

6.046 Recitation 12 Handout
May 9, 2008

1 Variations of SAT

1.1 3-SAT

3-SAT is a special case of SAT, where the formula ϕ is written in conjunctive normal form with exactly 3 literals per clause (also called a 3-CNF). Literals are occurrences of a variable or its negation. A boolean formula is in conjunctive normal form if it is an AND of clauses, each of which is the OR of some number of literals. For example, the following is a 3-CNF:

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

The question for 3-SAT is thus whether the given boolean formula ϕ is satisfiable.

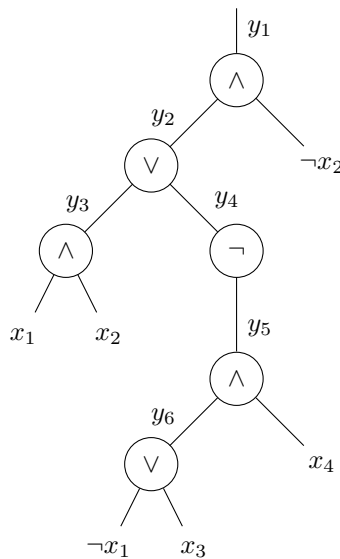
Theorem 1. *3-SAT is NP-complete.*

Proof. First of all, 3-SAT is clearly in NP, since it is a special case of SAT. We will prove that 3-SAT is NP-complete by showing that any boolean formula can be converted into a 3-CNF formula in polynomial time, and thus SAT reduces to 3-SAT.

For any given instance of SAT, we can build a binary “parse” tree for the boolean formula ϕ . For example, we can take the formula

$$\phi = ((x_1 \wedge x_2) \vee \neg((\neg x_1 \vee x_3) \wedge x_4)) \wedge \neg x_2$$

and construct the following binary parse tree. We also define a new variable y_i that indicates the value for each subtree of the parse tree as illustrated.



Using this representation, we can thus rewrite ϕ as a new boolean formula ϕ' that is true exactly when y_1 , the value of ϕ , is true and all variables y_i are constrained to be true exactly when their corresponding subtrees are true.

$$\begin{aligned}
\phi' = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge (y_3 \leftrightarrow (x_1 \wedge x_2)) \\
& \wedge (y_4 \leftrightarrow \neg y_5) \\
& \wedge (y_5 \leftrightarrow (y_6 \wedge x_4)) \\
& \wedge (y_6 \leftrightarrow (\neg x_1 \vee x_3))
\end{aligned}$$

Note that we now have a series of clauses where each clause involves at most 3 variables. A new clause and variable y_i is defined for each operator in the original formula, so the new formula ϕ' has length polynomial in the size of the original formula ϕ . Now we just have to get them into conjunctive normal form.

We convert each clause to CNF by writing out a truth table, constructing a formula in disjunctive normal form (DNF) for when the clause is false, and then using DeMorgan's laws to get a CNF formula for when the clause is true. A DNF formula is pretty much the opposite of a CNF formula—it is the OR of a series of clauses, each of which are ANDs of some number of literals.

For example, if we want to translate the clause $C'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$, we can basically enumerate all the combinations that make the clause false and write the DNF:

$$\neg C'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

Using DeMorgan's laws (the fact that $\neg(A \vee B) = (\neg A) \wedge (\neg B)$ and $\neg(A \wedge B) = (\neg A) \vee (\neg B)$), we can write the clause C'_1 as a new CNF:

$$C''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Note that since the number of rows in the truth table is a constant since there are at most 3 variables in the clause, we only increase the number of clauses by a constant factor.

The last step is to ensure that all clauses have exactly 3 literals. For all clauses that have less than 3 literals, we “pad” them with spare variable(s). For example, if a clause has 2 literals and is of the form $(x \vee y)$, then we use an extra variable p and replace the clause with two clauses $(x \vee y \vee p)$ and $(x \vee y \vee \neg p)$. No matter how we set p , one of x and y must be true for the two clauses to both be true. For clauses with only 1 literal, then we simply use 2 variables and replace the clause with 4 new clauses with all combinations of the literals of p, q , and their negations. Thus the formula is now a 3-CNF. This last step increases the number of clauses by only a constant factor, so the final boolean formula is also polynomial in length of the original formula, meaning that we have reduced SAT to 3-SAT. Therefore, 3-SAT is NP-complete. □

1.2 2-SAT

As might be expected, 2-SAT is the problem of determining the satisfiability of a 2-CNF formula. However, decreasing the number of literals in each clause actually makes the problem easier!

Theorem 2. *2-SAT is in P.*

Proof. We will prove this by providing a polynomial time algorithm.

First, we note that because this formula is made up of 2-literal clauses of the form $(x \vee y)$, we can infer implications of the form $\neg x \rightarrow y$ and $\neg y \rightarrow x$. We can construct a directed graph G with $2n$ vertices, one vertex for each possible literal. We draw an edge (u, v) if u implies v .

A subgraph of the graph is a strongly connected component if for every pair of vertices u and v in the subgraph, there is a directed path from u to v and from v to u . Intuitively, a strongly connected component in the graph we've created is a cluster of literals that all must be true at the same time or not at all.

We can then collapse each strongly connected component into a single vertex, resulting in a directed acyclic graph. (If there were cycles, then there'd be another strongly connected component that hasn't gotten collapsed.) We can then run a topological sort on this new graph using DFS (See the Recitation 7 Handout for info on the sort.).

First, we note that if x and $\neg x$ are in the same strongly connected component, then the formula is unsatisfiable because x would imply $\neg x$ and vice versa, which is a contradiction.

Otherwise, for all variables x , if x does not share a component with $\neg x$, then we can say that if the component of x appears topologically before that of $\neg x$, then we set x to false. Alternatively, if the components appear in the opposite order, we set x to true.

This assignment is a satisfying assignment. We prove this by contradiction. Suppose it is not a satisfying assignment. Then there is some clause $(x \vee y)$ that is false. This means that we had set both x and y to false. This would have happened exactly when the component containing $\neg x$ appears after the component containing x (and similarly for y). We know from the clause $(x \vee y)$ that we had the directed edges $(\neg x, y)$ and $(\neg y, x)$. These edges imply that the component containing $\neg x$ is before or the same as the component containing y , and the component containing $\neg y$ is before or the same as the component containing x . But if at the same time the component containing $\neg x$ must be later than the component containing x , this would imply that the component containing y must be later than the component containing $\neg y$; a contradiction.

Running time: $O(n)$ for DFS, etc. □

2 Subset Sum

We are given a finite set S of integers and a target t . The problem is that of determining whether there exists a subset of the integers in S that add up to the target t . For example, if $S = \{4, 11, 16, 21, 27\}$ and $t = 25$, then there exists such a subset since $4 + 21 = 25$.

Theorem 3. *The subset-sum problem is NP-complete.*

Proof. First, the subset-sum problem is in NP. If we are given a subset of numbers from S , we can add them up and verify that their sum is t in polynomial time.

We will reduce 3-SAT to the subset-sum problem to show that it is NP-complete. Again, we have m clauses over n variables with 3 literals in each clause. Without loss of generality, we can make the following assumptions:

- No clause contains both some variable x and $\neg x$. Any such clause is always going to be true, so we can just eliminate them from the formula.
- Each variable appears in at least one clause. Otherwise, we can just assign any arbitrary value to the variable and do not need to consider it in the formula.

These assumptions are necessary for our reduction as we will see later.

To convert a problem in 3-SAT to that of a subset-sum problem, we will work with integers that each have exactly $m + n$ digits in base 10. We then define the numbers:

- Set the target t to n 1's followed by m 4's.

- For the set S , we construct two numbers v_i and v'_i for the literals x_i and $\neg x_i$. For both of the values, the first n digits serve to indicate which variable this is—the i th digit is set to 1 and all others are 0. For the last m digits, the j th digit is set to 1 if the j th clause contains the respective literal. Note that $v_i \neq v'_i$ because of our earlier assumptions; they cannot both be in the same clause, and at least one of them has to have nonzero values in the last m digits.
- We also add the following numbers to S : two slack variables s_j and s'_j for each clause. The first n digits are 0, and in the last m digits, all are 0 except the j th index. For s_j , that value should be set to 1, and for s'_j , that value should be set to 2.

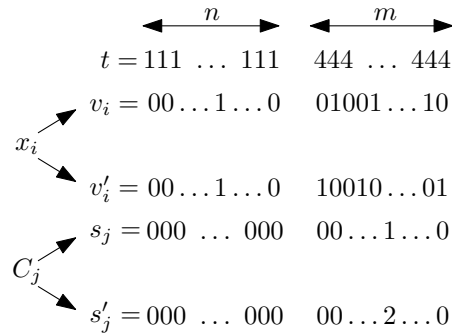


Figure 1: An example of the formation of t , v_i , v'_i , s_j , and s'_j .

Now that we have defined the numbers, note that the highest digit achievable is 6, meaning that no digit will ever carry over and interfere with other digits.

The original 3CNF formula is satisfiable if and only if there is a subset in S that adds up to t .

\Rightarrow : Assuming that there is a satisfying assignment, pick a subset containing all v_i for which x_i is set to true and v'_i for x_i set to false. We should then be able to achieve t by picking the necessary s_j and s'_j to get 4's in all the last m digits.

Since we pick one number for each variable x_i , we get a 1 in each index in the first n digits of our sum. The slack variables do not contribute to those digits. In the last m digits, we know that each clause had to be satisfied for this to be a satisfying assignment, so for each digit, at least one literal had to have a 1 in that position. In fact, in the sum without slack variables, each of those digits is 1, 2, or 3. We can therefore pick the appropriate combination of s_j and/or s'_j to add 3, 2, or 1 to that digit to achieve the value of 4.

\Leftarrow : Assuming that there is a subset $S' \subseteq S$ that adds up to t , it corresponds to a satisfying assignment. S' must include exactly one of v_i and v'_i or otherwise there'd be no other way to achieve a value of 1 in the i th digit. If $v_i \in S'$, set x_i to true; otherwise, set it to false. This must be a satisfying assignment. Since the slack variables can only add up to 3 for a particular digit and we have a subset that adds up to the target, at least one literal in each clause must be assigned to true. We thus have a satisfying assignment.

□