Minimal Circuits for Boolean Functions of Few Variables

Blair Seidler
blair@math.rutgers.edu

Rutgers University, New Brunswick

Rutgers Experimental Mathematics Seminar
April 7, 2022

Motivation

We had several motivations for this project:

Boolean functions are important and useful objects.

It is often useful to hit small cases with a hammer to help us understand the bigger picture.

We thought that a catalog of circuits for small Boolean functions should exist, but it seemed not to.

We wanted to build a set of tools for future projects exploring Boolean functions.

General Idea

- Create an empty catalog with all n variable Boolean functions, each of which has "NF" as its minimal circuit.
- Set the number of gates, starting with $g = 0$.
- Generate all n-variable g-gate circuits.
- Figure out what function each circuit computes. If that function does not have a circuit assigned, assign this one.
- If there are still any functions without a circuit, increase g by one and repeat.

Boolean Functions

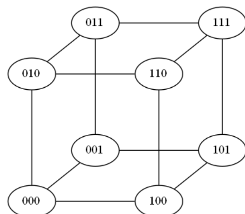
Let $K = \{false, true\}$. Then $f : K^n \to K$ is a Boolean function.

In practice, K can be any two-element set. The most common choice is $K = \{0, 1\}$. For reasons which will (hopefully) become clear in a moment, we use $K = \{-1, 1\}$ in our Maple implementation.

RUTGERS UNIVERSITY | NEW BRUNSWICK

Why use $K = \{-1, 1\}$?

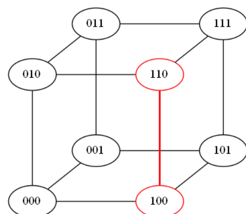
The reason that we use $\{-1, 1\}$ is for ease of referencing subcubes.



This is the standard Hamming cube for K^3

Why use $K = \{-1, 1\}$?

The reason that we use $\{-1, 1\}$ is for ease of referencing subcubes.



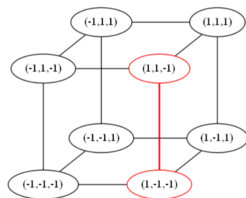
This is the standard Hamming cube for K^3

In much of the literature, the 1-dimensional subcube highlighted in red is written as $(1, *, 0)$, $(1, -, 0)$ or $(1, B, 0)$.

RUTGERS UNIVERSITY | NEW BRUNSWICK

Why use $K = \{-1, 1\}$?

The reason that we use $\{-1, 1\}$ is for ease of referencing subcubes.

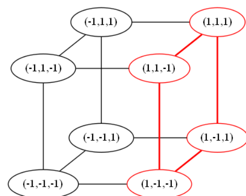


If we instead label this cube with $K = \{-1, 1\}$

The 1-dimensional subcube highlighted in red is written as $(1, 0, -1)$, which we represent in Maple as $[1, 0, -1]$.

Why use $K = \{-1, 1\}$?

The reason that we use $\{-1, 1\}$ is for ease of referencing subcubes.

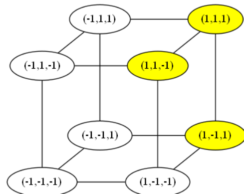


If we instead label this cube with $K = \{-1, 1\}$

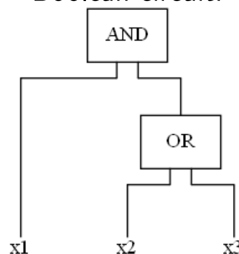
We can write the 2-dimensional subcube highlighted in red as $(1, 0, 0)$, which we represent in Maple as $[1, 0, 0]$.

Representing Boolean Functions

True points in Hamming cube:



Boolean circuit:



Set of true points: $\{[1,-1,1],[1,1,-1],[1,1,1]\}$

DNF: $x_1\overline{x_2}x_3 \vee x_1x_2\overline{x_3} \vee x_1x_2x_3$

Reduced expression: $x_1 \wedge (x_2 \vee x_3)$

Representing Boolean Functions

Within the Maple code, we usually represent functions as a set of true points. The motivation for doing so is that evaluating a function on a given input vector is just a simple membership check.

Sometimes we convert functions into integers in the range 0 to $2^{2^n} - 1$ for the sake of compactness and easy enumeration. More on this later...

Straight-Line Programs - Definition

The next modeling decision to make was how to represent Boolean cicuits in a way that made both generation of cicuits and evaluation of a cicuit on a given input vector relatively simple.

We chose Straight-Line Programs as the model. These are programs with no control structures (loops, branches, etc.). Each line is of the form $y_i =< \textit{expression} >$, and the output of the program is the output of the last line.

We allow a gate to be any of the 10 functions on two variables which depend on both inputs. Other line types handle reading the input and special cases for degenerate functions.

Straight-Line Programs - Gate Types

Line format	Function	Context
$[i]$	Sets $y_i = x_i$	Appears on line i
$[i]$	Outputs y_i $(= x_i)$	After line i (only for 0-gate fns)
$[NOT, i]$	Outputs $\neg y_i$ $(= \neg x_i)$	After line i (only for 0-gate fns)
$[TRUE]$	Outputs 1	Last line of constant function
$[FALSE]$	Outputs -1	Last line of constant function

RUTGERS
UNIVERSITY | NEW BRUNSWICK

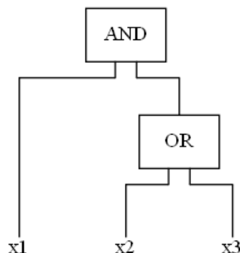
Straight-Line Programs - Gate Types

Line format	Function	Context
$[1, i, j]$	Sets $y_k = y_i \wedge y_j$	On line k with $i, j < k$
$[2, i, j]$	Sets $y_k = y_i \wedge \neg y_j$	On line k with $i, j < k$
$[3, i, j]$	Sets $y_k = \neg y_i \wedge y_j$	On line k with $i, j < k$
$[4, i, j]$	Sets $y_k = y_i \oplus y_j$	On line k with $i, j < k$
$[5, i, j]$	Sets $y_k = y_i \vee y_j$	On line k with $i, j < k$
$[6, i, j]$	Sets $y_k = \neg y_i \wedge \neg y_j$	On line k with $i, j < k$
$[7, i, j]$	Sets $y_k = y_i \equiv y_j$	On line k with $i, j < k$
$[8, i, j]$	Sets $y_k = y_i \vee \neg y_j$	On line k with $i, j < k$
$[9, i, j]$	Sets $y_k = \neg y_i \vee y_j$	On line k with $i, j < k$
$[10, i, j]$	Sets $y_k = \neg y_i \vee \neg y_j$	On line k with $i, j < k$

RUTGERS UNIVERSITY | NEW BRUNSWICK

Straight-Line Programs - Example

So, how would we construct a straight-line program for our earlier circuit example on input $[x_1, x_2, x_3]$?



Straight-line program:

$y_1 = x_1$	
$y_2 = x_2$	Read Inputs
$y_3 = x_3$	
$y_4 = y_2 \vee y_3$	OR gate
$y_5 = y_1 \wedge y_4$	AND gate

In Maple, we write this program as: $[[1], [2], [3], [5, 2, 3], [1, 1, 4]]$

Straight-Line Programs - Number of Circuits

So how big is the haystack? For example, with 4 variables and 7 gates, how many syntactically valid SLP's are there?

For each gate, we have 10 choices for the gate type, and each input can be chosen from every line number smaller than the current one, so we have:

$$\prod_{i=4}^{10} 10i^2 \approx 3.66 \times 10^{18} \text{ valid programs.}$$

Obviously, we will need to reduce the size of the haystack to have any hope at searching through it. First, we will take a look at the number of needles we need to find.

Big Equivalence Classes

We would like some way to find equivalence classes of functions such that once we have a circuit for one member of the class, we have some algorithmic way to produce circuits with the same number of gates for any other member of the class.

There are several ways to accomplish this, but we use the scheme from Tilman Piesk's webpage "Equivalence classes of Boolean functions".

`https://en.wikiversity.org/wiki/Equivalence_classes_of_Boolean_functions`

Big Equivalence Classes

Piesk describes two different types of equivalence classes:

1. A small equivalence class is a set of Boolean functions which are equivalent under the negation of some subset of the input variables (e.g. if $f(x_1, x_2, x_3) = g(\neg x_1, x_2, \neg x_3)$ for all $\mathbf{x} \in K^3$, then f and g are in the same SEC.)
2. A big equivalence class is a set of Boolean functions which are equivalent under some signed permutation of the input variables. (e.g. if $f(x_1, x_2, x_3) = g(\neg x_2, x_3, \neg x_1)$ for all $\mathbf{x} \in K^3$, then f and g are in the same BEC.)

RUTGERS UNIVERSITY | NEW BRUNSWICK

Big Equivalence Classes

We use BEC's to reduce the number of functions we need to consider. The number of Boolean functions on n variables is $\{4, 16, 256, 65536, 4294967296\}$ for $1 \leq n \leq 5$.

The number of BEC's (in OEIS sequence A000616 and confirmed by our code) is $\{3, 6, 22, 402, 1228158\}$ for $1 \leq n \leq 5$.

Searching for over a million 5 variable functions seems out of range for brute-force, but for the 4 variable functions 402 is a much more manageable number than $65, 536$.

RUTGERS UNIVERSITY | NEW BRUNSWICK

Big Equivalence Classes

Once we have a circuit for one member of a BEC, it is relatively simple to convert that circuit to accept a different function of the BEC given the signed permutation which maps one to the other. We leave the structure of the gates intact.

If the variables are permuted by $\sigma : [n] \to [n]$, we just switch any gate input $i \in [n]$ to $\sigma(i)$.

If a variable is negated by the signed permutation, we just change the type of any gate to which it is an input (e.g. if an XOR (type 4) gate has one of its inputs negated, change the gate type to EQUIV).

Assigning Integers to Boolean Functions

In order to take advantage of BEC's, we need a way to choose a canonical representative of each class. There are 2^{2^n} functions on n variables, so numbering them from 0 to $2^{2^n} - 1$ seems natural.

We again follow Piesk, using his numbering scheme and choosing the lowest numbered element of each BEC as the canonical representative. These representatives are listed in OEIS sequence A227723. The next slide shows the numbering scheme for 2 variable functions.

RUTGERS UNIVERSITY | NEW BRUNSWICK

Assigning Integers to Boolean Functions

Input	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
$\{-1,-1\}$	-1	-1	-1	-1	-1	-1	-1	-1
$\{-1,1\}$	-1	-1	-1	-1	1	1	1	1
$\{1,-1\}$	-1	-1	1	1	-1	-1	1	1
$\{1,1\}$	-1	1	-1	1	-1	1	-1	1

Input	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
$\{-1,-1\}$	1	1	1	1	1	1	1	1
$\{-1,1\}$	-1	-1	-1	-1	1	1	1	1
$\{1,-1\}$	-1	-1	1	1	-1	-1	1	1
$\{1,1\}$	-1	1	-1	1	-1	1	-1	1

RUTGERS UNIVERSITY | NEW BRUNSWICK

Reducing Number of SLP's

The most obvious optimization is to restrict all gates to be of the form $[g, i, j]$ with $i < j$.

If $i = j$, the gate output is either constant, y_i, or $\neg y_i$. Any such gate can be eliminated, meaning that the Boolean function could be computed by a circuit with fewer gates.

If $i > j$, we can reverse the inputs and change the gate type (by swapping types $2 \Leftrightarrow 3$ or $8 \Leftrightarrow 9$) if the gate is not symmetric with respect to its inputs.

Reducing Number of SLP's

The second, slightly less obvious simplification is to eliminate circuits and subcircuits which are mirror images of each other. If the final gate G (which produces the overall output of the circuit) has two inputs representing subcircuits A and B, then there is another circuit whose final gate has B as the first input and A as its second which is functionally identical.

We accomplish this in our Maple code by limiting the recursive construction of circuits. When we are building a circuit with g gates, we only allow the first input to be a subcircuit of between 0 and $\lfloor (g-1)/2 \rfloor$ gates.

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Reducing Number of SLP's

We realized one other (admittedly minor) optimization by restricting the inputs to any gate which has zero gates in exactly one of its subcircuits. The input corresponding to the zero-gate subcircuit is not allowed to match either input to the gate producing the other input. If we permitted this situation, which we can think of as having the child of a gate match a grandchild of that gate, we would be able to merge the two gates into one (which we leave as an exercise).

A Bridge Too Far

Our initial implementation went too far down this path in one respect. When one or both of a gate's subcircuits had zero gates, we only allowed those inputs to be between 1 and n.

Why is this a problem? This has the effect of restricting the fan-out of gates to 1, meaning that we cannot reuse the output of a gate later. Initially, we did not think this would matter for small circuits, but it does.

A Bridge Too Far

While tracking down references, we found a 2006 cryptography paper by Markku-Juhani Olavi Saarinen. It included the numbers of 4 variable functions with each circuit complexity. Assuming those numbers to be accurate, some of the functions we identified as complexity 7 are actually complexity 6.

In theory, fixing this was easy. All we needed to do was allow 0-gate subcircuits to be any number lower than the current line number. In practice, the number of circuits increased to an uncomfortable number.

One Last Tweak

After allowing for gate reuse, the number of 4 variable, 6 gate circuits is now about 42 billion, and the number of 7 gate circuits is about 5.4 trillion.

In order to try and process 42 billion circuits without having them all in memory at once, we decided to split the work. We now generate all possible shapes of the circuit with g gates, then fill in the gate types separately. For the 6 gate case, this means storing $42,336$ "skeleton" circuits and then running through the million possible gate type assignments for each in turn.

Circuit Complexity by BEC's

n	0g	1g	2g	3g	4g	5g	6g	7g	Total
1	3								3
2	3	3							6
3	3	3	8	5	3				22
4	3	3	8	34	59	139	130$*$	26$*$	402

The two $*$'ed entries in the table are probably incorrect. The preliminary results do not agree with Saarinen's enumeration of 4 variable functions. We have code running which will presumably find the additional 6-gate BEC's when it finishes.

Circuit Complexity by Functions

n	0g	1g	2g	3g	4g	5g	6g	7g	Total
1	4								4
2	6	10							16
3	8	30	114	80	24				256
4	10	60	456	2474	10624	24184	24784$*$	2944$*$	65536

The two $*$'ed entries in the table are probably incorrect. The preliminary results do not agree with Saarinen's enumeration of 4 variable functions. The first two columns on that line also disagree with Saarinen, but that is merely a difference in definitions. Saarinen defines "NOT x_1" as a gate, and we do not.

Which Functions Are Hardest?

In the 3-variable case, there are 3 BEC's with the maximum complexity of 4 gates. The representative functions of these classes are 22, 23, and 107. Function 22 is true when exactly two of the input variables are true. Function 23 is the majority function (at least two inputs are true). Function 107 is true when exactly one input is true OR its first two inputs are both true.

Taking the majority function as an example, the particular SLP our algorithm found to compute this function is $[[1],[2],[3],[4,1,2],[4,1,3],[1,4,5],[4,1,6]]$. In more traditional notation, this circuit is $x_1 \oplus ((x_1 \oplus x_2) \wedge (x_1 \oplus x_3))$. There are certainly other ways to compute the majority function with 4 gates, but it cannot be computed with 3 or fewer gates.

Monotone Functions

We next turned our attention to monotone functions.

Definition: Let f be a Boolean function on K^n, and let $k \in \{1, 2, \ldots, n\}$. We say that f is positive (respectively, negative) in the variable x_k if $f_{|x_k=-1} \leq f_{|x_k=1}$ (respectively $f_{|x_k=-1} \geq f_{|x_k=1}$). We say that f is monotone in x_k if f is either positive or negative in x_k.

Definition: A Boolean function is positive (respectively, negative) if it is positive (respectively, negative) in each of its variables. The function is monotone if it is monotone in each of its variables.

RUTGERS
UNIVERSITY | NEW BRUNSWICK

My first OEIS Sequence!

During this project, I was able to submit my first sequence (A349743) to the OEIS. It is the subsequence of A227723 containing representatives of the big equivalence classes representing functions which are monotone in each of their variables.

Because of the ordering of functions in the original sequence, the functions represented are actually positive functions (i.e., positive in each of their variables). One enormous advantage of these functions all being positive is that we only need AND and OR gates (types 1 and 5) to represent them, so the exponential growth of the number of circuits has a factor of 2^g instead of 10^g from choice of gate types.

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Advantages of Monotone Functions

In addition to the reduction in gate types required, there are also many fewer BEC's of monotone functions. Specifically for $n = 5$, there are $1,228,158$ BEC's of which only 210 represent monotone functions.

The number of monotone BEC's for n variables is described by the Dedekind numbers (OEIS sequence A003182). Why the Dedekind numbers? Because there is a bijection between antichains and positive Boolean functions. One could specify a positive function by choosing any antichain in the Hamming cube and letting the function's true points be the locus of points in K^n which are members of the antichain with a (possibly empty) subset of the -1's changed to 1's.

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Circuit Complexity by BEC's

n	0g	1g	2g	3g	4g	5g	6g	7g	Total
1	3								3
2	3	2							5
3	3	2	4		1				10
4	3	2	4	10	2	6	1	2	30
5	3	2	4	10	26	16	42	35	...
n	**8g**	**9g**	**10g**	**11g**	**12g**	**13g**			**Total**
5	44	18	3	6		1			210

Again, these results need to be confirmed with gate reuse available.

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Thank you!

Full results and a paper can be found on my website:

https://sites.math.rutgers.edu/~bas312/research.html

The paper and results will be updated when the new data runs are complete and I have had time to analyze the output.

Thank you to Dr. Z for all of his support, and to all of you for being here!

RUTGERS UNIVERSITY | NEW BRUNSWICK