UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

ALEX DAVIS NEUWIEM DA SILVA
LUAN DINIZ MORAES
LUCAS CASTRO TRUPPEL MACHADO

# Relatório da Atividade 1 de Grafos

Florianópolis, 2023.

**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**
**CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

**ALEX DAVIS NEUWIEM DA SILVA**
**LUAN DINIZ MORAES**
**LUCAS CASTRO TRUPPEL MACHADO**

**Relatório da Atividade 1 de Grafos**

Relatório submetido à Unidade Curricular de Grafos, da 4ª fase do Curso de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina, Câmpus Trindade.

Professor:
Dr. Rafael de Santiago

**Florianópolis, 2023.**

**COMO EXECUTAR O CÓDIGO**

Primeiramente, é importante ressaltar que os comandos a seguir devem ser executados dentro da pasta “A1” no terminal.

**1)** Para executar um único algoritmo por vez, você pode digitar o seguinte comando no terminal:

**python3 arquivo.py grafo.txt**

Onde “arquivo.py” é o arquivo com o algoritmo a ser executado e “grafo.txt” é o grafo passado como argumento.

Alguns algoritmos demandam um vértice como argumento, por isso o comando a ser executado nesse caso deve ser:

**python3 arquivo.py grafo.txt vertice**

Para que o algoritmo funcione corretamente, “vertice” deve ser um número inteiro positivo, diferente de zero.

**2)** Você também pode executar mais de um algoritmo por vez, basta executar o arquivo “main.py”, nesse caso, os arquivos do tipo “grafo.txt” e os vértices devem ser passados como argumento dentro do código:

**python3 main.py**

O código base de “main.py” apresenta quatro grafos como exemplo, basta alterar o nome do arquivo passado e/ou os vértices para a função.

## 1. REPRESENTAÇÃO DE UM GRAFO

Nesta atividade, optamos por utilizar a linguagem de programação python e implementamos uma classe chamada “Grafo” para representar um grafo não-dirigido e ponderado. Todo grafo é instanciado a partir da leitura de um arquivo .txt, nesse processo, é construída uma matriz de adjacências e os atributos da classe Grafo são obtidos. Abaixo, podemos encontrar o trecho de código em que o arquivo é lido e a matriz de adjacências é construída com base nos dados extraídos.

```python
def lerArquivo(self, nome_arquivo: str) -> None:
    arquivo = open(nome_arquivo, "r")
    lendo_vertices = True
    for linha_string in arquivo:
        linha = linha_string.split()
        if lendo_vertices and linha[0] == "*vertices":
            self.__quantidade_vertices = int(linha[1])
        elif lendo_vertices and linha[0] == "*edges":
            lendo_vertices = False
            self.__inicializar_matriz()
        elif lendo_vertices:
            self.__rotulos_vertices.append(linha[1])
        else:
            u = self.__get_indice(linha[0])
            v = self.__get_indice(linha[1])
            u, v = self.__ordernar_vertices(u, v)
            self.__matriz[u-1][v-1] = int(linha[2])
            self.__quantidade_arestas += 1
```

Na imagem abaixo, podemos encontrar tanto os métodos “qtdVertices” e “qtdArestas”, que retornam variáveis obtidas no trecho de código acima, quanto os métodos “grau” e “rotulo”, que são implementados com base nos dados retirados da matriz de adjacências do grafo.

```python
def qtdVertices(self) -> int:
    return self.__quantidade_vertices

def qtdArestas(self) -> int:
    return self.__quantidade_arestas

def grau(self, v: int) -> int:
    grau = 0
    for j in range(v-1):
        if self.__matriz[v-1][j] > 0:
            grau += 1
    for i in range(v, self.__quantidade_vertices):
        if self.__matriz[i][v-1]:
            grau += 1
    return grau

def rotulo(self, v: int) -> str:
    return self.__rotulos_vertices[v-1]
```

Abaixo, temos o restante dos métodos necessários para atender ao item 1. Todos estes métodos utilizam a matriz de adjacências em seu funcionamento.

```python
def vizinhos(self, v: int) -> list:
    vizinhos = []
    for j in range(v-1):
        if self.__matriz[v-1][j] > 0:
            vizinhos.append(j+1)
    for i in range(v, self.__quantidade_vertices):
        if self.__matriz[i][v-1]:
            vizinhos.append(i+1)
    return vizinhos

def haAresta(self, u: int, v: int) -> bool:
    u, v = self.__ordernar_vertices(u, v)
    haAresta = False
    if self.__matriz[u-1][v-1] > 0:
        haAresta = True
    return haAresta

def peso(self, u: int, v: int) -> float:
    u, v = self.__ordernar_vertices(u, v)
    if self.__matriz[u-1][v-1] > 0:
            peso = self.__matriz[u-1][v-1]
    else:
        peso = float("inf")
    return peso
```

Optamos em implementar o grafo desta maneira pois assim foi possível implementar os métodos e obter informações úteis como vértices, arestas e pesos por meio de um simples acesso à matriz.

## 2. BUSCA EM LARGURA

Para realizar a busca em largura proposta no item 2, utilizamos o pseudocódigo abaixo como base:

**Algoritmo 3:** Busca em largura.

**Input** :um grafo $G = (V, E)$, vértice de origem $s \in V$

`// configurando todos os vértices`

$C_v \leftarrow$ **false** $\forall v \in V$

$D_v \leftarrow \infty \; \forall v \in V$

$A_v \leftarrow$ **null** $\forall v \in V$

`// configurando o vértice de origem`

$C_s \leftarrow$ **true**

$D_s \leftarrow 0$

`// preparando fila de visitas`

$Q \leftarrow$ **Fila()**

$Q$.enqueue($s$)

`// propagação das visitas`

**while** $Q$.*empty()* = ***false*** **do**

    $u \leftarrow Q$.dequeue()

    **foreach** $v \in N(u)$ **do**

        **if** $C_v$ = ***false*** **then**

            $C_v \leftarrow$ **true**

            $D_v \leftarrow D_u + 1$

            $A_v \leftarrow u$

            $Q$.enqueue($v$)

**return** $(D, A)$

Fonte: SANTIAGO, 2023, p. 24

A imagem abaixo apresenta a função base que irá chamar o método de busca em largura para o grafo enviado como argumento. Primeiramente, o grafo é instanciado e, em seguida, a busca em largura se inicia.

Com o retorno da função, a saída é impressa na tela, de forma que cada linha contém os vértices encontrados em cada nível da árvore de busca em largura.

```python
@staticmethod
def buscar(nome_arquivo: str, s: int):
    grafo = Grafo(nome_arquivo)

    distancia, antecessor = Busca.__busca_largura(grafo, s)
    for nivel in range(max(distancia) + 1):
        saida_nivel = f"{nivel}: "
        for v in range(len(distancia)):
            if distancia[v] == nivel:
                saida_nivel += f"{v+1} "
        print(saida_nivel)
```

O algoritmo a seguir mostra a adaptação do pseudocódigo mostrado anteriormente. Durante sua execução, são utilizadas quatro listas que servirão para armazenar informações necessárias para a busca:

- A lista "visitados" é usada para verificar se um determinado vértice v já foi visitado ou não (Cv);
- A lista "distancia" armazena a distância percorrida para alcançar o vértice v (Dv);
- A lista "antecessor" indica o vértice antecessor a v em uma busca em largura a partir de s (Av);
- A lista “fila” armazena os vértices que serão visitados.

```python
@staticmethod
def __busca_largura(grafo: Grafo, s: int)  -> tuple:
    visitados = [False for i in range(grafo.qtdVertices())]
    distancia = [float("inf") for i in range(grafo.qtdVertices())]
    antecessor = [None for i in range(grafo.qtdVertices())]

    visitados[s-1] = True
    distancia[s-1] = 0

    fila = []
    fila.append(s)

    while len(fila) > 0:
        u = fila.pop(0)
        for v in grafo.vizinhos(u):
            if not visitados[v-1]:
                visitados[v-1] = True
                distancia[v-1] = distancia[u-1] + 1
                antecessor[v-1] = u
                fila.append(v)

    return (distancia, antecessor)
```

Este algoritmo foi escolhido por causa de sua eficiência em grafos densos, já que o algoritmo explora o grafo em camadas, limitando o número de nós a serem visitados antes de chegar ao destino. Além disso, a Busca em Largura é completa, garantindo encontrar um caminho mínimo do nó inicial para o nó objetivo, se existir um caminho entre eles.

## 3. ALGORITMO DE HIERHOLZER

Neste item, determinamos a existência de um Ciclo Euleriano em um grafo por meio do Algoritmo de Hierholzer. A seguir, podemos encontrar o pseudocódigo utilizado como base para a implementação.

**Algoritmo 5:** Algoritmo de Hierholzer.

```
Input  :um grafo não-orientado G = (V, E)
foreach e ∈ E do
    C_e ← false
v ← selecionar um v ∈ V arbitrariamente, que esteja conectado a uma aresta.
// "buscarSubcicloEuleriano" invoca o Algoritmo 6
(r, Ciclo) ← buscarSubcicloEuleriano(G, v, C)
if r = false then
    return (false,null)
else
    if ∃e ∈ E : C_e = false then
        return (false,null)
    else
        return (true, Ciclo)
```

Fonte: SANTIAGO, 2023, p. 34

No código abaixo, primeiramente é verificado se cada um dos vértices possui uma quantidade par de vizinhos (grau par), o que é um requisito para termos um Ciclo Euleriano em um grafo. Em seguida, é criada uma lista que determina quais arestas já foram visitadas e seguimos para o algoritmo auxiliar de busca por um Subciclo Euleriano. Com o resultado da busca, é impresso na tela “0” caso não haja um Ciclo Euleriano no grafo, mas caso um ciclo realmente exista, será mostrado “1” seguido de uma lista que contém a sequência de vértices visitados.

```python
@staticmethod
def encontrar_ciclo_euleriano(grafo: Grafo) :
    for i in range(grafo.qtdVertices()):
        if (len(grafo.vizinhos(i + 1)) % 2) != 0 :
            print("0")
            return
    conhecidas = [False for i in range(grafo.qtdArestas())]
    resposta, ciclo = Hierholzer.__encontrar_subciclo_euleriano(grafo, 1, conhecidas)
    if (resposta == False) :
        print("0")
        return
    else:
        for aresta in conhecidas:
            if (aresta == False) :
                print("0")
                return
        print("1")
        print(str(ciclo).replace(" ","").replace("[","").replace("]",""))
        return
```

A imagem a seguir apresenta o pseudocódigo do algoritmo auxiliar que determina se existe algum tipo de Ciclo Euleriano em um grafo.

**Algoritmo 6:** Algoritmo de Auxiliar "$buscarSubcicloEuleriano$".

**Input** :um grafo não-orientado $G = (V, E)$,um vértice $v \in V$, o vetor de arestas visitadas $C$

$Ciclo \leftarrow (v)$
$t \leftarrow v$
**repeat**
    // Só prossegue se existir uma aresta não-visitada conectada a $Ciclo$.
    **if** $\nexists u \in N(v) : C_{\{u,v\}} =$ ***false*** **then**
        **return** (**false**,**null**)
    **else**
        $\{v, u\} \leftarrow$ selecionar uma aresta $e \in E$ tal que $C_e =$**false**
        $C_{\{v,u\}} \leftarrow$ **true**
        $v \leftarrow u$
        // Adiciona o vértice $v$ ao final do ciclo.
        $Ciclo \leftarrow Ciclo \cdot (v)$
**until** $v = t$

/* Para todo vértice $x$ no $Ciclo$ que tenha uma aresta adjacente não visitada. */

**foreach** $x \in \{u \in Ciclo : \exists\{u, w\} \in \{e \in E : C_e = \textit{false}\}\}$ **do**

  $(r, Ciclo') \leftarrow buscarSubcicloEuleriano(G, x, C)$

  **if** $r = \textit{false}$ **then**

    **return** (**false**,**null**)

  Assumindo que $Ciclo = \langle v_1, v_2, \ldots, x, \ldots, v_1 \rangle$ e $Ciclo' = \langle x, u_1, u_2, \ldots, u_k, x \rangle$, alterar $Ciclo$ para $Ciclo = \langle v_1, v_2, \ldots, x, u_1, u_2, \ldots, u_k, x, \ldots, v_1 \rangle$, ou seja, inserir o $Ciclo'$ no lugar da posição de $x$ em $Ciclo$.

**return** (**true**, $Ciclo$)

Fonte: SANTIAGO, 2023, p. 35

Primeiramente, o algoritmo apresentado na imagem abaixo inicia com um laço de repetição que só irá prosseguir se existir uma aresta não visitada conectada ao ciclo que queremos encontrar. Em seguida, a lista “ciclo” armazena um circuito que começa e termina com o mesmo vértice “v”, sendo que a variável “t” nos auxilia a encontrá-lo. Caso tal ciclo não exista, o algoritmo retorna a tupla “(False, None)”.

```
@staticmethod
def __encontrar_subciclo_euleriano(grafo: Grafo, v: int, conhecidas: list) -> tuple:
    ciclo = [v]
    t = v
    while True:
        cont1 = 0
        cont2 = 0
        for i in range(grafo.qtdArestas()) :
            if (conhecidas[i] == False) :
                if (grafo.getArestasNaoDirigido()[i][0] == t) :
                    t = grafo.getArestasNaoDirigido()[i][1]
                    conhecidas[i] = True
                    ciclo.append(t)
                elif (grafo.getArestasNaoDirigido()[i][1] == t) :
                    t = grafo.getArestasNaoDirigido()[i][0]
                    conhecidas[i] = True
                    ciclo.append(t)
                else:
                    cont2 += 1
            else:
                cont1 += 1
        if ((cont1 + cont2) == grafo.qtdArestas()) :
            return (False, None)
        if (t == v) :
            break
```

Enquanto houver alguma aresta não visitada no grafo, devemos encontrar um vértice no ciclo atual que ainda tenha uma aresta não visitada. Com isso, o código abaixo percorre a lista “ciclo” e, caso encontre tal vértice, realiza uma recursão para encontrar um subciclo e inseri-lo, se ele existir, no lugar da posição de x em “ciclo”.

```
for i in range(grafo.qtdArestas()) :
    if (conhecidas[i] == False) :
        for x in range(len(ciclo)) :
            if (grafo.getArestasNaoDirigido()[i][0] == ciclo[x] or grafo.getArestasNaoDirigido()[i][1] == ciclo
            [x]) :
                resposta, ciclo2 = Hierholzer.__encontrar_subciclo_euleriano(grafo, ciclo[x], conhecidas)
                if (resposta == True) :
                    ciclo[x:x + 1] = ciclo2
return (True, ciclo)
```

Escolhemos o algoritmo de Hierholzer pois ele é eficiente para encontrar Ciclos Eulerianos em grafos grandes e é relativamente simples de implementar, com etapas básicas como a identificação de vértices de grau ímpar e a construção de Subciclos. Além disso, o algoritmo funciona em qualquer grafo que possua um Ciclo Euleriano e garante que o circuito encontrado seja válido, passando por todas as arestas do grafo exatamente uma vez.

## 4. ALGORITMO DE BELLMAN-FORD

Para informar o caminho de um vértice inicial até todos os outros vértices do grafo, bem como a distância necessária para percorrer esse caminho, optamos por utilizar o Algoritmo de Bellman-Ford, que aparece na imagem a seguir.

**Algoritmo 10:** Algoritmo de Bellman-Ford.

**Input** :um grafo $G = (V, E, w)$, um vértice de origem $s \in V$

`// inicialização`

$D_v \leftarrow \infty \; \forall v \in V$

$A_v \leftarrow$ **null** $\forall v \in V$

$D_s \leftarrow 0$

**for** $i \leftarrow 1$ ***to*** $|V| - 1$ **do**

    **foreach** $(u, v) \in E$ **do**

        `// relaxamento`

        **if** $D_v > D_u + w\big((u, v)\big)$ **then**

            $D_v \leftarrow D_u + w\big((u, v)\big)$

            $A_v \leftarrow u$

**foreach** $(u, v) \in E$ **do**

    **if** $D_v > D_u + w\big((u, v)\big)$ **then**

        **return** (**false**,**null**,**null**)

**return** (**true**, $D$, $A$)

Fonte: SANTIAGO, 2023, p. 50

O algoritmo abaixo começa definindo um nó de origem e inicializando a distância para si mesmo como zero e a distância para todos os outros nós como infinito. Em seguida, para cada aresta no grafo, o algoritmo relaxa a aresta atualizando as distâncias do nó inicial para o nó final se a soma das distâncias até o nó inicial e o peso da aresta atual for menor do que a distância atual do nó final.

Esse processo é repetido para cada nó no grafo, garantindo que todas as arestas tenham sido relaxadas corretamente. O algoritmo então verifica se existe um ciclo negativo no grafo, verificando se ainda é possível relaxar uma aresta. Se for possível relaxar uma aresta, então existe um ciclo negativo no grafo. Por fim, o algoritmo retorna as distâncias mais curtas encontradas do nó inicial para todos os outros nós. A verificação de ciclo negativo é importante para garantir que a solução do problema seja válida.

```python
@staticmethod
def BellmanFord(grafo: Grafo, s: int) -> tuple:
    distancia = [float('inf') for i in range(grafo.qtdVertices())]
    antecessor = [None for i in range(grafo.qtdVertices())]

    distancia[s - 1] = 0

    for _ in range(1, grafo.qtdVertices() - 1):
        for u,v in grafo.getArestas():
            #relaxamento
            if distancia[v-1] > distancia[u-1] + grafo.peso(u,v):
                distancia[v-1] = distancia[u-1] + grafo.peso(u,v)
                antecessor[v-1] = u

    #verifica se ha ciclo negativo
    for _ in range(1, grafo.qtdVertices() - 1):
        for u,v in grafo.getArestas():
            if distancia[v-1] > distancia[u-1] + grafo.peso(u,v):
                return (False,None,None)

    return (True, distancia, antecessor)
```

Escolhemos o Algoritmo de Bellman-Ford porque ele funciona em grafos ponderados com pesos negativos, tornando-o útil quando o peso das arestas é negativo, o que o algoritmo de Dijkstra não é capaz de lidar. Além disso, o algoritmo de Bellman-Ford pode detectar ciclos negativos no grafo, que podem levar a uma solução inválida.

## 5. ALGORITMO DE FLOYD-WARSHALL

Utilizamos o algoritmo de Floyd-Warshall para encontrar os caminhos mais curtos entre todos os pares de nós em um grafo ponderado. Abaixo, está o pseudocódigo utilizado como base para a implementação.

**Algoritmo 12:** Algoritmo de Floyd-Warshall.

**Input** :um grafo $G = (V, E, w)$

$D^{(0)} \leftarrow W\left(G\right)$

// Assumindo que os vértices estão rotulados de $1, 2, \ldots, |V|$

**foreach** $k \in \{1, 2, \ldots, |V|\}$ **do**

  seja $D^{(k)} = \left(d_{uv}^{(k)}\right)$ uma nova matriz $|V| \times |V|$

  **foreach** $u \in V$ **do**

    **foreach** $v \in V$ **do**

      $d_{uv}^{(k)} \leftarrow \min\left\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\right\}$

**return** $D^{(|V|)}$

Fonte: SANTIAGO, 2023, p. 57

O algoritmo começa inicializando uma matriz de distâncias, em que cada entrada (i, j) representa a distância do nó “i” ao nó “j”. Em seguida, o algoritmo usa um processo iterativo para atualizar a matriz de distâncias, comparando a distância direta entre cada par de nós com a distância através de outros nós intermediários.

Em cada iteração, o algoritmo considera um nó intermediário “k” e atualiza a distância entre os nós “i” e “j” como a soma da distância entre “i” e “k” e a distância entre “k” e “j”. Ao final das iterações, a matriz de distâncias contém as distâncias mais curtas entre todos os pares de nós no grafo.

```python
@staticmethod
def FloydWarshall(grafo: Grafo) -> list:
    number_of_vertices = grafo.qtdVertices()

    distance = [([float('inf') for j in range(number_of_vertices)]) for i in range(number_of_vertices)]   #Cria a
    matriz de distancias
    for i in range(0, number_of_vertices):
        distance[i][i] = 0
    for u,v in grafo.getArestas():
        distance[u-1][v-1] = grafo.peso(u,v)

    for k in range(1, number_of_vertices):
        for j in range(1, number_of_vertices):
            for i in range(1, number_of_vertices):
                if distance[i-1][j-1] > distance[i-1][k-1] + distance[k-1][j-1]:
                    distance[i-1][j-1] = distance[i-1][k-1] + distance[k-1][j-1]

    return distance
```

Este algoritmo foi escolhido porque ele garante encontrar o caminho mais curto entre todos os pares de nós em um grafo, desde que exista um caminho entre eles. Mesmo que tenha uma complexidade alta, é uma boa opção para grafos pequenos, com até algumas centenas de vértices, e pode ser mais rápido que outros algoritmos, como o Dijkstra, nesse cenário.

## REFERÊNCIAS BIBLIOGRÁFICAS

SANTIAGO, Rafael de. Anotações para a Disciplina de Grafos - versão de 27 de março de 2023. Universidade Federal de Santa Catarina, 2023.

*O restante das imagens deste relatório pertencem aos autores.