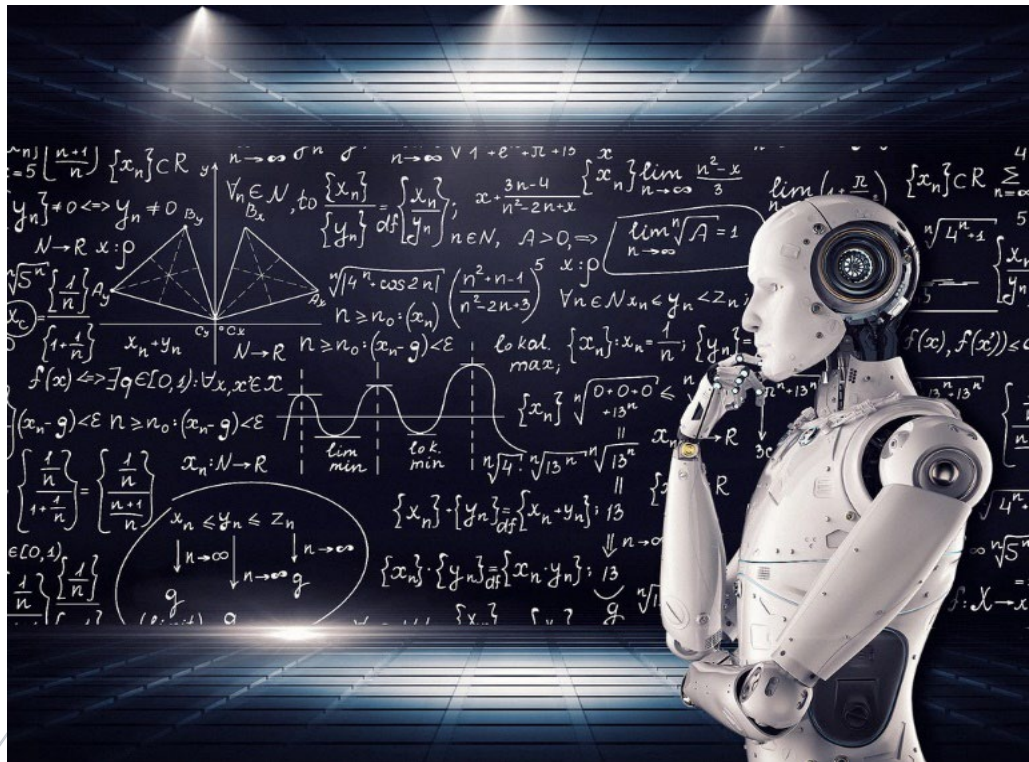


3/16/2022

# Inference Engine

## Assignment 2(Introduction to Artificial Intelligence)



### ISHAAN MANGA

102878937

**Tutorial** : Tuesday 10:30 – 12:30

**Tutor** : Charles Harold

**Unit** : COS30019 – Introduction to Artificial Intelligence

# Table of Contents

Summary .....	2
Instructions .....	2
Introduction .....	2
Forward chaining.....	3
Backward chaining.....	3
Implementation .....	4
ForwardChaining.cs .....	4
BackwardChaining.cs.....	5
TruthTable.cs.....	6
Program.cs.....	6
KnowledgeBase.cs .....	7
Test Cases:.....	7
test3.txt .....	7
test6.txt .....	8
Research.....	8
Modified Truth Table.....	8
DPLL .....	9
Features and Bugs.....	10
Features.....	10
Bugs .....	10
Missing.....	11
Conclusion .....	11
Acknowledgement/ Resources .....	12
References.....	13

## Summary

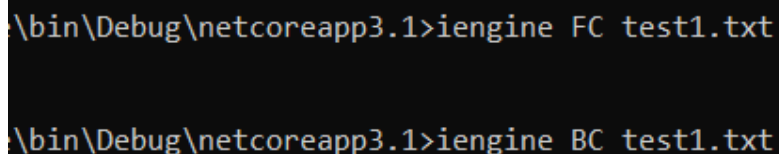
This is an Individual Assignment. We need to create an inference engine for propositional logic that uses Truth Table (TT) checking as well as Backward Chaining (BC) and Forward Chaining (FC) methods. Our inference engine will accept a Horn-form Knowledge Base KB and a proposition symbol query  $q$  as arguments and decide if  $q$  can be entailed from KB. We have been given text files which shows knowledgebase and query.

## Instructions

The program is coded in **C# language** using principles of Object-Oriented Programming in Console Application Visual Studio. Follow the set of instructions to successfully run the provided code.

- Build the program and then explore the bin folder where we will see iengine command application.
- Make sure our required text files are located in that same folder.
- Explore the path of the file in command prompt.
- In cmd, we execute the following arguments to run our code:
  - Name of the Project, i.e. iengine
  - Name of the method, eg: FC, BC, TT etc.
  - Name of the text file we wish to read

Overall, our argument should look like **> iengine FC test1.txt**



```
.\bin\Debug\netcoreapp3.1>iengine FC test1.txt  
  
.\bin\Debug\netcoreapp3.1>iengine BC test1.txt
```

## Introduction

Artificial intelligence is a cutting-edge technology that is now being employed in a wide range of fields. AI requires a particular component – an inference engine – to correctly interpret and use the data included in semantic web texts. (Demchenko, 2021)

An inference engine makes a judgement based on the facts and rules included in an expert system's knowledge base or a deep learning AI system's algorithm. In contrast to the fact gathering or learning side of the system, the inference engine is the processing component.

Inference engines use two main approaches to obtain new information:

- Forward Chaining
- Backward Chaining

**Forward chaining** begins with the data that is currently accessible and then utilises inference rules to extract more data until a goal is met. A forwards chaining inference engine searches the inference rules until it finds one with a known true antecedent. This method will be repeated until the desired result is achieved.

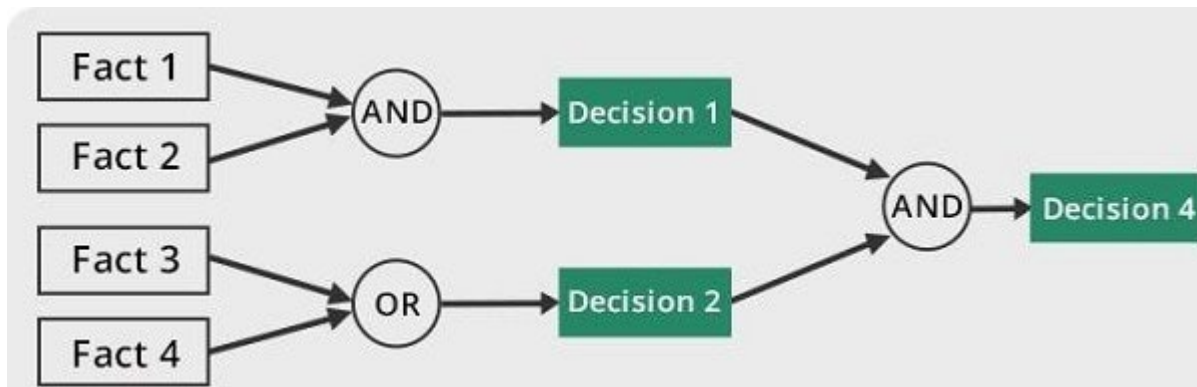


Figure 1 Forward Chaining

**Backward chaining** entails going backwards from the endpoint or objective to the stages that lead to it. This method of chaining begins with the goal and works backwards to understand the steps that were done to achieve it.

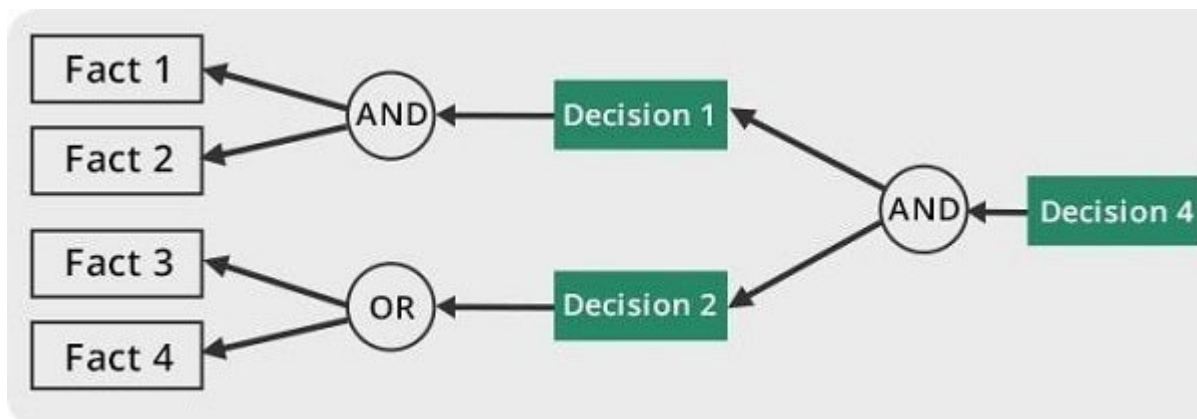


Figure 2 Backward Chaining

Comparison	Forward Chaining	Backward Chaining

<b>Approach</b>	Follows bottom-up approach	Follows top-down approach
<b>Strategy</b>	Applies Breadth First Strategy	Applies Depth First Strategy
<b>Technique</b>	Data-Driven technique	Goal-driven technique
<b>Goal</b>	Goal is to get conclusion	Goal is to get possible facts
<b>Speed</b>	Slow as it has to use all the rules	Fast as it has to use only a few rules
<b>Number of Conclusion</b>	can generate an infinite number of possible conclusions	generates a finite number of possible conclusions

## Implementation

ForwardChaining.cs

```

function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                    inferred, a table, indexed by symbol, each entry initially false
                    agenda, a list of symbols, initially the symbols known to be true

  while agenda is not empty do
    p ← POP(agenda)
    unless inferred[p] do
      inferred[p] ← true
      for each Horn clause c in whose premise p appears do
        decrement count[c]
        if count[c] = 0 then do
          if HEAD[c] = q then return true
          PUSH(HEAD[c], agenda)

  return false

```

Figure 3 Forward Chaining Algorithm

From the lecture notes of week 7, we can see the algorithm of Forward Chaining Method. In our project, we created a ForwardChaining.cs class where we made a function PL\_FC\_Entails() which is the Main Algorithm for Forward Chaining that returns a Boolean value. In the ForwardChaining Constructor, we declared local variables of knowledge base and query. The FCSolution function run the FC algorithm, it gets the comma separated path value and prints list of symbols (a, b, p2, p3, p1, d) if the result is YES.

The InitialiseAgenda() Queue determines whether a clause in the knowledge base has no Premise, If this is case, adds clause to agenda.

InitialiseCount() Dictionary initializes count for a clause with the number of items in its Premise and stores it in a dictionary. And InitialiseInferred() Dictionary initializes all symbols to be false in a dictionary.

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine FC test1.txt
YES: a,b,p2,p3,p1,d
```

## BackwardChaining.cs

```
function BACK-CHAIN(KB, q) returns a set of substitutions
    BACK-CHAIN-LIST(KB, [q], {})



---


function BACK-CHAIN-LIST(KB, qlist,  $\theta$ ) returns a set of substitutions
    inputs: KB, a knowledge base
             qlist, a list of conjuncts forming a query ( $\theta$  already applied)
              $\theta$ , the current substitution
    static: answers, a set of substitutions, initially empty

    if qlist is empty then return { $\theta$ }
     $q \leftarrow \text{FIRST}(\text{qlist})$ 
    for each  $q'_i$  in KB such that  $\theta_i \leftarrow \text{UNIFY}(q, q'_i)$  succeeds do
        Add COMPOSE( $\theta, \theta_i$ ) to answers
    end
    for each sentence  $(p_1 \wedge \dots \wedge p_n \Rightarrow q'_i)$  in KB such that  $\theta_i \leftarrow \text{UNIFY}(q, q'_i)$  succeeds do
         $\text{answers} \leftarrow \text{BACK-CHAIN-LIST}(\text{KB}, \text{SUBST}(\theta_i, [p_1 \dots p_n]), \text{COMPOSE}(\theta, \theta_i)) \cup \text{answers}$ 
    end
    return the union of BACK-CHAIN-LIST(KB, REST(qlist),  $\theta$ ) for each  $\theta \in \text{answers}$ 
```

30

Figure 4 Backward Chaining Algorithm

For Backward Chaining Algorithm, we created BackwardChaining.cs class where we created a class constructor in which we declared local variable for Knowledge Base and query. The main algorithm for BC is done in PL\_BC\_Entails() which passes parameter (string query) and returns a Boolean value. The BCSolution() function gets comma separated path value and prints the path symbol (p2, p3, p1, d) if its YES.

The Stack InitialiseAgenda() initializes an agenda stack to contain the initial query.

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine BC test1.txt
YES: p2,p3,p1,d
```



## TruthTable.cs

```

function TT-ENTAILS?( $KB, \alpha$ ) returns true or false
     $symbols \leftarrow$  a list of the proposition symbols in  $KB$  and  $\alpha$ 
    return TT-CHECK-ALL( $KB, \alpha, symbols, []$ )

function TT-CHECK-ALL( $KB, \alpha, symbols, model$ ) returns true or false
    if EMPTY?( $symbols$ ) then
        if PL-TRUE?( $KB, model$ ) then return PL-TRUE?( $\alpha, model$ )
        else return true
    else do
         $P \leftarrow$  FIRST( $symbols$ );  $rest \leftarrow$  REST( $symbols$ )
        return TT-CHECK-ALL( $KB, \alpha, rest, EXTEND(P, true, model)$ ) and
            TT-CHECK-ALL( $KB, \alpha, rest, EXTEND(P, false, model)$ )

```

Figure 5 Truth Table Algorithm

Again from our week 7 lecture slides, we can see the algorithm for truth table. So, we created a class TruthTable.cs in which we again created a class constructor and declared local variables of knowledge base and query. The main algorithm for TT is TT\_Entails() which returns a Boolean value, here we created a List Symbol to store a list of the propositional symbols in KB and alpha. The TTSolution() function prints model count if TT\_Entails is True. The Dictionary Extend initialise model dictionary with each symbol value of true.

```

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine TT test1.txt
YES: 3

```

## Program.cs

The first step was to read the text file and take arguments in command prompt. We used if-else to distinguish the knowledge base and query, if text on first line is "TELL", then the following line will be knowledge base; and if the third line is "ASK", next line shows the query. We separated Antecedent(premise) and Consequent(conclusion), trimmed any extra unwanted spaces and added it into a List.

We created separate classes for each method and called them using switch method in program.cs.

The ProblemReader() function

- Reads the file for knowledgebase and query,
- Initialises clause object for each premise and conclusion.

- Builds knowledgebase of clauses

## KnowledgeBase.cs

In this class, the List GetFacts() Function returns a list of known facts from the knowledge base, while the List Getcharacters() function extract the symbols that are contained within the Knowledge Base.

## Test Cases:

We created a `test2.txt` file to showcase what if the result is False, i.e. NO. The text file had following statements:

TELL

$a \Rightarrow c; b \Rightarrow c; c \Rightarrow p1; b;$

ASK

d

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine FC test2.txt
NO
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine BC test2.txt
NO
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine tt test2.txt
NO
```

*test2.txt*

Similarly, we created we more test text files to run and test our program.

## test3.txt

TELL

$p1 \& p2 \& p3 \Rightarrow p4; p5 \& p4 \Rightarrow p7; p1 \Rightarrow p2; p1 \& p2 \Rightarrow p3; p5 \& p1 \Rightarrow p6; p2 \& p3 \Rightarrow p5; p1;$

ASK

p7

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine fc test3.txt
YES: p1,p2,p3,p4,p5,p7
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine BC test3.txt
YES: p5,p5,p1,p2,p5,p1,p2,p1,p5,p1,p1,p1,p2,p3,p4,p7
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine TT test3.txt
YES: 1
```

*test3.txt*



## test6.txt

This text file is for extended research implementation of truth table, so this will not work for backward chaining algorithm.

TELL

(a & b) => c; a; ~c;

ASK

c

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine TT test6.txt
NO

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine BC test6.txt
Unhandled exception. System.NullReferenceException: Object reference not set to an instance of an object.
   at iengine.BackwardChaining.PL_BC_Entails(String query) in C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\BackwardChaining.cs:line 59
   at iengine.BackwardChaining.BCSolution() in C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\BackwardChaining.cs:line 79
   at iengine.Program.Main(String[] args) in C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\Program.cs:line 56

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine FC test6.txt
NO
```

## Research

### Modified Truth Table

In the research, we extended our truth table to deal with general knowledge bases dealing with OR, NOT and bi-conditionals. Implementing it was quite challenging. There are three classes namely NewKnowledgeBase.cs, ModifiedTruthTable.cs and ExpressionResearch.cs. To run the Modified Truth Table, we need to pass project name followed by method name (here it is **MTT**) followed by text file name.

In NewKnowledgeBase.cs, we have just created readonly expression property; and in ModifiedTruthTable.cs we defined a class constructor with local variables of knowledge base and query. Classes are well commented and we can easily understand the use of a particular function. As usual TT\_Check\_All() function calculate model count and TT\_Entails() is the main algorithm for modified truth table. MTTSolution() prints model count if its YES

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine MTT test4.txt
NO

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine MTT test3.txt
YES: 2

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine MTT test2.txt
NO

C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine MTT test1.txt
YES: 6
```

## DPLL

In artificial intelligence, the DPLL algorithm is commonly utilised. The DPLL algorithm should be understood in order to better comprehend and work on the theoretical side of artificial intelligence and decision science.

The algorithm of DPLL (Davis-Putnam-Lagemann-Loveland) is taken from the book Artificial Intelligence A modern approach.

```

function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, [])



---


function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, EXTEND(P, value, model))
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, EXTEND(P, value, model))
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, EXTEND(P, true, model)) or
    DPLL(clauses, rest, EXTEND(P, false, model))
  
```

Figure 6 DPLL Algo, Source: Intro to AI A Modern Approach

The recursive calls are based on the DPLL, which is essentially a backtracking algorithm.

The method is constructing a solution while attempting assignments; you have a partial solution, which may or may not be successful as time goes on. The algorithm's brilliance is in how it constructs the partial solution. (Amit, 2012)

The DPLL technique is commonly used to determine whether or not a logical assertion is unsatisfiable. The DPLL technique speeds up our calculations while also reducing the number of situations we need to check.

We try to check the most likely model for the statement using the DPLL algorithm by making some propositions true or untrue. If we see that, even in such scenario, the proposition remains unsatisfiable; we conclude that the proposition is unsatisfiable.

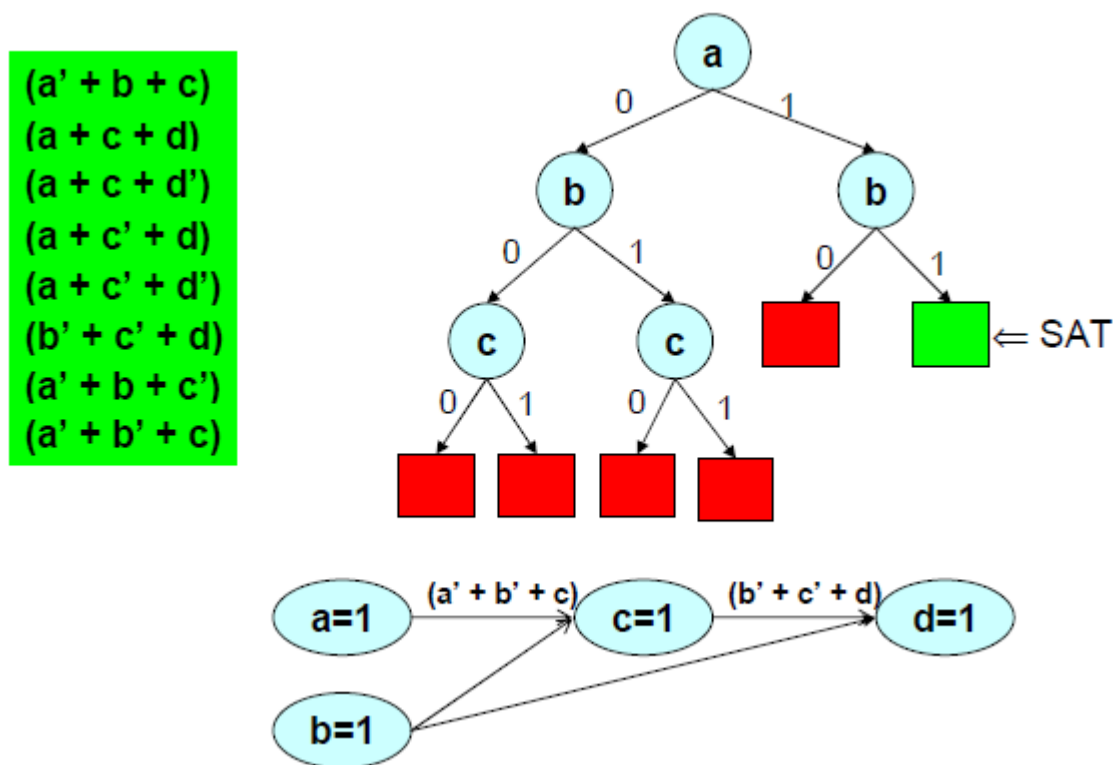


Figure 7 SOURCE: Wikipedia

## Features and Bugs

### Features

Good programming practice have been followed with appropriate and well-defined comments to understand the functionality of code. The files have well defined names and precise codes. All errors occurring in the code were successfully removed.

### Bugs

One bug that I found in the code was for Backward Chaining. If in test file (test11.txt), we put the following statement:

TELL

$d \Rightarrow d$

ASK

$d$

It will give NO for all methods except Backward Chaining. BC gives the following output:

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\engine\bin\Debug\netcoreapp3.1>engine FC test11.txt
NO
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\engine\bin\Debug\netcoreapp3.1>engine BC test11.txt
YES: d
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\engine\bin\Debug\netcoreapp3.1>engine TT test11.txt
NO
```

Since in knowledge base,  $d \Rightarrow d$  ( $d$  implies  $d$ ), it should form recursion and repeat itself again and again, therefore should not print YES. Hence, Backward chaining is not always correct.

## Missing

Another thing I found is when in I use test4.txt file i.e.

TELL

$a \Rightarrow d$

ASK

$d$

It shows NO for all method (which is correct), except Backward Chaining. Since here it is not given that 'a' is true, it should be taken as false and return a NO, but in BC algorithm, it is considering 'a' as True. This case would be perfect if in text file, we would have mentioned 'a' to be true, i.e.  $a \Rightarrow d; a;$  where single  $a;$  represent that the symbol is TRUE.

```
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine fc test4.txt
NO
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine bc test4.txt
YES: a,d
C:\Users\hp\OneDrive - Swinburne University\sem 3\Intro to AI\Assign2\iengine\bin\Debug\netcoreapp3.1>iengine tt test4.txt
NO
```

## Conclusion

An inference engine is a component of an artificial intelligence system that uses logical rules to deduce new information from a knowledge base.

Forward chaining works by identifying some well-known facts and then identifying implications when those well-known facts are part of the premise. If all of the premise's facts are known, the conclusion is added to the list of known facts. This process continues until the query is resolved or no more consequences can be investigated.

Backward chaining operates in reverse, determining the implications of the query's conclusion and then checking each of the premise's components. It goes over each item in the premise, backwards chaining until either an answer is found or there is nothing else to check.

Truth Table Checking works by allocating all of the atomic sentences in the knowledge base to all of the potential true and false combinations. We check whether the query is valid if all the clauses in a single row of the kb are true. If the question is true in the model every time the knowledge involves the model, then the query is true.

Overall it was a good assignment. Writing algorithm for Truth Table for me was a challenge where I had to go through lecture notes and search each concept over Internet to dive deep into the topic. Research implementation i.e. extended Truth Table was also quite tricky and I was getting a lot of errors initially, but somehow managed to resolve all of them successfully.

**NOTE:** This is an Individual Assignment done by Ishaan Manga (102878937)

## Acknowledgement/ Resources

**Bao Vo:** This assignment would not have been possible without his lectures and extensive educational resources.

**Charles Harold,** whose advice, direction, and flawless teaching in tutorials were vital to this project.

**Swinburne University of Technology,** for providing access to its library, from which substantial expertise in the field of artificial intelligence was gathered.

The infamous book "**Artificial Intelligence: A Modern Approach**" (Russell & Norvig, 2009), by Peter Norvig and Stuart Russel, provided significant background reading and an introduction to the field of Inference Engine.

## References

Amit, 2012. *StackOverflow*. [Online]

Available at: <https://stackoverflow.com/questions/12547160/how-does-the-dpll-algorithm-work>  
[Accessed 26 May 2022].

Demchenko, A., 2021. *DATAOX*. [Online]

Available at: <https://data-ox.com/what-are-inference-engines-in-artificial-intelligence/>  
[Accessed 16 05 2022].