COS30019

Inference Engine

Introduction to Artificial Intelligence

ANH MINH NGUYEN
STUDENT ID: 103178955
DUY KHOA PHAM
STUDENT ID: 103515617

1. Introduction

This report aims to provide a concise overview of the Inference Engine Problem, encompassing various aspects such as algorithms, test case generation, exploration of additional algorithms, and a review of the team's summary report. In addition, the report will introduce five key algorithms, namely Truth Table, Forward Chaining, Backward Chaining, Resolution, and Davis–Putnam–Logemann–Loveland (DPLL). These algorithms are responsible for interpreting a text file comprising a knowledge base along with a query, primarily assessing if the query is inferred from the knowledge base.

2. Features/ Bugs/ Missing

Feature

In this assignment, several features have been implemented to meet the fundamental requirements of the Inference Engine. First, the program will read the input given format by the assignment and parse it into postfix form using the Shunting Yard Algorithm (Rastogi et al., 2015). By doing this, no matter how the complex clauses are nested in the parentheses, the program is still able to read the input file. Afterward, inside this program, there are 3 fundamental logical reasoning algorithms required to implement such as "Truth Table", "Forward Chaining" and "Backward Chaining". The implementation detail of these algorithms will be discussed later. It is important to note that the scope of the assignment restricts the input to Horn Form clauses, which are best suited for Forward Chaining and Backward Chaining. Besides the basic requirements, this report will extend beyond the simple algorithm by researching some techniques, strategies, and algorithms to deal with general form. This is coming from the idea that any general logic clauses can be converted to Conjunctive Normal Form (CNF). By facilitating this idea, this report will outline 2 algorithms which are Resolution and Davis–Putnam–Logemann–Loveland (DPLL) and discuss how performance they are.

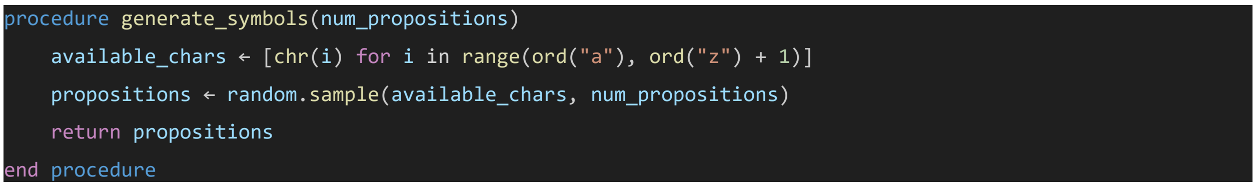
Bugs

The Resolution algorithm can take a significantly longer time to run in complex test cases. This behavior arises from the nature of its implementation, which will be explained later in this report.

Test Cases

Automated Tests Generator

In terms of test cases generator, there are several functions for randomly created test cases. The idea behind this will be first to select a random number of symbols for created clauses, as illustrated in Figure 1, this can be done through the "random" library and "sample" function.

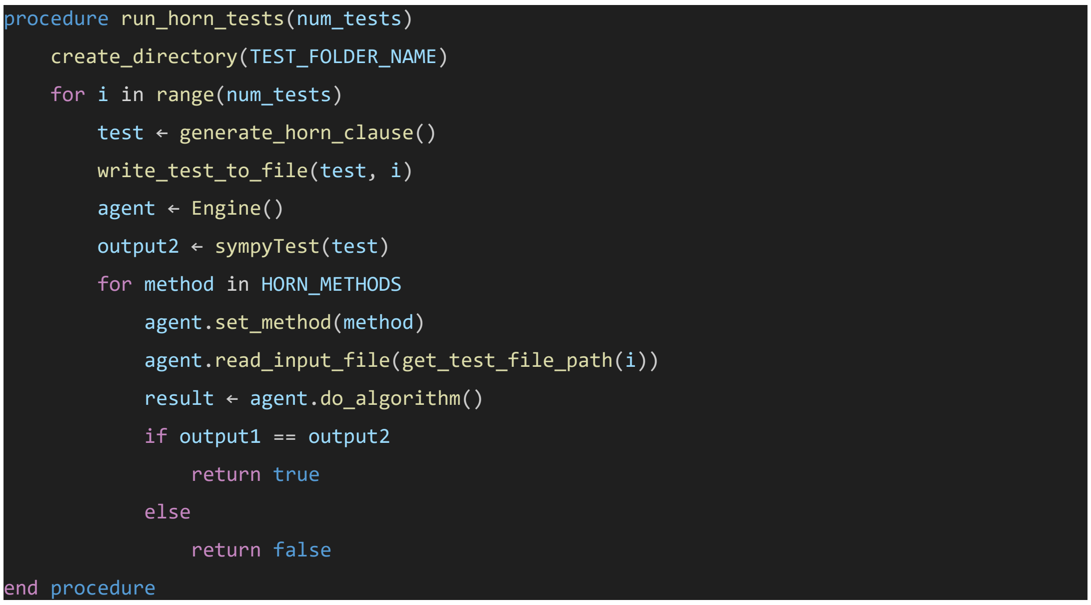


```
procedure generate_symbols(num_propositions)
    available_chars ← [chr(i) for i in range(ord("a"), ord("z") + 1)]
    propositions ← random.sample(available_chars, num_propositions)
    return propositions
end procedure
```

Figure 1. Generate symbol function

The reason for picking the fixed symbols before generating clauses is for increasing the higher chance of getting the output "YES", which means the knowledge base entails the query. Otherwise, most of the test cases will have an output is "NO" leading to the bias in generating random data.
Afterward, Horn form clauses can be created through "generate_horn_clause()" which is depicted in Figure 2. Its primary function is to randomly select fixed symbols, subsequently joining them with the operators "&" (And) and "=>" (Implies). A vital part of this generation process is to add a fact within the knowledge base, especially a standalone symbol. However, a careful balance is crucial as an excess of facts may lead to an overabundance of "YES" output without running any complicated computation. For that reason, the probability of a horn query being a composite clause will be 0.7, and it will have a 30% of appearance as a fact inside the knowledge base. Another consideration in this implementation

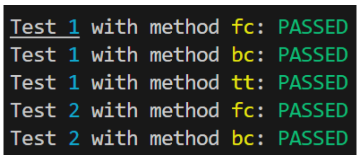
is that only 30% of chance to have a fact, therefore, it is important to avoid this fact to be duplicated, so a “set” has been used to avoid the duplication.
Generating test cases for logical reasoning relates to string manipulation. Hence, the idea for a general logical expression generator is quite like the Horn clause. It is a bit more complicated because of utilizing the recursive code to nest the expression inside the parentheses. It also leverages a parameter called “depth” to control how much depth users want the expression will be nested inside parenthesis; this is corresponding directly to the complexity of the logical clauses.



```
procedure run_horn_tests(num_tests)
    create_directory(TEST_FOLDER_NAME)
    for i in range(num_tests)
        test ← generate_horn_clause()
        write_test_to_file(test, i)
        agent ← Engine()
        output2 ← sympyTest(test)
        for method in HORN_METHODS
            agent.set_method(method)
            agent.read_input_file(get_test_file_path(i))
            result ← agent.do_algorithm()
            if output1 == output2
                return true
            else
                return false
end procedure
```

Figure 2: Pseudocode for auto-running horn test function

In addition to generating random clauses and knowledge bases along with queries, inside the program, it also has an automatic executer to run these tests and examine their accuracy. Thanks to the support of the “sympy” 3-party library, the program can take the output of this library as a standard result and provide any wrong test if the outputs are mismatched. Inside this automatic execution, it will specify several parameters to control the number of tests, the complexity of the clauses, the number of symbols wanted to have, and the folder name and file name to put the test inside in the case outputs of two program mismatches. To utilize this, a folder test will be created automatically, and any inputs, outputs, method names, and test case numbers whose mismatch results will be printed in Figure 3. Besides that, the program also facilitates the terminal to show the test case running status:



```
Test 1 with method fc: PASSED
Test 1 with method bc: PASSED
Test 1 with method tt: PASSED
Test 2 with method fc: PASSED
Test 2 with method bc: PASSED
```

Figure 3: Tests Cases Running Terminal Interface

Example Test Cases

Horn Clause Test

Test Case	Output
TELL r & g => i; i & r => t; i & g => t; i & t => r; g & t => i; g & i & r => t; g => i; r => g; g; ASK r	YES
TELL x => z; z => s; s & w => x; s & x => w; w => s; s => w; w => z; w & z & x => s; z => x; z; ASK	YES

w	
TELL m & r => n; r => m; r => u; u => r; m & u => r; n & u => m; m => n; n & r & u => m; u; n; ASK m	YES
TELL x & q => m; m & x => n; m => n; q => m; q => n; x; q; m; ASK m	YES
TELL s & v => p; c & s & p => v; p => s; c => v; c & s => v; p => c; s => c; v => p; s; p; ASK v	YES

Table 1: 5 Horn Form Test Cases among 1000 test cases which are generated

General Clause Test

Test Cases	Output
TELL ((p || v) & (p => r)); ((r <=> v) => (p & r) => (p & v)); ((v || r) || (v & p) || (p & r)); ((v & p) & (r <=> p) & (r || v)); ASK p	YES
TELL ~(v & v & w); ~~z; ((v & w) || ~w); ((v || w) || ~v); ASK w	NO
TELL ((u & i) || (s <=> u) || (s & i)); ((s & u) => ~u); ((i => s) || (s || u)); ~(u & s); ASK u	NO
TELL ~(y => f); ((d || y) => (d <=> f) => (f & y)); ~(y & f & f); ((d || y) => (d & f) => (y <=> f)); ASK ~(y & f & d)	YES
TELL ((m & w) => (z <=> w) => (z & m)); ~(z & z & m); ~~m; ((m & w) || (z & w) || (z || m)); ASK ((m & w) || (z & w))	YES

Table 2: 5 General Form Test Cases among 1000 test cases which are generated

Comparing Between Algorithms

All results are validated by using the “sympy” library in Python with its “built-in” entail function. Two team members carried out 1000 random Horn form test cases to observe some interesting insights from 3 different algorithms. It demonstrates that three algorithms provide 100% accuracy output. However, the runtime for each algorithm is different. From Figure 4 below, the “Truth Table” algorithm has the highest runtime among the three algorithms. Hence, it can conclude that Backward Chaining and Forward Chaining outperform the Truth Table algorithm in terms of running time. Besides that, the table also shows that Backward Chaining performs than Forward Chaining.

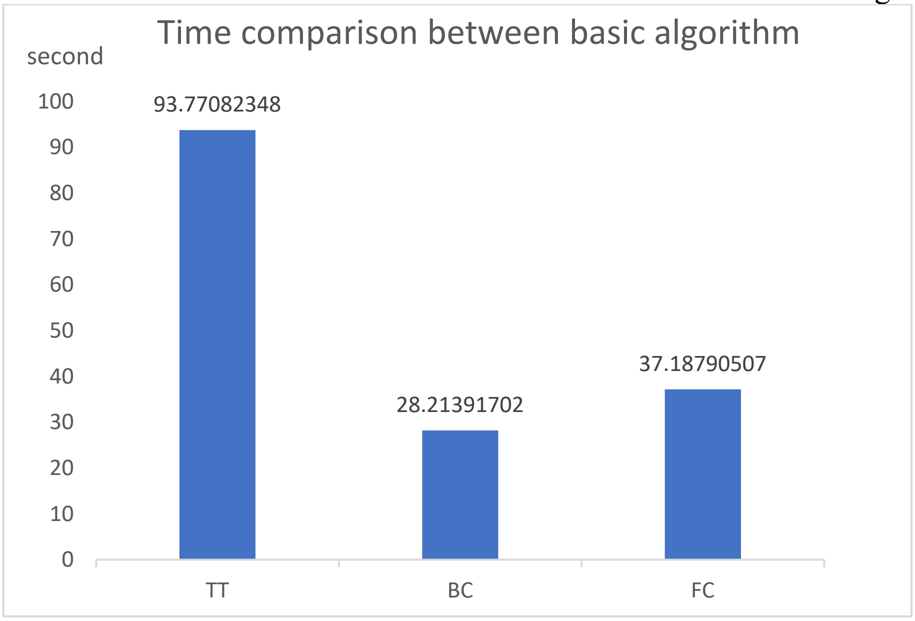


Figure 4: Time comparison between basic algorithm using 1000 horn clause test

3. Implementation Details Of Fundamental Algorithms

Truth Table Algorithm

This algorithm is one of the simplest and most accurate algorithms for logical reasoning. The idea of this algorithm is to check every combination of all clauses and all single symbols. By utilizing recursive code, the algorithm seeks a “model” or a recursive instance where the knowledge base is true. Subsequently, using the “model”, the algorithm will verify whether the query is true or not. If it is true, the algorithm continues to find the next model. However, if the query is false then it outputs “NO”. The advantage of this algorithm is its sound and completeness. However, its drawback is the time complexity. Checking every combination of all clauses and symbols necessitates a runtime of $O(2^n)$ to complete the algorithm. Consequently, a larger number of unique clauses and symbols leads to a more extended execution time for the algorithm. Here is the pseudocode:

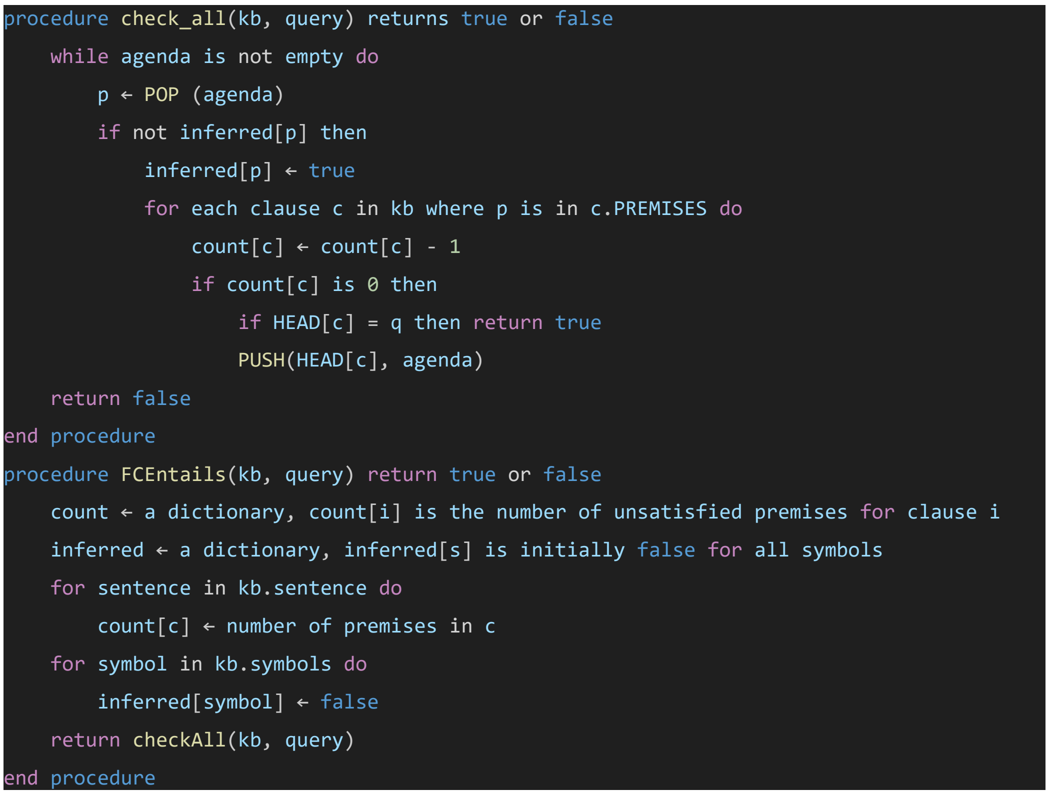
```
procedure TT-Entails(kb, symbols query, model) return true or false
    if symbol is empty then
        if check(model, kb) then return check(model, query)
        else
            return true
        else
        p ← symbols[0]
        rest ← rest[symbols]
    return TT-Entails(kb, query, rest, Extend(p, true, model)) and
            TT-Entails(kb, query, rest, Extend(p, false, model))
end procedure
```

Figure 5: Pseudocode for Truth Table

Within the recursive function, the model will initialize with an empty array, after that it will extract every symbol along with its value to create a model. If the model is full or the symbol list is empty, the model is fed into the knowledge base to verify if it satisfies the criteria. If it does not return true, it proceeds to verify whether the knowledge base entails the query.

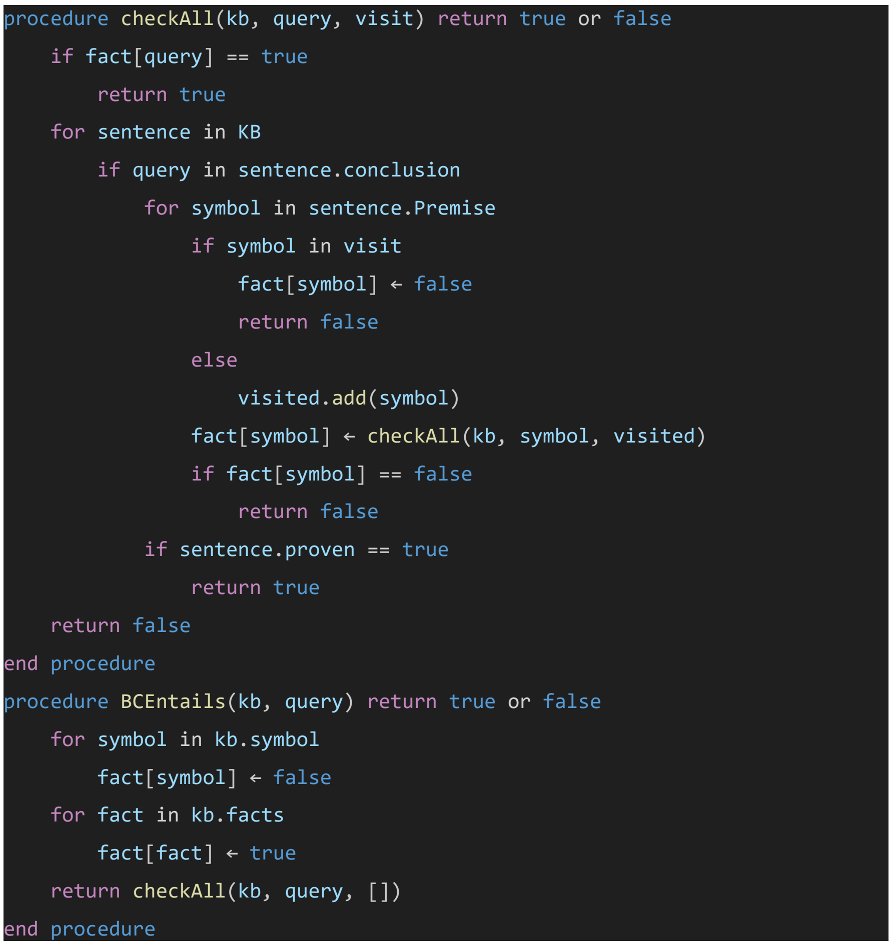
Forward chaining

Forward chaining is a “data-driven” algorithm that starts with the knowledge base and uses inference rules to decide whether the query is entailed or not (Russell and Norvig, 2010). The major merit is simple implementation when the algorithm is straightforward. In Figure 6, the algorithm uses a dictionary count to track unsatisfied premises and updates it when a list agenda mark symbols as inferred.



```
procedure check_all(kb, query) returns true or false
    while agenda is not empty do
        p ← POP (agenda)
        if not inferred[p] then
            inferred[p] ← true
            for each clause c in kb where p is in c.PREMISES do
                count[c] ← count[c] - 1
                if count[c] is 0 then
                    if HEAD[c] = q then return true
                    PUSH(HEAD[c], agenda)
    return false
end procedure
procedure FCEntails(kb, query) return true or false
    count ← a dictionary, count[i] is the number of unsatisfied premises for clause i
    inferred ← a dictionary, inferred[s] is initially false for all symbols
    for sentence in kb.sentence do
        count[c] ← number of premises in c
    for symbol in kb.symbols do
        inferred[symbol] ← false
    return checkAll(kb, query)
end procedure
```

Figure 6: Pseudocode for Forward Chaining



```
procedure checkAll(kb, query, visit) return true or false
    if fact[query] == true
        return true
    for sentence in KB
        if query in sentence.conclusion
            for symbol in sentence.Premise
                if symbol in visit
                    fact[symbol] ← false
                    return false
                else
                    visited.add(symbol)
                fact[symbol] ← checkAll(kb, symbol, visited)
                if fact[symbol] == false
                    return false
            if sentence.proven == true
                return true
    return false
end procedure
procedure BCEntails(kb, query) return true or false
    for symbol in kb.symbol
        fact[symbol] ← false
    for fact in kb.facts
        fact[fact] ← true
    return checkAll(kb, query, [])
end procedure
```

Figure 7: Pseudocode for Backward Chaining

Nevertheless, when the knowledge base is large enough, the algorithm might not yield a desirable outcome (Genesereth and Nilsson, 1987). Specifically, due to its design, the algorithm might explore irrelevant clauses or literals, leading to a high computational cost.

Backward Chaining

If the forward chaining starts from single facts, the backward chaining will start from the query (Figure 7). This algorithm utilizes a depth-first search idea to reduce the number of clauses need to check, it will look for all conclusions in the knowledge bases to match the query. Afterward, it will recursively recheck every premise of that query, and the loop will be kept running. This algorithm starts from the goal and works backward to find the known facts. If it is sufficient facts to prove the goal, then it will return "YES"; otherwise, it will return "NO". Instead of examining the whole knowledge base like Forward Chaining, Backward Chaining relatively perform better in terms of runtime by focusing on the specific goal. Therefore, in complicated test cases, backward chaining tends to require less time compared to forward chaining and the truth table algorithm.

4. Acknowledgments/Resources

Russell and Norvig's book, titled "Artificial Intelligence: A Modern Approach," has been instrumental in providing a comprehensive understanding of the functioning of inference engines and offering insights into their implementation. The book, available in both a 3rd edition from Prentice-Hall in 2010 and a 4th edition from Pearson in 2020, has provided invaluable guidance in gaining an overview of inference engine operations and conceptualizing their practical implementation.

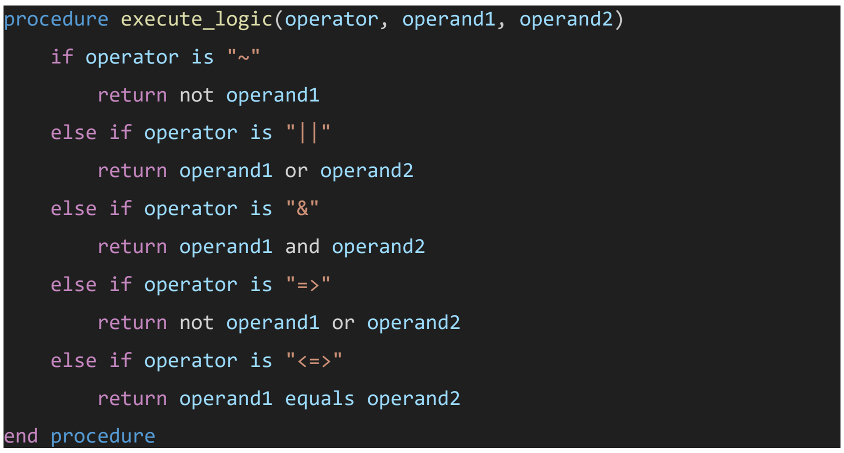
5. Notes

To use the program, the syntax is $iengine\ [method][filename]$. The available methods are TT, FC, BC, Resolution, and DPLL which are not case-sensitive. The filename is a text file formatted as demonstrated in the assignment. To run a test for Horn clauses, the command syntax would be $python\ GenTest.py\ horn$. In the FC function, the team improves the algorithm by checking if the query is initially a fact in the knowledge base, it will return the yes result immediately.

6. Research/Additional Algorithms

Generic Handler For Truth Table

To be able to handle generic clauses using a truth table, the implementation of postfix to infix conversion should be considered which is shown in Figure 8. This uses the stack to convert postfix to infix form, and during the conversion, it also executes the logic inside by implementing a logic handler below:

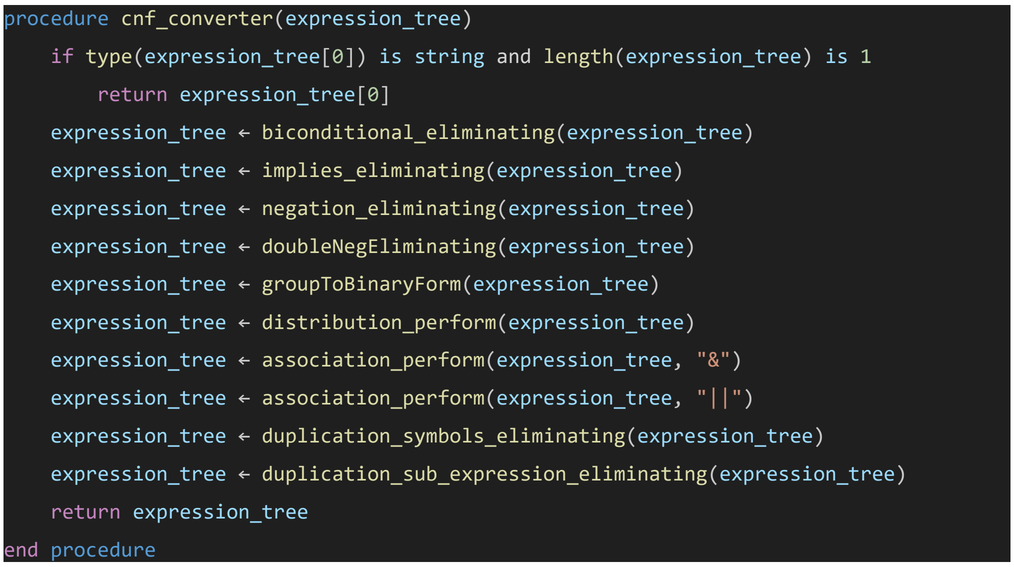


```
procedure execute_logic(operator, operand1, operand2)
    if operator is "~"
        return not operand1
    else if operator is "||"
        return operand1 or operand2
    else if operator is "&"
        return operand1 and operand2
    else if operator is "=>"
        return not operand1 or operand2
    else if operator is "<=>"
        return operand1 equals operand2
end procedure
```

Figure 8: Pseudocode for handling generic of Truth Table

Conjunctive Normal Form Conversion

The General form, containing various logic operators such as "&" (And), "~" (Not), "||" (Or), "=>" (Imply), and "<=>" (Biconditional) is incompatible with Resolution and DPLL algorithms due to their requirement of Conjunctive Normal Form (CNF) to be executed successfully. Basically, the CNF is a conjunction (And) of disjunction (Or) of a literal clause (single symbols). The most important characteristic of the CNF is that every general form can convert to the Conjunctive Normal form. Inside the Conjunction Normal Form converter, there will be several logic operations performed including biconditional elimination, imply elimination, De Morgan, Distribution Perform, Association Perform, and Duplication elimination. Figure 9 is the abstraction form of the CNF converter:



```
procedure cnf_converter(expression_tree)
    if type(expression_tree[0]) is string and length(expression_tree) is 1
        return expression_tree[0]
    expression_tree ← biconditional_eliminating(expression_tree)
    expression_tree ← implies_eliminating(expression_tree)
    expression_tree ← negation_eliminating(expression_tree)
    expression_tree ← doubleNegEliminating(expression_tree)
    expression_tree ← groupToBinaryForm(expression_tree)
    expression_tree ← distribution_perform(expression_tree)
    expression_tree ← association_perform(expression_tree, "&")
    expression_tree ← association_perform(expression_tree, "||")
    expression_tree ← duplication_symbols_eliminating(expression_tree)
    expression_tree ← duplication_sub_expression_eliminating(expression_tree)
    return expression_tree
end procedure
```

Figure 9: Pseudocode for CNF Converter

The first step of this algorithm is to convert the string expression into prefix form and construct the expression tree. Inside this report, an example will be shown to see the result of implementation inside of the CNF converter. Example:

~(a <=> b)

The prefix form of this will be represented in the form of an expression tree like:

[~, [<=>, a, b]]

Having this expression tree, it will come to the process of eliminating the bicondition (<=>) and implication (=>), the logic behind these to is to use equivalent logic transformation:

$$A <=> B = (A => B) \& (B => A)$$
$$A => B = (\sim A) \,||\, B$$

Using these ideas, the logic expression will be simplified as:

$$[\sim, [\&, [||, [\sim, a], b], [||, [\sim, b], a]]]$$

Next, applying DeMorgan law to eliminate the "~" (Not) expression outside the parentheses enable retrieval of the atomic symbols with the negation rather than the negation to a big expression. It is applied as follows:

$$\sim(A \& B) = \sim A \mid \sim B$$
$$\sim(A \mid B) = \sim A \& \sim B$$

The DeMorgan law implementation uses a self-training algorithm that repeats until convergence. The algorithm will stop once the string remains unchanged before and after transformation, serving as the convergence threshold.

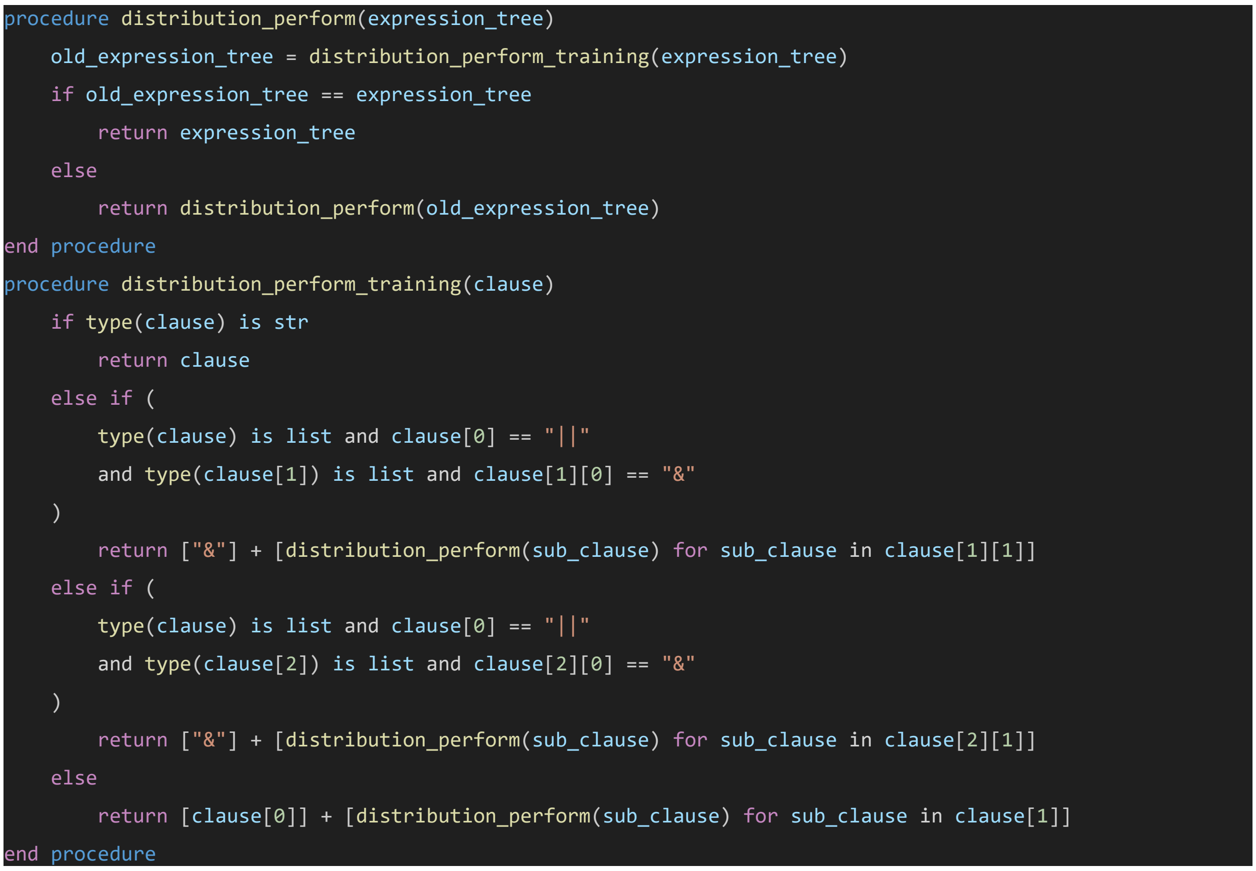
Following this, there will be cases like double negation and function "doubleNegEleminating()" will be handled that convert $\sim\sim A = A$. In addition, to prepare for effective distribution, an extra function has been implemented called "groupToBinaryForm". This function is to group a bunch of single literals into binary form, for example:

[&, p, q, r, s] = [&, p, [&, q, [&, r, s]]]

This approach enhances distribution effectiveness, simplifying handling by focusing on only two literals per clause, thereby minimizing errors during distribution performance.

The most important part of this CNF conversion algorithm is the distribution and association which are shown in Figure 10 and Figure 11 respectively. These two functions will simplify the logic expression to atomic form. Both functions continue to apply the training process, stopping only once the algorithm converges. The algorithm employs the distribution of "||" (Or) for conversions:

$$A \,||\, (B \,\&\, C) \;=\; (A \,||\, B) \,\&\, (A \,||\, C)$$



```
procedure distribution_perform(expression_tree)
    old_expression_tree = distribution_perform_training(expression_tree)
    if old_expression_tree == expression_tree
        return expression_tree
    else
        return distribution_perform(old_expression_tree)
end procedure
procedure distribution_perform_training(clause)
    if type(clause) is str
        return clause
    else if (
        type(clause) is list and clause[0] == "||"
        and type(clause[1]) is list and clause[1][0] == "&"
    )
        return ["&"] + [distribution_perform(sub_clause) for sub_clause in clause[1][1]]
    else if (
        type(clause) is list and clause[0] == "||"
        and type(clause[2]) is list and clause[2][0] == "&"
    )
        return ["&"] + [distribution_perform(sub_clause) for sub_clause in clause[2][1]]
    else
        return [clause[0]] + [distribution_perform(sub_clause) for sub_clause in clause[1]]
end procedure
```

Figure 10: Pseudocode for distributivity on the prefix logical expression

This is the core nature of the CNF converter that requires the conjunction (And) of disjunction (Or). The association function eliminates unnecessary parentheses and nested arrays in the expression tree such as transforming:

$$A \,\&\, (B \,\&\, C) \;=\; (\,A \,\&\, B) \,\&\, C$$

Finally, the final step is to eliminate any duplication of a single literal, or clause that have been appeared in the clause for example:

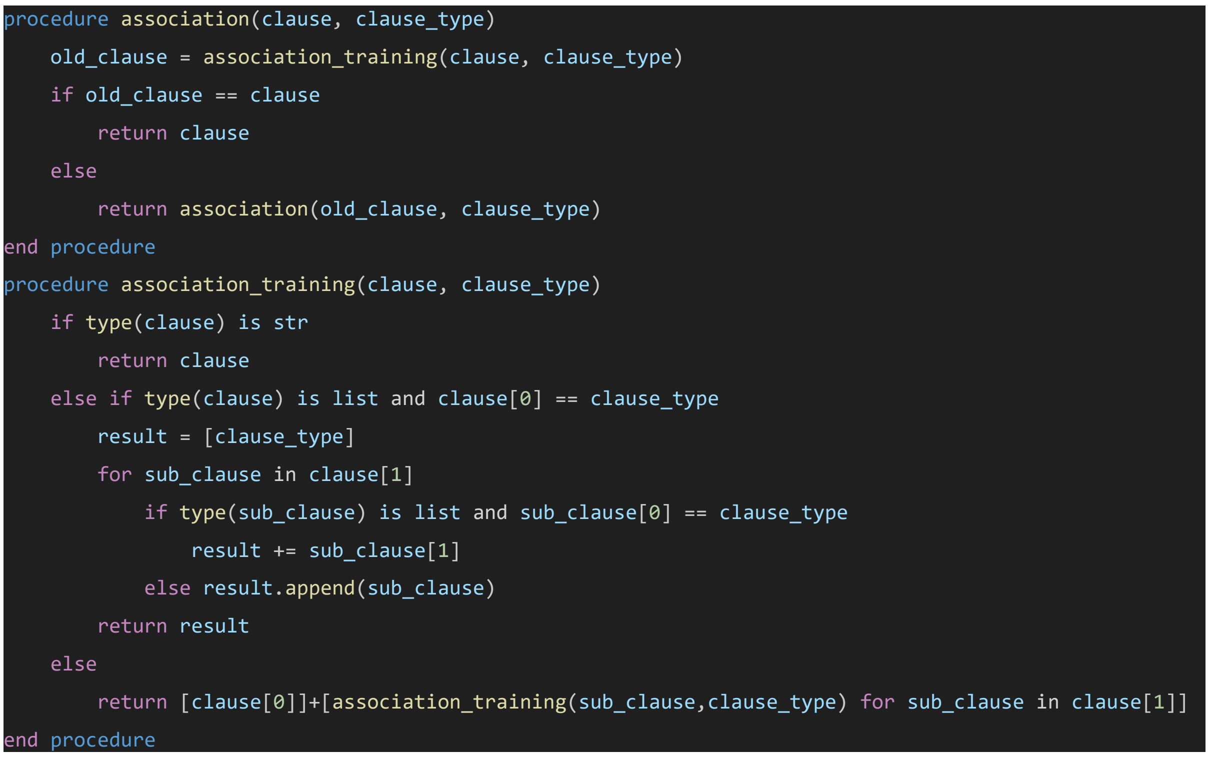
$$b \,\&\, a \,\&\, b \,\&\, a \;=\; b \,\&\, a$$

$$(a \,||\, b) \,\&\, (b||a) \;=\; a \,||\, b$$

And here is an example output of the algorithm:

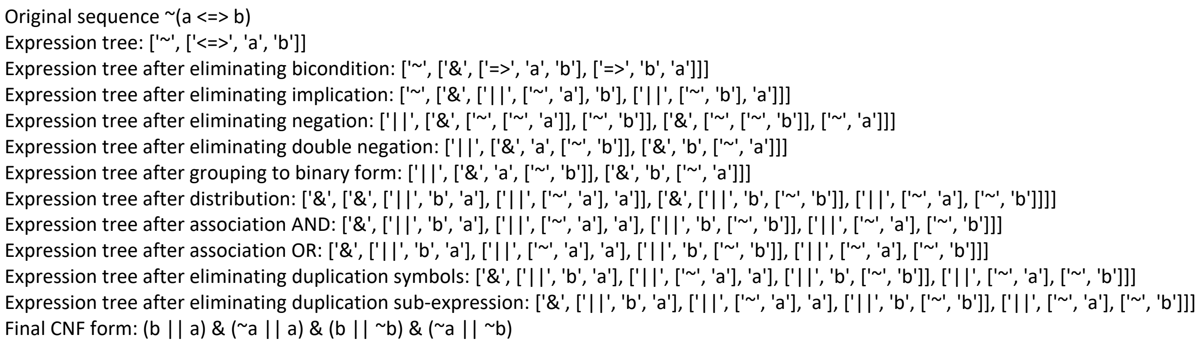
$$(b \,||\, a) \,\&\, (\sim a \,||\, a) \,\&\, (b \,||\, \sim b) \,\&\, (\sim a \,||\, \sim b)$$

The CNF conversion still needs a program to test its validity. Thanks to the 3rd party library called “sympy” with its “to_cnf” function has been implemented to take as a standard result and compared with manual code CNF converter. Fortunately, all the test cases have passed, and this algorithm would be accurate. The step-by-step example is demonstrated in Figure 12.



```
procedure association(clause, clause_type)
    old_clause = association_training(clause, clause_type)
    if old_clause == clause
        return clause
    else
        return association(old_clause, clause_type)
end procedure
procedure association_training(clause, clause_type)
    if type(clause) is str
        return clause
    else if type(clause) is list and clause[0] == clause_type
        result = [clause_type]
        for sub_clause in clause[1]
            if type(sub_clause) is list and sub_clause[0] == clause_type
                result += sub_clause[1]
            else result.append(sub_clause)
        return result
    else
        return [clause[0]]+[association_training(sub_clause,clause_type) for sub_clause in clause[1]]
end procedure
```

Figure 11: Pseudocode for associativity on the prefix logical expression



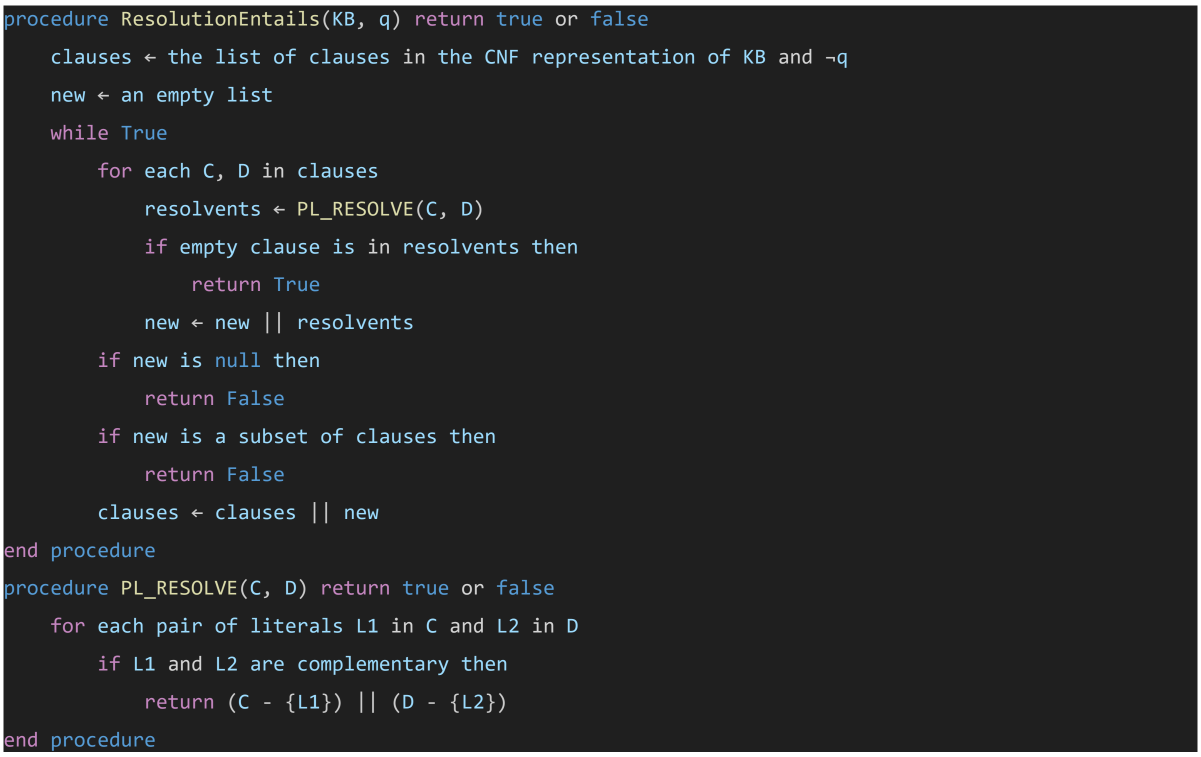
```
Original sequence ~(a <=> b)
Expression tree: ['~', ['<=>', 'a', 'b']]
Expression tree after eliminating bicondition: ['~', ['&', ['=>', 'a', 'b'], ['=>', 'b', 'a']]]
Expression tree after eliminating implication: ['~', ['&', ['||', ['~', 'a'], 'b'], ['||', ['~', 'b'], 'a']]]
Expression tree after eliminating negation: ['||', ['&', ['~', ['~', 'a']], ['~', 'b']], ['&', ['~', ['~', 'b']], ['~', 'a']]]
Expression tree after eliminating double negation: ['||', ['&', 'a', ['~', 'b']], ['&', 'b', ['~', 'a']]]
Expression tree after grouping to binary form: ['||', ['&', 'a', ['~', 'b']], ['&', 'b', ['~', 'a']]]
Expression tree after distribution: ['&', ['&', ['||', 'b', 'a'], ['||', ['~', 'a'], 'a']], ['&', ['||', 'b', ['~', 'b']], ['||', ['~', 'a'], ['~', 'b']]]]
Expression tree after association AND: ['&', ['||', 'b', 'a'], ['||', ['~', 'a'], 'a'], ['||', 'b', ['~', 'b']], ['||', ['~', 'a'], ['~', 'b']]]
Expression tree after association OR: ['&', ['||', 'b', 'a'], ['||', ['~', 'a'], 'a'], ['||', 'b', ['~', 'b']], ['||', ['~', 'a'], ['~', 'b']]]
Expression tree after eliminating duplication symbols: ['&', ['||', 'b', 'a'], ['||', ['~', 'a'], 'a'], ['||', 'b', ['~', 'b']], ['||', ['~', 'a'], ['~', 'b']]]
Expression tree after eliminating duplication sub-expression: ['&', ['||', 'b', 'a'], ['||', ['~', 'a'], 'a'], ['||', 'b', ['~', 'b']], ['||', ['~', 'a'], ['~', 'b']]]
Final CNF form: (b || a) & (~a || a) & (b || ~b) & (~a || ~b)
```

Figure 12: CNF converter step-by-step example

Resolution Algorithm

Implementation

Resolution is a complete inference algorithm dealing with logical assertions, queries, and propositional logic (AIMaterials, n.d.). In general, it works by proofing the contradiction. To demonstrate $KB \vDash \alpha$, the authors illustrate that $(KB \wedge \neg\alpha)$ is unsatisfiable (Russell and Norvig, 2010). In the implemented code, the representation of $(KB \wedge \neg\alpha)$ is initially converted to CNF sentences, which is the major requirement of the algorithm. The algorithm function "PL_RESOLVE" checks the resolution rule by identifying pairs of clauses containing complementary literals, which are the literal and its negation. To resolve it, the founded pair would remove the literals from its clause and simultaneously combines the remaining literals to produce a new clause called resolvent, which is latterly added to the new set of clauses if it has not already appeared. The "entails" function is a core method to represent the result of the test cases. To start with, it has a list of clauses which achieve by negating the query and adding it to the list of clauses with a knowledge base in CNF form. The algorithm then iterates to get new resolvents. If an empty clause is founded, it concludes that $KB \vDash \alpha$ is true. Otherwise, if the algorithm does not yield any new clauses when all possible pairs of clauses have been resolved, it means that KB does not entail α (Russell and Norvig, 2010).



```
procedure ResolutionEntails(KB, q) return true or false
    clauses ← the list of clauses in the CNF representation of KB and ¬q
    new ← an empty list
    while True
        for each C, D in clauses
            resolvents ← PL_RESOLVE(C, D)
            if empty clause is in resolvents then
                return True
            new ← new || resolvents
        if new is null then
            return False
        if new is a subset of clauses then
            return False
        clauses ← clauses || new
end procedure
procedure PL_RESOLVE(C, D) return true or false
    for each pair of literals L1 in C and L2 in D
        if L1 and L2 are complementary then
            return (C - {L1}) || (D - {L2})
end procedure
```

Figure 13: Pseudocode for Resolution

Properties

The algorithm is guaranteed to find a contradiction if existed in the CNF representation of the knowledge base and negation of the query. This property ensures that the algorithm is complete which is emphasized by Russell and Norvig (2010). However, it is not the most effective algorithm regarding optimality to solve the problems. It would generate a big number of inferences which are sometimes not necessary for finding a solution. As mentioned in AI Materials, the approach could potentially generate a really large number of new sentences when it tries to look up for contradiction. The naive implementation triggers a high computational cost which can make it infeasible for a big knowledge base and sometimes make it no solution for those problems.

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

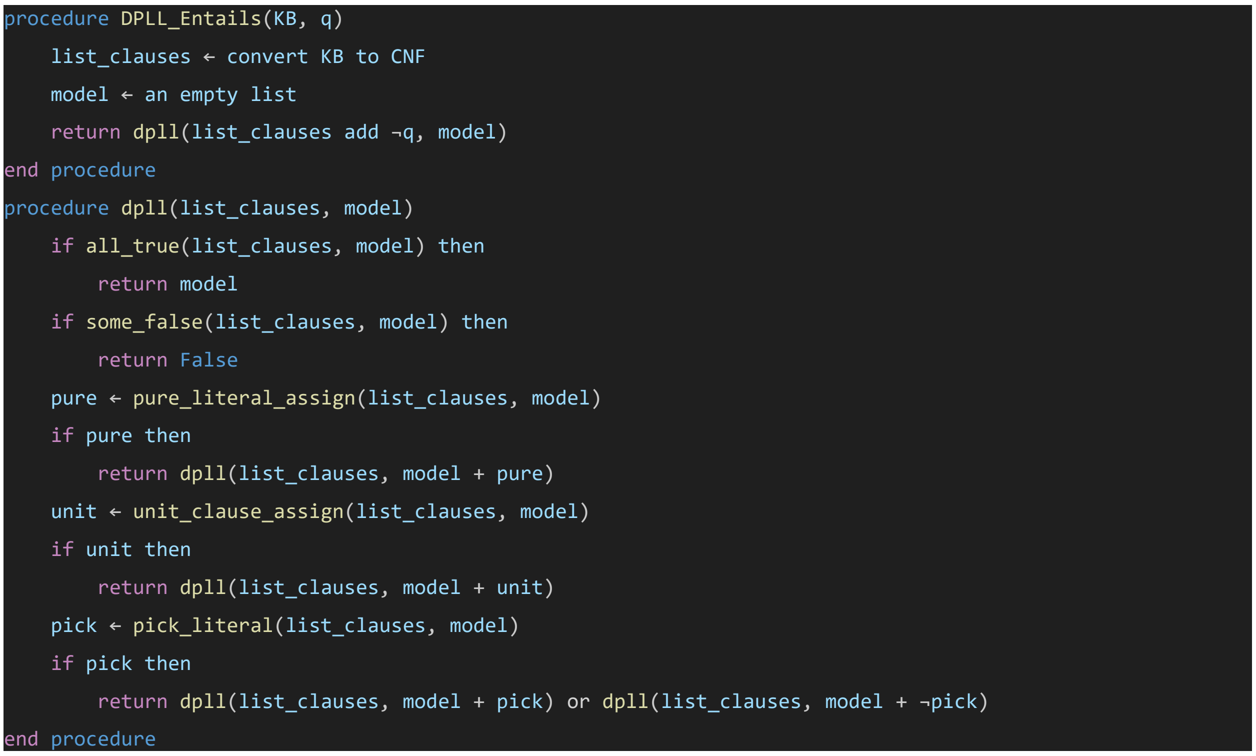
Implementation

The DPLL is a backtracking algorithm for the satisfiability of the knowledge base formulated in CNF form (Ozlutiras, n.d.). This approach uses for solving the Boolean satisfiability problem which is known as SAT. In the code implementation, the recursions are made used to explore the different assignments of literals. The two cored function is "unit_clause_assign" and "pure_literal_assign" following the rule of unit propagation and pure literal elimination respectively (Wikiwand, n.d.). A unit clause is a clause containing only one unassigned literal. This clause needs to be assigned true to satisfy the clause according to the rule. In the code, the function finds a unit clause from a list of clauses, and it will assign that literal to be true in the model if it exists. This act as a precursor to reducing a significant problem size since the search space is pruned. As for the pure literal elimination, it initially detects the pure literal which either appears as a positive or negative literal in all clauses but not both. The method in the implementation searches for pure literal in the list of clauses as well as assigns it to be true in the model and returns the literal if exists. Therefore, these clauses can be deleted from the search leading to a reduction in problem size.

Properties

The algorithm is made sure to be completed according to the research of Liberatore (n.d.). Nevertheless, this approach is not guaranteed the optimality of the search as it is not associated with the SAT problem.

As for time complexity, despite the enhancement compared with the truth table, the worst case is still $O(2^n)$ (Russell and Norvig, 2010). The algorithm is proven to perform well in practice as it reduces the number of clauses thanks to the principles mentioned above.



```
procedure DPLL_Entails(KB, q)
    list_clauses ← convert KB to CNF
    model ← an empty list
    return dpll(list_clauses add ¬q, model)
end procedure
procedure dpll(list_clauses, model)
    if all_true(list_clauses, model) then
        return model
    if some_false(list_clauses, model) then
        return False
    pure ← pure_literal_assign(list_clauses, model)
    if pure then
        return dpll(list_clauses, model + pure)
    unit ← unit_clause_assign(list_clauses, model)
    if unit then
        return dpll(list_clauses, model + unit)
    pick ← pick_literal(list_clauses, model)
    if pick then
        return dpll(list_clauses, model + pick) or dpll(list_clauses, model + ¬pick)
end procedure
```

Figure 14: Pseudocode for DPLL

Comparing general clause handling algorithm

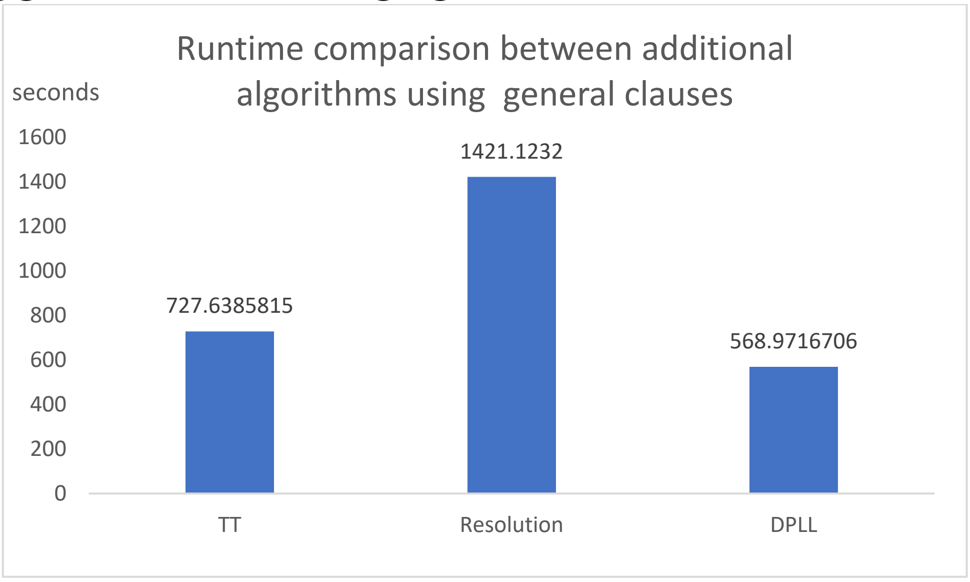


Figure 15: Running time comparison between three algorithms using 1000 generic tests

From the bar chart above, it can be easily seen that the resolution algorithm has the longest running time, which is 1421.1232 seconds. In other words, its naive implementation when trying to take the complement of all clauses makes high computational costs. As for the DPLL, thanks to the two core rules mentioned above, it saves an amount of problem size triggering a faster running time than TT.

7. Team summary report

The team includes two members Anh Minh Nguyen (103178955) and Duy Khoa Pham (103515617). The communication between team members is through Discord exchanging ideas about the algorithms.

Anh Minh manages our GitHub repository, with both of us following commit lint rules and engaging in code review and discussions. Weekly meeting plays a pivotal part in our team's workflow. These scheduled sessions allow us to update progress and share insight gained from working on our respective tasks. Each member tries to help each other in strength to improve the team's collective skills. Skill and mutual support are of paramount importance to foster individual growth but also enhance our team's productivity and collaborative spirit. To be specific, while Anh Minh has more than one year of working as a software developer with outstanding coding skills, Duy Khoa is a research assistant at the university with a strong academic perspective. As a result, we share experiences with each other and consciously distribute tasks based on our individual areas of expertise, enabling us to maximize each member's potential and consequently yield high-quality results. We believe that with the experience of practical coding skills and academic research, our team excellently achieves our goals.

Component	Minh	Khoa
File parser	48%	52%
Knowledge Base	45%	55%
Postfix Prefix Operation	55%	45%
Truth Table Algorithm	55%	45%
Forward Chaining Algorithm	45%	55%
Backward Chaining Algorithm	55%	45%
CNF conversion	50%	50%
Testing Generator	51%	49%
Resolution	53%	47%
DPLL	49%	51%
Overall Contribution	50.18%	49.82%

Table 3: Team contribution toward the assignment

8. Reference

AIMaterials. (n.d.). Resolution. AI Materials. Retrieved May 22, 2023, from https://aimaterials.blogspot.com/p/resolution.html

Genesereth, M. R., & Nilsson, N. J. (1987). Logical Foundations of Artificial Intelligence. Morgan Kaufmann

Liberatore, P. (n.d.). The DPLL Algorithm. Department of Computer, Control, and Management Engineering Antonio Ruberti at Sapienza University of Rome. Retrieved May 24, 2023, from http://www.diag.uniroma1.it/~liberato/ar/dpll/dpll.html.

Ozlutiras, M. (n.d.). Brief explanation of DPLL (Davis-Putnam-Logemann-Loveland) Algorithm. Medium. Retrieved May 24, 2023, from https://mertozlutiras.medium.com/brief-explanation-of-dpll-davis-putnam-logemann-loveland-algorithm-663f4a603c1.

Rastogi, R., Mondal, P., & Agarwal, K. (2015, March). An exhaustive review for infix to postfix conversion with applications and benefits. In 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 95-100).

Russell, S., & Norvig, P. (2010). Artificial intelligence: a modern approach (3rd ed.). Pearson.

Wikiwand. (n.d.). DPLL Algorithm. Retrieved May 24, 2023, from https://www.wikiwand.com/en/DPLL_algorithm.