Inference Methods for Propositional Logic

Erin Witmer
ewitmer@ur.rochester.edu
University of Rochester
Rochester, New York

ABSTRACT

This paper provides a detailed overview of the data structures and algorithms used to implement two propositional theorem provers: basic model checking and propositional resolution.

KEYWORDS

propositional logic, automated reasoning, truth-table enumeration, propositional resolution

ACM Reference Format:
Erin Witmer. 2019. Inference Methods for Propositional Logic. In *Proceedings of CSC 442 (Erin Witmer).* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

The objective of Project 2: Automated Reasoning was to design and develop implementations of two inference methods for propositional logic:

- *Basic Model Checking*: The truth-table enumeration method. The goal is to decide whether KB $\models \alpha$ for some sentence α. In other words, based on what we know (KB: the knowledge base) will α be TRUE in every possible model of our in which our KB is TRUE.
- *Propositional Resolution*: The resolution-based theorem proving uses proof by contradiction of any entailment which follows from the KB. In other words, if we know there there is no case in which $\neg\beta$ is consistent with α, then we can infer that KB $\alpha \models \beta$

This write-up will detail the following:

- *Program design*: Data structures and algorithm design.
- *Member contribution*: Attribution of work.
- *Problems*: Results of the four required problems.
- *Discussion*: Takeaways and challenges.

2 PROGRAM DESIGN

The program was written in Python3. At a high level, a very basic data structure was used to represent logical statements. As suggested in the assignment, I used a tree structure to represent statements of logic. The major components of the program are the following:

- `Node`: A class, the basic building block of the tree structure representing a statement of logic.
- `KB`: A class, stores all known statements of logic.
- `basic`: A function, takes a KB and α, and checks whether KB $\models \alpha$ by model checking
- `resolution`: A function, takes a KB and α, and checks whether KB $\models \alpha$ by resolution theorem

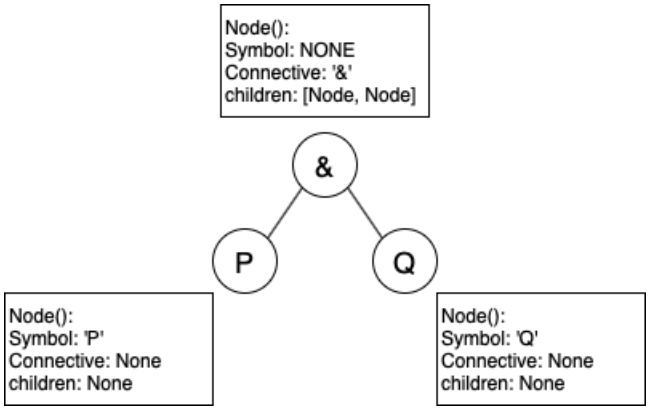

Figure 1: Tree representation of logical statement

- `parse_logic`: A function, takes a string and converts it to logical form, then converts it to conjunctive normal form.

As well as a script from which the `main` program is run: `problems.py`.

2.1 Node Class

The `Node` class is the basic building block of a propositional logic statement.

2.1.1 Class Properties. The class is instantiated with two arguments, `symbol` and `connective`. One of the two properties will be NONE. The third property is `children`, a list of all the descendent nodes. In hindsight, implementing this with an additional link to the parent node may have made some of the parsing functions more straightforward, however I was able to implement everything with a single link from parent node to child node. Figure 1 shows an example of how the propositional statement $P \wedge Q$ is represented.

2.1.2 Class Methods. The class contains a number of basic methods, such as adding a child (append a new child to the list `children`) and checking whether the node is a symbol or connective. The class also contains the more complex methods required for transforming the logical statement into conjunctive normal form. For example, the `replace_cond()` method performs a transformation from $P \rightarrow Q$ to $\neg P \vee Q$. This is shown in Figure 2.

2.2 KB Class

This a simple class which when instantiated, creates a root node: $\wedge$.

2.2.1 Class Methods. Statements are added to the KB with the `addKnowledge(node)` method. If the root of the node being added is 'and', the children are detached and added to the KB root 'and' node.

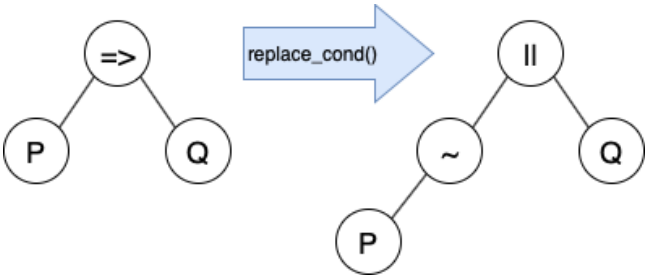

Figure 2: Transformation methods in the Node class

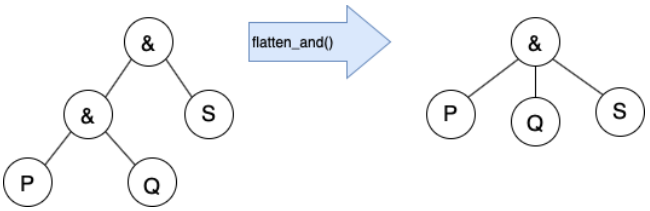

Figure 3: Flattening the tree

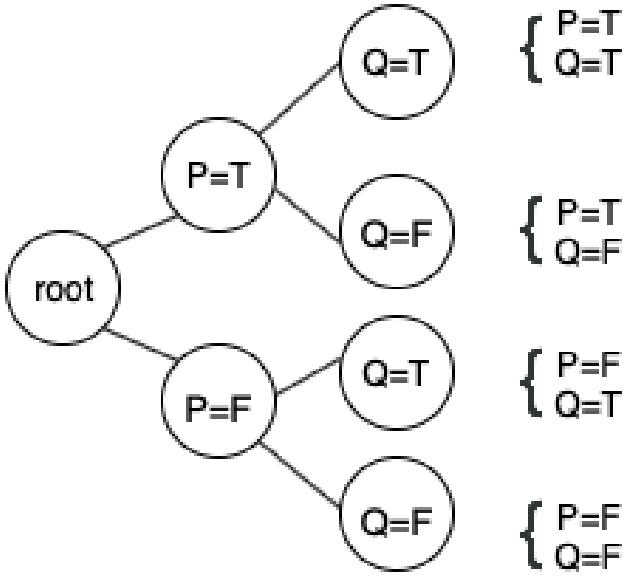

Figure 4: Building the models

2.3 basic.py functions

The `basic.py` file contains the functions for running model checking if KB $\models \alpha$ is TRUE. The main function, `tt_entails(KB,a)` is written as described in the text.[1] The function takes in a KB, α and returns TRUE if KB $\models \alpha$.

The first step of the function is to get the list of symbols from both the KB and α. For example, if the KB is $P \models Q \wedge P$ and α is Q, `symbols` = (P,Q). A helper method, `get_symbols(node)` traverses the trees and uses Python sets to get a list of unique symbols.

The function then calls the recursive function, `tt_check_all(KB, a, symbols, model)`. The function is initially called with the KB (the root node of the tree), α, a node, the aggregate list of symbols we extracted from `get_symbols(node)`, and model, which is initially passed in as an empty hash table: {}. The base case for this recursive function is that the length of symbols in the list equals zero. In other words, all symbols are represented in the current model we are checking. The recursive case builds a binary tree of all possible representations of the model. Once the base case is reached, we check if the KB is true in the current model. This is done with another recursive function:
`pl_true(node, model)`

The KB is passed in as the node, and is checked for truth in the current model. This is done recursively with depth first traversal of the tree, looking up the truth value of the deepest leaf nodes and passing those values up the tree. Each connective has its own evaluation logic. For example, the $\wedge$ connective will only evaluate to TRUE if each of its children evaluate to TRUE. That value is then passed up the tree until the root node evaluates to a final value. If it returns TRUE, the α node is evaluated. If $model(\alpha)$ returns FALSE, the function returns FALSE, because we have found a model in which the KB does not entail α. If the α evaluation returns TRUE, model checking continues. If all models are checked, and in each model where KB evaluates to true, α also evaluates to TRUE, then we have proven the $KB \models \alpha$.

2.4 resolution.py functions

The `resolution.py` file contains the functions for running the resolution based theorem prover. The goal is to achieve proof by contradiction, because we know if $KB \models \alpha$, then $KB \wedge \neg\alpha$ is unsatisfiable. So if we can show $\neg\alpha$ is unstaisfiable, we can infer that $KB \models \alpha$.

The primary function is `pl_resolution(KB, a)`. The function is called with a KB and α. It is important to note the KB must be in conjunctive normal form. The function written to convert a node into CNF is discussed in a later section. A node in conjunctive normal form will have a maximum depth of 4. The root node (level=0) is the $\wedge$ node connecting the statements in the KB. The children of this node (level=1) will either be $\vee$, $\neg$ or a symbol, their children (level=2) will either be a $\neg$ or a symbol, and the final children will be symbols (level=3). With the node in CNF, we can extract the conjunctions because they are all children of the root node. The negation of the α is added to the list of clauses. A function, `clause_convert(node)` takes each clause (a node) and converts it into a hash table representing the conjunction. So the conjunction: $(A \vee B \vee \neg C)$ is converted to A: TRUE, B: TRUE, C: FALSE. Any resolution containing a tautology is removed. Each pair of clauses is then run through the function, `resolve_pair(c1, c2)`. If there are no possible resolutions, the pair is passed back to the updated list of clauses. If there is a tautology in the clause, the clause is removed. If the pair can be resolved, the pair is resolved and the updated clauses is returned to the list, only if the length of the new clause is less than the length of the original clause. If it's not, the original clauses are passed back. This prevents the list of resolutions from growing exponentially large. The function checks to make sure progress is being made. If progress is not made, the

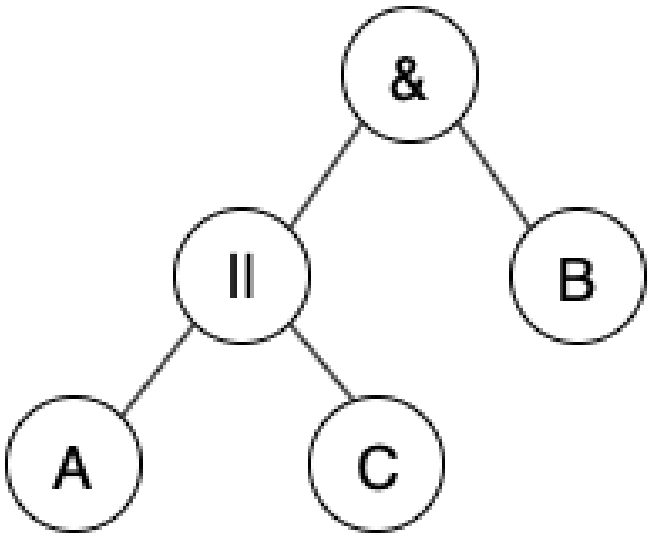

Figure 5: Node in CNF: $(A \vee C) \wedge B$

function returns FALSE when no more clauses can be resolved. If at any point two clauses resolve to the empty clause, $\neg\alpha$ can not be satisfied and therefore, we can infer that $KB \models \alpha$ and the function returns TRUE.

2.5 parse_logic.py functions

The two main functions in this file are `parse_logic(logic)` and `to_cnf(node)`. `parse_logic(logic)` takes in a string representing a statement of propositional logic and parses it into the node data structure used throughout the program. The statement must be in fully parenthesized form with no spaces. The parentheses are required because a stack is used to keep track of nested clauses. A parse tree is used to convert the list of symbols into a node representing those symbols. The node shown in Figure 3 would be entered as `parse_logic('((P&Q)&S)')`. Once the statement is in a tree structure, the tree is flattened into CNF using the function `to_cnf(node)`. Again, this is done utilizing basic tree traversal with DFS. This follows the method shown in the text to convert to CNF. First, biconditionals are replaced with conditionals. Next, conditionals are replaced. DeMorgan's law is applied, and double negations are removed. Finally, $\vee$ is distributed over $\wedge$. Throughout the process, the node is flattened to remove excess $\vee$ and $\wedge$ nodes.

2.6 problems.py

This file is where all of the code is tested on the sample problems provided. There are five main functions. `create_alpha(symbol)` creates an α node, while `create_p[1-5]()` creates an instance of the KBs for problems 1-4(b). The results of the problems are discussed in a subsequent section.

3 MEMBER CONTRIBUTIONS

I (Erin Witmer) worked alone on this project. The project design, implementation, analysis and write-up were all done without authorized or unauthorized assistance.

4 PROBLEMS

There were four problems of varying complexity we were asked to use to test our program. The results are discussed below. Run time for all model checking was near zero, so run times shown are for resolution only.

4.1 Modus Ponens

α	model checking	resolution	run time (s)
Q	True	True	0

We can prove if P models Q and P is true, that Q is also true.

4.2 Wumpus World

α	model checking	resolution	run time (s)
¬P12	True	True	0

We can prove there is no pit in (1,2).

4.3 Horn Clauses

α	model checking	resolution	run time (s)
MY	False	False	0
¬MY	False	False	0
MA	True	True	0
HO	True	True	0

We can't prove that the unicorn is mythical, however we can prove it is magical and horned.

4.4 Doors of Enlightenment

α	model checking	resolution	run time (s)
X	True	True	0.02
Y	False	False	55.36
¬ Y	False	False	55.52
Z	False	False	52.26
¬ Z	False	False	55.86

We can't prove that doors Y or Z will lead to the inner sanctum, however we can prove that door X will.

In the modified version of the problem, I can not get the model checker and set of statements to resolve to True (which I assume, based on the way the question is written, is the correct answer). In all cases, the model checker and resolution prover return consistent answers. However, if I run the statement by H through the model checker and resolver, I can prove that H is true and can therefore prove that X is true by forward chaining. H is true, so A is true, A is true so X is true.

5 DISCUSSION

Overall, while the write up makes this project seem very straightforward, I found this to be a very challenging project. The basic model checker was much easier to implement because the algorithm was described in detail in the book and the hint to represent the statements as trees was helpful. The resolution prover was much more challenging, specifically how to resolve a pair of clauses. This was not described in great detail in the text. Writing the parser and cnf converter were somewhat challenging, but likely ended up saving time over the course of the project as hand coding the more complex propositional logic statements takes a lot of time.

REFERENCES

[1] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.