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 3

Considerações Gerais:

A linguagem escolhida para resolução dos problemas foi Java, a estrutura utilizada foi diferente para os diferentes requerimentos dos exercícios.

Exercício 1:

Diferente das estruturas utilizadas nos exercícios anteriores, como o exercício é baseado em arestas, ficou mais fácil de trabalhar com uma lista de arestas, sendo que apontam para os diferentes vértices, tem seu peso/capacidade, além de que quando uma aresta é adicionada, outra com o sentido inverso e capacidade 0 é adicionada simultaneamente.

```
//Formato para cada aresta
   public Edge(int s, int t, int cap) {
     this.s = s;
     this.t = t;
     this.cap = cap;
   }

   //Adiciona as vértices no grafo, com todas as suas informações
         public static void addAresta(List<Edge>[] grafo, int s, int t, int cap) {
                  grafo[s].add(new Edge(s, t, cap));
                  grafo[t].add(new Edge(t, s, 0));
         }
```

A leitura do grafo foi feita por meio da dissecação de strings do arquivo e as colocando em Arrays, nos quais são posteriormente adicionados como arestas, com todas as informações necessárias para resolução do problema.

Após a dissecação para formar o grafo e suas arestas são implementados os seguintes códigos dentro do método "ler(File file)":

```
List<Edge>[] grafo = criarGrafo(Vert.length);
for (int i = 0; i < Arest.length; i++) {
         addAresta(grafo,Arest[i][0]-1,Arest[i][1]-1,(int)Pesos[i]);
}
return grafo;
```

Método "ler(File file)" completo:

```
// Leitura do arquivo e inserção nos atributos do Grafo
public static List<Edge>[] 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);
                        }
                List<Edge>[] grafo = criarGrafo(Vert.length);
                for (int i = 0; i < Arest.length; i++) {
                        addAresta(grafo,Arest[i][0]-1,Arest[i][1]-1,(int)Pesos[i]);
                }
                return grafo;
        }
```

Como dito anteriormente a criação do grafo é uma lista de arestas:

```
//Criação do grafo, no caso um array de Edges com as vértices
public static List<Edge>[] criarGrafo(int nodes) {
	List<Edge>[] grafo = new List[nodes];
	for (int i = 0; i < nodes; i++)
		grafo[i] = new ArrayList<>();
	return grafo;
	}
```

O método Baseado em Edmond-karp para fluxo máximo funciona da seguinte forma:
Ele faz uma busca em largura, baseado no s, grava os caminhos que chegam até t, quando um caminho chega em t volta no sentido oposto, aumentando as capacidades das vértices opostas pelo valor mínimo de capacidade de todas as vértices inclusas no caminho, além de aumentar o valor do fluxo, se não encontra o destino, volta a procurar. Quando a capacidade de uma aresta no sentido oposto chega ao valor da "normal", ela é inutilizada por qualquer caminho posterior, quando a busca acaba e as capacidades mínimas dos caminhos são adicionadas no fluxo, o fluxo retorna com o valor de fluxo máximo do grafo.

```
//Edmond-karp fluxo maximo
public static int fluxo(List<Edge>[] grafo, int s, int t) {
	//fluxo
	int fluxo = 0;
	//seta uma lista Fila
	int[] Fila = new int[grafo.length];
	//loop para rodar até finalizar o processo
	while (1 == 1) {
		//incializa o contador
		int count = 0;
		//Fila recebe o primeiro elemento
		Fila[count++] = s;
		//lista de arestas/caminho
		Edge[] caminho = new Edge[grafo.length];
		//loop referente a fila enquanto n acessa o elemento final e o contador n passe
do loop
		for (int x = 0; x < count && caminho[t] == null; x++) {
			//loop para os vizinhos
			for (Edge e : grafo[Fila[x]]) {
				//caso a capacidade é maior que o peso e o elemento destino
da aresta exista
				if (caminho[e.t] == null && e.cap > e.f) {
					//caminho recebe a aresta
					caminho[e.t] = e;
					//Fila recebe o destino da aresta
					Fila[count++] = e.t;
				}
			}
		}
```

```
                //Caso o caminho n encontre o elemento final
                if (caminho[t] == null)
                        break;
                // variavel infinita positiva que no caso é a capacidade minima do caminho
                int minCap = Integer.MAX_VALUE;
                //loop do caminho de volta
                for (int u = t; u != s; u = caminho[u].s)
                        //verifica a menor capacidade do caminho
                        minCap = Math.min(minCap, caminho[u].cap - caminho[u].f);
                //loop do caminho de volta
                for (int w = t; w != s; w = caminho[w].s) {
                        //incremento no valor da capacidade
                        caminho[w].f += minCap;
                }
                //adiciona o valor no fluxo
                fluxo += minCap;
        }
        //retorna o fluxo
        return fluxo;
}
```

Exercício 2: Para implementar o algoritmo de Hopcrft-Karp utilizamos arrays de int para identificação dos vértices e dos valores em D. Para facilitar no desenvolvimento do código também criamos um método que diferencia o conjunto X e Y do grafo recebido, com base nas arestas.

```
public void encontraPartes() {
		List<Integer> elements = new ArrayList<Integer>();
		for (int i=0; i < super.E.size(); i++) {
			if (i > 0) {
				if (super.E.get(i).get(0).intValue() !=
super.E.get(i-1).get(0).intValue()) {
					elements.add(super.E.get(i).get(0));
				} else {

				}
			} else {
				elements.add(super.E.get(i).get(0));
			}
		}
		this.X = new int[elements.size()];
		int restante = super.V.size() - elements.size();
		if (restante < 0) { restante = restante * (-1); }
		this.Y = new int[restante];
		int indexX = 0;
		int indexY = 0;
		for (int i=1; i<=super.V.size(); i++) {
			if (elements.contains(i)) {
				this.X[indexX] = i;
				indexX += 1;
			} else {
				this.Y[indexY] = i;
				indexY += 1;
			}
		}

	}
```

Basicamente Adicionamos todos os vértices do lado esquerdo da identificação das arestas em uma lista, logicamente sem repetir esses vértices, e ao fim esse conjunto será o X e todos os outros vértices que não estiverem nele iram para o conjunto Y.

Não é a melhor abordagem para essa situação, mas se mostrou eficiente para o nosso propósito. Talvez um abordagem melhor seria utilizar os vizinho num modelo semelhante ao que utilizamos para encontrar conjuntos independentes no Exercício 3.

Fora isso é interessante notar que ao instanciarmos a estrutura D do algoritmo, definimos seu tamanho como "número de vértices + 1", esse mais um serviu como vértice nulo para o resto do código.
Também foi necessário criar uma nova classe para Grafos Não-dirigidos e Não-ponderados, pois até então estávamos apenas ignorando o valor das arestas, entretanto nos grafos fornecidos para atividade nessa questão havia a necessidade de mudar a forma da leitura dos arquivos pela ausência do peso das arestas.

Exercício 3: Para a implementação do algoritmo de Lawler para coloração de grafos, vários outros métodos tiveram que ser implementados.

```
public List<List<Integer>> geraS() {
		List<List<Integer>> S = new ArrayList<List<Integer>>();
		for (int i=0; i<Math.pow(2.0, Double.valueOf(super.V.length)); i++) {
			S.add(null);
			if (i > 0) {
				char[] binario = Integer.toBinaryString(i).toCharArray();
				int difer = super.V.length - binario.length;
				List<Integer> verts = new ArrayList<Integer>();
				for (int j=0; j<binario.length; j++) {
					if (binario[j] == '1') {
						verts.add(Integer.valueOf(super.V[j+difer]));
					}
				}
				S.set(i, verts);
			}
		}
		return S;
	}
```

Usando como base a representação binária demonstrada pelo professor, fizemos esse método que faz a permutação de todas as possibilidades do conjunto de vértices do grafo e retorna uma lista dessas listas de vértices representados por inteiros.

```
public Grafo_ND_P criaGrafoS(List<Integer> S) {
		String[] newV = new String[S.size()];
		float[]  neww = null;
		for (int i=0; i<newV.length; i++) {
			newV[i] = S.get(i).toString();
		}
		List<List<Integer>> arestas = new ArrayList<List<Integer>>();
		for (int i=0; i<super.E.length; i++) {
			if(S.contains(super.E[i][0]) && S.contains(super.E[i][1])) {
				List<Integer> values = new ArrayList<Integer>();
				values.add(super.E[i][0]);
				values.add(super.E[i][1]);
				arestas.add(values);
			}
		}
		int[][] newE = new int[arestas.size()][2];
		for (int i=0; i<newE.length; i++) {
			newE[i][0] = arestas.get(i).get(0);
			newE[i][1] = arestas.get(i).get(1);
		}
		Grafo_ND_P gLinha = new Grafo_ND_P(newV, newE, neww);
		return gLinha;
	}
```

Tendo a lista de permutações pronta sabíamos que teríamos que utilizar ela para gerar diferentes grafos durante a execução, somente com os vértices de S. Assim

desenvolvemos este método que faz exatamente isso, basicamente excluindo os vértices inexistentes e as arestas com vértices que não pertencem a esse novo grafo.

```
public List<List<Integer>> encontraConjIndep(Grafo_ND_P grafo) {
		Queue<Integer> q = new PriorityQueue<Integer>();
		List<List<Integer>> result = new ArrayList<List<Integer>>();
		int indexResult = -1;
		for (int i=0; i < grafo.V.length; i++) {
			q.add(Integer.valueOf(grafo.V[i]));
		}
		while (!q.isEmpty()) {
			int x = q.poll();
			indexResult += 1;
			result.add(new ArrayList<Integer>());
			result.get(indexResult).add(x);
			List<Integer> vizX = super.vizinhos(x);
			Integer[] disponiveis = new Integer[q.size()];
			disponiveis = q.toArray(disponiveis);
			for (int i=0; i<disponiveis.length; i++) {
				if (!(vizX.contains(disponiveis[i])) && disponiveis[i] != x) {
					if (result.get(indexResult).size() == 1) {

result.get(indexResult).add(Integer.valueOf(disponiveis[i]));
						if (q.contains(Integer.valueOf(disponiveis[i]))) {
							q.remove(Integer.valueOf(disponiveis[i]));
						}
					} else {
						boolean ehVizinho = false;
						for (int j=1; j<result.get(indexResult).size(); j++) {
							if
(super.vizinhos(result.get(indexResult).get(j)).contains(Integer.valueOf(disponiveis[i]))) {
								ehVizinho = true;
							}
						}
						if (!ehVizinho) {

result.get(indexResult).add(Integer.valueOf(disponiveis[i]));
							if
(q.contains(Integer.valueOf(disponiveis[i]))) {

q.remove(Integer.valueOf(disponiveis[i]));
							}
						}
					}

				}
			}
		}
		return result;
	}
```

Com tudo isso em mão chegou a hora de descobrir como encontraríamos os conjuntos independentes. Esse método usa a vizinhança como base, seguindo os seguintes passos:

1) **Adiciona todos os vértices em uma fila e executa até a fila ficar vazia;**
2) **Pega o primeiro vértice da fila (Vk) e coloca em um conjunto C;**
3) **Itera pelos vértices da fila até encontrar um (Vp) que não seja vizinho de Vk;**
4) **Se certifica que Vp não é vizinho de nenhum vértice do conjunto C, caso seja volte a iterar;**
5) **Adiciona Vp ao conjunto C e o remove da fila;**
6) **Retorna todos os conjuntos encontrados**

Por fim, um pequeno método para encontrar a qual item do conjunto S determinado conjunto independente de G' corresponde

```
public int f(List<List<Integer>> S, List<Integer> I) {
        for (int i=1; i<S.size(); i++) {
                if (I.containsAll(S.get(i)) && I.size() == S.get(i).size()) {
                        return i;
                }
        }
        return 0;
}
```

Usamos este para encontrar o I da linha 9 do algoritmo encontrado. Uma vez que sabemos o número que I (conjunto independente) corresponde na estrutura S, podemos simplesmente fazer s-i para descobrir como seria o grafo' do S atual sem os vértices do conjunto independente I.