Assignment 2 - Inference Engine for Propositional Logic (Group 4)

Ta Quang Tung - 104222196 **Nguyen Quang Huy - 104169507**

Contents

Introduction

This report describes our work on the inference engine for propositional logic as part of Assignment 2 of COS30019. For this assignment we have implemented the required **truth table checking**, **forward chaining**, and **backward chaining** algorithms. We have also implemented a **general logic parser** and the **resolution theorem prover** as part of the research component. This project is the collaborative effort between Ta Quang Tung (104222196) and Nguyen Quang Huy (104169507) of group 4.

Usage instructions

This project requires **dotnet 7.0** to run. Please install it *here* and make sure the link to its executable is available in the `PATH` environment variable. You can run `dotnet -v` to check if it is installed.

In the terminal, navigate to the project's directory (the directory containing `iengine.bat`) and run the following command:

```
iengine <method> <filename>
```

where `<method>` is one of `TT`, `FC`, `BC`, or `RP` and `<filename>` is the name of the file containing the knowledge base and query.

For the sake of organization, we have put all input files in a folder named `cases` on the same level as the `iengine.bat` file. **If you plan to run your own test cases, please make sure to put the file into that folder.**

If you choose the method `TT` (truth table checking), the output will be `YES` followed by the **number of models in which both KB and the query is true** or `NO`. If you choose the method `FC` (forward chaining) or `BC` (backward chaining), the output will be `YES` followed by a list of symbols or `NO`. If you choose the method `RP` (resolution theorem prover), the output will be `YES` or `NO`.

Features

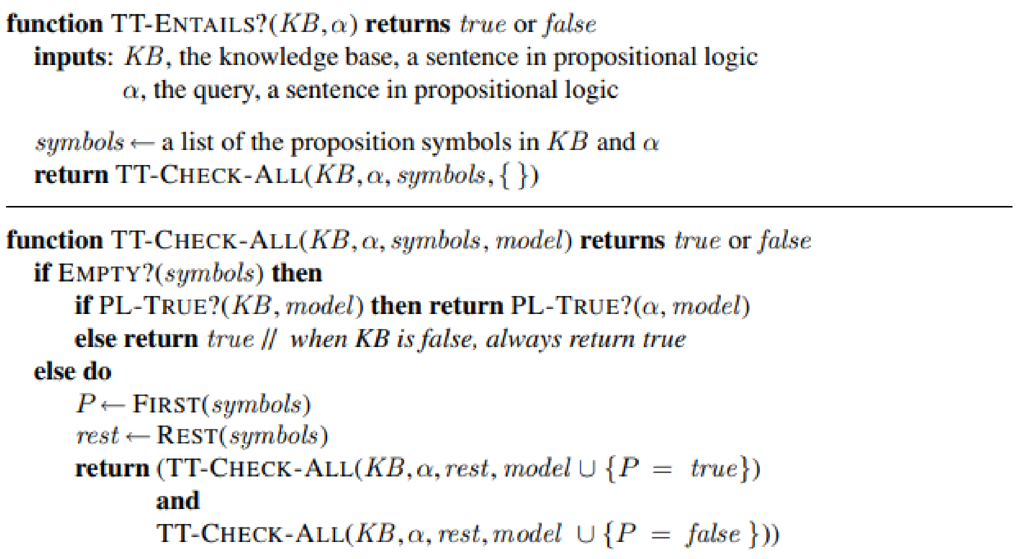
In this assignment, we have implemented all the required features, which are the truth table checking, forward chaining, and backward chaining algorithms. Besides that, we have also implemented some advanced features which are the resolution theorem solver and general knowledge parser.

General parser

Our parser is capable of reading any sentence in propositional logic, not just those in Horn form. This allows our truth table checking and resolution theorem prover algorithms to work with an array of complex inputs. The inner workings of the parser, along with the structure of the sentences it constructs, is described in detail in *this section*.

Truth table checking

The Truth Table checking is an exhaustive method that thoroughly explores all possible models for a set of symbols in a Knowledge Base, checking the validity of each model. In this program, the TT method has been expanded to handle various Knowledge Bases, not limited to specific clause forms like the Horn form. This extension was achieved by implementing a general logic parser for the general knowledge base, creating an object-oriented interface to assess clauses as true or false within a given Knowledge Base. Here is the pseudocode implementation of this method.



```
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 ← 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
     $P$ ← FIRST(symbols)
     rest ← REST(symbols)
     return (TT-CHECK-ALL($KB$, $\alpha$, rest, model $\cup$ {$P$ = true})
            and
            TT-CHECK-ALL($KB$, $\alpha$, rest, model $\cup$ {$P$ = false }))
```

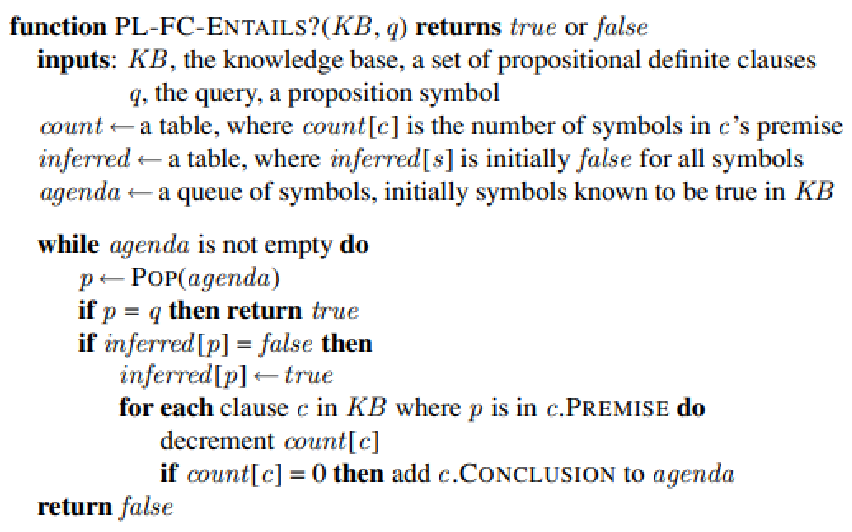
Figure 1: The pseudocode implementation of Truth table checking method

Truth Table checking is like trying out all possible combinations to see if a set of statements (Knowledge Base) supports another statement (query). It checks if, in every combination, the initial statements are true and if the final statement is true as well. This way, it thoroughly examines all possibilities to figure out if one statement logically follows from another.

Forward chaining

Forward chaining is like solving a puzzle step by step. It starts with what we know (conditions and rules) and uses them to figure out more things until we reach a solution. It's a bit like connecting the dots, where each dot is a piece of information, and we connect them one by one until we see the whole picture.

In this assignment, we implement the forward chaining function using the pseudocode provided on the book *AI: A Modern Approach (3rd edition).*



```
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 2: The pseudocode implementation of Forward Chaining method

In this algorithm, a queue called the agenda keeps track of symbols known to be true, while a count table monitors the number of symbols in each clause's premise yet to be determined. The algorithm iterates through the agenda, processing symbols, and updating the count based on known premises. If a symbol's count reaches zero, its associated conclusion is added to the agenda. The process continues until the agenda is empty or the query symbol is found, indicating that the query is entailed by the knowledge base. This method of reasoning, often used in artificial intelligence, systematically applies known conditions and rules to progressively reach logical conclusions.

Backward chaining

In contrast with Forward Chaining, Backward chaining is like solving a puzzle by starting from the answer and figuring out the steps that lead to it. Instead of moving forward from what's known, it works backward to find the rules or conditions that result in a specific outcome. Backward chaining is a form of goal-directed reasoning. The method only proves the relevant facts, which makes the cost of backward chaining much less than the value of the knowledge base.

Backward chaining, as the name implies, operates in reverse from the query. When the truth of the query q is established, no additional work is necessary. However, if q's validity is uncertain, the algorithm identifies knowledge base implications with q as the conclusion. By proving all the premises of one of these implications through backward chaining, the algorithm confirms the truth of q.

Resolution theorem prover

In this section, we explore a different approach to determining if something is true through theorem proving. Unlike model checking where we enumerate various configurations to show that a statement holds in all models where the knowledge base holds, theorem proving involves applying direct logical rules to the statements in our knowledge base. This allows us to construct a logical proof for the desired statement without consulting multiple models. If the number of models is extensive but the length of the proof is concise, then theorem proving can be a more efficient method than model checking.

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

  $clauses \leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  $new \leftarrow \{\}$
  loop do
     for each pair of clauses $C_i, C_j$ in $clauses$ do
        $resolvents \leftarrow$ PL-RESOLVE($C_i, C_j$)
        if $resolvents$ contains the empty clause then return $true$
        $new \leftarrow new \cup resolvents$
     if $new \subseteq clauses$ then return $false$
     $clauses \leftarrow clauses \cup new$
```

Figure 3: The pseudocode implementation of the Resolution theorem prover

Inference procedures based on resolution operate on the principle of proof by contradiction. The goal is to demonstrate that if KB logically entails α, then the conjunction of KB and the negation of α (KB ∧ ¬α) is unsatisfiable, essentially proving a contradiction. The resolution algorithm first transforms (KB ∧ ¬α) into Conjunctive Normal Form (CNF). Then, the resolution rule is applied to each pair of resulting clauses. Complementary literals in each pair are resolved, generating new clauses that are added to the set if not already present. The process continues until either no new clauses can be added, indicating that KB does not entail α, or two clauses resolve to yield the empty clause, signifying that KB logically entails α.

Test cases

During this assignment, we have implemented several test cases to ensure that our program still provides the correct answers in various scenarios.

Test case	TT	FC	BC	RP
test_generic_1.txt	YES: 3	X	X	YES
test_generic_2.txt	NO	X	X	NO
test_generic_3.txt	YES: 2	X	X	YES
test_generic_4.txt	YES: 2	X	X	YES
test_horn_1.txt	YES: 3	YES: a b p2 p3 p1 d	YES: p2 p3 p1 d	YES
test_horn_2.txt	YES: 1	YES: a h j i f b c d e g	YES: a c e g	YES
test_horn_3.txt	YES: 1	YES: a h j i f b c d e g	YES: a c e g	YES
test_horn_4.txt	YES: 1	YES: v a b c e f j g	YES: a e f j g	YES
test_horn_5.txt	YES: 1	YES: p3	YES: p3	YES
test_horn_6.txt	NO	NO	NO	NO
test_horn_7.txt	YES: 4	YES: d	YES: d	YES
test_horn_8.txt	YES: 1	YES: a h j i f b c d e g	YES: a c e g	YES

Table 1: Result of all the test cases

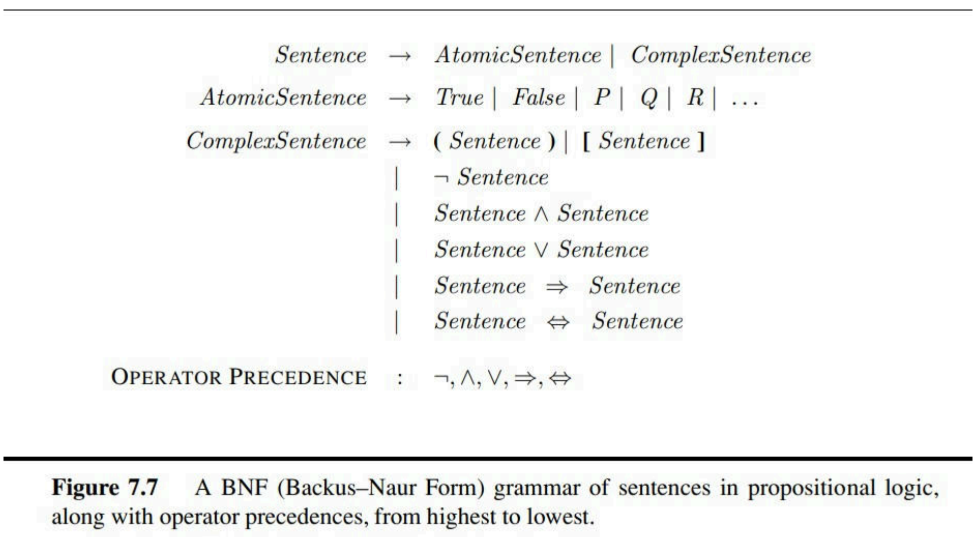
Forward chaining and backward chaining don't work with generic test cases because the knowledge base is not in Horn form. We have a total of 12 test cases and all of them have been successfully tested.

Research

General parser

For this assignment, we built a general parser that can parse complex sentences in propositional logic, allowing us to also work with non-Horn knowledge bases. Before describing how this parser works, we will first discuss how sentences are expressed in our program.

Sentence representation



$$\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & & \\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 4: The grammar of propositional logic. Credit: AI: A Modern Approach (3rd edition.)

We follow the object-oriented paradigm to represent the sentences. Our design is based closely on the grammar of propositional logic described in (Russell & Norwig, 2009). An abstract class `Sentence` captures the key functionalities of a sentence, such as the calculation of its truth in a given model. Inheriting this class are `AtomicSentence`, which represents a single propositional symbol, and the abstract class `ComplexSentence` which consists of one or two sentences and an operator. The classes `Not`, `And`, `Or`, `Imply`, and `Iff` (also known as biconditional) inherit `ComplexSentence` and implement the individual operators. While `Not` technically only has one operand, we grouped it with the other operators for the sake of convenience.

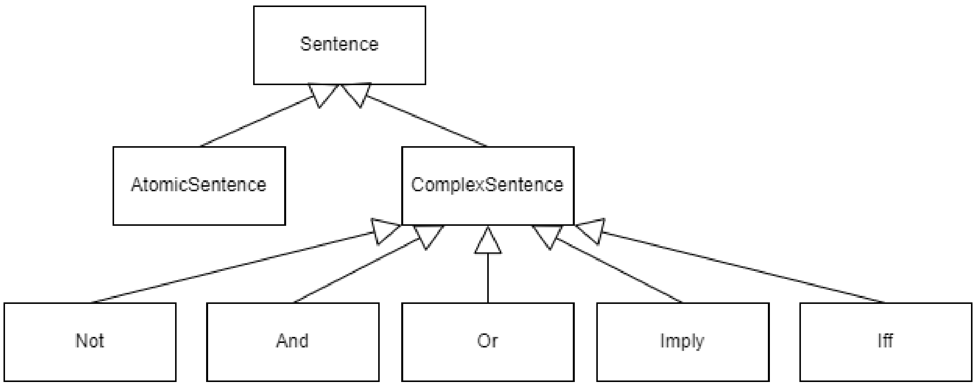


Figure 5: The UML class diagram showing the relationship between the sentence classes.

The parser

`Parser(string)` $\longrightarrow$ `Sentence`

The task of the general logic parser is to receive an input string representing a knowledge base or sentence and translating it into a Sentence object. Given the various items that can be encountered in the input string, from symbols, operators to parentheses, working with the string directly can be quite challenging. Because of this, before constructing the `Sentence` object, we run the string through a step we call **Tokenization**.

$$\texttt{Tokenize(string)} \rightarrow \texttt{List<Token>}$$

Tokenization is the process of identifying meaningful parts of the raw input string. Each meaningful component of the sentence is captured in a Token object. A token object can store a symbol, an operator, or a bracket. For instance, the tokens for this sentence are as follows:

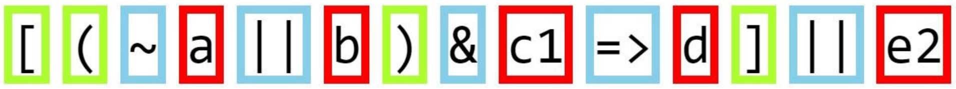


Figure 6: The tokens for this propositional logic sentence. Red represents the symbols, Green represents the brackets, and Blue represents the operators.

After extracting the tokens from the sentence, we pass the list of tokens to a function to construct the sentence object. Working with brackets directly can be cumbersome and difficult, so we first consider how a list of bracket-free tokens can be turned into a meaningful sentence. Let us consider the following sentence with the highlighted tokens:

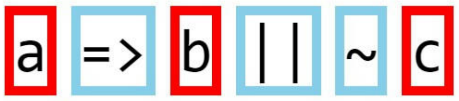


Figure 7: The tokens for this propositional logic sentence. Red represents the symbols and Blue represents the operators.

The first step is to determine the least significant operator (the one that is performed last). We can do this by iterating over the list of tokens from left to right, checking ones that contain an operator. In this case, the `Imply` operator (=>) is performed last. After determining the least significant operator, we split the sentence into two parts, one on the left of the operator and the other on its right. These two parts now form two new sentences: `a` and `b || ~c`. Using recursion, we run the parse function on these parts to produce two new sentences, which correspond to the left and right operands of `Imply`. The base case of recursion is when we encounter just a single symbol token, in which case we return an `AtomicSentence` object containing that symbol. In the example, the left part of `Imply` is the base case, and only an `AtomicSentence` containing `a` is returned. In sentences where `Not` is the least significant operator (which will be the case in the example after we split `b || ~c` by `||`), we only need to consider the part on the right, because `Not` is an unary operator. This bracket-free construction process yields an `Imply` object which looks like `Imply(a, Or(b, Not(c)))`.

With a bracket-free parser as the foundation, we can work on tackling bracket tokens. An interesting fact about brackets is that brackets that open last are closed first. Everything that lies between a pair of open and close brackets is part of a sentence. This pattern lends itself very well to the stack data structure. We can iterate over the initial list of tokens and push them onto the stack as we go along. Every time we encounter a close bracket, we pop all tokens until the last open bracket. The tokens between these open and close brackets are guaranteed to be bracket-free because if there were valid brackets within them, we would have popped all tokens within these open and close brackets first. We can use the bracket-free construction process described above to obtain a sub-sentence, which is then stored in a token and added back to the stack. This sub-sentence is treated similar to a symbol by the bracket-free constructor, and a bigger sentence can be created to wrap around it.

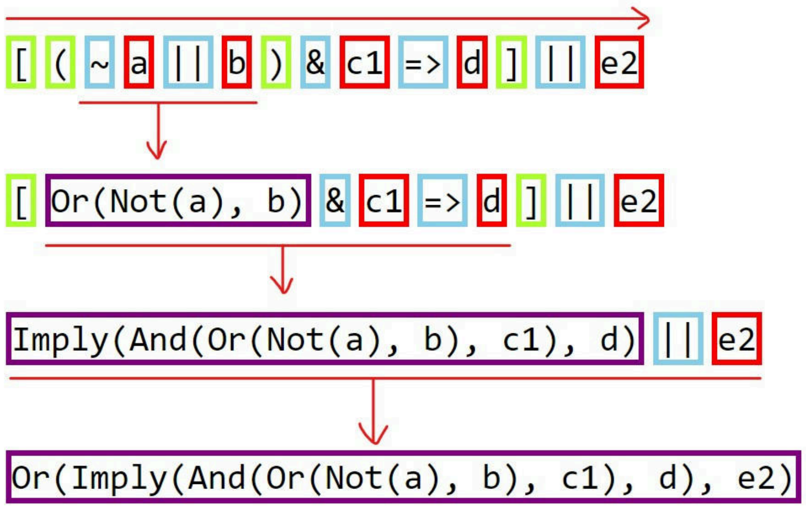

[(~ a || b) & c1 => d] || e2
[Or(Not(a), b) & c1 => d] || e2
Imply(And(Or(Not(a), b), c1), d) || e2
Or(Imply(And(Or(Not(a), b), c1), d), e2)

Figure 8: How the initial list of tokens are turned into the final sentence. The purple outline indicates a sentence token. Note how each time a close bracket is encountered, everything between it and the last open bracket is translated into a sentence. When no close brackets are encountered, the final sentence is constructed.

This process yields a complete sentence that is ready for inference processes.

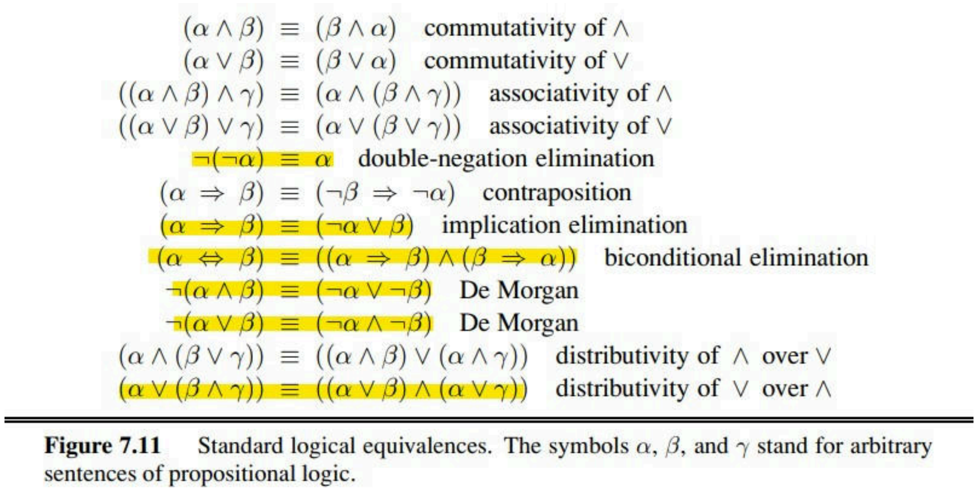
Resolution theorem prover

The basic algorithms of truth table checking, forward chaining, and backward chaining can be quite restrictive. For complex knowledge bases with a large number of propositional symbols, truth table checking's running time grows exponentially as it systematically explores every possible True/False combination of symbols. Meanwhile, forward chaining and backward chaining are limited to Horn-form knowledge bases. A better approach to inference is that of theorem proving, which seeks to provide a logical rationale by expressing why things cannot go wrong in the form of theorems. This method involves understanding and articulating precise reasoning steps, convincing the theorem prover of the soundness of the argument. For that reason, even if the number of models is large but the length of the proof is short, theorem proving can be more efficient than model checking.

Resolution-based theorem proving is a form of theorem proving which centers around the rule of **resolution**. Resolution is an inference rule that takes two clauses A and B (a disjunction of positive or negative literals) and produces a new clause C containing the literals from the original clauses except a pair of complementary literals x and $\neg x$ where x belongs to A and $\neg x$ belongs to B.

The resolution-based theorem prover is based on the idea of proof by contradiction. To prove that KB entails a sentence α, we prove that $KB \wedge \neg\alpha$ is unsatisfiable. This is essentially saying that we have to prove there is no model in which KB is true and α is false.

The resolution prover algorithm first retrieves all clauses from the sentence $KB \wedge \neg\alpha$. However, $KB \wedge \neg\alpha$ needs to be put in conjunctive normal form (CNF) which means a conjunction of clauses. The process for converting from any form to CNF can be summarized in 5 steps: (1) eliminate biconditionals, (2) eliminate implications, (3) eliminate operations inside `not`s and (4) distributing `or`s over `and`s using the rules of propositional logic.



$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of $\wedge$
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of $\wedge$
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of $\vee$
$\neg(\neg\alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of $\wedge$ over $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of $\vee$ over $\wedge$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

Figure 9: The rules of propositional logic relevant to the conversion to CNF are highlighted in yellow.

After obtaining the set of clauses from $KB \wedge \neg\alpha$, we apply the resolution rule for each pair of clauses to produce new clauses called resolvents. If the resolvents contain an empty clause (created by resolving to complementary literals), we reach a contradiction and $KB \models \alpha$ is proved. If we cannot produce any new clauses (resolvents is a subset of clauses), KB does not entail α. If we can produce new clauses without creating the empty clause, we add them to the set of clauses and repeat the pairwise resolution process.

Team summary report

For this assignment, we both contributed to the project and attempted to finish it together. The source code is shared between team members via GitHub and the communication channel is Messenger. Whenever one member is stuck with his part, we will hold a meeting in class or online. Simple questions can be solved by texting between members.

Ta Quang Tung

Total contribution: 55%, including:
- General parser
- Truth table checking algorithm
- Resolution theorem prover (converting to CNF, resolution rule)
- Test case development

Nguyen Quang Huy

Total contribution: 45%, including:
- Forward chaining
- Backward chaining
- Resolution theorem prover (main resolution function)
- Testing and debugging

Acknowledgements and resources

AI: A Modern Approach (3rd edition) (This textbook provides us with all the foundational knowledge to complete this assignment. Much of our code is based on the pseudocode implementations in the book.)

https://homepage.cs.uri.edu/~cingiser/csc481/chapter_notes/amarant.pdf (This page helps us come up with the implementation of the backward chaining method) https://wiki.eecs.yorku.ca/course_archive/2016-17/W/4315/_media/public:lecture24.pdf (This page compares the differences between theorem prover and model checking)

https://www.datacamp.com/blog/what-is-tokenization (Inspired the Tokenization method)

Bibliography

Russell, S., & Norwig, P. (2009). *Artificial Intelligence: A Modern Approach (Third Edition).*