Assignment 2: Inference Engine

Kien Nguyen, Tuan Doan

May 2023

Contents

1 Student Detail

Group ID: COS30019 _A02 _T013

Student 1
Name: Nguyen Trung Kien
ID: 103506132

Student 2
Name: Anh Tuan Doan
ID: 103526745

2 Introduction

The report is written to provide information about the Inference Engine Assignment. It will convey knowledge about the algorithms developed in the program. The remaining sections of the report are "Features", "Basic Algorithms", "Research/Additional Algorithms", "Test Cases", "Reference" and "Team Summary Report". The "Feature" section provides a list of all features that are implemented in the program. While "Basic Algorithm" conveys knowledge about the required algorithms such as "Forward Chaining Algorithm", "Backward Chaining Algorithm", and "Truth Table Algorithm", the "Research/Additional Algorithm" mentions the additional works and extension that has been carried out in the project. In the "Test cases" part, the report provides test cases used to evaluate the algorithms. Finally, the last section aims to acknowledge the external resources used in the paper and summarize how the tasks are divided among each team member.

3 Features

- Truth Table Algorithm that can work with both Horn clauses and generic clauses.
- Forward Chaining Algorithm
- Backward Chaining Algorithm
- Resolution Algorithm
- DPLL Algorithm
- General knowledge base handling and converting generic clauses to CNF

4 Basic Algorithms

4.1 Truth Table Algorithm

One of the simplest algorithms to check propositional logic entailments is the Truth Table Algorithm. The algorithm works by enumerating all possible models of the knowledge base and checking if they are also the model of the query. Especially, the algorithm can handle not only the Horn clauses but also all the generic clauses. This algorithm is sound and complete. However, the major downside of this algorithm is its runtime performance. The program has to find all the possible models which are 2^n models for n symbols. Additionally, there are algorithms that also use the same technique but have better runtime performance such as the DPLL algorithm that will be discussed in the Additional Algorithms/Research section. Below is the pseudocode for the Truth Table Algorithm:

Algorithm 1 Truth Table Algorithm

```
1: procedure TTAlgorithm(kb, query, symbols, model)
2:     if symbol is empty then
3:         if check(model, kb) then return check(model, query)
4:         else
5:             return true
6:     else
7:         p ← symbols[0]
8:         rest ← rest[symbols]
9:         return TTAlgorithm(kb, query, rest, extend(p, true, model)) and
   TTAlgorithm(kb, query, rest, extend(p, false, model))
```

The algorithm starts with a knowledge base, a query, a list of symbols and an empty model. If the list of symbols is empty and all symbols have been assigned a value, the program will check if the model satisfies the knowledge base and the query to return the correct boolean value. If the list of symbols is not empty, then the function will choose a symbol from the list to assign a value and add it to the current model. Then the function will restart recursively with the newly updated model with the new value.

4.2 Forward Chaining Algorithm

Another algorithm for the entailment problem is the Forward Chaining Algorithm. Especially, this algorithm only works with Horn clauses. Horn clauses are either a simple propositional symbol or a conjunction of symbols that implies another symbol. For example, a and a & b =>c are both Horn clauses. The Forward Chaining algorithm runs in linear time and it is both sound and complete. The algorithm's pseudo code is provided below.

Algorithm 2 Forward Chaining Algorithm

```
1: procedure FORWARD CHAINING ALGORITHM(kb, symbols, query)
2:     remove all duplicate clauses
3:     count ← a dictionary with the clauses as the key and the number of
   premises of the clause as the values
4:     inferred ← a dictionary with the symbols as the key and a boolean
   value as the values
5:     agenda ← a list of symbols that are initially known to be true
6:     while agenda is not empty do
7:         if query in agenda then
8:             return true
9:         p ← agenda.pop(0)
10:        if inferred[p] is false then
11:            inferred[p] ← true
12:            for each horn clause c in whose premise p appears do
13:                decrement count[c]
14:                if count[c] = 0 then
15:                    if head[c] = query then
16:                        return true
17:                    agenda.push(head[c])
18:    return false
```

The forward chaining algorithm starts with a list of symbols that are known to be true. Then it goes through the knowledge base to check all the Horn clauses whose premises have been satisfied and add the conclusion of those satisfied clauses to the agenda list. This process will be repeated until the query is found or all items in the agenda list have been explored which indicates that the knowledge base does not entail the query.

4.3 Backward Chaining Algorithm

The backward chaining algorithm is a method used in rule-based reasoning systems and artificial intelligence to determine the set of conditions (facts or rules) that must be satisfied to reach a specific goal or conclusion. Unlike forward chaining, which starts with known facts and works towards conclusions, backward chaining starts with a goal and works backwards to find the necessary conditions.

The algorithm's pseudo code is provided below:

Algorithm 3 Backward Chaining Algorithm

```
procedure BACKWARD CHAINING ALGORITHM(knowledge_base, goal)
    if goal is already known in knowledge_base then
        return true
    for all rule in knowledge_base do
        if goal can be inferred from rule then
            for all subgoal in rule's premises do
                if ¬BACKWARD CHAINING ALGORITHM(knowledge_base, subgoal) then
                    return false
            add goal to knowledge_base
            return true
    return false
```

In essence, backward chaining works by starting with a goal and working backward through the knowledge base to determine the necessary conditions or facts that must be true to achieve that goal. It recursively explores the antecedents of rules until it reaches a point where the antecedents are either known or given facts.

Backward chaining is commonly used in expert systems, inference engines, and theorem proving. It is particularly useful when you have a specific goal or conclusion in mind and want to determine the conditions or facts required to reach that goal. The algorithm returns a boolean value that indicates if the query clause is verified based on the given knowledge base.

5 Research/Additional Algorithms

5.1 Introduction

This section will introduce more advanced algorithms and additional research that have been implemented in the program. The research aims to answer some given problems below:

- How to parse generic clauses that are not in Horn form?
- How to convert all the clauses from general to CNF form?
- How to implement the Resolution and DPLL algorithm?

To answer these interrogative queries, the research will provide details about the process of developing these features of the program, as well as carrying out some testing to evaluate the final work.

The remaining part of the report will have 5 sections, which are "Methodology", "Result", "Discussion" and "Conclusion". The "Methodology" section explains how the research is carried out and the "Result" section will talk about the final work of the program, as well as the testing of the results. Finally, the

"Discussion" and "Conclusion" parts will have a further evaluation on the work, and summarize the research respectively.

5.2 Methodology

5.2.1 Problem

To develop the program, the application must solve the problems related to parsing the input, converting the input into the correct knowledge base forms, and generating test cases to evaluate the algorithms. With regard to the first concern, the inputs provided to the program can be generic clauses rather than Horn clauses. This means that the parsing function has to cover a variety of cases which includes many different logical operators and parentheses. To be more specific, the evaluation's sequence of the clause can be affected by the parentheses as well as the precedence of the logical operators. Therefore, the program must be able to generate clauses that maintain the correct order. Another point that needs to be mentioned is the process of converting the general clauses to Conjunctive Normal Form clauses. This form can be described as the conjunction of disjunction of literals clauses (e.g. (A or B) and (C or D)). To convert the generic clause into CNF we can apply the logic laws such as distributive and De Morgan's law. Moreover, the function must also be able to tell if a clause is already in CNF. Last but not least, the test cases generator function should be developed correctly to cover all of the edge cases that can appear.

5.2.2 Parsing the input

The program defines two classes PropositionalSymbol and Clause to represent the symbols and clauses respectively. The "Propositional Symbol" class will have 2 attributes, which are symbol (indicating the character), and value (indicating its boolean value), as well as some getters and setters. The "Clause" class will have 3 attributes, which are "left" (indicates the left-side clause), "right" (indicates the right-side clause), and "operator" (indicates the logical operator used in the clause". Simple clauses with just one symbol has the "right" attribute assigned to a PropositionalSymbol object. To represent generic clauses in the program, we implemented a parseClause method that can turn a clause string into a Clause object. The parsing process implemented recursion to loop through the string. The process starts by checking if the current clause string does not contain any operators which means that the clause string only contains a simple symbol. If that is the case, the function will return a simple Clause object with the "right" attribute assigned to the clause string. On the other hand, if the string contains any of the operators, the program will go through the string to find the operator. When it finds an operator, the function will split the string into 2 substrings at the position of the operator: the left substring (from the start to the operator position) and the right substring (from the operator position to the end). The two substrings will also be called with the parseClause function

and be put together in a new Clause object that will be returned. Finally, if the function cannot find any operator, it will strip the most outer parentheses and call the paresClause function again with the new string.

Algorithm 4 Parsing Input

```
procedure parseClause(clauseString)
    if clauseString does not contain any logic operators then
        return Clause(right=clauseString)
    if the clauseString contains <=>and <=>is not in any inner parentheses then
        left ← parseClause(sub string from the start to the operator)
        right ← parseClause(sub string from operator to the end)
        return Clause(left=left, right=right,operator="<=>")
    if the clauseString contains =>and =>is not in any inner parentheses then
        left ← parseClause(the sub string from the start to the operator)
        right ← parseClause(the sub string from operator to the end)
        return Clause(left=left, right=right,operator="=>")
    if the clauseString contains || and || is not in any inner parentheses then
        left ← parseClause(the sub string from the start to the operator)
        right ← parseClause(the sub string from operator to the end)
        return Clause(left=left, right=right,operator="||")
    if the clauseString contains & and & is not in any inner parentheses then
        left ← parseClause(the sub string from the start to the operator)
        right ← parseClause(the sub string from operator to the end)
        return Clause(left=left, right=right,operator="&")
    if the clauseString contains ¬ and ¬ is not in any inner parentheses then
        right ← parseClause(the sub string from operator to the end)
        return Clause( right=right,operator="¬")
    return parseClause(clauseString.strip("()"))
```

5.2.3 Conjunctive Normal Form Converting

The Resolution and DPLL algorithms only work on clauses that are in the Conjunctive Normal Form (CNF). Therefore, we have implemented an algorithm to convert general clauses into CNF. This algorithm works by recursively applying logic laws to transform the clause into conjunctive normal form. Below is the pseudocode for the algorithm:

5.2.4 Resolution Algorithm

Another algorithm to check the entailment is the Resolution algorithm. Unlike the FC and BC algorithms, resolution can work with both Horn clauses and

Algorithm 5 convert generic clauses into CNF

```
procedure convertToCNF(clause)
    if clause is a literal then
        return clause
    if clause.operator = "=>" then
        newClause ← EliminateImplication(clause)
        return convertToCNF(newClause)
    if clause.operator = "<=>" then
        newClause ← EliminateBidirectionalImplication(clause)
        return convertToCNF(newClause)
    if clause.operator = "¬" then
        newClause ← ApplyDeMorgan(clause)
        return convertToCNF(newClause)
    if clause.operator = "||" and clause.right.operator = "&" then
        newClause ← ApplyDistributive(clause)
        return convertToCNF(newClause)
    if clause.operator = "||" and clause.left.operator = "&" then
        newClause ← ApplyDistributive(clause)
        return convertToCNF(newClause)
    newClause ← Clause(right = convertToCNF(clause.right), left =
convertToCNF(clause.left), operator = clause.operator)
    if isCNF(newClause) then
        return newClause
    else
        return convertToCNF(newClause)
```

generic clauses. However, the clauses must be transformed into CNF before the algorithm can be applied. The resolution algorithm works on the basis of the resolution inference rule. For example, given 2 premises: $(a \mid\mid b)$ and $(c \mid\mid \neg b)$, we can draw the conclusion $(a \mid\mid c)$. Especially, resolving a and $\neg a$ gives an empty clause. Below is the pseudocode for the algorithm:

Algorithm 6 Resolution Algorithm

```
procedure RESOLUTION(knowledge_base, query)
    clauses ← knowledge_base & ¬query in CNF
    new_clause ← {}                          ▷ Initialize an empty clause
    while True do
        for all C1 in clauses do
            for all C2 in clauses do
                if C1 ≠ C2 then
                    resolvents ← RESOLVE(C1, C2)
                    if resolvents contains the empty clause then
                        return true                      ▷ Query is entailed
                    new_clause ← new_clause ∪ resolvents
        if new_clause is a subset of clauses then
            return false                                 ▷ No new resolvents
        clauses ← clauses ∪ new_clause
function RESOLVE(C1, C2)
    resolvents ← {}                          ▷ Initialize an empty set
    for all l1 in C1 do
        if C2 contains the negation of l1 then
            resolvent ← Resolve C1 and C2
            resolvents ← resolvents ∪ resolvent
    return resolvents
```

The resolution algorithm is a key component of many automated reasoning systems and theorem provers, including the famous resolution-based theorem prover called "Resolution Refutation." It has applications in various fields, including logic programming, artificial intelligence, and formal verification.

5.2.5 DPLL Algorithm

The DPLL (Davis-Putnam-Logemann-Loveland) algorithm is a complete and efficient algorithm used for determining the satisfiability of propositional logic formulas. The DPLL algorithm is similar to the Truth Table algorithm but it uses a combination of unit propagation and backtracking to systematically explore the search space of truth assignments and determine the satisfiability of the propositional logic formula. The algorithm uses a unit clause which is a clause with only a single literal to simplify the formula. We know that a unit clause must be true for the formula to satisfy; therefore, we can apply these 2 rules to other clauses:

- clauses that contain the literal can be removed
- the negation of *literal* can be removed in clauses it appears in

The algorithm ensures completeness, meaning it will eventually find a solution if one exists, and it can efficiently handle large propositional logic formulas. It is widely used in automated theorem proving, logic programming, and formal verification.

Algorithm 7 DPLL Algorithm

```
procedure DPLL(knowledge_base)
    if knowledge_base is empty then
        return true                                  ▷ All clauses are satisfied
    else if knowledge_base contains an empty clause then
        return false                                 ▷ A contradiction is found
    if knowledge_base contains an unit clause a then
        return DPLL(Simplify(knowledge_base,a))
    p ← select a random symbol from knowledge_base
    if DPLL(Simplify(knowledge_base,p)) then
        return true
    elsereturn DPLL(Simplify(knowledge_base,¬p))
procedure SIMPLIFY(knowledge_base,literal)
    remove clauses in knowledge_base that contains literal
    remove the negation of literal in clauses in knowledge_base
```

The DPLL algorithm has had a significant impact on automated reasoning and has served as a foundation for many SAT solvers (algorithms for the Boolean satisfiability problem). It has applications in areas such as formal verification, artificial intelligence, and logic programming.

5.2.6 Test Generator

To ensure that the developed algorithm provide the expected outcome, each algorithm are tested carefully. There are 3 type of test cases that are required, which are "Horn clause", "General logic", "General clause". While the "Horn clause" and "Generic clause" test cases are generated in the form of "ASK" and "TELL" mentioned in the problem to test the algorithm, the "General Logic" clause will be a list of "general clause", to test the parsing input and converting function of the program.
The test case will be generated randomly. To ensure that the tests generator can provide a wide variety of test cases with high complexity, the generator first define a list of 7 alphabet characters. Then, the program select the items in the list, combine with the operator which are also generate randomly to form a clause. The clause is then appended to the list to be reused to form a new clause

with complexity hierarchy. Moreover, the function also creates some option to control the complexity and number of test cases for a more flexible usage.

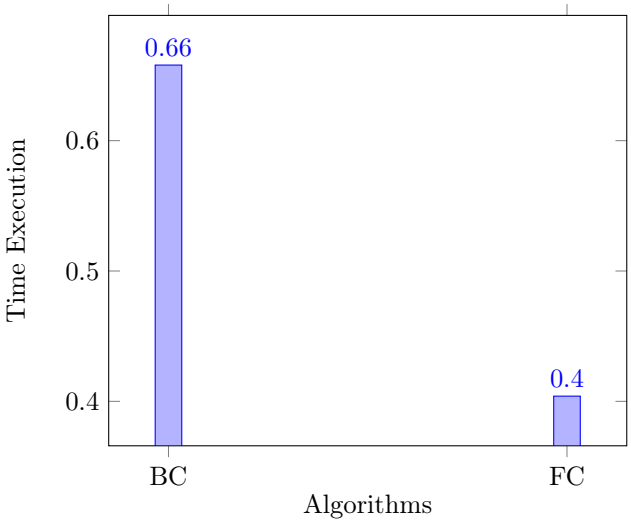
5.3 Result

Each feature of the application is tested with a variety of test cases. The result of the test cases will be visualized, evaluated, and compared.

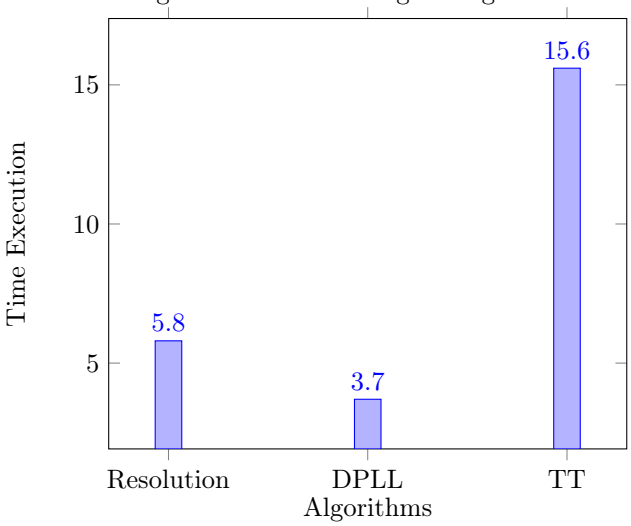
The bar graph below provides the time execution of Forward Chaining Algorithm and Backward Chaining Algorithm after running through 20 test cases.

Test Case	**Algorithm**				
	Algorithm 1	**Algorithm 2**	**Algorithm 3**	**Algorithm 4**	**Algorithm 5**
1	Pass	Pass	Pass	Pass	Pass
2	Pass	Pass	Pass	Pass	Pass
3	Pass	Pass	Pass	Pass	Pass
4	Pass	Pass	Pass	Pass	Pass
5	Pass	Pass	Pass	Pass	Pass
6	Pass	Pass	Pass	Pass	Pass
7	Pass	Pass	Pass	Pass	Pass
8	Pass	Pass	Pass	Pass	Pass
9	Pass	Pass	Pass	Pass	Pass
10	Pass	Pass	Pass	Pass	Pass
11	Pass	Pass	Pass	Pass	Pass
12	Pass	Pass	Pass	Pass	Pass
13	Pass	Pass	Pass	Pass	Pass
14	Pass	Pass	Pass	Pass	Pass
15	Pass	Pass	Pass	Pass	Pass
16	Pass	Pass	Pass	Pass	Pass
17	Pass	Pass	Pass	Pass	Pass
18	Pass	Pass	Pass	Pass	Pass
19	Pass	Pass	Pass	Pass	Pass
20	Pass	Pass	Pass	Pass	Pass

Table 1: Results of Testing with 20 Test Cases



The bar graph below provides the time execution of DPLL Algorithm and Resolution Algorithm after running through 20 test cases.



5.4 Discussion

There are some points that can be further discussed. The most important of which is the usage of Foward Chaining and Backward Chaining. While both algorithms are used in rule-based reasoning systems, the difference in the direction of reasoning makes both each algorithm become more outstanding than the other in some scenario. To be more specific, Forward Chaining Algorithm is more efficient when there are many facts and few rules, since it will reduces the redundancy of reasoning. This is because Forward Chaining Algorithm uses all

of the rules given in the knowledge base . On another hand, Backward Chaining Algorithm is more effective in the scenario where there are many facts and few rules, as it focuses on the goal and determines the necessary conditions to satisfy it.
Another point that should be considered is the use case of DPLL and Resolution algorithm. Although both algorithms are used in theorem proving and logical reasoning, each algorithm has different purpose and characteristic. While the DPLL algorithm is primarily used to determine the satisfiability of propositional logic formulas. Moreover, with regard to the efficiency, DPLL algorithm shows its advantage since it implements a variety of optimization techniques, such as conflict-drive clause learning as well as suitable data structure. However, although Resolution is a complete algorithm, DPLL Algorithm does not guarantee to output a result in every solution.
Last but not least, with regard to the future enhancement, there are some techniques than can be applied to improve the performance of the algorithms above. Since both algorithms is a type of search algorithm, heuristic can be applied to reduce the search space of the algorithm; Therefore, the search process can be speed up. In additional, the probability aspects can also be considered so that the algorithm's can be enhanced.

6 Test Cases

Test case	ASK	TELL
Test case 1	(b||g);(a=>b); (f=> e); (f=> a); (a=>d); ((a=>b)=> b);	b
Test case 2	(b=> d); (c=>f); ((c=>f)& d); ((b=> d)&c); (b& g); ((b=> d)=> d);	c
Test case 3	(f& a); (f||d);(g&(f||d)); ((f||d)||g); (c&b); ((f||d)=> g);	f
Test case 4	(c& b); (f&b); (f||(f&b)); ((f&b)|| d); (g&a); (f=>(c& b));	g
Test case 5	(g&d); (a&(g&d)); ((g&d)||d);(c=>e); ((g&d)& e); (e=> (c=>e));	g

Table 2: 5 samples test cases of 2500 general test cases

Test case	ASK	TELL
Test case 1	b =>d; b =>d; a&e&b =>d; b&d =>a; a&d =>e; c; d;	b
Test case 2	b&c =>a; b&c =>a; c =>d; a =>c; b =>c; d&b&c =>a; b; d;	c
Test case 3	b&d&e =>a; a&d =>b; b =>a; a&e =>d; b&c&e =>d; a; d;	b
Test case 4	a =>e; b&e =>a; a&c =>e; a&e&c =>d; a&e =>c; b; e;	d
Test case 5	c&a =>b; e&c =>d; b =>d; b&a&c =>e; c&b =>d; e&c =>d; a; b; e;	c

Table 3: 5 samples test cases of 2500 "Horn form" test cases

7 Note

Instruction for the program: To use the program, the user must specify 2 arguments: iengine [method] [filename]. The first argument is the method that will be used and the available methods are TT, FC, BC, RES, and DPLL. The filename is the path to the knowledge base. All of the algorithms can deal with Horn clauses while only TT, RES, and DPLL can work with generic clauses.

8 Team Summary Report

Task	Team Member's Name
Truth Table Algorithm	Tuan Doan
Forward Chaining Algorithm	Tuan Doan
Backward Chaining Algorithm	Kien Nguyen
Resolution Algorithm	Tuan Doan
DPLL Algorithm	Kien Nguyen
Convert general clause to CNF form clause	Kien Nguyen, Tuan Doan
File parsing	Tuan Doan
Tests generator	Kien Nguyen
Testing	Kien Nguyen

9 Acknowledgements/Resources

- Sympy
  Sympy is an open-source library that provides features for symbolic mathematics. In this assignment, Sympy is used for testing the result of developed algorithm

- unittest
  "unittest" is the library used for carrying out the testing in this assignment.
- Article "Difference Between Backward Chaining and Forward Chaining - Javatpoint"
  This article compares the Forward Chaining Algorithm and Backward Chaining Algorithm. It clarifies the distinction between the two algorithms mentioned.
- Article "Resolution Algorithm in Artificial Intelligence - GeeksforGeeks"
  This article provides some insight into the Resolution Algorithm in the research.
- Article "Brief Explanation of DPLL"
  This article provides some insight into the DPLL Algorithm in the research.

References

[1] *Difference Between Backward Chaining and Forward Chaining - Javatpoint*. URL: `https://www.javatpoint.com/difference-between-backward-chaining-and-forward-chaining` (visited on 05/25/2023).

[2] *How does DPLL Algorithm Work? A Brief Explanation of DPLL | by Mert Özlütıraş | Medium*. URL: `https://mertozlutiras.medium.com/brief-explanation-of-dpll-davis-putnam-logemann-loveland-algorithm-663f4a603c1` (visited on 05/25/2023).

[3] Mert Özlütıraş. *Brief Explanation of DPLL (Davis — Putnam — Logemann-Loveland) Algorithm*. Medium. Feb. 18, 2021. URL: `https://mertozlutiras.medium.com/brief-explanation-of-dpll-davis-putnam-logemann-loveland-algorithm-663f4a603c1` (visited on 05/25/2023).

[4] *Resolution Algorithm in Artificial Intelligence - GeeksforGeeks*. URL: `https://www.geeksforgeeks.org/resolution-algorithm-in-artificial-intelligence/` (visited on 05/25/2023).

[5] *Resolution in First-order logic - Javatpoint*. URL: `https://www.javatpoint.com/ai-resolution-in-first-order-logic` (visited on 05/25/2023).