SAT solver performance on Circuit Satisfiability Problem

Mitchell Billard, Adebayo Emmanuel Adeoya

July.6, 2018

1 Introduction

The Circuit Satisfiability problem is defined as follows:

CSP (Circuit Satisfiability Problem).

The Circuit Satisfiability problem (also known as CIRCUIT-SAT, Circuit-SAT, CSAT, etc.) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output True/False. Circuit-SAT has been proven to be NP-Complete. [?]...

2 Reducing Circuit Satisfiability Problem to SAT

We reduce the Circuit Satisfiability Problem to SAT as follows:

Variables:
Circuit = $\{G | \exists$ some combination of $g_1, g_2, ...g_n, ..., g_m \in \{0, 1\}$ $\}$
The circuit has n inputs ($g_1, g_2, ...g_n$) and m gates ($g_{n+1}, ...g_m$) where g_m is the output gate. In our SAT solver we will introduce variables ($z_1, z_2, ...z_n$, $z_{n+1}, ...z_m$) which will encode values that circuit gates take.

Constraints:

1. If g_i is a NOT gate with input g_j, then add to U :
   $(\neg z_i \iff z_j) \equiv (\neg z_i \rightarrow z_j) \wedge (z_j \rightarrow \neg z_i)$
   $\equiv (\neg z_i \vee \neg z_j) \wedge (z_i \vee z_j)$

2. If g_i is an AND gate with input g_j and g_k, then add to U :
   $(z_i \iff z_j \wedge z_k)$

$\equiv (z_i \to z_j \wedge z_k) \wedge (z_j \wedge z_k \to z_i)$
$\equiv (\neg z_i \vee (z_j \wedge z_k)) \wedge (\neg(z_j \wedge z_k) \vee z_i)$
$\equiv (\neg z_i \vee z_j) \wedge (\neg z_i \vee z_k) \wedge (\neg z_j \vee \neg z_k \vee z_i)$

3. If g_i is an OR gate with input g_j and g_k, then add to U :
   $(z_i \iff z_j \vee z_k)$
   $\equiv (z_i \to z_j \vee z_k) \wedge (z_j \vee z_k \to z_i)$
   $\equiv (\neg z_i \vee (z_j \vee z_k)) \wedge (\neg(z_j \vee z_k) \vee z_i)$
   $\equiv (\neg z_i \vee z_j \vee z_k) \wedge [(\neg z_j \wedge \neg z_k) \vee z_i]$
   $\equiv (\neg z_i \vee z_j \vee z_k) \wedge (\neg z_j \vee z_i) \wedge (\neg z_k \vee z_i)$

4. To get the final CNF form of a circuit, 'AND' all of the encoded gate values together with z_m.
   $U = z_1 \wedge z_2 \wedge \cdots \wedge z_n \wedge \cdots \wedge z_m$

The SAT Solver will output satisfying assignments for all z_i. We only care about z_1 to z_n because we only need assignments for the input gates. We will assign z_1 ... z_n to g_1 ... g_n respectively.

3 Generating random instances of the Circuit Satisfiability problem

Our generator will created random "layered" boolean circuit. This means that gates in layer i will only be able to connect to gates from the previous layer $i-1$. The generator will take in 4 parameters:

1. n = the number of input gates
2. m = the number of gates in the circuit
3. $fanin$ = the maximum number of inputs a gate can take (at most it will equal the number of gates in the previous layer. The minimum fanin for a NOT gate = 1, and for both AND and OR gates is 2)
4. d = depth of the circuit. (Where this is a layered circuit, d also equals the number of layers in the circuit.)

The generator will take integer values for each of these parameters. We will have to limit the range of these choices so that we don't make excessively large circuits that will not be able to be solved in a reasonable amount of time. This range can be experimented with in the future.

To start, each layer will have the same amount of gates, calculated by dividing m by d (give or take 1 due to integer division). In the future we can compare how circuits of different orientations perform (i.e. 2x5 vs. 5x2).

Each gate in the layer will be chosen randomly from one of the 3 options (AND, OR, NOT), and will be given a random amount of inputs (min $fanin \leqslant$ amount of inputs $\leqslant$ generator $fanin$). We could experiment with giving each gate the same amount of inputs, and compare that with giving each gate a random amount of inputs. The inputs that each gate will take in will be randomly chosen from the outputs of the previous layer of gates.

The generator will have a gate class, so that when each gate is created its information can be stored. Each gate will have the following parameters:

1. $layer$ = The layer the gate is in
2. i = The number of that gate in its layer
3. *type* = Either AND, OR, or NOT
4. $fanin$ = The number of inputs the gate has
5. *inputs* = What the inputs to the gate are (i.e. outputs from other gates)
6. *output* = The output of the gate

The generator will output a list of these gate objects.

4 Experimental results

We used the Syrup Solver running on Ubuntu 16.04 LTS. The system has 8 Gb of ram, and has a quad-core Intel Core i5-4440 CPU @ 3.10GHz. The Syrup Solver is a SAT Solver for doing symbolic circuit analysis.It reads a circuit description written in DIMACS CNF notation, generates the nodal equation and returns the solutions. The Reducer.py program writes the DIMACS reduction of the circuit directly into the input.cnf file for the solver.

In our testing, we ran 2 batches of tests. The first one n went from 5 to 100, incrementing by 5. The second had n go from 5 to 1000, incrementing by 50. We ran both batches for 16 times each, varying the values of m and d. m set to 10, 100, 1000, and 10000. d was set to 3, 10, 100, and 1000. For each increment of n for a given m and d, 10 circuits were generated and solved. For example, for n = (5, 100, 5), m = 1000, and d = 10, 10 circuits were created and solved. The average time to solve the 10 instances at that n is what is plotted on our graphs. Table 1 shows the experiments ran. Some spaces are left blank because d had to be less than or equal to m.

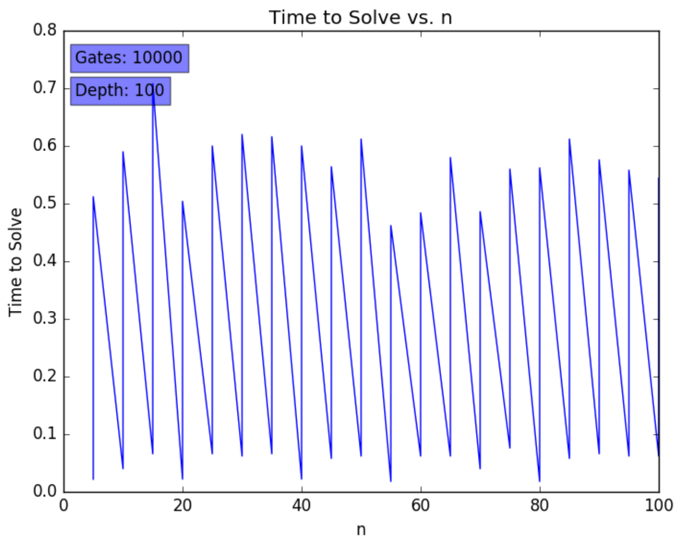

Figure 1: Gates: 10000, Depth: 100

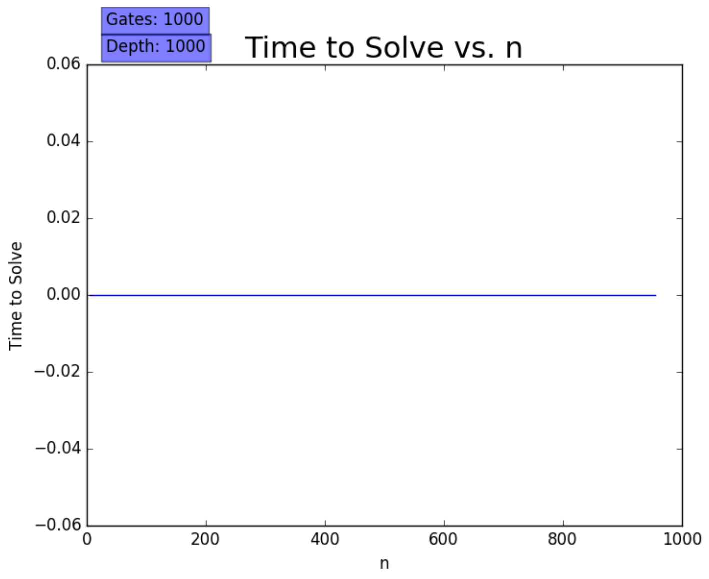

Figure 2: Gates: 1000, Depth: 1000

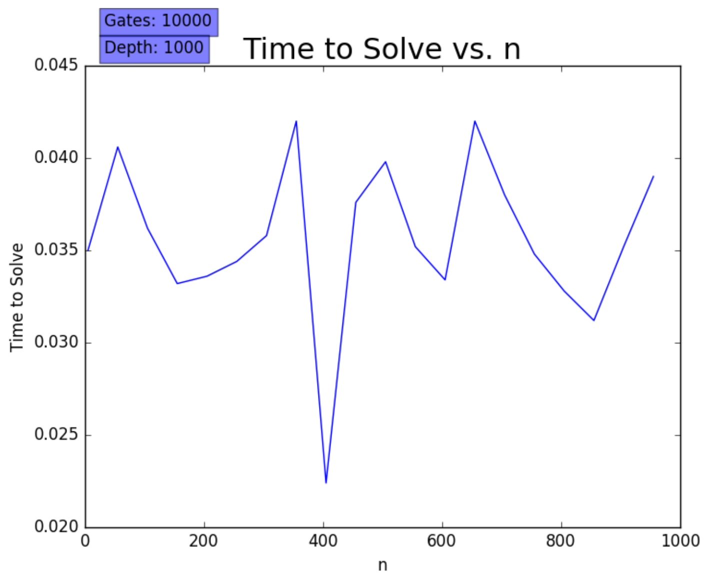

Figure 3: Gates: 10000, Depth: 1000

For all of our experiments, the fanin of each gate was constant. It was always 1 for NOT gates, and always 2 for AND and OR gates. Allowing for more fanin would require considerable changes to the Generator and the Reducer. We plan to make these changes and report back with the results.

The solver ran surprisingly fast for most instances. A lot of the time it solved the circuit so fast that it would record it taking 0 seconds. We found that in situations where m = d, the solver would take 0 seconds to run. If the graphs for the first batch (n = 5, 100, 5) weren't 0, then they were "Zig-Zaggy", showing that as n increases, the solver does not fine the circuit harder to solve on average. For the second batch (n = 5, 1000, 50) there were some interesting results, but nothing that showed the solver finding significant differences in time to find the solution. These graphs just connect the shattered data points. Implementing a line of best fit would help to visualize the results better.

Our solver seems to not depend on n. Again, I believe this is due to the fact that with the limited fanin, the solver is able to very quickly identify if it is satisfiable or not. When the solver doesn't solve it instantly (i.e. when d=m), the number of satisfiable circuits seems to range between $10\% - 80\%$. Below are some example figures of how long the solver took to solve a problem with respect to n.

5 Additional experiments

In terms of additional work, I believe that altering the parameters for m and d suffice to show more than just the solvability over time. We ran hundreds of iterations of the problem for each change in variables. We will have to revisit the problem, altering the fanin values as well as produce graphs with lines of best fit to visualize the results better.

6 Conclusion

We experimented with changing the values for n (5 to 1000), m (10 to 10000), and d (3 to 1000). For all smaller instances of the problem, the solver always solved it very fast. When d = m, it was always find a solution immediately as well. Even when the solver took some time to find a solution, there doesn't seem to be a relation between the time it took to solve the problem and n. We will have to retry the experiment allowing for variance in fanin.

n (start, stop, increment)	**m**	**d (<= m)**	**Max Time**	**Figure No.**
5, 100, 5	10	3	0	
”	100	3	0	
”	1000	3	0	
”	10000	3	0.45	
”	10	10	0	
”	100	10	0.004	
”	1000	10	0.012	
”	10000	10	0.5	
”	10			
”	100	100	0	
”	1000	100	0.025	
”	10000	100	0.8	1
”	10			
”	100			
”	1000	1000	0	
	10000	1000	0.5	
5, 1000, 50	10	3	0	
”	100	3	0	
”	1000	3	0.025	
”	10000	3	0.6	
”	10	10	0	
”	100	10	0	
”	1000	10	0.25	
”	10000	10	0.058	
”	10			
”	100	100	0	
”	1000	100	0.0025	
”	10000	100	0.075	
”	10			
”	100			
”	1000	1000	0	2
”	10000	1000	0.05	3