Experimental Analysis of Sat Solving Algorithms

Andrew Zhao

September 25, 2025

1 Introduction

1.1 Overview of the Field of Study

The Boolean Satisfiability Problem, commonly called SAT, is a key puzzle in computer science and mathematics. It asks whether a logical formula, written as a combination of statements connected by "and," "or," and "not," can be made true by choosing the right true or false values for its variables. For example, consider the formula $(x \vee y) \wedge (\neg x \vee z)$, where $\vee$ means OR, $\wedge$ means AND, and $\neg$ means NOT. This means "(x or y) and (not x or z)." The goal is to find values for x, y, and z (true or false) that make the whole formula true. If such values exist, the formula is "satisfiable"; if not, it is "unsatisfiable."

SAT is important because it is a very hard problem to solve for large formulas. In 1971, a researcher named Stephen Cook showed that SAT is "NP-complete," meaning that while checking a solution is quick, finding one can take a very long time because the number of possible combinations grows rapidly [Cook, 1971]. This makes SAT a benchmark for testing how well computers can handle tough problems.

SAT is useful in many areas. In software development, SAT solvers (programs that solve SAT problems) help find bugs in code by checking if a program behaves correctly. In hardware design, they ensure circuits work as intended before they are built. In artificial intelligence, SAT helps robots plan actions or solve puzzles. It is also used in cryptography to test security and in biology to study things like protein shapes or DNA sequences. It is also used in scheduling problems, the Traveling Salesman Problem, and Integer Programming. Improving SAT solvers can make these tasks faster and more reliable.

SAT solvers have come a long way. In 1962, researchers developed the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which systematically tries different variable assignments [Davis et al., 1962]. DPLL uses tricks like "unit propagation," which means automatically setting a variable when only one choice makes sense (e.g., if a clause is just "x," then x must be true). Another trick is "pure literal elimination," where if a variable only appears as true (or false), it can be set that way without guessing. Later, in the 1990s, solvers added "conflict-driven clause learning" (CDCL), which learns from mistakes to avoid repeating them. Modern solvers like MiniSat, Glucose, and Kissat can handle huge problems with millions of variables, but some special cases, like those based on mathematical principles, remain very hard.

1.2 Scientific Justification for the Research

This project is motivated by the fact that SAT problems are extremely difficult to solve, even for powerful computers. SAT solvers often struggle with large or cleverly structured formulas, where the number of possible variable combinations is enormous—like 2^{100} for 100 variables, which is more than the number of atoms in the universe. A classic example of this difficulty is the pigeonhole principle (PHP), which is simple to understand but very hard for solvers. The PHP says that if you have more pigeons (n) than holes (m), with n greater than m, at least one hole must have more than one pigeon. When turned into a SAT formula, this is always unsatisfiable, but proving it requires the solver to check a huge number of options, making it a tough test case that shows the limits of SAT solving techniques.

To explore this difficulty, I tested how different strategies, called "heuristics," affect how fast a SAT solver can work. A heuristic is like a rule of thumb that helps the solver decide which variable to try next when guessing true or false. Choosing the right variable can make the solver much faster by reducing unnecessary guesses. I focused on PHP formulas because they are a good way to show SAT's challenges. I tested five heuristics: Random, DLIS, CDF, MOM, and JW. These heuristics are defined in the methodology section. I ran these on PHP formulas with varying n and m.

This research matters because SAT solvers are used in real-world tasks like scheduling workers, designing secure systems, or planning AI actions. By finding out which heuristics work best on hard problems like pigeonhole, we can make solvers better at handling difficult cases. The project also helps understand why some problems are harder than others and how to improve solver designs.

2 Methodology

The SAT solver was built in Python 3 using the DPLL algorithm. The SAT solver was developed in Python using the DPLL algorithm as the main framework. In the implementation, DPLL operates recursively: it selects an unassigned variable and explores both true and false assignments. During this process, unit propagation is applied, forcing assignments when a clause reduces to a single literal. If a contradiction arises, the algorithm backtracks and continues along a different branch until either a satisfying assignment is found or all possibilities are exhausted.Two key techniques make it faster:

- Unit Propagation: When a clause (a group of variables connected by "or") has only one variable left, that variable must be set to make the clause true. For example, if a clause is just (x), then x must be true. The solver applies this rule repeatedly to simplify the formula. - Backtracking: If a choice leads to a contradiction (e.g., a clause becomes impossible to satisfy), the solver goes back and tries a different choice.

Five heuristics were used to choose which variable to try next:

- Random: Picks a variable randomly, like flipping a coin. This is simple but not very smart.

- Dynamic Largest Individual Sum (DLIS): Counts how often each variable appears in the remaining clauses and picks the most frequent one, hoping to simplify the formula quickly.

- Conflict-Driven Frequency (CDF): Tracks which variables cause conflicts (dead ends) and picks the one with the most conflicts. If no conflicts are recorded, it uses DLIS.

- Maximum Occurrences in Minimum-length clauses (MOM): Looks at the shortest clauses (which are harder to satisfy) and picks the variable that appears most in them.

- Jeroslow-Wang (JW): Gives each clause a score based on its length (shorter clauses get higher scores, e.g., score = $2^{-\text{length}}$), adds up scores for each variable, and picks the highest. These were coded in a file called methods.py.

The JW heuristic code is shown in Code Block 1.

Block 1: The JW heuristic code

```python
from collections import Counter

def jw_method(clauses, unassigned, conflict_counts):
    if not unassigned:
        return None
    scores = Counter()
    for clause in clauses:
        weight = 2 ** -len(clause)
        for literal in clause:
            var = abs(literal)
            if var in unassigned:
                scores[var] += weight
    return max(unassigned, key=lambda x: scores[x]) if scores
       else None
```

Block 2: The pigeonhole formula generation code

```python
def generate_pigeonhole(n, m):
    num_vars = n * m  # Variables x_ij for i=1 to n, j=1 to m
    clauses = []

    # Constraint 1: Each pigeon is in at least one hole
    for i in range(1, n + 1):
        clause = [(i - 1) * m + j for j in range(1, m + 1)]
        clauses.append(clause)

    # Constraint 2: No two pigeons are in the same hole
    for j in range(1, m + 1):
        for i1 in range(1, n + 1):
            for i2 in range(i1 + 1, n + 1):
                var1 = (i1 - 1) * m + j
                var2 = (i2 - 1) * m + j
                clauses.append([-var1, -var2])

    return clauses, num_vars
```

The solver itself is in sat.py, which manages the DPLL process. For brevity, only the key structure is shown, but it handles clauses, assignments, and recursion.

For creating pigeonhole formulas P(n,m), I used pigeonhole.py. These use $n \times m$ variables x_{ij}, where x_{ij} means pigeon i is in hole j. The formula has two rules: - Each pigeon must be in

at least one hole: for each pigeon i, a clause like $(x_{i1} \vee x_{i2} \vee \cdots \vee x_{im})$. - No two pigeons can share a hole: for each hole j and pigeons i1, i2, a clause like $(\neg x_{i1\,j} \vee \neg x_{i2\,j})$.

The code that builds these formulas is shown in Code Block 2.

Since $n > m$, P(n,m) is always unsatisfiable. Experiments were run using an updated script that tests different families of P(pigeons, holes) instances, stopping when any heuristic exceeded 10 minutes (600 seconds) for an instance. This allowed us to see how time scales with size and classify if it's polynomial-like (slow growth) or exponential-like (fast growth). The families tested are P(n, n-1) (pigeons = n, holes = n-1), P(2n, n) (pigeons = 2n, holes = n), and P(10, n) (pigeons = 10, holes = n). All unsatisfiable instances have null assignment; satisfiable ones have an assignment dictionary (truncated for space). Tests were run on a standard desktop computer.

The maincode.py script is the main control program. With two arguments (n m), it solves a specific P(n,m); otherwise, it runs experiments with m=5 (adjustable). Tests were run on a standard desktop computer.

3 Results

The results were collected by running the updated script for the pigeonhole experiments. Below are the detailed results for each experiment, showing how long each heuristic took and whether each formula was satisfiable.

3.1 Pigeonhole Experiments

The pigeonhole experiments were conducted to test different families of P(pigeons, holes) instances, stopping when any heuristic exceeded 10 minutes (600 seconds) for an instance. This allowed us to see how time scales with size and classify if it's polynomial-like (slow growth) or exponential-like (fast growth). The families tested are P(n, n-1) (pigeons = n, holes = n-1), P(2n, n) (pigeons = 2n, holes = n), and P(10, n) (pigeons = 10, holes = n). All unsatisfiable instances have null assignment; satisfiable ones have an assignment dictionary (truncated for space).

P(n, n-1) Family (Pigeons = n, Holes = n-1, Start n=5) This is unsatisfiable since pigeons > holes. Methods ran until n=11, where some timed out.

Random:

n	Time (s)
5	0.000801
6	0.007432
7	0.076580
8	0.940765
9	12.470848
10	186.783055
11	null

DLIS:

n	Time (s)
5	0.000756
6	0.005384
7	0.040805
8	0.344360
9	3.372826
10	40.964139
11	524.746007

CDF:

n	Time (s)
5	0.000883
6	0.007581
7	0.075375
8	0.801038
9	10.031276
10	140.296020
11	null

MOM:

n	Time (s)
5	0.001044
6	0.008887
7	0.087848
8	1.040972
9	13.537162
10	214.038100
11	null

JW:

n	Time (s)
5	0.000795
6	0.005680
7	0.042291
8	0.367461
9	3.508375
10	45.712554
11	551.670325

n	DLIS	JW	MOM	CDF	Random
10	40.964 139	45.712 554	214.038 100	140.296 020	186.783 055
11	524.746 007	551.670 325	null	null	null

From the results, DLIS and JW perform significantly better than Random, CDF, and MOM, with shorter times for larger n. For example, at n=10, DLIS took 40.96s while Random took 186.78s. Given their efficiency, these heuristics could potentially be tested on larger instances like n=12, though they might still take considerable time.

Growth Classification: All methods show exponential-like scaling (ratios $>$1.5).

P(2n, n) Family (Pigeons = 2n, Holes = n, Start n=2) This is unsatisfiable for all tested n (pigeons $>$ holes). Methods ran until n=9, where some timed out.

Random:

n	Time (s)
2	0.000 040
3	0.000 346
4	0.004 832
5	0.049 846
6	0.953 052
7	14.752 042
8	297.328 010
9	null

DLIS:

n	Time (s)
2	0.000 046
3	0.000 388
4	0.003 319
5	0.024 375
6	0.266 352
7	2.639 716
8	28.771 470
9	340.844 093

CDF:

n	Time (s)
2	0.000 053
3	0.000 502
4	0.004 182
5	0.038 775
6	0.500 602
7	6.557 919
8	93.360 501
9	null

MOM:

n	Time (s)
2	0.000046
3	0.000393
4	0.003591
5	0.037159
6	0.534982
7	7.609976
8	113.870322
9	null

JW:

n	Time (s)
2	0.000049
3	0.000418
4	0.003632
5	0.029749
6	0.326885
7	3.181734
8	34.562265
9	422.863442

n	DLIS	JW	MOM	CDF	Random
8	28.771470	34.562265	113.870322	93.360501	297.328010
9	340.844093	422.863442	null	null	null

DLIS and JW again outperform the others, with DLIS taking 340s at n=9 compared to Random timing out.

Growth Classification: All methods show exponential-like scaling (ratios >1.5).

P(10, n) Family (Pigeons = 10, Holes = n, Start n=5) This is unsatisfiable for $n < 10$ (pigeons $>$ holes) and satisfiable for $n \geq 10$ (pigeons $\leq$ holes, with assignment shown truncated). Methods ran until n=205, where some timed out.

Random:

n	Time (s)
5	0.055811
6	0.421436
7	3.01267
204	2.59045
205	2.51181

DLIS:

n	Time (s)
5	0.028572
6	0.146372
7	0.931863
204	9.46762
205	9.8673

CDF:

n	Time (s)
5	0.045672
6	0.285943
7	2.24635
204	9.39065
205	9.54054

MOM:

n	Time (s)
5	0.052012
6	0.363920
7	2.797236
204	599.207
205	null

JW:

n	Time (s)
5	0.026734
6	0.162191
7	0.954462
204	10.4183
205	10.6577

For the unsatisfiable cases ($n < 10$), times increase exponentially, with JW and DLIS being fastest. For satisfiable ($n \geq 10$), times are polynomial-like and similar across heuristics, though MOM timed out at n=205.

Growth Classification: For unsat part ($n < 10$), exponential-like; for sat part ($n \geq 10$), polynomial-like (times stabilize as n increases, since satisfiable and easy to find assignment).

4 Analysis

The pigeonhole results for P(n, n-1) show exponential-like growth for all methods, with times multiplying as n increases (e.g., Random from 0.0008s at n=5 to 186s at n=10). This confirms PHP's hardness, as the solver must explore a growing search tree to prove unsat.

For P(2n, n), similar exponential growth, e.g., DLIS from 0.000046s at n=2 to 340s at n=9, showing that doubling pigeons amplifies difficulty quickly.

For P(10, n), when unsat ($n < 10$), exponential-like (times rise sharply); when sat ($n \geq 10$), polynomial-like (times flat, 0.0002s, as easy to find assignment). This highlights that unsat cases are harder.

The solver worked correctly for all tests, proving the DPLL code is reliable. However, it could be improved by adding CDCL (Conflict-Driven Clause Learning, which analyzes conflicts to learn new clauses that prevent similar mistakes in future searches, thus pruning the search space more effectively) or rewriting it in a faster language like C++ instead of Python, which is slower for big calculations.

5 Conclusion

This project shows that the choice of heuristic makes a big difference for hard, unsatisfiable pigeonhole problems. DLIS and JW were much faster than Random, CDF, and MOM in the tested cases. The experiments successfully demonstrated how problem size affects solving time, with exponential growth for unsatisfiable cases and polynomial growth for satisfiable ones.

These results matter because SAT solvers are used in important tasks like AI planning and checking software. Better heuristics could save time in these areas. In the future, I could add CDCL (Conflict-Driven Clause Learning, which analyzes conflicts to learn new clauses that prevent similar mistakes in future searches, making the search more effective) to make the solver smarter, try bigger values of m, or combine heuristics to get even faster results.

6 Acknowledgments

I would like to thank Professor William Gasarch from the University of Maryland for his valuable help and guidance throughout this project.

References

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, USA, 1971.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.