# Searching for circuit complexity lower bounds using SAT solvers

May 8, 2022

**Abstract**

Where the abstract will go

## Contents

# 1 Introduction

## 1.1 Motivation

Finding lower bounds on the size of Boolean circuits computing given Boolean functions is one of the most fundamental and most challenging problems in computer science. Despite the fact that the majority of functions require exponential circuits, the best lower bounds proven on unrestricted circuits are only linear [**?**] and we still do not have optimal circuits or close upper and lower bounds for many important functions. Results in this area can be used to separate computational complexity classes, possibly even P and NP (i.e. to show that a large class of fundamental and practically useful problems do not have polynomially fast algorithms). [**?**]

If functions with circuits of a given complexity could be identified by an efficient algorithm, this could be used to construct efficient learning algorithms for circuits of a smaller complexity; this would resolve one of the biggest questions in learning theory. [**?**] Furthermore, such an algorithm could be used to break pseudorandom generators (i.e. distinguish random from pseudorandom inputs). [**?**] As a result, this problem has strong connections in both learning theory and cryptography.

A heuristic approach is to find minimal circuits for small instances of the functions. This can be used to prove new bounds [**?**] and may lead to theoretical insights on the structure of their optimal circuits generally [**?**]. Finding efficient circuits is also necessary in practice when designing electronic systems.

One approach to automating the search for efficient circuits is to reduce the problem of designing a correct circuit (logical design synthesis) to the Boolean satisfiability problem (SAT). Existing algorithms (SAT solvers) can then be used to solve the problem, either finding a correct circuit of fixed size or generating a proof that none exists [**?**]. SAT solvers have become increasingly powerful in recent years *cite* and now have many industrial applications*cite*; the wide range of heuristics implemented by different SAT solvers means that one suited to the problem at hand could be found or developed. *cite*

*how to extend it*

## 1.2 Previous work

Circuits are a popular model of non-uniform computation. In the past there have been theoretical proofs of lower bounds for restricted circuit classes, e.g. AC0[**?**][**?**], AC0[k] (i.e. AC0 additionally with MOD-k gates)[**?**][**?**], and monotone circuits[**?**]. However, no lower bounds better than 5n-c[**?**] have been found for general circuits. Furthermore, there are results stating that it is impossible to use circuit lower bounds to separate P and NP by relativising methods[?], algebrizing methods[?], or (under standard crpytpgraphic assumptions) so-called 'natural proofs'[?]. It has been proved that circuit lower bounds are hard to prove using resolution and resolution with k-DNFS, and it is conjectured that they are also hard under stronger proof systems such as Frege systems[?]. Due to this difficulty in theoretical progress, we would like to investigate the problem empirically.

Kamath et al [?] propose a reduction from logical design synthesis to SAT which fixes the DNF structure of the circuit and number of disjunctions used. Experiments using this reduction to find minimal circuits for 2-4 bit adders and multipliers are reported in [?]. Kojevnikov et al [?] demonstrate a more general reduction allowing any circuit with a fixed number of gates, proving linear bounds for $MOD_3^n$ circuits over different Boolean bases. This was done by checking for circuit 'building blocks' with 5 to 11 gates. In [?] Kulikov et al. used the same reduction to improve large circuit building blocks by searching for smaller versions of their sub-circuits. Specifically, linear upper bounds were proved or verified for $SUM$, $MOD_4^n$, and $MOD_3^n$ by checking for circuits with up to 13 gates (they remarked that checking for existence of a Boolean circuit of size 13 was out of reach). Knuth also used SAT solvers to investigate lower bounds for $MOD_3^n$ [?].

## 1.3 Our contributions

We investigate feasible ways to find small circuits using SAT reductions. Previous research in this area has focused on a small number of target Boolean functions and a single reduction at a time. Instead, we use multiple combinations of reductions and SAT solvers to see which leads to the best performance. We test them on a range of symmetric functions whose lower bounds are already somewhat understood, in order to analyse the performance in relation to the bound.

We also search for a general relationship between solver performance and minimum circuit size by applying our methods to a sample of randomly generated functions. We then use solvers to prove a result, that a lower bound conjecture about MOD-3 functions due to Knuth[?] does not hold for the de Morgan basis.

From these experiments we note that available solvers running on a personal computer can decide the existence of some circuits of size 11 in just a few hours, while others of size 11 and 12 can take many hours on a high-powered computing cluster (13 and above were not investigated). That is to say, the amount of time taken to solve comparable problems can vary considerably.

Exploring an additional possibility for improving performance, we find that adding extension axioms to a problem can improve the performance of a SAT solver on it, even when they are chosen at random. However, this effect varies between different target functions, even when searching for circuits of the same size.

## 2 Background

### 2.1 General setting

Denote by $B_{n,m}$ the set of all Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$ and let $B_n = B_{n,1}$. A circuit over the basis $A \subseteq B_2$ is a directed acyclic graph with nodes of in-degree 0 or 2. Nodes of in-degree 0 are called inputs. Nodes of in-degree 2 are assigned functions from $A$ and are called gates. Some nodes may be also be marked as outputs.

Without loss of generality, we can also assert that every gate has a larger index than both of its predecessors (i.e. the gates are sorted topologically w.r.t. the used numbering); that every gate's second predecessor has a larger index than its first predecessor; and that the last gate is an output. [**?**]

A circuit $c$ of size N is said to compute the function $f \in B_{n,m}$ if, for all input vectors $v \in \{0,1\}^n$ in the range of $f$, $c(v) = f(v)$, where $c(v) = (o_1, ...o_m)$ are the values of the output gates of $c$, each input $x_i$ takes value $v_i$, and each gate takes the value of its assigned function when applied to the value of its two predecessor nodes.

In this paper, we mainly consider circuits over the De Morgan basis

$$C_2 = \{\vee, \wedge, \neg\}$$

where $\neg$ denotes the Boolean function which outputs the negation of its first argument.

The size of a circuit is its number of gates. We define the circuit existence problem as the question of whether, given a truth table for a function $f : \{0,1\}^n \to \{0,1\}^m$, there exists a circuit of size N computing $f$.

### 2.2 SAT solvers

SAT solvers have previously seen success in proving mathematical theorems, most famously in 2016 when Heule et al solved the Boolean Pythagorean Triples problem[**?**] and in 2020 when Brakensiek, Heule et al solved Keller's conjecture[**?**]. In general, SAT solvers have the advantage of being effective even when heuristics are not known for solving the original problem, as in the case of circuit existence.

We used three SAT solvers, all based on conflict-driven clause learning. This is an algorithm closely related to DPLL, and which also makes use of the resolution proof system.[**?**][**?**] As mentioned earlier, this means that they cannot decide general circuit lower bounds with a less than exponential-length proof.[**?**] However, this asymptotic bound still allows for reasonably-sized proofs for small inputs such as the ones investigated in this paper.

The general idea of CDCL is to choose an assignment to a variable, propagate it (simplify the problem by removing clauses that now evaluate to true), and repeat until either finding a satisfying assignment, or finding that our assignment makes the formula unsatisfiable: in which case, add a 'learned clause' to the formula according to a procedure incorporating resolution, and then 'backtrack' (remove recent assignments so the formula is no longer unsatisfiable - if this is impossible, the answer is UNSAT) before trying again.

MiniSAT[?] was the winner of the 2005 SAT Competition and was last updated to version 2.2 in 2010[?]. Despite being old, it is well-documented and the other solvers we used were based on it, making it a good baseline for performance. It was also one of the solvers used to decide circuit existence in [?].

PicoSAT is a solver inspired by miniSAT 1.14, with a focus on improving low-level performance by using optimised data structures[?]. It was submitted in the 2010 SAT-Race, and was also one of the solvers used in [?] and the main one in [?].

MapleSAT is the most recent solver, with variants having won the SAT Competition 2016 and placed second in 2017. It was based off MiniSAT 2.2 but has a different branching heuristic, incorporating ideas from machine learning[?].

Finally, in order to make full use of the Oxford ARC service and its distributed computing, the parallel solver Glucose-Syrup was used. This is a 2014 parallel version of the Glucose solver, which placed 4th in both the 2014 and 2015 SAT Competition. Glucose is a solver also based on MiniSAT 2.2, which uses a new heuristic to more accurately estimate the 'usefulness' of its learned clauses and regularly delete ones that are not as useful [?]. Glucose-Syrup is an unusual 'non-portfolio' parallel solver which runs multiple instances of the same solver in parallel, but rather than taking a divide and conquer approach, uses new heuristics for sharing learned clauses between the instances in order to speed them up [?].

## 2.3 Extension axioms

A relatively unexplored method for speeding up SAT solvers is adding extension axioms to the initial formula by the following method:

1. Let $l_1$, $l_2$ be existing literals chosen from the formula and let q be a fresh variable (the extension variable).

2. Add the clauses
$$\neg l_1, q, \neg l_2, q, l_1, l_2, \neg q$$
These are known as extension axioms.

3. Repeat arbitrarily many times with new choices for $l_1$, $l_2$ and new fresh variable $q$. Note that the previously added q may also be chosen as a literal.

These axioms allow the SAT solver to use 'extended resolution' - resolving on a formula incorporating multiple literals, rather than on a single literal. This is possible since $q \leftrightarrow l_1 l_2$ and $l_1$, $l_2$ may also be extension variables, allowing for larger formulae to be built. [?]

## 2.4 Reduction to SAT

We encoded the circuit existence problem as a Boolean formula using three reductions. Let $t$ be the target function with $n$ inputs, and let $N$ be the number of gates required in the circuit.

### 2.4.1 Kojevnikov reduction

The first is due to Kojevnikov et al [**?**] who present it together with bounds on the growth rate of the formula. In particular, the growth rate of the number of clauses is $O(N \cdot 2^n)$. The basis of the circuit can be specified as any subset of $B_2$.

It uses the following variables:

1. $t_{ib_1b_2}$ $(n \le i < n+N, 0 \le b_1 < 2, 0 \le b_2 < 2)$ is the value of the $i$-th gate if its first predecessor takes value $b_1$ and its second takes value $b_2$. The four variables $t_{i00}, t_{i01}, t_{i10}, t_{i11}$ thus define the function assigned to the $i$-th gate. This gives $O(N)$ variables.

2. $c_{ikj}$ $(n \le i < n+N-1, 0 \le k < 2, 0 \le j < n+N)$ is true iff $j$ is the $k$-th predecessor of $i$. This gives $O(N^2)$ variables.

3. $o_{ij}$ $(n \le i < n+N, 0 \le j < m)$ is true iff the $j$-th output is computed by the $i$-th gate. This gives $O(Nm)$ variables.

4. $v_{it}$ $(0 \le i \le n+N, 0 \le t < 2^n)$ is the output value of the $i$-th gate if the input variables take values represented by the bits of $t$. These are used to constrain correctness of the circuit on all possible inputs, including its outputs matching the truth table of the encoded function.

We encode the requirements for the circuit by the following clauses:

1. The binary function assigned to each gate belongs to our desired basis.

2. For all $(i, k)$, exactly one variable $c_{ikj}$ is true (there is exactly one $k$-th predecessor of the $i$-th gate). This gives $O(N^3)$ 2-clauses and $O(N)$ $O(N)$-clauses.

3. For all $j$, exactly one variable $o_{ij}$ is true (the $j$-th output is computed by exactly one gate). This gives $O(N^2m)$ 2-clauses and $O(m)$ $O(N)$-clauses.

4. For all $0 \le i < n$ and $0 \le t < 2^n$, $v_{it}$ is equal to the corresponding bit in $t$. This gives $O(n \cdot 2^n)$ 1-clauses.

5. For all $n \le i < n+N$ and $0 \le t < 2^n$, $v_{it}$ is equal to the value computed by the $i$-th gate. This constrains all the gates with predecessors. Clauses of this type are written for all $n \le i < n+N$, $n \le j_0 < i$, $j_0 \le j_1 < i$, $0 \le i_0 < 2$, $0 \le i_1 < 2$, $0 \le r < 2^n$, and look as follows:

$$\neg c_{i0j_0} \vee \neg c_{i1j_1} \vee \neg(v_{j_0r} = i_0) \vee \neg(v_{j_1r} = i_1) \vee (v_{ir} = t_{ii_0i_1})$$

This gives $O(N^3 \cdot 2^n)$ 6-clauses.

6. The outputs of the circuit match the truth table. Clauses of this type are written for all $0 \leq k < m, 0 \leq r < 2^n, n \leq i < n+N$, and look as follows:

$$\neg o_{ik} \vee (v_{ir} = value_{kr})$$

where $value_{kr}$ is the required value of the $k$-th output when the circuit is given input $r$, according to the truth table. This gives $O(N2^n m)$ clauses.

### 2.4.2 Razborov reduction

The second is due to Razborov [**?**], assumes the function outputs only a single Boolean value, and requires a circuit over the basis $C_2$. Note that it has a lower growth rate (in terms of number of clauses) than the other two reductions - $O(N \cdot 2^n)$ as opposed to $O(N^3 \cdot 2^n)$ - suggesting it encodes the requirements more efficiently.

It uses the following variables:

1. $y_{av}$ ($a \in 0, 1^n, v \in [N]$) is the value taken by node $v$ on circuit input $a$. This gives $O(N \cdot 2^n)$ variables.

2. $y_{a\nu v}$ ($a \in 0, 1^n, \nu \in 1, 2, v \in [N]$) is the value of the $\nu$-th predecessor to $v$ on input $a$. This gives $O(N \cdot 2^n)$ variables.

3. $Fanin(v)$ is 0 if $v$ is a $\neg$-gate and 1 otherwise. This gives $O(N)$ variables.

4. $Types(v)$ - when $Fanin(v) = 1$, this is 0 if $v$ is a $\wedge$-gate and 1 if $v$ is a $\vee$-gate. This gives $O(N)$ variables.

5. $InputType_\nu(v)$ is 0 if the $\nu$-th predecessor to $v$ is a constant or an input, 1 if it is another gate. This gives $O(N)$ variables.

6. $InputType'_\nu(v)$ - when $InputType_\nu(v) = 0$, this is 0 if the $\nu$-th predecessor to $v$ is a constant, 1 if it is an input. This gives $O(N)$ variables.

7. $InputType''_\nu(v)$ - when $InputType_\nu(v) = InputType'_\nu(v) = 0$, this is the value of the constant $\nu$-th predecessor to $v$. This gives $O(N)$ variables.

8. $InputVar_\nu(v, i)$ ($i \in [n]$) - when $InputType_\nu(v) = 0, InputType'_\nu(v) = 1$, this is 1 iff the $\nu$-th predecessor to $v$ is the $i$-th input. This gives $O(N \cdot n)$ variables.

9. $INPUTVAR_\nu(v, i)$ is equal to $\bigvee_{i' < i} InputVar_\nu(v, i')$, which will be used to bound the fan-in below. This gives $O(N \cdot n)$ variables.

10. $InputNode_\nu(v, v')$ ($v' < v$) - when $InputType_\nu(v) = 1$, this is 1 iff $\nu$-th predecessor to $v$ is the gate $v'$. This gives $O(N^2)$ variables.

11. $INPUTNODE_\nu(v, v')$ is analogous to $INPUTVAR_\nu(v, i)$. This gives $O(N^2)$ variables.

We encode the requirements for the circuit by the following expressions:

1. Constant predecessors take the correct values, giving $O(N)$ clauses:
   $\neg InputType_\nu(v) \wedge \neg InputType'_\nu(v) \to (y_{a\nu v} = InputType''_\nu(v))$

2. Gates have at most one input per input predecessor, giving $O(N \cdot n^2)$ clauses:
   $\neg InputType_\nu(v) \wedge InputType'_\nu(v) \to \neg(InputVar_\nu(v, i) \wedge InputVar_\nu(v, i'))$ for $i \neq i'$

3. INPUTVAR variables take the correct value, giving $O(N \cdot n)$ clauses:
   $\neg InputType_\nu(v) \wedge InputType'_\nu(v) \to (INPUTVAR_\nu(v, i) = (InputType_\nu(v), i-1) \vee InputVar_\nu(v, i)))$, where $InputType_\nu(v, 0) = 0$

4. Gates have at least one input per input predecessor, giving $O(N)$ clauses:
   $\neg InputType_\nu(v) \wedge InputType'_\nu(v) \to INPUTVAR_\nu(v, n)$

5. Input predecessors take the correct values, giving $O(N \cdot n \cdot 2^n)$ clauses:
   $\neg InputType_\nu(v) \wedge InputType'_\nu(v) \wedge InputVar_\nu(v, i) \to (y_{a\nu v} = a_i)$

6. The analogous clauses to the above, but for InputNode and gate predecessors.

7. Gates take the correct value, giving $O(N \cdot 2^n)$ clauses.

8. The final gate (i.e. the output) matches the truth table, giving $O(2^n)$ clauses.

### 2.4.3 Naive reduction

The third reduction is straightforward. It also assumes the function outputs only a single Boolean value and requires a circuit over the basis $C_2$. The growth rate for the number of clauses is $O(N \cdot 2^n)$, the same as for the Kojevnikov reduction.

It uses the following variables:

1. $e_a^i$ $(n \leq i < N, a \in \{0,1\}^n)$ is the value of the $i$-th gate when the input to the circuit is $a$. This gives $O(N \cdot 2^n)$ variables.

2. $x_{i,j}$ $(0 \leq i < N + n, 0 \leq j < i)$ is true iff j is a predecessor of i. This gives $O(N \cdot (N + n))$ variables.

3. $g_i^0, g_i^1$ $(n \leq i < N)$ together encode the function assigned to the $i$-th gate by their truth values, giving $2N$ variables:

| $g_i^0$ | $g_i^1$ | Function |
|---|---|---|
| 0 | 0 | $\neg$ |
| 0 | 1 | $\vee$ |
| 1 | 0 | $\wedge$ |
| 1 | 1 | $\neg$ |

We encode the requirements for the circuit by the following clauses:

1. For all $(i, j, k)$ with $n \leq i < n + N$, $0 \leq j < i$ and $j < k < i$, and for all $a \in \{0, 1\}^n$, if $j$ and $k$ are both predecessors of $i$, then $e_a^i$ must take the value computed by the $i$th gate's assigned function on $e_a^j$ and $e_a^k$. Clauses must be added for each function in the basis. For example, the expression for when the $i$th gate is an $\wedge$-gate:

$$(g_i^0 \wedge \neg g_i^1) \wedge (x_{i,j} \wedge x_{i,k}) \rightarrow (e_a^i \leftrightarrow e_a^j \wedge e_a^k)$$

   Note that the gate must compute the conjunction of any two predecessors' variables. This means it can have at most two predecessors, otherwise the conjunctions of different pairs of predecessors may disagree and the gate will have no correct value. This gives $O(2^n \cdot N^3)$ 7-clauses.

2. For all $i$ with $n \leq i < n + N$, gate $i$ must have at least one predecessor. This gives $O(N)$ $O(N + n)$-clauses.

3. For all $(i, j)$ with $n \leq i < n + N$, $0 \leq j < i$, if $i$ is an $\vee$-gate or $\wedge$-gate with $j$ its predecessor, it must have another distinct predecessor (i.e. it must have at least 2 overall). For example, the expression for when the $i$th gate is an $\wedge$-gate:

$$(g_i^0 \wedge \neg g_i^1) \rightarrow \left( x_{i,j} \rightarrow \bigvee_{j<k<i} x_{i,k} \right)$$

   This gives $O(N^2)$ $O(N)$-clauses.

4. For all $i$ with $0 \leq i < n$, $a \in \{0, 1\}^n$, the $i$-th input must equal $a_i$. This gives $O(n \cdot 2^n)$ 1-clauses.

5. For all $a \in \{0, 1\}^n$, the final m gates (i.e. the outputs) should agree with the output values given for $a$ in the target truth table. This gives $O(2^n)$ 1-clauses.

## 3 Results

Timed experiments were carried out on a 3.400GHz Intel i3-8130U processor running Linux. The times shown were obtained by running the software on the problem 3 times and taking the median.

Further experiments were conducted on the Oxford ARC service *cite* using 1 node which could run up to 32 processes in parallel. *add more details about ARC pleas

## 3.1 Comparison of Boolean functions

We investigated the relationship between the size of a circuit and the time taken for a SAT solver to confirm or deny its existence. We encoded a range of simple symmetric functions using the Kulikov encoding:

1. AND on 7-bit and 8-bit inputs
2. Parity on 3-bit and 4-bit inputs (1 if the function is 1 mod 2, 0 otherwise)
3. $MOD_3$ on 3-bit and 4-bit inputs (0 if the function is 0 mod 3, 1 otherwise)
4. Majority on 3-bit and 4-bit (1 if the majority of input bits are set to true)

We chose to investigate the existence of circuits up to 9 gates, as these were reliably solved within less than 9 hours on the single-processor hardware described.

The size of the minimal circuit is shown in red*need to re-plot the graphs to make this work better*. It was found that the time needed to solve a problem tended to spike on or around the optimal circuit size. This may be because it is difficult to confirm or rule out circuits that come close to satisfying the requirements. Another possibility is that when close to the optimal size, there is the lowest proportion of satisfying circuits relative to the total amount of possible circuits, meaning that more of the search space must be explored (as opposed to quickly exploring it all at small sizes or quickly landing upon a satisfying circuit at large sizes). A similar phenomenon was reported in [**?**].

Different functions took different amounts of time overall, varying between seconds and hours, something also seen in Section **??**. Performance for each function was not especially consistent between the solvers; usually they found the same things difficult, but the length of time each solver took for the difficult problems varied, for example in **??** or. The extent of this can be seen in Section **??**.

A complete display of graphs can be found in the appendix.

## 3.2 Comparison of SAT solvers and reductions

We investigated whether there was a relationship between the reduction used to encode a Boolean function, the solver used, and the time taken by the solver. We took the total of the times each combination took to solve the problems from Section **??** and compared them as shown in **??**.

It can be seen that MapleSAT on the Kulikov encoding performs the best. Apart from this, the effect of the reduction and the solver seem to be independent, with MapleSAT consistently performing better than MiniSAT and

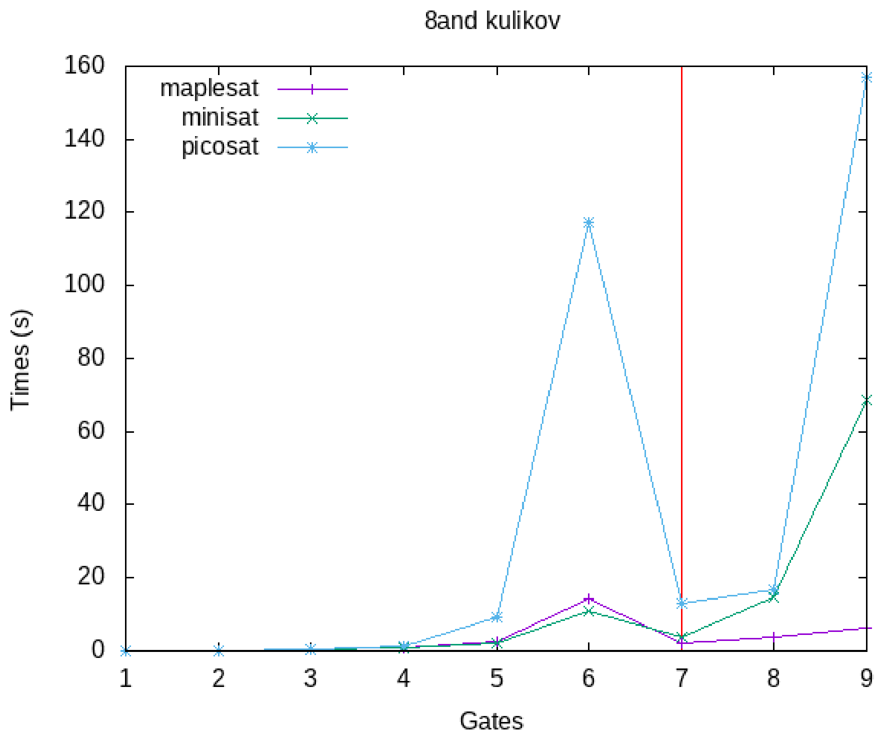

Figure 1: Times taken by SAT solvers to check for existence of a circuit computing AND on 8 bits

PicoSAT, and the Naive and Kojevnikov reductions performing better than the Razborov reduction. It is possible that the efficient encoding of the Razborov reduction is detrimental to the performance of the solver. However, these results are only applicable to small values of n.

## 3.3 Extension axioms

We investigated the effect of adding 50,000, 100,000, and 200,000 extension variables to the encodings for existence of 9-gate circuits for $MOD_3$ and 4-bit parity functions. The literals were chosen uniformly at random using a pseudorandom number generator simulating a uniform distribution (specifically, xorshift128+ due to the TypeScript implementation). Taking note of the results from the previous section, we used the Kojevnikov reduction and MapleSAT.

On the $MOD_3$ function, adding extension variables consistently increased the time for the solver, possibly due to a larger amount of clauses and variables needing to be dealt with at each step. However, for the parity function, adding

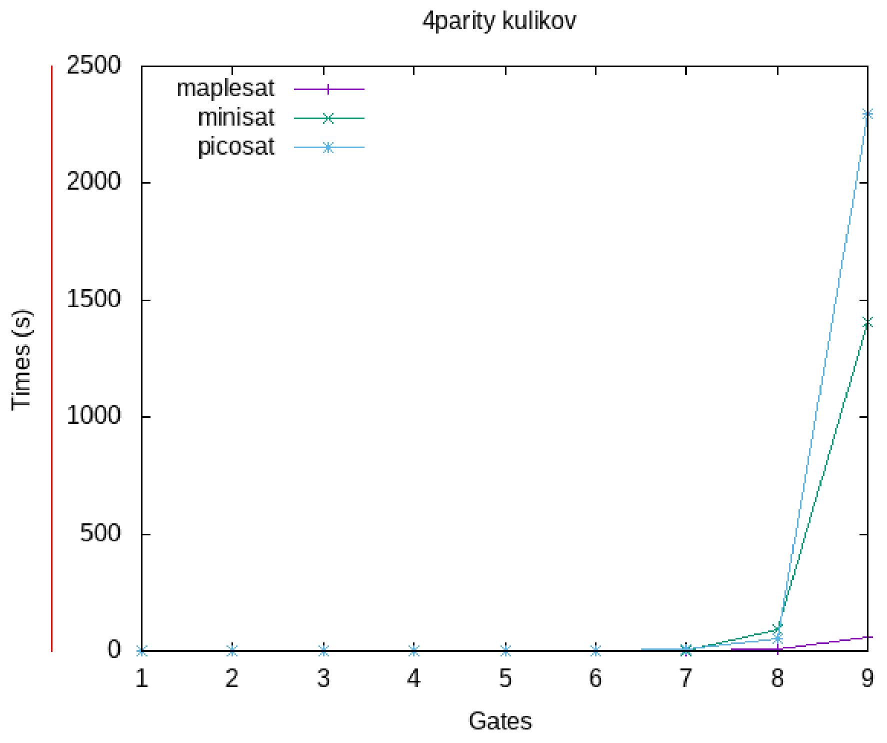

Figure 2: Times taken by SAT solvers to check for existence of a circuit computing parity on 4 bits

100,000 and 200,000 variables led to a speedup over 50 percent of the time. For both functions the median time was lowest when adding 100,000 variables, suggesting that this is a proportionate amount for the size of the problem (both problems had 401 variables and 37500 clauses, although a few of these were single-literal clauses that can be immediately simplified away by the solver).

## 3.4 Random functions

64 truth tables were generated for 4-bit inputs using the same pseudorandom generator as above. The existence of circuits of size 1 to 11 for these tables were encoded with the Kojevnikov reduction and solved using MapleSAT. The times taken to check for circuits are shown plotted against their optimal circuit size in **??** (note that 2 tables could not be fully solved within 9 hours; these functions have been omitted leading to a sample size of 62). The distribution of optimal circuit sizes is shown in **??**.

The scatter graph shows a clustering in times between functions of the same

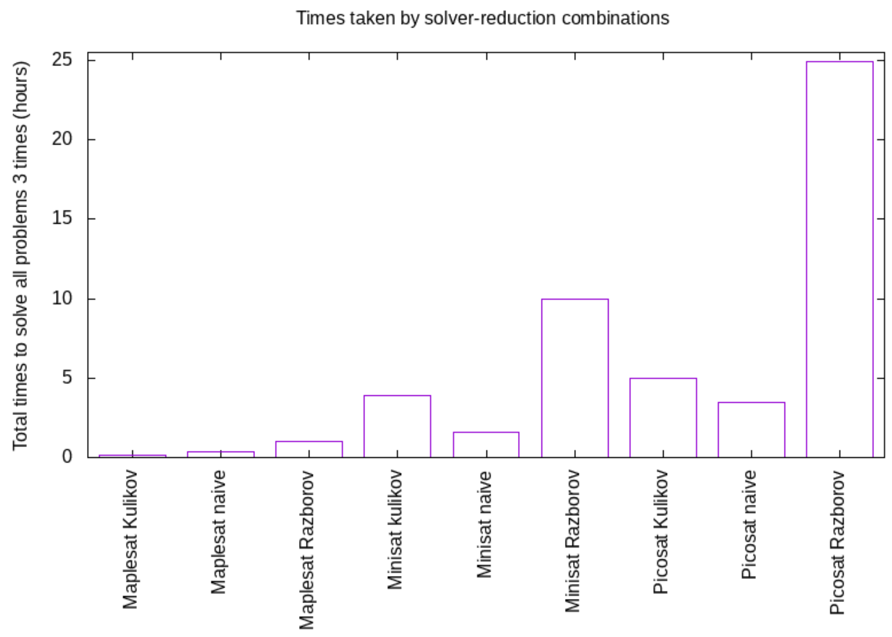

Figure 3: Comparison of times taken by SAT solvers on different reductions

optimal circuit size, but with a few outliers taking more time. This is most easily seen for sizes 10 and 11 (although the same is visible for 8 and 9 when plotted on a smaller scale). Also, the clustered functions of each optimal circuit size took strictly less time to solve than those with larger sizes, despite the fact that we always checked for existence up to 11 gates. This shows a generally strong relationship between the difficulty of checking circuit existence and the complexity of the function.

While we expect a large proportion of circuits to be hard (i.e. have a large optimal circuit size) for large input sizes [**?**], the usual counting arguments break down for such small values of n. However, using the SAT solver we can check the number of hard circuits directly. While the small sample size can account for the irregular shape of the graph in **??**, there does appear to be a peak around 10 gates. This contradicts the intuition that there should be exponentially more harder functions, and suggests that there is quite a low upper limit on the complexity of a function which only can only vary over 4 bits of input. However, this is all under the assumption that the pseudorandom generator used did output a truly random sample; in reality, it may have sampled disproportionately many easy (or hard) functions.

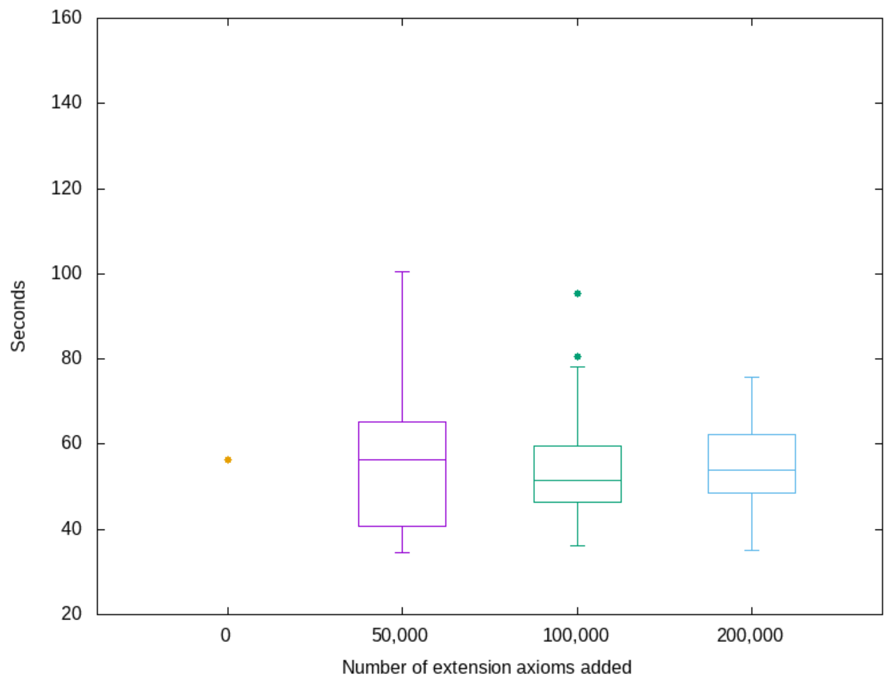

Figure 4: Comparison of times taken to check existence of a 9-gate circuit for 4-bit parity

## 3.5 Knuth's MOD-3 conjecture

Define $MOD_n^{m,r}(x_1 \dots x_n) = [x_1 + +x_n \equiv r \text{mod} m]$, where $[b]$ is the Iverson bracket, i.e. evaluates to 1 if $b$ evaluates to true, else false.

Based on the minimum circuit sizes he found for small sizes of n, Knuth conjectured that for all $r$, and $n \geq 3$, and over a full circuit basis

$$minsize(MOD_n^{m,r}) = 3n - 5 - [(n + r) \equiv 0 mod 3]$$

He was able to confirm this for $3 \leq n \leq 5$ and all $r$, and for the case $n = 6, r = 0$, but not for the case $n = 6, r = 1$.

We decided to try and check the hypothesis for $n = 6, r = 0$ and $n = 6, r = 1$ for the De Morgan basis. This involved checking for circuits for the function up to size 11 and 12, and 12 and 13 respectively. The Glucose-Syrup solver was used, running on the Oxford ARC service, which displayed gains in performance as the number of threads (so, parallel instances of the server working together) increased up to 32. *need to include some specs of ARC and citation*

For the De Morgan basis, it was shown that Knuth's conjecture does not hold for $n = 6, r = 0$; confirming that there is no circuit size 11 took only 17 minutes, but confirming there is none of size 12 took 72.97 hours.

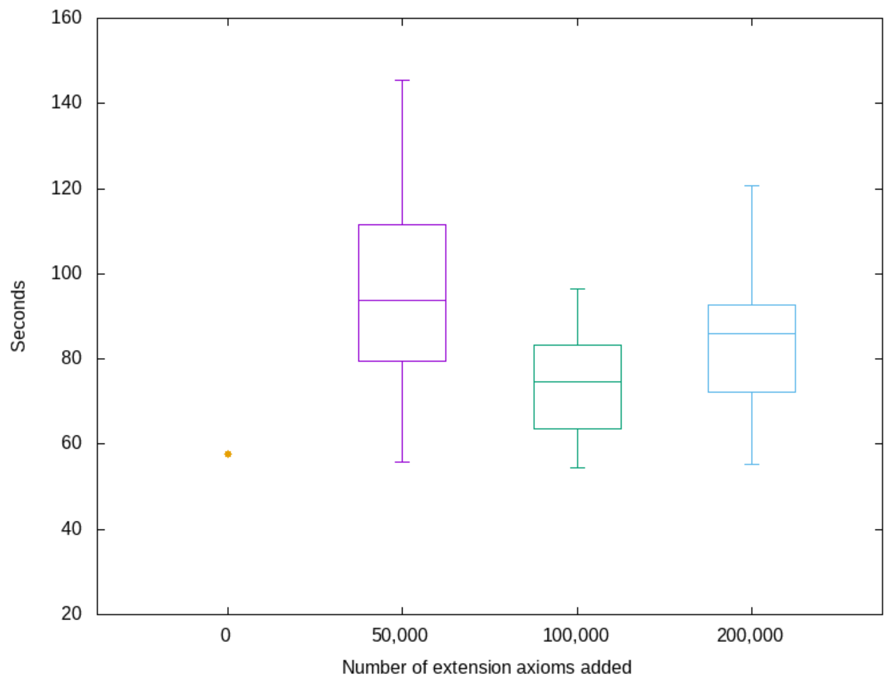

Figure 5: Comparison of times taken to check existence of a 9-gate circuit for $MOD_3^0$

For $n = 6, r = 0$, confirming that there is no circuit size 11 took 175.37 hours, while checking for size 12 did not terminate within the alloted 2 weeks of runtime.

*honestly, not sure how to draw conclusions from this or whether I can add a few more results*

## 4 Discussion

### 4.1 Reflection

The main practical lesson learned from this project was to think in the long term. Firstly, by writing code that can easily be adapted, i.e. following good OOP practices, so that less refactoring will be needed later, and by writing code that can scale to dealing with larger numbers as the scope and ambition of the project expands. This could be done by choosing a well-suited programming language, algorithms and data structures - I chose TypeScript and used arrays to store my CNF formulae, since these were familiar and easy for me to get started with, but the performance of my code on large circuits was an unexpected barrier later

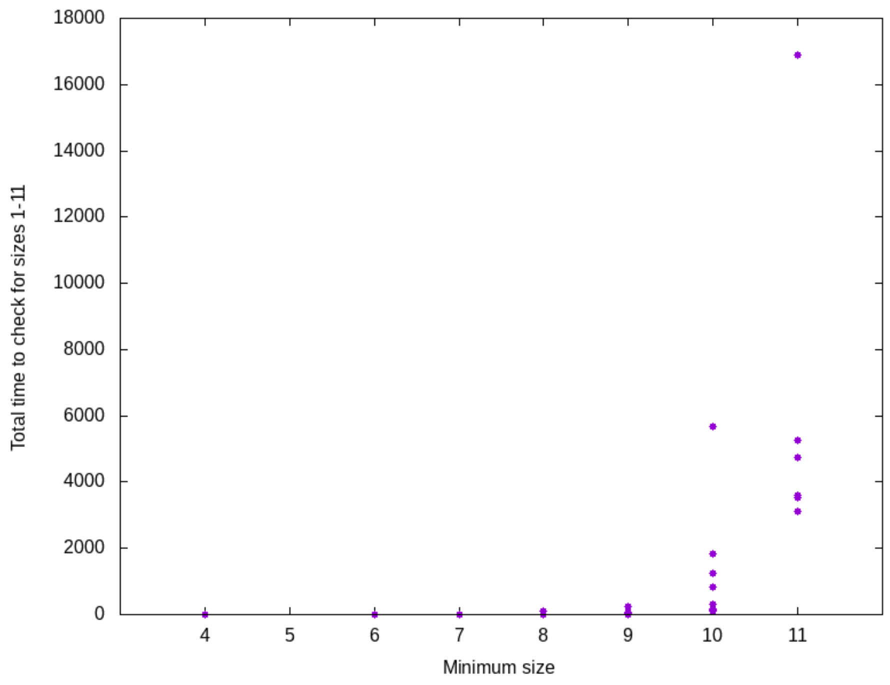

Figure 6: Time taken to check up to 11 gates for randomly chosen functions

on, as well as the poor documentation of the different pseudorandom generators in TypeScript. Also, I was often too focused on quickly getting results rather than investing time in creating a convenient interface for the experiments or processing the resulting data, which led to wasted time in the long run.

On the academic side, I have begun to learn how to read and extract the information I need from papers, as well as how to write in the academic style and include a sufficient amount of references and background for my own report. I discovered that doing academic research is like any creative creative process, involving exploration, creativity, backtracking and the need to unify the ideas of the project into a central narrative. Furthermore, clear communication and precision is important, something I found out when I wrongly assumed the definitions of some common Boolean functions. I practiced analysing data, looking for patterns, applying my theoretical knowledge and drawing conclusions about the things that I was investigating. I also learned some techniques for visualising data in order to highlight the patterns within it to the reader.

I started out by writing a program to reduce a single circuit to SAT using the Kojevnikov reduction, but quickly found it unwieldy and wrote a library of TypeScript objects to make it easy to read truth tables from files, store CNFs, quickly add certain types of clauses to them, and output them in the DIMACS

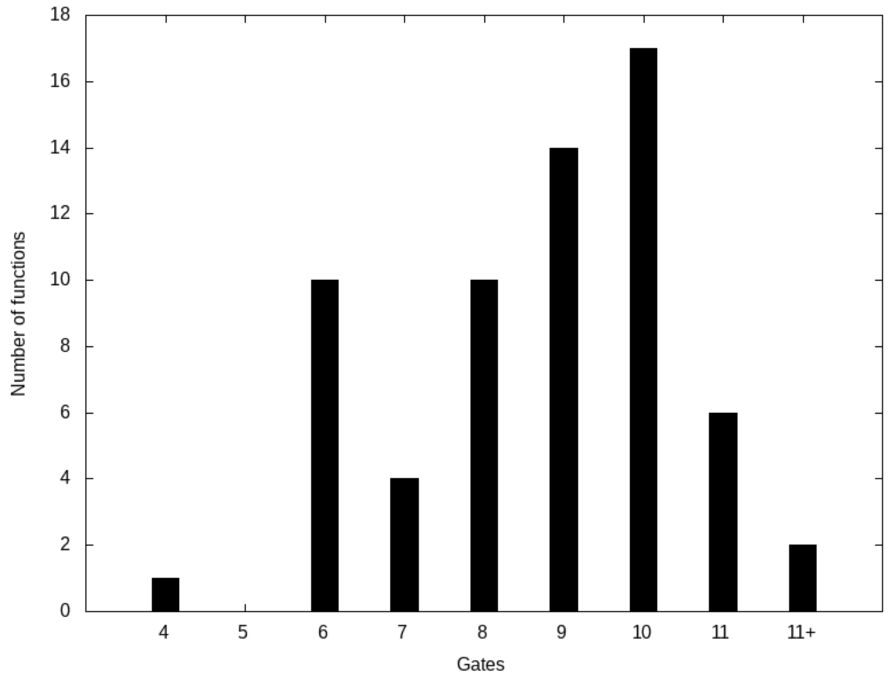

Figure 7: Sizes of optimal circuits for 4-bit functions with randomly chosen truth tables

format used by the SAT solvers.

Next I wrote reductions to various types of restricted circuits, such as monotonic, bounded-depth, and unbounded fan-in. However, since the problems for restricted circuits did not seem to be consistently harder or easier to solve in my preliminary experiments, I decided to focus on improving the performance for general circuits.

I read about the Razborov reduction and wrote the program for it, as well as for the naive reduction. I investigated the different reductions' performances with the three SAT solvers at this point, in order to select the fastest one for use in further experiments, and concluded that I should use the Kulikov reduction with the MapleSAT solver.

Realising that more precise measurements were needed for the experiments, I investigated better methods to record the runtime of programs on the command line, beginning with using the 'time' command but progressing to storing the local start and end time in shell variables. I also gained a greater proficiency in using stream editors and gnuplot on the command line in order to process the data from my experiments, and learned LaTeX in order to present the results and theory necessary for my report.

The rest of the project proceeded smoothly, conducting experiments for extension axioms and randomly generated functions, as well as learning how to use the Oxford ARC service with a parallel solver to investigate Knuth's conjecture.

## 4.2 Conclusions

-Choice of solver matters a lot (although this is quite obvious) -Choice of reduction also matters a lot -The time it takes to check for circuit existence is strongly related to its minimum size (bigger -¿ more time) -Same number of gates may be easier or harder for different boolean functions -Alternative search methods (for min gates) than linear search may be a good idea, since we are looking for a 'spike' not an ordinary upward curve/straight line to the min.size -we can get to 12 gates

## 4.3 Further directions

-Alternative search methods -trying some of the restricted circuits and bases from the early experimental stages -Search for the circuit size for all 4-bit functions (or bigger eg. maybe 9 bits will allow the bounding trick to apply) -searching for hard functions -more research in extension variables - the fact that extension axioms leading to improvements were found, even with no strategy or constraints on the axioms added, is promising for research in the topic. Even without a theoreticcal approach, you could just repeatedly generate bunch of ext vars, and do data analysis E.g. comparing performance to depth/size of circuits added, to whether we include literals for gate types or graph edges, doing different problem sizes and comparing performance to the proportion of variables added.. -Investigating why maplesat is best/comparing it to other recent-er solvers and seeing if any modifications can be made improving the solver for circuit problems in particular; same for the kojevnikov reduction, and thinking about how to modify it; also other reductions such as valiant circuits.