Shared Integer Dichotomy

Jean Vuillemin

Shared Integer Dichotomy

Jean Vuillemin[1]

École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France.

Abstract. The *Integer Dichotomy Diagram IDD*(n) represents a natural number $n \in \mathbf{N}$ by a *Directed Acyclic Graph* in which equal nodes are shared to reduce the size $\mathbf{s}(n)$. That *IDD* also represents some finite set of integers by a *Digital Search DAG* where equal subsets are shared. The same *IDD* also represents representing *Boolean Functions*, *IDD*s are equivalent to (Zero-suppressed) ZDD or to (Binary Moment) BMD *Decision Diagrams.* The *IDD* data-structure and algorithms combines three standard software packages into one: arithmetics, sets and Boolean functions.

Unlike the binary length $\mathbf{l}(n)$, the *IDD* size $\mathbf{s}(n) < \mathbf{l}(n)$ is not monotone in n. Most integers are *dense*, and $\mathbf{s}(n)$ is near $\mathbf{l}(n)$. Yet, the *IDD* size of *sparse* integers can be arbitrarily smaller.

We show that a single *IDD* software package combines many features from the best known specialized packages for operating on integers, sets, Boolean functions, and more.

Over *dense* structures, the time/space complexity of *IDD* operations is proportional to that of its specialized competitors. Yet equality testing is performed in unit time with *IDD*s, and the complexity of some integer operations (e.g. $n < m$, $n \pm 2^m$, 2^{2^n}, $\ldots$) is exponentially lower than with bit-arrays.

In general, the *IDD* is *best in class* over sparse structures, where both the space and time complexities can be arbitrarily lower than those of un-shared representations. We show that sparseness is preserved by most integer operations, including arithmetic and logic operations, but excluding multiplication and division.

Keywords: *computer arithmetic, integer dichotomy & trichotomy, sparse & dense structures, dictionary package, digital search tree, minimal acyclic automata, binary Trie, boolean function, decision diagram, store/compute/code once .*

1 Introduction

The *Integer Dichotomy Diagram IDD* is a *Directed Acyclic Graph* DAG introduced in [28] to represent natural numbers and to perform arithmetic operations in novel ways. We extend [28] by showing that the very same *IDD* structure results from *binary Tries* [15] digital search trees, by simply sharing equal sub-trees. For Boolean functions [16], the *IDD* is shown equivalent to the bit-level *Binary Moment Diagram* BMD from [5], and we further clarify its equivalence to the *Zero Suppressed Decision Diagram* ZDD from [19] and [16].

Plan We review the unshared representations of binary integers by arrays and tree in sec. 2. We next present the shared *IDD* data-structure from its three key facets: binary numbers in sec. 3, digital search trees and Boolean functions in sec. 4. All operations on dictionaries (*search, insert, delete*, *merge, intersect, sort, split*, *size, min, max, median*,$\cdots$) and on Boolean functions ($\neg, \cap, \cup, \oplus, \otimes, \cdots$) are reduced to efficient *IDD* integer operations.

Sec. 5 presents the key *IDD* algorithms in more details than [28].

The data-structure and algorithms are analyzed in sec. 6. In general, most structures are **dense**: their size is near the worst case. Nevertheless, we show that the running time of each presented *IDD* algorithm is within a constant factor c of that from the best known specialized package: bit-arrays for integers [1, 10, 14], binary Tries for dictionaries [15], and BMD/ZDD for Boolean functions [5, 19, 16]. Constant c can be large (say 20) for some integer operations, due to the overheads involved in hash-tables, memo functions and storage allocation. Constant c is lower for dictionaries, and we observe that the overhead involved in the key **search** operation is minimal. With proper implementation, no overhead is involved in the manipulation of Boolean functions: same data-structure, same algorithms.

At the same time, many natural structures are **sparse**: their size is smaller than in the worst case; operating over sparse structures can be arbitrarily faster than over dense ones. The success of binary decision diagrams is a key illustration for Boolean functions and circuits [4, 16]. Indeed, we show that all Boolean operations over *IDD*s preserve sparseness. A similar observation applies to dictionaries, and it can be systematically exploited in order to reduce the size of some of the huge tables which are routinely used to index the web[1].

Likewise, many arithmetics operations (add, subtract, multiply by a constant, ...) also preserve sparseness. While in general, integer product and quotient do not preserve sparseness, we nevertheless demonstrate that it is possible to operate efficiently over huge sparse numbers: our reader will encounter the largest integers ever operated upon!

We finally conjecture that the *IDD* is a universal compression method: for a long enough stochastic input source s, the size of *IDD*(s) is proportional to the *entropy*(s), from information theory [24]. Some theoretical evidence in this direction comes from Flajolet [unpublished], and we provide further experimental evidence.

2 Binary Numbers

We first review two representations of natural numbers by bit-arrays and bit-trees.

[1] At this time, we cannot disclose more in this topic.

2.1 Bit-arrays

Arithmetic packages [1, 10] store the binary representation of $n \in \mathbf{N}$ in a finite array of memory words containing all $n_{0\cdots l-1} = {}_2[\mathtt{B}^n_0 \cdots \mathtt{B}^n_{l-1}]$ the significant l bits $n_k = \mathtt{B}^n_k \in \mathbf{B} = \{0, 1\}$ of

$$n = n_{0\cdots l-1} = \sum_{k \in \mathbf{N}} \mathtt{B}^n_k 2^k = \sum_{k<l} n_k 2^k = \sum_{k \in n} 2^k.$$

The *binary length* $l = \mathbf{l}(n)$ of $n \in \mathbf{N}$ is defined here by

$$\mathbf{l}(n) = \lceil \log_2(n+1) \rceil = (n = 0)\ ?\ 0\ :\ 1 + \mathbf{l}(n \div 2).$$

Note that $0 = \mathbf{l}(0)$ and $\mathbf{l}(n) = 1 + \lfloor \log_2(n) \rfloor$ for $n > 0$. For example, integer 818 (3 decimal digits) has $\mathbf{l}(818) = 10$ bits, namely $818 = {}_2 0100110011$. The subscript 2 is explicitly written here to indicate *Little Endian L.E.* (from least to most significant bit), rather than *Big Endian B.E.* (from *MSB* to *LSB*) which we write $818 = 1100110010_2$ as in [16]. Depending on the algorithm's processing order, one endian is better than the other: say *L.E.* for add and subtract, *B.E.* for compare and divide.

Since *bit-arrays* yield the value of any bit in n in constant time (in the RAM model of computations), one can traverse the bits with equal efficiency in both *L.E.* & *B.E.* directions. We assume familiarity with [14] which details the arithmetic operations and their analysis over bit-arrays.

An important *hidden* feature of bit-array packages is their reliance on sophisticated storage management, capable of allocating & de-allocating consecutive memory blocks of arbitrary length. The complexity of such storage allocation [13] is proportional to the size of the block, at least if we amortize over long enough sequences. Yet, the constants involved in allocating/de-allocating bit-arrays are not negligible against those of other linear time $O(\mathbf{l}(n))$ arithmetic operations, such as comparison, add and subtract. Performing the arithmetic operations *in-place* (without changing memory allocation) is a well-known key to the efficient processing of arbitrary precision integers [10]. For example, changing the LSB of n requires a single *in-place* operation ($n := n \oplus 1$), against $O(\mathbf{l}(n))$ if we copy ($m = n \oplus 1$). The functional programming community has long recognized this ”functional array” problem, where ”updating” a single element in a large array becomes exceedingly expensive [9].

2.2 Bit-trees & Dichotomy

Lisp [18] languages allocate computer memory differently: a word per *atom* (0 or 1 on a single bit computer), and a pair of word-pointers for each *cons*. In exchange, storage allocation and garbage collection are fully automatic and (hopefully) efficient.

Representing an integer by its bit-list[2], say $818 = (0\,1\,0\,0\,1\,1\,0\,0\,1\,1)$ is good for *L.E.* operations. It is bad for *B.E.* operations, which must be preceded by a (linear time and memory) *list reversal* [18].

Arithmetics in the LeLisp [2] language uses a *middle endian* approach. Each integer is represented by a (perfectly balanced binary) bit-tree, such as:

$$818 = ((((0.1).(0.0)).((1.1).(0.0))).(((1.1).(0.0)).((0.0).(0.0)))).$$

[2] The list notation $(a\,b)$ stands in Lisp for $(a.(b.()))$

A number $n > 1$ is represented by a tree whose leaves are the bits of $n = n_{0 \cdots 2^p - 1}$, padded with zeroes up to length $2^p \geq \mathbf{l}(n)$. The bit-tree has $2^p < 2\mathbf{l}(n)$ (atomic) leaves, and $2^p - 1$ internal *cons* nodes. The height of the tree is the *binary depth* $p = \mathbf{ll}(n)$ of n

$$\mathbf{ll}(n) = (n < 2) \mathbin{?} 0 \ : \ \mathbf{l}(\mathbf{l}(n) - 1),$$

such that $p = \lceil \log_2 \log_2 (n+1) \rceil$ for $n > 0$. A tree of depth p can represent all integers

$$\mathbf{N}_p = \{ n \ : \ n < \mathbf{x}(p) \}$$

smaller than the *base*: $\mathbf{x}(p) = 2^{2^p}$, where $\mathbf{x}(0) = 2$ and $\mathbf{x}(p+1) = \mathbf{x}(p) \times \mathbf{x}(p)$.

Dichotomy thus represents a number $n \in \mathbf{N}_{p+1}$ of depth $p+1$ by a pair $cons(n0, n1) = n0 \cdot n1$ of integers $n0, n1 \in \mathbf{N}_p$ of depth p. The MSD[3] is the quotient $n1 = n \div \mathbf{x}(q) \in \mathbf{N}_p$ in the division of n by $\mathbf{x}(p)$; the LSD[4] is the remainder $n0 = n \cdot\!|\!\cdot \mathbf{x}(q) \in \mathbf{N}_p$, so that $n = n0 + \mathbf{x}(p) n1$. Both digits $n0$ and $n1$ are padded with non significant zeroes up to length 2^p, and the decomposition continues recursively down to the bit level $\mathbf{N}_0 = \mathbf{B} = \{0, 1\}$, or to that $\mathbf{N}_m$ of the machine word (sec. 5.5) on a computer whose bit-width is $w = 2^m$.

A *dichotomy* package implements arithmetics on bit-trees as in Baker [3]. It accesses any bit (word) of n in time proportional to the depth $p = \mathbf{ll}(n)$ of the tree. It achieves $L.E.$ for *pre-order* [13] tree-traversal, and $B.E.$ for *post-order*.

Dichotomy codes integer operations on 2 digits numbers by sequences of (recursive *divide & conquer*) operations on single digits numbers.

For instance, the *multiply-add-accumulate* operation

$$M(p)(a, b, c, d) = a \times b + c + d = r0 + \mathbf{x}(p) r1 = (r0 \cdot r1) = cons(r0, r1)$$

is defined, for $a, b, c, d \in \mathbf{N}_p$ by the (recursive Lisp like, pseudo Jazz) code:

$$\begin{array}{rll}
M(p)(a,b,c,d) = (r0 \cdot r1) & & \backslash\backslash a \times b + c + d = r0 + \mathbf{x}(p) r1 \\
\{(r0 \cdot r1) = (p = 0) \mathbin{?} (s0 \cdot s1) \ : \ (i0 \cdot i1); & & \\
(s0 \cdot s1) = a \times b + c + d; & & \backslash\backslash \text{ Four 1 bit inputs, result on 2 bits} \\
(a0 \cdot a1) = a; \ (b0 \cdot b1) = b; & & \backslash\backslash \text{ Name input digits from } a, b \\
(c0 \cdot c1) = c; \ (d0 \cdot d1) = d; & & \backslash\backslash \text{ Name input digits from } c, d \\
(e0 \cdot e1) = M(p-1)(a0, b0, c0, d0); & & \backslash\backslash \ a0 \times b0 + c0 + d0 \\
(f0 \cdot f1) = M(p-1)(a0, b1, e1, d1); & & \backslash\backslash \ a0 \times b1 + e1 + d1 \\
(g0 \cdot g1) = M(p-1)(a1, b0, c1, f0); & & \backslash\backslash \ a1 \times b0 + f0 + c1 \\
(h0 \cdot h1) = M(p-1)(a1, b1, f1, g1); & & \backslash\backslash \ a0 \times b1 + c1 + e1 \\
(i0 \cdot i1) = ((e0 \cdot g0) \cdot (h0 \cdot h1))\} & & \backslash\backslash \text{ Name the four result digits}
\end{array} \tag{1}$$

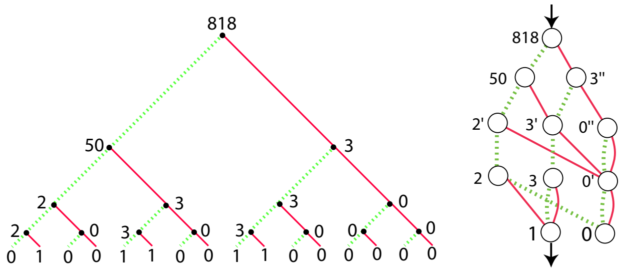
Computing a two-digits product $M(p+1)$ by (1) entails to (recursively) compute 4 single-digit products $M(p)$, plus a few data movements. The time complexity of this (naive dichotomy) product $M(p)$ is quadratic $O(l^2)$ in the operand's length $l = 2^p$. [5]

[3] Most Significant Digit

[4] Least Significant Digit

[5] Never mind here the (cute) dichotomy code for Karatsuba, with just 3 (recursive) single-digit products (see sec. 5.3). Its complexity is sub-quadratic over *dense* numbers and our LeLisp experiments show that the point where it gets faster than the naive product (1) is much lower with bit-trees than with bit-arrays. Yet, the pre/post-processing digit operations dilute sharing, and Karatsuba can be worse than (29) over *sparse shared* numbers.

$818 =$
${}_20100110011$



$818 = 50 \cdot 3''$
$50 = 2' \cdot 3'$ $3'' = 3' \cdot 0''$
$2' = 2 \cdot 0$ $3' = 3 \cdot 0'$ $0'' = 0' \cdot 0'$
$0' = 0 \cdot 0$ $2 = 0 \cdot 1$ $3 = 1 \cdot 1$

Fig. 1. Bit-array, bit-tree, minimal deterministic automaton and cons-chain for decimal integer 818. The integer labels $\{0, 1, 2, 3, 50, 818\}$ at tree and dag nodes are *computed* rather than *stored*. The labels $\{0', 2', 3', 0'', 2'', 3''\}$ name copies which are padded with non-significant zeroes, and they will be eliminated by trichotomy.

In general, bit-trees require (nearly) twice the storage for bit-arrays, in exchange for automatic memory allocation & garbage collection *"à la Lisp"* [2].

2.3 Bit-trees & Continued Fractions CF

One benchmark on *real life* CF [26] for bit-trees [2] against bit-arrays [10] shows that (after word size optimization - sec. 5.5) on a $w = 16 = 2^4$ bits (vintage 1982!) computer, the average time for bit-tree operations is slower than bit-arrays (on the same computer) by a factor $c < 3$. This benchmark is biased towards small numbers by Paul Lévy's theorem [17] on the coefficients of a "random" CF: the probability $p_k = \lim_{n\to\infty} P(x_n = k)$ (famously studied by Kuz'min, Lévy, Khinchin and Wirsing) that the n-th coefficient be $k > 0$ is

$$p_k = -\log_2\left(1 - \frac{1}{(1+k)^2}\right).$$

As 91% of the CF coefficients "fit" within 16 bits, our benchmark only accounts for 9% of the tree/array overhead, beyond word size optimization at $w = 16$ bits. As 74% of the coefficients still fit within 4 bits, the same benchmark showed in 1993 a slow-down beneath $c < 4$, on a $w = 4$ bits computer, namely the symbolic (Lisp based) pocket calculators manufactured by Hewlett Packard at the time.

3 Data Structure

We now combine integer dichotomy with a motto: ***store it once!***

Sharing equal sub-trees in the bit-tree leads to a representation of integers by a binary DAG, which is a *Minimal Deterministic Automaton* MDA.

Sharing equal integers further reduces the structure to a ternary DAG, the *IDD*.

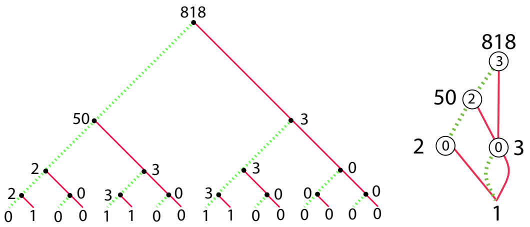
All these structures are shown in fig. 1 & 2.

3.1 Dichotomy & Minimal Automata

The lisp constructor $cons(n, m) = n \cdot m$ is now implemented by the (hash cons) memo-function *hcons*. Both functions construct equivalent Lisp structures

$$\forall n, m \ : \ hcons(n, m) = cons(n, m) = n \cdot m, \tag{2}$$

$$\begin{aligned} 818 &= {}_20100110011 \\ &= 1100110010_2 \end{aligned}$$



$$\begin{aligned} 818 &= \tau(50,3,3) \\ 50 &= \tau(2,2,3) \\ 3 &= \tau(1,0,1) \\ 2 &= \tau(0,0,1) \end{aligned}$$

Fig. 2. Bit-array, bit-tree, *IDD* and minimal triplet-chain for decimal integer 818. The labels $\{0, 1, 2, 3, 50, 818\}$ outside tree and DAG nodes are *computed* rather than *stored*. The labels $\{0, 2, 3\}$ inside the DAG nodes are implicit pointers to the nodes externally labeled by 0, 2 and 3. The decimal labels in the code denote symbolic variables.

yet in computer memory *hcons* shares equal structures while *cons* duplicates. A Lisp expression (such as the above for 818) is represented in computer memory (with *cons*) by a (unshared complete binary) Tree. Using *hcons* yields a (maximally shared binary) DAG - fig. 1.

Elementary automata theory (in the finite language case) reveals that the above DAG for $n \in \mathbf{N}_p$ is isomorphic to the *Minimal Deterministic Automaton* $MDA(n)$. The language $L_n \subseteq \mathbf{B}^p$ accepted by $MDA(n)$ is the set of length $p = \mathbf{ll}(n)$ binary words

$$L_n = \{k_0 \cdots k_{p-1} \in \mathbf{B}^p \ : \ k = \sum_i k_i 2^i, \ 1 = n_k = \mathbf{B}_k^n\}$$

which index all bits $n_k = 1$ at one in $n = \sum_k n_k 2^k$.

For instance $L_{818} = \{0001, 0100, 0101, 1000, 1001\}$ and $MDA(818)$ is shown in fig 1.

One (inefficient) way to build the MDA is to start from the bit-tree, and to systematically share all tree nodes of equal list value, as in fig. 1. The resulting *Minimal Deterministic Automaton* is a binary DAG which represents n by $|MDA(n)|$ nodes in computer memory.

3.2 Trichotomy & *IDD*

An alternative to (2) is to decompose every number $n > 1$ into three integers $n0, p, n1$ such that

$$n = \tau(n0, p, n1) = n0 + \mathbf{x}(p)n1 \text{ and } \mathbf{x}(p) = 2^{2^p}. \tag{3}$$

Decomposition (3) by *trichotomy* is made unique for $n > 1$ by imposing that

$$p = \mathbf{ll}(n) - 1 \text{ and } \max(n0, n1) < \mathbf{x}(p), \tag{4}$$

Note that (4) implies that $n1 > 0$. The *IDD*(n) results from applying (3) recursively, and by sharing all equal integer nodes along the way.

One (inefficient) way to build the *IDD* is to start from the bit-tree, and to systematically share all tree nodes with equal integer value, as in fig. 2. The resulting *Integer Dichotomy Diagram* is a ternary DAG which represents n by $\mathbf{s}(n)$ nodes in computer memory.

If we regard symbol τ as a ternary primitive operator, the most efficient way to construct (say) *IDD*(818) is to start from 0 and 1, and to execute the following symbolic code (fig. 2):

$$2 = \tau(0, 0, 1);\ 3 = \tau(1, 0, 1);\ 50 = \tau(2, 2, 3);\ 818 = \tau(50, 3, 3).$$

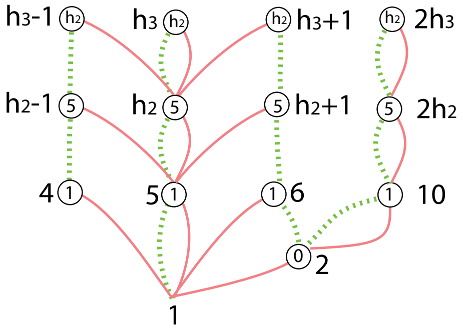


Fig. 3. *These 14 DAG nodes represent the 4 huge numbers $h_3 - 1, h_3, h_3 + 1$ and $2h_3$.*

Bit 0 is stored exactly once, for example at the specific memory address 0. Likewise, bit 1 is stored exactly once (say at the memory address 1). Any other node $n = \tau(n0, p, n1)$ is stored (exactly once) at memory location $n.a$ by a triplet of pointers to the memory addresses of the operands, and we let $n.0$, $n.p$ and $n.1$ denote the access functions to each field.

By construction, the numbers represented at these memory addresses are related by:

$$n = n0 + \mathbf{x}(p)n1,\ p = \mathbf{ll}(n) - 1,\ \ n0 = n \cdot\!|\!\cdot \mathbf{x}(p),\ n1 = n \div \mathbf{x}(p) \text{ where } \mathbf{x}(p) = 2^{2^p}.$$

Based on conditions (3) and(4), we construct a unique triplet $n = \tau(n0, p, n1)$ at some unique memory address $n.a$. This is achieved through a *global* hash table

$$\mathcal{H} = \{(n.h, n.a)\ :\ n \text{ is already present in memory}\}.$$

Table $\mathcal{H}$ stores all pairs of unique hash-codes $n.h = hash(n.0.a, n.p.a, n.1.a)$ and addresses $n.a$ amongst all numbers constructed thus far. If $(h, a) \in \mathcal{H}$, we return the address a of the already constructed result. Else, we allocate a new triple $n = \tau(n0, p, n1)$ at the next available memory address $a = n.a$, we update table $\mathcal{H} := (n.a, n.h) \cup \mathcal{H}$, and we return a. In other words, the constructor $\mathbf{T}$ is a *global memo function* implemented through table $\mathcal{H}$, and defined by:

$$\mathbf{T}(n0, p, n1) = (n1 = 0)\ ?\ n0\ :\ ((h, m) \in \mathcal{H})\ ?\ m\ :\ \tau(n0, p, n1) \tag{5}$$

where $h = hash(n0.a, p.a, n1.a)$ and $\max(n0.p, n1.p) < p$.

We assume that searching & updating table $\mathcal{H}$ is performed in (average amortized) constant time [15]. It follows that constructing node n, and accessing its trichotomy fields $n.0$, $n.p$, $n.1$ are all computed in *constant time* with bit-dags.

A key consequence of the **store once** paradigm is that testing equality between numbers n and m reduces to testing equality between their memory addresses $n.a$ and $m.a$. This is achieved in *one machine instruction*, regardless of the respective sizes of n and m. Note that testing equality with bit arrays or trees requires $\mathbf{l}(n)$ operations in the worst case.

Huge Numbers The virtues of *IDD*s appear over *sparse integers*. To illustrate this fact, consider the largest integer $h_s = \max\{n\ :\ \mathbf{s}(n) = s + 1\}$ which can be represented by an *IDD* with $s + 1$ nodes:

$$h_0 = 1,\ h_{s+1} = h_s \times (1 + 2^{2^{h_s}}) = \mathbf{T}(h_s, h_s, h_s). \tag{6}$$

The letter h stands here for **huge**: $h_1 = 5$; $h_2 = 21474836485$ and $h_3 > 2^{2^{h2}}$ far exceeds the number of earth's atoms - it will **never** be represented by a bit-array! It follows from (6) that $h_n > 2^*(2n)$, where the super-exponential 2^* is defined by: $2^*(0) = 1$ and $2^*(n+1) = 2^{2^*(n)}$. Some vital statistics for h_n:

$$\mathbf{l}(h_n) = \sum_{k<n} 2^{h_k};\ \mathbf{ll}(h_n) = h_{n-1};\ \nu(h_n) = \sum_k \mathrm{B}_k^{h_n} = 2^n.$$

Yet, fig. 3 shows a few small DAGs related to h_3, and we note that

$$\mathbf{s}(1 + h_n) = 2n,\ \mathbf{s}(h_n - 1) = 2n,\ \mathbf{s}(2h_n) = 2n + 1.$$

Our *IDD* package routinely performs (some) arithmetic operations involving h_8 and other related huge sparse integers within milliseconds, and the humongous h_{128} within minutes.

3.3 *IDD* Size

Integer labels in *IDD*$(n_1 \cdots n_k)$ form the least integer set which contains $\{n_1 \cdots n_k\}$, and is closed by trichotomy. This set $\mathcal{S}(n_1 \cdots n_k)$ of labels is defined by:

$$\begin{array}{l} \mathcal{S}(0) = \{\},\ \mathcal{S}(1) = \{1\}, \\ \mathcal{S}(n_1, n_2 \cdots n_k) = \mathcal{S}(n_1) \cup \mathcal{S}(n_2 \cdots n_k), \\ \mathcal{S}(n = \tau(g, p, d)) = \{n\} \cup \mathcal{S}(g, p, d). \end{array}$$

The zero node is present in all *IDD*s. It is convenient to let all references to 0 remain implicit: neither drawn, nor counted. The *size* of n is then the number $\mathbf{s}(n) = |\mathcal{S}(n)|$ of nodes in *IDD*(n), excluding 0. For example (fig. 2) $\mathcal{S}(818) = \{1, 2, 3, 50, 818\}$ and $\mathbf{s}(818) = 5$.

Since $\mathcal{S}(1 \cdots n) = \{1 \cdots n\}$, consecutive integers are optimally coded by *IDD*

$$\mathbf{s}(1 \cdots n) = n. \tag{7}$$

The corresponding size for bit-arrays is $n \times \mathbf{l}(n)$.

Another (near) optimal example is the *IDD* for representing the powers of 2:

$$\mathbf{s}(2^0 \cdots 2^n) < n + \mathbf{l}(n).$$

The corresponding size for bit-arrays is (at least) $n(n+1)/2$.

A third (optimal) case is the *IDD* for the representing the powers of powers of 2:

$$\mathbf{s}(2^{2^0} \cdots 2^{2^n}) \leq 2n.$$

The corresponding size 2^{n+1} for bit-arrays is exponentially larger.

Sparse Sequences So far, we have used the word sparse in an intuitive way, say 2^n is sparse while $n \in \mathbf{N}$ may not be (i.e. n is dense). Of course, this makes no sense for small numbers (is 1 dense or sparse?), or for that matter any fixed integer (is 818 sparse?).

One brings flesh to the concept by considering infinite sequences of numbers.

Definition 1. *A sequence $k \mapsto n_k$ of integers $n_k \in \mathbf{N}$ is* **sparse** *if there exists a (fixed) polynomial P such that*

$$\mathbf{s}(n_k) < P(\mathbf{ll}(n_k))$$

for all $k \in \mathbf{N}$ large enough.

Once said, it makes sense to call **dense** a sequence which is not sparse. It will soon be apparent that both sequences $k \mapsto 2^k$ and $k \mapsto 2^{2^k}$ are sparse, while $k \mapsto k$ is not.

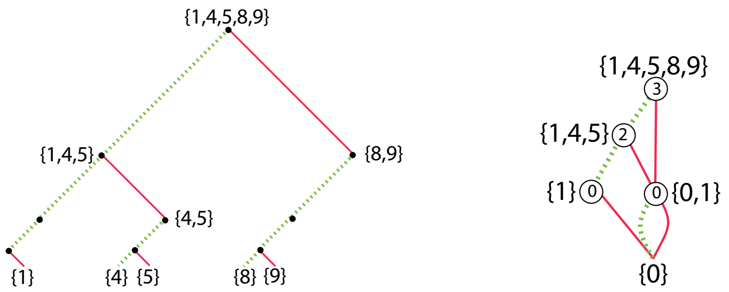


Fig. 4. *The (incomplete) digital search tree to the left is the* binary Trie *[15] for the integer set* $set(818) = \{k : k \in 818\} = \{1,4,5,8,9\}$*; the corresponding* IDD *nodes are labeled here by sets of integers, rather than by integers as in fig. 2.*

4 Coding Sets & Boolean Functions by Integers

We introduce the *IDD* data-structure in two other isomorphic ways, for representing sets of integers and Boolean functions, by relying on the coding power of binary numbers.

4.1 Dictionaries

Finite sets of natural numbers are efficiently represented by *IDD*s through the isomorphism

$$set(n) = \{k \,:\, k \in n\} \rightleftharpoons n = \textstyle\sum_{k \in set(n)} 2^k \tag{8}$$

which transforms set operations ($\cap$, $\cup$, $\oplus$) into logical ones (AND , OR , XOR). We further extend set operations to integers $k, n \in \mathbf{N}$ by letting

$$k \in n \Leftrightarrow 1 = \mathbf{B}_k^n \text{ and } k \subseteq n \Leftrightarrow k = k \cap n.$$

For example, $set(818) = \{1,4,5,7,8,9\}$ and the set h_3 defined by (6) contains 8 elements, namely $set(h3) = \{1, 4, \mathbf{x}(5), 4\mathbf{x}(5), \mathbf{x}(h_2), 4\mathbf{x}(h_2), \mathbf{x}(5)\mathbf{x}(h_2), 4\mathbf{x}(5)\mathbf{x}(h_2)\}$.

Through isomorphism (8), the size of $set(n) = \{k : k \in n\}$ is the *binary weight*[6]:

$$\nu(n) = \sum_{k < \mathbf{l}(n)} \mathbf{B}_k^n = |set(n)|.$$

The binary length $\mathbf{l}(n) = i + 1$ of $n > 0$ is then equal to the largest $i = \max\{k : k \in n\}$ element plus one, and its depth is $\mathbf{ll}(n) = \mathbf{l}(i)$.

Testing for membership $k \in n$ amounts to computing bit k of n. Likewise, operations on *dictionaries* [15] such as *member, insert, delete, min, max, merge, size, intersect, median, range search* all translate by (8) into (efficient) integer operations over *IDD*s.

Fig. 4 illustrates why sharing all equal sub-sets in the representation of $set(n)$ by a binary Trie [15] leads to $IDD(n)$.

The set of integers $set(n)$ which labels the *IDD* node $n = \tau(n0, p, n1)$ is defined by:

$$set(n) = set(n0) \cup \{k + 2^p : k \in set(n1)\}.$$

[6] a.k.a Hamming weight, population count, sideways sum, number of ones in the binary representation.

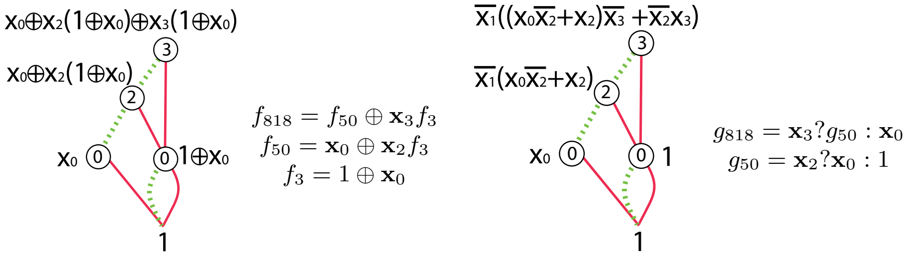

x₀⊕x₂(1⊕x₀)⊕x₃(1⊕x₀)
x₀⊕x₂(1⊕x₀)
3
2
x₀
0
0
1⊕x₀
1
$f_{818} = f_{50} \oplus \mathbf{x}_3 f_3$
$f_{50} = \mathbf{x}_0 \oplus \mathbf{x}_2 f_3$
$f_3 = 1 \oplus \mathbf{x}_0$
$\overline{x}_1((x_0\overline{x}_2+x_2)\overline{x}_3 +\overline{x}_2 x_3)$
$\overline{x}_1(x_0\overline{x}_2+x_2)$
3
2
x₀
0
0
1
1
$g_{818} = \mathbf{x}_3 ? g_{50} : \mathbf{x}_0$
$g_{50} = \mathbf{x}_2 ? \mathbf{x}_0 : 1$

Fig. 5. *The BMD(f) and the ZDD(g) are both isomorphic to* IDD(818) *from fig. 2. These Boolean functions are* $f_{818} = fun_x(818)$ *and* $g_{818} = fun_d(4, 818)$.

Since $\mathbf{s}(n) \leq \nu(n)\mathbf{ll}(n)$, the size of set representations is smaller with *IDD*s than with sorted lists of integers (size $\nu(n)\mathbf{l}(n)$). Because of sharing and regardless of number density, the *IDD* is smaller than all un-shared tree representations of dictionaries described in [15].

4.2 Boolean functions

A Boolean function $f \in \mathbf{B}^* \mapsto \mathbf{B}$ may coded by its truth-table $t = table(f)$, a bit-array which represents a (binary) integer $n = num(t)$. It is finally realized by $IDD(n)$ so as to map the operations {NOT , AND , OR , XOR } over Boolean functions to efficient operations over *IDD*s. We consider two ways to code Boolean functions by integers (fig. 5):

- The **exclusive** number $n_x = num_x(f)$ is a one-to-one map ($f = fun_x(n_x)$) between functions $f \in \mathbf{B}^* \mapsto \mathbf{B}$ and natural numbers $n_x \in \mathbf{N}$. There results a one-to-one map between $BMD(f)$ [5] and $IDD(n_x)$.
- The **disjunctive** number $n_d = num_d(i, f)$ is a one-to-one map ($f = fun_d(i, n_d)$) between functions $f \in \mathbf{B}^i \mapsto \mathbf{B}$ and natural numbers $n_d \in \mathbf{N}_i = \mathbf{N}\,(mod\,\mathbf{x}(i))$. There results a one-to-one map between $ZDD_i(f)$ [19] and $IDD(n_d)$.

n	0	1	2	3	4	5	10	15	17	85	170	255
$bin(n)$	${}_20$	${}_21$	${}_201$	${}_211$	${}_2001$	${}_2101$	${}_20101$	${}_21111$	${}_210001$	${}_21010101$	${}_201010101$	${}_211111111$
$fun_x(n)$	0	1	$\mathbf{x}_0$	$\overline{\mathbf{x}}_0$	$\mathbf{x}_1$	$\overline{\mathbf{x}}_1$	$\mathbf{x}_0\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_1\mathbf{x}_2$	$\mathbf{x}_0\overline{\mathbf{x}}_1\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1\overline{\mathbf{x}}_2$
$fun_d(1,n)$	0	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1$	$\mathbf{x}_0\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_0\mathbf{x}_1$	$\overline{\mathbf{x}}_0$	$\mathbf{x}_0$	1	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_0$	$\mathbf{x}_0$	1
$fun_d(2,n)$	0	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1\overline{\mathbf{x}}_2$	$\mathbf{x}_0\overline{\mathbf{x}}_1\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_1\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_0\mathbf{x}_1\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_2$	$\mathbf{x}_0\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_2$	$\overline{\mathbf{x}}_0\overline{\mathbf{x}}_1$	$\overline{\mathbf{x}}_0$	$\mathbf{x}_0$	1

Fig. 6. The exclusive and disjunctive numbers of a few Boolean functions. We let $\overline{\mathbf{x}}_n = 1 - \mathbf{x}_n$ for $n \in \mathbf{N}$.

For $n < \mathbf{x}(i)$, we note that the Boolean functions $g = fun_d(i, n)$ and $f = fun_x(n)$ are related by the (order i) *Boolean Moebius Transform* (as defined say by [6]): peep in fig. 6.

Exclusive Number Let expression $f(\mathbf{x}_0 \cdots \mathbf{x}_i)$ denote a Boolean function $f \in \mathbf{B}^{i+1} \mapsto \mathbf{B}$ whose $i + 1$ inputs are named by $\mathbf{x}_0 \cdots \mathbf{x}_i$, in order. We note by $f[\mathbf{x}_k = b]$ the result $f(\mathbf{x}_0 \cdots \mathbf{x}_{k-1}, b, \mathbf{x}_{k+1} \cdots \mathbf{x}_i)$ of partially evaluating f at $\mathbf{x}_k = b$, for $b \in \mathbf{B}$. The partial derivative (a.k.a. binary moment [5]) of f with respect to $\mathbf{x}_k$ is

$$\partial_k f = f[\mathbf{x}_k = 0] \oplus f[\mathbf{x}_k = 1].$$

Note that function f depends on variable $\mathbf{x}_k$ if and only if $\partial_k f \neq 0$, i.e. $\mathbf{x}_k$ is *active* in f. We let $i = \mathbf{msv}(f) = \max\{k \,:\, 0 \neq \partial_k f\}$ and $\mathbf{msv}(f) = -\infty$ when f is *constant* ($f = 0$ or $f = 1$). It follows that $\mathbf{x}_i$ is the *most significant active variable* in a non-constant Boolean function f.

The *exclusive number* $n = num_x(f) \in \mathbf{N}$ of a Boolean function $f \in \mathbf{B}^* \mapsto \mathbf{B}$ is defined by:

$$num_x(f) = (i < 0) \,?\, f \;:\; num_x(f[\mathbf{x}_i = 0]) + \mathbf{x}(i) \times num_x(f[\mathbf{x}_i = 1]), \tag{9}$$

where $i = \mathbf{msv}(f)$ and $\mathbf{x}(i) = 2^{2^i}$. Conversely, the Boolean function $f = fun_x(n)$ numbered by n is:

$$fun_x(n) = (i < 0) \,?\, n \;:\; fun_x(n \mathbin{\cdot\!|\!\cdot} \mathbf{x}(i)) \oplus \mathbf{x}_i \cap fun_x(n \div \mathbf{x}(i)), \tag{10}$$

where $i = \mathbf{ll}(n) - 1$ and $\mathbf{x}(i) = 2^{2^i}$. It follows that num_x and fun_x are respective inverses:

$$\forall n \in \mathbf{N} : n = num_x(fun_x(n)) \text{ and } \forall f \in \mathbf{B}^* \mapsto \mathbf{B} : f = fun_x(num_x(f)).$$

Relation (10) yields a direct way to label *IDD* nodes by Boolean functions: node $n = \tau(n0, p, n1)$ represents the function $fun_x(n) = f0 \oplus \mathbf{x}_p f1$, where $f0 = fun_x(n0)$ and $f1 = fun_x(n1)$.

This establishes a direct one-to-one correspondence between Bryant's $BMD(f)$ [5] and *IDD*($num_x(f)$). A very different proof results from exer. 256 & 257 in [16].

Logical operations over Boolean functions map to integer operations as follows:

$$\begin{array}{rr} num_x(0) = 0, & num_x(1) = 1, \\ num_x(\text{NOT } f) = 1 \oplus nf, & num_x(f \text{ XOR } g) = nf \oplus ng, \\ num_x(f \text{ AND } g) = nf \otimes ng, & num_x(f \text{ OR } g) = nf \oplus ng \oplus (nf \otimes ng), \end{array}$$

where $nf = num_x(f)$ and $ng = num_x(g)$. The *multilinear-modular-product* which is defined above and noted by $\otimes$ is further discussed and efficiently implemented with *IDD*s in sec. 5.3.

Disjunctive Number A more classical representation of Boolean functions relies on the *disjunctive truth-table* [16] which we code, for $f \in \mathbf{B}^i \mapsto \mathbf{B}$, by the integer:

$$num_d(i, f) = \sum_{n < 2^i} f(\mathrm{B}_0^n \cdots \mathrm{B}_{i-1}^n) 2^n.$$

Unlike before, the correspondence is only one-to-one between Boolean functions $\mathbf{B}^i \mapsto \mathbf{B}$ and integers $\mathbf{N}_i = \mathbf{N} \pmod{\mathbf{x}(i)}$.

In exchange, Boolean operations map to integer operations by:

$$\begin{array}{rr} num_d(i, 0) = 0, & num_d(i, 1) = \mathbf{x}(i) - 1, \\ num_d(i, \text{NOT } f) = nf \oplus (\mathbf{x}(i) - 1), & num_d(i, f \text{ XOR } g) = nf \oplus ng, \\ num_d(i, f \text{ AND } g) = nf \cap ng, & num_d(i, f \text{ OR } g) = nf \cup ng, \end{array}$$

where $nf = num_d(i, f)$ and $ng = num_d(i, g)$.

We refer to exer. 256 in [16] to prove that *IDD*($num_d(i, f)$) is isomorphic to the *Zero Suppressed Decision Diagram* $ZDD_i(f)$ from [19].

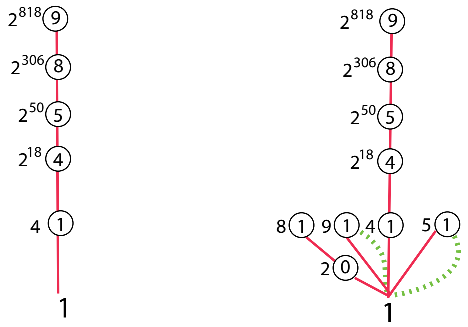


Fig. 7. *The BMD or ZDD for truth-table* 2^{818} *has* $6 = W(2^{818})$ *nodes. So does an* IDD *optimized for word size* $w \geq 4$. *The corresponding bit-level* IDD *has* $10 = \mathbf{s}(2^{818})$ *nodes.*

Depth Coding In the BMD and ZDD above, the depth level is coded by integers, while the *IDD* relies on pointers to DAG nodes - see fig. 7. In theory, this matters when processing very large numbers like h_{16} from (6). In practice (once optimized for word size $w = 64$ as in sec. 5.5), this hardly matters: it currently limits BMDs or ZDDs to handle integers of depth less than 2^{64} which is just enough for representing h_3, but not h_4.

Let $W(n) = |\mathcal{S}'(n)|$ be the number of labels met in traversing the bit-dag for n, ignoring depth labels, i.e. the size of both $BMD(fun_x(n))$ and $ZDD_i(fun_d(i,n))$ for $i = \mathbf{ll}(n)$.

The corresponding set of labels (ignoring depth nodes) is given by:

$$\mathcal{S}'(0) = \{\}, \mathcal{S}'(1) = \{1\},$$
$$\mathcal{S}'(\mathbf{T}(g,p,d)) = \{g + \mathbf{x}(p)d\} \cup \mathcal{S}'(g) \cup \mathcal{S}'(d).$$

Since $\mathcal{S}'(n) \subseteq \mathcal{S}(n) \subseteq \mathcal{S}'(n) \cup \{1 \cdots n.p\}$, it follows from (7) that

$$W(n) \leq \mathbf{s}(n) < W(n) + \mathbf{ll}(n). \tag{11}$$

5 Algorithms

We postpone discussing machine specific word size optimizations to sec. 5.5, and (till then) apply all *IDD* algorithms down to the bit-level. Likewise, negative numbers $\mathbf{Z}$ are introduced in sec. 5.4 and we restrict (till then) to natural numbers $\mathbf{N}$.

We first present some basic *IDD* operations whose running time is intrinsically faster than with bit-arrays.

5.1 Fast Operations

Testing for integer equality in *IDD* reduces to testing equality between memory addresses, in *one machine cycle*, since $n = m \Leftrightarrow n.a = m.a$.

Integer Comparison Comparison $\mathit{cmp}(n,m) = \mathit{sign}(n-m) \in \{-1,0,1\}$ is computed (for $n,m > 1$) over *IDD*s by

$$\begin{array}{rll} \mathit{cmp}(n,m) = & (n=m)\,?\,0\ : & \backslash\backslash\text{terminate when equal} \\ & (n.p \neq m.p)\,?\,\mathit{cmp}(n.p,m.p)\ : & \backslash\backslash\text{unless equal, compare depths} \\ & (n.1 \neq m.1)\,?\,\mathit{cmp}(n.1,m.1)\ : & \backslash\backslash\text{unless equal, compare MSD} \\ & \mathit{cmp}(n.0,m.0) & \backslash\backslash\text{otherwise, compare LSD} \end{array} \tag{12}$$

At most 3 equality tests are performed at each node. Exactly one path is followed down the respective *IDD*s and the computation of $\mathit{cmp}(n,m)$ visits at most $\min(\mathbf{ll}(n),\mathbf{ll}(m))$ nodes. Altogether, this amounts to less than $3\mathbf{ll}(\min(n,m))$ cycles.

In the worst case, this is exponentially faster than with bit-arrays.

Decrement Computing $D(n) = n-1$ for $n > 0$ follows a single DAG path:

$$\begin{array}{rl} D(n) = & (n=1)\,?\,0\ : \\ & (n.0 \neq 0)\,?\,\textsc{t}(D(n.0),n.p,n.1)\ :\ \textsc{t}(\mathbf{x}'(n.p),n.p,D(n.1)). \end{array} \tag{13}$$

Function $\mathbf{x}'(q) = \mathbf{x}(q) - 1 = 2^{2^q} - 1$ is computed in time $O(q)$ by

$$\mathbf{x}'(0) = 1 \text{ and } \mathbf{x}'(q+1) = \textsc{t}(a,q,a)\ \{a = \mathbf{x}'(q)\}.$$

It follows that $\mathbf{s}(\mathbf{x}'(q)) < q$ is small. Implementing $\mathbf{x}'$ as a memo function, and making it *global* (to share its computations with other *IDD* operations) effectively washes out this extra cost.

Either way, we compute $D(n)$ in $O(\mathbf{ll}(n))$ operations.

Increment Likewise, computing $I(n) = n+1$ follows a single DAG path:

$$\begin{array}{rl} I(n) = & (n=0)\,?\,1\ :\ (n=1)\,?\,\textsc{t}(0,0,1)\ : \\ & (n.0 \neq \mathbf{x}'(n.p))\,?\,\textsc{t}(I(n.0),n.p,n.1)\ : \\ & (n.1 \neq \mathbf{x}'(n.p))\,?\,\textsc{t}(0,n.p,I(n.1))\ :\ \textsc{t}(0,I(n.p),1). \end{array} \tag{14}$$

Altogether, computing $I(n)$ or $D(n)$ requires at most $\mathbf{ll}(n)$ operations (along a single DAG path), to which we may (or not by declaring $\mathbf{x}'$ as MF) add $p = \mathbf{ll}(n) - 1$ operations to compute $\mathbf{x}(p) - 1$. In the worst case, this is again exponentially faster than bit-arrays. Moreover, since

$$\mathbf{s}(n,n-1) < \mathbf{s}(n) + \mathbf{ll}(n),$$

the incremental memory cost for representing $n \pm 1$ and n (dense or not) is (at most) $\mathbf{ll}(n)$ for *IDD*s, against (at least) $\mathbf{l}(n)$ for bit-arrays (fig. 3).

Powers of 2 The 2-power of the binary representation of $n = \sum_{k\in n} 2^k$ is

$$2^n = 2^{\sum_{k\in n} 2^k} = \prod_{k\in n} 2^{2^k} = \prod_{k\in n} \mathbf{x}(k). \tag{15}$$

The *IDD* for 2^n is small: since $W(2^n) = \nu(n)$, it follows from (11) that $\mathbf{s}(2^n) < \mathbf{l}(n) + \mathbf{ll}(n)$ for $n > 1$. So, the sequence $n \mapsto 2^n$ is *sparse*: its bit-size $\mathbf{l}(2^n) = n$ is exponential in that of the *IDD* (fig. 7).

We compute $2^n = I_m(0,n)$ by (17) below in time $O(\mathbf{ll}(n))$ and space

$$\mathbf{s}(2^n) < \nu(n) + \mathbf{l}(n) \leq 2\mathbf{l}(n).$$

Both complexity measures are exponentially lower than the corresponding $O(n)$ for bit-arrays.

The 2-powers are the *atoms* from which *IDD*s get built (fig. 4). Indeed, an integer $n > 0$ is a sum of 2-powers $n = \sum_{k\in n} 2^k$. Replacing each 2^k by (15) yields the following sum of products:

$$n = \sum_{k\in n} 2^k = \sum_{k\in n} \prod_{i\in k} \mathbf{x}(i).$$

Replacing each number $\mathbf{x}(i)$ by the symbolic input variable $\mathbf{x}_i$ expresses the Boolean function

$$fun_x(n) = \bigoplus_{k\in n} \prod_{i\in k} \mathbf{x}_i$$

as a multilinear sum of products, a.k.a Reed-Muller form and *Algebraic Normal Form* [6].

Binary length Computing 2^n and the length $l = \mathbf{l}(n)$ of n are mutual *IDD* operations.

1. For $n > 0$, let **remove** MSB[7] be $R_m(n) = (m, i)$, where $i = \mathbf{l}(n) - 1$ and $m = n - 2^i$.
2. For $m < 2^i$, let **insert** MSB be $I_m(m, i) = m + 2^i$.

Both are computed by the mutually recursive *IDD* pair:

$$\begin{aligned} R_m(1) &= (0,0); \\ R_m(\tau(g,p,d)) &= (\mathtt{T}(g,p,e), I_m(l,p)) \\ &\quad \{(e,l) = R_m(d)\}. \end{aligned} \tag{16}$$

$$\begin{aligned} I_m(0,0) = 1; \quad I_m(0,1) &= 2;\; I_m(1,1) = 3; \\ I_m(m,i) &= (l > m.p)\; ?\; \mathtt{T}(m,l,I_m(0,e)) \;: \\ &\qquad \mathtt{T}(m.0,l,I_m(m.1,e)) \\ &\{(e,l) = R_m(i)\}. \end{aligned} \tag{17}$$

The justification for (16) is:
$n = g + \mathbf{x}(p)d = g + \mathbf{x}(p)(e + 2^l) = m + 2^i$, where $m = g + \mathbf{x}(p)e$ and $i = l + 2^p$.

The justification for (17) is: $n = m + 2^i = m + \mathbf{x}(l)2^e$, where $i = e + 2^l$ and $l \geq m.p$ since $m < 2^i$; if $l = m.p$, we finish by $n = m.0 + \mathbf{x}(l)(m.1 + 2^e)$ else by $n = m + \mathbf{x}(l)(0 + 2^e)$.

The analysis of (16,17) shows that, if $n = m + 2^i$ and $m < 2^i$, the time for computing $I_m(m,i)$ and $R_m(n)$ is $O(\mathbf{ll}(n))$: indeed, both (17) and(16) follow a single DAG path with at most $\mathbf{ll}(n)$ nodes, and there are at most four *atomic* (constant time) operations at each node.

Computing $m \pm 2^i$ when $2^i \leq m$ is a variation on the algorithms I, D presented above which follow two DAG pathes rather than one. It follows that $\mathbf{s}(m \pm 2^i) < \mathbf{s}(m) + 2\mathbf{ll}(m)$ in this restricted case. It follows that, for all m, i, the sparseness of m implies the sparseness of both $m \pm 2^i$. Computing $n \pm 2^i$ is (always) exponentially faster with *IDD*s than with bit-arrays.

[7] Most Significant Bit

Dictionaries The 2-powers are the elements of the set $\{k : k \in n\}$ represented by $n \in \mathbf{N}$. Each element corresponds to a path from the root to a leaf labeled by 1, in both the digital search trees and in the corresponding *IDD* (sec. 4.1).

With digital search trees [15], each dictionary operation $\{$*search, insert, delete, min, max*$\}$ follows a single path from root to leaf. This path is determined by a simple decision made at each node: go left or right, and the decision depends on the specific operation. The same applies to *IDD*s, and we leave it as an exercise to code the above operations in time $O(\mathbf{ll}(n))$, i.e. the same complexity as *binary Tries.*

5.2 Arithmetic & Logic Operations

We rely on the above efficient operations to derive the other integer operations over *IDD*s.

Constructor The node constructor $\boldsymbol{\tau}(g,p,d)$ is constrained in (5) by $\max(g,d) < \mathbf{x}(p)$. It is convenient to remove this constraint and to implement a general linear constructor C such that

$$C(g,p,d) = g + \mathbf{x}(p)d$$

for all $g,p,d \in \mathbf{N}$. This is achieved (for $g > 1$) by:

$$\begin{aligned} C(g,p,d) = (p > g.p)\ ?\ & C1(g,p,d)\ : \\ & (p = g.p)\ ?\ C1(g.0,p,A(d,g.1))\ :\ C(C(g.0,p,d),g.p,g.1). \end{aligned} \tag{18}$$

Note that definition (18) is mutually recursive with (the forthcoming) addition A (21), and with the next constructor $C1(g,p,d) = g + \mathbf{x}(p)d$, which is constrained by $g < \mathbf{x}(p)$:

$$\begin{aligned} C1(g,p,d) = (d < 2 \cup p > d.p)\ ?\ & \boldsymbol{\tau}(g,p,d)\ : \\ & (p = d.p)\ ?\ \boldsymbol{\tau}(\boldsymbol{\tau}(g,p,d.0),I(p),d.1)\ : \\ & \qquad C(C1(g,p,d.0),d.p,C1(0,p,d.1)). \end{aligned} \tag{19}$$

Note that $C1$ now relies on increment I. The justification for the mutually recursive codes C (18), $C1$ (19), I (14), $\times 2$ (20) and A (21) is a tedious exercise in linear algebra and inequalities.

Twice As a warm-up, we consider function $2 \times n = n + n = 2n$ which is a special case for add, multiply and shift. One computes twice by the straightforward *IDD* definition

$$\begin{aligned} & 2 \times 0 = 0,\ 2 \times 1 = 2, \\ & 2 \times \tau(g,p,d) = C(2 \times g, p, 2 \times d), \end{aligned} \tag{20}$$

which relies on constructor C to pass "bit carries" from one digit to the next.

Let us then compute $2 \times h_n$ for say h_n as defined by (6). We know from sec. 3 that both input and output remain small: $\mathbf{s}(h_n) = n+1$, $\mathbf{s}(2h_n) = 2n+1$ and $\mathbf{s}(h_n, 2h_n) = 2n+1$.

Yet tracing the computation of $2 \times h_n$ shows that 2×1 is recursively evaluated 2^n times, 2^{n-1} times for 2×5 ... Computing $2 \times h_n$ in this naive way takes exponential time $O(2^n)$!

Operating on shared structures like h_n is hopeless, without a second motto: **compute once!**

We fix the problem by (automatically) turning $2\times$ into a *local* memo function. On the first call, a table $\mathcal{H}_{2\times}$ of computed values is created. Each recursive call $2 \times m$ gets handled by first checking if $2m$ has been computed before i.e. $m \in \mathcal{H}_{2\times}$: if so, we simply return the address of the already computed value; if not, we compute $2 \times m$ and duly record the address of the result in $\mathcal{H}_{2\times}$ for further use. After returning the final result, table $\mathcal{H}_{2\times}$ is released and garbage-collected.

Once implemented as a memo function, the number of operations for computing $2 \times h_n$ becomes linear $O(n)$. The *IDD* package relies extensively on (local and global) memo functions, for many operations. The benefit is to effectively compute over "monster numbers" like h_{128}.

We still have to pass the carries hidden by C in (20). The number of nodes in $\mathcal{S}(n)$ at depth $q < \mathbf{ll}(n)$ can at most *double* in $\mathcal{S}(2 \times n)$: after shifting, and depending on the position, the 0 parity may change to 1 (as shifted out of the previous digit). Finally, at most one node may be promoted to depth $p + 1$, when carry 1 is shifted out from the MSB. For $n > 1$, it follows that

$$\mathbf{s}(2n) < 2\mathbf{s}(n).$$

Note that the above argument applies to any *ALU like* function which takes in a single carry bit, and releases a single carry out. In particular, it follows that $\mathbf{s}(3n) < 2\mathbf{s}(n)$ for $n > 1$. Thus, if n is sparse, so are $2n$ and $3n$ (fig. 3). If $k \mapsto n_k$ is a sparse sequence, this argument can be generalized to show that the product $k \mapsto c \times n_k$ by a fixed constant $c \in \mathbf{N}$ is sparse as well.

Add Trichotomy defines integer addition $A(a, b) = a + b$ recursively by:

$$\begin{aligned} A(a,b) = {} & (a = 0)\ ?\ b\ :\ (a = 1)\ ?\ I(b)\ : \\ & (a = b)\ ?\ 2 \times a\ :\ (a > b)\ ?\ A(b,a)\ : \\ & (a.p < b.p)\ ?\ C(A(a, b.0), b.p, b.1)\ :\ A_m(a,b). \end{aligned} \tag{21}$$

For $1 < a < b$ and $a.p = b.p$, integer addition gets defined by

$$A_m(a,b) = C(A(a.0, b.0), a.p, A(a.1, b.1)). \tag{22}$$

The reason for separating case (22) from (21) is to declare A_m a *local memo function*, rather than A. In this way, table $\mathcal{H}_A$ only stores the results of (recursively computed) additions $A_m(a, b)$ when $a < b$ and both operands have equal depth $a.p = b.p$. The size of table $\mathcal{H}_A$ is thus (strictly) less than $\mathbf{s}(a)\mathbf{s}(b)$. By the argument used above to analyze $2\times$, releasing the carries hidden by C in (21,22) can at most double that size. It follows for $a > b > 1$ that

$$\mathbf{s}(a + b) < 2\mathbf{s}(a)\mathbf{s}(b). \tag{23}$$

So, the sum $k \mapsto n_k + n'_k$ of two *sparse* sequences $k \mapsto n_k$ and $k \mapsto n'_k$ is sparse as well.

Subtract The *one's complement* $n' = n \oplus \mathbf{x}'(p)$ of $n \in \mathbf{N}$, for $p > n.p$ and $\mathbf{x}'(p) = \mathbf{x}(p) - 1$ is reduced to computing one exclusive-or (sec. 5.2). The effect is to negate all bits of n up to $2^p - 1$, so that $n' + n = \mathbf{x}'(p)$ and $\mathbf{s}(n') < \mathbf{s}(n) + p$ follows for $n < \mathbf{x}(p)$.

For $a > b$ we compute the positive difference by the *two's complement* formula

$$a \dot{-} b = (1 + a + b') \cdot 0 \text{ where } b' = Oc(b, a.p). \tag{24}$$

Combining $\mathbf{s}(b') < \mathbf{s}(b) + p$ with (23) yields

$$\mathbf{s}(a \dot{-} b) < 2\mathbf{s}(a)(\mathbf{s}(b) + \mathbf{ll}(a) - 1).$$

So, the difference $k \mapsto n_k - n'_k$ of two *sparse* sequences is a sparse sequence.

Logic Operations The *IDD* code for the logical intersection $And(a, b) = a \cap b$ is:

$$\begin{aligned} And(a, b) = (a = 0)\ ?\ 0\ :\ &(a = 1)\ ?\ 1 \cap b\ : \\ &(a = b)\ ?\ a\ :\ (a > b)\ ?\ And(b, a)\ : \\ &(a.p < b.p)\ ?\ And(a, b.0)\ :\ A_m(a, b) \end{aligned} \tag{25}$$

which relies (when $a.p = b.p$ and $a < b$) on the memo-code

$$And_m(a, b) = \mathrm{T}(And(a.0, b.0), a.p, And(a.1, b.1)). \tag{26}$$

The codes for $\textsc{or}\ (a, b) = a \cup b$ and $\textsc{xor}\ (a, b) = a \oplus b$ have a similar structure. Unlike addition (21), the codes for logical operations rely on constructor T rather than C: no carry!

For $a, b > 1$, it follows that:

$$\max\{\mathbf{s}(a \cup b), \mathbf{s}(a \cap b), \mathbf{s}(a \oplus b)\} < \mathbf{s}(a)\mathbf{s}(b).$$

So, for all Arithmetic & Logic ALU integer operations $op \in \{+, -, \cup, \cap, \oplus\}$, the output sequence $k \mapsto op(n_k, n'_k)$ is *sparse* when both input sequences are sparse.

Dictionaries Most basic dictionary operations have already been treated, except for *size, median* and *sort.* The size of $set(n)$ is reduced to computing $\nu(n)$ by:

$$\begin{gathered} \nu(0) = 0, \quad \nu(1) = 1, \\ \nu(\tau(g, p, d)) = \nu(g) + \nu(d). \end{gathered} \tag{27}$$

Clearly, function ν must (again) be a *memo function.* Computing $\nu(n)$ entails one addition per node over integers smaller than $\mathbf{l}(n)$. The running time for (27) is thus bounded by $O(\mathbf{s}(n)\mathbf{ll}(n))$.

Once the *IDD* weights are computed and stored, it is straightforward to find the median (or the k-th element) in $set(n)$, by following the appropriate DAG pathes in time $O(\mathbf{s}(n)\mathbf{ll}(n))$.

Sorting is a simple pre-order DAG traversal, with a *strong caveat*: the output size (number of elements in the set) may be exponentially bigger than the input size (*IDD* size): it follows from (6) that $\mathbf{s}(h_n) = n + 1$ while $\nu(h_n) = 2^n$.

A way out is use the next "print" routine, whose size is proportional to that of the input.

Print & Read The *triplet-list* of $n \in \mathbf{N}$ is the (topologically sorted) sequence of atomic operations required to construct n from 0 and 1 according to *IDD*(n). Rather than physically implemented in computer memory as before, the triplet-list is now printed as a symbolic sequence of instructions.

Our running example *IDD*(818) can be "symbolically printed" by the triplet-list

$$n = \tau(n1, n3, n3);\ n1 = \tau(n2, n2, n3);\ n3 = \tau(1, 0, 1);\ n2 = \tau(0, 0, 1).$$

in which the symbols represent the integers $n = 818$, $n1 = 50$, $n3 = 3$ and $n2 = 2$. There are $\mathbf{s}(n)$ symbols in the triplet-list and its bit-size (sec. 3.3) is (at most) $\textsc{Bs}(n) < 3\mathbf{s}(n)\mathbf{ll}(n)$.

It is a simple exercise in symbolic manipulation to traverse the DAG and to output the triplet-list. Conversely, on can "read" the triplet-list for n and construct *IDD*(n) in linear time.

5.3 Multiplications

Multilinear Modular Product The multilinear-modular-product is defined in sec. 4.2:

$$a \otimes b = num_x(fun_x(a) \cap fun_x(b)).$$

An alternative definition for $\otimes$ is based on the *Boolean Moebius Transform* [6].

Either way, it simply follows that:

$$\forall a, b \in \mathbf{N} : 0 \otimes b = 0,\ 1 \otimes b = b \text{ and } a \otimes b = b \otimes a,\ a \otimes a = a.$$

If $b = b0 + \mathbf{x}(q)b1$ and $a.p < q$, it follows that:

$$a \otimes (b0 + \mathbf{x}(q)b1) = a \otimes b0 + \mathbf{x}(q)(a \otimes b1).$$

The above relations allow one to reduce the computation of $a \otimes b$ to the case: $1 < a < b$ and $\mathbf{l}(a) = \mathbf{l}(b) = p + 1$. This final case could be handled by:

$$(a0 + \mathbf{x}(p)b1) \otimes (b0 + \mathbf{x}(p)b1) = a0 \otimes b0 + \mathbf{x}(p)(a0 \otimes b1 \oplus a1 \otimes b0 \oplus a1 \otimes b1). \tag{28}$$

Yet, this formula for computing $a \otimes b$ involves 4 recursive calls to $\otimes$, over half-length arguments. A quadratic $O(\mathbf{l}(a)^2)$ complexity would classically result. The distribute law

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c).$$

lets us rewrite (28) in the equivalent form

$$a \otimes b = a0 \otimes b0 + \mathbf{x}(p)(a0 \otimes b1 \oplus a1 \otimes (b0 \oplus b1)).$$

There are only 3 recursive calls, and the time complexity is reduced to $O(\mathbf{l}(a)^{\log_2(3)})$, à la Karatsuba [14]. Our final solution replaces (28) by the equivalent:

$$a \otimes b = p0 + \mathbf{x}(p)(p0 \oplus p1) \text{ where } p0 = a0 \otimes b0 \text{ and } p1 = (a0 \oplus a1) \otimes (b0 \oplus b1).$$

Since the computation of $\otimes$ now involves 2 recursive $\otimes$ (and 3 $\oplus$) over half-length operands, its final complexity is $O(\mathbf{l}(a)\mathbf{ll}(a))$ - an improvement over the original BMD algorithm from [5].

Open question: is the $\otimes$ product of sparse operands sparse, or not?

Integer product The *IDD* product $P(n, m) = n \times m$ is defined by:

$$\begin{aligned} P(a, b) = (a = 0)\ ?\ 0\ :\ (a = 1)\ ?\ b\ : \\ (a > b)\ ?\ P(b, a)\ :\ C(P(a, b.0), b.p, P(a, b.1)). \end{aligned} \tag{29}$$

We declare P as memo code. The size of its hash table $\mathcal{H}_P$ is bounded by the product $\mathbf{s}(n)\mathbf{s}(m)$ of the operand's sizes. In practice, this is sufficient to compute some remarkably large products, such as $818 \times h_{128}$ and $h_{128} \times h_3$.

Yet constructor C in (29) is now hiding *digit carries*, as opposed to bit carries in the previous ALU operations. It follows that, in general, the product of sparse numbers is not sparse. A first example was found by Don Kuth ([16], exer. 256). Another example is provided by the product (shift) $n \times 2^m$ whose *IDD* size can be (nearly) as big as $2^m\mathbf{s}(n)$.

It nevertheless follows from the *safe haven* analysis in sec. 6.3 that the naive code (29) for computing $a \times b$ has a quadratic $O(\mathbf{l}(a) \times \mathbf{l}(b))$ running time: indeed, unwinding once the recursion in (29) yields a code which is equivalent to the 4 digit products of *multiply-add-accumulate* (1).

The algorithms for *big-shift* (product by $\mathbf{x}(p)$) and *shift* (product by 2^i) are left as fun exercises for the reader. Big-shift has an unusual property: the bigger p, the faster! Indeed, the product $d \times \mathbf{x}(p) = \mathbf{T}(0, p, d)$ is computed in unit time as soon as $d < \mathbf{x}(p)$.

Karatsuba Squaring An alternative is to reduce multiply to squaring by:

$$n \times m = ((n+m)^2 - (n-m)^2)/4. \tag{30}$$

We reduce the computation of a 2-digits square to 3 single-digit squares by:

$$\begin{aligned}
&0^2 = 0,\ 1^2 = 1,\\
&\tau(g,p,d)^2 = C(C(s0,p,q),1+p,s1)\\
&\{s0 = g^2;\ s1 = d^2;\ s2 = (g-d)^2;\\
&q = s0 + s1 - s2;\}
\end{aligned} \tag{31}$$

There are advantages to (30,31) over (29): the memo-table has a single argument rather than two, although the benefits from this memo-table are debatable [21]. The worst-case complexity for computing n^2 is $O(\mathbf{l}(n)^{\log_2(3)})$ [14] rather than $O(\mathbf{l}(n)^2)$.

The drawback lies in the overhead from the extra add/sub: 6 are visible and 2 are hidden by C in (30,31). It follows that Karatsuba squaring is only faster than the naive product for n is large enough. A package which uses Karatsuba must first be bench-marked in order to determine the break-even point against the naive method. This technique is classical [14, 21] over *dense* numbers. How to extend it over *sparse* numbers remains a puzzle.

Division Baker [3] performs *dichotomy division* based on the 2-adic inverse a.k.a. Hensel's odd division [22], Montgomery division [20] or GB division [25]. The odd division is the basis for the Stehlé & Zimmermann *dichotomy* algorithm [25] which is the most efficient known for computing the *greatest common divisor* of two integers. We won't further discuss division.

5.4 Negative numbers

We represent a signed integer $z \in \mathbf{Z}$ by the pair $(s = sign(z), n = abs(z))$ formed by its sign (1 if $z < 0$, else 0) and its absolute value represented by *IDD*. The opposite is then defined by

$$-z = (n = 0)\ ?\ (0,0)\ :\ (1-s, n).$$

Logical negation follows by $\neg z = -(1+n)$, and thus $\mathbf{s}(\neg n) < \mathbf{s}(n) + \mathbf{ll}(n)$. We extend all previously defined operations from $\mathbf{N}$ to $\mathbf{Z}$, in obvious ways [28]. For example, subtract is derived from the (above) positive subtract by

$$a - b = (a = b)\ ?\ 0\ :\ (b < a)\ ?\ a \dot{-} b)\ :\ -(b \dot{-} a).$$

5.5 Word size optimization

For the sake of software efficiency, the recursive decomposition (4) is not carried out all the way down to bits, but stopped at the machine-word size, say $w = 32 = \mathbf{x}(5)$ or $w = 64 = \mathbf{x}(6)$ bits. In this manner, primitive machine operations rather than recursive definitions are used on word size operands, at minimal cost. Our package is parameterized by a single constant the word-depth d, for a machine word-size $w = 2^d$. The triplet interpretation which remains invariant throughout the code changes from (3) (for $d = 0$ on a $w = 1$ bit machine) to:

$$\tau(n0, p, n1) = n0 + 2^{2^{p+d}} n1. \tag{32}$$

Knuth [16] discusses the same idea for BDDs in exer. 113, and dismisses it in his answer. Nevertheless, it is clear from our benchmarks that word-size optimization is worthwhile for *IDD*s: rather than being orders of magnitude slower than their competitors for $w = 1$, our benchmarks become less than one order of magnitude slower for $w = 64$ on a standard PC. At the same time, we retain the ability to efficiently process sparse structures.

6 Analysis of *IDD*

6.1 Worst & Average Size

While bit-trees are (on average twice) bigger than bit-arrays, node sharing guaranties that bit-dags are always smaller [28]: $n > 0 \Rightarrow \mathbf{s}(n) < \mathbf{l}(n)$.

For n large enough, more sharing must take place [28]:

$$\mathbf{s}(n) < \frac{2\mathbf{l}(n)}{\mathbf{ll}(n) - \mathbf{l}(\mathbf{ll}(n))}.$$

The worst and average *IDD* sizes are reduced in [28] to those of the BDD or ZDD, as analyzed in [16]. It is follows from [27] that the *worst size* $\mathbf{w}(p) = \max\{\mathbf{s}(k) \ : \ k < \mathbf{x}(p)\}$ is such that

$$\mathbf{w}(p) = c(p)\frac{2^p}{p}$$

where $c(p)$ oscillates between $1 = \liminf_{p\to\infty} c(p)$ and $2 = \limsup_{p\to\infty} c(p)$.

The *average IDD* size

$$\mathbf{a}(p) = \frac{1}{\mathbf{x}(p)} \sum_{k<\mathbf{x}(p)} \mathbf{s}(k)$$

is near the worst. It follows from [27] that $1 = \limsup_{p\to\infty} \mathbf{w}(p)/\mathbf{a}(p)$, and

$$1 - \frac{1}{2e} = \liminf_{p\to\infty} \mathbf{w}(p)/\mathbf{a}(p) \simeq 0.81606\cdots$$

So, a "random" integer n is *dense*: its average size is near the worst size $\mathbf{s}(n) \simeq \mathbf{w}(\mathbf{ll}(n))$, with probability approaching 1 for n large enough.

Note that the worst and average size of *IDD*(n) is near twice Shannon's [23] lower bound on the size of any Boolean circuit which has n for exclusive-number. It follows that we can synthesize from *IDD*(n) a circuit whose size is no worse than twice the minimal one: a small price to pay for such automation.

6.2 Flajolet's Conjecture

Shared DAGs (like Bryant's BDDs [4] and others) are known to compress data and to asymptotically meet Shannon's [24] **entropy** lower bound, under various stochastic data models. Such results are surveyed by Kieffer in [11]. Yet, known proofs of this phenomenon (say [12]) all seem to rely on specific (Huffman or Arithmetic) codes for some finite subset of the source string.

We try to abstract away from specific codes by considering the *bit-size* of $n \in \mathbf{N}$:

$$\textsc{Bs}(n) = \mathbf{s}(n) \times \mathbf{l}(\mathbf{s}(n)). \tag{33}$$

The triplet-list for $n \in \mathbf{N}$ is coded by a sequence of $3 \times \textsc{Bs}(n)$ bits. Shorter codes exist [12], but never mind. It follows from sec. 6.1 that the average bit-size is expressed by

$$\textsc{Bs}(n) = c(\mathbf{ll}(n))\mathbf{l}(n)(1 + o(1)),$$

where c is a bounded oscillating function and bits of n are equaly likely. If the bits of n are drawn by a stochastic process of **entropy** H Shannon/bit [24], we conjecture that:

$$\textsc{Bs}(n) = H \times os(\mathbf{ll}(n))\mathbf{l}(n)(1 + o(1)). \tag{34}$$

Flajolet [unpublished] went a fair way into proving (34) in its simplest probabilistic setting: a Bernoulli memoryless source where 0 has probability p_0, and the entropy is

$$H = -p_0 \log_2(p_0) - p_1 \log_2(p_1).$$

The challenge in Flajolet's [unfinished] proof resides in deriving manageable expressions for the oscillating os and asymptotic $o(1)$ terms in (34), from powerful techniques introduced by [7].

6.3 Time Analysis

Sec. 5 shows that operations on *dictionaries* and *Boolean functions* preserve *sparseness*; likewise for add & subtract, but not for multiply and divide. Over sparse structures, *IDD* algorithms can be arbitrarily faster than their competition from unshared structures.

Dichotomy is a safe haven where to analyze the time of *IDD* algorithms over random dense structures. Indeed, let us turn off all hash-tables in the *IDD* package: we end up performing the very same (bit per bit) operations as bit-trees. It follows that the space & time complexity of trichotomy is bounded, in the worst case (no sharing) by that of dichotomy, within a constant time factor c to account for the cost of searching the hash-tables.

With $w = 32$ word size optimization, our experimental benchmarks over *dense structures* achieves $c < 20$ for arithmetic operations against bit-arrays, and $c < 4$ for dictionary operations against binary Tries. At the same time, they use less memory and remain much faster over (large) *sparse structures.*

7 Conclusion

A number of natural extensions can be made to the *IDD* package, for multi-sets, polynomials, and sets of points in the Euclidean space: quad-trees [3] in 2D and oct-trees in 3D or more dimensions. In each case, the extension is made through some natural integer encoding which maps the operations from the application domain to natural operations over binary integers.

In theory, the running time of the extension should be proportional to that of the best known specialized implementations, and it should use less memory on account of sharing. In practice, each extension should perform faster than the competition over *sparse structures*, such as stellar galaxies or internet indexes.

One common underlying principle is: do it once!

1. **Store once** by systematically sharing equal sub-structures.
2. **Compute once** by systematically using memo-functions.
3. **Code once** by systematically abstracting & sharing common code patterns.

7.1 HUGNUM

The *holly grail* is to design & implement a HUGNUM (Huge Numbers) package which would replace all at once the ancillary packages which most software developers must import as preliminaries to their own code: hash-tables, memory management, dictionaries and arbitrary precision arithmetics BIGNUM.

To achieve this goal, the theory and implementation of HUGNUM must improve to the point where the gains in generality & code sharing (one package replaces many) overcome the time performance loss over dense structures, and keep alive the demonstrated advantages over sparse structures.

In its current Jazz [8] implementation, our *IDD* package relies on external software for both hash tables and memory management. Yet, after word-size optimization, our experimental benchmarks reveal that both are critical issues in the overall performance. It would be nice to incorporate either or both features into HUGNUM. After all, a hash-table is a sparse array with few primitives: is-in, insert, remove-all. The memory available for the application is another sparse resource, where one merely allocate & free HUGNUM nodes.

It would be nice to be able to efficiently distinguish dense sub-structures from sparse ones. One could then implement HUGNUM as a hybrid structure, where dense integers are represented by bit-arrays, operated upon in-place without memo functions, and allocated through some *IDD* indexed *buddy-system* [13]. Sparse integers would be dealt with as before, and one could then hope for the best of both worlds, over dense & sparse structures.

7.2 Acknowledgments

I am indebted to Henry Baker for valuable comments, references and elegant Lisp code [3].

I am grateful to Philippe Flajolet for his kind help, friendship, and for advancing the art of shared structures analysis [7].

I thank Don Knuth for his enlightening criticisms and exercise 256 = $\mathbf{x}(3)$ in [16].

I thank Stéphane Caron for his C++ implementation of *IDD* as a student project at ENS.

This work was supported by grant 12-15-1432-HiCi from the Deanship of Scientific Research at King Abdulaziz University, and by the French Agence Nationale de la Recherche *ANR action Boole*. I acknowledge and thank DSR for technical and financial support.

References

1. The gnu multiple precision arithmetic library: http://gmplib.org/. Free Software Foundation, 2006.
2. J. Chailloux & al. *LE-LISP de l'INRIA : le manuel de reference*, volume 15-2. INRIA, 1986.
3. H. Baker. Lisp integers in "recursive binary" rb "really bignum" format. 2012: http://home.pipeline.com/ hbaker1/rb.lsp.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35:8:677–691, 1986.
5. R. E. Bryant. Symbolic boolean manipulations with ordered binary decision diagrams. *ACM Comp. Surveys*, 24:293–318, 1992.
6. C. Carlet. Boolean functions for cryptography and error correcting codes. *Cambridge University Press*, 16:1–183, 2010.
7. P. Flajolet and R. Sedgewick. *Analytic Combinatorics.* Cambridge University Press, 2009.
8. A. Frey, G. Berry, P. Bertin, F. Bourdoncle, and J. Vuillemin. The jazz home page. 1998: http://www.exalead.com/jazz/index.html.

9. P. Hdak and A. Bloss. The aggregate update problem in functional programming systems. 12'th ACM POPL. ACM, 1985.
10. J-C. Hervé, B. Serpette, and J. Vuillemin. Bignum: a portable efficient package for arbitrary-precision arithmetic. In *PRL report 2, Paris Research Laboratory.* Digital Equipment Corp., 1989.
11. J. C. Kieffer. A survey of advances in hierarchical data compression, 2000.
12. J. C. Kieffer, E. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46:1227–1245, 2000.
13. D. E. Knuth. *The Art of Computer Programming, vol. 1, Fundamental Algorithms.* Addison Wesley, 3-rd edition, 1997.
14. D. E. Knuth. *The Art of Computer Programming, vol. 2, Seminumerical Algorithms.* Addison Wesley, 3-rd edition, 1997.
15. D. E. Knuth. *The Art of Computer Programming, vol. 3, Sorting and Searching.* Addison Wesley, 2-nd edition, 1998.
16. D. E. Knuth. *The Art of Computer Programming, vol. 4A, Combinatorial Algorithms, Part 1.* Addison Wesley, 2009.
17. P. Lévy. Sur les lois de probabilité dont dépendent les quotients complets et incomplets d'une fraction continue. volume 57, pages 178–194. Bull. Soc. Math., 1929.
18. J. McCarthy. The implementation of lisp. In *History of Lisp*, pages 2–12. Stanford University, 1979.
19. S. I. Minato. Zero suppressed decision diagrams. *ACM/IEEE Design Automation Conf.*, 30:272–277, 1993.
20. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
21. E. Nkya. Karatsuba algorithm using sums of. differences. *ITU University, Copenhagen*, pages 1–57, 2007.
22. M. Shand and J. Vuillemin. Fast implementation of RSA cryptography. In *11-th IEEE Symposium on Computer Arithmetic*, 1993.
23. C. E. Shannon. The synthesis of two-terminal switching circuits. In *Bell System Tech. J. 28*, pages 59–98, 1949.
24. C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, 1947.
25. D. Stehlé and P. Zimmermann. A binary recursive gcd algorithm. In *Proceedings of ANTS 04, Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, 2004.
26. J. Vuillemin. Exact real arithmetic with continued fractions. In *1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 14–27. ACM, 1988. Best LISP conference paper award for 1988.
27. J. Vuillemin and F. Béal. On the bdd of a random boolean function. In M.J. Maher, editor, *ASIAN04*, volume 3321 of *Lecture Notes in Computer Science*, pages 483 – 493. Springer-Verlag, 2004.
28. J. E. Vuillemin. Efficient data structure and algorithms for sparse integers, sets and predicates. In *19th IEEE Symposium on Computer Arithmetic*, Proceedings, pages 7–14. IEEE, 2009.