EECS 219B: Formal Methods — Assignment 1

Parker Ziegler

February 13, 2022

1 Horn-SAT and Renamable Horn-SAT

1.1 A Linear Time Algorithm for Deciding the Satisfiability of HornSAT Formulae

We base this algorithm off of the work of Dowling and Gallier [3].

1.1.1 Data Structures

Let us assume that we have a HornSAT formula, ϕ, in CNF. To begin, we initialize a collection of data structures to assist with tracking truth assignments to literals.

1. For each positive literal p in ϕ, we compute a list cl_p containing those clauses in which p appears as a negative literal. Thinking of Horn clasues as implications, this list represents the set of all clauses for which p appears on the left-hand side of an implication.
2. We compute an array nls such that $nls[c]$ returns the number of negative literals in clause c that have current truth value $false$ (0). A clause c is ready for processing by our algorithm if $nls[c] = 0$. In other words, all of its negative literals have a truth assignment $true$ (1), so its positive literal *is forced* to be $true$ (1).
3. We also create an array pls such that $pls[c]$ returns the positive literal in clause c, if one exists.
4. Last, we instantiate a queue q that will contain clauses that are ready to be processed. As stated in (2), these are those clauses for which $nls[c] = 0$.

1.1.2 Procedure

To begin, we initialize q to contain those clauses that are comprised of a single positive literal, since, by definition, $nls[c] = 0$ for these clauses. Thinking of Horn clasues as implications, these clauses are of the form $(true \Rightarrow x)$. We *know* that these positive literals must be assigned the value $true$ (1) if ϕ is satisfiable. We then enter a **while** loop guarded by the condition that q is not empty, that is, that there are still some clauses for which $nls[c] \neq 0$.

Within the body of the **while** loop, we enter a **for** loop iterating from 0 to the length of the queue, which is initially equivalent to the number of positive unit clauses in ϕ. In the body of the **for** loop, we pop off the head of the queue and store its value in a variable c_1. We then access the positive literal p_1 in c_1 by indexing into pls and set its value to $true$. We also store p_1 in a variable named $next$.

We then use $next$ to index into ϕ and obtain the clause list cl_p related to p_1, which was just assigned the value $true$(1). Recall that cl_p represents the list of all clauses c' in which p_1 appears as a negative literal. We "remove" this literal from these clauses by setting $nls[c'] = nls[c'] - 1$, in essence decrementing the count of negative literals in c' that have current truth assignment $false$ (0).

We then check in a conditional if $nls[c'] = 0$, meaning that c' is now ready to be processed. If so, we use c' to index into pls and find its positive literal p'. If the positive literal is not yet assigned a truth value, then we set its current assignment to *true* (1) and add c' to q to be processed in a later iteration of the **while** loop. If the positive literal is assigned $false$ (0), then we've hit a clause for which all negative literals have current truth value *true* (1) and its positive literal is $false$ (0). Therefore, we return UNSAT. The **while** loop continues to process entries in the queue until it is empty and, if this point is reached, we return SAT. See **Algorithm 1** below for the full algorithm.

Algorithm 1 Linear Time HornSAT Satisfiability

```
1:  procedure LinearHorn(φ)
2:      for all p in φ do                                   ▷ p is a positive literal.
3:          Initialize:
4:          p.cl ← list[c s.t. p is a negative literal in c]   ▷ We encode cl as a pointer field on p.
5:      end for
6:      Initialize:
7:      nls ← array[0..number of clauses in φ] of 0..max num of literals
8:      pls ← array[0..number of clauses in φ] of 0..max num of literals
9:      q ← queue of clauses comprised of a single positive literal
10:
11:     while q.length ≠ 0 do
12:         for 0 to q.length do
13:             c_1 ← q.pop()
14:             pls[c_1].val ← true; next ← pls[c_1]
15:
16:             for all c' in φ[next].cl do          ▷ Iterate over all clauses in which next is a negative literal.
17:                 nls[c'] ← nls[c'] − 1                ▷ Decrement the count of negative literals set to 0.
18:
19:                 if nls[c'] = 0 then                  ▷ A clause, c, is ready to be processed if nls[c] = 0.
20:                     p' ← pls[c'].val
21:
22:                     if p' ≠ false then
23:                         Set p' to true
24:                         q.push(c')                       ▷ Add c' to q for a subsequent loop iteration.
25:                     else
26:                         return UNSAT
27:                     end if
28:                 end if
29:             end for
30:         end for
31:     end while
32:
33:     return SAT
34: end procedure
```

1.1.3 Algorithm Correctness

The key to the algorithm's correctness lies in the fact that each clause in ϕ enters the queue *at most* once. Entrance into the queue is guarded by the condition that all negative literals have current truth value *true* (1) while the single positive literal is unassigned. This *forces* the *true* assignment to the positive literal, which subsequently prevents future re-entrance into the queue on a later iteration of the **while** loop. Popping a clause off of the queue causes all clauses in the clause list of its positive literal (*next*) to be evaluated. The procedure then effectively deletes all occurrences of *next* from clauses in which it appears as a negative literal (L17). These occurrences are, by definition, disjoint. Therefore, the number of times the *while* loop

executes is proportional to the number of negative occurrences of literals in ϕ, which is less than or equal to the total number of variables in ϕ.

1.1.4 Algorithm Runtime

The algorithm has a worst-case complexity of $O(n)$. Initialization of the arrays nls, pls, and the clause lists cl_p (L2-9) can all be completed in linear time with respect to the number of variables n. As described above, the complexity of the body of the *while* loop is also linear in n because each clause is entered at most once into the queue. Assuming every variable appeared as a positive literal in at least one clause (worst case), this would require evaluating one clause per positive literal. Therefore, the overall runtime of this algorithm is $O(n)$.

1.2 A Polynomial Time Algorithm for Deciding Whether a CNF Formula is Renamable Horn

We base this algorithm off of the proof given by Lewis [4] that provides a mechanism for determining if a formula S is renamable Horn. We start by describing this proof before giving the algorithm.

1.2.1 Preliminaries

Given a CNF formula S, our goal is to determine whether there is some way in which to complement the literals in S to produce a Horn formula.

Let us represent S as a set of clauses $\{C_1, ..., C_m\}$, with each clause C_i comprised of a set of literals $\{L_{i1}, ..., L_{il}\}$. A *renaming* of S entails replacing all literals in S whose variables also appear in a separate set $A = \{n_1, ..., n_n\}$ with their complements. We represent this renaming as $r_A(S)$.

Lewis describes how to determine the satisfiability of this set of clauses S. He states that S is satisfiable if and only if there is a *model*, M, for S. A model is a set of literals that:

1. Does not contain any complementary literals (e.g. $L \in M \vee L \notin M$)
2. Has a non-empty intersection with each clause in S (e.g. $\forall C \in S. M \cap C \neq \emptyset$)

Lewis begins his theorem by constructing a set of clauses $S*$ from literals in the clauses of S as follows:

$$S* = \bigcup_{i=1}^{m} \bigcup_{1 \leq j < k \leq l_i} \{\{L_{ij}, L_{ik}\}\}$$

His theorem claims that our original formula S is renamable Horn if-and-only-if S* is satisfiable.

1.2.2 Case analysis on the structure of an arbitrary clause $\{L_{ij}, L_{ik}\} \in S*$

The proof is by cases on the values of an arbitrary clauses $\{L_{ij}, L_{ik}\} \in S*$. First, assume that S is renamable Horn and let A contain the set of variables such that $r_A(S)$ is Horn. Also assume that we have a set M consisting of the variables in A and the complements of all variables that appear in S but not in A.

$$M = A \cup \{\textit{complement of } n \mid n \in S \wedge n \notin A\}$$

We also claim that M is a model of $S*$. The case analysis begins by picking an arbitrary clause $\{L_{ij}, L_{ik}\} \in S*$.

L_{ij} and L_{ik} are both positive

If L_{ij} and L_{ik} are both positive literals then $r_A(S)$ must complement one of them, since both appear in the same clause of S and $r_A(S)$ can only have at most one positive literal. Thus, $L_{ij} \in A \iff L_{ik} \notin A$ and similarly $L_{ik} \in A \iff L_{ij} \notin A$.

L_{ij} and L_{ik} are both negative

If L_{ij} and L_{ik} are both negative literals, then one of them must be left uncomplemented in $r_A(S)$, since $r_A(S)$ would have two positive literals otherwise. This can be expressed as $\neg L_{ij} \in A \iff \neg L_{ik} \notin A$ and similarly $\neg L_{ik} \in A \iff \neg L_{ij} \notin A$. We can reformulate this as $L_{ij} \in A \iff L_{ik} \notin A$ and $L_{ik} \in A \iff L_{ij} \notin A$, as above.

L_{ij} and L_{ik} have opposite signs

Let's say L_{ij} has positive polairty and L_{ik} has negative polarity. Then $r_A(S)$ would have to complement L_{ij} whenever it complemented L_{ik} (and vice versa) to maintain a maximum of one positive literal at a time. So $L_{ij} \in A \iff L_{ik} \notin A$ (and vice versa). Expanding the full conclusion, we arrive at $L_{ij} \in A \iff L_{ik} \notin A$ and $L_{ik} \in A \iff L_{ij} \notin A$, as above.

1.2.3 Concluding the Proof

Now assume that $S*$ is satisfiable. Then $r_A(S)$ (the A renaming of S) must be Horn by our theorem. Let M be a model for $S*$ and A be the set of positive literals in M. If we pick an arbitrary clause $C_i \in S$ and $r_A(\{C_i\})$ had two positive literals L_{ij} and L_{ik}, then neither L_{ij} nor L_{ik} could be in M; however $\{L_{ij}, L_{ik}\} \in S*$. By contradiction, we eliminate the possibility of a clause in $r_A(S)$ with more than one positive literal.

1.2.4 Procedure

Lewis' proof is quite useful in that it reduces our algorithm to constructing the set $S*$ and determining its satisfiability. If we can construct $S*$ in polynomial time, we can apply Aspvall, Plass, and Tarjan's algorithm [1] to determine its satisfiability in linear time.This is because $S*$ is composed solely of 2-literal clauses; determining its satisfiability is a 2-SAT problem. Below, we show the construction of $S*$ in **Algorithm 2**.

Algorithm 2 Linear Time HornSAT Satisfiability

```
procedure ConstructS*(φ)
    S* ← ∅

    for all C in φ do
        for all L in C do
            for L' in range(L + 1, C.length) do
                S* ← S* ∪ {{L, L'}}                ▷ Add {L_ij, L_ik} to S*.
            end for
        end for
    end for
end procedure
```

1.2.5 Algorithm Correctness

The correctness of the algorithm is bolstered by the proof by Lewis, which reduces the problem of determining if S is renamable Horn to constructing $S*$ and determining if it is satisfiable. Constructing $S*$ involves:

1. Iterating through all clauses C in ϕ.
2. For each clause, we iterate through all of its literals L.
3. For each literal L, we pair it with a succeeding literal L' in C where the index of L' in C is greater than the index of L.
4. We add $\{L, L'\}$, which represents $\{L_{ij}, L_{ik}\}$ in Lewis' proof, to $S*$.

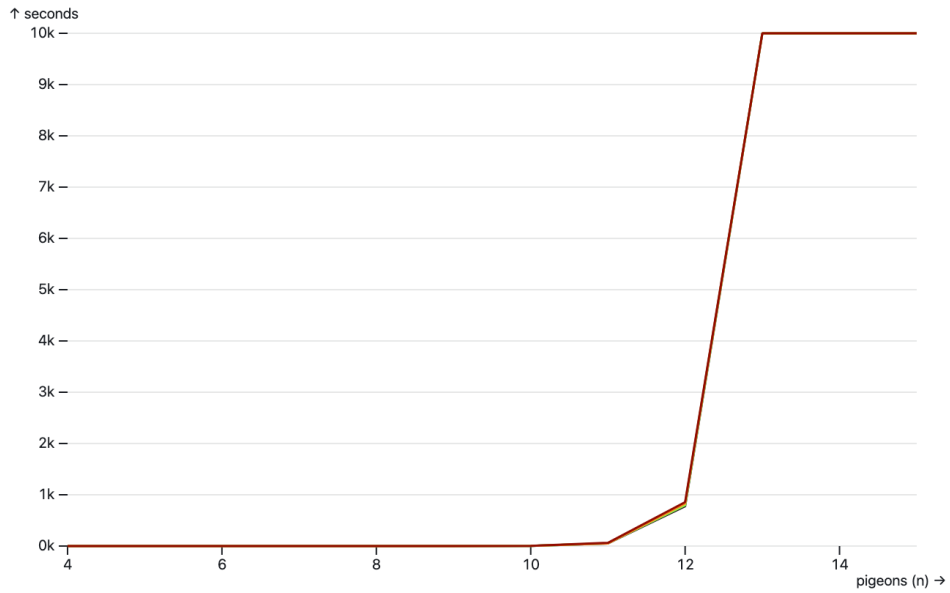

Figure 1: **The relationship between the number of variables n in the encoding of the Pigeon-hole SAT problem and the total solver time across 5 runs for all values $n = 4, 5, 6, ..., 15$. Solver runs were capped to 10,000 seconds.**

Determining the satisfiability of $S*$ is a 2-SAT problem. Aspvall, Plass, and Tarjan provide a linear-time algorithm for deciding 2-SAT satisfiability [1]. Additionally, a simpler polynomial time algorithm based on resolution, in which we combine clauses with an arbitrary variable n and its complement $\neg n$ into successively fewer implied clauses, would also work for determining the satisfiability of $S*$ due to its 2-SAT nature.

1.2.6 Algorithm Complexity

The algorithm has a worst-case complexity of $O(mn^2)$. Constructing $S*$ can take up to $O(mn^2)$ while determining the satisfiability of $S*$ can be accomplished in $O(n)$ (using Aspvall, Plass, and Tarjan's algorithm) and, in the worst case, $O(n^2)$ (using resolution).

2 The Pigeon-hole Problem

2.1 (a) Using SAT solvers

The source code for my encoding of the Pigeon-hole SAT problem is located in the `hw1/dimacs/src/main.rs` file in the zipped attachment and is implemented in Rust. The zipped attachment also contains the output `.cnf` files generated by this code. These files are named `pigeonhole-<npigeons>p-<nholes>h.cnf`. I opted to use the CaDiCaL SAT solver [2] to solve the generated CNF formulas. I conducted 5 runs of the solver across all `.cnf` files with a 10,000 second timeout. The results of these 5 runs are shown in Figure 1.

As we can see from the running times, we appear to hit an inflection point at $n = 12$ where solving times skyrocket and hit our maximum running time of 10,000 seconds. To assess the time complexity of running the SAT solver, we plot n against $log(seconds)$. The results appear in Figure 2.

We can see that the relationship between n and $log(seconds)$ is linear, suggesting that solving the Pigeon-hole problem takes the solver exponential time. If solving took polynomial time we would expect the relationship between n and $log(seconds)$ to appear logarithmic rather than linear. This suggests that solving the Pigeon-hole problem is an especially difficult problem for SAT solvers.

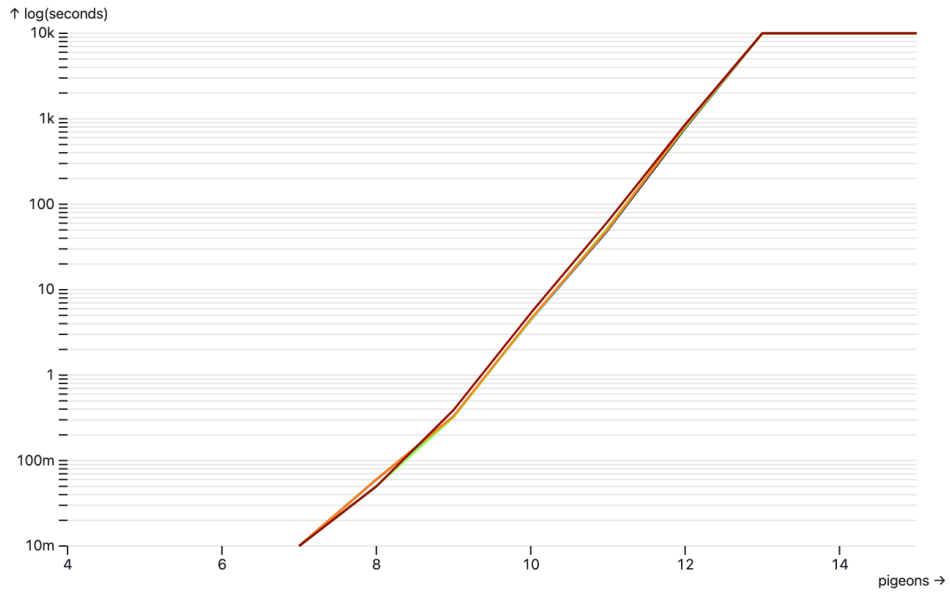

Figure 2: **The relationship between the number of variables n in the encoding of the Pigeon-hole SAT problem and $log(seconds)$ across 5 runs for all values $n = 4, 5, 6, ..., 15$. Solver runs were capped to 10,000 seconds.**

2.2 (b) BDDs

The source code for my encoding of the Pigeon-hole SAT problem as a BDD is located in `hw1/bdds/hw1.py`.

We see a similar relationship between n and the running time of our BDD encoding of the Pigeon-hole SAT problem as we did with our DIMACS encoding; that is, determining satisfiability requires exponential time. Figure 3 shows the relationship between n and the number of seconds to compute the BDD. One interesting difference between the two encodings is the significant difference in solving time. For example, for $n = 12$ the BDD encoding took just 88 seconds to return UNSAT ($false$), while the CaDiCaL SAT solver took 775 seconds. When we plot n againt $log(seconds)$, we notice the same linear relationship as was found when plotting n against $log(seconds)$ for the DIMACS encoding. This shows that constructing the BDD still requires exponential time for the Pigeon-hole SAT problem. Figure 4 shows this relationship.

Applying dynamic reordering from the `dd` package by calling `bdd.configure(reordering=True)` did not appear to result in a difference in the runtime of the algorithm. While execution times increased somewhat comparted to invoking the algorithm without dynamic reordering, the pattern of exponential runtime remained the same. Figures 5 and 6 plot the results.

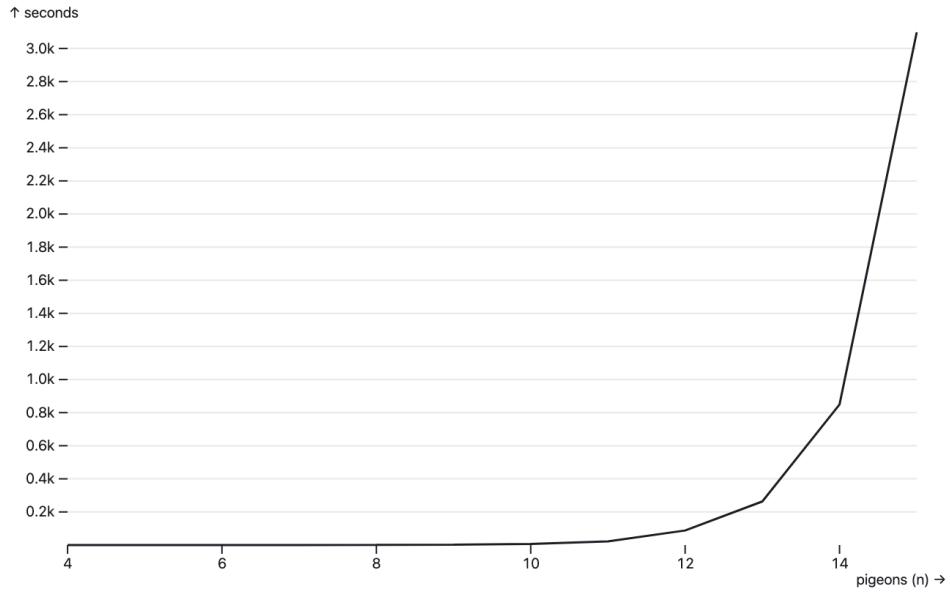

Figure 3: **The relationship between the number of variables n in the BDD encoding of the Pigeon-hole SAT problem and the number of seconds required to compute the BDD for all values $n = 4, 5, 6, ..., 15$.**

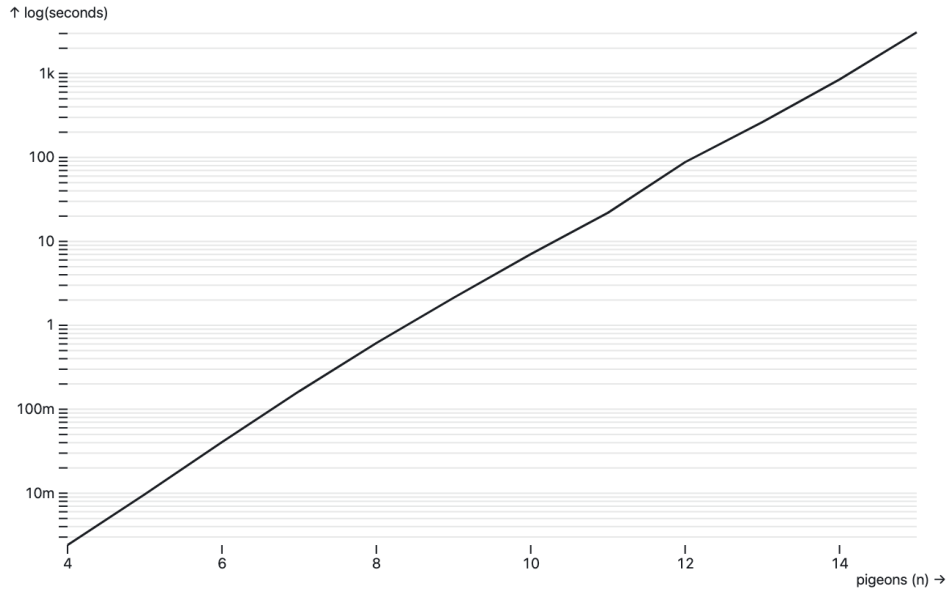

Figure 4: **The relationship between the number of variables n in the BDD encoding of the Pigeon-hole SAT problem and $log(seconds)$ required to compute the BDD for all values $n = 4, 5, 6, ..., 15$.**

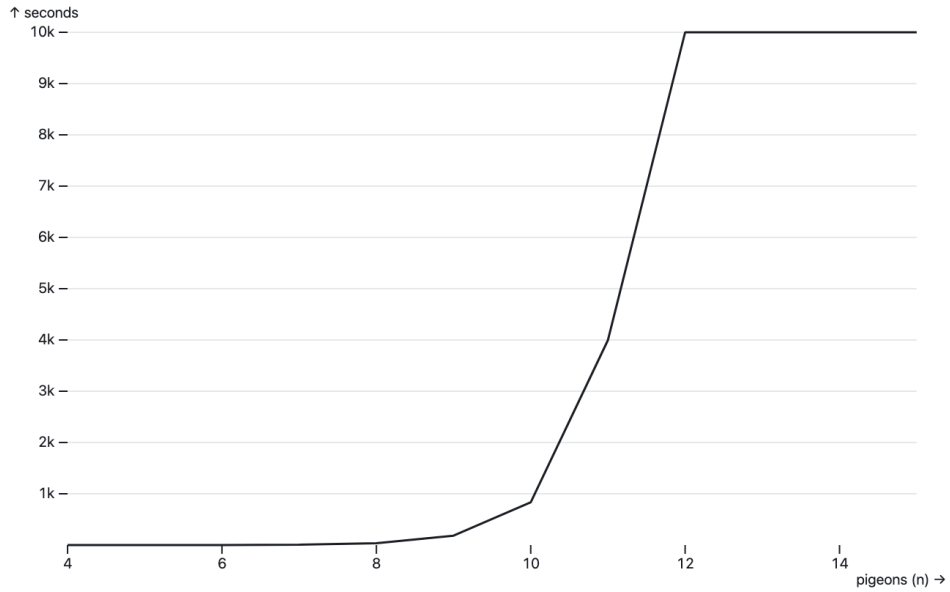

Figure 5: **The relationship between the number of variables n in the BDD encoding of the Pigeon-hole SAT problem and the number of seconds required to compute the BDD for all values $n = 4, 5, 6, ..., 15$ with dynamic reordering enabled. Similar to previous results we enforced a 10,000 second timeout.**

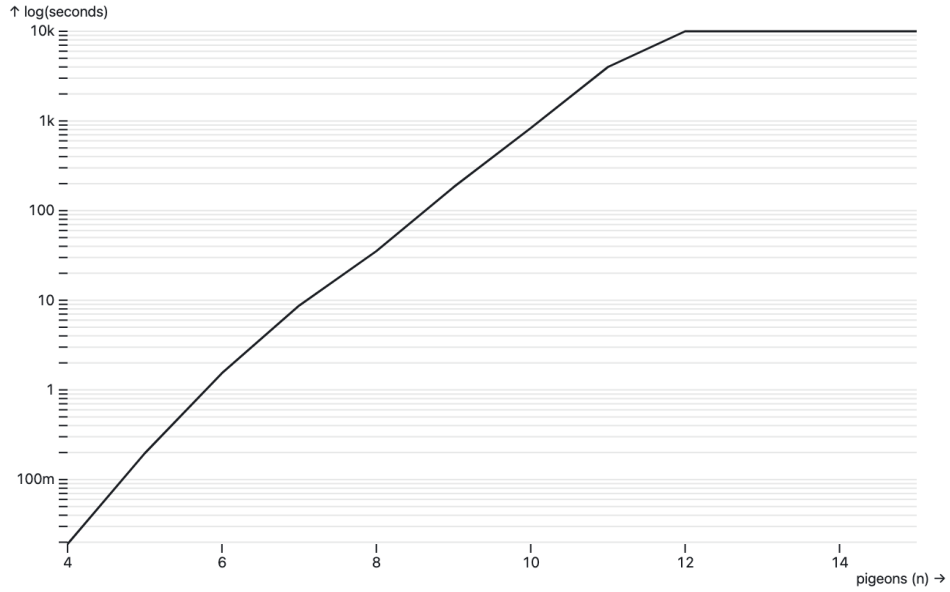

Figure 6: **The relationship between the number of variables n in the BDD encoding of the Pigeon-hole SAT problem and $log(seconds)$ required to compute the BDD for all values $n = 4, 5, 6, ..., 15$ with dynamic reordering enabled. Similar to previous results we enforced a 10,000 second timeout.**

References

[1] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979.

[2] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[3] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[4] Harry R. Lewis. Renaming a set of clauses as a horn set. *J. ACM*, 25(1):134–135, jan 1978.