

Swinburne University Of Technology  
Semester 1, 2023

COS30019

Introduction to Artificial Intelligence



# **Assignment 2 Report**

## **Inference Engine**

**Authors**

**Team 36**

Nghia Hieu Pham	103533868
Kim Duong Pham	103792850

## Table of Contents

Features/Bugs/Missing.....	3
Truth Table Algorithm.....	4
Forward Chaining .....	5
Backward Chaining.....	5
Test Cases .....	7
Research Initiative.....	11
General Knowledge handling.....	11
Team Summary Report.....	13
Acknowledgements/Resources .....	14

## Features/Bugs/Missing

The program was written in C# using object-oriented programming style. All the required algorithms (Truth Table checking, Backward Chaining, Forward Chaining) were implemented in the program. The program takes a Knowledge Base (**KB**) and a query (**q**) as arguments in the following format:

ASK

**KB (clauses separated by “;”)**

TELL

**q**

Then the program will determine whether q can be entailed from KB. If q cannot be entailed, the program will print out “NO”, and if q can be entailed, the outputs will be in the following format according to each algorithm:

- Forward Chaining & Backward Chaining: “YES:” + list of propositional symbols entailed from KB that has been found during the execution of the algorithm
- Truth Table algorithm: “YES: “+ number of models of KB

The program can also handle general knowledge bases with the Truth Table algorithm, using an infix and postfix parser along with a corresponding evaluation function.

To run the program, the user should execute the following command:

***InferenceEngine.exe methodname filename***

Where:

- ***InferenceEngine.exe***: the name of the program’s .exe file
- **methodname**: specifies the algorithm
  - **Truth Table**: TT
  - **Forward Chaining**: FC
  - **Backward Chaining**: BC
- **filename**: name of the input text file

Details of the implementation of the algorithms are discussed in the following section.

### Truth Table Algorithm

The Truth Table enumeration algorithm will perform depth-first enumeration of all models, which is sound and complete. The enumeration process is executed recursively by calling the function `CheckAll`, with the knowledge base, query, list of symbols in the knowledge base and an empty model set as the initial input in the `Entails` function.

Symbols are repeatedly assigned True and False values, while the function is called recursively with the new symbol list without the assigned symbols and the newly extended model. The process will repeat until the algorithm has reached the end of branches, with the completed models with all the symbols assigned with either True or False as a result.

After that, the model will be evaluated by the `PL_True` function, which will return true if it's true within the given knowledge base. Then, it will be evaluated with the query, so that if it is a model of the query, then the `Entails` function returns true.

The implementation of Truth Table algorithm in the program can be expressed by the following pseudocode:

**local variable:** `_numofmodels`, an int with default value 0

**function** `ENTAILS()`: **returns** true or false

**return** `CHECKALL(KB, KB.Query, KB.symbols, {})`      //new empty model set

**function** `CHECKALL(KB, KB.Query, symbols, model)` **returns** true or false

**if** (`EMPTY(symbols)`) **then**

**if** `PL_TRUE(KB, model)` **then**

**if** `PL_TRUE(KB.Query, model)` **then**

`_numofmodels++`

**return** true;      //entails

else **return** false;

**else** **return** true;      //when KB is false, always returns true

**else do**

`first` <- `FIRST(symbols)`      //extract the first symbols from the input symbols list

`rest` <- `REST(symbols)`      //get the remaining symbols (with first excluded)

`modelT` <- `model.EXTEND(first, true)`      //extend the model with the true value

`modelF` <- `model.EXTEND(first, false)`      //extend the model with the false value

**return** `CHECKALL(KB, KB.Query, rest, modelT) && CHECKALL(KB, KB.Query, rest, modelF)`

### Forward Chaining

The Forward Chaining algorithm starts from the propositional symbols which are already given true in the knowledge base. Then it examines each clause in the knowledge base, if the premises of any clause is satisfied, its conclusion will be added to the knowledge base. The algorithm continues until the query is found or there is no new conclusion being added to the knowledge base.

The implementation of Forward Chaining in the program is expressed in the following pseudocode:

```
function Entails ()
    local variable: _symbols, a queue, contains already true or newly inferred symbols
                   _solution, a list, contains solution path
                   _entails, a bool, initially false
    string symbol;
    while (_symbols is not empty) do
        symbol  $\leftarrow$  DEQUEUE(_symbols)
        _solution  $\leftarrow$  ADD(symbol)
        for each clause c in KB do
            if (c.Count > 0 && c.getPremises().Contains(symbol)) then
                decrement c.Count
                if (c.Count = 0) then
                    if (c.getConclusion() = query) then
                        _solution  $\leftarrow$  ADD(c.getConclusion())
                        _entails  $\leftarrow$  true
                        break;
                    _symbols  $\leftarrow$  ENQUEUE(c.getConclusion())
```

### Backward Chaining

The idea of Backward Chaining algorithm is to work backwards from the query, which is opposite the idea of Forward Chaining algorithm. To prove the query, the algorithm checks if the query is already known, or prove the sub-goals, which are the premises of any clause concluding the query.

The implementation of Backward Chaining in the program is expressed in the following pseudocode:

```
function Entails ()
    local variable: _entails, a bool, initially true
    _entails  $\leftarrow$  TruthValue(query)
function TruthValue(symbol) returns true or false
```

**local variable:** *\_closed\_subgoals*, a queue, contains past sub goals  
                  *\_facts*, a list, contains already true symbols given in KB  
                  *truthvalue*, a bool, initially false

```
_closed_subgoals ← ENQUEUE(symbol)
if (_facts.Contains(symbol)) then
    return true;
List sub_goal_sentences;
for each clause c in KB do
    if (c.getConclusion() = symbol) then
        sub_goal_sentences ← ADD(c)
for each clause c in sub_goal_sentences do
    for each string sym in c.getPremises() do
        if (_closed_subgoals.Contains(sym)) then
            break;
        else then
            truthvalue ← Truthvalue(sym)
if truthvalue
    _facts ← ADD(symbol)
return truthvalue;
```

## Test Cases

Other than the provided test cases, we tried to develop special test cases that might cause the program to be bugged or print incorrect outputs. Overall, the program runs and gives correct outputs with all test cases developed.

### Test 1 (provided):

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

- Output for FC:

```
Engine> FC test_1
YES: a b p2 p3 p1 d
```

- Output for BC:

```
Engine> BC test_1
YES: p2 p3 p1 d
```

- Output for TT:

```
Engine> TT test_1
YES: 3
```

### Test 2 (provided):

```
TELL
(a <=> (c => ~d)) & b & (b => a); c; ~f || g;
ASK
d
```

Output for TT:

```
Engine> TT test_2
NO
```

### Test 3 (provided):

```
TELL
(a <=> (c => ~d)) & b & (b => a); c; ~f || g;
ASK
~d & (~g => ~f)
```




Output for TT:

```
Engine> TT test_3
YES: 3
```

### Test 4: Cycle knowledge base with no solution

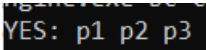
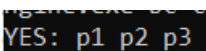
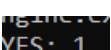
```
TELL
p1=>p2;p2=>p3;p3=>p4;p4=>p1;
ASK
p4
```

If not implemented correctly, the Backward Chaining algorithm may print “YES:” for this test case.

- Output for FC: 
- Output for BC: 
- Output for TT: 

#### Test 5: Cycle knowledge base with solution

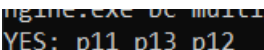
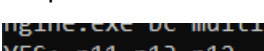
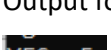
```
TELL
p1=>p2; p2=>p3; p3=>p1;p1;
ASK
p3
```

- Output for FC: 
- Output for BC: 
- Output for TT: 

#### Test 6: Knowledge Base contains multiple character symbols

```
TELL
p13 =>p12; p11=>p13; p11; g252 => f109; u230 => f109;
ASK
p12
```

The reading file method might not be able to handle multiple character symbols if being hard coded.

- Output for FC: 
- Output for BC: 
- Output for TT: 



**Test 7: Query does not exist in KB without solution**

```
TELL
a => b; a => c; d; e; e=>b;
ASK
f
```

- Output for FC:

NO

- Output for BC:

NO

- Output for TT:

NO

**Test 8: Clause has duplicated symbols in the premises**

```
TELL
a&a => b; b&b => c; c=>d;a;
ASK
d
```

If not implemented correctly, the Backward Chaining and Forward Chaining algorithms may give incorrect outputs for this test case.

- Output for FC:

YES: a b c d

- Output for BC:

YES: a b c d

- Output for TT:

YES: 1

**Test 9: Multiple negations (even)**

```
TELL
~~p1; p1&p2 => p3; p2;
ASK
p3
```

Output for TT:

YES: 1

**Test 10: Multiple negations (odd)**

```
TELL
~~~~~p1; p1&p2 => p3; p2;
ASK
p3
```

Output for TT:

```
NO
```

**Test 11: Different solutions**

```
TELL
p=>q; l&m=>p; b&l=>m; a&p=>l; a&b=>l; a; b;p;
ASK
q
```

- Output for FC:

```
engine.exe FC mode
YES: a b p q l m
```

- Output for BC:

```
engine.exe
YES: p q
```

- Output for TT:

```
engine.exe
YES: 1
```

**Test 12: General knowledge with complex query**

```
TELL
~p5;b5<=>p9||p12;b12<=>p5||p35||p37;~b5;b12
ASK
~p9 => b12
```

Output for TT:

```
YES: 3
```

## Research Initiative

### General Knowledge handling

In order to handle general knowledge bases, an Infix to Postfix parser have been implemented, along with a corresponding evaluator for the parsed sentence.

To implement this, each connective is assigned with a precedence value:

- ~ - negation: 4
- & - conjunction: 3
- || - disjunction: 3
- => - implication: 2
- <=> - biconditional: 2

**The Infix to Postfix parser was implemented using the following rules, using the stack data structure:**

- The input for the parser is a sentence in string format.
- The input string will be scanned from left to right.
- If a symbol is scanned, append it to the output result variable.
- Else, if the scanned character (or combination of characters) is a connective or braces:
  - If the stack is empty or the top of the stack is the open brace, or the precedence (priority value) of the connective is greater than that on top of the stack, then push the scanned characters/combination of characters that represents a connective to the stack.
  - Else, pop all the operators from the stack with the priority greater than or equal to the scanned connective or until the open parenthesis is met, or the stack is empty. Then, push the scanned connective to the stack.
- If the scanned character is "(", then push it to the stack.
- If the scanned character is ")", then pop all the operators to the result string until the open parenthesis "(" is encountered. Then, discard both open and close parentheses.
- This process should be repeated until all characters are scanned. Then, pop all the remaining connectives until the stack is empty and add those to the result string.

**The Postfix evaluation function was implemented using the following rules, using the stack data structure:**

- The parsed string can be obtained using the parser above. Then, the string will be scanned from left to right.
- If a symbol is scanned, it will then be pushed to the stack in the form of its bool value (which can be obtained using the model).
- Else, it should be a connective:
  - If the connective is Negation, only the value on top of the stack will be popped, and then negated.
  - Else, two values at the top of the stack (operand A - first and operand - second) will be popped and evaluate with the current connective (in the order of operandB – connective – operandA).
- The process should be repeated until the input string is scanned completely.
- The result will be on top of the stack.

By using the parser and the evaluating function, sentences will be evaluated with proper priority.

For example, statement  $(a \wedge b) \Rightarrow f$ , with a: True, b: True, f:False will be handled in the following process:

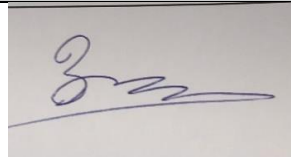

1. Convert to postfix:  $ab \wedge f \Rightarrow$  (eliminate the parentheses, while still maintaining the priority)
2. Push the value of a – True onto the stack
3. Push the value of b – True onto the stack
4. Connective  $\wedge$  is encountered, b-True and a-True values are popped respectively from the stack.
5. Evaluate the statement  $a \wedge b$  – True
6. Push True onto the stack.
7. Push the value of f – False onto the stack.
8. Connective  $\Rightarrow$  is encountered, f-False and  $(a \wedge b)$  – True values are popped respectively from the stack
9. Evaluate the statement  $(a \wedge b) \Rightarrow f$  - False
10. Push False on top of the stack
11. The postfix input is scanned completely, the result of the process is now on top of the Stack – False

## Team Summary Report

In order to work with this program, our team has divided the task into two parts:

- Truth Table Algorithm and General Knowledge Handling
- Forward Chaining and Backward Chaining

Each of us will be responsible for a section while designing and improving the program together. Our progress is tracked through communication and a common GitHub repository, which allow both to make modifications to the common code and be informed when a new modification has been made. This has allowed us to peer review our code versions and give feedback, therefore making updates for our program's performance, or resolving any code conflicts in the program.

Name (Student No.)	Contribution	Signature
Kim Duong Pham (103792850)	Program designing  Truth table and General Knowledge handling  Report writing (Truth Table algorithm, Research Initiative, Acknowledgements/Resources)  Develop test cases and debug the program  <b>Percentage of Contribution: 50%</b>	
Nghia Hieu Pham (103533868)	Program designing  Forward Chaining and Backward Chaining handling  Report writing (Forward Chaining, Backward Chaining, Test Cases)  Develop test cases and debug the program  <b>Percentage of Contribution: 50%</b>	

## Acknowledgements/Resources

1. Bhagav Khalasi (2020), *Expression Parsing: Part 1*, Codesdope, viewed 25 May2023, <<https://www.codesdope.com/blog/article/expression-parsing/>>
  - This website helps us to gain ideas about the infix to postfix parsing implementation.
2. Bhagav Khalasi (2020), *Expression Parsing: Part 2*, Codesdope, viewed 25 May2023, <<https://www.codesdope.com/blog/article/expression-parsing-part-2/>>
  - This website helps us to gain ideas about the infix to postfix parsing implementation.
3. Lecture slides and unit materials
  - The lecture slides and unit materials help us understand the overall concepts of each algorithm.