Detecting Relevant Arguments in Boolean Functions

by Sven Nilsen, 2018

In an asymmetric path the `unit` function is used to erase input arguments. If the path exists, then the argument is irrelevant. Here is a function of two variables where the first argument is irrelevant:

f[unit × id → id]

When reverse engineering a boolean function of many variables, it is desirable to eliminate all irrelevant arguments. This must be done without the ability to introspect the function, but only manipulate the input values and computing the output. The problem is to find an efficient algorithm for doing so.

One idea is to select a random input for all values, and then turn on/off the arguments one by one. Some changes will affect the output, while other changes will give the same output. If the output is affected by the change, then the function depends on that particular argument.

There is a probability P associated with selecting a random boolean function of N arguments, selecting a random set of input bits and then toggling one input value to see whether the output value changes. This probability does not depend on which input that is selected, because this is a symmetry in the space of functions.

P0	The argument is irrelevant when all other inputs are fixed
P1	The argument is relevant when all other inputs are fixed

One can use a binary encoding of the output to count the functions of type `bool → bool`:

00	P0
01	P1
10	P1
11	P0

In the function space `bool → bool` the probability of P0 is 0.5, or 50%. Since there is just one argument, this is the same probability as the argument being irrelevant for all inputs.

The same is true for any boolean function space: 50% probability of an argument being irrelevant when all other inputs are fixed (P0). Assuming the last input bit is toggled, one has made a choice of the other inputs such that there is only a judgement of the pattern `00` which is P0 when the bits are equivalent.

It is easy to see why this is true by pairing up P0s with P1s after taking the average of each choice:

0000	P0P0	P0
0001	P0P1	P0.5
0010	P0P1	P0.5
0011	P0P0	P0
0100	P1P0	P0.5
0101	P1P1	P1
0110	P1P1	P1
0111	P1P0	P0.5
1000	P1P0	P0.5
1001	P1P1	P1
1010	P1P1	P1
1011	P1P0	P0.5
1100	P0P0	P0
1101	P0P1	P0.5
1110	P0P1	P0.5
1111	P0P0	P0

An argument is relevant when there exists some configuration of the other inputs where toggling the value changes the output value:

$$\exists\ oi : other\_input\ \{\ f(oi, false) \neg= f(oi, true)\ \}$$

An argument is irrelevant when toggling it for all configurations of the other inputs does not change the output:

$$\forall\ oi : other\_input\ \{\ f(oi, false) = f(oi, true)\ \}$$

Finding out whether an argument is relevant or irrelevant is equally hard. Assume there was an easy way to detect whether an argument is relevant, but only hard ways to figure out whether an argument is irrelevant. Since an argument is irrelevant when it is not relevant, one could use the same easy algorithm to detect whether an argument is irrelevant by flipping the answer. Therefore, there can not be only hard ways to figure out whether an argument is irrelevant, which proves by contradiction that it is equally hard to detect both properties:

$$irrelevant = not \cdot relevant$$

If an argument is relevant for some configuration of the other inputs, then there is no need to look further. This means that an algorithm that detects whether a single argument is relevant can terminate. The probability that an algorithm terminates after N tries, selecting random inputs and toggles the argument, is:

$$1 - 0.5^N$$

So, this is the confidence that the argument is irrelevant after N tries.

However, one can do better than just selecting random inputs when detecting all arguments and when the number of relevant arguments are few compared to the total number of arguments. By checking all values of known relevant arguments and selecting random values for the remaining arguments, one can assign a confidence to the belief that all the remaining arguments are irrelevant. This is done by taking the product of the individual confidence for each remaining argument:

$$C = \prod i \{ c_i \}$$

Where `c` is the confidence that the argument is irrelevant for argument `i`.

Imagine an algorithm that selects random values for all inputs and then toggles each input, not found to be relevant so far, to see whether the output changes. It keeps track of all relevant arguments. Occasionally, the algorithm checks all values of the known relevant arguments. Since the algorithm performs the same toggle check for all inputs, the confidence that an argument is irrelevant is the same, so the product can be simplified. It depends on the number of tries the algorithm has done overall:

$$C = (1 - 0.5^N)^A$$

Where N is number of steps and A is the number of remaining arguments. From this one can derive a formula for the expected number of steps to reach a level of confidence:

$$C^{1/A} = 1 - 0.5^N$$
$$0.5^N = 1 - C^{1/A}$$
$$N = \ln(1 - C^{1/A})/\ln(0.5)$$
$$N = -\ln(1 - C^{1/A})/\ln(2)$$
$$N = -\log_2(1 - C^{1/A})$$

The number of steps required to reach a level of confidence is always lower for fewer remaining arguments. Therefore, if some arguments were found to be relevant during the search to reach the level of confidence, then it only increases the confidence that the remaining arguments are irrelevant. This algorithm can be used to estimate the number of steps required to each a level of confidence simply by counting the number of input bits in the boolean function.

Usually, one is only interested in confidence levels close to 100%. Assume a boolean function has A number of arguments. To reach 0.99 confidence from 0.9, it requires a certain number of extra steps. Likewise, to reach 0.999 from 0.99, it requires another extra steps. It turns out that the number of extra steps taken to add another `9` digit largely ignores the number of arguments A. It converges to:

$$\ln(10)/\ln(2) = 3.3219280948873626$$

So, after just 10 extra steps, 3 new digits is added to the confidence. As a thumb rule, to become 1000 times more confident one needs 10 extra steps. The cool thing about this is that it is not necessary to run the check for all known relevant arguments for each step. One can run the steps until the estimated required level of confidence, then run the check once and add additional relevant arguments, repeating the check. The confidence that the remaining arguments are irrelevant is the same whether the checks are run during the search or at the end.

This algorithm can also be used to find examples of a sub-type.

For example, assume a type `A` has a binary encoding. A function `g` takes type `A` and returns `true` only when a few bits has a particular configuration.

$$x : [g]\ \text{true}$$

$$g : A \to \text{bool}$$

The algorithm that detects relevant bits in `A` does so quickly, reaching a high level of confidence. However, in order for a bit in the encoding to be relevant, it must toggle the value between `true` and `false` during the search. At this moment the algorithm has found a solution to `x`!

This tells us something about problems that are hard to solve. In order for a problem to be hard to solve, the function `g` needs to depend on lots of bits. Still, even it depends on lots of bits, it must return `false` for most of these configurations. So, it seems like most of the relevant arguments behave like irrelevant arguments most of the time! The `g` function returns `false` for most inputs.

The problem is kind of like figuring out what came first of the chicken and the egg. In order to find a solution easier, one must filter out irrelevant arguments, but this is not possible without producing a solution. Therefore, a hard problem does not become easier over time. We already know that the confidence level that all arguments are irrelevant will only increase toward 100% without finding a solution. The algorithm is doing no better than just stumbling upon a solution by pure chance.

For example, a logical proof must return `true` for all input arguments. If it was easy to find some input for which the logical proof returned `false`, then it would also be easy to check a proof. Now, one can imagine that if one could remove all irrelevant arguments, it would make the check much easier. The problem is that in a black-box proof one does not know which arguments are relevant! On the other hand, in a proof where the expressions can be analyzed, it is possible to know which arguments that are relevant for a particular part of the proof, and thus a solver can be designed to take advantage of this knowledge.

Detecting relevant arguments is only efficient when there is an easy function to be learned. So, the confidence that the remaining arguments are irrelevant can also be interpreted as “irrelevant or hard to detect the difference”.