Satisfying Circuit Satisfiability

Ethan Joachim Eldridge

***Abstract*—The natural NP-Complete problem Circuit Satisfiability (CIRCUIT-SAT) is a well known problem in the field of Computer Science. In this paper the author proposes a linear time polynomial algorithm to solve any arbitrary CirSat instance. This is done by first transforming a boolean expression of the circuit into a graph-like tree structure, then traversing this graph and using the concept of restrictiveness and constraints upon each depth of the expression tree. The resultant of this algorithm is a firm answer on whether or not the given circuit is satisfiable, as well as the set of inputs to the circuit which cause a circuit to output the required result.**

CONTENTS

I. THE PROBLEM AT HAND

The problem of circuit satisfiability (CIRCUIT-SAT) can be described as follows: Given a circuit with a single output and any combination of logical OR, AND, or NOT gates, does there exist a set of inputs to the circuit which will cause the circuit to output a high or true signal? To simplify notation we will refer to each gate by their symbolic notation noted in table I.

CIRCUIT-SAT is clearly an NP problem, as it is obvious that any given circuit can be constructed and its output verified to be 1 or 0. In fact, it has been proven to be NP-Complete[1]. This would suggest by definition that CIRCUIT-SAT is as hard any other NP-Hard problem, and many proofs of NP-Completeness use CIRCUIT-SAT during the polynomial time reduction step of proving that a decision problem is difficult. However, by formulating CIRCUIT-SAT in the right manner, and approaching its analysis not through the size of its input we show that CIRCUIT-SAT's decision problem can be solved in polynomial time. What this means for all proofs and problems based around the n̈atural completenessöf circuit satisfiability is left as a thought experiment for the reader.

[1]http://people.clarkson.edu/~alexis/PCMI/Notes/lectureB07.pdf

Gate	Symbol
OR	$\vee$
NOT	$\neg$
AND	$\wedge$
HIGH INPUT	1
LOW INPUT	2
UNSET INPUT	ι

Table I
LOGICAL GATES AND THEIR CORRESPONDING NOTATION

II. PROBLEM TRANSFORMATION

In its natural state, a circuit can be described simply through symbols. For example, a simple negation circuit with an input x can be expressed by $(\neg x)$. We can represent any number of inputs through variables, and will use I to indicate a set of inputs, and ι to express an unset input. To clarify, an unset input is simply a variable which has not been changed to either 1 or 0 yet. Thus, for a circuit C, its inputs may be expressed as $\iota \in I$. Since any circuit is simply a combination of logical gates ($\vee$,$\neg$,$\wedge$), we can express any arbitrary gate as γ, and the full set of gates within a circuit as Γ. Using this notation we have a formal description of what any circuit is, A circuit is simply the combination of its gates Γ with an input set I, which when resolved, output either 1 or 0.

Using this notation we can take a circuit expressed in boolean logic and create an expression tree. An expression tree is a directed graph data

structure. And is in fact the same as a binary tree with two exceptions, leaf nodes represent $\iota \in I$. Every other node represents a single $\gamma \in \Gamma$. The number of children a node has is dependent on which type of logical gate it is. For the purpose of simplicity and without loss of generality, expression tree nodes may have a maximum of two children[2]. In addition to the number of children each node may have, as in a circuit, an input node may have its value mapped to any input within the tree. This can cause a given circuit C to satisfy to take on a somewhat cumbersome expression tree[3], but it can be done through the use of pointers and proper formulation of an expression tree.

As of this writing, the author has yet to write a parser or processor that takes as input a circuit expressed in boolean notation and creates the neccesary data expression tree. But that is simply a pre-processing step and more than surely can be done in polynomial time as well. Once the problem has been translated into the proper data structure, the algorithm proposed here will not only determine the existence of the set I that satisfy the circuit C, but also implicitly return the solution set $\iota = (1|0)\forall \iota \in I$ that satisfies C.

III. The Expression Tree Data Structure

As noted previously, an expression tree is a directed graph structure with similarities to a binary tree. Each node contains two variables, a node type variable, and a list of children. The node type can take on any of the values in table I, and is used to determine the course of action taken at each node during a traversal. All leaf nodes will have their node type taken from the input set I, and all gates from the root down to the deepest level of the tree are taken from the gate set Γ.

To see, compare the follow representations of the circuit $C = (\iota_1 \vee \iota_2) \wedge (\iota_1)$. If we were to draw the diagram for this circuit it would look something like the one shown in figure 1. Its corresponding expression tree could be visualized as in figure 2.

[2]It is easy to see that this is in fact general. As a 3-input $\wedge$ gate may be represented as two 2-input $\wedge$ gates with the output of one directed to the input of the other. And similarally for n-input $\vee$ gates. Reducing a complex circuit using n-arry logic gates can be done in polynomial time by simply replacing each n-arry gate with its equavalent logic in binary gate form.

[3]As we keep the number of children of any node to be two, if a gate takes three inputs it is neccesary to express this logic using binary logic and a combination of gates instead of a singular one.

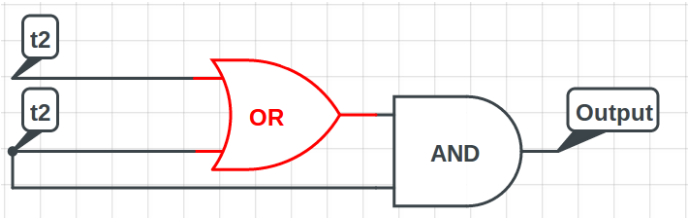

Figure 1. The circuit diagram of $C = (\iota_1 \vee \iota_2) \wedge (\iota_1)$. t1 and t2 correspond to ι_1 and ι_2 respectfully.

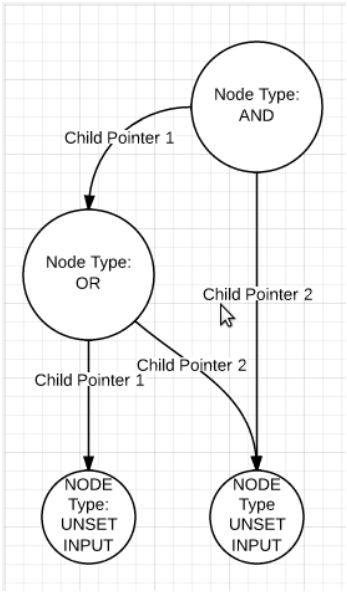

Figure 2. The (strict) expression tree generated for the circuit $C = (\iota_1 \vee \iota_2) \wedge (\iota_1)$

Its simple to see the basic mapping between the boolean expression of a circuit, its circuit diagram, and its representation as an expression tree. An expression tree can be traversed in the same manner as a binary tree, starting at the root and following each child recursively through the tree and performing some operation upon each node. It may not be immediately obvious, but this will take time directly related to the number of outputs of each individual node. Take the circuit which maps the output of an $\wedge$ gate to the two inputs of an $\vee$ gate, it is clear that the $\wedge$ gate will be visited twice, once on each recursive call down the $\vee$ gates child edges. In order to simplify the analysis of the runtime of a tree traversal of an expression tree, let us define the notion of a strict expression tree. A strict expression tree does not allow any gates who are not leaves to have multiple parents. While this does bloat the size of the tree, it is semantically equivalent to an non-strict tree. If we define the size of a strict expression

tree T' to be the number of nodes N within the tree, then we can define the cardinality $|T|$ of an expression tree T to be equal to the size of its strict equivalent[4]. Using this notion, we can express the runtime of a tree traversal on an expression tree T to take $\emptyset(|T|)$ time.

IV. Gate Constraints and Restrictions

In order to understand the algorithm, one must understand the intuition behind its design first. If we specify that the output of a circuit must be either be 1 or 0, then what we truly have done is constrained the output to be what we desire. In terms of the expression tree, we have required that its root node, must return a specific value. In order to meet this constraint, a gates inputs must be set accordingly. Logically, one can see this as the stored values in a truth table for any given gate. For example, an $\wedge$ gate, if constrained to output a 1 must have both its children inputs return a 1 as well. At the same time, we can see that the number of possible ways to satisfy this constraint on any node varies between what is required of it.

This leads to the concept of restrictiveness. In its most intuitive form, the more ways a node can meet the constraint of its parent, the less restrictive it is. We define an ordering to the restrictiveness of the gate set Γ, expressed in inequality 1. The logic is as follows: A set input gate cannot be changed, thus it is the most restricted and restrictive. An unset input ι has two possible options and is able to become either, in this sense, its placement in the inequality does not make sense. However, this ordering will be used within the algorithm and the reasoning for ι's inclusion in the top bracket of restrictivenes will be explained in the next section. Next, $\wedge$ gates and the combination of an $\vee$ and $\neg$ gate are very restrictive since there is only one of four possible set of inputs that can cause a high output.Similarally, $\vee$ gates are not very restrictive becase they only require a high output from one of their children. It is worth mentioning that the concept of restrictiveness is attached to the idea of any one node outputing a 1 signal, and the ordering would need to be rearranged if desiring the opposite. However, in practice it is found that this ordering works well.

$$1 = 0 > \iota > \wedge = \neg\vee > \vee = \neg\wedge > \neg \qquad (1)$$

[4]Because no nodes may have multiple parents, a strict expression tree may have an excess number of nodes that a regular expression tree could do without, because of this, analysis using an strict expression tree defines an upper bound on the time and not a completely strict one. For example the simple $(x \wedge y) \vee (x \wedge y)$ can be expressed by 4 nodes in a regular expression tree but would be represented by 5 in a strict one (as leaf nodes must be allowed to have multiple parents in order to find contradictions)

V. The Algorithm

In essence, the algorithm works as follows:

Data: Node n, Constraint c
Result: Boolean answer to decision problem, implicit solution stored in tree

```
1  if isInput(n) then
2      if n meets constraint OR unset then
3          n.type = constraint;
4          return True;
5      else
6          return False;
7      end
8  end
9  orderChildren(n);
10 switch n.type do
11     case ∧
12         leftSatisfy = CIR-SAT(n.leftChild,c);
13         rightSatisfy = CIR-SAT(n.rightChild,c);
14         if c is 0 then
15             return leftSatisfy OR rightSatisfy;
16         else
17             return leftSatisfy AND rightSatisfy;
18         end
19     case ∨
20         leftSatisfy = CIR-SAT(n.leftChild,c);
21         rightSatisfy = CIR-SAT(n.rightChild,c);
22         if c is 0 then
23             return leftSatisfy AND rightSatisfy;
24         else
25             return leftSatisfy OR rightSatisfy;
26         end
27     case ¬
28         return CIR-SAT(n, negate c);
29 end
```

Algorithm 1: The CIR-SAT algorithm.

A few notes about the algorithm. The call to the *orderChildren* is performed only on binary gates and swaps the most restrictive child to be the left child. This is done because the algorithm traverses the tree from left to right, and it makes sense to attempt to find contradictions as soon as possible. Swapping children of any node does not change the semantic meaning of the expression tree even if it does alter

the structure itself. This ordering is fast, taking six comparisons at most[5]. The constraint c is either 1 or 0, this is why the assignment of the input node's type occurs. Recall from previous sections that each Node within an expression tree has only a type and a list of children.

The written algorithm itself has a few minor differences from the listing here, such as checks against null pointers and the number of children each node has, but the core concepts and method of the algorithm is listed in listing 1. The code itself is written in C, and relys on using pointers in order to update each node so that contradictions in what is required of each leaf may be found during the run of the algorithm. The key concept of the algorithm itself is based on the notion of passing a constraint throughout the tree and trying to find a contradiction whilst doing so. While listing 1 will perform a full tree traversal, there are checks one could do to prune the expression tree and return prior to full traversal[6]. Another optimization woudl be to simply recursively call the function on each child directly instead of creating the *leftSatisfy* and *rightSatisfy* variables. Doing so could result in short circuiting the **AND** clauses of the return statements from $\wedge$ and $\vee$ gates[7].

A. Correctness and Runtime

It is obvious that any well formed expression tree will result in termination of the algorithm. The algorithm will traverse the tree from left to right in its entirety, ending at the last return to the root node and determining if the entire tree is satisfied. Whenever the traversal reaches an input node, it will either set that node to be what the nodes parents have constrained it to be, or will return a contradiction. Because of the use of pointers, even if multiple gates map one of their children to the same input, once the value of the node is set, it cannot be changed to satisfy another parent.

Essentially, although we use a top down approach, one can see the return statements of the algorithm acting as a bottom up solution to each smaller gate. Take a bottom layer of all $\wedge$ gates. Once the set of inputs have been determined, you could replace each of these gates with the result of the boolean expression between the *leftSatisify* and *rightSatisfy*. If a gate requires this resultant to be something else, then there is a contradiction and the circuit could not possibily be satisfied for root gates of $\wedge$ or $\neg$[8]. Because the constraint is passed from the top level down, a circuit which cannot be satisfied at a deeper level cannot be satisfied at a shallow level if the gates between do not include an $\vee$ gate[9].

Macroscopically, it is easy to see that the runtime analysis is trivial given the strict form of the expression tree of a given circuit. Since each node will be touched once by the traversal[10], then the time taken at each node must simply be multiplied by the cardinality $|T|$ of the expression tree T [11]. If we allow the time spent at each gate to be denoted G we can see that the **CIR-SAT** algorithm will take time $\emptyset(|T|G)$. Note that G is different for unary gates, inputs, and binary gates. To have a more precise runtime we can examine the number of operations and their runtime done.

At each $\vee$ or $\wedge$ gate, we perform the *orderChildren* operation, which as noted before takes six comparison operations to determine the order of the two children, and then we perform the actual swap, which takes three assignments[12]. We allocate space for the left and right satisfied variables, and after performing a recursive call, we perform one more comparison against the constraint before performing the final logical expression in the return statement. In total, we perform 13 operations and 2 recursive

[5]One comparison for each type of gate.

[6]One such early return could be done when exploring a node whose type is $\wedge$ and the constraint is 1. If the left childs return from a recursion results in false, one could return early without testing the right child.

[7]By this we mean that after finding the left satisfiability of a node, if false, the statement would read $0\textbf{AND}CIR - SAT(n.rightChild, c)$ which we immediately know will be false.

[8]An $\vee$ gate at the root would require that either both branches be satisfied, which is why pruning can be difficult.

[9]Or its equivalent of $\neg\wedge$.

[10]This is true in the case of strict expression trees because multiple parents of non-leaf nodes are disallowed, however it is noted that strict and non-strict expression tree's semantic meaning is the same, and thus there is no loss of generality between the runtime of the algorithm on a strict or non-strict expression tree. Leaf nodes may be touched multiple times by the algorithm as an input may be used multiple times in any given circuit expression, this does not fundamentally effect the runtime, as the number of touches on a leaf node will not be greater than the total number of leaf nodes themselves.

[11]The expression tree's cardinality, as defined before, is the size or number of nodes of its strict expression tree semantic equivalent

[12]This also involves the time spent allocating a temporary variable, but this could be overcome using XOR operations to be done with just two variables.

calls. At any $\neg$ gates we perform two operations to negate the constraint given to the node before passing it down to the recursive call. Finally, at each 1,0, or ι type node we perform two comparisons and possibly one assignment if the input is unset. For all nodes[13], we have to perform the time it takes to allocate the two variables, as well as a check to see if the node is of type $\neg$ or not[14].

For a tight bound, let the number of $\neg$ gates in a circuit be expressed as $|\neg|$, the number of $\wedge$ gates be expressed as $|\wedge|$, the number of $\vee$ gates be expressed as $|\vee|$, and similarally, the number of gates with type 1,0, or ι be expressed as $|I|$. Then, working with a strict expression tree, it is obvious that the number of operations during algorithm execution is:

$$\emptyset(13(|\wedge|+|\vee|)+2|\neg|+3|I|+3|T|) \qquad (2)$$

Given that operations at each node are fixed, we can consider them one constant time operation for each node, again calling this G as we did previously, we arrive at $G|T|+3|T|$, which of course can be reduced to $O(|T|G)$ if $G>3$ (Which it will be on most gates).

B. Extensions to n-arry Nodes

Although the algorithm above is phrased in the terms of a maximum of two children per node, one could use a list of any arbirary size of children as well as a list of s̈atisfiedsïnstead of *leftSatisfy* and *rightSatisfy*. Using this would result in the ability to keep the number of nodes to a minumum and possibly decrease the runtime of the algorithm even more. For example, A circuit with an 8-input $\wedge$ gate could be represented by nine nodes using n-arry nodes, or by 17 total gates (8 input and 9 $\wedge$) if we are dealing with a strict expression tree. In the case of the first input being previously set to 0, we could cease execution of the algorithm there given that we optimized the algorithm according to the suggestions from section V. This would save us a fair amount of time depending on the types of circuits we are given.

[13]We could optimize this allocation out from the $\neg$ case of each node.

[14]This check is performed in order to apply the *orderChildren* function to binary gates only, as $\neg$ gates are unary and have no children to swap.

VI. Conclusions

In this paper, the author proposes a solution to the NP-Complete problem of circuit satisfiability which runs in polynomial time. While the runtime analysis is ad-hoc, The claim of a polynomial time algorithm is support by the fact that the algorithm itself is only a tree traversal. While it is traversing a unique type of tree, this tree is semantically equivalent to a stricter version of the data structure that has a worse time than the regular. Becuase the scrict expression tree's runtime is also polynomial due to each gate having constant time operations, and the algorithm only touching each non-leaf node once, its clear to see that this tree traversal takes at worst $O(N)$, where N is the number of leaf nodes in the strict expression tree. By defining the cardinality of an expression tree $|T|$ to be equal to or less than N, we can see that the regular tree traversal will take no longer than a strict tree traversal, and therefore is in polynomial time.