ENGF0002 Team 24 Scenario 1 Report

(**Written by:** Eva Miah, Guodong Liu, Cheng Loo. **Unreachable member**: Safwan Shah.)

1. Introduction

The study of logic and maths can be quite daunting at times. But for computer scientists, they are inextricably linked to the fundamentals of our studies, and are essential for us to be proficient and effective learners. This report outlines the tool that we have brainstormed which we believe will aid all those who want to practise their concepts learnt during lectures.

We have decided to implement a truth table filler that allows the user to input a true or false value (0 or 1) in the respective cells. The program will then evaluate the answers made by the user and highlight anything that is incorrect in the truth table, and also provide details as to why it is incorrect.

This tool will allow the user to grasp logic concepts more quickly when used extensively, and hence be more efficient at solving similar problems.

2. Components of Tool

2.1 Overview of Tool

This tool will primarily use Python as our programming language, as it is more suitable to support classes and has UI capabilities.

There will be a UI to allow users to interact with the tool more seamlessly and give the user a more immersive learning experience.

The program will generate logic formulas and the correct truth values to fill in the truth table. With this data, the user will be shown an empty truth table on the UI with just the headings of columns, and they will be able to input the truth values (0 or 1) in the empty cells. When the table is completely filled, they will be able to submit their answer and they will be given immediate feedback. The background colour of cells will turn red if a truth value entered is incorrect.

2.2 Data Structures

Truth tables are formed primarily by rows corresponding to each combination of truth values and columns for each proposition and/or term included in the formula. To represent the data in this table, we can implement a 2D array where the first index points to a row and the second index

points to the column in that row. In such a way, each pair of indexes will store the value in a single cell in the truth table.

2.2.1 Calculating array size

In order to display a table on the UI, the program needs to know the size of the table, or in other words, the number of rows and columns needed for the formula's truth table.

Table rows

The number of rows in the table is calculated by $2^n + 1$ where *n* is the number of propositions in the formula and the extra row is used to store the column headings. For example, a formula containing only 2 propositions, such as A ∨ B, will need $2^2 + 1 = 5$ rows, the first for the heading and the remaining 4 to store the combination of different truth values for that specific proposition. In the array implementation, this calculation will return the size of the 2D array, that is to say, the number of *arrays* it stores. Here is a visualisation of it:

A	**B**	**A ∨ B**
0	0	0
0	1	1
1	0	1
1	1	1

Fig 2.2.1.1

2D Array representation = [['A', 'B', "A ∨ B"], [0, 0, 0], [0,1,1], [1,0,1], [1,1,1]]

Table Columns

The number of columns will depend on the complexity of the formula and to which level we are breaking down the steps in filling the truth table. Initially, the number of columns is calculated with n + 1, where *n* is the number of propositions and the additional column stores the truth values for the overall formula. For instance, the previous formula A ∨ B would only require three columns. However, if the formula is more complex and contains multiple statements, i.e. (¬ A ∨ B) ∧ C, in order to facilitate the learning process it is ideal to have another additional column for the negation term (¬A) as well as another column for the first statement in the conjunction (¬ A ∨ B), and then have the final result column.

To recognise the cases in which these extra columns should be added, we will identify the formulas that have multiple statements by the number of brackets the formula contains. In logic syntax, this means that the truth value of the statement in brackets can be calculated first, and then used with the rest of the formula. Therefore, the total number of columns is determined by

n + 1 + *neg* + *s* where *n* is the number of propositions, *neg* is the number of negations and *s* is the number of statements in brackets. This calculation will return the size of each array stored in the 2D array, or in other words, the number of *items* in each array. Here is a visualisation for the example mentioned:

A	**B**	**C**	**¬ A**	**¬ A ∨ B**	**(¬ A ∨ B) ∧ C**
0	0	0	1	1	0
0	0	1	1	1	1
0	1	0	1	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	0	1	1

Fig 2.2.1.2

2D Array representation = [[A','B','C',"¬A","¬ A ∨ B","(¬ A ∨ B) ∧ C"],
[0,0,0,1,1,0], [0,0,1,1,1,1], [0,1,0,1,1,0],
[0,1,1,1,1,1], [1,0,0,0,0,0], [1,0,1,0,0,0],
[1,1,0,0,1,0], [1,1,1,0,1,1]]

2.3 Algorithms

The program will require an algorithm that includes the different parts of the tool. First, we need to generate logic questions and their corresponding truth table values. Then, the user will fill in an empty table and the program will store the answers received after checking that the responses are valid. The user will then get feedback on which inputs are incorrect and why before moving on to the next question.

2.3.1 Overview

The sequence of the program is represented by a flowchart as shown below:

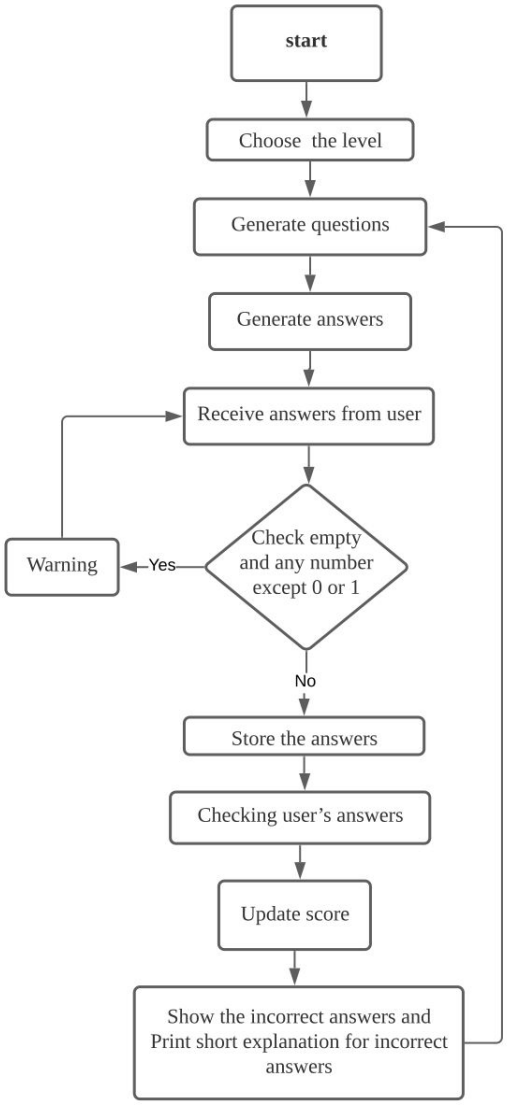

Fig 2.3.1.1

2.3.2 Basic user functions

In order to achieve that, our program should contain following functions :

1. Generate questions and store values in a 2D array.
2. Display an empty truth table on the UI for the generated formula to the user.
3. Receive the answers from the user and check whether the input is empty.
4. Give a warning when receiving an invalid input and prompt another input, repeat until input is satisfied.
5. Store the answers in a 2D array.
   [The answer will be stored in a pre-created list sequentially.]
6. Check the answers sheet with the generated truth table values.
   [compare them sequentially, and store indexes of wrong answers.]
7. Highlight wrong answers in red on the UI and give feedback to the user.

2.3.3 Generating Questions

The difficulty of questions depends on the level that the user has chosen. It is defined as such:

a. Easy - 2 propositions
b. Intermediate - 3 propositions
c. Advanced - 3 propositions + use of implication(→) & biimplication(↔)

A question which has *n* propositions will produce a table that has (2^n + *1*) rows and (*n* + 1 + *neg* + *s*) columns where *n* is the number of propositions, *neg* is the number of negations and *s* is the number of statements in brackets (breakdown is shown in section 2.2 under Data Structures). For a table with 2 propositions, an example is shown in Fig 2.2.1.1.

Every proposition equation after the *nth* column on the 1st row will be randomly generated in the following manner:

$$k_1 \; op \; k_2 \; op \; k_3 \; op \; \ldots \; k_n$$

where $k_i \in \{k_1, k_2, k_3, \ldots k_n\}$ represents a proposition while *op* ∈ the logic operators (→, ↔, ⋀, ⋁).

Special operators:

1. **()** (parentheses) This can be used to encompass one or more sets of 2 propositions and 1 operator, used as such: (k_1 *op* k_2). Used to denote precedence, equations within parentheses must be resolved first.
2. ¬ (negation) This can be attached to propositions themselves, such as $\neg k_1$, which is then assigned as a new proposition. This can also be attached to groups of operations, such as $\neg(k_1 \; op \; k_2)$.

2.3.4 Generating Answers

For a table that has *n* rows, the program will start with the 1st row of values (i.e 2nd row from the top) and read the values of the n number of literals and store them in a 2D array (shown in 2.2). Using these values, the program then evaluates the equations generated in 2.3.2 by comparing the values using the appropriate logic operators. The output of either 0 or 1 is then stored in the respective column in that row.

This process is then repeated until the *nth* row is done. When all the elements in the 2D array have the same length, the array of answers is then considered generated.

2.3.5 Checking user answers

After the user inputs values in each cell of the truth table from the UI, the program will loop through each row and generate a 2D array for the user's response. To assess the correctness of their answer, the generated correct answer will be compared with the 2D array containing the user's response.

If a value is found that does not match the correct answer, the program will store the index of that value. This will allow the program to identify the location of the incorrect answer (cell) in the truth table on the UI and highlight it in order to output the appropriate feedback to the user.

If the user manages to get all the values correct, the program will display it as such and the user can then proceed to the next question.

2.3.6 Score System

There will be a simple score system used to keep track of the user's progress with the questions and also to gauge how well the user is doing.

For every row that the user inputs matches the answer, the user's score will increase by a certain value. The entire row must match for the user to get the points to show that the user can grasp the concepts firmly.

3. Platform

3.1 Using a UI

Since it is entirely possible to implement this tool on a website, we think that using a UI is the best way to bring our tool across to the user. A UI provides a more interactive user experience and gives the user a greater source of motivation to complete the practices.

First, we have to start with a wireframe of the tool that we are implementing. Wireframes provide a clear overview of the functionality, design, and layout of the tool. As such, we are starting off with a low-fidelity wireframe, and will slowly improve to mid-fidelity wireframe as we progress in the development of the tool.

3.2 Examples of Practices

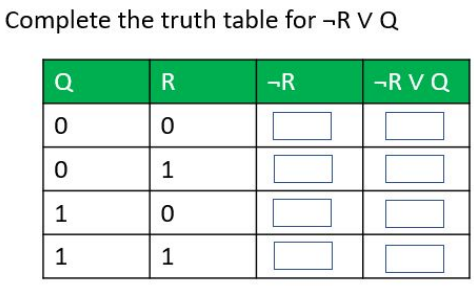

Fig 3.2.1

In Fig 3.2.1, it shows an example of how the truth table questions will be presented to the user. They can then input values (‘1’ or ‘0’) in the empty boxes.

Complete the truth table for ¬R V Q

Q	R	¬R	¬R V Q
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	1

Fig 3.2.2

In Fig 3.2.2, it shows an instance where a user inputs incorrect values (‘0’ when it is supposed to be ‘1’). The cell of the incorrect answer is highlighted in red.

Notes:
The expression in the form of ¬X is a negation, meaning **not** X. It is true if X is false and false if X is true.

The expression in the form of X V Y is a disjunction, it means X **or** Y. It is true if either X or Y are true, and false otherwise.

Solution for ¬R V Q

Q	R	¬R	¬R V Q
0	0	**1**	**1**
0	1	**0**	**0**
1	0	**1**	**1**
1	1	**0**	**1**

Fig 3.2.3

In Fig 3.2.3, since the user gave the incorrect values, the program gives a short explanation of why the input answers are incorrect. The solution is then shown at the bottom with all the correct values in blue.

These are just some basic examples of how the program is supposed to function, and there will be more complex truth tables with more advanced logic symbols involved at the higher levels.

4. Other Considerations

4.1 Identify weak areas

After completing a certain number of practices, the program is able to identify which areas that the user is weak in (e.g. frequently gets implication columns filled incorrectly) and gives more questions corresponding to that specific type. To do this, the program can keep track of the type of question answered incorrectly in order to determine the area of weakness.

4.2 Saving questions

As the program only generates questions when it is run, the user may want to save some of the formulas that they were given and attempt to answer them later on. To do this, the generated formulas and corresponding truth tables should be stored externally so that they could be loaded later on. This feature would enable the user to re-attempt questions that they had answered incorrectly and/or did not understand, thus allowing them to work on their learning through their past mistakes. If we were to add this feature within the time constraints, we would need to save the generated formulas and truth tables into a CSV file.

The CSV file would contain the formula and the data that is stored in every cell of the truth table, including headings. Each row in the CSV file will represent the rows in the truth table, and it will be possible to identify when the truth table data changes when a row only contains headings.

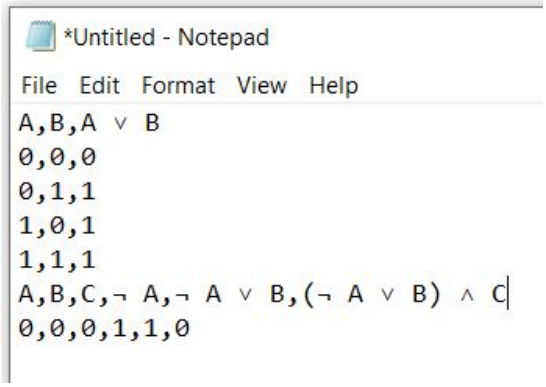

Fig 4.2.1

As illustrated in Fig 2.2.3, the CSV file will contain the data that is required to fill in a correct truth table. To load the saved data, the program will read this line by line and add each value separated by a comma in an array, and then append the array of each row to another array representing the whole table, hence forming the 2D array structure.