Problem 1. Consider the SIR-model from class, with permanent immunity and recovery rate $\delta = 0.1$. Plot a simulation of the infection process (S, I, R denoted as the fraction of people in each group) in a population of 1000 people, and the transmission rate of 0.2. The initial recovery group size is 0, and the initial infected group size is 1. (Hint: You might want to use first- order differential equation solver in your language. Or just calculate the size in an iterative way)

Solution. The code is attached in `./src/ex1.py`
Note that β means the transmission rate while δ denotes the recovery rate.

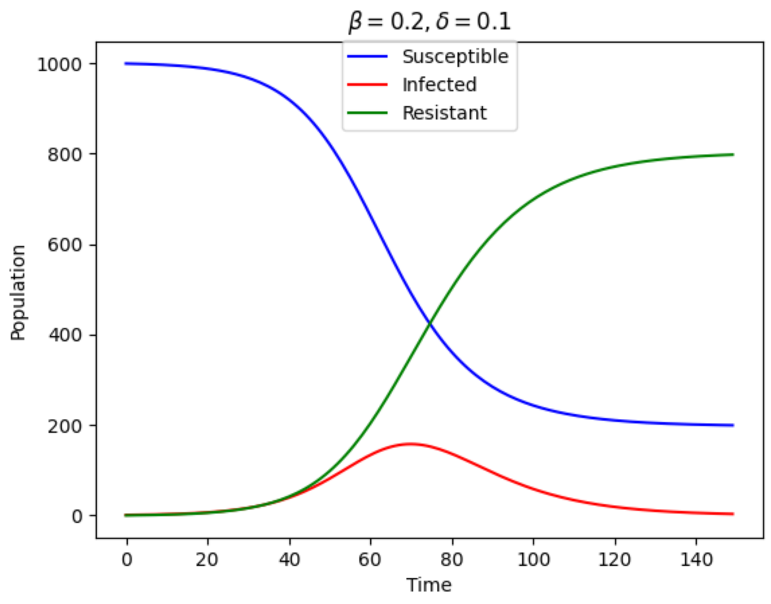

Figure 1: SIR plot

Problem 2. Suppose we have eight employees at a small company, where their relationship score can be measured as values between -100 to +100. A value of zero means the two employees haven't interacted or are indifferent. Each employee is further asked to choose 3 movies that they would most enjoy watching for the upcoming company movie night.

1. Build a friend (with friendship value greater than zero) network, and write a program to plot the degree distribution graph of this friend graph.

   The implementation for basic structure of vertex and edge is in `./src/util.py` The following code utilises the structure and create a friend network while plotting the degree distribution histogram.

```python
from util import Edge
from util import Vertex
import numpy as np
import matplotlib.pyplot as plt

class FriendNet:
    def __init__(self):
        self.mapping = dict() ## map name to id
        self.employee_node = []
        self.employee_name = []
        self.friends = []
        self.num_friend = 0
        self.num_employee = 0
        self.degree = []

    def AddEmployee(self, name):
        if name not in self.employee_name:
            new_employee = Vertex(name)
            self.employee_name.append(name)
            self.employee_node.append(new_employee)
            self.mapping[name] = self.num_employee
            self.num_employee += 1
            self.degree.append(0)
        else:
            print('This employee is already in the network!')

    def AddFriend(self, name_a, name_b, score):
        if name_a not in self.employee_name:
            self.AddEmployee(name_a)
        if name_b not in self.employee_name:
            self.AddEmployee(name_b)

        friendship = Edge(name_a, name_b, score)
        if friendship not in self.friends:
            self.friends.append(friendship)
            index_a = self.mapping[name_a]
            index_b = self.mapping[name_b]
            self.degree[index_a] += 1
            self.degree[index_b] += 1
            self.num_friend += 1
        else:
            print('They are already friends!')

    def plot(self):
        fig = plt.figure()
        x = np.arange(8)
        plt.bar(self.employee_name, height=self.degree)
        plt.xticks(x, self.employee_name)
```

```python
        plt.xlabel('Employees')
        plt.ylabel('Degree')
        plt.legend(loc="best")
        plt.savefig('Degree.png')
        plt.show()

if __name__ == '__main__':
    Mynet = FriendNet()
    f = open("Employee_Relationships.txt")
    lines = f.readlines()
    for line in lines:
        score = int(line.split()[-1])
        if score > 0:
            EmployeeA = line.split()[0]
            EmployeeB = line.split()[1]
            Mynet.AddFriend(EmployeeA, EmployeeB, score)
    f.close()
    Mynet.plot()
```

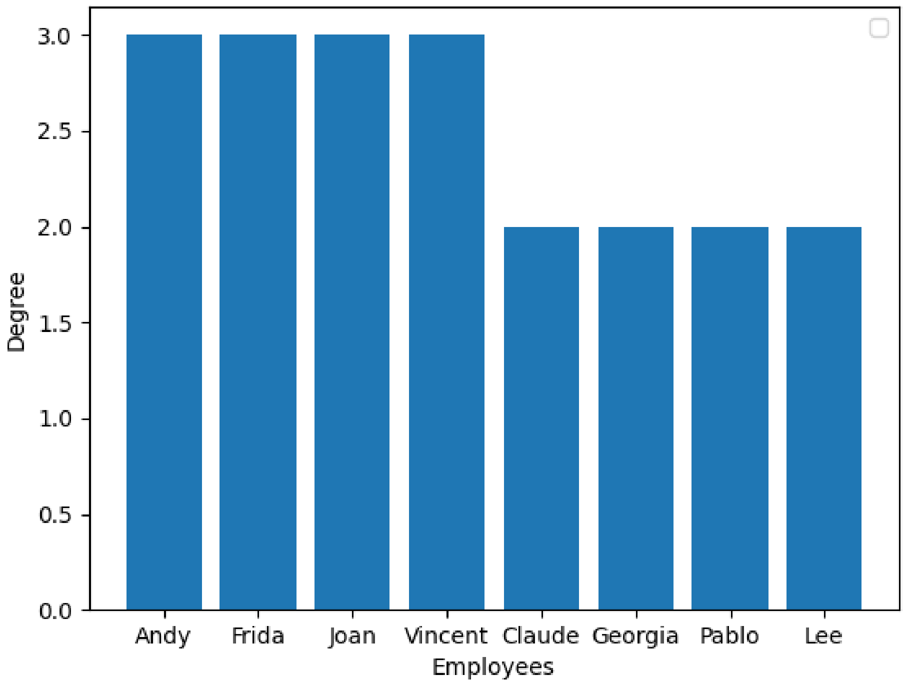

Figure 2: Degree Distribution Histogram

2. Implement the greedy algorithm we introduced in the class to generate a list of movies to display in the movie night, so that each employee could at least have one movie they like, and the total number of movies that needs to be displayed is also minimized.

Solution. There are several possible results while providing a list of 3 movies, like

```
['The Shawshank Redemption', 'Mean Girls', 'Snakes on a Plane']
['Anaconda', 'The Shawshank Redemption', 'Mean Girls']
['Anaconda', 'Forrest Gump', 'The Shawshank Redemption']
```

In my implementation, in line 12, the `ramdom` package will shuffle the order of the movie pool list, such that every time the program might be looking for different list of moives that satisfies the requirements since there are many cases the activated number of nodes is equal. The related file is in `./src/ex2-2.py`

```
1  from collections import defaultdict
2
3  def HillClimbing(pool,choice,satisfied):
4      result = []
5      best = ''
6      now_satisfied = 0
7      real_satisfied = satisfied.copy()
8
9      while now_satisfied != 8:
10         ## Initialize max before every round, update after each movie
11         max_satisfied = 0
12         random.shuffle(pool)
13         for movie in pool:
14             ## Before comparison of movies, update
15             num_satisfied = 0
16             satisfied = real_satisfied.copy()
17
18             for person in choice[movie]:
19                 if not satisfied[person]:
20                     satisfied[person] = True
21                     num_satisfied += 1
22             if num_satisfied > max_satisfied:
23                 max_satisfied = num_satisfied
24                 best = movie
25
26         result.append(best)
27         now_satisfied += max_satisfied
28         for person in choice[best]:
29             real_satisfied[person] = True
30         if best in pool:
31             pool.remove(best)
32     return result
33
34 def getChoices():
35     map = defaultdict(list)
```

```
    satisfied = {}
    f = open("Employee_Movie_Choices.txt")
    lines = f.readlines()[1:]
    for line in lines:
        person = line.split('\t', 1)[0]
        movie_name = line.split('\t', 1)[1].rstrip()
        satisfied[person] = False
        if person not in map[movie_name]:
            map[movie_name].append(person)
    f.close()
    return map, satisfied

def getMovies():
    names = []
    f = open("Employee_Movie_Choices.txt")
    lines = f.readlines()[1:]
    for line in lines:
        movie_name = line.split('\t',1)[1].rstrip()
        if movie_name not in names:
            names.append(movie_name)
    f.close()
    return names

if __name__ == '__main__':
    pool = []
    choice = defaultdict(list)
    satisfied = {}
    pool = getMovies()
    choice,satisfied = getChoices()

    result = HillClimbing(pool,choice,satisfied)
    print(result)
```

Problem 3. Suppose we have 6 nodes in the network which forms a directed graph with 9 edges. The connection relationship is given as source node set s = [1 1 2 2 3 3 3 4 5], and destination node set t = [2 5 3 4 4 5 6 1 1], where edge i is from s[i] to t[i].

1. Display the adjacency matrix M of this directed graph, where element $M_{ij} = 1$ if there is an edge from node i to node j.

   Solution. The idea is simple, just need to create a 6 by 6 adjacency matrix. The output is given as follows and the code is in `./src/ex3.py`

```
def Adjacency_Display(s,t):
    res = [[0 for j in range(6)] for i in range(6)]
    for i in range(9):
        res[s[i]-1][t[i]-1] = 1
    for j in res:
        print(j)

if __name__ == '__main__':
    s = [1, 1, 2, 2, 3, 3, 3, 4, 5]
    t = [2, 5, 3, 4, 4, 5, 6, 1, 1]
    Adjacency_Display(s,t)


**************** The sample output ****************

                [0, 1, 0, 0, 1, 0]
                [0, 0, 1, 1, 0, 0]
                [0, 0, 0, 1, 1, 1]
                [1, 0, 0, 0, 0, 0]
                [1, 0, 0, 0, 0, 0]
                [0, 0, 0, 0, 0, 0]
```

2. We introduced in page49 of slide: 12-web_pagerank.pdf and display the pagerank score for each node. The random teleport probability β is set to be 0.85.

Solution. Based on the previous adjacency matrix, it's easy to implement the pagerank algorithm, while the code can still be found at `./src/ex3.py`, there is something new:

(a) A `getDegree()` function to retrive all the degree of the nodes from the graph.

(b) A `PageRank()` function utilizing random walk.

```
def PageRank(AdMatrix, degree, beta=0.85):
    # beta is the probability of following a link, set 0.85 by
    ↪ default
    rankscore = [1/6 for i in range(6)]
    old_rankscore = rankscore[:]
    N = 6
    j = 0
    while True:
        i = 0
        curr_score = 0
        for row in AdMatrix:
            if row[j] == 1:
                curr_score += beta*(rankscore[i]/degree[i])
            i += 1
        rankscore[j] = ((1-beta)/N) + curr_score
        j += 1
        if j == 6: # finish one round updating
            j = 0  # start new round updating
            if rankscore == old_rankscore: # converge
                break
            else:
                # update a copy to check if converge
                old_rankscore = rankscore[:]
    return rankscore

def getDegree(AdMatrix):
    deg = [0 for i in range(6)]
    for i in AdMatrix:
        for k in range(6):
            if i[k] == 1:
                deg[k] += 1
    # print(deg)
    return deg

**************** The sample output ****************

NodeName     PageRankScore
1            0.2190908459347428
2            0.1181136095222657
3            0.12539656809392585
4            0.2319836509737628
5            0.22470069240210264
6            0.13158708287983698
```

Some thoughts. In my implementation, the pagerank algorithm would update each node's value and update the rest nodes one by one, this might lead to a problem that another node's rank will based on the updated rank of previous node, therefore the answer might be different compared with the conventional approach that utilizing matrix multiplication, which will update the rank for all the nodes in one iteration.