Algebraic Polynomial Sum Solver Over \{0, 1\}

Frank Vega
CopSonic, 1471 Route de Saint-Nauphary 82000 Montauban, France
vega.frank@gmail.com

Abstract
Given a polynomial \(P(x_1, x_2, \ldots, x_n)\) which is the sum of terms, where each term is a product of two distinct variables, then the problem APSS consists in calculating the total sum value of \(\sum_{x \in U} P(u_1, u_2, \ldots, u_n)\), for all the possible assignments \(U_i = \{u_1, u_2, \ldots, u_n\}\) to the variables such that \(u_j \in \{0, 1\}\). APSS is the abbreviation for the problem name Algebraic Polynomial Sum Solver Over \{0, 1\}. We show that APSS is in \#L and therefore, it is in FP as well. The functional polynomial time solution was implemented with Scala in https://github.com/frankvegadelgado/sat using the DIMACS format for the formulas in MONOTONE-2SAT.

1 Introduction

1.1 Polynomial time verifiers
Let \(\Sigma\) be a finite alphabet with at least two elements, and let \(\Sigma^*\) be the set of finite strings over \(\Sigma\) [2]. A Turing machine \(M\) has an associated input alphabet \(\Sigma\) [2]. For each string \(w\) in \(\Sigma^*\) there is a computation associated with \(M\) on input \(w\) [2]. We say that \(M\) accepts \(w\) if this computation terminates in the accepting state, that is \(M(w) = \text{"yes"}\) [2]. Note that \(M\) fails to accept \(w\) either if this computation ends in the rejecting state, that is \(M(w) = \text{"no"}\), or if the computation fails to terminate, or the computation ends in the halting state with some output, that is \(M(w) = y\) (when \(M\) outputs the string \(y\) on the input \(w\)) [2].

The language accepted by a Turing machine \(M\), denoted \(L(M)\), has an associated alphabet \(\Sigma\) and is defined by:

\[ L(M) = \{w \in \Sigma^* : M(w) = \text{"yes"}\}. \]

Moreover, \(L(M)\) is decided by \(M\), when \(w \notin L(M)\) if and only if \(M(w) = \text{"no"}\) [4]. We denote by \(t_M(w)\) the number of steps in the computation of \(M\) on input \(w\) [2]. For \(n \in \mathbb{N}\) we denote by \(T_M(n)\) the worst case run time of \(M\); that is:

\[ T_M(n) = \max\{t_M(w) : w \in \Sigma^n\} \]

where \(\Sigma^n\) is the set of all strings over \(\Sigma\) of length \(n\) [2]. We say that \(M\) runs in polynomial time if there is a constant \(k\) such that for all \(n\), \(T_M(n) \leq n^k + k\) [2]. In other words, this means the language \(L(M)\) can be decided by the Turing machine \(M\) in polynomial time. Therefore, \(P\) is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [4]. A verifier for a language \(L_1\) is a deterministic Turing machine \(M\), where:

\[ L_1 = \{w : M(w, c) = \text{"yes" for some string } c\}. \]

We measure the time of a verifier only in terms of the length of \(w\), so a polynomial time verifier runs in polynomial time in the length of \(w\) [2]. A verifier uses additional information, represented by the symbol \(c\), to verify that a string \(w\) is a member of \(L_1\). This information
APSS is in P

is called certificate. \(NP\) is also the complexity class of languages defined by polynomial time verifiers [7].

A decision problem in \(NP\) can be restated in this way: There is a string \(c\) with \(M(w,c) = \text{"yes"}\) if and only if \(w \in L_1\), where \(L_1\) is defined by the polynomial time verifier \(M\) [7]. The function problem associated with \(L_1\), denoted \(FL_1\), is the following computational problem: Given \(w\), find a string \(c\) such that \(M(w,c) = \text{"yes"}\) if such string exists; if no such string exists, then reject, that is, return \("no\) [7]. The complexity class of all function problems associated with languages in \(NP\) is called \(FNP\) [7]. \(FP\) is the complexity class that contains those problems in \(FNP\) which can be solved in polynomial time [7].

To attack the \(P\) versus \(NP\) question the concept of \(NP\)-completeness has been very useful [6]. A principal \(NP\)-complete problem is \(SAT\) [6]. An instance of \(SAT\) is a Boolean formula \(\phi\) which is composed of:

1. Boolean variables: \(x_1, x_2, \ldots, x_n\);
2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as \(\wedge\) (AND), \(\lor\) (OR), \(\neg\) (NOT), \(\Rightarrow\) (implication), \(\Leftrightarrow\) (if and only if);
3. and parentheses.

A truth assignment for a Boolean formula \(\phi\) is a set of values for the variables in \(\phi\). On the one hand, a satisfying truth assignment is a truth assignment that causes \(\phi\) to be evaluated as true. On the other hand, a truth assignment that causes \(\phi\) to be evaluated as false is a unsatisfying truth assignment. A Boolean formula with some satisfying truth assignment is satisfiable and without any satisfying truth assignment is unsatisfiable. The problem \(SAT\) asks whether a given Boolean formula is satisfiable [6].

An important complexity is \(\text{Sharp-P}\) (denoted as \(#P\)) [9]. We can also define the class \(#P\) using polynomial time verifiers. Let \(\{0, 1\}^*\) be the infinite set of binary strings, a function \(f : \{0, 1\}^* \rightarrow \mathbb{N}\) is in \(#P\) if there exists a polynomial time verifier \(M\) such that for every \(x \in \{0, 1\}^*\),

\[
f(x) = |\{y : M(x,y) = \text{"yes"}\}|
\]

where \(|\cdots|\) denotes the cardinality set function [2]. We could use the parsimonious reduction for the completeness of this class [2]. In computational complexity theory, a parsimonious reduction is a transformation from one problem to another that preserves the number of solutions [2].

1.2 Logarithmic space verifiers

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [8]. The work tapes may contain at most \(O(\log n)\) symbols [8]. In computational complexity theory, \(L\) is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [7]. \(NL\) is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [7].

We can give a certificate-based definition for \(NL\) [2]. The certificate-based definition of \(NL\) assumes that a logarithmic space Turing machine has another separated read-only tape [2]. On each step of the machine, the machine’s head on that tape can either stay in place or move to the right [2]. In particular, it cannot reread any bit to the left of where the head currently is [2]. For that reason, this kind of special tape is called \("read-once\) [2].
A language $L_1$ is in $NL$ if there exists a deterministic logarithmic space Turing machine $M$ with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0, 1\}^*$:

$$x \in L_1 \iff \exists\ u \in \{0, 1\}^{p(|x|)} \text{ such that } M(x, u) = \text{"yes"}$$

where by $M(x, u)$ we denote the computation of $M$ where $x$ is placed on its input tape, and the certificate $u$ is placed on its special read-once tape, and $M$ uses at most $O(\log|x|)$ space on its read/write work tapes for every input $x$, where $[\ldots]$ is the bit-length function [2]. $M$ is called a logarithmic space verifier [2].

An interesting complexity class is $\text{Sharp-L}$ (denoted as $\#L$). $\#L$ has the same relation to $L$ as $\#P$ does to $P$ [1]. We can define the class $\#L$ using logarithmic space verifiers as well.

Let $\{0, 1\}^*$ be the infinite set of binary strings, a function $f : \{0, 1\}^* \to \mathbb{N}$ is in $\#L$ if there exists a logarithmic space verifier $M$ such that for every $x \in \{0, 1\}^*$:

$$f(x) = |\{u : M(x, u) = \text{"yes"}\}|$$

where $|\ldots|$ denotes the cardinality set function [1]. We could use the parsimonious reduction for the completeness of this class too [2].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [8]. The work tapes must contain at most $O(\log n)$ symbols [8]. A logarithmic space transducer $M$ computes a function $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after $M$ halts when it is started with $w$ on its input tape [8]. We call $f$ a logarithmic space computable function [8]. We say that a language $L_1 \subseteq \{0, 1\}^*$ is logarithmic space reducible to a language $L_2 \subseteq \{0, 1\}^*$, written $L_1 \leq L_2$, if there exists a logarithmic space computable function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$:

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

For example, this kind of reduction is used for the completeness in the $NL$.

A literal in a Boolean formula is an occurrence of a variable or its negation [4]. A Boolean formula is in conjunctive normal form, or $\text{CNF}$, if it is expressed as an AND of clauses, each of which is the OR of one or more literals [4]. A Boolean formula is in 2-conjunctive normal form or $2\text{CNF}$, if each clause has exactly two distinct literals [7]. A relevant $NL$-complete language is $2\text{CNF}$ satisfiability, or $2\text{SAT}$ [7]. In $2\text{SAT}$, it is asked whether a given Boolean formula $\phi$ in $2\text{CNF}$ is satisfiable. The instances of $\text{MONOTONE-2SAT}$ does not contain any negated variable.

### 1.3 A polynomial time problem

Let’s define the following problem

➤ **Definition 1. #Algebraic Polynomial Sum Solver Over $\{0, 1\}$ (APSS)**

**INSTANCE:** A polynomial $P(x_1, x_2, \ldots, x_n)$ which is the sum of terms, where each term is a product of two distinct variables.

**ANSWER:** Calculate the total sum value of $\sum_{U_i} P(u_1, u_2, \ldots, u_n)$, for all the possible assignments $U_i = \{u_1, u_2, \ldots u_n\}$ to the variables such that $u_j \in \{0, 1\}$.

Let’s see an example:

**Instance:** $P(x_1, x_2, x_3) = x_1 \times x_2 + x_2 \times x_3$. 
4 APSS is in P

Table 1 Evaluation for all possible assignments

| $x_1$ | $x_2$ | $x_3$ | $P(x_1, x_2, x_3)$ |
|-------|-------|-------|-------------------|
| 1     | 1     | 1     | 2                 |
| 1     | 1     | 0     | 1                 |
| 0     | 1     | 1     | 1                 |
| 0     | 0     | 0     | 0                 |
| 1     | 0     | 1     | 0                 |
| 0     | 0     | 1     | 0                 |
| 1     | 0     | 0     | 0                 |
| 0     | 1     | 0     | 0                 |

Answer: The total sum value is 4 for all the possible assignments:
Total: $2 + 1 + 1 + 0 + 0 + 0 + 0 + 0 = 4$ (see it in Table 1).

We solve this problem reducing in logarithmic space and parsimoniously to another problem #CLAUSES-2UNSAT. We show an algorithm for the problem #CLAUSES-2UNSAT which is in #L and therefore, it is in FP as well. In this way, we prove that APSS can be solved in polynomial time.

2 Results

Definition 2. Given a Boolean formula $\phi$ with $m$ clauses, the density of states $n(E)$ for some integer $0 \leq E \leq m$ counts the number of truth assignments that leave exactly $E$ clauses unsatisfied in $\phi$ [5]. The weighted density of states $m(E)$ is equal to $E \times n(E)$. The sum of the weighted densities of states of a Boolean formula in 2CNF with $m$ clauses is equal to $\sum_{E=0}^{m} m(E)$.

Let’s consider a function problem:

Definition 3. #CLAUSES-2UNSAT

INSTANCE: Two natural numbers $n, m$, and a Boolean formula $\phi$ in 2CNF of $n$ variables and $m$ clauses. The clauses are represented by an array $C$, such that $C$ represents a set of $m$ set elements, where $C[i] = S_i$ if and only if $S_i$ is exactly the set of literals into a clause $c_i$ in $\phi$ for $1 \leq i \leq m$. Besides, each variable in $\phi$ is represented by a unique integer between 1 and $n$. In addition, a negative or positive integer represents a negated or non-negated literal, respectively. This is similar to the format [DIMACS](http://www.satcompetition.org/2009/format-benchmarks2009.html) for the formulas where the literals are represented by negative or nonnegative integers.

ANSWER: The sum of the weighted densities of states of the Boolean formula $\phi$.

Theorem 4. #CLAUSES-2UNSAT $\in$ #L.

Proof. We are going to show there is a nondeterministic Turing machine $M$ such that $M$ runs in logarithmic space in the length of $(n, m, C)$. We use the nondeterministic logarithmic space Algorithm 1, where this routine generates a truth assignment in logarithmic space just selecting a negation or a positive representation of a variable $1 \leq i \leq n$, since every variable is represented by an integer between 1 and $n$ in $C$. We also assume the value of each literal selected within $y$ is false over the generated truth assignment.

First of all, the Algorithm 1 select the index in $C$ of a clause from the value of the variable $k$. Later, we increment the variable count as much as the literal $y$ appears in the clause $C[k]$.
ALGORITHM 1: ALGO

Data: \((n, m, C)\) where \((n, m, C)\) is an instance of \#CLAUSES-2UNSAT

Result: Accept whether there is an unsatisfied clause for a generated truth assignment

// Generate nondeterministically an arbitrary integer between 1 and \(m\)
\[ k \leftarrow \text{random}(1, m); \]

// Initialize the variable \(\text{count}\)
\[ \text{count} \leftarrow 0; \]

for \(i \leftarrow 1\) to \(n + 1\) do
    if \(i = n + 1\) then
        if \(\text{count} = 2\) then
            // The clause \(C[k]\) is unsatisfied for the generated truth assignment
            return "yes";
        else
            // The clause \(C[k]\) is satisfied for the generated truth assignment
            return "no";
        end
    end
    else
        // Generate nondeterministically either the integer \(i\) or \(-i\)
        \[ y \leftarrow \text{random}(i); \]
        for \(j \leftarrow 1\) to \(m\) do
            if \(y \in C[j] \land j = k\) then
                // Increment the value of the variable \(\text{count}\)
                \[ \text{count} \leftarrow \text{count} + 1; \]
            end
        end
    end
end
6 APSS is in P

Since a clause contains only two literals, then if we finish the iteration of the possible values in the generated truth assignment, then we can say the clause indexed with the number \( k \) in \( C \) is unsatisfied when \( \text{count} = 2 \).

Furthermore, we can make this Algorithm 1 in logarithmic space, because the variables that we could use for the iteration of the variables and elements in \( C \) have a logarithmic space in relation to the length of the instance \((n, m, C)\). Besides, the Algorithm 1 is nondeterministic, since we generate in a nondeterministic way the values of the variables \( k \) and \( y \). In addition, every generated truth assignment is always stored in logarithmic space in relation to the instance \((n, m, C)\), since we only focus in a single literal of the truth assignment from the for loop each time.

For every unsatisfying truth assignment represented by a generated truth assignment, then there will be always as many acceptance paths as unsatisfied clauses have the evaluation of that truth assignment in the formula \( \phi \). Consequently, we demonstrate that \#\text{CLAUSES-2UNSAT} belongs to the complexity class \#L. Certainly, the number of all accepting paths in the Algorithm 1 is exactly the sum of the number of unsatisfied clauses from all the truth assignments in \( \phi \), that is exactly the sum of the weighted densities of states of the Boolean formula \( \phi \). In conclusion, we show that \#\text{CLAUSES-2UNSAT} is indeed in \#L. ◀

Let’s consider an interesting reduction:

▶ Theorem 5. \( \text{APSS} \leq_l \#\text{CLAUSES-2UNSAT} \), where this logarithm space reduction is a parsimonious reduction.

Proof. We solve this problem reducing in logarithmic space the polynomial \( P(x_1, x_2, \ldots, x_n) \) into a \textsc{Monotone-2Sat} formula \( \phi \) such that for each term \( x_i \times x_j \), we make a clause \((x_i \lor x_j)\) and join all the summands by a disjunction with the \& (AND) operator. Let’s take as example the previous instance \( P(x_1, x_2, x_3) = x_1 \times x_2 + x_2 \times x_3 \) of \textsc{APSS} which could be reduced to \( \phi = (x_1 \lor x_2) \land (x_2 \lor x_3) \) (the sum of the weighted densities of states for the Boolean formula \( \phi \) is 4). This is equivalent to

\[
\begin{align*}
p &\text{ cnf } 3 \ 2 \\
1 &\ 2 \ 0 \\
2 &\ 3 \ 0
\end{align*}
\]

in the format DIMACS. Certainly, we can affirm the value of a term \( x_i \times x_j \) is equal to 1 when \((x_i \lor x_j)\) is unsatisfied. Consequently, the sum of the weighted densities of states of the Boolean formula \( \phi \) will be equal to the answer of the instance for \textsc{APSS}, that is a parsimonious reduction. Indeed, every unsatisfying truth assignment \( T_i = \{t_1, t_2, \ldots, t_n\} \) in \( \phi \) with \( K \) unsatisfied clauses corresponds to an assignment \( U_i = \{u_1, u_2, \ldots, u_n\} \) such that \( P(u_1, u_2, \ldots, u_n) = K \), where for each \( j \) we have “\( u_j = \neg t_j \)” (which actually means \( u_j = 1 \) if and only if \( t_j \) is false).

▶ Theorem 6. \( \text{APSS} \in \#L \) and therefore, \( \text{APSS} \in \text{FP} \).

Proof. We know \#L is closed under a logarithm space reduction when this one is also a parsimonious reduction. Furthermore, we know that \#L is contained in the class \text{FP} [1], [3], [2]. ◀

2.1 Code

This project was implemented on February 8th of 2021 in a GitHub Repository [10]. This was a partial implementation since this project receives as input the already reduced \textsc{Monotone-2Sat} formulas in the format DIMACS instead of instances from \textsc{APSS}. 

Preprints (www.preprints.org)  |  NOT PEER-REVIEWED  |  Posted: 11 February 2021
doi:10.20944/preprints201908.0037.v6
References

1 Carme Álvarez and Birgit Jenner. A Very Hard Log-Space Counting Class. *Theor. Comput. Sci.*, 107(1):3–30, January 1993. doi:10.1016/0304-3975(93)90252-0.
2 Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.
3 Allan Borodin, Stephen A. Cook, and Nick Pippenger. Parallel Computation for Well-Endowed Rings and Space-Bounded Probabilistic Machines. *Inf. Control*, 58(1–3):113–136, July 1984. doi:10.1016/S0019-9958(83)80060-6.
4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
5 Stefano Ermon, Carla P. Gomes, and Bart Selman. Computing the Density of States of Boolean Formulas. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, pages 38–52, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.5555/1886008.1886016.
6 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1 edition, 1979.
7 Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
8 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.
9 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, (2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.
10 Frank Vega. Algebraic Polynomial Sum Solver Over $\{0, 1\}$, February 2021. In GitHub Repository at https://github.com/frankvegadelgado/sat. Retrieved February 9, 2021.