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

Jean E. Vuillemin

March 12, 2009

Abstract

We construct natural number $n \in \mathbf{N} + 2$ through *trichotomy*

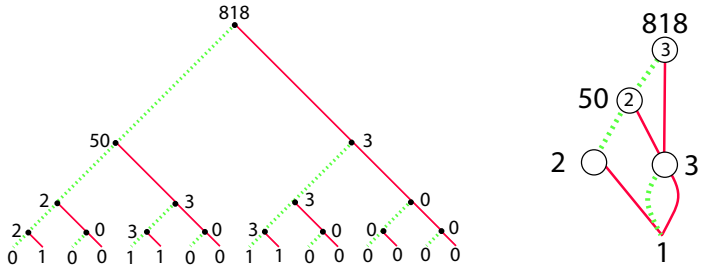
$$n = g + \mathbf{x}_p d,\ \mathbf{x}_p = 2^{2^p},\ 0 \leq g < \mathbf{x}_p,\ 0 < d < \mathbf{x}_p,$$

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

Over *dense* numbers, the worst/average time/space complexity of *IDDs* arithmetic operations is proportional to that of bit-arrays. Yet, equality testing is performed in unit time with *IDDs* and the time/space complexity of some operations (e.g. $sign(n - m)$, $n \pm 2^m$, 2^{2^n}) is (at least) exponentially faster with *IDDs* than bit-arrays, even over *dense* operands. Over *sparse* operands, the time/space complexity of ALU operations $\{\cap, \cup, \oplus, +, -\}$ is also in general arbitrarily better with *IDDs* than bit-arrays.

The coding powers of integers lets us use *IDDs* to represent *binary strings/integer sets/predicates/polynomials* as well as numbers. The result a single alternative to a number of successful (yet rather different) packages for processing large numbers, dictionaries, Boolean functions & more. Performance levels are comparable over dense structures, and *IDDs* prove *best in class* over sparse structures.

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



$818 = {}_2 0100110011$

Figure 1: Bit-array, bit-tree and bit-dag for decimal integer 818. The integer labels $\{0, 1, 2, 3, 50, 818\}$ at tree & dag nodes are *computed* rather than *stored*. Labels $\{2, 3\}$ inside the DAG (*Directed Acyclic Graph*) represent pointers to nodes in the bit-dag labeled by integers 2 and 3.

1 Introduction

To ease the presentation, we postpone discussing machine specific word size optimizations to sect. 2.5. Pretend till then to operate on a (bare-bone) computer with minimal word size $w = 1$ bit. Also, negative numbers $\mathbf{Z}$ are only introduced in sect. 2.4; meanwhile, we only discuss natural numbers $\mathbf{N}$.

1.1 Bit arrays, trees and dags

1.1.1 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

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

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

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

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

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

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. 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 constant involved in allocating/de-allocating bit-arrays is not negligible against the complexity of other linear time $O(\mathbf{l}(n))$ arithmetic operations, such as comparison and *in-place* ALU operations. Efficient storage allocation is a well-known key to efficient processing of very large numbers [7].

1.1.2 Bit-tree

Lisp languages [2] 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.

Representing an integer by its bit-list[1], 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* [2].

In LeLisp arithmetics [2], we use a *medium 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 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 ($p = \lceil \log_2 \log_2(n+1) \rceil$ for $n > 0$):

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

Dichotomy gives access to any bit of n in time proportional to the depth p of the bit-tree. Dichotomy achieves $L.E.$ for *pre-order* [9] bit-tree traversal, and $B.E.$ for post-order. Dichotomy concisely codes each operation on 2 digits numbers, through *divide & conquer*, by a sequence of (recursive) operations

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

on single digits integers. Dichotomy requires near twice the storage for bit-arrays, in exchange for an automatic memory allocation scheme.

An experimental benchmark (recorded on *real life* continued fractions [14]) of bit-trees in [2] vs. bit-arrays in [7] shows that, on a word-size $w = 16$ bits computer, the average time for bit-tree operations is within a factor $c < 4$ slower than bit-arrays - in agreement with the (easy to do) theory.

1.1.3 Bit-dag

The *IDD* representation combines dichotomy with a motto:

store once!

A bit-dag represents a bit-tree by sharing a single memory big-dag address for all the nodes with equal integer value (fig 1). We make sure (at creation time sec. 2) that two nodes with equal integer value n share a pointer to the same memory address $n.a$ in the DAG.

An integer is uniquely decomposed $n = \mathtt{T}(n.0, p, n.1) = n.0 + \mathtt{x}_p n.1$ into the triple: $p = \mathbf{ll}(n) - 1$ is the depth minus one, so that $n < \mathtt{x}_{p+1}$ and $n \geq \mathtt{x}_p$ for $n > 0$; the most $n.1 = n \div \mathtt{x}_p$ and least $n.0 = n \cdot|\cdot \mathtt{x}_p$ significant digits of n are the quotient and rest in the integer division by $\mathtt{x}_p$. By construction: $\max(n.0, n.1) < \mathtt{x}_p$ and $(n > 1 \Leftrightarrow 0 \neq n.1)$. Conversely, a triple g, p, d of integers results from the trichotomy decomposition $n = \mathtt{T}(n.0, n.p, n.1)$ of $n = g + \mathtt{x}_p d$, i.e. $g = n.0$, $p = n.p$, $d = n.1$ if and only if

$$\max(g, d) < \mathtt{x}_p,\ d \neq 0. \tag{3}$$

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 in less than $c < 8$ the time for bit-arrays; and within less than a third of the memory, since many continued fraction expansions in the benchmark from [14] are sparse.

DAG Sizes Integer labels in the bit-dag are the least set containing n and closed under trichotomy. This set $\mathcal{S}(n)$ of labels to DAG nodes is defined by:

$$\begin{array}{rcl} \mathcal{S}(0) = \mathcal{S}(1) & = & \{\}, \\ (3) \Rightarrow \mathcal{S}(\mathtt{T}(g, p, d)) & = & \{g + \mathtt{x}_p d\} \cup \mathcal{S}(g) \cup \mathcal{S}(p) \cup \mathcal{S}(d). \end{array} \tag{4}$$

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

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

$$n > 0 \Rightarrow \mathbf{s}(n) < \mathbf{l}(n). \tag{5}$$

More sharing takes place as n gets bigger, since

$$\mathbf{s}(n) \quad < \quad \frac{2\mathbf{l}(n)}{\mathbf{ll}(n) - \mathbf{l}(\mathbf{ll}(n))}. \tag{6}$$

The *worst case* $\mathbf{w}(p) = \max\{\mathbf{s}(k) \,:\, k < \mathbf{x}_p\}$ is shown in [15] to be such that:

$$1 = \liminf_{p \mapsto \infty} p2^{-p}\mathbf{w}(p),\ 2 = \limsup_{p \mapsto \infty} p2^{-p}\mathbf{w}(p),$$

so inequality (6) is "tight". The *average* $\mathbf{a}(p) = \frac{1}{\mathbf{x}_p}\sum_{k<\mathbf{x}_p} \mathbf{s}(k)$ is near worst:

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

In other words, a "random" integer $n < \mathbf{x}_p$ is *dense*: the size of its bit-dag is near the worst case $\mathbf{s}(n) \simeq \mathbf{w}(p)$ with probability near 1. An equivalent way is to regard as *dense* any number $n \in \mathbf{N}$ whose *bit-size* $\mathbf{B_s}(n) > \mathbf{l}(n)$ is greater than the binary length. Indeed, we have

$$\mathbf{B_s}(n) = \mathbf{s}(n)(\mathbf{l}(\mathbf{s}(n)) - 1) \quad < \quad 2\mathbf{l}(n) \tag{7}$$

for all n, and $\mathbf{B_s}(n) > \mathbf{l}(n)$ for "almost all" integers. Observe that $3\mathbf{B_s}(n)$ is a naive upper-bound on the total number of bits in all pointers needed to represent n by trichotomy. Using a more elaborate coding scheme (i.e. a smart print routine for bit-dags), Kiefer & al. [8] show that the bit-size is proportional to the source entropy [5], under some specific stochastic model of the input bits from n. In other words, representing binary sequences by integer *IDDs* is a general purpose *entropy compression scheme*.

In particular, we see from (16) and fig. 2 that consecutive numbers are efficiently coded by bit-dags. In the limit, coding all consecutive numbers up to n is optimal (against $n\mathbf{l}(n)$ for bit-arrays):

$$s(2, \cdots, n) \quad = \quad n - 1. \tag{8}$$

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 $n(n+1)/2$.

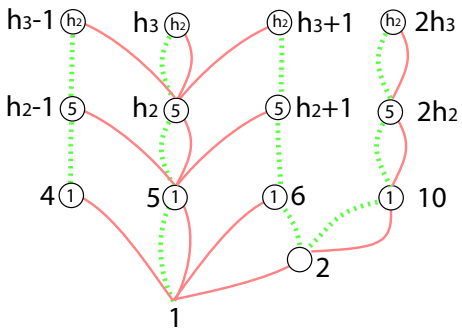


Figure 2: Huge? These 13 DAG nodes represents $\{h_3 - 1, h_3, h_3 + 1, 2h_3\}$.

1.2 Trichotomy

A theoretical advantage of *IDDs* 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.

1.2.1 Dictionaries

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

$$\begin{array}{rcl} \{n_1, \cdots, n_s\} & \rightleftharpoons & 2^{n_1} + \cdots + 2^{n_s} \\ \{k \,:\, k \in n\} & \rightleftharpoons & n = \sum_{k \in n} 2^k \end{array} \tag{9}$$

which transforms set operations ($\cup$, $\cap$, $\oplus$) into logical ones (&, |, $\oplus$). We similarly define, for $k, n \in \mathbf{N}$: $k \in n \Leftrightarrow 1 = \mathtt{B}_k^n$ and $k \subseteq n \Leftrightarrow k = k \cap n$. For example, $\{k : k \in 818\} = \{1, 4, 5, 7, 8, 9\}$. The set defined by (12) has 2^n elements, e.g. $\{k : k \in h_3\} = \{1, 4, \mathbf{x}_5, 4\mathbf{x}_5, \mathbf{x}_{h_2}, 4\mathbf{x}_{h_2}, \mathbf{x}_5\mathbf{x}_{h_2}, 4\mathbf{x}_5\mathbf{x}_{h_2}\}$.

The size of set $\{k \,:\, k \in n\}$ is the *binary weigth*[2] of $n \in \mathbf{N}$:

$$\nu(n) = \sum_{k < \mathbf{l}(n)} \mathtt{B}_k^n = |\{k \,:\, k \in n\}|. \tag{10}$$

The binary length $\mathbf{l}(n) = i + 1$ of $n > 0$ is equal to the largest $i = \max\{n_i \,:\, n_i \in N\}$ element of N plus one, and its depth to $\mathbf{ll}(n) = \mathbf{l}(i)$. Testing for membership $k \in$ 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, which all translate by (9) to efficient trichotomy operations.

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

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 (9). An example which *IDDs* can handle, unlike no other structure, is the set $\{k \ : \ k \in h_n\}$ of one bits in (say) h_{1024} (12).

In general, $\mathbf{s}(n) \leq \nu(n)\mathbf{ll}(n)$ and the size of set representations is smaller with *IDDs* than with sorted lists of integers (size $\nu(n)\mathbf{l}(n)$). Because of sharing and regardless of density, this size is also smaller than any (unshared) tree representation of dictionaries, such as binary tries or Patricia trees [11].

1.2.2 Predicates

Boolean function are just as efficiently represented by *IDDs* through the natural (truth-table) isomorphism

$$f \in \mathbf{B}^i \mapsto \mathbf{B} \quad \rightleftharpoons \quad F = \tau(f) = \textstyle\sum_{n<2^i} f(\mathtt{B}_0^n, \cdots, \mathtt{B}_{i-1}^n)2^n. \tag{11}$$

Note that (11) transforms again set operations ($\cup$, $\cap$, $\oplus$, $\cdots$) into logical ones (&, |, $\oplus$, $\cdots$). Is is observed in [12] (exer. 256) that the *IDD* for the truth-table $F = \tau(f)$ is isomorphic to the *zero suppressed decision diagrams* ZDD [13]. With a theoretical difference: depth pointers are coded by integers in ZDDs, and by pointers to DAG nodes in *IDDs*. In practice (once optimized for word size - sec. 2.5), 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 *IDDs* to those on *Binary Moment Diagrams* (BMD [4]) and *Binary Decisions Diagram* (BBD [3]). So, in theory, the proposed integer *IDD* package can manipulate boolean functions (i.e. set operations) just as well as the best known packages designed for that single purpose.

1.2.3 Dense and Sparse Integers

In addition, *IDDs* are also good at integer arithmetics.

Admittedly, most numbers are dense. Over dense numbers, the basic integer operations $+, -, \times, \cdot|\cdot, \div$ are slower with *IDDs* than with bit-arrays, by at most constant factor c (say $c < 8$). Over dense numbers, the space for *IDDs* is (arbitrarily) smaller than for bit-arrays. So, bit-dags and bit-arrays trade time for space over dense numbers. In addition, *IDDs* are arbirarily more efficient than bit-arrays for a number of useful operations (see sect. 2).

The advantages of *IDDs* appear over *sparse integers*. To illustrate the concept, consider the largest integer $h_s = \max\{n \ : \ \mathbf{s}(n) = s\}$ which can be

represented by s bit-dag nodes:

$$h_0 = 1, \qquad h_{s+1} = h_s(1 + 2^{2^{h_s}}) = \mathtt{r}(h_s, h_s, h_s). \tag{12}$$

Letter h stands here for *huge*: $h_1 = 5$, $h_2 = 21474836485 > 2^{34}$; the binary length of h_3 exceeds the number atoms on earth, and it will *never* be physically represented by bit-arrays. Physicists will 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 simply from (12) 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:

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

Yet, fig. 2 illustrates some DAG sizes related to h_3, and we find:

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

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

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

$$\begin{array}{rcl} \nu(0) = 0, & & \nu(1) = 1, \\ \nu(\mathtt{r}(g,p,d)) & = & \nu(g) + \nu(d). \end{array} \tag{13}$$

Tracing $\nu(h_n)$ reveals 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 (13) takes exponential time $O(2^n)$. This 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 already computed value; if not, we recursively compute $\nu(m)$ and duly record the result in $\mathcal{H}_\nu$, for further use. Upon returning the final result $\nu(n)$, table $\mathcal{H}_\nu$ is garbage collected.

Once (13) 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 *IDDs*.

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 $cmp(n,m) = (n = m)\,?\,0\ :\ (n > m)\,?\,1\ :\ -1$ is computed by

$$\begin{array}{rcl} cmp(n,m) & = & (n = m)\,?\,0\ : \\ & & (n.p \neq m.p)\,?\,cmp(n.p, m.p)\ : \\ & & (n.1 \neq m.1)\,?\,cmp(n.1, m.1)\ :\ cmp(n.0, m.0). \end{array} \tag{14}$$

At most 3 equality tests are performed at each node, and exactly one path is followed down the respective *IDDs* . 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.

A number of useful operations (see sect. 2.1) are also (at least) exponentially faster with bit-dags than bit-arrays: either *sparse operations*, whose result is sparse regardless of the operands (like 2^n), or any other operation on *sparse* enough operands (sect. 2).

2 Integer Decision Diagrams

Under condition (3) ($g < \mathbf{x}_p$ and $0 < d < \mathbf{x}_p$), we build a unique triple $n = \mathbf{t}(g,p,d) = g + \mathbf{x}_p d$ 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 allocated addresses $a = n.a$, 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 $n = \mathbf{t}(g,p,d)$ 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 word, the triple constructor $\mathbf{t}$ is a *global* memo function (using table $\mathcal{H}$).

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

$$(3) \Rightarrow n = \mathbf{t}(g,p,d) = g + \mathbf{x}_p d$$

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.

2.1 Fast Operations

Computing $\mathbf{x}_p = 2^{2^p}$ is performed in unit time and size $\mathbf{s}(\mathbf{x}_p) = 1 + \mathbf{s}(p) \leq \mathbf{l}(p)$ with *IDDs*, compared with time and space 2^p with bit-arrays. Similarly, 2^n

is computed below (20) by $2^n = A_M(0,n)$ in time $O(\mathbf{l}(n)\mathbf{ll}(n))$ and space

$$\mathbf{s}(2^n) \quad < \quad \nu(n) + \mathbf{l}(n). \tag{15}$$

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

$$\mathbf{s}(n, n-1) < \mathbf{s}(n) + \mathbf{ll}(n), \tag{16}$$

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

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

$$\begin{array}{rcl} n > 0 \Rightarrow D(n) & = & (n = 0)\,?\,-1\ :\ (n = 1)\,?\,0\ : \\ & & (n.0 \neq 0)\,?\,\mathtt{r}(D(n.0), n.p, n.1)\ : \\ & & (n.1 = 1)\,?\,x'(n.p)\ :\ \mathtt{r}(x'(n.p), n.p, D(n.1)). \end{array} \tag{17}$$

Function $x'(q) = \mathbf{x}_q - 1 = 2^{2^q} - 1$ is computed in time $O(q)$ by $x'(0) = 1$ and $x'(q+1) = \mathtt{r}(x'(q), q, x'(q))$. It follows that $\mathbf{s}(\mathbf{x}_q - 1) < 2q$ 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 time/space price $q = \mathbf{ll}(n) - 1$ at each individual I/D operation (without memo).

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

$$\begin{array}{rcl} I(n) & = & (n = 0)\,?\,1\ :\ (n = 1)\,?\,\mathtt{r}(0,0,1)\ : \\ & & (n.0 \neq x'(n.p))\,?\,\mathtt{r}(I(n.0), n.p, n.1)\ : \\ & & (n.1 \neq x'(n.p))\,?\,\mathtt{r}(0, n.p, I(n.1))\ :\ \mathtt{r}(0, I(n.p), 1). \end{array} \tag{18}$$

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

Add/remove MSB The operations of removing $R_M(n) = (m, i)$ the MSB ($n > 0$, $n = m + 2^i$, $i = \mathbf{l}(n) - 1$, $m = n - 2^i$), or adding the MSB $A_M(m, i) = m + 2^i$ (for $m < 2^i$) are defined by the mutually recursive pair:

$$\begin{array}{rcl} R_M(1) & = & (0,0); \\ R_M(\mathtt{r}(g,p,d)) & = & (m, A_M(l,p)) \\ & & \{(e,l) = R_M(d);\ \ //d = e + 2^l \\ & & \ \ m = (e = 0)\,?\,g\ :\ \mathtt{r}(g,p,e)\}. \end{array} \tag{19}$$

$$\begin{array}{rcl} A_M(m,0) & = & m+1;\ A_M(m,1) = m+2; \\ A_M(m,i) & = & (l > m.p)\,?\,\mathtt{r}(m,l,A_M(0,e))\ :\ \mathtt{r}(m.0,l,A_M(m.1,e)) \\ & & \{(e,l) = R_M(i) \qquad\qquad\qquad\qquad //i = e + 2^l\}. \end{array} \tag{20}$$

The justification for (19) 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 (20) 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 $n = m + \mathbf{x}_l(0 + 2^e)$. An analysis of (19,20) shows that, for $n = m + 2^i$, the time for computing $A_M(m,i)$ or $R_M(n)$ is $O(\mathbf{l}(n)\mathbf{ll}(n))$: indeed, (20,19) 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 $O(\mathbf{l}(n))$ 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 (18,17) 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 $m \pm 2^i$.

2.2 ALU operations

Constructors Let us weaken the pre-condition (3) on triplets $\mathtt{r}$ to

$$\max(g,d) < \mathbf{x}_{p+1}, \tag{21}$$

and define constructor $C(g,p,d) = g + \mathbf{x}_p d$ (assuming (21) by:

$$\begin{array}{rcl} C(g,p,d) & = & (d = 0)\,?\,g\ : \\ & & (p = g.p)\,?\,C(g.0,p,A(d,g.1))\ : \\ & & (p = d.p)\,?\,\mathtt{r}(C(g,p,d.0),I(p),d.1)\ :\ \mathtt{r}(g,p,d). \end{array} \tag{22}$$

Note that C relies on incrementation (18) and addition A defined below (25).

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:

$$2 \times 0 = 0,\ 2 \times 1 = 2,\ 2 \times \mathtt{r}(g,p,d) = C(2 \times g, p, 2 \times d). \tag{23}$$

It relies on constructor C (22) 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, their 0 parity may change to a 1 (shifted out of the previous digit). 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

$$\mathbf{s}(2a) \leq 2\mathbf{s}(a). \tag{24}$$

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:

$$\begin{aligned} A(a,b) \;=\; & (a=0)\,?\,0 \;:\; (a=1)\,?\,I(a) \;: \\ & (a=b)\,?\,2\times a \;:\; (a>b)\,?\,A(b,a) \;: \\ & (a.p>b.p)\,?\,C(A(a,b.0),b.p,b.1) \;:\; A_m(a,b), \end{aligned} \tag{25}$$

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

$$A_m(a,b) = C(A(a.0,b.0),a.p,A(a.0,b.0)). \tag{26}$$

The reason for separating the case $a.p=b.p$ (26) from the others (25) is to declare A_m as a (local) memo function, but not A. In this way, table $\mathcal{H}_{A_m}$ only stores the results of the additions $A(a,b)$ (with $a<b$ and $a.p=b.p$) which are recursively computed. The size of table $\mathcal{H}_{A_m}$ 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 (26) can at most double that size. It follows that

$$\mathbf{s}(a+b) < 2\mathbf{s}(a)\mathbf{s}(b). \tag{27}$$

Again, 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 $a-b = 1+a+(\mathbf{x}_p-b-1)-\mathbf{x}_p = d.0$, where $d=I(A(a,b'))$ and $b'=\mathbf{x}_p-b-1$ is b's *two's complement*. An easy exercise in trichotomy shows that $\mathbf{s}(b') < \mathbf{s}(b)+p$. Combining with (27) yields

$$a>b \Rightarrow \mathbf{s}(a-b) < 2\mathbf{s}(a)(\mathbf{s}(b)+\mathbf{ll}(a)-1) \tag{28}$$

and the difference of two sparse enough numbers is sparse.

Logic Operations Due to space limitations, we refer to [12] and the correspondance with ZDDs to perform logical operations on bit-dags, and find:

$$\max\{\mathbf{s}(a\cup b), size(a\cap b), size(a\oplus b)\} < \mathbf{s}(a)\mathbf{s}(b). \tag{29}$$

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

2.3 Multiplication

Integer multiplication and division have nice trichotomy definitions, although we don't present division, due to lack of space. The trichotomy product $P(n, m) = n \times m$ is defined by:

$$\begin{array}{rcl} P(a,b) & = & (a=0)\,?\,0\ :\ (a=1)\,?\,b\ :\ (a>b)\,?\,P(b,a)\ : \\ & & C(P(a,b.0),b.p,P(a,b.1)). \end{array} \tag{30}$$

We declare P to be a memo-function, and the size of the 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 (30) is now hiding *digit carries*, as opposed to bit carries in 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). A simpler example is provided by the product (shift) $n \times 2^m$ whose size can be as big as $2^m \mathbf{s}n$.

2.4 Negative numbers

We represent a relative number $z \in \mathbf{Z}$ by the pair $(s = sign(z), n = abs(n))$ of its sign (1 if $z < 0$, else 0) and its absolute value represented by a bit-dag. We define the opposite by $-z = (n = 0)\,?\,(0,0)\ :\ (1-s, n)$ and the logical negation by $\neg z = -(1+n)$. We extend all previously defined operations from $\mathbf{N}$ to $\mathbf{Z}$ in the obvious way.

2.5 Word size optimization

Finally, for the sake of software efficiency, the recursive decomposition should not be carried out all the way down to bits, but rather be stopped at the machine-word size, say $32 = \mathbf{x}_5$ or $64 = \mathbf{x}_6$ bits. In this manner, primitive machine operations rather than recursive definitions are used on word size operands, at unit cost.

3 Conclusion

With word size optimization, the dichotomy package becomes competitive, and one 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 one, 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 substructures from sparse ones. One could then implement a hybrid structure, where dense integers are represented by their bit-arrays, operated upon without memo functions (useless 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 We thank Don Knuth for his truly constructive criticism of an early draft.

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

[14] 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.

[15] 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. Celebrating Jean-Louis Lassez's fifth cycle birthday.