Automated Theorem Proving in Intuitionistic Propositional Logic

Gabriel Wu, Montgomery Blair High School

Under the guidance of William Gasarch, University of Maryland Department of Computer Science

November 2020

Abstract. Automated theorem proving uses algorithms to search for mathematical proofs. This paper describes three original theorem provers that operate in a branch of logic that lacks the law of excluded middle ($P \vee \neg P$), called intuitionistic propositional logic. One prover employs a randomized depth-first search (DFS) to construct a proof tree, another uses DFS with memoization, and the third uses a DFS in LJT sequent calculus. After timing the provers on test cases from the Intuitionistic Logic Theorem Proving Library, it was found that the LJT prover was the most efficient on all tests and the randomized DFS prover was the slowest. While these provers are all significantly slower than most modern techniques, the simplicity of the algorithms allows those without extensive backgrounds in logic to explore automated theorem proving.

Keywords: automated theorem proving, intuitionism, sequent calculus

1 Introduction

Logic is the systematic study of inference and deduction. It is considered a sub-field of both math and philosophy. We can establish the truth of a mathematical statement by constructing a proof that follows the structure of a proof system. For large propositions, finding such a proof may be difficult – in fact, intuitionistic propositional logic has been shown to be PSPACE-complete (therefore it is thought to be hard to solve) [9]. Thus, mathematicians use automated theorem provers to leverage computational power for complex proof searches. This paper describes three intuitionistic provers that take any propositional formula F as input, then attempt to either prove F or determine that F is not a theorem.

2 Preliminaries

2.1 Propositional Logic

In propositional logic, variables can be assigned the value of *True* or *False* and are usually represented by a lowercase letter. Variables can be combined using logical *connectives* to form propositional *formula*. Formally, a propositional formula A must be of the form p, $\top$, $\bot$, $B \wedge C$, $B \vee C$, $B \to C$, or $\neg B$, where p is a variable and B and C are other propositional formulas. In English, these represent a variable, True, False, B *and* C, B *or* C, B *implies* C, and *not* B, respectively. These connectives can be used to express other operators such as $\leftrightarrow$ and $\oplus$. In this paper, I will use the uppercase letters for formulas and lowercase letters for variables.

The value of a propositional formula can be either True or False, defined recursively based on the values of its immediate sub-formulas:

B	C	$B \wedge C$	$B \vee C$	$B \to C$	$\neg B$
T	T	T	T	T	F
T	F	F	T	F	F
F	T	F	T	T	T
F	F	F	F	T	T

A formula is considered a *tautology* if it is True no matter how one assigns values to its variables. This corresponds to a column full of T's in a truth-table. Some examples of tautologies are $p \vee \neg p$, $(p \to q) \to (q \vee \neg p)$, and $(p \wedge q) \to (p \vee q)$.

2.2 Intuitionistic Logic

Intuitionistic logic encompasses the study of logical reasoning when only constructive inferences are allowed. In intuitionism, the only valid mathematical arguments are those where an object of interest can be constructed from a proof.

For example, consider the following *non-constructive* proof that there exist two irrational numbers a and b such that a^b is rational [1]. Let $a = b = \sqrt{2}$. If $\sqrt{2}^{\sqrt{2}}$ is rational, we are done. Otherwise, let $a = \sqrt{2}^{\sqrt{2}}$ (which we are now assuming to be irrational) and $b = \sqrt{2}$. Then, $a^b = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2 \in \mathbb{Q}$. While the use of disjoint cases seen here is perfectly reasonable in our classical mode of reasoning, this would not be a valid proof in constructive logic. The fact that we do not know if $\sqrt{2}^{\sqrt{2}}$ is rational means that we cannot provide definite values for a and b.

On a propositional level, intuitionistic logic can be described as classical logic without the law of the excluded middle ($P \vee \neg P$) or double negation elimination ($\neg\neg P \to P$) [7]. Logical connectives in intuitionistic logic have constructive interpretations which differ from classical logic. To prove $P \vee Q$, we must either have a proof of P or a proof of Q. A proof of $P \to Q$ is an algorithm which can convert a proof of P into a proof of Q. $\neg P$ can be interpreted as $P \to \bot$, meaning we can derive a contradiction from P.

As a result, many tautologies in classical propositional logic are not valid theorems in intuitionistic logic. Some examples include $\neg(p \wedge q) \to (\neg p \vee \neg q)$ and $(p \wedge (\neg q \to \neg p)) \to q$. However, since intuitionistic deduction is strictly weaker than classical deduction, any intuitionistic theorem must also be a classical theorem. In other words, intuitionistic theorems make up a subset of classical tautologies.

2.3 Sequent Calculus

Gentzen-style sequent calculus is one of the many proof systems used to establish the truth of propositional statements. In the sequent calculus used for intuitionistic proofs, called the LJ system, sequent statements take the form:

$$A_1, A_2, \ldots, A_n \vdash B$$

The formulas $\{A_i\}$ to the left of the turnstile are called the *antecedents*, while the formula B to the right of the turnstile is called the *consequent* [5]. Semantically, the sequent is equivalent to $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \to B$. Thus, proving the formula P from no premises is equivalent to deriving the sequent "$\vdash P$". Although there are formal structural rules dealing with commuting $\{A_i\}$, in this paper the antecedents will be treated as unordered.

A proof in sequent calculus consists of a tree of sequents, where each sequent follows logically from zero (if it is an axiom) or more "child sequents", according to the rules of inference. The inference rules for the LJ system are as follows [5]:

$$\frac{}{A, \Gamma \vdash A}\text{ Init} \qquad \frac{}{\bot, \Gamma \vdash G}\ \bot\text{-Left} \qquad \frac{A, B, \Gamma \vdash G}{A \wedge B, \Gamma \vdash G}\ \wedge\text{-Left} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-Right}$$

$$\frac{A, \Gamma \vdash G \qquad B, \Gamma \vdash G}{A \vee B, \Gamma \vdash G}\ \vee\text{-Left} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-Right}_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-Right}_2$$

$$\frac{A \to B, \Gamma \vdash A \qquad B, \Gamma \vdash G}{A \to B, \Gamma \vdash G}\ \to\text{-Left} \qquad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}\ \to\text{-Right}$$

In these rules, A, B, and G represent arbitrary formulas. Γ represents a set of additional antecedent formulas (possibly empty). Fig. 1 gives an example proof tree which establishes the commutativity of $\vee$ using these inference rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\,\text{Init}}{P \vdash Q \lor P}\,\lor\text{-R}_2 \qquad \dfrac{\dfrac{}{Q \vdash Q}\,\text{Init}}{Q \vdash Q \lor P}\,\lor\text{-R}_1}{P \lor Q \vdash Q \lor P}\,\lor\text{-L}}{\vdash (P \lor Q) \to (Q \lor P)}\,\to\text{-R}$$

Figure 1: An LJ proof of $(P \lor Q) \to (Q \lor P)$

A modification of the LJ system, called LJT, replaces the $\to$-Left inference with four new inference rules. LJT is logically equivalent to LJ, with the benefit that all inference rules sprout child sequents that are smaller than the parent sequent. This ensures that any proof search must eventually terminate. The four new inference rules that replace $\to$-Left are [3]:

$$\dfrac{B, a, \Gamma \vdash G}{a \to B, a, \Gamma \vdash G}\,\to\text{-Left}_1\ (a \text{ is atomic}) \qquad \dfrac{C \to (D \to B), \Gamma \vdash G}{(C \land D) \to B, \Gamma \vdash G}\,\to\text{-Left}_2$$

$$\dfrac{C \to B, D \to B, \Gamma \vdash G}{(C \lor D) \to B, \Gamma \vdash G}\,\to\text{-Left}_3 \qquad \dfrac{D \to B, \Gamma \vdash C \to D \qquad B, \Gamma \vdash G}{(C \to D) \to B, \Gamma \vdash G}\,\to\text{-Left}_4$$

3 Methods

3.1 Implementation

The automated theorem provers are coded in Python 3. Propositional formulas are represented recursively in a `Sentence` class. Each `Sentence` object is either a propositional variable, or it stores up to two children `Sentence` objects along with a logical connective. Storing a formula this way creates an expression tree which ensures there is a single connective on the highest level. This makes it easy to determine applicable inference rules during the proving phase. Fig. 2 shows the expression tree of an example formula, where each node represents a `Sentence` object.

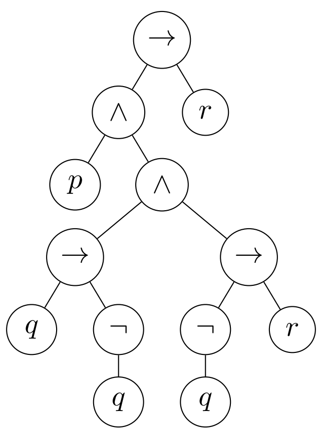

Figure 2: The expression tree of $(p \land (p \to \neg q) \land (\neg q \to r)) \to r$

Each `Sequent` stores a list of `Sentence` objects in the antecedent, as well as a single `Sentence` succedent as described in the LJ intuitionistic sequent calculus system. When comparing two `Sequent` classes for equality, I use the hashing function:

$$H(seq) := \left(\sum_{x \in ant} h_1(x) \right) + h_2(suc)$$

where ant is a list of antecedent formulas, suc is the succedent formula, and h_1 and h_2 are two built-in string hash functions. For the purposes of hashing, the propositional formulas x and suc were represented as strings. This particular hashing function was chosen because it remains invariant to permutations of the antecedent formulas, which is important because the antecedents are unordered.

3.2 Prover Framework

When a propositional formula F is submitted to the prover, it does two things before calling the principle `prove` method. First, it calls a Boolean satisfiability solver, commonly known as a SAT solver, on $\neg F$. The SAT solver checks if there is any setting of variables that makes $\neg F$ true. If so, then F itself cannot be a tautology, meaning it cannot be proven in a classical logic system. Since intuitionistically valid theorems must be classical tautologies, the method automatically returns `False`. I coded my own SAT solver which converts the formula into conjunctive normal form, then uses backtracking and watchlists to efficiently check for satisfiability. While it could be sped up using more advanced SAT methods, the formulas I used on my prover were short enough for this simple solver.

Once F is found to be a tautology, it is converted into a modified formula F' where all occurrences of $\neg A$ were replaced with $A \to \bot$, as specified in the intuitionistic sequent calculus. Formally, $F' = conv(F)$ where

$$conv(F) := \begin{cases} conv(A) \to \bot & \text{if } F = \neg A \\ conv(A) \;\; op \;\; conv(B) & \text{if } F = A \; op \; B, \;\text{ for } op \in \{\wedge, \vee, \to\} \\ p & \text{if } F = p \end{cases}$$

Finally, F' is sent to one of three principal intuitionistic prover methods, detailed below.

3.3 Randomized Depth-First Search

The first prover method uses a depth-first search with randomized inference rules to construct the proof tree. When called to prove a goal sequent S, the method first determines the valid inference rules that could be applied to derive S. It cycles through these valid inference rules in a random order, recursively calling itself on the resultant child sequents. If none of the inference rules produce children that were able to be proved, the method returns false – the prover was unsuccessful at proving S. However, if the method is able to prove all of the children of a certain inference rule, it returns true because it has generated a valid proof tree for S. To prevent infinite recursion, there is a `max_depth` variable that caps the height of the proof tree, typically set to 1000.

Algorithm 1 gives pseudocode for this randomized DFS process.

```
Define random_dfs($S$, $curr\_depth$)
    /* Input:  Sequent $S$ to be proved, current depth in proof tree    */
    /* Output:  Whether or not $S$ was successfully proved              */
    if $curr\_depth > max\_depth$
        return False
    $valid\_rules = \{\}$
    for $inference$ in $INFERENCE\_RULES$
        if $inference$ could lead to $S$
            Add $inference$ to $valid\_rules$
    Randomly shuffle $valid\_rules$

    for $inference$ in $valid\_rules$
        Let $C$ be the list of sequent premises required for $inference$ to derive $S$
        Add $C$ as children to $S$ in the proof tree
        $success$ = True
        for $child$ in $C$
            if not random_dfs($child$, $curr\_depth+1$)
                $success$ = False
                break
        if $success$
            return True
        else
            Remove all children of $S$ in the proof tree
    return False
```

Algorithm 1: Randomized DFS prover method

3.4 DFS with Memoization

Sometimes equivalent sequents can appear in multiple places in a single proof tree. One weakness of the randomized DFS prover described in section 3.3 is that it does not remember what it has already seen, so it may end up in an infinite recurrence trying to prove the same sequent multiple times. This decreases its efficiency. To eliminate redundant computation, we can store a global record of previous calls to the prover method. This technique, known as memoization, allows the prover to reference previous parts of the proof tree when it encounters a duplicate sequent. It also lets the prover give up early when it encounters a sequent it has already unsuccessfully tried to prove, which saves computational time.

I implemented memoization with a hash table, also called a "dictionary" in Python. Using the hashing function described in Section 3.2, the table stores {`sequent: depth`} pairs. Each entry represents a previous `dfs_memoization` call on `sequent` at a certain `depth`. Each entry can also be marked as *complete* to indicate that it has been successfully proven. The pseudocode for this method is given in Algorithm 2. Notice that in marked line 1, the prover only abandons its search if a previous call to S was unsuccessful *and was started at a smaller depth.* This is important because, with more room to grow until reaching `max_depth`, the current call may find a proof which the previous call did not have the space to explore. In marked line 2, the current call is labeled complete before exiting so that future calls can reference the same proof subtree.

```
Define dfs_memoization($S$, $curr\_depth$)
    /* Input: Sequent $S$ to be proved, current depth in proof tree    */
    /* Output: Whether or not $S$ was successfully proved              */
    if $curr\_depth > max\_depth$
        return False
    if $S$ in $all\_calls$                    /* Previous attempt to prove S */
        Let $P$ be the previous call of $S$ in $all\_calls$
        if $P$ is marked complete              /* P has already been proven */
            Set the children of $S$ to point to $P$
            return True
        else                                   /* Failed to prove P previously */
1           if depth of $P \leq curr\_depth$
                return False
    Add $\{S, curr\_depth\}$ to $all\_calls$

    $valid\_rules = \{\}$
    for $inference$ in $INFERENCE\_RULES$
        if $inference$ could lead to $S$
            Add $inference$ to $valid\_rules$
    Randomly shuffle $valid\_rules$

    for $inference$ in $valid\_rules$
        Let $C$ be the list of sequent premises required for $inference$ to derive $S$
        Add $C$ as children to $S$ in the proof tree
        $success$ = True
        for $child$ in $C$
            if not dfs_memoization($child$, $curr\_depth+1$)
                $success$ = False
                break
        if $success$
2           Mark $S$ as complete in $all\_calls$
            return True
        else
            Remove all children of $S$ in the proof tree
    return False
```

Algorithm 2: DFS with memoization prover method

3.5 LJT Sequent Calculus

While the first two provers employ LJ sequent calculus, the third prover method searches for proofs in the LJT sequent calculus. As discussed in Section 2.3, the LJT calculus is logically equivalent to the LJ calculus. However, LJT is useful for automated theorem proving because all children sequents are guaranteed to be smaller than their parent sequent. This means that the LJT proof of any propositional sentence is bounded in depth, often leading to more concise proof trees. The `dfs_LJT` method was implemented the same way as the `dfs_memoization` method, except the →-Left LJ inference rule was replaced with its LJT counterparts.

Table 1: ILTP problems used to test `random_dfs`, `dfs_memoization`, and `dfs_LJT`

Problem	Intuit. status	Description
SYJ201	Theorem	$[((p_1 \leftrightarrow p_2) \rightarrow p_1) \wedge ((p_2 \leftrightarrow p_3) \rightarrow (p_1 \wedge p_2)) \wedge \cdots \wedge ((p_{2N+1} \leftrightarrow p_1) \rightarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_{2N+1}))] \rightarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_{2N+1})$
SYJ202	Theorem	Pigeonhole principle with N holes and $N+1$ pigeons
SYJ203	Theorem	Double negation of a De Morgan's-style statement: $\neg\neg[(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \vee (\neg p_1 \vee \neg p_2 \vee \cdots \vee \neg p_n)]$
SYJ204	Theorem	$[p_n \wedge (p_1 \rightarrow (p_1 \rightarrow p_0)) \wedge (p_2 \rightarrow (p_2 \rightarrow p_1)) \wedge \cdots \wedge (p_n \rightarrow (p_n \rightarrow p_{n-1}))] \rightarrow p_0$
SYJ205	Theorem	Complex statement involving the conjunction of many implications.
SYJ206	Theorem	Reverse of a $\leftrightarrow$ permutation: $(\ldots(((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \leftrightarrow p_4) \cdots \leftrightarrow p_n) \leftrightarrow (\ldots((p_n \leftrightarrow p_{n-1}) \leftrightarrow p_{n-2}) \cdots \leftrightarrow p_1)$
SYJ207	Non-theorem	$[((p_1 \leftrightarrow p_2) \rightarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_{2n})) \wedge ((p_2 \leftrightarrow p_3) \rightarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_{2n})) \wedge \cdots \wedge ((p_{2n} \leftrightarrow p_1) \rightarrow (p_1 \wedge p_2 \wedge \cdots \wedge p_{2n}))] \rightarrow [p_0 \vee (p_1 \wedge p_2 \wedge \cdots \wedge p_{2n}) \vee \neg p_0]$
SYJ208	Non-theorem	Pigeonhole principle with a double negation
SYJ209	Non-theorem	SYJ203 with $\neg\neg\neg p_1$ instead of $\neg p_1$
SYJ210	Non-theorem	SYJ204 with $\neg\neg p_n$ instead of p_n
SYJ211	Non-theorem	SYJ205 slightly modified and with double negations
SYJ212	Non-theorem	SYJ206 with $\neg\neg p_1$ instead of p_1

3.6 The Intuitionistic Logic Theorem Proving Library

The Intuitionistic Logic Theorem Proving (ILTP) Library provides a collection of intuitionistic problems commonly used to benchmark automated theorem provers [8]. The SYJ section of this library includes propositional intuitionistic problems, including both theorems and non-theorems. Some of these problems represent common ideas – for example, SYJ202 represents the pigeon hole principle with N holes and $N + 1$ pigeons. Other SYJ problems do not have simple interpretations. The three methods were tested on the ILTP (version 1.1.2) problems listed in Table 1.

4 Results

4.1 Data

A call to a prove method was considered successful if it terminated within 5 seconds, by either finding a valid proof or returning a verdict that no proof exists. Note that these prover methods conduct exhaustive searches, meaning that if a proof with a height less than `max_depth` exists, it will be found eventually – although it may take a long time to terminate. This observation eliminates the possibility of false negative verdicts.

Among all test cases, the `dfs_LJT` method had a higher success rate than the other two methods. Fig. 3 shows that neither `random_dfs` nor `dfs_memoization` terminated for any cases with $N \geq 11$. For test cases with $N \leq 10$, `dfs_memoization` had more success than `random_dfs`.

The test sets varied in difficulty, so it necessary to examine individual test sets when comparing the provers' run speed. Fig. 4 shows the run time (averaged over 4 trials) of each method on the SYJ211 test cases. Points on this scatter plot were removed if the method took longer than 5 seconds to run, which is why the data is cut off at different N values. `dfs_LJT` was far more efficient than the other two methods, making it to $N = 15$ with under 5 second run times.

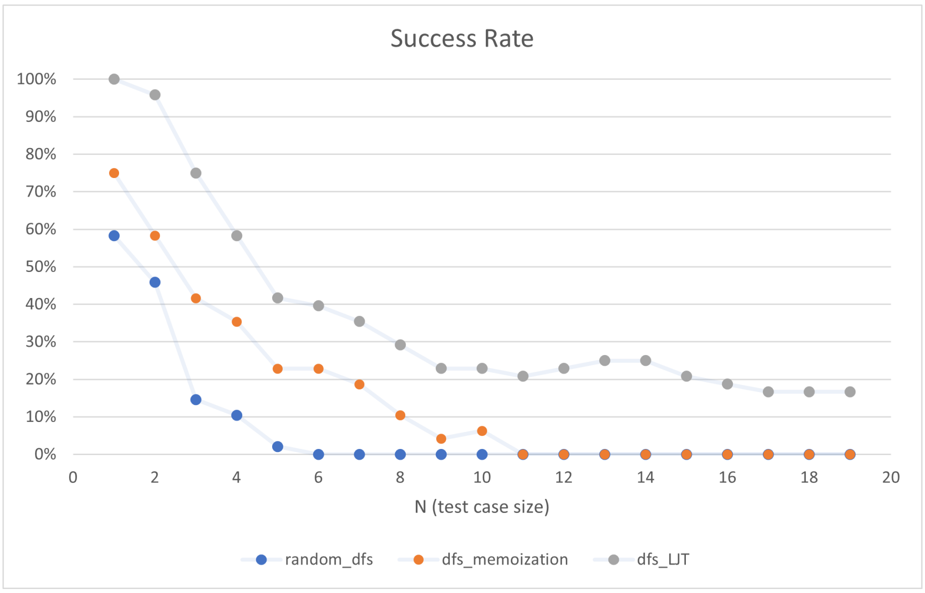

Figure 3: Success rates of each method over all chosen ILTP test cases with $N \leq 20$

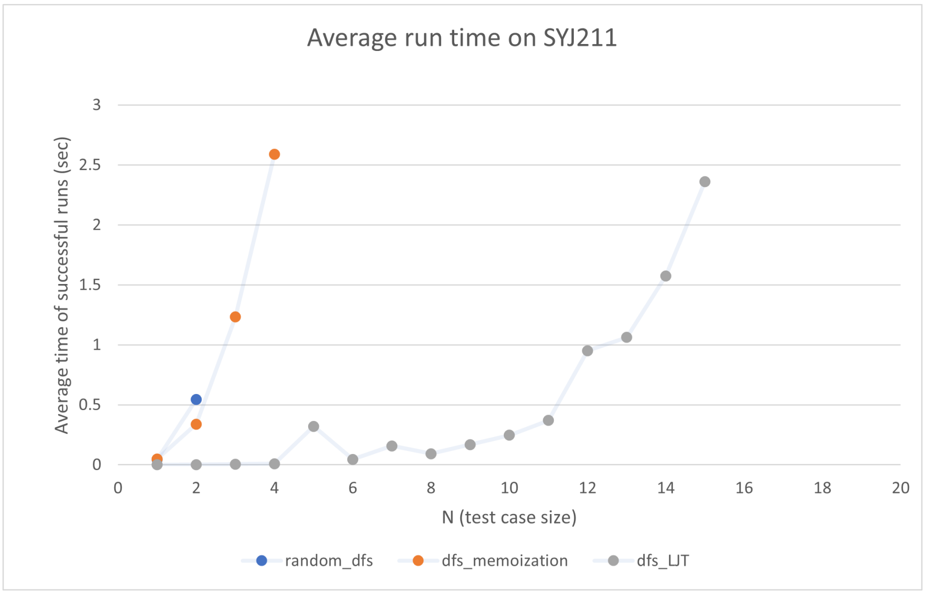

Figure 4: Average run time of each method on SYJ211 test cases

Table 2: Proportion of tautologies that are intuitionistically valid

		Formula Size *(n)*							
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Max distinct variables *(k)*	**1**	48%	62%	64%	64%	63%	67%	68%	72%
	2	51%	51%	55%	64%	62%	63%	67%	71%
	3	55%	58%	61%	60%	59%	57%	58%	59%
	4	50%	49%	53%	60%	59%	58%	61%	62%
	5	54%	50%	61%	53%	60%	62%	49%	58%
	6	53%	54%	42%	57%	54%	61%	54%	66%
	7	59%	50%	56%	50%	50%	54%	58%	53%
	8	53%	46%	56%	54%	58%	54%	57%	56%
	9	56%	55%	50%	61%	56%	62%	64%	57%

4.2 Additional Experiments

Although it is not related to measuring the effectiveness of the automated prover methods, I was interested in investigating the fraction of propositional tautologies in classical logic that can be proven intuitionistically. To measure this proportion, I created a random propositional formula generator `rand_sentence(n, k)` which selects a propositional sentence at random from the set of all propositional sentences of length n with at most k distinct variables. This method involved generating a random binary tree with $2n - 1$ leaf nodes using the algorithm described in Fusy [4], then filling each leaf node with a random variable from $\{P_1, P_2, \ldots, P_k, \neg P_1, \neg P_2, \ldots, \neg P_k\}$. Each non-leaf node was assigned a random propositional connective from $\{\wedge, \vee, \rightarrow\}$, then passed through a $\neg$ operator with a $\frac{1}{2}$ chance.

Random propositional sentences were generated until 200 tautologies had been collected for each pair of (n, k) values less than or equal to 9. These tautologies were passed into the `dfs_LJT` method. Table 2 shows the proportion of tautologies were intuitionistically valid for each (n, k) pair. It is interesting to note that tautologies are more likely to be intuitionistically valid when they are longer and have many repeated variables.

5 Discussion

The results of the experiment indicate that the `dfs_LJT` method is superior to the other prover methods which use the traditional LJ sequent calculus. This finding is to be expected since the LJT sequent calculus was designed to be contraction-free, making it more suitable to automated theorem proving. The data also confirms the hypothesis that memoization provides a big improvement to the efficiency of a basic DFS prover, which makes sense because it prevents the prover from getting stuck in infinite loops.

It is interesting to note that, although SYJ209 and SYJ211 are listed as non-theorems in the ILTP library, my programs succeeded in finding proofs for these tests. I confirmed by hand that the proofs were valid. It is possible that SYJ209 and SYJ211 were mislabelled in the official library, but it is more likely that I made a mistake – either by incorrectly transcribing the tests or missing a flaw in the proofs. After the data was collected, I found mistakes in the transcription of tests SYJ206 and SYJ207. These errors did not have a large effect on the results.

Previous research has already established many powerful automated theorem proving techniques in intuitionistic logic, most of which are far more efficient than the the algorithms discussed in this paper. These papers include industry standards such as the Coq `tauto` tactic [2], as well as provers employing deep reinforcement learning [5] and focused polarization [6]. Considering the abundance of previous literature, the techniques presented in this paper do not offer cutting-edge advancements to intuitionistic provers. However, the simplicity of these algorithms offer insight into the structure of automated provers, allowing those without extensive backgrounds in logic to appreciate and explore automated theorem proving and intuitionistic logic itself.

I developed the memoization technique with inspiration from problems in graph theory and dynamic programming. The effectiveness of the basic optimizations presented here demonstrate that perhaps future breakthroughs in automated reasoning will come from simple ideas drawn from other areas of computer science. Advancements in automated theorem proving would improve proof assistants and proof verification programs often used by mathematicians. Faster proving methods would also benefit industrial settings, where automated theorem provers are used to verify integrated circuit designs and catch potential logic errors in CPUs.

Exploring intuitionistic logic, as well as other non-classical logical frameworks, gives logicians a better understanding of the structure of argumentation itself. Intuitionism may reveal new, potentially useful models of understanding truth, in the same way that Euclid's neutral geometry helped mathematicians discover elliptical and hyperbolic geometries. From a philosophical standpoint, it is important to examine the foundations of mathematics, even something as fundamental as the law of the excluded middle. Ideas and insights gained from intuitionistic theorem proving may also inspire similar discoveries in classical logic.

In the future, I would like to extend these prover methods to first-order intuitionistic logic by adding deduction rules for quantifiers. It would also be worthwhile to explore proof systems other than sequent calculus, such as resolution or natural deduction.

6 Code

The Python code for this project can be found on GitHub at `https://github.com/GabrielDWu/intuitionistic-theorem-proving`

Acknowledgements

I would like to thank Professor William Gasarch at the University of Maryland for his insight and advice during the research process. His guidance helped me convert my initial ideas into concrete research goals. I am also grateful to Ms. Angelique Bosse for her coordination of the Blair Senior Research Project program.

References

[1] D. Bridges and E. Palmgren, “Constructive Mathematics”, in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2018, Metaphysics Research Lab, Stanford University, 2018. [Online]. Available: `https://plato.stanford.edu/archives/sum2018/entries/mathematics-constructive/`.

[2] *Coq reference manual*, version 8.12.0. [Online]. Available: `https://coq.inria.fr/refman/`.

[3] R. Dyckhoff, “Contraction-free sequent calculi for intuitionistic logic”, *The Journal of Symbolic Logic*, vol. 57, no. 3, pp. 795–807, 1992.

[4] E. Fusy, *Random generation*, 2011. [Online]. Available: `http://www.lix.polytechnique.fr/Labo/Eric.Fusy/Teaching/notes.pdf`.

[5] M. Kusumoto, K. Yahata, and M. Sakai, “Automated theorem proving in intuitionistic propositional logic by deep reinforcement learning”, *CoRR*, 2018. arXiv: `1811.00796`. [Online]. Available: `http://arxiv.org/abs/1811.00796`.

[6] S. McLaughlin and F. Pfenning, “Imogen: Focusing the polarized inverse method for intuitionistic propositional logic”, in *Lecture Notes in Computer Science*, I. Cervesato, H. Veith, and A. Voronkov, Eds., 2008, pp. 174–181.

[7] J. Moschovakis, “Intuitionistic Logic”, in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Winter 2018, Metaphysics Research Lab, Stanford University, 2018. [Online]. Available: `https://plato.stanford.edu/archives/win2018/entries/logic-intuitionistic/`.

[8] T. Raths, J. Otten, and C. Kreitz, “The ILTP Problem Library for Intuitionistic Logic”, *Journal of Automated Reasoning*, 2006.

[9] R. Statman, “Intuitionistic propositional logic is polynomial-space complete”, *Theoretical Computer Science*, vol. 9, no. 1, pp. 67–72, 1979.