Pippenger's Multiproduct and Multiexponentiation Algorithms

(Extended Version)

Ryan Henry

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
rhenry@cs.uwaterloo.ca

Technical Report CACR 2010-26

Version 1.0

September 14, 2010

Abstract. In this paper we treat Pippenger's algorithms for the multiproduct problem and the multiexponentiation problem. This paper is intended to be self-contained; we begin by presenting the relevant background material and motivating the problem with some example applications. We then present the algorithms in their original graph-theoretic formulations, and then translate our graph-theoretic formulations into pseudocode which can be applied to directly solve the problems.

Key words: Multiproduct, multiexponentiation, Pippenger's algorithm.

A note from the author: This is version 1.0 of a living document; please consult `http://www.cacr.math.uwaterloo.ca/tech_reports.html` or `http://www.cs.uwaterloo.ca/~rhenry` to ensure that you have the latest version. I have made every effort to keep this paper completely self-contained. The text comprises a vast amount of material with many diagrams and worked examples illustrating difficult or potentially unfamiliar concepts. As such, there have been numerous opportunities for your fallible author to make calculation errors, typos, etc. If you discover an error in this paper, please (after confirming that the error exists in the latest version) inform the author at rhenry@cs.uwaterloo.ca.

1 Introduction

Let $\mathbf{M} \in \mathbb{Z}_2^{p\times q}$; that is, $\mathbf{M}$ is a matrix of dimension $p \times q$ with entries drawn from the set $\{0, 1\}$. Let the $(i, j)^{\text{th}}$ entry (i.e., the entry in the i^{th} row and j^{th} column) of $\mathbf{M}$ be denoted by $m_{i,j}$.

Given a finite vector $\vec{\mathbf{x}} = \langle x_0, x_1, \ldots, x_{q-1}\rangle$ of q elements from a commutative monoid[1], the *multiproduct problem* is to produce the output vector $\vec{\mathbf{y}} = \langle y_0, y_1, \ldots, y_{p-1}\rangle$, where

$$\begin{aligned} y_0 &= x_0^{m_{0,0}} x_1^{m_{0,1}} \cdots x_{q-1}^{m_{0,(q-1)}}, \\ y_1 &= x_0^{m_{1,0}} x_1^{m_{1,1}} \cdots x_{q-1}^{m_{1,(q-1)}}, \\ &\vdots \\ y_{p-1} &= x_0^{m_{(p-1),0}} x_1^{m_{(p-1),1}} \cdots x_{q-1}^{m_{(p-1),(q-1)}}. \end{aligned}$$

Pippenger's Multiproduct Algorithm is an algorithm for solving instances of the multiproduct problem while attempting to minimize the total number of multiplications used to compute the output sequence; Pippenger proved in [12] that his algorithm is asymptotically optimal.[2]

Example 1. Suppose $\vec{\mathbf{x}} = \langle x_0, x_1, x_2\rangle$ is given and let $\mathbf{M} = \begin{bmatrix} 1\,0\,1 \\ 1\,1\,0 \\ 0\,0\,1 \\ 1\,1\,1 \end{bmatrix}$. Then, the solution to the multiproduct problem is the sequence $y_0 = x_0x_2$, $y_1 = x_0x_1$, $y_2 = x_2$ and $y_3 = x_0x_1x_2$. Substituting the values $x_0 = 3$, $x_1 = 5$, $x_2 = 7$ (and working over $\mathbb{Z}$) yields: $y_0 = 21$, $y_1 = 15$, $y_2 = 7$ and $y_3 = 105$. ♢

Now, consider the generalization of the multiproduct problem where the entries of $\mathbf{M}$ are drawn from $\mathbb{Z}_{k+1}$ for some positive integer $k \geq 1$: this is the *multiexponentiation problem.* As with Pippenger's Multiproduct Algorithm, *Pippenger's Multiexponentiation Algorithm* is an algorithm for solving instances of the multiexponentiation problem while attempting to minimize the total number of multiplications required to compute the output sequence. The

[1] A *monoid* $(S, \cdot)$ is a set S together with a binary operation '$\cdot$' that satisfies three properties:

Closure: For all $a, b \in S$, the result of the operation $a \cdot b$ is also in S,

Associativity: For all $a, b, c \in S$, the equation $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ holds; and,

Identity: There exists an element $e \in S$ such that $e \cdot a = a \cdot e = a$ for all $a \in S$.

Typically we consider '$\cdot$' to be the operation of multiplication, however other operations are possible (most notably addition).

We define the power a^k as usual by the repeated application of '$\cdot$' to the element a; i.e., $a^1 = a$, $a^2 = a \cdot a$ and $a^k = \underbrace{a \cdot a \cdots a}_{k \text{ times}}$. By convention, $a^0 = e$. In additive notation it is common to write a^k as $k * a$.

Throughout this paper we will omit the symbol '$\cdot$' when writing $a \cdot b$, so that ab is synonymous with $a \cdot b$.

[2] Strictly speaking, Pippenger never gave an algorithm; instead, he gave a *constructive proof* that, in the worst-case, instances of the multiproduct problem require a number of two-operand multiplications that is asymptotically bounded above by $\frac{pq}{\log pq}(1 + o(1))$. It was previously known that worst-case instances of the problem are asymptotically bounded below by this same expression [8, 10], thus Pippenger's constructive proof yields an asymptotic formula for the best possible complexity of **any** algorithm that solves the multiproduct problem. The algorithm typically referred to as Pippenger's Multiproduct Algorithm is an algorithmic realization of his constructive proof.

algorithm works by reducing an instance of the multiexponentiation problem to an instance of the multiproduct problem, and then solves it using Pippenger's Multiproduct Algorithm. Pippenger proved in [13] that his algorithm is asymptotically optimal.[3]

Example 2. Suppose $\vec{\mathbf{x}} = \langle x_0, x_1, x_2 \rangle$ is given and let $\mathbf{M} = \begin{bmatrix} 3\,0\,2 \\ 1\,2\,0 \\ 0\,0\,2 \\ 3\,2\,1 \end{bmatrix}$. Then, the solution to the multiexponentiation problem is $y_0 = x_0^3 x_2^2$, $y_1 = x_0 x_1^2$, $y_2 = x_2^2$ and $y_3 = x_0^3 x_1^2 x_2$. Substituting the values $x_0 = 3$, $x_1 = 5$, $x_2 = 7$ (and working in $\mathbb{Z}$) yields: $y_0 = 1323$, $y_1 = 75$, $y_2 = 49$ and $y_3 = 4725$. ◊

The multiexponentiation problem has an obvious application in evaluating sparse multivariate polynomials; i.e., the terms in the input vector $\vec{\mathbf{x}}$ are taken to be the q indeterminates of a multivariate polynomial f and the terms in the output vector $\vec{\mathbf{y}}$ are taken to be the monomials in f. An efficient algorithm for the multiexponentiation problem can be used to evaluate f using a (nearly) minimal number of two-operand multiplications.

Example 3. Suppose we are given a sparse multivariate polynomial $f(x_0, x_1 \ldots, x_{q-1})$ in q indeterminates, and we wish to evaluate it at the point $(a_0, a_1, \ldots, a_{q-1})$. We use Pippenger's Multiexponentiation Algorithm to evaluate the indeterminate part of the p monomials comprising f at $(a_0, a_1, \ldots, a_{q-1})$:

$$\begin{gathered} a_0^{m_{0,0}} a_1^{m_{0,1}} \cdots a_{q-1}^{m_{0,(q-1)}} \\ a_0^{m_{1,0}} a_1^{m_{1,1}} \cdots a_{q-1}^{m_{1,(q-1)}} \\ \vdots \\ a_0^{m_{(p-1),0}} a_1^{m_{(p-1),1}} \cdots a_{q-1}^{m_{(p-1),(q-1)}}, \end{gathered}$$

and complete the evaluation of f by taking a linear combination of these values with the appropriate coefficients. ◊

Another interesting application emerges upon "taking the logarithm" of this problem [12]; i.e., computing linear transforms using the minimum number of two-operand additions. This problem arises in signal processing applications, where resource constrained devices (with no efficient multiply instruction) are called upon to compute linear transforms.

Example 4. Suppose we are given two matrices

$$\mathbf{M} = \begin{bmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,(q-1)} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,(q-1)} \\ \vdots & \vdots & \vdots & \vdots \\ m_{(p-1),0} & m_{(p-1),1} & \cdots & m_{(p-1),(q-1)} \end{bmatrix} \quad \text{and} \quad \vec{\mathbf{x}} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{q-1} \end{bmatrix}$$

[3] The same caveats apply to this statement as to the statement regarding Pippenger's Multiproduct Algorithm; that is, Pippenger actually gave a constructive proof that the worst-case complexity of the multiexponentiation problem is asymptotically bounded (both above and below) by $\frac{pq \lg k}{\lg(pq \lg k)}(1 + o(1))$ two-operand multiplications.

and we seek to compute the matrix product

$$\mathbf{M}\cdot\vec{\mathbf{x}} = \begin{bmatrix} m_{0,0} * x_0 + m_{0,1} * x_1 + \cdots + m_{0,(q-1)} * x_{q-1} \\ m_{1,0} * x_0 + m_{1,1} * x_1 + \cdots + m_{1,(q-1)} * x_{q-1} \\ \vdots \\ m_{(p-1),0} * x_0 + m_{(p-1),1} * x_1 + \cdots + m_{(p-1),(q-1)} * x_{q-1} \end{bmatrix}.$$

Using Pippenger's Multiexponentiation Algorithm (and the operation '+' instead of '·' as our monoid operation) we can compute this set of equations using a (nearly) minimal number of addition instructions. ◊

Note: The preceding example could be made more realistic by allowing negative coefficients, subtractions, and short shifts. In [12], Pippenger remarked that "these changes would not affect our analysis or results in any significant way", however we leave investigation of this idea to future work.

Finally, we note that many cryptographic protocols (especially in the setting of anonymous credential systems and zero-knowledge proofs) require the users to solve many instances of the multiexponentiation problem. One example of this is the author's own work on Nymbler [5, 6]; in this scheme, a user is required to evaluate large numbers of exponential equations, and then prove in zero knowledge that each equation was evaluated correctly. While the exponentiations themselves are reasonably expensive operations, the computation time (as well as bandwidth usage) is dominated by the zero-knowledge proofs that are needed to convince a semi-trusted third party that each exponentiation was computed faithfully. As originally presented in [5, 6], these proofs work by computing the exponents using the regular 'square-and-multiply' exponentiation algorithm, and then transmitting a zero-knowledge proof *for each step* of the algorithm. Each of these proofs, in turn, requires the verifier to compute a number of exponentiations. Therefore, any approach that can reduce the number of steps required to compute these exponents would lead to a decrease in the size of, and computational effort required to create and verify, these zero-knowledge proofs.

Before describing Pippenger's Multiproduct Algorithm and Pippenger's Multiexponentiation Algorithm in §2 and §3, we introduce the relevant theoretical framework in the following subsections.

1.1 Addition chains

An *addition chain* (of length s) for a positive integer n is a sequence of $s+1$ positive integers $(n_0, n_1, \ldots, n_s)$ such that:

- $n_0 = 1$;
- $n_s = n$; and,
- for all $0 < i \leq s$, there exists a pair (n_j, n_k) satisfying $n_i = n_j + n_k$ and $0 \leq j, k < i$.

Note: the pair (n_j, n_k) is not necessarily unique for a given i; we only require that *at least* one such pair exists.

Example 5. The sequence $(1, 2, 3, 6, 12, 24, 30, 31)$ is an addition chain of length 7 for the positive integer 31, since $2 = 1+1$, $3 = 2+1$, $6 = 3+3$, $12 = 6+6$, $24 = 12+12$, $30 = 24+6$, and $31 = 30+1$. $\lozenge$

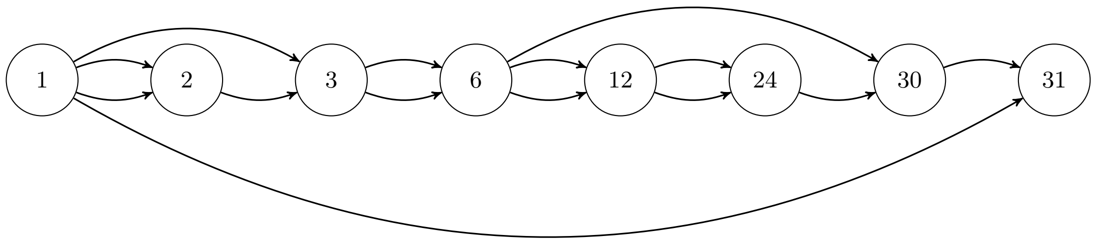


Fig. 1. A pictorial representation of the addition chain in Example 5.

See [7, 16] for the length of the shortest addition chain for some small values of n.

Addition-chain exponentiation is a technique for exponentiation by a positive integer that attempts to minimize the number of multiplications required; i.e., an addition chain is computed *for the exponent* and each element in the chain is evaluated by multiplying two of the earlier exponentiation results.

Example 6. Consider the problem of computing x^{15}. The naive approach of computing $x \cdot x \cdots x$ requires 14 multiplications, while using the ordinary square-and-multiply technique yields

$$x^{15} = x \cdot \left(x \cdot \left(x \cdot x^2\right)^2\right)^2, \text{ 6 multiplications.}$$

However, addition-chain exponentiation (with the optimal chain of $(1, 2, 3, 6, 12, 15)$) yields

$$x^{15} = x^3 \cdot \left(\left(x \cdot x^2\right)^2\right)^2, \text{ just 5 multiplications.}$$

$\lozenge$

Algorithm 1 gives a pseudocode implementation of addition chain exponentiation assuming that the addition chain is given as input.

Algorithm 1: Addition Chain Exponentiation (single output version)

```
Input: a base $x$;
    an addition chain $(m_0, m_1, \ldots, m_s)$; and,
    a sequence $\{(a_1, b_1), \ldots, (a_s, b_s)\}$, such that $m_0 = 1$,
    and for $0 < i \leq s$, $m_i = m_{a_i} + m_{b_i}$ with $0 \leq a_i, b_i < i$.
Output: $x^{m_s}$
1: define: $x_0 := x$
2: for $\big(i$ from $1$ to $s\big)$ do
3:     compute: $x_i \leftarrow x_{a_i} \cdot x_{b_i}$
4: end for
Return: $x_s$
```

Remark 1. Algorithm 1 corresponds to the special case of the multiexponentiation problem with $p = q = 1$ and $k \geq m_s$.

There are several known algorithms for computing such addition chains. Perhaps the most well-known algorithm is the ordinary square-and-multiply algorithm referred to in Example 1.1 (and presented in [9, §14.6.1] under the name *binary exponentiation*). A more efficient algorithm is Brauer's Algorithm [3]. When supplemented by several improvements as summarized in [2, 7], Brauer's chain can be computed recursively via the following formula:

$$T_k[n] = \begin{cases} 1, 2, 3, 5, 7, \ldots, 2^k - 1, n & \text{if } n < 2^{k+1} \text{ and } n \text{ is even} \\ T_k[n/2], n & \text{if } n \geq 2^{k+1} \text{ and } n \text{ is even} \\ T_k[n - (n \bmod 2^{\lceil \lg n \rceil})], n & \text{if } n < 2^{2k} \text{ amd } n \text{ is odd} \\ T_k[n - (n \bmod 2^k)], n & \text{if } n \geq 2^{2k} \text{ and } n \text{ is odd.} \end{cases} \tag{1}$$

A more general notion of addition chains considers a predefined ordered set $\{m_1, m_2, \ldots, m_p\}$ and asks for an addition chain for m_p that also contains each of $m_1, m_2, \ldots, m_{p-1}$. The problem of finding a minimal length addition chain of this type is known to be NP-complete [4]; nonetheless, there are several known techniques that can efficiently find *relatively short* addition chains that contain such a predefined set of values. In many settings (such as our own), this relaxed requirement is still quite useful.

Example 7. Consider the problem of computing x^3, x^6, x^{12} and x^{15}. The extremely naive approach of computing $(x \cdot x \cdot x)$, $(x \cdot x \cdot x \cdot x \cdot x \cdot x)$, $(x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x)$ and $(x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x)$ requires 31 multiplications, while the slightly less naive approach of computing $x^{15} = (x \cdots x)$ and remembering the intermediate results for x^3, x^6 and x^{12} reduces this to 14 multiplications. On the other hand, reusing our result from Example 1.1, we can compute all four of these values using just 5 multiplications, by remembering intermediate results in the computation

$$x^{15} = x^3 \cdot \Big(\big(x \cdot x^2\big)^2\Big)^2.$$

We also obtain x^2 as an intermediate result. Additionally, one can compute any one of x^4, x^5, x^7, x^8, x^9, x^{13}, x^{14}, x^{16}, x^{17}, x^{18}, x^{21}, x^{24}, x^{27} or x^{30} by using a single additional multiplication. $\Diamond$

Algorithm 2: Addition Chain Exponentiation (multiple output version)

```
Input: a base $x$;
    an addition chain $(m_0, m_1, \ldots, m_s)$;
    a sequence $\{(a_1, b_1), \ldots, (a_s, b_s)\}$, such that $m_0 = 1$,
    and for $0 < i \le s$, $m_i = m_{a_i} + m_{b_i}$ with $0 \le a_i, b_i < i$; and,
    a set of desired output indices $\{i_0, i_1, \ldots, i_{p-1}\}$, such that $i_{p-1} = s$.
Output: $\{x^{m_{i_0}}, x^{m_{i_1}} \ldots, x^{m_{i_{p-1}}}\}$
1: define: $x_0 := x$
2: for $(i$ from $1$ to $s)$ do
3:    compute: $x_i \leftarrow x_{a_i} \cdot x_{b_i}$
4: end for
Return: $\{x_{i_0}, x_{i_1}, \ldots, x_{i_{p-1}}\}$
```

Remark 2. Algorithm 2 corresponds to the special case of the multiexponentiation problem with $q = 1$ and $k \ge \max\{m_i \mid 1 \le i \le s\}$.

An even more general notion of an addition chain replaces scalars by q-element vectors of scalars. Given a predefined ordered set of vectors $\{\vec{v_1}, \vec{v_2}, \ldots, \vec{v_p}\}$ over $\mathbb{Z}$, an addition chain of length s begins with the q unit vectors

$$\begin{aligned}\vec{\mathbf{1}}_0 &= \langle 1, 0, \ldots, 0\rangle\\ \vec{\mathbf{1}}_1 &= \langle 0, 1, \ldots, 0\rangle\\ &\vdots\\ \vec{\mathbf{1}}_{q-1} &= \langle 0, 0, \ldots, 1\rangle,\end{aligned}$$

contains each of $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_p}$, and satisfies the chain condition: for all $q \le i < q + s$, there exists a pair $(\vec{v_j}, \vec{v_k})$ satisfying $\vec{v_i} = \vec{v_j} + \vec{v_k}$ and $0 \le j, k < i$.

Pippenger's Multiexponentiation Algorithm can be viewed as an algorithm for finding a short addition chain of this third type. The addition chain is then used to compute the desired products as described above.

Example 8. Consider the problem of computing $x^{15}y^9z$, $x^{12}y^6z^2$ and x^2z^{15}. By using, for example, the addition chain

$$\big(\langle 1,0,0\rangle,\ \langle 0,1,0\rangle,\ \langle 0,0,1\rangle,\ \langle 2,0,0\rangle,\ \langle 0,2,0\rangle,\ \langle 0,0,2\rangle,\ \langle 3,0,0\rangle,\\ \langle 0,3,0\rangle,\ \langle 0,0,3\rangle,\ \langle 3,3,0\rangle,\ \langle 6,0,0\rangle,\ \langle 0,0,6\rangle,\ \langle 0,0,12\rangle,\ \langle 0,0,15\rangle,\\ \langle 6,3,0\rangle,\ \langle 12,6,0\rangle,\ \langle 15,9,0\rangle,\ \langle \mathbf{15},\mathbf{9},\mathbf{1}\rangle,\ \langle \mathbf{12},\mathbf{6},\mathbf{2}\rangle,\ \langle \mathbf{2},\mathbf{0},\mathbf{15}\rangle\big)$$

we can compute all three desired values using just 17 two-operand multiplications. Note that computing the required powers of x, y and z independently (i.e., using the chains of minimum possible length that contain the predefined sets $\{2, 12, 15\}$ for x, $\{6, 9\}$ for y, and $\{1, 2, 15\}$ for z) requires at least $5 + 4 + 5 = 14$ multiplications, and combining the results to produce the desired monomials requires an additional 5 multiplications. This yields a total of at least $14+5 = 19 (> 17)$ multiplications. This example thus illustrates the imporant observation that an instance of the multiexponentiation problem can be solved using fewer multiplications than is theoretically possible by independently computing the powers for each base and multiplying the subproducts. ♢

Algorithm 3: Addition Chain Exponentiation (multiexponent version)

```
Input: a set of q bases $\{x_0, \ldots, x_{q-1}\}$;
    an addition chain of length-q vectors $(m_0, \ldots, \vec{m}_{(q-1)+s})$;
    a sequence $\{(a_q, b_q), \ldots, (a_{(q-1)+s}, b_{(q-1)+s})\}$, such that $(m_0, \ldots, m_{q-1}) = (\vec{\mathbf{1}}_0, \ldots, \vec{\mathbf{1}}_{q-1})$,
    and for $q \le i \le (q-1)+s$, $\vec{m}_i = \vec{m}_{a_i} + \vec{m}_{b_i}$ with $0 \le a_i, b_i < i$; and,
    a set of desired output indices $\{i_0, i_1, \ldots, i_{p-1}\}$, such that $i_{p-1} = (q-1)+s$.
Output: $\left\{x_0^{m_{i_0,0}} x_1^{m_{i_0,1}} \cdots x_{q-1}^{m_{i_0,(q-1)}}, \ldots, x_0^{m_{i_{(p-1)},0}} x_1^{m_{i_{(p-1)},1}} \cdots x_{q-1}^{m_{i_{(p-1)},(q-1)}}\right\}$,
    where $m_{i,j}$ is the $j^{\text{th}}$ component of $\vec{m}_i$.
1: define: $y_0 := x_0,\ y_1 := x_1, \ldots,\ y_{q-1} := x_{q-1}$
2: for $\big(i$ from $q$ to $(q-1)+s\big)$ do
3:     compute: $y_i \leftarrow y_{a_i} \cdot y_{b_i}$
4: end for
Return: $\{y_{i_0}, y_{i_1}, \ldots, y_{i_{p-1}}\}$
```

Remark 3. Algorithm 3 corresponds to the general case of the multiexponentiation problem with $k \ge \max\{m_{i,j} \mid 0 \le i < q+s \text{ and } 0 \le j < q\}$.

1.2 Some graph theory

A *directed graph* (also called a *digraph*) is an ordered pair of sets $G = (V, E)$ such that $E \subseteq V \times V$; that is, each element of E is an ordered pair of elements from V. The elements of V are called the *vertices* of G, while the elements of E are its *directed edges.*

A sequence $P = ((v_{0,1}, v_{0,2}), (v_{1,1}, v_{1,2}), \ldots, (v_{s,1}, v_{s,2}))$ of directed edges in G is called a *directed path* in G if $v_{i,2} = v_{i+1,1}$ for all $0 \le i < s$. In this case, we call P a directed path of length s from $v_{0,1}$ to $v_{s,2}$. A directed graph is called *acyclic* if it satisfies the following property: if there exists a directed path from $v_{0,1}$ to $v_{s,2}$, then there does not exist a directed path from $v_{s,2}$ to $v_{0,1}$.

All graphs we will make use of are assumed to be directed, and we will henceforth omit this word; thus, we will refer to directed graphs, directed edges and directed paths as graphs, edges and paths, respectively.

A graph G is called *bipartite* if V can be partitioned into two disjoint subsets A and B such that $E \subseteq A \times B$; that is, every edge in E contains a vertex from A as its first component, and a vertex from B as its second. Note that in a bipartite graph there is a one-to-one correspondence between paths and edges.

Suppose $G = (V, E)$ is a graph and $A, B \subseteq V$ such that $|A| = q$, $|B| = p$ and $A \cap B = \emptyset$ (note that we are not necessarily assuming that G is bipartite; indeed, $V - (A \cup B)$ may be nonempty and there may be edges that are not contained in $A \times B$). Denote the elements of A by a_i, for $0 \le i < q$, and the elements of B by b_j, for $0 \le j < p$. Then G is said to *realize* the matrix $\mathbf{M} \in \mathbb{Z}_{k+1}^{p\times q}$ if the number of distinct paths from a_i to b_j is $m_{i,j}$. In this case, we refer to the vertices in A as *input vertices* and the vertices in B as *output vertices.* A graph G that realizes $\mathbf{M}$ is said to be *minimal* if no proper subgraph of G realizes $\mathbf{M}$.

Lemma 1. *For any $\mathbf{M} \in \mathbb{Z}_2^{p\times q}$ there exists a unique (up to relabeling vertices) minimal bipartite graph G that realizes $\mathbf{M}$.*

Proof. That such a graph exists is easy to see: let $A = \{a_0, \ldots, a_{q-1}\}$, $B = \{b_0, \ldots, b_{p-1}\}$, and $V = A \cup B$, and set $E = \{(a_i, b_j) \mid m_{i,j} = 1\}$; then $G = (V, E)$ is the desired graph.

Now suppose that $G' = (V', E')$ is another minimal bipartite graph that realizes $\mathbf{M}$. Let $V' = A' \cup B'$ where $A' = \{a'_0, \ldots, a'_{p-1}\}$ is the set of inputs and $B' = \{b'_0, \ldots, b'_{p-1}\}$ is the set of outputs. Choose $(a'_i, b'_j) \in E'$. Then, since G' realizes $\mathbf{M}$, we know that $m_{i,j} = 1$ and therefore there exists and edge $(a_i, b_j) \in E$. Conversely, if $(a_i, b_j) \in E$, then since G' realizes $\mathbf{M}$ we know that there is an edge $(a'_i, b'_j) \in E'$. Thus, for each $0 \le i < q$, relabel a'_i as a_i and, for each $0 \le j < p$, relabel b'_j as b_j. Finally, observe that $G' = G$.

Example 9. Let $\mathbf{M} = \begin{bmatrix} 1\,0\,1 \\ 1\,1\,0 \\ 0\,0\,1 \\ 1\,1\,1 \end{bmatrix}$, and let $A = \{a_0, a_1, a_2\}$, $B = \{b_0, b_1, b_2, b_3\}$ and $V = A \cup B$. If $E = \{(a_0, b_0), (a_0, b_1), (a_0, b_3), (a_1, b_1), (a_1, b_3), (a_2, b_0), (a_2, b_2), (a_2, b_3)\}$, then $G = (V, E)$ is the bipartite graph realizing $\mathbf{M}$. $\Diamond$

By redefining the set of edges E to be a multiset[4], the above lemma can be generalized to apply to any matrix $\mathbf{M} \in \mathbb{Z}_{k+1}^{p\times q}$. To do this, we simply replace any single edge $e = (a_i, b_j)$ by $m_{i,j}$ distinct copies $e_1 = e_2 = \ldots = e_{m_{i,j}} = (a_i, b_j)$. Technically, the resulting structure is not strictly a graph; rather it is what we call a *multigraph.* We will not distinguish between graphs and multigraphs, and instead will assume that any graph may contain repeated edges.

Example 10. The illustration shown in Figure 1 (§1) is a pictorial representation of a non-bipartite graph that realizes the 1×1 matrix whose only entry is 31. $\Diamond$

[4] A *multiset* is a mathematical object much like a set, except that it allows repeated elements and the multiplicity of the occurrence of these elements is relevant. We will use $\uplus$ to denote the union of multisets where multiplicity is preserved.

Algorithm 4: Bipartite Graph Construction

```
Input: A matrix $\mathbf{M} \in \mathbb{Z}_{k+1}^{p \times q}$.
Output: The bipartite graph $G$ that realizes $\mathbf{M}$.
1:  set: $A \leftarrow \{a_j \mid 0 \leq j < q\}$
2:  set: $B \leftarrow \{b_i \mid 0 \leq i < p\}$
3:  set: $E \leftarrow \emptyset$                                  [Note: $E$ is a multiset]
4:  for $\big(i$ from $0$ to $p-1\big)$ do
5:     for $\big(j$ from $0$ to $q-1\big)$ do
6:        for $\big(k$ from $1$ to $m_{i,j}\big)$ do
7:           set: $E \leftarrow E \uplus \{(a_j, b_i)\}$
8:        end for
9:     end for
10: end for
Return: $G = (A \cup B, E)$
```

A function T that maps a graph G to another graph G' is called a *transformation*.

Definition 1. *Let* $\mathbf{M} \in \mathbb{Z}_{k+1}^{p \times q}$ *be given. Then a transformation* T *on* G *is called a* **realized invariant transformation** *with respect to* $\mathbf{M}$ *if, for all graphs* G *that realize* $\mathbf{M}$*, it follows that* $T(G)$ *also realizes* $\mathbf{M}$*.*

Example 11. Let $G = (\{a_0, a_1, b_0\}, \{(a_0, b_0), (a_0, b_0), (a_1, b_0)\})$ be the bipartite graph that realizes $\mathbf{M} = \begin{bmatrix} 2 & 1 \end{bmatrix}$. Then the transformation T that maps G to

$$G' = (\{a_0, a_1, x, b_0\}, \{(a_0, x), (a_0, x), (x, b_0), (a_1, b_0)\})$$

is a realized invariant transformation with respect to $\mathbf{M}$. To convince yourself of this, simply count the numbers of paths from a_0 to b_0 and from a_1 to b_0 in both graphs. $\Diamond$

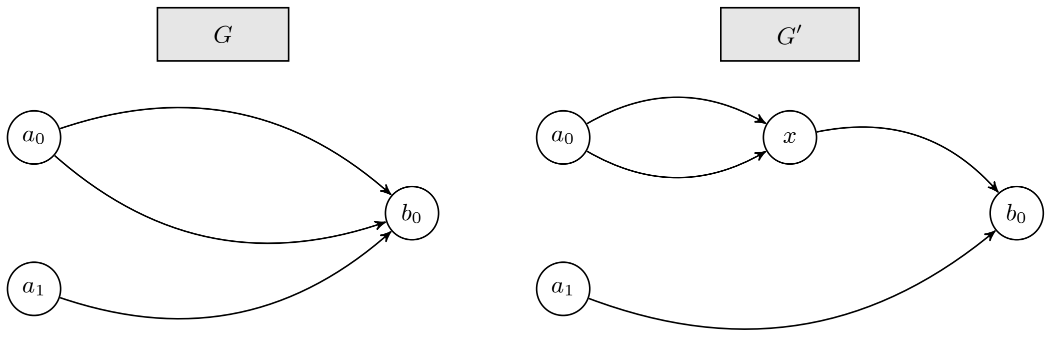


Fig. 2. A pictorial representation of the two graphs described in Example 11.

The *in-degree* of a vertex is the number of edges that lead into that vertex; i.e., the number of edges that contain that vertex as their second component. Equivalently, the in-degree of

$v \in V$ in the graph $G = (E, V)$ is equal to $|\{(a, v) \mid (a, v) \in E\}|$. The *out-degree* of a vertex is defined similarly.

Intuitively, the in-degree of each output vertex in the bipartite graph that realizes $\mathbf{M}$ corresponds to (one plus) the number of multiplications required to compute the corresponding output in the multiexponentiation problem using the naive approach. More generally, the number of multiplications required to evaluate the multiproduct problem associated with $\mathbf{M}$ according to a graph $G = (V, E)$ that realizes $\mathbf{M}$ is equal to $|E| - |V - A|$, where $A \subset V$ is the set of input vertices in G and $|E|$ is the cardinality of E including multiplicity. We call this value the *weight* of G. Thus, viewing the multiexponentiation problem in the setting of directed acyclic multigraphs reveals a potential approach to minimizing the length of the addition chain for $\mathbf{M}$ and, thereby, efficiently evaluating a problem instance. In particular, start with the trivially constructible bipartite graph G that realizes $\mathbf{M}$, apply a series of realization invariant transformations $T_0, T_1, \ldots, T_\ell$ such that each transformation T_i acts to reduce the weight of its image. Then, simply compute the output sequence according to the final output graph.

1.3 The ℓ Fuction and the L Function

Let $\ell(\mathbf{M})$ denote the minimum possible length of an addition chain for a given matrix $\mathbf{M} \in \mathbb{Z}_{k+1}^{p\times q}$. Let $L(p, q, k) = \max\left\{\ell(\mathbf{M}) \mid \mathbf{M} \in \mathbb{Z}_{k+1}^{p\times q}\right\}$ be the maximum value of $\ell(\mathbf{M})$ for any matrix $\mathbf{M} \in \mathbb{Z}_{k+1}^{p\times q}$. Pippenger's multiproduct and multiexponentiation algorithms are the end product of a long line of incremental work on studying the asymptotic behaviour of $L(p, q, k)$. Before Pippenger's work, many of the special cases were addressed in the literature.

The first and most obvious of these is the case of $L(1, 1, k)$; i.e., that of computing a regular exponentiation. In 1937, Scholz [15] observed that

$$\lg k \le L(1, 1, k) \le 2 \lg k.$$

This follows from the regular square-and-multiply approach (cf. [9, §14.6.1]). If $k = 2^n$ is a power of 2 then exactly $\lg k = n$ multiplications are required with this method; at the other extreme, if $k = 2^n - 1$ is a power of 2 less one, then exactly $2 \lg k - 2 = 2(n-1)$ multiplications are required ($n-1$ squarings interleaved with $n-1$ multiplications). Later, Brauer [3] proposed a chain construction method that tightened this bound to

$$\lg k \le L(1, 1, k) \le \lg k \left(1 + \frac{1}{\log_{10} \log_{10} k}\right) + 2(\log_{10} k)^{\log_{10} 2}.$$

The addition chain built from the sequence given by Equation 1 (pg. 6) is based on Brauer's construction. See [2] or [7] for additional details.

Later, Straus [17] showed for, for any fixed q,

$$L(1, q, k) \sim \lg k$$

and Yao [18] showed that, for any fixed p,

$$L(p, 1, k) \sim \lg k$$

when $p = o(\lg \lg k)$.

Lupanov [8] proved that if

1. $\lg p = o(q)$ and $\lg q = o(p)$; and,
2. $\frac{\lg p}{\lg q}$ tends to zero or ∞,

then

$$L(p, q, 1) \sim \frac{pq}{\lg pq}.$$

Generalizing an idea of Nechiporuk [10], Pippenger [12, 13] removed the second condition and showed that, if the first condition is generalized to $p = (k+1)^{o(q)}$ and $q = (k+1)^{o(p)}$,

$$L(p, q, k) \sim \min\{p, q\} \log k + \frac{pq \lg (k+1)}{\lg (\lg (pq \lg (k+1)))} + o\left(\frac{pq \lg (k+1)}{\lg (\lg (pq \lg (k+1)))}\right).$$

His proof was constructive, and forms the basis of the algorithms presented in the remainder of this paper.

1.4 Cost model

The complexity analysis of the algorithm uses the following cost model:

- The zero vector,
$$\vec{\mathbf{0}} = \langle 0, \ldots, 0 \rangle$$
and the q unit vectors,
$$\begin{aligned} \vec{\mathbf{1}}_0 &= \langle 1, 0, \ldots, 0 \rangle \\ \vec{\mathbf{1}}_1 &= \langle 0, 1, \ldots, 0 \rangle \\ &\vdots \\ \vec{\mathbf{1}}_{q-1} &= \langle 0, 0, \ldots, 1 \rangle, \end{aligned}$$
are available at no cost.
- The sum of any two (not necessarily distinct) vectors that were previously computed (including the unit vectors) is available at a cost of 1.

Note that the entries in these vectors correspond to exponents in the output (i.e., summing two vectors is equivalent to computing a single multiplication). Thus, the cost of the algorithm corresponds in an obvious way to the number of multiplications required to compute the output sequence; i.e., to the length of the addition chain used to compute the output sequence.

Of course, not all multiplications are created equal, and minimizing exponentiation time does not necessarily amount to minimizing the number of multiplications. For example, computing the square of a number is generally faster than computing the product of two different numbers (especially if the magnitude of the two multiplicands is dramatically different); on the other hand, multiplication by 2^k in a binary computer (or, more generally, by b^k in any base-b number system) is simply a matter of shifting each digit to the left by k position and filling the resulting void with zeros. There are also well-known algebraic techniques that can be employed when computing exponents modulo a prime or a composite of known factorization. We will not pursue this line of thought any further and we refer the reader to Bernstein's paper [2] for additional discussion on the matter. We do, however, note that in practice the strategy of minimizing addition chain length is generally an effective one.

Also note that the cost model does not take into consideration the overhead due to decomposing the input problem, storing intermediate values, partitioning and grouping values, etc. When dealing with inputs of sufficiently large bit-length, this cost model is likely to be a reasonable one as the costs associated with these operations will be negligible; however, for many practical applications this overhead may potentially contribute noticeably to the speed of computations.

2 Pippenger's Multiproduct Algorithm

Using the ideas and terminology that we developed in the first section, we now present Pippenger's Multiproduct Algorithm.

Conceptually, the algorithm is best understood as using a combination of iteration and recursion to obtain a result. Each iteration splits the problem into two subproblems; one subproblem is solved by applying Pippenger's Multiproduct Algorithm recursively, while the other subproblem is further reduced into two more subproblems in the next iteration (or, eventually, solved directly). Each iteration is parameterized by the following (all nonnegative integers):

- the current *iteration index*, denoted by i;
- the *clumping factor*, denoted by α_i; and,
- the *grouping factor*, denoted by β_i.

Additionally, the algorithm is parameterized by the *iteration limit* (a positive integer), denoted by ℓ; the *initial clumping factor* (also a positive integer), denoted by c; and, an optional (boolean) *toggle*, denoted by t. The clumping and grouping factors α_i, β_i and c are subject to the following constraints:

- $c \geq 2$;
- $\beta_i \geq 2$; and,
- $\alpha_i \geq \beta_i$.

Typically, c will be logarithmic in pq, β_i will be exactly 2, and α_i will be fairly large (only a polylogarithmic factor smaller than p). We will return to these parameters in §4.

In §2.1, we first present Pippenger's Multiproduct Algorithm as it applies to a graph that realizes a matrix $\mathbf{M}$ (which is how Pippenger originally presented it in [12,13]) Then, in §2.2, we translate our graph-theoretic formulations into pseudo-code that can be applied to directly solve instances of the multiproduct problem.

2.1 Graph-theoretic formulation

Let $\mathbf{M} \in \mathbb{Z}_2^{p \times q}$ and let $G_{-1} = (A_{-1} \cup B_{-1}, E_{-1})$ be the minimal bipartite graph realizing $\mathbf{M}$, where A_{-1} is the set of input vertices and B_{-1} is the set of output vertices. On the i^{th} iteration, the algorithm applies a realized invariant transformation (with respect to $\mathbf{M}$) T_i to $G_{i-1} = (V_{i-1}, E_{i-1})$, to produce the graph $G_i = (V_i, E_i)$.

In the description that follows, we will use a_S to refer to an input vertex in A_i corresponding to a set S, and b_X to refer to an output vertex in B_i corresponding to a set X. That is, the original graph consists of: q input vertices $a_{S_0}, \ldots, a_{S_{q-1}}$, each of which is associated with a singleton set $S_i = \{x_i\}$; and p output vertices $b_{X_0}, \ldots, b_{X_{p-1}}$, each of which is associated with a union of input set $X_i = S_{i_1} \cup \cdots \cup S_{i_n}$. The i^{th} transformation, T_i, adds new vertices in A_i or B_i, and replaces edges in E_{i-1} by new edges in E_i, as follows (note that we omit consideration of the optional toggle t from our discussion):

if i = 0: (Input Partitioning)

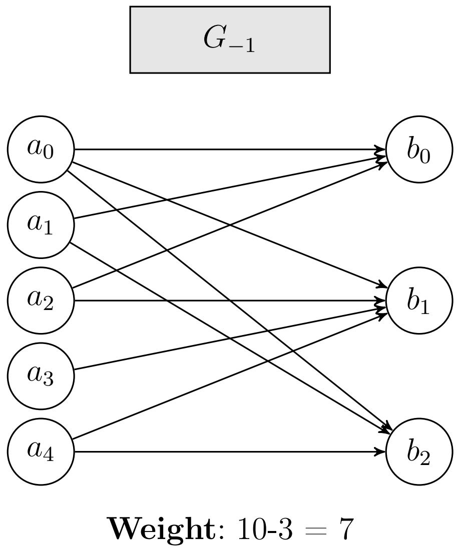
```
1:  set: $A_0 := \emptyset$
2:  set: $B_0 := \emptyset$
3:  set: $H_0 := \emptyset$
4:  set: $F_0 := \emptyset$
5:  partition: $A_{-1}$ into $\lceil \frac{q}{c} \rceil$ parts $P$, each of size at most $c$
6:  for each (partition $P$) do
7:     for each (subset $S \subseteq P$) do
8:        if ($|S| \geq 2$ and $S \subseteq X$ for some output $X$) then
9:           insert: $A_0 \leftarrow A_0 \cup \{a_S\}$
10:          for each ($x \in S$) do
11:             insert: $H_0 \leftarrow H_0 \cup \{(a_{\{x\}}, a_S)\}$          $[(a_{\{x\}}, a_S) \in A_{-1} \times A_0]$
12:          end for
13:       end if
14:    end for
15:    for each (output $b_X \in B_{-1}$) do
16:       let: $P_{b_X} := \{a_S \in P \mid (a_S, b_X) \in E_{-1}\}$          $[P_{b_X} \subseteq P]$
17:       if ($P_{b_X} \neq \emptyset$) then
18:          insert: $F_0 \leftarrow F_0 \cup \{(a_{P_{b_X}}, b_X)\}$          $[(a_{P_{b_X}}, b_X) \in (A_0 \cup A_{-1}) \times B_{-1}]$
19:       end if
20:    end for
21: end for
22: set: $E_0 := H_0 \cup F_0$
         $V_0 := (A_{-1} \cup A_0) \cup (B_{-1} \cup B_0)$
23: then $G_0 := (V_0, E_0)$
```

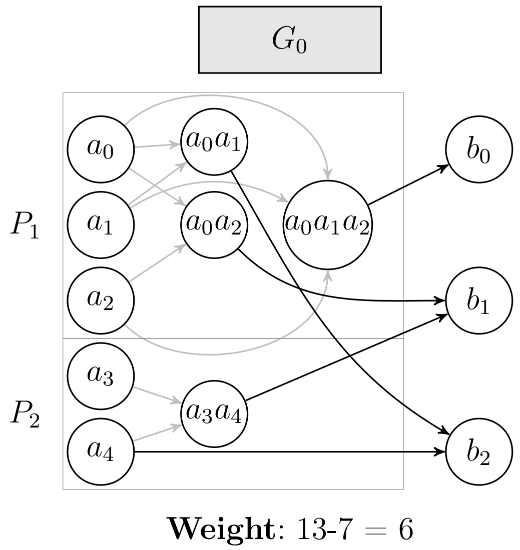
Intuitively, input partitioning replaces all edges from a partition P of inputs to an output X with a single edge. The vertices in A_{-1} are the original inputs and those in B_{-1} are the original outputs, while those in A_0 are called *auxiliary inputs* (and B_0 is empty). The edges in H_0 are called *auxiliary edges*, and those in F_0 are called *active edges*.

The bipartite graph $G''_0 = (A_{-1} \cup A_0, H_0)$ can be viewed as a smaller multiproduct problem with A_{-1} as inputs and A_0 as outputs; solve this subproblem recursively using Pippenger's Multiproduct Algorithm. The graph $G'_0 = \big((A_{-1} \cup A_0) \cup (B_{-1} \cup B_0), F_0\big)$ can also be viewed as a smaller subproblem with $A_{-1} \cup A_0$ as inputs and $B_{-1} \cup B_0$ as outputs; the remainder of the algorithm will deal with solving this second subproblem.

A simple pictorial example of input partitioning is given in Example 12. The structure of the graph after input partitioning is given in Figure 3.

Example 12. (Graph-theoretic Input Partitioning)





This diagram illustrates input partitioning. The gray edges in the diagram on the right are auxiliary edges, while the black edges are active edges. ◊

$$\begin{tikzcd}
B_{-1} & B_0 \\
A_{-1} \arrow[u, "F_0"] \arrow[r, "H_0"'] & A_0 \arrow[ul, "F_0"']
\end{tikzcd}$$

Fig. 3. This figure shows the structure of the graph after the $(i = 0)^{\text{th}}$ transformation. The four boxes represent the four sets of vertices; an arrow from one box to another indicates that there are (probably) directed edges from vertices in the originating box to those in the distination box, with the label on that arrow specifying the set in which these edges are contained. A black arrow indicates that the edges are active edges, while a gray arrow indicates that they are auxiliary edges.

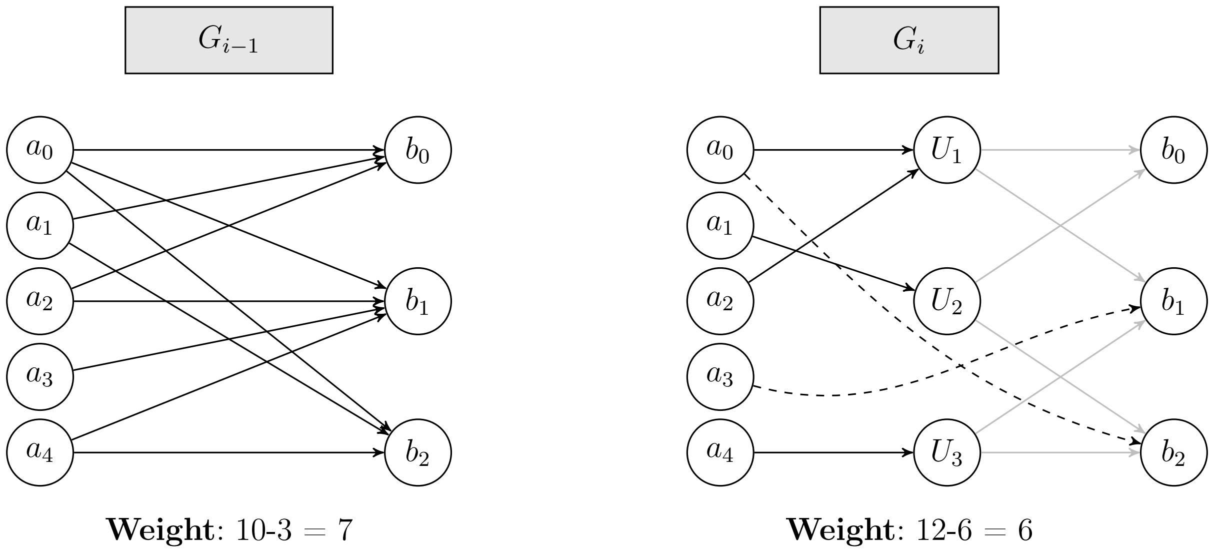
```
if 1 ≤ i < ℓ, and i is odd: (Output Clumping)
1: set: $A_i := \emptyset$
2: set: $B_i := \emptyset$
3: set: $I_i := \emptyset$
4: set: $H_i := \emptyset$
5: set: $F_i := \emptyset$
6: partition: $B_{i-2}$ into $\left\lceil \frac{|B_{i-2}|}{\alpha_i} \right\rceil$ parts $P$, each of size at most $\alpha_i$
7: for each (partition $P$) do
8:    let: $\mathbf{U} = \{U_1, \ldots, U_s\}$ be all size-$\beta_i$ subsets of $P$        $[s \le \binom{\alpha_i}{\beta_i}]$
9:    for $\big(S_i \in A_{-1} \cup \cdots \cup A_i\big)$ do
10:      let: $S(i) := \{X \mid (a_{S_i}, b_X) \in F_{i-1}\}$
11:      decompose: $S(i)$ as a disjoint union of $U_j \in \mathbf{U}$, plus $k \le \alpha_i - 1$ ‘overflow’ outputs
12:      for each (overflow output $X$ in the chosen decomposition of $S(i)$) do
13:         insert: $I_i \leftarrow I_i \cup \{(a_{S_i}, b_X)\}$        $[(a_{S_i}, b_X) \in (A_{-1} \cup \cdots \cup A_i) \times B_{i-2}]$
14:      end for
15:   end for
16: end for
17: for each (size-$\beta_i$ subset $U_j \in \mathbf{U}$) do
18:    set: $T(U_j) := \{S_i \mid U_j$ is used in the chosen decomposition of $S(i)\}$
19:    if $\big(T(U_j) \ne \emptyset\big)$ then
20:       insert: $B_i \leftarrow B_i \cup \{b_{U_j}\}$
21:       for each $\big(S_i \in T(U_j)\big)$ do
22:          insert: $F_i \leftarrow F_i \cup \{(a_{S_i}, b_{U_j})\}$        $[(a_{S_i}, b_{U_j}) \in (A_{-1} \cup \cdots \cup A_i) \times B_i]$
23:       end for
24:       for each $\big(X \in U_j\big)$ do
25:          insert: $H_i \leftarrow H_i \cup \{(b_{U_j}, b_X)\}$        $[(b_{U_j}, b_X) \in B_i \times B_{i-2}]$
26:       end for
27:    end if
28: end for
29: set: $E_i := (I_0 \cup \cdots \cup I_i) \cup (H_0 \cup \cdots \cup H_i) \cup F_i$
         $V_i := (A_{-1} \cup \cdots \cup A_i) \cup (B_{-1} \cup \cdots \cup B_i)$
30: then $G_i := (V_i, E_i)$
```

The vertices in A_{-1} are the original inputs, and those in B_{-1} are the original outputs, while the vertices in B_i (and $B_0, \ldots, B_{i-2}$) are called *auxiliary outputs* (and B_{i-1} is empty), and those in $A_0, \ldots, A_i$ are the auxiliary inputs. The edges in I_i are called *residual edges*, while those in H_i are the auxiliary edges, and those in F_i are the active edges.

The graph $G''_i = \big((A_{-1} \cup \cdots \cup A_i \cup B_i) \cup B_{i-2}, I_i \cup H_i\big)$ can be viewed as a smaller subproblem with $A_{-1} \cup \cdots \cup A_i \cup B_i$ as inputs and B_{i-2} as outputs. Similarly, the graph $G'_i = \big((A_{-1} \cup \cdots \cup A_i) \cup B_i, F_i\big)$ can be viewed as a smaller subproblem with $A_{-1} \cup \cdots \cup A_i$ as inputs and B_i as outputs. The first of these subproblems is solved recursively using Pippenger’s Multiproduct Algorithm; first, however, the remainder of the algorithm is applied to solve the second subproblem.

A simple pictorial example of output clumping is given in Example 13. The structure of the graph after the first application of output clumping is given in Figure 4.

Example 13. (Graph-theoretic Output Clumping)



This diagram illustrates output clumping. The gray edges in the diagram on the right are auxiliary edges, while the dashed edges are residual edges and the black edges are active edges.

◊

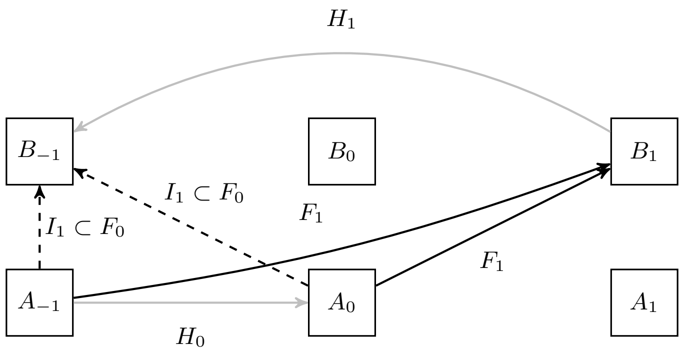


Fig. 4. This figure shows the structure of the graph after the $(i = 1)^{\mathsf{th}}$ transformation (i.e., after the first application of output clumping). The six boxes represent the six sets of vertices; an arrow from one box to another indicates that there are (probably) directed edges from vertices in the originating box to those in the destination box, with the label on that arrow specifying the set in which these edges are contained. A black arrow indicates that the edges are active edges, a gray arrow indicates that they are auxiliary edges, and a dashed arrow indicates that they are residual edges.

if $2 \le i < \ell$, and i is even: (Input Clumping)

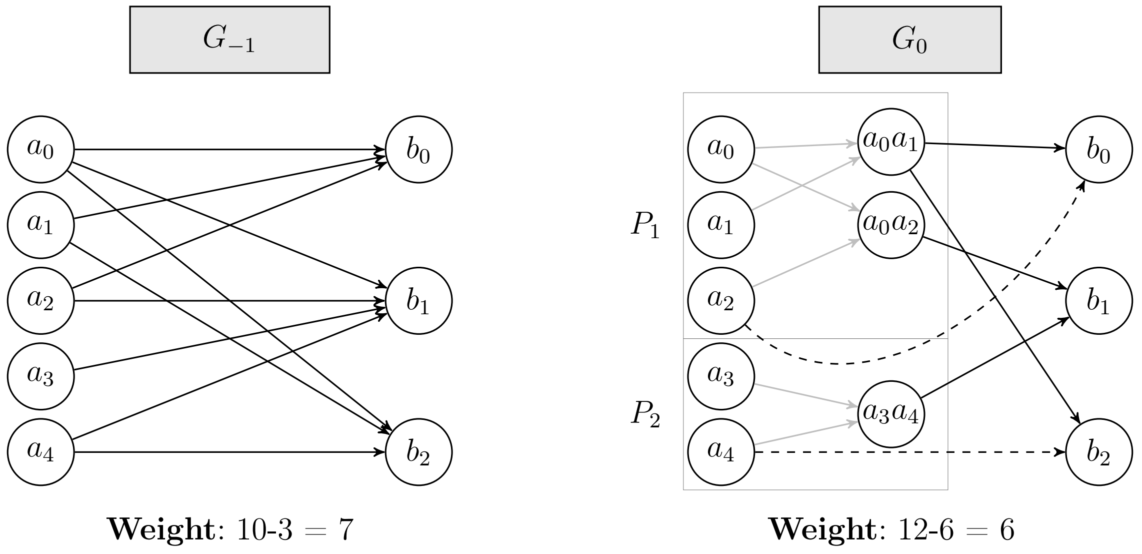
```
1: set: $A_i := \emptyset$
2: set: $B_i := \emptyset$
3: set: $I_i := \emptyset$
4: set: $H_i := \emptyset$
5: set: $F_i := \emptyset$
6: partition: $A_{-1} \cup \cdots \cup A_{i-1}$ into $\left\lceil \frac{|A_{-1} \cup \cdots \cup A_{i-1}|}{\alpha_i} \right\rceil$ parts $P$, each of size at most $\alpha_i$
7: for each (partition $P$) do
8:   let: $\mathbf{S} = \{S_1, \ldots, S_t\}$ be all size-$\beta_i$ subsets of $P$        $[t \le \binom{\alpha_i}{\beta_i}]$
9:   for each (size-$\beta_i$ subset $S_j \in \mathbf{S}$) do
10:    if $S_j \subseteq X$ for some output $X$ then
11:      insert: $A_i \leftarrow A_i \cup \{a_{S_j}\}$
12:      for each (element $x \in S_j$) do
13:        insert: $H_i \leftarrow H_i \cup \{(a_{\{x\}}, a_{S_j})\}$        $[(a_{\{x\}}, a_{S_j}) \in (A_{-1} \cup \cdots \cup A_{i-1}) \times A_i]$
14:      end for
15:      for each (output $b_X \in B_{i-1}$) do
16:        if $((S_j \subseteq X)$ and $(S_j \cap S_k = \emptyset$ for all $a_{S_k} \in A_i$ with $(a_{S_k}, b_X) \in F_i))$ then
17:          insert: $F_i \leftarrow F_i \cup \{(a_{S_j}, b_X)\}$        $[(a_{S_j}, b_X) \in A_i \times B_{i-1}]$
18:        end if
19:      end for
20:    end if
21:  end for
22: end for
23: for each (output $b_X \in B_{i-1}$) do
24:   let: $R_X = X - \bigcup\{S_j \mid (a_{S_j}, b_X) \in F_i\}$        $[|R_X| \le \alpha_i - 1$ for all $b_X \in B_{i-1}]$
25:   for each (element $x \in R_X$) do
26:     insert: $I_i \leftarrow I_i \cup \{(a_{\{x\}}, b_X)\}$        $[(a_{\{x\}}, b_X) \in (A_{-1} \cup \cdots \cup A_{i-1}) \times B_{i-1}]$
27:   end for
28: end for
29: set: $E_i := (I_0 \cup \cdots \cup I_i) \cup (H_0 \cup \cdots \cup H_i) \cup F_i$
         $V_i := (A_{-1} \cup \cdots \cup A_i) \cup (B_{-1} \cup \cdots \cup B_i)$
30: then $G_i := (V_i, E_i)$
```

The vertices in A_{-1} are the original inputs, and those in B_{-1} are the original outputs, while the vertices in A_i (and in $A_0, \ldots, A_{i-1}$) are the auxiliary inputs, and those in $B_0, \ldots, B_i$ are the auxiliary outputs. Again, the edges in I_i are the residual edges, those in H_i are the auxiliary edges, and those in F_i are the active edges.

The graph $G_i'' = \big((A_{-1} \cup \cdots \cup A_{i-1}) \cup A_i, H_i\big)$ can be viewed as a smaller multiproduct problem with $A_{-1} \cup \cdots \cup A_{i-1}$ as inputs and A_i as outputs; solve this subproblem recursively using Pippenger's Multiproduct Algorithm. Similarly, the graph $G_i' = \big((A_{-1} \cup \cdots \cup A_i) \cup (B_{-1} \cup \cdots \cup B_i), I_i \cup F_i\big)$ can be viewed as a smaller multiproduct problem with $A_{-1} \cup \cdots \cup A_i$ as inputs and $B_{-1} \cup \cdots \cup B_i$ as outputs. The remainder of the algorithm will deal with solving this latter subproblem.

A simple pictorial example of input clumping is given in Example 14. The structure of the graph after the first application of input clumping is given in Figure 5.

Example 14. (Graph-theoretic Input Clumping)



This diagram illustrates input clumping. The gray edges in the diagram on the right are auxiliary edges, while the dashed edges are residual edges and the black edges are active edges. ◊

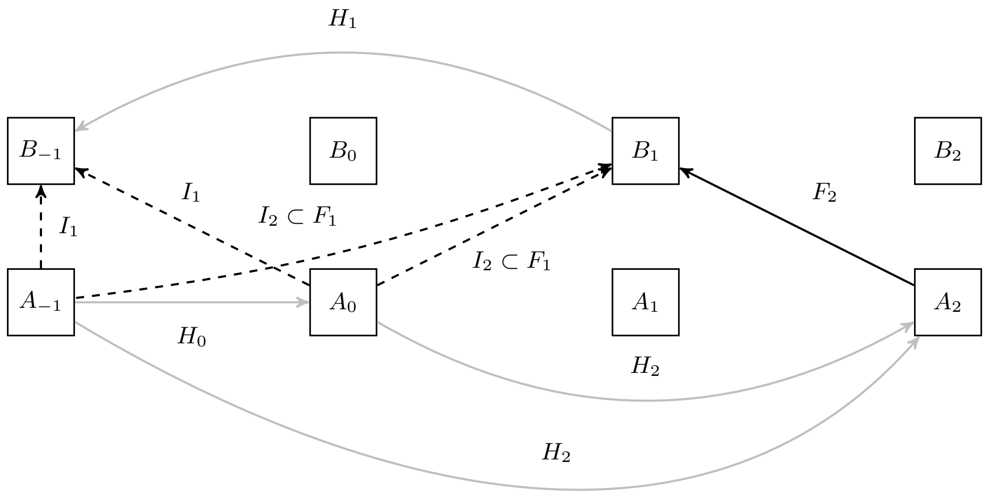


Fig. 5. This figure shows the structure of the graph after the $(i = 2)^{\mathsf{th}}$ transformation (i.e., after the first application of input clumping). The eight boxes represent the eight sets of vertices; an arrow from one box to another indicates that there are (probably) directed edges from vertices in the originating box to those in the destination box,with the label on that arrow specifying the set in which these edges are contained. A black arrow indicates that the edges are active edges, a gray arrow indicates that they are auxiliary edges, and a dashed arrow indicates that they are residual edges.

2.2 Pseudo-code formulation

We now give a pseudo-code description, in the form of five subroutines, which implements Pippenger's Multiproduct Algorithm directly (i.e., without transforming the problem into a graph). One of the subroutines, `GetParams`, is not presented; see §4 for further discussion on this subroutine. Again, we also omit consideration of the optional toggle t, although we remark that its inclusion would be pretty straightforward. In particular, the effect of having $t =$ `true` would be to replace the initial call to `InputPartition` by a call to an analogous subroutine, `OutputPartition`, and then to swap the order of `OutputClump` and `InputClump` for the remainder of the algorithm.

Following each pseudo-code algorithm is a fully worked example that illustrates the effect of that algorithm on a set of sample inputs and outputs. For ease of illustration, some portions of the intermediate work in these examples more closely follows the graph-theoretic versions of the algorithms presented in the last section; the outputs, however, are in all cases consistent with the output of the algorithms as presented in this section.

Algorithm 5: `MultiProd`($\mathbf{x}$, $\mathbf{y}$)

```
Input: the set of inputs $\mathbf{x}$; and,
    the set of desired outputs $\mathbf{y}$.                      [$\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})$]
Output: the set of outputs $\mathbf{X}$.                           [$\mathbf{X} = \left\{\prod_{x \in y} \mid y \in \mathbf{y}\right\}$]
1: call: $\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}, c \leftarrow$ GetParams($\mathbf{x}$, $\mathbf{y}$)     [$\boldsymbol{\alpha}=\{\alpha_1, \ldots, \alpha_\ell\}$, $\boldsymbol{\beta}=\{\beta_1, \ldots, \beta_\ell\}$]
2: call: $\mathbf{X} \leftarrow$ComputeMultiProd(0, $\mathbf{x}$, $\mathbf{y}$, $\ell$, $c$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)     [Algorithm 6]
Return: $\mathbf{X}$
```

Algorithm 6: `ComputeMultiProd`(i, $\mathbf{x}$, $\mathbf{y}$, ℓ, c, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)

```
Input: the current iteration index $i$;                             [$i \geq 0$]
    the set of inputs $\mathbf{x}$;
    the set of desired outputs $\mathbf{y}$;                        [$\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})$]
    the iteration limit $\ell$;                                     [$\ell \geq 0$]
    the initial clumping factor $c$;                                [$c \geq 2$]
    the clumping factors $\boldsymbol{\alpha}$; and,                [$= \{\alpha_0, \ldots, \alpha_\ell\}$]
    the grouping factors $\boldsymbol{\beta}$.                      [$= \{\beta_0, \ldots, \beta_\ell\}$]
Output: the set of outputs $\mathbf{X}$.                           [$\mathbf{X} = \left\{\prod_{x \in y} \mid y \in \mathbf{y}\right\}$]
1: if $(i = \ell)$ then
2:     return NaiveMultiply($\mathbf{x}$, $\mathbf{y}$)                                  [Algorithm 10]
3: else if $(i = 0)$ then
4:     call: $\mathbf{x}', \mathbf{y}' \leftarrow$InputPartition($\mathbf{x}$, $\mathbf{y}$, $c$)                  [Algorithm 7]
5:     return ComputeMultiProd(1, $\mathbf{x}'$, $\mathbf{y}'$, $\ell$, $c$, $\boldsymbol{\beta}$, $\boldsymbol{\alpha}$)            [Algorithm 6]
6: else if $(i$ is odd$)$ then
7:     call: $\mathbf{y}', \mathbf{y}'' \leftarrow$OutputClump($\mathbf{x}$, $\mathbf{y}$, $\alpha_i$, $\beta_i$)          [Algorithm 8]
8:     call: $\mathbf{x}'' \leftarrow$ComputeMultiProd($i+1$, $\mathbf{x}$, $\mathbf{y}'$, $\ell$, $c, \boldsymbol{\alpha}, \boldsymbol{\beta}$)      [Algorithm 6]
9:     return MultiProd($\mathbf{x} \cup \mathbf{x}''$, $\mathbf{y}''$)                           [Algorithm 5]
10: else if $(i$ is even$)$ then
11:     call: $\mathbf{x}', \mathbf{y}' \leftarrow$InputClump($\mathbf{x}$, $\mathbf{y}$, $\alpha_i$, $\beta_i$)         [Algorithm 9]
12:     return ComputeMultiProd($i+1$, $\mathbf{x}'$, $\mathbf{y}'$, $\ell$, $c$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)      [Algorithm 6]
13: end if
```

Remark 4. On Line 9, Algorithm 6 recursively calls `MultiProd`, as do `InputPartition` and `InputClump` on lines 12 and 24, respectively. We need to be sure that at some point a base case is reached; i.e., that each recursive subproblem is somehow smaller than its predecessor, until it is eventually small enough to be solved directly. The recursive call on Line 9 uses the original set of outputs, and a new set of inputs that is strictly smaller than the original set of inputs. The recursive call on Line 12 of `InputPartition` uses a subset of the original set of inputs, and a new set of outputs that is strictly smaller than the original set of outputs. Finally, the recursive call on Line 24 of `InputClump` is similar to the call in `InputPartition`, except that the set of outputs is even more restricted (i.e., is probably smaller). Thus, in all instances, the recursive call does indeed address a smaller instance of the problem. When the problem instance becomes sufficiently small, `GetParams` returns $\ell = 0$, and the recursive call to `MultiProd` simply results in a call to `NaiveMultiply`, which is the base case.

Algorithm 7: `InputPartition`($\mathbf{x}$, $\mathbf{y}$, c)

```
Input: the set of inputs $\mathbf{x}$;
    the set of desired outputs $\mathbf{y}$; and                  [$\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})$]
    the initial clumping factor $c$.                              [$c \geq 2$]
Output: the set of new inputs $\mathbf{x}'$; and,
    the set of new outputs $\mathbf{y}'$.                         [$\mathbf{y}' \subseteq \mathcal{P}(\mathbf{x}')$]
 1: set: $\mathbf{x}' := \emptyset$
 2: for each $(y_j \in \mathbf{y})$ do
 3:     set: $y'_j := \emptyset$
 4: end for
 5: for $\left(i = 0 \text{ to } \left\lceil \frac{|\mathbf{x}|}{c} \right\rceil - 1\right)$ do
 6:     set: $Y_i := \emptyset$
 7:     set: $P_i := \{x_{i \cdot c}, \ldots, x_{i \cdot c + (c-1)}\}$
 8:     for each $(y_j \in \mathbf{y})$ do
 9:         define: $x_{i,j} := P_i \cap y_j$
10:         set: $Y_i \leftarrow Y_i \cup \{x_{i,j}\}$
11:     end for
12:     call: $X_i :=$ MultiProd($P_i$, $Y_i$)                    [Algorithm 5]
13:     for each $(y_j \in \mathbf{y})$ do
14:         set: $x'_{i,j} := \{x \in X_i \mid x = \prod x_{i,j}\}$
15:         add: $y'_j \leftarrow y'_j \cup x'_{i,j}$
16:     end for
17:     compute: $\mathbf{x}' \leftarrow \mathbf{x}' \cup X_i$
18: end for
19: compute: $\mathbf{y}' := \{y'_0, \ldots, y'_{|\mathbf{y}|-1}\}$
Return: $\mathbf{x}'$, $\mathbf{y}'$
```

Example 15. (Algorithm 7, Input Partitioning) Let the inputs be $\vec{\mathbf{x}} = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$; the outputs be $\vec{\mathbf{y}} = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9\}$, where

$$y_0 = \{x_0, x_2, x_3, x_4, x_5, x_6\}, \quad y_1 = \{x_0, x_1, x_2, x_4\}, \quad y_2 = \{x_0, x_3, x_4, x_5, x_6, x_7\},$$
$$y_3 = \{x_1, x_2, x_3, x_4, x_6, x_7\}, \quad y_4 = \{x_0, x_1, x_3, x_4, x_5, x_7\}, \quad y_5 = \{x_0, x_1, x_2, x_3, x_6, x_7\},$$
$$y_6 = \{x_0, x_1, x_2, x_3, x_4, x_5, x_7\}, \quad y_7 = \{x_0, x_2, x_4, x_6\}, \quad y_8 = \{x_1, x_2, x_3, x_7\},$$
$$y_9 = \{x_0, x_2, x_4, x_5, x_6\};$$

and, let $c = 3$. We first partition the inputs into $\left\lceil \frac{8}{3} \right\rceil = 3$ groups:

$$P_0 = \{x_0, x_1, x_2\}, \qquad P_1 = \{x_3, x_4, x_5\}, \qquad P_2 = \{x_6, x_7\},$$

Next we compute $x_{i,j}$ for each pair of partition and output above:

$$\begin{array}{lll}
x_{0,0} = \{x_0, x_2\} & x_{0,1} = \{x_0, x_1, x_2\} & x_{0,2} = \{x_0\} \\
x_{0,3} = \{x_1, x_2\} & x_{0,4} = \{x_0, x_1\} & x_{0,5} = \{x_0, x_1, x_2\} \\
x_{0,6} = \{x_0, x_1, x_2\} & x_{0,7} = \{x_0, x_2\} & x_{0,8} = \{x_1, x_2\} \\
x_{0,9} = \{x_0, x_2\} & x_{1,0} = \{x_3, x_4, x_5\} & x_{1,1} = \{x_4\} \\
x_{1,2} = \{x_3, x_4, x_5\} & x_{1,3} = \{x_3, x_4\} & x_{1,4} = \{x_3, x_4, x_5\} \\
x_{1,5} = \{x_3\} & x_{1,6} = \{x_3, x_4, x_5\} & x_{1,7} = \{x_4\} \\
x_{1,8} = \{x_3\} & x_{1,9} = \{x_4, x_5\} & x_{2,0} = \{x_6\} \\
x_{2,1} = \{\} & x_{2,2} = \{x_6, x_7\} & x_{2,3} = \{x_6, x_7\} \\
x_{2,4} = \{x_7\} & x_{2,5} = \{x_6, x_7\} & x_{2,6} = \{x_7\} \\
x_{2,7} = \{x_6\} & x_{2,8} = \{x_7\} & x_{2,9} = \{x_6\}
\end{array}$$

We compute the products of elements in these sets with a recursive call:

$$\begin{array}{lll}
x'_{0,0} = x_0x_2 & x'_{0,1} = x_0x_1x_2 & x'_{0,2} = x_0 \\
x'_{0,3} = x_1x_2 & x'_{0,4} = x_0x_1 & x'_{0,5} = x_0x_1x_2 \\
x'_{0,6} = x_0x_1x_2 & x'_{0,7} = x_0x_2 & x'_{0,8} = x_1x_2 \\
x'_{0,9} = x_0x_2 & x'_{1,0} = x_3x_4x_5 & x'_{1,1} = x_4 \\
x'_{1,2} = x_3x_4x_5 & x'_{1,3} = x_3x_4 & x'_{1,4} = x_3x_4x_5 \\
x'_{1,5} = x_3 & x'_{1,6} = x_3x_4x_5 & x'_{1,7} = x_4 \\
x'_{1,8} = x_3 & x'_{1,9} = x_4x_5 & x'_{2,0} = x_6 \\
x'_{2,1} = 1 & x'_{2,2} = x_6x_7 & x'_{2,3} = x_6x_7 \\
x'_{2,4} = x_7 & x'_{2,5} = x_6x_7 & x'_{2,6} = x_7 \\
x'_{2,7} = x_6 & x'_{2,8} = x_7 & x'_{2,9} = x_6
\end{array}$$

Finally, we express the outputs in terms of these newly computed values:

$$\begin{array}{ll}
y'_0 = \{x'_{0,0}, x'_{1,0}, x'_{2,0}\} & y'_1 = \{x'_{0,1}, x'_{1,1}, x'_{2,1}\} \\
y'_2 = \{x'_{0,2}, x'_{1,2}, x'_{2,2}\} & y'_3 = \{x'_{0,3}, x'_{1,3}, x'_{2,3}\} \\
y'_4 = \{x'_{0,4}, x'_{1,4}, x'_{2,4}\} & y'_5 = \{x'_{0,5}, x'_{1,5}, x'_{2,5}\} \\
y'_6 = \{x'_{0,6}, x'_{1,6}, x'_{2,6}\} & y'_7 = \{x'_{0,7}, x'_{1,7}, x'_{2,7}\} \\
y'_8 = \{x'_{0,8}, x'_{1,8}, x'_{2,8}\} & y'_9 = \{x'_{0,9}, x'_{1,9}, x'_{2,9}\}
\end{array}$$

These values will be computed in a subsequent iteration. $\Diamond$

Algorithm 8: `OutputClump`$(\mathbf{x}, \mathbf{y}, \alpha, \beta)$

Input: the set of inputs $\mathbf{x}$;
the set of desired outputs $\mathbf{y}$; $[\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})]$
the grouping factor β; and, $[\beta \geq 2]$
the clumping factor α. $[\alpha \geq \beta]$
Output: the set of new outputs $\mathbf{y}'$; and, $[\mathbf{y}' \subseteq \mathcal{P}(\mathbf{x})]$
the set of original outputs $\mathbf{y}''$, expressed in terms of both new and old outputs.

```
1: set: $\mathbf{y}' := \emptyset$
2: for $\left(i = 0 \text{ to } \left\lceil \frac{|\mathbf{y}|}{\alpha} \right\rceil - 1\right)$ do
3:     set: $U := \emptyset$
4:     set: $P_i := \{y_{i\cdot\alpha}, \ldots, y_{i\cdot\alpha+(\alpha-1)}\}$
5:     for each $(x_j \in \mathbf{x})$ do
6:         set: $S(x_j) := \emptyset$
7:         set: $S_j := \emptyset$
8:         for each $(y_k \in P_i)$ do
9:             if $(x_j \in y_k)$ then
10:                add: $S(x_j) \leftarrow S(x_j) \cup \{y_k\}$
11:            end if
12:        end for
13:        rewrite: $S(x_j) = \{y_{k_1}, \ldots, y_{k_m}\}$
14:        for $\left(l = 0 \text{ to } \left\lfloor \frac{m}{\beta} \right\rfloor\right)$ do
15:            set: $U_l := \{y_{k_{l\cdot\beta}}, \ldots, y_{k_{l\cdot\beta+(\beta-1)}}\}$
16:            add: $U \leftarrow U \cup \{U_l\}$
17:            add: $S_j \leftarrow S_j \cup \{U_l\}$
18:        end for
19:    end for
20:    rewrite: $U = \{U_0, \cdots, U_n\}$
21:    for $(k = 0 \text{ to } n)$ do
22:        set: $T(U_k) := \{x_j \mid U_k \in S_j\}$
23:        set: $\mathbf{y}' \leftarrow \mathbf{y}' \cup \{T(U_k)\}$
24:    end for
25:    for each $(y_k \in P_i)$ do
26:        set: $y'_k := \emptyset$
27:        set: $y''_k := y_k$
28:        for each $(U_l \in U)$ do
29:            if $(U_l \subseteq y''_k)$ then
30:                set: $y'_k \leftarrow y'_k \cup \{T(U_l)\}$
31:                set: $y''_k \leftarrow y''_k - U_l$
32:            end if
33:            set: $y'_k \leftarrow y'_k \cup y''_k$
34:        end for
35:    end for
36: end for
37: set: $\mathbf{y}'' := \{y'_0, \ldots, y'_{|\mathbf{y}|-1}\}$
Return: $\mathbf{y}', \mathbf{y}''$
```

Example 16. (Algorithm 8, Output Clumping) Let the inputs be $\vec{\mathbf{x}} = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$; the outputs be $\vec{\mathbf{y}} = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9\}$, where

$$y_0 = \{x_0, x_2, x_3, x_4, x_5, x_6\}, \quad y_1 = \{x_0, x_1, x_2, x_4\}, \quad y_2 = \{x_0, x_3, x_4, x_5, x_6, x_7\},$$
$$y_3 = \{x_1, x_2, x_3, x_4, x_6, x_7\}, \quad y_4 = \{x_0, x_1, x_3, x_4, x_5, x_7\}, \quad y_5 = \{x_0, x_1, x_2, x_3, x_6, x_7\},$$
$$y_6 = \{x_0, x_1, x_2, x_3, x_4, x_5, x_7\}, \quad y_7 = \{x_0, x_2, x_4, x_6\}, \quad y_8 = \{x_1, x_2, x_3, x_7\},$$
$$y_9 = \{x_0, x_2, x_4, x_5, x_6\};$$

and, let $\alpha_i = 4$ and $\beta_i = 2$. We first partition the outputs into $\lceil \frac{10}{4} \rceil = 3$ groups:

$$P_0 = \{y_0, y_1, y_2, y_3\}, \qquad P_1 = \{y_4, y_5, y_6, y_7\}, \qquad P_2 = \{y_8, y_9\}.$$

Focusing on P_0, we enumerate all $\binom{4}{2} = 6$ size-2 subsets:

$$\begin{aligned} U_0 &= \{y_0, y_1\}, & U_1 &= \{y_0, y_2\}, & U_2 &= \{y_0, y_3\}, \\ U_3 &= \{y_1, y_2\}, & U_4 &= \{y_1, y_3\}, & U_5 &= \{y_2, y_3\}. \end{aligned}$$

Next we compute $S(x_i)$ for each input x_i:

$$\begin{aligned} S(x_0) &= \{y_0, y_1, y_2\}, & S(x_1) &= \{y_1, y_3\}, & S(x_2) &= \{y_0, y_1, y_3\}, \\ S(x_3) &= \{y_0, y_2, y_3\}, & S(x_4) &= \{y_0, y_1, y_2, y_3\}, & S(x_5) &= \{y_0, y_2\}, \\ S(x_6) &= \{y_0, y_2, y_3\}, & S(x_7) &= \{y_2, y_3\}. \end{aligned}$$

Then we compute the S_i's from $S(x_i)$ and the U_j's as follows:

$$\begin{aligned} S_0 &= \{U_0, \{y_2\}\}, & S_1 &= \{U_4\}, & S_2 &= \{U_0, \{y_3\}\}, \\ S_3 &= \{U_1, \{y_3\}\}, & S_4 &= \{U_0, U_5\}, & S_5 &= \{U_1\}, \\ S_6 &= \{U_1, \{y_3\}\}, & S_7 &= \{U_5\}. \end{aligned}$$

Thus, we obtain the following $T(U_j)$'s, which will be computed in the subsequent iterations:

$$\begin{aligned} T(U_0) &= \{x_0, x_2, x_4\}, & T(U_1) &= \{x_3, x_5, x_6\}, & T(U_2) &= \emptyset, \\ T(U_3) &= \emptyset, & T(U_4) &= \{x_1\}, & T(U_5) &= \{x_4, x_7\}. \end{aligned}$$

We express the outputs in P_0 in terms of these newly computed values:

$$\begin{aligned} y_0' &= T(U_0) \cup T(U_1); \\ y_1' &= T(U_0) \cup T(U_4); \\ y_2' &= T(U_1) \cup T(U_5) \cup \{x_0\}; \text{ and,} \\ y_3' &= T(U_4) \cup T(U_5) \cup \{x_2, x_3, x_6\}. \end{aligned}$$

Finally, we compute these values using a recursive call. $\Diamond$

Example 17. (Algorithm 9, Input Clumping) Let the inputs be $\vec{\mathbf{x}} = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$; the outputs be $\vec{\mathbf{y}} = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9\}$, where

$$\begin{aligned} y_0 &= \{x_0, x_2, x_3, x_4, x_5, x_6\}, & y_1 &= \{x_0, x_1, x_2, x_4\}, & y_2 &= \{x_0, x_3, x_4, x_5, x_6, x_7\}, \\ y_3 &= \{x_1, x_2, x_3, x_4, x_6, x_7\}, & y_4 &= \{x_0, x_1, x_3, x_4, x_5, x_7\}, & y_5 &= \{x_0, x_1, x_2, x_3, x_6, x_7\}, \\ y_6 &= \{x_0, x_1, x_2, x_3, x_4, x_5, x_7\}, & y_7 &= \{x_0, x_2, x_4, x_6\}, & y_8 &= \{x_1, x_2, x_3, x_7\}, \\ y_9 &= \{x_0, x_2, x_4, x_5, x_6\}; \end{aligned}$$

Algorithm 9: `InputClump`($\mathbf{x}$, $\mathbf{y}$, α, β)

```
Input: the set of inputs $\mathbf{x}$;
    the set of desired outputs $\mathbf{y}$;                 [$\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})$]
    the grouping factor $\beta$; and,                        [$\beta \geq 2$]
    the clumping factor $\alpha$.                            [$\alpha \geq \beta$]
Output: the set of new inputs $\mathbf{x}'$; and,
    the set of new outputs $\mathbf{y}'$.                    [$\mathbf{y}' \subseteq \mathcal{P}(\mathbf{x}')$]
 1: set: $\mathbf{x}'' := \emptyset$
 2: for $\left(j = 0 \text{ to } |\mathbf{y}| - 1\right)$ do
 3:     set: $y'_j := \emptyset$
 4:     set: $y''_j := y_j$
 5: end for
 6: for $\left(i = 0 \text{ to } \left\lceil \frac{|\mathbf{x}|}{\alpha} \right\rceil - 1\right)$ do
 7:     set: $X_i := \{x_{i\cdot\alpha}, \ldots, x_{i\cdot\alpha+(\alpha-1)}\}$
 8:     for $\left(j = 0 \text{ to } |\mathbf{y}| - 1\right)$ do
 9:         set: $x_{i,j} := X_i \cap y''_j$
10:         while $\left(|x_{i,j}| \geq \beta\right)$ do
11:             rewrite: $x_{i,j} = \{x_{i_1}, \ldots, x_{i_n}\}$
12:             set: $x'_{i,j} := \{x_{i_1}, \ldots, x_{i_\beta}\}$     [$x'_{i,j} \subseteq x_{i,j}$]
13:             add: $y'_j \leftarrow y'_j \cup \{x'_{i,j}\}$
14:             remove: $y''_j \leftarrow y''_j - x'_{i,j}$
15:             add: $\mathbf{x}'' \leftarrow \mathbf{x}'' \cup \{x'_{i,j}\}$
16:             set: $x_{i,j} := X_i \cap y''_j$
17:         end while
18:     end for
19:     for $\left(j = 0 \text{ to } |\mathbf{y}| - 1\right)$ do
20:         add: $y'_j \leftarrow y'_j \cup y''_j$
21:         add: $\mathbf{x}'' \leftarrow \mathbf{x}'' \cup y''_j$
22:     end for
23: end for
24: set: $\mathbf{x}' :=$ MultiProd($\mathbf{x}, \mathbf{x}''$)            [Algorithm 5]
25: set: $\mathbf{y}' := \{y'_0, \ldots, y'_{|\mathbf{y}|-1}\}$
Return: $\mathbf{x}', \mathbf{y}'$
```

and, let $\alpha_i = 4$ and $\beta_i = 2$. We first partition the inputs into $\left\lceil \frac{8}{4} \right\rceil = 2$ groups:

$$P_0 = \{x_0, x_1, x_2, x_3\}, \qquad P_1 = \{x_4, x_5, x_6, x_7\},$$

Focusing on P_0, we enumerate all $\binom{4}{2} = 6$ size-2 subsets:

$$\begin{aligned} U_0 &= \{x_0, x_1\}, & U_1 &= \{x_0, x_2\}, & U_2 &= \{x_0, x_3\}, \\ U_3 &= \{x_1, x_2\}, & U_4 &= \{x_1, x_3\}, & U_5 &= \{x_2, x_3\}. \end{aligned}$$

And for P_1, we enumerate all $\binom{4}{2} = 6$ size-2 subsets:

$$\begin{aligned} U_6 &= \{x_4, x_5\}, & U_7 &= \{x_4, x_6\}, & U_8 &= \{x_4, x_7\}, \\ U_9 &= \{x_5, x_6\}, & U_{10} &= \{x_5, x_7\}, & U_{11} &= \{x_6, x_7\}. \end{aligned}$$

We compute the products of elements in these sets with a recursive call:

$$\begin{aligned}
x'_0 &= x_0x_1, & x'_1 &= x_0x_2, & x'_2 &= x_0x_3,\\
x'_3 &= x_1x_2, & x'_4 &= x_1x_3, & x'_5 &= x_2x_3,\\
x'_6 &= x_4x_5, & x'_7 &= x_4x_6, & x'_8 &= x_4x_7,\\
x'_9 &= x_5x_6, \\
x'_{10} &= x_5x_7, \\
x'_{11} &= x_6x_7.
\end{aligned}$$

We express the outputs in terms of these newly computed values:

$$\begin{aligned}
y'_0 &= \{x'_1, x'_6, x_3, x_6\}; & y'_1 &= \{x'_0, x_2, x_4\};\\
y'_2 &= \{x'_2, x'_6, x'_{11}\}; & y'_3 &= \{x'_3, x_3, x'_7, x_7\};\\
y'_4 &= \{x'_0, x_3, x_4, x'_{10}\}; & y'_5 &= \{x'_0, x'_5, x'_{11}\};\\
y'_6 &= \{x'_0, x'_5, x'_6, x_7\}; & y'_7 &= \{x'_1, x'_7\};\\
y'_8 &= \{x'_3, x_3, x_7\};\ \text{and,} & y'_9 &= \{x'_1, x'_6, x_6\}.
\end{aligned}$$

These values will be computed in a subsequent iteration. $\Diamond$

Algorithm 10: `NaiveMultiply`($\mathbf{x}_i$, $\mathbf{y}_i$)

```
Input: the set of inputs $\mathbf{x}$; and,
    the set of desired outputs $\mathbf{y}$.                  [$\mathbf{y} \subseteq \mathcal{P}(\mathbf{x})$]
Output: the set of outputs $\mathbf{X}$.      [$\mathbf{X} = \left\{\prod_{x\in y} x \mid y \in \mathbf{y}\right\}$]
 1: set: $\mathbf{X} := \emptyset$
 2: for each $(S_i \in \mathbf{y})$ do
 3:     if $(S_i \neq \emptyset)$ then
 4:         set: $y_i := 1$
 5:         for each $(x_j \in \mathbf{x})$ do
 6:             if $(x_j \in S_i)$ then
 7:                 compute: $y_i \leftarrow y_i \cdot x_j$
 8:             end if
 9:         end for
10:     else if $(S_i = \emptyset)$ then
11:         set: $y_i := 0$
12:     end if
13:     insert: $\mathbf{X} \leftarrow \mathbf{X} \cup \{y_i\}$
14: end for
Return: $\mathbf{X}$
```

3 Pippenger's Multiexponentiation Algorithm

Pippenger's Multiexponentiation Algorithm reduces an instance of the multiexponentiation problem to an instance of the multiproduct problem, then solves it using Pippenger's Multiproduct Algorithm.

As with his multiproduct algorithm, Pippenger originally presented the result as it applies to graphs. We omit the graph construction here and present pseudo-code that directly solves an instance of the multiexponentiation algorithm (using the multiproduct algorithm as a subroutine).

Algorithm 11: `MultiExp`$(\mathbf{x}, \mathbf{y})$

```
Input: the set of inputs $\mathbf{x}$; and,                    [$\mathbf{x} = \{x_0, \ldots, x_{q-1}\}$]
    the set of desired outputs $\mathbf{y}$.                    [$\mathbf{y} = \{y_0, \ldots, y_p\}$, where $y_i = \langle m_{i,0}, \ldots, m_{i,(q-1)} \rangle$]
Output: the set of outputs $\mathbf{X}$.                        [$\mathbf{X} = \{x_0^{m_{i,0}} \cdots x_{q-1}^{m_{i,(q-1)}} \mid y_i \in \mathbf{y}\}$]
 1: set: $k := 0$
 2: for each $(y_i \in \mathbf{y})$ do
 3:     set: $e :=$ the largest exponent in $y_i$
 4:     set: $k \leftarrow \max\{e, k\}$
 5: end for
 6: set: $a := \left\lceil \sqrt{\frac{|\mathbf{x}| \lg(k+1)}{|\mathbf{y}|}} \right\rceil$
 7: set: $b := \left\lceil \sqrt{\frac{|\mathbf{y}| \lg(k+1)}{|\mathbf{x}|}} \right\rceil$
 8: if $(|\mathbf{y}| \geq |\mathbf{x}|)$ then
 9:     call: $\mathbf{x}', \mathbf{y}', \mathbf{y}'' \leftarrow$Decompose($\mathbf{x}$, $\mathbf{y}$, $a$, $b$)       [Algorithm 12]
10: else if $(|\mathbf{y}| < |\mathbf{x}|)$ then
11:     call: $\mathbf{x}', \mathbf{y}', \mathbf{y}'' \leftarrow$Decompose($\mathbf{x}$, $\mathbf{y}$, $b$, $a$)       [Algorithm 12]
12: end if
13: call: $\mathbf{x}'' \leftarrow$MultiProd($\mathbf{x}'$, $\mathbf{y}'$)                          [Algorithm 5]
14: call: $\mathbf{X} \leftarrow$Combine($\mathbf{x}''$, $\mathbf{y}''$)                            [Algorithm 13]
Return: $\mathbf{X}$
```

Algorithm 12: `Decompose`($\mathbf{x}$, $\mathbf{y}$, r, b)

```
Input: the set of inputs $\mathbf{x}$;                                    [$\mathbf{x} = \{x_0, \ldots, x_{q-1}\}$]
    the set of desired outputs $\mathbf{y}$; and       [$\mathbf{y} = \{y_0, \ldots, y_p\}$, where $y_i = \langle m_{i,0}, \ldots, m_{i,(q-1)}\rangle$]
    a radix $r$ and word-length $b$              [$rb \geq \lceil \lg(m_{i,j}+1) \rceil$ for all $0 \leq i < p$, $0 \leq j < q$]
Output:
    $\mathbf{x}'$: the set of new inputs
    $\mathbf{y}'$: the set of new outputs
    $\mathbf{y}''$: the original outputs, expressed in terms of new outputs $\mathbf{y}'$
1: set: $\mathbf{x}' := \emptyset$
2: set: $\mathbf{y}'' := \emptyset$
3: for ($i$ from 0 to $p-1$) do
4:     for ($j$ from 0 to $q-1$) do
5:         rewrite: $m_{i,j}$ in binary, padded with zeros to a length of $rb$
6:         partition: binary representation of $m_{i,j}$ into $b$ sequential blocks, each of length $r$; label them $B_0, \ldots, B_{b-1}$,
           where bit 1 of $B_0$ is the least significant bit of $m_{i,j}$, and bit $r$ of $B_{b-1}$ is the most significant bit of $m_{i,j}$
7:         for ($l = 0$ to $b-1$) do
8:             set: $y_{i,l} := \emptyset$
9:         end for
10:        set: $x_{j,1} := x_j$
11:        for ($k$ from 1 to $r$) do
12:            if ($k \geq 1$) then
13:                set: $x_{j,k} := (x_{j,k-1})^{2^r}$                        [by squaring $r$ times]
14:            end if
15:            for ($l$ from 0 to $b-1$) do
16:                if (bit $k$ in $B_l$ is set) then
17:                    add: $y_{i,l} \leftarrow y_{i,l} \cup \{x_{j,k}\}$
18:                    add: $\mathbf{x}' \leftarrow \mathbf{x}' \cup \{x_{j,k}\}$
19:                end if
20:            end for
21:        end for
22:    end for
23: end for
24: set: $\mathbf{y}' := \{y_{0,0}, \ldots, y_{0,(b-1)}, \ldots, y_{(p-1),0}, \ldots, y_{(p-1),(b-1)}\}$
25: set: $\mathbf{y}'' := \{\langle y_{0,0}, \ldots, y_{0,(b-1)}\rangle, \ldots, \langle y_{(p-1),0}, \ldots, y_{(p-1),(b-1)}\rangle\}$
Return: $\mathbf{x}'$, $\mathbf{y}'$, $\mathbf{y}''$
```

Algorithm 13: `Combine`($\mathbf{x}$, $\mathbf{y}$)

```
Input: the set of inputs $\mathbf{x}$; and,        [$\mathbf{x} = \{x_{0,0}, \ldots, x_{0,(b-1)}, \ldots, x_{(p-1),0}, \ldots, x_{(p-1),(b-1)}\}$]
    the set of desired outputs $\mathbf{y}$.    [$\mathbf{y} = \{\langle y_{0,0}, \ldots, y_{0,(b-1)}\rangle, \ldots, \langle y_{(p-1),0}, \ldots, y_{(p-1),(b-1)}\rangle\}$]
Output: the set of outputs $\mathbf{X}$.                [$\mathbf{X} = \{(x_{i,(b-1)})^{2^{b-1}} \cdots (x_{i,0})^{2^0} \mid y_i \in \mathbf{y}\}$]
1: set: $\mathbf{X} := \emptyset$
2: set: $b := |y|$ for $y \in \mathbf{y}$
3: for (each $y_i \in \mathbf{y}$) do
4:     write: $y_i = \{y_{i,0}, \ldots, y_{i,(b-1)}\}$
5:     set: $y_i' := 1$
6:     for ($j = 0$ to $b-1$) do
7:         find: $x_{i,j} \in \mathbf{x}$ corresponding to $y_{i,j}$
8:         compute: $x_{i,j}' := (x_{i,j})^{2^j}$                          [by squaring $j$ times]
9:         compute: $y_i' \leftarrow y_i' \cdot x_{i,j}'$
10:    end for
11:    set: $\mathbf{X} \leftarrow \mathbf{X} \cup \{y_i'\}$
12: end for
Return: $\mathbf{X}$
```

Example 18. (Algorithm 12, Output decomposition) Suppose we are given the vector $\langle a, b, c\rangle$ and we wish to solve the multiexponentiation problem given by the following system of equations

$$\begin{aligned} y_1 &= a^{141}b^{216}c^{143} \\ y_2 &= a^{39}b^{225}c^{250} \\ y_3 &= a^{147}b^{237}c^{218} \end{aligned}$$

Observe that $p = q = 3$ and $k = 250$, so that $r = b = \left\lceil \sqrt{\lg(251)} \right\rceil = 3$. We decompose the term a^{141} in the first expression as follows:

$$\begin{tikzcd}[column sep=tiny]
141 = & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \Rightarrow & (\,a\,)^4 & \left(\,a^{64}\,\right)^2 & \left(\,a^{8} \right. & \left. a\,\right)^1
\arrow[from=1-2, to=1-4, phantom, "\underbrace{\qquad\qquad}_{2^{2\cdot 3}=64}"']
\arrow[from=1-5, to=1-7, phantom, "\overbrace{\qquad\qquad}^{2^{1\cdot 3}=8}"]
\arrow[from=1-8, to=1-10, phantom, "\underbrace{\qquad\qquad}_{2^{0\cdot 3}=1}"']
\arrow[from=1-3, to=1-13, bend left=40]
\arrow[from=1-8, to=1-12, bend left=40]
\arrow[from=1-10, to=1-15, bend left=40]
\arrow[from=1-7, to=1-14, bend right=40]
\end{tikzcd}$$

Each of the other terms is decomposed (and then grouped) similarly:

a^{141}: $141 = 010\ 001\ 101 \Rightarrow (a)^4\,(a^{64})^2\,(a^8a)$
b^{216}: $216 = 011\ 011\ 000 \Rightarrow (b^{64}b^8)^2\,(b^{64}b^8)$
c^{143}: $143 = 010\ 001\ 111 \Rightarrow (c)^4\,(c^{64}c)^2\,(c^8c) \quad \Rightarrow \underbrace{(ac)^4}_{(y_{0,2})^{2^{r-1}}}\,\underbrace{(a^{64}b^{64}b^8c^{64}c)^2}_{(y_{0,1})^{2^1}}\,\underbrace{(a^8ab^{64}b^8c^8c)}_{(y_{0,0})^{2^0}}$

a^{39}: $\ \ 39 = 000\ 100\ 111 \Rightarrow (a^8a)^4\,(a)^2\,(a)$
b^{225}: $225 = 011\ 100\ 001 \Rightarrow (b^8)^4\,(b^{64})^2\,(b^{64}b)$
c^{250}: $250 = 011\ 111\ 010 \Rightarrow (c^8)^4\,(c^{64}c^8c)^2\,(c^{64}c^8) \Rightarrow \underbrace{(a^8ab^8c^8)^4}_{(y_{1,2})^{2^{r-1}}}\,\underbrace{(ab^{64}c^{64}c^8c)^2}_{(y_{1,1})^{2^1}}\,\underbrace{(ab^{64}bc^{64}c^8)}_{(y_{1,0})^{2^0}}$

a^{147}: $147 = 010\ 010\ 011 \Rightarrow (a^{64}a^8a)^2\,(a)$
b^{237}: $237 = 011\ 101\ 101 \Rightarrow (b^8b)^4\,(b^{64})^2\,(b^{64}b^8b)$
c^{218}: $218 = 011\ 011\ 010 \Rightarrow (c^{64}c^8c)^2\,(c^{64}c^8) \quad \Rightarrow \underbrace{(b^8b)^4}_{(y_{2,2})^{2^{r-1}}}\,\underbrace{(a^{64}a^8ab^{64}c^{64}c^8c)^2}_{(y_{2,1})^{2^1}}\,\underbrace{(ab^{64}b^8bc^{64}c^8)}_{(y_{2,0})^{2^0}}$

We use repeated squaring to compute the inputs to Pippenger's Multiproduct Algorithm:

$$\mathbf{x} = \{a, a^8, a^{64}, b, b^8, b^{64}, c, c^8, c^{64}\},$$

while the desired outputs are

$$\mathbf{y}' = \{y_{0,0}, y_{0,1}, y_{0,2}, y_{1,0}, y_{1,1}, y_{1,2}, y_{2,0}, y_{2,1}, y_{2,2}\}.$$

Once this multiproduct problem has been solved, `Combine` (Algorithm 13) is called with inputs

$$\mathbf{y}'' = \{\langle y_{0,0}, y_{0,1}, y_{0,2}\rangle, \langle y_{1,0}, y_{1,1}, y_{1,2}\rangle, \langle y_{2,0}, y_{2,1}, y_{2,2}\rangle\}$$

to compute the original output set $\mathbf{y} = \{y_0, y_1, y_2\}$. ◊

Example 19. (Algorithm 13, Input Combining) Continuing with Example 18, suppose that we have used Pippenger's Multiproduct Algorithm to compute the inputs

$$\mathbf{x}'' = \{x_{0,0}, x_{0,1}, x_{0,2}, x_{1,0}, x_{1,1}, x_{1,2}, x_{2,0}, x_{2,1}, x_{2,2}\}$$

corresponding to

$$\mathbf{y}' = \{y_{0,0}, y_{0,1}, y_{0,2}, y_{1,0}, y_{1,1}, y_{1,2}, y_{2,0}, y_{2,1}, y_{2,2}\},$$

and are given the desired outputs

$$\mathbf{y}'' = \{\langle y_{0,0}, y_{0,1}, y_{0,2}\rangle, \langle y_{1,0}, y_{1,1}, y_{1,2}\rangle, \langle y_{2,0}, y_{2,1}, y_{2,2}\rangle\}.$$

We compute the needed powers of $x_{i,j}$'s by repeated squaring as follows

$$\begin{aligned}
x'_{0,0} &= (x_{0,0})^{2^0} = \left(a^8ab^{64}b^8c^8c\right)^1 & x'_{0,1} &= (x_{0,1})^{2^1} = \left(a^{64}b^{64}b^8c^{64}c\right)^2 & x'_{0,2} &= (x_{0,2})^{2^2} = (ac)^4 \\
&= a^8ab^{64}b^8c^8c & &= a^{128}b^{128}b^{16}c^{128}c^2 & &= a^4c^4 \\
x'_{1,0} &= (x_{1,0})^{2^0} = \left(ab^{64}bc^{64}c^8\right)^1 & x'_{1,1} &= (x_{1,1})^{2^1} = \left(ab^{64}c^{64}c^8c\right)^2 & x'_{1,2} &= (x_{1,2})^{2^2} = \left(a^8ab^8c^8\right)^4 \\
&= ab^{64}bc^{64}c^8 & &= a^2b^{128}c^{128}c^{16}c^2 & &= a^{32}a^4b^{32}c^{32} \\
x'_{2,0} &= (x_{2,0})^{2^0} = \left(ab^{64}b^8bc^{64}c^8\right)^1 & x'_{2,1} &= (x_{2,1})^{2^1} = \left(a^{64}a^8ab^{64}c^{64}c^8c\right)^2 & x'_{2,2} &= (x_{2,2})^{2^2} = \left(b^8b\right)^4 \\
&= ab^{64}b^8bc^{64}c^8 & &= a^{128}a^{16}a^2b^{128}c^{128}c^{16}c^2 & &= b^{32}b^4.
\end{aligned}$$

Then we compute the final outputs in terms of these values:

$$\begin{aligned}
y_1 &= (x_{0,0})^{2^0}(x_{0,1})^{2^1}(x_{0,2})^{2^2} = a^{(8+1+128+4)}b^{(64+8+128+16)}c^{(8+1+128+2+4)} = a^{141}b^{216}c^{143} \\
y_2 &= (x_{1,0})^{2^0}(x_{1,1})^{2^1}(x_{1,2})^{2^2} = a^{(1+2+32+4)}b^{(64+1+128+32)}c^{(64+8+128+16+2+32)} = a^{39}b^{225}c^{250} \\
y_3 &= (x_{3,0})^{2^0}(x_{3,1})^{2^1}(x_{3,2})^{2^2} = a^{(1+128+16+2)}b^{(64+8+1+128+32+4)}c^{(64+8+128+16+2)} = a^{147}b^{237}c^{218}.
\end{aligned}$$

$\lozenge$

4 Parameter derivations

In [12, 13], Pippenger presented details that are sufficient to derive a reasonable parameter sequence under certain restrictions on p, q and k (recall that p is the number of equations, q the number of unknowns, and k the largest exponent, respectively). Unfortunately, Pippenger was interested only in the asymptotic complexity, and his method makes use of the assumption that $\sqrt{\frac{\lg p}{\lg\lg p}} \geq 10$ (or $\sqrt{\frac{\lg q\lceil\lg(k+1)\rceil}{\lg\lg q\lceil\lg k+1\rceil}} \geq 10$), which implies that $p \geq 6.7 * 10^{299}$ (or $q\,\lceil\lg(k+1)\rceil \geq 6.7 * 10^{299}$). Clearly, this assumption is unrealistic for practical applications.[5] In [2], Bernstein remarks that "One can quickly compute Pippenger's parameter sequence given p, q, although

[5] Having $p \geq 6.7 * 10^{299}$ would imply an available address space of more than 996-bits, just to reference each equation!

Pippenger did not say this explicitly"; however, no additional details are given and the citations he provides contain only the aforementioned result with the unrealistic restrictions on the size of the problem to be solved. Personal communications with Pippnger confirm that he is unaware of any method for deriving parameters for problems of a realistic size [14]. To the best of the author's knowledge, the only known approach for finding suitable parameter sequences for problem instances of a realistic size is by experimentation; i.e., by a brute-force search. In particular, for a fixed p, q, and k, try random problem instance-parameter sequence pairs until you are reasonably confident that you have found an optimal parameter sequence. While this may seem a daunting task, it is made easier by two important facts: 1) in practice, both p and q will typically be fairly small; and, 2) we are actually only interested in $\lceil \lg(k+1) \rceil$ rather than k itself. This latter point ensures that applications involving, for example, 4096-bit exponents (which may be needed for certain cryptographic applications) are still tractable to solve by brute force. Once suitable parameters have been found for a particular application, they can be hard coded into the algorithms implementation and simply looked up from a table when needed by the algorithm.

Acknowledgements. The author would like to thank Nicholas Pippenger for his prompt and helpful email communications during the preparation of, as well as Ian Goldberg and Arne Storjohann for their valuable feedback on early versions of, this manuscript. This work was supported in part by an NSERC Alexander Graham Bell Graduate Scholarship (NSERC CGS-M) and a David R. Cheriton Graduate Scholarship. Finally, Daniel J. Bernstein's paper [2] on Pippenger's exponentiation algorithm was extremely helpful in the preparation of this paper.

References

1. Richard Bellman. Advanced Problem 5125. *The American Mathematical Monthly*, 70(7):765, September 1963. Problem 5125 was proposed here and solved in [17] by Ernst Gabor Straus.
2. Daniel J. Bernstein. Pippenger's Exponentiation Algorithm. Preprint. (To be incorporated into the author's forthcoming book 'High-speed Cryptography'), January 2002. Published electronically at: http://cr.yp.to/papers.html#pippenger.
3. Alfred Brauer. On Addition Chains. *Bulletin of the American Mathematical Society (AMS)*, 45(10):736–739, October 1939.
4. Peter J. Downey, Benton L. Leong, and Ravi Sethi. Computing Sequences with Addition Chains. *SIAM Journal on Computing (SICOMP)*, 10(3):638–646, August 1981.
5. Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble using VERBS. In Mikhail Atallah and Nick Hopper, editors, *Proceedings of the 10th International Symposium on Privacy Enhancing Technologies (PETS 2010)*, volume 6205 of *Lecture Notes in Computer Science*, pages 110–129. Springer-Verlag, Berlin Heidelberg, July 2010. An extended version of this paper is available [6].
6. Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble using VERBS (Extended Version). Computer Science Technical Report CACR 2010-05, University of Waterloo, Centre for Applied Cryptographic Research, Waterloo, Ontario, Canada, March 2010. 24 pages. Published electronically at: http://www.cacr.math.uwaterloo.ca/tech_reports.html. This is the extended version of [5].
7. Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley Professional, Reading, Massachusetts, USA, November 1997. ISBN 0-201-89684-2.
8. Oleg B. Lupanov. О вентильных и контактно-вентильных схемах. *Doklady Akademii Nauk SSSR*, 111(6):1171–1174, 1956. Journal title translates from Russian to English as: *Proceedings of the USSR Academy of Sciences*; article title translates as: 'On Rectifier and Contact Rectifier Circuits'.
9. Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996. ISBN 0-8493-8523-7. Fifth Printing (August 2001).
10. É. I. Nechiporuk. О Вентильных Схемах. *Doklady Akademii Nauk SSSR*, 148(1):50–53, 1963. Journal title translates from Russian to English as: *Proceedings of the USSR Academy of Sciences*; this article was translated in [11].

11. É. I. Nechiporuk. Rectifier Networks. *Soviet Physics, Doklady*, 8(1):5–7, 1963. This is an English translation of the Russian article [10].
12. Nicholas Pippenger. On the Evaluation of Powers and Related Problems (Preliminary Version). In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, pages 258–263. Institute of Electrical and Electronics Engineers (IEEE) Computer Society, Washington, DC, USA, October 1976.
13. Nicholas Pippenger. The Minimum Number of Edges in Graphs with Prescribed Paths. *Theory of Computing Systems (TOCS)*, 12(1):325–346, December 1978. From Volume 1 (1967) to Volume 29 (1996), this journal was published as *Mathematical Systems Theory*.
14. Nicholas Pippenger. Personal communication via email, March 2010.
15. Arnold Scholz. Aufgabe 253. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 47(II):41–42, January 1938. Journal title translates from German to English as: *Annual Report of the German Mathematical Society*; article title translates as: ‘Problem 253’.
16. Neil J. A. Sloane. Sequence A003313: Length of the shortest addition chain for n. In Neil J. A. Sloane, editor, *The On-Line Encyclopedia of Integer Sequences (OEIS)*. AT&T Labs Research, Florham Park, New Jersey, USA, 2010. Published electronically at: `http://www.research.att.com/~njas/sequences/A003313`.
17. Ernst Gabor Straus. Addition Chains of Vectors (Problem 5125). *The American Mathematical Monthly*, 71(7):806–808, September 1964. Problem 5125 was proposed by Richard Bellman, The RAND Corporation, Santa Monica, California in [1] and solved here.
18. Andrew Chi-Chih Yao. On the Evaluation of Powers. *SIAM Journal on Computing (SICOMP)*, 5(1):100–103, March 1976.