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

\[ n = g + x_p d, \quad x_p = 2^p, \quad 0 \leq g < x_p, \quad 0 < d < x_p, \]

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 \(s(n)\) nodes in computer memory. IDDs compete with bit-arrays, which represent the consecutive bits of \(n\) within roughly \(l(n)\) contiguous bits in memory. Unlike the binary length \(l(n)\), the size \(s(n)\) 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\((n - m)\), \(n \pm 2^m\), \(2^{2n}\)) are (at least) exponentially better with IDDs 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 IDDs than bit-arrays.

The coding powers of integers lets IDDs implement integer sets and predicates as well as arithmetics. The IDD package is a one-shop alternative to 3 (and more) successful yet rather different packages for processing large numbers, dictionaries and Boolean functions. 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.

1 Introduction

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

1.1 Bit arrays, trees and dags

1.1.1 Bit-array

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

\[ n_{0\ldots l-1} = 2^n \prod_{i\in N} \cdot n_{i-1} \]

the significant \(k < l\) bits \(n_k = n_k^B \in B = \{0, 1\}\) of

\[ n = n_0\ldots \sum_{k \in \mathbb{N}} n_k 2^k = \sum_{k<l} n_k 2^k. \]

The binary length \(l = l(n) = \left \lfloor \log_2(n + 1) \right \rfloor\) of \(n \in \mathbb{N}\) is defined by

\[ l(n) = (n = 0) ? 0 : 1 + l(n \div 2). \]

Note that \(0 = l(0)\) and \(l(n) = 1 + \left \lfloor \log_2(n) \right \rfloor\) for \(n > 0\).

For example, integer 818 (3 decimal digits) has \(l(818) = 10\) bits, namely: \(818 = \_01001100110\). Index 2 is written

\[ 818 = \_01001100110. \]
which are smaller than the length are the bits of balanced binary bit-tree, like tree compromise. Each integer is represented by a (perfectly

efficient.

tion and garbage collection is fully automatic and hopefully word per

time and memory) 1.1.2 Bit-tree

1.1.3 Bit-dag

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. 2) 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

\[ n = r(n, 0, n, 1) = n.0 + x_p n.1, \]

such that: \( p = \Pi(n) - 1 \) is the depth minus one (hence \( x_p \leq n < x_{p+1} \) for \( n > 0 \)); the MSD and LSD digits of \( n \) are the quotient \( n.1 = n \div x_p \) and rest \( n.0 = n \div x_p \) in the integer division of \( n \) by \( x_p \). By construction: \( \max(n, 0, n.1) < x_p \) and \((n > 1) \Rightarrow 0 \neq n.1 \). Conversely, three integers \( g, p, d \) result from the trichotomy decomposition of \( n = g + x_p d = r(n, 0, n, 1) \) (i.e. \( g = n.0, d = n.1 \) and \( p = n.p \)) if and only if

\[ \max(g, d) < x_p \text{ and } d \neq 0. \]  

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)

--

---

1 The list notation \((a b)\) stands in Lisp for \((a.(b.))\)

2 Most Significant Digit
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 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:

\[
\mathcal{S}(0) = \mathcal{S}(1) = \{\},
\]

\[
\mathcal{S}(x(g, p, d)) = \{ g + x_p d \} \cup \mathcal{S}(g) \cup \mathcal{S}(p) \cup \mathcal{S}(d).
\]  

(6)

The size of $n$ is the number $s(n) = |\mathcal{S}(n)|$ 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 $s(818) = 4$.

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

\[
n > 0 \Rightarrow s(n) < I(n).
\]  

(7)

The bigger $n$, the more sharing [12, 15]:

\[
s(n) < \frac{2l(n)}{\Pi(n) - I(\Pi(n))}.
\]  

(8)

In the limit, the size is (roughly) proportional to twice the length divided by the depth. It is shown in [15] that the worst size $w(p) = \max\{s(k) : k < x_p\}$ is such that:

\[
1 = \liminf_{p \to \infty} p2^{-p}w(p), \quad 2 = \limsup_{p \to \infty} p2^{-p}w(p).
\]

The average size $a(p) = \frac{1}{x_p} \sum_{k < x_p} s(k)$ is near the worst:

\[
1 = \limsup_{p \to \infty} w(p)/a(p), \quad 1 - \frac{1}{2e} = \liminf_{p \to \infty} w(p)/a(p) \approx 0.81606 \cdots
\]

Moreover, every "random" integer $n < x_p$ is dense: its size is near worst $s(n) \simeq w(p)$ with probability near 1.

The bit-size $u_\infty(n) = s(n)(I(s(n)) - 1)$ is a measure of the code length for representing the bit-dag by a finite sequence of bits. The analysis in [15] can be sharpened:

\[
u_\infty(n) = s(n)(I(s(n)) - 1) < 2l(n)
\]  

(9)

for all $n$. It also follows that $u_\infty(n) > 1(n)$ for "almost all" integers. We may as well call dense any number $n \in \mathbb{N}$ whose bit-size $u_\infty(n) > 1(n)$ is greater than its binary length.

![Figure 2. Huge? These 13 DAG nodes represents \{h_3 - 1, h_3, h_3 + 1, 2h_3\}.](image)

Observe that $3u_\infty(n)$ 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 stochastic model of the input bits from $n$. In other words, representing binary sequences by integer IDD's is a form of entropy compression.

In particular, we see from (17) 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

\[
s(1, \ldots, n) = n - 1,
\]  

(10)

against $nI(n)$ for bit-arrays. Another (near) optimal example is the bit-dag for all 2-powers up to $n$:

\[
s(2^0, \ldots, 2^n) < n + 1(n).
\]

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

1.2 Trichotomy

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.

1.2.1 Dictionaries

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

\[
\{n_1, \ldots, n_s\} \cong 2^{n_1} + \cdots + 2^{n_s}, \quad \{k : k \in n\} \cong n = \sum_{k \in n} 2^k
\]  

(11)

which transforms set operations ($\cup$, $\cap$, $\oplus$) into logical ones ($\lor$, $\land$, $\oplus$). We similarly define, for $k, n \in \mathbb{N}$:

\[
k \in n \iff 1 = b_k^n
\]
representation, and parallel counter. The size of set \( \{ k : k \in n \} \) is the binary weight3 of n:

\[
\nu(n) = \sum_{k \leq n} v_k^3 = | \{ k : k \in n \} |. \tag{12}
\]

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

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

The \textit{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 (11). An example which \textit{IDDs} handle like no other structure, is the set \( \{ k : k \in h_n \} \) of one bits in (say) \( h_{1024} \).

In general, \( s(n) \leq \nu(n)I(n) \) and the size of set representations is smaller with \textit{IDDs} than with sorted lists of integers (size \( \nu(n)I(n) \)). Because of sharing and regardless of density, the bit-dag is also smaller than any (un-shared) tree representation of dictionaries, such as binary tries or Patricia trees [11].

1.2.2 Predicates

Boolean function are just as efficiently represented by \textit{IDDs} through the natural (truth-table) isomorphism [12]

\[
f \in B^i \leftrightarrow B \vdash \tau(f) = \sum_{n \leq 2^i} f(n_0, \ldots, n_{i-1})2^n.
\]

which also transforms set operations (\( \cup, \cap, \oplus \)) into logical ones (\&, \|, \oplus). Is observed in [12] (exer. 256) that the \textit{IDD} for the integer truth-table \( \tau(f) \) 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 \textit{IDDs}. In practice (once optimized for word size - sec. 2.5), this hardly matters; it limits ZDDs to 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 \textit{IDDs} to those on Binary Moment Diagrams (BMD [4]) and Binary Decisions Diagram (BBD [3]). So, in theory, the proposed integer \textit{IDD}

Admittedly, most numbers are dense, and over dense numbers, the basic integer operations \( +, -, \times, \div, \vdash \) are slower with \textit{IDDs} than with bit-arrays: by at most constant factor \( c - \text{say} c < 8 \). Over dense numbers, the space for \textit{IDDs} is (arbitrarily) smaller than for bit-arrays. So, bit-dags and bit-arrays trade time for space over dense numbers.

In addition, \textit{IDDs} are arbitrarily more efficient than bit-arrays at a number of useful operations (sec. 2).

The advantages of \textit{IDDs} appear over sparse integers. To illustrate the concept, consider the largest integer \( h_s = \max \{ n : s(n) = s \} \) which can be represented by a bit-dag with \( s \) nodes:

\[
\begin{align*}
h_0 &= 1, \\
h_{s+1} &= h_s + 2^{h_s} = \nu(h_s, h_s, h_s). \tag{13}
\end{align*}
\]

Letter \( h \) stands here for huge: \( h_1 = 5 \), and \( h_2 = 21474836485 > 2^{34} \); 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 (13) 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 \( h_n \):

\[
\begin{align*}
I(h_n) &= \sum_{k < n} 2^{h_k}; \\
\nu(h_n) &= h_n - 1; \\
\nu(h_n) &= 2^n.
\end{align*}
\]

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

\[
\begin{align*}
(s(h_n + 1) &= 2n, \\
(s(h_n - 1) &= 2n, \\
s(2h_n) &= 2n + 1.
\end{align*}
\]

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

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

\[
\begin{align*}
\nu(0) &= 0, \\
\nu(1) &= 1, \\
\nu(\nu(g, p, d)) &= \nu(g) + \nu(d). \tag{14}
\end{align*}
\]

Tracing the computation of \( \nu(h_n) \) according to (14) 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 (14) takes exponential time \( O(2^n) \). The problem is fixed by (automatically) turning \( \nu \) into a local
memo function. On the first call to $\nu(n)$, a table $\mathcal{H}_v$ 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. $m \in \mathcal{H}_v$; 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 $\mathcal{H}_v$, for further use. Upon returning the final result $\nu(n)$, table $\mathcal{H}_v$ is garbage collected. Once (14) 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, and we can perform some genuine computations with "monsters" like $h_{1024}$.

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

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

(15)

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

A number of useful operations (see sec. 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 (see sec. 2).

### 2 Integer Decision Diagrams

Under condition (5) ($g < x_p$ and $0 < d < x_p$), we build a unique triple $n = \tau(g, p, d) = g + x_p d$ at a unique memory address $n.a$. This is achieved through a global hash table $\mathcal{H} = \{(n,h,n.a) : n \in \text{memory}\}$, which stores all pairs of unique hash-code $h = n.h = \text{hash}(g,a,p,a,d.a)$ 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 $n = \tau(g, p, d)$ 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 $\tau$ 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

$$
5 \Rightarrow n = \tau(g, p, d) = g + x_p d
$$

and accessing the trichotomy fields

$$
n.0 = g, \ n.p = p = l(n) - 1, \ n.1 = d
$$
or the node address $n.a$, are all computed in constant time with bit-dags.

### 2.1 Fast Operations

Computing $x_p = 2^{sp}$ is performed in unit time and size $s(x_p) = 1 + s(p) \leq l(p)$ with IDDs, compared with time and space $2^{sp}$ with bit-arrays. Similarly, $2^n$ is computed by $2^n = A_M(0, n)$ (see (21)) in time $O(l(n)l(n))$ and space

$$
s(2^n) < \nu(n) + l(n). \quad (16)
$$

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 $O(l(n))$, by descending the DAG along a single path. In the worst case, this is exponentially faster than bit-arrays. Since

$$
s(n, n - 1) < s(n) + l(n), \quad (17)
$$

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

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

#### Decrement

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

$$
D(n) = \begin{cases} 
(n = 1) \? 0 : \\ (n.0 \neq 0) \? \tau(D(n.0), n.p, n.1) : \\ (n.1 = 1) \? \tau(x'(n.p)) : \\ \tau(x'(n.p), n.p, D(n.1)).
\end{cases}
$$

(18)

Function $x'(q) = x_q - 1 = 2^n - 1$ is computed in time $O(q)$ by $x'(0) = 1$ and $x'(q + 1) = x(x'(q), q, x'(q))$. It follows that $s(x_q - 1) < 2q$ 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 $q = l(n) - 1$ at each increment and decrement operation (without memo).

#### Increment

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

$$
I(n) = \begin{cases} 
(n = 0) \? 1 : \\ (n.0 \neq x'(n.p)) \? \tau(I(n.0), n.p, n.1) : \\ (n.1 \neq x'(n.p)) \? \tau(0, n.p, I(n.1)) : \\ \tau(0, I(n.p), 1).
\end{cases}
$$

(19)
Altogether, computing \( I(n) \) or \( D(n) \) requires \( II(n) \) operations to follow the single DAG path, to which we may (or not) have to add \( p = II(n) - 1 \) operations to compute \( x_p - 1 \).

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

\[
R_M(1) = (0, 0);
R_M((0, p, d)) = (\tau(g, p, e), A_M(l, p)) \quad \text{if} \quad (e, l) = R_M(d). \quad (20)
\]

\[
A_M(m, 0) = m + 1; \quad A_M(m, 1) = m + 2;
A_M(m, i) = (l > m, p) ? \tau(m, 1, A_M(0, e)) : \tau(0, 0, l, A_M(m, 1, e)) \quad \text{if} \quad (e, l) = R_M(i). \quad (21)
\]

The justification for (20) is: \( n = g + xₚd = g + xₚ(2^{e+2^1}) = m + 2^{e+2^1} \), where \( m = g + xₚe \) and \( i = l + 2^p \). The justification for (21) is: \( n = m + 2^l = m + x_{l+2^1} \), where \( i = e + 2^1 \) and \( l \geq m,p \) since \( m < 2^l \); if \( l = m.p \), we finish by \( n = m.0 + x_{s.(1+2^c)} \), else \( n = m + x_{l+2^c} \).

An analysis of (20,21) shows that, for \( n = m + 2^l \), \( m < 2^l \), the computing time for \( A_M(m, i) \) and \( R_M(n) \) is \( O(l(n)II(n)) \): indeed, (21,20) collectively follow a single DAG path with at most \( II(n) \) nodes; the operation at each node happens on numbers of length at most \( l(n) \), and its complexity is thus at most \( O(l(n)) \), 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 (19,18) and \( s(m \pm 2^{i}) = s(m) + 2II(m) \) in this case. Thus in general, the sparseness of \( m \) implies the sparseness of both \( m \pm 2^i \).

### 2.2 ALU operations

**Constructor** Replace pre-condition (5) by the weaker

\[
\max(g, d) < x_{p+1}, \quad (22)
\]

and define constructor \( C(g, p, d) = g + x_p d \) by:

\[
C(g, p, d) = \begin{cases} 
(0 = 0) \quad : \quad g \\
(p = g,p) \quad : \quad C(g, 0, p, A(d, g, 1)) \\
(p = d, p) \quad : \quad \tau(C(g, p, d, 0), I(p), d, 1) : \tau(g, p, d) 
\end{cases} 
\quad (23)
\]

Note that \( C \) uses incrementation (19), and its definition is mutually recursive with that of addition \( A \) (26).

**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:

\[
2 \times 0 = 0, \quad 2 \times 1 = 2,
2 \times \tau(g, p, d) = C(2 \times g, p, 2 \times d). \quad (24)
\]

It relies on constructor \( C \) (23) to pass "carries" from one digit to the next. Obviously, twice must be declared as a local memo function (with hash table \( H_{2 \times} \)), to stand a chance of computing, say \( 2 \times h_n \) (see. fig. 2).

The number of nodes in \( S(n) \) at depth \( q < ll(n) \) can at worst double in \( S(2 \times n) \): 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

\[
s(2a) \leq 2s(a). \quad (25)
\]

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 \( s(3n) \leq 2s(n) \). Thus, if \( n \) is sparse, so are \( 2n \) and \( 3n \) (fig. 2).

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

\[
A(a, b) = \begin{cases} 
(a = 0) \quad : \quad 0 \quad \text{n.o} \quad \text{I}(a) \\
(a = b) \quad : \quad 2 \times a \quad \text{n.} \quad \text{I}(b, a) \\
(a.p > b.p) \quad : \quad C(A(a, b, 0), b.p, b, 1) \quad : \quad A_m(a, b), \quad (26)
\end{cases}
\]

and, for \( 1 < a < b \) and \( a.p = b.p \), by

\[
A_m(a, b) = C(A(a, b, 0), a.p, A(a, 0, b, 0)). \quad (27)
\]

The reason for separating the case \( a.p = b.p \) (27) from the others (26) is to declare \( A_m \) as a (local) memo function, but not \( A \). In this way, table \( H_A \) 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 \( H_A \) is (strictly) less than \( s(a)s(b) \). By the argument already used to analyze \( 2 \times \), releasing the carries hidden in \( C \) by (27) can at most double that size, and

\[
s(a + b) < 2s(a)s(b). \quad (28)
\]

Hence, the sum two sparse enough numbers is sparse.

**Subtract** For \( a > b > 1 \) and \( p = a.p = II(a) - 1 \), we compute the difference by \( a - b = 1 + a + (x_p - b - 1) - x_p = d.0 \), where \( d = I(A(a, b')) \) and \( b' = x_p - b - 1 \) is \( b \)'s two's complement. An easy exercise in trichotomy shows that \( s(b') < s(b) + p \). Combining with (28) yields

\[
a > b \Rightarrow s(a - b) < 2s(a)(s(b) + II(a) - 1) \quad (29)
\]

and the difference of two sparse enough numbers is sparse.
Logic Operations  Due to space limitations, we refer to [12] and the correspondence with ZDDs to perform logical operations on bit-dags, to find:

$$\max\{s(a \cup b), size(a \cap b), size(a \oplus b)\} < s(a)s(b).$$ (30)

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. We won’t discuss division for lack of space. The trichotomy product $P(n, m) = n \times m$ is defined by:

$$P(a, b) = (a = 0) ? 0 : (a = 1) ? b : (a > b) ? P(b, a) : C(P(a, b, 0), b, p, P(a, b, 1)).$$ (31)

We declare $P$ to be a memo-function. The size of its hash table $H_P$ is bounded by the product $s(n)s(m)$ 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 (31) 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 $2^m s(n)$. The algorithms for big-shift (product by $x_p$) and shift (product by $2^p$) are left as fun exercises in trichotomy for the reader. Big-shift has an unusual property: the bigger, the better! Indeed, the product $d \times x_p = \tau(0, p, d)$ is computed in unit time and space, as soon as $d < x_p$.

2.4 Negative numbers

We represent a relative number $z \in \mathbb{Z}$ 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 $s(-n) = s(n)$. The logical negation follows by $\neg z = -(1 + n)$, and thus $s(-n) = s(1 + n) < s(n) + \Pi(n)$.

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 $\mathbb{N}$ to $\mathbb{Z}$, in the obvious way.

2.5 Word size optimization

For the sake of software efficiency, the recursive decomposition (5) is not carried out all the way down to bits, but stopped at the machine-word size, say $32 = x_5$ or $64 = x_6$ bits. In this manner, primitive machine operations rather than recursive definitions are used on word size operands, at minimal 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 worthwhile 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 (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  I am grateful to Don Knuth for his enlightening criticisms of an earlier draft.
References


