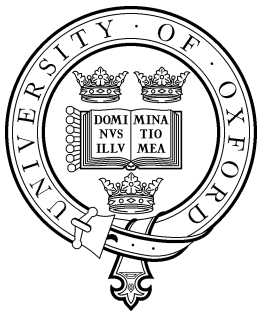
Factorisation Algorithms for Univariate and Bivariate Polynomials over Finite Fields

Fatima Khaled Abu Salem
Merton College

Thesis submitted for the degree of Doctor of Philosophy
Trinity Term, 2004



Oxford University Computing Laboratory
Programming Research Group

Factorisation Algorithms for Univariate and Bivariate Polynomials over Finite Fields

Fatima Khaled Abu Salem

Trinity Term, 2004

Abstract

In this thesis we address algorithms for polynomial factorisation over finite fields. In the univariate case, we study a recent algorithm due to Niederreiter [102] where the factorisation problem is reduced to solving a linear system over the finite field in question, and the solutions are used to produce the complete factorisation of the polynomial into irreducibles. We develop a new algorithm for solving the linear system using sparse Gaussian elimination with the Markowitz ordering strategy, and conjecture that the Niederreiter linear system is not only initially sparse, but also preserves its sparsity throughout the Gaussian elimination phase [3]. We develop a new bulk synchronous parallel (BSP) algorithm based on the approach of Göttfert (1994) for extracting the factors of a polynomial using a basis of the Niederreiter solution set over $\mathbb{F}_2$. We improve upon the complexity and performance of the original algorithm, and produce binary univariate factorisations of trinomials up to degree 400000 [1].

We present a new approach to multivariate polynomial factorisation which incorporates ideas from polyhedral geometry, and generalises Hensel lifting [2]. The contribution is an algorithm for factoring bivariate polynomials via polytopes which is able to exploit to some extent the sparsity of polynomials. We further show that the polytope method can be made sensitive to the number of nonzero terms of the input polynomial. We describe a sparse adaptation of the polytope method over finite fields of prime order which requires fewer bit operations and memory references for polynomials which are known to be the product of two sparse factors [4]. Using this method, and to the best of our knowledge, we achieve a world record in binary bivariate factorisation of a sparse polynomial with degree 20000. We develop a BSP variant of the absolute irreducibility testing via polytopes given in [45], producing a more memory and run time efficient method that can provide wider ranges of applicability [5]. We achieve absolute irreducibility testing of a bivariate and trivariate polynomial of degree 30000, and of multivariate polynomials with up to 3000 variables.

Acknowledgments

First and foremost, I thank my supervisor, Richard Brent, for his ceaseless support and guidance, and for having made possible that I join the DPhil program at Oxford University. I also thank my in-college tutors, Luke Ong and Ulrike Tillmann, and my advisor, Jeff Sanders, for valuable non-academic support whenever I sought it. I am grateful to Alan Lauder and Shuhong Gao, for collaboration during some stages of my work, and to Jon Lockley, Joe Pitt-Francis, and Fred Youhanaie, for their assistance as part of the Oxford Supercomputing Centre (OSC) support scheme. I acknowledge EPSRC for generous financial support during the past three years, and OSC for the computing facilities provided to run all the experiments described in this thesis. I also thank the Center for Advanced Mathematical Sciences at the American University of Beirut for allowing the use of their facilities during my numerous visits to Lebanon.

Finally, I am infinitely indebted to my parents, Karam and Khaled, for the countless ways in which they have supported me: my mother, for her comforting voice every other night, and my father, for his daily e-mails that never got interrupted. I am also grateful to my husband, Wafic, for his incessant concern and attention, having stood by me through the long time and distance, and for sincere advice on how to survive a new research experience. I thank my brother, Walid, for the many mathematical discussions, my sister, Dina, for the beautiful images reminding me that there is a lot to life other than a thesis, and my twin sister, Bushra, for reflecting a side of her that I would like to be. Last, special thanks to my precious niece, Nahida, whose existence since a few months ago and her smiling image through the cameras has provided me with a therapeutic inspiration during hard times.

To my Parents, and late Grandmother, Bushra

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Outline . . . 2

2 Preliminaries **5**
2.1 Rings and fields . . . 5
2.2 Field extensions and structure of finite fields . . . 8
2.3 Construction of finite fields . . . 10
2.4 Univariate polynomial arithmetic over finite fields . . . 12
2.5 The bulk synchronous parallel model (BSP) . . . 13

3 Factorisation algorithms **15**
3.1 Univariate factorisation . . . 15
3.1.1 Niederreiter's algorithm for small finite fields . . . 16
3.1.2 The square-free case . . . 17
3.1.3 The algorithm . . . 19
3.1.4 Acceleration of Niederreiter's algorithm over the binary field . . . 19
3.2 Bivariate factorisation . . . 23
3.2.1 Hensel lifting for bivariate polynomials . . . 24
3.3 Polynomials and Newton polytopes . . . 27
3.3.1 Indecomposable polytopes and absolute irreducibility . . . 29
3.3.2 Testing indecomposability of polytopes . . . 29
3.3.3 Constructing convex hulls in two dimensions . . . 32

4 A new sparse Gaussian elimination algorithm and the Niederreiter linear system for trinomials over $\mathbb{F}_2$ **35**
4.1 Introduction . . . 35
4.2 Setting the Niederreiter matrix over $\mathbb{F}_2$ . . . 36
4.3 Data structures for the sparse matrix M . . . 39
4.4 A new sparse Gaussian elimination algorithm . . . 41
4.4.1 Accessing entries along a column . . . 42
4.4.2 Implementing the Markowitz strategy . . . 42
4.4.3 Interchanging columns . . . 44
4.4.4 Adding rows . . . 48
4.4.5 A complete sparse Gaussian elimination algorithm . . . 54
4.5 Implementation and run times . . . 54

4.6 Conclusion . . . 55

5 A BSP model of the Göttfert algorithm for polynomial factorisation over $\mathbb{F}_2$ 57
5.1 Introduction . . . 57
5.2 A parallel approach to Göttfert's refinement of the Niederreiter algorithm . . . 57
5.2.1 Detecting parallelism in the Göttfert setting . . . 58
5.2.2 The parallel Göttfert algorithm . . . 61
5.2.3 The BSP cost of the algorithm . . . 65
5.2.4 Reduction of the algorithm's memory requirements . . . 69
5.3 Implementation and run times . . . 71
5.4 Conclusion . . . 72

6 Factoring polynomials via polytopes 74
6.1 Introduction . . . 74
6.2 Newton polytopes and Ostrowski's theorem . . . 74
6.3 Extending partial factorisations . . . 75
6.4 The polytope method . . . 78
6.4.1 An outline of the method . . . 78
6.4.2 Hensel lifting equations . . . 78
6.5 A geometric lemma . . . 80
6.5.1 On identifying irredundant dominating sets . . . 82
6.6 The main theorem . . . 84
6.7 On long division with remainder of Laurent polynomials . . . 87
6.8 The algorithm . . . 89
6.9 Examples and implementation . . . 92
6.9.1 Example . . . 92
6.9.2 Implementation . . . 97
6.10 Conclusion . . . 98

7 An efficient sparse adaptation of the polytope method over $\mathbb{F}_p$ and a record-high binary bivariate factorisation 99
7.1 Introduction . . . 99
7.2 Input model . . . 100
7.3 Pre-lifting stages . . . 101
7.3.1 Floating-point operations and integer overflow . . . 101
7.3.2 Computing Newt(f) . . . 102
7.3.3 Finding all irredundant sets of dominating edges . . . 103
7.3.4 Determining univariate edge polynomials . . . 105
7.3.5 Intersecting arbitrary lines with the polytope . . . 107
7.3.6 Computing the set of all integral points in Newt(f) . . . 109
7.3.7 Counting factors of univariate factorisations . . . 110
7.3.8 Summand counting and recovering algorithm . . . 112
7.3.9 Choosing coprime dominating edges factorisations . . . 116
7.4 The sparse lifting algorithm . . . 119
7.4.1 Detecting specialised coefficients . . . 120
7.4.2 Counting unspecialised terms . . . 121

7.5 Investigating one lifting step . . . 122
7.5.1 A complete description of a sparse lifting step . . . 123
7.5.2 Representing unknown polynomials and expressions . . . 126
7.5.3 Recovering H_{k_δ} . . . 130
7.6 Total run time and memory . . . 131
7.7 Computational results . . . 133
7.8 Conclusion . . . 134

8 Parallel absolute irreducibility testing via polytopes 136
8.1 Introduction . . . 136
8.2 Parallel bivariate absolute irreducibility testing . . . 137
8.2.1 Computing the set of all integral points in a polygon . . . 138
8.2.2 Constructing sets of points along paths of edges . . . 138
8.2.3 Detecting independent computations . . . 139
8.2.4 Constructing a balanced load scheme . . . 139
8.2.5 Constructing a balanced data distribution . . . 140
8.2.6 Removing repetitions in computation . . . 142
8.2.7 A BSP algorithm for testing polygon indecomposability . . . 143
8.3 Parallel multivariate absolute irreducibility testing . . . 148
8.3.1 A BSP algorithm for testing polytope indecomposability . . . 148
8.4 Implementation and Run Times . . . 154
8.5 Conclusion . . . 159

9 Conclusion 160
9.1 Discussion and future work . . . 160

Bibliography 163

Chapter 1

Introduction

1.1 Motivation

Symbolic computation has been the object of computer algebra systems for decades now, focusing on the use of computers to perform symbolic mathematics. The rudimentary elements of a computer algebra system consist in numbers and polynomials over a field, and the basic domains are the natural numbers, rational numbers, finite fields and polynomial rings [29, 55]. One major problem in this area has been the factorisation of polynomials over finite fields, an active area of research that has gained a lot of interest in the last few decades. Finite fields in general, and factorisation of polynomials over such fields in particular, have widespread applications in both theory and practice. In many instances it becomes essential to be able to factor any large degree polynomial or otherwise establish its irreducibility. Applications within mathematics appear in a variety of situations: Equation solving, such as in modeling the Cyclohexane molecule and studying the spatial conformations of Cyclohexane ([55], Chapter 24), and symbolic summation and integration [111, 112, 126]. In number theory, univariate polynomial factorisation can be used in finding complete partial fraction decompositions, and computing the number of points on elliptic curves [37]. In coding theory, univariate polynomial factorisation is used in constructing a useful class of cyclic codes, the BCH codes [9, 69, 95], some of which use polynomials over the binary field [16]. Some cryptosystems, in turn, are based on the Goppa codes, a generalisation of BCH codes [92, 93]. In cryptography, the factorisation of random polynomials over finite fields is used in some randomised methods for computing discrete logarithms over finite fields [37]. High degree sparse univariate polynomials in this respect have recently been used in developing public key cryptosystems: SPIFI [7], based on the difficulty of finding a sparse polynomial with specified values at some given points, and EnRoot [61], based on the difficulty of finding a solution to a given system of sparse polynomial equations over certain large rings.

Binary trinomials are also important in their role in constructing linear feedback shift registers, hence contributing to uses in random number generation, to constructing stream ciphers in cryptography, and to generating Hamming and BCH error correcting codes [9, 93, 98]. Primitive trinomials over $\mathbb{F}_2$ [18, 19] also play an important role in constructing fields of even characteristic and for computing with elements within such fields. In general, computation over the binary field is particularly fashionable because of its simplicity in computation, serving as an example to illustrate some of the attractive features of a particular algorithm over finite fields. It is also often used as the underlying field where applications in cryptography and coding theory employ

computations over finite fields [93].

Bivariate polynomial factorisation can be used to solve systems of polynomial equations using Gröbner bases ([55], Chapter 21). Methods for solving systems of algebraic equations have also been developed using multivariate factorisation [28]. Algebraic simplification and proving combinatorial identities can also make use of bivariate polynomial factorisations [55]. Absolute irreducibility testing of families of multivariate polynomials can be an important tool in deformation theory and number theory [47]. Multivariate factorisation has also been used in the classification of algebraic varieties [109]

Apart from applications in these domains, univariate polynomial factorisation over finite fields serves as a subproblem of other factorisation problems, namely, those pertaining to multivariate polynomials over finite fields, and univariate and multivariate polynomials over the field of rational numbers and finite extensions of the rationals [56].

Contemporary research in computer algebra aims both at wide functionality of the present algorithms (by solving a wide range of different problems) and at their speed (how large can the problems to solve be made, using reasonable resources of time and machinery) [55]. Our motives behind undertaking this line of research from a computational point of view stem from our belief that algorithmic trends in mathematics need to be examined and re-evaluated using the very same tools for which the algorithms were originally intended, which is the actual "computer" machine. A continuous challenge lies in connecting the new mathematical ideas on how to best perform a certain algorithm, with the computing approaches and tools that help materialise a particular algorithm. A variety of tools are made available towards this end. Improvements can touch upon issues exploiting sparsity, memory management and space reduction. Parallelism in computer algebra has also been interestingly demonstrated [31, 118, 129]. One of the main contributions of this thesis is to also investigate areas of parallelism whenever an improvement in run time or spatial complexity is desired. The main focus is on recent algorithms which have still not been thoroughly used: Niederreiter's algorithm for univariate polynomials and the polytope method for bivariate polynomials. The ultimate aim of our work is to bring about the best performance possible in those two approaches using a given hardware and reasonable time, achieving competitive factorisation records that in some cases have not been achieved before. In particular, we achieve the factorisation of a sparse binary bivariate polynomial with degree 20000, and the absolute irreducibility testing of a bivariate and trivariate polynomial of degree 30000, and of multivariate polynomials with up to 3000 variables.

1.2 Outline

Chapters 2 and 3 contain a preview of the mathematical ideas on which our work is based. In Chapter 4 we examine the Niederreiter algorithm [102, 103, 104, 105] which has been predicted to perform very well for sparse polynomials over the binary field. This prompts us to investigate the sparsity feature for trinomials in particular, those providing the most immediate model for sparse polynomial factorisation. We prove that the Niederreiter matrix is sparse in the case of a trinomial, and establish the exact sparsity pattern and density of the Niederreiter matrix [3]. We also develop a new algorithm for solving the sparse linear system directly to produce a basis for the solution set through Gaussian elimination and using the data structure of Gustavson [65], and show how the new algorithm circumvents the problems that have always been associated with this data structure in terms of elbow space and compression. Although it can be easily

modified to become a general linear solver for other various applications over $\mathbb{F}_2$, our experiments show that the algorithm can be very efficient in the cases when the matrix maintains a high level of sparsity throughout the reduction phase, typically an observed feature of the Niederreiter matrix.

These results are later incorporated into an algorithm for extracting the factors using a basis for the solution set, based on Göttfert's acceleration of the Niederreiter algorithm over $\mathbb{F}_2$ [59]. In Chapter 5, we develop a new BSP (bulk synchronous parallel) algorithm that outlines a clear dependency between the major computations involved in the factors extraction process, so that the resulting algorithm comprises a completely new task distribution process [1]. Our main reasons behind adopting such a model of parallelism are due to its features simplifying the cost analysis and its clear distinction between the three important phases of computation, communication, and synchronisation. Our BSP theoretical model results in an efficient BSP cost requiring relatively small communication and synchronisation costs, and the parallel algorithm achieves very good efficiency as confirmed by our experimental results. Combining the results of Chapters 4 and 5, the resulting hybrid algorithm provides a cheaper and more memory efficient alternative to the factorisation of trinomials over $\mathbb{F}_2$ than previously known dense implementations of the Niederreiter algorithm [110].

In Chapter 6 we become interested in algorithms for bivariate polynomial factorisations over finite fields. Based on joint work with Shuhong Gao and Alan Lauder [2], we introduce a new approach to bivariate polynomial factorisation which incorporates ideas from polyhedral geometry, and generalises Hensel lifting. The method exploits the sparsity of input polynomials so that bivariate polynomials can be processed significantly more quickly than using ordinary Hensel lifting. Given a bivariate polynomial over a field, one may associate with it a convex polytope in the two dimensional real space called its Newton polytope. A well known result is that if the polynomial factors, then its Newton polytope decomposes, in the sense of the Minkowski sum, into the Newton polytopes of the factors. If the polytope does not decompose, one immediately deduces that the polynomial must be irreducible. However, the converse is not necessarily true, and we are faced with the following problem: Given a decomposition of the polytope, can we recover a factorisation of the polynomial whose factors have Newton polytopes of that shape, or show that one does not exist. Our approach, motivated by Hensel lifting, is to assume that, along with the decomposition of the polytope, we are given appropriate factorisations of the polynomials defined by the edges of the Newton polytope. These polynomials will be essentially in one variable less, and the boundary factorisation of the input polynomial is then lifted into the Newton polytope, where the coefficients of the possible factors of the polynomial are revealed in successive layers. In standard Hensel lifting, instead of lifting from the boundary, one does so from a single edge. Uniqueness of the linear systems encountered during lifting can then be ensured by randomising the polynomial to enforce coprimality conditions and to make sure the edge being lifted from is sufficiently long. However, this randomisation is by substitution of linear forms, and this destroys the sparsity of the input polynomial. With the polytope method, uniqueness can be shown to hold in the bivariate case, only under certain coprimality conditions, and without restrictions on the lengths of the edges. As with Hensel lifting, the polytope method has an exponential worst-case running time, since the number of summands of a Newton polytope could in the worst case be exponential in the total degree of the associated polynomial. However, our experiments performed very efficiently in factorisations of sparse polynomials whose polytopes have few edges, and hence very few Minkowski decompositions. This leads us to consider possible extensions of this work, in the belief that it could be

a promising new method with fast performance in practice, despite its worst-case exponential time.

In Chapter 7, we pursue this work in an attempt to get as close as possible to solving the open problem of devising a sparse bivariate algorithm. The polytope method has potential features that one can exploit, but as it stands above, it argues for one major advantage suiting sparse polynomials, namely, the fact that the worst-case exponential search for summands can in fact be very small if the input polynomial is sparse with a Newton polytope having few decompositions. However, for an input bivariate polynomial of total degree d, the amount of work per extension of a given boundary factorisation is still of the order $O(d^4)$. We investigate whether this complexity can be reduced, in the case when the ground field is of prime order, to some bound which is directly dependent on the number of terms, say t. The method, which exploits both the fact that many of the coefficients corresponding to lattice points in the Newton polytope of the input polynomial (and hopefully its factors) are zero, and also the fact that many of the polynomials generated during the lifting steps are zero in general and sparse in the worst-case analysis, results in very high degree, sparse, bivariate binary factorisations for input degree equal to 20000. To the best of our knowledge, this is by far the highest binary factorisation achieved to date [4].

In Chapter 8, we examine multivariate polynomial absolute irreducibility over finite fields, a sub-problem which is indispensable for examining input polynomials before feeding them into a possibly expensive and nontrivial factorisation algorithm. In particular, we revisit an algorithm due to Gao and Lauder [45] that is also based on the use of polytopes. Motivated by their original findings and the special feature which makes absolute irreducibility testing largely dependent on the shape and the size of Newton polytopes, we investigate a BSP scheme that serves to extend the range of applicability of the algorithm, by making it possible to tackle significantly higher degrees, and by allowing a more efficient performance for low degree yet denser polynomials than those reported in [47]. We show that the algorithm can be optimally parallelised by constructing a balanced load scheme using the pattern of computations in the sequential case as in [45], and by adopting a corresponding data distribution representing lattice points inside polytopes in $\mathbb{R}^2$. The distribution not only adheres to the proposed load scheme, but also allows for a scalable parallel performance whose efficiency is reflected in our experiments. This then paves the way for the multivariate case, where a model involving parallelism at two different levels is described. The resulting improvement is shown to perform well for a wide range of input polynomials, achieving absolute irreducibility testing of bivariate and multivariate polynomials up to degree 30000, and of lower degree multivariate polynomials with up to 3000 variables [5].

We conclude with a summary of our work in Chapter 9, and outline possible lines of research of relevance to this thesis that can be undertaken in the future.

Chapter 2

Preliminaries

In this chapter we present a brief collection of classical terminologies and results describing the tools upon which the rest of the thesis is based. We start our discussion by defining several types of algebraic structures and properties necessary for the construction of finite fields. We also discuss issues related to the representation of polynomials over such fields, and for performing arithmetic of univariate polynomials over finite fields. We finally conclude with a brief description of the BSP model that we will adopt in all our parallel algorithms.

2.1 Rings and fields

For a complete overview of the statements and proofs of all assertions coming forward, we refer the reader to [9, 40, 55, 85, 93, 97].

Definition 2.1.1 *A group G is a set together with a binary operation $*$ operating on elements of G such that:*

i. $*$ is associative.

ii. There exists a unique element e in G (called the identity element) such that for all $a \in G$ we have:

$$a * e = e * a = a$$

iii. For each $a \in G$, there exists a unique element $a^{-1} \in G$ such that:

$$a * a^{-1} = a * a^{-1} = e$$

If the group also satisfies $a * b = b * a$ for all $a, b \in G$, then the group is called abelian (or commutative).

In what follows, 0 denotes the identity under $+$ and 1 the identity under $\cdot$.

Definition 2.1.2 *A ring $(R, +, \cdot)$ is a set R endowed with two binary operations $+$ and $\cdot$ (not necessarily the common operations of addition and multiplication) such that:*

i. R is an abelian group under $+$.

ii. $\cdot$ is associative.

iii. The distributive laws hold. That is, for all $a, b, c \in R$, we have:

$$a \cdot (b + c) = a \cdot b + a \cdot c \text{ and } (b + c) \cdot a = b \cdot a + c \cdot a.$$

a

Definition 2.1.3 *i. A ring R is called a ring with identity if the ring has a multiplicative identity; that is, if there exists an element $e \neq 0$ such that $a \cdot e = e \cdot a = a$ for all $a \in R$.*

ii. A ring is commutative if $\cdot$ is commutative.

iii. An integral domain is a commutative ring with identity $e \neq 0$ in which $a \cdot b = 0$ implies $a = 0$ or $b = 0$.

iv. A ring is called a division ring (or skew field) if its nonzero elements form a group under the operation $\cdot$.

v. A field is a commutative division ring.

In what follows we write ab as a shorthand for $a \cdot b$.

Definition 2.1.4 *A subset S of a ring R is called a subring of R provided S is closed under $+$ and $\cdot$ and S forms a ring under these operations.*

Definition 2.1.5 *Let $J \subset R$. Then J is called an ideal of R if J is a subring of R and we have $ra \in J$ and $ar \in J$ for all $a \in J$ and $r \in R$.*

For a commutative ring R, the smallest ideal containing a given element $a \in R$ is the ideal $(a) = \{ra + na : r \in R, n \in \mathbb{Z}\}$. If R contains an identity, then $(a) = \{ra : r \in R\}$.

Definition 2.1.6 *Let R be a commutative ring and J an ideal of R. If there exists an $a \in R$ such that $J = (a)$, then J is called the principal ideal generated by a.*

Definition 2.1.7 *Let J be an ideal of a ring R. We say that b and c in R belong to the same residue class modulo J if $b - c \in J$.*

If $a \in R$, the residue class of a modulo J will be denoted by $[a] = a + J$, consisting of all elements of R of the form $a + c$ for some $c \in J$.

Definition 2.1.8 *The ring of residue classes of the ring R modulo the ideal J under the operations:*

$$(a + J) + (b + J) = (a + b) + J$$
$$(a + J)(b + J) = ab + J$$

is called the residue class ring (or factor ring) of R modulo J. We denote this ring by R/J.

Theorem 2.1.1 *The ring $\mathbb{Z}_p$ of residue classes of the integers modulo the principal ideal generated by a prime p is a field.*

Definition 2.1.9 *Let p be a prime and let $\mathbb{F}_p = \{0, 1, ..., p - 1\}$. Let*

$$\phi : \mathbb{Z}/(p) \to \mathbb{F}_p$$

be the mapping defined by $\phi([a]) = a$ for $a = 0, 1, ..., p - 1$. Then ϕ induces a field structure on $\mathbb{F}_p$ which we call the Galois field of order p.

Theorem 2.1.2 *The mapping ϕ, as defined above, is an isomorphism, i.e.:*

$$\phi([a]+[b]) = \phi([a]) + \phi([b]) \text{ and } \phi([a][b]) = \phi([a])\phi([b]).$$

It is clear that the finite field $\mathbb{F}_p$ has zero element 0 and identity 1. Moreover, its structure is identical to that of $\mathbb{Z}_p$. An obvious advantage to this is the fact that computing with elements of $\mathbb{F}_p$ reduces to ordinary arithmetic of integers modulo p.

Definition 2.1.10 *Let R be an arbitrary ring. Suppose that there exists a positive integer n such that $nr = 0$ for every $r \in R$. We call the characteristic of R the least such positive integer n. If no such positive integer exists, R is said to have characteristic 0.*

Theorem 2.1.3 *A finite field has prime characteristic.*

Theorem 2.1.4 *Let R be a commutative ring of prime characteristic p. Let $a, b \in R$ and $n \in \mathbb{N}$. Then*

$$(a+b)^{p^n} = a^{p^n} + b^{p^n}.$$

In the following, all polynomials are assumed to be in one variable.

Definition 2.1.11 *Let R be an arbitrary ring. The ring formed by polynomials over R with the usual operations of polynomial addition and multiplication is called the polynomial ring over R and denoted by $R[x]$.*

Definition 2.1.12 *Let $\mathbb{F}$ be a field. The field of fractions of the polynomial ring $\mathbb{F}[x]$ is the set $\mathbb{F}(x)$ of rational functions in x with coefficients in the field $\mathbb{F}$.*

Definition 2.1.13 *Let $f(x) = \sum_{i=0}^{n} f_i x^i$ be a nonzero polynomial over R such that $f_n \neq 0$. Then we call n the degree of the polynomial f (denoted by $\deg(f)$), f_n the leading coefficient of $f(x)$, and f_0 the constant term. If R has identity 1 and the leading coefficient of $f(x)$ is 1, then $f(x)$ is called a monic polynomial.*

Definition 2.1.14 *A Laurent polynomial with coefficients in the field $\mathbb{F}$ is a polynomial of the form*

$$a_{-m}x^{-m} + a_{-(m-1)}x^{-(m-1)} + ... + a_{-1}x^{-1} + a_0 + a_1 x + ... + a_n x^n$$

where $a_i \in \mathbb{F}$, for $-m \leq i \leq n$, $m, n \in \mathbb{Z}_{\geq 0}$, and where only finitely many of the a_i's are nonzero.

Definition 2.1.15 *Given a nonzero Laurent polynomial $f = \sum_{i=m}^{n} a_i x^i$, where $m, n \in \mathbb{Z}$, its degree is defined to be the difference $n - m$.*

Definition 2.1.16 *A regular polynomial is a polynomial whose indeterminates cannot have negative exponents.*

Within applications involving only regular polynomials, we set $\deg(0) = -\infty$ or -1 depending on the suitability of each. If $f = 0$ is treated as a Laurent polynomial, we set $\deg(0) = -\infty$. Polynomials of degree equal to zero are the nonzero constant polynomials. In all the following, $\mathbb{F}$ denotes an arbitrary field.

Theorem 2.1.5 *Let $g \neq 0$ be a polynomial in $\mathbb{F}[x]$. Then for any $f \in \mathbb{F}[x]$ there exist polynomials $q, r \in \mathbb{F}[x]$ such that*

$$f = qg + r, \text{ where } deg(r) < deg(g).$$

Definition 2.1.17 *A polynomial $f \in \mathbb{F}[x]$ is said to be irreducible over $\mathbb{F}$ if f has positive degree and $f = bc$ with $b, c \in \mathbb{F}[x]$ implies that either b or c is a constant polynomial (equivalently, either b or c belongs to $\mathbb{F}$). In other words, f is irreducible if and only if it has only trivial factors.*

Theorem 2.1.6 *Let $f \in \mathbb{F}[x]$ have positive degree. Then f can be written in the form*

$$f = ag_1^{e_1} \cdots g_r^{e_r},$$

where $a \in \mathbb{F}$, $g_1, ..., g_r$ are distinct irreducible polynomials in $\mathbb{F}[x]$, and $e_1, ..., e_r$ are positive integers. Moreover, this factorisation is unique up to the order of factors and multiplication by units (the nonzero constants from $\mathbb{F}$).

Theorem 2.1.7 *Let $f \in \mathbb{F}[x]$. Then the residue class ring $\mathbb{F}[x]/(f)$ is a field if and only if f is irreducible over $\mathbb{F}$. In particular, the residue classes comprising $\mathbb{F}[x]/(f)$ are of the form $r + (f)$, where r runs through all polynomials in $\mathbb{F}[x]$ with $deg(r) < deg(f)$. Thus, if $\mathbb{F} = \mathbb{F}_p$ and $deg(f) = n \geq 0$, then $\mathbb{F}_p[x]/(f)$ has p^n elements.*

Theorem 2.1.8 *Let $f \in \mathbb{F}[x]$ be a polynomial of degree n over $\mathbb{F}$. Then f can have at most n distinct roots in $\mathbb{F}$.*

2.2 Field extensions and structure of finite fields

We call a subfield K of $\mathbb{F}$ that subset of $\mathbb{F}$ which itself is a field under the operations of $\mathbb{F}$. We call $\mathbb{F}$ an extension field of K.

Let $\mathbb{K}$ be a field and $\mathbb{F}$ a finite extension of it. Then $\mathbb{F}$ can be viewed as a vector space over K. $\mathbb{F}$ is called a finite extension if it is a finite dimensional vector space. Its degree (denoted by $[\mathbb{F} : \mathbb{K}]$) is precisely its dimension as a vector space.

Lemma 2.2.1 *Let $\mathbb{F}$ be a finite field containing a subfield $\mathbb{K}$ with q elements. Let $m = [\mathbb{F} : \mathbb{K}]$. Then $\mathbb{F}$ has q^m elements.*

Theorem 2.2.1 *The order of a finite field $\mathbb{F}$ is a power of its characteristic. Consequently, every finite field has order p^n, where p is prime and n is the degree of an irreducible polynomial over $\mathbb{F}_p$ such that $\mathbb{F}$ is isomorphic to $\mathbb{F}_p[x]/(f)$.*

Definition 2.2.1 *A field $\mathbb{F}$ is said to be algebraically closed if every univariate polynomial of degree at least 1 with coefficients in $\mathbb{F}$ has a zero in $\mathbb{F}$.*

Definition 2.2.2 *A field extension $\mathbb{L}$ of $\mathbb{F}$ is said to be algebraic if every element of $\mathbb{L}$ is a root of a nonzero polynomial with coefficients in $\mathbb{F}$.*

Definition 2.2.3 *An algebraic closure $\overline{\mathbb{F}}$ of a field $\mathbb{F}$ is an algebraic extension of $\mathbb{F}$ that is algebraically closed.*

Theorem 2.2.2 *The algebraic closure of a field $\mathbb{F}$ is unique up to isomorphism which fixes all elements of $\mathbb{F}$.*

Definition 2.2.4 *Let $\mathbb{F}$ be a finite field and $\mathbb{K}$ a subfield of $\mathbb{F}$. Let $f \in \mathbb{K}[x]$ be a polynomial of degree $n > 0$ and leading coefficient a. Then f is said to split in $\mathbb{F}$ if it can be written as a product of linear factors in $\mathbb{F}[x]$ or equivalently, if there exists some $\alpha_1, ..., \alpha_n \in \mathbb{F}$ such that*

$$f(x) = a(x - \alpha_1) \cdots (x - \alpha_n).$$

The field $\mathbb{F}$ is called a splitting field of f.

The following theorem states that finite fields of any prime power order exist and are essentially unique in structure though their representation may vary.

Theorem 2.2.3 *For every prime p and every positive integer n there exists a finite field with p^n elements. Any finite field with $q = p^n$ elements is isomorphic to the splitting field of $x^q - x$ over $\mathbb{F}_p$.*

Theorem 2.2.4 *Let $\mathbb{F}$ be a finite field with q elements and $a \in \mathbb{F}$. Then*

$$a^q = a.$$

Theorem 2.2.5 *Let $\mathbb{F}$ be a finite field with q elements and $\mathbb{K}$ a subfield of $\mathbb{F}$. Then the polynomial $x^q - x$ in $\mathbb{K}[x]$ factors in $\mathbb{F}[x]$ and the factorisation is given by:*

$$x^q - x = \prod_{a \in \mathbb{F}} (x - a).$$

In this case, $\mathbb{F}$ is a splitting field of $x^q - x$ over $\mathbb{K}$.

Theorem 2.2.6 *For $q = p^m$, $\mathbb{F}_q$ contains an isomorphic copy of $\mathbb{F}_p$ as a subfield. In other words, $\mathbb{F}_q$ is an extension field of $\mathbb{F}_p$ of degree m.*

Theorem 2.2.7 *Let $\mathbb{F}_q$ be a finite field and $n \in \mathbb{N}$. The product of all monic irreducible polynomials over $\mathbb{F}_q$ whose degrees divide n is equal to $x^{q^n} - x$.*

In what follows let $\mathbb{F}_q^*$ denote the set of nonzero elements of $\mathbb{F}_q$.

Theorem 2.2.8 *The nonzero elements of $\mathbb{F}_q$ form a group under multiplication. Furthermore, this group is cyclic of order $q - 1$.*

Definition 2.2.5 *An irreducible polynomial $f(x) \in \mathbb{F}_p[x]$ of degree m is called a primitive polynomial if x is a generator of $\mathbb{F}_{p^m}^*$, the cyclic multiplicative group of nonzero elements in $\mathbb{F}_{p^m} = \mathbb{F}_p[x]/(f(x))$.*

Theorem 2.2.9 *For each $m \geq 1$, there exists a monic primitive polynomial of degree m over $\mathbb{F}_p$.*

In conclusion, we define the formal derivative over fields bearing in mind that the definition entailed does not involve the idea of a limit (because of the absence of the notion of distance or topology on a field). Instead, we adopt the following definition which applies to arbitrary commutative rings and not just fields :

Definition 2.2.6 *Let R be an arbitrary commutative ring with 1. Let $f = \sum_{0 \leq i \leq n} f_i x^i \in R[x]$. We define the formal derivative of f by*

$$f' = \sum_{1 \leq i \leq n} i f_i x^{i-1}.$$

Theorem 2.2.10 *Let R be a commutative ring with 1, $g_1, ..., g_r \in R[x]$, and $e_1, ..., e_r$ positive integers. We then have:*

$$(g_1^{e_1} \cdots g_r^{e^r})' = \sum_{1 \leq i \leq r} e_i g_i' g_i^{e_i - 1} \prod_{j \neq i} g_j^{e_j} = \sum_{1 \leq i \leq r} e_i g_i' \frac{f}{g}_i$$

where $f = g_1^{e_1} \cdots g_r^{e_r}$.

2.3 Construction of finite fields

We have seen in the previous section that, given $q = p^m$ for some $m \geq 1$, we can always determine a finite field of order q. In addition, any two fields of the same order are isomorphic, i.e. structurally the same; however, the difference in representing isomorphic copies of those fields is essential to practical applications where one has to come up with the most suitable choice of field representation.

We start our discussion on how to construct finite fields by considering the case $q = p$. In this case, we know that $\mathbb{F}_p$ is isomorphic to $\mathbb{Z}_p$ and so the finite field can be taken to be the set of integers $\{0, ..., p-1\}$. If $q = p^m$ where $m > 1$, the representation can become more difficult. One efficient way to construct fields of prime power order is the following. We have seen that, if $f(x)$ is an irreducible polynomial of degree m over $\mathbb{F}_p$, for some $m \geq 1$ and p a prime number, then $\mathbb{F}_p[x]/(f(x))$ is a field consisting of p^m elements. By the uniqueness (up to isomorphism) of finite fields, we know that $\mathbb{F}_p[x]/(f(x))$ can represent all fields $\mathbb{F}_q$ of order $q = p^m$. Thus, elements of $\mathbb{F}_q$ can be represented as polynomials taken modulo the polynomial f. Moreover, if $f(x)$ is primitive, we know that x is a generator of the group $\mathbb{F}_q^*$; in other words, all $q-1$ nonzero elements of $\mathbb{F}_q$ are obtained by computing $x^i \bmod f(x)$ for $i = 1, ..., q-1$, where $f(x)$ is the primitive polynomial of degree m over $\mathbb{F}_p$ used to construct the finite field. It is now obvious that elements of the finite field $\mathbb{F}_q$, where $q = p^m$, can be represented by polynomials in $\mathbb{F}_p[x]$ of degree less than m. Subtraction and addition of elements of $\mathbb{F}_q$ are the usual operations as performed among polynomials in $\mathbb{F}_p[x]$. The product of two elements $g_1(x)$ and $g_2(x)$ of $\mathbb{F}_q$, however, is obtained by multiplying $g_1(x)$ with $g_2(x)$ and reducing the result modulo $f(x)$. Multiplicative inverses and gcds of elements of $\mathbb{F}_q$ can be obtained using the Extended Euclidean Algorithm in $\mathbb{F}_p[x]$, always followed by reduction modulo $f(x)$ [97, 98].

As a result, the representation of a univariate polynomial in $\mathbb{F}_q[x]$ becomes, informally speaking, that of a bivariate polynomial (a polynomial with two variables) in $\mathbb{F}_p[x]$. To illustrate, if $h(x) \in \mathbb{F}_q[x]$, then we can write

$$h(x) = \sum_{0 \leq i \leq n} h_i x^i \text{ where } h_i \in \mathbb{F}_q.$$

However, $h_i \in \mathbb{F}_q$ implies that h_i is a polynomial over $\mathbb{F}_p$, except that it is regarded as a "constant" in $\mathbb{F}_q$.

The following example illustrates how we can construct a finite field and represent a polynomial in the ring of polynomials over that field.

Example [98]

We are given $p = 2$, $m = 4$, and $f(x) = x^4 + x + 1$ a primitive polynomial over $\mathbb{F}_2$.

i. Elements of $\mathbb{F}_{2^4}$ can be generated in two ways as follows:

Method 1 :

We know that $\mathbb{F}_{2^4}$ consists of all polynomials over $\mathbb{F}_2$ of degree less than 4. As a result, we have:

$$\mathbb{F}_{2^4} = \{a_3x^3 + a_2x^2 + a_1x + a_0 \mid a_i \in \{0, 1\}\},$$

which confirms that $\mathbb{F}_{2^4}$ consists of 16 elements.

Method 2 :

Since $f(x)$ is a primitive polynomial of degree 4 over $\mathbb{F}_2$, we can generate all elements of $\mathbb{F}_{2^4}$ by computing x^i for $i = 0, ..., 2^4 - 2$ and reducing the result modulo $f(x)$. The computations are summarised in table 1.

t	$x^t \bmod f(x)$
0	1
1	x
2	x^2
3	x^3
4	$x + 1$
5	$x^2 + x$
6	$x^3 + x^2$
7	$x^3 + x + 1$
8	$x^2 + 1$
9	$x^3 + x$
10	$x^2 + x + 1$
11	$x^3 + x^2 + x$
12	$x^3 + x^2 + x + 1$
13	$x^3 + x^2 + 1$
14	$x^3 + 1$

Table 2.1: The powers of x modulo $f(x) = x^4 + x + 1$.

ii. We can add any two elements of $\mathbb{F}_{16}$ using regular polynomial addition. Since all elements are reduced modulo $f(x)$, the resulting polynomial sum would need only be reduced modulo 2.

iii. Two elements in $\mathbb{F}_{16}$ can be multiplied as polynomials and then reduced modulo $f(x)$. For instance,

$$(x^3 + x^2 + 1) \cdot (x^3 + 1) = x^6 + x^5 + x^2 + 1 \equiv (x^3 + x^2 + x + 1) \bmod f(x).$$

It is worth noting that multiplication can be performed more easily using a look-up of indices

only. For example,

$$x^3 + x^2 + 1 = x^{13} \bmod f(x),$$
$$x^3 + 1 = x^{14} \bmod f(x),$$

and thus to perform the product $(x^3 + x^2 + 1) \cdot (x^3 + 1)$ we calculate

$$x^{13} \cdot x^{14} = x^{27} \equiv x^{12} \bmod f(x) = (x^3 + x^2 + x + 1).$$

This kind of table look-up (called the Zech logarithms representation) can be precomputed and is efficient for small values of q only.

iv. The inverse of $x^3 + x + 1$ in $\mathbb{F}_{16}$ is given by $x^2 + 1$. Using the Extended Euclidean algorithm for polynomials, we can verify that

$$(x^3 + x + 1) \cdot (x^2 + 1) = x^5 + x^2 + x + 1 \equiv 1 \bmod f(x).$$

v. Polynomials in $\mathbb{F}_{16}[x]$ can be constructed with the help of elements of the field serving as the constant coefficients. For instance, an example of a polynomial of degree 5 over $\mathbb{F}_{16}$ is given by:

$$f(y) = (x^3 + x + 1)y^5 + (x + 1)$$

where $x + 1$ is the constant term of the polynomial.

2.4 Univariate polynomial arithmetic over finite fields

Arithmetic of univariate polynomials over finite fields covers operations such as addition, multiplication, division with a remainder, gcd computation, and repeated squaring. Such algorithms fall into two categories, the first of which is the classical arithmetic, where the operations are implemented literally as in their definition. The complexity of these algorithms is hopefully greater than the corresponding ones in the second category of "fast" arithmetic, such as Karatsuba's multiplication algorithm, Schönhage and Strassen's multiplication algorithm, and the Fast Fourier Transform [55]. Since fast arithmetic does not always provide an improvement in performance for input size under certain cross-over points, one has to make a careful choice on which options to use based on the problem at hand. In our implementations, we use the classical algorithms for general arithmetic purposes. Detailed discussions of classical and fast arithmetic algorithms can be found in [21, 29, 55].

The complexity of the algorithms below is measured in terms of the maximum number of arithmetic operations required over $\mathbb{F}_q$, where $q = p^m$ for $m \geq 1$, and all polynomials are understood to be univariate. In what follows, let $\mathbb{F}_q$ be again a field with q elements where, as usual, $q = p^m$ for some prime p and a positive integer m. $\log x$ denotes the binary (base 2) logarithm of x.

Theorem 2.4.1 *Two polynomials of degree at most n over $\mathbb{F}_q$ can be added using at most $O(n)$ operations in $\mathbb{F}_q$.*

We note that there is no useful alternative to the classical addition algorithm.

Theorem 2.4.2 *Two polynomials of degree at most n over $\mathbb{F}_q$ can be multiplied by the classical algorithm using at most n^2 operations in $\mathbb{F}_q$. Fast algorithms perform this multiplication using $O(n \log n \log\log n)$ operations in $\mathbb{F}_q$.*

Definition 2.4.1 *Let R be a commutative ring with 1. Let*

$$M : \mathbb{N}_{>0} \to \mathbb{R}_{>0}$$

be a function such that two polynomials in $R[x]$ of degree less than n can be multiplied using at most $M(n)$ operations in R. Then M is called a multiplication time for $R[x]$.

We have the following from [55]:

$$\begin{gathered} M(n) \geq n, \\ M(n)/n \geq M(k)/k \text{ if } n \geq k, \\ M(nk) \leq k^2 M(n), \\ M(nk) \geq kM(n), \\ M(n+k) \geq M(n) + M(k), \end{gathered}$$

for all $m, n \in \mathbb{N}_{>0}$.

Theorem 2.4.3 *Let f be a polynomial of degree $n > 0$ over $\mathbb{F}_q$ and g be a polynomial of degree m such that $0 < m \leq n$. Then the division with remainder of f by g requires $O(M(n))$ operations in $\mathbb{F}_q$.*

The Euclidean algorithm can be applied to compute the gcd of polynomials over finite fields. In particular, the Extended Euclidean algorithm can also be used to determine the inverse of a polynomial over a finite field. There are also two classes (classical and fast) corresponding to these algorithms.

Theorem 2.4.4 *Let f and g be two polynomials of degree at most n over $\mathbb{F}_q$. Then $\gcd(f, g)$ can be found using $O(M(n)\log(n))$ operations in $\mathbb{F}_q$, where $M(n)$ is the multiplication cost defined above.*

Theorem 2.4.5 *Let $\mathbb{F}$ be a field and $f \in \mathbb{F}[x]$ of degree n. Let R be the corresponding residue ring $\mathbb{F}[x]/(f)$. Then a multiplication in R can be performed using $6M(n) + O(n)$ arithmetic operations in $\mathbb{F}$, and an inverse with at most $(24M(n) + O(n)) \log n$ operations in $\mathbb{F}$.*

Corollary 2.4.1 *Every arithmetic operation in the finite field of order p^m for some prime p and positive integer m can be performed using at most $O(m \log m \log\log m)$ operations in $\mathbb{F}_p$.*

2.5 The bulk synchronous parallel model (BSP)

The *bulk synchronous parallel model* is a model for parallel programming which provides a simple framework to achieve portable parallel algorithms independent of the architecture of the computer on which the parallel work is carried out. The model is attractive because of its simple cost function which helps predict the running time of parallel algorithms before implementing them, and has been successfully used in a variety of applications (see [14, 71] for instance). For

a detailed description of the BSP model, we refer the reader to [15, 127]. In our implementation of the BSP model, we use the standard BSP library [15, 66, 67]. An alternative is the Paderborn University BSP (PUB) library [20] which has the extra feature of allowing subset synchronisation and hence, very importantly, nested parallelism.

A BSP computer consists of a set of p processors each with its own private memory, and having remote access to other processors' private memories through a communication network. A BSP algorithm consists of a sequence of parallel steps, denoted by *supersteps*. A computation superstep is a series of computations performed on local data available to the processor before the superstep. A communication superstep is a series of communications in the form of sending or receiving a number of non-local data between processors that are needed to perform local computations. Communication supersteps are followed by *synchronisation barriers*, whereby all transferred data is updated. A BSP computer can be described by the following four parameters:
• p, the number of processors available;
• s, the processor speed in flop/sec;
• $g(p)$, the time (in flop time units) it takes to communicate (send or receive) a data element among p processors;
• $\ell(p)$, the time (in flop time units) it takes all p processors to synchronise.
We distinguish between the BSP cost of an algorithm and its expected running time. The BSP cost is established using the parameters g and ℓ and the estimate of the execution time is obtained by dividing the BSP cost in flop time units by s, the single processor speed. The BSP cost of an algorithm is simply the sum of the BSP costs of its supersteps. The complexity of a superstep is defined as

$$w_{max} + g(p) \cdot h_{max} + \ell(p),$$

where w_{max} is the maximum number of flops performed, and h_{max} is the maximum number of messages sent or received by any one processor during that superstep. In our applications over the binary field, floating point operations correspond to binary operations, and thus all costs are understood to be expressed in bit operations.

Chapter 3

Factorisation algorithms

In this chapter we present the key algorithms in the literature of polynomial factorisation on which our work is based. We discuss in more detail Niederreiter's algorithm and Göttfert's refinement of the algorithm over fields of characteristic 2, give a brief outline of Hensel lifting for bivariate polynomial factorisation, and present a review of Newton polytopes in relation to absolute irreducibility of multivariate polynomials. For a broad survey on the origins of polynomial factorisation we refer the reader to [29, 55, 74, 79].

3.1 Univariate factorisation

A basic premise in the discussion of polynomial factorisation is its uniqueness, the fact that for any field $\mathbb{F}$ the polynomials in $\mathbb{F}[x_1, ..., x_n]$ can be uniquely factored into a product of irreducible polynomials, and that this factorisation is unique up to the order of the factors and the multiplication by units (which are the nonzero constants in $\mathbb{F}[x_1, ..., x_n]$). Since we are interested in polynomials over finite fields, we restrict the discussion to the case when $\mathbb{F}_q$ is a finite field of order $q = p^m$, for some $m \geq 1$, and where the corresponding ring of polynomials over $\mathbb{F}$ is denoted by $\mathbb{F}_q[x]$.

Definition 3.1.1 *A polynomial f is square-free if and only if it is not divisible by a non-constant square.*

Definition 3.1.2 *If $f = g_1^{e_1}...g_r^{e_r}$ for some irreducible polynomials $g_1, ..., g_r$ and exponents $e_i \geq 0$, for $i = 1, ..., r$, then the product $g_1...g_r$ is called the square-free part of f.*

In what follows, let $f \in \mathbb{F}_q[x]$ denote a polynomial of degree n. The factorisation of f over $\mathbb{F}_q$ consists of determining pairwise distinct monic irreducible polynomials $g_1, ..., g_r \in \mathbb{F}_q[x]$ and positive integers $e_1, ..., e_r$ such that

$$f(x) = lc(f)g_1^{e_1}...g_r^{e_r},$$

where $lc(f)$ denotes the leading coefficient of f.

The algorithms fall into two classes: deterministic algorithms and probabilistic (or randomised) ones. The general aim is to devise algorithms with running time bounded by a polynomial in the input size, i.e. the total degree of the polynomial to be factorised and the logarithm of the order of the finite field. Musser presented an algorithm for the square-free decomposition

of univariate polynomials over the integers [101], which has been extended to the multivariate case by Yun [133]. Berlekamp gave the first deterministic algorithm for univariate factorisation over a finite field [8]. Kaltofen and Shoup [82] and Shoup [120] provided an algorithm with subquadratic time based on asymptotically fast power series composition algorithms from [17], and the algorithm is known to perform best for large finite fields. Kaltofen and Lobo introduced a black box representation for factoring high degree polynomials over finite fields using the Berlekamp algorithm [81]. The work of Berlekamp [8, 10], Cantor and Zassenhaus [22], and Kaltofen and Shoup [82] are among many examples where univariate factorisation algorithms have successfully achieved quadratic or subquadratic running times in the input size.

Perhaps the simplest approach with which to view factoring univariate polynomials over finite fields consists of performing the three stages of:

- square-free factorisation,
- distinct-degree factorisation, and
- equal-degree factorisation.

Square-free factorisation concerns eliminating multiple factors of a non-square-free polynomial f, which amounts to computing $g_1 ... g_r$ such that $f = lc(f) g_1^{e_1} ... g_r^{e_r}$ for positive integers e_i. A well known algorithm in this respect is due to Yun [133], and returns the square-free part of f deterministically using $O(M(n) \log n + n \log(q/p))$ operations in $\mathbb{F}_q$. Once a polynomial is reduced to a square-free form, its factorisation can be achieved by simply decomposing the square-free part. First, one calls Gauss's distinct-degree factorisation, which operates on the polynomial to separate its factors according to their degree. Let f be a non-constant, square-free polynomial of degree n over $\mathbb{F}_q$. The distinct-degree decomposition of f consists of the sequence $(w_1, ..., w_s)$ $(w_s \neq 1)$ of polynomials such that w_i is the product of all monic irreducible polynomials in $\mathbb{F}_q[x]$ of degree i that divide f. Moreover, the distinct-degree factorisation is the process of computing this sequence [55, 56]. The process is deterministic and can be shown to require $O(sM(n) \log(nq))$ operations in $\mathbb{F}_q$, where s is the largest degree of an irreducible factor of f. Finally, equal-degree factorisation solves the remaining problem by splitting all the factors of the same degree whose product has been generated by the preceding algorithm for distinct-degree factorisation. Proposed in probabilistic form by Cantor and Zassenhaus [22], the algorithm takes as input a monic byproduct of the distinct-degree factorisation, say g, with r monic irreducible factors all known to have some degree $d \leq n$. It returns all such irreducible

factors with probability of failure less than or equal to half, using an expected number of $O((d \log q + \log n) M(n) \log r)$ field operations. For more extensive details related to the above algorithms, we refer the reader to [29, 55, 56].

3.1.1 Niederreiter's algorithm for small finite fields

We now discuss an important class of *Linear algebra* based algorithms, so called since they reduce the factorisation algorithm to solving a linear system over the field in question, and using the solutions to produce non-trivial factors of the input polynomial. The earliest work in this respect was due to E. Berlekamp, and appeared successively in two versions, a deterministic and a probabilistic one, designed to work over small and large finite fields respectively [8, 9, 10, 22, 29, 55, 93]. Berlekamp's algorithm for small finite fields achieves deterministically a complete factorisation into irreducibles in $O(n^\omega + rqM(n) \log n)$ field operations, where r is the number

of distinct monic irreducible factors of f and ω the exponent for solving an $n \times n$ linear system over $\mathbb{F}_q$, and where $r \approx O(\log n)$ [83, 84, 100]. Note that $\omega = 3$ using classical direct methods and $\omega = \log_2 7$ using for instance Strassen's fast matrix multiplication algorithm. For large finite fields, it succeeds with probability at least 1/2 and requires $O(n^\omega + rM(n) \log n \log q \log r)$ field operations. The bottlenecks associated with these methods usually concern the linear algebra phase, where the costs of setting up the linear system, solving it or even storing it dominate the operational and spatial complexities. However, it is perhaps paradoxically this very aspect which makes these algorithms attractive in practice, since many techniques already known to improve upon the performance of linear solvers over finite fields can be used to expedite the entire factorisation algorithm, as can be demonstrated in the factorisation records of [1, 38, 39, 110, 117].

In this section we report on a relatively recent factorisation algorithm for univariate polynomials in the linear algebra based class. For proofs of the results given below, we refer the reader to [102, 103, 104, 105]. First proposed by Niederreiter in [102], the algorithm has received a lot of attention particularly for its effectiveness over binary fields. Niederreiter's original contribution first addressed small finite fields, specifically, fields of prime order only. At the heart of the algorithm is the study of the differential equation:

$$y^{(p-1)} + y^p = 0$$

of order $p-1$ in the rational field $\mathbb{F}_p(x)$. Here, $y^{(p-1)}$ denotes differentiation of order $p-1$ and y^p denotes exponentiation of order p. Although this equation can still be used for factorisation over fields with prime power order, we shall restrict our discussion to the case when the ground field is $\mathbb{F}_p$, where p is a prime. Since we have seen that any arbitrary polynomial can be made square-free, we can also assume that f is a square-free polynomial of degree $n > 0$ over $\mathbb{F}_p$, and that we aim to determine its monic irreducible factors $g_1, g_2, ..., g_r \in \mathbb{F}_p[x]$.

3.1.2 The square-free case

Let $L(y)$ denote the expression $y^{(p-1)} + y^p$. Several properties characterise the differential equation $L(y) = 0$, most important of which are that $L(y)$ is a linear operator on the vector space $\mathbb{F}_p(x)$ over $\mathbb{F}_p$, and that the solutions of $L(y) = 0$ form a linear subspace of $\mathbb{F}_p(x)$ [102]. Niederreiter considers those solutions of $L(y) = 0$ with fixed denominator f: If we write $y = h/f$ with $h \in \mathbb{F}_p[x]$, the corresponding solution space defined by:

$$\mathcal{N} = \{h \in \mathbb{F}_p[x] : \left(\frac{h}{f}\right)^{(p-1)} + \left(\frac{h}{f}\right)^p = 0\}$$

constitutes the so called *Niederreiter* linear space, which forms an $\mathbb{F}_p$-vector space [102]. Elements of this set can be described explicitly as follows:

Theorem 3.1.1 *[102] Let $f = g_1 ... g_r$ be the decomposition of f into distinct irreducible factors and let $\deg(f) = n > 0$. The solutions to $L(y) = 0$ such that $y = \frac{h}{f}$ are given by*

$$y = \sum_{j=1}^{r} a_j f \frac{g_j'}{g_j} \text{ with } a_1, ..., a_r \in \mathbb{F}_p.$$

If f is as above and h is an unknown polynomial of degree less than $\deg(f) = n$, then the polynomials on both sides of the equation

$$f^p \left(\frac{h}{f}\right)^{(p-1)} = -h^p \tag{3.1}$$

have degrees less than or equal to $(n-1)p$, and both sides of the equation are polynomials in x^p. The major implications of this lead to the crucial result:

Theorem 3.1.2 *[102] If $y = \frac{h}{f}$ with f fixed of degree n and $h(x)$ is an unknown polynomial in $\mathbb{F}_p[x]$ of degree less than n, then solving $y^{(p-1)} + y^p = 0$ reduces to solving an $n \times n$ linear system over $\mathbb{F}_p$.*

For the sake of clarity and illustration we shall replicate the proof of the above theorem from [102]:

Proof: Write $h(x) = \sum_{k=0}^{n-1} h_k x^k$. Rewrite $y^{(p-1)} + y^p = 0$ as

$$f^p \left(\frac{h}{f}\right)^{(p-1)} = -h^p.$$

Since the polynomials on both sides of Eq. (3.1) are of degree less than or equal to $(n-1)p$ and are polynomials in x^p, this indicates that (3.1) holds if and only if the coefficients of x^{jp}, $0 \leq j \leq n-1$, agree on both sides of the equation. Identifying the coefficients of $f^p \left(\frac{h}{f}\right)^{(p-1)}$ with those of $-h^p$ results in an $n \times n$ system of linear equations in $h_0, ..., h_{n-1}$, the unknown coefficients of h. Let N_f be the $n \times n$ coefficient matrix of $f^p \left(\frac{h}{f}\right)^{(p-1)}$. Then since

$$h(x)^p = h(x^p) = \sum_{k=0}^{n-1} h_k x^{kp},$$

this system can be rearranged as

$$(N_f - I_n)\,\mathbf{h}^T = \mathbf{0}$$

where $\mathbf{h} = (h_0, ..., h_{n-1}) \in \mathbb{F}_p^n$ and $(N_f - I_n)$ is an $n \times n$ matrix over $\mathbb{F}_p$.

It can be shown, as a consequence of the above in relation to polynomial reducibility, that $\text{Rank}(N_f - I_n) = n - r$, where n is the degree of f and r is the number of irreducible factors of f. In particular, f is irreducible over $\mathbb{F}_p$ iff $\text{Rank}(N_f - I_n) = n - 1$ [102]. Upon reducing the matrix $(N_f - I_n)$, and if we find that its rank is $n-1$, then f is irreducible and the algorithm halts. So, we may assume that the rank is less than or equal to $n-2$ or equivalently that $r \geq 2$. Now let h be a solution of (3.1). By theorem 3.1.1, h can be expressed as $\sum_{i=1}^{r} a_i f \frac{g_i'}{g_i} \in \mathbb{F}_p[x]$, for some $a_1, ..., a_r \in \mathbb{F}_p$. Let $J(h) = \{1 \leq j \leq r : a_j = 0\}$. One can then show that $\gcd(f, h) = \prod_{j \in J(h)} g_j$, and that:

Theorem 3.1.3 *[102] The probability that a random solution h produces a nontrivial factorisation of f is approximately $\frac{r}{p}$, if $p \gg r$.*

3.1.3 The algorithm

The following algorithm returns a monic nontrivial factor of f (or all factors of f if p and r are not too large). In the former case, all irreducible factors can be found by recursing on the output of the algorithm.

Algorithm 3.1.1 *Input: f a square-free monic polynomial of degree n over $\mathbb{F}_p$ and r the number of its distinct irreducible factors.*
Output: A nontrivial factor of f.

1. Determine the matrix N_f then calculate the rank of $(N_f - I_n)$. If the rank is equal to $n-1$, output f as an irreducible polynomial and halt the algorithm.

2. If $r \geq 2$, solve the linear system of equations

$$(N_f - I_n)\,\mathbf{h}^T = \mathbf{0}$$

over $\mathbb{F}_p$. Each solution $\mathbf{h}$ results in a polynomial h over $\mathbb{F}_p$ whose coefficients are the coordinates of $\mathbf{h}$.

3. Consider a nonzero polynomial h produced in step 2 and calculate $\gcd(f,h)$. Repeat until $\gcd(f,h) \neq 1$ or $\gcd(f,h) \neq f$. Then, $\gcd(f,h)$ is a nontrivial factor of f. If p and r are not too large, then we can consider all p^r polynomials h from step 2, so that $\gcd(f,h)$ yields all monic factors of f (with repetitions if $p > 2$).

3.1.4 Acceleration of Niederreiter's algorithm over the binary field

We now discuss how Niederreiter's algorithm can be extended to deal with the case when f is not square-free and when the finite field is of order $q = 2$. The approach taken here becomes more complicated for fields of order q^t, for $t > 1$, and we refer the reader to [41, 103, 104, 105] for more details.

The algorithm lends itself to major simplifications in the case $p = 2$. Let us consider again the differential equation

$$f^p \left(\frac{h}{f}\right)^{(p-1)} + h^p = 0.$$

For $p = 2$, this simplifies to

$$\begin{aligned} f^2 \left(\frac{h}{f}\right)' + h^2 = 0 &\iff (h'f - f'h) + h^2 = 0 \\ &\iff (fh)' = h^2. \end{aligned}$$

Theorem 3.1.4 *[103] Let $\mathbb{F}_q$ be an arbitrary field of characteristic 2, and $f \in \mathbb{F}_q[x]$ a monic polynomial such that $f = g_1^{e_1} \ldots g_r^{e_r}$, where the g_i's are distinct monic irreducible polynomials and $e_i \geq 1$, for $1 \leq i \leq r$. Let b be a polynomial running through all square-free monic factors of f. Then the polynomials h solving the differential equation*

$$(fh)' = h^2 \tag{3.2}$$

are given by

$$h = \frac{f}{b} b'.$$

Furthermore, different solutions h are determined for different choices of b, and hence the above differential equation has exactly 2^r distinct solutions.

Again, for the sake of continuity in further arguments, we shall replicate the proof from [103]:

Proof: Let b be a monic factor of f. Then

$$b = \prod_{i=1}^{r} g_i^{c_i}, \qquad \text{for } 0 \le c_i \le e_i \text{ and } 1 \le i \le r.$$

For $h = (f/b)b'$, we have

$$\frac{h}{f} = \frac{b'}{b} = \sum_{i=1}^{r} c_i \frac{g_i'}{g_i}.$$

Since $\mathbb{F}_q$ has characteristic 2, the c_i's are either 0 or 1, and hence it suffices to let b range over the square-free monic factors of f only. Let h be a solution to the differential equation $(hf)' = h^2$. We can assume that $h \neq 0$, since the case $h = 0$ is obtained by choosing $b = 1$. Let $a = \gcd(f, h)$. Then we can find b and c such that $f = ab$, $h = ac$, and $\gcd(b, c) = 1$. Now

$$(hf)' = h^2 \iff (f/h)' = 1 \iff (b/c)' = 1 \iff b'c - bc' = c^2.$$

Hence, $c \mid bc'$, and with $\gcd(b, c) = 1$ we get that $c \mid c'$. But this happens only when $c' = 0$. Since we also have $b'c - bc' = c^2$, we must have $c = b'$ or equivalently $h = \frac{f}{b}b'$.

On the other hand, if $h = (f/b)b'$ for some monic square-free factor b of f, then

$$\begin{aligned}
(fh)' &= \left[\left(\tfrac{f^2}{b}\right) b'\right]' \\
&= \left[\left(\tfrac{f}{b}\right)^2 bb'\right]' \\
&= \left[\left(\tfrac{f}{b}\right)^2\right]' bb' + \left(\tfrac{f}{b}\right)^2 (bb')' \\
&= 2\left(\tfrac{f}{b}\right)\left(\tfrac{f}{b}\right)' bb' + \left(\tfrac{f}{b}\right)^2 (b'b' + bb'') \\
&= \left(\tfrac{f}{b}\right)^2 \left((b')^2 + bb''\right) \quad \text{(since all arithmetic is performed modulo 2)} \\
&= \left(\tfrac{fb'}{b}\right)^2 = h^2,
\end{aligned}$$

where here we have used the fact that $g'' = 0$ for all $g \in \mathbb{F}_q[x]$ when $\mathbb{F}_q$ is a field of characteristic 2 (this can be seen as a result of the fact that the only nonzero summands appearing in g' occur at powers x^i where i is divisible by 2). Thus, h satisfies $(fh)' = h^2$.

To show that different choices of b result in distinct solutions h, we proceed as follows. Since b is square-free, this is equivalent to $\gcd(b, b') = 1$, and so for $h = (f/b)b'$ we have

$$\gcd(f, h) = \frac{f}{b}\gcd(b, b') = \frac{f}{b}.$$

Thus, it is clear that different choices of b lead to distinct solutions h.

The above theorem provides an explicit description of elements that solve the differential equation $(fh)' = h^2$, and one can proceed as in the general case for $\mathbb{F} = \mathbb{F}_p$. In particular, we have:

Theorem 3.1.5 *[103] The differential equation* $(fh)' = h^2$ *results in a system of quadratic equations in the unknown coefficients of* h.

As a result of the above theorem, we have:

$$\gcd(f, h) = \frac{f}{b} \gcd(b, b') = \frac{f}{b}$$

where $\gcd(b, b') = 1$ since b is square-free. At this stage, the r irreducible factors can be determined using any of the following strategies:

1. Choose any solution $\mathbf{h}$ of the linear system such that the corresponding polynomial whose coefficients form the row vector $\mathbf{h}$ is not equal to zero or f'. If $\mathbf{h} \neq \mathbf{0}$, then $\gcd(f, \mathbf{h}) \neq f$ and if $\mathbf{h} \neq f'$, then $b \neq f$ and so $\gcd(f, \mathbf{h}) \neq 1$. We can apply the same factorisation again to this non-trivial factor and its complement and call the procedure recursively on the output. This may become inconvenient in practice, as it requires setting up a new matrix and solving the associated system once again. For an input polynomial of degree n, there will be about $O(\log n)$ irreducibles [83, 84, 100], each with multiplicity at most n. In the worst-case analysis, one will have to call the Niederreiter algorithm about $O((\log n)^n)$ times.

2. Determine the 2^r solution polynomials h. The corresponding polynomials $\frac{f}{\gcd(f,h)} = b$ will then cover all 2^r monic factors of the square-free part $g_1 ... g_r$ of f and in particular, all the irreducible factors of f.

In this context, R. Göttfert introduced a third strategy leading to a polynomial time algorithm for extracting all irreducible factors of f [59]. Perhaps more striking is that Göttfert restricted his attention to the set of basis elements $\{h_1, ..., h_r\}$ spanning the solution set of $(fh)' = h^2$, rather than an arbitrary solution of the linear system. In [59], Göttfert showed how this can be used together with at most r^2 gcd and division operations to obtain a complete factorisation. We shall recall the algorithm briefly and refer the reader to the original paper for details and proofs of the forthcoming results.

Consider the set of basis elements $\{h_1, ..., h_r\}$ of the Niederreiter linear system and the corresponding polynomials

$$b_i = \frac{f}{\gcd(f, h_i)} \in \mathbb{F}_q[x], \quad \text{for } i = 1, ..., r$$

representing monic square-free factors of f. Those factors are then listed in a collection of at most r rows as follows. The first row contains only b_1. The second row consists of at most three polynomials, specifically, the non-constant polynomials among

$$\gcd(b_2, b_1), \quad \frac{b_1}{\gcd(b_2, b_1)}, \quad \frac{b_2}{\gcd(b_2, b_1)}.$$

The third row consists of

$$\begin{array}{l} \gcd(b_3, r_1), \quad \dfrac{r_1}{\gcd(b_3, r_1)}, \\ \gcd(b_3, r_2), \quad \dfrac{r_2}{\gcd(b_3, r_2)}, \\ \gcd(b_3, r_3), \quad \dfrac{r_3}{\gcd(b_3, r_3)}, \\ \dfrac{b_3}{\gcd(b_3, r_1)\gcd(b_3, r_2)\gcd(b_3, r_3)} \end{array}$$

where

$$r_1 = \gcd(b_2, b_1), \quad r_2 = \frac{b_1}{\gcd(b_2, b_1)} \quad \text{and } r_3 = \frac{b_2}{\gcd(b_2, b_1)}.$$

In general, the polynomials of row k, for $k = 2, ..., r$, consist of the non-constant polynomials among

$$d_1, \frac{r_1}{d_1}, ..., d_s, \frac{r_s}{d_s}, \frac{b_k}{d_1 ... d_s},$$

where $r_1, ..., r_s$ are the polynomials in row $k-1$ and $d_j = \gcd(b_k, r_j)$, for $j = 1, ..., s$.

Theorem 3.1.6 *[59] Any polynomial row constructed in this way has the following properties:*

i. The polynomials in any row are pairwise relatively prime monic square-free factors of f.

ii. The polynomial b_k appears in row k, either in its original form or split up into some non-trivial factors.

iii. Every polynomial in row $k-1$ also appears in row k, either in its original form or split up into two non-trivial factors.

Theorem 3.1.7 *[59] The irreducible monic square-free factors of f are determined once a row containing r polynomials has been reached.*

The following theorem shows that this procedure always results in a row with r elements:

Theorem 3.1.8 *[59] The row of index at most r contains the polynomials $g_1 ... g_r$, the distinct monic irreducible factors of f.*

Summarising all, Göttfert's algorithm takes up the following form :

Algorithm 3.1.2 *[59]*
Input: A polynomial f of degree n over $\mathbb{F}_q$ where $q = 2^t$, for some $t \geq 1$.
Output: The r irreducible factors of f.

1. Set up the $n \times n$ matrix $N_f - I$. By a rank computation, determine the number of irreducible factors of f. If this is equal to 1, output f as an irreducible polynomial and halt the algorithm.

2. Determine a basis $\{h_1, ..., h_r\}$ of the solution space of the system

$$(N_f - I_n)\mathbf{h}^T = \mathbf{0}. \tag{3.3}$$

3. Compute $b_1, ..., b_r$ *defined as*

$$b_i = \frac{f}{\gcd(f, h_i)} \text{ for } i = 1, ..., r.$$

4. Set up a table of polynomials of at most r *rows as has been described earlier. The rows are set up inductively and non-constant polynomials are removed from each row. Stop the process when a row containing* r *non-constant polynomials is obtained. This may be row* r *or any other earlier one. The polynomials found on that row are the* r *irreducible factors of* f*.*

Since at most r rows have to be set up, each containing at most r polynomials, we need at most r^2 gcd operations and at most r^2 division operations to find all irreducibles. Over $\mathbb{F}_2$, this brings the total cost of such computations to $O(r^2 M(n) \log n)$ bit operations, where $\log n$ is the binary logarithm of n, $M(n)$ is the time to multiply (or divide) two polynomials of degree at most n over $\mathbb{F}_2$, and $O(M(n) \log n)$ is the time to perform the gcd of two such polynomials over $\mathbb{F}_2$.

Theorem 3.1.9 *[59] Using Göttfert's acceleration of Niederreiter's algorithm, a polynomial* f *of degree* n *over* $\mathbb{F}_2$ *can be factorised using* $O(n^\omega + r^2 M(n) \log n)$ *operations in* $\mathbb{F}_2$*.*

3.2 Bivariate factorisation

In the multivariate case, algorithms have been concerned with two types of factorisations over finite fields: *Rational* factorisation into irreducible factors over the ground field, and *absolute* factorisation of the input polynomial into irreducible factors over the algebraic closure of the ground field. In the former case, Lenstra, Lenstra and Lovász [91] gave the first polynomial time algorithm for factoring univariate polynomials over rational numbers, through the so called LLL lattice basis reduction. This was used later on by A. K. Lenstra [88, 89, 90], Chistov [23, 24, 25], Grigoryev [63], and Chistov and Grigoryev [62], to obtain polynomial time algorithms for multivariate polynomials over various fields, including finite fields. Kaltofen [75], and von zur Gathen and Kaltofen [51], introduced polynomial time algorithms using Newton approximation for multivariate factorisation over rational numbers and finite fields. Algorithms for factoring multivariate polynomials using a black box representation were given by Kaltofen and Trager [78], Diaz and Kaltofen [32], and Rubinfeld and Zippel [113]. The work in [78] and [113] also used modular interpolation to reduce the problem to univariate factorisation. Bernardin developed a polynomial time extension of Yun's algorithm [133] for square-free factorisation of multivariate polynomials over finite fields [11].

Among the well known algorithms for absolute factorisation are the following: Duval [34] used special function spaces based on algebraic geometry to obtain an algorithm that is only conjectured to run in polynomial time. Kaltofen proposed algorithms for absolute factorisation also using Newton approximation [77, 80]. Gao obtained an algorithm for multivariate polynomial factorisation over any field of characteristic zero or of relatively large characteristic that is based on a simple partial differential equation [44]. All the above algorithms run in polynomial time or are conjectured so. For instance, given a bivariate polynomial $f \in \mathbb{F}_q[x, y]$ of total degree n, where the input size is of the order $N = O(n^2)$, Lenstra's algorithm for rational factorisation requires $O(N^4)$ field operations [89], ignoring logarithmic factors. The work of von zur Gathen

and Kaltofen based on Newton approximation requires $O(N^6)$ field operations [51]. When applied to finite fields, Gao's algorithm using PDE's requires the characteristic of the field to be at least $6mn$, where m and n denote the upper bounds on the degrees in x and y respectively. In that case, Gao's algorithm has a cost of $O(N^{2.5})$ field operations [44].

Another special class of rational multivariate factorisation are Hensel lifting based techniques which have been shown to be efficient in practice [13, 101, 128, 131]. Despite its worst-case exponential running time, Gao and Lauder showed that the application of such algorithms to bivariate polynomials over finite fields has an average running time that is almost linear in the input size [46], which explains why they are fast in practice. Since this relates strongly to the factorisation via polytopes algorithm in Chapter 6, we shall dedicate the rest of the discussion to a summary of Hensel lifting.

3.2.1 Hensel lifting for bivariate polynomials

We recall briefly the main ideas behind this approach following the bivariate version in [46]. Let $\mathbb{F}_q$ denote a finite field of order q and let $T(n,q)$ denote the set of all polynomials in $\mathbb{F}_q[x,y]$ of total degree n that are monic in x and have degree n in x. As shown in [46], this model of polynomials on which the Hensel lifting algorithm will be based is not trivial because any polynomial of total degree n can be transformed into a polynomial in $T(n,q)$ that has the same factorisation pattern. In particular, let $h(y) = \sum_{i=0}^{n} c_i y^i$ where $\sum_{i=0}^{n} c_i x^{n-i} y^i$ is the homogeneous part of f of degree n (in other words, each $c_{j'}$ represents the coefficient of a term $a_{(j,j')} x^j y^{j'}$ in f such that $j + j' = n$). Then $g = f(x, y + \alpha x)$ still has total degree n and the coefficient of x^n in g is $h(\alpha)$. Since h is nonzero and has degree at most n, we only have to choose $\alpha \in \mathbb{F}_q$ such that $h(\alpha) \neq 0$. When $h(\alpha) \neq 0$, g can be made monic in x, and hence can be viewed as belonging to $T(n,q)$. Obviously, the factors of f are easily obtained from those of g by the inverse transformation (say $f = g(x, y - \alpha x)$) [46].

Let $f \in T(n,q)$ and suppose that $f = gh$, where f, g and h are all lying in $\mathbb{F}_q[x,y]$. Let $n = \deg(f)$, $r = \deg(g)$, and $s = \deg(h)$, so that $n = r + s$. Write

$$f = \sum_{k=0}^{n} f_k y^k, \quad g = \sum_{k=0}^{r} g_k y^k \quad \text{and } h = \sum_{k=0}^{s} h_k y^k,$$

where $\deg(g_k) \leq r - k$ and $\deg(h_k) \leq s - k$, for $k = 0, ..., n$ (here we consider g_k and h_k to be zero whenever $r - k$ and $s - k$ are negative). Equating the coefficients of y^k, for $k = 0, ..., n$, on both sides of $f = gh$, we see that $f_0 = g_0 h_0$ and for $k \geq 1$:

$$f_k = \sum_{i=0}^{k} g_i h_{k-i},$$

or

$$g_0 h_k + g_k h_0 = f_k - \sum_{i=1}^{k-1} g_i h_{k-i}. \tag{3.4}$$

Let $d = \gcd(g_0, h_0)$ with u and v chosen so that

$$u g_0 + v h_0 = d \tag{3.5}$$

and $\deg(u) < \deg(h_0)$, $\deg(v) < \deg(g_0)$. Then d divides $g_0h_k + g_kh_0 = f_k - \sum_{i=1}^{k-1} g_ih_{k-i}$. By (3.4) we have

$$g_kh_0 \equiv (f_k - \sum_{i=1}^{k-1} g_ih_{k-i}) \bmod g_0,$$

and by (3.5) we have

$$vh_0 \equiv d \bmod g_0.$$

Thus,

$$g_kvh_0 \equiv v(f_k - \sum_{i=1}^{k-1} g_ih_{k-i}) \bmod g_0$$

or

$$g_kd \equiv v(f_k - \sum_{i=1}^{k-1} g_ih_{k-i}) \bmod g_0.$$

Since d divides $f_k - \sum_{i=1}^{k-1} g_ih_{k-i}$, we can find $w_k \in \mathbb{F}_q[x]$ such that

$$g_k = v\frac{f_k - \sum_{i=1}^{k-1} g_ih_{k-i}}{d} + w_k\frac{g_0}{d}. \tag{3.6}$$

On the other hand, we have

$$\begin{aligned} g_0h_k &= f_k - \textstyle\sum_{i=1}^{k-1} g_ih_{k-i} - g_kh_0 \\ &= f_k - \textstyle\sum_{i=1}^{k-1} g_ih_{k-i} - v\frac{h_0(f_k - \sum_{i=1}^{k-1} g_ih_{k-i})}{d} - w_k\frac{g_0h_0}{d} \\ &= (d - vh_0)\frac{f_k - \sum_{i=1}^{k-1} g_ih_{k-i}}{d} - w_k\frac{g_0h_0}{d} \end{aligned}$$

so that

$$h_k = u\frac{f_k - \sum_{i=1}^{k-1} g_ih_{k-i}}{d} - w_k\frac{h_0}{d}. \tag{3.7}$$

Turning this observation around, assume we have been given a polynomial $f \in \mathbb{F}_q[x, y]$ of total degree n and a factorisation of the reduction of f modulo y given as $f_0 = g_0h_0$, where $f = \sum_{k=0}^{n} f_ky^k$ and $g_0, h_0 \in \mathbb{F}_q[x]$. Assume further that $f \in T(n, q)$ so that $\deg(f_0) = n$. Let $r = \deg(g_0)$ and $s = \deg(h_0)$, so that $n = r + s$. The question one seeks now is whether it would be possible to use Equations (3.6) and (3.7) to define a sequence of polynomials $\{g_k\}_{k\geq 0}$ and $\{h_k\}_{k\geq 0}$ such that $g = \sum_{0\leq k\leq n} g_ky^k$, $h = \sum_{0\leq k\leq n} h_ky^k$, and $f = gh$, under the restrictions $\deg(g_k) \leq r - k$, $\deg(h_k) \leq s - k$, $g_k = 0$ if $r - k < 0$ and $h_k = 0$ if $s - k < 0$. It turns out that this is possible provided at each stage w_k is chosen so that d divides the polynomials $f_k - \sum_{i=1}^{k-1} g_ih_{k-i}$. If $d \neq 1$, then the choice we make of w_k may not be unique, resulting in exponentially many choices for g_k's and h_k's. If $d = 1$, however, there will be at most one way of doing this, and the equations (3.6) and (3.7) uniquely determine g_k and h_k, for $k \geq 1$, as

$$g_k \equiv v(f_k - \sum_{i=1}^{k-1} g_ih_{k-i}) \bmod g_0 \tag{3.8}$$

$$h_k \equiv u(f_k - \sum_{i=1}^{k-1} g_i h_{k-i}) \bmod h_0 \tag{3.9}$$

and the lifting can be carried out uniquely as high as one wishes, after checking whether $\deg(g_k) \leq r-k$ and $\deg(h_k) \leq s-k$. In the parlance of Chapter 6, the Newton polytope of f as given above lies in a triangle with vertices $(n,0),(0,n),(0,0)$, and lifting is initiated along the horizontal edge, since all terms in f_0 have a zero exponent in the variable y.

The algorithm

For $n \geq 1$, let $M(n,q) \subseteq T(n,q)$ denote the subset of all polynomials whose reduction modulo y is square-free. The condition in the previous section requiring that $d = \gcd(g_0, h_0) = 1$ shows that Hensel lifting works for all polynomials in $M(n,q)$. We shall first present a version from [46] which accepts only polynomials in $M(n,q)$. With slight modifications, this can later be used to factor polynomials in $T(n,q)$.

Algorithm 3.2.1 *(Hensel Factorisation)*
Input: A polynomial $f = \sum_{k=0}^{n} f_k y^k$ *in* $M(n,q)$, *where* $f_k \in \mathbb{F}_q[x]$.
Output: All monic factors of f *with total degree between* 1 *and* $\lfloor n/2 \rfloor$.

Step 1: Use a univariate polynomial factorisation algorithm to factor f_0, *a square-free polynomial. If* f_0 *is irreducible, then halt the algorithm.*

Step 2: List all pairs (g_0, h_0) *of monic factors of* f_0 *such that* $f_0 = g_0 h_0$, *and* $\deg(g_0) < \deg(h_0)$, *say. Let* $r = \deg(g_0)$ *(so that* $1 \leq r \leq \lfloor n/2 \rfloor$*). For each pair* (g_0, h_0), *repeat Steps 3-5:*

Step 3: Compute polynomials u *and* v *with* $ug_0 + vh_0 = 1$ *and* $\deg(u) < \deg(h_0)$, $\deg(v) < \deg(g_0)$.

Step 4: For $k = 1, ..., r$, *compute*

$$g_k \equiv v(f_k - \sum_{i=1}^{k-1} g_i h_{k-i}) \bmod g_0,$$

and

$$h_k \equiv u(f_k - \sum_{i=1}^{k-1} g_i h_{k-i}) \bmod h_0.$$

In the case that $r \geq k$ *check whether* $\deg(g_k) \leq r-k$ *and in the case that* $k > r$ *check whether* $g_k = 0$. *Also, in the case that* $s \geq k$ *check whether* $\deg(h_k) \leq s-k$ *and in the case that* $k > s$ *check whether* $h_k = 0$. *If any of those two checkings fail, halt the computation for this pair of* (g_0, h_0).

Step 5: Check whether $g = \sum_{k=0}^{r} g_k y^k$ *divides* f. *If so, then output* g.

Correctness of the above algorithm follows easily from the preceding discussion. The worst-case running time is clearly dependent on the total number of pairs of monic factors (g_0, h_0), which is exponential in the total number of irreducible factors of f_0. The average such number for a univariate polynomial of degree n is about $O(\log n)$ [83, 84, 100]. For each pair (g_0, h_0),

the inner-most computations of the algorithm is dominated by $O(n)$ polynomial divisions in $\mathbb{F}_q$. If $d(n,q)$ denotes the bound on the worst-case number of $\mathbb{F}_q$ operations required to factor univariate polynomials of degree n over $\mathbb{F}_q$, the above algorithm has a worst-case complexity of the order $O(d(n,q) + 2^{\log n} n^3)$ field operations, assuming classical polynomial arithmetic.

In general though, $f_0 = f \bmod y$ may not be square-free in $\mathbb{F}_q[x]$, and hence, cannot be factorised using the above version of Hensel lifting. To this end, a randomisation technique is introduced in [46] addressing square-free polynomials f in $\mathbb{F}_q[x,y]$ whose reduction modulo y is not square-free. In particular, it was shown the following:

Lemma 3.2.1 *[46] Let S be a subset of $\mathbb{F}_q$ and $f \in T(n,q)$ square-free. For random $\beta \in S$, we have $g = f(x, y+\beta) \in M(n,q)$ with probability at least $1 - n(2n-1)/|S|$.*

Thus, if $q > 4n^2$, one can take $S = \mathbb{F}_q$ and so the probability in the above lemma will be at least $1/2$. If q is small, one needs to go to an extension of $\mathbb{F}_q$ of sufficient size and factor f over there, then combine the factors to go down to $\mathbb{F}_q$. For more details on this and on Hensel lifting techniques in general, we refer the reader to [29, 101, 128, 130, 131, 134].

3.3 Polynomials and Newton polytopes

For an extensive review of the theory of convex polytopes we refer the reader to [64]. Let $\mathbb{R}$ denote the field of real numbers and $\mathbb{R}^n$ the Euclidean n-space. A *convex polytope* in $\mathbb{R}^n$ is the smallest convex set containing a given nonempty finite set of points in $\mathbb{R}^n$. A point of a polytope is a *vertex* if it does not belong to the interior of any line segment in the polytope. A hyperplane cuts the polytope if both of the open half-spaces determined by it contain points of the polytope. A hyperplane which does not cut a polytope, but has a non-empty intersection with it is called a *supporting hyperplane*. The intersection of a supporting hyperplane and a polytope is called a *proper* face, and the union of all proper faces is the *boundary*. 1-dimensional faces are *edges*. By *proper* we simply refer to the non-trivial case when the dimension of the face is less than the dimension of the polytope.

Let $\mathbb{F}[X_1, X_2, \ldots, X_n]$ be the ring of polynomials in n variables over an arbitrary field $\mathbb{F}$. For any vector $e = (e_1, \ldots, e_n)$ of non-negative integers, define $X^e := X_1^{e_1} \cdots X_n^{e_n}$. Let $f \in \mathbb{F}[X_1, \ldots, X_n]$ be given by

$$f := \sum_e a_e X^e$$

where the sum is over finitely many points e in $\mathbb{N}^n$ called *support vectors* of f, and $a_e \in \mathbb{F}$. Let $Supp(f)$ denote the set of all its support vectors. The *total degree* of f when f is not a constant is defined to be the maximum value of $\sum_{1 \le i \le n} e_i$ over all $(e_1, ..., e_n) \in Supp(f)$. The Newton polytope of f, denoted by $\mathrm{Newt}(f)$, is the polytope in $\mathbb{R}^n$ obtained as the convex hull of all exponents e for which the corresponding coefficient a_e is nonzero. It has integer vertices, since all the e are integral points. We call such polytopes *integral*. Given two polytopes Q and R, their *Minkowski sum* is defined to be the set

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

When Q and R are integral polytopes, so is $Q + R$. If we can write an integral polytope P as a Minkowski sum $Q + R$ for integral polytopes Q and R then we call this an *integral decomposition*.

The decomposition is *trivial* if Q or R has only one point, and P is *integrally decomposable* if it has at least one non-trivial decomposition. If a polytope has no non-trivial decompositions then it is *integrally indecomposable*. The Minkowski sum of two convex polytopes is also a convex polytope.

The following result, demonstrated in [36, 64, 117], describes how faces decompose in a Minkowski sum of polytopes:

Lemma 3.3.1 *Let Q and R be polytopes in $\mathbb{R}^n$ and $P = Q + R$.*

1. *Each face of P is a Minowski sum of unique faces of Q and R.*
2. *Let P_1 be any face of P and $v_0, ..., v_{m-1}$ be all of its vertices. Suppose that $v_i = q_i + r_i$ for some $q_i \in Q$, $r_i \in R$, and $i = 0, ..., m-1$. Let Q_1 and R_1 denote the convex hulls of $\{q_0, ..., q_{m-1}\}$ and $\{r_0, ..., r_{m-1}\}$, respectively. Then Q_1 and R_1 are faces of Q and R, respectively, and $P_1 = Q_1 + R_1$.*

A polytope of dimension 2 is a *polygon*, where the only proper faces are edges and vertices. The above lemma can then be rephrased as follows:

Corollary 3.3.1 *[45] Let P, Q and R be convex polygons in $\mathbb{R}^2$ with $P = Q + R$. Then every edge of P decomposes uniquely as the sum of an edge of Q and an edge of R, possibly one of them being a point. Conversely, any edge of Q or R is a summand of exactly one edge of P.*

Let P be a convex polygon in $\mathbb{R}^2$, and let $v_0, ..., v_{m-1}$ denote its vertices ordered cyclically in a counter-clockwise direction. The edges of P are vectors of the form $E_i = v_{i+1} - v_i = (a_i, b_i)$, for $0 \leq i \leq m-1$, where $a_i, b_i \in \mathbb{Z}$ and the indices are taken modulo m. A vector $v = (a, b) \in \mathbb{Z}^2$ is called a primitive vector if $\gcd(a, b) = 1$. If $n_i = \gcd(a_i, b_i)$ and $e_i = (a_i/n_i, b_i/n_i)$, then $E_i = n_i e_i$, where e_i is a primitive vector, for $0 \leq i \leq m-1$. The sequence of vectors $\{n_i e_i\}_{0 \leq i \leq m-1}$ is called the edge sequence or polygonal sequence and uniquely identifies the polygon up to translation determined by v_0. Since the boundary of a polygon forms a closed path, we have that $\sum_{0 \leq i \leq m-1} n_i e_i = (0, 0)$. For convenience, an edge sequence can be identified with that obtained by extending the sequence by inserting an arbitrary number of zero vectors, and so we can assume that the edge sequence of a summand of P has the same length as that of P. The following lemma gives an explicit description of edge sequences describing all possible summands of a given integral polygon.

Lemma 3.3.2 *[45] Let P be a polygon with edge sequence $\{n_i e_i\}_{0 \leq i \leq m-1}$ where $e_i \in \mathbb{Z}^2$ are primitive vectors. Then an integral polygon is a summand of P iff its edge sequence is of the form $\{k_i e_i\}_{0 \leq i \leq m-1}$, $0 \leq k_i \leq n_i$, with $\sum_{0 \leq i \leq m-1} k_i e_i = (0, 0)$.*

For proof, see [45].

We conclude with a final useful result describing the notion of the length of an edge of an integral polygon, designating the number of integral points falling on the edge:

Lemma 3.3.3 *[43] Given $\mathbf{v_0}$ and $\mathbf{v_1}$ two distinct integral points in $\mathbb{R}^2$, the number of integral points on the line segment $\mathbf{v_0 v_1}$, including $\mathbf{v_0}$ and $\mathbf{v_1}$, is equal to $\gcd(\mathbf{v_0} - \mathbf{v_1})$.*

For proof, see [43].

3.3.1 Indecomposable polytopes and absolute irreducibility

Recall from the above discussion that f is *absolutely irreducible* over $\mathbb{F}$ if it has no non-trivial factors over $\overline{\mathbb{F}}$, the algebraic closure of $\mathbb{F}$. Absolute irreducibility forms a stronger irreducibility criterion than rational irreducibility because f has no irreducible factors over $\mathbb{F}$ if it is absolutely irreducible. The multivariate factorisation algorithms presented earlier can all serve as irreducibility tests. The following theorem is at the heart of a different kind of absolute irreducibility criterion:

Theorem 3.3.1 (Ostrowski) *[108] Let $f, g, h \in \mathbb{F}[X_1, \ldots, X_n]$. If $f = gh$ then* $\mathrm{Newt}(f) = \mathrm{Newt}(g) + \mathrm{Newt}(h)$.

Corollary 3.3.2 (Irreducibility Criterion) *[43] Let $f \in \mathbb{F}[X_1, ..., X_n]$ with f not divisible by any non-constant X_i, for $1 \leq i \leq n$. If* $\mathrm{Newt}(f)$ *is not integrally decomposable, then f is absolutely irreducible.*

For proof, see [43].

We conclude this section with a brief discussion on a relevant concept of *homothetic* decomposability [42, 43, 64, 96, 99, 115, 121, 122] affecting integral decomposability. The relevance of this will become clearer towards the end of this section.

Definition 3.3.1 *Let P and Q be polytopes in $\mathbb{R}^n$ (not necessarily integral). We say that Q is homothetic to P if there exists a real number $t \geq 0$ and a vector $a \in \mathbb{R}^n$ such that*

$$Q = tP + a = \{tb + a : b \in P\}.$$

Definition 3.3.2 *A polytope P is called homothetically indecomposable whenever $P = P_1 + P_2$ for two polytopes P_1 and P_2 implies that P_1 or P_2 is homothetic to P. Otherwise, P is called homothetically decomposable.*

The following proposition outlines the relationship between homothetic and integral polytope indecomposability:

Proposition 3.3.1 *[45] Let Q be an integral polytope in $\mathbb{R}^n$ with vertices v_i, where $0 \leq i \leq k$. If Q is homothetically indecomposable and* $\gcd(v_0 - v_1, ..., v_0 - v_k) = 1$, *then Q is integrally indecomposable.*

For proof, see [45]

Remark: Note that if Q is integrally indecomposable, then Q is homothetically indecomposable. To see this, write $Q = T + S$ for some integral polytopes T and S. Then T, say, must be a trivial summand consisting of one point, v, in which case $T = 0 \cdot Q + v$.

3.3.2 Testing indecomposability of polytopes

Polygons

Following the discussion above, testing absolute irreducibility of multivariate polynomials over arbitrary fields is thus reduced to deciding whether a given polytope is integrally decomposable. Assuming that the polytope is given as a list of its vertices, the input size of this problem is the

length of the binary representation of the coordinates of the vertices. It was established in [45] that deciding polygon indecomposability (and hence indecomposability of higher dimensional polytopes) is NP-complete, and thus it remains an open problem to develop an efficient, polynomial time, deterministic or even randomised algorithm for testing general integral polytopes for indecomposability. Gao and Lauder developed a pseudo-polynomial time algorithm (see [48]) with a run-time complexity that is polynomial in the lengths of the sides of the polygon, rather than in the logarithm of the lengths [45]. We recall the algorithm and refer the reader to the original paper [45] for more details.

Algorithm 3.3.1 *Input: The edge sequence* $\{n_i e_i\}_{0 \leq i \leq m-1}$ *of an integral convex polygon* P *starting at a vertex* v_0 *where* $e_i \in \mathbb{Z}^2$ *are primitive vectors.*
Output: Whether P *is decomposable.*

Step 1: Compute the set IP *of all the integral points in* P*, and set* $A_i = \emptyset$*, for* $i = -1, ..., m-1$.

Step 2: For $i = 0, ..., m-2$*, compute the set of points in* IP *that are reachable via the vectors* $e_0, ..., e_i$*:*

2.1: For each $k = 1, ..., n_i$*, if* $v_0 + ke_i \in IP$*, then add it to* A_i*;*
2.2: For each $u \in A_{i-1}$ *and* $k = 0, ..., n_i$*, if* $u + ke_i \in IP$*, then add it to* A_i.

Step 3: Compute the last set A_{m-1}*: For each* $u \in A_{m-2}$ *and* $k = 0, ..., n_{m-1} - 1$*, if* $u + ke_{m-1} \in IP$*, add it to* A_{m-1}.

Step 4: Return "Decomposable" if $v_0 \in A_{m-1}$ *and "Indecomposable" otherwise.*

Theorem 3.3.2 *[45] The above algorithm decides decomposability correctly in* $O(tmN)$ *vector operations where* t *is the number of integral points in* P*,* m *is the number of its edges, and* N *is the maximum number of integral points on an edge.*

For a detailed proof of the above theorem we refer the reader to [45] and we sketch only the basic idea for the sake of clarity. All the points in A_{m-1} that are points in IP reachable via the vectors $e_0, ..., e_{m-1}$ are constructed to be of the form $v_0 + \sum_{i=0}^{m-1} k_i e_i$, $0 \leq k_i \leq n_i$. If one of the points in A_{m-1} is equal to v_0, then $\sum_{i=0}^{m-1} k_i e_i = (0, 0)$, and so the sequence $\{k_i e_i\}$ forms the edge sequence of an integral summand Q of P. On the other hand, it can be easily shown that the edge sequence of every proper integral summand of P will be detected by the above algorithm.

Higher dimensional polytopes

Carrying this work further to deal with general polytopes in $\mathbb{R}^n$, Gao and Lauder integrated the above algorithm into a heuristic randomised test for higher dimensional polytope indecomposability. Their approach relies on the use of random integral linear maps which project a given polytope into a polygon in a plane. If the projected polygon is indecomposable, and under certain conditions presented in the lemma below, one deduces that the original polytope is indecomposable.

Lemma 3.3.4 *[45] Let* P *be any integral polytope in* $\mathbb{R}^n$ *and let* $\pi : \mathbb{R}^n \mapsto \mathbb{R}^m$ *be any integral linear map which maps integral points in* $\mathbb{R}^n$ *to integral points in* $\mathbb{R}^m$*. If* $\pi(P)$ *is integrally*

indecomposable, and each vertex of $\pi(P)$ has only one pre-image in P, then P must be integrally indecomposable.

For proof, see [45].

Corollary 3.3.3 *[45] Let Q be any integrally indecomposable polytope in $\mathbb{R}^m$ and $\pi : \mathbb{R}^n \mapsto \mathbb{R}^m$ any integral linear map. Let S be any set of integral points in $\pi^{-1}(Q)$ having exactly one point in $\pi^{-1}(v)$ for each vertex v of Q. Then the polytope in $\mathbb{R}^n$ consisting of the convex hull of all points in S is integrally indecomposable.*

The above results can be used in a multivariate polynomial absolute irreducibility test as follows. Given a non-constant polynomial $f \in \mathbb{F}[X_1, ..., X_n]$, let $S = Supp(f)$, and let P denote the convex hull of the finite set of points in S. We need to decide whether P is integrally indecomposable. Note that P need not be computed at this stage, since the points of S that are mapped to vertices of a polygon by a random integral linear map will be vertices of P, provided each vertex of the polygon has only one pre-image in S.

To describe a suitable projection, we write the points of S in $\mathbb{R}^n$ as column vectors. If S has c points, then it can be represented as an $n \times c$ matrix, where each column stands for a point. For convenience, we shall also denote the matrix by S. Its columns are distinct since the support vectors of f are so. Let $u, v \in \mathbb{R}^n$ be two integral points; then for any $w \in \mathbb{R}^n$, the matrix-vector product $(u, v)^T w$ represents a point in $\mathbb{R}^2$. This defines an integral projection $\pi : \mathbb{R}^n \mapsto \mathbb{R}^2$ where $(u, v)^T S$ is the image of S under π in $\mathbb{R}^2$. The polygon defined by the convex hull of the points in this image is called the *shadow* of P. The next statement is a special case of Lemma 2.9 in [44] and determines how likely it is that the projection is injective on the set S, where elements of $Supp(f)$ are viewed as vectors with entries from $\mathbb{Q}$:

Lemma 3.3.5 *[44] Let S be an $n \times c$ matrix over $\mathbb{Q}$ with no repeated columns, and let K be any finite subset of cardinality k of $\mathbb{Z}$. Choose $u_i \in K$ randomly and independently, for $1 \leq i \leq n$, and let*

$$(a_1, ..., a_c) = (u_1, ..., u_n)S.$$

Then the entries $a_1, ..., a_c$ are all distinct with probability at least $1 - \frac{c(c-1)}{2k}$.

The above lemma can be used to establish a lower bound on the probability that a randomly chosen linear integral map satisfies the conditions in Lemma 3.3.4 above. In particular, if we choose $K = \{-c^2, ..., -1, 0, 1, .., c^2\}$, then K has $k = 2c^2 + 1$ integers. If we further choose the entries of u and v from K randomly and independently, then the points in $(u, v)^T S$ are distinct with probability at least $3/4$. In this case, each vertex of the shadow has only one pre-image with the same probability, which can be increased arbitrarily close to 1 if one increases the size of the set K [45]. The polytope decomposability test (and hence the multivariate absolute irreducibility test) is now as follows:

Algorithm 3.3.2 *[45]*
Input: $f \in \mathbb{F}[X_1, ..., X_n]$ with no non-constant monomial factors, and S_f the set of exponent vectors of nonzero terms of f of cardinality c.
Output: Absolutely irreducible or Failure, where the latter case means that decomposability of

$conv(S_f)$ (and hence absolute irreducibility of f) is not decided.

Step 1: Re-arrange the points in S_f as an $n \times c$ matrix S. If $n = 2$, let A be the trivial projection and go to Step 4. Else, choose positive integers b and e. Let $M(b)$ denote the set of all $2 \times n$ matrices with integer coefficients bounded in absolute value by b. Repeat Steps 2-4 up to e times.

Step 2: Select a matrix A uniformly at random from $M(b)$ and compute the set of points in $\mathbb{R}^2$ defined by $A(S) := \{As | s \in S\}$.

Step 3: Compute the convex hull, $conv(A(S))$, of $A(S)$ and check that each vertex of $conv(A(S))$ has only one pre-image in S under the projection A. If this condition is not met, return to Step 2.

Step 4: Call Algorithm 3.3.1 above using the edge sequence of $conv(A(S))$. If this polygon is integrally indecomposable, output "Absolutely Irreducible" and halt. Else, if $n > 2$, return to Step 2. Else, if $n = 2$, output "Failure".

Step 5: Output "Failure".

Theorem 3.3.3 *[47] Algorithm 3.3.2 works correctly and requires at most*

$$eO(((nbd)^3 + c(c+n)) \log^2(nbd))$$

binary operations and $O((nbd)^2 \log(nbd))$ bits of storage for a polynomial in n variables, with c nonzero terms, and degree at most d in each variable. If f has no more that $c = O(nd)$ nonzero terms, the run-time becomes cubic in the total degree of the input polynomial.

For proof, see [47].

Though promisingly efficient for polynomials whose number of terms is not much greater than their total degree, the above method is still considered a heuristic for the following reasons: Although the probability that the condition in Step 3 is satisfied can be determined, it still needs to be determined how likely it is that the algorithm will show indecomposability if $\text{Newt}(f)$ is indecomposable, since it is possible that there are indecomposable polytopes whose shadow polygons are always decomposable [45]. On the other hand, since it has been proven that most polytopes in $\mathbb{R}^n$, for $n \geq 3$, are homothetically indecomposable [117], a direct consequence of Proposition 3.3.1 is that, most random integral polytopes may be expected to be indecomposable. Algorithm 3.3.2 may detect these quickly in most of the cases, and hence should be particularly effective for random sparse polynomials.

3.3.3 Constructing convex hulls in two dimensions

We now conclude with a discussion of a fast algorithm for computing convex hulls in two dimensions, based on the pioneering work of R. Graham [60], who gave the first $O(n \log n)$ algorithm for computing the hull of n points in the plane. We shall give a brief description and refer the reader to the comprehensive texts in computational geometry [35, 107].

The input to the convex hull algorithm will be a set S of n arbitrary points in the plane, and the output we seek will be a subset of these points representing *extreme* points or vertices ordered in a counter-clockwise direction around a chosen pivot. By vertices we refer to those points of the hull at which the interior angle is strictly convex (less than π). Also, a point is

extreme if and only if there exists a line through that point which otherwise does not touch the convex hull. Alternatively, a point is non-extreme if and only if it is inside some triangle whose vertices are points of S and it is not itself a corner of that triangle. An edge of a convex hull is also called extreme if every point of S is on or to one side of the line determined by the edge. We shall do this by treating the edge as directed, specifying the left side of a directed edge to be the "inside". As such, a directed edge is not extreme if there exists some point that is not left of it or on it.

The above definition of extreme points and edges will be crucial to our understanding of the simple Graham's algorithm. In particular, the convex hull of points of S will be constructed successively in a stack of points, each representing an extreme point. The stack is constructed using a subset of S representing sorted points around a chosen pivot. The sorting rule is as follows. The pivot, say p_0, is chosen as the lowest rightmost point in S, which is clearly on the hull. The remaining $n-1$ points are then sorted around the pivot, according to "leftedness" from p_0, or according to increasing values of their counter-clockwise angles from the horizontal ray emanating from p_0. If there exist two points forming the same angle with p_0, we define a to be less than b if the distance from a to p_0, defined by the euclidean distance $|a-p_0|$, is less than $|b-p_0|$. In that case, point a is deleted, since it belongs strictly to the interior of the hull. Assume that the number of sorted points (after deletion) is $s \leq n$, and let $p_0, ..., p_{s-1}$ denote the ordered set of points around the pivot. The stack is now built iteratively as follows. As indicated above, the first point is the pivot, since it belongs to the hull. The second point is p_1, since it forms an extreme angle with p_0 (no point of the hull is to the right of the directed edge p_0p_1). The rest of the points are then processed in their sorted order incrementally around the set. At any step, the hull will be correct for the points examined so far, but newly added points may cause earlier decisions to be reverted. To illustrate, suppose that we wish to examine whether p_2 belongs to the hull. Since the edge p_0p_1 is extreme, the directed sequence of points (p_0, p_1, p_2) makes a strict left turn at p_1, so that p_2 is pushed to the head of the stack. Now, if p_3 is also such that the directed sequence of points (p_1, p_2, p_3) forms a left turn, p_3 is pushed to the head of the stack. Else, the earlier decision (i.e. to add p_2) is reversed, and p_2 is deleted. One then checks for the new directed sequence of points p_0, p_1, p_3, and repeats the above process, for all points p_i, $i = 3, ..., s-1$. The algorithm can be simply stated as follows; for a detailed proof of its correctness as well as several important implementation issues, we refer the reader to [107]:

Algorithm 3.3.3 *(Graham's algorithm for computing convex hulls)*
Input: A set S of n points in the plane.
Output: $conv(S)$, the convex hull of S, as a list of vertices ordered cyclically in a counter-clockwise direction around a pivot.

Step 1: Find the rightmost lowest point and label it as the pivot p_0.

Step 2: Sort all other points angularly around p_0; if two points have the same angle with the horizontal ray emanating from p_0, delete the point closer to p_0. Let s denote the total number of sorted points.

Step 3: Set $conv(S) \leftarrow (p_1, p_0) = (p_t, p_{t-1})$; t indexes top.

Step 4: Set $i \leftarrow 2$; while $i < s$ do:
4.1: If p_i is strictly left of $p_{t-1}p_t$ then push p_i to the top of the stack $conv(S)$, and set $i \leftarrow i+1$.

4.2: Else, delete p_t.

Step 5: Output $conv(S)$.

Chapter 4

A new sparse Gaussian elimination algorithm and the Niederreiter linear system for trinomials over $\mathbb{F}_2$

4.1 Introduction

Various methods have been used to solve the Niederreiter linear system using explicit and dense linear algebra in [110] and implicit linear algebra in [39]. While those implementations have achieved factorisation records for polynomials over $\mathbb{F}_2$, confirming the argument that the Niederreiter algorithm is a very efficient linear algebra based algorithm for the factorisation of such polynomials (see [8, 102, 103, 104, 105]), we attempt to investigate the sparsity feature of the algorithm which renders it more efficient for sparse polynomials, and in particular, trinomials. Trinomials are very sparse and hence provide a good model for investigating this aspect of the Niederreiter algorithm. In this chapter, we prove the argument that the Niederreiter matrix is sparse in the case of a trinomial, and establish the exact sparsity pattern and density of the Niederreiter matrix. We also develop a new algorithm for solving the sparse linear system directly to produce a basis for the solution set through Gaussian elimination and using the data structure of Gustavson [65]. The new algorithm is mainly aimed at circumventing the problems that have always been associated with this data structure in terms of elbow space and compression (see Section 4.4). Although it can be easily modified to solve general sparse linear systems over $\mathbb{F}_2$, the algorithm proves to be very efficient when the matrix maintains a high level of sparsity throughout the reduction phase. Our experimental results confirm that the Niederreiter matrix is initially sparse, and maintains its sparsity throughout the reduction phase. Our results can then be incorporated into Göttfert's acceleration of the Niederreiter algorithm over $\mathbb{F}_2$ which uses all the elements of the basis set [59].

We refer to Chapter 3 for the earlier summary of the Niederreiter algorithm over the binary field, and to [38, 41, 87, 102, 103, 104, 105] for an extensive review of the algorithm. In Section 4.2, we prove the assumption that the Niederreiter matrix is sparse for trinomials, and establish that the initial number of entries in the sparse matrix of dimension $d \times d$ does not exceed $3d$. In Section 4.3 we review some of the widely used data structures for storing sparse matrices. In section 4.4 we discuss the new algorithm for performing sparse Gaussian elimination with pivotal ordering using the Markowitz strategy. In Section 4.5, we report on our experimental results,

from which we conjecture that the Niederreiter linear system maintains a high percentage of sparsity throughout the Gaussian elimination phase.

4.2 Setting the Niederreiter matrix over $\mathbb{F}_2$

Let $\mathbb{F}_2$ be the binary field of order 2 consisting only of the elements $0, 1$; it is thus understood that all polynomials described in this chapter are monic. Let f be a polynomial of degree d over $\mathbb{F}_2$, and $f = g_1^{e_1} ... g_m^{e_m}$ be its canonical factorisation over the field. Let N_f denote the Niederreiter matrix defined in Chapter 3. In [102], Niederreiter proposes that, due to the simple form of Eq. (3.2), the system (3.3) has the form:

$$\sum_{k=\max(2j+1-d,0)}^{\min(2j+1,d-1)} f_{2j+1-k} h_k = h_j \text{ for } 0 \leq j \leq d-1. \tag{4.1}$$

The proof of this assertion can be sketched as follows:

First, we know that $h^2(x) = h(x^2)$ in $\mathbb{F}_2[x]$ and so the coefficients of x^{2j} in the right-hand side of Eq. (3.2) are given by h_j for $0 \leq j \leq d-1$. On the other hand, recall that the matrix N_f is obtained by comparing the coefficients of x^{2j} for $0 \leq j \leq d-1$ in both $(fh)'$ and h^2. Let

$$f(x) = \sum_{i=0}^{d} f_i x^i$$

and

$$h(x) = \sum_{k=0}^{d-1} h_k x^k.$$

The coefficient of x^{2j} in $(fh)'$ is the coefficient of x^{2j+1} in fh, and so the coefficients of $(fh)'$ can be expressed as

$$\sum_{k=\max(2j+1-d,0)}^{\min(2j+1,d-1)} f_{2j+1-k} h_k$$

for $0 \leq j \leq d-1$. The bounds on k follow because of the following facts:

a. Since $\deg(h) \leq d-1$, $h_k = 0$ for all $k > d-1$ and so k has to satisfy $k \leq d-1$. Also, since $f_{2j+1-k} = 0$ for all $2j+1-k < 0$, we have to maintain $k \leq 2j+1$. As a result, an upper bound for k would be $\min(2j+1, d-1)$.

b. In a similar way, $h_k = 0$ for $k < 0$ and $f_{2j+1-k} = 0$ for $2j+1-k > d$ (since $\deg(f) = d$). Therefore, k has to satisfy $k \geq 0$ and $k \geq 2j+1-d$ and so a lower bound for k is $\max(2j+1-d, 0)$.

The above proposition establishes a fixed structure for the Niederreiter matrix from which the algorithm derives many of its attractive features over $\mathbb{F}_2$. The following result appears originally in [102]. For a detailed proof of it, we refer the reader to our report in [3].

Theorem 4.2.1 *[102] Let $f = f_d x^d + ... + f_1 x + f_0$ be a polynomial of degree d over $\mathbb{F}_2$. The elements in the Niederreiter matrix N_f can be obtained as follows:*
1. If d is even, then:

a. $f_{2k'}$ appears in rows $i = k' + 1, ..., d/2 + k'$ and occupies column $2(i - k')$ in row i, for $k' = 0, ..., d/2$.

b. $f_{2k'+1}$ appears in rows $i = k' + 1, ..., d/2 + k'$ and occupies column $2(i - k') - 1$ in row i, for $k' = 0, ..., d/2 - 1$.

2. If d is odd, then

a. $f_{2k'}$ appears in rows $i = k' + 1, ..., \frac{d-1}{2} + k'$ and occupies column $2(i - k')$ in row i, for $k' = 0, ..., \frac{d-1}{2}$.

b. $f_{2k'+1}$ appears in rows $i = k' + 1, ..., \frac{d-1}{2} + k' + 1$ and occupies column $2(i - k') - 1$ in row i, for $k' = 0, ..., \frac{d-1}{2}$.

In other words, the Niederreiter matrix over $\mathbb{F}_2$ can be written as

$$\begin{pmatrix}
f_1 & f_0 & 0 & 0 & 0 & 0 & 0 & ... & ... & ... & 0 \\
f_3 & f_2 & f_1 & f_0 & 0 & 0 & ... & ... & ... & ... & 0 \\
f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & 0 & ... & ... & ... & 0 \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
f_{d-1} & f_{d-2} & ... & ... & ... & ... & ... & ... & ... & ... & f_0 \\
0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & ... & f_2 \\
0 & 0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & f_4 \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
0 & 0 & ... & ... & ... & ... & 0 & 0 & f_d & f_{d-1} & f_{d-2} \\
0 & 0 & ... & ... & ... & ... & ... & ... & ... & 0 & f_d
\end{pmatrix}$$

if d is even, and

$$\begin{pmatrix}
f_1 & f_0 & 0 & 0 & 0 & 0 & 0 & ... & ... & ... & 0 \\
f_3 & f_2 & f_1 & f_0 & 0 & 0 & ... & ... & ... & ... & 0 \\
f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & 0 & ... & ... & ... & 0 \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & ... & ... & f_1 \\
0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & f_3 \\
0 & 0 & 0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & f_5 \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
0 & 0 & ... & ... & ... & ... & 0 & 0 & f_d & f_{d-1} & f_{d-2} \\
0 & 0 & ... & ... & ... & ... & ... & ... & ... & 0 & f_d
\end{pmatrix}$$

if d is odd.

Theorem 4.2.1 is at the basis of the assumption that the matrix N_f is sparse if the polynomial f is sparse. It also establishes that there is no cost associated with arithmetic operations for setting up the matrix N_f, in the sense that the matrix coefficients can be read off immediately from those of f. In this chapter, we support the main argument that the Niederreiter matrix is sparse by determining exactly the percentage of sparsity of the matrix upon set-up. The following theorem not only describes a consistent pattern of where the entries occur in the matrix $N_f - I$ if f is a trinomial over $\mathbb{F}_2$, but also the exact number of entries that initially

occur in $N_f - I$. By a simple abuse of notation, we define the length of a row to be the number of nonzero entries appearing in it, rather than its actual length d. We then claim the following:

Theorem 4.2.2 *Let $f(x) = f_d x^d + f_s x^s + f_0$ denote a trinomial over $\mathbb{F}_2$. Let $M = N_f - I$ where N_f is the $d \times d$ Niederreiter matrix and I the $d \times d$ identity matrix over $\mathbb{F}_2$. Then:*

a. if d is even, the matrix M contains exactly one row of length 0, one row of length 1, $\frac{d}{2} - 1$ rows of length 2, and $\frac{d}{2} - 1$ rows of length 3.

b. if d is odd, the matrix M contains exactly one row of length 0 and one row of length 1. In addition, if s is odd, then M contains exactly $\frac{d-3}{2}$ rows of length 2 and $\frac{d-1}{2}$ rows of length 3; else, if s is even, then M contains $\frac{d-1}{2}$ rows of length 2 and $\frac{d-3}{2}$ rows of length 3.

Proof: Suppose that d is even and write $i = j + 1$. By Theorem 4.2.1, f_0 falls along rows $i = j + 1$ of N_f, for $j = 0, ..., d/2$, and f_d falls along rows $i = j + 1$ of N_f, for $j = d/2, ..., d$. Since f is a trinomial, Theorem 4.2.1 also implies that the maximum number of entries in any row of N_f is two, those consisting of the pair (f_s, f_0) or (f_d, f_s), and so, the maximum number of entries in any row of M is three. Let

$$Set_h = \{j : 0 \leq j \leq d - 1 | \text{ row } i \text{ of } M \text{ contains } h \text{ entries}\},$$

for $h = 0, 1, 2, 3$. We aim to show that $||Set_0|| = 1, ||Set_1|| = 1, ||Set_2|| = d/2 - 1$, and $||Set_3|| = d/2 - 1$. Let

$$R_{f_0} = \{j : 0 \leq j < \frac{d}{2} | f_0 \text{ appears in row } i \text{ of matrix } M\},$$
$$R_{f_d} = \{j : \frac{d}{2} \leq j < d | f_d \text{ appears in row } i \text{ of matrix } M\},$$
$$R_{f_s} = \{j : 0 \leq j < \frac{d}{2} | f_s \text{ appears in row } i \text{ of matrix } M\},$$
$$R'_{f_s} = \{j : \frac{d}{2} \leq j < d | f_s \text{ appears in row } i \text{ of matrix } M\}.$$

We first claim that f_0 can never appear as a diagonal entry in N_f; otherwise, if we write $f_0 = f_{2k}$ for $k = 0$, and since f_0 occupies the position $(i, 2(i-k))$ for some $i = 1, .., d/2$ (Theorem 4.2.1), we must have $i = 2(i - k) = 2i$, a contradiction, since $i \geq 1$. Thus, when computing row i in M for $i = 1, ..., d/2$, f_0 in N_f is never cancelled out by a diagonal entry in I_d, or equivalently, $||R_{f_0}|| = \frac{d}{2}$. Similarly, if we write $f_d = f_{2k}$ for $k = \frac{d}{2}$, then for f_d to be a diagonal entry, we must have $i = 2(i - \frac{d}{2})$ or that $i = d$. It follows that f_d appears as a diagonal entry of N_f only in the last row d, and so never appears in row d of M. This implies that

$$||R_{f_d}|| = \frac{d}{2} - 1.$$

Note that $s \neq 0, d$. For f_s to occupy a diagonal entry in N_f, Theorem 4.2.1 implies that

$$i = 2(i - t) \leftrightarrow i = 2t$$

if $s = 2t$ for $t \geq 1$, or that

$$i = 2(i - t) - 1 \leftrightarrow i = 2t + 1$$

if $s = 2t+1$, for $t \geq 0$. It follows that f_s is a diagonal entry of N_f (and hence disappears in M) only in row s, which as such contains either f_0 or f_d alongside f_s. Also, by Theorem 4.2.1, f_s appears in $\frac{d}{2}$ rows of N_f. As a result,

$$||R_{f_s}|| + ||R'_{f_s}|| = \frac{d}{2} - 1.$$

Now, since f_0 is never a diagonal entry in rows $i = 1, ..., d/2$ of N_f, since f_d is a diagonal entry only in row d of the last $d/2$ rows, and since row d does not contain f_s, it follows that $Set_0 = \{d\}$. Since f_s is a diagonal entry of N_f only in row $s \neq 0, d$ (which also contains either f_0 or f_d), $Set_1 = \{s\}$. Also,

$$(R_{f_0} - R_{f_s}) \cup (R_{f_d} - R'_{f_s}) = Set_2 \cup \{i = s\}$$

where Set_2 contains the diagonal entry and one of f_0 or f_d. Thus,

$$\begin{aligned} ||Set_2|| &= ||R_{f_0} - R_{f_s}|| + ||R_{f_d} - R'_{f_s}|| - 1 \\ &= ||R_{f_0}|| + ||R_{f_d}|| - (||R_{f_s}|| + ||R'_{f_s}||) - 1 \\ &= \frac{d}{2} - 1. \end{aligned}$$

Since

$$||Set_0|| + ||Set_1|| + ||Set_2|| + ||Set_3|| = d,$$

we have $||Set_3|| = d/2 - 1$.

A similar proof can be sketched when d is odd and we refer the reader to our report in [3] for details.

4.3 Data structures for the sparse matrix M

In this section we describe briefly the data structures that are widely used to represent sparse matrices. A major review of the subject can be found in [33] (see also [123]). Without loss of generality we assume the matrix to be a square $d \times d$ matrix. Let τ denote the number of nonzero elements appearing in the sparse matrix, and *entries* refer to the nonzero elements. Initially, it is always convenient to supply the matrix in a *coordinate scheme*, a set of triples containing the value of each entry, together with its row and column indices. Since we are working modulo 2, an entry can only have the value 1, hence our disposal of the data structure and algorithmic details which deal with storing the nonzero values, modifying them, or maintaining the numerical stability of the algorithm. The coordinate scheme is one favoured form of supplying the matrix elements, but not for performing Gaussian elimination, for instance, since this would require easy access to rows and columns. For this, two main data structures can be used. In [65], a data structure which transforms the matrix into a collection of sparse row and column vectors is introduced. For simplicity, we assume that the indexing of arrays starts at 1 (not 0). In the collection of sparse row vectors, and for each row of the matrix, we store a pointer to its starting entry and its length. For each entry in that row, we store its column location. For this, we use two integer arrays (say, row_start and row_length) of size d, and one integer array (say jcn) of

size τ. The components in each row vector can be ordered or unordered. As such, all entries along a particular row i have indices $s = row\_start[i], ..., row\_start[i] + row\_length[i] - 1$. Setting the row vector representation from the coordinate scheme can be established using $O(\tau) + O(d)$ operations (see [33]). The collection of rows allows access for entries along a particular row, but not along columns. To perform this, a similar structure designating a collection of sparse columns is established, with arrays col_start, col_length, and irn allowing access to the start of each column and information about its length, as well as the row indices of entries found in the column vectors. This data structure requires four integer arrays of size d and two integer arrays of size τ in total. One main difficulty associated with it arises when a new row becomes longer as a result of row operations introducing new entries. In this case, the current space allocated for the row vector has to be wasted temporarily, so that the new row vector is added to the end of the structure. Subsequently, rows become disordered, and after several such additions requiring what is called *elbow room* at the end of the data structure, one is forced to compress the structure by re-ordering the rows to occupy the free space wasted previously, a process that is known as *compression*. This constitutes the only major disadvantage of the data structure, hence the alternative of linked lists.

A sparse matrix can be represented as a collection of rows, each in a linked list. For each row we store a pointer to its starting entry, say in a one dimensional integer array of length d, denoted by row_header. All subsequent elements in the row are forwardly linked by links, stored in a one dimensional integer array of length τ, denoted by row_fwd_link. The column locations of all entries are stored in a one dimensional integer array jcn of length τ. The fact that the list can be updated without reference to the actual physical location of entries allows proper insertion of new elements upon fill-in without having to have elbow room or perform compression. The list can be ordered or unordered by increasing index of entries. To make insertion or deletion easier and less expensive, the list can be modified into a doubly linked one where entries in a particular row are further linked to their backward neighbors in the list (this is made possible through the use of a one dimensional integer array row_backwd_link of length τ). Since the collection by rows allows only access to entries within a particular row but not a column, a similar solution as above consists in establishing a collection of the matrix columns as linked lists and storing the row indices of entries found along the columns. This transforms the structure into a two dimensional list, whose elements can be singly or doubly linked, ordered or unordered. However, this scheme results in large memory overheads by associating four integers with each entry if the list is not doubly linked, and six integers otherwise. Curtis and Reid [27] suggested that the arrays irn and jcn can be discarded, so that the negation of the row (column) indices of entries along a particular row (column) are stored in the last link of the row (column). A basic difficulty associated with this data structure is that the integers stored can be as large as τ, in contrast to the corresponding upper bound d associated with the Gustavson's data structure, which allows for the use of half-word storage if the array entries fit in 16-bit (32-bit) computer words. Although the issues of elbow room and compression are circumvented, the initial memory requirements can be larger than that of Gustavson's if the doubly linked list is used, consisting of two arrays of size d, and four arrays of size τ.

Summarising, the advantages and disadvantages of the data structures can be listed as follows. The singly linked lists need the same amount of memory as that required by Gustavson's data structure at the time of set-up, but less memory than the doubly linked lists. However, their use requires extensive searches as entries are added or deleted, and the alternative is at the expense of increasing memory requirements through the use of doubly linked lists, or elbow

room and compression through Gustavson's structure. The sequential operational complexities for performing sparse Gaussian elimination using all of the structures above are the same (see [33]), although differences emerge for parallel applications, where the two dimensional unordered double list is of lowest operational complexity (see [123] for a detailed account of complexities). The decision as to which data structure to use becomes a problem-dependent choice which serves the priorities of the application, those being either improvements on running time or savings in memory.

4.4 A new sparse Gaussian elimination algorithm

Solving the linear system constitutes the bottle-neck in the Niederreiter algorithm for very large polynomial degrees. As a result, the challenges in implementing polynomial factorisation consist in first, performing the algorithm for as large an input size as feasible, and second, performing it in the fastest possible way. Accordingly, our main interest in this chapter is aimed at achieving savings in memory to deal with very large sparse linear systems over $\mathbb{F}_2$. These facts, coupled with our interests in factorising as large a trinomial as our resources can afford, motivated us to consider an effective data structure such as Gustavson's and investigate how the problems of elbow room and compression associated with it can be avoided to yield a more space-efficient data structure. Our new algorithm for performing sparse Gaussian elimination using Gustavson's data structure consists of a series of major sub-tasks, each of which is described in due course, together with its operational complexity. We first give a few definitions and notations. Recall that we have to compute the left nullspace of the system (3.3), and as a result we have to perform row operations consisting in interchanging two rows, adding a multiple of one row to another, and multiplying any row by a nonzero field element. Since we are working over the binary field, our algorithm consists of the first two operations only, where the second operation simplifies to adding one row to another modulo 2. To further apply the pivotal Markowitz strategy, we also have to perform column interchanges. To establish the representation of the matrix M as a collection of sparse row vectors, we claim that this does not require that the matrix be given in a coordinate structure. In particular, we have

Proposition 4.4.1 *The matrix M for a trinomial over $\mathbb{F}_2$ can be represented as a collection of ordered sparse row vectors without the use of a coordinate scheme.*

Proof: By Theorem 4.2.2, entries along any row of M can be specified according to their column location. This directly provides the information in arrays row_start, row_length, and jcn.

The collection of sparse column vectors can be easily set up by scanning the rows, in what requires $O(\tau)$ operations. We define $A(a' \to b', c' \to d')$ to be the block matrix comprising rows a' to b' and columns c' to d' of some matrix A. We let γ_c and γ_r denote the maximum number of entries along a nonzero column or row respectively, $M^{(q)}$ the transformed matrix corresponding to M during some stage of Gaussian elimination, and M' the image of $M^{(q)}$ under a transformation which involves any of the following:

1. An interchange I_c of columns.
2. An interchange I_r of rows.
3. Replacing row i with $i + j$ where j is some other row in the matrix. Over $\mathbb{F}_2$, this step reduces to a process of adding and/or removing 1's from row i wherever applicable, and hence

constitutes a composition of a finite number of transformations of the form A_e and R_e, where A_e and R_e represent adding and removing an entry respectively.

The row and column operations in a sparse algorithm appear differently from what can be seen in dense algorithms, where any of these operations is performed as in their literal definition, such that elements of the dense data structure are interchanged whenever rows/columns are so, or added together whenever a row is added to another. In our sparse algorithm, the operations are achieved through a series of changes updating the information in the various arrays describing the data structure, based on the assumption that the transformed matrix is still sparse and hence can be represented by the same structure as that of the original matrix. The sequence according to which the updates are performed is not arbitrary, in that some arrays need to be modified before others. For instance, the lengths of rows and columns have always to be updated first, affecting the pointers to the starts, which then affect the row or column indices of entries. Our algorithm maintains this order of dependence among arrays and this is implicitly assumed to hold in all forthcoming descriptions of the sub-tasks.

4.4.1 Accessing entries along a column

Often enough during any stage of Gaussian elimination one has to be able to locate entries below the pivotal element. To check whether there exists an entry in some position (a, b), we can choose to either access the row a looking for an entry whose column is b, or access the column b looking for an entry whose row is a. If we know that this entry is likely to be situated in the start of a column (for instance, if b happens to be a pivotal column), then accessing the entries by columns would be more efficient. Locating an entry in some position (a, b) can be performed by scanning all entries $s \in \{col\_start[b], ..., col\_start[b] + col\_length[b] - 1\}$. In the remainder of this chapter we denote by $Location\_by\_column(a, b)$ the sub-routine which when input the length and starting index of column b, returns $PASS$ if there exists an entry in location (a, b) and $FAIL$ otherwise. It can be seen that the sub-routine requires at most γ_c field operations, since in the worst-case analysis, one would have to scan an entire column before finding an entry in location (a, b).

4.4.2 Implementing the Markowitz strategy

In our present implementation we use the Markowitz criterion [94] for locally minimising the *fill-in* during each step of the Gaussian elimination, as opposed to other global methods which preserve the general sparsity pattern of the matrix [33]. The Markowitz strategy consists in locating good candidates for pivotal elements during each step of Gaussian elimination. By a good pivotal candidate a_{ij} we mean a nonzero entry in the active part of the matrix which minimises the *Markowitz count* $(r_i - 1)(c_j - 1)$, where r_i and c_j represent the lengths of row i and column j respectively, and where the minimum is over all entries of the active sub-matrix. Note that the Markowitz count represents the maximum amount of fill-in that could arise using a pivotal entry a_{ij}. The Markowitz criterion requires a further numerical stability test to be satisfied by the pivotal candidate, but this is not of concern in our implementation over $\mathbb{F}_2$, since all arithmetic is exact. A straightforward implementation of the Markowitz search is likely to require scanning all entries in the active sub-matrix before a candidate is found. Curtis and Reid (see [27]) introduced methods to avoid an expensively naive search. We describe the method only very briefly here and refer the reader to [33] for more details. In principle, the approach

consists of storing the various row and column lengths and searching for a pivotal candidate through the rows and columns in increasing order of counts (i.e. number of entries). This can be achieved by collecting all the rows (columns) in a set of doubly linked lists of rows (columns) having the same count. The row (column) lists can be constructed using two integer arrays, row_fwd_link and row_backwd_link (or col_fwd_link and col_backwd_link), each of size d, and can be accessed through header pointers, stored in a one dimensional integer array of length d and denoted by row_header (col_header). The search for pivotal elements begins along rows and columns of least count, and progresses in increasing order, whereby the corresponding row and column lists are scanned. For a fixed count w, one first scans all rows of length w, then all columns of the same length. At any stage of the search, one can determine a bound on the Markowitz count of all unsearched entries, which helps terminate the search before all rows and columns are scanned. The following description can be found in [33] and we repeat it only briefly here. Suppose that one is about to search rows with r_i entries, so that all columns whose count is less than r_i have been scanned. This leaves only entries whose count is

$$(r_i - 1)(c_j - 1) \geq (r_i - 1)(r_i - 1) = (r_i - 1)^2.$$

Thus, the first entry whose count is equal to $(r_i - 1)^2$ is chosen as pivot. If no such entry exists, one simply chooses an entry having least count among all other entries in the active sub-matrix. On the other hand, if one is searching columns with count c_j, then, since rows of count c_j have already been scanned, the count of remaining unsearched entries along columns with count c_j is

$$(r_i - 1)(c_j - 1) \geq (c_j + 1 - 1)(c_j - 1) = c_j(c_j - 1),$$

so that as above the search is terminated as soon as an entry whose count is equal to $c_j(c_j - 1)$ is encountered. Else, one again chooses an entry having least count among others in the active sub-matrix. Although there is no theoretical justification that this method always improves upon the $O(\tau)$ process through a naive search, experience has shown that it is likely to be successful after looking through only a few rows and columns (see [33] for experimental results).

Updating lists of columns having the same length

As before, all following arguments work for columns as well as rows, by replacing the arrays with appropriate ones. A number of row and column operations may result in the column lengths being changed, and as a result, the lists of columns having the same count must undergo a corresponding modification. Suppose, for instance, that column i, of original length a, becomes of a new length b. This corresponds to removing i from the list of columns of length a and inserting it into the list of columns of length b. There are several positions into which one can choose to insert the column, the most natural being the head of the list. Obviously, this does not preserve the order of columns by increasing indices, something which, we argue, has advantages in the following:

1. Our application of the Markowitz strategy requires that we have an efficient way of determining the rows and columns of minimum count but only those in the active sub-matrix. Ordering the linked lists allows for a quick way of locating the active rows and columns of some particular count.

2. Interchanging columns takes place when a pivotal candidate is chosen having some minimum Markowitz count c and along a column j that is different from the pivotal column,

say i. As will become clearer later on, our algorithm for interchanging two columns i and j is most efficient when the total number of entries found along columns $k = i, ..., j$ is the least possible. All other columns containing entries whose count is c and having the same length as j belong to the same list, which when ordered in increasing order of column indices helps that we choose column j such that j is closest to i among all columns of its list.

With these advantages, we are inclined to accept the extra cost of maintaining ordered linked lists. We denote the sub-routine for updating the lists as a result of changes in column lengths $Column\_chain(i, a, b)$*, which updates the lists of lengths a and b as the length of i changes from a to b. This can be easily seen to require at most $O(d)$ operations by considering the worst-case analysis of inserting a column to the end of a list of d columns.

4.4.3 Interchanging columns

Because of the symmetry involved in exchanging rows and columns, we discuss only column interchanges. For $k = 1, ..., d$, let c_k denote the vector occupying column k, $L(k)$ the length of c_k in $M^{(q)}$ and $I_c(L(k))$ its length in $M' = I_c(M^{(q)})$. Similarly, if s denotes the index of an entry in the representation of M, then $I_c(s)$ denotes the index of that same entry in the representation of M', whether viewed as a collection of sparse rows or columns. We will further require information about the greatest column index c less than k such that column c is nonzero. This integer can be stored at location k of the integer array $previous\_column$ of size d.† If no such integer exists, $previous\_column[k]$ is set to -1. The array $previous\_column$ can be set at the start of the algorithm and later modified only when a zero column is displaced, a nonzero column has become of length zero, or a zero column has become nonzero. This modification will be assumed to hold implicitly in any of these cases and we leave it to the reader to verify that this comes at a negligible cost. We note that, if $c \neq -1$, then

$$column\_start[k] = column\_start[c] + column\_length[c].$$

The interchange of columns forces us to keep track of the corresponding change in the order of the coordinate entries of the unknown column vector $\mathbf{h^T}$ solving the linear system (3.3). This is necessary for the correctness of the final solution of the system after column interchanges have been performed. For every column $k = 1, ..., d$, we associate an integer y representing the original index of c_k in M and store it in $original\_column[k]$, where $original\_column$ is an integer array of size d. The array can also be initialised at the beginning of the program such that $original\_column[k] = k$ for $k = 1, ..., d$ and modified accordingly when two columns get interchanged.

Now, let i and j denote the two columns to be interchanged such that $i < j$. The matrix $M' = I_c(M^{(q)})$ can be obtained through a series of changes affecting its column lengths and starts, as well as the row and column indices of its entries appearing. The algorithm for interchanging columns is presented as follows:

Algorithm 4.4.1 *Interchange_Columns*(i, j)
Input: The matrix $M^{(q)}$*;*

*A similar algorithm, *Row_chain*, can be analogously constructed, by changing reference to the appropriate arrays.

†A similar array, *previous_row*, can be used in the collection of rows.

Output: The matrix $M' = I_c(M^{(q)})$, *where* I_c *represents the interchange of two columns* i *and* j *of* $M^{(q)}$. *Without loss of generality we may assume that* $i < j$ *and that at most one of* i *or* j *is zero.*

1. $Switch(original\_column[i], original\_column[j])$;
2. $y \leftarrow col\_length[i]$, $col\_length[i] \leftarrow col\_length[j]$;
3. $Column\_chain(i, y, col\_length[i])$;
4. $col\_length[j] \leftarrow y$;
5. $Column\_chain(j, col\_length[i], y)$;
for $k \in \{i, ..., j\}$ *do*
    If $(col\_length[k] = 0)$ *do*
6. $col\_start[k] \leftarrow 0$;
    else do
7. $c \leftarrow previous\_column[k]$;
        If $(c = -1)$ *do*
8. $col\_start[k] \leftarrow 1$;
        else do
9. $col\_start[k] \leftarrow col\_length[c] + col\_start[c]$;
        end;
    end;
end;
10. $Initialise\_to\_zero(new\_array)$, $sum \leftarrow 0$;
for $k \in \{i + 1, ..., j - 1\}$ *do*
11. $sum \leftarrow sum + col\_length[k]$;
end;
If $(col\_length[i] \neq 0)$ *do*
12. $a \leftarrow sum + col\_length[j]$;
    for $y \in \{col\_start[i], ..., col\_start[i] + col\_length[i] - 1\}$ *do*
13. $new\_array[y] \leftarrow irn[y + a]$;
    end;
end;
14. $a \leftarrow col\_length[j] - col\_length[i]$;
for $k \in \{i + 1, ..., j - 1\}$ *do*
    If $col\_length[k] \neq 0$ *do*
        for $y \in \{col\_start[k], ..., col\_start[k] + col\_length[k] - 1\}$ *do*
15. $new\_array[y] \leftarrow irn[y + a]$;
        end;
    end;
end;
If $(col\_length[j] \neq 0)$ *do*
16. $a \leftarrow sum + col\_length[i]$;
    for $y \in \{col\_start[j], ..., col\_start[j] + col\_length[j] - 1\}$ *do*
17. $new\_array[y] \leftarrow irn[y - a]$;
    end;
end;
18. $Copy(new\_array, irn)$, $Initialise\_to\_zero(new\_array)$;
for $k \in \{1, ..., d\}$ *do*

```
    If (row_length[k] ≠ 0) do
19.     sum ← 0;
        for s ∈ {row_start[k], ..., row_start[k] + row_length[k] − 1} do
            If (jcn[s] > i) and (jcn[s] < j) do
20.             sum ← sum + 1;
            end;
        end;
21.     t1 ← Location_by_column(k, i), t2 ← Location_by_column(k, j);
        If (t1 = PASS) and (t2 = FAIL)
        do
            for s ∈ {row_start[k], ..., row_start[k] + row_length[k] − 1}
            do
                If (jcn[s] > i) and (jcn[s] < j) do
22.                 new_array[s − 1] ← jcn[s];
                else if (jcn[s] = i) do
23.                 new_array[s + sum] ← j;
                end;
            end;
        end;
        If (t1 = FAIL) and (t2 = PASS) do
            for s ∈ {row_start[k], ..., row_start[k] + row_length[k] − 1}
            do
                If (jcn[s] > i) and (jcn[s] < j) do
24.                 new_array[s + 1] ← jcn[s];
                else if (jcn[s] = j) do
25.                 new_array[s − sum] ← i;
                end;
            end;
        end;
    end;
end;
27. Copy(new_array, jcn), Initialise_to_zero(new_array).
```

Proposition 4.4.2 *Algorithm 4.4.1 performs correctly and requires* $O\left(d(\gamma_r + \gamma_c)\right)$ *field operations.*

Proof: Steps 1-5 of the algorithm perform the initial changes that have to do with switching the lengths of columns as well as the original indices of columns occupying positions i and j. As a result of changes in column lengths, the corresponding lists of columns of the same count have to be updated through calls to $Column\_chain$, each requiring $O(d)$ operations.

In steps 6-9 we perform the changes to the starting pointers of columns. We claim that only columns $i, ..., j$ of $M^{(q)}$ can have the pointers to their starting entries changed in the representation of M'. Let s denote the index of the starting entry of any column in the matrix. Since columns less than $i-1$ retain the same number and distribution of entries, it follows that all columns $k \in \{1, ..., i-1\}$ retain the same pointers to their starting entries. Suppose $k = i$ and let $c = previous\_column[i]$. If i is a zero column in M' (i.e. has no nonzero entries, something

which, according to our original assumption, implies that i is not a zero column in $M^{(q)}$), then $I_c(s) = 0 \neq s$. Else, if i is not a zero column, and if $c = -1$, then $I_c(s) = 1 = s$; else, if i is not a zero column and $c \neq -1$, let $S(c)$ denote the starting index of column c. We then have

$$\begin{aligned} I_c(s) &= I_c(L(c)) + I_c(S(c)) \\ &= L(c) + S(c) \text{ since } 1 \leq c < i \\ &= s. \end{aligned}$$

Now suppose that $k = i+1, ..., j$ and column k is not zero. If $I_c(L(i)) \neq L(i)$, the start of column k changes as a result. In particular, if $c = previous\_column[k]$, then $I_c(S(k)) = L(c) + S(c)$ if $c \neq -1$, and $I_c(S(k)) = 1$ otherwise. If $k > j$, then since the total number of entries in the matrix block $M'(1 \to n; 1 \to j)$ is the same as that in the matrix block $M^{(q)}(1 \to n, 1 \to j)$, the start of column k remains unchanged. The loop across steps 6-9 is iterated $j - i + 1$ times, involving only array accesses, so that it requires at most $O(j-i)$ steps. In the worst-case analysis when $i = 1$ and $j = d$, the operational count is of the order $O(d)$.

It is immediate to see that the lengths of rows in M' do not change as a result of column interchanges, and hence, the pointers to the starts of rows remain unchanged.

Steps 10-18 update the row indices of entries in the representation of M' as a collection of columns. When i and j are interchanged, some entries in that collection will be displaced and as a result, the values in irn will have to be updated accordingly to fit the new displacement. Because of the dependence of the new values of irn on former values of the same array, the updates on irn are copied first into the auxiliary array new_array which is then copied onto irn when all necessary changes have been performed. Since only columns $k = i, ..., j$ in M' had the pointers to their starting entries changed, it follows that only entries along these columns undergo a shift in their indices which then affects the values in irn. In particular, if s denotes the index of an entry e in $M^{(q)}$ as a collection of columns and $I_c(s)$ is its index in M', then the row index of s in $M^{(q)}$ is equal to the row index of $I_c(s)$ in M'. To establish what the exact changes to irn will be, it suffices to determine the exact value of $I_c(s)$ in each of the following cases:

a. Suppose $e \in j$ in $M^{(q)}$, then $e \in i$ in M'. Since $i < j$, and since the indexing of entries along the collection of columns is ordered increasingly, the index of e in M' decreases as e gets displaced from j to i. Furthermore, the amount of reduction corresponds to the total number of entries being moved ahead of e as i and j are interchanged. In particular, this amounts to the total number of entries found on columns $k = i, ..., j-1$ in M. Summarising, $I_c(s)$ can be written as $I_c(s) = s - a$, where

$$a = \sum_{i \leq l < j} L(l) = sum + L(i) = sum + I_c(L(j))$$

and sum is as defined in the algorithm. Write $s' = I_c(s)$. Then $s' \in \{col\_start[i], ..., col\_start[i] + col\_length[i] - 1\}$. Since $new\_array[s'] = irn[s]$, we have $new\_array[s'] = irn[s' + a]$. Analogously, it is easy to see that, if $e \in i$ in $M^{(q)}$, then $e \in j$ in M' so that $I_c(s) = s + a$, where

$$a = \sum_{i < l \leq j} L(l).$$

The loops in steps 13 and 17 cover all entries in columns i and j and involve mainly array accesses so that their total cost amounts to $O(\gamma_c)$ field operations.

b. Suppose $e \in k$ such that $k = i+1, ..., j-1$. Interchanging i and j would result in decreasing the index of e by $L(i) - L(j)$, since $i < j$, so that $I_c(s) = s - a$, where

$$a = L(i) - L(j) = I_c(L(j)) - I_c(L(i)).$$

The loop covers all entries of indices $s' = I_c(s)$ along k and assigns $new\_array[s'] = irn[s' + a]$. The iterations of the loop cover all entries e along columns $k = i+1, ..., j-1$ involving mainly array accesses so that the total cost of step 15 is $O((j-i)\gamma_c)$. When all the updates are performed, we copy new_array onto irn and re-initialise new_array to zero. This requires $O(\tau)$ operations. In the worst-case analysis when $j = d$ and $i = 1$, the total cost for updating irn thus becomes $O(\tau + d\gamma_c)$.

Steps 19-27 aim at updating the columns of entries as their indices in the collection by rows change. As before, we store the new values of jcn in new_array. Upon interchanging columns i and j, we have seen that only entries found along columns $k' = i, ..., j$ get displaced. Also, if s denotes the index of an entry in the representation of $M^{(q)}$ as a collection of rows such that $I_c(s)$ is the index of that same entry in the representation of M', then the column index of s in $M^{(q)}$ is equal to the column index of $I_c(s)$ in M'. For all rows $k = 1, ..., d$ we argue as follows:

a. If k has entries in both columns i and j, or does not have entries in both columns i and j, then its representation in the collection by rows does not change as a result of interchanging the two columns. As a result, the column indices of all its entries remain unchanged.

Let S denote the number of entries occupying position (k, k') for $k' = i+1, ..., j-1$ and let $sum = \#S$.

b. If k has an entry e in column i but not in column j of $M^{(q)}$, and since $i < j$, then interchanging the two columns augments the index s of e by an amount equal to sum as a result of displacing elements of S to the left. In other words, we have $I_c(s) = s + sum$, and in particular, $new\_array[I_c(s)] = j$. On the other hand, if e' is an entry along row k of $M^{(q)}$ such that $jcn[e'] = i+1, ..., j-1$, then switching columns i and j causes only e to be shifted to the right ahead of e'. If s' denotes the index of e' in $M^{(q)}$, we have

$$I_c(s') = s' - 1, \text{ and } new\_array[s' - 1] = jcn[s'].$$

c. In a very similar way, if there exists an entry e in column j but not in column i of $M^{(q)}$, then $I_c(s) = s - sum$ where s is the index of e in $M^{(q)}$ and sum is as above. In particular, we have $new\_array[I_c(s)] = i$. Also, for e' occupying (k, k') and $k' = i+1, ..., j-1$, $I_c(s') = s' + 1$ so that $new\_array[s' + 1] = jcn[s']$.

The updates on the column indices can thus be performed through a loop ranging over all rows of the matrix. Each loop involves two calls of the function $Location\_by\_column$, whose cost was seen to be $O(\gamma_c)$, as well as a series of updates on entries falling between columns i and j. In the worst-case analysis when $j = d$ and $i = 1$, steps 19-27 require $d(\gamma_r + \gamma_c)$ operations.

Summing up the sub-costs of this algorithm, and using $\gamma_c < d$, it can be seen to require $O(d(\gamma_r + \gamma_c)) + O(\tau) = O(d(\gamma_r + \gamma_c))$ field operations.

4.4.4 Adding rows

Let i denote the pivotal row in a particular stage of Gaussian elimination. We choose to view the process of replacing row $j > i$ with $j + i$ as a series of a composition of two sub-tasks which

involve inserting a new entry to, or removing an already existing one from, row j, resulting in *fill-in* and *fill-out* respectively. Since we are working with integers modulo 2, where an entry is either zero or one, the *fill-out* becomes of considerable significance, for the zeros we obtain as a result of elimination are not *accidental zeros*. In other words, if there exists some entry e in the location (i,k) and another entry e' in (j,k), then e' is definitely (and not accidentally) transformed to zero as a result of the operation $j \leftarrow j+i$. Furthermore, our algorithm for replacing row j with one that is probably longer than itself escapes previous restrictions in that it does not require adding a fresh copy of the modified row at the end of the data structure. In particular, our approach does not require the use of any elbow room beyond what is needed to accommodate for only the extra number of *fill-in* - *fill-out*. In some cases when this quantity is negative, the empty space is simply allocated at the end of the structure. Most of the updates capable of achieving this involve a shifting procedure as described in the previous section, and since this comes at a higher operational cost than using compression and elbow room, the trade off we establish is between savings in memory versus increase in running time.

We first present the following two sub-algorithms:

Algorithm 4.4.2 *Remove_entry*(j,k)
Input: $M^{(q)}$, *where location* (j,k) *is occupied by a nonzero entry.*
Output: $M' = R_e(M^{(q)})$, *where location* (j,k) *is empty.*
1. $\tau \leftarrow \tau - 1$, $x \leftarrow row\_length[j]$, $row\_length[j] \leftarrow row\_length[j] - 1$;
2. $Row\_chain(j, x, row\_length[j])$;
3. $x \leftarrow col\_length[k]$, $col\_length[k] \leftarrow col\_length[k] - 1$;
4. $Column\_chain(k, x, col\_length[k])$;
If $(row\_length[j] = 0)$ *do*
5. $row\_start[j] \leftarrow 0$;
end;
If $(col\_length[k] = 0)$ *do*
6. $col\_start[k] \leftarrow 0$;
end;
for $s \in \{j+1, ..., d\}$ *do*
If $(row\_start[s] \neq 0)$ *do*
7. $row\_start[s] \leftarrow row\_start[s] - 1$;
end;
end;
for $s \in \{k+1, ..., d\}$ *do*
If $(col\_start[s] \neq 0)$ *do*
8. $col\_start[s] \leftarrow col\_start[s] - 1$;
end;
end;
9. $new\_array[\tau + 1] \leftarrow 0$;
If $row\_start[j] \neq 0$ *do*
for $s \in \{row\_start[j], ..., row\_start[j] + row\_length[j] - 1\}$ *do*
If $(jcn[s] \geq k)$ *do*
10. $new\_array[s] \leftarrow jcn[s+1]$;
end;
end;

end;
for $t \in \{j+1, ..., d\}$ *do*
    If $(row\_start[t] \neq 0)$ *do*
        for $s \in \{row\_start[t], ..., row\_start[t] + row\_length[t] - 1\}$ *do*
11. $new\_array[s] \leftarrow jcn[s+1]$;
        end;
    end;
end;
12. $Copy(new\_array, jcn)$, $Initialise\_to\_zero(new\_array)$;
13. $new\_array[\tau + 1] \leftarrow 0$;
If $(col\_start[k] \neq 0)$ *do*
    for $s \in \{col\_start[k], ..., col\_start[k] + col\_length[k] - 1\}$ *do*
        If $(irn[s] \geq j)$ *do*
14. $new\_array[s] \leftarrow irn[s+1]$;
        end;
    end;
end;
while $t \in \{k+1, ..., d\}$ *do*
    If $(col\_start[t] \neq 0)$ *do*
        for $(s \in \{col\_start[t], ..., col\_start[t] + col\_length[t] - 1\}$ **do**
15. $new\_array[s] \leftarrow irn[s+1]$;
        end;
    end;
end;
16. $Copy(new\_array, irn)$, $Initialise\_to\_zero(new\_array)$.

Proposition 4.4.3 *Algorithm 4.4.2 performs correctly and requires* $O(d)+O(\tau)$ *field operations.*

Proof: As one entry is deleted, the total number of entries in the matrix and the lengths of row j and column k are all decreased by 1. Any change in the lengths of rows or columns has to be followed by the corresponding changes in the lists of rows and columns of the same count. In total, steps 1-4 of the algorithm require $O(d)$ field operations through the two calls to *Row_chain* and *Column_chain*.

Steps 5-8 perform the updates on row and column starts. If row j becomes zero, we set its row start to be zero. Else, let s denote the index of the starting entry of j in $M^{(q)}$. If $j = 1$, then $R_e(s) = s = 1$. Now suppose that $j > 1$ and let $M' = R_e(M^{(q)})$. Since the total number and distribution of entries in the matrix block $M'(1 \rightarrow j-1; 1 \rightarrow d)$ is the same as that in $M^{(q)}(1 \rightarrow j-1, 1 \rightarrow d)$, it follows that all rows less than or equal to j retain the same pointer to their starting entries. If we further have $j < d$, we claim that, for rows $r = j+1, ..., d$, $R_e(s) = s - 1$ (where s is the index of the starting entry of r) since removing one entry from row j shifts all entries in the remaining rows one unit to the left. As such, the updates on row starts require $O(d-j)$ operations, which in the worst-case analysis ($j = 1$) require $O(d)$ operations. A similar argument holds for the updates on column starts and hence can be skipped.

Steps 9-12 perform the updates on the column indices of entries in the collection by rows. As one entry e is removed from row j, the location at the end of the array jcn is freed and the

indices of all entries following e in the collection by rows are decreased by 1. As before, the new updates are stored in new_array which is then copied onto jcn before being re-initialised to zero. A very similar argument holds for the row indices of entries in the collection by columns when an entry is removed from column k, and we leave the details to the reader. The updates on jcn and irn thus require at most $O(\tau)$ steps, considering the worst case when $j = k = 1$. This brings the total cost of the algorithm to $O(d) + O(\tau)$ field operations.

Algorithm 4.4.3 *Add_entry*(j, k)
Input: $M^{(q)}$, *where location* (j, k) *is empty.*
Output: $M' = R_e(M^{(q)})$, *where location* (j, k) *is occupied by a nonzero entry.*
1. $\tau \leftarrow \tau + 1$, $x \leftarrow row\_length[j]$, $row\_length[j] \leftarrow row\_length[j] + 1$;
2. $Row\_chain(j, x, row\_length[j])$;
3. $x \leftarrow col\_length[k]$, $col\_length[k] \leftarrow col\_length[k] + 1$;
4. $Column\_chain(k, x, col\_length[k])$;
If $(row\_start[j] = 0)$ *do*
5. $c \leftarrow previous\_row[j]$;
If $(c \neq -1)$ *do*
6. $row\_start[j] \leftarrow row\_start[c] + row\_length[c]$;
else do
7. $row\_start[j] \leftarrow 1$;
end;
end;
for $t \in \{j + 1, ..., d\}$ *do*
If $(row\_start[t] \neq 0)$ *do*
8. $row\_start[t] \leftarrow row\_start[t] + 1$;
end;
end;
for $t \in \{k + 1, ..., d\}$ *do*
If $(col\_start[t] \neq 0)$ *do*
9. $col\_start[t] \leftarrow col\_start[t] + 1$;
end;
end;
10. $t \leftarrow PASS$;
If $(row\_length[j] = 1)$ *do*
11. $new\_array[row\_start[j]] \leftarrow k$, $t \leftarrow FAIL$;
else do
for $s \in \{row\_start[j], ..., row\_start[j] + row\_length[j]) - 2\}$ *do*
If $(jcn[s] > k)$ *and* $(t = PASS)$ *do*
12. $new\_array[s] \leftarrow k$, $t \leftarrow FAIL$;
end;
If $(jcn[s] > k)$ *and* $(t = FAIL)$ *do*
13. $new\_array[s + 1] \leftarrow jcn[s]$;
end;
end;
end;

If $(t = PASS)$ *do*
14. $new\_array[s] \leftarrow k$;
end;
for $t \in \{j+1, ..., d\}$ *do*
If $(row\_start[t] \neq 0)$ *do*
for $s \in \{row\_start[t], ..., row\_start[t] + row\_length[t] - 1\}$ *do*
15. $new\_array[s] \leftarrow jcn[s-1]$;
end;
end;
end;
16. $Copy(new\_array, jcn)$, $Initialise\_to\_zero(new\_array)$;
17. $t \leftarrow PASS$;
for $s \in \{col\_start[k], ..., col\_start[k] + col\_length[k]) - 2\}$ *do*
If $(irn[s] > k)$ *and* $(t = PASS)$ *do*
18. $new\_array[s] \leftarrow j$, $t \leftarrow FAIL$;
end;
If $(irn[s] > j)$ *and* $(t = FAIL)$ *do*
19. $new\_array[s+1] \leftarrow irn[s]$;
end;
end;
If $(t = PASS)$ *do*
20. $new\_array[s] \leftarrow j$;
end;
for $t \in \{k+1, ..., d\}$ *do*
If $(col\_start[t] \neq 0)$ *do*
for $s \in \{col\_start[t], ..., col\_start[t] + col\_length[t] - 1\}$ *do*
21. $new\_array[s] \leftarrow irn[s-1]$;
end;
end;
end;
22. $Copy(new\_array, irn)$, $Initialise\_to\_zero(new\_array)$.

Proposition 4.4.4 *Algorithm 4.4.3 performs correctly and requires* $O(d) + O(\tau)$ *operations.*

Proof: When an entry is added to the location (j, k), the total number of entries in the matrix increases by 1, and so do the lengths of row j and column k. The corresponding lists of rows and columns of the same length are modified through the calls to *Column_chain* and *Row_chain*. Steps 1-4 as such require $O(d)$ field operations.

Steps 5-9 perform the updates on the row and column starts. As in algorithm 4.4.2, and for rows $r = 1, ..., j$, we have $A_e(s) = s$ (where s is the index of the starting entry of a nonzero row r -check proof of Proposition 4.4.3 above), unless j was originally a zero row, in which case we argue as follows. If j becomes the first nonzero row of M', then $A_e(s) = 1$; else,

$$A_e(s) = row\_start[c] + row\_length[c]$$

where $c = previous\_row[j]$. If we further have $j < d$, then for rows $r = j+1, ..., d$, $A_e(s) = s+1$ since adding one entry to row j shifts the indexing of all entries in the remaining rows one unit

to the right. A similar argument holds for the updates on the starts of columns in the matrix. For column k, however, we argue that $A_e(s) = s$, since the inserted entry will always appear after the start of column k when this is the pivotal column. It can be further established that the operational complexity of the updates on the starts of rows and columns is of the order $O(d-j) + O(d-k)$ field operations, which in the worst-case analysis ($j = k = 1$) becomes of the order $O(d)$.

Steps 10-16 perform the updates on the column indices of entries in the collection of rows. Let e' denote the entry to be inserted. If row j becomes of length 1, then e' is given the column index k, and the algorithm skips to step 15 where updates on the column indices for entries in rows $r > j$ take place. Else, suppose that row j becomes of length greater than 1. Let e denote an entry in j and of index s in $M^{(q)}$. If $jcn[s] < k$, then e also represents an entry of j in M' whose index is not affected, since it appears before e'. Else, if e is the first entry of j in $M^{(q)}$ such that $jcn[s] > k$, then e' takes up the index of e so that $new\_array[s] = k$ in M'. For all entries e in row j whose columns are greater than k in $M^{(q)}$, $A_e(s) = s + 1$ so that $new\_array[s+1] = jcn[s]$. If no entry e of row j in $M^{(q)}$ was found to have a column index greater than k (step 14), then e' is the last entry to appear in row j and hence has index $s = row\_start[j] + row\_length[j] - 1$, which is achieved through exit of the loop in the preceding step 12. For all entries e along rows r greater than j and of index s in $M^{(q)}$, $A_e(s) = s + 1$ as a result of inserting e' in j. When all updates are performed, new_array is copied onto jcn and re-initialised to zero. In total, and considering the worst-case analysis when $j = 1$, this step can be seen to require at most $O(\tau)$ field operations. A very similar argument can be established for updating the row values of entries in M' as a collection of columns and we leave the details to the reader. Summing up the sub-costs of the algorithm, the total cost is $O(d) + O(\tau)$ field operations.

The algorithm for adding rows can now be described using the previous two sub-tasks. Our discussion above also demonstrates that no compression or creation of elbow room is required to accommodate for new copies of modified rows (or columns). Whatever extra space created corresponds only to the amount of fill-in minus fill-out, when this amount is positive. In particular, the algorithm can be stated as follows:

Algorithm 4.4.4 *Add_Rows*(i, j)
Input: Rows i and j such that i is the pivotal row during one stage of Gaussian elimination and j is some other row in the active sub-matrix of $M^{(q)}$.
Output: Row j such that $j \leftarrow j + i$.
for $s \in \{row\_start[i], ..., row\_start[i] + row\_length[i] - 1\}$ *do*
1. $k \leftarrow jcn[s]$;
If $location\_by\_column(j, k) = PASS$ *do*
2. *Remove_entry*(j, k);
else do
3. *Add_entry*(j, k);
end;
end.

Proposition 4.4.5 *Algorithm 4.4.4 performs correctly and requires $O(d\gamma_r^2)$ field operations.*

Proof: Replacing j with $j+i$ modulo 2 can be performed in distinct steps as follows. If both rows i and j contain an entry in the same column k, then performing $j+i$ will cancel out this entry in location (j,k); else, this will introduce a new entry in (j,k). The loop through steps 1-3 iterates γ_r times during which a call to *Remove_entry* or *Add_entry* is performed. By Propositions 4.4.3 and 4.4.4, and since $\tau = O(\gamma_r d)$, the total running time of algorithm 4.4.4 is at most $O(d\gamma_r^2)$ field operations.

4.4.5 A complete sparse Gaussian elimination algorithm

The results of the previous subsections can now be used to construct a complete sparse Gaussian Elimination algorithm. The forward sweep of the algorithm which reduces the matrix into Echelon form consists of at most d pivotal steps. In each step, a Markowitz search is performed to find a suitable pivotal element. Accordingly, an interchange of rows and/or columns is performed, and/or an elimination of entries falling below the pivotal element is performed by adding suitable rows. If we consider the worst-case analysis in which the Markowitz search requires $O(\tau)$ operations, and combining the costs in Propositions 4.4.2 and 4.4.5, we find that the forward sweep is of the order

$$O\left(d^2(\max(\gamma_r,\gamma_c))^2\right) \tag{4.2}$$

field operations. Our experience has shown that roughly speaking, the maximum number of entries in a row or column is negligible compared to d (see table 4.1 below). In particular, if α denotes the maximum ratio τ/d attained during any stage of the reduction, we have $\alpha = O(\max(\gamma_r,\gamma_c))$ and the cost in (4.2) is of the order $O(\alpha^2 d^2)$. It can also be seen that the backward sweep of the algorithm, which produces the basis elements once the matrix is reduced, requires $O(d\gamma_r)$ field operations, so that the total cost of the algorithm involving both its forward and backward sweeps is bounded by $O(\alpha^2 d^2)$. When compared with the dense and explicit Gaussian elimination of order $O(d^3)$, our algorithm performs faster provided that $\alpha < \sqrt{d}$. In terms of its memory requirements, we have shown that the algorithm requires twelve integer arrays of size d, and three integer arrays of size $\tau = O(\alpha d)$. Compared to the spatial requirement d^2 of the explicit methods, our algorithm is more memory efficient provided $\alpha < (d-12)/3$. Both upper bounds hold easily in our implementations.

4.5 Implementation and run times

All programs were written in C. The work was carried out at Oxford University Supercomputing center (OSC) on the Oswell machine. Only one processor of the Sun cluster was used to perform the serial work. The UltraSPARC III processors run at about 122.2 Mflop/sec each. We generated a number of trinomials over $\mathbb{F}_2$ completely randomly. Table 4.1 lists the run times in minutes for setting up and solving the Niederreiter linear system for random binary trinomials of degree d and having m irreducible factors over $\mathbb{F}_2$. As before, α denotes the maximum ratio τ/d attained at any stage, and $\max(\gamma_c,\gamma_r)$ denotes the maximum number of entries appearing in any row or column during the entire Gaussian elimination phase. We note the negligible values of $\max(\gamma_r,\gamma_c)$ compared to d. The run times verify the effect of increasing values of α or d on the total execution time, and our findings assert that the matrix remains considerably

d	m	$\alpha = \max(\tau/d)$	$\max(\gamma_r, \gamma_c)$	Time in minutes
8000	6	6	253	39.4
8000	7	2	33	1.2
8000	30	4	150	7.5
16000	7	2	34	4.7
16000	10	9	430	258.5
16000	14	2	126	6.1
32000	31	3	240	54.9
32000	10	15	854	19400
64000	7	2	40	77.8
64000	10	3	248	215.7
128000	10	3	363	812.9
128000	14	2	178	343.5
256000	14	2	288	1896.1
300000	11	2	52	1634.3
400000	11	2	49	5061.1

Table 4.1: Run times for setting up the Niederreiter matrix and solving the associated system.

sparse throughout the reduction phase no matter how large the degree of the trinomial grows. As a result, our sparse algorithm performs efficiently well without having to be transformed into a dense algorithm towards the final stages of Gaussian elimination. Our impression is that a similar behaviour might still be observed for sparse polynomials of more than three terms, particularly because of the distribution of the Niederreiter matrix, which heuristically seems to preserve its sparsity throughout the elimination phase.

4.6 Conclusion

In this chapter we have investigated the initial and most limiting phase of the Niederreiter algorithm for trinomials over the binary field and determined the exact initial sparsity level of the associated Niederreiter linear system. A new sparse Gaussian elimination algorithm using the Markowitz strategy was developed for producing a basis of the solution set of the sparse linear system. The new algorithm exploits the Gustavson data structure but circumvents the problems associated with it regarding creation of elbow room and compression. The problem of requiring extra space and modifying the data structures is however shifted within the subroutines for adding one row to another. Yet, our approach does not require the use of any elbow room beyond what is needed to accommodate for only the extra number of *fill-in* - *fill-out*, which in the case of Niederreiter's linear system for trinomials over $\mathbb{F}_2$ remains considerably small. This was supported by our experimental results where the linear system remained considerably sparse throughout the Gaussian elimination phase. The resulting algorithm is also more memory efficient than the two dimensional doubly linked list which has been the most efficient structure among linked-lists based data structures. The gains in spatial requirements come at the expense of running time where our new algorithm requires $O(\alpha^2 d^2)$ field operations, in contrast to the $O(\alpha^2 d)$ cost of other sparse algorithms, where $\alpha = \tau/d$. Our algorithm was used in solving very large sparse Niederreiter linear systems for trinomials over $\mathbb{F}_2$, but can also serve as an

irreducibility test for trinomials over $\mathbb{F}_2$, where a trinomial is irreducible if and only if the rank of the reduced Niederreiter system is such that $m = d - rank = 1$ [102]. Although our algorithm can be easily modified to become a general linear solver in various other applications, we expect it to be particularly effective in solving the Niederreiter linear system for sparse polynomials over $\mathbb{F}_2$. Our work in this chapter can be combined with results of [1] where the irreducible factors of f are extracted from a basis of the solution set using a parallel approach to the Göttfert algorithm over $\mathbb{F}_2$. When compared with work in [110] for a dense explicit linear algebra approach to the Niederreiter algorithm, the resulting hybrid algorithm is of a better spatial complexity for the factorisation of large trinomials over $\mathbb{F}_2$, provided $\alpha < (d - 12)/3$, a criterion that is easy to establish in the sparse Niederreiter linear system. Our algorithm achieves factorisation degrees that are inaccessible to the dense implementation up to 16 processors (see [1]), and performs the nullspace stage for $d = 300000$ in about 27 hours using only one processor in contrast to the performance in [110] requiring about 10 hours and 256 nodes for a random (possibly dense) polynomial of the same degree. Our algorithm also achieves factorisations beyond this degree.

Chapter 5

A BSP model of the Göttfert algorithm for polynomial factorisation over $\mathbb{F}_2$

5.1 Introduction

The solutions of the Niederreiter linear system can lead to a complete factorisation in a variety of ways, one of which was presented by Göttfert [59] for fields of characteristic 2, leading to a simple and polynomial time algorithm for extracting the factors using only the basis elements of the solution set of the Niederreiter linear system.

In this chapter, we develop a new BSP parallel approach to the Göttfert algorithm over $\mathbb{F}_2$. The BSP model offers simplicity in terms of its cost analysis and its clear distinction between the three important phases of computation, communication, and synchronisation. It also has the advantage of being independent of the underlying architecture of the machine, thus providing portable software that can be used efficiently in a variety of applications (see [14, 71]). Our algorithm achieves high efficiency in many of our test cases and can thus be used efficiently to factorise very large polynomials over $\mathbb{F}_2$ provided a basis of the solution set is given. For a brief survey of the algorithms underlying our work and some background information describing the BSP parallel model, we refer the reader to Chapter 3. In Section 5.2 we present our parallel algorithm, prove its correctness and discuss its BSP cost analysis. In Section 5.3 we report on our experimental results and discuss the scalability of the algorithm.

5.2 A parallel approach to Göttfert's refinement of the Niederreiter algorithm

As in the earlier chapter, let $\mathbb{F}_2$ be the binary field of order 2 consisting only of the elements $0, 1$; it is thus understood that all polynomials described in this chapter are monic. Let f be a polynomial of degree d over $\mathbb{F}_2$, and

$$f = g_1^{e_1} \cdots g_m^{e_m}$$

be its canonical factorisation over the field. Let N_f be the Niederreiter matrix of coefficients of f. Let $\mathbf{h} = (h_0, ..., h_{d-1})$ denote the coefficient row vector of an unknown polynomial $\mathbf{h}$ over $\mathbb{F}_2$

of degree less than d. In [102], Niederreiter establishes that the solutions $\mathbf{h}$ of the system (3.3) form a linear subspace of the vector space $\mathbb{F}_2[x]$ of dimension m over $\mathbb{F}_2$ and that they are given by

$$\mathbf{h} = \frac{f}{b} b'$$

where b denotes a factor of $g_1 \cdots g_m$. Since b is square-free (so that $\gcd(b, b') = 1$), we have also seen that

$$\gcd(f, \mathbf{h}) = \frac{f}{b} \gcd(b, b') = \frac{f}{b}.$$

Let $\{\mathbf{h_1}, ..., \mathbf{h_m}\}$ be a basis spanning the solution set of the system (3.3). The corresponding polynomials

$$b_i = \frac{f}{\gcd(f, \mathbf{h_i})} \in \mathbb{F}_2[x] \quad \text{for } i = 1, ..., m$$

are square-free factors of f. In the present chapter, all flops are considered as binary operations, since we are working over the binary field.

5.2.1 Detecting parallelism in the Göttfert setting

Let $\#r_n$ denote the maximum number of non-constant polynomials P_i, for $i = 1, ..., \#r_n$, that can appear in any row n described in the Göttfert construction of Chapter 3. Each P_i can be the result of a gcd or a division operation, in which case we denote it by a D-polynomial or an R-polynomial respectively. Furthermore, we assert the following:

Claim 5.2.1 *$\#r_n = 2^n - 1$ for $n = 1, ..., m$.*

Proof: We prove the claim by induction on n. For $n = 1$, we know that row 1 consists of the polynomial b_1 only. Suppose the assertion is true for n. We know that any row $n + 1$ has at most one plus twice the number of non-constant polynomials in row n so that

$$\#r_{n+1} = 2\#r_n + 1 = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1.$$

It is easy to see that there are at most $(\#r_n - 1)/2$ non-constant D-polynomials and at most $(\#r_n + 1)/2$ non-constant R-polynomials in each row n. We denote D and R-polynomials in row n by $n; D_j$ and $n; R_{j'}$ respectively, where j and j' are the polynomials' indices along row n. For consistency throughout the text, we can arrange the computations along rows so that all the D polynomials are computed first, their corresponding R polynomials next, and the polynomial $b_n / \prod_j n; D_j$ (where the product is over non-constant polynomials $n; D_j$) last. With this notation, it is also easy to see that, if the polynomials in row $n - 1$ are written as

$$(n-1); D_i, \text{ for some } i = 1, ..., (\#r_{n-1} - 1)/2,$$

and

$$(n-1); R_i, \text{ for some } i = (\#r_{n-1} + 1)/2, ..., \#r_{n-1},$$

then row n consists of

$$n; D_i = \begin{cases} \gcd(b_n, (n-1); D_i), & \text{if } 1 \le i \le (\#r_{n-1} - 1)/2, \\ \gcd(b_n, (n-1); R_{i-(\#r_{n-1}-1)/2}), & \text{if } (\#r_{n-1} + 1)/2 \le i \le \#r_{n-1}, \end{cases}$$

$$n; R_i = \begin{cases} (n-1); D_i / n; D_i, & \text{if } 1 \le i \le (\#r_{n-1} - 1)/2, \\ (n-1); R_{i-(\#r_{n-1}-1)/2} / n; D_i, & \text{if } (\#r_{n-1} + 1)/2 \le i \le \#r_{n-1}, \end{cases}$$

and

$$n; R_{\#r_{n-1}+1} = \frac{b_n}{\prod\limits_{\substack{i=1 \\ n;D_i \neq 1}}^{\#r_{n-1}} n; D_i}.$$

The first step in our parallel approach consists of studying the dependencies between the gcd and division computations and structuring these dependencies in a parallel hierarchy. Without loss of generality we may assume that the number of threads coincides with the number of processors available. We introduce the concept of a parallel queue, which consists of a list of polynomials that can be computed independently by a number of p processors using a number of supersteps. The queue comprises a set of jobs that are not necessarily performed in the same parallel superstep; however, the jobs are entirely independent (and hence can be performed in any order) and do not require that the processors synchronise at any point before the queue is fully tackled. The first parallel queue consists of the polynomials b_i, for $i = 1, ..., m$, where m polynomials can be computed simultaneously in parallel. The second parallel queue consists of the polynomial $2; D_1$ only, since all other polynomials (in its row or in following rows) depend on it. This constitutes the only queue where not enough distinct tasks are available to engage all processors. In fact, the ensuing queues start filling up immediately according to an iterative formula derived from the dependencies that we describe in the following algorithm:

Algorithm 5.2.1 *Set_Queues*$(queue_k, queue_{k'})$
Input: $queue_k = \{P_1, ..., P_s\}$, *a list of non-constant polynomials from the Göttfert setting computed in a parallel queue* $k \ge 2$.
Output: $queue_{k'}$, *a list of polynomials that can be computed in the parallel queue* $k' > k$.
1. $queue_{k'} = \emptyset$;
for $j \in \{1, ..., s\}$ *do*
  if $P_j = n; D_i$ *for some* $n = 2, ..., m$ *and some* $i = 1, ..., \#r_{n-1}$ *do*
2. $queue_{k'} \leftarrow queue_{k'} \cup \{n; R_i\} \cup \{(n+1); D_i\}$.
  end;
  if $P_j = n; D_i$ *for some* $n = 2, ..., m$ *and* $i = \#r_{n-1}$ *do*
3. $queue_{k'} \leftarrow queue_{k'} \cup \{n; R_{\#r_{n-1}+1}\}$.
  end;
  if $P_j = n; R_i$ *for some* $n = 2, ..., m$ *and some* $i = 1, ..., \#r_{n-1} + 1$ *do*
4. $queue_{k'} \leftarrow queue_{k'} \cup \{(n+1); D_{i+((\#r_n-1)/2)}\}$.
  end;
end.

Theorem 5.2.1 *The algorithm works correctly as specified, producing all the rows in the Göttfert algorithm required to achieve a complete factorisation. As a result, the algorithm requires at most* $3s$ *steps for a list of size* s.

Proof: We assume that the polynomials $b_1, ..., b_m$ are already computed. Correctness of the algorithm follows as a result of justifying the steps 2-4:

Step 2: Suppose $P_j = n; D_i$ for some $n = 2, ..., m$ and some $i = 1, ..., \#r_{n-1}$. If $1 \leq i \leq (\#r_{n-1} - 1)/2$, then

$$P_j = \gcd(b_n, (n-1); D_i) \text{ and } n; R_i = \frac{(n-1); D_i}{P_j}.$$

Since P_j has already been computed in queue k, we know that $(n-1); D_i$ must be a non-constant polynomial computed in queue $j < k$. As such, $n; R_i$ has both its components available and can be assigned to $queue_{k'}$. Else, if $(\#r_{n-1} + 1)/2 \leq i \leq \#r_{n-1}$, then

$$P_j = \gcd(b_n, (n-1); R_{i-((\#r_{n-1}-1)/2)}) \text{ and } n; R_i = \frac{(n-1); R_{i-((\#r_{n-1}-1)/2)}}{P_j}.$$

Again, $(n-1); R_{i-((\#r_{n-1}-1)/2)}$ must be a non-constant polynomial computed in queue $j < k$, so that $n; R_i$ can be assigned to $queue_{k'}$. On the other hand, for $i = 1, ..., \#r_{n-1}$ (or $i = 1, ..., (\#r_n - 1)/2$) we know that

$$(n+1); D_i = \gcd(b_{n+1}, n; D_i).$$

The proof now follows as above.

Step 3: Suppose $P_j = n; D_i$ for some $n = 2, ..., m$ and $i = \#r_{n-1}$. Since $n; D_{\#r_{n-1}}$ is the D-polynomial to be computed last in row n, and since

$$n; R_{\#r_{n-1}+1} = \frac{b_n}{\prod\limits_{\substack{i=1 \\ n;D_i \neq 1}}^{\#r_{n-1}} n; D_i},$$

the proof follows as above.

Step 4: Suppose that $P_j = n; R_i$ for some $n = 2, ..., m$ and some $i = 1, ..., \#r_{n-1} + 1$. We know that i also satisfies $i = 1, ..., (\#r_n + 1)/2$ or

$$\frac{(\#r_n + 1)}{2} \leq i + \frac{(\#r_n - 1)}{2} \leq \#r_n.$$

Moreover, we have

$$(n+1); D_i = \gcd(b_{n+1}, n; R_{i-((\#r_n-1)/2)}) \text{ if } \frac{(\#r_n + 1)}{2} \leq i \leq \#r_n,$$

or equivalently

$$(n+1); D_{i+\frac{\#r_n-1}{2}} = \gcd(b_{n+1}, n; R_i) \text{ if } \frac{(\#r_n + 1)}{2} \leq i + \frac{(\#r_n - 1)}{2} \leq \#r_n.$$

The proof now follows as above.

With the correctness of the algorithm now established, it becomes immediate to see that any polynomial in $queue_k$ can lead to at most 3 polynomials in $queue_{k'}$ (e.g. the polynomial satisfying the conditions in steps 2 and 3 above).

5.2.2 The parallel Göttfert algorithm

One major characteristic of the algorithm is that it consists of task parallelism, since distributing the data would require much more synchronisation between processors in the inner loops than would be the case in our present algorithm. To minimise the number of synchronisation barriers, we choose to make all initial data available globally at the beginning of the algorithm and all recently computed data available to all processors once they are obtained. The following details are crucial in following up on the algorithm and describe some of the data structures as well as the notations we adopt throughout this section. The polynomials are represented by integer arrays whose entries are either zero or one. The coefficients are packed into bit-words (where w_l is the bit-size of the computer word being used). This not only speeds up the polynomial arithmetic sub-routines but also minimises the number of messages to be transmitted among processors, and hence the BSP cost of the algorithm. We describe several arrays that store either the values of the polynomials or information about them. Unless otherwise stated, all arrays are global.

We first define two copies of three integer arrays, $Type_{j,j'}$, $Row_{j,j'}$, and $Index_{j,j'}$, each of size m. Those serve to hold temporarily information about a polynomial P_i being computed in some parallel queue. In particular, $Type_{j,j'}[i]$ denotes the type of the polynomial (whether a D or an R polynomial), $Row_{j,j'}[i]$ the row to which it belongs, and $Index_{j,j'}[i]$ its index within that row. Those arrays are embedded within two queues $queue_j$ and $queue'_j$ such that $queue_j$ is a sequence of triples $(Type_j[i], Row_j[i], Index_j[i])$, for $i = 0, ..., \#(queue_j) - 1$, and each such triple describes a polynomial already computed in some parallel queue. On the other hand, $queue_{j'}$ consists of similar triples describing polynomials to be computed in a forthcoming parallel queue.

By a simple abuse of notation we also define what we call an array of polynomials $Poly_k$ of size $\lceil d/w_l \rceil$ (where w_l is the bit-size of the computer word being used). By this we simply mean a two dimensional array of integers $Poly$ such that the k'th row of the two dimensional array contains the coefficients of the polynomial $Poly_k$. The array is used to store permanently the values of all non-constant polynomials computed in the parallel process. Similarly, we define an array of polynomials B_k of size $\lceil d/w_l \rceil$ containing the coefficients (in bit-words) of the polynomials $b_i = f/gcd(f, \mathbf{h_i})$, for $i = 0, ..., m-1$. We note that the polynomial indices are global variables indicating that $Poly_k$ and B_k are global polynomials whose individual values are computed by one particular processor then broadcast to all at one fixed location k independent of the processor id. To keep track of the number of non-constant polynomials in each row, we define the integer array $Length$ of size m such that $Length[i]$ denotes the total number of non-constant polynomials located along row i during any phase of the parallel process. We also define two dimensional arrays of pointers, D and R, of approximate sizes $m \times (2^m - 1)$. $D[n][i]$ points to null if the polynomial $n; D_i$ is constant; else, it contains the address of the row in $Poly$ where the polynomial $n; D_i$ is stored. A similar description holds for the array R. Finally, we define two integer arrays sum, and $local\_sums$, of sizes m and $p \times m$ respectively, that are used to update the lengths of rows individually by each processor, as will be described later on.

All communication between processors is achieved through the bsp_put command [66, 67]. We use the short-hand of the function call as in

$$y \leftarrow BSP\_Put(s, N, x).$$

where N is an integer greater than or equal to zero. If $N = 0$, this indicates that the processor meeting the command is sending its individual value of variable x onto variable y found on

processor s. Otherwise, x denotes a polynomial whose coefficients form the first N entries of array x and are being sent to processors s at the same corresponding locations in the global array y. *Signal* is an indicator which controls the flow of the loops in that, if the length of any row becomes equal to m, we set *Signal* to *Stop*, indicating that all irreducibles have been found (see Theorem 3.1.7); else, we set it equal to OK in which case all loops continue to operate. $queue\_length$ always designates the number of polynomials to be computed in a new parallel queue, and $total\_poly$ designates the total number of non-constant factors determined during any stage of the parallel algorithm. Our parallel algorithm now takes the following form:

Algorithm 5.2.2 $Parallel\_Göttfert(f, d, m, \{\mathbf{h_0}, ...\mathbf{h_{m-1}}\}, p, id)$
Input: f *a polynomial of degree* d *over* $\mathbb{F}_2$, $m > 1$ *the number of irreducible factors of* f, $\{\mathbf{h_0}, ..., \mathbf{h_{m-1}}\}$ *a basis for the solution set of (3.3),* p *the total number of processors operating in parallel, and* id *the processor identification number ranging from* $0, ..., p-1$.
Output: the m *irreducible factors of* f *and their multiplicities in* f.
1. $Signal \leftarrow OK$, $k \leftarrow id$;
while $(k < m)$ *do*
2. $b_k \leftarrow f/\gcd(f, \mathbf{h_k})$, $degree \leftarrow \deg(b_k)$;
for $y \in \{0, ..., p-1\}$ *do*
3. $b_k \leftarrow BSP\_Put(y, degree + 1, b_k)$;
end;
4. $k \leftarrow k + p$;
end.
5. $BSP\_synchronise()$.
6. $P_0 \leftarrow \gcd(b_0, b_1)$;
if $(P_0 \neq 1)$ *do*
7. $Poly_0 \leftarrow P_0$, $D[2][1] \leftarrow \&Poly_0$, $total\_poly \leftarrow 1$, $Length[2] \leftarrow 1$;
else do
8. $total\_poly \leftarrow 0$;
end;
9. $queue_j \leftarrow \{P_0\}$, $Set\_Queues(queue_j, queue_{j'})$, $queue\_length \leftarrow \#queue_{j'}$;
while $(Signal = OK)$ *do*
10. $k \leftarrow id$, $Set\_to\_zero(sum, local\_sums)$;
while $(k < queue\_length)$ *do*
11. $Type \leftarrow Type_{j'}[k]$, $n \leftarrow Row_{j'}[k]$, $i \leftarrow Index_{j'}[k]$,
$P_k \leftarrow Compute\_Polynomial(Type, n, i)$;
if $(P_k \neq 1)$ *do*
12. $Poly_{(k+total\_poly)} \leftarrow P_k$;
if $(Type = D\_type)$ *do*
13. $D[n][i] \leftarrow \&Poly_{(k+total\_poly)}$;
else do
14. $R[n][i] \leftarrow \&Poly_{(k+total\_poly)}$;
end;
15. $sum[n] \leftarrow sum[n] + 1$, $degree \leftarrow \deg(Poly_k)$;
for $y \in \{0, ..., p-1\}$ *do*
16. $Poly_{(k+total\_poly)} \leftarrow BSP\_Put(y, degree + 1, Poly_{(k+total\_poly)})$;
end;

```
                    if (Type = D_type) do
                        for y ∈ {0, ..., p − 1} do
17.                         D[n][i] ← BSP_Put(y, 0, D[n][i]);
                        end;
                    else do
                        for y ∈ {0, ..., p − 1} do
18.                         R[n][i] ← BSP_Put(y, 0, R[n][i]);
                        end;
                    end;
                end;
19.             k ← k + p;
            end;
            for y ∈ {0, ..., p − 1} do
                for w ∈ {2, ..., m} do
20.                 local_sums[y][w] ← BSP_Put(y, 0, sum[w]);
                end;
            end;
21.         BSP_synchronise();
            for (y ∈ {0, ..., p − 1}) do
                for (w ∈ {2, ..., m}) do
22.                 Length[w] ← Length[w] + local_sums[y][w],
                    total_poly ← total_poly + local_sums[y][w];
                    if (Length[w] = m) do
23.                     Signal ← Stop, last_row ← w;
                    end;
                end;
            end;
            if (Signal = OK) do;
24.             queue_j ← queue_j', queue_length ← Sort(queue_j), queue_j' ← (),
                Set_Queues(queue_j, queue_j');
            end;
        end;
25.     i ← id + 1;
        while (i ≤ 2^(last_row−1)) do
            if (D[last_row][i] ≠ NULL) do
26.             factor ← *D[last_row][i], exp ← Multiplicity(f, factor),
                return (factor, exp);
            end;
            if (R[last_row][i] ≠ NULL) do
27.             factor ← *R[last_row][i], exp ← Multiplicity(f, factor),
                return (factor, exp);
28.         i ← i + p;
        end.
```

The algorithm is called by all processors which implement the same copy of it for various data, conforming to the SPMD model: A single program with multiple data is encountered by

all processors, which then execute their own version of the program, distinguished by their own identification number, $id = 0, ..., p-1$. In step 1 of the algorithm, few initialisations are set. The first "while" loop is a parallel loop met by all processors which compute the square-free factors $b_k = f/\gcd(f, \mathbf{h_k})$, for $k = 0, ..., m-1$. k is a global variable which when first set to id and then incremented by p guarantees that all processors compute almost an equal number of polynomials b_k. This constitutes the first parallel queue according to Algorithm 5.2.1. Every processor then broadcasts its own value of b_k to all other processors, but no synchronisation barrier is met until all the b_k's are computed, since they are not needed in any loop computation. A synchronisation point in the loop as such would only incur an extra cost of synchronisation without actually being required.

The second parallel queue consists of the polynomial $2; D_1$ (see Algorithm 5.2.1) which is computed by all processors. Although this constitutes a sequential step, the processors start to engage in distinct computations soon after the second queue is set up. If $P_0 = 2; D_1$ is not trivial, it is stored in a permanent location in $Poly_0$, $D[2][1]$ is set to point to the location of $Poly_0$ (which we denote by $\&Poly_0$), and the length of row 2 and the total number of non-constant factors computed so far are updated. We call Algorithm 5.2.1 to set up the ensuing $queue_{j'}$ of polynomials to be computed in parallel. $queue\_length$ denotes $\#queue_{j'}$.

Thereafter, the main loop of the algorithm is iterated so long as $Signal$ is not set to $Stop$ (indicating that none of the rows has attained m non-constant polynomials). The global variable k loops over indices in $queue_{j'}$, and as above, the increment it receives arranges for the processors to compute almost an equal number of polynomials P_k in $queue_{j'}$. The processors receive information about the polynomials they should compute through the global data found in $Type = Type_{j'}[k]$, $n = Row_{j'}[k]$, and $i = Index_{j'}[k]$, and call the sub-routine $Compute\_Poly$ which determines the polynomial P_k as defined in the Göttfert setting. If P_k is non-constant, processor id stores it permanently in $Poly_{(k+total\_poly)}$, and sets $D[n][i]$ (or $R[n][i]$) to point to the address of $Poly_{(k+total\_poly)}$. Because $total\_poly$ represents the total number of non-constant factors computed so far, this particular index of $Poly$ is such that all new polynomials do not overwrite previous ones, and no two processors store their results in the same location. The local value of the polynomial and its pointer are then broadcast by processor id to all processors, and the total number of non-constant factors found along row n by processor id during the set up of $queue_{j'}$ is increased by 1 in the processor's local copy of $sum[n]$ (it is assumed that sum and $local\_sums$ are initialised to zero before every new iteration of the main loop of step 10). When all polynomials in $queue_{j'}$ have been computed, each processor id places its own copies of $sum[n]$, for $n = 2, ..., m$ in global locations at $local\_sums[id][n]$. A synchronisation barrier is now met, which updates the values of the non-constant polynomials, their pointers, and the partial lengths of rows as computed by every individual processor. We note the absence of a synchronisation point immediately after the broadcasting of $Poly_{(k+total\_poly)}$ and the pointer to it, due to the fact that they were not needed in any computation within the loop of step 11. We also note that, although updating the total row lengths inside the loop of step 11 (i.e. while processors are still operating within the same parallel queue) definitely discards any unnecessary gcd or division operations remaining in the queue, our choice not to perform accordingly can be justified by the fact that this will require a synchronisation point within the innermost loop, one whose repeated application could prove to be costly. Each processor now has all the partial sums available to it globally in $local\_sums[id][n]$ and can thus sum them all up into one global quantity in $Length[n]$. The total number of non-constant factors is also updated as being the sum of all row lengths. If any row length becomes equal to m, all processors are signalled to

stop. Else, $queue_{j'}$ is transferred onto $queue_j$ (so that the most recent polynomials can help determine what the new parallel queue will be), and $queue_j$ is sorted through a call to *Sort*. Since some processors compute constant polynomials whose index k leaves the corresponding location in the array *Poly* empty, the *Sort* sub-routine re-arranges the elements stored in *Poly* (and their corresponding pointers in the arrays D or R) so that the non-constant factors are stored consecutively after each other. *Sort* also returns the length of the sorted list. Finally, a new $queue_{j'}$ is set according to Algorithm 5.2.1. The loop of step 10 can be shown to end, since we are bound to reach a row containing m non-constant polynomials which constitute all the irreducible factors of f (Theorem 5.2.1). At this point, $last\_row$ contains the index to that row. All processors scan in parallel the non-constant D and R pointers to the polynomials found along $last\_row$ (using our notation, the polynomials are accessed by applying $*$ to a particular location in the D or R arrays). By Claim 5.2.1, there is a maximum of $\#r_{last\_row-1} = (2^{last\_row-1} - 1)$ D-polynomials and $\#r_{last\_row-1} + 1 = 2^{last\_row-1}$ R-polynomials in $last\_row$, which by Theorems 3.1.7 and 3.1.8 constitute the m non-constant factors of f. Each processor then determines the multiplicity of that factor in f (by a call to the sub-routine *Multiplicity*). At this stage, we can choose not to distribute the results globally so that each processor outputs its own set of $(factor, exp)$ pairs. The algorithm terminates with the last iteration of this loop.

5.2.3 The BSP cost of the algorithm

In this section we establish the parallel complexity of our algorithm. To this end, we first state and prove several preliminary results.

Lemma 5.2.1 *In the parallel setting described in Algorithm 5.2.1, every row n has its first element $n; D_1$ computed in the parallel queue n and its last element $n; R_{\#r_{n-1}+1}$ computed in the parallel queue $2n-1$.*

Proof: We prove the result by induction on n. For $n = 2$, we know that queue 2 starts with $2; D_1$, and by Algorithm 5.2.1, queue 3 contains the polynomials $3; D_1$, $2; R_1$ and $2; R_2$, where $2; R_2$ is the last polynomial to be computed in row 2. Suppose that $n; D_1$ can be first computed in queue n. Since $(n+1); D_1 = \gcd(b_{n+1}, n; D_1)$, the first queue which assigns the computation of $(n+1); D_1$ is $n+1$. Furthermore, suppose that queue $2n-1$ contains the polynomial $n; R_{\#r_{n-1}+1}$ which is computed last in row n. Since

$$(n+1); D_{\#r_n} = \gcd(b_{n+1}, n; R_{\#r_{n-1}+1}),$$

this polynomial can be determined at the earliest in the parallel queue $2n$. But $(n+1); R_{\#r_n+1}$ depends on the values of all polynomials $(n+1); D_i$, for $i = 1, ..., \#r_n$, and hence can be can be computed at the earliest when $(n+1); D_{\#r_n}$ is available, which is in the parallel queue $2n+1$.

Corollary 5.2.1 *It takes at most $2m-1$ parallel queues for a complete factorisation into irreducibles to be established.*

Proof: By Theorem 3.1.8 it takes at most m rows to compute all irreducible factors of f (see [59] for proof). By Lemma 5.2.1, row m requires at most $2m-1$ parallel queues before all non-constant polynomials appearing in it are computed. This concludes the proof.

Lemma 5.2.2 *If n is odd, then queue n contains polynomials belonging only to rows $(n+1)/2+j$, for $j = 0, ..., (n-1)/2$, if $2 \leq n \leq m$, and for $j = 0, ..., m-(n+1)/2$, if $m < n \leq 2m-1$. Else, if n is even, then queue n contains polynomials belonging only to rows $n/2+1+j$, for $j = 0, ..., n/2-1$, if $2 \leq n \leq m$, and for $j = 0, ..., m-(n/2+1)$, if $m < n \leq 2m-1$.*

Proof: For all queues n appearing in the parallel set-up, Corollary 5.2.1 maintains that $1 \leq n \leq 2m-1$. Suppose now that n is odd. Let k be a row occupying queue n. By Lemma 5.2.1, we must have $n \leq 2k-1$ (or $k \geq (n+1)/2$). Write $k = (n+1)/2 + j$ for $j \geq 0$ (since n is odd, this expression is an integer). For $n = 2, ..., 2m-1$, we must have $k \leq n$ (so that $j \leq (n-1)/2$); otherwise, row k starts appearing in queues $n+1$ onwards, a contradiction. If $m < n \leq 2m-1$, the upper bound on k can be strengthened to satisfy $k \leq m$ (or $j \leq m-(n+1)/2$), since the last row to be set up in the Göttfert representation is row m.

If n is even, $n \leq 2k-1$ implies that $n \leq 2k-2$ since $2k-1$ is odd, or that $k \geq n/2+1$. Write $k = n/2+1+j$ for $j \geq 0$ (again, this expression is an integer since n is even). A similar argument as above follows to establish the upper bounds on j, and we leave the straightforward proof to the reader.

Lemma 5.2.3 *Each parallel queue consists of at most $2m$* gcd *and division operations and contributes to at most m non-constant polynomials.*

Proof: First, we note that, since each row k in the Göttfert representation requires at most $2m$ gcd and division computations and has at most m non-constant polynomials appearing in it, and since each such row requires a number of at most k queues to be fully set up, we would expect, roughly and on average, each parallel queue n to require a number of $2m/k$ gcd and division operations leading to about m/k non-constant polynomials for each row k assigned to queue n. Let n be odd (the case when n is even can be proven similarly and hence can be omitted). If $2 \leq n \leq m$, Lemma 5.2.2 implies that queue n has polynomials belonging to rows $k = (n+1)/2 + j$, for $j = 0, ..., (n-1)/2$, where each row k has roughly m/k non-constant polynomials appearing in queue n. Thus, the total number of gcd and division operations to be performed in queue n is approximately

$$\sum_{j=0}^{(n-1)/2} \frac{2m}{\frac{n+1}{2}+j} < \left(\frac{n-1}{2}+1\right)\frac{2m}{(n+1)/2} = 2m.$$

Furthermore, if $m < n \leq 2m-1$, this number is approximately

$$\begin{aligned}\sum_{j=0}^{m-(n+1)/2} \frac{2m}{\frac{n+1}{2}+j} &< \left(m-\frac{n+1}{2}+1\right)\frac{2m}{(n+1)/2} \\ &< (m-\frac{m+1}{2}+1)\frac{2m}{(m+1)/2} = 2m.\end{aligned}$$

Using a very similar calculation, the total number of non-constant polynomials produced is easily shown to be at most m.

Since the number of processors will be fixed throughout the text, we shall refer to the communication and synchronisation BSP parameters as simply g and ℓ respectively, where it is implicitly understood that the two parameters depend on p.

Theorem 5.2.2 *Assuming classical polynomial arithmetic with multiplication time $M(d) = O(d^2)$, the BSP cost of Algorithm 5.2.2 is of the order*

$$O\left(\frac{m^2}{p}M(d)\log d + gm^2(\left\lceil\frac{d}{w_l}\right\rceil + p) + m\ell\right) \textit{ flops.} \tag{5.1}$$

Proof: In our proof, we note the following remarks. Since all polynomials appearing in the course of the algorithm are factors of f, their degrees are at most equal to $d = \deg(f)$. It is also understood that any computational complexity is the maximum work load achieved by any one processor. Sorting a list of size at most k can be achieved in $O(k \log k)$ flops (e.g. see [26]). Computing the multiplicity of a factor of f requires at most d polynomial multiplications over $\mathbb{F}_2$ (and hence in our case, $M(d)$ flops). We also assume that accessing an array entry requires almost as much time as one flop. A *message* denotes one computer word (i.e. a message of size 1). The total cost of the algorithm is the summation of the costs of its supersteps. The individual BSP costs of the main supersteps (i.e. those whose cost is not constant) can be detailed as follows:

Supersteps 2-4: we have seen earlier that this loop is divided almost equally among all processors. As a result, the parallel loop is accessed at most $\lceil m/p \rceil$ times. Each iteration of the outer loop involves mainly one gcd and one division computation, and an inner loop consisting of a *bsp_put* operation, whereby each processor sends $\lceil degree/w_l \rceil \leq \lceil d/w_l \rceil$ messages to all processors (and hence p copies of these) and receives $p\lceil degree/w_l \rceil \leq p\lceil d/w_l \rceil$ messages. Thus, $h_{max} = p\lceil degree/w_l \rceil < p\lceil d/w_l \rceil$. Superstep 5 is a synchronisation point, so that the total BSP cost of supersteps 2-4 is at most

$$O\left(\left\lceil\frac{m}{p}\right\rceil\left[M(d)\log d + gp\left\lceil\frac{d}{w_l}\right\rceil\right] + \ell\right) \text{ flops.} \tag{5.2}$$

Supersteps 6-9: consist mainly of one gcd operation and a call to *Set_Queues* with input $queue_j = \{P_0\}$, which according to Theorem 5.2.1 requires at most 3 steps. Thus, the BSP cost of these supersteps is of the order

$$O(M(d)\log(d)) \text{ flops.} \tag{5.3}$$

Supersteps 10-28: constitute the main body of the parallel algorithm. The outer–most loop designates the total number of times the processors set up parallel queues before a complete factorisation is achieved. This number has been shown in Corollary 5.2.1 to be at most $2m - 1$. Supersteps 11-19 are embedded within an inner loop ranging over all polynomials along $queue_{j'}$. The tasks within the queue, consisting mainly of gcd and division operations, are divided almost equally among processors (check the initial value of the loop variable k and the increment it receives). By Lemma 5.2.3, each queue consists of at most $2m$ gcd and division operations to perform equally among all processors, and hence the loop is accessed at most $\lceil 2m/p \rceil$ times. The bulk of the work appears in the following: step 11 consists of either a gcd or a division operation, step 16 consists of sending at most $p\lceil d/w_l \rceil$ messages and receiving at most $p\lceil d/w_l \rceil$ messages (resulting in $h_{max} = p\lceil d/w_l \rceil$), and step 17 or 18 consists of sending p messages and receiving one message (resulting in $h_{max} = p$). Within each inner loop iteration, the BSP cost of supersteps 11-19 is at most

$$O\left(M(d)\log(d) + gp\left\lceil\frac{d}{w_l}\right\rceil\right)$$

which across both the outer and inner loops becomes of the order

$$O\left(m\left\lceil\frac{m}{p}\right\rceil\left[M(d)\log(d)+gp\left\lceil\frac{d}{w_l}\right\rceil\right]\right)\text{ flops.}\tag{5.4}$$

Supersteps 20-28 are found outside the inner loop of step 11 but inside the loop of step 10. In the communication superstep 20 each processor sends pm messages and receives pm messages (so that $h_{max}=pm$). Superstep 21 is a synchronisation point, and superstep 22 consists of about $2pm$ additions. Superstep 24 consists mainly of a call to *Sort* with input list of size at most $2m$ (producing a list of size at most m by Lemma 5.2.3), as well as a call to *Set_Queues* with input list of size at most m (this requiring at most $3m$ flops). Summing up, a single application of supersteps 20-28 requires at most

$$O\left(m(p+\log m)+gpm+\ell\right)$$

flops which when iterated across the outer loop of step 10 becomes of the order

$$O(m^2(p+\log m)+gpm^2+m\ell)\text{ flops.}\tag{5.5}$$

Supersteps 25-28 consist of computing the multiplicity of all m factors in parallel, where each processor takes up almost an equal number of D and R-factors. If each call to *Multiplicity* requires about $M(d)$ flops, the supersteps will require

$$O\left(\left\lceil\frac{m}{p}\right\rceil M(d)\right)\text{ flops.}\tag{5.6}$$

Summing up the individual costs (5.2), (5.3), (5.4), (5.5), and (5.6), the total BSP cost can be found to be of the order

$$O\left(m\left\lceil\frac{m}{p}\right\rceil M(d)\log d+m^2(p+\log m)+gp(m\left\lceil\frac{d}{w_l}\right\rceil\left\lceil\frac{m}{p}\right\rceil+m^2)+m\ell\right)$$

flops. Since

$$\left\lceil\frac{m}{p}\right\rceil<\frac{m}{p}+1, p=O(d), M(d)=O(d^2)\text{ and }m=O(d),$$

(where the second inequality easily holds in implementations involving large polynomial degrees – see Table 5.2, the third estimate holds in our implementations of classical polynomial arithmetic [55]), it follows that

$$m^2(p+\log m)=O(m^2d)=O\left(m\left\lceil\frac{m}{p}\right\rceil M(d)\log d\right),$$

from which the total BSP cost (5.1) can be derived.

Corollary 5.2.2 *Assuming the given in Theorem 5.2.2 above, Algorithm 5.2.2 has low synchronisation and communication requirements.*

Proof: We claim that our algorithm has very good synchronisation and communication requirements, in that the number of flops required by both can be negligible compared to the computation cost. In particular, and based on the values of the BSP parameters in table 5.2, it can be easily attained that

$$g = O(\frac{d \log d}{p}), \quad p = O(d), \text{ and } \ell = O(\frac{m}{p} M(d) \log d)$$

for large values of d, so that

$$gm^2 \left(\left\lceil \frac{d}{w_l} \right\rceil + p \right) + m\ell = O(\frac{m^2}{p} M(d) \log d).$$

5.2.4 Reduction of the algorithm's memory requirements

As defined in Algorithm 5.2.2, the two dimensional arrays D and R of size $m \times (2^m - 1)$ each can be easily seen to constitute an infeasible (exponential) space requirement unless m is very small. To this end, we describe how a linked-list structure can be adopted which reduces the memory requirements to four integer arrays of size m each and four integer arrays of size $2m^2$ each, requiring only a polynomial order space complexity.

Recall that, for $n = 2, ..., m$ and $i = 1, ..., 2^m - 1$, $D[n][i]$ represents a pointer which is NULL if $n; D_i$ is a constant polynomial, and which points to the location of $n; D_i$ in the array $Poly$, otherwise. Similarly for $R[n][i]$. We also note that there are many more constant polynomials than there are non-constant ones, and hence, the distribution of non-constant polynomials among all possibilities is a sparse one. Our algorithm in its present form contains many NULL pointers, and a candidate for a more efficient method has to replace this structure with one which locates only non-constant polynomials without any reference to the others.

The improvement can be described as follows. We first order the non-constant factors in a list $\mathcal{F}$, in which each factor occupies an index corresponding to its position in the array $Poly$. For instance, if a non-constant factor is stored at location k in $Poly$, it would appear as the $k+1$ polynomial in $\mathcal{F}$. In this way, a non-constant factor in the entire collection can be compared to a nonzero entry in a collection of sparse row vectors (or columns) representing a sparse matrix.

We define four global integer arrays, D_header, D_tail, R_header and R_tail, each of size m, and four global integer arrays, D_fwd_link, R_fwd_link, D_index and R_index, each of size $2m^2$. The aim would be to arrange polynomials in $\mathcal{F}$ in lists of polynomials of the same type and row. The description below concerns only D-arrays but a very similar one holds for R-arrays.

Suppose we want to link all D- polynomials in $\mathcal{F}$ appearing in row n. For $n = 2, ..., m$, $D\_header[n]$ represents the index in $\mathcal{F}$ of the first (non-constant) D-polynomial appearing in row n. All ensuing non-constant D-polynomials in row n are adjoined to the list, and as such, we need to preserve various information describing each one of them as they appear together. For each row n, we keep track of the index in $\mathcal{F}$ of the last D-polynomial appearing in the row by storing it in $D\_tail[n]$. Thus, for a start, if one non-constant D-polynomial of row n appears in $\mathcal{F}$ at location k, we set $D\_header[n] = k$ and $D\_tail[n] = k$. We also store its index along row n in $D\_index[k]$, which completes all information about the polynomial. For each non-constant

D-polynomial in row n and of index k in $\mathcal{F}$, we maintain a pointer to the index in $\mathcal{F}$ of the next such polynomial and store it in $D\_fwd\_link[k]$. If $D\_fwd\_link[k] = 0$, polynomial k is the last one in the list of non-constant D-polynomials belonging to row n, which can be generated completely as follows:

Algorithm 5.2.3 *Input: A non-constant D-polynomial of index i in some row $n = 2, ..., m$ and index k in $\mathcal{F}$.*
Output: The polynomial adjoined to the end of the list of non-constant D-polynomials belonging to row n.
if $(D\_header[n] = 0)$ *do*
1. $D\_header[n] \leftarrow k$, $D\_tail[n] \leftarrow k$, $D\_index[k] \leftarrow i$, $D\_fwd\_link[k] \leftarrow 0$;
else do
2. $D\_fwd\_link[D\_tail[n]] \leftarrow k$, $D\_tail[n] \leftarrow k$, $D\_index[k] \leftarrow i$;
end.

It is trivial to see that the call to this algorithm comes at a very negligible constant cost and hence can be embedded within the estimate of (5.1). We now illustrate the use of this structure in locating non-constant factors whenever required for new computations. Suppose, for instance, that we need to compute $n; D_i$ for some $n = 3, .., m$ and $i = 1, ..., (\#r_{n-1} - 1)/2$ (so that $n; D_i = \gcd(b_n, (n-1); D_i)$). Using the two dimensional array D, and if $D[n-1][i]$ is NULL, one concludes that $n; D_i$ is constant; otherwise, the polynomial pointed to by $D[n-1][i]$ is used to compute the required gcd. Using the linked list structure, the process can be described as follows:

Algorithm 5.2.4 *Input: A D-polynomial of index i in some row $n-1$, $n = 3, ..., m$.*
Output: The location of $(n-1); D_i$ in the array Poly if it is non-constant or FAIL otherwise.
1. $t \leftarrow FAIL$, $x \leftarrow D\_header[n-1]$;
if $(D\_header[n-1] \neq D\_tail[n-1])$ *do*
  while $(x \neq 0$ *and* $t = FAIL)$ *do*
    if $(D\_index[x] = i)$ *do*
2. $t \leftarrow PASS$;
3. $k \leftarrow x$, $x \leftarrow D\_fwd\_link[x]$;
    end;
  end;
end;
else do
  if $(D\_index[x] = i)$ *do*
4. $t \leftarrow PASS$;
  end;
5. $k \leftarrow x$;
end;
if $(t = PASS)$, *return* $(k-1)$;
else return FAIL.

If $D\_header[n-1] \neq D\_tail[n-1]$, the list of D-polynomials belonging to row n contains more than one polynomial and hence has to be scanned entirely; else, the list contains only one element. x takes up indices in this list, starting with its header, and moving across the forward links. t is an indicator which when set to $PASS$ indicates that a non-constant polynomial

$(n-1)$; D_i has been found whose location in $\mathcal{F}$ is k (or location in $Poly$ is $k-1$). If x becomes zero, this signals the end of the list. Finally, the loop across the list ends either when t becomes PASS or $x = 0$. The algorithm demonstrates how checking $D[n][i]$ can be substituted with scanning a list of size at most m. The increase in the total computational cost as a result of using the improved data structure can be realised as follows. We have seen that the call to $Compute\_Polynomial$ in step 11 of algorithm 5.2.2 is issued at most $2m \cdot \lceil m/p \rceil$ times. Each such polynomial computation will require two calls to algorithms 5.2.3 and 5.2.4, thus increasing the total cost of Algorithm 5.2.2 by $O(2m^2 \cdot \lceil m/p \rceil)$ flops. Since $m = O(M(d) \log d)$, the upper bound estimate given in (5.1) remains of the same order, and as such, the new improvement can be introduced at little cost to the worst-case analysis initially provided.

5.3 Implementation and run times

Table 5.1: Parallel run times.

				Processors				
d	m	max	seq.	1	2	4	8	16
8000	6	7446	2.37	2.32 (1)	1 (1.2)	0.8 (0.7)	0.2 (1.5)	0.6 (0.2)
8000	7	3200	2.62	2.6 (1)	0.9 (1.5)	0.7 (0.9)	1.4 (0.2)	1.1 (0.1)
8000	30	600	48.98	48.36 (1)	27 (0.9)	17.4 (0.7)	6.8 (0.9)	6.6 (0.5)
16000	10	10224	29.9	29.9 (1)	14.2 (1.1)	10.4 (0.7)	10.4 (0.4)	3.7 (0.5)
16000	14	5600	8.78	8.74 (1)	4.7 (0.9)	2.4 (0.9)	2.6 (0.4)	1.4 (0.4)
32000	31	19360	145.8	144.89 (1)	76.4 (1)	31.5 (1.2)	13.8 (1.3)	10.4 (0.9)
64000	10	26400	82.6	82.01 (1)	64.2 (0.6)	29.3 (0.7)	27 (0.4)	29.05 (0.2)
128000	14	44800	2629.62	2609 (1)	794.8 (1.7)	761.4 (0.9)	760.6 (0.4)	468.8 (0.4)
256000	14	92800	6819.12	6761.89 (1)	2696.3 (1.3)	1420.4 (1.2)	1417 (0.6)	325.1 (1.3)
300000	11	120000	621.8	621.32 (1)	408.68 (0.8)	242.2 (0.6)	51.2 (1.5)	57.6 (0.7)
400000	11	160000	1658.2	1658 (1)	829 (1)	592 (0.7)	188.4 (1.1)	148 (0.7)

Table 5.2: BSP parameters.

p	g	ℓ
1	0.34	55
2	1.64	1496
4	2.48	1683
8	2.34	2562
16	4.83	3431

All programs were written in C and extended using the standard BSP library [66, 67]. The work was carried out at the Oxford University Supercomputing Centre (OSC) using the Oswell machine. Oswell is a Sun cluster of 84 processors and a shared memory system where the processors are arranged in three groups of 24 processors and a group of 12 processors (each processor having 2 GBytes of memory). With this scheme, any work submitted to the machine is queued to one of those four boxes, and the number of processors available for use by any one job is at most 24. In practice, however, we had access to 16 processors only.

The various input data were taken from the results of Chapter 4 where the Niederreiter linear system for a trinomial over $\mathbb{F}_2$ is solved and a basis for the solution set is produced. We remark that, in spite of the input polynomial being sparse, the intermediary polynomials in the Göttfert representation are not necessarily so, which renders such a case study for trinomials not so special a case as it appears. The run times in table 5.1 represent the times in seconds for producing the entire factorisation using our BSP parallel algorithm given a particular basis set. The timings correspond to the sequential as well as the parallel results. max represents the maximum over all $i = 1, ..., m$ of $deg(g_i^{e_i})$, and d and m are as defined previously. The absolute efficiencies are shown in parentheses. All polynomial arithmetic was performed using classical algorithms so that the multiplication time for polynomials of degree at most d is $O(d^2)$ [55].

Let T_p and T_s denote the parallel run time using p processors and the sequential run time respectively. Our run times suggest a speed gain in almost all cases, an outcome that is to be expected according to our BSP cost (5.1), which roughly suggests that

$$T_p < T_s \text{ iff } \frac{m^2}{p} < m^2 \text{ iff } p > 1,$$

ignoring the negligible communication and synchronisation requirements of our algorithm as well as any other parallel overheads associated with the creation and management of threads. To measure the scalability of our parallel algorithm, we calculate the absolute efficiency E_p [86] for all cases, where

$$E_p = \frac{T_s}{pT_p},$$

which when achieving values close to one indicates a good parallel performance. Accordingly, and upon examining our efficiencies, we note that almost all our experiments scale very well for up to 8 processors. Thereafter, the efficiency remains very good for a fixed d either as m increases (e.g. compare trinomials with $m > 30$ to others), or as max increases (for $d = 8000$, compare the trinomials with $max = 7446$ and $max = 3200$). We also note that efficiency remains almost constant around 1 for $d \geq 256000$. We remark the absence of a sharp fluctuation in the efficiency levels mainly because our algorithm does not involve data partitioning (but only task parallelism), which results in the computation being either entirely *in cache* or *out of cache* across all processors for the same d. This has the advantage of revealing the real scalability of the algorithm and avoiding cache effects.

5.4 Conclusion

In this chapter we presented and analyzed a complete BSP algorithm for extracting the factors of a polynomial over $\mathbb{F}_2$ using the Göttfert refinement of the Niederreiter algorithm, which,

given a basis for the solution set of the Niederreiter linear system, performs the last phase of the factorisation algorithm in polynomial time. Our BSP theoretical model resulted in an efficient BSP cost requiring relatively small communication and synchronisation costs. The parallel algorithm not only achieves considerable speed gains as the number of processors increases up to 16, but maintains a moderate to very good efficiency that is better maintained as the degree of the polynomial, the number of its irreducible factors or the maximum over its irreducible factors' degrees increases. The algorithm can be applied over fields of characteristic 2 in general, provided an input basis is available. When combined with our work in Chapter 4 which exploits sparsity in the Niederreiter linear system, the hybrid algorithm provides a cheaper and more memory efficient alternative to the factorisation of trinomials over $\mathbb{F}_2$ than the implementation in [110], which uses dense explicit linear algebra and a maximum of 256 nodes to achieve a polynomial record of degree 300000. When compared with the Black Box Niederreiter algorithm of [39], the hybrid algorithm is a simpler approach for moderately high record factorisations of trinomials over $\mathbb{F}_2$ as those allowed by the use of implicit linear algebra, requiring reasonable running times (see Chapter 4). Apart from the significance of its experimental results, our algorithm provides a good model of how parallelism in general, and the BSP model in particular, can be incorporated elegantly and successfully into problems in symbolic computation.

Chapter 6

Factoring polynomials via polytopes

6.1 Introduction

This chapter is based on joint work with Shuhong Gao and Alan Lauder [2]. As mentioned in Chapter 1, factoring multivariate polynomials is a fundamental problem in all major computer algebra systems. There is an extensive literature on this problem — we refer the reader to the references in [23, 44, 52, 63, 75, 78, 88, 89, 90, 101, 131, 130]. Most of these papers deal with dense polynomials, two notable exceptions being [52, 78]. These two papers reduce sparse polynomials with more than two variables to bivariate or univariate polynomials which are then treated as dense polynomials. It is still open whether there is an efficient algorithm for factoring sparse bivariate or univariate polynomials. The goal in this chapter is to study sparse bivariate polynomials using their connection to integral polytopes.

Newton polytopes of multivariate polynomials reflect to a certain extent the sparsity of polynomials and they carry a lot of information about the factorisation patterns of polynomials as demonstrated in the recent work of Gao [43] and Gao and Lauder [45]. In this chapter the focus is on the more difficult problem of factoring sparse polynomials. We do not solve this problem completely. However, our approach is a practical new method which generalises Hensel lifting; its running time will in general improve upon that of Hensel lifting and sparse bivariate polynomials can often be processed significantly more quickly. As with Hensel lifting, it has an exponential worst-case running time. Also, our method does not work for all polynomials, but only for those that are square-free on certain subsets of the edges of their Newton polytopes (see Theorem 6.6.1).

In Section 6.2 we present a brief introduction to Newton polytopes and their relation to multivariate polynomials, and in Section 6.3 we state the central problem. Section 6.4 contains an outline of our method, and highlights the theoretical problems we need to address. The main theorem underpinning our method is proved in Section 6.6, after a key geometric lemma in Section 6.5. Section 6.8 contains a detailed description of the algorithm. Finally in Section 6.9 we present a small example, as well as details of our computer implementation of the algorithm.

6.2 Newton polytopes and Ostrowski's theorem

This chapter considers polynomial factorisation over a field $\mathbb{F}$ of arbitrary characteristic. We denote by $\mathbb{N}$ the *non-negative* integers, and $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the integers, rationals and reals. For

an earlier introduction on polytopes and polynomials we refer the reader to chapter 3.

Let $\mathbb{F}[X_1, X_2, \ldots, X_n]$ be the ring of polynomials in n variables over the field $\mathbb{F}$. We recall the motivating theorem behind our investigation:

Theorem 6.2.1 (Ostrowski) *Let $f, g, h \in \mathbb{F}[X_1, \ldots, X_n]$. If $f = gh$ then* $\mathrm{Newt}(f) = \mathrm{Newt}(g) + \mathrm{Newt}(h)$.

An immediate result of this theorem relates to testing polynomial irreducibility: In the simplest case in which the polytope does not decompose, one immediately deduces that the polynomial must be irreducible. This was explored in [43, 45, 47]. In this chapter, we address the more difficult problem: Given a decomposition of the polytope, how can we recover a factorisation of the polynomial whose factors have Newton polytopes of that shape, or show that one does not exist?

In the remainder of this chapter, we restrict our attention to bivariate polynomials, and f always denotes a bivariate polynomial in the ring $\mathbb{F}[x, y]$. For $e = (e_1, e_2) \in \mathbb{N}^2$, we redefine the notation X^e to mean $x^{e_1} y^{e_2}$.

6.3 Extending partial factorisations

Let $\mathrm{Newt}(f) = Q + R$ be a decomposition of the Newton polytope of f into integral polygons in the first quadrant. For each lattice point $q \in Q$ and $r \in R$ we introduce indeterminates g_q and h_r. The polynomials g and h are then defined as

$$\begin{aligned} g &:= \textstyle\sum_{q \in Q} g_q X^q \\ h &:= \textstyle\sum_{r \in R} h_r X^r. \end{aligned}$$

We call g and h the *generic* polynomials given by the decomposition $\mathrm{Newt}(f) = Q + R$. Let $\#\mathrm{Newt}(f)$ denote the number of lattice points in $\mathrm{Newt}(f)$. The equation $f = gh$ defines a system of $\#\mathrm{Newt}(f)$ quadratic equations in the coefficient indeterminates obtained by equating coefficients of each monomial X^e with $e \in \mathrm{Newt}(f)$ on both sides. The aim is to find an efficient method of solving these equations for field elements. Our approach, motivated by Hensel lifting, is to assume that, along with the decomposition of the Newton polytope, we are given appropriate factorisations of the polynomials defined along its edges. This "boundary factorisation" of the polynomial is then "lifted" into the Newton polytope, and the coefficients of the possible factors g and h revealed in successive layers. Unfortunately, to describe the algorithm properly we shall need a considerable number of elementary definitions — the reader may find the figures in Section 6.9.1 useful in absorbing them all.

Let S be a subset of $\mathrm{Newt}(f)$. An *S-partial factorisation* of f is a specialisation of a subset of the indeterminates 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 specialisation is just a substitution of field elements in place of indeterminates.) The case $S = \mathrm{Newt}(f)$ is equivalent to a factorisation of f in the traditional sense, and we will call this a *full factorisation*. Now suppose we have an S-partial factorisation and an S'-partial factorisation. Suppose further $S \subseteq S'$ and the indeterminates specialised in the S-partial factorisation have been specialised to the same field elements as the corresponding ones in the S'-partial factorisation. Then we say the S'-partial factorisation *extends* the S-partial factorisation. We call this extension *proper* if S' has strictly more lattice points than S.

Let $\mathrm{Edge}(f)$ denote the set of all edges of $\mathrm{Newt}(f)$. Each edge $\delta \in \mathrm{Edge}(f)$ 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 $d = \gcd(u_2 - u_1, v_2 - v_1)$, $u_0 = (u_2 - u_1)/d$, and $v_0 = (v_2 - v_1)/d$. Then (u_0, v_0) represents the direction of δ and the integral points on δ are of the form

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

Assuming the Euclidean plane is endowed with an orthonormal system of coordinates, let $(\nu_1, \nu_2) := (-v_0, u_0)$ be a rotation of (u_0, v_0) by 90 degrees counter clockwise. For any edge δ of $\mathrm{Newt}(f)$, all integral points of $\mathrm{Newt}(f)$ lying on the left hand side of δ are of the form

$$(u_1, v_1) + i(u_0, v_0) + j(-v_0, u_0), \quad \text{for some integers } j \geq 0, i \in Z.$$

Let $\eta = v_0 u_1 - u_0 v_1$. Define

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

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 $\ell = 0$. We call this function ℓ *the primitive affine function* associated with δ, denoted by ℓ_δ.

The function ℓ_δ has another nice property: 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

$$z := x^{\nu_2} y^{-\nu_1} \text{ and } w := x^{\zeta_1} y^{\zeta_2}. \tag{6.1}$$

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 remains 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 $\delta = \delta' + \delta''$, and the lines supporting the edges δ, δ' and δ'' are parallel (see [36] for instance). One can also show that there exists a unique integer c_δ such that

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

where η is the constant coefficient of ℓ_δ. In particular, let c_δ be the unique positive integer such that

$$\delta' \;=\; \{r \in Q \,|\, l_\delta(r) = c_\delta\}.$$

For any $r'' \in \delta''$, we know that $r'' = r - r'$ for some $r \in \delta$ and $r' \in \delta'$. Write $r = r' + r''$ for $r' = (r'_1, r'_2) \in \delta'$ and $r'' = (r''_1, r''_2) \in \delta''$. Then

$$\begin{aligned} l_\delta(r) &= \nu_1(r'_1 + r''_1) + \nu_2(r'_2 + r''_2) + \eta \\ &= \nu_1 r'_1 + \nu_2 r'_2 + \nu_1 r''_1 + \nu_2 r''_2 + \eta \\ &= l_\delta(r') + \nu_1 r''_1 + \nu_2 r''_2 \\ &= c_\delta + \nu_1 r''_1 + \nu_2 r''_2. \end{aligned}$$

But $l_\delta(r) = 0$, and so

$$\nu_1 r''_1 + \nu_2 r''_2 + \eta = \eta - c_\delta$$

which gives $l_\delta(r'') = \eta - c_\delta$. We then have

$$\delta'' \;=\; \{r \in R \,|\, l_\delta(r) = -c_\delta + \eta\}.$$

Let $\Gamma \subseteq \mathrm{Edge}(f)$, and let $K = (k_\gamma)_{\gamma\in\Gamma}$ be a vector of positive integers labelled by Γ. Define

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

This defines a strip along the interior of $\mathrm{Newt}(f)$, or a union of such strips.

We denote by $Q|_{\Gamma,K}$ and $R|_{\Gamma,K}$ the subsets of Q and R respectively given by

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

Once again these denote strips along the inside of Q and R whose sum contains the strip $\mathrm{Newt}(f)|_{\Gamma,K}$ in $\mathrm{Newt}(f)$.

We now come to the main definition of this section.

Definition 6.3.1 *A $\mathrm{Newt}(f)|_{\Gamma,K}$-factorisation is called a $(\Gamma, K; Q, R)$-factorisation if the following two properties hold:*

- *Exactly the indeterminate coefficients of g and h indexed by lattice points in $Q|_{\Gamma,K}$ and $R|_{\Gamma,K}$, respectively, have been specialised.*
- *Let $K' = (k'_\gamma)_{\gamma\in\Gamma}$ be a vector of positive integers with $k'_\gamma \ge k_\gamma$ 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.*

The second property will be used only once, in the proof of Lemma 6.6.1.

In most instances Q, R and Γ will be clear from the context. If so we will omit them and refer simply to a K-factorisation. Furthermore, if K is the all ones vector, denoted $(\underline{1})$, of the appropriate length indexed by elements of some set Γ, then we call this a $(\Gamma; Q, R)$-*boundary factorisation*. We shall simplify this to *partial boundary factorisation* or $(\underline{1})$-*factorisation* when

Γ, Q and R are evident from the context. This special case will be the "lifting off" point for our algorithm.

The central problem we address is

Problem 6.3.1 Let $f \in \mathbb{F}[x,y]$ have Newton polytope $\mathrm{Newt}(f)$ and fix a Minkowski decomposition $\mathrm{Newt}(f) = Q + R$ where Q and R are integral polygons in the first quadrant. Suppose we have been given a $(\Gamma; Q, R)$-boundary factorisation of f for some set $\Gamma \subseteq \mathrm{Edge}(f)$. Construct a full factorisation of f which extends it, or show that one does not exist.

Moreover, one wishes to solve the problem in time bounded by a small polynomial function of $\#\mathrm{Newt}(f)$.

6.4 The polytope method

6.4.1 An outline of the method

We now give a basic sketch of our polytope factorisation method for bivariate polynomials. Throughout this section Γ will be a fixed subset of $\mathrm{Edge}(f)$ and $\mathrm{Newt}(f) = Q + R$ a fixed decomposition. We shall need to place certain conditions on Γ later on, but for the time being we will ignore them. Since Γ, Q and R are fixed we shall use our abbreviated notation when discussing partial factorisations.

We begin with $K = (\underline{1})$ the all-ones vector of the appropriate length and a K-factorisation (partial boundary factorisation). Recall this is a partial factorisation in which exactly the coefficients in the sets $Q|_{\Gamma,K}$ and $R|_{\Gamma,K}$, subsets of points on the boundaries of Q and R, have been specialised.

At each step of the algorithm we either show that no full factorisation of f exists which extends this partial factorisation, and halt, or that at most one can exist, and we find a new K'-factorisation with $K' = (k'_\delta)$ such that

$$\sum_{\delta \in \Gamma} k'_\delta > \sum_{\delta \in \Gamma} k_\delta.$$

(Usually the sum will be incremented by just one.) Iterating this procedure either we halt after some step, in which case we know that no factorisation of f exists which extends the original partial boundary factorisation, or we eventually have $\mathrm{Newt}(f) \subseteq \mathrm{Newt}(f)|_{\Gamma,K}$, for the updated K (or just $Q \subseteq Q|_{\Gamma,K}$ or $R \subseteq R|_{\Gamma,K}$ will do). At that point all of the indeterminates in our partial factors have been specialised, and we may check to see if we have found a pair of factors by multiplication. (In the case, say, that just $Q \subseteq Q|_{\Gamma,K}$ we only know that the partial factor g has all of its coefficients specialised, so we may use division to see if this is a factor.)

Note that in the situation in which $\mathrm{Newt}(f)$ is just a triangle with vertices $(0,n), (n,0)$ and $(0,0)$ for some n, our method reduces to the standard Hensel lifting method for bivariate polynomial factorisation. As such, our "polytope method" is a natural generalisation of Hensel lifting from the case of "generic" dense polynomials to arbitrary, possibly sparse, polynomials.

6.4.2 Hensel lifting equations

In this section we derive the basic equations which are used in our algorithm.

For any $\delta \in \text{Edge}(f)$ recall that l_δ is the associated normalised affine functional. For $i \geq 0$ we define

$$f_i^\delta := \sum_{l_\delta(e)=i} a_e X^e.$$

Thus f_i^δ is just the polynomial obtained from f by removing all terms whose exponents do not lie on the "ith translate of the supporting line of δ into the polytope Newt(f)". We call the polynomials f_0^δ *edge polynomials.*

Given the decomposition $\text{Newt}(f) = Q + R$ let δ' and δ'' denote the unique faces of Q and R which sum to give δ. As before assume $l_\delta(\delta') = c_\delta$ and $l_\delta(\delta'') = -c_\delta + \eta$. Let g and h denote generic polynomials with respect to Q and R. For $i \geq 0$ define

$$\begin{aligned} g_i^\delta &:= \sum_{q \in Q,\, l_\delta(q) = c_\delta + i} g_q X^q \\ h_i^\delta &:= \sum_{r \in R,\, l_\delta(r) = -c_\delta + \eta + i} h_r X^r. \end{aligned}$$

Once again g_i^δ and h_i^δ are obtained from g and h by considering only those terms which lie on particular lines. The next result is elementary but fundamental.

Lemma 6.4.1 *Let $f \in \mathbb{F}[x,y]$ and $\text{Newt}(f) = Q + R$ be an integral decomposition with corresponding generic polynomials g and h. Let $\text{Edge}(f)$ denote the set of edges of $\text{Newt}(f)$ and $\delta \in \text{Edge}(f)$. 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:*

$$f_0^\delta = g_0^\delta h_0^\delta, \text{ and } g_0^\delta h_k^\delta + h_0^\delta g_k^\delta = f_k^\delta - \sum_{j=1}^{k-1} g_j^\delta h_{k-j}^\delta \text{ for } k \geq 1. \tag{6.2}$$

Thus any specialisation of coefficient indeterminates which is a solution of equations (6.2) will give a full factorisation of f.

Proof: In the equation $f = gh$ gather together on each side all monomials whose exponent vectors lie on the same translate of the line supporting δ. We then have $f_0^\delta = g_0^\delta h_0^\delta$ and

$$\begin{aligned} f_k^\delta &= \textstyle\sum_{j=0}^{k} g_j^\delta h_{k-j}^\delta \quad \text{for } k \geq 1 \\ &= g_0^\delta h_k^\delta + \textstyle\sum_{j=1}^{k-1} g_j^\delta h_{k-j}^\delta + h_0^\delta g_k^\delta \end{aligned}$$

or that

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

where a sum over the empty set is understood to be zero.

These are precisely the equations which are used in Hensel lifting to try and reduce the non-linear problem of selecting a specialisation of the coefficients of g and h to give a factorisation

of f, to a sequence of linear systems which may be recursively solved. We recall precisely how this is done, as our method is a generalisation.

We begin with a specialisation of the coefficients in the polynomials g_0^δ and h_0^δ which yields a full factorisation of the polynomial f_0^δ. Equation (6.2) with $k = 1$ gives a linear system for the indeterminate coefficients of g_1^δ and h_1^δ. In the special case in which standard Hensel lifting applies this system may be solved uniquely, and thus a unique partial factorisation of f is defined which extends the original one. This process is iterated for $k > 1$ until all the indeterminate coefficients in one of the generic polynomials have been specialised, at which stage one checks whether a factor has been found by division.

The problem with this method is that in general there may *not* be a unique solution to each of the linear systems encountered. There will be a unique solution in the dense bivariate case mentioned at the end of 6.4.1, subject to a certain coprimality condition. General bivariate polynomials may be reduced to ones of this form by randomisation, but the substitutions involved destroy the sparsity of the polynomial. Our approach avoids this problem, although again is not universal in its applicability. As explained earlier, our method extends a *special* kind of partial boundary factorisation of f, rather than just the factorisation of one of its edges. In this way uniqueness in the bivariate case is restored.

6.5 A geometric lemma

This section contains a geometric lemma which ensures our method can proceed in a unique way at each step provided we start with a special type of partial boundary factorisation. We begin with a key definition.

Definition 6.5.1 *Let Λ be a set of edges of a polygon P in $\mathbb{R}^2$ and r a vector in $\mathbb{R}^2$. We say that Λ dominates P in direction r if the following two properties hold:*

- *P is contained in the Minkowski sum of the set Λ and the infinite line segment $r\mathbb{R}_{\geq 0}$ (the positive hull of r). Call this sum Mink(Λ, r).*
- *Each of the two infinite edges of Mink(Λ, r) contains exactly one point of P.*

Thus Mink(Λ, r) comprises a region bounded by the interior strip between its two infinite edges and all edges in Λ. This definition is illustrated in Figure 1 where Λ consists of all the bold edges on the boundary indicated by T.

We will call Λ an *irredundant* dominating set if there exists a vector $r \in \mathbb{R}^2$ such that Λ dominates P in direction r and no two edges e_i, e_j in Λ, for $i \neq j$, are such that

$$r\mathbb{R}_{>0} + e_i \subset r\mathbb{R}_{>0} + e_j.$$

The edges in an irredundant dominating set are necessarily connected.

The next lemma is at the heart of our algorithm.

Lemma 6.5.1 *Let P be an integral polygon and Λ an irredundant dominating set of edges of P. Suppose Λ_1 is a polygonal line segment in P such that each edge of Λ_1 is parallel to some edge of Λ. If Λ_1 is different from Λ then Λ has at least one edge that has strictly more lattice points than the corresponding edge of Λ_1.*

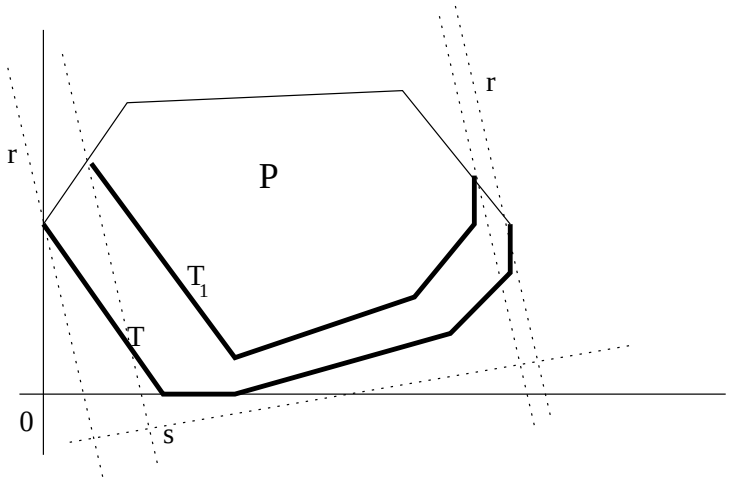


Figure 1: Dominating set of edges

The lemma is illustrated in Figure 1, where T denotes the union of the edges in Λ and T_1 the union of the line segments in Λ_1.

Before proving this lemma we make one more definition. We define a map π_r onto the orthogonal complement $\langle r\rangle^{\perp} := \{s \in \mathbb{R}^2 \mid (s \cdot r) = 0\}$ of the vector r as follows:

$$\pi_r(v) = v - \left(\frac{v \cdot r}{r \cdot r}\right) r.$$

We call this *projection by* r, and we have that $\pi_r(P) = \pi_r(\Lambda)$. To see this, it suffices to show that $\pi_r(v) \in \pi_r(\Lambda)$ for any $v \in P$. Since P is contained in $Mink(\Lambda, r)$, we can write $v = r + \lambda$ for some point $\lambda \in \Lambda$. Then

$$\begin{aligned}
\pi_r(v) &= \pi_r(r+\lambda) \\
&= (r+\lambda) - \left(\tfrac{(r+\lambda).r}{r.r}\right) r \\
&= r + \lambda - \left(\tfrac{r.r}{r.r} + \tfrac{\lambda.r}{r.r}\right) r \\
&= \lambda - \left(\tfrac{\lambda.r}{r.r}\right) r \\
&= \pi_r(\lambda)
\end{aligned}$$

from which one concludes that $\pi_r(P) \subseteq \pi_r(\Lambda)$.

Now, notice that if e_1 and e_2 are edges in an irredundant dominating set, then the length of the projection by r of the polygonal line segment e_1e_2 is just the sum of the lengths of the projections by r of the individual edges e_1 and e_2. For otherwise, we would have, say, $\pi_r(e_1) \subseteq \pi_r(e_2)$ and hence $r\mathbb{R}_{\geq 0} + e_1 \subseteq r\mathbb{R}_{\geq 0} + e_2$, a contradiction, since e_1 and e_2 belong to an irredundant dominating set. The same is true if we replace e_1 and e_2 by any line segments parallel to them — we still obtain an "additivity" in the lengths, which shall be used in the proof of the lemma.

Proof: We assume that Λ dominates P in the direction r as shown in Figure 1. Let $\delta_1, \cdots, \delta_k$ be the edges in Λ and $\delta'_1, \cdots, \delta'_k$ the corresponding edges of Λ_1. Let n_i be the number of lattice points on δ_i, and m_i that on δ'_i, $1 \leq i \leq k$. We want to show that $n_i > m_i$ for at least one i, $1 \leq i \leq k$. Suppose otherwise, namely

$$n_i \leq m_i, \quad 1 \leq i \leq k. \tag{6.3}$$

We derive a contradiction by considering the lengths of Λ and Λ_1 on the projection by π_r. Note that if $m_i = 0$ for some i then certainly $n_i > m_i$ and we are done; thus we may assume that $m_i \geq 1$ for all i.

First, certainly $\pi(\Lambda_1) \subseteq \pi(\Lambda)$ as Λ is a dominating set. Since Λ_1 is different from Λ, their corresponding end points must not coincide. Hence at least one end point of Λ_1 will not be on the infinite edges in the direction r. Hence $\pi_r(\Lambda_1)$ lies completely inside $\pi_r(\Lambda)$, so has length strictly shorter than $\pi_r(\Lambda)$.

Now for $1 \leq i \leq k$ let ϵ_i be the length of the projection of a primitive line segment on δ_i (which means that the line segment has both end points on lattice points but no lattice points in between). Certainly $\epsilon_i \geq 0$. Since the end points of δ_i are lattice points, the length of $\pi_r(\delta_i)$ is exactly $(n_i - 1)\epsilon_i$ for $1 \leq i \leq k$, hence $\pi_r(\Lambda)$ has length $\sum_{i=1}^{k}(n_i - 1)\epsilon_i$. (Here we need the fact that the dominating set is irredundant, to give us the necessary "additivity" in the lengths.) For δ_i', since it is parallel to δ_i, the projected length of a primitive line segment on it is also ϵ_i. Hence the length of $\pi_r(\Lambda_1)$ is at least $\sum_{i=1}^{k}(m_i - 1)\epsilon_i$ and from (6.3) we know that

$$\sum_{i=1}^{k}(m_i - 1)\epsilon_i \geq \sum_{i=1}^{k}(n_i - 1)\epsilon_i.$$

This contradicts our previous observation that $\pi_r(\Lambda_1)$ is strictly shorter than $\pi_r(\Lambda)$. The lemma is proved.

6.5.1 On identifying irredundant dominating sets

Before concluding this section we 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 6.5.2 *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 6.5.2. Let the *range of admissible slopes* of v_i, denoted by 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 6.5.3 *Let P be a polygon with m vertices in $\mathbb{R}^2$ and let Λ denote a set of consecutive edges of P connecting (either in the clockwise or counterclockwise direction) any two vertices v_i*

and v_j, for $i, j = 0, ..., m-1$, and $i < j$. Then Λ is an 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 no two edges in Λ are parallel. 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 v_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 6.5.2 and the fact that r_i and r_j have admissible slopes.

2. Proof of the converse is immediate by noting that, since Λ is a 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 6.5.1 *Input: A polygon P in $\mathbb{R}^2$ with m vertices v_k, for $k = 0, ..., m-1$.*
Output: The collection D of all irredundant sets of dominating edges of P, in the form $\{(i, j, d)\}$, 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, and d is the direction (clockwise or counterclockwise) of the path connecting the two vertices.

Step 1: $S \leftarrow \emptyset$, $num\_sets \leftarrow 0$.

Step 2: For $i = 0, ..., m-1$ determine admiss(i) using Lemma 6.5.2 above.

Step 3: For $i = 0, ..., m-1$ do
For $j = i+1, ..., m-1$ do
If $admiss(i) \cap admiss(j) \neq \emptyset$:
Set $S \leftarrow S \cup \{(i, j)\}$, $I_{num\_sets} \leftarrow admiss(i) \cap admiss(j)$, and $num\_sets \leftarrow num\_sets + 1$.

Step 4: Repeat Steps 4.1-4.3 for $d =$ clockwise and $d =$ counterclockwise:
4.1: Consider the num_sets dominating sets found so far.
For $k = 0, ..., num\_sets - 1$ do
For $h = k+1, ..., num\_sets - 1$
If $I_k = I_h$ and either the directed dominating set of index h is a subset of the directed dominating set of index k or the converse is true:
Mark the larger set for deletion from S.

4.2: Choose only the unmarked sets in S of the form (i,j) and store (i,j,d) in D.
4.3: Unmark all sets in S.

Step 5: Return D.

Proposition 6.5.1 *Algorithm 6.5.1 works correctly as specified and requires $O(m^4)$ arithmetic operations, where m is the number of vertices of the polygon P.*

Proof: That Steps 1-3 above produce all sets of dominating edges is a direct consequence of Lemma 6.5.3. Note that for a fixed pair of vertices (i,j), both the clockwise and counterclockwise paths of edges connecting them are dominating. In Step 4, we sort all such sets in order to keep only the irredundant ones. The sorting rule is as follows. Suppose for instance that we are given two dominating sets (i_1,j_1) and (i_2,j_2) such that the clockwise path of edges connecting i_1 to j_1, say, is a subset of the clockwise path connecting i_2 to j_2. For $(i_2,j_2,clockwise)$ to be irredundant, one must be able to find at least one direction $r \in \mathbb{R}^2$ such that P can be embedded in a strip along the direction of r using (i_2,j_2,d) as a dominating set, but not (i_1,j_1,d). This can happen only when $I_1 \neq I_2$, where $I_1 = \text{admiss}(i_1) \cap \text{admiss}(j_1)$ and $I_2 = \text{admiss}(i_2) \cap \text{admiss}(j_2)$. The same argument can be repeated for the counterclockwise direction along which the paths are considered.

The cost can be easily established by noting the following. The loops in Step 3 iterate $O(m^2)$ times in total, producing $O(m^2)$ dominating sets. These are then sorted in Step 4, where the two loops iterate $O((m^2)^2)$ times in total. All operations in the above algorithm require only arithmetic operations for calculating and comparing slopes as well as intersection of rational sets. We will see in Chapter 7 how the latter intersections can be made to involve strictly integral values.

6.6 The main theorem

Let Γ be an irredundant dominating set of $\text{Newt}(f)$. 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 as Laurent polynomials (see Definition 2.1.14 of Chapter 2), up to monomial factors.

We are now ready to state our main theoretical result.

Theorem 6.6.1 *Let $f \in \mathbb{F}[x,y]$ and $\text{Newt}(f) = Q + R$ be a fixed Minkowski decomposition, where Q and R are integral polygons in the first quadrant. Let Γ be an irredundant dominating set of $\text{Newt}(f)$ 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 $\#\text{Newt}(f)$.*

We shall prove this theorem inductively through the next two lemmas.

Lemma 6.6.1 *Let f, Q, R and Γ be as in the statement of Theorem 6.6.1. Suppose we are given a K-factorisation of f, where $K = (k_\delta)_{\delta \in \Gamma}$ (more specifically, a $(\Gamma, K; Q, R)$-factorisation). For*

each $\delta \in \Gamma$, denote by δ' the face of Q supported by $l_\delta - c_\delta$. There exists $\delta \in \Gamma$ with the following properties

- *The face δ' is an edge (rather than a vertex).*
- *The number of unspecialised coefficients of $g^\delta_{k_\delta}$ is nonzero but strictly less than the number of integral points on δ'.*
- *All the unspecialised terms of $g^\delta_{k_\delta}$ have exponents being consecutive integral points on the line defined by $\ell_\delta = (c_\delta + k_\delta)$.*

Proof: Let $\bar{Q}$ be the polygon

$$\bar{Q} := \{r \in Q \,|\, \ell_\delta(r) \geq c_\delta + k_\delta \text{ for all } \delta \in \Gamma\}.$$

Note that the lattice points in $\bar{Q}$ correspond to unspecialised coefficients of g. Let Λ denote the set of edges $\delta \in \Gamma$ of Newt(f) such that the functional $\ell_\delta - c_\delta$ supports an edge of Q (rather than just a vertex). Note that $\Lambda \neq \emptyset$, for otherwise Q must be a single point or a line segment in direction r, contradicting our assumption. We denote the edge by δ', and write $\bar{\delta}$ for the face of $\bar{Q}$ supported by $\ell_\delta - (c_\delta + k_\delta)$. Note that each $\bar{\delta}$ contains at least one lattice point. (This follows from the second property in Definition 6.3.1.) Certainly, $\bar{\delta}$ is parallel to δ' for each $\delta \in \Lambda$, and the edge sequence $\{\bar{\delta}\}_{\delta\in\Lambda}$, forms a polygonal line segment in Q. Since Γ is an irredundant dominating set for Newt(f), the set of edges $\{\delta'\}_{\delta\in\Lambda}$ is an irredundant dominating set for Q. By Lemma 6.5.1, there is at least one edge $\delta \in \Lambda$, such that δ' has strictly more lattice points than $\bar{\delta}$. This edge δ has the required properties. This completes the proof.

Lemma 6.6.2 *Let f, Q, R and Γ be as in the statement of Theorem 6.6.1. Suppose we are given a K-factorisation of f, where $K = (k_\delta)_{\delta\in\Gamma}$. Moreover, assume this factorisation extends a coprime dominating edges factorisation, i.e., the polynomials g^δ_0 and h^δ_0 are coprime up to monomial factors for all $\delta \in \Gamma$. Then there exists $\delta \in \Gamma$ such that the coefficients of $g^\delta_{k_\delta}$ are not all specialised, but they may be specialised in at most one way consistent with equations (6.2). This specialisation may be computed in time polynomial in* #Newt(f).

Proof: The basic idea of the proof is to first transform the bivariate equation (6.2) into equations of univariate polynomials determined by the individual edges, then to determine the existence or uniqueness of solutions.

Select $\delta \in \Gamma$ such that the properties in Lemma 6.6.1 hold. Let n_δ and m_δ be the number of integral points on the edges δ' and $\bar{\delta}$ respectively, where δ' and $\bar{\delta}$ are defined as in the proof of Lemma 6.6.1. Thus we have $m_\delta < n_\delta$ and $m_\delta \geq 1$. With the notation from Section 6.3, write $\ell_\delta(e_1, e_2) = \nu_1 e_1 + \nu_2 e_2 + \eta$, where ν_1 and ν_2 are coprime.

Let z and w be new variables. Using the transform (6.1), any monomial of the form $x^{e_1}y^{e_2}$ can be written as

$$x^{e_1}y^{e_2} = z^{i_1}w^{i_2} \tag{6.4}$$

where

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

Every monomial in g_i^δ is of the form $x^{e_1}y^{e_2}$ where $\ell_\delta(e_1,e_2) = c_\delta + i$. 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 $\ell_\delta - c_\delta$ and $\ell_\delta + c_\delta - \eta$, respectively. Thus 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. With this construction, $G_0(z), H_0(z)$ and $F_0(z)$ have nonzero constant term and are "ordinary polynomials", i.e., contain no negative powers of z. For $i < k_\delta$ all of the coefficients in the polynomials $G_i(z)$ and $H_i(z)$ have been specialised. Moreover $G_0(z)$ is of degree n_δ, and all but m_δ of the coefficients of $G_{k_\delta}(z)$ have been specialised. Equations (6.2) with this change of variables may be written as $F_0(z) = G_0(z)H_0(z)$, and for $k \geq 1$

$$G_k(z)H_0(z) + G_0(z)H_k(z) = F_k(z) - \sum_{j=1}^{k-1} G_j(z)H_{k-j}(z).$$

We know that all of the coefficients of $G_i(z)$ and $H_i(z)$ have been specialised for $0 \leq i < k_\delta$ in such a way as to give a solution to $F_0 = G_0H_0$ and the first $k_\delta - 1$ equations above. Thus we need to try and solve

$$G_{k_\delta}H_0 + G_0H_{k_\delta} = F_{k_\delta} - \sum_{j=1}^{k_\delta-1} G_jH_{k_\delta-j}. \tag{6.5}$$

for the unspecialised indeterminate coefficients of G_{k_δ} and H_{k_δ}.

We first compute using Euclid's algorithm ordinary polynomials $U(z)$ and $V(z)$ such that

$$V(z)H_0(z) + U(z)G_0(z) = 1$$

where $\deg_z(U(z)) < \deg_z(H_0(z))$ and $\deg_z(V(z)) < \deg_z(G_0(z))$. (Note that $G_0(z)$ and $H_0(z)$ are coprime since we have a coprime partial boundary factorisation.) Any solution G_{k_δ} of Equation (6.5) must be of the form

$$G_{k_\delta} = \{V(F_{k_\delta} - \sum_{j=1}^{k_\delta-1} G_jH_{k_\delta-j}) \bmod G_0\} + \varepsilon G_0 \tag{6.6}$$

for some Laurent polynomial $\varepsilon(z)$ with undetermined coefficients.

We rearrange (6.6) as

$$G_{k_\delta} - \{V(F_{k_\delta} - \sum_{j=1}^{k_\delta-1} G_jH_{k_\delta-j}) \bmod G_0\} = \varepsilon G_0 \tag{6.7}$$

Let the degree in z of the Laurent polynomial on the left hand side of this equation be d. Now the degree of the polynomial $G_0(z)$ as a Laurent polynomial (and an ordinary polynomial) is $n_\delta - 1$. If $d < n_\delta - 1$ then we must have $d = 0$. In other words, (6.6) has a unique solution, namely that with $\varepsilon = 0$. Otherwise $d \geq n_\delta - 1$ and the degree in z of $\varepsilon(z)$ as a Laurent polynomial is $d - (n_\delta - 1)$. Hence in this case we need to also solve for the $d - n_\delta + 2$ unknown coefficients of $\varepsilon(z)$. We know that all but m_δ coefficients of G_{k_δ} have already been specialised, and these unspecialised ones are adjacent terms. Hence exactly $(d+1) - m_\delta$ coefficients on the left hand

side of (6.7) have been specialised, which are adjacent lowest and highest terms. By assumption we have that $m_\delta < n_\delta$, and hence $(d+1) - m_\delta \geq d - n_\delta + 2$.

All of the coefficients of the right hand side of Equation (6.7) have been specialised, except those of the unknown polynomial $\varepsilon(z)$. On the left hand side all but the middle m_δ coefficients have been specialised. This defines a pair of triangular systems from which one can either solve for the coefficients of ε uniquely, or show that no solution exists (this may happen when $n_\delta > m_\delta + 1$). We describe precisely how this is done: Suppose that exactly r of the lowest terms on the left hand side have been specialised, and hence also $(d+1) - (m_\delta + r)$ of the highest terms. We can solve uniquely for the r lowest terms of $\varepsilon(z)$ using the triangular system defined by considering coefficients of the powers $z^a, z^{a+1}, \ldots, z^{a+r-1}$ on both sides of Equation (6.6), where z^a is the lowest monomial occurring on the left hand side. One may also solve for the coefficients of the $(d+1) - (m_\delta + r)$ highest powers uniquely using a similar triangular system. (Note that to ensure the triangular systems each have unique solutions we use here the fact that the constant term of G_0 is nonzero, and the polynomial is of degree exactly $n_\delta - 1$.) Noticing that $(d+1) - (m_\delta + r) + r = (d + 1 - m_\delta) \geq d - n_\delta + 2$, we see that all the coefficients of ε have been accounted for. However, if $d + 1 - m_\delta > d - n_\delta + 2$ (i.e. $n_\delta > m_\delta + 1$) there will be some "overlap", and the two triangular systems might not have a common solution. In this case there can be no solution to the Equation (6.6). If an $\varepsilon(z)$ does exist which satisfies Equation (6.7) then the remaining coefficients of G_{k_δ} can now be computed uniquely. Having computed the only possible solution of (6.6) for G_{k_δ} we can substitute this into Equation (6.5) and recover H_{k_δ} directly. More precisely compute

$$\frac{(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) - G_{k_\delta} H_0}{G_0}. \tag{6.8}$$

If its coefficients match with the known coefficients of H_{k_δ} then we have successfully extended the partial factorisation; otherwise we know no extension exists.

These computations can be done in time quadratic in the degree of the largest polynomial occurring in the above equations. Since all polynomials are Newton polytopes which are line segments lying within $\mathrm{Newt}(f)$ this is certainly quadratic in $\#\mathrm{Newt}(f)$. (In fact, the running time is most closely related to the length of the side n_δ from which we are performing the lifting step. We shall show in Chapter 8 that this number is of the order $O(n)$, where $n = \deg(f)$.) This completes the proof.

Theorem 6.6.1 may now be proved in a straightforward manner: Specifically, one first shows that for any partial factorisation extending a coprime dominating edges factorisation, there exists at most one full factorisation extending it, and this may be efficiently found. This is proved by induction on the number of unspecialised coefficients in the partial factorisation using Lemma 6.6.2. Theorem 6.6.1 then follows easily as a special case.

6.7 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 [30]. 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 (i, j) 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 (6.7) to that between two regular polynomials (see definition 2.1.16 in Chapter 2). 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

$$V(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0, \tag{6.9}$$

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 (6.8) 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

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

6.8 The algorithm

We now gather everything together and state our algorithm:

Algorithm 6.8.1 *Input: A polynomial $f \in \mathbb{F}[x,y]$ of total degree n, and a positive integer M. Output: A factorisation of f or "failure" or "f is irreducible".*

Step 1: [Convex hull]
Compute a vertex-edge description of $\mathrm{Newt}(f)$ *using a polygon convex hull algorithm. Let the edges be $\delta_0, \ldots, \delta_{m-1}$, cyclically joining vertices $v_0, \ldots, v_{m-1}$. So* $\mathrm{Edge}(f) = \{\delta_i\}$.

Step 2: [Factor edge polynomials]
Compute a complete factorisation of all edge polynomials f_0^{δ}, $\delta \in \mathrm{Edge}(f)$.

Step 3: [Admissible edge decompositions]
For each edge $\delta_i \in \mathrm{Edge}(f)$ compute the set $\{m_j^{(i)} \mid 0 \leq j \leq \deg(f_0^{\delta_i})\}$, where $m_j^{(i)}$ is the number of monic factors of the edge polynomial $f_0^{\delta_i}$ of degree j.

Step 4: [Dominating sets]
List all sets $\{\Gamma_i\}$ of square-free dominating facets of $\mathrm{Newt}(f)$ *using algorithm 6.5.1. If there are no such sets then fail.*

Step 5: [Count coprime dominating facets factorisations]
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 $\mathrm{Newt}(f)$.

Step 6: [Select a dominating set]
Select the dominating set Γ for which the number computed in Step 5 is minimal. If this number is greater than M then fail.

Step 7: By repeatedly applying the method in the proof of Lemma 6.6.2, lift each coprime dominating edges factorisation of f as far as possible. If any of these lift to a full factorisation output this factorisation and halt. If none of them lifts to a full factorisation then output "irreducible".

Proposition 6.8.1 *Algorithm 6.8.1 outputs correctly.*

Proof: The algorithm will always succeed when one finds a dominating set Γ of $\mathrm{Newt}(f)$ such that the polynomials f_0^{δ}, $\delta \in \Gamma$, are all square-free (up to a monomial factor), provided we take M "sufficiently large" (an upper bound on M is d^m, where d is the maximum number of integral points falling along any edge and m is the number of edges of $\mathrm{Newt}(f)$). One might call polynomials for which such sets exist *nice*. Suppose that the polynomial is reducible, and we have a proper factorisation $f = gh$ with corresponding non-trivial decomposition $\mathrm{Newt}(f) = Q + R$. This is a full factorisation extending a Γ-boundary factorisation, which is necessarily coprime by the assumption on Γ. It will therefore be found during one of the liftings, by Theorem 6.6.1.

Prior to discussing the time complexity we shall discuss in details some of the steps above. In the forthcoming discussion we shall treat $\mathbb{F}$ as a finite field whose characteristic fits in a machine word.

Steps 1,2

Steps 1 and 2 can be classified as a pre-computation since they are performed only once for each input polynomial. If s denotes the number of nonzero terms in f, the convex hull may be computed in time $\mathcal{O}(s \log(s))$ (see [60]). Note that $s \leq \#(\text{Newt}(f))$. Ignoring logarithmic factors, Step 2 may be performed using a univariate factorisation algorithm over finite fields in $O(dM(d))$ field operations, where d is the maximum degree of any of the edge polynomials. Certainly $d \leq \#(\text{Newt}(f))$.

Step 3

This is also another pre-computation step which is performed only once during the entire algorithm. We refer the reader to the recursive counting Algorithm 7.3.4 of Chapter 7. Given a univariate polynomial $P(z)$ of degree d over $\mathbb{F}$ and its canonical factorisation (not necessarily square-free) into irreducibles, the algorithm returns the number of factors of $P(z)$ of degree $k = 1, ..., d$ using $O(d^{1+h}h)$ bit operations, where h denotes the number of irreducible factors of $P(z)$. Obviously, $h = O(d)$, although on average it is approximately $\log d$ [83, 84, 100].

Step 4

The maximum number of edges is certainly s, and thus one may easily find all suitable Γ using algorithm 6.5.1, which in this case requires $O(s^2)$ arithmetic operations.

Steps 5

For Step 5, one may use a modified version of the polygon summand counting algorithm in [45]. The modification needed is that one only considers summands of the polygon whose edges have lengths matching the degrees of the known univariate factors of the edge polynomials. Also, one counts two different factorisations of the edge polynomials on the dominating set separately even if the decomposition of the polytope is the same in each case. It is easily seen that the algorithm has running time polynomial in $\#(\text{Newt}(f))$. More precisely the subroutine is as follows:

Algorithm 6.8.2 (Step 5) *Input: The edge sequence* $\{n_i e_i\}_{0 \leq i \leq m-1}$ *of the Newton polytope of a bivariate polynomial, starting at vertex* v_0 *where* $e_i \in \mathbb{Z}^2$ *are primitive vectors (i.e. have coprime integer coordinates) and* n_i *are positive integers, a set* Γ *of dominating facets of* $\text{Newt}(f)$*, and a set* $\{m_j^{(i)} \mid 1 \leq j \leq n_i\}$ *of admissible edge decompositions lengths for each edge* $n_i e_i$*.*
Output: The number of coprime Γ*-boundary factorisations of* $\text{Newt}(f)$ *and an array A. Each cell in A contains a pair* (u, S) *where u is a non-negative integer and S is a subset of* $\{(k, i) : 1 \leq k \leq n_i, 0 \leq i \leq m-1\}$*.*

Step 5.1: Compute the set IP of all the integral points in $\text{Newt}(f)$ *(so* $v_0 \in IP$*); say IP has* $t(= \#(Newt(f)))$ *points. Initialize a t-array A indexed by the points in IP. Set* $A_{-1}[v] := (0, \emptyset)$ *for all* $v \in IP$ *except the cell* $A_{-1}[v_0]$ *which is set to* $(1, \emptyset)$*.*

Step 5.2: For i from 0 *up to* $m-1$*, compute the t-array* A_i *from* A_{i-1}*:*

5.2.1 First copy the contents of all the cells of A_{i-1} *into* A_i *(this step is for* $k = 0$*).*

5.2.2 For each $v \in IP$ with the first number of the cell $A_{i-1}[v]$ nonzero, and for each $0 < k \leq n_i$ for which $m_k^{(i)} > 0$, if $v' = v + ke_i \in IP$ then update the cell $A_i[v']$ as follows: if (u_1, S_1) is the value of $A_{i-1}[v]$ and (u_2, S_2) the current value of $A_i[v']$ then the new value of $A_i[v']$ is $(u, S_2 \cup \{(k,i)\})$. Here we take $u = u_2 + u_1$ in the case $n_i e_i \notin \Gamma$ and $u = u_2 + m_k^{(i)} u_1$ in the case $n_i e_i \in \Gamma$.

Step 5.3: Return the number u and the array $A = A_m$, where (u, S) is the content of cell $A_{m-1}[v_0]$.

Proposition 6.8.2 *The above algorithm works correctly, producing the total number of integral summands in time polynomial in $\#(\mathrm{Newt}(f))$.*

Proof: Correctness of the algorithm follows by a suitable modification of Theorem 18 in [45]. By Lemma 13 of [45], the number of integral summands of $\mathrm{Newt}(f)$ corresponds to the total number of closed paths $\sum_{0 \leq i \leq m-1} k_i e_i$, such that $k_i \neq 0$ for all i and $k_{m-1} \neq n_{m-1}$. We shall show that this number is the integer stored in $A_{m-1}[v_0]$. Since the length of an edge is defined to be the number of integral points lying on it, which in turn corresponds to one plus the degree of the edge polynomial associated with it, the condition $m_k^{(i)} > 0$ guarantees that we count only those summands whose edges have lengths corresponding to degrees of "known" univariate factors of the original edge polynomial in $\mathrm{Newt}(f)$. Now, as seen in the original proof, we suppose that $v = v_0 + k_0 e_0 + ... + k_i e_i$, for any $v \in IP$. We can then view the vector sum as a path from v_0 to v, so that the number of such paths is equal to the sum of the number of paths from v_0 to $v - ke_i$, for $0 \leq k \leq n_i$, using $e_0, ..., e_{i-1}$. However, if we further know that $n_i e_i$ belongs to Γ, then for each $k = 1, ..., n_i$, we should count all possible factorisations of the edge polynomial corresponding to the same edge, as indicated by the number of factors of degree k of the edge $n_i e_i$ polynomial. As a result, and for each admissible $k = 1, ..., n_i$, the value of u in $A_i[v]$ is incremented as follows: by the number of paths from v_0 to $v - ke_i$, using $e_0, ..., e_{i-1}$, if $n_i e_i$ is not in Γ, or by $m_k^{(i)}$ times this number, otherwise. The loops in the above process can be easily seen to be of the order $O(\#(\mathrm{Newt}(f)).md)$, where d is the maximum number of integral points on any edge. The innermost loop computations involve updating the integer u through integer addition and updating the set S through the set union operation. Since u requires an upper bound of $M < d^m$, its update has an upper bound of $O(m \log d)$ bit operations. If we further consider set union to require a single bit operation, the complexity of the above algorithm becomes of the order $O(\#(\mathrm{Newt}(f)).m^2 d)$, ignoring logarithmic factors.

Step 6

Having selected a dominating set, one can recover all coprime dominating facets factorisation in the array output by Algorithm 6.8.2. We describe how one such factorisation can be found. Suppose the cell $A[v_0]$ contains the pair (u, S). Choose any $(k, i) \in S$. The line segment ke_i will be the "final edge" in our summand of $\mathrm{Newt}(f)$. Since by assumption $m_k^{(i)} > 0$, we can also choose a factor $g_0^{\delta_i}$ of the edge polynomial of δ_i which has degree k. This is the "final edge polynomial" in our dominating facets factorisation. Let (u', S') be the contents of cell $A[v_0 - ke_i]$. Pick any $(k', i') \in S'$ with $i' < i$. The line segment $k'e_{i'}$ will be the "penultimate edge" in our summand of $\mathrm{Newt}(f)$, and we can further choose a "penultimate edge polynomial" if $n_i e_i \in \Gamma$.

As our sequence of i's is decreasing we shall eventually return to the cell $A[v_0]$. At that point we will have recovered one summand in a decomposition of Newt(f).

The complexity of the above process is M times the time required to find one dominating facets factorisation, which is linear in the number of edges m. Certainly $m = \mathcal{O}(\text{Newt}(f))$.

Step 7

Now one lifts each coprime dominating facets factorisation using the method described in Lemma 6.6.2. Using Theorem 6.6.1, lifting from each coprime dominating edges factorisation can be done in time polynomial (in fact cubic) in d, which itself is bounded by Newt(f). However, although one can find such a dominating edges factorisation efficiently, the number of them may be exponential in the degree. In practice we recommend that a relative small number of dominating edges factorisations are tried before the polynomial is randomised and one resorts to other "dense polynomial" techniques.

Time complexity and comments

Proposition 6.8.3 *Assuming Step 3 above is performed as a precomputation, Algorithm 6.8.1 halts in time polynomial in M and $\#(Newt(f))$.*

Proof: Steps 1, 2, 4, 5, and 6 are performed in time polynomial in $\#(Newt(f))$. In Step 7 one performs at most M liftings. The result now follows using the estimates discussed above.

This algorithm should be compared with the standard method of factoring "nice" polynomials using Hensel lifting [46]. Precisely, in the literature a bivariate polynomial of total degree n which is square-free upon reduction modulo y is often called "nice". The standard Hensel lifting algorithm will factor "nice" bivariate polynomials, on average very quickly [46], although in exponential time in the worst case. Notice that a "nice" polynomial would be one whose Newton polytope has "lower boundary" a single edge of length n which is square-free. The above algorithm factors not just these polynomials, but also any polynomials which have a "square-free dominating set". In the case of a generic dense "nice" polynomial, it reduces to a modified form of standard Hensel lifting. (The algorithm also includes as a special case that given in Wan [128], where one "lifts downward" from the edge joining $(n, 0)$ and $(0, n)$)

6.9 Examples and implementation

6.9.1 Example

Suppose we want to factor the following polynomial over $\mathbb{F}_2$

$$f = x^{12} + x^{19} + (x^{10} + x^{11} + x^{13})y + (x^8 + x^9 + x^{12} + x^{17})y^2 + x^7y^3 + (x^4 + x^{11})y^4$$

$$+(x^2 + x^5 + x^{10})y^5 + y^6 + x^{10}y^8 + (x^8 + x^{11})y^9 + x^6y^{10} + x^9y^{12} + x^{15}y^{16}$$

with Newton polytope pictured in Figure 2 where a star indicates a nonzero term of f.

Newt(f) is found to have three non-trivial decompositions, and eight irredundant dominating sets. None of these sets have edge polynomials which are all square-free; however, fortunately

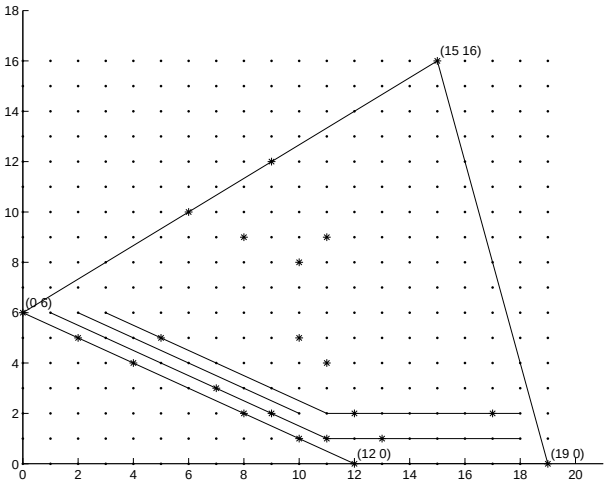


Figure 2: Newton polytope of f

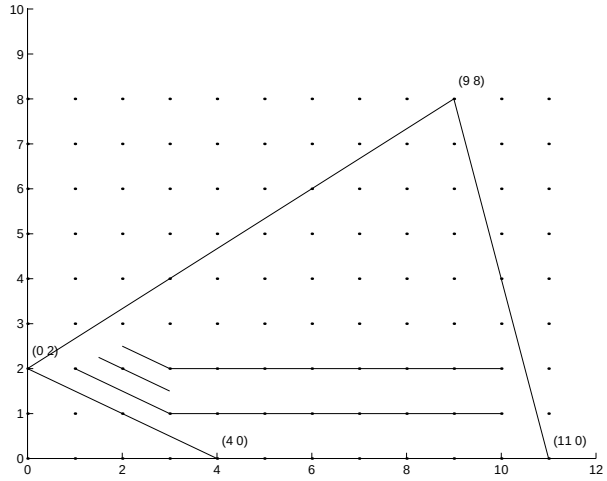


Figure 3: Newton polytope Q of the generic polynomial g

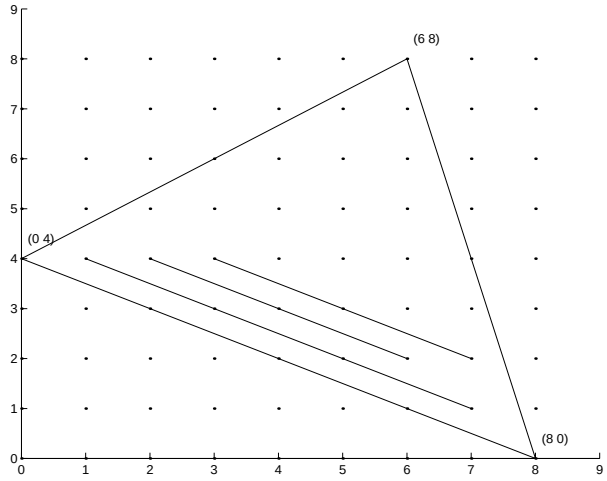


Figure 4: Newton polytope R of the generic polynomial h

we are still able to lift successfully from one of the coprime partial boundary factorisations. Specifically, consider the decomposition $\mathrm{Newt}(f) = Q + R$, where Q and R are the convex hulls of the sets $\{(0,2),(4,0),(11,0),(9,8)\}$ and $\{(0,4),(8,0),(6,8)\}$ respectively (see Figures 3 and 4). The generic polynomials for this decomposition are as usual denoted g and h. The dominating edges of $\mathrm{Newt}(f)$ which allow a coprime edge factorisation are given by

$$\delta_1 = \mathrm{conv}\{(0,6),(12,0)\}, \quad \delta_2 = \mathrm{conv}\{(12,0),(19,0)\}$$

and the corresponding edge polynomials are

$$\begin{aligned} f_0^{\delta_1} &= y^6 + x^2y^5 + x^4y^4 + x^8y^2 + x^{10}y^1 + x^{12} \\ f_0^{\delta_2} &= x^{12} + x^{19}. \end{aligned}$$

The coprime factors from which the lift begins are

$$\begin{aligned} g_0^{\delta_1} = y^2 + x^2y + x^4, \quad h_0^{\delta_1} = y^4 + x^8 \\ g_0^{\delta_2} = x^4 + x^{11}, \quad h_0^{\delta_2} = 1. \end{aligned}$$

The lifting process is then initiated; Figures 3 and 4 help illustrate the process in that the lines drawn in the interior of the polygons indicate the first few layers of coefficients which are revealed during the lifting, and the lines in the interior of $\mathrm{Newt}(f)$ the known coefficients of f which are used to do this. We recall some notation defined earlier in the chapter. For $i = 1$ and 2, the normalised affine functional of δ_i is denoted l_{δ_i}. In this case $l_{\delta_1}(r_1, r_2) = r_1 + 2r_2 - 12$ and $l_{\delta_2}(r_1, r_2) = r_2$. The constants c_{δ_i} such that $l_{\delta_i} - c_{\delta_i}$ defines a face of Q are $c_{\delta_1} = -8$ and $c_{\delta_2} = 0$ respectively.

We start then with a K-factorisation where $K = (k_{\delta_1}, k_{\delta_2})$ and $k_{\delta_1} = k_{\delta_2} = 1$. At this stage, we would like to extend this partial factorisation to either a $(2,1)$-factorisation, or a $(1,2)$-factorisation. By Lemma 6.6.2 we are guaranteed that it will be possible in at least one of these two cases. (We shall borrow notation from the proof of Lemma 6.6.1 for the next few paragraphs.) The polygon $\bar{Q}$ is that obtained from Q by moving the two lower facets "one step in" as indicated in the diagram. Examining Q we see that the edge supported by $l_{\delta_2} - c_{\delta_2}$,

namely the line joining $(4,0)$ and $(11,0)$, has 8 integral points. The edge of $\bar{Q}$ supported by $l_{\delta_2} - (c_{\delta_2} + 1)$ (this is the line joining $(3,1)$ and $(10,1)$) also has eight integral points. Thus we cannot lift from δ_2. So Lemma 6.6.2 assures us we must be able to lift from δ_1. Indeed, the edge of Q supported by $l_{\delta_1} - c_{\delta_1}$ has three points, and the corresponding edge of $\bar{Q}$ only two points.

We now describe explicitly the first lifting step from δ_1. To simplify notation let δ_1 be replaced simply by δ. We shall now also use notation from the proof of Lemma 6.6.2. We have $l_\delta(r_1, r_2) = \nu_1 r_1 + \nu_2 r_2 + \eta$ where $\nu_1 = 1, \nu_2 = 2, \eta = -12$. Also $\zeta_1\nu_1 + \zeta_2\nu_2 = 1$ where $\zeta_1 = 1$ and $\zeta_2 = 0$. Thus the change of variables is $z := x^2y^{-1}$ and $w := x^1y^0$. The monomials s and t are y^2 and y^4 respectively. Hence we have that $g_0^\delta = y^2 + x^2y + x^4 = y^2(1 + +(x^2y^{-1}) + (x^2y^{-1})^2) = sw^0G_0(z)$ where $G_0(z) = 1 + z + z^2$. Also, $g_1^\delta = g_{(1,2)}xy^2 + g_{(3,1)}x^3y + g_{(5,0)}x^5$ where the $g_{(i,j)}$ are indeterminates. The indeterminate $g_{(5,0)}$ has already been specialised to the value 0 in our partial factorisation. We thus have $g_1^\delta = sw^1(g_{(1,2)} + g_{(3,1)}z)$, and so $G_1(z) = g_{(1,2)} + g_{(3,1)}z$. Similarly, $H_{(0)}(z) = 1 + z^4$ and $H_{(1)}(z) = h_{(1,4)} + h_{(3,3)}z + h_{(5,2)}z^2 + h_{(7,1)}z^3$; and $F_0(z) = 1 + z + z^2 + z^4 + z^5 + z^6$, $F_1(z) = z^3 + z^4 + z^5$.

The equation we shall use in the lifting step is

$$G_0(z)H_{(1)}(z) + G_1(z)H_{(0)}(z) = F_1(z).$$

The polynomials $U(z)$ and $V(z)$ such that

$$U(z)G_0(z) + V(z)H_{(0)}(z) = 1$$

are $U(z) = 1 + z^2 + z^3$ and $V(z) = z$. Thus

$$G_1(z) + (V(z)F_1(z) \bmod G_0(z)) \quad = \quad \varepsilon(z)G_0(z)$$

for some polynomial $\varepsilon(z) = \varepsilon_0$ with undetermined coefficients. Now the second term on the left hand side is just 0 and hence the degree of the left hand side as a Laurent polynomial is 1. This is less than the degree of G_0, and so the only solution is that with $\varepsilon = 0$. Thus $G_1(z) = 0$.

$$\frac{F_1 - H_{(0)}G_1}{G_0} = z^3,$$

and we deduce that $H_{(1)}(z) = z^3$. Thus $h_{(7,1)} = 1$ and $h_{(1,4)} = h_{(3,3)} = h_{(5,2)} = 0$. This completes the first lifting step.

At this stage one may continue to lift from δ_1, or alternatively start to lifting from δ_2. The latter has the advantage that more coefficients will be revealed at each step; however, the computations required involve higher degree polynomials and as such it may be preferable to keep lifting from the shorter edge. We do this and next obtain a $(3,1)$-factorisation of K, with $g_{(2,2)} = g_{(4,1)} = h_{(2,4)} = h_{(4,3)} = h_{(6,2)} = 0$. Notice that this lifting step is somewhat easier since $F_2 - G_1H_{(1)} = 0$ which again results in $G_2 = H_{(2)} = 0$. One may continue lifting from δ_1 to obtain a $(4,1)$-factorisation. In this we find $g_{(5,1)} = 1$ and $g_{(3,2)} = h_{(3,4)} = h_{(5,3)} = h_{(7,2)} = 0$. At this stage lifting further from δ_1 becomes impossible. Thus one must now lift from δ_2 to get a $(3,2)$-factorisation. We explain briefly how this step is performed as it illustrates somewhat the role of the triangular systems.

So let $\delta := \delta_2$. The change of variable is now $z := x$ and $w := y$ and we have the equation

$$G_1(z) - (V(z)F_1(z) \bmod G_0) = \varepsilon(z)G_0.$$

Here

$$G_1(z) = 1z^{-2} + 0z^{-1} + 0 + 1z + g_{(6,1)}z^2 + g_{(7,1)}z^3 + g_{(8,1)}z^4 + g_{(9,1)}z^5 + g_{(10,1)}z^6$$

and $G_0(z) = 1 + z^7$. Also V is the inverse of $H_{(0)}(z) = 1$ modulo $G_0(z)$, which is just 1. The polynomial $F_1(z) = z^{-2} + z^{-1} + z$. We first compute $(VF_1 \bmod G_0)$ as

$$z^{-2}(1 + z + z^3) \bmod (1 + z^7) = z + z^5 + z^6.$$

Hence the left hand side is

$$z^{-2} + g_{(6,1)}z^2 + g_{(7,1)}z^3 + g_{(8,1)}z^4 + (1 + g_{(9,1)})z^5 + (1 + g_{(10,1)})z^6.$$

This has degree 8 as a Laurent polynomial, and hence the degree of our unknown polynomial $\varepsilon(z)$ is $8 - 7 = 1$. Let $\varepsilon(z) = (\varepsilon_{-1}z^{-1} + \varepsilon_{-2}z^{-2})$. Then equating the powers of z^{-2} and z^{-1} we get the triangular system

$$\begin{aligned} 1\varepsilon_{-2} + 0\varepsilon_{-1} &= 1 \\ 0\varepsilon_{-2} + 1\varepsilon_{-1} &= 0 \end{aligned}$$

which has solution $\varepsilon_{-2} = 1$ and $\varepsilon_1 = 0$. Hence we get that

$$\begin{aligned} G_1(z) &= z + z^5 + z^6 + z^{-2}(1 + z^7) \\ &= z^{-2} + z + z^6. \end{aligned}$$

This completes the lifting step.

Now one may once again choose to lift from δ_1 another few steps to get a $(7, 2)$-factorisation. Then one may lift for two steps from δ_2 to obtain a $(7, 4)$-factorisation. One continues in this manner until all the indeterminate coefficients in one of the two generic factors g and h have been specialised. (Of course, if we are not lifting an actual full factorisation, we may have to abandon the lifting at some stage because either our triangular systems have no common solution, or the computed coefficients in H do not match with the known coefficients.)

It is perhaps appropriate at this stage to make a few observations on how sparse polynomials may be factored more quickly using Algorithm 6.8.1. Using standard Hensel lifting the polynomial f above would first be randomised to obtain a dense polynomial of total degree 31. It could have as many as $(32 \times 33)/2 = 528$ nonzero terms, and heuristically around half this many since f is over the binary field. The factor g we found above would then correspond to a "dense" factor of our original polynomial of total degree 17. It would be found by Hensel lifting a degree 17 factor of the reduction modulo y of our randomised version of f, and $(17 \times 18)/2 = 153$ terms (heuristically half of them nonzero) need to be determined. In our algorithm, one restricts attention to unknown terms in possible factors whose exponents lie within certain polygons. Thus for the factor g we found we only need to determine 57 coefficients. Moreover, if the polynomial f is sparse, there is good chance that most of these terms, and those in h, will be zero and so one can exploit sparse data structures (see Chapter 7). The main benefit, though, of our approach appears to be for very sparse but composite polynomials of very high degree. In this case, one expects few coprime partial boundary decompositions, and as one can try and lift each one to a full factorisation, the algorithm will succeed (or fail) relatively quickly. If one randomises the polynomial by substitution of linear forms, the special sparse structure is completely lost. To factor the randomised polynomial using Hensel lifting, for example, one expects to have to try a large number of lifts. Thus, as demonstrated in the next chapter, our algorithm can be used to factor very sparse polynomials of degree beyond the reach of classical Hensel lifting.

6.9.2 Implementation

We have developed a preliminary implementation of the algorithm with the aim of demonstrating how it would work for bivariate polynomials over $\mathbb{F}_2$. The work was carried out at the Oxford University Supercomputing Centre (OSC) on the Oswell machine. The implementation was written using a combination of C and Magma programs, and was divided into three phases. In the first phase, the input polynomial is read and its Newton polytope computed using the asymptotically fast Graham's algorithm for computing convex hulls [60]. In that phase we also compute all irredundant dominating sets, and output the edge polynomials. In the second phase, a Magma program invokes a univariate factorisation algorithm to perform the partial boundary factorisations, and the results are directed into the third phase program. In this last phase, a search for coprime dominating edges factorisations is performed, and when appropriate, the lifting process is started. The polynomial arithmetic was performed using classical multiplication and division, and the triangular systems were solved using dense Gaussian elimination over $\mathbb{F}_2$.

We generated a number of random experiments as follows: The input polynomial f was constructed by multiplying two random polynomials g and h of degree $d/2$, each with a given number of nonzero terms. Specifically, for each polynomial the given number of exponent vectors (e_1, e_2) were chosen uniformly at random subject to $0 \leq e_1 + e_2 \leq d/2$. These vectors always included ones of the form $(e_1, 0)$, $(0, e_2)$ and $(e_3, (d/2) - e_3)$ to ensure the polynomial was of the correct degree and had no monomial factors. As the polynomials chosen were sparse the corresponding Newton polytopes had very few edges. In all these cases, the components of edge vectors of Newt(f) had a very small gcd, so that the edges had few integral points and consequently the polygon itself had very few summands. The table below gives the running times (in seconds) of the total factorisation process to find at least one non-trivial factor involving all three phases described above. Here s is the number of nonzero terms of the input polynomial f; #Newt(f), #Newt(g), and #Newt(h) are the total number of lattice points in Newt(f), Newt(g) and Newt(h) respectively; and t is the total running time in seconds. The actual polynomials f, g and h in each of the five cases are also listed.

Table 6.1: Run time data for random experiments.

d	s	#Newt(f)	#Newt(g)	#Newt(h)	t
50	14	561	166	50	2.3
100	16	2234	472	222	11.6
500	15	52940	12758	11282	21.5
1000	30	206461	28582	56534	42.9
2000	28	848849	133797	132932	619.7

$d = 50$:

$f = x^9 + x^{18} + x^{22}y^8 + x^{14}y^{16} + (x^4 + x^{13})y^{20} + (x^8 + x^{17})y^{21} + x^{18}y^{24} + x^{17}y^{28} + x^{21}y^{29} + x^1y^{32} + y^{36} + x^4y^{37}$,

$g = x^4 + x^{13} + x^{17}y^8 + y^{16}$,

$h = x^5 + x^1y^{16} + y^{20} + x^4y^{21}$.

$d = 100$:

$f = x^{26} + x^{29}y^3 + x^{31}y^5 + x^{34}y^8 + x^{20}y^{13} + x^{25}y^{18} + x^6y^{19} + (x^9 + x^{48})y^{22} + x^{53}y^{27} + y^{32} + x^{28}y^{41} + x^{11}y^{45} + x^{14}y^{48} + x^5y^{58} + x^{33}y^{67}$,

$g = x^{20} + x^{25}y^5 + y^{19} + x^5y^{45}$,

$h = x^6 + x^9y^3 + x^{28}y^{22} + y^{13}$.

$d = 500$:

$f = x^{99} + x^{151}y^{30} + x^{176}y^{130} + x^{151}y^{142} + x^{228}y^{160} + x^{99}y^{181} + x^{56}y^{220} + x^{43}y^{223} + x^{108}y^{250} + x^{228}y^{272} + x^{176}y^{311} + x^{120}y^{353} + x^{108}y^{362} + x^{56}y^{401} + y^{443}$,

$g = x^{56} + x^{108}y^{30} + x^{108}y^{142} + x^{56}y^{181} + y^{223}$,

$h = x^{43} + x^{120}y^{130} + y^{220}$.

$d = 1000$:

$f = x^{727} + x^{678}y^3 + x^{935}y^{13} + x^{886}y^{16} + x^{679}y^{67} + x^{600}y^{79} + x^{887}y^{80} + x^{551}y^{82} + x^{469}y^{86} + x^{420}y^{89} + x^{448}y^{93} + x^{399}y^{96} + x^{279}y^{136} + x^{636}y^{143} + x^{552}y^{146} + x^{487}y^{149} + x^{421}y^{153} + x^{844}y^{156} + x^{400}y^{160} + x^{152}y^{215} + (x^{21} + x^{509})y^{222} + (1 + x^{378})y^{229} + x^{357}y^{236} + x^{611}y^{251} + x^{562}y^{254} + x^{563}y^{318} + x^{163}y^{387} + x^{520}y^{394}$,

$g = x^{448} + x^{399}y^3 + x^{400}y^{67} + y^{136} + x^{357}y^{143}$,

$h = x^{279} + x^{487}y^{13} + x^{152}y^{79} + x^{21}y^{86} + y^{93} + x^{163}y^{251}$.

$d = 2000$:

$f = x^{875} + x^{856}y^6 + x^{1469}y^{18} + x^{1450}y^{24} + x^{776}y^{66} + x^{1370}y^{84} + x^{722}y^{157} + x^{703}y^{163} + x^{963}y^{190} + x^{944}y^{196} + x^{623}y^{223} + x^{864}y^{256} + x^{487}y^{291} + x^{468}y^{297} + x^{647}y^{334} + x^{628}y^{340} + x^{982}y^{375} + x^{548}y^{400} + x^{235}y^{514} + x^{476}y^{547} + x^{769}y^{619} + x^{1363}y^{637} + x^0y^{648} + x^{160}y^{691} + x^{616}y^{776} + x^{857}y^{809} + x^{381}y^{910} + x^{541}y^{953}$,

$g = x^{487} + x^{468}y^6 + x^{388}y^{66} + y^{357} + x^{381}y^{619}$,

$h = x^{388} + x^{982}y^{18} + x^{235}y^{157} + x^{476}y^{190} + x^{160}y^{334} + y^{291}$.

6.10 Conclusion

In this chapter we have investigated a new approach for bivariate polynomial factorisation based on the study of their Newton polytopes. The approach combines results on polytopes with generalised Hensel lifting. In standard Hensel lifting, one lifts a factorisation from a single edge, and uniqueness can be ensured by randomising the polynomial to enforce coprimality conditions and make sure the edge being lifted from is sufficiently long. However, this randomisation is by substitution of linear forms which destroys the sparsity of the input polynomial. We show how uniqueness may be ensured in the bivariate case without destroying the sparsity of the polynomial, only under certain coprimality conditions, and without restrictions on the lengths of the edges. For certain classes of sparse polynomials, namely those whose Newton polytopes have few Minkowski decompositions, this gives a practical new approach which greatly improves upon Hensel lifting. As with Hensel lifting, our method has an exponential worst-case running time; however, we have demonstrated the practicality of our algorithm on several randomly chosen composite and sparse binary polynomials of high degree.

Chapter 7

An efficient sparse adaptation of the polytope method over $\mathbb{F}_p$ and a record-high binary bivariate factorisation

7.1 Introduction

In the previous chapter, we 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 the earlier chapter 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 a 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. 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 Chapter 6).

The rest of this chapter is organised as follows: In Section 7.2 we describe the model of sparse polynomials to which this algorithm is best suited. In Section 7.3 we describe the implementation in C and the sub-routines comprising the pre-lifting stages. In Section 7.4 we present our sparse adaptation which affects the polytope method at the lifting stage. In Sections 7.5 and 7.6 we analyse the complexity of the sparse method, and in Section 7.7 we report on the run times of our experiments producing high record degree factorisations over $\mathbb{F}_2$.

7.2 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 [2], 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

$$t^3 < d^2.$$

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^ϵ terms each, for some constant $0 < \epsilon < 1$. 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

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

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 $Q = \text{Newt}(g)$ and $R = \text{Newt}(h)$. 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 $\max(t_g, t_h) = O(t^\epsilon)$, for $0 < \epsilon < 1$.

Generic shape of Newt(f)

We now describe few aspects characterising the generic shape of Newt(f) for a non-trivial input f. By non-trivial we refer to the case when f is 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 integers $e_1, e_2 \geq 0$, and $f_{(e_1,e_2)} \in \mathbb{F}_p$.

Lemma 7.2.1 *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* Newt(f) *that lie on the x-axis and y-axis respectively.*

Proof: Recall that for a bivariate regular polynomial f, Newt(f) lies entirely in the quadrant of positive coordinates. Suppose that Newt(f) has at least two vertices lying on the x-axis and

y-axis respectively. Since Newt(f) is the convex hull of all exponents of f whose corresponding coefficients are nonzero, 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 $f_{(i,0)}x^i$ and $f_{(0,j)}y^j$, for $i, j = 1, ..., d$. Then the points $(i, 0)$ and $(0, j)$ belong to Newt(f). We want to show that $p = (i, 0)$ cannot but be a vertex or lying on an edge of 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 $f_{(i,0)}x^i$ or $f_{(0,j)}y^j$). Suppose to the contrary that p is a strictly interior point of the polytope. By convexity of 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 Newt(f) contains points in the half-plane

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

a contradiction, since Newt(f) lies entirely in the quadrant of positive coordinates.

Corollary 7.2.1 *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 integers $i, j \geq 0$, $f_{(i,j)} \in \mathbb{F}_p$, if and only if* 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, Newt(f) would then have at least two vertices on the x and y axes. The converse is immediate to establish.

7.3 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.

7.3.1 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. [53]), 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.

7.3.2 Computing Newt(f)

The first phase of the algorithm consists in computing 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 [60]), 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 [107]. 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 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 rightmost of all input points. If m denotes the total number of vertices of 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 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.

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 $O(t^\epsilon d)$ bits of memory, for some constant ϵ, $0 < \epsilon < 1$.

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 7.2.1, 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 $fterms[k-1]$ denotes the number of terms of f whose degree in y is k. 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 Newt(f) is of the order $O(t)$ [35], 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 $O(t)$.

7.3.3 Finding all irredundant sets of dominating edges

For determining all irredundant sets of dominating edges we use the algorithm reported in Chapter 6. Recall that the procedure depends on the notion of *admissible slopes* associated with a vertex V_i, and denoted by admiss(i), which designates the range of slopes of all straight lines that can be drawn through V_i such that their intersection with the polytope is only one point. It was shown that such straight lines have to be lying in the angular sector defined by the two edges $v_{i-1}v_i$ and v_iv_{i+1}, and that a necessary condition for any set of edges connecting two distinct vertices v_i, v_j to form an irredundant set of dominating edges is that admiss(i) $\cap$ admiss(j) $\neq \emptyset$. The input to this sub-routine are the vertices of Newt(f) as computed above. The process can be achieved through several implementations of the following:

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.

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 admiss(i) $\cap$ 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 two dominating sets of edges comprising the counter-clockwise and clockwise sequence of edges connecting them. A further condition that examines the difference of two intervals representing intersections of admissible slopes of various dominating sets is required to select the irredundant ones (see Chapter 6. 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, and to the direction of the set connecting the two vertices (whether clockwise or counterclockwise). In Chapter 6, the entire procedure has been shown to require an order of $O(m^2)$ operations of set intersections and complements. From the discussion above, these 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 $O(m^4)$ bit operations (see Chapter 6). 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 admiss(i) for some i), as well as three 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.

7.3.4 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 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 6.8.1.

Identifying integral points

Let $\delta \in \text{Edge}(f)$ 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 Chapter 8, we know that, for a polynomial $f \in \mathbb{F}_p[x, y]$ of degree d, each edge of

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.

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 Chapter 6. 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 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 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 6.8.1 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 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.

7.3.5 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 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 $\delta \in \text{Edge}(f)$. 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 Newt(f) in some point, this should lie in the smallest rectangle $\mathcal{R}$ containing 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 7.3.1 *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* Newt(f)*, or the empty set (where the latter implies that the line does not contribute to any terms in the lifted polynomials).*

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 $(i', k') \in$ 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 $(i'', k'') \in$ 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 Newt(f). By convexity of 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 #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 $\delta \in$ Edge(f). For this, we adopt the test for "leftedness" suggested in [107]: 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 7.3.2 *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 7.3.1 *Algorithm 7.3.2 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 [107]. 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 7.3.1 *Algorithm 7.3.1 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 7.3.1 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.

7.3.6 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 7.3.2 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 is 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 7.3.3 *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 7.3.2 *Algorithm 7.3.3 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 7.3.2 *Assuming that the above representation of IP is obtained as a precomputation, Algorithm 7.3.1 for finding the intersection of an arbitrary straight line with* Newt(f) *can be modified to require $O(d)$ 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 $Intersection'$.

7.3.7 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 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 [8]) 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

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

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 $j = 0, ..., d_\delta$. Let h denote the total number of irreducible factors of $F_0^{(\delta)}$. We define two integer arrays, mul_δ and $\deg_\delta$, of size h each, such that $\text{mul}_\delta[s]$ denotes the multiplicity in F_0^δ, and $\deg_\delta[s]$ denotes the degree, of the s irreducible in $irred_\delta$. The crucial idea behind our approach is that any degree j factor R_j of $F_0^{(\delta)}$ has to satisfy the following:

1. $R_j = (irred_\delta^{(0)})^{p_0} \cdot (irred_\delta^{(1)})^{p_1} \cdots (irred_\delta^{(s)})^{p_s}$, 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. $p_k \leq \text{mul}_\delta[k]$, for $k = 0, ..., s$,
3. $\sum_{k=0,...,s} \deg_\delta[k] \cdot p_k = j$.

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 $p_k = 0, ..., \text{mul}_\delta[k]$, 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 7.3.4 *Input: A degree d_δ univariate polynomial $F_0^{(\delta)}$ factorised completely into powers of irreducibles.*
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 $\sum_{k=0,...,tail} \deg_\delta[k] \cdot p_k = j$, 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 7.3.3 *Algorithm 7.3.4 works correctly as specified. When used to determine the number of factors of all possible degrees, the algorithm requires at most $O(d^{1+h}h)$ bit operations*

and $O(td)$ bits of memory, where h denotes the maximum number of irreducible factors of edge polynomials $F_0^{(\delta)}$ over all $\delta \in \text{Newt}(f)$[†].

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, $1 \leq j \leq d$, since h denotes the maximum number of irreducible factors of edge polynomials $F_0^{(\delta)}$ over all $\delta \in \text{Newt}(f)$, there are $O(h)$ irreducibles (or spots) for every edge, and each spot k can have at most $\text{mul}_\delta[k] \leq O(d)$ choices. When each choice is made a test is performed involving at most $O(h)$ multiplications and additions of integers bounded by d. Since $j \leq d$, the total run time now follows.

It can be easily seen that the list $irred_\delta$ has size at most $O(h)$. 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 $O(h)$ in value and thus fit in a machine word, as $h \leq d$. Since multiplicities and degrees of all irreducible factors are bounded by d, the arrays mul_δ and $\deg_\delta$ are of size $O(h)$ 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.

7.3.8 Summand counting and recovering algorithm

The summand counting algorithm used in Chapter 6 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 $\text{Newt}(f)$, 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 7.3.4. By Lemma 13 of [45], an integral polygon is a summand of $\text{Newt}(f)$ 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

[†]Note that although $h \leq d$, it has been shown in [83, 84, 100] that h is approximately $\log d$, so that in practice, the counting algorithm achieves its output in a reasonable amount of time

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 7.3.5 *Input: The edge description* $\{n_i e_i\}_{i=0,...,m-1}$ *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* 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* $i = 0, ..., m - 1$*, 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 7.3.4 *Algorithm 7.3.5 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 7.3.3 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 Newt(f), is $O(d)$. The naive method has to count over $O(d^m) = O(d^t)$ different summands of 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 6.8.1 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 Newt(f).

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 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 7.2.1 requires that we seek a vertex description allowing the proper translation of Q and R according to the fact below:

Corollary 7.3.3 *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* 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 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 7.2.1, 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 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 $\{q_i = c_ie_i\}_{i=0,...,m-1}$ denote the edge sequence of a summand of 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 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 $j = i+1, ..., m-1$, 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 7.3.6 *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 of correctness and that the algorithm requires $O(t)$ bit operations is immediate, bearing in mind 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.

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 $gn_{\delta_i'}$ falling on every edge δ_i' of Q, using the $O(d)$ technique of Subsection 7.3.4. We also note that the location in memory used to store information about the summands can be reused upon each new choice.

7.3.9 Choosing coprime dominating edges factorisations

As a result of the changes we have introduced to the summand counting algorithm recommended in Chapter 6, the procedure with which we choose a coprime edges factorisation has to be modified as well. 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 $n_ie_i, ..., n_{j-1}e_{j-1}$ 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. $G_0^{(\delta_i)} = irred_0^{p_0} \cdots irred_s^{p_s}$, 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. $\sum_{k=0,...,s} p_s \cdot \deg_{\delta_i}[k] = ||\delta_i||$.

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

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

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 7.3.7 *Input: A fixed integral decomposition* 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* 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 edge sequence comprising Γ*:*
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 7.3.5 *Algorithm 7.3.7 works correctly and requires $O(t^2 d^{1+h} h)$ bit operations and $O(td)$ bits of memory, where h denotes the maximum number of irreducible factors of edge polynomials $F_0^{(\delta)}$ over all $\delta \in \mathrm{Newt}(f)$.*

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 7.3.3 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 7.3.3. 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 7.3.3, finding all possible choices of polynomials per edge is of the order $O(d^{1+h}h)$. Across all edges of Γ, this becomes of the order $O(md^{1+h}h)$. 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 $O(m^2d^{1+h}h) = O(t^2d^{1+h}h)$.

Now, since any of the summands Q and R will have at most $\#\text{Newt}(f) = O(d^2)$ 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 $\text{Newt}(f)$, 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 $\delta \in \text{Edge}(f)$ 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 [2]. 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 7.3 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 [2], 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 7.3 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.

7.4 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 $O(d^{1+h}h)$ bit operations and $O(t^2 + d)$ bits of storage. As noted in [2], 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 7.1 and 7.2 demonstrate that in practice, and given very sparse polynomials such as those defined in Section 7.2, 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 [2].

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

Algorithm 7.4.1 *Input: A* $(\Gamma, K; Q, R)$ *coprime dominating edges factorisation, where* $K = (\underline{1})_{\delta_i \in \Gamma}$, *for* $i = 0, ..., m' - 1$.
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:*
. $u_{\delta_i'} < gn_{\delta_i'}$
. $u_{\delta_i'} = \max(u_{\delta_s'})_{s=0,...,m'-1}$

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 6.8.1, which has been proven to terminate either with a failure or with a factor of f (see [2] 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 [2], 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.

7.4.1 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 7.4.2 *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 7.4.1 *Algorithm 7.4.2 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 [2]. But these are precisely the points given by:

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

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 7.4.3 *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 $g_{(i,j)}x^i y^j$ 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 7.4.2 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 7.4.2.

7.4.2 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 7.4.4 *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) *between and including 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 7.4.2 *Algorithm 7.4.4 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 [2], 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.

7.5 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.

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

The most basic of these tasks is 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 7.5.1 *$Sum(f_0, f_2)$*
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
2.1.1: Set $sum[index] \leftarrow f_1[index_1]$.
2.1.2: Set $index_1 \leftarrow index_1 + 1$.
2.1.3: Set $index \leftarrow index + 1$.
2.2: Else, if $f_0[index_0] < f_1[index_1]$ and $index_0 < t_0$ do
2.2.1: Set $sum[index] \leftarrow f_0[index_0]$.
2.2.2: Set $index_0 \leftarrow index_0 + 1$.
2.2.3: Set $index \leftarrow index + 1$.
2.3: Else, if $f_0[index_0] = f_1[index_1]$ do
2.3.1: Let c_0 and c_1 denote the coefficients of terms whose exponents are $f_0[index_0]$ and $f_1[index_1]$ respectively.
2.3.2: If $c_0 + c_1 \neq 0 \bmod p$, set $sum[index] \leftarrow f_0[index_0]$, and $index \leftarrow index + 1$.
2.3.3: If $index_0 < t_0$ set $index_0 \leftarrow index_0 + 1$.
2.3.4: 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 7.5.1 *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 $O(t_0 t_1)$ bit operations, and requires $O(t_0 t_1)$ bits of memory for storing the final product.

7.5.1 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

$$G_{k_\delta} - V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] = e(z) G_0, \tag{7.1}$$

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 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 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 7.5.1 *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* 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_g and t_h terms respectively, such that* $\max(t_g, t_h) = O(t^\epsilon)$ *for some constant ϵ satisfying $0 < \epsilon < 1$. Then, given the decomposition* Newt(f) = Newt(g) + Newt(h) *such that* 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,* Newt(g) *and* Newt(h)*, there will be at most $O(t^\epsilon)$ 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 (6.2).*

Proof: Consider all possible liftings from some edge $\delta \in \Gamma$, and let δ' denote its summand in Newt(g). Since g has $O(t^\epsilon)$ nonzero terms, Newt(g) will have $O(t^\epsilon)$ lattice points which correspond to specialised nonzero terms of g. In the worst-case analysis, none of these points will fall on the same translate of the supporting line of δ' into Newt(g). In that case, the lifted polynomials whose terms correspond to lattice points of Newt(g) on these translates will be nonzero, and there will be at most $O(t^\epsilon)$ 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 $s < max\_lift$, 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 bits of memory for $\mathcal{G}$, and $O(t^\epsilon)$ bits for the array G_{poly}. For all edges, this amounts to $O(m(d+t)) = O(t(d+t))$ 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 7.5.2 $Form\_sum(k_\delta)$
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 7.5.2 *Algorithm 7.5.2 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 $O(t^\epsilon)$ 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 7.5.1, 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 $O(t^\epsilon)$ 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 7.5.3 *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

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

or

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

. 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 7.5.3 *Algorithm 7.5.3 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 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 $f_{(i,j)}x^i y^j$ 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

$$F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j} \tag{7.2}$$

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

$$(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0 \tag{7.3}$$

using division with remainder for Laurent polynomials (see Chapter 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 (7.3) 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

$$V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] \tag{7.4}$$

using $O(d^2)$ bit operations, since $\deg(V) < \deg(G_0) = O(d)$. From the discussion on Laurent polynomial division with remainder in Chapter 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 $O(d)$.

7.5.2 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. We represent G_{k_δ} temporarily using an 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 nonzero or unspecialised, and assuming that entries in $temp_g$ are stored in increasing order of exponents, we define the *possible* degree of the unknown polynomial 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 7.5.4 *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

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

or

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

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 $temp_g$.

Data structure for expressions using unknown polynomials

Another example of an unknown symbolic entity is the expression

$$G_{k_\delta} - V[(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) \bmod G_0] \tag{7.5}$$

where as we have seen above, G_{k_δ} is a polynomial whose coefficients are partially specialised, and where 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 (7.1). 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 (7.4) 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 (7.4) 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 (7.4), 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 (7.1), 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 (7.5). 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 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 (7.4), 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 (7.4). In total, this requires that we treat *lhs* as a double array of size $2 \times O(d)$. We now have the following:

Algorithm 7.5.5 *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 (7.4) 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 6.8.1 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 (7.4).

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 (7.1). 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 7.5.6 *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 6.8.1 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$(a,b) = -1$

1.3.1: 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 7.5.4 *Algorithm 7.5.6 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 [2] that each triangular system arising from comparing coefficients on both sides of $LHS = e(z)G_0$ can be solved uniquely. We now claim the following:

Lemma 7.5.2 *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 $O(t^\epsilon d)$ nonzero elements over $\mathbb{F}_p$, for some constant ϵ such that $0 < \epsilon < 1$.*

Proof: Write

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

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

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

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, for instance, 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 $O(t^\epsilon)$ coefficients of G_0 are nonzero, for $0 < \epsilon < 1$, it follows that each row of A contains at most $O(t^\epsilon)$ 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 $O(t^\epsilon d)$ 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 $O(t^\epsilon t^\epsilon d) = O(t^2 d)$ bit operations with no more than $O(t^\epsilon d)$ bits of temporary storage memory using any of the well known sparse direct methods (see for instance [33] 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 7.5.6 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.

7.5.3 Recovering H_{k_δ}

Invoking Algorithm 7.5.4 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

$$H_{k_\delta} = \frac{(F_{k_\delta} - \sum_{j=1}^{k_\delta - 1} G_j H_{k_\delta - j}) - G_{k_\delta} H_0}{G_0}. \tag{7.6}$$

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^ϵ 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 7.5.6, using at most $O(td)$ bit operations.

7.6 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 7.6.1 *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_g and t_h terms respectively, such that* $\max(t_g, t_h) = O(t^\epsilon)$ *for some constant ϵ satisfying $0 < \epsilon < 1$. 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 $\Gamma, \mathrm{Newt}(g)$ and* $\mathrm{Newt}(h)$*, there exists one full factorisation of f which extends it in $O(td^2) + O(t^3 d)$ bit operations and $O(t^\epsilon d)$ 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 6.6.1 of Chapter 6. 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 $max\_lift \cdot O(td) = O(td^2)$ bit operations and a temporary $O(d)$ bits of memory. We have also shown that computing the quantity in (7.2) requires in the worst-case analysis $O(t)$ bit operations and $O(t)$ bits of temporary storage, so that the total cost is $max\_lift \cdot O(t)$ $= O(td)$ bit operations and $O(t)$ bits of memory. Computing the polynomial in (7.4) 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 (7.2) is nonzero, it suffices to determine the maximum number of times that the latter can happen in order to obtain the total cost of long division throughout the lifting stage. Note that the polynomial (7.2) is nonzero in at most one of these cases:

- 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 7.5.1, there exist at most $O(t^\epsilon)$ nonzero polynomials G_j and H_i, for $i, j = 1, \ldots, k_\delta - 1$, where $i + j = k_\delta \leq max\_lift$, 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 $\sum_{j=1,\ldots,k_\delta-1} G_j H_{k_\delta - j}$ is nonzero whenever one pair $G_j H_{k_\delta - j} \neq 0$ for some fixed j. Hence, 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 (7.3) 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 (7.4) 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 (7.3) is nonzero. In the worst-case analysis, this in turn is nonzero whenever the polynomial (7.2) 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 (7.4) 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 $O(t^2d)$ bit operations and $O(t^\epsilon d)$ 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 (7.4) 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 Newt(g), and the number of integral points on δ' of 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 (7.4) 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 (7.4) 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 $O(t^\epsilon)$ of the cases, one has to set up and solve a triangular system at most $O(t)$ times. The total cost for solving any of the triangular systems is hence $O(t^3d)$ bit operations, and $O(t^\epsilon d)$ 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 (7.6) is non-trivial only when the numerator is nonzero. This, in turn, happens in at most one of the two cases:

- The polynomial in (7.2) 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 7.5.6 requires at most $O(td)$ bit operations, so that in total this will be at most $max\_lift \cdot O(td) = O(td^2)$ bit operations. Since the total number of terms of all

such polynomials should not exceed $O(t^\epsilon)$, 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 associated with the decomposition $\text{Newt}(f) = \text{Newt}(g) + \text{Newt}(h)$ can be extended using at most $O(td^2) + O(t^3 d)$ bit operations and $O(t^\epsilon d)$ 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 $\text{Newt}(f) = O(d^2)$, the standard polytope method requires $O(max\_lift \cdot d^3) = O(d^4)$ bit operations in total and $O(d^2)$ bits of memory. When

$$t^3 < d^2,$$

we certainly have that $t < d$, and hence

$$O(t^2 d) + O(t^3 d) \subset O(d^3).$$

By Theorem 7.6.1, the sparse adaptation outperforms the dense one in both the operational and spatial complexity.

7.7 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 7.2 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 $\text{Newt}(f) = Q + R$, 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 $\text{Newt}(f)$, 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 7.1 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 7.2 increase upon incrementing either the degrees or the terms of the input polynomials. Note that in almost all cases Newt(f) has extremely few non-trivial integral decompositions, as predicted earlier in [2] 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 Newt(f).

Table 7.1: 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 7.2: 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

7.8 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 [46], 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 $O(td^2) + O(t^3 d)$ bit operations and $O(t^\epsilon d)$ bits of memory, $0 < \epsilon < 1$, 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. To the best of our knowledge, the highest dense bivariate factorisation to date achieved using Hensel lifting techniques is for a dense polynomial of degree 2000 over $\mathbb{F}_{17}$ [13].

Chapter 8

Parallel absolute irreducibility testing via polytopes

8.1 Introduction

Absolute irreducibility testing of polynomials is of importance in various fields in algebra and geometry (see for instance [6, 68, 92, 124, 125]). Convex polytopes have been studied in connection with multivariate polynomials over arbitrary fields. We have seen how recent work has examined the connection that convex polytopes bear to factorising bivariate polynomials [2], and to testing absolute irreducibility of multivariate polynomials [43, 45], over arbitrary fields. In the latter case, the problem of testing indecomposability of Newton polytopes formed part of a pseudo-polynomial time algorithm (see [48]) for bivariate absolute irreducibility testing, and of a heuristic and randomised algorithm in the multivariate case. The fact that the nonzero coefficients of the input polynomial do not matter in the testing process makes it possible to show absolute irreducibility of families of polynomials, rather than single polynomials [45]. The empirical sequential tests and timing results reported in [47] indicated a high success rate for a large class of low degree and sparse multivariate polynomials, whose number of terms is bounded by $O(nd)$, where n is the number of variables and d is the upper bound on the degree in each variable. In the instances when the algorithm is highly successful and the input polynomials have a number of nonzero terms bounded by a constant multiple of their total degree, the runtimes have been shown to be at most cubic in the total degree of the input polynomial [47]. For a degree q polynomial with n variables and coefficients from a finite field, for instance, the input size of the irreducibility problem is $N = O(q^n)$, ignoring logarithmic factors. Expressing the complexity of Gao and Lauder's algorithm [45] in terms of the input size, this requires $O(N^{\frac{3}{n}})$, which for polynomials with more than 3 variables implies a run time that is almost linear in the input size. As such, the algorithm can be used as a fast pretest before any of the infallible yet slower irreducibility tests are invoked [34, 44, 51, 63, 75, 89, 90].

Motivated by these original findings and the special feature which makes absolute irreducibility testing dependent in large upon the shape and the size of Newton polytopes, we investigate a parallel scheme mainly set to widen the range of applicability of the algorithm, by making it possible to tackle significantly higher degree, sparse polynomials, and by allowing a more efficient performance for low degree yet denser polynomials. We show that the algorithm can be efficiently parallelised, thanks to some of its inherent features: First, we discuss a balanced

load scheme which can be constructed using the pattern of computations in the sequential case in [45]. Second, we discuss how a corresponding data distribution representing lattice points inside polytopes in $\mathbb{R}^2$ can be constructed, adhering to the balanced load scheme, and allowing a scalable parallel algorithm aimed at high degree irreducibility testing. We adopt the BSP model for parallel computation [15, 66, 67, 127], and we analyse the conditions necessary for an efficient parallel performance in the bivariate case. This then serves as a sub-problem for the multivariate case, where a model involving parallelism at two different levels is described. The BSP algorithm makes it possible to test significantly higher degree polynomials than can be allowed in the sequential case in [45], and hence, to the best of our knowledge, using any other known absolute irreducibility testing algorithm. We further study both parallel models for issues relating to efficiency, and establish some conditions under which a good performance is guaranteed. Our empirical results agree closely with the theoretical estimates, and we report on our implementation in this respect.

The reader may wish to recall some terminology and results relating to polynomials and polytopes from Chapter 3. In Section 8.2 we study a parallel scheme for bivariate polynomial irreducibility testing, and in Section 8.3 we address the multivariate case. Finally, empirical results are presented and analysed in Section 8.4.

8.2 Parallel bivariate absolute irreducibility testing

In all the following we shall assume that nd fits in a machine word for an input polynomial with n variables and upper bound d on each variable. We will thus measure run-time in bit operations and space in bits.

Since the Newton polytope of a bivariate polynomial already lies in $\mathbb{R}^2$, a strong feature of Algorithm 3.3.2 is that it will always decide indecomposability of polygons, and hence will always establish absolute irreducibility of the input bivariate polynomial f if $\text{Newt}(f)$ is indecomposable. Initial empirical results in [47] indicate that the algorithm has a high success rate when the number of terms is bounded by a constant factor of the total degree, and that the probability of success increases with increasing degrees of input polynomials. While these observations still stand as conjectures based on empirical results, it is important to extend the range of applicability of the algorithm for higher degrees, and to support initial arguments related to the rate of success of the algorithm. However, as degrees of the input bivariate polynomials increase, so do the sizes of their Newton polytopes, and hence one major difficulty limiting attempts to improve the performance of Algorithm 3.3.1 becomes the space requirement. Yet, the fact that one can shift the focus from a data structure and distribution representing bivariate polynomials to one that represents shapes of Newton polytopes in $\mathbb{R}^2$ suggests that a parallel re-construction of the irreducibility testing algorithm which exploits the shapes of the polygons can achieve considerable improvements, both in the operational and spatial complexities. In the remainder of this section, we discuss our first BSP model for a bivariate absolute irreducibility testing based on a parallel polygon indecomposability testing algorithm.

Let $f \in \mathbb{F}[x, y]$ be of total degree q with no non-constant monomial factors and let IP denote the set of integral points belonging to $\text{Newt}(f)$. Since all vectors in $\text{Supp}(f)$ have positive integral coordinates that are at most equal to q, $\text{Newt}(f)$ can be embedded within a square of dimension $O(q^2)$. Moreover, Algorithm 3.3.1 requires a description of IP in two different ways. First, one has to have, at the beginning of the algorithm, a structure by which one can test an arbitrary

point for inclusion in IP. Second, the algorithm re-constructs integral points in IP following paths of the form $v_0 + \sum_{0 \leq i \leq j} k_i e_i$, for $0 \leq k_i \leq n_i$ and $j = 0, ..., m-1$. In the following discussion, we aim to show how the problems associated with each of the two representations can be circumvented by reducing the quadratic spatial factor.

8.2.1 Computing the set of all integral points in a polygon

Often in the innermost loops of Steps 2 and 3 of Algorithm 3.3.1, one will have to test for inclusion of arbitrary points in IP. Computing and then storing the set of all integral points belonging to the polytope requires about $O(q^2)$ bits of storage, which can become highly restrictive even for moderately large input degrees. As has been already noted in Chapter 7, the alternative approach which does not require that we store any lattice points but performs a test of inclusion based on the "leftedness" of an arbitrary point with respect to all directed edges of the polytope has a cost of $O(m)$ integer operations. In our application, it can be costly to invoke this test very frequently, incurring a cost of $O(m^2)$ for each loop iteration in Steps 2 and 3 of Algorithm 3.3.1 above, and hence a cost of $O(t'm^3N)$ integer operations in total, where t' is the total number of lattice points in Newt(f) and N is the maximum number of integral points along any edge of Newt(f). We hence adopt the strategy earlier introduced in Chapter 7 which requires that we store only a "useful" subset of IP with no more than $O(q)$ points. Let x_{min}, x_{max}, y_{min} and y_{max} denote respectively the lowest and highest x coordinates, and the lowest and highest y coordinates, of all $e \in \text{Supp}(f)$. We then store only the intersection points between all horizontal (or vertical) lines $y = k$, for $k = y_{min}, ..., y_{max}$, using only $y_{max} - y_{min} + 1 = O(q)$ bits of memory. However, unlike the case in Chapter 7 where computations are performed such that all output is integral, we need a weaker condition affecting the following. The intersection of the polytope with a horizontal straight line is obtained by intersecting the line with at most all of the polytope's edges (until two points of intersection are found, not necessarily distinct). If $ux + vy + w = 0$ denotes the generic equation of a line ℓ' defining an edge of Newt(f), then finding the intersection of ℓ' with $y = k$ requires that we solve for x in

$$ux + vk + w = 0 \text{ or } x = \frac{-w - vk}{u},$$

when $u \neq 0$. Although this division operation involves only integral values, the quotient itself may not be an integer, in which case x has to be declared as a float or a double, for otherwise, the division might produce a rounded-off integral value $\lfloor x \rfloor$, which may not correspond to a lattice point in IP. Note that computing this subset of IP must be performed as a pre-computation, requiring at most $O(qm)$ floating point operations, where m is the number of edges in Newt(f).

8.2.2 Constructing sets of points along paths of edges

Obviously, the above strategy of reducing the description of IP by preserving only a smaller subset of it cannot be extended when dealing with the main computations of Algorithm 3.3.1. Specifically, one needs to have all points of IP that are reachable via a subset of edges $e_0, ..., e_i$, for $0 \leq i \leq m-2$, in order to find a larger subset reachable via $e_0, ..., e_{i+1}$. Eventually, the last path requires that one has available almost all of the points in IP, and hence, the best that one can attempt is to distribute the points in IP among a fixed number of processors. Since this task is never so immediate, we need to address several of the following issues: Whether there exists at all any potential parallelism in the main computations of Algorithm 3.3.1 that

can make such an approach possible, whether there exists a balanced load scheme that ensures all processors are almost equally engaged in the independent computations, and finally, whether there exists a data distribution which not only ensures that a given number of n_p processors, say, store at most $O(q^2)/n_p$ lattice points in IP, but also adheres to the pattern of the proposed balanced load scheme.

8.2.3 Detecting independent computations

Recall that the main computations in Algorithm 3.3.1 build up iteratively by constructing subsets of IP in each step of the iteration. However, it is almost immediate to see that the inner-most computations of the iterative loop can themselves be independent, generating a "horizontal" inherent parallelism across each iteration of the main loop over edges of the input polygon. In particular, we have:

Lemma 8.2.1 *Algorithm 3.3.1 for polygon indecomposability testing contains two patterns of computations, one which describes a sequence of inter-dependent iterative steps for constructing new subsets of IP using previous subsets, and another pattern describing completely independent tasks for vector operations across a fixed iterative step.*

Proof: During a fixed stage $i = 0, ..., m-2$ of the main loop iteration (Step 2) of Algorithm 3.3.1, one finds all points $u' \in A_i$ satisfying the following:

1. $u' = v_0 + ke_i$, for $0 < k \leq n_i$, in which case the computations over all k require no information from the previous loop iteration of index $i-1$, and each computation per fixed k requires no information apart from e_i.

2. $u' = u + ke_i$, for $0 \leq k \leq n_i$ and for all $u \in A_{i-1}$, in which case the computations over all u require information from the previous loop iteration of index $i-1$, each computation per fixed u and over all $k = 0, ..., n_i$ requires no information apart from n_i and e_i, and each computation per fixed u and fixed k requires no information apart from e_i.

For $i = m-1$, one finds all points $u' \in A_i$ satisfying $u' = u + ke_{m-1}$, for $0 \leq k < n_{m-1}$ and $u \in A_{m-2}$, which involves the dependencies described in 2 above.

8.2.4 Constructing a balanced load scheme

We will now attempt to examine the geometric pattern of the computations above, on which we can base a possible load balancing scheme. To this end, we will consider a slight modification of Algorithm 3.3.1 to produce sets of points B_i in $\mathbb{R}^2$, for $0 \leq i \leq m-1$, constructed as follows:

1. Initialise $B_i \leftarrow \emptyset$, for $i = -1, ..., m-1$.

2. For $i = 0, ..., m-1$, compute the set of points of the plane that are reachable from v_0 via the vectors $e_0, ..., e_i$, and store them in B_i:

3. For each $u \in B_{i-1}$ and $k = 0, ..., n_i$, add $u + ke_i$ to B_i.

The differences between the sets A_i (defined in Algorithm 3.3.1 in Chapter 3) and B_i are that v_0 is in every single set B_i, that B_i contains points reachable via $e_0, ..., e_i$ which are not necessarily in IP, and that the points $v_0 + ke_{m-1}$, for $k = 0, ..., n_{m-1}$, do lie in B_{m-1}. As such, it is clear that $A_i \subseteq B_i$ for every i. Note that $\text{Newt}(f) \subseteq B_{m-1}$ by construction of B_{m-1}. Because the sets B_i have weaker conditions characterising their points, it will be simpler to describe the geometric pattern they follow. Since $A_i \subseteq B_i$ for every i, any such pattern will apply to elements of A_i. Moreover, we will define every region B_i to be an *active* region in the sense that all computations and possible communications across a main loop iteration of index i in Algorithm 3.3.1 are restricted to only those points belonging to B_i, but not to $\mathbb{R}^2 - B_i$.

Let P denote a convex polygon and w a vector in $\mathbb{R}^2$, and let $Tr_w(P)$ denote the image of P under translation by w. The following result gives an explicit geometric description of B_i, for $0 \leq i \leq m-1$.

Lemma 8.2.2 *Let P be a convex polygon with vertices $v_0, ..., v_{m-1}$ and edge sequence $\{n_i e_i\}$, for $0 \leq i \leq m-1$, where $e_i \in \mathbb{Z}^2$ are primitive vectors. For each iterative step $i = 0, ..., m-1$ of the polygon indecomposability test in Algorithm 3.3.1, the computations are restricted to active regions B_i of the plane representing points reachable from v_0 via $e_0, ..., e_i$. Furthermore, the regions can be defined inductively as follows:*

1. *For $i = 0$, $B_0 = conv(v_0, v_1)$,*
2. *For $1 \leq i \leq m-1$, $B_i = conv(\cup\{Tr_{ke_i}(B_{i-1})\}_{0 \leq k \leq n_i})$.*

Proof: We prove the assertion by induction on i. For $i = 0$, B_0 consists only of the points $v_0 + ke_0$, for $k = 0, ..., n_0$. But this spans all integral points along the first edge $E_0 = n_0 e_0$, so that $B_0 = conv(v_0, v_1)$. We now assume the assertion is true for $i \leq m-2$, i.e., that the set of points in B_i of the form $v_0 + \sum_{0 \leq i \leq m-2} k_i e_i$, for $k_i = 0, ..., n_i$, constitutes a convex polygon as defined in 2 above. Since all the points $u \in B_i$ lead to points u' in B_{i+1} obtained as

$$u' = u + je_{i+1} = (v_0 + \sum_{0 \leq i < m-1} k_i e_i) + je_{i+1}, \tag{8.1}$$

for $k_i = 0, ..., n_i$ and $j = 0, ..., n_{i+1}$, this reduces to translating all points of B_i by je_{i+1}, for all j. Since B_i is convex, its image is also a convex set, whose vertices are defined by the images under the translation $Tr_{je_{i+1}}$ of vertices of B_i. Let C denote the convex hull of all vertices in the union of the sets $Tr_{je_{i+1}}(B_i)$ over all $j = 0, ..., n_{i+1}$. We shall show that $C = B_{i+1}$. Since C contains all possible points in the sets $Tr_{je_{i+1}}(B_i)$, we have that $C \subseteq B_{i+1}$. On the other hand, and by Eq. (8.1) above, any point in B_{i+1} belongs to some set $Tr_{je_{i+1}}(B_i)$, so that $B_{i+1} \subseteq C$. This establishes our inductive proof.

8.2.5 Constructing a balanced data distribution

The above lemma provides the general guidelines under which a balanced load scheme can be chosen. In particular, it is immediate that the bulk of the work during any iterative step of Algorithm 3.3.1 takes place in well-defined active zones of the plane. One should thus avoid any form of data distribution whereby the polygon is triangulated into zones and each zone is exclusively assigned to one single processor. Specifically, this risks having some processors completely idle when others are engaged in the active zones. Instead, we propose the following:

Lemma 8.2.3 *Let B_i, for $0 \leq i \leq m-1$, denote an active region of the plane as defined in Lemma 8.2.2 above, and let b_i denote the total number of lattice points belonging to the smallest square containing B_i. Let n_p denote the total number of processors operating in parallel such that $1 \leq n_p < \sqrt{b_i}$. The data distribution of integral points in IP which allocates every point $(k,k') \in B_i$ to the processor with identification number $id \equiv (k+k') \bmod n_p$ allows for a balanced load scheme as required by Lemma 8.2.2 above, and assigns to each processor $O(b_i)/n_p$ integral points, where b_i represents an upper bound on the number of integral points in B_i.*

Proof: Consider each of the convex active regions B_i of the plane, for $i = 0, ..., m-1$, defined in Lemma 8.2.2 above. Let x_1, y_1, x_2 and y_2 denote respectively the lowest x and y coordinates and the highest x and y coordinates appearing in any point in B_i. Assume without loss of generality that B_i is translated so that $x_1 = y_1 = 0$. Then an upper bound on $(k+k')$ over all $k = x_1, ..., x_2$ and $k' = y_1, ..., y_2$ is equal to $\sqrt{b_i}$. Let t'_{id} denote the total number of lattice points of B_i assigned to a processor with identification number $id = 0, ..., n_p - 1$. For any integer $z \geq 0$ define the class associated with z:

$$[z] = \{(x,y) | x, y \in \mathbb{Z}, x, y \geq 0, \text{ and } x + y = z\},$$

and let $\#[z]$ denote the number of elements in this class. Then $\#[z] = z+1$. Given an arbitrary point (k,k') of B_i and $0 \leq id < n_p$, the distribution which maps (k,k') to processor $id \equiv (k+k') \bmod n_p$ assigns to it all classes of the form $[id + hn_p]$ such that

$$id \leq id + hn_p \leq \sqrt{b_i}.$$

Since $n_p < \sqrt{b_0}$, we have that $n_p < \sqrt{b_i}$, for $i = 1, ..., m-1$, and we can require

$$0 \leq h \leq \left\lfloor (\sqrt{b_i} - id)/n_p \right\rfloor.$$

Hence, the total number of points in B_i that are assigned to processor id is at most

$$t'_{id} = \sum_{h=0}^{\lfloor (\sqrt{b_i} - id)/n_p \rfloor} (id + hn_p + 1).$$

For $0 \leq id \leq n_p - 1$, we have

$$\begin{aligned}
t'_{id} &\leq \textstyle\sum_{h=0}^{\lfloor (\sqrt{b_i} - id)/n_p \rfloor} (n_p + hn_p) \\
&\leq \textstyle\sum_{h=0}^{\lfloor \sqrt{b_i}/n_p \rfloor} (n_p + hn_p) \\
&= \frac{\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor (\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor + 1)}{2} n_p + (\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor + 1) n_p \\
&\leq \frac{\frac{\sqrt{b_i}}{n_p} (\frac{\sqrt{b_i}}{n_p} + 1)}{2} n_p + (\frac{\sqrt{b_i}}{n_p} + 1) n_p \\
&= \frac{\sqrt{b_i}(\sqrt{b_i} + n_p) + 2n_p(\sqrt{b_i} + 1)}{2n_p} \\
&< \frac{2(\sqrt{b_i})^2 + 2\sqrt{b_i}(\sqrt{b_i} + 1)}{2n_p} \quad \text{for } n_p < \sqrt{b_i}, i = 0, ..., m-1 \\
&< \frac{2(\sqrt{b_i})^2 + 4(\sqrt{b_i})^2}{2n_p} \quad \text{since } \sqrt{b_i} > n_p \geq 1 \\
&= \frac{3b_i}{n_p}.
\end{aligned}$$

This establishes $t'_{id} = \frac{O(b_i)}{n_p}$.

In practice, and even though the condition $n_p < \sqrt{b_0}$ may not easily hold (as B_0 is simply the first edge of Newt(f), in which case b_0 denotes the number of integral points on that edge), we note that the sizes of the sets B_i, for $i > 0$, start growing fast immediately afterwards; specifically, since the number of lattice points in B_1 is at least four, we expect many more processors to be engaged in their assigned computations as soon as the first edge of the input polygon is examined. Another similar result affecting the data distribution is the following:

Lemma 8.2.4 *Let $f \in \mathbb{F}[x,y]$ be a non-constant polynomial with no non-constant monomial factors and IP denote the set of integral points in* Newt(f). *Let x_{min}, y_{min}, x_{max} and y_{max} denote respectively the lowest x and y coordinates and the highest x and y coordinates appearing in any point belonging to* Newt(f), *and write $\gamma = \max(y_{max} - y_{min}, x_{max} - x_{min})$. Let n_p denote the total number of processors operating in parallel such that $1 \leq n_p < 2\gamma$. The data distribution of integral points in IP which allocates every point $(k,k') \in$ Newt(f) to the processor with identification number $id \equiv (k+k') \bmod n_p$ assigns to each processor $O(\gamma^2)/n_p$ integral points, where γ^2 is an upper bound on the number of lattice points in* Newt(f).

Proof: Assume without loss of generality that Newt(f) is translated so that $x_{min} = y_{min} = 0$. Note that an upper bound on $(k+k')$ over all $k = x_{min}, ..., x_{max}$ and $k' = y_{min}, ..., y_{max}$ is equal to 2γ. Let t'_{id} denote the total number of lattice points of Newt(f) assigned to a processor with identification number $id = 0, ..., n_p - 1$. For any integer $z \geq 0$ consider the class associated with z defined in the proof of Lemma 8.2.3 above. Given an arbitrary point (k,k') of Newt(f) and $0 \leq id < n_p$, the distribution which maps (k,k') to processor $id \equiv (k+k') \mod n_p$ assigns to it all classes of the form $[id + hn_p]$ such that

$$id \leq id + hn_p \leq 2\gamma.$$

Since $n_p < 2\gamma$, we can require

$$0 \leq h \leq \lfloor (2\gamma - id)/n_p \rfloor .$$

Hence, the total number of points in Newt(f) that are assigned to processor id is at most

$$t'_{id} = \sum_{h=0}^{\lfloor (2\gamma - id)/n_p \rfloor} (id + hn_p + 1).$$

We can now proceed similarly as in the proof of Lemma 8.2.3 above and we obtain $t'_{id} = \frac{O(\gamma^2)}{n_p}$.

8.2.6 Removing repetitions in computation

Recall that in Algorithm 3.3.1 above, one has to test for repetitions in appending points u to each set A_i. The reason that this needs to be done is that it may be possible to find two points u and u' in A_{i-1}, and two positive integers $k, k' \leq n_i$, such that $u + ke_i = u' + k'e_i \in IP$. If left unchecked, this may produce up to $O(N)$ copies of the same point, so that in the worst-case scenario, each multi-set A_i will have $O(t'N^i)$ points, and the run-time of polygon indecomposability testing

will become exponential in the edges of the input polygon. Checking for repetitions in the sequential case can be made at a cost not exceeding that of an integer operation per point. In particular, one can use a double array of integers, say $Flag$, of total size t', and whose entries are all initialised to PASS. $Flag[k][k']$ is then set to FAIL when a point (k, k') is first added to A_i, so that any future attempts to add another copy of the point are halted upon a simple check of the value in $Flag[k][k']$.

The above strategy, however, can become extremely inefficient in the parallel setting as follows. Let $A_i(id)$ denote the set of points in IP that are reachable via $e_0, ..., e_i$ and that are assigned to processor id. Assume that $i < m - 1$ and that some processor p_a computes the vector operation $v = u + ke_i$, for some $u \in A_i(a)$, and some $k = 0, ..., n_i$. Suppose further that all processors have their own copy of a $Flag$ array, $Flag(id)$, labelling points that have already been added to their sets $A_i(id)$. If $v \in IP$ and v has to be assigned to some other processor p_b, then p_a has to read the information in the entry of array $Flag(b)$ corresponding to point v. This involves a communication step for every such point v. Moreover, reading from remote locations requires a synchronisation barrier for all processors to update their communicated values. This also requires a synchronisation step for every such point v. Consequently, a process like the above will require extremely expensive communication and synchronisation costs which can even overwhelm the computational cost of the sequential algorithm. Alternatively, we introduce the following iterative strategy that will later be shown to come at a very modest cost: For any processor $id = 0, ..., n_p - 1$, we know that $A_0(id)$ does not contain any redundant points. For $i > 0$, we proceed as follows:

- Start with a set $A_{i-1}(id)$ that contains no repeated occurrences of points in IP.
- Add all relevant points to their corresponding locations in $A_i(id')$, for some processor id' different from or equal to id, without any check on repetitions.
- When the computations across the iterative step of index i are over, remove all repetitions in $A_i(id)$.

More details illustrating where communication and synchronisation steps should be invoked in the above will be given next in our parallel algorithm for polygon indecomposability testing.

8.2.7 A BSP algorithm for testing polygon indecomposability

We are now ready to present our parallel algorithm based on the BSP model for parallelisation. The algorithm is designed as an SPMD model, where a single program with multiple data is encountered by all processors, which then execute their own version of the program, as distinguished by their own identification number, $id = 0, ..., p - 1$.

Algorithm 8.2.1 *(Parallel Polygon Decomposability Test)*
Input: The edge sequence $\{n_i e_i\}_{0 \leq i \leq m-1}$ of an integral convex polygon P starting at a vertex v_0 where $e_i \in \mathbb{Z}^2$ are primitive vectors, and a number n_p of processors such that $n_p \geq 1$. Let y_{min}, y_{max}, x_{min} and x_{max} denote the respective smallest and largest y coordinates, and the smallest and largest x coordinates, of vertices of P, and assume further that

$$n_p < 2\gamma,$$

where $\gamma = \max(x_{max} - x_{min}, y_{max} - y_{min})$.

Output: Whether P is decomposable.

Step 1: Define a double integer array Int of size $2(y_{max} - y_{min} + 1)$. Let id denote the processor identification number, and N denote the maximum over all n_i's, for $i = 0, ..., m-1$.

Step 2: Set $h \leftarrow y_{min} + id$, and while $h \leq y_{max}$, do:

2.1: Compute the points of intersection of the line $y = h$ with P.
2.2: Broadcast the x coordinates of the points of intersection to all processors at the entries $Int[h][0]$ and $Int[h][1]$ respectively.
2.3: Set $h \leftarrow h + n_p$.

Step 3: $bsp\_sync()$.

Step 4: Set $A_i \leftarrow \emptyset$, for $i = -1, ..., m-1$, and $Result \leftarrow Indecomposable$. Define an auxiliary array of integer vectors, Aux, of size $O(\gamma^2 N)/n_p$, an index array of integers, $Index$, of size n_p, and a flag double array of integers, $Flag$, of size $O(\gamma^2)/n_p$.

Step 5: For $i = 0, ..., m-2$, compute the set $A_i(id)$ of points (k, k') in IP that are reachable via the vectors $e_0, ..., e_i$ and satisfying $(k + k') \equiv id \bmod n_p$:

5.1: For $h = 0, ..., n_p - 1$, set $Index[h] \leftarrow 0$.
5.2: Set $l \leftarrow id + 1$, and while ($l \leq n_i$) do:
5.2.1: If $v_0 + le_i = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$, and send (k, k') to processor id' in the array Aux at location $Index[id']$.
5.2.2: Set $l \leftarrow l + n_p$ and $Index[id'] \leftarrow Index[id'] + 1$.
5.3: For each $u \in A_{i-1}(id)$ and $l = 0, ..., n_i$, if $u + le_i = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$, send (k, k') to processor id' in the array Aux at location $Index[id']$, and set $Index[id'] \leftarrow Index[id'] + 1$.
5.4: $bsp\_sync()$.
5.5: Initialise all flags in array $Flag$ to $PASS$. For each $u = (k, k') \in Aux$ do:
5.5.1: Find the smallest integer $j \geq y_{min}$ such that $k + j \equiv id \bmod n_p$.
5.5.2: Set $h \leftarrow (k' - j)/n_p$; if $Flag[k - x_{min}][h] \neq FAIL$, add u to $A_i(id)$ and set $Flag[k - x_{min}][h]$ to $FAIL$.

Step 6: Compute the last set $A_{m-1}(id)$:

6.1: For $h = 0, ..., n_p - 1$, set $Index[h] \leftarrow 0$.
6.2: For each $u \in A_{m-2}(id)$ and $l = 0, ..., n_{m-1} - 1$: if $u + le_{m-1} = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$ and send (k, k') to id' in the array Aux at location $Index[id']$, and set $Index[id'] \leftarrow Index[id'] + 1$.

Step 7: $bsp\_sync()$.

Step 8: Remove the repetitions in Aux as described in 5.5 above and store the unique points in $A_{m-1}(id)$.

Step 9: Let $v_0 = (k, k')$. If $id \equiv (k + k') \bmod n_p$ and if $v_0 \in A_{m-1}(id)$, set $Result \leftarrow Decomposable$ and broadcast $Result$ to all processors.

Step 10: $bsp\_sync()$.

Step 11: Return "Result".

Theorem 8.2.1 *Let y_{min}, y_{max}, x_{min} and x_{max} denote the respective smallest and largest y coordinates, and the smallest and largest x coordinates, of vertices of P, and assume as above that $n_p < 2\gamma$, where $\gamma = \max(x_{max} - x_{min}, y_{max} - y_{min})$. Then the above algorithm works correctly as specified, has a BSP cost equal to*

$$\frac{O(\gamma^2 mN)}{n_p} + \left(\frac{O(\gamma^2 mN)}{n_p} + O(\gamma + n_p)\right) g(n_p) + (m+2) \cdot \ell(n_p)$$

flops, and requires $\frac{O(\gamma^2 N)}{n_p}$ bits of storage per processor, assuming that the maximum of absolute values of all coordinates of vertices of P fits in a machine word. Here, m denotes the number of edges in P, N denotes the maximum number of integral points along any edge in P, and γ^2 denotes an upper bound on the total number of integral points in P.

Proof: The algorithm is executed by all processors which implement their own copy of the ensuing instructions. In the first step, all processors define a global array that will be used to store the x coordinates of points of intersection between horizontal lines $y = y_{min}, ..., y_{max}$ and P. We assume that there are always two points to be stored even if they were identical, in which case they designate a vertex. The amount of memory required to keep this global information about IP is $2(y_{max} - y_{min} + 1)$ bits. Step 2 is done in parallel where processors perform almost an equal number of intersections between horizontal lines and the polygon. Since each such intersection consists of at most $m(y_{min} - y_{max} + 1)$ floating point operations followed by a communication of two integers to n_p processors, the computation and communication costs for this step are

$$\lceil m(y_{min} - y_{max} + 1)/n_p \rceil + 2n_p \lceil (y_{min} - y_{max} + 1)/n_p \rceil \, g(n_p)$$

flops. Step 3 is a synchronisation point needed for all processors to update the values of the above points of intersection and hence for each to have a complete description of IP. Note that $y_{max} - y_{min} + 1 \leq \gamma + 1$ and hence Steps 1-3 require

$$O(m\gamma)/n_p + O(\gamma)g(n_p) + \ell(n_p)$$

flops and $O(\gamma)$ bits of storage.

In the remaining steps, each processor id computes its own subset of A_i, for $i = 0, ..., m-1$, denoted by $A_i(id)$. According to our data distribution in Lemma 8.2.4, each processor will be in charge of $O(\gamma^2)/n_p$ points, but makes use of a private copy of an auxiliary array, Aux, which temporarily stores the points as they build up in one iterative step, even when they occur redundantly up to N times. The inner-most computations in each iteration in Steps 5 and 6 require that each processor id performs a vector sum (finding a new point $v = u + le_i$), two integer comparisons (checking for inclusion of v in IP), an integer division (determining the relevant processor id'), and a communication of two machine words representing the coordinates of the communicated point. Note that the communication is done only to the relevant processor id' in the auxiliary array Aux, and that processor id keeps an updated index on the address in $Aux(id')$ where it can communicate v to its relevant processor id'. By Lemma 8.2.1, all the computations within a single iteration of the loops over the edges of the polygon are independent, and hence, a synchronisation barrier is needed only at the end of the sequence of computations and communications detecting paths along $e_0, ..., e_i$ for $i = 0, ..., m-1$. This ensures that all

processors update the values of the newly added points in the array Aux. When this is done, each processor can then remove the repetitions in its storage as follows. For $k = x_{min}, ..., x_{max}$, processor id stores all points (k, k') such that $k + k' \equiv id \bmod n_p$, and $y_{min} \leq k' \leq y_{max}$. In removing repetitions, each point (k, k') has to have a flag associated with it, which is initially set to $PASS$ but then permanently set to $FAIL$ signalling that it has been copied from Aux to $A_i(id)$ exactly once. We have seen in Lemma 8.2.4 that this distribution allocates $O(\gamma^2)/n_p$ integral points per processor, hence the size of the array $Flag$. The flag of (k, k') has to occupy an address in $Flag$ that is dependent solely on both the order of the entries and their values in the pair (k, k'). Flags also have to occupy entries in the double array successively starting from the location $(0, 0)$. This justifies the index $k - x_{min}$, specifying the row in which the flag of (k, k') will be found, where $k - x_{min} = 0, ..., x_{max} - x_{min} \leq \gamma$. For a fixed such k, the choice of the integer j such that j is the first integer greater than or equal to y_{min} (and of course less than or equal to y_{max} since $n_p \leq y_{max} - y_{min} + 1$ and $y_{min} \leq j \leq y_{min} + n_p - 1$) and satisfying $k + j \equiv id \bmod n_p$ implies that any k' such that $(k, k') \in A_i(id)$ has to satisfy

$$k' = j + hn_p, \qquad \text{for some integer } h = 0, ..., \left\lfloor \frac{y_{max} - j}{n_p} \right\rfloor$$

(note that $j \leq k' \leq y_{max}$ for (k, k') to belong to IP). For $h = (k' - j)/n_p$, we have that h uniquely identifies k' such that $k + k' \equiv id \bmod n_p$ and that the flags of (k, k') for a fixed k and increasing values of k' occupy columns $h = 0, ..., \lfloor (y_{max} - j)/n_p \rfloor$ of array $Flag$. Since $j \geq y_{min}$, this ensures that the number of columns per row of the array $Flag$ is at most $O(\gamma)/n_p$, and hence the total size of the array does not exceed $\gamma \cdot O(\gamma)/n_p = O(\gamma^2)/n_p$.

The BSP cost of Steps 5-8 can then be computed as follows. The loop over all edges of the polygon iterates m times. In each iteration, every processor performs at most $O(\gamma^2 N)/n_p$ vector operations, communicates a set of possibly redundant $O(\gamma^2 N)/n_p$ points, each to one processor only, removes the repetitions in its auxiliary array using at most $O(\gamma^2 N)/n_p$ integer operations, and synchronises with other processors once. The BSP cost of Steps 5-8 is then

$$\frac{O(\gamma^2 mN)}{n_p} + \frac{O(\gamma^2 mN)}{n_p} g(n_p) + m \cdot \ell(n_p)$$

flops.

By the end of all computations, the processors assume the result to be "Indecomposable". The processor which is assigned the pivot v_0 checks whether v_0 is in its copy of $A_{m-1}(id)$. Only then does it broadcast to all other processors the result "Decomposable" (say in the form of a bit word $FAIl = 0$ or $PASS = 1$). A synchronisation barrier is finally met to update the ultimate result. The total cost of this superstep is $n_p g(n_p) + \ell(n_p)$. Summing up the above sub-costs, the final estimate in the theorem is established. Since $n_p < 2\gamma$, the spatial complexity is dominated by the space required by the array Aux, which is $O(\gamma^2 N)/n_p$ bits, assuming that the maximum over absolute values of all coordinates of vertices of P fits in a machine word.

Corollary 8.2.1 *Given a bivariate polynomial f over $\mathbb{F}$ of degree q with no non-constant monomial factors and with c nonzero terms, absolute irreducibility testing can be performed in parallel using*

$$\frac{O(cq^3)}{n_p} + \left(\frac{O(cq^3)}{n_p} + O(q + n_p) \right) g(n_p) + O(c)\ell(n_p)$$

flops and $\frac{O(q^2)}{n_p}$ *bits of storage, assuming* q *fits in a machine word, and* $1 \leq n_p = O(q)$.

Proof: Let y_{min}, y_{max}, x_{min}, x_{max} and γ be as defined above. Since $y_{max} - y_{min} \leq q$ and $x_{max} - x_{min} \leq q$, Newt(f) can be embedded in a $q \times q$ square whose lowest leftmost vertex is the origin of coordinates. Let u denote some unit of length on the horizontal or vertical axes. Then, the longest edge of Newt(f) will have length that is bounded by $\sqrt{2}qu$ representing the length of the diagonal of the square, and hence will have $O(q)$ integral points so that $N = O(q)$. Moreover, since $\gamma = \max(y_{max} - y_{min}, x_{max} - x_{min})$, we have $\gamma \leq q$. We also know that the number of edges of Newt(f) is bounded by the number of terms c of f. Summarising, the input to Algorithm 8.2.1 will then satisfy $N = O(q)$, $\gamma^2 = O(q^2)$ and $m = O(c)$, and by Theorem 8.2.1 above, the result follows immediately.

Let T_s denote the sequential run time in flops of Algorithm 3.3.1 and T_{n_p} denote the parallel run time in flops using n_p processors of the parallel version 8.2.1. We have seen that $T_s = O(\gamma^2 mN)$ where γ^2 represents an upper bound on the number of integral points in P. Let E_{n_p} denote the absolute efficiency (see [86]) defined by $E_{n_p} = T_s/(n_p T_{n_p})$, measuring the scalability of the above parallel algorithm. The following result establishes the conditions under which our BSP algorithm can achieve linear speed-up, where efficiency approaches 1.

Corollary 8.2.2 *Then Algorithm 8.2.1 for testing indecomposability of a convex polygon* P *in parallel achieves efficiency* $E_p \geq 1/2$ *under the conditions*

1. $g(n_p) = O(1)$, *(see note below)*
2. $n_p < \gamma mN$,
3. $n_p < \gamma \left(\frac{\gamma mN}{n_p} - 1 \right)$,
4. $n_p \ell(n_p) = O(\gamma^2 mN)$.

Proof: Recall from Theorem 8.2.1 that

$$T_{n_p} = T_s/n_p + \left(O(\gamma^2 mN)/n_p + O(\gamma + n_p) \right) g(n_p) + (m+2)\ell(n_p).$$

For $g(n_p) = O(1)$ we have

$$T_{n_p} = T_s/n_p + O(\gamma + n_p) + (m+2)\ell(n_p).$$

If $n_p < \gamma mN$, $\frac{\gamma mN}{n_p} - 1 > 0$ and hence we can require $n_p < \gamma \left(\frac{\gamma mN}{n_p} - 1 \right)$ or $\gamma + n_p < \gamma^2 mN/n_p$, so that

$$T_{n_p} = T_s/n_p + (m+2)\ell(n_p).$$

If we also have $n_p \ell(n_p) = O(\gamma^2 mN)$, then $(m+2)\ell(n_p) = O(\gamma^2 mN)/n_p$ and

$$T_{n_p} = \frac{T_s}{n_p}(1 + O(1))$$

from which one deduces that

$$E_p = \frac{T_s}{n_p T_{n_p}} \geq 1/2.$$

Note 1:
Although the first condition in the corollary above poses a heavy requirement on the communication parameter, we note that the experimental results obtained later on benefit mainly from the low synchronisation cost, which is dependent on the number of edges of the input polygon. In the case of sparse polynomials, this number is usually very small, hence the speed-up we report in our experiments. Furthermore, we view our communication cost as an affordable requirement in practice, especially that the parallel algorithm promises absolute irreducibility testing of polynomials with significantly higher degrees than can be tested using a sequential version of the polygon decomposability testing algorithm.

8.3 Parallel multivariate absolute irreducibility testing

As discussed earlier, multivariate polynomial absolute irreducibility testing through polytopes can be performed only heuristically with varying rates of success. For a multivariate polynomial of n variables and degree bound equal to d on each variable, the empirical results in [47] reflect a very high probability of success for polynomials whose number of terms is $O(nd)$. For polynomials whose number of terms exceeds this bound, the probability of success can be increased by loosening the bound on the absolute values of the randomly chosen matrices, or the bound on the number of projections that one can try before the algorithm outputs success. The first strategy obviously comes at the expense of a larger running time, since it implies increasing the sizes of the shadow polygons and hence the run time of the polygon indecomposability test. The second option also increases the run time of the algorithm, simply because it involves many more shadow polygons (of roughly the same size) to be tested for indecomposability. Unlike the case for bivariate absolute irreducibility testing, improving the sequential algorithm not only involves extending the range of success for higher degree polynomials, but also investigating how parallel techniques can improve upon the performance when any of the two above strategies is invoked, with the aim of increasing the chances of success even for polynomials that are denser than those for which the randomised algorithm is generally successful. A parallel approach to multivariate absolute irreducibility testing will naturally depend on the parallel bivariate case as a sub-problem; however, we shall emphasise the role that a parallel environment can have in the interplay between the number of projections and their "size", as one manipulates the two parameters to improve the success rate.

8.3.1 A BSP algorithm for testing polytope indecomposability

In the following, let n_p denote the maximum number of processors that can be made available, and let p denote the number of processors that are actually invoked. We then have $p \leq n_p$. As sizes of shadow polygons may change (according to the matrix bound one chooses in Step 2 of Algorithm 3.3.2), so can their numbers, and accordingly we identify the two parameters

governing the behaviour of our parallel algorithm: First, the number p_r of processors that we assign for testing indecomposability of one common shadow polygon using the parallel bivariate version, and the total number j of shadow polygons to which one applies Algorithm 8.2.1 simultaneously in parallel. Since $p_r \leq p$, one will have $j \geq 1$ blocks of p_r processors, each block performing a parallel polygon indecomposability test, resulting in a "doubly" parallel scheme aimed at improving the sequential performance at the two levels of size and number of projections. Before presenting our algorithm, we shall need a few more notations. Let p_{min} denote the minimum number of processors required to test indecomposability of one shadow polygon using the parallel algorithm in 8.2.1. For every shadow polygon P_i, let $y^{(i)}_{max}$, $y^{(i)}_{min}$, $x^{(i)}_{max}$, and $y^{(i)}_{min}$ denote respectively the largest and smallest y coordinates, and the largest and smallest x coordinates, among all lattice points in P_i. Let $\gamma_i = \max(y^{(i)}_{max} - y^{(i)}_{min}, x^{(i)}_{max} - x^{(i)}_{min})$ and define $\gamma_{max} = \max(\gamma_i)$, where the maximum is taken over all possible shadow polygons P_i. The number p_{min} thus corresponds to the minimum number of processors required to store a number of integral points that is bounded by γ^2_{max} using the data distribution in Lemma 8.2.4. Note that, given a polynomial f with total degree at most nd, the projections of the support vectors of f into the plane using a matrix whose entries are bounded by b have coordinates that are equal to at least $-2nbd$ and at most $2nbd$, and we consequently have $\gamma_{max} = O(nbd)$.

Let n_p, p and p_r be as defined above, and assume in the following that more than p_{min} processors can be allowed to test one shadow polygon. In particular, let $u \in \{0, ..., n_p - p_{min}\}$ be such that $p_r = p_{min} + u$, and let $j \in \{1, ..., \left\lfloor \frac{n_p}{p_{min}+u} \right\rfloor\}$ be such that $jp_r = p$. With this structure we can have blocks of j shadow polygons, each of which is tested for indecomposability by Algorithm 8.2.1 using $j(p_{min} + u) = jp_r = p \leq n_p$ processors. Taking the maximum over all shadow polygons, let N denote the maximum number of integral points along any edge, E denote the maximum number of edges, and γ_{max} be as defined above, denoting the maximum number of integral points belonging to any shadow polygon.

Algorithm 8.3.1 *(Parallel Polytope Indecomposability Test)*
Input: Let $f \in \mathbb{F}[X_1, ..., X_n]$, with $n > 2$, be a polynomial with c terms and no non-constant monomial factors, and let S_f denote the set of exponent vectors of nonzero terms of f.
Output: Absolutely irreducible or Failure, where the latter case means that indecomposability of $conv(S_f)$ (and hence absolute irreducibility of f) is not decided.

Step 1: Re-arrange the points in S_f as an $n \times c$ matrix S. Choose positive integers b and e. Let $M(b)$ denote the set of all $2 \times n$ matrices with integer coefficients bounded in absolute value by b.

Step 2: Determine the minimum number of processors p_{min} necessary to test indecomposability of a shadow polygon of size $O((nbd)^2)$.

Step 3: Choose a parameter $u \in \{0, ..., n_p - p_{min}\}$ such that any shadow polygon is tackled by $p_{min} + u$ processors. Set $p_r = p_{min} + u$.

Step 4: Choose a parameter $j \in \{1, ..., \left\lfloor \frac{n_p}{p_r} \right\rfloor\}$ such that $jp_r \leq n_p$.

Step 5: Invoke $p = jp_r$ processors to operate in parallel. If $e \bmod j \neq 0$, set $e \leftarrow e + j - (e \bmod j)$.

Step 6: Define an auxiliary array of integer vectors, Aux, of size $O(\gamma^2_{max}N)/p_r$, an index array of integers, Index, of size p_r, and a flag double array of integers, Flag, of size $O(\gamma^2_{max})/p_r$.

Step 7: Processors are divided into j blocks according to indices as follows:

$$block_0 = \{id | id = 0, ..., p_r - 1\},$$
$$block_1 = \{id | id = p_r, ..., 2p_r - 1\},$$
$$....$$
$$block_{j-1} = \{id | id = (j-1)p_r + jp_r - 1\}.$$

All processors in block w, for some $w = 0, ..., j-1$, perform Steps 8-18 repeatedly up to e/j times:

Step 8: Select a common matrix Mat_w uniformly at random from $M(b)$ and compute the set of points in $\mathbb{R}^2$ defined by $Mat_w(S) := \{Mat_w.s | s \in S\}$.

Step 9: Compute the convex hull and the edge sequence $\{n_i e_i\}_{0 \le i \le m-1}$ of $Mat_w(S)$. Check that each vertex of $conv(Mat_w(S))$ has only one pre-image in S under the projection Mat_w. If this condition is not met, all processors in block w return to Step 8.

Step 10: bsp_sync();

Step 11: Compute a description of the set IP of all the integral points in $Mat_w(S)$ as described in Steps 1 and 2 of Algorithm 8.2.1. Let $A_i^{(w)}(id)$ denote the set of points in $conv(Mat_w(S))$ that are reachable via $e_0, ..., e_i$ and that are assigned to processor id in block w. Set $A_i^{(w)} = \emptyset$, for $i = -1, ..., m-1$, and $Result^{(w)} \leftarrow Indecomposable$. Let id denote the processor's identification number.

Step 12: For $i = 0, ..., E-2$ do:

12.1: If $i \le m-2$, compute the set $A_i^{(w)}(id)$ of points (k, k') in IP that are reachable via the vectors $e_0, ..., e_i$ and satisfying $(k + k') \equiv id \bmod p_r$:

12.1.1: For $h = w.p_r, ..., (w+1).p_r - 1$, set $Index[h] \leftarrow 0$.

12.1.2: Set $l \leftarrow id - (w.p_r) + 1$, and while $(l \le n_i)$ do:

. If $v_0 + le_i = (k, k') \in IP$, set
$id' \leftarrow [(k + k') \bmod p_r] + w.p_r$, and send (k, k')
to processor id' in the array Aux at location $Index[id']$.
. Set $l \leftarrow l + p_r$ and $Index[id'] \leftarrow Index[id'] + 1$.

12.1.3: For each $u \in A_{i-1}^{(w)}(id)$ and $l = 0, ..., n_i$, if $u + le_i = (k, k') \in IP$, set $id' \leftarrow [(k + k') \bmod p_r] + w.p_r$, send (k, k') to processor id' in the array Aux at location $Index[id']$ and set $Index[id'] \leftarrow Index[id'] + 1$.

12.2: bsp_sync();

12.3: Initialise all flags in array $Flag$ to $PASS$. For each $u = (k, k') \in Aux$ do:
12.3.1: Find the smallest integer $j \ge y_{min}$ such that $k + j \equiv id \bmod p_r$.
12.3.2: Set $h \leftarrow (k' - j)/p_r$; if $Flag[k - x_{min}][h] \ne FAIL$, add u to $A_i^{(w)}(id)$ and set $Flag[k - x_{min}][h]$ to $FAIL$.

Step 13: Compute the last set $A_{m-1}^{(w)}(id)$:

13.1: For $h = wp_r, ..., (w+1)p_r - 1$, set $Index[h] \leftarrow 0$.

13.2: For each $u \in A_{m-2}^{(w)}(id)$ and $l = 0, ..., n_{m-1} - 1$: if $u + le_{m-1} = (k, k') \in IP$, set

$id' \leftarrow [(k+k') \bmod p_r] + w.p_r$, *send* (k,k') *to* id' *in the array Aux at location* $Index[id']$, *and set* $Index[id'] \leftarrow Index[id'] + 1$.

Step 14: bsp_sync().

Step 15: Remove the repetitions in Aux as described in 12.3 above and store the unique points in $A_{m-1}^{(w)}(id)$.

Step 16: Let $v_0 = (k,k')$. *If* $id = [(k+k') \bmod p_r] + w.p_r$ *and if* $v_0 \in A_{m-1}^{(w)}(id)$, *set Result* $\leftarrow$ *Decomposable and broadcast Result to all processors.*

Step 17: bsp_sync().

Step 18: If this polygon is integrally indecomposable, output "Absolutely irreducible" and halt. Else, all processors return to Step 8.

Step 19: All processors output "Failure".

Theorem 8.3.1 *Let* $f \in \mathbb{F}[X_1, ..., X_n]$, *with* $n > 2$, *be a polynomial with* c *terms and no non-constant monomial factors. Let* d *denote the upper bound on the degree in each variable of* f, *and let* b *denote the upper bound on the absolute values of integer coefficients of* $2 \times n$ *matrices representing random projections. Let* p_{min} *denote the minimum number of processors needed to store* $O((nbd)^2)$ *integral points using the distribution in Lemma 8.2.4. Let* $k \in \{0, ..., n_p - p_{min}\}$ *and* $j \in \{1, ..., \left\lfloor \frac{n_p}{p_{min}+k} \right\rfloor\}$ *such that* $p = j(p_{min}+k)$ *processors are operating in parallel and* $p \leq n_p$, $n_p = O(nbd)$. *Then Algorithm 8.3.1 can decide absolute irreducibility of* f *correctly or else produce "failure" using at most*

$$T_p = \frac{e}{j}\left(O(c^2+cn) + \frac{O(\gamma_{max}^2 EN)}{p_r}\right) + \frac{e}{j}\left(\frac{O(\gamma_{max}^2 EN)}{p_r} + O(p+\gamma_{max})\right) g(p) + \frac{e}{j} O(E)\ell(p)$$

flops, and requires $\frac{O(\gamma_{max}^2 N)}{p_r}$ *bits of storage per processor, assuming that nbd fits in a machine word. Here,* E *denotes the maximum over all shadow polygons of total number of edges,* N *denotes the maximum over all shadow polygons of number of integral points along any edge, and* γ_{max}^2 *denotes an upper bound on the total number of interior integral points belonging to any shadow polygon.*

Proof: The first four steps of the algorithm are performed sequentially, whereby a matrix bound b is chosen, which determines p_{min} according to the data distribution in Lemma 8.2.4. The two parameters u and j are also chosen such that $p_{min} + u = p_r \leq n_p$ processors can be assigned for any shadow polygon, and j shadow polygons can be tested simultaneously. The algorithm then invokes $p = jp_r \leq n_p$ processors to operate in parallel. The minor modification to the value of the number of projections e ensures that j divides e, so that all processors are made to enter the loop starting at Step 8. The reason we enforce such full participation of processors is that synchronisation barriers will be met throughout that particular loop, which causes a run-time error if any of these supersteps is not met by all processors. In Step 6, all arrays that will be used for removing repetitions in the upcoming computations are declared. Note that the amount of memory per array is dependent on the number of processors p_r operating in one block rather than the total number of processors. This is because only p_r processors will be allowed to share the work in the parallel polygon indecomposability testing. In Step 7, all p processors

re-cluster into j blocks as determined by their identification number. This re-grouping ensures processors within one block $w = 0, ..., j-1$ compute the same random projection $Mat_w(S)$, and perform the rest of the Steps 8-18 using this common input.

The loop starting at Step 8 iterates at most e/j times. We analyse each step in the iteration as follows. Step 8 is only a computation whereby all processors perform c matrix vector multiplications using their assigned matrix Mat_w. Each such multiplication requires $2n$ multiplications and $2(n-1)$ additions of integers bound in absolute value by nbd, and so Step 8 has a BSP cost of $O(cn)$ flops. In Step 9, all processors in block w compute the edge sequence of the shadow polygon $conv(Mat_w(S))$. This is only a computation step with BSP cost $O(c^2)$ flops. A synchronisation barrier is met at Step 10 to ensure that processors in a particular block w which have found a successful projection wait for others in different blocks still searching for a good candidate projection. Without this barrier, one risks having some but not all processors entering the loop of the parallel polygon indecomposability testing phase (which in turn contains a synchronisation barrier that should be met by all p processors). In Steps 11-17, each processor joins the others in its block to test indecomposability of their common shadow polygon. In Step 12, we enforce an upper bound of $E-2$ rather than $m-2$, since E is a global maximum of the number of edges belonging to any shadow polygon, whereas m is a private copy representing the number of edges of $conv(Mat_w(S))$. This is again to ensure that all processors enter the loop within which a synchronisation barrier is to be met in Step 12.2. However, the relevant computations and communications are performed only when the processors in block w can do so (as indicated by the condition $i \leq m-2$ in Step 12.1). The vector computations and the repetition checkings in Steps 12.1, 12.3 and 13 are similar to those in the parallel polygon indecomposability testing algorithm. However, we note the following essential differences. Note that any processor in the above scheme has two labels attached to it, one describing its identification number $id = 0, ..., p-1$, and another describing its index ind within its block, for $ind = 0, ..., p_r - 1$. Moreover, a processor id operates within block $w = \lfloor id/p_r \rfloor$, and has index $ind \equiv id \bmod p_r$ in that block. Conversely, a processor with index ind in block w has id equal to $w.p_r + ind$. The data allocation in the present algorithm should assign arbitrary points (k, k') of the polygon $conv(Mat_w(S))$ only to p_r processors. Thus, one checks for the value of $(k + k')$ modulo p_r rather than p, the total number of processors in action. But this gives the index of the processor to which (k, k') should be allocated. The actual id can then be retrieved as $[(k + k') \bmod p_r] + w.p_r$. As seen previously, the BSP cost required by Steps 11-15 is at most

$$\frac{O(\gamma_{max}^2 EN)}{p_r}) + \left(\frac{O(\gamma_{max}^2 EN)}{p_r} + O(\gamma_{max}) \right) g(p) + (E+2)\ell(p)$$

flops and $O(\gamma_{max}^2 N)/p_r$ bits of storage. Note that we make explicit the dependence of g and ℓ on p, since their cost depends on the total number of processors invoked despite the fact that the computations and communications are shared between blocks of p_r processors only.

In Step 16, all p processors resume contact to be able to know which of the w shadow polygons have been shown indecomposable. The processor in block w which is in charge of the pivot v_0 of $Conv(Mat_w(S))$ decides whether the polygon is indecomposable, and if so, signals to all processors in its block and other blocks to halt the algorithm. Else, all processors repeat Steps 8-18 choosing a different projection Mat_w. This involves p communications of a boolean representing "Indecomposable", and brings the total cost of the entire algorithm to

$$T_p = \frac{e}{j} \left(O(c^2 + cn) + \frac{O(\gamma_{max}^2 EN)}{p_r} \right) + \frac{e}{j} \left(\frac{O(\gamma_{max}^2 EN)}{p_r} + O(p + \gamma_{max}) \right) g(p) + \frac{e}{j} O(E)\ell(p)$$

flops. The memory requirement is dominated by $O(\gamma_{max}^2 N)/p_r$ bits needed per processor to store its subset of integral points belonging to any shadow polygon.

Corollary 8.3.1 *Algorithm 8.3.1 for absolute irreducibility testing of multivariate polynomials achieves efficiency*

$$E_p \geq \frac{1}{p_r}$$

under the conditions

1. $g(p) = O(1)$,
2. $p < c^2 + cn + (\gamma_{max}^2 EN/p_r) - \gamma_{max}$,
3. $\ell(p) < \frac{c^2+cn}{E} + \frac{\gamma_{max}^2 N}{p_r}$.

Proof: Recall that the sequential time for testing absolute irreducibility of a random multivariate polynomial is given by

$$T_s = eO(c^2 + cn + \gamma_{max}^2 EN)$$

bit operations, assuming *nbd* fits in a machine word. Rewrite this as

$$T_s = T_1 + T_2$$

where $T_1 = eO(c^2 + cn)$ denotes the cost of that part of the algorithm in which the projected points and their convex hull are computed, and $T_2 = eO(\gamma_{max}^2 EN)$ denotes the cost of testing integral indecomposability of the shadow polygons. We also have

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(\gamma_{max}^2 EN)}{p_r}\right) + \frac{e}{j}\left(\frac{O(\gamma_{max}^2 EN)}{p_r} + O(p + \gamma_{max})\right) g(p) + \frac{e}{j}O(E)\ell(p)$$

flops. For $g(p) = O(1)$,

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(\gamma_{max}^2 EN)}{p_r}\right) + \frac{e}{j}O(p + \gamma_{max}) + \frac{e}{j}O(E)\ell(p).$$

By the condition $p < c^2 + cn + (\gamma_{max}^2 EN/p_r) - \gamma_{max}$, $p + \gamma_{max} = O(c^2 + cn) + O(\gamma_{max}^2 EN)/p_r$ and so

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(\gamma_{max}^2 EN)}{p_r}\right) + \frac{e}{j}O(E)\ell(p),$$

and if $\ell(p) < \frac{c^2+cn}{E} + \frac{\gamma_{max}^2 N}{p_r}$,

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(\gamma_{max}^2 EN)}{p_r}\right)$$

or

$$T_p = \frac{1}{j}\left(T_1 + \frac{T_2}{p_r}\right)$$

for T_1 and T_2 as defined above. We then have

$$\begin{aligned} pT_p &= \frac{p}{j}\left(T_1 + \frac{T_2}{p_r}\right) \\ &= p_r T_1 + T_2 \\ &\leq p_r(T_1 + T_2) \\ &= p_r T_s \end{aligned}$$

from which one concludes that

$$\frac{T_s}{pT_p} \geq \frac{1}{p_r}.$$

The above discussion investigates the parallel efficiency for an increasing number of processors. However, the implications play an important role in the choice we have to make of the parameters u and j. In particular, the lower bound on E_p can be improved for decreasing values of p_r, which indicates that the best realistic performance is achieved by choosing $u = 0$, so that $p_r = p_{min}$. Since one cannot invoke Algorithm 8.3.1 without less than p_{min} processors per shadow polygon, we expect this to be the best case scenario describing the parallel scalability of the algorithm.

8.4 Implementation and Run Times

All programs were written in C and extended using the standard BSP library [66, 67]. The work was carried out at the Oxford University Supercomputing Centre (OSC) using the Oswell machine. In practice, we had access to 16 processors only.

In the following, n denotes the number of variables in the input polynomial f, D denotes its total degree, d denotes the upper bound on the degrees in each of its variables, and c denotes the number of its terms. Also, E and N denote the number of edges and the maximum number of integral points along any edge of Newt(f) if f is bivariate. If f is multivariate, E and N denote the maximum over the number of edges and the maximum number of integral points along any edge over all shadow polygons of Newt(f). S denotes the number of cases (out of 100) in which Newt(f) is integrally indecomposable. In the case that $n > 2$, MB denotes the matrix upper bound on absolute values of random coefficients of the projections, PB denotes the upper bound on the number of projections per polytope, AP denotes the average number of projections required to show that the input polynomial is absolutely irreducible, and p_r denotes the number of processors allocated per shadow polygon in the parallel multivariate algorithm. T_1 denotes the sequential running time in seconds, and T_p, for $p > 1$, denotes the parallel running time in seconds using p processors, to show absolute irreducibility successfully for one case that uses about the average number of projections. An empty column location appearing before the first reported running time T_p indicates that there is not enough memory using less than p processors to tackle the input polygon or shadow polygons of the input polytopes. An empty

column location appearing after the last reported running time T_p indicates that there are no more processors available in the system for our use. The absolute efficiencies E_p are shown in parentheses below their corresponding parallel times. Note that when the algorithm cannot be run using one processor for memory constraints, we are contended with calculating absolute efficiency using p processors as $p'T_{p'}/pT_p$, where $T_{p'}$ is the first reported parallel running time.

The input to the two parallel algorithms is generated as follows. A choice is first made on the parameters n, D or d, c, E and N. A hundred random polynomials satisfying the above parameters are then chosen: In the bivariate case, those polynomials should also satisfy the conditions governing their Newton polytopes (in terms of the number of edges E and the maximum number N of integral points appearing along any of their edges). In the multivariate case, the projections chosen for these 100 polynomials should produce shadow polygons satisfying the parameters E and N. In the process of generating the random polynomials, we exclude all cases where the corresponding input polygons or shadow polygons of the corresponding input polytopes have parameters E and N that do not satisfy the imposed restrictions.

In Table 8.1 we examine relatively small degree bivariate polynomials with 300 terms and whose Newton polytopes have no more than 6 edges, and we study the effect of variations in the average number N of integral points along any edge of the polygon. The performance generally has good efficiency up to 4 processors, but it can be improved for more processors by increasing the degree (and hence the total number of integral points in the polygon) or increasing values of N for a fixed degree. Specifically, and according to Corollary 8.2.2 and the values in Table 5.2, we see that the first three conditions are easily satisfied. For small degree polynomials whose Newton polytopes have a small value for the parameter N, the fourth condition might not hold as we increase the number of processors. In this case efficiency can be improved by increasing the maximum number of integral points along any edge of Newt(f). The rate of success is generally high, since as indicated previously in [47], one expects the algorithm to perform well for sparse polynomials whose number of terms is $O(nd)$. However, we note that the rate of success decreases with increasing values of N.

In Table 8.2 we examine larger degree bivariate polynomials with varying numbers of terms. Mostly, efficiency improves significantly for larger degree polynomials. As noted above, and for a fixed degree polynomial whose Newton polytope has a fixed number of edges, the run-time increases and efficiency improves when N increases for a fixed D or when D increases. Unlike the examples in Table 8.1 though, varying the number of terms c indicates that the rate of success decreases for increasing ratios N/c rather than simply for increasing values of N. Fixing the degree, the number of terms, and the parameter N, we also note an increase in the run-time and improvement in efficiency when the number of edges increases, which is to be expected from Corollary 8.2.2.

In Table 8.3 we study the performance of the parallel algorithm in the multivariate case and for small degree polynomials. The number of edges belonging to the shadow polygons is fixed. For all unstarred rows, the number of processors in each block is set to be the minimum required. Efficiency improves for a fixed degree as the number of terms increases. This relates to conditions 2 and 3 of Corollary 8.3.1. For a fixed degree and a fixed number of terms, efficiency improves when the matrix bound increases, since this implies larger shadow polygons and hence larger values for γ_{max} and N in conditions 2 and 3 of Corollary 8.3.1. In the starred rows we compare the performance of the algorithm when only p_r is increased beyond the minimum required, but all other parameters remain fixed. As predicted at the end of Section 8.3, and for a fixed total number of processors p, efficiency is better maintained when $p_r = p_{min}$. As earlier

noted in [47], the success rate decreases with increasing numbers of terms, and can be improved by either increasing the projection bound or the matrix bound.

In Table 8.4 we examine the performance for large degree trivariate polynomials. Here, the number of terms is fixed, and is significantly less than the total degree of the input. Also, E and N are fixed for all shadow polygons, and so are the matrix bound and projection bound. We note a much better parallel performance than in the case of small degree trivariate polynomials in Table 8.3. This also improves upon increasing degrees (as indicated by conditions 2 and 3 of Corollary 8.3.1). The algorithm has never failed for examples using only one projection per case and a very small matrix bound. This emphasises the expected high success rate of the algorithm for sparse polynomials [47].

Finally, in Table 8.5, we examine the performance for multivariate polynomials all with the same small bound on the degree in each of their variables. Also fixed are the number of terms, E and N of the shadow polygons, and the matrix bound and projection bound. Here, p_r is set to be the minimum number of processors required to tackle a shadow polygon. We increase the number of blocks to be tested in parallel by increasing the number of processors available. Efficiency is very good in all cases, even for polynomials whose total degree is less than others. This is to be expected since the sizes of shadow polygons are relatively large (conditions 2 and 3 of Corollary 8.3.1). The success rate is also high for these sparse polynomials, and so are the number of projections needed to produce a successful experiment.

Recall that:

- n = the number of variables in the input polynomial f
- D = the total degree of f
- d = the upper bound on the degrees in each of the variables in f
- c = the number of terms of f
- E = the number of edges of Newt(f) if f is bivariate, or else the maximum over the number of edges over all shadow polygons of Newt(f)
- N = the maximum number of integral points along any edge of Newt(f) if f is bivariate, or else the maximum number of integral points along any edge over all shadow polygons of Newt(f)
- S = the number of cases (out of 100) in which Newt(f) is integrally indecomposable
- MB = the matrix upper bound on absolute values of random coefficients of the projections
- PB = the upper bound on the number of projections per polytope
- AP = the average number of projections required to show that f is absolutely irreducible
- p_r = the number of processors allocated per shadow polygon in the parallel multivariate algorithm
- T_p = the parallel running time in seconds using p processors, to show absolute irreducibility successfully for one case that uses about the average number of projections

Table 8.1: $n = 2$, $c = 300$, $E = 6$

Input							
D	N	S	T_1	T_2	T_4	T_8	T_{16}
1500	10	100	11	7 (0.7)	9 (0.3)	14 (0.1)	15 (0.1)
1500	50	100	68	42 (0.8)	24 (0.7)	16 (0.5)	10 (0.4)
1500	100	97	83	46 (0.9)	26 (0.8)	17 (0.6)	10 (0.5)
1500	900	89	828	427 (1)	216 (1)	115 (0.9)	52 (1)
2500	10	100	35	20 (0.9)	12 (0.7)	7 (0.6)	4 (0.5)
2500	50	100	185	94 (1)	58 (0.8)	37 (0.8)	19 (0.6)
2500	100	100	240	122 (0.9)	67 (0.9)	37 (0.8)	25 (0.6)

Table 8.2: $n = 2$

Input										
D	c	E	N	S	T_1	T_2	T_4	T_8	T_{12}	T_{16}
3000	1000	7	2000	72	108	56 (1)	30 (0.9)	17 (0.8)	10 (0.9)	7 (1)
3000	1000	7	3000	6	129	65 (1)	36 (0.9)	18 (0.9)	12 (0.9)	9 (0.9)
5000	500	6	10	100	72	37 (1)	20 (0.9)	13 (0.7)	9 (0.7)	8 (0.6)
5000	500	10	10	98	168	84 (1)	47 (0.9)	24 (0.9)	18 (0.8)	15 (0.7)
5000	2000	10	4000	81	...	115	60 (1)	37 (0.8)	27 (0.7)	25 (0.6)
5000	2000	8	5000	5	...	230	117 (1)	64 (0.9)	48 (0.8)	37 (0.7)
10000	500	6	100	100	...	...	60	32 (0.9)	22 (0.9)	18 (0.8)
10000	500	10	100	100	...	...	190	95 (0.8)	79 (0.8)	50 (1)
10000	1000	8	3000	96	...	...	133	73 (0.9)	55 (0.8)	40 (0.8)
10000	3000	8	5000	86	...	...	371	206 (0.9)	155 (0.8)	116 (0.8)
20000	2000	8	1000	100	...	...	...	...	131	112 (0.9)
20000	5000	8	7000	90	...	...	...	...	177	142 (0.9)
30000	3000	8	3000	100	...	...	...	...	329	235 (1)

Table 8.3: $E = 8$, $N = 50$

Input											
n	D	c	p_r	MB	PB	S	AP	T_1	T_2	T_4	T_8	T_{16}
3	75	100	1	2	100	61	16	7	6 (0.6)	4 (0.4)	5 (0.2)	3 (0.1)
3	75	100	1	2	200	82	80	49	33 (0.7)	20 (0.6)	13 (0.5)	31 (0.1)
3	75	100	1	6	100	80	14	127	74 (0.9)	33 (1)	24 (0.7)	17 (0.5)
3*	75	100	2	6	100	80	14	...	74 (0.9)	66 (0.5)	29 (0.5)	26 (0.3)
3	75	200	1	2	100	30	18	7	5 (0.7)	5 (0.4)	6 (0.1)	4 (0.1)
3	75	200	1	2	200	45	127	232	129 (0.9)	72 (0.8)	41 (0.7)	24 (0.6)
3	75	200	1	6	100	51	8	51	25 (1)	18 (0.7)	12 (0.5)	7 (0.5)
3*	75	200	2	6	100	51	8	...	26 (1)	19 (0.7)	22 (0.3)	15 (0.2)

Table 8.4: $n = 3$, $c = 500$, $E = 8$, $N = 100$, $MB = 1$, $PB = 100$

Input							
D	S	AP	T_1	T_2	T_4	T_8	T_{16}
1500	100	1	19	10 (1)	6 (0.8)	3 (0.8)	2 (0.6)
3000	100	1	...	65	28 (1)	16 (1)	8 (1)
15000	100	1	...	...	138	74 (0.9)	39 (0.9)
21000	100	1	...	...	150	83 (0.9)	47 (0.8)
30000	100	1	...	...	...	237	118 (1)

Table 8.5: $d = 10$, $c = 500$, $E = 8$, $N = 50$, $MB = 1$, $PB = 100$

Input											
n	S	p_r	AP	T_1	$T_{1 \cdot p_r}$	$T_{2 \cdot p_r}$	$T_{3 \cdot p_r}$	$T_{4 \cdot p_r}$	$T_{5 \cdot p_r}$	$T_{6 \cdot p_r}$	$T_{8 \cdot p_r}$
500	100	2	20	...	2460	1250 (1)	828 (1)	617 (1)	517 (1)	410 (1)	313 (1)
1000	100	3	24	...	3004	1501 (1)	885 (1.1)	800 (1)	680 (1)	...	...
2000	100	5	24	...	854	427 (1)	280 (1)	...	...	...	...
3000	100	6	26	...	3224	1543 (1)	...	...	...	...	...

Note 2:
The efficiencies noted in Table 8.5 seem over-optimistic, but this can perhaps be attributed to the fact that E_p was not calculated using a sequential time but rather using the ratio $p'T_{p'}/pT_p$, for $p', p > 1$. In the cases when it is almost impossible to get a sequential time reference, this

slightly imprecise measurement of parallel performance is the best available in practice.
Note 3:
Although both our parallel algorithms have computation and communication costs growing almost together, the very efficient parallel performance can perhaps be a result of a constant factor within the computational complexity that is larger than the one in the computational estimate. In practice, this can happen when the computations within the inner-most loops produce many more lattice points that do not belong to the given polygon (or shadow polygon) than lattice points which do (and which hence have to be communicated). In this case, there is more computation that is performed and then discarded without being matched with a corresponding communication.

8.5 Conclusion

In this chapter we have revisited a fast irreducibility testing algorithm for multivariate polynomials over arbitrary fields. The algorithm works deterministically in the bivariate case but heuristically and randomly for polynomials with more than two variables. Although finding a polynomial time algorithm for multivariate polynomial irreducibility testing remains an open problem, the work in [45] gives a pseudo-polynomial time algorithm, which can be applied as a fast pre-test before any of the rigorous yet slower algorithms are invoked. Motivated by the original empirical findings in [47] which provide various ranges of applicability of the heuristic algorithm, we investigated potential parallelism with the aim of extending these ranges, both for large degree bivariate polynomials and for multivariate polynomials of all degrees. A crucial aspect of our work exploited the fact that absolute irreducibility testing can be reduced to polytope indecomposability testing in $\mathbb{R}^n$. For $n = 2$, we addressed the two important issues of load balancing and data distribution. Having set the parallel framework we provided an algorithm whose communication cost can be easily bounded by the computation, and whose synchronisation cost has a factor that is a constant multiple of the number of edges in the polygon. This immediately implies highly efficient parallel performance for sparse bivariate polynomials whose Newton polytopes have few edges. Empirical results in this case achieve overall efficiency under reasonable parametric conditions that are implied by our theoretical analysis of the algorithm, and by significantly higher degree absolute irreducibility testing up to degree 30000.

We incorporated the above for the parallel multivariate case into a doubly parallel scheme where several shadow polygons are tested in parallel by blocks of processors. This was done by identifying two parameters reflecting the size as well as the number of the shadow polygons. Conditions under which this algorithm achieves good efficiency were also studied, and reflected in the empirical results for the multivariate case. Those exhibited a good efficiency for both small degree polynomials and for high degree polynomials, where parallelism could be exploited not only for speeding up computations but also for increasing the rate of success of the algorithm. The algorithm was used to test absolute irreducibility of trivariate polynomials with degree up to 30000 and of low degree multivariate polynomials with up to 3000 variables.

Chapter 9

Conclusion

9.1 Discussion and future work

The various algorithms for polynomial factorisation over finite fields and many other related algorithms in symbolic computation have flourished under the assumption that better algorithms are those which tackle larger input sizes and achieve better running times. Thus, continuity in such a domain does not rely solely on progress in asymptotic analysis, and some other aspects of symbolic computation have to be investigated.

Over the past thirty years, the theory of polynomial factorisation over finite fields has been largely exhausted from a mathematical point of view; nevertheless, this progress has yet to be fully matched with advances concerning the machinery originally designed to foster such algorithms. A lot of challenge lies not only in coming up with faster sequential algorithms, but also in trying to increase the problem sizes for the already existing algorithms, and in comparing the various algorithms when one actually embarks on their implementation. A sequential algorithm may outperform another one simply because it requires a smaller number of operations; however, much else has to be said, for instance, when the algorithms are approached from a parallel point of view.

Among the many approaches in a computer algebra system are the following. Primarily, introducing new mathematical algorithms gains most credit for the creativity that this entails, obtaining better complexity bounds. However, there comes a time when these need to be tested in practice, before they can be branded as efficient as they are claimed to be. Heuristics can play an important role in bringing about improvements, especially in special cases like sparse or binary polynomials. The disadvantage of this approach is that heuristics still need to be proven to work before they can be generally accepted by the mathematical community. Data structures also have an important role in improving existing mathematical algorithms. These are the basic building tools whose careful manipulation can have a decisive factor in determining the efficiency of a particular algorithm. Parallelism is an active area of research and it becomes almost immediate to try and incorporate this whenever possible. Apart from the inherent interest in parallel design, this can help when either run time or memory is a problematic issue in a particular sequential algorithm. The risk involved is that, unless problems are big enough, parallel performance can in fact be worse than the sequential one.

Based on the above, we have discovered an interest in tackling such algebraic problems from a computational point of view. In this thesis, we have focused on two recent algorithms in the

field of polynomial factorisation algorithms over finite fields. The focus of interest in the earlier chapters was on Niederreiter's algorithm and its applicability over the binary field. We developed a new sparse binary linear solver based on the Gustavson data structure, aimed at avoiding elbow room and compression. The method can be easily generalised to deal with arbitrary prime fields. This was incorporated in the linear algebra part of Niederreiter's algorithm, and helped assert that the algorithm performs favourably in the case of sparse polynomials, specifically trinomials. We conjectured that the system remains considerably sparse over $\mathbb{F}_2$.

The results of the first chapter were used in the following work on a BSP model for the Göttfert algorithm. An example where the BSP model can be used in computer algebra, the parallel algorithm was used in that phase of Niederreiter's algorithm where the factors have to be extracted using a basis of the linear system solution set. We demonstrated efficient and scalable parallel performance, thanks to the algorithm's low communication and synchronisation costs. The empirical results show that the algorithm can perform favourably in comparison with previous parallel implementations of Niederreiter's algorithm.

In later chapters we helped develop the polytope factorisation method with S. Gao and A. Lauder as a novel method for bivariate factorisation. Apart from the mathematical foundations of this algorithm, the challenges in that respect have been to develop those areas of the method that helped make it practically computable and accessible for use, and demonstrate through preliminary examples over the binary field that it can compete with the standard Hensel lifting method for bivariate factorisation. As a follow up, we developed a sparse variant which confirmed the original arguments in [2] that the method can work well for sparse polynomials. The complexity of the new variant was determined using the number of terms of the input polynomial and its degree rather than the degree only, so that both the run time and memory requirements are made dependent on the sparsity factor of the input. Although it works under specific conditions governing the sparsity of the expected factors of the input, we believe that the factorisation record achieved through this method could not have been achieved using any other algorithm.

We concluded with another instance of where parallelism can be used to achieve competent results in testing absolute irreducibility of multivariate polynomials. Investigating a new method based on the use of polytopes, we exploited the geometric features of the algorithm in a BSP parallel method based on a well defined load balancing scheme and data distribution. The parallel algorithm exhibited a scalable and efficient performance, also resulting in very high absolute irreducibility testing records.

We would always be interested in questioning some of the theoretical assertions labelling one algorithm as "better" than another, determining cross over points between the versatile approaches, and many other tasks that would not be possible to achieve without the computing tools available at the hand of a computer scientist. In relation to our previous work on univariate factorisation, it would be interesting to seek theoretical arguments why the Niederreiter linear system remains considerably sparse throughout the reduction phase, or else refute our conjecture that it does. It would also be interesting to investigate the usage of the Block Lanczos method in solving the linear systems associated with either Berlekamp's or Niederreiter's algorithm, and to compare this to previous versions which make use of black box methods like the Wiedemann's method. In relation to our work on bivariate factorisation, we first need to perform ample code optimisation of the polytope method (both dense and sparse), as the software available is still in preliminary form. We are also aiming at generalising our code for the sparse polytope method to deal with arbitrary fields of prime order rather than the binary field exclusively.

We will be interested in investigating whether an average case analysis can be developed for the polytope method, to help explain why this performs well in practice despite its worst case exponential running time. We will also be interested in investigating whether any of the previous improvements to Hensel lifting, such as quadratic Hensel lifting, can be used to improve on the present complexity of the polytope method. An empirical study comparing Hensel lifting and the polytope method and determining the cross-over points between the two methods is necessary before the latter can be widely made available. Last, it would be of great use to determine whether a theoretical justification can be found at all, explaining why the probability of success of the absolute irreducibility testing via polytopes is best when the number of nonzero terms of a multivariate polynomial is bounded by a constant multiple of its total degree.

Bibliography

[1] F. Abu Salem, "A BSP parallel model of the Göttfert algorithm for polynomial factorization over $\mathbb{F}_2$", *Proc. of the Fifth International Conference on Parallel Processing and Applied Mathematics, PPAM 2003*, *Lecture Notes in Computer Science* **3019** (2004), 217–224, Springer-Verlag.

[2] F. Abu Salem, S. Gao and A. G. B. Lauder, "Factoring polynomials via polytopes", *Proc. International Symposium on Symbolic and Algebraic Computing, Spain, ISSAC 2004*, 4–11, ACM Press.

[3] F. Abu Salem, "A new sparse Gaussian elimination algorithm and the Niederreiter linear system for trinomials over $\mathbb{F}_2$", Report PRG-RR-03-18, Oxford University Computing Laboratory, September 2003, submitted to *Computing*.

[4] 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*.

[5] F. Abu Salem, "Parallel absolute irreducibility testing via polytopes", Report PRG-RR-04-14, Oxford University Computing Laboratory, June 2004, submitted to *Journal of Supercomputing*.

[6] L. M. Adleman, "The function field sieve", in *Algorithmic Number Theory* (Ithaca NY, 1994), *Lecture Notes in Computer Science* **877** (1994), Springer-Verlag, 108–121.

[7] W. Banks, D. Lieman, and I. Shparlinski, "An identification scheme based on sparse polynomials", *Proc. PKC '2000* (2000), *Lecture Notes in Computer Science* **1751**, Springer-Verlag, 68–74.

[8] E. R. Berlekamp, "Factoring polynomials over finite fields", *The Bell System Technical Journal* **46** (1967), 1853–1859.

[9] E. R. Berlekamp, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[10] E. R. Berlekamp, "Factoring polynomials over large finite fields", *Mathematics of Computation* **24** (1970), no. 111, 713–735.

[11] L. Bernardin, "On square-free factorization of multivariate polynomials over a finite field", *Theoretical Computer Science* **187** (1997), 105–116.

[12] L. Bernardin, "On bivariate Hensel lifting and its parallelization", in *Proc. of the International Symposium on Symbolic and Algebraic Computation* (1998), ACM Press, 96–100.

[13] L. Bernardin, "Factoring multivariate polynomials over a finite field", Diss. ETH (PhD thesis), 1999.

[14] R. H. Bisseling and W. F. McColl, "Scientific computing on bulk synchronous parallel architectures", *Proc. IFIP 13th World Computer Congress*, vol. I, North-Holland, 1994, 509–514.

[15] R. H. Bisseling, *Parallel Scientific Computation: A structured Approach using BSP and MPI*, Oxford University Press, Oxford, 2004.

[16] R. C. Bose and D. Y. Ray-Chaudhuri, "On a class of error correcting binary group codes", *Information and Control* **3** (1960), 68–79.

[17] R. P. Brent and H. T. Kung, "$O((n \log n)^{\frac{3}{2}})$ algorithms for composition and reversion of power series", *Analytic Computational Complexity* (1976), J. Traub, Ed., Academic Press, 217–225.

[18] R. P. Brent, S. Larvala and P. Zimmermann,"A fast algorithm for testing reducibility of trinomials mod 2 and some new primitive trinomials of degree 3021377", *Mathematics of Computation* **72** (2003), 1443–1452.

[19] R. P. Brent, S. Larvala and P. Zimmermann, "A primitive trinomial of degree 6972593", *Mathematics of Computation*, to appear.

[20] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, "The Paderborn University BSP (PUB) library - design, implementation, and performance", 13^{th} *International Parallel Processing Symposium* & 10^{th} *Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, 1999.

[21] D. Cantor and E. Kaltofen, "On fast multiplication of polynomials over arbitrary algebras", *Acta Informatica* **28** (1991), 693–701.

[22] D. Cantor and H. Zassenhaus, "A new algorithm for factoring polynomials over finite fields", *Mathematics of Computation* **36** (1981), no. 154, 587–592.

[23] 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).]

[24] A. L. Chistov, "Efficient factorization of polynomials over local fields" (Russian), *Dokl. Akad. Nauk SSSR* **293** (1987), no. 5, 1073–1077. [English translation: *J. Sov. Math.* **35** (1987), no. 2, 434–438.]

[25] A. L. Chistov, "Efficient factoring polynomials over local fields and its applications", *Proc. of the International Congress of Mathematicians*, Vol. I, II (Kyoto, 1990), 1509–1519, Math. Soc. Japan, Tokyo, 1991.

[26] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[27] A. R. Curtis and J. K. Reid, "The solution of large sparse unsymmetric systems of linear equations", *J. Inst. Maths. Applics.* **8** (1971), 344–353.

[28] S. R. Czapor, "Solving algebraic equations: combining Buchberger's algorithm with multivariate factorization", *Journal of Symbolic Computation* **7** (1989), no. 1, 49–54.

[29] S. R. Czapor, K. O. Geddes, and G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic, Boston, 1996.

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

[31] J. Della Dora and J. Fitch, *Computer Algebra and Parallelism*, Academic Press, London, 1989.

[32] A. Diaz and E. Kaltofen, "On computing greatest common divisors with polynomials given by black boxes for their evaluations", *Proc. of the International Symposium on Symbolic and Algebraic Computation* (1995), A. H. M. Levelt, Ed., ACM Press, 232–239.

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

[34] D. Duval, "Absolute factorization of polynomials: a geometric approach", *SIAM Journal on Computing* **20** (1991), 1–21.

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

[36] G. Ewald, *Combinatorial Convexity and Algebraic Geometry*, GTM 168, Springer, 1996.

[37] P. Flajolet, X. Gourdon, and D. Panario,"The complete analysis of a polynomial factorization algorithm over finite fields", *Journal of Algorithms* **40** (2001), 37–81.

[38] P. Fleischmann, "Connections between the algorithms of Berlekamp and Niederreiter for factoring polynomials over finite fields", *Linear Algebra and its Applications* **192** (1993), 101–108.

[39] P. Fleischmann, M. Holder, and P. Roelse, "The black-box Niederreiter algorithm and its implementation over the binary field", *Mathematics of Computation* **72** (2003), 1887–1899.

[40] J. B. Fraleigh, *A First Course in Abstract Algebra*, 4^{th} Ed., Addison-Wesley, Reading, Mass., 1989.

[41] S. Gao and J. von zur Gathen, "Berlekamp's and Niederreiter's polynomial factorization algorithms", *Contemporary Mathematics* **168** (1994), 101–116.

[42] D. Gale, "Irreducible convex sets", *Proc. Intern. Congr. Math.* **2** (1954), Amsterdam, 217–218.

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

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

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

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

[47] S. Gao and A.G.B. Lauder, "Fast absolute irreducibility testing via Newton polytopes", preprint 2003.

[48] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[49] J. von zur Gathen and E. Kaltofen, "Polynomial-time factorization of multivariate polynomials over finite fields", *Proc. of ICALP '83* (1983), *Lecture Notes in Computer Science* **154**, Springer-Verlag, 250–262.

[50] J. von zur Gathen, "Irreducibility of multivariate polynomials", *Journal of Computer and System Sciences* **31** (1985), 225–264.

[51] J. von zur Gathen and E. Kaltofen, "Factorization of multivariate polynomials over finite fields", *Mathematics of Computation* **45** (1985), no. 171, 251–261.

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

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

[54] J. von zur Gathen and J. Gerhard, "Polynomial factorization over $\mathbb{F}_2$", *Mathematics of Computation* **71** (2002), no. 240, 1677–1698.

[55] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, Cambridge, 1999.

[56] J. von zur Gathen and D. Panario, "Factoring polynomials over finite fields: A survey", *Journal of Symbolic Computation* **31** (2001), 3–17.

[57] J. von zur Gathen and V. Shoup, "Computing Frobenius maps and factoring polynomials", *Computational Complexity* **2** (1992), 187–224.

[58] P. Gianni and P. Trager, "Square-free algorithms in positive characteristic", *Journal of Applicable Algebra in Engineering, Communication and Computing* **7** (1996), 1–14.

[59] R. Göttfert, "An acceleration of the Niederreiter factorization algorithm in characteristic 2", *Mathematics of Computation* **62** (1994), no. 206, 831–839.

[60] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set", *Information Processing Letters* **1** (1972), 132–133.

[61] D. Grant, K. Krastev, D. Lieman, and I. Shparlinski, "A publick key cryptosystem based on sparse polynomials", *Proc. of an International Conference on Coding Theory, Cryptography, and Related Areas* (2000), Springer-Verlag, 114–121.

[62] D. Yu Grigoryev and A. L. Chistov, "Fast factorization of polynomials into irreducible ones and the solution of systems of algebraic equations" (Russian), *Dokl. Akad. Nauk SSSR* **275** (1984), no. 6, 1302–1306. [English translation: *J. Sov. Math.* **29** (1984), no. 2, 380–383.]

[63] 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).]

[64] B. Grünbaum, *Convex Polytopes*, John Wiley and Sons, New York, 1967.

[65] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations", in *Sparse Matrices and their Applications*, D. J. Rose and R. A. Willoughby, Eds., Plenum Press, New York, 1972, 41–52.

[66] J. M. D. Hill, W. F. McColl, and D. B. Skillicorn, "Questions and answers about BSP", *Scientific Programming* **6** (1997), 249–274.

[67] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudrea, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. H. Bisseling, "BSPlib: The BSP programming library", *Parallel Computing* **24** (1998), 1947–1980.

[68] J. W. P. Hirschfeld, *Projective Geometries over Finite Fields*, Clarendon Press, Oxford, 1979.

[69] A. Hocquenghem, "Codes correcteurs d'erreurs", *Chiffres* **2** (1959), 147–156.

[70] M. van Hoeij, "Factoring polynomials and the knapsack problem," *Journal of Number Theory* **95** (2002), 167–189.

[71] M. A. Inda and R. H. Bisseling, "A simple and efficient parallel FFT algorithm using the BSP model", *Parallel Computing* **27** (2001), no. 14, 1847-1878.

[72] N. Jacobson, *Basic Algebra I*, Freeman, San Francisco, 1974.

[73] J. Jájá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, Mass., 1992.

[74] E. Kaltofen, "Polynomial factorization 1982-1986", *Computers and Mathematics* (1986), D. V. Chudnosvky and R. D. Jenks, Eds., Marcel Dekker Inc., New York, 285–309.

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

[76] E. Kaltofen, "Sparse Hensel lifting", *Proc. of Eurocal '85, Vol. II* (1985), B. F. Caviness, Ed., *Lecture Notes in Computer Science* **204**, Springer-Verlag, 469–489.

[77] E. Kaltofen, "Computing the irreducible real factors and components of an algebraic curve", *Applicable Algebra in Engineering, Communication and Computing* **1** (1990), 135–148.

[78] 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.

[79] E. Kaltofen, "Polynomial factorization 1987-1991", *Proc. of LATIN '92* (1992), *Lecture Notes in Computer Science* **583**, Springer-Verlag, 294–313.

[80] E. Kaltofen, "Effective Noether irreducibility forms and applications", *Journal of Computer and System Sciences* **50** (1995), no. 2, 274–295.

[81] E. Kaltofen and A. Lobo, "Factoring high-degree polynomials by the black box Berlekamp algorithm", *Proc. International Symposium on Symbolic and Algebraic Computing, ISSAC 1994*, 90–98, ACM Press.

[82] E. Kaltofen and V. Shoup, "Subquadratic-time factoring of polynomials over finite fields", *Mathematics of Computation* **67** (1998), no. 223, 1179–1197.

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

[84] D. Knuth, *The Art of Computer Programming*, vol.2, 3rd Ed., Addison-Wesley, Reading, Mass., 1997.

[85] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, New York, London, 1994.

[86] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[87] T.C.Y. Lee, Scott A. Vanstone, "Subspaces and polynomial factorizations over finite fields", *Proc. of AAECC 6* (1995), *Lecture Notes in Computer Science* **357**, Springer-Verlag, 147–157.

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

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

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

[91] A. K. Lenstra, H.W. Lenstra, Jr. and L. Lovász, "Factoring polynomials with rational coefficients", *Mathematische Annalen*, **161** (1982), 515–534.

[92] R. Lidl and H. Niederreiter, *Finite Fields*, Encyclopedia of Mathematics and its Applications, Vol. 20, Addison-Wesley, Reading, Mass., 1983.

[93] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge University Press, Cambridge, 1986.

[94] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming", *Management Science* **3** (1957), 255–269.

[95] J. L. Massey, "Step by step decoding of the Bose-Chaudhuri-Hocquenghem codes", *IEEE Transactions on Information Theory* **IT-11** (1965), 580–585.

[96] P. McMullen, "Indecomposable convex polytopes", *Israel Journal of Mathematics* **58** (1987), no. 3, 321–323.

[97] R. J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, Boston, Lancaster, 1987.

[98] A. J. Menezes, P. C. Vam Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, Boca Raton, Fla., London, CRC, 1996.

[99] W. Meyer, "Indecomposable polytopes", *Transactions of the American Mathematical Society* **190** (1974), 77-86.

[100] M. Mignotte, *Mathematics for Computer Algebra*, Springer-Verlag, New York, 1992.

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

[102] H. Niederreiter, "A New efficient factorization algorithm for polynomials over small finite fields", *Applicable Algebra in Engineering, Communication and Computing* **4** (1993), 81–87.

[103] H. Niederreiter, "Factorization of polynomials and some linear algebra problems over finite fields", *Linear Algebra and its Applications* **192** (1993), 301–328.

[104] H. Niederreiter, "Factoring polynomials over finite fields using differential equations and normal bases", *Mathematics of Computation* **62** (1994), 819–830.

[105] H. Niederreiter and R. Göttfert, "Factorization of polynomials over finite fields and characteristic sequences", *Journal of Symbolic Computation* **16** (1993), no. 5, 401–412.

[106] M. Noro and K. Yokoyama, "Yet another practical implementation of polynomial factorization over finite fields", *Proc. of the International Symposium on Symbolic and Algebraic Computation* (2002), T. Mora, Ed., ACM Press, 200–206.

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

[108] A. M. Ostrowski, "Über die Bedeutung der Theorie der konvexen Polyeder für die formale Algebra", *Jahresberichte Deutsche Math. Verein* **30** (1921), 98-99.

[109] H. Popp, *Moduli Theory and Classification Theory of Algebraic Varieties*, *Lecture Notes in Computer Science* **620** (1977), Springer-Verlag.

[110] P. Roelse, "Factoring high-degree polynomials over $\mathbb{F}_2$ with Niederreiter's algorithm on the IBM SP2", *Mathematics of Computation* **68** (1999), no. 226, 869–880.

[111] M. Rothstein, *Aspects of Symbolic Integration and Simplification of Exponential and Primitive Functions*, PhD thesis, University of Wisconsin-Madison, 1976.

[112] M. Rothstein, "A new algorithm for the integration of exponential and logarithmic functions", *Proc. of the 1977 MACSYMA Users Conference*, 263–274.

[113] R. Rubinfeld and R. Zippel, "A new modular interpolation algorithm for factoring multivariate polynomials (extended abstract)", *Proc. of 1994 Algorithmic Number Theory Symposium* (1994), L. M. Adleman and M.-D. Huang, Eds, *Lecture Notes in Computer Science* **877**, Springer-Verlag, 93–107.

[114] W. M. Ruppert, "Reducibility of polynomials $f(x, y)$ modulo p", *Journal of Number Theory* **77** (1999), 62–70.

[115] G. C. Shephard, "Decomposable convex polyhedra", *Mathematika* **10** (1963), 89-95.

[116] H. Schildt, *C: The Complete Reference*, McGraw-Hill, New York, California, 1995.

[117] R. Schneider, *Convex Bodies: the Brunn-Minkowski Theory*, Encyclopedia of Math. and its Appl., Vol. 44, Cambridge University Press, Cambridge, 1993.

[118] W. Schreiner, "Parallelizing the big prime Berlekamp algorithm with distributed Maple", *Technical Report 02-07*, RISC-Linz, Johannes Kepler University, Linz, Austria, July 2001.

[119] V. Shoup, "On the deterministic complexity of factoring polynomials over finite fields", *Information Processing Letters* **33** (1990), 261–267.

[120] V. Shoup, "A new polynomial factorization algorithm and its implementation", *Journal of Symbolic Computation* **20** (1995), no. 4, 363–397.

[121] Z. Smilansky, "An indecomposable polytope all of whose facets are decomposable", *Mathematika* **33** (1986), no. 2, 192–196.

[122] Z. Smilansky, "Decomposability of polytopes and polyhedra", *Geometriae Dedicata* **24** (1987), no. 1, 29–49.

[123] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst, "Parallel sparse LU decomposition on a mesh network of transputers", *SIAM Journal on Matrix Analysis and Applications* **14** (1993), no. 3, 853–879.

[124] H. Stichtenoth, *Algebraic Function Fields and Codes*, Universitext, Springer-Verlag, Berlin, 1993.

[125] T. Szönyi, "Some applications of algebraic curves in finite geometry and combinatorics", in *Surveys in Combinatorics, 1997*, London Mathematical Society Lecture Notes Series **241**, Cambridge University Press, 197–236.

[126] B. M. Trager, "Algebraic factoring and rational function integration", *Proc. of the International Symposium on Symbolic and Algebraic Computation* (1976), R. D. Jenks, Ed., ACM Press, 219–226.

[127] L. G. Valiant, "A bridging model for parallel computation", *Communications of the ACM* **33** (1990), no. 8, 103–111.

[128] D. Wan, "Factoring polynomials over large finite fields", *Mathematics of Computation* **54** (1990), no. 190, 755–770.

[129] P. S. Wang, "Symbolic computation and parallel software", *Proc. First International Conference of the Austrian Center for Parallel Computation* (1991), *Lecture Notes in Computer Science* **591**, 316–337.

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

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

[132] D. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Transactions on Information Theory* **IT-32** (1986), no. 1, 54–62.

[133] D. Y. Y. Yun, "On square-free decomposition algorithms", *Proc. of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, ACM Press, 26–35.

[134] H. Zassenhaus, "On Hensel factorization I", *Journal of Number Theory* **1** (1969), 291–311.