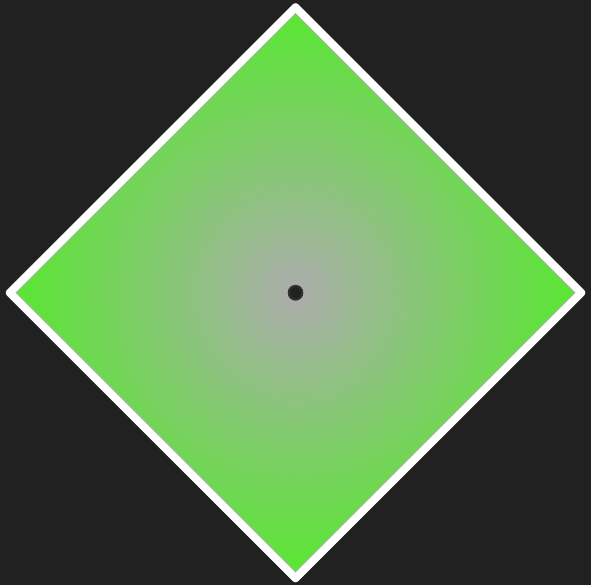
Naïve search

We always search the entire function space (works for 6 variables, but not for 7).



Naïve search

Using the observations we can avoid some functions

How: By filtering nodes

For example, for an inconsistent function of v4, using as variables v1, v2 and v3:

Nodes:

v1   v2   v3

v1, v2   v1, v3   v2, v3

v1, v2, v3

test each node with the observations

Filtered Nodes:

v1, v2   v1, v3   v2, v3

v1, v2, v3

(if a node produces 1 when 0 was expected, whatever function that node would be a part of would always produce that wrong result (DNF))

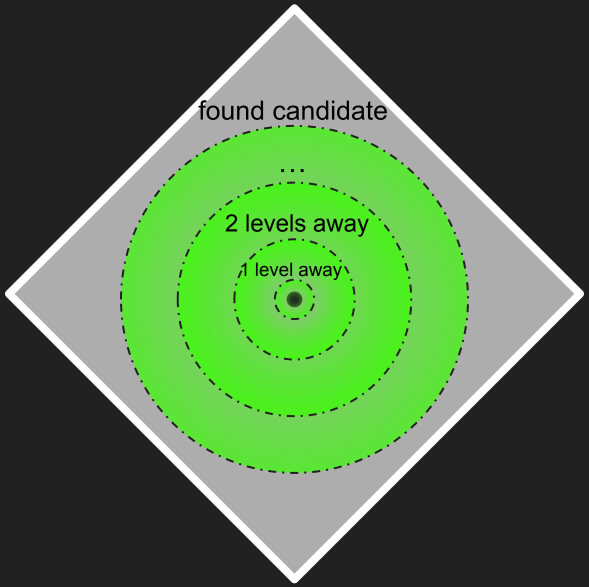
Naïve search

This reduces the search space somewhat, but not enough to get us from 6 to 7 variables (in a reasonable amount of time, anyway)

While in some cases only 1 or 2 nodes are filtered, in other cases we may filter over 20 nodes

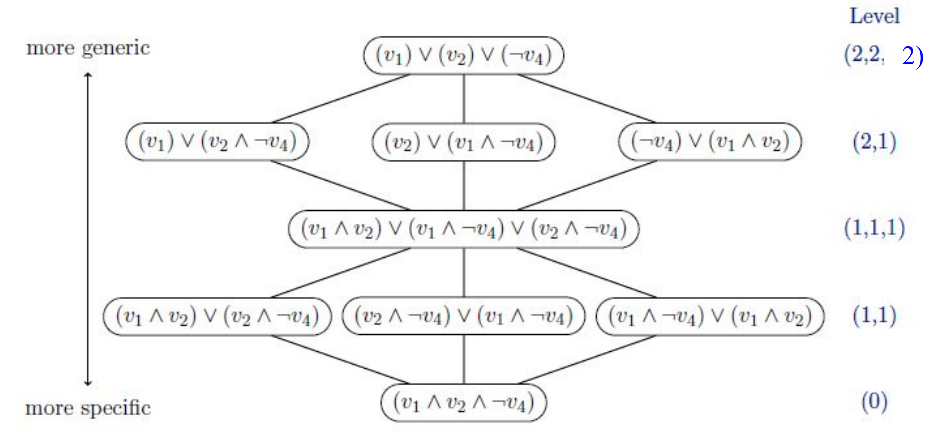
Level-based search

We begin our search by looking at all the functions that have the same level as the original. If we cannot find a suitable candidate, we search for functions that are 1 level away, then 2 levels away, etc, until we find a candidate.



Level-based search

- The level of a function is a tuple, with as many elements as terms in the function.
- Each element is an integer which tells us how many variables are missing from each term (elements are ordered in decreasing order)



Level
more generic
$(v_1) \vee (v_2) \vee (\neg v_4)$
(2,2, 2)
$(v_1) \vee (v_2 \wedge \neg v_4)$
$(v_2) \vee (v_1 \wedge \neg v_4)$
$(\neg v_4) \vee (v_1 \wedge v_2)$
(2,1)
$(v_1 \wedge v_2) \vee (v_1 \wedge \neg v_4) \vee (v_2 \wedge \neg v_4)$
(1,1,1)
$(v_1 \wedge v_2) \vee (v_2 \wedge \neg v_4)$
$(v_2 \wedge \neg v_4) \vee (v_1 \wedge \neg v_4)$
$(v_1 \wedge \neg v_4) \vee (v_1 \wedge v_2)$
(1,1)
more specific
$(v_1 \wedge v_2 \wedge \neg v_4)$
(0)

Figure 2.8: Hasse diagram for monotone non-degenerate Boolean functions of three arguments (regulators v_1, v_2, and $\neg v_4$).

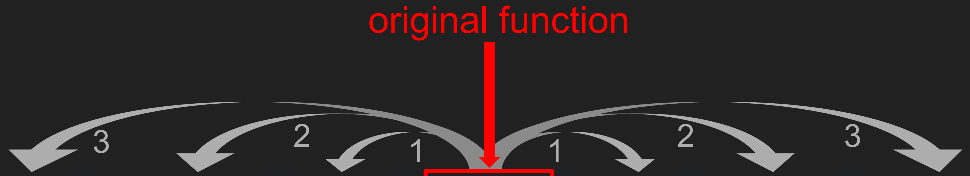
Level-based search

- We can establish a total order relation between the levels of any two functions in the BCF of the same dimension.
  - For example, for 4 variables all possible levels would be:

(0) ≤ (1,1) ≤ (1,1,1) ≤ (1,1,1,1) ≤ (2,1) ≤ (2,1,1) ≤ (2,2) ≤ (2,2,1) ≤ (2,2,2) ≤ (2,2,2,2)
≤ (2,2,2,2,2) ≤ (2,2,2,2,2,2) ≤ (3,1) ≤ (3,2,2) ≤ (3,2,2,2) ≤ (3,3,2) ≤ (3,3,3,3)

Level-based search

- Strategy: We start looking at candidates that have the same level of the inconsistent function, and then get progressively farther away until we find one or more candidates in level L which satisfy the observations
  - this strategy is an iterative process

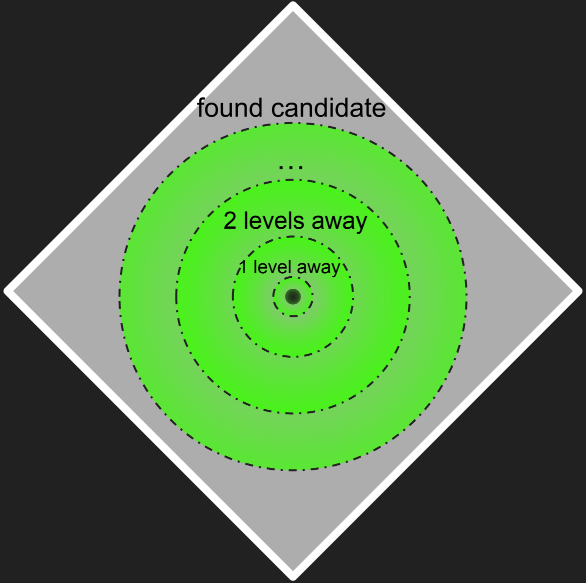


$(0) \le (1,1) \le (1,1,1) \le (1,1,1,1) \le (2,1) \le \boxed{(2,1,1)} \le (2,2) \le (2,2,1) \le (2,2,2) \le (2,2,2,2)$
$\le (2,2,2,2,2) \le (2,2,2,2,2,2) \le (3,1) \le (3,2,2) \le (3,2,2,2) \le (3,3,2) \le (3,3,3,3)$

Level-based search

Pros

- smaller search space than naïve search
- is guaranteed to find an optimal function (in regards to the number of terms in the function and number of variables in each term)



Level-based search

Cons

- search might still take some time (in case the inconsistent function’s level is far from the level of the first suitable candidate – not very likely in real-world scenarios)
- it is not trivial to discover what levels have and do not have functions
- it is not trivial to find previous levels efficiently

Level-based search

Cons

- it is not trivial to discover what levels have and do not have functions

For 4 variables, say our original function has level (2,2) :

- We know level (2,2) has at least 1 function
- But does the next level (2,2,1) have functions? What about the previous level (2,1,1,1,1,1)?
  - (we know (2,1,1,1,1,1) is the previous level because the maximum number of terms a function with at most 2 missing variables can have can be obtained by ${}_4C_2 = 6$)

(0) ≤ (1,1) ≤ (1,1,1) ≤ (1,1,1,1) ≤ (2,1) ≤ (2,1,1) ≤ (2,2) ≤ (2,2,1) ≤ (2,2,2) ≤ (2,2,2,2)
≤ (2,2,2,2,2) ≤ (2,2,2,2,2,2) ≤ (3,1) ≤ (3,2,2) ≤ (3,2,2,2) ≤ (3,3,2) ≤ (3,3,3,3)

Level-based search

Cons

- current solution: ask clingo to produce a single function. If it is able to, the level exists. Otherwise, the level doesn't exist (even though this is very fast, when we are doing many iterations it may turn out to be inefficient…)

- possible alternative: come up with some mathematical formula that is able to tell us, given the total number of variables and a level (and maybe some more information…), if any possible function exists

Level-based search

Cons

- it is not trivial to find previous levels efficiently

Next levels aren't a big issue. If our original function has level (2,1):

- Does level (2,1,1) exist? Yes. Does (2,1,1,1) exist? No. So we know for sure (2,1,1,1,1) won't exist either, and can stop adding terms. At this point we can instead increase the number of missing variables of the left-most lowest term, thus going straight from (2,1,1,1) to (2,2)

$$(0) \leq (1,1) \leq (1,1,1) \leq (1,1,1,1) \leq (2,1) \leq (2,1,1) \leq (2,2) \leq (2,2,1) \leq (2,2,2) \leq (2,2,2,2)$$
$$\leq (2,2,2,2,2) \leq (2,2,2,2,2,2) \leq (3,1) \leq (3,2,2) \leq (3,2,2,2) \leq (3,3,2) \leq (3,3,3,3)$$

Level-based search

Cons

- it is not trivial to find previous levels efficiently

Previous levels don’t have this ‘cheat’. If our original function has level (2,2):

- Does level (2,1,1,1,1,1) exist? No. But that doesn’t tell us anything about the previous levels. (2,1,1,1,1) could exist, and even if it doesn’t, (2,1,1,1) could exist. We have no obvious choice but to check every single level here.

(0) ≤ (1,1) ≤ (1,1,1) ≤ (1,1,1,1) ≤ (2,1) ≤ (2,1,1) ≤ (2,2) ≤ (2,2,1) ≤ (2,2,2) ≤ (2,2,2,2)
≤ (2,2,2,2,2) ≤ (2,2,2,2,2,2) ≤ (3,1) ≤ (3,2,2) ≤ (3,2,2,2) ≤ (3,3,2) ≤ (3,3,3,3)

Level-based search

Cons

- possible solution: implement some sort of binary search, that will check first if some lower level than the previous exist. If it does not, we should be able to tell that between that level and the previous, no functions exist.
    - e.g. if (2,1,1,1,1,1) doesn’t exist, check if (2,1) exists. If it doesn’t, we know nothing in between exists.

✅

Level-based search

Performance (averaged over 3 measurements)

- 5 variables
  - Total levels explored: 34
  - Total time taken: 0.169s
  - Time taken by clingo: 0.146s

  - Original level: (4,4,3,2)
  - Found level: (4,3,3,3)

- 6 variables
  - Total levels explored: 1403
  - Total time taken: 7.130s
  - Time taken by clingo: 5.691s

  - Original level: (4,4,4,4,4)
  - Found level: (4,3,2)

- 7 variables
  - Total levels explored: 7
  - Total time taken: 0.273s
  - Time taken by clingo: 0.249s

  - Original level: (0)
  - Found level: (2,1)

- 8 variables
  - Total levels explored: 22
  - Total time taken: 2.333s
  - Time taken by clingo: 2.275s

  - Original level: (0)
  - Found level: (2,2,2)

Level-based search

Problems regarding optimality of found solutions

- For example:

$$\begin{tikzcd}[column sep=small]
\ldots \leq (2,2,2,2,2) \leq & \overset{\text{found candidate}}{\boxed{(2,2,2,2,2,2)}} & \leq & \overset{\text{original function}}{\boxed{(3,1)}} \arrow[ll, bend left, "1"] \arrow[rr, bend right, "1"'] & \leq & \overset{\text{found candidate}}{\boxed{(3,2,2)}} & \leq (3,2,2,2) \leq (3,3,2) \leq (3,3,3,3)
\end{tikzcd}$$

- Even though they are both 1 distance away, are they both “optimal”?
- (2,2,2,2,2,2) seems farther from (3,1) than (3,2,2), no matter how we look at it, even though the distance is the same
- If we want an optimal solution, maybe we should define some equation that takes into account the difference in number of terms, as well as variables within each term when compared to the original function?
  - e.g. optimality = |no. original terms - candidate terms| + sum(|difference between original terms’ variables and candidate terms’ variables|)
    - 2,2,2,2,2,2: |2 - 6| + |3-2| + |1-2| = 6
    - 3,2,2: |2-3| + |3-3| + |1-2| = 2 (closer to 0 is better)

Level-based search

Strategy (to ensure optimality):

- We start looking at candidates that have the same level of the inconsistent function, and then get progressively farther away until we find one or more candidates in level L+N or L-M which satisfy the observations
- Then, we calculate how optimal that candidate is. We continue our search in the other direction if we haven't surpassed the value of optimality of the candidates.
- We stop the search if we find a candidate with a lower optimality value (keep the new candidate), or if we reach the same optimality level (keep the old candidate)

$$\begin{tikzcd}
 & & \text{first candidate found} & \text{original function} & & & \text{second candidate found} & \\
\dots \arrow[r, phantom, "\leq"] & (2,2,2,2,2) \arrow[r, phantom, "\leq"] & \boxed{(2,2,2,2,2,2)} \arrow[r, phantom, "\leq"] & \boxed{(3,1)} \arrow[l, bend left, "\text{optimality: 6}"] \arrow[r, bend right, "\text{optimality: 2}"'] \arrow[r, phantom, "\leq"] & (3,2,2) \arrow[r, bend right, "\text{optimality: 3}"'] \arrow[r, phantom, "\leq"] & (3,2,2,2) \arrow[r, bend right, "\text{optimality: 3}"'] \arrow[r, phantom, "\leq"] & \boxed{(3,3,2)} \arrow[r, phantom, "\leq"] & (3,3,3,3)
\end{tikzcd}$$