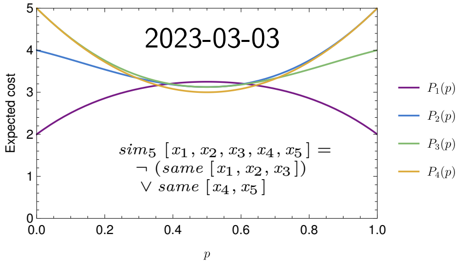
Level-p-complexity of Boolean Functions

Using thinning, memoization, and polynomials

Patrik Jansson[1]

Functional Programming unit, Chalmers University of Technology



[1] Joint work with Julia Jansson, PhD student, Math@Chalmers. Pre-print on arXiv, source code on GitHub.

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For the example 5-bit function sim_5 there are already **54192** different decision trees.

$$t_5 = size\ (genAlg\ 5\ sim_5 :: Set\ DecTree)$$

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For the example 5-bit function sim_5 there are already **54192** different decision trees.

$$t_5 = size\ (genAlg\ 5\ sim_5 :: Set\ DecTree)$$

For a 9-bit function (we aim for maj_3^2) we can estimate the number of trees as

$$t_9 = 9 * t_8^2; \quad t_8 = 8 * t_7^2; \quad t_7 = 7 * t_6^2; \quad t_6 = 6 * t_5^2$$

This estimate gives us 10^{89} decision trees!
Clearly, a naive implementation will take far too long to terminate.

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For the example 5-bit function sim_5 there are already **54192** different decision trees.

$$t_5 = size\ (genAlg\ 5\ sim_5 :: Set\ DecTree)$$

For a 9-bit function (we aim for maj_3^2) we can estimate the number of trees as

$$t_9 = 9 * t_8^2; \quad t_8 = 8 * t_7^2; \quad t_7 = 7 * t_6^2; \quad t_6 = 6 * t_5^2$$

This estimate gives us 10^{89} decision trees!
Clearly, a naive implementation will take far too long to terminate.

By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

$$\begin{aligned} ps_9 &= genAlgThinMemo\ 9\ maj_3^2 :: Set\ (Poly\ \mathbb{Q}) \\ check_9 &= ps_9 == fromList\ [P\ [4, 4, 6, 9, -61, 23, 67, -64, 16]] \end{aligned}$$

Motivating example: two-level iterated majority

Our running example is a simple case of two-level majority ($maj_3^2 : \mathbb{B}^9 \to \mathbb{B}$).

$$\underbrace{\underbrace{x_{(1,1)}, x_{(1,2)}, x_{(1,3)}}_{m_1 = maj_3\ (\ldots)}, \underbrace{x_{(2,1)}, x_{(2,2)}, x_{(2,3)}}_{m_2 = maj_3\ (\ldots)}, \underbrace{x_{(3,1)}, x_{(3,2)}, x_{(3,3)}}_{m_3 = maj_3\ (\ldots)}}_{maj_3(m_1, m_2, m_3)}$$

Motivating example: two-level iterated majority

Our running example is a simple case of two-level majority ($maj_3^2 : \mathbb{B}^9 \to \mathbb{B}$).

$$\underbrace{\underbrace{x_{(1,1)}, x_{(1,2)}, x_{(1,3)}}_{m_1 = maj_3\ (\ldots)}, \underbrace{x_{(2,1)}, x_{(2,2)}, x_{(2,3)}}_{m_2 = maj_3\ (\ldots)}, \underbrace{x_{(3,1)}, x_{(3,2)}, x_{(3,3)}}_{m_3 = maj_3\ (\ldots)}}_{maj_3(m_1, m_2, m_3)}$$

Example 1: Five $\mathbf{0}$ votes, four $\mathbf{1}$ votes, even distribution:

$$\underbrace{\underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_1 = \mathbf{0}}, \underbrace{\mathbf{1}, \mathbf{0}, \mathbf{1}}_{m_2 = \mathbf{1}}, \underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_3 = \mathbf{0}}}_{maj_3 = \mathbf{0}}$$

Motivating example: two-level iterated majority

Our running example is a simple case of two-level majority ($maj_3^2 : \mathbb{B}^9 \to \mathbb{B}$).

$$\underbrace{\underbrace{x_{(1,1)}, x_{(1,2)}, x_{(1,3)}}_{m_1 = maj_3\ (\ldots)}, \underbrace{x_{(2,1)}, x_{(2,2)}, x_{(2,3)}}_{m_2 = maj_3\ (\ldots)}, \underbrace{x_{(3,1)}, x_{(3,2)}, x_{(3,3)}}_{m_3 = maj_3\ (\ldots)}}_{maj_3(m_1, m_2, m_3)}$$

Example 1: Five $\mathbf{0}$ votes, four $\mathbf{1}$ votes, even distribution:

$$\underbrace{\underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_1 = \mathbf{0}}, \underbrace{\mathbf{1}, \mathbf{0}, \mathbf{1}}_{m_2 = \mathbf{1}}, \underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_3 = \mathbf{0}}}_{maj_3 = \mathbf{0}}$$

Example 2: Five $\mathbf{0}$ votes, four $\mathbf{1}$ votes, regrouped (perhaps through gerrymandering):

$$\underbrace{\underbrace{\mathbf{1}, \mathbf{1}, \mathbf{0}}_{m_1 = \mathbf{1}}, \underbrace{\mathbf{1}, \mathbf{0}, \mathbf{1}}_{m_2 = \mathbf{1}}, \underbrace{\mathbf{0}, \mathbf{0}, \mathbf{0}}_{m_3 = \mathbf{0}}}_{maj_3 = \mathbf{1}}$$

Some key types and definitions

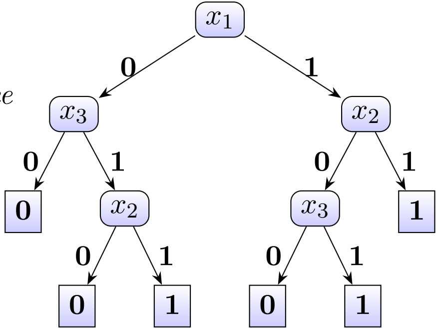
- A Boolean function $f : \mathit{BoolFun}\ n$ takes a tuple of n bits to one bit.
- A tuple $t : \mathbb{B}^n$ is just a vector of n bits (**type** $\mathbb{B} = \mathit{Bool}$; $\mathbf{0}$ = False; $\mathbf{1}$ = True).
- A decision tree describes a way to evaluate a $\mathit{BoolFun}$:

Haskell version:

$$
\begin{array}{l}
\textbf{type } \mathit{Index} = \mathbb{N} \\
\textbf{data } \mathit{DecTree} \textbf{ where} \\
\quad \mathit{Res}\ \ :: \mathbb{B} \to \mathit{DecTree} \\
\quad \mathit{Pick} :: \mathit{Index} \to \mathit{DecTree} \to \mathit{DecTree} \to \mathit{DecTree}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ex}_1 :: \mathit{DecTree} \\
\mathit{ex}_1 = \mathit{Pick}\ 1\ (\mathit{Pick}\ 3\ (\mathit{Res}\ \mathbf{0}) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{Pick}\ 2\ (\mathit{Res}\ \mathbf{0})\ (\mathit{Res}\ \mathbf{1}))) \\
\qquad\qquad\qquad\quad (\mathit{Pick}\ 2\ (\mathit{Pick}\ 3\ (\mathit{Res}\ \mathbf{0})\ (\mathit{Res}\ \mathbf{1})) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{Res}\ \mathbf{1}))
\end{array}
$$

A decision tree (ex_1) for the 3-bit majority function:



Some key types and definitions

A decision tree describes a way to evaluate a $BoolFun$.

Haskell version:

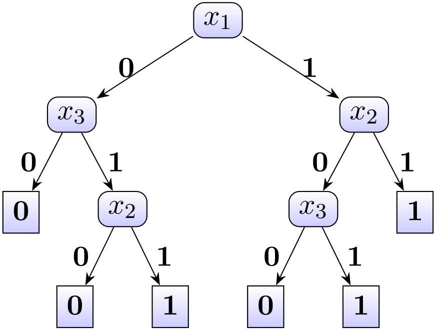
$$
\begin{array}{l}
\textbf{type}\ Index = \mathbb{N} \\
\textbf{data}\ DecTree\ \textbf{where} \\
\quad Res\ \ :: \mathbb{B} \to DecTree \\
\quad Pick :: Index \to DecTree \to DecTree \to DecTree
\end{array}
$$

Agda version (captures more invariants):

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ DecTree : (n : \mathbb{N}) \to (f : BoolFun\ n) \to Set\ \textbf{where}} \\
\quad Res\ : & & (b : \mathbb{B}) \to & DecTree\ n\ (const_n\ b) \\
\quad Pick : & \{f : BoolFun\ (suc\ n)\} \to & (i : Fin\ (suc\ n)) \to & DecTree\ n\ (setBit\ i\ \mathbf{0}\ f) \to \\
& & & DecTree\ n\ (setBit\ i\ \mathbf{1}\ f) \to \\
& & & DecTree\ (suc\ n)\ f
\end{array}
$$

$$setBit : Fin\ (suc\ n) \to \mathbb{B} \to BoolFun\ (suc\ n) \to BoolFun\ n$$

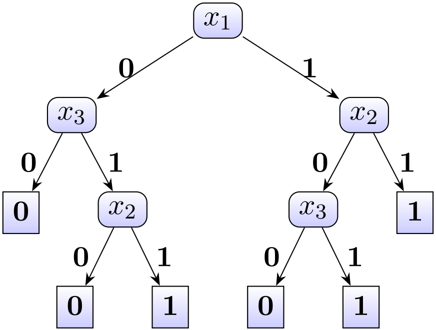
Decision trees, cost and complexity



The decision tree ex_1 for the 3-bit majority function maj:

- Note that the left subtree is for the function $f_0 = setBit\ 1\ \mathbf{0}\ maj = (\wedge)$
- and the right subtree for the function $f_1 = setBit\ 1\ \mathbf{1}\ maj = (\vee)$

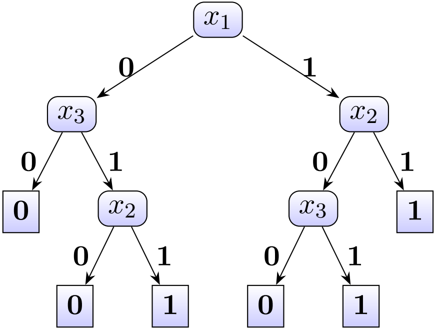
Decision trees, cost and complexity

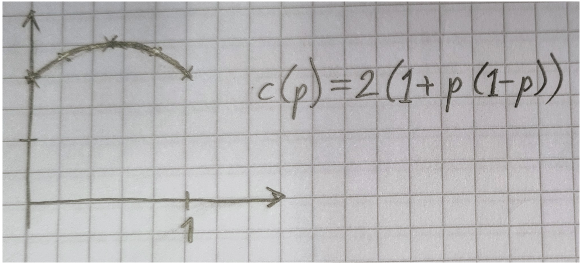


The decision tree ex_1 for the 3-bit majority function maj:

- Note that the left subtree is for the function $f_0 = setBit\ 1\ \mathbf{0}\ maj = (\wedge)$
- and the right subtree for the function $f_1 = setBit\ 1\ \mathbf{1}\ maj = (\vee)$

- The *cost* of a *DecTree* is a function from $\mathbb{B}^n$ to $\mathbb{N}$:
  $cost : \{n : \mathbb{N}\} \to \{f : BoolFun\ n\} \to DecTree\ n\ f \to \mathbb{B}^n \to \mathbb{N}$

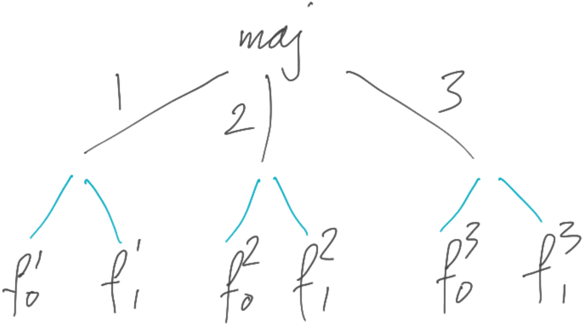




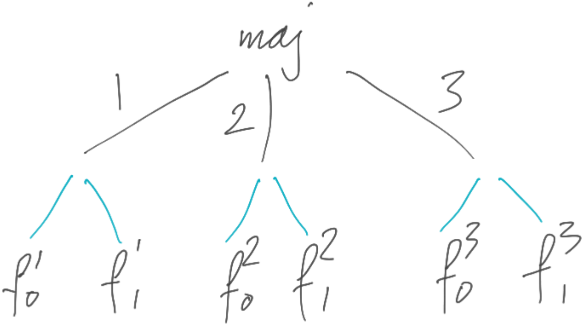
$expCost\ ex_1 == P\ [2, 2, -2]$

- Finally, the *expected cost*[2], is a polynomial in the probability p that a bit is **1**:
  $expCost : \{n : \mathbb{N}\} \rightarrow \{f : BoolFun\ n\} \rightarrow DecTree\ n\ f \rightarrow Poly_n\ a$

[2]We assume n independent bits, all with probability p to be $\mathbf{1}$. The polynomials are defined over any $Ring\ a$.



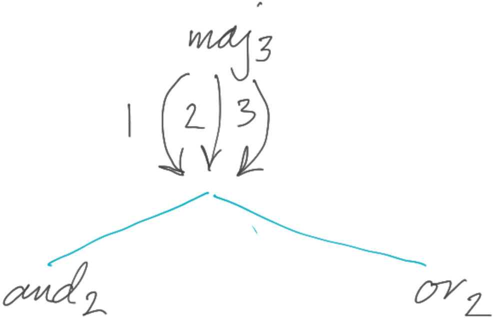
$$f_b^i = \text{setBit } i \; b \; \text{maj}$$



$$f_b^i = \text{setBit } i \; b \; \text{maj}$$

$$f_0^i = \wedge$$

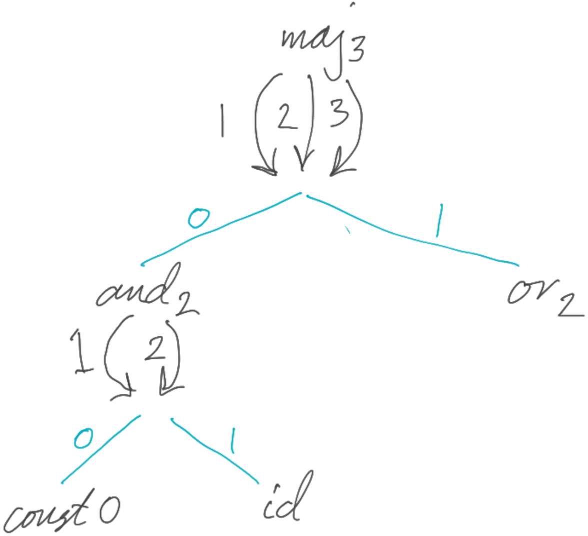
$$f_1^i = \vee$$



$$f_b^i = setBit\ i\ b\ maj$$

$$f_0^i = \wedge = and_2$$

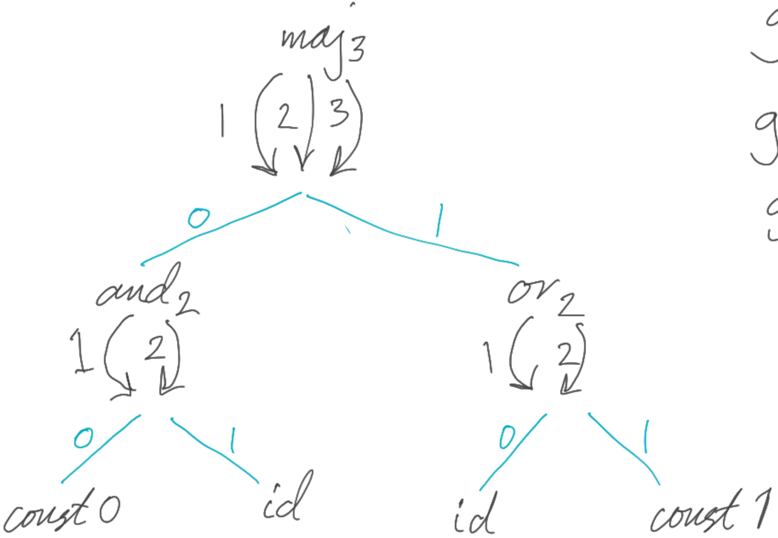
$$f_1^i = \vee = or_2$$



$$g^i_b = \mathit{setBit}\ i\ b\ \mathit{and}_2$$

$$g^i_0 = \mathit{const}\ 0$$

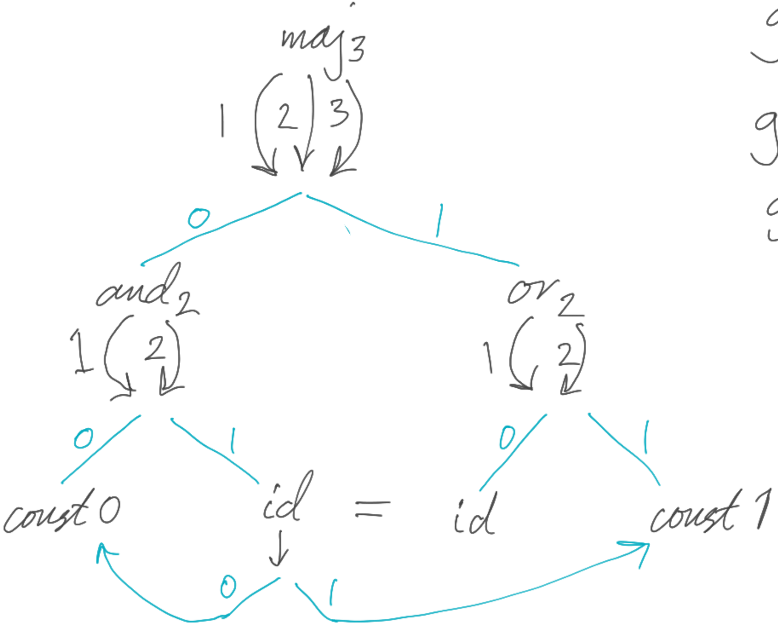
$$g^i_1 = \mathit{id}_{\mathbb{B}}$$



$$g^i_b = setBit\ i\ b\ or_2$$

$$g^i_0 = id_{\mathbb{B}}$$

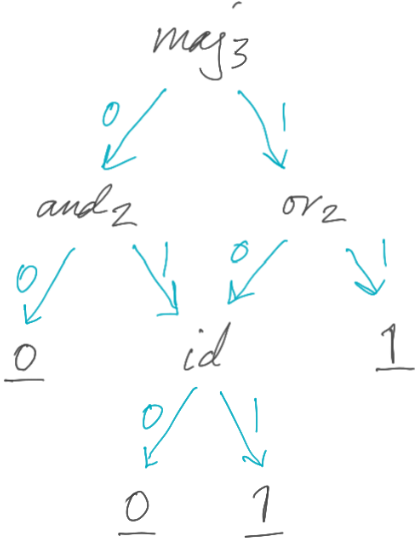
$$g^i_1 = const\ 1$$



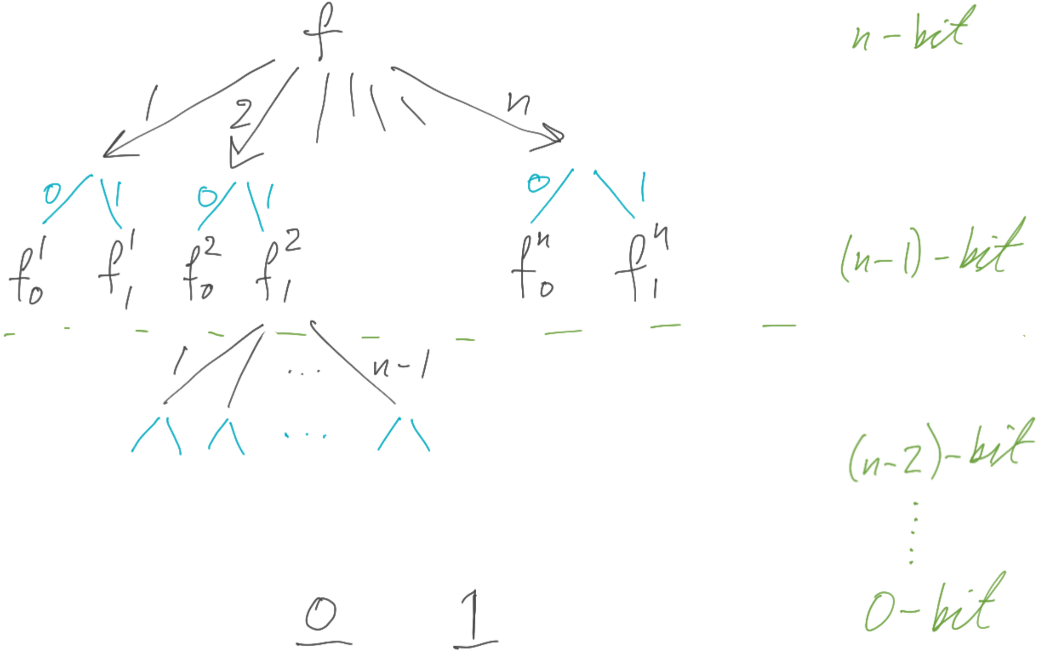
$$g_b^i = \mathit{setBit}\ i\ b\ \mathit{or}_2$$

$$g_0^i = \mathit{id}_{\mathbb{B}}$$

$$g_1^i = \mathit{const}\ 1$$



maj_3
0
1
and_2
or_2
0
1
0
1
0
id
1
0
1
0
1



f
1
2
n
0
1
f_0^1
f_1^1
f_0^2
f_1^2
f_0^n
f_1^n
n-1
0
1
n-bit
(n-1)-bit
(n-2)-bit
0-bit

A class of Boolean functions

We abstract from the actual type for Boolean functions to a type class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{BoFun}\ \mathit{bf}\ \textbf{where} \\
\quad \mathit{isConst} :: \mathit{bf} \rightarrow \mathit{Maybe}\ \mathbb{B} \\
\quad \mathit{setBit} \quad :: \mathit{Index} \rightarrow \mathbb{B} \rightarrow \mathit{bf} \rightarrow \mathit{bf}
\end{array}
$$

We only use one instance: Binary Decision Diagrams (BDDs) which allows good sharing and fast operations.

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$\begin{array}{l} \textbf{class}\ \mathit{TreeAlg}\ a\ \textbf{where} \\ \quad \mathit{res} :: \mathbb{B} \to a \\ \quad \mathit{pic} :: \mathit{Index} \to a \to a \to a \end{array}$$

$$\begin{array}{ll} \mathit{foldDT} :: \mathit{TreeAlg}\ a \Rightarrow \mathit{DecTree} \to a & \\ \mathit{foldDT}\ (\mathit{Res}\ b) & = \mathit{res}\ b \\ \mathit{foldDT}\ (\mathit{Pick}\ i\ t_0\ t_1) & = \mathit{pic}\ i\ (\mathit{foldDT}\ t_0)\ (\mathit{foldDT}\ t_1 \end{array}$$

$$\begin{array}{l} \mathit{ex}_1 :: \mathit{TreeAlg}\ a \Rightarrow a \\ \mathit{ex}_1 = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\ \qquad\qquad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1})) \end{array}$$

$$\begin{array}{llll} \textbf{instance} & \mathit{TreeAlg}\ \mathit{DecTree} & \textbf{where}\ \mathit{res} = \mathit{Res}; & \mathit{pic} = \mathit{Pick}; \\ \textbf{instance} & \mathit{TreeAlg}\ \mathit{CostFun} & \textbf{where}\ \mathit{res} = \mathit{resC}; & \mathit{pic} = \mathit{pickC} \\ \textbf{instance}\ \mathit{Ring}\ a \Rightarrow & \mathit{TreeAlg}\ (\mathit{ExpCost}\ a) & \textbf{where}\ \mathit{res} = \mathit{resPoly}; & \mathit{pic} = \mathit{pickPoly} \end{array}$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$\begin{aligned}
&\textbf{class } \mathit{TreeAlg}\ a\ \textbf{where} \\
&\quad \mathit{res} :: \mathbb{B} \to a \\
&\quad \mathit{pic} :: \mathit{Index} \to a \to a \to a
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldDT} :: \mathit{TreeAlg}\ a \Rightarrow \mathit{DecTree} \to a \\
&\mathit{foldDT}\ (\mathit{Res}\ b) = \mathit{res}\ b \\
&\mathit{foldDT}\ (\mathit{Pick}\ i\ t_0\ t_1) = \mathit{pic}\ i\ (\mathit{foldDT}\ t_0)\ (\mathit{foldDT}\ t_1
\end{aligned}$$

$$\begin{aligned}
&\mathit{ex}_1 :: \mathit{TreeAlg}\ a \Rightarrow a \\
&\mathit{ex}_1 = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\
&\qquad\qquad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance } \mathit{TreeAlg}\ \mathit{DecTree}\ \textbf{where } \mathit{res} = \mathit{Res}; \mathit{pic} = \mathit{Pick}; \\
&\textbf{data } \mathit{DecTree}\ \textbf{where} \\
&\quad \mathit{Res} :: \mathbb{B} \to \mathit{DecTree} \\
&\quad \mathit{Pick} :: \mathit{Index} \to \mathit{DecTree} \to \mathit{DecTree} \to \mathit{DecTree}
\end{aligned}$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$\begin{aligned}&\mathbf{class}\ \mathit{TreeAlg}\ a\ \mathbf{where}\\&\quad \mathit{res} :: \mathbb{B} \to a\\&\quad \mathit{pic} :: \mathit{Index} \to a \to a \to a\end{aligned}$$

$$\begin{aligned}&\mathit{foldDT} :: \mathit{TreeAlg}\ a \Rightarrow \mathit{DecTree} \to a\\&\mathit{foldDT}\ (\mathit{Res}\ b) = \mathit{res}\ b\\&\mathit{foldDT}\ (\mathit{Pick}\ i\ t_0\ t_1) = \mathit{pic}\ i\ (\mathit{foldDT}\ t_0)\ (\mathit{foldDT}\ t_1\end{aligned}$$

$$\begin{aligned}&\mathit{ex}_1 :: \mathit{TreeAlg}\ a \Rightarrow a\\&\mathit{ex}_1 = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1})))\\&\qquad\qquad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))\end{aligned}$$

$$\begin{aligned}&\mathbf{instance}\ \mathit{TreeAlg}\ \mathit{CostFun}\ \mathbf{where}\ \mathit{res} = \mathit{resC}; \mathit{pic} = \mathit{pickC}\\&\mathbf{type}\ \mathit{CostFun} = \mathbb{B}^n \to \mathit{Int}\\&\mathit{resC}\ b = \mathit{const}\ 0\\&\mathit{pickC}\ i\ c_0\ c_1 = \lambda t \to 1 + \mathbf{if}\ \mathit{index}\ t\ i\ \mathbf{then}\ c_1\ t\ \mathbf{else}\ c_0\ t\end{aligned}$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$\begin{aligned}
&\textbf{class } \mathit{TreeAlg}\ a\ \textbf{where} \\
&\quad \mathit{res} :: \mathbb{B} \rightarrow a \\
&\quad \mathit{pic} :: \mathit{Index} \rightarrow a \rightarrow a \rightarrow a
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldDT} :: \mathit{TreeAlg}\ a \Rightarrow \mathit{DecTree} \rightarrow a \\
&\mathit{foldDT}\ (\mathit{Res}\ b) \qquad\quad = \mathit{res}\ b \\
&\mathit{foldDT}\ (\mathit{Pick}\ i\ t_0\ t_1) = \mathit{pic}\ i\ (\mathit{foldDT}\ t_0)\ (\mathit{foldDT}\ t_1
\end{aligned}$$

$$\begin{aligned}
&ex_1 :: \mathit{TreeAlg}\ a \Rightarrow a \\
&ex_1 = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\
&\qquad\qquad\qquad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance } \mathit{Ring}\ a \Rightarrow \mathit{TreeAlg}\ (\mathit{ExpCost}\ a)\ \textbf{where } \mathit{res} = \mathit{resPoly}; \mathit{pic} = \mathit{pickPoly} \\
&\textbf{type } \mathit{ExpCost}\ a = \mathit{Poly}\ a \\
&\mathit{resPoly}\ _b = \mathit{zero} \\
&\mathit{pickPoly}\ _\ p_0\ p_1 = \mathit{one} + (\mathit{one} - \mathit{xP}) * p_0 + \mathit{xP} * p_1
\end{aligned}$$

Problem formulation: compute the *complexity*

- generate all DecTrees for a given function

$$genAlg :: (BoFun\ bf, TreeAlg\ ta) \Rightarrow bf \ \rightarrow\ Set\ ta$$

- find a small(est) set of dominating trees

$$\begin{aligned}
&minWith :: Preorder\ b \Rightarrow (a\ \rightarrow\ b)\ \rightarrow\ Set\ a\ \rightarrow\ Set\ a\\
&min \qquad\ \ :: Preorder\ a \Rightarrow \qquad\qquad\quad\ \ Set\ a\ \rightarrow\ Set\ a\\
&min = minWith\ id
\end{aligned}$$

- then just keep their costs

$$\begin{aligned}
&complexity :: (BoFun\ bf, Ring\ a) \Rightarrow bf\ \rightarrow\ Set\ (Poly\ a)\\
&complexity = mapS\ \ expCost \circ minWith\ expCost \circ genAlg
\end{aligned}$$

Calculation steps

- We can push the map on the other side of $minWith$

$$complexity = mapS\ expCost \circ minWith\ expCost \circ genAlg$$

-- is equal to

$$complexity = minWith\ id \circ mapS\ expCost \circ genAlg$$

- Note that this makes the computation more efficient, beacause several trees map to the same polynomial

Calculation step, cont.

- Next, we push the $expCost$ computation into the tree generation, to get another gain in efficiency.

$$complexity = min \circ mapS\ expCost \circ genAlg$$
-- is equal to
$$complexity = min \circ genPoly$$

- Still generating too many polynomials - we can push some of $minWith$ into the $genPoly$ computation. This is the thinning step.

$$complexity = min \circ thin \circ genPoly$$
-- is equal to
$$complexity = min \circ genPolyThin$$

Unfolding boolean functions

$genAlg$, $genPoly$, and $genPolyThin$ all share the same structure:

$$
\begin{aligned}
&genAlg_n :: (BoFun\ bf, TreeAlg\ ta) \Rightarrow bf \rightarrow Set\ ta \\
&genAlg_n \quad f \mid Just\ b \leftarrow isConst\ f \\
&\qquad\qquad = \{res\ b\} \\
&genAlg_{n+1}\ f \ = \{pic\ i\ t_0\ t_1 \mid i \leftarrow \{1\,..\,n\}, t_0 \leftarrow genAlg_n\ f^i_{\mathbf{0}}, t_1 \leftarrow genAlg_n\ f^i_{\mathbf{1}}\}
\end{aligned}
$$

[3] For maj_3^2 as estimate 100 million call, but just 215 distinct subfunctions.

Unfolding boolean functions

$genAlg$, $genPoly$, and $genPolyThin$ all share the same structure:

$$
\begin{aligned}
&genAlg_n :: (BoFun\ bf, TreeAlg\ ta) \Rightarrow bf \rightarrow Set\ ta \\
&genAlg_n \quad f \mid Just\ b \leftarrow isConst\ f \\
&\qquad\qquad = \{ res\ b \} \\
&genAlg_{n+1}\ f \ = \{ pic\ i\ t_0\ t_1 \mid i \leftarrow \{1 \mathinner{..} n\}, t_0 \leftarrow genAlg_n\ f^i_{\mathbf{0}}, t_1 \leftarrow genAlg_n\ f^i_{\mathbf{1}} \}
\end{aligned}
$$

Unfold structure:

- We have an "unfold" (co-algebra) structure when we consume a boolean function and produce all dectrees.
- This can cause an **many**[3] function calls, which indicates the need to do memoization (dynamic programming) when building up the "call graph".

[3]For maj_3^2 as estimate 100 million call, but just 215 distinct subfunctions.

Memoization reminder

- Naively computing $\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n$ leads to exponential growth in the number of function calls.
- But if we fill in a table indexed by n with already computed results we get a the result in linear time.

Memoization in our case

- Similarly, here we "just" need to tabulate the result of the calls to complexity so as to avoid recomputation.
- The challenge is that the input is now a boolean function which is not as nicely structured as a natural number index.
- Fortunately, thanks to Hinze and others we have generic Tries only a hackage library away.
- Finally, putting these components together we can compute the level-p-complexity of (small) boolean functions in reasonable time.

$$\begin{aligned} \mathit{genAlgThinMemo} :: (&\mathit{BoFun}\ \mathit{bf}, \mathit{Memoizable}\ \mathit{bf}, \\ &\mathit{TreeAlg}\ a, \mathit{Thinnable}\ a) \Rightarrow \\ &\mathbb{N} \to \mathit{bf} \to \mathit{Set}\ a \end{aligned}$$

Motivating example: 2-level iterated majority maj_3^2:

$$\begin{aligned} ps_9 &= \mathit{genAlgThinMemo}\ 9\ maj_3^2 :: \mathit{Set}\ (\mathit{Poly}\ \mathbb{Q}) \\ check_9 &= ps_9 == \mathit{fromList}\ [P\ [4,4,6,9,-61,23,67,-64,16]] \end{aligned}$$

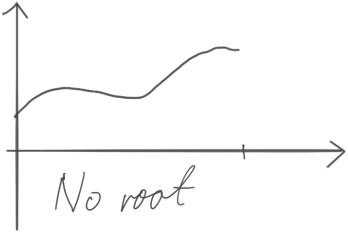
Comparing polynomials

- One challenge was (exact) comparison of polynomials.
- Remember that the "complexity" we compute for each decision tree is actually a polynomial in the probability for a bit to be $\mathbf{1}$.
- And the mathematical definition of the level-p-complexity includes a "min" over all $DecTree$s, which in general results in a piecewise polynomial function.
- Ideally we should have a partition of the unit interval, labelled with polynomials, but we choose to represent such piecewise polynomial function simply as a set of polynomials.

Comparing polynomials, cont.

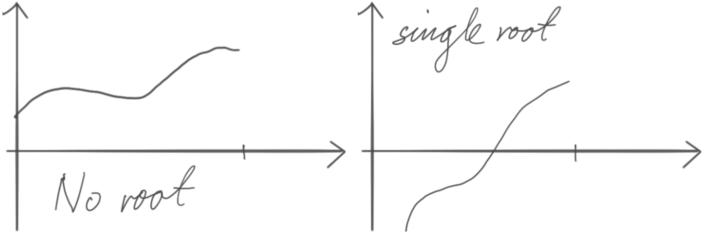
- For thinning we need a preorder on our polynomials, specified as $l \leqslant r = \forall\, p.\ \mathit{eval}\ l\ p \leqslant \mathit{eval}\ r\ p$.
- This can be simplified to $\forall\, p.\ \mathit{eval}\ (r - l)\ x \geqslant 0$ where $r - l$ is also a polynomial.
- We started with some ad-hoc special cases, but the sets did not shrink enough.
- Then we went to the wonderful world of polynomial algebra and root-counting.

Root-counting

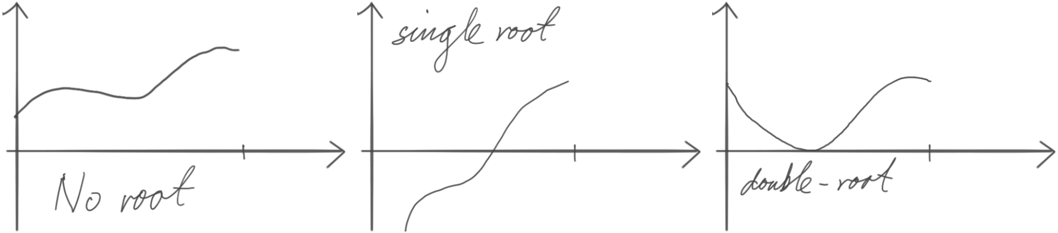


- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.

Root-counting

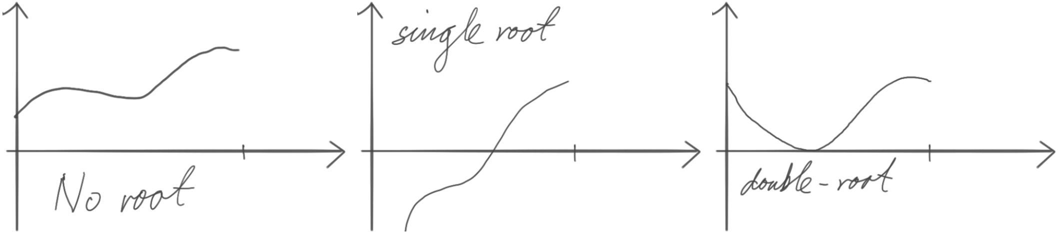


- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $l \leqslant r$ nor $r \leqslant l$).



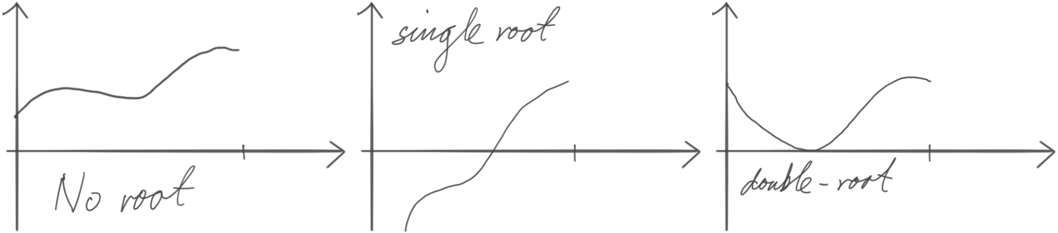
- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $l \leqslant r$ nor $r \leqslant l$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.

Root-counting



- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $l \leqslant r$ nor $r \leqslant l$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.

Root-counting



- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $l \leqslant r$ nor $r \leqslant l$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.
- Note that we do not need to actually compute the roots, only count them.

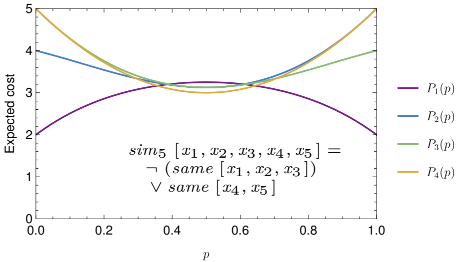
Root-counting

- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $l \leqslant r$ nor $r \leqslant l$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.
- Note that we do not need to actually compute the roots, only count them.
- We use Yun's algorithm for square-free factorisation and Descartes rule of signs in combination with interval-halving.

Conclusions

- By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

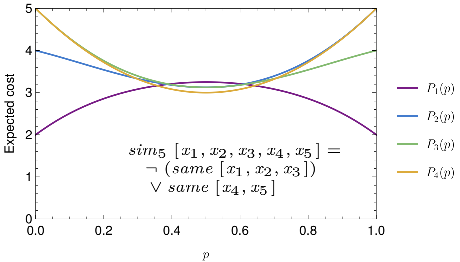
$$
\begin{aligned}
&ps_9 = genAlgThinMemo\ 9\ \ maj_3^2 :: Set\ (Poly\ \mathbb{Q}) \\
&check_9 = ps_9 == fromList\ [\ P\ [4,4,6,9,-61,23,67,-64,16]] \\
&ps5 = genAlgThinMemo\ 5\ sim_5 :: Set\ (Poly\ \mathbb{Q}) \\
&check5 = ps5 == fromList\ [P\ [2,6,-10,8,-4], P\ [4,-2,-3,8,-2], \\
&\qquad\qquad\qquad\qquad\qquad P\ [5,-8,9,0,-2],\ \ P\ [5,-8,8]]
\end{aligned}
$$



Conclusions

- By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

$$
\begin{aligned}
&ps_9 = genAlgThinMemo\ 9\ \ maj_3^2 :: Set\ (Poly\ \mathbb{Q}) \\
&check_9 = ps_9 == fromList\ [\ P\ [4,4,6,9,-61,23,67,-64,16]] \\
&ps5 = genAlgThinMemo\ 5\ sim_5 :: Set\ (Poly\ \mathbb{Q}) \\
&check5 = ps5 == fromList\ [P\ [2,6,-10,8,-4], P\ [4,-2,-3,8,-2], \\
&\qquad\qquad\qquad\qquad\qquad P\ [5,-8,9,0,-2],\ \ P\ [5,-8,8]]
\end{aligned}
$$



Future work:

- Agda proof
- More efficient memoization
- Three-level iterated majority (27 bits)