MINIMAL CIRCUITS FOR BOOLEAN FUNCTIONS OF FEW VARIABLES

BLAIR A. SEIDLER

Abstract. Using enumerative techniques, we produce minimum-size circuits for every Boolean function of four or fewer variables and every monotone Boolean function of five variables.

1. Introduction

"Boolean functions, meaning $\{0,1\}$-valued functions of a finite number of $\{0,1\}$-valued variables, are among the most fundamental objects investigated in pure and applied mathematics." [CH11]

1.1. **Motivation.** Mathematicians (including theoretical computer scientists) have spent a great deal of time and effort considering the asymptotic circuit complexity of Boolean functions as the number of variables increases. One desirable result of such explorations would be to find a class of functions representing a language in **NP** for which the asymptotic circuit complexity is superpolynomial. This discovery would prove that $\mathbf{P} \neq \mathbf{NP}$ and resolve perhaps the most perplexing open problem in theoretical computer science.

While this is a worthy endeavor, the result has proven to be elusive. Paul [Pau77] and Blum [Blu84] used an excellent gate-elimination argument to achieve lower bounds for carefully constructed functions. They used similar inductive arguments in which each stage of the induction fixed the value of a single variable in order to eliminate some number of gates. But these heroic constructions were only able to produce a linear lower bound ($2.5n - O(1)$ for Paul and $3n - O(1)$ for Blum). It has recently been shown that gate-elimination can never produce a superlinear bound [GHKK18].

We have chosen to look at Boolean functions of a small number of variables instead. One of the attractions of the small, finite cases is that we should be able to understand them completely. There are only $65,536$ Boolean functions of four variables, which makes it practical to enumerate them on any modern computer. Our

Date: March 20, 2022.

primary objective is to find the circuit complexity of every Boolean function of four or fewer variables.

1.2. **Definition and Representation of Boolean Functions.**

Definition 1. Let $K = \{-1, 1\}$. A *Boolean function* of n variables is a function $f : K^n \to K$.

We deviate from the traditional notation $K = \{0, 1\}$ in part because the labels are irrelevant. The two elements of K are abstract labels, the former representing *false* and the latter representing *true*. In practice, these labels were chosen early in this project to simplify an implementation of the Quine-McCluskey algorithm [WVQ52] for minimizing Boolean functions, also known as "the method of prime implicants." In this implementation, a zero in a particular position in a prime implicant indicates that the corresponding variable can be either true or false.

We represent Boolean functions in several ways in our code and its output depending on the context. The simplest form is as the set $F = \{v \in K^n : f(v) = 1\}$. This is the set of "true points", easiest to visualize as the subset of the Hamming cube at which the function evaluates to *true*. One advantage of this representation is that it provides both a simple way to evaluate the function at a point v (by checking the condition $v \in F$). It also confirms that the number of Boolean functions of n variables is 2^{2^n}, the size of the power set of K^n.

The second representation is as a set of vectors in $\{-1, 0, 1\}^n$. This representation is used in our implementation of the Quine-McCluskey algorithm mentioned earlier in this section. As an example, the two functions

$$\{\{1, 0, -1, 0\}\} \quad \text{and} \quad \{\{1, -1, -1, -1\}, \{1, -1, -1, 1\}, \{1, 1, -1, -1\}, \{1, 1, -1, 1\}\}$$

are equivalent. Several of the functions in our code support this representation of functions. For example, the function *EvalSLPn*, which evaluates a straight-line program on a given input, supports input vectors containing zeroes. It does so by replacing the first zero in the input vector by each of -1 and 1, making a recursive call on each of those vectors, and returning *true* if either of those assignments returns *true*.

The third representation, used only internally in our Maple code, is functionally equivalent to the first. Each vector in the set of true points is converted to an integer between 1 and 2^n by considering the vector as a binary number (using the traditional 0 for *false*) and adding one. The motivitation for this representation is efficiency in permuting and negating the variables. Arrays for the permutation and negation of variables can be initialized and stored, and then each member of the set

can be shifted quickly using these arrays without having to treat each bit separately. We add one to the binary values for the mundane technical reason that Maple array indices are constrained to start at one. Yes, this constraint makes us long for the more civilized arrays of the C programming language which have indices starting at zero as nature intended.

The final representation of Boolean functions is as a single integer. This is the most compact representation, and it was implemented to be consistent with OEIS sequence A227723 [OEISe]. As shown in Table 1, the input strings are arranged from least to greatest when interpreted as binary numbers. The number of the function is assigned by reading down the column of truth values, and again interpreting as a binary number.

TABLE 1. Function numbers for Boolean functions of 2 variables.

Input	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	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	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

1.3. **Equivalence Classes.** It is often useful to think about Boolean functions in terms of equivalence classes, particularly when designing Boolean circuits to compute them. Tilman Piesk [TP20] proposed Small Equivalence Classes and Big Equivalence classes as the names of particular groupings of Boolean functions.

A Small Equivalence Class (SEC) is a group of functions which can be expressed as one another by negating some subset of the input variables. If $f(x_1, x_2, x_3) = g(\neg x_1, x_2, \neg x_3), \forall(x_1, x_2, x_3)$, then f and g are members of the same SEC. OEIS sequence A000231 [OEISa] counts the number of SEC's for n variables. We say that the *representative* of a SEC is the lowest numbered function in the SEC using the numbering scheme in section 1.2 and table 1. OEIS sequence A227722 [OEISd] lists the first 10000 functions which are SEC representatives. Some of our early work on this project investigated SEC's, but the following concept has proven much more efficient for our needs.

A Big Equivalence Class (BEC) is a group of functions which can be expressed as one another by negating and permuting some subset of the input variables. If $f(x_1, x_2, x_3) = g(\neg x_3, x_1, \neg x_2), \forall(x_1, x_2, x_3)$, then f and g are members of the same BEC. OEIS sequence A000616 [OEISb] counts the number of BEC's for n variables.

We say that the representative of a BEC is the lowest numbered function in the BEC using the numbering scheme in section 1.2 and table 1. OEIS sequence A227723 [OEISe] lists the first 10000 functions which are BEC representatives.

Any two functions in a single BEC can be computed by Boolean functions with the same number of gates. In order to permute variables, we need only replace each input to a gate by its image variable under the permutation. To negate a variable, we instead change the type of gate as discussed in section 1.4 below. Since we are interested primarily in finding the circuit complexity of small Boolean functions, it will suffice to find circuits which compute the representative function of each BEC. Why is this useful? Because there are $2^{2^4} = 65536$ Boolean functions of 4 variables, but there are only 402 Big Equivalence Classes. Since one of our primary objectives was to create a catalog of a minimal circuit for every Boolean function of four variables, this reduction in the size of the catalog was critical both the the efficiency of our search and the manageability of the results.

1.4. **Straight-Line Programs.** Straight-line programs are a method of implementing Boolean circuits. Each line of the program has a Boolean output associated with it, possibly dependent on the outputs of previous lines. There are no loops, if-then statements, or other control statements in a straight-line program. The input to a straight-line program is the vector $(x_1, \ldots, x_n) \in K^n$. We denote the output of line i as y_i. For an n-variable function, the first n lines are always $[1], [2], \ldots, [n]$, which sets each y_i equal to the corresponding input x_i. Program lines after the first n represent gates which take the outputs of previous lines as inputs. Table 2 lists the various line formats and the contexts in which those formats are used.

These programs are represented as lists of lists in Maple. The program $[[1], [2], [3], [2]]$ takes an input vector (x_1, x_2, x_3) and outputs the value of x_2, ignoring the other variables. The program $[[1], [2], [3], [1, 1, 2], [5, 3, 4]]$ outputs $x_3 \vee (x_1 \wedge x_2)$.

Gate types 1 through 10 represent the ten Boolean functions of two variables which depend on both of their inputs. The other six functions of two variables are the two constant functions, the two functions whose outputs are equal to one of the input variables, and the two functions whose outputs are the negation of one of the input variables. Gates representing these six degenerate functions are only used to represent zero-gate circuits and will never be used in any circuit with one or more non-degenerate gates.

As mentioned in the previous section, we need to demonstrate that the SLP representation of a cicuit computing a BEC representative function can be modified to

TABLE 2. Straight-line program line formats.

Line format	Function	Context
$[i]$	Sets $y_i = x_i$, where x_i is ith bit of input	Appears on line i
$[i]$	Outputs y_i $(= x_i)$	After line i (only for 0-gate functions)
$[NOT, i]$	Outputs $\neg y_i$ $(= \neg x_i)$	After line i (only for 0-gate functions)
$[TRUE]$	Outputs 1	Last line of constant function
$[FALSE]$	Outputs -1	Last line of constant function
$[1, i, j]$	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 $i, j < k$

compute any other member of that BEC without changing the number of gates. If f is a BEC representative and g is some other member of that BEC, we proceed as follows. Let $\sigma : [n] \to [n]$ be a permutation of the variables and $\nu : [n] \to \{-1, 1\}$ be a function where the variables such that $\nu(x_i) = -1$ are negated. By the definition of BEC's, there exist σ, ν such that $g(x_1, \ldots, x_n) = f(\nu(1)x_{\sigma(1)}, \ldots, \nu(n)x_{\sigma(n)})$ for all input vectors $(x_1, \ldots, x_n)$.

We can now modify the circuit for f to compute g. First, we take every input i to a gate which is one of the input variables (i.e. a number in $[n]$), and replace that input with $\sigma(i)$. We now proceed systematically through every j such that $\nu(j) = -1$. Wherever j is an input to a type 4 or 7 gate (the XOR-type gates), we switch the gate type to the other one. Wherever j is the first input to a gate of types $1, 2, 3, 5, 6, 8, 9, 10$, we change the gate to type $3, 6, 1, 9, 2, 10, 5, 8$ respectively. Wherever j is the second input to a gate of types $1, 2, 3, 5, 6, 8, 9, 10$, we change the gate to type $2, 1, 6, 8, 3, 5, 10, 9$ respectively. By making these modifications for each j in turn, we may end up switching the type of a particular gate twice. If both inputs to an XOR-type gate are negated, we will in fact end up with the original gate type. But this new circuit will compute g using the same number and structure of gates as the original circuit for f.

2. General Functions of Four or Fewer Variables

In this section, we describe the process through which we produce a catalog of minimal circuits for every Boolean function of up to four variables. The resultant catalogs contain a (not necessarily unique) minimal SLP for the representative function of each Big Equivalence Class. As described above, these SLP's could be modified to produce an SLP for any Boolean function in the BEC by finding the BEC representative, determining the permutation and set of negations mapping the functions to each other, and applying the above algorithm to transform the circuit.

2.1. **Methodology.**

We first wrote procedures which created a list of the BEC representative functions of n variables. This admittedly inelegant procedure loops through function numbers $0 \dots 2^{2^n} - 1$ using the previously discussed function numbering convention. The function is converted into our canonical format and then checked to see if it is equivalent to any previous function under signed permutation of the input variables. If not, the function is added to the list of BEC representative functions. When the process completes, the functions are written to a catalog file with each function designated "Not Found".

After the catalog has been created, we use a procedure which generates a set of SLP's with a given number of variables and gates. We then check each SLP to see if it computes a function for which we do not have an SLP in the catalog. If it does, we associate that SLP to the function in our catalog and remove the function from the list of functions we are seeking. By running this routine for n variables and $0, 1, 2, \dots$ gates, we eventually complete the catalog of n-variable functions.

A central issue which needed to be addressed was which SLP's to generate in such a procedure. We would like to generate all syntactically valid SLP's for n variables and g gates to be guarantee that we are checking every possible circuit of a given size. But even for the relatively innocuous case of $n = 4$ and $g = 7$, there are $10^7 \prod_{i=4}^{10} i^2 \approx 3.66 \times 10^{18}$ syntactically valid SLP's. We therefore needed to find a subset of those SLP's which a) was small enough that we could generate it practically, b) was complete in the sense that if there existed an SLP in the entire set for a given f there would be one in the subset, and c) we knew how to find.

The most obvious optimization is to restrict all gates to be of the form $[g, i, j]$ with $i < j$. We can safely do this because if $i = j$, the gate output is either constant (for gate types $2, 3, 4, 7, 8, 9$), the input value y_i (for gate types 1 and 5), or the negation of y_i (for gate types 6 and 10). In fact, 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.

The second, slightly less obvious simplification is to eliminate circuits and subcircuits which are mirror images of each other. The justification for doing this is that 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. Swapping the inputs may require changing the gate type of G (by swapping types $2 \Leftrightarrow 3$ or $8 \Leftrightarrow 9$) if the gate is not symmetric with respect to its inputs. 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.

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 to the reader).

We were, however, somewhat overzealous with our original attempt at these reductions. In early versions of our code, any gate which had a zero-gate subcicuit as its left input was given one of the x_i as its input. While this is efficient in the sense that it provides for a very significant reduction in the number of SLP's generated, we eventually realized a flaw in this optimization. It does not allow for fan-out of gates. It is possible that the smallest circuit for computing a particular function may use the output of a gate more than once. We therefore modified the code to allow the inputs of any gate which does not have at least one gate in each of its input subcircuits to be any prior line number in the SLP (with the exception noted in the immediately preceding paragraph).

This ability to reuse prior gates dramatically increased the total number of SLP's being generated. In order to limit the impact of the change, we added one other level of pruning. As each recursive call in the chain is producing its set of SLP's, we select only one which computes the same Boolean function when considered as a SLP in its own right. This means that each level of recursion can return at most 2^{2^n} SLP's. This is the version of the code which produced the output on the author's webpage.

After completing this catalog, we realized that there is a potential issue with this last pruning in the case that the optimal circuit for computing a function f uses some

gate G in each subcircuit of its output gate. If there are multiple ways to compute the function represented by the left subcircuit, we may return an equivalent subcircuit which does not contain G. Unfortunately, this means that the last pruning we did is technically invalid. At the time of this writing, we have not yet optimized the code to be able to handle the number of SLP's generated without it in the four variable case. The circuits are still small enough that we have reason to believe that gate reuse is not an issue, but it likely will be for larger numbers of variables.

2.2. **Maple Program.** The full set of Maple code for this project is available on the author's webpage. This is the code from `SLprog.txt` which generates the straight-line programs.

```
AllSLPng:=proc(n,lo,hi) local g,i,j,vars,rvs,rff,slp,slp1,slp2,slpn,SLP,SLPn,S1,S2
option remember:
g:=hi-lo+1:
print("lo",lo,"hi",hi):
#vars:={seq(k,k=1..n)}:    #Changed to allow for reuse of prior gates
vars:={seq(k,k=1..hi-1)}:
if g<1 then print('blorf'): RETURN(FAIL): fi:
prefix:=[seq([i],i in 1..n),seq([FALSE],j in 1..lo-n-1)]:
SLPn:={}:
SLP:={}:
if g=1 then
  for rff from 1 to 10 do
#    for rvs in choose(n,2) do    #Changed to allow for reuse of prior gates
    for rvs in choose(hi-1,2) do
      slp:=[[rff,op(rvs)]]:
      slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
      if not (slpn in SLPn) then
        SLPn:=SLPn union {slpn}:
        SLP:=SLP union {slp}:
      fi:
    od:
  od:
  RETURN(SLP):
fi:
for i from 0 to floor((g-1)/2) do
  if i=0 then
   S1:=AllSLPng(n,lo,hi-1):
    for rff from 1 to 10 do
```

```
      for slp1 in S1 do
        for j in vars minus {op(2..3,slp1[-1])} do
          slp:=[op(slp1),[rff,j,hi-1]]:
          slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
          if not (slpn in SLPn) then
            SLPn:=SLPn union {slpn}:
            SLP:=SLP union {slp}:
          fi:
        od:
      od:
    od:
  else
   S1:=AllSLPng(n,lo,lo+i-1):
   S2:=AllSLPng(n,lo+i,hi-1):
   for rff from 1 to 10 do
     for slp1 in S1 do
       for slp2 in S2 do
        slp:=[op(slp1),op(slp2),[rff,lo+i-1,hi-1]]:
        slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
        if not (slpn in SLPn) then
          SLPn:=SLPn union {slpn}:
          SLP:=SLP union {slp}:
        fi:
       od:
     od:
   od:
  fi:
od:
SLP:
end:
```

These are the procedures from `CanonicalBF.txt` which initialize and then populate the catalog file. For each number of variables n, we run `GenBFCatFileAll(n)` first. We then run `CatFileFindSLPAll(n,g)` successively for $g = 0, 1, 2, \ldots$ until every function has been computed.

```
GenBFCatFileAll:=proc(n) local fn,F,f,T:
fn:=cat("BFCatFileA",n,"var.txt"):
F:=FindCanonBFNP(n):
T := table():
```

```
for f in F do
    T[BoolToInt(n, f)] := f;
od:
FileTools[Text][WriteLine](fn, cat("Functions: ",convert(nops(F),string)," Found:
for f in indices(T,indexorder) do
  FileTools[Text][WriteLine](fn, cat(convert(op(f),string),": ",convert(T[op(f)],s
  FileTools[Text][WriteLine](fn, "NF"):
  FileTools[Text][WriteLine](fn, ""):
od:
FileTools[Text][Close](fn):
fn:
end:


CatFileFindSLPAll:=proc(n,g) local numf,numfd,l1,l2,l3,fn,fnold,A,T,F,f,SLP,slp,sl
fn:=cat("BFCatFileA",n,"var.txt"):
fnold:=cat("BFCatFileA",n,"var-OLD.txt"):
FileTools[Text][Open](fn):
l1:=FileTools[Text][ReadLine](fn):
l2:=sscanf(l1,"%s %d %s %d"):
numf:=l2[2]:
numfd:=l2[4]:
F:={}:
T:=table():
for i from 1 to numf do
  l1:=FileTools[Text][ReadLine](fn):
  if l1=NULL then break: fi:
  l2:=FileTools[Text][ReadLine](fn):
  l3:=FileTools[Text][ReadLine](fn):
  l3:=sscanf(l1,"%d: %s"):
  f:=l3[1]:
  T[f]:=parse(l2):
  if l2="NF" then
    F:=F union {f}:
  fi:
od:
if nops(F)=0 then RETURN(F): fi:
FileTools[Text][Close](fn):
SLP:=AllSLPn(n,g):
print("Have SLP's - entering loop"):
```

```
print(nops(SLP)," - number of SLPs"):
for i from 1 to nops(SLP) while nops(F)>0 do
  slp:=SLP[i]:
  slpf:=BoolToInt(n,SLPntoBool(slp)):
  if T[slpf]=NF then
    F:=F minus {slpf}:
    T[slpf]:=slp:
    numfd:=numfd+1:
  fi:
  if i mod 1000=0 then print(i): fi:
od:
if FileTools[Exists](fnold) then FileTools[Remove](fnold): fi:
FileTools[Rename](fn,fnold):
FileTools[Text][WriteLine](fn, cat("Functions: ",convert(numf,string)," Found: ",convert
for f in indices(T,indexorder) do
    FileTools[Text][WriteLine](fn, cat(convert(op(f),string),": ",convert(IntToBool(n,op
    FileTools[Text][WriteLine](fn, convert(T[op(f)],string)):
  FileTools[Text][WriteLine](fn, ""):
od:
FileTools[Text][Close](fn):
F:
end:
```

2.3. **Results.** By running the above code for $1 \leq n \leq 4$ and starting at $g = 0$, we have compiled catalogs of minimal circuits for all Boolean functions of up to 4 variables. In the catalogs we produced, each entry is the representative function of a Big Equivalence Class. The first line contains the number of the representative function (as in OEIS sequence A227723 [OEISe]) followed by the set of true points for the function. The next line is the encoding of a minimal straight-line program which computes the function. Using the techniques discussed in sections 1.3 and 1.4 above, the circuits for a BEC representative function can be modified to compute any function in the BEC.

The catalog for functions of 3 variables is:

```
Functions: 22 Found: 22
0: {}
[[1], [2], [3], [FALSE]]

1: {[1, 1, 1]}
```

```
[[1], [2], [3], [1, 1, 2], [1, 3, 4]]

3: {[1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 1, 2]]

6: {[1, -1, 1], [1, 1, -1]}
[[1], [2], [3], [4, 2, 3], [1, 1, 4]]

7: {[1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 2, 3], [1, 1, 4]]

15: {[1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1]]

22: {[-1, 1, 1], [1, -1, 1], [1, 1, -1]}
[[1], [2], [3], [5, 1, 2], [3, 3, 4], [4, 1, 5], [4, 2, 6]]

23: {[-1, 1, 1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [4, 1, 2], [4, 1, 3], [1, 4, 5], [4, 1, 6]]

24: {[-1, 1, 1], [1, -1, -1]}
[[1], [2], [3], [4, 1, 2], [4, 1, 3], [1, 4, 5]]

25: {[-1, 1, 1], [1, -1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 1, 2], [3, 3, 4], [4, 2, 5]]

27: {[-1, 1, 1], [1, -1, -1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [4, 1, 2], [1, 3, 4], [4, 1, 5]]

30: {[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1]}
[[1], [2], [3], [1, 2, 3], [4, 1, 4]]

31: {[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 2, 3], [5, 1, 4]]

60: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1]}
[[1], [2], [3], [4, 1, 2]]

61: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, 1]}
[[1], [2], [3], [10, 1, 3], [1, 2, 4], [4, 1, 5]]
```

```
63: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1],
       [1, 1, 1]}
[[1], [2], [3], [5, 1, 2]]

105: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, 1, 1]}
[[1], [2], [3], [4, 1, 2], [4, 3, 4]]

107: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 1, 2], [5, 3, 4], [4, 1, 5], [4, 2, 6]]

111: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, -1, 1], [1, 1, -1],
         [1, 1, 1]}
[[1], [2], [3], [4, 2, 3], [5, 1, 4]]

126: {[-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1],
         [1, 1, -1]}
[[1], [2], [3], [4, 1, 2], [4, 1, 3], [5, 4, 5]]

127: {[-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1],
         [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 1, 2], [5, 3, 4]]

255: {[-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], [2], [3], [TRUE]]
```

The catalogs for functions of 1-variable, 2-variable, and 4-variable functions can be found on the author's web page.

A summary of the circuit complexities for functions of up to 4 variables is shown in tables 3 and 4. As expected, 1-variable functions never require any gates because they are either constant or take the value or negation of the input variable (and we do not consider NOT to be a gate). Because our definition of a gate was any function of two variables which depends on both inputs, the 2-variable functions are similarly unexciting. There are 6 trivial functions ($false$, x_1, $\neg x_1$, x_2, $\neg x_2$, and $true$), and the other 10 functions are each computed by one of the gates we defined.

TABLE 3. Number of BEC's by circuit complexity

# of variables	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	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

TABLE 4. Number of functions by circuit complexity

# of variables	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	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

For 3 or 4 variables, a natural question is which functions have the highest circuit complexity? 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.

In the 4-variable case, there are 26 BEC's which have the maximum circuit complexity of 7 gates. Many of them have a similar flavor to the 3-variable functions requiring 4 gates. A few typical examples are function 278 (exactly three inputs are true), function 279 (at least three inputs are true), and function 6015 (at least two inputs are true). Most of the other functions in this group are more akin to 107 in the 3-variable case. Function 1633 is probably best described as ($\neg x_4$ and exactly two of $\{x_1, x_2, x_3\}$) or ($x_1 \wedge x_4$ and $x_2 \equiv x_3$).

3. Monotone Functions of Five or Fewer Variables

There are two major advantages to restricting our analysis to monotone functions. The first is that there are many fewer monotone Boolean functions (and Big Equivalence Classes thereof) than there are general Boolean functions. The second is that we need fewer types of gates to construct Boolean circuits which compute monotone functions. In practice, the representative of each Big Equivalence Class is a positive function, i.e. changing the value of an input variable from true to false will never change the output from false to true. We can therefore model the circuits for these functions using only AND and OR gates, which are gate types 1 and 5 in our existing representation.

3.1. **Definitions and Notation.** For the most part, we use the same conventions as we do in the general case. We do need a few definitions, which we will take directly from Crama and Hammer [CH11] with only notational modification.

Definition 2. 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.

Here the notation $f \leq g$ for Boolean functions f and g means $f(\overline{x}) = 1 \Rightarrow g(\overline{x}) = 1$ for all $\overline{x} \in K^n$. The notation $f_{|x_k=-1}$ is the restriction of f to input vectors where input x_k is false.

Definition 3. 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.

3.2. **Methodology.** As mentioned above, a useful byproduct of the numbering scheme we are using for Boolean functions is that the lowest numbered function in each monotone BEC is a positive function. We therefore do not need to worry about negating variables in our gates, and we can use only gate types 1 and 5. This mean that while the number of circuits with each number of gates still grows exponentially, the base of the exponential is 2 rather than 10 as it is in the general case.

The other numerical advantage we have is that even with 5 variables, there are only 210 monotone BEC's, so we are hunting for many fewer needles in our proverbial haystack. By comparison, there are over 1.2 million BEC's for general functions of 5 variables, which is why we did not attempt to catalog those.

The methodology for producing this catalog is nearly identical to that for general functions. The most interesting difference (see Maple code in section 3.3) is in the initialization of the catalog with the BEC representative functions. In the general case, we loop through every function and discard it if we have already seen an equivalent function under signed permutation of the variables. That is obviously not feasible for the 2^{2^5} functions of 5 variables.

Instead, we start with the function $\{[1,1,1,1,1]\}$ as our working list. The main loop then pulls a function from the working list, adds it to the list of monotone functions, then calculates all of the functions which have one additional true point formed by negating one value in an existing true point. If that function is already in the main list, already in the working list, or not monotone, it is discarded. Otherwise it is appended to the working list. This allows us to generate the initial blank catalog in a reasonable amount of time.

The only other relevant difference is that the SLP generation procedure only uses AND and OR gates at every level beyond the 0-gate SLP's.

3.3. **Maple Code.**

```
IsMonotone:=proc(n,f) local i,j,v,s,nv:
if nops(f)=0 then RETURN(true): fi:
for v in f do
  s:={}:
  for i from 1 to n do
    if v[i]=-1 then s:=s union {i}: fi:
  od:
  for i in powerset(s) do:
    nv:=v:
    for j in i do nv[j]:=1: od:
    if not (nv in f) then RETURN(false): fi:
  od:
od:
RETURN(true):
end:

FindMonoBFP:=proc(n) local PT,i,j,v,nv,nv2,f,F,ff,W:
F:={{}}: #Adds empty function because it is only monotone function not containing
W:={{[1$n]}}: # Adds the AND of all variables as the seed function in working set
while nops(W)>0 do
  ff:=W[1]:  #pull function from W
```

```
  if not (ff in F) then
    F:=F union {ff}:
    for v in ff do
      for i from 1 to n do
        if v[i]=1 then
          nv:=[op(1..i-1,v),-1,op(i+1..n,v)]:
          f:=ff union {nv}:
          if not (f in F) and IsMonotone(n,f) then W:=W union {f}: fi:
        fi:
      od:
    od:
  fi:
  W:=W minus {ff}:
od:
ff:={}:
for i in F do
  if IsBECrep(n,i) then
    ff:=ff union {i}:
  fi:
od:
ff:
end:
```

3.4. **Results.** By running the above code for $1 \leq n \leq 5$ and starting at $g = 0$, we have compiled catalogs of minimal circuits for all monotone Boolean functions of up to 5 variables. In the catalogs we produced, each entry is the representative function of a Big Equivalence Class. The first line contains the number of the representative function (as in OEIS sequence A349743 [OEISf], submitted to OEIS as part of this project) followed by the set of true points for the function. The next line is the encoding of a minimal straight-line program which computes the function.

As in the general case, these SLP's could be modified to handle any function in the respective BEC. If any variables are negated in a signed permutation, so that the function is monotone without being positive, we would need to introduce gates of types other than 1 or 5. We would still not need either of the XOR-type gates, since any function which depends on the output of such a gate is not monotone (or at least could be written without such a gate).

The catalog for functions of 3 variables is:

```
Functions: 10 Found: 10
0: {}
[[1], [2], [3], [FALSE]]

1: {[1, 1, 1]}
[[1], [2], [3], [1, 1, 2], [1, 3, 4]]

3: {[1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 1, 2]]

7: {[1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 2, 3], [1, 1, 4]]

15: {[1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1]]

23: {[-1, 1, 1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 1, 2], [1, 1, 2], [5, 3, 5], [1, 4, 6]]

31: {[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 2, 3], [5, 1, 4]]

63: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 1, 2]]

127: {[-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1], [
[[1], [2], [3], [5, 1, 2], [5, 3, 4]]

255: {[-1, -1, -1], [-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1],
[[1], [2], [3], [TRUE]]
```

We note that each of these functions has the same number of gates as it did in the catalog of general Boolean functions, although the precise SLP may be different. The catalogs for functions of 1-variable, 2-variable, 4-variable, and 5-variable functions can be found on the author's web page. A summary of the circuit complexities for monotone functions of up to 5 variables is shown in table 5.

TABLE 5. Number of monotone BEC's by circuit complexity

# of variables	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	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	...
# of variables	8-gate	9-gate	10-gate	11-gate	12-gate	13-gate			Total
5	44	18	3	6		1			210

The most surprising result is that among the 5-variable functions, 18290559 requires 13 gates while no other function requires more than 11. Perhaps less surprising is the observation that this is the majority function on five variables (at least three inputs are true). As in the general case, the majority function is always in a BEC with the maximum circuit complexity.

REFERENCES

[Blu84] Norbert Blum, *A Boolean function requiring* $3n$ *network size*, Theoret. Comput. Sci. **28** (1984), no. 3, 337–345.

[CH11] Yves Crama and Peter L. Hammer, *Boolean Functions*, Cambridge University Press, 2011. (cited on page xv).

[GHKK18] Alexander Golovnev, Edward A. Hirsch, Alexander Knop, and Alexander S. Kulikov, *On the limits of gate elimination*, Journal of Computer and System Sciences **96** (2018), 107-119.

[OEISa] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A000231`.

[OEISb] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A000616`.

[OEISc] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A227722`.

[OEISd] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A227722`.

[OEISe] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A227723`.

[OEISf] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A349743`.

[Pau77] Wolfgang J. Paul, *A* $2.5n$*-lower bound on the combinational complexity of Boolean functions*, SIAM J. Comput. **6** (1977), no. 3, 427–443. MR451856

[TP20] Tilman Piesk, *Equivalence Classes of Boolean Functions*, 2020. `https://en.wikiversity.org/wiki/Equivalence_classes_of_Boolean_functions`.

[WVQ52] W. V. Quine, *The Problem of Simplifying Truth Functions*, The American Mathematical Monthly **59** (1952), no. 8, 521–531.

Department of Mathematics, Rutgers University (New Brunswick), Hill Center-Busch Campus, 110 Frelinghuysen Rd., Piscataway, NJ 08854-8019, USA

Email address: `blair@math.rutgers.edu`