CAB203
Graphs Project

1 Introduction

In this project will demonstrate your knowledge and skill with the material in the unit by developing a solution to a real world problem by translating it to mathematical language, relating the problem to well known problems on mathematical structures, and implementing software to solve the problem.

Your submission will consist of two parts: a report detailing the mathematical descriptions of the problem and solutions, and a Python file containing your implementation of the solution.

This assignment is worth 40% of your final grade.

Please see the Canvas Assignments page for due dates.

2 Scenario

Congratulations on your acceptance to the organising committee of the 2024 CAB203 Bonkers tournament! Bonkers is an extremely exciting and fictitious game.

There are many tasks that need to be done, which have been distributed amongst the organising committee members. Your acumen with discrete structures is well known, and hence you have been assigned several organisational tasks which will benefit from some computerisation.

All tasks are intended to make significant use of graph theoretic tools and problems.

1. The tournament structure needs to be set; that is to say, who plays who. For this tournament a structure is used where all games are set at the beginning. Another committee member will propose the tournament structure and your job is to determine whether it has the required properties. The committee has determined that the following properties are sufficient:

   - For every pair of distinct players, either they play against each other, or there are at least two other players that they both play against.
   - All players have the same number of games.

   So for example, if Alice and Bob don't play against each other then it's still OK as long as they both play against two players in common, say, Charlie and Denise.

   Your job is to find a method that, given a tournament structure, determines whether the above properties hold.

   Implement your method in a Python function like so:

```
def gamesOK(games):
```

where `games` is a set of proposed games to play, given as pairs of players, like so:

```
games = { ("Alice", "Bob"), ("Charlie", "Bob") }
```

Note that the order of the players in a particular game does not matter: `('Alice','Bob')` specifies the same game as `('Bob','Alice')`. You may assume that in the games given no player plays against themselves, every player plays in at least one game, there is at least one player, and no game is given more than once.

Your function should return `True` if the required property holds for the games given, otherwise `False`.

2. All Bonkers games must have a referee, but the referee for each game cannot have any conflicts of interest for the game. In particular, the referee cannot be a player in the game, and the referee must also declare any other conflicts of interest (eg. cannot be a player's relative, friend, boss, etc..). In order to minimise the burden on any individual referee, the policy is to assign each referee at most one game.

   Your task is to determine a method for assigning referees to games, given the games, referees, and conflicts of interest for each referee. You must assign at most one game to each referee, and exactly one referee to each game, or determine that it is impossible to do so.

   Implement your solution as a Python function of the form

   ```
   def referees(games, refereecsvfilename):
   ```

   `games` is structured the same as in the previous question. `refereecsvfilename` is the name for a CSV file which you should read to obtain the list of referees and their declared conflicts of interest. Below is a sample CSV file showing the structure.

   ```
   Referee,Conflict1,Conflict2,Conflict3,Conflict4
   Alice,Bob,Dave,Eve
   Bob,Dave,Eve,Fiona,Jamal
   Fiona
   ```

   You may assume that for each referee there is exactly one row giving their conflicts of interest.

   Your function should return a data structure of games and assigned referees similar to:

   ```
   # dictionary of all (player1, player2) : referee
   { ('Ashley', 'Bob'): 'Rene' }
   ```

   or `None` if it is not possible.

3. The games now need to be scheduled, which will happen in two steps. Your first task is to group together games that can be played simultaneously into *game groups*. The requirements for the game groups are:

   - People are not double-booked: each person is involved in at most one game in any game group., either as a player or referee
   - Games are played once: each game is in exactly one game group

   Each game group will be played in a single time slot, so the committee wants the smallest number of game groups possible.

   Implement your solution as a Python function of the form

   ```
   def gameGroups(assignedReferees):
   ```

   where `assignedReferees` is in the same format as you returned in the previous question, i.e. as a dictionary with games as keys and referees as values. Your output should be a schedule in a data structure of the form

```
# list of timeslots
# each timeslot is a set of games
# each game is a pair (player1, player2)
[
   { ("Ashley", "Bob"), ("Charlie", "Dave") },
   { ("Bob", "Charlie"), ("Rene", "Elaine") }
]
```

4. The previous question only dealt with which games can be played at the same time, but not on the order of the game groups. The preference is that if a player is also a referee then they play all of their games before refereeing. Your task is to determine if it is possible to order the game groups so that this is true.

   Implement your solution as a Python function of the form

   ```
   def gameSchedule(assignedReferees, gameGroups):
   ```

   where `assignedReferees` is as in the previous question, and `gameGroups` is of the form of the return value in the previous question. Your function should return the game groups in the required order, in the same format as the return value in the previous question, or `None` if there is no such order.

5. Once all games are played, it is necessary to find the tournament winners. For this, Bonkers uses a points based system. The system is a little technical, so we first provide some intuition for it.

   Since not all players play against each other in Bonkers, a points based system based on the number of wins alone is not sufficient. Perhaps Alice and Bob have the same number of wins, but Alice played against skilled opponents and Bob did not. So the number of wins alone doesn't indicate a player's skill very well. To compensate for this, players gain points for their own wins, and also for the wins of players that they defeated. For example, if Alice defeats Bob and Bob defeats a lot of other players (say, Charlie, Dave, Evalyn) this probably means that Bob is skilled, and hence Alice must also be skilled to defeat Bob. Hence Alice earns some points for Charlie, Dave and Evalyn as well. Now, we need to put some limits on these extra points. In the previous example, if Alice defeated Charlie then she will get points for that win directly; she shouldn't also get points for Bob defeating Charlie. Also, suppose that Bob is quite skilled and defeated many other players. Alice can access those points if she plays Bob, but another player (say, Fred) who doesn't play Bob wouldn't have access to those points, even if he is as skilled as Bob. If Fred only plays against weak players then he is at a disadvantage. For this reason we limit the number of points that Alice can claim for Bob's wins. Finally, suppose that Alice defeats Bob who defeats Charlie, and Alice also defeats Dave who defeats Charlie. Charlie is apparently not very skilled, so it isn't fair for Alice to gain points for both Bob defeating Charlie and Dave defeating Charlie. So Alice gets the same amount of points from Charlie as if only Bob defeated Charlie. Put differently, Alice gets points once for second-hand defeating Charlie, not for each way that the second-hand defeats him. But, because there is the limit on points that Alice gets through Bob, she can get some of the points for Charlie through Bob and the rest through Dave.

   Traditionally, each player's points are calculated through a *scoring game*, which works as follows.

   (1) The player writes out the names of all the players that they defeated (called *primary wins*) out on a big piece of paper. Above them the player writes all the names of all the players that the primary wins defeated (called *secondary wins*). They also draw a line from each primary win to each secondary win that the primary win defeated.

   (2) A referee places p tokens on the paper on name of each primary win.

   (3) The referee places s tokens on the paper on name of each secondary win.

   (4) The player now moves tokens one at a time along lines from secondary wins to primary wins. They can choose which lines to follow with each token.

(5) While doing the above, each primary win can have at most c tokens on it. (This is sometimes done by drawing c circles around each primary win so that it is easy to see if there is capacity to add more tokens.) This means that it may happen that not all tokens on secondary wins can be moved to primary wins.

(6) The player collects tokens on the primary wins counts them to determine their score.

Here is an example to illustrate the above rules based on a partial set of wins. We set $p = 4, s = 3, c = 5$:

- Alice defeats Bob, which is a primary win for Alice.
- Bob defeats Charlie, which is a secondary win for Alice against Charlie via Bob (Bob is the intermediate player.)
- Similarly, Alice defeats Dave (primary win), and Dave defeats Charlie (a secondary win for Alice.)
- Bob defeats Dave. This is not a secondary win for Alice because Dave is already a primary win.

Alice plays the scoring game as follows:

(a) Alice writes "Bob" and "Dave" on the paper for primary wins, and "Charlie" for secondary wins.

(b) Alice draws lines from "Charlie" to "Bob" and to "Dave".

(c) The referee puts 4 tokens on "Bob", 4 tokens on "Dave" and 3 tokens on "Charlie".

(d) Alice moves 1 token from "Charlie" to "Bob". "Bob" now has 5 tokens, so there is no space for more.

(e) Alice moves 1 token from "Charlie" to "Dave". "Dave" now has 5 tokens, so there is no space for more.

(f) Alice decides not to reset.

(g) The referee counts 10 tokens total on "Bob" and "Dave", so Alice's score is 10.

Your task is, given the outcomes of all games in the tournament, determine the maximum possible score for each player. Note that you do not need to implement the scoring game itself, just determine what the maximum score can be. Implement your solution as a Python function of the form


```
def scores(p, s, c, games):
```

where `games` is a data structures similar to:

```
# each pair is a game (winner, loser)
games = { ("Ashley", "Bob"), ("Bob", "Charlie") }
```

Your function should return a dictionary mapping players to points like so:

```
{ "Ashley": 5, "Bob": 4, "Charlie": 4 }
```

3 Report

Your report should have five sections, one for each of your assigned tasks. Each section should discuss the following (preferably in this order)

1. Use mathematical language, concepts and notation from the unit to describe the problem. You should make use of mathematical tools and problems discussed in the unit, for example quantified predicates or finding a shortest path in a graph. Describe the information given in mathematical terms and using mathematical notation (e.g. the games are given a set of pairs of players like $\{(a, b), (c, d)\}$).

2. Describe, using mathematical terms and notation, how to find a solution for a given instance of the problem. If applicable, describe how the information given relates to other mathematical objects that are needed. For example, how are the vertices and edges of a graph related to the given games. Describe how the solution to the mathematical problem relates to the solution to the original problem.

3. Describe your implementation in Python. Mention the role of important variables, library calls (eg. to `graphs.py` or `digraphs.py`), and source of reused code if applicable and any modifications required to it. If you need to make a choice about data structures, explain why your choice. Please note that this section should be about programming considerations, not solution methods, which is the above point.

Additionally, your report should include a bibliography using consistent, appropriate style. Some standard citation styles are explained at The QUT Citation tool, any of which is acceptable. At minimum you should cite at least two sources, including lecture material and a standard reference such as a textbook. The sample report — available on Canvas — demonstrates reasonable citations and bibliography.

Overall, report should be understandable by another student in CAB203 who knows the material but hasn't thought about the tasks. It is not necessary to define terms already used in the unit, but you should point out the significance of particular details about the problem and the choices that you make in modelling and solving it.

Your report will be a single file in PDF format. There is no minimum or maximum page length, but a concise, easy-to-understand report is better than a long wordy report. Five pages is about right, not including diagrams if you have any.

3.1 Python implementation

Your solution should be a reasonable implementation of the mathematical solution described in your report. The problems are all solvable using the Python concepts and syntax used in the unit. You can use additional syntax if you like as long as it is compatible with Python 3.10. One exception is that using loops incurs a penalty (see the marking criteria below.) The purpose of penalising loops is to encourage you to think in terms of mathematical style declarative structures rather than procedures. All tasks are solvable in the intended way without using loops.

You are allowed to use or modify any functions defined throughout the lectures, tutorials, and assignment solutions. Many of these functions and more are collected in Python files `graphs.py` and `digraphs.py`. We prefer that you import these files rather than copying from them; the questions are designed so that you can use the functions directly without modifying them. You can assume that these files are available; there is no need to include them in your submission. Additionally, you are allowed to use the `csv` Python module. Before using other modules, please contact the unit coordinator.

A submission template file is available from the Canvas Assessments module. If your solution includes modified code from the unit, say so in a comment explaining where you obtained it and what modifications you made. One line is enough detail.

A test file is included to help you debug your code, available from the Canvas Assessments module. **Please make sure that you run it before submission.** You can run the test file directly: `python test_project.py`. Mac and Linux users may need to run `python3 test_project.py`. Or, run it through Thonny. Be sure that `test_project.py` is in the same directory as your solution, and that your solution is called `project.py`.

The test file is structured using `unittest`, which is part of the Python standard library. It can be run directly to test your solutions:

```
python test_project.py
```

The tasks are all chosen so that they can be solved with a relatively short function (Matt's solution is 57 lines, excluding comments and blank lines). There is no limit on the length of your program. However, the marking system will impose a time limit of about 5 seconds to avoid problems with infinite loops. This should be plenty of time to solve the tasks given.

Your code submission will be a single Python file.

4 Marking criteria

Your mark is made of two parts. The report is graded out of 30 and the Python code is graded out of 10, for a total of 40. Each mark counts 1% towards your final grade.

4.1 Report

The marking rubric is available on Canvas under Assignments > Graph Project Report.

Note that some criteria are worth more than others.

4.2 Python code

Your Python code will be graded automatically by running test cases. The tests will be similar, but not identical, to those found in `test_project.py`. There will be 30 tests, 6 for each task. Each test is 1/3 of a mark, for a total of 10 marks. For each task, one of the tests checks to see if you have used `for` or `while` loops.

The marking system will use Python 3.10, so if you are using a later version be sure not to use any syntax newer than 3.10.

Your code is not assessed for quality, format, comments, length, etc.. Only the automated tests count for marks.

5 Submission

Submission process: You will need to make two submissions through two separate links in Canvas:

- Your report, in PDF format (extension `.pdf`).

- Your Python code, as a Python file (extension `.py`).

You can find the submission pages on Canvas on the Assignments page.

Extensions: Information about extensions is available on the About Assessments module on Canvas. If you obtain an extension, you may *optionally* attach confirmation *as a separate file* to your report submission.

Citing your sources: You are welcome to source information and code from the internet or other sources. However, to avoid committing academic misconduct, you must cite any sources that you use. See https://www.citewrite.qut.edu.au/cite/ for guidelines on citing sources and how to properly format and acknowledge quoted material.

You are welcome to use resources, including code, from within the unit. Please cite the unit like *CAB203, Tutorial 7* or similar. This is only necessary when explicitly quoting unit material. There is no need, for example, to cite the definition of a graph or similar, unless you are directly quoting the lecture's definition.

For code, please include your citation as a comment within the code. For example

```
# modified from CAB203 graphs.py
```


Policy on collaboration: We encourage you to learn from your peers. However, for assessment you need to turn in your own work, and you will learn best if you have spent some time thinking about the problem for yourself before talking with others. For this reason, talking with other students about the project is encouraged, as long as you are putting in the effort on the problems yourself as well. But do not share your code or your report with other students and do not copy from others. It is considered academic misconduct to copy from other students or to provide your work to others for the purposes of copying.

For the purposes of this unit, the use of generative AI (e.g. copilot, ChatGPT) is treated the same as colluding with another person.

For Teams and other online discussions, please do not post about solutions. Keep your discussions private so that everyone gets a chance to get to the solutions on their own. You can direct message or email the unit coordinator or one of the tutors if you wish to discuss specifics of your solution. Feel free, however, to ask general questions about the project on the Teams channels.