Atividade 2

O código abaixo é responsável por gerar a rede de amigos descrita no enunciado da atividade 2. As únicas partes que devem ser alteradas neste código são: (i) a função `_search` e (ii) os parâmetros (números de vértices e arestas) da função que gera a rede. O entendimento do código é bem importante e pode facilitar na implementação.

Problema #1

Implementação normal da busca em largura.

In [1]:
```
import random
import uuid

class Person(object):

    def __init__(self, uid, s_type):
        self._uid = uid
        self._s_type = s_type

    def get_uid(self):
        return self._uid

    def get_s_type(self):
        return self._s_type


class FriendNetwork(object):

    def __init__(self, people_num, connections_num):
        self._people_num = people_num
        self._connections_num = connections_num
        self._graph = self._generate_graph()

    def _generate_graph(self):

        people = []
        for person_index in range(self._people_num):
            uid = str(uuid.uuid4())
            s_type = 'female' if person_index < (self._people_num // 2)  else 'male'
            people.append(Person(uid, s_type))

        conn_num = 0
        graph = {}
        graph_aux = {} # criando um grafo auxiliar para agilizar algumas buscas
        while conn_num < self._connections_num:
            person, friend = random.sample(people, 2)
            person_uid = person.get_uid()
            friend_uid = friend.get_uid()

            if person_uid not in graph:
                graph[person_uid] = {
                    'this': person,
                    'friends': []
                }
                # criando um índice auxiliar para os vizinhos de cada vértice inserido n
                graph_aux[person_uid] = {}
```

```
            if friend_uid not in graph:
                graph[friend_uid] = {
                    'this': friend,
                    'friends': []
                }
                # criando um índice auxiliar para os vizinhos de cada vértice inserido n
                graph_aux[friend_uid] = {}

            # if person_uid == friend_uid or \
            #     friend in graph[person_uid]['friends']: # fazer essa verificação em um
            #     continue
            if person_uid == friend_uid or \
                friend_uid in graph_aux[person_uid]: # fazer essa verificação em um índi
                continue

            graph[person_uid]['friends'].append(friend)
            graph[friend_uid]['friends'].append(person)
            # adicionar vizinho também nos índices do grafo auxiliar
            graph_aux[person_uid][friend_uid] = True
            graph_aux[friend_uid][person_uid] = True
            conn_num += 1

        people_to_remove = []
        for person_uid in graph:
            friends_types = [*map(lambda p: p.get_s_type(), graph[person_uid]['friends']
            person_type = graph[person_uid]['this'].get_s_type()
            if ('male' not in friends_types or 'female' not in friends_types) and person
                people_to_remove.append({'person_uid': person_uid, 'remove_from': graph[

        for person_props in people_to_remove:
            for friend in person_props['remove_from']:
                person_index = [*map(lambda friend: friend.get_uid(),
                    graph[friend.get_uid()]['friends'])].index(person_props['person_uid'
                del graph[friend.get_uid()]['friends'][person_index]
            del graph[person_props['person_uid']]

        return graph

    def get_person_by_uid(self, uid):
        return self._graph[uid]['this']

    def _search(self, person_uid, friend_uid):
        """
        Busca em largura no grafo de relacionamentos entre `person_uid` e `friend_uid`.

        A busca foi implementada usando uma lista de `fronteira` que armazena todos os v
        sendo avaliado. A lista `visitados` armazena todos os vértices que já foram expa
        expandidos novamente. Com isso, são evitados os problemas de ciclos no grafo.

        A função retorna uma lista menor caminho entre os dois vértices ou lança uma exc
        caminho entre eles.

        :param person_uid:          estado inicial do grafo
        :param friend_uid:          estado final do grafo
        :return:                    lista com menor caminho entre o estado inicial e fin

        :raise RuntimeException:    erro quando não existe uma conexão entre as duas pes
        """
        from collections import namedtuple
        # print(person_uid, friend_uid)

        # Estrutura dos vértices da árvore de busca
        Vertice = namedtuple('Vertice', ['uid', 'pai'])

        fronteira = []      # Lista com todos os vértices a serem explorados
        expandidos = {}     # Lista com todos os vértices já expandidos
```

```
        def expandir_fronteira(vertice: Vertice):
            """Adiciona na fronteira todos os vértices vizinhos que ainda não foram expa
            vizinhanca = [Vertice(p.get_uid(), vertice) for p in self._graph[vertice.uid
            fronteira.extend(vizinhanca)

            # Garante que os vértices que já entraram na fronteira não sejam expandidos
            for vizinho in vizinhanca:
                expandidos[vizinho.uid] = vizinho


        # Expande a raiz da árvore
        raiz = Vertice(person_uid, None)

        expandidos[raiz.uid] = raiz
        expandir_fronteira(raiz)

        # Realiza o processo de busca até encontrar a solução
        while fronteira:
            # Remove o primeiro vértice da fronteira
            #     - FIFO para a busca em largura
            #     - LIFO para a busca em profundidade
            vertice = fronteira.pop(0)

            # Verifica se encontrou o estado final
            if vertice.uid == friend_uid:

                # Recupera o caminho entre o vertice atual e a raiz
                caminho = []
                while vertice.pai != None:
                    caminho.append(vertice.uid)
                    vertice = expandidos[vertice.uid].pai
                caminho.append(raiz.uid)
                caminho.reverse()
                # print(caminho)
                # print(len(caminho) - 1)
                # print()
                return caminho

            # Expande a fronteira a partir do vertice atual
            else:
                expandir_fronteira(vertice)

        # Não encontrou solução - neste caso, qualquer valor retornado vai bagunçar com
        raise RuntimeError(f'Não existe conexão entre {person_uid} e {friend_uid}!')

    def get_separation_degree(self):

        total_paths_len = 0
        for _ in range(100):
            person_uid, friend_uid = random.sample([*self._graph.keys()], 2)
            path = self._search(person_uid, friend_uid)
            total_paths_len += len(path) - 1

        return total_paths_len / 100
```

Teste #1

In [3]:

```
import numpy as np
from math import sqrt

n_vertices = [100, 1000, 10000, 100000]

print(f"+-{'-'*7}-+-{'-'*9}-+-{'-'*9}-+-{'-'*9}-+")
```

```
print(f"| {'n':<7} | {'5':^9} | {'sqrt(5)':^9} | {'n/5':^9} |")
print(f"+-{'-'*7}-+-{'-'*9}-+-{'-'*9}-+-{'-'*9}-+")

for n in n_vertices:
    n_arestas = [int(5 * n), int(sqrt(n) * n), int(n / 5 * n)]

    medias = []
    for a in n_arestas:
        if a == 2_000_000_000 and n == 100_000:
            medias.append(-1)
            break

        friend_network = FriendNetwork(n, a)

        resultados = []
        resultados.append(friend_network.get_separation_degree())

        medias.append(np.mean(resultados))

    print(f'| {n:<7} | {medias[0]:^9.2f} | {medias[1]:^9.2f} | {medias[2]:^9.2f} |')

print(f"+-{'-'*7}---{'-'*9}---{'-'*9}---{'-'*9}-+")
```

```
+---------+-----------+-----------+-----------+
| n       |     5     |  sqrt(5)  |    n/5    |
+---------+-----------+-----------+-----------+
| 100     |   2.30    |   1.83    |   1.54    |
| 1000    |   3.28    |   1.95    |   1.60    |
| 10000   |   4.26    |   2.01    |   1.63    |
| 100000  |   5.27    |   2.00    |   -1.00   |
+-------------------------------------------+
```

Problema #2

Implementação da busca em largura alternando gênero.

In [4]:

```
import random
import uuid

class Person(object):

    def __init__(self, uid, s_type):
        self._uid = uid
        self._s_type = s_type

    def get_uid(self):
        return self._uid

    def get_s_type(self):
        return self._s_type


class FriendNetwork(object):

    def __init__(self, people_num, connections_num):
        self._people_num = people_num
        self._connections_num = connections_num
        self._graph = self._generate_graph()

    def _generate_graph(self):

        people = []
        for person_index in range(self._people_num):
            uid = str(uuid.uuid4())
```

```
            s_type = 'female' if person_index < (self._people_num // 2)  else 'male'
            people.append(Person(uid, s_type))

        conn_num = 0
        graph = {}
        graph_aux = {} # criando um grafo auxiliar para agilizar algumas buscas
        while conn_num < self._connections_num:
            person, friend = random.sample(people, 2)
            person_uid = person.get_uid()
            friend_uid = friend.get_uid()

            if person_uid not in graph:
                graph[person_uid] = {
                    'this': person,
                    'friends': []
                }
                # criando um índice auxiliar para os vizinhos de cada vértice inserido n
                graph_aux[person_uid] = {}

            if friend_uid not in graph:
                graph[friend_uid] = {
                    'this': friend,
                    'friends': []
                }
                # criando um índice auxiliar para os vizinhos de cada vértice inserido n
                graph_aux[friend_uid] = {}

            # if person_uid == friend_uid or \
            #     friend in graph[person_uid]['friends']: # fazer essa verificação em um
            #     continue
            if person_uid == friend_uid or \
                friend_uid in graph_aux[person_uid]: # fazer essa verificação em um índi
                continue

            graph[person_uid]['friends'].append(friend)
            graph[friend_uid]['friends'].append(person)
            # adicionar vizinho também nos índices do grafo auxiliar
            graph_aux[person_uid][friend_uid] = True
            graph_aux[friend_uid][person_uid] = True
            conn_num += 1

        people_to_remove = []
        for person_uid in graph:
            friends_types = [*map(lambda p: p.get_s_type(), graph[person_uid]['friends']
            person_type = graph[person_uid]['this'].get_s_type()
            if ('male' not in friends_types or 'female' not in friends_types) and person
                people_to_remove.append({'person_uid': person_uid, 'remove_from': graph[

        for person_props in people_to_remove:
            for friend in person_props['remove_from']:
                person_index = [*map(lambda friend: friend.get_uid(),
                    graph[friend.get_uid()]['friends'])].index(person_props['person_uid'
                del graph[friend.get_uid()]['friends'][person_index]
            del graph[person_props['person_uid']]

        return graph

    def get_person_by_uid(self, uid):
        return self._graph[uid]['this']

    def _search(self, person_uid, friend_uid):
        """
        Busca em largura no grafo de relacionamentos entre `person_uid` e `friend_uid`.

        A busca foi implementada usando uma lista de `fronteira` que armazena todos os v
        sendo avaliado. A lista `visitados` armazena todos os vértices que já foram expa
```

```
        expandidos novamente. Com isso, são evitados os problemas de ciclos no grafo.

        A função retorna uma lista menor caminho entre os dois vértices ou lança uma exc
        caminho entre eles.

        :param person_uid:          estado inicial do grafo
        :param friend_uid:          estado final do grafo
        :return:                    lista com menor caminho entre o estado inicial e fin

        :raise RuntimeException:    erro quando não existe uma conexão entre as duas pes
        """
        from collections import namedtuple
        # print('='*10)
        # print('pessoa:', person_uid, self.get_person_by_uid(person_uid).get_s_type())
        # print('amigo: ',friend_uid, self.get_person_by_uid(friend_uid).get_s_type())
        # print('='*10)

        # Estrutura dos vértices da árvore de busca
        Vertice = namedtuple('Vertice', ['uid', 'pai'])

        fronteira = []      # Lista com todos os vértices a serem explorados
        expandidos = {}     # Lista com todos os vértices já expandidos

        def expandir_fronteira(vertice: Vertice):
            """Adiciona na fronteira todos os vértices vizinhos que ainda não foram expa
            Fronteira modificada para incluir os vértices com sexos alternados.
            """
            vizinhanca = [Vertice(p.get_uid(), vertice)
                          for p in self._graph[vertice.uid]['friends']
                          if p.get_uid() not in expandidos
                              and self.get_person_by_uid(vertice.uid).get_s_type() !=
            fronteira.extend(vizinhanca)

            # Garante que os vértices que já entraram na fronteira não sejam expandidos
            for vizinho in vizinhanca:
                expandidos[vizinho.uid] = vizinho

        # Expande a raiz da árvore
        raiz = Vertice(person_uid, None)

        expandidos[raiz.uid] = raiz
        expandir_fronteira(raiz)

        # Realiza o processo de busca até encontrar a solução
        while fronteira:
            # Remove o primeiro vértice da fronteira
            #    - FIFO para a busca em largura
            #    - LIFO para a busca em profundidade
            vertice = fronteira.pop(0)

            # Verifica se encontrou o estado final
            if vertice.uid == friend_uid:

                # Recupera o caminho entre o vertice atual e a raiz
                caminho = []
                while vertice.pai != None:
                    caminho.append(vertice.uid)
                    vertice = expandidos[vertice.uid].pai
                caminho.append(raiz.uid)
                caminho.reverse()

                # for uid in caminho:
                #     print(uid, self.get_person_by_uid(uid).get_s_type())

                # print(caminho)
                # print(len(caminho) - 1)
```

```
                # print()
                return caminho

            # Expande a fronteira a partir do vertice atual
            else:
                expandir_fronteira(vertice)

        # Não encontrou solução - neste caso, qualquer valor retornado vai bagunçar com
        raise RuntimeError(f'Não existe conexão entre {person_uid} e {friend_uid}!')

    def get_separation_degree(self):

        total_paths_len = 0
        for _ in range(100):
            person_uid, friend_uid = random.sample([*self._graph.keys()], 2)
            path = self._search(person_uid, friend_uid)
            total_paths_len += len(path) - 1

        return total_paths_len / 100
```

Teste #2

In [5]:
```
import numpy as np
from math import sqrt

n_vertices = [100, 1000, 10000, 100000]

print(f"+-{'-'*7}-+-{'-'*9}-+-{'-'*9}-+-{'-'*9}-+")
print(f"| {'n':<7} | {'5':^9} | {'sqrt(5)':^9} | {'n/5':^9} |")
print(f"+-{'-'*7}-+-{'-'*9}-+-{'-'*9}-+-{'-'*9}-+")

for n in n_vertices:
    n_arestas = [int(5 * n), int(sqrt(n) * n), int(n / 5 * n)]

    medias = []
    for a in n_arestas:
        if a == 2_000_000_000 and n == 100_000:
            medias.append(-1)
            break

        friend_network = FriendNetwork(n, a)

        resultados = []
        resultados.append(friend_network.get_separation_degree())

        medias.append(np.mean(resultados))

    print(f'| {n:<7} | {medias[0]:^9.2f} | {medias[1]:^9.2f} | {medias[2]:^9.2f} |')

print(f"+-{'-'*7}---{'-'*9}---{'-'*9}---{'-'*9}-+")
```

```
+---------+-----------+-----------+-----------+
| n       |     5     |  sqrt(5)  |    n/5    |
+---------+-----------+-----------+-----------+
| 100     |   3.13    |   2.37    |   1.98    |
| 1000    |   4.67    |   2.60    |   2.06    |
| 10000   |   6.05    |   2.61    |   2.14    |
| 100000  |   7.33    |   2.62    |   -1.00   |
+-----------------------------------------------+
```

In [ ]:
```

```