SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Solution methods to polynomial equations over $\mathbb{Z}_2$

av

Mark Anderstam

2014 - No 2

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Solution methods to polynomial equations over $\mathbb{Z}_2$

Mark Anderstam

Självständigt arbete i matematik 15 högskolepoäng, Grundnivå

Handledare: Samuel Lundqvist

2014

Abstract

We take a look at multivariable polynomial equations over $\mathbb{Z}_2$ and describe the tools needed to find solutions in $\mathbb{Z}_2^n$. We start by presenting a particular kind of solution method based on the fundamental "trial and error" strategy and we then use this to, eventually, find three different methods for solving these polynomial equations in a more systematic way. We later implement the solution methods in Python using arrays of booleans to represent monomials and asks which algorithm is more time efficient. We also say something about what could be improved in the future to make the algorithms more efficient and explain why further development of one algorithm is of greater interest to us.

Contents

1 Introduction

This paper deals with various methods for solving multivariable polynomial equations over the finite field $\mathbb{Z}_2$. We are only interested of solutions in $\mathbb{Z}_2^n$ and since the number of possible solutions is finite in $\mathbb{Z}_2^n$, one method could of course be to try (by evaluation) each and every point in $\mathbb{Z}_2^n$. Something that is not possible if considering polynomial equations over an infinite field, since this would require us to try an infinite number of values. But, nonetheless, even in a finite field this technique becomes quite cumbersome with increasing n. Despite this we still base our first (and only this) solution method on this "trial and error" technique (not using polynomial evaluation however); which will then act as a foundation for the rest of the solution methods. But before all this we present, in Theorem 3.1, a way to form a single polynomial equation from a system of polynomial equations over $\mathbb{Z}_2$ and thus enabling us to solve the system using these solution methods.

We go through all the theory behind finding these solution methods and once they have been understood and stated, we aim to implement three out of four of these methods in the programming language Python, compare the efficiency, in terms of execution time, and analyse the implemented algorithms for both better and worse. We end with a discussion on possible improvements of, particularly, one of the algorithms.

Before we say something about the originality and origin of some of the results, we give some background and a short lead-in to closely related topics.

It is highly unlikely not to encounter the theory of Gröbner bases when researching polynomials over a field F [1] [2]. Gröbner bases can be used to determine things about the solution set. If we have a system $f_1 = 0, \ldots, f_m = 0$ in the polynomial ring $F[x_1, \ldots, x_n]$, it has a finite number of solutions in the algebraic closure of F if and only if the number of monomials that are not a multiple of any leading monomials in the Gröbner basis of $f_1, \ldots, f_m$ are finite (see Theorem 6 and Proposition 8 in [3]). When $F = \mathbb{Z}_2$ and we are only interested of solutions over $\mathbb{Z}_2$, and not in the algebraic closure of $\mathbb{Z}_2$, then we can add the field equations $x_1^2 - x_1, \ldots, x_n^2 - x_n$ (see Definition 3.4 in [2]) to our system. This forces the solution set to be in $\mathbb{Z}_2^n$, since the field equations only can be solved by points in $\mathbb{Z}_2^n$ (see Theorem 3.5 in [2]). It then holds that the number of solutions in $\mathbb{Z}_2^n$, to the system $f_1 = 0, \ldots, f_m = 0$, equals the number of monomials that are not a multiple of any leading monomials in the Gröbner basis of $f_1, \ldots, f_m, x_1^2 - x_1, \ldots, x_n^2 - x_n$. We can thus get the exact number of solutions in this way. Gröbner basis computation is the normal approach, when we are interested in the solution set of a system of polynomial equations over $\mathbb{Z}_2$. It gives the number of solutions, but not the structure of the solution set. The methods that we will present in this paper will serve as an alternative approach.

One related topic that often arise together with the problem of solving a $\mathbb{Z}_2$-polynomial equation is the SAT-problem [2]; the problem of boolean satisfiability. That is, given a boolean formula, the question is weather we can assign truth values so that the formula evaluates to true. It turns out that we can recursively transform a boolean formula to a $\mathbb{Z}_2$-polynomial, while preserving satisfiability as solvability (see Theorem 3.1 in [2]). This can further be reduced to look at the corresponding boolean polynomial (a polynomial where both coefficient and degree per variable are in $\{0, 1\}$) of the $\mathbb{Z}_2$-polynomial, which will be discovered in Theorem 2.3. We know that all $\mathbb{Z}_2$-polynomials f has a solution in $\mathbb{Z}_2^n$, except when $f = 1$ (see Theorem 3.2 in [1]). Hence a boolean formula is satisfiable if and only if its transformation to a boolean polynomial does not equal 1. If we instead have a system of boolean formulas it will be satisfiable if and only if 1 is not in the Gröbner basis for the corresponding boolean polynomials plus the field equations (see Theorem 3.8 in [2]). Alternatively, after the transformation to a system of boolean polynomials, we can create the single equation by the formula presented in Theorem 3.1 and the problem reduces to simply determine if this single equation equals 1. But, as often when something is recursive, the transformation in itself is the hard part; as seen in Example 18 in [2].

One area where the problem of determining the solutions in $\mathbb{Z}_2^n$ to a system of equations over $\mathbb{Z}_2$ naturally arise, is coding theory (error-correcting codes) [4]. If we have a code determined by a parity-check matrix (a basis for linear code generating all its possible codewords) and if we consider each row as an equation equating to zero, the different codewords that can be generated is equal to the solutions to the system of equations.

Own results presented in Section 4 are Lemma 4.6, Theorem 4.7 and the second part of both Theorem 4.22 and Theorem 4.23. Whereas the rest of the results in that section as well as in Section 3 are already established, in one form or another, in [1].

2 Preliminaries

Let F be a field. Then a polynomial ring $F[x_1, \ldots, x_n]$ is the set of polynomials in the variables $x_1, \ldots, x_n$ with coefficients in F. We will exclusively stick to multivariable (n variable) polynomials with coefficients in $\mathbb{Z}_2$, in other words polynomials in $\mathbb{Z}_2[x_1, \ldots, x_n]$. Furthermore, we also introduce the term boolean polynomials, when we speak of polynomials where both coefficients and degree per variable are in $\{0, 1\}$. We define the boolean monomial of a monomial $m = x_1^{i_1} \cdots x_n^{i_n}$ as $\text{bool}(m) = \prod_{i_j \neq 0} x_j$ and the boolean polynomial of a polynomial $f = m_1 + \ldots + m_s$ is thus $\text{bool}(f) = \text{bool}(m_1) + \ldots + \text{bool}(m_s)$, where $m_1, \ldots, m_s$ are monomials. From now on, when working in the set of boolean polynomials, we regard

the equality $m_1 \cdot m_2 = \text{bool}(m_1 m_2)$, often omitting to print it out.

Example 2.1. Let $f = x_1^2 x_2^3 x_3 + x_1 x_2^3 + x_1 x_3^4 + x_1^2 + x_2 + 1$ be a polynomial in $\mathbb{Z}_2[x_1, x_2, x_3]$. The corresponding boolean polynomial of f, is simply $g = \text{bool}(f) = x_1 x_2 x_3 + x_1 x_2 + x_1 x_3 + x_1 + x_2 + 1$.

In other words, a boolean polynomial is a polynomial in $\mathbb{Z}_2[x_1, \ldots, x_n]$ of the form $x^{\alpha_1} + \ldots + x^{\alpha_s}$, where $\alpha_i = (\alpha_{i_1}, \ldots, \alpha_{i_n})$ is an exponent vector in $\{0,1\}^n$. The $x^{\alpha_i} = x_1^{\alpha_{i_1}} \cdots x_n^{\alpha_{i_n}}$ are, as mentioned, called boolean monomials. Sometimes, often when only regarding monomials, we will simply write x^α when α is in $\{0,1\}^n$ or simpler just x when the preliminaries are well understood.

The zero set (or solution set) in $\mathbb{Z}_2^n$ for a polynomial f is denoted by $V(f)$ and naturally $V(f) = \{p \in \mathbb{Z}_2^n \mid f(p) = 0\}$. We can also say that f vanishes exactly on the points of $V(f)$. We give an example of one way to find the zero set in $\mathbb{Z}_2^n$.

Example 2.2. Let $f = x_1 x_2 x_3 + x_1 x_2 + x_1 x_3 + x_1 + x_2 + 1$ be a polynomial in $\mathbb{Z}_2[x_1, x_2, x_3]$. To find the zero set of f in $\mathbb{Z}_2^3$, we evaluate $f(p)$ for all $2^3 = 8$ possible points p in

$$\mathbb{Z}_2^3 = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\},$$

one by one. Hence, $V(f) = \{(0,1,0), (0,1,1), (1,0,0), (1,1,0), (1,1,1)\}$.

The set of boolean polynomials f in n variables have a one-to-one correspondence with subsets of $\mathbb{Z}_2^n$, defined by $V(f)$ (see Sats 11 in [5]). Thus we can denote V^{-1} as the inverse function of V. We say that, for some collection of points P, $V^{-1}(P)$ is the boolean polynomial which vanishes excactly on P. Furthermore, it follows that since $V(0) = \mathbb{Z}_2^n$ and $V(1) = \varnothing$, boolean polynomials not in $\{0,1\}$ all have a unique zero set that equals a (proper) nontrivial subset of $\mathbb{Z}_2^n$. This makes the following (trivial) theorem, even more, fitting.

Theorem 2.3. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a polynomial in $\mathbb{Z}_2[x_1, \ldots, x_n]$. Then $V(f) = V(\text{bool}(f))$ in $\mathbb{Z}_2^n$.*

Proof. Consider x_i^a, $a \in \{1, 2, \ldots\}$, and x_i. For any b in $\mathbb{Z}_2$, we have $b^a = b$, and so the polynomial functions f and $\text{bool}(f)$ are equal when considered as functions from $\mathbb{Z}_2^n$ into $\mathbb{Z}_2^n$. □

3 System of $\mathbb{Z}_2$-polynomials and subsets of boolean monomials

If we have a system of equations $f_1 = 0, \ldots, f_m = 0$ in $\mathbb{Z}_2[x_1, \ldots, x_n]$, it can always be solved from $V(f_1) \cap \cdots \cap V(f_m)$. But an alternate, very convenient, method is presented in the following theorem. Note that from now on we work in $\mathbb{Z}_2$, unless otherwise stated.

Theorem 3.1. *Let $f_1 = 0, \ldots, f_m = 0$ be a system of equations in $\mathbb{Z}_2[x_1, \ldots, x_n]$. Then the zero set in $\mathbb{Z}_2^n$ is equal to the zero set in $\mathbb{Z}_2^n$ to the single equation $(f_1 + 1) \cdots (f_m + 1) + 1 = 0$ in $\mathbb{Z}_2[x_1, \ldots, x_n]$.*

Proof. Suppose $p \in V((f_1, \ldots, f_m))$, where $V((f_1, \ldots, f_m))$ is the zero set in $\mathbb{Z}_2^n$ for the system of equations $f_1 = 0, \ldots, f_m = 0$. Then

$$\begin{aligned}((f_1 + 1) \cdots (f_m + 1) + 1)(p) &= (f_1(p) + 1) \cdots (f_m(p) + 1) + 1 \\ &= (0 + 1) \cdots (0 + 1) + 1 = 1 + 1 = 0.\end{aligned}$$

Now, suppose instead that $p \notin V((f_1, \ldots, f_m))$. Then

$$\begin{aligned}((f_1 + 1) \cdots (f_m + 1) + 1)(p) &= (f_1(p) + 1) \cdots (f_m(p) + 1) + 1 \\ &= 0 + 1 = 1,\end{aligned}$$

since at least one of $f_1(p), \ldots, f_m(p)$ evaluates to 1. Hence $p \in V((f_1, \ldots, f_m))$ if and only if $p \in V((f_1 + 1) \cdots (f_m + 1) + 1)$, that is $V((f_1, \ldots, f_m)) = V((f_1 + 1) \cdots (f_m + 1) + 1)$. □

Example 3.2. Let $f_1 = x_1x_2x_3 + x_1x_2 + x_1x_3 + x_1 + x_2 + 1, f_2 = x_1x_2 + x_1 + x_2, f_3 = x_1 + x_2 + x_3$ be a system of polynomials in the three variables x_1, x_2, x_3. To find $V((f_1, f_2, f_3))$ in $\mathbb{Z}_2^3$, we first calculate $(f_1+1)(f_2+1)(f_3+1) + 1 = (x_1x_2x_3 + x_1x_2 + x_1x_3 + x_1 + x_2)(x_1x_2 + x_1 + x_2 + 1)(x_1 + x_2 + x_3 + 1) + 1 = (4 \cdot x_1x_2x_3 + 4 \cdot x_1x_2 + 2 \cdot x_1x_2x_3 + 2 \cdot x_1x_3 + 2 \cdot x_1x_2 + 2 \cdot x_1 + 2 \cdot x_1x_2 + 2 \cdot x_2)(x_1 + x_2 + x_3 + 1) + 1 = 0 \cdot (x_1 + x_2 + x_3 + 1) + 1 = 1$, when working in the set of boolean polynomial. But $V(1) = \varnothing$, so it turns out that the system is unsolvable in $\mathbb{Z}_2^3$.

The least common multiple of boolean monomials $m_1, \ldots, m_s$ is denoted by $\mathrm{lcm}(m_1, \ldots, m_s)$ and is equal to $\mathrm{bool}(m_1 \cdots m_s)$. The degree of a monomial $x^\alpha = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ is $\deg(x^\alpha) = \alpha_1 + \ldots + \alpha_n$. For example, we have $\deg(\mathrm{lcm}(x_1x_2, x_1x_3, x_2, 1)) = \deg(x_1x_2x_3) = 3$.

Now, let M be the set of all boolean monomials in $\mathbb{Z}_2[x_1, \ldots, x_n]$, that is $M = \{x^p \mid p \in \mathbb{Z}_2^n\}$. We define

$$(m_1, \ldots, m_s) = \{\mathrm{lcm}(m_i, m) \mid 1 \leq i \leq s,\, m \in M\}.$$

Or equivalently, $(m_1, \ldots, m_s) = \{\mathrm{bool}(m_i m) \mid 1 \leq i \leq s,\, m \in M\}$. So $(m_1, \ldots, m_s)$ consists of all multiples of each of the monomials $m_1, \ldots, m_s$. This means that $(1) = M$. We also say that $m_1, \ldots, m_s$ generates $(m_1, \ldots, m_s)$. The advanced reader recognizes $(m_1, \ldots, m_s)$ as an ideal in the monoid (M, lcm).

Theorem 3.3. *Let $I = (m_1, \ldots, m_s)$, in n variables. Then the number of elements in I is equal to $d_1 + d_2 + \ldots + d_s$, where*

$$d_i = (-1)^{i+1} \sum_{1 \leq j_1 < \cdots < j_i \leq s} 2^{n - \deg(\mathrm{lcm}(m_{j_1}, \ldots, m_{j_i}))}.$$

Proof. For each $1 \leq i \leq s$, let c_i be the condition that $x \in I$ is divisible by m_i. Then the elements in I satisfies at least one of the conditions c_i, where $1 \leq i \leq s$. Hence, we denote the number of elements in I by $N(c_1 \text{ or } c_2 \text{ or } \ldots \text{ or } c_s)$. Applying the Principle of Inclusion and Exclusion [6], we find that

$$N(c_1 \text{ or } \ldots \text{ or } c_s) = \sum_{1 \leq i \leq s} N(c_i) - \sum_{1 \leq i < j \leq s} N(c_i c_j) + \sum_{1 \leq i < j < k \leq s} N(c_i c_j c_k) + \ldots + (-1)^s N(c_1 \cdots c_s).$$

We note that, for example, $N(c_1 c_2 c_3)$ denotes the number of elements in I that is divisible by $\operatorname{lcm}(m_1, m_2, m_3)$ and this is equal to the number of elements in $(\operatorname{lcm}(m_1, m_2, m_3))$.

Now, we find that $N(c_i) = 2^{n - \deg(m_i)} = 2^{n - \deg(\operatorname{lcm}(m_i))}$, since every element $x \in (m_i)$ is equal to $m_i \cdot m$ for some $m \nmid m_i$, such that $\deg(m) \leq n - \deg(m_i)$. Also the α_i of $m = x^\alpha$ can be either 0 or 1. Furthermore, we find that $N(c_i c_j) = 2^{n - \deg(\operatorname{lcm}(m_i, m_j))}$, since every element $x \in (\operatorname{lcm}(m_i, m_j))$ is equal to $\operatorname{lcm}(m_i, m_j) \cdot m$ for some $m \nmid \operatorname{lcm}(m_i, m_j)$, such that $\deg(m) \leq n - \deg(\operatorname{lcm}(m_i, m_j))$. Continuing this reasoning, we see that $N(c_i \cdots c_t) = 2^{n - \deg(\operatorname{lcm}(m_i, \ldots, m_t))}$ for $1 \leq i < t \leq s$, which concludes the proof. □

Remark 3.4. *An interesting consequence of Theorem 3.3, is if we have a system of boolean monomials $m_1 = 0, \ldots, m_s = 0$ in n variables, the number of solutions to this system is equal to the number of boolean monomials outside $(m_1, \ldots, m_s)$ and this equals $2^n - (d_1 + \ldots + d_s)$, where $d_1, \ldots, d_s$ are denoted as in Theorem 3.3.*

We illustrate the previous theorem with an example.

Example 3.5. The number of elements in $(x_1x_3x_4, x_1x_2, x_2x_3, x_4)$, in four variables, is given by $d_1 + d_2 + d_3 + d_4$ where

$$\begin{aligned}
d_1 &= 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_2))} + 2^{4 - \deg(\operatorname{lcm}(x_2x_3))} + 2^{4 - \deg(\operatorname{lcm}(x_4))} \\
&= 2^{4-3} + 2^{4-2} + 2^{4-2} + 2^{4-1} = 18, \\
d_2 &= -(2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_1x_2))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_2x_3))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_4))} \\
&\qquad + 2^{4 - \deg(\operatorname{lcm}(x_1x_2, x_2x_3))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_2, x_4))} + 2^{4 - \deg(\operatorname{lcm}(x_2x_3, x_4))}) \\
&= -(2^{4-4} + 2^{4-4} + 2^{4-3} + 2^{4-3} + 2^{4-3} + 2^{4-3}) = -10, \\
d_3 &= 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_1x_2, x_2x_3))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_1x_2, x_4))} \\
&\qquad + 2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_2x_3, x_4))} + 2^{4 - \deg(\operatorname{lcm}(x_1x_2, x_2x_3, x_4))} \\
&= 2^{4-4} + 2^{4-4} + 2^{4-4} + 2^{4-3} = 4, \\
d_4 &= -2^{4 - \deg(\operatorname{lcm}(x_1x_3x_4, x_1x_2, x_2x_3, x_4))} = -2^{4-4} = -1.
\end{aligned}$$

Giving the total of $18 - 10 + 4 - 1 = 11$ elements. These elements form $\{x_1x_3x_4, x_1x_2x_3x_4, x_1x_2, x_1x_2x_3, x_1x_2x_4, x_2x_3, x_2x_3x_4, x_4, x_1x_4, x_2x_4, x_3x_4\}$.

4 Solution methods

In this section we present different methods to find the zero set in $\mathbb{Z}_2^n$ for polynomials in $\mathbb{Z}_2[x_1, \ldots, x_n]$. These methods are stated in the last theorem of each subsection; after we have gathered enough theory to understand and show them. They are thus Theorem 4.2, Theorem 4.8 and Theorem 4.23.

4.1 Relationship between boolean monomials and zeros

We now turn to boolean polynomials entirely, since by Theorem 2.3 we can always find the zero set in $\mathbb{Z}_2^n$ to $\text{bool}(f)$ and this will then equal the zero set in $\mathbb{Z}_2^n$ to f. In fact, the rest of the main results is based on the following lemma, which statement only is true for boolean monomials. It introduces a connection between exponent vectors and zeros of boolean monomials.

Lemma 4.1. *Let x^p be a boolean monomial. Then*

$$x^p(q) = \begin{cases} 1 & \text{if } x^p | x^q, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. $x^p \nmid x^q$ if and only if $q_i = 0$ when $p_i = 1$ for at least one $i = 1, 2, \ldots$. But then $x^p(q) = 0$, which concludes the proof. □

This result may seem trivial, but turn out to be very useful, which the following theorem shows.

Theorem 4.2. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a boolean polynomial. Then*

$$V(f) = \{p \mid p \in \mathbb{Z}_2^n,\ x^p \text{ is divisible by an even number of } x^{\alpha_i}\}.$$

Proof. Let k denote the number of x^{α_i} that divides x^p. By Lemma 4.1, we know that $x^{\alpha_i}(p) = 1$ if and only if $x^{\alpha_i} | x^p$, thus $f(p) = x^{\alpha_1}(p) + \ldots + x^{\alpha_s}(p) = \underbrace{1 + \ldots + 1}_{k \text{ times}} = k \cdot 1 = 0$ if and only if k is even. □

Remark 4.3. *A direct consequence of Theorem 4.2, is that given $V(f)$ we have that $V(f + 1) = V(f)^{\mathrm{c}}$.*

The following example shows how Theorem 4.2 can be used to find the zero set in $\mathbb{Z}_2^n$ of a boolean polynomial.

Example 4.4. Consider $f = x_1x_2x_3 + x_1x_2 + x_1x_3 + x_1 + x_2 + 1$ from Example 2.2. To find $V(f)$, instead of going through and evaluating the polynomial for all points in $\mathbb{Z}_2^3$, we go through all boolean monomials m in $\{x^p \mid p \in \mathbb{Z}_2^3\}$ and check if an even number of the monomials in f divides m. We find that an even number of the monomials in f divides x_2, x_2x_3, x_1, x_1x_2 and $x_1x_2x_3$. Likewise, we see that an odd number of monomials in f divides 1, x_3 and x_1x_3. Hence, $V(f) = \{(0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (1, 1, 1)\}$.

4.2 Alternating sum based on an inclusion-exclusion argument

We define the logarithm to base x of a boolean polynomial $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ as $\log(f) = \{\log(x^{\alpha_1}), \ldots, \log(x^{\alpha_s})\} = \{\alpha_1, \ldots, \alpha_s\}$. In order for the use of log to make sense it is natural to define $x^{\log(f)} = f$, and so $x^{\{\alpha_1,\ldots,\alpha_s\}} = x^{\alpha_1} + \ldots + x^{\alpha_s}$, where α_i is a point in $\mathbb{Z}_2^n$ (or, if an exponent vector, in $\{0,1\}^n$). Furthermore, given monomials $m_1, \ldots, m_s$, let $I = (m_1, \ldots, m_s)$ and define

$$H(I) = \sum_{x \in (m_1,\ldots,m_s)} x.$$

With the logarithm of a boolean polynomial previously introduced, we note that an alternate form of $H(I)$ is $x^{\log(\sum_{x\in(m_1)} x) \cup \cdots \cup \log(\sum_{x\in(m_s)} x)}$.

The reason for introducing the following definition will become clear in Theorem 4.8, the main result of this section.

Definition 4.5. *Given a sequence of boolean monomials $m_1, \ldots, m_s$, we define the sequence of* lcm*-combinations, from $m_1, \ldots, m_s$, as*

$$\begin{aligned}
h_0 &= H((1)),\\
h_1 &= H((m_1, m_2 \ldots, m_s)),\\
h_2 &= H((\mathrm{lcm}(m_1, m_2), \mathrm{lcm}(m_1, m_3), \ldots, \mathrm{lcm}(m_{s-1}, m_s))),\\
h_3 &= H((\mathrm{lcm}(m_1, m_2, m_3), \mathrm{lcm}(m_1, m_3, m_4), \ldots, \mathrm{lcm}(m_{s-2}, m_{s-1}, m_s))),\\
&\vdots\\
h_s &= H((\mathrm{lcm}(m_1, m_2, \ldots, m_s))).
\end{aligned}$$

Lemma 4.6. $\log(h_s) \subseteq \log(h_{s-1}) \subseteq \cdots \subseteq \log(h_0)$.

Proof. We show that $p \in \log(h_{i+1})$ implies $p \in \log(h_i)$, by considering x^p denoted x. Suppose $x \in h_{i+1}$. Then $x = \mathrm{lcm}(m_{j_1}, \ldots, m_{j_{i+1}}) \cdot m$ for some monomial m, where $j_1 < \cdots < j_{i+1}$. Consider

$$x' = \mathrm{lcm}(m_{j_1}, \ldots, m_{j_i}) \cdot \mathrm{lcm}(m_{j_{i+1}}, m) \in h_i,$$

where $j_1 < \cdots < j_i$. We are done if we can show that $x = x'$. But this follows directly from the fact that

$$\begin{aligned}
\mathrm{lcm}(m_{j_1}, \ldots, m_{j_{i+1}}) \cdot m &= \mathrm{lcm}(m_{j_1}, \ldots, m_{j_i}, m_{j_{i+1}}, m)\\
&= \mathrm{lcm}(m_{j_1}, \ldots, m_{j_i}) \cdot \mathrm{lcm}(m_{j_{i+1}}, m).
\end{aligned}$$

Thus, if $x^p \in h_{i+1}$, then $x^p \in h_i$, which is equivalent with what we wanted to show. $\square$

Theorem 4.7. *Let $v_0 = \log(h_0 - h_1), v_1 = \log(h_2 - h_3), \ldots, v_k = \log(h')$, where $h' = h_{2k} - h_{2k+1}$ if s is odd and $h' = h_{2k}$ if s is even. Here, $h_0, h_1, \ldots, h_s$ are defined as in Definition 4.5. Then $v_i \cap v_j = \varnothing$ for all $i \neq j$, so that*

$$\log(h_0 - h_1 + h_2 - h_3 + \ldots + (-1)^s h_s) = v_0 \sqcup v_1 \sqcup \cdots \sqcup v_k.$$

Proof. Assume, on the contrary, that there exists a point $p \in v_i \cap v_j$, $i \neq j$. We also assume, with no loss of generality, that $j > i$. Further let s be odd, so that all $v_0, v_1, \ldots, v_k$ have the same structure (the case when s is even is proved analogously). Then $p \in v_i = \log(h_{2i}) \setminus \log(h_{2i+1})$ which implies that $p \in \log(h_{2i})$ and $p \notin \log(h_{2i+1})$, and it also follows that $p \in v_j = \log(h_{2j}) \setminus \log(h_{2j+1})$ which implies that $p \in \log(h_{2j})$ and $p \notin \log(h_{2j+1})$. But since $\log(h_{2j+1}) \subseteq \log(h_{2j}) \subseteq \cdots \subseteq \log(h_{2i+1}) \subseteq \log(h_{2i})$, by Lemma 4.6, if $p \notin \log(h_{2i+1})$ then $p \in \log(h_{2j})$ is a contradiction. Hence $v_0, v_1, \ldots, v_k$ are pairwise disjoint. □

Before we continue we note that the summands of h_i are divisible by at least i of the monomials m_i. This fact will be useful in the proof to the following theorem.

Theorem 4.8. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a boolean polynomial. Then*

$$V(x^{\alpha_1} + \ldots + x^{\alpha_s}) = \log(h_0 - h_1 + h_2 - h_3 + \ldots + (-1)^s h_s),$$

where $h_0, h_1, \ldots, h_s$ is the sequence of lcm-combinations from $x^{\alpha_1}, \ldots, x^{\alpha_s}$, as defined in Definition 4.5.

Proof. We show that the expression $h_0 - h_1 + h_2 - h_3 + \ldots + (-1)^s h_s$ contains exactly all possible monomials that is divisible by an even number of x^{α_i}, $1 \leq i \leq s$. Since, by Theorem 4.7, we know that $v_0, v_1, \ldots, v_k$ do not contain any common elements, we can consider $v_0, v_1, \ldots, v_k$ separately. If we first examine $x^{v_0} = h_0 - h_1$: h_1 contains exactly all monomials divisible by at least one x^{α_i}, thus $h_0 - h_1$ equals exactly all monomials not divisible by any x^{α_i}. Now, considering $x^{v_1} = h_2 - h_3$: h_2 contains exactly all monomials divisible by two or more x^{α_i}, h_3 monomials divisible by three or more x^{α_i}. Thus $h_2 - h_3$ equals exactly all monomials divisible by exactly two x^{α_i}. Continuing in this manner we get that x^{v_j} equals all monomials divisible by an even number of x^{α_i}, for $j = 0, \ldots, k$. Hence $\log(h_0 - h_1 + h_2 - h_3 + \ldots + (-1)^s h_s)$ contains exactly every point p such that x^p is divisible by an even number of x^{α_i}. By Theorem 4.2, this concludes the proof. □

Remark 4.9. *As a side note: Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s} = 0$ be a multivariable boolean polynomial equation and $g = x^s + c_1 x^{s-1} + \ldots + c_s = 0$ a univariable equation over $\mathbb{C}$. We can assume that g factors into linear components $(x - d_1) \cdots (x - d_s)$, where d_i are the roots of the polynomial g. If we expand*

this expression we get $x^s - (d_1 + d_2 + \ldots + d_s)x^{s-1} + (d_1d_2 + d_1d_3 + \ldots + d_{s-1}d_s)x^{s-2} - (d_1d_2d_3 + d_1d_3d_4 + \ldots + d_{s-2}d_{s-1}d_s)x^{s-3} + \ldots + (-1)^s d_1d_2 \cdots d_s$. *It is interesting to note the similarity between this expanded expression and the alternating sum* $h_0 - h_1 + h_2 - h_3 + \ldots + (-1)^s h_s$, *for* f. *As known, the former holds the solutions to the univariable equation when solved as a system (however impossible it can be to solve) and the logarithm of the latter contains the solutions to the multivariable equation.*

We end this section with an example of the method to compute the zero set, that we established in Theorem 4.8.

Example 4.10. Consider $f = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3$ in $\mathbb{Z}_2[x_1, x_2, x_3]$. We have

$$h_1 = H((1, x_1, x_2, x_1x_2, x_1x_3, x_1x_2x_3)) = H((1)) = h_0,$$

and for $h_2, h_3, \ldots, h_6$ we calculate the lcm-combinations of $2, 3, \ldots, 6$ monomials of f, respectively. It follows that

$$\begin{aligned}
h_2 &= H((x_1, x_2, x_1x_2, x_1x_3, x_1x_2x_3, x_1x_2, x_1x_2, x_1x_3, x_1x_2x_3, x_1x_2, \\
&\qquad x_1x_2x_3, x_1x_2x_3, x_1x_2x_3, x_1x_2x_3, x_1x_2x_3)) = H((x_1, x_2)) \\
&= x_1 + x_2 + x_1x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3, \\
h_3 &= H((x_1x_2, x_1x_2, x_1x_3, x_1x_2x_3, x_1x_2, x_1x_2x_3, x_1x_2x_3, x_1x_2x_3, \\
&\qquad x_1x_2x_3, x_1x_2x_3, x_1x_2, x_1x_2x_3, x_1x_2x_3, \ldots, x_1x_2x_3)) = H((x_1x_2, x_1x_3)) \\
&= x_1x_2 + x_1x_3 + x_1x_2x_3, \\
h_4 &= H((x_1x_2, x_1x_2x_3, x_1x_2x_3, \ldots, x_1x_2x_3)) = H((x_1x_2)) = x_1x_2 + x_1x_2x_3, \\
h_5 &= H((x_1x_2x_3, x_1x_2x_3, \ldots, x_1x_2x_3)) = H((x_1x_2x_3)) = x_1x_2x_3 = h_6.
\end{aligned}$$

Hence the zero set is

$$\begin{aligned}
V(f) &= \log(h_0 - h_1 + h_2 - h_3 + h_4 - h_5 + h_6) \\
&= \log(h_2 - h_3) \sqcup \log(h_4 - h_5) \sqcup \log(h_6) \\
&= \log(x_1 + x_2 + x_2x_3) \sqcup \log(x_1x_2) \sqcup \log(x_1x_2x_3) \\
&= \{(1,0,0), (0,1,0), (0,1,1)\} \sqcup \{(1,1,0)\} \sqcup \{(1,1,1)\} \\
&= \{(1,0,0), (0,1,0), (1,1,0), (0,1,1), (1,1,1)\}.
\end{aligned}$$

4.3 The vanishing polynomial and the zero set as an exponent

The following definition introduce a new concept which will eventually help us to find the boolean polynomial which vanishes on a certain collection of points. This together with the properties of the expression $x^{V(f)}$ will in turn provide us with two additional solution methods, which at least improves on readability upon the other methods presented in previous sections.

Definition 4.11. *Let $P = \{p_1, \ldots, p_s\}$ be a collection of points $p_i \in k^n$ for some set k. Then the function $f_{p_i} \colon P \to \{0, 1\}$ defined by*

$$f_{p_i}(p_j) = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{otherwise,} \end{cases}$$

is called a separator of p_i with respect to P.

Remark 4.12. *A direct consequence of Definition 4.11 is that if $k = \mathbb{Z}_2$, then $V(f_p) = \mathbb{Z}_2^n \setminus \{p\}$, and in turn by Remark 4.3, $V(f_p + 1) = \{p\}$. This property of separators makes them very useful for us and it will be convenient to have a method in finding separators of arbitrary points in $\mathbb{Z}_2^n$.*

Example 4.13. We have that x_1 is a separator of $(1, 0, 0)$ in three variables with respect to $P = \{(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)\}$, since $x_1(p)$ evaluates to zero on every point $p \in P \setminus \{(1, 0, 0)\}$ and to one when $p = (1, 0, 0)$.

Consider $x_1x_2+x_1x_2x_3$ in three variables. We see that $V(x_1x_2+x_1x_2x_3) = \log(h_0-h_1+h_2) = \log(\sum_{p\in\mathbb{Z}_2^n} x^p-(x_1x_2+x_1x_2x_3)+x_1x_2x_3) = \mathbb{Z}_2^3\setminus\{(1, 1, 0)\}$. Thus $x_1x_2 + x_1x_2x_3$ is a separator of $(1, 1, 0)$ with respect to $\mathbb{Z}_2^3$.

The next theorem shows the relation between separators and monomial sets generated from a boolean monomial. But first we need a lemma. Note that a submonomial of m is a monomial that divides m, or in other words if $x_1^{q_1} \cdots x_n^{q_n}$ is a submonomial of $x_1^{p_1} \cdots x_n^{p_n}$, then $p_i = 0$ implies $q_i = 0$.

Lemma 4.14. *Let m be a boolean monomial. Then there are $2^{\deg(m)}$ submonomials of m.*

Proof. Let $m = x^\alpha = x^{(\alpha_1,\ldots,\alpha_n)}$. Then only $\deg(m)$ of the α_i equals 1 and since these α_i can be either 1 or 0 in the submonomial, the result follows. □

Theorem 4.15. *Let $p \in \mathbb{Z}_2^n$. Then $f_p = \sum_{x\in(x^p)} x$ is a separator of p with respect to $\mathbb{Z}_2^n$.*

Proof. We follow the proof to Lemma 3.32 in [1], with some minor clarifications.

We show that $f_p(q) = 1$ when $q = p$ and $f_p(q) = 0$ otherwise, by counting the number of monomials in f_p that divides x^q and applying Theorem 4.2 on the result.

Let $q = p$. Then $x^p \in f_p$ is the only monomial that divides x^q, since $x^p = x^q$ is the generator of f_p (every other monomial in f_p has higher degree than x^p). Hence $f_p(q) = 1$ if $q = p$.

Let $q \neq p$. If $x^q \notin f_p$, no monomial in f_p divides x^q and so $f_p(q) = 0$. If $x^q \in f_p$, then $x^q = x^p \cdot m$ for some monomial m with the property that $\deg(m) > 0$ (since $x^q \neq x^p$). Now, since there are $2^{\deg(m)}$ submonomials of m by Lemma 4.14 and since $x^p \cdot m | x^q$, the latter relation also holds for all submonomials of m. That is, it holds for an even number of monomials in f_p. Thus $f_p(q) = 0$ if $q \neq p$. □

The following corollary gives a way of computing the boolean polynomial that vanishes on a collection of points. From now on when we use f_p, it is implied that $f_p = \sum_{x \in (x^p)} x$.

Corollary 4.16. *Let $P = \{p_1, \ldots, p_s\}$ be a collection of points $p_i \in \mathbb{Z}_2^n$. Then $V(f_{p_1} + \ldots + f_{p_s} + 1) = P$, where $f_{p_i} = \sum_{x \in (x^{p_i})} x$. That is, $V^{-1}(P) = f_{p_1} + \ldots + f_{p_s} + 1$ is the boolean polynomial which vanishes on P.*

Proof. If $p \in P$, then $(f_{p_1} + \ldots + f_{p_s} + 1)(p) = f_{p_1}(p) + \ldots + f_{p_s}(p) + 1 = 1 + 1 = 0$, since one and only one of the separators equals 1 on p. If instead $p \notin P$, $(f_{p_1} + \ldots + f_{p_s} + 1)(p) = 0 + \ldots + 0 + 1 = 1$, since none of the separators equals 1 on p. □

We next introduce another new function whose properties will be of great importance.

Definition 4.17. *Given a boolean polynomial f. We denote φ to be the function from the set of boolean polynomials to itself defined by*

$$\varphi(f) = x^{V(f)}.$$

The following example shows some interesting consequences of iterative use of φ.

Example 4.18. Consider $f = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3$ in $\mathbb{Z}_2[x_1, x_2, x_3]$. We know that $V(f) = \{(1,0,0), (0,1,0), (1,1,0), (0,1,1), (1,1,1)\}$ from before. So we have

$$\begin{aligned} g = \varphi(f) = x^{V(f)} &= x^{(1,0,0)} + x^{(0,1,0)} + x^{(1,1,0)} + x^{(0,1,1)} + x^{(1,1,1)} \\ &= x_1 + x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3 \end{aligned}$$

and

$$\begin{aligned} V(g) &= \log(h_0 - h_1 + h_2 - h_3 + h_4 - h_5) = \log((1 + x_3) + (x_2x_3) + (0)) \\ &= \log(1 + x_3 + x_2x_3) = \{(0,0,0), (0,0,1), (0,1,1)\}, \end{aligned}$$

noting that $V(\varphi(f)) = V(g) = \log(1 + x_3 + x_2x_3) = \log(f)^{\mathrm{c}} \cup \{(0, \ldots, 0)\}$ in this case.

Continuing the iteration of φ, we get

$$h = \varphi^2(f) = x^{V(g)} = 1 + x_3 + x_2x_3$$

and

$$\begin{aligned} V(h) &= \log(h_0 - h_1 + h_2 - h_3) = \log((0) + (x_3 + x_1x_3)) \\ &= \log(x_3 + x_1x_3), \end{aligned}$$

making

$$i = \varphi^3(f) = x^{V(h)} = x_3 + x_1x_3$$

and

$$\begin{aligned} V(i) &= \log(h_0 - h_1 + h_2) = \log((1 + x_1 + x_2 + x_1x_2) + (x_1x_3 + x_1x_2x_3)) \\ &= \log(1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3). \end{aligned}$$

Finally, we see that

$$\varphi^4(f) = x^{V(i)} = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3 = f.$$

Hence we propose that φ^4 is the identity, which will be proved shortly.

We also observe that

$$\begin{aligned} \varphi(f) + \varphi^2(f) + \varphi^3(f) = g + h + i &= (x_1 + x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3) \\ &\quad + (1 + x_3 + x_2x_3) + (x_3 + x_1x_3) \\ &= 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3 = f \end{aligned}$$

and this will also be shown to be true for all boolean polynomials, in Theorem 4.22.

What is interesting with the function φ is what we find, in the following theorem, when we compute the zero set in $\mathbb{Z}_2^n$ of $\varphi(f)$. We saw one instance of this in Example 4.18.

Theorem 4.19. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a boolean polynomial. Then*

$$V(\varphi(f)) = V(x^{V(f)}) = \begin{cases} \log(f)^{\mathrm{c}} \cup \{(0, \ldots, 0)\} & \text{if } 1 \in \{x^{\alpha_1}, \ldots, x^{\alpha_s}\}, \\ \log(f)^{\mathrm{c}} \setminus \{(0, \ldots, 0)\} & \text{if } 1 \notin \{x^{\alpha_1}, \ldots, x^{\alpha_s}\}. \end{cases}$$

Proof. The proof can be found to Theorem 3.35 in [1]. But it is important to convey that the proof is based on a simplified form of Theorem 4.8. □

The following corollary states a more compact expression of the result in Theorem 4.19 and a direct consequence of this.

Corollary 4.20. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a boolean polynomial. Then*

$$V(\varphi(f)) = \log(f + \sum_{p \in \mathbb{Z}_2^n} x^p + 1),$$

so that

$$V^{-1}(\log(f + \sum_{p \in \mathbb{Z}_2^n} x^p + 1)) = \varphi(f).$$

Proof. If $1 \in \{x^{\alpha_1}, \ldots, x^{\alpha_s}\}$, then $\log(f)^{\mathrm{c}} \cup \{(0, \ldots, 0)\} = \log(f + 1)^{\mathrm{c}}$. If $1 \notin \{x^{\alpha_1}, \ldots, x^{\alpha_s}\}$, then $\log(f)^{\mathrm{c}} \setminus \{(0, \ldots, 0)\} = \log(f+1)^{\mathrm{c}}$. Hence the result follows from Theorem 4.19 and since $\log(f+1)^{\mathrm{c}} = \log(f + \sum_{p \in \mathbb{Z}_2^n} x^p + 1)$. □

Now, it is time to prove the propositions we made in Example 4.18 for the function φ. But first we need the following lemma.

Lemma 4.21. $\varphi^2(f) = f + \sum_{p\in\mathbb{Z}_2^n} x^p + 1$.

Proof. Using Corollary 4.20, we get

$$\varphi^2(f) = \varphi(\varphi(f)) = x^{V(\varphi(f))} = x^{\log(f+\sum_{p\in\mathbb{Z}_2^n} x^p+1)} = f + \sum_{p\in\mathbb{Z}_2^n} x^p + 1.$$

□

Theorem 4.22.

$$\textit{(i)} \qquad \varphi^4(f) = f$$

and

$$\textit{(ii)} \qquad \varphi(f) + \varphi^2(f) + \varphi^3(f) = f.$$

Proof. (i) Using Lemma 4.21, we get

$$\begin{aligned}\varphi^4(f) = \varphi^2(\varphi^2(f)) &= \varphi^2(f + \sum_{p\in\mathbb{Z}_2^n} x^p + 1)\\ &= (f + \sum_{p\in\mathbb{Z}_2^n} x^p + 1) + \sum_{p\in\mathbb{Z}_2^n} x^p + 1 = f.\end{aligned}$$

(ii) Again, by Lemma 4.21,

$$\begin{aligned}\varphi(f) + \varphi^2(f) + \varphi^3(f) &= \varphi(f) + \varphi^2(f) + \varphi^2(\varphi(f))\\ &= \varphi(f) + (f + \sum_{p\in\mathbb{Z}_2^n} x^p + 1) + (\varphi(f) + \sum_{p\in\mathbb{Z}_2^n} x^p + 1) = f.\end{aligned}$$

□

Before we present the final solution methods, based on the results in this section, note that by Theorem 4.15, $\sum_{p\in\mathbb{Z}_2^n} x^p = f_{(0,\ldots,0)}$.

Theorem 4.23. *Let $f = x^{\alpha_1} + \ldots + x^{\alpha_s}$ be a boolean polynomial. Then*

$$\textit{(i)} \qquad V(f) = \log(f_{\alpha_1} + \ldots + f_{\alpha_s} + f_{(0,\ldots,0)})$$

and alternatively

$$\textit{(ii)} \qquad V(f) = \log(\sum_{p\in\log(f+f_{(0,\ldots,0)}+1)} f_p + 1).$$

Proof. (i) Using the second part of Theorem 4.22, Lemma 4.21, the first part of Theorem 4.22 and Corollary 4.16, respectively, we get

$$\begin{aligned} V(f) &= \log(x^{V(f)}) = \log(\varphi(f)) = \log(\varphi^3(f) + \varphi^2(f) + f) \\ &= \log(\varphi^3(f) + (f + f_{(0,\ldots,0)} + 1) + f) = \log(\varphi^3(f) + f_{(0,\ldots,0)} + 1) \\ &= \log(V^{-1}(\log(f)) + f_{(0,\ldots,0)} + 1) = \log(f_{\alpha_1} + \ldots + f_{\alpha_s} + f_{(0,\ldots,0)}). \end{aligned}$$

(ii) We have, by Corollary 4.20, that

$$\varphi(f) = x^{V(f)} = V^{-1}(\log(f + f_{(0,\ldots,0)} + 1)),$$

so that

$$V(f) = \log(V^{-1}(\log(f + f_{(0,\ldots,0)} + 1)))$$

and since $\sum_{p \in \log(f + f_{(0,\ldots,0)} + 1)} f_p + 1$ is the boolean polynomial which vanishes on $\log(f + f_{(0,\ldots,0)} + 1)$ by Corollary 4.16, we get

$$V(f) = \log(\sum_{p \in \log(f + f_{(0,\ldots,0)} + 1)} f_p + 1).$$

□

Remark 4.24. *The method, of Theorem 4.23, that has the least number of separators to sum should be less time consuming. So if $s > n/2$, where n is the number of variables, the alternative method should be picked. But this might depend on the individual monomials in f since, for instance, $f_{(1,0,0)}$ contains more monomials than $f_{(1,1,0)}$. So it might also be necessary to check the overall degree of the monomials.*

Example 4.25. For our recurring example, $f = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3$, the second part of Theorem 4.23 should in this case be the faster method, if we belive our assumption made in Remark 4.24. But we will see if this is actually correct. First, we have that

$$\begin{aligned} \sum_{p \in \log(f)} f_p + f_{(0,0,0)} &= f_{(1,0,0)} + f_{(0,1,0)} + f_{(1,1,0)} + f_{(1,0,1)} + f_{(1,1,1)} \\ &= (x_1 + x_1x_2 + x_1x_3 + x_1x_2x_3) + (x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3) \\ &\quad + (x_1x_2 + x_1x_2x_3) + (x_1x_3 + x_1x_2x_3) + (x_1x_2x_3) \\ &= x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3, \end{aligned}$$

so that

$$\begin{aligned} V(f) &= \log(x_1 + x_2 + x_1x_2 + x_2x_3 + x_1x_2x_3) \\ &= \{(1,0,0), (0,1,0), (1,1,0), (0,1,1), (1,1,1)\}. \end{aligned}$$

Secondly, we get that $f + f_{(0,0,0)} + 1 = 1 + x_3 + x_2x_3 = g$, hence

$$\begin{aligned}\sum_{p\in\log(g)} f_p + 1 &= (1 + x_1 + x_2 + x_3 + x_1x_2 + x_1x_3 + x_2x_3 + x_1x_2x_3) \\ &\quad + (x_3 + x_1x_3 + x_2x_3 + x_1x_2x_3) + (x_2x_3 + x_1x_2x_3) + 1 \\ &= x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3\end{aligned}$$

and the logarithm of this equals $V(f)$ above.

We notice that the first method actually performs better (less operations). This seems to be because 1 is present in f and therefore also in g.

5 Implementation

In this section we implement the different solution methods from the previous section. What follows are first a presentation of the execution times of the algorithms for different polynomials and a comparisons between them. After that comes a more technical walkthrough of the algorithms in Python and finally we discuss what can/could be done to improve the efficiency, of especially one, of the algorithms.

5.1 Results

Before we precede with the results, we note that Section 4 gaves us four ways to calculate the zero set in $\mathbb{Z}_2^n$ for boolean polynomials. These methods are presented in Theorem 4.2 (M1), Theorem 4.8 (M2) and Theorem 4.23 (M3 and M4). But we choose not to implement M4 (the second part of Theorem 4.23), after what was discussed in Remark 4.24: For example, if $f = x_1 + x_6x_9 + x_9x_{10} + \sum_{p\in\mathbb{Z}_2^{10}} x^p$ in $\mathbb{Z}_2[x_1, \ldots, x_{10}]$, we note that with M3 we need to add $2^{10} - 3 = 1021$ separators modulo 2, while using M4 would reduce this to calculating the sum of three separators modulo 2. Thus M4 gives us a very convenient workaround in cases of large amount of monomials and it means that a polynomial with s monomials could approximately be solved in the same time as a polynomial with $2^n - s$ monomials, but it cannot fairly be compared to the other methods.

Execution times for our recurring example, $f = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3$ in three variables, are less than 0.001 seconds for all three algorithms and this tells us nothing about the efficiency of the respective algorithm. Larger examples are required and these are presented in Table 1. Examples 1 to 18 are randomly generated with a higher chance of lower

Ex	Variables	Monomials	M1	M2	M3
1	9	10	< **0.01**	0.01	< **0.01**
2	9	20	< **0.01**	3.43	< **0.01**
3	9	101	0.02	-	**0.01**
4	12	10	0.03	0.02	**0.01**
5	12	20	0.05	3.42	**0.04**
6	12	101	0.25	-	**0.11**
7	15	10	0.22	0.16	**0.12**
8	15	20	0.39	3.61	**0.15**
9	15	101	1.72	-	**0.46**
10	18	10	1.95	0.89	**0.80**
11	18	20	3.19	4.63	**0.93**
12	18	101	14.41	-	**2.64**
13	21	10	14.10	7.92	**6.83**
14	21	20	24.98	13.16	**8.00**
15	21	101	111.98	-	**11.41**
16	24	10	112.14	71.12	**61.28**
17	24	20	196.19	74.49	**61.32**
18	24	101	990.63	-	**73.76**
19	20	15	10.79	7.65	**3.41**
20	20	15	**10.81**	36.94	12.21

Table 1: Comparison between solution method 1 (M1), method 2 (M2) and method 3 (M3). Examples 1-18 are pseudorandomized and examples 19-20 are preset. Times are in seconds in Python 2.7.6 (64 bit) on a Intel Core i3-3220T CPU @ 2.80GHz with 8.0GB RAM. Boldface represent most efficient.

degree ($< n/2$) monomials. For instance, the polynomial in example 7 is

$$\begin{aligned}&x_6x_8x_9x_{11}x_{14}x_{15} + x_5x_8x_9x_{10}x_{12}x_{13}x_{14} + x_4x_{11}x_{12}x_{14} + x_4x_6x_7x_8x_{10}x_{11}x_{13}\\&\quad + x_2x_6x_9x_{14} + x_2x_5x_8x_{11}x_{13} + x_2x_3x_4x_5x_8x_{11}x_{12} + x_1x_3x_6x_9x_{10}x_{11}x_{12}x_{13}\\&\quad + x_1x_3x_5x_6x_7x_{12}x_{14}x_{15} + x_1x_3x_4x_7x_8x_{10}x_{11}.\end{aligned}$$

We observe that M3 performes best in examples 1 through 18, but M3 (and M2) is highly dependent on the degree of the monomials in the polynomial, unlike M1, which examples 19 versus 20 shows; the latter example is the same as the former except that one monomial of degree three and three monomials of degree five were swapped with four monomials with degree one. The polynomial in example 20 is

$$\begin{aligned}&1 + x_1 + x_2 + x_4 + x_7 + x_2x_{17}x_{18}x_{19}x_{20} + x_3x_5x_6x_7x_8\\&\quad + x_3x_9x_{11}x_{15}x_{17} + x_4x_5x_6x_7x_{17} + x_5x_6x_9x_{12}x_{16} + x_6x_{10}x_{14}x_{17}x_{19}\\&\quad + x_7x_8x_{16}x_{18}x_{20} + x_9x_{11}x_{12}x_{13}x_{14} + x_{10}x_{11}x_{12}x_{15}x_{19} + x_{11}x_{12}x_{18}x_{19}x_{20}.\end{aligned}$$

Furthermore, for M2, examples with 10 versus 20 monomials present, in most cases, a big difference and this is because we work with $\binom{10}{5} = 252$ combinations of monomials at the most if in 10 variables and in 20 variables this grows to $\binom{20}{10} = 184756$ combinations. For the same reason, when the polynomial contained 101 monomials a result could not even be established, since by the algorithm it will early on attempt to store $\binom{101}{6} = 1267339920$ elements in a list, which result in memory depletion.

The results can be interpreted as follows: If our polynomial contains monomials of "higher" degree we pick M3 and if "lower" degree we pick M1. And as observed, the result can vary greatly upon choosing the correct method.

5.2 Technical discussion

This section is for the reader interested in a closer look at the algorithms and the technical aspects of the code. The algorithms are written in Python for two reasons: Its readability and because its standard library is platform independent. The reader not familiar with Python will notice how simple it is to follow. It has one drawback in contrast with, for instance, C++: It is slower in most cases. But since the aim is to implement and compare algorithms, the speed of the individual execution (affected by both the programming language and the system processor) is not important in this case.

Boolean monomials are represented by their exponent vectors. So, for example, $x_1x_3x_5$ in five variables is represented by $(1, 0, 1, 0, 1)$ or bitarray('10101') in our case; an array of bits. Furthermore, we declare the class object BitArray, as seen in Listing 1, which is an hashable bitarray. It means that BitArray objects will be unique when considered an element in a set, whereas bitarray objects is not. This will shorten the execution times significantly, since the time complexity is $O(1)$ to check if an element exists in a set, versus $O(\log(n))$ if checking existens of an element in a sorted list of length n. But note that bitarray is used where the properties of BitArray is not needed and they are also considered equal as objects, hence we call them both by bitarray from now on (except in the code). The great benefits of bitarray is the ability to perform bitwise operations and that they are memory efficient. The bitwise operations we use are bitwise AND (denoted by & in the code) and bitwise OR (denoted by | in the code), which is used to check divisibility and calculate the lcm of bitarrays representing boolean monomials, respectively.

Listing 1: Declares BitArray as an hashable bitarray

```
class BitArray(bitarray):
    def __init__(self, seq):
        bitarray.__init__(self)
```

```
    def __hash__(self):
        return hash(self.__repr__())
```

We further need a way to get combinations (with and without repetition) and arrangements of bitarrays. For this the functions of itertools (standard library) is very efficient and is thus used.

The algorithms for M1, M2 and M3 all takes a list of bitarrays representing the monomials of the polynomial that we want solved and the number of variables, and returns a list of bitarrays representing the points of the zero set.

Now, the following listing presents our algorithm for M1. Note that when executed, the function zeros_by_m1 is called, which in turn use functions even_nr_divides and, indirectly, m_divides_p. The algorithm iterates through every point p in $\mathbb{Z}_2^n$ and checks if an even number of monomials in the polynomial divides x^p.

Listing 2: Algorithm for M1

```
def m_divides_p(m, p):
    return m & p == m

def even_nr_divides(monomials, point):
    count = 0
    for monomial in monomials:
        if m_divides_p(monomial, point):
            count += 1
    return count % 2 == 0

def zeros_by_m1(monomials, n):
    points = itertools.product(bitarray('01'), repeat = n)
    v = []
    for point in points:
        bit_point = bitarray(point)
        if even_nr_divides(monomials, bit_point):
            v.append(bit_point)
    return v
```

Before we present the algorithm for M2 we need a function that return the monomial set generated from a monomial. Hence the function generate_monomials in the following listing takes a monomial and returns all multiples of that monomial. This function is also used by the algorithm for M3.

Listing 3: Function to generate all multiples of a monomial

```
def generate_monomials(monomial, n):
    if monomial == bitarray('0' * n):
        return set(BitArray(tupl) for tupl in
                   itertools.product(bitarray('01'), repeat = n))
    monomials = set()
    if monomial.count() <= n / 2:
```

```
        one_indices = monomial.search(bitarray('1'))
        arrangements_of_zeros = itertools.product(
            bitarray('01'), repeat = n - len(one_indices))
        for arrangement in arrangements_of_zeros:
            point = bitarray(arrangement)
            for i in one_indices:
                point.insert(i, True)
            monomials.add(BitArray(point))
    else:
        zero_indices = monomial.search(bitarray('0'))
        zero_indices_enumerated = list(enumerate(zero_indices))
        arrangements_of_zeros = itertools.product(
            bitarray('01'), repeat = len(zero_indices))
        for arrangement in arrangements_of_zeros:
            for i, j in zero_indices_enumerated:
                monomial[j] = arrangement[i]
            monomials.add(BitArray(monomial))
    return monomials
```

We can now present, in the following listing, the algorithm for M2. The function zeros_by_m2 is called when the algorithm is executed (like with the algorithm for M1), which use functions combinations_r_plus_1 and generate_h and, indirectly, generate_monomials from Listing 3. Under the hypotheses of Theorem 4.7, we know that $v_0, v_1, \ldots, v_k$ are pairwise disjoint. This fact is utilized in the algorithm for M2; we calculate $h_i - h_{i+1}$ separately. We also hold on to the (so called) generating system

$$(\mathrm{lcm}(m_1, m_2 \ldots, m_i), \mathrm{lcm}(m_1, m_3 \ldots, m_{i+1}), \ldots, \mathrm{lcm}(m_{s-i+1}, m_{s-i+2}, \ldots, m_s))$$

for h_i even after the calculation of h_i, for $i > 1$. This generating system for h_i is then used to calculate the generating system for h_{i+1}, resulting in only one operation instead of $(i + 1) - 1 = i$ operations, when calculating each element of the generating system for h_{i+1}.

Listing 4: Algorithm for M2

```
def combinations_r_plus_1(monomials, s, r_combinations, r):
    if r == 1:
        return [el[0] | el[1] for el in
                itertools.combinations(monomials, 2)]
    repeat = (seq[r - 1] for seq in
              itertools.combinations_with_replacement(
                  reversed(xrange(s - r + 1)), r))
    combinations = []
    for monomial, use in itertools.izip(r_combinations, repeat):
        for j in xrange(use):
            combinations.append(monomial | monomials[s - use + j])
    return combinations

def generate_h(h_generators, n):
    h = set()
    for monomial in h_generators:
```

```
        if not BitArray(monomial) in h:
            h.update(generate_monomials(bitarray(monomial), n))
    return h

def zeros_by_m2(monomials, n):
    v = []
    if not bitarray('0' * n) in monomials:
        v += generate_h([bitarray('0' * n)], n) - generate_h(
            monomials, n)
    s = len(monomials)
    h_last_generators = []
    for r in xrange(2, s, 2):
        h_first_generators = combinations_r_plus_1(
            monomials, s, h_last_generators, r - 1)
        h_last_generators = combinations_r_plus_1(
            monomials, s, h_first_generators, r)
        v += generate_h(h_first_generators, n) - generate_h(
            h_last_generators, n)
    if s % 2 == 0:
        v += generate_h(combinations_r_plus_1(
            monomials, s, h_last_generators, s - 1), n)
    return v
```

Lastly, Listing 5 contains our algorithm for M3 together with the function generate_monomials in Listing 3. The algorithm just adds separators one by one modulo 2.

Listing 5: Algorithm for M3

```
def zeros_by_m3(monomials, n):
    monomials.sort()
    if bitarray('0' * n) in monomials:
        v = set()
        monomials = monomials[1:]
    else:
        v = generate_monomials(bitarray('0' * n), n)
    for monomial in monomials:
        v.symmetric_difference_update(generate_monomials(monomial, n))
    return list(v)
```

For readers further wanting to test one or more of these algorithms, what follows is an explanation of how this can be done and an example run of all three methods in a Windows Command Prompt on Python 2.7.6 (64 bit). We assume that Python 2.7 (only guaranteed to work with versions of 2.7) is already installed together with tools to install packages (setuptools or pip). We first need to install the python package bitarray 0.8.1 [1] (or latest version).

Next, we create a file named, for example, solution_methods.py and put "from bitarray import bitarray" and "import itertools" on the first two lines.

[1] This module is not part of the standard library but can be found, together with installation instructions, at: `https://pypi.python.org/pypi/bitarray`

After that we copy and paste the code from Listing 1 to Listing 5 into solution_methods.py, in order. Once in the Command Prompt, navigate to the folder (we call it folder_name here) containing the file we just created. The following listing shows the rest of the steps necessary and a few example runs, in the Command Prompt, where we once more try our recurring example, $f = 1 + x_1 + x_2 + x_1x_2 + x_1x_3 + x_1x_2x_3$ in three variables. Note that M2 and M3 does not necessarily return the zero set (lists in the algorithms) in "default" order as M1 does, so they should be ordered if we want to compare the result from the different methods.

Listing 6: Example run in Windows Command Prompt

```
C:\...\folder_name>python
Python 2.7.6 ...
>>> from solution_methods import *
>>> n = 3
>>> f = [bitarray('000'), bitarray('100'), bitarray('010'), bitarray('110'),
     bitarray('101'), bitarray('111')]
>>> v_m1 = zeros_by_m1(f, n)
>>> v_m2 = zeros_by_m2(f, n)
>>> v_m3 = zeros_by_m3(f, n)
>>> v_m2.sort()
>>> v_m3.sort()
>>> v_m1 == v_m2 == v_m3
True
>>> v_m2
[bitarray('011'), bitarray('010'), bitarray('100'), bitarray('110'),
    bitarray('111')]
```

5.3 Future improvements and discussion

The algorithms are not aimed to be best imaginable, but merely constitute a guidance (or prototype) for further development. That being said, we note that our conclusion in Section 5.1 is based on these particular algorithms and does not conclude anything about other possible algorithms of the different methods. But at least the algorithms presented are similar in complexity and structure (M2 and M3 even share a function), so one other conclusion can be drawn: The problem of implementing an efficient algorithm for M2 is a harder one, than doing so for M1 and M3. But what also makes the problem of finding an efficient algorithm for M2 a more interesting one is because M2 finds solutions in a more systematic way than M1 and M3, due to what we showed in Theorem 4.7. This gives a hint that this method could, at least in some aspects, outperform the other methods and, for example, in cases when we would only be interested in the number of solutions to a polynomial equation over $\mathbb{Z}_2$; since we would not have to store intermediate points. Or, in cases, when we are satisfied with part of the zero set (where solutions also possess a certain quality).

A few problems (or limitations) arose when running the algorithms for large examples. The most frequent being that lists could only hold a certain number of bitarrays. An alternative to lists of bitarrays is to let bitarrays replace the lists entirely. So, for instance, the zero set could be represented by a long bitarray, remembering that points are of length n. Then the zero set could, in theory, contain up to $2^{63}/n$ points on a 64 bit system. This method to store bitarrays more efficiently, could be used in M2 when the combinations become to large to handle, like mentioned in Section 5.1.

One extensive approach to make the algorithm for M2 more efficient is to apply the theory in [7]. A direct approach that could make the algorithm for M2 perform better is the following: First calculate h_{i+1} and use this information to calculate h_i. We would thus only keep elements not already in h_{i+1}, since h_{i+1} is a subset of h_i, by Lemma 4.6. This could save both time and memory. Keep in mind that we want to preserve the procedure that use the generating system for h_i to calculate the generating system for h_{i+1}.

We propose and end with the following two questions unanswered:

Problem 1. *Given information about the generating system for h_i. How to find $m'_1, \ldots, m'_k$ so that the generating system $(m'_1, \ldots, m'_k)$ for h_{i+1} is minimal?*

Problem 2. *Given a minimum generating system for each h_i. How to calculate $h_i - h_{i+1}$ efficiently?*

6 Acknowledgements

I would like to thank my supervisor Samuel Lundqvist for the suggestion of topic and excellent guidance along the way. I am also grateful for the valuable comments from Karl Rökaeus.

References

[1] Samuel Lundqvist, *Elementary algebra related to the SAT problem.* arXiv:1211.3398v2, 2013.

[2] John Sass, *Boolean polynomials and Gröbner bases: An algebraic approach to solving the SAT-problem.* Master thesis. Stockholm University, 2011.

[3] David Cox, John Little, Donal O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer, Pages 234-235, 2007.

[4] Norman L. Biggs, *Discrete Mathematics.* Oxford University Press, Pages 343-345, 2003.

[5] Tobias Andersen, *Ekvationssystem i* $\mathbb{Z}_2[x_1, \ldots, x_n]$. Bachelor thesis. Stockholm University, 2013.

[6] Ralph P. Grimaldi, *Discrete and Combinatorial Mathematics: An Applied Introduction.* Pearson, Pages 389-390, 2004.

[7] Dave Bayer, Mike Stillman, *Computation of Hilbert functions.* Journal of Symbolic Computation. Volume 14, Issue 1, Pages 31-50, 1992.