Programming Research Group

AN EFFICIENT SPARSE ADAPTATION OF THE POLYTOPE METHOD OVER $\mathbb{F}_p$ AND A RECORD-HIGH BINARY BIVARIATE FACTORISATION

Fatima Abu Salem*

PRG-RR-04-13

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, U.K. E-mail: fkas@comlab.ox.ac.uk. The author is supported by the EPSRC.

Abstract

A recent bivariate factorisation algorithm appeared in [1] based on the use of Newton polytopes and a generalisation of Hensel lifting. Although possessing of a worst case exponential running time like the Hensel lifting algorithm, the polytope method should perform well for sparse polynomials whose Newton polytopes have very few Minkowski decompositions. A preliminary implementation in [1] indeed reflects this property, but does not exploit the fact that the algorithm preserves the sparsity of the input polynomial, so that the total amount of work and space required are $O(d^4)$ and $O(d^2)$ respectively, for an input bivariate polynomial of total degree d. In this paper, we show that the polytope method can be made sensitive to the number of nonzero terms of the input polynomial, so that the input size becomes depedent on both the degree and the number of terms of the input bivariate polynomial. We describe a sparse adaptation of the polytope method over finite fields with prime order, which requires fewer bit operations and memory references for sparse polynomials which are known to be the product of two sparse factors. Under this condition, our refinement reduces the amount of work per extension of a coprime dominating edge factorisation and the total spatial complexity to $O(td^2 + t^3d)$ bit operations and $O(t^{O(1)}d)$ bits of memory respectively. To the best of our knowledge, the sparse binary factorisations achieved using this adaptation are of the highest degree so far, reaching a world record degree of 20000 for a very sparse bivariate polynomial over $\mathbb{F}_2$.

1 Introduction

Polynomial factorisation has long been one of the attractive areas in symbolic computation that is valued for both the challenges and open mathematical problems it addresses as well as its usefulness in applications in various fields. For multivariate polynomial factorisation over finite fields, many approaches have been developed over the past few decades to provide efficient algorithms in "practice" [6, 11, 12, 17, 18, 20, 21, 22, 23, 24, 26, 27], among which are various Hensel lifting based algorithms. Recently, the factorisation of a dense bivariate polynomial of total degree 2000 over $\mathbb{F}_{17}$ with over a million monomials was achieved by Bernardin in [5]. New approaches have emerged based on the connections between multivariate polynomials and their Newton polytopes. The work in [10, 13] examined absolute irreducibility testing of multivariate polynomials via polytopes, followed by the work in [1] which examined polynomial factorisation through a generalisation of Hensel lifting as applied to the Newton polytope of the input polynomial.

Despite its worst case exponential running time, the polytope method has been associated with a number of advantages promising to make it very efficient in practice. First, when applied to the special category of sparse polynomials whose Newton polytopes have very few Minkowski decompositions, one would expect to have a small number of edges to lift from. Although we do not yet have a heuristic estimate of the frequency with which this can happen, experiments reported in [1] clearly reflected this observation, whereby most random input polynomials had Newton polytopes with the above property, and the bulk of the work was spent in the lifting stage. However, the implementation used there was dense, where the total amount of work is of the order $O(d^4)$ for an input bivariate polynomial of total degree d, and requiring an order of $O(d^2)$ bits of memory, which prompts us to investigate further advantages resulting from the sparsity of the input polynomial. Since the polytope method has been shown to preserve the sparsity of the polynomial by avoiding the randomisation and substitution of linear forms in the classical Hensel lifting method, one natural question to answer is how to describe the sensitivity of the polytope method with respect to the number of nonzero terms of the input polynomial. We are equally motivated to investigate how exploiting this aspect can possibly increase the problem sizes which the polytope method can handle for the special class of sparse polynomials. The approach we present produces a sparse factorisation algorithm per se, where the operational and spatial complexities become dependent on both the degree of the input polynomial as well as the number of nonzero terms of its possible factors which the polytope method can detect. The aspects we exploit are that the input polynomial and its factors have many zero coefficients, and that most of the lifted polynomials are zero, or at worst very sparse. Further advantages we exploit are that, unlike the case in Hensel lifting when one sometimes has to extend the ground field in question to have enough evaluation points, the polytope method works very well even for finite fields of small cardinality, thus avoiding the extra cost that this incurs. As in the original algorithm, this method works only under certain coprimality conditions governing the edge factorisations along a special subset of edges of the Newton polytope (see [1] for more details).

The rest of this paper is organised as follows: In Section 2 we review the theory behind the polytope method as presented in [1]. In Section 3 we describe the model of sparse polynomials to which this algorithm is best suited. In Section 4 we describe the implementation in C and the sub-routines comprising the pre-lifting stages. In Section 5 we present our sparse adaptation which affects the polytope method at the lifting stage. In Sections 7 and 8 we analyse the complexity of the sparse method, and in Section 9 we report on the run times of our experiments producing high record degree factorisations over $\mathbb{F}_2$.

2 The polytope method

For a complete analysis of results and proofs in this section we refer the reader to [1]. In what follows we will consider the factorisation over a finite field $\mathbb{F}$ with q elements. Let $\mathbb{F}[x, y]$ be the ring of polynomials in two variables over $\mathbb{F}$. For any vector $e = (e_1, e_2)$ of non-negative integers define $X^e := x^{e_1}y^{e_2}$. Let $f \in \mathbb{F}[x, y]$ be given by $f := \sum_e a_e X^e$ where the sum is over finitely many points e in $\mathbb{N}^2$, and $a_e \in \mathbb{F}$. The Newton polytope of f is the *integral* polytope in $\mathbb{R}^2$ obtained as the convex hull of all exponents e for which the corresponding coefficient a_e is nonzero. Given two polytopes Q and R their *Minkowski sum* is defined to be the set

$$Q + R := \{q + r \mid q \in Q, r \in R\}.$$

Let $\mathrm{Newt}(f) = Q + R$ be a decomposition of the Newton polytope of f into integral polygons in the first quadrant. Define the polynomials g and h as

$$g := \sum_{q \in Q} g_q X^q, \quad \text{and } h := \sum_{r \in R} h_r X^r,$$

where g_q and h_r belong to $\mathbb{F}$. Let $\#\mathrm{Newt}(f)$ denote the number of integral points belonging to $\mathrm{Newt}(f)$, and $\mathrm{Edge}(f)$ denote the set of all edges of $\mathrm{Newt}(f)$. Each edge $\delta \in \mathrm{Newt}(f)$ is viewed as directed so that $\mathrm{Newt}(f)$ lies on the left hand side of the edge, and this directed edge can be defined by an affine function ℓ as follows. Suppose the edge δ is from (u_1, v_1) to (u_2, v_2), vertices with integral coordinates. (u_1, v_1) is called the starting vertex of the edge. Let

$$d = \gcd(u_2 - u_1, v_2 - v_1), \quad u_0 = (u_2 - u_1)/d, \quad v_0 = (v_2 - v_1)/d.$$

Then (u_0, v_0) represents the direction of δ and the integral points on δ are of the form

$$(u_1, v_1) + i(u_0, v_0), \quad i \in \mathbb{N}.$$

Let $(\nu_1, \nu_2) := (-v_0, u_0)$, a rotation of (u_0, v_0) by 90 degrees counter clockwise, and let $\eta = v_0 u_1 - u_0 v_1$. Define

$$\ell(e) = \nu_1 e_1 + \nu_2 e_2 + \eta, \quad \text{for } e = (e_1, e_2) \in \mathbb{R}^2.$$

Then ℓ has the property that $\ell(e) \geq 0$ for each point $e \in \mathrm{Newt}(f)$, with the equation holding iff $e \in \delta$, that is, $\mathrm{Newt}(f)$ lies in the positive side of the line $\ell = 0$. Note that $\gcd(\nu_1, \nu_2) = 1$. We call this function ℓ *the primitive affine function* associated with δ, denoted by ℓ_δ.

The function ℓ_δ has another nice property that will be extremely useful for our algorithm. Since $\gcd(\nu_1, \nu_2) = 1$, there exist integers ζ_1 and ζ_2 such that $\zeta_1 \nu_1 + \zeta_2 \nu_2 = 1$, and they are unique under the requirement that $0 \leq \zeta_2 < \nu_1$. Define the change of variables

$$z := x^{\nu_2} y^{-\nu_1} \text{ and } w := x^{\zeta_1} y^{\zeta_2}. \tag{1}$$

Then any monomial of the form $x^{e_1} y^{e_2}$ can be written as

$$x^{e_1} y^{e_2} = z^{i_1} w^{i_2}$$

where

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = \begin{pmatrix} \zeta_2 & -\zeta_1 \\ \nu_1 & \nu_2 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}.$$

Its inverse transform is

$$\begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = \begin{pmatrix} \nu_2 & \zeta_1 \\ -\nu_1 & \zeta_2 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}.$$

This change of variables has the nice property that when (e_1, e_2) moves along the direction (u_0, v_0) of the edge δ, then the exponent of w remain constant (as $i_2 = \ell_\delta(e_1, e_2) - \eta$), while the exponent of z strictly increases (by $1 = \zeta_2 u_0 - \zeta_1 v_0$ for each increment of (u_0, v_0)).

For each $\delta \in \mathrm{Edge}(f)$, there exists a unique pair of faces (either edges or vertices) δ' and δ'' of Q and R, respectively, such that $\delta = \delta' + \delta''$. One can also easily show that there exists a unique integer c_δ such that

$$\begin{aligned} \delta' &= \{e \in Q \mid \ell_\delta(e) = c_\delta\} \\ \delta'' &= \{e \in R \mid \ell_\delta(e) = -c_\delta + \eta\} \end{aligned}$$

where η is the constant coefficient of ℓ_δ.

For $i \geq 0$ we define:

$$f_i^\delta := \sum_{l_\delta(e)=i} a_e X^e, g_i^\delta := \sum_{q \in Q,\, l_\delta(q)=c_\delta+i} g_q X^q, \text{ and } h_i^\delta := \sum_{r \in R,\, l_\delta(r)=-c_\delta+\eta+i} h_r X^r,$$

where the polynomials for $i = 0$ are called edge polynomials, and the polynomials for $i > 0$ are those obtained from f, g, or h, by considering only those terms which lie on particular lines. We now have that the system of equations in the coefficient indeterminates of g and h defined by equating monomials on both sides of the equality $f = gh$ has the same solutions as the system of equations defined by the following:

$$f_0^\delta = g_0^\delta h_0^\delta, \text{ and } g_0^\delta h_k^\delta + h_0^\delta g_k^\delta = f_k^\delta - \sum_{j=1}^{k-1} g_j^\delta h_{k-j}^\delta \text{ for } k \geq 1. \tag{2}$$

Thus any specialisation of coefficient indeterminates which is a solution of equations (2) will give a full factorisation of f. However, one still has to ensure that the method can proceed in a unique way at each iterative step. To do this, we introduce a special type of boundary factorisations. Let $\Gamma \subseteq \mathrm{Edge}(f)$, and let $K = (k_\gamma)_{\gamma \in \Gamma}$ be a vector of positive integers labelled by Γ. Define

$$\mathrm{Newt}(f)|_{\Gamma,K} := \{e \in \mathrm{Newt}(f) \mid 0 \leq l_\gamma(e) < k_\gamma \text{ for some } \gamma \in \Gamma\}.$$

For each $\delta \in \mathrm{Edge}(f)$, there exists a unique pair of faces (either edges or vertices) δ' and δ'' of Q and R respectively such that $\delta = \delta' + \delta''$. One can easily show that there exists a unique integer c_δ such that

$$\begin{aligned} \delta' &= \{q \in Q \mid l_\delta(q) = c_\delta\} \\ \delta'' &= \{r \in R \mid l_\delta(r) = -c_\delta + \eta\} \end{aligned}$$

where $\eta = l_\delta(0)$. We denote by $Q|_{\Gamma,K}$ and $R|_{\Gamma,K}$ the subsets of Q and R respectively given by

$$\begin{aligned} Q|_{\Gamma,K} &:= \{e \in Q \mid 0 \leq l_\delta(e) < k_\delta + c_\delta \text{ for some } \delta \in \Gamma\} \\ R|_{\Gamma,K} &:= \{e \in R \mid 0 \leq l_\delta(e) < k_\delta - c_\delta + \eta \text{ for some } \delta \in \Gamma\}. \end{aligned}$$

If S is a subset of $\mathrm{Newt}(f)$, we define the *S-partial factorisation* of f to be a specialisation of a subset of the field elements g_q and h_r such that for each lattice point $s \in S$ the coefficients of monomials X^s in the polynomials gh and f are equal field elements. A $\mathrm{Newt}(f)|_{\Gamma,K}$-factorisation is called a $(\Gamma, K; Q, R)$-factorisation if exactly the indeterminate coefficients of g and h indexed by lattice points in $Q|_{\Gamma,K}$ and $R|_{\Gamma,K}$, respectively, have been specialised. Furthermore, if $K' = (k'_\gamma)_{\gamma\in\Gamma}$ is a vector of positive integers with $k'_\gamma \geq k_\gamma$ for all $\gamma \in \Gamma$, with the inequality strict for at least one γ, then not all of the indeterminate coefficients of g indexed by lattice points in $Q|_{\Gamma,K'}$ have been specialised. If K is the all ones vector, denoted $(\underline{1})$, of the appropriate length indexed by elements of some set Γ, and when Γ, Q and R are fixed and evident from the context, then we call this a *partial boundary factorisation* or *$(\underline{1})$-factorisation.*

We say that Γ is a set of dominating edges if there exists a vector r in $\mathbb{R}^2$ such that P is contained in the Minkowski sum $\mathrm{Mink}(\Gamma, r)$ of the set Γ and the infinite line segment $r\mathbb{R}_{\geq 0}$ (the positive hull of r), and such that each of the two infinite edges of $\mathrm{Mink}(\Gamma, r)$ contains exactly one point of P. Γ is further called *irredundant* if it does not strictly contain any other dominating set. We call a $(\Gamma; Q, R)$-boundary factorisation of f a *dominating edges factorisation* relative to Γ, Q and R. A *coprime* dominating edges factorisation is a $(\Gamma; Q, R)$-boundary factorisation with the property that for each $\delta \in \Gamma$ the edge polynomials g_0^δ and h_0^δ are coprime, up to monomial factors. The main theorem behind the polytope algorithm now follows [1]:

Theorem 1 *Let $f \in \mathbb{F}[x, y]$ and $\mathrm{Newt}(f) = Q + R$ be a fixed Minkowski decomposition, where Q and R are integral polygons in the first quadrant. Let Γ be an irredundant dominating set of $\mathrm{Newt}(f)$ in direction r, and assume that Q is not a single point or a line segment parallel to $r\mathbb{R}_{\geq 0}$. For any coprime dominating edges factorisation of f relative to Γ, Q and R, there exists at most one full factorisation of f which extends it, and moreover this full factorisation may be found or shown not to exist in time polynomial in $\#\mathrm{Newt}(f)$.*

A detailed version of the polytope algorithm as presented in [1] is given below. For correctness and analysis of the running time, we refer the reader to [1].

Algorithm 2
Input: A polynomial $f \in \mathbb{F}[x, y]$ of total degree d, and a positive integer M.
Output: A factorisation of f or “failure” or “f is irreducible”.

Step 1: Compute a vertex-edge description of $\mathrm{Newt}(f)$ using a polygon convex hull algorithm. Let the edges be $\delta_1, \ldots, \delta_m$, cyclically joining vertices $v_0, \ldots, v_{m-1}$.

Step 2: For each edge $\delta \in \mathrm{Edge}(f)$, compute a complete factorisation of all edge polynomials f_0^δ, and determine the set $\{m_j^{(i)} \mid 0 \leq j \leq \deg(f_0^\delta)\}$, where $m_j^{(\delta)}$ is the number of degree j monic factors of f_0^δ.

Step 3: List all sets $\{\Gamma_i\}$ of squarefree dominating facets of $\mathrm{Newt}(f)$. If there are no such sets then fail. For each set Γ_i, count the number of coprime $(\Gamma_i; Q, R)$-boundary factorisations, where Q and R range over all integral decompositions of $\mathrm{Newt}(f)$. Select the dominating set Γ for which this number is minimal. If this is greater than M then fail.

Step 4: For each coprime dominating facets factorisation of f using edges in Γ, do the following:

4.1: Set $K = (k_\delta)$ where $k_\delta = 1$ for all $\delta \in \Gamma$.

4.2: For each $\delta \in \Gamma$, let n_δ denote the number of integral points on δ. Let m_δ be the number of integral points on the face of the polyhedron $\{r \in Q \mid l_\delta(r) \geq c_\delta + k_\delta \text{ for all } \delta \in \Gamma\}$ supported by $l_\delta - c_\delta - k_\delta$.

4.3: Select $\delta \in \Gamma$ such that $m_\delta < n_\delta$. (If δ has already been chosen in a previous lifting step then the computations in Steps 4.3 and 4.4 need not be performed again.) Let the normalised affine function supporting δ be l_δ, where $l_\delta(e_1, e_2) = \nu_1 e_1 + \nu_2 e_2 + \eta$ and $(\nu_1, \nu_2) = 1$. Compute coprime integers ζ_1 and ζ_2 such that $\zeta_1\nu_1 + \zeta_2\nu_2 = 1$, and with $0 \leq \zeta_2 < \nu_1$. Define the change of variables $x^{e_1}y^{e_2} = z^{i_1}w^{i_2}$ where

$$i_1 = e_1\zeta_2 - e_2\zeta_1,$$
$$i_2 = e_1\nu_1 + e_2\nu_2 = l_\delta(e_1, e_2) - \eta.$$

Let the monomials s and t be the terms of g and h respectively whose exponents vectors are the starting vertices of the faces of Q and R defined by $l_\delta - c_\delta$ and $l_\delta + c_\delta - \eta$, respectively. We have $g_i^\delta(z, w) = sw^iG_i(z)$ for some univariate Laurent polynomial $G_i(z)$. Similarly $h_i^\delta(z, w) = tw^iH_i(z)$ and $f_i^\delta(z, w) = stw^iF_i(z)$, where $H_i(z)$ and $F_i(z)$ are univariate Laurent polynomials.

4.4: Compute $U(z)$ and $V(z)$ such that $V(z)H_0(z) + U(z)G_0(z) = 1$.

4.5: Solve for the unknown coefficients in G_{k_δ} uniquely, or show that no solution exists, using the two triangular systems for the coefficients of ε defined by the equation

$$G_{k_\delta} - \{V(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_jH_{k_\delta - j}) \bmod G_0\} = \varepsilon G_0 \tag{3}$$

If no solution exists halt.

4.6: Compute the polynomial

$$\frac{(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_jH_{k_\delta - j}) - G_{k_\delta}H_0}{G_0}. \tag{4}$$

If its coefficients match with the known coefficients of H_{k_δ} then set H_{k_δ} equal to this polynomial. Otherwise halt.

4.7: Increment k_δ by 1. If all coefficients in Q and R have not been specialised then return to Step 4.2.

4.8: All the coefficients in g and h have now been specialised. Check whether $f = gh$, and if so output the pair (g, h).

3 Input model

We choose to investigate the performance of the sparse adaptation when the input polynomial belongs to $\mathbb{F}_p[x, y]$, for a finite field $\mathbb{F}_p$ with prime order. As previously reported in the dense implementation of [1], the random experiments are generated by constructing a degree d input polynomial f using two random polynomials g and h of degree $d/2$ each, with a given number of nonzero terms. Let t_g and t_h denote the number of nonzero terms in g and h respectively, and let $t = t_g t_h$. The number of nonzero terms in f is thus $O(t)$. For reasons that will become apparent later on, we will assume the condition

$$td + t^3 < d^3.$$

This will make up our definition of a sparse polynomial f, where d^2 is an upper bound on the number of nonzero terms that can appear in a degree d polynomial in $\mathbb{F}_p[x, y]$. Note that one of t_g or t_h has to be at most $t^{1/2}$, and hence we can assume that g and h have $t^{O(1)}$ terms each. In the remainder of this chapter, we shall omit the reference to "nonzero terms" and refer to these as simply "terms". Also, when analysing the complexity of the sparse method with respect to an integral decomposition

$$\mathrm{Newt}(f) = Q + R,$$

we will restrict our attention to the case when Q and R correspond to the sparse factors g and h as defined above; i.e., when $Q = \mathrm{Newt}(g)$ and $R = \mathrm{Newt}(h)$. By this, we understand that an extension of a coprime dominating edges factorisation using our sparse method should be aborted once the number of specialised coefficients corresponding to Q or R exceeds $t^{O(1)}$.

3.0.1 Generic shape of $\mathrm{Newt}(f)$

We now describe few aspects characterising the generic shape of $\mathrm{Newt}(f)$ for a non-trivial input f. By non-trivial we refer to the case when f is nonzero, non-constant, and not known to be divisible by any monomial of the form $f_{(e_1,e_2)} x^{e_1} y^{e_2}$, for some positive integers e_1, e_2, and $f_{(e_1,e_2)} \in \mathbb{F}_p$.

Lemma 3 *Let $f \in \mathbb{F}_p[x, y]$ be of total degree d. Then f has at least one term with degree zero in y and one term with degree zero in x if and only if the corresponding exponent vectors of those two or more terms are vertices or form edges of* $\mathrm{Newt}(f)$ *that lie on the x-axis and y-axis respectively.*

Proof: Recall that for a bivariate regular polynomial f, $\mathrm{Newt}(f)$ lies entirely in the quadrant of positive coordinates. Suppose that $\mathrm{Newt}(f)$ has at least two vertices lying on the x-axis and y-axis respectively. Then f should have at least two terms which have a zero exponent in y and x respectively. Now suppose that f has at least two such terms denoted by x^i and y^j, for $i, j = 1, ..., d$. Then the points $(i, 0)$ and $(0, j)$ belong to $\mathrm{Newt}(f)$. We want to show that $p = (i, 0)$ cannot but be a vertex or lying on an edge of $\mathrm{Newt}(f)$ (the case for $(0, j)$ is established similarly and so are the cases when there are more than one term in f of the form x^i or y^j). Suppose to the contrary that p is a strictly interior point of the polytope. By convexity of $\mathrm{Newt}(f)$, p should lie strictly in the interior of every angular sector formed by any three consecutive vertices. But then the x-axis would intersect at least one edge of the polytope in one point different from the two endpoints of the edge, so that $\mathrm{Newt}(f)$ contains points in the half-plane

$$\{(x, y) \in \mathbb{R}^2 | y < 0\},$$

a contradiction, since $\mathrm{Newt}(f)$ lies entirely in the quadrant of positive coordinates.

Corollary 4 *Let $f \in \mathbb{F}_p[x, y]$ be of total degree d. Then f has no trivial monomial factors of the form $f_{(i,j)}x^iy^j$ for some positive integers i and j and $f_{(i,j)} \in \mathbb{F}_p$ if and only if $\mathrm{Newt}(f)$ has at least two of its vertices on the x-axis and y-axis respectively.*

Proof: Suppose that f has no trivial monomial factors of the form $f_{(i,j)}x^iy^j$ for some positive integers i and j. Then f must have at least one term that has zero degree in x and one term that has zero degree in y. By the lemma above, $\mathrm{Newt}(f)$ would then have at least two vertices on the x and y axes. The converse is immediate to establish.

4 Pre-lifting stages

The pre-lifting sub-routines all require a non-trivial implementation touching upon issues about proper data structure and careful manipulation of geometric data. In the course of this description we explore the primary data types representing the geometric structures such as edges, vertices, straight lines, and dominating sets of edges. We also address how operations involving geometric structures with integer coordinates can be performed correctly, under the restriction that all such computations should involve input and output integer values only.

4.1 Floating-point operations and integer overflow

As will be seen shortly, many aspects of our C implementation are designed so as to avoid both instances of floating-point operations (unless the result can be guaranteed to be exact) and integer overflow. Because one can predict the possible occurrence of a floating-point operation before such an operation is carried out, this type of problem is easier to handle than integer overflow. On most current machines, signed integers use 32 bits which represent numbers in the range $\pm 2.10^9$. This sets the first restriction on the size of the input data representing the exponent pairs (e_1, e_2) of terms of f. However, when operations such as addition or multiplication of signed integers produce an output that exceeds this range, standard C gives no error upon integer overflow. In the absence of a multi-precision package for dealing with arbitrary sizes of signed integers (see for e.g. [15]), one may opt to use doubles to represent integers, so that integer calculation can be performed accurately with floating-point numbers. This, combined with tests to depict the safe range of any computation prior to its execution, makes up a good strategy for avoiding integer overflow. In doing this, one has to consider possible cancellations of even incorrect calculations of the generic form $ab + cd$. There, for instance, it suffices to test for the sizes of ab and cd prior to cancellation (addition) and decide whether or not the upper bound on the size of the final sum exceeds the allowed bound.

Assuming the order of the finite field in question fits in a machine word, all field operations will be referred to as bit operations, and the spatial complexity will be measured in bits.

4.2 Computing $\mathrm{Newt}(f)$

The first phase of the algorithm consists in computing $\mathrm{Newt}(f)$ using a convex hull algorithm. We choose to use Graham's fast algorithm of complexity $O(t \log t)$ for an input of size t (see [16]), despite the fact that slower algorithms of quadratic time in t would still be efficient in the sparse case. We also adopt the efficient variant of Graham's algorithm found in [25]. The input polynomial f is given as a collection of points $\mathcal{P}$ representing the exponent vectors of terms of f. Graham's algorithm above produces a stack of vertices of $\mathrm{Newt}(f)$, which uniquely describes the entire polytope, since it is sufficient to store information about the vertices of a convex set to retrieve any further information about its edges or interior lattice points. The stack of vertices is built in a counter-clockwise order around a fixed pivot, chosen to be the lowest and rightmost of all input points. If m denotes the total number of vertices of $\mathrm{Newt}(f)$, the stack is represented by a singly linked list of pointers to the vertices $V_0, ..., V_{m-1}$. In turn, each vertex is a structure containing information about the index of the vertex in the stack, as well as the x coordinate (abscissa) and the y coordinate (ordinate) of the vertex. For further uses in the algorithm, we also store the edge description of $\mathrm{Newt}(f)$ as follows. For $i = 0, ..., m-1$, let E_i denote the edge defined by $V_{i+1} - V_i$. If n_i denotes the gcd of the two components of the edge vector, then E_i can be written as $n_i e_i$ for a primitive vector e_i whose components (a_i, b_i) are relatively prime. We shall adopt this notation throughout the text.

4.2.1 Representing terms of f

Since terms of f (and thereafter specialised terms of g and h) will have to be accessed during every lifting step, one has to modify the representation of f, originally given as an arbitrarily ordered collection of points, to allow quick accessibility. Ideally, this would be through the use of a dense representation, whereby the nonzero coefficient of a term $f_{(e_1,e_2)} x^{e_1} y^{e_2}$ of f is stored in the array location (e_1, e_2). A possible solution would be to balance the time it takes to search for a particular term and the total memory required for storing all of them, through the use of a "semi-sparse" representation, so long as this requires no more than the largest structure used in the entire algorithm, which will be shown later to be $O(t^{O(1)} d)$ bits of memory.

To illustrate, suppose that y_{max} and x_{max} denote the largest degree in y and x respectively, and y_{min} and x_{min} denote the smallest degree in y and x respectively, among all terms of f. By Corollary 4, we know that y_{min} and x_{min} are both equal to zero when f is a "non-trivial" polynomial. We then have $y_{max} \leq d$ and $x_{max} \leq d$. Without loss of generality we shall always assume that $y_{max} \leq x_{max}$, and that all arrays have starting index equal to 0 (rather than 1). We can now define a recursive structure as follows. Let $fterms$ denote an integer array of size y_{max} such that each entry $fterms[k]$ denotes the number of terms of f whose degree in y is $k-1$. Although this makes the array $fterms$ a dense one, it can now be used to incorporate a sparse data structure as follows: for $k = 0, ..., y_{max} - 1$ and $j = 0, ..., fterms[k] - 1$, define a list of integers $fabs_k$ such that the j'th element in the list contains the degree in x of the j'th term of f belonging to the list of terms of degree k in y. A similar list can be constructed to store the coefficients of terms over $\mathbb{F}_p$. In the worst case analysis, all terms of f will have the same degree in y, and we can allow the above sparse structure to occupy at most $O(td)$ bits of memory. Assuming that the coordinates of the input polynomial are no larger than a machine word size, and combining the requirements for storing the output in Graham's algorithm above, where the number of vertices of $\mathrm{Newt}(f)$ is of the order $O(t)$ ([9]), the total spatial complexity of this stage is dominated by $O(td)$ bits. With this structure, we can decide for the existence of a term $f_{(e_1,e_2)}x^{e_1}y^{e_2}$ of f through a simple scan of the list $fabs_{e_2-1}$ which contains at most $O(t)$ elements, so that a naive search is of the order $O(t)$.

4.3 Finding all irredundant sets of dominating edges

We now describe an algorithm for identifying all possible irredundant dominating sets of the polygon. This is preceded by some results which we present as follows:

Lemma 5 *Let P denote a convex polygon with m vertices in $\mathbb{R}^2$ ordered cyclically around a chosen pivot V_0 in a counter-clockwise direction. Let V_{i-1}, V_i, and V_{i+1}, for $i \geq 1$, denote any three consecutive vertices, and let v denote an arbitrary point in $\mathbb{R}^2$ different from V_{i-1}, V_i, and V_{i+1}. Then the line $(V_i v)$ cuts P only at V_i if and only if it does not lie in the angular sector defined by the two vectors $(V_i V_{i-1})$ and $(V_i V_{i+1})$.*

Proof: First, note that, since P is convex, the line segment joining V_{i-1} to V_{i+1} is completely lying in P. Suppose that $(V_i v)$ lies in the sector defined by the angle $(V_i V_{i-1}, V_i V_{i+1})$. Then $(V_i v)$ will necessarily intersect the line segment $[V_{i-1} V_{i+1}]$ at a point of P different from V_i. Conversely, suppose that $(V_i v)$ intersects P at a point v' different from V_i. By convexity of P, an arbitrary point of the plane is interior to P if and only if it lies to the left of the line supporting every directed edge of P. As a result, v' is to the left of the lines supporting e_i and e_{i+1}, where $e_i = V_i - V_{i-1}$ and $e_{i+1} = V_{i+1} - V_i$. Hence, the line $(V_i v)$ lies in the angular sector defined above.

Now, let V_{i-1}, V_i, V_{i+1} and v be as defined in Lemma 5. Let the *range of admissible slopes* of V_i, denoted by $\mathrm{admiss}(i)$, represent the union of all possible slopes of lines $(V_i v)$ such that $(V_i v)$ does not intersect the angular sector defined by $(V_i V_{i-1}, V_i V_{i+1})$.

Lemma 6 *Let P be a polygon with m vertices in $\mathbb{R}^2$ and let Λ denote a set of consecutive edges of P connecting any two vertices V_i and V_j, for $i, j = 0, ..., m-1$, and $i < j$. Then Λ is an irredundant set of dominating facets if and only if $admiss(i) \cap admiss(j) \neq \emptyset$.*

Proof:

1. Suppose that admiss(i) $\cap$ admiss(j) $\neq \emptyset$ and let a be an element of the intersection. We can then construct two parallel lines r_i and r_j passing through V_i and V_j respectively and having slope equal to a. Consider Mink(Λ, r), where r is a vector in $\mathbb{R}^2$ having the direction of the parallel lines. As such, each of the two infinite edges of Mink(Λ, r) contains exactly one point of Λ. To show that P is contained in Mink(Λ, r), it suffices to show that all vertices V_k, for $k = j+1, ..., m-1$, belong to Mink(Λ, r). Suppose there is one such vertex k not in the Minkowski sum. Then V_k lies outside the interior strip bounded by r_i and r_j. But this implies that either segment $[V_i, V_k]$ will intersect r_j in a point $v' \neq V_i, V_k$ or $[V_j, V_k]$ will intersect r_i in a point $v'' \neq V_j, V_k$. Since P is convex, v' or v'' is contained in P, a contradiction, by Lemma 5 and the fact that r_i and r_j have admissible slopes. Last, since Λ is a set of consecutive edges of P, it must be irredundant.

2. Proof of the converse is immediate by noting that, since Λ is an irredundant set of dominating facets connected by the two vertices V_i and V_j, we can construct two parallel lines r_i and r_j through V_i and V_j respectively such that P is contained in Mink(Λ, r), where r is a vector in $\mathbb{R}^2$ having the direction of the two parallel lines, and each of the two infinite edges of Mink(Λ, r) contains exactly one point of P, which implies that $admiss(i) \cap admiss(j) \neq \emptyset$.

The algorithm for finding all possible irredundant sets of dominating edges can now be stated as follows

Algorithm 7 Input: A polygon P in $\mathbb{R}^2$ with m vertices V_k, for $k = 0, ..., m-1$. Output: The collection of all irredundant sets of dominating edges of P, in the form $\{(i, j)\}$, for $i, j = 0, ..., m-1$ and $i < j$, where V_i and V_j are the first and last vertices to appear in any set of dominating edges.

Step 1: $S \leftarrow \emptyset$.

Step 2: For $i = 0, ..., m-1$ determine admiss(i) using Lemma 5 above.

Step 3: For $i = 0, ..., m-1$ do
    for $j = i+1, ..., m-1$ do
        if admiss(i) $\cap$ admiss(j) $\neq \emptyset$
            $S \leftarrow S \cup \{(i, j)\}$.

Step 3: Return S.

Proposition 8 *Algorithm 7 works correctly as specified and requires $O(m^2)$ arithmetic operations, where m is the number of vertices of polygon P.*

Proof: Correctness of the algorithm follows as a direct consequence of Lemma 6. The cost can be easily established by noting that the main loops in Step 3 are iterated at most m^2 times in total, and that all operations in the above algorithm require only arithmetic operations for calculating slopes and intersection of integral sets.

The implementation of the above process can be achieved through several implementations of the following:

4.3.1 Slopes of edges

We first need a careful manipulation of slopes which avoids any instances of floating-point operations while performing simple tasks such as determining and comparing slopes of lines supporting edges. In our implementation, slopes are always integral ratios a/b, which is one good property to start with. If $k \in \mathbb{Z}$ and $a, b \neq 0$, any equality of the form $a/b = k.(a/b)$ is not guaranteed to hold for floating point division in C. As a result, we have to treat this quantity as a discrete one. Using a recursive structure we define an array of pointers, $Slope$, indexed by the edges E_i, for $i = 0, ..., m-1$, such that the entry $Slope[i]$ points to a list of two elements: the numerator and denominator of the slope of E_i, in normalised form (having gcd equal to 1). When computing admiss(i) we are concerned with characterising all straight lines that fall in the angular sector whose vertex is V_i and whose rays are with endpoints V_{i-1} and V_{i+1}. Let s_1 and s_2 denote the slopes of lines $(V_{i-1}V_i)$ and (V_iV_{i+1}) respectively. Obviously, any straight line falling in the interior of the angular sector defined above has to have slope in the range

$$I = (min(s_1, s_2), max(s_1, s_2))$$

if s_1 and s_2 are of the same sign, or in the range

$$I = (-\infty, min(s_1, s_2)) \quad \cup \quad (max(s_1, s_2), +\infty)$$

otherwise. The set admiss(i) is then simply the complement of I, which can be a single interval or a union of two intervals. This can be represented by a structure of intervals of the form $A \cup B$, for two continuous intervals A and B, where only B can be empty. A continuous interval then extends recursively into a two dimensional array $Component$ of pointers such that $Component[0]$ and $Component[1]$ point to its lower and upper bounds respectively. In turn, such lower and upper bounds represent slopes of edges, and are hence represented by a two dimensional integer array $bound$, such that $bound[0]$ and $bound[1]$ denote the numerator and denominator of the quantity denoting slope. Of course, by this we understand that $bound[1] = 0$ whenever the corresponding slope is infinite.

4.3.2 Complements and intersections of sets

At this point we have at hand a representation of rational intervals that will allow us to perform the set operations of taking complements and of intersecting sets. If I denotes an interval or a union of two intervals as encountered previously, then $\mathbb{R} \setminus I$ can be trivially determined and must have the same representation of I as defined above. Furthermore, we will need to determine whether or not the intersection of two intervals $I = A \cup B$ and $I' = A' \cup B'$, for continuous intervals A, B, A' and B', is empty. That reduces easily to finding whether $A'' \cap B'' \neq \emptyset$ for some $A'' = (a, b)$ and $B'' = (a', b')$, where $a, a', b, b' \in \mathbb{Z} \cup \{\pm\infty\}$. To do this, it suffices to compare the corresponding lower and upper boundaries of the continuous intervals (a, b) and (a', b'), whose intersection is non-empty if and only if any of the following holds:

- both a and a' are $-\infty$,
- both b and b' are $+\infty$,
- only a is $-\infty$, b is finite, and $a' < b$,
- only a' is $-\infty$, b' is finite, and $a < b'$,
- only b' is $+\infty$, a' is finite, and $a' < b$,
- only b is $+\infty$, a is finite, and $a < b'$.
- a, b, a' and b' are finite, and $b > a'$ or $b' > a$.

According to our representation above, a, b, a' and b' are stored as rational quantities, where comparison of two such fractions n/d and n'/d' reduces to comparing the product nd' and $n'd$. Even though the numerators and denominators are bounded by the coordinates of exponent vectors of terms of f, performing the above integer products may result in overflow. For this, a possible test can be inserted at the beginning of each such multiplication to ensure that the size of any of the intervals boundaries are bounded by at most square root of $\pm 2.10^9$.

When $\text{admiss}(i) \cap \text{admiss}(j) \neq \emptyset$ for $i = 0, ..., m-1$ and $j = i+1, ..., m-1$, we conclude that the edges connecting the two vertices V_i and V_j form an irredundant set of dominating edges. In fact, one can conclude that any two such vertices result in two dominating sets, particularly, the counter-clockwise and clockwise sequence of edges connecting them. All such sets are represented by a singly linked list *Dominating_set* ordered according to increasing values of i. The k'th element of the list points to the two integers i and j such that V_i and V_j form the k'th irredundant dominating set. Recall from the discussion above that the entire procedure requires an order of $O(m^2)$ operations of set intersections and complements. These in turn require no more than integer multiplication and comparison. Assuming all such integers and intermediary products fit in a machine word, the total cost of this stage is of the order $O(m^2)$ bit operations. We have also seen that the amount of storage needed does not exceed eight integers per vertex (representing the total number of numerators and denominators of rational boundaries of intervals representing $\text{admiss}(i)$ for some i), as well as two integers per dominating set, whose number itself is dominated by m. As a result, this sub-routine requires at most $O(m) = O(t)$ bits of storage.

4.4 Determining univariate edge polynomials

At this point, we shall make the distinction between a sparse and dense polynomial representation as defined throughout the text. In particular, we denote by a sparse polynomial structure any such structure where only information about the exponents of the terms is available, even when the corresponding polynomial is not sparse enough. In the rest of the text it will be assumed that all entries in a sparse polynomial representation are ordered according to increasing values of exponents. For simplicity, we shall also always assume that the coefficients of terms in a sparse representation are stored in a structure matching the one used for exponents, and it will be implicit everywhere in our discussion that coefficients of terms are retrieved whenever their exponents are so.

On the other hand, we denote by a dense polynomial structure any such structure where information about the (zero and nonzero) coefficients of the corresponding polynomial is available, as indexed by the degrees of their terms. In the worst case analysis, both sparse and dense representations will require the same amount of storage for dense polynomials.

When f is sparse, so are the corresponding univariate edge polynomials along $\mathrm{Newt}(f)$. Thus, we require that they be represented using a sparse data structure. The entire process of determining these polynomials depends on a number of sub-tasks, such as identifying integral points belonging to the edge, choosing only those points (e_1, e_2) corresponding to a term of f, and determining the corresponding term in z as defined by the change of basis in Step 4 of Algorithm 2.

4.4.1 Identifying integral points

Let $\delta \in \mathrm{Edge}(f)$ where $\delta = n_i e_i = V_{i+1} - V_i$ for some $i = 0, ..., m-1$, so that δ has $n_i + 1$ integral points lying on it. To identify each of these points, one can start from one of the endpoints, say $V_i = (x_i, y_i)$, and use the gradients as defined by the slope of the line supporting the edge. Recall that the normalised slope $s = a/b$ of the line supporting δ can be retrieved using the pointer stored in $Slope[i]$, and hence, all integral points on the line supporting δ can be described by

$$x = x_i + kb, y = y_i + ka,$$

for $k \in \mathbb{Z}$. In particular, points (e_1, e_2) lying between V_{i+1} and V_i are defined by $k = 1, ..., k' - 1$, where $k' = (x_{i+1} - x_i)/b$. One can then test whether (e_1, e_2) is an exponent corresponding to a term of f by scanning the list $fabs_{e_2-1}$, this requiring no more than $O(t)$ bit operations. By Corollary 26 of [3], we know that, for a polynomial $f \in \mathbb{F}_p[x, y]$ of degree d, each edge of $\mathrm{Newt}(f)$ will have at most $O(d)$ integral points lying on it. Throughout the text, we shall refer to this number as max_int_pts.

4.4.2 Change of basis

To determine the change of basis associated with δ one first has to determine its associated primitive affine function as defined in the introduction above. Given only the slope and two end points of δ, one can first derive the equation of its supporting line. If (x_i, y_i) and $s = a/b$ are as defined above, then the equation of this line is given by

$$\frac{y - y_i}{x - x_i} = \frac{a}{b}.$$

To avoid any floating-point operations associated with the right hand side division, we view this as

$$-ax + by - (by_i - ax_i) = 0,$$

where we are faced with a possible integer overflow upon the calculation of $by_i - ax_i$. This, of course, can be avoided through a pre-test on the sizes of by_i and ax_i as mentioned previously. The primitive affine function $l_\delta = \nu_1 x + \nu_2 y + \eta$ can now be derived from the equation of δ in a straightforward way. Since $\mathrm{Newt}(f)$ should lie in the non-negative halfplane $\{r \in \mathbb{R}^2 | l_\delta(r) \geq 0\}$, one can simply choose any vertex of $\mathrm{Newt}(f)$ different from the two endpoints of δ, and substitute its coordinates in the equation of the line computed above. If the result is positive, we set

$$\nu_1 = -a, \nu_2 = b, \quad \text{and } \eta = -(by_i - ax_i).$$

Else, we set

$$\nu_1 = a, \nu_2 = -b, \quad \text{and } \eta = (by_i - ax_i).$$

These coefficients can now be stored in an integer array indexed by the position of δ in the stack of edges. Finally, we call the Extended Euclidean algorithm to compute the integers ζ_1 and ζ_2 such that

$$\zeta_1.\nu_1 + \zeta_2.\nu_2 = 1,$$

and we store ζ_1 and ζ_2 contiguously next to ν_1, ν_2 and η. Assuming all input and intermediary integer values do not exceed the required bound, the above process per edge is dominated by a constant number of bit operations.

The change in basis as described in Step 4 of Algorithm 2 can be retrieved using the coefficients ν_1, ν_2, ζ_1, and ζ_2. Let (e'_1, e'_2) denote the coordinates of the starting vertex of δ. For each integral point of δ corresponding to a term $f_{(e_1,e_2)} x^{e_1} y^{e_2}$ in f, a univariate term in $f_{(e_1,e_2)} z^\alpha$ can be found using

$$\alpha = (e_2 - e'_2)/(-\nu_1) \text{ if } \nu_1 \neq 0, \quad \text{or } \alpha = (e_1 - e'_1)/(\nu_2) \text{ if } \nu_2 \neq 0,$$

and every such exponent is stored in the sparse data structure representing the sparse edge polynomial. Since all coordinates of points in $\mathrm{Newt}(f)$ are assumed to fit in a machine word, this requires no more than $O(1)$ bit operations per term. Combining the costs of the previous tasks, the whole process of constructing the univariate edge polynomials is of the order $O(md) = O(td)$ bit operations, and requires no more than $O(mt) = O(t^2)$ bits of memory.

4.5 Intersecting arbitrary lines with the polytope

In many of the sub-routines to follow it becomes essential to investigate how a geometric intersection between arbitrary straight lines and $\mathrm{Newt}(f)$ can be performed under the restriction that all computations have to receive and produce only integer values. Determining the intersection between an arbitrary straight line ℓ' and the polytope reduces to finding the intersection between ℓ' and all edges $\delta \in \mathrm{Edge}(f)$. The main problem then lies in that the intersection points between any two lines may not be lattice points. But then, they would simply not contribute to any terms in the lifted polynomials and hence the algorithm as a whole, which makes them dispensable for our application. A possible solution resides in considering other suitable points which can still serve the same purpose, that of identifying all possible points of the polytope corresponding to terms in particular lifted polynomials. For this, we alternatively introduce the notion of a *near* intersection point, to be that lattice point common to the line and the edge of the polytope that is closest (or at best identical) to the real intersection point. The crucial idea behind our approach depends on that if ℓ' intersects an edge of $\mathrm{Newt}(f)$ in some point, this should lie in the smallest rectangle $\mathcal{R}$ containing $\mathrm{Newt}(f)$ and whose edges fall on the lines of equations $x = 0, x = x_{max}, y = 0$ and $y = y_{max}$. That this can be found is a result of the fact that the convex hull computed above is the smallest convex polygon containing all points corresponding to terms of f. The entire sub-routine is then as follows:

Algorithm 9 $Intersection(Newt(f), u, v, w)$
Input: The vertex description of $Newt(f)$ and an arbitrary line ℓ' of generic equation $ux + vy + w = 0$.
Output: The near intersection points of ℓ' and $\mathrm{Newt}(f)$.

Step 1: Set $k' \leftarrow 0$, and $i', i'' \leftarrow -1$;
repeat

1.1: If $(vk'+w) \bmod u = 0$ and $-(vk'+w)/u \in \{0, ..., x_{max}\}$, set $i' \leftarrow -(vk'+w)/u$.
1.2: If $(i', k') \in \mathrm{Newt}(f)$ then exit the loop;
1.3: Set $k' \leftarrow k' + 1$.

while $k' \leq y_{max}$;

Step 2: If $k' < y_{max}$, set $k'' \leftarrow y_{max}$ and repeat:

2.1: If $(vk''+w) \bmod u = 0$ and $-(vk''+w)/u \in \{0, ..., x_{max}\}$, set $i'' \leftarrow -(vk'+w)/u$.
2.2: If $(i'', k'') \in \mathrm{Newt}(f)$ then exit the loop.
2.3: Set $k'' \leftarrow k'' - 1$.

while $k'' > k'$;

Step 3: If $i' \neq -1$ output (i', k'), and if $i'' \neq -1$ output (i'', k'').

Correctness of the algorithm can be shown as follows. If $ux + vy + w = 0$ denotes the generic equation of ℓ', we know that all lattice points (a, b) of ℓ' and lying in $\mathcal{R}$ have a y coordinate in the range $\{0, ..., y_{max}\}$ such that $(-vb - w)/u$ is an integer between 0 and x_{max}. A possible approach to finding *near* intersections consists in identifying (and then excluding) lattice points that belong to $\ell' \cap \mathcal{R}$ but not in $\mathrm{Newt}(f)$. By convexity of $\mathrm{Newt}(f)$, the latter collection of points are non-adjacent, and form lower and upper lattice points in $\ell' \cap \mathcal{R}$, whose ordinates belong to the union of the two intervals

$$[0, k'] \cup [k'', y_{max}]$$

for some integers k' and k''.

Before establishing the cost of the above algorithm, we need to discuss how to decide whether an arbitrary lattice point of the plane belongs to Newt(f). In a rather straightforward approach one would just compute the set of all points belonging to the polytope, so that testing an arbitrary point for inclusion becomes almost an immediate task. However, this requires about $\#\text{Newt}(f) = O(d^2)$ bits of storage, which is highly restrictive for high degree factorisations. Furthermore, our upcoming sub-algorithm for computing all integral points of Newt(f) requires that we perform intersections between arbitrary lines and the polytope, so that a more efficient test for inclusion in Newt(f) is needed. The test we propose works best when the number of edges is significantly less than $\#$Newt(f). Recall that, in Graham's algorithm above, we constructed the vertices and edges in a counter-clockwise direction around the pivot. A simple consequence of this and the fact that Newt(f) is convex is that an arbitrary lattice point belongs to the polytope if and only if it belongs to one of its edges, or it lies to the left of the directed line of each edge $\delta \in \text{Edge}(f)$. For this, we adopt the test for "leftedness" suggested in [25]: Given three arbitrary points $A = (a_1, a_2), B = (b_1, b_2)$, and $C = (c_1, c_2)$, C is to the left of the directed line AB if and only if the signed area of the counterclockwise triangle ABC is positive. The formula we use, derived from the cross product of the two vectors $B - A$ and $C - A$, produces twice the value of the above area, and is given by

$$(b_1 - a_1)(c_2 - a_2) - (c_1 - a_1)(b_2 - a_2).$$

Combining, we have the following:

Algorithm 10 Input: An arbitrary point $C = (c_1, c_2)$, and the vertex description of Newt(f).
Output: PASS if $C \in Newt(f)$, FAIL otherwise.

Step 1; For $i = 0, ..., m - 1$ do

1.1: Retrieve $V_{i+1} = (b_1, b_2)$ and $V_i = (a_1, a_2)$; compute

$$E = (b_1 - a_1)(c_2 - a_2) - (c_1 - a_1)(b_2 - a_2).$$

1.2: If $E < 0$ return(FAIL).

Step 2: Return(PASS);

Proposition 11 *Algorithm 10 works correctly and requires $O(t)$ bit operations.*

Proof: Correctness of the above procedure relies on that of the signed area test as discussed in [25]. It suffices for an arbitrary point to fail the test for only one edge of

Newt(f) for us to conclude that it does not belong to the polytope. The run time of the algorithm is dominated by the cost of computing the signed area for every edge of Newt(f). Assuming the integer computations in E are all correct and fitting in a machine word, this brings the total cost to $O(m) = O(t)$ bit operations.

Corollary 12 *Algorithm 9 for computing the near intersection points of an arbitrary straight line with* Newt(f) *requires* $O(td)$ *bit operations.*

Proof: The two major loops in Algorithm 9 iterate at most $O(d)$ times, during which one multiplication, one addition, and one division operation of integers are performed, along with a test for inclusion in Newt(f). The cost of the inner loop computations is dominated by that of testing for inclusion in Newt(f), which is of the order $O(t)$. The claim now follows.

4.6 Computing the set of all integral points in Newt(f)

Often throughout the rest of the algorithm, we will need to test for inclusion of arbitrary points in the polytope. Although the test in Algorithm 10 is crucial for the *Intersection* sub-routine, it can be costly to invoke this test very frequently. Hence, we introduce a more efficient method for identifying all integral points in Newt(f), which uses the above version of *Intersection* only once, but which can later be used as a less expensive test for inclusion.

Let IP denote the set of all integral points in Newt(f). The solution we provide is enhanced by the fact that nowhere in our sparse adaptation will we need to have all elements of IP available at one and the same time. Accordingly, it's sufficient to store a significantly smaller subset of IP that still allows us to either retrieve all of its elements or check whether an arbitrary point of the plane belongs to it. As discussed previously, one possible idea is to examine lattice points of the polytope that lie on every horizontal line $y = k$, for $k = 0, ..., y_{max}$. This can be done by computing the *near* points of intersection between all such horizontal lines and Newt(f). Since these can be either actual integral points of intersection or integral points that are closest to the intersection, we are sure that all elements of IP falling on the line $y = k$ should lie between the two *near* points of intersection. Repeating the procedure for all $y = 0, ..., y_{max}$ labels in this way all elements of IP, and more. Given an arbitrary point of the plane (a, b), we can define a boolean function which returns whether $(a, b) \in IP$ or not, by simply retrieving the *near* intersection points between Newt(f) and $y = b$. Obviously, a is an integer lying between the abscissas of the two *near* intersection points if and only if $(a, b) \in IP$.

The data structure we define for this sub-routine will be a recursive one, where an array of y_{max} pointers *Int_pts* is such that *Int_pts*$[y]$ points to another array of two integers, *Int_coordinates*$_y$, representing the x coordinates of the two possible points of intersection with ordinate equal to y. Since we know that the line $y = k$ intersects the polytope for $k = 0, ..., y_{max}$ in at least one *near* point, the first entry of *Int_coordinates*$_y$

contains its coordinate. If there exists another *near* point of intersection, we use the remaining entry of $Int\_coordinates_y$; else, we set this to be -1.

Algorithm 13 Input: The vertex description of Newt(f).
Output: The set IP of all integral points in Newt(f).

Step 1: For $y = 0, ..., y_{max}$ do

1.1: Call $Intersection(Newt(f), 0, 1, -y)$; store the x coordinate of the first near point of
intersection in the corresponding location of $Int\_coordinates_y$;

1.2: If there is another near point of intersection store its coordinates in the corresponding
location of $Int\_coordinates_y$, and output all points lying between the two intersection
points.

1.3: Else, output the only near point of intersection.

Proposition 14 *Algorithm 13 works correctly as specified and produces a description of* IP *in* $O(td^2)$ *bit operations and* $O(d)$ *bits of memory.*

Proof: Correctness follows easily from the discussion above. The run time depends on the size of the range $y = 0, ..., y_{max}$ as well as the cost of one call to the function $Intersection$, which is of the order $O(td)$. In total, this brings the cost of finding all integral points to $O(td^2)$. Since at most two integers less than or equal to d (and hence fitting in a machine word) are stored for $y = 0, ..., y_{max}$, where $y_{max} = O(d)$, the spatial complexity follows.

Assuming this subset of points in IP is produced only once at the beginning of the algorithm, testing for inclusion of an arbitrary point in the plane comes at no cost beyond that of referencing two array entries. In the remainder of this chapter, we shall denote by $In(IP, a, b)$ the function call which tests whether a point (a, b) belongs to IP. Moreover, we can now obtain a more efficient procedure for determining the intersection of an arbitrary straight line with Newt(f). In particular, we have:

Corollary 15 *Algorithm 9 for finding the intersection of an arbitrary straight line with* Newt(f) *can be modified to require* $O(t)$ *bit operations.*

Proof: The proof is immediate to establish, by noting that $Intersection$ can replace its sub-routine for testing inclusion by the test $In(IP)$.

Here and hereafter, we shall refer to the modified intersection sub-routine as $Intersection'$.

4.7 Counting factors of univariate factorisations

Once the edge polynomials have been stored in univariate form, we are ready to perform the factorisations over the defining field. For simplicity of code we choose to make use of the computer algebra package MAGMA †, whereby the output of all previous stages is directed to a file that can serve as a MAGMA code file within which lies its input. The input in this case comprises the following: all pieces of data previously computed and that will be needed in the lifting stage, among which are the vertices of $\mathrm{Newt}(f)$, and the list of sets of dominating edges. Also available are the univariate polynomials which MAGMA has to factorise using its built-in function (based on the Berlekamp algorithm [4]) for univariate polynomial factorisation over finite fields. The output of the MAGMA code is now directed into a file which has, in addition to the original information above, the full factorisation of univariate edge polynomials into powers of irreducibles, and which can be fed into the following sub-routines forming the third phase of the implementation.

In all what follows let $F_0^{(\delta)}$ denote the univariate edge polynomial associated with δ and let d_δ denote its degree. Assuming that irreducible factors of $F_0^{(\delta)}$ are stored in sparse representation, we define a list $irred_\delta$ of integer pointers such that element s of $irred_\delta$ points to the address in memory of the s irreducible factor of $f_0^{(\delta)}$ produced by the MAGMA code above. We also define an integer array $head\_irred$ of size at most $O(d)$ such that $head\_irred_\delta[j]$ points to the location in the list $irred_\delta$ of the first irreducible polynomial of degree j, for $j = 0, ..., d_\delta$. Assuming that $F_0^{(\delta)}$ has on average $O(\log d)$ irreducible factors ([19]), we define two integer arrays, mul_δ and $\deg_\delta$, of size $\log d$ each, such that $\mathrm{mul}_\delta[s]$ denotes the multiplicity in F_0^{δ}, and $\deg_\delta[s]$ denotes the degree, of the s irreducible in $irred_\delta$.

The counting algorithm we propose for computing the number $m_j^{(\delta)}$ of all possible degree j factors of $F_0^{(\delta)}$ is based on that any degree j factor R_j of $F_0^{(\delta)}$ has to satisfy the following:

1. $R_j = (irred_\delta^{(0)})^{p_0}.(irred_\delta^{(1)})^{p_1}...(irred_\delta^{(s)})^{p_s}$, where $\{irred_\delta^{(k)}\}_{k=0,...,s}$ represents the set of all irreducible factors of $F_0^{(\delta)}$ of degree less than or equal to j,
2. $p_k \leq \mathrm{mul}_\delta[k]$, for $k = 0, ..., s$,
3. $\sum_{k=0,...,s} \deg_\delta[k].p_k = j$.

Using this notation, counting the number of all possible polynomials R_j reduces to counting all possible ways one can construct an object with $s+1$ spots, each of which can be occupied by some integer flag p_k, for $p_k = 0, ..., \mathrm{mul}_\delta[k]$, and then to excluding those choices of R_j which fail condition 3 above. We restate this simple counting problem through the following recursive procedure:

Algorithm 16 Input: A degree d_δ univariate polynomial $F_0^{(\delta)}$ factorised completely into powers of irreducibles.

†See http://magma.maths.usyd.edu.au/magma/

Output: the number $m_j^{(\delta)}$ of degree j factors of $F_0^{(\delta)}$, for $j = 0, ..., d_\delta$.

Part I:

$Count\_divisors(d_\delta)$
Step 1: for $j = 0, ..., d_\delta$, do
1.1: Let *tail* denote the address in memory of the last irreducible factor of $F_0^{(\delta_k)}$ of degree j, and set
$spot \leftarrow tail$.
1.2: $Vary\_count(spot, j, tail)$.

Part II:

$Vary\_count(spot, j, tail)$
Step 1: Set $p_{spot} \leftarrow -1$;
repeat
1.1: Set $p_{spot} \leftarrow p_{spot} + 1$;
1.2: If $(spot = 0)$ do
If $\sum_{k=0,...,tail} \deg_\delta[k].p_k = j$, set $m_j^{(\delta)} \leftarrow m_j^{(\delta)} + 1$.
1.3: Else if $(spot > 0)$, call $Vary\_count(spot - 1, j, tail)$.

while $(p_{spot} < \text{mul}_\delta[spot])$.

Proposition 17 *Algorithm 16 works correctly as specified. When used to determine the number of factors of all possible degrees, it requires at most $O(d^{1+\log d} \log d)$ bit operations and $O(t(d + \log d))$ bits of memory, where $M(d)$ denotes the number of field operations required to perform multiplication of polynomials whose degree is bounded by d.*

Proof: For a fixed $j = 0, ..., d_\delta$, **Part I** invokes the recursive function $Vary\_Count$ using all irreducible factors of the edge polynomial of degree less than or equal to j. Those factors have indices $0, ..., tail$ in the list $irred_\delta$, so that each occupies a "spot". We shall establish correctness of the recursive function by induction on the number of spots. Let N denote this number. If $N = 1$, the call to the recursive function determines the number of ways one can form factors of $F_0^{(\delta)}$ using only the first irreducible factor, $irred_0$. All such possible factors are of the form $irred_0^{p_0}$, for $p_0 = 0, ..., \text{mul}_\delta[0]$. Now, suppose that the algorithm is true for $N - 1$. The call to $Vary\_count(N, j, tail)$ lists all possible powers that can occupy spot N, and hence all possible powers of the irreducible having index N in the list $irred_\delta$. For each such choice, a call to $Vary\_count(N - 1, j)$ is assumed to have produced all possible ways to form products of powers of irreducibles occupying locations $0, ..., N - 1$ in the list $irred_\delta$. Combining, this results in new ways to form products of powers of irreducibles occupying locations $0, ..., N$ in the list. When each such product has total degree j, a suitable factor would have been found, and $m_j^{(\delta)}$ is incremented by 1.

For a fixed j, $1 \leq j \leq d$, there are about $O(\log d)$ irreducibles (or spots) for every edge, and each spot k can have at most $\mathrm{mul}_\delta[k] \leq O(d)$ choices. When each choice is made a test is performed involving at most $O(\log d)$ multiplications and additions of integers bounded by d. Since $j \leq d$, the total run time now follows.

Since $F_0^{(\delta)}$ has about $O(\log d_\delta) = O(\log d)$ irreducible factors, the list $irred_\delta$ has size at most $O(\log d)$. The array $head\_irred_\delta$ is of size $O(d)$ since each of its entries points to locations in memory of the first irreducible factor of $F_0^{(\delta)}$ of some degree less than or equal to d. Hence, its entries are also bounded by $O(\log d)$ in value and thus fit in a machine word. Since multiplicities and degrees of all irreducible factors are bounded by d, the arrays mul_δ and $\deg_\delta$ are of size $O(\log d)$ and have entries which fit in a machine word. In total, the algorithm will require $O(d)$ bits per edge polynomial, and so $O(td)$ bits in total.

4.8 Summand counting and recovering algorithm

The summand counting algorithm used in [2] requires all elements of IP to be available in about $O(d^2)$ bits of memory, which makes it one of several bottlenecks we would like to address before attempting very high degree factorisations. Since our application is designed so as to specifically target sparse polynomials, we allow again the use of a "naive" summand counting algorithm, which, despite its being exponential in the number of edges m of $\mathrm{Newt}(f)$, requires negligible storage. Since $m = O(t)$, we expect this trade-off between memory and run time to be effective only for significantly sparse polynomials. The process can be described in very similar terms as in the counting procedure in Algorithm 16. By Lemma 13 of [13], an integral polygon is a summand of $\mathrm{Newt}(f)$ if and only if it has an edge sequence of the form $c_i e_i$, for $0 \leq c_i \leq n_i$, for some $c_i \neq 0$ and $c_{m-1} \neq n_{m-1}$, and $\sum_{0 \leq i < m} c_i e_i$ is the zero vector (the other summand is understood to have the edge sequence $\{(n_i - c_i)e_i\}$, for $i = 0, ..., m-1$). Hence, we can view any possible summand of $\mathrm{Newt}(f)$ as an object consisting of m spots, each of which has a vector e_i associated with it, and can be occupied by a component $c_i = 0, ..., n_i$, such that

$$\sum_{i=0,...,m-1} c_i e_i = 0.$$

This again entails a recursive procedure examining all possible choices of a summand with m edges, each of which has a possible length in the range $c_i = 0, ..., n_i$. The extra restrictions we place over the c_i's are that they should match the degrees of the known univariate factors of the edge polynomials f_0^δ such that $\delta = n_i e_i$. In particular, we exclude any value for c_i such that $m_{c_i}^{(\delta)} = 0$. Once a complete choice using some combination of scalar multiples of vector edges has been formed, we check if the resulting vector sum is zero, in which case a summand would have been found.

Since the above approach uses information about the edges rather then the vertices of $\mathrm{Newt}(f)$, any of its possible summands will be produced using an edge sequence description. The data structure we use for edges is a circular doubly linked list of pointers: Each element of the list links to previous and following neighbours, and the last element links to the first. Each pointer is associated with some edge δ' in the summand, and points to an array *Edge* of integers such that $Edge_{\delta'}[0]$ and $Edge_{\delta'}[1]$ denote the respective x and y components of δ'. The algorithm can now be described as follows:

Algorithm 18 Input: The edge description $\{n_i e_i\}_{i=0,\ldots,m-1}$ of $\mathrm{Newt}(f)$, the set $\{m_j^{(i)} \mid 0 \leq j \leq \deg(F_0^{\delta_i})\}$, where $m_j^{(i)}$ is the number of degree j factors of $F_0^{\delta_i}$, and an upper bound M on the total number of summands.
Output: The edge description of all possible pairs Q, R such that $\mathrm{Newt}(f) = Q + R$, or "failure" if the number of such decompositions exceeds some large bound M.

Part I: $Count\_summands()$

Step 1: Set $spot \leftarrow m-1$, $first\_spot \leftarrow 0$.

Step 2: Call $Vary\_choice'(spot, first\_spot)$;

Part II: $Vary\_choice'(spot, first\_spot)$

Step 1: Set $c_{spot} \leftarrow -1$; repeat

1.1: $c_{spot} \leftarrow c_{spot} + 1$;

1.2: If $m_{c_{spot}}^{(spot)} > 0$ do

1.2.1: If $spot = first\_spot$, check if the chosen edge sequence forms the zero vector and that the sequence $c_i e_i$, for $i = 0, \ldots, m-1$, is not trivial. If so, output this summand.

1.2.2: Else, if $spot > last\_spot$, call $Vary\_choice'(spot - 1, first\_spot)$.

while $(c_{spot} < n_{spot})$.

Step 2: If total number of summands exceeds M, halt the polytope algorithm.

Proposition 19 *Algorithm 18 works correctly and requires $O(td^t)$ bit operations and no more than $O(t)$ bits of memory to list all pairs of summands.*

Proof: Correctness follows similarly as in the proof of Proposition 17 above, and we leave the details for the reader. To establish the running time, we first know that the maximum over all n_i's, denoting the maximum number of integral points along any edge of $\mathrm{Newt}(f)$, is $O(d)$. The naive method has to count over $O(d^m) = O(d^t)$ different summands of $\mathrm{Newt}(f)$, where every count is accompanied by $O(m)$ additions and multiplications of vector coordinates, each of which is an integer less than or equal to d, and hence can fit in a machine word for values of d used in our application. If the number of all possible pairs of summands is larger than some large parameter M, the entire code is halted producing Failure as in Algorithm 2 above.

Note that we do not need to keep information about more than one pair of summands at a time. For each such non-trivial pair, we can carry out further computations including the lifting stage. If unsuccessful, the process can be repeated using a different pair of summands which occupies the space of its predecessor. As such, the amount of storage needed is $O(m) = O(t)$ bits of memory, where we understand that the component vectors describing the edges of any summand are bounded in size by the component vectors of edges of $\mathrm{Newt}(f)$.

4.8.1 Recovering a vertex description of the summands

In the above algorithm, we obtained only the edge sequence describing a pair of summands Q and R of $\mathrm{Newt}(f)$, which describes a unique decomposition up to translation with an arbitrary vector in $\mathbb{R}^2$. However, it is essential that we identify which of these translated summands will correspond to possible factors of f. In particular, the following consequence of Corollary 4 requires that we seek a vertex description allowing the proper translation of Q and R according to the fact below:

Corollary 20 *Let $f \in \mathbb{F}_p[x, y]$ be of total degree d such that f has no trivial monomial factors of the form $f_{(i,j)}x^iy^j$ for some positive integers i and j, $f_{(i,j)} \in \mathbb{F}_p$, and let Q be any summand of* $\mathrm{Newt}(f)$ *that corresponds to a possible factor g of f. Then Q must have at least two vertices or edges on the x-axis and y-axis respectively.*

Proof: Suppose Q is a summand of $\mathrm{Newt}(f)$ corresponding to a possible non-trivial factor g of f. For f as above, g must have no trivial monomial factors of the form $f_{(i,j)}x^iy^j$. By Corollary 4, Q must have at least two vertices on the x and y axes.

The data structure we use for the vertex description of the summands is the stack of points as used in the representation of $\mathrm{Newt}(f)$. For consistency, we will always assume that each of the summands has m vertices and edges, though some of the edges may be trivial, in which case we also define appropriate trivial vertices with coordinate values $(-1, -1)$, as we show in the following procedure.

Let $\{q_i = c_ie_i\}_{i=0,\dots,m-1}$ denote the edge sequence of a summand of $\mathrm{Newt}(f)$. We first choose an arbitrary point to be the pivot V_0, such as the origin of coordinates $(0, 0)$ for instance, and then define an auxiliary vector sum, *sum*, initialised to zero. We then build around the pivot by adding to *sum* the vector edges given one at a time. Since the edges are directed in a counter-clockwise fashion around the pivot as in $\mathrm{Newt}(f)$, we expect to build the vertices in a stack structure. If some edge $q_i = V_{i+1} - V_i$ is trivial, we insert a trivial vertex at the appropriate index of the stack as follows. If q_i is followed by a non-trivial edge q_j, for some $j = i+1, \dots, m-1$, then V_{i+1} is a trivial vertex. Else, if q_i is not followed by any non-trivial edge, then, since the last non-trivial edge of the summand is q_{i-1}, it must connect V_i to V_0, so that V_i coincides with the pivot, at which point the edge sequence forms a closed zero sum. Accordingly, we define all the remaining vertices V_j, for $j \geq i$, to be trivial. When all non-trivial vertices have been determined with $(0, 0)$ as the pivot, we translate the polytope so that the following is satisfied:

1. Q lies completely in the positive quadrant $\{(x, y) | x, y \geq 0\}$,

2. Q intersects the x-axis and y-axis in at least one point respectively.

In simpler terms, the lowest abscissa and ordinate among all coordinates of vertices should be zero. As a result, it suffices to determine

$$a = min(x_i)_{i=0,...,m-1} \text{ and } b = min(y_i)_{i=0,...,m-1}$$

for $V_i = (x_i, y_i)$ and to translate each of the vertices of Q by the vector $(-a, -b)$. The procedure can now be summarised as follows:

Algorithm 21 Input: The edge sequence $q_i = \{c_i e_i\}_{i=0,...,m-1}$ of a summand Q of Newt(f), where not all the c_i's are zero and where $c_{m-1} \neq n_{m-1}$.
Output: The vertices $V_0, ..., V_{m-1}$ of Q.

Step 1: Set $sum \leftarrow (0,0)$, $V_0 \leftarrow (0,0)$;
For $i = 0, ..., m-1$ do

- 1.1: If q_i is not trivial, set $V_{i+1} \leftarrow sum + q_i$ and $sum \leftarrow sum + q_i$;
- 1.2: Else, if q_i is followed by a non-trivial edge, set $V_{i+1} \leftarrow (-1, -1)$;
- 1.3: Else, for $j = i, ..., m-1$, set $V_j \leftarrow (-1, -1)$.

Step 2: Let a and b denote the lowest abscissas and ordinates among all coordinates of the non-trivial vertices; translate all non-trivial vertices of Q by $(-a, -b)$.

The proof is immediate, bearing that $m = O(t)$ and that a vector sum in our application reduces to the addition of two integers which do not exceed d, and hence which fit in a machine word.

4.8.2 Computing the integral points belonging to Q and R

Once a vertex description of the appropriate translated images of summands has been computed, we can determine the sets of integral points IP_g and IP_h belonging to Q and R respectively, as we have seen earlier in the case of Newt(f), in $O(td^2)$ bit operations. We further keep track of two indices, denoted by rem_g and rem_h, and initialised to $\#IP_g$ and $\#IP_h$ respectively. The indices are decreased by 1 every time a new coefficient is specialised, so that a total specialisation of coefficients of g or h is reached when any of rem_g or rem_h is zero. For future use, we also determine the number of integral points gn_i falling on every edge δ_i' of Q, using the $O(d)$ technique of Subsection 4.4.1. We also note that the location in memory used to store information about the summands can be reused upon each new choice.

4.9 Choosing coprime dominating edges factorisations

Given a fixed pair of summands of Newt(f) and a set of all irredundant sets Γ of dominating edges, we want to identify all resulting coprime dominating edges factorisations. For every pair of vertices V_i and V_j such that $i < j$ and whose connecting edges form a dominating set, we can choose Γ to be the (forward) sequence of edges $n_i e_i, ..., n_{j-1} e_{j-1}$

or the (backward) sequence $n_je_j, ..., n_{m-1}e_{m-1}, n_0e_0, ..., n_{i-1}e_{i-1}$. Let Γ denote a fixed irredundant set of dominating edges of cardinality m', and suppose that all edges in Γ have been relabelled to occupy an index in $\{0, ..., m'-1\}$. We can generate a sequence of coprime edges factorisations by examining all possible ways we can form a m' sequence of polynomials $G_0^{(\delta_i)}$, for $i = 0, ..., m'-1$, each satisfying the following:

1. $G_0^{(\delta_i)} = irred_0^{p_0} ..., irred_s^{p_s}$, where $\{irred_k\}_{k=0,...,s}$ denotes the list of all irreducible factor of $F_0^{(\delta_i)}$ of degree less than or equal to length of δ_i.
2. $p_k = 0$ or $p_k = \text{mul}_{\delta_i}[k]$, for $k = 0, ..., s$.
3. $\sum_{k=0,...,s} p_s.\deg_{\delta_i}[k] = ||\delta_i||$.

Here, $||\delta_i||$ denotes the length of edge δ_i. The second condition above guarantees a coprime edges factorisation, since

$$H_0^{(\delta_i)} = \frac{F_0^{(\delta_i)}}{G_0^{(\delta_i)}} = irred_0^{\text{mul}_{\delta_i}[0]-p_0} ... irred_s^{\text{mul}_{\delta_i}[s]-p_s},$$

so that $G_0^{(\delta_i)}$ and $H_0^{(\delta_i)}$ have a non-trivial factor in common if and only if there exists at least one k such that $irred_k$ has a non-trivial multiplicity different from 0 and $\text{mul}_{\delta_i}[k]$ in both $G_0^{(\delta_i)}$ and $H_0^{(\delta_i)}$. Since we need only one edge polynomial decomposition $F_0^{\delta_i} = G_0^{\delta_i} H_0^{\delta_i}$ at a time to make a sequence of factorisations using all $\delta_i \in \Gamma$, and only one sequence of edges factorisations at a time to initiate lifting, we can interleave the two processes as follows:

Algorithm 22 Input: A fixed integral decomposition $\text{Newt}(f) = Q + R$ and all factorisations of the edge polynomials into powers of irreducibles.
Output: All possible coprime dominating edges factorisations associated with Q and R.

Part I: *Choose_coprime_factorisations*

Step 1: For all pairs of vertices (V_i, V_j) of $\text{Newt}(f)$, where $i = 0, ..., m-1$ and $j = i+1, ..., m-1$, and where the edges connecting V_i and V_j form an irredundant set of dominating edges:
while no factorisation has been achieved and any of the irredundant sets has not been used, do:

1.1: Consider the two edge sequences comprising Γ given by

$$\{n_ie_i, ..., n_{j-1}e_{j-1}\} \quad \text{and} \quad \{n_je_j, ..., n_{m-1}e_{m-1}, n_0e_0, ..., n_{j-1}e_{j-1}\}.$$

For each such sequence, and while no factorisation has been achieved, do:

1.1.1: Set $m' \leftarrow \#\Gamma$ and perform a change of index so that the edges in Γ are relabelled as $\{\delta_0, ..., \delta_{m'-1}\}$.

1.1.2: Let $\{\delta'_0, ..., \delta'_{m'-1}\}$ be the corresponding set of edges in Q such that δ'_k is a summand of δ_k, for $k = 0, ..., m' - 1$.

1.1.3: Let *tail* denote the address in memory of the last irreducible factor of $F_0^{(\delta_k)}$ of degree equal to $||\delta'_k||$.

1.1.4: Set $k \leftarrow m'-1$, and $spot \leftarrow tail$, and call $Vary\_choice\_across\_edges(\delta_k, \delta'_k, spot, tail)$.

Part II: $Vary\_choice\_across\_edges(\delta_k, \delta'_k, spot, tail)$

Step 1: Set $power_{(spot)} \leftarrow -1$;
repeat

1.1: If $power_{(spot)} = -1$, set $power_{(spot)} \leftarrow power_{(spot)} + 1$.

1.2: Else, set $power_{(spot)} \leftarrow \mathrm{mul}_{\delta_k}[spot]$.

1.3: If $spot = 0$, check if a coprime edges factorisation corresponding to the lengths of the summand edges has been found:

1.3.1: If $||\delta'_k|| = 0$, set

$$G_0^{(\delta_k)} = 1 \text{ and } H_0^{(\delta_k)} = F_0^{(\delta_k)}$$

and go to 1.3.3.

1.3.2: Else, find the product $Poly$ of all polynomials $poly_r$, for $r = 0, ..., tail$, such that $poly_r = irred_{\delta_k}[r]^{power_{(r)}}$. If $\deg(Poly) = ||\delta'_k||$, set

$$G_0^{(\delta_k)} \leftarrow Poly \text{ and } H_0^{(\delta_k)} \leftarrow F_0^{(\delta_k)}/G_0^{(\delta_k)}$$

and go to 1.3.3. This will constitute the coprime factorisation associated with edge δ'_k.

1.3.3: If δ_k is the first edge in Γ, all edges now have a coprime factorisation associated with them, and we start lifting using all dominating boundary factorisations.

1.3.4: Else, repeat Step 1 in **Part II** above for the preceding edge δ_{k-1}:
Let *tail* denote the address in memory of the last irreducible factor of $F_0^{(\delta_{k-1})}$ of degree equal to $||\delta'_{k-1}||$, set $spot \leftarrow tail$, and call $Vary\_choice\_across\_edges(\delta_{k-1}, \delta'_{k-1}, spot, tail)$.

Step 1.4: Else if $spot > 0$, we still need to choose powers of irreducible factors of $F_0^{(\delta_k)}$ of index $spot-1, spot-2, ..., 0$: Call $Vary\_choice\_across\_edges(\delta_k, \delta'_k, spot-1, tail)$.

while ($power_{(spot)} < \mathrm{mul}_{\delta_k}[spot]$ and factorisation still not achieved).

Proposition 23 *Algorithm 22 works correctly and requires $O(t^2 d^{1+\log d} \log d)$ bit operations and $O(td)$ bits of memory.*

Proof: We establish correctness by induction on the number of edges in Γ. If $m' = 1$, the procedure above reduces to finding all possible factors of $F_0^{(\delta_0)}$ of degree $||\delta_0'||$. Correctness in this case can be proven by induction as in the proof of Proposition 17 above. Now, suppose the algorithm is correct when the number of edges in Γ is less than m'. For the fixed edge $\delta_{m'-1} \in \Gamma$, the main loop in **Part II** examines all possible factors of $F_0^{(\delta_{m'-1})}$ of degree $||\delta'_{m'-1}||$, which is correct as shown in Proposition 17. For each such factor whose degree matches the length of $\delta'_{m'-1}$, and by the induction hypothesis, the call to $Vary\_choice\_across\_edges(\delta_{m'-2}, \delta'_{m-2}, spot, tail)$ determines all possible coprime edges factorisations along edges δ_k, for $k = 0, ..., m' - 2$. But now a full choice using one factor of $F_0^{(\delta_{m'-1})}$ and all possible factors of $F_0^{(\delta_k)}$, for $k = 0, ..., m' - 2$, has been made. At this stage, one can then initiate lifting using coprime factorisations for all edges in Γ.

The worst case run time of the algorithm can be derived from the fact that one would have to try all possible coprime factorisations using Γ. By Proposition 17, finding all possible choices of polynomials per edge is of the order $O(d^{1+\log d} \log d)$. Across all edges of Γ, this becomes of the order $O(md^{1+\log d} \log d)$. Assuming that there are at most $O(m)$ pairs of vertices (V_i, V_j) connecting two sets of dominating edges, the total worst case complexity is of the order $O(m^2 d^{1+\log d} \log d) = O(t^2 d^{1+\log d} \log d)$.

Now, since any of the summands Q and R will have at most $\#\mathrm{Newt}(f) = O(d^2)$ integral points, the maximum number of integral points found along any edge of Q or R is bounded by that same number found along any edge of $\mathrm{Newt}(f)$, which is $O(d)$. The amount of storage needed in the above algorithm is for one coprime edges factorisation only, where the univariate edge polynomials corresponding to Q or R have degrees bounded by $O(d)$, so that the memory needed is about $O(td)$ bits.

Once a coprime dominating edges factorisation has been chosen, we can perform further characterisations of the corresponding edge polynomials in Q and R. First, and for all pairs of coprime edge polynomials G_0 and H_0, we compute the unique polynomials U and V such that

$$UG_0 + VH_0 = 1$$

and $\deg(U) < \deg(H_0), \deg(V) < \deg(G_0)$, using the Extended Euclidean algorithm for polynomials. Even though G_0 and H_0 are sparse, the polynomials U and V may be dense themselves, having about $O(d)$ terms. However, we shall store these in a sparse data structure whereby only information about the terms is available. We also determine other edge characteristics, such as the primitive affine function associated with every edge of the summands. Given $\delta \in \mathrm{Edge}(f)$ and δ' its summand in Q, we have that

$$l_{\delta'}(x, y) = l_\delta(x, y) - c_\delta = \nu_1 x + \nu_2 y + \eta - c_\delta$$

for some unique integer c_δ [1]. Since ν_1, ν_2 and η have already been computed, it suffices to determine and store only c_δ, by finding the equation of the straight line with slope $-\nu_1/\nu_2$ and passing through one of the vertices of δ', as shown in Section 4 above. This immediately determines the primitive affine function of the edge $\delta'' = \delta - \delta'$. Furthermore, we can now use the coprime dominating edges factorisation chosen above to specialise a subset of the coefficients of the possible factors g and h corresponding to the fixed pair of summands we have used above. In particular, if we write

$$G_0^{(\delta'_k)} = \sum_{s=0,\ldots,||\delta'_k||-1} g_s z^s,$$

and if (a,b) denotes the coordinates of the head of the edge δ'_k, then, by Lemma 9 of [1], each term $g_s z^s$ of $G_0^{(\delta'_k)}$ will result in the specialisation of a coefficient of a term $g_s x^i y^j$ in g, such that

$$i = \nu_2.s + a \quad \text{ and } j = -\nu_1.s + b,$$

where (a,b) denote the coordinates of the starting vertex of δ'. As in the sparse representation of terms of f, we store only these terms of g and h specialised so far. The data structure is identical to the one described in Section 4 for representing terms of f, with gy_{max} and hy_{max}, gx_{max} and hx_{max}, $gterms$ and $hterms$, and $gabs$ and $habs$ the analogous terms of y_{max}, y_{min}, $yterms$, and $fabs$ respectively. This would require at most $O(td)$ bits of memory.

5 The sparse lifting algorithm

With the exception of the summand counting algorithm and the corresponding subroutine for determining dominating coprime edges factorisations, all the previous tasks comprise the pre-computation phase, whose complexity bounds are dominated by $O(d^{1+\log d}\log d)$ bit operations and $O(t^2+d)$ bits of storage. As noted in [1], the summands and boundary factorisations need not be all computed at once. Lifting can be initiated for each coprime edges factorisation one at a time until a factor is found or the lifting procedure fails. Our empirical findings in Tables 1 and 2 demonstrate that in practice, and given very sparse polynomials as those defined in Section 3, the total number of dominating coprime edges factorisations is considerably less than the total degree of the input polynomial. For such class of polynomials, we will consider that the sparse lifting procedure will be invoked a number of times that is bounded by some small constant M. It is recommended that the polytope method be discarded once the number of coprime edges factorisations attempted exceeds this bound, and one revert to other "dense" methods [1].

The main sub-routine of the polytope lifting stage is as follows:

Algorithm 24 Input: A $(\Gamma, K; Q, R)$ coprime dominating edges factorisation, where $K = (\underline{1})_{\delta_i \in \Gamma}$, for $i = 0, \ldots, m' - 1$.
Output: A factorisation of f, or "failure".

While a factor of f has not been found and not all coefficients in Q and R have been specialised, do:

Step 1: For every $\delta_i \in \Gamma$ do

1.1: Retrieve its summand $\delta'_i \in Q$ and the corresponding primitive affine function $l_{\delta'_i}$.

1.2: Count the number of unspecialised terms $u_{\delta'_i}$ on the $k_{\delta_i}+1$ translate of the supporting

line of δ'_i into Q, and whose equation is defined by $l_{\delta'_i} = (k_{\delta_i}+1)$;

Step 2: Choose one edge δ_i whose summand satisfies:

. $u_{\delta'_i} < gn_{\delta'_i}$

. $u_{\delta'_i} = \max(u_{\delta'_s})_{s=0,\ldots,m'-1}$

Step 3: Set $k_{\delta_i} \leftarrow k_{\delta_i}+1$ and perform a K lifting of the given partial factorisation. If this

extension produces failure, exit the loop and choose a new coprime edges factorisation.

End.

The pseudo-code above mostly reflects the operations in Step 4 of Algorithm 2, which has been proven to terminate either with a failure or with a factor of f (see [1] for full details). The only slight modification is in choosing the suitable edge to lift from. Primarily, one has to choose δ' such that the number of unspecialised terms on its $k_\delta+1$ translate is less than the number of integral points on δ'. In the dense implementation of [1], preference was given to "shorter" edges even though lifting from these edges revealed a smaller number of coefficients of g and h, since their corresponding lifted univariate polynomials had smaller degrees, and hence could be processed faster by polynomial arithmetic sub-routines. However, this argument does not hold in our sparse adaptation, where the complexity of sparse polynomial arithmetic becomes dependent on the number of terms in a polynomial rather than its degree. Also, since we expect many of the lifted univariate polynomials to be zero, non-trivial polynomial arithmetic is performed only very rarely. Preference is thus shifted to longer edges which reveal more coefficients per lifting step.

To establish the run time and spatial complexity of the above lifting module, we shall need to investigate each of its inner sub-routines, for which the rest of this section is dedicated.

5.1 Detecting specialised coefficients

A crucial aspect of our sparse adaptation consists in that only nonzero terms of g or h get stored as they are revealed during the lifting stages. Consequently, it becomes essential to find efficient ways of identifying whether or not an arbitrary point in Q (or R) corresponds to a specialised term of g (or h), a task which is otherwise immediate in a dense implementation, where information about all the lattice points is stored. For this, we propose the following algorithm:

Algorithm 25 Input: An arbitrary point (i,j) of Q, and a partial (Γ,K)-factorisation extending a coprime dominating edges factorisation.

Output: PASS if (i,j) corresponds to a specialised coefficient of g, and FAIL otherwise.

Step 1: For every $\delta \in \Gamma$, let δ' denote its summand in Q, let k_δ denote the entry in the K vector indexed by δ, and let $l_{\delta'}$ denote the primitive affine function associated with δ'.

Step 1.1: If $l_{\delta'}(i,j) < k_\delta$ return(PASS).

Step 2: Return(FAIL);

Proposition 26 *Algorithm 25 works correctly and requires $O(t)$ bit operations.*

Proof: The input to the algorithm presupposes a partial (Γ, K) factorisation, where exactly the coefficients of g indexed by lattice points in $Q|_{\Gamma,K}$ have been specialised [1]. But these are precisely the points given by:

$$Q|_{\Gamma,K} := \{e \in Q \mid 0 \leq l_{\delta'}(e) < k_\delta \text{ where } \delta' \text{ is a summand of some } \delta \in \Gamma\},$$

which establishes correctness. Assuming that no integer overflow occurs as a result of computing $l_{\delta'}(i,j)$, the main loop of the algorithm iterates at most $O(m) = O(t)$ times, during which only multiplication and addition of bits is performed.

The algorithm above still does not produce whether (i,j) corresponds to a nonzero term of g, if at all specialised. However, this can be readily checked by a simple scan of the list of nonzero terms of g that have already been specialised, so that in general, the following scheme would work best:

Algorithm 27 *Specialised*(i,j)
Input: An arbitrary point (i,j) of Q, and a partial (Γ, K)-factorisation extending a coprime dominating edges factorisation.
Output:

- The coefficient of $g_{(i,j)}x^iy^j$ in g, if (i,j) corresponds to a nonzero term of g,
- 0 if (i,j) corresponds to a zero term of g,
- and -1 otherwise.

Step 1: Scan the list $gabs_{j-1}$; if there exists an element with value i, return the value of its coefficient.

Step 2: Invoke Algorithm 25 with input (i,j); if output is PASS, return (0), else return (-1).

It is trivial to see why the above algorithm works correctly and requires an order of $O(t)$ bit operations, the cost for both scanning the list $gabs_{j-1}$ and calling Algorithm 25.

5.2 Counting unspecialised terms

In this section we discuss another frequently used procedure for counting unspecialised terms on translated edges of Q. The approach mirrors in many aspects that of identifying integral points interior to Newt(f), and is described as follows:

Algorithm 28 Input: δ in Γ, δ' its summand in Q, and a partial (Γ, K)-factorisation extending a coprime dominating edges factorisation.
Output: The number of unspecialised coefficients corresponding to integral points on the $k_\delta + 1$ translate of the supporting line of δ' into Q.

Step 1: Retrieve the primitive affine function $l_{\delta'} = \nu_1 x + \nu_2 y + \eta - c_\delta$, and consider the equation of the line ℓ' given by $l_{\delta'} = k_\delta + 1$; set $num \leftarrow 0$.

Step 2: Call $Intersection'(IP_g, \nu_1, \nu_2, \eta - (c_\delta + k_\delta + 1))$.

Step 3: For every integral point (i, j) lying between the two *near* points of intersection produced in Step 2 above, if $Specialised(i, j) = -1$, set $num \leftarrow num + 1$.

Step 4: Return num.

Proposition 29 *Algorithm 28 works correctly as specified and requires at most $O(td)$ bit operations.*

Proof: Given a partial (Γ, K)-factorisation, the algorithm aims to count the number of unspecialised terms of the polynomial $G^{(\delta)}_{k_\delta+1}$, for all $\delta \in \Gamma$. This is done by examining the corresponding number of integral points on the $k_\delta + 1$ translate of δ' into Q, such that δ' is a summand of δ. By Lemma 8 of [1], if $G^{\delta}_{k_\delta+1}$ has a nonzero number of unspecialised terms that is strictly less than the number of integral points on δ', then all of these have exponents which are adjacent integral points on the $k_\delta + 1$ translate of the supporting line of δ' into the polytope, whose equation is defined by $l'_\delta - (k_\delta + 1) = 0$, where $l'_\delta = l_\delta - c_\delta = \nu_1 x + \nu_2 y + \eta - c_\delta$. The above algorithm calls the intersection function $Intersection'$ to determine integral points on this line which are then tested for specialisation.

$Intersection'$ with the given input requires $O(d)$ bit operations, while a call to $Specialised$ requires $O(t)$ bit operations. Since there are about $O(d)$ integral points lying on ℓ' and inside Q, the total amount of time required by the above algorithm is at most $O(td)$ bit operations.

6 On long division with remainder of Laurent polynomials

In this section we discuss in some detail how to perform long division with remainder for Laurent polynomials. The set of all such polynomials forms a commutative ring $R[z, z^{-1}]$, where division with remainder between two Laurent polynomials is possible;

however, this division is not a unique operation [7]. Given two Laurent polynomials, say $a(z)$ and $b(z) \neq 0$, there always exists a Laurent polynomial $q(z)$ and a Laurent polynomial $r(z)$ so that $r(z) = a(z) - b(z)q(z)$ and $\deg(r(z)) < \deg(b(z))$. As such, $r(z)$ consists of $\deg(b(z))$ terms or less (where some of the middle terms can be zero), and hence $b(z)q(z)$ has to match $a(z)$ in at least $\deg(a(z)) - \deg(b(z)) + 1$ terms. However, since the remainder is also a Laurent polynomial, there exists more than one choice for the integer pair (a, b) such that

$$r(z) = \sum_{k=i}^{j} r_k z^k,$$

where $j - i = \deg(r(z))$. As a result, we are free to choose the matching terms of $a(z)$ and $b(z)q(z)$ in the beginning, the end, or divided between the beginning and the end of $a(z)$. For each choice of terms, a corresponding long division algorithm exists.

Since division is not unique, this allows us to transform the modular operations in (3) to that between two regular polynomials. We have seen earlier that since G_0 is an edge polynomial, it is a regular polynomial whose degree is equal to one plus the number of integral points found on its corresponding edge. We can thus require that the Laurent remainder be a strictly regular polynomial of degree less than that of G_0. As a result, and to compute the quantity

$$V(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0, \tag{5}$$

where

$$a(z) = V(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j})$$

is a Laurent polynomial, it suffices to rewrite $a(z) = z^{-m} reg(z)$, where $-m$ is the lowest negative exponent appearing in $a(z)$, and to compute the inverse of z^m modulo G_0, called $inv(z)$ (by construction, we also know that G_0 has a nonzero coefficient term, and hence is relatively prime to z^m, which makes z^m invertible modulo G_0, with $inv(z)$ a regular polynomial). We then have

$$V(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0 \equiv inv(z).reg(z) \bmod G_0,$$

where the right hand side reduces to an ordinary modular operation over $\mathbb{F}$ involving only regular polynomials, and whose remainder, if nonzero, has degree less than $\deg(G_0)$.

On the other hand, equation (4) requires that we compute the quotient of a Laurent polynomial over G_0. Note that in this case

$$[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) - G_{k_\delta} H_0] \bmod G_0$$

should be zero; else, we know that no extension exists for the partial factorisation. The quotient $q(z)$ can thus be found uniquely, by simply solving for $q(z)$ in

$$(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) - G_{k_\delta} H_0 = q(z) G_0.$$

7 Investigating one lifting step

The following section is dedicated to analysing the complexities of a number of major sub-routines used per one step of lifting from a fixed edge in Γ. It is in these tasks that other strong aspects exploiting sparsity will be highlighted.

7.0.1 Sparse univariate polynomial arithmetic over $\mathbb{F}_p$

The most basic of these tasks are related to sparse arithmetic of univariate polynomials over $\mathbb{F}_p$. Assuming that terms of sparse polynomials are stored in increasing order of their exponents, the following algorithm performs the summation of two sparse polynomials in time linear in the input size; in particular, we have:

Algorithm 30 $Sum(f_0, f_2)$
Input: Two polynomials f_0, $f_1 \in \mathbb{F}_p[z]$ in sparse format such that $f_j[i]$ represents the exponent of the i'th term in f_j, $j = 1, 2$. Also given are t_0 and t_1, the total number of terms in f_0 and f_1 respectively. We may also assume that f_0 and f_1 are nonzero.
Output: $f_0 + f_1$ in sparse format.

Step 1: $index \leftarrow 0$, $index_0 \leftarrow 0$, $index_1 \leftarrow 0$;

Step 2: while ($index_0 < t_0$ and $index_1 < t_1$) do
  2.1: If $f_0[index_0] > f_1[index_1]$ and $index_1 < t_1$ do
    · Set $sum[index] \leftarrow f_1[index_1]$.
    · Set $index_1 \leftarrow index_1 + 1$.
    · Set $index \leftarrow index + 1$.
  2.2: Else, if $f_0[index_0] < f_1[index_1]$ and $index_0 < t_0$ do
    · Set $sum[index] \leftarrow f_0[index_0]$.
    · Set $index_0 \leftarrow index_0 + 1$.
    · Set $index \leftarrow index + 1$.
  2.3: Else, if $f_0[index_0] = f_1[index_1]$ do
    · Let c_0 and c_1 denote the coefficients of terms whose exponents are $f_0[index_0]$ and $f_1[index_1]$ respectively.
    · If $c_0 + c_1 \neq 0 \bmod p$, set $sum[index] \leftarrow f_0[index_0]$, and $index \leftarrow index + 1$.
    · If $index_0 < t_0$ set $index_0 \leftarrow index_0 + 1$.
    · If $index_1 < t_1$ set $index_1 \leftarrow index_1 + 1$.

Step 3: If $index_0 < t_0$ set the remaining terms of sum to be the remaining terms of f_0; else, if $index_1 < t_1$, set the remaining terms of sum to be the remaining terms of f_1.

Proposition 31 *The above algorithm works correctly as specified, requiring $O(t)$ bit operations and $O(t)$ bits of memory, for input polynomials in sparse format with at most $O(t)$ terms.*

Proof: The algorithm is based on a comparison procedure whereby terms of f_0 and f_1 are collected together in increasing order of exponents, such that any two terms from f_0 and f_1 respectively and having the same exponent are added modulo p. When any one summand is examined entirely, the remaining terms in the other polynomial will all belong to the sum. The sub-routine above is linear in the number of comparisons it makes between exponents of terms, so that it requires no more than $O(t)$ bit operations for input polynomials with at most $O(t)$ terms, whose exponents do not exceed a machine word. The sum produced has at most $t_0 + t_1$ terms, and hence requires $O(t)$ bits of memory.

Note: The product of two polynomials with $O(t_0)$ and $O(t_1)$ terms respectively can now be achieved using repeated calls, say $O(t_0)$ of them, to the function *Sum*, using input of at most $O(t_1)$ terms. This costs $O(t_0t_1)$ bit operations, and requires $O(t_0t_1)$ bits of memory for storing the final product.

7.1 A complete description of a sparse lifting step

The most computationally extensive part of a single lifting step consists in solving for the known polynomial $e(z)$ such that

$$G_{k_\delta} - V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] = e(z)G_0, \tag{6}$$

where G_s (or H_s), for $s = 1, ..., k_\delta - 1$, are fully specialised univariate polynomials. In the worst case analysis, one would expect to lift at most $O(d)$ times from any edge of $\mathrm{Newt}(f)$ (a number we will denote by max_lift), where this measure is derived from the dimensions of the $d \times d$ square embedding $\mathrm{Newt}(f)$. Consequently, we would require to preallocate a region of memory sufficient to hold about $O(d)$ polynomials per edge, each having degree bounded by $O(max\_int\_pts) = O(d)$. But this amounts to $O(d^2)$ bits of memory, despite the fact that many of these polynomials may turn out to be zero. A crucial modification to the above dense scenario caters not only to the fact that these polynomials would at worst be as sparse as g or h, but that very few of them will be nonzero. Particularly, we have the following:

Lemma 32 *Let $f \in \mathbb{F}_p[x, y]$ be a polynomial of total degree d and at most t nonzero terms. Let r be a vector in $\mathbb{R}^2$ and let Γ be an irredundant dominating set of* $\mathrm{Newt}(f)$ *in direction r. Assume furthermore that $f = gh$ for two non-trivial monomial factors $g, h \in \mathbb{F}_p[x, y]$ with $t^{O(1)}$ terms each. Then, given the decomposition* $\mathrm{Newt}(f) = \mathrm{Newt}(g) + \mathrm{Newt}(h)$ *such that* $\mathrm{Newt}(g)$ *is not a single point or a line segment parallel to $r\mathbb{R}_{\geq 0}$, and for any coprime dominating edges factorisation of f relative to Γ,* $\mathrm{Newt}(g)$ *and* $\mathrm{Newt}(h)$*, there will be at most $t^{O(1)}$ non-constant polynomials g_{k_δ} and h_{k_δ} relative to any $\delta \in \Gamma$, for $k_\delta \geq 0$, and satisfying the Hensel lifting equations in (2).*

Proof: Consider all possible liftings from some edge $\delta \in \Gamma$, and let δ' denote its summand in Newt(g). Since g has $t^{O(1)}$ nonzero terms, Newt(g) will have $t^{O(1)}$ lattice points which correspond to specialised nonzero terms of g. In the worst case analysis, each of these points will fall on a different k_δ translate of the supporting line of δ' into Newt(g). The lifted polynomials in that case, whose terms correspond to lattice points of Newt(g) on these translates, will be nonzero, and there will be at most $t^{O(1)}$ of them. An identical argument applies for the lifted polynomials in Newt(h).

The discussion we shall present below applies for the representation of both G and H polynomials. The data structure we choose treats the distribution of the G_s's as a sparse one, from which information can be derived only about the nonzero lifted polynomials. We define a singly linked ordered list, $\mathcal{G}$, whose elements point only to nonzero polynomials G_s, ordered in increasing order of their translate index s. Another integer array, $Ghead$, of size $O(max\_lift) = O(d)$, is used to provide quick access to the list as follows. If $Ghead[s] = -1$, for some $s = O(max\_lift)$, then G_s is understood to be zero; else, if $Ghead[s] \geq 0$, then G_s is nonzero and occupies position $Ghead[s]$ in $\mathcal{G}$. Furthermore, each polynomial in the list is represented by a sparse array, G_{poly}, whose entries contain the exponents of its nonzero terms only. As before, we make implicit the construction of a similar structure for obtaining the coefficients of terms whose exponents are stored. The cost for updating $\mathcal{G}$ is constant, due to the fact that it's ordered in the same order in which nonzero polynomials appear during the entire lifting stage, so that new elements are appended to the end of the list. The total memory required per edge for this entire scheme is $O(max\_lift) = O(d)$ bits of memory for $Ghead$, $t^{O(1)}$ bits of memory for $\mathcal{G}$, and $t^{O(1)}$ bits for the array G_{poly}. For all edges, this amounts to $O(m(d+t^{O(1)})) = O(t(d+t))$ bits of memory. With this structure, the sub-routine for computing $\sum_{j=1}^{k_\delta-1} G_j H_{k_\delta-j}$ is as follows:

Algorithm 33 $Form\_sum(k_\delta)$
Input: A partial (Γ, K)-factorisation extending a coprime dominating edges factorisation, a fixed edge δ to lift from, and all univariate polynomials G_s and H_s, for $s = 1, ..., k_\delta - 1$, as fully specialised polynomials.
Output: $\sum_{j=1}^{k_\delta-1} G_j H_{k_\delta-j}$ in sparse format.

Step 1: Set $sum \leftarrow 0$; for $j = 1, ..., k_\delta - 1$ do
 If $Ghead[j] \neq -1$ do
  If $Hhead[k_\delta - j] \neq -1$ do
   1.1: Retrieve the polynomials in $\mathcal{G}$ and $\mathcal{H}$ pointed to by $Ghead[j]$ and $Hhead[k_\delta - j]$;
   1.2: Invoke a sparse multiplication sub-routine to form their product $prod$;
   1.3: Invoke a sparse addition sub-routine to form the summation of sum and $prod$;
   1.4: Store the result in sum.

Proposition 34 *Algorithm 33 works correctly as specified, and requires at most $O(t)$ bit* *operations per any lifting step and $O(t)$ bits of temporary storage.*

Proof: Correctness of the use of the data structure is an immediate consequence of the discussion above. To establish the run time cost, we know that the main loop iterates at most $O(max\_lift) = O(d)$ times. However, since there are at most $t^{O(1)}$ nonzero polynomials G_j or H_j, for $j = 1, ..., k_\delta - 1$, we know that in many cases the procedure will never perform the inner-most arithmetic polynomial computations. Hence, we need to redefine what a worst case scenario will be. By Lemma 32, there will be a collection $\mathcal{G}'$ of at most $O(t)$ polynomial pairs $(G_j, H_{k_\delta - j})$ per any lifting step such that both polynomials are nonzero. In the worst case analysis, there will be one pair $(G_{j'}, H_{k_\delta - j'})$ in $\mathcal{G}'$ with $t^{O(1)}$ terms per polynomial. The product of one such pair requires $O(t)$ bit operations. The remaining pairs in $\mathcal{G}' - (G_{j'}, H_{k_\delta - j'})$ will have $O(1)$ terms per polynomial, so that the product of one such pair requires $O(1)$ bit operations, and the sum of products of pairs in $\mathcal{G}' - (G_{j'}, H_{k_\delta - j'})$ requires $O(t)$ bit operations. Adding this sum to $G_{j'} H_{k_\delta - j'}$ is dominated by $O(t)$ bit operations, and produces a polynomial of at most $O(t)$ terms.

Unlike the lifted polynomials G_s and H_s, the polynomial F_{k_δ} is used only once in a particular calculation and so need not be stored for any edge. Hence, we represent this polynomial temporarily in a sparse data structure that can be reused by any edge from which one is lifting. The procedure for determining F_{k_δ} is as follows:

Algorithm 35 Input: A partial (Γ, K)-factorisation extending a coprime dominating edges factorisation, and a fixed edge $\delta \in \Gamma$ to lift from.
Output: The univariate polynomial F_{k_δ} stored in sparse format in array F_{poly}.

Step 1: Set $index \leftarrow 0$; retrieve the primitive affine function $l_\delta = \nu_1 x + \nu_2 y + \eta$ and consider the equation of the line $\ell' : l_\delta = k_\delta$.

Step 2: Call $Intersection'(IP, \nu_1, \nu_2, \eta - k_\delta)$.

Step 3: For every integral point (i, j) lying between the two *near* points produced in Step 2 above do:

3.1: Scan the list $fabs_{j-1}$; if there exists an element i in this list, do

. Let (a, b) denote the coordinates of the starting vertex of edge δ. Compute

$$exponent = (j - (b + (\zeta_2.k_\delta)))/ - \nu_1, \quad \text{if } \nu_1 \neq 0$$

or

$$exponent = (i - (a + (\zeta_1.k_\delta)))/\nu_2, \quad \text{if } \nu_2 \neq 0.$$

. Set $F_{poly}[index] \leftarrow exponent$ and $index \leftarrow index + 1$.

Step 4: Rearrange the entries in F_{poly} in increasing order of exponents and return F_{poly}.

Proposition 36 *Algorithm 35 works correctly and requires at most $O(td)$ bit operations.*

Proof: The above algorithm determines all possible integral points found along the k_δ translate of the supporting line of δ into $\mathrm{Newt}(f)$, using $O(d)$ bit operations. The maximum number of such integral points is $O(d)$, and for every integral point, we scan the lists of terms of f in no more than $O(t)$ bit operations to see if the point corresponds to some nonzero term of f. If such a term $f_{(i,j)}x^iy^j$ is found, a corresponding z term is formed using integer addition, multiplication, and division, all of whose input does not exceed a machine word size. Finally, the terms of the produced polynomial are rearranged in increasing order of exponents, to conform to the representation required by sparse polynomial arithmetic sub-routines. In total, the cost of the above algorithm is at most $O(td)$ bit operations.

Now that we have computed F_{k_δ}, we can form

$$F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j} \tag{7}$$

using sparse addition over $\mathbb{F}_p$, and

$$(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0 \tag{8}$$

using division with remainder for Laurent polynomials (see Section 6). Note that the number of terms in G_0 is bounded by the number of terms in g, and its degree is bounded by $O(max\_int\_pts) = O(d)$. Also, the dividend in (8) has at most $O(t)$ terms and has degree at most $O(d)$. Despite that both dividend and divisor are sparse, the intermediary remainders may not be necessarily so. The above will then require $O(d^2)$ operations over $\mathbb{F}_p$, producing a remainder with degree at most $O(d)$ and that has up to $O(d)$ terms. Finally, we can compute the product

$$V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] \tag{9}$$

using $O(d^2)$ bit operations, since $\deg(V) < \deg(G_0) = O(d)$. From the discussion on Laurent polynomial division with remainder in Section 6, the result is a regular polynomial of degree $O(d)$ and is stored temporarily in sparse format using an integer array, say *temp*, of size $O(d)$.

7.2 Representing unknown polynomials and expressions

All the computations so far have involved fully specialised polynomials, which led us to exploit commonly known data structures in their representation. We now discuss the more complex symbolic representation of polynomials with unknown coefficients and systems of equations involving several unknowns. The first such example is in representing the polynomial G_{k_δ}, whose coefficients are not all known at the time we start performing a partial (Γ, K)-factorisation. Although we expect G_{k_δ} to be a sparse polynomial once its coefficients are fully specialised, the fact that some of its coefficients are still unknown forces us to represent it temporarily using a dense array, $temp_g$, of which one copy can be used by any edge during any lifting stage. The array will have size bounded by $max\_int\_pts = O(d)$, and the entries of the array will contain the exponents in z of every possible term (whether zero, nonzero, or simply unknown) in G_{k_δ}. In a standard sparse polynomial structure, one can initialise the entries of the corresponding array to some negative number, assuming that the polynomial is regular and hence cannot have negative exponents. However, since G_{k_δ} is a Laurent polynomial whose terms can have negative exponents, it becomes essential to keep track of the maximum possible number of terms in G_{k_δ}, in order to avoid accessing unwanted entries in $temp_g$ that may contain information from previous computations. If lb_G and ub_G denote the respective lowest and highest exponents among terms of G_{k_δ} that are either 1 or unspecialised, and assuming that entries in $temp_g$ are stored in increasing order of exponents, we define the *possible* degree of G_{k_δ} to be the difference $ub_G - lb_G$. G_{k_δ} can then be represented using at most $O(td)$ bit operations as follows:

Algorithm 37 Input: A partial (Γ, K)-factorisation extending a coprime dominating edges factorisation, and a fixed edge $\delta \in \Gamma$ to lift from.
Output: The polynomial G_{k_δ} stored in sparse format in array $temp_g$.

Step 1: Set $index \leftarrow 0$; retrieve the primitive affine function $l_{\delta'} = \nu_1 x + \nu_2 y + \eta - c_\delta$, and consider the equation of the line $\ell : l_{\delta'} = k_\delta$.

Step 2: Call $Intersection'(IP_g, \nu_1, \nu_2, \eta - (c_\delta + k_\delta))$.

Step 3: For every integral point (i, j) lying between the two *near* points produced in Step 2 above do

3.1: Call $Specialised(i, j)$; if this returns 1 or -1, do
3.1.1: Let (a, b) denote the coordinates of the starting vertex of edge δ'. Compute

$$exponent = (j - (b + (\zeta_2.k_\delta)))/-\nu_1, \quad \text{if } \nu_1 \neq 0$$

or

$$exponent = (i - (a + (\zeta_1.k_\delta)))/\nu_2, \quad \text{if } \nu_2 \neq 0.$$

Set $temp_g[index] \leftarrow exponent$, and $index \leftarrow index + 1$.

Step 4: Rearrange the first *index* entries of $temp_g$ in increasing order of exponents, and return array $temp_g$.

7.2.1 Data structure for expressions using unknown polynomials

Another example of an unknown symbolic entity is the expression

$$G_{k_\delta} - V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] \tag{10}$$

where as we have seen above, G_{k_δ} is a polynomial whose coefficients are partially specialised, and the second summand is a fully known polynomial. Let this quantity be denoted by LHS, representing the left hand side of the main lifting equation (6). Because we need a symbolic structure matching the nature of LHS before the unknown coefficients are specialised, this has to deal with its two separate summands, the first of which is treated as a dense polynomial. Suppose we choose to use an array lhs: Two issues to resolve are the size and nature of lhs. We have seen that G_{k_δ} and the polynomial in (9) both have degree at most $O(d)$, so that in total LHS will have at most $O(d)$ terms. We store these in sparse format and only temporarily. Furthermore, the expression in (9) is a regular polynomial, but since G_{k_δ} can be a Laurent polynomial, LHS inherits the same structure. Let lb_G and ub_G be as defined above, and lb and ub denote the smallest and largest exponents of terms appearing in (9), so that $ub - lb = O(d)$. Then lhs should have entries whose exponents range from $\min(lb_G, lb)$ to $\max(ub_G, ub)$, which we shall denote by low_{lhs} and $high_{lhs}$ respectively. As such, $high_{lhs} - low_{lhs}$ represents the highest possible degree that LHS can attain after being fully specialised.

Because we have to use LHS in a process which involves comparing coefficients of terms on both sides of equation (6), it will be more convenient to store LHS in dense format, whereby information about the coefficients of terms rather than their exponents is revealed. Accordingly, the entries of lhs should point to the coefficients of the polynomial expression in (10). Since lhs represents a Laurent polynomial whose terms can have negative exponents, we have to discuss not only the dense structure of coefficients but also their address in the memory locations, which are always labelled by indices starting from zero. In particular, since the the lowest term of LHS has exponent low_{lhs}, and as its coefficient has to be stored in the zero location of the array lhs, all remaining coefficients c_i of the left hand side, for $i = low_{lhs} + 1, ..., high_{lhs}$, have to be stored in locations $i - low_{lhs}$. Now, since the coefficients can inherit two pieces of input, one from G_{k_δ}, representing an unknown, and one from (9), which is fully specialised, we allow each coefficient to reflect this structure, by associating with the $i - low_{lhs}$ entry of lhs two integers: the first containing the coefficient of z^i in G_{k_δ}, and the second containing the coefficient of z^i in (9). In total, this requires that we treat lhs as a double array of size $2 \times O(d)$. We now have the following:

Algorithm 38 Input: A partial (Γ, K)-factorisation extending a coprime dominating edges factorisation, and a fixed edge $\delta \in \Gamma$ to lift from. Also given is the representation of G_{k_δ} in array $temp_g$ with ind_g entries, the representation of (9) in $temp$ with ind entries, and low_{lhs} and $high_{lhs}$ designating the lowest and highest exponents in the unknown expression:

$$G_{k_\delta} - V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0].$$

Output: The dense representation of the expression LHS in the array lhs.

Step 1: Initialise array lhs to zero.

Step 2: For $i = 0, ..., ind_g - 1$ do
2.1: Set $e \leftarrow temp_g[i]$, and use the change of basis in Step 4 of Algorithm 2 to determine
the integers a and b such that $g_{(a,b)}x^a y^b$ is the bivariate term in g_{k_δ} corresponding to z^e
in G_{k_δ}.
2.2: Set $lhs_{e-low_{lhs}}[0] \leftarrow Specialised(a, b)$.

Step 3: For $i = 0, ..., ind - 1$, set $e \leftarrow temp[i]$ and $lhs_{e-low_{lhs}}[1]$ to be the coefficient of z^e in (9).

The above procedure is obviously correct, as it reads from locations of the sparse arrays $temp$ and $temp_g$ and writes to proper data locations of the dense array lhs. Data found in $temp_g$ represents powers of univariate terms of G_{k_δ} that are then transformed into equivalent powers (a, b) of bivariate terms, whose coefficients are determined by a call to $Specialised(a, b)$. Data in the array $temp$ however is simply translated into lhs at the proper locations, as it represents exponents of terms that are known to be nonzero. Since the number of terms in lhs is bounded by $O(d)$, and since each call to $Specialised$ requires $O(t)$ bit operations, the entire process requires $O(td)$ bit operations and $O(d)$ bits of temporary storage.

We now need to discuss how to make use of all the above representations to solve for the unknown polynomial $e(z)$ in (6). The trivial case when $\deg(G_0)$ is greater than the highest possible degree of the left hand-side results in $e(z)$ being the zero polynomial, so that LHS itself is zero. The unknown coefficients of z^i in G_{k_δ} can then be specialised as follows:

Algorithm 39 Input: A partial (Γ, K)-factorisation extending a coprime dominating edges factorisation, and a fixed edge $\delta \in \Gamma$ to lift from. Also given is LHS as a fully specialised polynomial in the form $\sum_{j=low_{lhs},...,high_{lhs}} c_j z^j$, for some known values $c_j \in \mathbb{F}_p$.
Output: G_{k_δ} as a fully specialised polynomial, and the specialisation of the corresponding coefficients of the polynomial g.

Step 1: For $i = lb_G, ..., ub_G$, if $lhs_{i-low_{lhs}}[0] = -1$ do
1.1: Set $lhs_{i-low_{lhs}}[0] \leftarrow (lhs_{i-low_{lhs}}[1] + c_i) \bmod p$ and $temp_g[i - lb_G] \leftarrow lhs_{i-low_{lhs}}[0]$.
1.2: Use the change of basis in Step 4 of Algorithm 2 to determine the integers a and b

such that $g_{(a,b)}x^a y^b$ is the bivariate term in g_{k_δ} corresponding to z^i in G_{k_δ}.

1.3: If $Specialised(a,b) = -1$

· If $lhs_{i-low_{lhs}}[0] \neq 0$, add $g_{(a,b)}x^a y^b$ to the list of nonzero terms of g, and reduce rem_g by 1.

1.4: Else if $Specialised(a,b)$ is not equal to the coefficient of the term in g whose exponent is $temp_g[i - lb_G]$, output "failure" for this choice of coprime dominating edges factorisation.

Step 2: For $i = lb_G, ..., ub_G$, form the polynomial in z whose nonzero terms have exponents stored in $temp_g$; if this polynomial is nonzero, add it to the end of the list $\mathcal{G}$, store it permanently in sparse format in the array G_{poly}, and let $Ghead[k_\delta]$ point to its location in the list.

Proposition 40 *Algorithm 39 works correctly as specified and requires $O(td)$ bit operations.*

Proof: The above procedure identifies those entries in LHS whose partial summands are terms in G_{k_δ}. For all unspecialised terms $g_i z^i$ of G_{k_δ}, the corresponding coefficients are determined such that the coefficient of z^i in LHS is zero. This specialisation of terms in G_{k_δ} leads to a specialisation of g coefficients, among which only the nonzero elements are added to the sparse representation of g. A check is made so that g coefficients match previously known values, if those exist. Finally, if the polynomial G_{k_δ} is nonzero, the polynomial (in fact, its address in memory) is appended to the end of $Glist$, with $Ghead[k_\delta]$ pointing to its position in the list. Obviously, the above process requires $O(td)$ bit operations, since only $O(d)$ terms of G_{k_δ} require a call to *Specialised*.

If $\deg(G_0)$ is greater than $high_{lhs} - low_{lhs}$, we know from the discussion in Lemma 9 of [1] that each triangular system arising from comparing coefficients on both sides of $LHS = e(z)G_0$ can be solved uniquely. In the following, we shall only describe how to set up the lower triangular system, since the upper system can be treated similarly. Recall that a lower triangular system can be set up by comparing known coefficients of lower terms in LHS to corresponding ones in $e(z)G_0$. We now claim the following:

Lemma 41 *The triangular systems resulting from equating coefficients of polynomials on both sides of $LHS = e(z)G_0$ are sparse linear systems with at most $O(t^{O(1)}d)$ nonzero elements over $\mathbb{F}_p$.*

Proof: Write

$$G_0(z) = \sum_{j=0,\dots,\deg(G_0)} g_j z^j,$$

where at most $t^{O(1)}$ number of the g_j's are nonzero over $\mathbb{F}_p$, and write

$$LHS = \sum_{j=low_{lhs},...,high_{lhs}} c_j z^j,$$

where not all of the c_j's are specialised. We know that $e(z)$ is a Laurent polynomial satisfying $LHS = e(z)G_0$. Let low_e and $high_e$ denote the respective lowest and highest exponents of terms of $e(z)$. Then, since G_0 is a regular polynomial with a nonzero constant term over $\mathbb{F}_p$, we have

$$low_e = low_{lhs} \quad \text{and } high_e = high_{lhs} - \deg(G_0).$$

Write $e(z) = \sum_{i=low_e,...,high_e} e_i z^i$. Using our data structure for LHS above, there exist two integers, say low_range and $high\_range$, such that the known lower and higher terms of LHS have exponents lying in the two intervals $[low_{lhs}, low\_range]$ and $[high\_range, high_{lhs}]$ respectively. Suppose that we wish to solve for the unknown coefficients of $e(z)$ using the lower known terms of LHS. Let B be the matrix of coefficients of terms $c_i z^i$ in LHS, for $i = low_{lhs}, ..., low\_range$, such that each c_i occupies row $i - low_e$ in the column vector B. Let A be the matrix of coefficients in $e(z)G_0$ corresponding to coefficients c_i, for $i = low_{lhs}, ..., low\_range$. We then have

$$c_i = \sum_{j=low_e,...,high_e} e_j g_{i-j},$$

so that row $i - low_e$ of A contains $e_j g_{i-j}$ in column $j - low_e$, for $j = low_e, ..., high_e$. Since at most $t^{O(1)}$ coefficients of G_0 are nonzero, it follows that each row of A contains at most $t^{O(1)}$ nonzero entries over $\mathbb{F}_p$. Since the number of specialised lower terms in LHS is bounded by $O(d)$, the system

$$Ax = B$$

contains at most $O(d)$ columns and so $O(t^{O(1)}d)$ nonzero entries in total.

Assuming the entries of any of the triangular systems belong to a finite field with prime order which fits in a machine word, one can now obtain a solution using $O(t^{O(1)}t^{O(1)}d) = O(t^2 d)$ bit operations with no more than $O(t^{O(1)}d)$ bits of temporary storage memory using any of the well known sparse direct methods (see for instance [8] on a broad survey of data structures and algorithms for sparse Gaussian elimination). When one or two of the triangular systems have been solved uniquely, and assuming the results of the two triangular systems have been consistent, one can then immediately retrieve $e(z)$. Note that Algorithm 39 above can be applied in the general case when LHS is a fully specialised, not necessarily zero polynomial, and hence can be invoked to determine the polynomial G_{k_δ} and the corresponding g coefficients when $e(z)$ is not zero.

7.3 Recovering H_{k_δ}

Invoking Algorithm 37 using the polynomial H_{k_δ} and the summand Q, we can set up a representation of H_{k_δ} using a temporary array $temp_h$ and solve for the unknown coefficients of H_{k_δ} using the equation

$$H_{k_\delta} = \frac{(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) - G_{k_\delta} H_0}{G_0}. \tag{11}$$

Note that the only new computations are for determining the product $G_j H_{k_\delta - j}$ if G_{k_δ} is nonzero, and finding the quotient over G_0, with both divisor and dividend having at most $O(t)$ and $t^{O(1)}$ terms respectively. The intermediary remainders have degree at most $O(d)$, and hence at most $O(d)$ terms. When the numerator is non-trivial, this will require at most $O(d^2)$ bit operations and $O(d)$ bits of temporary storage. The unknown coefficients of $temp_h$ are matched with their corresponding coefficients in the quotient, and the corresponding coefficients of h are specialised through a process similar to Step 2 of Algorithm 39, using at most $O(td)$ bit operations.

8 Total run time and memory

We are now ready to establish the total complexity of the sparse adaptation, combining the above subcosts across all possible liftings per one coprime dominating edges factorisation. In particular, we shall distinguish between two categories of sub-tasks, those that will be carried out during every possible lifting step, and those which will be performed a number of times that is dependent on the sparsity factor t of f. We have the following concluding result:

Theorem 42 *Let $f \in \mathbb{F}_p[x, y]$ be a polynomial of total degree d and at most t nonzero terms such that $t < d$. Let r be a vector in $\mathbb{R}^2$ and let Γ be an irredundant dominating set of* $\mathrm{Newt}(f)$ *in direction r. Assume furthermore that $f = gh$ for two non-trivial monomial factors $g, h \in \mathbb{F}_p[x, y]$ with $t^{O(1)}$ terms each. Then, there exists an integral decomposition* $\mathrm{Newt}(f) = \mathrm{Newt}(g) + \mathrm{Newt}(h)$ *such that* $\mathrm{Newt}(g)$ *is not a single point or a line segment parallel to $r\mathbb{R}_{\geq 0}$. Furthermore, for any coprime dominating edges factorisation of f relative to Γ,* $\mathrm{Newt}(g)$ *and* $\mathrm{Newt}(h)$*, there exists one full factorisation of f which extends it in $O(td^2 + t^3 d)$ bit operations and $O(t^{O(1)} d)$ bits of memory, assuming that d and p fit in a machine word.*

Proof: That there exists an integral decomposition of $Newt(f)$ into two Newton polytopes corresponding to g and h, and that the algorithm can recover the two factors using any coprime dominating edges factorisation is a result of Ostrowski's theorem and Theorem 1 of [1]. We now establish the total running time and memory required by the sparse method. In the following, δ denotes an edge in Γ from which lifting can take place, and δ' denotes its summand in $\mathrm{Newt}(g)$. During a single lifting step, one first has to determine the fully specialised polynomial F_{k_δ} and construct the unknown polynomials G_{k_δ} and H_{k_δ}, all using at most $O(td)$ bit operations and $O(d)$ temporary bits of temporary storage. Hence, the total cost of representing the lifted polynomials is $max\_lift.O(td) = O(td^2)$ bit operations and a temporary $O(d)$ bits of memory. We have also shown that computing the quantity in (7) requires in the worst case analysis $O(t)$ bit operations and $O(t)$ bits of temporary storage, so that the total cost is $max\_lift.O(t) = O(td)$ bit operations and $O(t)$ bits of memory. Computing the polynomial in (9) may in the worst case require $O(d^2)$ bit operations and $O(d)$ bits of temporary storage per any lifting step. However, we claim that this need not be done during every lifting step. In particular, and since the modular operation is non-trivial only when the polynomial (7) is nonzero, it suffices to determine the maximum number of times that this can happen in order to obtain the total cost of long division throughout the lifting stage. Note that the polynomial (7) is nonzero in at most one of these cases:

- either F_{k_δ} is nonzero,
- or $\sum_{j=1,\ldots,k_\delta-1} G_j H_{k_\delta-j}$ is nonzero.

However, there are at most $O(t)$ nonzero polynomials F_{k_δ} for all $k_\delta \leq max\_lift$, since at most t lattice points in $\mathrm{Newt}(f)$ correspond to nonzero terms of f. By Lemma 32, there exist at most $t^{O(1)}$ nonzero polynomials G_j and H_i, for $i, j = 1, \ldots, k_\delta - 1$, where $k_\delta \leq max\_lift$, and so there will be at most $O(t)$ nonzero polynomial expressions of the form $G_j H_i$. In the worst case analysis, no two such products $G_j H_i$ and $G_{j'} H_{i'}$ will be such that $j + i = j' + i' = k_\delta$, so that there will be at most $O(t)$ nonzero sums of the form $\sum_{j=1,\ldots,k_\delta-1} G_j H_{k_\delta-j}$, for all $k_\delta \leq max\_lift$. This implies that the polynomial (8) is nonzero in at most $O(t)$ of the total number of lifting steps, which brings the total cost of computing its remainder modulo G_0 to $O(td^2)$ bit operations, and $O(d)$ bits of temporary storage.

The quantity in (9) has been shown to require at most $O(d^2)$ bit operations and $O(d)$ bits of temporary storage. But as seen above, this should only be performed when the polynomial in (8) is nonzero. In the worst case analysis, this in turn is nonzero whenever the polynomial (7) is nonzero, which has been seen to happen in at most $O(t)$ of the total number of lifting steps. Hence, computing the polynomial in (9) requires at most $O(td^2)$ bit operations and $O(d)$ bits of memory in total.

The sparse triangular system(s) for solving for the unknown coefficients of $e(z)$ in $LHS = e(z)G_0$ have been shown to require at most $O(t^2 d)$ bit operations and $O(t^{O(1)}d)$ bits of temporary storage. However, we now claim that one does not require to set up and solve a triangular system when

- the polynomial in (9) is zero, and
- G_{k_δ} has no specialised terms.

To see this, let u_δ and gn_δ denote respectively the number of unspecialised terms on the $k_\delta+1$ translate of the supporting line of δ' into $\mathrm{Newt}(g)$, and the number of integral points on δ' of $\mathrm{Newt}(g)$. We know that if G_{k_δ} has no specialised terms, the possible degree of G_{k_δ} is given by $u_\delta - 1$, which is less than $\deg(G_0) = gn_\delta - 1$, because of the inequality $u_\delta < gn_\delta$. This, combined with the fact that (9) is zero, results in the degree of LHS

being less than $\deg(G_0)$, from which one concludes that $e(z)$ is zero. Consequently, one has to set up a triangular system in at most one of the two following cases:

- the polynomial in (9) is nonzero, or
- G_{k_δ} has at least one specialised term.

Since the first condition can happen in at most $O(t)$ of the cases, and the second can happen in at most $O(t^{O(1)})$ of the cases, the total cost for solving any of the triangular systems is $O(t^3 d)$ bit operations, and $O(t^{O(1)}d)$ bits of temporary storage, throughout the entire lifting stage.

Determining H_{k_δ} per one lifting step has been seen to require $O(d^2)$ bit operations and $O(d)$ bits of temporary storage. Similarly as above, the long division to be performed in (11) is non-trivial only when the numerator is nonzero. This, in turn, happens in at most one of the two cases:

- The polynomial in (7) is nonzero, or
- $G_{k_\delta} H_0$ is nonzero.

This can be easily seen to happen in at most $O(t)$ of the cases, which brings the total cost of determining an expression of H_{k_δ} to $O(td^2)$ bit operations and $O(d)$ bits of temporary storage.

When fully specialised, only the nonzero polynomials among all G_{k_δ}'s and H_{k_δ}'s ought to be stored in sparse form. Specialising the coefficients of these polynomials during one lifting step and using Algorithm 39 requires at most $O(td)$ bit operations, so that in total this will be at most $max\_lift.O(td) = O(td^2)$ bit operations. Since the total number of terms of all such polynomials should not exceed $t^{O(1)}$, the total amount of memory for storing the lifted polynomials is of the order $O(t^2) = O(td)$, for $t < d$.

Combining all of the above, any coprime dominating edges factorisation can be extended using at most $O(td^2 + t^3 d)$ bit operations and $O(t^{O(1)}d)$ bits of memory.

The above result helps justify the earlier conditions we imposed on t as follows: Since the lifted polynomials are bounded in degree by $O(d)$, and since $\mathrm{Newt}(f) = O(d^2)$, the standard polytope method requires $O(max\_lift.d^3) = O(d^4)$ bit operations in total and $O(d^2)$ bits of memory. When

$$td + t^3 < d^3,$$

we certainly have that $t < d$, and by Theorem 42, the sparse adaptation outperforms the dense one in both the operational and spatial complexity.

9 Computational results

The work was carried out at the Oxford University Supercomputing Centre (OSC) on the Oswell machine, using an UltraSPARC III processor running at about 122 Mflop/sec and with 2 GBytes of memory. All experiments were carried out over $\mathbb{F}_2$. The input polynomials have been constructed as explained in Section 3 above. For each of the random polynomials g and h the exponent vectors (e_1, e_2) were chosen uniformly at random such that $0 \leq e_1 + e_2 \leq d/2$, and at least three of them are of the form $(e_1, 0)$, $(0, e_2)$ and $(e_3, (d/2) - e_3)$, so that f was of degree d and had no monomial factors. The table below gives the running times (in seconds) of the total factorisation process to find at least one non-trivial factor f. In the following, t denotes the number of terms of the input polynomial f, $T.Sum.$ denotes the total number of non-trivial integral decompositions $\mathrm{Newt}(f) = Q + R$, s_s denotes the run time in seconds of the sparse method, corresponding only to the successful liftings which produce at least one factor of f, and s_d is the corresponding run time in seconds of the dense method wherever applicable (as allowed by the machine's memory resources). Also, $T.Bd.F.$ denotes the total number of coprime edges factorisations associated with all possible summands and irredundant set of dominating edges of $\mathrm{Newt}(f)$, whereas $A.Bd.F.$ denotes the number of coprime edges factorisations attempted before a successful extension produces the two factors g and h. Finally, $\#N_f$, $\#N_g$, and $\#N_h$ denote the number of lattice points in the Newton polytopes of f, g and h respectively.

The run times in Table 1 indicate that the sparse algorithm is faster than the dense one for input polynomials which can be handled by both methods. Obviously, the run times also increase for increasing input degrees. For larger degree polynomials where the dense algorithm no more applies, we monitor the variations in running times by fixing all parameters apart from the number of terms of the input. For this, we construct families of random polynomials having the same Newton polytope as well as the same boundary factorisations along a fixed dominating set of edges. Different polynomials with varying number of terms can then be chosen by randomly selecting the appropriate number of lattice points in the interior of the polytope. As predicted earlier, The run times in Table 2 increase upon incrementing either the degrees or the terms of the input polynomials. Note that in almost all cases $\mathrm{Newt}(f)$ has extremely few non-trivial integral decompositions, as predicted earlier in [1] for sparse polynomials. Although the number of all possible coprime edges factorisations is not small in all cases, it is still significantly smaller than the input degree of the polynomial, and hence the size of $\mathrm{Newt}(f)$. The largest degree factorisation $f = gh$ attained is given by

$$f = x^{4120} + x^{4118}y^2 + x^{3708}y^{400} + x^{3706}y^{402} + x^{2781}y^{1300} + x^{2779}y^{1302} + x^{1339}y^{2700}$$

$$+x^{927}y^{3100} + y^{4000} + x^{7172}y^{4167} + x^{8349}y^{4432} + x^{8347}y^{4434} + x^{6760}y^{4567} + x^{5833}y^{5467}$$

$$+x^{5568}y^{7132} + x^{11401}y^{8599},$$

$$g = x^{2781} + x^{2779}y^2 + x^{5833}y^{4167} + y^{2700},$$

and

$$h = x^{1339} + x^{927}y^{400} + x^{5568}y^{4432} + y^{1300}.$$

Table 1: Smalll degree polynomials

d	t	s_s	s_d	$T.Sum.$	$T.Bd.F.$	$A.Bd.F.$	$\#N_f$	$\#N_g$	$\#N_h$
50	14	4	3	1	8	2	561	166	50
100	16	8	12	2	15	0	2234	472	222
500	15	13	22	1	25	17	52940	12758	11282
2000	28	540	620	1	21	9	848849	133797	132932

Table 2: Large degree polynomials

d	t	s_s	$T.Sum.$	$T.Bd.F.$	$A.Bd.F.$	$\#N_f$	$\#N_g$	$\#N_h$
6000	36	$23'50''$	3	36	16	8496181	502330	2615634
6000	100	$28'28''$	...	...	...	...	...	...
6000	196	$59'51'$	...	...	...	...	...	...
10000	12	$22'57''$	1	15	7	15521707	2417337	3063179
10000	60	$38'36''$	...	...	...	...	...	...
20000	16	$48hr\ 4'53''$	1	42	18	39374376	5716256	9914429

10 Conclusion

It has been previously shown that, despite the fact that a randomly chosen bivariate polynomial over a finite field is unlikely to be reducible, there is still a significant number of bivariate polynomials that are reducible [14], which justifies continuing efforts in developing efficient factorisation algorithms. Of particular interest in real life applications are sparse polynomials, for which no well defined 'sparse' factorisation algorithm has still been devised. In this chapter we have attempted to address the open question of finding such an algorithm by investigating potentially strong areas of the polytope method in application to sparse bivariate polynomials over $\mathbb{F}_p$. In addition, we have been able to address another significant aspect in which the algorithm can be adapted so that the run time of the lifting stage is made dependent on the number of terms belonging to the input polynomial, rather than its degree only. Assuming an upper bound on the sparsity of the possible factors of the input polynomial, the gains for sparse polynomials that are a product of sparse factors are demonstrated not only through the improved run time of the algorithm during its lifting stage, but also in the reduced memory requirements, so that the sparse adaptation requires $O(td^2 + t^3d)$ bit operations and $O(t^{O(1)}d)$ bits of memory, compared to the corresponding dense costs of $O(d^4)$ and $O(d^2)$.

In addition to the above, this chapter has covered complete details of the implementation we have carried out, where problems related to computing with geometric structures and maintaining correct exact arithmetic have been highlighted. The combination of our sparse adaptation has led to a very fast and high record in sparse binary bivariate factorisation of degree 20000, which we believe has not been previously achieved using any other different algorithm. We expect our adaptation of the polytope method to perform equally well for sparse and high degree bivariate polynomials over fields of other prime orders.

References

[1] F. Abu Salem, S. Gao, and A.G.B. Lauder, "Factoring polynomials via polytopes", to appear in *Proc. ISSAC 2004*, ACM Press.

[2] F. Abu Salem, S. Gao, and A.G.B. Lauder, "Factoring polynomials via polytopes: Extended version", Report PRG-RR-04-07, Oxford University Computing Laboratory, April 2004.

[3] F. Abu Salem, "An efficient sparse adaptation of the polytope method over $\mathbb{F}_p$ and a record-high binary bivariate factorisation", Report PRG-RR-04-13, Oxford University Computing Laboratory, June 2004, submitted to *Journal of Symbolic Computation.*

[4] E. R. Berlekamp, "Factoring polynomials over finite fields", *The Bell System Technical Journal*, 1967, vol. 46, pp. 1853-1859.

[5] Laurent Bernardin, "Factoring Multivariate Polynomials over a Finite Field", Diss. ETH (PhD thesis), 1999.

[6] A. L. Chistov, "An algorithm of polynomial complexity for factoring polynomials, and determination of the components of a variety in a subexponential time" (Russian), *Theory of the complexity of computations, II.*, *Zap. Nauchn. Sem. Leningrad. Otdel. Mat. Inst. Steklov. (LOMI)* **137** (1984), 124–188. [English translation: *J. Sov. Math.* **34** (1986).]

[7] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps", Technical report, Bell Laboratories, Lucent Technologies, May 2000.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid, "Direct Methods for Sparse Matrices", Oxford University Press, 1986.

[9] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin Heidelberg, 1987.

[10] S. Gao, "Absolute irreducibility of polynomials via Newton polytopes," *Journal of Algebra* **237** (2001), 501–520.

[11] S. Gao, "Factoring multivariate polynomials via partial differential equations," *Mathematics of Computation* **72** (2003), 801–822.

[12] J. von zur Gathen and E. Kaltofen, "Factoring sparse multivariate polynomials", *Journal of Computer and System Sciences* **31** (1985), 265–287.

[13] S. Gao and A.G.B. Lauder, "Decomposition of polytopes and polynomials", *Discrete and Computational Geometry* **26** (2001), 89–104.

[14] S. Gao and A.G.B. Lauder, "Hensel lifting and bivariate polynomial factorisation over finite fields", *Mathematics of Computation* **71** (2002), 1663–1676.

[15] Granlund, T., *The GNU Multiple Precision Arithmetic Library*, The Free Software Foundation, Boston, 2000.

[16] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters* **1** (1972), 132–3.

[17] D. Yu Grigoryev, "Factoring polynomials over a finite field and solution of systems of algebraic equations" (Russian), *Theory of the complexity of computations, II.*, *Zap. Nauchn. Sem. Leningrad. Otdel. Mat. Inst. Steklov. (LOMI)* **137** (1984), 124–188. [English translation: *J. Sov. Math.* **34** (1986).]

[18] E. Kaltofen, "Polynomial-time reductions from multivariate to bi- and univariate integral polynomial factorisation", *SIAM Journal on Computing* **14** (1985), 469–489.

[19] J. Knopfmacher and A. Knopfmacher, "Counting irreducible factors of polynomials over a finite field", *Discrete Mathematics* **112** (1993), 103-118.

[20] E. Kaltofen and B. Trager, "Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators", *Journal of Symbolic Computation* **9** (1990), 301–320.

[21] A. K. Lenstra, "Factoring multivariate integral polynomials", *Theoretical Computer Science* **34** (1984), no. 1-2, 207–213.

[22] A. K. Lenstra, "Factoring multivariate polynomials over finite fields", *Journal of Computer and System Sciences* **30** (1985), no. 2, 235–248.

[23] A. K. Lenstra, "Factoring multivariate polynomials over algebraic number fields", *SIAM Journal on Computing* **16** (1987), no. 3, 591–598.

[24] D.R. Musser, "Multivariate polynomial factorization", *Journal of the ACM* **22** (1975), 291–308.

[25] J. O'Rourke, *Computational Geometry in C*, second edition, Cambridge University Press (1998).

[26] P. S. Wang, "An improved multivariate polynomial factorization algorithm", *Mathematics of Computation* **32** (1978), 1215–1231.

[27] P. S. Wang and L. P. Rothschild, "Factoring multivariate polynomials over the integers," *Mathematics of Computation* **29** (1975), 935–950.