Searching for optimal Boolean chains

Adam P. Goucher

February 28, 2023

Boolean chains

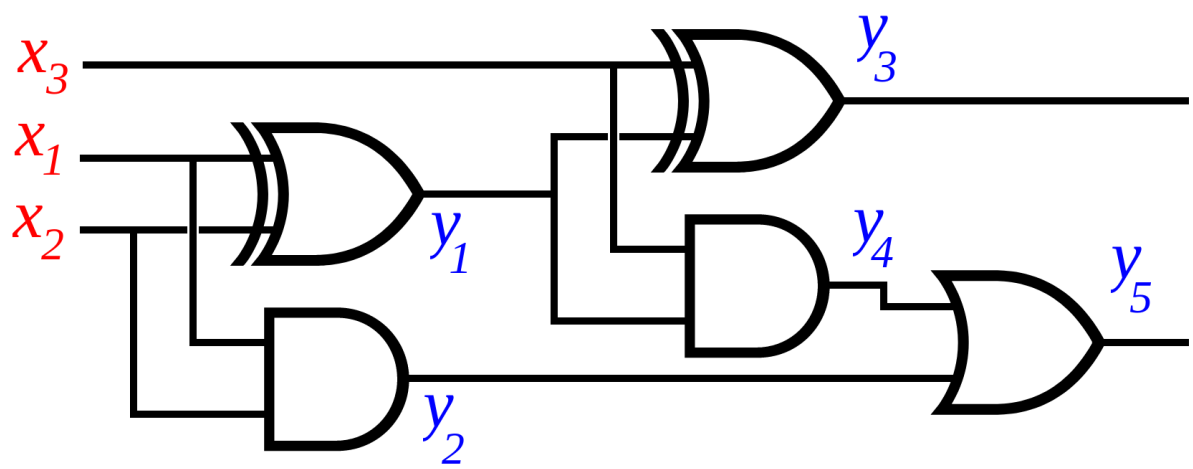
A **Boolean chain** is a sequence of 2-input Boolean gates.

Boolean chains

A **Boolean chain** is a sequence of 2-input Boolean gates.

For example, the full adder has $n = 5$ gates and $k = 3$ inputs:

1. $y_1 = x_1 \oplus x_2$;
2. $y_2 = x_1 \wedge x_2$;
3. $y_3 = y_1 \oplus x_3$;
4. $y_4 = y_1 \wedge x_3$;
5. $y_5 = y_2 \vee y_4$.



Each gate can only depend on inputs or previously-computed values.

Five normal gates

Without loss of generality we can assume that all gates $\circ$ are:

- **Nontrivial**: $a \circ b$ depends on both a and b;
- **Zero-preserving**: $0 \circ 0 = 0$.

Knuth (2011) calls these **normal** chains.

Five normal gates

Without loss of generality we can assume that all gates $\circ$ are:

- **Nontrivial**: $a \circ b$ depends on both a and b;
- **Zero-preserving**: $0 \circ 0 = 0$.

Knuth (2011) calls these **normal** chains.

Out of the $2^{2^2} = 16$ functions, 8 are zero-preserving, of which 5 are nontrivial:

$$\mathcal{O} = \{\oplus, \wedge, \vee, <, >\}$$

Five normal gates

Without loss of generality we can assume that all gates $\circ$ are:

- **Nontrivial**: $a \circ b$ depends on both a and b;
- **Zero-preserving**: $0 \circ 0 = 0$.

Knuth (2011) calls these **normal** chains.

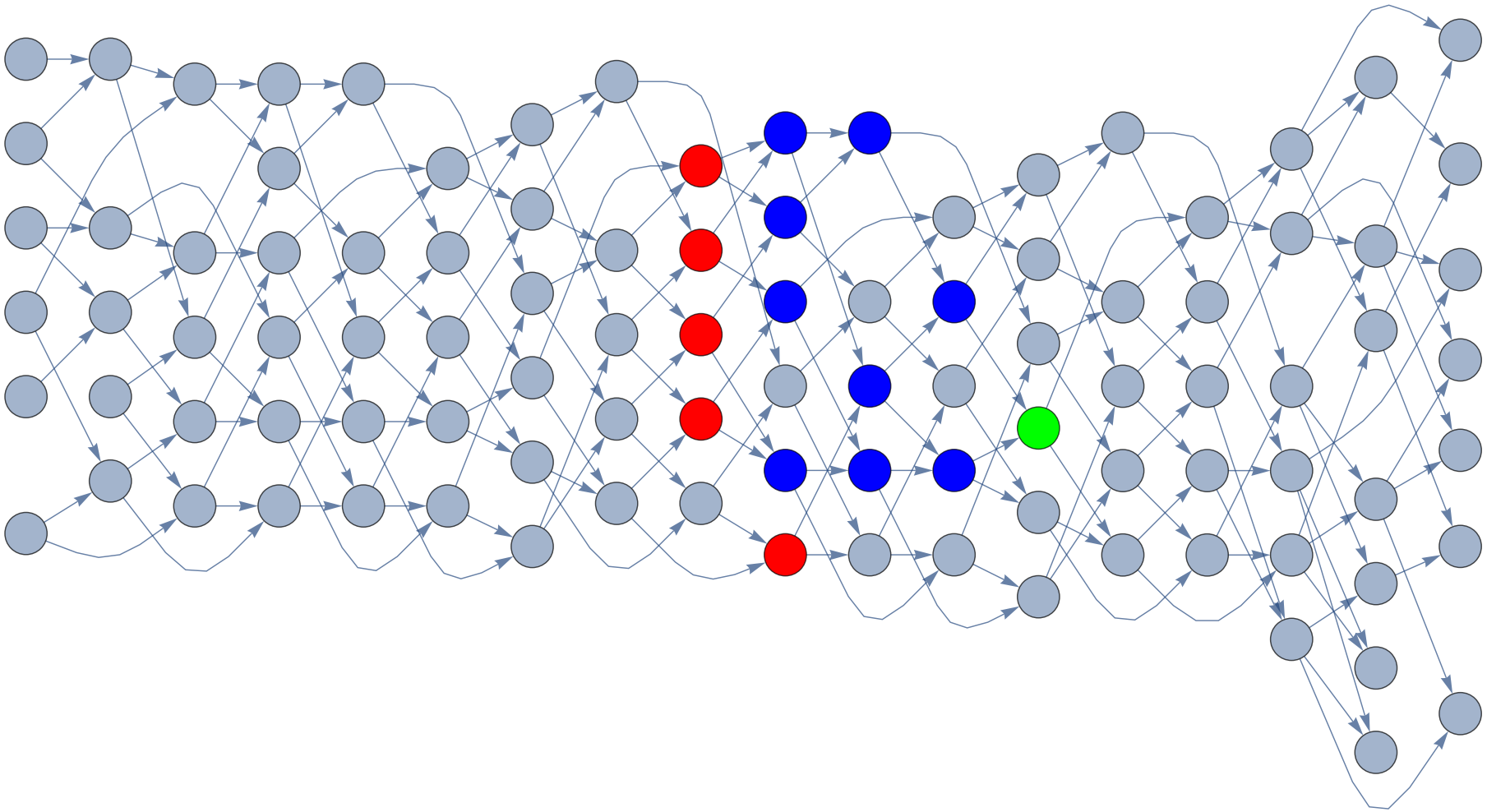
Out of the $2^{2^2} = 16$ functions, 8 are zero-preserving, of which 5 are nontrivial:

$$\mathcal{O} = \{\oplus, \wedge, \vee, <, >\}$$

These correspond to AVX instructions **vpxor**, **vpand**, **vpor**, **vpandn**.

Rewriting

Many logic synthesis tools (e.g. Berkeley's ABC) work by **local rewriting**: replacing small subcircuits with more efficient equivalents.



Prior work

Berkeley's ABC finds 4-input cuts and optimally rewrites those.

Prior work

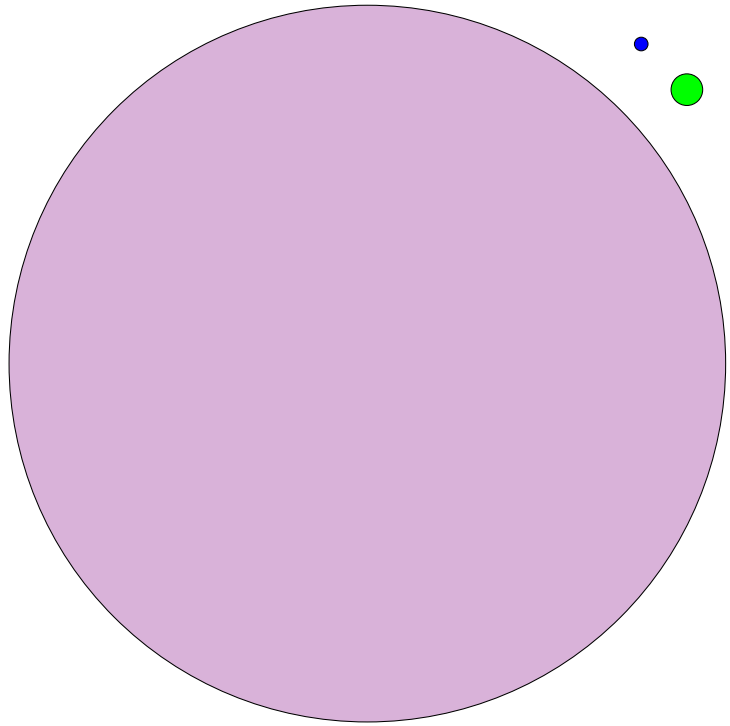
Berkeley's ABC finds 4-input cuts and optimally rewrites those.

Nan Li and Elena Dubrova (2011) found significant benefits (5% cost reduction) by using a library of **1200** 5-input functions.

Prior work

Berkeley's ABC finds 4-input cuts and optimally rewrites those.

Nan Li and Elena Dubrova (2011) found significant benefits (5% cost reduction) by using a library of **1200** 5-input functions.



We shall take this to its ultimate logical conclusion: finding all optimal chains for **616125** of the 616126 equivalence classes of 5-input functions.

Equivalence classes

We can transform a k-input ℓ-output function into an equivalent function by:

- Permuting inputs ($k!$ possibilities);
- Negating inputs (2^k possibilities);
- Permuting outputs ($\ell!$ possibilities);
- Negating outputs (2^ℓ possibilities);

which generate the group $(S_2 \wr S_k) \times (S_2 \wr S_\ell)$ of order $2^{k+\ell}(k!)(\ell!)$.

Equivalence classes

We can transform a k-input ℓ-output function into an equivalent function by:

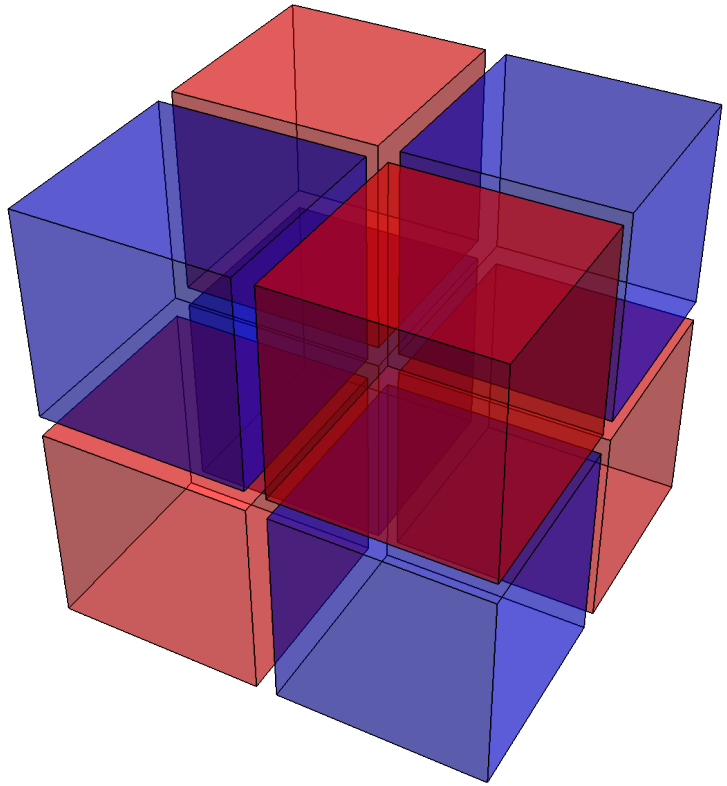
- Permuting inputs ($k!$ possibilities);
- Negating inputs (2^k possibilities);
- Permuting outputs ($\ell!$ possibilities);
- Negating outputs (2^ℓ possibilities);

which generate the group $(S_2 \wr S_k) \times (S_2 \wr S_\ell)$ of order $2^{k+\ell}(k!)(\ell!)$.

$$0001 < 0010 < 0100 < 0111 < 1000 < 1011 < 1101 < 1110$$

We describe the lexicographically first truth table in an equivalence class as **canonical**.

Counting equivalence classes



We can count the equivalence classes using Burnside's lemma; the dominant term will be:

$$|C| \approxeq \frac{2^{2^k \ell}}{2^{k+\ell}(k!)(\ell!)}$$

Number of classes of functions of each cost

n	5-input 1-output	4-input 2-output
0	2	4
1	2	8
2	5	38
3	20	193
4	93	916
5	389	4869
6	1988	27219
7	11382	135402
8	60713	475926
9	221541	713796
10	293455	117828
11	26535	19
12	1	0
Total	**616126**	**1476218**

Size of search space

With k inputs and n gates, the total number of Boolean chains is:

$$\prod_{i=0}^{n-1} 5\binom{i+k}{2}$$

Size of search space

With k inputs and n gates, the total number of Boolean chains is:

$$\prod_{i=0}^{n-1} 5\binom{i+k}{2}$$

In particular, for 5 inputs and 11 gates, the number of chains is:

$$188749394231835937500000000 \approxeq 1.89 \times 10^{25}$$

Size of search space

With k inputs and n gates, the total number of Boolean chains is:

$$\prod_{i=0}^{n-1} 5\binom{i+k}{2}$$

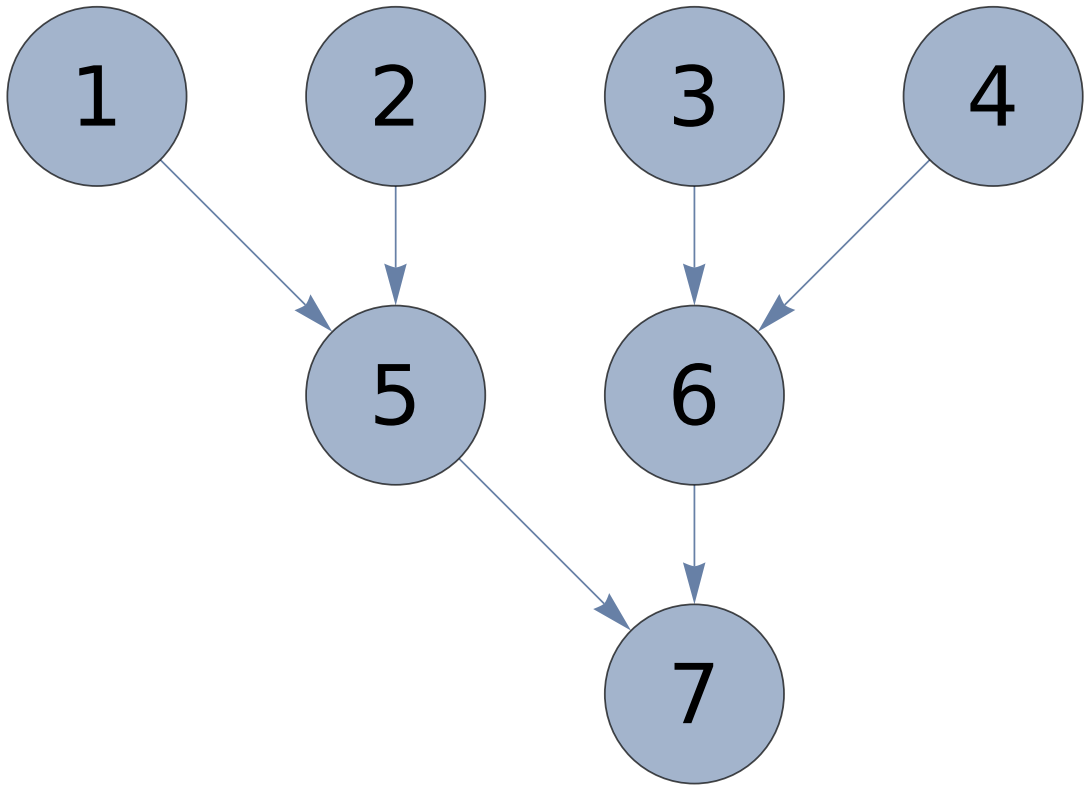
In particular, for 5 inputs and 11 gates, the number of chains is:

$$1887493942318359375000000000 \approxeq 1.89 \times 10^{25}$$

This is about **60x more** than the number of floating-point operations used to train GPT-3 (3.14×10^{23}).

Room for improvement I: canonical ordering

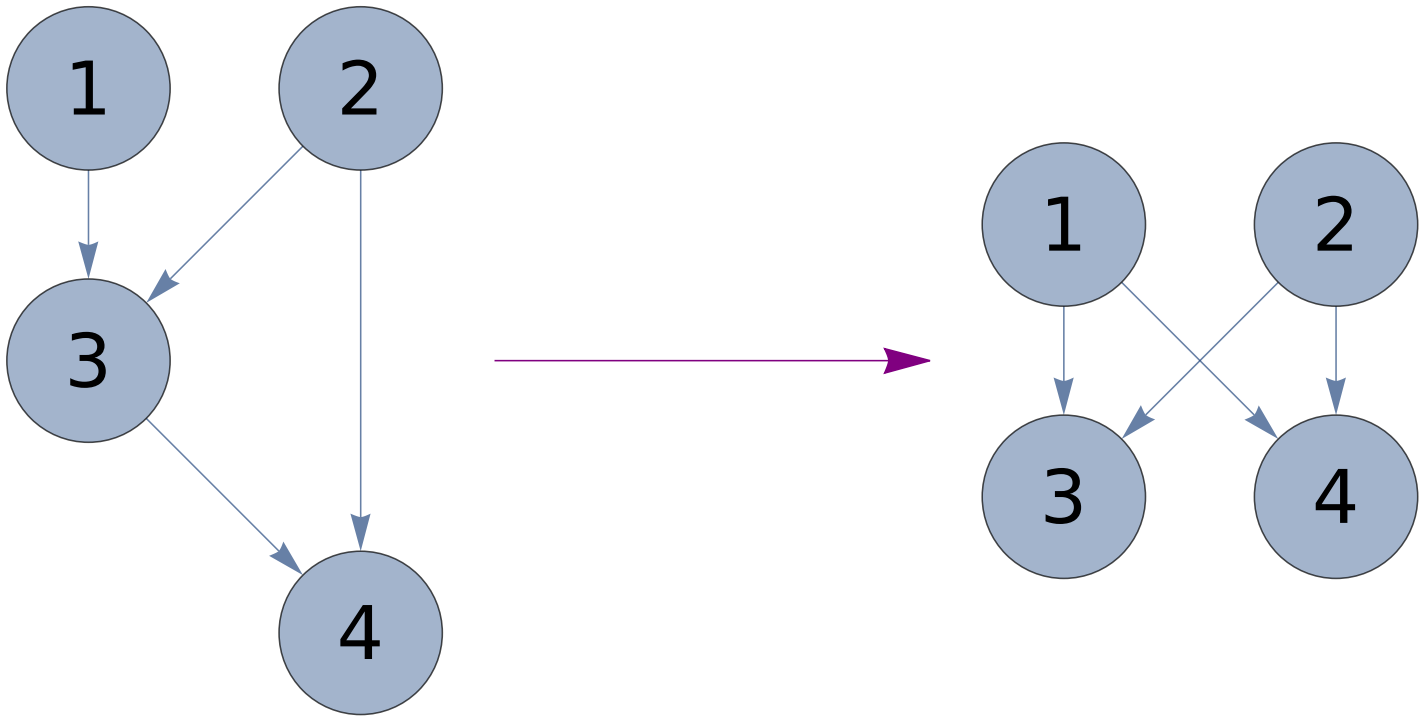
In many cases, a single DAG can correspond to multiple Boolean chains:



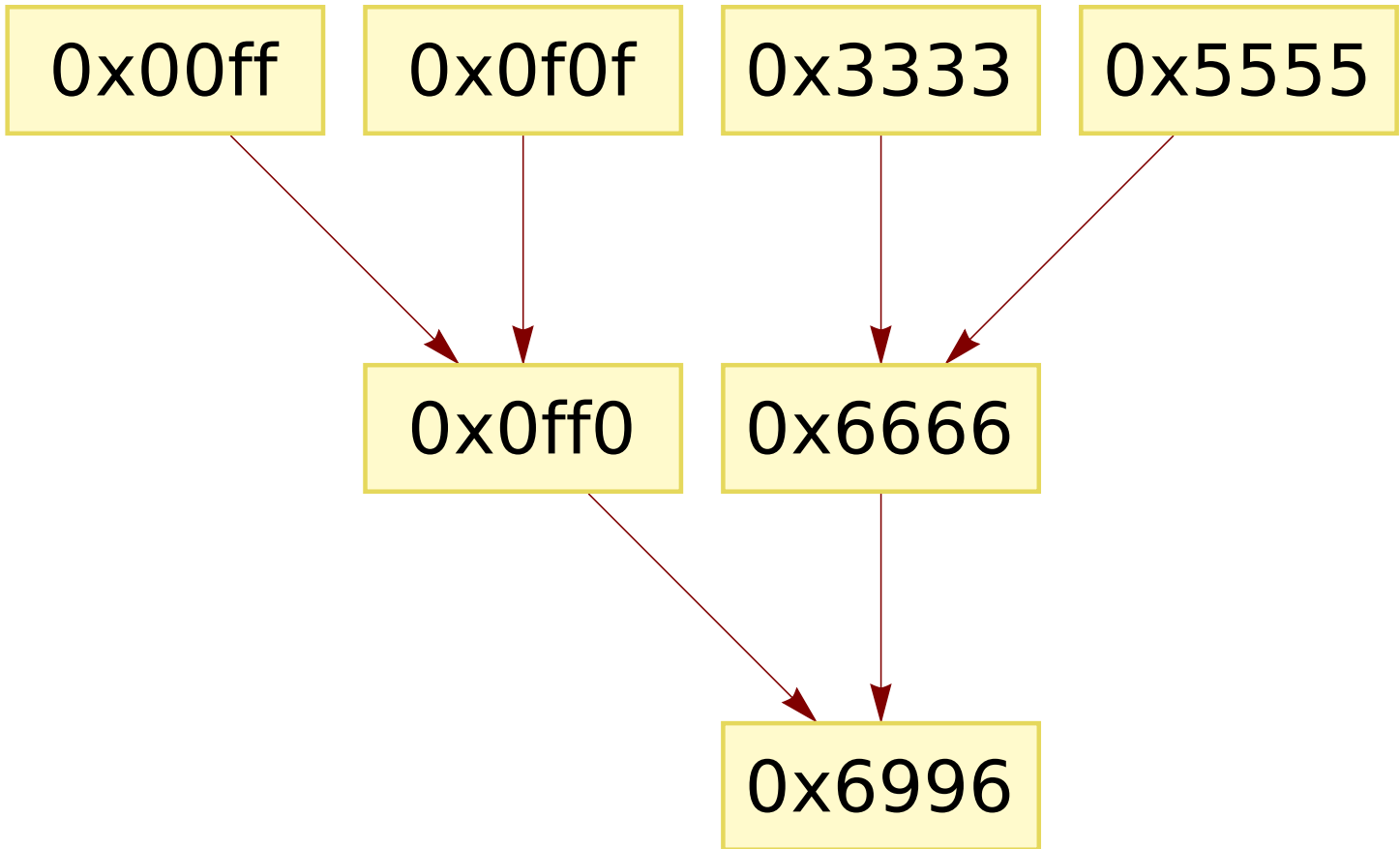
For example, the order of computing variables 5 and 6 could be swapped.

Room for improvement II: triangle-free

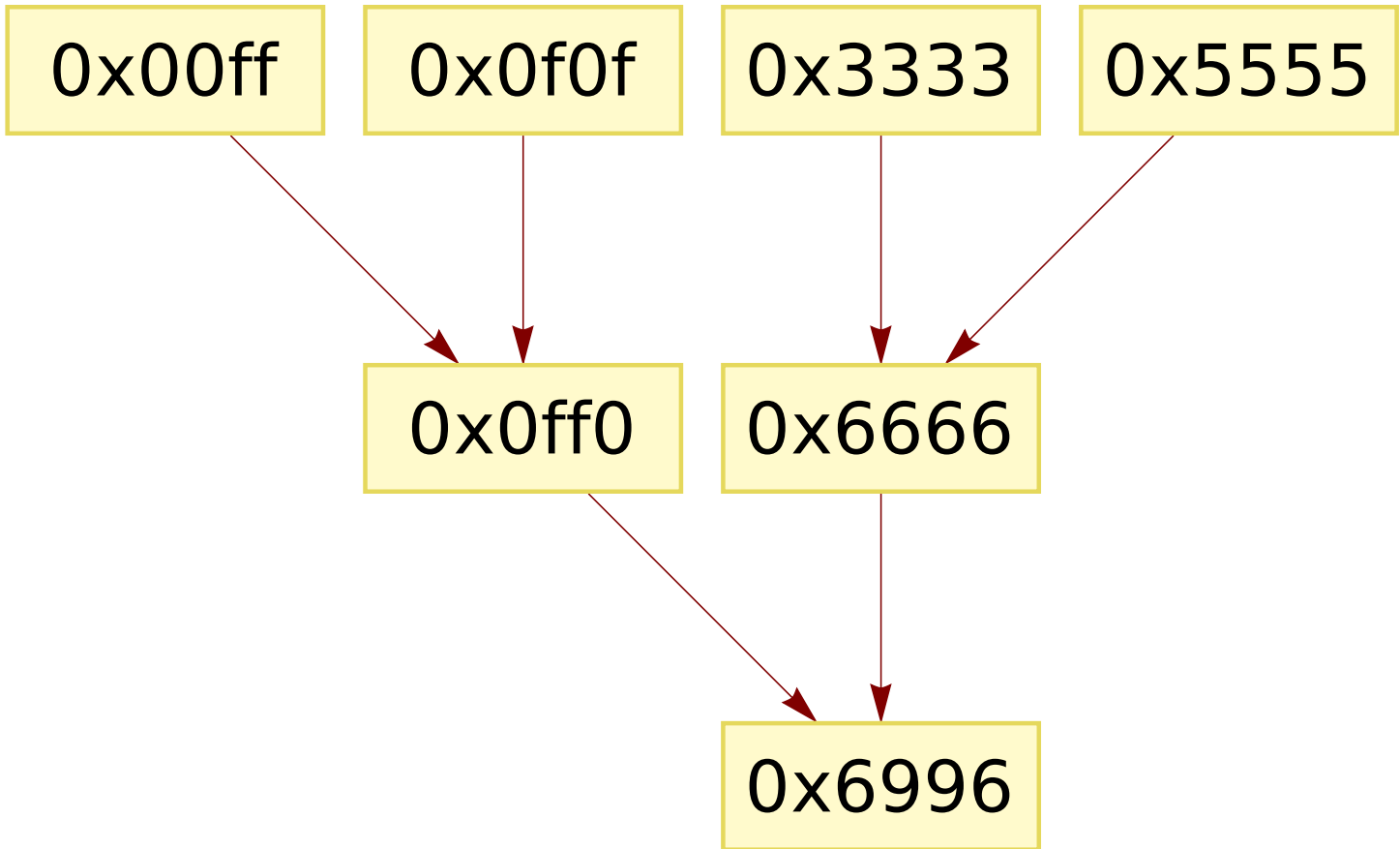
Also, triangles can be removed from our DAG without loss of generality:



Key insight



Key insight



Simply represent this as the set {`0x0ff0,0x6666,0x6996`}.

Reconstruction of chains from tt-sets

By performing a **backtracking search**, we can reconstruct all possible chains corresponding to a tt-set.

- Beginning with the set of truth tables corresponding to inputs, try applying all possible gates which result in a truth table belonging to the tt-set.

Reconstruction of chains from tt-sets

By performing a **backtracking search**, we can reconstruct all possible chains corresponding to a tt-set.

- Beginning with the set of truth tables corresponding to inputs, try applying all possible gates which result in a truth table belonging to the tt-set.
- We enforce the ‘canonical ordering’ and ‘triangle free’ properties at all times.

Reconstruction of chains from tt-sets

By performing a **backtracking search**, we can reconstruct all possible chains corresponding to a tt-set.

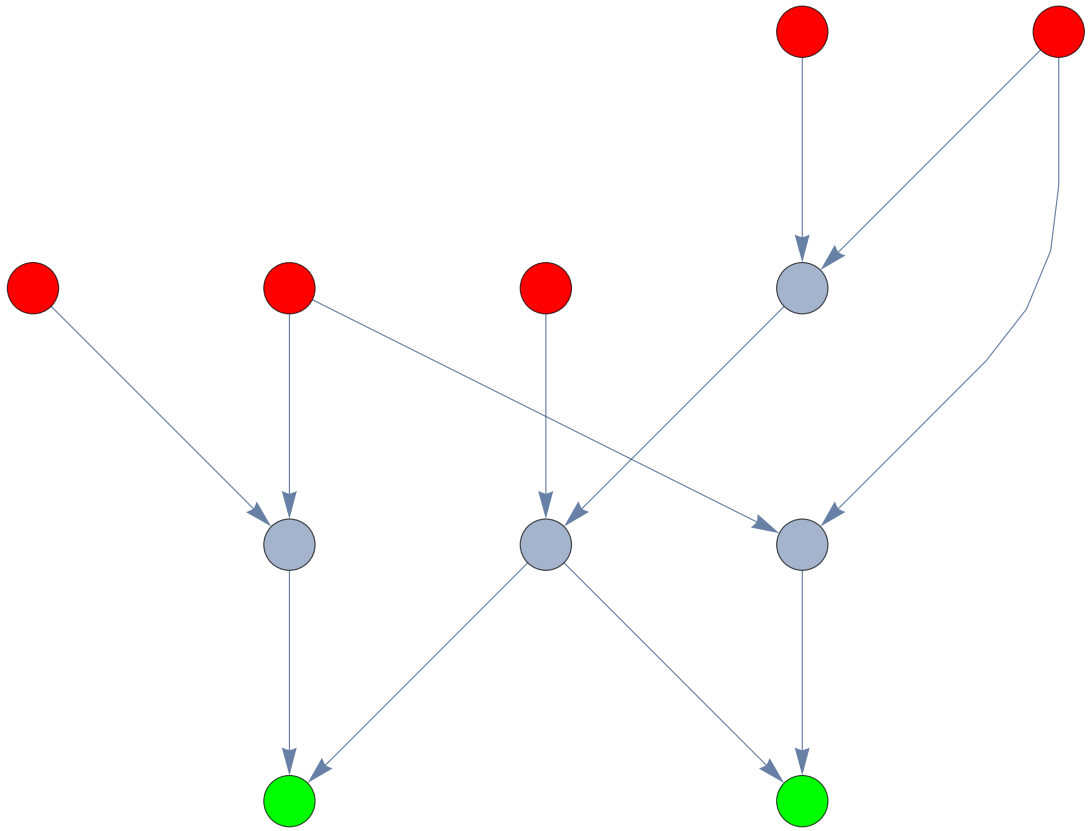
- Beginning with the set of truth tables corresponding to inputs, try applying all possible gates which result in a truth table belonging to the tt-set.
- We enforce the 'canonical ordering' and 'triangle free' properties at all times.
- The total runtime is $O(Sn^3)$ where S is the number of chains outputted by the algorithm.

Ends of tt-sets

A tt-set T is *attainable* if it corresponds to at least one chain.

Ends of tt-sets

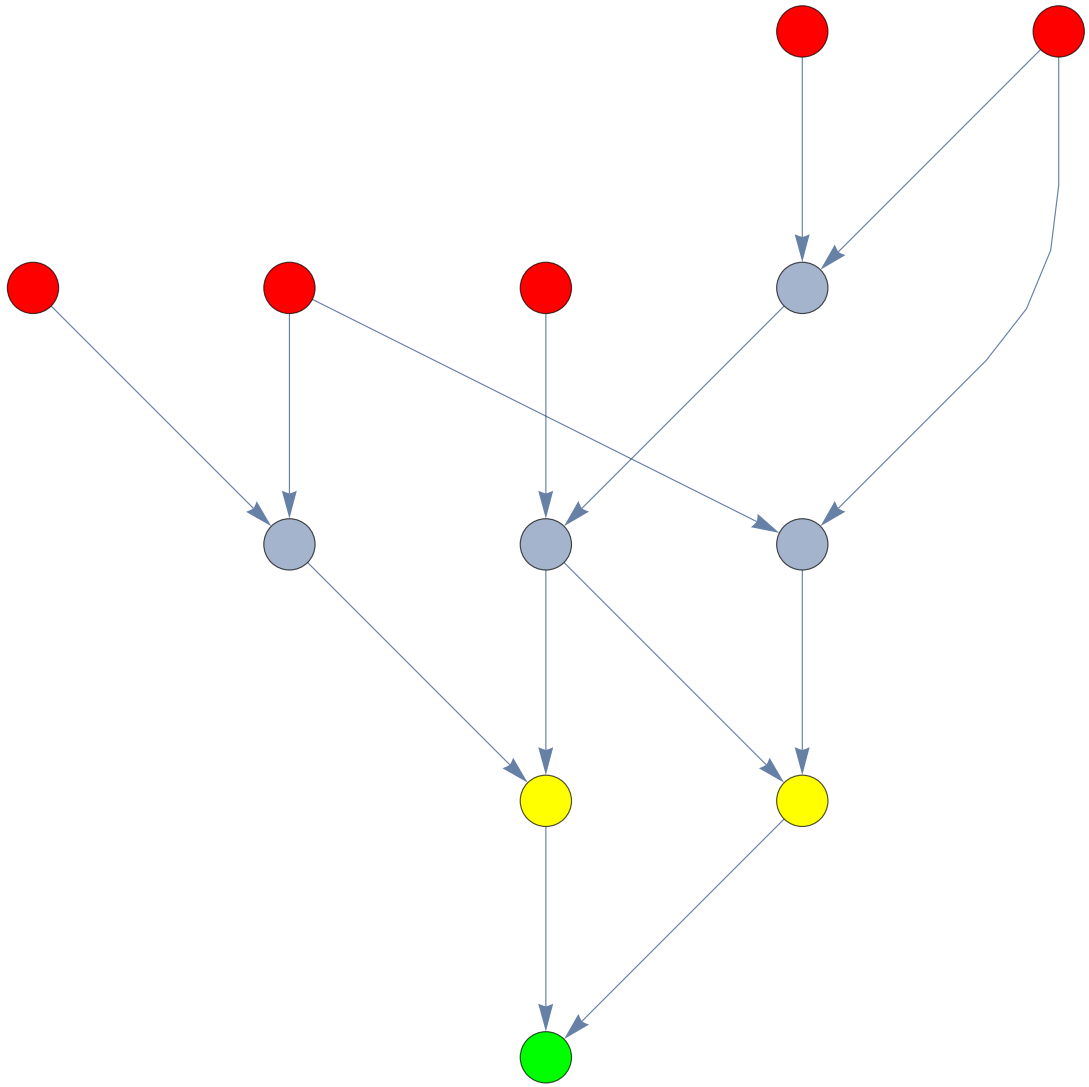
A tt-set T is *attainable* if it corresponds to at least one chain.



If $T \setminus \{e\}$ is still attainable, then we say that e is an *end* of T.

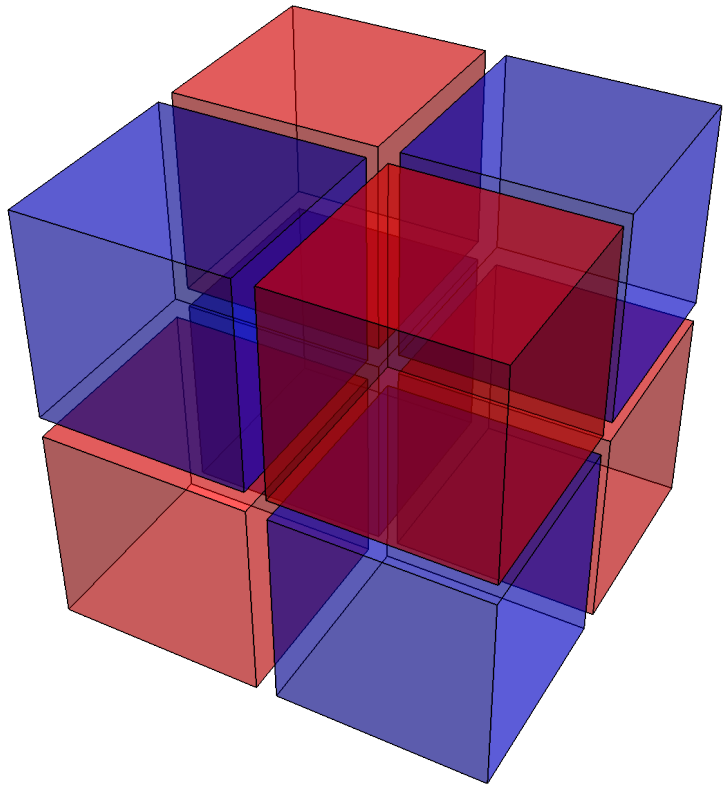
Appending a gate

Appending a gate cannot cause the number of ends to decrease by more than 1.



Symmetries

Recall that we have a large symmetry group at play.



The search becomes much faster if we only store **canonical** tt-sets.

Breadth-first search

Define $N(T)$ to be the set of possible 'next functions':

$$N(T) = \{x \circ y : x, y \in T \cup I \text{ and } \circ \in \mathcal{O}\} \setminus (T \cup I \cup \{0\})$$

Breadth-first search

Define $N(T)$ to be the set of possible 'next functions':

$$N(T) = \{x \circ y : x, y \in T \cup I \text{ and } \circ \in \mathcal{O}\} \setminus (T \cup I \cup \{0\})$$

We construct all canonical attainable tt-sets by induction on size:

- Define $X_0 = \{\{\}\}$ to be the set containing the empty tt-set.
- Define
  $X_{n+1} = \{\text{canonicalise}(T \cup \{e\}) : T \in X_n \text{ and } e \in N(T)\}$

Breadth-first search

Define $N(T)$ to be the set of possible 'next functions':

$$N(T) = \{x \circ y : x, y \in T \cup I \text{ and } \circ \in \mathcal{O}\} \setminus (T \cup I \cup \{0\})$$

We construct all canonical attainable tt-sets by induction on size:

- Define $X_0 = \{\{\}\}$ to be the set containing the empty tt-set.
- Define
  $X_{n+1} = \{\text{canonicalise}(T \cup \{e\}) : T \in X_n \text{ and } e \in N(T)\}$

For a given search depth, we can also discard tt-sets with too many ends.

How does previous work differ?

Previous approaches directly searched the (much larger) space of DAGs rather than canonical attainable tt-sets.

How does previous work differ?

Previous approaches directly searched the (much larger) space of DAGs rather than canonical attainable tt-sets.

- **Donald Knuth (2011)** used 'top-down' and 'bottom-up' reductions together with brute-force searches for 'special' (irreducible) DAGs.
- Knuth was interested in determining the minimum cost for each function, rather than finding all optimal chains.

How does previous work differ?

Previous approaches directly searched the (much larger) space of DAGs rather than canonical attainable tt-sets.

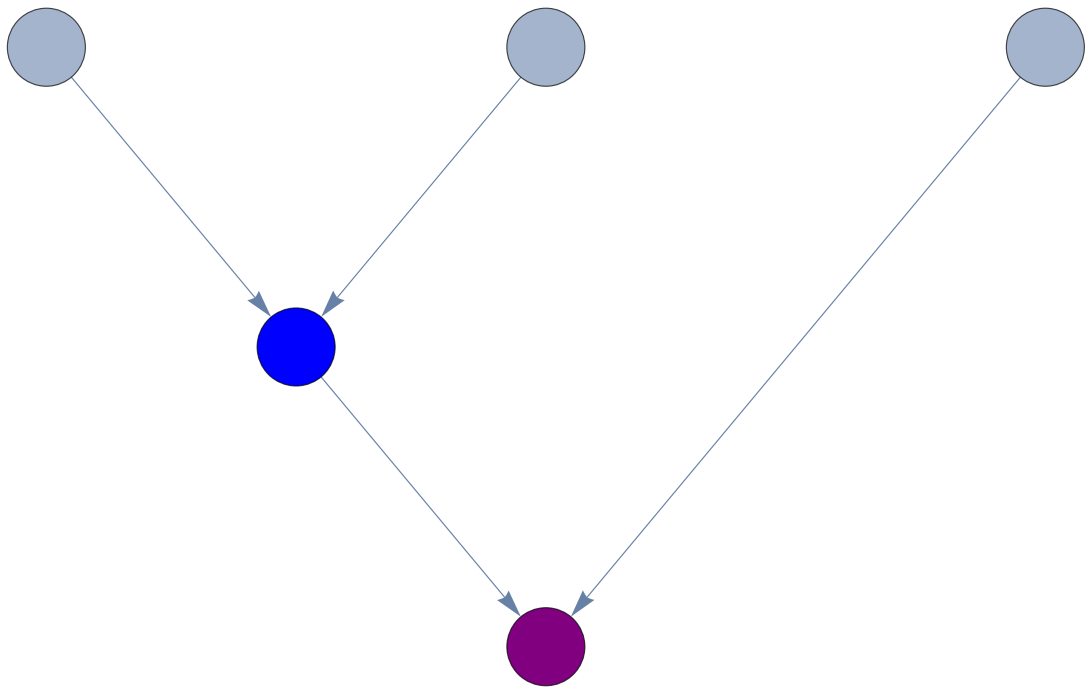
- **Donald Knuth (2011)** used 'top-down' and 'bottom-up' reductions together with brute-force searches for 'special' (irreducible) DAGs.
- Knuth was interested in determining the minimum cost for each function, rather than finding all optimal chains.

Haaswijk, Soeken, Mishchenko, and De Micheli (2018) used SAT solvers to search for DAGs of a particular topology implementing a given function.

Reducing constant factors

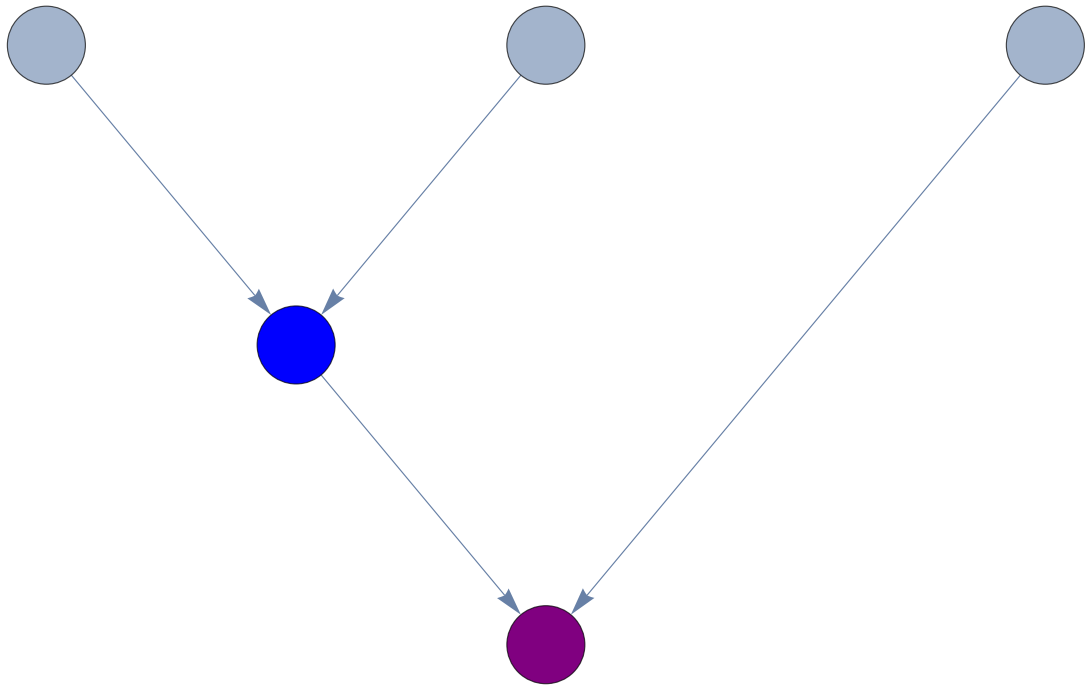
Canonicalising a tt-set is expensive. To do an 11-gate search, we do the following:

- Use the algorithm to compute all tt-sets in X_9 with ≤ 3 ends.
- Brute-force all possibilities for the last two gates.



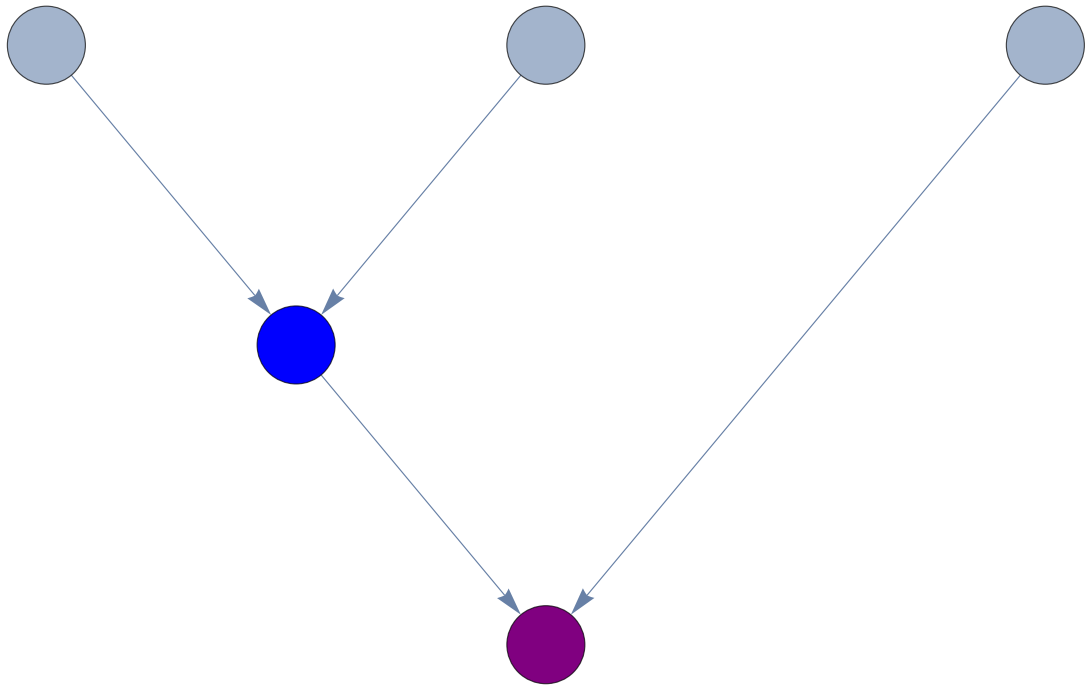
Checking optimality

We want to check optimality, i.e. determine whether the purple vertex cannot be attained with fewer gates.



Checking optimality

We want to check optimality, i.e. determine whether the purple vertex cannot be attained with fewer gates.



We use a 256 MB array (one bit for each of the 2^{31} normal functions) to indicate whether they've previously been constructed with lower cost.

Practical problems

Even a depth-9 search does not fit in memory (11 terabytes uncompressed).

n	tt-sets	uncompressed size	compressed size
6	3 million	73 MB	5 MB
7	130 million	3.6 GB	159 MB
8	6.2 billion	197 GB	6.4 GB
9	(310 billion)	(11 TB)	

Practical problems

Even a depth-9 search does not fit in memory (11 terabytes uncompressed).

n	tt-sets	uncompressed size	compressed size
6	3 million	73 MB	5 MB
7	130 million	3.6 GB	159 MB
8	6.2 billion	197 GB	6.4 GB
9	(310 billion)	(11 TB)	

We need to somehow **partition the search space** into manageable chunks, but this is not particularly easy.

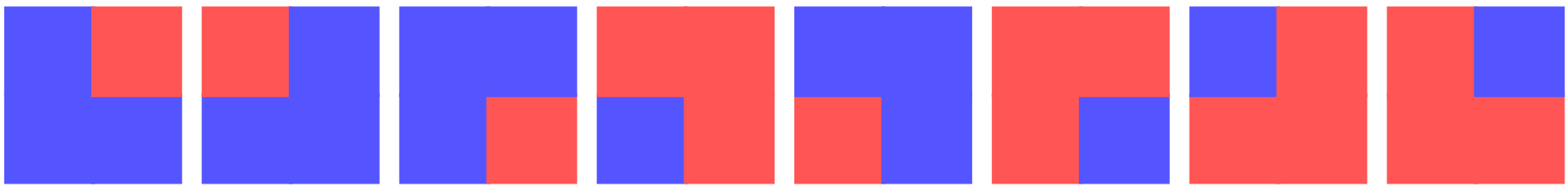
Invariants and signatures

An **invariant** is an easily-computable function f from truth tables to an arbitrary finite set which is constant on equivalence classes.

Invariants and signatures

An **invariant** is an easily-computable function f from truth tables to an arbitrary finite set which is constant on equivalence classes.

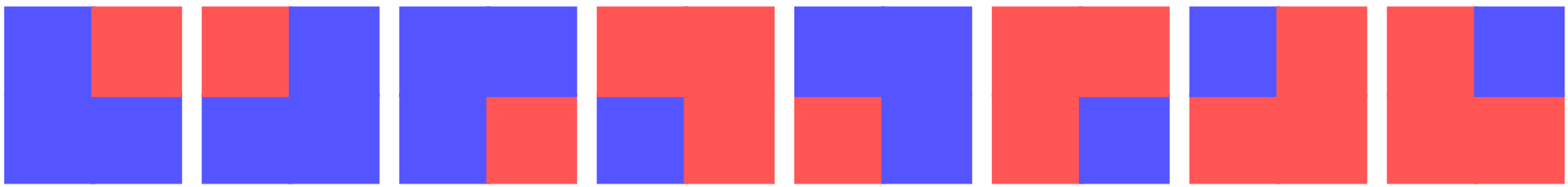
For example, the **bias**, or absolute difference between the number of '1's and '0's in the truth table, is an invariant.



Invariants and signatures

An **invariant** is an easily-computable function f from truth tables to an arbitrary finite set which is constant on equivalence classes.

For example, the **bias**, or absolute difference between the number of '1's and '0's in the truth table, is an invariant.



Given an invariant f, the **signature** of a tt-set T is the **multiset** $\{f(x) : x \in T\}$.

Distributing the workload

Proceed as before, but partition the creation of each X_n into separate tasks, one for each size-n signature.

Distributing the workload

Proceed as before, but partition the creation of each X_n into separate tasks, one for each size-n signature.

The task corresponding to a size-n signature S reads the files saved behind by the tasks corresponding to size-$(n-1)$ signatures $S \setminus \{s\}$ for each $s \in S$.

Distributing the workload

Proceed as before, but partition the creation of each X_n into separate tasks, one for each size-n signature.

The task corresponding to a size-n signature S reads the files saved behind by the tasks corresponding to size-$(n-1)$ signatures $S \setminus \{s\}$ for each $s \in S$.

We avoid saving tt-sets to disk in the last level of the search tree ($n = 9$), because there are no downstream tasks to consume them.

Results

We applied this depth-11 exhaustive search procedure to two search problems:

- 5-input 1-output functions (616126 equivalence classes);
- 4-input 2-output functions (1476218 equivalence classes).

Each of the two searches took a few days on an AWS **r5a.24xlarge** instance (96 virtual cores + 768 GB memory), costing $< \$1000$.

Results

We applied this depth-11 exhaustive search procedure to two search problems:

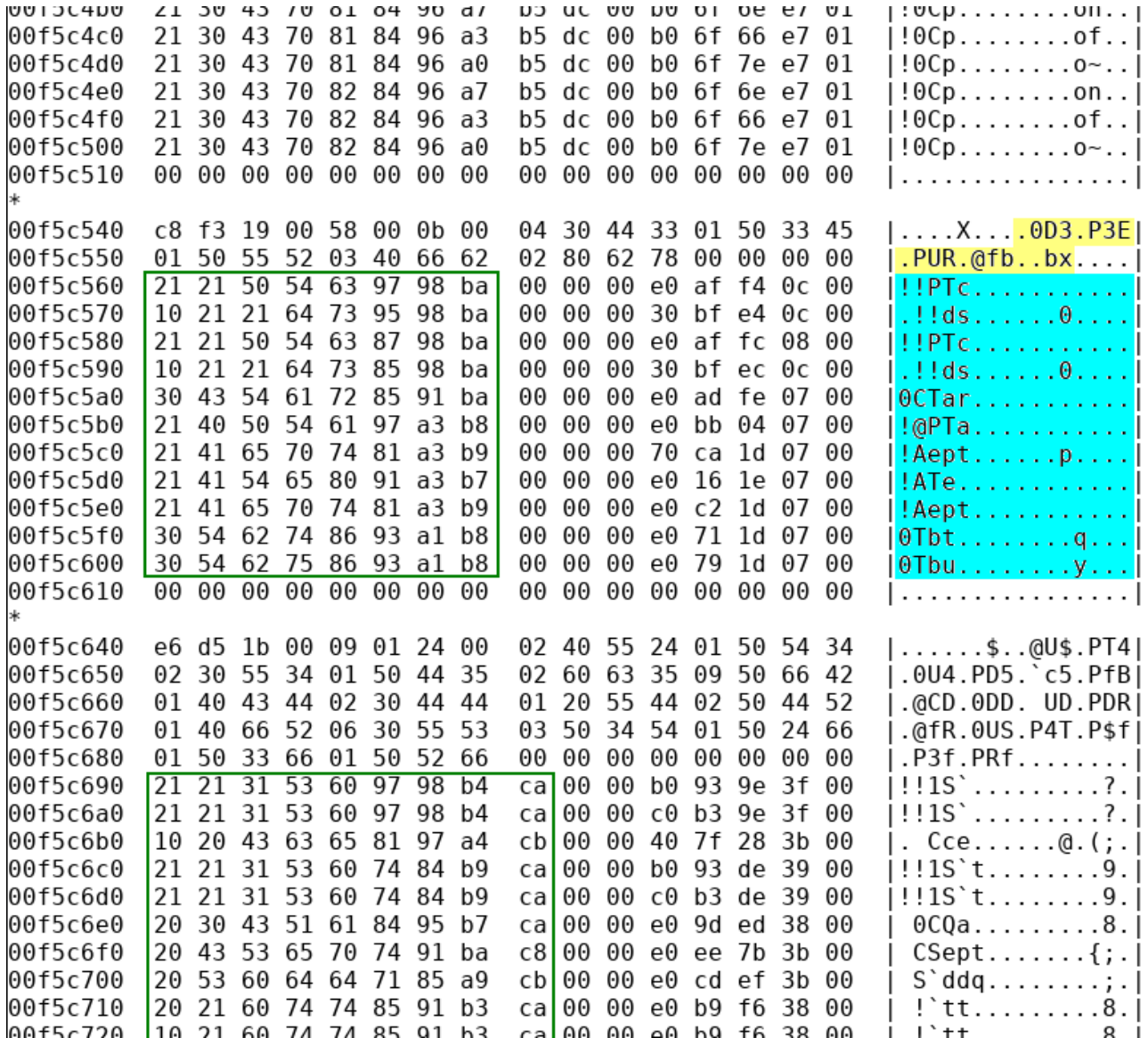
- 5-input 1-output functions (616126 equivalence classes);
- 4-input 2-output functions (1476218 equivalence classes).

Each of the two searches took a few days on an AWS **r5a.24xlarge** instance (96 virtual cores + 768 GB memory), costing $< \$1000$.

But how do we use these results?

Building a database

For each equivalence class of functions, we took all Boolean chains with **minimum cost** and **delay on the Pareto frontier**.



The k-perfect hashtable

We hash a canonical 5-input 1-output truth table using the following pair of functions:

```
auto h1 = (x ^ (x >> 7) ^ (x >> 4)) & 0x1ffff;
auto h2 = (x ^ (x >> 7) ^ (x >> 24)) & 0x1ff;
```

The k-perfect hashtable

We hash a canonical 5-input 1-output truth table using the following pair of functions:

```
auto h1 = (x ^ (x >> 7) ^ (x >> 4)) & 0x1ffff;
auto h2 = (x ^ (x >> 7) ^ (x >> 24)) & 0x1ff;
```

The function h_1 is 15-perfect, mapping at most 15 of the 616124 canonical nontrivial truth tables to the same bucket.

The function h_2 is 1-perfect (injective) within each bucket.

Looking up a canonical truth table

By squeezing each bucket into 64 bytes, it is possible to lookup the cost and location of any canonical function by retrieving a single cache line:

```
uint32_t get_cl_and_cost(const uint32_t* db, uint32_t x, int n_bits) {

    uint32_t clc = 0;
    uint32_t hash1 = hh::boolchains::hthash1(x, n_bits);
    uint32_t hash2 = hh::boolchains::hthash2(x);

    uint32_t cache_line[16];

    memcpy(cache_line, &(db[hash1 << 4]), 64);

    for (size_t i = 0; i < 15; i++) {
        uint32_t element = cache_line[i];
        if ((element & 0x1ff) == hash2) {
            uint32_t cl = ((element >> 9) & 0x7fff) + cache_line[15];
            uint32_t cost = (element >> 24);
            clc = (cl << 4) | cost;
            break;
        }
    }

    return clc;
}
```

Overview

To find all chains for a (not necessarily canonical) function:

- **Canonicalize it**: this would naively take 10 μs, but we obsessively reduced it down to **200 ns**.

Overview

To find all chains for a (not necessarily canonical) function:

- **Canonicalize it**: this would naively take 10 μs, but we obsessively reduced it down to **200 ns**.
- **Lookup the cost and location**: this is a single cache line retrieval plus a handful of instructions, so takes around **100 ns**.

Overview

To find all chains for a (not necessarily canonical) function:

- **Canonicalize it**: this would naively take 10 μs, but we obsessively reduced it down to **200 ns**.
- **Lookup the cost and location**: this is a single cache line retrieval plus a handful of instructions, so takes around **100 ns**.
- **Iterate over optimal chains**: this step takes variable time, depending on the number of chains, but they're contiguous in memory so this is again reasonably fast.