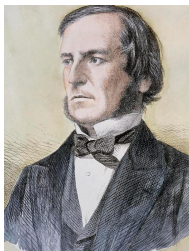


2SAT and Max 2-SAT

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 $\neg x$)



George Boole,
1815-1864

XOR ("exclusive-or")

(TRUE iff $x_1 \neq x_2$)

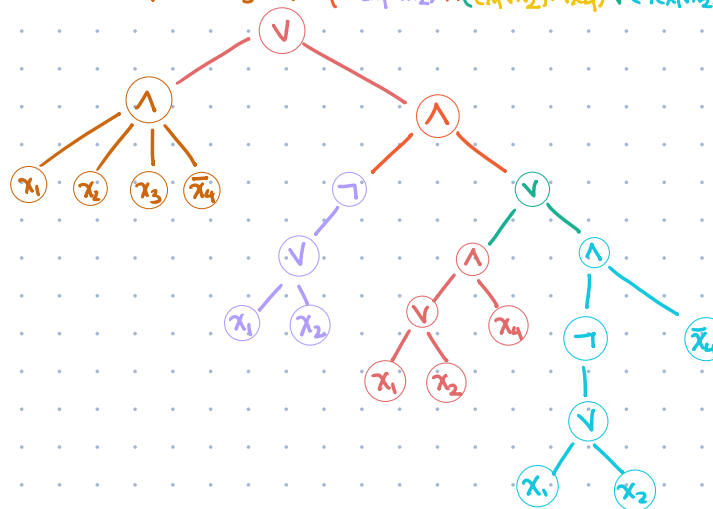
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

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) or (x=0, z=1)"

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

So Boolean algebra is very expressive

Boolean Satisfiability

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

• n variables • "size" m

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

Goal: decide if φ is "satisfiable";

i.e., 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 an answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x .
- harder "search problem" than "over/under"

Boolean Satisfiability

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

- n variables
- "size" m

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

Go The Big Question

Is there a polynomial time algorithm for Boolean satisfiability?

"See"

answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x .
- harder "search problem" than "over/under"

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

"x and y" $x \wedge y$

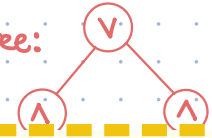
"x or y" $x \vee y$

"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:



x_2

$$x_2 = x_3 x_4$$

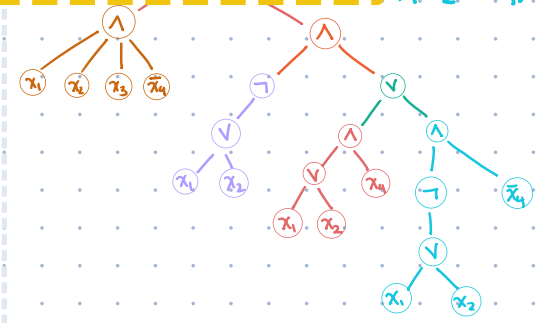
wise

$$x_3 \wedge \bar{x}_4$$

)

$$x_2) \wedge x_4$$

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



"SAT Programming Language"

★ "z = x"

★ "if x then y else z"

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

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

"and-of-or's"



$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (m \vee n) \wedge m$$

Given boolean formula f in CNF, decide if f is satisfiable

Is there a polynomial time algorithm for CNF-satisfiability?

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

"x and y"

$x \wedge y$

"x or y"

xy

"not x"

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

A. "z=x"

• "both true or both false"

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

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

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

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

- ★ "if x then y else z"

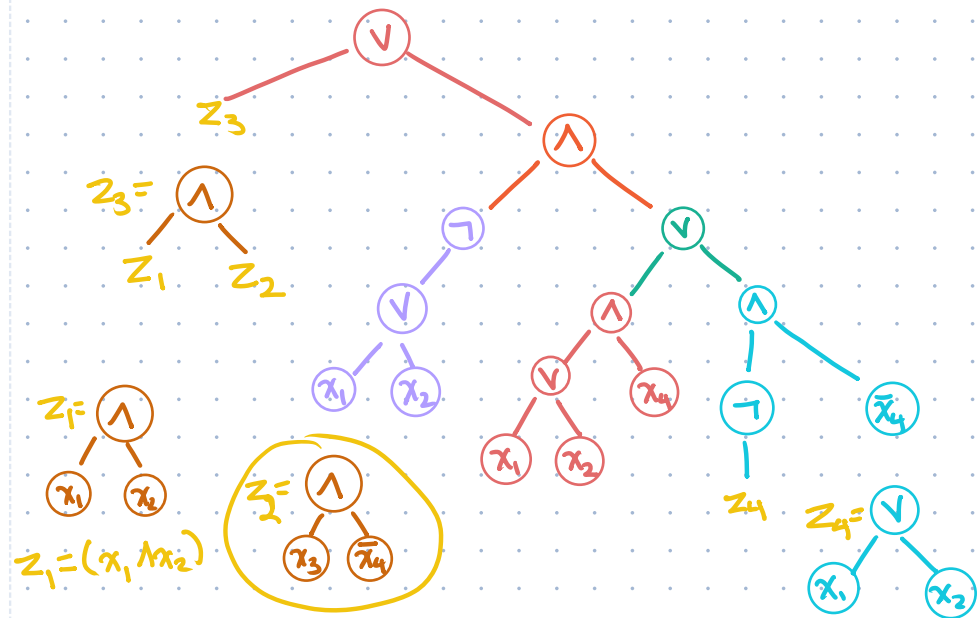
- $"(x=1, y=1) \text{ or } (x=0, y=1)"$

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

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

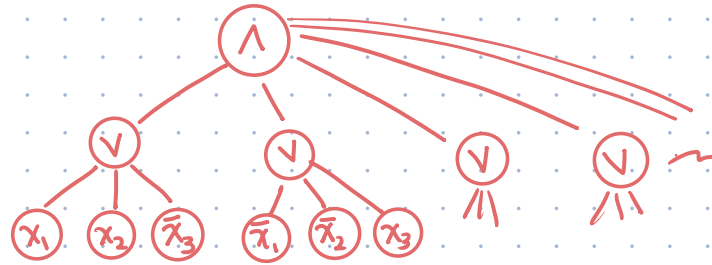
$$= (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

• So CNF is also very expressive.



more special case: **3-SAT**

like CNF-SAT, except w/ exactly 3 variables per clause.



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\sim \vee \sim \vee \sim) \wedge \sim$$

"clause"

Is there a polytime algo for 3SAT?

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"

$$f(x_1, \dots, x_n) =$$



$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\sim \vee \sim \vee \sim) \wedge \sim$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

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

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

★ "if x then y else z"

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

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

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

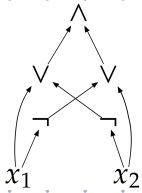
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

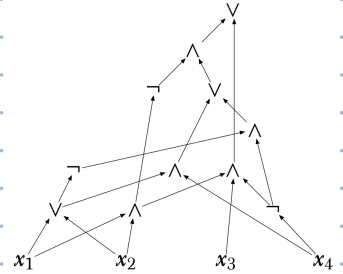
e.g. circuits

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

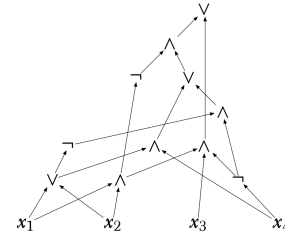
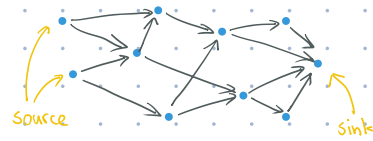


$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

Addition



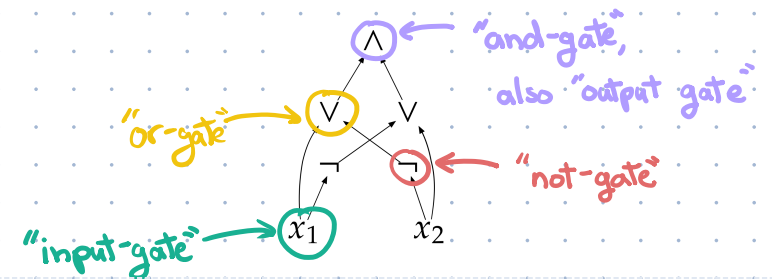
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$



(definition)

logical gates arranged as vertices
in a directed acyclic graph

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



Circuit SAT

given a circuit, decide if it
is satisfiable

Theorem: there is a poly-time algo for
Boolean SAT iff there is a poly-time algo
for Circuit SAT

Conclusion

We want, but don't have, polynomial time algos for

Boolean SAT, CNF SAT, 3-SAT, or Circuit SAT

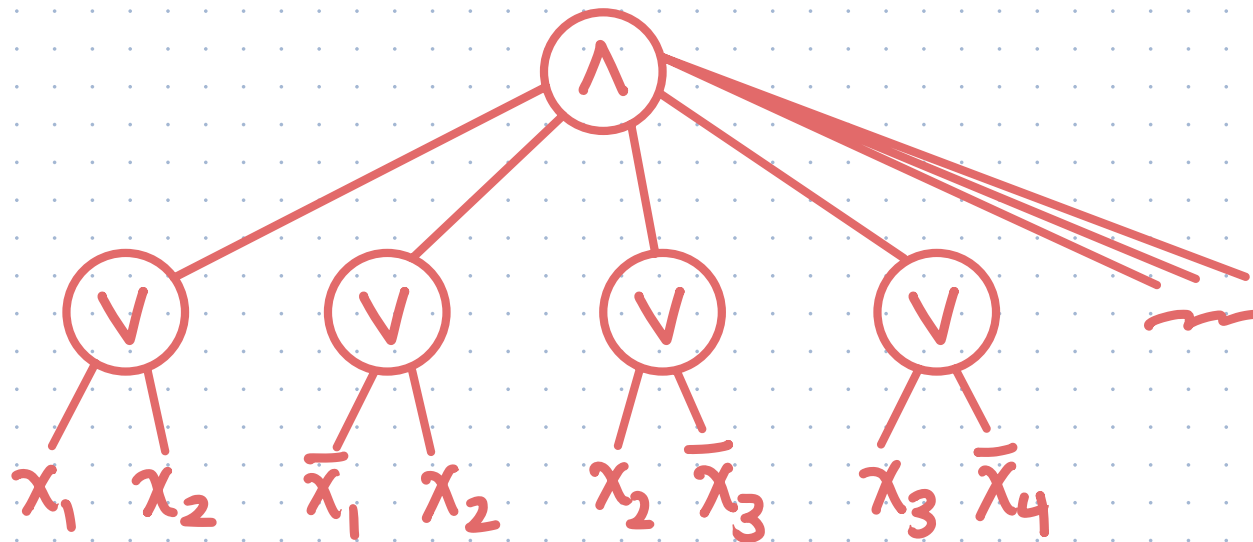
But we do know they are equivalent up to polynomial time!

2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.

$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge \overbrace{(x_2 \vee \bar{x}_3)}^{\text{clause}} \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$



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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

"clause"

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

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

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

★ "if x then y else z"

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

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

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

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

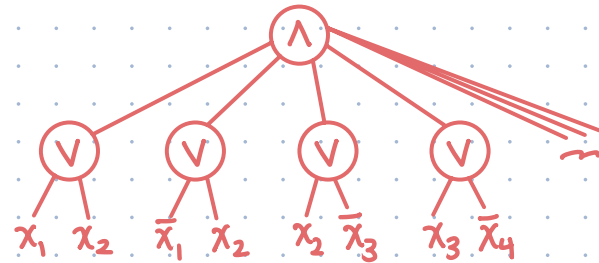
Is there a polytime algo for 2-SAT?

2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.

$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge \underbrace{(x_2 \vee \bar{x}_3)}_{\text{clause}} \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$



Max 2-SAT

like 2-SAT, except goal is to satisfy maximum number of clauses.
(not necessarily all)

Is there a polytime algo for Max 2-SAT?

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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

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

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

★ "if x then y else z"

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

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

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

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \cdot n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

	Polytime Algo	Hard as ^(CNF-) SAT	Abstain
2SAT			
Max 2-SAT			

2-SAT

e.g. $\mathcal{F}(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$

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

ideas?

2-SAT

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

- if $x_1 = \text{true}$:

$$(\bar{x}_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(\bar{x}_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

X

- if $x_1 = \text{false}$:

$$(x_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(x_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

X

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

Implication graph

- vertex for each literal $x_i, \bar{x}_i$
- for clause $(a \vee b)$, add edges

$$\bar{a} \xrightarrow{(a \vee b)} b$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

- if $x_1 = \text{true}$:

$$(\bar{x}_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(\bar{x}_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

- if $x_1 = \text{false}$:

$$(x_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(x_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

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

Implication graph

- vertex for each literal $x_i, \bar{x}_i$
- for clause $(a \vee b)$, add edges

$$\bar{a} \xrightarrow{(a \vee b)} b$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

- if $x_1 = \text{true}$:

$$(\bar{x}_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(\bar{x}_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

- if $x_1 = \text{false}$:

$$(x_1 \vee x_2) \Rightarrow x_2 = \text{true}$$

$$(x_1 \vee \bar{x}_2) \Rightarrow x_2 = \text{false}$$

- Implication graph
- vertex for each literal $x_i, \bar{x}_i$
 - for clause $(a \vee b)$, add edges

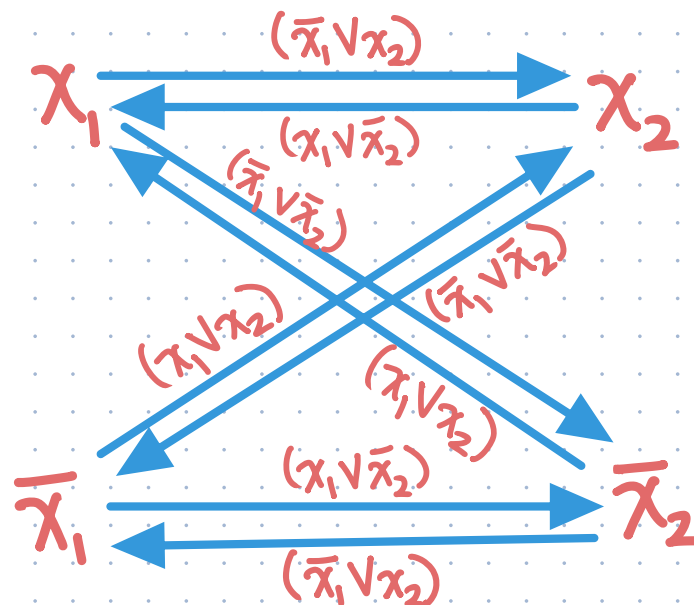
$$\bar{a} \xrightarrow{(a \vee b)} b$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

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



Implication graph

- vertex for each literal $x_i, \bar{x}_i$
- for clause $(a \vee b)$, add edges

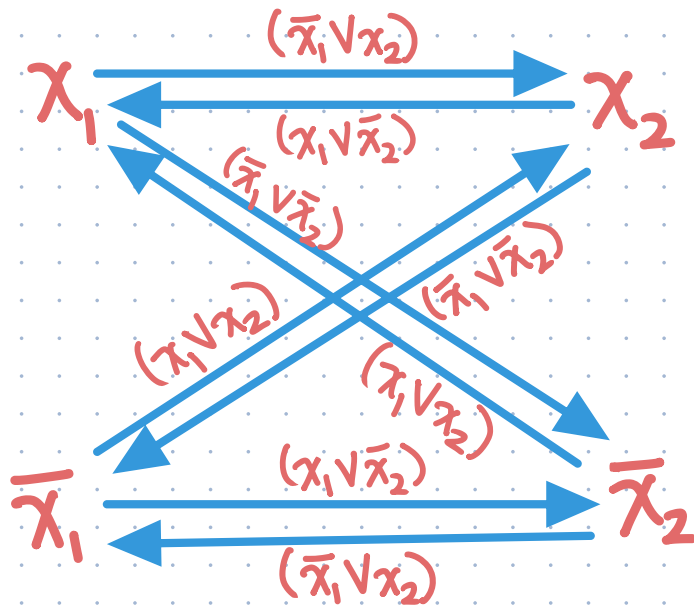
$$\bar{a} \xrightarrow{(a \vee b)} b$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

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



an assignment

$$x_1, \dots, x_n \in \{\text{TRUE}, \text{FALSE}\}$$

is satisfying iff we don't have any violating edges

"true $\rightarrow$ false"

in implication graph

- Implication graph
- vertex for each literal $x_i, \bar{x}_i$
 - for clause $(a \vee b)$, add edges

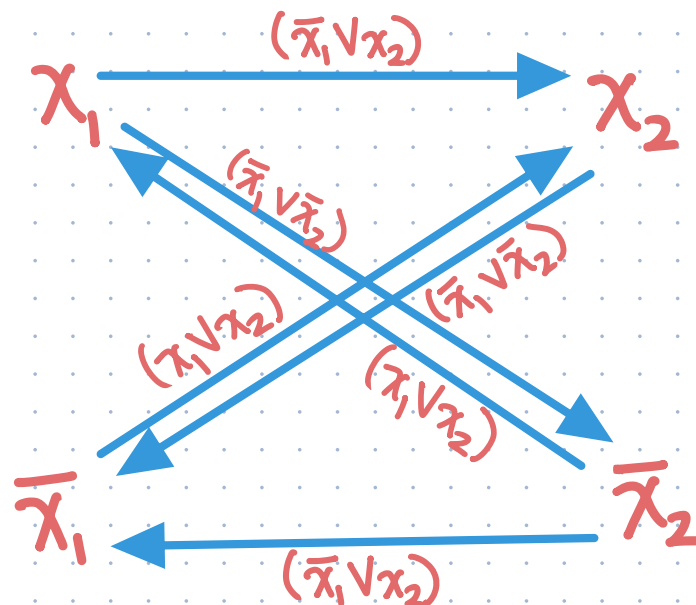
$$\bar{a} \xrightarrow{(a \vee b)} b$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

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



Implication graph

- vertex for each literal $x_i, \bar{x}_i$
- for clause $(a \vee b)$, add edges

$$\bar{a} \xrightarrow{(a \vee b)} b$$

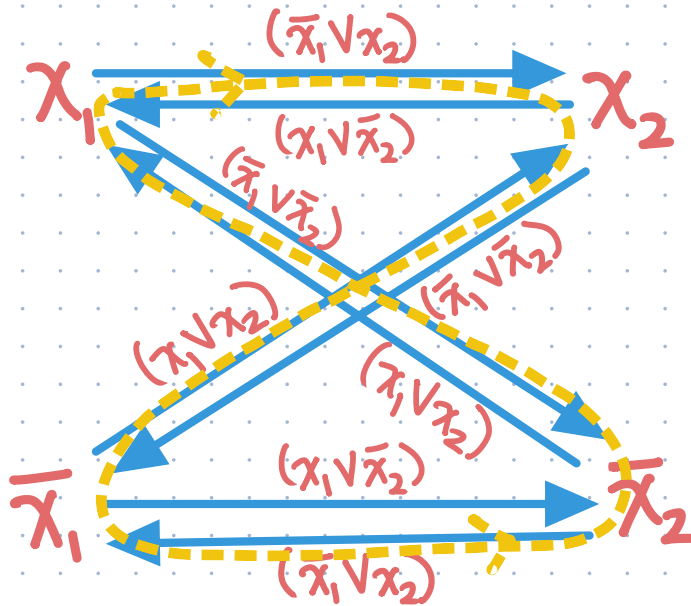
" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ "

$$\bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

$$\mathcal{I}(x_1, x_2) =$$

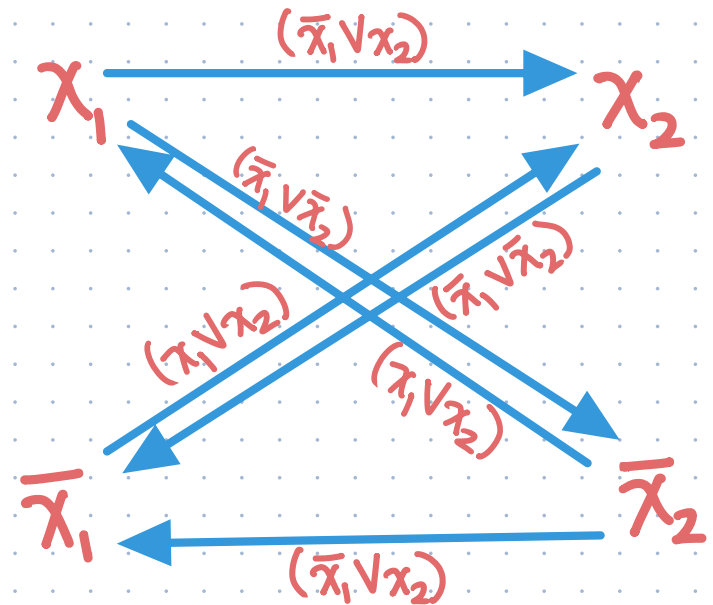
$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$



directed cycle w/ x_1 & $\bar{x}_1$

$$\mathcal{I}(x_1, x_2) =$$

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



no directed cycle w/
complementary literals

claim: 2-SAT formula $f(x_1, \dots, x_n)$ is satisfiable
 iff for all x_i , x_i and $\bar{x}_i$ are not strongly
 connected in implication graph.

Easy direction: if x_i and $\bar{x}_i$ are strongly
 connected, then f is not satisfiable.

Harder direction: suppose x_i and $\bar{x}_i$ are
 not strongly connected for all i .

need to compute a satisfying assignment

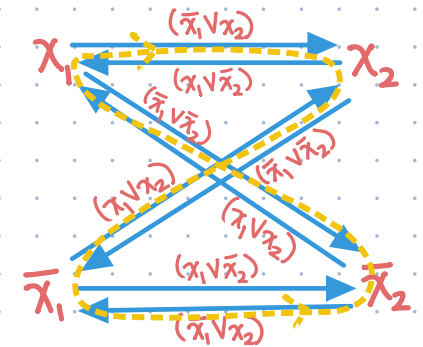
Implication graph

- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,

$$\bar{a} \xrightarrow{(a \vee b)} b \quad \bar{b} \xrightarrow{(a \vee b)} a$$

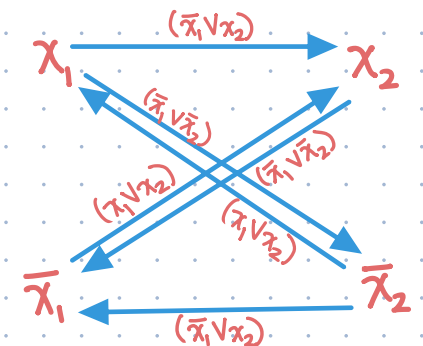
" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

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



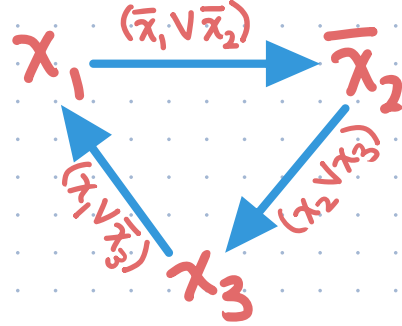
directed cycle w/ x_i & $\bar{x}_i$

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



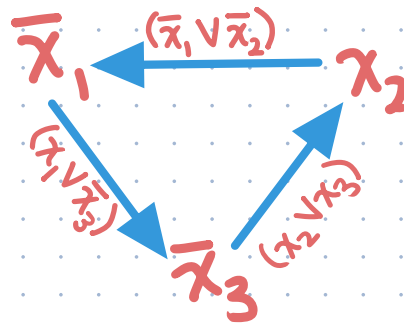
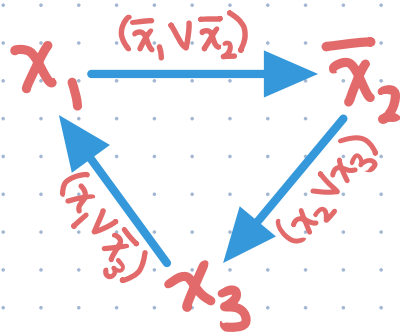
no directed cycle w/
 complementary literals

Consider an ^(strongly connected component) SCC of implication graph



all literals must have same assignment

each arc $(a \rightarrow b)$ has complementary arc $(\bar{b} \rightarrow \bar{a})$
 $\Rightarrow x$ can reach y iff $\bar{y}$ can reach $\bar{x}$



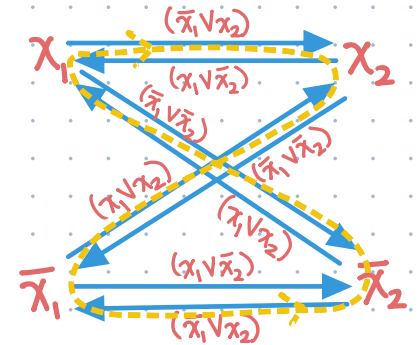
each SCC C has complement SCC $\bar{C}$
w/ negated literals & reversed edges

Implication graph

- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,
 $\bar{a} \xrightarrow{(a \vee b)} b$ $\bar{b} \xrightarrow{(a \vee b)} a$
" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

$\mathcal{I}(x_1, x_2) =$

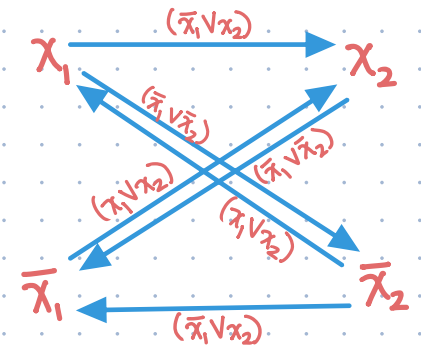
$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$



directed cycle w/ x_1 & $\bar{x}_1$

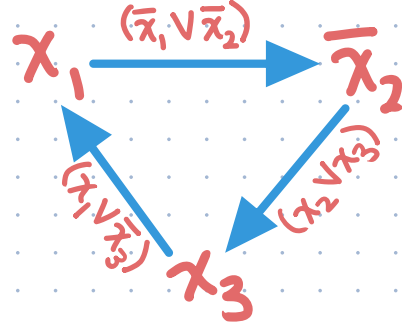
$\mathcal{I}(x_1, x_2) =$

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



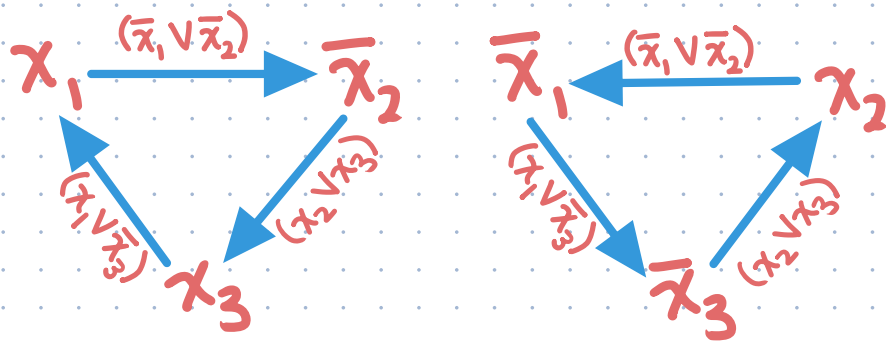
no directed cycle w/ complementary literals

Consider an ^(strongly connected component) SCC of implication graph



all literals must have same assignment

each arc $(a \rightarrow b)$ has complementary arc $(\bar{b} \rightarrow \bar{a})$
 $\Rightarrow x$ can reach y iff $\bar{y}$ can reach $\bar{x}$

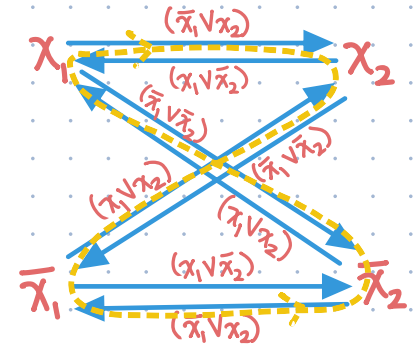


each SCC C has complement SCC $\bar{C}$
w/ negated literals & reversed edges

Implication graph

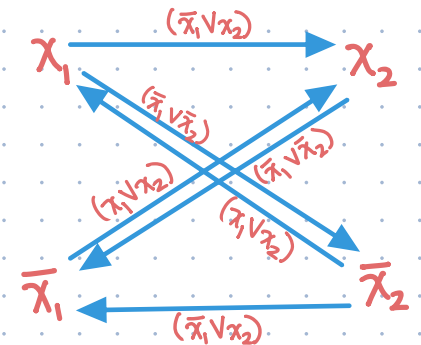
- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,
 $\bar{a} \xrightarrow{(a \vee b)} b$ $\bar{b} \xrightarrow{(a \vee b)} a$
" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

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



directed cycle w/ x_1 & $\bar{x}_1$

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



no directed cycle w/ complementary literals

claim: 2-SAT formula $f(x_1, \dots, x_n)$ is satisfiable
 iff for all x_i , x_i and $\bar{x}_i$ are not strongly
 connected in implication graph.

Easy direction: if x_i and $\bar{x}_i$ are strongly
 connected, then f is not satisfiable.

Harder direction: suppose x_i and $\bar{x}_i$ are
 not strongly connected for all i .

need to compute a satisfying assignment

Implication graph

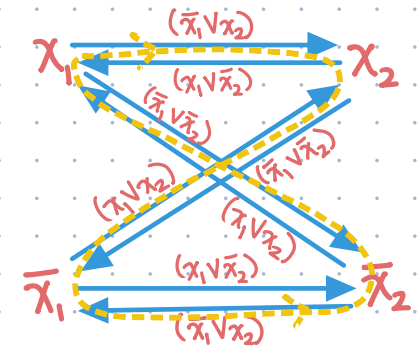
- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,

$$\bar{a} \xrightarrow{(a \vee b)} b \quad \bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "

$$f(x_1, x_2) =$$

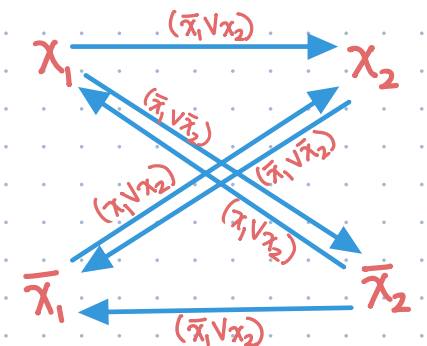
$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$



directed cycle w/ x_i & $\bar{x}_i$

$$f(x_1, x_2) =$$

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



no directed cycle w/
 complementary literals

let C be a sink SCC.

Complement $\bar{C}$ is a source SCC (why?)



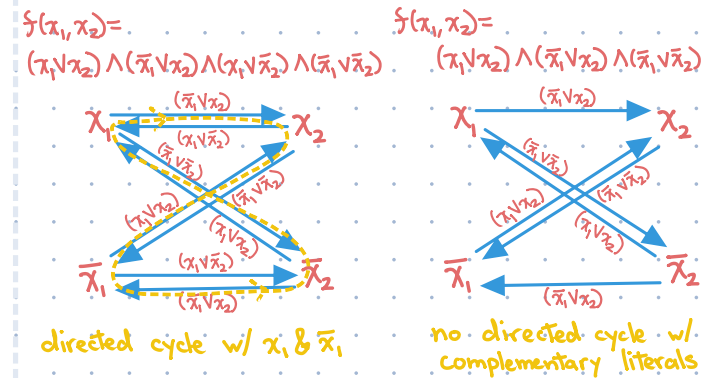
Assign (all literals in) $C = \text{true}$, $\bar{C} = \text{false}$

Remove C and $\bar{C}$ and recurse on remaining SCC's

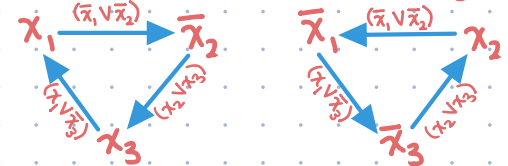
Key idea: assigning sinks to be true, sources to be false does not violate any edges

Implication graph

- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,
 $\bar{a} \xrightarrow{(a \vee b)} b$ $\bar{b} \xrightarrow{(a \vee b)} a$
 " $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "



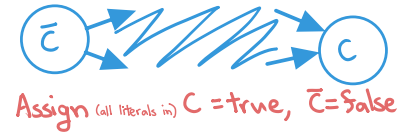
all literals in SCC have same assignment



each SCC C has complement SCC $\bar{C}$ w/ negated literals & reversed edges

2-SAT-solver(2-SAT formula $f(x_1, \dots, x_n)$)

1. Construct the implication graph for f .
2. Compute the strongly connected components (SCCs) of the implication graph.
3. If some SCC C contains both literals x_i and $\bar{x}_i$ of some variable x_i , then return C as a certificate that f is unsatisfiable.
4. For each SCC C in reverse topological order:
 - A. If C is unassigned, then assign (all literals in) C to true and $\bar{C}$ to false.
5. Return the assignment to $x_1, \dots, x_n$ computed above.

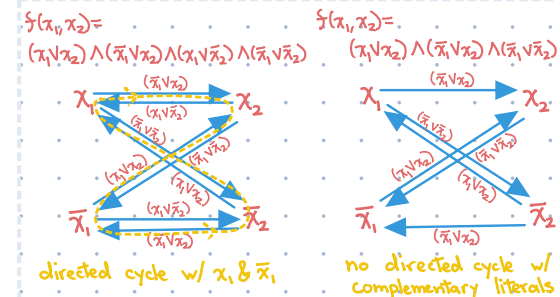
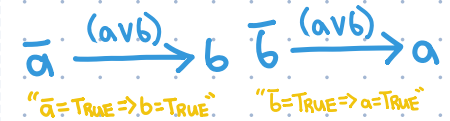


Running time

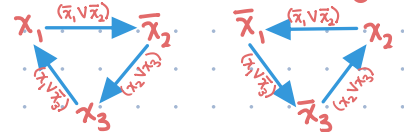
Implication graph

- vertex for each $x_i, \bar{x}_i$

- for clause $(a \vee b)$,



all literals in SCC have same assignment



each SCC C has complement SCC $\bar{C}$ w/ negated literals & reversed edges

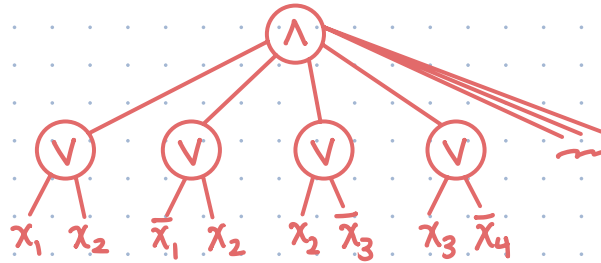
✓ 2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.

$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$

clause



Max 2-SAT

like 2-SAT, except goal is to satisfy maximum number of clauses.
(not necessarily all)

Is there a polytime algo for Max 2-SAT?

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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\neg x_4) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

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

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

★ "if x then y else z"

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

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

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

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

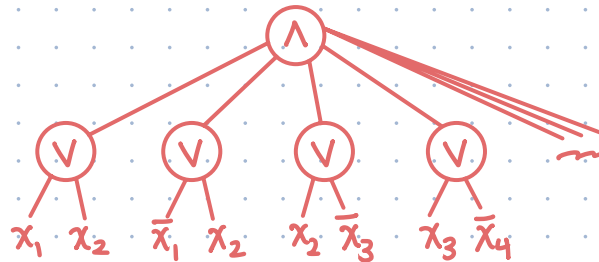
	Polytime Algo	Hard as ^(CNF-) SAT	Abstain
--	---------------	-------------------------------	---------

2SAT	X		
------	---	--	--

Max 2-SAT			
-----------	--	--	--

✓ 2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.



$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$

Max 2-SAT

like 2-SAT, except goal is to satisfy maximum number of clauses.
(not necessarily all)

Max (≤ 2)-SAT

like max 2-SAT, except each clause has at most 2 literals per clause

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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"

$$f(x_1, \dots, x_n) =$$



$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\neg \vee \neg) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

* "z = x"

"both true or both false"

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

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

* "if x then y else z"

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

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

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

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

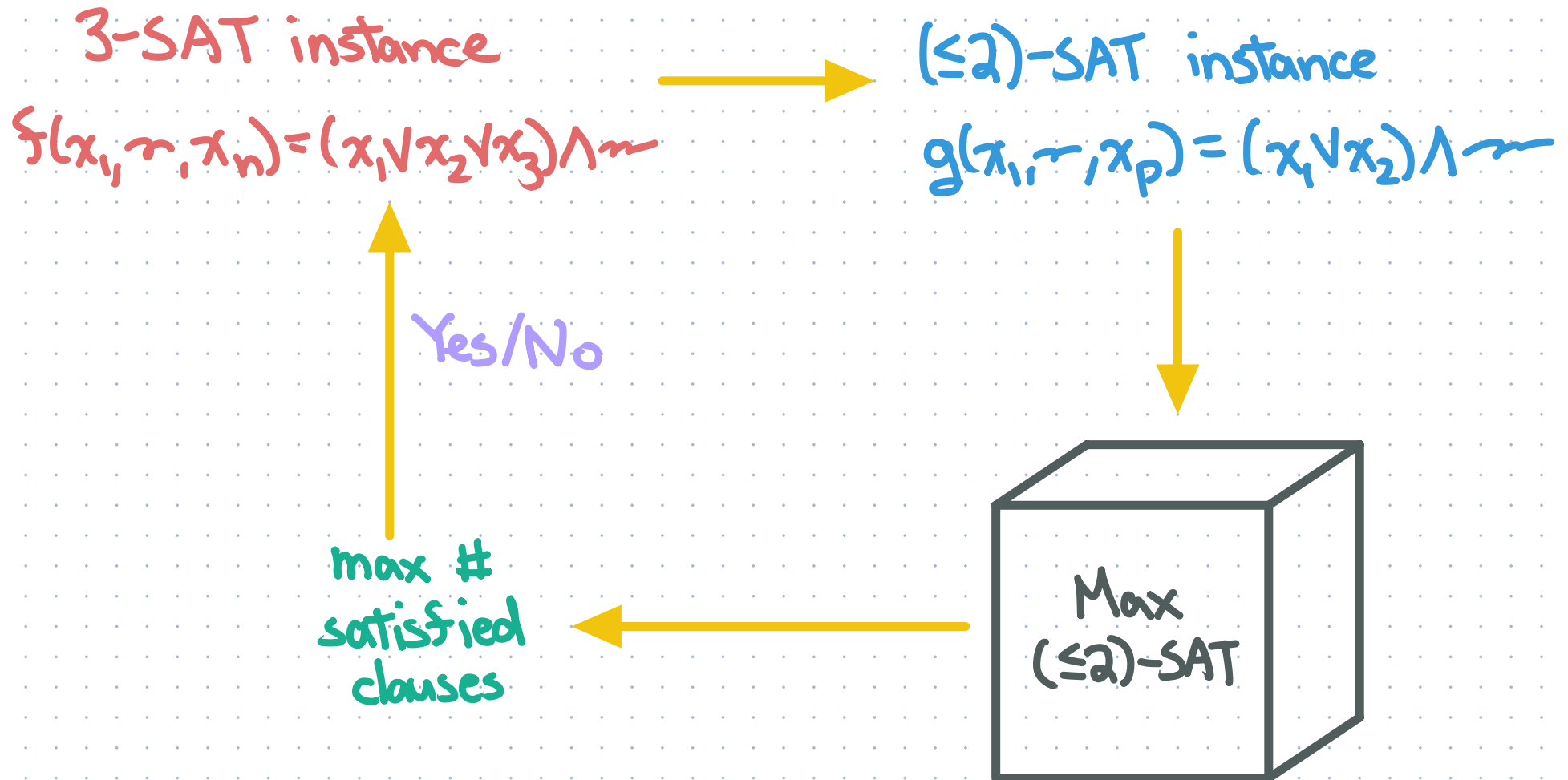
Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.

Proof setup



need to convert

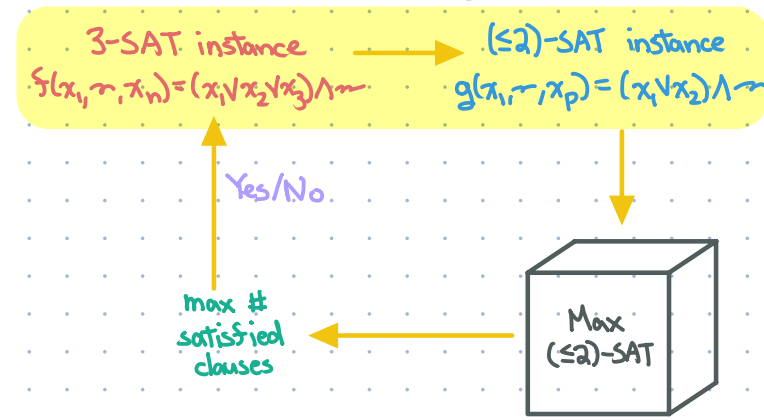
CNF w/ 3 literals per clause to

CNF w/ ≤ 2 literals per clause

ideas?

how about one clause, $(a \vee b \vee c)$?

Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



Fix a clause $(a \vee b \vee c)$

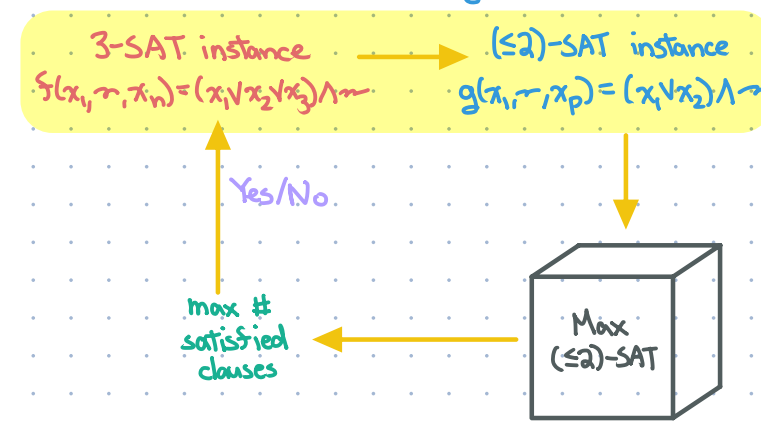
introduce new variable y

consider the 10 clauses

$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$

$(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$

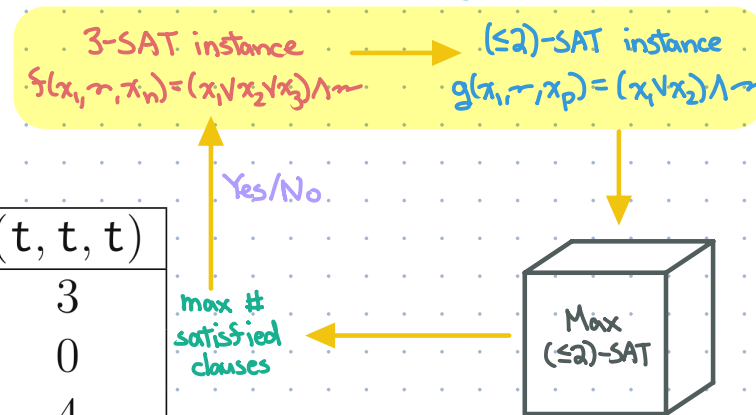
Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



$(a, b, c) =$	(f, f, f)	(t, f, f)	(t, t, f)	(t, t, t)
$(a), (b), (c)$				
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$				
$\max_{y \in \{t, f\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$				
total				

$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$
 $(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$

Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



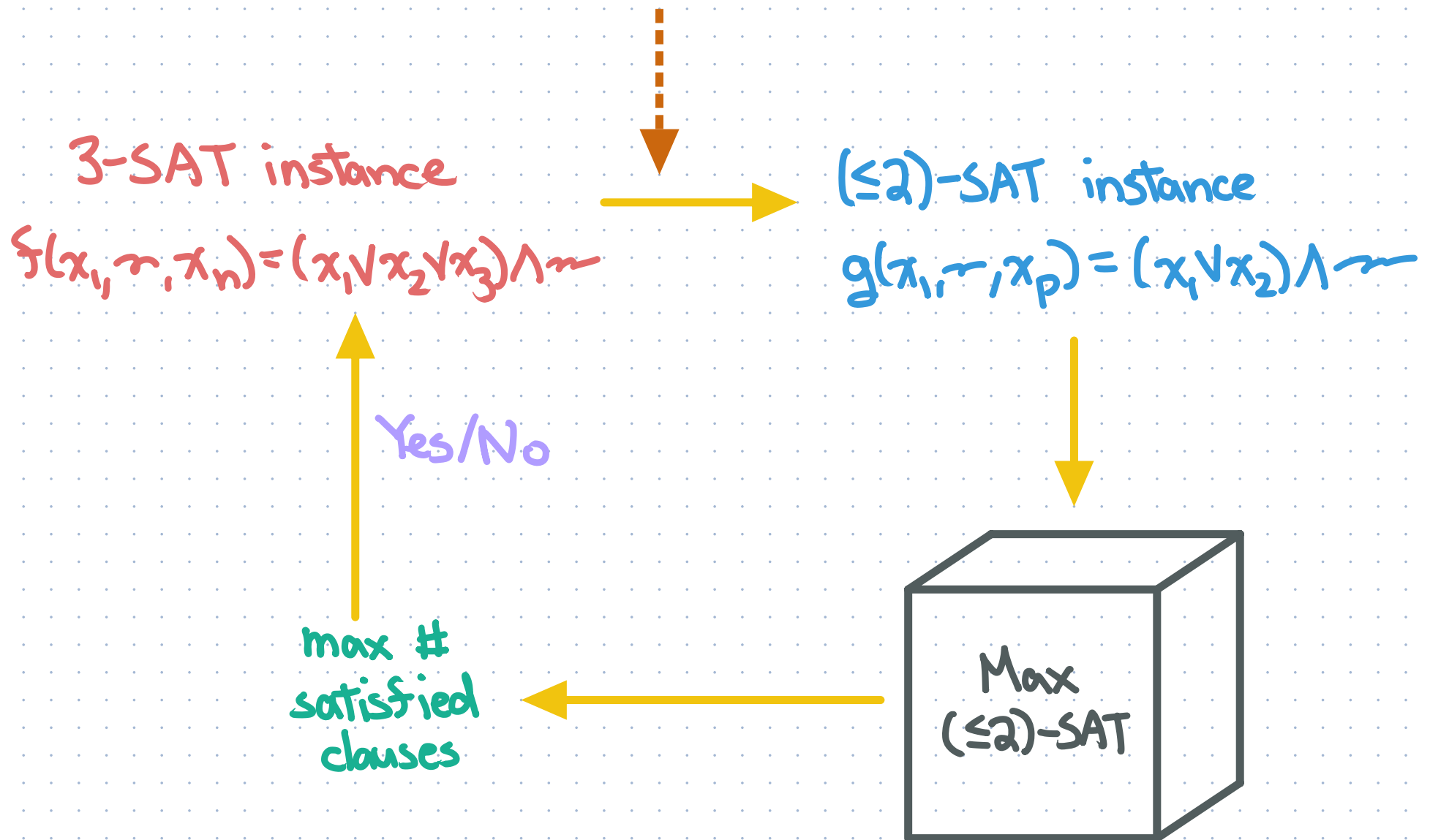
$(a, b, c) =$	(f, f, f)	(t, f, f)	(t, t, f)	(t, t, t)
$(a), (b), (c)$	0	1	2	3
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$	3	3	2	0
$\max_{y \in \{t, f\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$	3	3	3	4
total	6	7	7	7

Lemma 10.3. Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :

- (a) At most 7 clauses in $\mathcal{F}_C$ can be satisfied.
- (b) If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.
- (c) If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.

Lemma 10.3. Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :

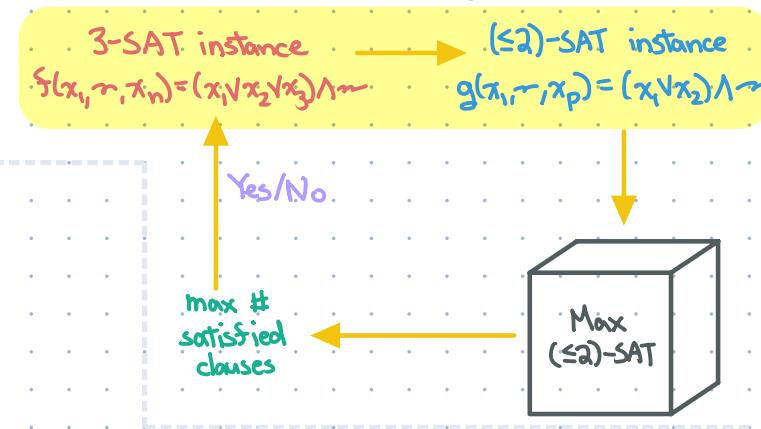
- (a) At most 7 clauses in $\mathcal{F}_C$ can be satisfied.
- (b) If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.
- (c) If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.



Lemma 10.3. Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :

- (a) At most 7 clauses in $\mathcal{F}_C$ can be satisfied.
- (b) If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.
- (c) If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.

Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



given $f(x_1, \dots, x_n)$ w/ m clauses $C_1, \dots, C_m$:

for each C_i , Lemma $\Rightarrow$ 10 clause $\mathcal{F}_{C_i}$, new var. y_i

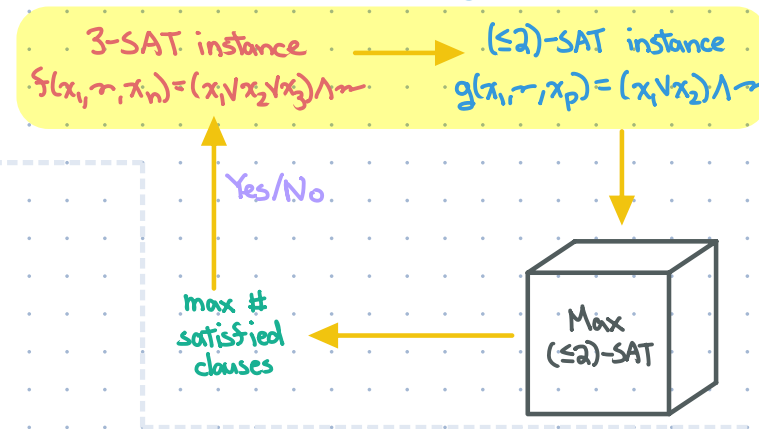
def. $g(x_1, \dots, x_n, y_1, \dots, y_m)$ as conjunction of union of $\mathcal{F}_{C_i}$'s

if f is satisfiable:

Lemma 10.3. Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :

- (a) At most 7 clauses in $\mathcal{F}_C$ can be satisfied.
- (b) If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.
- (c) If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.

Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



given $f(x_1, \dots, x_n)$ w/ m clauses $C_1, \dots, C_m$:

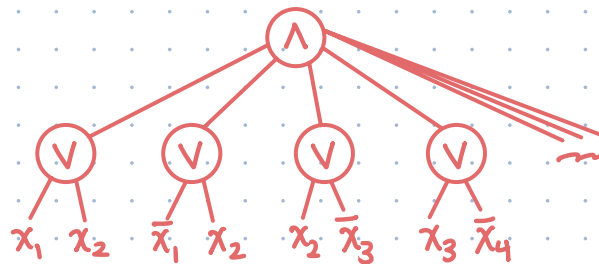
for each C_i , Lemma $\Rightarrow$ 10 clause $\mathcal{F}_{C_i}$, new var. y_i

def. $g(x_1, \dots, x_n, y_1, \dots, y_m)$ as conjunction of union of $\mathcal{F}_{C_i}$'s

if f is not satisfiable:

✓ 2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.



$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge \underbrace{(x_2 \vee \bar{x}_3)}_{\text{clause}} \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$

Max 2-SAT

like 2-SAT, except goal is to satisfy maximum number of clauses.
(not necessarily all)

✓ Max (≤ 2)-SAT

like max 2-SAT, except each clause has at most 2 literals per clause

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 $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"

$$f(x_1, \dots, x_n) =$$



$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

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

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

★ "if x then y else z"

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

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

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

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Theorem: A polytime algorithm for max (≤ 2)-SAT implies a polynomial time algorithm for SAT.

Theorem: A polytime algorithm for max 2-SAT implies a polynomial time algorithm for SAT.

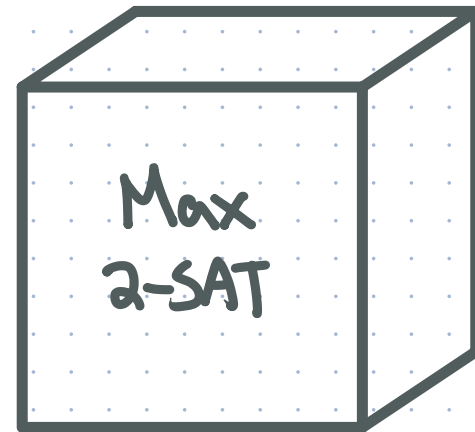
(≤ 2) -SAT instance

$$f(x_1, \dots, x_n) = (x_1 \vee x_2) \wedge \dots$$



2-SAT instance

$$g(x_1, \dots, x_p) = (x_1 \vee x_2) \wedge \dots$$



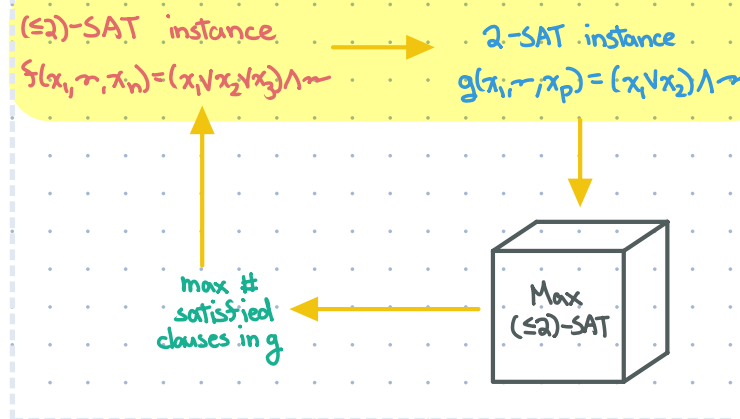
max #
satisfied
clauses in g



Suppose $f(x_1, \dots, x_n)$ has

- k singleton clauses $C_1, \dots, C_k$
- l 2-variable clauses $D_1, \dots, D_l$

Theorem: A polytime algorithm for max 2-SAT implies a polynomial-time algorithm for SAT.



Q: how to convert a singleton clause (a) to 2-variable clause?

$(a \vee y), (a \vee \bar{y})$ for new variable y

- if $a = \text{true}$, both satisfied no matter y
- if $a = \text{false}$, 1 satisfied no matter y

Suppose $f(x_1, \dots, x_n)$ has

- k singleton clauses $C_1, \dots, C_k$
- l 2-variable clauses $D_1, \dots, D_l$

let $g(x_1, \dots, x_n, y_1, \dots, y_k)$

replace each $C_i = (a_i)$ w/ $(a_i \vee y_i) \wedge (a_i \vee \bar{y}_i)$

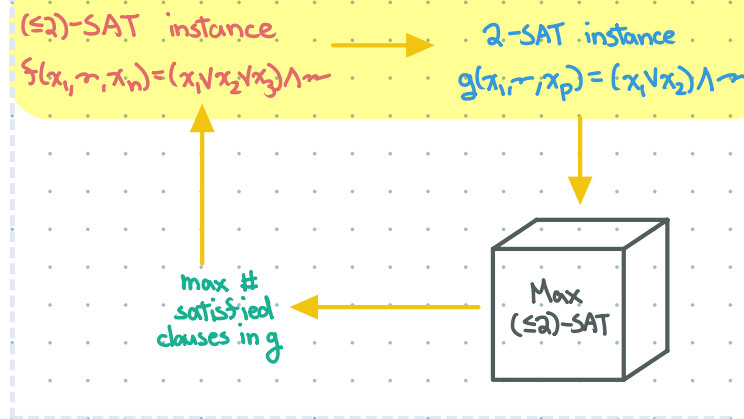
if $x_1, \dots, x_n$ satisfies p clauses in f ,

then $\forall y_1, \dots, y_k \in \{\text{TRUE}, \text{FALSE}\}$,

$x_1, \dots, x_n, y_1, \dots, y_k$ satisfy clauses in g

$$\begin{array}{l} \text{max \#} \\ \text{satisfied} \\ \text{clauses in } g \end{array} = \begin{array}{l} \text{max \#} \\ \text{satisfied} \\ \text{clauses in } f \end{array} + k$$

Theorem: A polytime algorithm for max 2-SAT implies a polynomial-time algorithm for SAT.



Theorem: A polytime algorithm for max 2-SAT implies a polynomial time algorithm for SAT.

$$(a_i \vee y_i) \wedge (a_i \vee \neg y_i)$$

(≤ 2) -SAT instance

$$f(x_1, \dots, x_n) = (x_1 \vee x_2) \wedge \dots$$

2-SAT instance

$$g(x_1, \dots, x_p) = (x_1 \vee x_2) \wedge \dots$$

$-k$

max #
satisfied
clauses in g



Chapter 10

2-SAT and Max 2-SAT

Chapter 9 introduced various forms of *satisfiability problems*, such as Boolean SAT, Circuit SAT, and CNF SAT. (CNF SAT is often just called *SAT*.) Each takes as input some kind of function $f(x_1, \dots, x_n)$ of n boolean variables $x_1, \dots, x_n$ and asks if there is an assignment $x_1, \dots, x_n \in \{\mathbf{true}, \mathbf{false}\}$ that satisfies f ; i.e., such that $f(x_1, \dots, x_n) = \mathbf{true}$. We were not able to obtain polynomial time algorithms for any of these problems. However, we were able to derive polynomial time reductions between them. Consequently, there is a polynomial algorithm for any one of them iff there is a polynomial time algorithm for all of them.

The most restrictive version of SAT we discussed was *3-SAT*. This is where the boolean formula is expressed in *conjunctive normal form* with exactly three (distinct) literals per clause. Surprisingly, this restricted version turns out to be just as hard as boolean SAT and circuit SAT.

Here we go even further, discussing 2-SAT. *2-SAT* is the special case of (CNF-)SAT where each clause has exactly two literals. *Max 2-SAT* is like 2-SAT except rather than decide if the formula is satisfiable, the goal is to find an assignment of variables that satisfies as many clauses as possible. We are interested in the following questions:

1. Is there a polynomial time algorithm for 2-SAT, and/or does a polynomial time algorithm for 2-SAT imply a polynomial time algorithm for (CNF-)SAT?
2. Is there a polynomial time algorithm for Max 2-SAT, and/or does a polynomial time algorithm for Max 2-SAT imply a polynomial time algorithm for SAT?

10.1 2-SAT

We first consider 2-SAT. Formally, the input consists of a function $f(x_1, \dots, x_n)$ in CNF form and exactly two variables per clause. For example,

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

The goal is to find a satisfying assignment for f , or declare that none exists.

We want to either (a) design a polynomial time algorithm for this problem, or (b) relate 2-SAT to a problem we already believe is hard, like 3-SAT, via a polynomial time reduction.

2-SAT, unlike 3-SAT, has the following “forcing” characteristic. Consider the example above, and in particular the first clause,

$$(x_1 \vee x_2).$$

To satisfy f , either x_1 or x_2 have to be true. If we set $x_1 = \text{false}$, then we have to set $x_2 = \text{true}$. If we set $x_2 = \text{false}$, then we have to set $x_1 = \text{true}$.

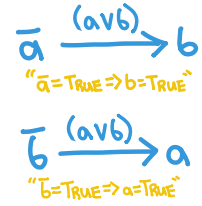
For the formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$, we can iterate this logic as follows.

Suppose $x_1 = \text{false}$. Then the first clause, $(x_1 \vee x_2)$, requires $x_2 = \text{true}$. But if $x_2 = \text{true}$, then the third clause, $(x_1 \vee \bar{x}_2)$, requires $x_2 = \text{false}$. So it is impossible to satisfy f if $x_1 = \text{false}$.

Similarly, if $x_1 = \text{true}$, then the second clause forces $x_2 = \text{true}$. If $x_2 = \text{true}$, then the last clause forces $x_1 = \text{true}$. So it is impossible to satisfy all clauses if $x_1 = \text{false}$. We conclude that f is unsatisfiable.

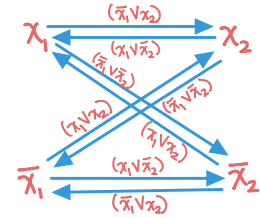
We encode these “implications” in the following *implication graph*.

- For each variable x_i , we create two vertices for the two literals x_i and $\bar{x}_i$.
- For each clause $(a \vee b)$, where a and b are literals (such as x_1 or $\bar{x}_2$), we add arcs $(\bar{a}, b)$ and $(\bar{b}, a)$.

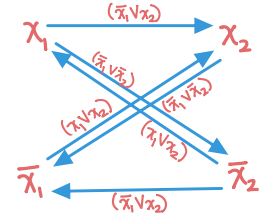


We call this graph the *implication graph* for the following reason: for any satisfying assignment, and any arc (a, b) , if $a = \text{true}$, then $b = \text{true}$.

On the right, we draw the implication graph for the unsatisfiable formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$. Each arc is labeled by the clause that produced it. Observe that x_1 and $\bar{x}_1$ both lie in the same directed cycle $(x_1, x_2, \bar{x}_1, \bar{x}_2, x_1)$. This cycle is a certificate that it is impossible to satisfy f .



Next we draw the implication graph for the formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$. (This formula omits the third clause from the previous example.) Observe that this time, there is no directed cycle containing both literals x_i and $\bar{x}_i$ for the same variable x_i . f is also satisfiable, with a satisfying assignment given by $x_1 = \text{false}$ and $x_2 = \text{true}$.

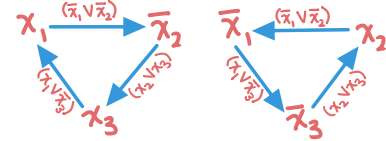


Is it that the formula with no “bad” cycle in the implication graph is satisfiable? We claim it is necessary. That is, we claim that a 2-SAT formula is satisfiable iff for any variable x_i , the literals x_i and $\bar{x}_i$ are not strongly connected in the implication graph.

One direction is easier: if x_i and $\bar{x}_i$ are in the same directed cycle, then setting $x_i = \text{true}$ implies that $\bar{x}_i = \text{true}$ (i.e., $x_i = \text{false}$), and setting $\bar{x}_i = \text{true}$ implies that $x_i = \text{true}$. Since x_i is assigned exactly one of **true** or **false**, we conclude that the formula is unsatisfiable.

Conversely, suppose that the literals x_i and $\bar{x}_i$ are not strongly connected for any variable x_i . We describe an algorithm that constructs a satisfying assignment. The algorithm is based on strongly connected components and leverages a certain *complementary* structure among the SCC’s.

For a set of literals S , let $\bar{S} = \{\bar{u} : u \in S\}$ be the set of literals obtained by negating the variables in S . Observe that for every arc (u, v) , we have the arc $(\bar{v}, \bar{u})$. Consequently, for every strongly connected component C , $\bar{C}$ is also a strongly connected component. We call $\bar{C}$ the complementary SCC of C .



Consider an SCC C of the implication graph. In a satisfying assignment, all in C must be set to **true**, or to **false**, simultaneously. Likewise, all literals in the complementary SCC $\bar{C}$ must have the same assignment, and the opposite assignment as (all the literals in) C .

We now describe the algorithm. Given a 2-SAT formula f , we construct the implication graph and compute the strongly connected components. If any strongly connected component contains both literals of the same variable, we return **false**. Otherwise we need to construct a satisfying assignment.

We process the SCCs in reverse topological order. The first component C is a sink component. We mark all literals in C as **true** and mark all the literals in $\bar{C}$ as **false**. For each subsequent component C in reverse topological order, if the literals in C are not yet marked, we mark all the literals in C as **true**, and all the literals in $\bar{C}$ as **false**.

We need to argue that this produces a satisfying assignment. We say that a

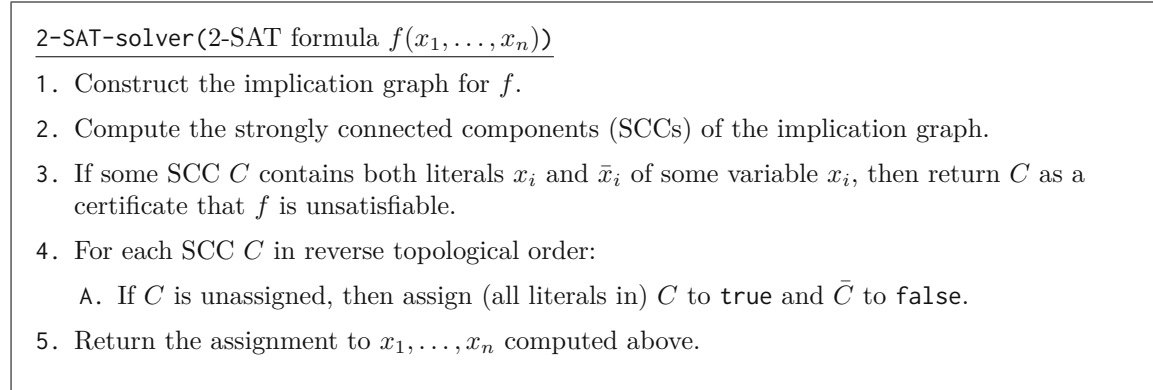


Figure 10.1: Solving 2-SAT in linear time via strongly connected components.

true/false labeling of the SCCs is *valid* if for every arc (C, D) between SCCs C and D , if $C = \mathbf{true}$, then $D = \mathbf{true}$. A valid labeling of the SCCs implies a satisfying assignment to the SAT formula.

Suppose we label an SCC C as **true** and $\bar{C}$ as **false**. We want to show that the label is valid for all the arcs leaving C and all the arcs entering $\bar{C}$.

Consider an arc (C, D) . D succeeds C in the topological order, so it is labeled before C . Moreover, the arc (C, D) implies that D was not a source among the unmarked SCCs when it was labeled. Thus D must have been a sink, and labeled **true**, as desired.

Now consider an arc $(B, \bar{C})$. By complementarity, $(C, \bar{B})$ is also an arc. As argued above, $\bar{B}$ is labeled **true**, hence B is labeled **false**, as desired.

As for the running time, building the graph and computing the SCCs takes $O(m + n)$ time. The iterative marking process marks each vertex once, and takes $O(n)$ time. So the algorithm takes $O(m + n)$ time overall.

Altogether we have established the following.

Theorem 10.1. *Given a 2-SAT instance $f(x_1, \dots, x_n)$ with n variables and m clauses, in $O(m + n)$ time, one can compute either a satisfying assignment or a certificate of infeasibility.*

10.2 Max 2-SAT

Thus 2-SAT is polynomial time solvable. The next problem to discuss is max 2-SAT. Recall that max 2-SAT is like 2-SAT, except the goal is to satisfy the maximum number of clauses. In particular the input formula need not be satisfiable.

Of course max 2-SAT solves 2-SAT, while 2-SAT does not solve max 2-SAT. So the polynomial time algorithm for 2-SAT does not imply a polynomial time algorithm for max 2-SAT.

Max (≤ 2)-SAT. It is easier to first consider a closely related problem we call *max at-most-2-SAT*, or $\max(\leq 2)$ -SAT. The problem is the same as max 2-SAT except each clause has *at most* 2 literals; in particular, we may have singleton clauses.

Theorem 10.2. *A polynomial time algorithm for $\max(\leq 2)$ -SAT implies a polynomial time algorithm for SAT.*

We reduce from 3-SAT. The setup is as follows. We take as input a formula $f(x_1, \dots, x_n)$ in CNF with exactly three literals per clause. We want to decide if f is satisfiable.

We assume access to a black box that can solve $\max(\leq 2)$ -SAT. To leverage this black box, we want to produce a formula $g(y_1, \dots, y_p)$ in CNF with at most 2 literals per clause, such that the maximum number of satisfying clauses in g can somehow tell us if f is satisfiable or not.

The main challenge is converting a 3-literal disjunction to clauses with only 1 or 2 variables. Note that our “assignment” trick from chapter 9 does not work as that gadget produces disjunctions with 3 variables. We apply a different gadget, described in the following lemma.

Lemma 10.3. *Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :*

- (a) *At most 7 clauses in $\mathcal{F}_C$ can be satisfied.*
- (b) *If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.*
- (c) *If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.*

Proof. Suppose we have a clause $(a \vee b \vee c)$, where a, b, c are literals. Let $y \in \{\text{true}, \text{false}\}$ be a new variable, and the following 10 clauses:

$$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}), (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$$

An assignment satisfies $(a \vee b \vee c)$ iff there is some choice of $y \in \{\text{true}, \text{false}\}$ that 7 of the 10 clauses in the gadget for $(a \vee b \vee c)$. If the assignment does not satisfy $(a \vee b \vee c)$ then any choice of y leads to satisfying at most 6 of the clauses. See fig. 10.2 on the next page. $\square$

$(\mathbf{a}, \mathbf{b}, \mathbf{c}) =$	$(\mathbf{f}, \mathbf{f}, \mathbf{f})$	$(\mathbf{t}, \mathbf{f}, \mathbf{f})$	$(\mathbf{t}, \mathbf{t}, \mathbf{f})$	$(\mathbf{t}, \mathbf{t}, \mathbf{t})$
$(a), (b), (c)$	0	1	2	3
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$	3	3	2	0
$\max_{y \in \{\mathbf{t}, \mathbf{f}\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$	3	3	3	4
total	6	7	7	7

Figure 10.2: Number of satisfied gadget clauses for $(a \vee b \vee c)$. The columns are based on the number of literals among a, b, c that are satisfied.

We apply this to the 3-CNF formula f as follows. Suppose we have m clauses $C_1, \dots, C_m$. For each i , we apply lemma 10.3 to C_i . Let $\mathcal{F}_i$ denote the corresponding set of 10 disjunctions, and let y_i be the new auxiliary variable, produced by lemma 10.3 when applied to C_i . Finally, let $g(x_1, \dots, x_n, y_1, \dots, y_m)$ be the formula taking the conjunction of all the clauses in all the $\mathcal{F}_i$'s. g is in CNF, and has $m+n$ variables, $10m$ clauses, and at most 2 literals in every clause. We claim that f is satisfiable iff one can satisfy at least $7m$ clauses of g .

If f has a satisfying assignment $x_1, \dots, x_n$, then we can choose $y_1, \dots, y_m$ so that the combined assignment $(x_1, \dots, x_n, y_1, \dots, y_m)$ satisfies 7 clauses in each $\mathcal{F}_i$. Summing over $i \in [m]$ gives $7m$ satisfied clauses, total.

Conversely suppose f does not have a satisfying assignment. Consider any assignment $x_1, \dots, x_n$. $x_1, \dots, x_n$ does not satisfy some clause C_i , and consequently, any assignment of y_i leads to at most 6 clauses in $\mathcal{F}_i$ being satisfied. Since we can satisfy at most 7 clauses in any other $\mathcal{F}_j$, we conclude that at most $7m - 1$ clauses can be satisfied in g .

To recap, given f , we construct the formula g by applying lemma 10.3 to each clause of f . We apply a black box to g that computes the maximum number of satisfiable clauses.

From $\max(\leq 2)$ -SAT to max 2-SAT. We now show that max 2-SAT is hard by reducing from $\max(\leq 2)$ -SAT.

Theorem 10.4. *A polynomial time algorithm for max 2-SAT implies a polynomial time algorithm for SAT.*

We give a reduction for $\max(\leq 2)$ -SAT to max 2-SAT. The only difference between the two problems is that (≤ 2) -SAT allows for singleton clauses of the form (a) , where a is a literal. We first describe a clause to transform a singleton clause to a constant number of 2-literal disjunctions.

Consider a singleton clause a . Let $y \in \{\text{true}, \text{false}\}$ be a new variable, and consider the clauses

$$(a \vee y) \text{ and } (a \vee \bar{y}).$$

If $a = \text{true}$, then both clauses are satisfied. If $a = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies 1 of the 2 clauses.

Now suppose we have an instance of Max (≤ 2) -SAT. The input consists of a CNF formula $f(x_1, \dots, x_n)$ with k singleton clauses $C_1, \dots, C_k$ and ℓ clauses $D_1, \dots, D_\ell$ with 2 clauses each. For each singleton clause $C_i = (a_i)$ we introduce a new variable y_i and create the 2 clauses

$$(a \vee y_i) \text{ and } (a \vee \bar{y}_i).$$

Consider the 2-sat formula $g(x_1, \dots, x_n, y_1, \dots, y_k)$ where we replace the k singleton clauses in f with the $2k$ auxiliary clauses above.

Given an assignment to $x_1, \dots, x_n$ that satisfies p of the k singleton clauses, and q of the non-singleton clauses, any choice of $y_1, \dots, y_k$ satisfies exactly $p + k$ of the auxiliary clauses replacing the singleton clauses. Thus the maximum number of satisfiable clauses in g is exactly k more than the maximum number of satisfiable clauses in f .

Given a black box for max 2-SAT, and a max (≤ 2) -SAT instance $f(x_1, \dots, x_n)$, we construct the 2-SAT formula g as described above, pass it to the black box, and subtract k from the value returned by the black box.

This completes the proof of theorem 10.4

10.3 Additional notes and materials

Max 2-SAT was shown to be NP-Hard by [GJS76]. The linear time algorithm for 2-SAT is from [APT79]. Previously a polynomial time algorithm was given by [Kro67] and a different linear time algorithm was given by [EIS76].

10.4 Exercises

Exercise 10.1. Let $G = (V, E)$ be an undirected graph, with edge weights $w : E \rightarrow \mathbb{R}_{>0} \cup \mathbb{R}_{>0}$. A *cut* is a set of edges of the form

$$\delta(S) \stackrel{\text{def}}{=} \{\{u, v\} \in E : u \in S, v \notin S\}$$

for a set of vertices $S \subseteq V$. Note that $\delta(S) = \delta(V \setminus S)$. S and $V \setminus S$ are called *sides* of the cut. The *weight* of a cut $\delta(S)$ is the sum of weights over all edges $e \in \delta(S)$.

For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. Decide if $E = \delta(S)$ for some set $S \subseteq V$.
2. Maximize the weight of $\delta(S)$ over all $S \subseteq V$.¹
3. Maximize the cardinality of $\delta(S)$ over all $S \subseteq V$. (I.e., the unweighted version of the previous problem. G is allowed to have parallel edges.)
4. Decide if $|\delta(S)| \geq |E|/2$ for some $S \subseteq V$.

¹Hint: Here are the first few steps of a potential reduction from max 2-SAT. Given a 2-CNF formula, for each variable x_i , create two vertices representing the literals x_i and x'_i . Create two vertices representing **true** and **false**. Add very heavy edges between x_i and x'_i for all i , and a very heavy between **true** and **false**, so that any weight-maximizing cut has to separate x_i from x'_i for all i and **true** from **false**. In such a cut, if x_i is on the same side of the cut as **true**, we can interpret that as setting $x_i = \text{true}$. Otherwise x_i is on the same side of the cut as **false** which we can interpret as setting $x_i = \text{false}$. We now leave it to you to figure out how to model the clauses and complete the reduction.