Satisfiability of Boolean Expressions

In the previous section, we have analysed in great detail one particular problem—reachability in directed graphs. The reason for the attention devoted to this one problem is that studying algorithms for it provides insight into nondeterministic space complexity classes in general. The problem Reachability can be solved by a nondeterministic machine using logarithmic space and, as we have seen, the computation of any nondeterministic machine M on a given input x can be represented by an instance of the reachability problem, by considering the configuration graph of M on x. These facts allow Reachability to stand in for *all* problems in NL.

In this section, we examine a specific algorithmic problem that captures nondeterministic time-bounded computation in the same sense that Reachability space-bounded computations. The problem is the *satisfiability of Boolean expressions.* We can take an arbitrary nondeterministic machine M, an input string x, and a time bound, and produce a Boolean expression whose satisfiability represents exactly the computation of M on x. Moreover, the Satisfiability problem is in NP, and as we shall see, it can be made to stand, in a precise sense, for all problems in NP.

Boolean Expressions

To start to make these notions precise, we define the collection of Boolean expressions to be the set of expressions formed from a given infinite set $X = \{x_1, x_2, \ldots\}$ of variables and the two constants `true` and `false` by means of the following rules:

- a constant or variable by itself is an expression;
- if ϕ is a Boolean expression, then so is $(\neg\phi)$;
- if ϕ and ψ are both Boolean expressions, then so are $(\phi \wedge \psi)$ and $(\phi \vee \psi)$.

If an expression contains no variables (that is, it is built up from just `true` and `false` using $\wedge$, $\vee$, and $\neg$), then it can be evaluated to either `true` or `false`. If an expression ϕ contains variables, then ϕ is not by itself true or false, Rather, we say that it is true or false for a *given* assignment of truth values. A *truth assignment* is just a function $T : X \to \{\mathtt{true}, \mathtt{false}\}$. We say that T *makes* ϕ *true* or T *satisfies* ϕ if, when we substitute $T(x)$ for each variable x in ϕ, we get an expression that evaluates to true. Conversely, if substituting $T(x)$ for each x in ϕ results in an expression that evaluates to `false`, we say that T *makes* ϕ *false* or T *does not satisfy* ϕ.
Examples

1. $(\mathtt{true} \vee \mathtt{false}) \wedge (\neg\mathtt{false})$
   evaluates to `true`.
2. $(x_1 \vee \mathtt{false}) \wedge ((\neg x_1) \vee x_2)$
   is satisfied by some truth assignments, but not by all.
3. $(x_1 \vee \mathtt{false}) \wedge (\neg x_1)$
   is not satisfied by any truth assignment.

4. $(x_1 \vee (\neg x_1)) \wedge \texttt{true}$

is satisfied by both possible truth assignments.

Evaluation With the definition of Boolean expressions established, we can begin to look at some algorithms for manipulating them. The first problem we look at is the evaluation problem. That is, given a Boolean expression *which does not contain any variables*, determine whether it evaluates to `true` or `false`. There is a deterministic Turing machine which can decide this, and which runs in time $O(n^2)$ on expressions of length n. The algorithm works by scanning the input, looking for subexpressions that match the left hand side of one of the rules below, and replacing it by the corresponding right hand side.

- $(\texttt{true} \vee \phi) \Rightarrow \texttt{true}$
- $(\phi \vee \texttt{true}) \Rightarrow \texttt{true}$
- $(\texttt{false} \vee \phi) \Rightarrow \phi$
- $(\texttt{false} \wedge \phi) \Rightarrow \texttt{false}$
- $(\phi \wedge \texttt{false}) \Rightarrow \texttt{false}$
- $(\texttt{true} \wedge \phi) \Rightarrow \phi$
- $(\neg\texttt{true}) \Rightarrow \texttt{false}$
- $(\neg\texttt{false}) \Rightarrow \texttt{true}$

Each scan of the input, taking $O(n)$ steps must find at least one subexpression matching one of the rules. This can be formally proved by an induction on the structure of expressions, but should be intuitively clear. Since the application of a rule always removes at least one symbol from the expression, we can conclude that we cannot have more than n successive rule applications, as otherwise we would be left with no symbols at all. Thus, the algorithm takes $O(n^2)$ steps overall.

Satisfiability

While a Boolean expression without variables can simply be evaluated, for an expression that contains variables, we can ask different questions. One question is whether such an expression is *satisfiable*. That is, is there a truth assignment that makes it true?

We define SAT to be the set of all satisfiable Boolean expressions. This language can be decided by a deterministic Turing machine which runs in time $O(n^2 2^n)$. The straightforward algorithm is to try all possible truth assignments. There are at most 2^n possible assignments, since there are at most n variables in the expression. For each one in turn, we can use the $O(n^2)$ algorithm described above to see if the resulting expression without variables is true.

A related problem is checking the validity of a Boolean expression. An expression is said to be *valid* if every possible assignment of truth values to its variables makes it true. We write VAL for the set of valid expressions. By an algorithm completely analogous to the one outlined for SAT above, we see that VAL is in time $O(n^2 2^n)$.

VAL and SAT are dual problems in the sense that a Boolean expression ϕ is valid if, and only if, the expression $\neg\phi$ is not satisfiable. This shows that an algorithm for deciding one language can easily be converted into an algorithm for deciding the other. No polynomial time algorithm is known for either problem, and it is widely believed that no such algorithm exists.

However, it is not difficult to establish that there is a nondeterministic algorithm that decides SAT in polynomial time. Indeed, time $O(n^2)$ suffices. The algorithm, in n nondeterministic steps, *guesses* a truth value for each variable. That is, for each variable in the expression, it makes a nondeterministic branch, where along one branch the variable is set to `true` and along the other it is set to `false`. Since the number of variables is no more than the length of the expression, n steps are sufficient for this process. Now, we have a Boolean expression ϕ and a truth assignment T for all its variables, so we can use the $O(n^2)$ deterministic algorithm described above to determine whether T satisfies ϕ. If there is any T that satisfies ϕ, some path through this nondeterministic computation will result in acceptance, so ϕ will be accepted.

Unfortunately, this approach does not yield an algorithm for VAL. To decide VAL, we have to determine, given ϕ, whether *every* possible truth assignment to the variables of ϕ will satisfy it. This *universal*, rather than *existential* requirement on computation paths does not sit well with the definition of acceptance by a nondeterministic machine. What we can establish is that VAL is in co-NP. There is a nondeterministic machine, running in $O(n^2)$ time, which can decide whether ϕ is *falsifiable*, that is to say, it is in the complement of VAL. This machine is just like the nondeterministic machine for SAT described above, accept it rejects whenever the latter accepts and *vice versa*.

SAT is typical of problems in NP. In general, a language in NP is characterised by a solution space which can be searched. For each possible input string x, there is a space of possible solutions to be searched. The number of such candidate solutions may be exponential in the length of x, but each solution can be represented by a string whose length is bounded by a polynomial p in the length of x. In particular, this means that the space can be searched by a backtracking search algorithm, where the depth of nesting of choice points is never greater than $p(|x|)$.

Another way of viewing the same class of problems is the generate and test paradigm. A language L in NP is one that can be solved by an algorithm with two components: a `Prover` and a `Verifier`. Given an input x, the `Prover` generates a proof V_x which demonstrates that x is in L. The `Verifier` then checks that the proof V_x is correct. As long as the length of V_x is bounded by a polynomial in the length of x *and* the `Verifier` runs in time polynomial in its input, the algorithm establishes that L is in NP (Fig. 3).

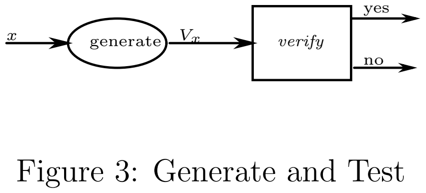

Figure 3: Generate and Test

SAT is typical of the problems in NP in the sense that the independent choices of truth assignments that can be made to the variables in a Boolean expression can encode the choices of any nondeterministic computation. It is in this sense that SAT "captures" nondeterministic time bounded computation, just as Reachability captures nondeterministic space bounded computation. In the following, we make this notion of encoding nondeterministic choices precise.

Reductions

The idea of a *reduction* is central to the study of computability. It is used to establish undecidability of languages. Formally, given two languages $L_1 \subseteq \Sigma_1^{|star}$, and $L_2 \subseteq \Sigma_2^\star$, a reduction of L_1 to L_2 is a *computable* function $f : \Sigma_1^\star \rightarrow \Sigma_2^\star$, such that for every string $x \in \Sigma_1^\star$, $f(x) \in L_2$ if, and only if, $x \in L_1$.

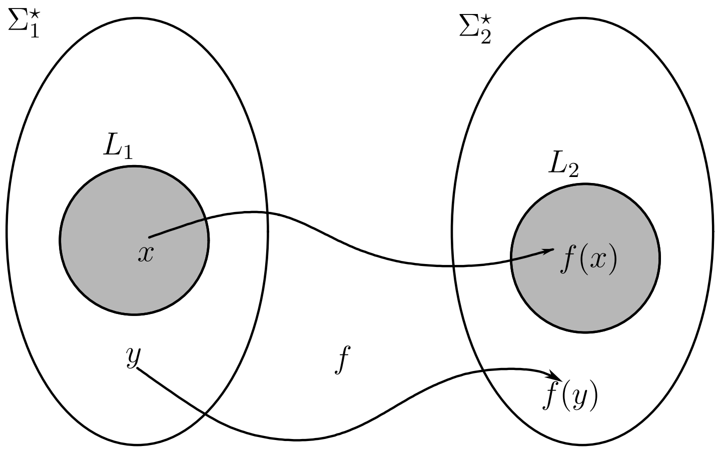

Figure 4: Reduction

In other words, every string in L_1 is mapped by f to a string in L_2, and every string that is not in L_1 is mapped to a string that is not in L_2. Note that there is no requirement for f to be surjective. In general, the range of f may be a small part of L_1, and a small part of its complement.

Reductions are useful in showing decidability (or, more usually, undecidability). If there is a reduction from a language L_1 to a language L_2, and L_2 is decidable, then we also know that L_1 is decidable. An algorithm to decide L_1 is obtained by computing, on any input x, $f(x)$ and then using the decision procedure for L_2 to determine a yes or no answer. If $f(x) \in L_2$, we know that $x \in L_1$ and if $f(x) \notin L_2$, we know that $x \notin L_1$. This argument is most useful in its contrapositive form: if there is a reduction from L_1 to L_2, and L_1 is known to be undecidable, then L_2 is undecidable as well. So, once we have proved one language (such as the Halting problem) to be undecidable, we can use reductions from it to prove other problems undecidable as well.

Resource Bounded Reductions When we are concerned about polynomial time computability rather than questions of decidability, we consider reductions that can be computed within bounded resources. To be precise, if f is a reduction from L_1 to L_2 and f is computable by an algorithm running in polynomial time, we say that L_1 is *polynomial time reducible* to L_2, which we write as

$$L_1 \leq_P L_2.$$

If $L_1 \leq_P L_2$, we can say that L_2 is at least as hard as L_1, at least in the sense of being polynomial time computable. In short, if $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$, then $L_1 \in \mathsf{P}$. This is for reasons analogous to those for decidability. We can compose the algorithm computing the reduction with the decision procedure for L_2 to get a polynomial time decision procedure for L_1. One point to be noted is that the string $f(x)$ produced by the reduction on f on input x must be bounded in length by a polynomial in the length of x, since it is computed by a polynomial time algorithm. This is why the decision procedure for L_2 which runs in polynomial time on its input $f(x)$ is still running in time polynomial in the length of x.

NP-Completeness

The usefulness of reductions is in allowing us to make statements about the *relative complexity* of problems, even when we are not able to prove absolute lower bounds. So, even if we do not know whether or not there is a polynomial time algorithm for L_2, if $L_1 \leq_P L_2$, we can say if there were one, there would also be one for L_1. In this sense of relative complexity, Stephen Cook first showed that there are problems in NP that are maximally difficult.
Definition

> *A language L is said to be* NP-hard *if for every language $A \in \mathsf{NP}$, $A \leq_P L$.*
>
> *A language L is* NP-complete *if it is in* NP *and it is* NP*-hard.*

What Cook showed was that the language SAT of satisfiable Boolean reductions is NP-complete. In this sense, it is as hard as any problem in NP. A polynomial time algorithm for SAT would yield a polynomial time algorithm for every problem in NP.

We have already seen that SAT is in NP. To prove that it is NP-complete, we need to show that for every language L in NP, there is a polynomial time reduction from L To SAT. Since L is in NP, we know that there is a nondeterministic machine $M = (K, \Sigma, s, \delta)$ and a polynomial p such that a string x is in L if, and only if, it is accepted by M within $p(|x|)$ steps. In what follows, we will assume, without loss of generality, that p is of the form n^k, where n is the length of x and k is a constant.

To establish the polynomial time reduction from L to SAT, we need to give, for each $x \in \Sigma^\star$, a Boolean expression $f(x)$ which is satisfiable if, and only if, there is an accepting computation of M on x. We construct $f(x)$ using the following variables:

$$\begin{array}{ll} S_{i,q} & \text{for each } i \leq n^k \text{ and } q \in K \\ T_{i,j,\sigma} & \text{for each } i,j \leq n^k \text{ and } \sigma \in \Sigma \\ H_{i,j} & \text{for each } i,j \leq n^k. \end{array}$$

The total number of variables is $|K|n^k + |\Sigma|n^{2k} + n^{2k}$. The intended reading of these variables is that $S_{i,q}$ will be true if the machine at time i is in state q; $T_{i,j,\sigma}$ will be set to true if at time i, the symbol at position j in the tape is σ; and $H_{i,j}$ indicates that at time i, the head is pointing at position j on the tape.

Of course, these meanings are not inherent in the symbols. We have to construct the expression $f(x)$ to enforce them. The intention is that the only way to consistently assign truth values to these variables is to encode a possible computation of the machine M. The expression $f(x)$ is built up as the *conjunction* of the following expressions:

$$S_{1,s} \wedge H_{1,1} \tag{1}$$

which asserts that at the beginning, the state is s and the head is at the beginning of the tape.

$$\bigwedge_i \bigwedge_j (H_{i,j} \rightarrow \bigwedge_{j' \neq j} (\neg H_{i,j'})) \tag{2}$$

which asserts that the head is never in two places at once. That is, if $H_{i,j}$ is true for any i and j, then $H_{i,j'}$ must be false for any other j'.

$$\bigwedge_q \bigwedge_i (S_{i,q} \rightarrow \bigwedge_{q' \neq q} (\neg S_{i,q'})) \tag{3}$$

which asserts that the machine is never in two states at once. That is, for each state q and each state i, if $S_{i,q}$ is true, then $S_{i,q'}$ must be false for all other q'.

$$\bigwedge_i \bigwedge_j \bigwedge_\sigma (T_{i,j,\sigma} \rightarrow \bigwedge_{\sigma' \neq \sigma} (\neg T_{i,j,\sigma'})) \tag{4}$$

which asserts that each tape cell contains only one symbol. In other words, if $T_{i,j,\sigma}$ is true for any i, j and σ, then $T_{i,j,\sigma'}$ must be false for all other σ'.

$$\bigwedge_{j \leq n} T_{1,j,x_j} \wedge \bigwedge_{n<j} T_{1,j,\sqcup} \tag{5}$$

where x_j denotes the jth symbol in the string x. This expression asserts that at time 1, the tape contains the string x in its first n cells, and is blank after that.

$$\bigwedge_i \bigwedge_j \bigwedge_{j' \neq j} \bigwedge_\sigma (H_{i,j} \wedge T_{i,j',\sigma}) \rightarrow T_{i+1,j',\sigma} \tag{6}$$

which asserts that the tape only changes under the head. That is, if the head at time i is at position j, and at the same time position j' on the tape (for some other j') contains σ, then position j' still contains σ at time $i+1$.

$$\bigwedge_i \bigwedge_j \bigwedge_\sigma \bigwedge_q (H_{i,j} \wedge S_{i,q} \wedge T_{i,j,\sigma}) \rightarrow \bigvee_\Delta (H_{i+1,j'} \wedge S_{i+1,q'} \wedge T_{i+1,j,\sigma'}) \tag{7}$$

where Δ is the set of all triples (q', σ', D) such that $((q,\sigma),(q',\sigma',D)) \in \delta$ and

$$j' = \begin{cases} j & \text{if } D = S \\ j-1 & \text{if } D = L \\ j+1 & \text{if } D = R. \end{cases}$$

This asserts that the change from time step i to $i+1$, for each i is according to the transition relation δ. That is, if at time i, the head position is j, the state is q, and the symbol at position j is σ, then the state q' and head position j' at time $i+1$, as well as the symbol at position j at time $i+1$ are obtained by one of the possible transitions allowed by δ.

$$\bigvee_i S_{i,\mathrm{acc}} \tag{8}$$

which asserts that at some time i, the accepting state is reached.

Conjunctive Normal Form

A Boolean expression is in *conjunctive normal form* (or CNF) if it is the conjunction of a set of *clauses*, each of which is the disjunction of a set of *literals*, each of these being either a *variable* or the *negation* of a variable. Every Boolean expression is equivalent to one in CNF. In fact, any expression can be turned into an equivalent expression in CNFby repeated application of DeMorgan's laws, the laws of distributivity (of $\vee$ over $\wedge$) and by the law of double negation (which says that $\neg\neg\phi$ is equivalent to ϕ. There is, therefore, an algorithm for converting any Boolean expression into an equivalent expression in CNF. This is not, however, a polynomial time algorithm. We can prove that it requires exponential time (a rare example of a real lower bound result). This is because there are (for arbitrarily large n) expressions ϕ of length n such that the shortest CNFexpression equivalent to ϕ has length $\Omega(2^n)$.

However, if we consider the reduction constructed above from any language L in NP to SAT, and take the Boolean expressions that result from the reduction, then there is a polynomial time algorithm that will convert them into equivalent CNFexpressions. This is because the formulas are almost in CNFalready. In particular, the expression is a conjunction of expressions, of which (1) is just a conjunction of literals (and is therefore in CNF). The expressions in (2), (3) and (4) can be easily converted into CNFby distributing the implication over the innermost conjunction. For example, (2) can be rewritten as

$$\bigwedge_i \bigwedge_j \bigwedge_{j' \neq j} (H_{i,j} \rightarrow (\neg H_{i,j'})).$$

This is now in CNF(recall that $\phi \rightarrow \psi$ is just shorthand for $\neg\phi \vee \psi$). Note also that while (2) contains only one occurrence of the variable $H_{i,j}$ for each i and j, the CNFversion above has $n^k - 1$ occurrences of each $H_{i,j}$. This is however, a fixed polynomial increase. Similarly, (5) and (6) are already in CNF. The expression (7) requires a bit more work, but it is not too difficult, and is left here as an exercise.

We can conclude that, for each language L in NP, there is, in fact, a polynomial time computable function f such that $f(x)$ is a CNFexpression for all x, and $f(x)$ is satisfiable if, and only if, $x \in L$. In other words, if we define CNF-SAT to be the collection of all satisfiable CNFexpressions, then we can say that we have shown that CNF-SAT is NP-complete.

We define a further restriction on our expressions. A Boolean expression ϕ is said to be in 3CNF if it is in CNF, i.e. $\phi \equiv C_1 \wedge \ldots \wedge C_m$, and each clause C_i is the disjunction of no more than 3 literals. We also define 3SAT to be the set of those expressions in 3CNF that are satisfiable.

While it is not the case that every Boolean expression is equivalent to one in 3CNF, what we can say is that for every CNFexpression ϕ, there is an expression ϕ' in 3CNF so that ϕ' is satisfiable if, and only if, ϕ is. Moreover, there is an algorithm, running in polynomial time, which will convert ϕ to ϕ'. We will illustrate this with an example. Suppose we have a clause C with four literals

$$C \equiv (l_1 \vee l_2 \vee l_3 \vee l_4).$$

Introducing new variables n_1 and n_2, we can write down an expression ψ in 3CNF

$$\psi \equiv (l_1 \vee l_2 \vee n_1) \wedge (\neg n_1 \vee l_3 \vee n_2) \wedge (\neg n_2 \vee l_4).$$

This expression is not equivalent to C because, for one thing, it contains variables that are not in C. But, ψ is satisfiable if, and only if, C is. The idea can be easily generalised to clauses with any number of variables. Moreover, we can verify that the number of new variables and clauses introduced is no more than the number of literals in the clause being replaced. This ensures that the conversion can be carried out in polynomial time.

What we can conclude from this is that there is a polynomial time computable reduction from CNF-SAT to 3SAT. This can be combined with the fact that CNF-SAT is NP-complete to show that 3SAT is also NP-complete.