CompSci 102 Discrete Mathematics for CS
Spring 2005 Forbes Minesweeper

From Umesh Varirani's notes!

CNF

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 25.1: *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 $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

2. Eliminate $\Rightarrow$, replacing it $A \Rightarrow B$ 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:

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

   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 25.2: *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 $\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \Rightarrow P(B_1 \wedge B_2)]$.
  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 $\forall B_1, B_2 \in \mathbf{B}\ [P(B_1) \wedge P(B_2) \Rightarrow P(B_1 \vee B_2)]$.
  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. $\square$

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.

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?

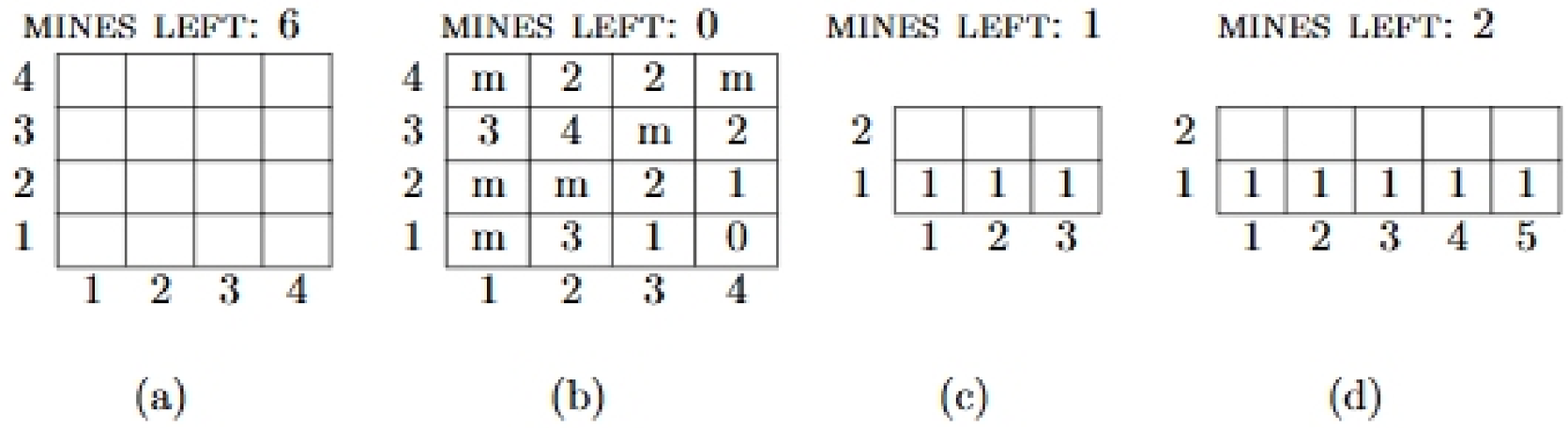

Figure 1: Minesweeper examples. (a) Initial display for a 4×4 game. (b) Final display after successful discovery of all mines. (c) Simple case: only one solution. (d) Two possible solutions, but both have (3,1) blank.

- 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.

LINEAR PROGRAMMING

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.

Minesweeper

The rules of Minesweeper are as follows:

- 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 probed. (See Figure 1(b).)