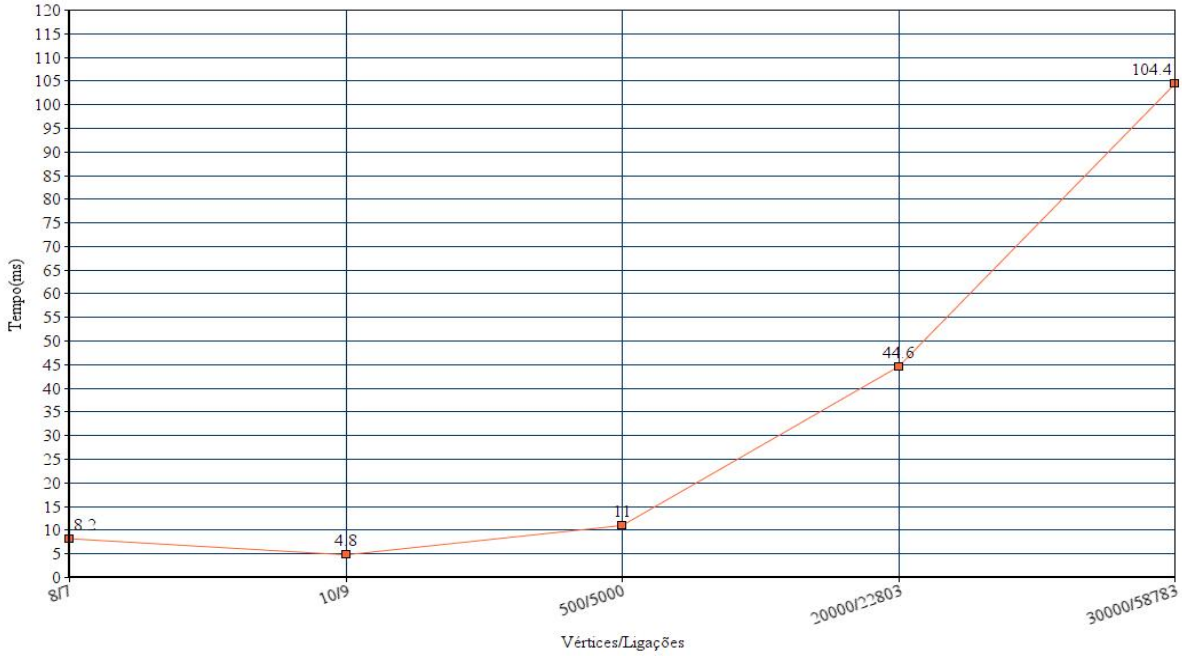
De seguida é executada a função verticesFundamentais que executa o algoritmo de Hopcroft-Tarjan sobre uma lista de adjacências e portanto a sua complexidade é O(V+E).

Por fim é apresentado o output do programa que corre em O(V).

Assim o nosso projeto tem complexidade final de O(1 + E + V + E +V) que é igual a O(V + E).

Análise Experimental

Para esta avaliação, foram utilizados os testes disponibilizados pelos responsáveis da disciplina para testar o desempenho do programa em



diferentes situações. Cada teste foi realizado 5 vezes, sendo que os valores presentes no gráfico abaixo representam as médias desses testes. Os tempos apresentados correspondem à componente *real* dos tempos obtidos.

Assim, verifica-se que o tempo de execução do programa aumenta quanto mais vértices e ligações entre eles houver como era esperado pela complexidade O(V+E).

Referências

https://en.wikipedia.org/wiki/Biconnected_component

http://www.eecs.wsu.edu/~holder/courses/CptS223/spr08/slides/graphapps.pdf

https://wiki.algo.informatik.tu-darmstadt.de/Hopcroft-Tarjan