# Relatório terceiro trabalho ED 2

O presente trabalho consiste em implementar a estrutura de dados **Grafo** e realizar as seguintes operações dentro do programa:

- Leitura do grafo por um arquivo.
- Realizar a busca em largura e retornar quantas arestas há de um vértice a outro.
- Imprimir a lista de adjacências na ordem inversa de carregamento.
- Calcular a excentricidade de um vértice.
- Calcular o diâmetro do grafo.
- Calcular o raio do grafo.
- Exibir a MTS do grafo pelo Algoritmo de Prim.
- Exibir a MTS do grafo pelo Algoritmo de Kruskal.
- Retornar os vértices com distancia menor que x de determinado vértice.

## Arquivos

Os arquivos utilizados foram arquivos pequenos de 5 a 7 vértices apenas para testar se os algoritmos estão funcionando, não foram realizados testes com arquivos grandes.

## Estruturas Implementadas

Foram realizadas duas implementações de grafos, uma com lista de adjacências e outra com matriz de adjacências, porém alguns algoritmos foram implementados apenas na implementações com lista de adjacências.

### Matriz de Adjacências

A implementação com matriz de adjacências foi realizado implementando apenas uma classe, a classe Graph, dentro desta classe foi implementado o atributo *adjMatrix* onde é atribuída a matriz de adjacências, também possui uma flag *oriented* para saber de o grafo é orientado ou não, um atributo *size* para a quantidade de vértices, e vetores para variáveis dos vértices como antecessores (*pi*), distancia (*dist*), cor, etc.

```
class Graph(object):
 def __init__(self, size, oriented = "not_oriented"):
  self.oriented = oriented
  self.adjMatrix = [[0 for x in range(size)] for y in range(size)]
  for i in range(size):
   for j in range(size):
    self.adjMatrix[i][j] = None
  self.size = size

  #define predecessor
  self.pi = []
  for i in range(size):
   self.pi.append(None)


  self.alfa = []
  for i in range(size):
   self.alfa.append(None)

  self.dist = []
  for i in range(size):
   self.dist.append(None)


  self.cor = []
  for i in range(size):
   self.dist.append("branco")
```

### Lista de Adjacências

A implementação com lista de adjacências foi realizado implementando usando três classes, a classe Graph que possui uma lista com os vértices (*self.vertex*), também possue uma flag *oriented* para saber de o grafo é orientado ou não, um atributo *size* para a quantidade de vértices.

```
class Graph(object):
 def __init__(self, size, oriented = "not_oriented"):
  self.oriented = oriented
  self.vertex = []
  for i in range(size):
   self.vertex.append(Vertex(num=i))
  self.size = size
```

Dentro da lista *self.vertex* há objetos do tipo *Vertex*, essa classe possui a lista de adjacências do vértice além de atributos como predecessor, cor, distância que são utilizados no algoritmos implementados.

```
class Vertex(object):
 def __init__(self, num):
  self.num = num
  self.listAdj = []
  self.listWeight = []
  self.pi = None
  self.alfa = None
  self.dist = None
  self.cor = "branco"
```

Também foi implementado a classe EDGE, porém ela é utilizada apenas nos algoritmos de Prim, para utilização do Heap da biblioteca padrão do Python e no Kruskal para utilização do método list.sort() que implementa o quicksort.

```
class EDGE(object):
 def __init__(self, weight, u, v):
  self.weight = weight
  self.u = u
  self.v = v
```

## Algoritmos Implementados

Para o presente trabalho foram implementados os seguintes Algoritmos:

- Busca em Largura
- Prim
- Kruskal
- Dijkstra

### Busca em largura

A busca em largura foi implementada segundo os pseudo-códigos do slide, pega-se um vértice inicial coloca-se os vértices adjacentes ao vértice atual, que não foram descobertos na fila, definindo o parâmetro *dist* destes vértices para a distancia do vértice atual +1, e colocando como predecessor o vértice atual, pinta o vértice atual de preto, define como vértice atual o próximo da fila e repita as operações enquanto a fila não estiver vazia.

```
def BFS(self, u = None):
 self.resetGraph()
 if u is None:
  u = self.vertex[0]
 else:
  u = self.vertex[u]
 queue = []
 #dist, Vertex.
 queue.append([0,u])
 while len(queue) > 0:
  #print(queue)
  count, u = queue.pop()
  EDGEs = u.getEDGEs()
  for EDGE in EDGEs:
   if EDGE.cor == "branco":
    queue.append([count+1,EDGE])
    EDGE.cor = "cinza"
    EDGE.dist = count+1
    EDGE.pi = u
  u.cor = "preto"
```

### Prim

O algoritmo de Prim também foi implementado de forma parecida como a encontrada no slide, as arestas foram convertias em objeto para utilizar o heap já implementado na biblioteca padrão do Python.
Para representar a MTS foi criado um grafo com o tamanho do original, porém com apenas as arestas presentes na arvores geradora minima. A MTS é guardada dentro da classe Graph() com um atributo *self.mts*.

```
def Prim(self, u):
 self.resetGraph()
 self.MTS = Graph(self.size)
 u = self.vertex[u]
 init = u
 components = []
 components.append(u)
 u.cor = "preto"
 EDGEs = u.getEDGEsObj()
 while len(EDGEs) > 0:
  EDGEs = self.SecurityEDGEsPrim(EDGEs)
  heapq.heapify(EDGEs)
  EDGE = heapq.heappop(EDGEs)
  self.MTS.addEDGE(EDGE.u.num, EDGE.v.num, weight= EDGE.weight)
  if EDGE.u.cor != "preto":
   components.append(EDGE.u)
   EDGE.u.cor = "preto"
  if EDGE.v.cor != "preto":
   EDGE.v.cor = "preto"
   components.append(EDGE.v)
  EDGEs = []
  for x in components:
   EDGEs.extend(x.getEDGEsObj())
  EDGEs = self.SecurityEDGEsPrim(EDGEs)
```

## Kruskal

O algoritmo de Kruskal também foi implementado de forma parecida como a encontrada no slide, as arestas foram convertias em objeto para utilizar o o método list.sort() já implementado na biblioteca padrão do Python.
Para representar a MTS foi criado um grafo com o tamanho do original, porém com apenas as arestas presentes na arvores geradora minima. A MTS é guardada dentro da classe Graph() com um atributo *self.mts*.

```
def Kruskal(self):
 self.resetGraph()
 self.MTS = Graph(self.size)
 components = []
 for x in range(self.size):
  components.insert(x, [])
  components[x].append(self.vertex[x])
  self.vertex[x].alfa = x
 EDGEs = []
 for vertex in self.vertex:
  EDGEs.extend(vertex.getEDGEsObj())
 self.RemoveDuplicate(EDGEs)
 EDGEs.sort()
 while len(EDGEs) > 0:
  Edge = EDGEs.pop(0)
  if self.SecurityEDGEKruskal(Edge.u, Edge.v):
   self.MergeComponents(components, Edge.u, Edge.v)
   self.MTS.addEDGE(Edge.u.num, Edge.v.num, weight= Edge.weight)
```

### Dijkstra

O algoritmo de Dijkstra também foi implementado de forma parecida como a encontrada no slide. Os vértices predecessores foram referenciados no atributo *pi* de cada vértice, e a distancia de todos os vértices é inicializado com o máximo que a variável do tipo *int* suporta, e posteriormente atualizado com as iterações do algoritmo.

```
def Relax(self, u):
 count = 0
 for EDGE in u.listAdj:
  if EDGE.dist > u.dist + u.listWeight[count]:
   EDGE.dist = u.dist + u.listWeight[count]
   EDGE.pi = u
  count += 1
```

```
def Dijkstra(self, u):
 self.resetGraph()
 for x in self.vertex:
  x.dist = sys.maxsize
 u = self.vertex[u]
 u.dist = 0
 EDGEs = []
 EDGEs.append(u)
 while len(EDGEs) > 0:
  EDGEs.sort()
  u = EDGEs.pop(0)
  EDGEs.extend(u.getEDGEs())
  u.cor = "preto"
  self.Relax(u)
  EDGEs = self.SecurityVertexsDijkstra(EDGEs)
  EDGEs = self.RemoveDuplicate(EDGEs)
```

## Métodos Implementados

Os seguintes métodos foram implementados com base nos algoritmos acima:

- Graph.BFS_orig_dest(u, v)
- Graph.eccentricity(u)
- Graph.Diameter()
- Graph.Radius()
- Graph.DistMenor(u, dist)

### Graph.BFS_orig_dest(u, v)

O método executa a busca em largura no vértice u, e retorna o parâmetro *dist* do vértice v.

```
def BFS_orig_dest(self, u, v):
 self.BFS(u)
 vertDest = self.vertex[v]
 return vertDest.dist
```

### Graph.eccentricity(u)

O método realiza a busca em largura a partir do vértice u, e retorna o maior parâmetro *dist* atribuído pela busca em largura.

```
def eccentricity(self, u = None):
 self.resetGraph()
 if u is None:
  u = self.vertex[0]
 elif u is int:
  u = self.vertex[u]
 max = [0, None]
 queue = []
 queue.append([0,u])
 while len(queue) > 0:
  count, u = queue.pop()
  EDGEs = u.getEDGEs()
  for EDGE in EDGEs:
   if EDGE.cor == "branco":
    queue.append([count+1,EDGE])
    EDGE.cor = "cinza"
    EDGE.dist = count+1
    EDGE.pi = u
    if max[0] <= count+1:
     max = [count+1, EDGE]
  u.cor = "preto"
 return max[0]
```

### Graph.Diameter()

O método executa o método *Graph.eccentricity(u)* a partir de todos os vértices do grafo, e salva o **maior** valor no em uma variável, ao fim da execução retorna o valor da variável. O definição de diâmetro do grafo foi obtida através de [1] e [2].

```
def Diameter(self):
 maxEcc = self.eccentricity(u = self.vertex[0])
 for EDGE in self.vertex[1:]:
  x = self.eccentricity(u = EDGE)
  if x > maxEcc:
   maxEcc = x
 return maxEcc
```

### Graph.Radius()

O método executa o método *Graph.eccentricity(u)* a partir de todos os vértices do grafo, e salva o **menor** valor no em uma variável, ao fim da execução retorna o valor da variável. O definição de raio do grafo foi obtida através de [1] e [2].

```
def Radius(self):
 minEcc = self.eccentricity(u = self.vertex[0])
 for EDGE in self.vertex[1:]:
  x = self.eccentricity(u = EDGE)
  if x < minEcc:
   minEcc = x
 return minEcc
```

### Graph.DistMenor(u, dist)

O método executa o algoritmo de Dijkstra a partir do vértice u, e adiciona em uma lista dos os vértices que possuem o parâmetro *dist* menor que a distancia passada por parâmetro no método, e retornando a lista.

```
def DistMenor(self, orig, dist):
 lista = []
 self.Dijkstra(orig)
 for vert in self.vertex:
  if vert.dist <= dist:
   lista.append(vert.num)
 return lista
```

## Resultados

Foi realizado os seguintes testes:

### Grafo Carregado

O seguinte grafo foi carregado

```
5
1 2 3.65
2 4 1.43
2 3 7.98
3 4 6.87
4 0 9.12
```

### Opções do menu

```
=====MENU=====
0 - SAIR
1 - BUSCA EM LARGURA (NUMERO DE VERTICES DA ORIGEM AO DESTINO)
2 - IMPRIMIR GRAFO AO INVERSO DA ORDEM DE CARREGAMENTO
3 - CALCULA A EXCENTRICIDADE DE UM VÉRTICE V
4 - CALCULA O DIÂMETRO DO GRAFO
5 - CALCULA O RAIO DO GRAFO
6 - MST POR PRIM
7 - MST POR KRUSKAL
8 - VERTICES COM DISTANCIA MENOR QUE X
9 - PRINTAR GRAFO
```

## 1 - BUSCA EM LARGURA

Do vértice 1 até o 0
O resultado:

```
Vertice de origem:1
Vertice de destino:0
A quantidade de arestas do vertice  1 até o vertice  0  é  3
```

## 2 - IMPRIMIR GRAFO AO INVERSO DA ORDEM DE CARREGAMENTO

O resultado para grafo não orientado:

```
0: 4
1: 2
2: 3 4 1
3: 4 2
4: 0 3 2
```

O resultado para grafo orientado:

```
0:
1: 2
2: 3 4
3: 4
4: 0
```

## 3 - CALCULA A EXCENTRICIDADE DE UM VÉRTICE V

A excentricidade do vértice 2
O resultado grafo não orientado:

```
Vertice escolhido:2
A excentricidade do vertice  2  é  2
```

O resultado grafo orientado:

```
Vertice escolhido:2
A excentricidade do vertice  2  é  2
```

## 4 - CALCULA O DIÂMETRO DO GRAFO

O resultado grafo não orientado:

```
4
O diametro do grafo é  3
```

O resultado grafo orientado:

```
4
O diametro do grafo é  3
```

## 5 - CALCULA O RAIO DO GRAFO

O resultado grafo não orientado:

```
5
O raio do grafo é  2
```

O resultado grafo orientado:

```
5
O raio do grafo é  0
```

## 6 - MST POR PRIM

Resultado

```
Vertice de origem:2
MST do grafo por Prim, iniciando pelo vertice  2
[
0 :
  4, 9.12000
1 :
  2, 3.65000
2 :
  4, 1.43000
  1, 3.65000
3 :
  4, 6.87000
4 :
  2, 1.43000
  3, 6.87000
  0, 9.12000
]
```

## 7 - MST POR KRUSKAL

Resultado

```
7
MST do grafo por Kruskal
[
0 :
  4, 9.12000
1 :
  2, 3.65000
2 :
  4, 1.43000
  1, 3.65000
3 :
  4, 6.87000
4 :
  2, 1.43000
  3, 6.87000
  0, 9.12000
]
```

## 8 - VÉRTICES COM DISTANCIA MENOR QUE X

Resultado

```
8
Vertice de origem:0
Distancia maxima:12
Os vertices com distancia menor que 12.0 do vertice 0 são:
[0, 2, 4]
```

## 9 - PRINTAR GRAFO

O resultado grafo não orientado:

```
9
[
0 :
  4, 9.12000
1 :
  2, 3.65000
2 :
  1, 3.65000
  4, 1.43000
  3, 7.98000
3 :
  2, 7.98000
  4, 6.87000
4 :
  2, 1.43000
  3, 6.87000
  0, 9.12000
]
```

O resultado grafo orientado:

```
9
[
0 :
1 :
  2, 3.65000
2 :
  4, 1.43000
  3, 7.98000
3 :
  4, 6.87000
4 :
  0, 9.12000
]
```

## Referencias


[1] . Graph measurements: length, distance, diameter, eccentricity, radius, center, disponível em: *https://www.geeksforgeeks.org/graph-measurements-length-distance-diameter-eccentricity-radius-center/*
[2]. Grafos - Departamento de Computação e Matemática - USP, disponível em: *http://dcm.ffclrp.usp.br/~augusto/teaching/aedii/AED-II-Grafos.pdf*