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 I

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.

Objeto:

```
// Construtor com parâmetros
	public Grafo_ND_P(String[] Vertice,int[][] E_Aresta,float[] w_Peso) {
		// Array de vertices
		V = Vertice;
		// Array de Arestas, sendo N por 3, [N][0] = recebe primeiro elemento da aresta,
		// [N][1] = recebe segundo elemento da aresta, [N][2] = recebe o index desta aresta
		// [N][3] = Marcação, usado para saber o estado da aresta "fechado" ou "aberto"
		E = E_Aresta;
		// Array de pesos
		w = w_Peso;
	}
```

O Grafo é separado entre vértices, arestas e pesos, na classe Objeto "Grafo_ND_P", temos um array de vértice "V" que guarda todos os vértices, tendo os rótulos em seu conteúdo, um multiArray que guarda quais vértices pertencem a aresta além do index e uma marcação que é utilizada para o ciclo Euclidiano, além de um array que guarda os valores dos pesos.

Exercício 1:

- **qtdVertices(): retorna a quantidade de vértices**

  A quantidade de vértices é dada pelo tamanho do array "V"

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

- **qtdArestas(): retorna a quantidade de arestas**

  A quantidade de arestas é dada pelo tamanho do multiArray "E"

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

- **grau(v): retorna o grau do vértice v**

Utiliza um contador, para verificar quantas arestas diferentes aquele vértice possui.

```
// Verifica o grau de uma vertice por meio de um contador de arestas
	protected int grau(int v) {
		// Contador utilizado para contagem de grau
		int count = 0;
		// 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) {
				// Incrementa contador
				count++;
			}
			// Verifica se a aresta possue o vertice do parâmetro no segundo
elemento
			if(E[i][1] == v ) {
				// Incrementa contador
				count++;
			}
		}
		// Retorna o contador
		return count;
	}
```

- **rotulo(v): retorna o rótulo do vértice v**

Retorna o rótulo, ou seja o conteúdo do array “V”

```
// Retorna o conteudo do Array de Vertices(Rótulo)
	protected String rotulo(int index) {
		return V[index-1];
	}
```

- **vizinhos(v): retorna os vizinhos do vértice v**

Utiliza uma lista para retornar todos os vértices, os quais possuem ligação com o vértice analizado.

```
// 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;
        }
```

- **haAresta(u, v): se {u, v} ∈ E, retorna verdadeiro; se não existir, retorna falso**

Percorre pelas arestas, até encontrar uma aresta entre "v" e "u", caso não encontre retorna "false";

```
// Verifica existência de aresta entre u e v
        protected boolean haAresta(int u, int v) {
                // Percorre todas as arestas
                for (int i = 0; i < E.length; i++) {
                        // Verifica se u é o primeiro elemento da aresta e v é o segundo
elemento, além de verificar o caso inverso
                        if ((E[i][0] == u && E[i][1] == v) || (E[i][0] == v && E[i][1] == u))
{
                                // Retorna verdadeiro
                                return true;
                        }
                }
                // Retorna false por default
                return false;
        }
```

- **peso(u, v): se {u, v} ∈ E, retorna o peso da aresta {u, v}; se não existir, retorna um valor infinito positivo**

Verifica todas as arestas para encontrar uma que exista entre "u" e "v", caso encontre retorna o peso localizado no array "w", caso não exista retorna uma constante infinita positiva.

```
// Verifica o peso da aresta entre u e v
        protected Float peso(int u, int v) {
                // Percorre todas as arestas
                for (int i = 0; i < E.length; i++) {
                        // Verifica se u é o primeiro elemento da aresta e v é o segundo
elemento, além de verificar o caso inverso
                        if ((E[i][0] == u && E[i][1] == v) || (E[i][0] == v && E[i][1] == u))
{
```

```
                        // Retorna seu peso
                        return w[E[i][2]];
                    }
                }
                // Retorna peso infinito positivo caso não exista aresta entre os dois
                return POSITIVE_INFINITY;
        }
```

- **ler(arquivo)2 : deve carregar um grafo a partir de um arquivo no formato especificado ao final deste documento**

  Utiliza dois métodos, um deles é um método que lê o arquivo e coloca em uma lista, e após isso manipula essa lista para que os conteúdos fiquem em arrays para serem introduzidos ao objeto.

```
// Leitura de um arquivo e inserção do mesmo em uma lista, sendo que cada indice contém uma linha completa
        public static List<String> readFileInList(String fileName) {
          List<String> lines = Collections.emptyList();
          try
          {
            lines =
             Files.readAllLines(Paths.get(fileName), StandardCharsets.UTF_8);
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
          return lines;
        }


// Leitura do arquivo e inserção nos atributos do Grafo
        protected void ler(File file) throws IOException {
                // Chama o método para colocar o arquivo em uma lista
                List l = readFileInList(file.getPath());

                // Variavel para conta de Vertices(numero de linhas até chegar nas arestas - 1)
                int lineEdges = -1;

                // Coloca a primeira linha em uma string
                String strFor1 = (String) l.get(0);
                // Manipula a string para pegar o valor de Vertices
                lineEdges = Integer.parseInt(strFor1.substring(10));

                // Array para as Vertices
                String[] Vert = new String [lineEdges];
```

```
		// Calculo para verificar o numero de Arestas
		int calcAr = l.size() - lineEdges-2;

		// Array para os Pesos
		float[] Pesos = new float[calcAr];

		// MultArray para as Arestas
		int[][] Arest = new int[calcAr][4];

		// Padrão utilizado para pegar os valores das arestas e pesos
		Pattern p = Pattern.compile("[0-9]*\\.?[0-9]+");
		// Lista que recebe os valores dos pesos e arestas
		List<Float> arestasPesosList = new ArrayList<Float>();

		// Percorre as linhas do arquivo que possuem as arestas e pesos
		for (int i = lineEdges+1; i < l.size(); i++) {
			// Verifica o pardrao com o texto
			Matcher m = p.matcher((CharSequence) l.get(i));
			// Onde o padrão bater com o descrito
			while (m.find()) {
				// Adiciona a lista os valores
				arestasPesosList.add(Float.parseFloat(m.group()));
			}
		}

		// Percorre o numero de vertices vezes
		for (int i = 0; i < lineEdges; i++) {
			// Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
			// Para inserir no array de Pesos
			// Index do Peso - Index da arestasPesoList
			// 0-2 1-5 2-8 3-11 4-14 5-17 6-20
			Pesos[i] = arestasPesosList.get((i)+2*(i+1));
			// Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
			// Para inserir no array de Arestas
			// Index da Aresta[i][0] - Index da arestasPesoList
			// 0-0 1-3 2-6 3-9  4-12 5-15 6-18
			Arest[i][0] = Math.round(arestasPesosList.get((i)*3));
			// Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
			// Para inserir no array de Arestas
			// Index do da Aresta[i][1] - Index da arestasPesoList
			// 0-1 1-4 2-7 3-10 4-13 5-16 6-19
			Arest[i][1] = Math.round(arestasPesosList.get(((i)*3)+1));
			// Recebe o index
			Arest[i][2] = (i);
			// str1 definida para pegar todas as Vertices
```

```
                String str1 = (String) l.get(i+1);
                // Retira "" da string
                str1 = str1.replace("\"", "");
                // Separa a string em partes
                String[] parts = str1.split(" ");
                // Retira o numero do vertice da string
                parts[0] = "";
                // Junta todas as partes
                str1 = String.join(" ",parts);
                // Retira espaço em branco no inicio da string
                str1 = str1.substring(1);
                // Coloca o rotulo da vertice no array
                Vert[i] = str1;
            }

            // Percorre desde o numero de vertices até o numero de arestas
            for (int i = lineEdges; i < calcAr; i++) {
                // Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
                // Para inserir no array de Pesos
                // Index do Peso - Index da arestasPesoList
                // 0-2 1-5 2-8 3-11 4-14 5-17 6-20
                Pesos[i] = arestasPesosList.get((i)+2*(i+1));
                // Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
                // Para inserir no array de Arestas
                // Index da Aresta[i][0] - Index da arestasPesoList
                // 0-0 1-3 2-6 3-9  4-12 5-15 6-18
                Arest[i][0] = Math.round(arestasPesosList.get((i)*3));
                // Como a lista de valores possue tanto pesos quanto arestas, utiliza
lógica de PA
                // Para inserir no array de Arestas
                // Index do da Aresta[i][1] - Index da arestasPesoList
                // 0-1 1-4 2-7 3-10 4-13 5-16 6-19
                Arest[i][1] = Math.round(arestasPesosList.get(((i)*3)+1));
                // Recebe o index
                Arest[i][2] = (i);
            }
            // w recebe os Pesos
            w = Pesos;
            // E recebe as Arestas
            E = Arest;
            // V recebe os Vertices
            V = Vert;
        }
```

Exercício 2:

Busca é uma questão muito importante de grafos. Neste exercício colocamos em prática um algoritmo de busca em largura que retorna, a partir de um vértice, uma árvore com todos os caminhos atingíveis passando pelo menor número de arestas.
Na implementação foi criada uma classe filha de Grafo_ND_P, com os atributos Distancia(array de inteiros) e Ancestral (array de Strings) que compõe a árvore:

```
public class BuscaEmLargura extends Grafo_ND_P{
        private int[] Distancia;
        private String[] Ancestral;
```

O objeto pode ser criada apartir de um arquivo de grafo ou de um grafo já existente:

```
// Constroi um grafo apartir do arquivo e executa a busca em largura com a raiz indicada
public BuscaEmLargura(File fileGrafo, int raiz) throws IOException {
        super(fileGrafo);
        executarBusca(raiz);
}

// executa a busca em largura com a raiz indicada apartir de um grafo ja existente
public BuscaEmLargura(Grafo_ND_P parent, int raiz) {
        super(parent.V, parent.E, parent.w);
        executarBusca(raiz);
}
```

O algoritmo em si é bastante semelhante ao apresentado em sala, com diferenças apenas na inicialização das estruturas, de forma a se moldar melhor ao Java:

```
public void executarBusca(int raiz) {
        //Definindo estruturas que serão utilizadas
        Distancia = new int[super.V.length];
        for (int i = 0; i < Distancia.length; i++) {
                Distancia[i] = -1;          //Usando -1 como representação de distância infinita
        }
        Ancestral = new String[super.V.length];          // não precisa ser inicializada pois java inicia como vazio
        boolean[] conhecido = new boolean[super.V.length];  // não precisa ser inicializada pois java inicia como false

        // Definindo valores da raiz
        conhecido[raiz-1] = true;
        Distancia[raiz-1] = 0;

        // Cria lista de vértices a visitar e adiciona raiz
        Queue<Integer> q = new LinkedList<>();
        q.add(raiz);
        while(q.peek() != null) {
                // Retira primeiro da lista
                int u = q.poll();
                // Encontra os vizinhos do vértice atual
```

```
                    List<Integer> vizinhosU = super.vizinhos(u);
                    // Iteração por todos os vizinhos
                    for (int i=0; i<vizinhosU.size(); i++) {
                        // Se o vizinho não foi visitado, defina que ele foi visitado, defina sua
distância como seu antecessor +1
                        // e defina seu antecessor e adiciona vizinho na fila
                        if (conhecido[vizinhosU.get(i)-1] == false) {
                            conhecido[vizinhosU.get(i)-1] = true;
                            Distancia[vizinhosU.get(i)-1] = Distancia[u-1] + 1;
                            Ancestral[vizinhosU.get(i)-1] = super.rotulo(u);
                            q.add(vizinhosU.get(i));
                        }
                    }
                }
                printArvore();
        }
```

Após isso é feito apenas o print da árvore de acordo com o indicado pelo professor. Neste exercício o ancestral é registrado como String pois no print não precisamos fazer um "traceback" dos ancestrais para para encontrar o caminho.

Exercício 3:

Existe uma propriedade em Ciclos Euclidianos, que se existe um vértice com grau ímpar, a existência de um Ciclo Euclidiano é impossível.

```
// Verifica se todos os graus são par, caso não sejam, o ciclo Euleriano não existe
        protected int existeCicloEuleriano() {
                // Percorre o numero de Vertices
                for (int i = 1; i <= V.length; i++) {
                    // Chama o método que pega o grau e verifica se é par
                    if ((grau(i)) % 2 != 0) {
                        // Caso não par retorna 0
                        return 0;
                    }
                }
                // Caso todos sejam pares retorna 1
                return 1;
        }
```

Método utilizado para retornar uma aresta disponível, sendo que retorna o seu index e vértice.

```
        // Encontra o index e outro vertice
        protected List<Integer> findByNumber( int path, int v) {
                // Lista para index e Vertice
                List<Integer> indexN2 = new ArrayList<Integer>();
```

```
		// Percorre todas as Arestas
		for (int i = 0; i < E.length; i++) {
			// Caso aresta esteja disponivel, e valores das arestas estejam corretas
			if (E[i][1] == path && E[i][0] == v && E[i][3] == 1) {
				// Adiciona index
				indexN2.add(i);
				// Adiciona Vertice da aresta
				indexN2.add(E[i][1]);
				// retorna Lista
				return indexN2;
			}
			// Caso aresta esteja disponivel, e valores das arestas estejam corretas
			if (grafo.E[i][0] == path && E[i][1] == v && E[i][3] == 1) {
				// Adiciona index
				indexN2.add(i);
				// Adiciona Vertice da aresta
				indexN2.add(E[i][0]);
				// retorna Lista
				return indexN2;
			}
		}
		// Retorna null, caso n tenha
		return null;
	}
```

Encontra o caminho euleriano, utilizando o método anterior, além indisponibilizar arestas usadas, para que apenas faça o caminho com todas as arestas até que elas fiquem indisponíveis e volte para a origem, sempre começando pelo vértice 1.

```
	// Busca o caminho euleriano por meio do primeiro Vertice
	protected List<Integer> caminhoCicloEuleriano() {
		// Arestas podem estar "abertas" ou "fechadas", caso abertas o caminho pode passar por elas, caso fechadas
		// não passa

		// Deixa todas as Arestas como abertas
		for (int i = 0; i < E.length; i++) {
			E[i][3] = 1;
		}

		// Lista do Caminho
		List<Integer> pathFinal = new ArrayList<Integer>();

		// Lista para o index e o outro vertice
		List<Integer> indexN2 = new ArrayList<Integer>();

		// Vertice usado para o caminho
		int v = 1;
```

```
                // Adiciona vertice ao caminho
                pathFinal.add(v);
                while (true) {
                        // Lista de vizinhos para verificar os caminhos
                        List<Integer> paths = vizinhos(v);
                        // Sort para sempre buscar o caminho com menor indice dos Vertices
disponiveis
                        paths.sort(null);
                        // Percorre o tamanho dos vizinhos
                        for (int i = 0; i < paths.size(); i++) {
                                // Busca o vizinho que possui disponibilidade
                                indexN2 = findByNumber(paths.get(i), v);
                                // Se está aresta está disponível
                                if (indexN2 != null) {
                                        // Adiciona vertice ao caminho
                                        pathFinal.add(indexN2.get(1));
                                        // v recebe novo valor
                                        v = indexN2.get(1);
                                        // Fecha a aresta
                                        grafo.E[indexN2.get(0)][3] = 0;
                                        // "Pula" o for
                                        break;
                                }
                        }
                        // Caso v volte a ser 1, encerra o while
                        if (v == 1) {
                                // "Pula" o while
                                break;
                        }
                }
                // Retorna o caminho
                return pathFinal;
        }
```

Exercício 4:

Bellman-Ford é um algoritmo de caminho mínimo de um para todos, isto é, encontra o caminho mínimo a partir de um vértice raiz para todos os outros vértices, considerando o peso das arestas.

Na implementação foi feita uma classe filha de Grafo_ND_P, com os atributos Distancia (array de float) e Ancestral (array de inteiros). Já que esse algoritmo os pesos das arestas são considerados foi necessário registrar a distância como ponto flutuante. Para os ancestrais, no momento de printar o caminho fazemos um "traceback" por isso registramos, não o label do vértice ancestral, mas sim sua posição no array de vértices.

```
        public class BellmanFord extends Grafo_ND_P{
                private float[]  Distancia;
                private int[] Ancestral;
```

Para construir o objeto podemos tanto utilizar um arquivo de grafo, quanto um grafo já existente.

```
	// Constroi um grafo apartir do arquivo e executa o algoritmo de Bellman-Ford com a raiz indicada
	public BellmanFord(File fileGrafo, int raiz) throws IOException {
		super(fileGrafo);
		executarBellmanFord(raiz);
	}

	// executa o algoritmo de Bellman-Ford com a raiz indicada apartir de um grafo ja existente
	public BellmanFord(Grafo_ND_P parent, int raiz) {
		super(parent.V, parent.E, parent.w);
		executarBellmanFord(raiz);
	}
```

O algoritmo em si é semelhante ao apresentado em sala. Entretanto na verificação do caminho mínimo adicionamos outra etapa, que verifica também o caminho contrário. Explicado nos comentários do código:

```
	public void executarBellmanFord(int raiz) {
		// Inicialização
		Distancia = new float[super.V.length];
		for (int i=0; i<Distancia.length; i++) {
			Distancia[i] = Float.MAX_VALUE;   // Utilizando maior valor de float como infinito
		}
		Ancestral = new int[super.V.length];
		for (int i=0; i<Ancestral.length; i++) {
			Ancestral[i] = -1;		// Utiliando -1 para ancestral nulo
		}
		// Adicionando o valor da raiz
		Distancia[raiz-1] = 0;
		for (int i=1; i<=super.qtdVertices()-1; i++) {
			for (int j=0; j<super.qtdArestas(); j++) {
				// Aqui foram colocados 2 pois a verificação é feita através das arestas do grafo, o que gerou problemas já que o grafo é não-direcionado, foi adicionado mais um 'if' para garantir que o menor caminho fosse encontrado independente da ordem indicada no array das arestas.
				if (Distancia[super.E[j][1]-1] > Distancia[super.E[j][0]-1] + super.w[j]) {
					Distancia[super.E[j][1]-1] = Distancia[super.E[j][0]-1] + super.w[j];
					Ancestral[super.E[j][1]-1] = super.E[j][0]-1;
				}
				if (Distancia[super.E[j][0]-1] > Distancia[super.E[j][1]-1] + super.w[j]) {
					Distancia[super.E[j][0]-1] = Distancia[super.E[j][1]-1] + super.w[j];
```

```
                                        Ancestral[super.E[j][0]-1] = super.E[j][1]-1;
                                }
                        }
                }
                printBellmanFord();
        }
```

No print final fazemos o traceback e utilizamos uma estrutura de pilha para registrar o caminho apartir do vértice que estamos até a raiz. Foi utilizada a pilha pois nesse métodos encontramos o caminho de trás para frente, logo, a característica Last in First out(LIFO) serviu perfeitamente:

```
        public void printBellmanFord() {
                // Garante o print para cada uma das arestas
                for (int i=0; i<super.V.length; i++) {
                        // Pilha com o caminho mínimo
                        Stack<Integer> caminho = buscaAncestrais(i);
                        // printa o vértice atual
                        System.out.print(i+": ");
                        // printa toda a pilha do caminho
                        while (!caminho.empty()) {
                                // Aqui deixamos printando o rótulo do vértice, mas pode-se
deixar printando sua posição/número no array
                                System.out.print(super.rotulo(caminho.pop()+1)+", ");
                                //System.out.print((caminho.pop()+1)+" ");
                        }
                        // print da distância
                        System.out.println(" d="+Distancia[i]);
                }
        }

        public Stack<Integer> buscaAncestrais(int vertice) {
                // Aqui monta-se uma pilha indicando o caminho
                Stack<Integer> caminho = new Stack<Integer>();
                // While que verifica se chegamos a raiz, que tem como antecessor -1 (nulo)
                while (vertice != -1) {
                        // se adiciona o vértice a pilha e define o novo vértice como o
ancestral dele
                        caminho.push(vertice);
                        vertice = Ancestral[vertice];
                }
                return caminho;
        }
```

Por fim, ao rodar o main, o professor poderá verificar que temos um código preparado que executará os principais métodos de todos os exercícios indicando os grafos utilizados e os resultados.