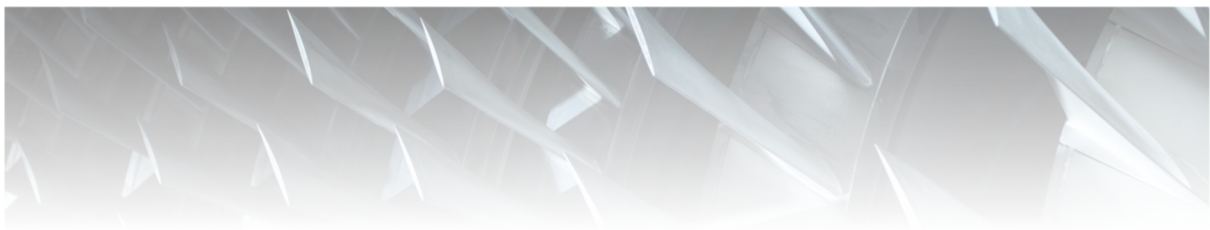


COS30019 – Introduction to Artificial Intelligence

Assignment 2 – Inference Engine

By Jake Varrese – 102578350



Contents

Introduction	3
Features	3
Forward Chaining Method	3
Backward Chaining Method	3
Truth Table (TT) Checking Method	4
Implementation.....	4
IMethod Class.....	5
Sentence Class.....	5
Terms Class	5
Forward Chaining Method Class	6
Backward Chaining Method Class	7
Truth Table Checking Method Class	8
Missing Features	10
Known Bugs.....	10
Testing the programming	11
Testing: Forward Chaining	11
Test 1: Checking Successful Ask valid symbol	11
Test 2: Checking Failure when Asking non-existent symbol	11
Test 3: Checking Successful Ask valid symbol	12
Test 4: Checking Successful Ask valid symbol with a different knowledge base.....	12
Testing: Backward Chaining	12
Test 1: Checking Successful Ask valid symbol	12
Test 2: Checking Successful Ask valid symbol	13
Test 3: Checking Failure when Asking non-existent symbol	13
Test 4: Checking Successful Ask valid symbol with Disjunction operation	13
Test 5: Checking Failed Ask with Conjunction operation.....	14
Testing: Truth Table Checking.....	14
Test 1: Checking Successful TT Checking	14
Test 2: Checking Failed TT Checking	14
Acknowledgements / Resources	15

Introduction

In this documentation, I will outline my progress made on Assignment 2 for COS30019 (Introduction to Artificial Intelligence). The requirement of assignment 2 was to implement an ***inference engine*** for propositional logic based on **three different methods**. The methods I have implemented in my program are *Truth Table (TT) Checking*, *Backward Chaining*, and *Forwarding Chaining*. I have worked through and completed the assigned tasks independently as I chose not to work in a group.

Features

Forward Chaining Method

The Forward Chaining method of inference on a knowledge base is a data-driven approach to determining where an unknown truth is entailed from the knowledge base. It works by using existing data within the knowledge base to extract information to reach a goal state.

Below is the example of the Horn Form KB file to determine whether ***d*** is entailed from the KB:

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

Using the Forwarding Chaining in the program, we can produce the following results:



```
Microsoft Visual Studio Debug Console
[>] ASK KB : d
YES: a; b; p2; p3; p1; d;
```

This states during the execution, a; b; p2; p3; p1; d where entailed from ***KB*** (Knowledge Base)

Backward Chaining Method

The Backward Chaining method of inference on a knowledge base is the opposite of Forward Chaining. Rather than starting with already existing data to extract information to find the goal, Backward Chaining starts with the goal state (therefore known as the goal-driven approach).

Below is the example of the Horn Form KB file to determine whether ***d*** is entailed from the KB:

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

Using the Backward Chaining in the program, we can produce the following results:



```
Microsoft Visual Studio Debug Console
[>] ASK KB : d
YES: p2; p3; p1; d;
```

This states during the execution, p2; p3; p1; d where entailed from ***KB*** (Knowledge Base)

Truth Table (TT) Checking Method

The Truth Table (TT) Checking method of inference determines if the ASKed query is entailed by the knowledge base. This method of inference requires a lot more calculations as it searches through every possible model of the truth table. In the example provided, there are 2^{11} (2,048) rows in the truth table. The method requires far more computations compared to Forward and Backward Chaining.

Below is the example of the Horn Form KB file to determine whether **d** is entailed from the KB:

TELL

$p2 \Rightarrow p3$; $p3 \Rightarrow p1$; $c \Rightarrow e$; $b \& e \Rightarrow f$; $f \& g \Rightarrow h$; $p1 \Rightarrow d$; $p1 \& p3 \Rightarrow c$; a; b; p2;

ASK

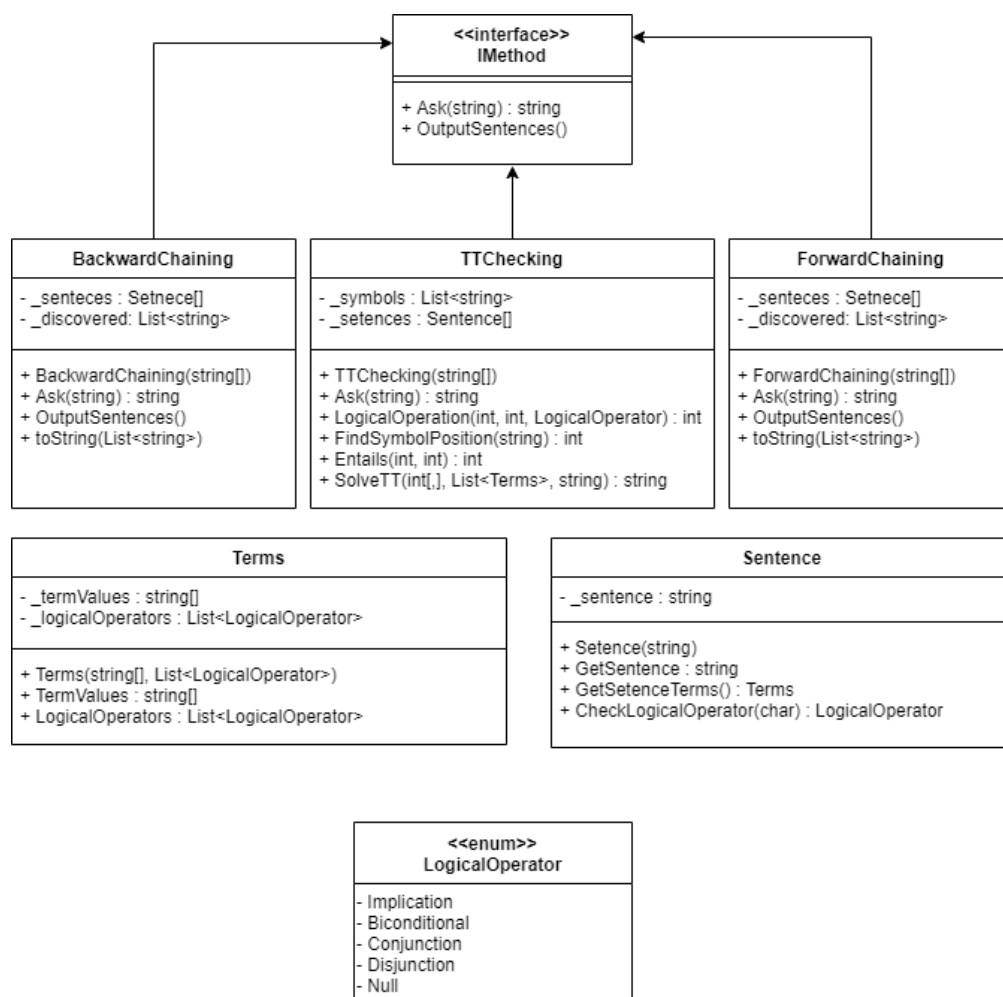
d

Using the TT Checking in the program, we can produce the following results:

```
Microsoft Visual Studio Debug Console
[>] ASK KB : d
YES: 3
```

This states that there are **3** models where KB entails d.

Implementation



IMethod Class

The IMethod class in my program provides the interface structure used by the three different method classes (TTCheking, ForwardChaining, BackwardChaining). This interface declares two functions used by the different methods.

- **Ask(string)**
- **OutputSentence()**

Sentence Class

The Sentence class is used to store information on individual sentences within the knowledge base. The Sentence class is responsible for knowing the sentence string and building Terms objects based on the value of the sentence.

The sentence class includes the following methods:

- **GetSentence()** – Returns string value of raw sentence
- **GetSentenceTerms()** – Returns a Terms object (will be explained) that is built based on the value of the sentence string.
- **CheckLogicalOperator(char)** – Accepts a char value as a parameter and returns a LogicalOperator value (enum) based on the value parsed in. *(Values such as =, <, &, | where each represents Implication, Biconditional, Conjunction, and Disjunction. The reason this function assumes = as the symbol Implication (=>) is the only operator that beings with the = sign. The same applies to Biconditional, where the first symbol in the operator is <.)*

Terms Class

The Terms class is responsible for breaking elements of the sentence object into their atomic parts. The Terms class breaks sentences up into the individual variables used in the sentence (like a, b, c, p1, p2) then it finds the different logical operators used within the sentence as which variables they're associated with.

The Terms class includes the following methods:

- TermValues()** – Returns an array of string values consisting of all the different variables within the sentence.
- LogicalOperators()** – Returns a List of LogicalOperator values associated with the variables within the sentence.

Forward Chaining Method Class

The Forward chaining class (ForwardChaining) inherits from the IMethod interface and implements the methods Ask() and OutputSetences(). The Forward Chaining class implements the behavior of the Forward Chaining Method explained on page 2.

The Forward Chaining class includes the following private variables:

- **_setences : Setence[]** (An array of sentence objects obtained from the knowledge base)
- **_discovered : List<string>** (A list of string objects which stores variables which have already been observed during the process)

Below I will explain how the methods included within the Forward Chaining class fulfill the behaviors to receive appropriate output.

- **Ask(string aQuery)** – This function accepts a one string parameter which will be the query value ASKed to produce a list of propositional symbols that are entailed from the knowledge base. The Ask() method initializes the list at the beginning of the main operations called **entailed** (List<string>).

The method uses a for-loop, to determine whether the query is entailed from the knowledge base. Using the for-loop as an index, we create a new term object via the Sentence object function **GetSentenceTerms()**. This is stored in the variable **sentenceComponants**.

The method then stores the last term value in a string and creates a boolean variable called flag, initializing it as true.

Another for-loop is enter using j as an index with the condition that (index j less than the number of term values within the **sentenceComponant** object minus one since we are checking the last value).

With each iteration of the loop, the program checks where the term value is not contained in the **discovered** list. If this condition is met, the flag is set to false and the loop is broken. After the for-loop, the flag is checked if it's true and if it's true, the **lastTermElement** is checked whether it's not contained within the **discovered** list and then it is added to both the **entailed** and **discovered** list

After the first if-statement, the exit condition is checked seeing if the last element of the **entailed** list is equal to the **aQuery** (the argument value) and if this is true, all the elements in the **entailed** list are return as a concatenated string value (i.e a, b, c, etc).

Backward Chaining Method Class

The Backward chaining class (BackwardChaining) inherits from the IMethod interface and implements the methods Ask() and OutputSetences(). The BackwardChaining class implements the behavior of the Backward Chaining Method explained on page 2.

The Backward Chaining class includes the following private variables:

- **_setences : Setence[]** (An array of sentence objects obtained from the knowledge base)
- **_discovered : List<string>** (A list of string objects which stores variables which have already been observed during the process)

Below I will explain how the methods included within the Backward Chaining class fulfill the behaviors to receive appropriate output.

- **Ask(string aQuery)** – This function accepts a one string parameter which will be the query value ASKed to produce a list of propositional symbols that are entailed from the knowledge base. The Ask() method initializes the list at the beginning of the main operations called **entailed** (List<string>). Unlike the Forward Chaining class, the Backward chaining class also includes another list object called **termFocus** (List<string>) and before the main for-loop begins, the aQuery is added to the **discovered, entailed and termFocus** list.

To explain the Ask() function in the Backward Chaining class, I will step through the example to show how it produces the correct result.

If we consider the provided problem in **test_HorbKB.txt** we are given a knowledge base with the following rules:

1. $p2 \Rightarrow p3$;
2. $p3 \Rightarrow p1$;
3. $c \Rightarrow e$;
4. $b \& e \Rightarrow f$;
5. $f \& g \Rightarrow h$;
6. $p1 \Rightarrow d$;
7. $p1 \& p3 \Rightarrow c$;

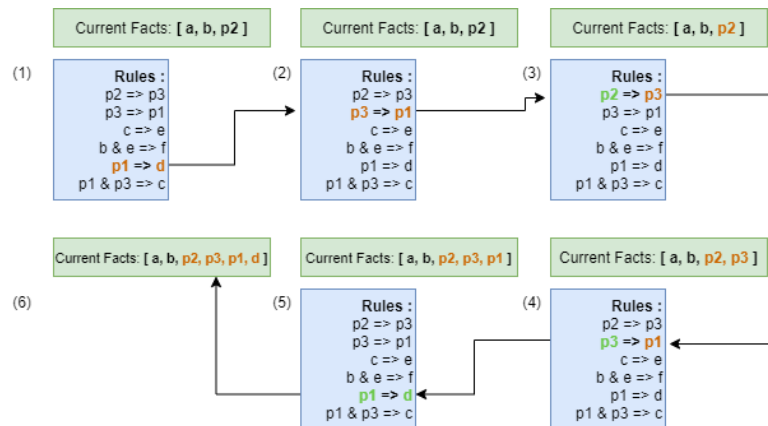
The following are the already known facts in the database are:

1. a
2. b
3. p2

The file ASKs the knowledge base:

- d

Using this information can be used to begin the Backward Chaining method. The following figure illustrates the process of my code. It demonstrates how the determining whether the “ASKed” symbol is entailed from the knowledge base by breaking up its components into sub-goals (represented by each block in the figure) to find a list of proposition symbols which have been discovered during the execution of the algorithm.



The first step in the execution of the algorithm is to check if the asked symbol is discovered (which is checked at each iterator of the sub-goal creation), then if not find the symbol which implies this symbol. In the first iteration, we are checking if **p1** implies **d** although the values of **p1** are not yet known so we can not finish the execution yet. The program now needs to find where **p1** is implied and repeat this solution until the program can successfully trace back to the goal state.

At step (3), the algorithm is checking whether **p2** implies **p3** and since **p2** is a known fact (discovered symbol) and it completes the operation and adds the **p3** symbol to the facts (discovered list). Subsequently, step (4) follows a similar approach by checking whether **p3** implies **p1** which is a successful operation as **p3** is a known fact. Finally, the algorithm checks if **p1** implies **d** and this operation would execute successfully, adding **d** to the facts which would conclude the algorithm.

The program's output would be: **YES: p2; p3; p1; d**

Truth Table Checking Method Class

The Truth Table Checking method class (TTChecking) inherits from the IMethod interface and implements the methods Ask() and OutputSentences(). This method of inference on the knowledge base takes a different approach compared to both Forward Chaining and Backward Chaining (as explained on page 4).

The TTChecking class includes the following private variables:

- **_symbols : List<string>** - data structure uses to store all symbols (ex. A, b, c, d, e, p1, p2)
- **_sentences : Sentence []** – data structure used to store all sentences in the KB.

The TTChecking Class includes the following methods:

- **GenerateTTValues(int[,] aTT, int aLimit) : int[,]**

The GenerateTTValue function is responsible for generating the base values for a given 2D array initialized with the value 0 for all spaces. The parameters accepted in this function are the 2D int array which will be manipulated and another int value called limit to restrict changing value of columns which are dedicated for sentence calculations (columns which contain a sentence and are not a symbol). This function returns the result 2D int array.

To explain the concept of the **limit** parameter I will illustrate a small example:

A	B	C	A=>C	C & A	C B
0	0	0	*	*	*
0	0	1	*	*	*
0	1	0	*	*	*
0	1	1	*	*	*
1	0	0	*	*	*
1	0	1	*	*	*
1	1	0	*	*	*
1	1	1	*	*	*

In the graph above, the limit would be **3** as A, B and C are the symbols while A=>C, C&A and C | A are the sentences. As you can see, the GenerateTTValue function will ignore the other 3 columns and generate base values for the symbols (being 3 * (2^3) values).

- **LogicalOperation(int aT1, int aT2, LogicalOperator aOperator)**

The LogicalOperation function is responsible for completing different Logical Operations such as Bicondition, Implication, Conjunction, and Disjunction. The function simply returns an int value (0 for false 1 for true) depending on the output from the two int input values.

- **FindSymbolPosition(string aSymbol)**

The FindSymbolPosition function takes a string potentially representing a symbol in the _symbols list and returns the location associated with that value in integer form. If that symbol is not found, the value of -1 is returned.

- **SolveTT (int[,] aTT, List<Terms> aKB, string aQuery) : string**

The SolveTT function is responsible for assigning values to the columns where the sentence value resides by using the values generated by the GenerateTT function. This function takes the parameters of a 2D int array which represents the Truth Table consisting base values, the list of Terms objects, and the string query in which the algorithm checks to see if the knowledge base entails the query symbol for each model. This function returns a string that represents the number of worlds where the KB entails aQuery.

- **Entails(int aKBValue, int aQueryValue) : int**

The Entails function is responsible for checking if the given row's KB value entails the value of the aQuery value. This function will return an integer value (0 for false and 1 for true)

- **Ask(string aQuery) : string**

The Ask function is responsible for generating the Truth table (using the GenerateTT function) and sorting the symbols in ascending order and creating the 2D int array used to represent the Truth table. The ask function delegates the job of solving the truth table and obtaining the number of models where the KB entails aQuery to the SolveTT function. The Ask function returns a string containing the number of models where KB entails aQuery. If there are no models, it returns an empty string.

Missing Features

I have failed to implement:

- Bidirectional operator support for Forward and Backward Chaining
- Negation operator support for TT Checking, Forward chaining, and Backward chaining

Known Bugs

As of this moment, I have no know bugs with the execution of this program with all three inference engine methods.

END OF SECTION

Testing the programming

The following test cases have been produced to test the effectiveness of the three different methods include in the inference engine program.


Testing: Forward Chaining

Test 1: Checking Successful Ask valid symbol

The code below checks whether the following Horn Form KB entails the symbol d. The expected result from this should be "a; b; p2; p3; p1; d;".

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "d")]  
public void Test1ForwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new ForwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("a; b; p2; p3; p1; d; ", result);  
}
```

Test Name: Test1ForwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "d")


Test Outcome:  Passed

Test 2: Checking Failure when Asking non-existent symbol

The code below checks whether the following Horn Form KB doesn't entail the symbol z. The expected result from this should return null;

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "z")]  
public void Test2ForwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new ForwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual(null, result);  
}
```

Test Name: Test2ForwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "z")


Test Outcome:  Passed

Test 3: Checking Successful Ask valid symbol

The code below checks whether the following Horn Form KB doesn't entail the symbol p3. This check is used to see if the results work with different Asks compared to the provided Ask symbol

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "p3")]  
public void Test3ForwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new ForwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("a; b; p2; p3; ", result);  
}
```

Test Name: Test3ForwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "p3")


Test Outcome:  Passed

Test 4: Checking Successful Ask valid symbol with a different knowledge base

The code below checks whether the following Horn Form KB entails the symbol f in a different knowledge base. This test ensures the results aren't varying or incorrect with different KBs.

```
[TestCase(new string[] { "z => c", "a => z", "z & c => f", "a", "x" }, "f")]  
public void Test4ForwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new ForwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("a; x; z; c; f; ", result);  
}
```

Test Name: Test4ForwardChaining(["z => c", "a => z", "z & c => f", "a", "x"], "f")

Test Outcome:  Passed


Testing: Backward Chaining

Test 1: Checking Successful Ask valid symbol

The code below checks whether the following Horn Form KB entails the symbol d. The expected result from this should be "p2; p3; p1; d; ".

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "d")]  
public void Test1BackwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new BackwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("p2; p3; p1; d; ", result);  
}
```

Test Name: Test1BackwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "d")


Test Outcome:  Passed

Test 2: Checking Successful Ask valid symbol

The code below checks whether the following Horn Form KB entails the symbol c. The expected result should be "p2; p3; p1; c; "

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "c")]  
public void Test2BackwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new BackwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("p2; p3; p1; c; ", result);  
}
```

Test Name: Test2BackwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "c")


Test Outcome:  Passed

Test 3: Checking Failure when Asking non-existent symbol

The code below checks whether the following Horn Form KB doesn't entail a non-existent symbol (v). The expected result is null.

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2" }, "v")]  
public void Test3BackwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new BackwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual(null, result);  
}
```

Test Name: Test3BackwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "v")


Test Outcome:  Passed

Test 4: Checking Successful Ask valid symbol with Disjunction operation

The code below checks whether the following Horn Form KB entails the symbol c. The symbol c is implied from $z \vee p1$. As the p1 symbol will be a discovered value, it should not matter if symbol z is known. The expected result is "p2; p3; p1; c; ".

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "z || p1 => c", "a", "b", "p2" }, "c")]  
public void Test4BackwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new BackwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("p2; p3; p1; c; ", result);  
}
```

Test Name: Test4BackwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "c")


Test Outcome:  Passed

Test 5: Checking Failed Ask with Conjunction operation

The code below checks whether the following Horn Form KB entails the symbol c. The symbol c is implied from p1 & z. As p1 should be discovered at this state but z is an unknown symbol the expected result is null;

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & z => c", "a", "b", "p2", }, "c")]  
public void Test5BackwardChaining(string[] aSentences, string aQuery)  
{  
    Method = new BackwardChaining(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual(null, result);  
}
```

Test Name: Test5BackwardChaining(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "c")

Test Outcome:  Passed


Testing: Truth Table Checking

Test 1: Checking Successful TT Checking

The code below checks whether KB entails p and check the number of models where this is true. The expected result is 1

```
[TestCase(new string[] { "p => r", "q & r => p", "q", "r", }, "p")]  
public void Test1TTChecking(string[] aSentences, string aQuery)  
{  
    Method = new TTChecking(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual("1", result);  
}
```

Test Name: Test1TTChecking(["p => r", "q & r => p", "q", "r"], "p")


Test Outcome:  Passed

Test 2: Checking Failed TT Checking

The code below checks whether KB entails z. Since the symbol isn't entailed from KB and doesn't exist, the expected result is null.

```
[TestCase(new string[] { "p2 => p3", "p3 => p1", "c => e",  
    "b & e => f", "f & g => h", "p1 => d",  
    "p1 & p3 => c", "a", "b", "p2", }, "z")]  
public void Test2TTChecking(string[] aSentences, string aQuery)  
{  
    Method = new TTChecking(aSentences);  
  
    string result = Method.Ask(aQuery);  
  
    Assert.AreEqual(null, result);  
}
```

Test Name: Test2TTChecking(["p2 => p3", "p3 => p1", "c => e", "b & e => f", "f & g => h", ...], "z")

Test Outcome:  Passed

Acknowledgments

Swinburne University Faculty

Both **A/Prof. Bao Vo and** (COS30019 Convenor and Lecturer) and **Mahbuba Afrin** (COS30019 Tutor) have worked exceptionally to teach and demonstrate the concepts of AI and explaining concepts

Draw.io

Resource link: <http://www.draw.io/> This resource assisted me in making some of the diagrams and UML featured in this assignment which assisted me in understanding the task ahead and to present my thinking.

References

Margaret Rouse - Forwarding Chaining Explanation

<https://whatis.techtarget.com/definition/forward-chaining>

Backward Chaining Information

<https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai>

END OF REPORT