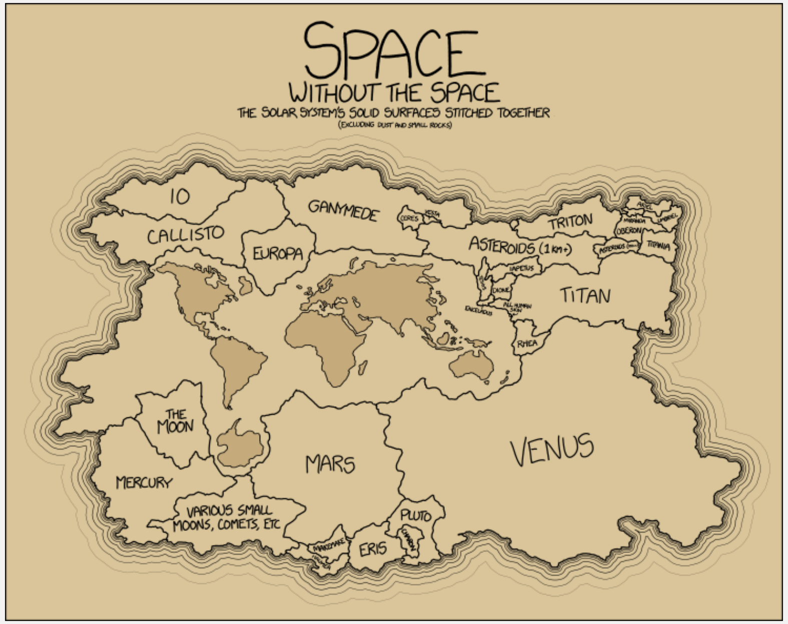
UMB CS622

Space Complexity

Wed, November 24, 2021



Announcements

- HW 9 due Sun 11:59pm EST
  - (after break)

- Happy Thanksgiving!

First: One More **NP**-Complete Problem

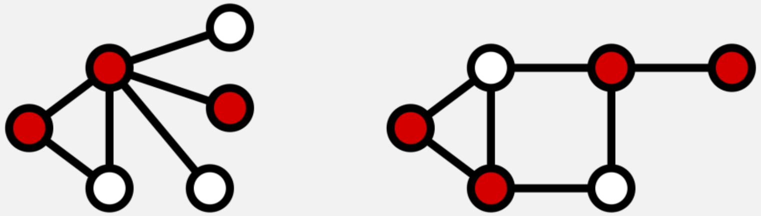
- $SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}$, we have $\Sigma y_i = t\}$
  - (reduce from *3SAT*)

- $VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$
  - (reduce from *3SAT*)

Theorem: *VERTEX-COVER* is NP-complete

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}$$

- A vertex cover of a graph is …
  - … a subset of its nodes where every edge touches one of those nodes



THEOREM

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Theorem: *VERTEX-COVER* is NP-complete

$VERTEX\text{-}COVER = \{\langle G, k \rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

3 steps to prove *VERTEX-COVER* is **NP**-complete:

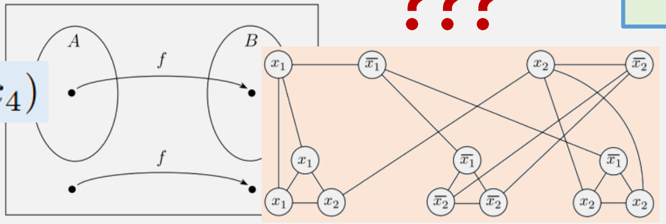
☑ 1. Show *VERTEX-COVER* is in **NP**

☑ 2. Choose the **NP**-complete problem to reduce from: *3SAT*

3. Show a poly time mapping reduction from *3SAT* to *VERTEX-COVER*

To show poly time mapping reducibility:
1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction** (or **contrapositive** of **forward direction**)

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$$



???

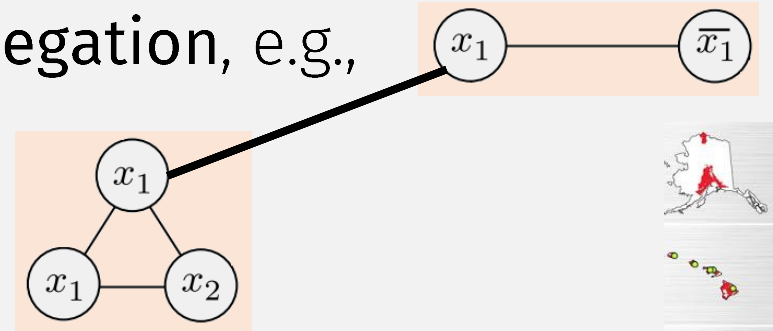
Theorem: *VERTEX-COVER* is NP-complete

$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

- A vertex cover of a graph is …
  - … a subset of its nodes where every edge touches one of those nodes

Proof Sketch: Reduce *3SAT* to *VERTEX-COVER*

- The reduction maps:
- Variable x_i → 2 connected nodes
  - corresponding to the var and its negation, e.g.,
- Clause → 3 connected nodes
  - corresponding to its literals, e.g.,

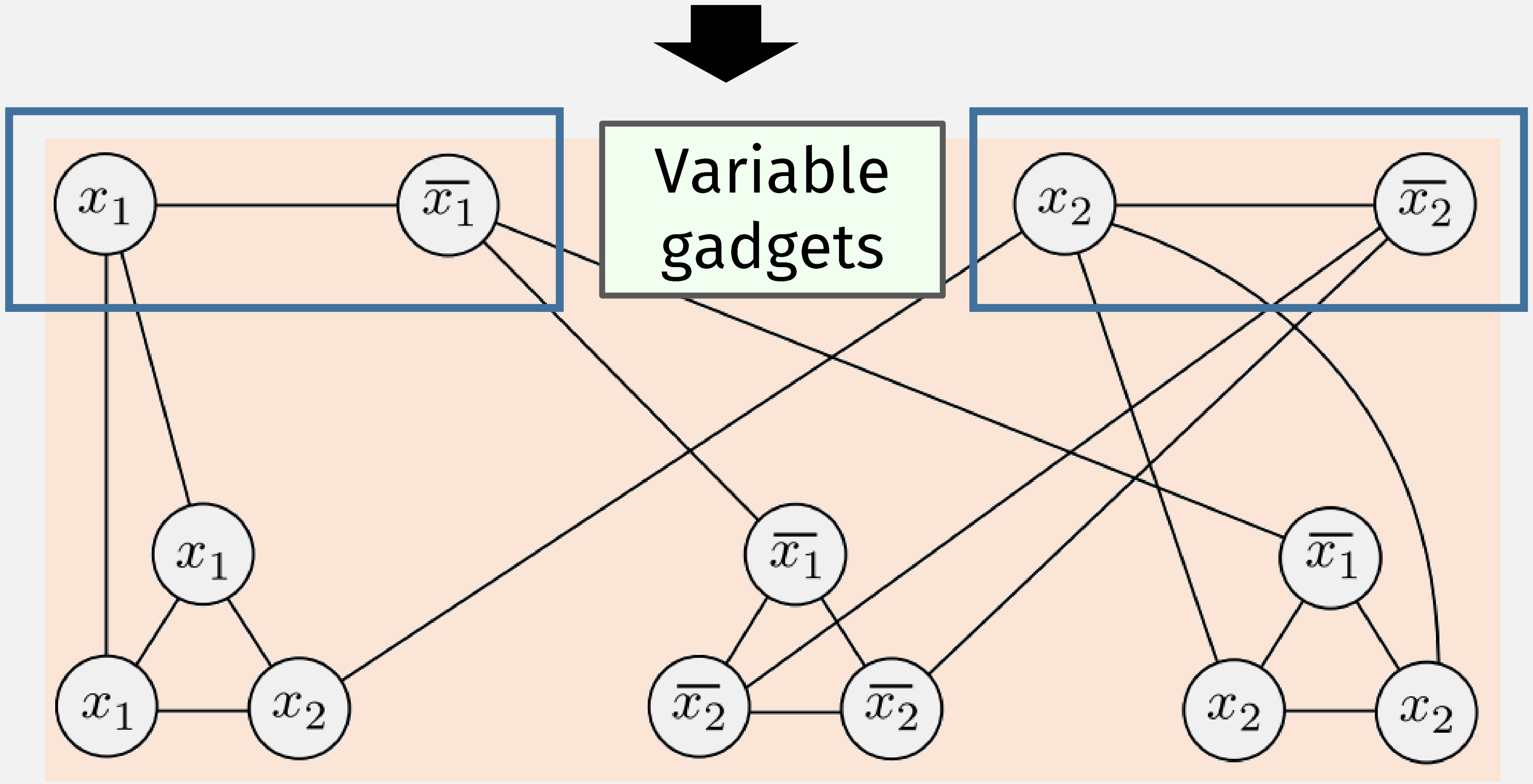


- Additionally,
  - connect var and clause gadgets by …
  - … connecting nodes that correspond to the same literal

VERTEX-COVER example

VERTEX-COVER $= \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

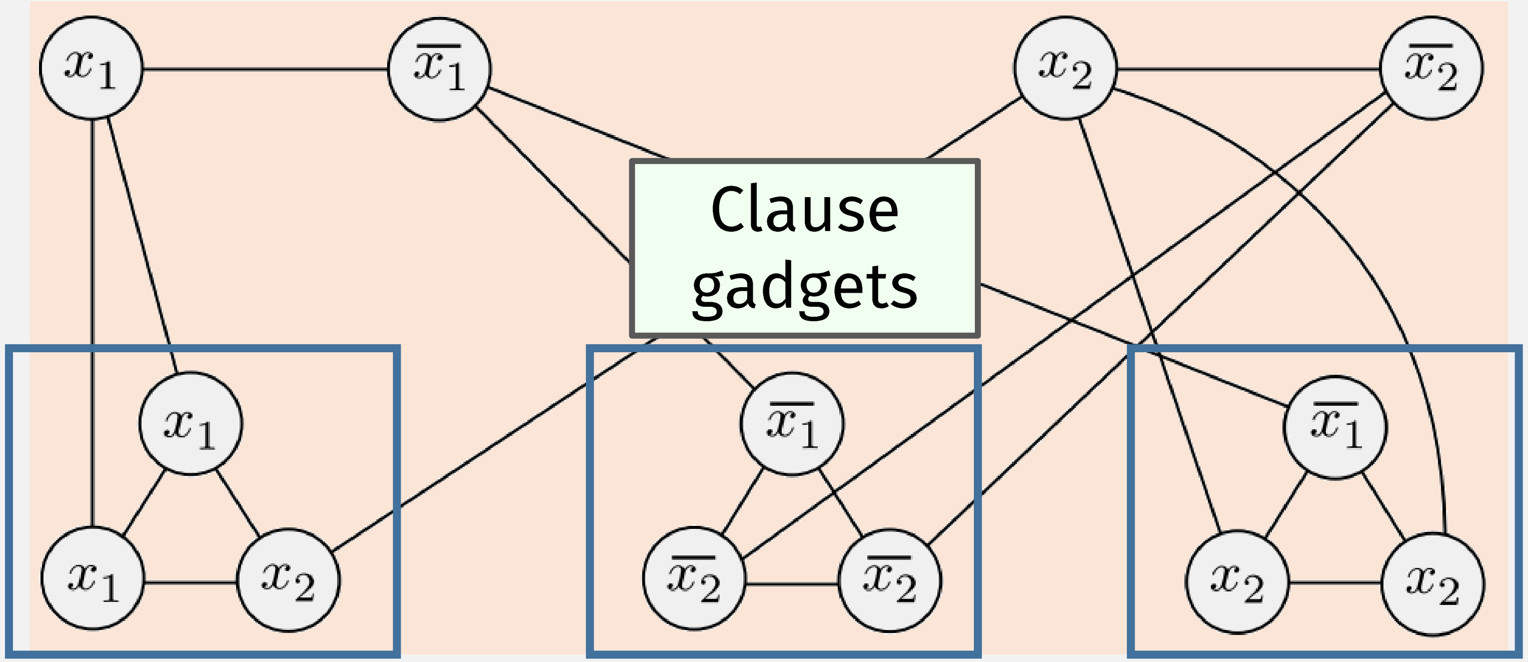
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



VERTEX-COVER example

VERTEX-COVER = $\{\langle G, k\rangle |$ G is an undirected graph that has a k-node vertex cover$\}$

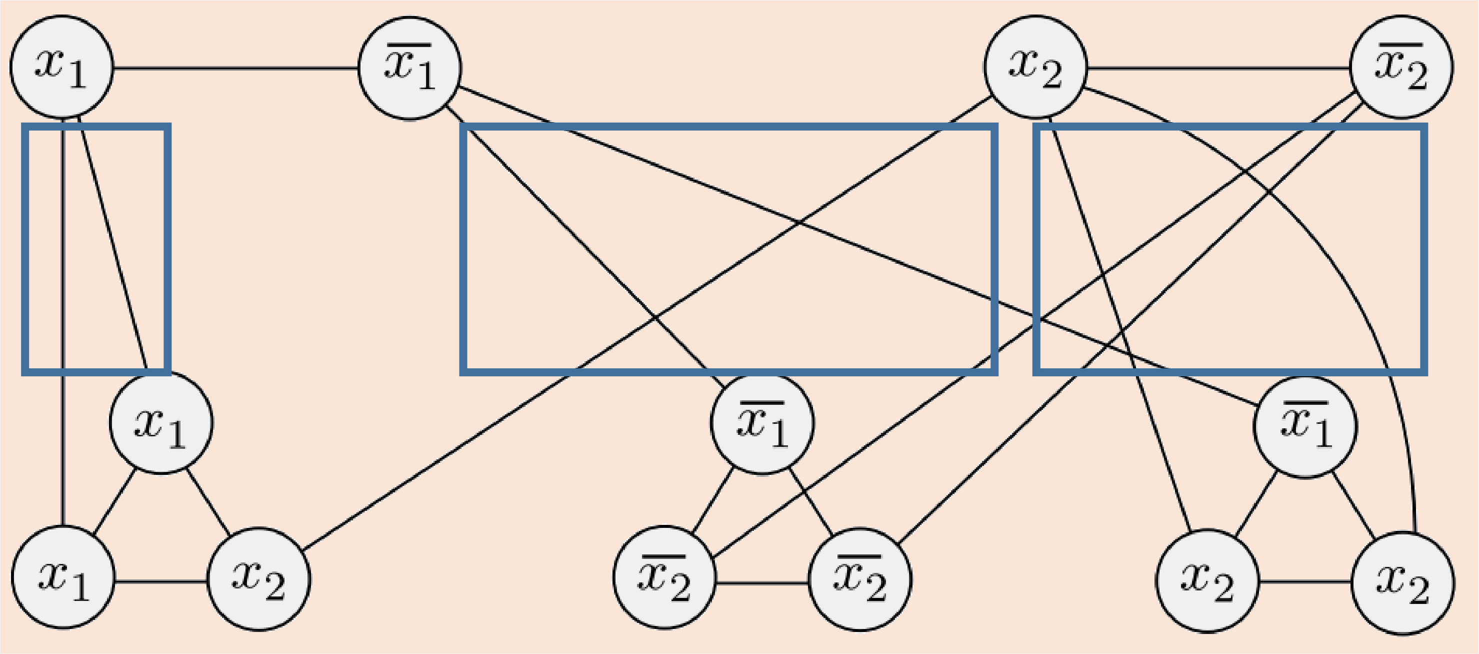
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



VERTEX-COVER example

VERTEX-COVER $= \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

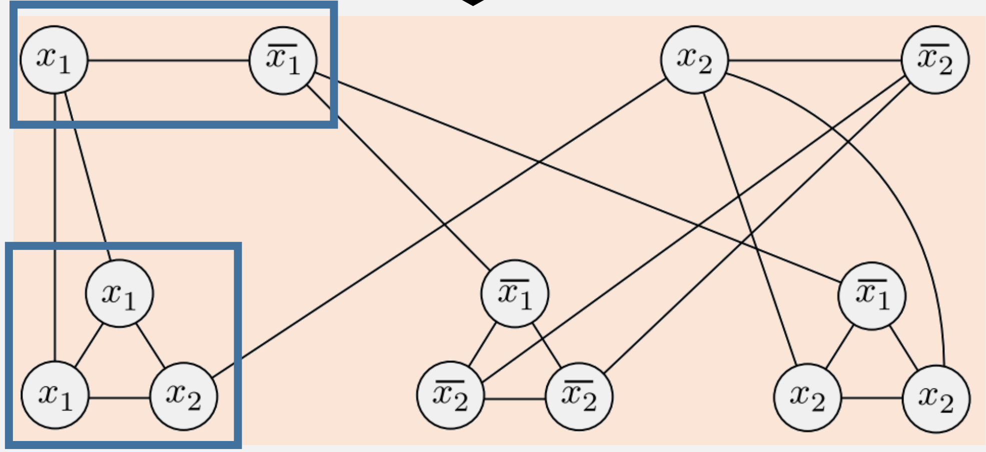
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



Extra edges connecting variable and clause gadgets together

$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

VERTEX-COVER example



- If formula has …
  - ***m*** = # variables
  - ***l*** = # clauses
- Then graph has …
  - # nodes = 2 × #vars + 3 × #clauses = 2***m*** + 3***l***

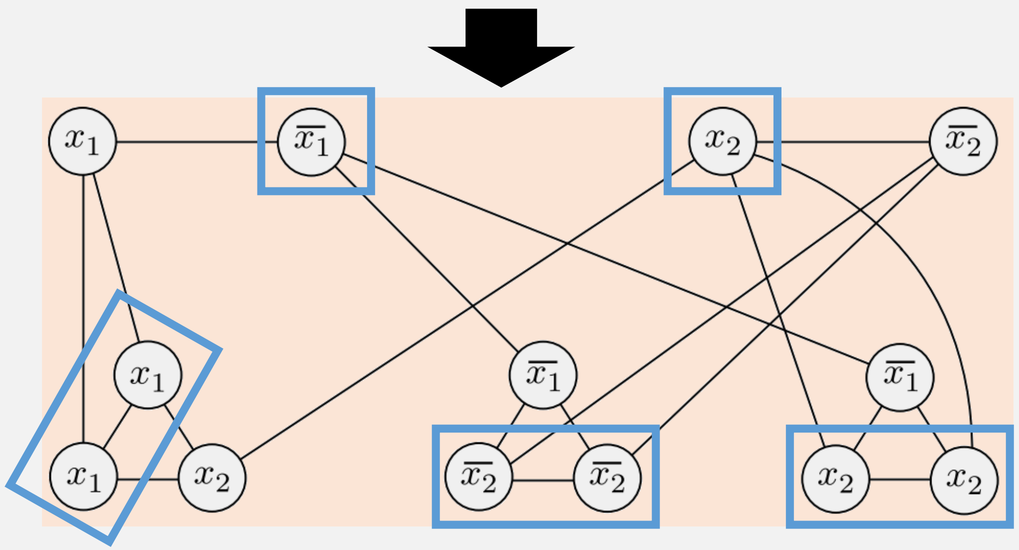
⇒ If satisfying assignment, then there is a ***k***-cover, where ***k*** = ***m*** + **2*l***

- Nodes in the cover are:
  - In each of ***m*** var gadgets, choose 1 node corresponding to TRUE literal
  - For each of ***l*** clause gadgets, ignore 1 TRUE literal and choose other 2
  - Since there is satisfying assignment, each clause has a TRUE literal
  - Total nodes in cover = ***m*** + **2*l***

VERTEX-COVER = $\{\langle G, k\rangle |$ G is an undirected graph that has a k-node vertex cover$\}$

$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

VERTEX-COVER example



- If formula has …
  - ***m*** = # variables
  - ***l*** = # clauses
- Then graph has …
  - # nodes = 2***m*** + 3***l***

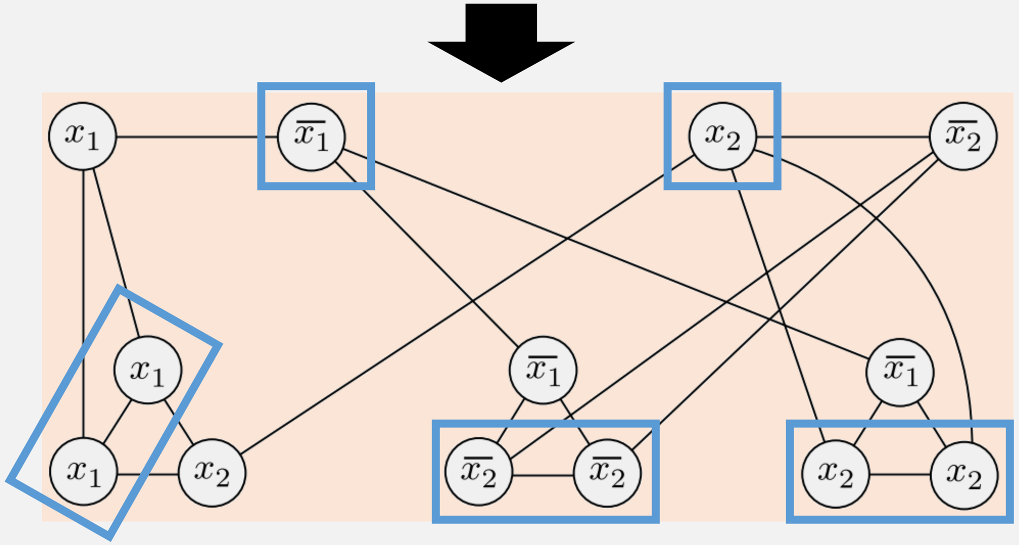
Example:
x_1 = FALSE
x_2 = TRUE

⇒ If satisfying assignment, then there is a ***k***-cover, where ***k* = *m* + 2*l***

- Nodes in the cover are:
  - In each of ***m*** var gadgets, choose 1 node corresponding to TRUE literal
  - For each of ***l*** clause gadgets, ignore 1 TRUE literal and choose other 2
  - Since there is satisfying assignment, each clause has a TRUE literal
  - Total nodes in cover = ***m* + 2*l***

VERTEX-COVER = $\{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



VERTEX-COVER example

- If formula has …
  - ***m*** = # variables
  - ***l*** = # clauses
- Then graph has …
  - # nodes = 2***m*** + 3***l***

Example:
x_1 = FALSE
x_2 = TRUE

⇐ <u>If</u> there is a ***k*** = ***m* + 2*l*** cover,

- Then it can <u>only</u> be a ***k***-cover as described on the last slide …
  - 1 node (and only 1) from each of “var” gadgets
  - 2 nodes (and only 2) from each “clause” gadget
  - <u>Any other set of ***k*** nodes is not a cover</u>
- Which means that input has satisfying assignment:
  - x_i = TRUE if node x_i is in cover, else x_i = FALSE

VERTEX-COVER = $\{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

Last Time: **NP**-Completeness

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

These are the "hardest" problems (in **NP**) to solve

NP-Completeness vs **NP**-Hardness

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

"**NP**-Hard" → (condition 2)

"**NP**-Complete" = in **NP** + "**NP**-Hard"

So a language can be **NP**-hard but not **NP**-complete!

Flashback: The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

Thm: $HALT_{\mathsf{TM}}$ is undecidable

Proof, by contradiction:

- Assume $HALT_{\mathsf{TM}}$ has *decider* R; use it to create decider for A_{TM} :

- …

- But A_{TM} is undecidable and has no decider!

Flashback: The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ halts on input } w\}$

<u>Thm</u>: $HALT_{\mathsf{TM}}$ is undecidable

<u>Proof</u>, by contradiction:

- Assume $HALT_{\mathsf{TM}}$ has *decider* R; use it to create decider for A_{TM}:

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*. ← This means M loops on input w
3. If R accepts, simulate M on w until it halts. ← This step always halts
4. If M has accepted, *accept*; if M has rejected, *reject*."

Flashback: The Halting Problem

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ halts on input } w\}$

<u>Thm</u>: $HALT_{\mathsf{TM}}$ is undecidable

<u>Proof</u>, by contradiction:

- Assume $HALT_{\mathsf{TM}}$ has *decider* R; use it to create decider for A_{TM}:

~~S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:~~
~~**1.** Run TM R on input $\langle M, w\rangle$.~~
~~**2.** If R rejects, *reject*.~~
~~**3.** If R accepts, simulate M on w until it halts.~~
~~**4.** If M has accepted, *accept*; if M has rejected, *reject*."~~

- But A_{TM} is undecidable!
  - I.e., this decider that we just created cannot exist! So $HALT_{\mathsf{TM}}$ is undecidable

The Halting Problem is **NP**-Hard

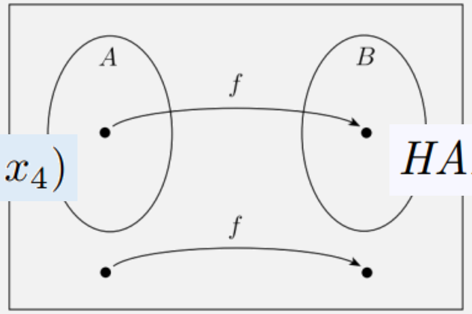
$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Proof: Reduce *3SAT* to the Halting Problem

(Why does this prove that the Halting Problem is **NP**-hard?)

Because *3SAT* is **NP**-complete! (so every **NP** problem is poly time reducible to *3SAT*)

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$$



$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

The Halting Problem is **NP**-Hard

$HALT_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ halts on input } w\}$

Computable function, from $3SAT \rightarrow HALT_{\mathsf{TM}}$:

On input ϕ, a formula in 3cnf:

- Construct TM M

M = on input ϕ

- Try all assignments
  - If any satisfy ϕ, then accept
- When all assignments have been tried, start over

This loops when there is no satisfying assignment!

- Output <M, ϕ >

$\Rightarrow$ If ϕ has a satisfying assignment, then M halts on ϕ
$\Leftarrow$ If ϕ has no satisfying assignment, then M loops on ϕ

Review:

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So a language can satisfy condition #2 but not condition #1

But can a language satisfy condition #1 but not condition #2?

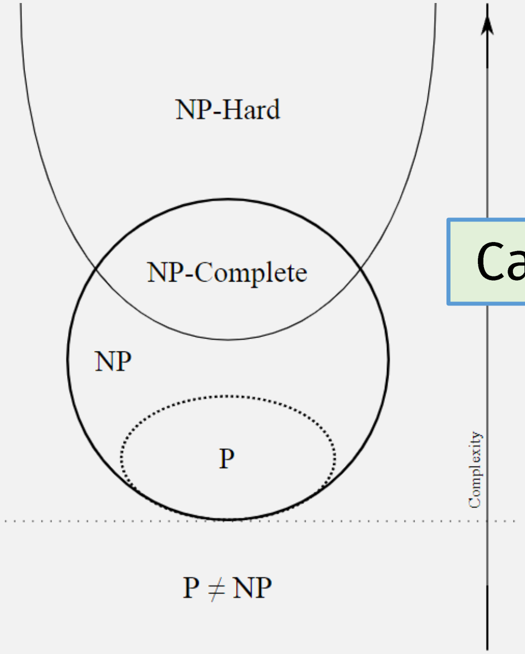
Yes, every language in **P** ...

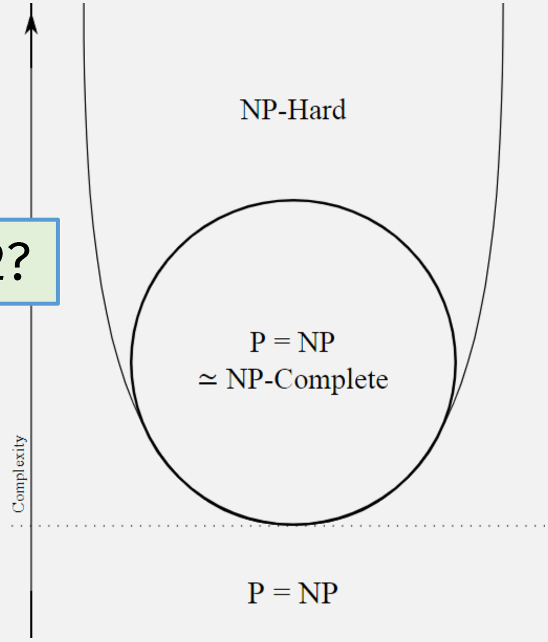
... unless **P** = **NP**

Can a non-**P** language satisfy condition #1 but not condition #2?

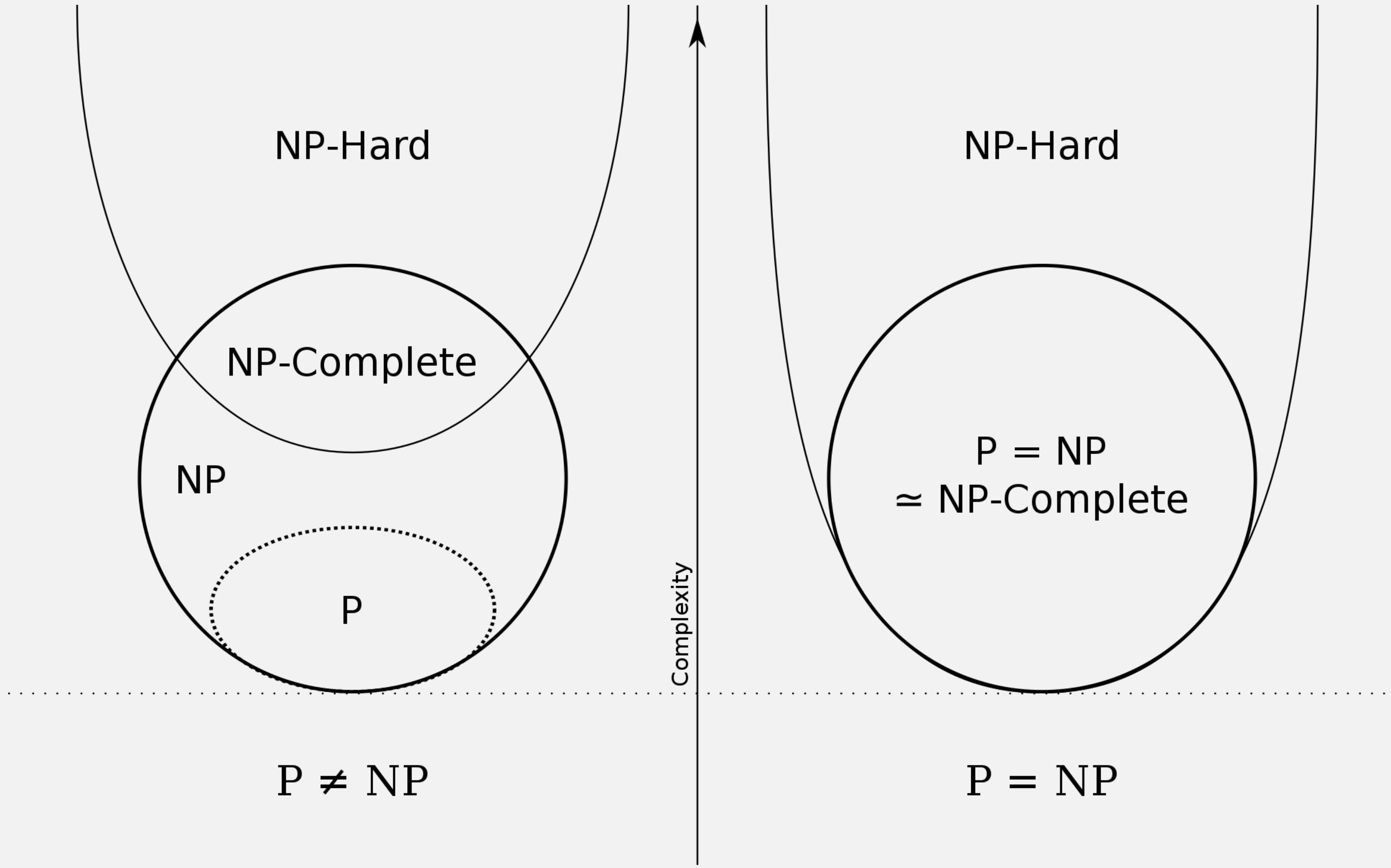
Yes ...

... but that implies **P** ≠ **NP**,
so it's not known for sure





NP-Completeness vs **NP**-Hardness



On to Space …

FINAL REMAINING "FRONTIERS," ACCORDING TO POPULAR USAGE

Flashback: Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

We are gaining time …

… by spending more space!

Substring end char

Substring start char

	b	**a**	**a**	**b**	**a**
b	vars for "b"	vars for "ba"	vars for "baa"	…	
a		vars for "a"	vars for "aa"	vars for "aab"	
a			…		
b					
a					

Space Complexity, Formally

TMs have a **space complexity**

DEFINITION

Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

Space Complexity Classes

Languages are in a **space complexity class**

DEFINITION

Let $f: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. The ***space complexity classes***, **SPACE($f(n)$)** and **NSPACE($f(n)$)**, are defined as follows.

SPACE($f(n)$) = $\{L |$ L is a language decided by an $O(f(n))$ space deterministic Turing machine$\}$.

NSPACE($f(n)$) = $\{L |$ L is a language decided by an $O(f(n))$ space nondeterministic Turing machine$\}$.

Compare:

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, **TIME($t(n)$)**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

NTIME($t(n)$) = $\{L |$ L is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$.

Example: *SAT* Space Usage

$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

$2^{O(m)}$ exponential time machine

$M_1 =$ "On input $\langle \phi \rangle$, where ϕ is a Boolean formula:

1. For each truth assignment to the variables $x_1, \ldots, x_m$ of ϕ:

2. Evaluate ϕ on that truth assignment.

3. If ϕ ever evaluated to 1, *accept*; if not, *reject*."

Each loop iteration requires $O(m)$ space

But the space is re-used on each loop!
(nothing is stored from the last loop)

So the entire machine only needs $O(m)$ space!

Example: Nondeterministic Space Usage

$ALL_{\text{NFA}} = \{\langle A \rangle \mid A \text{ is an NFA and } L(A) = \Sigma^*\}$

Nondeterministic decider for $\overline{ALL_{\text{NFA}}}$

N = "On input $\langle M \rangle$, where M is an NFA:

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M:
3. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.
4. *Accept* if stages 2 and 3 reveal some string that M rejects; that is, if at some point none of the markers lie on accept states of M. Otherwise, *reject*."

Machine tracks "current" states of NFA:
q states = 2^q possible combinations
(so exponential time)

Each loop uses only $O(q)$ space!

Additionally, need a counter to count to 2^q: requires $\log(2^q) = q$ extra space

So the whole machine runs in (nondeterministic) linear $O(q)$ space!

Flashback: TM Variations and Time

- If a multi-tape TM runs in: $t(n)$ time
- Then an equivalent single-tape TM runs in: $O(t^2(n))$
  - **Quadratically** slower

- If a non-deterministic TM runs in: $t(n)$ time
- Then an equivalent deterministic TM runs in: $2^{O(t(n))}$
  - **Exponentially** slower

What about space?

TM Variations and Space

THEOREM

Savitch's theorem For any function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

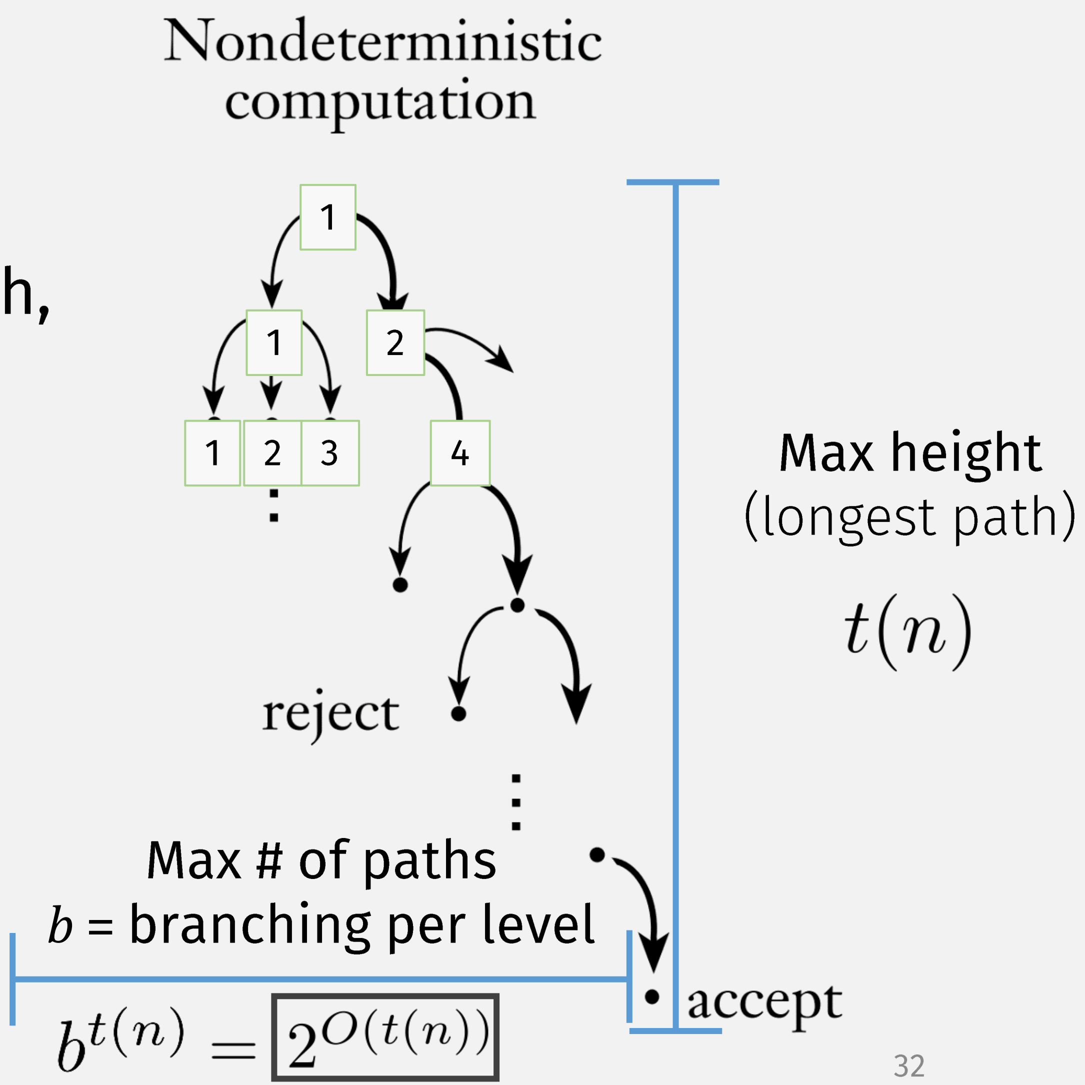
$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- If a non-deterministic TM runs in: $f(n)$ space
- Then an equivalent deterministic TM runs in: $f^2(n)$ space
  - ~~**Exponentially**~~ Only **Quadratically** slower!

Flashback: Nondet. TM → Deterministic TM

$t(n)$ time → $\boxed{2^{O(t(n))}}$ time

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - 1
    - 1-1
    - 1-2
    - **1-1-1**



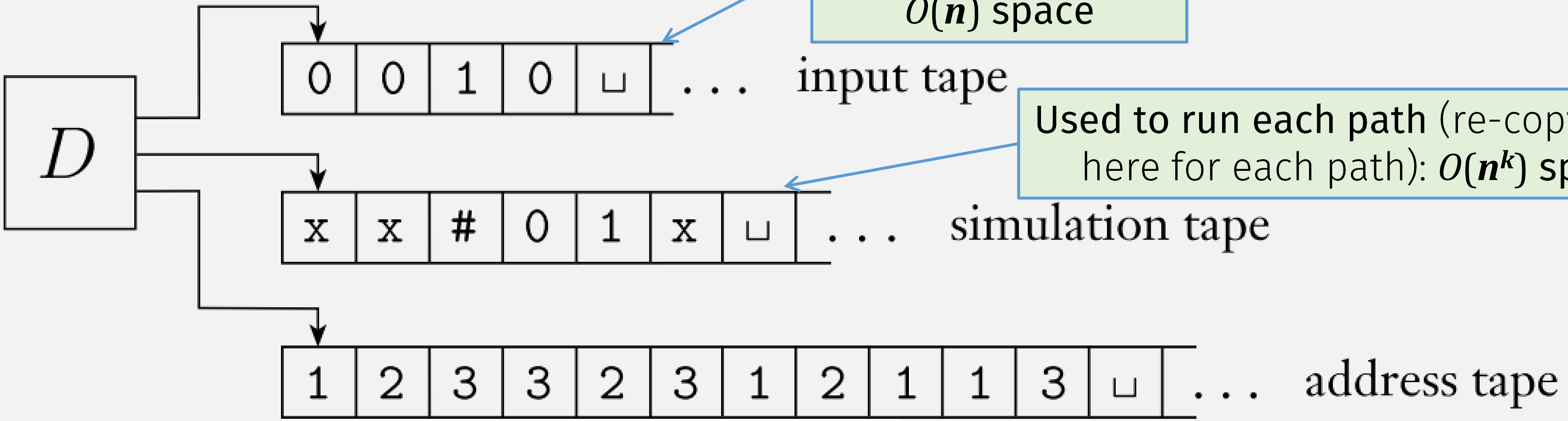
Flashback: NTM → Deterministic

3 tapes

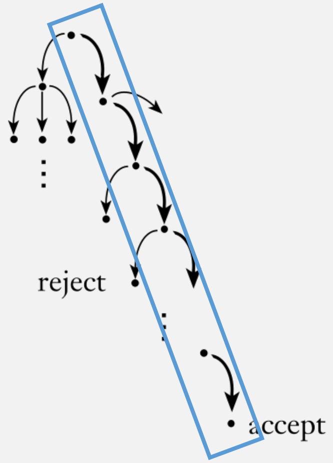
Always has input, never changes: $O(n)$ space

Used to run each path (re-copy input here for each path): $O(n^k)$ space

Tracks which node we are on, e.g., 1-1-2, etc: $2^{O(n)}$ space??

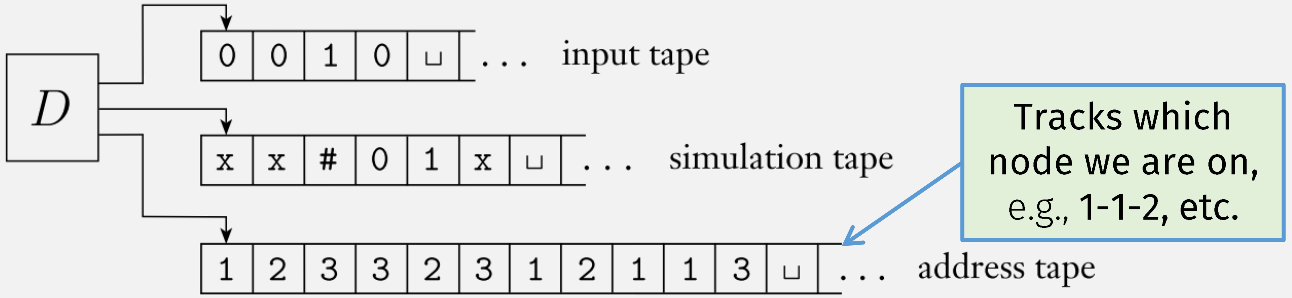


NTM→Deterministic TM: Space Version



Let N be an NTM deciding language A in space $f(n)$

- This means a single path could use $f(n)$ space
- That path could take $2^{O(f(n))}$ steps
  - (That's the possible ways to fill the space)
  - Where each step could be a branch
- So naïvely tracking these branches requires $2^{O(f(n))}$ space!



- Instead, let's "divide and conquer" to save space!

"Divide and Conquer" TM Config Sequences

- Want to check whether:

TM configs → c_{start} —— $2^{O(f(n))}$ (possibly branching) **steps** ——→ c_{accept}

Remembering the branch at every step costs exponential space

This requires: $\log(2^{O(f(n))}) = O(f(n))$ **splits**

Each split must remember 1 configuration "c_m" = $O(f(n))$ **space**

- Instead, we check whether:

So long as we save the intermediate config

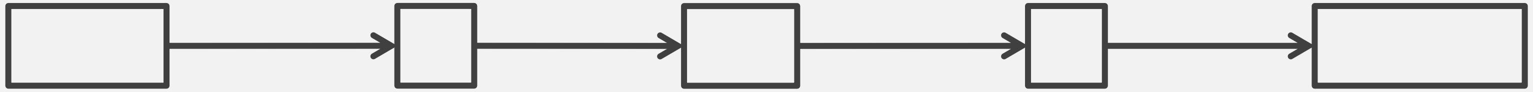
c_{start} —— $2^{O(f(n))}/2$ steps ——→ c_m —— $2^{O(f(n))}/2$ steps ——→ c_{accept}

Remembering these steps costs half the space …

… and we can reuse that space to check the second half

Total:
$O(f(n)) * O(f(n))$
= $\boldsymbol{O(f^2(n))}$ **space**

- Keep dividing …



Formally: A "Yielding" Algorithm

Start config | End config | # steps

CANYIELD = "On input c_1, c_2, and t:

1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether c_1 yields c_2 in one step according to the rules of N. *Accept* if either test succeeds; *reject* if both fail.
2. If $t > 1$, then for each configuration c_m of N using space $f(n)$:
3. Run CANYIELD$(c_1, c_m, \frac{t}{2})$.
4. Run CANYIELD$(c_m, c_2, \frac{t}{2})$.
5. If steps 3 and 4 both accept, then *accept*.
6. If haven't yet accepted, *reject*."

What's the middle config? Try them all (it doesn't use any more space, per loop)

"divide and conquer"

Savitch's Theorem: Proof

- Let N be an NTM deciding language A in space $f(n)$
- Construct equivalent deterministic TM M using $O(f^2(n))$ space:

> M = "On input w:
>
> **1.** Output the result of CANYIELD$(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$."

Extra d constant depends on size of tape alphabet

  - c_{start} = start configuration of N
  - c_{accept} = new accepting config where all N's accepting configs go

PSPACE

DEFINITION

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

NPSPACE

DEFINITION

NPSPACE is the class of languages that are decidable in polynomial space on a nondeterministic Turing machine. In other words,

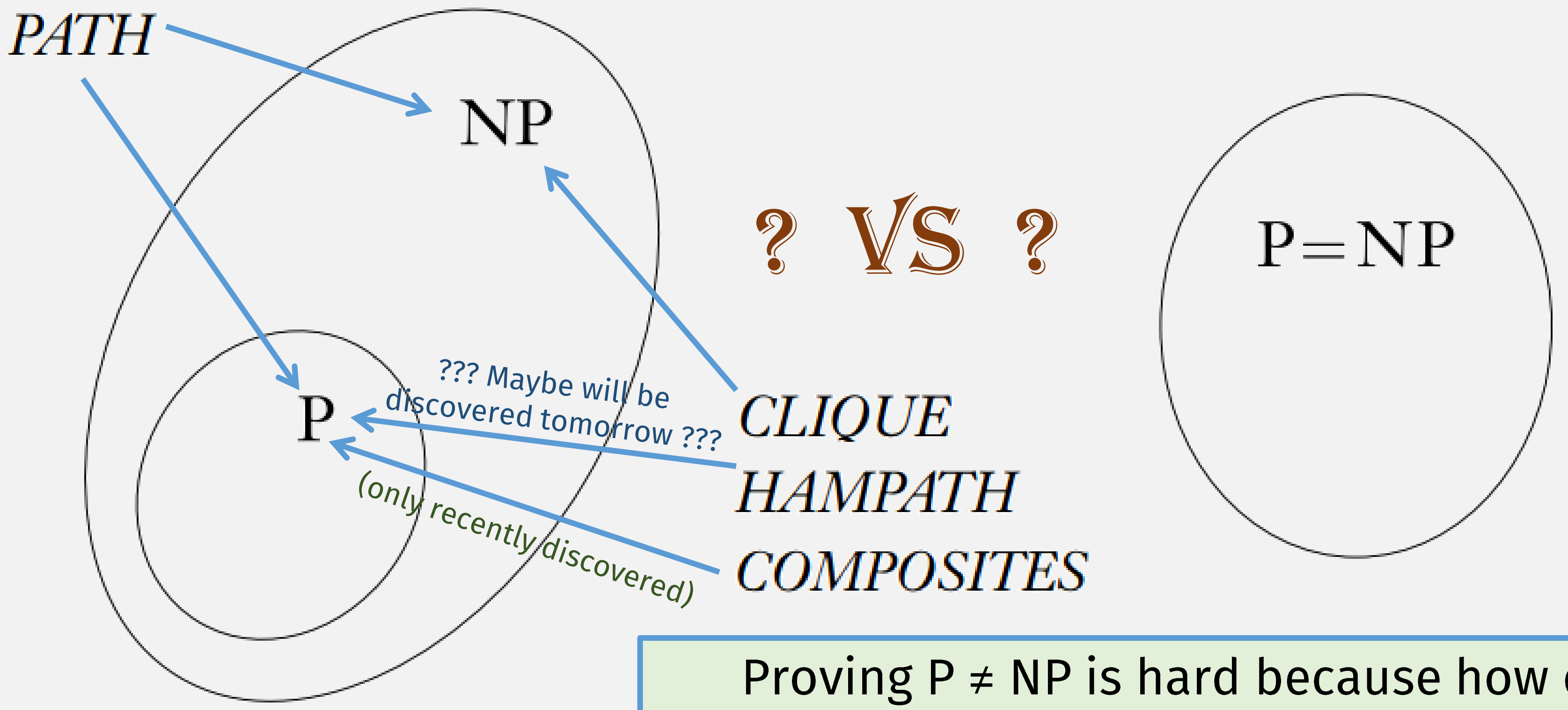
$$\text{NPSPACE} = \bigcup_k \text{NSPACE}(n^k).$$

Analogous to **P** and **NP** for time complexity

PSPACE vs NPSPACE

- **PSPACE**: langs decidable in poly space on <u>deterministic</u> TM
- **NPSPACE**: langs decidable in poly space on <u>nondeterministic</u> TM

Flashback: Does P = NP?



Proving P ≠ NP is hard because how do you prove an algorithm <u>doesn't</u> have a poly time algorithm?
(in general it's hard to prove that something <u>doesn't</u> exist)

PSPACE vs NPSPACE

- **PSPACE**: langs decidable in poly space on <u>deterministic</u> TM
- **NPSPACE**: langs decidable in poly space on <u>nondeterministic</u> TM

<u>Theorem</u>: **PSPACE = NPSPACE** !!!

<u>Proof</u>: By Savitch's Theorem!

THEOREM

Savitch's theorem For any function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f^2(n)).$$

Space vs Time

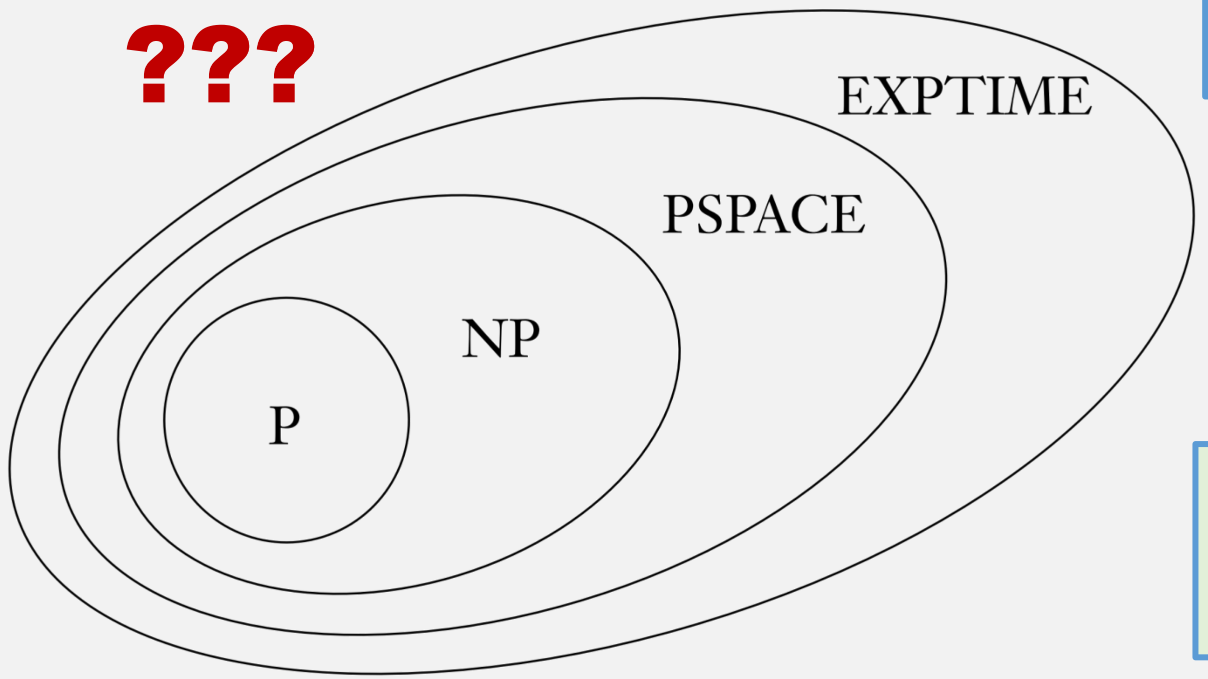
- $\mathbf{P} \subseteq \mathbf{PSPACE}$ and $\mathbf{NP} \subseteq \mathbf{NPSPACE}$
  - Because each step can use at most one extra tape cell
  - And space can be re-used

- $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$
  - Because an $f(n)$ space TM has $2^{O(f(n))}$ possible configurations
  - And a halting TM cannot repeat a configuration

- We already know $\mathbf{P} \subseteq \mathbf{NP}$ and $\mathbf{PSPACE} = \mathbf{NPSPACE}$ … so:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}$$

Space vs Time: Conjecture



Researchers believe these are all completely contained within each other

But this is an open conjecture!

The only progress so far is:
P ⊂ EXPTIME
(we will prove next week)

P ⊂ NP ⊂ PSPACE = NPSPACE ⊂ EXPTIME

No quiz 11/24!