COS30019: Introduction to Artificial Intelligence

Assignment 2: Propositional Logic Inference Engine

Minh-Hieu Tran
104850021

Anh-Quan Nguyen
[YOUR STUDENT ID]

Contents

1 Instructions

1.1 Command-Line Interface (CLI)

The program is executed via a command-line interface using the `iengine.py` script. The script requires two arguments: the path to the knowledge base file and a keyword for the desired inference method.

Command Format:

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

- `<filename>`: The path to the text file containing the knowledge base and query (e.g., `test_HornKB.txt`).
- `<method>`: A keyword specifying the algorithm: `TT`, `FC`, or `BC`.

Example Usage:

```
python iengine.py test_cases/horn_multi_premise_yes.txt FC
```

The program will output "YES" or "NO" to indicate if the query is entailed by the knowledge base. For "YES" cases, additional information is provided based on the method used.

2 Introduction

This report details the design and implementation of a propositional logic inference engine, a system capable of determining if a logical conclusion can be derived from a given set of facts and rules. The engine processes a knowledge base (KB) and a query (q) to determine if the knowledge base logically entails the query, denoted as $KB \models q$.

2.1 Core Concepts

The inference engine operates on propositional logic, using a set of standard logical connectives to construct sentences:

- **Negation ($\sim$):** Represents 'not'. Example: $\sim p$ (p is false).
- **Conjunction (&):** Represents 'and'. Example: $p \ \& \ q$ (p and q are both true).
- **Disjunction (||):** Represents 'or'. Example: $p \ || \ q$ (either p is true, or q is true, or both).
- **Implication ($\Rightarrow$):** Represents 'if-then'. Example: $p \Rightarrow q$ (if p is true, then q must be true).
- **Biconditional ($\Leftrightarrow$):** Represents 'if and only if'. Example: $p \Leftrightarrow q$ (p is true if and only if q is true).

The engine implements three distinct core algorithms to perform inference: Truth Table checking, Forward Chaining, and Backward Chaining.

3 Inference Methods

The project implements three core inference algorithms from propositional logic, each providing a distinct methodology for solving the entailment problem ($KB \models q$). The choice of algorithm depends on the structure of the knowledge base and the desired trade-offs between universality and efficiency.

3.1 Truth Table (TT) Checking

The Truth Table checking algorithm embodies the most fundamental definition of logical entailment. It is a model-checking approach that is both sound and complete for any sentence in propositional logic, not just Horn clauses.

The core idea is to perform an exhaustive, brute-force enumeration of all possible worlds. A "world" or "model" is a single, complete assignment of truth values (True or False) to every propositional symbol present in the knowledge base. The algorithm iterates through every single model and performs two checks:

1. Is the entire Knowledge Base (KB) true in this model?
2. Is the query (q) true in this model?

Entailment holds if and only if there is no model in which the KB is true while the query is false. The algorithm's primary strength is its universality. However, its primary weakness is its computational complexity. For a KB with n unique symbols, there are 2^n possible models to check, making this method computationally infeasible for all but the smallest knowledge bases.

3.2 Forward Chaining (FC)

Forward Chaining is a highly efficient inference algorithm designed specifically for knowledge bases in **Horn form**. A Horn clause is a disjunction of literals with at most one positive literal, which can be expressed as an implication where the premise is a conjunction of positive symbols leading to a single positive conclusion (e.g., $A \ \& \ B \Rightarrow C$).

FC operates using a **data-driven** or **bottom-up** reasoning approach. It begins with the known facts (atomic clauses) in the KB. It then iteratively fires rules whose premises are satisfied by the set of known facts, adding the rule's conclusion to the set of known facts. This process continues, like a domino effect, until the query is proven or no new facts can be derived. Because it only explores the logical consequences of the initial facts, it is significantly more efficient than Truth Table checking for problems that fit its constraints.

3.3 Backward Chaining (BC)

Like Forward Chaining, Backward Chaining is also designed for Horn-form knowledge bases. However, it employs a **goal-driven** or **top-down** reasoning strategy.

The algorithm begins with the conclusion it is trying to prove—the query. It works backward, searching for rules in the KB that have the query as their conclusion. When it finds such a rule, it treats the premises of that rule as new sub-goals that must be proven. This process is inherently recursive, as the algorithm calls itself to prove each new sub-goal. The chain of reasoning terminates when a sub-goal is found to be an initial fact in the KB, or when all rules for a sub-goal have been exhausted without success. This goal-oriented approach can be more efficient than Forward Chaining if the query is not a consequence of many other facts, as it avoids exploring irrelevant inference paths.

4 Core Algorithm Implementation

This section provides a detailed walkthrough of the implementation of the inference engine, focusing on the system's architecture, data structures, and the specific logic of each core algorithm.

4.1 System Architecture and Parsing

The engine is built with a modular design in Python to separate concerns and enhance maintainability.

- **iengine.py:** This script serves as the main entry point and controller. Its sole responsibility is to parse command-line arguments, identify the requested file and method, and instantiate the corresponding algorithm class to begin the inference process.
- **kb.py:** The `KB` class is the primary data container. Upon initialization, it reads the specified file, uses regular expressions to separate the raw TELL and ASK strings, and then calls the parser to process these strings into tokenized lists of clauses and the query. It also extracts the set of all unique propositional symbols.
- **parser.py:** A crucial design decision was to implement the parser as a static utility class rather than a stateful object. This provides a centralized library of parsing functions that can be called by any other module without needing to create an instance. Its methods are responsible for tokenizing logical expressions from strings into lists (e.g., `"a  b => c"` becomes `['a', '&', 'b', '=>', 'c']`) and for validating that a knowledge base adheres to the constraints of Horn form, a check required by FC and BC.

4.2 CNF Converter Implementation

The `Converter` module is a core component designed to transform any propositional logic sentence into Conjunctive Normal Form (CNF). This standardization is essential for many advanced logical operations. The conversion process is implemented as a series of recursive static methods that apply logical equivalences, supported by several preprocessing helper methods.

4.2.1 Preprocessing and Helper Methods

Before the main conversion steps can be applied, several helper methods are used to analyze and prepare the tokenized expressions.

- `_find_main_connectives` and `_find_main_operator`: These functions are crucial for all recursive operations. They scan a token list from right to left, respecting parentheses, to find the main logical connective based on a defined order of precedence ($\Leftrightarrow, \Rightarrow, ||, \&$). This ensures that an expression like $A \vee B \wedge C$ is correctly parsed as $A \vee (B \wedge C)$, allowing the recursive algorithms to divide expressions correctly.
- `_wrap_in_parentheses`: This method ensures that complex sub-expressions are properly encapsulated in parentheses. This is a vital preprocessing step that guarantees the recursive logic can consistently operate on well-formed sub-problems, preventing ambiguity.
- `_flatten_associativity`: This is a final cleanup step that simplifies the CNF output. It removes redundant parentheses from associative chains (e.g., converting $(A \vee B) \vee C$ to $A \vee B \vee C$), making the final CNF easier to read and process.

4.2.2 Main Conversion Steps

The conversion process applies logical equivalences in a specific, sequential order.

1. **Eliminate Biconditional:** The first step removes all biconditional connectives ($\Leftrightarrow$) by replacing them with a conjunction of two implications, based on the logical equivalence:

$$P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

2. **Eliminate Implication:** Next, all implication connectives ($\Rightarrow$) are removed using the equivalence:

$$P \Rightarrow Q \equiv \neg P \vee Q$$

3. **Move Negation Inwards:** This step uses De Morgan's laws to push negation symbols ($\sim$) inward until they apply only to individual propositional symbols. The two key equivalences are:

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

4. **Distribute Disjunction over Conjunction:** The final step ensures the sentence is a conjunction of disjunctions by applying the distributive law:

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

4.3 Truth Table Implementation

The implementation of the Truth Table algorithm follows the theoretical definition directly and consists of two main phases: model generation and evaluation.

1. Model Generation: The `_generate_models` method is responsible for creating an exhaustive list of all possible truth assignments. To do this efficiently, it uses Python's `itertools.product` function. This function creates an iterator that yields all possible combinations of True/False for the n symbols in the knowledge base, generating a total of 2^n unique models. Each model is stored as a dictionary mapping symbols to boolean values.

2. Evaluation: The core of the evaluation is the `_evaluate` method, which determines the truth value of any given clause for a specific model. It is a recursive, divide-and-conquer function that operates as follows:

- **Base Case:** If the expression is a single symbol, it returns the symbol's truth value from the current model.
- **Recursive Step:** To handle complex expressions and respect operator precedence, the function finds the "main connective" of the expression. It does this by scanning the token list from right to left, searching for connectives in reverse order of precedence (`<=>`, then `=>`, then `||`, etc.), while also respecting parentheses. Once the main connective is found, the function recursively calls itself on the left and right sub-expressions and combines their boolean results according to the connective's logic.

The main `infer` method then iterates through every generated model. For each model, it checks if all clauses in the KB evaluate to true. If they do, it then checks if the query also evaluates to true. If a model is found where the KB is true but the query is false, entailment fails, and the process terminates. Otherwise, it counts the number of valid models.

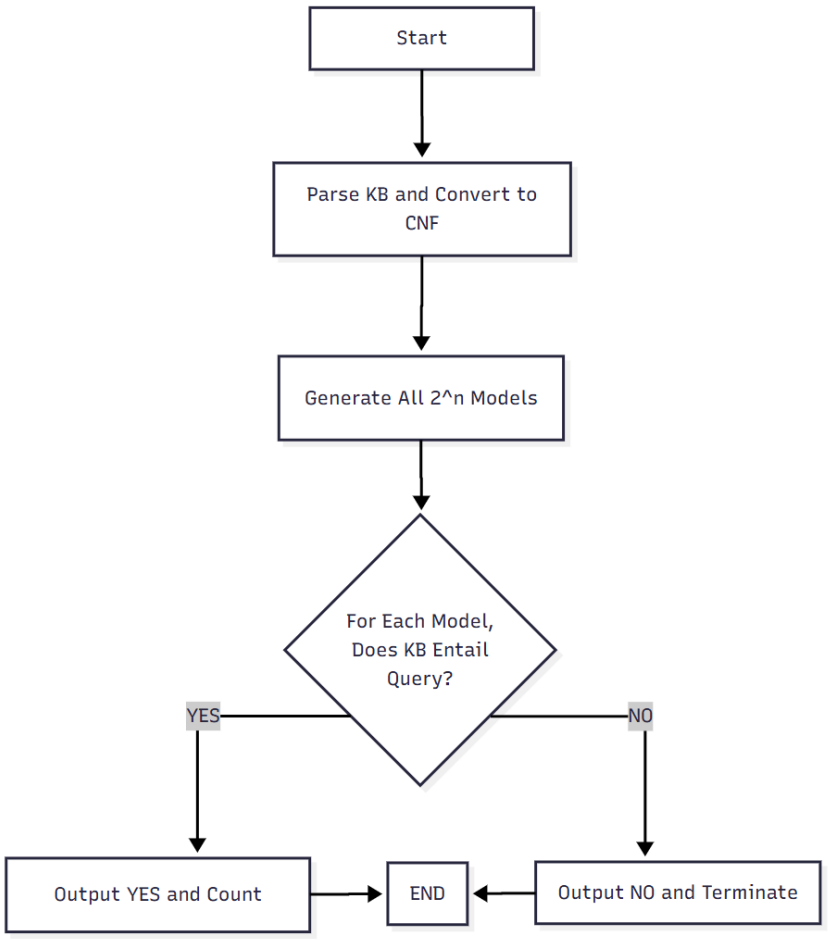

Figure 1: TruthTable Algorithm Flowchart

4.4 Forward Chaining Implementation

The Forward Chaining algorithm is implemented iteratively to avoid potential recursion depth limits and to manage the state of the inference process explicitly. The implementation relies on four key data structures chosen for their performance characteristics:

- `agenda`: A `collections.deque` object, used as a First-In-First-Out (FIFO) queue. This stores facts that have been proven but whose consequences have not yet been explored. A queue ensures a breadth-first search of the inference space.
- `inferred`: A `set` that stores all symbols that are known to be true. Using a set provides O(1) average time complexity for checking if a fact has already been proven, preventing redundant work.
- `premise_count`: A `dictionary` that maps each rule's conclusion to an integer. This integer represents the number of premises for that rule that are not yet known to be true. When this count reaches zero, the rule can "fire".
- `is_premise_of`: A `dictionary` that maps each symbol to a list of rules where that symbol is a premise. This serves as an inverted index, allowing the algorithm to efficiently find all rules that might be affected when a new fact is proven.

The process begins by parsing the KB and populating these data structures. All simple facts are added to the `agenda` and `inferred` set. Then, the main inference loop runs as long as

the `agenda` is not empty. In each iteration, a fact is taken from the agenda. The algorithm uses the `is_premise_of` dictionary to find all rules where this fact is a premise and decrements their respective `premise_count`. If a rule's count drops to zero, its conclusion is added to the `agenda` and the `inferred` set. The process terminates when the query is proven or the agenda becomes empty.

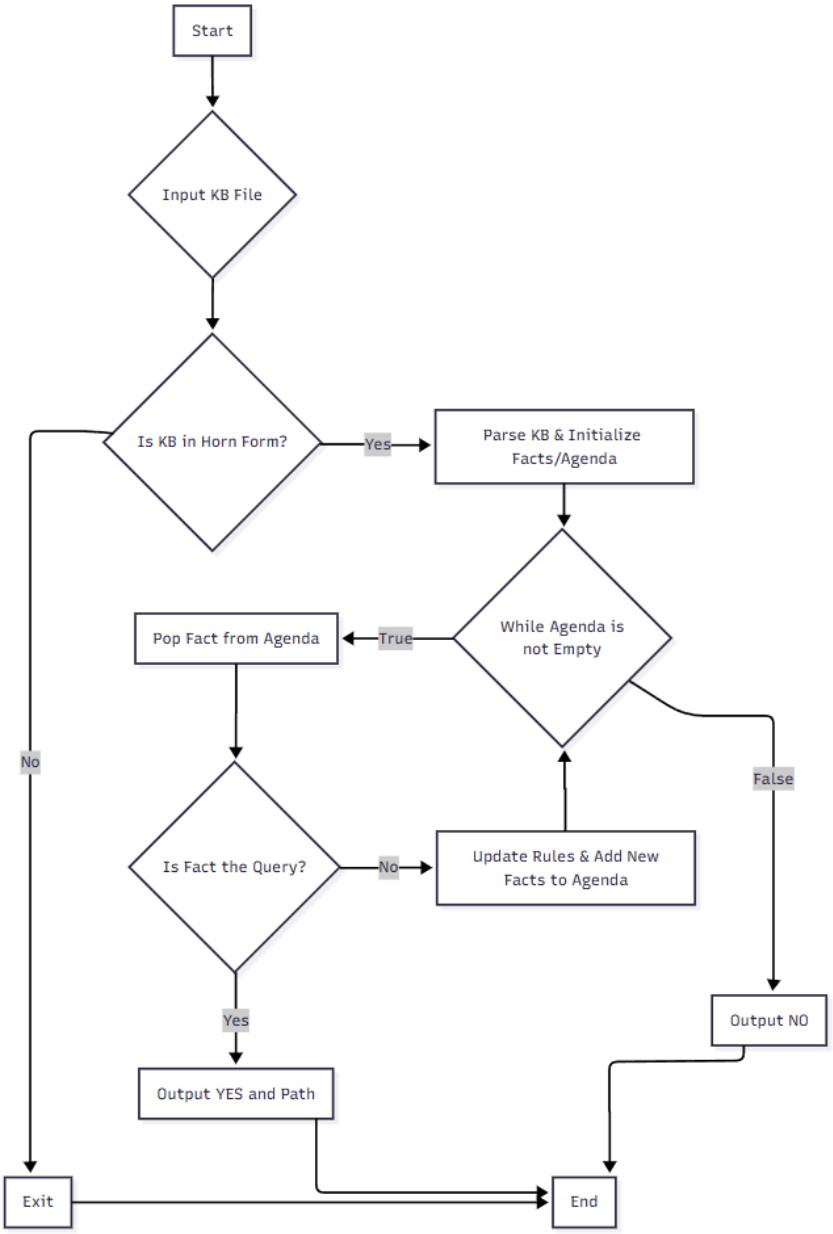

Figure 2: ForwardChaining Algorithm Flowchart

4.5 Backward Chaining Implementation

The Backward Chaining algorithm is implemented recursively, as this paradigm naturally mirrors the goal-driven, top-down reasoning process. The implementation is centered around the recursive function `_bc_entails`.

First, during initialization in `_initialize_bc`, the knowledge base is processed. Simple facts are stored in a `facts` set for fast lookups. All rules (implications) are stored in a `rules`

dictionary, where the keys are the conclusions and the values are lists of their premises. This structure allows the algorithm to instantly find all rules that can potentially prove a given goal.

The `_bc_entails(goal)` function operates as follows:

- **Base Cases:** The recursion terminates successfully if the `goal` is found in the `facts` set, or if it has already been proven and added to the current proof path.
- **Recursive Step:** If the goal is not a fact, the function looks up the goal in the `rules` dictionary. It iterates through each rule that can conclude the goal. For each rule, it recursively calls `_bc_entails` on all of the rule's premises. If all premises of a single rule can be proven, the original goal is considered proven.
- **Infinite Loop Prevention:** A critical feature of the implementation is the management of cyclic KBs (e.g., $A \Rightarrow B; B \Rightarrow A$). To prevent infinite recursion, the function uses a `recursion_stack` set that tracks all goals currently being pursued in the active recursion chain. Before attempting to prove a goal, it checks if the goal is already in this set. If it is, that path of reasoning is immediately abandoned, breaking the loop.

The main `infer` method initiates the process by calling `_bc_entails` on the main query and reports the result.

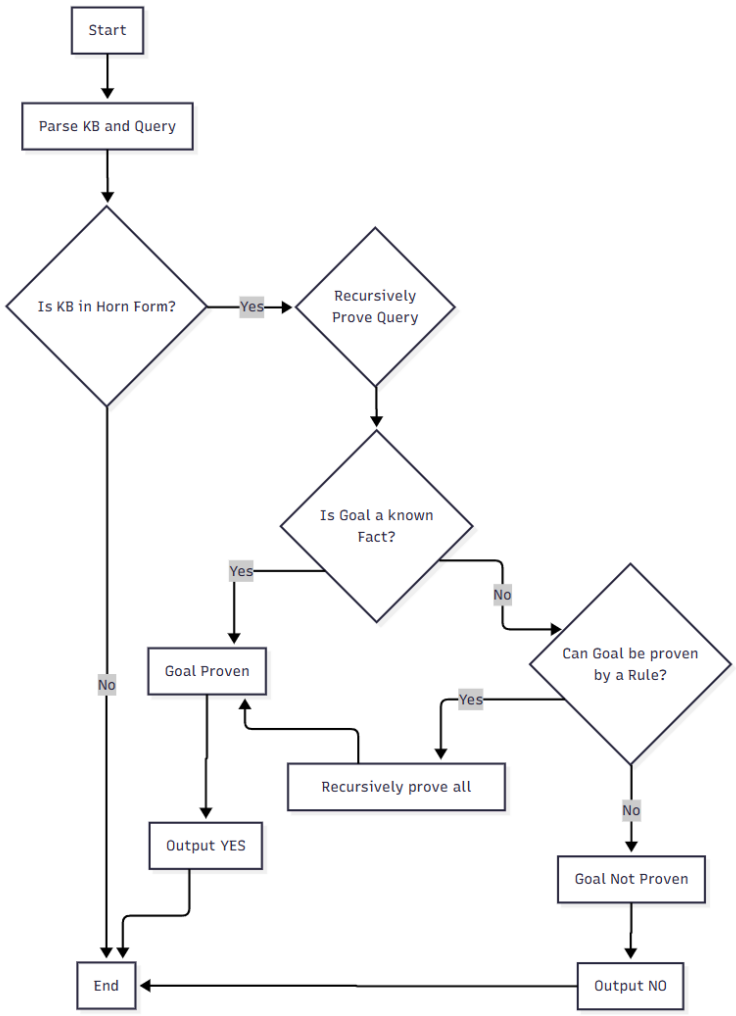

Figure 3: BackwardChaining Algorithm Flowchart

5 Extended Research: Refutation-Based Solvers

As part of the research component of the assignment, the engine's capabilities were extended to handle any generic propositional logic knowledge base using two powerful, modern algorithms: Resolution and the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Both of these methods operate on a principle known as **refutation**.

Instead of trying to prove that the knowledge base entails the query ($KB \models q$) directly, a refutation-based approach tries to prove that the opposite is impossible. It works by checking if the sentence ($KB \wedge \neg q$) is **unsatisfiable**—that is, if it leads to a logical contradiction. The fundamental equivalence is:

$$KB \models q \iff (KB \wedge \neg q) \text{ is unsatisfiable}$$

This transforms the entailment problem into a satisfiability (SAT) problem, for which highly optimized algorithms exist. Both Resolution and DPLL require the input sentences to be in Conjunctive Normal Form (CNF), making our `Converter` module an essential prerequisite.

5.1 Resolution

The Resolution algorithm is a refutation-complete inference procedure. This means that if a set of clauses is unsatisfiable, Resolution is guaranteed to eventually derive a contradiction.

5.1.1 Theoretical Basis

The algorithm is centered around a single, powerful rule of inference: the **resolution rule**. Given two clauses that contain complementary literals (e.g., one clause contains p and the other contains $\neg p$), the rule allows us to infer a new clause (the *resolvent*) that contains all literals from the two parent clauses except for the complementary pair.

$$\frac{(L_1 \vee \cdots \vee L_k \vee \alpha), \quad (\neg\alpha \vee M_1 \vee \cdots \vee M_n)}{(L_1 \vee \cdots \vee L_k \vee M_1 \vee \cdots \vee M_n)}$$

The process aims to generate the **empty clause** (a clause with no literals), which represents a direct contradiction.

5.1.2 Implementation

Our implementation of the Resolution algorithm operates as follows:

1. **Initialization:** All clauses from the knowledge base are converted to CNF using the `Converter` module. The query is negated, also converted to CNF, and its resulting clauses are added to the set of clauses from the KB. All clauses are stored as sets of string literals for efficient processing.

2. **Iterative Resolution:** The algorithm enters a loop where it systematically selects pairs of clauses from the current set. For each pair, it checks for complementary literals.

3. **Generating Resolvents:** If a complementary pair is found (e.g., `"p"` in one clause and `"p̃"` in the other), a new resolvent clause is generated by taking the union of the two parent clauses and removing the complementary pair.

4. **Checking for Contradiction:** If a resolvent is the empty set, it signifies that a contradiction has been found. The algorithm terminates and returns "YES", indicating that the original query is entailed.

5. **Termination:** If the loop completes an entire pass without generating any new, unique clauses, it means no further inferences can be made. In this case, no contradiction was found, and the algorithm terminates and returns "NO".

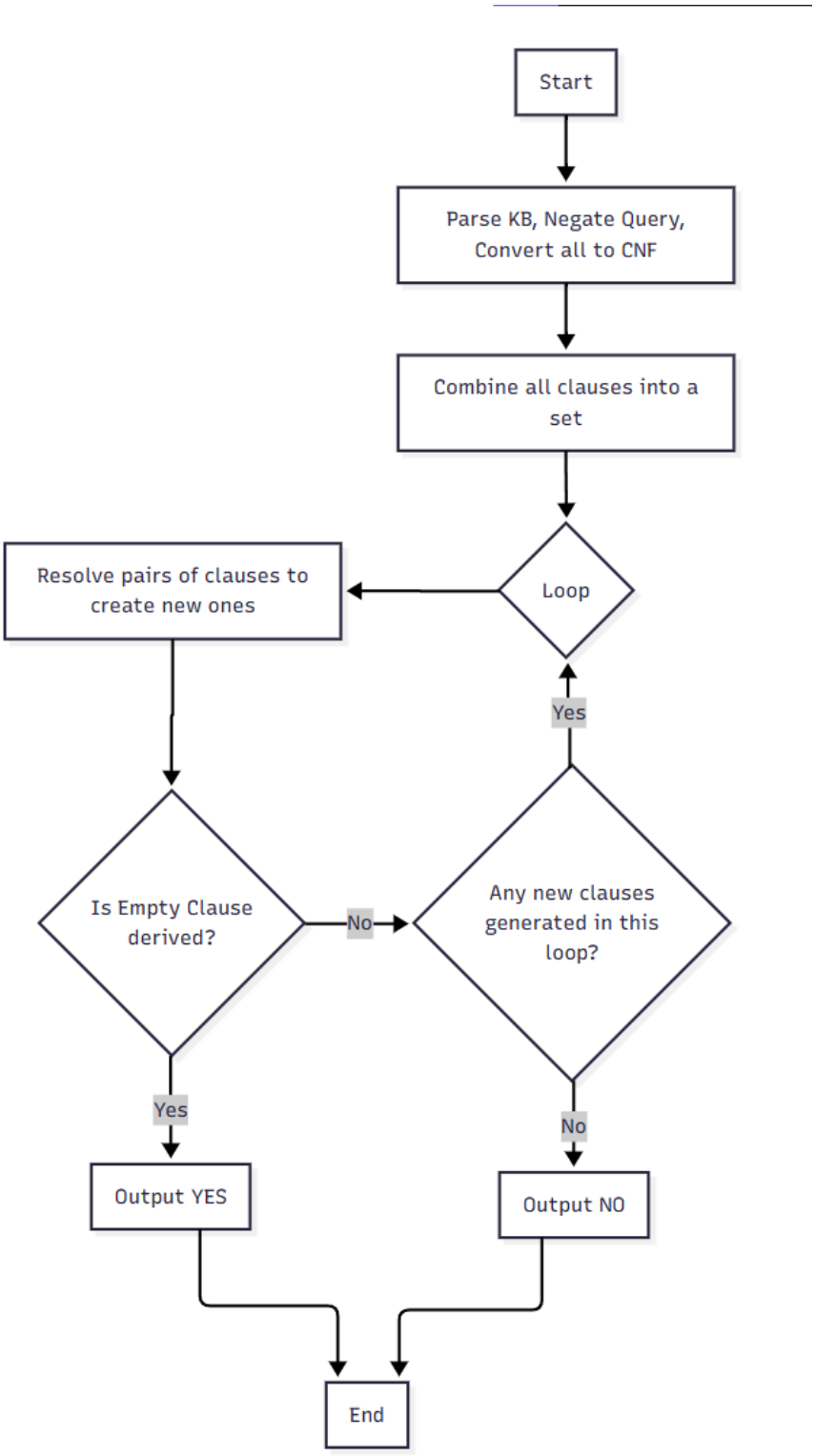

Figure 4: Resolution Algorithm Flowchart

5.2 Davis-Putnam-Logemann-Loveland (DPLL)

The DPLL algorithm is a highly efficient, backtracking-based search algorithm for solving the Boolean Satisfiability (SAT) problem. It is the foundation of most modern SAT solvers. Like Resolution, our implementation uses it to determine if the set of clauses from $(KB \wedge \neg q)$ is unsatisfiable.

5.2.1 Theoretical Basis

DPLL performs a depth-first search through the space of possible truth assignments for all symbols. However, it is much more efficient than a simple brute-force search because it uses powerful heuristics to prune large parts of the search space.

5.2.2 Implementation

Our implementation is a recursive function, `_dpll`, that attempts to find a satisfying model (a valid truth assignment).

1. **Initialization:** As with Resolution, the KB and the negated query are converted to CNF and combined into a single list of clauses.
2. **Recursive Simplification:** The core of the algorithm is a loop that repeatedly applies two powerful simplification heuristics:
   - **Unit Clause Heuristic:** The algorithm scans for any "unit clauses" (clauses with only one literal). If a unit clause like $\{p\}$ is found, the symbol p *must* be assigned True for the KB to be satisfiable. This assignment is added to the model, and the entire set of clauses is simplified accordingly (clauses containing p are removed, and $\neg p$ is removed from clauses where it appears).
   - **Pure Literal Heuristic:** The algorithm searches for "pure literals"—symbols that only appear in one form (either always positive or always negative) throughout the entire set of clauses. For example, if q appears but $\neg q$ never does, q can be safely assigned True. This assignment is added to the model, and all clauses containing the pure literal are removed.
3. **Splitting (Decision and Backtracking):** If neither heuristic can be applied, the formula cannot be simplified further. The algorithm then chooses an unassigned variable to "split" on.
   - It first assumes the variable is True and makes a recursive call to `_dpll` with this new assignment.
   - If this recursive call returns `True` (meaning a satisfying model was found), the original call also returns `True`.
   - If it returns `False`, the algorithm backtracks and makes a second recursive call, this time assuming the variable is False.
4. **Conclusion:** If the initial call to `_dpll` returns `False`, it means no satisfying assignment could be found for $(KB \wedge \neg q)$, so the formula is unsatisfiable, and the engine reports "YES". If it returns `True`, a model was found, so the query is not entailed, and the engine reports "NO".

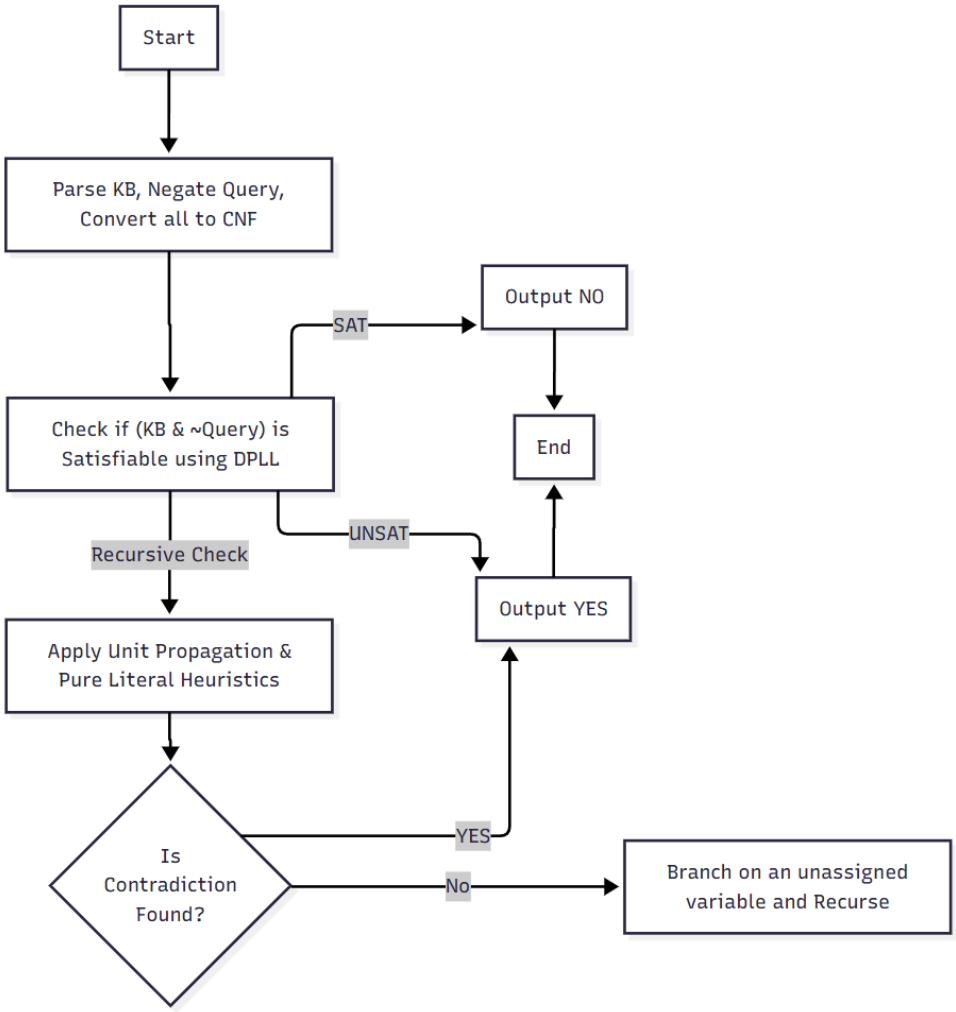

Figure 5: DPLL Algorithm Flowchart

6 Testing

A robust testing strategy was employed to validate the correctness and analyze the performance of the inference engine. This involved two distinct phases: accuracy testing to ensure logical soundness and performance benchmarking to evaluate efficiency.

6.1 Accuracy Testing

The primary goal of accuracy testing was to verify that each algorithm produced the correct logical output across a wide variety of scenarios. An automated test suite was created using Python's `unittest` framework, allowing for repeatable and systematic validation. The test cases were strategically designed to cover the following categories.

Table 1: Accuracy Test Case Design Summary

Test Case	Description	Purpose
horn_multi_premise_yes	KB has facts p1, p2, p3 and rule p1&p2&p3=¿q. Query: q.	To verify that FC and BC can correctly handle rules with multiple premises.
horn_disjunction_yes	KB has fact ‘a‘ and rule ‘ a——b‘. Query: b.	To verify that FC and BC can correctly parse and reason with Horn clauses written in their disjunctive form.
horn_missing_fact_no	KB has fact ‘a‘ and rule ‘a&b=¿c‘. Query: c.	To verify that algorithms correctly return NO when a required premise is missing.
generic_biconditional_yes	KB uses ‘a¡=¿b‘ and fact ‘a‘. Query: b.	To test the ability of TT, RES, and DPLL to handle biconditional connectives.
generic_disjunctive_yes.txt	KB contains ‘a——b‘ and ‘ a‘. Query: b.	To test a classic rule of inference for generic solvers.
generic_fallacy_no	KB has ‘p=¿q‘ and ‘q‘. Query: ‘p‘ (fallacy).	To ensure the engine does not commit common logical fallacies and correctly returns NO.
adv_horn_dead_end	KB has a rule requiring fact ‘b‘, but ‘ b‘ is a fact.	To test that algorithms correctly terminate and return NO when an inference path is a dead end.
adv_generic_tautology_query	Query is a tautology (‘a—— a‘) which is always true.	To verify that the generic solvers correctly identify universally true statements.
adv_deep_nesting_1	KB with deeply nested parentheses and many connectives.	To stress-test the robustness of the parser and the recursive evaluation logic of the TT and CNF converters.
edge_bc_cycle	KB has a logical loop: ‘a=¿b; b=¿a‘. Query: ‘a‘.	To specifically test the infinite loop prevention mechanism in the Backward Chaining implementation.
edge_contradiction_simple	KB contains a direct contradiction: ‘p; p‘.	To verify that generic solvers handle unsatisfiable knowledge bases correctly.
edge_empty_kb	The TELL section is empty.	To ensure all algorithms correctly return NO when the knowledge base is empty.
edge_fc_irrelevant_facts	KB has many facts/rules irrelevant to the query’s derivation path.	To test that the reasoning of goal-driven algorithms like BC is not affected by irrelevant information.

6.2 Performance Benchmarking

As part of the extended research, a performance analysis was conducted to empirically compare the efficiency of the implemented algorithms. This helps to validate their theoretical complexities and understand their practical strengths and weaknesses.

6.2.1 Methodology

A dedicated script, `visualizeEngine.py`, was created to automate the benchmarking process. This script uses Python's `timeit` module for accurate timing, executing each inference multiple times to get a stable average. The results are then plotted into bar charts using the `matplotlib` library for clear visual comparison. A dedicated folder, `test_cases/performance`, was created to hold KBs specifically designed to highlight the performance characteristics of each algorithm.

6.2.2 Test Case Analysis and Hypothesis

The performance tests were divided into two main comparison groups:

1. Forward Chaining vs. Backward Chaining on Horn Clauses: This comparison uses three specific KBs to highlight the differences between the data-driven and goal-driven approaches.

- `fc_favored.txt`: This KB contains a large number of initial facts but a very simple rule set that is not immediately triggered by those facts. **Hypothesis:** FC will be significantly faster because it will process the initial facts and terminate quickly, whereas BC will waste time working backward through an inference chain that is ultimately unprovable from the given facts.

- `bc_favored.txt`: This KB contains very few initial facts but a large number of complex and irrelevant inference paths. **Hypothesis:** BC will be significantly faster because its goal-driven approach will follow the single, direct path to the solution, ignoring all the irrelevant rules. FC, in contrast, would waste considerable time exploring and deriving every possible fact from the irrelevant rules.

- `balanced.txt`: This KB provides a more typical scenario with a moderate number of facts and a moderately deep inference path. **Hypothesis:** The performance difference between FC and BC will be less pronounced, providing a baseline for comparison.

2. Generic Solvers (TT vs. DPLL/Resolution): This comparison is designed to demonstrate the critical importance of efficient algorithms when dealing with non-Horn clauses and a growing number of variables.

- `generic_small.txt` (4 symbols): With only $2^4 = 16$ models to check, the overhead of the more complex DPLL and Resolution algorithms might make them slower than the brute-force Truth Table.

- `generic_medium.txt` (8 symbols): With $2^8 = 256$ models, the exponential nature of TT should become apparent, and it is expected to be significantly slower than DPLL and Resolution.

- `generic_large.txt` (16 symbols): With $2^{16} = 65,536$ models, this test is designed to show that the Truth Table approach becomes computationally infeasible. **Hypothesis:** The execution time for TT will be orders of magnitude greater than for DPLL and Resolution, highlighting why these more advanced SAT-solving techniques are essential for practical applications.

6.2.3 Results and Discussion

The empirical results from our performance benchmarks align closely with the theoretical expectations, providing clear insights into the practical strengths and weaknesses of each algorithm.

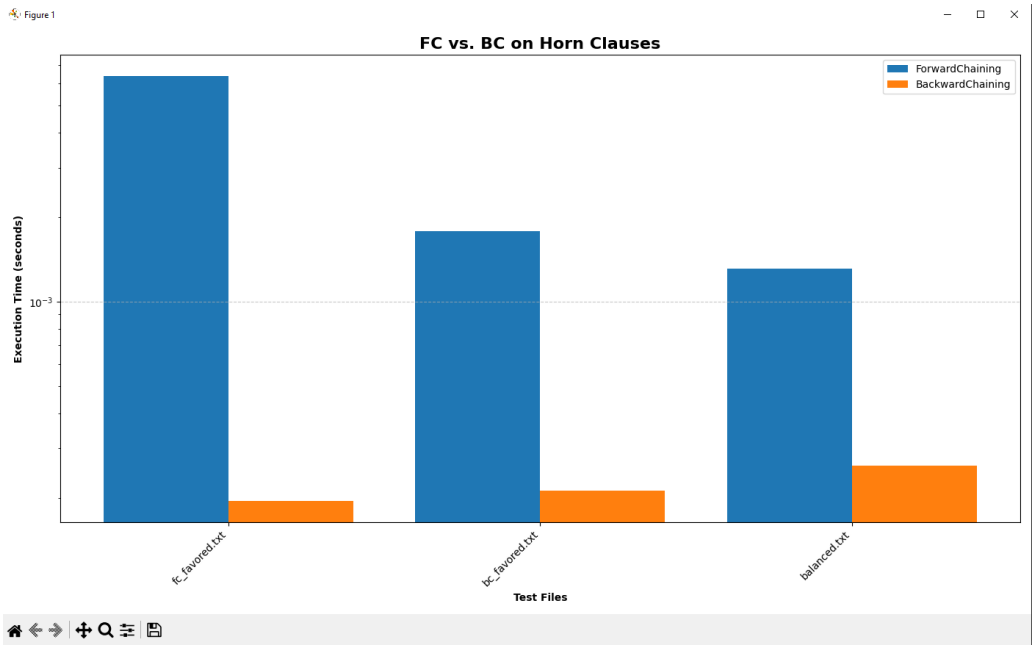

Figure 6: Performance comparison of Forward Chaining vs. Backward Chaining.

Forward Chaining vs. Backward Chaining: As shown in Figure 6, the performance of the chaining algorithms is highly dependent on the structure of the knowledge base. In the `fc_favored.txt` case, which contained many irrelevant initial facts, Backward Chaining was orders of magnitude faster. This is because its goal-driven search completely ignored the irrelevant data, whereas Forward Chaining's data-driven approach forced it to process every single fact. Conversely, in the `bc_favored.txt` case, which contained many irrelevant rule paths, Backward Chaining's focused search once again proved superior by only exploring the single chain of reasoning required to prove the query. Forward Chaining, while not needing to process irrelevant facts, still incurred overhead from initializing its data structures for all the irrelevant rules. These results confirm that BC is generally more efficient when the query is specific and the KB is large and complex.

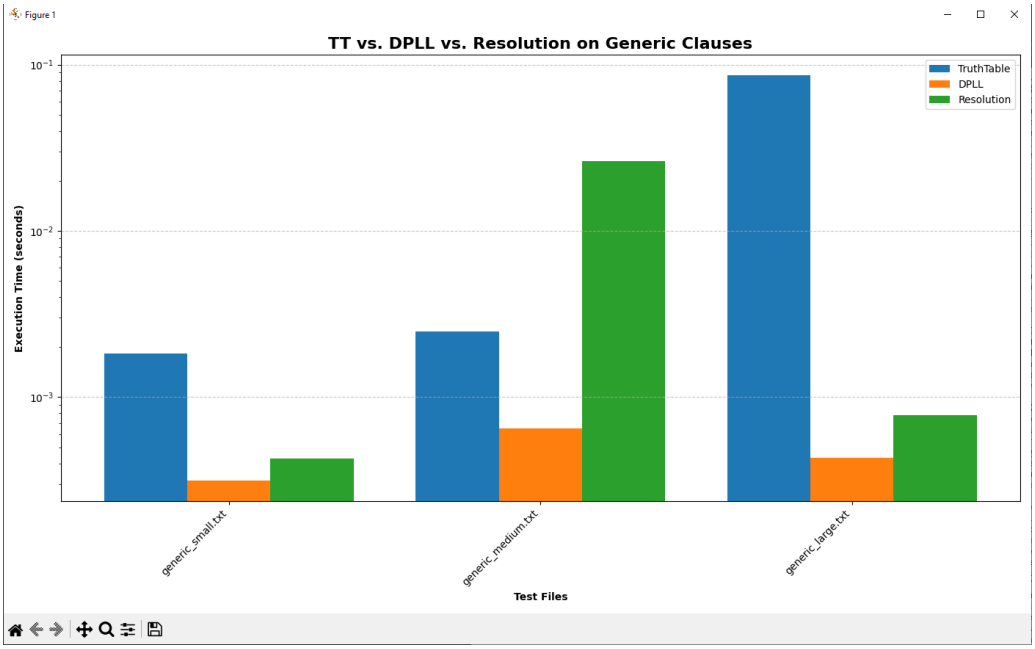

Figure 7: Performance comparison of generic solvers on KBs of increasing complexity.

Generic Solvers: The comparison of generic solvers, shown in Figure 7, dramatically illustrates the problem of computational complexity. The execution time for the Truth Table algorithm grew exponentially as the number of symbols increased from 4 (`generic_small`) to 16 (`generic_large`). This is a direct result of its $O(2^n)$ complexity, as it must evaluate $2^{16} = 65,536$ models for the largest test case, rendering it practically unusable. In contrast, the DPLL algorithm remained exceptionally fast across all test cases, demonstrating the power of its simplification heuristics to prune the search space. Resolution was consistently faster than the Truth Table but showed more variability than DPLL, which is likely due to the naive implementation's generation of many intermediate clauses before finding a contradiction. These results unequivocally demonstrate the necessity of advanced SAT-solving techniques like DPLL for any non-trivial generic logic problem.

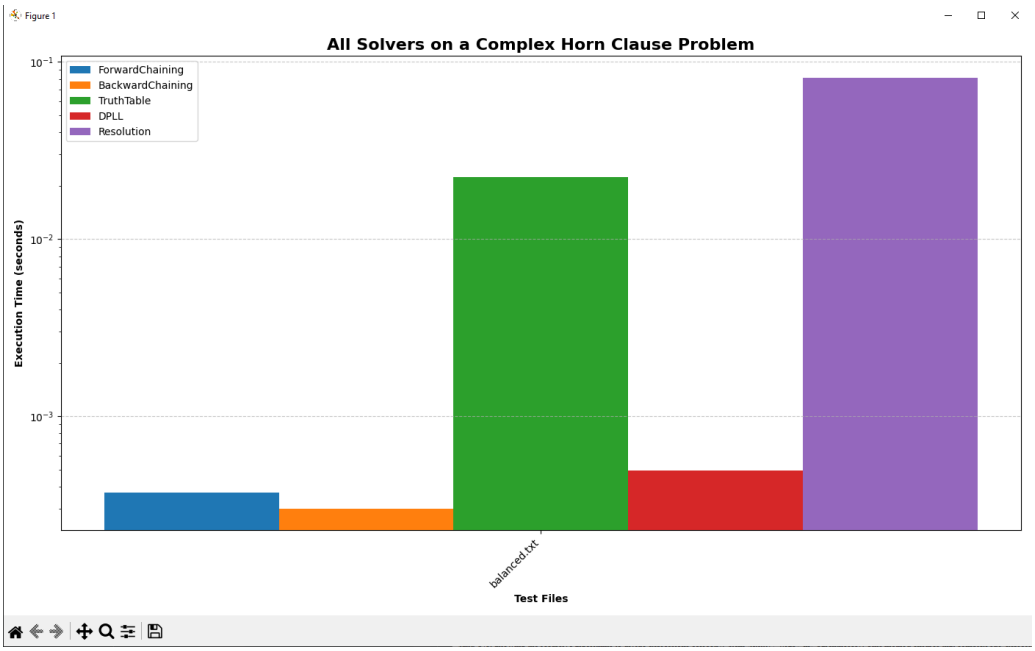

Figure 8: Performance comparison of all implemented solvers on a balanced Horn clause problem.

Overall Comparison: The final benchmark on the `balanced.txt` file (Figure 8) provides a holistic view. It confirms that for problems they are designed for (Horn clauses), the specialized FC and BC algorithms are the most efficient due to their low overhead. DPLL remains a strong competitor, showcasing its power as a general-purpose solver. Truth Table and Resolution, however, are shown to be highly inefficient on this task compared to the others, highlighting the trade-off between universality and performance.

Table 2: Team Contribution Summary

Task	**Minh-Hieu Tran**	**Anh-Quan Nguyen**
Algorithm Implementation	30%	70%
Report and Documentation	60%	40%
Testing (Accuracy & Performance)	60%	40%

7 Conclusion

Overall, this project successfully achieved the design and implementation of a fully-functional propositional logic inference engine. The engine correctly implements the three core algorithms—Truth Table, Forward Chaining, and Backward Chaining—and extends its capabilities with two advanced refutation-based solvers, Resolution and DPLL, as part of a research initiative.

The performance analysis proposed insights that align with theoretical computer science principles. The exponential complexity of the Truth Table method was well demonstrated, confirming its impracticality for problems of significant scale. In contrast, the linear-time performance of Forward and Backward Chaining on Horn clauses highlighted the efficiency gains of specialized algorithms. The benchmarks clearly illustrated the trade-offs between the data-driven approach of FC, which excels when knowledge bases are fact-rich, and the goal-driven strategy of BC, which is superior for specific queries in large, complex rule sets.

Furthermore, the implementation of Resolution and DPLL showed the power of modern SAT-solving techniques. DPLL, with its heuristics, proved to be an exceptionally efficient general-purpose solver, outperforming the more naive Resolution implementation and the brute-force Truth Table.

Ultimately, our development demonstrates that there is no single "best" algorithm for logical inference. The optimal choice is dependent on the specific characteristics of the problem: the structure of the knowledge base (Horn vs. Generic), the nature of the query, and the scale of the problem. The developed engine provides a flexible framework for applying the right tool for the right task. Future work could explore more advanced SAT-solving heuristics for DPLL or extend the engine's capabilities to the more expressive domain of First-Order Logic.