Python Project Adjacency

Assignment Overview
This assignment focuses on the implementation of Python programs using classes. We reuse the concept of an adjacency matrix from Project Facebook and set intersection from Project Co-occurrence.

Assignment Background

SATPro TAs are to be distributed among many help rooms running simultaneously. Each TA can serve the room they are in AND the neighboring rooms. Here neighboring rooms to a room are the ones where there is a direct path (connection) between the rooms. You are given a map of SATPro help rooms and your task is to determine the minimum number of TAs that are needed to serve all of the help rooms.
Your program takes as input: the total number of SATPro help rooms, the number of paths (connections) between help rooms, and each connection in the following input format. Connections are bidirectional in the sense that if we specify a connection between rooms 2 and 3 that implies that there is a connection both ways: from 2 to 3 and 3 to 2.

Example 1: Consider this map of the SATPro help rooms: 1—2—3—4—5—6

Test 1
Input file `test1.txt`:

```
6
1 2
2 3
3 4
4 5
5 6
```

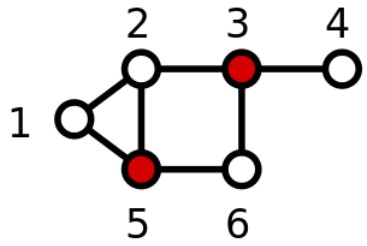
Output:

```
Enter a file name: test1.txt
TAs needed: 2
TAs assigned to rooms: 2, 5
```

Explanation:

We can place a TA in room 2 and room 5. The TA in room 2 can serve rooms 2, 1 and 3, and the TA in room 5 can serve rooms 4, 5 and 6. One TA is insufficient.

Example 2: Consider this map of the SATPro help rooms:



Test 2:
Input file `test2.txt`

```
6
1 2
1 5
2 3
2 5
5 6
3 6
3 4
```

Output:

```
Enter a file name: test2.txt
TAs needed: 2
TAs assigned to rooms: 3, 5
```

Explanation: TA in room 5 can serve room 5 and 1. TA in room 3 can serve rooms: 2, 3, 4 and 6. Note that there are multiple other possible explanations such as the TA in room 5 can serve rooms 1, 2, 5, and 6 while the TA in room 3 can serve rooms 3 and 4. One TA is insufficient in either case. The result is the same: 2 TAs are needed. Also, there are other, equally valid solutions such as placing the 2 TAs in rooms 1 and 3. You only need to determine one valid solution.

Implementation

1. You are required to implement your solution using classes.
2. Use what is known as the Greedy Algorithm to find the minimum number of TA. In this case a Greedy Algorithm is:
    a. Try to solve the problem using 1 TA.
    b. If 1 TA can cover all rooms, you are done.
    If 1 TA cannot reach all rooms, try 2 TAs.
    c. Then try 3 TAs and so on until you find that N TAs together can cover all rooms.
3. What does it mean to "solve the problem using 3 TAs"?
    Try every *combination* of mapping 3 TAs to the rooms. Finding all combinations to try is

messy, but we can `import itertools` and use
`itertools.combinations(list_of_rooms, TAs)`
The first argument is a list of ints (list of room numbers), the second argument is an int (number of TAs). For example try the following in the shell:

```
print( list(itertools.combinations([1,2,3,4,5],3)))
```

which produces the following output that represents every way you can place three TAs into five rooms. That is: (2,3,5) means that there is a TA in each room: 2, 3, and 5.

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)]
```

4. How do you find out if a TA in each room of (2,3,5) solves the problem?
Remember in social networks where you created a list of lists representing who was friends with whom? That list of lists is called an *adjacency matrix* and we can use a similar data structure here to represent the connections among rooms. Here it is best to use a **list of sets**. With the adjacency matrix in hand, sets are useful for collecting the rooms that TAs can cover: walk through each tuple of rooms (from step 3 above) and add (using set *union*) the rooms to a set (consult the adjacency matrix to determine connected rooms). If you can reach all the rooms (what Boolean tells you that?), you have found a solution. Hint: remember to consider the rooms that the TAs are in.
5. The adjacency `Matrix` class. You are to construct a class named `Matrix` to implement your adjacency matrix. A framework for the class is provided in `Proj_Adj.py`. Two class methods are complete: `__init__` and `__repr__`. You need to implement the other methods. You may modify `__init__` if you wish, and you may add additional methods (but these should be sufficient). The methods you need to implement are:
    a. `read_file`: this method takes a file pointer as a parameter. It reads a file and fills in the matrix named `self._matrix`. The first line of the file is an integer that is the number of rooms. Each subsequent line contains two integers separated by spaces. The pair of integers indicates a connection between a pair of rooms and the connection is bi-directional. That is, the line `2 3` indicates that 2 is connected to 3 and symmetrically 3 is connected to 2. Hint: the room numbers start at 1 whereas list indices start at 0 so you need to account for that (you already knew that, right?).
    b. `__str__`: this method returns a string that represents the matrix. If you have an instance of a matrix named M, then `print(M)` calls `__str__` and prints the string returned by `__str__`. **Important**: your `__str__` method CANNOT have any `print` statements—your method builds a string that a `print` statement will use.
    c. `adjacent`: this method takes a parameter representing a room and returns one row of your matrix (that row is a set). That row represents the rooms that are adjacent to the room specified by the parameter `index`.
    d. `rooms`: this method returns the number of rooms (rows) in the matrix.
6. `open_file`: It should keep prompting until a file is opened.

7. `main()` The `main` function of your program opens a file, creates an instance of the adjacency matrix, call the matrix `read_file` method to fill the matrix and then loops to find the minimum number of TAs and the rooms they cover. Note that when you find a combination of assigning TAs to rooms that works you will likely be deep within a couple of loops so a simple `break` will not be sufficient. I used a Boolean named `done` that controlled multiple `breaks` to exit the multiple loops. Finally, print the adjacency matrix.
   Call to main required to be:
   ```
   if __name__ == "__main__":
       main()
   ```

Sample Output

Function Test

```
Testing __str__ by calling __str__ through print():
1: 2
2: 1 3
3: 2 4
4: 3 5
5: 4 6
6: 5

Testing adjacent method:
rooms adjacent to room 2: {1, 3}
Testing rooms method:
number of rooms: 6
```

Test 1

```
Enter a file name: test1.txt
TAs needed: 2
TAs assigned to rooms: 2, 5

Adjacency Matrix
1: 2
2: 1 3
3: 2 4
4: 3 5
5: 4 6
6: 5
```

Test 2

```
Enter a file name: test2.txt
TAs needed: 2
TAs assigned to rooms: 1, 3

Adjacency Matrix
1: 2 5
2: 1 3 5
3: 2 4 6
4: 3
5: 1 2 6
6: 3 5
```

Test 3

```
Enter a file name: test3.txt
TAs needed: 3
TAs assigned to rooms: 2, 5, 7

Adjacency Matrix
1: 2
2: 1 3 5
3: 2 4
4: 3 7
5: 2 6
6: 5
7: 8 4
8: 7
```