CS 70 Spring 2005 — Discrete Mathematics for CS — Clancy/Wagner — Notes 8

Why CNF?

Lecture Notes 6 introduced DNF and CNF, and stated that CNF is more natural for applications involving reasoning. Why is this? There are two obvious reasons. First, reasoning uses collections of sentences (often called **databases** or **knowledge bases**) that are naturally expressed as a conjunction of the sentences—the knowledge base asserts that *all* the sentences it contains are true. Translating this conjunction into DNF might involve unnecessary work and expansion of the size of the representation.

DATABASES
KNOWLEDGE BASES

Second, many sentences used in reasoning are implications with several antecedents and a single conclusion. These have the form $P_1 \wedge \ldots \wedge P_k \implies Q$. (Logic programs consist entirely of such sentences.) We have

$$\begin{aligned}(P_1 \wedge \ldots \wedge P_k \implies Q) &\equiv (\neg(P_1 \wedge \ldots \wedge P_k) \vee Q)\\ &\equiv ((\neg P_1 \vee \ldots \vee \neg P_k) \vee Q) \text{ (de Morgan's)}\\ &\equiv (\neg P_1 \vee \ldots \vee \neg P_k \vee Q) \text{ (associativity)}\end{aligned}$$

Hence, any implication sentence converts easily into a clause with the same number of literals.

Many problems of interest in CS can be converted into CNF representations; solved using theorem-proving algorithms for CNF; and then the solution is translated back into the original language of the problem. Why would we do this?

- Because we can work on finding efficient algorithms for CNF instead of finding efficient algorithms for hundreds of different problems.
- Because we can take advantage of all the work other people have done in finding efficient algorithms for CNF.
- Because often we find, once we reach CNF, that we have one or other *special case* of CNF for which very efficient (e.g., linear-time) algorithms are known.

There are other "canonical problem" targets besides CNF, including matrix inversion and determinants, **linear programming**, and finding roots of polynomials. As one becomes a good computer scientists, one develops a mental "web" of interrelated standard computational problems and learns to map any new problem onto this web. Minesweeper is a good example.

LINEAR
PROGRAMMING

Minesweeper

The rules of Minesweeper are as follows:

MINES LEFT: 6

4				
3				
2				
1				
	1	2	3	4

(a)

MINES LEFT: 0

4	m	2	2	m
3	3	4	m	2
2	m	m	2	1
1	m	3	1	0
	1	2	3	4

(b)

MINES LEFT: 1

2			
1	1	1	1
	1	2	3

(c)

MINES LEFT: 2

2					
1	1	1	1	1	1
	1	2	3	4	5

(d)

Figure 1: Minesweeper examples. (a) Initial display for a 4×4 game. (b) Final display after successful discovery of all mines. (c) Simple case: only one solution. (d) Two possible solutions, but both have (3,1) blank.

- The game is played by a single player on an $X \times Y$ board. (We will use Cartesian coordinates, so that (1,1) is at bottom left and (X,1) is at bottom right.) The display is initially empty. The player is told the total number of mines remaining undiscovered; these are distributed uniformly at random on the board. (See Figure 1(a).)
- At each turn the player has three options:
  1. *Mark* a square as a mine; the display is updated and the total mine count is decremented by 1 (regardless of whether the mine actually exists).
  2. *Unmark* a square; the mine mark is removed from a square, returning it to blank.
  3. *Probe* a square; if the square contains a mine, the player loses. Otherwise, the display is updated to indicate the *number* of mines in adjacent squares (adjacent horizontally, vertically, or diagonally). If this number is 0, the adjacent squares are probed automatically, recursing until non-zero counts are reached.
- The game is won when all mines have been correctly discovered and all non-mine squares have been probed. (See Figure 1(b).)

SAFE

Let us define a **safe** square as one that, given the available information, cannot contain a mine. Obviously, one would like to probe only safe squares, and to mark as mines only those squares that are certain to be there. Hence, the notion of logical proof is central to Minesweeper.

Many steps in Minesweeper simply involve "completing" around a square—the square is known to have k mines around it, and those k are already discovered, so all remaining adjacent squares are safe. (Some implementations offer to do this with a single click.) The dual case is where a square is known to have k adjacent mines and has k blank adjacent squares, so they must all be mines. The vast majority of turns involve one of these two kinds of steps.

Some simple examples of nontrivial reasoning in Minesweeper: First, consider Figure 1(c). Starting with the 1 in (1,1); this implies there's a mine in (1,2) or (2,2). This mine "satisfies" the 1 in (2,1); hence (3,2) is safe (has no mine). Similarly, starting with the 1 in (3,1), we can show that (1,2) is safe. Hence, (2,2) has the mine.

In Figure 1(d), we can repeat the reasoning above from either end to establish that (3,2) is safe. But there are two possible worlds consistent with all the information (i.e., the knowledge base has *two models*): mines in (1,2) and (4,2), or mines in (2,2) and (5,2). We cannot tell without more information; probing (3,2) will not help us.

Playing Minesweeper as a human, one gradually learns to recognize a set of patterns with associated logical proofs of varying difficulty. Each one seems rather *ad hoc*, and they're certainly not systematic or complete, in the sense that we certainly miss some instances where a logical move can be made.

Minesweeper in CNF

Now we're ready to start formulating Minesweeper as a logical reasoning problem. In this lecture we'll just do a simple example and concentrate on the logical reasoning processes. In the next lecture we'll do the full formulation.

Let's start with the example in Figure 1(c). First, we decide on the variables. We'll let $X_{x,y}$ be true iff (x,y) contains a mine. For example, $X_{1,2}$ is true if $(1,2)$ (top left) contains a mine. For this problem instance, the variables of interest are the three unknown squares $X_{1,2}$, $X_{2,2}$, and $X_{3,2}$. We have the following known facts:

- (1,1) has one adjacent mine, so exactly one of $X_{1,2}$ and $X_{2,2}$ is true. Let's call this proposition $N_{1,1}$; it is equivalent to two disjunctions. The first says that at least one is true, the second says that at least one is false.

  $$(X_{1,2} \vee X_{2,2}) \wedge (\neg X_{1,2} \vee \neg X_{2,2})$$

- (2,1) has one adjacent mine, so exactly one of $X_{1,2}$, $X_{2,2}$ is true. This is the proposition $N_{2,1}$:

  $$(X_{1,2} \vee X_{2,2} \vee X_{3,2}) \wedge (\neg X_{1,2} \vee \neg X_{2,2}) \wedge (\neg X_{2,2} \vee \neg X_{3,2}) \wedge (\neg X_{3,2} \vee \neg X_{1,2})$$

- (3,1) has one adjacent mine, so exactly one of $X_{2,2}$ and $X_{3,2}$ is true. This is the proposition $N_{3,1}$:

  $$(X_{2,2} \vee X_{3,2}) \wedge (\neg X_{2,2} \vee \neg X_{3,2})$$

- There is exactly one mine left. This is the "global constraint" G:

  $$(X_{1,2} \vee X_{2,2} \vee X_{3,2}) \wedge (\neg X_{1,2} \vee \neg X_{2,2}) \wedge (\neg X_{2,2} \vee \neg X_{3,2}) \wedge (\neg X_{3,2} \vee \neg X_{1,2})$$

  Notice that in this case G is exactly the same as $N_{2,1}$.

Tthe conjunction of propositions $N_{1,1} \wedge N_{2,1} \wedge N_{3,1} \wedge G$ is a CNF representation of everything we know given the displayed board. Let d be the display, and $CNF(d)$ be the CNF representation of it. Then we are interested in deciding which squares are safe and which are mines. For example, the question of whether (1,2) is safe corresponds to deciding whether

$$CNF(d) \models \neg X_{1,2}$$

A proof that $CNF(d)$ entails $\neg X_{1,2}$ offers a complete guarantee that (1,2) is safe, because it means that there is no mine in (1,2) in any possible world (configuration of mines) consistent with what the display tells us.

Entailment and proof

This section offers a simple, complete proof method, which comes directly from the definition of entailment. We offered a definition in Lecture Notes 1 in terms of "possible worlds." We can be slightly more concise here. We say that a complete assignment M **is a model of** a proposition P if P is true in M. Then we have the following definition:

Definition 8.1 (Entailment): $P \models Q$ iff Q is true in every model of P.

Let us illustrate this idea for Minesweeper. P is the proposition corresponding to all the known information—$CNF(d)$. Q is the proposition that (1,2) is safe, i.e., $\neg X_{1,2}$. The variables are $X_{1,2}$, $X_{2,2}$, and $X_{3,2}$, so there are 8 models. We can check each one (i.e., each configuration of mines), see if it is a model of $CNF(d)$ (i.e., consistent with the display), and, if so, check that it is also a model of $\neg X_{1,2}$ (i.e., has no mine in (1,2)).

Similarly, if (2,2) contains a mine in every model of $CNF(d)$, then we have proved that (2,2) contains a mine. In Figure 1(d), some models of $CNF(d)$ have a mine in (2,3), some do not, hence we cannot prove anything.

For propositional logic in general, finite expressions can contain only a finite number of variables, so the number of possible models is finite. Therefore, we can always use this proof-by-truth-table, which is also
MODEL-CHECKING called **model-checking**:

To determine whether $P \models Q$, where P, Q are Boolean expressions on $X_1, \ldots, X_n$:
  For each possible model $M = \{X_1 = t_1, \ldots, X_n = t_n\}$
    If P is true in M
      then if Q is false in M return "no"
  Return "yes"

We will make this algorithm more concrete in the next section. For now, notice that its worst-case runtime is $O(2^n)$, because there are 2^n models to check. Notice also that we represent a model as a *set* of individual variable assignments.

Validity and satisfiability

VALID **Definition 8.2 (Validity)**: A **valid** proposition (also known as a **tautology**) is a proposition that is true in *every* possible model.

Since T is true in every possible model, a valid sentence is logically equivalent to T. What good are valid sentences? From our definition of entailment, we can derive the following fact:

Theorem 8.1: $P \models Q$ *if and only if the proposition* $(P \implies Q)$ *is valid.*

This is often called the *deduction theorem*. The proof follows directly from the definition of implication. We can think of a model-checking proof as a test of validity, requiring a check over all models.

For some problems, we are happy to find *any* model where the known information is true. For example, suppose we are given a Minesweeper board with some mines marked and some known and unknown squares and asked to determine if the information given is consistent with some possible configuration (if not, then someone has made a mistake!). Then we are asking if the proposition describing the given board is satisfiable:

SATISFIABLE **Definition 8.3 (Satisfiability)**: A **satisfiable** proposition is a proposition that is true in *some* model.

SATISFIES

We say that if proposition P is true in model M, then M **satisfies** P. Satisfiability can be checked by the obvious variant of the above algorithm for proof:

> To determine whether P is satisfiable, where P is a Boolean expression on $X_1, \ldots, X_n$:
> For each possible model $M = \{X_1 = t_1, \ldots, X_n = t_n\}$
> If P is true in M then return "yes"
> Return "no"

CONSTRAINT SATISFACTION

Many problems in computer science are really satisfiability problems. As one example, we might give a timetabling problem in one of your homeworks, and this is an instance of a huge class called **constraint satisfaction** problems, wherein one must find a set of values for some variables such that a collection of constraints are all satisfied.

Validity and satisfiability are of course connected: P is valid iff $\neg P$ is unsatisfiable; contrapositively, P is satisfiable iff $\neg P$ is not valid. We also have the following useful result:

Theorem 8.2: *$P \models Q$ if and only if the proposition $(P \wedge \neg Q)$ is unsatisfiable.*

As an exercise, try to prove this. It corresponds exactly to a proof by *reductio ad absurdum*—unsatisfiability means a contradiction must be contained in $(P \wedge \neg Q)$. What it means in practice is that we can test entailment as well as satisfiability using a satisfiability algorithm. So, from now on, we'll talk about how to implement satisfiability-testing rather than entailment.

Recursive satisfiability testing

Lecture Notes 7 gave a simple definition (*eval*) for evaluating a Boolean expression in a model. If all our expressions are in CNF, we can use an even simpler method: a CNF expression is true iff every clause is true; a clause is true iff some literal is true.

Now, how do we generate the models? We prefer not to build the entire truth table first and then run through it! (This would require exponential space as well as exponential time; and space is more expensive than time.) Instead, we can enumerate the models recursively as follows. Let $M_{1\ldots i}$ be a partial model specifying values for variables $X_1, \ldots, X_i$, and let a *completion* of $M_{1\ldots i}$ be any model for $X_1, \ldots, X_n$ agreeing with $M_{1\ldots i}$ on $X_1, \ldots, X_i$. Now define $satisfies(P, M_{1\ldots i})$ to be true iff P is true in some model that is a completion of $M_{1\ldots i}$. Obviously, P is satisfiable iff $satisfies(P, \{\})$ is true.

> $satisfies(P, M_{1\ldots n})$ is true iff P is true in $M_{1\ldots n}$.
> If $i < n$, $satisfies(P, M_{1\ldots i})$ is true iff
> $satisfies(P, M_{1\ldots i} \cup \{X_{i+1} = T\})$ is true
> or
> $satisfies(P, M_{1\ldots i} \cup \{X_{i+1} = F\})$ is true

Whereas the truth table takes exponential space, this algorithm takes only linear space—the depth of the recursion is at most n. The runtime of the algorithm is still $O(2^n)$ in the worst case, which is when P is unsatisfiable. If P is easily satisfiable, the runtime may be much less.

For the purposes of minesweeper, one expects that most of the squares are neither guaranteed mines nor guaranteed safe (especially when the "obvious" cases have already been taken care of), so most proof attempts will terminate without necessarily enumerating all the models.