This lecture returns to the topic of propositional logic. Whereas in Lecture 1 we studied this topic as a way of understanding proper reasoning and proofs, we now study it from a formal and then a computational perspective. Eventually we will find ways to manipulate logical expressions algorithmically so as to solve hard problems automatically. In so doing, we will come across some fundamental notions of complexity. We will also have a pretty good Minesweeper program.

Boolean expressions and Boolean functions

Just as arithmetic deals with all the mathematics that arises from operations on numbers, the study of Boolean functions deals with all the mathematics that arises from operations on the Boolean values true and false, which we will denote by $T$ and $F$. (1 and 0 are also commonly used.) Despite there being just two values, lots of interesting mathematics arises.

We begin with a formal constructive definition of the set of Boolean expressions or (Boolean formulæ or propositional logic expressions or propositional sentences). Notice that this is very similar to the definition of binary trees, etc. It’s more complex because the set is more complex.

Definition 7.1 (Boolean expressions):

\[
T \in B \text{ and } F \in B \\
\forall X \in \mathcal{X} \ [X \in B] \\
\forall B \in \mathcal{B} \ [\neg B \in \mathcal{B}] \\
\forall B_1, B_2 \in \mathcal{B} \ [B_1 \land B_2 \in \mathcal{B}] \\
\forall B_1, B_2 \in \mathcal{B} \ [B_1 \lor B_2 \in \mathcal{B}] \\
\forall B_1, B_2 \in \mathcal{B} \ [B_1 \implies B_2 \in \mathcal{B}] \\
\forall B_1, B_2 \in \mathcal{B} \ [B_1 \iff B_2 \in \mathcal{B}]
\]

To prove something about all Boolean expressions, we will need the following induction principle:

Axiom 7.1 (Induction over Boolean expressions):

For any property $P$, if $P(T)$ and $P(F)$ and $\forall X \in \mathcal{X} \ P(X)$ and $\forall B \in \mathcal{B} \ [P(B) \implies P(\neg B)]$ and $\forall B_1, B_2 \in \mathcal{B} \ [P(B_1) \land P(B_2) \implies P(B_1 \land B_2)]$ and $\forall B_1, B_2 \in \mathcal{B} \ [P(B_1) \land P(B_2) \implies P(B_1 \lor B_2)]$ and $\forall B_1, B_2 \in \mathcal{B} \ [P(B_1) \land P(B_2) \implies P(B_1 \implies B_2)]$ and $\forall B_1, B_2 \in \mathcal{B} \ [P(B_1) \land P(B_2) \implies P(B_1 \iff B_2)]$ then $\forall B \in \mathcal{B} \ P(B)$. 
Some useful terminology: an expression of the form $B_1 \land B_2$ is called a **conjunction**; $B_1$ and $B_2$ are its **conjuncts**. An expression of the form $B_1 \lor B_2$ is called a **disjunction**; $B_1$ and $B_2$ are its **disjuncts**.

A Boolean expression on $\{X_1, \ldots, X_n\}$ has a **truth value** for any complete **assignment** of $T/F$ to $\{X_1, \ldots, X_n\}$. (Complete assignments are also called **models**, as we will see later; an assignment for which an expression has value $T$ is called a **model of that expression**.) Any Boolean expression $B$ therefore represents a function that maps $n$-tuples of Boolean values into a Boolean value:

$$B(X_1, \ldots, X_n) : \{T,F\}^n \mapsto \{T,F\}$$

(Here the notation $\{T,F\}^n$ means the set $\{T,F\}$ Cartesian-producted with itself $n-1$ times.) For example, a Boolean expression $X_1 \land X_2$ on the set of symbols $\{X_1, X_2\}$ maps pairs of Boolean values into a Boolean value that is the “and” of the two inputs.

The rules for evaluation of an expression with respect to an assignment are given by the truth tables for all the Boolean operators (see Table 1). That is, every symbol can be replaced by its value according to the assignment, then the expression can be evaluated “bottom-up” just like any arithmetic expression. For example, with the assignment $\{A = T, B = F\}$, the expression $(A \land (A \implies B)) \implies B$ becomes

$$[(T \land (T \implies F)) \implies F] = [(T \land F) \implies F] = [F \implies F] = T$$

We can also provide a top-down recursive definition of the truth value of an expression. Let $M$ be an assignment, and let $X_M$ denote the value of $X$ according to $M$. Then

$$\forall X \in X \ [eval(X,M) = X_M]$$
$$\forall B \in B \ [eval(\neg B,M) = \neg(eval(B,M))]$$
$$\forall B_1, B_2 \in B \ [eval(B_1 \land B_2, M) = eval(B_1,M) \land eval(B_2,M)]$$

etc.

Notice that in the above evaluation rules we use the Boolean operators $\neg, \land$, and so on as functions operating on Boolean values rather than as logical operators in the defining propositions.

Given a precise definition of what expressions mean, we can define the following useful notion:

**Definition 7.2 (Logical equivalence):**

Two Boolean expressions on the same set of variables are **logically equivalent** iff they return the same truth value for every possible assignment of values to the variables; that is, they represent the same Boolean function.

---

1Note that there are several different arithmetic expressions that represent the same arithmetic function! For example, $(x + y + x - x)/1$ is the same function of $x, y$ as $x + y$ is. Note also that we use the word “function” here in the mathematical sense. You can read more about functions in Rosen, Ch.1.6. For now, it’s just something that maps each possible input value to a specific output value.
We’ll use the symbol $\equiv$ as a shorthand for “is logically equivalent to.” Some obvious equivalences, all of which can be checked using truth tables:

- $(A \land B) \equiv (B \land A)$ (commutative)
- $(A \lor B) \equiv (B \lor A)$ (commutative)
- $((A \land B) \land C) \equiv (A \land (B \land C))$ (associative)
- $((A \lor B) \lor C) \equiv (A \lor (B \lor C))$ (associative)
- $(A \implies B) \equiv (\neg A \lor B)$
- $(A \iff B) \equiv ((A \implies B) \land (B \implies A))$
- $(A \land B) \equiv (\neg A \lor \neg B)$ (de Morgan)
- $\neg (A \lor B) \equiv (\neg A \land \neg B)$ (de Morgan)
- $(A \land B) \equiv (A \lor B)$ (distributivity)
- $(A \lor (B \land C)) \equiv ((A \lor B) \land (A \lor C))$ (distributivity)
- $(A \land (B \lor C)) \equiv ((A \land B) \lor (A \land C))$ (distributivity)

Because $\land$ and $\lor$ are associative, we can write expressions such as $A \land B \land C$ and $A \lor B \lor C$—that is, omitting the parentheses that would normally be required—without fear of ambiguity. Given commutativity also, these expressions can be thought of as conjunction or disjunction applied to sets of expressions.

From the above set of equivalences, we can see (at least informally) that every Boolean expression can be written using just the operators $\land$ and $\lor$. We can replace $\iff$ by $\implies$ and $\land$. Then replace $\implies$ by $\lor$ and $\neg$. Then replace $\lor$ by $\land$ and $\neg$. (A similar argument shows that $\lor$ and $\neg$ also suffice.) This informal argument can be made rigorous by applying the induction principle for Boolean expressions. Later in this lecture we will show how to use the induction principle to prove a stronger result: that every Boolean expression can be rewritten using just a single logical operator.

### How many Boolean functions?

So far, we have been somewhat careful in distinguishing between Boolean expressions and Boolean functions. One way to think about this difference is to imagine that you are a circuit designer looking to implement a certain behaviour—say bitwise addition—using logical gates that correspond to the Boolean operators The function is the “input–output mapping” while the expression is a specific “circuit” that implements the function. One of the major activities of circuit designers is choosing a Boolean expression that implements a given function most efficiently, out of all the Boolean expressions that implement the function.

If you still think that expressions and functions are really the same thing, let’s count them. First, how many Boolean expressions are there on $n$ variables? Obviously, infinitely many!

Now let’s count the Boolean functions. A function is defined by its input–output mapping. For a Boolean function on $n$ variables, there are $2^n$ possible inputs. Any given function is therefore defined by a truth table with $2^n$ rows. Each row could be $T$ or $F$, so there are $2^{2^n}$ different ways to fill out the table; that is, $2^{2^n}$ Boolean functions of $n$ variables.

$2^{2^n}$ is easy to say, but frightening when you look at what it means:

- $2^0 = 2^1 = 2$ functions of no variables—these are $T$ and $F$.
- $2^1 = 2^2 = 4$ functions of one variable (What are they?).
- $2^2 = 2^4 = 16$ functions of two variables.
- $2^3 = 2^8 = 256$ functions of three variables.
\[ 2^{2^4} = 2^{16} = 65,536 \text{ functions of four variables.} \]
\[ 2^{2^3} = 2^{32} = 4,294,967,296 \text{ functions of five variables.} \]
\[ 2^{2^6} = 2^{64} = 18,446,744,073,709,551,616 \text{ functions of six variables.} \]

The large size of the space of Boolean functions has an important corollary: almost every Boolean function requires an exponentially large Boolean expression to represent it!

For a circuit designer, this could be depressing! Certainly, some Boolean functions can be represented compactly—for example, the function that returns \( T \) only if all the inputs are \( T \) can be represented by \( X_1 \wedge \ldots \wedge X_n \), i.e., an expression of size \( O(n) \). So how do we know this can’t always be done? The answer is just a matter of counting. Let’s make the claim more precise:

**Theorem 7.1:** The fraction of Boolean functions of \( n \) variables that can be represented in any fixed encoding using at most \( 2^{n-1} \) bits is less than one in \( 2^{2n-1} \) of all such functions.

Notice that \( 2^{n-1} \) is half the number of bits one would need to simply write out the full truth table values for each function, so we’re not asking for much compactness here!

**Proof:** This is really a proof sketch, because we do not have proper axioms for encoding using bit strings. But the idea is simple: any expression can be reduced to a string of bits—e.g., we could simply encode the Boolean expression in ASCII. The most compact possible encoding has every possible bit string representing a different expression; for example, we could have the bit string 0 represent the expression \( T \), 1 represent \( X_1 \), 10 represent \( X_2 \), and so on. There are exactly \( 2^i \) bit strings of length \( i \), so there are exactly \( 2^0 + 2^1 + \cdots + 2^{n-1} = 2^{n-1} + 1 \) strings of length less than or equal to \( 2^{n-1} \). (We’re being generous here because we’re not charging for the “end of string” marker that is required if one uses bit strings of different lengths.) Now the fraction of the total number of functions is given by

\[
\frac{2^{2^{n-1}+1} - 1}{2^{2^n}} = \frac{1}{2^{2^{n-1}} - 1} = \frac{1}{2^{2^{n-1}} - 1}
\]

\( \square \)

**A minimalist representation**

Besides finding a compact representation for a Boolean function, circuit designers often prefer expressions that use only a single type of Boolean operator—preferably one that corresponds to a simple transistor circuit on a chip. The “nand” Boolean operator, written as \( A \mid B \) and equivalent to \( \neg(A \wedge B) \), is easily implemented on a chip. We also have the following interesting fact:

**Theorem 7.2:** For every Boolean expression, there is a logically equivalent expression using only the \( \mid \) operator.

We will do a full inductive proof (or some of one anyway) to show you what an induction over Boolean expressions looks like.

**Proof:** The proof is by induction over Boolean expressions on the variables \( X \). Let \( P(B) \) be the proposition that \( B \) can be expressed using only the \( \mid \) operator.

- **Base case:** prove \( P(T) \), \( P(F) \), and \( \forall X \in X \) \( P(X) \).
  These are true since the expressions require no operators.

- **Inductive step (\( \neg \)):** prove \( \forall B \in B \) \( [P(B) \implies P(\neg B)] \).
1. The inductive hypothesis states that $B$ can be expressed using only $\mid$. Let $NF(B)$ (NAND-form of $B$) be such an expression.

2. To prove: $\neg B$ can be expressed using only $\mid$.

3. From the definition of $\mid$, we have

$$\neg B \equiv (B \mid B)$$

$$\equiv (NF(B) \mid NF(B)) \text{ by the induction hypothesis}$$

4. Hence, there is an expression equivalent to $\neg B$ that contains only $\mid$.

**Inductive step ($\land$):** prove $\forall B_1, B_2 \in B \ [P(B_1) \land P(B_2) \implies P(B_1 \land B_2)]$.

1. The inductive hypothesis states that $B_1$ and $B_2$ can be expressed using only $\mid$. Let $NF(B_1)$ and $NF(B_2)$ be such expressions.

2. To prove: $B_1 \land B_2$ can be expressed using only $\mid$.

3. Now $\land$ is the negation of $\mid$, so we have

$$(B_1 \land B_2) \equiv \neg(B_1 \mid B_2) \equiv ((B_1 \mid B_2) \mid (B_1 \mid B_2))$$

$$\equiv ((NF(B_1) \mid NF(B_2)) \mid (NF(B_1) \mid NF(B_2))) \text{ by the induction hypothesis}$$

4. Hence, there is an expression equivalent to $(B_1 \land B_2)$ that contains only $\mid$.

**The remaining steps (for $\lor$, $\implies$, $\iff$) are left as an exercise.**

Hence, by the induction principle for Boolean expressions, for every Boolean expression, there is a logically equivalent expression using only the $\mid$ operator.

Notice the crucial use of the induction hypothesis in this proof! For example, in the proof for $\neg B$, the expression that contains only $\mid$ is the expression $NF(B) \mid NF(B)$. The expression $B \mid B$ could contain anything at all, since $B$ is just an arbitrary Boolean expression.

Notice that, as is often the case with inductive proofs, the proof gives a recursive conversion algorithm directly. Conversion to NAND-form can, however, give a very large expansion of the expression.

The steps omitted in the proof above can be done by further equivalences involving $\mid$. A similar proof, using just the standard equivalences given earlier, establishes that every Boolean expression can be written using $\land$ and $\neg$ (or using $\lor$ and $\neg$). Essentially, we use the equivalence that replaces $\iff$ by $\implies$ and $\land$; and the equivalence that replaces $\implies$ by $\lor$ and $\neg$; and the equivalence that replaces $\lor$ by $\land$ and $\neg$.

**Normal forms**

A **normal form** for an expression is usually a subset of the standard syntax of expressions, such that either every expression can be rewritten in the normal form, or that expressions in the normal form have certain interesting properties. By restricting the form, we can often find simple and/or efficient algorithms for manipulating the expressions.

The first normal form we will study is called **disjunctive normal form** or DNF. In DNF, every expression is a disjunction of conjunctions of literals. A **literal** is a Boolean variable or its negation. For example, the following expression is in DNF:

$$(A \land \neg B) \lor (B \land \neg C) \lor (A \land \neg C \land \neg D)$$
Notice that DNF is generous with operators but very strict about nesting: a single level of disjunction and a single level of conjunction within each disjunct. DNF is a complete normal form, that is, we can establish the following:

**Theorem 7.3**: For every Boolean expression, there is a logically equivalent DNF expression.

**Proof**: Given a Boolean expression $B$, consider its truth-table description. In particular, consider those rows of the truth table where the value of the expression is $T$. Each such row is specified by a conjunction of literals, one literal for each variable. The disjunction of these conjunctions is logically equivalent to $B$. □

DNF is very commonly used in circuit design. Note that the DNF expression obtained directly from the truth table has as many disjuncts as there are $T$’s in the truth table’s value column. Logic minimization deals with methods to reduce the size of such expressions by eliminating and combining disjuncts.

In the area of logical reasoning systems, conjunctive normal form (CNF) is much more commonly used. In CNF, every expression is a conjunction of disjunctions of literals. A disjunction of literals is called a clause. For example, the following expression is in CNF:

$$\neg(A \lor B) \land (\neg B \lor \neg C) \land (A \lor C \lor D)$$

We can easily show the following result:

**Theorem 7.4**: For every Boolean expression, there is a logically equivalent CNF expression.

**Proof**: Any Boolean expression $B$ is logically equivalent to the conjunction of the negation of each row of its truth table with value $F$. The negation of each row is the negation of a conjunction of literals, which (by de Morgan’s law) is equivalent to a disjunction of the negations of literals, which is equivalent to a disjunction of literals. □

Another way to find a CNF expression logically equivalent to any given expression is through a recursive transformation process. This does not require constructing the truth table for the expression, and can result in much smaller CNF expressions.

The steps are as follows:

1. Eliminate $\iff$, replacing $A \iff B$ with $(A \implies B) \land (B \implies A)$.
2. Eliminate $\implies$, replacing it $A \implies B$ with $\neg A \lor B$.
3. Now we have an expression containing only $\land$, $\lor$, and $\neg$. The conversion of $\neg CNF(A)$ into CNF, where $CNF(A)$ is the CNF equivalent of expression $A$, is extremely painful. Therefore, we prefer to “move $\neg$ inwards” using the following operations:
   
   $$\neg(\neg A) \equiv A$$
   $$\neg(A \land B) \equiv (\neg A \lor \neg B) \text{ (de Morgan)}$$
   $$\neg(A \lor B) \equiv (\neg A \land \neg B) \text{ (de Morgan)}$$

   Repeated application of these operations results in an expression containing nested $\land$ and $\lor$ operators applied to literals. (This is an easy proof by induction, very similar to the NAND proof.)

4. Now we apply the distributivity law, distributing $\land$ over $\lor$ wherever possible, resulting in a CNF expression.

We will now prove formally that the last step does indeed result in a CNF expression, as stated.
Theorem 7.5: Let B be any Boolean expression constructed from the operators \(\land, \lor, \text{ and } \neg\), where \(\neg\) is applied only to variables. Then there is a CNF expression logically equivalent to B.

Obviously, we could prove this simply by appealing to Theorem 6.4; but this would leave us with an algorithm involving a truth-table construction, which we wish to avoid. Let’s see how to do it recursively.

Proof: The proof is by induction over Boolean expressions on the variables X. Let \(P(B)\) be the proposition that \(B\) can be expressed in CNF; we assume \(B\) contains only \(\land, \lor, \text{ and } \neg\), where \(\neg\) is applied only to variables.

- Base case: prove \(P(T), P(F), \text{ and } \forall X \in X P(X) \text{ and } \forall X \in X P(\neg X)\).
  These are true since a conjunction of one disjunction of one literal is equivalent to the literal.

- Inductive step (\(\land\)): prove \(\forall B_1, B_2 \in B [P(B_1) \land P(B_2) \implies P(B_1 \land B_2)]\).
  1. The inductive hypothesis states that \(B_1\) and \(B_2\) can be expressed in CNF. Let \(CNF(B_1)\) and \(CNF(B_2)\) be two such expressions.
  2. To prove: \(B_1 \land B_2\) can be expressed in CNF.
  3. By the inductive hypothesis, we have
     \[
     B_1 \land B_2 \equiv CNF(B_1) \land CNF(B_2) \\
     \equiv (C_1^1 \land \ldots \land C_1^n) \land (C_2^1 \land \ldots \land C_2^n) \quad (C_i^j \text{ are clauses}) \\
     \equiv (C_1^1 \land \ldots \land C_1^n \land C_2^1 \land \ldots \land C_2^n)
     \]
  4. Hence, \(B_1 \land B_2\) is equivalent to an expression in CNF.

- Inductive step (\(\lor\)): prove \(\forall B_1, B_2 \in B [P(B_1) \lor P(B_2) \implies P(B_1 \lor B_2)]\).
  1. The inductive hypothesis states that \(B_1\) and \(B_2\) can be expressed in CNF. Let \(CNF(B_1)\) and \(CNF(B_2)\) be two such expressions.
  2. To prove: \(B_1 \lor B_2\) can be expressed in CNF.
  3. By the inductive hypothesis, we have
     \[
     B_1 \lor B_2 \equiv CNF(B_1) \lor CNF(B_2) \\
     \equiv (C_1^1 \land \ldots \land C_1^n) \lor (C_2^1 \land \ldots \land C_2^n) \quad (C_i^j \text{ are clauses}) \\
     \equiv (C_1^1 \lor C_2^1) \land (C_1^1 \lor C_2^1) \land \ldots \land (C_1^n \lor C_2^n) \land (C_1^n \lor C_2^n)
     \]
  4. By associativity of \(\lor\), each expression of the form \((C_i^j \lor C_i^j)\) is equivalent to a single clause containing all the literals in the two clauses.
  5. Hence, \(B_1 \lor B_2\) is equivalent to an expression in CNF.

Hence, any Boolean expression constructed from the operators \(\land, \lor, \text{ and } \neg\), where \(\neg\) is applied only to variables, is logically equivalent to an expression in CNF. \(\square\)

This process therefore “flattens” the logical expression, which might have many levels of nesting, into two levels. In the process, it can enormously enlarge it; the distributivity step converting DNF into CNF can give an exponential blowup when applied to nested disjunctions (see below). As with the conversion to NAND-form, the proof gives a recursive conversion algorithm directly.

+ ++
Direct conversion between CNF and DNF

Let’s look briefly at direct conversion of DNF into CNF and CNF into DNF. We’ll be using the distributivity rules; since these are symmetrical with respect to $\land$ and $\lor$, whatever we say about one direction applies to the other. So let’s look at DNF into CNF.

Let’s start with a very simple case:

$$(A \land B) \lor (C \land D)$$

$$\equiv (A \lor (C \land D)) \land (B \lor (C \land D))$$

$$\equiv (A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D)$$

Now let’s add one further term:

$$(A \land B) \lor (C \land D) \lor (E \land F)$$

$$\equiv [(A \lor C) \land (A \lor D)] \land (B \lor C) \land (B \lor D) \lor (E \land F)$$

$$\equiv \{(A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D)\} \lor (E \land F)$$

$$\land \{(A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D)\} \lor (E \land F)$$

$$\equiv (A \lor C \lor E) \land (A \lor D \lor E) \land (B \lor C \lor E) \land (B \lor D \lor E)$$

$$\land (A \lor C \lor F) \land (A \lor D \lor F) \land (B \lor C \lor F) \land (B \lor D \lor F)$$

The pattern becomes clear: the CNF clauses consist of every possibly $k$-tuple of literals taken, one each, from the $k$ terms of the DNF. Thus, we conjecture that if a DNF expression has $k$ terms, each containing $l$ literals, the equivalent CNF obtained by distributivity will have $l^k$ clauses, each containing $k$ literals. (This can be verified by induction.) Thus, there can be an exponential blowup in converting from DNF to CNF; and, by symmetry, in converting from CNF into DNF. We will see in the next lecture that it is almost inevitable that some small CNF expressions have a smallest DNF equivalent that is exponentially larger.