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:

   $\neg(\neg A) \equiv A$
   $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ (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 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. □

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?

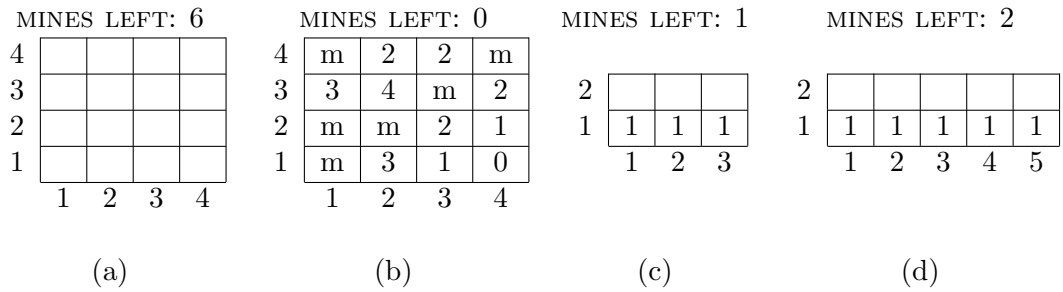


Figure 1: Minesweeper examples. (a) Initial display for a 4×4 game. (b) Final display after successful discovery of all mines. (c) Simple case: only one solution. (d) Two possible solutions, but both have (3,1) blank.

- 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).)

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 formulate Minesweeper in CNF. We would like to be able to make logical moves given the information available. We formulate the problem as follows:

- The **knowledge base** contains the known facts: the facts revealed by probes, the total number of mines left, and the locations of marked mines. (We will assume that only guaranteed mines are marked.)

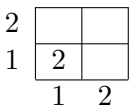
QUERY

- The **query** to the knowledge base asks, for each unknown square, whether it is safe and whether it is a mine.
- The answers for each question can be "yes" or "I don't know". Asking both questions therefore gives us all available information.

This section deals with formulating the knowledge base.

First, we decide on the variables. We'll let $X_{x,y}$ be true iff (x, y) contains a mine. For example, $X_{1,1}$ is true if $(1, 1)$ contains a mine.

Now consider a corner square (1,1) that has been probed and has 2 adjacent mines. Assume also that the contents of the adjacent squares $(1, 2)$, $(2, 1)$, and $(2, 2)$ are unknown. That is,



We know that

> Exactly two of the squares $(1,2)$, $(2,1)$, and $(2,2)$ contain a mine.

This is logically equivalent to the conjunction of two sentences:

> L: At least two of the squares $(1,2)$, $(2,1)$, and $(2,2)$ contain a mine.
> U: At most two of the squares $(1,2)$, $(2,1)$, and $(2,2)$ contain a mine.

Notice that L is equivalent to a sentence L' with the same form as M:

> L': At most one of the squares $(1,2)$, $(2,1)$, and $(2,2)$ doesn't contain a mine.

Let's translate U into CNF first. CNF uses disjunctions of literals, and a disjunction means "at least one..." We can get an expression of this form as follows:

> U': For any three of the squares $(1,2)$, $(2,1)$, and $(2,2)$, at least one does not contain a mine.

Thus, we construct all clauses using triples of negated variables. There is just one in this case:

$$U \equiv U' \equiv (\neg X_{1,2} \vee \neg X_{2,1} \vee \neg X_{2,2})$$

By a similar argument, L' becomes L'':

> L'': For any two of the squares $(1,2)$, $(2,1)$, and $(2,2)$, at least one contains a mine.

This gives us three clauses:

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

Thus, our information from the square $(1,1)$ gives the conjunction of four clauses altogether.

Let $KN(k,n)$ be the proposition that k of n variables X_1 to X_n are true. Then, by the above reasoning, we obtain

$$KN(k,n) \equiv (L(k,n) \wedge U(k,n))$$

where $L(k,n)$ is the set of clauses consisting of all possible disjunctions of $n-k+1$ literals from $\{X_1, \ldots, X_n\}$. and $U(k,n)$ is the set of clauses consisting of all possible disjunctions of $k+1$ literals from $\{\neg X_1, \ldots, \neg X_n\}$. (Note that these expressions are valid when $k > 0$ and $k+1 \leq n$. The case $k = 0$ simply means that $KN(0,n)$ is the conjunction of the clauses $\neg X_i$ for all i. The case $k+1 > n$ can only arise if $k = n$, i.e., all n variables are mines; then we simply have the clauses X_i for all i.

We can also generate a CNF expression recursively as follows:

$$KN(k,n) \equiv ((X_n \;\Rightarrow\; KN(k-1,n-1)) \wedge (\neg X_n \;\Rightarrow\; KN(k,n-1)))$$

with base cases at $k = n$ and $k = 0$, as before. Assuming $KN(k-1, n-1)$ and $KN(k, n-1)$ can be expressed in CNF, it is a simple distributivity step to express $KN(k, n)$ in CNF. The expressions resulting from this recursion look slightly different from those obtained above, but are logically equivalent.

In addition to the "local" constraints arising from squares, we also have the global constraint from the total number of remaining mines, M:

> G: Exactly M of the unknown squares on the board contain mines.

If there are N remaining unknown squares, this is a set of clauses of the form $KN(M, N)$.

Let $|KN(n, k)|$ be the number of clauses in $KN(n, k)$, using our first construction. How many is clauses are there?

Probabilistic Minesweeper

Our final application of probability is to Minesweeper. The first step is to identify the set of random variables we need:

- As in the propositional logic case, we want one Boolean variable X_{ij} which is true iff square (i, j) actually contains a mine. Using N to refer to the total number of squares, we can also label these variables as $X_1, \ldots, X_N$, which will come in handy.
- We'll also have variables D_{ij} corresponding to the display contents *for those k squares that have been probed or marked as mines.* We can also label these variables $D_1, \ldots, D_k$ for simplicity. The domain for each of these variables is $\langle m, 1, 2, \ldots, 8\rangle$. We'll call these variables the $Numbers$, and the corresponding X_{ij} variables will be called $Known$. (Note that $Numbers$ includes marked mines; we assume that only logically guaranteed mines are marked.)

Finally, we'll use and M to refer to the total number of mines. (Both N and M are constants.)

The probability space

Now we must write down our probability space, i.e., the joint distribution over all the variables. It turns out to be easiest to do this in two parts, using the chain rule:

$$P(X_1, \ldots, X_N, D_1, \ldots, D_k) = P(D_1, \ldots, D_k | X_1, \ldots, X_N) P(X_1, \ldots, X_N)$$

Why this way? Because (1) the prior distribution of mines, $P(X_1, \ldots, X_N)$, is easy to compute because mines are scattered uniformly at random, and (2) the display variables are determined by the underlying mines, so the conditional distribution $P(D_1, \ldots, D_k | X_1, \ldots, X_N)$ is also relatively easy to describe.

First, the prior $P(X_1, \ldots, X_N)$. One is tempted to write

$$P(X_1, \ldots, X_N) = P(X_1) P(X_2) \ldots P(X_n) \quad \text{WRONG}$$

because the mines are scattered "uniformly at random," which suggests independence. Furthermore, the probabilities $P(X_i)$ are simply $\langle (M/N), (1 - M/N)\rangle$. But independence does not hold,

because the total number of mines is fixed! For example, if the first M squares all get mines, then the next square has probability 0 of getting a mine.

Instead, let's begin with the obvious fact that any sample point containing anything other than M mines must have probability 0. Using the notation $\#(x_1, \ldots, x_N)$ to denote the number of "trues" in the sample point $x_1, \ldots, x_N$, we have, for all combinations of values $x_1, \ldots, x_N$,

$$P(x_1, \ldots, x_N) = 0 \quad \text{if} \quad \#(x_1, \ldots, x_N) \neq M$$

Now, by symmetry, all the remaining sample points—those with exactly M mines—have equal probability. There are $C(N, M)$ such sample points; their total probability must be 1; therefore,

$$P(x_1, \ldots, x_N) = \frac{1}{C(N, M)} \quad \text{if} \quad \#(x_1, \ldots, x_N) = M$$

Turning to the display variables, we know they are determined precisely according to the rules of Minesweeper, given the actual contents of all the squares. I.e., for all combinations of values $d_1, \ldots, d_k, x_1, \ldots, x_N$,

$$P(d_1, \ldots, d_k | x_1, \ldots, x_N) = \begin{cases} 1 \text{ if } d_1, \ldots, d_k \text{ correctly displays } x_1, \ldots, x_N \\ 0 \text{ otherwise} \end{cases}$$

Finding safer squares

The strategy we choose to implement (which is *not* necessarily optimal), is to make logical moves where possible, and, if not, to play in the square that has the lowest probability of being a mine. Therefore we need to compute, for each unknown square (i, j), the quantity $P(X_{ij} | known, numbers)$. We develop a simple and computable expression for this probability in a series of steps.

First, we need to massage the expression into a form containing the terms that we know about—the prior over all the X-variables, and the conditional probability of the display variables given those variables. The expression $P(X_{ij} | known, numbers)$ is missing the unknown variables other than X_{ij}; call these $Unknown$. So we introduce them by summing over them (the standard summation over constituent sample points for an event):

$$P(X_{ij} | e) = P(X_{ij} | known, numbers) = \sum_{unknown} P(X_{ij}, unknown | known, numbers)$$

We'd like to have an expression with $numbers$ conditioned on X-variables, so we apply Bayes' rule:

$$P(X_{ij} | e) = \alpha \sum_{unknown} P(numbers | known, X_{ij}, unknown) P(known, X_{ij}, unknown)$$

So far, so good; the variables $known, X_{ij}, unknown$ constitute all the X-variables, so we have terms here that we know how to compute. The only problem is that we have too many of them! The number of unknown variables could be as large as N, so the summation is over $O(2^N)$ cases.

The solution to this problem is to identify a subset of these variables that affect the display and to simplify the expressions using conditional independence so that the summation covers only this subset.

Let $Fringe$ denote those unknown variables (not including X_{ij}) that are adjacent to numbered squares. The idea is that the $Numbers$ are completely determined just by $Known$, $Fringe$, and X_{ij} (if it is adjacent to a number). Given these, the $Numbers$ are conditionally independent of the remaining unknown variables, which we call $Background$. There are two cases to deal with, depending on whether X_{ij} is adjacent to a number.

Case 1: X_{ij} is adjacent to a number.

$$\begin{aligned}
&P(X_{ij}|e) \\
&\quad = \alpha \sum_{fringe,background} P(numbers|known, fringe, background, X_{ij}) P(known, fringe, background, X_{ij}) \\
&\quad = \sum_{fringe,background} P(numbers|known, fringe, X_{ij}) P(known, fringe, background, X_{ij}) \\
&\quad = \sum_{fringe} P(numbers|known, fringe, X_{ij}) \sum_{background} P(known, fringe, background, X_{ij})
\end{aligned}$$

where the second line is derived using conditional independence. Now, in this last expression, the term $P(numbers|known, fringe, X_{ij})$ is 0 unless, to use our logical terminology, the assignment denoted by $fringe, X_{ij}$ is a *model* of the CNF expression implied by the evidence. If it *is* a model, then the probability is 1. So the summation over $fringe$ reduces to a simpler summation over the models of the evidence:

$$P(X_{ij}|e) = \alpha \sum_{\{fringe:\ \langle fringe, X_{ij}\rangle \in models(e)\}} \sum_{background} P(known, fringe, background, X_{ij})$$

The sum over the background variables, which may still have a huge number of cases, can be simplified because the prior probability term $P(known, fringe, background, X_{ij})$ is $1/C(N, M)$ or 0, depending on whether $\#(known, fringe, background, X_{ij}) = M$. Therefore, we just have to count the number of cases where the background has the right number of mines. This is given by $C(B, M - L)$, where B is the size of the background and L is $\#(known, fringe, X_{ij})$ the number of mines not in the background. Finally, we obtain

$$P(X_{ij}|e) = \alpha \sum_{\{fringe:\ \langle fringe, X_{ij}\rangle \in models(e)\}} C(B, M - L)$$

which is simple to compute provided we can enumerate the models for the fringe variables. This costs $O(2^{|Fringe|})$, roughly the same as the logical algorithm.

Case 2: X_{ij} is not adjacent to a number.
The derivation is very similar, but X_{ij} acts like a background variable rather than a fringe variable:

$$\begin{aligned}
P(X_{ij}|e) &= \alpha \sum_{fringe,background} P(numbers|known, fringe, background, X_{ij}) P(known, fringe, background, X \\
&= \sum_{fringe,background} P(numbers|known, fringe) P(known, fringe, background, X_{ij}) \\
&= \sum_{fringe} P(numbers|known, fringe) \sum_{background} P(known, fringe, background, X_{ij})
\end{aligned}$$

Now the fringe variables constitute the entire model:

$$P(X_{ij}|e) = \alpha \sum_{fringe \in models(e)} \sum_{background} P(known, fringe, background, X_{ij})$$

and the final expression is very similar.

$$P(X_{ij}|e) = \alpha \sum_{fringe \in models(e)} C(B, M - L)$$

It is clear that this expression is the same for all X_{ij} that are not adjacent to a number, so we need only do this once to get the probability of safety for background squares. Each number-adjacent square must be evaluated separately.

(Note: in the two derivations, the meaning of $Fringe$ and $Background$ is different. In the first, $Fringe$ includes all number-adjacent variables except X_{ij}, while $Background$ includes all other variables. In the second, $Fringe$ includes all number-adjacent variables, while $Background$ includes all other variables except X_{ij}. This makes for somewhat simpler mathematical expressions.)