INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Estrutura de Dados e Algoritmos

Projeto Final – Implementação de um TAD Grafo para rede social

Grupo A2

João Portásio – 94754, Turma CDA1PL

Sandra Silva – 98372, Turma CDA1PL

Tiago Madeira – 95088, Turma CDA1PL

Docentes: Ana Almeida, Filipe Santos, Ricardo António

1º Ano da Licenciatura em Ciência de Dados – Pós Laboral

Ano Letivo 2020/2021 – 2º Semestre

Índice

1. Introdução

Com este trabalho pretende-se criar uma estrutura de dados - Grafo - com métodos de consulta, inserção, remoção e pesquisa a ser usado numa aplicação prática tendo por base dados de ligações (seguido-seguidor) de redes sociais. Para tal, implementaram-se as classes *Vertex* e *Edge*, para representar os vértices e as arestas desse mesmo grafo. Aplicado ao conceito de rede social, os vértices correspondem aos utilizadores e as arestas são as ligações entre estes.

Para calcular os caminhos mais curtos entre dois vértices, mais concretamente, entre um vértice origem e todos os restantes utilizadores, foi utilizado o algoritmo de Dijkstra, sendo que o caminho mais curto é o caminho que tem um menor custo, isto é, o que tem a menor soma dos pesos das arestas entre os utilizadores.

Na segunda fase do projeto, pretende-se determinar o caminho mais curto entre quaisquer dois vértices de origem e destino.

2. TAD Grafo

2.1 Definição

Um grafo é uma estrutura de dados que consiste numa coleção de nós ou vértices que estão ligados entre eles por arestas: um par do tipo (x,y) refere-se a uma aresta em que o vértice x está ligado ao vértice y.

Os grafos são utilizados para resolver problemas reais que retratam um problema de redes, como por exemplo, redes telefónicas, mapas ou redes sociais, como é o caso do Facebook, Instagram ou GitHub.

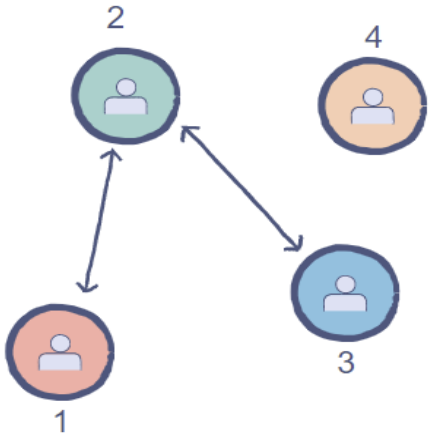

Figura 1. Estrutura de Grafo

No caso de uma rede social, é possível dizer que um utilizador pode ser representado como um vértice enquanto que as suas conexões podem ser representadas como arestas entre vértices. Além disso, cada vértice pode conter informação sobre o utilizador: nome, idade ou género, por exemplo.

2.2 Tipos de Grafos

Existem dois tipos de grafos: grafos dirigidos e grafos não dirigidos. No caso de grafos dirigidos, os vértices estão ligados por uma aresta direcionada, isto é, uma aresta com um sentido.

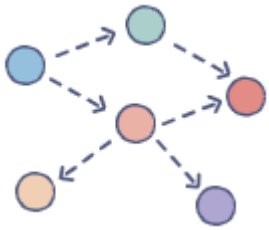

Figura 2. Grafo dirigido

Por exemplo, apenas é possível fazer uma travessia entre os vértices 1 e 2 ou 1 e 3, no sentido de 1 para 2 e de 1 para 3, mas nunca o oposto.

Nos grafos não dirigidos, os vértices estão conectados por arestas que são bidirecionais. Deste modo, se uma aresta ligar os vértices 1 e 2, é possível fazer a travessia do grafo do vértice 1 para o vértice 2 ou do vértice 2 para 1.

Figura 3. Grafo não dirigido

2.3 Tipos de Representação de Grafos

Os grafos podem ser representados através de uma lista de adjacências ou de uma matriz de adjacências. Numa lista de adjacências, o grafo é representado como um vetor de listas ligadas, em que o tamanho das listas é igual ao número de vértices. Cada índice *i* do vetor representa um vértice e cada elemento das listas representa os vértices que formam arestas com o vértice *i*.

Uma representação deste tipo pode ser exemplificada através da figura seguinte:

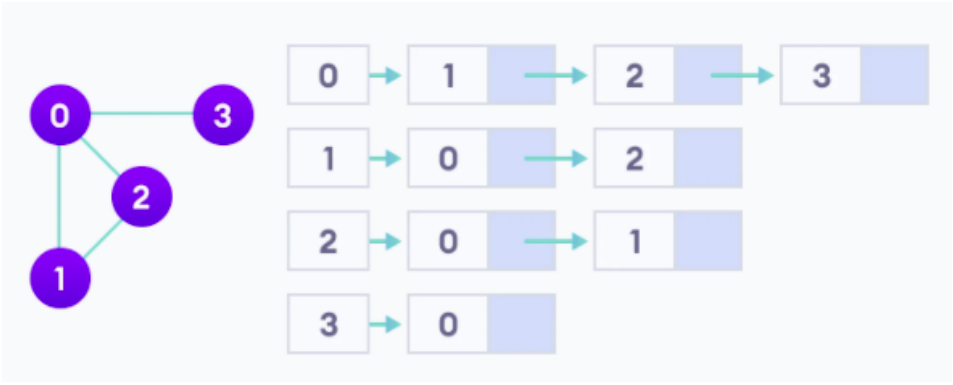

Figura 4. Representação de um grafo através de uma lista de adjacências

Esta estrutura tem vantagens como:

- Os novos vértices podem ser adicionados facilmente e estes podem ligar-se aos existentes sem dificuldade;
- É simples saber quem são os vértices adjacentes ou vizinhos de um dado vértice;

No entanto, pode assinalar-se a complexidade de tempo de procura desta representação, como uma desvantagem. Saber se existe uma aresta entre dois nós tem uma ordem de complexidade igual a O(n).

Esta estrutura de dados pode, também, ser representada através de uma matriz de adjacências. Neste tipo de representação, é criada uma matriz com *n* linhas e *n* colunas, em que *n* corresponde ao número de vértices do grafo. Para cada posição [*i*,*j*] da matriz, o valor a colocar na matriz pode ser igual a um, caso os vértices *i* e *j* se encontrem ligados por uma aresta ou zero, caso contrário.

Figura 5. Representação de um grafo através de uma matriz de adjacências

As vantagens de uma matriz de adjacências são:

- É rápido saber se existe uma aresta entre quaisquer dois vértices *i* e *j*;

Como desvantagens é possível assinalar:

- Ocupa muito espaço em memória em grafos com poucas arestas;
- Tem informação redundante para grafos não dirigidos.

Neste trabalho, a implementou-se um grafo baseado num mapa de adjacências, sendo que é permitida a alteração da estrutura que guarda a informação associada a cada vértice em dicionários, tendo como chave o vértice adjacente e como valor a aresta.

Foram criadas três classes: *Vertex*, *Edge* e *Graph*. Nas duas primeiras, foram criados métodos *getter* para ser possível devolver os identificadores dos objetos. Além disso, na classe *Vertex*, utilizou-se o método **__hash__** que devolve um número inteiro que identifica o vértice como uma chave no dicionário. Na classe *Edge*, foram criados métodos adicionais:

- **endpoints()**: que devolve um 'par' (u,v) com os seus vértices terminais (que é um par orientado no caso de um dígrafo);
- **opposite(v):** que devolve o outro vértice na aresta, isto é, o vértice oposto ao vértice v;

3. Algoritmo de Dijkstra

O algoritmo de Dijkstra, concebido pelo cientista de computação holandês, Edsger Wybe Dijkstra, é um algoritmo de procura que resolve problemas de caminho mais curto entre um dado vértice de origem e todos os restantes vértices do grafo, tendo em conta as seguintes condições:

- Os grafos podem ser direcionados ou não direcionados;
- Todas as arestas devem ter pesos não negativos;
- O grafo tem de ser conexo;

O custo do caminho que liga dois vértices é calculado através da soma dos pesos de todas as arestas que compõem o percurso. O caminho mais curto consiste na sequência de vértices, pela ordem que são visitados, que resulta no custo mínimo para fazer a travessia entre dois nós.

Em primeiro lugar, o custo de todos os nós ou vértices é colocado como infinito, o que significa que aquele custo ainda não foi calculado. Estes valores vão sendo atualizados à medida que os caminhos mais curtos vão sendo encontrados. O pseudocódigo para esta implementação pode ser:

```
COST ← 0 //used to index cost
PREVIOUS ← 1 //used to index previous node

FUNCTION dijkstras_shortest_path (graph, start_node)

  // initialise visited and unvisited lists
  unvisited ← {} // Unvisited list as empty dictionary
  visited ← {} // Visited list as empty dictionary

  // add every node to the unvisited list
  FOR each key in graph
    //set distance to infinity and previous to Null value
    unvisited[key] ← [∞ , NULL]
  ENDFOR
  // Set the cost of the start node to 0
```

```
    unvisited[start_node][COST] ← 0
     // repeat the following steps until unvisited list is empty
    finished ← False
    WHILE finished = False
      IF unvisited.length = 0 THEN
          finished ← True // no nodes left to evaluate
      ELSE
         // get unvisited node with lowest cost as current node
          current_node ← get_minimum(unvisited)
         // examine neighbours
         FOR neighbour in graph[current_node]
           // only check unvisited neighbours
            IF neighbour not in visited THEN
               //calculate new cost
                cost ← unvisited[current_node][COST] + graph[current_node][neighbour]
               // check if new cost is less
               IF cost < unvisited[neighbour][COST] THEN
                    unvisited[neighbour][COST] ← cost   //update cost
                    unvisited[neighbour][PREVIOUS] ← current_node //update previous
               ENDIF
            ENDIF
         // add current node to visited list
          visited[current_node] ← unvisited[current_node]
         // remove from unvisited list
         delete (unvisited[current_node])
       ENDIF
     ENDWHILE
     RETURN visited
ENDFUNCTION
```

Cada vez que o ciclo principal é executado, um dos vértices é retirado da lista prioritária. Se o grafo tiver n vértices, a fila prioritária terá O(n) vértices. Cada operação *pop* consome O(log

n). Então o tempo total para a execução do ciclo principal será da ordem de O(n log n), sendo que no pior caso, atinge um valor de $O(n^2 \log n)$.

4. Medidas de Centralidade

No contexto de redes sociais, as medidas de centralidade permitem conhecer quais os utilizadores mais influentes. Existem várias medidas de centralidade, mas serão abordadas apenas duas das existentes: centralidade de grau e centralidade de proximidade.

4.1 Centralidade de Grau

A centralidade de grau descreve, de uma forma mais simplista, o número de contactos que um determinado vértice tem. Esta medida pode ser calculada como:

$$C_d = \frac{deg\ deg\ (v)}{n-1}$$

No caso de um grafo dirigido podem ser calculadas duas medidas de centralidade de grau: de saída e de entrada. Aplicado a redes sociais, a centralidade de entrada mede a popularidade do nó e a centralidade de saída, o seu grau de influência.

4.2 Centralidade de Proximidade

Esta medida tem por base a soma das distâncias de um vértice em relação aos restantes vértices do grafo. Esta medida é calculada como:

$$C_c = \frac{1}{\sum_{i=1}^{n} \quad d_C(x,i)}$$

Quanto mais próximo um nó estiver de todos os restantes, maior a importância do vértice na rede.

5. Resultados obtidos

Para melhor compreensão do código desenvolvido, este foi dividido em módulos para facilitar o entendimento lógico. Existem 2 ficheiros de código, nomeadamente:

- **Grupo_A2.py** - Bases para implementação do grafo;
- **Grupo_A2_heap_queue.py** - Fila prioritária para utilização do método Dijkstra para determinação dos caminhos mais curtos no grafo;

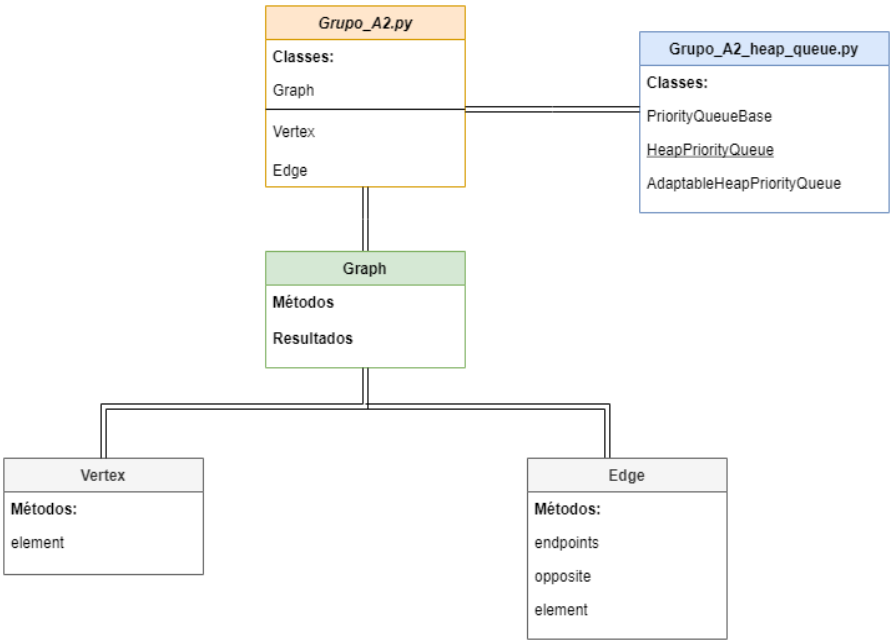

Figura 6. Estruturação do código

Primeiramente, criou-se a classe *Vertex* que visa a representar os objetos (vértices) do grafo, sendo que cada vértice representa os utilizadores da rede social.

```
class Vertex:
    __slots__ = '_element'

    def __init__(self, x):
        self._element = x

    def element(self):
        return self._element

    def __hash__(self):
        return hash(self._element)

    def __repr__(self):
        return '{0}'.format(self._element)

    def __eq__(self, other):
        if isinstance(other, Graph.Vertex):
            return self._element == other._element
        return False
```

Figura 7. Implementação da classe *Vertex*

De seguida, criou-se a classe *Edge* que representa as ligações entre os vértices no grafo.

```
class Edge:

    __slots__ = '_origin', '_destination', '_weight'

    def __init__(self, u, v, x):

        self._origin = u
        self._destination = v
        self._weight = x

    def endpoints(self):
        return (self._origin, self._destination)

    def opposite(self, v):
        return self._destination if self._origin == v else self._origin

    def element(self):
        return self._weight

    def __hash__(self):
        return hash((self._origin, self._destination))

    def __repr__(self):
        if self._weight is None:
            return '({0}, {1})'.format(self._origin, self._destination)
        return '({0}, {1}, {2})'.format(self._origin, self._destination, self._weight)
```

Figura 8. Implementação da classe *Edge*

Na classe *Edge*, foi necessário criar os seguintes métodos:

endpoints(): que devolve um 'par' (u,v) com os seus vértices terminais (que é um par orientado no caso de um dígrafo)

opposite(v): que devolve o outro vértice na aresta (i.e., o oposto a v)

A classe principal para a implementação do TAD Grafo, é a classe *Graph*, sendo que representa toda a estrutura da rede social. Alguns dos métodos relevantes para a construção do grafo, são nomeadamente:

- ***insert_vertex(x)***: insere e devolve um novo vértice com o elemento x;
- ***insert_edge(u,v,x)***: cria u e v e insere e devolve uma nova aresta entre u e v com peso x;
- ***remove_vertex(x)***: remove o vértice v e todas as arestas adjacentes a v;
- ***remove_edge(u,v)***: remove a aresta entre u e v;
- ***adjacent_vertices(v)***: retorna todos os vértices adjacentes a um dado vértice;
- ***incident_edges(v)***: gerador - indica todas as arestas (*outgoing*) incidentes em v. Se for um grafo dirigido e o *outgoing* for igual a *False*, devolve as arestas em *incoming*;
- ***vertex_count()***: devolve a quantidade de vértices no grafo;
- ***vertices()***: devolve um iterável sobre todos os vértices do grafo;
- **edge_count()**: devolve a quantidade de arestas do grafo;
- **edges()**: devolve o conjunto de todas as arestas do grafo;
- **degree()**: devolve a quantidade de arestas incidentes no vértice v. Se for um grafo dirigido, conta apenas as arestas *outcoming* ou em *incoming*, de acordo com o valor de *outgoing*;
- ***get_vertex(x)***: devolve o vértice do grafo com o elemento correspondente a x;
- ***get_edge(u,v)***: devolve a aresta que liga u e v, ou *None*, se não forem adjacentes;
- **printG()**: mostra o grafo por linhas;

```
class Graph:

    def __init__(self, directed=False):

        self._outgoing = {}
        self._incoming = {} if directed else self._outgoing

    def __getitem__(self, arg):
        return self._incoming[arg]

    def is_directed(self):
        return self._outgoing is not self._incoming

    def vertex_count(self):
        return len(self._outgoing)

    def vertices(self):
        return self._outgoing.keys()

    def get_vertex(self, x):
        for vertex in self.vertices():
            if vertex.element() == x:
                return vertex
        return None

    def edges_count(self):
        edges = set()
        for secondary_map in self._outgoing.values():
            edges.update(secondary_map.values())
        return len(edges)

    def edges(self):
        edges = set()
        for secondary_map in self._outgoing.values():
            edges.update(secondary_map.values())
        return edges

    def get_edge(self, u, v):
        return self._outgoing[u].get(v)

    def degree(self, v, outgoing=True):
        inc = self._outgoing if outgoing else self._incoming
        return len(inc[v])
```

```
    def incident_edges(self, v, outgoing=True):
        inc = self._outgoing if outgoing else self._incoming
        if v not in inc:
            return None
        for edge in inc[v].values():
            yield edge

    def adjacent_vertices(self, v, outgoing=True):
        if outgoing:
            if v in self._outgoing:
                return self._outgoing[v].keys()
            else:
                return None
        else:
            if v in self._incoming:
                return self._incoming[v].keys()
            else:
                return None

    def insert_vertex(self, x=None):
        for vertex in self.vertices():
            if vertex.element() == x:
                return vertex

        v = self.Vertex(x)

        self._outgoing[v] = {}
        if self.is_directed:
            self._incoming[v] = {}

        return v

    def insert_edge(self, u, v, x=None):
        if (v not in self._outgoing) or (v not in self._outgoing):
            raise Exception('One of the vertices does not exist')

        if self.get_edge(u, v):
            e = self.Edge(u, v, x)
            return e

        e = self.Edge(u, v, x)

        self._outgoing[u][v] = e
        self._incoming[v][u] = e

        return e
```

```
    def remove_edge(self, u, v):
        if not self.get_edge(u, v):
            raise Exception('Edge is already non-existent.')

        u_neighbours = self._outgoing[u]
        del u_neighbours[v]
        v_neighbours = self._incoming[v]
        del v_neighbours[u]

    def remove_vertex(self, x):

        if (x not in self._outgoing) and (x not in self._incoming):
            raise Exception('Vertex already non-existent')

        secondary_map = self._outgoing[x]
        for vertex in secondary_map:
            if self.is_directed():
                del self._incoming[vertex][x]
            else:
                del self._outgoing[vertex][x]
        del self._outgoing[x]

    def get_weight(self, v, outgoing=True):
        weights = {}
        queue = AdaptableHeapPriorityQueue()
        inc = self._outgoing if outgoing else self._incoming
        if v not in inc:
            return None
        for edge in inc[v].values():
            weights[v] = v
            weights[v] = queue.add(weights[v], edge.element())
        print(weights)
```

```
def printG(self):
    print('Grafo Orientado:', self.is_directed())

    print("Número de Vertices: {}".format(G.vertex_count()))

    print("Número de Arestas: {}".format(G.edges_count()))

    for v in self.vertices():
        print('\nUser: ', v, ' grau_in: ', self.degree(v, False), end=' ')
        if self.is_directed():
            print('grau_out: ', self.degree(v, False))
        for i in self.incident_edges(v):
            print(' ', i, end=' ')
        if self.is_directed():
            for i in self.incident_edges(v, False):
                print(' ', i, end=' ')
```

Figura 9. Implementação da classe *Graph*

Afim de proceder à leitura de ficheiros com a extensão (.csv), foi criado um método para carregamento desses dados, que obedece ao seguinte formato:

1. Por linha existem três valores de dados: o primeiro e o segundo indicam os nomes dos vértices e o terceiro, um peso.
2. Ignora a primeira linha do ficheiro csv.
3. Caso não exista uma terceira coluna do ficheiro csv, assume-se que o peso das arestas é 1.

```
def read_csv(filename):
    G = Graph()

    with open(filename, 'r') as csv_file:
        data = csv.reader(csv_file)
        next(data)

        for linha in data:
            id_origem = linha[0]
            id_destino = linha[1]
            peso = linha[2] if len(linha) > 2 else 1


            v_origem = G.insert_vertex(id_origem)
            v_destino = G.insert_vertex(id_destino)

            G.insert_edge(v_origem, v_destino, int(peso))

    return G
```

Figura 10. Implementação do método para leitura de um ficheiro .csv

De seguida, foi utilizado o algoritmo de Dijkstra para determinar os caminhos mais curtos no grafo, utilizando os pesos das arestas. [1]

A implementação do algoritmo é sob a forma de uma função que recebe como parâmetros, um grafo e um vértice de origem. No seguimento, devolve um dicionário, chamado ***cloud***, que mapeia cada vértice *v* que é alcançável desde a origem até todos os outros vértices do grafo, com o valor mais baixo do percurso **d(s,v)**. Foi também necessário recorrer à classe *AdaptableHeapPriorityQueue* (ficheiro Grupo_A2_heap_queue.py), que representa uma fila de prioridade adaptável. [1]

O valor ('inf'), do tipo ***float***, fornece um valor numérico que representa o infinito positivo.

```
def shortest_path_lengths(G, src):
    d = {}
    cloud = {}
    pq = AdaptableHeapPriorityQueue()
    pqlocator = {}
    source = Graph.Vertex(src)
    for v in G.vertices():
        if v == source:
            d[v] = 0
        else:
            d[v] = float('inf')
        pqlocator[v] = pq.add(d[v], v)

    while not pq.is_empty():
        key, u = pq.remove_min()
        cloud[u] = key
        del pqlocator[u]
        for e in G.incident_edges(u):
            v = e.opposite(u)
            if v not in cloud:
                wgt = e.element()
                if d[u] + wgt < d[v]:
                   d[v] = d[u] + wgt
                   pq.update(pqlocator[v], d[v], v)

    return cloud
```

Figura 11. Implementação do algoritmo de Dijsktra

O resultado obtido é idêntico a:

```
==================================================
shortest_path_lengths(G, "171316")

{171316: 0, 9236: 1, 1570: 1, 1563: 1, 171339: 1, 45703: 1, 2400: 1, 171337: 1, 96349: 1, 170787: 1, 91801: 1
```

Figura 12. Resultado da implementação do algoritmo de Dijsktra

Para a implementação das medidas de centralidade, foram criados os seguintes métodos:

- *degree_centrality(G)*;
- *closeness_centrality(G, src)*;

O método ***degree_centrality*** recebe um grafo como parâmetro e tem como objectivo contar o número de ligações incidentes sobre um nó, ou seja, o número de ligações que um nó tem. Desta forma quanto maior for o grau, mais central é o nó.

Passo 1: Criação do dicionário ***degrees*** que irá guardar os graus de centralidades, após terem sido calculados. Adicionalmente, criou-se a variável *vertex_count,* que guarda o número de vértices existentes no grafo.

Passo 2: Por cada vértice no grafo (utilização de um ciclo for), é calculado o grau desse vértice, usando o método ***degree***, anteriormente criado para o efeito e o valor é guardado na variável *vertex_degree*.

Passo 3: Utilizou-se a fórmula para calcular o grau de centralidade, ou seja, *vertex_degree* / (*vertex_count* - 1) e por fim, adicionar ao dicionário ***degrees*** um par de {key: value}, onde a *key* corresponde ao vértice e, o *value*, o grau de centralidade desse vértice.

O resultado obtido é idêntico a:

```
==================================================
degree_centrality(G)
{1570: 0.19, 9236: 0.11, 13256: 0.06, 171316: 0.01, 96349: 0.07, 2159: 0.04, 52402: 0.04, 1563: 0.13}
```

Figura 13. Resultado da implementação do método *degree_centrality*

O método ***closeness_centrality*** recebe um grafo e um vértice de origem como parâmetro e tem por base a soma das distâncias de um vértice em relação aos demais vértices do grafo. Para a implementação deste método foi necessário recorrer ao algoritmo de *Dijkstra*.

Passo 1: Criação da lista ***distances*** que guarda os pesos das arestas. Utilizou-se o método ***shortest_path_lengths*** (algoritmo de *Dijkstra*) afim de determinar o caminho mais curto entre o vértice de origem (*src*), aos demais vértices do grafo. O resultado é guardado na variável ***short***, que por sua vez, o output será um dicionário.

Passo 2: Por cada valor nos *values* do dicionário ***short***, se o *value* for diferente de *float('inf')*, então adiciona à lista ***distances***, esse respetivo *value*. A variável **soma** guarda a soma dos valores existentes na lista ***distances***.

Passo 3: Foi necessário criar uma condição que retorna um erro de exceção, caso o valor presente na variável **soma** seja igual a zero. Se a soma for diferente de zero, então é aplicada a

fórmula para calcular o grau de centralidade, ou seja, (*vertex_number* - 1) / soma, retornando o respetivo valor resultante desse cálculo.

O resultado obtido é idêntico a:

```
==========================================================
closeness_centrality(G, "1570")
0.5088836741535367
```

Figura 14. Resultado da implementação do método *closeness_centrality*

Afim de calcular os dez vértices mais interligados na rede, foi criado o método ***top_degree_centrality(G)***, que recebe um grafo como parâmetro e segue os seguintes passos:

Passo 1: Criação da lista ***top*** que guarda os tuplos (*key*, *value*) e foi criado a variável ***degrees*** que guarda os graus de centralidade resultantes da função ***degree_centrality*.**

Passo 2: Por cada par {*key: value*} no dicionário ***degrees*,** adiciona à lista ***top***.

Passo 3: Para organizar os valores na lista, utilizou-se a função ***sort_tuple*** que permite ordenar a lista de tuplos pelo segundo item. Por fim, são retornados os maiores dez graus de centralidade existentes na lista ***top*** (após terem sido organizados).

```
def top_degree_centrality(G):
    top = []
    degrees = degree_centrality(G)
    for key, value in degrees.items():
        top.append((str(key), value))
    return sort_tuple(top[:10])
```

Figura 15. Implementação do método *top_degree_centrality*

```
def sort_tuple(tup):
    lst = len(tup)
    for i in range(0, lst):

        for j in range(0, lst - i - 1):
            if tup[j][1] < tup[j + 1][1]:
                temp = tup[j]
                tup[j] = tup[j + 1]
                tup[j + 1] = temp
    return tup
```

Figura 16. Implementação do método *sort_tuple*

O resultado obtido é idêntico a:

```
==================================================
top_degree_centrality(G)
[('1570', 0.19), ('1563', 0.13), ('9236', 0.11), ('96349', 0.07), ('13256', 0.06), ('78115', 0.05), ('2159', 0.04), ('5
```

Figura 17. Resultado da implementação do método *top_degree_centrality*

Para calcular os dez vértices que apresentam maior proximidade aos restantes vértices do grafo, foi criado o método ***top_closeness(G)***, que recebe um grafo como parâmetro e segue os seguintes passos:

Passo 1: Criação da lista ***top*** que guarda os resultados, neste caso, os tuplos (vértice, grau de centralidade).

Passo 2: Por cada vértice no grafo (utilização de um ciclo *for*), é utilizado o método ***closeness_centrality*** que calcula o grau de proximidade desse mesmo vértice e guarda o valor na variável **d.** De seguida é feito o *append* de um tuplo (vértice, grau de proximidade), à lista ***top*.** Por fim, são retornados os dez vértices com maiores graus de proximidade existentes na lista ***top*** (após terem sido organizados através do método ***sort_tuple***).

O resultado obtido é idêntico a:

```
==================================================
top_closeness(G)
[(1570, 0.51), (9236, 0.47), (1563, 0.47), (13256, 0.43), (96349, 0.43), (2159, 0.43), (78115, 0.43), (52402, 0.41), (1
```

Figura 18. Resultado da implementação do método *top_closeness*

Na segunda fase do projeto, implementou-se o método ***DFS(graph, start, dest)***, que tem como objetivo determinar qual o caminho mais curto entre quaisquer dois vértices de origem e destino.

```
def DFS(graph, start, dest):
    stack = list()
    visited = list()
    s = Graph.Vertex(start)
    d = Graph.Vertex(dest)
    stack.append(s)
    visited.append(s)
    print('Visited', s)
    result = ["Not reachable", list()]
    while stack:
        node = stack.pop()
        if node == d:
            print('Destination node found', node)
            result[0] = 'Reachable'
            break
        print(node,'Is not a destination node')
        for child in graph[node]:
            if child not in visited:
                visited.append(child)
                stack.append(child)
    result[1] = visited
    return result
```

Figura 19. Implementação do método *DFS*

Para o teste deste método, foi utilizado o ficheiro 'Data_Facebook.csv'.

```
Lynch,Arnold,23
Murray,Douglas,29
Clark,Thornton,30
Schneider,Chambers,80
Ryan,Wilson,19
Wells,Schwartz,15
Berry,Silva,21
Garrett,Neal,17
Hughes,Bailey,13
Wong,Robinson,39
```

Figura 20. Extrato do ficheiro Data_Facebook.csv

Aplicando o método **DFS(graph, start, dest)**, que recebe um grafo, um vértice de origem e um vértice de destino como parâmetros, obteve-se o seguinte resultado:

```
==================================================
DFS(G, "Lynch", "Schmidt")
Visited Lynch
Lynch Is not a destination node
Howell Is not a destination node
Castillo Is not a destination node
Horton Is not a destination node
Destination node found Schmidt
['Reachable', [Lynch, Myers, Peterson, Vasquez, Garrett, Moreno, Pearson, Mccoy, Estrada, Powers, Reynolds, Wong, Newma
```

Figura 21. Resultado da implementação do método *DFS*

No **resultado** da figura anterior, foi escolhido o "Lynch" como vértice de origem e o "Schmidt", como vértice de destino. No entanto, a implementação deste método não está completamente correta, pois estão a ser retornadas ligações adicionais, ou seja, o método não está a guiar-se pelos pesos nas arestas, mas sim, pelas ligações dos vértices.

Bibliografia

[1] *Data Structures and Algorithms in Python*, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

[2] *Data model*. 3. Data model - Python 3.9.5 documentation. (n.d.). https://docs.python.org/3/reference/datamodel.html.

[3] *What is a graph (data structure)?* Educative. (n.d.). https://www.educative.io/edpresso/what-is-a-graph-data-structure.

[4] ArijitGayen@ArijitGayen. (2019, November 21). *Network Centrality Measures in a Graph using Networkx: Python*. GeeksforGeeks. https://www.geeksforgeeks.org/network-centrality-measures-in-a-graph-using-networkx-python/.

[5] Vicente, R. (n.d.). Redes Complexas. *Complex Systems EACH USP*.

[6] Laranjeira, P. A., & Cavique, L. (2014). Métricas de Centralidade em Redes Sociais. Revista De Ciências Da Computação, (9).

[7] randerson112358. (2019, April 10). Dijkstra's Algorithm. Medium. https://randerson112358.medium.com/dijkstras-algorithm-51c73caa1c39.

[8] Isaac Computer Science. (n.d.). https://isaaccomputerscience.org/concepts/dsa_search_dijkstra.

[9] Adjacency Matrix. Adjacency Matrix - an overview | ScienceDirect Topics. (n.d.). https://www.sciencedirect.com/topics/mathematics/adjacency-matrix.

[10] Adjacency Matrix. Programiz. (n.d.). https://www.programiz.com/dsa/graph-adjacency-matrix.

[11] Adjacency List. Programiz. (n.d.). https://www.programiz.com/dsa/graph-adjacency-list.