INSTITUTO UNIVERSITÁRIO DE LISBOA

Unidade Curricular: Estrutura de Dados e Algoritmos

Relatório do Trabalho de Projeto Final

Grupo A13

ANÁLISE DE LIGAÇÕES EM REDES SOCIAIS

Entregue em 30/05/2021

Elaborado por:
Catarina Brito, nº 98521
Sebastião Rosalino, n.º 98437

Coordenadora da UC:
Professora Ana Maria de Almeida
Docentes:
Professora Ana Maria de Almeida
Professor Filipe Santos
Professor Ricardo António

Licenciatura de Ciência de Dados - 1º ano - Turna CDA1
Ano Letivo 2020/2021 - 2º Semestre

Índice

1. Introdução e contextualização

O presente trabalho foi realizado no âmbito da UC de Estrutura de Dados e Algoritmos, tendo por objetivo principal a implementação de um TAD Grafo com métodos de consulta, inserção, remoção e pesquisa, a partir da aplicação de bases de dados previamente disponibilizadas relativas a utilizadores das redes sociais GitHub e Facebook.

Importa começar por referir que a análise de redes sociais é um campo de estudos destinado a investigar a estrutura das relações que conectam indivíduos (ou outras unidades sociais, como organizações) e como essas relações influenciam comportamentos e atitudes. Em geral, os estudos de análise de redes sociais são representados por Grafos, que não são mais do que representações gráficas dos indivíduos e das suas relações sociais.

Uma rede social consiste num conjunto de nós (indivíduos) e de ligações (conexões) entre eles. Os nós representam pessoas, entidades, organizações, sistemas, elementos computacionais, etc., que podem ser analisados individualmente ou coletivamente, observando a relação entre eles. Um dos objetivos dessas análises é compreender o impacto social dos nós através das suas conexões, na formação da estrutura da rede através dos seus relacionamentos.

As redes sociais podem ser formalizadas e analisadas através da teoria dos Grafos, utilizando formulações matemáticas para compreender os elementos e os seus relacionamentos dentro da rede. Nesse sentido, a teoria dos Grafos pode ser considerada um ramo da matemática aplicada que procura resolver problemas dentro de uma rede de relacionamentos.

Como referido, uma boa maneira para identificar uma rede é como um Grafo, composto por um conjunto de pontos também chamado de vértices unidos por linhas chamadas arestas. Grafos são estruturas que podem representar diversos tipos de dados. Esta representação é, em geral, bastante eficiente para representar relacionamentos entre indivíduos, do que resultou, naturalmente, a sua adoção por cientistas destas áreas como meio para estudar e representar redes sociais.

Nesse sentido, a teoria dos Grafos tem vindo a ser, cada vez mais, usada em análises de redes sociais devido a sua capacidade de representação e elevada simplicidade.

Utilizando as teorias e os algoritmos de redes de Grafos é possível analisar uma rede social e extrair diversas métricas desta rede. Como por exemplo, o número de comunidades, ou grupos existentes nessas redes e quais os vértices (nós) mais importantes da rede. A partir deste conhecimento, é possível direcionar esforços para os pontos mais importantes da rede, otimizando recursos e tempo.

Por exemplo, observando o exemplo de Grafo abaixo, é fácil perceber qual é o vértice mais central da rede. Do mesmo modo, o número de conexões do um vértice reflete o seu tamanho, por isso podemos também observar que o vértice mais central é também o mais conectado. Assim, uma informação entregue a esta rede por meio desse vértice, tem maior probabilidade de chegar aos demais vértices da rede mais rapidamente do que escolhendo outro vértice ao acaso.

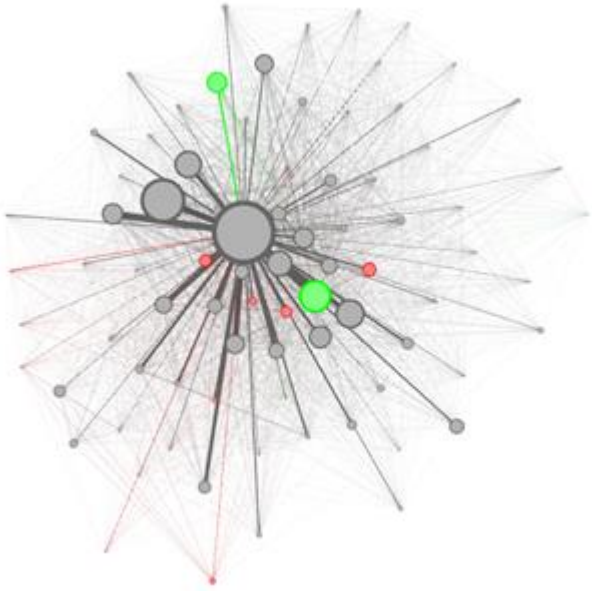

Figura 1 - Exemplo de Grafo

Nesse sentido, o presente trabalho visa analisar, a partir de dados das redes sociais GitHub e Facebook, as ligações entre os seus membros, avaliando dimensões como influência, centralidade e proximidade.

Para o efeito, foi implementado um algoritmo (Dijkstra) com o objetivo de determinar a centralidade e o caminho mais curto entre as ligações dentro daquelas redes sociais. A solução implementada pode ser útil para analisar qualquer outro tipo de rede social, bem como outros tipos de aplicações da vida real, tais como navegação em aplicações de transportes, na gestão de tráfego de dados em redes de computadores, ou até, em trocas financeiras (como, por exemplo, de moeda).

No presente trabalho e tendo por base os conceitos definidos, foi desenvolvido código com o objetivo de implementar métodos para determinar os caminhos mais curtos no Grafo criado para as redes sociais usadas: (a) sem usar os pesos nas arestas); (b) usando os pesos nas arestas. Foram, ainda, implementadas medidas de centralidade de grau (*degree centrality*) e centralidade de proximidade (*closeness*).

Na sequência da implementação das medidas de centralidade, foram apresentados resultados relativos ao: top 10 dos vértices mais interligados na rede; ao top 10 dos vértices que apresentam maior proximidade aos restantes; e ao top 10 dos vértices menos interligados na rede.

São apresentados testes de aplicação sobre as bases de dados disponibilizadas (GitHub e Facebook), com o objetivo de testar os métodos desenvolvidos e obter resultados de centralidade, de proximidade e de obtenção dos caminhos mais cursos.

Como elemento extra e com o objetivo de visualizar o Grafo, utilizou-se as bibliotecas networkx e matplotlib e apresentou-se gráficos de ligação dos utilizadores das redes sociais GitHub e Facebook.

2. Preparação para carregamento da base de dados

Tal como pedido na Fase 1 do projeto, foi implementada a TAD Grafo, de acordo com a prática nos exercícios do módulo 7, neste caso foi utilizada na forma de **mapas de adjacência** e consequentemente utilizou-se a interface e as classes **Vertex, Edge e Grafo**, já implementadas em aula. Adicionalmente, de modo a carregar a base de dados csv, foi utilizada a biblioteca csv.

Mapa de Adjacência:

Uma matriz de adjacência é uma das formas de se representar um grafo.

Dado um grafo G com n vértices, podemos representá-lo em uma matriz n x n A(G)=[aij] (ou simplesmente A). A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral o valor aij guarda informações sobre como os vértices vi e vj estão relacionados (isto é, informações sobre a adjacência de vi e vj).

Para representar um grafo não direcionado, simples e sem pesos nas arestas, basta que as entradas aij da matriz A contenham 1 se vi e vj são adjacentes e 0 caso contrário. Se as arestas do grafo tiverem pesos, aij pode conter, ao invés de 1 quando houver uma aresta entre vi e vj, o peso dessa mesma aresta.

Como será referido à frente, foi criada uma definição para o peso ser igual 1, caso a base de dados não tivesse esta coluna de informação.

```
def leitura_csv():          # Função para leitura do ficheiro
    filename = 'Github1.csv'
    with open(filename, 'r') as file:   # Abertura do ficheiro para leitura
        data = csv.reader(file, delimiter=",")
        next(data)      # A primeira linha não deve ser lida
        for valores in data:    # Para cada linha
            id_origem = valores[0]      # O id_origem será a primeira posição da linha, ou seja, o follower
            id_destino = valores[1]     # O id_origem será a segunda posição da linha, ou seja, o followed
            peso = int(valores[2] if len(valores) == 3 else 1)   # O peso será a terceira posição caso ela exista, se não, será por default 1
            if not g.exists_vertex(id_origem):      # Caso o vértice não exista
                g.insert_vertex(id_origem)          # Procede-se à sua inserção
            if not g.exists_vertex(id_destino):     # Caso o vértice não exista
                g.insert_vertex(id_destino)         # Procede-se à sua inserção
            g.insert_edge(id_origem, id_destino, peso)     # Posteriormente, é inserida a edge correspondente
```

Figura 2 - Implementação para a leitura da base de dados "Github1"

Adicionalmente, criou-se um método na TAD Grafo, de modo a verificar se o vértice inserido já se encontrava no grafo, com o intuito de evitar a duplicação de vértices.

```
def exists_vertex(self, x):
    return x in self._outgoing and x in self._incoming
```

Figura 3 - Método adicionado à classe Grafo, de modo a evitar vértices duplicados

3. Implementação de métodos para determinar caminhos mais curtos num grafo

Com o objetivo de determinar o caminho mais curto entre ligações, foi executada uma função baseada no algoritmo Dijkstra.

Considerando que é necessário um método que calcula o caminho mais curto entre as ligações, com ligações com pesos e sem pesos, apresenta-se de seguida os dois métodos:

a. Uso de pesos entre ligações

Código que faz a aplicação do método das ligações entre os vértices (seguido – seguidor):

```
def shortest_path(self, source):
    distance = {}
    for v in self.vertices():
        if v == source:
            distance[v] = 0
        else:
            distance[v] = np.inf
    dict_distance = {source: 0}
    while dict_distance:
        current_source_node = min(dict_distance, key=lambda k: dict_distance[k])
        del dict_distance[current_source_node]
        for node_dist in self.vertices()[current_source_node].values():
            adjnode = node_dist.destino
            lenght_to_adjnode = node_dist.peso
            if distance[adjnode] > distance[current_source_node] + lenght_to_adjnode:
                distance[adjnode] = distance[current_source_node] + lenght_to_adjnode
                dict_distance[adjnode] = distance[adjnode]
    return distance
```

Figura 4 - Caminho mais curto fazendo uso dos pesos das ligações

O método recebe como *input* uma *source*, ou seja, um vértice de começo. Seguidamente, foi criado um dicionário (*distance*), que consiste no output com as respetivas distâncias à *source*.

Tal como no algoritmo Dijkstra, a distância inicial da *source* será 0 e, consequentemente, os restantes vértices ainda não visitados terão como distância "infinito", para isso é efetuado um ciclo for que irá percorrer todos os vértices.

É usado um dicionário auxiliar dict_distance, cujo objetivo é atualizar as distâncias, de modo a obter a mais curta. Assim é iniciado um ciclo "while", que só é verificado enquanto o dicionário dict_distance não for vazio.

Para cada distância encontrada o adjnode será igual ao destino da edge; seguidamente o resultado de lenght_to_adjnode corresponderá ao peso da distância encontrada.

Posteriormente, foi colocada uma condição com o objetivo de atualizar o dicionário de input: caso o valor do nó de destino no dicionário de input seja maior que a soma entre o peso do nó source e peso da ligação entre ambos, faz-se uma atualização do caminho mais curto.

Para finalizar, quando termina o ciclo "while" é retornado então o output com todos os caminhos mais curtos em relação à source.

b. Sem uso do peso entre ligações

Utilizou-se o mesmo método com pesos entre ligações descrito anteriormente, porém optou-se por criar um peso *default* no ínicio do carregamento de dados, no caso de o ficheiro inserido não apresentar coluna com os respetivos pesos entre ligações, como por exemplo, a base de dados disponibilizada GitHub.

```
peso = int(valores[2] if len(valores) == 3 else 1)
```

Figura 5 - Default para peso igual 1, quando a base de dados não tem esta informação

Verifica-se pela condição acima que o peso por definição é 1, caso, existam 3 colunas, o peso das ligações, corresponderá ao indicado na respetiva coluna.

4. Implementação das medidas de centralidade

As medidas de centralidade aplicadas aos nodos de um grafo definem, no contexto de redes sociais, soluções para determinadas as características dos indivíduos dentro de uma rede social.

a. Centralidade de grau (*degree centrality*)

No caso da centralidade de grau, é medido o quão "popular" é um indivíduo, tendo por base o número de pessoas com quem tem relações de conhecimento.

A centralidade de grau de um vértice é dada pela quantidade de vértices adjacentes, ou seja, grau de um nó. A importância de um vértice deve-se, assim, ao seu número de ligações e à assunção de que importante será um nó por onde muitos dos caminhos no grafo passem.

Dado que ao utilizar a base de dados GitHub se obtém um grafo dirigido, então a popularidade corresponderá ao grau do *incoming* (entrada), e o número de seguidores e a influência é calculado pelo grau do *outgoing.*

```
def degree_centrality(self, v):
    popularidade = self.degree(v, False)
    influencia = self.degree(v, True)
    return "Este utilizador tem {} de popularidade e {} de influência".format(popularidade, influencia)
```

Figura 6- Método degree_centrality

b. Centralidade de proximidade (*closeness*)

Através da centralidade de proximidade mede-se o quão rapidamente (através de muitos ou poucos contactos) um indivíduo consegue comunicar com os restantes elementos do grupo.

A centralidade de proximidade é obtida pela quantidade de vezes que o vértice intervém no caminho mais curto.

Se o nó for várias vezes interveniente no processo de obtenção do caminho mais curto, então este pode-se dizer que tem um papel central.

No desenvolvimento do método definiu-se, primeiro, a distância total como ponto de partida igual a 0. Posteriormente, é inicializado um ciclo *for* para todos os vértices do Grafo para definir o caminho mais curto em relação ao input dado. Desse modo, se a distância for diferente de infinito, ou seja, se passou em determinado vértice, é então

atualizada a distância total com os respetivos valores encontrados no output do caminho mais curto.

Por último, é então retornada a centralidade de proximidade do utilizador que é dada pelo inverso da soma das distâncias.

```
def closeness(self, v):
    total_distance = 0
    for distance in self.shortest_path(v).values():
        if distance != np.inf:
            total_distance += distance
    if total_distance == 0:
        final_closeness = 0
    else:
        final_closeness = 1 / total_distance
    return final_closeness
```

Figura 7- Método closeness

5. Indicadores de centralidade e proximidade

a. Top 10 dos vértices mais interligados

```
def top10_interligados(self):   # Assumi que mais interligados = mais followeds
    all_users = {}
    for user in self._outgoing:
        all_users[user] = self.degree(user)
    all_users_sorted = sorted(all_users.items(), key=lambda x: x[1], reverse=True)
    final_top = []
    for e in all_users_sorted[0:10]:
        final_top.append(e[0])
    return final_top
```

Figura 8 - Método top 10 dos vértices mais interligados na sua rede

Como é possível verificar no método acima indicado, começa-se por inicializar um dicionário vazio, com o objetivo de guardar todos os seguidores. Seguidamente, é inicializado um ciclo *for* que irá percorrer todos os utilizadores e consequentemente, será criada uma chave, utilizador, e um valor que corresponderá ao número de seguidores que este contém, determinado pelo método *self.degree*.

Com o intuito de realizar uma hierarquia com os valores mais elevados, procedeu-se à ordenação por maior número de seguidores.

Por último, é então retornado os 10 utilizadores com maior número de seguidores.

Resultado:

['7580', '465', '2907', '2400', '14922', '20628', '1570', '4140', '1567', '2556']

b. Top 10 dos vértices que apresentam maior proximidade aos restantes

```
def top10_closeness(self):
    all_users = {}
    for user in self._outgoing:
        all_users[user] = self.closeness(user)
    all_users_sorted = sorted(all_users.items(), key=lambda x: x[1], reverse=True)
    final_top = []
    for e in all_users_sorted[0:10]:
        final_top.append(e[0])
    return final_top
```

Figura 9 - Top 10 dos vértices que maior proximidade aos restantes

Verifica-se que inicialmente é criado um dicionário vazio, com o objetivo de guardar os valores de proximidade dos vértices. Posteriormente, é realizado um ciclo *for* com o propósito de encontrar todos os utilizadores, sendo, consequentemente, criada uma chave, utilizador, fazendo correspondência a um valor, proximidade, que é determinado pelo método *closeness*.

De forma a retornar de forma hierárquica os utilizadores que apresentam maior proximidade, utiliza-se a função de ordenação predefinida *sorted*.

Para finalizar, o *output* traduz-se numa lista com os 10 utilizadores com maior proximidade.

Resultado:

['135978', '233103', '17407', '159663', '246444', '3460987', '1409312', '253969', '129594', '28121']

c. Top 10 dos vértices menos interligados

Acrescentou-se um método com o intuito de verificar quais os utilizadores com menos seguidores.

```
def top10_least_followed(self):     # Os 10 utilizadores com menos seguidores
    all_users = {}
    for user in self._outgoing:
        all_users[user] = self.degree(user)
    all_users_sorted = sorted(all_users.items(), key=lambda x: x[1])
    final_top = []
    for e in all_users_sorted[0:10]:
        final_top.append(e[0])
    return final_top
```

Figura 10 - Método top 10 dos vértices menos interligados na sua rede

Assim, contrariamente ao que foi definido no método top10_interligados, o processo de hierárquica é do menor para o maior, ou seja, a diferença encontra-se na predefinição de *reverse* já implementada na função predefinida *sorted*.

Resultado:

['157', '9635', '52402', '1713', '1326', '1708', '2239', '414', '130027', '13003']

6. Testes efetuados

a. Utilizando a base de dados GitHub

```
if __name__ == '__main__':
    g = Grafo(True)
    leitura_csv()
    print(g.top10_interligados())
    print(g.top10_least_followed())
    print(g.shortest_path('1570'))
    print(g.degree_centrality('1570'))
    print(g.closeness('1570'))
    print(g.top10_closeness())
```

Figura 11 - Primeiro teste efetuado com a base de dados GitHub1

Resultado:

```
Python Console>>> runfile('C:/Universidade_ISCTE/1°Ano/2.°_Semestre_20_21/1_Estrutura de Dados e Algoritmos/Programas_Codigo/Projeto_EDA_17_05.py', wdir='C:/Universidade_ISCTE/1°A
['7580', '465', '2907', '2480', '14922', '20628', '1570', '4140', '1567', '2556']
['157', '9635', '52402', '1713', '1326', '1708', '2239', '414', '130027', '13003']
{'1570': 0, '9236': 1, '157': 2, '13256': 1, '171316': 1, '9635': 2, '2159': 1, '52402': 2, '96349': 2, '1563': 1, '169553': 2, '78115': 2, '170787': 1, '1713': 2, '1326': 2, '170
Este utilizador tem 0 de popularidade e 83 de influência
0.0002759381898454746
['135978', '233103', '17407', '159663', '246444', '3460987', '1409312', '253969', '129594', '28121']
```

Figura 12 - Output do primeiro teste

b. Utilizando a base de dados Facebook

```
if __name__ == '__main__':
    g = Grafo(True)
    leitura_csv()
    print(g.top10_interligados())
    print(g.top10_least_followed())
    print(g.shortest_path('Schneider'))
    print(g.degree_centrality('Schneider'))
    print(g.closeness('Schneider'))
    print(g.top10_closeness())
```

Figura 13 - Teste com a base de dados Facebook

Resultado:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
>>> runfile('C:/Universidade_ISCTE/1°Ano/2.°_Semestre_20_21/1_Estrutura de Dados e Algoritmos/Programas_Codigo/Projeto_EDA_17_05.py', wdir='C:/Universidade_ISCTE/1°Ano/2.°_Semestr
['Nguyen', 'Gomez', 'Schneider', 'Mcdaniel', 'Lewis', 'Robinson', 'Garrett', 'Banks', 'Lee', 'Murray']
['Aguilera', 'Madrid', 'Valle', 'Irwin', 'Walden', 'Ratliff', 'Duarte', 'Camacho', 'Chappell', 'Pedersen']
{'Murray': 8, 'Douglas': 5, 'Clark': 21, 'Thornton': 23, 'Schneider': 0, 'Chambers': 16, 'Ryan': 15, 'Wilson': 12, 'Wells': 18, 'Schwartz': 16, 'Berry': 29, 'Silva': 13, 'Garrett'
Este utilizador tem 72 de popularidade e 72 de influência
8.208158909956496e-05
['Madrigal', 'Tatum', 'Hoang', 'Huff', 'Mccann', 'Hilliard', 'Campbell', 'Jeffries', 'Mayo', 'Corcoran']
```

Figura 14 - Output do teste realizado

c. Uso do networkx para visualização do Grafo

Com o objetivo de visualizar o Grafo, optou-se por testar com a biblioteca networkx e matplotlib. Porém, devido à quantidade de dados nas bases de dados, tanto na Data_Facebook.csv, como na GitHub.csv, tornou-se necessário a sua redução. Nesse sentido, foram criadas duas novas bases de dados (a partir das originais), de modo a realizar testes com menor tempo de execução.

Implementação das bibliotecas utilizadas:

```
import csv
import networkx as nx
import matplotlib.pyplot as plt
```

Figura 15 - Bibliotecas

O código implementado foi:

```
def visualizar(file):
    g = nx.DiGraph()
    filename = file
    with open(filename, 'r') as file:
        data = csv.reader(file, delimiter=",")
        next(data)
        for valores in data:
            id_origem = valores[0]
            id_destino = valores[1]
            peso = int(valores[2] if len(valores) == 3 else 1)
            g.add_edges_from([(id_origem,id_destino)],weight=peso)
            pos = nx.spring_layout(g)
        nx.draw_networkx_nodes(g, pos, node_size=100)
        nx.draw_networkx_edges(g, pos, edgelist=g.edges(), edge_color='black')
        nx.draw_networkx_labels(g, pos)
        plt.show()
```

Figura 16 - Código para implementação de networkx

Primeiro teste realizado com a base de dados “GitHub_short.csv”:

```
if __name__ == '__main__':
    g = Grafo(True)
    leitura_csv('Data_Facebook.csv')
    print(g.shortest_path('Clark'))
    print(g.degree_centrality('Ryan'))
    print(g.closeness('Silva'))
    visualizar('Github_short.csv')
```

Figura 17 - Uso da base de dados Github_short.csv

Obteve-se a seguinte visualização:

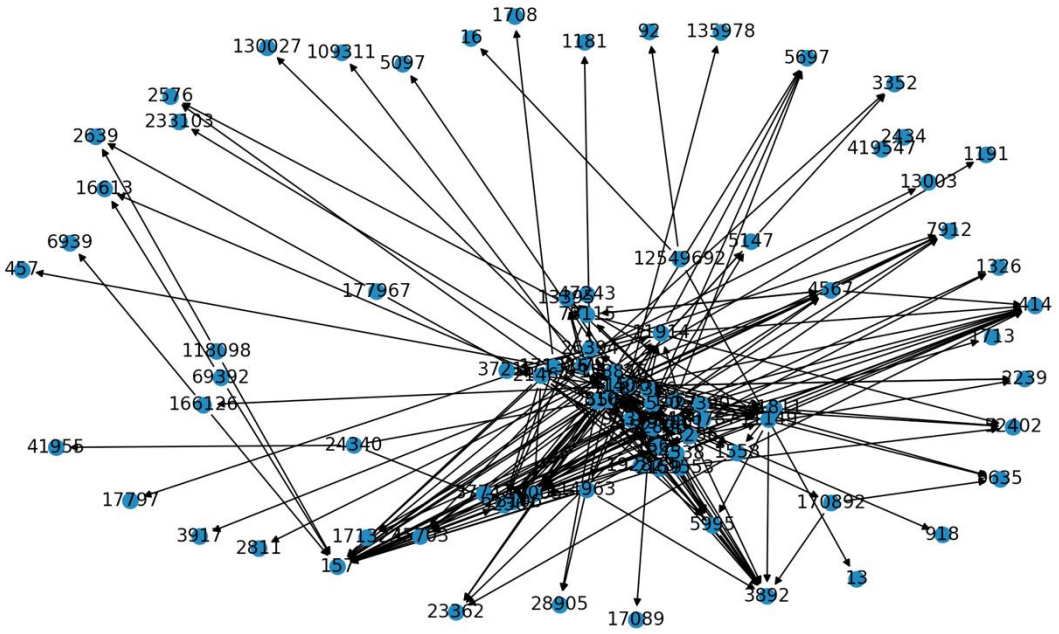

Figura 18 - Grafo obtido através de teste apresentado na figura 17

Seguidamente, foi utilizada a base de dados disponibilizada com pesos, Facebook_short.csv:

```
if __name__ == '__main__':
    g = Grafo(True)
    leitura_csv('Data_Facebook.csv')
    print(g.shortest_path('Clark'))
    print(g.degree_centrality('Ryan'))
    print(g.closeness('Silva'))
    visualizar('Facebook_short.csv')
```

Figura 19 - Teste para visualizar grafo com base de dados Facebook_short.csv

Obteve-se a seguinte visualização:

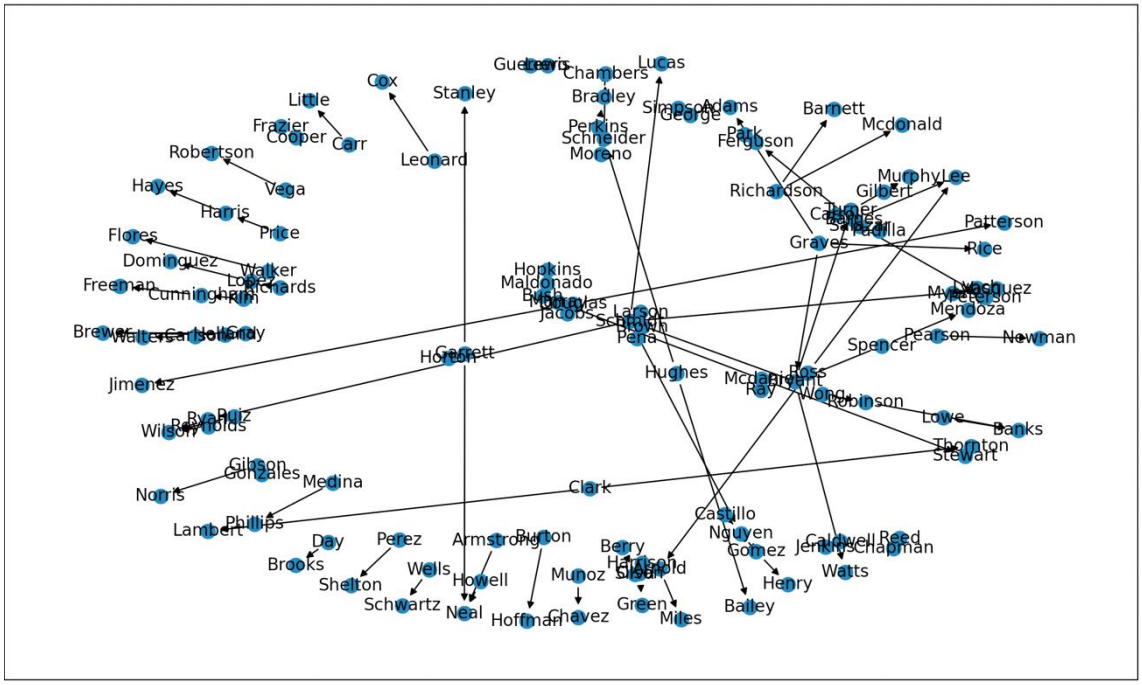

Figura 20 - Grafo originado pela base de dados Facebook_short.csv

7. Bibliografia e Webgrafia

Livros:

- M. Goodrich, R. Tamassia e M. Goldwasser, Data Structures & Algorithms in Python, Wiley, 2013.
- Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, Release 3.0, 2013.
- Network Science, A.-L. Barabási, disponível em http://networksciencebook.com

Foram ainda consultados os seguintes sites:

https://core.ac.uk/download/pdf/303042278.pdf

https://arxiv.org/pdf/1901.07901.pdf

https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-022-introduction-to-network-models-fall2018/lecture-notes/MIT1_022F18_lec4.pdf