Introduction to Artificial Intelligence

Assignment 2 Report
Inference Engine for Propositional Logic

Xuan Tuan Minh Nguyen - 103819212
Jordan Ardley - 104024703

Instruction

In order to execute the program, simply navigate to the Command Line Interface, change the directory to the directory containing iengine.py file, and use the following syntax as given in the Figure 1: "python iengine.py <filename> <method>". Where <filename> is the text file that has the propositional sentence and query to ask, and <method> can be entered in either lowercase and uppercase, as long as it matches the following list: fc, bc and tt. The expected output of the program will be either "YES" or "NO", which "NO" stands for no solutions are found within the iteration and "YES" stands for solutions are found within the iteration. The result may vary based on the chosen method, while the result of Truth Table Checking will show the amount of models that match with the query, Forward Chaining and Backward Chaining will print out a list of symbols entailed from the knowledge base during the program's execution.

Figure 1: Example syntax to run the command on Command Line Interface

Introduction

1. Inference Engine

In the field of Artificial Intelligence, Inference Engines have been one of the most interesting topics that are discussed and evaluated ever since the terminology Artificial Intelligence was introduced. Inference Engines use a set of logical rules to learn new knowledge and decide what to do based on that set of rules, which operates on a routine of matching the rules with given data, selecting which rules to apply and executing those rules to provide new data. Throughout the growth of Artificial Intelligence, more and more modern and efficient algorithms regarding Inference Engines have been introduced and released to the public with performance being optimized. However, the majority of those algorithms are based on the three root algorithms that made Inference Engine, which are Truth Table, Forward Chaining and Backward Chaining algorithm. This report is written to give a general perspective of the mentioned algorithms, which provide an overview on the algorithms, the complexity, and the performance of each algorithm on different scenarios. Moreover, this report aims to discuss the implementation of each algorithm using Python, points out features and bugs and a brief statistic on the contribution of each member.

Inference Engine Algorithms

1. General Concept

This category aims to introduce three most popular Inference Engine Algorithms, which are Truth Table Checking Algorithm, Forward Chaining Algorithm and Backward Chaining Algorithm. In addition, each given algorithm will be thoroughly examined in terms of space complexity (memory usage) and time complexity (efficiency).

2. Truth Table Checking

Overview

Truth table checking is an algorithm used for checking the validity of logical expressions by evaluating all possible truth value combinations of its variables. The process begins by identifying propositional variables involved in the given sentence. Then, once all propositional variables are identified, a truth table is created to list all possible combinations of true and false values for these variables. For each combination, according to its structure, the logical statement is evaluated by applying logical operations, such as AND, OR and NOT, and stored in the truth table. Finally, the results in the truth table will be analyzed to classify as a tautology statement or a contradiction statement, which if the expression is true for all combinations then it is a tautology statement, otherwise it is a contradiction statement.

Time Complexity

Suppose n is the amount of propositional statements and m is the amount of logical operations. The time complexity of this algorithm is $O\left(2^n * m\right)$, the reason for this is there are 2^n possible truth value combinations and for each combination, it requires $O(m)$ time to evaluate the expression. On the other hand, the space complexity of this algorithm is $O(2^n * n)$, as the algorithm must stores 2^n rows containing n variables and the expression of the result. Both the time and space complexity of this algorithm is exponential, thus making it impractical with large amounts of propositional statements.

3. Forward Chaining Algorithm

Overview

Forward Chaining is a popular inference algorithm that is used for inferring conclusions from a chain of known facts and rules. The underlying procedure of this algorithm begins with a set of facts and if-else rules to iteratively generate new facts until reaching the goal or no more rules are applicable. The algorithm matches the rules with satisfied antecedents, picks and applies a rule, and adds the created consequent to the given facts. This sequence continues until a goal fact is found or no new facts could be inferred.

Time Complexity

Suppose R is the number of rules and F is the number of facts. Since the Forward Chaining algorithm will perform checking known facts for each rules, thus the time complexity in the worst case for this algorithm is $O(F * R)$. The space complexity of this algorithm is $O(F + R)$ as the algorithm needs to store all inferred facts and rules for further processes.

4. *Backward Chaining Algorithm*

Overview

Similar to the forward chaining algorithm, backward chaining is also an inference method that uses a set of facts and rules to infer conclusions. The algorithm begins searching for the rules that conclude the goal. It then looks up if the conditions of the rule are true, otherwise these antecedents will be treated as new sub-goals. The process will continue recursively until the antecedents match the facts, then the goal is reached.

Time Complexity

Suppose d is the depth of the goal tree and b is the branching factor of each goal node (number of rules that can apply). Since the backward chaining algorithm heavily depends on the number of goals and subgoals to evaluate, thus the time complexity of backward chaining algorithm is: $O\left(b^{d}\right)$. And the space complexity of this algorithm is $O(d)$, as it is primarily driven by the depth of the recursion and the storage of intermediate goals.

Implementations

1. Horn Form and Generic Propositional Logic Form

Horn Form

```python
from Components.Interfaces.IHornForm import IHornForm
from Components.Decorator.Export import export
import re


@export
class HornForm(IHornForm):
    def __init__(self, sentence) -> None:
        ######
        # IHornForm initializer
        # self.clauses = []  # To keep the clauses of the sentence
        # self.symbols = set()  # Use a set to avoid duplicates
        # self.sentence = sentence  # The sentence that we want to convert into Horn Form
        # self.head = ""  # The head of the sentence
        # self.conjuncts = []  # The conjuncts of the sentence
        ######
        super().__init__(sentence)
        # Evaluate the given sentence to convert it into Horn form
        self.evaluate()

    def evaluate(self):
        # Split the sentence into clauses based on logical operators and parentheses
        self.clauses = re.split("(=>|&|\(|\)|~|\|\||<=>)", self.sentence)
        # Strip whitespace and filter out empty strings and parentheses from the clauses
        self.clauses = [c.strip()
                        for c in self.clauses if c.strip() and c not in "()"]

        # Check if the sentence contains any operators that are not allowed in Horn form
        if any(op in self.clauses for op in ['~', '||', '<=>']):
            # Raise an error if the sentence is not in Horn form
            raise ValueError("Sentence is not in horn form", self.clauses)

        # If the sentence is a fact (only one clause)
        if len(self.clauses) == 1:
            self.head = self.clauses[0]
            self.symbols.add(self.head)
        else:
            # Find the index of the implication operator
            index = self.clauses.index('=>')
            # Get the right-hand side (head) of the implication
            rhs = self.clauses[index + 1:]
            if len(rhs) > 1:
                # Raise an error if the right-hand side is not a single symbol
                raise ValueError("Error horn form format", self.clauses)
            self.head = rhs[0]
            self.symbols.add(self.head)

            # Get the left-hand side (body) of the implication
            lhs = self.clauses[:index]
            if lhs[0] == '&' or lhs[-1] == '&':
                # Raise an error if the body starts or ends with a conjunction operator
                raise ValueError("Error horn form format", self.clauses)

            # Check for consecutive conjunction operators
            for i in range(len(lhs) - 1):
                if lhs[i] == '&' and lhs[i + 1] == '&':
                    # Raise an error if there are consecutive conjunction operators
                    raise ValueError("Error horn form format", self.clauses)
            for value in lhs:
                if value != '&':
                    # Add each symbol in the body to the list of conjuncts and symbols
                    self.conjuncts.append(value)
                    self.symbols.add(value)

        # Convert the set of symbols back to a list for the final output
        self.symbols = list(self.symbols)
```

Figure 2: HornForm Class

The given code described in Figure 2 describes the implementation of 'HornForm' class, which converts the logical sentence into Horn form, which is a logical expression type used in propositional logic. The main functionality of this class is to process a logical sentence to ensure that it meets with the rules of Horn Form. Firstly, it divides the sentence into clauses based on the operators and parentheses, stripping any whitespaces and looking for any invalid characters or any operators that are not allowed in Horn Form, which are negation, disjunction and bi-conditional ('~', '||', '<=>'). For a sentence that is single-clause, the clause is treated as a fact and will be the head. For a sentence that contains implications ('=>'), the method will separate the left-hand side clause as the body and right-hand side clause as head, and guard for the right-hand side clause to be a single symbol (otherwise will raise an error). It also watches for the left-hand side clause (body) to ensure that it does not start or end with conjunctions ('&') and does not have any consecutive junctions ('&&'), and adds valid symbols to the conjuncts and symbols list. Finally, the symbols set will be converted to a list, ensuring that the sentence complies with the rules of Horn Form.

Knowledge base

```python
def formatOriginal(self, sentence):
    # Split the sentence into tokens using logical operators and parentheses as delimiters
    orig = re.split("(=>|&|\(|\)|~|\|\||<=>)", sentence)
    # Remove any empty strings from the list of tokens
    while "" in orig:
        orig.remove("")
    # Strip whitespace from each token and store the cleaned tokens in self.original
    self.original = [token.strip() for token in orig if token.strip()]
    # Split the sentence to extract symbols using only logical operators as delimiters
    symbols = re.split("=>|&|\(|\)|~|\|\||<=>", sentence)
    # Remove any empty strings from the list of symbols
    while "" in symbols:
        symbols.remove("")
    # Store unique symbols in self.symbols
    self.symbols = list(set(symbols))
    # Format the sentence to identify atomic propositions and its structure
    self.root = self.formatSentence(orig)
```

Figure 3: LogicalSentence's formatOriginal method

```python
def formatSentence(self, sentence):
    # Handle nested expressions within parentheses
    while '(' in sentence:
        # Find the index of the first opening parenthesis
        lhsIndex = sentence.index('(')
        countLeft = 1  # Initialize count of left parentheses
        countRight = 0  # Initialize count of right parentheses
        rhsIndex = 0  # Initialize the index for the matching closing parenthesis

        # Find the matching closing parenthesis for the first opening parenthesis
        for i in range(lhsIndex + 1, len(sentence)):
            if sentence[i] == '(':
                countLeft += 1  # Increment count of left parentheses
            elif sentence[i] == ')':
                countRight += 1  # Increment count of right parentheses
            if countLeft == countRight:
                rhsIndex = i  # Set the index of the matching closing parenthesis
                break
        # Raise an error if no matching closing parenthesis is found
        if rhsIndex == 0:
            raise ValueError(
                "Incorrect brackets format in sentence: ", sentence)
        # Recursively format the nested section
        section = sentence[lhsIndex + 1: rhsIndex]
        section = self.formatSentence(section)
        # If the nested section is a single token, replace the parentheses with the token
        if len(section) == 1:
            sentence[lhsIndex] = section[0]
            del sentence[lhsIndex + 1: rhsIndex + 1]
        else:
            raise ValueError("Incorrect section format: ", section)

    # Process negation operators
    while '~' in sentence:
        # Find the index of the negation operator
        index = sentence.index('~')
        # Process the negation operator
        sentence = self.appendAtomic(index, sentence, '~')

    # Process conjunction and disjunction operators
    while '&' in sentence or '||' in sentence:
        if '&' in sentence and ('||' not in sentence or sentence.index('&') < sentence.index('||')):
            # Find the index of the conjunction operator
            index = sentence.index('&')
        elif '||' in sentence:
            # Find the index of the disjunction operator
            index = sentence.index('||')
        else:
            break
        # Process the conjunction or disjunction operator
        sentence = self.appendAtomic(index, sentence, '&||')

    # Process implication operators
    while '=>' in sentence:
        # Find the index of the implication operator
        index = sentence.index('=>')
        # Process the implication operator
        sentence = self.appendAtomic(index, sentence, '=>')

    # Process biconditional operators
    while '<=>' in sentence:
        # Find the index of the biconditional operator
        index = sentence.index('<=>')
        # Process the biconditional operator
        sentence = self.appendAtomic(index, sentence, '<=>')

    return sentence  # Return the formatted sentence
```

Figure 4: LogicalSentence's formatSentence method

```python
def appendAtomic(self, index, sentence, operator):
    if operator == '~':
        # Handle negation operator
        # Create an atomic proposition for negation
        tempAtomic = [sentence[index], sentence[index + 1]]
        # Generate a unique key for the atomic proposition
        tempAtomicKey = "atom" + str(len(self.atom) + 1)
        # Add the atomic proposition to self.atom
        self.atom.update({tempAtomicKey: tempAtomic})
        # Replace the operator and its operand with the atomic proposition key
        sentence[index] = tempAtomicKey
        del sentence[index + 1]  # Remove the operand from the sentence
    else:
        # Handle binary operators (conjunction, disjunction, implication, biconditional)
        # Create an atomic proposition for binary operators
        tempAtomic = [sentence[index - 1],
                      sentence[index], sentence[index + 1]]
        # Generate a unique key for the atomic proposition
        tempAtomicKey = "atom" + str(len(self.atom) + 1)
        # Add the atomic proposition to self.atom
        self.atom.update({tempAtomicKey: tempAtomic})
        # Replace the operands and operator with the atomic proposition key
        sentence[index - 1] = tempAtomicKey
        # Remove the operands and operator from the sentence
        del sentence[index:index + 2]
    return sentence  # Return the updated sentence
```

Figure 5: LogicalSentence's appendAtomic method

```python
def evaluate(self, model):
    # Create a dictionary to map symbols to their boolean values from the provided model
    bool_pairs = {}
    for value in self.symbols:
        if value in model:
            # Add the symbol and its boolean value to bool_pairs
            bool_pairs.update({value: model[value]})
        else:
            # Raise an error if a symbol does not have a boolean value in the model
            raise ValueError("No boolean for all symbols.")

    # Evaluate each atomic proposition based on the logical operator
    for key in self.atom:
        if len(self.atom[key]) == 2:
            # Evaluate negation
            # Get the boolean value of the operand
            right = bool_pairs[self.atom[key][1]]
            # Negate the boolean value and update bool_pairs
            bool_pairs.update({key: not right})
        elif len(self.atom[key]) == 3:
            # Get the boolean value of the left operand
            left = bool_pairs[self.atom[key][0]]
            # Get the boolean value of the right operand
            right = bool_pairs[self.atom[key][2]]
            if self.atom[key][1] == '&':
                # Evaluate conjunction
                bool_pairs.update({key: left and right})
            elif self.atom[key][1] == '||':
                # Evaluate disjunction
                bool_pairs.update({key: left or right})
            elif self.atom[key][1] == '=>':
                # Evaluate implication
                bool_pairs.update({key: not left or right})
            elif self.atom[key][1] == '<=>':
                # Evaluate biconditional
                bool_pairs.update({key: left == right})
        else:
            # Raise an error if an atomic proposition is not in the correct format
            raise ValueError(
                "Atomic sentence in incorrect format: ", self.atom[key])

    # Return the boolean value of the root of the sentence
    return bool_pairs[self.root[0]]
```

Figure 6: Evaluation method for LogicalSentence

The four figures above show the logic of the LogicalSentence class, which is created to transform the propositional logic into the General Knowledge Base Form for use in TruthTable class. The procedure starts up by calling the 'formatOriginal' method, which is described in Figure 3, to strip out any whitespaces and any tokens, then finds and stores any unique symbols. The method 'formatSentence' (Figure 4) is used to handle nested expressions by recursively processing any sections within parentheses and ensuring allowed operations to convert into atomic propositions. The method 'appendAtomic' described in Figure 5 is used for replacing the operands and operators by generating atomic propositions for operators. Finally, the 'evaluate' method (Figure 6) will convert the symbols into the equivalence boolean values from a provided model and evaluate each atomic proposition based on its operator, returning the boolean value of the sentence's root for use in the TruthTable method.

2. Truth Table Checking

```python
from Components.Decorator.Export import export
from Components.Implementations.LogicalSentence import LogicalSentence
from Components.Interfaces.ITruthTable import ITruthTable


@export
class TruthTable(ITruthTable):
    def __init__(self, knowledgeBase):
        super().__init__(knowledgeBase)

    def __check(self, alphabetic, symbols, model):
        # Base case: if there are no more symbols to evaluate
        if len(symbols) == 0:
            all_eval = True  # Assume all sentences in the knowledge base evaluate to True
            for sentence in self.knowledgeBase.sentences:
                # Evaluate each sentence with the current model
                if not sentence.evaluate(model):
                    all_eval = False  # If any sentence evaluates to False, set all_eval to False
            if all_eval:
                # Evaluate the query with the current model
                alpha = alphabetic.evaluate(model)
                if alpha:
                    self.count += 1  # Increment the count if the query evaluates to True
                return alpha  # Return the evaluation of the query
            else:
                return True  # If not all sentences in the knowledge base evaluate to True, return True
        else:
            firstSymbol = symbols[0]  # Get the first symbol
            restSymbol = symbols[1:]  # Get the rest of the symbols
            model1 = model.copy()  # Copy the current model
            # Set the first symbol to True in the copied model
            model1.update({firstSymbol: True})
            model2 = model.copy()  # Copy the current model again
            # Set the first symbol to False in the copied model
            model2.update({firstSymbol: False})
            # Recursively check the rest of the symbols with both models
            return (self.__check(alphabetic, restSymbol, model1) and self.__check(alphabetic, restSymbol, model2))

    def __truth_table(self, alphabetic):
        symbols = self.knowledgeBase.symbols  # Get the symbols from the knowledge base
        for symbol in alphabetic.symbols:
            if symbol not in symbols:  # Add symbols from the query if they are not already in the knowledge base
                symbols.append(symbol)
        # Start the truth table check with an empty model
        return self.__check(alphabetic, symbols, {})

    def entails(self, query):
        # Create a LogicalSentence object for the query
        alpha = LogicalSentence(query)
        # Generate the truth table for the query
        solution = self.__truth_table(alpha)
        if solution:
            # If the query is entailed, return "YES" with the count
            result = "YES: " + str(self.count)
        else:
            result = "NO"  # If the query is not entailed, return "NO"
        return result  # Return the result

```

Figure 7: Truth table

This code introduces the `TruthTable` class, which determines if a query is supported by a given knowledge base by generating truth tables. It utilizes a recursive approach to construct the truth table and ascertain query support. The `entails` method translates the query into a logical sentence and checks its support, returning either "YES" with a count if supported, or "NO". The class employs imports such as `export` for class exportability, `LogicalSentence` for logical expression handling, and `ITruthTable` for interface definition.

3. Forward Chaining Algorithm

```python
from Components.Interfaces.IForwardChaining import IForwardChaining
from Components.Decorator.Export import export


@export
class ForwardChaining(IForwardChaining):
    def __init__(self, knowledgeBase) -> None:
        super().__init__(knowledgeBase)

    def __forward_chaining(self, query):
        # Clear the chain at the beginning of the forward chaining process
        self.chain.clear()

        # Initialize agenda with facts and count conjuncts for each sentence in the knowledge base
        for sentence in self.knowledgeBase.sentences:
            if len(sentence.conjuncts) == 0:
                # If the sentence is a fact (no conjuncts)
                if sentence.head == query:
                    # If the fact matches the query, add to the chain and return True
                    self.chain.append(query)
                    return True
                # Add the fact to the agenda
                self.agenda.append(sentence.head)
            else:
                # Initialize count of conjuncts for each rule
                self.count.update({sentence: len(sentence.conjuncts)})

        # Initialize inferred status for each symbol in the knowledge base
        for symbol in self.knowledgeBase.symbols:
            self.inferred.update({symbol: False})

        # Process symbols in the agenda
        while len(self.agenda) != 0:
            p = self.agenda.pop(0)
            if not self.inferred[p]:
                # Mark the symbol as inferred and add to the chain
                self.chain.append(p)
                self.inferred[p] = True
                for sentence in self.count:
                    if p in sentence.conjuncts:
                        # Decrement the count of conjuncts for the rules containing p
                        self.count[sentence] -= 1
                        if self.count[sentence] == 0:
                            # If all conjuncts of a rule are inferred
                            if sentence.head == query:
                                # If the rule's head matches the query, add to the chain and return True
                                self.chain.append(sentence.head)
                                return True
                            # Add the rule's head to the agenda
                            self.agenda.append(sentence.head)

        # If the query cannot be inferred, return False
        return False

    def entails(self, query):
        # Start the forward chaining process to determine if the query can be inferred
        isFound = self.__forward_chaining(query)

        # Format the result based on whether the query can be inferred or not
        if isFound:
            result = "YES: " + ", ".join(self.chain)
        else:
            result = "NO"

        # Return the result
        return result
```

Figure 8: Forward chaining

This script introduces a class named `ForwardChaining`, which implements forward chaining inference for a given knowledge base. The `entails` method triggers the forward chaining process to determine if a specific query is entailed by the knowledge base. If the query is inferred, it returns "YES" along with the inference chain; otherwise, it returns "NO". The internal method `__forward_chaining` drives the forward chaining process by iterating over facts and rules in the knowledge base and updating the agenda based on inferred symbols until reaching the query or exhausting possibilities. The class utilizes imports such as `IForwardChaining` for the interface and `export` for class exportability.

4. Backward Chaining Algorithm

```python
from Components.Decorator.Export import export
from Components.Interfaces.IBackwardChaining import IBackwardChaining


@export
class BackwardChaining(IBackwardChaining):
    def __init__(self, knowledgeBase) -> None:
        ###########
        # IBackwardChaining initializer
        # self.knowledgeBase = knowledgeBase  # The knowledge base that alogrithm will use
        # self.inferred = set()  # To keep track of already inferred symbols
        # self.chain = []  # To keep the chain of inferences
        # self.visited = set()  # To keep track of visited nodes to detect circular dependencies
        ###########
        super().__init__(knowledgeBase)

    def __backward_chaining(self, query):
        # If the query is already inferred, return True
        if query in self.inferred:
            return True

        # If the query is already visited, return False to avoid circular dependencies
        if query in self.visited:
            return False
        # Add the query to the visited set
        self.visited.add(query)

        # Check if the query is a fact in the knowledge base
        for sentence in self.knowledgeBase.sentences:
            if sentence.head == query and len(sentence.conjuncts) == 0:
                # If the query is a fact (no conjuncts), add it to the chain and inferred set and remove it from visited set
                self.chain.append(query)
                self.inferred.add(query)
                self.visited.remove(query)
                return True

        # Otherwise, try to infer the query using the rules in the knowledge base
        for sentence in self.knowledgeBase.sentences:
            if sentence.head == query:
                # Check if all conjuncts of the rule can be inferred recursively
                all_conjuncts_inferred = True
                for conjunct in sentence.conjuncts:
                    if not self.__backward_chaining(conjunct):
                        all_conjuncts_inferred = False
                        break

                # If all conjuncts are inferred, then the query can be inferred
                if all_conjuncts_inferred:
                    self.chain.append(query)
                    self.inferred.add(query)
                    self.visited.remove(query)
                    return True
        self.visited.remove(query)
        # If the query cannot be inferred from the facts or rules, return False
        return False

    def entails(self, query):
        # Clear the inferred set and chain list before starting
        self.inferred.clear()
        self.chain.clear()

        # Start the backward chaining process to determine if the query can be inferred
        isFound = self.__backward_chaining(query)

        # Format the result based on whether the query can be inferred or not
        if isFound:
            result = "YES: " + ", ".join(self.chain)
        else:
            result = "NO"

        # Return the result
        return result
```

Figure 9: Backward chaining

This code introduces a class called `BackwardChaining`, which employs backward chaining inference for a provided knowledge base. The `entails` function initiates the backward chaining process to ascertain if a specific query is supported by the knowledge base. If the query is established, it returns "YES" alongside the chain of inference; otherwise, it returns "NO". The internal method `__backward_chaining` drives the backward chaining process by examining if facts or rules in the knowledge base can deduce the query, employing recursion. The class utilizes imports like `IBackwardChaining` for interface implementation and `export` for class export capability.

Testing

Tell	Ask	Result
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p2&p1&p3=>d; p1&p3 => c; a; b; p2;	d	YES: 3
x => y; y => z; z => x; x&y&z => w; a; c; f; x;	w	YES: 1
a&e&b =>c; d =>c; a =>c; c&a&d =>e; a&b =>e; b&d =>e; d =>c; c&b =>e; b;	c	NO

Table 1: Test result for Truth Table in Horn Form

Tell	Ask	Result
(a <=> (c => ~d)) & b & (b => a); c; ~f || g;	~d & (~g => ~f)	YES: 3
(b=>a);(~d=>~(b=>a));(~b||a);(b=>~d);(a=>f);	(~(b=>a)<=>(a=>f))	NO
(~b<=>~c);(~e&~c);(~f<=>~ g);(d<=>~c);(~(~b<=>~c)=> ~(~e&~c));	(g||(~e&~c))	YES: 2

Table 2: Test result for Truth Table in Generic Knowledge Base Form

Tell	Ask	Result
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p2&p1&p3=>d; p1&p3 => c; a; b; p2;	d	YES: a, b, p2, p3, p1, d
x => y; y => z; z => x; x&y&z => w; a; c; f; x;	w	YES: a, c, f, x, y, z, w
c&b =>e; a =>e; b&d =>e; a&d =>c; a&d =>e; a&b&d =>e; a&b&d =>c; a;	c	NO

Table 3: Test result for Forward Chaining in Horn Form

Tell	Ask	Result
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p2&p1&p3=>d; p1&p3 => c; a; b; p2;	d	YES: p2, p3, p1, d
x => y; y => z; z => x; x&y&z => w; a; c; f; x;	w	YES: x, y, z, w
c&b =>e; a =>e; b&d =>e; a&d =>c; a&d =>e; a&b&d =>e; a&b&d =>c; a;	c	NO

Table 4: Test result for Backward Chaining in Horn Form

Algorithm	Knowledge Base	Average Result (ms, 50 cases, 6 times)
Truth Table	Horn Form	18.9ms
Truth Table	Generic Knowledge	21.2ms
Forward Chaining	Horn Form	3.1ms
Backward Chaining	Horn Form	2.8ms

Table 5: Execution time for each algorithm in 50 test cases

Looking into the given tables, which illustrate the performance and results of different propositional logic algorithms - Truth Table, Forward Chaining and Backward Chaining on both Horn Form and General Knowledge Base Form, it could be clearly seen that the Truth Table method was tested in both Horn and Generic Knowledge Base Form show an average execution times of 18.9ms and 21.2 ms with 50 test cases and running six times. On the other hand, Forward Chaining and Backward Chaining demonstrated significantly lower execution times, which are 3.1ms and 2.8ms respectively (50 test cases, repeat 6 times). These results show the efficiency of Forward and Backward Chaining over the Truth Table method in processing propositional logic. Moreover, the test results for specific queries show successful inferences and points out which algorithms can efficiently handle complex logical structures.

Features / Bugs

1. Features

Features that have been implemented:

- The program has arguments for users to input the text file that contains the queries in the format of "TELL" and "ASK", and their preferred method (Figure 1).
- Three inference algorithms have been implemented: Truth Table, Forward Chaining and Backward Chaining. To improve the ease of understanding, comments are placed line-by-line to provide valuable and easy-to-understand insights into the implementation procedures.
- Two converters have been created in order to convert queries into right form, which are: Horn Form and Generic Knowledge Base.
- In addition, this program also includes a custom @export decoration, which helps determine which object is exportable and usable by the global code (outside Components folder).
- A comprehensive test case generator code has been created, helping generate different test cases in different Knowledge Base Form.

2. Bugs

Throughout the process of implementation, we have encountered lots of errors regarding the logic of the code. For example, Jordan has used the wrong data structure to implement the HornForm class (must use a set to avoid duplications) , which results in the wrong result. In addition, in the class BackwardChaining, Simon did not add the logic to avoid circular dependencies, thus resulting in RecursiveError in many different test cases. However, our team has managed to overcome the errors, and the program has produced the right result.

Team Summary Report

Truth table	Jordan Ardley - 100 %
Forward chaining	Xuan Tuan Minh Nguyen - 100%
Backward chaining	Xuan Tuan Minh Nguyen - 100%
HornForm	Jordan Ardley - 100%
LogicalSentence	Jordan Ardley - 100%
Knowledge Base	Jordan Ardley and Xuan Tuan Minh Nguyen - 40% and 60%

ReadFile	Xuan Tuan Minh Nguyen - 100%
Report	Jordan Ardley and Xuan Tuan Minh Nguyen - 50% and 50%
Overall Contribution	Xuan Tuan Minh Nguyen: 55% Jordan Ardley: 45%

Table 6: Summary task of each member

Our team primarily communicated with each other via Discord, where each team member discussed each other's idea, planned the implementation strategies for the project and distributed the workloads. Moreover, GitHub is used as the main platform where each team member contributes their own workloads into the GitHub to let other team members update the changes and point out any bugs or suggestions regarding newest changes. Throughout the project, both team members were satisfied with each other and their contributions to the project throughout the assignment.

Conclusion

Throughout this report, we have analyzed three most-known inference algorithms in propositional logic, which are Truth Table Checking, Forward Chaining and Backward Chaining algorithms. Each of them has different characteristics: Although Truth Table Checking is comprehensive, due to its exponential time and space complexities, this algorithm is impractical for large datasets. Forward Chaining on the other hand, is efficient for processing large vectors of facts incrementally due to its non-exponential time and space complexities. Backward Chaining is known with high efficiency on focusing only on relevant parts of the knowledge base, thus has smaller time and space complexities. Our testing results also showed that Backward Chaining was the most effective algorithm, with an outstanding average execution time of 2.8ms. However, in our opinion, it could be optimizable by using more optimized data structures for better memory management and we also suggest implementing heuristics to reduce the search space.

Acknowledgment / Resources

The first article, which is written by Ikenaga, is the article that helps our team understand the mechanism of how the Truth Table Checking algorithm works and gives me the idea to implement the solution. The second article written by Garcia, Mangaba and Tanchoco, and the third article written by Poli and Langdon, give a

specific perspective on how Forward Chaining and Backward Chaining works. The last book written by Russell and Norvig is the summary of those three algorithms plus the Horn Form and Generic Knowledge Base Form.

References

[1] B. Ikenaga, “Truth Tables, Tautologies, and Logical Equivalences,” *Millersville.edu*, 2019. https://sites.millersville.edu/bikenaga/math-proof/truth-tables/truth-tables.html

[2] M. B. Garcia, J. B. Mangaba and C. C. Tanchoco, "Virtual Dietitian: A Nutrition Knowledge-Based System Using Forward Chaining Algorithm," *2021 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, Zallaq, Bahrain, 2021, pp. 309-314, doi: 10.1109/3ICT53449.2021.9581887.

[3] R. Poli and W. B. Langdon, “Backward-chaining evolutionary algorithms,” *Artificial Intelligence*, vol. 170, no. 11, pp. 953–982, Aug. 2006, doi: https://doi.org/10.1016/j.artint.2006.04.003.

[4] Stuart Russell, Peter Norvig: Artificial Intelligence: A Modern Approach (4th Edition). Pearson 2020, ISBN 9780134610993.