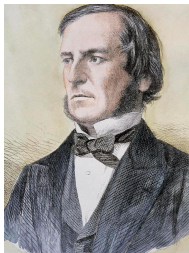


Independent Set

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 $\neg x$)



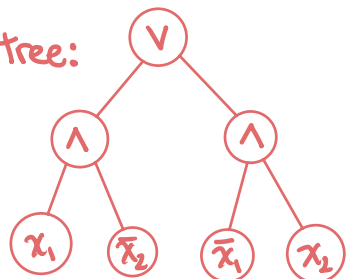
George Boole,
1815-1864

XOR ("exclusive-or")

(TRUE iff $x_1 \neq x_2$)

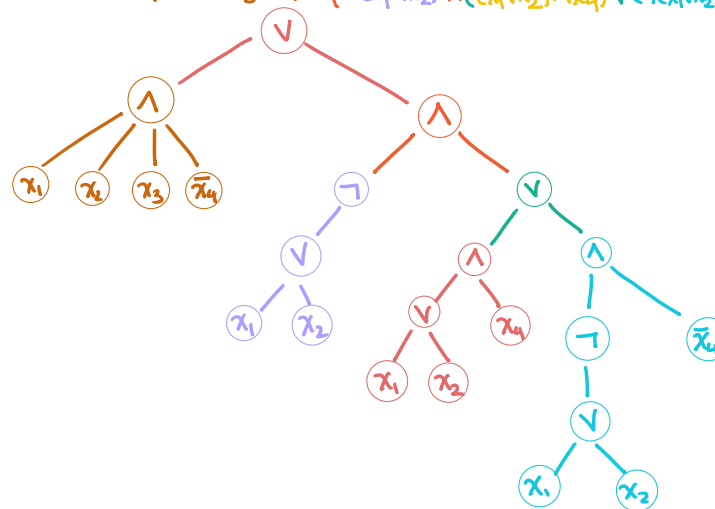
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:



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

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



Boolean Satisfiability

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

• n variables • "size" m

e.g. $\varphi(0, 1, 0, 0, \neg, 1, 1) = 0$

Goal: decide if φ is "satisfiable";

i.e., if there exists $x_1, x_2, \dots, x_n \in \{0, 1\}$
 s.t. $\varphi(x_1, x_2, \dots, x_n) = 1$

"SAT Programming Language"

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

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

★ "if x then y else z" $(x \wedge y) \vee (\bar{x} \wedge z)$

So Boolean algebra is very expressive

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

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 $x_1 \neq x_2$)

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



Go The Big Question

Is there a polynomial time algorithm for Boolean satisfiability?

x_2

$$x_2 = x_3 x_4$$

wise

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

)

$$x_2) \wedge x_4$$

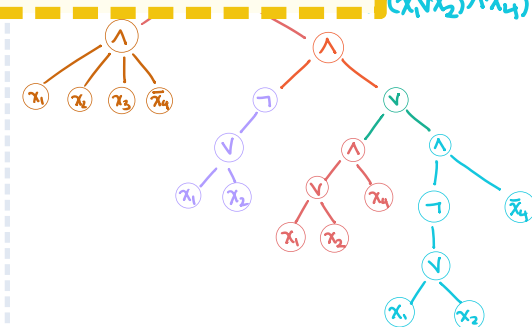
$$(x_1 \vee x_2) \wedge \bar{x}_4$$

"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"



"SAT Programming Language"

★ "z = x"

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

★ "if x then y else z"

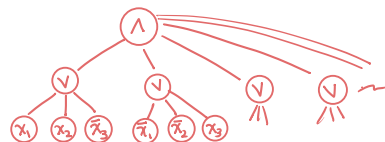
$$(x \wedge y) \vee (\bar{x} \wedge z)$$

CNF (Conjunctive Normal Form)

"and-of-or's"

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

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



(CNF-) SAT problem

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

More modest Question

Is there a polynomial time algorithm for CNF-satisfiability?

"CNF-SAT Programming Language"

"x and y"

$$x \wedge y$$

"x or y"

$$x \vee y$$

"not x"

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

★ "z = x"

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

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

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

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

★ "if x then y else z"

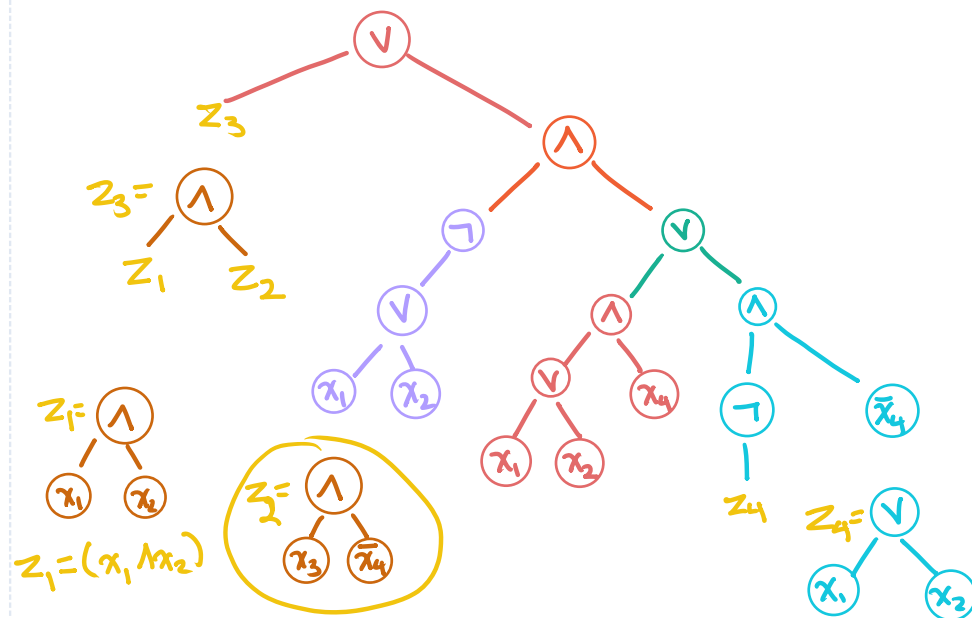
$$(x \wedge y) \vee (\bar{x} \wedge z)$$

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

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

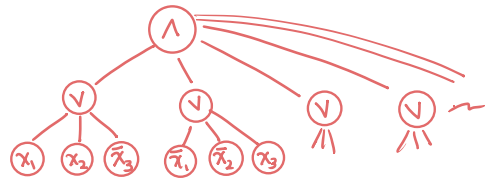
So CNF is also very expressive

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



more special case: **3-SAT**

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



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

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

"clause"

Is there a polytime algo for 3SAT?

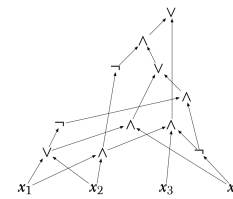
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Conclusion

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

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

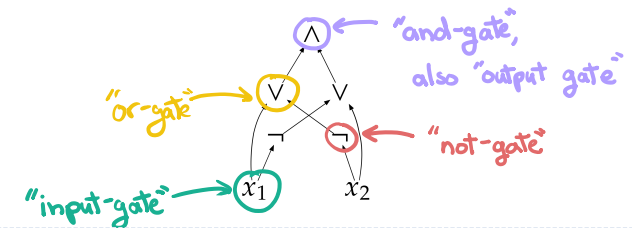
But we do know they are equivalent up to polynomial time!



(definition)

logical gates arranged as vertices in a directed acyclic graph

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



Circuit SAT

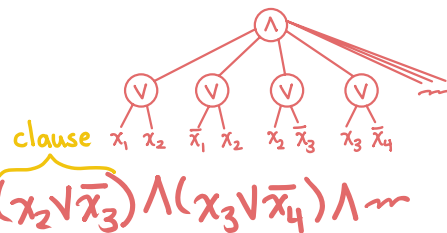
given a circuit, decide if it is satisfiable

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

2-SAT

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

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



$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$

Max 2-SAT

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

Is there a polytime algo for Max 2-SAT?

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

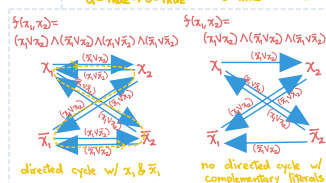
Key idea: assigning sinks to be true,
sources to be false
does not violate any edges

Implication graph

• vertex for each $x_i, \bar{x}_i$

• for clause $(a \vee b)$,

$\bar{a} \xrightarrow{(a \vee b)} b$ $\bar{b} \xrightarrow{(a \vee b)} a$
"a = TRUE $\Rightarrow$ b = TRUE" "b = TRUE $\Rightarrow$ a = TRUE"



all literals in SCC have same assignment



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

Polytime Algo

Hard as SAT

Abstain

2SAT

66%

44%

2

Max 2-SAT

60%

35%

5

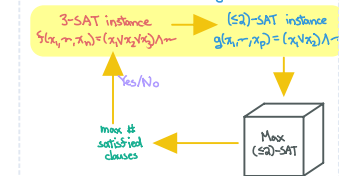
Fix a clause $(a \vee b \vee c)$
introduce new variable y

consider the 10 clauses

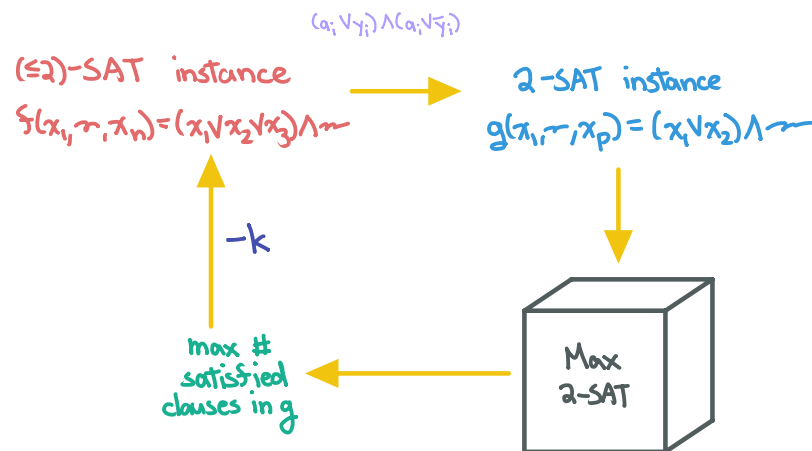
$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$
 $(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$

$(a, b, c) =$	(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

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.



Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}, T \in \mathbb{N}$
Goal: find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
T=52

Another "search" problem

	Polytime Algo	Hard as SAT (CNF)	Abstain
Subset Sum	10%	30%	○

★Dynamic Programming★

Recursive Spec
(Subset Sum)
SS(i, S) = TRUE if there is a subset of $x_1, \dots, x_n$ summing to S, and FALSE otherwise

Subset Sum
Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
T=52

Implementation

SS(i, S)
1. If $S < 0$ then return FALSE
2. If $S = 0$ then return TRUE
3. If $n = 0$ then return FALSE
4. Return $SS(i+1, S) \vee SS(i+1, S - x_i)$

Running time:
 $O(nT)$ subproblems
 $O(1)$ time per subproblem

 $O(nT)$ time overall

Theorem: A polytime algo for subset-sum implies a polytime algo for SAT

Subset Sum
Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
T=52

Literals $x = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ Long #'s $\{A_1, \bar{A}_1, \dots, A_m, \bar{A}_m\}$ and $\{B_1, \bar{B}_1, \dots, B_n, \bar{B}_n\}$
T = 33 m clauses 3 1 1 1 1 n variables
3-SAT instance $\phi(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$
Yes/No for $x_1, x_2, \dots, x_p \in \mathbb{N}$ T ∈ N
Subset Sum
Key idea: numbers are bit-strings too! use many digits each digit encodes 1 constraint

Key idea: numbers are bit-strings too!
use many digits each digit encodes 1 constraint

2. Start making #'s:
we use mtn digits base 4
for the literal x_j :

$A_j = \underbrace{0 \ 1 \ 1}_{m \text{ slots for each clause}} \underbrace{0 \ 0 \ 1}_{n \text{ slots for each variable}}$
(models) x_j

1. Recast 3SAT as "subset" problem
 $X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$
clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$
Goal: Find SCX st.
a) for each clause $C_i, C_i \cap S \neq \emptyset$
b) for each x_j, S contains 1 of $\{x_j, \bar{x}_j\}$ (not $\bar{x}_j$)

for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$, else 0
set " x_j th digit" to 1, else 0

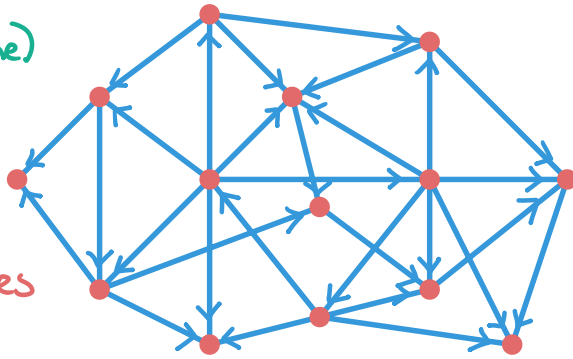
(vertices cannot repeat)

today: longest path problem

(eg. # edges)

find the (length of the) longest path in G

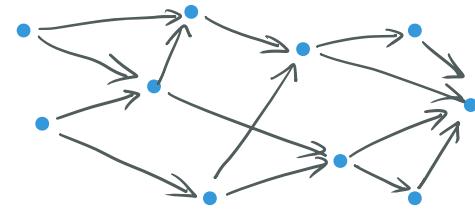
Hamiltonian path:
path visiting all vertices



does G have a Hamiltonian path?

related problems: longest walk, shortest paths

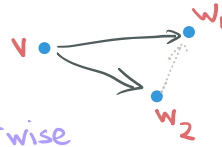
Longest walk in DAGs



Recursion

$LW(v)$ = length of Longest Walk from v

$$LW(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in E} LW(w) & \text{otherwise} \end{cases}$$



Running time w/ caching

$$\sum_{v \in V} (\text{time for } LW(v)) = \sum_{v \in V} 1 + \deg^+(v) = O(m+n)$$

Three settings:

1. DAG

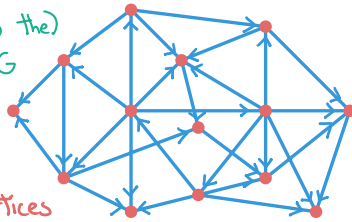
2. Directed

3. Undirected

today: longest path problem

find the (length of the) longest path in G

Hamiltonian path:
path visiting all vertices

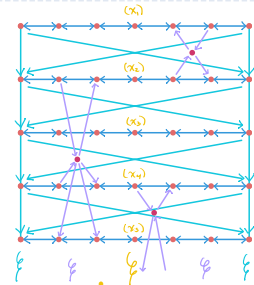


does G have a Hamiltonian path?

	Polytime Algo	Hard as SAT ^(CNF)	Abstain	Lions
DAG	90%	2		≥3
Directed	15%	85% - 1		
Undirected	0%	99%		

Theorem 7.1. A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.

$$\varphi(x_1, \dots, x_n) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \dots$$

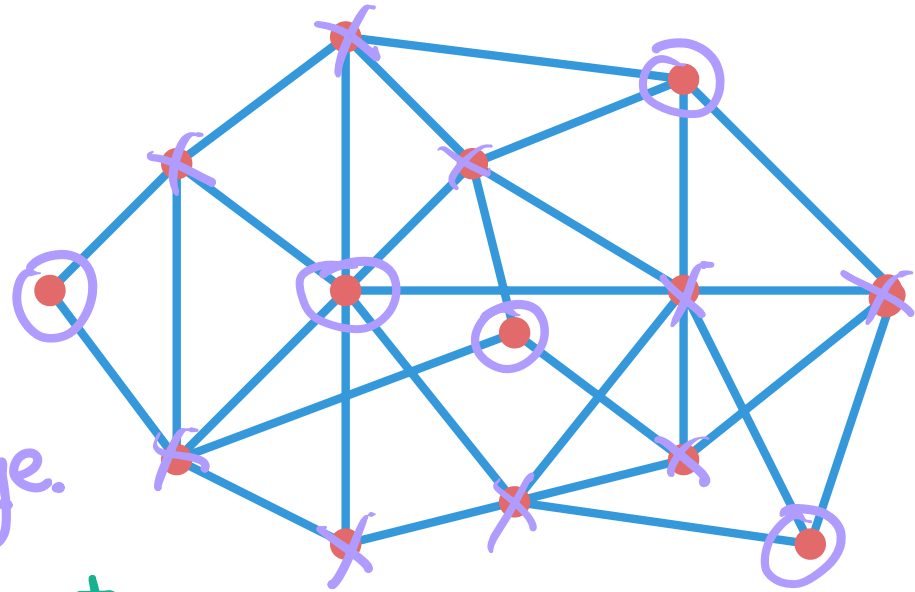


Independent set

Input: undirected $G=(V,E)$
(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$

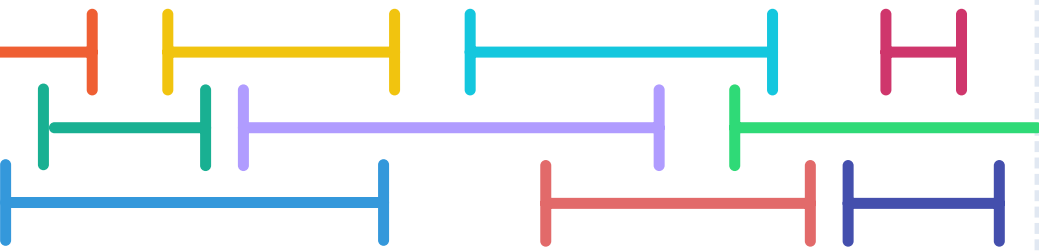
$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.

Goal: max weight independent set
(decision version: is there an independent set of weight $\geq k$?)



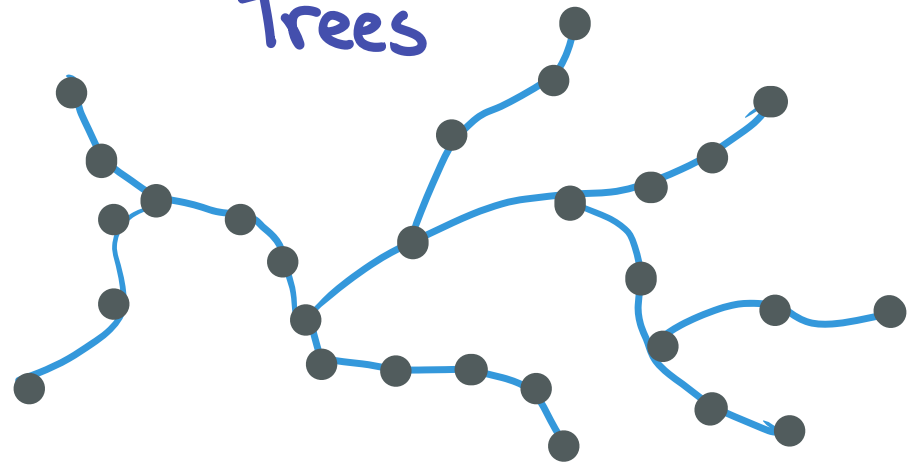
Special cases:

intervals



"independent" = non-overlapping

trees



Independent set

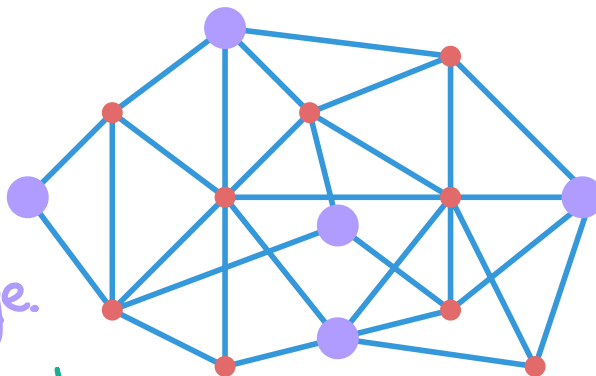
Input: undirected $G=(V,E)$

(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$

$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.

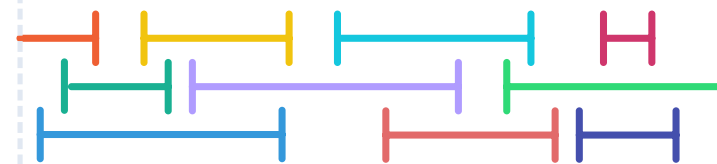
Goal: max weight independent set

(decision version: is there an independent set of weight $\geq k$?)



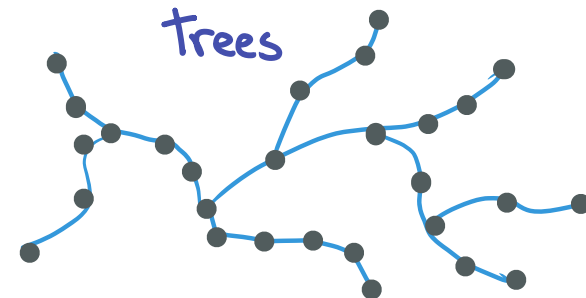
Special cases:

intervals



"independent" = non-overlapping

trees



Independent set in { graphs, intervals, trees }

Hard as SAT Polytime algo

14	0
5	9
1	13

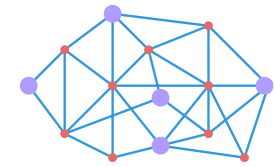
Abstain

0

Theorem: A polytime algo for max
(unweighted)
cardinality independent set implies a
polytime algo for SAT

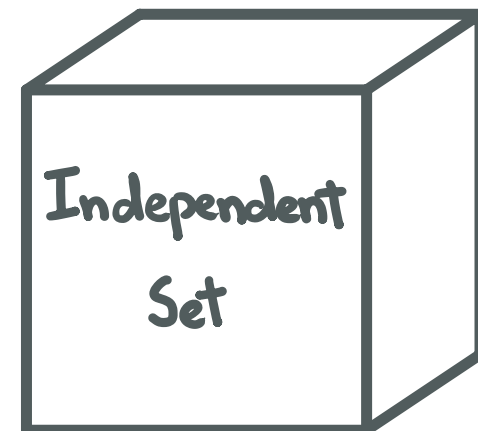
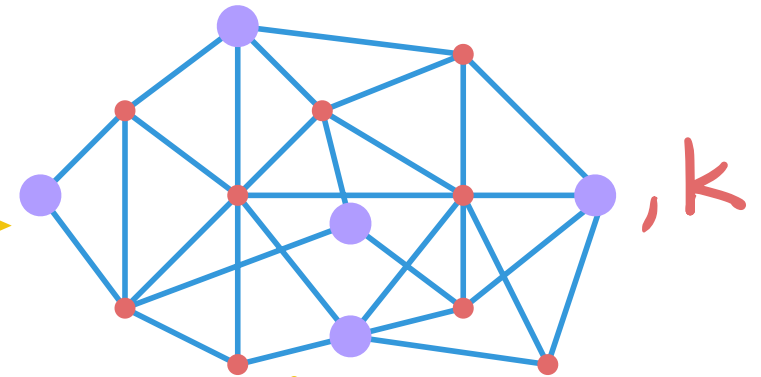
Proof setup

Independent set
Input: undirected $G=(V,E)$
(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$



$S \subseteq V$ is independent if no
 $u,v \in S$ are connected by an edge.
Goal: max weight independent set
(decision version: is there an independent set of weight $\geq k$?)

3-SAT instance
 $\{ (x_1, \neg x_2, x_3) \wedge \dots \}$

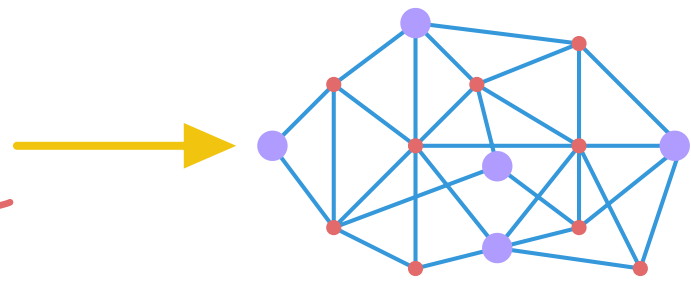


There is/isn't an
independent set
of size k



Goal: transform

3-SAT instance
 $f(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$



st. size of max indep. set informs satisfiability of f

e.g.

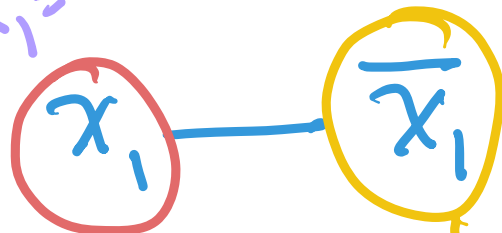
$(f \text{ is satisfiable}) \Leftrightarrow (\text{there is indep. set of size } k)$

(for some k TBD)

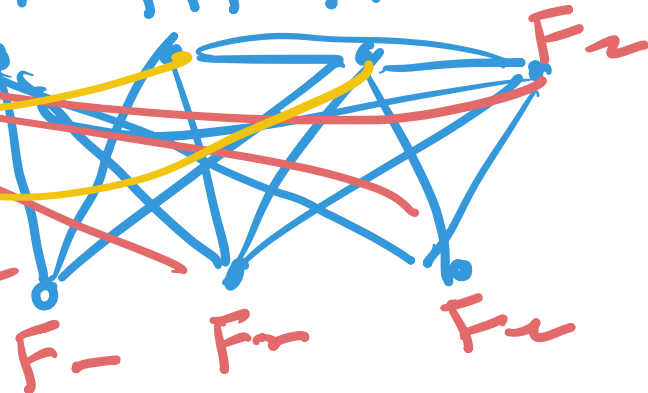
ideas?

$f(_)$
 m clauses
 n variables

$x_1 = \text{true}$



TTT TTF TFT



$2n + 7m$ vertices

$\uparrow$
var.

$\uparrow$
ways to satisfy each clause

clause gadgets

SAT

Graphs

literals
(in clauses)



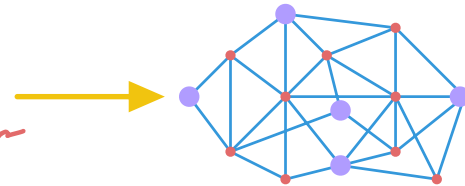
vertices

clauses



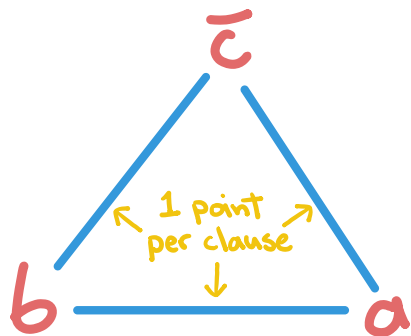
triangles

3-SAT instance
 $\mathcal{S}(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge \dots$

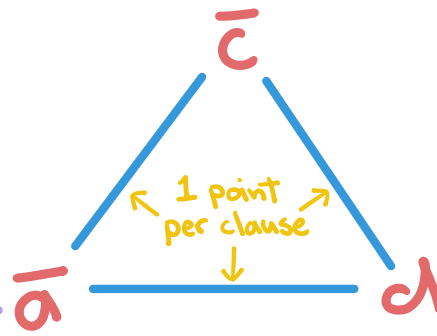


$(\mathcal{S} \text{ is satisfiable}) \Leftrightarrow (\text{there is indep. set of size } k)$
(for some k TBD)

$a \vee b \vee \bar{c}$

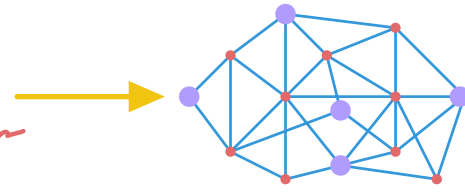


$\bar{a} \vee \bar{c} \vee d$



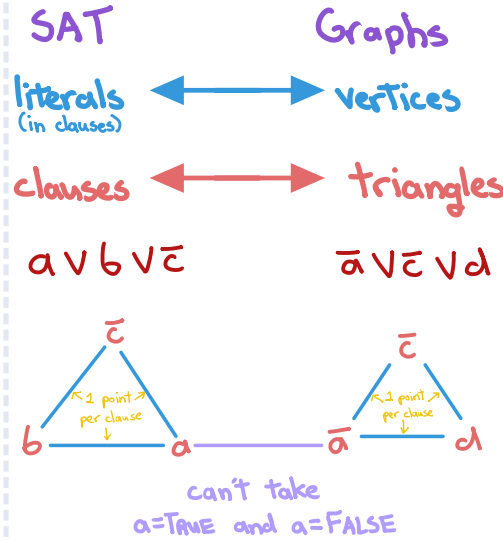
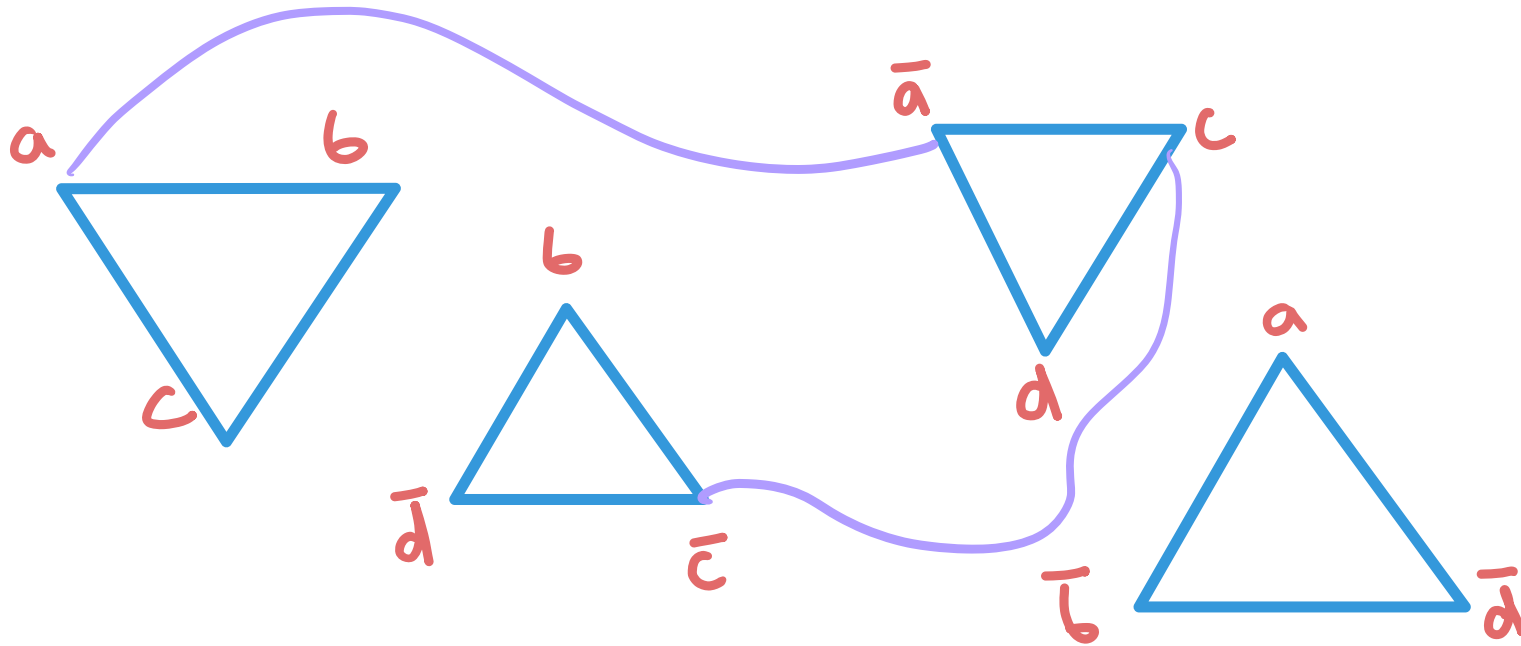
can't take
 $a = \text{TRUE}$ and $a = \text{FALSE}$

3-SAT instance
 $\mathcal{S}(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge \dots$



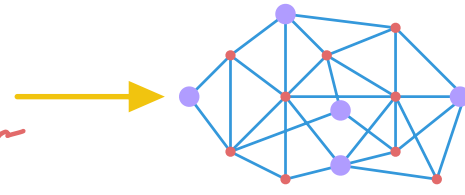
$(\mathcal{S} \text{ is satisfiable}) \iff (\text{there is indep. set of size } k)$
 (for some k TBD)

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{a}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$



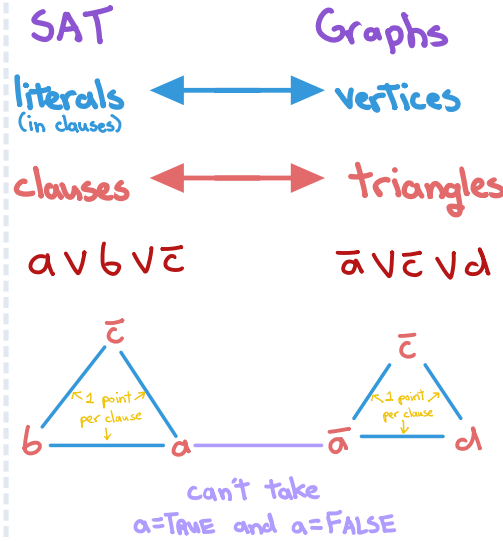
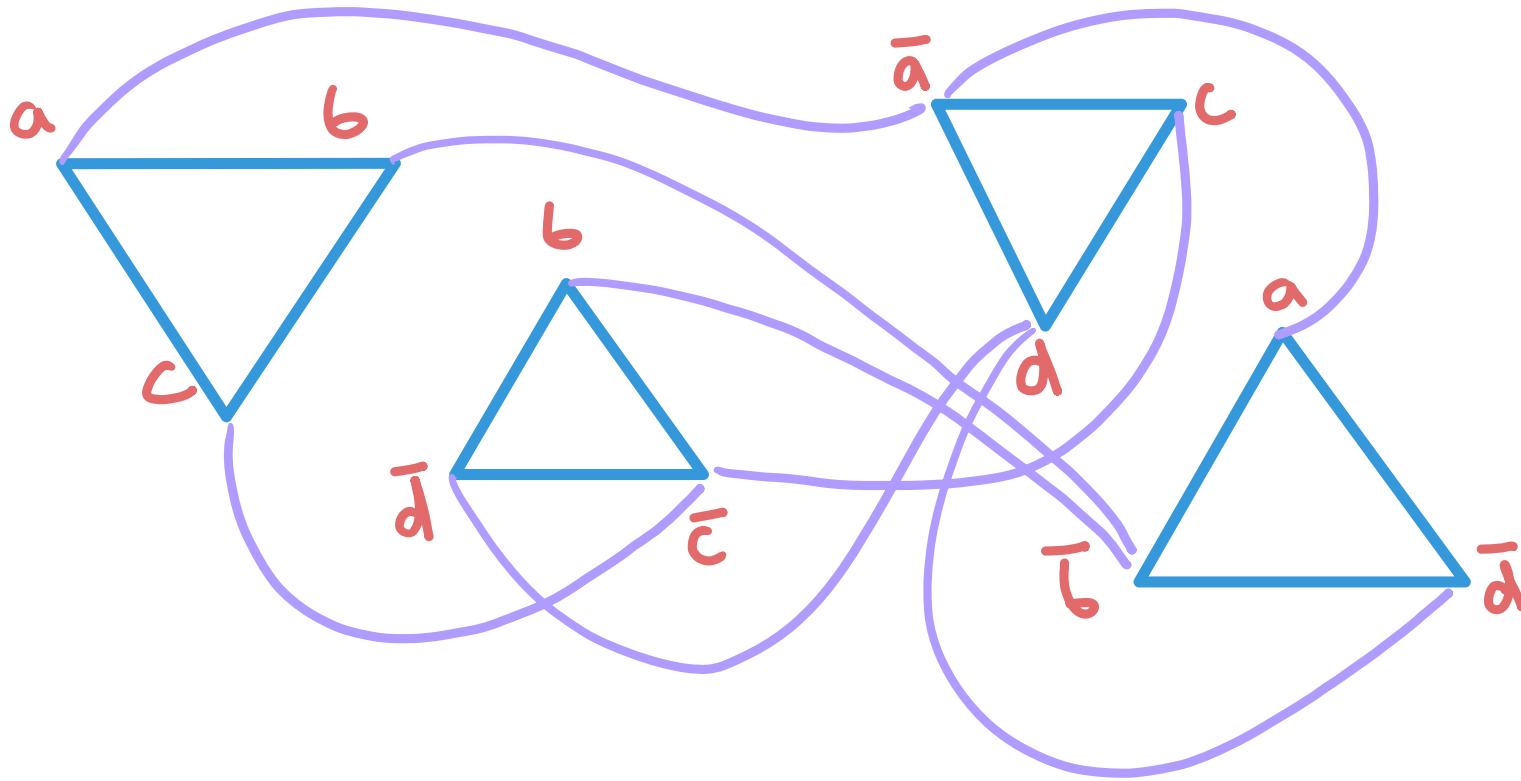
m clauses $\rightarrow m$ triangles

3-SAT instance
 $\mathcal{F}(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge \dots$



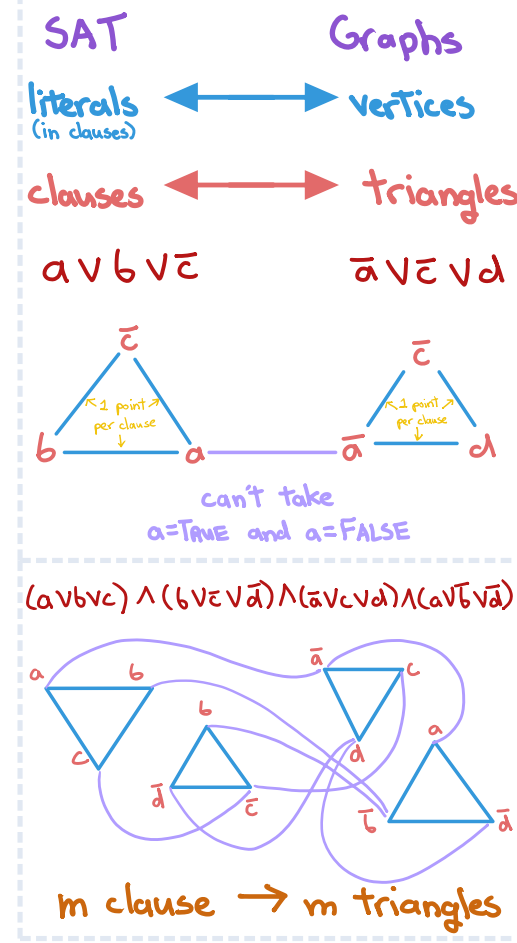
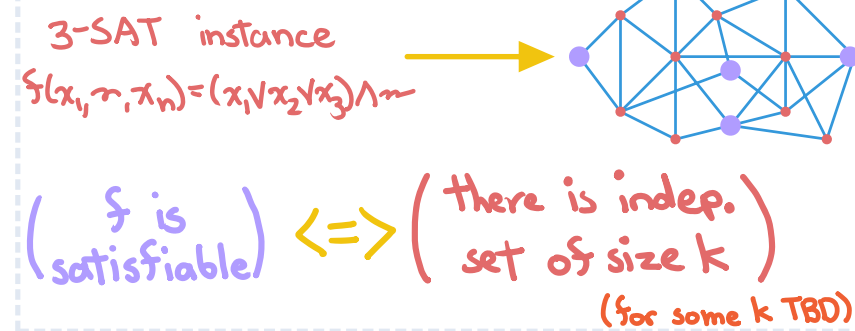
$(\mathcal{F} \text{ is satisfiable}) \iff (\text{there is indep. set of size } k)$
 (for some k TBD)

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$



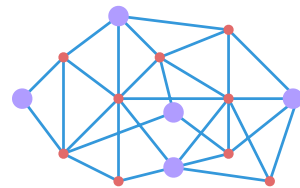
$m \text{ clauses} \rightarrow m \text{ triangles}$

claim: $\mathcal{F}$ is satisfiable iff there is an independent set of size m
 (where m is the # of clauses)



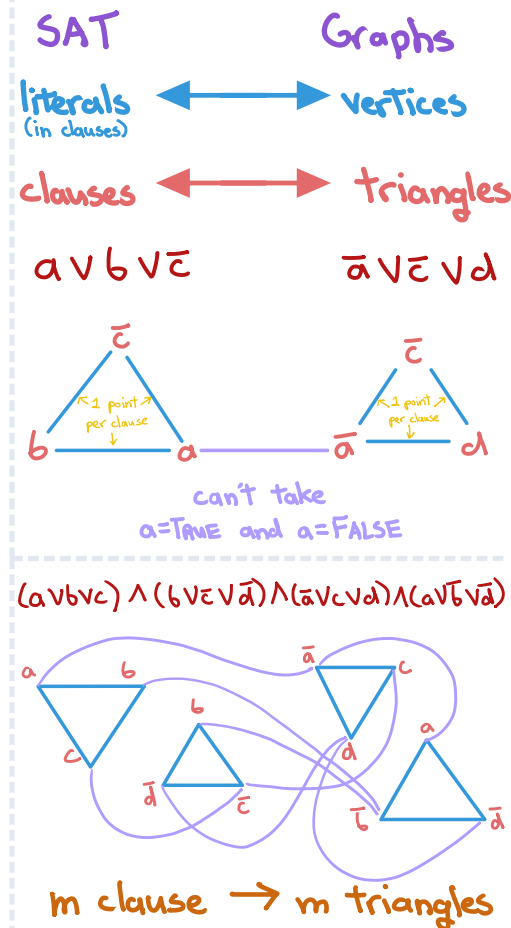
claim: $\mathcal{F}$ is satisfiable iff there
 is an independent set of size m
 (where m is the # of clauses)

3-SAT instance
 $\mathcal{F}(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge \dots$



$(\mathcal{F} \text{ is satisfiable}) \iff (\text{there is indep. set of size } k)$
 (for some k TBD)

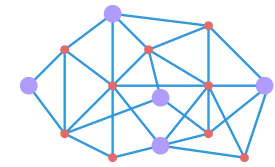
Proof. ($\Leftarrow$) Suppose there is an independent set
 of size k



Theorem: A polytime algo for max
(unweighted)
cardinality independent set implies a
polytime algo for SAT

Proof setup

Independent set
Input: undirected $G=(V,E)$
(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$



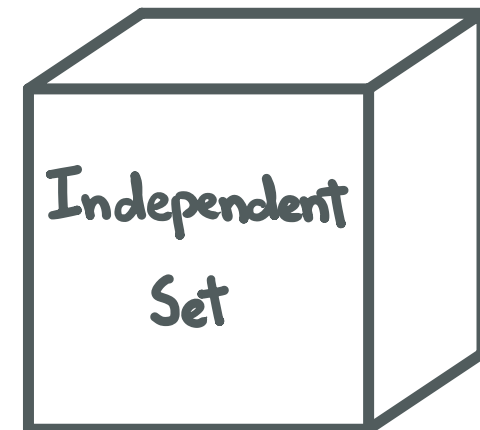
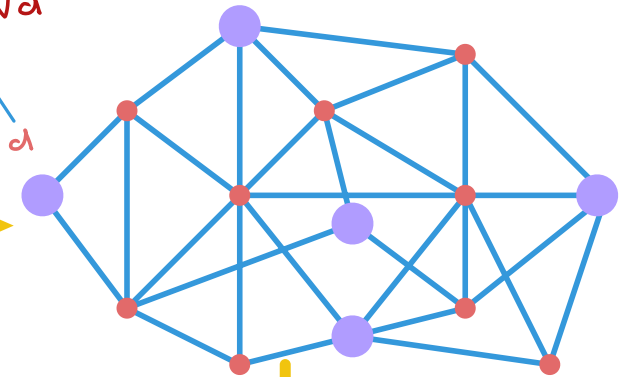
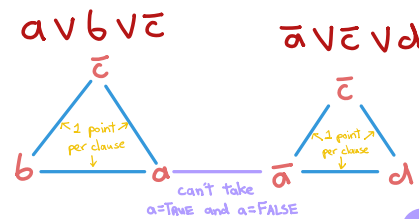
$S \subseteq V$ is independent if no
 $u,v \in S$ are connected by an edge.
Goal: max weight independent set
(decision version: is there an independent set of weight $\geq k$?)

3-SAT instance

$$\mathcal{F}(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$$

claim: $\mathcal{F}$ is satisfiable iff there
is an independent set of size m
(where m is the # of clauses)

there is/isn't an
independent set
of size m



Independent set

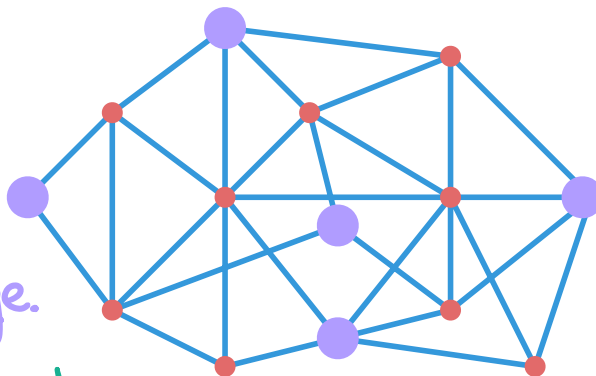
Input: undirected $G=(V,E)$

(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$

$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.

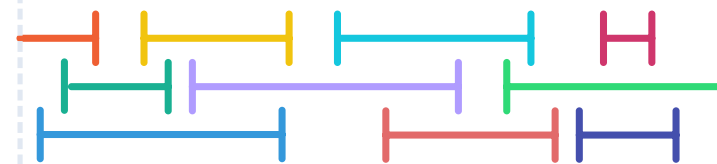
Goal: max weight independent set

(decision version: is there an independent set of weight $\geq k$?)



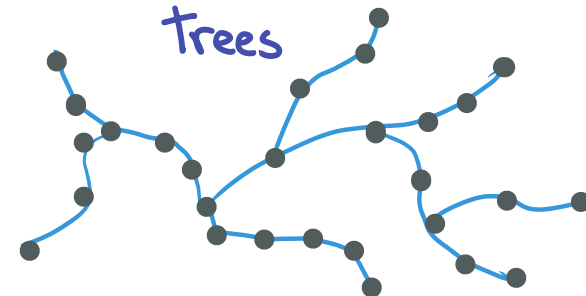
Special cases:

intervals



"independent" = non-overlapping

trees



Independent set in { graphs, intervals, trees }

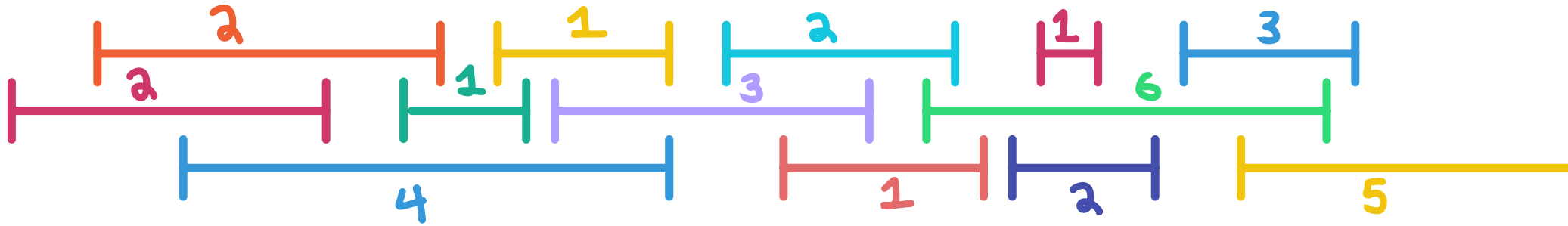
Hard as SAT Polytime algo

	graphs		✓	
	intervals			
	trees			

Abstain

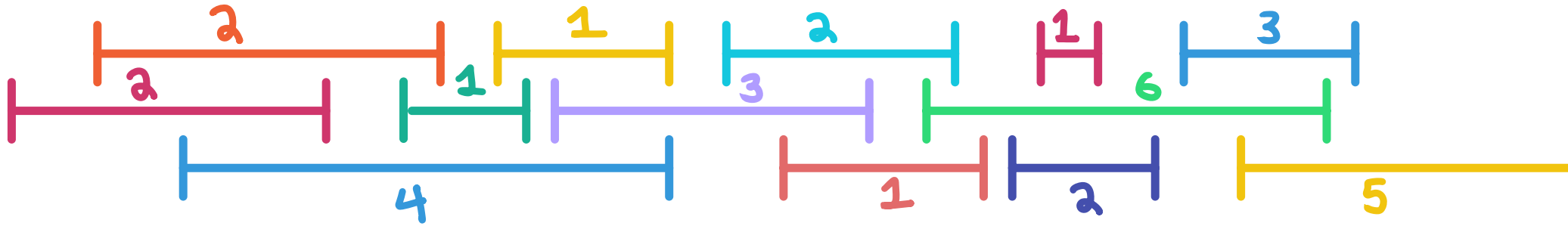
Maximum weight independent set of intervals

("independent" = non-overlapping)



ideas?

Maximum weight independent set of intervals
 ("independent" = non-overlapping)



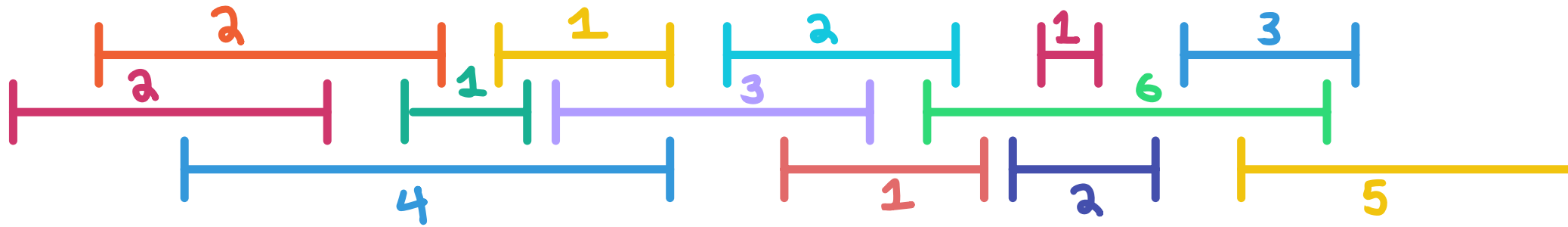
Recursive spec: for a family of weighted intervals I ,

$MWIS(I) = \max$ weight of any independent subset of I

$I' =$ intervals disjoint from I

Maximum weight independent set of intervals

("independent" = non-overlapping)



Recursive spec: for a family of weighted intervals I ,
 $MWIS(I) = \max \text{ weight of any independent subset of } I$

Implementation:

$MWIS(I)$

- 1 if $I = \emptyset$ return 0.
2. let $I \in I$ be any interval.
3. return max of
 - a. $MWIS(I \setminus \{I\})$
 - b. $\text{weight}(I) + MWIS(I')$ w/ $I' = \text{intervals disjoint from } I$

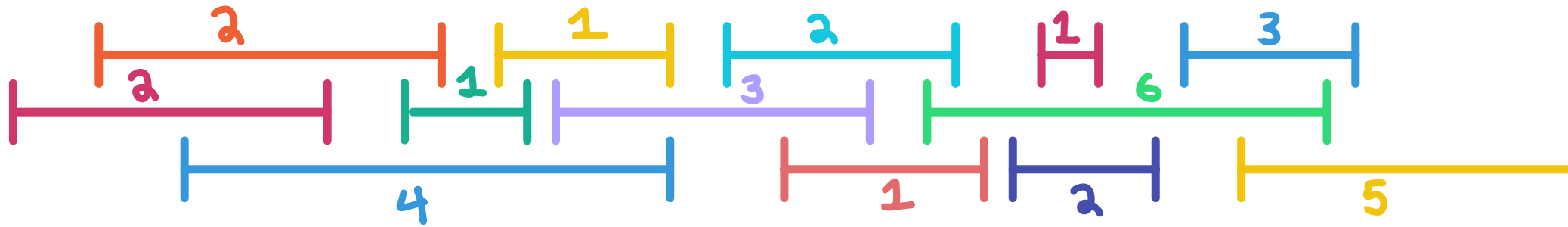
Running time

$$T(n) = 2T(n-1) + O(1) = 2^n$$

how to make this faster?

Maximum weight independent set of intervals

("independent" = non-overlapping)



Recursive spec: for a family of weighted intervals I ,
 $MWIS(I) = \text{max weight of any independent subset of } I$

Implementation:

$MWIS(I)$

1. if $I = \emptyset$ return 0.

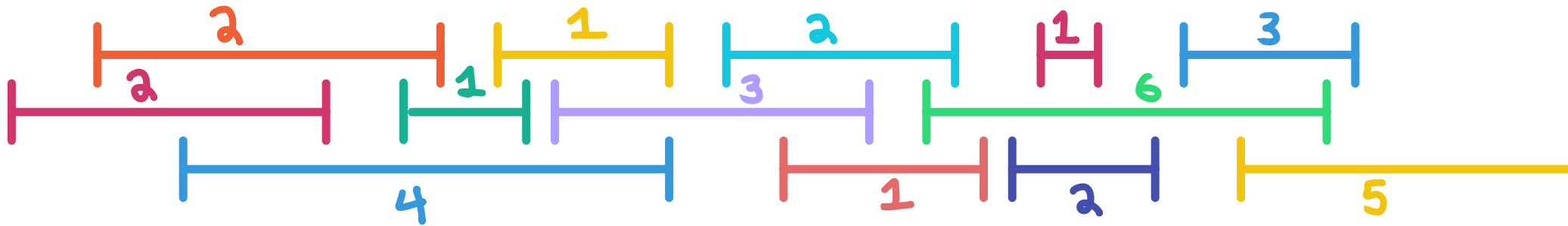
2. let $I \in I$ be any interval.

3. return max of

a. $MWIS(I \setminus \{I\})$

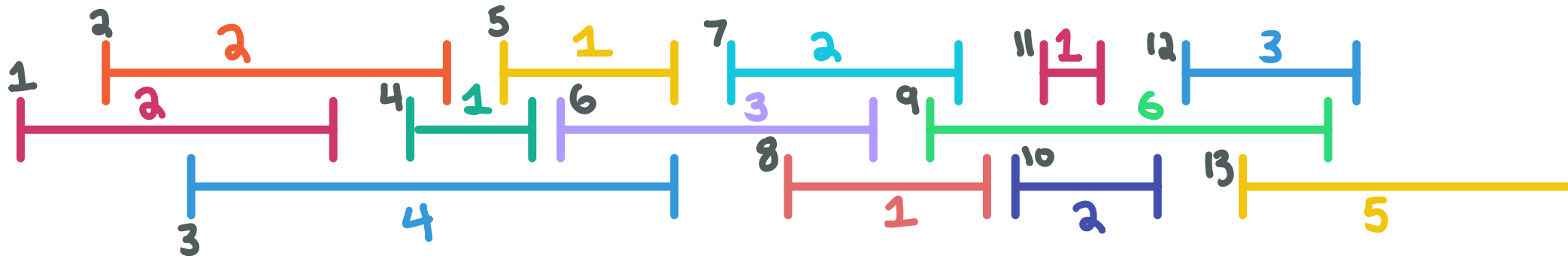
b. $\text{weight}(I) + MWIS(I')$ w/ $I' = \text{intervals disjoint from } I$

Idea: make problem "sequential" by sorting intervals



let $I_1=[a_1, b_1]$, $I_2=[a_2, b_2]$, ..., $I_n=[a_n, b_n]$ number the intervals in increasing order of left endpoint a_i

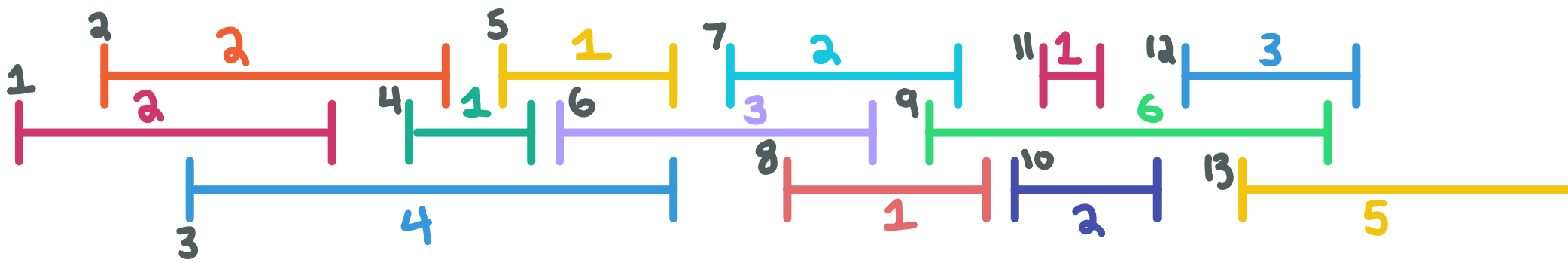
Idea: make problem "sequential" by sorting intervals



let $I_1=[a_1, b_1]$, $I_2=[a_2, b_2]$, ..., $I_n=[a_n, b_n]$ number the intervals in increasing order of left endpoint a_i

Recursive spec:

$MWIS(k) = \max \text{ weight of any indep. subset of } I_k, \dots, I_n$



let $I_1=[a_1, b_1]$, $I_2=[a_2, b_2]$, ..., $I_n=[a_n, b_n]$ number the intervals in increasing order of left endpoint a_i

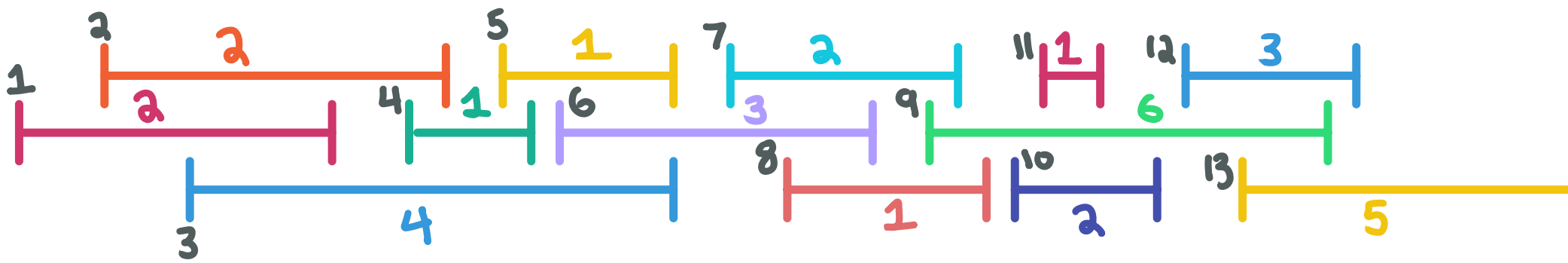
Recursive spec:

$MWIS(k)$ = max weight of any indep. subset of $I_k, \dots, I_n$

Implementation:

$MWIS(k) =$

$\begin{cases} 0 & k > n \\ \max \begin{cases} w(I_k) + MWIS(l) & \text{where } l \text{ first index } > k \text{ s.t. } a_l > b_k \\ MWIS(k+1) \end{cases} \end{cases}$



let $I_1=[a_1, b_1]$, $I_2=[a_2, b_2]$, ..., $I_n=[a_n, b_n]$ number the intervals in increasing order of left endpoint a_i

Recursive spec:

$MWIS(k) = \max$ weight of any indep. subset of $I_k, \dots, I_n$

Implementation:

$MWIS(k)$

0 if $k > n$

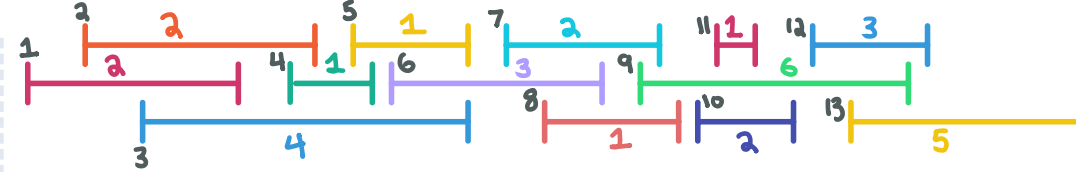
let $l > k$ be index of first interval disjoint from I_k

$\max \begin{cases} MWIS(k+1) \\ \text{weight}(I_k) + MWIS(l) \end{cases}$ otherwise

Recursive spec:

$MWIS(k) = \text{max weight}$

of any indep. subset of $I_k, \dots, I_n$



let $I_1=[a_1, b_1]$, $I_2=[a_2, b_2]$, ..., $I_n=[a_n, b_n]$ number the intervals in increasing order of left endpoint a_i

Implementation:

$MWIS(k)$

0 if $k > n$

let $l > k$ be index of first interval disjoint from I_k

$\max \begin{cases} MWIS(k+1) \\ \text{weight}(I_k) + MWIS(l) \end{cases}$ otherwise

Running time (w/ caching)

n problems

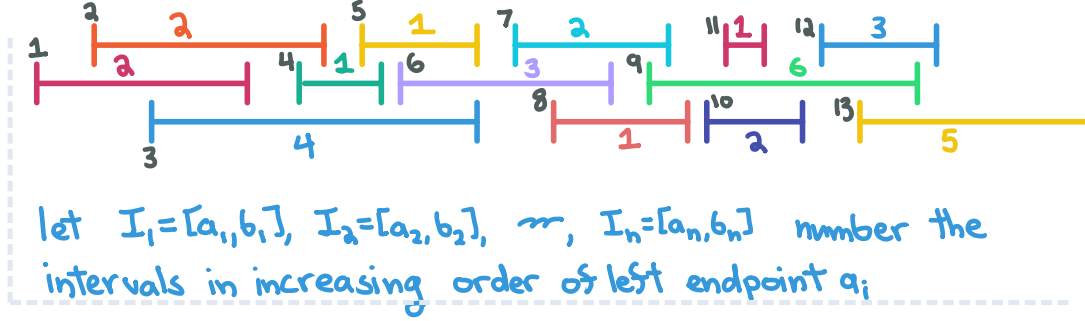
$\log n$ time each

$\Rightarrow n \log n$

$n \log n$

Recursive spec:

$MWIS(k) = \text{max weight}$
of any indep. subset of $I_k, \dots, I_n$



Implementation:

$MWIS(k)$

0 if $k > n$

↙ binary search!

let $l > k$ be index of first interval disjoint from I_k

$\max \begin{cases} MWIS(k+1) \\ \text{weight}(I_k) + MWIS(l) \end{cases}$ otherwise

Running time (w/ caching)

$O(n \log n)$ to sort

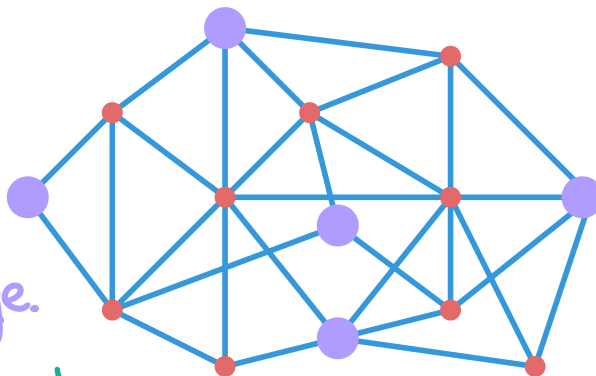
$O(n)$ subproblems $\times O(\log n)$ time per problem

Independent set

Input: undirected $G=(V,E)$
(optionally) weights $w: V \rightarrow \mathbb{R}_{\geq 0}$

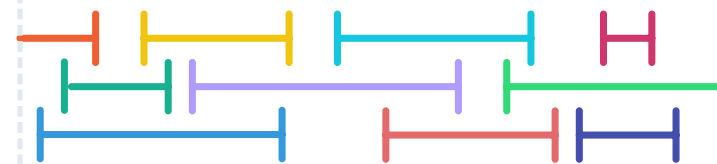
$S \subseteq V$ is independent if no
 $u,v \in S$ are connected by an edge.

Goal: max weight independent set
(decision version: is there an independent set of weight $\geq k$?)



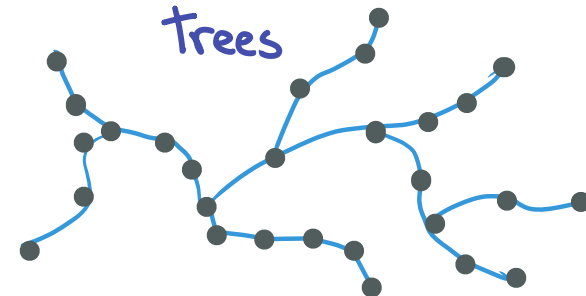
Special cases:

intervals



"independent" = non-overlapping

trees



Independent set in { graphs, intervals, trees }

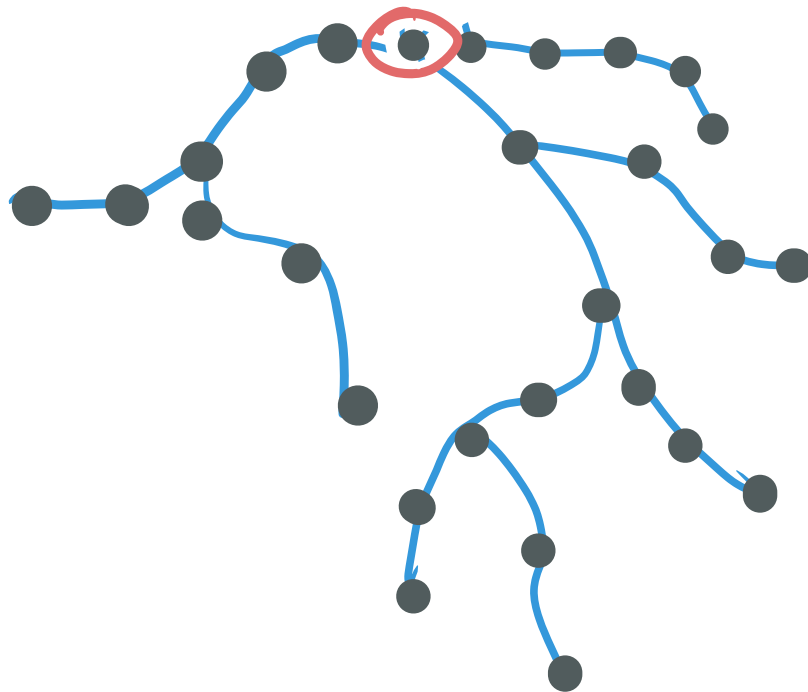
Hard as SAT Polytime algo

✓	
	✓

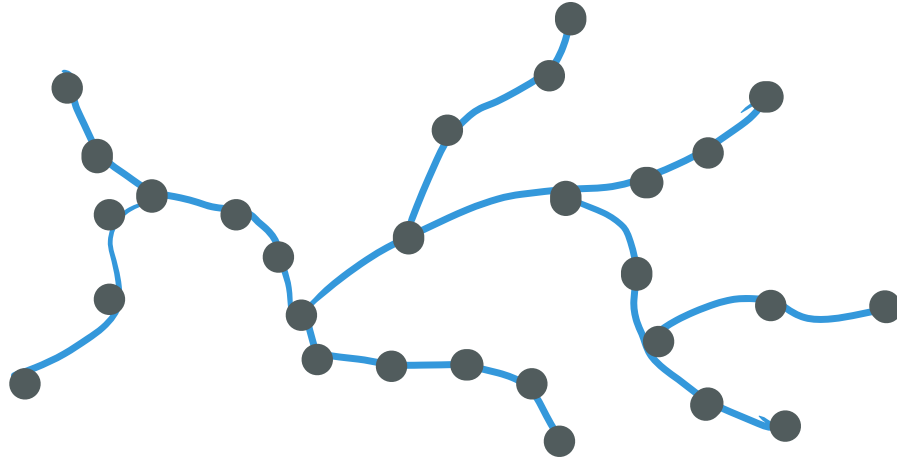
Abstain

MWIS in trees

ideas?



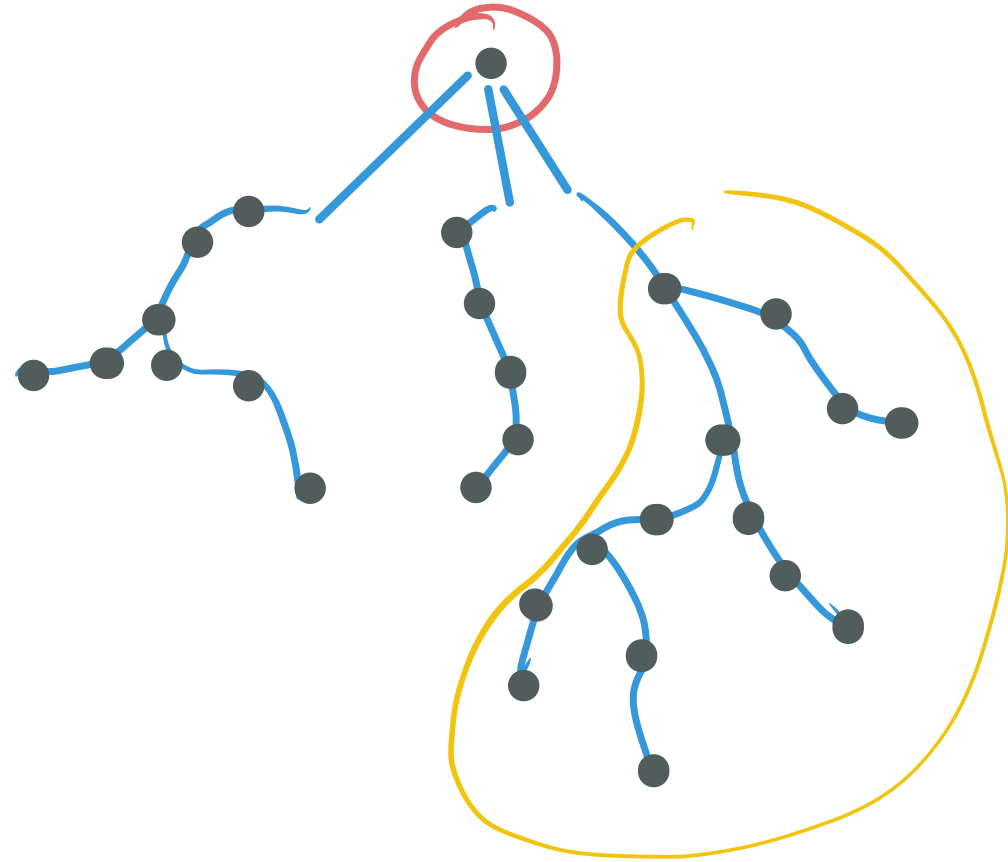
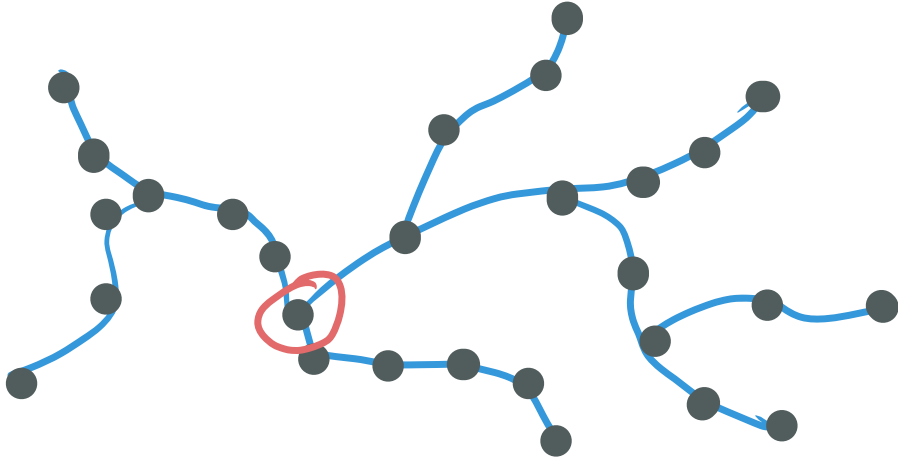
MWIS in trees



ideas?

how to order subproblems?

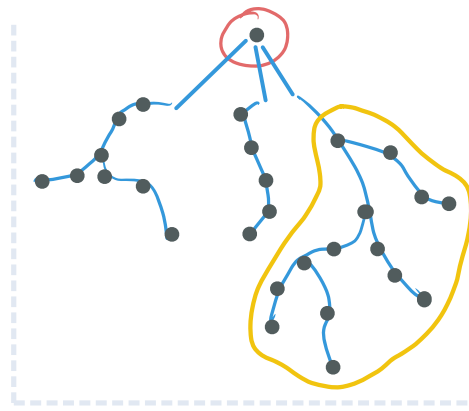
Root it!



Subtree
(= subproblem)

Recursive Spec: root T (arbitrariness)

$MWIS(v)$: ^{weight of} $MWIS$ of subtree rooted at v

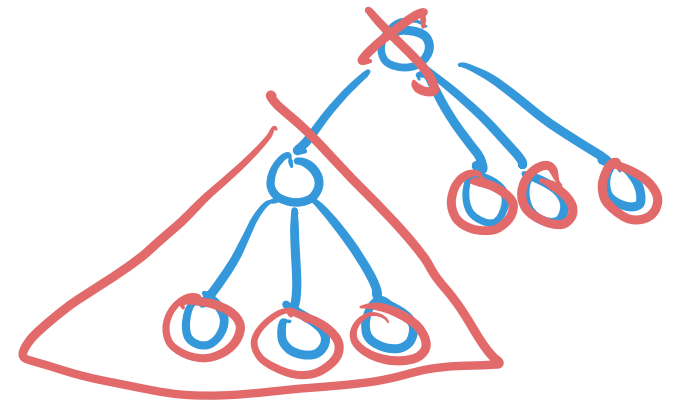
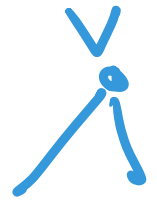


Recursive implementation

$MWIS(v)$:

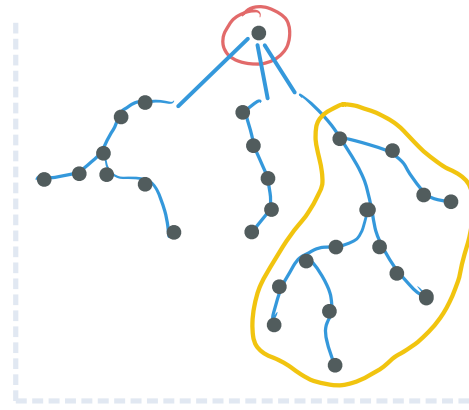
if v is a leaf, return $w(v)$

$$\text{max} \left\{ \begin{array}{l} \sum_{\text{child } x \text{ of } v} MWIS(x) \\ w(v) + \sum_{\text{grandchildren } x \text{ of } v} MWIS(x) \end{array} \right.$$



Recursive Spec:

$MWIS(v)$ = MWIS of the subtree rooted at v

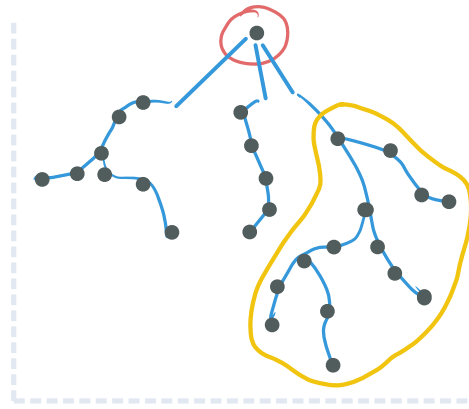


Recursive implementation

Recursive Spec:

$MWIS(v, \text{exclude-root}^{(\text{flag})}) =$

MWIS of the subtree rooted at v , excluding v if exclude-root is true.



Recursive implementation

$MWIS(v, \text{exclude-root}) =$

if exclude-root then return sum of $MWIS(x, \text{false})$ over all children x of v

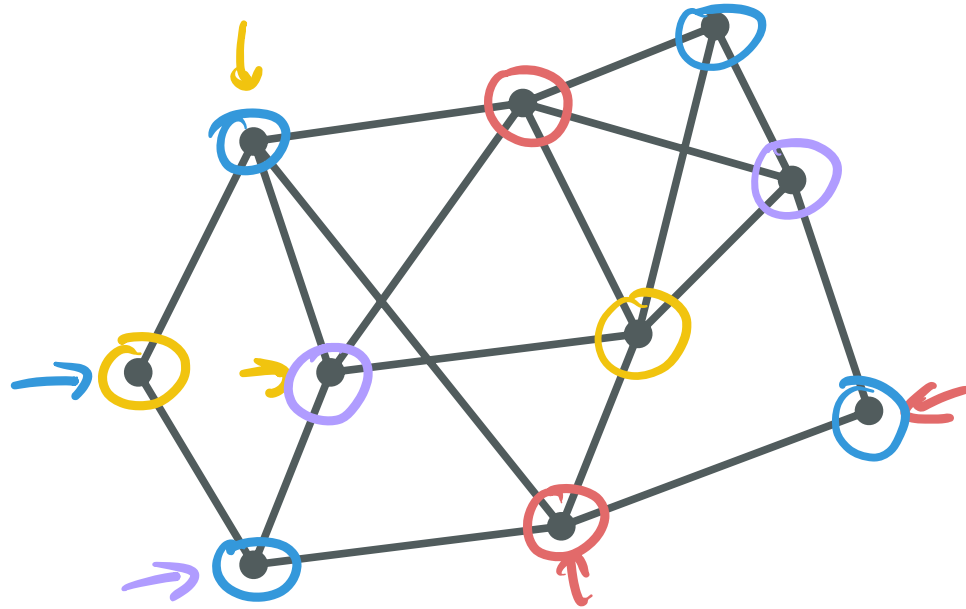
else return max of $MWIS(v, \text{true})$

and $\text{weight}(v) + \sum_{\text{children } x} MWIS(x, \text{true})$

Running time

$$\begin{aligned} & \sum_v \left(\text{time for } MWIS(v, T/F) \right) \\ &= \sum_v O(\# \text{ children of } v) \\ &= O(n) \end{aligned}$$

Graph Coloring



Goal: color vertices so that no adjacent vertices have the same color, w/ minimum # of colors.

decision version: does k colors suffice?

In "k-coloring problem", k is a fixed constant

Theorem: 3-coloring is hard as SAT

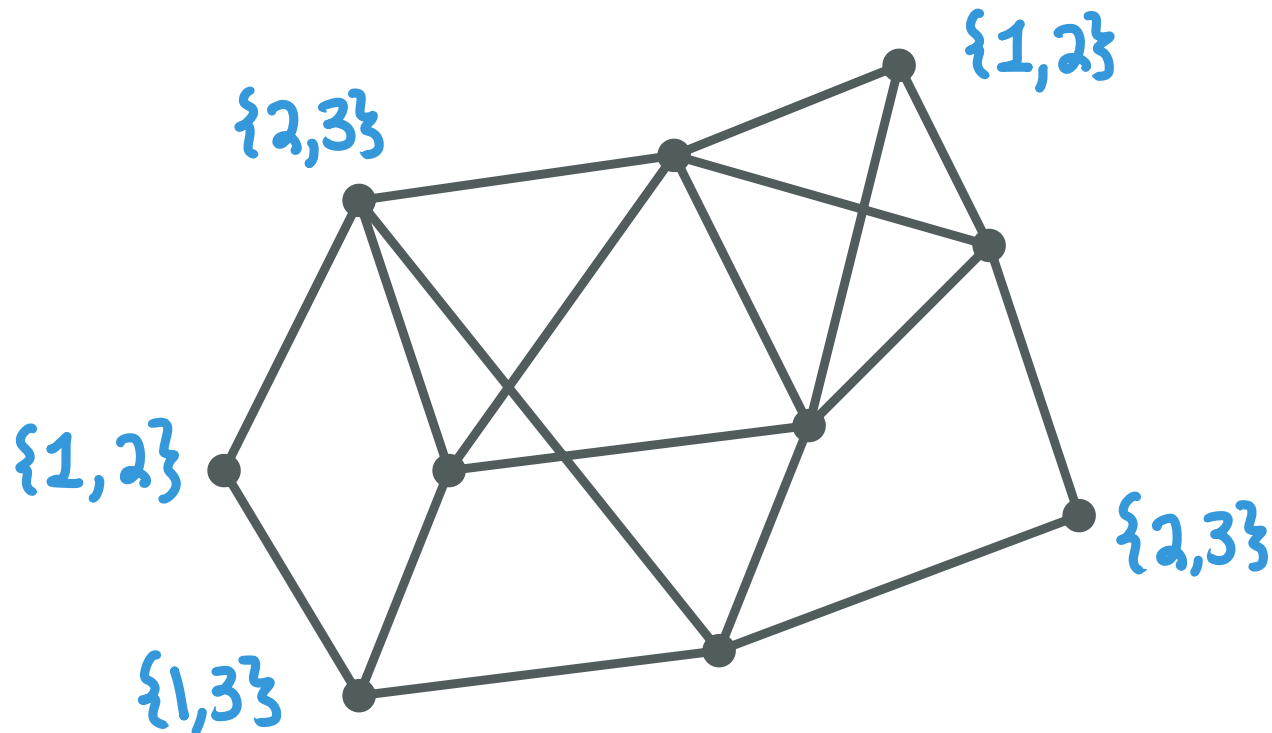
Theorem: 3-coloring is hard as SAT

High-level approach:

3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .

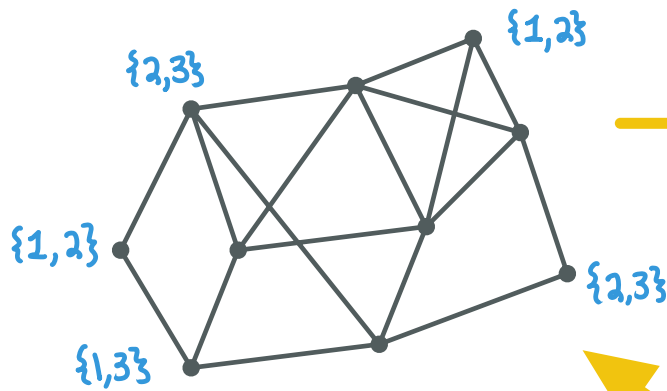
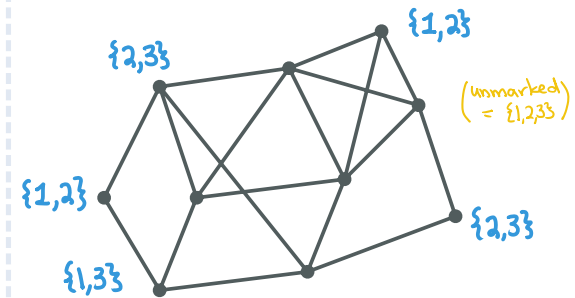


(unmarked)
= $\{1, 2, 3\}$

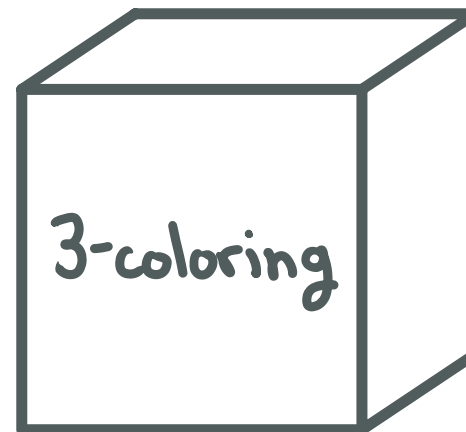
3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$
of colors allowed for v .



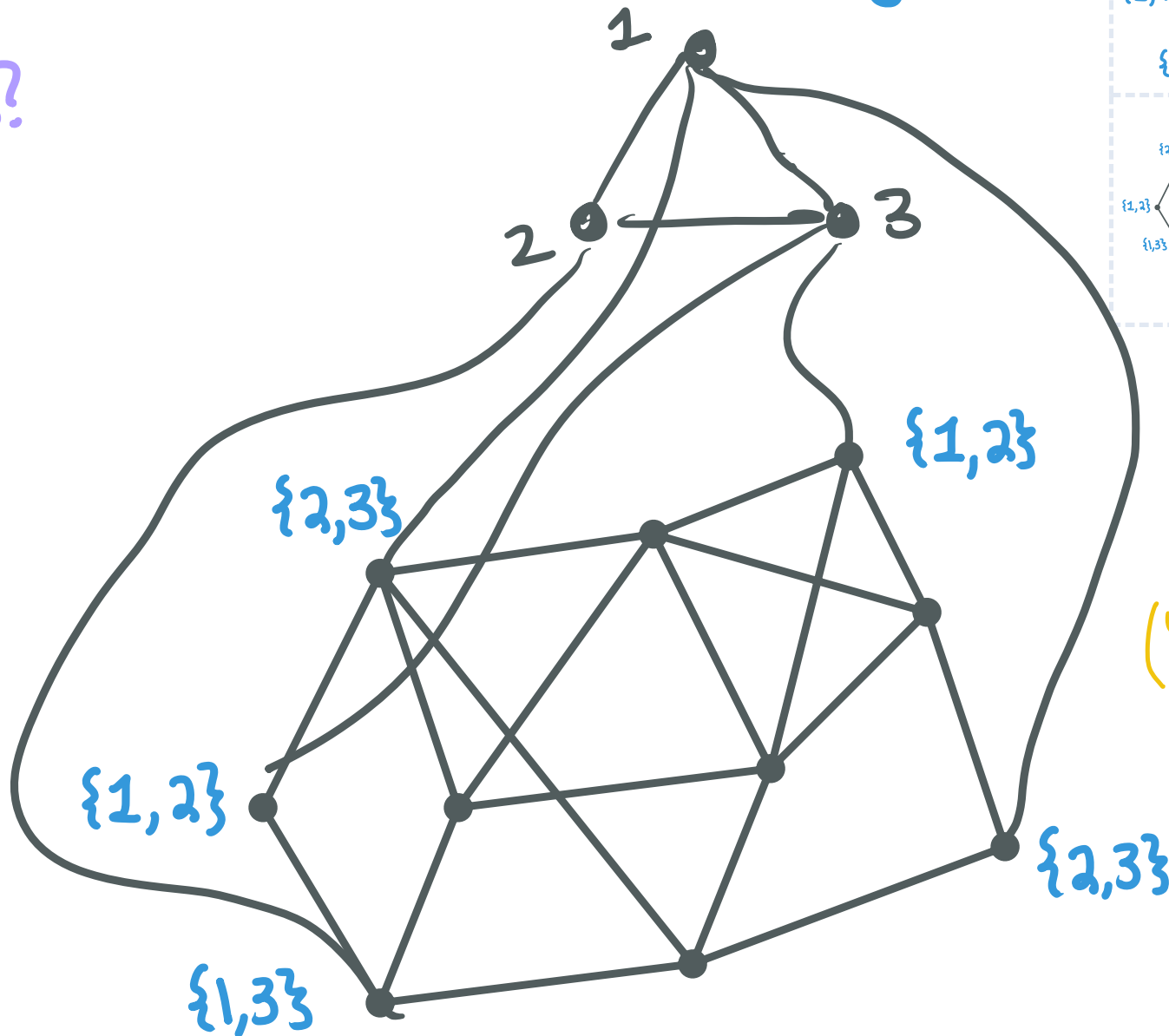
Yes/no



3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

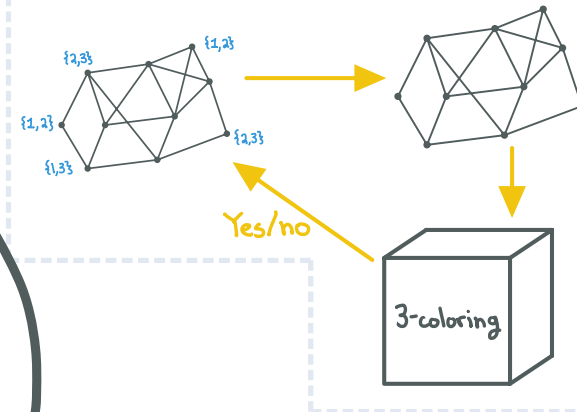
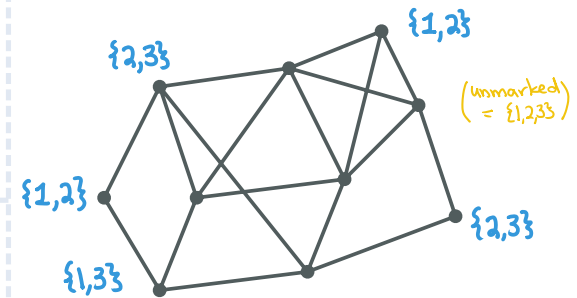
claim: polytime algo for 3-coloring implies
polytime algo for subset 3-coloring.

ideas?



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1,2,3\}$
of colors allowed for v .

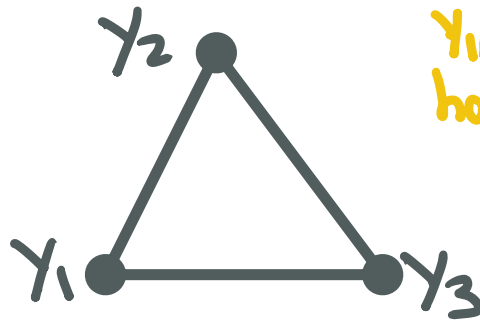


(unmarked)
= $\{1,2,3\}$

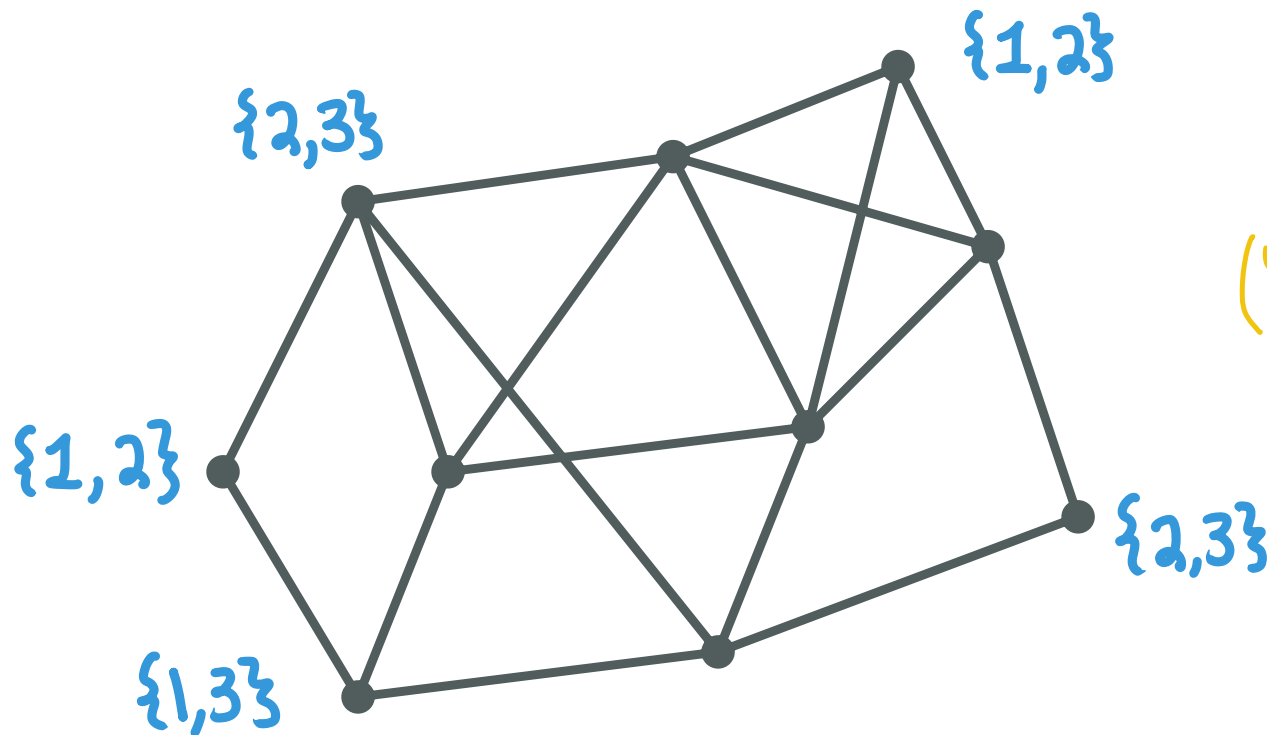
3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

claim: polytime algo for 3-coloring implies
polytime algo for subset 3-coloring.

"color palette"

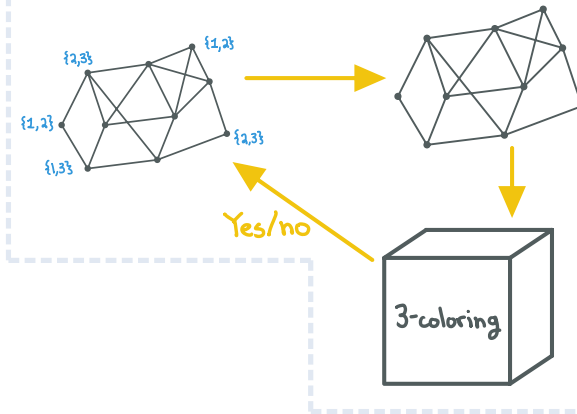
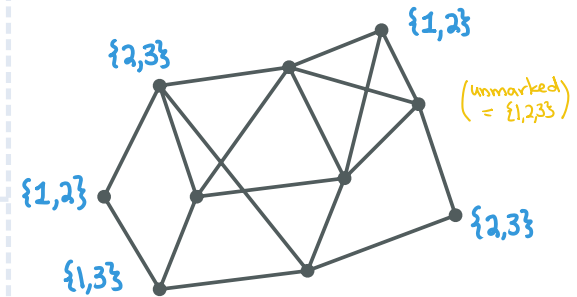


$\gamma_1, \gamma_2, \gamma_3$ must
have distinct
colors



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1,2,3\}$
of colors allowed for v .

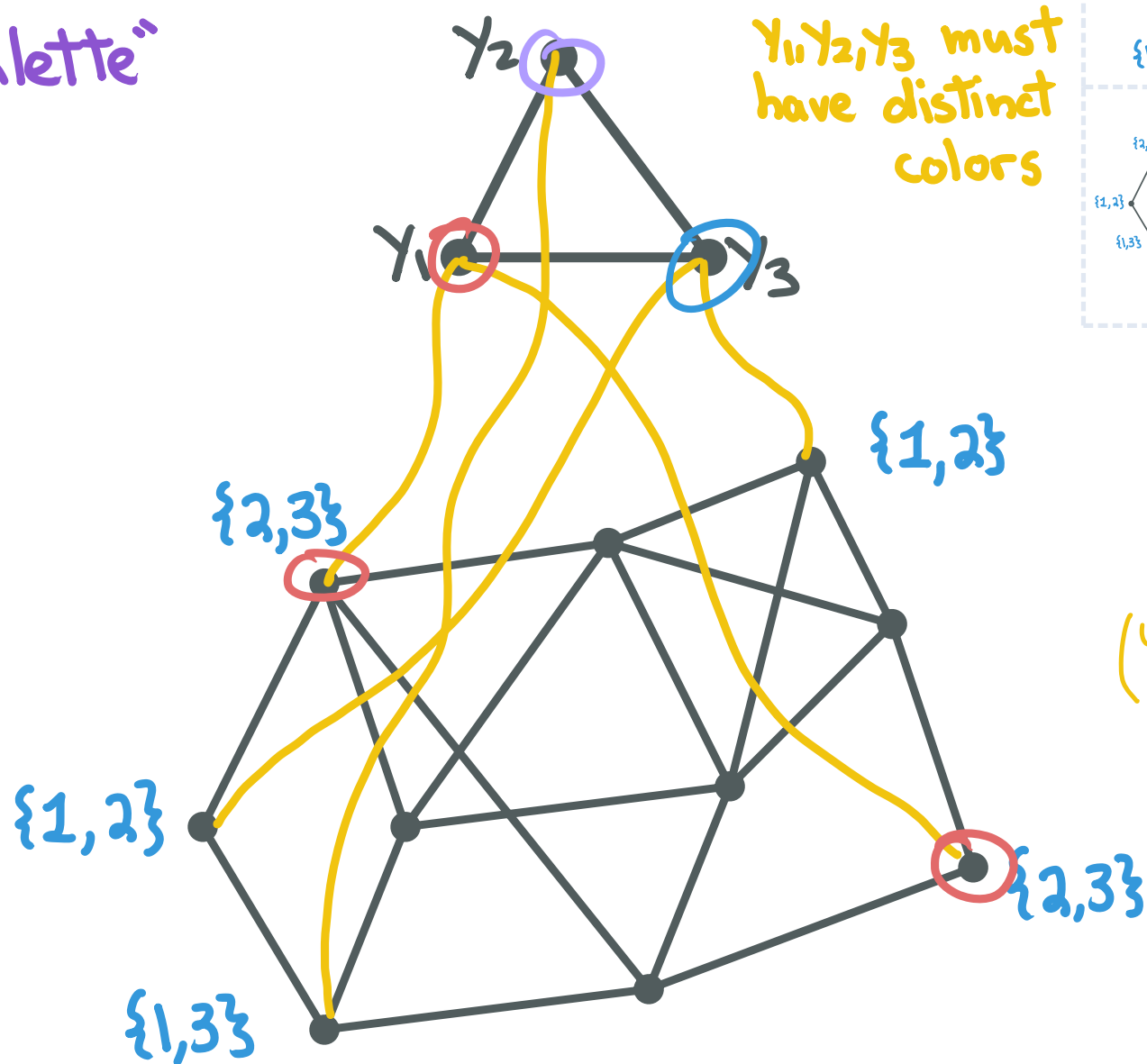


(unmarked)
= $\{1,2,3\}$

3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

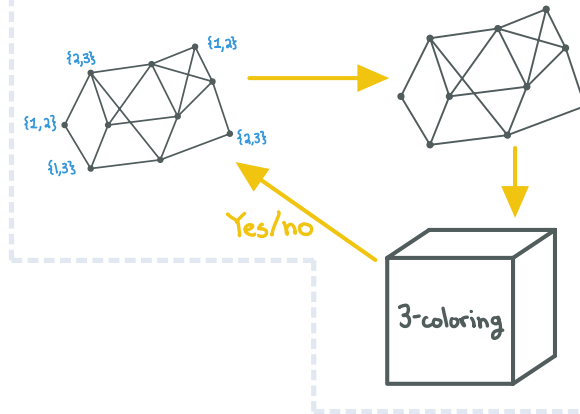
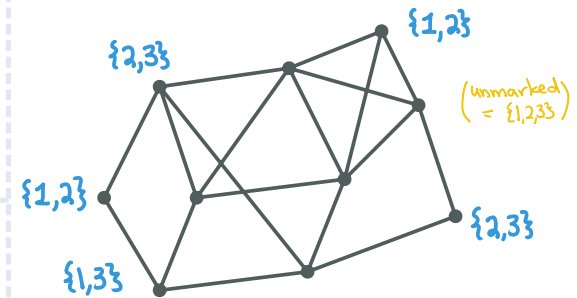
claim: polytime algo for 3-coloring implies
polytime algo for subset 3-coloring.

"color palette"



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1,2,3\}$
of colors allowed for v .



(unmarked)
= $\{1,2,3\}$

3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

claim: polytime algo for 3-coloring implies
polytime algo for subset 3-coloring.

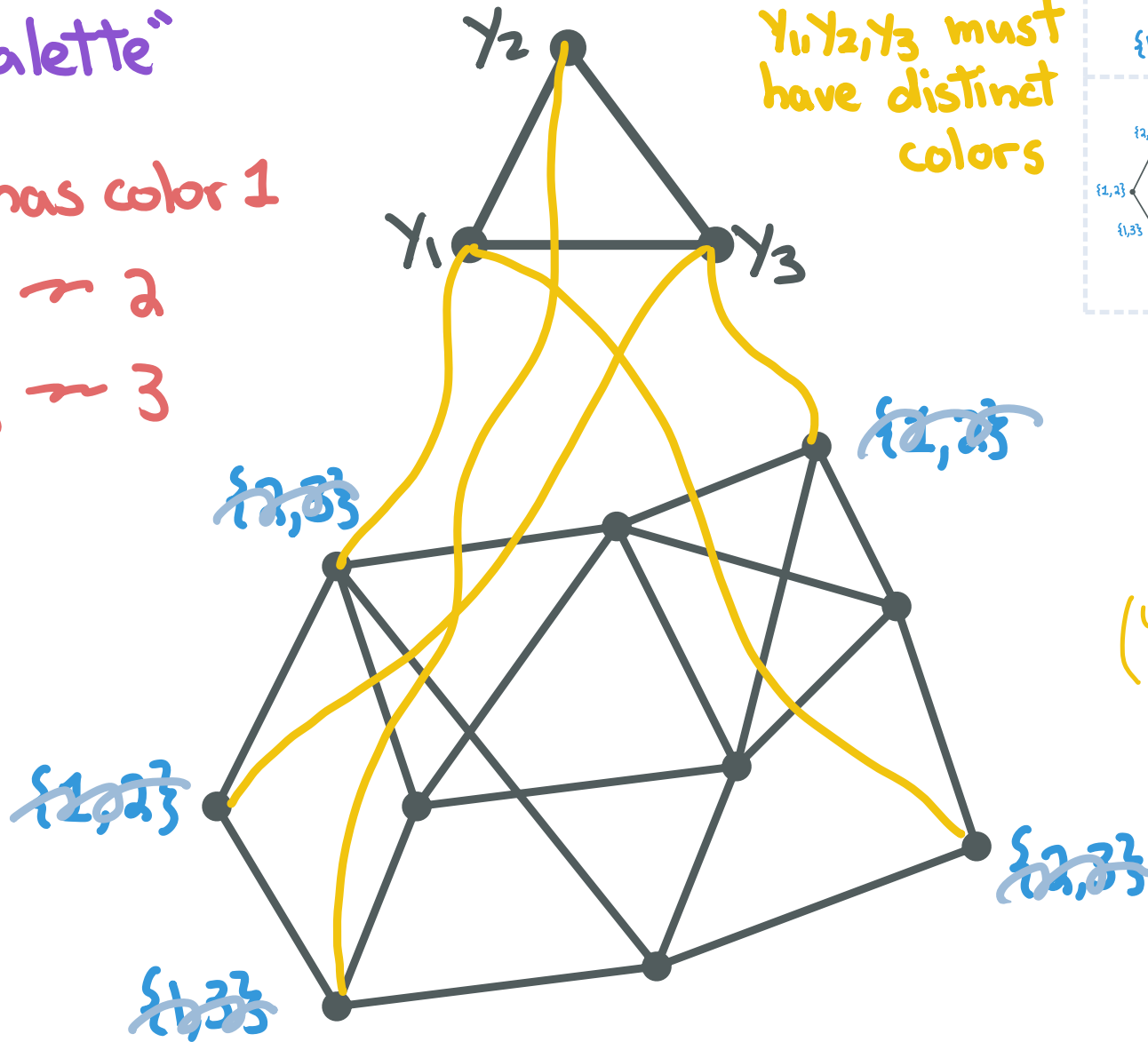
"color palette"

wlog γ_1 has color 1

$\gamma_2 \rightsquigarrow 2$

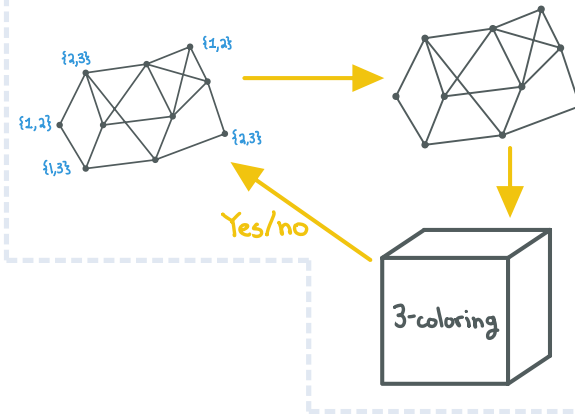
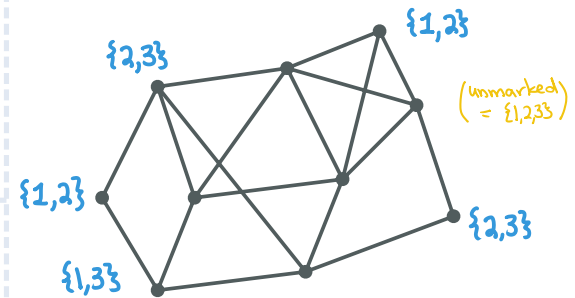
$\gamma_3 \rightsquigarrow 3$

$\gamma_1, \gamma_2, \gamma_3$ must
have distinct
colors



Subset 3-coloring

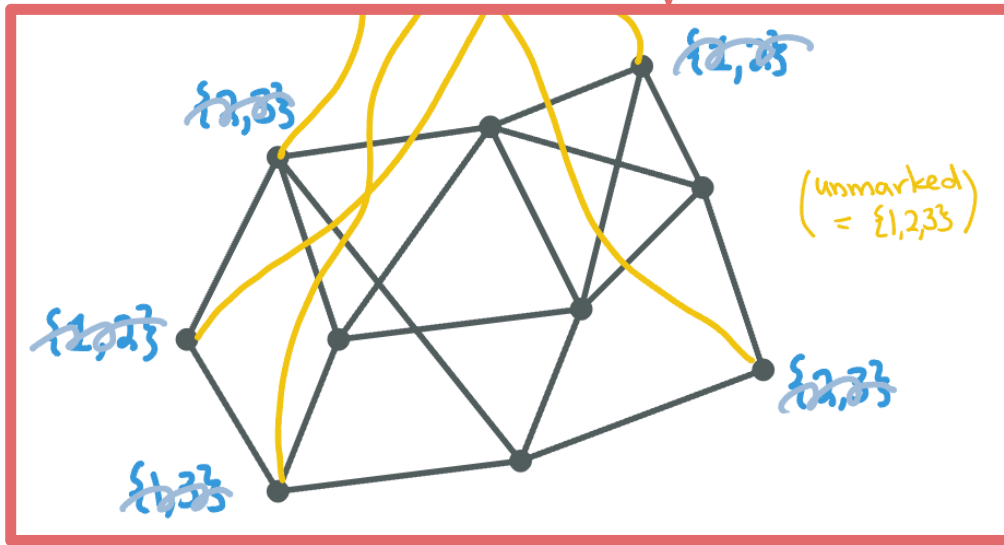
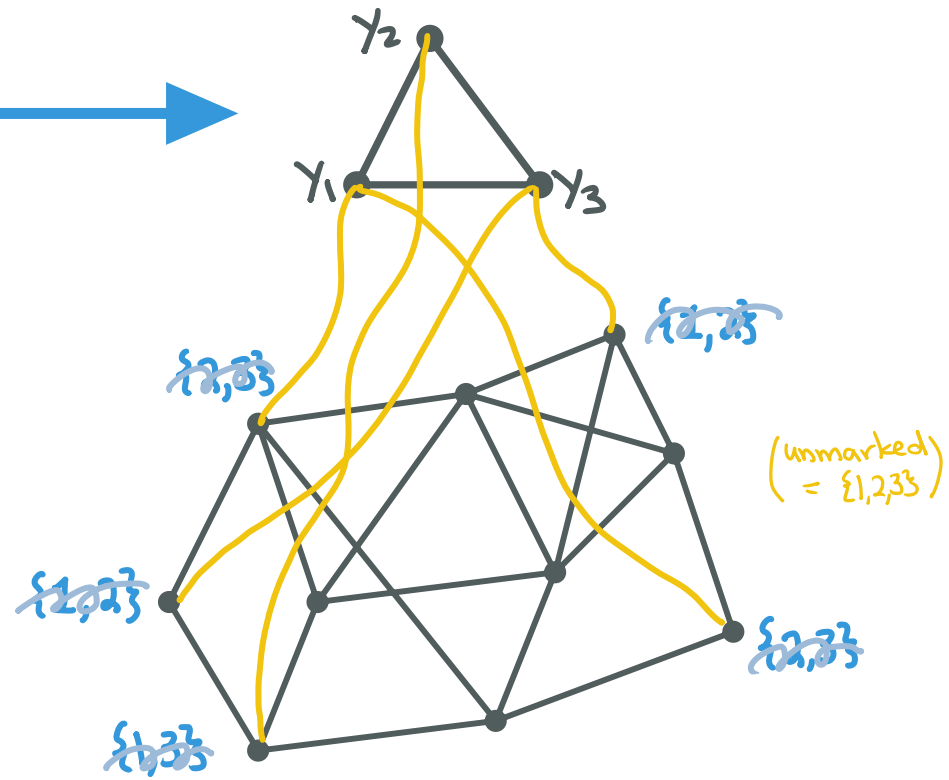
for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$
of colors allowed for v .



(unmarked)
= $\{1, 2, 3\}$

3-coloring on 

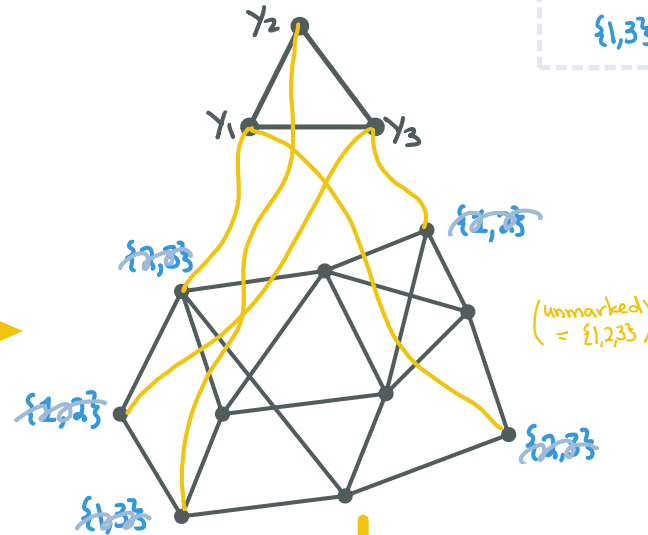
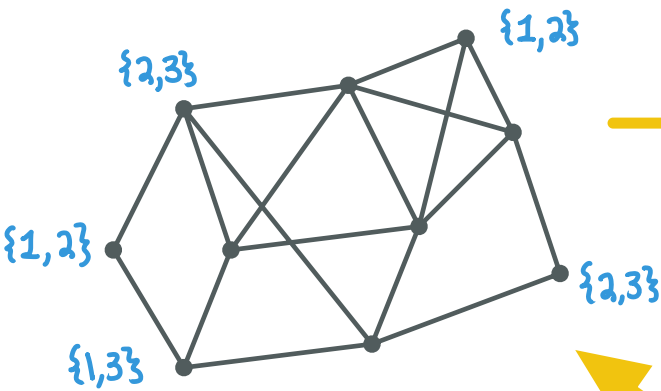
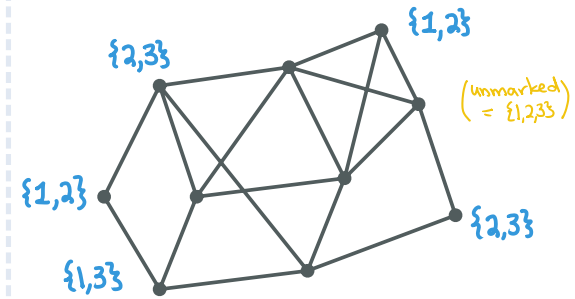
$\Rightarrow$ subset 3-coloring on



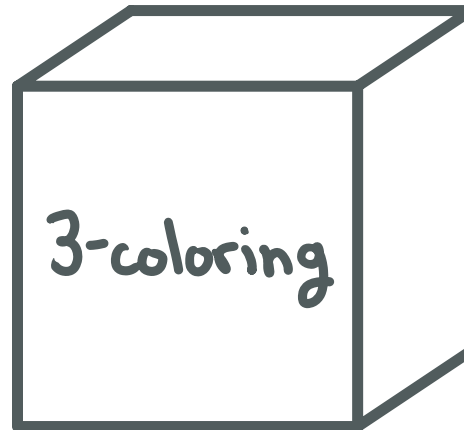
3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



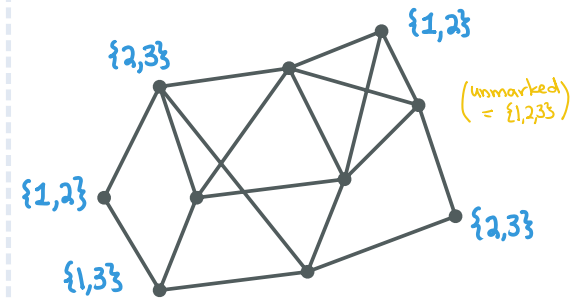
Yes/no



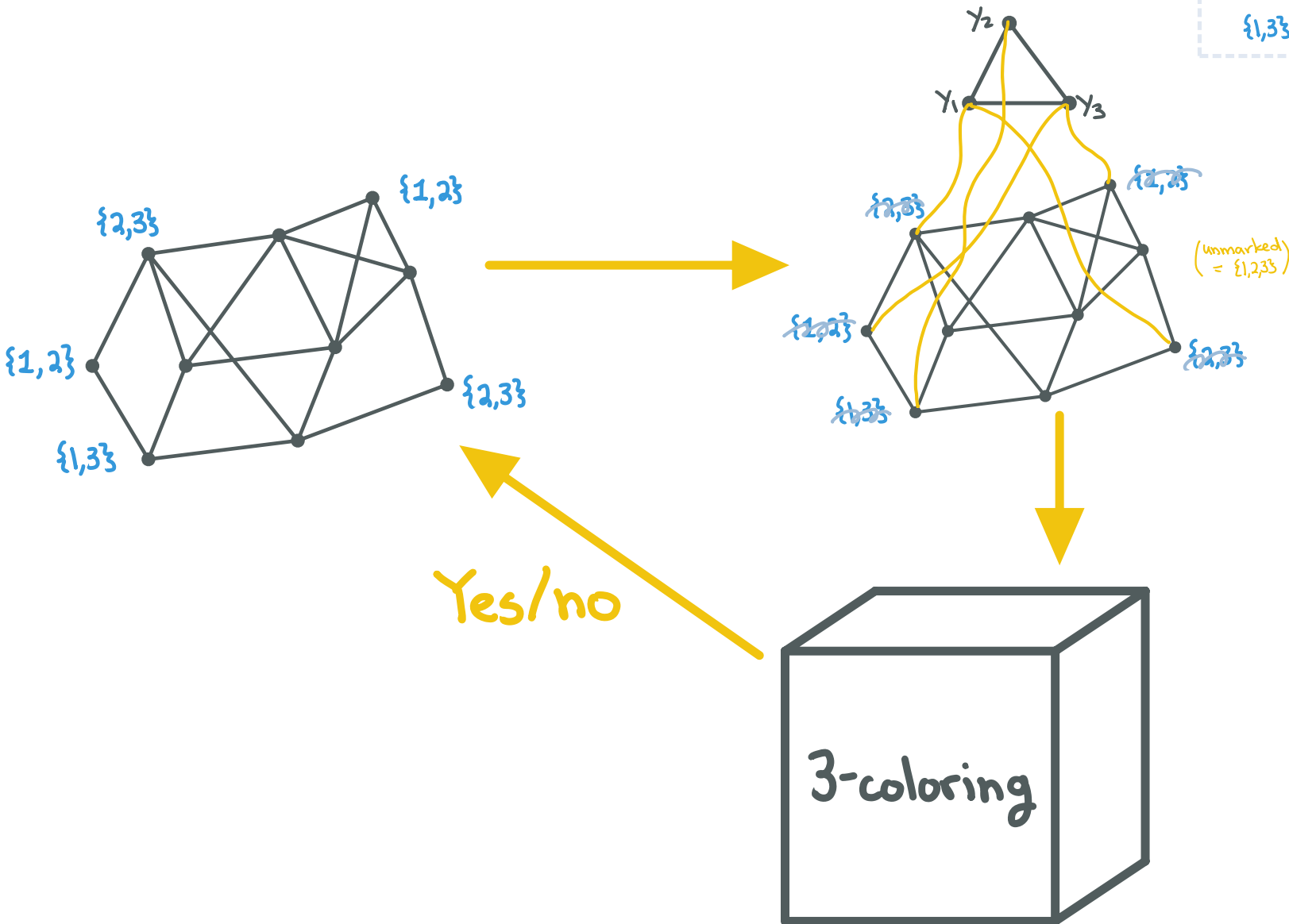
3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



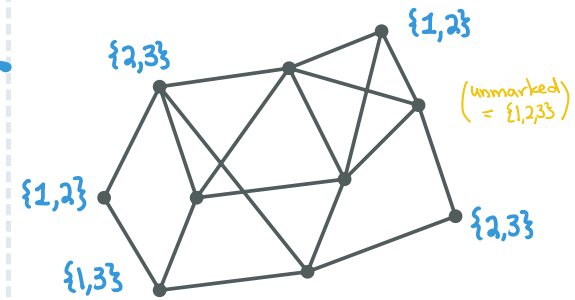
claim: polytime algo for 3-coloring implies polytime algo for subset 3-coloring.



3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .

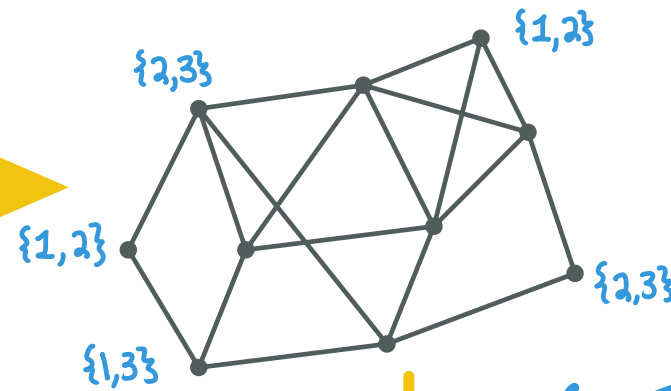


3-SAT

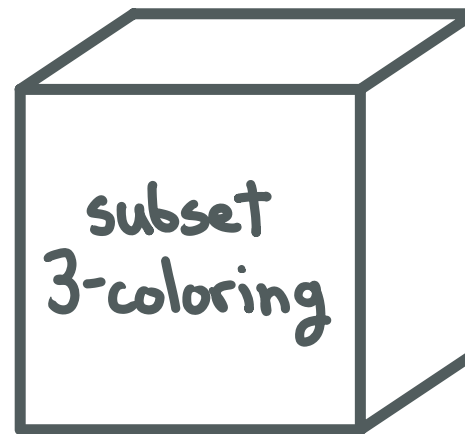
$$f(x_1, \dots, x_n) = (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \dots$$

color 1 = True
color 2 = False
color 3 = Neither

Yes/no



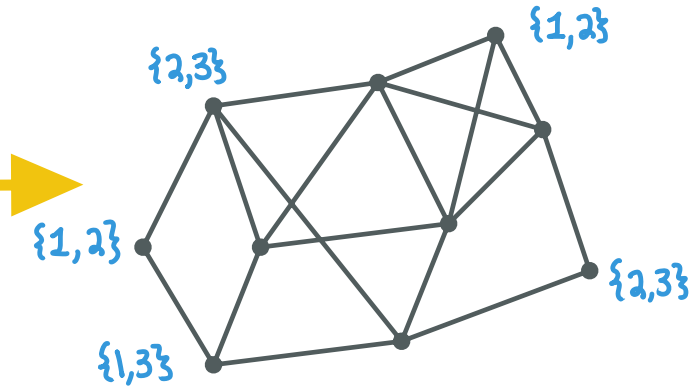
$\{T, F\}$
•
 x_i



3-SAT

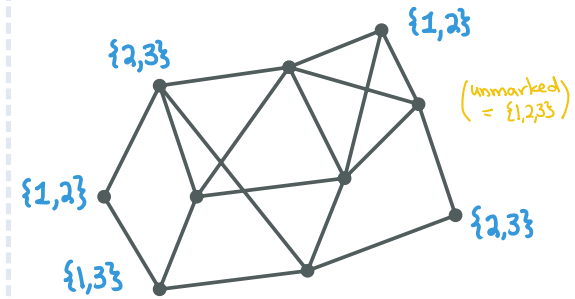
$$f(x_1, \dots, x_n) = (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \dots$$

ideas?



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .

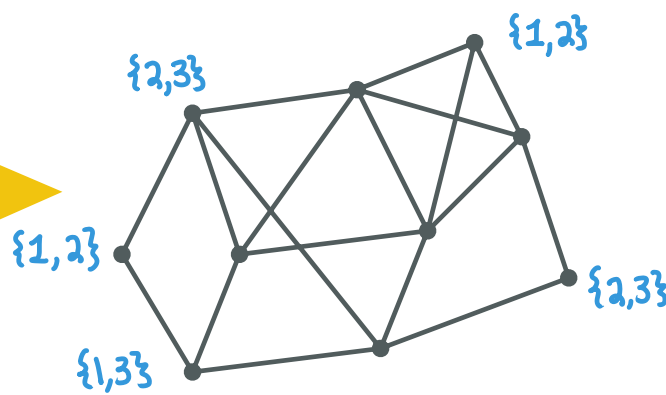


3-SAT

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

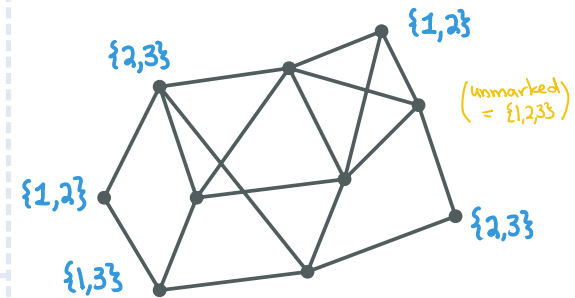
$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge$

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



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



Convenient to name colors:

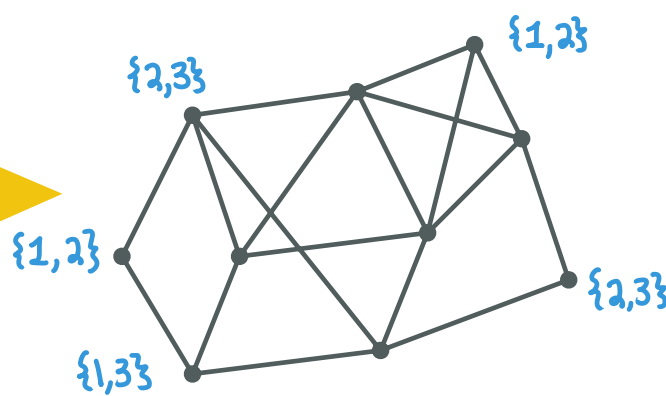
"T" = "True"

"F" = "False"

"N" = "Neither"

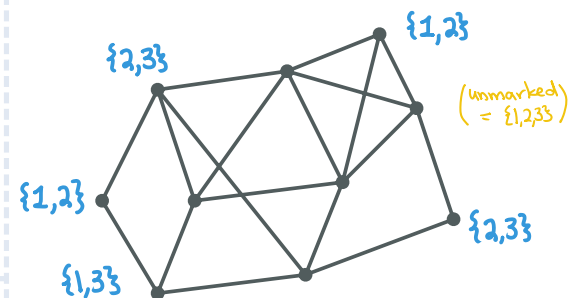
3-SAT

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



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



① variables

x_j

" A_j "

" $\bar{A}_j$ "

$\{T, F\}$

$\{T, F\}$

name colors:

"T" = "True"

"F" = "False"

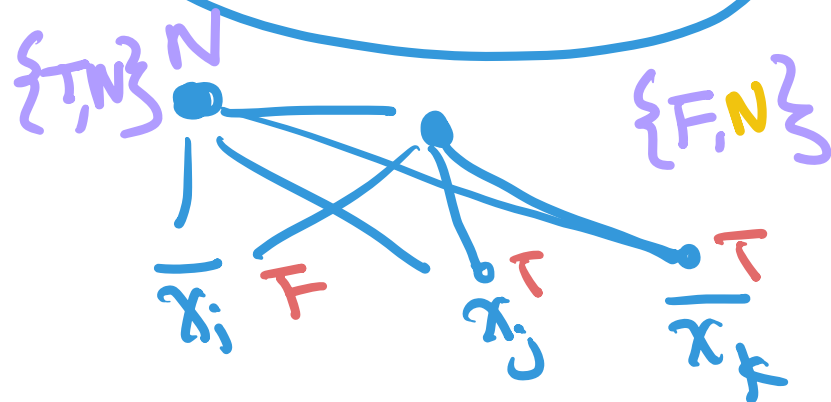
"N" = "Neither"

interpretation:

$x_j = A_j \cdot \text{color}$

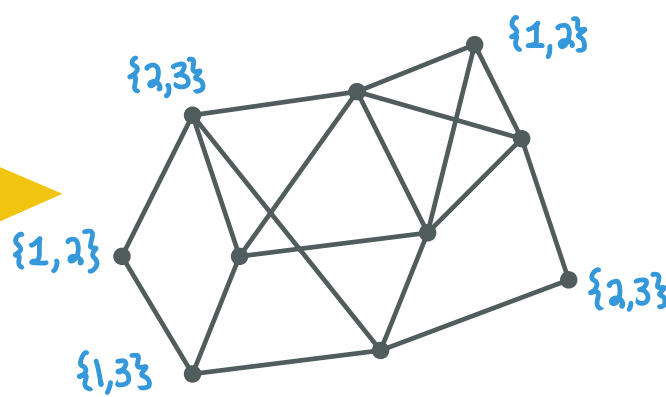
$\bar{x}_j = \bar{A}_j \cdot \text{color}$

$x_i \vee \bar{x}_j \vee x_k$



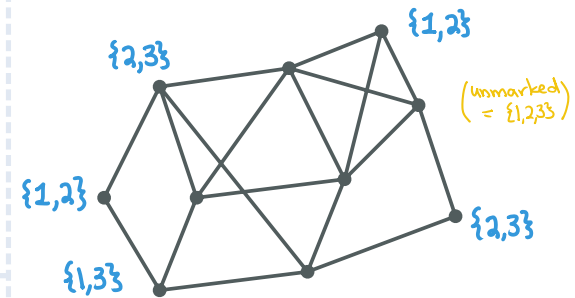
3-SAT

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



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



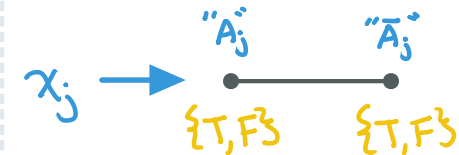
name colors:

"T" = "True"

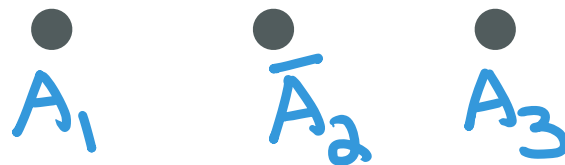
"F" = "False"

"N" = "Neither"

① variables

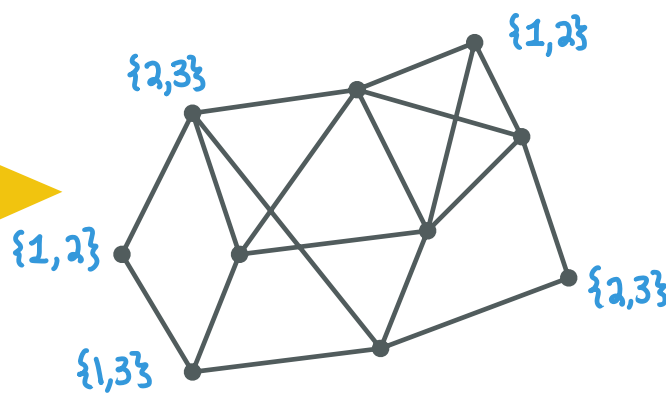


$$x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow$$



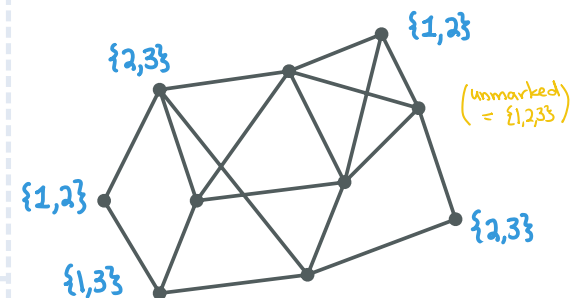
3-SAT

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



Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1,2,3\}$ of colors allowed for v .



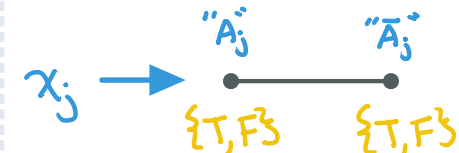
name colors:

"T" = "True"

"F" = "False"

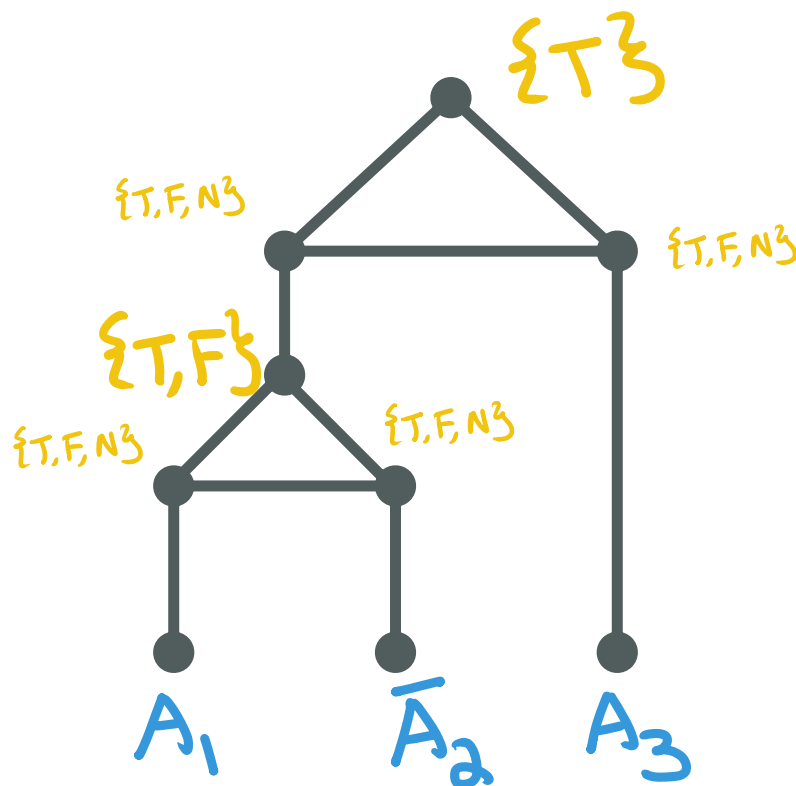
"N" = "Neither"

① variables

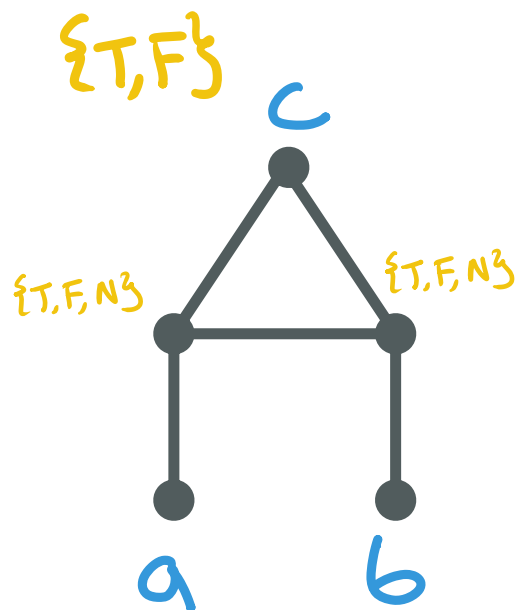


② clauses

$$x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow$$



Idea: "or-gadget"

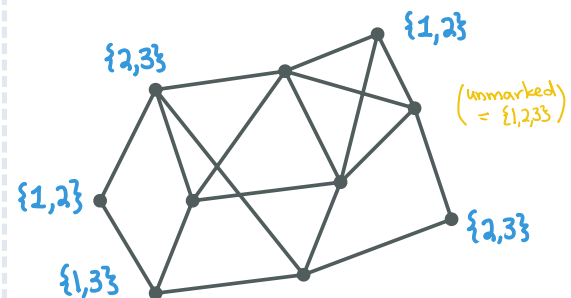


$$C = (a \vee b)$$

Lemma

$C.\text{color} = \text{True}$
possible iff $a \vee b = \text{True}$

Subset 3-coloring
for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$
of colors allowed for v .



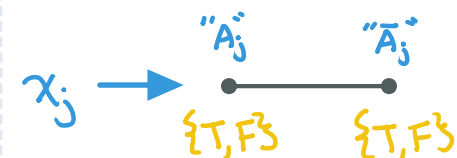
name colors:

"T" = "True"

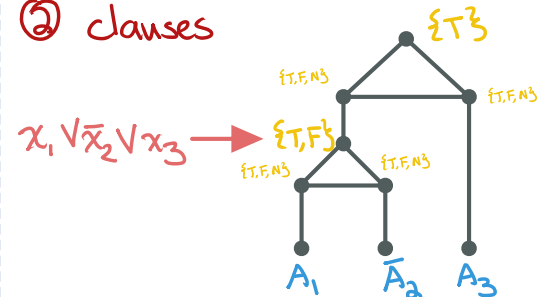
"F" = "False"

"N" = "Neither"

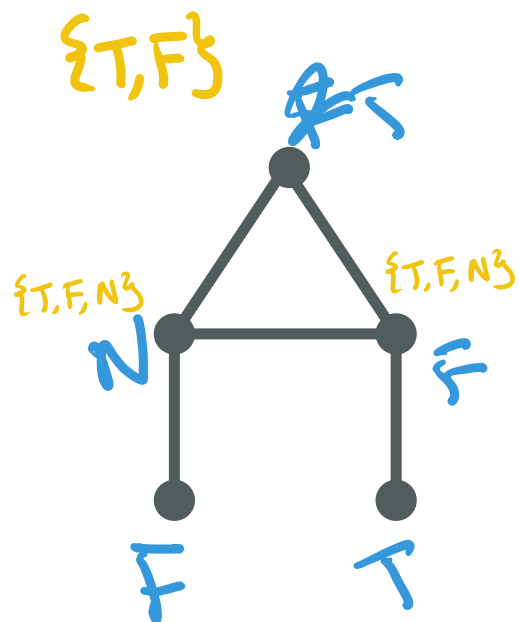
① variables



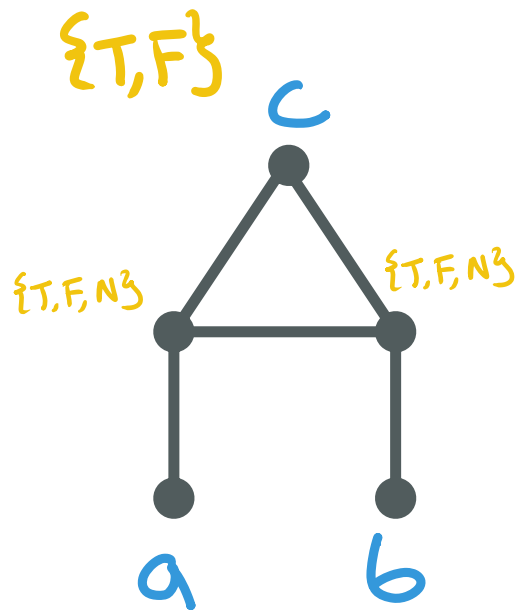
② clauses



Proof



Idea: "or-gadget"



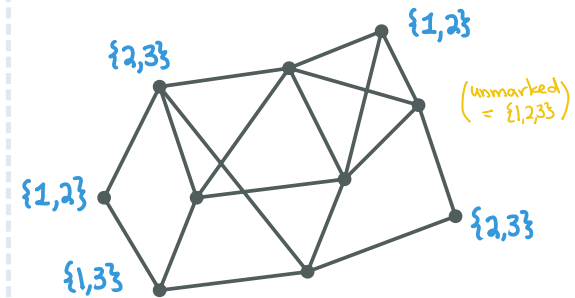
Lemma

$c.\text{color} = \text{true}$
possible iff $a \vee b = \text{False}$

$$x_1 \vee \bar{x}_2 \vee x_3$$

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .



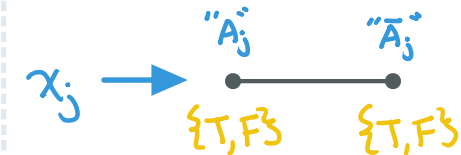
name colors:

"T" = "True"

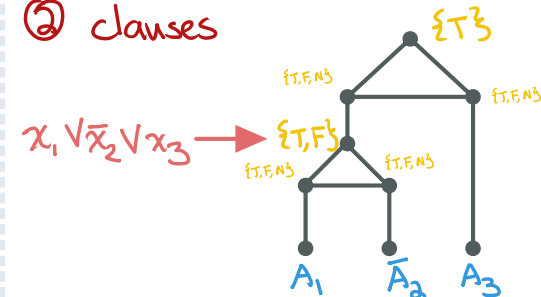
"F" = "False"

"N" = "Neither"

① variables



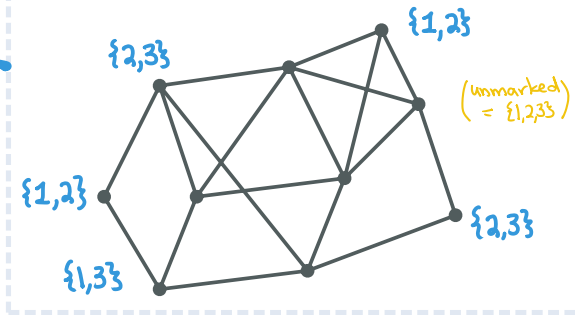
② clauses



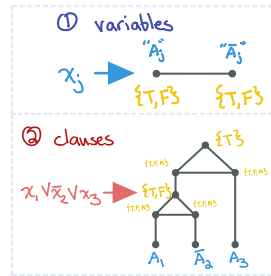
3SAT $\rightarrow$ "subset 3-coloring" $\rightarrow$ 3-coloring

Subset 3-coloring

for each vertex v , subset $S_v \subseteq \{1, 2, 3\}$ of colors allowed for v .

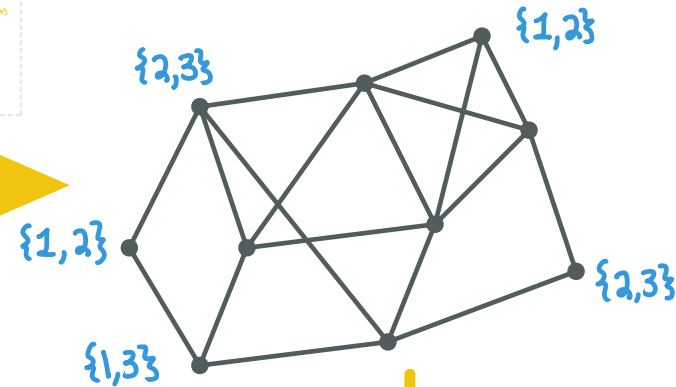


claim: polytime algo for subset 3-coloring
implies polytime algo for 3-SAT

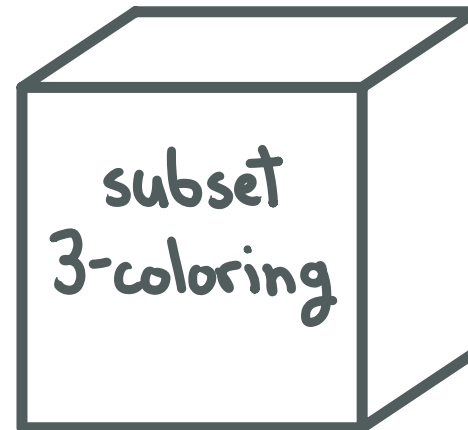


3-SAT

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



Yes/no



Chapter 13

Optimization in graphs

This lecture continues to investigate the subtle distinctions between problems that are polynomial time solvable, and those that resist our best algorithmic efforts. This time, all our problems are posed in undirected graphs. We will also consider special cases of undirected graphs to see if the additional structure changes the fundamental character of the problem. The problems introduced here are called the *independent set*, *(vertex) coloring*, and *dominating set* problems. They are all simple to define. Of these only the independent set problem will be presented completely. Vertex coloring and dominating set will be partly presented and partly delegated to exercises.

13.1 Independent set

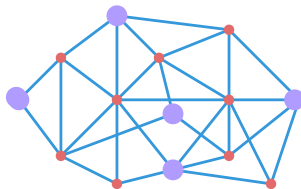
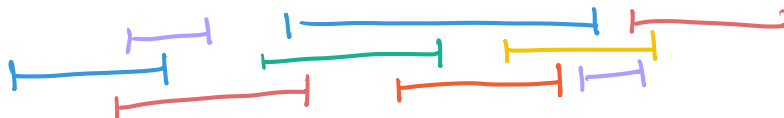


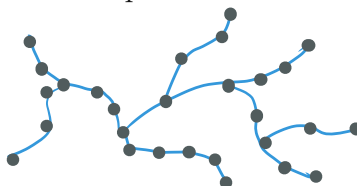
Figure 13.1: An independent set of size 5.

The independent set problem is a graph problem defined as follows. Let $G = (V, E)$ be an undirected graph. Recall that graphs consist of vertices (V) and edges (E) that connect pairs of vertices. A set of vertices $S \subset V$ is *independent* if any two vertices $a, b \in S$ are not connected by an edge. The *maximum independent set problem* asks for the maximum cardinality independent set. In the decision version of the problem, the input consists of a graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, and asks if there exists an independent set S with size $|S| \geq k$. One can also consider the weighted version where we are also given positive vertex weights $w : V \rightarrow \mathbb{R}_{>0}$, and the goal is to choose the maximum weight independent set (or decide if the maximum weight is at least an input parameter T).

There are many special cases of independent set that are useful. We will discuss two of them. The first is for *intervals on a line*. The input consists of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$, with positive edge weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$. We say that a set of intervals $S \subset \mathcal{I}$ is independent if they are pairwise disjoint; i.e., $I \cap J = \emptyset$ for every two intervals $I, J \in S$. This is the same as the graph definition if we identify each interval as a vertex, and have an edge between every two intervals that overlap.



The second example is the special case where the graph G is a tree T . Since trees are a particularly simple class of graphs, it is interesting to investigate whether one can obtain a better algorithm for independent set in trees than in graphs in general.



13.1.1 Independent set in graphs

Let us first consider the most general formulation of maximum weight independent set in graphs. Let $G = (V, E)$ be a graph with m edges and n vertices. Let $w : V \rightarrow \mathbb{R}_{>0}$ assign positive weights to the vertices. The most general goal is to compute the maximum weight independent set $S \subset V$.

We first observe that such a set S can be found by brute force. Here is a recursive algorithm.

The recursive algorithm will take exponential time. This is slow, but it establishes a baseline. Can we do better? What about the simpler, unweighted setting?

$\text{MWIS}(G = (V, E), w : V \rightarrow \mathbb{R}_{\geq 0})$

1. If $V = \emptyset$ then return 0

2. Let v be any vertex in V

3. Return the maximum of:

// all independent sets excluding v

A. $\text{MWIS}(G - v, w)$ where $G - v$ denotes the graph obtained by removing v and all its incident edges from G .

// all independent sets including v

B. $w(v) + \text{MWIS}(G', w)$ where G' obtained by removing v and its neighbors

Let us instead try to show that the problem should be hard - in particular, by proving the following.

Theorem 13.1. *A polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for SAT.*

We know that SAT can be reduced to 3-SAT in polynomial time, so it suffices to show that a polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for 3-SAT.

Let $f(x_1, \dots, x_n)$ be a 3-SAT formula with m clauses $C_1, \dots, C_m$ and n variables $x_1, \dots, x_n$. We interpret each C_i as a set of three elements out of the set of symbols $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. We want to convert f into a graph, where somehow an independent set larger than some particular size would indicate that f is satisfiable.

Clause gadgets. For each clause C , we make a triangle (i.e., three new vertices fully connected by edges). Each vertex in the triangle corresponds to a literal in the clause. For example, if $C = (x_1 \vee \bar{x}_2 \vee x_3)$, we create a triangle with three vertices, with one vertex corresponding to x_1 , $\bar{x}_2$, and x_3 . Thus a symbol such as x_1 or $\bar{x}_2$ may correspond to multiple vertices in the construction, with one vertex for each clause it appears in. So far, for m clauses, we have created a graph of m disjoint triangles, with $3m$ vertices and $3m$ edges in all. Note already that any independent set in G has at most one vertex per clause/triangle.

Our graph does not yet encode the fact that we can only set each variable x_i to be exactly one of true or false, and not both. Currently, we can take an independent set that (for example) contains vertices for both x_1 and $\bar{x}_1$. To encode this constraint, we add an edge between every pair of vertices of the form x_i and $\bar{x}_i$, for all i . With these

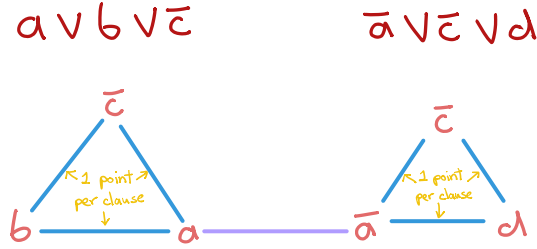


Figure 13.2: The gadget encoding disjunctions over three variables as triangles.

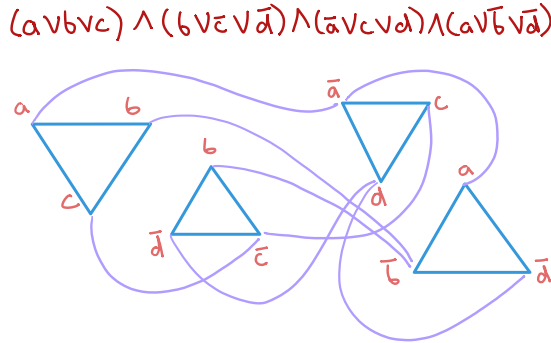


Figure 13.3: Encoding the 3-SAT formula $f(a, b, c, d) = (a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (a \vee \bar{b} \vee \bar{d})$ as a graph, where the maximum independent set indicates whether or not f is satisfiable.

additional edges, an independent set can no longer include variables corresponding to both x_i and $\bar{x}_i$.

This completes the constructions. In total the graph has $O(m)$ vertices and $O(m + n)$ edges. A minimal example is given in fig. 13.2. A more ornate example is given in fig. 13.3.

We now claim that $f(x_1, \dots, x_n)$ is satisfiable if and only if there is an independent set of size m in the graph G . Suppose first that there is an independent set S of size m . For each variable x_i , S may contain vertices corresponding to x_i , or vertices corresponding to $\bar{x}_i$, but not both, because of the edges we added in the second part of our construction. Consider the assignment where we set $x_i = 1$ if any of the x_i -vertices appear in S , or we set $x_i = 0$ if any of the $\bar{x}_i$ -vertices appear in S . (If neither type appears in S , then we set x_i to whatever we want.) We claim that this assignment satisfies all the clauses in f . We know that (a) there are m triangles (one per clause), (b) S has m vertices, and (c) S can have at most one vertex per triangle (to be independent). So S has exactly one vertex from every clause gadget. For each clause in f , our assignment satisfies the clause via the symbol for that vertex.

Conversely, suppose f is satisfiable, and fix a satisfying assignment. For each

clause C , choose a literal that satisfies it, and add the corresponding vertex from C 's gadget in our graph to a set S . Since these choices are disjoint, S will have m vertices. We claim that S is independent. Indeed, picking one vertex per clause implies we satisfy all the triangle edges. Moreover we only choose a vertex corresponding to the symbol x_i if $x_i = 1$ in the satisfying assignment, and we only choose a vertex corresponding to the symbol $\bar{x}_i$ if $x_i = 0$. In particular, we can't choose both. So S will be independent over the second set of edges as well. Thus S is an independent set of size m .

To conclude, given a 3-SAT formula f with m clauses, we have built a graph G , of size polynomial in the size of f , that has an independent set of size m iff f is satisfiable. If we had a polynomial time algorithm for maximum independent set, we could run it on this graph and decide if f is satisfiable. This completes the proof.

13.1.2 Independent sets of intervals

While independent set on graphs turned out to be difficult, this does not preclude a more efficient algorithm for intervals, since this is a special case. Thus let us consider the problem for intervals with renewed optimism.

Let $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$ be a set of intervals. Let $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$ assign positive weights to each interval. The goal is to compute the maximum weight independent set of intervals.

A lazy approach to this problem is to convert it to a graph by first comparing all pairs of intervals to identify all conflicting pairs. This would take quadratic time. (One can also do this faster - how?) Then one can compute brute force in the graphical setting as discussed above. But of course, this is far from efficient. More importantly, just because we can interpret the interval problem in terms of a graph, doesn't mean we should automatically treat it as a graph problem. This is because the intervals have more structure than graphs (in general), and this structure in particular makes it more difficult to reduce from SAT.

For graphs we explored brute force algorithms that recursively test one element at a time. For the sake of concreteness, let us quickly rewrite that algorithm in terms of intervals. First, the recursive spec: for a set of intervals $\mathcal{I}$ and weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$, we define

MWIS-intervals($\mathcal{I}, w$) = the weight of the maximum weight independent subset of intervals of $\mathcal{I}$.

We now implement **MWIS-intervals** as specified above.

MWIS-intervals($\mathcal{I}$, $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$)

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{J \in \mathcal{I} : J \cap I = \emptyset\}$ *// = set of intervals disjoint from I*
4. return the maximum of
 - /* The MWIS among those excluding I. */*
 - A. MWIS-intervals($\mathcal{I} \setminus \{I\}, w$)
 - /* The MWIS among those including I. */*
 - B. $w(I) + \text{MWIS-intervals}(\mathcal{I}', w)$

In our brute force algorithm, we are not selective about the order in which we process the elements. In a graph, there is no obvious order to the vertices anyway. In intervals, however, there are natural ways to order the intervals and this can make a big difference.

Let us sort $\mathcal{I}$ in increasing order of left endpoint (a_i) in $O(n \log n)$ time, and henceforth assume that

$$\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$$

lists the intervals in increasing order of a_i . Let us adjust the brute force algorithm again so that it processes the intervals in increasing order of left endpoint. The revised pseudocode is as follows.

MWIS-intervals($\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$, $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$)

- /* We assume $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$ is sorted in increasing order of a_i . */*
1. If $n = 0$ then return 0
 2. Let $i > 1$ be the first index such that $a_i > b_1$
 - // I_i is the first interval disjoint from I_1 , and $\{I_i, \dots, I_n\}$ = set of intervals disjoint from I_1*
 3. return the maximum of
 - /* The MWIS among those excluding I_1 */*
 - A. MWIS-intervals($\{I_2, \dots, I_n\}, w$)
 - /* The MWIS among those including I_1 */*
 - B. $w(I_1) + \text{MWIS-intervals}(\{I_i, \dots, I_n\}, w)$

In this ordered version of brute force, every recursive call is for a subset of intervals of the form $\{I_i = [a_i, b_i], I_{i+1} = [a_{i+1}, b_{i+1}], \dots, I_n = [a_n, b_n]\}$ for some index i ; that is, a suffix of our *ordered* sequence of intervals. This limits the number of combinations of intervals in our subproblems: there are only $n + 1$ suffixes (including the empty set). We have an ideal situation for caching.

Let us now give a dynamic programming algorithm for the maximum weight independent set of intervals. Fix an input consisting of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$ and weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$. Sorting the intervals $\mathcal{I}$ in $O(n \log n)$, we assume the intervals are indexed in increasing order of left endpoint a_i . For $i \in \mathbb{N}$, we define

MWIS-intervals(i) = the maximum weight of any independent set of intervals among $\{I_i, I_{i+1}, \dots, I_n\}$.

We implement this with the following pseudocode.

$$\text{MWIS-intervals}(i) = \begin{cases} 0 & \text{if } i > n, \\ \max \left\{ \begin{array}{l} \text{MWIS-intervals}(i+1), \\ w(I_i) + \text{MWIS-intervals}(j(i)) \end{array} \right\} & \text{otherwise,} \end{cases}$$

where $j(i)$ is the first index j such that $a_j > b_i$. We can identify $j(i)$ by scanning the list of intervals. We apply dynamic programming to the above recursive code to make sure we never solve the same subproblem twice.

For the running time, we have n subproblems, each of which takes $O(n)$ time to compute, excluding recursive subcalls. (The $O(n)$ time comes from scanning for the next index $j(i)$.) The space usage is $O(n)$, since there are $O(n)$ many subproblems. The solution to the original problem is given by the call **MWIS-intervals**(1). Correctness is inherited from brute force.

Can one do better? In fact, yes. For fixed i , we spend $O(n)$ time to find the first index $j(i)$ such that $a_j > b_i$. But instead one can do binary search to find $j(i)$, in $O(\log n)$ time. This improves the running time of each subproblem to $O(\log n)$, and the overall running time to $O(n \log n)$. This is our first examples where a dynamic programming algorithm is sped up by an adjustment to the implementation.

Taking a step back, we see that there is substantially more structure to intervals than a generic graph. A provably hard problem in graphs can be solved as fast as sorting in interval graphs!

13.1.3 Independent set in trees

We now consider the independent set problem in our second special case of graphs: trees. Here we are given a tree $T = (V, E)$ and (optionally) vertex weights $w : V \rightarrow \mathbb{R}$.

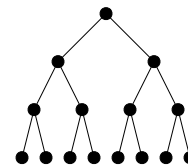
The goal is to compute the maximum weight independent set $S \subset V$. The question really is whether trees are as general as graphs, or as structured as intervals on a line. Let us make two initial observations relating trees to our previous reduction.

1. The reduction from 3SAT to independent set was full of triangles. But trees cannot have triangles, or any cycle, for that matter.
2. The graph corresponding to a set of intervals is also not a tree - any three overlapping intervals form a triangle.

There is clearly much more structure to trees than graphs. As mentioned above, trees have no cycles. If we remove an edge or a vertex, then we get two entirely disconnected parts. In fact one can always find a vertex whose removal breaks up the graph into subtrees with at most half the vertices (why?) - a remarkable property that is certainly not true for graphs.

Between this chapter and the last, we have seen multiple problems turn out to be easy when we take advantage of a clearly defined, *linear* ordering. But there is no clear notion of “left” and “right” in a tree. But while trees don’t have a linear structure, they do have a *hierarchical* structure that is very useful.

Fix any node $r \in V$ to be the root, and interpret T as a rooted tree. Now we can speak of “children”, “parents”, “subtrees” and so forth. There is a clear notion of “top” - the root r - and “bottom” - at the leaves. Let’s use this ordering to design a recursive algorithm for the maximum weight independent set. From a hierarchical point of view, it is natural to design a recursive algorithm using subtrees to define the subproblems. Consider the following recursive spec. For a vertex $v \in V$, we define



$\text{MWIS}(v)$ = the maximum weight of any independent set in the subtree rooted at v .

Let us now sketch out an implementation $\text{MWIS}(v)$. Fix v , and suppose we want to find the maximum weight independent set in the subtree rooted at v . A natural decision is whether or not to include v in the independent set. If we don’t include v , then we can recurse on each of the children to compute the maximum weight independent set in each subtree. If we do include v , then... wait. Actually, it’s not so obvious. When we recurse on a child x of v , this may return (the value of) an independent set that includes x . But this is incompatible with our decision to include v in the independent set. The following pseudocode summarizes our progress and poinpoints where we get stuck.

MWIS(v)

1. Return the maximum of
 - /* Don't include v in the independent set */*
 - A. The sum of MWIS(x) over all children x of v
 - /* Include v in the independent set */*
 - B. The weight of v plus the sum of ~~MWIS(x)~~ ??? over all children x of v

In step (1.B), where we got stuck, what did we really want? Not the MWIS in the subtree rooted at x , since this can include x . Rather we need the MWIS in the subtree rooted at x *among those that exclude x* . MWIS(x) simply doesn't offer this. The easiest way to address this deficiency is to strengthen the definition MWIS(x). We introduce a boolean flag, called `exclude-root` $\in \{0, 1\}$, as an additional parameter, and define MWIS(v , `exclude-root`) as follows.

MWIS(v , `exclude-root`): If `exclude-root` = `true`, then the maximum weight independent set in the subtree rooted at v among those excluding v . If `exclude-root` = `false`, then the maximum weight independent set in the subtree rooted at v (which may or may not include v).

This stronger definition makes the implementation effortless.

MWIS(v , `exclude-root`)

1. If `exclude-root`
 - /* Don't include v in the independent set */*
 - A. Return the sum of MWIS(x , `false`) over all children x of v
2. Otherwise return the maximum of:
 - /* Don't include v in the independent set */*
 - A. The sum of MWIS(x , `false`) over all children x of v .
 - /* Include v in the independent set */*
 - B. The weight of v plus the sum of MWIS(x , `true`) over all children x of v

Good. It was a little trickier, but we have now established a recursive algorithm for computing the maximum weight independent set. The overall answer is given by MWIS(r , `false`). But our algorithm is not yet efficient. To this end – as usual – we

apply caching. Let us save the answers to each $\text{MWIS}(v, \text{true})$ and $\text{MWIS}(v, \text{false})$ in a table¹ of size $2n$. Then we never compute the same answer twice.

Consider the running time. We have

$$(\# \text{ subproblems})(\text{time per subproblem}) = n \times O(n) = O(n^2).$$

Here we upperbound the “time per problem” by $O(n)$ because a vertex may have up to n children. But this analysis is too conservative. The total number of children over all nodes is n . A more refined analysis first observes that

$$(\text{time spent on } \text{MWIS}(v, \dots)) = O(\# \text{ children of } v).$$

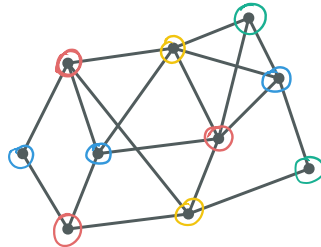
Summing over all v , we have

$$\sum_v (\text{time spent on } \text{MWIS}(v, \dots)) = O\left(\sum_v \# \text{ children of } v\right) = O(n).$$

So actually the running time is $O(n)$. We also mention that the space usage is $O(n)$ because there are $O(n)$ subproblems, each requiring constant space. Thus we have a $O(n)$ time algorithm for maximum weight independent set in trees. *Which is faster than for intervals! Faster than sorting!*

13.2 Graph coloring

Graph coloring problems are an important topic in graph theory. Generally speaking, vertex coloring problems are problems where you want to color the vertices of an undirected graph so that any two adjacent vertices have different colors. More formally, we can define a *proper vertex coloring* as an assignment $\pi : V \rightarrow [k]$, for some integer k , such that for any edge $\{u, v\} \in E$, $\pi(u) \neq \pi(v)$. The goal is to find a vertex coloring $\pi : V \rightarrow [k]$ for k as small as possible.



¹We assume the vertices are number 1 through n . Otherwise one can number them in a preprocessing step.

Besides being a useful problem in practice (e.g., assigning non-overlapping frequency ranges to nearby cell-phone towers), it is an important problem in combinatorics, where graph theorists have discovered remarkable structural theorems such as the following famous theorem.

Theorem 13.2 (Brook’s theorem, [Bro41]). *Let G be an undirected graph with maximum degree Δ . Then G can be colored in Δ colors unless G is either (a) a complete graph or (b) an odd cycle.*

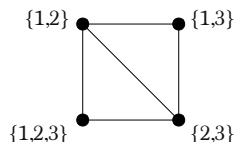
We briefly point out there are many related graph problems of interest. For example, in edge coloring, the goal is to color the edges of an undirected graph so that no two adjacent edges² have the same color. For edge coloring, *Vizing’s theorem* asserts that every graph with maximum degree Δ can be edge-colored with at most Δ colors.

Let us return to vertex coloring. Here we are interested in the optimization problem rather than absolute bounds such as Brook’s theorem. Given an undirected graph $G = (V, E)$, the goal is to color G with as few colors as possible. Or, as a decision problem: given an undirected graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, can G be colored in k or fewer colors? There is also the k -coloring problem where k is fixed, but otherwise asks the same question: Given as input a graph $G = (V, E)$, can G be colored in k or fewer colors? Note that for k -coloring, an $n^{f(k)}$ running time, for any function f , counts as polynomial time.

This problem is yet another search problem, since it is easy to check if an assignment is a proper coloring with at most k colors. So in particular we have an exponential time algorithm by brute force. Can one do better? What about the special case of $k = 3$ – 3-coloring?

Theorem 13.3. *A polynomial time algorithm for vertex 3-coloring implies a polynomial time algorithm for SAT.*

To prove the above it is convenient to introduce the following more general variant of k -coloring. In the *subset k -coloring problem*, each vertex v in the graph is annotated with a subset $S_v \subset [k]$ of colors available to that vertex. The goal is to properly color graph the graph $\pi : V \rightarrow [k]$ such that for each v , $\pi(v) \in S_v$.

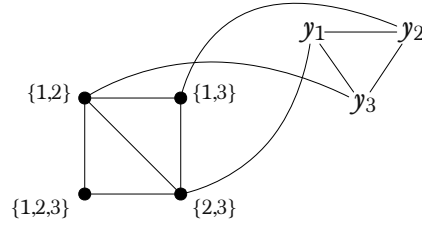


²Two edges are adjacent if they have a common endpoint.

Lemma 13.4. *Let $k \in \mathbb{N}$ be fixed. A polynomial time algorithm for k -coloring implies a polynomial time algorithm for subset- k coloring.*

Proof. Assume an instance of subset- k coloring defined by a graph $G = (V, E)$ with m edges and n vertices and, for each vertex v , a set $S_v \subset [k]$ of colors available to v .

To reduce to k -coloring, we will construct a new graph H that extends the original graph G . Starting from G , we first introduce a clique K_k of k colors. That is, we create k new vertices $y_1, \dots, y_k$ and add an edge between every pair of new vertices. Observe that any k -coloring of this clique requires every y_i to receive a distinct color, and this uses up all k colors. (Conceptually we interpret the color assigned to y_i as the i th color.) Now, for each vertex v , and each color i omitted from S_v , we add an edge between y_i and v . (Conceptually, this forbids v from taking i in a k -coloring.) Below we draw the construction for $k = 3$.



We claim that G is k -subset-colorable with respect to the sets $\{S_v : v \in V\}$ iff H is k -colorable. First suppose G is k -subset-colorable. Given any subset coloring π of G , we can extend π to a k -coloring on H by coloring $\pi(y_i) = i$ for each i . This does not create any monochromatic edges between the y_i 's, nor does it decrease any monochromatic between a y_i and a vertex $v \in V_G$ because $i \notin S_v$ for any edge $\{y_i, v\}$.

Conversely, consider any proper coloring of $\pi : V_H \rightarrow [k]$ of H . Without loss of generality we may assume that $\pi(y_i) = i$ for each i .³ We need to show that $\pi(v) \in S_v$ for all v . Indeed, for any color $i \notin S_v$, the edge $\{y_i, v\}$ implies that $\pi(v) \neq \pi(y_i) = i$. $\square$

To prove theorem 13.3, it now suffices to show that subset 3-coloring can be used to solve SAT.

Theorem 13.5. *A polynomial time algorithm subset 3-coloring implies a polynomial time algorithm for 3-SAT.*

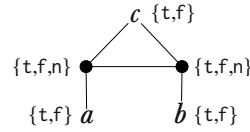
³This is because we only care if two colors have the same labels, which is invariant to permutation of the labels.

Let $f(x_1, \dots, x_n)$ be a 3-SAT formula with n variables $x_1, \dots, x_n$ and m clauses $C_1, \dots, C_m$. We want to create an instance of subset 3-coloring (of roughly the same size) that is feasible iff f is feasible.

We point out that the names of the colors in a coloring problem are arbitrary. For this reduction, it is convenient to name the colors **true**, **false**, and **nonsense**; or **t**, **f**, and **n** for short.

We first create vertices corresponding to the variables. For each variable x_j , we make two vertices A_j and $\bar{A}_j$ each constrained to have colors from the set $\{t, f\}$. We also add an edge between A_j and $\bar{A}_j$, which forces exactly one of A_j or $\bar{A}_j$ to be colored **true**, and the other to be colored **false**.

Lemma 13.6. *Consider the following instance of subset 3-coloring with five vertices.*

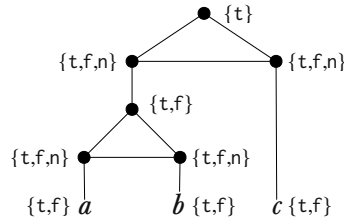


In any proper subset coloring, $c = \text{true}$ only if $a \vee b = \text{true}$.

Proof. By inspection. □

Now we can construct a gadget for 3-CNF clauses. Recall that $(a \vee b \vee c) = ((a \vee b) \vee c)$. Likewise we can compose two OR gadgets as follows.

Lemma 13.7. *Consider the following instance of subset 3-coloring with 8 vertices.*



Then in any proper subset coloring of the above, $a \vee b \vee c = \text{true}$.

For each clause C_i , we identify the three vertices corresponding to the literals in that clause. (For example, $x_1 \vee \bar{x}_2 \vee x_3$ corresponds to the vertices A_1 , $\bar{A}_2$, and A_3 .) We build the 3CNF gadget, per lemma 13.7 over these three vertices. Each clause generates 6 new variables.

Between the two vertices A_j and $\bar{A}_j$ for each variable x_j , and the constant-size gadgets built over these vertices for each C_i , we obtain an instance of subset 3-coloring that is satisfiable iff f is satisfiable.

Indeed, given a satisfying assignment x , we color $A_j = \text{true}$ and $\bar{A}_j = \text{false}$ if $x_j = \text{true}$, and reverse the colors if $x_j = \text{false}$. One can then color each clause gadget appropriately. Conversely, given any proper coloring of the clauses, one constructs an assignment x by setting x_j to be the color of A_j . One can verify that, since the coloring was able to color the clause-gadgets properly, each clause in the formula is satisfied by x . This completes the proof.

Coloring with more than 3 colors. We have not shown that k -coloring is hard for $k > 3$. This is left to the reader in exercise 13.1.

Coloring in interval graphs and trees. As with the independent set problem, the reduction from SAT to 3-coloring does not forbid polynomial time algorithms for trees or intervals. This is left to the reader in exercise 13.3.

13.3 Dominating set

[picture of dominating set]

Let $G = (V, E)$ be an undirected graph. A set of vertices $S \subseteq V$ is a *dominating set* if every vertex $v \in V$ is either in S or the neighbor of a vertex in S . The minimum dominating set problem is to compute the minimum cardinality dominating set. The decision version additionally specifies an integer k and the question is whether there is a dominating set of size k (or greater). One can consider weighted versions of the problem where we have weights $w : V \rightarrow \mathbb{R}_{>0}$ and the goal is to compute the dominating set of minimum weight.

Perhaps it is not so surprising, after discussing independent set and coloring, that dominating set is also as hard as SAT.

Theorem 13.8. *A polynomial time algorithm for dominating set implies a polynomial time algorithm for SAT.*

Let $f(x_1, \dots, x_n)$ be a Boolean formula in CNF with n variables, m clauses, and total size p . We will construct an undirected graph G such that G has a dominating set of size n iff f is satisfiable. Here are the first steps in making G .

1. For every variable x_i , we create two vertices denoted v_i and $\bar{v}_i$. v_i is associated with the symbol x_i and the assignment $x_i = \text{true}$; $\bar{v}_i$ is associated with the symbol $\bar{x}_i$ and the assignment $x_i = \text{false}$.
2. For every clause C_j , we create a vertex w_j .

3. For every clause C_j and every variable x_i appearing as x_i (without negation) in C_j , we add the edge $\{v_i, w_j\}$. For every variable x_i appearing in C_j negated as $\bar{x}_i$, we add an edge from.
4. For every variable x_i , we add an edge between v_i and $\bar{v}_i$.

Let us try to explain the intuition of the construction so far. We have vertices for each assignment to each variable. We also have vertices for the clauses. We have edges between (vertices representing) clauses and the (vertices representing the) single-variable assignments that satisfy the clauses. The total number of variables is $O(m + n)$, and the number of edges is $O(p)$.

Now consider any satisfying assignment for f . We will construct a dominating set S with n vertices as follows. If the satisfying assignment has $x_i = \text{true}$, then we include the corresponding vertex v_i . If the satisfying assignment has $x_i = \text{false}$, then we include the corresponding vertex $\bar{v}_i$. So S has n vertices, with each vertex corresponding to a variable's assignment. To see that this is a dominating set, we observe that:

1. For every clause-vertex w_j corresponding to clause C_j , the vertex corresponding to a variable satisfying C_j will dominate w_j .
2. For each variable-vertex v_i or $\bar{v}_i$, the fact that S takes either v_i or $\bar{v}_i$ ensures that both v_i and $\bar{v}_i$ are dominated.

Conversely, suppose our graph has a dominating set S of size n . Ideally S would contain exactly one of the vertices, v_i or $\bar{v}_i$, for each variable x_i , and form a satisfying assignment accordingly. However S could contain neither v_i or $\bar{v}_i$, or both v_i and $\bar{v}_i$. So it is not clear that a dominating set of size n implies a satisfying assignment (and the reader might think of some simple counterexamples).

So to summarize the current state of affairs: our construction is such that if f has a satisfying set, then the graph has a dominating set with (at most) n vertices. However, a dominating set with n vertices does *not* imply a satisfying assignment for f . We ask the reader how to improve the construction to complete the proof.

We would like to force S to take at least one of v_i or $\bar{v}_i$ for each variable x_i . If so, and S also has size (at most) n , then it must take exactly one of these vertices for each x_i . How can we force every dominating set to take at least one of v_i or $\bar{v}_i$? We can introduce another auxiliary variable, which we denote u_i , that can only be dominated by v_i or $\bar{v}_i$ (or itself). So we add the following steps to our construction.

5. For every variable x_i , we create a (third) vertex u_i .
6. We add the edges $\{u_i, \bar{v}_i\}$ and $\{u_i, v_i\}$.

The introduction of u_i means that we have to have either u_i , v_i or $\bar{v}_i$ in any dominating set. Moreover, if a dominating set S has u_i , it can always be replaced with v_i and $\bar{v}_i$ while keeping S a dominating set.

Now, we can complete the proof. As before, a satisfying assignment to f maps to a dominating set of size n . Now, suppose S is a dominating set with at most n vertices. We know that S includes at least one of v_i , $\bar{v}_i$, or u_i for each i , in order to dominate u_i . Whenever S contains some u_i , we replace it with v_i or $\bar{v}_i$ (either one works); now we know that S includes at least one of v_i or $\bar{v}_i$. On the other hand, $|S| \leq n/2$ implies that S contains exactly one of v_i or $\bar{v}_i$. Consider the assignment $x_1, \dots, x_n$ corresponding to the vertices in S . For each clause C_j , the corresponding variable w_j is dominating by some v_i or $\bar{v}_i$ in S ; the corresponding assignment ($x_i = \text{true}$ or $x_i = \text{false}$) satisfies C_j . Thus we have a satisfying assignment.

To summarize, we have a dominating set of size n iff f is satisfiable. Our auxiliary graph is polynomial in the size of f , and the construction takes polynomial time. A polynomial time algorithm that computes the minimum dominating set can, in particular, decide if there is a dominating set of size n in the auxiliary graph of a given CNF $f(x_1, \dots, x_n)$. We would obtain a polynomial time algorithm for CNF SAT by first converting the algorithm to the graph described above and then computing a dominating set.

Dominating set for intervals and trees. What about dominating set in interval graphs, or in trees? We leave these questions to the reader in exercise 13.6.

13.4 Additional notes and materials

Spring 2024 lecture 13 materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2024 lecture 15 materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2023 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)

- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2022 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

13.5 Exercises

[Eri19, Chapter 3] also has several exercises about dynamic programming on trees.

Exercise 13.1. Theorem 13.3 proved that a polynomial time algorithm for 3-coloring implies a polynomial time algorithm for SAT. Prove that for any $k > 3$, a polynomial time algorithm for k -coloring implies a polynomial algorithm for SAT.⁴

Exercise 13.2. Let k be fixed. Suppose you had access to a polynomial time algorithm for the decision version of k -coloring. Use this algorithm to obtain a polynomial time algorithm that obtains a k -coloring of a graph (if one exists).

Exercise 13.3. Recall that the k -coloring problem is to decide if there is a vertex coloring with at most k -colors. The subset k -coloring problem also specifies for each vertex v a subset $S_v \subseteq [k]$, and the k -coloring is restricted to have each vertex v colored by a color in S_v . 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. The k -coloring problem in trees.
2. The k -coloring problem for intervals.
3. The subset k -coloring problem in trees, for fixed k .
4. The subset k -coloring problem for intervals, for fixed k .

Exercise 13.4. Let $G = (V, E)$ be an undirected graph. We say that a set of vertices is *almost independent* if each vertex $v \in S$ has at most one neighbor in S .⁵ Consider the problem of computing the maximum cardinality of any almost independent set of

⁴It might be helpful to focus on $k = 4$ to develop your ideas.

⁵Two vertices u and v are neighbors if they are connected by an edge.

vertices. For this problem, 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.

Exercise 13.5. ⁶Let $G = (V, E)$ be an undirected graph with m edges and n vertices, and $k \in \mathbb{N}$. Recall that a *vertex k -coloring* (a.k.a. a *graph k -coloring*) is an assignment of colors (out of $\{1, \dots, k\}$) to each $v \in V$ so that no two adjacent vertices have the same color. When such an assignment exists, G is said to be *k -colorable*. Now suppose that for each color $i = 1, \dots, k$, and each vertex $v \in V$, there is a positive *assignment cost* $c(i, v) > 0$ for assigning color i to vertex v .

The following problem is called the *minimum cost graph coloring (MCGC)* problem. Given G and k as input, where $k \geq 3$, the goal is to find the minimum total assignment cost of any vertex k -coloring of V (which is $+\infty$ if no such coloring exists).⁷ We consider the problem both for general graphs and trees.

1. (5 points) Suppose $G = (V, E)$ is a graph with m edges and n vertices. For this problem, 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.
2. (5 points) Suppose $G = (V, E)$ is a tree with n vertices. For this problem, 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.

Exercise 13.6. Recall the dominating set problem from section 13.3. Here we will consider the weighted version where the vertices are given positive weights, and the goal is to compute the minimum weight dominating set. 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. The minimum weight dominating set problem for intervals, *with the additional assumption that no two intervals are nested*.

⁶You may assume as fact that a polynomial time algorithm for k -coloring implies a polynomial time algorithm for SAT; this fact is otherwise left to the reader as exercise 13.1.

⁷Technically this would be the “infimum” of all assignment costs of (valid) vertex k -colorings.

To state it more precisely: the input consists of n weighted intervals $\mathcal{I}$. The non-nested assumptions means that for any two intervals $I, J \in \mathcal{I}$, we never have I contained in J or J contained in I .

The goal is to compute the minimum weight subset $S \subseteq \mathcal{I}$ of intervals such that every interval in $\mathcal{I}$ is either in S or overlaps some interval in S .

- (For 1 pt. extra credit) Extend your algorithm to general intervals.⁸
2. The minimum weight dominating set problem in trees.

Exercise 13.7. Let $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$ be a collection of n intervals on the real line, and let $p_1, \dots, p_m \in \mathbb{R}$ be a collection of m points on the line with positive weights $w(p_i) > 0$. A point p_i *hits* an interval I_j if $p_i \in I_j$. A subset of points $p_{i_1}, \dots, p_{i_k}$ is a *hitting set* for $I_1, \dots, I_n$ if every interval I_j is hit by some point p_{i_h} in the subset. We assume that $p_1, \dots, p_m$ itself is a hitting set. We also assume for simplicity that all the points p_i are distinct. The goal is to compute the minimum weight of any hitting set of points. For this problem, 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.

Exercise 13.8. Given a graph $G = (V, E)$, a *clique* is a set of vertices $S \subseteq V$ such that every pair of vertices in S is connected by an edge. (This is sort of like the opposite of an independent set.) When the vertices are weighted by $w : V \rightarrow \mathbb{R}_{>0}$, one can ask for the maximum weight clique in the graph.⁹

1. Consider the problem of finding the largest weight of any clique from a set of weighted intervals. (Here a clique of intervals is a set of intervals where every two intervals are overlapping.) The input consists of n intervals $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$, and you can assume that you can look up the weight of an interval in constant time. For this problem, 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.

⁸Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.

⁹Note: we updated the problems so that one only needs to return the maximum weight of any clique, as opposed the actual clique itself, for the sake of simplicity.

2. Consider the problem of finding the largest weight of any clique in a tree. For this problem, 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.
3. Consider the problem of finding the largest weight of any clique in a graph G . For this problem, 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.¹⁰

Exercise 13.9. Here we consider a special case of independent set that we will call *k-neighbors independent-set*, for a fixed parameter $k \in \mathbb{N}$. (e.g., $k = 5$.) The input is an undirected graph $G = (V, E)$ with m edges and n vertices, with the following property: every set $S \subseteq V$ has at most k neighbors of S outside of S .¹¹ The goal is to compute the maximum size independent set. For this problem, 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.

Exercise 13.10. Let $I_1, \dots, I_n$ be a collection of n intervals on the real line, and let $p_1, \dots, p_m \in \mathbb{R}$ be a collection of m points on the line. Let each interval I_j be assigned a positive weight $w(I_j) > 0$.

An interval I_j *covers* a point p_i if $p_i \in I_j$. A collection of intervals $I_{j_1}, \dots, I_{j_k}$ is an *interval cover* of $p_1, \dots, p_m$ if every point p_i is covered by at least one interval I_{j_h} in the collection. We assume that the collection of all the intervals $I_1, \dots, I_n$ is an interval cover of $p_1, \dots, p_m$. We also assume that all the points are distinct.

Consider the problem of computing the minimum weight of any interval cover of $p_1, \dots, p_m$. For this problem, 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.

Exercise 13.11. Can't we all just get along? You want to throw a big party to celebrate finishing midterm 2, but not all of your friends get along. So you resolve to invite as many friends as possible while making sure there are no problems within the party.

Formally, we assume that you have n friends indexed by $1, \dots, n$. Your input consists of a list L of length m of all pairs of friends who don't get along. The goal is

¹⁰Hint: Consider the "complement graph" $G' = (V, E')$ where there is an edge $\{u, v\}$ iff $\{u, v\}$ is not an edge in G .

¹¹Here a vertex v is a *neighbor* of S if it is connected by an edge.

to compute the largest cardinality (of any) subset of friends, $S \subseteq [n]$, that does not contain any pair of friends in the list L .

1. (5 points) For this problem, 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.
2. (5 points) Now suppose there are no “cycles” among the pairs of enemies: that is, there is no sequence of friends $f_0, \dots, f_{k-1}$, where $k \geq 3$, such that f_i does not get along with $f_{i+1 \bmod k}$ for all i . For this setting, 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.

Sample solution.

Problem. Let $T = (V, E)$ be an undirected tree with positive vertex weights $w : V \rightarrow \mathbb{R}_{>0}$. Consider the problem of computing the weight of the maximum weight independent set in T . For this problem, 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.

Solution. We root T arbitrarily and henceforth refer to the root, children, subtrees, etc., accordingly.

1. *Recursive spec.* For a vertex $v \in V$, we define

$\text{MWIS}(v)$ = the weight of the maximum weight independent set in the subtree rooted at v .

$\text{MWIS-no-root}(v)$ = the weight of the maximum weight independent set in the subtree rooted at v among those that exclude v .

2. *Recursive implementation.*

$\text{MWIS}(v)$

1. Return the maximum of:

/ The maximum weight independent set containing v. */*

A. $w(v) + \sum_{\text{children } w \text{ of } v} \text{MWIS-no-root}(w)$

/ The maximum weight independent set excluding v. */*

B. $\sum_{\text{children } w \text{ of } v} \text{MWIS}(w)$

$\text{MWIS-no-root}(v)$

1. return $\sum_{\text{children } w \text{ of } v} \text{MWIS}(w)$

3. *Solving the original problem.* We return $\text{MWIS}(r)$ where r is the root of the tree.

4. *Say “caching” or “dynamic programming”.* We cache the answers to the recursive algorithm above.

5. *Running time.* There are two subproblems for each vertex v . For fixed v , the running time is proportional to the number of children of v (plus a constant). Summing over all v counts each vertex as a child once, so we obtain a $O(n)$ running time overall.

6. *Brief proof / justification.* Fix v and consider $\text{MWIS}(v)$. The MWIS in the subtree rooted at v either includes v or it doesn't. If it does, then the restriction of the MWIS to each subtree of v is an independent set excluding the root of that subtree. By induction on subtrees, for each subtree rooted by a child w , $\text{MWIS-no-root}(w)$ returns the maximum weight of any independent set in the subtree excluding w . Line (1.A) then ensures that we return at least the weight of this set.

Now suppose the MWIS in the subtree rooted at v does not include v . Then the restriction of the MWIS to each subtree of v is an independent set of that subtree. By induction on subtrees, for each subtree rooted by a child w , $\text{MWIS}(w)$ returns the maximum weight of any independent set in the subtree. Line (1.A) then ensures that we return at least the weight of this set.

Consider now $\text{MWIS-no-root}(v)$, and consider the MWIS S of the subtree rooted at v , among those excluding v . For each subtree w , the restriction of S to the subtree rooted by w is an independent set in the subtree rooted at w . By induction on subtrees, $\text{MWIS}(w)$ returns the weight of the MWIS in this subtree. So the sum we return gives the MWIS overall.

Remark 13.9. In the lecture notes we added a flag to MWIS ; this time I thought it might be clearer to just make two functions where the intent of the flag is declared in the name of the function.

Sample solution.

Problem. Let $\mathcal{I}$ be a family of n intervals with positive weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$, and consider the problem of computing the weight of the maximum weight independent (i.e., non-overlapping) subset of $\mathcal{I}$. For this problem, 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.

Solution. We sort the intervals in increasing order of left end-point (in $O(n \log n)$ time) and let $I_i = [a_i, b_i]$ refer to the i th interval in increasing order of left end-point a_i .

1. *Recursive spec.* For $i = 1, \dots, n + 1$, we declare

$\text{MWIS}(i)$ = the maximum weight independent set of intervals $I_i, \dots, I_n$. ($i > n$ implies the empty collection of intervals.)

2. *Recursive implementation.*

MWIS(i)

1. If $i > n$ then return 0.
2. Return the maximum of

/ Include I_i . */*

A. $w(I_i) + \text{MWIS}(j + 1)$ where j is the last index of any interval I_j overlapping I_i .

/ Don't include I_i . */*

B. $\text{MWIS}(i + 1)$

3. Say “caching” or “dynamic programming”. We apply dynamic programming to the above recursive implementation.

4. *Extracting the original problem.* The overall solution is given by $\text{MWIS}(1)$.

5. *Running time.* There are $O(n)$ subproblems. Naively, identifying the interval I_j in step (2.A) would take $O(n)$ time in a loop. However, having already sorting the intervals by left-endpoint, we can use binary search to identify I_j in $O(n \log n)$ time. Overall (with dynamic programming) the algorithm takes $O(n \log n)$ time.

6. *Proof by induction.* Consider $\text{MWIS}(i)$. In the base case $i > n$ in which case there are no intervals and the answer is 0. Suppose $i \leq n$.

Consider the (true) MWIS of $I_i, \dots, I_n$. The MWIS either includes I_i or doesn't. Suppose first that the MWIS includes I_i . Since I_i overlaps I_j and the intervals are indexed in increasing order of left endpoint, the intervals in between $(I_{i+1}, \dots, I_{j-1})$ also overlap I_i . So in particular the MWIS does not include any of these intervals either, and is an independent subset of $I_{j+1}, \dots, I_n$. By induction on the number of intervals, $\text{MWIS}(j + 1)$ returns the weight of the MWIS of $I_{j+1}, \dots, I_n$. So $w(I_i) + \text{MWIS}(j + 1)$ is (at least) the weight of the true MWIS, and is one of the two options considered in step (2).

Now suppose the MWIS does not include I_i . Then it is the MWIS of $I_{i+1}, \dots, I_n$. By induction on the number of intervals, $\text{MWIS}(i + 1)$ returns this weight, hence so does $\text{MWIS}(i)$.