CAB203
Special Topics Assignment

1 Introduction

In this assignment you will demonstrate your knowledge and skill with the material in the unit by developing a solution to one real world problem by translating it to mathematical language, using mathematical techniques from the unit to solve the problem, and implementing software to solve the problems.

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

This assignment is worth 30% of your final grade.

Please see the Canvas Assignments page for due dates.

2 Tasks

Choose ***one*** out of the following three tasks to solve.

2.1 Regular languages and finite state automata

It's time to write the ultimate Teams/Slack/Discord replacement, *reChat*! No more resource intensive GUIs, as reChat will be entirely text based with a command line interface.

Your task is to program the controlling logic that parses commands and keeps track of the state of a user's connection. To do this you will write a Python function:

```
def reChatParseCommand(message, state):
    # your code here
    return action, state
```

Here `message` contains the input (i.e. command or message) from the user. `state` is a variable that you can use to store whatever you like. It is initially set to `None` on the first invocation of your function. Further explanation appears below.

The returned variable `action` is a dictionary indicating what action should be taken along with additional parameters for that action, as described below. The returned variable `state` will be used as the `state`

argument on the next time the function is invoked. This mechanism allows you to keep track of the state of the connection without resorting to global variables or other mechanisms. The `state` return value is ignored by the tests.

The required behaviour is given below. Note that text surrounded by `'<>'` like `'<spec>'` in the action should be replaced with an appropriate value (eg. from the command):

- When the user first connects (indicated by the `message` argument set to `''` and the `state` argument set to `None`), the action is `{ 'action': 'greeting' }` and the connection enters *command* mode.
- In *command* mode the user has four possible commands:
  - `\list <spec>` : where `<spec>` is either `channels` or `users`. The connection remains in command mode and the action is
    `{ 'action': 'list', 'param': '<spec>' }`
  - `\quit` : The connection disconnects and the action is
    `{ 'action': 'quit' }`
  - `\join <channel>` : where `<channel>` is a channel name (see below). The connection enters *channel* mode and the action is
    `{ 'action': 'join', 'channel': '<channel>' }`
  - `\dm <username>` : where `<username>` is a username (see below). The connection enters *direct message* mode and the action is
    `{ 'action': 'dm', 'user': '<username>' }`
- In *channel* mode the users has three possible commands. In the following, `<channel>` refers to the channel currently joined by the previous `\join <channel>` command :
  - `\leave` : The connection enters *command* mode and the action is
    `{ 'action': 'leaveChannel', 'channel': '<channel>' }`
  - `\read` : The connection remains in *channel* mode and the action is
    `{ 'action': 'readChannel', 'channel': '<channel>' }`
  - `<message>` (not starting with \) : The connection remains in *channel* mode and the action is
    `{ 'action': 'postChannel', 'channel': '<channel>', 'message': '<message>', 'mentions': { '<username1>', '<username2>, ...' } }`
    where the `mentions` value gives all the usernames that appear in `<message>`.
- In *direct message* mode the users has three possible commands. Note that, in the following, `<username>` refers to the same username given in the pervious `\dm <username>` command, i.e. the user that is being messaged:
  - `\leave`: The connection enters *command* mode and the action is
    `{ 'action': 'leaveDM', 'user': '<username>' }`
  - `\read`: The connection remains in *direct message* mode and the action is
    `{ 'action': 'readDM', 'user': '<username>' }`
  - `<message>` (not starting with \) : The connection remains in *direct message* mode and the action is
    `{ 'action': 'postDM', 'user': '<username>', 'message': '<message>', 'mentions': { '<username1>', '<username2>, ...' } }`
    where the `mentions` value gives all the usernames that appear in `<message>`.

- In all other circumstances the mode does not change and the action is
  `{ 'error': 'Invalid command' }`

The following specifications apply to channels and usernames:

- A username is `@` followed immediately by a valid email address according to the limited criteria from Tutorial 10, Section 2, Question 5b. (You can use the regular expression in the tutorial solutions; there is no need for you to recreate it yourself.)
- A channel name is `#` followed immediately by a sequence of letters (upper or lower case) and numbers, beginning with a letter

Please note the following additional requirements:

- Your program should not use any global variables. All state information should be kept in the `state` argument/return value. Using global variables may cause problems with the testing system.
- While it is entirely possible to perform this task with a bunch of `if` statements, the intention is that you will use functionality analogous to a finite state automaton to keep track of the state of the connection and to provide structure to your code. You may want to read about dispatch tables.
- Your report must include a state change diagram depicting the state changes in the behaviour described above. You do not need to have state change diagrams for regular expressions that you use.

For this task the Python code will be marked automatically according to test cases similar to the sample test cases in `test_STA_fsa.py`. You can run the sample test cases with `python test_STA_fsa.py` while ensuring that your solution is called `specialtopics.py` and is in the same directory as `test_STA_fsa.py`. There are 10 test cases in total, including one test case which checks to see if you have used any `for` or `while` loops. As with the Graphs project, your priority should be to have working code as the loops criterion is only worth 1 mark.

2.2 Linear algebra

Dana is a wheat farmer. She has three fields which she has harvested and stored in three separate grain bins. Each field has different characteristics and hence the quality of the wheat stored in each bin is different. She wants to maximise her profit by blending wheat from the bins to maximise the amount of High Protein Grade wheat she has to sell.

High Protein grade wheat needs to meet two criteria [1]:

- minimum 14% protein by weight
- maximum 12.5% moisture by weight

[1] Australia currently has 32 grades for wheat determined by 47 criteria (See Wheat Standards 2023/24 All Grades). We're simplifying here.

You task is to determine how Dana should blend the wheat from her three bins to maximise the amount of High Protein grade wheat she can sell.

Important! Solving the problem as stated in full generality goes beyond what is learned the unit (it is what's called a *linear programming* problem) so we'll restrict the problem to cases where it is possible to *exactly* match the requirements, i.e. exactly 14% protein, and exactly 12.5% moisture. This will always be possible for our test cases and there is no need for your solution to work with other cases.

Create a Python function which calculates how much wheat to use from each bin to blend together:

```
def blendWheat(csvfilename):
    # your code here
    return blend, amount
```

Your input data will come from a CSV file with filename given by the `csvfilename` parameter, formatted like so:

```
Bin,Weight,Protein,Moisture
A,12,15,12.5
B,15,13.5,12
C,7,12,14
```


where the `Weight` column gives the amount of wheat in each bin, in tonnes, while the remaining two numbers are percentages.

Your function should return a pair consisting of a Python dictionary containing how many tonnes of wheat from each bin to blend, and the amount of High Protein grade wheat obtained, rounded to the nearest 0.01 tonnes. The solution for the CSV file above is:

```
( { 'A': 12.0, 'B': 10.29, 'C': 3.43 },  25.71)
```

In order to deal with numerical precision problems (because floating point numbers are not exact), all test cases will accept an error of up to 0.01 tonnes.

Hint: You can solve this in two steps by first determining the amount of wheat from each bin required to make exactly 1 tonne of High protein grade wheat with the correct percentages for protein and moisture. This is a system of 3 equations with 3 unknowns. The second step is to scale this so that you run out of wheat in one bin.

For this task the Python code will be marked automatically according to test cases similar to the sample test cases in `test_STA_linalg.py`. You can run the sample test cases with `python test_STA_linalg.py` while ensuring that your solution is called `specialtopics.py` and is in the same directory as `test_STA_linalg.py`. There are 10 test cases in total, including one test case which checks to see if you have used any `for` or `while` loops. As with the Graphs project, your priority should be to have working code as the loops criterion is only worth 1 mark.

2.3 Probability

Recently you have inherited your long lost uncle's wheat farm, and you are determined to make a go of it. It's exciting, but there's a lot to learn. While spending some time around the local coffee shop an old farmer offers you some advice:

"You'll be wanting to get some kind of crop insurance. Input costs are high these days, so don't want to lose it all if you have crop failure. There's a few kinds of crop insurance. The expensive kind pays out for any kind of crop failure. Then there's cheaper kinds that don't pay out as well, and some that don't cover all kinds of crop failure. Plus there's some that only work for certain things. Hail insurance is like that; only pays out if your crop gets hailed out."

You ask the old farmer how to decide what kind to get.

"Depends on your land. This area has a lot of different soil types. Sandy soils are bad if there is a drought. Clay tends to get flooded out if there is lots of rain. Then there's the hills that affect the weather patterns so some places get lots of rain or hail, and others tend to be dry. And for whatever reason, some places get grasshoppers real bad some years and they'll eat your crop right down to the ground. Now seems to me you've got one of old Charlie's fields. He used to keep track of everything that happened in this little book of his, and he gave it to me before he quit farming. I bet you could figure out which fields are prone to what from that."

From Charlie's book you have determined that there are three events that you need to worry about for your field: drought, hail, and grasshoppers (and of course, it is possible that none of these things happen!). Charlie recorded which of these happened for each of his fields for 20 years. Table 1 summarises the information.

Field	Drought	Hail	Grasshoppers	No failure
Home quarter	4	1	1	14
Breaking	3	3	3	11
Lyon quarter	0	4	0	16
Down south	1	1	3	15
Up north	2	2	2	14
The farm	1	1	1	17

Table 1: Number of years in which each event occurred out of 20 years, by field.

Unfortunately, Charlie used names for the fields that apparently made sense to him, but not to anybody else, so you don't know which of Charlie's fields you have.

You have managed to sign a contract with a company that will buy your entire crop at a fixed price, unless your crop fails[2].

Asking around to crop insurance companies, you find that there are five kinds available[3]:

- *Comprehensive.* Pays 80% of your contract price in the event of any kind of crop failure.
- *Hail.* Pays 80% of your contract price in the event of crop failure due to hail.

[2]This is not the way contracts usually work, but we're simplifying for this exercise.
[3]This is also not how crop insurance usually works.

- *Grasshopper*. Pays 80% of your contract price in the event of crop failure due to grasshoppers.
- *Basic*. Pays 50% of your contract price in the event of crop failure not due to hail.

Each year the crop insurance companies post their premiums (the cost of the insurance) and you get a new contract with a new price based on the global price of wheat for the year. Your input costs also vary each year due to fluctuations in the price of seed, fuel, fertilisers, etc..

Your task is to decide which type of crop insurance to buy each year in order to maximise your profits over a span of 20 years[4]. To do so, write a Python function:

```
def chooseCropInsurance(premiums, inputCost, contractPrice, lastYearOutcome, state):
    # your code here
    return insurance, state
```

The parameters are as follows:

- `premiums` is a dictionary with keys being the names of insurance plans and values the cost of each plan, like so:

  ```
  premiums = { 'comprehensive': 5000.0, 'hail':  1900.0,
               'grasshopper':   1600.0, 'basic': 2800.0 }
  ```

- `inputCost` is a number representing the fixed costs for farming for the year.
- `contractPrice` is a number representing the amount of money you receive if there is no crop failure.
- `lastYearOutcome` is a string containing one of the following: `'drought'`, `'hail'`, `'grasshoppers'`, `'no failure'`. On the first invocation of the function (when there is no last year) the value will be `None`.
- `state` is a variable which you can use to remember information from year to year in any format you like. This will be set to `None` for the first year. For subsequent years it will be set to the `state` variable that you returned on the previous year.

Your function should return a pair `(insurance, state)`. `insurance` is a string set to one of: `'comprehensive'`, `'hail'`, `'grasshopper'`, or `'basic'`. This represents which insurance package you want to purchase for the year. `state` is a data structure which you can use to keep track of any information you like. It will be given to you as the `state` parameter the next time the function is called.

Your function will be invoked 20 times (representing 20 crop years), each time with varying premiums, costs and prices. Your net profit will be calculated and added to a running total. The goal is to maximise the *expected* final total. To estimate this, we will run this whole process 5000 times — each time with a randomly selected field over 20 years — and take the average final total.

For this task the Python code will be marked automatically according to test cases similar to the sample test cases in `test_STA_probability.py`. You can run the sample test cases with `python test_STA_probability.py`

[4] Again, this is not really how things work. Insurance always *costs* money when averaged over all policyholders, otherwise insurance companies wouldn't make any money and wouldn't exist. The purpose of insurance is control risk associated with events that could otherwise cause unacceptable outcomes, like bankruptcy.

with your solution is called `specialtopics.py` in the same directory as `test_STA_probability.py`. There are 2 test cases in total, including one test case which checks to see if you have used any `for` or `while` loops, and is worth 1 mark. As with the Graphs project, your priority should be to have working code as the loops criterion is only worth 1 mark.

The second test case is special and is worth 9 marks. Unlike the other topics which have multiple separate test cases, the functionality of your code for this task is assigned based on your average net profit, compared to a constant strategy which always chooses the basic insurance policy. Let P be your average profit while $B = 61000$ is the average net profit for the basic strategy. This test case will assign points according the the following formula:

$$\left\lceil \frac{P - B}{1830} \right\rceil$$

While this test is considered to be out of 9, it is possible to achieve a higher grade. Any marks above 9 will be considered bonus marks.

For this task, the only difference between the given test cases and the assessed test cases will be that a different seed for the pseudo-random number generator will be chosen to ensure that your solution does not take advantage of the specific sequence of pseudo-random choices in the test. You can change the random seed yourself to see how your solution behaves with different pseudo-random choices. To do so, adjust line 10 of `test_STA_probability.py`.

3 Report

As with the graphs project, your report should use mathematical language, concepts and notation from the unit to describe your chosen task and your solution.

- 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. Describe the information given in mathematical terms and using mathematical notation. It is expected that your will make use of tools from unit special topics week corresponding to your chosen task.
- 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. Describe how the solution to the mathematical problem relates to the solution to the original problem.
- Describe your implementation in Python. Mention the role of important variables, library calls (eg. to `probability.py` or `numpy`), 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.

There is no need to use separate sections. A format similar to the Graphs project sample report is fine, but note that the problems here are more complex and hence your report will likely be as well.

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 for the Graphs project — available on Canvas — demonstrates reasonable citations and bibliography.

Overall, your 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. One or two pages is about right, not including diagrams if you have any.

4 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. Some of these functions and more are collected in the Python file `probability.py`. 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 `re`, `csv` and `numpy` Python modules. Before using other modules, please contact the unit coordinator.

A submission template file is available from the Canvas Assessments module. If your solution includes significant amounts of modified code from the unit (more than a couple of lines from one place), say so in a comment explaining where you obtained it and what modifications you made. One line is enough detail.

Each task has an associated Python file for testing your solution. See the task description for more details.

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.

5 Marking criteria

Your mark is made of two parts. The report is graded out of 20 and the Python code is graded out of 10, for a total of 30. Each mark counts 1% towards your final grade.

5.1 Report

The marking rubric is available on Canvas under Assignments > Special Topics Assignment Report.

5.2 Python code

Your Python code will be graded automatically according to tests as described in the task descriptions above. The tests will be similar, but not identical, to those found in the sample test files.

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.

6 Submission

Please remember, you only need to solve *one* task. If you submit solutions for more than one task then we will grade the first one only.

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 (filename `specialtopics.py`).

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

Extensions: Information about extensions is available in 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 (eg. 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.