Double-base scalar multiplication revisited

Daniel J. Bernstein[1,2], Chitchanok Chuengsatiansup[1], and Tanja Lange[1]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
c.chuengsatiansup@tue.nl, tanja@hyperelliptic.org
[2] Department of Computer Science, University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to

Abstract. This paper reduces the number of field multiplications required for scalar multiplication on conservative elliptic curves. For an average 256-bit integer n, this paper's multiply-by-n algorithm takes just 7.47**M** per bit on twisted Edwards curves $-x^2 + y^2 = 1 + dx^2y^2$ with small d. The previous record, 7.62**M** per bit, was unbeaten for seven years.

Unlike previous record-setting algorithms, this paper's multiply-by-n algorithm uses double-base chains. The new speeds rely on advances in tripling speeds and on advances in constructing double-base chains. This paper's new tripling formula for twisted Edwards curves takes just 11.4**M**, and its new algorithm for constructing an optimal double-base chain for n takes just $(\log n)^{2.5+o(1)}$ bit operations.

Extending this double-base algorithm to double-scalar multiplications, as in signature verification, takes 8.80**M** per bit to compute $n_1P + n_2Q$. Previous techniques used 9.34**M** per bit.

Keywords: scalar multiplication, Edwards curves, triplings, double-base chains, directed acyclic graphs, double-scalar multiplication, signatures.

1 Introduction

Elliptic-curve computations have many applications ranging from the elliptic-curve method for factorization (ECM) to elliptic-curve cryptography. The most important elliptic-curve computation is scalar multiplication: computing the nth multiple nP of a curve point P. For example, in ECDH, n is Alice's secret key, P is Bob's public key, and nP is a secret shared between Alice and Bob.

This work was supported by the National Science Foundation under grants 1018836 and 1314919, by the Netherlands Organisation for Scientific Research (NWO) under grants 639.073.005 and 613.001.011, and by the European Commission through the ICT program under contract ICT-645421 ECRYPT-CSA. Date of this document: 2017.01.13.

There is an extensive literature proposing and analyzing scalar-multiplication algorithms that decompose $P \mapsto nP$ into field operations in various ways. Some speedups rely on algebraically "special" curves, but we restrict attention to the types of curves that arise in ECM and in conservative ECC: large characteristic, no subfields, no extra endomorphisms, etc. Even with this restriction, there is a remarkable range of ideas for speeding up scalar multiplication.

It is standard to compare these algorithms by counting field multiplications **M**, counting each field squaring **S** as 0.8**M**, and disregarding the overhead of field additions, subtractions, and multiplications by small constants. (Of course, the most promising algorithms are then analyzed in more detail in implementation papers for various platforms, accounting for all overheads.) For seven years the record number of field multiplications has been held by an algorithm of Hisil, Wong, Carter, and Dawson [19]; for an average 256-bit scalar n, this algorithm computes $P \mapsto nP$ using 7.62**M** per bit.

This paper does better. We present an algorithm that, for an average 256-bit scalar n, computes $P \mapsto nP$ using just 7.47**M** per bit. Similarly, for double-scalar multiplication, we use just 8.80**M** per bit, where previous techniques used 9.34**M** per bit.

We emphasize that the new 7.47**M** and 8.80**M** per bit, like the old 7.62**M** and 9.34**M** per bit, are averages over n; the time depends slightly on n. These algorithms leak some information about n through total timing, and much more information through higher-bandwidth side channels, so they should be avoided in contexts where n is secret. However, there are many environments in which scalars are not secret: examples include ECC signature verification and ECM. We present new algorithms for single and double scalar multiplication, covering these important applications.

We also emphasize that these operation counts ignore the cost of converting n into a multiply-by-n algorithm. The conversion cost is significant for the new algorithm. This does not matter if n is used many times, but it can be a bottleneck in scenarios where n is not used many times. To address this issue, this paper introduces a fast new conversion method, explained in more detail below; this reduces the number of times that n has to be reused to justify applying the conversion. Note also that, in the context of signature verification, this conversion can be carried out by the *signer* and included in the signature as an encoding of n. This helps ECC signatures in applications where so far RSA was preferred due to the faster signature verification speeds (with small public RSA exponent).

1.1. Double-base chains. One way to speed up scalar multiplication is to reduce the number of point operations, such as point additions and point doublings. For a simple double-and-add algorithm, using the binary representation of n, the number of point doublings is approximately the bitlength of n and the number of point additions is the Hamming weight of n, i.e., the number of bits set in n. Signed digits, windows, and sliding windows write n in a more general binary representation $\sum_{i=1}^{\ell} d_i 2^{a_i}$ with d in a coefficient set, and reduce the number of point additions to a small fraction of the bitlength of n.

Even though the binary representation is widely and commonly used, it is not the only way to express integers. A double-base representation, specifically a $\{2,3\}$-representation, writes n as $\sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}$, where $a_1 \geq a_2 \geq \cdots \geq a_\ell \geq 0$; $b_1 \geq b_2 \geq \cdots \geq b_\ell \geq 0$; and d_i is restricted to a specified set S, such as $\{1\}$ or $\{-1,1\}$. This representation is easy to view as a double-base chain to compute nP: a chain of additions, doublings, and triplings to compute nP, where each addition is of some $d_i P$. Double-base chains were introduced by Dimitrov, Imbert, and Mishra in [12] for the case $S = \{-1,1\}$, and were generalized to any S by Doche and Imbert in [17].

Representing an integer as a double-base chain allows various tradeoffs between doublings and triplings. This representation of n is not unique; selecting the fastest chain for n can save time. As a concrete example, we comment that the computation $2^{12}(2^3+1)-1$ used in [7] for ECM computations can be improved to $2^{12}3^2-1$, provided that two triplings are faster than three doublings and an addition.

For *some* elliptic-curve coordinate systems, the ratio between tripling cost and doubling cost is fairly close to $\log_2 3 \approx 1.58$. It is then easy to see that the total cost of doublings and triplings combined does not vary much across double-base chains, and the literature shows that within this large set one can find double-base chains with considerably fewer additions than single-base chains, saving time in scalar multiplication.

For example, twisted Hessian curves [4] cost 7.6**M** for doubling, 11.2**M** for tripling (or 10.8**M** for special fields with fast primitive cube roots of 1), and 11**M** for addition. A pure doubling chain costs $7.6(\log_2 n)\mathbf{M}$ for doublings, and a pure tripling chain costs $11.2(\log_3 n)\mathbf{M} \approx 7.1(\log_2 n)\mathbf{M}$ for triplings. Many different double-base chains have costs between $7.1(\log_2 n)\mathbf{M}$ and $7.6(\log_2 n)\mathbf{M}$ for doublings and triplings, while they vary much more in the costs of additions. The speeds reported in [4] use double-base chains, and are the best speeds known for scalar multiplication *for curves with cofactor 3*.

However, the situation is quite different for twisted Edwards curves. Compared to twisted Hessian curves, twisted Edwards curves cost more for tripling, less for doubling, and less for addition. The higher tripling-to-doubling cost ratio (close to 2) means that trading doublings for triplings generally loses speed, and the higher tripling-to-addition cost ratio (around 1.5) means that the disadvantage of extra triplings easily outweighs the advantage of reducing the number of additions.

The literature since 2007 has consistently indicated that the best scalar-multiplication speeds are obtained from (1) taking a curve expressible as a twisted Edwards curve and (2) using single-base chains. These choices are fully compatible with the needs of applications such as ECM and ECC; see, for example, [7] and the previous Edwards-ECM papers cited there. Bernstein, Birkner, Lange and Peters [3] obtained double-base speedups for some coordinate systems but obtained their best speeds from single-base Edwards. Subsequent improvements in double-base techniques have not been able to compete with single-base Edwards; for example, the double-base twisted Hessian speeds reported in [4],

above 8**M** per bit, are not competitive with the earlier single-base Edwards speed from [19], just 7.62**M** per bit.

Our new speed, 7.47**M** per bit, marks the first time that double-base chains have broken through the single-base barrier. This speed relies on double-base chains, more specifically optimal double-base chains (see below), and also relies on new Edwards tripling formulas that we introduce. These tripling formulas use just $9\mathbf{M} + 3\mathbf{S}$, i.e., 11.4**M**, saving 1**S** compared to the best previous results. This is almost as fast as the tripling speeds from tripling-oriented Doche–Icart–Kohel [16] and twisted Hessian [4], and has the advantage of being combined with very fast doublings and additions.

1.2. Converting n to a chain. Because there are so many choices of double-base chains for n (with any particular S), finding the optimal chain for n is not trivial (even for that S).

Doche and Habsieger [15], improving on chain length compared to previous papers, proposed a tree-based algorithm to compute double-base chains. They start with n at the root of a tree, and perform the following steps at each tree node: (1) remove all 2 and 3 factors; (2) add ± 1 to the unfactored part; (3) branch two new nodes for the obtained values. They repeat these processes until they obtain a node with value 1. To limit the size of the tree they prune each level as it is constructed, keeping only the smallest nodes at that level.

It does not seem to have been noticed that the algorithm of [15], without any pruning, finds an optimal double-base chain in time polynomial in n: the tree has only polynomially many levels, and there are only polynomially many possible nodes at each level. A recent paper by Capuñay and Thériault [8] presents an algorithm with an explicit $(\log n)^{4+o(1)}$ time bound to find an optimal double-base chain for n, assuming $S \subseteq \{-1, 1\}$.

We observe that finding an optimal double-base chain is equivalent to a shortest-path computation in an explicit directed acyclic graph with $O(\omega(\log n)^2)$ nodes, assuming $S \subseteq \{-\omega, \ldots, -1, 0, 1, \ldots, \omega\}$; in particular, $(\log n)^{2+o(1)}$ nodes when $\omega \in (\log n)^{o(1)}$. We actually build two such graphs, and show how one of the graphs allows optimized arithmetic, reducing the total chain-construction time to just $(\log n)^{2.5+o(1)}$. For comparison, scalar multiplication takes time $(\log n)^{3+o(1)}$ using schoolbook techniques for field arithmetic, time $(\log n)^{2.58\ldots+o(1)}$ using Karatsuba, or time $(\log n)^{2+o(1)}$ using FFTs.

1.3. Organization of this paper. Section 2 presents our new tripling formulas. Section 3 recasts the search for an optimal double-base chain as the search for a shortest path in an explicit DAG. Section 4 constructs our second explicit DAG, with a rectangular structure that simplifies computations; using this DAG to find the optimal double-base chain for n takes $(\log n)^{3+o(1)}$ bit operations. Section 5 shows how to use smaller integers to represent nodes in the same graph, reducing $(\log n)^{3+o(1)}$ to $(\log n)^{2.5+o(1)}$. Section 6 extends these ideas to generate optimal double-base chains for double-scalar multiplication. Section 7 compares our results to previous results.

2 Faster point tripling

This section presents faster point tripling formulas for twisted Edwards curves $ax^2 + y^2 = 1 + dx^2y^2$ in both projective and extended coordinates. We also show how to minimize the cost when mixing additions, doublings and triplings in different coordinate systems.

2.1. Projective coordinates. Recall that projective coordinates $(X : Y : Z)$, for nonzero Z, represent $(x, y) = (X/Z, Y/Z)$. These formulas are faster by one squaring than the previously best tripling formulas in projective coordinates [3], which in turn are faster than performing point doubling followed by point addition.

Here are the formulas for computing $(X_3 : Y_3 : Z_3) = 3(X_1 : Y_1 : Z_1)$, i.e., point tripling in projective coordinates. These formulas cost $9\mathbf{M} + 3\mathbf{S} + 1\mathbf{M}_a + 2\mathbf{M}_2 + 7\mathbf{A}$, where as before $\mathbf{M}$ denotes field multiplication, $\mathbf{S}$ denotes field squaring, $\mathbf{M}_a$ denotes field multiplication by curve parameter a, $\mathbf{M}_2$ denotes field multiplication by constant 2, and $\mathbf{A}$ denotes a general field addition.

$$YY = Y_1^2;\ aXX = a \cdot X_1^2;\ Ap = YY + aXX;\ B = 2(2Z_1^2 - Ap);\ xB = aXX \cdot B;$$
$$yB = YY \cdot B;\ AA = Ap \cdot (YY - aXX);\ F = AA - yB;\ G = AA + xB;$$
$$X_3 = X_1 \cdot (yB + AA) \cdot F;\ Y_3 = Y_1 \cdot (xB - AA) \cdot G;\ Z_3 = Z_1 \cdot F \cdot G.$$

For affine inputs where $Z_1 = 1$, computing $(X_3 : Y_3 : Z_3) = 3(X_1 : Y_1 : 1)$ costs only $8\mathbf{M} + 2\mathbf{S} + 1\mathbf{M}_a + 1\mathbf{M}_2 + 7\mathbf{A}$. That is, we save $1\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_2$ by ignoring multiplication and squaring of Z_1.

2.2. Extended coordinates. We also present similar tripling formulas in extended coordinates. Recall that extended coordinates $(X : Y : Z : T)$ represent $(x, y) = (X/Z, Y/Z)$, like projective coordinates, but also have an extra coordinate $T = XY/Z$. The importance of extended coordinates is that addition of points in extended coordinates is faster than addition of points in projective coordinates.

The following formulas compute $(X_3 : Y_3 : Z_3 : T_3) = 3(X_1 : Y_1 : Z_1)$, i.e., point tripling in extended coordinates. These formulas cost $11\mathbf{M} + 3\mathbf{S} + 1\mathbf{M}_a + 2\mathbf{M}_2 + 7\mathbf{A}$; in other words, it costs $2\mathbf{M}$ extra to compute the coordinate T.

$$YY = Y_1^2;\ aXX = a \cdot X_1^2;\ Ap = YY + aXX;\ B = 2(2Z_1^2 - Ap);\ xB = aXX \cdot B;$$
$$yB = YY \cdot B;\ AA = Ap \cdot (YY - aXX);\ F = AA - yB;\ G = AA + xB;$$
$$xE = X_1 \cdot (yB + AA);\ yH = Y_1 \cdot (xB - AA);\ zF = Z_1 \cdot F;\ zG = Z_1 \cdot G;$$
$$X_3 = xE \cdot zF;\ Y_3 = yH \cdot zG;\ Z_3 = zF \cdot zG;\ T_3 = xE \cdot yH.$$

For affine inputs where $Z_1 = 1$, computing $(X_3 : Y_3 : Z_3 : T_3) = 3(X_1 : Y_1 : 1 : T_1)$ costs only $9\mathbf{M} + 2\mathbf{S} + 1\mathbf{M}_a + 1\mathbf{M}_2 + 7\mathbf{A}$. That is, we save $2\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_2$ by ignoring multiplication and squaring of Z_1.

Note that the input for these formulas is projective $(X_1 : Y_1 : Z_1)$; to triple an extended $(X_1 : Y_1 : Z_1 : T_1)$ we simply discard the extra T_1 input. We could

instead compute $T_3 = T_1 \cdot (yB + AA) \cdot (xB - AA)$ and skip computing one of zF and zG but this would not save any multiplications.

2.3. Mixing doublings, triplings, and additions. Point doubling in extended coordinates [6] also takes projective input, and costs only 1**M** extra to compute the extra T output. The best known single-base chains compute a series of doublings in projective coordinates, with the final doubling producing extended coordinates; and then an addition, again producing projective coordinates for the next series of doublings.

In the double-base context, because triplings cost 2**M** extra to produce extended coordinates while doublings cost only 1**M** extra, we suggest replacing DBL-TPL-ADD with the equivalent TPL-DBL-ADD. More generally, the conversion from projective to extended coordinates should be performed after point doubling and not tripling (if possible). A good sequence of point operations and coordinate systems is as follows: For every nonzero term, first compute point tripling(s) in projective coordinates; then compute point doubling(s) in projective coordinates, finishing with one doubling leading to extended coordinates; finally, compute the addition taking both input points in extended coordinates and outputting the result in projective coordinates.

We still triple into extended coordinates if a term does not include any doublings (e.g., computing $(3^6 + 1)P$): i.e., compute point tripling(s) in projective coordinates, finishing with one tripling leading to extended coordinates; finally, as before, compute the addition taking both input points in extended coordinates and outputting the result in projective coordinates.

2.4. Cost of point operations when $a = -1$. Table 1 summarizes the costs for point operations for twisted Edwards curves. Our new tripling formulas presented in Sections 2.1 and 2.2 are for twisted Edwards curves for any curve parameter a; the table assumes $a = -1$ to include the point-addition speedup from [19]. The rest of the paper builds fast scalar multiplication on top of the point operations summarized in this table.

3 Graph-based approach to finding double-base chains

This section shows how to view double-base chains for n as paths from n to 0 in an explicit DAG. If weights are assigned properly to the edges of the DAG then the weight of a path is the same as the cost of the double-base chain. Finding the lowest-cost double-base chain for n is therefore the same as finding the lowest-cost ("shortest") path from n to 0. Dijkstra's algorithm [11] finds this path in time $(\log n)^{O(1)}$.

3.1. Double-base chains. We formally define a double-base chain as a finite sequence of operations, where each operation is either "$\times 2 + c$" for some integer c or "$\times 3 + c$" for some integer c. The integers c are allowed to be negative, and when c is negative we abbreviate "$+c$" as "$-|c|$"; e.g., the operation "$\times 3 + -7$" is abbreviated "$\times 3 - 7$"; c is also allowed to be 0.

Table 1: Cost of point operations for twisted Edwards curves with $a = -1$.

Operations	Coordinate systems	Cost
Mixed Addition	$\mathcal{E} + \mathcal{A} \to \mathcal{P}$	$6\mathbf{M} \approx 6.0\mathbf{M}$
Addition	$\mathcal{E} + \mathcal{E} \to \mathcal{P}$	$7\mathbf{M} \approx 7.0\mathbf{M}$
Doubling	$\mathcal{P} \to \mathcal{P}$	$3\mathbf{M}+4\mathbf{S} \approx 6.2\mathbf{M}$
Doubling	$\mathcal{P} \to \mathcal{E}$	$4\mathbf{M}+4\mathbf{S} \approx 7.2\mathbf{M}$
Tripling	$\mathcal{P} \to \mathcal{P}$	$9\mathbf{M}+3\mathbf{S} \approx 11.4\mathbf{M}$
Tripling	$\mathcal{P} \to \mathcal{E}$	$11\mathbf{M}+3\mathbf{S} \approx 13.4\mathbf{M}$
Doubling + Mixed Addition	$\mathcal{P} \to \mathcal{E}$; $\mathcal{E} + \mathcal{A} \to \mathcal{P}$	$10\mathbf{M}+4\mathbf{S} \approx 13.2\mathbf{M}$
Doubling + Addition	$\mathcal{P} \to \mathcal{E}$; $\mathcal{E} + \mathcal{E} \to \mathcal{P}$	$11\mathbf{M}+4\mathbf{S} \approx 14.2\mathbf{M}$
Tripling + Mixed Addition	$\mathcal{P} \to \mathcal{E}$; $\mathcal{E} + \mathcal{A} \to \mathcal{P}$	$17\mathbf{M}+3\mathbf{S} \approx 19.4\mathbf{M}$
Tripling + Addition	$\mathcal{P} \to \mathcal{E}$; $\mathcal{E} + \mathcal{E} \to \mathcal{P}$	$18\mathbf{M}+3\mathbf{S} \approx 20.4\mathbf{M}$

Note: We use symbols $\mathcal{A}$ for (extended) affine coordinates $(X : Y : 1 : T)$; $\mathcal{P}$ for projective coordinates $(X : Y : Z)$; and $\mathcal{E}$ for extended coordinates $(X : Y : Z : T)$.

A double-base chain represents a computation of various multiples nP of a group element P. This computation starts from $0P = 0$, converts each "$\times 2{+}c$" into a doubling $Q \mapsto 2Q$ followed by an addition $Q \mapsto Q + cP$, and converts each "$\times 3{+}c$" into a tripling $Q \mapsto 3Q$ followed by an addition $Q \mapsto Q + cP$, after an initial computation of all relevant multiples cP. For example, the chain

$$(\text{“}\times 2{+}1\text{”}, \text{“}\times 3{+}0\text{”}, \text{“}\times 3{+}0\text{”}, \text{“}\times 2{+}1\text{”}, \text{“}\times 2{+}0\text{”})$$

computes successively $0P, 1P, 3P, 9P, 19P, 38P$.

Formally, given a double-base chain $(o_1, o_2, \ldots, o_\ell)$, define a sequence of integers $(n_0, n_1, n_2, \ldots, n_\ell)$ as follows: $n_0 = 0$; if $o_i =$ "$\times 2{+}c$" then $n_i = 2n_{i-1} + c$; if $o_i =$ "$\times 3{+}c$" then $n_i = 3n_{i-1} + c$. This is the sequence of **intermediate results** for the chain, and the chain is a **chain for** n_ℓ. Evidently one can compute $n_\ell P$ from $0P$ using one doubling and one addition of cP for each "$\times 2{+}c$" in the chain, and one tripling and one addition of cP for each "$\times 3{+}c$" in the chain. Note that the sequence of intermediate results does not determine the chain, and does not even determine the cost of the chain.

3.2. Restrictions on additions. Several variations in the definition of a double-base chain appear in the literature. Often the differences are not made explicit. We now introduce terminology to describe these variations.

Some definitions allow double-base chains to carry out two additions in a row, with no intervening doublings or triplings. Our double-base chains are **reduced**, in the sense that "$+c$", "$+d$" is merged into "$+(c + d)$".

Obviously some limit needs to be placed on the set of c for the concept of double-base chains to be meaningful: for example, the double-base chain ("$\times 2{+}1$", "$\times 2{+}314157$") computes $314159P$ with two additions and an intermediate doubling but begs the question of how the summand $314157P$ was computed. Some papers require "$+c$" to have $c \in \{-1, 0, 1\}$, while other papers allow $c \in S$ for a larger set S of integers.

We consider the general case, and define an S**-chain** as a chain for which each "$+c$" has $c \in S$. We require the set S here to contain 0. We focus on sets S for which it is easy to see the cost of computing cP for all $c \in S$, such as the set $S = \{-\omega, \ldots, -2, -1, 0, 1, 2, \ldots, \omega\}$. Subtracting cP is as easy as adding cP for elliptic curves (in typical curve shapes), so we focus on sets S that are closed under negation, but our framework also allows nonnegative sets S; this means that the distinction between addition chains and addition-subtraction chains is subsumed by the distinction between different sets S.

A double-base chain $(o_1, o_2, \ldots, o_\ell)$ is **increasing** if the sequence of intermediate results $(n_0, n_1, n_2, \ldots, n_\ell)$ has $n_0 < n_1 < n_2 < \cdots < n_\ell$. For example, ("×2+5", "×2+ − 1") is increasing since $0 < 5 < 9$; but ("×2+1", "×2+ − 1" is not increasing. Any $n_i > -\min S$ (with $i < \ell$) automatically has $n_{i+1} > n_i$, so allowing non-increasing chains for positive integers n cannot affect anything beyond initial computations of integers bounded by $-\min S$.

A double-base chain is **greedy** if each intermediate result that is a multiple of 2 or 3 (or both) is obtained by either "×2+0" or "×3+0". This is a more serious limitation on the set of available chains.

3.3. The DAG. Fix a finite set S of integers with $0 \in S$. Define an infinite directed graph D as follows. There is a node n for each nonnegative integer n. For each $c \in S$ and each nonnegative integer n, there is an edge $2n + c \to n$ with label "×2+c" if $2n + c > n$, and there is an edge $3n + c \to n$ with label "×3+c" if $3n + c > n$.

Each edge points from a larger nonnegative integer to a smaller nonnegative integer, so this graph D is acyclic, and the set of nodes reachable from any particular n is finite. Theorem 1 states that this set forms a directed acyclic graph containing at most $O((\log n)^2)$ nodes for any particular S.

Theorem 1. *Assume that $S \subseteq \{-\omega, \ldots, -1, 0, 1, \ldots, \omega\}$. Let n be a positive integer. Then there are at most $(2\omega + 1)(\lfloor \log_2 n + 1 \rfloor \lfloor \log_3 n + 1 \rfloor + 1)$ nodes in D reachable from n.*

Proof. First step: Show that each node v reachable in exactly s steps from n has the form $n/(2^a 3^b) + d$ for some integers a, b and some rational number d with $a \geq 0$, $b \geq 0$, $|d| \leq \omega$, and $a + b = s$.

Induct on s. If $s = 0$ then v must be n, so $v = n/(2^a 3^b) + d$ with $(a, b, d) = (0, 0, 0)$. If $s \geq 1$ then there is an edge $u \to v$ for some node u reachable in exactly $s - 1$ steps from n. By the inductive hypothesis, u has the form $n/(2^a 3^b) + d$ with $a \geq 0$, $b \geq 0$, $|d| \leq \omega$, and $a + b = s - 1$.

If the edge has label "×2+c" then $u = 2v + c$ so $v = (u - c)/2 = n/(2^{a+1} 3^b) + (d - c)/2$; and $c \in S$ so $|c| \leq \omega$ so $|(d - c)/2| \leq \omega$. Hence v has the correct form. Similarly, if the edge has label "×3+c" then $u = 3v + c$ so $v = (u - c)/3 = n/(2^a 3^{b+1}) + (d - c)/3$, and $|(d - c)/3| \leq (2/3)\omega \leq \omega$. This completes the proof of the first step.

Second step: We count the nodes v with $a \leq \log_2 n$ and $b \leq \log_3 n$. There are at most $\lfloor \log_2 n + 1 \rfloor$ possibilities for a, and at most $\lfloor \log_3 n + 1 \rfloor$ possibilities

for b. Any particular (a, b) limits v to the interval $[n/(2^a 3^b) - \omega, n/(2^a 3^b) + \omega]$, which contains at most $2\omega + 1$ integers.

Third step: We count the remaining nodes v. Here $a > \log_2 n$ so $2^a > n$, or $b > \log_3 n$ so $3^b > n$, or both; in any case $n/(2^a 3^b) < 1$ so $|v| < 1 + |d| < 1 + \omega$; i.e., $v \in \{-\omega, \ldots, \omega\}$. This limits v to at most $2\omega + 1$ possibilities across all of the possible pairs (a, b). □

Theorem 2 states a straightforward correspondence between the set of paths in D from n to 0 and the set of increasing double-base S-chains for n. The correspondence simply reads the edge labels in reverse order.

Theorem 2. *Let n be a nonnegative integer. If $(e_\ell, \ldots, e_1)$ is a path from n to 0 in D with labels $(o_\ell, \ldots, o_1)$ then $(o_1, \ldots, o_\ell)$ is an increasing double-base S-chain for n. Conversely, every increasing double-base S-chain for n has this form.*

Proof. Each o_i is an edge label in D, which by definition of D has the form "$\times 2 + c$" or "$\times 3 + c$", so $C = (o_1, \ldots, o_\ell)$ is a double-base chain; what remains is to show that it is an increasing S-chain for n.

Specifically, say $o_i =$ "$\times t_i + c_i$". Define $(n_0, n_1, \ldots, n_\ell)$ as the corresponding sequence of intermediate results; then $n_0 = 0$, and $n_i = t_i n_{i-1} + c_i$ for $i \in \{1, \ldots, \ell\}$.

We now show by induction on i that e_i is an edge from n_i to n_{i-1}. If $i = 1$ then by hypothesis of the theorem $e_i = e_1$ is an edge to $0 = n_0 = n_{i-1}$. If $i > 1$ then e_{i-1} is an edge from n_{i-1} by the inductive hypothesis so e_i is an edge to n_{i-1}. For any i, e_i is an edge to n_{i-1}. The label $o_i =$ "$\times t_i + c_i$" then implies that e_i is an edge from $t_i n_{i-1} + c_i$, i.e., from n_i, as claimed.

In particular, e_ℓ is an edge from n_ℓ, but also e_ℓ is an edge from n by hypothesis of the theorem, so $n = n_\ell$. Hence C is a chain for n. Furthermore, C is an S-chain since each $c_i \in S$ by definition of D, and C is increasing since each edge decreases by definition of D.

Conversely, take any increasing double-base S-chain C for n. Write C as $(o_1, \ldots, o_\ell)$, write o_i as "$\times t_i + c_i$", and define $(n_0, n_1, \ldots, n_\ell)$ as the corresponding sequence of intermediate results. Then D has an edge e_i with label o_i from n_i to n_{i-1}, so $(e_\ell, \ldots, e_1)$ is a path from $n_\ell = n$ to 0 in D. □

3.4. Chain cost and path cost. Theorem 2 suggests the following simple strategy to find an optimal increasing double-base S-chain for n: use Dijkstra's algorithm to find a shortest path from n to 0 in D. This takes time polynomial in $\log n$ if ω is polynomially bounded: the number of nodes visited is polynomial by Theorem 1, and it is easy to see that constructing all of the outgoing edges from a node takes polynomial time.

Dijkstra's algorithm requires each edge to be assigned a positive weight, and requires the cost of a path to be defined as the sum of edge weights. This strategy therefore requires the cost of a chain to be defined "locally": the cost of doubling nP and adding cP must not depend on any context other than (n, c),

and similarly for tripling. This limitation is not a problem for, e.g., accounting for free additions of $0P$; accounting for a free initial doubling of $0P$; accounting for lower-cost addition of P, i.e., $c = 1$, if P is kept in affine coordinates while other multiples of P are kept in projective or extended coordinates; accounting for the cost of tripling into extended coordinates (see Section 2); etc.

One can also handle non-increasing chains by allowing negative integers and dropping the conditions $2n + c > n$ and $3n + c > n$. This allows cycles in D, not a problem for typical breadth-first shortest-path algorithms building a shortest-path tree; see, e.g., [10]. For simplicity we focus on acyclic graphs in this paper.

3.5. Example. Figure 1 shows the subset of D reachable from $n = 17$ when $S = \{-1, 0, 1\}$. We omit the 0 node from the figure. We replace the remaining edge labels with costs according to Table 1, namely, 11.4 for tripling, 6.2 for doubling, 7 for mixed addition.

The choice $S = \{-1, 0, 1\}$ means that each even node t has two outgoing edges: one for $t/2$ and one for $(t + c)/3$ for a unique c, because exactly one of $t, t+1, t-1$ is divisible by 3. Each odd node t has three outgoing edges: one for $(t - 1)/2$, one for $(t + 1)/2$, and one for $(t + c)/3$ for a unique c. For example, 8 is reached from 17 as $(17 - 1)/2$ costing one addition and one doubling; 6 is reached as $(17+1)/3$ costing one addition and one tripling. There are two edges between 5 and 2, namely, one corresponding to $2 = (5-1)/2$ and one (obviously worse) corresponding to $2 = (5 + 1)/3$.

3.6. The DAG approach vs. the tree approach. The Doche–Habsieger tree-based algorithm summarized in Section 1 considers only some special greedy chains: when it sees an even node it insists on dividing by 2, prohibiting nontrivial additions, and similarly when it sees a multiple of 3 it insists on dividing by 3; when it sees a number that is neither a multiple of 2 or of 3 it uses an addition. This reduces the number of nodes by roughly a factor 3, but we have found many examples where the best greedy chain is more expensive than the best chain.

Each possible intermediate result appears exactly once in the DAG in this section, but can appear at many different tree levels in the tree-based algorithm. The tree has $\Theta(\log n)$ levels, and a simple heuristic suggests that an intermediate result will, on average, appear on $\Theta((\log n)^{0.5})$ of these levels. The tree-based algorithm repeatedly considers the same edges out of the same coefficient, while the DAG avoids this redundant work. Pruning the tree reduces the cost of the tree-based algorithm but compromises the quality of the resulting chains.

4 Rectangular DAG-based approach

The DAG that we introduced in Section 3 reduces the target integer n to various smaller integers, each obtained by subtracting an element of S and dividing by 2 or 3. It repeats this process to obtain smaller and smaller integers t, stopping at 0.

In this section we build a slightly more complicated DAG with a three-dimensional structure. Each node in this new DAG has the form (a, b, t), where

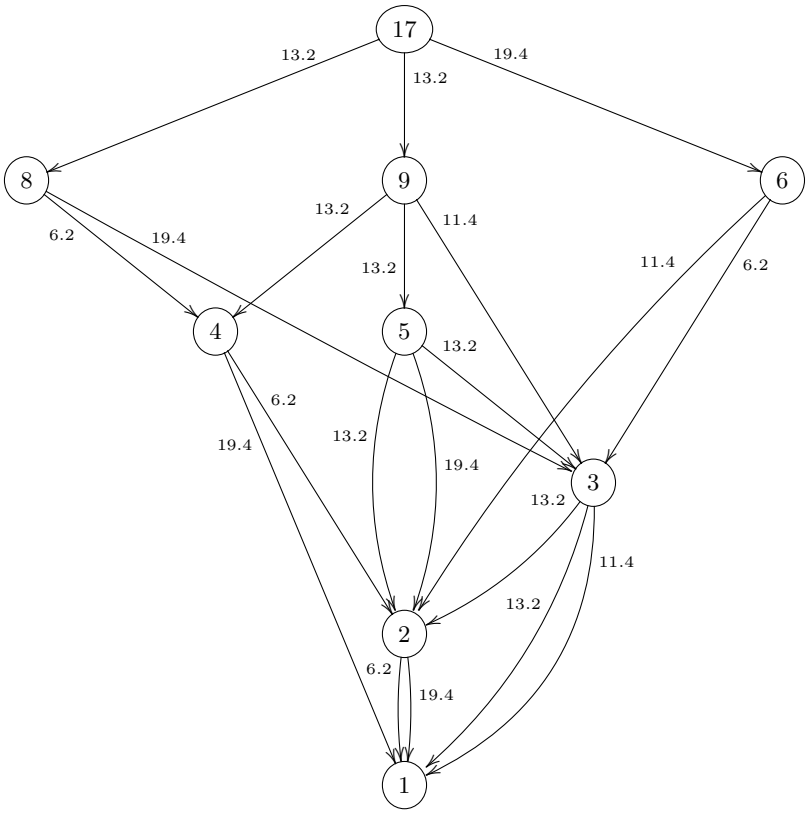


Fig. 1: Example of a DAG for finding double-base chains for $n = 17$

a is the number of doublings used in paths from n to t, and b is the number of triplings used in paths from n to t.

One advantage of this DAG is that it is now very easy to locate nodes in a simple three-dimensional array, rather than incurring the costs of the associative arrays typically used inside Dijkstra's algorithm. The point is that for t between $n/(2^a 3^b) - \omega$ and $n/(2^a 3^b) + \omega$, as in the proof of Theorem 1, we store information about node (a, b, t) at index $(a, b, t - \lfloor n/(2^a 3^b) - \omega \rfloor)$ in the array.

Another advantage of this DAG is that we no longer incur the costs of maintaining a list of nodes to visit inside Dijkstra's algorithm. Define the "position" of the node (a, b, t) as (a, b): then each doubling edge is from position (a, b) to position $(a + 1, b)$, and each tripling edge is from position (a, b) to position $(a, b+1)$. There are many easy ways to topologically sort the nodes, for example by sweeping through positions in increasing order of $a + b$.

The disadvantage of this DAG is that a single t can now appear multiple times at different positions (a, b). However, this disadvantage is limited: it can occur only when there are near-collisions among the values $n/(2^a 3^b)$, something

that is quite rare except for the extreme case of small values. We show that the DAG has $(\log n)^{2+o(1)}$ nodes (assuming $\omega \in (\log n)^{o(1)}$), like the DAG in Section 3, and thus a total of $(\log n)^{3+o(1)}$ bits in all of the nodes.

We obtain an algorithm to find shortest paths in this DAG, and thus optimal double-base chains, using time just $(\log n)^{3+o(1)}$. Section 5 explains how to do even better, reducing the time to $(\log n)^{2.5+o(1)}$ by using reduced representatives for almost all of the integers t.

4.1. The three-dimensional DAG. Fix a positive integer ω. Fix a subset $S \subseteq \{-\omega, \ldots, -1, 0, 1, \ldots, \omega\}$ with $0 \in S$. Fix a positive integer n. Define a finite directed acyclic graph R_n as follows.

There is a node (a, b, v) for each integer $a \in \{0, 1, \ldots, \lfloor \log_2 n \rfloor + 1\}$, each integer $b \in \{0, 1, \ldots, \lfloor \log_3 n \rfloor + 1\}$, and each integer v within ω of $n/(2^a 3^b)$. Note that not all nodes will be reachable from n in general.

If (a, b, v) and $(a+1, b, u)$ are nodes, $v > u$, and $v = 2u + c$ with $c \in S$, then there is an edge $(a, b, v) \to (a+1, b, u)$ with label "$\times 2{+}c$".

Similarly, if (a, b, v) and $(a, b+1, u)$ are nodes, $v > u$, and $v = 3u + c$ with $c \in S$, then there is an edge $(a, b, v) \to (a, b+1, u)$ with label "$\times 3{+}c$".

Theorem 3, analogously to Theorem 1, says that R_n does not have many nodes. Theorem 4, analogously to Theorem 2, says that paths in R_n from $(0, 0, n)$ to $(\ldots, \ldots, 0)$ correspond to double-base chains for n.

Theorem 3. *There are at most* $(2\omega+1)(\lfloor \log_2 n + 2 \rfloor \lfloor \log_3 n + 2 \rfloor)$ *nodes in* R_n.

Proof. There are $\lfloor \log_2 n + 2 \rfloor$ choices of a and $\lfloor \log_3 n + 2 \rfloor$ choices of b. For each (a, b), there are at most $2\omega + 1$ integers v within ω of $n/(2^a 3^b)$. □

Theorem 4. *Let n be a positive integer. If $(e_\ell, \ldots, e_1)$ is a path from $(0, 0, n)$ to $(a, b, 0)$ in R_n with labels $(o_\ell, \ldots, o_1)$ then $(o_1, \ldots, o_\ell)$ is an increasing double-base S-chain for n with at most $\lfloor \log_2 n \rfloor + 1$ doublings and at most $\lfloor \log_3 n \rfloor + 1$ triplings. Conversely, every increasing double-base S-chain for n with at most $\lfloor \log_2 n \rfloor + 1$ doublings and at most $\lfloor \log_3 n \rfloor + 1$ triplings has this form.*

Proof. Given a path from $(0, 0, n)$ to $(a, b, 0)$ in R_n, remove the first two components of each node to obtain a path from n to 0 in D. This path has the same labels $(o_\ell, \ldots, o_1)$, so $(o_1, \ldots, o_\ell)$ is an increasing double-base S-chain for n by Theorem 2. It has at most $\lfloor \log_2 n \rfloor + 1$ doublings since each doubling increases the first component within $\{0, 1, \ldots, \lfloor \log_2 n \rfloor + 1\}$, and similarly has at most $\lfloor \log_3 n \rfloor + 1$ triplings.

Conversely, given an increasing double-base S-chain for n, construct the corresponding path in D by Theorem 2. Insert two extra components into each node to count the number of doublings and triplings. If the chain has at most $\lfloor \log_2 n \rfloor + 1$ doublings and at most $\lfloor \log_3 n \rfloor + 1$ triplings then these components are contained in $\{0, 1, \ldots, \lfloor \log_2 n \rfloor + 1\}$ and $\{0, 1, \ldots, \lfloor \log_3 n \rfloor + 1\}$ respectively, producing a path in R_n from $(0, 0, n)$ to $(a, b, 0)$. □

Figure 2 illustrates R_{17}. Only nodes reachable from $(0, 0, n)$ are included, nodes $(\ldots, \ldots, 0)$ are omitted. The rectangular plane shows positions (a, b). Full

description of how to use the rectangular DAG to find double-base chains for $n = 17$ can be found in Appendix A.

4.2. Cost analysis. We now analyze the performance of shortest-path computation using this DAG, taking account of the cost of handling multiprecision integers such as n. A Python script suitable for carrying out experiments appears in Appendix B.

Recall that information about node (a, b, t) is stored in a three-dimensional array at index $(a, b, t - \lfloor n/(2^a 3^b) - \omega \rfloor)$. We keep a table of these base values $\lfloor n/(2^a 3^b) - \omega \rfloor$. Building this table takes one division by 2 or 3 at each position (a, b). Each division input has $O(\log n)$ bits, and dividing by 2 or 3 takes time linear in the number of bits; the total time is $O((\log n)^3)$.

The information stored for (a, b, t) is the minimum cost of a path from $(0, 0, n)$ to (a, b, t). We optionally store the nearest edge label for a minimum-cost path, to simplify output of the path, but this can also be efficiently reconstructed afterwards from the table of costs.

We sweep through positions (a, b) in topological order, starting from $(0, 0, n)$. At each node (a, b, t) with $t > 0$, for each $s \in S$ with $2 \mid t - s$, we update the cost stored for $(a + 1, b, (t - s)/2)$; and, for each $s \in S$ with $3 \mid t - s$, we update the cost stored for $(a, b + 1, (t - s)/3)$. An easy optimization is to stop with any $t \in S$, rather than just $t = 0$.

There are $O(\omega(\log n)^2)$ nodes, each with $O(\omega)$ outgoing edges, for a total of $O(\omega^2(\log n)^2)$ update steps. Each update step takes time $O(\log n)$, for total time $O(\omega^2(\log n)^3)$.

5 Reduced rectangular DAG-based approach

Recall that the shortest-path computation in Section 4 takes time $(\log n)^{3+o(1)}$, assuming $\omega \in (\log n)^{o(1)}$. This section explains how to reduce the exponent from $3 + o(1)$ to $2.5 + o(1)$.

5.1. Multiprecision arithmetic as a bottleneck. There are $O(\omega(\log n)^2)$ nodes in the graph R_n; there are $O(\omega)$ edges out of each node; there are $O(1)$ operations at each edge. The reason for an extra factor of $\log n$ in the time complexity is that the operations are multiprecision arithmetic operations on integers that often have as many bits as n. We now look more closely at where such large integers are actually used.

Writing down a position (a, b) takes only $O(\log \log n)$ bits. Writing down a cost also takes only $O(\log \log n)$ bits. We are assuming here, as in most analyses of Dijkstra's algorithm, that edge weights are integers in $O(1)$; for example, each edge weight in Figure 1 is (aside from a scaling by 0.1) one of the three integers $62, 132, 184$, so any s-step path has weight at most $184s$.

Writing down an element of S, or an array index $t - \lfloor n/(2^a 3^b) - \omega \rfloor$, takes only $O(\log(2\omega + 1))$ bits. However, both t and the precomputed $\lfloor n/(2^a 3^b) - \omega \rfloor$ are usually much larger, on average $\Theta(\log n)$ bits. Several arithmetic operations are forced to work with these large integers: for example, writing down the first

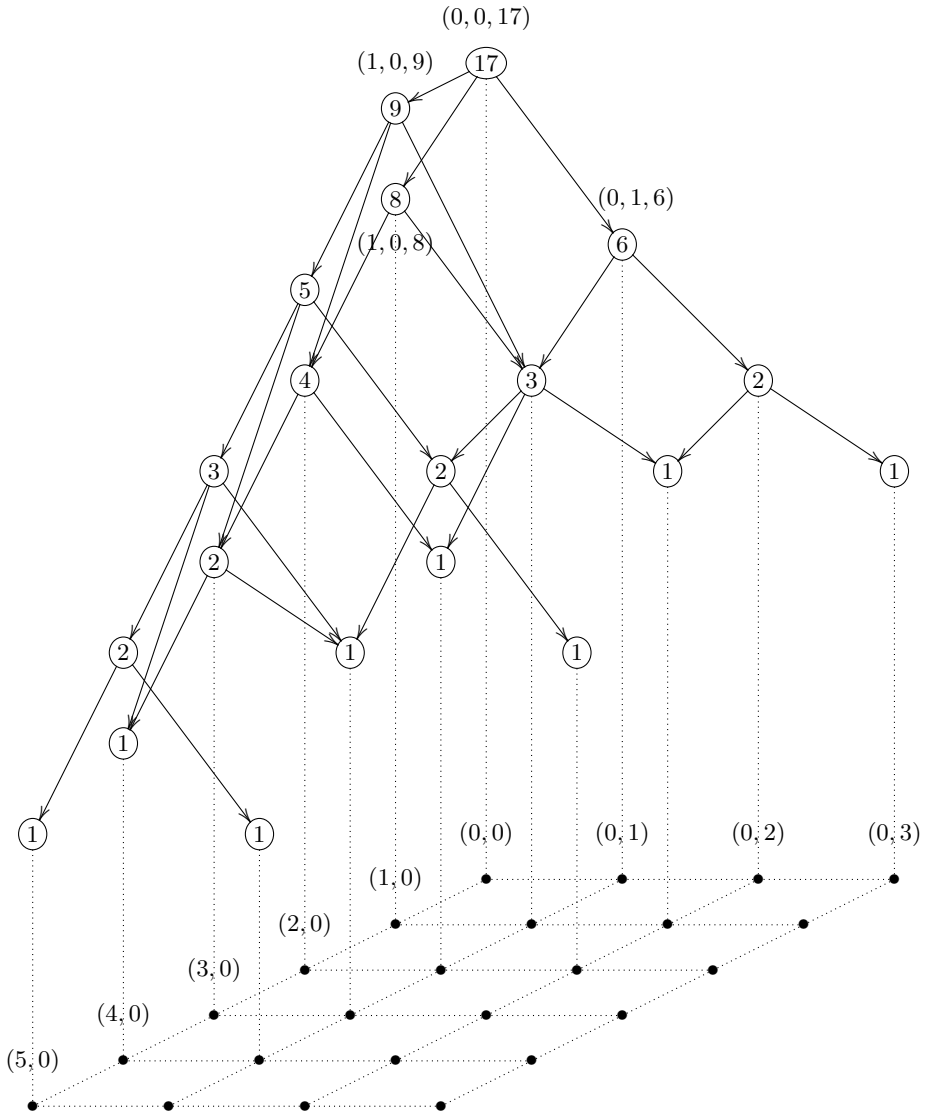


Fig. 2: Example of a 3D-DAG for finding double-base chains for $n = 17$

t at a position (a, b), computing $t - s$ for the first $s \in S$, and checking whether $t - s$ is divisible by 3.

5.2. Reduced representatives for large numbers. This section saves time by replacing almost all of the integers t with reduced representatives. Each of these representatives occupies only $(\log n)^{0.5+o(1)}$ bits, for a total of $(\log n)^{2.5+o(1)}$

bits. There are occasional "boundary nodes" that each use $(\log n)^{1+o(1)}$ bits, but there are only $(\log n)^{1.5+o(1)}$ of these nodes, again a total of $(\log n)^{2.5+o(1)}$ bits. Arithmetic becomes correspondingly less expensive.

Specifically, (a, b, t) is a **boundary node** if a is a multiple of α or b is a multiple of β or both. Here α and β are positive integers, parameters for the algorithm. For example, the solid lines in Figure 3 connect the boundary nodes for $\alpha = \beta = 2$. We will choose α and β as $(\log n)^{0.5+o(1)}$, giving the above-mentioned $(\log n)^{1.5+o(1)}$ boundary nodes.

These boundaries partition the DAG into **subgraphs**. Each subgraph has $O(\omega\alpha\beta)$ nodes at $(\alpha + 1)(\beta + 1)$ positions covering an area $\alpha \times \beta$ within the space of positions. Specifically, subgraph (q, r) covers all positions (a, b) where $a \in \{q\alpha, q\alpha + 1, \ldots, q\alpha + \alpha\}$ and $b \in \{r\beta, r\beta + 1, \ldots, r\beta + \beta\}$. Some subgraphs are truncated because they stretch beyond the maximum positions (a, b) allowed in R_n; one can, if desired, avoid this truncation by redefining R_n so that the maximum a is a multiple of α and the maximum b is a multiple of β.

To save time in handling a node (a, b, t) at the top position $(a, b) = (q\alpha, r\beta)$ in subgraph (q, r), we work with the remainder $t \bmod 2^\alpha 3^\beta$. More generally, to save time in handling a node (a, b, t) at position $(a, b) = (q\alpha + i, r\beta + j)$ in the subgraph, we work with the remainder $t \bmod 2^{\alpha-i}3^{\beta-j}$. Each of these remainders is below $2^\alpha 3^\beta$, and therefore occupies only $(\log n)^{0.5+o(1)}$ bits when α and β are $(\log n)^{0.5+o(1)}$.

The critical point here is that the remainder $t \bmod 2^\alpha 3^\beta$ is enough information to see whether $t - s$ is divisible by 2 or 3. This remainder also reveals the next remainder $((t - s)/2) \bmod 2^{\alpha-1}3^\beta$ if $t - s$ is divisible by 2, and $((t - s)/3) \bmod 2^\alpha 3^{\beta-1}$ if $t - s$ is divisible by 3. There is no need to take a detour through writing down the much larger integer $t - s$; we instead carry out a computation on a much smaller number $(t - s) \bmod 2^\alpha 3^\beta$. Continuing in the same way allows as many as α divisions by 2 and as many as β divisions by 3, reaching the bottom boundary of the subgraph. We reconstruct the original nodes at this boundary and then continue to the next subgraph.

Note that remainders do *not* easily allow testing whether t is 0. These tests are used in Section 4, but only as a constant-factor optimization to save time in enumerating some cost-0 edges. What is important is testing divisibility of $t - s$ by 2 or 3. Similarly, there is no reason to limit attention to increasing chains.

We have found it simplest to allow negative remainders inside each subgraph, skipping all intermediate mod operations. For example, if $t \bmod 2^\alpha 3^\beta$ happens to be 0 and $s = 2$, then we write down $(0 - 2)/2 = -1$ rather than $2^{\alpha-1}3^\beta - 1$. The integer operations inside each subgraph then consist of divisions by 2, divisions by 3, and subtractions of elements of S. To reconstruct a node at the bottom boundary of the subgraph, we first merge the sequence of operations into a single subtract-and-divide operation, and then apply this subtract-and-divide operation to the top node t. For example, we merge "subtract 1, divide by 2, subtract 1, divide by 2" into "subtract 3, divide by 4" and then directly compute $(t - 3)/4$, without computing the large intermediate result $(t - 1)/2$. Note that arbitrary divisions of $O(\log n)$-bit numbers take time $(\log n)^{1+o(1)}$, as shown by

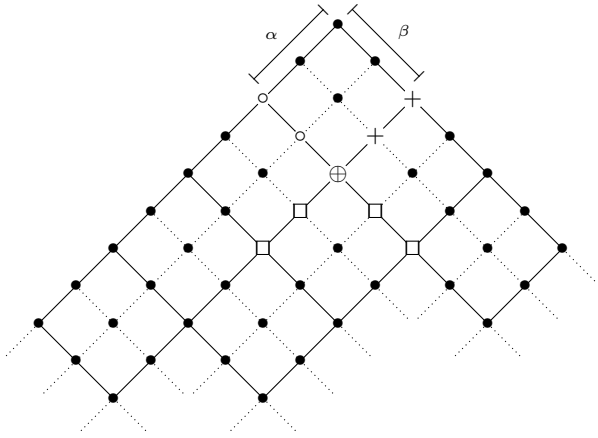


Fig. 3: Example of graph division where $\alpha = \beta = 2$.

Cook in [9, pages 81–86], using Simpson's division method from [21, page 81] on top of Toom's multiplication method from [22].

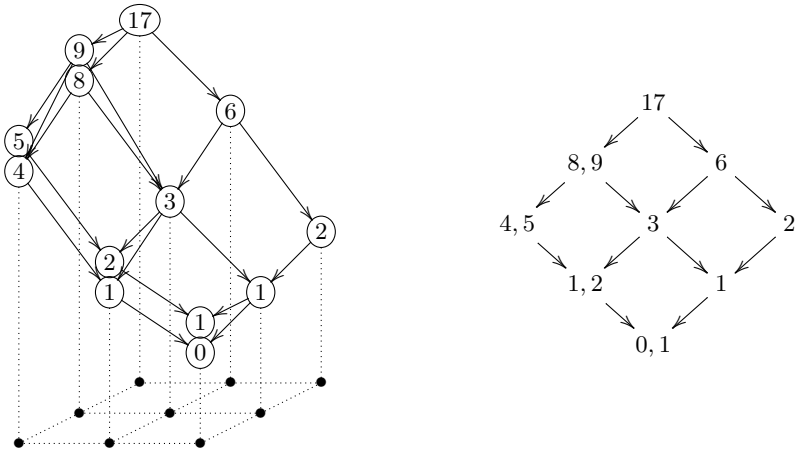
In the case $\alpha = \beta = 2$ depicted in Figure 3, we begin by computing the small integer $n \bmod 2^2 3^2$ at the top of the first subgraph. This is enough information to reconstruct the pattern of edges involving as many as $\alpha = 2$ divisions by 2 and as many as $\beta = 2$ divisions by 3. The boundary nodes involving 2 divisions by 2 are marked $\circ$ and $\oplus$. These boundary nodes have values close to $n/2^2$, $n/(2^2 3)$, and $n/(2^2 3^2)$; we reconstruct these values, reduce them again, and continue with the next subgraph down and to the left. Similarly, the boundary nodes involving 2 divisions by 3 are marked $+$ and $\oplus$, with values close to $n/3^2$, $n/(2 \cdot 3^2)$, and $n/(2^2 3^2)$; we reconstruct these values, reduce them again, and continue with the next subgraph down and to the right. The fourth subgraph similarly begins by reconstructing its boundary nodes, marked $\oplus$ and $\square$.

A Python script suitable for experiments appears in Appendix C.

Example 1. Consider the problem of computing double-base chains for $n = 917$, using $S = \{-1, 0, 1\}$. We use the same cost model as in previous examples. We take $\alpha = \beta = 2$.

Since $917 \equiv 17 \pmod{2^2 3^2}$, we compute a subgraph of size 2×2 starting with 17. We shall refer to this subgraph as Subgraph17. The result of this computation is depicted using a 3-dimensional graph in Figure 4a and using a projective view in Figure 4b.

To reconstruct nodes from the original graph at the bottom boundary of the subgraph, we start at the root, or more generally at any known node in the original graph, and then follow a path to the boundary. Consider, for example, the node $(2, 0, 4)$ in Figure 4a. This 4 was (optimally) computed as $(17-1)/2/2$, i.e., as $(17-1)/4$, so we compute $(917-1)/4 = 229$, obtaining the corresponding node $(2, 0, 229)$ in the original graph. Similarly, the 5 in $(2, 0, 5)$ was (optimally)



(a) Three-dimensional view of Subgraph17 (b) Projection of Subgraph17

Fig. 4: Example of a subgraph of $17 \equiv 917 \pmod{2^2 3^2}$

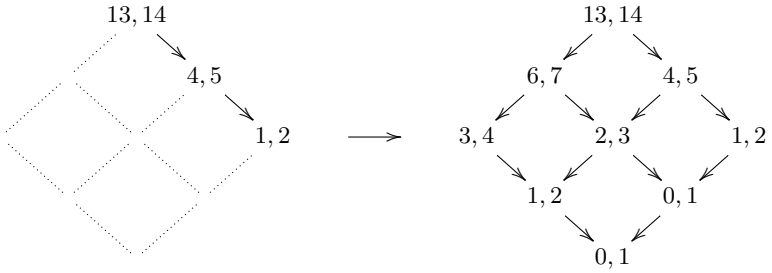


Fig. 5: Example of right boundary and subgraph computation

computed as $((17+1)/2+1)/2 = (17+3)/4$, so we compute $(917+3)/4 = 230$, obtaining the corresponding node $(2, 0, 230)$ in the original graph. There are several ways to speed this up further, such as computing 230 as $229+5-4$, but computing each node separately is adequate for our $(\log n)^{2.5+o(1)}$ result.

Once we have recomputed all nodes in the original graph at the bottom boundary of the subgraph, we move to the next subgraph, for example replacing 229 with $229 \bmod 2^2 3^2 = 13$ and replacing 230 with $230 \bmod 2^2 3^2 = 14$. Figure 5 shows the values provided as input (left) and obtained as output (right) in the next subgraph to the bottom left. Similarly, Figure 6 shows the values provided as input (left) and obtained as output (right) in the next subgraph to the bottom right.

These procedures of computing intermediate and boundary nodes repeat, eventually covering all subgraphs of the original graph, although one can save a constant factor by skipping computations of some subgraphs. For example,

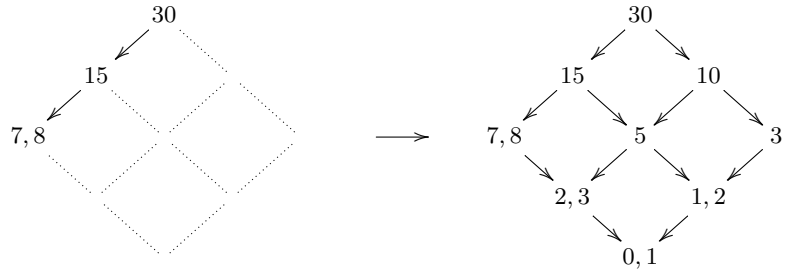


Fig. 6: Example of left boundary and subgraph computation

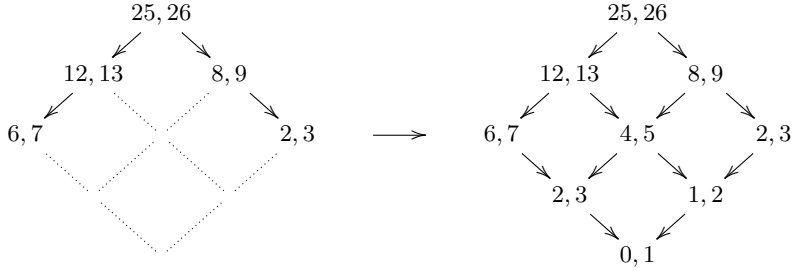


Fig. 7: Example of both boundaries and subgraph computation

Figure 7 shows the fourth subgraph computation. Reconstruction shows that the bottom $0, 1$ in this subgraph corresponds to $0, 1$ in the original graph, so at this point we have found complete chains to compute 917, and there is no need to explore further subgraphs below these $0, 1$ nodes.

6 Double-base double-scalar multiplication

Our graph-based approaches to finding double-base chains for single-scalar multiplication can easily be extended to finding double-base chains for double-scalar multiplication. In this section, we explain how to extend the reduced rectangular DAG-based approach to finding double-base chains for double-scalar multiplication.

Recall that inputs for double-scalar multiplication are two scalars (which we will call n_1 and n_2), and two points on an elliptic curve (which we will call P and Q). Given these inputs, the algorithm returns a double-base chain computing $n_1P + n_2Q$, where n_1 and n_2 are expressed as $(n_1, n_2) = \sum_{i=1}^{\ell}(c_i, d_i)2^{a_i}3^{b_i}$ and pairs (c_i, d_i) are chosen from a precomputed set S.

Note that the precomputed set S for double-scalar multiplication is defined differently from the single-scalar case. Recall that in the latter case, each member in the set S is a *single* integer which is the coefficient in a double-base chain

representation as defined in Section 1.1. In the former case, each member in the set S is a *pair* of integers (c_i, d_i) where c_i and d_i are coefficients in a double-base chain representation of n_1 and n_2 respectively.

If $d_i = 0$ this means that the precomputed point depends only on P, e.g., $(2,0)$, $(3,0)$, $(4,0)$, $(5,0)$ correspond to $2P$, $3P$, $4P$, $5P$ respectively. Similarly, if $c_i = 0$ this means that the precomputed point depends only on Q, e.g., $(0,2)$, $(0,3)$, $(0,4)$, $(0,5)$ correspond to $2Q$, $3Q$, $4Q$, $5Q$ respectively. If both c_i and d_i are nonzero, this means that the precomputed points depend on both P and Q, e.g., $(1,1)$, $(1,-1)$, $(2,1)$, $(2,-1)$, $(1,2)$, $(1,-2)$ correspond to $P+Q$, $P-Q$, $2P+Q$, $2P-Q$, $P+2Q$, $P-2Q$ respectively.

For example, $S = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$ or equivalently $S = \{(0,0),(1,0),(-1,0),(0,1),(0,-1),(1,1),(-1,-1),(1,-1),(-1,1)\}$ means that we allow addition of 0 (no addition), $P, -P, Q, -Q, (P+Q), (-P-Q), (P-Q)$, and $(-P+Q)$ in double-base chains. Note that in this case, S requires space to store 4 points (since negation can be computed on the fly) but it costs only 2 point additions for precomputation, namely, $P+Q$ and $P-Q$.

The algorithm for generating double-base chains for double-scalar multiplication starts by computing $t_1 \equiv n_1 \pmod{2^\alpha 3^\beta}$ and $t_2 \equiv n_2 \pmod{2^\alpha 3^\beta}$, and then initializes the root node $t_{0,0}$ to the pairs (t_1, t_2), i.e., the reduced representation of (n_1, n_2). For each pair (t_r, t_s) at each node $t_{i,j}$ where $0 \le i \le \alpha$ and $0 \le j \le \beta$, we follow a subgraph computation similar to that explained in Section 5, namely,

- If $t'_1 = (t_r - c)/2$ and $t'_2 = (t_s - d)/2$ are integers, where (c,d) is from the set S, then insert this pair (t'_1, t'_2) to the node $t_{i+1,j}$ if it does not exist yet or update the cost if cheaper.
- If $t''_1 = (t_r - c)/3$ and $t''_2 = (t_s - d)/3$ are integers, where (c,d) is from the set S, then insert this pair (t''_1, t''_2) to the node $t_{i,j+1}$ if it does not exist yet or update the cost if cheaper.

Once all pairs (t_r, t_s) at all nodes $t_{i,j}$ are visited, apply the boundary node computation. The algorithm continues by repeating subgraph and boundary node computation until pairs $(c,d) \in S$ are reached in a subgraph for which the values of (t_r, t_s) at the root node are less than $2^\alpha 3^\beta$ as integers, i.e., without performing modular reduction (see example below). Notice that the concepts of reduced representative, boundary nodes, and subgraphs are the same as explained in Section 5.

Example 2. Consider the problem of computing double-base chains for $n_1 = 83$ and $n_2 = 125$, using $S = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$, and taking $\alpha = \beta = 2$. We use the same cost model as in previous examples.

Since $83 \equiv 11 \pmod{2^2 3^2}$ and $125 \equiv 17 \pmod{2^2 3^2}$, we compute a subgraph of size 2×2 starting with $(11, 17)$. We shall refer to this subgraph as Subgraph1117. Figure 8 depicts the result of this computation.

To compute the next subgraphs, we have to recompute all pairs of boundary nodes as if they were computed from the original integers (no modular reduction by $2^2 3^2$). For example, the pair $(2,4)$ of the leftmost node, are computed as

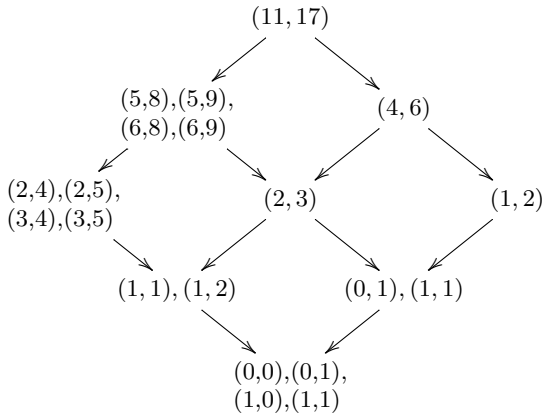


Fig. 8: Computation of Subgraph1117

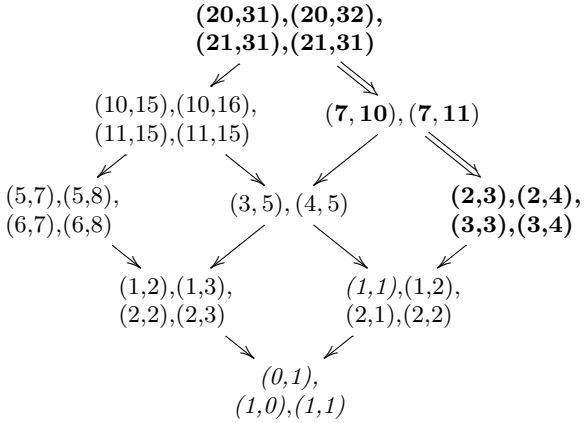


Fig. 9: Computation of Subgraph2031, the bottom left to Subgraph1117. Note that pairs from Subgraph1117 are labeled with bold face and doubled arrows; pairs in S are labeled in italics.

$2 = ((11-1)/2-1)/2$ and $4 = ((17-1)/2)/2$. Therefore, these numbers map to $((83-1)/2-1)/2 = 20$ and $((125-1)/2)/2 = 31$.

Apply similar recomputations for all pairs along the bottom left boundary nodes: (2,5), (3,4), (3,5) map to (20,32), (21,31), (21,32); (1,1), (1,2) map to (7,10), (7,11); and (0,0), (0,1), (1,0), (1,1) map to (2,3), (2,4), (3,3), (3,4) respectively. We shall refer to this subgraph as Subgraph2031. Figure 9 depicts the result of the boundary node recomputation and the subgraph computation of that subgraph.

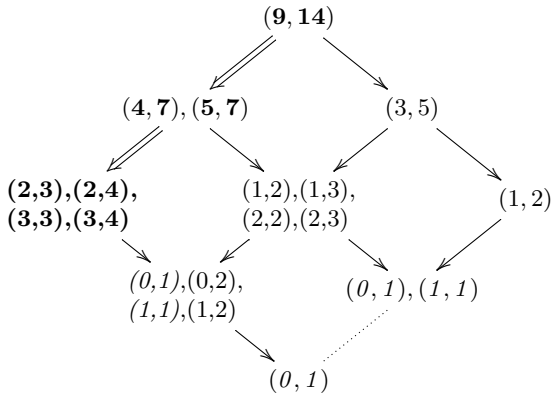


Fig. 10: Computation of Subgraph0914, the bottom right to Subgraph1117. Note that pairs from Subgraph1117 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted line means no computations.

Similar recomputations are also applied to all pairs along the bottom right boundary nodes of Subgraph1117. We shall refer to this subgraph as Subgraph0914. Figure 10 depicts the result of the boundary node recomputation and the subgraph computation of that subgraph.

Notice that we do not need to compute the subgraph underneath Subgraph1117 (i.e., the bottom right of Subgraph2031 and bottom left of Subgraph0914) because (1) the root nodes of Subgraph2031 and Subgraph0914 are already less than $2^2 3^2$, meaning that the reduced representations are the same as without performing modular reduction; (2) both bottom left and bottom right boundaries reach pairs $(c, d) \in S$, i.e., both left and right child nodes of (2,3),(2,4),(3,3),(3,4) reach pairs in S.

We continue recomputing the bottom left boundary of Subgraph2031. We shall refer to this subgraph as Subgraph0507. Figure 11 depicts the result of this computation. We also continue recomputing the bottom right boundary of Subgraph0914. We shall refer to this subgraph as Subgraph0102. Figure 12 depicts the result of this computation. We do not perform further subgraph computation because Subgraph0507 and Subgraph0102 reach pairs $(c, d) \in S$. An overview of subgraphs is depicted in Figure 13.

7 Results and comparison

We conducted several experiments to measure the performance of our algorithms for single-scalar and double-scalar multiplications where in each case we compared to both single-base and double-base algorithms, namely, we considered single-base single-scalar, double-base single-scalar, single-base double-scalar, and double-base double-scalar multiplications. These experiments were designed to

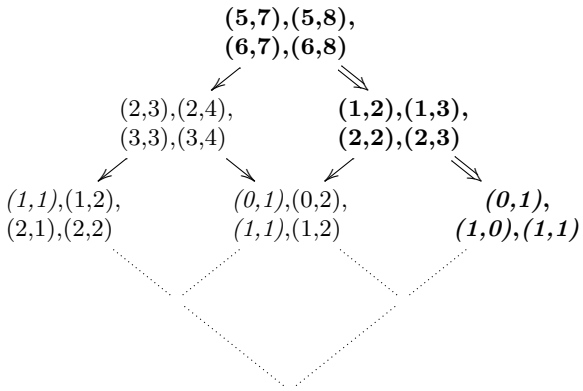


Fig. 11: Computation of Subgraph0507, the bottom left to Subgraph2031. Note that pairs from Subgraph2031 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted lines mean no computations.

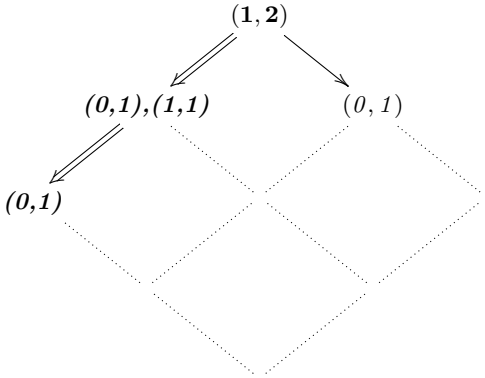


Fig. 12: Computation of Subgraph0102, the bottom right to Subgraph0914. Note that pairs from Subgraph0914 are labeled with bold face and doubled arrows; pairs in S are labeled in italics; dotted lines mean no computations.

separately measure the performance due to the new tripling formulas, the performance due to the graph-based approach to generate double-base chains, and the overall performance of combining these two. Comparisons among various algorithms and previous results are also presented.

In all experiments we used at least 1000 randomly chosen integers between 0 and $2^\ell - 1$. The average number of multiplications observed in these experiments, and the average divided by ℓ, are rounded to a limited number of digits after the decimal point and reported below as "Mults" and "Mults/ℓ". The rounding means that dividing "Mults" by ℓ does not exactly produce "Mults/ℓ".

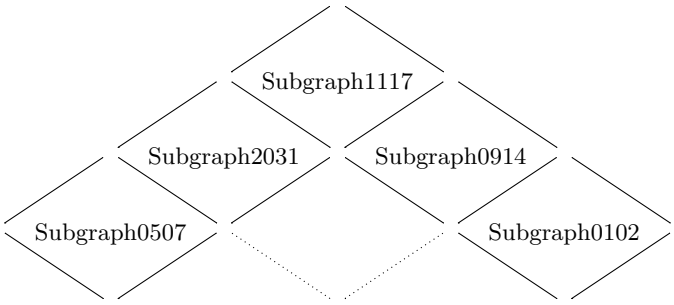


Fig. 13: An overview of subgraphs. Note that dotted lines mean no computation for that subgraph.

7.1. Overall performance. To illustrate the overall performance when using the new tripling formulas together with optimal double-base chains, we performed experiments on many different sets of coefficients focusing on 256-bit scalars. In these experiments, we used the traditional $\mathbf{S} = 0.8\mathbf{M}$ and $\mathbf{M}_a = \mathbf{M}_2 = \mathbf{A} = 0\mathbf{M}$.

We used mixed coordinate systems: projective coordinates as input to doubling and tripling mixed with extended coordinates as input to addition. Each "$\times 2{+}c$" and "$\times 3{+}c$" thus produces projective coordinates as output but uses extended coordinates internally if $c \neq 0$. We take P in affine coordinates but compute further multiples cP in extended coordinates: an inversion to convert to affine coordinates is not worthwhile.

Recall the operation costs shown in Table 1: a doubling without an addition costs $3\mathbf{M} + 4\mathbf{S}$; a tripling without an addition costs $9\mathbf{M} + 3\mathbf{S}$; a doubling followed by a mixed addition of an extended point costs $11\mathbf{M} + 4\mathbf{S}$ ($4\mathbf{M} + 4\mathbf{S}$ for the doubling producing extended coordinates, and $7\mathbf{M}$ for the mixed addition producing projective coordinates); a doubling followed by a mixed addition of an affine point costs $10\mathbf{M} + 4\mathbf{S}$; a tripling followed by a mixed addition of an extended point costs $18\mathbf{M} + 3\mathbf{S}$; a tripling followed by a mixed addition of an affine point costs $17\mathbf{M} + 3\mathbf{S}$.

For double-base *single*-scalar multiplication, we considered many sets of coefficients, including those in [3]. Table 2 shows the results from the four best coefficient sets S, along with $S = \{-1, 0, 1\}$ for comparison. Mults/ℓ denotes the number of field multiplications per bit, including the cost of initial cP computations and the cost of the subsequent double-base chain. The best result is $7.47\mathbf{M}$ per bit to compute scalar multiplication. A comparison to previous single-scalar multiplication results is presented in Section 7.5 and Table 9.

For double-base *double*-scalar multiplication, we considered all sets of coefficients as in [18], namely, $S_1 = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$ by precomputing $P{+}Q$ and $P{-}Q$, $S_5 = S_1 \cup \pm\{(5,0),(0,5)\}$ by precomputing $5P$ and $5Q$ in addition to S_1, $S_7 = S_5 \cup \pm\{(7,0),(0,7)\}$ by precomputing $7P$ and $7Q$ in addition

Table 2: New results for double-base single-scalar multiplication for 256-bit numbers

Mults	Mults/ℓ	S	Pre cost	Table size
1994.84	7.79233	$\pm\{0,1\}$	–	1
1915.82	7.48369	$\pm\{0,1,2,4,5,7,11,13\}$	44.4**M**	7
1914.77	7.47959	$\pm\{0,1,5,7,11,13,17,19,23,25\}$	76.4**M**	9
1913.14	7.47320	$\pm\{0,1,5,7,11,13,17,19\}$	60.4**M**	7
1912.91	**7.47229**	$\pm\{0,1,2,4,5,7,11,13,17,19\}$	60.4**M**	9

Note: "Pre cost" denotes the precomputation cost. This cost is already included in the first column.

to S_5, and $S_{5^2} = S_5 \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$ by precomputing $P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q$ and $5P-5Q$ extra to S_5. We also considered sets of coefficients that we label $S_{5a}, S_{5b}, S_{5c}, S_{5d}, S_{5e}, S_{7a}, S_{7b}$, S_{7c}, S_{best1} (see below). Sets S_{5*} extend S_5 to include more precomputation points. Similarly, sets S_{7*} extend S_7 to include more precomputation points. Set S_{best1} is derived from the best precomputation set of the double-base single-scalar to work for double-base double-scalar multiplication. Table 3 summarizes precomputation sets for double-base double-scalar multiplication.

The results from the four best coefficient sets S are shown in Table 4 along with the result using $S_1 = \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$ for comparison. Note that these costs already include the cost of precomputation. The best result is 2250.76**M** using set S_{5e} with table size 16.

7.2. More curve shapes, more tripling formulas, more S/M ratios. We also conducted experiments to observe the impact of our algorithms performed on other curve shapes of interest, including traditional $a = -3$ short Weierstrass curves in Jacobian coordinates and twisted Hessian curves in projective coordinates. We considered several coefficient sets but, for each curve shape, present only the coefficient set producing the best results for that curve shape. In order to explicitly distinguish the gain due to the new tripling formulas from the gain due to the optimal double-base chain generation algorithm, we also considered twisted Edwards curves with the old tripling formulas. We also varied the ratio of costs between field squaring and field multiplication, considering $\mathbf{S} = 1\mathbf{M}$, $\mathbf{S} = 0.8\mathbf{M}$, and $\mathbf{S} = 0.67\mathbf{M}$. All of these experiments used double-base single-scalar multiplications with 256-bit scalars. Table 5 shows this comparison expressed in the number of field multiplications per bit.

7.3. Comparison of tree-based vs. graph-based for single scalars. To compare the gain due to the optimal double-base chain generation algorithms with the previous (non-optimal) heuristic algorithms, we present the cost to perform scalar multiplication together with the cost to generate double-base chains. The former is measured by counting the number of field multiplications needed to perform scalar multiplication. The latter is measured by counting the number of nodes generated during the double-base chain generation of each algorithm,

Table 3: Precomputation sets for double-base double-scalar multiplications

Set S		Precomputation	Pre cost	$\lvert T\rvert$
S_1	$= \pm\{(0,0),(1,0),(0,1),(1,1),(1,-1)\}$	$P+Q, P-Q$	12.0**M**	4
S_5	$= S_1 \cup \pm\{(5,0),(0,5)\}$	$5P, 5Q$ (and S_1)	52.8**M**	6
S_7	$= S_5 \cup \pm\{(7,0),(0,7)\}$	$7P, 7Q$ (and S_5)	68.8**M**	8
S_{5^2}	$= S_5 \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$	$P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q, 5P-5Q$ (and S_5)	96.8**M**	12
S_{5a}	$= S_5 \cup \pm\{(2,0),(0,2),(4,0),(0,4)\}$	$2P, 2Q, 4P, 4Q$ (and S_5)	52.8**M**	10
S_{5b}	$= S_{5a} \cup \pm\{(1,2),(1,-2),(2,1),(2,-1),(1,4),(1,-4),(4,1),(4,-1),(1,5),(1,-5),(5,1),(5,-1)\}$	$P+2Q, P-2Q, 2P+Q, 2P-Q, P+4Q, P-4Q, 4P+Q, 4P-Q, P+5Q, P-5Q, 5P+Q, 5P-Q$ (and S_{5a})	136.8**M**	22
S_{5c}	$= S_{5a} \cup \pm\{(2,2),(2,-2),(4,4),(4,-4),(5,5),(5,-5)\}$	$2P+2Q, 2P-2Q, 4P+4Q, 4P-4Q, 5P+5Q, 5P-5Q$ (and S_{5a})	100.8**M**	16
S_{5d}	$= S_{5a} \cup \pm\{(2,4),(2,-4),(4,2),(4,-2),(2,5),(2,-5),(5,2),(5,-2),(4,5),(4,-5),(5,4),(5,-4)\}$	$2P+4Q, 2P-4Q, 4P+2Q, 4P-2Q, 2P+5Q, 2P-5Q, 5P+2Q, 5P-2Q, 4P+5Q, 4P-5Q, 5P+4Q, 5P-4Q$ (and S_{5a})	148.8**M**	22
S_{5e}	$= S_{5a} \cup \pm\{(1,5),(1,-5),(5,1),(5,-1),(5,5),(5,-5)\}$	$P+5Q, P-5Q, 5P+Q, 5P-Q, 5P+5Q, 5P-5Q$ (and S_{5a})	96.8**M**	16
S_{7a}	$= S_{5a} \cup \pm\{(7,0),(0,7)\}$	$7P, 7Q$ (and S_{5a})	68.8**M**	12
S_{7b}	$= S_{5b} \cup \pm\{(7,0),(0,7),(1,7),(1,-7),(7,1),(7,-1)$	$7P, 7Q, P+7Q, P-7Q, 7P+Q, 7P-Q$ (and S_{5b})	180.8**M**	28
S_{7c}	$= S_{5c} \cup \pm\{(7,0),(0,7),(7,7),(7,-7)$	$7P, 7Q, 7P+7Q, 7P-7Q$ (and S_{5c})	132.8**M**	20
S_{best1}	$= S_1 \cup \pm\{(2,0),(0,2),(4,0),(0,4),(5,0),(0,5),(7,0),(0,7),(11,0),(0,11),(13,0),(0,13),(17,0),(0,17),(19,0),(0,19)\}$	$2P, 4P, 5P, 7P, 11P, 13P, 17P, 19P, 2Q, 4Q, 5Q, 7Q, 11Q, 13Q, 17Q, 19Q$ (and S_1)	132.8**M**	20

Note: "Pre cost" denotes the precomputation cost. "$\lvert T\rvert$" denotes the table size.

namely, the tree-based [15] (for double-base single-scalar), tree-JBT [18] (for double-base double-scalar), DAG-based (DAG, Section 3), rectangular DAG-based (rDAG, Section 4), and reduced rectangular DAG-based (rrDAG, Section 5). Note that all our DAG-based algorithms are applicable for both double-base single-scalar and double-base double-scalar.

Table 4: New results for double-base double-scalar multiplication for 256-bit numbers

Mults	Mults/ℓ	S	Table size
2351.86	9.18695	S_1	4
2266.32	8.85281	S_{5d}	22
2264.67	8.84637	S_{5b}	22
2251.70	8.79570	S_{5^2}	12
2250.76	**8.79203**	S_{5e}	16

Table 5: Impact of other tripling formulas, curve shapes, and **S**/**M** ratios on double-base single-scalar for 256-bit numbers

Curve shape		**S**/**M** ratio		
		1	0.8	0.67
Jacobian-3	[5]	-	9.34297	-
	(new)	10.20950	9.12516	8.39722
Hessian	[4]	-	8.77382	-
	(new)	9.16351	8.52279	8.09017
Twisted Edwards	[19]	8.40625	7.62109	-
	(new)	8.27195	7.52247	7.01979
Twisted Edwards (new formulas)		**8.20036**	**7.47415**	**6.97923**

The number of nodes together with the operation cost per node reflect the time required to run these algorithms. We categorize nodes into 2 types, ones with operation cost $\log_2 n$ and ones with operation cost $\log_2 2^\alpha 3^\beta$. In tree-based, tree-JBT, DAG-based and rDAG-based algorithms, only the first type appears. In rrDAG-based, both types appear; nodes with $\log_2 2^\alpha 3^\beta$ operation cost are intermediate nodes while nodes with $\log_2 n$ operation cost are boundary nodes. The cost to generate double-base chains is computed using the number of nodes and operation cost per node.

We emphasize that the tree-based and tree-JBT algorithms *do not* produce optimal chains while DAG-based, rDAG-based and rrDAG-based algorithms *do* produce optimal chains. Beware that these costs have somewhat high variance, and this variance is visible despite the number of experiments that we carried out.

Table 6 displays the results of these experiments for double-base *single*-scalar multiplications using 256-bit scalars, assuming $\mathbf{S} = 0.8\mathbf{M}$. We varied the bound B used for tree pruning, i.e., the maximum number of nodes kept for each level in the tree-based approach, by using $B = 10^2$, $B = 10^3$, $B = 10^4$, and $B = 10^5$; costs increase as B increases, until B is large enough to eliminate the pruning. For the rrDAG, we set the size of subgraphs to be $\alpha = \lfloor (\log_2 n)^{0.5} \rceil$ and $\beta = \lfloor (\log_3 n)^{0.5} \rceil$. These experiments focus on the case where no precomputation is allowed, i.e., $S = \{-1, 0, 1\}$, for comparability to previous work using this S.

These results suggest that in double-base single-scalar multiplication only the most extreme pruning produces results faster with the tree-based method than

Table 6: Tree-based and graph-based comparison for 256-bit numbers

	Method	Mults	Mults/ℓ	Optimal	Nodes	Cost
Tree	$B = 10^2$	2035.56	7.95141	no	6072 + 0	1554432
	$B = 10^3$	2034.46	7.94711	no	41948 + 0	10738688
	$B = 10^4$	2034.46	7.94711	no	171739 + 0	43965184
	$B = 10^5$	2034.46	7.94711	no	173206 + 0	44340736
Graph	DAG	1994.83	7.79230	yes	10449 + 0	2674944
	rDAG	1994.83	7.79230	yes	40845 + 0	10456320
	rrDAG	1994.83	7.79230	yes	4607 + 38106	2574244

Table 7: Tree-based and graph-based comparison at 256-bitlength for double-based double-scalar multiplication

Method	Mults	Mults/ℓ	Optimal	Nodes	Cost
Tree-JBT	2392.17	9.34441	no	34339 + 0	8790841
DAG	2335.97	9.12488	yes	40309 + 0	10319079
rDAG	2335.97	9.12488	yes	80193 + 0	20529550
rrDAG	2335.97	9.12488	yes	11286 + 93267	6303312

DAG, but trees produce suboptimal chains. It depends on the application, e.g., on how often the chain will be used, whether the extra effort is justified. It is interesting to see that even for large pruning bounds B the tree-based algorithm does not reach optimal chains.

The results clearly show the obvious savings of rrDAG over rDAG in the cost of computing the chains and the less obvious benefit of rrDAG over DAG.

7.4. Comparison of tree-based vs. graph-based for double scalars. Table 7 displays the results of analogous experiments for *double*-scalar multiplications. For the rrDAG, we set the size of subgraphs to be $\alpha = \lfloor (\log_2 n)^{0.5} \rceil$ and $\beta = \lfloor (\log_3 n)^{0.5} \rceil$. Because we focus on counting the number of nodes generated during the search for the chain and not on finding the best precomputation set, these experiments focus on the simple case where no precomputation is allowed, i.e., $S = \pm\{(0,0),(1,0),(0,1)\}$. Extensive comparisons of the cost to evaluate double-base double-scalar multiplication where precomputation is allowed can be found in Table 10.

These results suggest that in double-base double-scalar multiplication the cost to generate optimal double-base chains using rrDAG is less than using the tree-JBT approach. Moreover, the non-optimal chains are indeed worse than the optimal ones. This means that our new rrDAG algorithm achieves a better performance in both the time to generate double-base chains and the time to evaluate those chains.

7.5. Single-base vs. double-base for single scalars. We also implemented a conventional signed-sliding-window double-and-add algorithm for computing single-base single-scalar and single-base double-scalar multiplication. We used the fastest formulas available, namely, twisted Edwards with parameter $a = -1$,

Table 8: New results for single-base single-scalar multiplication for 256-bit numbers

Mults	Mults/ℓ	S	Table size
2170.39	8.47808	$\pm\{0,1\}$	1
1947.55	7.60762	$\pm\{0,1,3,5,7,\ldots,31\}$	16
1939.85	7.57752	$\pm\{0,1,3,5,7,\ldots,17\}$	9
1939.02	7.57428	$\pm\{0,1,3,5,7,\ldots,25\}$	13
1938.57	**7.57252**	$\pm\{0,1,3,5,7,\ldots,21\}$	11

Table 9: Comparison of single-scalar multiplication to previous works

Base	Mults	Mults/ℓ	S	Table size
double	2092.60 [14]	8.17422	$\pm\{0,1\}$	1
double	**1994.84** (new)	**7.79233**	$\pm\{0,1\}$	1
single	1950.60 [19]	7.61953	$\pm\{0,1,3,5,7,9,11,13,15\}$	8
single	1938.57 (new)	7.57252	$\pm\{0,1,3,5,7,\ldots,21\}$	11
double	1913.14 (new)	7.47320	$\pm\{0,1,5,7,11,13,17,19\}$	7
double	**1912.91** (new)	**7.47229**	$\pm\{0,1,2,4,5,7,11,13,17,19\}$	9

together with the state-of-the-art technique of using mixed coordinate systems. We ran the experiment using various precomputation sets, namely, the 21 sets listed in [3], and 6 other sets: $\pm\{0,1,2,4,5,7\}$, $\pm\{0,1,2,4,5,7,11,13,17,19,23,25\}$, $\pm\{0,1,2,4,5\}$, $\pm\{0,1,2,4,5,7,11\}$, $\pm\{0,1,2,4,5,7,11,13\}$, $\pm\{0,1,2,4,5,7,11,13,17,19\}$.

Note that the set $\pm\{0,1,5,7\}$ was considered in [3]. The reason we also considered the set $\pm\{0,1,2,4,5,7\}$ is that in order to compute $5P$ we need to compute $2P$ and $4P$. Therefore, we included $2P$ and $4P$ in the precomputed set. Similar reasons apply to the other 5 extra sets.

Table 8 shows results from the best precomputation sets for single-scalar multiplication. We also include the case of no precomputation, i.e., $S = \{-1,0,1\}$. The best result is $7.57\mathbf{M}$ per bit to compute single-base single-scalar multiplication.

We compare our results to previous works. Comparison for single-scalar multiplication is shown in Table 9. If no precomputation is allowed, Doche [14] reported $2092.60\mathbf{M} \approx 8.17\ell\mathbf{M}$ using "near optimal controlled" method. Our results show that our algorithm requires only $1994.84\mathbf{M} \approx 7.79\ell\mathbf{M}$. For the case that precomputation is allowed, Bernstein and Lange [5] reported $2038.7\mathbf{M} \approx 7.96\ell\mathbf{M}$ using $S = \pm\{0,1,3,5,\ldots,17\}$ with inverted Edwards coordinates. Hisil, Wong, Carter and Dawson [19] then reported $1950.60\mathbf{M} \approx 7.62\ell\mathbf{M}$ using "4-NAF" with mixed (projective/extended) Edwards coordinates and the $a = -1$ addition speedup. Table 8 shows that replacing 4-NAF with $\pm\{0,1,3,5,\ldots,21\}$ does better, needing $\approx 7.57\ell\mathbf{M}$. This is further beaten by our best result for double-base chains, only $1912.91\mathbf{M} \approx 7.47\ell\mathbf{M}$.

Table 10: Comparison of double-scalar multiplication to previous works

B	Method		$\|T\|$	192-bit	256-bit	320-bit	384-bit	448-bit	512-bit
single	s.sld $_{\omega=2}$	[20]	16	2355	3140	3925	4710	5495	6280
	s.sld $_{\omega=3}$	[20]	34	2286	3049	3811	4573	5335	6097
	inter $_{\omega=4}$	[20]	20	2295	3060	3825	4590	5356	6121
	inter $_{\omega=5}$	[20]	44	2294	3058	3823	4587	5352	6116
	JSF	[18]	4	2044	2722	3401	4062	4758	5436
double	RHBTJF	[1]	2	1997	2661	3326	3990	4654	5319
	Tree-JBT	[18]	4	1953	2602	3248	3896	4545	5197
	RHBTJF + new tpl	(new)	2	1946	2594	3241	3889	4536	5183
	Tree-JBT$_5$	[18]	6	1920	2543	3168	3792	4414	5042
	HBTJF	[1]	14	1914	2530	3145	3761	4376	4992
	Tree-JBT$_7$	[18]	8	1907	2521	3137	3753	4365	4980
	Tree-JBT$_{5^2}$	[18]	12	1890	2485	3079	3677	4270	4862
	HBTJF + new tpl	(new)	14	1859	2456	3053	3650	4247	4844
single	slide $_{\omega=3}$		4	1884	2494	3103	3715	4324	4935
	slide $_{\omega=4}$		8	1838	2424	3009	3595	4181	4767
	slide $_{\omega=5}$		16	1836	2391	2944	3500	4055	4610
	slide $_{\omega=6}$		32	1931	2478	3016	3550	4084	4617
double	Tree-JBT + new tpl	(new)	4	1848	2467	3074	3695	4314	4919
	Tree-JBT$_5$ + new tpl	(new)	6	1823	2408	3010	3592	4182	4792
	Tree-JBT$_7$ + new tpl	(new)	8	1800	2394	2973	3554	4142	4727
	Tree-JBT$_{5^2}$ + new tpl	(new)	12	1787	2354	2918	3489	4056	4621
	rrDAG	(new)	4	1768	2352	2937	3521	4105	4690
	rrDAG$_5$	(new)	6	1748	2315	2883	3450	4018	4585
	rrDAG$_7$	(new)	8	1734	2292	2851	3410	3969	4527
	rrDAG$_{5^2}$	(new)	12	1709	2252	2794	3337	3879	4422

Note: "B" in the first column specifies the base used. Abbreviations in the second column: "s.sld" means simutaneous sliding (precompute $c_iP + d_iQ$), "inter" means interleaving (precompute multiples of P and Q individually, i.e., c_iP and d_iQ), "slide" means sliding window, and "new tpl" means using our new tripling formulas. "$|T|$" in the third column denotes the table size.

7.6. Single-base vs. double-base for double scalars. Table 10 shows an analogous comparison for double-scalar multiplication. The results show that single-base signed-sliding-window methods use fewer multiplications than double-base Tree-JBT (with old tripling formulas) for double-scalar multiplication. Our rrDAG algorithms, even without precomputations, use fewer multiplications than signed-sliding-window methods with the best precomputation. This means that our algorithm is less costly and at the same time requires less space for look-up tables.

The Tree-JBT and JSF costs in Table 10 are copied from [18] which obtained by implicitly converting precomputed points into affine coordinates. However, these cost *do not* include the cost of conversion. Therefore, the total cost to perform scalar multiplication using Tree-JBT and JSF would be higher than the

costs shown in the table. The other costs in Table 10 are the *total cost* to perform scalar multiplication.

The results attributed to [20] are quoted from [20] for 160-bit integers; we extrapolated linearly to larger bitlengths. The sliding-window results and rrDAG results in Table 10 are from our own experiments.

To summarize, our new double-base chains are (for 256-bit scalars) more than 6% better than all previous results for double-scalar multiplication, and more than 10% better than all previous double-base results for double-scalar multiplication.

References

[1] Jithra Adikari, Vassil S. Dimitrov, Laurent Imbert, *Hybrid binary-ternary number system for elliptic curve cryptosystems*, IEEE Transactions on Computers **60** (2011), 254–265. Citations in this document: §10, §10.

[2] Rana Barua, Tanja Lange (editors), *Progress in cryptology — INDOCRYPT 2006, 7th international conference on cryptology in India, Kolkata, India, December 11–13, 2006, proceedings*, Lecture Notes in Computer Science, 4329, Springer, 2006. ISBN 3-540-49767-6. See [17].

[3] Daniel J. Bernstein, Peter Birkner, Tanja Lange, Christiane Peters, *Optimizing double-base elliptic-curve single-scalar multiplication*, in Indocrypt 2007 (2007), 167–182. URL: https://eprint.iacr.org/2007/414. Citations in this document: §1.1, §2.1, §7.1, §7.5, §7.5.

[4] Daniel J. Bernstein, Chitchanok Chuengsatiansup, David Kohel, Tanja Lange, *Twisted Hessian curves*, in Latincrypt 2015 (2015), 269–294. URL: https://eprint.iacr.org/2015/781. Citations in this document: §1.1, §1.1, §1.1, §1.1, §5.

[5] Daniel J. Bernstein, Tanja Lange, *Analysis and optimization of elliptic-curve single-scalar multiplication*, in Finite fields and applications *Fq*8 (2008), 1–19. URL: https://cr.yp.to/antiforgery/efd-20071204.pdf. Citations in this document: §5, §7.5.

[6] Daniel J. Bernstein, Tanja Lange (editors), *Explicit Formulas Database*, accessed 3 October 2015 (2015). URL: https://hyperelliptic.org/EFD. Citations in this document: §2.3.

[7] Joppe W. Bos, Thorsten Kleinjung, *ECM at work*, in Asiacrypt 2012 (2012), 467–484. URL: https://eprint.iacr.org/2012/089. Citations in this document: §1.1, §1.1.

[8] Alex Capuñay, Nicolas Thériault, *Computing optimal 2-3 chains for pairings*, in Latincrypt 2015 (2015), 225–244. Citations in this document: §1.2.

[9] Stephen A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, Department of Mathematics, Harvard University, 1966. URL: https://cr.yp.to/bib/1966/cook.html. Citations in this document: §5.2.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to algorithms*, MIT Press, 2009 (3rd edition). Citations in this document: §3.4.

[11] Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271. Citations in this document: §3.

[12] Vassil Dimitrov, Laurent Imbert, Pradeep Kumar Mishra, *Efficient and secure elliptic curve point multiplication using double-base chains*, in Asiacrypt 2005 (2005), 59–78; see also newer version [**13**]. URL: https://www.iacr.org/archive/asiacrypt2005/059/059.pdf. Citations in this document: §1.1.

[13] Vassil Dimitrov, Laurent Imbert, Pradeep Kumar Mishra, *The double-base number system and its application to elliptic curve cryptography*, Mathematics of Computation **77**, 1075–1104; see also older version [**12**]. URL: https://www.ams.org/mcom/2008-77-262/S0025-5718-07-02048-0/S0025-5718-07-02048-0.pdf.

[14] Christophe Doche, *On the enumeration of double-base chains with applications to elliptic curve cryptography*, in Asiacrypt 2014 (2014). URL: https://eprint.iacr.org/2014/371. Citations in this document: §7.5, §9.

[15] Christophe Doche, Laurent Habsieger, *A tree-based approach for computing double-base chains*, in Information Security and Privacy, 13th Australasian Conference (2008), 433–446. URL: https://web.science.mq.edu.au/~doche/tree_DBNS.pdf. Citations in this document: §1.2, §1.2, §7.3.

[16] Christophe Doche, Thomas Icart, David R. Kohel, *Efficient scalar multiplication by isogeny decompositions*, in Public Key Cryptography 2006 (2006), 191–206. URL: https://eprint.iacr.org/2005/420. Citations in this document: §1.1.

[17] Christophe Doche, Laurent Imbert, *Extended double-base number system with applications to elliptic curve cryptography*, in [**2**] (2006), 335–348. URL: https://eprint.iacr.org/2006/330. Citations in this document: §1.1.

[18] Christophe Doche, David R. Kohel, Francesco Sica, *Double-base number system for multi-scalar multiplications*, in Eurocrypt 2009 (2009), 502–517. URL: https://www.iacr.org/archive/eurocrypt2009/54790501/54790501.pdf. Citations in this document: §7.1, §7.3, §10, §10, §10, §10, §10, §7.6.

[19] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards curves revisited*, in Asiacrypt 2008 (2008), 326–343. URL: https://eprint.iacr.org/2008/522. Citations in this document: §1, §1.1, §2.4, §5, §7.5, §9.

[20] Katsuyuki Okeya, Kouichi Sakurai, *Fast multi-scalar multiplication methods on elliptic curves with precomputation strategy using Montgomery trick*, in CHES 2002 (2002), 564–578. Citations in this document: §10, §10, §10, §10, §7.6, §7.6.

[21] Thomas Simpson, *Essays on several curious and useful subjects in speculative and mix'd mathematics, illustrated by a variety of examples*, 1740. URL: https://cr.yp.to/bib/1740/simpson.html. Citations in this document: §5.2.

[22] Andrei L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Soviet Mathematics Doklady **3** (1963), 714–716. ISSN 0197-6788. Citations in this document: §5.2.

A Example of rDAG-based approach

In this appendix we detail the steps in computing double-chains for $n = 17$ under the set of coefficients $S = \{-1, 0, 1\}$ and operation costs according to Table 1, namely, tripling = 11.4, doubling = 6.2, doubling followed by addition and/or subtraction by one = 13.2, and tripling followed by addition and/or subtraction by one = 19.4.

The algorithm starts by initializing $(0, 0, 17)$.

At $(0, 0, 17)$: since 17 is odd, there are 3 outgoing edges to:
- $(1, 0, 8)$ having cost $13.2 = 0 + 13.2$
- $(1, 0, 9)$ having cost $13.2 = 0 + 13.2$
- $(0, 1, 6)$ having cost $19.4 = 0 + 19.4$

At $(1, 0, 8)$: since 8 is even, there are 2 outgoing edges to:
- $(2, 0, 4)$ having cost $19.4 = 13.2 + 6.2$
- $(1, 1, 3)$ having cost $32.6 = 13.2 + 19.4$

At $(1, 0, 9)$: since 9 is odd, there are 3 outgoing edges to:
- $(2, 0, 4)$ having cost $26.4 = 13.2 + 13.2$. This node already exists, and the new cost is more expensive than the previous one. Therefore, the previous cheaper cost 19.4 remains.
- $(2, 0, 5)$ having cost $26.4 = 13.2 + 13.2$
- $(1, 1, 3)$ having cost $24.6 = 13.2 + 11.4$. This node also already exists, but the new cost is cheaper than the previous one. Thus, we update the cost at this node.

At $(0, 1, 6)$: since 6 is even, there are 2 outgoing edges to:
- $(1, 1, 3)$ having cost $25.6 = 19.4 + 6.2$. However, this is not cheaper than the previous cost. Therefore, no update at this node.
- $(0, 2, 2)$ having cost $30.8 = 19.4 + 11.4$

At $(2, 0, 5)$: since 5 is odd, there are 3 outgoing edges to:
- $(3, 0, 2)$ having cost $39.6 = 26.4 + 13.2$
- $(3, 0, 3)$ having cost $39.6 = 26.4 + 13.2$
- $(2, 1, 2)$ having cost $45.8 = 26.4 + 19.4$

At $(2, 0, 4)$: since 4 is even, there are 2 outgoing edges to:
- $(3, 0, 2)$ having cost $25.6 = 19.4 + 6.2$. This is cheapter than before, so update the cost.
- $(2, 1, 1)$ having cost $38.8 = 19.4 + 19.4$

At $(1, 1, 3)$: since 3 is odd, there are 3 outgoing edges to:
- $(2, 1, 1)$ having new cost $37.8 = 24.6 + 13.2$
- $(2, 1, 2)$ having new cost $37.8 = 24.6 + 13.2$
- $(1, 2, 1)$ having cost $36 = 24.6 + 11.4$

At $(0, 2, 2)$: since 2 is even, there are 2 outgoing edges to:
- $(1, 2, 1)$ having no cost update
- $(0, 3, 1)$ having cost $50.2 = 30.8 + 19.4$

At $(3, 0, 3)$: since 3 is odd, there are 3 outgoing edges to:
- $(4, 0, 1)$ having cost $52.8 = 39.6 + 13.2$
- $(4, 0, 2)$ having cost $52.8 = 39.6 + 13.2$
- $(3, 1, 1)$ having cost $51 = 39.6 + 11.4$

At $(3, 0, 2)$: since 2 is even, there are 2 outgoing edges to:
- $(4, 0, 1)$ having new cost $31.8 = 25.6 + 6.2$
- $(3, 1, 1)$ having new cost $45 = 25.6 + 19.4$

At $(2, 1, 2)$: since 2 is even, there are 2 outgoing edges to:
- $(3, 1, 1)$ having new cost $44 = 37.8 + 6.2$
- $(2, 2, 1)$ having cost $57.2 = 37.8 + 19.4$

At $(2, 1, 1)$, $(1, 2, 1)$ and $(0, 3, 1)$, since $1 \in S$, continue to next node.

At $(4, 0, 2)$, there are 2 outgoing edges to:
- $(5, 0, 1)$ having cost $59 = 52.8 + 6.2$
- $(4, 1, 1)$ having cost $72.2 = 52.8 + 19.4$.

At $(4, 0, 1)$, $(3, 1, 1)$, $(2, 2, 1)$, $(5, 0, 1)$ and $(4, 1, 1)$, since $1 \in S$, continue to next node.

Once all nodes are visited, by selecting the path with minimum cost, we obtain the optimal $\{2, 3\}$ chain, namely $2^4 + 1$, having cost 31.8.

B Code for rectangular DAG-based algorithm

```
import math

# cost of point arithmetic on twisted Edwards #
mad = 7        # mixed addition (Z = 1)  need to +1 in case tpl -> mad
add = 8        # general addition        need to +1 in case tpl -> add
dbl = 3 + 0.8*4      # doubling
tpl = 9 + 0.8*3      # tripling

w = 1         # max coefficient (d_i)
s = []        # precomputation
for i in range (w,-(w+1),-1):
  s.append(i)

def basic(n,s):

  amax = int(math.log(n,2) + w + 0.5) + 1    # max power of 2
  bmax = int(math.log(n,3) + w + 0.5) + 1    # max power of 3
  wmax = 2*w + 1                             # max element at (i,j)

  # init table #
  table = []
  for i in range (amax+1):
    table.append([])
    for j in range (bmax+1):
      table[i].append([])
      for k in range (wmax+1):
        table[i][j].append([])

  table[0][0][0] = [n,0,[0,0,0]]  # init root
  # table[i][j][k] = [data1, data2, data3]
  # data1 = curr n
  # data2 = cost to reach this curr n,
  # data3 = [div 2 or 3, amount add, prev k index]

  result = []

  for i in range (amax):    # for each power of 2
    for j in range (bmax):  # for each power of 3
      base2 = -1
      base3 = -1
      for k in range (wmax):  # for each element at vertex (i,j)
        if table[i][j][k] == []: continue
        t = table[i][j][k][0]

        if t in s and not(t == 0):  # reach known chain integer
          if len(result) == 0 or table[i][j][k][1] < result[0][1][1]:
            result.insert(0,[[i,j,k],table[i][j][k]])  # add result
          continue

        for ic in range (len(s)):  # for each coefficient
          c = s[ic]
          if (t-c)%2 == 0:
            curr2 = (t-c)/2
            if base2 == -1: base2 = curr2
            idx2 = curr2 - base2
            cost = table[i][j][k][1] + dbl
            if abs(c) == 1: cost += mad
            elif abs(c) > 0: cost += add
            if table[i+1][j][idx2] == [] or table[i+1][j][idx2][1] > cost:
            # update if new number or less cost
              table[i+1][j][idx2] = [(t-c)/2,cost,[2,c,k]]

          if (t-c)%3 == 0:
            curr3 = (t-c)/3
            if base3 == -1:
```

```
            if table[i][j+1][0] == []:
              base3 = curr3
            else:
              base3 = table[i][j+1][0][0]
          idx3 = curr3 - base3
          cost = table[i][j][k][1] + tpl
          if abs(c) == 1: cost += mad + 1   # p2e
          elif abs(c) > 0: cost += add + 1  # p2e
          if table[i][j+1][idx3] == [] or table[i][j+1][idx3][1] > cost:
          # update if new number or less cost
            table[i][j+1][idx3] = [(t-c)/3,cost,[3,c,k]]
return table,result
```

C Code for reduced rectangular DAG-based algorithm

```
import math

# cost of point arithmetic on twisted Edwards #
mad = 7        # mixed addition (Z = 1)  need to +1 in case tpl -> mad
add = 8        # general addition        need to +1 in case tpl -> mad
dbl = 3 + 0.8*4      # doubling
tpl = 9 + 0.8*3      # tripling

w = 1         # max coefficient (d_i)
s = []        # precomputation
for i in range (w,-(w+1),-1):
  s.append(i)

wmax = 2*w + 1      # max element at (i,j)

amod = 16
bmod = 13

def getchain(table,x,rev):
  chain = []
  strchain = ""     # sequence of (mul,add) e.g., (2,0) (3,0) (3,-1) = 3*(3*(2-0)-0)-1 = 17
  [i,j,k] = x[0]
  t = x[1][0]       # value
  d = x[1][2][0]    # dbl or tpl
  a = x[1][2][1]    # add
  while i > 0 or j > 0:
    k = table[i][j][k][2][2]
    if d == 2: i -= 1
    elif d == 3: j -= 1
    strchain += "(" + str(d) + "," + str(a) + ") "
    if rev: chain.insert(0,[d,a])
    else: chain.append([d,a])
    d = table[i][j][k][2][0]
    a = table[i][j][k][2][1]
    t = table[i][j][k][0]
  return chain

def boundary3(table,idxa,idxb,n,mincost):
  i = 0
  init = False
  for j in range (bmod+1):
    for k in range (wmax):

      if idxa*amod+i >= len(table): continue
      elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
      elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue

      if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
      if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

      nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
        # [starting coordinate, table at that coordinate]
      ch = getchain(table,nd,True)
        # chain from that coordinate back to the original n
      t = n
      for x in ch:
        t -= x[1]
        t /= x[0]
      if not init:
        nb = t
        t %= 2**amod * 3**bmod
        nm = t
        table[idxa*amod+i][idxb*bmod+j][k] = [t,nd[1][1],nd[1][2]]
        init = True
      else:
```

```
        tt = t % (2**amod * 3**bmod)
        table[idxa*amod+i][idxb*bmod+j][k] = [nm+(t-nb),nd[1][1],nd[1][2]]
    if not init: continue
    nb/=3; nm/=3

def boundary2(table,idxa,idxb,n,mincost):
  j = 0
  init = False
  for i in range (amod+1):
    for k in range (wmax):

      if idxa*amod+i >= len(table): continue
      elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
      elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue

      if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
      if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

      nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
        # [starting coordinate, table at that coordinate]
      ch = getchain(table,nd,True)
        # chain from that coordinate back to the original n
      t = n
      for x in ch:
        t -= x[1]
        t /= x[0]
      if not init:
        nb = t
        t %= 2**amod * 3**bmod
        nm = t
        table[idxa*amod+i][idxb*bmod+j][k] = [t,nd[1][1],nd[1][2]]
        init = True
      else:
        tt = t % (2**amod * 3**bmod)
        table[idxa*amod+i][idxb*bmod+j][k] = [nm+(t-nb),nd[1][1],nd[1][2]]
    if not init: continue
    nb/=2; nm/=2

def subgraph(table,idxa,idxb,n,check,result,mincost):
  for i in range (amod+1):    # for each power of 2
    for j in range (bmod+1):  # for each power of 3
      base2 = -1
      base3 = -1
      for k in range (wmax):  # for each element at vertex (i,j)

        if idxa*amod+i >= len(table): continue
        elif idxb*bmod+j >= len(table[idxa*amod+i]): continue
        elif k >= len(table[idxa*amod+i][idxb*bmod+j]): continue

        if table[idxa*amod+i][idxb*bmod+j][k] == []: continue
        t = table[idxa*amod+i][idxb*bmod+j][k][0]

        if table[idxa*amod+i][idxb*bmod+j][k][1] > mincost[0]: continue

        # check condition #
        if check:
          nd = [[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]]
          ch = getchain(table,nd,True)
          m = n
          for x in ch:
            m -= x[1]
            m /= x[0]
          if m == 1:  # reach known chain integer
            if len(result) == 0 or table[idxa*amod+i][idxb*bmod+j][k][1] < result[0][1][1]:
              result.insert(0,[[idxa*amod+i,idxb*bmod+j,k],table[idxa*amod+i][idxb*bmod+j][k]])
              # add result
              mincost[0] = result[0][1][1]
            continue
```

```
        for ic in range (len(s)):  # for each coefficient
          c = s[ic]

          if i < amod and (t-c)%2 == 0:
            curr2 = (t-c)/2
            if base2 == -1: base2 = curr2
            idx2 = curr2 - base2
            cost = table[idxa*amod+i][idxb*bmod+j][k][1] + dbl
            if abs(c) == 1: cost += mad
            elif abs(c) > 0: cost += add
            if table[idxa*amod+i+1][idxb*bmod+j][idx2] == [] or \
              table[idxa*amod+i+1][idxb*bmod+j][idx2][1] > cost:
              # update if new number or less cost
              table[idxa*amod+i+1][idxb*bmod+j][idx2] = [(t-c)/2,cost,[2,c,k]]

          if j < bmod and (t-c)%3 == 0 and not (idxa>0 and i==0):
            curr3 = (t-c)/3
            if base3 == -1:
              if table[idxa*amod+i][idxb*bmod+j+1][0] == []:
                base3 = curr3
              else:
                base3 = table[idxa*amod+i][idxb*bmod+j+1][0][0]
            idx3 = (k+ic)/3
            cost = table[idxa*amod+i][idxb*bmod+j][k][1] + tpl
            if abs(c) == 1: cost += mad + 1   # p2e
            elif abs(c) > 0: cost += add + 1  # p2e
            if table[idxa*amod+i][idxb*bmod+j+1][idx3] == [] or \
              table[idxa*amod+i][idxb*bmod+j+1][idx3][1] > cost:
              # update if new number or less cost
              table[idxa*amod+i][idxb*bmod+j+1][idx3] = [(t-c)/3,cost,[3,c,k]]

def advance(n,s):

  mincost = [n]

  amax = int(math.log(n,2) + w + 0.5) + 1    # max power of 2
  bmax = int(math.log(n,3) + w + 0.5) + 1    # max power of 3

  # init table #
  table = []
  for i in range (amax+1):
    table.append([])
    for j in range (bmax+1):
      table[i].append([])
      for k in range (wmax+1):
        table[i][j].append([])

  table[0][0][0] = [n%(2**amod * 3**bmod),0,[0,0,0]]  # init root
  # table[i][j][k] = [data1, data2, data3]
  # data1 = curr n
  # data2 = cost to reach this curr n,
  # data3 = [div 2 or 3, amount add, prev k index]

  result = []

  for idxJ in range (bmax/bmod):
    for idxI in range (amax/amod):
      if idxI > 0:
        boundary3(table,idxI,idxJ,n,mincost)
      if idxJ > 0:
        boundary2(table,idxI,idxJ,n,mincost)
      subgraph(table,idxI,idxJ,n,True,result,mincost)
  return table,result
```