title

Projeto Final - Implementação de um TAD Grafo para Rede Social

Licenciatura em Ciência de Dados

UC: Estrutura de Dados e Algoritmos

Docentes:

Ana Maria de Almeia

Filipe Santos

Ricardo António

Grupo A2:

João Portásio, nº 94754

Sandra Silva, nº 98372

Tiago Madeira, nº 95088

30 de Maio de 2021

Índice

Introdução

No projeto final de EDA, a primeira fase consistiu em implementar um TAD Grafo com métodos de consulta, inserção, remoção e pesquisa, que vai ser usado numa aplicação prática, usando dados de ligações (seguido-seguidor) do GitHub. Para a implementação do grafo, foi necessário implementar a classe Vertex e Edge. Os utilizadores da rede social, correspondem aos vértices do grafo e as arestas são as ligações entre os utilizadores.

Afim de calcular os caminhos mais curtos no grafo, foi utilizado o algoritmo de Dijkstra. Este algoritmo permitiu determinar o caminho mais curto entre dois utilizadores. O caminho mais curto é aquele que tem o menor custo (peso da aresta) entre os utilizadores.

Na segunda fase do projeto...*Complementar*

Grafos e Redes

Grafos são representações de objetos que se relacionam entre si. Os objetos denominam-se de vértices e as relações entre os objetos, arestas. Se as relações não forem dirigidas, ou seja, se afetarem ambos os vértices de igual forma, trata-se de um grafo não dirigido (undirected) ou não orientado. Caso contrário, denomina-se grafo dirigido (directed) ou orientado. Neste caso, as arestas são normalmente designadas por arcos, já que implicam uma única direção. É de notar que basta existir uma aresta dirigida, para todo o grafo se considerar dirigido.

Grafo não dirigido

title

Grafo dirigido

title Uma aresta num grafo dirigido costuma designar-se por arco e um vértice por nó. Cada arco tem um nó origem (antecessor) e um nó fim (sucessor).

No projeto, a estrutura utilizada para a implementação do **TAD Grafo**, foi baseada em mapa de adjacências, sendo que permite alterar os "contentores" associados a cada vértice, para um dicionário, tendo por "chave", o vértice adjacente e por "valor", a aresta.

Nas classes Vertex e Edge, foram criados métodos '**getter**', para devolver os identificadores dos objetos. Na classe Vertex, utilizou-se o método **hash**(), que devolve um número inteiro (integer) que identifica o vértice como uma chave no dicionário.

Algoritmo de Dijkstra

In [ ]:

Medidas de Centralidade

Em ciências da computação, quando se estuda redes, pretende-se identificar vértices influentes ou importantes. A identificação destes vértices usa medidas de centralidade com base nas diferentes noções de importância dos vértices ou arestas.

Algumas destas medidas são: a centralidade de grau (degree centrality), a de proximidade (closeness) e a de intermediação (betweeness). Nestes casos, a importância de um vértice devesse ao seu número de ligações e à assunção de quão importante será um nó por onde muitos dos caminhos no grafo passem.

Centralidade de grau

A centralidade de grau de um determinado vértice, é calculada como a quantidade de "vizinhos" (nós adjacentes), ou seja, o grau de um nó. Se o grafo for dirigido, é importante saber a centralidade de entrada e a de saída, uma vez que a primeira mede a popularidade do nó e a segunda, o grau de influência.

Centralidade de proximidade

A centralidade de proximidade de um vértice é calculada como o inverso da soma das distâncias do vértice a todos os restantes. Quanto mais próximo um nó estiver de todos os restantes, maior a importância do vértice na rede.

Resultados obtidos

Para melhor compreensão do código desenvolvido, decidimos dividi-lo em alguns 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.

title

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.

In [20]:
```
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._elementx


        return False
```

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

In [21]:
```
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)
```

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 class 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 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

In [22]:
```
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=' ')
```

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 3 valores de dados, o 1.º e o 2.º, indicam os nomes dos vértices e o 3.º, 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.

In [23]:
```
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
```

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

A nossa implementação do algoritmo é sob a forma de uma função, com o comprimento mais curto do caminho, recebendo como parâmetros, um grafo e um vértice de origem. No seguimento, devolve um dicionário, chamado **cloud**, mapeando cada vértice v que é alcançável desde a origem até à sua distância mais curta, de percurso **d(s,v)**. Foi também necessário recorrer à classe **AdaptableHeapPriorityQueue**(ficheiro Grupo_A2_heap_queue), que representa uma fila de prioridade adaptável. [1]

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

In [24]:
```
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
```

Output:

In [ ]:
```
========================================================
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,
```

In [ ]:
```
Para a implementação das medidas de centralidade, foram criados os seguintes métodos:

degree_centrality(G)
closeness_centrality(G, src)
top_degree_centrality(G)
```

Bibliografia

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

[2] https://docs.python.org/3/reference/datamodel.html

[3] https://www.educative.io/edpresso/what-is-a-graph-data-structure