COS30019 Assignment 2 Report

Student Name: Duy Anh Vuong

Student ID: 102603197

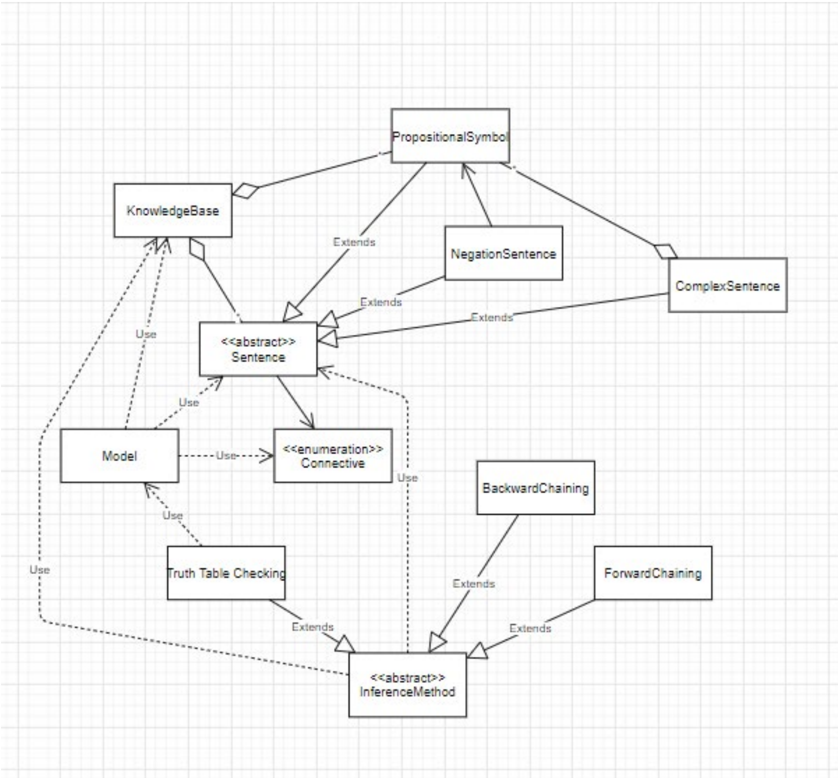
Group Number: COS30019_A02_T008 (Individual Group)

Contents

Features/Bugs/Missing:

Features

For this assignment, I have implemented a propositional inference engine from scratch in C# programming language. The program's structure could be demonstrated in this simplified UML diagram:



The program executes via command line operation, starting with iengine.Bat (the batch file) followed by 2 arguments:

1. file name: the text file of the problem

2. method: the code of the inference method used by the engine to resolve the problem

For example, to solve the problem in "test1.txt" with Truth Table (code: "TT"), the program must be executed via the following command:

```
C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt TT
```

In the program, I implemented a method to parse the problem file as it is passed into the program:

```
static void ConstructProblem(string filename)
{
    try
    {
        //create a reader
        StreamReader reader = new StreamReader(filename);
        if (reader.ReadLine() != "TELL")
        {
            Console.WriteLine("Wrong file format! The file should start with TELL");
            Environment.Exit(1);
        }
        string kb = reader.ReadLine(); //read KB
        string nw = kb.Replace(" ", string.Empty); //get rid of spaces
        string[] stcs = nw.Split(";"); //split sentences by ";"
        kbase = new KnowledgeBase(new List<Sentence>());
        foreach (string str in stcs)
        {
            Sentence stn = ConstructSentence(str, kbase);
            if (stn != null) //if the sentence could be constructed
            {
                kbase.Sentences.Add(stn);
                foreach(PropositionalSymbol sy in stn.GetSymbols())
                {
                    kbase.AddSymbol(sy);
                }
            }
        }
        if (reader.ReadLine() != "ASK")
        {
            Console.WriteLine("Wrong file format! The query should start with ASK");
            Environment.Exit(1);
        }
        string ask = reader.ReadLine(); //read query
        query = ConstructSentence(ask,kbase); //construct query
        if (query == null)
        {
            Console.WriteLine("Cannot form query from file");
            Environment.Exit(1);
        }
        reader.Close();//close the stream reader once finished
    }
```

Parser

The sentences then will be constructed based on their complexity, by 4 methods that call each other and themselves: ConstructSentence, Construct, ConstructComplexSentence and ConstructSentenceWithSimilarConns. They all use the partially constructed Knowledge Base to avoid replications of Propositional Symbol added:

```
6 references
static Sentence ConstructSentence(string stc, KnowledgeBase kb)
{
    Sentence stn;
    //Check which connective is considered first based on their order of precedence
    if (stc == "")
    {
        return null;
    }
    else if (stc.Contains("<=>"))
    {
        stn = Construct(stc, "<=>", Connective.BICONDITIONAL, kb);
    }
    else if (stc.Contains("=>"))
    {
        stn = Construct(stc, "=>", Connective.IMPLICATION, kb);
    }
    else if(stc.Contains("||"))
    {
        stn = Construct(stc, "||", Connective.OR, kb);
    }
    else if (stc.Contains("&"))
    {

        stn = Construct(stc, "&", Connective.AND, kb);

    }
    else if (stc.Contains("~"))
    {
        string[] parts = stc.Split("~");
        foreach (PropositionalSymbol p in kb.GetSymbols()) //if KB has already had that symbol, use that symbol as the negation sentence's PS
        {
            if (p.Symbol == parts[1])
            {
                return new NegationSentence(p);
            }
        }
        stn = new NegationSentence(ConstructSentence(parts[1],kb) as PropositionalSymbol);
    }
    else
    {
        foreach(PropositionalSymbol ps in kb.GetSymbols()) //if KB has already had that symbol, do not create anything new
        {
            if (ps.Symbol == stc)
            {
                return ps;
            }
        }
        stn = new PropositionalSymbol(stc);
        kb.AddSymbol(stn as PropositionalSymbol);
    }
    return stn;
}
2 references
```

ConstructSentence

```
4 references
static Sentence Construct(string stc, string icon, Connective conn,KnowledgeBase kb)
{
    Sentence stn;
    string[] parts = stc.Split(icon);
    if ((parts.Length > 2)) //if there happens to be more than 2 parts
    {
        List<string> mults = new List<string>(parts); //store the parts into the list
        string init = mults[0]; //take the first part out
        mults.Remove(init); //remove it from the list
        stn = ConstructSentenceWithSimilarConns(init, conn, mults, kb);
    }
    else
    {
        stn = ConstructComplexSentence(parts[0], conn, parts[1], kb);
    }
    return stn;
}
6 references
```

Construct

```
2 references
static Sentence ConstructComplexSentence(string Lstc, Connective conn, string Rstc, KnowledgeBase kb)
{
    return new ComplexSentence(ConstructSentence(Lstc, kb), conn, ConstructSentence(Rstc, kb));
}
```

ConstructComplexSentence

```
static Sentence ConstructSentenceWithSimilarConns(string initial, Connective conn, List<string> remains, KnowledgeBase kb)
{
    if (remains.Count == 1) //if there is only one item left in the list
    {
        return ConstructComplexSentence(initial,conn,remains[0],kb);
    }
    //take the first item out of the list, and recursively construct a sentence out of them
    List<string> newL = new List<string>(remains);
    string newI = newL[0];
    newL.Remove(newI);
    return new ComplexSentence(ConstructSentence(initial,kb),conn,ConstructSentenceWithSimilarConns(newI, conn, newL, kb));
}
```

ConstructSentenceWithSimilarConns

After the query and the knowledge base had been set up, I moved on to create the inference methods, each of which will be discussed in the next sections.

Truth Table Checking

The TruthTableChecking class is a child of the abstract class InferenceMethod, which implements the Entails() abstract method of its parent as follows:

```
public override bool Entails(KnowledgeBase kb, Sentence alpha)
{
    List<PropositionalSymbol> symbols = kb.GetSymbols();
    foreach(PropositionalSymbol s in alpha.GetSymbols())
    {
        if (!symbols.Contains(s)) //KB does not contain the symbol?
        {
            symbols.Add(s);
        }

    }
    return TTCheckAll(kb, alpha, symbols, new Model());
```

This implementation takes the idea from the lecture’s given pseudocode. It has a list of propositional symbols from the Knowledge Base and Alpha.

I also checked if the symbol in alpha had already been included in the KB, to avoid reassignment of value in the process.

After the symbols have been collected, the program will begin to check the table with the TTCheckAll() method. It will take 4 arguments: The Knowledge Base, the query (alpha), the symbol list and a Model – whose function will be discussed later.

```
private bool TTCheckAll(KnowledgeBase kb, Sentence alpha, List<PropositionalSymbol> symbols, Model model)
{
    if (symbols.Count == 0) //all symbols have been assigned a value?
    {
        if (model.IsTrue(kb))
        {
            _num_models++;
            return model.IsTrue(alpha); //if model is true to KB, check if model is true to alpha
        }
        else
        {
            //always return true when KB is false
            return true;
        }
    }
    PropositionalSymbol p = symbols[0];
    List<PropositionalSymbol> rest = new List<PropositionalSymbol>(symbols);
    rest.Remove(p);
    return TTCheckAll(kb, alpha, rest, model.Union(p, true))
    && TTCheckAll(kb, alpha, rest, model.Union(p, false)); //recursively assign boolean values to symbols
}
```

TTCheckAll()

```
private Dictionary<PropositionalSymbol, bool> _assignments;
public Model()
{
   _assignments = new Dictionary<PropositionalSymbol, bool>();
}
public Model(Dictionary<PropositionalSymbol,bool> asm)
{
    _assignments = new Dictionary<PropositionalSymbol, bool>(asm);
}
public Dictionary<PropositionalSymbol,bool> Assignment
{
    get
    {
        return _assignments;
    }
}
```

Model class

A model simply consists of a Dictionary with Propositional Symbols as Keys and Boolean Values as Values.

The method would first check if the list of symbols is empty or not. If it is not, then TTCheckAll will be called recursively, with the model now has been assigned the first symbol in the list and a Boolean value as a new Key – Value pair via Union() method; the rest of the symbols will now be the new list.

```
public Model Union(PropositionalSymbol p, bool b)
{
    Model m = new Model(_assignments);
    if(m.Assignment.ContainsKey(p))
    {
        m.Assignment[p] = b;
    }
    else
    {
        m.Assignment.Add(p, b);
    }
    return m;
}
```

Model.Union()

If all the symbols have been assigned a value, the method will then check if the Knowledge Base is true to that model.

```
public bool IsTrue(KnowledgeBase kb)
{
    foreach(Sentence s in kb.Sentences)
    {
        if (!IsTrue(s))
        {
            return false;
        }
    }
    return true;
}
```

PL_IsTrue?(KB)

Each sentence in the Knowledge Base will be checked based on their connectives:

```
2 references
public bool IsTrue(PropositionalSymbol p)
{
    return Assignment[p];
}
12 references
public bool IsTrue(Sentence s)
{

    if (s is PropositionalSymbol)
    {

        return IsTrue((s as PropositionalSymbol));
    }
    else
    {
        //check if the sentence is true or false based on its connective
        Connective conn = s.GetConnective();
        if (conn == Connective.BICONDITIONAL)
        {
            return (!IsTrue((s as ComplexSentence).LSentence) || IsTrue((s as ComplexSentence).RSentence)) && (IsTrue((s as ComplexSentence).LSentence) || !IsTrue((s as ComplexSentence).RSentence));
        }
        if (conn == Connective.IMPLICATION)
        {
            return !IsTrue((s as ComplexSentence).LSentence) || IsTrue((s as ComplexSentence).RSentence);
        }
        else if (conn == Connective.OR)
        {
            return IsTrue((s as ComplexSentence).LSentence) || IsTrue((s as ComplexSentence).RSentence);
        }

        else if (conn == Connective.AND)
        {
            return IsTrue((s as ComplexSentence).LSentence) && IsTrue((s as ComplexSentence).RSentence);
        }
        else if(conn == Connective.NOT)
        {
            return !IsTrue((s as NegationSentence).PropositionalSymbol);
        }
        else
        {
            throw new NotImplementedException();
        }


    }
}
```

Is the model true to Sentence "s"?

Implementation done, I moved on to test the method with the given text file test1.txt:

```
test1.txt
TELL
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2;
ASK
d
```

The result for query "d" is true, "d" is entailed from the KB as it is true in all models (in this case: 3) that the KB is true:

```
C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt TT
Solving test1.txt with Truth Table
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: 3
C:\Users\Duy Anh\Desktop\Assignment2>
```

Forward Chaining

Forward Chaining only works with Horn Form Knowledge Base – Knowledge Base that is formed entirely by Horn Form Clauses.

The method initiates with a count table – a table indicating the number of propositional symbols in each sentence's premise from the Knowledge Base.

```
protected Dictionary<Sentence, int> MakeCountTable(KnowledgeBase kb)
{
    Dictionary<Sentence, int> count = new Dictionary<Sentence, int>();
    foreach (Sentence s in kb.Sentences)
    {
        if (!s.IsInHornForm())
        {
            Console.WriteLine("KB must be formed by Horn clauses!");
            Environment.Exit(1);
        }
        else
        {
            count.Add(s, s.GetSymbols().Count - 1);
        }
    }
    return count;
}
```

Make Count Table

```
1 reference
public bool IsInHornForm()
{
    if(this is PropositionalSymbol)
    {
        return true;
    }
    else if (GetConnective() == Connective.IMPLICATION)//if this is implication sentence
    {
        if (!((this as ComplexSentence).RSentence is PropositionalSymbol)) //right sentence must be a propositional symbol
        {
            return false;
        }
        if ((this as ComplexSentence).LSentence is PropositionalSymbol)//left sentence is either a propositional symbol...
        {
            return true;
        }
        else //...or conjunction of propositional symbols
        {
            List<Connective> Lconn = (this as ComplexSentence).LSentence.GetConnectives();
            foreach (Connective conn in Lconn)
            {
                if (conn != Connective.AND)
                {
                    return false;
                }
            }
            return true;

        }
    }
    else //if this is disjunction of negation sentences and exactly one propositional symbol
    {
        foreach (Connective con in GetConnectives())
        {
            if (con != Connective.OR && con != Connective.NOT)
            {
                return false;
            }
        }

        int count = 0;
        foreach (Sentence simplerS in GetUnarySentences())
        {
            if (!(simplerS is NegationSentence))
            {
                count++;
            }
            if (count > 1) //there could only be one unary sentence that is not a negation sentence
            {
                return false;
            }

        }
        if (count == 0) //if there are all negation sentences, the overall sentence is not in horn form
        {
            return false;
        }
        return true;
    }
}
```

Is the sentence in Horn Form?

Next, create an inferred dictionary, initially false for all symbols in the Knowledge Base:

```
public Dictionary<PropositionalSymbol, bool> MakeInferredTable(KnowledgeBase kb)
{
    Dictionary<PropositionalSymbol, bool> inferred = new Dictionary<PropositionalSymbol, bool>();
    foreach (PropositionalSymbol p in kb.GetSymbols())
    {
        inferred.Add(p, false);
    }
    return inferred;
}
```

MakeInferredTable

After that, create an agenda list, which consists of propositional symbols that are considered “known” in the Knowledge Base:

```
2 references
protected List<PropositionalSymbol> InitAgenda(Dictionary<Sentence,int> count)
{
    List<PropositionalSymbol> agd = new List<PropositionalSymbol>();
    foreach(Sentence s in count.Keys)
    {
        //if that sentence is a propositional symbol, add it to the agenda list
        if(count[s]==0)
        {
            agd.Add(s as PropositionalSymbol);
        }
    }
    return agd;
}
```

InitAgenda

Finally, create another dictionary that has:

Keys: a propositional symbol

Values: Lists of sentences that have their key in the premises.

```
1 reference
private Dictionary<PropositionalSymbol,List<Sentence>> InitializeInPremiseList(Dictionary<PropositionalSymbol,bool> inferred, Dictionary<Sentence,int> count)
{
    Dictionary<PropositionalSymbol, List<Sentence>> pInPremise = new Dictionary<PropositionalSymbol, List<Sentence>>();
    foreach(PropositionalSymbol p in inferred.Keys) //for each propositional symbol in the knowledge base
    {
        List<Sentence> sentenceWithPInPremise = new List<Sentence>();
        foreach(Sentence s in count.Keys) //for each sentence in the knowledge base
        {
            if (count[s] > 0)
            {
                if(s.GetConnective() == Connective.IMPLICATION) //if it is an implication sentence
                {
                    if ((s as ComplexSentence).LSentence.GetSymbols().Contains(p)) //if the left side contains p
                    {
                        sentenceWithPInPremise.Add(s);
                    }
                }
                else //if it is a disjunction of literals
                {
                    foreach(Sentence ss in s.GetUnarySentences())
                    {
                        if(ss is NegationSentence)
                        {
                            if((ss as NegationSentence).PropositionalSymbol == p) //if one of negative literals in this sentence has p as its propositional symbol
                            {
                                sentenceWithPInPremise.Add(s);
                            }
                        }
                    }
                }
            }
        }
        pInPremise.Add(p, sentenceWithPInPremise);
    }
    return pInPremise;
}
```

InitializeInPremiseList

All items needed for the method have been created, the inference could be carried out:

```
4 references
public override bool Entails(KnowledgeBase kb, Sentence alpha)
{
    if(!(alpha is PropositionalSymbol))
    {
        Console.WriteLine("Query for Forward Chaining must be a Propositional Symbol");
        Environment.Exit(1);
    }
    //count table - indicating how many propositional symbols are in a sentence's premise
    Dictionary<Sentence, int> count = MakeCountTable(kb);
    //inferred table, initially false for all symbols
    Dictionary<PropositionalSymbol, bool> inferred = MakeInferredTable(kb);
    //agenda - a list of propositional symbols that are already known in the KB
    List<PropositionalSymbol> agenda = InitAgenda(count);
    //a dictionary with:
    // key: a propositional symbol
    // value: a list of sentences that have that symbol in their premise
    Dictionary<PropositionalSymbol, List<Sentence>> pInPremise = InitializeInPremiseList(inferred, count);

    while (agenda.Count>0) //if the agenda list is not empty
    {
        PropositionalSymbol p = PopAgenda(agenda); //take a propositional symbol out of the agenda
        if(p==alpha) //is alpha? return true
        {
            return true;
        }
        if (!inferred[p]) //if the symbol is not inferred
        {
            inferred[p] = true;
            foreach(Sentence s in pInPremise[p]) //for each sentence that has this symbol in premise
            {
                count[s] -= 1;
                if (count[s] == 0) //if the premise is satisfied
                {
                    //Add the conclusion of this sentence to the agenda
                    if (s.GetConnective() == Connective.IMPLICATION)
                    {
                        agenda.Add((s as ComplexSentence).RSentence as PropositionalSymbol);
                    }
                    else
                    {
                        foreach(Sentence simplS in s.GetUnarySentences())
                        {
                            if(simplS is PropositionalSymbol)
                            {
                                agenda.Add(simplS as PropositionalSymbol);
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

```
C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt FC
Solving test1.txt with Forward Chaining
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: a, b, p2, p3, p1, d,
C:\Users\Duy Anh\Desktop\Assignment2>
```

Result from the terminal

Backward Chaining

This inference Method also used a count and an inferred table, as well as an agenda like in the Forward Chaining method. The difference is that it uses a dictionary that has keys as propositional symbols, and values as lists of sentences that have the keys as their conclusions.

```
private Dictionary<PropositionalSymbol, List<Sentence>> InitializeConclusionList(Dictionary<PropositionalSymbol, bool> inferred, Dictionary<Sentence, int> count)
{
    Dictionary<PropositionalSymbol, List<Sentence>> pAsConclusion = new Dictionary<PropositionalSymbol, List<Sentence>>();
    foreach (PropositionalSymbol p in inferred.Keys) //for each propositional symbol in the inferred list
    {
        List<Sentence> sentenceWithPAsConclusion = new List<Sentence>(); //list of sentences that has "p" as conclusion
        foreach (Sentence s in count.Keys) //for each sentence
        {
            if(count[s]>0) //if the sentence is a complex sentence
            {
                if (s.GetConnective() == Connective.IMPLICATION)
                {
                    if((s as ComplexSentence).RSentence == p)
                    {
                        sentenceWithPAsConclusion.Add(s);
                    }
                }
                else
                {
                    if (s.GetUnarySentences().Contains(p))
                    {
                        sentenceWithPAsConclusion.Add(s);
                    }
                }
                //and that sentence contains propositional symbol "p", add it to the list
            }
        }
        //add "p" and its conclusion list to the dictionary
        pAsConclusion.Add(p, sentenceWithPAsConclusion);

    }
    return pAsConclusion;
}
```

InitializeConclusionList

```csharp
4 references
public override bool Entails(KnowledgeBase kb, Sentence alpha)
{
    if (!(alpha is PropositionalSymbol))
    {
        Console.WriteLine("Query for Backward Chaining must be a Propositional Symbol");
        Environment.Exit(1);
    }
    //count table - number of propositional symbols in a sentence's premise
    Dictionary<Sentence, int> count = MakeCountTable(kb);
    //inferred table, initially false for all symbols
    Dictionary<PropositionalSymbol, bool> inferred = MakeInferredTable(kb);
    //a dictionary with:
    //key: a Propositional Symbol
    //value: a list of sentences that has the key as its conclusion
    Dictionary<PropositionalSymbol, List<Sentence>> sentencesWithPAsConclusion = InitializeConclusionList(inferred, count);
    //agenda: a list of propositional symbols that are already known
    List<PropositionalSymbol> agenda = InitAgenda(count);
    return BC_Entails(alpha as PropositionalSymbol, inferred, sentencesWithPAsConclusion, agenda,count);
}
3 references
private void AddToResult(PropositionalSymbol p)
{
    if (!_chain.Contains(p))
    {
        _chain.Add(p);
    }
}
3 references
private bool BC_Entails(PropositionalSymbol p, Dictionary<PropositionalSymbol, bool> inferred, Dictionary<PropositionalSymbol, List<Sentence>> sentencesWithPAsConclusion, List<PropositionalSymbol> agenda,Dictionary<Sentence, int> count)
{
    if (agenda.Contains(p)) //if P is already known
    {
        AddToResult(p);
        return true;
    }
    if (!inferred[p]) //if p has not been considered yet
    {
        inferred[p] = true;
        foreach(Sentence s in sentencesWithPAsConclusion[p]) //for each sentence with p as conclusion
        {
            if (s.GetConnective() == Connective.IMPLICATION) //if p is in "(conjunction of propositional symbols) => proposional symbol" form
            {
                foreach (PropositionalSymbol ps in (s as ComplexSentence).LSentence.GetSymbols()) //recursively search for a node included in the agenda
                {
                    if (!BC_Entails(ps, inferred, sentencesWithPAsConclusion, agenda,count))
                    {
                        break;
                    }
                    count[s] -= 1;
                }
                if(count[s]==0) //if all the premises of that sentence is satisfied, add the conclusion to the agenda
                {
                    agenda.Add(p);
                    AddToResult(p);
                    return true;
                }
            }
            else //if p is in "disjunction of unary sentences that exactly one of them is positive"
            {

                foreach (Sentence stc in s.GetUnarySentences())
                {
                    if(stc is NegationSentence) //take only the negation sentences into account
                    {
                        if(!BC_Entails((stc as NegationSentence).PropositionalSymbol,inferred,sentencesWithPAsConclusion, agenda,count))
                        {
                            break;
                        }
                        count[s] -= 1;
                    }
                }
                if (count[s] == 0) //if all the premises of that sentence is satisfied, add the conclusion to the agenda
                {
                    agenda.Add(p);
                    AddToResult(p);
                    return true;
                }
            }

        }
    }
    return false;
}
```

Main part of the method

```
C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt BC
Solving test1.txt with Backward Chaining
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: p2, p3, p1, d,
C:\Users\Duy Anh\Desktop\Assignment2>
```

Result from terminal

Research/Extension:

Since my work for this assignment is entirely individual, I am not able to develop extra inference method, however, I have partially extended the Truth Table method so that it could work with all connectives.

One thing lacked is that my program could not form sentences with brackets, yet the program could still calculate the Boolean value of a sentence considering its connectives' order of precedence.

Also, Forward Chaining and Backward Chaining could not only work with Horn Clauses in the form of *(conjunction of propositional symbols) => propositional symbols* but also effective to the clauses in the form of *disjunction of literals in which exactly one is positive.*

Test Cases:

test1.txt – Given Horn Form KB test case

<table>
<tr><td>Truth Table</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt TT
Solving test1.txt with Truth Table
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: 3</td></tr>
<tr><td>Forward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt FC
Solving test1.txt with Forward Chaining
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: a, b, p2, p3, p1, d,</td></tr>
<tr><td>Backward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test1.txt BC
Solving test1.txt with Backward Chaining
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: p2, p3, p1, d,</td></tr>
</table>

test2.txt – Disjunction of Literals Horn Form test case

<table>
<tr><td>Truth Table</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test2.txt TT
Solving test2.txt with Truth Table
TELL
~p2|p3;~p3|p1;~c|e;~b|~e|f;~f|~g|h;~p1|d;~p1|~p3|c;a;b;p2;
ASK
d
RESULT: YES: 3</td></tr>
<tr><td>Forward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test2.txt FC
Solving test2.txt with Forward Chaining
TELL
~p2|p3;~p3|p1;~c|e;~b|~e|f;~f|~g|h;~p1|d;~p1|~p3|c;a;b;p2;
ASK
d
RESULT: YES: a, b, p2, p3, p1, d,</td></tr>
<tr><td>Backward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test2.txt BC
Solving test2.txt with Backward Chaining
TELL
~p2|p3;~p3|p1;~c|e;~b|~e|f;~f|~g|h;~p1|d;~p1|~p3|c;a;b;p2;
ASK
d
RESULT: YES: p2, p3, p1, d,</td></tr>
</table>

test3.txt – Lecture 7's example for Forward Chaining and Backward Chaining

<table>
<tr><td>Truth Table</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test3.txt TT
Solving test3.txt with Truth Table
TELL
p=>q;l&m=>p;b&l=>m;a&p=>l;a&b=>l;a;b;
ASK
q
RESULT: YES: 1</td></tr>
<tr><td>Forward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test3.txt FC
Solving test3.txt with Forward Chaining
TELL
p=>q;l&m=>p;b&l=>m;a&p=>l;a&b=>l;a;b;
ASK
q
RESULT: YES: a, b, l, m, p, q,</td></tr>
<tr><td>Backward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test3.txt BC
Solving test3.txt with Backward Chaining
TELL
p=>q;l&m=>p;b&l=>m;a&p=>l;a&b=>l;a;b;
ASK
q
RESULT: YES: a, b, l, m, p, q,</td></tr>
</table>

test4.txt – Extra Horn Form KB test case

<table>
<tr><td>Truth Table</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test4.txt TT
Solving test4.txt with Truth Table
TELL
p1&p2&p3&p4&p5&p6&p7=>p8;p1&p3&p4&p6&p7&p8=>p9;p6&p7&p4&p5&p9=>p10;p13&p12&p11=>p14;p11&p12&p4&p5&p10=>p13;p11&p1&p4&p5&
p6=>p12;p1&p2&p6&p8&p10=>p11;p1&p2&p6&p8=>p;p1;p2;p3;p4;p5;p6;p7;
ASK
p14
RESULT: YES: 1</td></tr>
<tr><td>Forward Chaining</td><td>C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test4.txt FC
Solving test4.txt with Forward Chaining
TELL
p1&p2&p3&p4&p5&p6&p7=>p8;p1&p3&p4&p6&p7&p8=>p9;p6&p7&p4&p5&p9=>p10;p13&p12&p11=>p14;p11&p12&p4&p5&p10=>p13;p11&p1&p4&p5&
p6=>p12;p1&p2&p6&p8&p10=>p11;p1&p2&p6&p8=>p;p1;p2;p3;p4;p5;p6;p7;
ASK
p14
RESULT: YES: p1, p2, p3, p4, p5, p6, p7, p8, p9, p, p10, p11, p12, p13, p14,</td></tr>
</table>

Backward Chaining	C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test4.txt BC Solving test4.txt with Backward Chaining TELL p1&p2&p3&p4&p5&p6&p7=>p8;p1&p3&p4&p6&p7&p8=>p9;p6&p7&p4&p5&p9=>p10;p13&p12&p11=>p14;p11&p12&p4&p5&p10=>p13;p11&p1&p4&p5& p6=>p12;p1&p2&p6&p8&p10=>p11;p1&p2&p6&p8=>p;p1;p2;p3;p4;p5;p6;p7; ASK p14 RESULT: YES: p1, p2, p6, p3, p4, p5, p7, p8, p9, p10, p11, p12, p13, p14,

test5.txt – Partially Generic KB test case (works only with Truth Table)

Truth Table
C:\Users\Duy Anh\Desktop\Assignment2>iengine.Bat test5.txt TT Solving test5.txt with Truth Table TELL ~p11;b11<=>p12|p21;b21<=>p11|p22|p31;~b11;b21; ASK ~p21 RESULT: YES: 3

Notes

I have compiled the C# source code after the coding process is finished, so that the installation of .NET Core is not necessary for execution, the command line operation is now:

iengine <filename> <method>

```
C:\Users\Duy Anh\Desktop\Assignment2>iengine test1.txt FC
Solving test1.txt with Forward Chaining
TELL
p2=>p3;p3=>p1;c=>e;b&e=>f;f&g=>h;p1=>d;p1&p3=>c;a;b;p2;
ASK
d
RESULT: YES: a, b, p2, p3, p1, d,
```

Acknowledgement/Resources:

- http://aima.cs.berkeley.edu/: The page to the textbook used in the unit. It helps me with understanding the fundamentals of propositional logic and inference methods.
- https://github.com/aimacode: This website provides sample codes for problems mentioned in the textbook. Though its implementations are all in Java and is way too completed for application into a logical agent, it is a good starting point for me to get ideas on how to construct classes.