Area Minimization Using Decision Diagrams Without Constructing Them

Kristina Cherevko
National University of Kyiv-Mohyla Academy, Ukraine
christina.cherevko@gmail.com

Alan Mishchenko
University of California, Berkeley
alanmi@berkeley.edu

Abstract

This paper proposes a method for minimizing the number of two-input nodes in the circuit representation of a completely specified Boolean function. To find the smallest circuit, the method uses truth tables to enumerate variable orders and, for each variable order, attempts to use three canonical expansions while merging shared cofactors. Conceptually, it is similar to using decision diagrams without constructing them. Experimental results confirm that the method works well for a number of benchmarks.

1. Introduction

Area minimization is a central topic in logic synthesis, which, in turn, is one of the most challenging and research-worthy aspects of the electronic design automation (EDA) tools used in hardware design flows.

With the decrease in the feature size, the cost of silicon wafers continues to grow [10], which makes high-effort circuit minimization increasingly more important.

Other methods for area optimization use circuit transforms, such as rewriting [4], divisor extraction from the sum-of-product (SOP) representation [7], and various types of Boolean decomposition, for example, disjoint-support decomposition [3].

Some optimization methods are based on decision diagrams, such as binary decision diagrams (BDDs) [9], Kronecker Functional Decision Diagrams (KFDDs) [19], etc. In particular, [11] uses KFDDs to minimize the number of product terms in the exclusive SOP (ESOP) expressions.

The contribution of this paper is a novel KFDD-based method for area minimization, which extends [11] to minimize the number of two-input nodes rather than the number of ESOP product terms.

The proposed method is efficiently implemented using truth tables, without the need to construct and manipulate KFDDs in a dedicated software package.

Experimental results on the benchmarks from the IWLS 2022 Programming Contest [12] show that the results compare favorably with other methods.

The rest of this paper is organized as follows. Section 2 describes the background. Section 3 shows a motivating example. Section 4 describes the algorithm. Section 5 reports experimental results. Section 6 concludes the paper and outlines future work.

2. Background

2.1 Boolean network

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. In this paper, we consider only combinational Boolean networks.

A node *n* has zero or more *fanins,* i.e., nodes that are driving *n*, and zero or more *fanouts*, i.e., nodes driven by *n*. The *primary inputs* (PIs) are nodes without *fanins* in the current network. The *primary outputs* (POs) are a subset of network nodes, which deliver the functionality of the network to its environment.

The *size* (*area*) of a network is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations is to reduce area and delay of the network.

A *local* function of a network node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*. The *global function* of a node is its function expressed in terms of the PIs of the network.

2.2 And-inverter graph

A combinational *and-inverter graph* (AIG) is a Boolean network composed of two-input and-gates and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the and- and or-gates of the factored forms are converted into two-input and-gates and inverters using DeMorgan's rule, and these two-input and-gates are added to the AIG manager in a topological order.

AIGs can represent both local and global functions. Because of low memory usage, speed of manipulation, and scalability, AIGs have recently emerged as a widely used data-structure in logic synthesis and formal verification.

The following publications on AIGs [14] and AIG-based synthesis [15][16] contain additional information.

2.3 Structural and functional hashing

Structural hashing of AIGs ensures that all constants are propagated, and, for each pair of nodes, there is at most one two-input and-node with them as *fanins* (up to a permutation). Structural hashing is performed by hash-table lookups when and-nodes are created and added to an AIG manager. Structural hashing can be applied on-the-fly during AIG construction, which reduces the AIG size.

Functional hashing of the AIGs ensures that each internal node has a unique Boolean function up to the complement. Functional hashing leads to a “semi-canonical” AIG representation (also known as FRAIGs [17]) because no two nodes in the AIG have the same global functions, but the same global function in different AIGs can be expressed using different circuit structures.

2.4 Xor-and-inverter graph

The notion of an *and-inverter graph* can be extended to include two-input xor-gates along with two-input and-gates, resulting in an xor-and-inverter graph (XAIG). An XAIG is constructed in a similar way to an AIG. In the software implementation, an xor-gate and an and-gate are represented by a pair of integers. Since both operations are commutative, the node fanins can be swapped without changing the node function. Thus, the ordering of the fanins can be used to distinguish and-gates and xor-gates: when the first *fanin* is smaller than the second, this is an and-gate; otherwise, it is an xor-gate.

In the future, we will use AIGs, but all the definitions and computations are naturally extended to work with XAIGs.

2.5 Canonical expansions

The proposed algorithm is based on three canonical expansions of Boolean functions.

Shannon $$f = \bar{x}_i f_i^0 \vee x_i f_i^1 \qquad (1)$$

Positive Davio $$f = f_i^0 \oplus x_i f_i^2 \qquad (2)$$

Negative Daio $$f = f_i^1 \oplus \bar{x}_i f_i^2 \qquad (3)$$

Here f_i^0 (f_i^1) denote the *cofactor* of f with respect to $x_i = 0$ ($x_i = 1$) and f_i^2 is defined as $f_i^2 = f_i^0 \oplus f_i^1$, where $\oplus$ denotes the XOR operation.

The canonical expansions have multiple applications, in particular, they are the basis for creating various families of decision diagrams, such as BDDs, KFDDs, etc.

3. Motivation

This section shows a motivating example, followed by the outline of the proposed algorithm in the next section.

3.1 Illustrative example

Integer multiplication plays the central place in many computations, including machine learning. Hardware implementation of multipliers is the key element of many designs, including those used to accelerate computations in machine learning. A frequently used multiplier design is based on the Booth algorithm [6].

This is why, as an illustrative example in this section, we consider the partial product in a radix-4 Booth multiplier.

The partial product is a 5-input Boolean function whose truth table in the hexadecimal notation is 0xF335ACC0.

Figure 1 shows the circuit we get by applying only the Shannon expansion. Essentially, this is a circuit with 16 nodes derived from a BDD [9] with a fixed variable order. Using positive and negative Davio expansions results in a circuit with 9 two-input nodes (Figure 2). In this case, the circuit is a KFDD [19] with the same fixed variable order.

Our algorithm that uses truth tables to efficiently enumerate all variable orders leads to a circuit with 8 two-input nodes (Figure 3). The smallest circuit for this function is composed of 7 nodes (Figure 4) and was computed using SAT-based exact synthesis [13], as implemented in command *twoexact* in ABC [1]. (Figures 1-4 were generated using command *show*.)

One line in the functional representation of a node defines AND node, two lines – EXOR node.

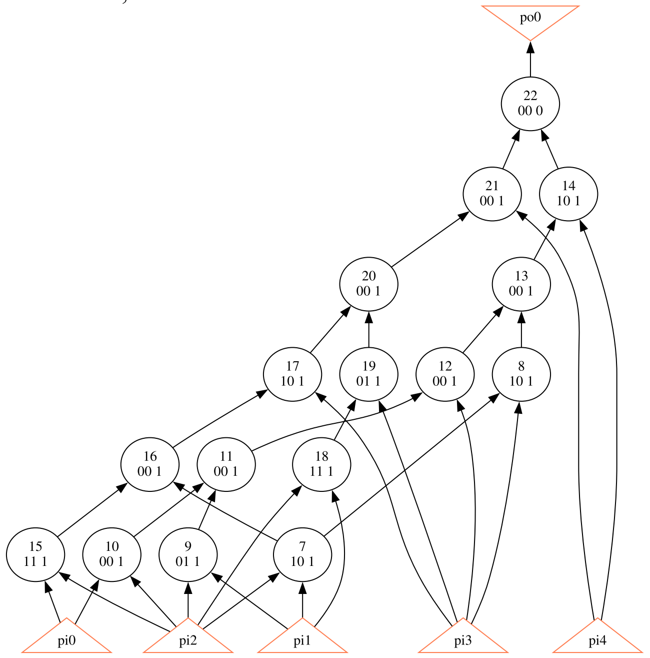

Figure 1: An AIG derived using the Shannon expansion with a fixed variable order.

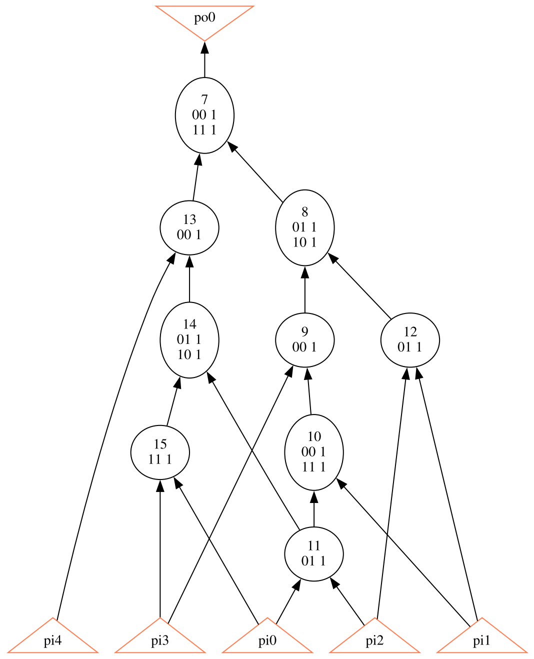

Figure 2: A minimized XAIG derived using three canonical expansions with a fixed variable order.

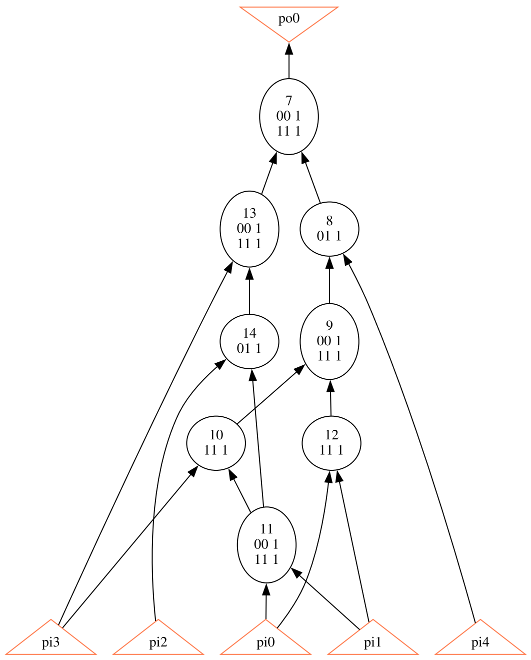

Figure 3: A minimized XAIG derived using three canonical expansions and different variable orders.

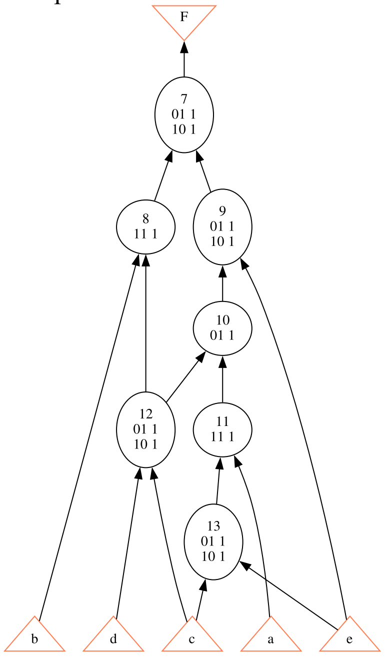

Figure 4: A provably minimum-node XAIG derived using SAT-based exact synthesis.

Although in this case our algorithm did not find the smallest possible implementation, it came close enough, missing the minimum circuit by only one node. Also, our algorithm was much faster, taking less than 0.01 sec vs about 1 sec for exact synthesis. It is likely that, by relaxing the variable order in the cofactors and adding heuristics for targeted search, a modified version of our algorithm could find a minimum-node implementation.

4. Algorithm

This section represents the main contribution of paper, an algorithm to build XAIG based on three canonical expansions. Algorithm is implemented using truth tables for functional representation.

4.1 Recursive procedure

The pseudo-code of the recursive traversal procedure is displayed in Figure 5. Instead of building a *decision diagram* and then traversing it, our algorithm builds *xor-and-inverter graph* (XAIG) on-the-fly. This is accomplished by first computing the subgraph size for each successor f_i^0, f_i^1 and f_i^2, and then determining the area of XAIG for each canonical expansion. The algorithm decides what canonical expansion leads to the minimal area and constructs the two-input nodes accordingly.

```
int Synthesis_XAIG_Rec(gg *gateGraph, int truthTableId, int varId) {
    int res0, res1, res2, res, n01, n02, n12;
    /* a hash table lookup to avoid synthesizing the same function twice */
    /* this step also handles the trivial cases of the constant functions */
    int iLit;
    if ((iLit = hash_function(gateGraph, truthTableId)) >= 0)
        return iLit;
    int f0 = cofactor0(gateGraph.funcs, truthTableId, varId);
    int f1 = cofactor1(gateGraph.funcs, truthTableId, varId);
    int f2 = xor(gateGraph.funcs, f0, f1);
    /* simplification for equal cofactors */
    if (f0 == f1)
        return Synthesis_XAIG_Rec(gateGraph, truthTableId, varId - 1);

    /* recursive traversal */
    int lit0 = Synthesis_XAIG_Rec(gateGraph, f0, varId - 1);
    int lit1 = Synthesis_XAIG_Rec(gateGraph, f1, varId - 1);
    int lit2 = Synthesis_XAIG_Rec(gateGraph, f2, varId - 1);
    kc_vt_shrink(gateGraph.funcs, 3);

    /* compute subgraph size */
    /* Shannon */
    n01 = nodeCount(lit0, lit1) + shannonNodeCount(lit0, lit1);
    /* positive Davio */
    n02 = nodeCount(lit0, lit2) + davioNodeCount(lit0, lit2);
    /* negative Davio */
    n12 = nodeCount(lit1, lit2) + davioNodeCount(lit1, lit2);

    /* construct nodes for minimal expansion */
    int minVal = min(n01, n02, n12);
    if (minVal == n01) {
        return mux(gateGraph, litPos(varId + 1), lit0, lit1);
    } else if (minVal == n02) {
        return and_xor(gateGraph, litPos(varId + 1), lit2, lit0);
    } else {
        return and_xor(gateGraph, litNeg(varId + 1), lit2, lit1);
    }
    return -1;
}
```

Figure 5. Recursive procedure for minimizing XAIG.

4.2 Generating permutations

To minimize the resulting XAIG, a recursive algorithm is used to enumerate all possible variable orders. The algorithm for generating variable permutations [5] gets the next permutation based on the current one, as shown in the pseudo-code in Figure 6.

```
void get_Next_Permutation(int *currentPerm, int nVars) {
    int i = nVars – 1;
    while (i ≥ 0 && currentPerm[i – 1] ≥ currentPerm[i])
        i = i – 1;
    if (i ≥ 0) {
        int j = nVars;
        while (j > 0 && currentPerm[j – 1] ≤ currentPerm[i – 1])
            j = j – 1;
        swapElements(j – 1, i – 1);
        i = i + 1;
        j = nVars;
        while (i < j) {
            swapElements(i – 1, j – 1);
            i = i + 1;
            j = j – 1;
        }
    }
}
```

Figure 6. The algorithm for generating permutations.

4.3 Improvements and simplifications

A simplification has been made in the recursive traversal procedure for the case of equal cofactors. Another improvement is to construct the truth table for verification during the XAIG building. This helps maintain uniqueness of the functions and enables efficient verification.

5. Experiments

The proposed algorithm is implemented in C/C++ and tested on the 33 testcases with 9 or less variables from IWLS 2022 Programming Contest [12]. The results are verified by constructing the truth table of the circuit and comparing it against the specification.

Table 1 lists the results of the synthesis algorithms. Column "Function" lists names of files containing truth tables. Column "Ins" ("Outs") shows the number of PIs (POs). Columns "Shannon", "Three expansions", "Three expansions and all variable orders" shows the areas of the circuits synthesized by the three algorithms: the one based on Shannon expansion, the one based on three canonical expansions, and the proposed algorithm, respectively. The runtime varied between 0.01 seconds and 80 hours, depending on the benchmark. The high runtime is explained by the need to enumerate all variable orders.

Although our algorithm typically produces better results than the other two algorithms, experiments with ex16 – ex20 have shown that circuit based on the Shannon expansion can be smaller. This is because our algorithm selects expansions based on the sizes of the subgraphs for the current node, rather than the total size of the graph. In these testcases, it appears that the Shannon expansion results in a smaller circuit size. Nonetheless, our algorithm remains a strong performer overall.

Unfortunately, the synthesized circuits cannot be directly compared against those produced by the participants of the IWLS 2022 Programming Contest, because the contest circuits use only and-nodes, while our circuits use both and- and xor-nodes. One way to compare the results, is to convert our results into and-nodes and post-process them using command *&deepsyn –I 10 –J 1000* in ABC.

The last two columns in Table 1 show the result of this comparison. The conclusion is that, after post-processing, the synthesized circuits for 14 out of 33 testcases have the same sizes as the best one produced at the competition, and in four cases (ex20, ex29, ex51, ex55) the circuit is smaller.

6. Conclusions

This paper introduces a new algorithm for minimizing the number of two-input nodes in the circuit implementations of multi-output completely specified Boolean functions.

Future work may include exploring heuristic tradeoffs between quality and runtime. For example, it may be possible to limit the exhaustive enumeration of variable orders while preserving the minimality of the resulting circuits or allow for cofactoring paths to have different variables orders, as in free BDDs [2].

REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] J. Bern, J. Gergov, C. Meinel and A. Slobodova´, "Boolean manipulation with free BDDs. First experimental results", *Proc. DATE'94*, pp. 200-207.

[3] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD '97*, pp. 78-82.

[4] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[5] E. Blurock, *Generate all Permutations of an Array,*

[6] A. D. Booth. "A signed binary multiplication technique". *Quarterly Journal of Mechanics and Applied Mathematics*. 1951, IV (2), pp. 236–240.

[7] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[8] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[9] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation" *IEEE Tr. Comp*., vol. C-35, pp. 677–691, Aug. 1986.

[10] I. Cutress, *More than Moore*, https://morethanmoore.substack.com/p/tsmc-financial-year-2022

[11] R. Drechsler, "Pseudo-Kronecker expressions for symmetric functions". *IEEE Trans. Comp*. Vol. 48(8), Sept. 1999, pp. 987-990.

[12] IWLS 2022 Programming Contest, https://github.com/alanminko/iwls2022-ls-contest

[13] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism", *IEEE Trans. CAD*, 2020, Vol. 39(4), pp. 871-884.

[14] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification", *IEEE TCAD*, Vol. 21(12), Dec. 2002, pp. 1377-1394

[15] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[16] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*, pp. 15-22.

[17] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification". ERL Technical Report, EECS Dept., UC Berkeley, March 2005.

[18] Y. Miyasaka, A. Mishchenko, J. Wawrzynek, and N. J. Fraser, "Synthesizing practical Boolean functions using truth tables", *Proc. IWLS'22*.

[19] M. A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, and I. Schafer, "Multi-level logic synthesis based on Kronecker Decision Diagrams and Boolean Ternary Decision Diagrams for incompletely specified functions", *VLSI Design*, 1995, Vol. 3, Nos. 3-4, pp. 301-313.

Table 1. The number of two-input nodes in the minimized circuits produced by different algorithms.

Function	Ins	Outs	Shannon (AIG)	Three expansions (XAIG)	Three expansions and variable orders (XAIG)	Our result after post-processing (AIG)	The best result from IWLS'22 contest (AIG)
ex00	6	1	41	29	23	25	23
ex01	6	1	42	28	24	31	27
ex02	8	1	136	96	83	99	88
ex03	8	1	41	21	19	27	24
ex08	8	8	787	588	554	676	544
ex09	8	8	795	579	552	662	555
ex10	5	1	18	14	14	10	10
ex11	7	1	36	24	24	20	20
ex12	9	1	60	45	45	30	30
ex16	5	5	30	33	33	18	18
ex17	6	6	45	53	53	24	24
ex18	7	7	63	76	76	34	32
ex19	8	8	84	108	108	44	38
ex20	9	9	108	140	140	55	56
ex28	7	10	134	65	50	39	39
ex29	9	1	61	51	51	37	39
ex31	9	18	2188	1751	1571	1671	1351
ex33	5	28	178	154	131	84	77
ex34	9	5	432	187	107	46	46
ex35	7	2	34	22	19	17	15
ex37	8	63	618	533	336	147	147
ex38	8	7	98	51	47	31	28
ex41	5	3	37	17	17	17	17
ex42	7	3	75	31	31	28	28
ex43	8	4	100	42	42	37	37
ex46	5	8	74	39	34	34	32
ex49	7	10	189	106	52	39	39
ex50	8	2	62	22	12	18	18
ex51	8	2	91	21	19	28	29
ex52	8	2	36	27	22	19	19
ex53	8	2	117	71	51	42	40
ex54	8	2	25	13	13	13	12
ex55	8	7	292	189	157	148	156
Total			**7127**	**5226**	**4510**	**4250**	**3658**