A polynomial algorithm for SAT

Yuval Filmus

April 25, 2017

1 Computation model

In this short note, we exhibit polynomial algorithms for SAT and TQBF, in an appropriate computation model. Our model allows performing both arithmetic operations and bitwise operations, charging 1 for each operation. In more detail, all our variables are integers, and we support the following operations:

1. $a \leftarrow b + c$.
2. $a \leftarrow b - c$.
3. $a \leftarrow bc$.
4. $a \leftarrow \lfloor b/c \rfloor$. (If $c = 0$, the output is undefined.)
5. $a \leftarrow b \bmod c$. (If $c = 0$, the output is undefined.)
6. $a \leftarrow b\&c$ (bitwise AND). (If b, c are not both non-negative, the output is undefined.)
7. $a \leftarrow b|c$ (bitwise OR). (If b, c are not both non-negative, the output is undefined.)
8. $a \leftarrow b\hat{\ }c$ (bitwise XOR). (If b, c are not both non-negative, the output is undefined.)

In addition, we allow control operations (comparison, IF, WHILE, and so on).

2 A polynomial algorithm for SAT

SAT (or rather, its variant Formula-SAT) is the following problem. Given a formula φ over the de Morgan basis $(\vee, \wedge, \neg)$, is there a truth assignment for the variables in φ which satisfies φ? If so, we say that φ is *satisfiable*, and otherwise φ is *unsatisfiable*. For simplicity, we assume that the variables are $x_1, \ldots, x_n$ for some n.

For example, the formula

$$(x_1 \vee (x_2 \wedge \neg x_3)) \wedge x_4$$

is satisfiable (assigning TRUE to all variables satisfies it), whereas the formula

$$x_1 \wedge \neg x_1$$

is unsatisfiable.

We say that an algorithm for SAT is *polynomial* if on a formula on n variables of length N it runs in time $\mathsf{poly}(n, N)$, in the computation model described in the introduction.

Our algorithm will use the *truth table* representation of Boolean function, in which we identify TRUE with 1 and FALSE with 0. Given a Boolean function f on $x_1, \ldots, x_n$, we define

$$\langle f \rangle_n = \sum_{t_1=0}^{1} \cdots \sum_{t_n=0}^{1} 2^{2^0 t_1 + 2^1 t_2 + \cdots + 2^{n-1} t_n} f(t_1, \ldots, t_n).$$

In other words, bit $t_n \dots t_1$ (interpreted as a binary number) of $\langle f \rangle_n$ is $f(t_1, \dots, t_n)$.

For example, let $f = x_1 \vee \neg x_2$. Then

$$\langle f \rangle_2 = (1011)_2 = 11.$$

Reading the bits from right to left and starting with zero, the zeroth bit corresponds to $(x_1, x_2) = (0, 0)$, the first to $(1, 0)$, the second to $(0, 1)$, and the third to $(1, 1)$.

Here is the idea of our algorithm. We will give an efficient algorithm for computing $\langle x_i \rangle_n$ (that is, $\langle f \rangle_n$ for $f(x_1, \dots, x_n) = x_i$) for all i. Then we will give efficient algorithms to compute $\langle \neg f \rangle_n$ from $\langle f \rangle_n$, and $\langle f \wedge g \rangle_n, \langle f \vee g \rangle_n$ from $\langle f \rangle_n$ and $\langle g \rangle_n$. Using all of these, it is possible to compute $\langle \varphi \rangle_n$ efficiently, and then φ is satisfiable iff $\langle \varphi \rangle_n \neq 0$.

Computing $\langle x_i \rangle_n$ We have

$$\begin{aligned}
\langle x_i \rangle_n &= \sum_{t_1=0}^{1} \cdots \sum_{t_{i-1}=0}^{1} \sum_{t_{i+1}=0}^{1} \cdots \sum_{t_n=0}^{1} 2^{2^0 t_1 + \cdots + 2^{i-2} t_{i-1} + 2^{i-1} + 2^i t_{i+1} + \cdots 2^{n-1} t_n} \\
&= \sum_{r=0}^{2^{i-1}-1} \sum_{s=0}^{2^{n-i}-1} 2^{r + 2^{i-1} + 2^i s} \\
&= 2^{2^{i-1}} \left(\sum_{r=0}^{2^{i-1}-1} 2^r \right) \left(\sum_{s=0}^{2^{n-i}-1} (2^{2^i})^s \right) \\
&= 2^{2^{i-1}} \cdot \left(2^{2^{i-1}} - 1 \right) \cdot \left(\frac{2^{2^n} - 1}{2^{2^i} - 1} \right) \\
&= \frac{2^{2^{i-1}} (2^{2^n} - 1)}{2^{2^{i-1}} + 1}.
\end{aligned}$$

We can compute 2^{2^x} using x multiplication operations by repeatedly squaring 2, so the entire computation uses $O(n)$ operations.

Computing $\langle \neg f \rangle_n$ from $\langle f \rangle_n$ It is not hard to check that

$$\langle \neg f \rangle_n = 2^{2^n} - 1 - \langle f \rangle_n.$$

Indeed, if we think of $\langle f \rangle_n$ as a bitstring of length 2^n, then we obtain $\langle \neg f \rangle_n$ by complementing it (negating all bits), and this is the same as subtracting it from the bitstring 1^{2^n}, whose numerical value is $2^{2^n} - 1$.

Computing $\langle f \wedge g \rangle_n$ and $\langle f \vee g \rangle_n$ given $\langle f \rangle_n$ and $\langle g \rangle_n$ This is just bitwise AND and bitwise OR, which are primitive operations in our model.

3 A polynomial algorithm for TQBF

TQBF (Totally Quantified Boolean Formulas) is a generalization of SAT. Given a formula φ on the variables $x_1, \dots, x_n$, in SAT we are interested in the truth value of

$$\exists x_1 \exists x_2 \cdots \exists x_n \varphi(x_1, \dots, x_n).$$

In TQBF, we are given a formula φ and n quantifiers $Q_1, \dots, Q_n$ (each either $\exists$ or $\forall$), and we are interested in the truth value of

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \varphi(x_1, \dots, x_n).$$

For example,

$$\forall x_1 \exists x_2 (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$$

belongs to TQBF (it is a true statement), but if we switch the quantifiers to $\exists x_1 \forall x_2$ it doesn't belong to TQBF (it is a false statement).

The polynomial algorithm for TQBF is an extension of the polynomial algorithm for SAT, and it uses the same building blocks. It first computes $\langle \varphi \rangle_n$. The crucial observation is that *as bitstrings* we have the equality

$$\langle \varphi \rangle_n = \langle \varphi|_{x_n=1} \rangle_{n-1} \langle \varphi|_{x_n=0} \rangle_{n-1},$$

where $\varphi|_{x_n=b}$ is the formula (or function) on the $n-1$ variables $x_1, \ldots, x_{n-1}$ whose value at $x_1, \ldots, x_{n-1}$ is $\varphi(x_1, \ldots, x_{n-1}, b)$, that is, we obtain it by substituting b for x_n. We can extract the two parts as follows:

$$\langle \varphi_{x_n=1} \rangle_{n-1} = \lfloor \langle \varphi \rangle_n / 2^{2^{n-1}} \rfloor, \ \langle \varphi_{x_n=0} \rangle_{n-1} = \langle \varphi \rangle_n \bmod 2^{2^{n-1}}.$$

We can then compute $\langle Q_n x_n \varphi \rangle_{n-1}$ using the formulas

$$\langle \forall x_n \varphi \rangle_n = \langle \varphi_{x_n=1} \rangle_{n-1} \& \langle \varphi_{x_n=0} \rangle_{n-1}, \ \langle \exists x_n \varphi \rangle_n = \langle \varphi_{x_n=1} \rangle_{n-1} | \langle \varphi_{x_n=0} \rangle_{n-1}.$$

Repeating this $n-1$ more times, we can compute $\langle Q_1 x_1 \cdots Q_n x_n \varphi \rangle_0$, whose numeric value is the truth value of $Q_1 x_1 \cdots Q_n x_n \varphi$.

4 So P=NP=PSPACE?

SAT is NP-complete, and TQBF is PSPACE-complete. So it would seem that we have shown that $\mathsf{P} = \mathsf{NP} = \mathsf{PSPACE}$, which is considered unlikely. What went wrong? If we try to convert the algorithms into a more conventional model such as Turing machines or the RAM model, then we only get *exponential time* algorithms, since $\langle \varphi \rangle_n$ has length 2^n, and so operations on it take time $\Omega(2^n)$. Our model is thus "too strong". Nevertheless, similar models are used in arithmetic complexity theory, especially in contexts where "cheating" (using very large numbers) is ruled out for some reason.