

## 3SAT

- **Definition:** A literal is either a variable or its negation (i.e. either  $x$  or  $\bar{x}$ ). A clause is a disjunction (OR) of literals. A formula  $\varphi$  is in CNF (“Conjunctive Normal Form”) if it is a conjunction (AND) of clauses. It is in 3CNF if it is in CNF and every clause has exactly three literals.
- Example: the following formula is in CNF (but not 3CNF):

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_3)$$

- Note that, if a formula  $\varphi$  is in CNF and every clause has at most three literals, then  $\varphi$  can easily be converted to 3CNF by repeating a literal in every clause of size two or less. For example, the formula in the above example can be modified as follows:

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_3 \vee \bar{x}_3 \vee \bar{x}_3)$$

- **Definition:**  $3SAT = \{\langle \varphi \rangle \mid \varphi \text{ is a satisfiable formula in 3CNF}\}$ .
- **Theorem:** 3SAT is **NP**-complete.
- **Proof:** we need to show that 3SAT is in **NP**, and that 3SAT is **NP**-hard.
  - The non-deterministic algorithm for 3SAT is the same as for SAT: on input  $\varphi$ , guess a truth assignment  $\tau$  and evaluate  $\varphi(\tau)$ , and accept iff  $\varphi(\tau) = 1$ . Note that in the case of 3SAT, checking whether  $\varphi(\tau) = 1$  is even easier, as we need only check that every clause has one literal that is true under  $\tau$ .
  - To prove 3SAT is **NP**-hard, we can show  $SAT \leq_p 3SAT$ . Since SAT is **NP**-hard, it then follows that 3SAT is **NP**-hard. The idea of the reduction is as follows: given a formula  $\varphi$ , consider  $\varphi$  as a rooted tree  $T = (V, E)$  whose leaves are labelled by variables, and whose internal nodes are labelled by the connectives  $\vee, \wedge, \neg$ . Let  $\{x_1, \dots, x_n\}$  be the variables occurring in  $\varphi$ . We will output a formula  $\psi$  whose variables are  $\{x_1, \dots, x_n\} \cup \{y_v\}_{v \in V}$ , i.e.  $\psi$  has an additional variable for each connective in  $\varphi$ . A truth assignment  $\tau$  for  $\{x_1, \dots, x_n\}$  implicitly assigns a truth value to every node  $v \in V$  (the truth value of a leaf is just the value of the variable that labels it; an internal node  $v$  labelled  $\wedge$  has the value 1 iff both its children have value 1, and so on). The clauses of  $\psi$  are described below.
  - For each leaf  $v$  of  $T$  labelled  $x_i$ ,  $\psi$  contains clauses that assert

$$\begin{aligned} (y_v \leftrightarrow x_i) &\equiv (y_v \rightarrow x_i) \wedge (x_i \rightarrow y_v) \\ &\equiv (\bar{y}_v \vee x_i) \wedge (y_v \vee \bar{x}_i) \end{aligned}$$

Thus we need to add two clauses, each containing two literals, for each leaf of  $T$ .

- For each internal node  $v$  of  $T$  labelled  $\neg$ , with child  $u$ ,  $\psi$  contains clauses

$$(y_v \leftrightarrow \overline{y_u}) \equiv (\overline{y_v} \vee \overline{y_u}) \wedge (y_v \vee y_u)$$

Thus for these nodes we also need to add two clauses, each of size two.

- For each internal node  $v$  of  $T$  labelled  $\wedge$ , with children  $u$  and  $w$ ,  $\psi$  contains clauses

$$\begin{aligned} (y_v \leftrightarrow (y_u \wedge y_w)) &\equiv (\overline{y_v} \vee (y_u \wedge y_w)) \wedge (\overline{(y_u \wedge y_w)} \vee y_v) \\ &\equiv (\overline{y_v} \vee y_u) \wedge (\overline{y_v} \vee y_w) \wedge (\overline{y_u} \vee \overline{y_w} \vee y_v) \end{aligned}$$

So for these nodes we have added three clauses, each of size three or less.

- For each internal node  $v$  of  $T$  labelled  $\vee$ , with children  $u$  and  $w$ ,  $\psi$  contains clauses

$$\begin{aligned} (y_v \leftrightarrow (y_u \vee y_w)) &\equiv (\overline{y_v} \vee y_u \vee y_w) \wedge (\overline{(y_u \vee y_w)} \vee y_v) \\ &\equiv (\overline{y_v} \vee y_u \vee y_w) \wedge ((\overline{y_u} \wedge \overline{y_w}) \vee y_v) \\ &\equiv (\overline{y_v} \vee y_u \vee y_w) \wedge (\overline{y_u} \vee y_v) \wedge (\overline{y_w} \vee y_v) \end{aligned}$$

So for these nodes we have also added three clauses, each of size three or less.

- Finally,  $\psi$  contains the clause  $(y_r)$ , where  $r$  is the root of  $T$  (note:  $\psi$  is the conjunction (AND) of all the clauses described above)

- **Claim:**  $\varphi$  is satisfiable iff  $\psi$  is satisfiable
- **Proof:** ( $\Rightarrow$ ) Assume a truth assignment  $\tau : \{x_1, \dots, x_n\} \mapsto \{0, 1\}$  satisfies  $\varphi$ . As we said above,  $\tau$  implicitly assigns a truth value to each node  $v$  in  $T$ . Define  $\tau'$  to be the same as  $\tau$  on the variables  $x_1, \dots, x_n$ , and set  $\tau'(y_v)$  equal to the truth value assigned by  $\tau$  to  $v$ . Then  $\tau'$  satisfies each of the clauses corresponding to the leaves of  $T$ , and it also satisfies each of the clauses corresponding to the internal nodes of  $T$ . Since  $\tau$  satisfies  $\varphi$ ,  $\tau$  must assign a value of 1 to the root  $r$ , and therefore  $\tau'(y_r) = 1$ . Thus  $\tau'$  satisfies  $\psi$ .

( $\Leftarrow$ ) Assume a truth assignment  $\tau'$  satisfies  $\psi$ . Define a truth assignment  $\tau$  on  $\{x_1, \dots, x_n\}$  that is identical to  $\tau'$  on these variables, i.e.  $\tau(x_i) = \tau'(x_i)$ , for all  $1 \leq i \leq n$ . Then  $\tau$  is a satisfying truth assignment for  $\varphi$ : the value which it implicitly assigns to each node  $v \in T$  is identical to the value assigned by  $\tau'$  to  $y_v$ , and since  $\tau'(y_r) = 1$  it follows that  $\tau$  assigns a value of 1 to the root  $r$ , i.e.  $\varphi(\tau) = 1$ .

- We must also argue that  $\psi$  can be computed from  $\varphi$  in polynomial time. Note that parsing  $\varphi$  into a tree structure done in quadratic time by searching for the “top-level” connective, recursively parsing the subformulae on each side into two trees, and then joining these two trees with a new node labelled by the top-level connective. Once this structure has been found, the formula  $\psi$  can be generated in linear time by traversing the tree and outputting the appropriate clauses for each node.

## Decision/Search/Optimization problems

- Until now, we have been concerned with the decidability of languages. This corresponds to a type of problem called a “decision problem”, where the answer is either “yes” or “no”. For example:
  - SAT or 3SAT: is the formula  $\varphi$  satisfiable?
  - CLIQUE: does the graph  $G$  have a clique of size  $k$ ?
  - SUBSET-SUM: is there a subset of  $S$  that sums to  $t$ ?
- In real life, we are generally more interested in *finding* a solution than in simply ascertaining whether one exists. This leads to the definition of corresponding search problems. For example, the problem SAT-SEARCH is: given a formula  $\varphi$ , output a satisfying truth assignment if one exists. Similarly, the problem CLIQUE-SEARCH is: given a graph  $G$  and a number  $k$ , output a clique of size at least  $k$  in  $G$  if one exists.
- Often we are interested in finding a solution that optimizes some criterion: for example, we may be interested in finding the largest clique in a graph. This type of problem is called an optimization problem. For example, the problem CLIQUE-OPT is: given a graph  $G$ , output a largest clique in  $G$ .
- It is clear that, in some sense, optimization problems are “harder” than search problems, and that search problems are “harder” than decision problems: for example, a solver for CLIQUE-OPT can be easily used to solve CLIQUE-SEARCH, by finding a largest clique  $C$  and outputting it if it has size  $k$  or larger (otherwise no solution exists). Similarly, a solver for CLIQUE-SEARCH can be used to decide CLIQUE by asking it to search for a clique of size  $k$ , and accepting iff one is found.
- What is not as obvious is that, for the problems we are studying, the converse is also true: a polynomial-time algorithm for the decision problem can be used to obtain a polynomial-time algorithm for the corresponding search problem or optimization problem.

**Example: 3SAT**

- The problem 3SAT-SEARCH is:

**3SAT-SEARCH**

**Instance:** A formula  $\varphi$  in 3CNF

**Output:** A truth assignment  $\tau$  such that  $\varphi(\tau) = 1$ , if one exists; or else  $\perp$

- **Theorem:** If 3SAT can be decided in polynomial time, then 3SAT-SEARCH can be solved in polynomial time.
- First we need a definition. If  $\varphi$  is a formula in CNF, and  $x$  is a variable of  $\varphi$ , then  $\varphi|_{x=1}$  is the formula obtained from  $\varphi$  by the following modifications:

- For each clause  $C$  of  $\varphi$  containing the literal  $x$ , delete  $C$
- For each clause  $C$  of  $\varphi$  containing the literal  $\bar{x}$ , remove  $\bar{x}$  from  $C$

Similarly  $\varphi|_{x=0}$  is obtained by the following modifications:

- For each clause  $C$  of  $\varphi$  containing the literal  $\bar{x}$ , delete  $C$
- For each clause  $C$  of  $\varphi$  containing the literal  $x$ , remove  $x$  from  $C$

- The idea is that  $\varphi|_{x=1}$  corresponds to the formula obtained by replacing  $x$  with the value 1, and making simplifications using the identities  $F \vee 0 \equiv F$ ,  $F \vee 1 \equiv 1$ , and  $F \wedge 1 \equiv F$ . Similarly  $\varphi|_{x=0}$  corresponds to replacing  $x$  with the value 0 and simplifying.
- Now assume that  $M$  is a TM that decides 3SAT in polynomial machine. Then the following TM  $M'$  solves 3SAT-SEARCH in polynomial time:  $M' =$  “on input  $\langle \varphi \rangle$ ...

1. run  $M$  on  $\langle \varphi \rangle$ ; if it rejects then output  $\perp$
2. for  $i \leftarrow 1, \dots, n$  do
3.     run  $M$  on  $\langle \varphi|_{x=1} \rangle$
4.     if it accepts then
5.          $\varphi \leftarrow \varphi|_{x=1}$
6.          $\tau(x_i) \leftarrow 1$
7.     else
8.          $\varphi \leftarrow \varphi|_{x=0}$
9.          $\tau(x_i) \leftarrow 0$
10. output  $\tau$

If step 1 is passed, then the formula is satisfiable. Thus it either has a satisfying assignment that sets  $x_1 = 1$ , or it has a satisfying assignment that sets  $x_1 = 0$  (or both). Using  $M$  we can check whether  $\varphi|_{x_1=1}$  is satisfiable: if so, then we set  $\tau(x_1) = 1$  and search for a satisfying assignment for  $\varphi|_{x_1=1}$ . If not, then  $\varphi|_{x_1=0}$  is satisfiable, so we set  $\tau(x_1) = 0$  and search for a satisfying assignment for  $\varphi|_{x_1=0}$ .

**Example: CLIQUE**

- The problem CLIQUE-OPT is defined as:

**CLIQUE-OPT**

**Instance:**  $G = (V, E)$   
**Solution:** A clique  $C$  in  $G$   
**Objective:** Maximize  $|C|$

That is, the problem is to find a clique in the input graph of maximum size.

- **Theorem:** If CLIQUE is decidable in polynomial time, then CLIQUE-OPT can be solved in polynomial time.
- **Proof:** Suppose  $M$  is a polynomial-time Turing machine that decides CLIQUE. Then the following TM  $M'$  solves CLIQUE-OPT in polynomial time:  $M' =$  “on input  $\langle G \rangle$ ...
  1. for  $i \leftarrow 1, \dots, n$  do
  2.     run  $M$  on input  $\langle G, k \rangle$ ; if it accepts then set  $k \leftarrow i$
  3.  $C \leftarrow V$
  4. for each  $v \in V$  do
  5.     let  $G'$  be the subgraph of  $G$  induced by  $C \setminus \{v\}$
  6.     run  $M$  on input  $\langle G', k \rangle$
  7.     if  $M$  accepts then set  $C \leftarrow C \setminus \{v\}$
  8. return  $C$

At the end of the loop in lines 1 – 2,  $k$  is equal to the size of the largest clique in  $G$ . The remainder of the algorithm attempts to find a clique of size  $k$  in  $G$ . The idea is to remove a vertex, and ask whether the resulting graph  $G'$  still has a clique of size  $k$ . If so, then we can throw away that vertex and keep looking for a clique of size  $k$  in the remaining graph. If not, then that vertex was necessary (i.e. it is a vertex in the only  $k$ -clique in  $G$ ), so we keep it and continue.

- It is clear that the above algorithm always outputs a set  $C$  that *contains* a clique of size  $k$ , since it only removes a vertex from  $C$  if the resulting set contains a clique of size  $k$ . What is not as obvious is that the output  $C$  does not contain extraneous vertices. To see that this is indeed true, notice that if the output set  $C$  has size  $k + 1$  or greater, then it contains a vertex  $v$  such that  $C \setminus \{v\}$  contains a  $k$ -clique. But then at the point when  $v$  was considered, the set  $C'$  computed by the algorithm at that point was a superset of  $C$ , and so  $v$  would have been discarded (as  $C' \setminus \{v\}$  contains the same  $k$ -clique that is contained in  $C \setminus \{v\}$ ).