LSAT - An Algorithm for the Synthesis of Two Level Threshold Gate Networks

Arlindo L. Oliveira

Dept. of EECS
UC Berkeley
Berkeley CA 94720

Alberto Sangiovanni-Vincentelli

Dept. of EECS
UC Berkeley
Berkeley CA 94720

Abstract

We present an algorithm for the synthesis of two-level threshold gate networks inspired in techniques used in classical two-level minimization of logic circuits. We specifically address a restricted version of the problem where the on and off set minterms are explicitly listed. Experimental results show that a simple branch and bound algorithm can be used to obtain solutions close to the absolute minimum in a set of standard problems, outperforming other minimizers even when restricted to use only classic logic gates as building blocks.

1 Introduction

Threshold gate networks have been the focus of an increasing interest in the research community in the last few years. In part, this interest is due to theoretical work that shows that threshold gates are more powerful than logic (*and* & *or*) gates, in the sense that polynomial size, bounded level networks of threshold gates can implement functions that require unbounded level networks of purely logic gates. For example, it has been shown that functions like multiple-addition, multiplication, division and sorting can be implemented by polynomial-size threshold circuits of small constant depth [1, 2]. While it is unclear whether or not these results will have practical implications in the design of real integrated circuits, the study of algorithms for the synthesis of threshold gate networks remains important in other areas, such as artificial neural networks and machine learning.

This paper proposes an algorithm for the synthesis of two-level, single output threshold gate networks when the target function is specified by two sets of minterms, the *on* and the *off* set. This particular formulation turns out to be the natural one in a class of problems that is known as the *rule induction from examples* problem in the machine learning literature.

An instance of the problem is created as follows: minterms are randomly chosen according to some probability distribution and classified as 0 or 1, according to the value of some boolean function, f. The target of the learning system is to derive g, an approximation of function f by just looking at the classified samples it was presented with. Performance is measured by checking the agreement between the output of functions g and f for a set of minterms drawn according to the same distribution. Theoretical results predict that, under reasonable assumptions [3], simpler descriptions for g should be expected to lead to better results when predicting the class of future samples.

2 Definitions

Let f be an incompletely specified function of n boolean variables, $\{x_1, ..., x_n\}$, i.e., a mapping from $\{0,1\}^n \rightarrow \{0,1,\star\}$, where the $\star$ is used to identify the input points with unspecified output values.

Function f can be described by the following disjoint sets of minterms: f_{ON}, f_{DC} and f_{OFF}. The f_{ON} (f_{OFF}) set specifies which minterms should turn the output *on (off)*. Non trivial boolean functions should have non-empty f_{ON} and f_{OFF} sets. The f_{DC} (don't care) set contains all minterms not contained in the previous sets. A function g is compatible with f iff the value of g for every point in the f_{ON} and f_{OFF} sets if compatible with f, i. e., iff $f_{ON} \subseteq g_{ON} \subseteq f_{ON} \cup f_{DC}$.

From a set of positive and negative examples defining the f_{ON} and f_{OFF} sets of a function f, we want to derive a compatible function g such that the size of the network needed to implement g is minimal.

2.1 Cubes and pyramids

Let a literal be either a variable or its negation. A cube is a conjunction of k literals ($1 \leq k \leq n$), where no two literals corresponding to the same variable appear. A cube with n literals corresponds to a minterm. A cube c_1 is said to contain another cube c_2 if $c_2 \Rightarrow c_1$, i.e., if the truth values defined in c_2 make c_1 true. If $c_1 \neq c_2$, then such a containment is proper. The dimension of a cube c is the number of variables not present in c. The distance between two cubes, c_1 and c_2, $\delta(c_1, c_2)$ is the number of variables

CH3026-2/91/0000/0130/$01.00 © 1991 IEEE

that appear negated in one cube and non-negated in the other. For example, $x_1\overline{x_3}x_4$ is at distance 2 from $\overline{x_1}x_2\overline{x_4}$.

A cube is identified with the boolean function implemented by an *and* gate. Consider now a threshold gate of k input variables with weights of value -1 or +1 and let the input variables assume the values 0 or 1. Let the threshold gate be *on* when $w_1x_1 + w_2x_2 + ... + w_kx_k \geq V_{thr}$, where it is assumed, without loss of generality, that the gate implements a function of the first k variables in an n-dimensional input space. Let V_{max} be the maximum value that the sum in the previous expression can assume and cube c be defined by the literals $c_1c_2...c_k$ where $c_i = x_i$ if $w_i = +1$ and $c_i = \overline{x_i}$ if $w_i = -1$. Consider now a minterm $m = m_1m_2..m_k..m_n$. This minterm will turn the gate *on* iff no more that $V_{max} - V_{thr}$ literals are different from the literals in c, i.e., if $\delta(c, m) \leq V_{max} - V_{thr}$. Let $h = V_{max} - V_{thr}$. We define a pyramid as a pair $(c : h)$, where c is a the apex cube and h is a non-negative integer value, the height. The minterms contained by a pyramid are at distance h or less from the apex cube.[1] Figure 1 shows graphical representations of a cube and a pyramid.

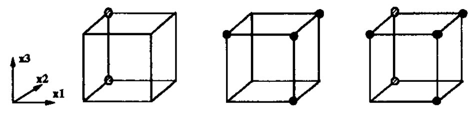

Figure 1: Cube $\overline{x_1}x_2$ and pyramid $(x_1\overline{x_2}x_3 : 1)$

The distance between a cube c_1 and a pyramid $(c_2 : h)$ is given by $\max(\delta(c_1, c_2) - h, 0)$. Cube $\overline{x_1}x_2$ and pyramid $(x_1\overline{x_2}x_3 : 1)$ (above) are at distance 1.

A pyramid p is a prime pyramid, relatively to some boolean function f, iff there is no other pyramid contained in the $f_{ON} \bigcup f_{DC}$ set that properly contains pyramid p.

Any pyramid $p = (c : h)$ has a complement pyramid, $\overline{p} = (\tilde{c} : k - h - 1)$, where k is the number of literals in c and $\tilde{c}$ is the cube obtained by complementing all such literals. It is trivial to show that $p\overline{p} = \emptyset$ and $p \cup \overline{p} = \{0, 1\}^n$.

2.2 Covers and *M*-covers

The concept of cube cover of a boolean function can be generalized in a way that preserves the relation between a cover and a two-level implementation of f but allows for the use of general threshold gates in both the first and second levels.

A set of pyramids is a pyramid cover for a function f if every minterm in f_{ON} is contained in at least one pyramid and no minterm in the f_{OFF} set is contained in any pyramid. This concept of cover can be further extended to a more general one that leads to implementations where the second level gate is also a general threshold gate instead of an *or* gate. A bag of pyramids, B, is an *M*-cover ($M \geq 1$) for a function f iff all minterms in the f_{ON} set are covered by at least M pyramids and all minterms in the f_{OFF} set are covered by at most $M - 1$ pyramids in B. Given an *M*-cover for f, it is a trivial task to derive a two-level network of threshold gates that implements f: simply allocate one threshold gate for every pyramid in B and connect them to a gate in the second level with all $w_i = 1$ and $V_{thr} = M$. The search for a two-level network that minimizes the number of gates is therefore equivalent to the search for an *M*-cover for the specified function with a minimal number of pyramids in B.

[1]For a more detailed description of the relation between threshold gates and logic functions, see [6].

3 The synthesis algorithm

Assume we have one solution, i.e., an *M*-cover for the function is known. A branch and bound algorithm is used to search for an *M*-cover with one less pyramid. A search tree is built with a predefined branching factor r, and, at each step, the most promising node in the tree is explored.

The root corresponds to the initial solution with one pyramid removed. Every node n_i in the search tree corresponds to a bag B_i of pyramids and has a value v_i given by the heuristic $Cu - z$, where u is the number of f_{ON} minterms covered less than M times, z is the number of f_{ON} minterms covered more than M times and C some large constant. If v_i is non-positive, then node n_i represents a solution with one less pyramid than the root. The value of a node estimates how close it is to a solution. Minterms covered more that M times are weighted negatively because some pyramids that are currently covering these minterms are potentially free to cover other points in the space.

At each step, the node with a smaller value of v_i is chosen, and its descendents generated. The bag of pyramids for each children is obtained by changing one of the pyramids in B_i such that it covers a new minterm. This is performed by applying the following algorithm to node n_i:

```
$m \leftarrow$ pick_one_uncovered_minterm($n_i$);
for $j = 1$ to $r$ do {
    $p_{old} \leftarrow$ choose_next_pyr($m$);
    $(\mathcal{S}_{on}, \mathcal{S}_{off}) \leftarrow$ build_s_sets($m, n_i, p_{old}$);
    $p \leftarrow$find_max_red_pyr($\mathcal{S}_{on}, \mathcal{S}_{off}$);
    if $p \neq \emptyset$
        $n_j \leftarrow$ create_child($n_i$, $B_i \cup \{p\} \setminus \{p_{old}\}$);
}
```

After picking one of the f_{ON} minterms covered less than M times, the algorithm selects the r pyramids closer to it as the candidates. Function build_s_sets creates the $\mathcal{S}_{on}$ set by adding minterm m to the list of

f_{ON} minterms covered by p and not covered more than M times. The $\mathcal{S}_{off}$ set consists of all the minterms in the f_{OFF} set that are already covered by $M-1$ pyramids but not the selected one.

3.1 Expanding and reducing pyramids

Function find_max_red_pyr derives the maximally reduced pyramid covering all minterms in the $\mathcal{S}_{on}$ set and none in the $\mathcal{S}_{off}$ set. The algorithm starts by identifying $\mathcal{S}_{red}$, the set of minterms in $\mathcal{S}_{on}$ already covered by the pyramid. The pyramid is first reduced with respect to this set[2] and then expanded until either all minterms in the $\mathcal{S}_{on}$ set are covered or a prime is obtained. If the second condition holds and the first doesn't, it reports failure. Otherwise, it returns the expanded pyramid.[3]

A pyramid can be expanded by either: a) expanding the apex cube, by dropping one literal. b) reducing the apex cube, by adding one literal while increasing the pyramid height. It can be reduced by either applying the reverse operations or by expanding its complement pyramid. The reader may verify that the expand (reduce) operation does indeed generate a pyramid that properly contains (is contained) in the original one. Figure 2 shows an example of the application of the expand operations.

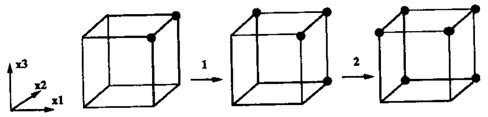

Figure 2: $(x_1x_3:0) \rightarrow (x_1x_2x_3:1) \rightarrow (x_2x_3:1)$

A detailed analysis shows that the procedure has a worst case complexity of $O(tm^2n)$ and an expected complexity of $O((t+m)mn)$, where m is the number of minterms and t the maximum number of nodes in the search tree.

3.2 Changing M

The algorithm is started with a 1-cover ($M=1$) consisting of all the minterms in the f_{ON} set. This cover is then reduced as much as possible while keeping M constant. When the maximum size of the search tree is reached, the value of M is increased and a new search for a smaller M-cover is started,

In the outer loop M is treated like a variable and increased when the search for a better solution fails. The procedure stops when a value of M is reached for which no solution is found. This procedure may fail to find the best M-cover, even if the inner loop always finds the global optimum for a given M. In fact, we may fail to find a good solution for $M = M_{opt}$ if no good solution exist for $M_{opt}-1$ because the algorithm will stop when no $(M_{opt}-1)$-cover is found. Empirical validation, however, has shown that, in general, there is an optimal value for M, M_{opt}, and the cardinality of the solutions increases with $|M - M_{opt}|$.

If an M-cover is the final solution, $O(M)$ searches for a cover are conducted in the inner loops. Together with the results in the previous section, we should expect the algorithm to take an $O(M(t+m)mn)$ time to find and optimal M-cover.

3.3 Non-unitary weights

As described above, a pyramid corresponds to a threshold gate with weights of +1 or -1. A search for an implementation with larger integer weights is equivalent to a search for a pyramid in a space of higher dimensionality. Consider a function f of n variables, $\{v_1, v_2 ... v_n\}$ defined by its f_{ON} and f_{OFF} sets. Let z be an integer and f^z be a function of zn variables defined by its f^z_{ON} and f^z_{OFF} sets. Every minterm in f^z_{ON} (f^z_{OFF}) is obtained by replicating z times every variable in the corresponding minterm of f_{ON} (f_{OFF}). For example, if $z=2$ and $n=3$ the minterm $x_1x_2\overline{x_3}$ is converted to $x_{1_1}x_{1_2}x_{2_1}x_{2_2}\overline{x_{3_1}}\,\overline{x_{3_2}}$. Now, let f be a boolean function of n variables such that f is implementable by a single threshold gate with integer input weights $\{w_1, w_2 ... w_n\}$. Let z be the maximum value of $|w_1|, |w_2| ... |w_n|$. Then there exists a pyramid in the space of zn dimensions that covers all the minterms in the f^z_{ON} set and none in the f^z_{OFF}. This fact implies that if we want to restrict the input weights of the first level threshold gates to the range $[-z,+z]$ then, for a function of n variables, we should perform a circuit minimization in a space of zn dimensions.

4 Experimental results

We present the results obtained in two sets of examples. All results were obtained in a DECstation 3100 and all CPU times are in seconds. In the first set, we compared the results obtained with the theoretical minimum cost realizations. All the problems require the use of non-degenerate threshold gates (i.e., threshold gates different from either *and* and *or* gates) to achieve the minimum realization. The asymmetry problems also require weights larger than 1, and this is reflected in the number of variables (n) for each problem. The results are shown in table 1. As before, m is the number of minterms and E and T are the number of gates obtained experimentally and the theoretical minimum. These results show that we were able to obtain the minimum size realization for all but the two larger parity problems. The sharp increase in CPU time for the larger asymmetry problems is due, in part, to the large weights needed.

[2] I.e., it is reduced as much as possible while still covering all the minterms in $\mathcal{S}_{red}$.

[3] A detailed description of the procedure can be found in [7].

Problem	m	n	T	E	T_{cpu}
6-parity	64	6	6	6	4.3
7-parity	128	7	7	7	10.7
8-parity	256	8	8	8	61.9
9-parity	512	9	9	11	878.1
10-parity	1024	10	10	24	1759.8
6-asymmetry	64	24	2	2	1.5
8-asymmetry	256	64	2	2	23.2
10-asymmetry	1024	160	2	2	408.6
12-asymmetry	4096	384	2	2	11752.9

Table 1: Experiments using threshold gates.

In the second set of problems, we compared the performance of the described algorithm with the performance of a popular two-level minimizer, ESPRESSO [4]. To allow for a fair comparison, LSAT was constrained to use only *and* gates in the first level and *or* gates in the second level. A detailed description of the functions used for these tests can be found in [8]. The input data sets were created by randomly generating a minterm and getting its correct classification from the function description. For each function, 2 different inputs were tried, with a number of minterms of 200 and 600. According to the generation procedure, all instances of the problem should accept a solution with no more cubes than the theoretical minimum.

Problem				ESPRESSO		LSAT	
Name	m	n	T	E	T_{cpu}	E	T_{cpu}
dnf1	200	80	6	6	144	6	87
	600	80	6	14	840	15	173
dnf2	200	40	8	8	84	9	21
	600	40	8	10	236	8	161
dnf3	200	32	6	7	19	6	15
	600	32	6	6	110	6	54
dnf4	200	64	10	13	183	9	80
	600	64	10	25	1997	10	506
mux11	200	32	8	14	39	8	33
	600	32	8	20	208	8	99
ex5of32	200	32	16	15	49	15	21
	600	32	16	40	363	41	111

Table 2: Experiments using logic gates.

Table 2 shows that although no specific code optimization was performed for the special case when a cube cover is to be found, the performance of the algorithm still compares well with a classic two-level optimizer. In particular, it obtains results that are either similar or better than ESPRESSO and much faster. Moreover, the speed gain increases with the size of the problem. This is due, in part, to the fact that LSAT does not require an explicit cover for the f_{DC} set while ESPRESSO does. Is is also clear from these results that both programs obtain results very far from the minimum in a large number of cases. A different approach to this optimization problem may lead to better results in these and other problems.

5 Conclusions and future work

We designed and implemented and algorithm for the synthesis of threshold gate networks and have shown, experimentally, that results near the theoretical minimum are obtainable in many problems. The algorithm has a run time polynomial in the input size and its performance degrades slowly with the size of the problem. Experimental results have also pointed out the need for a deeper understanding of this particular version of this optimization problem. In particular, the reason why, in some cases, different algorithms find similar solutions very far from the optimum requires further study.

The most interesting single direction for future work in this area is the extension of these techniques to the synthesis of multi-level networks. Classic multilevel techniques [5] can provide the starting point for such an algorithm. Another interesting research topic related with this work is the design of multi-valued threshold gate networks.

Acknowledgments

This work was supported by the Air Force Office of Scientific Research (AFOSR/JSEP) under Contract No. F49620-90-C-0029 and the Portuguese INVOTAN committee. The authors would like to thank Gregory Sorkin who suggested the use of threshold gates in logic synthesis and Jaijeet Roychowdhury who introduced the concept of pyramids.

References

[1] K. Y. Siu & J. Bruck "On the Power of Threshold Circuits with Small Weights", to appear in SIAM J. Discrete Math.

[2] K. Y. Siu & J. Bruck "Neural Computation of Arithmetic Functions" Proc. IEEE, 78, No. 10:1669-1675, October 1990.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler & M. Warmuth "Occam's Razor", Information Processing Letters, vol 24, pp. 377-380, North-Holland, 1987.

[4] R. Brayton, G. Hachtel, C. McMullen & A. Sangiovanni-Vincentelli "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[5] R. Brayton, G. Hachtel & A. Sangiovanni-Vincentelli "Multilevel Logic Synthesis", Proceedings of the IEEE, vol. 78:2, pp. 264-300, February 1990.

[6] S. Muroga "Threshold Logic and its Applications", Wiley-Interscience, 1971.

[7] A. Oliveira "Logic Synthesis Using Threshold Gates", Internal Memo, UC Berkeley, 1990.

[8] G. Pagallo & D. Haussler "Boolean Feature Discovery in Empirical Learning", Machine Learning, 5:71-99, 1990.