Assignment

In this assignment we're going to explore the Ice Breaker problem: given a group of n people, what's the minimum number of teams they can be partitioned into such that no team has 2 (or more) people who are friends. The hope is that this encourages people to make new friends.

We will assume for this problem that the n people are named 0, 1, 2, …, $n-1$, and that we have what we will call a friendship graph, i.e. a graph with n nodes labeled 0 to $n-1$ (each node is a person), where nodes i and j are connected by an edge just when i and j are friends. Friendship is symmetric: if i is a friend of j, then j is a friend of i. Plus a person cannot be friends with themselves.

This assignment permits solutions where a team can have only 1 person on it. While this is not a very practical assumption for a real ice breaker event, we will use it here to simplify things.

Do not use any modules or code except from the standard Python 3 library, or from the textbook code from Github.

Solving the Ice Breaker Problem with CSPs

The basic solution you should use in this assignment is a CSP one variable for each of the n people: $X_1, \ldots, X_n$. The domains for these variables are team names, which will be integers 0, 1, 2, … etc. All the domains are the same. The constraints on the CSP are all not-equal constraints that come from the friendship graph: if person i and j are friends, then we have the constraint $X_i \neq X_j$, i.e. two friends cannot be on the same team.

The problem is to find the smallest number of teams that satisfies all the constraints. In other words, we want to find the smallest domain size for the variables that satisfies all the constraints of the friendship graph.

The word smallest is very important here. If we were satisfied with any number of teams, then we could just put each person on a team by themselves.

Using csp.py

For this assignment, use the code in `csp.py` from the textbook code. You can modify this code if you want to, but do not change the input or output formats for any of the functions.

The format of friendship graphs is a dictionary where the keys are the people (integers from 0 to $n-1$), and the corresponding values are their friends in a list. For example, here is a 4-person friendship graph:

```python
g = {0: [1, 2], 1: [0], 2: [0], 3: []}    # format of graphs for this assignment
```

This graph has 4 people, named 0 to 3, and it looks like this:

```
0---2
|
|
1   3
```

0 and 2 are friends, and 0 and 1 are friends. Person 3 is not friends with anyone. In Python, `g[0]` is the list of all people 0 is friends with, `g[1]` is the list of all people 1 is friends with, and so on.

To solve this ice breaker problem, we use three CSP variables: X_0, X_1, X_2, X_3. The constraints correspond to edges in the graph, and for `g` they are: $X_0 \neq X_1, X_0 \neq X_2$.

It is not hard to see for this graph that 2 teams are enough, e.g. put 0 and 3 on team 0, and 1 and 2 on on team 1. As a CSP solution, this would be written $X_0=0, X_1=1, X_2=1, X_3=0$. In Python, it would be represented as this dictionary:

```python
X = {0:0, 1:1, 2:1, 3:0}
```

Make sure that graphs and CSP solutions have exactly the format described here!

Question 1 (warm-up: Erdos-Renyi random graphs)

Create a function called `rand_graph(n, p)` that returns a new random graph with \(n\) nodes numbered 0 to \(n-1\) such that every different pair of nodes is connected with probability \(p\). Assume \(n > 1\), and \(0 \leq p \leq 1\).

For example:

```
>>> rand_graph(5, 0.5)
{0: [2], 1: [2, 4], 2: [0, 1, 4], 3: [], 4: [1, 2]}

>>> rand_graph(10, 0.1)
{0: [4, 7], 1: [3, 6], 2: [9], 3: [1, 6], 4: [0, 5],
 5: [4], 6: [1, 3], 7: [0], 8: [], 9: [2]}
```

Notice that if `a` appears in the list for key `b`, then `b` also appears in the list for key `a`.

The higher the value of \(p\), the more edges the resulting graph will have.

Put all your code for this into a file named `a2_q1.py` so that the marker can test it.

Question 2 (warm-up: Checking Solutions)

Write a function called `check_teams(graph, csp_sol)` that returns `True` if the given CSP solution dictionary `csp_sol` satisfies all the constraints in the friendship `graph`, and `False` otherwise. `graph` and `csp_sol` are dictionaries formatted as described above.

Do not use any code from `csp.py` to implement `check_teams`. The idea is to use `check_teams` to double-check the correctness of your results in the questions that follow.

Remember that teams of 1 person are permitted.

Put all your code for this into a file named `a2_q2.py` so that the marker can test it.

Question 3 (Exact)

For \(n=30\), generate 5 random friendship graphs as follows:

```
graphs = [rand_graph(30, 0.1), rand_graph(30, 0.2), rand_graph(30, 0.3),
          rand_graph(30, 0.4), rand_graph(30, 0.5)]
```

For each of these friendship graphs, calculate the exact minimum number of teams that the people can be put into such that no team has 2 (or more) people on it who are friends.

Important: This question asks for an exact answer, so make sure the method you are using to solve this problem guarantees this!

Note: If you like, you can calculate answers for graphs with \(p\) values greater than 0.5, but the running times might be very slow.

Repeat the above step at least 5 times, using 5 different random graphs each time (make sure to use the same \(p\) values for each set of 5 graphs). For each solution, keep track of:

- the number of teams that the people are divided into
- the running time of the solver (don't include the time to create the graphs)
- the count of the number of times CSP variables were assigned and unassigned
- at least one other piece of information (your choice) that you think is useful for helping to understand the performance of the algorithm; choose something useful!

When you are done, you should have 25 different solutions, with the above data recorded for each solution.

In an Excel worksheet named `a2_q3.xlsx` (e.g. use Excel, or Google Sheets to make it), make a neat and easy to understand table summarizing all the data you collected. You should include things like averages for the 5 runs for each \(p\) value, and a chart/graph to help visualize your results.

Put all your code for this into a file named `a2_q3.py`, and include a function called `run_q3()` that the marker can, if they like, call to re-run your experiment.

Question 4 (Approximate)

Re-do the above experiment, but this time for \(n=100\). For such a large graph, exact values are going to be hard to calculate in a reasonable amount of time, so you can (should!) use an algorithm that returns an approximate solution, i.e. something close to the smallest number of teams for a given friendship graph, but not necessarily the smallest.

Put your results into an Excel worksheet named `a2_q4.xlsx`. Put all your code for this question in a file named `a2_q4.py` and include a function called `run_q4()` that the marker can, if they like, call to re-run your experiment.

What to Submit

For this assignment you should submit at least these 6 files:

- `a2_q1.py`
- `a2_q2.py`
- `a2_q3.py` and `a2_q3.xlsx`
  - make sure `a2_q3.py` contains a function called `run_q3()` that runs the code that generates your data
- `a2_q4.py` and `a2_q4.xlsx`
  - make sure `a2_q4.py` contains a function called `run_q4()` that runs the code that generates your data

Make the spreadsheets beautiful, informative, and easy to read. Be sure to include helpful descriptive statistics like the min, max, average, and median values. You are encouraged to include helpful or informative graphs of your data. Spelling, grammar, and neatness count!

In addition, don't modify any of the files in the textbook code. The markers will be running your code using the textbook software as-is. If you do want to modify textbook code, copy the code you want to change into the appropriate `.py` file this assignment asks for, and make the changes there.

As with the first assignment, do not use any modules or code except from the standard Python 3 library, or from the textbook code from Github.

Put all the files needed to re-run your questions into a single `.zip` archive named `a2.zip`, and submit it on Canvas before the due date listed there.

Hints

- At the Linux command-line, you can use the `zip` command to compress a folder. For example, if you do all your work in a folder named `aima-a2`, then this creates `a2.zip`:

  ```
  zip -r a2.zip aima-a2
  ```

  The `-r` option causes `zip` to recursively include all folders in the resulting `.zip` file.
- Remember, teams of 1 person are permitted. This is just a simplifying assumption for this assignment, and is not realistic.
- Note that there are a couple of extreme cases to be aware of in the ice breaker problem:
  - If no one is friends with anyone else, then only 1 team — the entire group of \(n\) people — is needed.
  - If everyone is friends with everyone else, then \(n\) teams are needed. Everyone would have to be on a team by themselves.

    - If there is someone who is friends with everyone else, then they must be on a team by themselves.
    - Someone who is friends with no one can be added to any team.
- In the file `csp.py` from the textbook code, there is a very useful class called `MapColoringCSP`.
- The ice breaker problem is really just asking you to find the chromatic number of a graph.