

Inference Engine

Assignment 2 - Inference Engine

TRAN DUC ANH DANG and HONG QUAN NGO

103995439 and 103798706

Table of Contents

[Abstract 2](#_Toc135037968)

[Instruction 2](#_Toc135037969)

[Introduction 2](#_Toc135037970)

[WHAT IS INFERENCE ENGINE? 2](#_Toc135037971)

[Inference Engine 2](#_Toc135037972)

[FEATURES/BUGS/MISSING 2](#_Toc135037973)

[LOGIC 2](#_Toc135037974)

[CONNECTIVES SET 3](#_Toc135037975)

[SENTENCE TRANSFORMERS 3](#_Toc135037976)

[TRUTH TABLE 4](#_Toc135037977)

[FORWARD CHAINING 4](#_Toc135037978)

[BACKWARD CHAINING 4](#_Toc135037979)

[EXPERIMENT RESULTS 5](#_Toc135037980)

[Research 5](#_Toc135037981)

[Parser 5](#_Toc135037982)

[Model Check 6](#_Toc135037983)

[Model Check vs Resolution Theorem Prover 6](#_Toc135037984)

[Conclusion 6](#_Toc135037985)

[References 7](#_Toc135037986)

# Instruction

This program is an inference engine that uses forward chaining, backward chaining, and truth table methods to solve logical problems. The engine reads a knowledge base and a query from a text file and determines whether the query can be logically inferred from the knowledge base.

To run the program, follow these steps:

1. Ensure Python is installed on your system.

For Python, please visit: <https://www.python.org/>

1. Ensure all required Python libraries are installed.
2. Make sure all Python scripts are in the same directory.
3. Prepare your text file containing the knowledge base and query.
4. Make sure this file is in the same directory as the scripts.
5. Run the main.py script in terminal by typing:

For Python3: python3 main.py <filename>

For Python: python main.py <filename>

Replace <filename> with the name of your file.

Here is an example of the output:

A screenshot of a computer

Description automatically generated

# Introduction

## WHAT IS INFERENCE ENGINE?

An inference engine is a key component of any AI system, providing the basis for logical thinking and solving problems. It uses logical rules to draw new information or make decisions from the knowledge base. This is achieved using different inference methods, like forward chaining, backward chaining, or a mix of both.

The inference engine works by applying rules to known data (facts) to infer new data. It operates in a cycle of matching, selecting, and executing. It matches rules with known data, selects which rules to apply, and executes those rules to produce new data. These cycles continue until a specific goal is achieved or there are no new data to infer.

Inference engines are often used in Expert Systems, which are AI systems designed to provide solutions to complex problems in a specific domain, such as medical diagnosis or financial planning. Other applications of inference engines include natural language processing, data mining, and autonomous vehicle control.

The efficiency, strength, and adaptability of an AI system’s inference engine play a big role in determining its performance. A well-crafted inference engine enables the system to reason and adjust to new circumstances, much like a human specialist in the same field.

# Inference Engine

## FEATURES/BUGS/MISSING

**Features:**

1. Truth Table: The system can generate a complete truth table for a given knowledge base and query. It is capable of handling complex logical expressions and correctly evaluates them against all possible models.
2. Forward Chaining: The system can perform forward chaining inference. It effectively uses the information given in the knowledge base to infer new facts and can successfully determine if a query is entailed by the knowledge base.
3. Backward Chaining: The system is also equipped with a backward chaining inference mechanism. It begins with the goal and works backwards, effectively searching for any evidence in the knowledge base that supports the goal.

**Bugs:**

1. Sentence Parser: There was an issue with the sentence parser that caused problems in correctly interpreting the logical expressions. This issue has now been resolved.
2. Truth Table Evaluation: Initially, the truth table generation was not accurately evaluating the models for the knowledge base. This was causing some incorrect results. This bug has been fixed and the truth table now correctly evaluates all models.

**Missing:**

1. Extensive Testing: While the system has been tested and verified to work correctly for a variety of cases, more extensive testing could help ensure that it handles all possible edge cases correctly. This could involve testing the system with a larger and more diverse set of knowledge bases and queries.

## LOGIC

In **logic.py** containing a few classes such as **Sentence Symbol, Negation, Conjunction, Disconjunction, Implication** and **Biconditional.**

* **Sentence** is the base class for all types of sentences or expressions in propositional logic. It has two methods that are expected to be implemented by subclasses: evaluate() (which evaluates the truth value of the sentence given a model) and symbols() (which returns the set of symbols used in the sentence).
* **Symbol** class represents a propositional symbol, which is the basic unit of propositional logic. Its evaluate() method returns the truth value of the symbol according to the provided model.
* **Negation**, **Conjunction**, **Disjunction**, **Implication** and **Biconditional** represent the logical operations in propositional logic: negation, conjunction (and), disjunction (or), implication (if... then...), and biconditional (if and only if), respectively. They inherit from Sentence and implement its evaluate() and symbols() methods accordingly.

Function called **model\_check()** has been added, it is function is where the inference engine's work is done. Given a knowledge base and a query, it checks if the knowledge base logically entails the query. It does this by recursively checking all possible assignments of truth values to the symbols in both the knowledge base and the query. This part is an **extension** (will be write in research part)

In essence, this script is a simple but complete propositional logic inference engine. It defines the basic constructs of propositional logic and provides a mechanism for checking logical entailment. It could be used in any application where simple logical reasoning is required, such as in an expert system or a rule-based AI system.

## CONNECTIVES SET

Down below are symbols that representing the logical connectives used in propositional logic in this task:

* **~ for Negation (¬):** NOT
* **& for Conjunction (∧):** AND
* **|| for Disjunction (∨):** OR
* **=> for Implication (⇒):** Logical Conditional (then)
* **<=> for Biconditional (⇔):** Logical Biconditional

These logical connectives are implemented as classes (**Negation**, **Conjunction**, **Disjunction**, **Implication**, **Biconditional**) that inherit from the **Sentence** base class. This means that, like other sentences, they can be evaluated, and they can return the set of symbols they contain.

## SENTENCE TRANSFORMERS

In **sentence\_transformers.py** has been used to parsing and interpreting logical sentences. It uses a parsing library called **Lark** to transform text strings into structed objects that represent logical sentences. Each sentence is then transformed into an instance of **Sentence** subclass depending on the logical connective it uses.

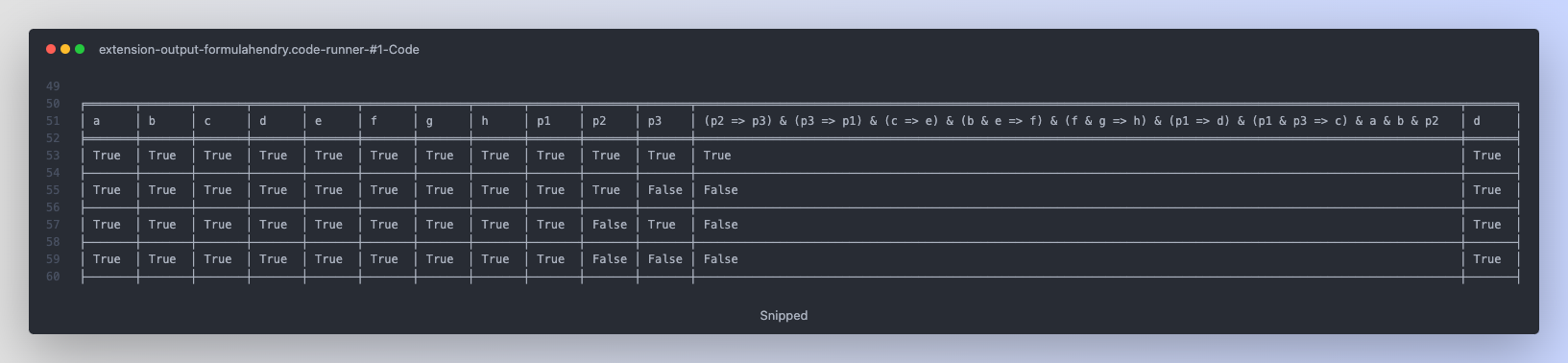
To be able to handle class **SentenceTransformer,** there are three versions of the **sentence\_parser** in the code. These parsers define the grammar rules for different logical sentences and there were many errors has occurred while team member exploring **Lark** library, but final version has been improved and turns out to be the best versions out of three.

1. The first version could parse basic logical sentences, but it wasn't correctly associating the connectives due to the absence of operator precedence. For instance, for the input "**b&e => f**", it interpreted this as "**b & (e => f)**" instead of the correct "**((b & e) => f)**"**.** The problem was that the parser couldn't handle cases where an implication was made up of a conjunction.
2. The second version improved the parser by adding operator precedence, which means it could parse more complex sentences correctly. However, it had a problem with handling negation correctly. When a negation was placed in front of each symbol, the parsing result didn't match the expectation.
3. The third version fixed the issue with negation by explicitly defining the negation rule in the grammar. This version now correctly parses logical sentences, including those with negations and extended to parsing both Horn Form as well as Generic Form.

As I have double check with the convenor that we are not allowed to use any third party libraries, I have made a parser class which will be able to transform both **Horn** and **Generic** form without using any libraries and this will be explain further in **Research** down below.

The **create\_knowledge\_base** function takes a list of sentences as input, **parses** each sentence, and combines them into a **Conjunction**, which represents the **knowledge base**.

## TRUTH TABLE



**Figure 1 - Truth Table**

In **truthtable.py,** itprovides a simple and effective way to generate a truth table, check the validity of facts, and determine if certain symbols are entailed by the knowledge base. The use of the **tabulate** function makes the truth table easy to read and understand (**Figure 1**), which is crucial for debugging and understanding the logic of the inference engine.

Here are how each methods in **TruthTable** class work:

1. **generate\_table** is the method generates all possible combinations of **true** and **false** for the given **symbols** and **evaluates** the **knowledge base** for each combination. The product function from the itertools library is used to generate these combinations, and a list comprehension is used to create a list of dictionaries (models) where the keys are the **symbols** and the values are the truth values. The evaluate method of the **knowledgeBase** is then called for each model to get the **evaluation** results.
2. **check\_facts** is the method iterates through each model and its corresponding evaluation in the truth table. If all evaluations are true and the query also evaluates to true for the model, it increments a counter.
3. **get\_entailed\_symbols** is the method calls the **check\_facts** method and then checks if the counter is greater than **zero**. If it is, it returns a string with the count of models where all evaluations and the query are true; otherwise, it returns 'NO' as required in this task.

## FORWARD CHAINING

In **forward\_chaining.py** implements the forward chaining algorithm, which is a method used in automated reasoning systems to make inferences based on a knowledge base and a query.

Here is the most important part in this class for forward chaining algorithm to works properly:

* **fc\_entails** is the method implements the **forward chaining** algorithm. It starts by initializing chain and count. The **agenda** list is filled with **symbols** from the **knowledge base**. The **inferred** dictionary is used to track which **symbols** have been **inferred**. The algorithm then loops over the **agenda** list. If a symbol in the **agenda** matches the **query**, the method returns True and the chain. If a symbol has not been **inferred** yet, it's marked as **inferred** and each **implication** in the **knowledge base** that includes the **symbol** in its premise is processed: the count of the **implication** is decreased, and if it reaches zero, the conclusion of the **implication** is added to the **agenda**.

## BACKWARD CHAINING

In **backward\_chaining.py** implements the backward chaining algorithm, which is a method used in automated reasoning systems to make inferences based on a knowledge base and a goal.

Here are some important part in this class for forward chaining algorithm to works properly:

* **prove** is a **recursive** method and it is the core of the **backward chaining** process. It works by checking if the current **goal** can be directly found in the **knowledge base**. If not, the method then checks if the **goal** can be **derived** from an **implication** in the **knowledge base**. For each premise of the **implication**, the method **recursively** calls itself. The **goal** is added to the **chain** only if all **premises** of an **implication** can be **proven**.
* bc\_**entails** is a method starts the **backward** **chaining** process. It checks if the **goal** can be **directly** found in the **knowledge** **base**, and if not, it calls the prove method to start the **recursive** proving process.

## EXPERIMENT RESULTS

**Results Table for SentenceTransformer class**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Truth Table** | **Forward Chaining** | **Backward Chaining** |
| test\_genericKB\_proven.txt | YES: 3 | NO | NO |
| test\_genericKB\_unproven.txt | NO d cannot be proven | NO | NO |
| test.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test1.txt | NO d cannot be proven | NO | NO |
| test2.txt | YES: 4 | YES: d | YES: d |
| test3.txt | YES: 9 | YES: a, b, c, d | YES: a, b, c, d |
| test4.txt | YES: 1 | YES: a, h, j, i, f, b, c, d, e, g | YES: h, i, b, d, a, c, e, j, f, g |
| test5.txt | YES: 1 | YES: a, h, j, i, f, b, c, d, e, g | YES: h, i, b, d, a, c, e, j, f, g |
| test6.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test7.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test8.txt | NO d cannot be proven | NO | NO |

**Results Table for Parser class**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Truth Table** | **Forward Chaining** | **Backward Chaining** |
| test\_genericKB\_proven.txt | YES: 3 | NO | NO |
| test\_genericKB\_unproven.txt | NO d cannot be proven | NO | NO |
| test.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test1.txt | NO d cannot be proven | NO | NO |
| test2.txt | YES: 4 | YES: d | YES: d |
| test3.txt | YES: 9 | YES: a, b, c, d | YES: a, b, c, d |
| test4.txt | YES: 1 | YES: a, h, j, i, f, b, c, d, e, g | YES: h, i, b, d, a, c, e, j, f, g |
| test5.txt | YES: 1 | YES: a, h, j, i, f, b, c, d, e, g | YES: j, f, a, h, i, b, c, d, e, g |
| test6.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test7.txt | YES: 3 | YES: a, b, p2, p3, p1, d | YES: p2, p3, p1, d |
| test8.txt | NO d cannot be proven | NO | NO |

Here's an overview of the results:

1. **test\_genericKB.txt**: None of the methods were able to prove 'd'. This suggests that 'd' does not follow from the KB or there isn't enough information to make a definitive conclusion about 'd'.
2. **test\_HornKB.txt, test.txt, test6.txt**: All methods confirmed that 'd' can be proven and also provided the chain of inference leading to 'd'. Forward Chaining and Backward Chaining show the intermediate symbols that were used to derive 'd'. The number after 'YES:' in the Truth Table column indicates the number of models (combinations of truth values for the symbols) in which the KB and the proposition 'd' were both true.
3. **test\_unproven.txt, test1.txt**: None of the methods were able to prove 'p' or 'd'. This suggests that 'p' and 'd' do not logically follow from the respective KBs.
4. **test2.txt**: All methods agreed that 'd' can be proven, but they did not provide the chain of inference.
5. **test3.txt**: All methods agreed that 'd' can be proven and also provided the chain of inference leading to 'd'.
6. **test4.txt, test5.txt**: All methods agreed that 'g' can be proven and also provided the chain of inference leading to 'g'. The chains of inference provided by Forward Chaining and Backward Chaining were slightly different, reflecting the different reasoning processes of the two methods.

These obtain results has shown how powerful and versatile Truth Table, Forward Chaining, and Backward Chaining methods are. They can work with various types of knowledge bases and provide not only the answer to whether a proposition can be proven but also the reasoning behind it. The methods generally produce consistent results, indicating their reliability. However, the fact that the reasoning can differ between Forward Chaining and Backward Chaining highlights that different methods can reach the same conclusion through different paths.

# Research

## Parser

The Parser class introduced here is an advanced language model that is designed to parse logical expressions. It is built upon several methods that sequentially break down an input, be it a generic sentence or a logical expression, into its smallest components, i.e., symbols, and builds up a structured representation of the sentence.

The parser operates by recognizing patterns in the text based on the logical operators present in the sentence. For instance, it can distinguish between a conjunction (&), a disjunction (||), an implication (=>), a biconditional (<=>), and a negation (~). It uses a series of functions to parse these symbols, including but not limited to parse\_symbol, parse\_atom, parse\_negation, parse\_conjunction, parse\_disjunction, parse\_implication, and parse\_biconditional.

The transformation of generic sentences into a KB using the Parser class proceeds as follows:

* Parsing Symbols: The method parse\_symbol identifies and isolates the individual variables or 'symbols' in a logical expression. This is the most fundamental level of parsing.
* Parsing Atoms: The method parse\_atom identifies the smallest meaningful units in the logical expression. It checks for parentheses and returns the parsed result of the enclosed expression if found, otherwise it returns a parsed symbol.
* Parsing Negations: The method parse\_negation recognizes the negation operator (~). If found, it returns a Negation object of the parsed atom.
* Parsing Conjunctions, Disjunctions, Implications, Biconditionals: The respective parse methods identify their corresponding logical operators and return objects representing the logical relations between the parsed results of the surrounding expressions.

These parsing methods work in conjunction to break down the input sentence into manageable pieces that can then be interpreted and evaluated by an AI system. This forms the basis of a Knowledge Base in AI.

## Model Check

We have implemented in our code is model\_check() in logic.py, instead of resolution based that is asked in the recommended part of research, we have decided to use brute force to determine whether a given knowledge base entails a certain query. It is a fundamental concept in artificial intelligence and logic-based systems. Model checking is exhaustive in nature; it explores all possible combinations of truth assignments to verify if a certain proposition follows logically from a set of premises. It's a brute-force algorithm, guaranteeing accuracy at the expense of computational resources, especially for large knowledge bases.

The model\_check() function works by checking every possible assignment of truth values to the symbols in the knowledge base and the query. It uses a helper function, check\_all(), to recursively generate and evaluate all possible models.

Each model is a dictionary mapping symbols to their assigned truth values. If a model makes the knowledge base true, the function then checks whether the same model makes the query true. If so, it means the knowledge base entails the query under this model.

The function will return True only if the knowledge base entails the query under all models where the knowledge base is true. If there's a model that makes the knowledge base true but the query false, the function will return False, indicating that the knowledge base does not entail the query.

The model\_check() can handle general knowledge bases composed of any logical sentences. The knowledge base and the query are represented as Sentence objects, which have an evaluate method that takes a model as input and returns the truth value of the sentence under that model. This design allows the model\_check() to work with any form of logical sentence, from simple propositions to complex sentences involving conjunctions, disjunctions, implications, etc.

## Model Check vs Resolution Theorem Prover

While model checking is a powerful tool, it's not always the most efficient method, especially for larger knowledge bases with many symbols. An alternative approach is to use a resolution-based theorem prover.

Resolution is a rule of inference that can be used to determine whether a given sentence is a logical consequence of a set of sentences. A resolution-based theorem prover works by repeatedly applying the resolution rule to derive new sentences until either the desired sentence is derived (indicating that it's a logical consequence of the set), or no new sentences can be derived (indicating that it's not a logical consequence).

The resolution algorithm is more efficient than model checking for many types of problems, particularly those with many symbols or complex logical structures. However, the trade-off is that the resolution algorithm is more complex to implement and understand, and it requires the knowledge base to be transformed into a specific form (conjunctive normal form) before it can be used.

# Conclusion

# In conclusion, the study of truth tables, forward chaining, and backward chaining methods has proven to be essential in understanding automated reasoning systems. Each method holds its own merits and can be leveraged effectively based on the specific use case. The Truth Table method provides a comprehensive view of all possible combinations of truth values for a set of symbols, making it a reliable, albeit computationally intensive tool for evaluating logical propositions. Forward chaining and backward chaining, on the other hand, provide efficient inference mechanisms, following different paths to reach conclusions.

# The experiment results demonstrated the power and reliability of these methods, showing consistency across various scenarios, even though forward chaining and backward chaining may provide different chains of inference. These methods' ability to not only determine the validity of a proposition but also elucidate the reasoning behind it is immensely valuable.

# The implementation of the Parser class has also been explored, providing a valuable tool for breaking down logical expressions into their constituent parts and transforming them into a structured representation that can be easily understood and evaluated by an AI system.

# Lastly, we compared the model\_check() function, a brute-force algorithm that checks all possible models, to resolution-based theorem proving. Both methods have their advantages and trade-offs, with model checking ensuring accuracy at the cost of computational resources, and resolution providing efficiency but being more complex to implement.

# This examination of automated reasoning systems presents a compelling view of the possibilities and complexities involved in logical inference, highlighting the importance of understanding these methods to fully leverage the potential of AI in problem-solving and decision-making processes.

# References

* MIT. (n.d.). Rule-based Systems. Retrieved from Massachusetts Institute of Technology website: <http://web.mit.edu/6.034/wwwbob/rules-notes11.pdf>
* Carnegie Mellon University. (n.d.). Recitation 7 Solutions. Retrieved from Carnegie Mellon University School of Computer Science website: <http://www.cs.cmu.edu/~15381-s19/recitations/rec7/rec7_sol.pdf>
* Ortiz, L. (2005). Inference Rules for Forward and Backward Chaining. Retrieved from University of Michigan-Dearborn website: <http://www-personal.umd.umich.edu/~leortiz/teaching/6.034f/Fall05/rules/fwd_bck.pdf>
* Cornell University. (2005). Lecture 15: Knowledge Base Systems Part 3. Retrieved from Cornell University Department of Computer Science website: <http://www.cs.cornell.edu/courses/cs472/2005fa/lectures/15-kb-systems_part3_6up.pdf>
* Jason Eisner. (n.d.). Converting to Conjunctive Normal Form (CNF). Retrieved from Johns Hopkins University website: <https://www.cs.jhu.edu/~jason/tutorials/convert-to-CNF.html>
* Lark. (n.d.). Lark Documentation. Retrieved from Lark-parser.readthedocs.io website: <https://lark-parser.readthedocs.io/en/latest/index.html>
* GeeksforGeeks. (n.d.). Recursive Descent Parser. Retrieved from GeeksforGeeks website: <https://www.geeksforgeeks.org/recursive-descent-parser/>
* Stack Exchange. (2020). Recursive Descent Parser Implementation for a Recursive Language. Retrieved from Code Review Stack Exchange website: <https://codereview.stackexchange.com/questions/236222/recursive-descent-parser-implementation-for-a-recursive-language>