Prime-field-complete functions and factoring polynomials over finite fields [1]

Lajos Rónyai *Ágnes Szántó*

Computer and Automation Institute
Hungarian Academy of Sciences
Budapest, Lágymányosi u. 11.
H-1111 Hungary
e-mail: lajos@nyest.ilab.sztaki.hu

Abstract. We relate the arithmetic straight-line complexity over a field $GF(p)$ (p is a prime) of the parity function l_p to the Boolean complexity of the problem of factoring polynomials over finite fields of characteristic p. A procedure is described which converts an arithmetic straight-line program for l_p into a factoring algorithm. As a consequence, a short straight-line program for l_p would imply the existence of an efficient factoring method.
Key words. Arithmetic complexity, Boolean complexity, straight-line program, factoring polynomials over finite fields.

1 Introduction

In a recent paper [SF] Sturtivant and Frandsen compared the computational power of Boolean circuits (straight-line programs) and arithmetic circuits (straight-line programs) over finite fields. They focused on functions having polynomial size Boolean circuits and studied the arithmetic complexity of such functions. One of the interesting results in [SF] is that there are hardest functions in this class, the *prime-field-complete* functions.

Let p be a prime. The *standard representation* of the field $GF(p)$ with bit strings is obtained by taking the binary representation of the smallest nonnegative integer representing an element $a \in GF(p)$. With this representation fixed, we can consider Boolean circuits (straight-line programs) computing

[1] Research partially supported by Hungarian National Foundation for Scientific Research (OTKA), Grants 2581, T016503 and EC Cooperative Action IC 1000, Algorithms for Future Technologies (ALTEC).

functions on $GF(p)$. In this setting the field operations $\{+,-,*,/\}$ clearly all have polynomial-size Boolean circuits.

A function $f : GF(p) \to GF(p)$ is prime-field-complete, if f has a Boolean circuit of size $(\log p)^{O(1)}$ in the standard representation of $GF(p)$; and any function $g : GF(p) \to GF(p)$ that also has polynomial-size Boolean circuits has polynomial-size arithmetic circuits over $GF(p)$ if and only if f has polynomial-size arithmetic circuits over $GF(p)$.

In [SF] several prime-field-complete functions are exhibited and studied, one of them being the *parity function* l_p. For a residue class $a \in GF(p)$, $l_p(a) = 0$ if the smallest nonnegative integer representing a is even, $l_p(a) = 1$ otherwise. Another example is the function $f_p : GF(p) \to GF(p)$ defined by

$$f_p(a) = \frac{z - z^p}{p} \mathrm{mod}\ p,$$

where z is the standard integer representative of the residue class $a \in GF(p)$, and the fraction on the right is computed in integer arithmetic.

The aim of this note to point to a possible connection of arithmetic complexity and traditional (Turing machine) complexity. We do this by relating the arithmetic complexity of the parity function l_p to the Turing machine complexity of an important problem from computational number theory: the problem of factoring polynomials over finite fields. The input to this problem is a polynomial $f \in GF(q)[x]$ over a finite field $GF(q)$, and we seek the irreducible factors of f over $GF(q)$. No deterministic polynomial time algorithm is known for the problem. For a background on factoring polynomials over finite fields, the reader is referred to [Ba], [Le], [LN].

We show (Theorem 1) that an arithmetic straight-line program for l_p (and hence for many other prime-field-complete fuctions) can be converted efficiently into a factoring algorithm for finite fields of characteristic p. In particular, from an explicitly given straight-line-program of length $(\log p)^{O(1)}$ a deterministic polynomial time algorithm for the factoring problem could be obtained.

2 The result

Let $h : GF(p) \to GF(p)$ be a function on $GF(p)$, and $f \in GF(p)[x]$ be a polynomial such that $f(x) | x^p - x$. The *greatest common divisor* $gcd(h, f) \in$

$GF(p)[x]$ of h and f is defined as follows. Take an arbitrary polynomial $f_1 \in GF(p)[x]$ which represents h in the sense that $h(a) = f_1(a)$ for every $a \in GF(p)$, and set $gcd(h, f) := gcd(f_1, f)$, where the gcd in the right is the usual (monic) greatest common divisor of polynomials. To see that the definition makes sense, we observe first that the existence of f_1 follows via Lagrange-interpolation. Moreover, if $f_2 \in GF(p)[x]$ is also a representative of h, then $f_1 - f_2$ vanishes on $GF(p)$, implying that $x^p - x$ and consequently f divides $f_1 - f_2$. We infer that $gcd(f_1, f) = gcd(f_2, f)$, showing that $gcd(h, f)$ is independent of the representative f_1 selected.

In a similar fashion, we can define $h(\text{mod}f)$ the reduction of h modulo f as $h(\text{mod}f) := f_1(\text{mod}f)$. Clearly the polynomial $h(\text{mod}f)$ is also independent of the particular selection of f_1.

It is immediate, that $gcd(h, f) = \prod_{a \in S}(x - a)$, where S is the set of common roots of h and f:

$$S = \{a \in GF(p);\ h(a) = f(a) = 0\}.$$

Recall that an arithmetic straight-line program P over $GF(p)$ is a sequence of functions $h_1, h_2, \ldots, h_k$, $h_i : GF(p) \to GF(p)$ such that h_i is either a constant $a \in GF(p)$, or the identity function on $GF(p)$, or $h_i = h_j \circ h_l$, where $j, l < i$ and $\circ$ is one of the field-operations $+, -, *, /$. We say that P computes h_k, and k is the length of P. The program P is *division-free* if only the operations $+, -, *$ are employed in P.

The statement below formulates the observation that if h has a short division-free program over $GF(p)$ then $gcd(h, f)$ can be computed efficiently. Suppose that we have a division-free program

$$P = h_1, h_2, \ldots, h_k$$

over $GF(p)$ which computes the function $h : GF(p) \to GF(p)$.

Proposition 1 *Let $f \in GF(p)[x]$ be a polynomial, $f(x)|x^p - x$. Suppose further, that we have a division-free program*

$$P = h_1, h_2, \ldots, h_k$$

over $GF(p)$ which computes the function $h : GF(p) \to GF(p)$. Then the polynomial $gcd(h, f)$ can be computed in deterministic time $k(\log k + deg(f) + \log p)^{O(1)}$.

Remark. The output is the dense representation of the polynomial $gcd(h, f)$, and the bound is for the number of bit operations of the algorithm.
Proof. As P is a division-free arithmetic straight-line program, the functions h_i can be considered as polynomials (of possibly very high degree) in a natural way, the last one h_k representing h. It suffices to find efficiently the gcd of the polynomials h_k and f. This task is easy if we can compute $h_k(\mathrm{mod} f)$, the remainder of h_k after division by f efficiently. This is accomplished by successively computing the polynomials $f_i = h_i(\mathrm{mod} f)$ for $i = 1, \ldots, k$. If h_i is a constant, then f_i is (the same) constant. If h_i is the identity function, then we set $f_i(x) = x$. Otherwise we use the obvious relation $f_i = f_j \circ f_l(\mathrm{mod} f)$ if $h_i = h_j \circ h_l$ holds in P, where $\circ \in \{+, -, *\}$. We work with polynomials of degree less than $2deg(f)$, hence the cost of the arithmetics for computing f_i from f_j and f_l is $(deg(f) + \log p)^{O(1)}$. To manage the efficient retrieval of of f_j and f_l, we build a balanced search tree. The nodes of the tree are pairs of the form (f_j, j), and the key is j. When computing f_i, we retrieve (f_j, j) and (f_l, l) and after the computation insert (f_i, i). This requires no more than $(\log k + deg(f) + \log p)^{O(1)}$ bit operations. Now the statement follows. □

We apply this to the parity function and the factoring problem.

Theorem 1 *Suppose that we have a division-free program*

$$P = h_1, h_2, \ldots, h_k$$

over $GF(p)$ a for the parity function $l_p : GF(p) \to GF(p)$. Then we can factor a polynomial $f \in GF(p)[x]$ into irreducible factors in deterministic time $k(\log k + deg(f) + \log p)^{O(1)}$.

Proof. By Berlekamp's reduction ([Be] or [LN, Chapter 4]) it suffices to prove the statement for polynomials f dividing $x^p - x$. We can then interpret the roots of f as integers from the interval $[0, p-1]$.

The algorithm is based on the *reduction step* described below. Suppose that from the polynomial f to be factored we know that the roots of f are all in the interval $[0, k]$, for an integer $0 < k < p$, and write $n = deg(f) > 1$.

We compute $f^* = gcd(l_p, f)$. Three cases are distinguished.
(a) $0 < deg(f^*) < deg(f)$. In this case f^* is a nontrivial divisor of f.
(b) $f^* = 1$. Then put $f_1(x) = \frac{1}{2^n} f(2x + 1)$.
(c) $f^* = f$. Then put $f_1(x) = \frac{1}{2^n} f(2x)$.
This finishes the description of the reduction step.

We claim that in Cases (b) and (c) we have $f_1|x^p - x$, the roots of f_1 are in the interval $[0, \lfloor k/2 \rfloor]$ and the roots of f can be efficiently recovered from the roots of f_1.

In Case (b) the roots of f are all odd. If $a \leq k$ and $f(a) = 0$, then $b = \frac{a-1}{2}$ is an integer, $f_1(b) = 0$, and $b \leq \lfloor k/2 \rfloor$. Similarly, in Case (c) the roots of f are all even. If $a \leq k$ and $f(a) = 0$, then $b = \frac{a}{2}$ is an integer, $f_1(b) = 0$, and $b \leq \lfloor k/2 \rfloor$.

In the factoring algorithm we repeat the reduction step until at some stage partial factorization is obtained (Case (a) is encountered); this is then 'pulled back' to obtain partial factorization of the input polynomial f. More formally, we compute the sequence of polynomials $f_i \in GF(p)[x]$ and integers k_i for $i = 1, \ldots, last$, where $f_1 = f$, $k_1 = p-1$; f_{i+1} is obtained from f_i by an application of the reduction step for $i > 1$, provided that termination occurs in Case (b) or (c). Otherwise f_{i+1} is not defined. If f_{i+1} is defined, then we set $k_{i+1} = \lfloor k_i/2 \rfloor$.

It is immediate that $deg(f_i) = n$, $f_i(x)|x^p - x$, the roots of f_i are in the interval $[0, k_i]$, and from the roots of f_{i+1} the roots of f_i can be obtained by a simple linear substitution. These facts imply that $last < \log_2 p$. The reduction step must terminate in Case (a) after at most $\log_2 p$ applications, giving a partial factorization $f_{last} = g_1 g_2$ of f_{last}, and working backwards along the sequence $f_1, \ldots, f_{last}$, we obtain a partial factorization $f = f'f''$, $1 < deg(f') < n$ of f. The gcd-computation contributes $k(\log k + n + \log p)^{O(1)}$ to the cost of one run of the reduction step. The rest is $(n + \log p)^{O(1)}$, hence the total cost of the reduction step is $k(\log k + n + \log p)^{O(1)}$. From the inequality for $last$ we infer the bound $k(\log k + n + \log p)^{O(1)}$ for the cost of a partial factorization of f. To obtain the complete factorization of f, we call the partial factorization procedure for the nonlinear factors obtained. We need $n - 2$ more such calls, where the input polynomials have degree less than n, and the statement follows. □

Corollary 1 *Suppose that we have an arithmetic straight-line program*

$$P = h_1, h_2, \ldots, h_k$$

over $GF(p)$ for the parity function $l_p : GF(p) \to GF(p)$. Then we can factor a polynomial $f \in GF(p)[x]$ into irreducible factors in deterministic time $k(\log k + deg(f) + \log p)^{O(1)}$.

Proof. We modify first P to eliminate divisions. If a function h_i is obtained in P as a result of division, say $h_i = h_j/h_l$, then h_i is replaced by $h_j h_l^{p-2}$. This is justified, because h_l can not have a root in $GF(p)$, therefore we have $1/h_l(a) = h_l^{p-2}(a)$ for every $a \in GF(p)$. We augment P by a subprogram computing h_l^{p-2}. This process does not increase the length of P significantly. The length of the division-free program is at most $2k(\log_2 p + 1)$. The statement now readily follows from Theorem 1. $\square$

As a direct application, a very short program for l_p can be turned into an efficient algorithm for the factoring problem.

Corollary 2 *Suppose that we have an arithmetic straight-line program*

$$P = h_1, h_2, \ldots, h_k$$

over $GF(p)$ for the parity function $l_p : GF(p) \to GF(p)$, where $k < (\log p)^c$ for a positive constant c. Then we can factor polynomials over finite fields of characteristic p in deterministic polynomial time. $\square$

3 Concluding remarks

1. A 'uniformity' assumption about the arithmetic straight-line program for l_p seems to be essential. Our algorithm explicitly makes use of a program for l_p, mere existence of a short program is not sufficient. For this reason l_p can not automatically be replaced in the statements with an arbitrary other prime-field-complete function g. That is possible, however, if the reduction of l_p to g is efficient, and this is the case with the functions considered in [SF]. In particular, Theorem 1 and the Corollaries remain valid with f_p in the place of l_p.

On the other hand, the efficient probabilistic algorithms for factoring imply the existence of (non-uniform) small deterministic Boolean circuits for the problem. Thus, the result of the paper is interesting only if uniform straight-line programs are considered.

2. The dependence on p of the running time of the best known deterministic procedure for factoring polynomials over finite fields of characteristic p is $\sqrt{p}$ (cf. Shoup [Sh]). To obtain an improvement in the way outlined here, a division-free program for l_p with $k = O(p^\alpha)$ would be needed where $0 < \alpha < 1/2$.

We note here (we owe this observation to the anonymous referee) that there is an arithmetic straight-line program of length $\sqrt{p}(\log p)^{O(1)}$ for l_p. This is done using fast algorithms (cf. [BP, Chapter 1]) for polynomial multiplication and multi-point polynomial evaluation: let p be an odd prime and set

$$d = \lfloor\sqrt{\frac{p+1}{2}}\rfloor, \quad d' = \lfloor\frac{p+1}{2d}\rfloor \text{ and } g(x) := \prod_{i=0}^{d-1}(x-2i).$$

Next we put

$$h_j(x) := g(x-2jd) \text{ for } 0 \leq j < d', \text{ and } h^*(x) := g(x-(p-1-2d+2)).$$

We have $h_j(x), h^*(x) \in GF(p)[x]$. Moreover, for $a \in GF(p)$ we have $l_p(a) = (h^*(a)\cdot h_0(a)\cdots h_{d'-1}(a))^{p-1}$. Thus, to obtain $l_p(a)$, it suffices to evaluate $g(x)$ at about $\sqrt{p}$ points. As $deg(g) \leq \sqrt{p}$, this multi-point evaluation problem can be solved by using $O(\sqrt{p}\log^2 p)$ arithmetical operations. This approach leads to an arithmetic straight-line program of length $\sqrt{p}(\log p)^{O(1)}$ for l_p. With this program the algorithm of Theorem 1 roughly matches Shoup's bound.

References

[Ba] Bach, E. Number theoretic algorithms, *Annual Review of Computer Science* 4(1990), 119-172.

[Be] Berlekamp, E. R. Factoring polynomials over large finite fields, *Mathematics of Computation* 24(1970), 713-735.

[BP] Bini, D. and Pan, V. *Polynomial and matrix computations, Vol. I*, Birkhäuser, 1994.

[Le] Lenstra, H. W., Jr. Algorithms for finite fields, *Proc. Conf. on Number Theory and Cryptography (Macquarie University, Sydney, 1989)* Cambridge University Press, 1989.

[LN] Lidl, R. and Niederreiter, H. *Finite fields*, Addison-Wesley, Reading, Ma, 1983.

[Sh] Shoup, V. On the deterministic complexity of factoring polynomials over finite fields, *Information Processing Letters* 33(1990), 261-267.

[SF] Sturtivant, C. and Frandsen, G. S. The computational efficacy of finite-field arithmetic, *Theoretical Computer Science* 112(1993), 291-309.