Searching for optimal Boolean chains

Adam P. Goucher

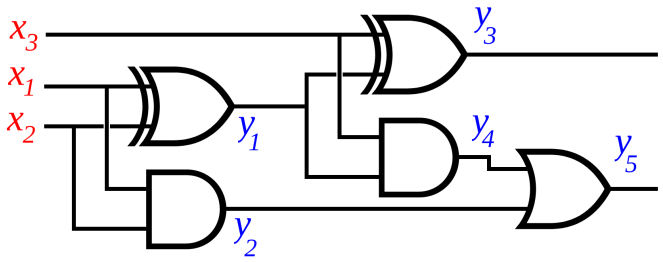
February 28, 2023

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.

This ensures that everything is well-defined as a function of the inputs; there can't be any cyclic references. Structurally the variables form a directed acyclic graph, where the inputs have in-degree 0 and everything else has in-degree 2.

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:

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

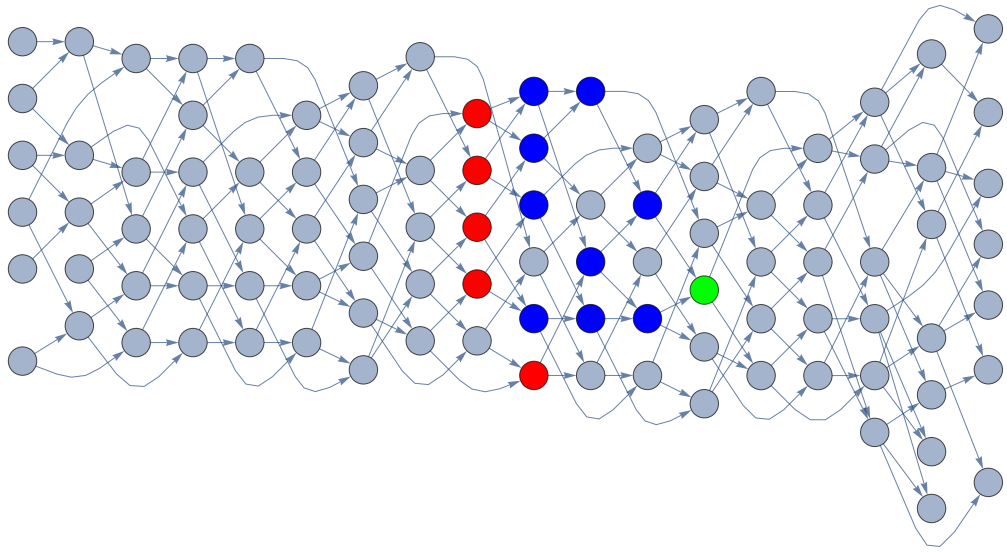
> From left to right, these are the three symmetric gates XOR, AND, and OR, and two asymmetric gates, both of which are just an AND gate with one of the two inputs complemented.

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

> These are processor instructions which are capable of performing a Boolean operation in a vectorised manner on vectors of 256 bits. More advanced vector processors, such as CPUs with AVX-512 and modern GPUs, have even wider vector registers and support for arbitrary 3-input Boolean operations.
>
> This means that if you can represent your computation as a Boolean circuit, then you can evaluate it in a massively parallel manner on many distinct sets of inputs. This strategy is called *bitslicing*, and I've used it before for accelerating a regular expression engine and cellular automata library on both the CPU and GPU.
>
> The speed of a bitsliced computation will depend principally on the number of gates and the amount of instruction-level parallelism we can use, so it's advantageous to optimise Boolean circuits. How do we do that?

Rewriting

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

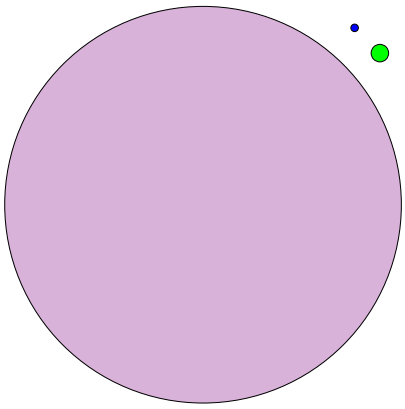


Tools such as ABC also include a mixture of other optimisation passes, such as refactoring, balancing, and SAT-sweeping, but for this talk we'll focus on rewriting.

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.

We've included a to-scale illustration of the number of equivalence classes of functions handled by each of these approaches. In blue we have the 222 equivalence classes of 4-input functions; in green we have Li and Dubrova's 1200 equivalence classes of 5-input functions; in lilac, we have our collection. But what exactly do we mean by equivalence classes, and what's the reason for only computing optimal chains for all but one of them, rather than all of them?

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 $2^{k+\ell}(k!)(\ell!)$.

If two functions are equivalent in this sense, then a Boolean chain for one of these functions can be easily converted into a Boolean chain for the other.

For example, there are a total of $2^3 = 8$ functions that are equivalent to the 2-input AND gate, by complementing any combination of its inputs and output. We say that these eight functions form an *equivalence class*. Let's write down the truth tables of these eight functions:

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

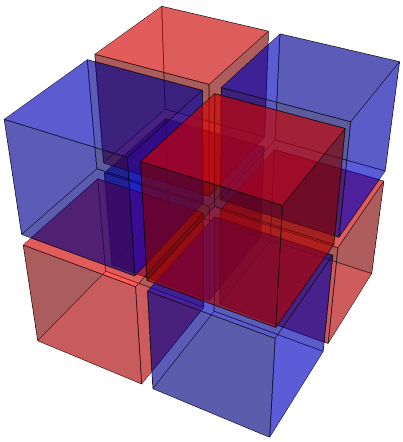
We describe the lexicographically first truth table in an equivalence class as **canonical**.

In this case, it's the truth table 0001. By only concentrating on canonical functions and finding optimal Boolean chains for those, we significantly reduce the amount of storage space that we'll need for a database of optimal Boolean chains.

We've introduced another idea here: by representing functions as truth tables which are in turn encoded in binary as integers, we can store functions on a computer very efficiently: a 5-input truth table, for example, can be represented as a 32-bit integer, because $2^5 = 32$. Lexicographical comparison of truth tables is then just numerical comparison, which is extraordinarily cheap, and we can similarly compute the effects of applying our primitive operations to existing truth-tables by using bitwise instructions.

Anyway, how many equivalence classes do we have, in general?

Counting equivalence classes



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

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

The colourful cube here is the truth table of the 3-input XOR function. Different ways to permute and complement the inputs of the function simply correspond to rotating and reflecting this cube, and negating the output corresponds to swapping the two colours.

The dominant term is just the total number of truth tables divided by the size of the symmetry group. This doesn't quite give the correct answer, because it doesn't account for truth tables which have nontrivial symmetries, such as the 3-input XOR function visualised above. In particular, it doesn't even give you an integer! Burnside's lemma gives the appropriate additional correction terms to compute the exact answer.

If you apply this with $k = 5$ and $\ell = 1$, you get 616126.

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**

Here are the number of equivalence classes for $k = 5, \ell = 1$ and for $k = 4, \ell = 2$, categorised according to the *cost*, or length of the minimal Boolean chain realising a function in that class.

The middle column of this table, namely the costs for the 5-input 1-output functions, was computed by Donald Knuth in 2011 using a couple of computing clusters, one at Stanford and another borrowed from Sun Research. We both independently verified these results and computed the right-hand column as a side-effect of our own search methodology, which is more cost-effective: we managed to do it in a few days on a budget of around $1000. Moreover, rather than just getting the cost of each function, we go much further and compute all optimal Boolean chains.

This is where the number 616126 comes from, and the reason that we only compute all optimal chains for 616125 of these equivalence classes is that the remaining one has cost 12, and a depth-12 exhaustive search would take about two orders of magnitude longer to complete than a depth-11 exhaustive search.

Let's get a handle on how large the search space is.

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:

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

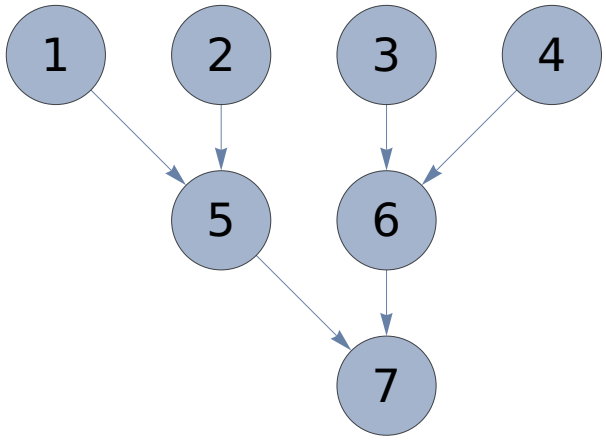
This is about **60x more** than the number of floating-point operations used to train GPT-3 (3.14×10^{23}).

Note that this is only a numerical comparison to give a rough idea of the size of the search space that we're dealing with. Evaluating the truth table of a Boolean chain and checking whether it's optimal is much more expensive than doing a single floating-point operation, so '60 times GPT-3' is a massive *underestimate* of the cost of brute-forcing this space.

In the remainder of the talk, we'll discuss some ideas that manage to reduce the size of this search by about 9 or 10 orders of magnitude, as well as how to make this search efficiently distributable and how to maximise the number of nodes of the search tree that we can traverse per unit time.

Room for improvement I: canonical ordering

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

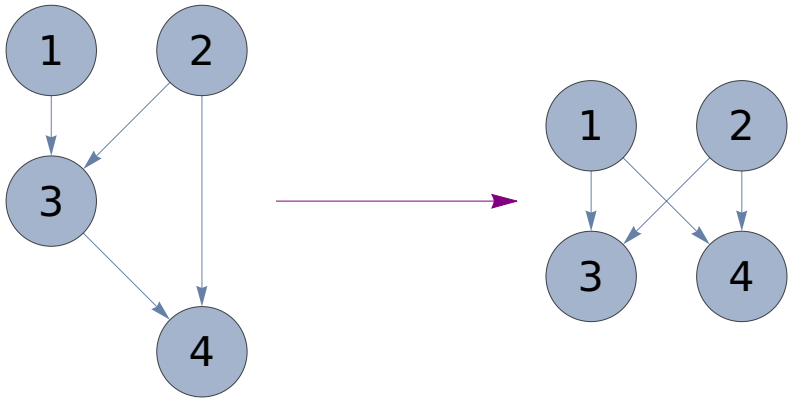


For example, the order of computing variables 5 and 6 could be swapped.

There's a way to make the ordering of a DAG canonical: essentially we inductively order the nodes in the DAG by looking at the relative orders of the inputs to that node, using the operation $\circ \in \mathcal{O}$ as a tie-breaker.

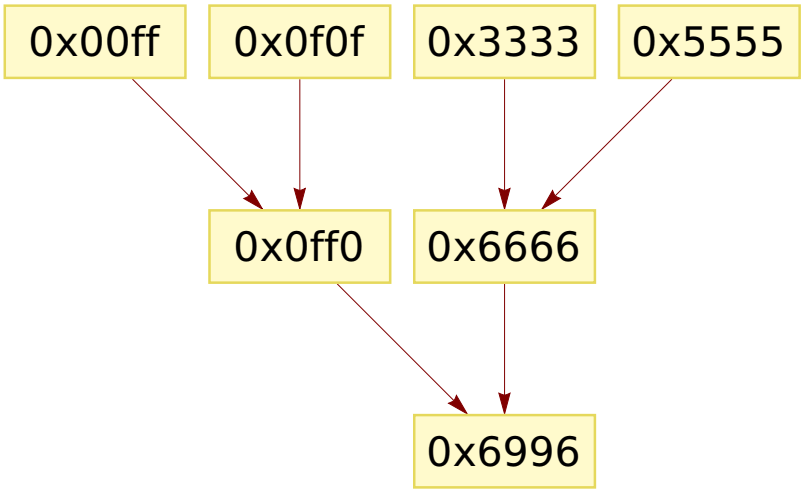
Room for improvement II: triangle-free

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



In particular, if you have a situation like the DAG on the left, then we can replace it with the equivalent circuit on the right which has strictly better instruction-level parallelism.

Key insight



Here's an example of a Boolean chain which computes the XOR of four inputs. We've annotated each node by its truth table, written in hexadecimal rather than binary for brevity. Now we're going to do something quite drastic, which is to forget the whole DAG structure and just represent this by the set of its non-input truth tables.

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

This may feel like it throws away a lot of information, and it does, because many different Boolean chains correspond to the same set of truth tables, henceforth abbreviated to *tt-set*. But we'll see that we can efficiently recover all of these chains from just 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.

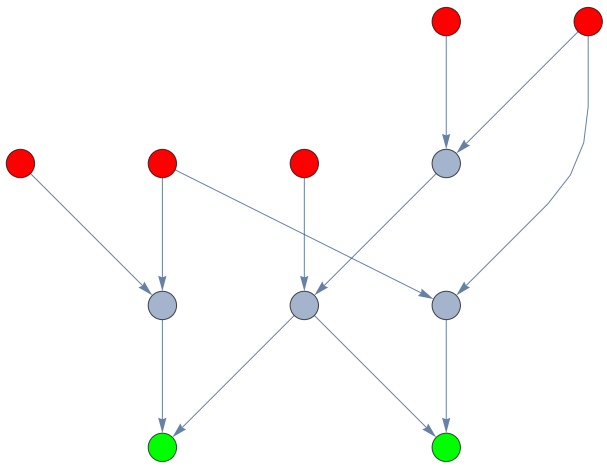
- Beginning with the set of truth tables corresponding to inputs, try applying all possible gates which result in a truth table belonging to the tt-set.
- We enforce the 'canonical ordering' and 'triangle free' properties at all times.
- The total runtime is $O(Sn^3)$ where S is the number of chains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

The way that we do this is to first construct Cayley tables telling us which elements in the tt-set can be expressed by applying one of our primitive operations to a pair of other elements in the set. Given these Cayley tables, we can then very efficiently perform a depth-first search by applying a gate at a time. The amount of time that we spend in each node of the search tree is $O(n^2)$, so a single path through this search tree takes time $O(n^3)$, and because we never reach any 'dead ends' in the search tree, this gives an upper bound of $O(Sn^3)$ for the whole process.

Ends of tt-sets

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

We can just use the previous algorithm to determine attainability: either it completes without producing any output chains, in which case we know that the set is unattainable. As soon as it does yield the first output, we can prematurely interrupt the algorithm because it has verified that the tt-set is attainable. This takes $O(n^3)$ time.



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

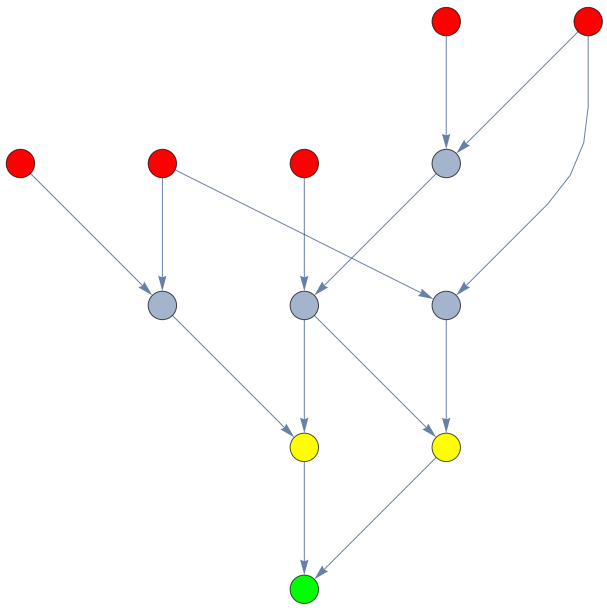
We can therefore determine the ends of a tt-set in time $O(n^4)$ by just checking the attainability of each tt-set obtained by removing one of the elements.

Why do we call these 'ends'? The reason is that they're exactly the functions that appear last in at least one Boolean chain corresponding to our tt-set.

The reason that they're interesting is that if we have an *optimal* chain for a k-input ℓ-output function, then the ends must form a subset of the outputs: if there's an end that isn't one of our outputs, we can remove it from the tt-set and find a cheaper Boolean chain. As such, each optimal chain has at most ℓ ends. It could have fewer, because one of the outputs might be a dependency of another output.

Appending a gate

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

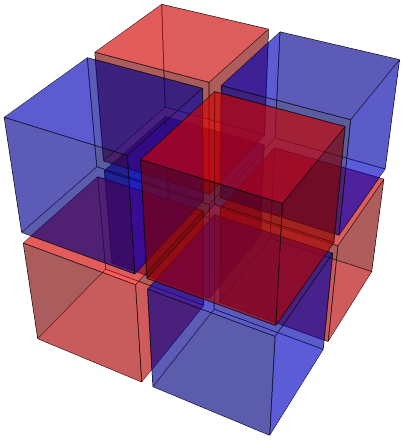


Specifically, each new gate that we add introduces an end, namely its output, and consumes at most two ends, namely its inputs.

This notion of ends allows us to backtrack when searching for tt-sets corresponding to optimal chains. If we have a tt-set containing c gates, and it contains more than $\ell + n - c$ ends, then it cannot possibly occur as a subset of a tt-set corresponding to an optimal chain of length $\leq n$. Effectively we have too many ends and not enough remaining gates left to tie them all together.

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':

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

In other words, it's the set of nontrivial functions that are not already in the tt-set, but can be obtained in a single step.

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.

- **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.

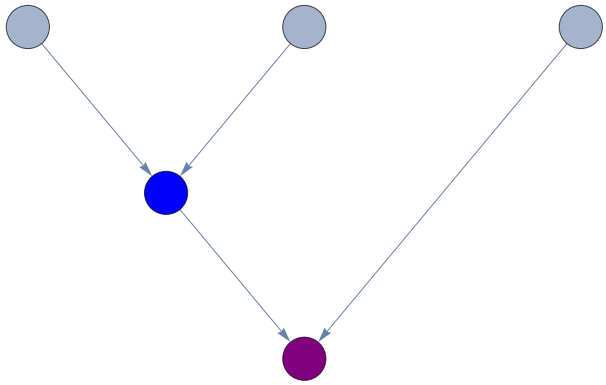
This approach works well if you want to find a Boolean chain for a specific function, because it can work backwards from the end result as well as forwards from the inputs. But if you want to find all optimal Boolean chains for all equivalence classes, then you'd need to repeat this lots of times, one for each equivalence class of target function and choice of circuit topology, and that would take infeasibly long.

We've described our search algorithm, but there are a few practical considerations that need addressing.

Reducing constant factors

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

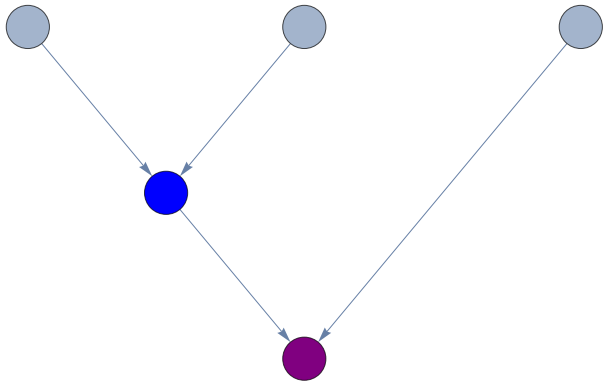
- Use the algorithm to compute all tt-sets in X_9 with ≤ 3 ends.
- Brute-force all possibilities for the last two gates.



The whole point of canonicalising the tt-sets is to deduplicate identical branches of our search tree so that we don't unnecessarily repeat work. We do this for the first 9 layers, but not for the last 2, where the cost of deduplicating the work outweighs the benefit.

Checking optimality

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



We could canonicalise the truth table and check whether it's already been constructed with lower cost. But, as we discussed, canonicalisation is expensive because the symmetry group is so large. How can we avoid it?

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.

Adding this flat bit-set cache in front of the expensive checking code caused it to run approximately 20x faster.

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)	

The figures in this table are measured, apart from the last row, which is estimated by extrapolation. The reason for the compressed size is that we use LZMA when saving search progress to disk, but in memory the data is uncompressed.

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

If we had an ordinary search tree, then the branches of this tree could be mapped to different subtasks. But we ensure that we deduplicate equivalent tt-sets at each level, so different branches of the tree can converge into a single branch, and that makes it much harder to cleanly

divide into tasks.

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.

A constant function is a valid, albeit useless, invariant. In the other extreme, the finest invariant is the canonicalisation function which outputs the canonical form of a truth table; the problem with that is that it's too expensive to evaluate. We'd ideally like something between these two extremes.

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

This is easy to compute: the popcount processor instruction gives the number of '1's in the truth table, and a handful of arithmetic functions suffice to calculate the number of '0's and determine the absolute difference.

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

So, how does this help with our problem of partitioning the search space?

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.

This way, we only need to save a few gigabytes of compressed data instead of hundreds of gigabytes.

Results

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

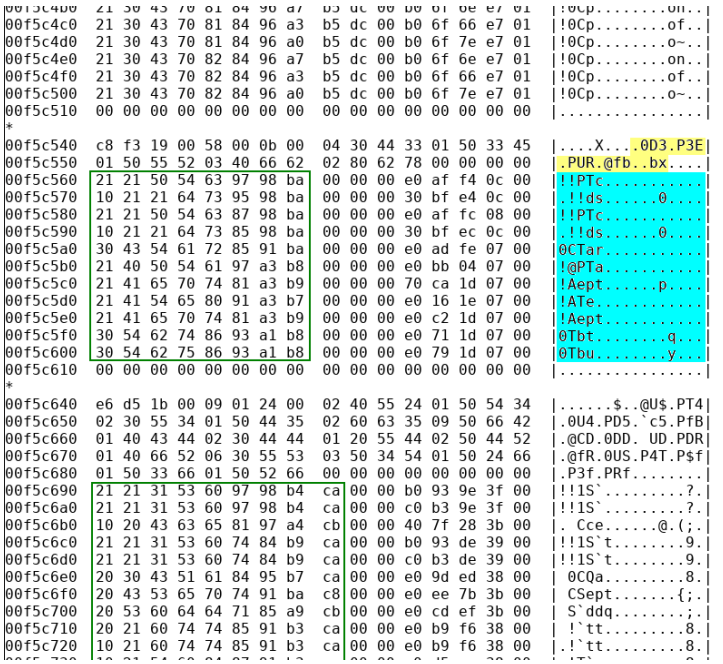
- 5-input 1-output functions (616126 equivalence classes);
- 4-input 2-output functions (1476218 equivalence classes).

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 delay of a Boolean chain is the k-tuple of path lengths from each of the inputs to the output, and it's on the Pareto frontier if there isn't another chain which strictly dominates it, in the sense of having delay that's at least as good with respect to all inputs, and strictly better for at least one of those inputs.

The section in the middle of this hexdump corresponds to one of the equivalence classes of Boolean functions. The reason for the all-zeroes rows above and below this section is to ensure that it's aligned on a 64-byte cache line boundary.

The first four bytes, 0x0019f3c8, specify the canonical truth table for this equivalence class. The next four bytes encode the cost of the function (8), number of delay patterns on the Pareto frontier (5), and total number of chains (11, which is 'b' in hexadecimal). Then we have this yellow region, containing a four-byte word for each delay pattern, encoding the delay pattern itself along with the number of chains with that pattern. Then, highlighted in blue, we have the 11 chains themselves, each occupying 16 bytes.

We can see that out of the first 11 bytes, only the first 8 are nonzero,

because these chains contain 8 gates. For the equivalence class directly above, there are 10 gates in each optimal chain; for the class below, there are 9. These bytes specify the topology of the DAG, and the last few bytes specify the types of operations for each gate.

So, this is how they're stored in the database, but how do we know where in the database to look to find the optimal chains for a particular function?

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;
```

These hash functions are really cheap to evaluate, and the identical parts of h_1 and h_2 are amenable to common subexpression elimination. But what's so special about these functions?

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.

These functions were found by writing down the functional form and trying all possible values of the shift constants to see which ones have the desired k-perfectness properties.

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;
}
```

Each 64-byte cache line in the hashtable is treated as 16 words, each four bytes. Up to the first 15 of these contain a value of h_2, corresponding cost of the Boolean function, and a 'local offset' which gets added to the 'global offset' in the sixteenth word to yield the position of the optimal chains within the database file. This function outputs both that position (in the upper 28 bits) and the cost (in the lower 4 bits) as a single 32-bit integer.

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.