Inference Engine: Assignment 2

Introduction to Artificial Intelligence

Aston Lynch, 103964552

Hai Nam Ngo, 103488515

COS30019_A02_T064

Contents

1 Introduction

This report will cover our response to the provided instructions for Assignment 2.

1.1 Background

Within this assignment, we have been tasked with implementing an inference engine for propositional logic in python. An inference engine works as a component of an expert system that applies logical rules to the knowledge bases to deduce information, or make decisions.

Files are formatted to consist of a KB and a q, where, the KB follows the keyword TELL and consists of Horn clauses separated by semicolons, and the q follows the keyword ASK and consists of a proposition symbol which is evaluated for entailment in the provided KB. Figure 7 provides the expected formatting of a Horn-based input file.

```
TELL
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p2&p1&p3=>d; p1&p3 => c; a; b; p2;
ASK
d
```

Figure 1: Example of an expected Horn statement.

This program has Truth Table (TT) checking, and Backward Chaining (BC) and Forward Chaining (FC) algorithms. The program works to take arguments a Horn-form Knowledge Base (KB) and query (q) which is a proposition symbol and determine whether q can be entailed from KB.

2 Inference Methods

Within the creation of our inference engine, we have implemented multiple inference methods, each of which employs a different technique to the others in how to logically determine whether (q) is entailed by any given KB.

2.1 Truth Table Checking

Truth Table Checking is a brute-force approach to solving given logical problems [1]. It works by listing all possible truth assignments for the propositional symbols present in the KB and (q). Truth

Table checking in the circumstances of this project work to be the most complete and sound method to logically deduce propositions. This method works by extracting all different propositional symbol from KB. Once this is done, a truth table is generated, containing every possible combination of truth values (True or False) for these symbols. Each row represents a different model, which is a specific combination of truth values assigned to the symbols. For every model, the KB and the (q) are evaluated to determine their truth values under any given assignment. This checks if the clause and the query is true. If a model is true in both cases, it is considered a valid logical proposition. Truth tables are the best way to guarantee if a query is entailed by a KB, the method will find a model to prove this. A drawback to using this method is that it becomes impractical and difficult for large-scale problems due to the exponential scaling of rows for each model[2].

2.2 Forward Chaining

Forward Chaining is yet another method of solving given logical problems, particularly in scenarios with Horn clauses where there is only a single positive literal. The technique involves a systemic analysis of facts and rules within a KB. Forward chaining begins by identifying facts within a KB. Each rule is then scrutinised to ascertain whether it's premises are satisfied by pre-established facts [3]. If rule conditions are met, conclusion is drawn from that rule, and added as a new fact to the KB. This process is sequential, and continues until no further inferences can be made or until a query (q) is true. Forward chaining really excels in reasoning in linear time complex scenarios, which further ensures it's completeness within Horn clause cases [1]. While Forward Chaining is efficient for horn-clause environments, it struggles in situations where generalised logic or interconnected rule sets are present.

2.3 Backward Chaining

In contrast to Forward Chaining, Backwards Chaining focuses on the query (q) from the outset. This method begins with a query (q) and works itself backwards to determine if there is a set of facts in the KB that match (q). Backwards chaining also works to understand if the implications in the KB could conclude the query. If these premises are not established facts, they become sub-goals, in which case, the Backwards Chain works to recursively find their truth [1]. This recursion works until it reaches a fact that is known to be true, or fails to find a logical truth within the KB. This is a better approach in situations where the ruling for a KB and query (q) is verbose, but the solution path may not require evaluating each implication. This makes it work well in use-cases related to Horn clauses, where it is likely to find a solution if one does exist. It suffers the same problem as Forward Chaining, where it's application is likely ineffective (to other algorithms) in situations in which general logic is being evaluated.

Each of these methods have their own strengths and weaknesses, making them ideal as solutions for differing problems. Within the assignment we have utilised each of these inference methods to demonstrate their usage, as well as, their performance against each other in the same environment.

3 Implementation

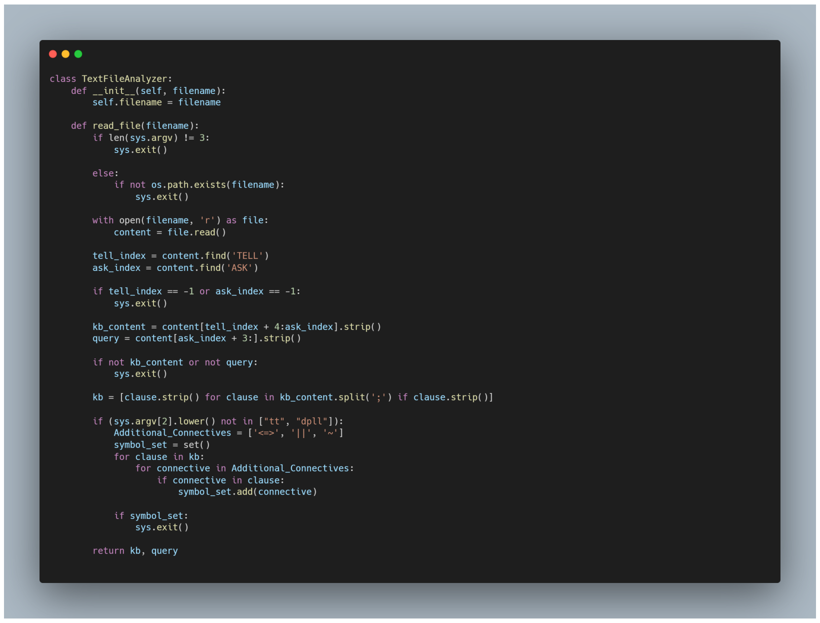
In this section, we provide an overview of the implementation of the propositional logic inference engine. Our solution, written in Python, supports four inference method: Truth Table Checking (TT), Forward Chaining (FC), Backward chaining (BC), and DPLL (Davis-Putnam-Logemann-Loveland). The implementation provided by us in response to the assignment brief is located within one central file 'iengine.py' which has different classes which encapsulate the logic for each inference method. We will now introduce these classes, as well as, what they do in respect to our entire solution.

3.1 Text File Analyser

The 'TextFileAnalyser' class is responsible for reading and parsing our structured input file. This class extracts the Knowledge Base (KB) and the query (q) from a text file. As shown previously shown in Figure 7, these text files are formatted to contain the KB after the 'TELL' keyword, and (q) after the 'ASK' keyword. The class first works by ensuring that the input file is correctly formatted, where, the code is able to use '.find' and some indexes to conditionally ensure that the keywords 'TELL' and 'ASK' can be found, and that the lines after each keyword contain the KB and (q). If a valid, formatted file is input, the KB content is formatted and split into individual clauses based on the semicolon separator. An extra bit of code checks to see if any extra symbols are present within the KB or (q), which is used for separating the handling of Horn clauses to our generic logic formatted files. If generic logic is found, it is processed in a separate way due to the extra connectives, but ensures that only the TT or DPLL methods are ran on this data - FC, BC independent for Horn clauses. The class returns two variables - kb, and query, which we manipulate further.

3.2 Inference Methods

This section of the implementation documentation will provide insight into the integration of the previously mentioned inference methods within the solution. The classes that are associated with inference method operations consist of:



```python
class TextFileAnalyzer:
    def __init__(self, filename):
        self.filename = filename

    def read_file(filename):
        if len(sys.argv) != 3:
            sys.exit()

        else:
            if not os.path.exists(filename):
                sys.exit()

        with open(filename, 'r') as file:
            content = file.read()

        tell_index = content.find('TELL')
        ask_index = content.find('ASK')

        if tell_index == -1 or ask_index == -1:
            sys.exit()

        kb_content = content[tell_index + 4:ask_index].strip()
        query = content[ask_index + 3:].strip()

        if not kb_content or not query:
            sys.exit()

        kb = [clause.strip() for clause in kb_content.split(';') if clause.strip()]

        if (sys.argv[2].lower() not in ["tt", "dpll"]):
            Additional_Connectives = ['<=>', '||', '~']
            symbol_set = set()
            for clause in kb:
                for connective in Additional_Connectives:
                    if connective in clause:
                        symbol_set.add(connective)

            if symbol_set:
                sys.exit()

        return kb, query
```

Figure 2: Code of TextFileAnalyser class

- TT - Truth Table Checker
- Chaining - A base class for shared logic between chain operations
- FC - Forwards Chaining
- BC - Backwards Chaining

3.2.1 Truth Table Checker

The 'TT' class, is responsible for the handling of the Truth Table Checking inference methodology. The design of this code allows me to verify whether a given query, (q) is logically entailed within a KB using the truth table approach. This class has multiple functions that are useful in extracting logic from the Horn clauses, and later, generic logic too. Some notable functions within this class would be:

1. CreateTruthTable

The 'CreateTruthTable' function within the 'TT' class is where the truth table is constructed and evaluated. It works systemically to evaluate the logical entailment of (q) based on KB. This function has multiple important operations including:

- Symbol and Model Extraction

The function invokes the 'ExtractSymbols' method to retrieve all unique propositional symbols from KB. After that, we create the models variable which utilises product from itertools. In this case, product will generate all possible combinations of truth values (True, False) for the extracted symbols, which give us our list of models. These are important for the rest of our code to work as each 'model' represents a unique assignment of truth values across all symbols.

```
symbols = self.ExtractSymbols(self.kb)
models = list(product([True, False], repeat=len(symbols)))
```

Figure 3: Code Snippet of symbol extraction/model generation

- Evaluation of Knowledge Base and Query for Model

For every model created, the method constructs a row for the truth table. The row starts with the truth values of each symbol in the current model. The function evaluates the truth of every clause in the KB against the current model by leveraging the "Check_if_clause_true" method, where the truth of the query is evaluated against the current model. The TT class features a few other

```
for model_values in models:
    model = dict(zip(symbols, model_values))
    kb_values = [self.Check_if_clause_true(clause, model) for clause in self.kb]
    query_value = self.Check_if_clause_true(self.query, model)
```

Figure 4: Evaluation of Knowledge Base and Query for each model

methods that are important for the overall operation of the solution, and locally within the TT class. The function of these is simple to understand and not worth writing about in grave detail. I will also cover two important methods within the TT class "EvaluateClause" and "FindMainOperator" as they are more relevant to the research initiative taken on.

3.2.2 Chaining

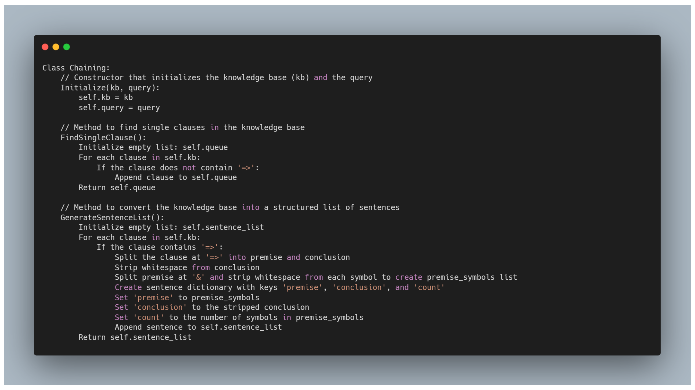
The Chaining class within our solution serves as a base class for the Forward Chaining and Backward Chaining inference methods. It is designed as a helper for these methods by preparing the parsed KB and query for the inference process. This base class allows for us to have a layer of abstraction within our components.

- Finding Single Clauses

This method iterates through each clause in the KB. It specifically looks for clauses that don't contain an implication =>, evaluating them as truths within the KB. These clauses are added to a queue, which we use for the FC later.

- Generating Sentence Lists

This method processes the clauses in the KB that contain =>. Each clause is interpreted as logical ruling, and split into premises and conclusion (ref AIMA). The premises are later split by the conjunction operator (&) and stripped. Once processed, these are stored in a dictionary 'sentence_list'. We use this for both the FC and BC within the solution.



```
Class Chaining:
    // Constructor that initializes the knowledge base (kb) and the query
    Initialize(kb, query):
        self.kb = kb
        self.query = query

    // Method to find single clauses in the knowledge base
    FindSingleClause():
        Initialize empty list: self.queue
        For each clause in self.kb:
            If the clause does not contain '=>':
                Append clause to self.queue
        Return self.queue

    // Method to convert the knowledge base into a structured list of sentences
    GenerateSentenceList():
        Initialize empty list: self.sentence_list
        For each clause in self.kb:
            If the clause contains '=>':
                Split the clause at '=>' into premise and conclusion
                Strip whitespace from conclusion
                Split premise at '&' and strip whitespace from each symbol to create premise_symbols list
                Create sentence dictionary with keys 'premise', 'conclusion', and 'count'
                Set 'premise' to premise_symbols
                Set 'conclusion' to the stripped conclusion
                Set 'count' to the number of symbols in premise_symbols
                Append sentence to self.sentence_list
        Return self.sentence_list
```

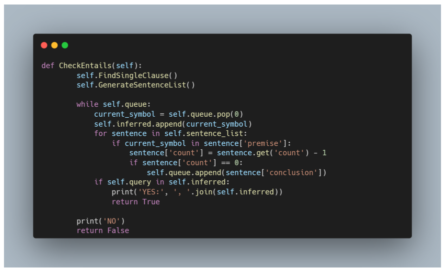
Figure 5: Pseudocode of Chaining class

3.2.3 Forward Chaining

The Forward Chaining (FC) class within our solution builds off of the functions spoken about in the Chaining class. It operates in a data-driven logical processing manner for handling our Horn clause inputs. As mentioned previously in the inference methods section, forward chaining works to derive new information from leveraging known information. Within the class, "CheckEntails" is the compute method that builds builds off of the chaining class.

- Entailment Checking

This method is the core of our logic. It populates a queue with facts from the KB and organises the implications into a 'sentence_list'. After this, it will process each symbol in the 'queue' and update the state of the knowledge base accordingly. For each symbol, the FC will check the premises of implication. In this case, a premise refers to the conditions outlined for rules to apply. When a symbol forms part of the condition the algorithm reduces the number of remaining conditions that need to be proven for the premise to be true. Once all conditions of a premise are confirmed (index to 0), conclusion is drawn as true, and added to the queue.



```
def CheckEntails(self):
        self.FindSingleClause()
        self.GenerateSentenceList()

        while self.queue:
            current_symbol = self.queue.pop(0)
            self.inferred.append(current_symbol)
            for sentence in self.sentence_list:
                if current_symbol in sentence['premise']:
                    sentence['count'] = sentence.get('count') - 1
                    if sentence['count'] == 0:
                        self.queue.append(sentence['conclusion'])
            if self.query in self.inferred:
                print('YES:', ', '.join(self.inferred))
                return True

        print('NO')
        return False
```

Figure 6: Code of entailment checking in FC

3.2.4 Backwards Chaining

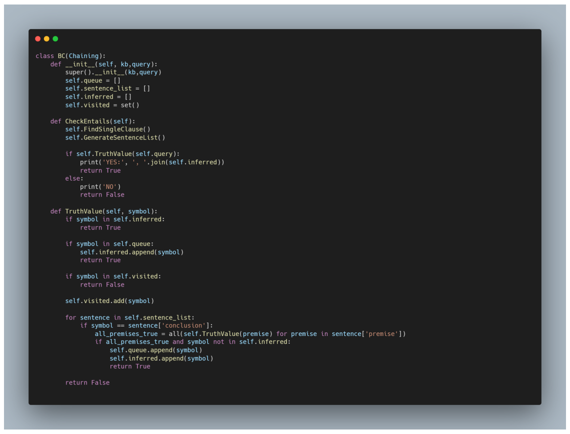
Similar to the FC class, our Backwards Chaining (BC) class builds off of the methods derived from the Chaining base class. As outlined in the inference methods section, Backwards Chaining adopts a goal-driven approach to logical reasoning. Backward chaining begins with the query and works backwards to ascertain if the conditions will support the given query, this is contrary to the data-driven approach taken by FC, where, known information is used to deduce new information.

- Entailment Checking

The core function in this class is "CheckEntails", which leverages the structured knowledge base preprocessed by the Chaining class. It initiates the process by placing facts into the 'queue' and organising implications into a 'sentence_list', similar to the Forward Chaining approach. The BC algorithm will then derive the truth value of the query by tracing it back through the parsed facts within the queue. For each symbol that is analysed, if not already confirmed as true or processed, BC explores premise of implications to ensure all conditions leading to the conclusion are satisfiable.

4 Testing

This section of the report will provide an overview of the test cases that have been made to test our program. This will be insightful, and each test-case will have a description of what it is testing and how each test case is uniquely checking for something different.



```python
class BC(Chaining):
    def __init__(self, kb,query):
        super().__init__(kb,query)
        self.queue = []
        self.sentence_list = []
        self.inferred = []
        self.visited = set()

    def CheckEntails(self):
        self.FindSingleClause()
        self.GenerateSentenceList()

        if self.TruthValue(self.query):
            print('YES:', ', '.join(self.inferred))
            return True
        else:
            print('NO')
            return False

    def TruthValue(self, symbol):
        if symbol in self.inferred:
            return True

        if symbol in self.queue:
            self.inferred.append(symbol)
            return True

        if symbol in self.visited:
            return False

        self.visited.add(symbol)

        for sentence in self.sentence_list:
            if symbol == sentence['conclusion']:
                all_premises_true = all(self.TruthValue(premise) for premise in sentence['premise'])
                if all_premises_true and symbol not in self.inferred:
                    self.queue.append(symbol)
                    self.inferred.append(symbol)
                    return True

        return False
```

Figure 7: Code of entailment checking an truth assignment in BC

4.0.1 Horn Testing

The following is a table that represents the test cases that were conducted to evaluate that horn clauses were handled correctly within the program. This testing was extensive and for different reasons. All test cases were logically proofed by online tools and my own interpretation of propositional logic.

Name	**Expected**	**TT**	**FC**	**BC**
hornKB	YES	YES, 3	YES: a, b, p2, p3, p1, d	YES: p2, p3, p1, d
hornKB_1	YES, NO	YES, 3	NO	NO
hornKB_2	YES	YES, 1	YES: A, B, L, M, P, Q	YES: A, B, L, M, P, Q
hornKB_3	YES	YES, 4	YES: K	YES: K
hornKB_4	YES, NO	YES, 8	NO	NO
hornKB_5	YES, NO	YES, 9	NO	NO
hornKB_6	YES, NO	YES, 6	NO	NO
hornKB_7	YES	YES, 1	YES: M, N, P, O, Q, R	YES: M, N, O, P, Q, R
hornKB_8	YES	YES, 1	YES: A	YES: A
hornKB_9	YES, NO	YES, 1	NO	NO
hornKB_10	YES	YES, 9	YES: A	YES: A
hornKB_11	YES	YES: Too many values	YES: Too many values	YES: Too many values

Table 1: Results of Horn Test Cases

- **hornKB**: Testing inference with multiple premises to lead to conclusion.

- **hornKB_1**: Simple inference test with missing facts to support the query.
- **hornKB_2 to hornKB_3**: Basic chaining proving a direct and indirect conclusion.
- **hornKB_4 to hornKB_6**: Tests with insufficient data to prove the conclusion.
- **hornKB_7 to hornKB_10**: Testing different things for BC and FC. This includes: Long chain with facts, direct verification handling, infinite loop handling, and ensure that chain doesn't populate irreverent rules.
- **hornKB_11**: Stress test for inference methods.

4.0.2 General Logic Testing

The following table represents the test cases conducted to evaluate the handling of general propositional logic within the program, especially to verify the implementation of DPLL alongside the TT method.

Name	**Expected**	**TT**	**DPLL**
GenericKB	NO	NO	NO
GenericKB_1	YES	YES, 3	YES
GenericKB_2	YES	YES, 3	YES
GenericKB_3	NO	NO	YES
GenericKB_4	NO	NO	YES
GenericKB_5	NO	NO	YES
GenericKB_6	NO	NO	YES
GenericKB_7	YES	YES, 2	YES
GenericKB_8	YES	YES, 264	YES
GenericKB_9	NO	NO	NO

Table 2: Results of General Logic Test Cases

- **GenericKB**: Validate unsatisfiability with mixed conditions.
- **GenericKB_1 to GenericKB_2**: Simple generic logic to test correct handling of satisfiability and model generation.
- **GenericKB_3 to GenericKB_6**: Complex propositions to test the robustness and correctness of the DPLL implementation in contrast with expectations.
- **GenericKB_7 to GenericKB_8**: Validation of satisfiability with large number of models and complex logical relationships.
- **GenericKB_9**: Testing the edge cases of contradictory conditions to confirm the non-satisfiability.

5 Features/Bugs

This section of the report will quickly go over the satisfaction of required components within the assignment description and outline, as well as, introduce any shortrcomings or problems that we have within our overall solution.

- **Features**

As for the features we have implemented within the overall solution, I would mention that we have implemented all required inference methods (tested thoroughly), Truth table output customisation, research initiatives taken.

- **Bugs**

As alluded to within the last item in the features list, we have been suffering from some problems in the implementation of our DPLL algorithm. After we had implemented our parsing of general logic, we wanted to take on the DPLL. Our implementation (as discussed in the next section) is not correct in it's output, but when testing, was found to work in some simple cases. Given more time, we are confident we could've got this implementation working perfectly.

6 Research

As alluded to in previous sections, the research component for this assignment involved extending the program from handling only Horn clauses to General propositional logic. This allows the system to be much more useful, and act as a pretty good representation of the type of inference engine found in any given expert system.

6.1 Information about Research Component

For this research component, the assignment required the extension of our inference engine to process general propositional logic. It is expected that this data is not only parsed into the engine, but evaluated in someway, using an algorithm like the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. The goal was to enhance the engine's capability to work with a broader set of logical connectives, specifically including negation , disjunction ('||'), and biconditional ('<=>').

6.2 General Propostional Logic and DPLL

General propositional logic extends propositional logic by incorporating a more generalised set of connectives, allowing for more complex expressions and reasoning capabilities[6]. The DPLL algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic, by using conjunctive normal form (CNF) [5]. It is highly efficient as an algorithm because of it's usage of unit propagation, pure literal elimination, and splitting rules.

6.3 Logic Implementation

The extension of the inference engine to support general propositional logic involved intricate developments in parsing and logical evaluation. This section delves into the specific code implementations that facilitate this functionality. Following parsing, the 'TT' class takes over to generate models and evaluate the logic. The model generation is handled by the 'CreateTruthTable' function, which employs Python's 'itertools.product' to enumerate all possible truth assignments for the symbols identified in the KB.

6.3.1 Logical Evaluation

Each model is evaluated against the KB and the query to determine if the KB entails the query under the current assignment. This evaluation is executed within the 'Check_if_clause_true' function, which iterates through each model and assesses every clause: The 'EvaluateClause' method plays a large part in the parsing and interpretation of each clause based on the logical operators identified by the 'FindMainOperator' function. This function detects and splits clauses around the primary logical operator, facilitating recursive evaluation of complex logical expressions.

6.4 DPLL implementation

Upon init, the 'DPLL' class receives the general knowledge base and query, converting them into a conjunctive normal form (CNF) where necessary.

6.4.1 Satisfiability Check

The 'dpll' method is the central function within the operation of this inference method. Part of it's responsibility lies in recursively evaluating clauses for their satisfiability - in regards to the assignment of values to symbols. The method also handles base cases, unit propagation, pure literal elimination, and branching of the algorithm.

```
# Recursive DPLL algorithm to determine satisfiability
def dpll(self, clauses, assignment):
    if all(self.evaluate_clause(clause, assignment) for clause in clauses):
        return True, 1
    if any(self.evaluate_clause(clause, assignment) == False for clause in clauses):
        return False, 0
```

Figure 8: Base cases and recursion in DPLL

6.4.2 Unit Propagation and Literal Elimination

The DPLL optimises it's search space by using unit propagation and literal elimination. Unit propagation will identify single-variable clauses that are forcing assignments, whilst literal elimination removes redundant or universally true clauses based on found literal occurrences.

```
unit_clause = next((clause for clause in clauses if len(clause) == 1), None)
if unit_clause:
    return self.dpll(simplify_clauses(clauses, unit_clause[0]), ...)
```

Figure 9: Unit propagation in DPLL

6.4.3 Branching and model counting

If the DPLL cannot find a base case or simplifications do not apply yet, it will select a literal and explore both boolean truth values, effectively branching the search, but using boolean logic:

```
literal = next(iter(literals))
result_true, count_true = self.dpll(..., {**assignment, literal: True})
result_false, count_false = self.dpll(..., {**assignment, literal: False})
return (result_true or result_false), (count_true + count_false)
```

Figure 10: Branching in DPLL

We are happy with the attempt made at implementing a DPLL algorithm within the inference engine, but possibly had not planned well enough to implement it and have it work with the pre-existing components well enough. As prefaced prior, the DPLL is working in some conditions, in others, not at all. For a more insightful explanation of the implementation of this method, please see the docstrings within the code, as they provide insight as to all components within the method.

7 Student Contributions

This section of the report will provide detail and context as to the contribution amounts within the completion of this assignment. As a general note, Aston and Hai Nam Ngo are satisfied with the manner in which each other has approached the completion of this assignment, and believe that the contribution was very fairly divided. We can attribute this to our usage of tools like GitHub and Discord, which have allowed us to work together whenever, which has been tremendously helpful for times in which we were stuck with problems. Also to note, we both have attempted at creating each component that has been implemented within the final solution, but evaluated what solution worked better in the overall creation of our inference engine.

- **Aston's contributions**

Components in the final solution that were created by Aston were: Chaining, FC, and BC, Test cases. Aston also constructed and edited this report.

- **Hai Nam Ngo's contributions**

Components in the final solution that were created by Hai Nam Ngo were: TextfileAnalyser, TT, DPLL. Hai Nam Ngo has also assisted Aston with the construction of the report, especially in regard to the research component.

Overall, we are happy with the way that we worked as a group. If we had to decide on a number to mark how much of each members contributions were utilised within the final solution, we would be happy to say the contribution percentage would be Aston - 45%, Hai Nam Ngo - 55%.

8 Conclusion

To conclude, this assignment has provided a challenge for both of us that has allowed us to learn deeper in regards to the construction of inference methods within a propositional logic inference engine. This assignment ensured that we had solid understandings of lectured content which introduced inference methods, as well as advanced algorithms. This knowledge builds off of the content from the start of the semester, where we were tasked with implementing basic searching algorithms within a program. We encountered challenges throughout the entire process of creating our solution, but, our use of teamwork and industry standards in project management allowed us to be fully involved within the process of creating our solution together. The process of creating an inference engine seems abstract until you understand the underlying implications that tools like this have within expert systems, and within the realm of AI as a whole.

9 Acknowledgements/Resources

Within this assignment, we used a great deal of assistance from various sources:

`https://www.youtube.com/watch?v=WtLiZODXD8w&t=255s` - Rich While-Cooper's video on creating and evaluating Python truth tables provided a simplified approach to assigning True/False values in the truth table. His clear explanations and practical examples significantly aided our logical evaluations, making the process more efficient and understandable.

`https://www.youtube.com/watch?v=Smf68icE_as` - NeuralNine tutorial on using the tabulate library helped us create professional-looking tables in Python. The video demonstrated various features of the tabulate library, which enhanced the readability and presentation of our data, leading to more polished and professional output.

`https://swinburne.instructure.com/courses/56950/pages/assignment-helper?module_item_id=3841146` - The Assignment2 helper on Canvas offered valuable insights and hints for tackling the assignment. This resource provided detailed guidance on problem-solving techniques and implementation strategies. Most of our class implementations are modeled after these methods, ensuring our approach was both robust and efficient. Combining these resources enabled us to develop a comprehensive solution.

10 References

[1] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2020.
[2] "Propositional Logic: Logical Analysis using Truth Tables," LibreTexts, 2024. [Online]. Available: `https://human.libretexts.org/Bookshelves/Philosophy/Thinking_Well_-_A_Logic_And_Critical_Thinking_Textbook_4e_(Lavin)/07%3A_Propositional_Logic/7.05%3A_Logical_Analysis_using_Truth_Tables`. [Accessed: May 12, 2024].
[3] F. Hayes-Roth, D. Waterman, and D. Lenat, Building Expert Systems. Reading, MA: Addison-Wesley, 1983.
[4] E. Feigenbaum, The Rise of the Expert Company. New York, NY: Times Books, 1988, p. 317.
[5] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," Commun. ACM, vol. 5, no. 7, pp. 394-397, Jul. 1961, doi: 10.1145/368273.368557.
[6] J. E. Whitesitt, Boolean Algebra and Its Applications. Mineola, NY: Courier Corporation, 2012 [1961]. ISBN 978-0-486-15816-7.