Universidade Federal de Santa Catarina
Ciências da Computação

Alunos: Gabriel Dutra e Lucas Silva da Costa

Matriculas: 18205598 e 18200432

Professor: Rafael de Santiago

Turma: INE5413-04208 (20212)

Relatório da Atividade 2

Considerações Gerais:

A linguagem escolhida para resolução dos problemas foi Java, A estrutura utilizada para ser o grafo guardado e analisado, fora um objeto, sendo que seus atributos representam listas de vértices, arestas(multiArray), e pesos.

Métodos Gerais:

Alguns métodos foram utilizados da atividade 1, porém se necessário adaptados, como por exemplo "vizinhos", no qual trazia todos os vizinhos em um grafo não direcionado, porém, comentando a "volta" do vizinho, traz apenas os vizinhos no qual o elemento analisado pode chegar.

```
// Verifica a quantidade de vertices por meio do tamanho do array V
protected int qtdVertices() {
	return V.length;
}

// Verifica a quantidade de Arestas por meio do tamanho do array E
protected int qtdArestas() {
	return E.length;
}

// Verifica nas arestas quais ligações o vertice possue, e retorna o seus vizinhos
protected List<Integer> vizinhos(int v) {
	// Lista para armazenar os vizinhos
   List<Integer> viz = new ArrayList<Integer>();
   // Percorre todas as arestas
   for (int i = 0; i < E.length; i++) {
	// Verifica se a aresta possue o vertice do parâmetro no primeiro elemento
	if (E[i][0] == v) {
		// Adiciona o segundo elemento como vizinho
		viz.add(E[i][1]);
		}
	// Verifica se a aresta possue o vertice do parâmetro no segundo elemento
		//if(E[i][1] == v ) {
			// Adiciona o primeiro elemento como vizinho
			//viz.add(E[i][0]);
		//}
	}
   // Retorna lista de vizinhos
	return viz;
}
```

Exercício 1: Componentes Fortemente Conexos

Após fazer a leitura do grafo, e o armazenar em um multiArray, no qual é utilizado para qualquer método de manipulação e exposição do mesmo. É chamado o método Forte_Conexo, no qual coloca os elementos do grafo em uma lista, realizando de uma forma similar ao "Algoritmo 15" das "Anotações da Disciplina", porém faz o print no próprio método.

```
protected int[] Forte_Conexo(Grafo_D_NP grafo) {

        //Criando CTFA, onde armazeno Cv,Tv,Fv,Av
        int[][] CTFA = new int[grafo.qtdVertices()][4];
        //Criando CTFA, onde armazeno Cvt,Tvt,Fvt,Avt (os criados a partir do grafo transposto)
        int[][] CTFAt = new int[grafo.qtdVertices()][4];

        //Chamo DFS, para preencher o CTFA
        CTFA = DFS(grafo);
        //Crio o grafo que sera o transposto
        Grafo_D_NP grafoT = grafo;
        //Chamo o metodo que retorna o grafo transposto e armazeno no grafoT
        grafoT = grafo_transp(grafoT);

        //Crio Cvt,Tvt,Fvt,Avt
        boolean [] Cvt = new boolean [grafo.qtdVertices()];
        int[] Tvt = new int [grafo.qtdVertices()];
        int[] Fvt = new int [grafo.qtdVertices()];
        int[] Avt = new int [grafo.qtdVertices()];

        //Chamo o DFS adaptado para que o loop funcione a partir do tempo decorrido no Fv
        CTFAt = DFS_adap(grafoT,CTFA);

        //loop que armazena o Avt
        for (int i = 0; i < grafo.qtdVertices(); i++) {
                Avt[i] = CTFAt[i][3];
        }

        //Lista utilizada para armazenar os elementos fortemente conexos
        List<Integer> final2 = new ArrayList<Integer>();

        //Loop para colocar aos pares ligados dos elementos fortemente conexos
        for (int i = 0; i < grafo.qtdVertices(); i++) {
                if(Avt[i] != 0) {
                        final2.add(i+1);
                        final2.add(Avt[i]);
                }
        }
```

```
		//Método usado para remover os elementos repetidos da lista
		final2 = removeDuplicatas((ArrayList<Integer>) final2);

		//Lista utilizada para print
		int[][] final3 = new int[grafo.qtdVertices()][2];

		//Caso tenha algum elemento fortemente conexo
		if(final2.size() > 0) {
			//Adiciono final2 e Avt para a lista de print, no qual consigo fazer a separacao
dos conjuntos
			for (int i = 0; i < grafo.qtdVertices(); i++) {
				final3[i][0] = final2.get(i);
				final3[i][1] = Avt[i];
			}
			//for e ifs para organizar o print
			for (int i = 0; i < final3.length-1; i++) {
				if(final3[i][1] != 0 && final3[i+1][1] != 0) {
					System.out.print(final3[i][0]+",");
				}
				if(final3[i][1] != 0 && final3[i+1][1] == 0) {
					System.out.print(final3[i][0]);
				}
				if(final3[i][1] == 0 && i != 0) {
					System.out.print("] ["+final3[i][0]+",");
				}
				if(i == 0) {
					System.out.print("[");
				}
			}
			System.out.print(final3[final3.length-1][0]+"]");
		//Quando não possui nenhum elemento fortemente conexo com ligacao a outro
		}else {
			//Print todos os elementos separadamente
			for (int i = 0; i < grafo.qtdVertices(); i++) {
				System.out.print("["+(i+1)+"]");
			}
		}
		//retorno a lista Avt
		return Avt;
	}
```

O primeiro método a ser chamado a partir do "Forte_Conexo", é o "DFS", no qual retorna os tempos de inicio, fim, antecessor e se o elemento foi visitado anteriormente, por meio de um multiArray.

```
protected int[][] DFS(Grafo_D_NP grafo) {

	//Criar Cv, Tv, Fv, Av
	int [] Cv = new int [grafo.qtdVertices()];
	int[] Tv = new int [grafo.qtdVertices()];
	int[] Fv = new int [grafo.qtdVertices()];
	int[] Av = new int [grafo.qtdVertices()];
	//Inicializa Cv,Tv,Fv,Av
	for (int i = 0; i < grafo.qtdVertices(); i++) {
		Cv[i] = 0;
		Tv[i] = (int) Double.POSITIVE_INFINITY;
		Fv[i] = (int) Double.POSITIVE_INFINITY;
		Av[i] = 0;
	}
	//Inicializa variavel tempo, utilizada em Tv e Fv
	int tempo = 0;

	//loop para fazer as atribuicoes iniciais
	for (int i = 0; i < grafo.qtdVertices(); i++) {
		//Testa se o elemento observado foi visitado anteriormente
		if (Cv[i] == 0){
			//Chama o metodo DFS_Visit, onde fara as atribuicoes
			tempo = DFS_Visit(grafo,i,Cv,Tv,Fv,Av, tempo);
		}
	}

	//Cria CTFA, para armazenar os dados obtidos referente ao metodo anterior
	int[][] CTFA = new int[grafo.qtdVertices()][4];
	//Armazena os dados
	for (int i = 0; i < grafo.qtdVertices(); i++) {
		CTFA[i][0] = Cv[i];
		CTFA[i][1] = Tv[i];
		CTFA[i][2] = Fv[i];
		CTFA[i][3] = Av[i];
	}
	//Retorna a matriz com Cv,Tv,Fv,Av
	return CTFA;
}
```

A partir do método “DFS”, chegamos em DFS_Visit, no qual é chamado quando o elemento não foi visitado, o setando para “true”(no caso 1), incrementando seu tempo, e chamando por meio recursivo o método com os elementos vizinhos ao analisado.

```
protected int DFS_Visit(Grafo_D_NP grafo,int v,int [] Cv,int[] Tv,int[] Fv,int[] Av, int tempo)
{
	//Atribui o elemento a marcacao de visitado
	Cv[v] = 1;
	//Aumenta o tempo
```

```
        tempo++;
        //Insere o tempo atual em Tv(Tempo de inicio)
        Tv[v] = tempo;
        //loop que observa os visinhos do elemento vizitado
        for (int u : grafo.vizinhos(v+1)) {
            //Testa se o elemento observado foi visitado anteriormente
            if (Cv[u-1] == 0) {
                //Insere o valor V para o Antecessor
                Av[u-1] = v+1;
                //Utiliza recursao para continuar a busca no grafo
                tempo = DFS_Visit(grafo,u-1,Cv,Tv,Fv,Av, tempo);
            }
        }
        //Aumenta o tempo
        tempo++;
        //Armazena o tempo atual em Fv(tempo de fim)
        Fv[v] = tempo;
        //Retorna o tempo
        return tempo;
    }
```

Esse método tem a função de retornar um grafo transposto ao grafo do parâmetro, logo, o sentido das arestas mudam.

```
    protected Grafo_D_NP grafo_transp(Grafo_D_NP grafoT) {
        //Loop que modifica o sentido das arestas
        for (int i = 0; i < E.length; i++) {
            int temporario = grafoT.E[i][0];
            grafoT.E[i][0] = grafoT.E[i][1];
            grafoT.E[i][1] = temporario;
        }
        //Retorna o grafico transposto
        return grafoT;
    }
```

Para o funcionamento do algoritmo, é necessário a utilização de um "DFS" alterado, sendo que ele percorre os elementos baseados no tempo de finalização de cada elemento analisado anteriormente.

```
    protected int[][] DFS_adap(Grafo_D_NP grafo, int[][] CTFA) {
        //Criar Cv, Tv, Fv, Av
        int [] Cv = new int [grafo.qtdVertices()];
        int[] Tv = new int [grafo.qtdVertices()];
        int[] Fv = new int [grafo.qtdVertices()];
        int[] Av = new int [grafo.qtdVertices()];

        //Inicializlo Cv,Tv,Fv,Av
        for (int i = 0; i < grafo.qtdVertices(); i++) {
```

```
                Cv[i] = 0;
                Tv[i] = (int) Double.POSITIVE_INFINITY;
                Fv[i] = (int) Double.POSITIVE_INFINITY;
                Av[i] = 0;
        }
        //Inicializo o tempo, utilizado para Tv e Fv
        int tempo = 0;
        //Array criado para auxiliar no loop de Fv
        int[] index = new int[grafo.qtdVertices()];

        //Variaveis auxiliares em relacao ao array index
        int count = 0;
        int largest = 0;
        //Loop para ordenar os indexes do maior para o menor(esse é o conteudo contido no
array index)
        for (int j = 0; j < grafo.qtdVertices(); j++) {
                for ( int i = 0; i < grafo.qtdVertices(); i++ ) {
                        //Comparação para armazenar o maior valor no array
                        if ( CTFA[i][2] > CTFA[largest][2]) {
                                largest = i;
                        }
                }
                //Armazenar index referente ao maior valor de Fv no index
                index[count] = largest;
                count++;
                //Retirar o valor de Fv, para que possa continuar fazendo as comparacoes
                CTFA[largest][2] = 0;
        }

        //loop adaptado funcionando a partir de Fv
        for (int i : index) {
                //Testa para ver se o elemento analizado foi visitado anteriormente
                if (Cv[i] == 0){
                        //Chama o metodo DFS_Visit, no qual fara as atribuicoes
                        DFS_Visit(grafo,i,Cv,Tv,Fv,Av, tempo);
                }
        }

        //Criacao de CTFAt, para armazenar todos os dados e os retorna-los posteriormente
        int[][] CTFAt = new int[grafo.qtdVertices()][4];
        //Armazenar os dados
        for (int i = 0; i < grafo.qtdVertices(); i++) {
                CTFAt[i][0] = Cv[i];
                CTFAt[i][1] = Tv[i];
                CTFAt[i][2] = Fv[i];
                CTFAt[i][3] = Av[i];
        }
        //Retorna todos os dados obtidos
```

```
        return CTFAt;
    }
```

Esse método tem como função, a remoção de elementos repetidos em uma lista, é utilizado para que o print do método "Forte_Conexo" ocorra corretamente.

```
    public static <Integer> ArrayList<Integer> removeDuplicatas(ArrayList<Integer> list) {

        //Cria lista Hash
        Set<Integer> set = new LinkedHashSet<>();
        //Adiciona os elementos
        set.addAll(list);
        //Limpa a lista
        list.clear();
        //Adiciona sem dupliucatas
        list.addAll(set);
        //Returna a lista
        return list;
      }
```

Exercício 2: Ordenação Topológica

Utilizado para contagem de vértices, na qual é utilizado para alocação de dados para arrays e listas, além de ser utilizado para loops.

```
    // Verifica a quantidade de vertices por meio do tamanho do array V
    protected int qtdVertices() {
        return V.length;
    }
```

Após fazer a leitura do grafo, e o armazenar em um multiArray, no qual é utilizado para qualquer método de manipulação e exposição do mesmo. É chamado o método OrdenaTopo, no qual coloca os elementos do grafo em uma lista, realizando de uma forma similar ao "Algoritmo 15" das "Anotações da Disciplina".

```
    protected List<Integer> OrdenaTopo(Grafo_D_NP grafo) {

        // Criando CV, Tv, Fv para poder armazenar os valores
        boolean [] Cv = new boolean [grafo.qtdVertices()];
        double[] Tv = new double [grafo.qtdVertices()];
        double[] Fv = new double [grafo.qtdVertices()];

        //Inicializando os valores de Cv, Tv, Fv
        for (int i = 0; i < grafo.qtdVertices(); i++) {
            Cv[i] = false;
```

```
                Tv[i] = Double.POSITIVE_INFINITY;
                Fv[i] = Double.POSITIVE_INFINITY;
        }
        //Inicializando a contagem de tempo
        int tempo = 0;

        //Criando lista para a Ordenacao
        List<Integer> Ordena = new ArrayList<Integer>();

        //Loop para a quantidade de vertices do grafo
        for (int i = 0; i < grafo.qtdVertices(); i++) {
                //Testa se o elemento ja foi visitado
                if (Cv[i] == false){
                        //Chama o método DFS_Visit_Ord, quando o elemento n foi visitado
                        DFS_Visit_Ord(grafo,i,Cv,Tv,Fv, tempo,Ordena);
                }
        }
        //Retorna a lista ordenada
        return Ordena;
}
```

O método visto, chama outro, sendo um DFS_Visit, porém alterado, de modo que adiciona o elemento a lista criada, sem necessitar retorno, pois a função principal dele é a própria ordenação.

```
public void DFS_Visit_Ord(Grafo_D_NP grafo,int v,boolean [] Cv,double[] Tv,double[] Fv,
int tempo,List Ordena) {
        //Defini que o elemento foi visitado
        Cv[v] = true;
        //Aumenta o tempo
        tempo++;
        //Define o tempo de inicio
        Tv[v] = tempo;
        //Loop que olha pelos vizinhos do elemento observado
        for (int u : grafo.vizinhos(v+1)) {
                //Testa se o elemento foi visitado
                if (Cv[u-1] == false) {
                        //Chama recursivamente o metodo, caso o elento vizinho nao tenha
sido visitado
                        DFS_Visit_Ord(grafo,u-1,Cv,Tv,Fv, tempo, Ordena);
                }
        }
        //Aumenta o tempo
        tempo++;
        //Define o tempo final
        Fv[v] = tempo;
```

```
            //Adiciona o elemento ao inicio do lista de ordenacao
            Ordena.add(0,v);
    }
```

Utiliza uma lista para retornar todos os vértices, os quais possuem ligação com o vértice analisado, porém com direção e sentido das ligações.

```
    // Verifica nas arestas quais ligações o vertice possue, e retorna o seus vizinhos
    protected List<Integer> vizinhos(int v) {
    // Lista para armazenar os vizinhos
            List<Integer> viz = new ArrayList<Integer>();
            // Percorre todas as arestas
            for (int i = 0; i < E.length; i++) {
                    // Verifica se a aresta possue o vertice do parâmetro no primeiro elemento
                    if (E[i][0] == v) {
                            // Adiciona o segundo elemento como vizinho
                            viz.add(E[i][1]);
                    }

            }
    // Retorna lista de vizinhos
    return viz;
    }
```

Utilizado para transformar os índices que são retornados pelo método OrdenaTopo, em seus rótulos para o Print dos dados

```
    // Retorna o conteudo do Array de Vertices(Rótulo)
    protected String rotulo(int index) {
            return V[index];
    }
```

Exercício 3: Árvore geradora mínima, algoritmo de Kruskal

Foi criado uma classe para o algoritmo, filha da classe de Grafos não-dirigidos e ponderados. Também foi definido como atributo a estrutura A, o retorno do algoritmo.

```
public class kruskal extends Grafo_ND_P {
    protected List<List<Integer>> A = new ArrayList<List<Integer>>();
```

Logo após a criação de um objeto da classe Kruskal o método executarKruskal() **é chamado. Dentre as principais execuções dele está a comparação de Su e Sv, tudo explicado nos comentários do código. Além disso foi utilizado listas de listas de integer por conveniência na utilização da abordagem de conjuntos.**

```
protected void executarKruskal() {
    // Lista de Listas de cada vértice
    List<List<Integer>> S = new ArrayList<List<Integer>>();
    // Alimentando S com seus proprios vértices
    for (int i=0; i < super.qtdVertices(); i++) {
        List<Integer> Si = new ArrayList<Integer>();
        Si.add(i+1);
        S.add(Si); // Adicionando os indexes dos vértices em ordem
    }
    // Lista de vértices em ordem crescente
    int [][] Elinha = super.arestasCrescentes();
    for (int i=0; i < super.qtdArestas(); i++) {
        // Definição de Su e Sv para facilitar a leitura do código
        List<Integer> Su = S.get(Elinha[i][0] -1);
        List<Integer> Sv = S.get(Elinha[i][1] -1);
        // se Su != Sv
        // Essa comparação foi feita utilizando teoria de conjunto
        // Eles são diferentes sse a intersecção entre eles for vazia
        // ou a diferença entre eles NÃO for vazia
        HashSet<Integer> intersec = new HashSet<>();
      intersec.addAll(Su);
      intersec.retainAll(Sv);
      HashSet<Integer> dif = new HashSet<>();
      dif.addAll(Su);
      dif.removeAll(intersec);
      if (intersec.isEmpty() || !(dif.isEmpty())) {
        //System.out.println("in\n\n");
        // A recebe {u, v}
        List<Integer> uv = new ArrayList<Integer>();
        uv.add(Elinha[i][0]);
        uv.add(Elinha[i][1]);
        // Adicionando {u, v} à A
        A.add(uv);
        // Definindo x como a união dos conjuntos Su e Sv
        List<Integer> x = new ArrayList<Integer>();
```

```
                        HashSet<Integer> union = new HashSet<>();
                        union.addAll(Su);
                        union.removeAll(intersec);
                        union.addAll(Sv);
                        x.addAll(union);
                        // Preenchendo os conjuntos dos vértices envolvidos
                        for (int y=0; y<x.size(); y++) {
                                List<Integer> Sy = completaSy(S.get(x.get(y)-1), x);
                                S.set(x.get(y)-1, Sy);
                        }
                    }
                }
                printKruskal();
        }
```

O método que ordena as arestas foi adicionado à classe Grafo_ND_P, e trata-se de um bubble sort simples. No array "result" as posições '0' e '1' são os índices dos vértices ligados pela aresta, enquanto a posição '2' trata-se do índice do peso daquela aresta no array "w".

```
        protected int[][] arestasCrescentes() {
                int [][] result = this.E;
                for (int j=0; j<this.qtdArestas()-1; j++) {
                        for (int i=0; i<this.qtdArestas()-1-j; i++) {
                                if (this.peso(result[i][0], result[i][1]) > this.peso(result[i+1][0],
result[i+1][1])) {
                                        int[] temp = new int[3];
                                        temp[0] = result[i+1][0];
                                        temp[1] = result[i+1][1];
                                        // a terceira casa no array de arestas representa o index no
array de peso
                                        temp[2] = result[i+1][2];
                                        result[i+1][0] = result[i][0];
                                        result[i+1][1] = result[i][1];
                                        result[i+1][2] = result[i][2];
                                        result[i][0] = temp[0];
                                        result[i][1] = temp[1];
                                        result[i][2] = temp[2];
                                }
                        }
                }
                return result;
        }
```

Por fim, os resultados são printados na tela conforme as exigências da questão:

```
protected void printKruskal() {
	float cost = 0;
	for (int i=0; i<A.size(); i++) {
		cost = cost + super.peso(A.get(i).get(0), A.get(i).get(1));
	}
	System.out.println(cost);
	for (int i=0; i<A.size(); i++) {
		System.out.print(A.get(i).get(0) + "-" + A.get(i).get(1));
		if (i != A.size()-1) {
			System.out.print(", ");
		}
	}
	System.out.println();
}
```