SAT Solvers and The Pigeonhole Principle

Daniel Hoang
Mentored by William Gasarch

October 11, 2025

1 Introduction

Many situations in our world can be reduced down to a Boolean formula. Suppose that a meeting is being planned between five friends; it is plausible to have a pair of friends that will show up to the meeting only when both people in the pair are present. This constraint in the situation can be written as a conjunction of two Boolean variables. As more constraints are added, the whole situation can be represented as a Boolean satisfiability problem (SAT). SAT solvers solve the SAT problem; they are algorithms that determine if a given Boolean formula can be satisfied by some assignment of the Boolean variables within it. A formula is UNSAT if it cannot be satisfied, and SAT if it can be.

The SAT problem was the first problem proven to be NP-complete, meaning the SAT problem can be verified in polynomial time and any other NP problem can be reduced to the SAT problem [3]. So, if a way to efficiently solve the SAT problem is discovered, all other NP problems can also be efficiently solved. This motivates the investigation of SAT solvers and the problems SAT solvers struggle with.

One problem that SAT solvers struggle with is the Pigeonhole principle problem. Thus, conducting this research did not only involve coding SAT solvers, but also generator algorithms for Boolean formulas based on the Pigeonhole principle. The SAT solvers are based on different heuristics, and their performance will be compared by their runtime when solving the Pigeonhole

principle problems. Simultaneously, we can explore how runtime grows as the Pigeonhole principle problems scale larger in size.

2 Background

2.1 Boolean Logic

SAT solvers take boolean formulas as input; SAT solvers naturally rely on Boolean Logic. A Boolean formula refers to a logical statement made up of Boolean variables and Boolean operators. These operators include the conjunction *AND* ($\wedge$), disjunction *OR* ($\vee$), and negation *NOT* ($\neg$). Boolean variables are simply variables that can be assigned values TRUE or FALSE.

2.2 CNF Boolean Formulas

CNF means conjunctive normal form. A Boolean Formula in CNF refers to formulas structured such that it is a conjunction of clauses. A *clause* is a disjunction of literals. *Literals* can be either a Boolean variable or a Boolean variable negated. For example, the following CNF Boolean formula has 4 clauses:

$$(x_1) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_3) \tag{1}$$

Observe how the first clause has only one literal, it is a *unit clause*. Further, x_2 and x_4 are considered *pure literals* because they are literals that only show up as non-negated or only show up as negated. To be exact, x_2 is a *pos-pure literal* because it is a pure literal and a variable, while x_4 is a *neg-pure literal* because it is a pure literal and a negation of a variable.

2.3 2-SAT

When every clause in a SAT problem has exactly two literals, it is 2-SAT. Although SAT is NP-complete, 2-SAT can be solved efficiently using graph theory and Kosaraju's algorithm [2]. Any 2-SAT problem can be converted into a directed graph, and Kosaraju's algorithm traverses the graph to find strongly connected components within those graphs. Strongly connected components are subsets in a directed graph such that all vertices in the subset has a path to every other vertex in the same subset. One last traversal of the

graph will discover any contradictions in our strongly connected components. If there are contradictions, the 2-SAT problem is UNSAT, otherwise it is SAT.

2.4 Davis-Putnam-Logemman-Loveland Algorithm

The Davis-Putnam-Logemman-Loveland Algorithm (DPLL) is the refinement of the very first SAT solver algorithm [1]. DPLL is a backtracking algorithm that relies on "cleverly" picking variables—the "cleverness" depends on the heuristic of your design. DPLL first checks if there is already an assignment in the formula and simplifies. Any FALSE in a clause disappears, while any clauses with a TRUE disappears from the Boolean formula. DPLL then adds assignments such that any unit clauses and pure literals resolve to TRUE. Next, DPLL checks if any further simplification can be done or if any unit clauses or pure literals arose to redo the steps. If not, DPLL then uses a heuristic to pick a variable "cleverly." That chosen variable is then given an assignment and DPLL goes through recursion. If any original clause resolves to FALSE, DPLL backtracks and tries the other assignment of the previously chosen variable. If there is an assignment that works, DPLL outputs SAT, else UNSAT for no valid assignment.

2.5 Pigeonhole Principle

The Pigeonhole principle states that given n pigeons and m holes, where $n > m$, at least 1 hole must contain more than 1 pigeon when fitting pigeons into holes. It is possible for any Pigeonhole principle problem to be represented as a CNF Boolean formula. Naturally, a SAT solver can determine whether or not any Pigeonhole principle problem is UNSAT or SAT based on the number of given n pigeons and m holes. For example, the Pigeonhole principle problem with 3 pigeons ($n = 3$) and 2 holes ($m = 2$) can be written as the conjunction of the following clauses:

1. $x_{11} \vee x_{12}$
2. $x_{21} \vee x_{22}$
3. $x_{31} \vee x_{32}$
4. $\neg x_{11} \vee \neg x_{21}$
5. $\neg x_{11} \vee \neg x_{31}$

6. $\neg x_{21} \vee \neg x_{31}$
7. $\neg x_{12} \vee \neg x_{22}$
8. $\neg x_{12} \vee \neg x_{32}$
9. $\neg x_{22} \vee \neg x_{32}$

Notation 2.1 Variable x_{ij} represents putting the i-th pigeon into the j-th hole.

Note that clauses 1-3 are disjunctions of non-negated variables. These beginning clauses represent the statement that each pigeon can go into any hole of their choosing. Consider clause 1; pigeon 1 can go into either hole 1 or hole 2. On the other hand, clauses 4-9 are disjunctions of negated variables. These later clauses represent the constraint that no hole can contain more than one pigeon. Consider clause 4; either pigeon 1 is not in hole 1 or pigeon 2 is not in hole 1—this clause also includes that it's possible for neither pigeon 1 nor 2 to be in hole 1. Since there are more pigeons than holes, this SAT problem is UNSAT. In this paper, all Pigeonhole principle problems the SAT solvers are tested on will be generated such that they are UNSAT.

3 Methods

3.1 SAT Solver Implementation

For our purposes, we programmed SAT solvers based on the same DPLL implementation, but differing based on their heuristic of "cleverly" choosing a variable for assignment.

The DPLL implementation shared by the SAT solvers in this paper differs from the general algorithm described in the background because it also checks if the SAT problem has been reduced to 2-SAT. This 2-SAT checking is done immediately before searching for unit clauses and pure literals. So, once the SAT solvers reduce the SAT problem to 2-SAT, the efficient 2-SAT algorithm is executed.

There are three different SAT solvers, each implementing our DPLL algorithm with a different heuristic, that this paper explores:

1. **Maximum Occurrences (MO)** As a simple heuristic, each clause is

iterated over to count the number of occurrences of each variable. The variable with the greatest number of occurrences is selected.

2. **Maximum Occurrences in Minimum Sized Clauses (MOMS)** Rather than counting the number of occurrences throughout all clauses, the MOMS heuristic only counts variable occurrences that are in clauses of the minimum size [4]. Suppose there is a SAT problem with clauses of sizes 2, 3, and 4. For this SAT problem, the MOMS heuristic will only count occurrences in the clauses of size 2. Once again, the variable with greatest counted occurrences is selected.

3. **Jeroslow-Wang (JW)** The JW heuristic was developed by Robert G. Jeroslow and Jian Kang Wang. The JW still puts an emphasis on smaller clauses like MOMS, but it also counts occurrences in other clauses [4]. The emphasis is maintained because the JW heuristic assigns a weight to each clause based on the number of literals in that clause:

$$JW(x) = \sum_{x \in C_i} 2^{-|C_i|} \tag{2}$$

For a variable x, the JW heuristic calculates a value that is the sum of all the weights of each clause C_i that the variable x occurs in. As seen, the weight of a clause C_1 is $2^{-|C_i|}$ where $|C_i|$ is the length of clause C_1. So, as the length of a clause increases, its weight decreases. As such, the JW heuristic selects the variable x with the greatest $JW(x)$ value.

These heuristics focus on smaller clauses because smaller clauses have less literals in them to satisfy the clause. So, the assignment of the literals in smaller clauses should be focused.

3.2 Pigeonhole Formula Generators

Before our SAT solvers can execute, we need the Pigeonhole CNF formulas to test them against. Using the Pigeonhole CNF formula structure described in the background, generators for the following problems were programmed:

1. n pigeons, $n - 1$ holes
2. n pigeons, $n - 2$ holes
3. n pigeons, $n - 3$ holes

4. n pigeons, $n - 4$ holes
5. $2n$ pigeons, n holes
6. $10n$ pigeons, n holes

3.3 Testing Environment

Each SAT solver will be tested against each Pigeonhole problem, measuring the SAT solver's runtime on the Pigeonhole problem for a given value n. The tests will start with $n = 5$ and increment by 1 after each test, scaling up the size of the Pigeonhole problem. Additionally, tests between a SAT solver and a Pigeonhole problem that take longer than 900 seconds will be left incomplete and only values of n prior to the incompletion will be considered. 900 seconds is a timeout cutoff.

Furthermore, Python is the programming language the SAT solvers, generator functions, and data collectors were written in. Similarly, the data will be graphed with Python's matlibplot library. In order to examine the growth of Pigeonhole problems as n increases, an exponential curve will be fit to the data.

All testing was conducted on hardware with the following properties

- Windows 10 Operating System
- AMD Ryzen 5 7600X 6-Core Processor @ 4.70 GHz
- 16 GB RAM @ 4800 MHz

4 Results

4.1 n Pigeons, n-k Holes

For early values of n, the problems are too trivial to have substantial the differences between each heuristic. The SAT solvers always timed out after $n = k + 10$. The final runtime was never near the 900-second cutoff, runtime was never even over 500 seconds. For the final value of n that didn't timeout, it is clear that the JW SAT Solver always performed worst, while MOMS performed the best. Additionally, the runtime of each test grew exponentially as n increased. All tests had fit an exponential curve with $R^2 = 0.99$.

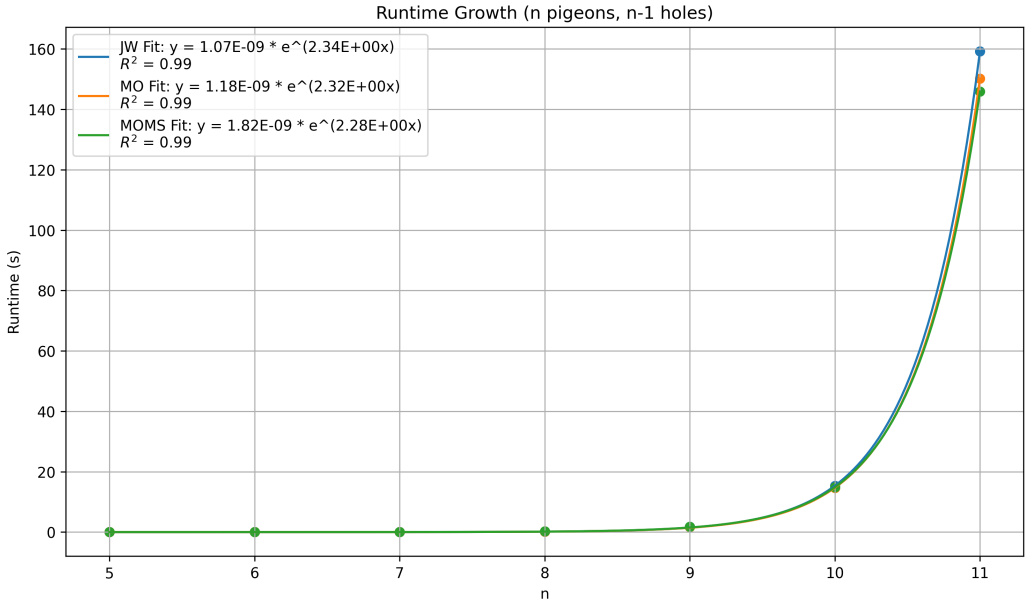

Figure 1: Runtime of SAT solvers on n pigeons, $n-1$ holes ($k=1$)

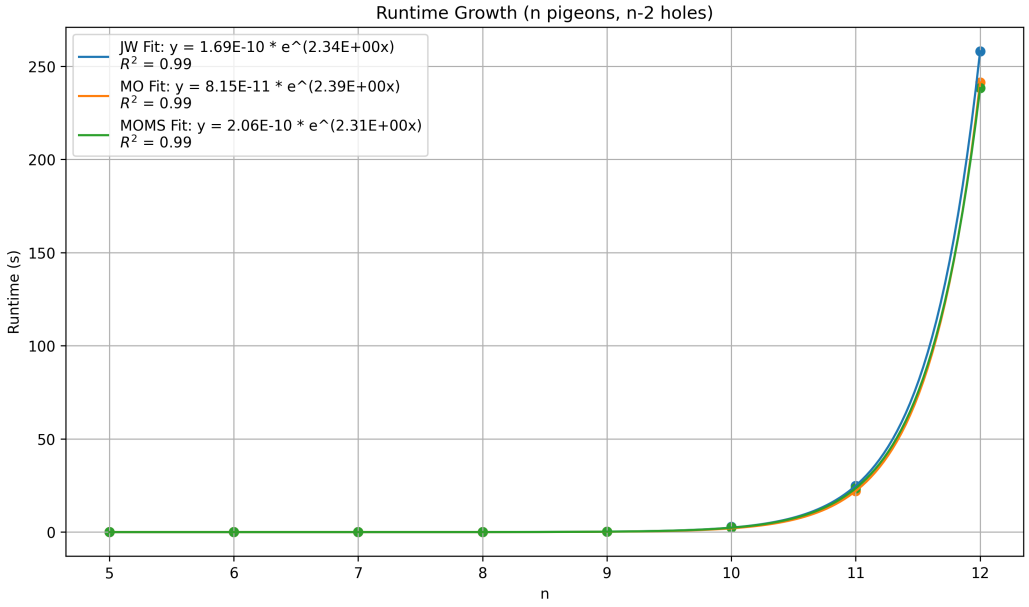

Figure 2: Runtime of SAT solvers on n pigeons, $n-2$ holes ($k=2$)

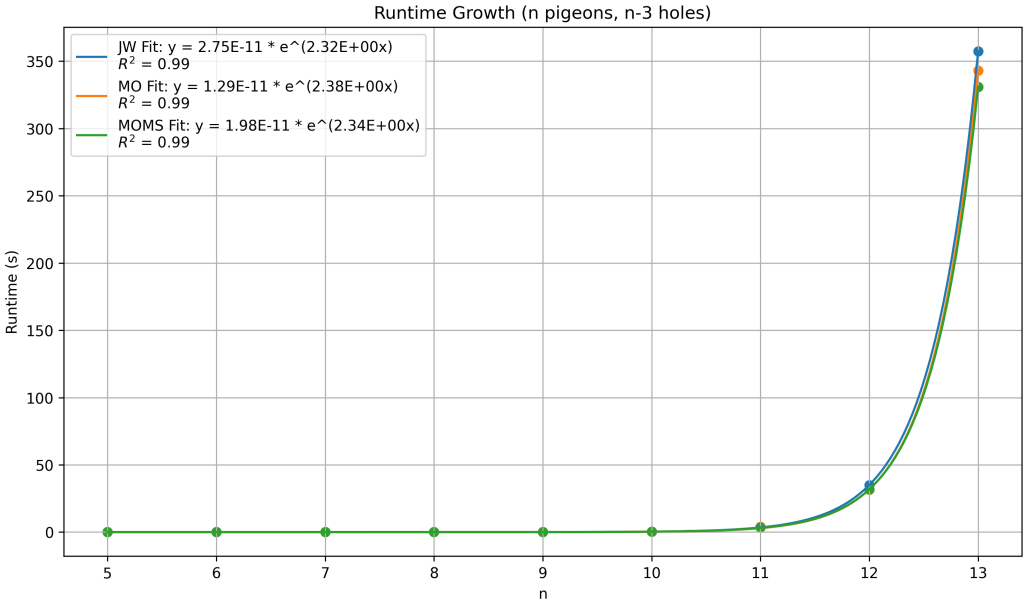

Figure 3: Runtime of SAT solvers on n pigeons, $n-3$ holes ($k=3$)

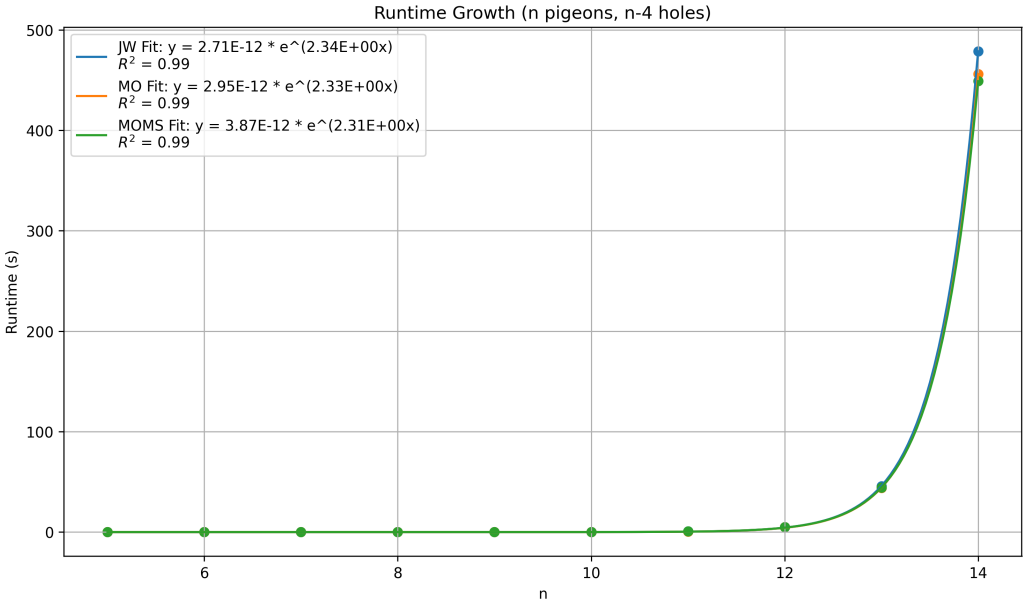

Figure 4: Runtime of SAT solvers on n pigeons, $n-4$ holes ($k=4$)

To further explore the Pigeonhole problems of this category, consider only the SAT solver runtime with the MOMS heuristic.

As k increases, the runtime of the SAT solver on for the same values of n decrease. For example, when $k = 2$, solving for $n = 12$ takes over 200 seconds. However, when $k = 4$, solving for $n = 12$ takes under 5 seconds. Naturally, it was possible for us to reach greater values of n before stopping due to our time constraint.

Interestingly, the graph visually suggests that a single linear line can pass through all the last data points for the different values of k. The same intuition appears for the second-to-last and third-to-last data points for the different values of k.

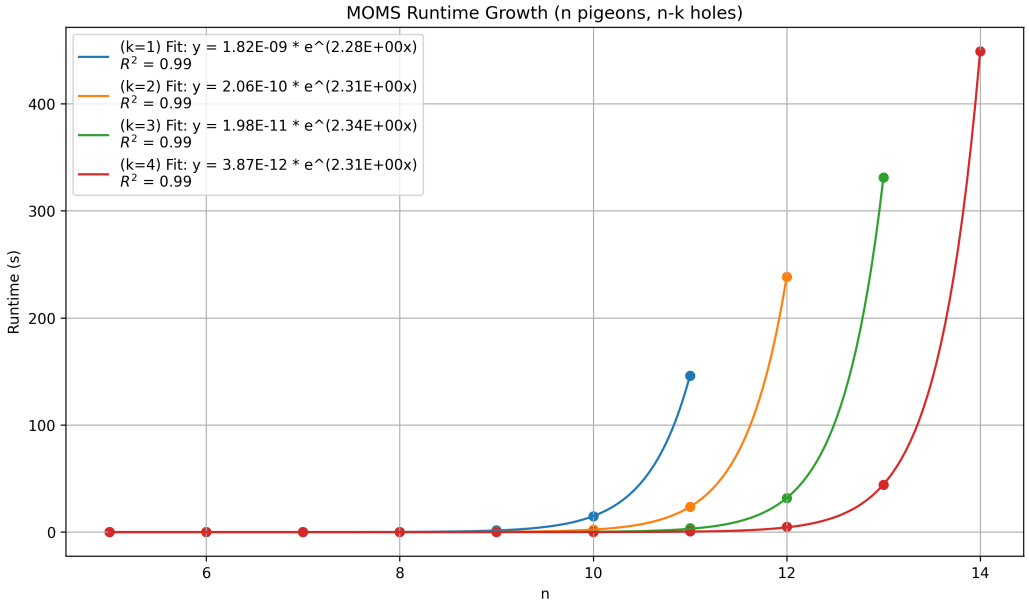

Figure 5: Runtime of SAT solver with MOMS heuristic on n pigeons, $n - k$ holes (k varying)

4.2 kn Pigeons, n Holes

Here, solving Pigeonhole problems with early values of n was once again too trivial to have great differences between the heuristics. This time, the SAT

solvers timed out after $n = 9$ for problems with $2n$ pigeons, while they timed out after $n = 8$ for problems with $10n$ pigeons. Final runtime got as high as 800 seconds. There was not a clear relationship describing the last value of n before timing out. For the runtime of later n values, the JW SAT solver was once again the worst-performing, but this time the MO SAT solver performed the best—not MOMS. Similar to the last Pigeonhole problems, the runtime growth also fit an exponential curve with $R^2 = 0.99$, regardless of heuristic.

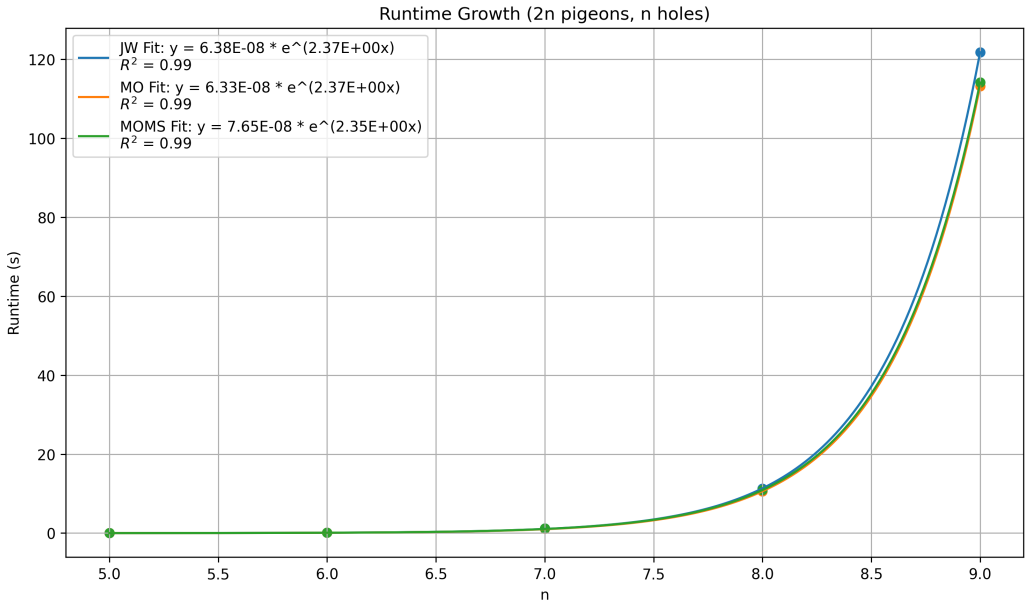

Figure 6: Runtime of SAT solvers on $2n$ pigeons, n holes

When considering only the MO SAT solver, it is evident that an increase in k once again decreases runtime for same values of n. For $n = 8$, there is a near 800 second difference between $k = 2$ and $k = 10$.

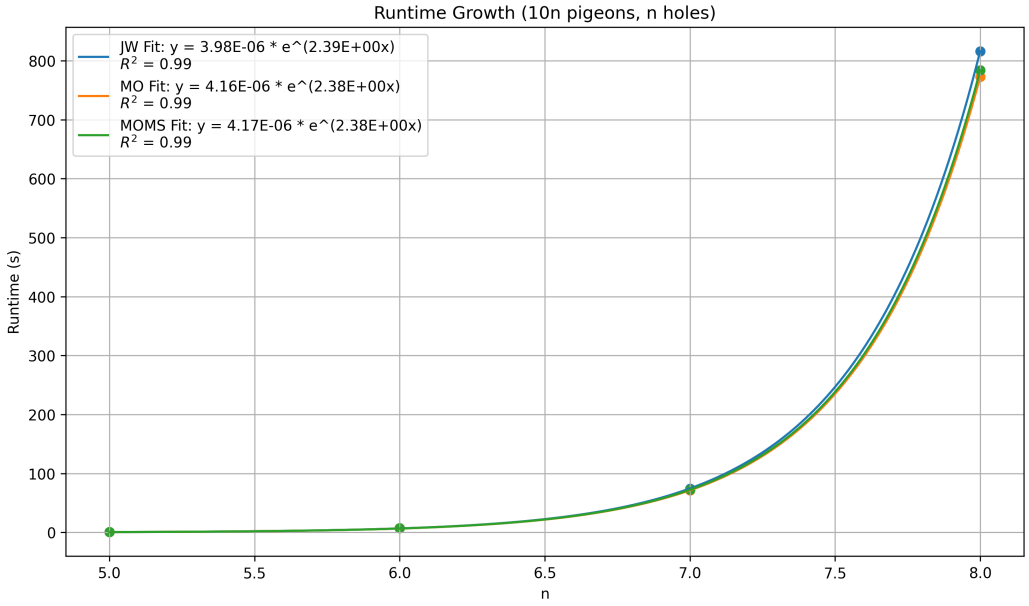

Figure 7: Runtime of SAT solvers on $10n$ pigeons, n holes

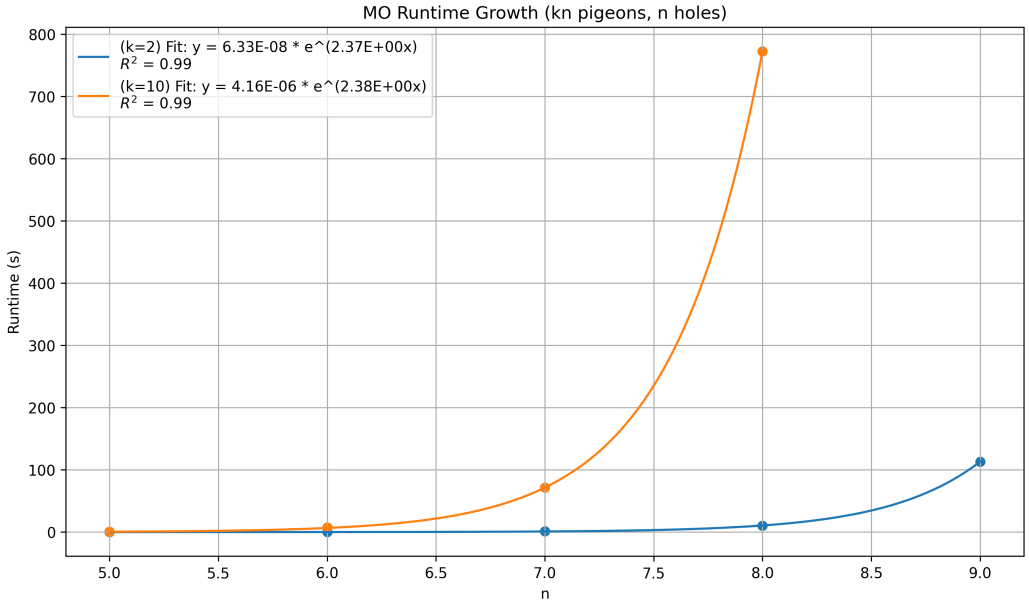

Figure 8: Runtime of SAT solver with MO heuristic on kn pigeons, n holes (k varying)

5 Conclusion

Although there were differences in runtime between the SAT solvers, we cannot confidently declare one as the "best." Since we were unable to test runtime on many values of n due to time constraints, it is possible our findings on SAT solver runtime do not extend for values of n greater than the ones investigated. Additionally, while MOMS performed the best for Pigeonhole problems of n pigeons and $n-k$ holes, MO performed the best for Pigeonhole problems of kn pigeons and n holes. Still, the JW SAT solver performed the worst in both categories of Pigeonhole problems, so it is likely the worst-performing of the three SAT solvers in general.

This paper has confirmed the exponential growth rate of runtime on solving Pigeonhole problems, as every problem was able to fit an exponential curve with little error. Unfortunately, the growth rate of runtime was too high to collect data for values of n beyond 14.

Further open-ended questions that arise from this work includes whether or not an increase in pigeons affects runtime more than a decrease in holes. Knowing that the value of n helps determine the number of clauses and variables, is it possible to have a Pigeonhole problem with more clauses or variables than another, but be solved with less time?

It is clear that the Pigeonhole principle is still one of SAT solver's "hard" problems to solve. Future investigation could involve SAT solvers that are not as naive and use adaptive heuristics. With more modern implementations, it is possible to investigate beyond this paper.

References

[1] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of sat, Nov 2023.

[2] Algorithms for Competitive Programming, Oct 2025.

[3] Lance Fortnow and Steven Homer. Computational complexity. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 495–521. North-Holland, 2014.

[4] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1(1–4):167–187, September 1990.