Assignment 2

Inference Engine for Propositional Logic

Due: 4:30pm Friday 26th May 2023

Contributes: 20% of unit grade

Task Type: Group.

Students: Nathan Hoorbakht (103865794)

Nicholas Gustin (103995882)

Group: COS30019_A02_T005

Table of Contents

Instructions:

To run the program, navigate to the directory where iengine.exe file is located. Then open the command prompt from that directory.

Input:
Type the command: InferenceEngine.exe <search_strategy> <path_to_test_file>

Available search strategies:

- TT
- FC
- BC

Output:

Example output for Truth Table Algorithm:	*YES: 3*
Example output for Forward Chaining:	*YES: a, b, p2, p3, p1, d*
Example output for Backward Chaining:	*YES: p2, p3, p1, d*

Glossary:

Terms (underlined)	**Definition**
Artificial intelligence	A branch of computer science dealing with the simulation of intelligent behaviour in computers
Hidden properties	Aspects of an environment that an agent is (initially) unaware of
Symbolic logic	The use of symbols to denote propositions, terms, and relations in order to assist reasoning.
Propositions	A statement or assertion that expresses a judgement or opinion.
Truth table	A diagram in rows and columns showing how the truth or falsity of a proposition varies with that of its components.
Positive literal	The simplest form of a statement that is accepted in propositional logic
Logical entailment	A set of sentences can logically entail a sentence **if and only** if every assigned truth value that satisfies the first sentence also satisfies the second sentence.
Postfix expression	A representation of an expression where the **operator** is written **after** its operands

Stack	An abstract data type that serves as a collection of elements.
Atomic Propositions	Known facts or basic statements that serve as the foundation of the knowledge base.
Inference rules	Rules or logical operations used to derive new conclusions from existing information in the knowledge base.
Knowledge base	A repository of information that contains the known facts, rules, and relationships used in a reasoning system.
Forward chaining:	A reasoning approach that iteratively applies inference rules to deduce new conclusions based on the available information.
Backward chaining:	A reasoning approach that involves breaking down the conclusion into simpler subgoals and recursively proving each subgoal using inference rules and known facts.

Introduction:

An ideal artificial intelligence will aim to **act rationally** at all times - that is, always choosing the course of action expected to maximise the achievement of a set goal. Previously, this unit has explored intelligent agents which operate on **reflex** - choosing an action only based on its current and previous perceptions of a simple environment.

However, in a complex and unknown environment with potentially hidden properties, we cannot expect our agent to act rationally based on reflex alone. It also needs the ability to **think rationally** - our agent needs to be able to choose the 'correct' course of action based on **irrefutable reasoning**. This approach to artificial intelligence uses **symbolic logic** to represent the laws of rational thought [1]. The manipulation and application of this symbolic logic teach an agent more about the state of its environment (both known and unknown).

Logical (or Knowledge-based) agents combine **general knowledge** with **current percepts** to **infer hidden aspects** of a current state [2]. There are two primary components of a knowledge-based agent:

1. The knowledge base
2. The inference engine

The knowledge base contains **domain-specific content**. It is a database of symbols that represent aspects of the environment an agent operates within. It may initially contain rules about the environment and/or properties of an agent's immediate surroundings.

The aim of an inference engine is to discover (or **infer**) aspects of an agent's environment that are not initially obvious (hidden properties). It is able to access the knowledge base and, with the known content, perform algorithmic reasoning to further analyse the environment. This further analysis is then **added back into the knowledge base**, expanding it for future reasoning. Notably, the algorithms used in an inference engine are **domain-independent**, meaning that, once coded, it can work in many different environments.

The ultimate intention behind this cycle is (generally) to answer a **query** about the environment provided by the agent. Ideally, a series of queries lead to achieving a set goal.

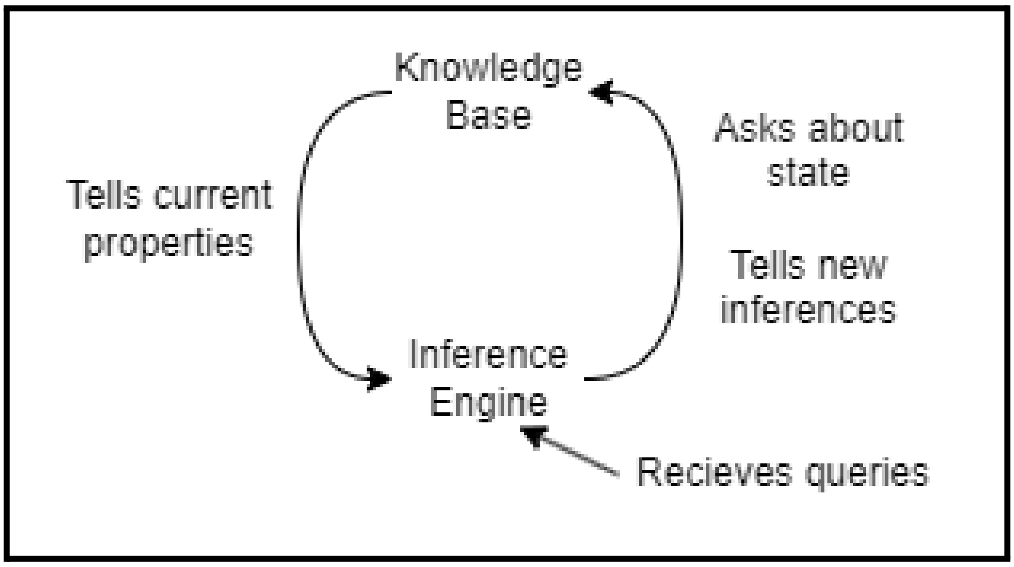


Figure 1 - Simplified diagram of the interaction between the Knowledge Base and Inference Engine

This assignment required students to implement an inference engine for propositional logic in software. **Propositional logic** is a field of logic that focuses on the meaning of and relationships between propositions. It is closely linked to the field of symbolic logic and applies specific **logical operators** (propositional connectives) that describe relationships [3]. The inference engine implemented uses **three types of algorithms**, namely:

1. Truth Table Checking
2. Forward Chaining
3. Backward Chaining

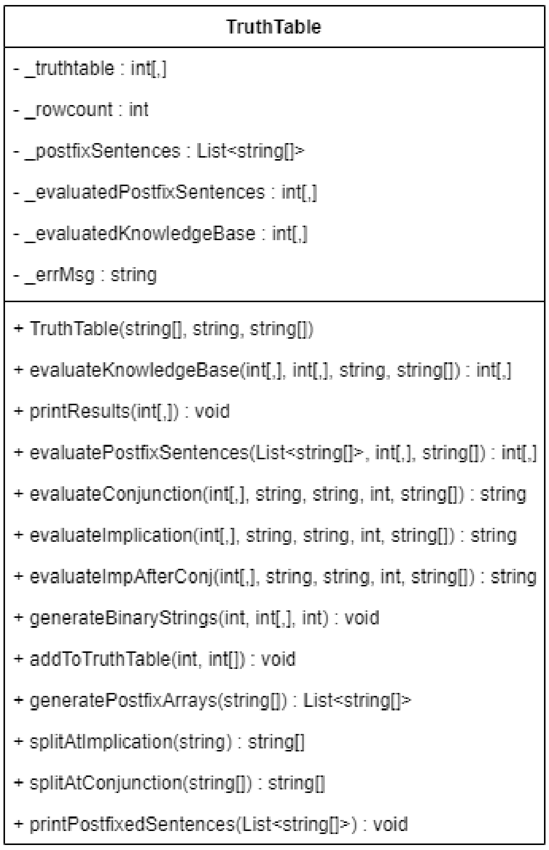
As input, the program will accept a **Horn-Form Knowledge Base** as well as a **query**. A horn-form knowledge base is made up of a series of clauses in **horn normal form**, which is a clause with **at most one positive literal.** The aim of the program is to determine whether the provided query can be logically entailed from the provided knowledge base.

Features and Bugs

Truth Table

This algorithm relies on **inference by enumeration**. It utilises a **binary table**, known as a Truth Table, to generate every possible permutation of true or false for every symbol in the knowledge base. This binary table will have a number of rows equivalent to 2^n, where n = the number of symbols. The engine will then scan through every model and evaluate each sentence.

This implementation makes use of **postfix expressions** and a **stack**. Having successfully evaluated the knowledge base for every possible model, we can then do a simple lookup of whether the query entails the knowledge base.

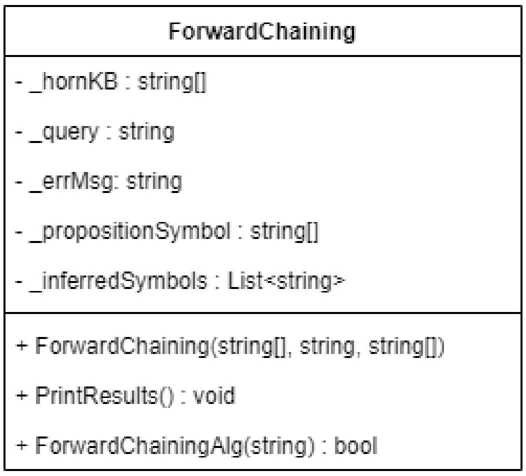


```
TruthTable
- _truthtable : int[,]
- _rowcount : int
- _postfixSentences : List<string[]>
- _evaluatedPostfixSentences : int[,]
- _evaluatedKnowledgeBase : int[,]
- _errMsg : string
+ TruthTable(string[], string, string[])
+ evaluateKnowledgeBase(int[,], int[,], string, string[]) : int[,]
+ printResults(int[,]) : void
+ evaluatePostfixSentences(List<string[]>, int[,], string[]) : int[,]
+ evaluateConjunction(int[,], string, string, int, string[]) : string
+ evaluateImplication(int[,], string, string, int, string[]) : string
+ evaluateImpAfterConj(int[,], string, string, int, string[]) : string
+ generateBinaryStrings(int, int[,], int) : void
+ addToTruthTable(int, int[]) : void
+ generatePostfixArrays(string[]) : List<string[]>
+ splitAtImplication(string) : string[]
+ splitAtConjunction(string[]) : string[]
+ printPostfixedSentences(List<string[]>) : void
```

Below is some pseudocode that covers the basic implementation of this algorithm in the engine. *Note it is not intended to be a one-to-one recreation of every line, rather to provide a general idea of the approach*:

```
Take in a list of parsed symbols & generate binary tree
For each horn clause in the KB:
    Identify implication/conjunction, split at appropriate symbols and place in a postfix-style array
For each postfixed ‘sentence’ created:
    If sentence contains only one character:
        Identify that symbol’s boolean value in this model and assign to a 2D array
    If sentence contains more than one character:
        Use a stack to evaluate conjunctions and implications and assign to a 2D array
For each evaluated model:
    Perform a conjunction on each value to evaluate the entire knowledge base
For each model where KB = true:
    If the query = true, add to the count of logical entailment models
    If the query = false, return false (query does not entail the knowledge base)
```

Forward Chaining

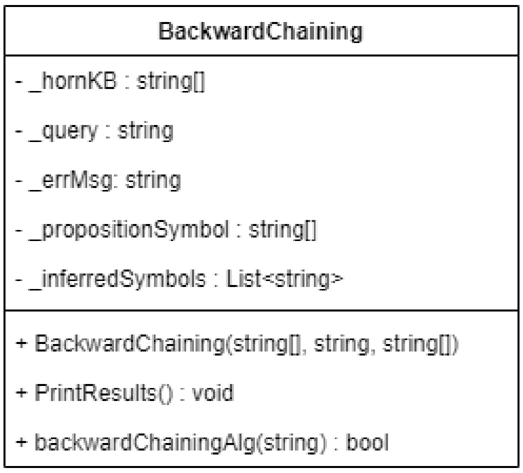

ForwardChaining
- _hornKB : string[]
- _query : string
- _errMsg: string
- _propositionSymbol : string[]
- _inferredSymbols : List<string>
+ ForwardChaining(string[], string, string[])
+ PrintResults() : void
+ ForwardChainingAlg(string) : bool

Forward chaining is a reasoning approach that begins with the initial set of known facts, also known as atomic propositions, stored in the knowledge base. The process involves iteratively applying inference rules to deduce new conclusions based on the available information. This continuous application of inference rules allows for the gradual accumulation of knowledge and the derivation of additional conclusions. The goal of forward chaining is to continue this process until the desired conclusion is reached or until no further new conclusions can be inferred.

Below is some pseudocode that covers the basic implementation of this algorithm in the engine:

```
Initialize the parsed data: HornKB, Query, Proposition Symbol, and Inferred Symbols

For each symbol in PropositionSymbol:
    For each rule in HornKB:
        Identify implication/conjunction and split the rule into premise and conclusion

        If the rule contains only one character:
            Add the premise to Inferred Symbols as its known truth

        If the rule contains more than one character:
            Check if the premise is in the Inferred Symbols array
            If it is, add the conclusion to Inferred Symbols

        If Inferred Symbols contains the query:
            Return true (the goal state can be proven based on the premises)
Return false (the goal state cannot be proven based on the given knowledge base)
```

Backwards Chaining


BackwardChaining
- _hornKB : string[]
- _query : string
- _errMsg: string
- _propositionSymbol : string[]
- _inferredSymbols : List<string>
+ BackwardChaining(string[], string, string[])
+ PrintResults() : void
+ backwardChainingAlg(string) : bool

Backward chaining is a reasoning approach that begins with the desired conclusion and proceeds in a reverse direction, working backwards towards the known facts stored in the knowledge base. The process involves breaking down the desired conclusion into simpler subgoals or subconclusions. Through recursive steps, backward chaining attempts to prove each subgoal by identifying rules that can derive it from either other known facts or subgoals.

In this approach, the focus is on finding a path from the desired conclusion to the known facts by tracing the dependencies between subgoals and the available information. By recursively applying inference rules and establishing the logical connections between subgoals and known facts, backward chaining aims to validate the consistency and validity of the reasoning process.

Below is some pseudocode that covers the basic implementation of this algorithm in the engine:

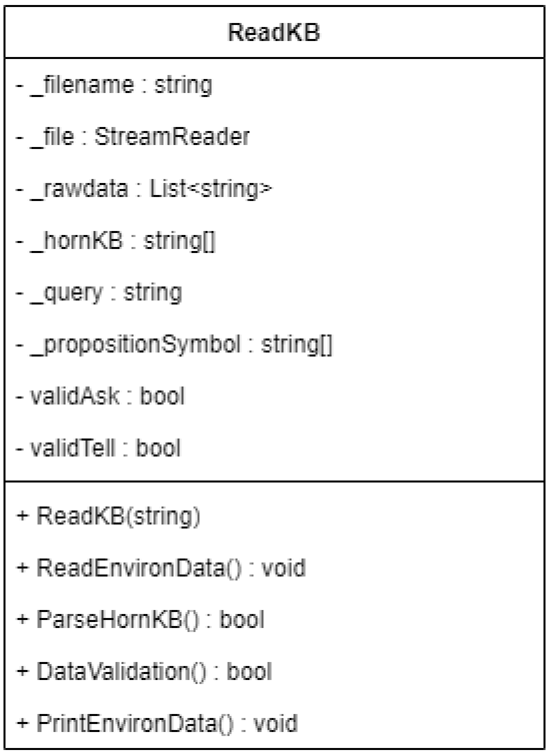
```
Initialize the parsed data: HornKB, Query, Proposition Symbol, and Inferred Symbols
For each rule in HornKB:
       Identify implication/conjunction and split the rule into premise and conclusion
If conclusion is equal to the query:
       Add conclusion to Inferred Symbols
For each premise in the rule:
       Call the backwardChainingAlg method with the premise as the query
If the result is true (premise can be proven):
       Add the premise to Inferred Symbols
       If the result is false (premise cannot be proven):
              Continue to the next rule
       If the query is present in Inferred Symbols:
              Return true (the goal state can be proven based on the premises)
Return false (the goal state cannot be proven based on the given knowledge base)
```

ReadKB

A simple file that is able to read in a Horn-Form KB and parse it out into its individual parts, namely a list of sentences, a list of symbols and the query asked. It also contains some **data validation components**, which ensure that the file fed in is formatted correctly as well as ensuring the file and arguments placed into the program do exist.

Below is some pseudocode that covers the basic implementation of this algorithm in the engine:

```
Initialize the data file
For each line in data file:
       Add to raw data
If raw data line is TELL
       Split to get Horn KB lines
       Split to get Propositional Symbols
If raw data line is ASK
       Get Query
```



ReadKB

- _filename : string
- _file : StreamReader
- _rawdata : List<string>
- _hornKB : string[]
- _query : string
- _propositionSymbol : string[]
- validAsk : bool
- validTell : bool

+ ReadKB(string)
+ ReadEnvironData() : void
+ ParseHornKB() : bool
+ DataValidation() : bool
+ PrintEnvironData() : void

Test Cases

Where a test case resulted in the fixing of a bug, it has been indicated with original/actual output.

Test 1 - Default [4]

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

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 3	YES: 3	✓
FC	YES: a, b, p2, p3, p1, d	YES: a, b, p2, p3, p1, d	✓
BC	YES: p2, p3, p1, d	YES: p2, p3, p1, d	✓

Test 2 - Impossible Solution

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

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	NO	YES: 2	NO	✓
FC	NO	NO	-	✓
BC	NO	ERROR	NO	✓

Bug fixed (Truth Table): the engine assumed a single symbol in a sentence defaulted to true. It now looks up the correct value of that symbol for that model in the truth table.
Bug fixed (Backward Chaining): there was an event handling error that cause the app to crash when conclusion was being assigned. A length check on implication was added such that only when it exceeds 1 the conclusion is defined.

Test 3 - Query is an Atomic Symbol

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

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 3	YES: 3	✓
FC	YES: a	YES: a	✓
BC	YES: a	YES: a	✓

Test 4 - Query not in Inferred Symbols

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

Algorithm	Expected Output	Actual Output	Pass
TT	NO	NO	✓
FC	NO	NO	✓
BC	NO	NO	✓

Test 5 - Premise has an &

TELL f=> g; c&e => f; r&s => u;c;e; **ASK g**

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	YES: 7	YES: 7	-	✓
FC	YES: c, e, f, g	YES: c, e, f, c, e, g	YES: c, e, f, g	✓
BC	YES: c, e, f, g	YES: c, e, f, g	-	✓

Bug fixed (Forwad Chaining): there was no check to see if the propositions a and c were already added to the inferred symbols list. Thus when the code loops its added again, a if check was added to prevent repetition.

Test 6/7 - File Read Integrity Check

6. TEqL p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2; **ASK d**
7. TELL p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2; **ASqK d**

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	Failed File Read! Please Check Your Data File.	ERROR + OUTPUT ATTEMPT	Failed File Read! Please Check Your Data File.	☑
FC	Failed File Read! Please Check Your Data File.	ERROR + OUTPUT ATTEMPT	Failed File Read! Please Check Your Data File.	☑
BC	Failed File Read! Please Check Your Data File.	ERROR + OUTPUT ATTEMPT	Failed File Read! Please Check Your Data File.	☑

Bug fixed (All): Integrity checks were put into place (ReadKB.DataValidation()) to ensure that test cases were entered with the correct spelling. Further integrity checks were added including checking both arguments were passed in and checking the file passed in actually exists.

Test 8 - Impossible Solution

TELL p1 & p3 => a; h => p1; h; **ASK p3**

Algorithm	Expected Output	Actual Output	Pass
TT	NO	NO	☑
FC	NO	NO	☑
BC	NO	NO	☑

Test 9 - Logic that contains an infinite loop

TELL x=>y; y=>z; a&b => c; c&d => e; e&f => g; g&h => i; y=>a; y&z => x; b; d; z; **ASK i**

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	NO	NO	-	☑
FC	NO	NO	-	☑
BC	Due to the recursive nature of the backwards changing, this particular test will cause a stack overflow exception as Y will recursively search using X and X recursively search using Y.			☐

Test 10 - Test for the Parser (symbols = words)

TELL dog=>animal; animal=>cat; animal&living => dog; dog; living; **ASK nature**

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 1	YES: 1	☑
FC	YES: dog, living, animal	YES: dog, living, animal	☑
BC	YES: dog, animal	YES: dog, animal	☑

Test 11 - Standard Test

TELL p2 & a => p3; n => a; n => p2; n; **ASK p3**

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	YES: 1	YES: 1	-	☑
FC	YES: n, a, p2, p3	YES: n, a, p2, p3	-	☑
BC	YES: n, a, p2, p3	YES: n, p2, n, a, p3	YES: n, a, p2, p3	☑

Test 12 - Standard Test

TELL p2 => p4; a & e => p2; a => e; a; **ASK p2**

Algorithm	Expected Output	Original Output	Corrected Output	Pass
TT	YES: 1	YES: 1	-	☑
FC	Yes a, e, p2	YES: a, e, p2	-	☑
BC	Yes a, e, p2	YES: a, a, e, p2	Yes a, e, p2	☑

Test 13 - Standard Test

TELL e => c; f&c => p1; r =>f; e; r; **ASK p1**

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 1	YES: 1	☑
FC	YES: e, r, c, f, p1	YES: e, r, c, f, p1	☑
BC	YES: r, f, e, c, p1	YES: r, f, e, c, p1	☑

Test 14 - Simple Test

TELL p1 => p2; p1; **ASK p2**

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 1	YES: 1	☑
FC	YES: p1, p2	YES: p1, p2	☑
BC	YES: p1, p2	YES: p1, p2	☑

Test 15 - Standard Test

TELL c => r; p2&r => a; p2 & b => c; a; b; p2; **ASK r**

Algorithm	Expected Output	Actual Output	Pass
TT	YES: 1	YES: 1	☑
FC	YES: a, b, p2, c, r	YES: a, b, p2, c, r	☑
BC	YES: p2, b, c, r	YES: p2, b, c, r	☑

Acknowledgements/Resources

1. **Stanford University Truth Table Generator [5]**
   This online tool by Stanford University has been useful in testing our test cases to get an ideal (correct) output. We were then able to test our engine against these outputs and identify/fix bugs

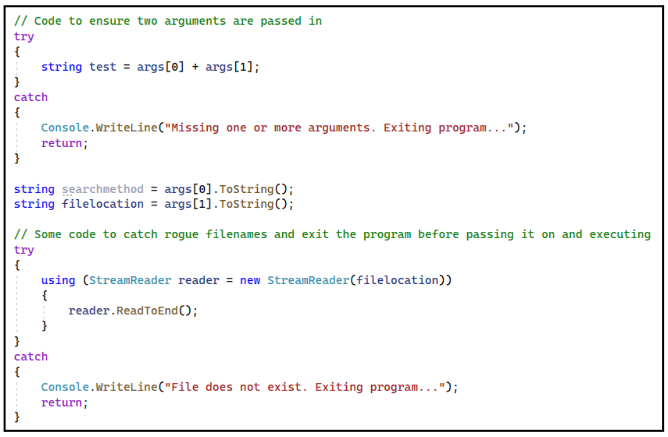
2. **Geeksforgeeks page on generating binary strings [6]**
   This blog post by author *shashipk11* was the backbone of the method generateBinaryStrings() in TruthTable.cs. It provides a very helpful structure and explanation of the recursion and backtracking required to generate all permutations of n bits. This method is used to generate the truth table for all symbols in the knowledge base.

3. **Simon Fraser University, CMPT 125 Summer Course (2012) [7]**
   This page provides a great overview of what postfix expressions are and how to evaluate them. The information was critical to understanding how to split and parse postfix expressions using a stack, the same method implemented in evaluatePostfixSentences() in TruthTable.cs

Research

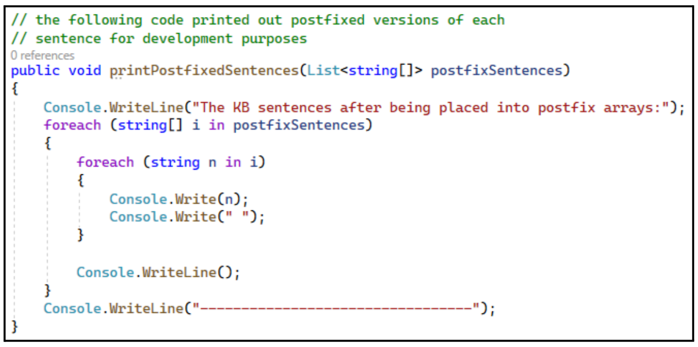
The program implements a fair bit of self-error checking, which was built up over the course of programming. This comes in many forms, including:

Explicit catches for exceptions in the C# code
Multiple different places in the program include *try… catch* code blocks in order to prevent invalid function. 'Invalid function' could include feeding in a non-existent file or not feeding in enough arguments. It also works within the algorithms themselves, catching validly-formatting files which do not make logical sense. Examples of this include where a specified query doesn't exist in the given knowledge base, or preventing an infinite loop.



```
// Code to ensure two arguments are passed in
try
{
    string test = args[0] + args[1];
}
catch
{
    Console.WriteLine("Missing one or more arguments. Exiting program...");
    return;
}

string searchmethod = args[0].ToString();
string filelocation = args[1].ToString();

// Some code to catch rogue filenames and exit the program before passing it on and executing
try
{
    using (StreamReader reader = new StreamReader(filelocation))
    {
        reader.ReadToEnd();
    }
}
catch
{
    Console.WriteLine("File does not exist. Exiting program...");
    return;
}
```

Debugging methods
There also exist throughout the program multiple lines of code and methods which can be used to provide additional information to a user. Examples of this additional information include printing a formatted truth table for n bits, as well as providing state information at specific points throughout the program (i.e. during a conjunction/implication). Most (if not all) references to these methods (such as TruthTable.printPostfixedSentences()) have been removed for the purposes of batch testing.



```
// the following code printed out postfixed versions of each
// sentence for development purposes
0 references
public void printPostfixedSentences(List<string[]> postfixSentences)
{
    Console.WriteLine("The KB sentences after being placed into postfix arrays:");
    foreach (string[] i in postfixSentences)
    {
        foreach (string n in i)
        {
            Console.Write(n);
            Console.Write(" ");
        }

        Console.WriteLine();
    }
    Console.WriteLine("--------------------------------");
}
```

There is also a deprecated feature of all three algorithms - **_errMsg** - which, upon a **'NO'** output, would print further information to explain the failure. These included *"No Solution!", "Invalid Query!", "Infinite Loop!"* and were also removed for the purposes of conforming to batch testing.

Team Summary Report

Team: COS30019_A02_T005
Members: Nicholas Gustin, Nathan Hoorbakht

Nicholas Gustin

Contribution Percentage = 50%
Complete work (Inference Engine):

- Reading in and parsing the knowledge base (ReadKB.cs)
- Implementation of depth-first enumeration of all models (TruthTable.cs)

Complete work (Report):

- Introduction, Glossary, Features and Bugs (Truth Table, ReadKB), Test Cases, Acknowledgements/Resources, Research, Team Summary Report, Bibliography

Nathan Hoorbakht

Contribution Percentage = 50%
Complete work (Inference Engine):

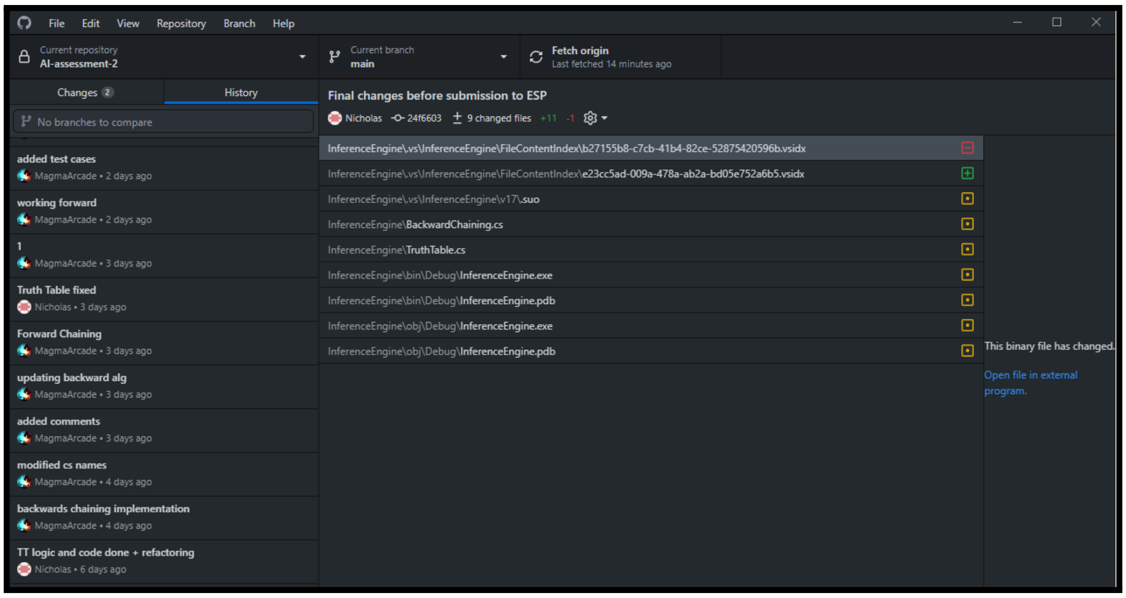
- Reading in and parsing the knowledge base (ReadKB.cs)
- Implementation of Forward Chaining algorithm (ForwardChaining.cs)
- Implementation of Backward Chaining algorithm (BackwardChaining.cs)

Complete work (Report):

- Instructions, Glossary, Features and Bugs (Forwarding Chaining, Backward Chaining, ReadKB), Test Cases, Team Summary Report, Bibliography

Feedback mechanisms

This was an extremely collaborative and open process at every step of the way. Feedback between team members was given through many different channels, including during class, over SMS and over Discord. The **shared GitHub repository** (screenshot below), which enabled up-to-date collaboration on the same code, was also a very useful tool for updating on progress and providing feedback on fixes, etc.



AI-assessment-2
main
Last fetched 14 minutes ago
No branches to compare

added test cases
MagmaArcade • 2 days ago

working forward
MagmaArcade • 2 days ago

1
MagmaArcade • 3 days ago

Truth Table fixed
Nicholas • 3 days ago

Forward Chaining
MagmaArcade • 3 days ago

updating backward alg
MagmaArcade • 3 days ago

added comments
MagmaArcade • 3 days ago

modified cs names
MagmaArcade • 4 days ago

backwards chaining implementation
MagmaArcade • 4 days ago

TT logic and code done + refactoring
Nicholas • 6 days ago

Final changes before submission to ESP
Nicholas 24f6603 9 changed files +11 -1

```
InferenceEngine\.vs\InferenceEngine\FileContentIndex\b27155b8-c7cb-41b4-82ce-52875420596b.vsidx
InferenceEngine\.vs\InferenceEngine\FileContentIndex\e23cc5ad-009a-478a-ab2a-bd05e752a6b5.vsidx
InferenceEngine\.vs\InferenceEngine\v17\.suo
InferenceEngine\BackwardChaining.cs
InferenceEngine\TruthTable.cs
InferenceEngine\bin\Debug\InferenceEngine.exe
InferenceEngine\bin\Debug\InferenceEngine.pdb
InferenceEngine\obj\Debug\InferenceEngine.exe
InferenceEngine\obj\Debug\InferenceEngine.pdb
```

This binary file has changed.
Open file in external program.

Bibliography (IEEE)

[1] City University of New York, “Introduction to Intelligent Systems,” in *Design and implementation of software applications 2*, 2010

[2] B. Vo, “Logical Agents & Knowledge Representation,” in *COS30019: Introduction to Artificial Intelligence*, 2023

[3] C. Franks, “Propositional logic,” Stanford Encyclopedia of Philosophy, https://plato.stanford.edu/entries/logic-propositional/#:~:text=Propositional%20logic%20is%20the%20study, sentences%27%20truth%20or%20assertability%20conditions

[4] B. Vo, “Assignment 2 – Inference Engine,” Assignment 2 – Inference Engine for Propositional Logic, https://swinburne.instructure.com/courses/49155/files/23954335?wrap=1.

[5] Truth table generator, https://web.stanford.edu/class/cs103/tools/truth-table-tool/.

[6] shashipk11, “Generate all the binary strings of N bits,” Geeks for Geeks, https://www.geeksforgeeks.org/generate-all-the-binary-strings-of-n-bits/.

[7] T. Donaldson, “7. Parsing Postfix Expressions,” CMPT 125 Summer 2012 1 documentation, https://www2.cs.sfu.ca/CourseCentral/125/tjd/postfix.html.

[8] Javapoint, “Forward chaining and backward chaining in AI,” www.javatpoint.com, https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai (accessed May 10, 2023).