

# Implementação de biblioteca de grafo não direcionado ponderado

Ana C.O. Soares(5896)<sup>1</sup>, Maria E.D. Lacerda(5920)<sup>1</sup>,  
Pedro J. Miranda(4912)<sup>1</sup>, Rafael R. Soares(5891)<sup>1</sup>

<sup>1</sup>Instituto de Ciências Exatas– Universidade Federal de Viçosa - Campus Florestal (UFV)

{ana.o.soares, maria.eduarda.lacerda, pedro.d.miranda, rafael.resende}@ufv.br

**Resumo.** Este trabalho é parte avaliativa da disciplina Teoria e Modelo de Grafos(CCF 331) e tem como objetivo implementar uma biblioteca para a criação e manipulação de grafos para um contexto de uma empresa de transportes. A implementação utiliza o conceito de grafo não direcionado ponderado, sendo as cidades representadas pelos vértices e as estradas entre elas pelas arestas e o peso representa a distância em quilômetros entre as duas cidades. A biblioteca fornece uma gama de funções extremamente úteis para o contexto.

**Abstract.** This work is part of the course assessment for Graph Theory and Models (CCF 331) and aims to implement a library for the creation and manipulation of graphs in the context of a transportation company. The implementation uses the concept of a weighted, undirected graph, where cities are represented by vertices and the roads between them by edges; the weight represents the distance in kilometers between the two cities. The library provides a range of extremely useful functions for this context.

## Contents

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Desenvolvimento</b>	<b>3</b>
2.1	Organização . . . . .	3
2.2	Estrutura . . . . .	3
2.3	Operações . . . . .	5
2.3.1	Básicas . . . . .	5
2.3.2	Menor caminho (Dijkstra) . . . . .	7
2.3.3	Determinar pontos de articulação . . . . .	9
2.3.4	Identificação de ciclos com 4 ou mais vértices . . . . .	11
<b>3</b>	<b>Conclusão</b>	<b>11</b>

## 1. Introdução

O trabalho foi feito utilizando a linguagem de programação Python e a estrutura de representação computacional para grafos denominada "Lista de Adjacência". A implementação das funções especificadas foram feitas seguindo os conceitos e algoritmos apresentados na disciplina.

## 2. Desenvolvimento

### 2.1. Organização

A organização do trabalho foi feita a partir da criação de um repositório na plataforma GitHub[1], onde todos os integrantes do grupo puderam contribuir para a realização do trabalho prático. Além disso, houve uma divisão das tarefas, avaliando a dificuldade de cada uma, para que nenhum integrante tivesse uma sobrecarga de trabalho, a organização final ficou da seguinte forma:

- Retornar o número de cidades no grafo (Ana)
- Retornar a quantidade de estradas no grafo (Ana)
- Retornar os vizinhos de uma cidade fornecida (Ana)
- Determinar a quantidade de vizinhos de uma cidade fornecida (Rafael)
- Calcular o menor caminho entre duas cidades escolhidas. (Rafael)
- Verificar se a rede é conexa. (Rafael)
- Identificar cidades críticas cuja remoção desconectaria a rede (Ana)
- Indicar se a empresa consegue criar um passeio turístico com a seguinte característica: passeio turístico circular que passa por pelo menos 4 cidades diferentes e retorne à cidade de origem sem repetir estradas. (Pedro)
- Se o passeio do item anterior existir, forneça um exemplo de qual passeio seria (Pedro)
- Implementação da estrutura da Lista de Adjacência e Menu interativo. (Maria Eduarda)

### 2.2. Estrutura

A implementação da biblioteca foi iniciada a partir da criação de uma classe para representar uma lista encadeada, que na lista de adjacência representa quais vértices possuem ligação com uma cidade específica. Foi criada uma classe para representar os nós da lista encadeada e neles possuem a identificação da cidade e o peso daquela aresta, além de um campo para armazenar qual o próximo nó da lista. A figura 1 mostra a implementação da classe mencionada acima.



```
1 class No:
2     def __init__(self, dado, peso) -> None:
3         self.dado = dado
4         self.proximo = None
5         self.peso = peso
6
```

Figure 1. Implementação da Classe Nó

A figura 2 mostra a implementação da classe "ListaEncadeada" que possui apenas três métodos, sendo um deles o construtor. A referência utilizada para a criação do código foi [Nunes 2021].

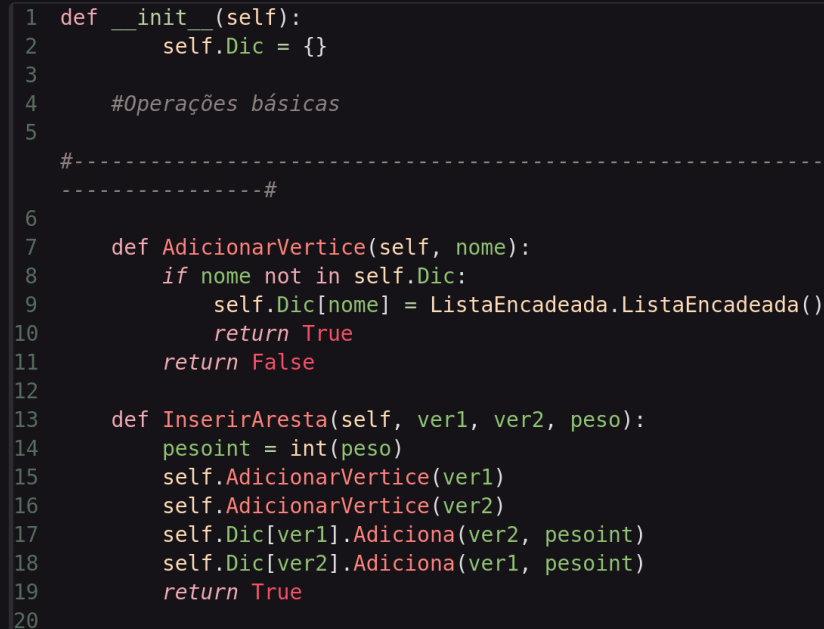


```
1 class ListaEncadeada:
2
3     def __init__(self) -> None:
4         #Cria uma lista encadeada vazia
5         self.cabeca = None
6
7     def Adiciona(self, dado, peso) -> None:
8
9         #Esta funcao sempre ira adicionar os elementos no final da lista
10        novo = No(dado, peso)
11        if self.cabeca == None: #Verifica se a lista esta vazia
12            novo.proximo = self.cabeca
13        #Como ela vai estar vazia o proximo é None
14        self.cabeca = novo
15    else:
16        noatual = self.cabeca
17        while noatual.proximo != None:
18            #Encontra o último elemento da lista
19            noatual = noatual.proximo
20
21        noatual.proximo = novo
22        novo.proximo = None
23
24    def ImprimiLista(self):
25        #Percorre a lista encadeada para realizar a impressão
26        current = self.cabeca
27        while current:
28            print(f"-> [{current.dado}-{current.peso}]", end=" ")
29            current = current.proximo
30        print()
```

**Figure 2. Implementação da Classe Lista Encadeada**

Para representar a Lista de adjacência como um todo a estrutura escolhida foi o Dicionário. Esta é uma estrutura que guarda dados em pares de chave e valor e utiliza estas chaves para encontrar os elementos associados a elas, então para esta implementação a chave do dicionário foi inicializada como o nome da cidade e o valor a lista encadeada contendo todas as ligações que está cidade possui.

As operações básicas para a inicialização do grafo são o método construtor, um método para inicialização de vértices e outro para inicialização de arestas. São métodos que seguem a lógica para a utilização da estrutura do dicionário, além de utilizar os métodos da classe Lista Encadeada. A figura 3 mostra o código desta parte essencial para o funcionamento correto da biblioteca.



```

1 def __init__(self):
2     self.Dic = {}
3
4     #Operações básicas
5
6     #-----#
7
8     def AdicionarVertice(self, nome):
9         if nome not in self.Dic:
10             self.Dic[nome] = ListaEncadeada.ListaEncadeada()
11             return True
12         return False
13
14     def InserirAresta(self, ver1, ver2, peso):
15         peso = int(peso)
16         self.AdicionarVertice(ver1)
17         self.AdicionarVertice(ver2)
18         self.Dic[ver1].Adiciona(ver2, peso)
19         self.Dic[ver2].Adiciona(ver1, peso)
20         return True

```

**Figure 3. Implementação da Classe Lista Adjacência**

Um ponto importante é que utilizamos a biblioteca NextworkX para criarmos uma representação visual do grafo inserido, sendo esta uma função adicional do trabalho. No README possui como instalar esta biblioteca caso seja necessário. Para aprendermos como utilizar esta biblioteca utilizamos este artigo [Nex ], além disso foram utilizadas outras fontes de estudos como [Cormen et al. 2012], [Choudhary 2022].

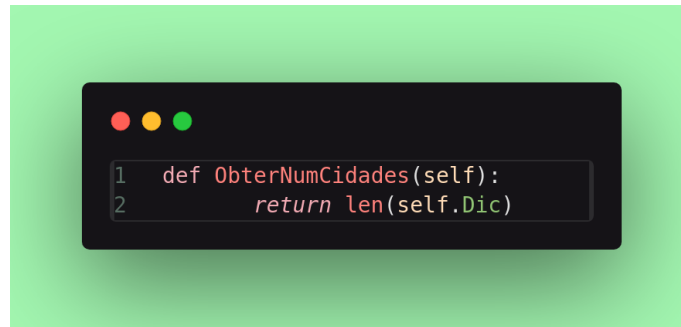
## 2.3. Operações

### 2.3.1. Básicas

Como identificado anteriormente, algumas das operações pedidas no trabalho foram entendidas como simples, o que levou a implementação dessas na própria classe da lista de adjacência. Sendo elas:

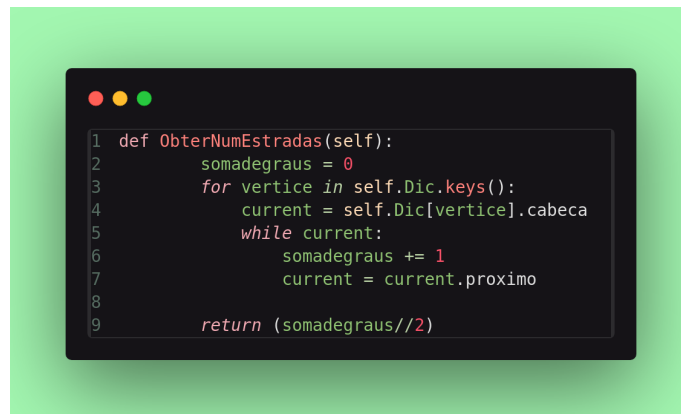
- 1 - Retornar o número de cidades no grafo
- 2 - Retornar a quantidade de estradas no grafo
- 3 - Retornar os vizinhos de uma cidade fornecida
- 4 - Determinar a quantidade de vizinhos de uma cidade fornecida

Para a primeira operação básica temos uma lógica bem simples utilizando a função len() para retornar o tamanho do dicionário, ou seja, o número de vértices inseridos no dicionário. A figura 4 mostra o código deste método.




**Figure 4. Operação básica 1**

A segunda operação consistia em retornar a quantidade de estradas no grafo, para isso foi utilizado o conceito do somatório de graus ser duas vezes o número de arestas do grafo. Logo é um algoritmo bem simples apenas para percorrer a lista encadeada de cada vértice realizando a soma. A figura 5 mostra como ficou o código deste método.



**Figure 5. Operação básica 2**

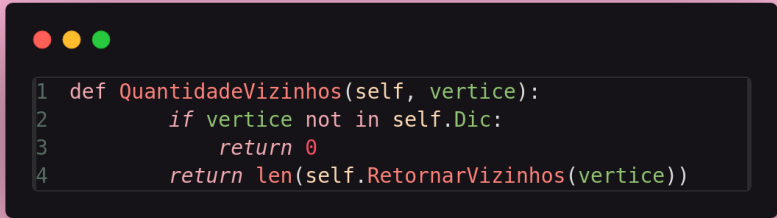
A terceira operação básica consistia em retornar os vizinhos de uma cidade específica. Esta operação foi realizada apenas percorrendo a lista encadeada e armazenando os vértices em um vetor para retornar. A figura 6 mostra como ficou o código deste método.



```
1 def RetornarVizinhos(self, vertice):
2     lista = []
3     current = self.Dic[vertice].cabeca
4     while current:
5         lista.append(current.dado)
6         current = current.proximo
7     return lista
```

**Figure 6. Operação básica 3**

A quarta e última operação básica consiste em contar quantos vizinhos uma cidade possui. Este método foi implementado a partir do anterior e basicamente utiliza a função `len()` para retornar o tamanho do vetor gerado pelo método "RetornaVizinhos". A figura 7 mostra como ficou o código deste método.



```
1 def QuantidadeVizinhos(self, vertice):
2     if vertice not in self.Dic:
3         return 0
4     return len(self.RetornarVizinhos(vertice))
```

**Figure 7. Operação básica 4**

### **2.3.2. Menor caminho (Dijkstra)**

O problema do menor caminho em grafos ponderados com pesos não-negativos consiste em determinar a rota de menor custo entre dois vértices.

O algoritmo de Dijkstra[Dijkstra 1959] é um dos métodos mais utilizados para esse problema. Ele se baseia em uma estratégia de programação gulosa, expandindo iterativamente o vértice com menor distância conhecida até que todos os caminhos mínimos sejam encontrados.

**Inicialização:**

```

1  def MenorCaminho(self, origem, destino):
2
3      #Retorna a distância mínima e o caminho
4      if origem not in self.Dic or destino not in self.Dic:
5          return None, []
6
7      dist = {v: float('inf') for v in self.Dic}
8      anterior = {v: None for v in self.Dic}
9      dist[origem] = 0
10
11     heap = [(0, origem)]
12     while heap:
13         d, u = heapq.heappop(heap)
14         if d > dist[u]:
15             continue
16         if u == destino:
17             break
18
19         atual = self.Dic[u].cabeca
20         while atual:
21             v = atual.dado
22             peso = atual.peso
23             if dist[u] + peso < dist[v]:
24                 dist[v] = dist[u] + peso
25                 anterior[v] = u
26                 heapq.heappush(heap, (dist[v], v))
27             atual = atual.proximo
28
29     if dist[destino] == float('inf'):
30         return None, []

```

Figure 8. Algoritmo do menor caminho.

### Algoritmo de Menor Caminho (Dijkstra)[1]

O problema do menor caminho em grafos ponderados é de grande importância em diversas áreas, como logística, telecomunicações e análise de redes. Uma das soluções mais conhecidas para esse problema é o algoritmo proposto por Edsger W. Dijkstra em 1959. O método busca determinar a menor distância entre dois vértices de um grafo, desde que os pesos das arestas sejam não negativos.

O algoritmo inicia atribuindo a cada vértice uma distância inicial infinita, com exceção da origem, que recebe valor zero. Em seguida, seleciona iterativamente o vértice com a menor distância conhecida e, a partir dele, atualiza as distâncias de todos os seus vizinhos. Caso o caminho passando pelo vértice atual ofereça uma rota mais curta até um vizinho, a distância desse vizinho é atualizada e o vértice atual é registrado como seu predecessor. Esse processo continua até que todos os vértices tenham sido explorados ou até que o destino seja alcançado.

Ao final da execução, o algoritmo fornece não apenas a menor distância entre a origem e o destino, mas também o caminho percorrido, que pode ser reconstruído a partir da lista de predecessores. Por sua natureza gulosa e uso de estruturas de dados eficientes,



como filas de prioridade, o algoritmo de Dijkstra é considerado bastante eficiente para grafos de médio e grande porte.

### 2.3.3. Determinar pontos de articulação

The image shows a code editor window with a dark background and light-colored text. The code is a Python function named 'EncontraArticulacoes' that iterates through the vertices of a graph and performs a DFS to find articulation points. The code is as follows:

```
1 def EncontraArticulacoes(self):
2     for vertice in self.grafo.Dic.keys():
3         self.visitado[vertice] = False
4         self.baixo[vertice] = float('inf')
5         self.pai[vertice] = None
6
7     inicio = 0
8     for vertice in self.grafo.Dic.keys():
9         if not self.visitado[vertice]:
10            print(f"Iniciando o subgrafo conexo em {vertice}")
11            self._dfs(vertice)
12            if inicio != 0:
13                print("Grafo desconexo")
14            inicio += 1
15
16     return self.articulacoes
```

Figure 9. Algoritmo para encontrar articulações.

#### Algoritmo de Pontos de Articulação

Outra questão relevante no estudo de grafos diz respeito à robustez de uma rede. Em particular, é importante identificar os vértices cuja remoção compromete a conectividade do sistema. Esses vértices são chamados de pontos de articulação. Um algoritmo eficiente para esse problema foi desenvolvido por Robert Tarjan em 1972[Tarjan 1972], baseado na técnica de busca em profundidade (DFS)[Soares 2021].

```

1  def _dfs(self, u):
2      self.visitado[u] = True
3      self.baixo[u] = self.tempo
4      disc = self.tempo
5      self.tempo += 1
6
7      filhos = 0
8
9      current = self.grafo.Dic[u].cabeca
10
11     while current:
12         v = current.dado
13
14         if not self.visitado[v]:
15             self.pai[v] = u
16             filhos += 1
17             self._dfs(v)
18
19             self.baixo[u] = min(self.baixo[u], self.baixo[v])
20
21             if self.pai[u] is None and filhos > 1:
22                 self.articulacoes.add(u)
23
24             if self.pai[u] is not None and self.baixo[v] >= disc:
25                 self.articulacoes.add(u)
26
27         elif v != self.pai[u]:
28             self.baixo[u] = min(self.baixo[u], self.baixo[v])
29
30         current = current.proximo

```

**Figure 10. Busca em Profundidade.**

O método consiste em percorrer o grafo atribuindo a cada vértice um tempo de descoberta, que corresponde ao instante em que foi visitado na busca. Além disso, calcula-se para cada vértice um valor denominado “baixo”, que indica o vértice mais antigo que pode ser alcançado a partir dele, seja por caminhos diretos ou por arestas de retorno.

Com essas informações, é possível determinar as condições que caracterizam um ponto de articulação. Um vértice é classificado dessa forma se for a raiz da busca e possuir mais de um filho, ou se não for raiz e existir pelo menos um filho cuja subárvore não possua conexão alternativa com os ancestrais desse vértice. Em ambos os casos, a remoção do vértice em questão provocaria a divisão do grafo em mais de uma componente conexa.

O algoritmo de Tarjan percorre cada vértice e aresta exatamente uma vez, garantindo complexidade linear em relação ao tamanho do grafo. Dessa forma, ele se mostra uma solução prática e eficiente para a análise estrutural de redes complexas, sendo amplamente empregado em áreas como engenharia de redes, biologia computacional e segurança de sistemas.

### 2.3.4. Identificação de ciclos com 4 ou mais vértices

```
1 def CalculaCiclos(self, v, pai, caminho, inicio, ciclos):
2     caminho.append(v) #guarda um vertice em um caminho atual
3
4     atual = self.Dic[v].cabeca
5     while atual:
6         vizinho = atual.dado
7         if vizinho not in caminho:
8             self.CalculaCiclos(vizinho, v, caminho, inicio, ciclos)
9         elif vizinho != pai and vizinho == inicio and len(caminho) >= 4:
10            # se a proxima aresta nao é o pai, nao ter uma caminho de volta
11            # Para ver se o proximo vertice é o primeiro vertice do caminho confirma o ciclo
12            # Se o caminho tem mais de 4 arestas
13            ciclo = caminho + [vizinho]
14            if len(ciclos) >= NUMCAMINHOS:
15                return ciclos
16            if ciclo not in ciclos:
17                ciclos.append(ciclo)
18            atual = atual.proximo
19        caminho.pop()
20
21 def Ciclos(self): #retorna um vetor que tem tamanho NUMCAMINHOS com vetores dos caminhos percorridos
22     ciclos = []
23     for vertice in self.Dic:
24         caminho = []
25         if len(ciclos) >= NUMCAMINHOS:
26             return ciclos
27         self.CalculaCiclos(vertice, None, caminho, vertice, ciclos)
28     return ciclos
```

Figure 11. Algoritmo para encontrar os ciclos.

Este algoritmo implementa uma busca em profundidade (DFS) modificada para identificar ciclos simples em grafos não direcionados. No contexto de uma empresa de transportes, essa funcionalidade permite detectar circuitos fechados entre cidades, otimizando o planejamento de rotas e identificando alternativas de trajetos.

O método principal Ciclos() inicia a busca a partir de cada vértice do grafo, utilizando cada um como ponto de partida potencial. Para cada vértice, é acionado o método recursivo CalculaCiclos(), que explora sistematicamente todos os caminhos possíveis. O algoritmo interrompe a busca quando atinge o número máximo de ciclos definido por NUMCAMINHOS.

A detecção de ciclos ocorre quando quatro condições são atendidas simultaneamente: o vizinho atual corresponde ao vértice inicial (indicando circuito completo); a conexão não é meramente uma aresta de volta ao vértice pai; o caminho possui pelo menos quatro vértices; e o ciclo identificado é inédito. A estratégia emprega backtracking com caminho.append(v) e caminho.pop() para gerenciar eficientemente a exploração de rotas.

Com complexidade  $O(V + E)$ , o algoritmo é otimizado para grafos não direcionados e retorna uma lista de ciclos representados por sequências de vértices. Entre suas limitações, destaca-se a identificação apenas de ciclos simples e a dependência do parâmetro NUMCAMINHOS para a completude dos resultados.

## 3. Conclusão

A implementação da biblioteca para manipulação de grafos não direcionados ponderados demonstrou-se eficaz no contexto proposto de uma empresa de transportes. Através da

estrutura de Lista de Adjacência e da linguagem Python, foi possível desenvolver um conjunto abrangente de funcionalidades que atendem às necessidades práticas de análise de redes de transporte.

Os algoritmos implementados - incluindo Dijkstra para menor caminho, Tarjan para identificação de pontos de articulação, e a busca em profundidade modificada para detecção de ciclos - mostraram-se adequados para resolver problemas reais de conectividade e otimização de rotas. A capacidade de identificar cidades críticas na rede e circuitos turísticos viáveis comprova a utilidade prática da biblioteca desenvolvida.

O trabalho também evidenciou a importância dos conceitos teóricos de Teoria de Grafos aplicados a problemas concretos, permitindo aos integrantes do grupo consolidar o conhecimento adquirido na disciplina através da implementação prática. A divisão equilibrada de tarefas e o uso de ferramentas colaborativas como GitHub facilitaram o desenvolvimento coordenado do projeto.

## References

- Choudhary, R. (2022). Algoritmo de dijkstra com python: um guia completo.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos: teoria e prática*. Elsevier, Rio de Janeiro, 3 edition.
- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.
- Nunes, E. E. A. (2021). Listas encadeadas utilizando python.
- Soares, L. R. (2021). Busca em profundidade - python.
- Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160.