CSCI3390-Lecture 15: Boolean satisfiability; polynomial-time reductions

1 Propositional Logic

(I've posted the first two chapters of the text from the Logic and Computation course on our course website. This contains detailed explanations of the syntax and semantics of propositional logic, disjunctive and conjunctive normal form, and also the reduction of Sudoku to boolean satisfiability. The treatment of these matters in the present notes, in sections 1 and 3, is just an executive summary.)

Propositional formulas are built from *variables,* usually denoted p, q, r, p_1, p_2, *etc.* using binary connectives $\wedge$ (for AND), $\vee$ (for OR) and a unary operation $\neg$ (for NOT). We also sometimes write a horizontal line over a formula in place of this negation operator. Parentheses are used as required. For example

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

which is equivalent to

$$\overline{(p \vee (\overline{(q \wedge \bar{r})} \wedge (p \vee q)}$$

is an instance of such a formula. You are no doubt familiar with the fact that when we substitute truth values **T** and **F** for the variables in the formula, we obtain a truth value for the formula. You are also probably familiar with the method of truth tables, in which we tabulate the value of a formula for all possible assignments of truth values to the variables.

We sometimes use other connectives than the ones given above—for example, implication and exclusive-or. But these can all be defined in terms of the original ones. So we consider $p \rightarrow q$ as an abbreviation for $\neg p \vee q$, and $p \oplus q$ as an abbreviation for $(p \vee q) \wedge \neg(p \wedge q$.

The *boolean satisfiability problem* is:

Input: A propositional formula ϕ. **Output:** Yes if and only there is some assignment of truth values to the variables of ϕ that gives the value True–*i.e.,* if ϕ is *satisfiable.* No otherwise.

2 Boolean satisfiability is in NP

Computing one row of the truth table for a formula can be carried out quickly, and if the row happens to correspond to a satisfying assignment, this will verify that the formula is satisfiable. However, if the formula has n variables, there are 2^n rows to compute in order to be sure whether or not the formula is satisfiable. This is typical of what we find with problems in **NP**. Let's flesh this out a bit.

A witness for satisfiability of ϕ is the assignment of truth values to the variables—that is, a sequence of n bits, where n is the number of variables in ϕ. This is less than the size of ϕ, and thus bounded by a polynomial in the size of ϕ.

Verifying whether an assignment is satisfying consists of evaluating the formula on the assignment. This is done by standard parsing algorithms whose running time is linear in the size of the formula. The algorithm may need to consult the assignment to find the right value to use while parsing, but even using crude search methods, each such consultation requires time linear in the number of variables. So at worst, the running time for verification is $O(N^2)$ where N is the size of the formula. So we have the two properties required for a problem to be in **NP**: polynomial-size witnesses and polynomial-time verification of witnesses.

So the **big question** is whether boolean satisfiability is in **P**. As we noted above, the naïve algorithm for boolean satisfiability, which consists of computing the entire truth table, runs in time exponential in the number of variables in the formula.

3 Finding a solution versus determining whether a solution exists

While we present most of these computational problems as decision problems with yes-no answers, when you have a propositional formula, you'd really like to know not just whether it is satisfiable but also find a satisfying assignment. Here we prove: If boolean satisfiability is in **P**, then there is a polynomial-time algorithm for finding a satisfying assignment for a given formula, when one exists.

Here is the algorithm: Let $p_1, \ldots, p_n$ be the variables in the formula ϕ. Use the supposed poly-time algorithm for satisfiability to determine if ϕ is satisfiable. If it is, substitute **T** for p_1 in ϕ, obtaining a new formula ϕ_1, which only has $n-1$ variables. Use the algorithm again to determine if ϕ_1 is satisfiable. If it is not, then go back and substitute **F** for p_1 in ϕ. The new formula, which we also call ϕ_1, must be satisfiable, and we have determined a value for p_1 that leads to a satisfying assignment.

We now repeat the procedure with ϕ_1 using the next variable p_2, and continue like this with each variable in turn. The result is that we determine a satisfying assignment for ϕ, provided one exists.

This algorithm makes $n+1$ calls to the algorithm for determining satisfiability, and each of the formulas it checks has size no bigger than the size of ϕ. Substitution in each of the formulas also runs in time polynomial in the size of the formula, and we do $2n$ substitutions altogether. Thus if our algorithm for determining satisfiability runs in polynomial time, this algorithm for finding a satisfying assignment does as well.

I hasten to add that the widely-held view is that the premise of this problem is *false*—that there is no polynomial-time algorithm for determining satisfiability. However, this argument can still be used to find poly-time algorithms for finding satisfying assignments for special classes of formulas for which we know how to determine satisfiability (like the 2-CNF formulas) discussed below.

4 CNF and SAT

If you have a propositional formula, then you can use DeMorgan's Laws:

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$
$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

to move all negations inside parentheses. Doing this repeatedly results in an equivalent formula in which all the negations are at the level of the variables. Thus the resulting formula is built from a base of *literals* $p, \bar{p}, q, \bar{q}$, *etc.* by repeated applications of $\wedge$ and $\vee$. Furthermore, we can repeatedly apply the pair of distributive laws:

$$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$$
$$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

to obtain an equivalent formula that is an OR of ANDs of literals (*disjunctive normal form*) and, alternatively, to obtain an equivalent formula that is an AND of ORs of literals (*conjunctive normal form*).

For example, the following, which is true if and only if exactly two of the three variables are true, is in disjunctive normal form.

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

If we apply the second distributive law repeatedly, we get a CNF formula that is the AND of $3^3 = 27$ terms, each of which is the OR of 3 literals. Since there are only 8 different ways to make an OR of 3 literals, there is obviously a lot of repetition in this formula. Alternatively, we can take the negation of the above formula in DNF, which is the OR of 5 ANDs, and apply De Morgan's Law to get the CNF formula

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

The five conjuncts in this formula are called the *clauses* of the CNF formula.

The computational problem SAT is the special case of the boolean satisfiability problem in which the input is a formula in CNF.

Computational problems seem to arise naturally in this form, as a conjunction of a large number of relatively simple constraints. We'll see as we go along that there is a precise sense in which SAT is just as hard as the general boolean satisfiability problem.

5 2-SAT is in P

Let $k > 0$. k-SAT is the special case of SAT in which the input is a CNF formula in which every clause has no more than k literals.

For example, an instance of 2-SAT is the formula

$$(p \vee \bar{q}) \wedge (p \vee q) \wedge (q \vee r) \wedge (\neg p \vee \neg r).$$

We will prove here that 2-SAT is in P. That is, there is a polynomial-time algorithm for determining whether a given 2-SAT formula is satisfiable.

The proof is based on a graphical representation of the 2-SAT formula. Given such a formula, each clause has the form $\ell_1 \vee \ell_2$, where ℓ_1 and ℓ_2 are literals. We construct a directed graph whose vertices are all possible literals—that is, for each

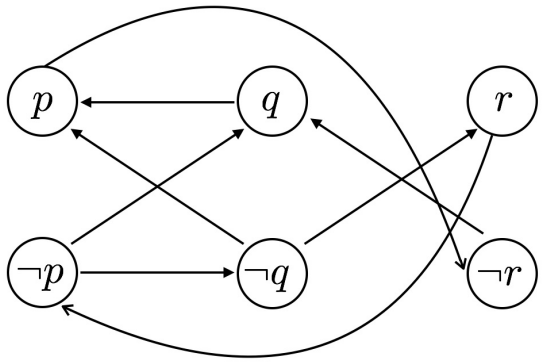

Figure 1: Digraph associated with the 2-CNF formula in the text

variable x in the formula, both x and $\bar{x}$ are vertices of the graph. For each clause $\ell_1 \vee \ell_2$, both $\bar{\ell}_1 \to \ell_2$, and $\bar{\ell}_2 \to \ell_1$ are directed edges.

For example, for the formula given above, the digraph is shown in Figure 1.

Let us write $\ell \Rightarrow \ell'$ if there is a directed *path* in the graph from ℓ to ℓ'. We claim that *the formula is satisfiable if and only if the graph we constructed has no circuit of the form*

$$\ell \Rightarrow \ell' \Rightarrow \ell.$$

To show this, let's suppose that our graph has this property—that it contains no such 'forbidden' circuits. We pick some literal ℓ arbitrarily and assign it the value **T**. We then find all the vertices of the graph that can be reached by a directed path beginning at ℓ. If ℓ is assigned the value **T** in a satisfying assignment, then all of these literals labeling these vertices must also be assigned the value **T**. If it happens that there is a literal ℓ_1 and paths

$$\ell \Rightarrow \ell_1, p \Rightarrow \bar{\ell}_1,$$

then we would get a contradiction, so we cannot have a satisfying assignment in which ℓ is assigned **T**. So then $\bar{\ell}$ must be assigned **T**. We repeat the procedure above, finding all vertices that can be reached from $\bar{\ell}$. Now suppose that this, too, leads to a contradiction. Then we have a literal ℓ_2 and paths

$$\bar{\ell} \Rightarrow \ell_2, \bar{\ell} \Rightarrow \bar{\ell}_2$$

Because of the symmetry in the construction of the graph, a path $\bar{\ell} \Rightarrow \bar{\ell}_2$ implies the existence of a path $\ell_2 \Rightarrow \ell$, and a path $\ell \Rightarrow \bar{\ell}_1$ implies the existence of a path

$\ell_1 \Rightarrow \bar{\ell}$. The result is that we get a path

$$\ell \Rightarrow \ell_1 \Rightarrow \bar{\ell} \Rightarrow \ell_2 \Rightarrow \ell,$$

contradicting the assumption that no such forbidden circuits exist. Thus one of these two choices must lead to an assignment free of contradictions. It may be that in this process some variables were not assigned a value. If that is the case, we pick one of these variables, say q, and provisionally set it to be **T**. Note that there cannot be a path from q to a literal ℓ assigned **F** in the first phase, because otherwise we would have a path $\bar{\ell} \Rightarrow \bar{q}$, so $\bar{q}$ would have already been assigned the value **T**. Thus the only way that this assignment can lead to a contradiction is through a forbidden circuit involving variables that were not previously assigned. As before, if we obtain a contradiction, we switch and set $\bar{q}$ to **T**. By continuing in this manner, we find an assignment to every variable that does not lead to a contradiction, so the formula is satisfiable. The converse direction of the claim is trivial: If the graph contains such a forbidden cycle, then there can be no satisfying assignment, since whether we set ℓ or $\bar{\ell}$ true, we would have to make the other true as well.

Let's see how this plays out for the graph in Figure 1. If we begin by assigning $\bar{p}$ the value **T**, we get a path $\neg p \rightarrow q \rightarrow p$, a contradiction. So we assign p the value **T** instead. From p we have paths to $\bar{r}$ and q and back to p again, but can reach no other vertices. This gives the satisfying assignment p **T**, q **T** and r **F**. You can verify that this is a satisfying assignment (and in fact is the only one). Suppose we added a new clause $(\neg q \vee r)$ to the formula. We would then get the graph in Figure 2.

We now have a forbidden circuit

$$p \rightarrow \bar{r} \rightarrow \bar{p} \rightarrow q \rightarrow p,$$

so the formula has no satisfying assignment.

Why does this property tell us that 2-SAT is in **P**? First, because we can construct the graph from the formula in time polynomial in the size of the formula, simply scanning the formula left to right and adding two edges for each clause. The size of the graph (number of vertices and edges) is bounded above by a polynomial in the size of the formula. Second, because we can verify the required property of the graph in time polynomial in the size of the graph: Whenever we have two vertices i, j, we can determine, using breadth-first search, whether there is a path from i to j in time polynomial in the size of the graph. We apply this algorithm $2n$ times, where n is the number of vertices in the graph, to all the pairs

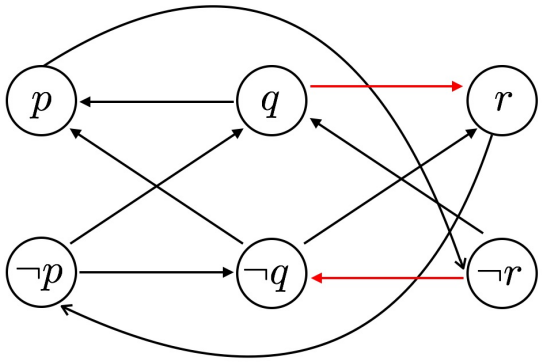

Figure 2: Digraph associated with the 2-CNF formula with the new clause $(\neg q \vee r)$ appended.

$(\ell, \bar{\ell})$, where ℓ is a literal, to tell us if there are any forbidden cycles. Multiplying by $2n$ keeps the running time polynomial in the size of the graph, and thus in the size of the formula. So we can decide satisfiability in **P**.

6 3-SAT is as hard as SAT

If we make the clauses just a little bigger, then we get a quite different result. We will show here that if there is a polynomial time algorithm for deciding 3-SAT, then there is such an algorithm for SAT. So in a sense, 3-SAT is 'just as hard' as the more general problem.

To show this, we will see how to translate a CNF formula ϕ to a 3-CNF formula ψ such that ϕ is satisfiable if and only if ψ is. This algorithm runs in time polynomial in the size of the original formula ϕ, and the new formula ψ, while longer, has its size bounded by a polynomial in the size of ϕ. Thus if we had a polynomial time algorithm for 3-SAT, we obtain one for the general SAT problem: Translate the given formula ϕ to the 3-CNF formula ψ, and apply the algorithm for 3-SAT of ψ.

Here are the details. Let us pick a clause of ϕ that has more than three literals. We'll illustrate the procedure supposing that this clause has 5 literals, but this is just to make the notation simpler. Thus we write ϕ as

$$(p_1 \vee p_2 \vee p_3 \vee p_4 \vee p_5) \wedge \phi',$$

where ϕ' is the conjunction of all the other clauses of ϕ. We introduce a new variable z, and replace our formula above by

$$\rho = (p_1 \vee p_2 \vee z) \wedge (z \rightarrow (p_3 \vee p_4 \vee p_5)) \wedge \phi'.$$

Now suppose that there is a satisfying assignment for ϕ. If this assignment makes p_1 or p_2 true, then we can make z false, and the resulting assignment satisfies ρ. If p_1 and p_2 are both assigned false, then at least one of p_3, p_4, p_5 must be assigned true, so $z \rightarrow (p_3 \vee p_4 \vee p_5)$ is true regardless of the value of z. We can thus make z true, and the resulting assignment satisfies ρ. Thus if ϕ is satisfiable, ρ is as well. Conversely, if ρ is satisfiable with an assignment making z false, then p_1 or p_2 must be true; and if ρ is satisfiable with an assignment making z true, then p_3, p_4 or p_5 must be true. So restricting the satisfying assignment to the variables in ϕ gives a satisfying assignment for ϕ.

We can rewrite ρ as

$$(p_1 \vee p_2 \vee z) \wedge (z \vee p_3 \vee p_4 \vee p_5) \wedge \phi'.$$

We have replaced a clause with 5 literals by the conjunction of a clause with three literals and one with four literals. We can apply the same trick to our new clause with four literals. In general, if a clause has $k > 3$ literals, then in $k - 3$ steps we can transform it to the conjunction of $k - 2$ clauses with three literals. We do this clause by clause, finally getting a 3-CNF formula ψ that is satisfiable if and only if ϕ is. If the size of ϕ is n, then it has no more than n clauses and n variables. The new formula ψ will have fewer than n^2 clauses, each with 3 literals: this observation gives us our required polynomial time bound.

7 Polynomial-time reducibility

The last two arguments illustrate a general phenomenon. This is a version of reducibility, which we saw before in the context of decidability, applied to polynomial-time algorithms.

In the first case we had a problem, let's call it FORBIDDEN-CIRCUIT, in which the instances are digraphs with vertices $\{x_1, \ldots, x_n.\bar{x}_1, \ldots, \bar{x}_n\}$ and showed how to map instances of 2-SAT into instances of FORBIDDEN-CIRCUIT, so that yes-instances mapped to yes-instances and no-instances to no-instances. This is a reduction, just as we described earlier in the course. The important new property is that the reduction is carried out in polynomial time. We thus write this as

$$\text{2-SAT} \leq_P \text{FORBIDDEN-CIRCUIT},$$

and say that 2-SAT is polynomial-time reducible to FORBIDDEN-CIRCUIT.

Likewise, in the next section, we showed how to map instances of SAT to instances of 3-SAT in a way that preserved yes- and no-instances. We also argued that this mapping can be computed in polynomial time. Thus

$$\text{S}AT \leq_P \text{3-SAT}.$$