Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates

Jean E. Vuillemin
Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris France

Abstract

We construct a natural number $n > 1$ *by* trichotomy

$$n = g + \mathrm{x}_p d,\ \mathrm{x}_p = 2^{2^p},\ 0 \le g < \mathrm{x}_p,\ 0 < d < \mathrm{x}_p,$$

applied recursively, and by systematically sharing nodes with equal integer value. The resulting Integer Decision Diagram IDD *is a* directed acyclic graph DAG *which represents* n *by* $\mathtt{s}(n)$ *nodes in computer memory.* IDD*s compete with* bit-arrays*, which represent the consecutive bits of* n *within roughly* $\mathtt{l}(n)$ *contiguous bits in memory. Unlike the binary length* $\mathtt{l}(n)$*, the size* $\mathtt{s}(n)$ *is not monotone in* n*. Most integers are* dense*: their size is near worst & average. The* IDD *size of* sparse *integers can be arbitrarily smaller.*

Over dense *numbers, the worst/average time/space complexity of* IDD*s arithmetic operations is proportional to that of bit-arrays. Yet, equality testing is performed in unit time with* IDD*s and the time/space complexity of some operations (e.g.* $sign(n-m)$, $n \pm 2^m$, 2^{2^n}*) are (at least) exponentially better with* IDD*s than with bit-arrays, even over* dense *operands.*

Over sparse *operands, the time and space complexity of all ALU operations* $\{\cap, \cup, \oplus, +, -\}$ *are (in general) arbitrarily better with* IDD*s than with bit-arrays.*

The coding power of integers lets IDD*s implement* integer sets *and* predicates *as well as arithmetics. The* IDD *package is a one-shop alternative to three successful yet rather different packages for processing large numbers, dictionaries and Boolean functions. Performance levels are comparable over dense structures, and* IDD*s are* best in class *over sparse structures.*

Keywords*:* integer dichotomy and trichotomy, sparse numbers, dictionaries, tries, boolean functions, store/compute/code once, binary decision diagrams IDD/BDD/BMD/ZDD.

1 Introduction

To ease the presentation, we postpone discussing machine specific word size optimizations to sec. 3.5, and pretend to operate on a (bare-bone) computer with minimal word size $w = 1$ bit. Also, negative numbers $\mathbf{Z}$ are only introduced in sec. 3.4 and till then, we only discuss natural numbers $\mathbf{N}$.

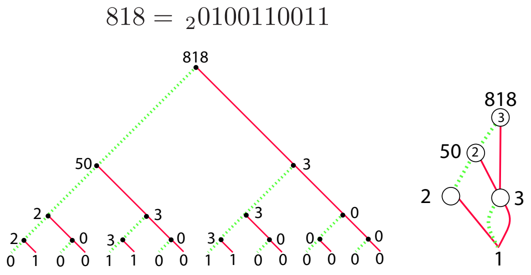

Figure 1. *Bit-array, bit-tree and bit-dag for decimal integer* 818*. The integer labels* $\{0, 1, 2, 3, 50, 818\}$ *at tree and dag nodes are* computed *rather than* stored*. The labels 2 and 3 inside the DAG nodes represent pointers to the nodes externally labeled by 2 and 3.*

The very same data structure can be introduced as *tries* [11] search trees, turned into *search diagrams* by systematically sharing all equal sub-trees, as in fig. 4. So, *IDD*s are dictionaries plus integer arithmetics. They support the basic $\{search, insert, delete\}$, as in [11]; arithmetics enriches the dictionary operations of *IDD*s by $\{merge, intersect, sort, split\}$, $\{size, min, max, median\}$, and other efficient coding.

The very same data structure can be introduced as decision diagrams for representing Boolean functions [12]: the *IDD* is equivalently a *Zero Suppressed Decision Diagram* ZDD [14] or a *Binary Moment Diagram* BMD [3]. In addition to Boolean, the *IDD* supports arithmetic operations.

We choose to present the arithmetic aspects in sec. 3, and extend to dictionaries and Boolean functions in sec. 4.

2 Bit arrays, trees and dags

2.1 Bit-array

Arithmetic packages like [1, 7] store the binary representation of $n \in \mathbf{N}$ as a finite array of consecutive memory words containing all

$$n_{0\cdots l-1} = {}_2[\mathrm{B}_0^n \cdots \mathrm{B}_{l-1}^n]$$

the significant $k < l$ bits $n_k = \mathrm{B}_k^n \in \mathbf{B} = \{0, 1\}$ of

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

The *binary length* $l = \mathbf{l}(n) = \lceil \log_2(n+1) \rceil$ of $n \in \mathbf{N}$ is defined by

$$\mathbf{l}(n) = (n = 0)\,?\,0 \;:\; 1 + \mathbf{l}(n \div 2).$$

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: $818 = {}_2 0100110011$. Index 2 is written here to reminds us of *Little Endian* $L.E.$ (from least to most significant bit), rather than *Big Endian* $B.E.$ (from *MSB* to *LSB*) which we write (as in [12]) $818 = 1100110010_2$. 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 unit time, we can traverse the bits with equal efficiency in both $L.E.$ and $B.E.$ directions.

We assume familiarity with [10] which details the arithmetic operations and their analysis for bit-arrays.

The weight[1] of n is the sum of the ones in its binary representation:

$$\nu(n) = \sum_{k < \mathbf{l}(n)} \mathrm{B}_k^n = |\{k \;:\; k \in n\}|.$$

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 [9] is proportional to the size of the block, at least if we amortize the cost 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 and *in-place* operations. Storage allocation is a well-known key to the efficient processing of very large integers [7].

2.2 Bit-tree

Lisp [13] languages allocate computer memory differently: a word per *atom* (0 and 1 on a single bit computer), and a pair of word-pointers for each *cons*. In exchange, storage allocation and garbage collection is fully automatic and (hopefully) efficient.

[1] a.k.a population count, sideways sum, number of ones in the binary representation [12], and parallel counter.

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* [13].

In LeLisp arithmetics [2], we use a *middle endian* bit-tree compromise. Each integer is represented by a (perfectly balanced binary) bit-tree, like

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

which is also drawn in fig. 1.

Dichotomy represents $n > 1$ by a bit-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 $2^p < 2\mathbf{l}(n)$ leaves, and $2^p - 1$ internal *cons* nodes. The height of the bit-tree is the *binary depth* $p = \mathbf{ll}(n)$ of n

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

such that $p = \lceil \log_2 \log_2(n+1) \rceil$ for $n > 0$. A bit-tree of depth p represents all the integers $\mathbf{N}_p = \{n \;:\; n < \mathrm{x}_p\}$ which are smaller than the *base*:

$$\mathrm{x}_p = 2^{2^p},\ \mathrm{x}_0 = 2,\ \mathrm{x}_{p+1} = \mathrm{x}_p \times \mathrm{x}_p.$$

Dichotomy represents a number $n \in \mathbf{N}_{p+1}$ of depth $p+1$ by a pair (*cons* in Lisp) $(n0 \cdot n1)$ of integers of depth p. The MSD[3] is the quotient $n0 = n \div \mathrm{x}_q \in \mathbf{N}_p$ in the integer division of n by x_p; the LSD[4] is the rest $n0 = n \cdot\!|\!\cdot \mathrm{x}_q \in \mathbf{N}_p$:

$$n = n0 + \mathrm{x}_p n1.$$

Both digits $n0$ and $n1$ are padded with non significant zeroes up to length 2^p. The decomposition continues recursively down to the bits $\mathbf{N}_0 = \mathbf{B} = \{0, 1\}$.

A *dichotomy* package realizes arithmetics on bit-trees. Dichotomy gives access to any bit of n in time proportional to the depth p of the bit-tree. Dichotomy achieves $L.E.$ for the *pre-order* [9] traversal of the bit-tree, and $B.E.$ for *post-order*. Dichotomy efficiently codes each operation on 2 digits numbers, through *divide & conquer*, by a sequence of (recursive) operations on single digits integers. Dichotomy requires nearly twice the storage for bit-arrays, in exchange for an automatic memory allocation scheme *à la Lisp* [2].

An experimental benchmark (recorded on *real life* continued fractions [15]) of bit-trees in [2] vs. bit-arrays in [7] shows that, after word size optimization (see sec. 3.5) on a $w = 16$ bits computer (dating back to 1983), the average time for bit-tree operations is a factor $c < 4$ slower than for bit-arrays - in agreement with this theory.

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

[3] Most Significant Digit

[4] Least Significant Digit

2.3 Bit-dag

*IDD*s combine dichotomy and a motto:

store it once!

A bit-dag represents a bit-tree by sharing a single memory big-dag address for all the nodes which have equal integer value (fig 1). We ensure at creation time (sec. 3) that two nodes with equal integer value n are both represented at the same memory address $n.a$ in the DAG.

An integer is decomposed into a unique triple

$$n = \textsc{t}(n.0, p, n.1) = n.0 + \textsc{x}_p n.1,$$

such that: $p = \mathbf{ll}(n) - 1$ is the depth minus one (hence $\textsc{x}_p \leq n < \textsc{x}_{p+1}$ for $n > 0$); the MSD and LSD digits of n are the quotient $n.1 = n \div \textsc{x}_p$ and rest $n.0 = n \mathbin{\cdot|\cdot} \textsc{x}_p$ in the integer division of n by $\textsc{x}_p$. By construction: $\max(n.0, n.1) < \textsc{x}_p$ and $(n > 1 \Leftrightarrow 0 \neq n.1)$. Conversely, three integers g, p, d result from the trichotomy decomposition of $n = g + \textsc{x}_p d = \textsc{t}(n.0, n.p, n.1)$ (i.e. $g = n.0$, $d = n.1$ and $p = n.p$) if and only if

$$\max(g, d) < \textsc{x}_p \text{ and } d \neq 0. \tag{1}$$

Safe Haven Dichotomy is a safe haven where to analyze *trichotomy*. If we turn off all hash-tables in a trichotomy package, we end up performing the very same (bit for bit) operations with bit-dags and 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 factor to account for the cost of searching hash-tables. Indeed, our experimental implementation of trichotomy in Jazz [6] realizes the above benchmark (after word size optimization) in less than $c < 8$ the time for bit-arrays, and within less than half the memory, since many continued fraction expansions in the benchmark from [15] are sparse.

2.4 *IDD* Size

Integer labels in the bit-dag form the least set which contains n, and is closed by trichotomy. This set $\mathcal{S}(n)$ of labels to DAG nodes is defined by:

$$\begin{aligned}&\mathcal{S}(0) = \mathcal{S}(1) = \{\},\\ &\mathcal{S}(\textsc{t}(g, p, d)) = \{g + \textsc{x}_p d\} \cup \mathcal{S}(g) \cup \mathcal{S}(p) \cup \mathcal{S}(d).\end{aligned}$$

The *size* of n is the number $\mathbf{s}(n) = |\mathcal{S}(n)|$ of nodes in the bit-dag for n, excluding the leaves 0 and 1. For example (fig. 1): $\mathcal{S}(818) = \{2, 3, 50, 818\}$ and $\mathbf{s}(818) = 4$.

The number $W(n) = |\mathcal{S}'(n)|$ of labels met in traversing the bit-dag for n and ignoring depth nodes

$$\begin{aligned}&\mathcal{S}'(0) = \mathcal{S}'(1) = \{\},\\ &\mathcal{S}'(\textsc{t}(g, p, d)) = \{g + \textsc{x}_p d\} \cup \mathcal{S}(g) \cup \mathcal{S}(d).\end{aligned}$$

is the size of the WDD [12] for n. Since $\mathcal{S}'(n) \subseteq \mathcal{S}(n)$ and $\mathcal{S}(n) \subseteq \mathcal{S}(n) \cup \{0, \cdots, n.p\}$, it follows from (2) that

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

While bit-trees are (almost) twice bigger than bit-arrays, sharing nodes guaranties that bit-dags are always smaller:

$$n > 0 \Rightarrow \mathbf{s}(n) < \mathbf{l}(n).$$

The bigger n, the more sharing happens [12, 16]:

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

In the limit, the size is roughly proportional to twice the length divided by the depth.

2.4.1 Dense Numbers

It is shown in [16] that the *worst size*

$$\mathbf{w}(p) = \max\{\mathbf{s}(k) \ : \ k < \textsc{x}_p\}$$

is such that: $1 = \liminf_{p \mapsto \infty} p2^{-p}\mathbf{w}(p)$, and $2 = \limsup_{p \mapsto \infty} p2^{-p}\mathbf{w}(p)$. The *average* size

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

is near the worst, as $1 = \limsup_{p \mapsto \infty} \mathbf{w}(p)/\mathbf{a}(p)$, and

$$1 - \frac{1}{2e} = \liminf_{p \mapsto \infty} \mathbf{w}(p)/\mathbf{a}(p) \simeq {}_{10}0.81606\cdots$$

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

The *bit-size* $\textsc{Bs}(n) = \mathbf{s}(n)(\mathbf{l}(\mathbf{s}(n)) - 1)$ is a measure of the code length for representing the bit-dag by a finite sequence of bits. The analysis in [16] can be sharpened to

$$\textsc{Bs}(n) = \mathbf{s}(n)(\mathbf{l}(\mathbf{s}(n)) - 1) \quad < \quad 2\mathbf{l}(n)$$

for all n. It also follows that $\textsc{Bs}(n) > \mathbf{l}(n)$ for "almost all" integers and we could call *dense* numbers whose *bit-size* is greater than the binary length. Note that $3\textsc{Bs}(n)$ is a naive upper-bound on the total number of bits in all the pointers needed to represent n by trichotomy.

We see from (2) and fig. 2 that consecutive numbers are efficiently coded by bit-dags. In the limit, coding all consecutive numbers up to n by bit-dags is optimal

$$s(1, \cdots, n) \quad = \quad n - 1, \tag{2}$$

against $n\mathbf{l}(n)$ for bit-arrays. Another (near) optimal example is the bit-dag for all 2-powers up to n:

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

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

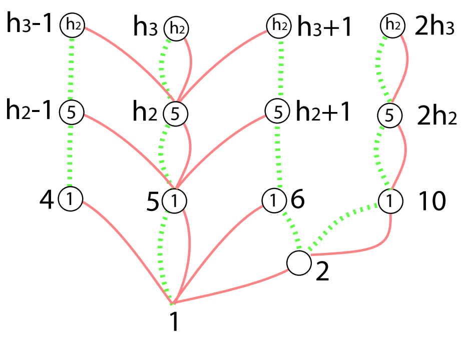

Figure 2. *Huge? These 13 DAG nodes represents* $\{h_3-1, h_3, h_3+1, 2h_3\}$.

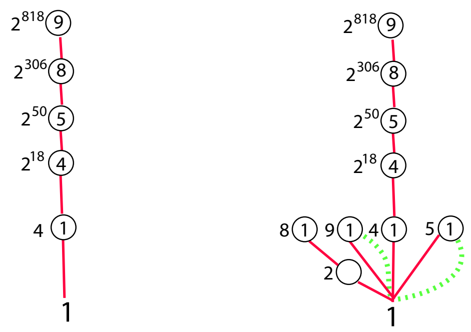

Figure 3. *The 2-power* 2^{818} *has* $5 = W(2^{818})$ *WDD nodes, i.e.* $5 = \nu(818)$ *and* $9 = \mathrm{s}(2^{818})$ IDD *nodes.*

2.4.2 Sparse Numbers

The advantages of *IDD*s appear over *sparse integers*.

Huge To illustrate the concept, consider the largest integer $h_s = \max\{n : \mathrm{s}(n) = s\}$ which can be represented by a bit-dag with s nodes:

$$h_0 = 1,\ h_{s+1} = h_s(1 + 2^{2^{h_s}}) = \mathrm{T}(h_s, h_s, h_s). \quad (3)$$

Letter h stands here for *huge*: $h_1 = 5$, and $h_2 = 21474836485$, $h_2 > 2^{34}$; the binary length of h_3 exceeds the number of earth's atoms, and it will *never* be physically represented by bit-arrays. Physicists will also grant that h_{1024} is bigger than any estimate on the number of physical particles in the known universe, by orders of magnitude. It follows from (3) that $h_n \geq 2^*(2n)$, where the *generalized 2-exponential* is defined by: $2^*(0) = 1$ and $2^*(n+1) = 2^{2^*(n)}$. Some humongous vital statistics ($n > 0$) for h_n:

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

Yet, fig. 2 shows small DAG sizes related to h_3:

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

Powers of 2 The 2-power of the binary representation of $n = \sum_{i<\mathrm{l}(n)} n_i 2^i = \sum_{k\in n} 2^k$ is

$$2^n = 2^{\sum_{k\in n} 2^k} = \prod_{k\in n} 2^{2^k} = \prod_{k\in n} \mathrm{x}_k.$$

It follows that the bit-dag for 2^n is small: $W(2^n) = \nu(n)$ and $\mathrm{s}(2^n) < \mathrm{l}(n) + \mathrm{ll}(n) - 1$ for $n > 1$. So, for *sparse* numbers like 2^n, the bit-size $\mathrm{l}(2^n) = n$ is exponential in the dag-size: see fig. 3. The 2-powers are thus the *atoms* for building *IDD*s: see also fig. 4.

***IDD* Compression** Using a more elaborate coding scheme than above (smart print routine for bit-dags) , Kiefer & al. [8] show that the achieved code size is proportional to the source entropy [5], under some stochastic model of the bits from input n. In other words, representing binary sequences by integer *IDD*s is a form of *universal entropy compression*.

Compute Once Operating on huge numbers like h_{1024} would be hopeless, without a second motto: ***compute it once!***

Consider for example the (cute) computation of $\nu(n)$ by trichotomy:

$$\begin{array}{rcl} \nu(0) = 0, & & \nu(1) = 1, \\ \nu(\mathrm{T}(g, p, d)) & = & \nu(g) + \nu(d). \end{array} \quad (4)$$

Tracing the computation of $\nu(h_n)$ according to (4) shows that $\nu(1)$ is recursively evaluated 2^n times, 2^{n-1} times for $\nu(5)$, 2^{n-2} times for $\nu(h_2)$, and so on. Altogether, computing $\nu(h_n)$ by (4) takes exponential time $O(2^n)$. The problem is fixed by (automatically) turning ν into a *local memo* function. On the first call to $\nu(n)$, a table $\mathcal{H}_\nu$ of recursively computed values is created. Each recursive call $\nu(m)$ is handled by first checking if $\nu(m)$ has been computed before, i.e. $m \in \mathcal{H}_\nu$: if so, we simply return the address of the already computed value; if not, we recursively compute $\nu(m)$ and duly record the address of the result in $\mathcal{H}_\nu$, for further use. Upon returning the final result $\nu(n)$, table $\mathcal{H}_\nu$ is garbage collected. Once (4) is implemented as a local memo function, the number of additions for computing $\nu(h_n)$ becomes linear $O(n)$. The *IDD* package relies extensively on (local and global) memo functions, for most operations. The purpose is to never compute twice the same operation on the same operands. One consequence is that $m + h_{1024}$ and $m \times h_{1024}$ (for any small or sparse m) are both computed efficiently with *IDD*s, and we can perform some genuine computations with "monsters" like h_{1024}.

3 *IDD* Arithmetics

Under condition (1) ($g < \mathtt{x}_p$ and $0 < d < \mathtt{x}_p$), we build a unique triple $n = \mathtt{T}(g,p,d) = g + \mathtt{x}_p d$ at a unique memory address $n.a$. This is achieved through a *global* hash table $\mathcal{H} = \{(n.h, n.a) \ : \ n \text{ in memory}\}$, which stores all pairs of unique hash-code $h = n.h = hash(g.a, p.a, d.a)$ and unique address $a = n.a$, among 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 $n = \mathtt{T}(g,p,d)$ 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 triplet constructor $\mathtt{T}$ is a *global* memo function, implemented with hash table $\mathcal{H}$.

We assume that searching & updating table $\mathcal{H}$ is performed in (average amortized) constant time [11]. It follows that constructing node

$$(1) \Rightarrow n = \mathtt{T}(g,p,d) = g + \mathtt{x}_p d$$

and accessing the trichotomy fields

$$n.0 = g, \ n.p = p = \mathbf{ll}(n) - 1, \ n.1 = d$$

or the node address $n.a$, are all computed in *constant time* with bit-dags.

Integer Comparison Testing for integer equality in a DAG reduces to testing equality between memory addresses, in *one machine cycle*: $n = m \Leftrightarrow n.a = m.a$. Note that equality testing requires at worst $\mathbf{l}(n)$ cycles with bit-arrays/trees/list. Comparison $cmp(n,m) = sign(n - m) \in \{-1, 0, 1\}$ is computed by

$$\begin{array}{rcl} cmp(n,m) & = & (n = m) \,?\, 0 \ : \\ & & (n.p \neq m.p) \,?\, cmp(n.p, m.p) \ : \\ & & (n.1 \neq m.1) \,?\, cmp(n.1, m.1) \ : \\ & & cmp(n.0, m.0). \end{array}$$

At most 3 equality tests are performed at each node, and exactly one path is followed down the respective *IDD*s. The computation of $cmp(n,m)$ visits recursively at most $\min(\mathbf{ll}(n), \mathbf{ll}(m))$ nodes, with up to 3 operations at each node, requires $O(\min(\mathbf{ll}(n), \mathbf{ll}(m)))$ cycles; in the worst case, this is exponentially faster than with bit-arrays.

3.1 Fast Operations

Besides integer comparison, computing $\mathtt{x}_p = 2^{2^p}$ is performed in unit time and size $\mathbf{s}(\mathtt{x}_p) = 1 + \mathbf{s}(p) \leq \mathbf{l}(p)$ with *IDD*s, compared with time and space 2^p with bit-arrays. Similarly, 2^n is computed by $2^n = A_M(0,n)$ (see (7)) in time $O(\mathbf{l}(n)\mathbf{ll}(n))$ and space

$$\mathbf{s}(2^n) \quad < \quad \nu(n) + \mathbf{l}(n). \qquad (5)$$

Both are exponentially smaller than the corresponding $O(n)$ for bit-arrays. We show that decrement $D(n) = n - 1$ and increment $I(n) = n + 1$ are both computed in time $O(\mathbf{ll}(n))$, by descending the DAG along a single path. In the worst case, this is exponentially faster than bit-arrays. Since

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

the incremental cost of representing $n \pm 1$ as well as n (dense or not) is $\mathbf{ll}(n)$ for *IDD*s, against $\mathbf{l}(n)$ for bit-arrays.

To summarize, computing $n \pm 2^i$ is exponentially more efficient with bit-dags than with bit-arrays.

Decrement Computing $D(n) = n - 1$ follows a single path in the DAG:

$$\begin{array}{rcl} D(n) & = & (n = 1) \,?\, 0 \ : \\ & & (n.0 \neq 0) \,?\, \mathtt{T}(D(n.0), n.p, n.1) \ : \\ & & (n.1 = 1) \,?\, x'(n.p) \ : \\ & & \qquad \mathtt{T}(x'(n.p), n.p, D(n.1)). \end{array}$$

Function $x'(q) = \mathtt{x}_q - 1 = 2^{2^q} - 1$ is computed in time $O(q)$ by $x'(0) = 1$ and $x'(q+1) = \mathtt{T}(x'(q), q, x'(q))$. It follows that $\mathbf{s}(\mathtt{x}_q - 1) < 2q$ is small. We implement x' as a memo function, and make it *global* to share its computations with those of other I/D operations. The alternative is to pay the (small) time/space penalty of $q = \mathbf{ll}(n) - 1$ at each increment and decrement operation (without memo).

Increment Computing $I(n) = n + 1$ also follows a single path in the DAG:

$$\begin{array}{rcl} I(n) & = & (n = 0) \,?\, 1 \ : \ (n = 1) \,?\, \mathtt{T}(0,0,1) \ : \\ & & (n.0 \neq x'(n.p)) \,?\, \mathtt{T}(I(n.0), n.p, n.1) \ : \\ & & (n.1 \neq x'(n.p)) \,?\, \mathtt{T}(0, n.p, I(n.1)) \ : \\ & & \qquad\qquad \mathtt{T}(0, I(n.p), 1). \end{array}$$

Altogether, computing $I(n)$ or $D(n)$ requires $\mathbf{ll}(n)$ operations to follow the single DAG path, to which we may (or not) have to add $p = \mathbf{ll}(n) - 1$ operations to compute $\mathtt{x}_p - 1$.

Add/remove MSB Computing the length $\mathbf{l}(n) = l$ of n and its 2-power 2^n are mutually recursive trichotomy operations. They are generalized by the operations of removing $R_M(n) = (m, i)$ the MSB ($n > 0, n = m + 2^i, i = \mathbf{l}(n) - 1$, $m = n - 2^i$), and adding the MSB $A_M(m,i) = m + 2^i$ (for $m < 2^i$) which are defined by the recursive pair:

$$\begin{array}{rcl} R_M(1) & = & (0,0); \\ R_M(\mathtt{T}(g,p,d)) & = & (\mathtt{T}(g,p,e), A_M(l,p)) \\ & & \{(e,l) = R_M(d)\}. \end{array} \qquad (6)$$

$$\begin{array}{rcl} A_M(m,0) & = & m + 1; \ A_M(m,1) = m + 2; \\ A_M(m,i) & = & (l > m.p) \,?\, \mathtt{T}(m, l, A_M(0,e)) \ : \\ & & \qquad \mathtt{T}(m.0, l, A_M(m.1, e)) \\ & & \{(e,l) = R_M(i)\}. \end{array} \qquad (7)$$

The justification for (6) is: $n = g + \mathtt{x}_p d = g + \mathtt{x}_p(e + 2^l) = m + 2^i$, where $m = g + \mathtt{x}_p e$ and $i = l + 2^p$. The justification for (7) is: $n = m + 2^i = m + \mathtt{x}_l 2^e$, where $i = e + 2^l$ and $l \geq m.p$ since $m < 2^i$; if $l = m.p$, we finish by $n = m.0 + \mathtt{x}_l(m.1 + 2^e)$, else by $n = m + \mathtt{x}_l(0 + 2^e)$.

An analysis of (6,7) shows that, for $n = m + 2^i$, $m < 2^i$, the computing time for $A_M(m, i)$ and $R_M(n)$ is $O(\mathbf{l}(n)\mathbf{ll}(n))$: indeed, (7,6) collectively follow a single DAG path with at most $\mathbf{ll}(n)$ nodes; the operation at each node happens on numbers of length at most $\mathbf{l}(n)$, and its complexity is thus at most $O(\mathbf{l}(n))$, by the above *safe haven* argument.

Note that computing $m \pm 2^i$ for $2^i \leq m$ is a simple (dual pathes) variation on I, D and $\mathbf{s}(m \pm 2^i) < \mathbf{s}(m) + 2\mathbf{ll}(m)$ in this case. Thus in general, the sparseness of m implies the sparseness of both $m \pm 2^i$.

3.2 ALU operations

Constructor Replace pre-condition (1) by the weaker

$$\max(g, d) < \mathtt{x}_{p+1}, \tag{8}$$

and define constructor $C(g, p, d) = g + \mathtt{x}_p d$ by:

$$\begin{array}{rcl} C(g,p,d) & = & (d = 0)\,?\;\; g \;: \\ & & (p = g.p)\,?\,C(g.0, p, A(d, g.1)) \;: \\ & & (p = d.p)\,?\,\mathtt{T}(C(g, p, d.0), I(p), d.1) \;: \\ & & \qquad \mathtt{T}(g, p, d). \end{array} \tag{9}$$

Note that C uses incrementation I, and its definition is mutually recursive with that of addition A.

Twice & Thrice As a warm-up, consider the function $2 \times n = n + n = 2n$ which is a special case for add and multiply. Trichotomy recursively computes twice by:

$$\begin{array}{l} 2 \times 0 = 0,\; 2 \times 1 = 2, \\ 2 \times \mathtt{T}(g, p, d) = C(2 \times g, p, 2 \times d). \end{array}$$

It relies on constructor C (9) to pass "carries" from one digit to the next. Obviously, twice must be declared as a local memo function (with hash table $\mathcal{H}_{2\times}$), to stand a chance of computing, say $2 \times h_n$ (see. fig. 2).

The number of nodes in $\mathcal{S}(n)$ at depth $q < \mathbf{ll}(n)$ can at worst *double* in $\mathcal{S}(2 \times n)$: after shifting, and depending on the position, the 0 parity may change to a 1 (shifted out of the previous digit). Finally, a (single) node may be promoted to depth $n.p + 1$, when a 1 is shifted out from the MSB at the bottom level. It follows that

$$\mathbf{s}(2a) \leq 2\mathbf{s}(a).$$

Note that the above argument applies just as well to any *ALU like* function which takes in a single carry bit, and releases a single carry out. In particular, it follows that $\mathbf{s}(3n) \leq 2\mathbf{s}(n)$. Thus, if n is sparse, so are $2n$ and $3n$ (fig. 2).

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

$$\begin{array}{rcl} A(a,b) & = & (a = 0)\,?\,0 \;:\; (a = 1)\,?\,I(a) \;: \\ & & (a = b)\,?\,2 \times a \;:\; (a > b)\,?\,A(b, a) \;: \\ & & (a.p > b.p)\,?\,C(A(a, b.0), b.p, b.1) \;: \\ & & A_m(a, b), \end{array} \tag{10}$$

and, for $1 < a < b$ and $a.p = b.p$, by

$$A_m(a, b) = C(A(a.0, b.0), a.p, A(a.0, b.0)). \tag{11}$$

The reason for separating the case $a.p = b.p$ (11) from the others (10) is to declare A_m as a (local) memo function, but not A. In this way, table $\mathcal{H}_A$ only stores the results of the additions $A(a, b)$ which are recursively computed, when $a < b$ and both operands have the same depth $a.p = b.p$. The size of table $\mathcal{H}_A$ is (strictly) less than $\mathbf{s}(a)\mathbf{s}(b)$. By the argument already used to analyze $2\times$, releasing the carries hidden in C by (11) can at most double that size, and

$$\mathbf{s}(a + b) < 2\mathbf{s}(a)\mathbf{s}(b). \tag{12}$$

Hence, the sum two sparse enough numbers is sparse.

Subtract For $a > b > 1$ and $p = a.p = \mathbf{ll}(a) - 1$, we compute the difference by $a - b = 1 + a + (\mathtt{x}_p - b - 1) - \mathtt{x}_p = d.0$, where $d = I(A(a, b'))$ and $b' = \mathtt{x}_p - b - 1$ is b's *two's complement*. An easy exercise in trichotomy shows that $\mathbf{s}(b') < \mathbf{s}(b) + p$. Combining with (12) yields

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

and the difference of two sparse enough numbers is sparse.

Logic Operations We refer to [12] and the correspondance with ZDDs to perform logical operations on bit-dags, and it follows that:

$$\max\{\mathbf{s}(a \cup b), \mathit{size}(a \cap b), \mathit{size}(a \oplus b)\} < \mathbf{s}(a)\mathbf{s}(b). \tag{13}$$

So, for all ALU operations $\{+, -, \cup, \cap, \oplus\}$, the output is sparse when both inputs are sparse enough.

3.3 Multiplication

Integer multiplication and division have nice trichotomy definitions. We won't discuss division for lack of space. The trichotomy product $P(n, m) = n \times m$ is defined by:

$$\begin{array}{rcl} P(a,b) & = & (a = 0)\,?\,0 \;:\; (a = 1)\,?\,b \;: \\ & & (a > b)\,?\,P(b, a) \;: \\ & & C(P(a, b.0), b.p, P(a, b.1)). \end{array} \tag{14}$$

We declare P to be a memo-function. 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 $808 \times h_{1024}$ and $h_{1024} \times h_{1024}$.

Yet constructor C in (14) 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 ([12], exer. 256). Another example is provided by the product (shift) $n \times 2^m$ whose size can be as big as $2^m\mathbf{s}(n)$.

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

3.4 Negative numbers

We represent a relative number $z \in \mathbf{Z}$ by the pair $(s = sign(z), n = abs(z))$ of its sign (1 if $z < 0$, else 0) and its absolute value represented by *IDD*. We define the opposite by $-z = (n = 0)\,?\,(0, 0)\;:\;(1 - s, n)$, so that $\mathbf{s}(-n) = \mathbf{s}(n)$. The logical negation follows by $\neg z = -(1 + n)$, and thus $\mathbf{s}(\neg n) = \mathbf{s}(1 + n) < \mathbf{s}(n) + \mathbf{ll}(n)$.

Subtract is defined by

$$a - b = (a = b)\,?\,0\;:\;(b < a)\,?\,A(a, -b)\;:\;-A(b, -a).$$

Finally, we extend all previously defined operations from $\mathbf{N}$ to $\mathbf{Z}$, in the obvious way.

3.5 Word size optimization

For the sake of software efficiency, the recursive decomposition (1) is not carried out all the way down to bits, but stopped at the machine-word size, say $32 = \mathrm{x}_5$ or $64 = \mathrm{x}_6$ bits. In this manner, primitive machine operations rather than recursive definitions are used on word size operands, at minimal cost.

4 Sets & Boolean Functions

A theoretical advantage of *IDD*s is combine, in a single package, all operations from (at least three) currently distinct packages: dictionaries [11], Boolean functions [12] and integers [10], within state-of-the-art performance.

4.1 Dictionaries

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

$$\begin{array}{lcl} \{n_1, \cdots, n_s\} & \rightleftharpoons & 2^{n_1} + \cdots + 2^{n_s} \\ \{k\;:\;k \in n\} & \rightleftharpoons & n = \sum_{k \in n} 2^k \end{array} \qquad (15)$$

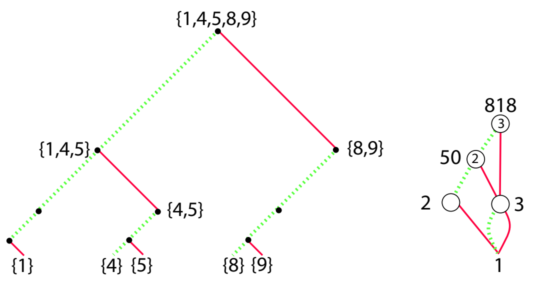

Figure 4. *The incomplete binary decision tree to the left is the* trie *[11] for set* $\{k \in 818\} = \{1, 4, 5, 8, 9\}$*; the corresponding* IDD *is from fig. 1. This very* IDD *also represents the Boolean function* $f \in \mathbf{B}^4 \mapsto \mathbf{B}$ *defined by* $f = \mathrm{x}_0 \oplus (\mathrm{x}_2 \oplus \mathrm{x}_3)g$ *and* $g = 1 \oplus \mathrm{x}_0 = \neg \mathrm{x}_0$.

which transforms set operations ($\cap$, $\cup$, $\oplus$) into logical ones (&, |, $\wedge$). We similarly extend other set operations to integers $k, n \in \mathbf{N}$: $k \in n \Leftrightarrow 1 = \mathrm{B}_k^n$ and $k \subseteq n \Leftrightarrow k = k \cap n$. For example, $\{k : k \in 818\} = \{1, 4, 5, 7, 8, 9\}$and the set $S_3 = \{k : k \in h_3\}$ defined by (3) has $8 = 2^3$ elements: $S_3 = \{1, 4, \mathrm{x}_5, 4\mathrm{x}_5, \mathrm{x}_{h_2}, 4\mathrm{x}_{h_2}, \mathrm{x}_5\mathrm{x}_{h_2}, 4\mathrm{x}_5\mathrm{x}_{h_2}\}$.

The size of set $\{k\;:\;k \in n\}$ is the *binary weight*$\nu(n)$ of n.

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

The *IDD* package implements *dictionaries* [11], with extensive operation support: member, insert, delete, min, max, merge, size, intersect, median, range search, which all translate by (15) to efficient trichotomy operations.

The *IDD* dictionary time complexity is within a constant factor of the best-in-class specialized data-structures, such as tries and Patricia trees [11]. It can be arbitrarily less for *sparse dictionaries* which map to sparse integers by (15). An example which *IDD*s handle like no other structure, is the set $\{k\;:\;k \in h_n\}$ of one bits in (say) h_{1024} (3).

In general, $\mathbf{s}(n) \leq \nu(n)\mathbf{ll}(n)$ and 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 bit-dag is also smaller than all unshared tree representations of dictionaries such as [11].

4.2 Predicates

Boolean function are just as efficiently represented by *IDD*s through the natural (truth-table [12]) isomorphism

$$f \in \mathbf{B}^i \mapsto \mathbf{B} \rightleftharpoons \tau(f) = \sum_{n < 2^i} f(\mathrm{B}_0^n, \cdots, \mathrm{B}_{i-1}^n)2^n.$$

which also transforms set operations ($\cap$, $\cup$, $\oplus$) into logical ones (&, |, $\wedge$). Is is observed in [12] (exer. 256) that the *IDD* for the integer truth-table $\tau(f)$ is isomorphic to the *zero suppressed decision diagrams* ZDD [14]. With a theoretical difference: depth pointers are coded by integers in ZDDs, and by pointers to DAG nodes in *IDD*s. In practice (once optimized for word size as in sec. 3.5), this hardly matters; it limits ZDDs to handle integers of depth less than 2^{64} (barely enough for h_3, but h_4 won't do). Beyond ZDD, other explicit one-to-one correspondences relate the complexity of Boolean operations on *IDD*s to those on *Binary Moment Diagrams* (BMD [4], [12] exer. 257) and *Binary Decision Diagrams* BBD [3, 12]. So, in theory, the proposed integer *IDD* package can manipulate boolean functions just as efficiently as the best known packages designed for that single purpose.

5 Conclusion

With word size optimization, the dichotomy package becomes competitive, and one arithmetic benchmark shows that its performance is less than an order of magnitude slower than bit-arrays over *dense* numbers (within less memory), while it keeps all its advantages over *sparse* numbers. It seems worth-while investing time & efforts into improving the theory & implementation of integer *IDD* software packages. The goal is to reach the point where the gains in generality & code sharing (one package replaces many) overcome the time performance loss over dense structures, and to keep alive most of the demonstrated advantages over sparse structures.

In its current implementation, our *IDD* package relies on external software, for hash tables and memory management. Yet, once the word-size optimization is made, our experimental benchmarks show that both are critical issues in the overall performance of the package. It would be interesting, both in theory & practice, to somehow incorporate either or both features into some extended *IDD* package. After all, a hash-table is a sparse array with few primitives: is-in, insert, release-all. The memory available for the application is another, into which we merely allocate & free *IDD* nodes.

Another closely related question is the following. It would be interesting, both in theory & practice, to efficiently mark & distinguish dense sub-structures from sparse ones. One could then implement a hybrid structure, where dense integers are represented by their bit-arrays, operated upon without memo functions (harmful there), and allocated through some *IDD* indexed *buddy-system* [9]. Sparse integers would be dealt with as usual, and one could then hope for the best of both worlds - dense & sparse.

A number of natural extensions can be made to the *IDD* package, for multi-sets, polynomials, and sets of points in the plane. In each case, the extension is made through some integer encoding which transforms the operations from the application area into natural operations over binary numbers. In theory, each extension has the same order complexity as the best known specialized implementations, and it uses less memory. In practice, each extension performs faster than the specialized one over *sparse structures*.

Acknowledgments I am grateful to Don Knuth for his enlightening criticisms of an earlier draft, and for number $256 = \mathtt{x}_4$ exercise in [12].

References

[1] The gnu multiple precision arithmetic library. *http://gmplib.org/*, 2007.

[2] J. Chailloux & al. *LE-LISP de l'INRIA : le manuel de reference*, volume 15-2. I.N.R.I.A, 1986.

[3] R. E. Bryant. Symbolic boolean manipulations with ordered binary decision diagrams. *ACM Comp. Surveys*, 24:293–318, 1992.

[4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. *Design Automation Conf.*, pages 535–541, 1995.

[5] W. Weaver C. E. Shannon. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 19497.

[6] A. Frey, G. Berry, P. Bertin, F. Bourdoncle, and Jean Vuillemin. The jazz home page http://www.exalead.com/jazz/index.html. 1998.

[7] 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.

[8] J. C. Kiefer, E. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46:1227–1245, 2000.

[9] D. E. Knuth. *The Art of Computer Programming, vol. 1, Fundamental Algorithms*. Addison Wesley, 3-rd edition, 1997.

[10] D. E. Knuth. *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*. Addison Wesley, 3-rd edition, 1997.

[11] D. E. Knuth. *The Art of Computer Programming, vol. 3, Sorting and Searching*. Addison Wesley, 2-nd edition, 1998.

[12] D. E. Knuth. *The Art of Computer Programming, vol. 4A, Enumeration and Backtracking: http://www-cs-faculty.stanford.edu/ uno/taocp.html*. Addison Wesley, 2009.

[13] J. McCarthy. The implementation of lisp. Stanford University.

[14] S. I. Minato. Zero suppressed decision diagrams. *ACM/IEEE Design Automation Conf.*, 30:272–277, 1993.

[15] 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.

[16] 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.