SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Solving polynomial equations over $\mathbb{Z}_2$ using DPLL methods

av

Assar Andersson

2014 - No 15

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Solving polynomial equations over $\mathbb{Z}_2$ using DPLL methods

Assar Andersson

Självständigt arbete i matematik 15 högskolepoäng, Grundnivå

Handledare: Samuel Lundqvist

2014

Solving polynomial equations over $\mathbb{Z}_2$ using DPLL methods

Assar Andersson

June 15, 2014

Abstract

We start by proving some general properties of polynomials over $\mathbb{Z}_2$, and their connection to the boolean formulas. Next, we present computer representations, and algorithms to compute addition and multiplication, of polynomials over $\mathbb{Z}_2$. Finally, we implement and test some variations of the DPLL procedure to solve certain polynomial equations over $\mathbb{Z}_2$. We also say something about why certain DPLL variations preforms better than others.

Contents

1 Introduction

The aim of this paper is to tighten the connection between the SAT-problem, which is the problem of determining whenever a boolean formula is satisfiable or not, and polynomial equations over $\mathbb{Z}_2$.

Various authors have studied algebraic approaches related to polynomials over $\mathbb{Z}_2$ to decide if a boolean formula is satisfiable or not [6], [5], [3]. However, it appears that there still is a huge gap in performance between these methods and the top of the line methods. It is still an open question whether this is because we have not studied these methods enough, or that there is simply no hope for these methods.

In this paper, we will adapt the Davis-Putnam-Logemann-Loveland (DPLL) procedure, which is the base of most of the top of the line SAT-solvers [4], to find solutions to polynomial equations over $\mathbb{Z}_2$. By doing so, we hope to get a better picture of what we are missing in our algebraic approaches to the SAT-problem.

We will begin by discussing some properties of polynomials over $\mathbb{Z}_2$, and present some computer implementations for handling boolean polynomials.

2 Theory

In this section we will discuss some properties of polynomials over $\mathbb{Z}_2$.

2.1 Preliminaries

Definition 2.1. *Let $\Bbbk$ be a field, and let $f_1, \dots, f_m$ be polynomials in $\Bbbk[x_1, \dots, x_n]$. Then*

$$V(\langle f_1, \dots, f_m\rangle) = \{(a_1, \dots, a_n) \in \Bbbk^n : f(a_1, \dots, a_n) = 0 \textit{ for all } f \in \langle f_1, \dots, f_m\rangle\}$$

*is called the **variety** of the ideal generated by $f_1, \dots, f_m$.*

Note that if

$$f_i(a_1, \dots, a_n) = 0 \text{ for all } i \in \{1, \dots, m\}, \tag{1}$$

then $(a_1, \dots, a_n) \in V(\langle f_1, \dots, f_m\rangle)$.

Since if (1) holds, then, for all $f \in \langle f_1, \dots, f_m\rangle$,

$$f(a_1, \dots, a_n) = (g_1 f_1 + \dots + g_m f_m)(a_1, \dots, a_n) = g_1 \cdot 0 + \dots + g_n \cdot 0 = 0.$$

Conversely if (1) does not hold for some $i \in \{1, \dots, m\}$, then $(a_1, \dots, a_n) \notin V(\langle f_1, \dots, f_m\rangle)$, since f_i is a function in $\langle f_1, \dots, f_m\rangle$ such that $f_i(a_1, \dots, a_n) \neq 0$.

To simplify the notation we will often write write $V(f_1,\dots,f_m)$ instead of $V(\langle f_1,\dots,f_m\rangle)$. We may also view V as a function

$$V:\mathbb{Z}_2[x_1,\dots x_n]\to\mathcal{P}(\mathbb{Z}_2^n),$$

where $\mathcal{P}(\mathbb{Z}_2^n)$ is the set of all subsets of $\mathbb{Z}_2^n$.

Definition 2.2. *Let $x_1^{a_1}\cdots x_n^{a_n}$ be a monomial in a polynomial ring $\Bbbk[x_1,\dots,x_n]$. The element $(a_1,\dots,a_n)\in\mathbb{Z}_+^n$ is called the **exponential vector** of $x_1^{a_1}\cdots x_n^{a_n}$, and $\log(x_1^{a_1}\cdots x_n^{a_n}):=(a_1,\dots,a_n)$.*

Example 2.3. Let $x_1x_3^2\in\mathbb{Z}_2[x_1,x_2,x_3]$. Then

$$\log(x_1x_3^2)=(1,0,2).$$

2.2 Boolean Polynomials

Definition 2.4. *An element of the form $x_1^{a_1}\cdots x_n^{a_n}\in\mathbb{Z}_2[x_1,\dots x_n]$, where $a_i\in\{0,1\}$, is called a **boolean monomial**.*

Definition 2.5. *An element of the form $f=m_1+\cdots+m_s\in\mathbb{Z}_2[x_1,\dots x_n]$, where m_i are **boolean monomials**, for all $i\in\{1,\dots,s\}$, is called a **boolean polynomial**.*

Definition 2.6. *For any element $x_1^{a_{1,1}}\cdots x_n^{a_{n,1}}+\cdots+x_1^{a_{1,s}}\cdots x_n^{a_{n,s}}\in\mathbb{Z}_2[x_1,\dots,x_n]$. Put*

$$\mathrm{bool}(x_1^{a_{1,1}}\cdots x_n^{a_{n,1}}+\cdots+x_1^{a_{1,s}}\cdots x_n^{a_{n,s}}):=x_1^{b_{1,1}}\cdots x_n^{b_{n,1}}+\cdots+x_1^{b_{1,s}}\cdots x_n^{b_{n,s}},$$

where $b_{i,j}=0$ if $a_{i,j}=0$ and $b_{i,j}=1$ otherwise.

Example 2.7. Let $x_1^2x_2+x_3^3\in\mathbb{Z}_2[x_1,x_2,x_3]$. Then

$$\mathrm{bool}(x_1^2x_2+x_3^3)=x_1x_2+x_3.$$

Theorem 2.8. *Let $f\in\mathbb{Z}_2[x_1,\dots x_n]$. Then $V(\mathrm{bool}(f))=V(f)$.*

Proof. Let $f=m_1+\cdots+m_s$ be a polynomial in $\mathbb{Z}_2[x_1,\dots,x_n]$. Suppose that

$$V(\mathrm{bool}(f))\neq V(f).$$

Then there must be a point $(p_1,\dots,p_n)\in\mathbb{Z}_2^n$ such that

$$\mathrm{bool}(f)(p_1,\dots,p_n)\neq f(p_1,\dots,p_n).$$

This implies that there must exist at least one monomial $m_j=x_1^{a_1}\cdots x_n^{a_n}$ in f such that

$$\mathrm{bool}(m_j)(p_1,\dots,p_n)\neq m_j(p_1,\dots,p_n).$$

This implies that there exists at least one $i \in \{1, \dots n\}$, and $a_i \geq 0$ such that

$$x_i^{a_i}(p_1, \dots, p_n) \neq x_i(p_1, \dots, p_n).$$

This is that $1^{a_i} \neq 1$ or $0^{a_i} \neq 0$ for some $a_i \geq 1$, which is impossible. Thus there cannot exist a polynomial $f \in \mathbb{Z}_2[x_1, \dots, x_n]$ such that $V(\text{bool}(f)) \neq V(f)$ $\square$

Theorem 2.9. *1. There are 2^{2^n} distinct subsets of $\mathbb{Z}_2^n$.*

2. There are 2^{2^n} distinct boolean polynomials in $\mathbb{Z}_2[x_1, \dots x_n]$

3. $V(f+g) = (V(f) \cap V(g)) \cup (V(g)^c \cap V(f)^c)$

4. $V(fg) = V(f) \cup V(g)$

This is proven in [1], Theorem 8, Theorem 6, Theorem 10 and Theorem 9 respectively.

Theorem 2.10. *The function $V : \mathbb{Z}_2[x_1, \dots x_n] \to \mathcal{P}(\mathbb{Z}_2^n)$ is onto.*

Proof. Let $X \subseteq \mathbb{Z}_2^n$ consist of one point $A = (a_1, \dots, a_n) \in \mathbb{Z}_2^n$. Then the polynomial $f_A = t_1 t_2 \dots t_n + 1 \in \mathbb{Z}_2[x_1, \dots x_n]$ where $t_i = (x_i + a_i + 1)$ has a root in $(a_1, \dots, a_n)$ but no other point. So $V(f_A) = \{A\}$. If X consists of the points $A_1, \dots, A_m$ then $X = \{A_1\} \cup \dots \cup \{A_m\} = V(f_{A_1}) \cup \dots \cup V(f_{A_m}) = V(f_{A_1} f_{A_2} \dots f_{A_m})$. $\square$

Theorem 2.11. *The function V induces a **one-to-one correspondence** between the boolean polynomials f of n variables and subsets of $\mathbb{Z}_2^n$.*

Proof. Theorem 2.8 and Theorem 2.10 implies that V is onto. Now, since there are just as many boolean polynomials in n variables as there are subsets of $\mathbb{Z}_2^n$, V must also be one-to-one. $\square$

Consider the set $\mathcal{P}(\mathbb{Z}_2^n)$ with $\mathbb{Z}_2[x_1, \dots, x_n]$ and the subsets of $\mathbb{Z}_2^n$, with addition

$$A + B := (A \cap B) \cup (A^c \cap B^c),$$

and multiplication

$$A \cdot B := A \cup B.$$

It follows from Theorem 2.10 that every $A \in \mathcal{P}(\mathbb{Z}_2^n)$ can be written as $A = V(f)$, for some $f \in \mathbb{Z}_2[x_1, \dots, x_n]$. Next, by Theorem 2.9 and the fact that $\mathbb{Z}_2[x_1, \dots, x_n]$ is a ring, we have that $\mathcal{P}(\mathbb{Z}_2^n)$ is a ring with the multiplication and addition defined above. We also have that $V : \mathbb{Z}_2[x_1, \dots, x_n] \to \mathcal{P}(\mathbb{Z}_2^n)$ is a ring homomorphism.

Further, Theorem 2.8 and Theorem 2.11 implies that the boolean polynomials, with addition as in $\mathbb{Z}_2[x_1, \dots, x_n]$ and $\text{bool}(fg)$ as multiplication, is a ring isomorphic to $\mathcal{P}(\mathbb{Z}_2^n)$, and the ring isomorphism is given by V.

Definition 2.12. *Let f and g be members of some polynomial ring $\Bbbk[x_1, \dots, x_n]$. Then we say that $g|f$, if there exists a polynomial $h \in \Bbbk[x_1, \dots, x_n]$, such that $f = gh$.*

Lemma 2.13. *Let f and g be polynomials in $\mathbb{Z}_2[x_1, \dots, x_n]$, such that $g|f$. Then $V(g) \subseteq V(f)$.*

Proof. Suppose that $g|f$, so that $f = gh$, for some $h \in \mathbb{Z}_2[x_1, \dots, x_n]$. Then $V(f) = V(gh) = V(h) \cup V(g)$, which implies $V(g) \subseteq V(f)$ □

Lemma 2.14. *Let $m = x_{i_1} \cdots x_{i_s}$ be a boolean monomial in $\mathbb{Z}_2[x_1, \dots, x_n]$. Then $m(a_1, \dots, a_n) = 1$ if and only if $m|x_1^{a_1} \cdots x_n^{a_n}$.*

Proof. Let $f = x_1^{a_1} \cdots x_n^{a_n}$ such that $m|f$. Then, by Lemma 2.13,

$$V(m) \subseteq V(f). \tag{2}$$

We also have that

$$f(a_1, \dots, a_n) = \prod_{a_j = 1} a_j = 1. \tag{3}$$

Now (2) and (3) implies that

$$m(a_1, \dots, a_n) = 1. \tag{4}$$

Conversely choose $(a_1, \dots, a_n)$, such that $m(a_1, \dots, a_n) = 1$. Then $a_{i_k} = 1$ for all $k \in \{1, \dots, s\}$. This implies that

$$m|x_1^{a_1} \cdots x_n^{a_n}. \tag{5}$$

□

Definition 2.15. *The function S_i from a set of points $X = \{p_1, \dots, p_s\}$ to $\{0, 1\}$ such that $S_i(p_i) = 1$ and $S_i(p_j) = 0$ if $i \neq j$. is called the **separator** of p_i with respect to X*

Proposition 2.16. *The separator for a point $A = (a_1, \dots, a_n) \in \mathbb{Z}_2^n$, with respect to $\mathbb{Z}_2^n$, is a polynomial function, where the polynomial equals to the sum of all boolean monomials $m \in \langle x_1^{a_1} \cdots x_n^{a_n} \rangle \subseteq \mathbb{Z}_2[x_1, \dots x_n]$.*

Proof. Let $A = (a_1, \cdots a_n) \in \mathbb{Z}_2^n$, and put $f = (x_1 + a_1 + 1) \cdots (x_n + a_n + 1)$. It is easy to verify that $f(x_1, \dots, x_n) = 1$ if and only if $(x_1, \dots, x_n) = (a_1, \dots, a_n)$. Hence $f = S_A$. We see that

$$f = \sum_{(p_1, \dots, p_n) \in \mathbb{Z}_2^n} x_1^{p_1} \cdots x_n^{p_n} (a_1 + 1)^{1 - p_1} \cdots (a_n + 1)^{1 - p_n},$$

so $x_1^{p_1} \cdots x_n^{p_n}$ is a term of f if and only if $(a_1 + 1)^{1 - p_1} \cdots (a_n + 1)^{1 - p_n} = 1$. This is if and only if $a_i = 0$ whenever $p_i = 0$, which is if and only if $x_1^{p_1} \cdots x_n^{p_n} \in \langle x_1^{a_1} \cdots x_n^{a_n} \rangle$. □

This is also proved in [2].

Example 2.17. The separator S_P for the point $P = (1, 0, 1)$ in $\mathbb{Z}_2^3$ is the sum of all boolean monomials in the ideal $\langle x_1x_3 \rangle$. Those are x_1x_3 and $x_1x_2x_3$. So $S_P = x_1x_3 + x_1x_2x_3$.

Definition 2.18. *Let A and B be two sets. Then $A \setminus B = \{x \in A : x \notin B\}$.*

Proposition 2.19. *If $f, g \in \mathbb{Z}_2[x_1, \ldots, x_n]$, then $V(f) \setminus V(g) = V(fg+g+1)$.*

Proof. Put $P \in \mathbb{Z}_2^n$, so that $P \notin V(f)$. Then

$$(fg+g+1)(P) = f(P)g(P)+g(P)+1 = 1 \cdot g(P)+g(P)+1 = 1 \Rightarrow P \notin V(fg+g+1). \tag{6}$$

Next set P, so that $P \in V(g)$, then

$$(fg+g+1)(P) = f(P)g(P)+g(P)+1 = f(P) \cdot 0+0+1 = 1 \Rightarrow P \notin V(fg+g+1). \tag{7}$$

Finally set P, so that $P \in V(f)$ and $P \notin V(g)$. Then

$$(fg + g + 1)(P) = 0 \cdot 1 + 1 + 1 = 0 \Rightarrow P \in V(fg + g + 1). \tag{8}$$

Now (6), (7) and (8) implies that $V(f) \setminus V(g) = V(fg + g + 1)$ □

Theorem 2.20. *Let $f \in \mathbb{Z}_2[x_1, \ldots x_n]$ be a boolean polynomial and let $P = (p_1, \ldots, p_n) \in \mathbb{Z}_2^n$. Then*

$$P \in V(f)$$

if and only if f contains an even number of monomials $x_{i_1} \cdots x_{i_s}$, such that

$$x_{i_1} \cdots x_{i_s} | x_1^{p_1} \cdots x_n^{p_n}$$

.

Proof. It follows from Lemma 2.14, that if f contains m monomials, such that $x_{i_1} \cdots x_{i_s} | x_1^{p_1} \cdots x_n^{p_n}$. Then $f(p_1, \ldots, p_n) = \sum_{i=1}^{m} 1$, which is 0 if m is even and 1 if m is odd. □

2.3 Monomial orders

In this section we introduce the concept of a monomial order.

Definition 2.21. *A relation $\prec$ between the monomials of a polynomials ring $\Bbbk[x_1, \ldots, x_n]$ is said to be a **monomial order** if for any monomials $m_1, m_2, m_3 \in \Bbbk[x_1, \ldots, x_n]$,*

1. *either $m_1 \prec m_2$, $m_2 \prec m_1$ or $m_1 = m_2$.*

2. *if $m_1 \prec m_2$ and $m_2 \prec m_3$, then $m_1 \prec m_3$.*

3. *if* $m_1 \neq 1$*, then* $1 \prec m_1$*.*

4. *if* $m_1 \prec m_2$*, then* $m_3 m_1 \prec m_3 m_2$*.*

Example 2.22. We have that $\prec_{lex}$, by $x_1^{a_1} \cdots x_n^{a_n} \prec_{lex} x_1^{b_1} \cdots x_n^{b_n}$ iff $\min_{a_i<b_i} i < \min_{b_i<a_i} i$, is a monomial order, since

1. if $x_1^{a_1} \cdots x_n^{a_n} \neq x_1^{b_1} \cdots x_n^{b_n}$, then $\min_{a_i<b_i} i < \min_{b_i<a_i} i$ or $\min_{b_i<a_i} i < \min_{a_i<b_i} i$, and if $x_1^{a_1} \cdots x_n^{a_n} = x_1^{b_1} \cdots x_n^{b_n}$, then neither $\min_{a_i<b_i} i < \min_{b_i<a_i} i$ or $\min_{b_i<a_i} i < \min_{a_i<b_i} i$.

2. if $\min_{a_i<b_i} i < \min_{b_i<a_i} i$ and $\min_{b_i<c_i} i < \min_{c_i<b_i} i$, then $\min_{a_i<c_i} i < \min_{c_i<a_i} i$.

3. if $x_1^{a_1} \cdots x_n^{a_n} \neq 1$, then $\min_{0<a_i} i < \min_{a_i<0} i$.

4. if $\min_{a_i<b_i} i < \min_{b_i<a_i} i$, then $\min_{a_i+c_i<b_i+c_i} i < \min_{b_i+c_i<a_i+c_i} i$.

Proposition 2.23. *There exist no monomial order* $\prec$ *such that* $\mathrm{bool}(v) \prec \mathrm{bool}(w) \Rightarrow \mathrm{bool}(uv) \prec \mathrm{bool}(uw)$*, where* u, v, w *are boolean monomials in* $\mathbb{Z}_2[x_1, \ldots, x_n]$*.*

Proof. Let a, b be boolean monomials such that $\mathrm{bool}(a) \prec \mathrm{bool}(b)$, and suppose that $\mathrm{bool}(a) \prec \mathrm{bool}(b)$, and put $c = ab$. Then, if

$$\mathrm{bool}(v) \prec \mathrm{bool}(w) \Rightarrow \mathrm{bool}(uv) \prec \mathrm{bool}(uw).$$

Then

$$\mathrm{bool}((ab)a) \prec \mathrm{bool}((ab)b) \Rightarrow \mathrm{bool}(ab) \prec \mathrm{bool}(ab),$$

which we do not allow. □

2.4 Boolean formulas as boolean polynomials

Definition 2.24. *A* ***boolean formula*** *of* n *variables is a function* $\phi : \{true, false\}^n \to \{true, false\}$ *which consists of either*

1. *a single variable,* $\phi = \psi_i$*, then* $\phi(\psi_1, \ldots, \psi_n) = true \Leftrightarrow \psi_i = true$*.*

2. *a conjunction of two boolean formulas,* $\phi = \varphi_1 \wedge \varphi_2$*, then* $\phi(\psi_1, \ldots, \psi_n) = true \Leftrightarrow \varphi_1(\psi_1, \ldots, \psi_n) = true$ *and* $\varphi_2(\psi_1, \ldots, \psi_n) = true$*.*

3. *a disjunction of two boolean formulas,* $\phi = \varphi_1 \vee \varphi_2$*, then* $\phi(\psi_1, \ldots, \psi_n) = true \Leftrightarrow \varphi_1(\psi_1, \ldots, \psi_n) = true$ *or* $\varphi_2(\psi_1, \ldots, \psi_n) = true$*.*

4. *a negation of a boolean formula,*$\phi = \neg\varphi$*, then* $\phi(\psi_1, \ldots, \psi_n) = true \Leftrightarrow \varphi(\psi_1, \ldots, \psi_n) = false$*.*

Definition 2.25. *A boolean* $\phi(\psi_1, \ldots, \psi_n)$ *is called* ***satisfiable*** *if there exists* $(\psi_1, \ldots, \psi_n) \in \{true, false\}^n$ *such that* $\phi(\psi_1, \ldots, \psi_n) = true$*.*

Let $\phi(\psi_1, \ldots, \psi_n)$ be a boolean formula, and let $a : \{true, false\} \to \{0, 1\}$, be a one-to-one correspondence. Then, by Theorem 2.11, there exists a unique boolean polynomial $f \in \mathbb{Z}_2[x_1, \ldots, x_n]$ such that

$$\phi(\psi_1, \ldots, \psi_n) \Leftrightarrow f(a(\psi_1), \ldots, a(\psi_n)) = a(\phi(\psi_1, \ldots, \psi_n)).$$

From this point on, we will only care about what our polynomials evaluate to, thus we will write $f = g$ if $\mathrm{bool}(f) = \mathrm{bool}(g)$, for any polynomials $f, g \in \mathbb{Z}_2[x_1, \ldots, x_n]$.

Definition 2.26. *For each boolean formula $\phi(\psi_1, \ldots, \psi_n)$, let $T_0(\phi)(x_1, \ldots, x_n)$ be the boolean polynomial such that*

$$\phi(\psi_1, \ldots, \psi_n) = \mathit{true} \Leftrightarrow T_0(\phi)(x_1, \ldots, x_n) = 0,$$

where $\psi_i = \mathit{true} \Leftrightarrow x_i = 0$.

Conversely, let $T_1(\phi)(x_1, \ldots, x_n)$ be the boolean polynomial such that

$$\phi(\psi_1, \ldots, \psi_n) = \mathit{true} \Leftrightarrow T_1(\phi)(x_1, \ldots, x_n) = 1,$$

where $\psi_i = \mathit{true} \Leftrightarrow x_i = 1$.

Theorem 2.27. *Let ϕ be a boolean formula. Then*

1. *if ϕ consists of a single variable, ψ_i, then $T_0(\phi) = x_i$.*
2. *if ϕ consists of a negation, $\phi = \neg\varphi$, then $T_0(\phi) = 1 + T_0(\varphi)$.*
3. *if ϕ consists of a conjunction $\phi = \varphi_1 \wedge \varphi_2$, then $T_0(\phi) = T_0(\varphi_1) + T_0(\varphi_2) + T_0(\varphi_1)T_0(\varphi_2)$.*
4. *if ϕ consists of a disjunction $\phi = \varphi_1 \vee \varphi_2$, then $T_0(\phi) = T_0(\varphi_1)T_0(\varphi_2)$.*

A proof of this can be found in [6] **Theorem 3.1**.

Lemma 2.28. *Let ϕ be a boolean formula. Then*

$$T_1(\phi)(x_1, \ldots, x_n) = 1 + T_0(\phi)(x_1 + 1, \ldots, x_n + 1).$$

Proof. It is clear that if

$$T_1(\phi)(x_1, \ldots, x_n) = 1,$$

then

$$T_0(\phi)(x_1 + 1, \ldots, x_n + 1) = 0.$$

Else, if

$$T_1(\phi)(x_1, \ldots, x_n) = 0,$$

then

$$T_0(\phi)(x_1 + 1, \ldots, x_n + 1) = 1.$$

Thus

$$T_1(\phi)(x_1, \ldots, x_n) = 1 + T_0(\phi)(x_1 + 1, \ldots, x_n + 1).$$

□

Theorem 2.29. *Let ϕ be a boolean formula. Then*

1. *if ϕ consists of a single variable, ψ_i, then $T_1(\phi) = x_i$.*
2. *if ϕ consists of a negation, $\phi = \neg\varphi$, then $T_1(\phi) = 1 + T_1(\varphi)$.*
3. *if ϕ consists of a conjunction, $\phi = \varphi_1 \vee \varphi_2$, then $T_1(\phi) = T_1(\varphi_1)T_1(\varphi_2)$.*
4. *if ϕ consists of a disjunction, $\phi = \varphi_1 \wedge \varphi_2$, then $T_1(\phi) = T_1(\varphi_1) + T_1(\varphi_2) + T_1(\varphi_1)T_1(\varphi_2)$.*

Proof. By Lemma 2.28 and Theorem 2.27. If ϕ consists of a single variable, ψ_i, then $T_1(\phi)(x_1, \ldots, x_n) = 1 + T_0(\phi)(x_1 + 1, \ldots, x_n + 1) = 1 + x_i + 1 = x_i$.

If ϕ consists of a negation, $\phi = \neg\varphi$.

$$T_1(\phi)(x_1, \ldots, x_n) = 1 + T_0(\phi)(x_1 + 1, \ldots, x_n + 1) =$$
$$T_0(\varphi)(x_1 + 1, \ldots, x_n + 1) = 1 + T_1(\varphi)(x_1, \ldots, x_n).$$

If ϕ consists of a conjunction, $\phi = \varphi_1 \wedge \varphi_2$, then

$$1 + T_0(\phi)(x_1 + 1, \ldots, x_n + 1) =$$
$$(1 + T_0(\varphi_1) + T_0(\varphi_2) + T_0(\varphi_1)T_0(\varphi_2))(x_1 + 1, \ldots x_n + 1) =$$
$$1 + (1 + T_1(\varphi_1)) + (1 + T_1(\varphi_2)) + (1 + T_1(\varphi_1))(1 + T_1(\varphi_2)) =$$
$$T_1(\varphi_1)T_1(\varphi_2).$$

If ϕ consists of a disjunction, $\phi = \varphi_1 \vee \varphi_2$, then

$$1 + T_0(\phi)(x_1 + 1, \ldots x_n + 1) = 1 + (T_0(\varphi_1)T_0(\varphi_2))(x_1 + 1, \ldots x_n + 1) =$$
$$1 + (T_0(\varphi_1)(x_1 + 1, \ldots x_n + 1)T_0(\varphi_2)(x_1 + 1, \ldots x_n + 1)) =$$
$$1 + (1 + T_1(\varphi_1)(x_1, \ldots x_n))(1 + T_1(\varphi_2)(x_1, \ldots x_n)) =$$
$$T_1(\varphi_1) + T_1(\varphi_2) + T_1(\varphi_1)T_1(\varphi_2).$$

□

Example 2.30. Let $\phi(\psi_1, \psi_2, \psi_3) = (\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_3)$. Then

$$T_1(\phi) = T_1((\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_3)) =$$
$$= T_1(\psi_1 \vee \psi_2)T_1(\neg\psi_1 \vee \psi_3) =$$
$$= (T_1(\psi_1) + T_1(\psi_2) + T_1(\psi_1)T_1(\psi_2))(T_1(\neg\psi_1) + T_1(\psi_3) + T_1(\neg\psi_1)T_1(\psi_3)) =$$
$$= (x_1 + x_2 + x_1x_2)(1 + x_1 + x_3 + (1 + x_1)x_3) =$$
$$= (x_1 + x_2 + x_1x_2)(1 + x_1 + x_1x_3) =$$
$$= x_1 + x_1 + x_1x_3 + x_2 + x_1x_2 + x_1x_2x_3 + x_1x_2 + x_1x_2 + x_1x_2x_3 =$$
$$= x_2 + x_1x_2 + x_1x_3,$$

and

$$
\begin{aligned}
T_0(\phi) = T_0((\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_3)) = \\
= T_0(\psi_1 \vee \psi_2) + T_0(\neg\psi_1 \vee \psi_3) + T_0(\psi_1 \vee \psi_2)T_0(\neg\psi_1 \vee \psi_3) = \\
= T_0(\psi_1)T_0(\psi_2) + T_0(\neg\psi_1)T_0(\psi_3) + T_0(\psi_1)T_0(\psi_2)T_0(\neg\psi_1)T_0(\psi_3) = \\
= x_1x_2 + (1+x_1)x_3 + x_1x_2(1+x_1)x_3 = \\
= x_1x_2 + x_3 + x_1x_3 + x_1x_2x_3 + x_1^2x_2x_3 = \\
= x_3 + x_1x_2 + x_1x_3.
\end{aligned}
$$

Definition 2.31. *A boolean formula which consists of conjunction of clauses*

$$\phi(\psi_1, \ldots, \psi_n) = C_1 \wedge C_2 \wedge \ldots \wedge C_s,$$

where each clause C_t consists of disjunctions of at most k literals

$$C_t = l_1 \vee l_2 \vee \ldots \vee l_k,$$

where each literal l_i is either a single variable $l_i = \psi_j$, or a negation of a variable $l_i = \neg\psi_j$, is said to be a ***k-CNF formula****. The problem of finding a solution to a k-CNF formula is called k-CNF-SAT.*

It is well known that k-CNF-SAT is NP-complete for $k \geq 3$ and P for $k < 3$.

Theorem 2.32. *Given $f = f_1 \cdots f_s \in \mathbb{Z}_2[x_1, \ldots, x_n]$, where, for each $i \in \{1, \ldots, s\}$, f_i is a boolean polynomial that contains at most k distinct variables, for some $k \geq 3$. Then, the problem of finding a point $P \in \mathbb{Z}_2^n$ such that $P \notin V(f)$ is NP-complete.*

Proof. Suppose that we have a point $P \notin V(f_1 \cdots f_s)$. Then this can be verified by checking $P \notin V(f_i)$ for each $i \in \{1, \ldots, s\}$. Since f_i only contains k distinct variables, f_i contains at most 2^k monomials. Since k does not depend on the size of the input, there must be a constant upper bound B on the time it takes to check if $P \notin V(f_i)$. Thus, we have that the time it takes to verify that a given solution is correct can be bounded by $s \cdot B$, where B is a constant and s is the number of polynomials in our product. This implies that our problem is in NP.

Next, let $C_1 \wedge C_2 \wedge \ldots \wedge C_s$ be a 3-CNF formula of n variables and s clauses. Then

$$T_1(C_1 \wedge C_2 \wedge \ldots \wedge C_s) = T_1(C_1)T_1(C_2) \cdots T_1(C_s)$$

and since C_i contains at most 3 variables, so does the polynomial $T_1(C_i)$. This implies that $T_1(C_1)T_1(C_2) \cdots T_1(C_s)$ satisfies the restrictions of our problem. To complete the transformation, we have to set $f_i = T_1(C_i)$ for each $i \in \{1, \ldots, s\}$. This can be done in polynomial time, since the number of monomials in $T_1(C_i)$ does not depend on the number of variables in $C_1 \wedge C_2 \wedge \ldots \wedge C_s$. This implies that our problem is NP-complete. $\square$

3 Implementations

In this section we will discuss implementations of boolean polynomials.

3.1 Addition and Multiplication of Boolean Polynomials

Let each boolean monomial $x_1^{b_1} \cdots x_n^{b_n}$ be represented by its exponential vector $b_1, \ldots, b_n$, and let a boolean polynomial $f = m_1 + \cdots + m_s$ be a *list* of boolean monomials. To be able to do elementary operations, such as checking if $f = g$ with reasonable effort, we should keep the polynomials sorted, so that $i < j \Rightarrow m_i \prec m_j$, for some monomial order $\prec$.

Algorithm 1 will act as addition of two sorted polynomials.

Algorithm 1 Addition of two boolean polynomials

Input: Two sorted polynomials $f = m_{f,1} + \ldots + m_{f,s}$ and $g = m_{g,1} + \ldots + m_{g,t}$.

Output: A sorted polynomial $h = f + g$

```
function ADD(f, g)
    sum ← 0
    i ← 1
    j ← 1
    while i ≠ s ∧ j ≠ t do
        if m_{f,i} ≺ m_{g,j} then
            sum ← sum + m_{f,i}
            i++
        end if
        if m_{g,j} ≺ m_{f,i} then
            sum ← sum + m_{g,j}
            j++
        end if
        if m_{f,i} = m_{g,j} then
            i++
            j++
        end if
    end while
    while i ≠ s do
        sum ← sum + m_{f,i}
        i++
    end while
    while j ≠ t do
        sum ← sum + m_{g,j}
        j++
    end while
end function
```

We will divide our multiplication algorithm into three different functions. One for multiplication of two boolean monomials, one for multiplication between a boolean monomial and a boolean polynomial, and finally one for multiplication of two boolean polynomials.

Algorithm 2 will act as multiplication of two boolean monomials.

Algorithm 2 multiplication of two boolean monomials

Input: Two boolean monomials $m_1 = x_1^{a_1} \cdots x_n^{a_n}$ and $m_2 = x_1^{b_1} \cdots x_n^{b_n}$.
Output: A boolean monomial $m_1 m_2$
  function MUL-MON-MON(m_1, m_2)
    return $x_1^{max(a_1,b_1)} \cdots x_n^{max(a_n,b_n)}$
  end function

Before we create the algorithm for multiplication between a boolean monomial u and a boolean polynomial $f = m_1 + \ldots, +m_s$, we should note that Proposition 2.23 implies that just using Algorithm 2, for every monomial m_i, $i \in \{1, \ldots, s\}$ is not guaranteed to return a sorted polynomial.

We will consider two algorithms for multiplying a boolean monomial with a boolean polynomial. The first is Algorithm 3, where we use the fact that

$$u(m_1 + \ldots + m_s) = u(m_1 + \ldots + m_{floor(s/2)}) + u(m_{\text{floor}(s/2)+1} + \ldots + m_s).$$

If both $u(m_1 + \ldots + m_{\text{floor}(s/2)})$ and $u(m_{\text{floor}(s/2)+1} + \ldots + m_s)$ are sorted, then we can use Algorithm 1 to tie them together.

Algorithm 3 multiplication of a boolean monomial and a boolean polynomial.

Input: A boolean monomial m and a boolean polynomial $f = m_1 + \ldots + m_s$
Output: A sorted boolean polynomial mf
  function MUL-MON-POL(m, f)
    if $f = m_1$ **then**
  return MUL-MON-MON(m, m_1)
    else
      $f_1 \leftarrow m_1 + \ldots + m_{\text{floor}(s/2)}$
      $f_2 \leftarrow m_{\text{floor}(s/2)+1)} + \ldots + m_s$
      return ADD(MUL-MON-POL(m, f_1), MUL-MON-POL(m, f_2))
    end if
  end function

Note that Algorithm 3 does not require f to be sorted. However it will always return a sorted polynomial.

Our other way to perform multiplication of boolean polynomial f with a boolean monomial $x_{i_1} \cdots x_{i_s}$ is Algorithm 4. To see that this returns a sorted polynomial, consider the following lemma.

Lemma 3.1. *Let m_1, m_2 be boolean monomials, $m_1 \prec m_2$. If $x_i | m_1$ and $x_i | m_2$, or if if $x_i \nmid m_1$ and $x_i \nmid m_2$, then*

$$\text{bool}(x_i m_1) \prec \text{bool}(x_i m_2).$$

Proof. If $x_i | m_1$ and $x_i | m_2$, then $\text{bool}(x_i m_1) = m_1 \prec m_2 = \text{bool}(x_i m_2)$. If $x_i \nmid m_1$ and $x_i \nmid m_2$, then $\text{bool}(x_i m_1) = x_1 m_1 \prec x_i m_2 = \text{bool}(x_i m_2)$ □

Lemma 3.1 implies that f_1 and f_2 in Algorithm 4 will be sorted polynomials.

Algorithm 4 multiplication of a boolean monomial and a boolean polynomial.

Input: A boolean monomial m and a boolean sorted polynomial $f = m_1 + \ldots + m_s$
Output: A sorted boolean polynomial mf

```
function MUL-MON-POL(m, f)
    for all x_i do
        if x_i|m then
            for all m_j do
                if x_i|m_j then
                    f_1 ← f_1 + m_j
                else
                    f_2 ← f_2 + x_i · m_j
                end if
                f ← ADD(f_1, f_2);
                f_1 ← 0;
                f_2 ← 0;
            end for
        end if
    end for
end function
```

A comparison between Algorithm 3 and Algorithm 4 is made in Section 3.3.1.

Finally, for our multiplication of two boolean polynomials, we will use the same trick as in Algorithm 3. This may be implemented as in Algorithm 5.

3.2 DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is widely used in SAT-solvers [4]. In this section we will adapt the DPLL procedure to determine if a product of boolean polynomials $f_1 \cdots f_s$ evaluates to zero everywhere without actually evaluating the product. We will write $f_1 \cdots f_s = 0$, if $f_1 \cdots f_s$ evaluates to zero everywhere.

Algorithm 5 multiplication between two boolean polynomials.

Input: Two sorted boolean polynomials $f = m_{f,1} + \dots m_{f,s}$ and $g = m_{g,1} + \dots + m_{g,t}$

Output: A sorted boolean polynomial $h = fg$

```
function MUL-POL-POL(f, g)
    if f = m_{f,1} then
        return MUL-MON-POL(m_{f,1}, g)
    else
        f_1 ← m_{f,1} + ... m_{f,(floor(s/2))}
        f_2 ← m_{f,(floor(s/2)+1)} + ... m_{f,s}
        return ADD(MUL-POL-POL(f_1, g), MUL-POL-POL(f_2, g))
    end if
end function
```

Definition 3.2. *Let $f \in \mathbb{Z}_2[x_1, \dots, x_n]$ be a boolean polynomial. Then*

$$d_i^0(f) = f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n),$$

and

$$d_i^1(f) = f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n).$$

Example 3.3. Let $f = x_1 + x_2 + x_1x_3$, then

$$d_3^0 = x_1 + x_2 + x_1 0 = x_1 + x_2$$

$$d_3^1 = x_1 + x_2 + x_1 1 = x_2$$

Proposition 3.4. *Let $f = f_1 f_2 \cdots f_s$, where $f_1, f_2, \dots, f_s$ are arbitrary boolean polynomials in $\mathbb{Z}_2[x_1, \dots, x_n]$. Then*

1. *If f_i contains the monomial "1" for each $i \in \{1, \dots, s\}$, then $f \neq 0$.*
2. *If $f_i = 0$ for some $i \in \{1, \dots, s\}$, then $f = 0$.*
3. *$f \neq 0$ if and only if $d_i^0(f) \neq 0$ or $d_i^1(f) \neq 0$, for any $i \in \{1, \dots, n\}$.*

Proof. Suppose that f_i contains the monomial "1" for each $i \in \{1, \dots, s\}$. Then

$$f_1 f_2 \cdots f_s(0, \dots, 0) = 1 \cdot 1 \cdots 1 = 1,$$

which implies (1).

(2) is obvious.

Next suppose that $f \neq 0$. Then, for any $i \in \{1, \dots, n\}$, there exists a point $(p_1, \dots, p_{i-1}, p_i, p_{i+1}, \dots, p_n) \in \mathbb{Z}_2^n$, such that

$$f(p_1, \dots, p_{i-1}, p_i, p_{i+1}, \dots, p_n) = 1.$$

Since p_i is either equal to 0 or 1, either $f(p_1, \ldots, p_{i-1}, 1, p_{i+1}, \ldots, p_n) = 1$ or $f(p_1, \ldots, p_{i-1}, 0, p_{i+1}, \ldots, p_n) = 1$. If $f = 0$ then it is obvious that

$$d_i^0(f) = 0$$

and

$$d_i^1(f) = 0$$

for any $i \in \{1, \ldots, n\}$. Thus (3). □

We see that it is possible to determine if a product of boolean polynomials $f_1 \cdots f_s$ evaluates to zero everywhere by using Proposition 3.4.

Example 3.5. Let $f_1 f_2 = (x_1 + x_1 x_2)(x_2)$. Then, by Proposition 3.4

$$\begin{gathered}
f_1 f_2 \neq 0 \Leftrightarrow \\
(x_1 + x_1 x_2)(x_2) \neq 0 \Leftrightarrow \\
d_1^1((x_1 + x_1 x_2)(x_2)) \neq 0 \text{ or } d_1^0((x_1 + x_1 x_2)(x_2)) \neq 0 \Leftrightarrow \\
(1 + 1 \cdot x_2)(x_2) \neq 0 \text{ or } (0)(x_2) \neq 0 \Leftrightarrow \\
(d_2^1(1 + 1 \cdot x_2)(x_2) \neq 0 \text{ or } d_2^0(1 + 1 \cdot x_2)(x_2)) \text{ or } (0)(x_2) \neq 0 \Leftrightarrow \\
((1 + 1)(1) \neq 0 \text{ or } d_2^0(x_2)(0) \neq 0) \text{ or } (0)(x_2) \neq 0 \Leftrightarrow \\
(0)(x_2) \neq 0 \Leftrightarrow \\
0 \neq 0,
\end{gathered}$$

which implies that $f_1 f_2 = 0$.

If we make an algorithm out of this, then we get the DPLL procedure.

The "X" in Algorithm 6 refers to a few optional lines which may reduce the search tree, and "choose-literal" is a function which decides the branching variable.

This will be discussed in Section 3.2.2 and Section 3.2.3, respectively

3.2.1 How to perform DPLL for boolean polynomials efficiently

In this section we will discuss implementation of boolean polynomials so that the Algorithm 6 runs smoothly.

We will limit ourselves to the case $f = f_1 \cdots f_s \in \mathbb{Z}_2[x_1, \ldots, x_n]$ where

$$f_i \in \mathbb{Z}_2[x_{i_1}, \ldots, x_{i_k}] \subseteq \mathbb{Z}_2[x_1, \ldots, x_n]$$

for each $i \in \{1, \ldots, s\}$, and k is so low so that 2^k bits is a manageable amount of memory. By Theorem 2.32, Algorithm 6 solves a NP-complete problem if $k \geq 3$.

Now, instead of letting each boolean monomial be represented by a bitvector, let $f_i \in \mathbb{Z}_2[x_{i_1}, \ldots, x_{i_k}]$ be represented by the k integers $(i_1, \ldots, i_k)$, and the coefficient vector $f_i[] = (f_i[1], f_i[2], \ldots, f_i[2^k])$, so that

Algorithm 6 DPLL for boolean polynomials.

Input: A list of polynomials $f_1, \ldots, f_s$
Output: $true$ if $f_1 \cdots f_s \neq 0$, $false$ if $f_1 \cdots f_s = 0$

```
function DPLL(f_1, ..., f_s)
    f_1, ..., f_s ← X(f_1, ..., f_s)
    for all f_i do
        if f_i does not contain 1 then
            not0 ← true
        end if
        if f_i = 0 then
            return false
        end if
    end for
    if not0 then
        return true
    else
        i=choose-literal(f_1, ..., f_s)
        return DPLL(d_i^1(f_1, ..., f_s)) or DPLL(d_i^0(f_1, ..., f_s))
    end if
end function
```

$$f_i = \sum_{(a_1,\ldots,a_k)\in\{0,1\}^k} f_i[a_1 2^0 + \ldots + a_k 2^{k-1}] x_{i_1}^{a_1} \cdots x_{i_k}^{a_k}.$$

Example 3.6. Let $f_i(x_3, x_5, x_8) = 1+x_3+x_8+x_3x_5x_8$, then f_i is represented by

$$(i_1, i_2, i_3) = (3, 5, 8),$$

and

$$(f_i[1], \ldots, f_i[2^3]) = (1, 1, 0, 0, 1, 0, 0, 1).$$

Since $f_i[t]$ is supposed to represent an element in $\mathbb{Z}_2$, we only need 1 bit for each $f_i[t]$, $t \in \{1, \ldots, 2^k\}$. Thus, $f_i[]$ becomes a bitvector of 2^k bits.

Definition 3.7. *For two bitvectors $f[], g[]$ of equal size.*

1. *Let $f[] \wedge g[]$ be "and" for each bit.*
2. *Let $f[] + g[]$ be "xor" for each bit.*
3. *Let $\neg f[]$ be the complement of $f[]$ ("not" for each bit).*
4. *For $j \in \{0, \ldots, 2^k - 1\}$ Let $f[]/j$ be $f[i]/j = f[i + j]$, for every $i \in \{0, \ldots, 2^k - j - 1\}$.*

In order to get good performance, we should use a data type for $f_i[]$ which allows us to perform the operations in Definition 3.7 quickly. We will also define bitvectors $h[]$ such that

$$\sum_{(a_1,\dots,a_k)\in\{0,1\}^k} h[a_1 2^0 + \ldots + a_k 2^{k-1}] x_{i_1}^{a_1} \cdots x_{i_k}^{a_k}$$

becomes useful polynomials.

Proposition 3.8. *For each $t \in \{1, \ldots, k\}$, let, $h_t[]$ be the bitvector such that*

$$\sum_{(a_1,\dots,a_k)\in\{0,1\}^k} h_t[a_1 2^0 + \ldots + a_k 2^{k-1}] x_{i_1}^{a_1} \cdots x_{i_k}^{a_k}$$

is the polynomial that contains all monomials m such that $x_{i_t}|m$.

Then for each polynomial $f_i \in \mathbb{Z}_2[x_{i_1}, \ldots, x_{i_k}]$,

$$d_{i_t}^0(f_i)[] = f_i[] \wedge \neg h_t[], \tag{9}$$

and

$$d_{i_t}^1(f_i)[] = (f_i[] \wedge \neg h_t[]) + (f_i[] \wedge (h_t[]/(2^t))). \tag{10}$$

Proof. A monomial m exists in $d_{i_t}^0(f_i)$ if and only if m exists in f_i and $x_{i_t} \nmid m$, thus (9).

Next, a monomial m exists in $d_{i_t}^1(f_i)$ if m or $x_{i_t}m$ exists in f_i but not both m and $x_{i_t}m$, thus (10). □

3.2.2 Reductions

If we somehow know that $V(d_i^1(f_1, \ldots, f_s)) \subseteq V(d_i^0(f_1, \ldots, f_s))$, then it is safe to let $f_1 \cdots f_s \leftarrow d_i^0(f_1 \cdots f_s)$ before we choose literal.

The original DPLL uses three rules to speed up the search [4].

1. Unit Propagation: If a clause C_t only contains one literal ϕ_i, then it is safe to assign ϕ_i such that C_t is satisfied.

   For boolean polynomials, we may interpret this as if $d_j^0(f_i) = 0$ for some $j \in \{i_1, \ldots, i_k\}$, then let $f_1, \ldots, f_s \leftarrow d_j^1(f_1, \ldots, f_s)$. Conversely if $d_j^1(f_i) = 0$, then let $f_1, \ldots, f_s \leftarrow d_j^0(f_1, \ldots, f_s)$.

2. Monotone Literals: If a literal ϕ_i appears in some clause but $\neg\phi_i$ does not appear in any clause, then ϕ_i may be assigned to $true$. Conversely if $\neg\phi_i$ appears in some clause but ϕ_i does not appear, then ϕ_i may be assigned to $false$.

   For boolean polynomials we could interpret this as if, for some $i \in [1, \ldots, n]$,

   $$d_i^1(f_t) | d_i^0(f_t),$$

for all t, then we have that $V(d_i^1(f_1,\ldots,f_s)) \subseteq V(d_i^0(f_1,\ldots,f_s))$, so we may put

$$f_1,\ldots,f_s \leftarrow d_i^0(f_1,\ldots,f_s).$$

Conversely, if

$$d_i^0(f_t)|d_i^1(f_t)$$

for all t, we may put

$$f_1,\ldots,f_s \leftarrow d_i^1(f_1,\ldots,f_s).$$

In the special case, where $f_1,\ldots,f_s = T_1(C_1),\ldots,T_1(C_s)$, for some 3-CNF formula $C_1 \wedge \ldots \wedge C_s$, we have that

$$d_i^{0,1}(f_t)|d_i^{1,0}(f_t)$$

for all t, if and only if ϕ_i is a monotone literal of $C_1 \wedge \ldots \wedge C_s$. However, we will not implement this in this paper.

3. Clause Submission: If a clause C_t is a subset of another clause C_u, then it is safe to remove C_t.
 This is not used in modern implementations of DPLL [4], so we will not consider this.

In this paper, we will test Algorithm 6 with an empty X and with X as in Algorithm 7, which is 1 until we get $f_1 \cdots f_s = 0$ or $d_j^{1,0}(f_i) \neq 0$ for all $i \in \{1,\ldots,s\}$, $j \in \{i_1,\ldots,i_k]$.

3.2.3 Choose literal

In this section we will discuss the function "choose-literal()" in Algorithm 6. This is a huge part of the DPLL algorithm.

Example 3.9. Consider $f_1f_2 = (x_1+x_1x_2)(x_2)$. In Example 3.5, we showed that $f_1f_2 = 0$. However, if we decided to assign a value to x_2 first, then

$$\begin{gathered}f_1f_2 \neq 0 \Leftrightarrow \\ d_2^0(f_1f_2) \neq 0 \text{ or } d_2^1(f_1f_2) \Leftrightarrow \\ (x_1)0 \neq 0 \text{ or } (x_1+x_1)1 \neq 0 \Leftrightarrow \\ 0 \neq 0\end{gathered}$$

which is much better.

In this paper, we will test two different tactics for choosing literal. The first is to just choose the first literal we can find in the polynomial $\min\limits_{i;f_i\neq 1} f_i$. This can be implemented as in Algorithm 8.

Algorithm 7 X

Input: A list of polynomials $f_1, \ldots, f_s$

Output: A list of polynomials $g_1, \ldots, g_s$ such that $g_1 \cdots g_s \neq 0 \Leftrightarrow f_1 \cdots f_s \neq 0$

```
function X(f_1, ..., f_s)
    while ¬ done do
        done ← true
        for all f_i do
            for all i ∈ {1, ..., s} do
                if d^0_{i_j}(f_i) = 0 then
                    f_1, ..., f_s ← d^1_{i_j}(f_1, ..., f_s)
                    done ← false
                else if d^1_{i_j}(f_i) = 0 then
                    f_1, ..., f_s ← d^0_{i_j}(f_1, ..., f_s)
                    done ← false
                end if
            end for
        end for
    end while
    return f_1, ..., f_s
end function
```

The second one is to choose the literal that appears most times in the shortest polynomials. By a shorter we mean a polynomial which contains fewer variables. This can be implemented as in Algorithm 9.

The principle that we use in Algorithm 9 is that we give $s^{(k-l)}$ "points" to a literal i if x_i appears in a polynomial f_t, where $t \in \{1, \ldots, s\}$, which contains l distinct variables, and then we choose the literal i which gets the most "points".

3.3 Run times

In this section we will test our algorithms. For this we used the 3-CNF formulas in Table 1, which can be found in [7].

The first column in Table 1 is the name of the problems, the second column shows the number of variables, the third column shows the number of clauses, and the fourth column shows whenever the formula is satisfiable or not.

3.3.1 Multiplication

In this section we test our multiplication algorithms. We have used C++ *std* :: *bitset* to represent each monomial, and then a *std* :: *vector* of monomi-

Algorithm 8 Choose the first literal.

Input: A list of boolean polynomials $f_1, \ldots, f_s$

Output: An integer i such that x_i exists in some polynomial f_j, $j \in \{1, \ldots s\}$

```
function CHOOSE-LITERAL(f_1, ..., f_s)
    for all f_i do
        for all j ∈ {1, ..., k} do
            if f_i contains x_{i_j} then
                return i_j
            end if
        end for
    end for
    return 0
end function
```

Algorithm 9 Choose literal.

Input: A list of boolean polynomials $f_1, \ldots, f_s$

Output: An integer i such that x_i exists in some polynomial f_j, $j \in \{1, \ldots s\}$

```
function CHOOSE-LITERAL(f_1, ..., f_s)
    for all i ∈ {1, ..., s} do
        v ← 1
        for all j ∈ {1, ..., k} do
            if f_i does not contain x_{i_j} then
                v ← v · s
            end if
        end for
        for all j ∈ {1, ..., k} do
            if f_i contains x_{i_j} then
                l_{i_j} ← l_{i_j} + v
                if l_{i_j} > l_{i_max} then
                    i_max ← i_j
                end if
            end if
        end for
    end for
    return i_max
end function
```

Name	Variables	Clauses	Satisfiable?
uuf50-01.cnf	50	218	No
uuf75-01.cnf	75	325	No
uuf100-01.cnf	100	430	No
uuf125-01.cnf	125	538	No
uuf150-01.cnf	150	645	No
uuf175-01.cnf	175	753	No
uuf200-01.cnf	200	860	No
uuf225-01.cnf	225	960	No
uf100-01.cnf	100	430	Yes
uf200-01.cnf	225	860	Yes

Table 1: cnf-3-sat formulas

als to represent our polynomials. We have two different ways of multiplying boolean polynomials,

1. mul_1, which is Algorithm 5 with Algorithm 3 as MUL-MON-POL.
2. mul_2, which is Algorithm 5 with Algorithm 4 as MUL-MON-POL.

We will test our multiplication by attempting to solve a 3-CNF formula $C_1 \wedge \ldots \wedge C_s$, which we convert into a product of polynomials $f_1 \cdots f_s = T_1(C_1) \cdots T_1(C_s)$. We will then try to compute

$$g_i \leftarrow \begin{cases} 1 & i = 0 \\ mul_{1,2}(f_i, g_{i-1}) & i > 1 \end{cases} \tag{11}$$

for $i \in \{0, \ldots, s\}$ until the computations takes longer than 5 minutes.

We will measure the time it takes to compute g_i given that g_{i-1}. We will also note the number of monomials in g_{i-1} and f_i.

Note that this may not be the best way compute the product $T_1(C_1) \cdots T_1(C_s)$. Next, since mul_2 is expected to perform worse for larger monomials, we will test

$$g \leftarrow mul_i(x_1 x_2 \cdots x_j, g_{15}).$$

for each $j \in \{10, 20, \ldots 50\}$. The results of this are displayed in Table 3.

3.3.2 DPLL

In this section we compare our variations of DPLL. We have four variations of DPLL:

1. DPLL-first, which is Algorithm 6 with Algorithm 8 as choose-literal() and nothing as X.

uuf50-01.cnf	monomials	mul_1	mul_2
g_2	5·5	0s	0s
g_3	17·2	0s	0s
g_4	34·5	0s	0s
g_5	170·5	0.10s	0.01s
g_6	850·5	0.08s	0.01s
g_7	3050·5	0.32s	0.09s
g_8	15250·3	1.06s	0.19s
g_9	45750·3	3.15s	0.42s
g_{10}	86250·7	14.03s	2.12s
g_{11}	132750·3	9.65s	0.94s
g_{12}	398250·7	78.49s	15.81s
g_{13}	1513350·5	237.29s	39.72s
g_{14}	2368350·2	142.24s	8.56s
g_{15}	2842020·2	178.44s	20.97s
g_{16}	5684040·2	344.76s	20.39s
g_{17}	8344800·5	-	300.07s

Table 2: Multiplication

	mul_1	mul_2
$x_1x_2\cdots x_{10}$	73.64s	15.21s
$x_1x_2\cdots x_{20}$	29.48s	15.23s
$x_1x_2\cdots x_{30}$	27.11s	15.13s
$x_1x_2\cdots x_{40}$	26.97s	15.83s
$x_1x_2\cdots x_{50}$	27.77s	15.64s

Table 3: $g \leftarrow m \cdot g_{15}$

2. DPLL, which is Algorithm 6 with Algorithm 9 as choose-literal() and nothing as X.
3. DPLL-first-X, which is which is Algorithm 6 with Algorithm 8 as choose-literal() and Algorithm 7 as X.
4. DPLL-X, which is Algorithm 6 with Algorithm 9 as choose-literal() and Algorithm 7 as X.

We will focus on cnf-3-sat, so we have used a C++ $std : bitset$ of 2^3 bits, with a vector 3 integers to represent our polynomials in $\mathbb{Z}_2[x_{i_1}, x_{i_2}, x_{i_3}]$. Then we have a $std : vector$ of polynomials in $\mathbb{Z}_2[x_{i_1}, x_{i_2}, x_{i_3}]$ to represent the product.

We will measure the performance of each DPLL variation in *time*, *decisions* and *propagations*, where *time* is the time it takes for the process to terminate, *decisions* is the number of times we choose literal, and *propagations* is the number of times we put $f_1, \ldots, f_s \leftarrow d^{0,1}_{i_k}(f_1, \ldots, f_s)$ in X. We will not include the time it takes to convert the cnf-3-sat formula into a product of boolean polynomials. We will also include the time it takes for Minisat, which is one of the best sat solvers available [6], to solve the formulas. We will begin with some unsatisfiable 3-CNF formulas problems to see how far our DPLL variations can go. For convenience, we cancelled the computations after 5 minutes. The results are displayed in Table 4,5,6,7,8,9,10. To get a better comparison with Minisat, we will remove the 5 minutes time constraint and solve uuf-225-01.cnf with DPLL-X and Minisat. The results are displayed in Table 11.

Finally, we will do the test our Algorithms on two satisfiable problems. The results are displayed in Table 12 and Table 13.

3.4 Further Development and Conclusions

3.4.1 Multiplication

Table 3 and Table 2 indicates that Algorithm 4 is a faster than Algorithm 3. To further improve our multiplication we may consider reconstructing Algorithm 5 to avoid re-computations in Algorithm 4.

However, we cannot hope to improve our multiplication algorithm so much so that evaluating a product of polynomials, as in (11), competes with our DPLL algorithms as as a 3-CNF-SAT solver. This is because, if we wish to solve a 3-CNF formula with more than 100 variables with multiplication of boolean polynomials, we may need to deal with polynomials with more than 2^{100} monomials.

uuf50-01-cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	116.01s	0.1s	0.05s	0.03s	0.004 s
Decisions	2145437	130	334	29	
Propagations	0	2273	0	534	

Table 4: DPLL 50 variables unsat

uuf75-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	1.64s	0.19s	0.12s	0.004s
Decisions	-	1205	1058	67	
Propagations	-	25319	0	1551	

Table 5: DPLL 75 variables unsat

uuf100-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	13.67s	0.92s	0.49s	0.004s
Decisions	-	6797	3724	196	
Propagations	-	168606	0	5496	

Table 6: DPLL 100 variables, unsatisfiable

uuf125-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	69.43s	4.08s	2.16s	0.008s
Decisions	-	25201	13891	624	
Propagations	-	732184	0	21095	

Table 7: DPLL 125 variables, unsatisfiable

uuf150-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	-	15.99s	7.85s	0.036s
Decisions	-	-	45096	1775	
Propagations	-	-	0	66998	

Table 8: DPLL 150 variables, unsatisfiable

uuf175-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	-	52.26s	25.77s	0.076s
Decisions	-	-	124453	4423	
Propagations	-	-	0	186698	

Table 9: DPLL 175 variables, unsatisfiable

uuf200-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	-	164.67s	77.19s	0.30s
Decisions	-	-	346865	11614	
Propagations	-	-	0	524127	

Table 10: DPLL 200 variables, unsatisfiable

uuf-225-01.cnf	DPLL-X	Minisat
Time	415.06s	1.85s
Decisions	52122	
Propagations	2517818	

Table 11: DPLL-X , Minisat

uf100-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	0.52s	1.04s	0.56s	0s
Decisions	-	268	4366	224	
Propagations	-	6578	0	6407	

Table 12: DPLL 100 variables, satisfiable

uf200-01.cnf	DPLL-first	DPLL-first-X	DPLL	DPLL-X	Minisat
Time	-	-	162.56s	74.53s	0.22 s
Decisions	-	-	341791	11160	
Propagations	-	-	0	499964	

Table 13: DPLL 200 variables, satisfiable

3.4.2 DPLL

From the results in Table 10,11,13 we can see that even though we have a long way to go until we can compete with Minisat, we were able to solve quite large 3-CNF formulas, and we should be able to see why DPLL with Algorithm 9 as choose-literal() performs so much better than DPLL with Algorithm 8 as choose-literal().

We can first conclude that if $d_i^{0,1}(f_1, \ldots, f_s) = 0$, for some $i \in \{1, \ldots, n\}$, then it is crazy to not assign the opposite value to x_i. Next if we choose the branching literal i from a polynomial which only contains two variables, then we will get at least one literal for "free" in $d_i^0(f_1, \ldots, f_s)$ or in $d_i^1(f_1, \ldots, f_s)$, and choosing the literal which appears in most such polynomials will give us will give us the most variables for "free" (with some exceptions).

However, this is not always be optimal.

Example 3.10. Consider the formula

$$(\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_2) \wedge (\psi_1 \vee \neg\psi_2) \wedge (\neg\psi_1 \vee \neg\psi_2)\wedge$$

$$\wedge(\psi_3 \vee \psi_4) \wedge (\psi_3 \vee \psi_5) \wedge (\psi_3 \vee \psi_6) \wedge (\psi_3 \vee \psi_7) \wedge (\psi_3 \vee \psi_8).$$

We see that ψ_3 is the literal which appears in most clauses. Thus Algorithm 9 would suggest ψ_3 as the branching literal. However choosing ψ_3 would force us to choose branching literal once more before we can conclude that the formula is unsatisfiable, while choosing ψ_1 or ψ_2 would allow us to conclude that the formula is unsatisfiable after our unit propagations.

For further improvements we may consider to take our time to look for pairs x_i, x_j which appears in multiple polynomials. When each polynomial $f_1, \ldots, f_s$ contains 3 variables we may even consider to look for triples $x_{i_1}, x_{i_2}, x_{i_3}$ which appears in multiple polynomials.

However, in [4] it is proven that finding an optimal branching literal is itself a NP-hard problem. Thus we cannot hope to always find an optimal branching literal in polynomial time.

So far, we have not seen any benefits from viewing our clauses as boolean polynomials. One could think that we could gain something by using the fact that each boolean polynomial may represent any boolean formula of k variables, and not just disjunctions of variables ψ_i or $\neg\psi_i$ as in a k-CNF formula. However, this might make it harder to choose the branching literal.

Example 3.11. Consider the 3-CNF formula

$$(\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_3) \wedge C_3 \wedge \ldots \wedge C_s.$$

In Example 2.30 we saw that

$$\begin{aligned} T_1((\psi_1 \vee \psi_2) \wedge (\neg\psi_1 \vee \psi_3) \wedge C_3 \wedge \ldots \wedge C_s) = \\ = (x_1 + x_2 + x_1x_2)(1 + x_1 + x_1x_3)T_1(C_3) \cdots T_1(C_s)). \end{aligned}$$

If we are allowing 3 variables in each polynomial, then we might compress this into

$$(x_2 + x_1x_2 + x_1x_3)T_1(C_3) \cdots T_1(C_s),$$

which will give us one less polynomial to check at each iteration of DPLL. However, if we do this and we choose literal as in Algorithm 9, then we will miss the fact that x_1 appears in two polynomials with only two variables.

Thus we might suspect that converting a boolean formula into boolean polynomials in order to perform a DPLL based algorithm will never serve any real purpose.

References

[1] Tobias Andersen : Ekvationssystem i $f \in \mathbb{Z}_2[x_1, \ldots, x_n]$, Bachelor thesis, Stockholm University, 2013.

[2] Mark Anderstam : Solution methods to polynomial equations over $\mathbb{Z}_2$, Bachelor thesis, Stockholm University, 2014.

[3] Michael Brickenstein, Alexander Dreyer : PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. J. Symb. Comput. 44 (2009), no. 9, 1326 – 1345.

[4] Paolo Liberatore : On the Complexity of Choosing the Branching Literal in DPLL. Artificial Intelligence Volume 116, Issues 1–2, January 2000, Pages 315–326

[5] Samuel Lundqvist : Elementary algebra related to the SAT problem, Preprint, 2012.

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

[7] http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html