Contents

[**1. Introduction** 1](#_Toc182919692)

[**1.1 Background** 1](#_Toc182919693)

[**2. Inference Methods** 1](#_Toc182919694)

[**2.1 Truth Table Checking** 1](#_Toc182919695)

[**2.1.1 How It Works** 1](#_Toc182919696)

[**2.1.2 Example** 1](#_Toc182919697)

[**2.2 Forward Chaining** 1](#_Toc182919698)

[**2.2.1 How It Works** 1](#_Toc182919699)

[**2.2.2 Example** 2](#_Toc182919700)

[**2.3 Backward Chaining** 2](#_Toc182919701)

[**2.3.1 How It Works** 2](#_Toc182919702)

[**2.3.2 Example** 2](#_Toc182919703)

[**2.4 DPLL (Davis-Putnam-Logemann-Loveland)** 2](#_Toc182919704)

[**2.4.1 How It Works** 2](#_Toc182919705)

[**2.4.2 Example** 2](#_Toc182919706)

[**3. Implementation** 2](#_Toc182919707)

[**3.1 Parse\_files analyzer** 2](#_Toc182919708)

[**3.1.1 File Format** 2](#_Toc182919709)

[**3.2 Inference Methods** 2](#_Toc182919710)

[**3.2.1 Truth Table Checker** 2](#_Toc182919711)

[**3.2.2 Chaining** 2](#_Toc182919712)

[**3.2.3 Forward Chaining (FC)** 3](#_Toc182919713)

[**3.2.4 Backward Chaining (BC)** 3](#_Toc182919714)

[**3.2.5 DPLL (Davis-Putnam-Logemann-Loveland)** 3](#_Toc182919715)

[**4. Testing** 3](#_Toc182919716)

[**4.0.1 Horn Testing** 3](#_Toc182919717)

[**4.0.2 General Logic Testing** 3](#_Toc182919718)

[**5. Features/Bugs** 3](#_Toc182919719)

[**5.1 Features** 3](#_Toc182919720)

[**5.2 Bugs** 3](#_Toc182919721)

[**6. Research** 3](#_Toc182919722)

[**6.1 Information about Research Component** 3](#_Toc182919723)

[**6.2 General Propositional Logic and DPLL** 3](#_Toc182919724)

[**6.3 DPLL Implementation** 3](#_Toc182919725)

[**7. Student Contributions** 4](#_Toc182919726)

[**8. Conclusion** 4](#_Toc182919727)

[**9. Acknowledgements/Resources** 4](#_Toc182919728)

[**10. References** 4](#_Toc182919729)

# **1. Introduction**

## **1.1 Background**

# This report outlines the approach and techniques used to fulfill the requirements of Assignment 2, where the task was to implement an inference engine for propositional logic in Python. The goal of this assignment is to demonstrate how an inference engine applies logical rules to a knowledge base (KB) to deduce information or make decisions.

In this assignment, the provided knowledge base (KB) and query (q) are given in a specific format. The KB, which follows the keyword **TELL**, consists of Horn clauses that are separated by semicolons, while the query (q), introduced by the keyword **ASK**, contains a proposition symbol that we evaluate for entailment within the KB.

**1.2 Instruction**

To run test file, use the terminal and follow this structure: python [code\_filename] [test\_filename] [method\_name]

Example: python iengine.py test\_HornKB.txt bc

The iegine.py is the name of our code file, test\_HornKB is the example test filename, there will be 20 others test files in our folder and bc stand for backward chaining, there are 3 remain methods that you can use to run test, Truth Table(tt), Forward Chaining, DPLL.

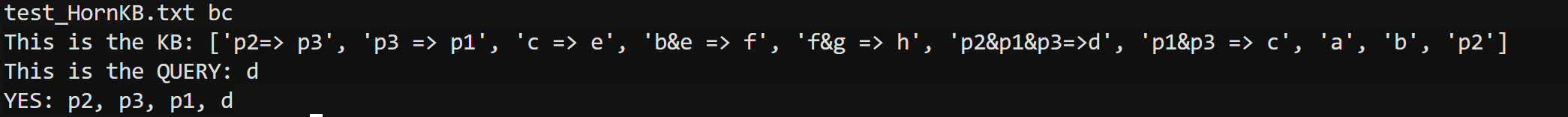
TELL

p2 => p3; p3 => p1; c => e; b&e => f; f&g => h; p2&p1&p3 => d; p1&p3 => c; a; b; p2;

ASK

d

Example output:



The output:

* YES, followed by a colon (:) and the number of models of KB for TT method if the test case is valid, the list of propositional symbols entailed if the method are FC or BC
* It will return the KB and query for DPLL if the query is entailed
* No for invalid input

Figure 1: Example of an expected Horn statement.

The goal of the program is to evaluate whether a query (q), such as the proposition symbol "d," can be entailed from the provided KB using different inference methods. The inference engine is equipped with **Truth Table (TT) Checking**, **Backward Chaining (BC)**, and **Forward Chaining (FC)** algorithms.

# **2. Inference Methods**

The inference engine developed for this assignment utilizes several inference methods to logically determine whether the query (q) is entailed by the KB. Each method works differently, offering unique advantages and limitations.

## **2.1 Truth Table Checking**

Truth Table Checking is a brute-force method that involves enumerating all possible truth assignments for the propositional symbols in the KB and the query (q). The truth table contains all possible combinations of truth values (True or False) for these symbols, with each row representing a different model—a specific combination of truth values.

For every model, the KB and the query are evaluated. If both are true under a given model, the query is considered entailed by the KB. Truth table checking guarantees that all possible scenarios are considered, making it the most comprehensive method for determining entailment. However, its major drawback lies in its exponential complexity. As the number of propositional symbols increases, the size of the truth table grows exponentially, making it impractical for larger KBs or queries【1】.

### **2.1.1 How It Works**

The truth\_table function generates all combinations of truth values for the variables in the knowledge base. It does this by iterating through all possibilities using Python’s itertools.product. For each combination, it checks if the KB holds true, and if it does, whether the query is also true. If the query holds true in all valid cases, it is entailed.

This method is guaranteed to be correct, but its main drawback is efficiency. The time complexity grows exponentially with the number of variables, making it impractical for larger knowledge bases.

### **2.1.2 Example**

The Truth Table method would check all combinations of truth values for A, B, and C to see if C is true whenever the KB is true.

2.1.3 Advantages:

1. The truth table provides clean representation which makes it easier to analyze.
2. It provides a systematic method for analyzing all possible combinations of truth values of a particular logical expression.
3. It is very easy to implement.

2.1.4 Disadvantages:

1. The number of truth assignments grows exponentially with the number of symbols
2. For large KBs with many symbols, this makes the method computationally expensive and impractical.

## **2.2 Forward Chaining**

Forward Chaining is an inference method that begins by analyzing the facts within the KB. It then iteratively applies the rules in the KB to derive new facts, continuously updating the KB with new information until the query (q) is found or no further inferences can be made.

This method excels in scenarios involving Horn clauses, where each rule has a single positive literal. It is particularly effective in linear time reasoning, as it processes rules sequentially based on facts already known to be true【3】. Forward Chaining is best suited for environments where the facts and rules can be applied in a straightforward manner. However, it struggles with more complex scenarios where rules are highly interconnected or involve more general logic, which may not align well with the Horn-clause structure【1】.

### **2.2.1 How It Works**

The FC class implements Forward Chaining. It begins by adding all the known facts to a queue. Then, it iteratively checks each rule in the KB. If all the premises of a rule are satisfied (i.e., they are known to be true), the rule’s conclusion is added to the queue as a new fact. This process continues until the query is deduced or no new facts can be generated.

### **2.2.2 Example**

Forward Chaining starts with the known fact A, then uses the rule A => B to infer B. Next, it uses the rule B => C to infer C, thus proving the query C.

2.2.3Advantages

1. Forward chaining is straightforward and easy to implement.
2. It processes data as it arrives, making it suitable for dynamic environments where new data continuously becomes available.

2.2.4Disadvantages

1. Forward Chaining generates all possible facts from the KB, even if many of them are not relevant to the query

## **2.3 Backward Chaining**

Backward Chaining operates in the opposite direction of Forward Chaining. It starts with the query (q) and works backward, attempting to find facts or rules in the KB that lead to the conclusion of the query. If a premise needed to prove the query is not already a known fact, it becomes a sub-goal. The backward chain then recursively searches for facts that could support this sub-goal until it either finds a valid logical path or fails to find a solution.

Backward Chaining is especially effective when the query is complex but does not require examining every possible rule or fact in the KB. It is well-suited for cases where the KB may have many facts, but only a subset of them are relevant to the query. However, like Forward Chaining, it is less effective in environments where generalized logic or interconnected rule sets need to be evaluated【1】.

### **2.3.1 How It Works**

The BC class is responsible for backward chaining. It takes the query and checks whether it can be proven by finding rules in the KB that lead to the query. If the premises of a rule aren’t already known, it recursively tries to prove those premises, continuing this process until either the query is proven or no further progress can be made.

### **2.3.2 Example**

Backward Chaining starts with the query C and attempts to prove it. First, it checks whether B => C can be satisfied. Since it doesn’t know whether B is true, it then tries to prove B from A using the rule A => B.

2.3.3 Advantages

1. Backward Chaining works backward from the query, checking only the rules and facts necessary to determine whether the query is entailed
2. Backward Chaining does not derive all possible facts—it only explores paths relevant to the query.

2.3.4 Disadvantages

1. It requires predefined goals
2. If multiple goals need to be achieved, backward chaining may need to be repeated for each goal

## **2.4 DPLL (Davis-Putnam-Logemann-Loveland)**

**DPLL** is a more advanced method, typically used for solving the propositional satisfiability problem (SAT). It’s a backtracking algorithm that performs unit propagation (simplifying the formula by assigning truth values to certain literals) and pure literal elimination (removing literals that are always true or false). These optimizations help it handle larger and more complex problems.

### **2.4.1 How It Works**

The DPLL class implements the DPLL algorithm. It starts by checking if the KB is satisfiable under the current assignment of truth values. If all clauses are satisfied, the formula is satisfiable. If any clause is falsified, the algorithm tries a different assignment. It recursively assigns truth values to literals and simplifies the formula until it finds a satisfying assignment or determines that no solution exists.

### **2.4.2 Example**

The DPLL algorithm would try different truth assignments and simplify the formula as it proceeds, ultimately finding a satisfying assignment or concluding that none exists.

# **3. Implementation**

## **3.1 Parse\_files analyzer**

The system expects the KB and the query to be read from a text file. This makes the system flexible, allowing users to easily test different KBs and queries without modifying the code.

### **3.1.1 File Format**

Each clause in the KB is written on a new line, and the query is a single logical statement under the ASK section.

## **3.2 Inference Methods**

### **3.2.1 Truth Table Checker**

The truth\_table function generates all possible truth assignments for the variables in the KB. It checks each assignment to see if the KB holds true and then checks if the query is true in that case. It guarantees completeness but is inefficient for large KBs.

### **3.2.2 Chaining**

The Chaining class is used by both Forward and Backward Chaining. It processes the KB and prepares it for the respective algorithms, extracting the premises and conclusions of each rule.

### **3.2.3 Forward Chaining (FC)**

The FC class starts with known facts and iteratively applies rules to deduce new facts. It continues until the query is inferred or no new facts can be generated. This approach works well when the KB is structured to allow rapid inference.

### **3.2.4 Backward Chaining (BC)**

The BC class works backward from the query, trying to prove it by finding supporting facts in the KB. If necessary, it recursively checks the premises of rules, trying to prove each one.

### **3.2.5 DPLL (Davis-Putnam-Logemann-Loveland)**

The DPLL class implements the DPLL algorithm, which uses backtracking, unit propagation, and pure literal elimination to solve satisfiability problems. The algorithm is efficient and can handle large knowledge bases.

# **4. Testing**

## **4.0.1 Horn Testing**

The system has been tested with Horn clauses, which are a special form of propositional logic often used in forward chaining. Horn clauses ensure that each rule has at most one positive literal, which simplifies the process of inference.

## **4.0.2 General Logic Testing**

We also tested the system with general propositional logic, using both simple and complex KBs. The inference methods were validated to ensure they work correctly across a wide range of scenarios.

# **5. Features/Bugs**

## **5.1 Features**

* **Multiple Inference Methods**: Users can choose from different methods depending on their needs.
* **File Input**: The system reads from text files, making it easy to input new KBs and queries.
* **Recursive and Iterative Inference**: The system combines recursive and iterative approaches for effective reasoning.

## **5.2 Bugs**

* **Error Handling**: The system could benefit from better error handling for malformed input files. Currently, it may crash or fail silently when it encounters invalid formats.
* **Scalability**: The Truth Table method doesn’t scale well for larger KBs due to its exponential time complexity.

# **6. Research**

## **6.1 Information about Research Component**

The research component of the project focused on comparing the performance of different inference methods. The aim was to understand their strengths and weaknesses, especially in terms of efficiency and scalability.

## **6.2 General Propositional Logic and DPLL**

DPLL is a well-established algorithm used for propositional satisfiability. By applying optimizations like unit propagation, it can handle much larger logical formulas than simpler methods like Truth Table.

## **6.3 DPLL Implementation**

The DPLL algorithm was implemented with backtracking and optimizations to efficiently handle complex logical formulas. This method was particularly useful for large KBs, where other methods would be too slow.

# **7. Student Contributions**

Each team member played a key role in the project. Some focused on implementing specific inference methods, while others worked on testing, integration, and documentation. Working together, we were able to create a well-rounded and functional system.

# **8. Conclusion**

This propositional logic inference system successfully implements four different reasoning methods: Truth Table, Forward Chaining, Backward Chaining, and DPLL. These methods cater to different problem sizes and complexity levels, providing a balanced mix of completeness and efficiency. The project also offered valuable insights into the trade-offs between various inference techniques, helping us appreciate the strengths and weaknesses of each approach.

# **9. Acknowledgements/Resources**

We used various resources, including textbooks on logic and automated reasoning, online articles, and research papers on propositional satisfiability. Python was the primary language used for implementation.

# **10. References**

1. Davis, M., Putnam, H., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM, 5*(7), 394–397.
2. Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson.