

Swinburne University of Technology
School of Science, Computing and Engineering Technologies
ASSIGNMENT AND PROJECT COVER SHEET

Subject Code: COS30019

Unit Title: Introduction to Artificial Intelligence

Assignment: 2 - Inference Engine for Propositional Logic

Due date: 2th August 2024

Tutorial Day and Time: Wednesday 13:00PM

Project Group: 2

Lecturer: Pham Thi Kim Dung

To be completed as this is a group assignment:

We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
<u>103803891</u>	<u>Tran Quoc Dung</u>	<u>Dung</u>
<u>104219253</u>	<u>Phan Vinh Khang</u>	<u>Khang</u>

Marker's comments:

Total Mark: _____

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener: _____

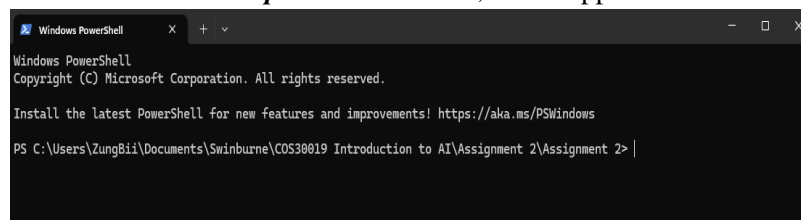
Table of Contents

1. Instruction.....	3
2. Introduction	4
3. Student Details (2 members)	4
4. Features & Implementation	4
<input type="checkbox"/> Truth Table	4
<input type="checkbox"/> Forward Chaining.....	5
<input type="checkbox"/> Backward Chaining	6
5. Test Cases.....	7
<input type="checkbox"/> Horn Form Cases:.....	7
<input type="checkbox"/> Generic Cases	8
6. Acknowledgement & Resources.....	8
7. Research Initiatives	9
<input type="checkbox"/> General knowledge base parser	9
<input type="checkbox"/> Theorem Prover using Resolution	10
8. Team Summary Report	11
9. Conclusion.....	11
10. References.....	11

1. Instruction

- Step 1: First and foremost, download the zip file named “**Assignment 2**”, then **unzip/extract** it to any place on your PC.
- Step 2: Our team’s project code is developed & implemented by **Visual Studio**. Therefore, to open and run the project, Visual Studio is required. If you have not downloaded Visual Studio, access this link: <https://visualstudio.microsoft.com/vs/> to download Visual Studio.
- Step 3: After downloading Visual Studio, select “**Open a project or solution**”, which will open a project through the **.sln** file. Then, select the file “**Assignment2.sln**” file in the extracted folder “**Assignment 2**” to open the project code.
- Step 4: To see the project’s output, you must access the **Terminal** of the Project.

On the one hand, you can access the Local Terminal of the project. By clicking on the recently extracted folder’s path at the header then type “**cmd**” and press **Enter**, or **right click** on the extracted folder then select “**Open in Terminal**”, it will appear:



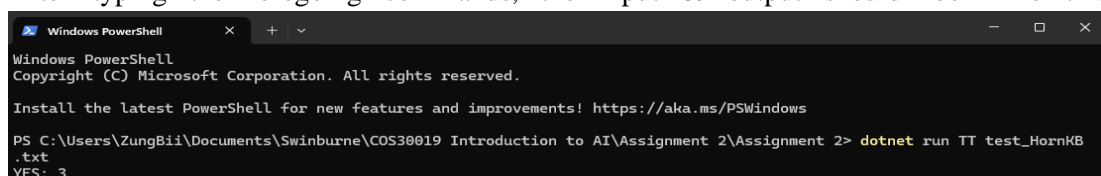
Local Terminal

On the other hand, you can access the Terminal through Visual Studio. After opening the project code on Visual Studio, select **View -> Terminal** at the header. Ensure the accuracy of the project local path on the Terminal, which should look like this:



Visual Studio’s Terminal

- Step 5: Type “**dotnet build**” and press **Enter**. This command helps build the project, along with its dependencies into a binary set, allowing us to run the project smoothly.
- Step 6: Select a technique for the project to implement: Truth Table (**TT**), Forward Chaining (**FC**), Backward Chaining (**BC**) or Resolution Based Theorem Prover (**RB**). Then type “**dotnet run (method’s short name) (filename)**”, while filename represents the file containing inputs. After typing the foregoing commands, the input & output should look like this:



2. Introduction

This project aims to implement a propositional logic inference engine using Truth Table, Forward Chaining, and Backward Chaining techniques. To implement the foregoing approaches, C# is chosen as our programming language by means of its stable type and adaptability. Using it, we can write software that ascertains whether a query within the Horn-form Knowledge Base can be logically entailed. The report covers the conception, design, testing, difficulties, and solutions for the project. Using a lot of testing, we evaluate the engine's capabilities and potential. This project demonstrates the application of artificial intelligence (AI) and logical thinking in real-world scenarios, highlighting C#'s capacity to build intricate logical algorithms and furthering the field of AI research.

3. Student Details (2 members)

- **Student 1:**
 - Name: Tran Quoc Dung
 - Email: 103803891@student.swin.edu.au
 - Group: 2
- **Student 2:**
 - Name: Phan Vinh Khang
 - Email: 104219253@student.swin.edu.au
 - Group: 2

4. Features & Implementation

- **Truth Table**

```
using System;
public class TruthTable : Method
{
    bool[][] table;
    int kb_model = 0;
    bool entail = true;
    // constructor
    public TruthTable(Knowledge_Base kb) : base(kb) { }
    // override
    public override void Run(string query)
    {
        List<string> literal = new List<string>();
        foreach (proposition proposition in Kbase.Propositions) // get a list of all literals that are not part of KB
        {
            if (proposition is Literal && !Kbase.KB.Contains(proposition.Name))
            {
                literal.Add(proposition.Name);
            }
        }
        table = new bool[Convert.ToInt32(Math.Pow(2, literal.Count))][1]; // create the table array with length equal the square of the number of literal
        for (int i = 0; i < (1 << literal.Count); i++) // loop through the table
        {
            Kbase.Facts.Clear(); // The literals in KB dont need to be assigned truth value and are assumed facts
            if (Kbase.Get(given) is Literal)
            {
                Kbase.AddFact(given);
            }
            table[i] = new bool[Kbase.Propositions.Count + 1];
            for (int j = 0; j < literal.Count; j++)
            {
                if ((i & (1 << j)) != 0) // assign truth value to literals using binary counter method
                {
                    Kbase.AddFact(literal[j]);
                }
            }
            for (int k = 0; k < Kbase.Propositions.Count; k++)
            {
                table[i][k] = Kbase.Propositions[k].IsTrue(); // populate the table with truth value of all propositions
            }
            bool kb = true;
            foreach (string condition in Kbase.KB) // check if the KB is true given the current truth assignment
            {
                if (!Kbase.Get(condition).IsTrue())
                {
                    kb = false;
                }
            }
            table[i][table[i].Length - 1] = kb;
            if (kb)
            {
                kb_model++;
                if (!Kbase.Get(Kbase.Goal).IsTrue()) // if sentence is false while kb is true then kb does not entail the sentence
                {
                    entail = false;
                }
            }
        }
    }
}
```

Truth Table C# implementation

A truth table is a mathematical tool utilised in logic to determine the results of an expression based on the Boolean values of its variables. Inference engines employ these tables to deduce new information from established facts and rules.

The function begins by getting a list of literals, which are considered always true if they are part of the knowledge base (KB). It then sets up a 2-dimensional Boolean array, with dimensions based on the square of the number of literals. The function iterates through each row of the table, assigning truth values to the literals similarly to a binary counter, ensuring that all possible combinations are represented. Literals that are assigned a true value are added to the facts list. The function then uses another loop to evaluate all propositions, using the overridden *IsTrue()* function from the children's proposition classes to populate the table. Once the truth values for all propositions in a row are determined, the function assesses whether the KB is true or false for that row. This is done by checking all propositions in the KB; if any are false, the KB is concluded to be false for that model.

- **Forward Chaining**

```
2 references
public class ForwardChaining : Method
{
    1 reference
    public ForwardChaining(Knowledge_Base kb) : base(kb) { }
    2 references
    public override void Run(string query)
    {
        int factcount = -1;
        while (factcount < Kbase.Facts.Count) // stop if no new fact is derived in the previous iteration
        {
            factcount = Kbase.Facts.Count;
            foreach (string proposition in Kbase.KB) // run Infer for all propositions in the KB
            {
                Kbase.Get(proposition).Infer();
                if (Kbase.Get(proposition) is Literal) // add literal to facts if they are part of the KB
                {
                    Kbase.AddFact(proposition);
                }
                if (Kbase.IsFact(query)) // stop if the goal is proven
                {
                    return;
                }
            }
        }
    }
}
```

Forward Chaining C# implementation

Forward chaining, also known as forward deduction or forward reasoning in inference engines, starts with established facts. The algorithm triggers all applicable rules whose conditions are satisfied and adds their conclusions to the set of known facts. This process repeats until the problem is fully resolved. Our forward chaining implementation is straightforward. We utilise proposition classes such as conjunction, implication, biconditional, and literal, which all have an inferencing method. The function uses a nested loop structure. The inner loop goes through all propositions provided as facts in the knowledge base (KB), calls the “Infer function” for each proposition, and checks if the sentence has already been proven, returning if it has. In each iteration of the outer loop, we set a fact count variable to the number of elements in the KB's fact list and then run the inner loop. The outer loop continues if the fact count is less than the total number of facts in the KB, indicating that new facts are being derived with each iteration.

- **Backward Chaining**

```
1 reference
public class BackwardChaining : Method
{
    List<string> visited = new List<string>();
    0 references
    public BackwardChaining(Knowledge_Base kb) : base(kb) { }
    4 references
    public override void Run(string query)
    {
        if (Kbase.Get(query) != null && !visited.Contains(query)) // check if proposition exist
        {
            visited.Add(query);
            foreach (string kb in Kbase.KB)
            {
                Proposition proposition = Kbase.Get(kb);
                if (proposition.SubPropositions.Count == 2 && proposition is Implication) // get all implication in KB
                {
                    if (proposition.SubPropositions[1] == query) // find implication where the current proposition is the consequent
                    {
                        Run(proposition.SubPropositions[0]); // recursively try to prove the antecedent
                        proposition.Infer();
                    }
                }
            }
            foreach (string subproposition in Kbase.Get(query).SubPropositions)
            {
                Run(subproposition); // recursively try to prove its subpropositions
            }
            if(Kbase.InKB(query)) // add literal to facts if they are part of the KB
            {
                Kbase.AddFact(query);
            }
        }
    }
}
```

Backward Chaining C# Implementation

Backward chaining, also known as backward deduction or backward reasoning in inference engines, starts with the end goal or final decision. The system works backwards from this goal to determine which facts must be established to reach it.

Backward chaining is carried out using recursion. The process begins by checking if a proposition exists and hasn't been visited yet. The function then examines all facts in the knowledge base to find those that are implications with the current query as their conclusion. This step helps in identifying the antecedents needed to prove the query. For each antecedent found, the function makes a recursive call with that antecedent. If the antecedent can be proven, the function then uses the “Infer method” on the implication where the query is the conclusion. If the antecedent is successfully proven, the query itself is confirmed as a fact. The function also makes recursive calls with the sub-propositions of the current query. If the query is a literal, it checks if it exists in the knowledge base and adds it to the facts if it does.

5. Test Cases

- **Horn Form Cases:**

- Input:

TELL

$p_2 \Rightarrow p_3$; $p_3 \Rightarrow p_1$; $c \Rightarrow e$; $b \& e \Rightarrow f$; $f \& g \Rightarrow h$; $p_1 \Rightarrow d$; $p_1 \& p_3 \Rightarrow c$; a ; b ; p_2 ;

ASK

d

- Output (Truth Table; Forward Chaining; Backward Chaining):

```
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run TT test_HornKB.txt
YES: 3
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run FC test_HornKB.txt
YES: a, b, p2, p3, p1, d,
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run BC test_HornKB.txt
YES: p2, p3, p1, d,
```

- Input:

TELL

$a \Rightarrow b$; $b \Rightarrow c$; $c \Rightarrow d$;

ASK

d

- Output (Truth Table; Forward Chaining; Backward Chaining):

```
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run TT test_HornKB.txt
NO
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run FC test_HornKB.txt
NO
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run BC test_HornKB.txt
NO
```

- Input:

TELL

$a \Rightarrow b$; $b \Rightarrow c$; $c \Rightarrow d$; $d \Rightarrow e$; f ;

ASK

f

- Output (Truth Table; Forward Chaining; Backward Chaining):

```
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run TT test_HornKB.txt
YES: 6
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run FC test_HornKB.txt
YES: f,
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run BC test_HornKB.txt
YES: f,
```

- **Generic Cases**

- Input:
TELL
 $(a \Leftrightarrow (c \Rightarrow \sim d)) \ \& \ b \ \& \ (b \Rightarrow a); \ c; \ \sim f \parallel g;$
ASK
d
- Output (Truth Table; Relation-Based Theorem Prover):

```
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run TT test_HornKB.txt
NO
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run RB test_HornKB.txt
NO
```

- Input:
TELL
 $(a \Leftrightarrow (c \Rightarrow \sim d)) \ \& \ b \ \& \ (b \Rightarrow a); \ c; \ \sim f \parallel g;$
ASK
 $\sim d \ \& \ (\sim g \Rightarrow \sim f)$
- Output (Truth Table; Relation-Based Theorem Prover):

```
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run TT test_HornKB.txt
YES: 3
PS C:\Users\ZungBii\Documents\Swinburne\COS30019 Introduction to AI\Assignment 2\Assignment 2> dotnet run RB test_HornKB.txt
YES
```

6. Acknowledgement & Resources

- Truth Table Generator: <https://truth-table.com/#>

We used this website to examine the truth table technique and observe the outputs, thereby evaluating the project's accuracy.

- CNF Calculator: <https://boolean-simplifier.com/truth-table-calculator-cnf-dnf>

The website allows us to verify the accuracy of our implementation and learn how to attain conjunctive normal form by translating the complex propositions, thereby developing the resolution base theorem prover.

7. Research Initiatives

- General Knowledge Base Parser

```
proposition = proposition.Trim(); //preprocess the input string
if (proposition.Count(c => c == '(') > proposition.Count(c => c == ')')) // remove invalid brackets
{
    proposition = proposition.Substring(1);
}
else if (proposition.Count(c => c == '(') < proposition.Count(c => c == ')'))
{
    proposition = proposition.Substring(0, proposition.Length - 1);
}
while(BreakParentheses(proposition, true) == "[") // remove unnecessary brackets
{
    proposition = proposition.Substring(1, proposition.Length - 2);
}
```

Pre-processing Procedure of String

The procedure starts by removing whitespace from the input string and eliminating any invalid or redundant brackets. The break-parentheses function then removes all content within the brackets.

```
else if (breakparentheses.Split("<=>").Length == 2) // proposition is a bicondition
{
    propositions.Add(new Biconditions(proposition, this));
    separator = "<=>";
}
else if (breakparentheses.Split("=>").Length == 2) // proposition is an implication
{
    propositions.Add(new Implication(proposition, this));
    separator = "=>";
}
```

Proposition-Type Checker

We use the break-parentheses function to isolate the parts of the string not enclosed by brackets. These segments are checked for separators such as “~, <=>, =>, //, &”. The appropriate preposition class is then initialised, and the separator is assigned. By focusing on the sections outside the brackets, we can identify the outermost separator first, ensuring the proposition is correctly split into sub-propositions.

```
if(separator != "")
{
    string p = proposition;
    foreach (string s in breakparentheses.Split("[") ) // replace the separator that are outside of brackets with *
    {
        if (s != "")
        {
            p = p.Replace(s, s.Replace(separator, "*"));
        }
    }
    subpropositions = p.Split("*"); // split the proposition into sub propositions
}
```

Attaining Sub-Propositions

We split the string using “[” since the break-parentheses function replaces all content inside brackets with “[”. The separator outside the brackets is replaced with *, and then the string is divided into sub-propositions. This approach ensures that separators within brackets do not interfere with the splitting process.

```
foreach(string subproposition in subpropositions) // repeat the process for subpropositions
{
    Get(proposition).SubPropositions.Add(Parse(subproposition)); // recursion
}
```

Sub-Propositions Recursion

We use recursion to call the function for each sub-proposition of the current proposition through a loop. Each sub-proposition is treated in the same way as its parent, and this process continues until the propositions are simplified to literals.

- **Theorem Prover using Resolution**

The initial phase of a resolution-based theorem prover involves converting all propositions into Conjunctive Normal Form (CNF). This task is managed by the “ToCNF function”, which transforms a general proposition into a set of clauses in CNF.

We also developed a resolution rule function to generate new facts and find contradictions. The function starts by checking if the two input propositions are negations of each other; if they are, it detects a contradiction. If they are not, the function gathers the literals from each disjunction and uses a loop to compare every literal from one disjunction with all literals from the other. For each matching pair, the literals are removed from the lists and counted as complementary. The remaining literals are then combined into a single list. If only one complementary literal remains, a new fact is derived. The function then creates a new disjunction string with the remaining literals and adds it to the list of facts.

```
2 references
public override void Run(string query)
{
    foreach (string given in Kbase.KB) // convert the KB to conjunctive normal form
    {
        ToCNF(given);
    }
    ToCNF(Kbase.Parse("~(" + Kbase.Goal + ")")); // convert the negation of the sentence to conjunctive normal form
    Kbase.Facts.Clear();
    foreach (string disjunction in clauses) // add the CNF propositions to facts
    {
        Kbase.AddFact(disjunction);
    }
    clauses.Clear();
    int factcount = -1;
    while (factcount < Kbase.Facts.Count) // limit to 50 iterations
    {
        factcount = Kbase.Facts.Count;
        for (int i = 0; i < factcount; i++)
        {
            for (int j = i + 1; j < factcount; j++)
            {
                if (ApplyRule(Kbase.Facts[i], Kbase.Facts[j])) // stop if a contradiction is found
                {
                    return;
                }
            }
        }
    }
}
```

Implementation of Resolution-Based Theorem Prover

The resolution-based theorem prover starts by converting all propositions in the knowledge base (KB) into conjunctive normal form (CNF). It also transforms the negation of the target proposition into CNF and adds the resulting disjunction clauses to the facts list. The function uses a looping mechanism similar to Forward Chaining, but with an extra nested loop to apply the resolution rule to every pair of disjunction clauses. This process continues to derive new facts until a contradiction is found or no additional inferences can be made.

8. Team Summary Report

- **Tasks & Contribution:**

- Tran Quoc Dung:
 - +) Developing Forward Chaining & Backward Chaining Implementation, along with Researched Technique.
 - +) Investigated ways to improve the engine's performance, and generated test cases for new researched features.
 - +) Responsible for Part 5,6,7 of the Assignment 2 Report.
- Phan Vinh Khang:
 - +) Improve the Truth Table Implementation for the Assignment 2, plus creating its test cases.
 - +) In charge of the project background, analysis and conclusion.
 - +) Collaborated with Tran Quoc Dung on researching initiatives, along with integrating and improving the project report's clarity.

- **Collaboration & Contribution Percentage:**

Tran Quoc Dung managed Forward & Backward Chaining, while Phan Vinh Khang focused on Truth Table techniques, also assisted in searching references and developing the researched technique. Regular communication ensured progress and a well-organised plan, enhancing our understanding of propositional logic as well as the project's outcome.

- Tran Quoc Dung: 60%

- Phan Vinh Khang: 40%

9. Conclusion

Building a propositional logic inference engine in C# with Truth Tables, Forward Chaining, and Backward Chaining was challenging but educational. Our team's efficient processes for ideation, development, and testing resulted in a solid product that illustrates how well C# can handle complex logical operations. This required examining Horn-form Knowledge Base with a query to determine logical entailment. We conquer numerous challenges, expanding our understanding of and proficiency with artificial intelligence and logical reasoning. We also enhanced our engine by integrating a Theorem Prover that requires propositions to be presented in conjunctive normal form and employs resolution to prove theorems more rapidly. This discovery not only expands the scope of our engine but additionally advances artificial intelligence by opening the way for more developments in inference engines and logical thinking.

10. References

Truth table - overview | ScienceDirect Topics. (no date). [Www.sciencedirect.com.
https://www.sciencedirect.com/topics/computer-science/truth-table](https://www.sciencedirect.com/topics/computer-science/truth-table)

Radke, P. (2023, September 22). Forward vs Backward Chaining in Artificial Intelligence | Built In. BuiltIn.com. <https://builtin.com/artificial-intelligence/forward-chaining-vs-backward-chaining>