CDM

Boolean Decision Diagrams

Klaus Sutner

Carnegie Mellon University

24-bdd 2017/12/15 23:21

1 Boolean Functions

- Clones and Post
- Minimal Automata
- Binary Decision Diagrams

Let us return to the two-element Boolean algebra $\mathbb{B} = \{\mathsf{ff}, \mathsf{tt}\}$.

To lighten notation we will overload and write $\mathbf{2} = \{0, 1\}$ instead.

Definition

A Boolean function of arity n is any function of the form

$$\mathbf{2}^n \to \mathbf{2}$$

Note that we only consider single outputs here, $f : \mathbf{2}^n \to \mathbf{2}$ rather than $f : \mathbf{2}^n \to \mathbf{2}^m$.

Slightly more vexing is the question of whether one should include $n = 0$. We'll fudge things a bit.

Obviously there are 2^{2^n} Boolean functions of arity n.

n	1	2	3	4	5	6	7	8
2^{2^n}	4	16	256	65536	4.3×10^9	1.8×10^{19}	3.4×10^{38}	1.2×10^{77}

$$2^{2^{20}} = 6.7 \times 10^{315652} = \infty$$

..., for all practical intents and purposes. Since there are quite so many, it is a good idea to think about how to describe and construct these functions.

- Combinatorics: $\mathbf{2}^n \to \mathbf{2}$
- Logic: propositional formula
- Circuits: logic gates
- Datatype: BDDs

In a sense, this is all the same–but different perspectives lead to different ideas and very different algorithms.

For example, from the logic perspective it is natural to generate tautologies, a boring idea from the viewpoint of circuits.

The propositional logic angle immediately suggests the following computational problems.

Problem:	**Tautology**
Instance:	A propositional formula φ.
Question:	Is φ a tautology?

Problem:	**Satisfiability**
Instance:	A propositional formula φ.
Question:	Is φ a contingency?

Problem:	**Equivalence**
Instance:	Two propositional formulae φ and ψ.
Question:	Are φ and ψ equivalent?

Alas, we know from complexity theory that these are all hard.

These are all closely connected:

- φ is a tautology iff $\neg\varphi$ fails to be satisfiable
- φ is a tautology iff φ is equivalent to $\top$
- φ and ψ are equivalent iff $\varphi \Leftrightarrow \psi$ is a tautology

Note, though, that some care is necessary to deal with normal forms (which are often required by specific algorithms).

- Boolean Functions

2. Clones and Post

- Minimal Automata

- Binary Decision Diagrams

For small arities n we can easily enumerate all Boolean functions.

Case $n = 0$: All we get is 2 constant functions $\mathbf{0}$ and $\mathbf{1}$.

Case $n = 1$: 2 constants, plus

- $H_{\mathsf{id}} = x$, the identity, and
- $H_{\mathsf{not}}(x) = 1 - x$, negation.

Incidentally, the latter two are reversible.

Case $n = 2$: This is much more interesting: here all the well-known operations from propositional logic appear.

x	y	$H_{\mathsf{and}}(x, y)$	$H_{\mathsf{or}}(x, y)$	$H_{\mathsf{imp}}(x, y)$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

x	y	$H_{\mathsf{equ}}(x, y)$	$H_{\mathsf{xor}}(x, y)$	$H_{\mathsf{nand}}(x, y)$
0	0	1	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

Via composition we can generate all Boolean functions from these.

It is well-known that some Boolean functions can be expressed in terms of others. For example, one can write "exclusive or" $\oplus$ in disjunctive normal form, conjunctive normal, or using only nand $\bar{\wedge}$ or nor $\bar{\vee}$ as follows:

$$
\begin{aligned}
x \oplus y &= (x \wedge \overline{y}) \vee (\overline{x} \wedge y) \\
&= (\overline{x} \vee \overline{y}) \wedge (x \vee y) \\
&= ((x \mathbin{\bar{\vee}} x) \mathbin{\bar{\vee}} (y \mathbin{\bar{\vee}} y)) \mathbin{\bar{\vee}} (x \mathbin{\bar{\vee}} y) \\
&= (x \mathbin{\bar{\wedge}} (x \mathbin{\bar{\wedge}} y)) \mathbin{\bar{\wedge}} ((x \mathbin{\bar{\wedge}} y) \mathbin{\bar{\wedge}} y)
\end{aligned}
$$

A set of Boolean functions is a basis or functionally complete if it produces all Boolean functions by composition.

As it turns out, there are lots of bases.

Theorem (Basis Theorem)

Any Boolean function $f : \mathbf{2}^n \to \mathbf{2}$ *can be obtained by composition from the basic functions (unary) negation and (binary) disjunction.*

Proof.

Consider the truth table for f.

For every truth assignment α such that $f(\boldsymbol{\alpha}) = 1$ define a conjunction $C_\alpha = z_1 \wedge \ldots \wedge z_n$ by $z_i = x_i$ whenever $\alpha(x_i) = 1$, $z_i = \overline{x_i}$ otherwise.

Then $f(\boldsymbol{x}) \equiv C_{\alpha_1} \vee \ldots \vee C_{\alpha_r}$.

Use Morgan's law to eliminate the conjunctions from this disjunctive normal form.

Here are some more bases:

- $\{\land, \neg\}$,
- $\{\Rightarrow, \neg\}$,
- $\{\Rightarrow, \mathbf{0}\}$,
- $\{\lor, \Leftrightarrow, \mathbf{0}\}$
- $\{\bar{\land}\}$
- $\{\bar{\lor}\}$

On the other hand, $\{\lor, \Leftrightarrow\}$ does not work.

One might be tempted to ask when a set of Boolean functions is a basis.

To tackle this problem we need to be a little more careful about what is meant by basis.

Definition (Clones)

Write $\mathbb{BF}$ for all Boolean functions and $\mathbb{BF}_n$ for Boolean functions of arity n.

A family $\mathbb{F}$ of Boolean functions is a clone if it contains all projections $\mathsf{P}^{\mathsf{n}}_{\mathsf{i}}$ and is closed under composition.

Recall that a projection is a simple map

$$\mathsf{P}^{\mathsf{n}}_{\mathsf{i}} : \mathbf{2}^n \to \mathbf{2} \qquad \mathsf{P}^{\mathsf{n}}_{\mathsf{i}}(x_1, \ldots, x_n) = x_i$$

and closure under composition means that

$$f : \mathbf{2}^m \to \mathbf{2} \qquad f(\boldsymbol{x}) = h(g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$$

is in $\mathbb{F}$, given functions $g_i : \mathbf{2}^m \to \mathbf{2}$ for $i = 1, \ldots, n$, and $h : \mathbf{2}^n \to \mathbf{2}$, all in $\mathbb{F}$.

More succinctly, we can write $h \circ (g_1, \ldots, g_n)$.

One can rephrase the definition of a clone into a longer list of simpler conditions (that may be easier to work with). Let $f, g \in \mathbb{F}$ be an arbitrary functions of suitable arity.

The following functions h must also be in $\mathbb{F}$.

monadic identity: P^1_1

cylindrification: $h(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1})$

diagonalization: $h(x_1, \ldots, x_n) = f(x_1, \ldots, x_n, x_n)$

rotation: $h(x_1, \ldots, x_n) = f(x_2, x_3, \ldots, x_n, x_1)$

transposition: $h(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-2}, x_n, x_{n-1})$

composition: $h(x_1, \ldots, x_{n+m}) = f(x_1, \ldots, x_n, g(x_{n+1}, \ldots, x_{n+m}))$

According to rules (2), (3), (4), (5) we can construct h from f via

$$h(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(m)})$$

for any function $\pi : [m] \to [n]$.

Together with the last rule this suffices to get arbitrary compositions.

Exercise

Show that this characterization of clones is correct.

Wild Question: Is there a way to describe all clones?

This may seem utterly hopeless at first, but at least a few simple clones can be dealt with easily.

It is clear that $\mathbb{BF}$ is a clone; we write $\top$ for this clone. At the other end, the smallest clone is the set all projections; we write $\bot$ for this clone.

The collections of all clones form a lattice under set inclusion. Operation $\sqcap$ is just intersection, but $\sqcup$ requires closure under composition.

For any Boolean function f we can define its dual $\widehat{f}$ by

$$\widehat{f}(\boldsymbol{x}) = \overline{f(\overline{\boldsymbol{x}})}$$

For example, $\widehat{\vee} = \wedge$ and $\widehat{\neg} = \neg$.

A function is self-dual if

$$\widehat{f}(\boldsymbol{x}) = f(\boldsymbol{x})$$

Examples of self-dual functions are $\neg$ and projections.

For any clone $\mathbb{F}$ we can define the dual clone by

$$\widehat{\mathbb{F}} = \{ \widehat{f} \mid f \in \mathbb{F} \}$$

So the claim here is that $\widehat{F}$ is again a clone.

Unfortunately, for our trivial clones $\top$ and $\bot$ duality does not buy us anything. But later we will find a nice symmetric structure.

Claim

The collection D of all self-dual Boolean functions is a clone.

To find more interesting examples of clones, consider the following.

Definition

Let f be a Boolean function.

f is 0-preserving if $f(\mathbf{0}) = 0$.

f is 1-preserving if $f(\mathbf{1}) = 1$.

f is bi-preserving if f is 0-preserving and 1-preserving.

For example n-ary disjunction and conjunction are both bi-preserving.

Claim

The collections P_0, P_1 and P of all 0-preserving, 1-preserving and bi-preserving Boolean functions are clones.

For Boolean vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same length write $\boldsymbol{a} \leq \boldsymbol{b}$ iff $a_i \leq b_i$ (the product order on $\mathbf{2}$).

Definition

A Boolean function f is monotone if

$$\boldsymbol{a} \leq \boldsymbol{b} \qquad \text{implies} \qquad f(\boldsymbol{a}) \leq f(\boldsymbol{b})$$

For example n-ary disjunction and conjunction are both monotone.

One can define a topology on $\mathbf{2}^n$ so that these functions are precisely the continuous ones.

Claim

The collection M of all monotone Boolean functions is a clone.

Counting the number of n-ary monotone Boolean functions is not easy, the problem was first introduced by R. Dedekind in 1897, see `https://oeis.org/A000372`.

Here are the first few Dedekind numbers D_n, for arities $n = 0, \ldots, 8$.

$$2, 3, 6, 20, 168, 7581, 7828354, 2414682040998, 56130437228687557907788$$

One can show that D_n is the number of anti-chains in the powerset of $[n]$. For example, there are 6 anti-chains over $[2]$

$$\emptyset, \{\emptyset\}, \{\{1\}\}, \{\{2\}\}, \{\{1,2\}\}, \{\{1\},\{2\}\}$$

And 20 anti-chains over $[3]$:

$$\emptyset, \{\emptyset\}, \{\{1\}\}, \{\{2\}\}, \{\{3\}\}, \{\{1,2\}\}, \{\{1,3\}\}, \{\{2,3\}\}, \{\{1,2,3\}\},$$
$$\{\{1\},\{2\}\}, \{\{1\},\{3\}\}, \{\{1\},\{2,3\}\}, \{\{2\},\{3\}\}, \{\{2\},\{1,3\}\}, \{\{3\},\{1,2\}\},$$
$$\{\{1,2\},\{1,3\}\}, \{\{1,2\},\{2,3\}\}, \{\{1,3\},\{2,3\}\}, \{\{1\},\{2\},\{3\}\},$$
$$\{\{1,2\},\{1,3\},\{2,3\}\}$$

Make sure you understand how the anti-chains correspond to monotone functions: every set in an anti-chain over $[n]$ defines a clause over n Boolean variables $x_1, \ldots, x_n$. Define f to be true on exactly these clauses.

These are functions that are essentially unary: they are formally n-ary but depend on only one argument.

Definition

A Boolean function f is quasi-monadic if there is some index i such that

$$x_i = y_i \qquad \text{implies} \qquad f(\boldsymbol{x}) = f(\boldsymbol{y})$$

Claim

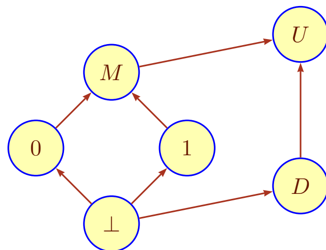
The collection U of all quasi-monadic Boolean functions is a clone.

Quasi-monadic functions are rather simple, so it is not too surprising that we can actually determine all clones containing only quasi-monadic functions: there are exactly 6 of them.

The first 4 contain only monotone functions.

- $\bot$, the trivial clone of projections,
- 0-preserving quasi-monadic functions,
- 1-preserving quasi-monadic functions,
- monotone quasi-monadic functions,
- self-dual quasi-monadic functions,
- all quasi-monadic functions.

It is not hard to see that each of these classes of functions is in fact a clone. The hard part is to make sure that each quasi-monadic clone appears on the the list.



The arrows are precise in the sense that there are no other clones in between.

Exercise

Figure out exactly what these clones look like.

We have mentioned that projections and negation are self-dual. Here is another important example.

Definition (Majority)

The ternary majority function maj is defined by

$$\mathrm{maj}(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$$

Then majority is self-dual:

$$\begin{aligned}
\overline{\mathrm{maj}(\overline{x}, \overline{y}, \overline{z})} &= \overline{(\overline{x} \wedge \overline{y}) \vee (\overline{x} \wedge \overline{z}) \vee (\overline{y} \wedge \overline{z})} \\
&= \overline{(\overline{x} \wedge \overline{y})} \wedge \overline{(\overline{x} \wedge \overline{z})} \wedge \overline{(\overline{y} \wedge \overline{z})} \\
&= (x \vee y) \wedge (x \vee z) \wedge (y \vee z) \\
&= \mathrm{maj}(x, y, z)
\end{aligned}$$

Majority is an important special case of a broader class of functions.

Definition

A threshold function thr^n_m, $0 \leq m \leq n$, is an n-ary Boolean function defined by

$$\mathrm{thr}^n_m(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \#(i \mid x_i = 1) \geq m, \\ 0 & \text{otherwise.} \end{cases}$$

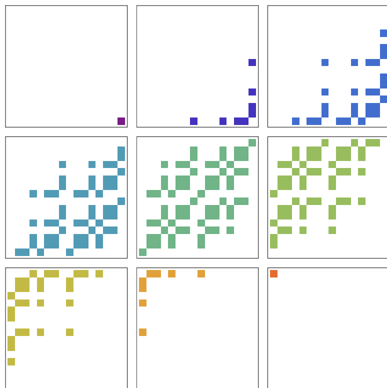
Lots of Boolean functions can be defined in terms of threshold functions.

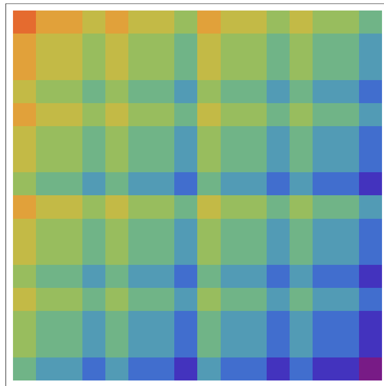
thr^n_0 is the constant tt.

thr^n_1 is n-ary disjunction.

thr^n_n is n-ary conjunction.

$\mathrm{thr}^n_k(\boldsymbol{x}) \wedge \neg\mathrm{thr}^n_{k+1}(\boldsymbol{x})$ is the counting function: "exactly k out of n."





Definition

An affine Boolean function has the form

$$f(\boldsymbol{x}) = a_0 \oplus \bigoplus_i a_i x_i$$

where the $a_i \in \mathbf{2}$ are fixed.

Here $\oplus$ denotes exclusive or (equivalently, addition modulo 2). If $a_0 = 0$ we have a linear Boolean function. Note that these functions are additive:

$$f(\boldsymbol{x} \oplus \boldsymbol{y}) = f(\boldsymbol{x}) \oplus f(\boldsymbol{y})$$

Claim

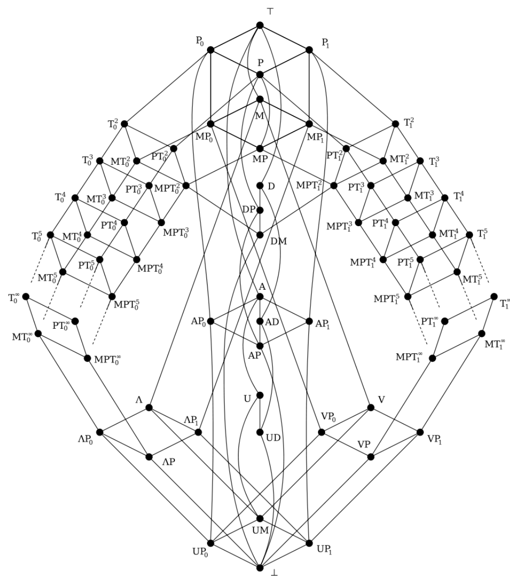
The collection A of all affine functions is a clone.

At this point the literate reader might wonder whether this is the beginning of an infinite story: every time you turn around, a new clone pops up somewhere. On and on and on . . .

As it turns out, the world is actually pretty tame.

The same Post as in Post's problem and Post tag systems.

Post found a classification of **all clones**, published in 1941.


⊤
P0
P1
P
M
MP0
MP1
MP
T0^2
T1^2
T0^3
MT0^2
PT0^2
PT1^2
MT1^2
T1^3
PT0^3
MPT0^2
D
MPT1^2
PT1^3
T0^4
MT0^3
MT1^3
T1^4
DP
PT0^4
MPT0^3
MPT1^3
PT1^4
T0^5
MT0^4
DM
MT1^4
T1^5
PT0^5
MPT0^4
MPT1^4
PT1^5
MT0^5
MT1^5
A
T0^∞
MPT0^5
MPT1^5
T1^∞
AP0
AD
AP1
PT0^∞
PT1^∞
MT0^∞
AP
MT1^∞
MPT0^∞
MPT1^∞
Λ
U
V
ΛP1
VP0
ΛP0
UD
VP1
VP
ΛP
UM
UP0
UP1
⊥

Careful inspection shows that all the clones in the diagram have a finite set of generators: for any clone $\mathcal{C}$ there are finitely many Boolean functions $f_1, \ldots, f_r$ so that the least clone containing all f_i is $\mathcal{C}$.

Here are some examples:

$\top$	$\vee, \neg$
P_0	$\vee, \oplus$
P_1	$\vee, \rightarrow$
P	if-then-else
M	$\vee, \wedge, 0, 1$
D	$\mathrm{maj}, \neg$
A	$\leftrightarrow, 0$
U	$\neg, 0$

Since our diagram contains all clones it follows that all Boolean clones are finitely generated.

However, one can also directly show that Boolean clones must be finitely generated. The proof rests on the following strange property.

Definition

An n-ary Boolean function f is a rejection function if for all $a \in \mathbf{2}$:

$$\#(i \mid x_i = a) \geq n - 1 \qquad \text{implies} \qquad f(\boldsymbol{x}) = a$$

For example, majority is a rejection function.

Lemma

A clone containing a rejection function is finitely generated.

One can generalize the property of being 0-preserving:

Definition

A Boolean function f is m-fold 0-preserving if

$$\boldsymbol{x}_1 \wedge \ldots \wedge \boldsymbol{x}_m = \mathbf{0} \quad \text{implies} \quad f(\boldsymbol{x}_1) \wedge \ldots \wedge f(\boldsymbol{x}_m) = \mathbf{0}$$

Call this class Z_m and let $Z_\infty = \bigcap Z_m$.

Claim

Z_m and Z_∞ are clones.

Note that $Z_m$is properly contained in Z_{m-1}. For $m \geq 2$, Z_m is generated by thr_m^{m+1} and $\neg \to$. As it turns out, $f \in Z_\infty$ iff $f(\boldsymbol{x}) \leq x_i$.

There also are variants by adding monotonicity and/or 1-preservation.

And we can do the same for 1-preserving functions.

- The largest clones other than $\top$ are M, D, A, P_0 and P_1.
- There are eight infinite families of clones, indexed by a simple numerical parameter.
- There are a few "sporadic" clones.

That's it, no other clones exist.

The original proof was supposedly 1000 pages long, the published version is still around 100 pages.

Suppose we have a finite set C of logical connectives.

How hard is Satisfiability if the formula is written in terms of the connectives in C?

Theorem (H. R. Lewis 1979)

Satisfiability for C is $\mathbb{NP}$-complete if the clone generated by C contains Z_∞. Otherwise the problem is in polynomial time.

The proof exploits Post's lattice from above.

There are exactly two functionally complete sets of size 1: NAND and NOR, discovered by Peirce around 1880 and rediscovered by Sheffer in 1913 (Pierce arrow and Sheffer stroke).

It follows from Post's result that a set of Boolean functions is a basis iff for each of the following 5 classes, it contains at least one function that fails to lie in the class:

- monotonic
- affine
- self-dual
- truth-preserving
- false-preserving

- Boolean Functions
- Clones and Post
3. Minimal Automata
- Binary Decision Diagrams

Is there an implementation for Boolean functions that makes it easy to decide equivalence?

Obviously there are complexity theoretic constraints: if the implementation worked perfectly we could easily solve Tautology.

So what we want is a data structure plus algorithms that work well in some interesting cases (but we full-well expect failure in others).

One plausible line of attack is to use some kind of canonical normal form: if we represent the function in normal form equivalence comes down to equality (or perhaps isomorphism).

Suppose we have some Boolean function $f : \mathbf{2}^n \to \mathbf{2}$. In the spirit of disjunctive normal form we can represent f by the collection of all the Boolean vectors $\boldsymbol{a} \in \mathbf{2}^n$ such that $f(\boldsymbol{a}) = 1$.

But then we can think of this collection as a binary language:

$$L_f = \{ \boldsymbol{a} \in \mathbf{2}^n \mid f(\boldsymbol{a}) = 1 \} \subseteq \mathbf{2}^n$$

This language is, of course, finite, so there is a finite state machine that accepts it. In fact, the partial minimal DFA is just a DAG: all words in the language correspond to a fixed-length path from the initial state to the unique final state.

We already know an example: the UnEqual language

$$L_k = \{ uv \in \mathbf{2}^{2k} \mid u \neq v \in \mathbf{2}^k \}$$

It turns out that the state complexity of L_k is $3 \cdot 2^k - k + 2 = \Theta(2^k)$.

This is uncomfortably large since this Boolean function has a very simple description as a propositional formula:

$$\mathsf{UE}(\boldsymbol{x}) = \bigvee x_i \oplus x_{i+k}$$

This formula clearly has size $\Theta(k)$.

The reason our DFA is large, while the formula is small, is that the machine has to contend with input bits x_i and x_{i+k} being far apart.

By contrast, in the formula (rather, the parse tree), they are close together.

This suggests a way to get around this problem. Change the language to

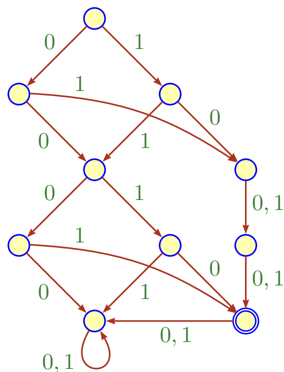
$$K_k = \{ x \in \mathbf{2}^{2k} \mid \exists\, i\, (x_{2i} \neq x_{2i+1}) \}$$

assuming 0-indexing. In other words, the corresponding formula is now

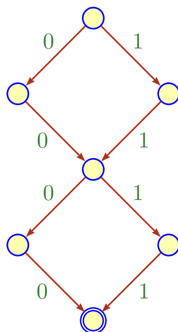
$$\mathsf{UE}'(\boldsymbol{x}) = \bigvee x_{2i} \oplus x_{2i+1}$$

You might worry that these shenanigans won't generalize, and you would be right. But let's ignore this for the time being.

The state complexity of K_k is much smaller than for L_k: $5k$.



The minimal DFA for K_2.



The partial minimal DFA for the negation of the UnEqual function (aka as the Equal function).

Note that complementation here is a bit weird: by flipping final and non-final states in a DFA M we get a DFA M' such that

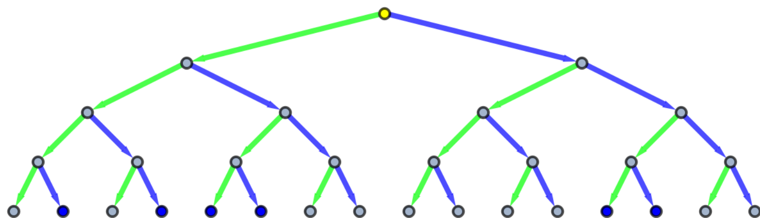
$$\mathcal{L}(M') = \mathbf{2}^\star - \mathcal{L}(M).$$

But that's not really what we want for $\overline{f}$, instead we need M' such that

$$\mathcal{L}(M') = \mathbf{2}^n - \mathcal{L}(M).$$

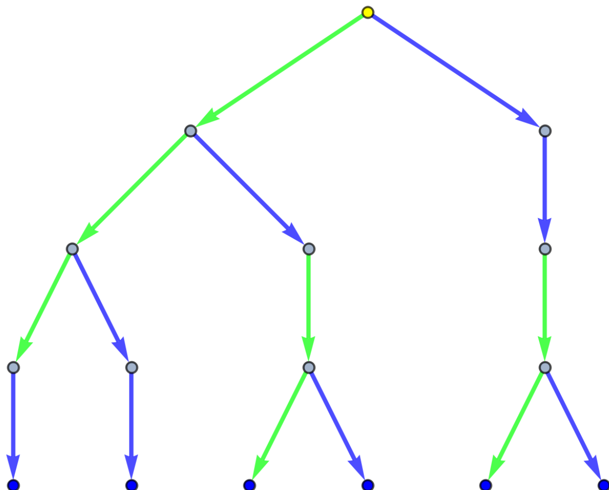
We need to fix things up a bit.

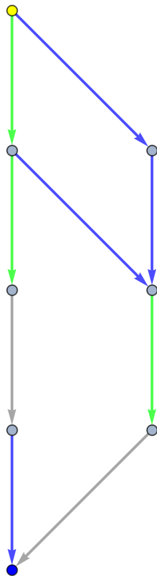
The same holds for product automata for unions and intersections when the variables are not the same. For example, one needs to be careful when constructing the machine for the conjunction of, say, $f(x, z)$ and $g(x, y, z, u)$.



The obvious tree automaton on 31 states for the Boolean function

$$(\neg x_1 \wedge \neg x_2 \wedge x_4) \vee (x_2 \wedge \neg x_3)$$





The good news is this: because of the uniqueness of the minimal DFA, we do have a some kind of canonical normal form. In particular, to check whether two Boolean functions are equivalent, we can check their minimal DFAs for isomorphism. This representation is known as minimal deterministic automaton (MDA).

Still, from the previous comments it looks like this is not quite the right setting.

Here is another nuisance: the DFA for K_k should not have the chain on the right: it should just "output" 1 when we get over there and be done. This is justified since $\mathsf{UE}'(0, 1, \boldsymbol{y}) = 1$ no matter what $\boldsymbol{y}$ is,

Here are the frequencies of the state complexities of all the minimal automata associated with the 65536 Boolean functions of 4 arguments.

1	1
6	81
7	162
8	828
9	3264
10	11040
11	22440
12	27720

Recall that the natural tree automata all have size 31.

The automaton of size 1 corresponds to the constant false functions, the others actually describe languages $L \subseteq \mathbf{2}^4$.

- Boolean Functions
- Clones and Post
- Minimal Automata
4. Binary Decision Diagrams

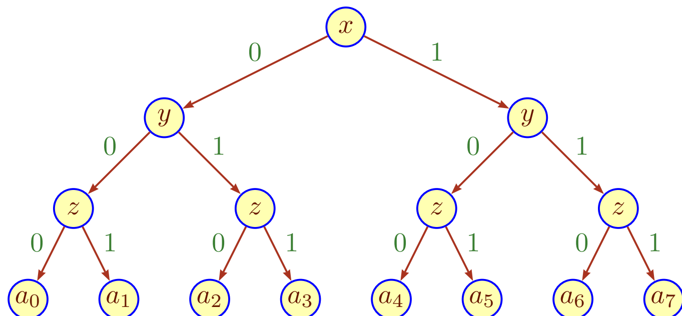
One might conjecture that we have to relax the constraints on our diagrams a bit: more choices should make it easier to build them (of course, while preserving normal form properties).

One fairly natural idea is to drop the fixed-length condition in an MDA: we may reach the exit without reading all the variables (in fact, always as soon as enough information has been accumulated to determine the value of the function).

C. Y. Lee
Binary Decision Programs
Bell System Tech. J., 4 (1959) 4: 985–999.

S. B. Akers
Binary Decision Diagrams
IEEE Trans. Computers C-27 (1978) 6: 509–516.

R. E. Bryant
Graph-based Algorithms for Boolean Function Manipulation
IEEE Trans. Computers C-35 (1986) 8: 677–691.



A general purpose method: express a Boolean function as a decision tree.

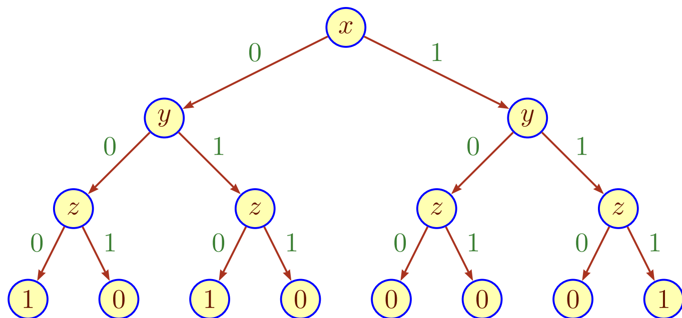
In its abstract form, there is not much one can do with a full decision tree.

But if we have concrete values for the a_i, things change.

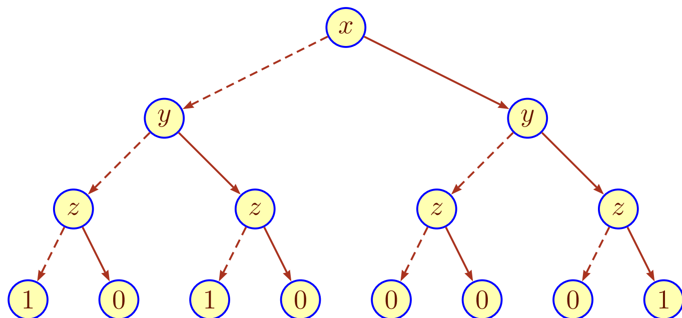
We can reduce the data structure according to the following reduction rules.

- Merge terminal nodes with the same label.
- Merge nodes with the same descendants.
- Skip over nodes with just one child.

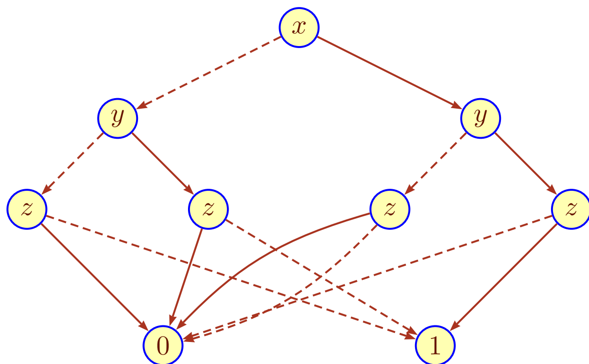
Except for the Skip part, this is the same as state merging in the realm of FSMs.

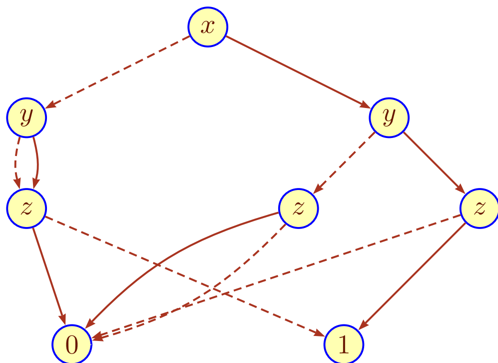


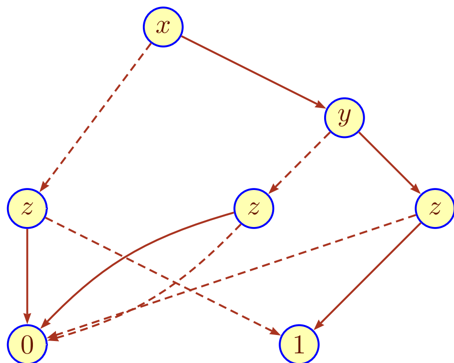
The function $f = (x + \overline{z})(\overline{x} + y)(\overline{x} + \overline{y} + z)$.

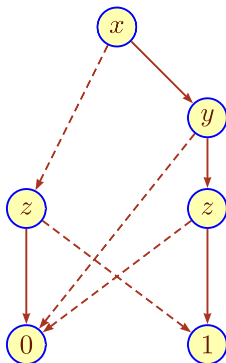


A more legible version.


x
y
y
z
z
z
z
0
1


x
y
y
z
z
z
0
1


x
y
z
z
z
0
1


x
y
z
z
0
1

Just to be clear: reducing a given decision tree is mostly of conceptual interest. The full decision tree always has exponential size, so we can only handle it for a few variables.

What we really need is a way to get at the reduced version directly: starting from the constant functions, use Boolean operators to construct more complicated ones.

All these intermediate functions are kept in reduced form, so hopefully none of them will exhibit exponential blowup.

Here is a more algebraic description of this basic idea.

Suppose f is a Boolean function with variable x. Define the cofactors of f by

$$f_x(\boldsymbol{u}, x, \boldsymbol{v}) = f(\boldsymbol{u}, 1, \boldsymbol{v})$$
$$f_{\overline{x}}(\boldsymbol{u}, x, \boldsymbol{v}) = f(\boldsymbol{u}, 0, \boldsymbol{v})$$

Note that f_x does not depend on x. Also $f_{xy} = f_{yx}$.

The internal nodes of our DAGs are based on the standard Boole-Shannon expansion

$$f = \overline{x}\, f_{\overline{x}} + x\, f_x$$

A good way to think about expansions is in terms of if-then-else:

$$\mathsf{ite}(x, y_1, y_0) = x\, y_1 + \overline{x}\, y_0$$

Together with the Boolean constants, if-then-else provides yet another basis:

$$\neg x = \mathsf{ite}(x, 0, 1)$$

$$x \wedge y = \mathsf{ite}(x, y, 0)$$

$$x \vee y = \mathsf{ite}(x, 1, y)$$

$$x \Rightarrow y = \mathsf{ite}(x, y, 1)$$

$$x \oplus y = \mathsf{ite}(x, \mathsf{ite}(y, 0, 1), y)$$

It follows that we can define if-then-else normal form (INF): the only allowed operations are if-then-else and constants. Moreover, tests are performed only on variables (not compound expressions).

We can express Boole-Shannon expansion in terms of if-then-else:

$$f = \mathsf{ite}(x, f_x, f_{\overline{x}})$$

We also assume that the variables are given in a fixed order $x, y, z \ldots$ so that we can lighten notation a bit and write f_{101} instead of $f_{x\overline{y}z}$.

Consider the Boolean function $f \equiv (x_1 = y_1) \wedge (x_2 = y_2)$ (strictly speaking we should have written $x_i \Leftrightarrow y_i$, but that's harder to read).

The INF of f is given by

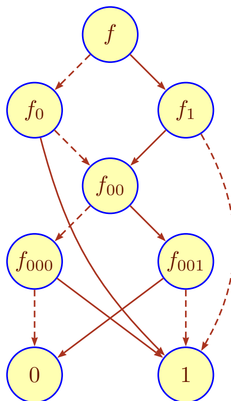
$$\begin{aligned}
f &= \text{ite}(x_1, f_1, f_0)\\
f_0 &= \text{ite}(y_1, 0, f_{00})\\
f_1 &= \text{ite}(y_1, f_{11}, 0)\\
f_{00} &= \text{ite}(x_2, f_{001}, f_{000})\\
f_{11} &= \text{ite}(x_2, f_{111}, f_{110})\\
f_{000} &= \text{ite}(y_2, 0, 1)\\
f_{001} &= \text{ite}(y_2, 1, 0)\\
f_{110} &= \text{ite}(y_2, 0, 1)\\
f_{111} &= \text{ite}(y_2, 1, 0)
\end{aligned}$$

Here the implicit order is x_1, y_1, x_2, y_2. Substituting back into the first line we get INF.

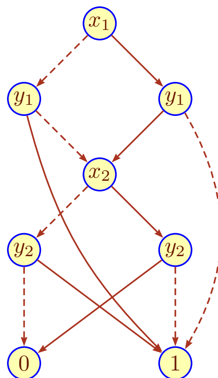
Sharing common subexpressions (such as f_{000} and f_{110}):

$$\begin{aligned} f &= \mathsf{ite}(x_1, f_1, f_0) \\ f_0 &= \mathsf{ite}(y_1, 0, f_{00}) \\ f_1 &= \mathsf{ite}(y_1, f_{00}, 0) \\ f_{00} &= \mathsf{ite}(x_2, f_{001}, f_{000}) \\ f_{000} &= \mathsf{ite}(y_2, 0, 1) \\ f_{001} &= \mathsf{ite}(y_2, 1, 0) \end{aligned}$$

We can now interpret these functions as nodes in a DAG just like the one obtained by reducing a decision tree.



The fully reduced DAG.



The DAG again, but this time labeled by the variables that are tested at each particular level.

Fix a set $\mathsf{Var} = \{x_1, x_2, \ldots, x_n\}$ of n Boolean variables.

Definition

A binary decision diagram (BDD) (over Var) is a rooted, directed acyclic graph with two *terminal* nodes (out-degree 0) and *interior* nodes of out-degree 2. The interior nodes are labeled in Var.

We write $\mathsf{var}(u)$ for the labels.

The two successors of an interior node are traditionally referred to as $\mathsf{lo}(u)$ and $\mathsf{hi}(v)$, corresponding to the false and true branches.

We can think of the terminal nodes as being labeled by constants 0 and 1, indicating values false and true.

Fix an ordering $x_1 < x_2 < \ldots < x_n$ on Var. For simplicity assume that $x_i < 0, 1$.

Definition

A BDD is ordered (OBDD) if the label sequence along any path is ordered.

Thus

$$\mathsf{var}(u) < \mathsf{var}(\mathsf{lo}(u)), \mathsf{var}(\mathsf{hi}(u))$$

In the corresponding INF, the variables are always ordered in the sense that

$$\mathsf{ite}(x, \mathsf{ite}(y, f_1, f_2), \mathsf{ite}(z, f_3, f_4)) \text{ implies } x < y, z$$

Definition

A OBDD is reduced (ROBDD) if it satisfies

Uniqueness: there are no nodes $u \neq v$ such that

$$\mathsf{var}(u) = \mathsf{var}(v), \mathsf{lo}(u) = \mathsf{lo}(v), \mathsf{hi}(u) = \mathsf{hi}(v)$$

Non-Redundancy: for all nodes u:

$$\mathsf{lo}(u) \neq \mathsf{hi}(u)$$

The uniqueness condition corresponds to shared subexpressions: we could merge u and v. Non-redundancy corresponds to taking shortcuts: we can skip ahead to the next test.

Since ROBDDs are the most important type, we simply refer to them as BDD.

By a straightforward induction we can associate any BDD with root u with a Boolean function F_u:

$$F_0 = \mathbf{0} \qquad F_1 = \mathbf{1}$$
$$F_u = \mathsf{ite}(\mathsf{var}(u), F_{\mathsf{lo}(u)}, F_{\mathsf{hi}(u)})$$

If the BDDs under consideration are also ordered and reduced we get a useful representation.

Theorem (Canonical Form Theorem)

For every Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ *there is exactly one ROBDD* u *such that* $f = F_u$*.*

The claim is clear for $n = 0$, so consider a function $f : \mathbb{B}^n \to \mathbb{B}$, written $f(x, \boldsymbol{y})$.

By IH we may assume that the BDDs for $f(0, \boldsymbol{y})$ and $f(1, \boldsymbol{y})$ are unique, say, with roots u and v, respectively.

If $f(0, \boldsymbol{y}) = f(1, \boldsymbol{y})$ this is also the BDD for f.

Otherwise introduce a new node w and set

$$\lambda(w) = x \qquad \mathsf{lo}(w) = u \qquad \mathsf{hi}(w) = v$$

The resulting BDD represents f and is unique.

$\square$

Suppose we have a decision tree for a Boolean function and would like to transform it into the (unique) BDD.

We can use a bottom-up traversal of the DAG to merge or eliminate nodes that violate Uniqueness or Non-Redundancy.

This traversal requires essentially only local information and can be handled in (expected) linear time using a hash table.

Of course, if the starting point is indeed a full decision tree this method is of little interest. But reduction is important as a clean-up step in the construction of complicated BDDs form simpler ones.

Here is the key idea: suppose we construct all the Boolean functions in a particular computation from scratch, starting from the constants and applying the usual arsenal of Boolean connectives.

In this scenario, all functions are kept in canonical form; if an intermediate result is not canonical, clean it up using the reduction machinery. As a consequence, every one of these functions has a unique representation: a particular node u in a BDD represents function F_u (over some set of variables).

But then equivalence testing is $O(1)$: we just compare two pointers. Very nice.

Obviously we need a collection of operations on BDDs.

Reduce Turn a decision tree into a BDD.

Apply Given two BDDs u and v and a Boolean operation $\diamond$, determine the BDD for $F_u \diamond F_v$.

Restrict Given a BDD u and a variable x, determine the BDD for $F_u[a/x]$.

Note: The purpose of Reduce is **not** to convert a raw (exponential size) decision tree, but to clean up after other operations.

The first operation to consider is simple negation: obtaining a BDD for $\overline{f}$ from a BDD for f.

Note that the two BDDs differ only in the terminal nodes, which are swapped.

This can be exploited to make complementation constant time.

Clearly all nodes in a BDD represent Boolean functions, it is natural to keep track of them via pointers. By adding a "negation bit" to the pointers (complement vs. regular pointers), we can obtain $\overline{f}$ at constant cost.

We can also add negation bits to all the internal edges of the diagram, indicating that the next node represents $\overline{g}$ rather than g.

In this setting, one can get away with a single terminal node, say, $\mathbf{1}$. Since all paths end at $\mathbf{1}$, evaluation comes down to counting the number of complement edges (modulo 2).

However, a little care is needed to preserve canonicity. We have

$$f = x f_x + \overline{x} f_{\overline{x}} = \overline{(x \overline{f}_x + \overline{x} \overline{f}_{\overline{x}})}$$

so there are two ways to represent f. We adopt the convention that the *then* branch is chosen to be regular.

With this convention, our BDDs are still canonical.

Cofactors coexist peacefully with Boolean operations.

Proposition

$\overline{f}_x = \overline{f_x}$, $(f+g)_x = f_x + g_x$, $(fg)_x = f_x g_x$

Hence, the apply operation can be handled by recursive, top-down algorithm since

$$\mathsf{ite}(x, s, t) \diamond \mathsf{ite}(x, s', t') = \mathsf{ite}(x, s \diamond s', t \diamond t')$$

Since complementation is $O(1)$, it even suffices to just implement, say, logical and.

So suppose we want to build the BDD for fg, given BDDs for f and g.

The algorithm uses top-down recursion. The exit conditions are easy:

$$
\begin{array}{rc}
f, g = \mathbf{0} & \mathbf{0} \\
f = \mathbf{1} & g \\
g = \mathbf{1} & f \\
f = g & g \\
f = \overline{g} & \mathbf{0}
\end{array}
$$

Note that they can all be checked in constant time.

Apply for Conjunction

- handle terminal cases
- suppose x is next variable
  - compute $h_1 = f_x\, g_x$
  - compute $h_2 = f_{\overline{x}}\, g_{\overline{x}}$
  - if $h_1 = h_2$ return h_1
  - return BDD for ite(x, h_1, h_2)

With proper hashing the running time is $O(|u||v|)$, though in practice the method is often faster.

Suppose we have constructed a BDD for a Boolean function f.

Then it is trivial to check if f is a tautology: the test is for $f = \mathbf{1}$ and is constant time.

Likewise, it is trivial to check if f is satisfiable: we need $f \neq \mathbf{0}$.

In fact, we can count satisfying truth assignments in $O(|u|)$: they correspond to the total number of paths (including potentially phantom variables) from the root to terminal 1.

We can even construct a BDD s for all satisfying truth assignments in time $O(n|s|)$ where $|s| = O(2^{|u|})$.

In principle, we can even handle quantified Boolean formulae by expanding the quantifiers:

$$\exists\, x\, \varphi(x) \equiv \varphi[0/x] \lor \varphi[1/x]$$

$$\forall\, x\, \varphi(x) \equiv \varphi[0/x] \land \varphi[1/x]$$

So this comes down to restriction, a (quadratic) apply operation, followed by reduction.

Alas, since SAT is just a problem of validity of an existentially quantified Boolean formula, this is not going to be fast in general. QBF in general is even PSPACE-complete.

Of course, there is no way to avoid exponential worst-case cost when building a BDD representing some Boolean function from scratch: after all, there are 2^{2^n} functions to be represented. So some representations must have size at least 2^n bits for information-theoretic reasons.

As it turns out, addition of k-bit natural numbers can be expressed nicely in a linear size BDD. Alas, multiplication causes problems (this should sound familiar).

Lemma

BDDs for multiplication require exponential size.

In general, the size of a BDD depends drastically on the chosen ordering of variables (see the UnEqual example above).

It would be most useful to be able to construct a variable ordering that works well with a given function. In practice, variable orderings are computed by heuristic algorithms and can sometimes be improved with local optimization and even simulated annealing algorithms.

Alas, it is $\mathbb{NP}$-hard to determine whether there is an ordering that produces a BDD of size at most some given bound.

Note that for a symmetric Boolean functions variable ordering does not matter: we always get the same BDD.

- conjunctions, disjunctions
- parity (xor)
- threshold (counting)

Exercise

Figure out what the BDD for n-ary xor looks like.

Again: for information-theoretic reasons, there is no way a canonical form such as a MDA or BDD is always small.

As a matter of fact, one can show that both worst case and the average sizes of MDA and BDD agree in the limit (for many variables).

Moreover, there are exponential gaps between the size of standard representations for some Boolean functions. So, to really deal with Boolean functions effectively, one has to have many different implementations available–and switch back and forth, to whatever works best under the circumstances.