

SAT

a different kind of search problem

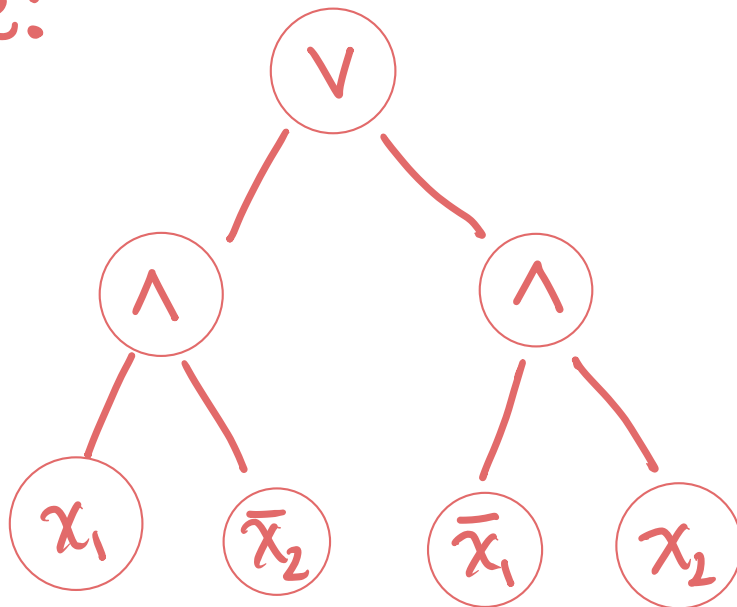
Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$
"x or y" $x \vee y$
"not x" $\bar{x}$ (or $\neg x$)

XOR ("exclusive-or") (TRUE iff $x_1 \neq x_2$)

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:



Adding two bits

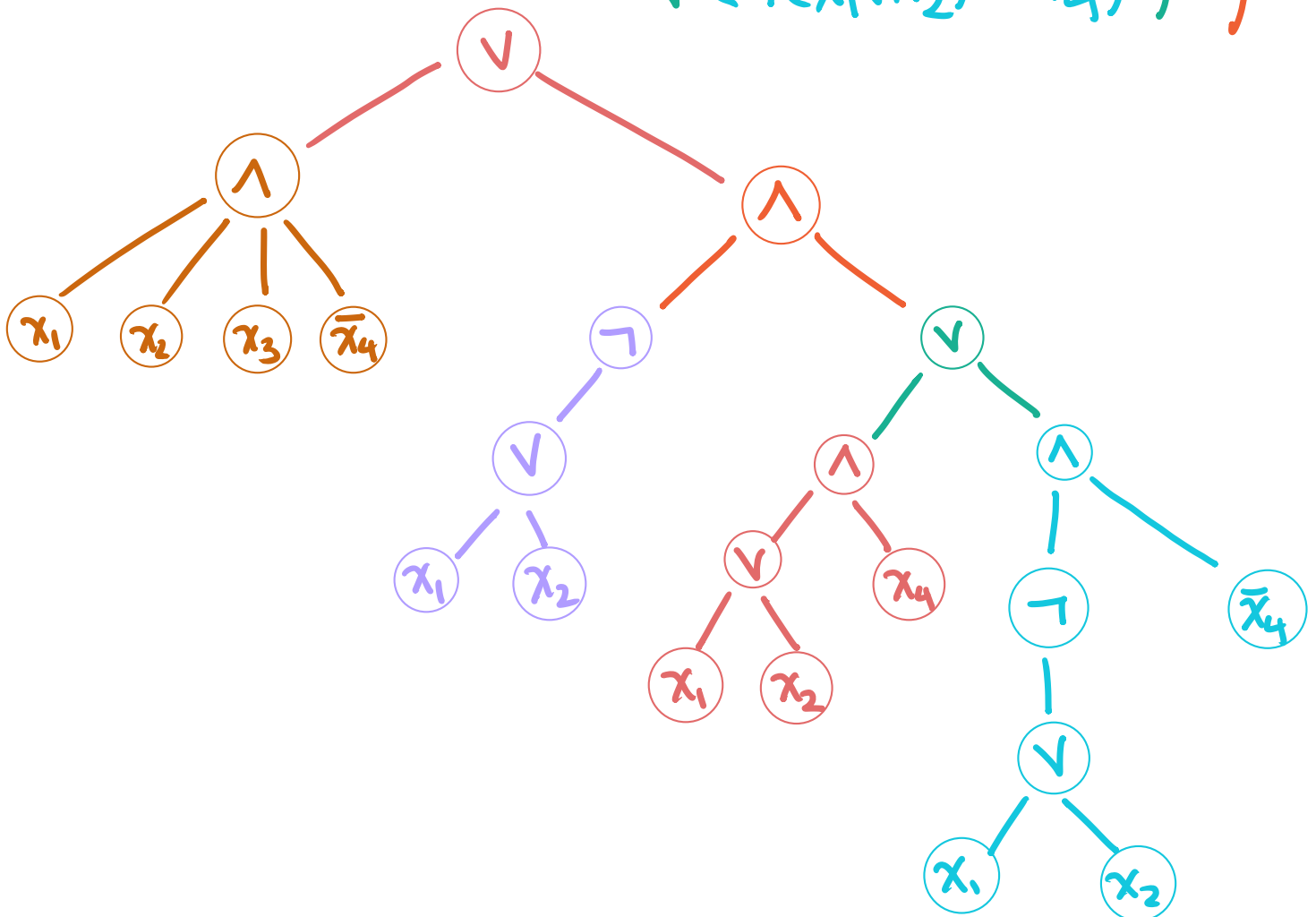
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$

$$f(x_1, x_2, x_3, x_4) =$$

$$(x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4)$$

$$\vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4)$$

$$\vee (\neg(x_1 \vee x_2) \wedge \bar{x}_4))$$



"SAT Programming Language"

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

★ "z = x"

"both true or both false"
 $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$

★ "if x then y
else z"

$(x=1, y=1) \text{ or } (x=0, z=1)$
 $(x \wedge y) \vee (\bar{x} \wedge z)$

So Boolean algebra is very expressive

Boolean Satisfiability

(informally)

Input: Boolean formula $\varphi(x_1, x_2, \dots, x_n)$

- n variables
- "size" m

e.g. $\varphi(\underset{\text{(false)}}{0}, \overset{\text{(true)}}{1}, \underset{\text{(false)}}{0}, \underset{\text{(false)}}{0}, \neg, 1, \underset{\text{(false)}}{0}) = 0$

Goal: decide if φ is "satisfiable";
if there exists $x_1, x_2, \dots, x_n \in \{0, 1\}$ s.t.
 $\varphi(x_1, x_2, \dots, x_n) = 1$

e.g. $\varphi(x_1, x_2) =$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$$

Boolean Satisfiability

(informally)

Input: Boolean formula $\varphi(x_1, x_2, \dots, x_n)$

- n variables
- "size" m

e.g. $\varphi(\overset{\text{(true)}}{0}, \overset{\text{(true)}}{1}, \overset{\text{(false)}}{0}, \overset{\text{(false)}}{0}, \neg, \overset{\text{(false)}}{1}, \overset{\text{(false)}}{0}) = 0$

Goal: decide if φ is "satisfiable";
if there exists $x_1, x_2, \dots, x_n \in \{0, 1\}$ s.t.
 $\varphi(x_1, x_2, \dots, x_n) = 1$

"Search problem": we can recognize the
answer (i.e. sat. assign x) when we see it
(by evaluating $f(x)$)

at the very least, brute force

harder "search problem" than over/under

The Big Question

Is there a polynomial time algorithm
for Boolean satisfiability?

We don't know (!)

Our best answer, loosely speaking:

a polytime algo might be
too good to be true

What about a restricted class of
Boolean formulas?

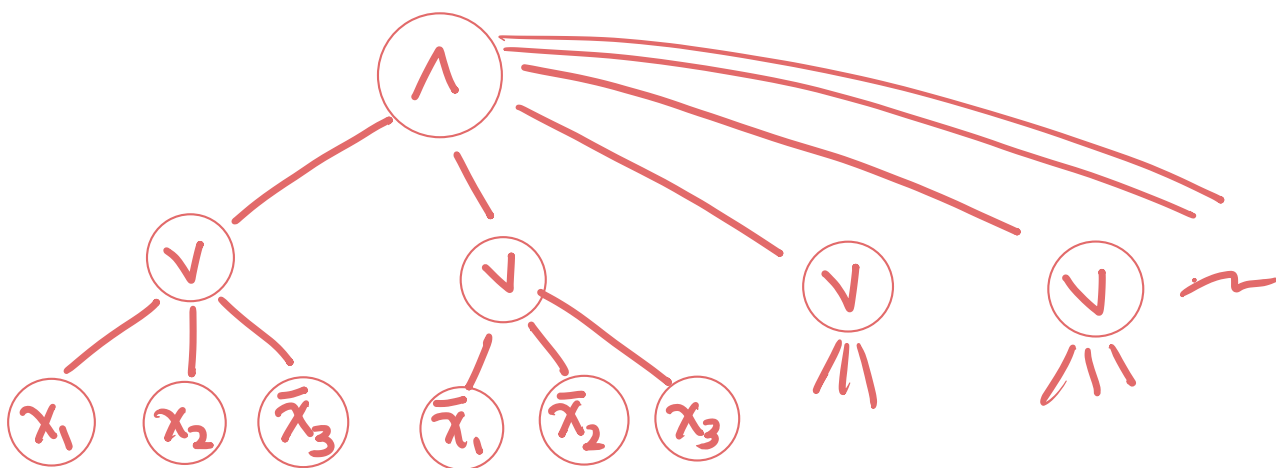
CNF (Conjunctive Normal Form)

"and - or's"

"clause"

$$S(x_1, \dots, x_n) = (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge x_3$$

"tree" for CNF is fairly flat



(CNF)

"SAT problem": given boolean formula in CNF
decide if there is a satisfying sol'n.

Q: polynomial time algorithm for SAT?

(CNF)

"SAT Programming Language"

"x and y"

$$x \wedge y$$

"x or y"

$$x \vee y$$

"not x"

$$\bar{x} \quad (\text{or } \neg x)$$

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$

★ "z = x"

$$= (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x}))$$

$$= (z \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x})$$

$$= (z \vee \bar{x}) \wedge (x \vee \bar{z})$$

(x=1, y=1) or (x=0, y=1)

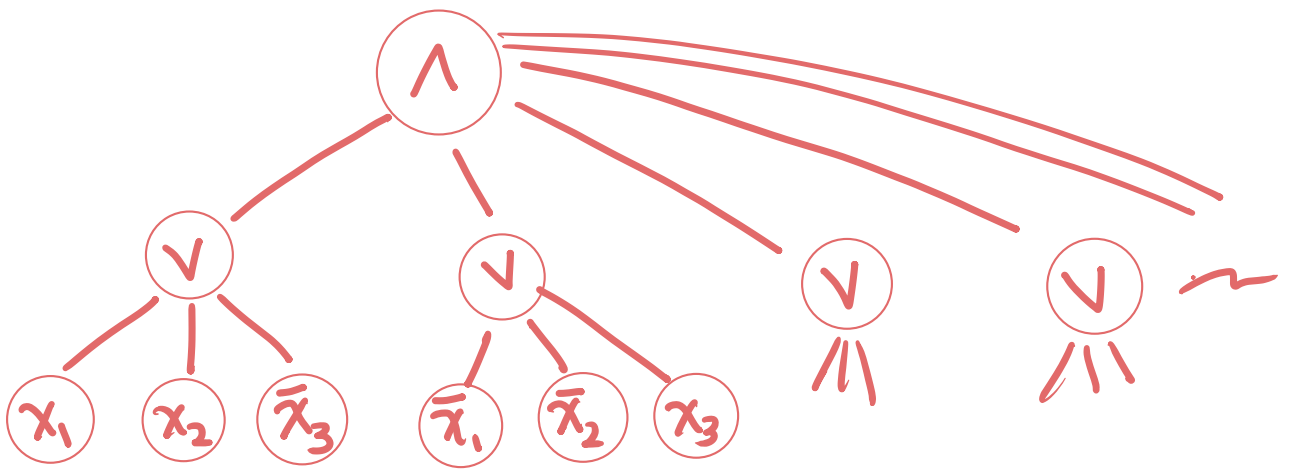
★ "if x then y
else z"

$$(x \wedge y) \vee (\bar{x} \wedge z)$$

$$= (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

$$= (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

So SAT is very expressive



"SAT problem" given boolean formula in CNF,
decide if there is a satisfying sol'n.

Q: polynomial time algorithm for SAT?

We don't know this either.

But we do know something:

Theorem Any boolean formula can be converted to CNF w/ roughly the same size while preserving satisfiability

More precisely:

Given a boolean formula

$\mathcal{F}(x_1, \dots, x_n)$ of size m ,

in polynomial time, one can construct

CNF $g(x_1, \dots, x_p)$ w/ $p = O(mn)$ vars,
size $O(p)$

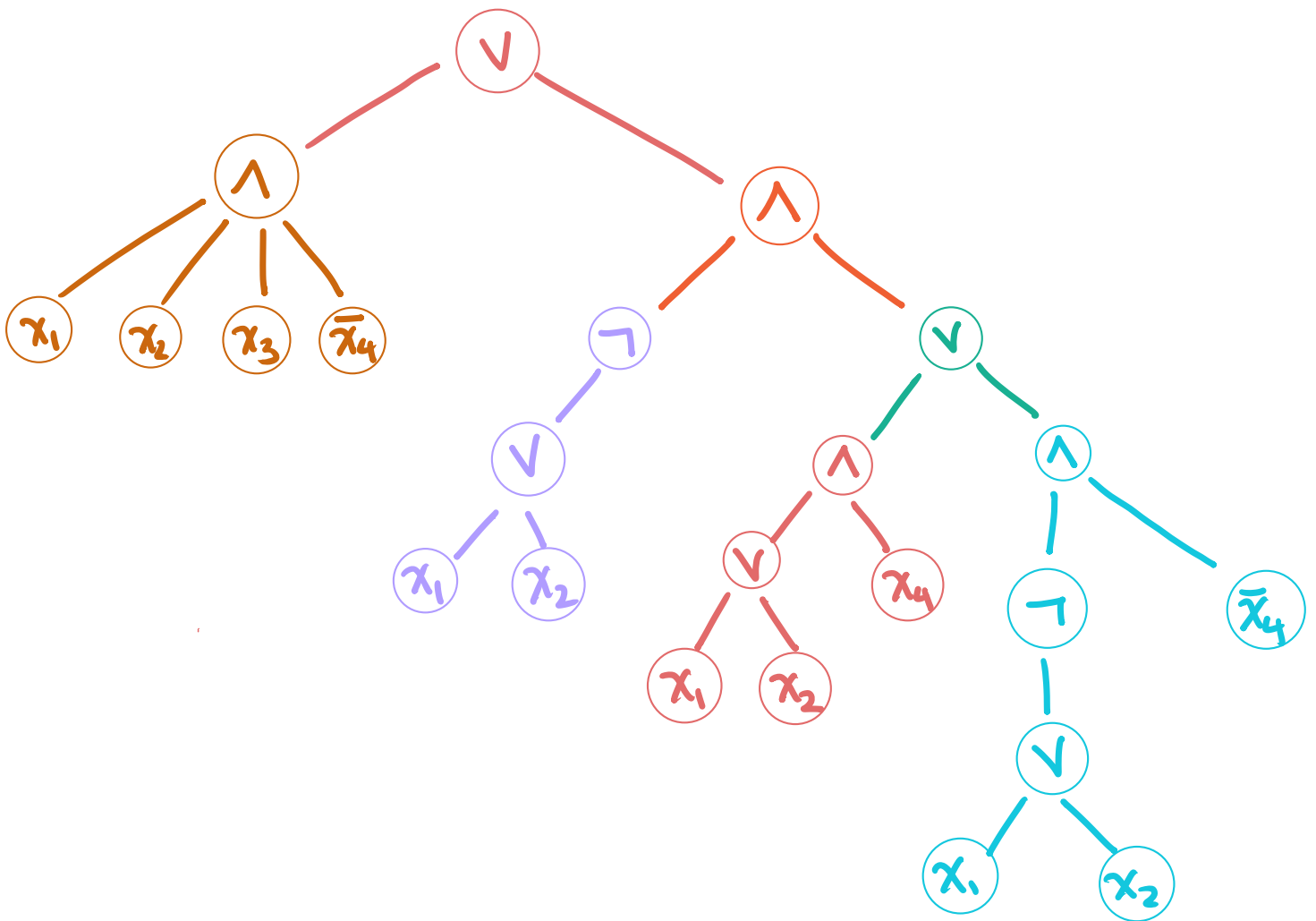
s.t. $\mathcal{F}$ is satisfiable iff g is satisfiable

Corollary: poly. time algo for CNF's

$\Rightarrow$ poly. time algo for all types of boolean formulas

Theorem Any boolean formula can be converted to CNF w/ roughly the same size.

Proof:



$$x_1 = (x_2 \wedge x_3)$$

"SAT Programming Language"

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

$$z = x$$

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$

$$= (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x}))$$

$$= (z \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x})$$

$$= (z \vee \bar{x}) \wedge (x \vee \bar{z})$$

★ "if x then y else z" $(x=1, y=1) \text{ or } (x=0, y=0)$

$$(x \wedge y) \vee (\bar{x} \wedge \bar{z})$$

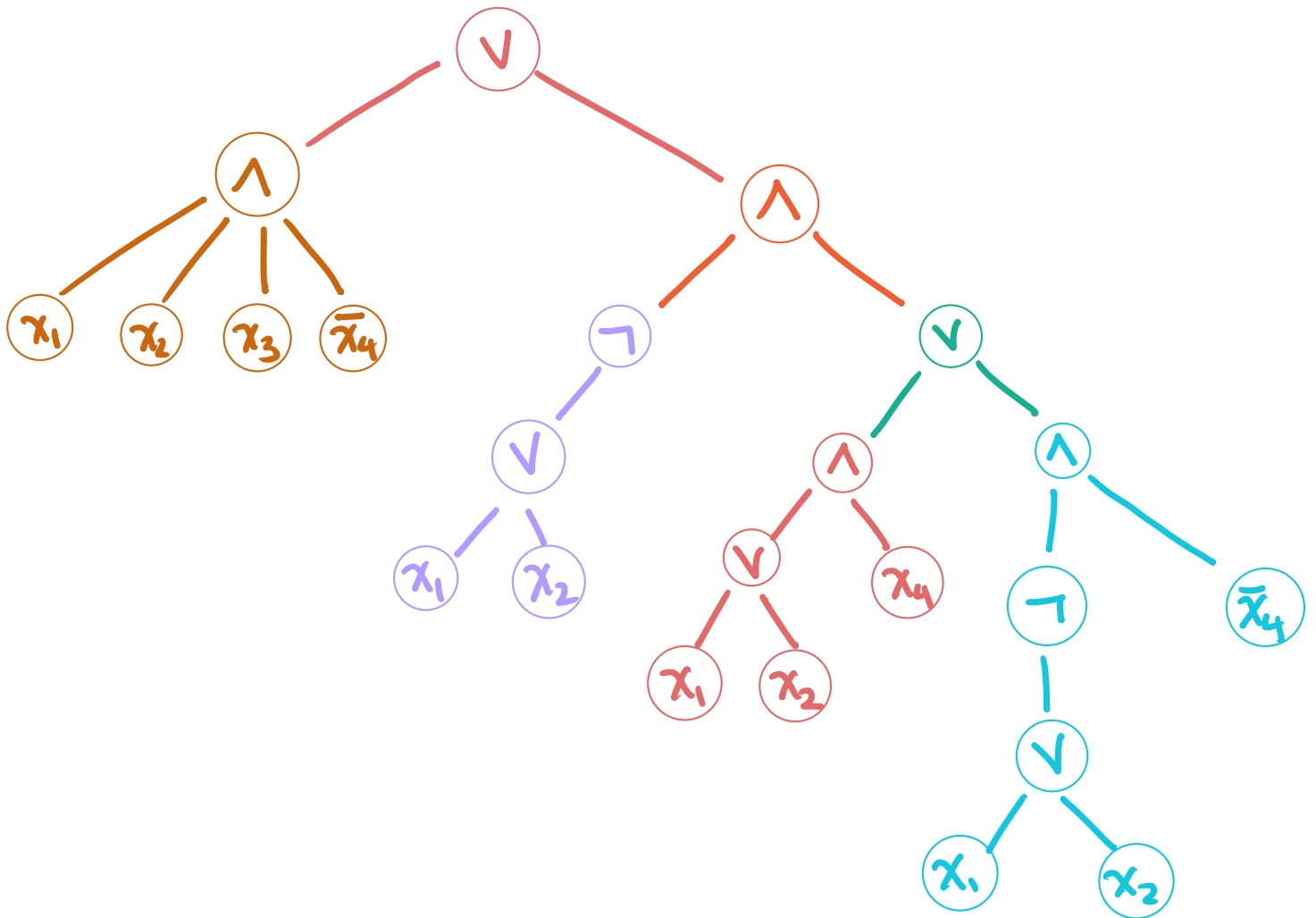
$$= (x \vee \bar{x}) \wedge (y \vee \bar{z}) \wedge (x \vee \bar{x}) \wedge (x \vee \bar{z})$$

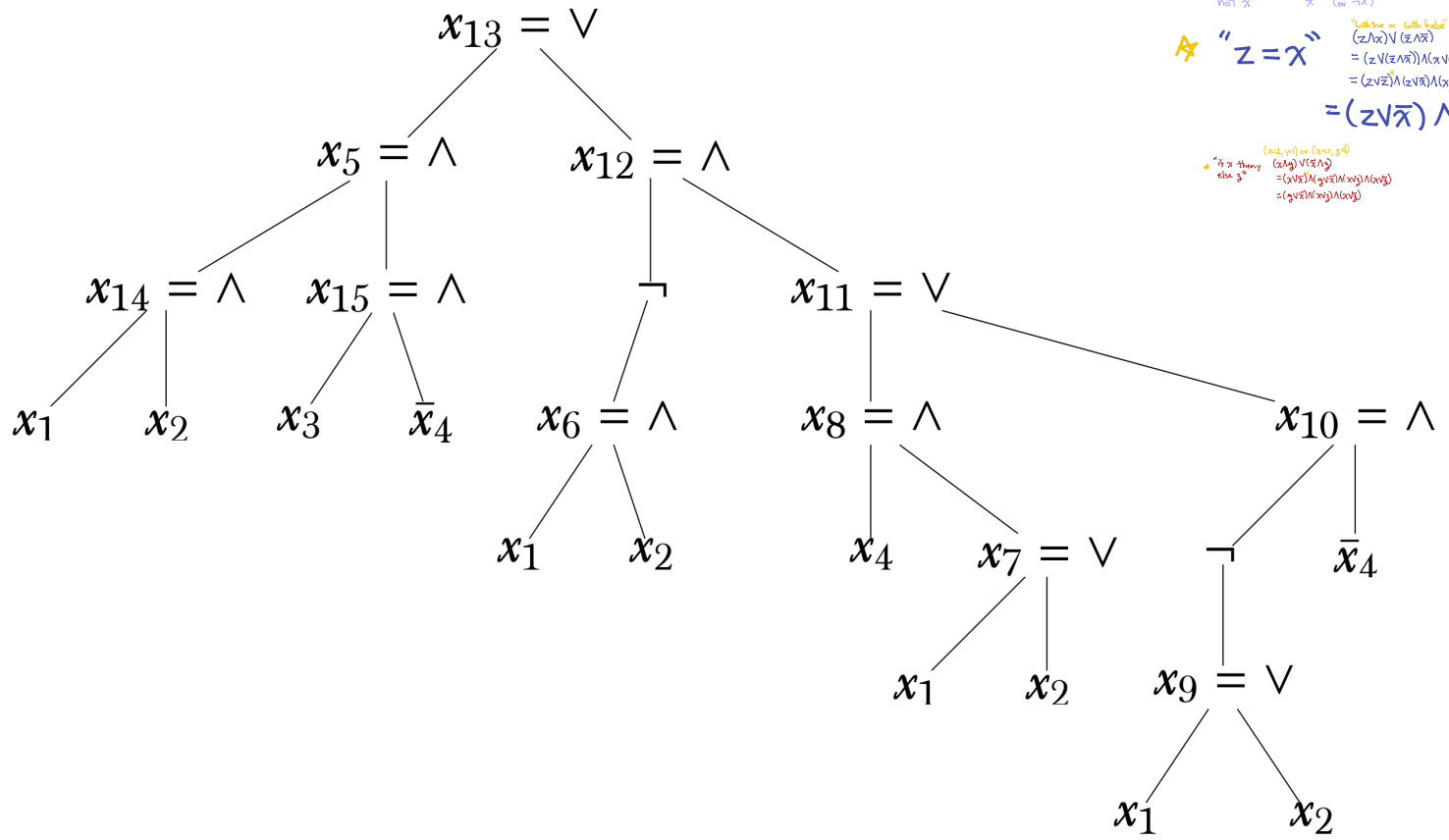
$$= (y \vee \bar{z}) \wedge (x \vee \bar{z})$$

$$x_1 = (x_2 \vee x_3)$$

Example

$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$





$$(\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2)$$

$$(\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4)$$

$$(\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2)$$

$$(x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2)$$

$$(\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7)$$

$$(x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2)$$

$$(\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4)$$

$$(x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10})$$

$$(\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11})$$

$$(x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12})$$

Even more special case: 3SAT

like SAT, except every "or"-clause
has exactly three variables "3CNF"

$$\begin{aligned} S(x_1, m, x_n) = & (x_1 \vee x_2 \vee \bar{x}_3) \\ & \wedge (\bar{x}_1 \vee x_4 \vee x_5) \\ & \wedge (x_2 \vee x_6 \vee x_7) \\ & \wedge m \end{aligned}$$

Is 3SAT easier?

Thm Any CNF can be converted to
3CNF of $\approx$ same size in poly time

similar proof ideas as before

(blank for proof)

Summary so far

Boolean SAT $\equiv$ CNF SAT $\equiv$ 3-SAT

for poly-time algorithms

surprising b/c 3SAT seems much simpler

III. Circuits

Boolean Formulas are to rooted trees
as

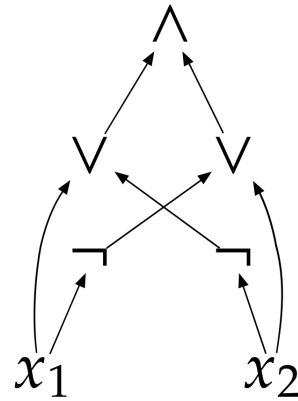
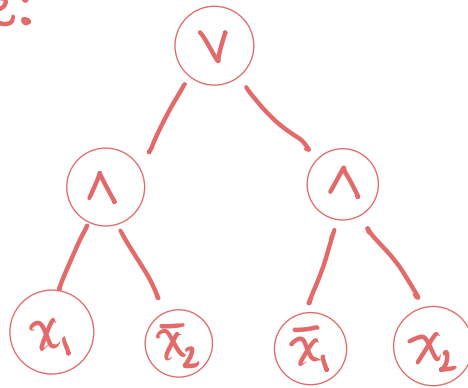
Circuits are to directed acyclic graphs

XOR ("exclusive-or")

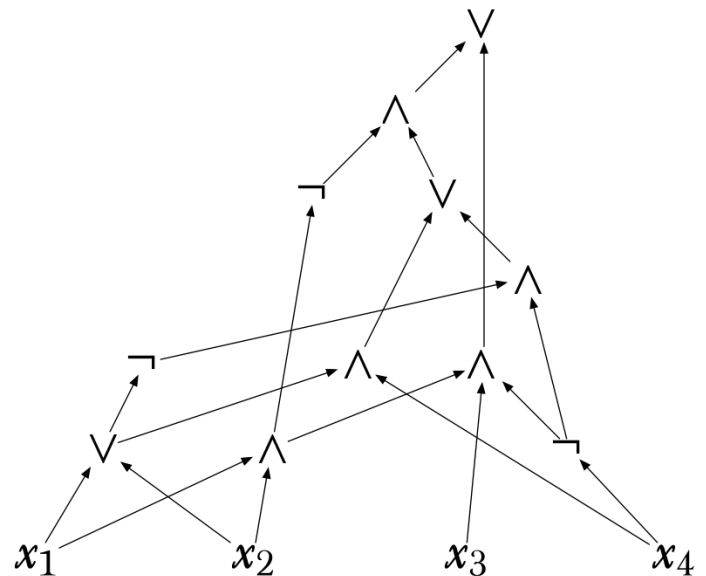
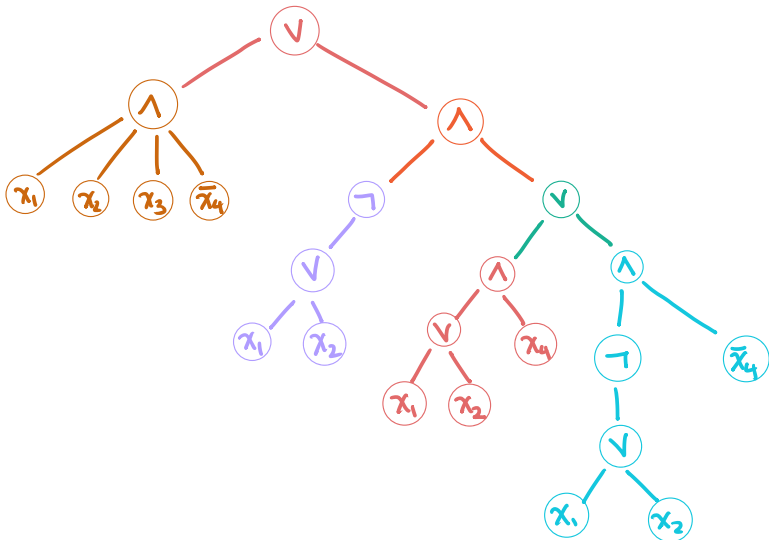
(TRUE iff $x_1 \neq x_2$)

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:

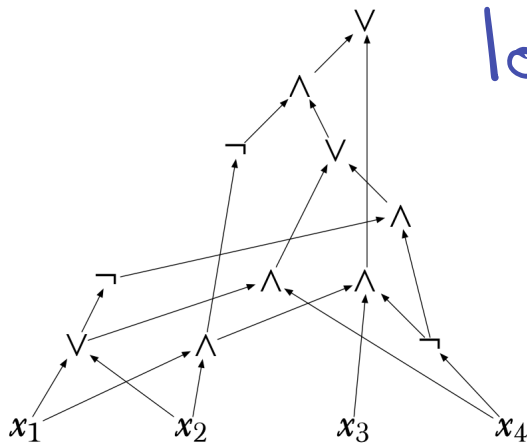


$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$



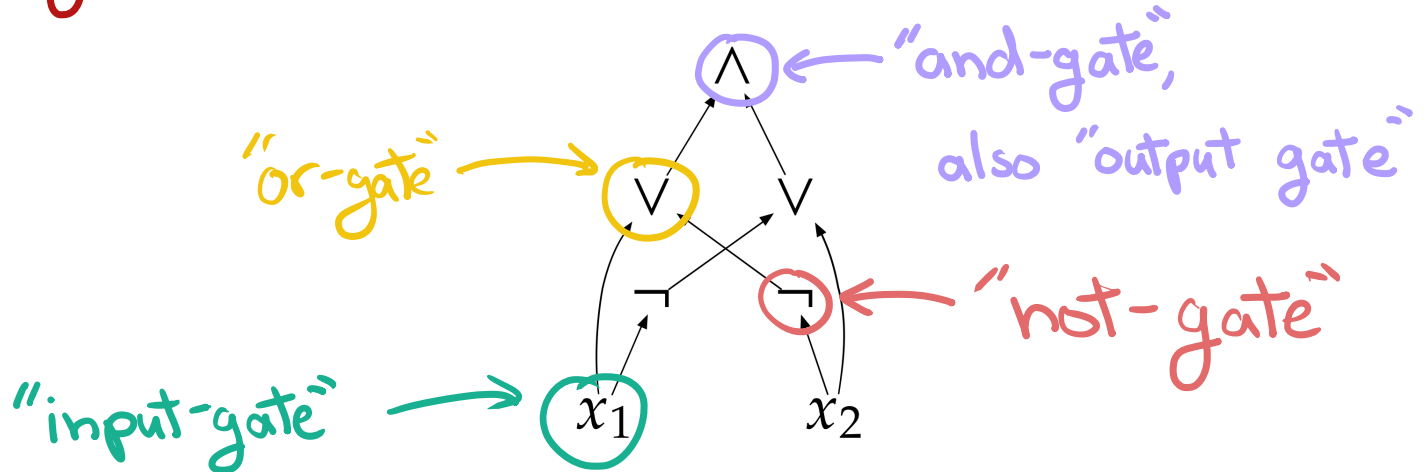
Boolean Formulas are to rooted trees as Circuits are to directed acyclic graphs

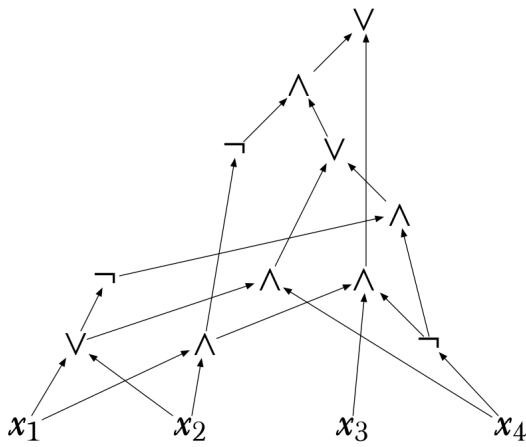
(definition)



logical gates arranged
as vertices in a
directed acyclic graph

"gates": take some inputs bits, output 1 bit





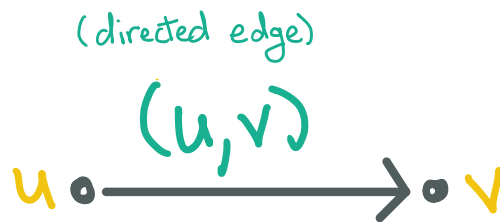
(definition)

logical gates arranged as vertices in a
directed acyclic graph

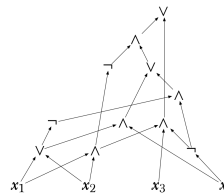
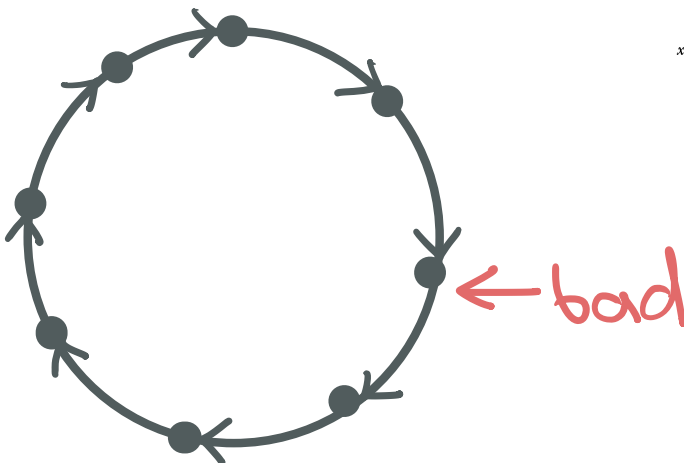
"directed graph" $G = (V, E)$

vertices
(a set of elements)

directed edges
(ordered pairs of vertices)
(aka arcs)



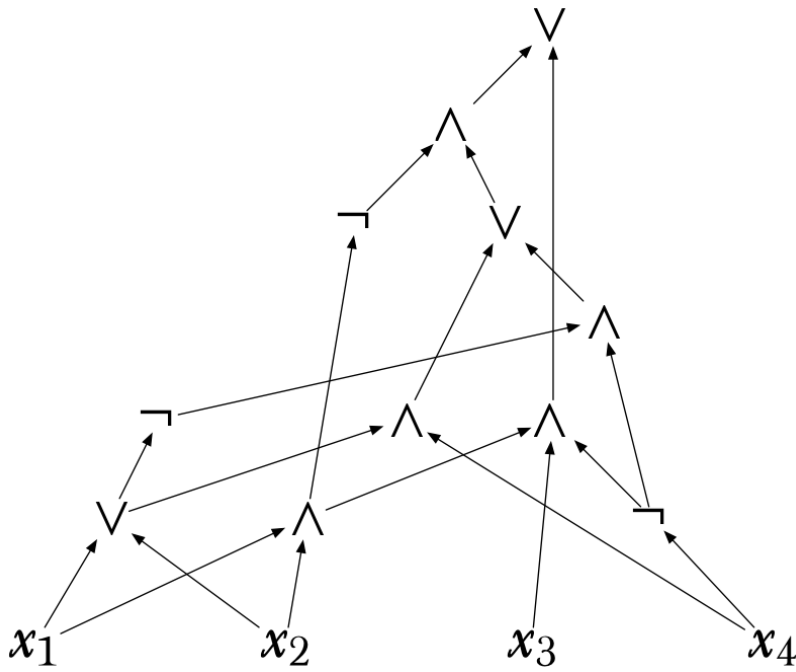
"acyclic": no cycles



(definition)

logical gates arranged as vertices in a
directed acyclic graph

How to evaluate a circuit?

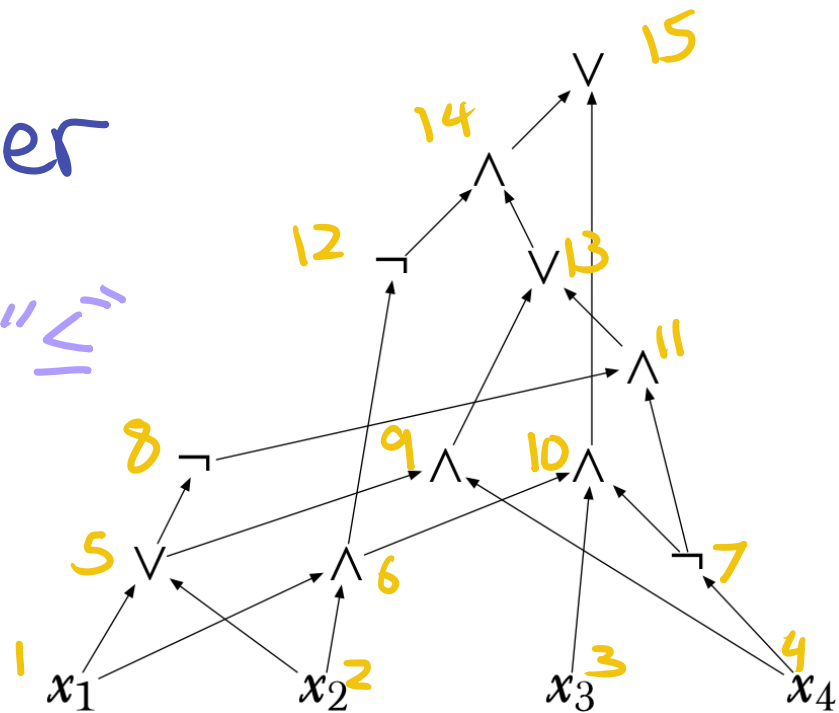
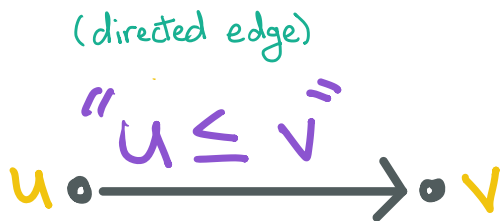


(conceptually)

"bottom up,"
one at a time

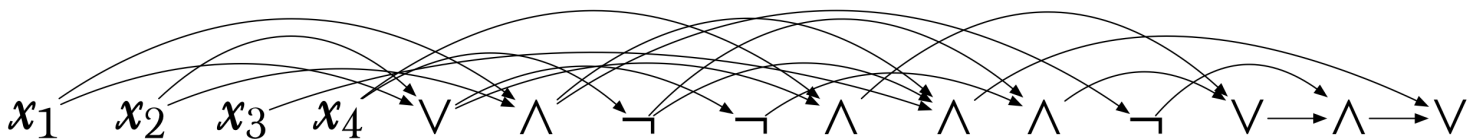
Topological order

think of arcs as " $\leq$ "



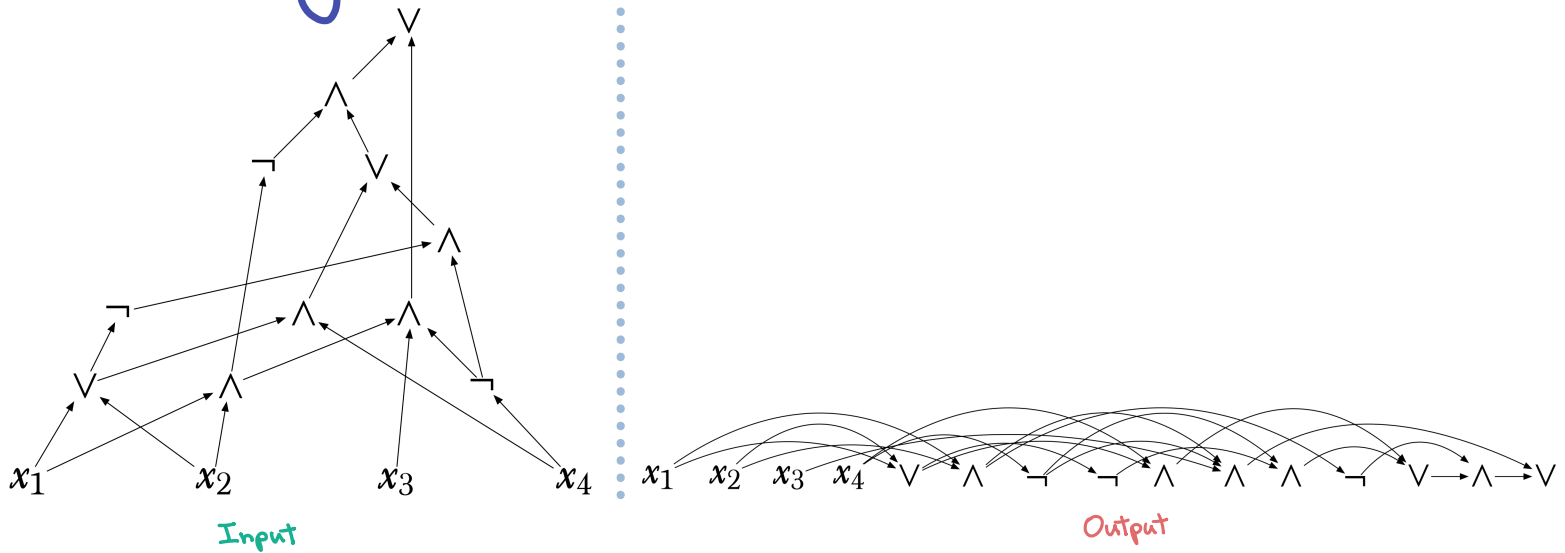
this defines a partial order on V

Topological ordering (of V):
any ordering consistent w/ edges



to evaluate, just process left to right!

"Topological Sort"



Topological sort:

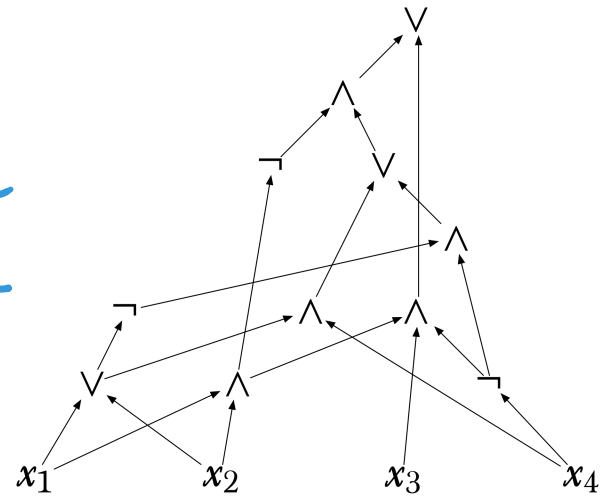
1. While there is a vertex v with in-degree 0:
 - A. Append v to the list.
 - B. For every edge (v, w) leaving v :
 1. Remove (v, w) from the graph (decreasing the in-degree of w by 1).
 - C. Remove v from the graph.

Theorem: topological sort in $O(m+n)$ time

Corollary: evaluate circuit in linear time

Circuit SAT

Given a circuit, is there a satisfying assignment of input bits?



(hence at least as hard as)

(Obv.) more general than $\wedge$ Boolean SAT

Q: are circuits substantially harder than Boolean formulas? than CNF?

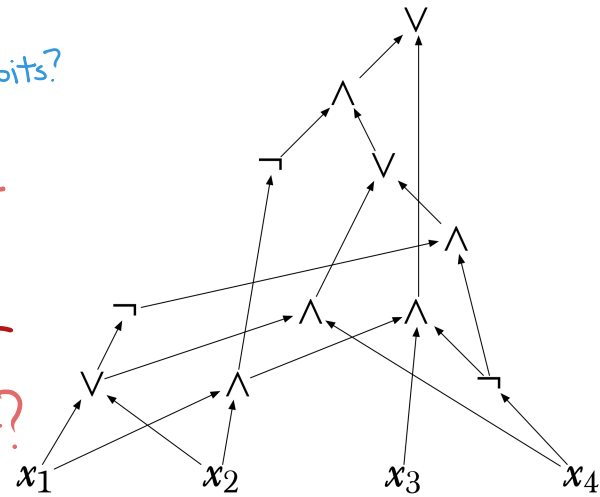
Circuit SAT

Given a circuit, is there a satisfying assignment of input bits?

(hence at least as hard as)

(Obv.) more general than $\wedge$ Boolean SAT

Q: are circuits substantially harder than Boolean formulas? than CNF?



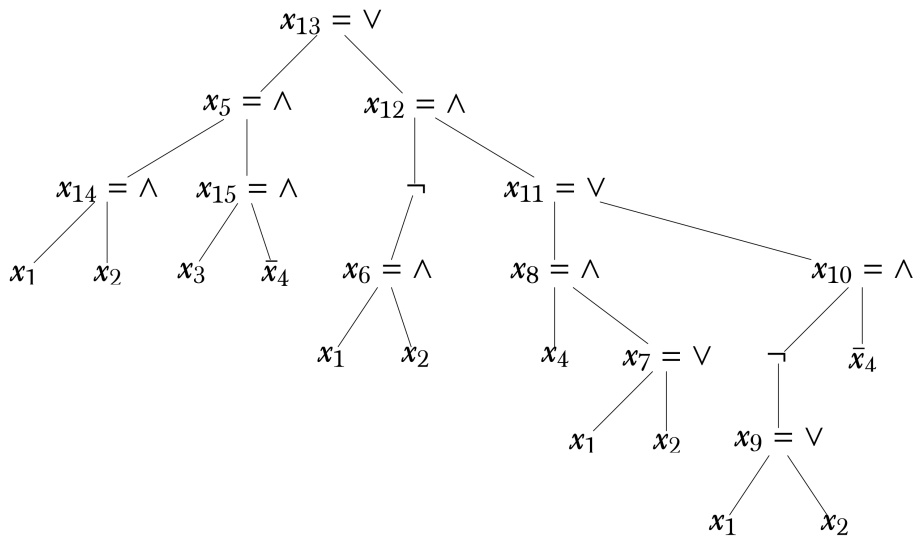
Theorem

(poly-time for Boolean SAT) $\Leftrightarrow$ (poly-time for Circuit SAT)

why?

Same trick as before!

(One equality gadget per gate)



$$\begin{aligned}
 &(\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2) \\
 &(\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4) \\
 &(\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2) \\
 &(x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2) \\
 &(\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7) \\
 &(x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2) \\
 &(\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4) \\
 &(x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10}) \\
 &(\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11}) \\
 &(x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12})
 \end{aligned}$$

Chapter 2

SAT: A different kind of search problem

2.1 Boolean formulas

A Boolean variable x is one where one can take only one of two values, true and false. In computers, the Boolean values are represented by a single bit, with true = 1 and false = 0. We adopt the convention here as well. In Boolean algebra, we have three basic operations.

1. **Negation**, also called a **not**, is a unary operator denoted $\neg x$ for a variable x . $\neg$ maps true to false and false to true. We sometimes abbreviate $\bar{x} \stackrel{\text{def}}{=} \neg x$.
2. A **disjunction**, also called an **or**, is a binary operator denoted $x \vee y$ for two variables x and y . We have $x \vee y = \text{true}$ if either x or y is true. If both x and y are false, then $x \vee y = \text{false}$.
3. A **conjunction**, also called an **and**, is a binary operator denoted $x \wedge y$. We have $x \wedge y = \text{true}$ if and only if both x and y are true; otherwise, $x \wedge y = \text{false}$.

These operations can be combined, using parentheses to notate the order of operations. For example, the formula

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

implements what is commonly called the “exclusive-or” of x_1 and x_2 : $f(x_1, x_2) = 1$ iff $x_1 \neq x_2$.

Observe that for any $x, y, z \in \{0, 1\}$,

$$(x \vee y) \vee z = x \vee (y \vee z);$$

that is, the placement of parentheses does not matter. This is called the **associative property**, and for this reason we may simply denote the above as

$$x \vee y \vee z$$

without introducing any ambiguity. Conjunction $\wedge$ is also associative. However, mixing $\wedge$ and $\vee$ is not associative: in general,

$$(x \vee y) \wedge z \neq x \vee (y \wedge z).$$

(Exercise 2.1 asks the reader for a counterexample.) Instead we have the following rules, called the **distributive property** of $\vee$ and $\wedge$.

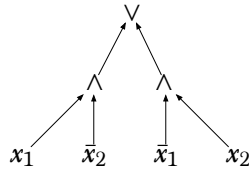
$$\begin{aligned} x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z), \\ (x \vee y) \wedge z &= (x \wedge z) \vee (y \wedge z). \end{aligned}$$

$\neg$ interacts with $\wedge$ and $\vee$ by the following identity, called **De Morgan's laws**.

$$\begin{aligned} \neg(x \wedge y) &= \bar{x} \vee \bar{y}, \\ \neg(x \vee y) &= \bar{x} \wedge \bar{y}. \end{aligned}$$

We leave it to the reader to verify these identities. Lastly we point out that disjunction and conjunction are *commutative*: $x \wedge y = y \wedge x$, and $x \vee y = y \vee x$.

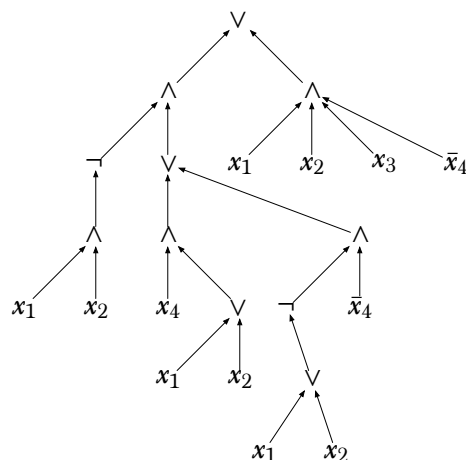
Boolean formulas can have nested parentheses. The part of a formula corresponding to one pair of matching parentheses is called a **clause**. Nested parentheses describe a hierarchy of clauses, which can be visualized in a rooted tree. The following is a tree for exclusive-or.



As a perhaps more interesting example, the formula

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4) \\ &\vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4) \vee (\neg(x_1 \vee x_2) \wedge \bar{x}_4)) \end{aligned}$$

effectively implements addition of two bits: it is satisfied iff $x_1 + x_2 = x_3 x_4$, where $x_3 x_4$ is interpreted as a two-digit binary number. The corresponding tree is as follows.



Both the algebraic and tree-like representations of Boolean formulas emphasize a digital quality of Boolean formulas. Boolean logic is exactly the kind of thing that computers are good at. One might naturally try to program an algorithm that can solve Boolean formulas, in the following sense:

Given a Boolean formula $f(x_1, \dots, x_n)$, find an assignment $x_1, \dots, x_n \in \{0, 1\}$ for which $f(x_1, \dots, x_n) = 1$, if one exists.

One can also consider the *decision* version of the problem, as follows.

Given a Boolean formula $f(x_1, \dots, x_n)$, output true (or 1) if there exists an assignment $x_1, \dots, x_n \in \{0, 1\}$ for which $f(x_1, \dots, x_n) = 1$, and otherwise output false.

Clearly the first problem is at least as hard as the second since being able to find a solution implies you can decide if there exists a solution. But the converse also holds; an algorithm that can decide if a Boolean formula is satisfiable can be used to give an algorithm that also constructs the satisfying assignment (see exercise 2.2). We refer to the above problems as that of **Boolean satisfiability**, abbreviated SAT. It is highly desirable to design a good algorithm for solving SAT.

To put an algorithmic discussion about satisfiability on firm ground, it is worth establishing the following parameters. Each Boolean formula $f(x_1, \dots, x_n)$ has a well-defined number of variables, n . We also define the **size** as the number of symbols (parentheses, variables, or operators) required to write it out. Up to constant factors, this is the same as the total number of times that a variable appears in the formula, counted with repetition (and assuming no trivial redundancies). (Why?) We generally don't worry about constants so these two quantities are essentially equivalent. Note that the size of formula (as defined above) is (basically) the input size from an algorithmic point of view.

A different kind of search problem. A fundamental feature of SAT is that, given a potentially satisfying assignment $x \in \{0,1\}^n$, it is very easy for us to verify that the answer is correct. We simply plug in the values and work through the Boolean algebra to see if the answer comes out as 0 or 1. (Here we would evaluate the innermost parentheses first, and work our way out. Or, in terms of the tree formulation, we work from the leaves up to the roots.) This can be done in linear time in the size of the formula. Now, because we can efficiently verify a solution, we have at least one algorithm at our disposal: brute force. We can enumerate all 2^n possible assignments to $x \in \{0,1\}^n$, and for each x , check if it satisfies the formula. This is not an efficient algorithm, but it is a correct algorithm nonetheless. Remarkably this is more or less the best we can do (in theory, in the worst case, etc.).

Zooming out, Boolean satisfiability is looking for an object that we can *recognize efficiently* when we see it. Any problem of this format is called a **search problem**. The over/under game is also a search problem. Whereas the over/under game is easy, it is challenging to come up with a polynomial time algorithm for Boolean satisfiability – even though both problems are of the same “search” metatype. Why is one so much harder than the other? Perhaps one reason is that Boolean satisfiability is so much more expressive.

For example, suppose we want to force two variables x and y to be equal. Then the following formula is true iff $x = y$, as the reader may verify:

$$(y \wedge x) \vee (\bar{y} \wedge \bar{x}).$$

Appending $\wedge((y \wedge x) \vee (\bar{y} \wedge \bar{x}))$ to an existing Boolean formula is like programming y to be the value of x . As another example, suppose we wanted to model the conditional statement, “if x then y else z ”. This is given by

$$(x \wedge y) \vee (\bar{x} \wedge z).$$

Through these examples, Boolean algebra starts to resemble a programming language.

The flip side of SAT being very expressive is that a polynomial time algorithm would be extremely valuable. We could effectively resolve any question that is logically posed! The consequences – practical, philosophical, and mathematical – would be profound; and alas, maybe too good to be true?

2.2 CNF

If Boolean formulas are too general to expect a good algorithm, then perhaps we can restrict ourselves to a special class of Boolean formulas that might be easier. An important class of SAT formulas are those in **conjunctive normal**

form (CNF). These are the formulas that are a conjunction (i.e, an “and”) of disjunctions (“or’s”), possibly with negations on the variables inside the disjunctions. For example, the following formula is in CNF.

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) = & (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \\ & (x_1 \vee x_4 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_4 \vee x_5) \wedge \\ & (x_2 \vee \bar{x}_3 \vee x_4 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4 \vee x_5) \wedge \\ & (\bar{x}_2 \vee x_3 \vee x_4 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4 \vee x_5). \end{aligned}$$

One example of a satisfying solution is the bit string 01011 satisfying the above. (Can you find another?)

The conjunctive normal form appears to clean up Boolean formulas quite a bit. As a tree, a formula in CNF would be almost flat, with one node representing the overarching conjunction, and a child for each disjunctive clause. But this is only a superficial observation. What we really want to know is whether conjunctive normal forms are meaningfully less expressive than Boolean formulas.

Above we showed how to implement assignment and conditional branching with Boolean formulas. As a test case, let us see if we can derive equivalent formulas in CNF.

1. **Assignment in CNF.** Suppose we wanted to model the constraint “ $z = x$ ” in CNF. We have already observe that this is encoded by $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$, but this is not a CNF. To this end,

$$\begin{aligned} (z \wedge x) \vee (\bar{z} \wedge \bar{x}) & \stackrel{(a)}{=} (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x})) \\ & \stackrel{(b)}{=} (z \vee \bar{z}) \wedge (z \wedge \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x}) \\ & \stackrel{(c)}{=} (z \vee \bar{x}) \wedge (x \vee \bar{z}); \end{aligned}$$

a CNF, as desired. Here (a) and (b) are by the distributive property. (c) drops the tautologies $(x \vee \bar{x})$ and $(z \vee \bar{z})$.

2. **Conditional branching in CNF.** Suppose we want to model the Boolean statement “if x then y else z ”. We have

$$\begin{aligned} (x \wedge y) \vee (\bar{x} \wedge z) & \stackrel{(d)}{=} (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z) \\ & \stackrel{(e)}{=} (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z); \end{aligned}$$

a CNF, as desired. Here (d) is by the distributive property and (e) drops the tautology $(x \vee \bar{x})$.

For at least the above two cases, CNF is just as expressive as general Boolean formulas. But these are just examples and do not constitute a complete proof. A much stronger theorem that truly establishes their (algorithmic) equivalence is as follows.

Theorem 2.1. *Given any Boolean formula $f(x_1, \dots, x_n)$ of n variables and size m , in polynomial time, one can construct a Boolean formula $g(x_1, \dots, x_n, x_{n+1}, \dots, x_p)$ with $p = O(m + n)$ variables and size $O(m + n)$ such that f is satisfiable iff g is satisfiable. Therefore, a polynomial time algorithm deciding if a Boolean formula in CNF is satisfiable implies a polynomial time algorithm for deciding if a (general) Boolean formula is satisfiable.*

The best way to see this is to prove it yourself. See exercise 2.4. (Yes, I believe you can do it.)

For (a wild and crazy) example, in the following formula, we convert the Boolean formula $f(x_1, x_2, x_3, x_4)$ for bitwise addition into a CNF formula g with 15 variables.

$$\begin{aligned} g(x_1, \dots, x_{15}) = & (\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge \\ & (x_{15} \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2) \wedge (x_7 \vee \bar{x}_1) \wedge \\ & (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2) \wedge (\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7) \wedge \\ & (x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2) \wedge (\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge \\ & (x_{10} \vee x_9 \vee x_4) \wedge (x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10}) \wedge (\bar{x}_{12} \vee \bar{x}_6) \wedge \\ & (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11}) \wedge (x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12}). \end{aligned}$$

The formula g above has the following more precise guarantee. For $x_1, x_2, x_3, x_4 \in \{0, 1\}$, we have $f(x_1, x_2, x_3, x_4) = 1$ iff there exists $x_5, \dots, x_{15} \in \{0, 1\}$ such that $g(x_1, \dots, x_{15}) = 1$.

3SAT. An even more special case of CNF is where every clause has exactly k variables, for a fixed integer $k \in \mathbb{N}$; this problem is called k -SAT. Of particular interest is **3SAT**. One might hope for a better algorithm for 3SAT, but again we are thwarted by a polynomial time reduction.

Theorem 2.2. *Given any Boolean formula $f(x_1, \dots, x_n)$ of n variables and size m , in polynomial time, one can construct a Boolean formula $g(x_1, \dots, x_n, x_{n+1}, \dots, x_p)$ with $p = O(m)$ variables and size $O(m + n)$ such that f is satisfiable iff g is satisfiable. In particular, a polynomial time algorithm deciding 3SAT implies a polynomial time algorithm for SAT.*

The proof is left as exercise 2.5.

2.3 Circuits

Boolean circuits are a generalization of Boolean formulas. Recall that a Boolean formula can be expressed as a rooted tree, where each internal node is associated

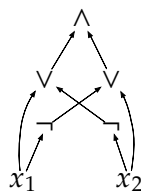


Figure 2.1: A circuit computing the exclusive-or of two bits (at the top gate).

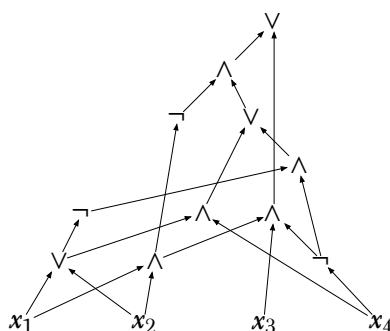


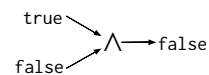
Figure 2.2: A circuit for adding two bits. Here, the gate at the top outputs true iff $x_1 + x_2 = x_3x_4$.

with a logical operation that is applied to the Boolean outputs from its children in the subtree. Circuits are similar except they do not require the logic is more generally arranged in a directed acyclic graph.

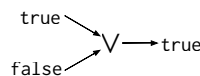
Likely the reader has encountered Boolean circuits before and we briefly review the components. **Figure 2.2** gives an example of a circuit for adding two bits. The picture is almost self-explanatory but let us define circuits formally away. At a high level, Boolean circuits are composed of *logical gates* arranged (as nodes) in a *directed acyclic graph*. We first define the gates and explain their graphical arrangement afterwards.

Each gate corresponds to either one bit of input or one of the three Boolean operators. Each gate takes at most two input bits, and outputs one bit. The basic types of gates are:

1. An input gate, which models a single bit in the input, and outputs that bit.
2. An “and-gate” (or a “conjunction gate”, or simple an “and” or a “conjunction”), takes as input the output bits from two other gates, and outputs the $\wedge$ of these two inputs.



3. An “or-gate” (or a “disjunction gate”, etc.) takes as input the output bits from two other gates, and outputs the $\vee$ of these two inputs.
4. A “not-gate” (or a “negation gate”, etc.) takes as input one output bit from another gate, and outputs the negation of that bit.



Physically, we imagine each circuit logically evaluating its inputs, producing a bit that is propagated to further circuits ahead.

We can model the dependencies between gates by a directed graph, where each gate is a node, and we have a directed edge from one gate x to gate y when x is an input to y . The dependencies are required to be *acyclic*. Here we recall that a *cycle* in a directed graph is a sequence of edges $(x_1, x_2), (x_2, x_3), \dots, (x_k, x_1)$ where the endpoint of one edge is the starting point of the next edge in the sequence, that starts and ends at the same vertex. A cycle would not be well-defined in a circuit: if gate x depends on y , and y depends on z , and z depends on x , then which gate is to be evaluated first? So to avoid these issues we disallow cyclic dependencies by definition¹.

In the simplest model, we assume there is one gate in particular whose output we are externally interested in, as a function of the inputs; we call this the *output gate*. We will focus on a single output gate for simplicity; in general, one can have many output gates. Now we consider the problem of **circuit satisfiability**: given a Boolean circuit, is there an assignment to its input gates that makes the output gate true?

While a Boolean circuit is physically implementable, a first computational task would be able to simulate a Boolean circuit on a given input. A natural approach would be to try to simulate the physical process as follows.

Repeatedly, take any gate that has all of its input bits computed that itself has not been evaluated. Evaluate the inputs according to its logical operator, and write down that output.

The algorithm will need to implement some bookkeeping to keep track of which gates have been evaluated, which of the unevaluated gates are ready to be evaluated, and so forth. It is fairly straightforward to sort out how to do this in linear time.

Let’s step back for a moment. Conceptually, this is really a question about directed acyclic graphs. Given a directed acyclic graph $G = (V, E)$, the goal is to order the vertices $V = \{v_1, \dots, v_n\}$ so that every edge $(v_i, v_j) \in E$ is increasing in index; i.e., $i < j$. That is, we want to list the vertices so that all edges are directed

¹One could consider evaluation *over time*, but this is a separate discussion.

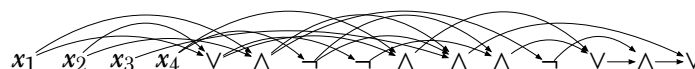


Figure 2.3: The addition circuit (cf. [Figure 2.2](#)) redrawn to list the logical gates in topological order from left to right. Observe that all directed edges go from left to right.

from left to right. Such an ordering is called a *topological order*; the computational task is to *topologically sort* V .

If we topologically sort the gates of a circuit, the input gates will be towards the beginning of the list. In particular, the first gate has to be an input gate. The second gate has to be either (a) another input gate, or (b) a negation of the first gate. In general we might as well put all the input gates at the beginning of the list. Now, given the topological order, we can evaluate the gates in order: when evaluating a gate x , observe that all its inputs must be earlier in the topologically ordered list, hence already evaluated.

It would be convenient, then, to have a generic routine that takes as input a directed acyclic graph and outputs a list of its vertices in topological order. Consider, then, the following algorithm, which iteratively “peels” vertices from the graph. Below, recall that the *in-degree* of a vertex v is the number of edges directed into v .

Topological sort:

1. While there is a vertex v with in-degree 0:
 - A. Append v to the list.
 - B. For every edge (v, w) leaving v :
 1. Remove (v, w) from the graph (decreasing the in-degree of w by 1).
 - C. Remove v from the graph.

Lemma 2.3. *Given a directed acyclic graph $G = (V, E)$ with m edges and n vertices, the peeling algorithm above returns the list of vertices in topological order in $O(m + n)$ time.*

Conceptualizing topological sort, and commoditizing it as a linear time subroutine, gives us the following concise approach to evaluating a circuit.

Topologically sort the gates of the circuit, and evaluate the circuits in topological order.

So much for evaluating a circuit. It is slightly more challenging to evaluate a circuit than a Boolean formula, but topological sort makes it much easier.

Importantly, we have also established that circuit satisfiability is also a search problem: if, indeed, a circuit is satisfiable, then the satisfying inputs provide a proof that can be verified in polynomial time. Clearly circuit SAT is more general than Boolean satisfiability, so circuit SAT must be at least as hard as boolean SAT. But is circuit SAT strictly *harder* than boolean SAT?

We encourage the reader to attempt the question themselves before proceeding.

The reader might have realized, or at least sensed, that any circuit can be converted to a boolean SAT of roughly the same size. We leave it as ?? to explain the construction explicitly. Such a construction implies the following theorem.

Theorem 2.4. *There is a polynomial time algorithm for boolean SAT iff there is a polynomial time algorithm for circuit SAT.*

2.4 Searching for answers

We have danced around but failed to resolve the following question.

Is there a polynomial time algorithm for deciding if a boolean formula is satisfiable?

This is a huge open question, the most important question in theoretical computer science and arguably in all of mathematics. (Extra emphasis on *argue*.)

The general feeling is no, insofar as such an algorithm would be too good to be true. If the answer was yes, then we would have an efficient algorithm for *anything* that can be expressed in boolean logic. But we have already seen that boolean logic is very expressive – maybe too expressive to expect a computer to efficiently process.²

As daunting as the question posed above might seem, we have asked and resolved big questions before. Consider the following question posed by David Hilbert and Wilhelm Ackermann in 1928, called the *Entscheidungsproblem*. Paraphrasing, and omitting mathematical details, the question is roughly:

Can a computer compute (i.e., decide) any logical problem?

For example, a computer can compute whether a boolean formula is satisfiable, by brute-forcing through all the possible solutions. (This is inefficient, but the point is that it works in a finite amount of time.) Is there anything a computer cannot compute, by brute-force or by other means? Remarkably this question has been answered. The answer is *no*, there are indeed *incomputable problems*,

²This description oversimplifies the debate; there are other, more nuanced opinions and subtle related results.

and it was proven independently by Church [Chu36] and Turing [Tur37]. We stress that the *Entscheidungsproblem* was posed *before* computers existed, purely in terms of mathematical logic. Church invented λ -calculus to lay the foundations for his proof; λ -calculus is the inspiration for lisp and functional programming languages since. A major contribution in Turing's approach was to define an abstract model of a computer (called a Turing machine), and then restate the *Entscheidungsproblem* in terms of this model. Turing's initial model for a computer (inspired by a typewriter) had a huge influence on modern computer design [Neug3] and is still the standard model for computation.

2.5 Additional results

While we do not have a polynomial time algorithm for SAT there are still many interesting developments that shed some insight into the character of the problem. We do not have the technical background to analyze the following topics at the moment but even at a high-level they are interesting to consider.

2.5.1 Approximating 3SAT by randomized rounding

While 3SAT might be hard to solve exactly, perhaps it is not so hard to *approximately* solve it. Consider 3SAT, where we are given a formula in CNF with exactly three variables per clauses. Instead of trying to satisfy all the clauses, let us consider the optimization perspective where the goal is to satisfy as many clauses as possible. Of course being able to obtain the true maximum will also let you solve 3SAT. But maybe we can argue that we can obtain *relatively good* solutions. For example, is there a way to satisfy at least half as many clauses as the optimum?

Let $f(x_1, \dots, x_n)$ be a 3CNF, for which we want to satisfy as many clauses as possible in polynomial time. Imagine assigning every variable x_i *randomly* to true or false, by independently flipping a coin for every x_i . How good is this random (and completely ridiculous) assignment?

Consider a single disjunction clause; for example, consider $(x_1, \vee \bar{x}_2 \vee x_3)$. If $x_1 = \text{true}$, or $x_2 = \text{false}$, or $x_3 = \text{true}$, then the clause is satisfied. In fact the only way the clause is *not* satisfied is if $x_1 = \text{false}$ *and* $x_2 = \text{true}$ and $x_3 = \text{false}$. That exact combination occurs with probability $1/8$, since each assignment is independent and uniformly random. So 7 out of 8 times the clause is satisfied!

How many clauses do we expect to satisfy? Think of the number of satisfied clauses as a *random sum*, to which each clause contributes 1 with probability $7/8$,

and contributes 0 otherwise. Now this random sum might seem tough to analyze because whether or not one clause is satisfied is *not* independent of whether another clause is satisfied, since they may share variables. Here we apply *linear of expectation: the average of the sum equals the sum of averages of the individual terms*. (We will introduce linear of expectation more slowly when discussing randomized algorithms in chapter 22.) But the point is that each clause contributes $7/8$ th “satisfied clauses”, on average; consequently, if there are m total clauses, we expect to satisfy $7m/8$ clauses. This implies, by the probabilistic method, that *every 3SAT formula (automatically) has an assignment satisfying $7/8$ -fraction of its clauses*.

It is easy to extend the above to k -SAT for any $k \in \mathbb{N}$, where each clause has exactly k variables, and obtain the following.

Theorem 2.5. *For all $k \in \mathbb{N}$, there is a randomized $(1 - 1/2^k)$ -approximation algorithm for k -SAT.*

2.5.2 Hardness of approximation for 3SAT, and the PCP theorem

The above algorithm is so simple, and essentially oblivious to the input f , that it is hard to believe it is a very good algorithm. Remarkably, there is reason to believe that it is the best possible polynomial time algorithm unless $P = NP$. The **PCP theorem** states that for all constants $\epsilon > 0$, getting better than a $(7/8 + \epsilon)$ -approximation to 3SAT is NP-Hard. The PCP theorem gives similar *hardness of approximation* results for many other problems besides SAT, and in general, has wide and deep consequences across theoretical computer science. The PCP theorem is far beyond the scope of this class although we did cover the proof in the Fall 2020 randomized algorithms course [Qua20b].

(More additional results to come.)

2.6 Additional notes and references

SAT is discussed in [KT06, §8.2], [Eri19, §12.1–12.6], and [DPVo8, §8.1], in a broader context (the theory of NP-completeness) that will emerge in upcoming chapters. The systematic study of understanding the limits of computation – of which SAT and circuits are central topics – is called *computational complexity theory*; see [ABog; Sip97; Wig19].

2.7 Exercises

Exercise 2.1. Show that $\wedge$ and $\vee$ are not jointly associative by finding assignments $x, y, z \in \{0, 1\}$ such that $(x \vee y) \wedge z \neq x \vee (y \wedge z)$.

Exercise 2.2. Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting true or false depending on whether there *exists* a satisfying assignment.). Show how to use this algorithm to obtain a polynomial time algorithm for the constructive version of Boolean satisfiability, outputting a satisfying assignment if one exists.

Exercise 2.3. Show that any Boolean formula with n variables and total size m can be converted to a Boolean formula with n variables and total size $O(m)$ such that every clause contains at most two sub-clauses or variables. That is, in the tree representation of the formula, each $\wedge$ and $\vee$ would have two children. For example, the formula for XOR, $f(x_1, x_2) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$, is in this form.

Exercise 2.4. Show that any boolean formula with n variables and total size m can be converted to a boolean formula in CNF with $O(m + n)$ variables, $O(m)$ clauses, and total size $O(m)$.^{3,4}

Exercise 2.5. Show that any CNF with n variables and total size m can be converted into a 3-SAT problem with $O(m)$ clauses and $O(m + n)$ variables.

Exercise 2.6. Let $k \geq 3$. Recall that the k -SAT problem is the special case of SAT where $f(x_1, \dots, x_n)$ is a CNF with exactly k -variables per clause.

1. Show that a polynomial time algorithm for k -SAT implies a polynomial time algorithm for $(k + 1)$ -SAT.
2. Show that a polynomial time algorithm for $(k + 1)$ -SAT implies a polynomial time algorithm for k -sat.

Exercise 2.7. Consider the following special case of SAT, which we will call **2-occurrence-SAT**. The input consists of a SAT formula $f(x_1, \dots, x_n)$ in CNF such that every variable x_i appears (as is, or negated) in at most 2 clauses. The problem is to decide where there is a satisfying assignment. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm for 2-occurrence-SAT implies a polynomial time algorithm for SAT.

Exercise 2.8. Consider the following variation of SAT, which we will call **opposite-SAT**. The input consists of a collection of m disjunction clauses over n variables $x_1, \dots, x_n$. The goal is to decide if there is an assignment such that every disjunction evaluates to false. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm implies a polynomial time algorithm for SAT.

³It might simplify things to first apply the preceding exercise.

⁴As a second (appropriately vague) hint, how do you express " $a = b \vee c$ " and " $a = b \wedge c$ " in CNF?

Exercise 2.9. Recall that a **palindrome** is a string spelled the same forward or backwards, such as “racecar”. Likewise we say that a bit string $x \in \{0,1\}^n$ if $x_i = x_{n+1-i}$ for all indices $i = 1, \dots, n$. Describe a SAT formula in CNF of size $O(n)$ for a Boolean formula $f : \{0,1\}^n \rightarrow \{0,1\}$ that accepts palindromes (i.e., $f(x) = 1$ iff x is a palindrome).

Exercise 2.10. In the **Max 2-SAT** problem, one is given a CNF $f(x_1, \dots, x_n)$ with *exactly* two variables per clause (i.e., a 2-CNF). The goal is to find an assignment that satisfies *as many* clauses as possible. (Note that we are not interested in knowing if f is exactly satisfiable.) In the decision version of the problem, we are given a 2-CNF formula f and an integer $k \in \mathbb{N}$, and the goal is to decide if there exists an assignment that satisfies at least k clauses in f .

Either (a) show that a polynomial time algorithm for Max 2-SAT implies a polynomial time algorithm for boolean satisfiability (in general), or (b) design and analyze a polynomial time algorithm that, given a 2-CNF formula f , computes an assignment that maximizes the number of satisfied clauses.

Exercise 2.11. In the **1-in-3-SAT** problem, one is given a 3-CNF $f(x_1, \dots, x_n)$. The goal is to find a satisfying assignment such that every disjunctive clause (with three variables) is satisfied by *exactly* one of the variables.

Either (a) show that a polynomial time algorithm for 1-in-3-SAT implies a polynomial time algorithm for SAT, or (b) design and analyze a polynomial time algorithm for 1-in-3-SAT.

Exercise 2.12. A Boolean formula is in **disjunctive normal form (DNF)** if it is a disjunction of conjunctions. For example, for formula

$$f(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z),$$

which models “if x then y else z ”, is in disjunctive normal form. Note that by DeMorgan’s law,

$$(a \vee b) \wedge (c \vee d) = (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d),$$

so any CNF formula can be converted to a DNF formula.

Either (a) show that a polynomial time algorithm for DNF satisfiability implies a polynomial time algorithm for boolean satisfiability (in general), or (b) design and analyze a polynomial time algorithm for deciding if a DNF formula is satisfiable.

Exercise 2.13. Let $f : \{0,1\}^n \rightarrow \{0,1\}$ be a Boolean function on n variables. (Here no formula for f is given, and we treat f as a black box.) Prove that there is a CNF $\varphi(x_1, \dots, x_n)$ of finite size such that $f = \varphi$.

2.A Proof of theorem 2.1

We left the proof of theorem 2.1 as exercise 2.4 (which you should really try, if you haven't). However the result is so fundamental that we should provide a proof (or at least a proof sketch) anyway. The following also suggests some ideas that may help you prove theorem 2.2 (which was left as exercise 2.5).

Here we describe a construction based on the “assignment” CNF gadget in the lecture notes. The idea is to prune all the clauses/subtrees by introducing new variables assigned to be equal to each clause. We hope the following construction is conceptually clear.

Let $f(x_1, \dots, x_n)$ be a Boolean formula with n variables and total size m . Without loss of generality (by the preceding exercise) we may assume that f is “binary”, in the sense that every clause has at most 2 sub-clauses or variables.

For each clause C_i , we introduce a new variable y_i . The idea is to replace each clause C_i by assigning its output to y_i .

To this end, observe that the expression “ $x_1 = x_2 \wedge x_3$ ” can be expressed as the CNF

$$(\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3), \quad (2.1)$$

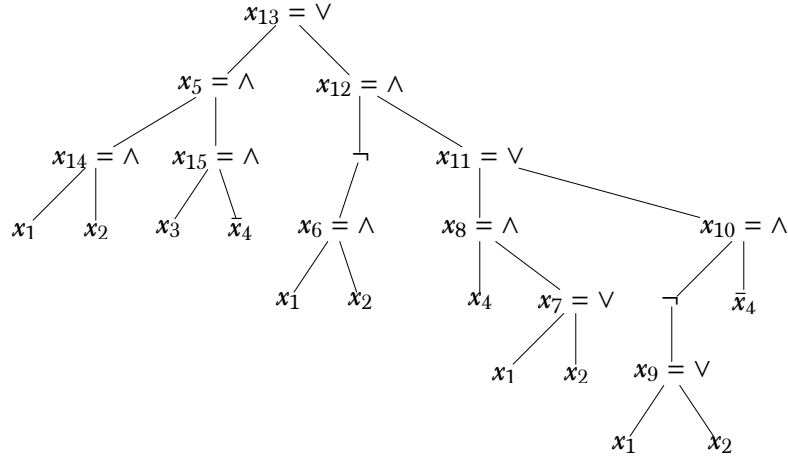
and the expression “ $x_1 = x_2 \vee x_3$ ” can be expressed as the CNF

$$(x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3). \quad (2.2)$$

We process all the clauses in the tree one by one starting from the bottom (and including the clause at the root). For each clause C_i , we replace C_i in the tree with a variable y_i . Because C_i is at the bottom of the tree, it is the conjunction or disjunction of at most two literals (involving either the x_i 's or previously processed y_j 's). We write the CNF corresponding to the assignment $y_i = C_i$ based on (2.1) or (2.2) or above. Note that this CNF has constant size.

All the CNF's we output are concatenated in an overarching conjunction. We produce a constant size CNF per clause in f , so the overall size is the same up to constant factors. We introduce a variable for each clause which gives $O(m + n)$ variables in all.

Below, we apply the construction to the bit-addition formula from the lecture notes. We have annotated the nodes in the tree with the variables introduced and assigned to them. (Here, rather than use y_i 's, we added x_j 's with larger indices j .) Below the tree, we list the CNF's that are produced, with one line per clause in the original formula.



$$(\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2)$$

$$(\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4)$$

$$(\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2)$$

$$(x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2)$$

$$(\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7)$$

$$(x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2)$$

$$(\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4)$$

$$(x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10})$$

$$(\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11})$$

$$(x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12})$$