Table of Contents

I. INSTRUCTIONS 2

1. How to use the UI mode 2
2. How to use the CLI mode 2

II. INTRODUCTION 2

III. INFERENCE METHODS 3

1. Truth Table (TT) checking 3
2. Forward Chaining (FC) 3
3. Backward Chaining (BC) 4
4. DPLL (Davis-Putnam-Logemann-Loveland) 4

IV. IMPLEMENTATION 5

1. LogicalConnective (enumeration, can remove if unneccessary) 5
2. KnowledgeBase 6
3. InferenceEngine 6
4. TruthTable 7
5. ChainingSolver 7
6. ForwardChaining 7
7. BackwardChaining 8
8. DPLL 8

V. TESTING 9

VI. FEATURES/BUGS/MISSING 9

1. Features 9
2. Known Bugs 10
3. Missing 10

b. Non-Horn Clause Support for FC/BC: 10

VII. RESEARCH 11

1. Extension to General Knowledge Bases 11
2. Automated Testing and Insights 12
3. Future Work 12

VIII. TEAM SUMMARY REPORT 12

1. Collaboration and Workflow 12
2. Contribution Breakdown 13

IX. CONCLUSION 13

X. ACKNOWLEDGEMENTS/RESOURCES 13

XI. REFERENCES 14

I. Instructions

1. **How to use the UI mode**

   a. **Running the Backend**
   - Download and extract the zip file provided to a directory
   - Navigate to the extracted directory
   - Install the required Python package
     ```
     python install -r requirements.txt
     ```
   - Start the FastAPI server
     ```
     uvicorn api:app --reload
     ```

   b. **Running the Frontend**
   - Open a new terminal and navigate to the UI directory
     ```
     cd iengine-ui
     ```
   - Install the required Node packages (once only for the initial setup)
     ```
     npm install
     ```
   - Start the development server
     ```
     npm run dev
     ```

   c. **Using the program for the UI mode**
   - Open your web browser and navigate to http://localhost:5173/
   - Upload your input file and select the inference method (TT, FC, BC, or DPLL).
   - The results will be displayed on the UI.

2. **How to use the CLI mode**

   Open a new terminal and use the command:
   ```
   python iengine.py <filename> <method>
   ```

Additional Notes:

- The project is divided into a backend implemented in Python using FastAPI and a frontend implemented using modern JavaScript frameworks.
- The inference methods implemented include Truth Table, Forward Chaining, Backward Chaining, and DPLL.
- The program supports both CLI and UI modes for user interaction.
- Ensure that all prerequisites such as Python, FastAPI, Uvicorn, and Node.js are installed before running the program.

II. Introduction

Propositional logic inference engines are fundamental components in artificial intelligence that enable automated reasoning and logical deduction. This report presents our implementation of a comprehensive inference engine system that determines logical entailment from knowledge bases using multiple reasoning methods.

Our implementation features four distinct inference methods: Truth Table (TT) checking for systematic evaluation of truth values, Forward Chaining (FC) for data-driven reasoning, Backward Chaining (BC) for goal-oriented deduction, and the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for enhanced satisfiability solving.

Our system combines a Python backend for processing logic with a JavaScript frontend interface, supporting both Horn-form and general propositional logic formulas with standard logical connectives.

III. Inference methods

1. **Truth Table (TT) checking**

   Truth Table checking is a complete but computationally intensive method for determining logical entailment.

   a. **How it works**
   - Enumerate all possible combinations of truth values for propositional symbols
   - Evaluate the knowledge base (KB) under each combination
   - Check if the query (α) is true in all models where KB is true
   - If KB $\models \alpha$ is true, then α must be true in every model where KB is true

   b. **Complexity**
   - Time complexity: $O(2^n)$ where n is the number of propositional symbols
   - Space complexity: O(n) for storing truth assignments

   c. **Advantages**
   - Complete and sound for propositional logic
   - Guarantees to find all models that satisfy the formula
   - Simple and straightforward to implement and understand

   d. **Limitations**
   - Exponential complexity makes it impractical for large knowledge bases
   - Must examine all possible combinations even when unnecessary

2. **Forward Chaining (FC)**

   Forward chaining is a data-driven reasoning method that works efficiently with Horn clauses.

   a. **How it works**
   - Start with known facts in KB
   - Apply Modus Ponens to derive new facts
   - Add newly inferred facts to KB
   - Repeat until either:
     - Query is proven
     - No new facts can be derived

   b. **Complexity**
   - Time complexity: O(n), where n is the number of propositional symbols
   - Space complexity: O(n) for storing the agenda and inferred facts

   c. **Advantages**
   - Effective for Horn clauses
   - Works well when many conclusions can be derived from facts
   - Can quickly generate new knowledge from existing facts

   d. **Limitations**
   - Only works with Horn clauses
   - Can generate many irrelevant facts, potentially leading to inefficiency
   - May require extensive rule sets for complex problems

3. **Backward Chaining (BC)**

   Backward Chaining is a goad-directed reasoning method that works backwards from the query.

   a. **How it works**
      - Start with the query goal
      - Find rules that could prove the goal
      - Add the premises of these rules as new subgoals
      - Recursively prove subgoals
      - Continue until reaching known facts or determining impossibility

   b. **Complexity**
      - Time complexity: O(n), where n is the number of propositional symbols
      - Space complexity: O(n) for storing the goal stack

   c. **Advantages**
      - Efficient in focusing only on relevant facts needed to achieve the goal
      - Reduces the search space by directly targeting the goal

   d. **Limitations**
      - Only works with Horn clauses
      - Can get stuck in infinite loop if not implemented with cycle detection
      - May require backtracking, which can be computationally expensive

4. **DPLL (Davis-Putnam-Logemann-Loveland)**

   DPLL is a complete algorithm for solving the satisfiability problem in propositional logic.

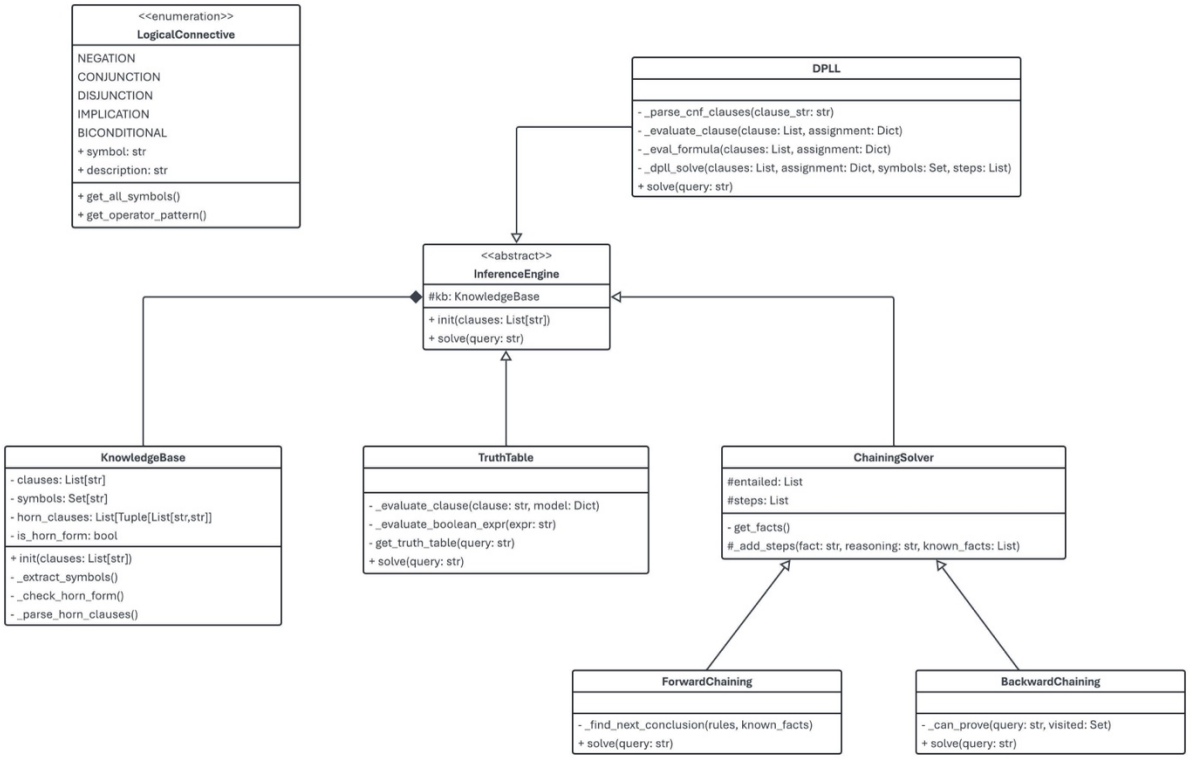
   a. **How it works**
      - Apply unit propagation
      - Apply pure literal elimination
      - Choose a literal and recursively:
        - Try setting it to true
        - If unsatisfiable, try setting it to false
      - Repeat until finding a satisfying assignment or proving unsatisfiability

   b. **Complexity**
      - Time complexity: $O(2^n)$ in worse case, but often much better in practice
      - Space complexity: O(n) for storing assignments

   c. **Advantages**
      - Complete for all propositional logic formulas
      - Generally more efficient than truth tables
      - Can handle non-Horn clauses

   d. **Limitations**
      - May still face exponential time complexity in the worst-case scenarios
      - Requires the formula to be in CNF, which may complicate preprocessing

IV. Implementation



1. LogicalConnective

Pseudocode:

```
Class LogicalConnective(Enum):
    NEGATION: ('~', 'negation')
    CONJUNCTION: ('&', 'conjunction')
    DISJUNCTION: ('||', 'disjunction')
    IMPLICATION: ('=>', 'implication')
    BICONDITIONAL: ('<=>', 'biconditional')

    Method __init__(symbol, description):
        Set self.symbol to symbol
        Set self.description to description

    Class Method get_all_symbols():
        Return a set of all symbols from the Enum members

    Class Method get_operator_pattern():
        Return a regex pattern to match all logical operators
```

2. KnowledgeBase

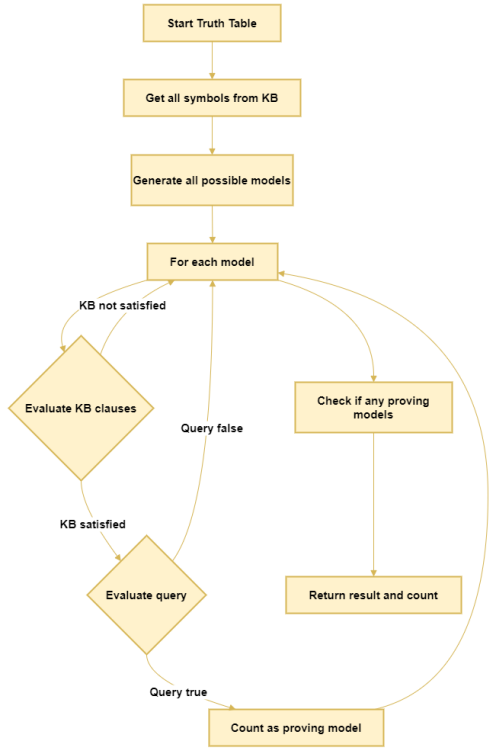
```
Class KnowledgeBase:
    Method __init__(clauses: List[str]):
        Set self.clauses to clauses
        Set self.symbols using _extract_symbols()
        Set self.horn_clauses using _parse_horn_clauses()
        Set self.is_horn_form using _check_horn_form()

    Method _extract_symbols():
        Get operator pattern from LogicalConnective
        Initialize an empty set for symbols
        For each clause in self.clauses:
            Replace operators in clause with spaces
            Extract alphanumeric tokens and add to symbols set
        Return symbols

    Method _check_horn_form():
        For each clause in self.clauses:
            If clause contains '||' or '<=>':
                Return False
            If clause contains '=>':
                Split clause by '=>'
                If the antecedent contains '~':
                    Return
        Return True

    Method _parse_horn_clauses():
        Initialize an empty list for parsed_clauses
        For each clause in self.clauses:
            If clause contains '||' or '<=>':
                Continue to next clause
            If clause contains '=>':
                Split clause by '=>'
                Split premises by '&' if contains '&'
                Add premises and conclusion to parsed_clauses
            Else:
                Split clause by '&'
                For each part in split clause:
                    If not contains '~':
                        Add empty premises and part to parsed_clauses
        Return parsed_clauses
```

3. InferenceEngine

```
Class InferenceEngine(ABC):
    Method __init__(clauses: List[str]):
        Try:
            Initialize self.kb with KnowledgeBase(clauses)
            If not instance of DPLL or TruthTable and not self.kb.is_horn_form:
                Print error message and exit
        Except Exception:
            Print error message and exit

    Abstract Method solve(query: str):
        Pass
```

4. TruthTable

Flowchart

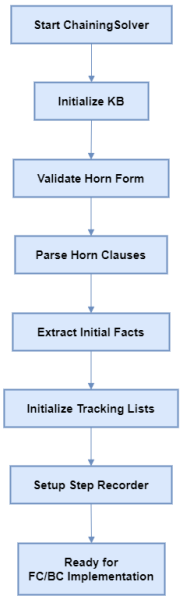


Pseudocode:

```
Class TruthTable(InferenceEngine):
    Method _evaluate_clause(clause: str, model: Dict[str, bool]):
        Replace symbols in clause with their boolean values from model
        Try:
            While '=>' in clause:
                Split clause by '=>'
                Evaluate left and right parts
                Set result to (not left) or right
                Replace clause with result
            Return evaluated clause as boolean
        Except Exception:
            Print error message and return False

    Method _evaluate_boolean_expr(expr: str):
        Replace logical operators in expr with Python boolean operators
        Try:
            Return evaluated expr as boolean
        Except:
            Return False

    Method get_truth_table(query: str):
        Initialize symbols, total_models, and truth_table
        For each model (combination of boolean values for symbols):
            Initialize model dictionary
            Evaluate each clause and query with model
            Add results to truth_table
        Calculate summary of truth_table
        Return truth_table

    Method solve(query: str):
        Generate truth table for query
        Return whether query is entailed and number of proving models
```

5. ChainingSolver

Flowchart:

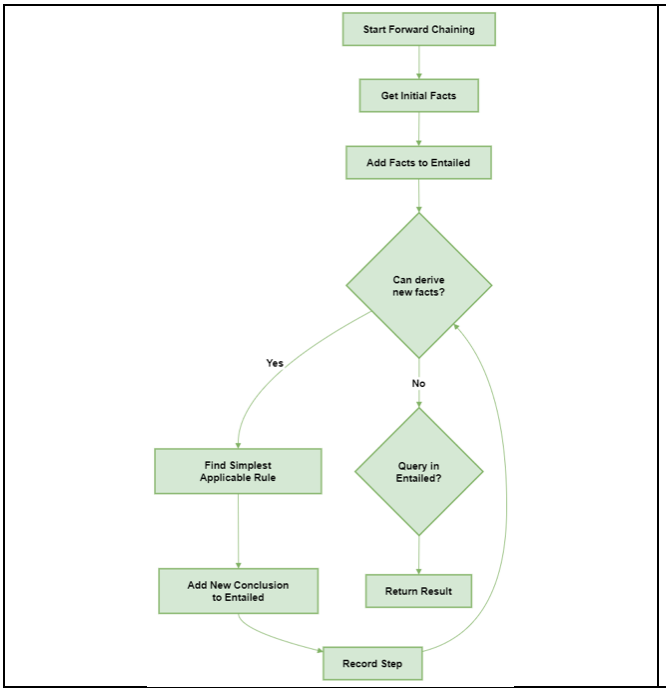


Pseudocode:

```
Class ChainingSolver(InferenceEngine):
    Method __init__(clauses: List[str]):
        Call super().__init__(clauses)
        Initialize self.entailed and self.steps

    Method _get_facts():
        Return set of conclusions with empty premises from self.kb.horn_clauses

    Method _add_step(fact: str, reasoning: str, known_facts: List[str] = None):
        Add reasoning step with fact, reasoning, and known_facts to self.steps
```

6. ForwardChaining

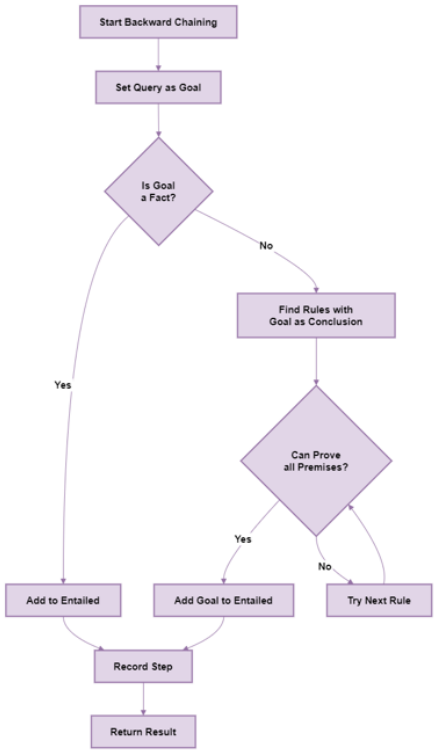
Flowchart	Pseudocode




```
Class ForwardChaining(ChainingSolver):
    Method _find_next_conclusion(rules, known_facts):
        Initialize candidate_rules
        For each rule in rules:
            If premises are satisfied by known_facts:
                Add rule to candidate_rules with complexity
        If no candidate_rules:
            Return None, None
        Sort candidate_rules by complexity
        Return premises and conclusion of simplest rule

    Method solve(query: str):
        Initialize self.entailed and self.steps
        Initialize facts from _get_facts()
        For each fact in sorted facts:
            Add fact to self.entailed and add reasoning step
        Get rules excluding pure facts
        While True:
            Find next conclusion using _find_next_conclusion()
            If no conclusion:
                Break
            Add conclusion to self.entailed and add reasoning step
            Remove used rule
        Return whether query is in self.entailed and self.entailed
```

7. BackwardChaining

Flowchart

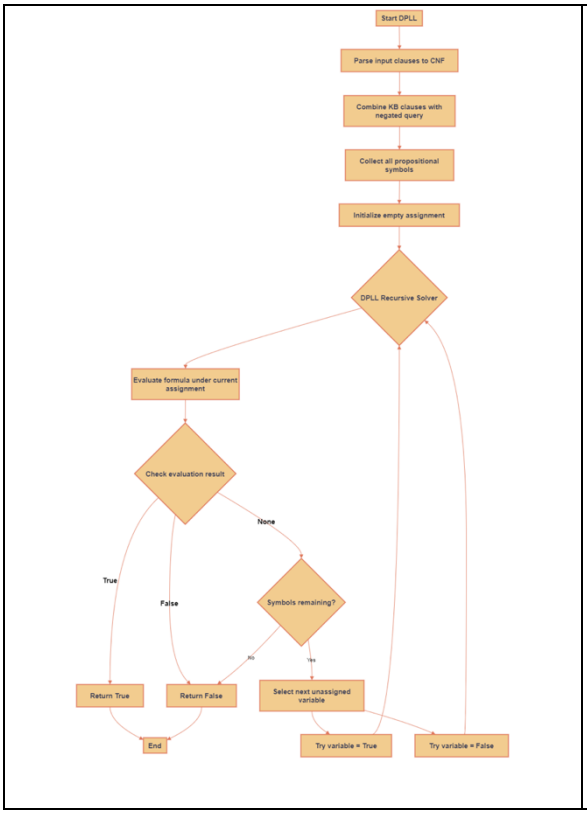


Pseudocode

```
Class BackwardChaining(ChainingSolver):
    Method _can_prove(query: str, visited: Set[str]):
        If query in visited:
            Return False
        Add query to visited
        If query is a fact:
            Add query to self.entailed and add reasoning step
            Return True
        For each premises and conclusion in self.kb.horn_clauses:
            If conclusion is query:
                Initialize all_premises_proven and required_premises
                For each premise:
                    If _can_prove(premise, visited.copy()):
                        Add premise to required_premises
                    Else:
                        Set all_premises_proven to False and break
                If all_premises_proven:
                    Add query to self.entailed and add reasoning step
                    Return True
        Return False

    Method solve(query: str):
        Initialize self.entailed and self.steps
        Add initial goal step with query
        Call _can_prove(query, set())
        Return result and self.entailed
```

8. DPLL

Flowchart	Pseudocode




```
Class DPLL(InferenceEngine):
    Method _parse_cnf_clauses(clause_str: str):
        Split clause_str into individual clauses by '&'
        Initialize result_clauses
        For each clause:
            Remove outer parentheses
            If clause contains '<=>':
                Convert to equivalent clauses and parse recursively
                Add results to result_clauses
                Continue
            If clause contains '=>':
                Convert to equivalent clause
            Split clause by '||' and add literals to result_clauses
        Return result_clauses

    Method _evaluate_clause(clause: List[Tuple[str, str]], assignment: Dict[str, bool]):
        For each literal in clause:
            If literal is satisfied by assignment:
                Return True
            Else:
                Return None
        Return False

    Method _eval_formula(clauses: List[List[Tuple[str, str]]], assignment: Dict[str, bool]):
        Initialize results
        For each clause in clauses:
            Evaluate clause with assignment
            If clause is False:
                Return False
            Add result to results
        If all clauses are evaluated:
            Return True
        Return None

    Method _dpll_solve(clauses: List[List[Tuple[str, str]]], assignment: Dict[str, bool], symbols: Set[str], steps: List[str]):
        Evaluate formula with assignment
        If formula is True:
            Return True
        If formula is False:
            Return False
        If no symbols left:
            Return False
        Get next symbol and remaining symbols
        Try assigning True to symbol and solve recursively
        If solved:
            Update assignment and return True
        Try assigning False to symbol and solve recursively
        If solved:
            Update assignment and return True
        Return False

    Method solve(query: str):
        Parse CNF clauses from self.kb.clauses and query
        Negate query clauses and add to all_clauses
        Initialize symbols from all_clauses
        Initialize assignment and steps
        Solve using _dpll_solve()
        Return whether formula is satisfiable and assignment
```

V. Testing

VI. Features/Bugs/Missing

1. **Features**

   a. **Inference Methods**
   - **Truth Table (TT)**:
     - Implements systematic evaluation of all truth assignments.
     - Provides detailed truth table generation and query evaluation.
   - **Forward Chaining (FC)**:
     - Implements rule-based forward derivation with efficient rule prioritization.
   - **Backward Chaining (BC)**:
     - Implements goal-directed reasoning using recursive query resolution.
   - **DPLL Algorithm**:
     - Implements SAT-solving with features like unit propagation and pure literal elimination.
     - Efficiently handles CNF-based propositional logic.

   b. **File Parsing and Command-Line Interface (CLI)**:
   - Parses TELL and ASK sections from input files.
   - Supports invocation via `python iengine.py <filename> <method>`.
   - Outputs results in specified format:
     - **YES/NO** for query entailment.
     - Additional information on models or derived facts.

c. **Web-Based User Interface (UI)**:
   - Allows users to upload `.txt` files and select inference methods.
   - Visualizes inference processes:
     - **ChainViz** for FC/BC visualizations.
     - **DPLLViz** for tree-based visualization of the DPLL process.
   - Displays truth tables for TT results.

d. **Error Handling**:
   - Provides error messages for invalid input formats, missing files, and unsupported features.
   - Handles edge cases, such as cyclic dependencies in chaining algorithms.

e. **Testing Support**:
   - Designed to be tested with diverse input files containing Horn clauses and queries.

f. **Visualization Enhancements**:
   - Clear and interactive visualization of inference steps in ChainViz and DPLLViz.
   - Step-by-step animation with playback controls for visual clarity.

2. **Known Bugs**

a. **DPLL Visualization**:
Even with the implementation of screen tracking that automatically focuses on nodes as they are explored, the visualization of very large CNF formulas can still be challenging. When the tree becomes exceedingly large, observing and interpreting the structure and steps may remain difficult due to the sheer number of nodes and connections.

b. **Frontend Responsiveness**:
While mechanisms to enhance responsiveness, such as progressive rendering and efficient table handling, are implemented, the UI may still experience delays when handling extremely large truth tables. This is particularly noticeable in the Truth Table (TT) method, where rendering numerous models and results can take a significant amount of time for large knowledge bases.

3. **Missing**

b. **Non-Horn Clause Support for FC/BC:**
   - Forward and Backward Chaining methods are limited to Horn-form knowledge bases as required.
   - General propositional logic formulas cannot be processed using FC/BC.

c. **Advanced Optimizations**:
   - Optimizations for Truth Table (TT) and DPLL algorithms (e.g., heuristic variable ordering) are not fully implemented.

Additional Notes

- **Visualization Contributions**:
  - ChainViz and DPLLViz provide intuitive insights into the inference process.
  - Include step-by-step visualizations and playback controls for enhanced understanding.
- **Testing**:
  - Comprehensive test cases are prepared for all methods, demonstrating correctness and edge case handling.
- **File Format Compliance**:
  - Ensures compatibility with the required TELL and ASK file format for smooth operation.

VII. Research

1. Extension to General Knowledge Bases

To address the requirement of extending the inference engine to handle general propositional logic knowledge bases, I explored and implemented two key approaches:

a. Truth Table (TT) for General Knowledge Bases

- The Truth Table method in my implementation is designed to work with all types of knowledge bases, including those beyond Horn form.
- By leveraging logical connectives such as disjunction (||), conjunction (&), negation (~), implication (=>), and biconditional (<=>), the TT method systematically evaluates every possible truth assignment for the propositional symbols in the knowledge base.
- Implementation Highlights:
  - A comprehensive truth table is generated for all models.
  - Logical expressions are evaluated in each model to determine the satisfaction of both the knowledge base and the query.
  - The method outputs whether the query is entailed and provides the count of models that support the query.
- Key Observations:
  - While the TT method guarantees completeness, its exponential complexity limits scalability for large knowledge bases with many symbols.
  - To address this, optimization techniques like incremental evaluation and caching of intermediate results could be considered in future work.

b. DPLL (Davis-Putnam-Logemann-Loveland) Algorithm for General Knowledge Bases

- The DPLL algorithm was implemented to solve the satisfiability problem efficiently for knowledge bases in Conjunctive Normal Form (CNF).
- DPLL is well-suited for general knowledge bases as it does not rely on Horn form restrictions and incorporates advanced features for efficiency.
- Implementation Highlights:
  - Conversion of logical formulas to CNF.
  - Utilization of unit propagation and pure literal elimination to simplify clauses.
- Recursive decision-making with backtracking to explore possible truth assignments.
- Enhancements:
  - Visualization: A tree-based visualization (DPLLViz) was developed to track the algorithm's decision-making process. Each node represents a step in the search tree, and edges indicate decisions or backtracks.
  - Screen Tracking: The UI includes automatic focus adjustments to ensure the active node remains in view as the algorithm explores the tree.
- Key Observations:
  - DPLL significantly outperforms TT for large and complex knowledge bases due to its pruning capabilities and targeted search.
  - However, even with visualization and screen tracking, large CNF formulas can produce trees too complex for effective observation.

2. **Automated Testing and Insights**

   To validate and benchmark the implementation, I conducted extensive automated testing across various types of knowledge bases:

   a. **Test Cases**:
      - Included both Horn-form and general propositional logic knowledge bases.
      - Varied the size and complexity of the knowledge bases to test scalability and robustness.
      - Incorporated edge cases such as unsatisfiable queries and redundant clauses.

   b. **Insights from Testing**:
      - **TT Method**:
        - Guaranteed correctness but experienced exponential growth in runtime and memory usage as the number of symbols increased.
        - Showed clear limitations for knowledge bases with more than 15 symbols.
      - **DPLL Algorithm**:
        - Demonstrated significantly better scalability compared to TT.
        - Handled larger knowledge bases efficiently by pruning large portions of the search space.
        - Provided valuable insights into the satisfiability of queries, particularly for complex CNF formulas.

3. **Future Work**

   - **Optimizations**:
     - Introduce heuristic-based decision-making in DPLL to prioritize variables with higher impact on clause satisfaction.
     - Implement lazy evaluation for TT to avoid unnecessary computation of unsatisfied models.
   - **Additional Research**:
     - Explore the implementation of a resolution-based theorem prover to complement the existing methods.
     - Investigate techniques to simplify and preprocess knowledge bases before applying inference methods.

   By implementing both DPLL and TT for general knowledge bases, this research advances the versatility and effectiveness of the inference engine, paving the way for further enhancements and practical applications.

VIII. Team summary report

1. **Collaboration and Workflow**

Throughout this project, we maintained a highly collaborative approach. Both of us were actively involved in all aspects of the assignment, working closely together to ensure a high-quality implementation and report. Here are the key aspects of our teamwork:

   a. **Coding**:

- We jointly contributed to the coding process, often sitting together to discuss and implement solutions in real time.
- We used Git for version control, regularly pushing and pulling code changes to stay synchronized. This allowed us to identify and fix bugs collaboratively and iteratively.

b. Bug Fixing and Testing:

- We tested the code extensively as a team, identifying edge cases and refining the implementation until both of us were satisfied with the results.
- Whenever a bug was encountered, we brainstormed solutions together and worked through the debugging process as a pair.

c. Report Writing:

- We shared the responsibility of writing the report.
- Each of us contributed to different sections of the report, reviewing and providing feedback on each other's writing to ensure clarity and coherence.

d. Feedback and Review:

- We provided constant feedback to each other during the development process, offering suggestions for improvements and alternative approaches.
- This iterative process of review and refinement helped us produce a polished and well-rounded submission.

2. Contribution Breakdown

We divided the work equally to ensure fairness and balanced effort. Below is the contribution percentage for each team member:

Team Member	Contribution Percentage	Areas of Contribution
Thien	50%	Coding, debugging, report writing, testing
Truc	50%	Coding, debugging, report writing, testing

Total Contribution: 100%

IX. Conclusion

X. Acknowledgements/Resources

1. **Python Official Documentation**:
   a. Used extensively to reference language-specific implementations, such as working with regular expressions and optimizing data structures.
   b. URL: Python Docs
2. **FastAPI Documentation**:
   a. Provided guidance on setting up the backend API for file handling and inference method integration.
   b. URL: FastAPI Docs
3. **React.js Documentation**:

   a. Supported the development of the frontend, particularly for building the ChainViz and DPLLViz components for visualization.
   b. URL: React Docs
4. **GitHub and Stack Overflow**:
   a. Leveraged GitHub repositories and Stack Overflow discussions for debugging issues and exploring best practices for algorithm implementations.
   b. URLs: GitHub | Stack Overflow

XI. References

Ref có chỉnh sửa

Russell, S., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.)
Davis, M., & Putnam, H. (1960). A Computing Procedure for Quantification Theory. Journal of the ACM, 7(3)
Davis, M., Logemann, G., & Loveland, D. (1962). A Machine Program for Theorem-Proving. Communications of the ACM, 5(7)
Artificial Intelligence: Foundations of Computational Agents (2nd ed.) by David L. Poole and Alan K. Mackworth - Available online at: https://artint.info/2e/html/ArtInt2e.html
Stanford CS221 Artificial Intelligence Course Materials - Available at: https://stanford-cs221.github.io/
MIT OpenCourseWare 6.034 Artificial Intelligence - Available at: https://ocw.mit.edu/courses/6-034-artificial-intelligence-fall-2010/

```
Class DPLL(InferenceEngine):
    Method _parse_cnf_clauses(clause_str: str):
        Split clause_str into individual clauses by '&'
        Initialize result_clauses
        For each clause:
            Remove outer parentheses
            If clause contains '<=>':
                Convert to equivalent clauses and parse recursively
                Add results to result_clauses
                Continue
            If clause contains '=>':
                Convert to equivalent clause
            Split clause by '||' and add literals to result_clauses
        Return result_clauses

    Method _evaluate_clause(clause: List[Tuple[str, str]], assignment: Dict[str, bool]):
        For each literal in clause:
            If literal is satisfied by assignment:
                Return True
            Else:
                Return None
        Return False

    Method _eval_formula(clauses: List[List[Tuple[str, str]]], assignment: Dict[str, bool]):
        Initialize results
        For each clause in clauses:
            Evaluate clause with assignment
            If clause is False:
                Return False
            Add result to results
        If all clauses are evaluated:
            Return True
        Return None

    Method _dpll_solve(clauses: List[List[Tuple[str, str]]], assignment: Dict[str, bool], symbols: Set[str], steps: List[str]):
        Evaluate formula with assignment
        If formula is True:
            Return True
        If formula is False:
            Return False
        If no symbols left:
            Return False
        Get next symbol and remaining symbols
        Try assigning True to symbol and solve recursively
        If solved:
            Update assignment and return True
        Try assigning False to symbol and solve recursively
        If solved:
            Update assignment and return True
        Return False

    Method solve(query: str):
        Parse CNF clauses from self.kb.clauses and query
        Negate query clauses and add to all_clauses
        Initialize symbols from all_clauses
        Initialize assignment and steps
        Solve using _dpll_solve()
        Return whether formula is satisfiable and assignment
```