

Logic in Computer Science

Chapter 7

Antonina Kolokolova

January 31, 2014

7.1 Normal forms of propositional formulas.

Any formula has an equivalent one in a normal form. In computer science, the two normal forms we are interested in are *conjunctive normal form (CNF)* and *disjunctive normal form (DNF)*. The first one is a \wedge of \vee s, where \vee is over variables or their negations (*literals*), the second is a \vee of \wedge s of literals. A \vee of literals is also called a *clause*, and a \wedge of literals a *term*.

Example 1. • $\neg p$, x , s are examples of literals, whereas $\neg\neg p$ or $(x \vee y)$ is not a literal

- $(x \vee \neg y \vee z)$, $(\neg p)$ are clauses,
- $(x \wedge \neg y \wedge z)$, $(\neg p)$ are terms.
- $(x \vee \neg z \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg y)$ is a CNF.
- $(x \wedge z) \vee (\neg y \wedge z \wedge x) \vee (\neg x \wedge z)$ is a DNF.
- $(x \wedge \neg(y \vee z) \vee u)$ is neither a CNF nor a DNF, but it is equivalent to the following DNF: $(x \wedge \neg y \wedge \neg z) \vee u$.

Why do we need CNFs and DNFs? Other than the fact that they are convenient normal forms, they are very useful for constricting formulas given a truth table. In particular, DNFs can be constructed from truth table by taking a disjunction (that is, \vee) of all satisfying truth assignments and CNFs by taking a conjunction (\wedge) of negations of falsifying truth assignments

Example 2. Recall the truth table for $(p \rightarrow q) \wedge q \rightarrow p$. Suppose that you are given just the first two and the last column. How do you construct a formula that would have such a truth table?

Let us start by constructing a DNF formula. It would say, basically, “either the first line satisfies me, or the second, or the last”. We write it as $(p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge \neg q)$. Each term here fully describes one satisfying assignment, and the whole formula is true iff $(p \rightarrow q) \wedge q \rightarrow p$ is. Now, to construct a formula in CNF we make it say “I am satisfied by neither of the following assignments”. One way to write it is as a negation of a DNF formula listing all falsifying assignments. In this example there is just one falsifying assignment, so the formula becomes $\neg(\neg p \wedge q)$. By DeMorgan’s law, this is equivalent to $(\neg\neg p \vee \neg q)$, and finally $(p \vee \neg q)$. If there were more than one falsifying assignment, here we would have several of clauses with \wedge between them.

This construction also shows us that for any formula there is an equivalent CNF and an equivalent DNF formula (these are the formulas constructed from the truth table of the original formula). In particular, if the formula is a tautology, its DNF and CNF consist of just $(p \vee \neg p)$ for an arbitrary p , and if it is a contradiction, of $(p \wedge \neg p)$.

Let us do a larger example of constructing a CNF and DNF from a truth table.

A crucial point is that every truth assignment can be encoded as a propositional formula which is true just on that assignment and false everywhere else. In order to make such a formula, take \wedge of all variables that are true in the truth assignment and negations of all variables that are false in that truth assignment. Since every variable is either true or false, every variable will occur exactly once in this conjunction, either positively or negatively. For example, a truth assignment $p = T, q = F, r = T$ can be represented by a formula $p \wedge \neg q \wedge r$. This formula will be true when $p = T, q = F$ and $r = T$ and false everywhere else.

This representation of truth assignments by formulas is our building block for constructing DNFs and CNFs from a truth table. A DNF says that one of the truth assignments listed is true, where the assignments listed are all the satisfying assignments of a formula. A CNF is just a negation of a DNF formula listing falsifying truth assignments, simplified using DeMorgan’s law to place negations on variables.

Let us look at a larger example for constructing CNFs and DNFs, equivalent to formulas (given by truth tables). That is, the CNF, the DNF and original formula, although they look quite different, share the same set of variables and are satisfied by exactly the same truth assignments. Their truth tables are identical.

Example 3. Last time we used a truth table of a propositional formula $(p \rightarrow q) \wedge q \rightarrow p$ for constructing DNFs and CNFs. Now let us add one more variable: consider a formula $((p \rightarrow q) \wedge q \rightarrow p) \wedge (\neg q \vee \neg r)$.

r	p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge q$	$(p \rightarrow q) \wedge q \rightarrow p$	$\neg q$	$\neg r$	$(\neg q \vee \neg r)$	$((p \rightarrow q) \wedge q \rightarrow p) \wedge (\neg q \vee \neg r)$
T	T	T	T	T	T	F	F	F	F
T	T	F	F	F	T	T	F	T	T
T	F	T	T	T	F	F	F	F	F
T	F	F	T	F	T	T	F	T	T
F	T	T	T	T	T	F	T	T	T
F	T	F	F	F	T	T	T	T	T
F	F	T	T	T	F	F	T	T	F
F	F	F	T	F	T	T	T	T	T

There are three F in the last column and five T, so our CNF will have 3 clauses (that is, it will be the \wedge of three \vee), and our DNF will have 5 terms.

So, to construct a CNF for this formula, take

$$\neg((r \wedge p \wedge q) \vee (r \wedge \neg p \wedge q) \vee (\neg r \wedge \neg p \wedge q))$$

After simplification (applying DeMorgan's law and double negation law), this becomes

$$(\neg r \vee \neg p \vee \neg q) \wedge (\neg r \vee p \vee \neg q) \wedge (r \vee p \vee \neg q)$$

You can check yourself that this formula is CNF and is indeed equivalent to $((p \rightarrow q) \wedge q \rightarrow p) \wedge (\neg q \vee \neg r)$; that is, it has the same truth table. A DNF for this formula is:

$$(r \wedge p \wedge \neg q) \vee (r \wedge \neg p \wedge \neg q) \vee (\neg r \wedge p \wedge q) \vee (\neg r \wedge p \wedge \neg q) \vee (\neg r \wedge \neg p \wedge \neg q)$$

Note that between the CNF and DNF all 8 truth assignments are mentioned: the falsifying ones in the CNF and satisfying in DNF.

How large would such CNF or DNF be? Well, potentially nearly as large as the truth table itself (say half of the entries are false, and half are true). But does it have to be so, or are there always smaller formulas? If a column is filled with T and F pretty much randomly, it is likely not to have a smaller description than the column itself, and so any formula encoding it would have to be nearly as large as a truth table itself. In a few lectures we will start talking about (classes of) Boolean functions, and describing them by sequences (one for each number of variables) of truth tables. In that case, the question would be if such a Boolean function can, as $n \rightarrow \infty$, be described by formulas asymptotically smaller than their truth table. In that case, although we know that a random such Boolean function would not have small descriptions, and we also know functions for which there are no small CNF/DNF, the general question of finding a natural example of a "hard" (that is, with no small formula) Boolean function is an open problem.

Recall that a formula is in the CNF (conjunctive normal form) if it is a \wedge of \vee s of literals (variables or their negation.)

In this lecture we will talk about proving (or, actually, finding contradictions) statements that are in this special form.

Definition 1 (Resolution rule). : Given two clauses of the form $C \vee x$ and $(D \vee \neg x)$, where C and D are (possibly empty) disjunction of variables, can derive a (possibly empty) clause $(C \vee D)$.

That is,

$$(C \vee x) \wedge (D \vee \neg x) \rightarrow (C \vee D)$$

where $C = (l_1 \vee \dots \vee l_k)$ and $D = (l'_1 \dots l'_{k'})$ for some literals. If there is a repeated literal, only write it once.

Note that you may end up deriving an “empty” clause, that is, a \vee of zero literals. But the only way this could happen is when two clauses being resolved are (x) and $(\neg x)$. But $x \wedge \neg x$ is always false, a contradiction. So deriving an empty clause proves that the original formula was a contradiction (which is what we usually want to show).

Example 4. Consider the following statements:

p : it is sunny

q : the weather is good

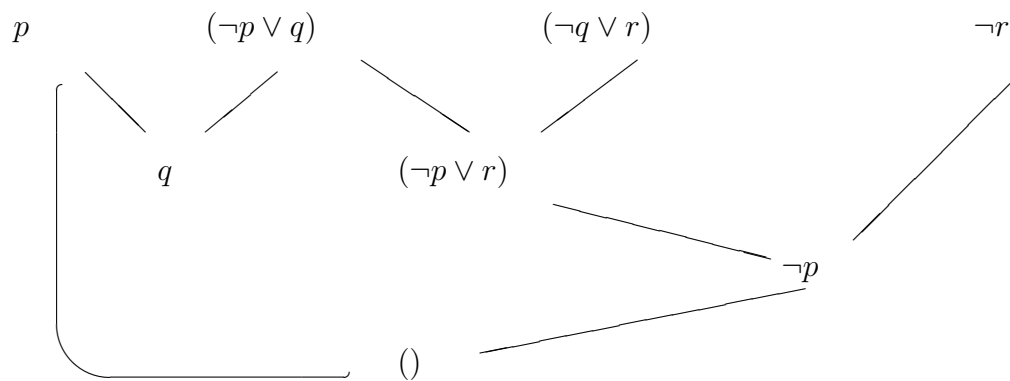
r : I spend the day outside

Now consider the following argument:

$$\begin{array}{l} p \\ p \rightarrow q \\ q \rightarrow r \\ \therefore r \end{array}$$

We want to prove that this is a valid argument, that is, p , $p \rightarrow q$ and $q \rightarrow r$ together imply r (this in general is called a *transitivity* law). Resolution proof method allows us to find contradictions: so we represent this problem as a contradiction $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \wedge \neg r$. Note that here again we are using the fact that the negation of an implication (in this case, premises imply the conclusion) is a conjunction of premises and negation of the conclusion.

Resolution works with CNF formula, so the first step is to convert the formula above into CNF. In this case, it is very easy: just apply the definition of implication to the second and third clause to obtain $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg r$.



Puzzle 6. Prove that in a big city like Toronto with more than a million people there would be at least two people with the same number of hairs on their heads. Assume that the number of hairs on somebody's head cannot be more than 300,000.

Chapter 8

8.1 Resolution.

Recall that a formula is in the CNF (conjunctive normal form) if it is a \wedge of \vee s of literals (variables or their negation.)

In this lecture we will talk about proving (or, actually, finding contradictions) statements that are in this special form.

Definition 1 (Resolution rule). : *Given two clauses of the form $(C \vee x)$ and $(D \vee \neg x)$, where C and D are (possibly empty) disjunction of variables, can derive a (possibly empty) clause $(C \vee D)$.*

That is,

$$(C \vee x) \wedge (D \vee \neg x) \rightarrow (C \vee D)$$

where $C = (l_1 \vee \dots \vee l_k)$ and $D = (l'_1 \dots l'_{k'})$ for some literals. If there is a repeated literal, only write it once.

Note that you may end up deriving an “empty” clause, that is, a \vee of zero literals. But the only way this could happen is when two clauses being resolved are (x) and $(\neg x)$. But $x \wedge \neg x$ is always false, a contradiction. So deriving an empty clause proves that the original formula was a contradiction (which is what we usually want to show).

Example 1. Consider the following statements:

p : it is sunny

q : the weather is good

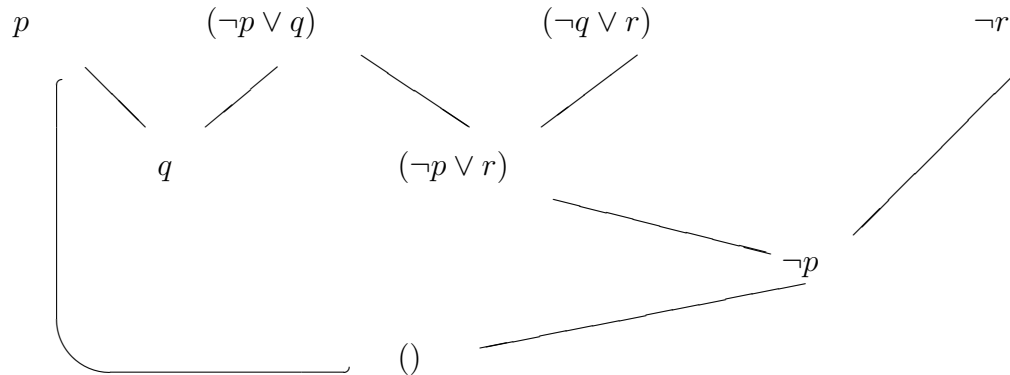
r : I spend the day outside

Now consider the following argument:

$$\begin{array}{l} p \\ p \rightarrow q \\ q \rightarrow r \\ \therefore r \end{array}$$

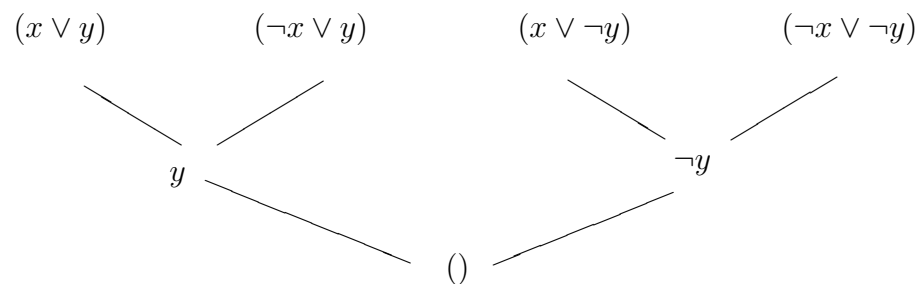
We want to prove that this is a valid argument, that is, p , $p \rightarrow q$ and $q \rightarrow r$ together imply r (this in general is called a *transitivity* law). Resolution proof method allows us to find contradictions: so we represent this problem as a contradiction $p \wedge (p \rightarrow q) \wedge (q \rightarrow r) \wedge \neg r$. Note that here again we are using the fact that the negation of an implication (in this case, premises imply the conclusion) is a conjunction of premises and negation of the conclusion.

Resolution works with CNF formula, so the first step is to convert the formula above into CNF. In this case, it is very easy: just apply the definition of implication to the second and third clause to obtain $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg r$.



For comparison, you can also prove this using rules of inference such as Modus Ponens. That allows you to derive q from the first two lines (p and $p \rightarrow q$ give q), and then derive r from q and $q \rightarrow r$.

Example 2. Let's look at another example. Now there are only two variables, and the four clauses contain all possible combinations (so the formula is a contradiction) $(x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$. You can easily see that any truth assignment to x and y falsifies one clause.



Resolution proof system is very powerful in that given any formula in CNF form it can check whether it is a contradiction. However, resolution is not very efficient. Although it is much more efficient in practice than using truth tables (for example, the $(x_1 \wedge (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_3) \wedge \dots \wedge (x_{n_1} \rightarrow x_n) \wedge \neg x_n$ can be done by a resolution proof system in n steps, whereas the truth table would have 2^n lines), there are examples of problems on which resolution proof system has to explore all possibilities, and so is similar to the truth table method in

efficiency. The most prominent such example is the PigeonHole Principle, which says that it is impossible to put $n + 1$ element (pigeon) into n places (holes) without two elements getting into the same place. You needed to use this principle, this kind of counting reasoning to solve the last lecture puzzle.

The resolution is still the most popular method of programming automated proof systems because of its clarity and simplicity. However, when working with or programming such a system keep in mind that resolution “can’t count”.

At this point a natural question is: how can we use resolution to deal with general propositional formulas, that is, ones not in CNF form. Of course, we could create a truth table and write a corresponding CNF formula, but if we have a truth table then we can just test whether a formula is a contradiction by checking if all entries in the last column are Fs. Another idea is to manipulate the formula using logic identities until it becomes a CNF. This is possible, but unfortunately can result in a very large (size comparable to the truth table) formula, especially when converting a DNF into a CNF (think of converting a formula $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \dots (x_n \wedge y_n)$ into a CNF: you will end up with 2^n clauses on n variables each, one for every combination of Xs and Ys).

A different idea is to add new variables in such a way that the new formula is a contradiction if and only if the original one was a contradiction. Here is one way of doing it.

- 1) Assign a new variable to every binary connector ($\wedge, \vee, \rightarrow, \leftrightarrow$) in the formula (starting from the outermost connector, operation to be done last). For example, if the original formula is $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$, then assign variable y_1 to outermost \wedge connector, then the variable y_2 to the \vee before x_4 and so on (think of writing a parse tree of the formula and assigning y_1 to the top, y_2 and (possibly) y_3 to its children, etc).
- 2) Next, write the conditions describing the new variables: for example, if our y_6 was assigned to \leftrightarrow in $\neg x_1 \leftrightarrow x_3$, then we need the condition $(y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$ which says that y_6 is true if and only if the expression $(\neg x_1 \leftrightarrow x_3)$ is true. The continue writing similar expressions for other y_i s, using other y_i variables to denote subformulas (for example, the top \wedge denoted by y_1 gets $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$). Finally, add clause (y_1) to the formula which will mean that the whole original formula is true.
- 3) We are almost done: our formula is a conjunction of small parts; most importantly, each small part $(y_i \leftrightarrow (\dots))$ only involves three variables (for example, y_1, y_2 and x_2 in definition of y_1 , or $y_2 \leftrightarrow (y_3 \vee \neg y_4)$ for y_3). Three variables give us a small enough truth table (8 lines), that we can write a CNF for this small table with just 7 or less clauses, each with at most 3 variables.. So the size of the new formula becomes approximately the number of logical connectives (times a constant).
- 4) The resulting formula is a CNF, and although it has many more variables, it is a contradiction if and only if the original formula was.

8.2 Complete set of connectives

From what we have done it is easy to see that any possible truth table (we will start saying “boolean function” soon) can be represented by a formula written with just \wedge, \vee and \neg . Such set of connectives is called *complete*.

A stronger result tells us that even \vee is not necessary in this set (homework exercise). But \neg is necessary, because every formula without \neg is true when all of its variables are set to true. So contradiction cannot be described by such a formula, and any truth table with a 0 in the last column when all of its inputs are true cannot be represented by a formula using only \vee and \wedge .

We have seen that there are two connectives that allow us to construct formulas representing all possible truth tables. Is there a single connective that can be used to construct formulas with all possible truth tables? The answer is yes, and such connective is called NAND (stands for “not-and”) in digital circuit design, or Sheffer’s stroke (written as $|$). It is defined so $p|q$ is false when both p and q are true, and true otherwise.

p	q	$p q$
1	1	0
1	0	1
0	1	1
0	0	1

You can see here that it has values exactly opposite of the values of $p \wedge q$, hence the name NAND. To show that it is complete, let us look at how to simulate \neg and \wedge with it. Once we have both \neg and \wedge , we can rewrite any formula to an equivalent one which only uses $|$ as a connective using ideas from the problem in your homework assignment.

To show how to represent $\neg p$, note that the first row of the truth table has 0 for the value of the Sheffer’s stroke, and the last row has a 1. So $\neg p \iff p|p$. Similarly, $p \wedge q \iff \neg(p|q) \iff (p|q)|(p|q)$, since Sheffer’s stroke is the negation of \wedge and we just showed how to do \neg .