Boolean Satisfiability

There are many alternative ways to define a Boolean expression, but for our discussion, we must fix one of them. We define a string to be a Boolean expression if it is generated by the following context-free grammar $G$, with start symbol $S$: Let $BOOL$ be the language of all strings generated by $G$.

$$
S \rightarrow !S \text{ (logical not)} \\
S \rightarrow S \Rightarrow S \text{ (implies)} \\
S \rightarrow S \equiv S \text{ (logical equal)} \\
S \rightarrow S \neq S \text{ (logical not equal)} \\
S \rightarrow S * S \text{ (logical and)} \\
S \rightarrow S + S \text{ (logical or)} \\
S \rightarrow (S) \\
S \rightarrow I \text{ (I generates all identifiers)} \\
I \rightarrow AN \text{ (The first symbol of an identifier must be a letter)} \\
A \rightarrow a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z \\
N \rightarrow AN | 0N | 1N | 2N | 3N | 4N | 5N | 6N | 7N | 8N | 9N | \lambda \\
S \rightarrow 0 \\
S \rightarrow 1
$$

The strings generated by $I$ are called identifiers. An assignment of a Boolean expression $E$ is an assignment of each identifier in $I$ to a logical value, either 0 (false) or 1 (true). We say that an assignment satisfies $E$ the if evaluation of $E$ yields 1, after replacing each identifier by its assigned value. Otherwise, $E$ is not satisfiable, i.e., a contradiction. Evaluation uses the rules of precedence of C++.

**Definition 1** A language $L$ is $\mathcal{NP}$-complete if there is a $\mathcal{P}$-time reduction of any given $\mathcal{NP}$-time language to $L$.

We define an instance of the Boolean satisfiability problem to be a Boolean expression, $E \in BOOL$, where $E \in SAT$ if $E$ is satisfiable.

**Theorem 1** Every $\mathcal{NP}$-time language has a $\mathcal{P}$-time reduction to $SAT$.

Thus, by definition, $SAT$ is $\mathcal{NP}$-complete. You can find the proof of Theorem 1 on the internet.

**Conjunctive Normal Form**

We say that a Boolean expression $E$ is in conjunction normal form, or CNF, if $E$ is the conjunction of clauses, each of which consists of the disjunction of terms, each of which is a variable or the negation of a variable. We say that $E \in CNF$ is in 3CNF if each of its clauses has three terms. That is,

$$
E = C_1 \ast C_2 \ast \cdots \ast C_k
$$

where $C_i = (t_{i1} + t_{i2} + t_{i3})$, and where each term $t_{ij}$ is a variable or the negation of a variable. 2CNF, 4CNF, etc. are defined similarly.

An instance of the 3SAT problem is a Boolean expression in 3CNF form. An expression $E$
is a member of the language 3SAT if it is satisfiable and in 3CNF form.\(^1\) Thus, \(3SAT = 3CNF \cap SAT\).

**Polynomial Time Reduction of SAT to 3SAT**

We define two Boolean expressions \(E\) and \(E'\) to be *sat-equivalent* if they both have the same satisfiability, *i.e.*, if either \(E\) and \(E'\) are both satisfiable or \(E\) and \(E'\) are both contradictions. We will define a \(P\)-time reduction of SAT to 3SAT, *i.e.*, a \(P\)-time function

\[
R : BOOL \rightarrow 3CNF
\]

such that \(E' = R(E)\) is sat-equivalent to \(E\), for any Boolean expression \(E\). We first construct a parse tree for \(E\), using the grammar \(G\). and we simplify the parse tree to combine equivalent nodes. We choose a set of identifiers that are not used for \(E\), such as \(e_0, e_1, \ldots\), and place one identifier at each internal node of the parse tree, where \(e_0\) is placed at the root. For each internal node, we write a Boolean expression stating that the variable at that node is equal to the concatenation of its children. Let \(E''\) be the \(e_0\) with the conjunction of those expressions. \(E''\) is sat-equivalent to \(E\). We then use the following table to replace each clause of \(E''\) by a 3CNF expression. The resulting expression is in 3CNF form, and is sat-equivalent to \(E\).

\[
\begin{align*}
a \equiv b + c & \quad \text{equals} \quad (a + !b) \ast (a + b + c) \ast (a + b + !c) \\
a \equiv b \ast c & \quad \text{equals} \quad (a + b) \ast (a + b + !c) \ast (a + b + !c) \\
a \equiv !b & \quad \text{equals} \quad (a + b) \ast (a + !b) \\
a \equiv b \Rightarrow c & \quad \text{equals} \quad (a + b) \ast (a + b + c) \ast (a + !b + !c)
\end{align*}
\]

**Theorem 2** If SAT is \(NP\)-complete then 3SAT is \(NP\)-complete.

**Example**

Let \(E = ! (x + y \Rightarrow z) \ast z\). We show the parse three and the compressed parse tree of \(E\), and then we replace each internal node by a unique auxiliary variable.

Then

\[
E'' = e_0 \ast (e_0 \equiv e_1 \ast z) \ast (e_1 \equiv !e_2) \ast (e_2 \equiv e_3 \Rightarrow z) \ast (e_3 \equiv x + y)
\]

Using the equalities given in the table, replace each clause of \(E''\) by an expression in CNF form:

\[
\begin{align*}
E' &= e_0 \ast (e_0 + e_1) \ast (e_0 + e_1 + !z) \ast (e_0 + e_1 + !z) \ast (e_1 + e_2) \ast (e_1 + e_2) \\
&\ast (e_2 + e_3) \ast (e_2 + e_3 + z) \ast (e_2 + e_3 + z) \ast (e_3 + x + y) \ast (e_3 + x + y)
\end{align*}
\]

We can pad with redundant terms to change \(E'\) into strict 3CNF form.

\(^{1}\)When convenient, We can allow clauses of fewer than three terms, by introducing redundant terms: For example, \((x + y)\) can be replaced by the equivalent \((x + y + y)\).
parsed tree of $E$

compressed parse tree of $E$

compressed parse tree of $E$ with auxiliary variables