Assignment 2: Inference Engine

COS30019 - Introduction to Artificial Intelligence

Bui Minh Thuan - 104486358 & Nguyen Trung Thinh - 104777544

Glossary

❖ **Inference Engine:** A system that applies logical rules to a knowledge base to derive conclusions or solve queries.

❖ **Knowledge Base (KB):** A collection of logical statements (facts and rules) used as input for inference algorithms.

❖ **Query (q):** The proposition or goal that the inference engine attempts to prove using the given knowledge base.

❖ **Entailment:** The logical relationship where a set of propositions (knowledge base) implies another proposition (query).

❖ **Horn Clause:** A clause in propositional logic with at most one positive literal, commonly used for efficient inference in logic systems.

❖ **Truth Table:** A tabular representation of all possible truth values for a given set of propositions, used to evaluate logical expressions.

❖ **Forward Chaining (FC):** A data-driven inference method that starts with known facts and applies rules to derive new facts until a goal is reached.

❖ **Backward Chaining (BC):** A goal-driven inference method that starts with a target conclusion and works backward to verify whether the known facts support it.

❖ **Resolution:** A rule of inference used to derive new clauses by resolving conflicting literals, often used in propositional logic and SAT solvers.

❖ **CNF (Conjunctive Normal Form):** A standardized logical format where expressions are written as a conjunction of disjunctions, simplifying computational processing.

❖ **DPLL Algorithm:** A backtracking-based search algorithm for solving the satisfiability problem for propositional logic in CNF.

1. Introduction

The objective of this assignment is to design and implement an inference engine for propositional logic. This engine serves as a tool to determine logical entailment using three core inference algorithms: Truth Table (TT) checking, Forward Chaining (FC), and Backward Chaining (BC). The implementation supports both Horn-form knowledge bases, which simplify logical representation, and, at an extended level, generic propositional logic clauses.

The inference engine is structured to process knowledge bases defined in a standardized text format, enabling queries to be tested systematically via a command-line interface. The functionality of the engine is validated through rigorous testing across various scenarios, ensuring its accuracy and efficiency in deriving logical conclusions. In addition to the core implementation, this assignment provides opportunities for extended research, including exploring rèfutation methods and extended Horn-form clauses.

This report outlines the foundational concepts of propositional logic and its inference methods, discusses the implementation details of the engine, and evaluates its performance through comprehensive testing. The inclusion of insights from extended research enhances the depth of analysis, culminating in a reflection on the strengths and limitations of the chosen approaches.

2. Instructions

The program for this assignment operates in **Command-line Interface (CLI) mode**, which is a mandatory requirement. Below are the details on how to execute the program in CLI mode:

```
python iengine.py <filename> <method>
```

The program is executed using the iengine.py script, which serves as the main Python file. To run the program, you need to specify the .txt file containing the layout of the given map in place of filename. Additionally, you must indicate the inference algorithm to use by providing the appropriate method name, ensuring it matches the corresponding keyword from the table below:

Algorithms	Keywords
Truth Table	TT
Forward Chaining	FC
Backward Chaining	BC
Resolution	RES
Davis–Putnam–Logemann–Loveland	DPLL

Keywords Table

Example: To execute the Truth Table (TT) algorithm using the input file test_genericKB.txt, the command would be:

```
python iengine.py test_genericKB.txt TT
```

3. Inference Algorithm

3.1. Truth Table

In an AI inference logic system, a truth table is a foundational tool for evaluating the logical relationships between propositions. By listing all possible truth values for given statements, truth tables allow the system to deduce conclusions and assess logical consistency, making them essential in AI for tasks like decision-making, knowledge representation, and automated reasoning.

A truth table systematically evaluates compound statements using basic logical operations such as AND, OR, NOT, and IMPLIES, mapping all possible combinations of truth values. Each row represents a unique truth-value combination, enabling a clear view of whether a statement holds under all conditions. This structure is particularly valuable for assessing logical validity and identifying contradictions, supporting reliable inferences.

Though effective for smaller systems, truth tables become impractical as the number of propositions grows exponentially. For larger, complex AI applications, alternative methods are often preferred. Nevertheless, truth tables provide a straightforward, essential foundation for understanding logical reasoning in AI, serving as a stepping stone to more advanced inference techniques.

3.2. Forward chaining

In an inference logic system, forward chaining is a reasoning strategy that starts with known facts and applies rules to infer new facts, working forward from the initial data. The system explores new knowledge by evaluating rules in a sequential manner, applying them to the current set of facts to deduce further conclusions.

In a forward chaining system, facts are added to a knowledge base as they are derived, and the inference engine continues this process until a goal is reached or no new facts can be inferred. This makes forward chaining particularly useful in situations where the goal is not immediately known, and the system needs to explore possible facts and outcomes based on existing information.

Similar to how a search algorithm might explore different paths, forward chaining applies rules to the facts it knows, expanding the knowledge base step by step. Unlike backward chaining, which works backward from the goal to find facts that support it, forward chaining builds up knowledge progressively from the facts at hand. It does not assume the goal exists at the start—instead, it works by applying rules and facts until it eventually reaches a conclusion.

Forward chaining relies on a process called data-driven reasoning, where the system focuses on systematically expanding the facts using available rules. This makes forward chaining a useful tool when dealing with dynamic and evolving information, such as in automated decision-making systems, diagnostic tools, or expert systems. However, one limitation is the potential for unnecessary exploration, especially in large knowledge bases, which can slow down the reasoning process.

3.3. Backward chaining

In an inference logic system, backward chaining is a reasoning strategy that works backward from a goal or hypothesis to determine the facts that support it. Unlike forward chaining, which

starts with known facts and applies rules to infer new facts, backward chaining begins with a goal and works backward to find which facts must be true for that goal to be satisfied.

In a backward chaining system, the reasoning starts from a specific hypothesis or conclusion (i.e., the goal) and tries to find the facts or rules that can justify it. If a rule's conclusion matches the goal, the system checks whether the premises of the rule are true. If they are not, the system recursively attempts to prove these premises by using other rules or facts. This process continues until the goal is either proven true or no further steps can be taken.

One key difference between backward chaining and forward chaining is that backward chaining is goal-directed. It operates in reverse, starting from a goal and asking "What do I need to prove this?" Whereas forward chaining works by expanding known facts and applying rules to build up to a conclusion, backward chaining works by decomposing a goal into smaller sub-goals and verifying each one.

In conclusion, backward chaining is a powerful approach for systems that need to deduce solutions based on specific goals or hypotheses, such as in expert systems, diagnostic systems, or theorem proving. It's particularly effective in situations where the goal is well-defined and the reasoning process is focused on validating specific outcomes.

3.4. Time Complexity and Space Complexity

The performance of each inference algorithm can be assessed based on their time and space complexities. The table below provides a detailed overview of the Big-O complexities for each algorithm:

Algorithm	**Time Complexity**	**Space Complexity**
TT	$O(2^n)$	$O(2^n)$
FC	$O(r \cdot p)$	$O(r + p)$
BC	$O(r \cdot d)$	$O(d)$

Big-0 Complexity

4. Algorithm Implementation

This section discusses how we construct the text parser - a very important part of the text parser as the main tasks of this assignment all focus on text manipulation. We also consider why we need to convert input clauses into "Conjunctive Normal Form" (CNF) and how to do this, before going into details how the core algorithms are implemented.

4.1. Parser

In the assignment helper module, it's suggested to construct a Binary Tree or apply an Infix/Postfix Parser to represent the logical clause for both knowledge base and query sentence. However, the main problem is that none of these satisfy all algorithms. For example, Binary Tree can work well with the Truth Table algorithm, but applying it to Forward Chaining or Backward Chaining leads to unnecessary complexity. Moreover, in the research section, we choose to

implement 2 extended algorithms that only work with input in CNF form, so trying to build 1 single parser is not a suitable approach.

But, of course, we really don't want and don't need to create a parser for each algorithm. The solution here is to turn the parser into a static helper class. That means, instead of letting the parser parse the input, each algorithm will do it itself, but with the helper functions from the helper parser. By doing so, we can avoid repeating the same code for functions that are used by multiple algorithms, but still ensure each algorithm is able to perform parsing in its own way.

4.2. CNF Converter

As mentioned above, the inference engine built for this assignment should be able to process both Horn-form clause and generic clause. Everything is fine working with these forms when they are short and simple, but imagine when we need to process a very large clause with extremely complicated structure, that's a completely different story, not to mention that some algorithms can't work with these forms in general.

$$(((A \lor B) \oplus (C \Rightarrow D)) \Leftrightarrow (E \lor F)) \Rightarrow (((G \oplus H) \lor (I \Leftrightarrow J)) \Rightarrow (K \land L))$$

A very complicated clause

Lucky for us, there is a way we can turn all clauses, no matter if it's Horn-form or generic form, into 1 single format. This format is called "Conjunctive Normal Form", or "CNF" for short, which is basically a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals, which is either a variable or its negation. By doing so, we standardize the overall format of all clauses, making them easier to work with. Note that "easier" here doesn't mean shortening the clause, in fact, most of the time, the CNF version is way longer than the original version. However, because its structure is consistent, logical operation on it will be much more simple.

CNF version of above clause. We can see how horrible it is visually

The process to convert a normal clause into CNF form includes 4 following steps:

1. **Eliminate Biconditional:**
   First, all biconditionals need to be removed as CNF only accepts conjunction or disjunction, based on the following formula:

$$P \iff Q \equiv (\neg P \lor Q) \land (\neg Q \lor P)$$

Biconditional Elimination

2. **Eliminate Implication:**
   Next, similar to biconditionals, we also eliminate implications from given clause, follow this rule:

$$P \implies Q \equiv \neg P \lor Q$$

Implication Elimination

3. **Move Negation Inwards:**
   The third step is to move all negation signs inwards, as we don't want any negated sub clause in our CNF sentence.

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

Move Negation Inwards

4. **Distribute Or Over And:**
   Finally, we perform the last step to distribute or OR sign over AND clause. This is the last step before we get a valid CNF sentence. Distribute OR over AND needs to follow the following formula:

$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

Distribute OR over AND

After we finish all these steps, we will get the desired output as a clause in CNF form. Final version should be something like this example:

$$(L_{1,1} \vee L_{1,2} \vee \cdots \vee L_{1,n}) \wedge (L_{2,1} \vee L_{2,2} \vee \cdots \vee L_{2,m}) \wedge \ldots$$

Valid CNF form clause

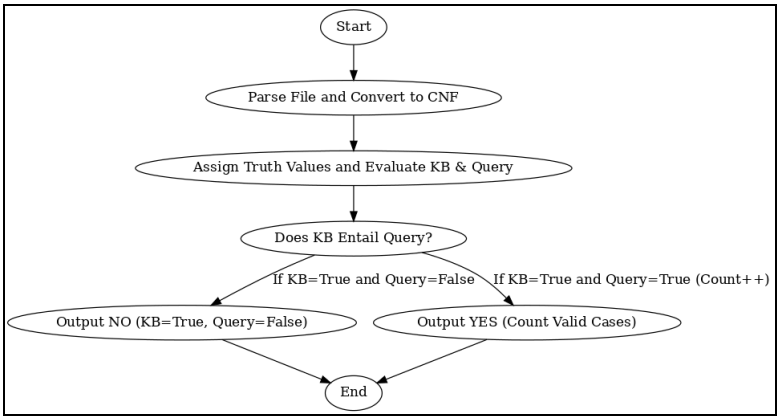
4.2. Algorithm Implementation

4.2.1 Truth Table algorithm (TT)

In the algorithm theory section, we understand that TT basically means trying all possible cases to check whether with the given Knowledge Base, we can entail the query or not. If the inference engine only needs to process Horn-form clauses, the implementation can be relatively easy, just like building a bunch of calculations with AND and OR operations. However, when it comes to generic clauses, the story is different: beside AND and OR, we now have IMPLICATION and BICONDITIONAL, and we also need to consider the parenthesis, which can affect the order of calculation. In this case, applying the same technique as Horn-form based engines like Binary Tree or Infix/Postfix is not recommended due to unnecessary complexity.

For this problem, our solution is to turn all sentences and clauses into CNF form. As mentioned earlier, one obvious advantage of CNF is consistent structure. Each CNF form sentence is conjunction of disjunction, that means, if we want to know whether a clause is TRUE with the given literal values, just need to check if all disjunction is TRUE, which equals to at least of literal inside this disjunction is TRUE, and just that, no matter how complex the original clause is, no matter how many literals or parenthesis order we need to consider.

Since we are required to use CNF for the two algorithms in the extended section, it makes sense to fully commit to this method without considering alternatives. CNF not only simplifies processing but also provides a standardized format that is universally compatible with inference engines. By converting everything to CNF, we reduce the complexity of handling different logical structures, allowing us to focus on implementing the core algorithms efficiently. Although CNF conversion introduces some overhead, its benefits in reducing runtime complexity during logical evaluations outweigh the initial cost, especially for large-scale problems.

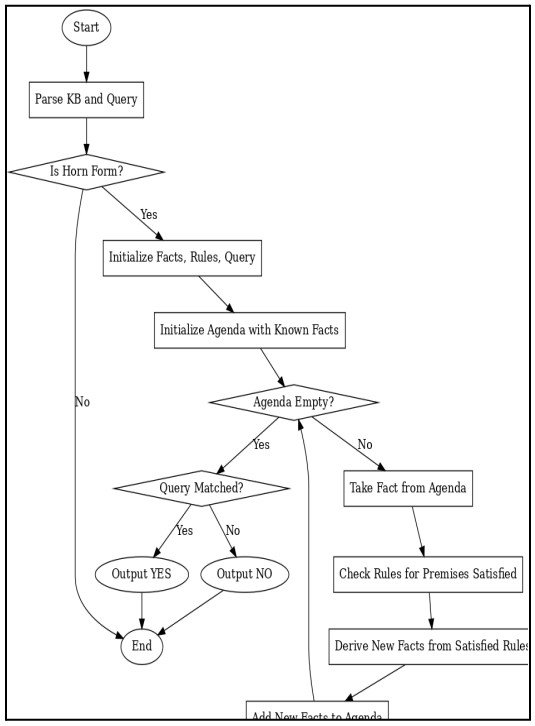


Truth Table Flow Chart

4.2.2 Forward Chaining algorithm

The **ForwardChaining** implementation parses a knowledge base (KB) into **facts**, **rules**, and a **query**. Facts are known truths, rules are Horn-form implications (**premises => conclusion**), and the query is the statement to verify. After ensuring the KB is in Horn form, the algorithm initializes an **agenda** with known facts.

The inference process dequeues facts from the agenda, checks them against the query, and adds them to an **entailments** list. If a fact matches the query, it outputs "YES" with the entailments. Otherwise, it evaluates rules, adding conclusions to the agenda if their premises are satisfied. This continues until the agenda is empty, outputting "NO" if the query isn't derived. By using efficient structures like a deque, dictionary, and set, this **ForwardChaining** implementation is both systematic and scalable, ideal for expert systems and rule-based reasoning. The provided flowchart mirrors these steps clearly.



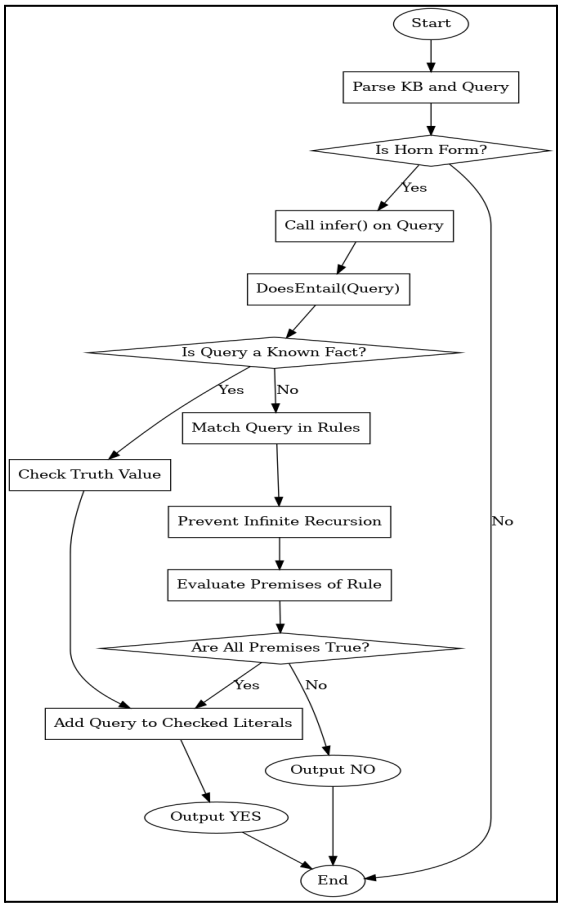
Forward Chaining Flow Chart

4.2.3 Backward Chaining algorithm

The implementation of the backward chaining algorithm in the provided Python file follows a structured process to determine if a query can be inferred from a knowledge base (KB). The **BackwardChaining** class initializes by parsing the KB and query from an input file. Facts are stored with their truth values, while rules are represented as implications with premises and conclusions. The KB is validated to ensure it is in Horn form; otherwise, the algorithm halts.

The core inference logic lies in the **infer** method, which calls **DoesEntail** to determine if the query is supported by the KB. **DoesEntail** delegates to the recursive **TruthValue** method, which evaluates whether a literal is a known fact or can be derived from matching rules. For each matching rule, all premises are recursively checked. A literal is considered true only if all premises of a rule are true. Preventing infinite recursion is managed through a tracking set **(prevent_infinite)**, ensuring the algorithm does not revisit literals currently under evaluation.

The algorithm transparently tracks "checked literals," which are literals successfully evaluated during the process. When the query is entailed, the program outputs "YES" along with the list of these literals; otherwise, it outputs "NO." The recursive evaluation and rule matching directly follow the logical flow of backward chaining, ensuring correctness and efficiency while adhering to the constraints of Horn form logic and avoiding infinite loops.



Backward Chaining Flow Chart

5. Extended research

5.1 Horn extend - Disjunction form

A **Horn clause** is a special kind of logical expression used primarily in computer science and mathematical logic, particularly in the fields of logic programming and automated reasoning. Named after the logician Alfred Horn, it is a clause (a disjunction of literals) with at most one positive literal. This constraint gives Horn clauses unique properties that make them computationally efficient for inference in logical systems. In its disjunctive form, a Horn clause can be written as:

$$L_1 \vee L_2 \vee \cdots \vee L_k$$

where $L_1, L_2, \ldots, L_k$ are literals (either positive or negative). The defining feature of a Horn clause is that it contains **only one positive literal**, denoted as P. If P exists, the Horn clause is expressed as:

$$\neg Q_1 \vee \neg Q_2 \vee \cdots \vee \neg Q_n \vee P$$

Here, $Q_1, Q_2, \ldots, Q_n$ are negative literals. Alternatively, this can be written in implication form:

$$(Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n) \implies P$$

If no positive literal is present, the Horn clause represents a purely negative clause, such as:

$$\neg Q_1 \vee \neg Q_2 \vee \cdots \vee \neg Q_n$$

This corresponds to an inconsistency if all Q_i are true simultaneously. However, in Backward Chaining and Forward Chaining, a Horn clause must contain at most one positive literal. This constraint ensures that the clause can be seamlessly converted into its implication form, enabling efficient reasoning and inference during the chaining process.

5.2. Refutation algorithms

For the second extension, we will introduce alternative approaches to infer whether a given Knowledge Base can entail a Query or not. The main idea here is refutation, which means instead of trying to prove the entailment between Knowledge Base and Query, we aim to prove the negation of the query leads to a contradiction when combined with the Knowledge Base. The general idea is expressed with formula below:

$$KB \models f \iff KB \cup \{\neg f\} \text{ is unsatisfiable}$$

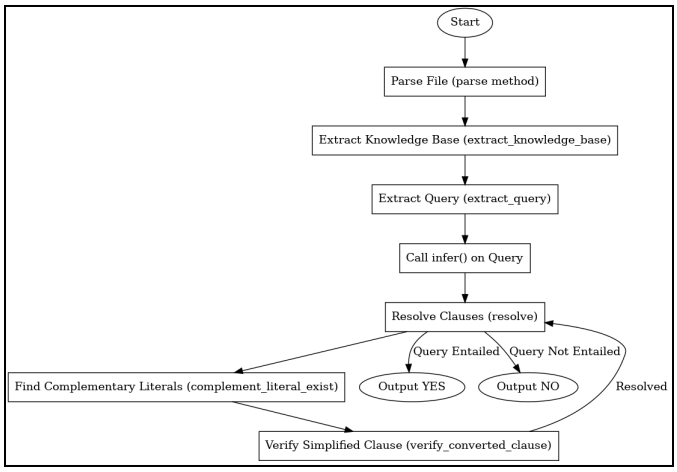
Basically, we transform the problem from checking entailment into SAT solver, verifying whether a clause is Satisfiable or not. To do this, we apply 2 algorithms that have the similar original idea but totally different logic, which are “Resolution-base Theorem” and Davis–Putnam–Logemann–Loveland *(DPLL).*

5.2.1 Resolution Theorem (TT)

Resolution is a rule of inference used to eliminate opposing literals and derive conclusions. The algorithm operates by repeatedly combining pairs of clauses that contain complementary literals—one clause containing a literal and another containing. These clauses are resolved to produce a new clause that excludes and if the resolution process leads to an empty clause, it proves that the query is entailed by the knowledge base.

$$C_1 = A \vee B,\ C_2 = \neg A \vee D \implies R = B \vee D$$

In order to perform this algorithm, all input clauses must be converted into CNF form, which is already done by the CNF Converter we discussed earlier. One key point is that we will not work with the original form of the query, but instead, we will take its negation form and convert that to CNF form. Then the algorithm is valid, and it will try to find complement literal between clauses until it can't do it anymore, or it reaches an empty clause, meaning contradiction.



Resolution Flow Chart

In our program, Resolution can work well with both Horn-form clause or Generic-form clause as input, but only if the query is simple. If the query contains multiple literals, this algorithm will not be stable, as it can't explore the contradiction when there is too much information.

5.2.2 Davis–Putnam–Logemann–Loveland *(DPLL)*

On the other hand, DPLL is a classic algorithm used to solve the Boolean satisfiability problem (SAT), which asks whether a given propositional formula can be satisfied. That is, it determines if there exists an assignment of truth values to variables such that the entire formula evaluates to true. It forms the basis for many modern SAT solvers and is widely applied in fields such as formal verification, artificial intelligence, and automated theorem proving.

The main idea of the DPLL algorithm is to systematically explore the search space of variable assignments in a propositional formula to determine if a satisfying assignment exists. What makes this algorithm more effective is that it includes some preprocessing to simplify the given clauses in order to reduce the computational required, and, in best cases, we can draw the conclusion whether a clause is Satisfiable or not right in this step. Overall, DPLL can be broken down into 3 steps:

1. **Unit propagation:**
   Unit propagation identifies unit clauses in the formula, which are clauses containing a single literal. Such clauses force the value of their literal to be true for the formula to be satisfied, therefore, we can immediately conclude the value for this literal without needing to check the whole sentence. After assigning the truth value to the unit clause's literal, we attempt to simplify the other sub clause with the following strategy:

$$F' = \{C \in F \mid l \notin C\}$$

All clauses in F that contains literal l are always True and can be removed

$$F' = \{C \in F \mid l \notin C\} \cup \{C \setminus \{\neg l\} \mid C \in F \text{ and } \neg l \in C\}$$

Complement of literal l are removed as it has no impact to final result

After this step, we can draw some conclusion about whether the clause is Satisfiable or not. If the final sentence is empty, meaning all sub clauses are removed, we can conclude that the original sentence is SAT, meaning the Knowledge Base can not entail the Query. However, if during Unit Propagation, an empty sub clause is created, that means the sentence is UNSAT, and the Knowledge Base is able to entail the given Query.

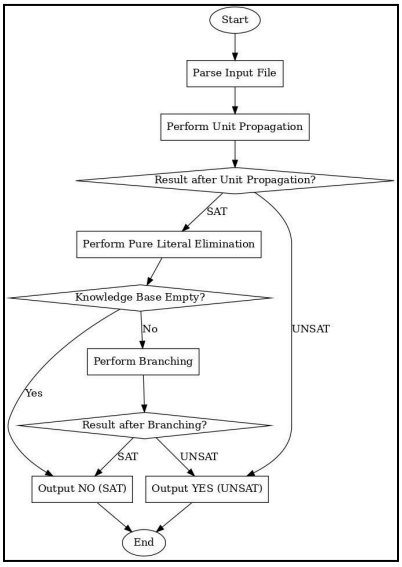
2. **Pure Literal Elimination:**
   A literal is considered "pure" if it consistently appears in the formula with only one polarity — either always as a positive literal or always as a negative literal but not both. In other words, a pure literal is never contradicted by its negation within the formula.

   Pure literals can be directly assigned a truth value that satisfies all clauses in which they appear. For example: If a literal is pure and appears positively in the formula, assigning its value as TRUE in all clauses. Similarly, if that literal is always negative, setting it as FALSE.

   This simplification step further reduces the formula by removing all clauses that contain the pure literal, as they are satisfied by the assigned value. If after we eliminate all Pure Literals, we notice the sentence is eventually empty, we can conclude the sentence is SAT and given Knowledge Base can not entail the Query.

$$F' = F \setminus \{C \in F \mid l \in C \text{ where } l \text{ is pure}\}$$

3. **Decision Making and Backtracking**
   If no more simplification is possible but the formula is not yet satisfied, choose a variable (heuristically) and assign it a truth value. This creates two branches in the search tree: one assuming the variable is true and the other assuming it is false. If the current assignment leads to an empty clause (a clause that cannot be satisfied), backtrack to the most recent decision point and try the opposite truth value for the variable. In theory, this step seems computational heavily, but in fact as the simplification processes above have significantly reduced the complexity of the whole clause, it becomes more manageable in practice.



DPLL Flow Chart

6. Testing

After implementing all the required inference algorithms, testing is essential to identify potential bugs and gain deeper insights into each algorithm. A total of 21 test cases files with 22 inline test cases have been prepared, divided into two categories, each serving distinct purposes.

6.1. Accuracy Testing (22 test cases)

22 test cases are created for accuracy testing in which I focus on testing the precision in the process of parsing input, returning the result of inference logic.

Test Case	**Test Case Description**	**Test Case Purpose**
test_findAllWords	Test word extraction from logic expressions	Ensure correct keyword identification
test_findStartParen	Test finding the start of parentheses	Ensure accurate opening parenthesis detection
test_findEndParen	Test finding the end of parentheses	Ensure accurate closing parenthesis detection
test_addParentheses	Test adding parentheses if needed	Ensure valid logical expressions
test_removeParentheses	Test removing unnecessary parentheses	Simplify logical expressions
test_extractLiterals	Test literal extraction from expressions	Ensure correct literal parsing
test_signedLiterals	Test extracting signed literals	Ensure signed literal accuracy
test_extractClauses	Test clause extraction	Ensure correct parsing of clauses
test_organizeSentence	Test organizing clauses and operators	Ensure logical structure correctness
test_reconstruct	Test clause reconstruction	Ensure proper clause rebuilding
test_isHornForm	Test if expression is in Horn Form	Ensure correct Horn Form detection
test_toImplication	Test converting OR to implication	Ensure accurate operator conversion
test_removeBicond	Test removing biconditional (<=>)	Simplify expressions
test_removeImplication	Test removing implication (=>)	Simplify expressions

test_negationInwards	Test moving negation inward	Ensure negation correctness
test_distributeLogic	Test distributing OR over AND	Ensure logical correctness
test_toCNF	Test conversion to CNF	Ensure normalization to CNF
test_truthTable	Test truth table algorithm	Ensure correct truth table results
test_forwardChain	Test Forward Chaining inference	Ensure correct algorithm behavior
test_backwardChain	Test Backward Chaining inference	Ensure correct algorithm behavior
test_resolution	Test Resolution inference	Ensure correct algorithm behavior
test_dpll	Test DPLL inference	Ensure correct algorithm behavior

Test case descriptions

These tests address critical potential issues that could arise from improper algorithm implementation. For example, careful attention is required to ensure accurate handling of logic structures, such as parentheses placement, literal extraction, and clause reconstruction, as errors in these areas could compromise the algorithm's correctness. Additionally, tests for inference methods like Forward Chaining, Backward Chaining, and DPLL prevent potential failures, such as infinite loops or incorrect results during inference. All identified issues have been resolved successfully, ensuring the algorithm's readiness for the final submission.

6.2. Performance Testing (16 test cases with 3 categories)

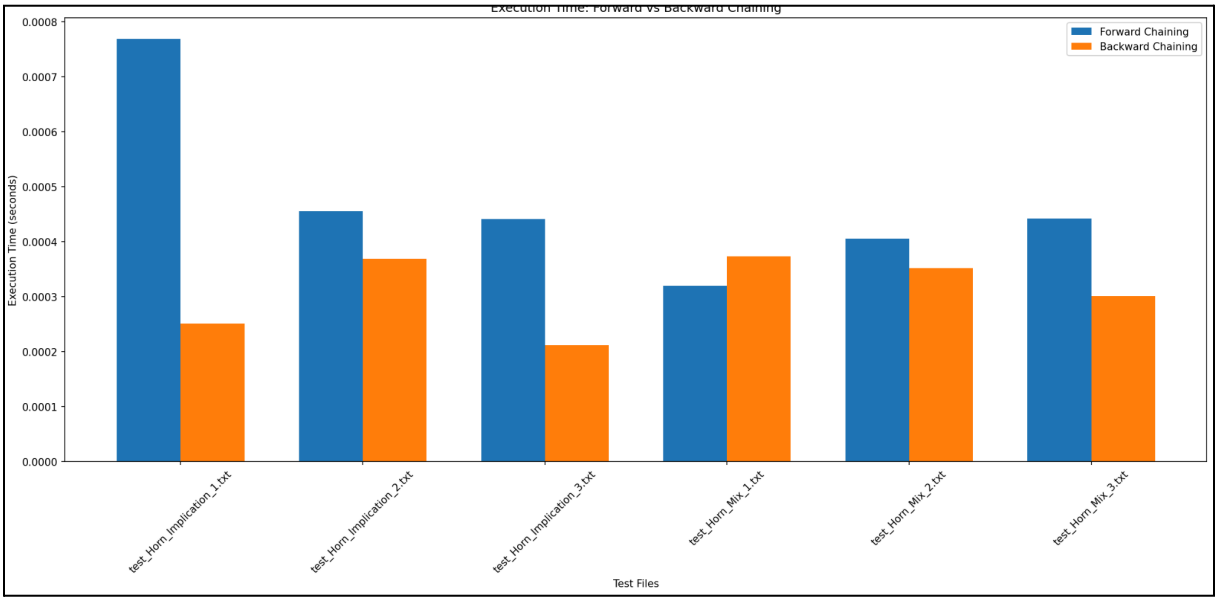
Beside those tests above, this section evaluates the runtime performance of the implemented inference algorithms using 16 test cases across three categories: **Horn-form** clauses, **generic-form** clauses, and **edge-case** scenarios. The focus is on assessing the time efficiency of each algorithm under varying problem structures and complexities.

The first test set evaluates algorithm performance on Horn-form clauses, featuring goal-oriented and data-driven problems of increasing complexity. It examines time efficiency in handling structured, rule-based knowledge bases.

Test case	Test case patterns
Test_Horn_Implication_1.txt	**Goal oriented** problem
Test_Horn_Implication_2.txt	**Simple, Direct** Problems with **Few Irrelevant** Rules
Test_Horn_Implication_3.txt	**Moderate** Goal Complexity
Test_Horn_Mix_1.txt	**Small** Rule Base with **Sparse** Data

Test_Horn_Mix_2.txt	**Data-Driven** Problems
Test_Horn_Mix_3.txt	**Mixed** Data- and **Goal-Driven** Scenarios

Test case descriptions



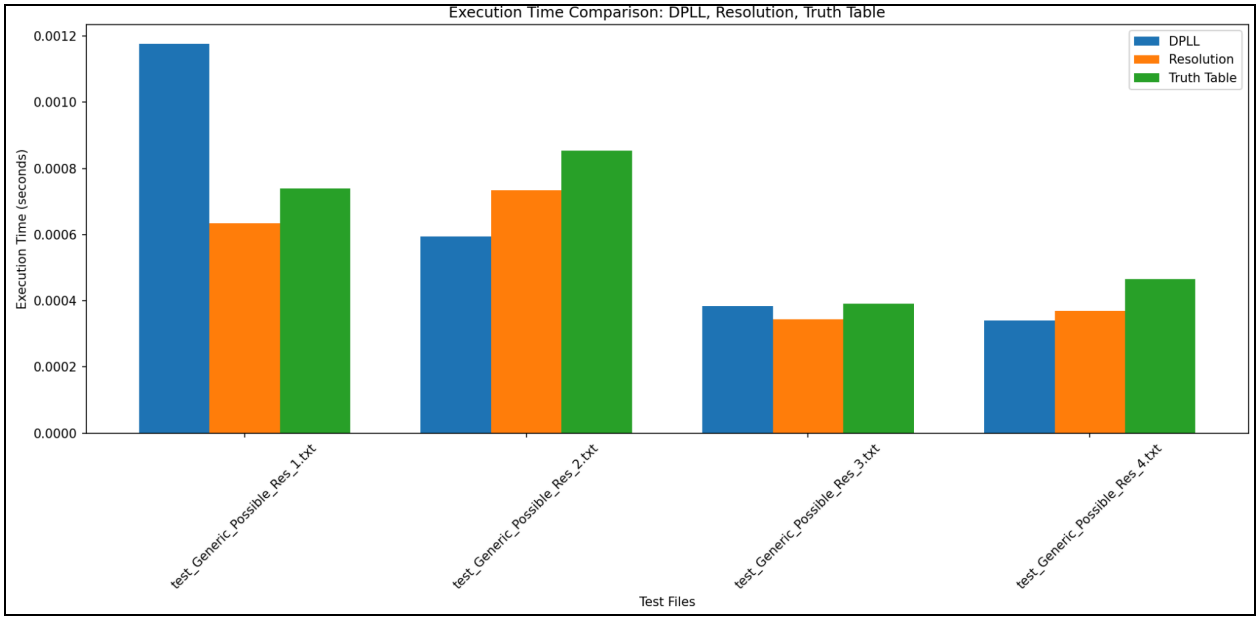
Forward Chaining and Backward Chaining Executing Time Result

These cases highlight the efficiency of Forward and Backward Chaining in goal-directed problems, while the Truth Table approach becomes less practical as the knowledge base grows. The Horn-form clauses' structured nature ensures that inference remains systematic, albeit with variations in runtime depending on the problem's complexity.

The second test set focuses on generic-form clauses with single-literal queries, evaluating algorithm robustness in handling large knowledge bases, unstructured constraints, and unsatisfiable scenarios.

Test case	Test case patterns
test_Generic_Possible_RES_1.txt	The problem has **structured constraints**
test_Generic_Possible_RES_2.txt	The problem has a **large number of variables**
test_Generic_Possible_RES_3.txt	The formula has **poor structure**
test_Generic_Possible_RES_4.txt	The formula is **unsatisfiable** and contains a **small refutation**

Test case descriptions



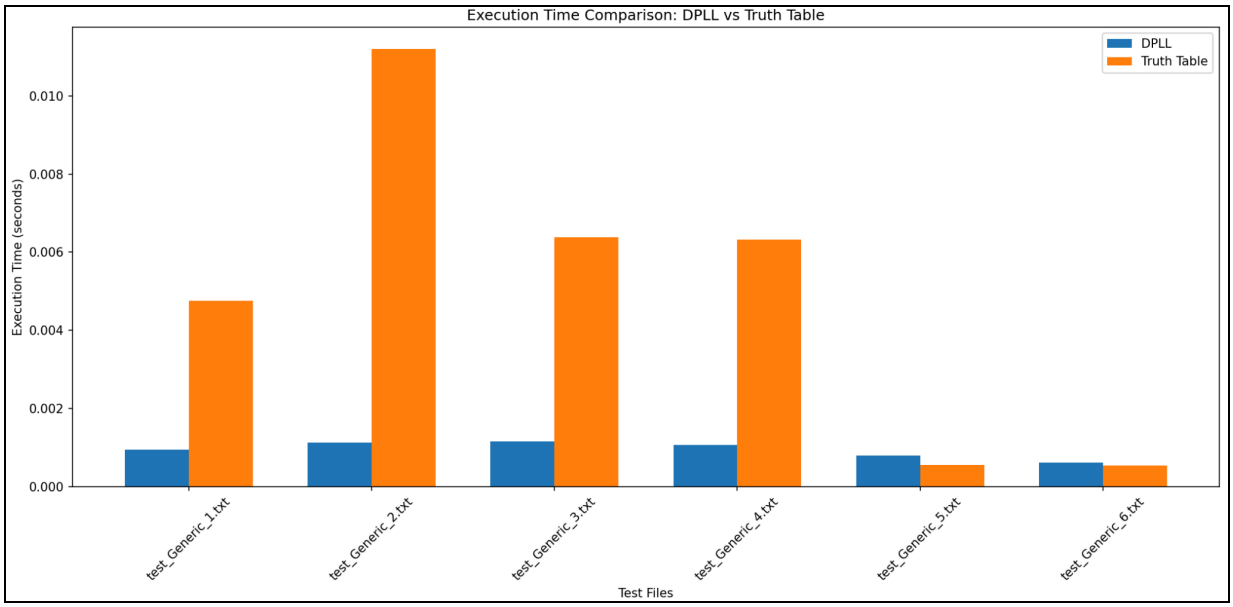
DPLL, Resolution and Truth Table Executing Time Result

The generic-form clauses introduce significant complexity, which highlights the limitations of Forward and Backward Chaining algorithms when handling poorly structured problems or larger variable sets. Conversely, refutation-based methods like Resolution and DPLL excel in these scenarios, demonstrating superior adaptability and efficiency, particularly for unsatisfiable or highly constrained problems.

The third test set examines edge cases with generic clauses, including structured patterns, random unstructured clauses, and symmetric or sparse solutions, to assess algorithm performance under irregular conditions.

Test case	Test case patterns
test_Generic_1.txt	**Highly structured** problem **(unit propagation helps)**
test_Generic_2.txt	**Large number of variables** problem
test_Generic_3.txt	**Sparse satisfiable** solutions
test_Generic_4.txt	**Early satisfiability or unsatisfiability** problem
test_Generic_5.txt	**Small-scale problem with highly symmetric** clauses
test_Generic_6.txt	**Unstructured, random** clauses

Test case descriptions



DPLL and Truth Table Executing Time Result

This set reveals the adaptability of Resolution and DPLL algorithms in edge cases. Unit propagation and pure literal elimination play a crucial role in structured problems, significantly reducing runtime. However, unstructured and random clauses pose challenges, with performance highly dependent on the search heuristics used in DPLL. These cases underscore the versatility and robustness of refutation-based methods in handling diverse logical structures.

7. Team Summary Report

This assignment is done by Group 5, including 2 members: Bui Minh Thuan and Nguyen Trung Thinh. During the working process, all works are splitted equally between us, and we also perform proof checking to ensure both get the total ideas of the final program. See work distribution table below for more details:

Task	Bui Minh Thuan	Nguyen Trung Thinh
Algorithm Implementation	60% (3/5)	40% (2/5)
Report and Documentation	50%	50%
Testing	40%	60%

8. Conclusion

Overall, performance testing highlights the strengths and limitations of the inference algorithms. Forward and Backward Chaining excel with structured Horn-form clauses but struggle with complex, generic-form scenarios. Refutation-based methods like Resolution and DPLL handle unsatisfiable cases and random structures effectively, aided by preprocessing techniques like unit propagation and heuristic decision-making. While no single algorithm is universally optimal, their combination offers a robust framework for diverse inference tasks. Future improvements could focus on hybrid approaches or better heuristics for complex problems.

References

Horn clause. (n.d.). Wikipedia. Retrieved November 24, 2024, from https://en.wikipedia.org/wiki/Horn_clause

How does DPLL Algorithm Work? A Brief Explanation of DPLL | by Mert Özlütıraş | Medium. (2021, February 18). Mert Özlütıraş. Retrieved November 24, 2024, from https://mertozlutiras.medium.com/brief-explanation-of-dpll-davis-putnam-logemann-loveland-algorithm-663f4a603c1

Huong L.T. (n.d.). *Chương 5: Biểu diễn Tri thức*. Trí Tuệ Nhân Tạo - Viện Công nghệ Thông tin và Truyền thông, Trường Đại học Bách Khoa Hà Nội. Retrieved 11 19, 2024, from https://users.soict.hust.edu.vn/huonglt/AI/Chuong%205.%20Bieu%20dien%20tri%20thuc.pdf

Jain, S. (2020, July 16). *Difference between Backward and Forward Chaining.* GeeksforGeeks. Retrieved November 24, 2024, from https://www.geeksforgeeks.org/difference-between-backward-and-forward-chaining/

Jain, S. (2024, June 17). *Forward Chaining and Backward Chaining inference in Rule-Based Systems*. GeeksforGeeks. Retrieved November 24, 2024, from https://www.geeksforgeeks.org/forward-chaining-and-backward-chaining-inference-in-rule-based-systems/