CS 70 Spring 2005 — Discrete Mathematics for CS — Clancy/Wagner

Notes 7

This lecture returns to the topic of propositional logic. Whereas in Lecture Notes 1 we studied this topic as a way of understanding proper reasoning and proofs, we now study it from 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

BOOLEAN VALUES

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.

BOOLEAN EXPRESSIONS

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.

PROPOSITION SYMBOLS

Let $\mathbf{X}$ be the set of **proposition symbols** $\{X_1, \ldots X_n\}$ (also called **Boolean variables**), and $\mathbf{B}$ be the set of Boolean expressions on $\mathbf{X}$. (Notice that we underline the expressions themselves to avoid confusion with the logical notation surrounding them. We won't do this from now on, but you might want to do it mentally if you find yourself getting confused.)

Definition 7.1 (Boolean expressions):

$$\underline{T} \in \mathbf{B} \text{ and } \underline{F} \in B$$
$$\forall X \in \mathbf{X}\ [\underline{X} \in \mathbf{B}]$$
$$\forall B \in \mathbf{B}\ [\underline{\neg B} \in \mathbf{B}]$$
$$\forall B_1, B_2 \in \mathbf{B}\ [\underline{B_1 \wedge B_2} \in \mathbf{B}]$$
$$\forall B_1, B_2 \in \mathbf{B}\ [\underline{B_1 \vee B_2} \in \mathbf{B}]$$
$$\forall B_1, B_2 \in \mathbf{B}\ [\underline{B_1 \implies B_2} \in \mathbf{B}]$$
$$\forall B_1, B_2 \in \mathbf{B}\ [\underline{B_1 \Leftrightarrow B_2} \in \mathbf{B}]$$

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 $\forall X \in \mathbf{X}\ P(X)$ and
$\forall B \in \mathbf{B}\ [P(B) \implies P(\neg B)]$ and
$\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \wedge B_2)]$ and
$\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \vee B_2)]$ and
$\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \implies B_2)]$ and
$\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \Leftrightarrow B_2)]$
then $\forall B \in \mathbf{B}\ P(B)$.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \implies Q$	$P \Leftrightarrow Q$
$False$	$False$	$True$	$False$	$False$	$True$	$True$
$False$	$True$	$True$	$False$	$True$	$True$	$False$
$True$	$False$	$False$	$False$	$True$	$False$	$False$
$True$	$True$	$False$	$True$	$True$	$True$	$True$

Table 1: Truth tables for all the Boolean operators

CONJUNCTION
DISJUNCTION

Some useful terminology: an expression of the form $B_1 \wedge B_2$ is called a **conjunction**; B_1 and B_2 are its **conjuncts**. An expression of the form $B_1 \vee B_2$ is called a **disjunction**; B_1 and B_2 are its **disjuncts**.

BOOLEAN FUNCTION

So much for the "syntax" of Boolean expressions. What do they mean? We can think of a Boolean expression as a representation of a **Boolean function**, in much the same way as arithmetic expressions such as $x+y$ represent the addition function.[1]

TRUTH VALUE
ASSIGNMENT
MODELS

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 $X_1 \wedge X_2$ 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 $(A \wedge (A \implies B)) \implies B$ becomes

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

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

$$\forall X \in \mathbf{X} \;\; [eval(X,M) = X_M]$$
$$\forall B \in \mathbf{B} \;\; [eval(\neg B,M) = \neg(eval(B,M))]$$
$$\forall B_1,B_2 \in \mathbf{B} \;\; [eval(B_1 \wedge B_2,M) = eval(B_1,M) \wedge eval(B_2,M)]$$
etc.

Notice that in the above evaluation rules we use the Boolean operators $\neg$, $\wedge$, 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):

LOGICALLY
EQUIVALENT

Two Boolean expressions on the same set of variables are **logically equivalent** iff they return the

[1] Note 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.

same truth value for every possible assignment of values to the variables; that is, they represent the same Boolean function.

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:

$(A \wedge B) \equiv (B \wedge A)$ (commutative)
$(A \vee B) \equiv (B \vee A)$ (commutative)
$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$ (associative)
$((A \vee B) \vee C) \equiv (A \vee (B \vee C))$ (associative)
$(A \implies B) \equiv (\neg A \vee B)$
$(A \Leftrightarrow B) \equiv ((A \implies B) \wedge (B \implies A))$
$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ (de Morgan)
$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ (de Morgan)
$(A \wedge B) \equiv \neg(\neg A \vee \neg B)$
$(A \vee B) \equiv \neg(\neg A \wedge \neg B)$
$(A \vee (B \wedge C)) \equiv ((A \vee B) \wedge (A \vee C))$ (distributivity)
$(A \wedge (B \vee C)) \equiv ((A \wedge B) \vee (A \wedge C))$ (distributivity)

Because $\wedge$ and $\vee$ are associative, we can write expressions such as $A \wedge B \wedge C$ and $A \vee B \vee C$—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 $\wedge$ and $\neg$. We can replace $\Leftrightarrow$ by $\implies$ and $\wedge$. Then replace $\implies$ by $\vee$ and $\neg$. Then replace $\vee$ by $\wedge$ and $\neg$. (A similar argument shows that $\vee$ 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.

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 $A|B$ and equivalent to $\neg(A \wedge B)$, is easily implemented on a chip. We also have the following interesting fact:

Theorem 7.1: *For every Boolean expression, there is a logically equivalent expression using only the $|$ 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 $\mathbf{X}$. Let $P(B)$ be the proposition that B can be expressed using only the $|$ operator.

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

- Inductive step ($\neg$): prove $\forall B \in \mathbf{B}\ [P(B) \implies P(\neg B)]$.

1. The inductive hypothesis states that B can be expressed using only $|$. Let $NF(B)$ (NAND-form of B) be such an expression.
2. To prove: $\neg B$ can be expressed using only $|$.
3. From the definition of $|$, we have

$$\begin{aligned} \neg B &\equiv (B|B) \\ &\equiv (NF(B)|NF(B)) \text{ by the induction hypothesis} \end{aligned}$$

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

- Inductive step ($\wedge$): prove $\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \wedge B_2)]$.

1. The inductive hypothesis states that B_1 and B_2 can be expressed using only $|$. Let $NF(B_1)$ and $NF(B_2)$ be such expressions.
2. To prove: $B_1 \wedge B_2$ can be expressed using only $|$.
3. Now $\wedge$ is the negation of $|$, so we have

$$\begin{aligned} (B_1 \wedge B_2) &\equiv \neg(B_1|B_2) \equiv ((B_1|B_2)|(B_1|B_2)) \\ &\equiv ((NF(B_1)|NF(B_2))|(NF(B_1)|NF(B_2))) \text{ by the induction hypothesis} \end{aligned}$$

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

- The remaining steps (for $\vee$, $\implies$, $\Leftrightarrow$) 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 $|$ operator. $\square$

Notice the crucial use of the induction hypothesis in this proof! For example, in the proof for $\neg B$, the expression that contains only $|$ is the expression $NF(B)|NF(B)$. The expression $B|B$ 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 $|$. A similar proof, using just the standard equivalences given earlier, establishes that every Boolean expression can be written using $\wedge$ and $\neg$ (or using $\vee$ and $\neg$). Essentially, we use the equivalence that replaces $\Leftrightarrow$ by $\implies$ and $\wedge$; and the equivalence that replaces $\implies$ by $\vee$ and $\neg$; and the equivalence that replaces $\vee$ by $\wedge$ and $\neg$.

Normal forms

NORMAL FORM

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.

DISJUNCTIVE NORMAL FORM

LITERAL

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:

$$(A \wedge \neg B) \vee (B \wedge \neg C) \vee (A \wedge \neg C \wedge \neg D)$$

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:

COMPLETE

Theorem 7.2: *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 Ts in the truth table's value column. **Logic minimization** deals with methods to reduce the size of such expressions by eliminating and combining disjuncts.

LOGIC MINIMIZATION

CONJUNCTIVE NORMAL FORM

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:

CLAUSE

$$(\neg A \vee B) \wedge (\neg B \vee \neg C) \wedge (A \vee C \vee D)$$

We can easily show the following result:

Theorem 7.3: *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 $\Leftrightarrow$, replacing $A \Leftrightarrow B$ with $(A \implies B) \wedge (B \implies A)$.

2. Eliminate $\implies$, replacing it $A \implies B$ with $\neg A \vee B$.

3. Now we have an expression containing only $\wedge$, $\vee$, 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:

   $\neg(\neg A) \equiv A$
   $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ (de Morgan)
   $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ (de Morgan)

   Repeated application of these operations results in an expression containing nested $\wedge$ and $\vee$ 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 $\wedge$ over $\vee$ 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 7.4: *Let B be any Boolean expression constructed from the operators $\wedge$, $\vee$, 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 $\mathbf{X}$. Let $P(B)$ be the proposition that B can be expressed in CNF; we assume B contains only $\wedge$, $\vee$, and $\neg$, where $\neg$ is applied only to variables.

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

- Inductive step ($\wedge$): prove $\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \wedge B_2)]$.

  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: $B_1 \wedge B_2$ can be expressed in CNF.
  3. By the inductive hypothesis, we have

  $$\begin{aligned} B_1 \wedge B_2 &\equiv CNF(B_1) \wedge CNF(B_2) \\ &\equiv (C_1^1 \wedge \ldots \wedge C_1^m) \wedge (C_2^1 \wedge \ldots \wedge C_2^n) \quad (C_j^i\text{s are clauses}) \\ &\equiv (C_1^1 \wedge \ldots \wedge C_1^m \wedge C_2^1 \wedge \ldots \wedge C_2^n) \end{aligned}$$

  4. Hence, $B_1 \wedge B_2$ is equivalent to an expression in CNF.

- Inductive step ($\vee$): prove $\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \implies P(B_1 \vee B_2)]$.

  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: $B_1 \vee B_2$ can be expressed in CNF.
  3. By the inductive hypothesis, we have

  $$\begin{aligned} B_1 \vee B_2 &\equiv CNF(B_1) \vee CNF(B_2) \\ &\equiv (C_1^1 \wedge \ldots \wedge C_1^m) \vee (C_2^1 \wedge \ldots \wedge C_2^n) \quad (C_j^i\text{s are clauses}) \\ &\equiv (C_1^1 \vee C_2^1) \wedge (C_1^1 \vee C_2^2) \wedge \ldots \wedge (C_1^m \vee C_2^{n-1}) \wedge (C_1^m \vee C_2^n) \end{aligned}$$

  4. By associativity of $\vee$, each expression of the form $(C_1^i \vee C_2^j)$ is equivalent to a single clause containing all the literals in the two clauses.
  5. Hence, $B_1 \vee B_2$ is equivalent to an expression in CNF.

Hence, any Boolean expression constructed from the operators $\wedge$, $\vee$, and $\neg$, where $\neg$ is applied only to variables, is logically equivalent to an expression in CNF. □

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 $\wedge$ and $\vee$, 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:

$$\begin{aligned}
&(A \wedge B) \vee (C \wedge D) \\
&\quad \equiv \quad (A \vee (C \wedge D)) \wedge (B \vee (C \wedge D)) \\
&\quad \equiv \quad (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)
\end{aligned}$$

Now let's add one further term:

$$\begin{aligned}
&(A \wedge B) \vee (C \wedge D) \vee (E \wedge F) \\
&\quad \equiv \quad [(A \vee C) \wedge (A \vee D)) \wedge (B \vee C) \wedge (B \vee D)] \vee (E \wedge F) \\
&\quad \equiv \quad \{[(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)] \vee E\} \\
&\quad \wedge \quad \{[(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)] \vee F\} \\
&\quad \equiv \quad (A \vee C \vee E) \wedge (A \vee D \vee E) \wedge (B \vee C \vee E) \wedge (B \vee D \vee E) \\
&\quad \quad \quad \wedge (A \vee C \vee F) \wedge (A \vee D \vee F) \wedge (B \vee C \vee F) \wedge (B \vee D \vee F)
\end{aligned}$$

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.