UCT

Binary Decision Diagrams

Klaus Sutner

Carnegie Mellon University

Spring 2022

Where Are We?

- Boolean satisfiability seems like a reasonable example of a decision problem that lies a bit outside of $\mathbb{P}$.
- It can be viewed as a (heavily constrained) Entscheidungsproblem, the mother of all hopeless problems.
- Lots of other combinatorial problems such as Vertex Cover or Hamiltonian Cycle can be naturally translated into SAT.
- DPLL is a relatively straightforward backtracking algorithm that works surprisingly well in many cases.

Here is an entirely different approach to tackling SAT and, more generally, Boolean function problems: a clever data structure. This provides more evidence that SAT is not too far from feasible.

Boolean Functions

Instead of propositional logic we can study functions over 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.

There Are Lots

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}

So the number of Boolean functions on 8 arguments is almost the same as the number of particles in the universe.

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

. . . , 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.

Different Angles

- 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 Problems

Propositional logic problems easily carry over:

Problem:	**Tautology BF**
Instance:	A Boolean function f.
Question:	Is f always true?

Problem:	**Satisfiability BF**
Instance:	A Boolean function f.
Question:	Is f sometimes true?

Problem:	**Equivalence BF**
Instance:	Two Boolean functions f and g.
Question:	Is $f = g$?

In the equivalence problem we assume f and g have the same arity n, and we want $\forall\, x \in \mathbf{2}^n\, (f(x) = g(x))$.

Representation

Consider, say, Equivalence BF.

- If f and g are given as bitvectors, this is boring.
- If f and g are given as Boolean formulae or circuits, this gets interesting: we want to check equivalence without exponentially many evaluations[†].
- If f and g are given as BDDs, this is trivial.

This is not a typo, in the world of BDDs equivalence testing is constant time. Of course, there is a catch . . .

[†] The formula is a succint representation of the bitvector.

Example: Threshold Functions

On occasion, Boolean functions have a clear combinatorial meaning and are also easy to understand.

Definition

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

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

thr_m^n is nicely symmetric:

$$\mathrm{thr}_m^n(\boldsymbol{x}) = \mathrm{thr}_m^n(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

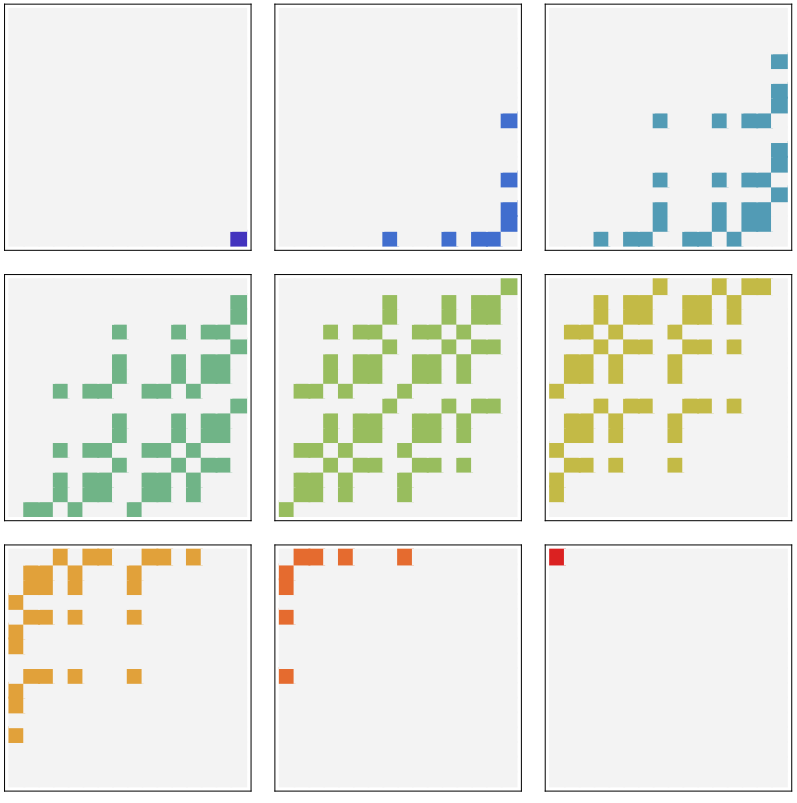
Related Functionss

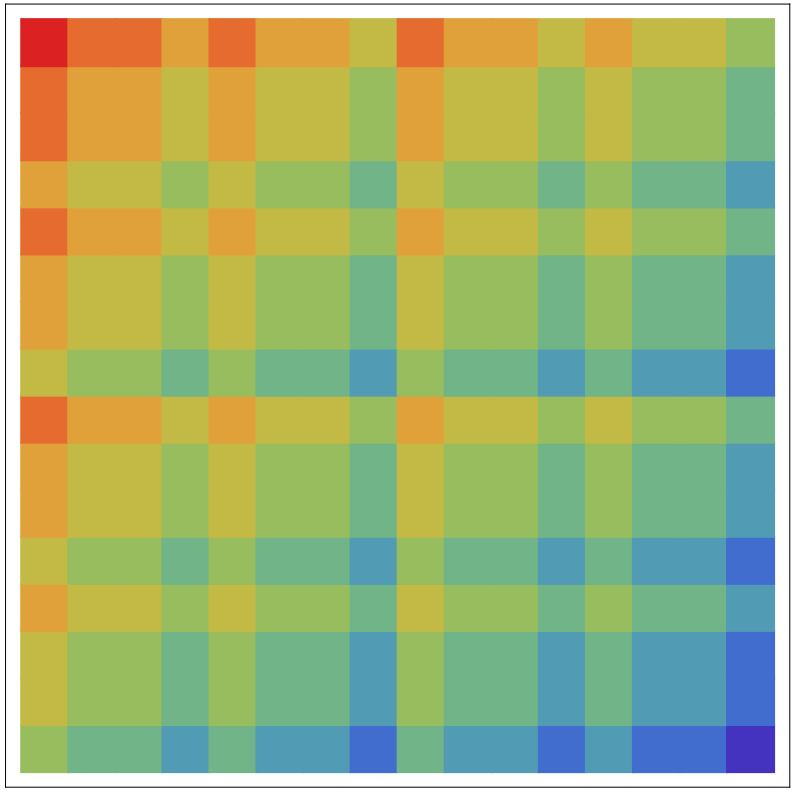
Lots of useful Boolean functions can be defined in terms of threshold functions.

- thr_0^n is the constant True.
- thr_1^n is n-ary disjunction.
- thr_n^n is n-ary conjunction.
- $\mathrm{thr}_k^n(\boldsymbol{x}) \wedge \neg \mathrm{thr}_{k+1}^n(\boldsymbol{x})$ is the counting function: "exactly k out of n are true."

As we have already seen, counting functions can be very helpful in constructing complicated Boolean formulae.

Counting with 8 Variables





4 Queens

$$
\begin{aligned}
&(x_{1,1} \Rightarrow \neg\, x_{2,2} \wedge \neg\, x_{3,3} \wedge \neg\, x_{4,4}) \wedge (x_{1,2} \Rightarrow \neg\, x_{2,3} \wedge \neg\, x_{3,4} \wedge \neg\, x_{2,1}) \wedge (x_{1,3} \Rightarrow \neg\, x_{2,4} \wedge \neg\, x_{2,2} \wedge \neg\, x_{3,1}) \wedge \\
&(x_{1,4} \Rightarrow \neg\, x_{2,3} \wedge \neg\, x_{3,2} \wedge \neg\, x_{4,1}) \wedge (x_{2,1} \Rightarrow \neg\, x_{3,2} \wedge \neg\, x_{4,3}) \wedge (x_{2,2} \Rightarrow \neg\, x_{3,3} \wedge \neg\, x_{4,4} \wedge \neg\, x_{3,1}) \wedge (x_{2,3} \Rightarrow \neg\, x_{3,4} \wedge \neg\, x_{3,2} \wedge \neg\, x_{4,1}) \wedge \\
&(x_{2,4} \Rightarrow \neg\, x_{3,3} \wedge \neg\, x_{4,2}) \wedge (x_{3,1} \Rightarrow \neg\, x_{4,2}) \wedge (x_{3,2} \Rightarrow \neg\, x_{4,3} \wedge \neg\, x_{4,1}) \wedge (x_{3,3} \Rightarrow \neg\, x_{4,4} \wedge \neg\, x_{4,2}) \wedge (x_{3,4} \Rightarrow \neg\, x_{4,3}) \wedge \\
&((x_{1,1} \wedge \neg\, x_{1,2} \wedge \neg\, x_{1,3} \wedge \neg\, x_{1,4}) \vee (\neg\, x_{1,1} \wedge x_{1,2} \wedge \neg\, x_{1,3} \wedge \neg\, x_{1,4}) \vee (\neg\, x_{1,1} \wedge \neg\, x_{1,2} \wedge x_{1,3} \wedge \neg\, x_{1,4}) \vee (\neg\, x_{1,1} \wedge \neg\, x_{1,2} \wedge \neg\, x_{1,3} \wedge x_{1,4})) \wedge \\
&((x_{2,1} \wedge \neg\, x_{2,2} \wedge \neg\, x_{2,3} \wedge \neg\, x_{2,4}) \vee (\neg\, x_{2,1} \wedge x_{2,2} \wedge \neg\, x_{2,3} \wedge \neg\, x_{2,4}) \vee (\neg\, x_{2,1} \wedge \neg\, x_{2,2} \wedge x_{2,3} \wedge \neg\, x_{2,4}) \vee (\neg\, x_{2,1} \wedge \neg\, x_{2,2} \wedge \neg\, x_{2,3} \wedge x_{2,4})) \wedge \\
&((x_{3,1} \wedge \neg\, x_{3,2} \wedge \neg\, x_{3,3} \wedge \neg\, x_{3,4}) \vee (\neg\, x_{3,1} \wedge x_{3,2} \wedge \neg\, x_{3,3} \wedge \neg\, x_{3,4}) \vee (\neg\, x_{3,1} \wedge \neg\, x_{3,2} \wedge x_{3,3} \wedge \neg\, x_{3,4}) \vee (\neg\, x_{3,1} \wedge \neg\, x_{3,2} \wedge \neg\, x_{3,3} \wedge x_{3,4})) \wedge \\
&((x_{4,1} \wedge \neg\, x_{4,2} \wedge \neg\, x_{4,3} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{4,1} \wedge x_{4,2} \wedge \neg\, x_{4,3} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{4,1} \wedge \neg\, x_{4,2} \wedge x_{4,3} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{4,1} \wedge \neg\, x_{4,2} \wedge \neg\, x_{4,3} \wedge x_{4,4})) \wedge \\
&((x_{1,1} \wedge \neg\, x_{2,1} \wedge \neg\, x_{3,1} \wedge \neg\, x_{4,1}) \vee (\neg\, x_{1,1} \wedge x_{2,1} \wedge \neg\, x_{3,1} \wedge \neg\, x_{4,1}) \vee (\neg\, x_{1,1} \wedge \neg\, x_{2,1} \wedge x_{3,1} \wedge \neg\, x_{4,1}) \vee (\neg\, x_{1,1} \wedge \neg\, x_{2,1} \wedge \neg\, x_{3,1} \wedge x_{4,1})) \wedge \\
&((x_{1,2} \wedge \neg\, x_{2,2} \wedge \neg\, x_{3,2} \wedge \neg\, x_{4,2}) \vee (\neg\, x_{1,2} \wedge x_{2,2} \wedge \neg\, x_{3,2} \wedge \neg\, x_{4,2}) \vee (\neg\, x_{1,2} \wedge \neg\, x_{2,2} \wedge x_{3,2} \wedge \neg\, x_{4,2}) \vee (\neg\, x_{1,2} \wedge \neg\, x_{2,2} \wedge \neg\, x_{3,2} \wedge x_{4,2})) \wedge \\
&((x_{1,3} \wedge \neg\, x_{2,3} \wedge \neg\, x_{3,3} \wedge \neg\, x_{4,3}) \vee (\neg\, x_{1,3} \wedge x_{2,3} \wedge \neg\, x_{3,3} \wedge \neg\, x_{4,3}) \vee (\neg\, x_{1,3} \wedge \neg\, x_{2,3} \wedge x_{3,3} \wedge \neg\, x_{4,3}) \vee (\neg\, x_{1,3} \wedge \neg\, x_{2,3} \wedge \neg\, x_{3,3} \wedge x_{4,3})) \wedge \\
&((x_{1,4} \wedge \neg\, x_{2,4} \wedge \neg\, x_{3,4} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{1,4} \wedge x_{2,4} \wedge \neg\, x_{3,4} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{1,4} \wedge \neg\, x_{2,4} \wedge x_{3,4} \wedge \neg\, x_{4,4}) \vee (\neg\, x_{1,4} \wedge \neg\, x_{2,4} \wedge \neg\, x_{3,4} \wedge x_{4,4}))
\end{aligned}
$$

The Boolean function corresponding to the queens formula for a 4×4 board: $x_{ij} = 1$ means there is a queen in position (i, j).

The two solutions of $f(\boldsymbol{x}) = 1$ are 0100000110000010 and 0010100000010100, which are mirror images of each other.

A Challenge

It is entirely straightforward to implement Boolean functions as propositional formulae, and those as labeled trees, essentially the parse trees of the corresponding expressions.

Straightforward, but not necessarily algorithmically smart: it completely ignores all the special properties of Boolean formulae (as opposed to some other terms, say arithmetic terms that have the same signature).

So is there a special purpose data structure that is custom designed just for Boolean functions?

More Precisely . . .

We would like our implementation to make it easy to deal with decision problems such as Tautology or Satisfiability, at least in some cases.

Since we chose these problems to be difficult, we should not expect to get all the way down to $\mathbb{P}$, but we don't want brute-force exponential time either.

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

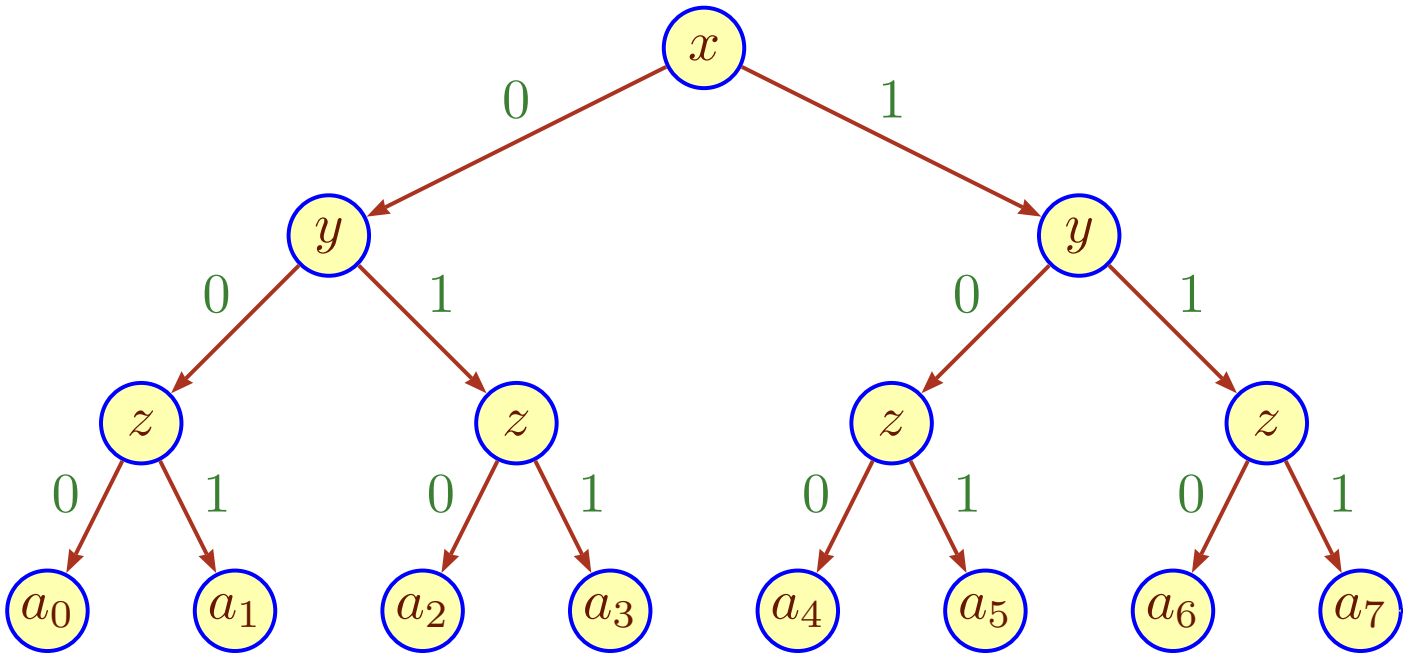
One plausible line of attack is to use some kind of canonical normal form: if we represent the function in normal form, for example equivalence comes down to equality (or perhaps isomorphism).

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.

Decision Trees



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

Reductions

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

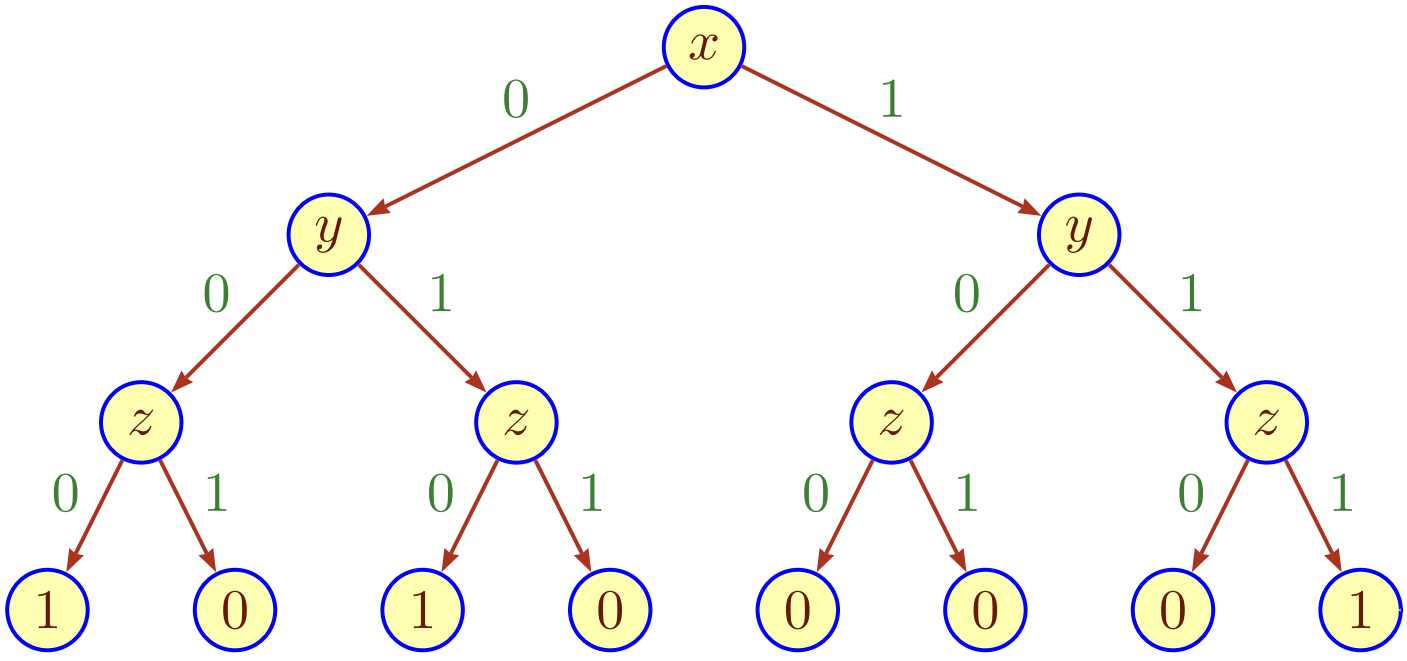
But if we have concrete values for the a_i, things change.

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.

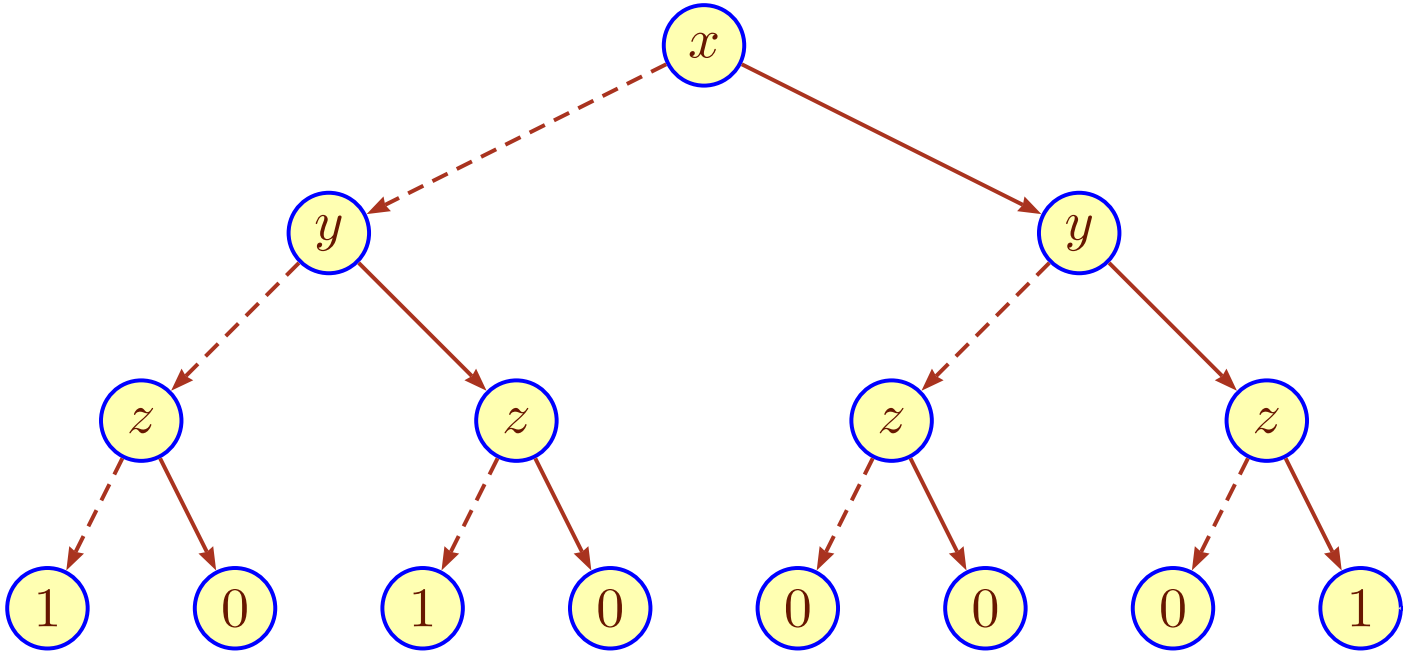
Except for the Skip part, this is the same as state merging in the realm of FSMs.

Example



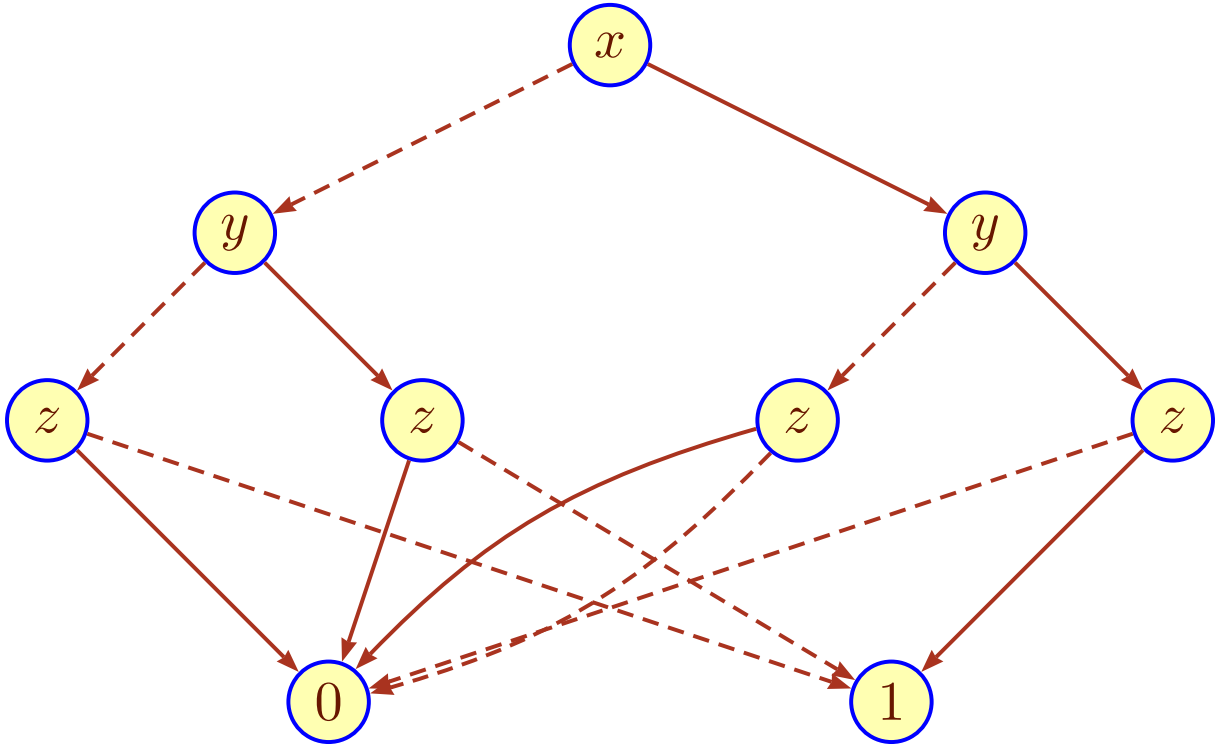
The function $f = (x + \overline{z})(\overline{x} + y)(\overline{x} + \overline{y} + z)$.

Preserving Sanity

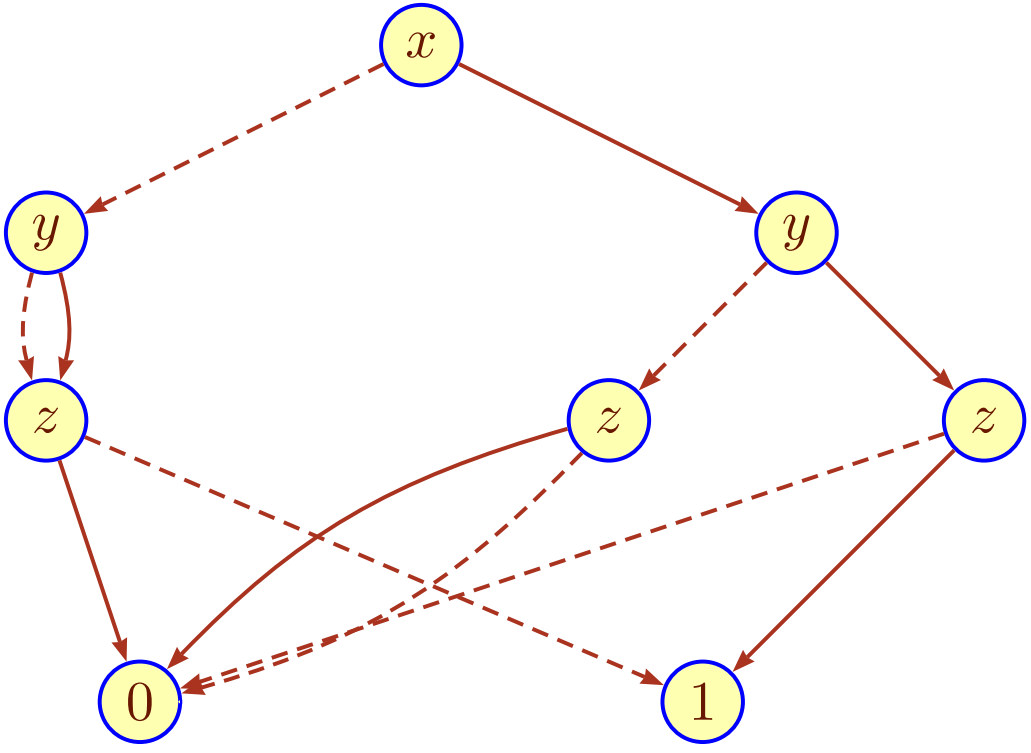


A more legible version.

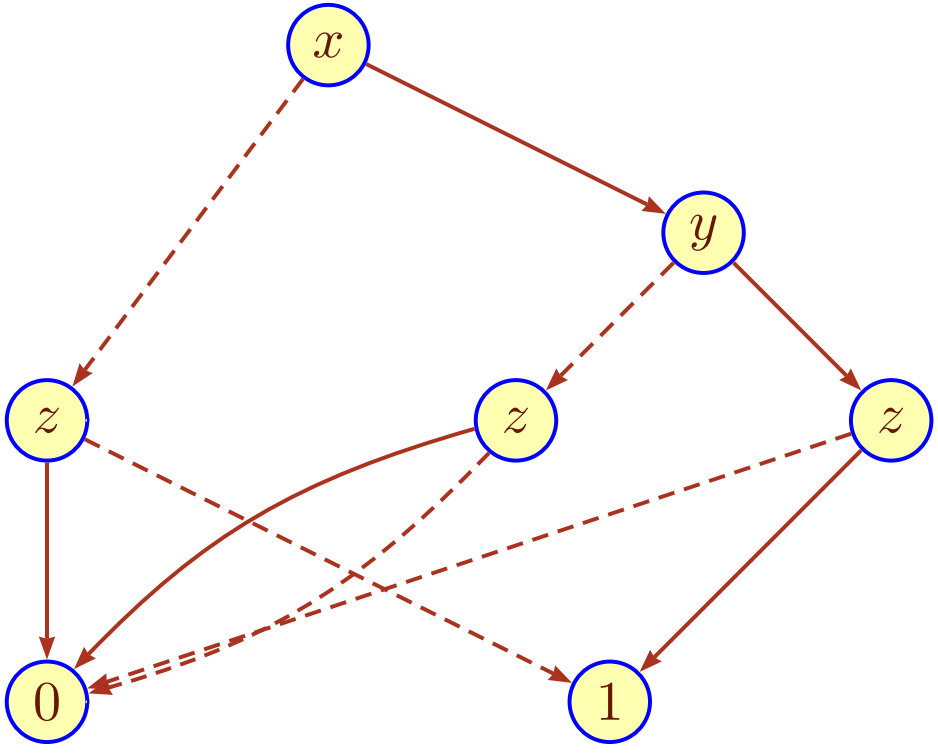
Merging Terminals



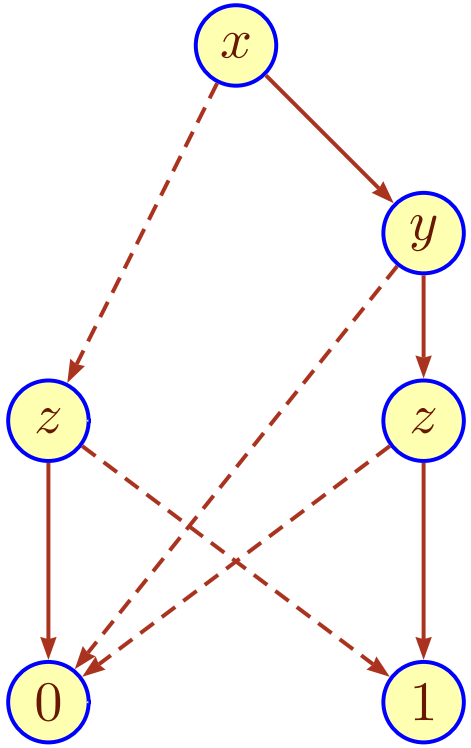
Merging Nodes



Skipping Nodes



Final Result



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

Reality Check

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.

Boole-Shannon Expansion

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

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

If-Then-Else

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:

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

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

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

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

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

If-Then-Else Normal Form

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 $f_{x\overline{y}z}$.

Example

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

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

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

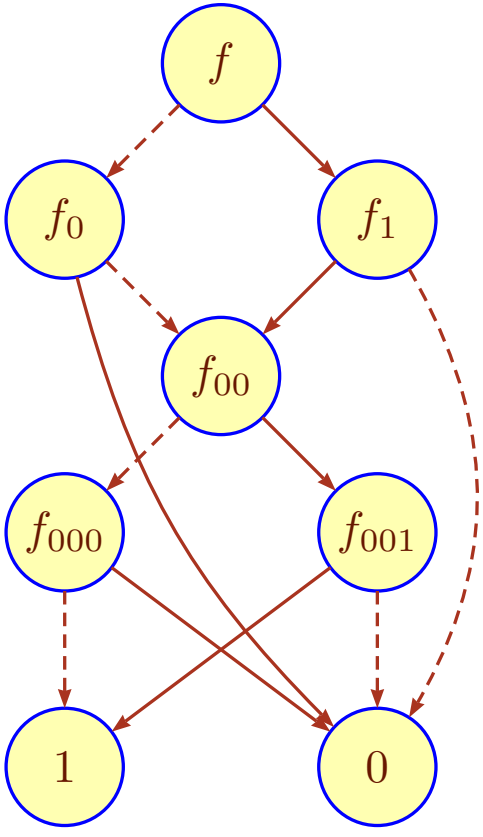
Example

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

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

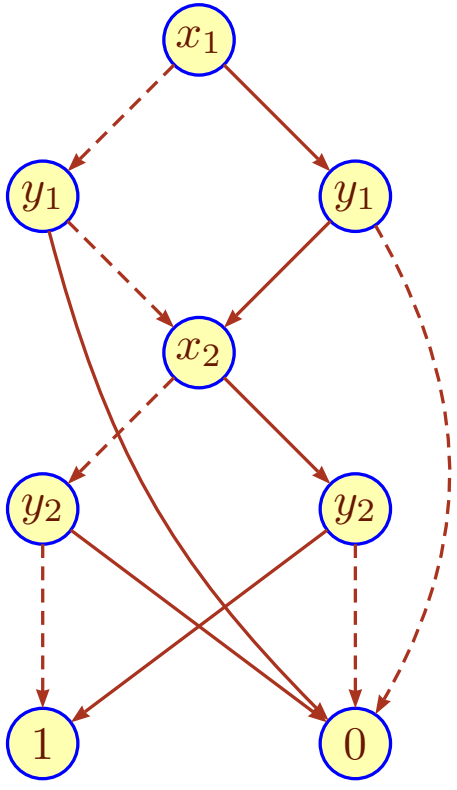
We can now interpret these functions as nodes in a DAG just like the one obtained by reducing a decision tree.

The DAG



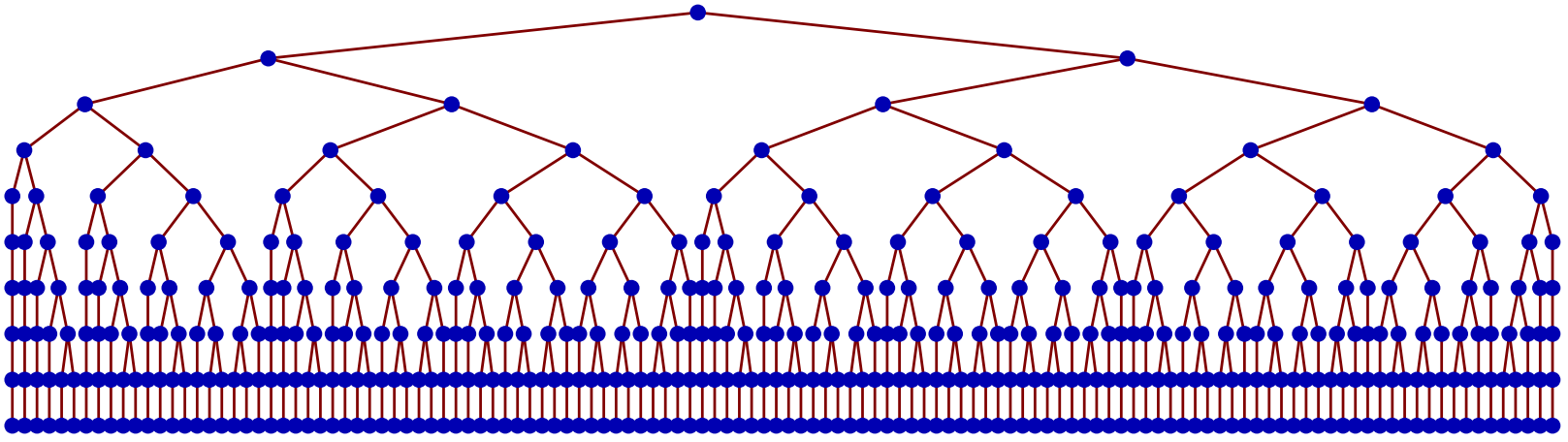
The fully reduced DAG.

Same



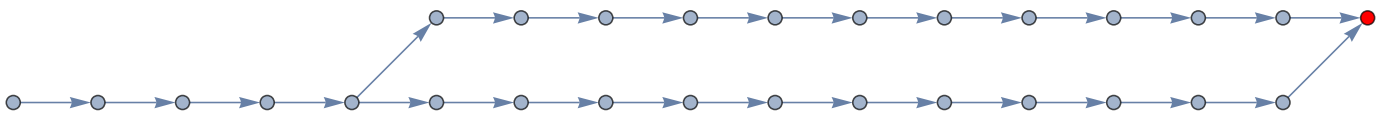
The DAG again, but this time labeled by the variables that are tested at each particular level.

Threshold



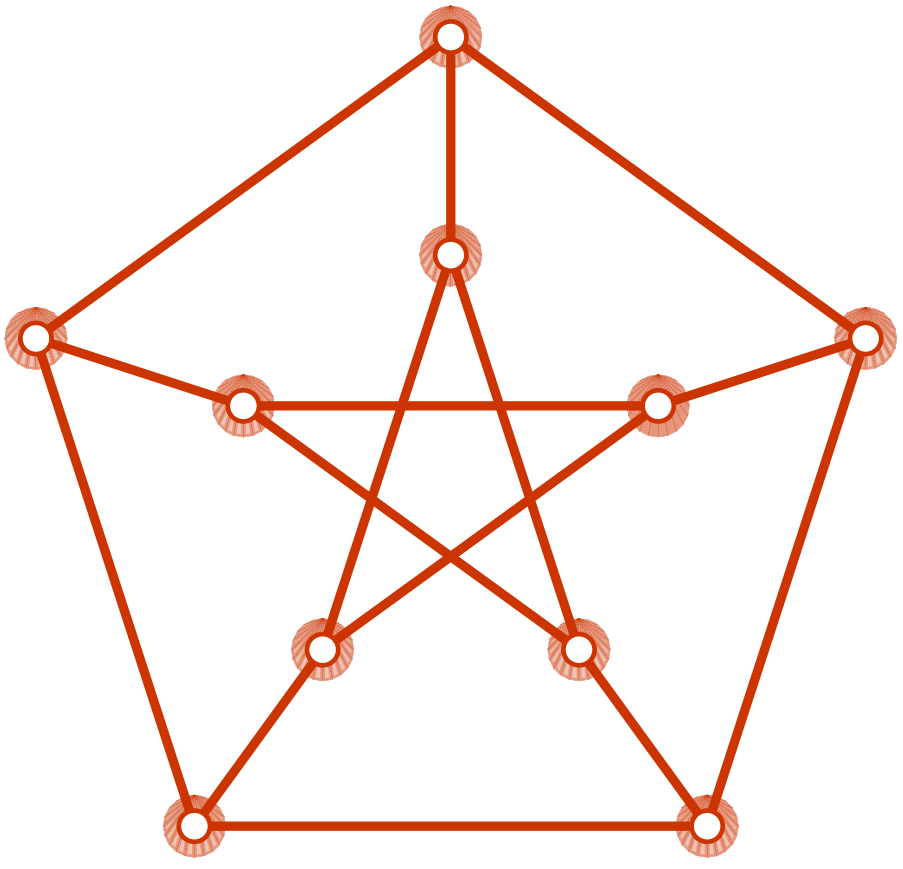
The BDT for the counting function “at most 4 out of 10” (True and False nodes omitted). Number of nodes is 463, uncomfortably close to a full binary tree.

```
If[x1,1, 0,
 If[x2,2, 0, If[x3,3, 0, If[x4,4, 0, If[x1,2, If[x2,3, 0, If[x3,4, 0, If[x2,1, 0, If[x1,3, 0, If[x2,4,
          If[x3,1, If[x1,4, 0, If[x3,2, 0, If[x4,1, 0, If[x4,3, If[x4,2, 0, 1], 0]]]], 0], 0]]]]],
     If[x2,3, 0, If[x3,4, If[x2,1, If[x1,3, If[x2,4, 0, If[x3,1, 0, If[x1,4, 0,
            If[x3,2, 0, If[x4,1, 0, If[x4,3, 0, If[x4,2, 1, 0]]]]]], 0], 0], 0]]]]]]]
```



The INF for the 4-queens formula, and the corresponding BDT (False node omitted). Clearly, there are 2 solutions.

Petersen Graph



The Petersen graph is a standard source of examples and counterexamples in graph theory.

Hamiltonicity

Recall our translation from Hamiltonian Cycle to SAT:

We have Boolean variables $p_{t,x}$ where $0 \leq t \leq n$, $1 \leq x \leq n$ with the intent that:

$p_{t,x} = 1$ iff the Hamiltonian cycle at time t passes through vertex x.

$$p_{0,1} \wedge p_{n,1}$$

$$\forall\, t\, \exists!\, x\, p_{t,x}$$

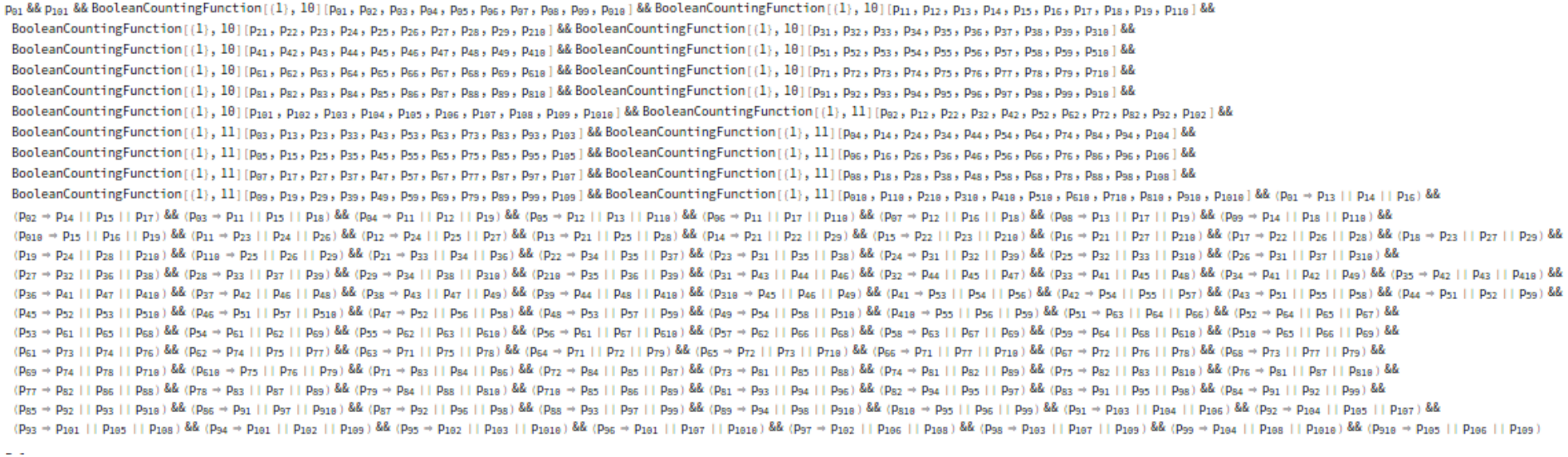
$$\forall\, x > 1\, \exists!\, t\, p_{t,x}$$

$$\forall\, t < n, x\, \left(p_{t,x} \Rightarrow p_{t+1,y_1} \vee \ldots \vee p_{t+1,y_k}\right)$$

where $y_1, \ldots, y_k$ is the neighborhood of x.

A similar formula encodes the existence of a Hamiltonian path instead.

Hamiltonian Cycle



A Boolean formula that is satisfiable iff the Petersen graph has a Hamiltonian cycle.

Converting it to a BDD produces `False`.

The corresponding formula for Hamiltonian paths has 24 truth assignments.

Fix a set 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 var(u) for the labels.

The two successors of an interior node are traditionally referred to as lo(u) and 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.

Ordered BDDs

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

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

Reduced Ordered BDDs

Definition

A OBDD is reduced (ROBDD) if it satisfies

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

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

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.

Canonicity

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 $f : \mathbf{2}^n \to \mathbf{2}$ there is exactly one ROBDD u such that $f = F_u$.

Proof

The claim is clear for $n = 0$, so consider a function $f : \mathbf{2}^n \to \mathbf{2}$, 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

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

The resulting BDD represents f and is unique.

□

Reduction

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.

Workspace Approach

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. All Boolean functions are kept in the same workspace.

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.

Operations on BDDs

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.

Negation

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.

Preserving Canonicity

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.

Ops

Cofactors coexist peacefully with Boolean operations.

Proposition

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

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.

Conjunction

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:

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

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 $|s| = O(2^{|u|})$.

Quantifiers

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

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

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

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.

Bad News

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. In fact, some (most) representations must have size at least $2^n/n$, 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.

Variable Ordering

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.

Symmetric Functions

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.

Average Performance

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.

Out Of The Box

So we want to develop a good data structure for Boolean functions.

It is natural to run through the list of the usual suspects, trees in particular might seem promising.

Here is a rather different first line of attack: translate everything into a problem about regular languages, and then use finite state machines, one of the huge success stories of computer science.

Boolean Functions as Languages

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 $f(\boldsymbol{a}) = 1$, the fibre of 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.

As an example, consider 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 Fix

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

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

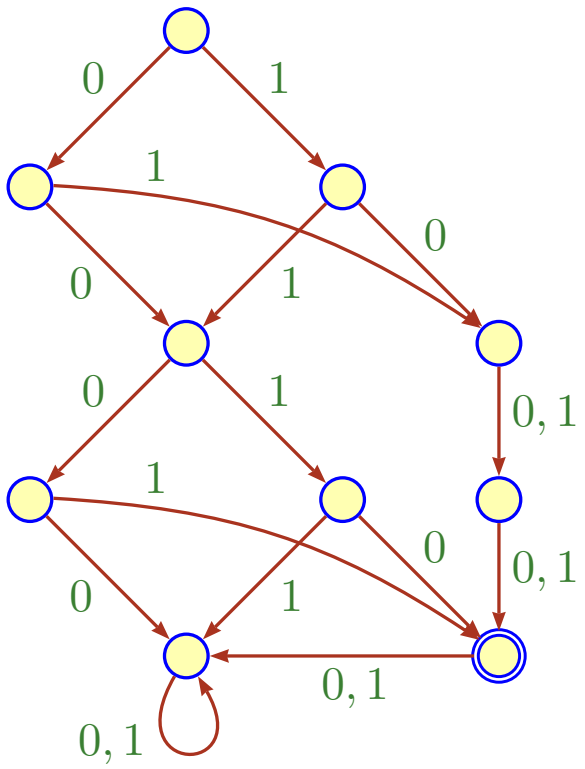
assuming 0-indexing. In other words, the corresponding formula is now

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

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

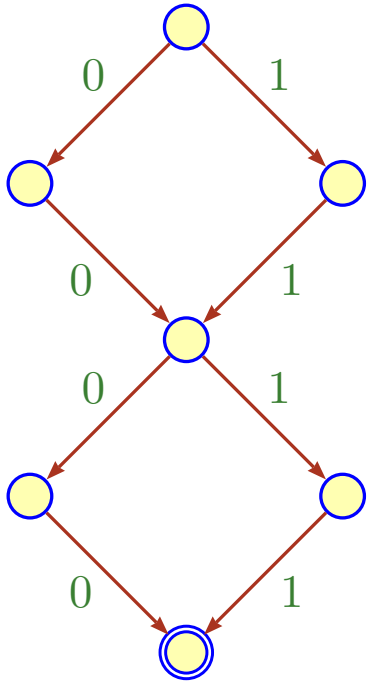
Much Better

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



The minimal DFA for K_2.

The Complement



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

The Complement?

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

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

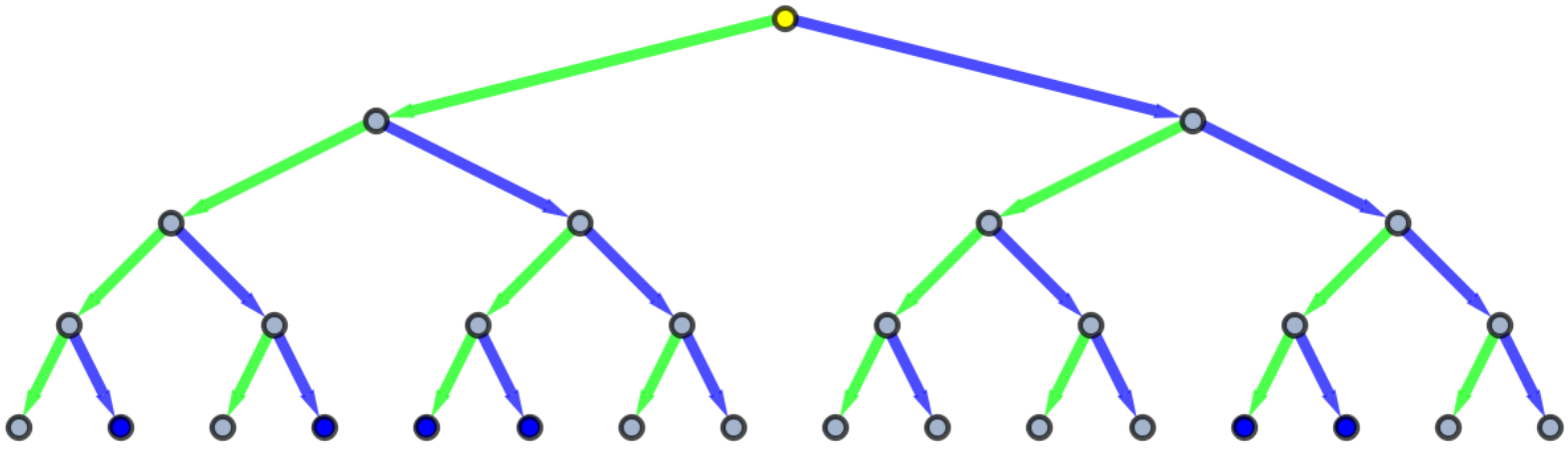
But that's not really what we want for $\overline{f}$, instead we need M' such that

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

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 $g(x, y, z, u)$.

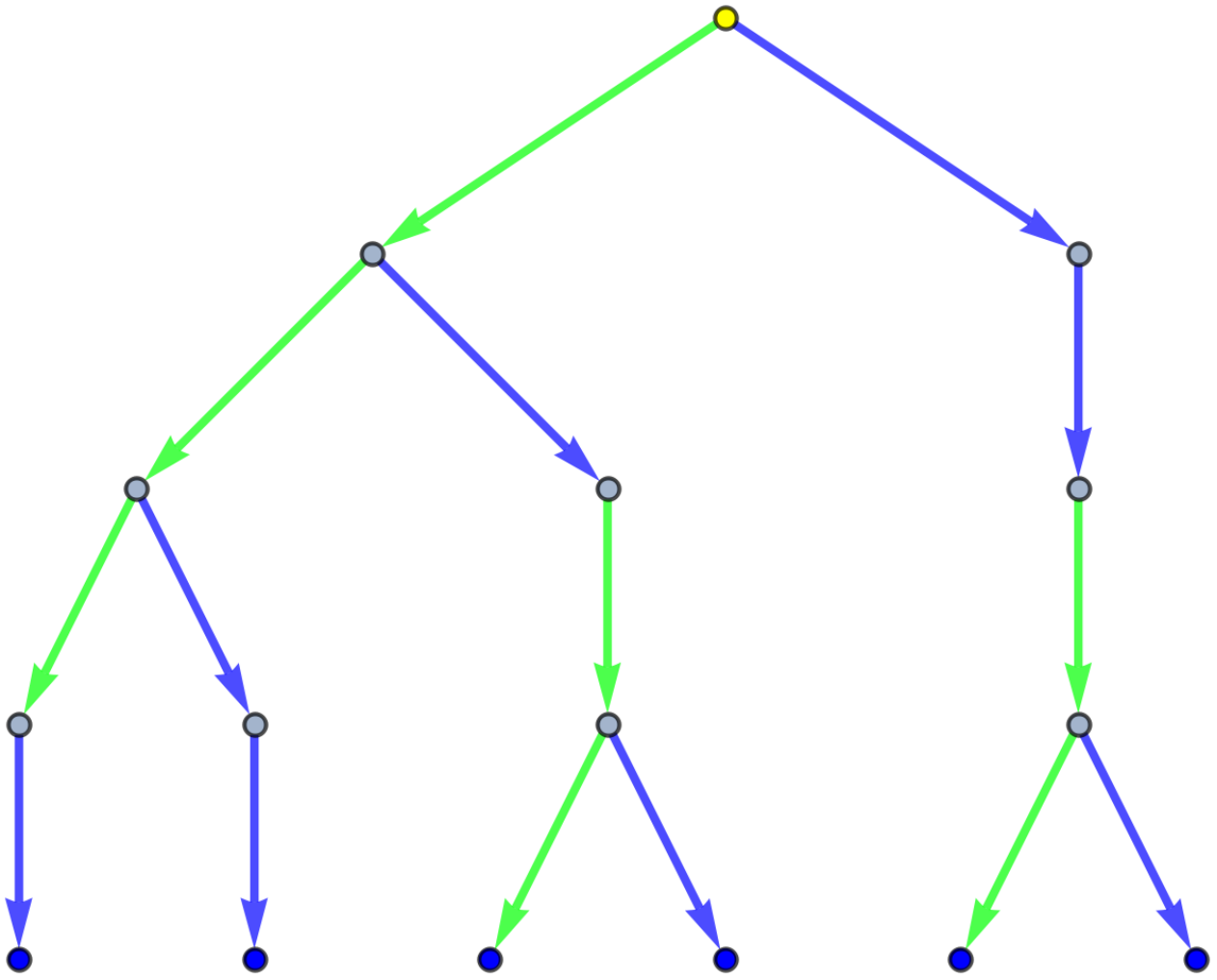
Example: $n = 4$, $\#12346$



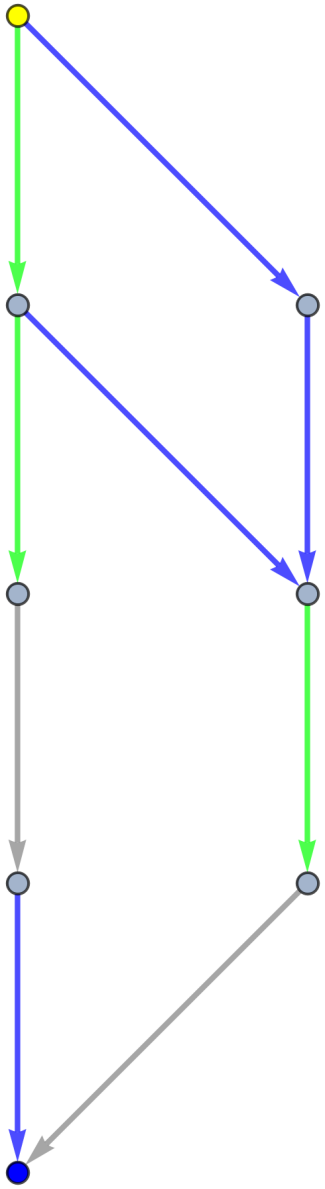
The obvious tree automaton on 31 states for the Boolean function

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

Green means input 0, blue means input 1.



Minimal Partial



The good news is this: because of the uniqueness of the minimal DFA, we do have 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 $\mathrm{UE}'(0, 1, \boldsymbol{y}) = 1$ no matter what $\boldsymbol{y}$ is,

Four Arguments

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, non-optimized tree automata all have size 31.

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

Size Bounds

In general, every n-ary Boolean function has a formula/circuit of size at most $n2^n$, ignoring $\neg$ signs.

Since the number of Boolean functions is 2^{2^n}, there is no way each and every one could have a short description in terms of a propositional formula (or a Boolean circuit).

Theorem (Shannon 1949)

Most n-ary Boolean functions do not have a formula/circuit of size at most $2^n/n$, for all n sufficently large.