ASSIGNMENT 2
INFERENCE ENGINE FOR PROPOSITIONAL LOGIC

Manh Dung Nguyen – 104181789 Vu Duc Tran – 104175614

MAY 14, 2024

SWINBURNE UNIVERSITY OF TECHNOLOGY

Table of Contents

Instructions

This project simply uses Python to run. Please make sure that all files are in the same folder. You can run by entering the following command:

“python iengine.py <filename> <method>”

where [methods] is one of TT, FC, BC <filename> is the name of the containing the knowledge edge base and query. **For instance**: python iengine.py test_hornform1.txt TT.

The result will show YES if you select the TT (truth table checking) technique, followed by the number of models in which both the query and the KB are true or false.

If you select the forward chaining (FC) or backward chaining (BC) approach, the result will either be NO or YES followed by a series of symbols.

Introduction

This paper outlines our work for Assignment 2 of COS30019 on the propositional logic inference engine. For this assignment, we implemented forward chaining, backward chaining, and truth table checking methods. Additionally, we developed a comprehensive logic parser as part of the research component. This project was a collaborative effort between Manh Dung Nguyen (104181789) and Vu Duc Tran (104175514).

Inference methods

Forward Chaining: Forward chaining begins with an initial set of facts and uses inference rules to deduce new facts iteratively. Each rule is examined to see if its premises are met by the known facts. If they are, the conclusion of the rule is added to the set of known facts.

Backward Chaining: Backward chaining starts with the desired conclusion and looks for rules that could lead to this conclusion. It then checks the premises of those rules, working backward until it reaches known facts.

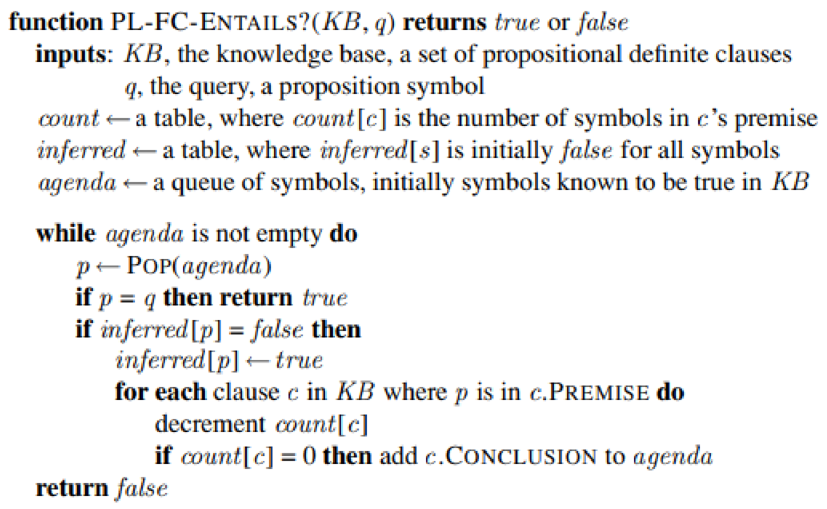
Truthtable Checking: A truth table lists all possible truth values for a given set of propositions. Each row of the table represents a unique combination of truth values for the variables. The method evaluates the truth of the overall proposition for each combination.

Implementation

Forward chaining

This implementation utilizes a systematic approach to check all possible facts derived from the initial set of known facts.

In this assignment, we implement the forward chaining function using the pseudocode provided on the book AI: A Modern Approach (3rd edition).



```
function PL-FC-ENTAILS?($KB$, $q$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a set of propositional definite clauses
          $q$, the query, a proposition symbol
  $count \leftarrow$ a table, where $count[c]$ is the number of symbols in $c$'s premise
  $inferred \leftarrow$ a table, where $inferred[s]$ is initially $false$ for all symbols
  $agenda \leftarrow$ a queue of symbols, initially symbols known to be true in $KB$

  while $agenda$ is not empty do
      $p \leftarrow$ POP($agenda$)
      if $p = q$ then return $true$
      if $inferred[p] = false$ then
          $inferred[p] \leftarrow true$
          for each clause $c$ in $KB$ where $p$ is in $c$.PREMISE do
              decrement $count[c]$
              if $count[c] = 0$ then add $c$.CONCLUSION to $agenda$
  return $false$
```

Figure 1

The Forward Chaining algorithm continues to infer new symbols from the knowledge base until it either confirms the query or can no longer make any inferences. It starts by setting up a count for the premises of each rule, indicating how many premises must be satisfied for the rule to be applicable.

It initializes a list (agenda) of known true symbols and tracks which symbols have been inferred. The algorithm processes each symbol from the agenda, and if the symbol matches the query, it returns "true" and shows the steps taken.

If a symbol has not yet been inferred, it marks it as inferred. The algorithm then examines each rule, decrementing the premise count if the current symbol is a premise. If all premises of a rule are satisfied, the rule's conclusion is added to the agenda.

If the query is not confirmed after exhausting all inferences, the algorithm returns "false".

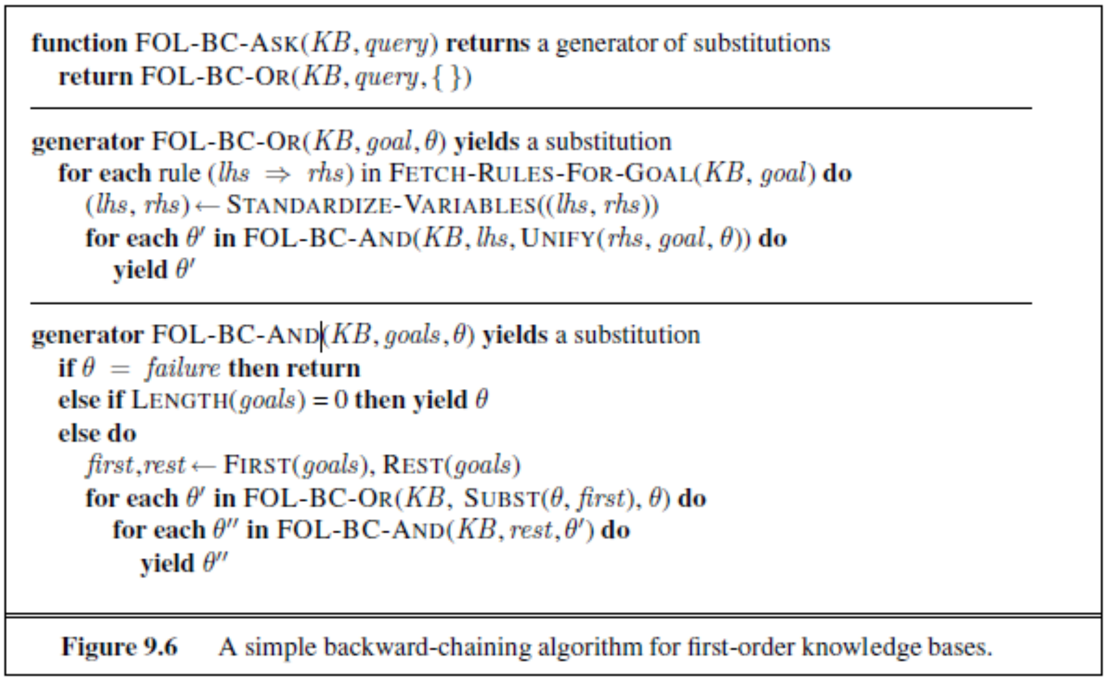
Approach:

- Data-driven: This method continuously expands the set of known facts, ensuring all possible conclusions are reached from the initial facts.
- Efficiency: May generate many intermediate facts, leading to potential inefficiency if the set of rules and facts is large.

Backward chaining

Backward chaining (BC), on the other hand, employs a different strategy by working from the desired goal back to the known facts to determine if the goal can be satisfied.

It begins with the query and attempts to find rules whose conclusions match the query. If a rule's conclusion matches the query, it checks if all the premises of that rule can be satisfied by the known facts. If so, the query is confirmed. If not, it recursively checks the premises of the rule to see if they can be satisfied, thus moving backward through the chain of reasoning.



```
function FOL-BC-ASK($KB$, $query$) returns a generator of substitutions
  return FOL-BC-OR($KB$, $query$, { })

generator FOL-BC-OR($KB$, $goal$, $\theta$) yields a substitution
  for each rule ($lhs \Rightarrow rhs$) in FETCH-RULES-FOR-GOAL($KB$, $goal$) do
    ($lhs$, $rhs$) ← STANDARDIZE-VARIABLES(($lhs$, $rhs$))
    for each $\theta'$ in FOL-BC-AND($KB$, $lhs$, UNIFY($rhs$, $goal$, $\theta$)) do
      yield $\theta'$

generator FOL-BC-AND($KB$, $goals$, $\theta$) yields a substitution
  if $\theta$ = $failure$ then return
  else if LENGTH($goals$) = 0 then yield $\theta$
  else do
    $first$,$rest$ ← FIRST($goals$), REST($goals$)
    for each $\theta'$ in FOL-BC-OR($KB$, SUBST($\theta$, $first$), $\theta$) do
      for each $\theta''$ in FOL-BC-AND($KB$, $rest$, $\theta'$) do
        yield $\theta''$
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

Figure 2

The Backward Chaining algorithm starts with the query and recursively evaluates whether the known facts can support it. It traces backward through the inference chain, checking each rule's premises until it either confirms the query or cannot find support for it.

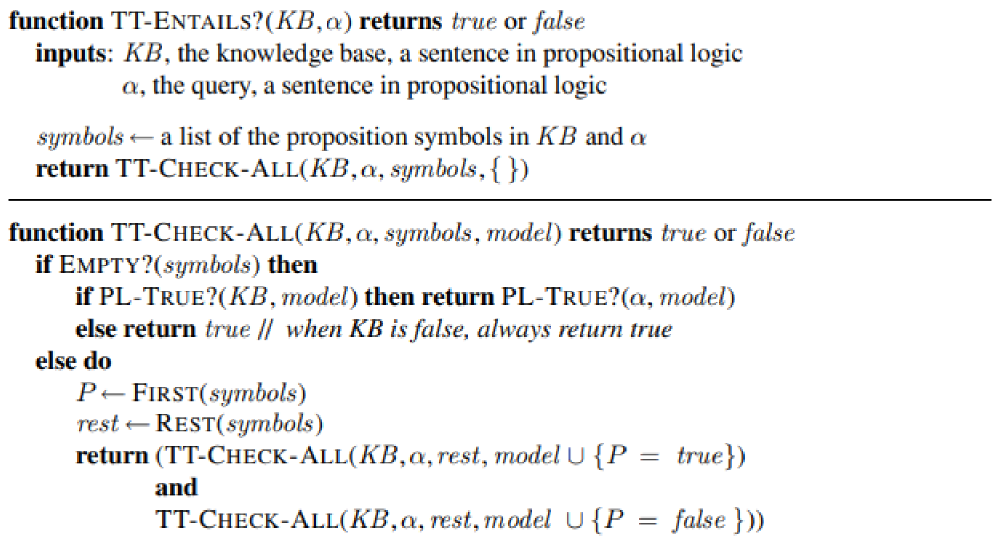
This process efficiently targets the specific goal but may involve tracing through numerous rules and premises.

Approach:

- Data-driven: Focuses directly on proving the goal, often resulting in fewer intermediate steps compared to forward chaining.
- Efficiency: More efficient for specific queries but can involve significant backtracking if the goal has multiple dependencies.

Truth table checking

Truth tables, unlike forward or backward chaining, offer a comprehensive and exhaustive method for evaluating the truth values of logical expressions or propositions.



```
function TT-ENTAILS?($KB, \alpha$) returns true or false
  inputs: $KB$, the knowledge base, a sentence in propositional logic
          $\alpha$, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in $KB$ and $\alpha$
  return TT-CHECK-ALL($KB, \alpha$, symbols, { })
```

```
function TT-CHECK-ALL($KB, \alpha$, symbols, model) returns true or false
  if EMPTY?(symbols) then
     if PL-TRUE?($KB$, model) then return PL-TRUE?($\alpha$, model)
     else return true //  when KB is false, always return true
  else do
     $P$ ← FIRST(symbols)
     rest ← REST(symbols)
     return (TT-CHECK-ALL($KB, \alpha$, rest, model $\cup$ {$P$ = true})
            and
            TT-CHECK-ALL($KB, \alpha$, rest, model $\cup$ {$P$ = false }))
```

Figure 3

Truth tables don't involve inference like chaining algorithms; instead, they directly evaluate the truth values based on the input propositions.

First, each premise and conclusion is symbolized. Then, a truth table is created with columns for each premise and the conclusion. The truth table is populated with all possible combinations of truth values for the variables involved. Each row represents a unique combination of truth values.

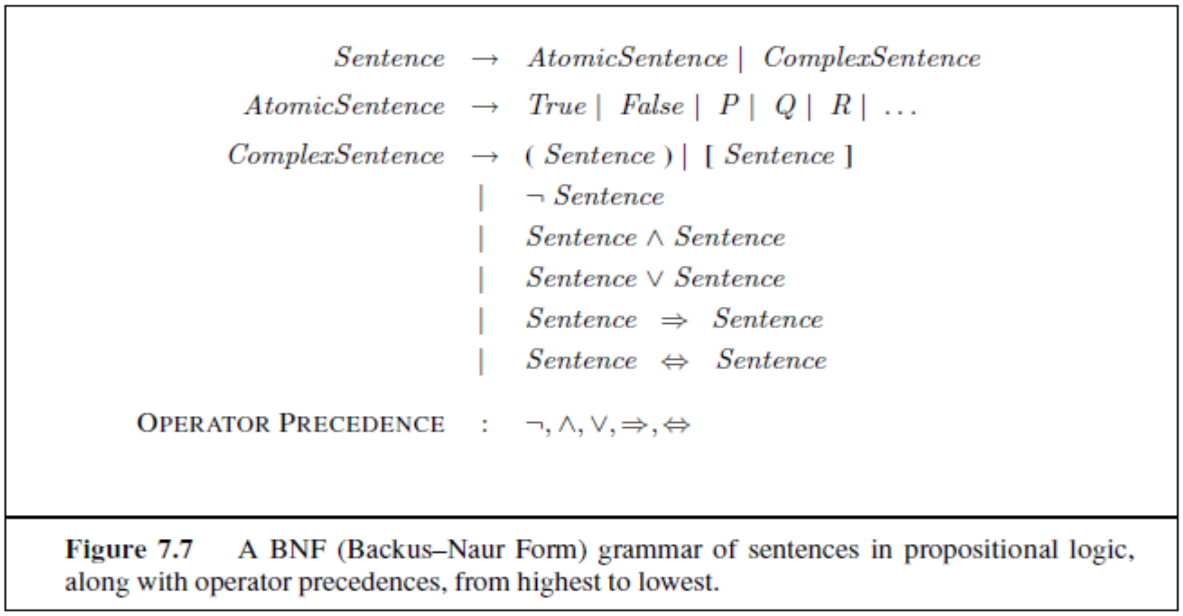
Subsequently, the premises are evaluated based on the truth values in each row. If all premises are true in a particular row and the conclusion is false, the argument is considered invalid. If there is no row where all premises are true and the conclusion is false, the argument is valid.

Approach:

- Data-driven: Employs a systematic evaluation of all possible combinations of truth values.
- Efficiency: May become impractical for large and complex systems due to their exponential growth in size.

Research

General Sentence



$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & & \\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 4

We follow above grammar of sentence to implement our General Sentence. The BNF (Backus–Naur Form) grammar offers a systematic framework for representing sentences in propositional logic. It defines the structure of a `Sentence` as being either an `AtomicSentence` or a `ComplexSentence`.

An `AtomicSentence` denotes the simplest form of a statement, representing either the truth values True or False, or employing propositional variables like P, Q, R, and so forth, serving as placeholders for specific assertions or predicates. Conversely, a `ComplexSentence` encompasses more intricate logical compositions, capable of encapsulating multiple `Sentences` through various means such as enclosure within parentheses or square brackets, negation, or combination using logical connectives like conjunction (∧), disjunction (∨), implication (⇒), or biconditional (⇔). This structured grammar lays the foundation for precise expression and manipulation of logical statements, facilitating sophisticated reasoning and deduction within the realm of propositional logic.

Notability, according to the requirements, in Sentence.py file we use the following syntax for different logical connectives:

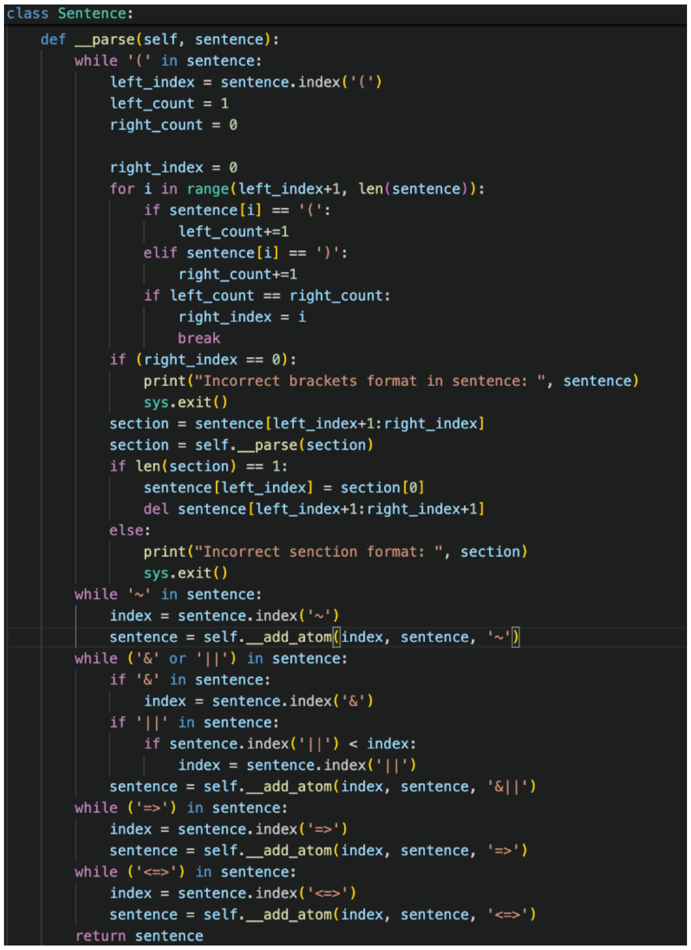
~ for negation (¬)

& for conjunction (∧)

|| for disjunction (∨)

=> for implication (⇒)

<=> for biconditional (⇔)



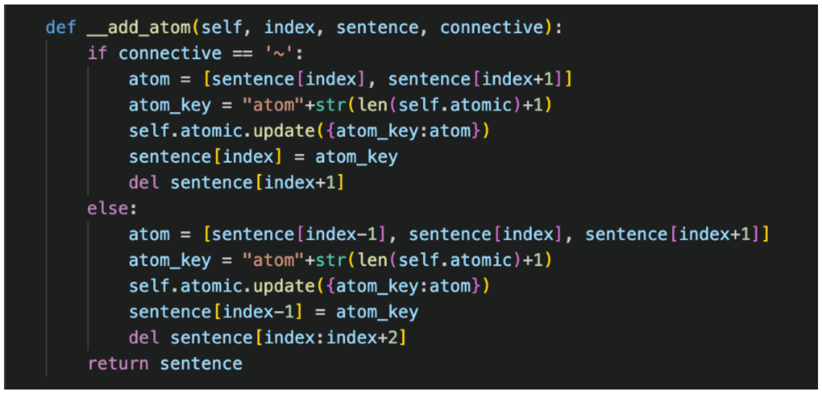
```python
class Sentence:
    def __parse(self, sentence):
        while '(' in sentence:
            left_index = sentence.index('(')
            left_count = 1
            right_count = 0

            right_index = 0
            for i in range(left_index+1, len(sentence)):
                if sentence[i] == '(':
                    left_count+=1
                elif sentence[i] == ')':
                    right_count+=1
                if left_count == right_count:
                    right_index = i
                    break
            if (right_index == 0):
                print("Incorrect brackets format in sentence: ", sentence)
                sys.exit()
            section = sentence[left_index+1:right_index]
            section = self.__parse(section)
            if len(section) == 1:
                sentence[left_index] = section[0]
                del sentence[left_index+1:right_index+1]
            else:
                print("Incorrect senction format: ", section)
                sys.exit()
        while '~' in sentence:
            index = sentence.index('~')
            sentence = self.__add_atom(index, sentence, '~')
        while ('&' or '||') in sentence:
            if '&' in sentence:
                index = sentence.index('&')
            if '||' in sentence:
                if sentence.index('||') < index:
                    index = sentence.index('||')
            sentence = self.__add_atom(index, sentence, '&||')
        while ('=>') in sentence:
            index = sentence.index('=>')
            sentence = self.__add_atom(index, sentence, '=>')
        while ('<=>') in sentence:
            index = sentence.index('<=>')
            sentence = self.__add_atom(index, sentence, '<=>')
        return sentence
```

Figure 5

The `__parse` method processes a logical sentence by recursively resolving nested expressions and converting them into atomic components. Initially, it identifies and handles the innermost parenthetical sections. It searches for matching parentheses, ensuring balanced pairs, and recursively calls itself to parse the contents within these parentheses. If the brackets are not balanced, the method outputs an error and terminates.

After processing the parentheses, the method addresses unary operations like negation (`~`) by locating each instance and calling the `__add_atom` method to replace these segments with unique atom keys. The method then handles binary operations (`&` and `||`) by identifying the leftmost occurrence of either operator and using `__add_atom` to atomize these expressions. This step ensures that binary operations are processed in the correct order of precedence.

Next, the method processes implication (`=>`) and biconditional (`<=>`) operators, again using `__add_atom` to convert these into atomic components. Each operation ensures that the logical expressions are systematically reduced to simpler forms, represented by atom keys, facilitating easier manipulation and evaluation in subsequent processes. The fully parsed and atomized sentence is then returned, with all complex expressions decomposed into atomic elements.



```python
def __add_atom(self, index, sentence, connective):
    if connective == '~':
        atom = [sentence[index], sentence[index+1]]
        atom_key = "atom"+str(len(self.atomic)+1)
        self.atomic.update({atom_key:atom})
        sentence[index] = atom_key
        del sentence[index+1]
    else:
        atom = [sentence[index-1], sentence[index], sentence[index+1]]
        atom_key = "atom"+str(len(self.atomic)+1)
        self.atomic.update({atom_key:atom})
        sentence[index-1] = atom_key
        del sentence[index:index+2]
    return sentence
```

Figure 6

The `__add_atom` method is designed to process a given sentence by replacing specific segments with unique identifiers, referred to as atom keys, and updating an internal dictionary (`self.atomic`) with these atomized segments.

For unary operations such as negation (`~`), the method identifies the segment comprising `sentence[index]` and `sentence[index+1]`, generates an atom, and substitutes this segment in the sentence with an atom key, subsequently removing the next element.

In the case of binary operations (e.g., `&`, `|`), the method extracts the segment consisting of `sentence[index-1]`, `sentence[index]`, and `sentence[index+1]`, generates an atom, and replaces this segment with an atom key, deleting the previous and next elements in the process.

The modified sentence, now containing atom keys instead of the original segments, is then returned.

This implementation enables the Truth Table Checking algorithm to read both Horn form and general sentence knowledge bases.

Testing

We have created a number of test cases during this project to make sure our software continues to produce the right results in a range of conditions. (including HornForm test and GeneralSentence test)

Test Name	Test type	Test Input	TT Checking	FC	BC
HORNFORM					
test_hornform1.txt	**Single Positive Literal**	TELL P; ASK P	YES:1	YES: p	YES: p
test_hornform2.txt	**Single Negative Literal:**	TELL ~P; ASK ~P	YES:1	YES: ~p	YES: ~p
test_hornform3.txt	**Definite Clause with One Literal:**	TELL P => Q; P; ASK Q	YES:1	YES: p, q	YES: p, q
test_hornform4.txt	**Definite Clause with Two Literals:**	TELL P & Q => R; P; Q; ASK R	YES:1	YES: p, q, r	YES: p, q, r
test_hornform5.txt	**Definite Clause with Three Literals:**	TELL P & Q & R => S; P; Q; R ASK S	YES:1	YES: p, q, r, s	YES: p, q, r, s
test_hornform6.txt	**Definite Clause with a Combination of Literals:**	TELL (P & Q) => R; P; Q ASK R	YES:1	YES: p, q, r	YES: p, q, r
test_hornform7.txt	**Definite Clause with Contradiction: (NOT HF)**	TELL P & ~P => Q; P; ASK Q	NO	NO	NO
test_hornform8.txt	**Definite Clause with Disjunction:**	TELL (P ∨ Q) => R; ASK R	NO	NO	NO
GENERAL SENTENCE					
test_general1.txt	**Simple Chain**	TELL P => Q => R; ASK P => (Q => R)	YES:5	x	x
test_general2.txt	**Longer Chain**	TELL	YES:11	x	x

		P => Q => R => S; ASK P => (Q => (R => S))			
test_general3.txt	**Chain with Negation**	TELL P => ~Q => R; ASK P => (~Q => R)	YES:5	x	x
test_general4.txt	**Chain with Mixed Literals**	TELL P => (Q & R) => S; ASK P => ((Q & R) => S)	YES:11	x	x
test_general5.txt	**Chain with Nested Implications**	TELL (P => Q) => (R => S); ASK (P => Q) => (R => S)	YES:13	x	x

test_general6.txt	**Chain with Complex Nested Implications**	TELL (P => (Q => R)) => (S => T); ASK (P => (Q => R)) => (S => T)	YES:25	x	x
test_general7.txt	**Chain with Multiple Negations**	TELL ~P => ~Q => R; ASK ~P => (~Q => R)	YES:5	x	x
test_general8.txt	**Chain with Disjunction in Middle**	TELL P => (Q ∨ R) => S; ASK P => ((Q ∨ R) => S)	YES:5	x	x

Forward chaining was useful for problems where a wide range of consequences needed to be derived from initial facts, providing a thorough exploration of all possible outcomes.

Backward chaining proved effective for targeted problem-solving, where specific goals were identified, allowing for a more focused approach.

Truth table checking was utilized for its thoroughness, ensuring that our logic system could verify propositions with complete accuracy, albeit at the cost of efficiency with larger datasets.

Features/ Bugs/ Missing

1. Features

- Truth Table, Forward Chaining, Backward Chaining
- Horn form, General sentence Knowledge Base

2. Bugs, Missing

Team summary report

We both tried to work together to complete this task and made contributions to it. GitHub is used by team members to exchange source code, while Messenger is used for communication.

We shall convene in person or virtually whenever a member is having trouble with his portion. Members can sent a message to another to find answers to simple inquiries.

Vu Duc Tran	Manh Dung Nguyen
Total contribution: 55%, including: • Forward chaining • Backward chaining • Horn Form • Testing and debugging • Test case development	Total contribution: 45%, including: • General Sentence • Truth table checking algorithm • Test case development • Writing Report

Acknowledgement/ Resources

AI: A Modern Approach (3rd edition) (This textbook provides us with all the foundational knowledge

to complete this assignment. Much of our code is based on the pseudocode implementations in the book.)

https://homepage.cs.uri.edu/~cingiser/csc481/chapter_notes/amarant.pdf (This page helps us come up with the implementation of the backward chaining method)

References

1. Russell, S., & Norvig, P. (2021). "Artificial Intelligence: A Modern Approach". Pearson.