COS30019 - Introduction to Artificial Intelligence

Assignment 2: Inference Engine

Date: 24/05/2024

Contributing Authors:
Nathan Trung (103885695)
Mark Saleh (103994313)

TABLE OF CONTENTS

1. INSTRUCTIONS

In order to run the program, there a few prerequisites that are to be fulfilled;

1. Knowledge Base File, in order to run the Inference Engine, there must be a text file to run it on. Within the text file a 'TELL' section and 'ASK' section is defined.
    a. TELL entails knowledge statements and can be separated by semicolons
    b. ASK entails a single query statement
2. Place this file in the iengine directory
3. Run the Program, within the working directory of iengine.py as well as accompanying text file. Within the terminal, run the following command:
    py iengine.py <filename> tt
    (tt may be replaced with fc or bc)

2. INTRODUCTION

2.1 Inference Engine Introduction

Within any A.I system, an Inference Engine drives logical reasoning as well as problem solving by applying logical rules to obtain new information as well as make decisions from a given knowledge base. By employing various inference methods such as forward chaining, backward chaining or a combination of both.

Within a repetitive the Inference Engine operates by matching rules with existing data, whilst selecting applicable rules and executing these to generate new data. This process will reiterate until a specific objective is met or until no further inference can be drawn.

Inference Engines are utilized within Expert Systems, the being specialized A.I systems tailored to address complex real-world such as medical diagnosis and financial planning. Inference Engines also serve within diverse applications such as natural language processing, data mining as well as autonomous vehicle navigation.

An A.I system's Inference Engine's performance is heavily dependent on the efficacy, robustness and adaptability. Thus, it is crucial that the design of Inference Engine to be thorough and adequate. In doing so, the system is able to reason effectively and adapt to evolving circumstances similar to a human expert in the respective field.

2.2 Connective Sets

During the development of Assignment 2, the following symbols have been used to indicate logical connectives used in propositional logic:

- ~, Negation (¬)): NOT
- &, Conjunction (^): AND
- ||, Disjunction (v): OR
- =>, Implicitation (=>): Logical Conditional (then)
- ⇔, Biconditional (<=>): Logical Biconditional

2.3 Truth Table

In propositional logic, truth tables are used to represent truth values of logical expressions for all possible combinations of input values. Each row of a truth table will correspond to a unique combination of truth values for the given input variables, with the resulting truth value of the logical expression being shown in the final column.
Truth tables provide a method for determining truth values of complex logical expressions and are essential tools in understanding the behavior of logical operators

2.4 Models of a Sentence

Model of a sentence is an interpretation/assignment of truth values to the sentence's atomic propositions, thus making a sentence true.
A model represents a possible state of affairs in which a sentence is true; these are often used in semantic analysis to evaluate the validity or satisfiability of logical formulas.
Understanding models are fundamental to various areas of logic, playing a crucial role in the study of logical systems and their applications.

3. INFERENCE METHODS

Within A.I, Inference Methods are tools for logical reasoning and problem solving. Among these methods that are implemented and discussed are forward chaining and backward chaining, both being prominent techniques where each offers distinct approaches to derive new insight from existing data. Understanding these methods is crucial to designing an A.I system capable of effective decision-making and problem-solving across various domains.

3.1 Forward Chaining

Forward Chaining is a inference method employed within A.I systems to derive new conclusions from existing data. Operating on the principle of iteratively applying logical rules to existing data until a desired goal is reached. By following process;

1. Starting Point, begins with a set of data stored within a Knowledge Base. Serve as a foundation for the inference process.
2. Rule Application, system will then apply logical rules to the data, these defining relationships and conditions. This allows the system to make logical inferences.
3. Iterative Process, proceeding iteratively with each application potentially leading to the derivation of new information. System will continue to apply rules and update its Knowledge Base.
4. Goal Achievement, iterative process continues until a specific goal is achieved, or until no further rules can be applied (all possible conclusions derived).

Forward Chaining has seen utilization within various domains and expert systems that include diagnostic reasoning and decision support systems. It is particularly useful in scenarios where data is abundant and the goal is to derive new insights or solutions.

3.2 Backward Chaining

Backward Chaining is another widely used inference method within A.I, particularly suited for scenarios where the desired goal is known in advance. Operating by working backwards from the goal to determine the sequence of steps required to achieve the goal. By following the process;

1. Goal Identification, begins with the identification of a specific goal that the system aims to achieve. This goal is typically defined.
2. Rule Application, system will then apply logical rules in reverse, from the goal backwards to determine factors leading to its fulfillment.
3. Iterative Process, continues the iterative process until the system identifies necessary conditions required to achieve the goal.
4. Prerequisite Determination, the system identifies conditions necessary for a goal to be satisfied.

Backward Chaining is commonly used within planning and scheduling applications as well as troubleshooting scenarios. Being particularly effective in scenarios where focus is on identifying causes leading to specific outcomes.

3.3 Truth Table

Truth Tables are an inference method which creates a table of every possible combination of truth values to check whether a certain query can be derived from the knowledge base. It follows this process:

1. Initialisation, identifying all unique symbols
2. Model generation, creates a table of all possible true/false values for each symbol. This results in 2^n combinations for n symbols.
3. Evaluation, checks whether each model satisfies the clauses in the knowledge base.
4. Query verification. If a model is found that satisfies all the clauses, the algorithm checks whether the query is true or not in this model. If it is, the query is inferred. If not, another model is checked, until either one works or they are all depleted

Truth tables are more exhaustive than forward and backward chaining, which allows them to be used in situations where all combinations must be checked. However, this method is computationally expensive and much less scalable than the others.

4. IMPLEMENTATION

4.1 Forward Chaining

Below is some brief pseudocode reflecting our implementation of Forward Chaining:

```
class ForwardChaining:
    def fc_check(self):
        agenda = the consequent of every clause in the knowledgebase if it
has no antecedents
         entailed = an empty set for now (this will track symbols that are
searched)
```

```
        while there are symbols in the agenda:
             p = the first symbol in the agenda (pop() removes it from the
agenda)
            append p to the entailed array

            for every clause in the knowledgebase:
                if p is in the antecedents of that clause:
                        if all antecedents of that clause are in entailed
(meaning all the antecedents of that clause have been searched):
                    append the consequent of that clause to the agenda

            if p is the query:
                return True (we found the query!)

          if after searching all the provable symbols you cannot find the
query, return False
```

Essentially, the code begins with the sentences that do not have any antecedents, that is the symbols that can be directly proven. Those are added to the agenda, which is an array of symbols that are known to be true, but that have not yet been searched for in the knowledgebase. It then takes the first symbol in the agenda and cycles through every sentence that does have antecedents, checking whether that symbol is in the antecedents of the sentence. If it is, and all the antecedents of that sentence are entailed, its consequent is assumed to be true, and added to the agenda. This is repeated until the query is proven.

4.2 Backward Chaining

Below is some pseudocode reflecting our implementation of backward chaining:

```
class BackwardChaining:
    def bc_check(self):
         goals = an array of symbols that have to be true for the query to
be true. For now, this is just the query
         entailed = an empty set for now (this will track symbols that are
searched)

        while there are goals:
             q = the first symbol in the goals (pop() removes it from the
goals)
            append q to the entailed array

            for every clause in knowledgebase:
                if the consequent of the clause is q:
```

```
                    for every antecedent of the clause:
                            if the antecedent is not in goals or entailed
(i.e. has not yet been checked):
                            append the antecedent to the goals

                if none of the consequents are q for any clause in the
knowledgebase:
                return False (the query is not provable)

        return True (we proved the query!)
```

Unlike Forward Chaining, the Backward Chaining algorithm begins with its goal, the query. It then searches every clause in the knowledgebase, looking for clauses whose consequents are in the goals. If it finds one, it adds the antecedents of that clause to its goals. It searches through everything in its goals until it finds a symbol that is not provable. If it does not find one, it assumes that the query is proven.

4.1 Truth table

Below is some pseudocode reflecting our implementation of the Truth Table:

```
class TruthTable:
    def tt_check_all(self):
        symbols = list all the symbols in the knowledgebase
        return _tt_check_all the symbols with an empty model

    def _tt_check_all(self, symbols, model):
        if there are no symbols left:
            if _pl_true(all clauses in the knowledgebase, model):
                    return self._pl_true(query, model), 1 (check if the
current model works, and return the result)
            else:
                return True, 0

        first = the first symbol, rest = the rest of the symbols
        model_true = a copy of the model with first set to True
        model_false = a copy of the model with first set to False

        true_count = _tt_check_all the rest using model_true
        false_count = _tt_check_all the rest using model_false

             return (true_count[0] and false_count[0]), true_count[1] +
false_count[1]
```

```
    def _pl_true(self, clauses, model):
        for clause in clauses:
            if clause is an instance of Clause:
                 antecedents_true = all antecedents of the clause are True
in the model
                 consequent_true = the consequent of the clause is True in
the model
                if antecedents_true but not consequent_true:
                    return False
            else:
                if the clause is not True in the model:
                    return False
        return True
```

The TruthTable class checks every possible combination of truth values for the symbols to determine whether a given query is true. tt_check_all initialises the algorithm by creating a list of all the symbols and calling the recursive function _tt_check_all with an empty model. _tt_check_all splits the current list of symbols into a first one and the rest (similar to the pop() function in the other algorithms). It then creates two new models, one where the first symbol is true and one where it is false. When there are no more symbols, it checks whether the current model satisfies its clauses, and then returns the result. _pl_true checks whether a model is viable or not. For every clause, if it is a Clause instance, it checks whether its antecedents and consequent are true. If they are not, it returns False. If it has checked every clause and none of them are False, it returns True.

5. TESTING

5.1 test_HornKB.txt

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
Output
TT - YES: 3
FC - YES: a, b, p2, p3, p1, d
BC - YES: d, p3, p1, p2

5.2 test1.txt

TELL
a => b; b =>c; c => d; d;
ASK

d
Output
TT - YES: 4
FC - YES: d
BC - NO
The BC code gets tricked when the query is already given as a fact.

5.3 test2.txt

TELL
a => b; b =>c; c => d; e=>g;l=>m; a;
ASK
d
Output
TT - YES: 9
FC - YES: a, b, c, d
BC - YES: d, c, b, a

5.4 test3.txt

TELL
a => b; b => c; c => d; d => e; e => f; g;
ASK
f
Output
TT - NO
FC - NO
BC - NO
They all returned the right output, since f is not inferable from g. In fact, g does not infer anything.

5.5 test4.txt

TELL
a&b => c; c = > d; d => e; e => f; a;
ASK
f
Output
TT - NO
FC - NO
BC - NO
They all returned the right output, since b is not inferable and thus neither are c, d, e or f.

5.6 test5.txt

TELL
a&c=> b; b&d => c; c&a => d; d&b => a; a; b;
ASK

d
Output
TT - NO
FC - NO
BC - YES: d, a, b, c
The BC is wrong here. c and d are not inferable because they depend on each other to be true. The BC is programmed to assume that a symbol is unprovable if it is not the consequent of any clause, so it was "tricked" by circular logic.

5.7 test6.txt

TELL
a=>b;b=>a;
ASK
a
Output
TT - NO
FC - NO
BC - YES: a, b
Here is a much simpler example of the circular logic that tricks BC. In this case, no facts were even given.

5.8 test7.txt

TELL
a&b=>c;c&d=>e;a&c=>d;b;a
ASK
e
Output
TT - YES: 1
FC - YES: b, a, c, d, e
BC - YES: e, d, a, c, b

5.9 test8.txt

TELL
a&b&c&d&e&f&g&h&i&j&k=>l;l
ASK
a
Output
TT - NO
FC - NO
BC - NO
The BC result was surprising.

5.10 test9.txt

TELL

ASK
Output
TT - NO
FC - NO
BC - NO

5.11 test10.txt (query not in any clause)

TELL
a&b=>c;c&d=>e;a&c=>d;b;a
ASK
e
Output
TT - NO
FC - NO
BC - NO

5.12 Blank text file

Output
TT - NO
FC - NO
BC - NO

5.13 Method not given

Output
Usage: python iengine.py <filename> <method>

5.14 Non-existent method

Output
Invalid method

5.15 Non-existent text file

Output
FileNotFoundError: [Errno 2] No such file or directory

5.16 Multiple methods given

Output
The method used is the first one

6. FEATURES/BUGS/MISSING

The features implemented were Forward Chaining, Backward Chaining, and Truth Tables. Although these all work, there is one notable bug with the Backward Chaining. It was coded under the assumption that an unprovable symbol is one that is not the consequent of any statement. However, we discovered during the testing phases that this is not necessarily so; a symbol can also be unprovable if it is its own prerequisite. Thus, circular logic (such as a=>b;b=>c) tricks our BC code.

We attempted the research project, but our implementation of it was extremely buggy and interfered with the normal FC, BC and TT functions, so it was decided to leave it out of the final code. At the last minute, we decided to use our earlier, more basic code since it was functional. Besides the research project, there does not appear to be anything missing from our project.

7. TEAM SUMMARY REPORT

Below is detailed the contribution between the team members, Nathan & Mark;

- Nathan 50%
    - Backward Chaining
    - Forward Chaining
    - FileReader
    - Report
- Mark 50%
    - Inference Engine
    - Truth Table
    - Sentence
    - KnowledgeBase

8. CONCLUSION

In concluding our exploration of inference methods, the consideration of search algorithms is essential as this will impact the performance of A.I systems. When addressing problems that involve logical reasoning and inference, selecting the most suitable search algorithms becomes crucial for efficient problem-solving.

For problems where Forward Chaining is employed, a search algorithm that explores the space of logical rules is essential. With this in mind, a Depth-First Search (DFS) may be a suitable search as with it's nature of exploring a branch depth first, this is well aligned with the iterative nature of Forward Chaining. Allowing the system to traverse through possible rule applications and inferred conclusions effectively.

Contrarily, where Backward Chaining is employed, a search algorithm that navigates from the goal to start identifying conditions of this path is preferred. Here once again Depth-First search can be of value, especially if goal is well-defined and a system needs to backtrack efficiently to identify necessary conditions for goal fulfillment.

To further enhance performance, several strategies incorporating heuristics can be employed to guide the search process, this leading to more informed decisions and potentially reducing search time. By

prioritizing the application of rules based on their relevance and importance can accelerate the inference process.

In essence, the choice of search algorithm and accompanying optimization strategies should be tailored to specific characteristics and requirements of the problem at hand. By utilizing appropriate search algorithms and performance enhancing techniques, A.I systems can achieve more efficient and effective reasoning and problem-solving outcomes within inference-based tasks.

9. ACKNOWLEDGMENTS/RESOURCES

Below is listed resources that have been utilized and a concise description of how they assisted during the development of the assignment;

- https://docs.python.org/3/ - Python Documentation referred to during implementation for programming concepts and syntax
- https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf - AI: A Modern Approach referred to for theoretical understanding and implementation guidance
- https://www.run.ai/guides/machine-learning-inference/inference-engine, https://en.wikipedia.org/wiki/Inference_engine, https://deepgram.com/ai-glossary/inference-engine - Several website detailing Inference Engines referred to as research on Inference Engines
- https://builtin.com/artificial-intelligence/forward-chaining-vs-backward-chaining, https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai - Websites detailing Methods referred to as research for understanding of methods

10. REFERENCES

- https://docs.python.org/3/
- https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf
- https://www.run.ai/guides/machine-learning-inference/inference-engine
- https://en.wikipedia.org/wiki/Inference_engine
- https://deepgram.com/ai-glossary/inference-engine
- https://builtin.com/artificial-intelligence/forward-chaining-vs-backward-chaining
- https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai