



COS30018

Inference Engine Implementation Report

Introduction to Artificial
Intelligence




Duc Thang Tran, Hung Gia Tran
103509827, 103509199

Table of Content

Instructions	2
Introduction	2
Inference Engine	2
Truth Table Checking	4
Forward Chaining	4
Backward Chaining	4
WalkSat	4
Implementation	5
Truth Table Checking	5
Forward Chaining	5
Backward Chaining	6
WalkSAT	6
Test Cases	6
Features/Bugs/Missing	9
Features	9
Bugs	9
Research and Extension	9
WSAT	10
Prevent Backward Chaining Infinite Loops	10
Support for Generic Knowledge Base	10
Test Generator	10
Team Summary Report	11
Acknowledgements/Resources	11
Notes	12
References	12

Instructions

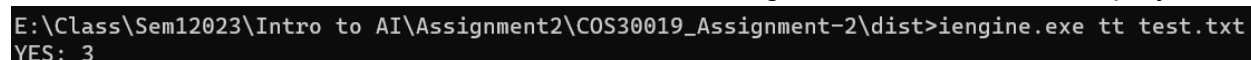
The program can be executed through the command line using the syntax "iengine <method> <filename>". It supports four different algorithms: Truth Table, Forward Chaining, Backward Chaining, and WalkSAT. The selected method can be entered in either lowercase or uppercase, as long as it matches the options in the following list: tt, fc, bc, wsat.



```
COS30019_Assignment-2\dist>iengine.exe tt test.txt
```

Figure 1: Example Input for program

The output of the program will be either "YES" or "NO". Depending on the chosen method, the output will vary when the knowledge base entails the query. For the truth table algorithm, it will display the number of models that satisfy the query. On the other hand, forward chaining and backward chaining will present a list of propositional symbols entailed from the knowledge base during the execution of the algorithm. The only distinction lies in the WalkSAT algorithm, where the program prompts the user to input the maximum number of iterations. If the program discovers a solution within the specified iterations, it will output "YES" along with the order of the iteration in which the solution was found. However, if no solution is found within the given iterations, it will display "NO".



```
E:\Class\Sem12023\Intro to AI\Assignment2\COS30019_Assignment-2\dist>iengine.exe tt test.txt  
YES: 3
```

Figure 2: Example Output for program

Forward and Backward chaining are limited to handling Horn clauses and facts within the knowledge base, restricting queries to single propositional symbols. In contrast, Truth Table and WalkSAT can handle both generic and Horn clauses, allowing for any propositional sentence to be queried. Additionally, generic knowledge bases support negation, disjunction, and biconditional operators, expanding the range of expressiveness beyond Horn clauses. Furthermore, the utilization of parentheses is readily facilitated in generic knowledge bases.

Introduction

Inference Engine

Inference engines have long been an interesting topic in artificial intelligence which can deduce new information from a knowledge base. In this assignment, we focus on four main inference algorithms (i.e., Truth Tables, Forward Chaining, Backward Chaining, and WalkSAT). Truth Table Checking is one of the most fundamental algorithms in the study of logic and computation while Forward and Backward chaining root in propositional logic.

On the other hand, WalkSAT is a local search method designed to tackle the challenge of propositional satisfiability. By delving into these algorithms, we aim to provide an insight into the performance and potential limitations of each algorithm.

In order to efficiently store a logical expression, we employ a postfix sentence representation, which offers significant advantages over alternative methods such as binary expression trees in terms of space utilization and time complexity. While the mentioned data structure exhibits linear time complexity ($O(n)$) for searching tasks, the queue data structure which is embedded in postfix sentences provides constant time complexity ($O(1)$) for insertion, unlike the linear time complexity associated with binary trees.

Moreover, to obtain the postfix sentence from the infix sentence as the input, we employ the Shunting Yard algorithm. This algorithm enables the conversion of any propositional sentence into a postfix sentence. By dequeuing the postfix sentence, we can assign values to the individual elements efficiently.

```

FUNCTION ShuntingYard(infixExpression):
    DEFINE operatorStack AS NEW STACK
    DEFINE outputQueue AS NEW QUEUE

    FOR EACH token IN infixExpression:
        IF token IS AN OPERAND:
            ADD token TO outputQueue

        IF token IS AN OPERATOR (NOT, AND, OR, IMPLIES, IFF):
            WHILE operatorStack IS NOT EMPTY AND
                (top of operatorStack HAS GREATER PRECEDENCE THAN token OR
                 top of operatorStack HAS EQUAL PRECEDENCE AS token AND token IS LEFT ASSOCIATIVE)
                POP operator FROM operatorStack
            ADD operator TO outputQueue
            PUSH token TO operatorStack

        IF token IS '(':
            PUSH token TO operatorStack

        IF token IS ')':
            WHILE top of operatorStack IS NOT '(':
                POP operator FROM operatorStack
                ADD operator TO outputQueue
            IF operatorStack IS EMPTY:
                THROW ERROR "Mismatched parentheses"
            POP '(' FROM operatorStack

    WHILE operatorStack IS NOT EMPTY:
        IF top of operatorStack IS '(' OR ')':
            THROW ERROR "Mismatched parentheses"
        POP operator FROM operatorStack
        ADD operator TO outputQueue

    RETURN outputQueue

```

Figure 3: ShuntingYard Pseudocode

Truth Table Checking

Truth table checking is an algorithm used to evaluate the validity of a logical statement based on a knowledge base. This is done by constructing different scenarios and checking whether the input query is true in that scenario.

The time complexity and space complexity for this algorithm is exponential at $O(2^n)$. This is the case for the time complexity because for each variable in the knowledge base, there are 2^n possible combinations of truth values. The space complexity is also exponential because each combination of truth values is required to be stored in the truth table.

Forward Chaining

Forward chaining is an inference method used to infer a statement based on a knowledge base. The algorithm starts with a set of known facts and applies 'if-then' rules to the clauses in the knowledge base in order to infer additional facts until the goal is reached. The time complexity of the method is $O(n*m)$ in the worst scenario where 'n' is the number of rules and 'm' is the number of known facts. This is due to the need to verify each rule with the known facts. The space complexity is $O(n+m)$ since the algorithm needs to store all the rules and facts for processing.

Backward Chaining

Backward chaining is an inference method, similar to forward chaining, where the algorithm applies logical 'if-then' rules to infer a statement from a given knowledge base. Backward chaining starts at the goal and iteratively backtracks until the facts are found that satisfies the goal.

The time complexity of this method is $O(n*m)$ where 'n' represents the number of rules and 'm' represents the depth of the rule chain. This is because the algorithm starts at the goal and needs to traverse the depth of the rule chain to find the facts needed to satisfy the goal. The space complexity is $O(m)$ where 'm' corresponds to the rule. For each iteration, only 1 rule needs to be stored for processing.

WalkSat

WalkSAT is a local search algorithm that relies on flipping the values of variables that minimize the number of satisfied clauses. The algorithm starts by randomly assigning truth values to all the variables. It then selects an unsatisfied clause and flips a symbol in that clause. The symbol that is chosen is either randomly chosen by a predetermined chance, typically 50%, or the symbol that would maximize the decrease of unsatisfied clauses. This process is then repeated until a model is found to have satisfied the query or the number of iterations has been reached.

The time complexity of this algorithm is not deterministic since it depends on the knowledge base, number of iterations, and chance of flipping the symbol. In the scenario where the number of iterations is infinite and there is no scenario that the knowledge base can entail the query, the algorithm would not terminate. In a scenario where there is a model in the knowledge base that satisfies the query, the time complexity will be $O(2^n)$. The space complexity of the algorithm is $O(n)$ where n is the number of variables. This is the case since the algorithm only stores the unsatisfied clauses and the assignment of truth values of the variables in each iteration.

Implementation

Truth Table Checking

```

function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
            $\alpha$ , the query, a sentence in propositional logic

  symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ 
  return TT-CHECK-ALL(KB,  $\alpha$ , symbols, { })

function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)
    else return true // when KB is false, always return true
  else do
    P  $\leftarrow$  FIRST(symbols)
    rest  $\leftarrow$  REST(symbols)
    return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { P = true })
           and
           TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { P = false })))

```

Figure 4: Truth Table Checking Pseudocode

Forward Chaining

```

function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
           q, the query, a proposition symbol

  count  $\leftarrow$  a table, where count[c] is the number of symbols in c's premise
  inferred  $\leftarrow$  a table, where inferred[s] is initially false for all symbols
  agenda  $\leftarrow$  a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p  $\leftarrow$  POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p]  $\leftarrow$  true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false

```

Figure 5: Forward Chaining Pseudocode

Backward Chaining

```
PerformBC(kb, query) return true or false

if TruthValue(kb, query)
    return true
else
    return false

TruthValue(kb, query) return true or false

if query is a Fact
    return true
for each sentence in KB:
    if query in sentence.Conclusion
        for each symbol in sentence.Premise
            kb.symbolList[symbol] = TruthValue(kb, symbol)

return false
```

Figure 6: Backward Chaining Pseudocode

WalkSAT

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
inputs: clauses, a set of clauses in propositional logic
         p, the probability of choosing to do a “random walk” move, typically around 0.5
         max_flips, number of flips allowed before giving up

model ← a random assignment of true/false to the symbols in clauses
for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause ← a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
return failure
```

Figure 7: WalkSAT Pseudocode

Test Cases

To ensure the accurate functioning of our inference algorithms, we have developed a random test case generator designed for Horn clause and generic knowledge base inference. This generator plays a significant role in guaranteeing the correctness and effectiveness of our inference engine, while also providing valuable insights into the performance of each algorithm employed. The primary objective of the test case generator is to validate the behavior and correctness of the inference engine when

presented with various input scenarios. To achieve this, we employ a randomized approach, where the length of the knowledge base and the number of symbols used within it are randomly generated. This enables us to thoroughly explore various potential test cases, ensuring comprehensive evaluation of the inference engine's capabilities. To cover most of the potential test cases, each algorithm is tested with 1000 random test cases. To verify the result generated, we relied on Sympy library in Python which provides the Entail function specifically designed to test whether a query logically entails a given knowledge base.

It is important to note that the current testing methodology does not include checking the number of entailed models in the Truth Table or the propositional symbol list in the Forward and Backward chaining algorithms. The focus of the testing is solely on determining whether a query successfully entails the provided knowledge base. The WalkSAT inference engine tested is set to maximum iteration of 700.

	TT	FC	BC	WSAT
Time(s)	2306.824	55.45	53.34	65.23
Accuracy(%)	100	100	100	82.7

Table 1. Test Result for Horn Clause

Tell	Ask	Result			
		TT	FC	BC	WSAT
$q \Rightarrow S; S \ \& \ u \Rightarrow S; x \Rightarrow u; S;$	u	Pass	Pass	Pass	Pass
$X \ \& \ X \ \& \ X \Rightarrow k; k \Rightarrow C; C; X;$	y	Pass	Pass	Pass	Pass
$M \ \& \ M \Rightarrow I; M \Rightarrow I; O \ \& \ I \Rightarrow M; O;$	M	Pass	Pass	Pass	Pass

Table 2: Horn Knowledge Base Example Test Generated

Upon evaluating the table, it is evident that Forward Chaining (FC) and Backward Chaining (BC) excel in the aspect of time efficiency, significantly outpacing the Truth Table (TT) and somewhat faster than WalkSAT (WSAT). This can be explained by the fact that truth table has time complexity of $O(2^n)$ for every case wherein 'n' denotes the number of symbols present in the knowledge base and query. On the other hand, WalkSAT still has a longer time of operation due to its randomized model creation. As it concludes upon finding a model that entails, there is a possibility of yielding incorrect output since there might exist a model that the knowledge base does not entail the query.

Moreover, WalkSAT carries the risk of reaching its maximum iteration limit, which could lead to a worst-case linear time complexity.

Additionally, the precision of Truth Table, Forward Chaining, and Backward Chaining algorithms is unimpeachable, with each achieving a flawless 100% success rate on Horn clause test cases. Contrarily, WalkSAT, despite its random nature, attains an average accuracy of 82.7%. This figure has been calculated through an aggregation of results obtained over five separate runs of 1000 tests each. Consequently, even though it fails to match the perfect accuracy of its counterparts, it still delivers a commendable high degree of correctness with Horn clauses.

	TT	WSAT
Time(s)	5597.31	503.6
Accuracy(%)	100	17

Table 3: Test Result for Generic Knowledge Base

Tell	Ask	Result	
		TT	WSAT
$Q \vee V \wedge (a \Rightarrow a); \sim Q \wedge \sim V \wedge (a \vee a); V;$	Q	Pass	Pass
$d \wedge U \wedge \sim h; h; \sim h;$	$\sim U$	Pass	Pass
$(a \vee a) \wedge W; e \wedge (a \Rightarrow a) \wedge \sim W \wedge (a \wedge a); e; W;$	$\sim e$	Pass	Pass

Table 4: Generic Knowledge Base Example Test Generated

To collect data, the method is the same with collecting data for the Horn clause scenario which calculates the average number of five separate runs of 1000 tests each. When confronted with generic sentences, both Truth Table (TT) and WalkSAT (WSAT) exhibit increased run times due to the intricate nature of these sentences. Despite its relatively slower operation, the Truth Table method continues to uphold its reliability as an inference engine, maintaining a perfect 100% success rate for generic sentences. On the other hand, WalkSAT witnesses a remarkable increase in running time from an average time of 65.23 seconds in Horn clause cases to 503.6 seconds for generic sentences. Moreover, its accuracy witnesses a sharp decline from the Horn clause scenario, managing a mere 17% success rate for generic sentences. This reduction is a noteworthy

deviation from the algorithm's prior performance. In conclusion, it's clear that both Truth Table and WalkSAT, while equipped to handle generic sentences, perform at a noticeably slower pace compared to their efficiency with Horn clauses. This comparison illustrates the impact of sentence complexity on the speed and accuracy of these algorithms.

Features/Bugs/Missing

Features

All the required features in the assignment requirement have been met.

- The program has a CLI for the user to enter their preferred method and the test file, following the specified syntax as outlined in the requirements.
- The implementation consists of four inference algorithms: Truth Table, Forward and Backward Chaining, and WalkSAT. To enhance clarity and understanding, comments are strategically placed throughout the source code, providing valuable insights into the implementation details.
- Truth Table and WalkSAT are enhanced to be able to deal with generic sentences, extending their capabilities beyond solely Horn clauses.
- The program includes a comprehensive unit test suite that automatically generates test cases for all implemented algorithms. The test cases and their corresponding results can be viewed in the "testGenerate.py" file.

Bugs

One noticeable issue that was seen during the testing process is that WSAT would often produce "YES" while other algorithms such as forward chaining produce "NO". Since WSAT's algorithm functions by shuffling the truth values in the model, it can randomly find a model where KB and the query are both true and return "YES" while not considering that there are such models where KB does not entail the query.

This issue stems from how WSAT algorithm works since it determines that KB entails the query at the first scenario when both KB and query are true and does not consider that there might be other scenarios that KB does not entail the query.

Research and Extension

In addition to the based requirement of the assignment, we have added some features to the program based on our research:

- Implemented the WSAT Algorithm
- Adjust Backward Chaining to prevent infinite loop

- Support for generic knowledge base for WalkSAT and Truth Table
- Carry out extensive testing through the use of our test generator for both Horn clause and generic knowledge base.

WSAT

The WSAT algorithm relies on shuffling the model and symbol in clauses. This algorithm iteratively selects an unsatisfied clause and flips a symbol in that clause to help maximize the decrease in number of unsatisfied clauses. In our implementation, we adjusted the algorithm where it will stop when it encounters a situation where the knowledge base does not entail the query. When using this method, the user will be prompted to enter the maximum number of iterations at the start of the program. This will tell the algorithm the number of iterations it can shuffle the model to check for query entailment.

Prevent Backward Chaining Infinite Loops

During our testing, we would come across scenarios where backward chaining would be stuck in an infinite loop. This problem would often occur when the conclusion of a clause appears in the premise of another sentence. This causes the algorithm to go back and forth between the 2 sentences infinitely. To solve this, we keep a list of explored symbols and make an argument for the recursive function. The list is updated whenever a symbol is inferred. This allows the algorithm to avoid clauses that would cause it to go in an infinite loop. If the iteration limit has been reached, the algorithm will return “NO” as the answer. If a match is found, the algorithm would stop and return “YES” along with the iteration that it is at.

Support for Generic Knowledge Base

To support a generic knowledge base, we converted infix sentences from the output to postfix form with the help of the Shunting Yard algorithm.

Test Generator

Our test generator was designed to maximize randomness by incorporating up to 40 sentences from each knowledge base, allowing us to control the degree of randomness through probability. This control extends to both sentence length and symbol variation. Although the generic test generator is more complex than the horn clause due to the increased number of possible scenarios, we made an effort to cover as many cases as possible.

To execute the tests, we utilized the unittest library provided in Python. This library facilitated data collection by automatically returning the execution time for each test. Additionally, it enabled us to easily calculate the accuracy of each algorithm. Furthermore, the Sympy library offers a robust toolkit for checking the entailment of a knowledge base to a query, regardless of their respective formats.

However, before feeding the input into the Sympy library, we needed to process the string to ensure that all operators are transformed into a format that Sympy can handle. The implementation details of the test generator can be found in the "testGenerator.py" file. To execute the test, simply run this Python file.

Team Summary Report

Component	Thang	Hung
Knowledge Base	50%	50%
Postfix Operations	55%	45%
Truth Table Model Checking	60%	40%
Forward Chaining	50%	50%
Backward Chaining	45%	55%
WalkSAT	40%	60%
Testing	50%	50%
Report	50%	50%
Overall Contribution	50%	50%

Communication was made primarily through Facebook where each team member discussed each other's ideas on the implementation of the algorithms and the work distribution. In addition, the GitHub is regularly updated so each team member can see the process of the project. Both team members were satisfied with each other and their work throughout the assignment.

Acknowledgements/Resources

Russell, S.J. and Norvig, P., "**Artificial Intelligence: A Modern Approach**," 3rd edition, Prentice-Hall, 2010 (or, 4th edition, Pearson, 2020).

The book gives us an overview of how the inference engine functions and provides an idea of the implementation of each algorithm.

Notes

In our implementation of Forward Chaining, we used a queue to store the symbols. In addition, the pseudocode for WalkSAT doesn't account for instances where a randomly detected world does not entail a query according to the knowledge base. Instead, it only returns false when it hits the maximum iteration without identifying a model that fulfills the necessary conditions. Therefore, in our WalkSAT implementation, we have adjusted the program to return false if it stumbles upon a scenario where the knowledge base fails to entail a query. Consequently, the program terminates immediately following this detection. Finally, we have a minor optimization for Forward Chaining: at the beginning, we check if the query is a known fact in the knowledge base. If it is, we can immediately return true without the need for further inference.

References

Russell, S.J. and Norvig, P., "***Artificial Intelligence: A Modern Approach***," 3rd edition, Prentice-Hall, 2010 (or, 4th edition, Pearson, 2020).