COS30019

ASSIGNMENT 2
INFERENCE ENGINE

NGUYEN TUAN DUC - 104189440
NGUYEN GIA BINH - 104219428

APRIL 8TH, 2024

SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30019

ASSIGNMENT 2 – Inference Engine

(DUE 04/08/24 - 23:59P.M)

NAME & STUDENTID	**: Nguyen Gia Binh – 104219428** **Nguyen Tuan Duc – 104189440**
CLASS	**COS30019 - INTRODUCTION TO AI**
LECTURER	**: PHAM THI KIM DUNG (dtpham@swin.edu.vn)**
TUTOR	**: NGUYEN THE PHUC (pnguyen2@swin.edu.au)**

Hanoi, April 8, 2024

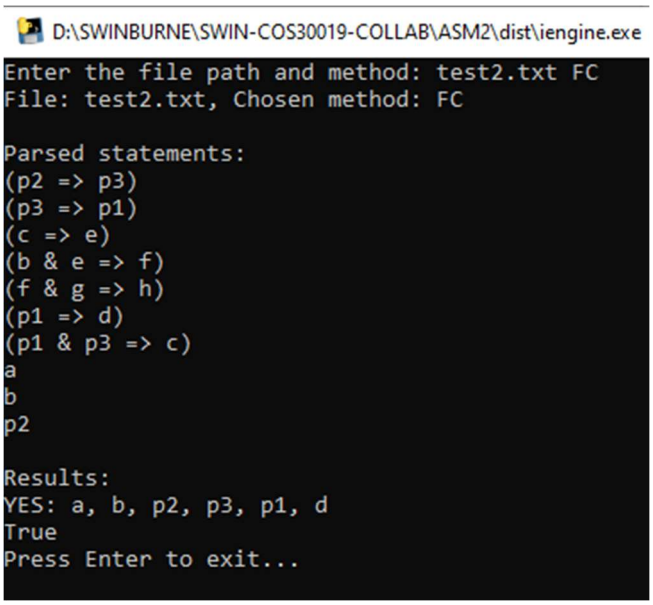
Table of Contents

A. INTRODUCTION

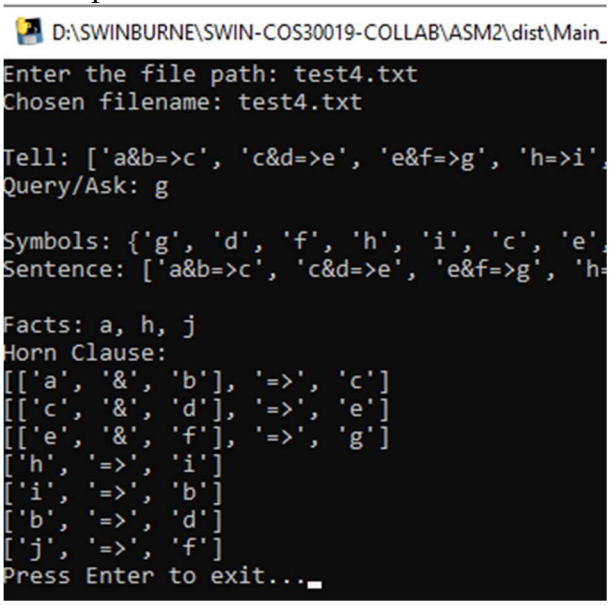
In our project, we aimed to develop an inference engine for propositional logic, utilizing algorithms such as Truth Table (TT) checking, along with Backward Chaining (BC) and Forward Chaining (FC). This engine processes inputs from a Horn-form Knowledge Base (KB) and a specific query (q), which is a proposition symbol, to assess whether q can be logically deduced from the provided KB. Moreover, we crafted a detailed report outlining the functionality of our program across various knowledge bases and queries. Notably, our development process employed Python as the primary programming language, enabling efficient implementation and testing of the inference engine.

B. INSTRUCTUON GUIDE

To get the main iengine running, please follow these step:
Step 1: download ASM2.zip file on your computer
Step 2: Locate folder “dist” and open it
Step 3: Run “iengine.exe” to test the result gfgfgf
Step 4: When prompt, enter *test file* *method* and hit enter
Example:



To get the main research program running, please follow these step:
Step 1: download ASM2.zip file on your computer
Step 2: Locate folder “dist” and open it
Step 3: Run “Main_research.exe” to test the result
Step 4: When prompt, enter *test file* and hit enter
Example:



C. FEATURES/ BUGS/ MISSING

1. Backward Chaining

The BC component of our inference engine is designed to determine if a given query, represented as a proposition symbol, can be entailed from a Horn-form Knowledge Base (KB). This process is integral to the functionality of our engine, particularly for scenarios where forward reasoning may not be as efficient or applicable. The implementation details are encapsulated within a Python class named BackwardChaining, which operates by recursively exploring the knowledge base to derive the query from available implications and facts.

Overview of Implementation

a. Module Integration:

The BackwardChaining class relies on an external Parser module for interpreting the structure of the knowledge base and queries. This modular approach facilitates a clean separation of concerns, where the parsing logic is decoupled from the reasoning mechanism.

b. Core Functionality:

The essence of backward chaining is captured within the verify method of the class. This method is tasked with recursively verifying if the target goal (query) can be deduced from the given knowledge base. It employs a depth-first search strategy, exploring implications that could potentially lead to the goal. Two critical data structures, skipped and sequence, are utilized to track the exploration path and to prevent revisiting previously examined nodes.

Critical Observations

- Algorithmic Strategy: The partial code reviewed suggests an appropriate application of backward chaining principles. It includes logic for handling both direct assertions in the KB and those that can be inferred through implications. This strategy aligns well with the expected behavior of a BC algorithm in propositional logic inference engines.

- Error Handling and Robustness: The initial review did not reveal explicit error handling or input validation mechanisms. Ensuring the resilience of the inference engine against malformed inputs or unsupported logical constructs is crucial for its reliability and usability.

- Output and Reporting: According to the assignment requirements, the outcome of the inference process should be communicated effectively to the user, indicating not just the feasibility of deriving the query, but also detailing the sequence of deductions made, if applicable. The examination did not extend into the implementation details concerning output formatting and compliance with these specifications.

- Integration and Usability: The interface through which the backward chaining logic integrates with the overall engine and interacts with users, particularly in terms of command-line operation and batch processing capabilities, remains to be fully assessed.

2. Forward Chaining

The ForwardChaining class is crafted with a modular architecture at its core, explicitly relying on an external Parser module for the interpretation of the knowledge base (KB) and queries. This strategic design choice underscores the importance of a clean separation between the logic for parsing and the mechanisms of reasoning. By decoupling these concerns, the implementation not only enhances the clarity and maintainability of the system but also facilitates potential future expansions or modifications to the parsing logic without impacting the reasoning core.

a) Algorithmic Essence and Data Structures

At the cornerstone of the ForwardChaining class is the evaluate method, which embodies the forward chaining logic through a thoughtfully devised algorithm. This method adopts a depth-first search strategy for traversing the KB, utilizing a carefully selected set of data structures. These include result_chain for tracing the path of deductions, premise_count for keeping tabs on the prerequisites required to activate implications, and a deque populated with facts to orchestrate the exploration sequence. This algorithmic approach is designed to iteratively process and reassess facts within the KB, dynamically updating premise counts and deducing new facts in a loop until the query is successfully derived or all avenues have been explored. This method showcases a rigorous application of forward chaining principles, adeptly navigating through the complexities of the KB.

b) Error Handling and User Experience Considerations:

Notably absent in the provided overview of the ForwardChaining class is a detailed discussion on error handling and input validation mechanisms. This omission signals an area ripe for development, where introducing robust error management and validation could significantly bolster the reliability and overall user experience of the inference engine. Ensuring the system's resilience against malformed inputs and logical inconsistencies is essential for maintaining user trust and facilitating seamless interaction.

c) Output Generation and User Interaction:

The methodology for generating outputs and facilitating user interaction, particularly through the solve method, hints at an underlying recognition of the importance of user-centric feedback. While the specifics of the output generation process and the interaction interface are not exhaustively detailed in the provided snippet, the mention of these elements suggests an awareness of the need to deliver clear, comprehensible results to the user. Developing and refining these aspects could greatly enhance the usability and accessibility of the system, making it more intuitive for users to engage with the engine and interpret its deductions.

3. Truth Table

The TruthTable class is meticulously designed to generate truth tables for evaluating logical propositions. It initiates with rigorous validations to ensure that the inputs—symbols, knowledge base, and query—adhere to specific data types and structures. Symbols must be strings, the knowledge base either a Conjunction object or a list of such objects, and the query must be a string or an object equipped with an evaluate method. These validations are crucial for maintaining data integrity and preventing runtime errors, showcasing a proactive approach to error handling right from the class initialization.

a) Table Generation and Data Structures:

The core functionality of generating the truth table is encapsulated within the generate_table method. This method leverages the itertools product function to iterate over all possible truth value combinations for the given symbols. For each combination, it constructs a model—a dictionary mapping symbols to their truth values—and evaluates both the knowledge base and the query against this model. The method's design reflects a comprehensive understanding of combinatorial logic and the application of Python's generator functions to efficiently handle potentially large sets of truth values.

b) Evaluating Logical Consistency:

The check_facts and brute_force_check methods together assess the logical consistency between the knowledge base and the query. check_facts counts the number of models where both the knowledge base and the query evaluate to True, indicating scenarios where the query logically follows from the knowledge base. The brute_force_check method further verifies this consistency by performing a model check, determining if the query is entailed by the knowledge base across all models. This two-pronged approach to validation underscores the class's robust mechanism for ensuring logical coherence.

c) Output Generation and Representation:

The get_entailed_symbols method synthesizes the findings from check_facts and brute_force_check to conclude whether the query is logically entailed by the knowledge base, presenting the outcome in a succinct and interpretable format. The __str__ method, on the other hand, beautifully leverages the tabulate library to format the generated truth table into a visually appealing grid. This method not only enhances the usability of the class by providing a clear and immediate visual representation of the logical landscape but also demonstrates an advanced application of third-party libraries to improve data presentation.

4. Logical Sentence Classes and Model Checking Analysis

a) Base Class and Subclasses Initialization:

The implementation begins with a base class, Sentence, designed to serve as a foundation for various logical sentences. Its constructor and methods are structured to ensure that subclasses adhere to a consistent interface, emphasising explicit initialization of arguments and providing placeholders for essential methods like evaluate and symbols. This approach enhances code readability and enforces a uniform pattern for subclass implementations, fostering a disciplined development environment.

b) Symbol Representation and Evaluation:

The Symbol class, a direct subclass of Sentence, specialises in representing logical propositions. It extends the base class with functionality specific to individual logical symbols, including error-handling mechanisms in the evaluate method that provide informative feedback for undefined symbols in the model. This class exemplifies careful attention to detail in handling edge cases and user errors, thereby improving the robustness of the logical reasoning process.

c) Logical Connectives Implementation:

Subsequent classes like Negation, Conjunction, Disjunction, Implication, and Biconditional extend the Sentence base class to implement various logical connectives. Each class provides a unique evaluate

method that encapsulates the logic specific to its respective connective, alongside a symbols method for identifying the constituent symbols. The implementation of these classes demonstrates a nuanced understanding of logical operations and their application in propositional logic, showcasing the flexibility and power of object-oriented design in creating a versatile logical reasoning toolkit.

d) Complex Logical Constructs and Helper Methods:

Particularly in the Conjunction and Implication classes, additional methods such as conjunct_premise, conjuncts, and conjunct_conclusion offer mechanisms for dissecting complex logical statements into their constituent parts. These methods, alongside print_arg_types, provide invaluable tools for introspecting logical structures, aiding in debugging and enhancing the interpretability of logical models.

e) Model Checking and Logical Entailment:

The model_check function, along with its helper check_all, represents a sophisticated mechanism for determining if a given knowledge base logically entails a query. This function employs a recursive strategy to explore all possible truth assignments to the symbols involved, efficiently evaluating the entailment without exhaustive enumeration of all models. This optimised approach to model checking underscores the application's commitment to computational efficiency and logical precision.

5. Parser

The parser code is designed for parsing logical expressions, specifically focusing on the logical constructs of conjunction (&), disjunction (||), negation (~), implication (=>), and biconditional (<=>). We use Lark - a modern parsing library for Python, to parse these expressions based on a defined grammar and then transform the parsed results into a structured format that can be further manipulated or evaluated.

a) General operation:

By pre-defining formal grammar in Lark syntax, the parser will parser logical expressions, detailing the structure of logical operators and symbols to accommodate nested and complex statements. Upon parsing an expression, the code employs Lark Transformer mechanism to transform the parsed results into an Abstract Syntax Tree (AST), which programmatically represents the logical structure of the expression. For example, an expression like a & b would be transformed into a tree where a conjunction node has two children, a and b.

6. Missing:

The parser code operates based on a predefined grammar and does not support dynamic modifications and it also does not support additional logical operators or different expression formats that would require changes to the grammar and potentially the transformation logic.

D. TEST CASE

	BC	FC	TT
Test	NO; True	NO; True	YES: 3; True
Test2	YES: d, p3, p2, p1; True	YES: a, b, p2, p3, p1, d; True	YES: 3, True
Test3	NO; False	NO; False	NO; False
Test4	YES:c, g, a, j, b, e, f, i, d, h; True	YES:a, h, j, i, f, b, c, d, e,g; True	YES: 1; True
Test5	NO; False	NO; False	NO; False
Test6	NO; False	NO; False	NO; False

Test 1 (test.txt):

Overview: The knowledge base does not conclusively entail the query regarding ~d & (~g => ~f). This suggests either an absence of information or that the provided information does not logically lead to the query, indicating a potential gap in the KB.

Test 2 (test2.txt):

Overview: The knowledge base clearly entails the query d, with a direct chain of implications present in the KB leading to it. This reflects a strong and logically coherent structure within the KB.

Test 3 (test3.txt):

Overview: The result of the query d is negative, which suggests that the knowledge base either contains a contradiction or lacks the necessary information to derive d.

Test 4 (test4.txt):

Overview: The knowledge base successfully entails the query g. The provided content outlines a complete sequence of implications and conjunctions that logically lead to the conclusion, indicating a well-connected KB.

Test 5 (test5.txt):

Overview: The KB does not support the query ~n & p & (s || t). This indicates that the KB either does not include all the necessary logical statements to support this specific combination or contains conflicting information.

Test 6 (test6.txt):

Overview: The knowledge base does not entail the query x. The result suggests that while there is a sequence that could lead to x, there is insufficient information provided to trigger the necessary implications, or there is a logical disconnect in the KB.

E. RESEARCH

Logical Expression Classifier

1. Data Structure

At the core of the classifier functionality is the parsing tree generated by pyparsing. This structure facilitates a hierarchical representation of logical expressions, breaking them down into elemental components and their composite forms. Such a data structure is vital for dissecting the logical constructs into variables and operators, enabling detailed manipulation and analysis.

2. Parsing Strategy Implementation

An important aspect of classifier is its implementation of the infixNotation method, which allows for the parsing of expressions in infix notation. This method's application is crucial for accurately interpreting expressions where operators are positioned between operands, a common format in logical statements. The strategy enables the script to manage a broad spectrum of expressions, from simple to highly complex, showcasing its versatility in handling logical analysis tasks.

3. Strength

- Adaptability: The ability to parse a wide range of logical expressions is a significant advantage, making it suitable for various applications, including automated reasoning and complex query filtering.
- Performance Enhancement: By enabling Packrat parsing, the script achieves improved parsing performance, an essential factor for handling extensive datasets or applications requiring immediate response times.

4. Weakness

- Dependencies: The reliance on the pyparsing library could potentially hinder the code adaptability across different environments or increase the complexity of initial setup processes.
- Focused Functionality: The code primarily concentrates on parsing without integrating direct mechanisms for further analysis or classification of the parsed data. This limitation suggests the need for additional components to fully exploit the processed information.

F.TEAM SUMMARY REPORT

This report presents a comprehensive overview of task assignments within our team and evaluates the satisfaction levels of each team member regarding their assigned tasks. It aims to provide transparency in task allocation and insight into how each assignment aligns with the team members' skills, preferences, and professional development goals. By examining both the distribution of responsibilities and the feedback on task satisfaction, we aim to foster a more engaged, productive, and harmonious work environment. This analysis serves as a foundation for future improvements in task management and team collaboration.

Task Title	Assigned To	Satisfaction Level (on scale 5)	Task Complexity (on scale 5)	Overall Completion Percentage
TruthTable	Nguyen Tuan Duc	5	3	10%
Forward Chaining	Nguyen Tuan Duc	4	4	25%
Backward Chaining	Nguyen Gia Binh	4	4	25%
Logic, Parser and Reader	Nguyen Gia Binh	5	4	25%
Research	Nguyen Gia Binh	5	5	6%
	Nguyen Tuan Duc	4	3	4%
Report	Nguyen Tuan Duc	4	2	5%

Total: Nguyen Gia Binh - 104219428 => 56/100%

Nguyen Tuan Duc - 104189440 => 44/100%

G. REFERENCE

1. GfG. (2023, June 9). Recursive descent parser. GeeksforGeeks. https://www.geeksforgeeks.org/recursive-descent-parser/
2. Welcome to Lark’s documentation!□. Welcome to Lark’s documentation! - Lark documentation. (n.d.). https://lark-parser.readthedocs.io/en/latest/index.html
3. Pyparsing. PyPI. (n.d.). https://pypi.org/project/pyparsing/