Pontifícia Universidade Católica de Minas Gerais
Departamento de Ciência da Computação

Assignment 2 - Pontes

João Augusto dos Santos Silva
Thiago Ribeiro de Campos Nolasco

Prof. Silvio Jamil Ferzoli Guimarães
Teoria dos Grafos e Contabilidade

Belo Horizonte, MG, outubro de 2022

Conteúdo

1 Introdução

Este documento está associado ao primeiro exercício prático da matéria de Teoria dos Grafos e Computabilidade (TGC). Esse exercício exigiu a solução de problemas recorrentes no mundo de grafos, nos pedindo para implementar algoritmos reconhecidos, como Naive, Tarjan, Fleury e Brute Force. Esses algoritmos são utilizados para identificar a conectividade nos grafos, identificar componentes fortemente conexos, identificar caminhos eulerianos e identificar pontes, respectivamente.

1.1 Motivação

Colocar em prática a teoria aprendida em sala de aula, gerando mais familiaridade e entendimento sobre o assunto, incentivando um maior estudo prático da matéria a partir da implementação dos diferentes códigos abordados em sala de aula.

1.2 Objetivos

Identificar pontes através dos algoritmos de Tarjan e Naive, e através do uso desses dois algoritmos, somos capazes de executar o algoritmo de Fleury para identificar os caminhos eulerianos no grafo.

1.3 Grafo Utilizado

Utilizamos o grafo ilustrado na figura 1 para a realização dos nossos testes em código. Grafo conexo, não direcionado com 5 vértices e 5 vértices, o suficiente para testar tudo o que foi solicitado no trabalho.

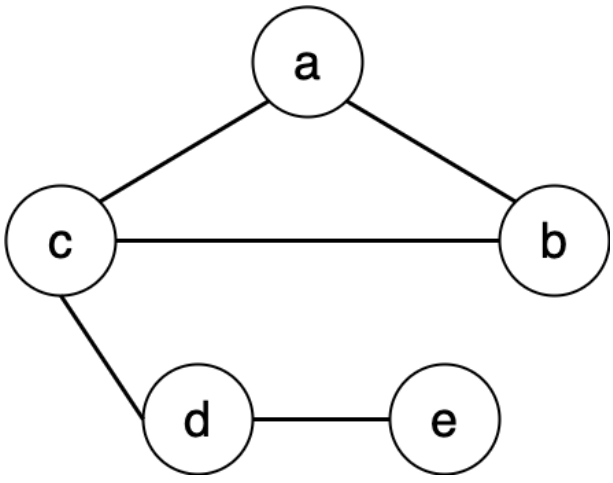

Figura 1: Grafo utilizado para os testes no código.

2 Algoritmos Implementados

Nessa seção do texto, aprofundaremos nos algoritmos implementados nesse trabalho.

2.1 Tarjan

O algoritmo de *Tarjan* foi implementado primeiramente para a identificação de *Strongly Connected Component* (SCC), em portugês Componentes Fortemente Conexos em grafos direcionados.

O algoritmo de *Tarjan* foi utilizado apenas como uma base para uma adaptação feita na finalidade do algoritmo, que assim como falado acima, foi criado para a identificação de componentes fortemente conexos nos grafos direcionados, mas como estamos trabalhando com grafos não-direcionados, não faz sentido buscarmos essas compontentes, tendo em vista que elas não existem para esse tipo de grafo. O algoritmo de *Tarjan* implementado no presente trabalho tem como objetivo identificar possíveis pontes no grafo. Para essa implementação, seguimos a ideia do *Tarjan* utilizando os *ID's* e *Low Link Values* para cada vértice do grafo, e através de uma *Deep First Search* (DFS), também conhecida como Busca em Profundidade a fim de modificarmos os valores nas listas de *ID's* e *Low Link Values* de cada vértice. Após definirmos os valores nas listas, fazemos as devidas verificações para identificarmos as arestas de ponte. Caso o *Low Link Values* do atual for maior que o *ID* do vértice sucessor, significa que estão em componentes distintos, logo caso a aresta que liga os dois vértices for removida do grafo, teremos dois componentes, logo a aresta é uma ponte.

2.1.1 Pseudocódigo

Algorithm 1 Algoritmo Tarjan

```
    Entrada: vértice Atual e vértice Anterior
 2: Saída: Uma lista contendo os vértices de ponte encontrados.
    G = GRAFO
 4: stack.push(atual)
    onStack[atual] = true
 6: ids[atual] = low[atual] = id++
    for sucessor in sucessores[atual] do
 8:     if ids[sucessor] == -1 then
            TARJAN(sucessor, atual)
10:    end if
       if sucessor != anterior  onStack[sucessor] == TRUE then
12:        low[atual] = min(low[atual], low(sucessor))
       end if
14:    if low[sucessor] > ids[atual] then
           resposta.append(concat(atual, sucessor))
16:    end if
    end for
18: if ids[atual] == low[atual] then
       for vert in stack do
20:        onStack[vert] = false
           stack.pop()
22:        low[vert] = ids[vert]
           if vert == atual then
24:            BREAK
           end if
26:    end for
    end if
28: Retorna resposta; FIM
```

2.2 Naive

O algoritmo *Naive* foi implementado para identificar, de maneira direta, arestas que são pontes em um Grafo.

O algoritmo *Naive*, também chamado de *Brute Force*, Força bruta em português, tem como ideia principal analisar um vértice do Grafo por vez, e no final, identificar todos as pontes existentes. Inicialmente é selecioado uma aresta qualquer, remove-se ela do conjunto de arestas, e executa um caminhamento de profundidade no Grafo com o novo conjunto de arestas. Ao final do caminhamento é analisado a quantidade total de vértices visitados no caminhamento, caso o número de vértices visitados seja menor do que a quantidade de vértices no Grafo, podemos afirmar que a aresta analisada é uma ponte.

2.2.1 Pseudocódigo

Algorithm 2 Algoritmo Naive

```
    Saída: Uma lista contendo os vértices de ponte encontrados.
 2: G = GRAFO
    visitado[aresta] = nova lista
 4: resposta = nova lista
    for aresta in Arestas do
 6:     if visitado[aresta] == false then
            visitado[aresta] = true
 8:         G.removeAresta(aresta)
            resultado = G.DFS()
10:         if resultado.size != G.vert.size() then
                resposta.append(aresta)
12:         end if
            G.adicionaAresta(aresta)
14:     end if
    end for
16: Retorna resposta; FIM
```

2.3 Fleury

O algoritmo de *Fleury* foi implementado com duas maneiras distintas de identificação de pontes para a definição do vértice inicial do nosso caminho. Primeiramente usamos o algoritmo de *Tarjan* modificado, que foi explicado anteriormente, e no segundo momento utilizamos o algoritmo *Naive* para a identificação dessas pontes.

O algoritmo de *Fleury* tem como papel principal a identificação de caminhos ou ciclos eulerianos em um Grafo, caso existam. A ideia central é identificar a quantidade de vértices com grau ímpar o Grafo possui. Caso seja 0, teremos um ciclo euleriano, e caso seja 2, um caminho euleriano. Com um ciclo euleriano, poderemos começar o caminhamento por qualquer vértice desejado, já no caminho euleriano, teremos que começar por um vértice de grau ímpar.

Após decidirmos qual vértice começar, deve-se identificar as pontes momentâneas, caso o vértice escolhido só tenha sucessores em arestas pontes, caminharemos para um sucessor qualquer, caso tenha arestas não pontes, devemos obrigatoriamente caminhar para ela.

2.3.1 Pseudocódigo

Algorithm 3 Algoritmo Fleury

Saída: Uma lista contendo os vértice do caminho ou ciclo euleriano do Grafo.
G = **GRAFO**
visitado[$aresta$] = **nova lista**
resposta = **nova lista**
bucket = **nova pilha**
vInicial = **novo vertice**
if Numero de vértices com grau ímpar == 0 **then**
    vInicial = vértice qualquer em G
else
    vInicial = vértice de grau ímpar qualquer em G
end if
bucket.$push(vInicial)$
while !bucket.$empty()$ **do**
    encontrado = *false*
    vRemovido = bucket.pop()
    removedSuc = sucessores de vRemovido
    pontes = $ALGORITMO(TARJAN/NAIVE)$
    while !removedSuc.$empty()$ **do**
        vVerificado = removedSuc.pop()
        if vVerificado não contido em **pontes then**
            vCaminhado = vVerificado
            removedSuc.$clear$()
            encontrado = *true*
        end if
    end while
    if encontrado != **TRUE then**
        vCaminhado = vVerificado
    end if
    arestaRemover = $concat(vRemovido, vCaminhado)$
    G.$removeAresta(arestaRemover)$
    bucket.$push(vCaminhado)$
    resposta.$push(vCaminhado)$
end while
Retorna resposta; FIM

Nº de Vértices					
Algoritmo	**5**	**100**	**1.000**	10.000	**100.000**
Fleury Tarjan	0,000834	0,01668	0,1668	1,668	16,68
Fleury Naive	0,001638	0,03276	0,3276	3,276	32,76
main.cpp	0,004103	0,08206	0,8206	8,206	82,06

Tabela 1: Tempo de execução em segundos.

3 Conclusão

Identificar todos as pontes e caminhos Eulerianos em um Grafo é uma tarefa árdua que porém pode ter muitas utilidades na computação. No trabalho em questão utilizamos duas abordagens para identificação de pontes (*Naive* e *Tarjan*) diferentes para verificar se a conectividade no Grafo se altera com a remoção de uma aresta e a outra abordagem utilizada no trabalho foi para a identificação de caminhos ou ciclos Eulerianos com o algoritmo de *Fleury* em duas versões distintas, na primeira, investigamos se uma aresta é ponte ou não através do algoritmo de *Tarjan* e na segunda abordagem, utilizamos o algoritmo de *Naive* para a identificação das mesmas pontes, e os resultados encontrados não foram tão surpreendentes. Como já esperado, o resultado final serão os mesmos, ambos os algoritmos encontrarão o mesmo resultado, o que os diferenciam é o custo computacional para a execução da tarefa.

Calculando os tempos de execução de cada algoritmo, notamos que a implementação do *Fleury* utilizando o algoritmo de *Tarjan* para a identificação das pontes possui o menor tempo de execução, gastando quase a metade do tempo que o algoritmo de *Fleury* utilizando o *Tarjan* para identificação de pontes demorou. As comparações estão claras na tabela 1, que preenchemos coletando o tempo de execução em uma máquina física utilizando o grafo ilustrado na figura 1, um grafo conexo de 5 vértices e a partir do tempo gasto por ele, fizemos a regra de três para encontrar o tempo de execução previsto para grafos de 100, 1.000, 10.000 e 100.000 vértices.

Nas figuras 2 e 3 está ilustrado o tempo de execução dos algoritmos desenvolvidos para a tarefa e com os seus respectivos resultados obtidos na máquina física.

Toda a documentação e código fonte utilizado no trabalho está disponível em um Repositório no GitHub.

```
Time taken by Fleury Naive: 0.001638 sec
Fleury length: 6
Fleury path: cabcde
```

Figura 2: Tempo de execução Fleury - Naive

```
Time taken by Fleury Tarjan: 0.000834 sec
Fleury length: 6
Fleury path: cabcde
```

Figura 3: Tempo de execução Fleury - Tarjan

Referências

Fiset, W. (2012), 'Tarjan's strongly connected components (scc) algorithm'.
URL: *https://www.youtube.com/watch?v=hKhLj7bfDKk*

Tarjan, R. E. (1974), 'A note on finding the bridges of a graph.'.

West, D. B. (2000), *Introduction to Graph Theory*, 2 edn, Prentice Hall.