GCC218 - Trabalho 1

***Nome**: Filipe Barros R.* ***Matrícula**: 201611121* ***Turma**:14A*

***Nome**: José M. Rivas* ***Matrícula**: 201520617* ***Turma**: 14A*

***Nome**: Raydson Ferreira C.* ***Matrícula**: 201611137* ***Turma**: 14A*

PROF. MAYRON CÉSAR DE O. MOREIRA
2017/1

Universidade Federal de Lavras

Departamento de Ciência da Computação

ALGORITMOS IMPLEMENTADOS

Para detecção de pontes respondendo a questão 1, (P1 Se for conexo, qual aresta que se retirada o torna desconexo? Existe apenas uma ou mais arestas com essa peculiaridade?) foi utilizado uma adaptação na busca em profundidade, contendo dois métodos para tal, um chamado '*verificaPonte*' que recebe um grafo, e a quantidade de vértices que ele possui, dentro deste método é usado um **for** que pega em cada interação um vértice, e é feita uma comparação se o vértice pego já foi visitado, se não ele chama o método *'findBridges'* que executa a busca pela quantidade de pontes, e por fim o método '*verificaPonte*' retorna a quantidade de pontes.
No método *'findBridges'* é recebido um grafo, um vértice inicial, a quantidade de vértices, um contador, a quantidade de pontes já encontradas, e os vértices com marcações. Dentro desse método é feito uma busca em profundidade a partir de um vértice, se os adjacentes desse vértice já foram visitados é feita uma comparação se o menor tempo de alcance em pré-ordem (low) do vértice é maior que o tempo de descoberta de seu adjacente, se sim o valor de pré ordem do vértice é atualizado para o tempo de descoberta de seu adjacente. Se o adjacente não foi visitado, é chamado uma recursão com esse vértice, ao retornar da recursão, é verificado se o menor tempo de alcance em pré-ordem (low) do vértice é maior que de seu adjacente, caso seja o valor é atualizado para o de pré ordem do seu adjacente, depois é feita uma comparação, se o tempo de pré-ordem do adjacente é igual ao tempo de descoberta dele, a aresta que liga os dois é uma ponte.

Algoritmo 1: Detecção de pontes:

```
1:procedure verificaPonte(G, tam)   // G = (V, E), tam = |V|, V = vértices, E = arestas
2:      cnt = bcnt = 0                  // cnt = contador, bcnt = quantidade de pontes
3:      for i=0 to tam do
4:          if V(i).visitado == false then
5:              findBridges(G, i, tam, cnt, bcnt, V)
6:      return bcnt
7:
8:procedure findBridges(G, v, tam, cnt, bcnt, V) // G = (V, E), V = vértices,
                  // E = arestas, v ∈ V, tam = |V|, cnt = contador, bcnt = número de pontes
9:      v.visitado = true
10:     v.low = v.num = cnt++   // low = alcance em pré-ordem, num = tempo de visita
11:         for i = adj(v).[0] to adj(v).[n] do      // n = número de adjacentes de v
12:         if V(i).visitado == false then
13:             V(i).anterior = v
14:             findBridges(G, i, tam, cnt, bcnt, V)
15:             if V(v).low > V(i).low then
16:                 V(v).low = V(i).low
17:             if V(i).low == V(i).num then
```

```
18:                                 bcnt ++          // Achou uma ponte
19:             else if i ≠ V(v).anterior and V(v).low > V(i).num then  // Se i não for o pai de v e
                                                     // se o menor tempo de pré ordem do vértice é
                                                     // maior que o tempo de descoberta do adjacente
20:                         V(v).low = V(i).num
```

Para a detecção de pontos de articulação, respondendo a questão 2, (P2 Existe um único vértice que, se retirado, causaria uma desconexão no grafo?) também foi utilizado uma adaptação na busca em profundidade, contendo dois métodos para tal, um chamado *'verificaPontoDeArticulacao'* que recebe um grafo, e a quantidade de vértices que ele possui, dentro deste método é usado um **for** que pega em cada interação um vértice, e é feita uma comparação se o vértice pego já foi visitado, se não ele chama o método *'cutVertices'* que executa a busca pelos pontos de articulação, depois é usado um outro **for** que verifica qual vértice foi marcado, e se foi marcado é incrementado em contador, depois retorna esse contador com a quantidade de articulações.
No método *'cutVertices'* é recebido um grafo, um vértice inicial, a quantidade de vértices e os vértices com marcações. Dentro desse método é feito uma busca em profundidade a partir de um vértice, se os adjacentes desse vértice já foram visitados é feita uma comparação se o adjacente não é o pai do vértice atual, se ele não é, o menor tempo de alcance em pré-ordem (low) dele recebe o valor mínimo entre o próprio valor e o tempo de alcance do seu adjacente. Se o adjacente não foi visitado, é chamado uma recursão com esse vértice, ao retornar da recursão, é verificado se o vértice é raiz e tem mais de um filho, o vértice é marcado como ponto de articulação, e se o vértice não é raiz e o menor tempo de pré ordem (low) dos adjacentes é maior ou igual ao tempo de alcance do vértice atual, esse vértice é marcado como articulação.

Algoritmo 2: Detecção de pontos de articulação

```
 1:procedure verificaPontoDeArticulacao(G, tam)        // G = (V, E), tam = |V|, V = vértices,
                                                       // E = arestas
 2:         numArticulacoes = 0          //numArticulacoes = número de articulações
 3:         for i=0 to tam do
 4:                 if V(i).visitado == false then
 5:                         cutVertices(G, i, tam, V)
 6:         for j=0 to tam do
 7:                 if V(j).marcado then
 8:                         numArticulacoes++
 9:         return numArticulacoes
10:
11:procedure cutVertices(G, u, tam, V)      // G = (V, E), V = vértices, E = arestas, u ∈ V, tam =
|V|
12:                 static cnt = 0
13:         filhos = 0
14:         V(u).visitado = true
```

```
15:     V(u).low = V(u).num = ++cnt       // low = alcance em pré-ordem, num = tempo de visita
15:     for v = adj(u).[0] to adj(u).[n] do       // n = número de adjacentes de u
16:         if V(v).visitado == false then
17:             filhos++
18:             V(v).anterior = u
19:             cutVertices(G, v, ta, V)
20:             V(u).low = min(V(u).low, V(v).low)
21:             if V(u).anterior == -1 and filhos > 1 then // vértice é raiz e tem mais de
                                                          //um filho
22:                 V(u).marcado = true
23:             if V(u).anterior ≠ -1 and V(v).low ≥ V(u).num then // vértice não é raiz e
                                                  // o menor tempo de pré-ordem dele é maior ou
                                                  // igual ao tempo de visitação do vértice atual
24:                 V(u).marcado = true
25:         else if v ≠ V(u).anterior then  // se o adjacente não é pai do vértice atual
26:             V(u).low = min(V(u).low, V(v).num)
```

Para detecção de vértices alcançáveis a partir de um vértice qualquer, respondendo a questão 3, (P3 Partindo de um vértice qualquer, quantos outros vértices podemos alcançar no grafo?) foi utilizado, também, uma adaptação da busca em profundidade, contendo dois métodos para tal, um chamado '*verticesAlcancaveis'* que recebe um grafo, um vértice onde será o ínicio da busca e a quantidade de vértices do grafo, dentro desse método é utilizado um contador que guarda a quantidade de vértices alcançáveis, e é chamado o método *'buscarVertices',* depois retorna a quantidade de vértices alcançáveis.

No método *'buscarVertices'* é recebido um grafo, um vértice de começo, uma marcação dos vértices,e a quantidade de vértices, Dentro desse método é feito uma busca em profundidade a partir de um vértice, e sempre que um adjacente do vértice atual é visitado pela primeira vez, o contador 'cnt' é incrementado, depois esse contador é retornado representando quantos vértices foram alcançados.

Algoritmo 3: Detecção de vértices alcançáveis

```
1:procedure verticesAlcancaveis(G, v, tam) // G = (V, E), V = vértices, E = arestas, v ∈ V,
                                           // tam = |V|
2:      cnt = -1                          //contador pra quantidade de vertices que foram alcançados
3:      alcancaveis = buscarVertices(G, v, V, cnt)
4:      return alcancaveis
5:
6:procedure buscarVertices(G, u, V, cnt) // G = (V, E), V = vértices, E = arestas, u ∈ V,
                                         // cnt = contador
7:      V(u).visitado = true
8:      cnt ++
```

```
 9:        for v = adj(u).[0] to adj(u).[n] do       // n = número de adjacentes de u
10:             if V(v).visitado == false then
11:                       buscarVertices(G, v, V, cnt)
12:       return cnt
```

Para calcular o diâmetro do grafo respondendo a questão 4, (P4 Qual é o diâmetro (maior entre as menores distâncias) do grafo?) foi utilizado uma adaptação no algoritmo de busca em largura, com um método chamado '*calcularDiametro'* onde é passado um grafo, e o tamanho do grafo. Dentro deste método, é criado uma fila, que auxilia na busca em largura, guardando o valor do vértice sempre que ele é visitado, assim é executado um **for** com a quantidade de vértices, é marcado então o primeiro vértice e é colocado na fila, dentro ainda do **for,** é executado um **while** até que a fila esteja vazia, dentro do **while** é retirado o primeiro elemento da fila e guardado em uma variável 'v', depois é pego os vizinhos de 'v', se eles não foram visitados é marcado como visitado, incrementa a distância, e coloca na fila. Depois com cada vizinho de 'v' se a distância do vizinho for maior que a maior distância encontrada nos adjacentes, o valor da maior distância é atualizado. Ao fim do **while** é feita uma comparação se a maior distância encontrada com os vizinhos de v é maior que a distância total, se sim o valor da distância total é atualizado, e é retornado a distância total com a maior distância encontrada.

Algoritmo 4: Cálculo do diametro

```
 1:procedure calcularDiametro(G, tam)     // G = (V, E), V = vértices, E = arestas, tam = |V|
 2:        distancia = maiorDistancia = 0
 3:        for i =0 to tam do
 4:             V(i).d = 0                  // atualiza a distância do primeiro vértice.
 5:             V(i).cor = 'C'              // marca cinza como visitação.
 6:             Enqueue (Fila, i)        // coloca i dentro da fila.
 7:             while Fila ≠ Ø do
 8:                  Dequeue (Fila, v)
 9:                  for u = adj(v).[0] to adj(v).[n] do       // n = número de adjacentes de v.
10:                       if V(u).cor ≠ 'C' then
11:                                 V(u).d = V(v).d + 1  //incrementa a distância do
vértice u.
12:                                 V(u).cor = 'C'         //marca como visitado.
13:                                 Enqueue (Fila, u)      // é colocado u na fila.
14:                       if V(u).d > distancia then     // se a distância é menor.
15:                                 distancia = V(u).d     // distância atualizada.
16:             if distancia > maiorDistancia then
17:                  maiorDistancia = distancia
18:       return maiorDistancia
```

Para encontrar o grafo simplificado para responder a questão 6, (Se o grafo for fortemente conexo (grafos direcionados), qual seria seu grafo simplificado, agrupando as componentes fortemente conexas?) foi utilizado uma adaptação do algoritmo de busca em profundidade, contendo dois métodos para tal, um chamado *'componentesFortementeConexas'* que recebe um grafo, uma variável com a quantidade de componentes conexas e uma variável com o número de arestas que ligam os novos vértices do grafo simplificado. Dentro desse método é chamado um **for** com a quantidade de vértices do grafo, se o vértice ainda não foi visitado, é chamado o método *'checarConectividade',* passando o grafo, o adjacente, uma identificação (label), os marcadores do vértice, e uma pilha. Ao retornar da função *'checarConectividade'*, é usado um **for** com a quantidade de vértices do grafo, se o vértice dado pelo **for** não foi visitado, ele é marcado como visitado.
Após percorrer os dois **for**'s fazendo as comparações já se tem o número de vértices conexos, para obter o número das arestas do novo grafo (Obs.: o novo grafo considerará todas as arestas que forem encontradas no mesmo sentido que liga um componente 'a' em outro 'b', e nunca de 'b' para 'a', sendo assim poderão existir muitas arestas que ligam os mesmos componentes, só que no mesmo sentido.) depois retorna a pilha.
Já o método *'checarConectividade'* ele recebe um grafo, um vértice, uma identificação (label), os vértices com as marcações, e uma pilha, dentro do método ele pega a lista de vizinhos do vértice recebido no parâmetro e faz um **for** para cada vizinho, se o vizinho não foi visitado, é chamado ele na recursão, ao retornar da recursão é feita umas comparações e alteração de valores do vértice atual e de seus adjacentes, ao sair do **for** ele faz uma comparação se o menor tempo de pré-ordem é igual ao tempo de alcance, se sim ele atualiza a identificação do vértice, agrupando todos que fazem parte do componente fortemente conexo.

Algoritmo 6: Cálculo da maior distância :

```
1:procedure componentesFortementeConexas(G, numVerticesConexos, numArestasConexas)//G =
          //(V, E), V = vértices, E = arestas, numVerticesConexos = número de vertices do novo
          //grafo, numArestasConexas = número de arestas do novo grafo
 2:       label = 0              // label = identificação de vértice
 3:       for i=0 to tam do      // tam = número de vértices
 4:            if V(i).visitado == false then
 5:                 checarConectividade(G, i, label, V, pilha)
 6:            for j = 0 to tam do
 7:                 if V(j).visitado
 8:                      V(j).marcado = true
 9:       numVerticesConexos = label
10:       numArestasConexas = 0
11:            for i=0 to tam do
12:            for j=0 to |adj(i)| do
```

```
13:                     if V(i).label ≠ V(j).label
14:                             numArestasConexas++
15:
16: procedure checarConectividade(G, u, label, V, pilha) // G = (V, E), V = vértices, E = arestas,
                                //u ∈ V, label = identificação de vértice, pilha = Stack
```

RESULTADOS

Tabela 1: Resultados computacionais.

Instância	P1	P2	P4	P5	P6	t (s)
ego-Facebook	053ms584µs	052ms632µs	03m03s596ms265µs	14m19s357ms290µs	054ms324µs	19m914ms095µs

***Pergunta6**: Número de vértices do novo Grafo: 1 Número de arestas do novo Grafo: 0*

Tabela 2: Resultados computacionais.

Instância	P1	P2	P4	P5	P6	t (s)
Paris.txt	0ms055µs	0ms052µs	4ms090µs	10ms751µs	0ms095µs	15ms43µs

***Pergunta6**: Número de vértices do novo Grafo: 1 Número de arestas do novo Grafo: 0*

Tabela 3: Resultados computacionais.

Instância	P1	P2	P4	P5	P6	t (s)
London.txt	0ms046µs	0ms042µs	2ms765µs	010ms745µs	0ms099µs	13ms697µs

***Pergunta6**: Número de vértices do novo Grafo: 1 Número de arestas do novo Grafo: 0*

Tabela 4: Resultados computacionais.

Instância	P1	P2	P4	P5	P6	t (s)
Instance14.txt	-	-	13m01s898ms137µs		1s486ms142µs	

***Pergunta6**: Número de vértices do novo Grafo:* **3** *Número de arestas do novo Grafo:* **582**

INTERPRETAÇÃO DOS RESULTADOS

Para encontrarmos uma ponte (Pergunta 1) na instância transporte, devemos encontrar uma aresta onde todos os veículos, precisam passar. Na instância de Rede Social, essa ponte é estabelecida entre dois usuários que não possuem outro caminho onde possam se conectar.

Para encontrarmos um ponto de articulação (Pergunta 2) na instância transporte, devemos encontrar uma cidade onde todos os veículos precisam passar nesse ponto. Na instância de Rede Social, esse ponto de articulação é determinado por um usuário que possui grande influência na rede ou um grande número de conexões.
Para encontrarmos o diâmetro (Pergunta 4) na instância transporte, devemos encontrar o caminho mais longo que há em uma rede, considerando-se todos os pontos. Na instância de Rede Social, este caminho seria a maior combinação de usuários interligados.

Para encontrarmos as componentes fortemente conexas (Pergunta 5) na instância transporte, devemos identificar quais os caminhos possíveis no grafo, já que muitos destes caminhos, não são conectados, são de um único sentido ou até mesmo inalcançáveis.

Para encontrarmos o grafo simplificado (Pergunta 6) na instância transporte, deve-se agrupar as componentes que possuem algum tipo de similaridade. Na instância de Rede Social, a agrupação dos componentes, pode ser feita por análise dos interesses em comum entre os usuários.