**Scenario 2 Report**

***Overview***

Our group’s planned proposal from Scenario 1, was to implement a propositional logic calculator to help people learn the topic. It would take a logical expression, which would become a question proposed to the user for them to simplify. The tool would calculate an answer (which was the simplified expression), whilst the user attempts to answer the question. Once the question had been answered, the working out and truth table could be displayed to the user. My approach to brainstorming was thinking about the utility of the tool. I was meticulous regarding the User Interface (UI) being easy to navigate and understand; in the sense that it is clear to a user what they should do and can do.

The tool implementation, in Scenario 2, was extremely well done. Our tool implemented all the core components planned from Scenario 1. It could take logical expressions with conjunctions, disjunctions, negations, implications and bi-implications and simplify it. The tool could simplify into AND/NOT form which means the final answer would only contain conjunctions and disjunctions. Or DNF form which would take an expression and return the Disjunctive Normal Form (DNF). Also it could display the truth table. I was assigned the task to code AND/NOT form. Also, I helped my team member responsible for the front-end to integrate my code to the front-end, and I wrote most of the backend code for the Truth Table.

***My Planning and Brainstorming***

My first task for the implementation was to understand the most effective and best way to simplify expressions into AND/NOT form. I first tackled a few example questions to try and build an algorithm. I deduced the best solution was to continuously apply logical equivalences to simplify the expressions. The reason being is that I simplified many logical expressions online with this method. Secondly, to remove all the implications, negations and Bi-Implications; using logical equivalences was the quickest and easiest method. For example, p ∨ q ≡ ¬p → q. By applying different logical equivalences, I was able to fully simplify nearly every logical expression. I then made a library of all the logical equivalences I had used during my experimentation. This was 13 equivalences. All these were included in the code, in the form of a remove Bi-Implications method and 12 logical equivalences. Using these equivalences, in any order, means that nearly any logical expression put into the tool can be simplified. This tool is almost ready to use as a fully working learning platform.

***My Code and Implementation***

The code starts by retrieving a list of all the propositional arguments/ letters used in the expression. This list does not include duplicates, therefore every letter at every list index is unique. After this, all the spaces are stripped from the logical expression allowing the 12 logical equivalences to be applied properly. All the equivalences use pattern matching, meaning that any number of propositional letters/ arguments could be included in the logical expression. The code can still recognise this and deal with it, through using the list which holds every propositional letter, or by using iteration of the function. For example, if ‘(p V p V p V p)’’ was passed into the function, it would recognise the first ‘p V p’ and replace it with ‘p’. Then ‘(p V p V p)’ would be passed into the function again and this process will repeat until we have the simplest form: ‘(p V p)’. This is an example of iteration, as the function will stop iterating once the expression passed into it is the same length before and after. The same applies for ‘p V p V p V p’, the answer would be ‘p V p’, the tool is bracket specific. For pattern matching, if ‘¬p → (q ∧ r)’ was passed into the function, it would be treated the same as ‘¬p → (q ∧ r ∧ s ∧ t ∧ p ∧ u)’. This is because it would pick up the negation, assert the implication and opening bracket ‘)’ is in the correct position. Then it will use the arguments list I mentioned earlier and assert that the number of conjunctions is one less than the size of the list. Then assert the closing bracket ‘)’ is in the correct position. In this case it would be: (size of the list of arguments \* 2) – 1. Then it’ll apply the equivalence by splitting the string at a point, replacing and deleting certain characters. Similar logic is applied in the code for every logical equivalence. Every time a logical equivalence is applied, the new logical expression is appended to a list. At the end, this list represents the working out. This is stored in a list for the purpose of being transferred to the front-end.

***Other Parts Of The Tool***

The Truth-Table component of the tool was implemented using an imported module. The code only needed to obtain a list of the propositional arguments and the string with the simplified logical expression. These were passed as parameters to a built-in function which generated the truth table. So, all that was required was my function which collects the list of all the propositional arguments/letters, and then reformatting the logical expression so that it can be simplified and passed into the function which generates the truth table.

The DNF component of the tool has a sophisticated implementation. The logical expression is reformatted to include symbols that the program would understand e.g., ‘|’ instead of ‘∨’.

The program then applies logical equivalences that follow the DNF form, for example a, remove Bi-Implications for implications method. This is repeated until the expression is simplified into DNF form, to give the final answer.

***Limitations, Design Choices, Shortcomings and Setbacks***

Despite my implementation of AND/NOT form working for nearly all logical expressions, there is 1 bug. The code sometimes has an inability to handle duplicates in long expressions. Here is an example: p ∧ ((p ∨ p) ∨ r ∨ s) and p ∧ (p ∨ p ∨ r ∨ s) can be simplified. However, p ∧ (p ∨ r ∨ p ∨ s) won’t work. This is because it won’t be able to see that there is a ‘p disjunction p’ as the p’s are not consecutive. The program only asserts that there is 1 more unique propositional letter than disjunctions, and that none of the disjunctions are next to each other. Similar logic is applied to most of the other logical equivalences. In hindsight this has a very easy fix. I could split the string at the opening bracket. And then use my function to obtain the number of propositional arguments in the split string. Then if the number of conjunctions/disjunctions is not 1 less than the number of arguments, then that means the argument list is too small, meaning there are duplicate arguments. After this an iterative method can be applied to identify the duplicate letter(s) and eliminate them, then the rest of the expression can be simplified. This logic should hold for nearly every logical equivalence in my code and hence every logical expression. Also, I wanted my code to be case specific, I chose this design implementation as it would be consistent throughout everyone’s code. However, this perhaps wasn’t optimal, but can be fixed by adding 1 line of code:

‘expression = expression.lower()’, always converting the expression to lowercase.

One shortcoming we had was fully integrating the AND/NOT form into the front-end, hence I helped. We integrated the part of the tool that reads questions from a text file. However, the part of the tool where the user inputs their own logical expressions to be simplified; the integration was not done in-time. Originally both parts did not work. The reading questions from a text file part of the tool, did not generate the correct answer for AND/NOT form (on the frontend), which set us back by 2 days. The issue was that the argument list kept getting appended to and never got emptied between questions. So, we set it to an empty list at the start of the function; so when the function is used for the first time for a question, it’s empty.

***Conclusion***

Over the course of Scenario 2, my python skills have developed greatly. I have learnt an entirely new style of coding; special class structures and files to integrate frontend and backend code. This indirectly led to me learning Tkinter frontend development. This has given me confidence to do my own projects. Also, my team-work skills have greatly improved. I have confidence in trusting my teammates to help me and equally helping them is beneficial for everyone. We worked effectively as a team; building and supporting each other’s ideas. Thus, the work was enjoyable, and our workload was well managed.