Level-p-complexity of Boolean functions using thinning, memoization, and polynomials

JFP **33**, e13, 21 pages, 2023.

doi:10.1017/S0956796823000102

Level-p-complexity of Boolean functions using thinning, memoization, and polynomials

JULIA JANSSON
Chalmers University of Technology and University of Gothenburg, Göteborg, Sweden
(*e-mail:* juljans@chalmers.se)

PATRIK JANSSON
Chalmers University of Technology and University of Gothenburg, Göteborg, Sweden
(*e-mail:* patrikj@chalmers.se)

Abstract

This paper describes a purely functional library for computing level-p-complexity of Boolean functions and applies it to two-level iterated majority. Boolean functions are simply functions from n bits to one bit, and they can describe digital circuits, voting systems, etc. An example of a Boolean function is majority, which returns the value that has majority among the n input bits for odd n. The complexity of a Boolean function f measures the *cost* of evaluating it: how many bits of the input are needed to be certain about the result of f. There are many competing complexity measures, but we focus on level-p-complexity — a function of the probability p that a bit is 1. The level-p-complexity $D_p(f)$ is the minimum expected cost when the input bits are independent and identically distributed with Bernoulli(p) distribution. We specify the problem as choosing the minimum expected cost of all possible decision trees — which directly translates to a clearly correct, but very inefficient implementation. The library uses thinning and memoization for efficiency and type classes for separation of concerns. The complexity is represented using (sets of) polynomials, and the order relation used for thinning is implemented using polynomial factorization and root counting. Finally, we compute the complexity for two-level iterated majority and improve on an earlier result by J. Jansson.

1 Introduction

Imagine a voting system with yes/no options, for example, direct democracy, indirect democracy, or dictatorship. How much information of the votes do we need until we can conclude the outcome of the election? For dictatorship, we only need the information of the dictator as he or she has all the power, but for a democratic majority we need at least half the votes. Depending on the order in which we find out what the votes are we might need all of them before we can conclude the result. More generally, this question is about complexity of Boolean functions which is application area of this paper.

Boolean functions are widespread in mathematics and computer science and can describe yes-no voter systems, hardware circuits, and predicates (Knuth, 2012; O'Donnell, 2014). A Boolean function is a function from n bits to one bit, for example, majority (maj_n), which returns the value that has majority among the n inputs. In the context of

voting systems, the next subsection gives an example of a Boolean function called iterated majority.

1.1 Vote counting example: iterated majority

In US elections, a presidential candidate can lose even if they win the popular vote. One reason for this is that the outcome is not directly determined by the majority, but rather majority iterated two times.[1] Our running example is a very much simplified case: consider 3 states with 3 voters in each.

$$\underbrace{\underbrace{x_{(0,0)}, x_{(0,1)}, x_{(0,2)}}_{m_0 = maj_3(\ldots)}, \underbrace{x_{(1,0)}, x_{(1,1)}, x_{(1,2)}}_{m_1 = maj_3(\ldots)}, \underbrace{x_{(2,0)}, x_{(2,1)}, x_{(2,2)}}_{m_2 = maj_3(\ldots)}}_{maj_3(m_0, m_1, m_2)}$$

We first compute the majority m_i in each "state", and then the majority of m_0, m_1, and m_2. For example we see below $\mathbf{0}, \mathbf{1}, \mathbf{0}$ which gives $m_0 = \mathbf{0}$, then $\mathbf{1}, \mathbf{0}, \mathbf{1}$ which gives $m_1 = \mathbf{1}$, and $\mathbf{0}, \mathbf{1}, \mathbf{0}$ again which gives $m_2 = \mathbf{0}$. The final majority is $\mathbf{0}$:

$$\underbrace{\underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_0 = \mathbf{0}}, \underbrace{\mathbf{1}, \mathbf{0}, \mathbf{1}}_{m_1 = \mathbf{1}}, \underbrace{\mathbf{0}, \mathbf{1}, \mathbf{0}}_{m_2 = \mathbf{0}}}_{maj_3 = \mathbf{0}}$$

But if we switch the first and 8th bit (perhaps through gerrymandering) we get another result (with the changed bits underlined and marked in red):

$$\underbrace{\underbrace{\underline{\mathbf{1}}, \mathbf{1}, \mathbf{0}}_{m_0 = \underline{\mathbf{1}}}, \underbrace{\mathbf{1}, \mathbf{0}, \mathbf{1}}_{m_1 = \mathbf{1}}, \underbrace{\mathbf{0}, \underline{\mathbf{0}}, \mathbf{0}}_{m_2 = \mathbf{0}}}_{maj_3 = \underline{\mathbf{1}}}$$

This changes m_0 from $\mathbf{0}$ to $\mathbf{1}$ without affecting m_1, or m_2. But now the two-level majority is changed to $\mathbf{1}$, just from the switch of two bits. Both examples have four $\mathbf{1}$'s and five $\mathbf{0}$'s but the result is different based on the positioning of the bits. In our case the two-level majority is $\mathbf{1}$ even though there are fewer $\mathbf{1}$'s than $\mathbf{0}$'s. This means that the $\mathbf{0}$'s "lose" even though they won the "popular vote".

1.2 Cost and complexity

The field of computational complexity is about "how much" computation is necessary and sufficient to perform certain computational tasks. For example, given a computational problem it tries to establish tight upper and lower bounds on the length of the computation (or on other resources, like space). Unfortunately, for many practically relevant computational problems no tight bounds are known. In our case we study one of the simplest models of computation: the decision tree. We are interested in the cost of evaluating Boolean functions, and we use binary decision trees to describe the evaluation order of Boolean functions. The depth of the decision tree corresponds to the number of votes needed to know the outcome for certain. This is called deterministic complexity. Another

[1] The actual presidential election is a direct majority vote among the electors who are not formally bound by their states' outcome.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

well-known notion is randomized complexity, and the randomized complexity bounds of iterated majority have been studied in Landau *et al*. (2006), Leonardos (2013), and Magniez *et al*. (2016). Iterated majority on two levels corresponds to the Boolean function for US elections as described above. We are particularly interested in this function due to its symmetry and simplicity, but still the complexity is non-trivial.

Diving into the literature for complexity of Boolean functions we find many different measures. Relevant concepts are certificate complexity, degree of a Boolean function, and communication complexity (Buhrman & De Wolf, 2002). Complexity measures related specifically to circuits are circuit complexity, additive, and multiplicative complexity (Wegener, 1987). Considering Boolean computation in practice we have combinational complexity which is the length of the shortest Boolean chain computing it (Knuth, 2012). Thus, there are many competing complexity measures, but we focus on level-p-complexity — a function of the probability p that a bit is **1** (Garban & Steif, 2014). We assume that the bits are independent and identically Bernoulli-distributed with parameter $p \in [0, 1]$. Then, for each Boolean function f and probability p, we get the level-p-complexity by minimizing the expected cost over all decision trees. The level-p-complexity is a piecewise polynomial function of p and has many interesting properties (Jansson, 2022).

1.3 Contributions

This paper presents a purely functional library for computing level-p-complexity of Boolean functions in general and for maj_3^2 in particular. The level-p-complexity of maj_3^2 was conjectured in Jansson (2022), but could not be proven because it was hard to generate all possible decision trees. This paper fills that gap by showing that the conjecture is false and by computing the true level-p-complexity of maj_3^2.

The strength of our implementation is that it can calculate the level-p-complexity for Boolean functions quickly and correctly, compared to tedious calculations by hand. Our specification uses exhaustive search and considers all possible candidates (decision trees). Some partial candidates dominate (many) others, which may be discarded. Thinning (Bird & Gibbons, 2020) is an algorithmic design technique which maintains a small set of partial candidates which provably dominate all other candidates. We hope that one contribution of this paper is an interesting example of how a combination of algorithmic techniques can be used to make the intractable tractable. The code in this paper is available on GitHub[2] and uses packages from Jansson *et al*. (2022). The implementation is in Haskell but should work also in other languages, and parts of it has been reproduced in Agda to check some of the stronger invariants. The choice of Haskell for the implementation is due to its strong compiler and the availability of libraries for BDDs, memoization, and polynomials.

1.4 Motivation

To give the flavor of the end result, we start with two examples which will be explained in detail later: the level-p-complexity of 2-level iterated majority maj_3^2 and of a 5-bit function we call sim_5, defined in Figure 1. The level-p-complexity is a piecewise polynomial function of the probability p and sim_5 is the smallest arity Boolean function we have found which has more than one polynomial piece contributing to the complexity. Polynomials are

[2] The paper repository is at `https://github.com/juliajansson/BoFunComplexity`.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

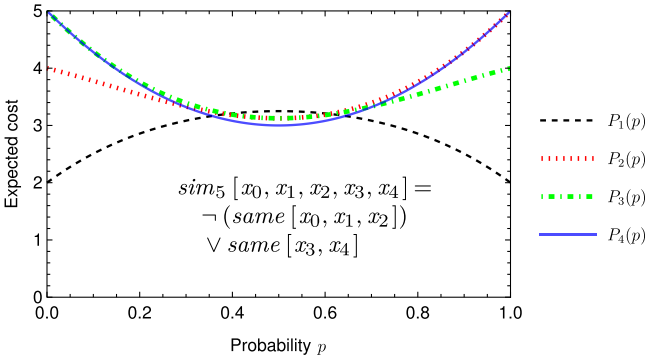

Fig. 1. The four polynomials computed by *genAlgThinMemo* 5 sim_5.

represented by their coefficients: for example, $P\,[5, -8, 8]$ represents $5 - 8x + 8x^2$. The function *genAlgThinMemo* uses thinning and memoization to generate a set of minimal cost polynomials.

$$ps5 = genAlgThinMemo\ 5\ sim_5 :: Set\ (Poly\ \mathbb{Q})$$
$$check5 = ps5 == fromList\,[P\,[2, 6, -10, 8, -4], P\,[4, -2, -3, 8, -2], P\,[5, -8, 9, 0, -2], \quad P\,[5, -8, 8]]$$

The graph, in Figure 1, shows that different polynomials dominate in different intervals. The polynomial P_1 is best near the end points, but P_4 is best near $p = 1/2$ (despite being really bad near the end points). The level-p-complexity is the piecewise polynomial minimum, a combination of P_1 and P_4. This computation can be done by exhaustive search over the 54192 different decision trees and 39 resulting polynomials, but for more complex Boolean functions the doubly exponential growth makes that impractical.

For our running example, maj_3^2, a crude estimate indicates we would have 10^{111} decision trees to search and very many polynomials. Thus, the computation would be intractable if it were not for the combination of thinning, memoization, and symbolic comparison of polynomials. Thanks to symmetries in the problem there turns out to be just one dominating polynomial, called P_* in Figure 2, computed by:

$$ps9 = genAlgThinMemo\ 9\ maj_3^2 :: Set\ (Poly\ \mathbb{Q})$$
$$check9 = ps9 == fromList\,[P\,[4, 4, 6, 9, -61, 23, 67, -64, 16]]$$

The graph in Figure 2, shows that only 4 bits are needed in the limiting cases of $p = 0$ or 1 and that just over 6 bits are needed in the maximum at $p = 1/2$. Figure 2 also shows the conjectured complexity polynomial P_t from Jansson (2022), and Figure 3 shows the (small) difference between the two polynomials.

2 Background

To explain what level-p-complexity of Boolean functions means, we introduce some background about Boolean functions, decision trees, cost, and complexity. The Boolean input type $\mathbb{B}$ could be $\{False, True\}$, $\{F, T\}$ or $\{0, 1\}$ and from now on we use **0** for false and **1** for true in our notation. In the running text, we write $e : t$ for "e has type t" which in the quoted Haskell code is written $e :: t$.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

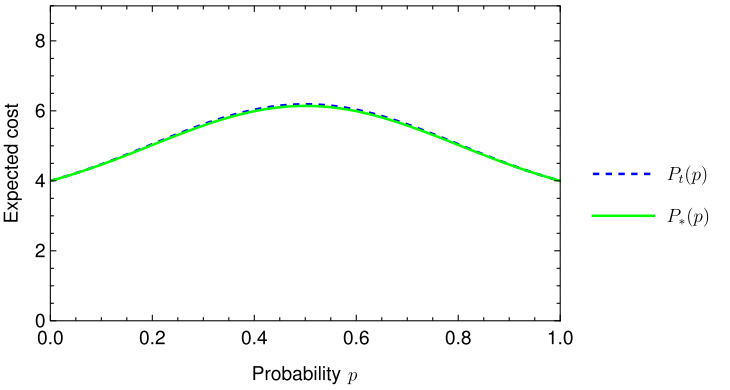

Fig. 2. Expected costs of the two different decision trees. Because they are very close we also show their difference in Figure 3.

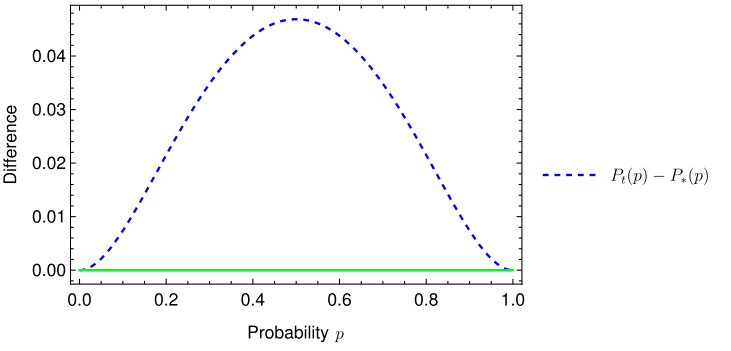

Fig. 3. Difference between the conjectured (P_t) and the true (P_*) complexity of maj_3^2.

2.1 Boolean functions

A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ is a function from n Boolean inputs to one Boolean output. We sometimes write *BoolFun n* for the type $\mathbb{B}^n \to \mathbb{B}$. The easiest examples of Boolean functions are the functions $const_n\ b$ which ignore the n input bits and return b. The usual logical gates like and_n and or_n are very common Boolean functions. Another example is the dictator function (also known as first projection), which is defined as $dict_{n+1}\ [x_0, ..., x_n] = x_0$ when the dictator is bit 0.

A naive representation of a Boolean function could be a pair of an arity and a function $f : [\mathbb{B}] \to \mathbb{B}$, but that turns out to be inefficient when we want to compare and tabulate them (see Section 3.3). Instead we use binary decision diagrams, *BDD*s (Bryant, 1986) as implemented in Masahiro Sakai's excellent Hackage package[3]. The package reimplements all the usual Boolean operations on what is semantically expressions in n Boolean variables. BDDs are an efficient way of representing Boolean functions, and they can be used for testing, verification, and complexity analysis. For readability, we will present Boolean functions in the naive representation, but the actual code uses the type *BDD a* from the

[3] https://github.com/msakai/haskell-decision-diagrams

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

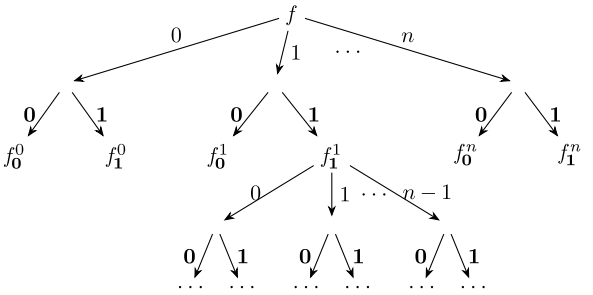

Fig. 4. The tree of subfunctions of a Boolean function f. This tree structure is also the call graph for our generation of decision trees. Note that this tree structure is related to, but not the same as, the decision trees.

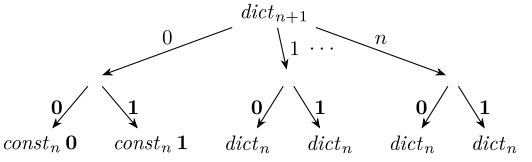

Fig. 5. The tree of subfunctions of the $dict_{n+1}$ function.

BDD package (where a keeps track of variable ordering). Note that we only use BDDs to represent our Boolean functions, not our decision trees.

In the complexity computation, we only need two operations on Boolean functions which we capture in the following type class interface:

```
class BoFun bf where
  isConst :: bf → Maybe 𝔹
  setBit  :: Index → 𝔹 → bf → bf
type Index = ℕ
```

The use of a type class here means we keep the interface to the BDD implementation minimal, which makes proofs easier and gives better feedback from the type system. The first method, $isConst\ f$, returns $Just\ b$ iff the function f is constant and always returns $b : \mathbb{B}$. The second method, $setBit\ i\ b\ f$, restricts a Boolean function (on $n + 1$ bits) by setting its ith bit to b. The result is a "subfunction" on the remaining n bits, abbreviated f_b^i, and illustrated in Figure 4.

As an example, for the function and_2 we have that $setBit\ i\ \mathbf{0}\ and_2 = const_1\ \mathbf{0}$ and $setBit\ i\ \mathbf{1}\ and_2 = id$. For and_2 we get the same result for $i = 0$, or 1 but for the dictator function it depends if we pick the dictator index (0) or not. We get $setBit\ 0\ b\ dict_{n+1} = const_n\ b$, because the result is dictated by bit 0. Otherwise, we get $setBit\ (i + 1)\ b\ dict_{n+1} = dict_n$ irrespective of the value of b since only the value of the dictator bit matters. This behavior is shown in Figure 5.

2.2 Decision trees

Consider a decision tree that picks the n bits of a Boolean function f in a deterministic way depending on the values of the bits picked further up the tree. Decision trees are referred

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

to as algorithms in Landau *et al*. (2006); Garban & Steif (2014); Jansson (2022). Given a Boolean function f, a decision tree t describes one way to evaluate the function f. The Haskell datatype is as follows:

```
data DecTree = Res 𝔹 | Pick Index DecTree DecTree
  deriving (Eq, Ord, Show)
```

Parts of the "rules of the game" in the mathematical literature are that you must return a *Res*ult if the function is constant and you may only *Pick* an index once. We can capture most of these rules with a type family version of the *DecTree* datatype (here expressed in *Agda* syntax). Here we use two type indices: $t : DecTree\ n\ f$ is a decision tree for the Boolean function f, of arity n. The *Res* constructor may only be used for constant functions (but for any arity), while *Pick i* takes two subtrees for Boolean functions of arity n to a tree of arity $suc\ n = n + 1$.

```
data DecTree : (n : ℕ) → (f : BoolFun n) → Set where
  Res  :                          (b : 𝔹) →          DecTree n (const_n b)
  Pick : {f : BoolFun (suc n)} → (i : Fin (suc n)) → DecTree n (setBit i 0 f) →
                                                     DecTree n (setBit i 1 f) →
                                                     DecTree (suc n) f
setBit : Fin (suc n) → 𝔹 → BoolFun (suc n) → BoolFun n
```

Note that the dependently typed version of *setBit* clearly indicates that the resulting function $g = (setBit\ i\ b\ f) : BoolFun\ n$ has arity one less that of $f : BoolFun\ (suc\ n)$. This helps maintaining the invariant that each input bit may only be picked once.[4] We use the Haskell versions, but the Agda versions capture the invariants better.

We can use these rules backward to generate all possible decision trees for a certain function. If the function is constant, returning $b : \mathbb{B}$, we immediately know that the only decision tree allowed is *Res b*. If it is not constant, we pick any index i, any decision tree t_0 for the subfunction $setBit\ i\ \mathbf{0}\ f$ and t_1 for the subfunction $setBit\ i\ \mathbf{1}\ f$ recursively. We get back to this in Section 3.1 after some preparation.

Note that we do **not** use binary decision diagrams (BDDs) to represent our decision trees. An example of a decision tree for the majority function maj_3 on three bits is defined by the expression *ex1* visualised in Figure 6.

```
ex1 = Pick 0 (Pick 2 (Res 0) (Pick 1 (Res 0) (Res 1)))
             (Pick 1 (Pick 2 (Res 0) (Res 1)) (Res 1))
```

We will define several functions as folds over *DecTree* and to do that we introduce a type class *TreeAlg* (for "Tree Algebra") which collects the two methods *res* and *pic* which are then used in the fold to replace the constructors *Res* and *Pick*.

```
class TreeAlg a where
  res :: 𝔹 → a
  pic :: Index → a → a → a
foldDT :: TreeAlg a ⇒ DecTree → a
```

[4] The use of *Fin n* also means that the interpretation of indices is local: the 3-bit example $ex0 = Pick\ 0\ (Pick\ 0\ (Res\ \mathbf{0}))\ (Pick\ 1\ (Res\ \mathbf{1}))$ in Agda corresponds to the global interpretation $Pick\ 0\ (Pick\ 1\ (Res\ \mathbf{0}))\ (Pick\ 2\ (Res\ \mathbf{1}))$. We use the global view in *ex1* and figures for readability.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

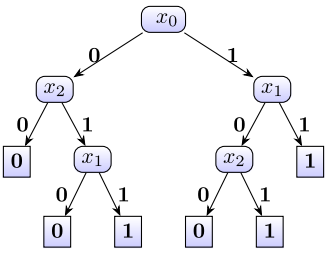

Fig. 6. An example of a decision tree for maj_3. The root node branches on the value of bit 0. If it is **0**, it picks bit 2, while if it is **1**, it picks bit 1. It then picks the last remaining bit if necessary.

```
foldDT (Res b)        = res b
foldDT (Pick i t0 t1) = pic i (foldDT t0) (foldDT t1)
```

The *TreeAlg* class is used to define our decision trees but also for several other purposes. (In the implementation, we additionally require some total order on a to enable efficient set computations.) We see that our decision tree type is the initial algebra of *TreeAlg* and that we can reimplement a generic version of *ex1* which can be instantiated to any *TreeAlg* instance:

```
instance TreeAlg DecTree where res = Res; pic = Pick;

ex1 :: TreeAlg a ⇒ a
ex1 = pic 0 (pic 2 (res 0) (pic 1 (res 0) (res 1)))
            (pic 1 (pic 2 (res 0) (res 1)) (res 1))
```

2.3 Expected cost

For a function f and a specific input $xs : \mathbb{B}^n$, the cost of evaluating f according to a decision tree t is the length of the path from root to leaf dictated by the bits in xs. We then let the bits be independent and identically distributed with probability $p \in [0, 1]$ for **1** and compute the *expected* cost (averaging over all 2^n inputs). Expected cost can be implemented as an instance of *TreeAlg*.

```
newtype Poly a = P [a]
type ExpCost a = Poly a
instance Ring a ⇒ TreeAlg (ExpCost a) where res = resPoly; pic = pickPoly

expCost :: Ring a ⇒ DecTree → Poly a
expCost = foldDT
```

Note that the expected cost of any decision tree for a Boolean function of n bits will always be a polynomial. We represent polynomials as lists of coefficients: $P\,[1, 2, 3]$ represents $\lambda p \to 1 + 2 \times p + 3 \times p^2$ and use $evalP : Ring\ a \Rightarrow Poly\ a \to (a \to a)$ to evaluate polynomials. The polynomial implementation borrowed from Jansson *et al.* (2022) includes the polynomial ring operations ($(+)$, $(-)$, $(\times)$), *gcd*, *divMod*, symbolic derivative, and ordering. The *res* and *pic* functions are as follows:

```
resPoly :: Ring a ⇒ B → a
resPoly b = zero
```

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

```
pickPoly :: Ring a ⇒ Index → Poly a → Poly a → Poly a
pickPoly i q0 q1 = one + (one − xP) × q0 + xP × q1
```

Here $zero = P\,[\,]$ and $one = P\,[1]$ represent *const* 0 and *const* 1, respectively, while $xP = P\,[0, 1]$ is "the polynomial x". For $pickPoly\ \_\ q_0\ q_1$, we first have to pick one bit and then if this bit is **0** (with probability $\mathbb{P}(x_i = \mathbf{0}) = (1 - p)$) we get q_0 which is the polynomial for this case. If the bit is instead **1** (with probability $\mathbb{P}(x_i = \mathbf{1}) = p$) we get q_1. The expected cost of the decision tree *ex1* is $2 + 2p - 2p^2$. From now on we will use Haskell's overloading to write 0 and 1 for *zero* and *one* even when working with polynomials.

2.4 Complexity

Now that we have introduced expected cost, we can introduce the level-p-complexity $D_p(f)$ as the pointwise minimum of the expected cost over all of f's decision trees:

```
D_p(f) = minimum {evalP (expCost t) p | t ← genAlg_n f}
genAlg_n :: (BoFun bf, TreeAlg a, Ord a) ⇒ bf → Set a
```

where the generation of decision trees is explained in Section 3.1. When minimizing we do not necessarily get a polynomial, but a piecewise polynomial function. For simplicity, we represent a piecewise polynomial function as a set of polynomials:

```
type PPoly a = Set (Poly a)
evalPP :: (Ring a, Ord a) ⇒ PPoly a → (a → a)
evalPP qs p = minimum (map (λq → evalP q p) qs)
```

This representation will be inefficient if the set is big, but as a specification it works fine and we will later use thinning to keep the set small (see Sections 3.2 and 3.4). We say that one polynomial q is "uniformly worse" than another polynomial p when $p\,x \leqslant q\,x$ for all $0 \leqslant x \leqslant 1$ and $p\,x < q\,x$ for some $0 < x < 1$. For some polynomials, we cannot determine which is worse, see Figure 1 where four polynomials all intersect. In this case, they are incomparable.

When computing the level-p-complexity, it would be possible to take both f and the probability p as arguments and return the smallest expected cost for that probability, but we prefer to just take f as an argument and compute a piecewise polynomial function representation. In this way, we can analyze the result symbolically to find minima, maxima, number of polynomial pieces, etc.

2.5 Examples of Boolean functions and their costs

Now that we have introduced expected cost and level-p-complexity, we give a few examples of Boolean functions and their costs to give a feeling of how the computations work. The impatient reader can skip forward to Section 3. As mentioned earlier (in Section 2.1), we present the Boolean functions as Haskell functions for readability, but every example has a BDD counterpart.

For the constant functions ($const_n\ b$), there is just one legal decision tree $t = Res\ b$ and thus $expCost\ t = 0$ which gives $D_p(const_n\ b) = 0$. For the dictator function, there are many decision trees, but as we can see in Figure 5, picking bit 0 first is optimal and gets us to the

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

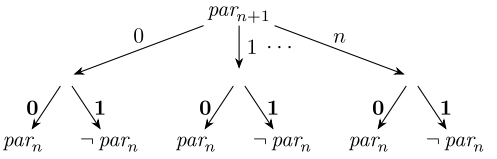

Fig. 7. The recursive structure of the parity function (par_n). The pattern repeats all the way down to $par_0 = const_0\ \mathbf{0}$.

constant case just covered. Thus, the optimal tree is $optTree = Pick\ 0\ (Res\ \mathbf{0})\ (Res\ \mathbf{1})$, and we can compute the expected cost as follows.

$$expCost\ optTree = 1 + (1 - xP) \times 0 + xP \times 0 = 1\ .$$

which gives $D_p(dict_n) = 1$.

The parity function can be defined as

$$count :: Eq\ a \Rightarrow a \to [a] \to Int$$
$$count\ x = length \circ filter\ (x ==)$$
$$par_n :: \mathbb{B}^n \to \mathbb{B}$$
$$par_n = odd \circ count\ \mathbf{1}$$

In this case, all bits have to be picked to determine the parity, regardless of input. We prove that for all decision trees t of par_n or $\neg\, par_n$, we have that $expCost\ t = n$ using induction over n. For the base case, $n = 0$ we have that $par_0 = const_0\ \mathbf{0}$ and $\neg\, par_0 = const_0\ \mathbf{1}$ so that $expCost\ t = 0$ for all decision trees t as shown above. For the induction step we assume that for all decision trees t of par_n or $\neg\, par_n$ we have that $expCost\ t = n$ and show that for all decision trees t of par_{n+1} or $\neg\, par_{n+1}$ we have that $expCost\ t = n + 1$. Any decision tree for par_{n+1} or $\neg\, par_{n+1}$ is of the form $Pick\ i\ t_0\ t_1$ where t_0 and t_1 are decision trees for par_n or $\neg\, par_n$ as seen in Figure 7.

To calculate the expected cost, we get

$$\begin{aligned} expCost\ (Pick\ i\ t_0\ t_1) &= 1 + (1 - xP) \times (expCost\ t_0) + xP \times (expCost\ t_1) \\ &= 1 + (1 - xP) \times n + xP \times n = 1 + n \end{aligned}$$

Thus, the induction proof is complete and as $expCost\ t = n$ for all decision trees then also the minimum is n, thus $D_p(par_n) = n$. Comparing Figure 5 with Figure 7, we see that the minimum depth of the dictator tree is 1, while the minimum depth of the parity tree is n. The parity function and the constant function are interesting extreme cases of Boolean functions as they have highest and lowest possible level-p-complexity n and 0. Either all bits have to be picked to determine the parity or none of them need to be picked to determine the constant function.

We now introduce the Boolean function *same* which checks if all bits are equal:

$$same :: \mathbb{B}^n \to \mathbb{B}$$
$$same\ bs = and\ bs \vee \neg\, (or\ bs)$$

Using *same* we construct the example sim_5 from the introduction. We first split the bits into two groups, one with the first three bits and the second with the last two bits. On the first group, called *as*, we check if the bits are not the same, and on the second group, called *cs* we check if the bits are the same.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

$$
\begin{aligned}
&sim_5 :: \mathbb{B}^5 \to \mathbb{B}\\
&sim_5\ bs = \neg\,(same\ as) \vee same\ cs\\
&\quad \textbf{where}\ (as, cs) = splitAt\ 3\ bs
\end{aligned}
$$

The point of this function is to illustrate a special case where the best decision tree depends on p so that the level-p-complexity consists of more than one polynomial piece. This computation is shown in Section 4.1.

One of the major goals of this paper was to calculate the level-p-complexity of 9 bit iterated majority called maj_3^2. When extending the majority function to maj_3^2, we use maj_3 inside maj_3.

$$
\begin{aligned}
&maj_3^2 :: \mathbb{B}^9 \to \mathbb{B}\\
&maj_3^2\ bs = maj_3\ [maj_3\ bs_0, maj_3\ bs_1, maj_3\ bs_2\,]\\
&\quad \textbf{where}\ (bs_0, rest) = splitAt\ 3\ bs\\
&\qquad\quad\ \ (bs_1, bs_2\ ) = splitAt\ 3\ rest\\
&maj_n :: \mathbb{B}^n \to \mathbb{B}\\
&maj_n\ bs = count\ \mathbf{1}\ bs \geqslant count\ \mathbf{0}\ bs
\end{aligned}
$$

It is hard to calculate $D_p(maj_3^2)$ by hand because there are very many different decision trees, and this motivated our Haskell implementation.

3 Computing the level-p-complexity

In this section, we explain how to compute the level-p-complexity of a Boolean function f by recursively "generating all candidates" followed by "picking the best one(s)". The naive approach would be to generate all decision trees of f and then minimizing, but already for the 9-bit function maj_3^2 that is intractable. To reduce the number of polynomials, we use the algorithm design technique thinning. We compare polynomials by using Yun's algorithm and Descartes rule of signs. Further, since the same subfunctions often appear in many different nodes, we can save a significant amount of computation time using memoization.

The top level complexity computation (from Section 2.4) can be simplified a bit:

$$
\begin{aligned}
D_p(f) &= minimum\ \{evalP\ (expCost\ t)\ p \mid t \leftarrow genAlg_n\ f\}\\
&\qquad \{\ \text{comprehension syntax}\ \}\\
&= minimum\ (map\ (\lambda t \to evalP\ (expCost\ t)\ p)\ (genAlg_n\ f))\\
&\qquad \{\ map\ (g \circ h) = map\ g \circ map\ h\ \}\\
&= minimum\ (map\ (\lambda q \to evalP\ q\ p)\ (map\ expCost\ (genAlg_n\ f)))\\
&\qquad \{\ \text{fuse } expCost \text{ into the tree algebra generation}\ \}\\
&= minimum\ (map\ (\lambda q \to evalP\ q\ p)\ (genAlg_n\ f))\\
&\qquad \{\ \text{let } best\ p = minimum \circ map\ (\lambda q \to evalP\ q\ p)\ \}\\
&= best\ p\ (genAlg_n\ f)
\end{aligned}
$$

and we start by explaining $genAlg_n$. The decision trees of a function f can be described in terms of the decision trees for the immediate subfunctions ($f_b^i = setBit\ i\ b\ f$) for different $i : Index$ and $b : \mathbb{B}$. In fact, we can immediately generate elements of any tree algebra, not only decision trees, by using *res* and *pic* instead of *Res* and *Pick*. (That is used in the "fuse" step of the calculation above.) When we explain the algorithm we write "decision tree" to make it feel more concrete, but we will in the end mostly use it to directly compute expected cost polynomials.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

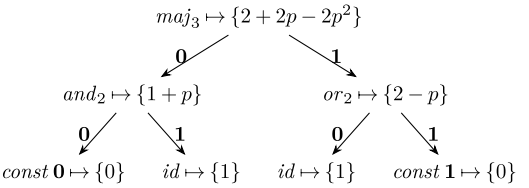

Fig. 8. A simplified computation tree of $genAlg_3\ maj_3$. In each node, $f \mapsto ps$ shows the input f and output $ps = genAlg_n\ f$ of each local call. As all the functions involved are "symmetric" in the index ($setBit\ i\ b\ f == setBit\ j\ b\ f$ for all i and j), we only show edges for **0** and **1** from each level.

3.1 Generating decision trees and other tree algebras

The complexity computation starts from a Boolean function $f : BoolFun\ n$ and generates all decision trees for it. There are two top level cases: either the function f is constant (and returns $b : \mathbb{B}$), in which case there is only one decision tree: $res\ b$; or the function f still depends on some of the input bits (and thus the arity is at least 1). In the latter case, for each index $i : Index$, we can generate two subfunctions $f^i_{\mathbf{0}} = setBit\ i\ \mathbf{0}\ f$ and $f^i_{\mathbf{1}} = setBit\ i\ \mathbf{1}\ f$. We then recursively generate a decision tree t_0 for $f^i_{\mathbf{0}}$ and t_1 for $f^i_{\mathbf{1}}$ and combine them to a bigger decision tree using $pic\ i\ t_0\ t_1$. This is done for all combinations of i, t_0, and t_1 in a set comprehension. To make it easier to later extend the definition (for thinning and memoization), we make the recursive step explicit.

$$
\begin{aligned}
&genAlg :: (BoFun\ bf, TreeAlg\ a, Ord\ a) \Rightarrow \mathbb{N} \to bf \to Set\ a\\
&genAlg = genAlgStep\ genAlg\\
&genAlgStep :: (BoFun\ bf, TreeAlg\ a, Ord\ a) \Rightarrow (\mathbb{N} \to bf \to Set\ a) \to (\mathbb{N} \to bf \to Set\ a)\\
&genAlgStep\ genA\ n \qquad f \mid Just\ b \leftarrow isConst\ f = \{res\ b\}\\
&genAlgStep\ genA\ (n+1)\ f = \{pic\ i\ t_0\ t_1 \mid i \leftarrow \{0\,..\,n\}, t_0 \leftarrow genA\ n\ f^i_{\mathbf{0}}, t_1 \leftarrow genA\ n\ f^i_{\mathbf{1}}\}
\end{aligned}
$$

We would like to enumerate the cost polynomials of all the decision trees of a particular Boolean function ($n = 9$, $f = maj^2_3$ is our main goal). Without taking symmetries into account, there are $2 \times n$ immediate subfunctions f^i_b and if T_g is the cardinality of the enumeration for subfunction g we have that

$$T_f = \sum_{i=0}^{n-1} T_{f^i_{\mathbf{0}}} \times T_{f^i_{\mathbf{1}}}$$

These numbers can be really big if we count all decision trees, but if we only care about their cost polynomials, many decision trees will collapse to the same polynomial, making the counts more manageable (but still possibly really big). Even the total number of subfunctions encountered (the number of recursive calls) can be quite big. If all the $2 \times n$ immediate subfunctions are different, and if all of them would generate $2 \times (n - 1)$ different subfunctions in turn, the number of subfunctions would be $2^n \times n!$. But in practice many subfunctions will be the same. When computing the polynomials for the 9-bit function maj^2_3, for example, only 215 distinct subfunctions are encountered.

As a smaller example, for the 3-bit majority function maj_3, choosing $i = 0$, 1, or 2 gives exactly the same subfunctions. Figure 8 illustrates a simplified call graph of $genAlg_3\ maj_3$ and the results (the expected cost polynomials) for the different subfunctions. In this case,

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

all the sets are singletons, but that is very unusual for more realistic Boolean functions. It would take too long to compute all polynomials for the 9-bit function maj_3^2, but there are 21 distinct 7-bit subfunctions, and the first one of them already has 18021 polynomials. Thus, we can expect billions of polynomials for maj_3^2, and this means we need to look at ways to keep only the most promising candidates at each level. This leads us to the algorithmic design technique of thinning.

3.2 Thinning

The general shape of the specification has two phases: "generate all candidates" followed by "pick the best one(s)." The first phase is recursive, and we would like to push as much as possible of "pick the best" into the recursive computation. In the extreme case of a greedy algorithm, we can thin the intermediate sets all the way down to singletons, but even if the sets are a bit bigger than that we can still reduce the computation cost significantly. A good (but abstract) reference for thinning is the Algebra of Programming book (Bird & de Moor, 1997, Chapter 8) and more concrete references are the corresponding developments in Agda (Mu *et al*., 2009) and Haskell (Bird & Gibbons, 2020). In this subsection, the main focus is on specification and correctness, with Agda-like syntax for the logic part.

The "pick the best" phase is $best\, p = minimum \circ map\, (\lambda q \to evalP\, q\, p)$ of type $Set\, (Poly\, r) \to r$ for some ring of scalars r (usually rational numbers). In this context, it is clear that in the generation phase, we can throw away any polynomial which is "uniformly worse" than some other polynomial and this is what we want to use thinning for. We are looking for some "smallest" polynomials, but we only have a preorder, not a total order, which means that we may need to keep a set of incomparable candidates (elements $x \not\equiv y$ for which neither $x \prec y$ nor $y \prec x$). We first describe the general setting and move to the specifics of our polynomials later.

We start from a strict preorder $(\prec) : a \to a \to Prop$ (an irreflexive and transitive relation). You can think of *Prop* as $\mathbb{B}$ because we only work with decidable relations and finite sets in this application. As we are looking for minima, we say that *y dominates x* if $y \prec x$.

We lift the order relation to sets in two steps. First, $ys \dot{\prec} x$ means that *ys dominates x*, meaning that some element in *ys* is smaller than *x*. If this holds, there is no need to add *x* to *ys* because we already have at least one better element in *ys*. Then $ys \ddot{\prec} xs$ means that *ys* dominates all of *xs*.

$$(\dot{\prec}) : Set\, a \to a \to Prop$$
$$ys \dot{\prec} x = \exists\, y \in ys.\ y \prec x$$
$$(\ddot{\prec}) : Set\, a \to Set\, a \to Prop$$
$$ys \ddot{\prec} xs = \forall\, x \in xs.\ ys \dot{\prec} x$$

Finally, we combine subset and domination into the thinning relation:

$$Thin\, ys\, xs = (ys \subseteq xs) \wedge ys \ddot{\prec} (xs \setminus\setminus ys)$$

We will use this relation in the specification of our efficient computation to ensure that the small set of polynomials computed, still "dominates" the big set of all the polynomials generated by $genAlg_n f$.

But first we introduce the helper function $thin : Set\, a \to Set\, a$ which aims at removing some elements, while still keeping the minima in the set. Later, we will use the function

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

$genAlgT_n f$ specified similarly to $genAlg_n f$ but using the helper function *thin*. It has to refine the relation *Thin* which means that if $ys = thin\ xs$ then ys must be a subset of xs ($ys \subseteq xs$) and ys must dominate the rest of xs ($ys \ddot{\prec} (xs \setminus\setminus ys)$). A trivial (but useless) implementation would be $thin = id$, and any implementation which removes some "dominated" elements could be helpful. The best we can hope for is that *thin* gives us a set of only incomparable elements. If *thin* compares all pairs of elements, it can compute a smallest thinning. In general that may not be needed (and a linear time greedy approximation is good enough), but in some settings almost any algorithmic cost which can reduce the intermediate sets will pay off. We collect the thinning functions in the type class *Thinnable*:

```
class Ord a ⇒ Thinnable a where
  thin      :: Set a → Set a
  thinStep  :: Set a → a → Set a
  cmp       ::      a → a → Maybe Ordering
  dominatesS :: Set a → a → 𝔹
    -- greedy default definitions inspired by Bird & Gibbons (2020)
  thin = S.foldl thinStep S.∅
  thinStep ys x = if ys ⊀̇ x then ys else S.insert x ys
  ys ⊀̇ x = S.member 1 (map (`check` x) ys)
    where check y x = cmp y x ∈ [Just LT, Just EQ]
```

The greedy *thin* starts from an empty set and considers one element x at a time. If the set ys collected thus far already dominates x, it is returned unchanged, otherwise x is inserted. The optimal version also removes from ys all elements dominated by x. It is easy to prove that *thin* implements the specification *Thin*.

The method *cmp* is a more informative version of $(\prec)$: it returns *Just LT*, *Just EQ*, or *Just GT* if the first element is smaller, equal, or greater than the second, respectively, or *Nothing* if they are incomparable.

Our use of thinning. Now we have what we need to specify when an efficient $genAlgT_n f$ computation is correct. Our specification ($spec\ n\ f$) states a relation between a (very big) set $xs = genAlg_n f$ and a smaller set $ys = genAlgT_n f$, we get by applying thinning at each recursive step. We want to prove that $ys \subseteq xs$ and $ys \ddot{\prec} (xs \setminus\setminus ys)$ because then we know we have kept all the candidates for minimality.

```
spec n f = let xs = genAlg_n  f
               ys = genAlgT_n f
           in (ys ⊆ xs) ∧ (ys ⊀̈ (xs \\ ys))
genAlgT = genAlgStepThin genAlgT
genAlgStepThin genT n f = thin (genAlgStep genT n f)
```

We can first take care of the simplest case (for any n). If the function f is constant (returning some $b : \mathbb{B}$), both xs and ys will be the singleton set containing $res\ b$. Thus, both properties trivially hold.

We then proceed by induction on n to prove $S_n = \forall f : BoolFun\ n.\ spec\ n\ f$. In the base case $n = 0$ the function is necessarily constant, and we have already covered that above. In the inductive step case, assume the induction hypothesis $IH = S_n$ and prove S_{n+1} for a function $f : BoolFun\ (n + 1)$. We have already covered the constant function case, so we

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

focus on the main recursive clause of the definitions of $genAlg_n\,f$ and $genAlgT_n\,f$ when the fixpoint definitions have been expanded:

$$\begin{aligned} genAlg_{n+1}\ f &= \{pic\,i\,x_0\,x_1 \mid i \leftarrow [1\,..\,n], x_0 \leftarrow genAlg_n\ f_{\mathbf{0}}^i, x_1 \leftarrow genAlg_n\ f_{\mathbf{1}}^i\} \\ genAlgT_{n+1}\,f &= thin\,\{pic\,i\,y_0\,y_1 \mid i \leftarrow [1\,..\,n], y_0 \leftarrow genAlgT_n f_{\mathbf{0}}^i, y_1 \leftarrow genAlgT_n f_{\mathbf{1}}^i\} \end{aligned}$$

All subfunctions $f_b^i : BoolFun\ n$ used in the recursive calls satisfy the induction hypothesis: $spec\ n\ f_b^i$. If we name the sets involved in these hypotheses xs_b^i and ys_b^i, we can thus assume $ys_b^i \subseteq xs_b^i$ and $ys_b^i \ddot{\prec} (xs_b^i \setminus\setminus ys_b^i)$.

First, the subset property: we want to prove that $genAlgT_{n+1}\,f \subseteq genAlg_{n+1}\,f$, or equivalently, $\forall\, y.\ (y \in genAlgT_{n+1}\,f) \Rightarrow (y \in genAlg_{n+1}\,f)$. Let $y \in genAlgT_{n+1}\,f$. We know from the specification of *thin* and the definition of $genAlgT_{n+1}\,f$ that $y = pic\,i\,y_0\,y_1$ for some $y_0 \in ys_0^i$ and $y_1 \in ys_1^i$. The subset part of the induction hypothesis gives us that $y_0 \in xs_0^i$ and $y_1 \in xs_1^i$. Thus, we can see from the definition of $genAlg_{n+1}\,f$ that $y \in genAlg_{n+1}\,f$.

Now for the "domination" property we need to show that $\forall\, x \in xs \setminus\setminus ys.\ ys \dot{\prec} x$ where $xs = genAlg_{n+1}\,f$ and $ys = genAlgT_{n+1}\,f$. Let $x \in xs \setminus\setminus ys$. Given the definition of xs it must be of the form $x = pic\,i\,x_0\,x_1$ where $x_0 \in xs_{\mathbf{0}}^i$ and $x_1 \in xs_{\mathbf{1}}^i$. The (second part of the) induction hypothesis provides the existence of $y_b \in ys_b^i$ such that $y_b \prec x_b$. From these y_b we can build $y' = pic\,i\,y_0\,y_1$ as a candidate element to "dominate" xs.

We can now show that $y' \prec x$ by polynomial algebra:

$$\begin{array}{ll} & true \\ \Longrightarrow & \text{-- Follows from the induction hypothesis} \\ & (y_0 \prec x_0) \wedge (y_1 \prec x_1) \\ \Longrightarrow & \text{-- In the interval (0, 1) both } 1 - xP \text{ and } xP \text{ are positive} \\ & 1 + (1 - xP) \times y_0 + xP \times y_1 \ \prec\ 1 + (1 - xP) \times x_0 + xP \times x_1 \\ \Leftrightarrow & \text{-- Def. of } pic \text{ for polynomials} \\ & pic\,i\,y_0\,y_1 \prec pic\,i\,x_0\,x_1 \\ \Leftrightarrow & \text{-- Def. of } y' \text{ and } x \\ & y' \prec x \end{array}$$

We are not quite done, because y' may not be in ys. It is clear from the definition of $genAlgT_{n+1}\,f$ that y' is in the set ys' sent to *thin*, but it may be "thinned away." But, either $y' \in ys = thin\,ys'$ in which case we take the final $y = y'$ or there exists another $y \in ys$ such that $y \prec y'$ and then we get get $y \prec x$ by transitivity.

To sum up, we have now proved that we can push a powerful *thin* step into the recursive enumeration of all cost polynomials in such a way that any minimum is guaranteed to reside in the much smaller set of polynomials thus computed.

The specific properties we need from $(\prec)$ (in addition to the general requirements for thinning) are that $(pos+)$ and $(pos\times)$ are monotonic (for polynomials $0 \prec pos$) and that $q_0 \prec q_1$ implies $evalP\,q_0\,p \leqslant evalP\,q_1\,p$ for all $0 \leqslant p \leqslant 1$.

3.3 Memoization

The call graph of $genAlgT_n\,f$ is the same as the call graph of $genAlg_n\,f$ and, as mentioned above, it can be exponentially big. Thus, even though thinning helps in making the intermediate sets exponentially smaller, we still have one source of exponential computational complexity to tackle. Fortunately, the same subfunctions often appear in many

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

different nodes and this means we can save a significant amount of computation time using memoization.

The classical example of memoization is the Fibonacci function. Naively computing $fib\,(n+2) = fib\,(n+1) + fib\,n$ 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 can compute $fib\,n$ in linear time.

Similarly, here we "just" need to tabulate the result of the calls to $genAlg_n\,f$ so as to avoid recomputation. The challenge is that the input we need to tabulate is now a Boolean function, which is not as nicely structured as a natural number index. Fortunately, thanks to Hinze (2000), Elliott, and others we have generic Trie-based memo functions only a hackage library away[5]. The *MemoTrie* library provides the *Memoizable* class and suitable instances and helper functions for most types. We only need to provide a *Memoizable* instance for *BDD*s, and we do this using *inSig* and *outSig* from the *BDD* package (decision-diagrams). They expose the top-level structure of a *BDD*: $Sig\,bf$ is isomorphic to $Either\,\mathbb{B}\,(Index, bf, bf)$ where $bf = BDDFun$. We define our top-level function *genAlgThinMemo* by applying memoization to $genAlgT_n$ (or, more specifically, to *genAlgStepThin*).

3.4 Comparing polynomials

As argued in Section 3.2, the key to an efficient computation of the best cost polynomials is to compare polynomials as soon as possible and throw away those which are "uniformly worse." The specification of $p \prec q$ is $p\,x \leqslant q\,x$ for all $0 \leqslant x \leqslant 1$ and $p\,x < q\,x$ for some $0 < x < 1$. Note that $(\prec)$ is a strict preorder — if the polynomials cross, neither is "uniformly worse" and we keep both. A simple example of two incomparable polynomials is xP and $1 - xP$ which cross at $p = 1/2$.

If we have two polynomials p and q, we want to know if $p \leqslant q$ for all inputs in the interval $[0, 1]$. Equivalently, we need to check if $0 \leqslant q - p$ in that interval.

```
cmpPoly :: (Ord a, Field a) ⇒ Poly a → Poly a → Maybe Ordering
cmpPoly p q = cmpZero (q − p)
```

As the difference is also a polynomial, we can focus our attention to locating polynomial roots in the unit interval.

If there are no roots (Figure 9a) in the unit interval, the polynomial stays on "one side of zero," and we just need to check the sign of the polynomial at any point. If there is at least one single root (Figure 9b), the original polynomials cross and we return *Nothing*. Similarly for triple roots or roots of any odd order. Finally, if the polynomial only has roots of even order (some double roots, or quadruple roots, etc. as in Figure 9c) the polynomial stays on one side of zero, and we can check a few points to see what side that is. (If the number of distinct roots is r we check up to $r + 1$ points to make sure at least one will be nonzero and thus tell us on which side of zero the polynomial lies.)

To compare polynomials, we thus need to implement the root-counting functions *numRoots* and *numRoots′*:

[5] Available on hackage as the MemoTrie Haskell package.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

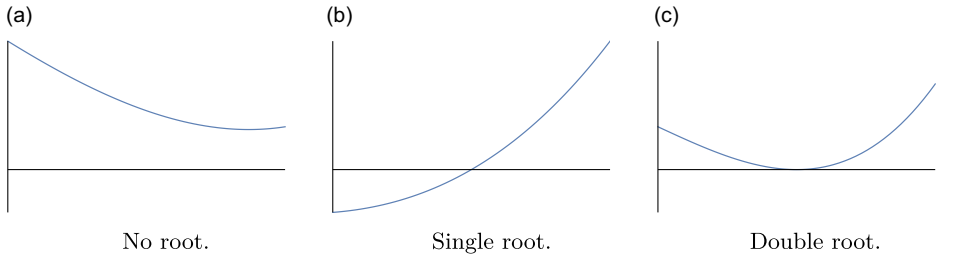

Fig. 9. To compare two polynomials p and q we use root counting for $q - p$ and these are the three main cases to consider.

```
numRoots :: (Ord a, Field a) ⇒ Poly a → Int
numRoots = sum ∘ numRoots′
numRoots′ :: (Ord a, Field a) ⇒ Poly a → [Int]
```

We will not provide all the code here, because that would take us too far from the main topic of the paper, but we will illustrate the main algorithms and concepts for root counting in Section 3.5. The second function computes real root multiplicities: $numRoots'\ p = [1, 3]$ means p has one single and one triple root in the open interval $(0, 1)$. From this we get that p has $2 = length\,[1, 3]$ distinct real roots and $4 = sum\,[1, 3]$ real roots if we count multiplicities.

Using the root-counting functions, the top level of the polynomial partial order implementation is as follows:

```
cmpZero :: (Ord a, Field a) ⇒ Poly a → Maybe Ordering
cmpZero p | isZero p                 = Just EQ
          | all even (numRoots′ p)   = if      any (0<) vals then Just LT
                                       else if any (0>) vals then Just GT
                                       else Just EQ
          | otherwise                = Nothing   -- incomparable
  where r      = length (numRoots′ p)   -- the number of distinct roots
        rp2    = fromIntegral (r + 2)
        points = [i / rp2 | i ← take (r + 1) (iterate (1+) 1)]
        vals   = map (evalP p) points
```

3.5 Isolating real roots and Descartes rule of signs

This section explains how to do root counting by combining Yun's algorithm and Descartes rule of signs. As explained in Section 3.4, the root counting is the key to implementing comparison, which is needed for thinning. First out is Yun's algorithm (Yun, 1976) for square-free factorization: given a polynomial p it computes a list of polynomial factors p_i, each of which only has single roots, and such that $p = C \prod_i p_i{}^i$. Note the exponent i: the factor p_2, for example, appears squared in p. If p only has single roots, the list from Yun's algorithm has just one element, p_1, but in any case we get a finite list of polynomials, each of which is "square-free."[6]

[6] Yun's algorithm is built around repeated computation of the polynomial greatest common divisor of p and its derivative, p'. See the associated code for the details.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

Second in line is Descartes rule of signs that can be used to determine the number of real zeros of a polynomial function. It tells us that the number of positive real zeros in a polynomial function f is the same, or less than by an even number, as the number of changes in the sign of the coefficients. Together with some polynomial transformations, this is used to count the zeros in the interval $[0, 1)$.

If the rule gives zero or one, we are done: we have isolated an interval $[0, 1)$ with either no root or exactly one root. For our use case, we do not need to know the actual root, just if it exists in the interval or not.

If the rule gives more than one, we do not quite know the exact number of roots yet (only an upper bound). In that case, we subdivide the interval into the lower $[0, 1/2)$ and upper $[1/2, 1)$ halves. Fortunately, the polynomial coefficients can be transformed to make the domain the unit interval again so that we can call ourselves recursively. After a finite number of steps, this bisection terminates, and we get a list of disjoint isolating intervals where we know there is exactly one root in each. (The number of steps is on the order of the two-logarithm of the minimum distance between two distinct roots.)

Combining Yun and Descartes, we implement our "root counter," and thus our partial order on polynomials.

4 Results

Using the method from the previous section, we can now calculate the level-p-complexity of Boolean functions with our function *genAlgThinMemo*. First, we return to our example from the beginning (sim_5), where we get several polynomials which are optimal in different intervals. Then, we calculate the level-p-complexity for maj_3^2 which is lower than the proposed result in Jansson (2022), which means that our current method is better.

4.1 Level-p-complexity for sim_5

When we run *genAlgThinMemo* 5 sim_5 it returns a set of four polynomials:

$$\{P_1(p) = 2 + 6p - 10p^2 + 8p^3 - 4p^4, \qquad P_2(p) = 4 - 2p - 3p^2 + 8p^3 - 2p^4,$$
$$P_3(p) = 5 - 8p + 9p^2 - 2p^4, \qquad P_4(p) = 5 - 8p + 8p^2\}$$

We do not compute their intersection points, but we know that they do intersect in the unit interval. The four polynomials were shown already in Figure 1. The level-p-complexity for sim_5 is the piecewise polynomial, pointwise minimum, of these four, with two different polynomials in different intervals: $D_p(sim_5) = P_4(p)$ for $p \in [\approx 0.356, \approx 0.644]$ and $D_p(sim_5) = P_1(p)$ in the rest of the unit interval. As seen in Figure 10, the level-p-complexity has two maxima.

4.2 Level-p-complexity for maj_3^2

When running *genAlgThinMemo* 9 maj_3^2 we get $\{P\,[4, 4, 6, 9, -61, 23, 67, -64, 16]\}$, which means that the expected cost (P_*) of the best decision tree (T_*) is

$$P_*(p) = 4 + 4p + 6p^2 + 9p^3 - 61p^4 + 23p^5 + 67p^6 - 64p^7 + 16p^8\,.$$

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

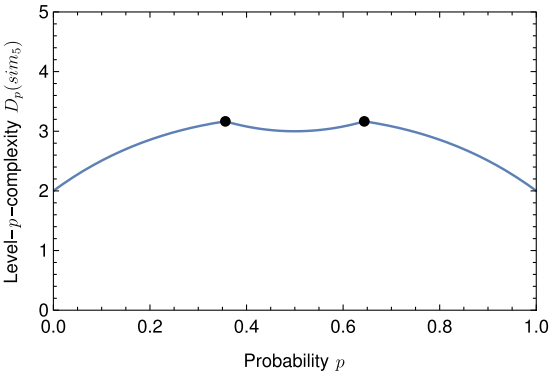

Fig. 10. Level-p-complexity of sim_5, where the dots show the intersections of the costs of the decision trees.

This can be compared to the decision tree (that we call T_t) conjectured in Jansson (2022) to be the best. Its expected cost is slightly higher (thus worse):

$$P_t(p) = 4 + 4p + 7p^2 + 6p^3 - 57p^4 + 20p^5 + 68p^6 - 64p^7 + 16p^8 \,.$$

The expected costs for decision trees T_* and T_t was shown already in Figure 2. Comparing the two polynomials using *cmpPoly* P_* P_t shows that the new one has strictly lower expected cost than the one from the thesis. The difference, illustrated in Figure 3, factors to exactly and we note that it is non-negative in the whole interval.

The value of the polynomials at the endpoints is 4, and the maximum of P_* is ≈ 6.14 compared to the maximum of P_t which is ≈ 6.19. The conjecture in Jansson (2022) is thus false and the correct formula for the level-p-complexity of maj_3^2 is P_*. At the time of publication of Jansson (2022), it was believed that sifting through all the possible decision trees would be intractable. Fortunately, using a combination of thinning, memoization, and exact comparison of polynomials, it is now possible to compute the correct complexity in less than a second on the author's laptop.

5 Conclusions

This paper describes a Haskell library for computing level-p-complexity of Boolean functions and applies it to two-level iterated majority (maj_3^2). The problem specification is straightforward: generate all possible decision trees, compute their expected cost polynomials, and select the best ones. The implementation is more of a challenge because of two sources of exponential computational cost: an exponential growth in the set of decision trees and an exponential growth in the size of the recursive call graph (the collection of subfunctions). The library uses thinning to tackle the first and memoization to handle the second source of inefficiency. In combination with efficient data structures (binary decision diagrams for the Boolean function input, sets of polynomials for the output), this enables computing the level-p-complexity for our target example maj_3^2 in less than a second.

From the mathematics point of view, the strength of the methods used in this paper to compute the level-p-complexity is that we can get a correct result to something which is

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

very hard to calculate by hand. From a computer science point of view, the paper is an instructive example of how a combination of algorithmic and symbolic tools can tame a doubly exponential computational cost.

The library uses type classes for separation of concerns: the actual implementation type for Boolean functions (the input) is abstracted over by the *BoFun* class; and the corresponding type for the output is modeled by the *TreeAlg* class. We also use our own class *Thinnable* for thinning (and preorders) and the *Memoizable* class from hackage. This means that our main function has the following type:

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

All the Haskell code is available on GitHub[7], and parts of it has been reproduced in Agda to check some of the stronger invariants. One direction of future work is to complete the Agda formalization so that we can provide a formally verified library, perhaps helped by Swierstra (2022); van der Rest & Swierstra (2022).

The set of polynomials we compute are all incomparable in the preorder and, together with the thinning relation this means that we actually compute what is called a Pareto front from economics: a set of solutions where no objective can be improved without sacrificing at least one other objective. It would be interesting to explore this in more detail and to see what the overlap is between thinning as an algorithm design method and different concepts of optimality from economics.

The computed level-p-complexity for maj_3^2 is better than the result conjectured in Jansson (2022), and the library allows easy exploration of other Boolean functions. With the current library, the level-p-complexity of iterated majority on 3 levels (27 bits) is out of reach, but with Christian Sattler and Liam Hughes we are exploring a version specialized to "iterated threshold functions" which can handle this case (see code in the GitHub repository).

Acknowledgments

The authors would like to extend their gratitude to Jeffrey Steif for the idea of exploring level-p-complexity and for supervising the preceding work, reported in Jansson (2022). Further, we would like to thank Tim Richter and Jeremy Gibbons for taking their time to give valuable feedback on the first draft of this paper. The authors thank the JFP editors and reviewers, whose helpful and constructive comments have lead to significant improvements of the original manuscript. The work presented in this paper heavily relies on free software, among others on GHC, Agda, Haskell, git, Emacs, LaTeX and on the Ubuntu operating system, Mathematica, and Visual Studio Code. It is our pleasure to thank all developers of these excellent products.

Conflicts of Interest

None.

[7] The paper repository is at https://github.com/juliajansson/BoFunComplexity.

https://doi.org/10.1017/S0956796823000102 Published online by Cambridge University Press

References

Bird, R. & de Moor, O. (1997) *Algebra of Programming*. Prentice-Hall.

Bird, R. & Gibbons, J. (2020) *Algorithm Design with Haskell*. Cambridge University Press.

Bryant, R. E. (1986) Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **C-35**(8), 677–691. https://doi.org/10.1109/TC.1986.1676819.

Buhrman, H. & De Wolf, R. (2002) Complexity measures and decision tree complexity: A survey. *Theor. Comput. Sci.* **288**(1), 21–43.

Garban, C. & Steif, J. E. (2014) *Noise Sensitivity of Boolean Functions and Percolation*. vol. 5. Cambridge University Press.

Hinze, R. (2000) Generalizing generalized tries. *J. Funct. Program.* **10**(4), 327–351.

Jansson, J. (2022) *Level-p-complexity of Boolean Functions*. Master's thesis. Chalmers University of Technology. Available at: https://hdl.handle.net/20.500.12380/304584.

Jansson, P., Ionescu, C. & Bernardy, J.-P. (2022) *Domain-Specific Languages of Mathematics*. vol. 24 of *Texts in Computing*. College Publications.

Knuth, D. E. (2012) *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Pearson Education India.

Landau, I., Nachmias, A., Peres, Y. & Vanniasegaram, S. (2006) The lower bound for evaluating a recursive ternary majority function: an entropy-free proof. *Undergraduate Research Reports, Department of Statistics, University of California, Berkeley*.

Leonardos, N. (2013) An improved lower bound for the randomized decision tree complexity of recursive majority. In *International Colloquium on Automata, Languages, and Programming*. Springer, pp. 696–708.

Magniez, F., Nayak, A., Santha, M., Sherman, J., Tardos, G. & Xiao, D. (2016) Improved bounds for the randomized decision tree complexity of recursive majority. *Random Struct. Algor.* **48**(3), 612–638.

Mu, S., Ko, H. & Jansson, P. (2009) Algebra of programming in Agda: Dependent types for relational program derivation. *J. Funct. Program.* **19**(5), 545–579.

O'Donnell, R. (2014) *Analysis of Boolean functions*. Cambridge University Press.

Swierstra, W. (2022) A well-known representation of monoids and its application to the function 'vector reverse'. *J. Funct. Program.* **32**, e10.

van der Rest, C. & Swierstra, W. (2022) A completely unique account of enumeration. *Proc. ACM Program. Lang.* **6**(ICFP).

Wegener, I. (1987) *The Complexity of Boolean Functions*. John Wiley & Sons.

Yun, D. Y. (1976) On square-free decomposition algorithms. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*. New York, NY, USA: Association for Computing Machinery, pp. 26–35.