Proof by Minecraft

The greatest proof simulator in recorded history

UQCS Hackathon
Pajorn, Josiah, Matt, Gus, and Zwe
August 16, 2025

Abstract

Minecraft is widely known as a sandbox for creativity, but we extend its potential to the domain of formal reasoning: First Order Logic (FOL) and Boolean Algebra. Our project, Proof by Minecraft, transforms logical expressions into interactive visual proofs. Using Python, we parse and simplify expressions via trees and graph theory, then generate corresponding Minecraft circuits. A Tkinter-based interface allows users to input one or two expressions; the system produces a truth table, evaluates equivalence, and constructs a Minecraft representation of the logic. This approach makes formal logic both intuitive and engaging, offering a novel way to explore proofs—because sometimes, the best argument really is Proof by Minecraft.

Keywords: First Order Logic/Boolean Algebra (FOL), Graph Theory, Trees

Rationale - Key Ideas, Requirements, and Goals.

The main idea behind 'Proof by Minecraft' is to make a formal logic system which is not only rigorous but visually and interactively intuitive. It both provides a 'graphical' representation of logical expressions, and also allows users to interact with the circuit, using levers to change the value of the inputs, and to evaluate the expression. The system also provides definitive feedback like truth tables, and equivalences which are essential for rigorously evaluating logical expressions. By embedding First Order Logic (FOL) expressions within Minecraft, we provide users with a tangible way to explore equivalence, simplification, and proofs. Our rationale stems from three guiding goals: accessibility, interactivity, and clarity.

Key Ideas

- Each FOL expression is represented as a Minecraft world where levers act as input variables and redstone lamps display output values.
- When two expressions are entered, they are rendered side by side with shared inputs, enabling direct comparison.
- To evaluate equivalence, the program systematically generates all input cases (2^n possibilities) and maps them into Minecraft.

Requirements

- **Parsing and Representation:** Logical expressions must be parsed into a graph/tree structure using graph theory and algorithmic simplification.
- **User Interface:** A Python GUI (Tkinter) must allow users to input expressions, following Nielsen's 10 heuristics for usability.
- **Minecraft Integration:** The parsed structure must be translated into a Minecraft world file, ensuring correct placement of redstone, lamps, and levers.

Goals

- Provide an engaging, gamified method to explore logical proofs.
- Demonstrate how graph theory and algorithms can be applied creatively.
- Produce both a truth table and equivalence check alongside the Minecraft visualization.

Architecture and methodology

1 System Design

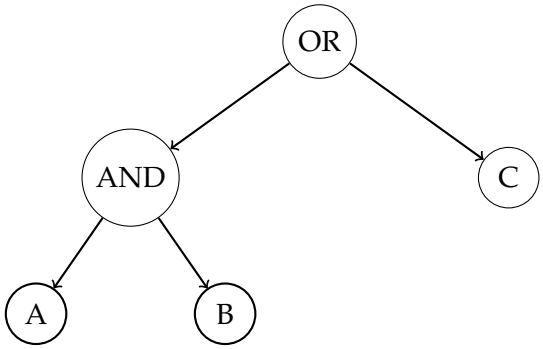
The overall architecture of *Proof by Minecraft* can be broken into four main components: parsing, graph representation, evaluation, and Minecraft world generation. Together these steps form a pipeline: a user enters one or two First Order Logic (FOL) expressions, the system parses and evaluates them, and then produces both a truth table and a Minecraft redstone circuit.

Parser

- Logical expressions are taken as strings and decomposed recursively.
- Operators supported include AND ($\wedge$), OR ($\vee$), and NOT ($\sim$).
- Bracketing ensures operator precedence is correctly maintained, with special care taken for NOTs.
- The parser outputs a clean, bracketed form ready for tree construction.
- We only need AND, OR, and NOT operators, as these are sufficient to express any logical expression. (Thank you Math1081)
- So our system will reduce any logical expression to a form that only uses these operators, then evaluates and compares them.

Graph Representation

- Each expression is represented as a binary tree where nodes correspond to operations or variables.
- Internal nodes store logical operators, while leaves represent input variables.
- Parent/child relationships encode logical structure, enabling recursive traversal.
- Example: $(a \wedge b) \vee c$ becomes an OR node at the root, with an AND node and variable c as children.



- This tree will functionally behave the same way, but will appear slightly different to a logic gate. The way you 'read', (and our algorithm reads) this tree. Is you've decomposed the expression into two remaining parts (think of order of operations) firstly you have the LHS $(a \wedge b)$ and then the RHS $\vee c$.
- The following is the Logic gate representation for the aforementioned expression $(a \wedge b) \vee c$:

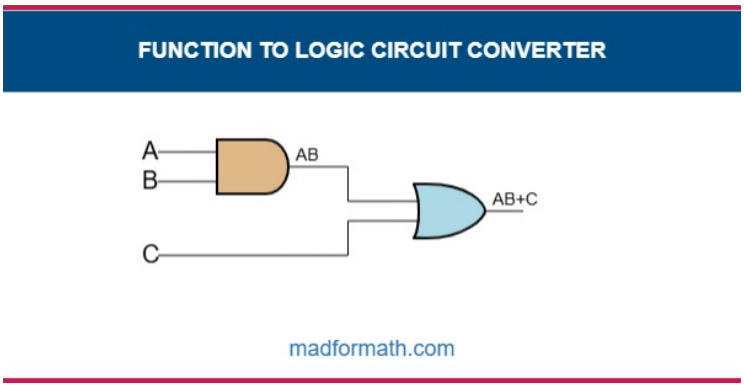


Figure 1: Logic gate representation of the expression $(a \wedge b) \vee c$.

Evaluation

- Once a tree is constructed, it can be recursively evaluated (with the method above) under any assignment of truth values.
- This process generates a complete truth table for the input variables.
- Equivalence between two expressions is determined by comparing truth tables row by row. Every row of each truth table must be exactly equal for the expressions to be considered equivalent.

– I will show an example of this in the following section, to show you how you (and our algorithm) can evaluate the equivalence of two expressions.

Comparison of Two Equivalent Expressions

We want to compare:

$$(A \wedge B) \vee C \quad \textit{and} \quad (A \vee C) \wedge (B \vee C)$$

A	B	C	$(A \wedge B) \vee C$	$(A \vee C) \wedge (B \vee C)$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Since the truth tables are identical, the two expressions are equivalent. This shows how the algorithm evaluates different syntactic forms but produces the same output. It essentially compares these two matrices line by line, and if they are equal, then the two expressions are equivalent. These truth tables will be produced in minecraft too, but with redstone lamps representing the truth values (on for true, off for false).

Minecraft World Generation

– Trees are translated into redstone contraptions:
  * Variables $\rightarrow$ levers.
  * Outputs $\rightarrow$ redstone lamps.
  * Logical gates $\rightarrow$ redstone structures implementing AND, OR, NOT.
– If two expressions are entered, both are placed side by side with shared inputs for direct comparison.
– To demonstrate equivalence, all 2^n possible input combinations can be rendered, allowing visual inspection of outputs.

Conclusion

References