Logic for Computer Science - Week 4
The SAT Problem

Ștefan Ciobâcă

November 30, 2017

1 Reminder

So far we have discussed two important topics in Formal Logic:

1. The syntax of propositional logic;
2. The semantics of propositional logic.

The syntax of propositional logic tells us with absolute precision what are the possible propositional formulae, and the semantics tells what is is the meaning of each such formula. See the previous lectures in order to recall that the syntax is given as an inductive definition of the set PL (the set of propositions formulae). The semantics of a propositional formula is a truth value that depends on both the formula and a truth assignment $\tau : A \to B$. The truth value of the formula φ in the truth assignment τ is denoted by $\hat{\tau}\Big(\varphi\Big)$.

Recall that a formula φ is said to be satisfiable iff there exists a truth assignment that makes it true, i.e. if there exists $\tau : A \to B$ such that $\hat{\tau}\Big(\varphi\Big) = 1$. Also recall that when $\hat{\tau}\Big(\varphi\Big) = 1$, we say that τ satisfies φ and that the assignment τ is a model of φ. Also recall from the last lecture the notions of validity, semantical consequence and equivalence.

2 Introduction to The SAT Problem

The satisfiability problem (or simply the SAT problem) is the following computational problem:

Input A formula $\varphi \in PL$;

Output "yes", if φ is satisfiable; "no", otherwise.

The satisfiability problem is a central problem is computer science because it occurs in various domains such as combinatorial optimization problems (e.g.,

finding a Hamiltonian path in a graph is equivalent to checking satisfiability of a certain propositional formula), verification (e.g., checking the functional equivalence of two circuits), medicine (e.g., the protein folding problem) and others.

So far, noone has come up with a fast algorithm for SAT (by fast we understand polynomial-time in the length of the formula) – all of the algorithms proven to solve SAT work in exponential time in the worst case. But also we have not been able to show that any algorithm solving SAT takes at least exponential time. This open problem, of either finding a fast algorithm for SAT or proving that none exists, is known as the P versus NP problem and it is so important (and difficult) that the Clay institute is offering 1 000 000 dollars to the person solving it (`http://www.claymath.org/millennium-problems`).

Despite the fact that, in the worst case, all algorithms proven to solve the SAT problem work in exponential time, the problem is of such importance in practice that many computer science researchers and practitioners spend a lot of time fine tuning *SAT solvers* – programs that solve the SAT problem. There is actually an important yearly event (the SAT conference, `http://www.satisfiability.org/`) dedicated to the SAT problem and a yearly competition (the SAT competition, `http://www.satcompetition.org/`) dedicated to benchmarking SAT solvers by the number of SAT problem they can solve in a given unit of time.

2.1 Plan of the Lecture

We will first study a variation of the SAT problem known as CNF-SAT. CNF-SAT is apparently easier than SAT, but it turns out that both problems are equally difficult (if we know how to solve one of them fastly, then we can also construct a fast algorithm for the other). Most SAT solvers actually solve the CNF-SAT problem (but since the two problems are computationally equivalent, this does not constitute a restriction).

We will then study a backtracking algorithm for solving SAT. Most current SAT solvers are optimized version of the basic backtracking algorithm.

We will then spend some time on applications of SAT, most notably SUDOKU solving.

3 Notational Conventions

In this section, we will go over a series of important notations that we will be using throughout the rest of the course.

3.1 Implication

We will use the notation $(\varphi_1 \to \varphi_2)$ as a shorthand for $(\neg\varphi_1 \vee \varphi_2)$. As an example, whenever we write $(\mathtt{p} \to \mathtt{q})$, our brain instantaneously transforms it into the formula $(\neg\mathtt{p} \vee \mathtt{q})$.

Therefore, since $(\varphi_1 \to \varphi_2)$ is just a notation for $(\neg\varphi_1 \vee \varphi_2)$, it follows that whenever $\varphi_1, \varphi_2 \in PL$, we have $(\varphi_1 \to \varphi_2) \in PL$ as well.

The symbol $\to$ is read as *implies* and it is a logical connective. In contrast to the logical connectives $\{\neg, \vee, \wedge\}$, it is defined through a notation.

3.2 Biconditional

Another important logical connective defined as a notation is the *biconditional connective* $\leftrightarrow$. We will use the notation $(\varphi_1 \leftrightarrow \varphi_2)$ for the formula $((\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1))$.

For example, the formula $(\mathtt{p} \leftrightarrow \mathtt{q})$ is a notation for $((\mathtt{p} \to \mathtt{q}) \wedge (\mathtt{q} \to \mathtt{p}))$, which is itself a notation for $((\neg\mathtt{p} \vee \mathtt{q}) \wedge (\neg\mathtt{q} \vee \mathtt{p}))$.

3.3 Alternative Viewpoint

An alternative point of view of the conditional $(\to)$ and the biconditional $(\leftrightarrow)$ connectives is that we may enlarge the alphabet to contain the symbols $\{\to, \leftrightarrow\}$ and then change the definition of PL (given in Lecture 2) with two new inductive cases:

$\vdots$

1. (Inductive Step iv) If $\varphi_1, \varphi_2 \in PL$, then $(\varphi_1 \to \varphi_2) \in PL$;
2. (Inductive Step v) If $\varphi_1, \varphi_2 \in PL$, then $(\varphi_1 \leftrightarrow \varphi_2) \in PL$;

Following this change, we must also change all functions recursively defined on formulae in PL. Most importantly, we need to update the semantics of propositional logic (see Lecture 3) with two new cases so that

$$\hat{\tau}\Big((\varphi_1 \to \varphi_2)\Big) = \overline{\hat{\tau}\Big(\varphi_1\Big)} + \hat{\tau}\Big(\varphi_2\Big)$$

and

$$\hat{\tau}\Big((\varphi_1 \leftrightarrow \varphi_2)\Big) = (\overline{\hat{\tau}\Big(\varphi_1\Big)} + \hat{\tau}\Big(\varphi_2\Big)) \cdot (\overline{\hat{\tau}\Big(\varphi_2\Big)} + \hat{\tau}\Big(\varphi_1\Big)).$$

It turns out that both points of view are really the same, in the sense that do not change any of the results that follow. However, whenever you are confused about $\to$ or $\leftrightarrow$, you should just think of $\to$ and $\leftrightarrow$ as defined by the notations introduced previously.

3.4 Getting Rid of (some) Brackets

The brakets are important in the definition of the formal syntax of propositional logic, as they make formulae syntactically unambiguous. In the absense of brackets in the definition of PL, the formula $\neg p \vee q$ could have been ambiguously understood as either $(\neg p \vee q)$ or $\neg(p \vee q)$.

However, brakets clutter our writing and, now that we have understood very well the formal syntax of propositional logic, we will make our lives easier by introducing a few simplifying notations.

First of all, we will make an analogy with arithmetic. When we write $3+4\times5$, it is well understood that we mean $3 + (4 \times 5)$ and not $(3 + 4) \times 5$. This is because of the priority of arithmetic operands: multiplication ($\times$) binds tighter than addition ($+$).

We will adopt a similar priority order for logical connectives:

$$\neg, \wedge, \vee, \rightarrow, \leftrightarrow .$$

That is, $\neg$ binds tightest, followed by $\wedge$, then $\vee$, then $\rightarrow$ and finally $\leftrightarrow$.

Therefore, a formula such as

$$\neg \mathtt{p} \vee \mathtt{q} \wedge \mathtt{r} \rightarrow \mathtt{r} \leftrightarrow \mathtt{p}$$

is actually a notation for

$$(((\neg \mathtt{p} \vee (\mathtt{q} \wedge \mathtt{r})) \rightarrow \mathtt{r}) \leftrightarrow \mathtt{p}).$$

We have seen in the previous lecture that, for any formulae $\varphi_1, \varphi_2, \varphi_3 \in PL$, the following equivalences hold:

1. $((\varphi_1 \vee \varphi_2) \vee \varphi_3) \equiv (\varphi_1 \vee (\varphi_2 \vee \varphi_3))$ (associativity of $\vee$);
2. $(\varphi_1 \vee \varphi_2) \equiv (\varphi_2 \vee \varphi_1)$ (commutativity of $\vee$);
3. $((\varphi_1 \wedge \varphi_2) \wedge \varphi_3) \equiv (\varphi_1 \wedge (\varphi_2 \wedge \varphi_3))$ (associativity of $\wedge$);
4. $(\varphi_1 \wedge \varphi_2) \equiv (\varphi_2 \wedge \varphi_1)$ (commutativity of $\wedge$).

Recall that two formulae are equivalent if, in any truth assignment τ, they both turn out to have the same truth value. Therefore, even though syntactically they are two different formulae, semantically they have the same meaning. For example, by applying associativity and commutativity of $\vee$, it follows easily that $((\mathtt{p} \vee \mathtt{q}) \vee \mathtt{r}) \equiv (\mathtt{p} \vee (\mathtt{r} \vee \mathtt{q}))$.

This justifies our next notation. We will write

$$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$$

instead of

$$(((\varphi_1) \vee \varphi_2) \vee \ldots \vee \varphi_n)$$

or any or permutation and bracketing of the n formulae $\varphi_1, \ldots, \varphi_n$. For example, we will write $\mathtt{p} \vee \mathtt{q} \vee \mathtt{r}$ instead of any of $((\mathtt{p} \vee \mathtt{q}) \vee \mathtt{r})$, $(\mathtt{r} \vee (\mathtt{q} \vee \mathtt{p}))$, etc. When the exact syntactic variation is important (almost never), we will use brackets as before to distinguish among the various possibilities.

4 The Conjunctive Normal Form and SAT

Now that we have introduced the simplifying notations in the previous section, we are ready to define what it means for a formula to be in *conjunctive normal form*.

Definition 4.1 (Literal)**.** *A formula $\varphi \in PL$ is called a literal if there exists a propositional variable $a \in A$ such that*

$$\varphi = a \text{ or } \varphi = \neg a.$$

Example 4.1. *The following formulae are all literals:*

$$\mathtt{p}, \neg\mathtt{p}, \mathtt{q}, \mathtt{r}', \neg\mathtt{r}_1 \in PL.$$

None of the following formulae is a literal:

$$\mathtt{p} \vee \mathtt{q}, \neg\mathtt{p} \wedge \mathtt{q}, \mathtt{p} \to \mathtt{q}, \neg\neg r \in PL.$$

We may sometimes paraphrase the definition above by saying that a literal is either a propositional variable or the negation of a propositional variable.

Note that the formula $\neg\neg\mathtt{r}$ is not a literal (it is neither a propositional variable nor the negation of a propositional variable). However, the formula $\neg\neg\mathtt{r}$ is equivalent to the formula $\mathtt{r}$, which is a literal.

Definition 4.2 (Clause)**.** *A formula $\varphi \in PL$ is called a* clause *if there exist the literals $\varphi_1, \varphi_2, \ldots, \varphi_n$ such that*

$$\varphi = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n.$$

We sometimes paraphrase the above definition as "a clause is a disjunction of literals".

Note that in the above definition we are critically using the notation described in the previous section.

Example 4.2. *The following formulae are all clauses:*

$$\mathtt{p} \vee \mathtt{q} \vee \mathtt{r}, \neg\mathtt{p} \vee \mathtt{q}, \neg\mathtt{p} \vee \neg\mathtt{r}, \mathtt{p} \to \mathtt{q}, \mathtt{p}, \neg\mathtt{p}.$$

None of the following formulae is a clause:

$$\mathtt{p} \wedge \mathtt{q}, \neg\neg\mathtt{p} \vee \mathtt{r}, \neg\mathtt{p} \to \mathtt{q}, \neg\neg\mathtt{p}.$$

The formula $\mathtt{p} \to \mathtt{q}$ is a clause since we agreed that it is a notation for $\neg\mathtt{p} \vee \mathtt{q}$, which is clearly a clause. Any literal is a clause (it is the disjunction of one literal).

Note that $\neg\mathtt{p} \to \mathtt{q}$ is not a clause since it is a notation for $\neg\neg\mathtt{p} \vee \mathtt{q}$, which is clearly not a clause.

Definition 4.3 (CNF)**.** *A formula φ is in* conjunctive normal form *if there exist the clauses $\varphi_1, \varphi_2, \ldots, \varphi_n$ such that*

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n.$$

That is, a formula is in conjunctive normal form if it is a conjunction of clauses. Sometimes, *conjunctive normal form* is also referred to as *clausal normal form* (conveniently, the abbreviation CNF works for both).

Example 4.3. *The following formulae are all in CNF:*

1. $(\neg \mathtt{p} \vee \mathtt{q}) \wedge (\mathtt{p} \vee \mathtt{q} \vee \mathtt{r}) \wedge (\neg \mathtt{q} \vee \neg \mathtt{r})$ *(a conjunction of three clauses);*
2. $(\neg \mathtt{p} \vee \mathtt{q})$ *(a conjunction of one clause);*
3. $(\neg \mathtt{p} \vee \mathtt{q}) \wedge \mathtt{p}$ *(a conjunction of two clauses, the second of which happens to be a literal);*
4. $\neg \mathtt{p} \wedge \mathtt{q}$ *(a conjunction of two clauses, which both happen to be literals);*
5. $\mathtt{r}$ *(a conjunction of one clause, which happens to be a literal).*

None of the following formulae are in CNF:

1. $\neg(\mathtt{p} \vee \mathtt{q}) \wedge (\neg \mathtt{p} \vee \mathtt{q})$ *(the first conjunct is not a clause);*
2. $\neg \mathtt{p} \vee (\mathtt{p} \wedge \mathtt{q})$ *(not a conjunction of clauses);*
3. $\neg\neg \mathtt{p}$.

CNFs are important because any formula is equivalent to a formula in CNF (see next section).

5 The CNF-SAT Problem

The CNF-SAT problem is the following:

Input A formula $\varphi \in PL$ that is known to be in CNF;

Output "yes", if φ is satisfiable; "no", otherwise.

Recall that SAT is the following problem:

Input A formula $\varphi \in PL$;

Output "yes", if φ is satisfiable; "no", otherwise.

Since the inputs of CNF-SAT are less general that the inputs of SAT, it follows that CNF-SAT is "easier": any algorithm solving SAT will solve CNF-SAT as well, but the reverse is not necessarily true. An algorithm for CNF-SAT could make use of the fact that it only needs to process formulas in CNF to speed up processing. Therefore, at first sight, there could be a fast algorithm for CNF-SAT, but not one for SAT.

We will see that in fact, this is not the case. Any fast algorithm solving CNF-SAT will lead to a similarly fast algorithm for SAT and therefore we will say that the two problems are computationally equivalent. We will prove this next.

6 Bringing a Formula in CNF

Theorem 6.1. *For any formula $\varphi \in PL$, there exists a formula $\varphi' \in PL$, such that:*

1. $\varphi \equiv \varphi'$;
2. φ' *is in conjunctive normal form.*

The formula φ' will then be called a conjunctive normal form of φ. Note that φ' is by no means unique (there may be several conjunctive normal forms of a formula).

The theorem above is constructive, in the sense that its proof suggests an algorithm to compute the formula φ' with the above properties from φ. We will describe the algorithm and sketch a proof of why it works as advertised.

We will first describe the algorithm on an example:

Example 6.1. *We will compute a CNF of the formula $\neg(\mathtt{p} \wedge (\mathtt{q} \vee \neg \mathtt{r}))$.*

We will make use of the following equivalences that hold for any $\varphi_1, \varphi_2, \varphi_3 \in PL$:

E1 $\neg(\varphi_1 \vee \varphi_2) \equiv (\neg\varphi_1 \wedge \neg\varphi_2)$ *(De Morgan's law);*

E2 $\neg(\varphi_1 \wedge \varphi_2) \equiv (\neg\varphi_1 \vee \neg\varphi_2)$ *(De Morgan's law);*

E3 $\neg\neg\varphi_1 \equiv \varphi_1$;

E4 $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$ *(left-distributivity of $\vee$ over $\wedge$).*

E5 $(\varphi_2 \wedge \varphi_3) \vee \varphi_1 \equiv (\varphi_2 \vee \varphi_1) \wedge (\varphi_3 \vee \varphi_1)$ *(right-distributivity of $\vee$ over $\wedge$).*

Getting back to our example, we obtain that:

$$
\begin{aligned}
\neg(\mathtt{p} \wedge (\mathtt{q} \vee \neg \mathtt{r})) &\equiv (\neg \mathtt{p} \vee \neg(\mathtt{q} \vee \neg \mathtt{r})) && \textit{by equivalence E2} \\
&\equiv (\neg \mathtt{p} \vee (\neg \mathtt{q} \wedge \neg\neg \mathtt{r})) && \textit{by equivalence E1} \\
&\equiv (\neg \mathtt{p} \vee (\neg \mathtt{q} \wedge \mathtt{r})) && \textit{by equivalence E3} \\
&\equiv ((\neg \mathtt{p} \vee \neg \mathtt{q}) \wedge (\neg \mathtt{p} \vee \mathtt{r})) && \textit{by equivalence E4}
\end{aligned}
$$

Now that we have seen an example, we will give a proof sketch of how to proceed in general, for any formula φ.

Proof Sketch of Theorem 6.1. Suppose $\varphi \in PL$ is an arbitrary formula.

We will apply rules E1-E4 for as long as possible. Obviously, the resulting formula, which we call φ', is equivalent to the initial formula φ.

Imagine the proof tree of the resulting formula. Above a $\wedge$ node there can only be other $\wedge$ nodes (since otherwise one of rules E2, E4, E5 would apply). Above $\vee$ nodes, there can only be $\wedge$s or other $\vee$s (other rule E1 would apply). Finnaly, above $\neg$ nodes there can only be $\vee$s (or possibly directly $\wedge$s), since otherwise rule E3 would apply. Therefore, we will obtain an abstract syntax tree with $\wedge$s towards the top $\vee$s in the middle and $\neg$s towards to bottom, which is exactly what a CNF is.

A subtle point of the proof is that the process of applying rules E1-E5 always stops. We can show that such a process stops by showing it always decreases some quantity that cannot decrease forever. In our case, we will not make a formal proof, but note intuitively that all of the rules either push the $\neg$s towards the bottom, or eliminate two sequential $\neg$s.

□

The only downside of the algorithm suggested by Theorem 6.1 is that there is not guarantee of the size of the formula φ'. Indeed, consider the following example:

Example 6.2. *Consider the class of formulae*

$$\varphi_n = (\mathtt{p}_1 \wedge \mathtt{q}_1) \vee (\mathtt{p}_2 \wedge \mathtt{q}_2) \vee \ldots \vee (\mathtt{p}_n \wedge \mathtt{q}_n),$$

where n ranges over the positive naturals.

Following the algorithm above, we obtain the following conjunctive φ'_n normal form of φ:

$$\varphi'_n = \bigwedge_{a_i \in \{p,q\}} (a_1 \vee a_2 \vee \ldots \vee a_n).$$

In other words, φ'_n is a conjunction of 2^n clauses. This constitutes what is called an exponential growth, and exponential growths are generally bad when they can be avoided. For example, when $n = 10$, we have that φ'_{10} has 1024 clauses, when $n = 20$ we have that φ'_{20} has 1048576 clauses, etc. In other words, when n increases by 1 unit, the resulting formula doubles its size.

Unfortunately, due to the exponential blowup explained in the previous example, it is not possible to use this transformation to obtain a fast algorithm for SAT from a fast algorithm for CNF-SAT.

7 Tseitin's Transformation

In order to avoid the exponential blowup, Tseitin (sometimes spelled Tseytin) proposed an algorithm that avoid the exponential blowup. Tseitin's transformation (or Tseitin's algorithm) takes as input a formula φ and produces a formula φ' that is in CNF. However, unlike the previous transformation, there is a weaker connection between φ and φ' in general: φ is only guaranteed to be equisatisfialble to φ', not equivalent. Two formulae are equisatisfiable if both are satisfiable or if both are not satisfiable.

The exact guarantees are described in the following theorem:

Theorem 7.1. *For any formula $\varphi \in PL$, there exists a formula $\varphi' \in PL$ such that:*

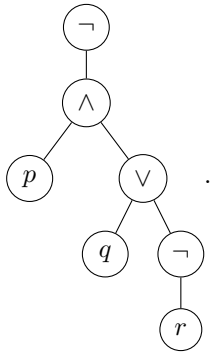
- *φ and φ' are equisatisfiable;*
- *φ' is in CNF;*
- *$size\Big(\varphi'\Big) \leq 16 \times size\Big(\varphi\Big)$ (the resulting formula is not much bigger);*
- *any model of φ' is also a model of φ.*

Recall that $size\Big(\varphi\Big)$ is the number of nodes of the abstract syntax tree of φ and that the function *size* was defined in the second lecture.

As with the previous theorem, we will explain the algorithm for finding φ' from φ on an example.

Step 1 The first step of Tseitin's algorithm is to start with the abstract syntax tree of the formula φ and associate to every node a propositional variable.

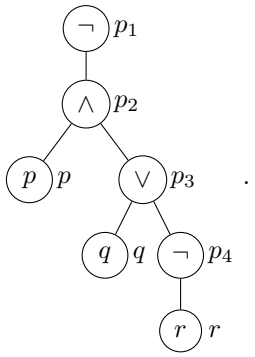
Let $\varphi = \neg(\mathtt{p} \wedge (\mathtt{q} \vee \neg \mathtt{r}))$. We have that

$ast\Big(\varphi\Big) =$ 

.

The first step of the algorithm it to associate to every internal node (node that is not a leaf) a fresh propositional variable (i.e. a propositional variable not seen before). To the leaves, we associate the variables that are already in

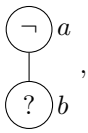
the leaves. Here is the resulting tree, where next to each node we write the propositional variable associated with the node:



The idea of the fresh propositional variables $\mathtt{p}_1, \mathtt{p}_2, \mathtt{p}_3, \mathtt{p}_4, \mathtt{p}_5$ is to construct the formula φ' in such a way so that, in any truth assignment τ satisfying φ', the value of the new propositional variables will be the same as the value of the subformula rooted in the node to which the variable is associated. For example, $\hat{\tau}\Big(p_3\Big)$ should be equal to $\hat{\tau}\Big(q \vee \neg r\Big)$.

Step 2 The formula φ' will be a conjunction of clauses and for each internal node we will have two or three clauses, as follows:

- if the node is a negation of the following form:

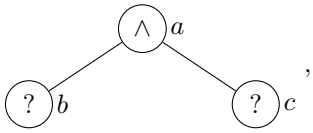


  where $a, b \in A$ are the propositional variables associated to the nodes, then we add to φ' clauses equivalent to the formula $a \leftrightarrow \neg b$. In this way, in any truth assignment satisfying φ', we have that a and $\neg b$ have the same truth value. How do we find the clauses? It is sufficient to rearrange the formula $a \leftrightarrow \neg b$ as follows:

$$\begin{aligned} \mathtt{a} \leftrightarrow \neg \mathtt{b} &\equiv (a \rightarrow \neg b) \wedge (\neg b \rightarrow a) \\ &\equiv (\neg a \vee \neg b) \wedge (\neg \neg b \vee a) \\ &\equiv (\neg a \vee \neg b) \wedge (b \vee a). \end{aligned}$$

  Therefore, we will add the clauses $\neg a \vee \neg b$ and $b \vee a$ to φ'.

- if the node is a conjunction of the following form:

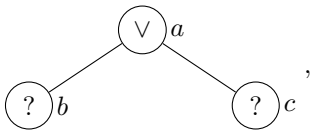


where $a, b, c \in A$ are the propositional variables associated to the nodes, then we add to φ' clauses equivalent to the formula $a \leftrightarrow b \wedge c$. In this way, in any truth assignment satisfying φ', we have that a and $b \wedge c$ have the same truth value. How do we find the clauses? It is sufficient to rearrange the formula $a \leftrightarrow b \wedge c$ as follows:

$$\begin{aligned} a \leftrightarrow b \wedge c &\equiv (a \rightarrow b \wedge c) \wedge (b \wedge c \rightarrow a) \\ &\equiv (\neg a \vee b \wedge c) \wedge (\neg(b \wedge c) \vee a) \\ &\equiv (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee a). \end{aligned}$$

Therefore, we will add the clauses $(\neg a \vee b)$, $(\neg a \vee c)$ and $(\neg b \vee \neg c \vee a)$ to φ'.

- if the node is a disjunction of the following form:



,

where $a, b, c \in A$ are the propositional variables associated to the nodes, then we add to φ' clauses equivalent to the formula $a \leftrightarrow b \vee c$. In this way, in any truth assignment satisfying φ', we have that a and $b \vee c$ have the same truth value. How do we find the clauses? It is sufficient to rearrange the formula $a \leftrightarrow b \vee c$ as follows:

$$\begin{aligned} a \leftrightarrow b \vee c &\equiv (a \rightarrow b \vee c) \wedge (b \vee c \rightarrow a) \\ &\equiv (\neg a \vee b \vee c) \wedge (\neg(b \vee c) \vee a) \\ &\equiv (\neg a \vee b \vee c) \wedge ((\neg b \wedge \neg c) \vee a) \\ &\equiv (\neg a \vee b \vee c) \wedge (\neg b \vee a) \wedge (\neg c \vee a). \end{aligned}$$

Therefore, we will add the clauses $(\neg a \vee b \vee c)$, $(\neg b \vee a)$ and $(\neg c \vee a)$ to φ'.

In our example, we would add the following clauses to φ':

1. $\mathtt{p}_1 \vee \mathtt{p}_2$, $\neg\mathtt{p}_1 \vee \neg\mathtt{p}_2$;
2. $\neg\mathtt{p}_2 \vee \mathtt{p}_1$, $\neg\mathtt{p}_2 \vee \mathtt{p}_3$, $\neg\mathtt{p} \vee \neg\mathtt{p}_3 \vee \mathtt{p}_2$;
3. $\neg\mathtt{q} \vee \mathtt{p}_3$, $\neg\mathtt{p}_4 \vee \mathtt{p}_3$, $\neg\mathtt{p}_3 \vee \mathtt{q} \vee \mathtt{p}_4$;
4. $\mathtt{p}_4 \vee \mathtt{r}$, $\neg\mathtt{p}_4 \vee \neg\mathtt{r}$.

Step 3 Finally, the last step consists of adding a clause to φ' consisting of just the variable associated to the root node.

In our example, we obtain

$$\begin{aligned}\varphi' = \quad & (\mathtt{p}_1 \vee \mathtt{p}_2) \wedge (\neg\mathtt{p}_1 \vee \neg\mathtt{p}_2) \wedge \\ & (\neg\mathtt{p}_2 \vee \mathtt{p}_1) \wedge (\neg\mathtt{p}_2 \vee \mathtt{p}_3) \wedge (\neg\mathtt{p} \vee \neg\mathtt{p}_3 \vee \mathtt{p}_2) \wedge \\ & (\neg\mathtt{q} \vee \mathtt{p}_3) \wedge (\neg\mathtt{p}_4 \vee \mathtt{p}_3) \wedge (\neg\mathtt{p}_3 \vee \mathtt{q} \vee \mathtt{p}_4) \wedge \\ & (\mathtt{p}_4 \vee \mathtt{r}) \wedge (\neg\mathtt{p}_4 \vee \neg\mathtt{r}) \wedge \\ & (p_1)\end{aligned}$$

By construction, we have that:

1. any model of the formula φ' is also a model of φ – meaning that if φ' is satisfiable then φ must also be satisfiable;
2. any model τ of the formula φ can be changed into a model of φ by setting $\tau[\mathtt{p}_i]$ (assuming $\mathtt{p}_i$ are the fresh variables) to the truth value of the corresponding subformula of φ in τ – therefore if φ is satisfiable then φ' must also be satisfiable;
3. for every internal node of φ, φ' contains 2 or 3 clauses, each clause having an ast of at most 16 nodes – this means that the size of the resulting formula φ' is at most 16 times larger than the size of the initial formula;
4. as φ' is a conjunction of clauses, it is obviously in CNF.

Therefore φ' fulfils all requirements stated in the theorem above.

We can use Tseitin's transformation to reduce SAT to CNF-SAT.

Suppose we had a fast algorithm for CNF-SAT. Let us call the algorithm A-CNF-SAT.

We could use A-CNF-SAT and Tseitin's transformation to construct a fast algorithm A-SAT for the SAT problem:

A-SAT(φ)

1. compute φ' from φ using Tseitin's transformation;
2. return A-CNF-SAT(φ').

Intuitively, as φ' is not much bigger than φ, it follows that the call to A-CNF-SAT will also be fast. The algorithm above is called a reduction from SAT to CNF-SAT because it reduces the problem of solving SAT to the problem of solving CNF-SAT.

Most SAT solvers therefore concentrate on solving the CNF-SAT problem.

8 The DIMACS File Format

Most SAT solvers get their input in a format set by the DIMACS.

Here is an example input:

```
c
c start with comments
c
c
p cnf 3 2
-1 -2 0
-1 3 0
```

Lines starting with 'c' are comments. There must be a line starting with 'p' (for problem) that contains the problem description. The string following 'p' is the type of file. We will only use 'cnf', meaning that the file contains a CNF formula. The two numbers following 'cnf' are the number of propositional variables in the formula and respectively the number of clauses.

Let n denote the number of propositional variables and m be the number of clauses. In a DIMACS file, the n propositional variables are denoted by the integers $1, 2, \ldots, n$ (not the more friendly $\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{p}', \mathtt{q}_1, \ldots$ that we use in this text).

To represent the negation of a propositional variable, the '-' sign is used. The m clauses follow and are terminated by the number 0 (conveniently, 0 cannot represent any literal).

Using the friendlier notation $\mathtt{p}_1, \mathtt{p}_2, \mathtt{p}_3, \ldots$ instead of the DIMACS integers $1, 2, 3, \ldots$, the above file encodes the formula

$$(\neg p_1 \vee \neg p_2) \wedge (\neg p_1 \vee p_3).$$

You can install any SAT solver listed, e.g., here:

`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem#Offline_SAT_solvers`

and run it on the file above. Good choices include MiniSAT, CryptoMiniSAT, Glucose. CryptoMiniSAT should be the easiest to install if you use Windows. You can also work with the following online version of MiniSAT (compiled to asm.js using emscripten): `https://jgalenson.github.io/research.js/demos/minisat.html`.

Running a SAT solver on the input file above would produce output similar to the following:

```
SAT -1 -2 -3
```

The output means that the input is satisfiable, and that a satisfying assignment can be obtained by setting all three propositional variables to false (the '-' indicates false).

Running a solver on the following input:

```
p cnf 3 3
-1 -2 0
-1 3 0
1 0
```

could produce the following output:

```
SAT 1 -2 3
```

Finally, running the solver on an unsatisfiable formula such as:

```
p cnf 3 4
-1 -2 0
-1 3 0
1 0
2 0
```

should simply produce something like:

```
UNSAT
```

9 A backtracking approach to (CNF)-SAT

An easy way to solve the (CNF-)SAT problem is to perform backtracking in order to search for a satisfying truth assignment.

The backtracking search would take time $O(2^n)$, where n is the number of propositional variables occuring the formula. You can check out the accompanying file `back.c` that reads from the standard input a CNF formula encoded in DIMACS format and searches using backtracking for a satisfying assignment.

The running time of the algorithm can be checked against files `random_ksat_n.dimacs`, where `n` ranges from `10` to `35`. Each such file contains a CNF formula with `n` variables and $10 \times$ `n` clauses. You can confirm the exponential running time in practice by running the `back.c` program on these inputs and observing how running time approximately doubles whenever `n` is increased by 1 unit.

Modern SAT solver are actually highly optimized and finely tuned backtracking searches, with several additional optimizations such as unit clause propagation, pure literal removal, advanced data structures, good heuristisc for variable orderings, backjumping, random restarts, clause learning, and several others. These optimizations go beyond to scope of the current course, but the source code of most solvers is available for study on the internet (e.g. `https://baldur.iti.kit.edu/sat-competition-2017/solvers/main/` contains the source code of the solvers competing in SAT 2017 in the main track).