COMPLEXITY THEORY

DISCRETE MATHEMATICS AND THEORY 2

MARK FLORYAN

GOALS!

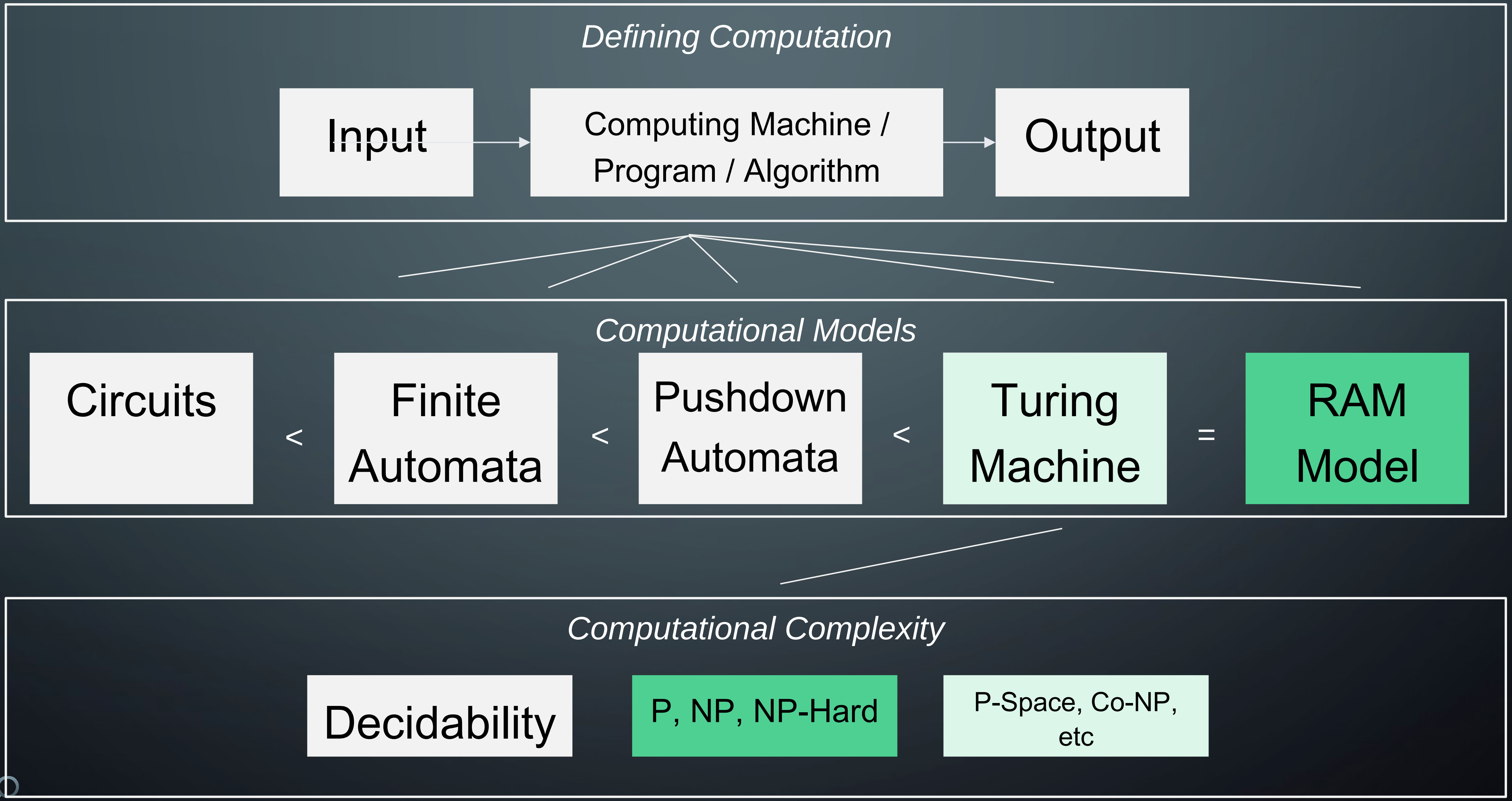
1. Measuring Time and Space complexity of algorithms on Turing Machines (You already know a lot of this!)

2. Introducing the most famous complexity classes (P, NP, NP-Hard, etc.)

3. Showing how a difficult a problem is through the use of mapping reductions (you’ve already seen some of this in DSA2)!

PART 1: INTRODUCTION!

OVERVIEW OF THEORY OF COMPUTATION



PART 1: MEASURING TIME AND SPACE COMPLEXITY

TIME COMPLEXITY

Let be a deterministic Turing machine that halts on all inputs. The running time or time complexity of is the function , where is the maximum number of steps that uses on any input of length . If is the running time of , we say that runs in time and that is an time Turing machine. Customarily we use to represent the length of the input.

You should already be familiar with this definition / concept

Short version: is the worst case runtime for machine as a function of input size .

REVIEW: TIME COMPLEXITY

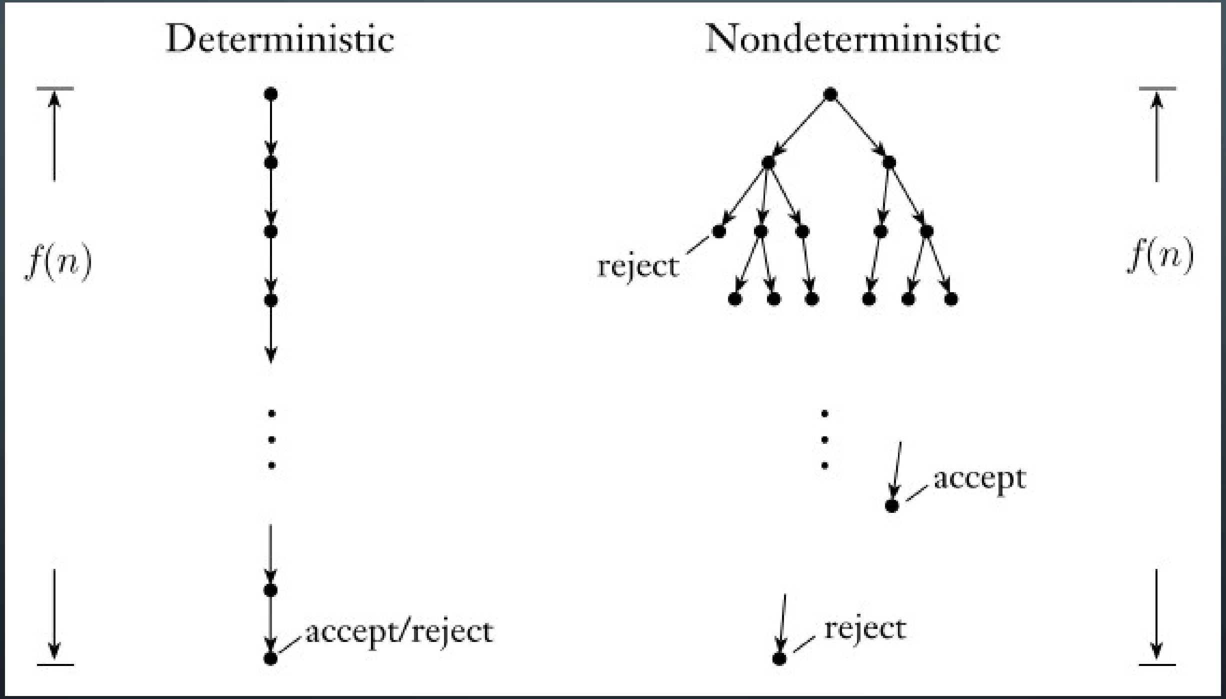
The following items, you should already know from previous courses.

,	*Asymptotic **upper** bounds*
,	*Asymptotic **lower** bounds*
$\Theta(f(n))$	*Asymptotic **tight** bound*
$1, \log(n), n, n\log(n), n^2, n^3$	*Some common complexity classes*
$\log_a n \in o(n^b) \in o(c^n)$	*Every log is bounded by any polynomial is bounded by any exponential*

QUICK NOTE ON NON-DETERMINISTIC TIME

What about ***non-deterministic*** Turing machines (NTMs)? How do we measure running time of such a device?

With deterministic computation, we simply look at longest the one branch of computation can possibly be.



For non-deterministic deciders (does not loop forever), we measure the length of the longest branch of computation

QUICK NOTE ON NON-DETERMINISTIC TIME

Theorem: Every NTM that runs in time has an equivalent DTM that runs in time

COMPARING NTM AND DTM

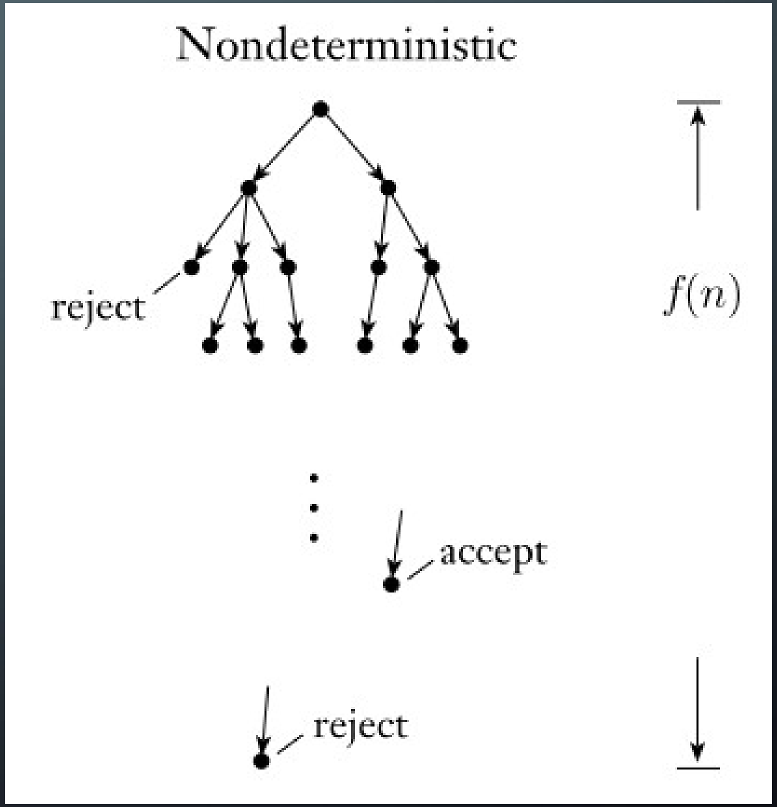
Theorem: Every NTM that runs in time has an equivalent DTM that runs in time

let be the maximum number of branches this computation can have

The computation tree has at most leaves and each branch to each node has length at most

Construct a DTM with three tapes that simulates this NTM as we did in the Turing Machine section earlier. This machines manually computes / simulates each branch individually.

Thus, this machine simulates branches at time each for total time



Here, is the longest branch of computation

PART 1: COMPLEXITY CLASSES

PROBLEM TYPES

PROBLEM TYPES

Given a problem we want to solve, there are three important variations of that problem

Traveling Salesperson Problem: Given a weighted graph G and start node s, find the minimum weight path starting and ending at s that visits every node exactly once.

Function Problem:
Return the actual solution

Given G and s, return the weight of the path P (or maybe the list of nodes to visit) that minimizes the sum of the weights of the edges along P.

Decision Problem:
Convert problem to have Boolean output

Given G, s, and integer k, can you find a valid path with total weight less than or equal to k?

Verification Problem:
Given a solution, verify if it works

Given G, s, path P, and integer k

Is path P valid and is it weight less than k?

WHY DO THESE MATTER?

Function Problem:
Return the actual solution

Given G and s, return the weight of the path P (list of nodes to visit in order) that minimizes the sum of the weights of the edges along P.

Decision Problem:
Convert problem to have Boolean output

Given G, s, and integer k, can you find a valid path with total weight less than k?

Verification Problem:
Given a solution, verify if it works

Given G, s, path P, and integer k

Is path P valid and is it weight less than k?

If you can solve the decision problem you can also solve the function problem Why?

Because if you can solve the decision problem, you can repeatedly invoke it with lower values of k until the Yes responses change to No

WHY DO THESE MATTER?

Function Problem:
Return the actual solution

Given G and s, return the weight of the path P (list of nodes to visit in order) that minimizes the sum of the weights of the edges along P.

Decision Problem:
Convert problem to have Boolean output

Given G, s, and integer k, can you find a valid path with total weight less than k?

Verification Problem:
Given a solution, verify if it works

Given G, s, path P, and integer k

Is path P valid and is it weight less than k?

Answer: Yes! If verifier exists, we can call the verifier over and over again with possible paths until we get a Yes response. We will see soon though that this is usually NOT efficient

If you can solve the verification problem, does it help you solve the decision problem?

WHY DO THESE MATTER?

Function Problem:
Return the actual solution

Given G and s, return the weight of the path P (list of nodes to visit in order) that minimizes the sum of the weights of the edges along P.

Decision Problem:
Convert problem to have Boolean output

Given G, s, and integer k, can you find a valid path with total weight less than k?

Verification Problem:
Given a solution, verify if it works

Given G, s, path P, and integer k

Is path P valid and is it weight less than k?

We will focus on these two from now on because Turing machines return Yes/No answers.

A NOTE ON VERIFICATION

Verification Problem:
Given a solution, verify if it works

Given G, s, path P, and integer k

Is path P valid and is it weight less than k?

Verification is technically more general than “given a solution, verify it if works”.

Formal Definition: Given a string w and certificate c, use c as proof to verify that w is in the language.

Given a language A, a verifier V is correct if and only if accepts

COMPARING NTM AND DTM

Theorem: A problem P is verifiable in polynomial time by a DTM if and only if it is solvable (decision problem) in polynomial time by an NTM

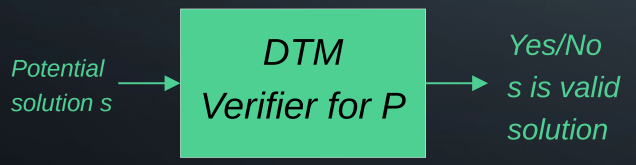
Here, polynomial time means the runtime of the machine is worst-case for

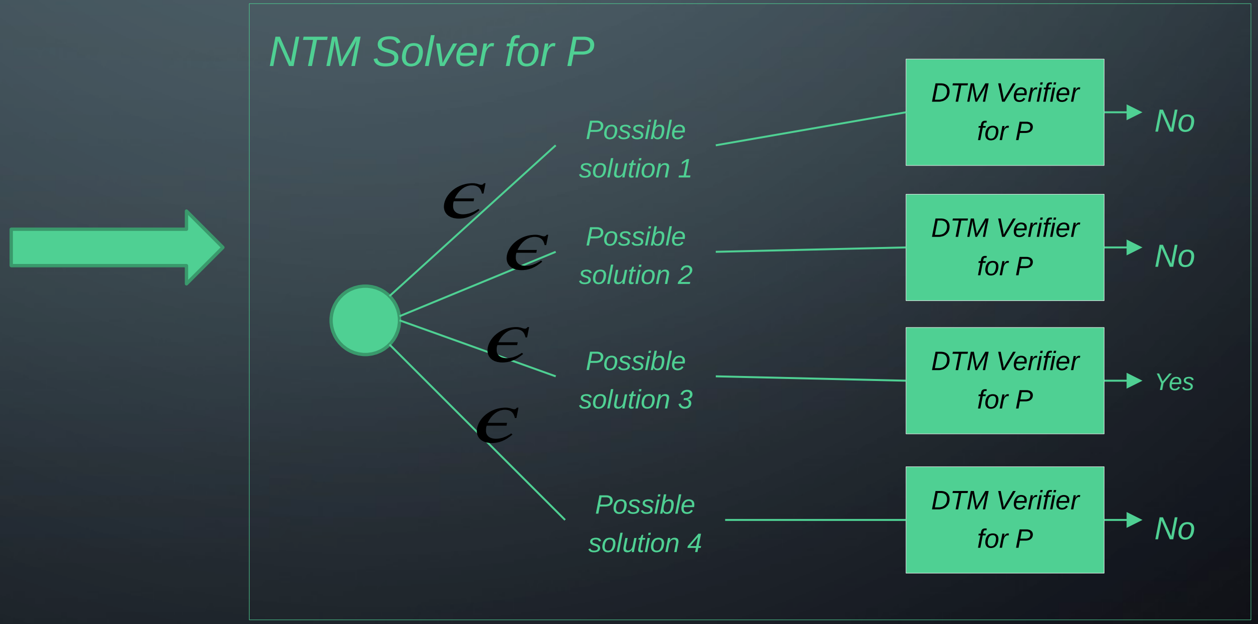
COMPARING NTM AND DTM

Theorem: A problem P is verifiable in polynomial time by a DTM if and only if it is solvable (decision problem) in polynomial time by an NTM

Direction 1: *If a problem is verifiable by a DTM in polynomial time, then it is solvable in polynomial time by an NTM.*

Given: P is verifiable by a DTM. Thus, the DTM that verifies instances of this problem exists



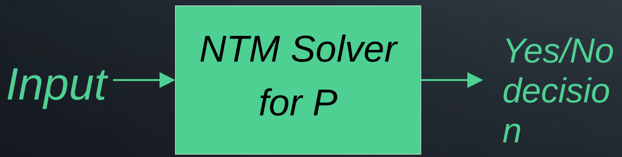


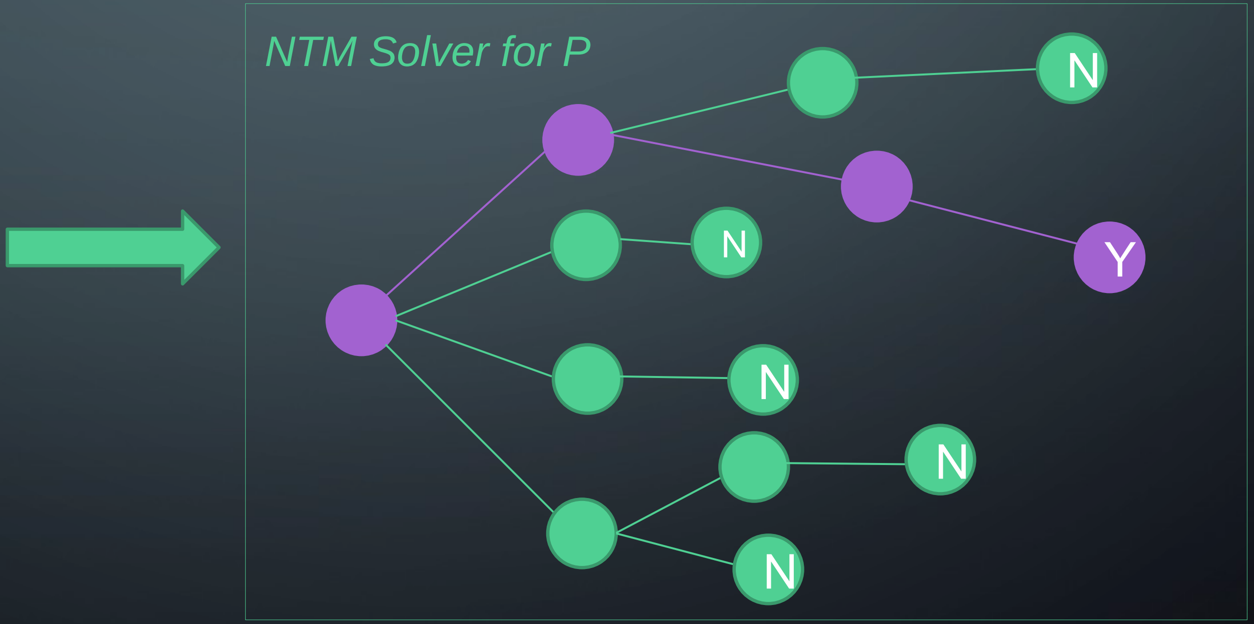
COMPARING NTM AND DTM

Theorem: A problem P is verifiable in polynomial time by a DTM if and only if it is solvable (decision problem) in polynomial time by an NTM

Direction 2 (Harder): *If a problem is solvable by an NTM in polynomial time, then it is verifiable in polynomial time by a DTM.*

Given: P is solvable by an NTM. Thus, the NTM that exists



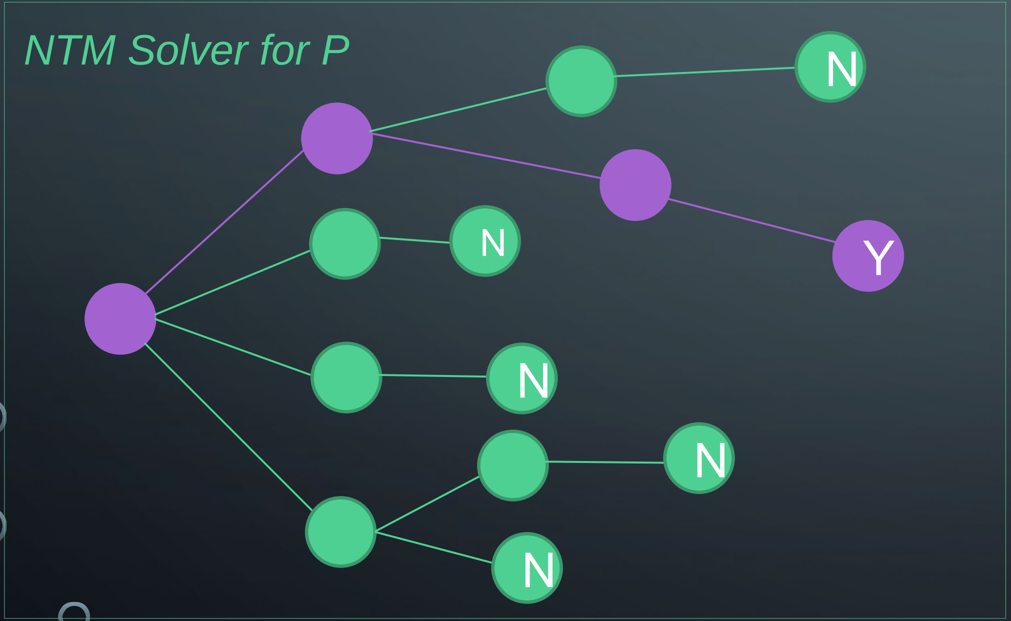


Purple path that leads to Yes is a certificate for P. Why?

COMPARING NTM AND DTM

Theorem: A problem P is verifiable in polynomial time by a DTM if and only if it is solvable (decision problem) in polynomial time by an NTM

Direction 2 (Harder): *If a problem is solvable by an NTM in polynomial time, then it is verifiable in polynomial time by a DTM.*



Verifier for this language:

Given w (input) and c (list of which branch to take at each step

Simulate P

At each step, check c to see which branch to take

Accept iff P accepts

COMPARING NTM AND DTM

Theorem: A problem P is verifiable in polynomial time by a DTM if and only if it is solvable (decision problem) in polynomial time by an NTM

This theorem is critical to remember! It will be very important in a moment.

COMPLEXITY CLASSES (FINALLY!)

THE CLASS P

Important: P is a set of problems (not solutions, not algorithms)

The class P is the set of all problems that can be solved by a deterministic Turing machine in time such that

***Example problems in this set include*:**

Sorting a list of numbers
Inserting into a binary tree
Computing the average of a list of numbers
Printing "hello world"
Find() in a hash table
…and many more

THE CLASS NP

***Remember**: We also showed that any NTM solver has an equivalent exponential time DTM. So all problems in NP are solvable in exponential time.*

***Example problems in this set include**:*

Everything in P (will prove shortly)
Traveling Salesperson Problem
Circuit Satisfiability
Vertex Cover
Independent Set
Subset Sum
…and many more

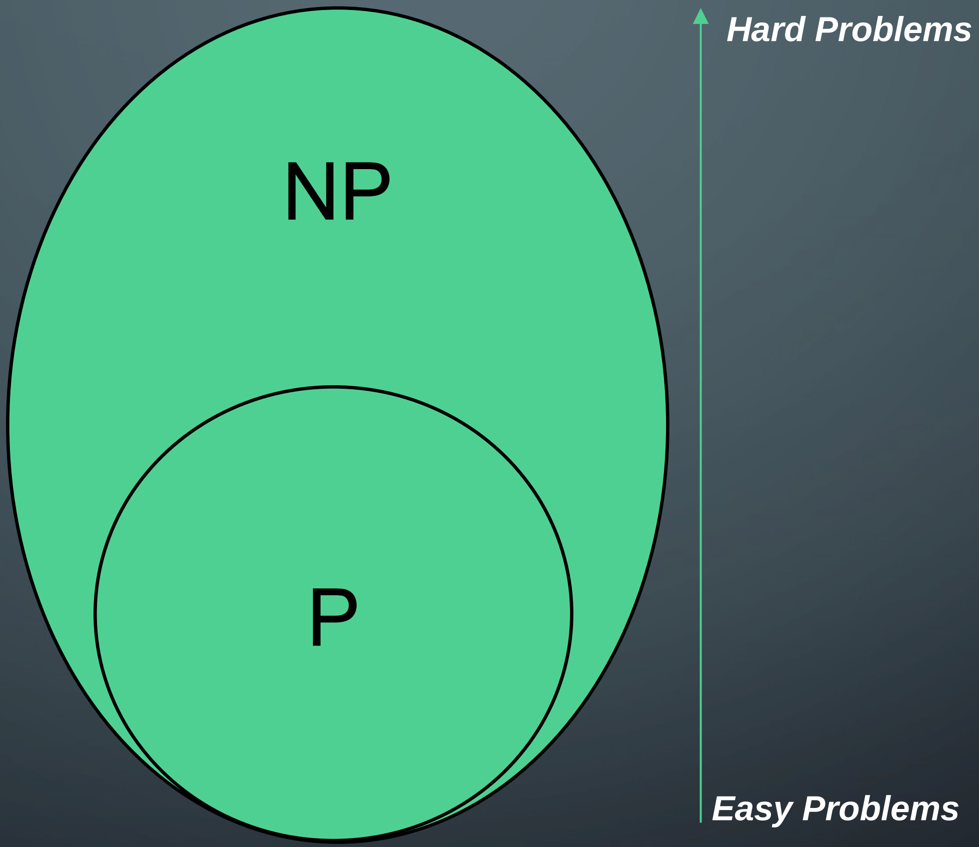
***Equivalent Definition**: By our recently proved theorem, this also means these problems can be verified in polynomial time using a deterministic Turing machine!*

The class NP is the set of all problems that can be solved by a **non-deterministic** Turing machine in time such that

$P \subseteq NP$

Proof:

Everything in P can be solved in polynomial time by a DTM, so it can definitely be verified as well (just solve it and then verify the solution)

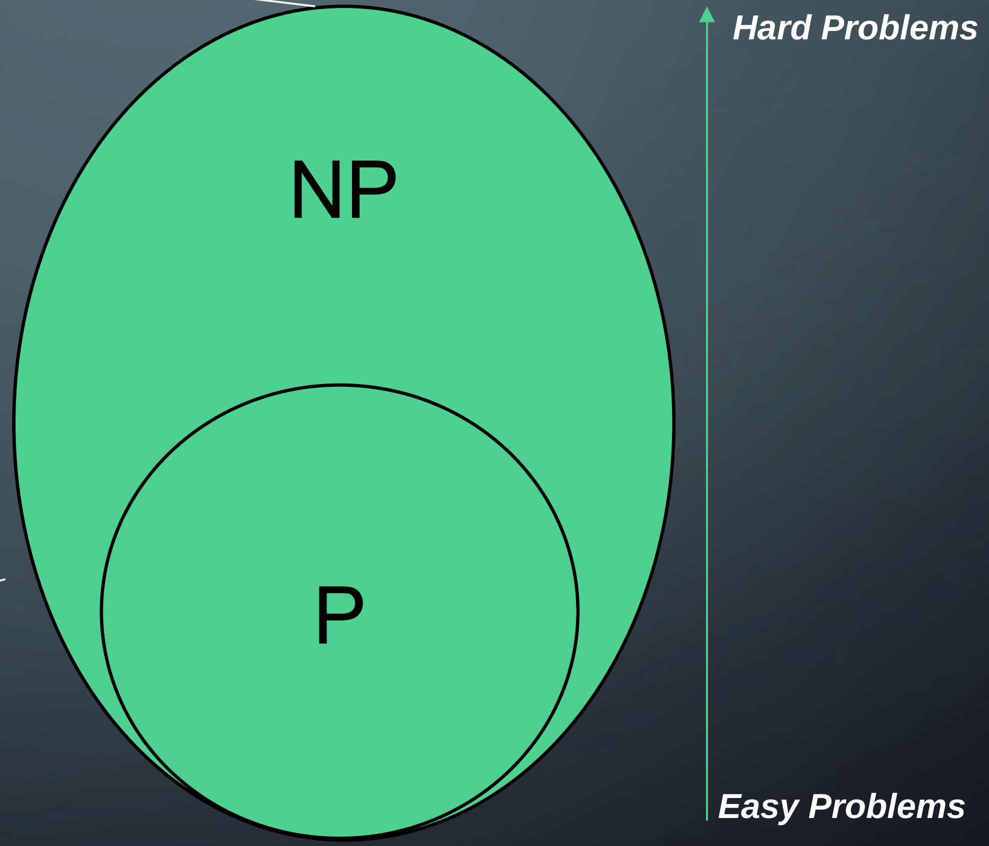


Is ? This is still unknown today!

$P \subseteq NP$

We are interested in finding the hardest problem in NP (at the VERY top of the bubble). Why? It is the MOST likely to not be in P if

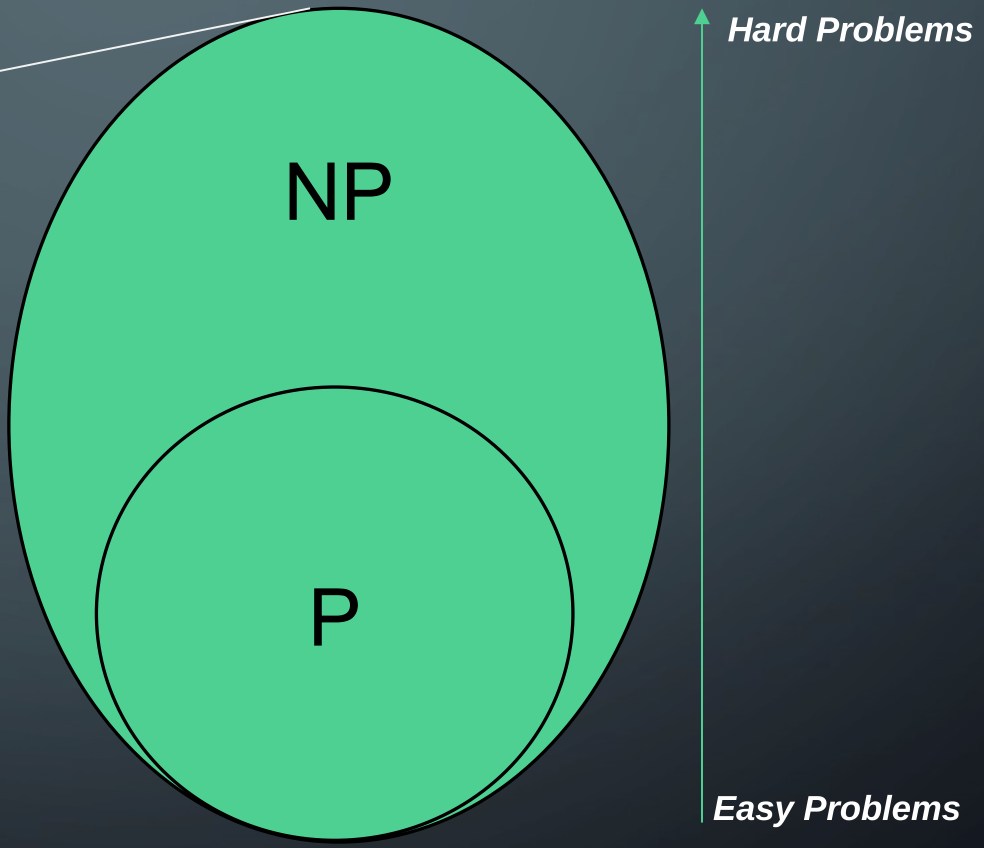
It is true that we DO NOT know if there are actually any unique problems in NP (that are not also in P).



NP-HARD

Suppose we have find the hardest problem in NP

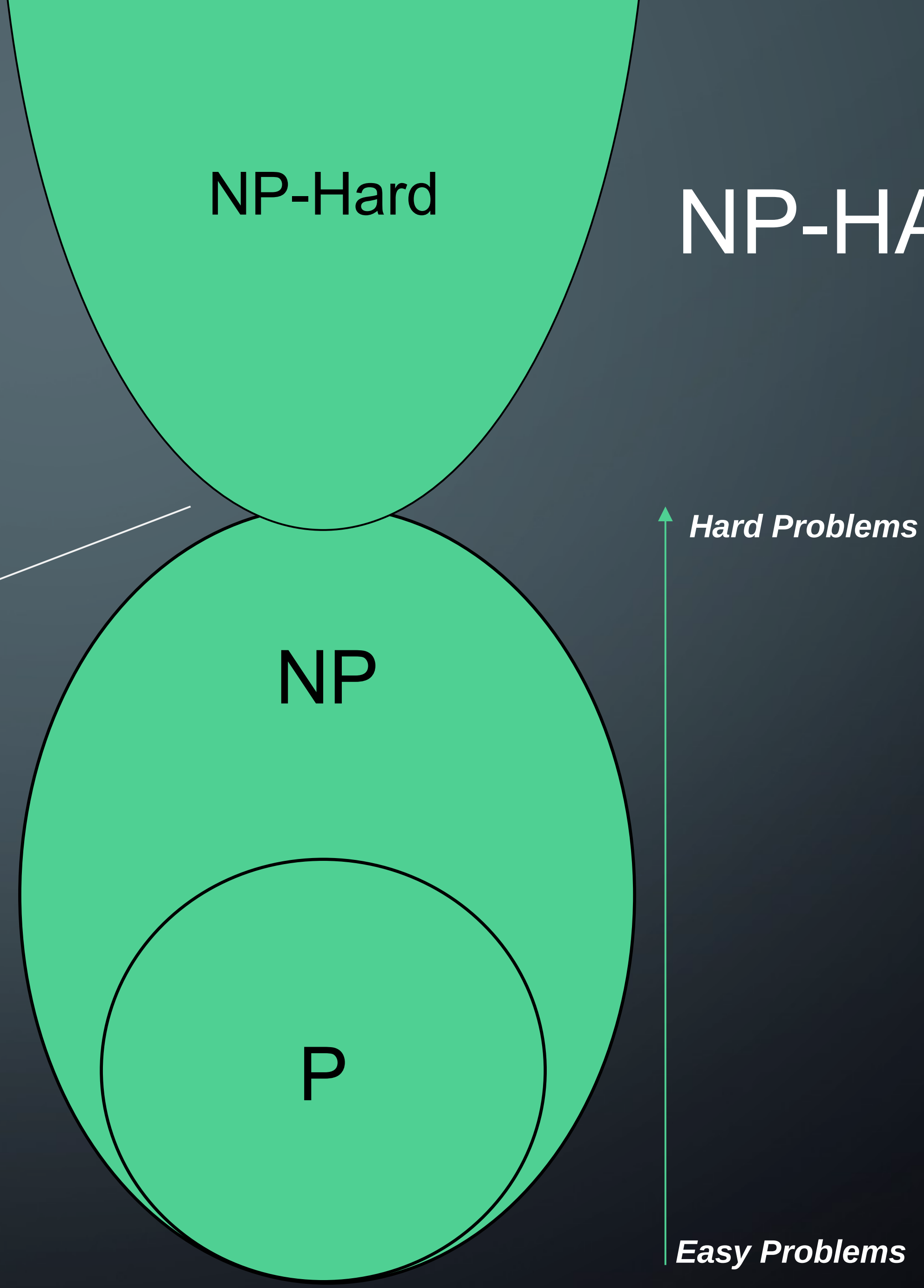
NP-Hard problems are defined to be all problems that are this hard OR harder.



NP-HARD

Goes up to indefinite difficulty.

Note that NP-Hard and NP intersect here. Problems in this intersection are the hardest problems in NP

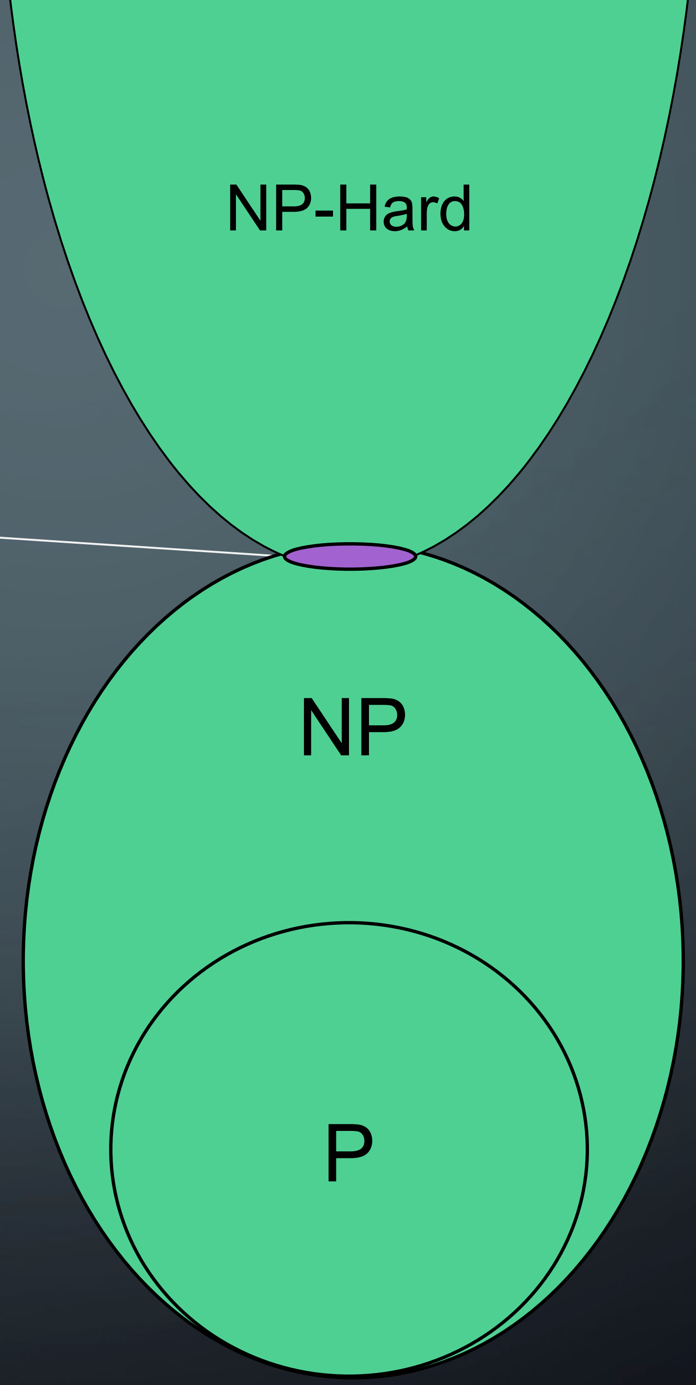


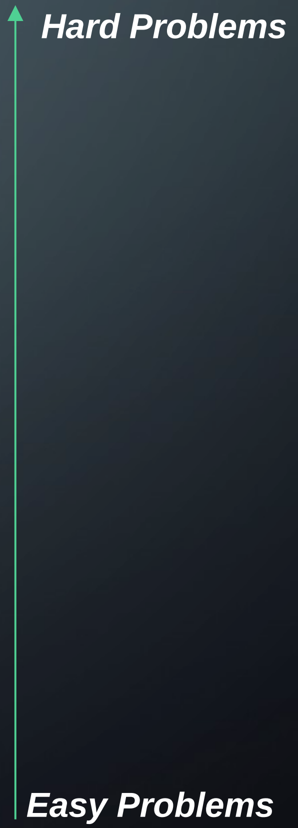
NP-COMPLETE

This section (purple) is the set of NP-Complete problems. The hardest problems in NP

Definition: *A problem is* ***NP-Complete*** *if and only if the problem:*

1. *Is in NP*
2. *Is NP-Hard*



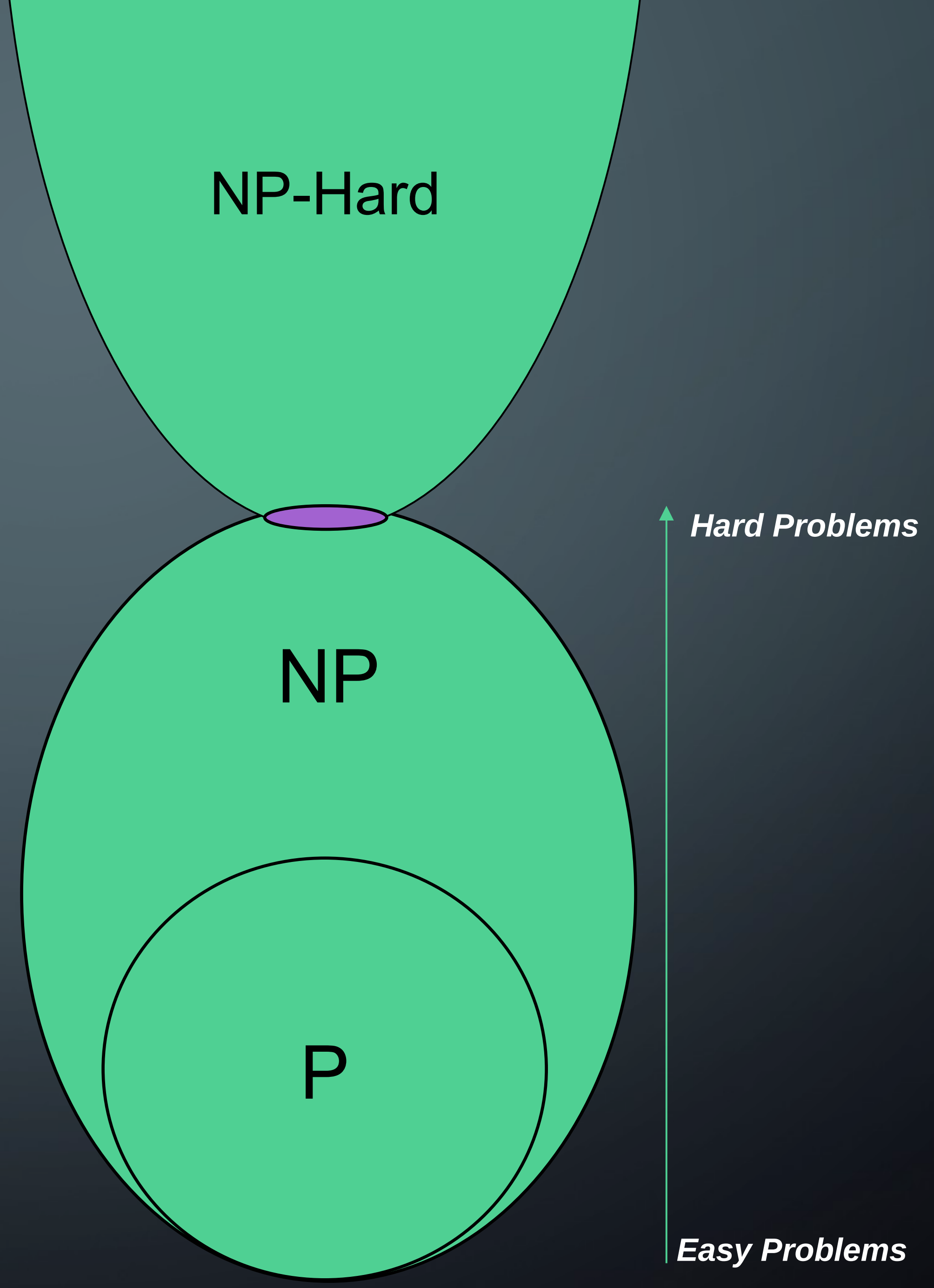


NP-COMPLETE

A different definition of NP-Complete

***Definition**: A problem A is **NP-Complete** if and only if*

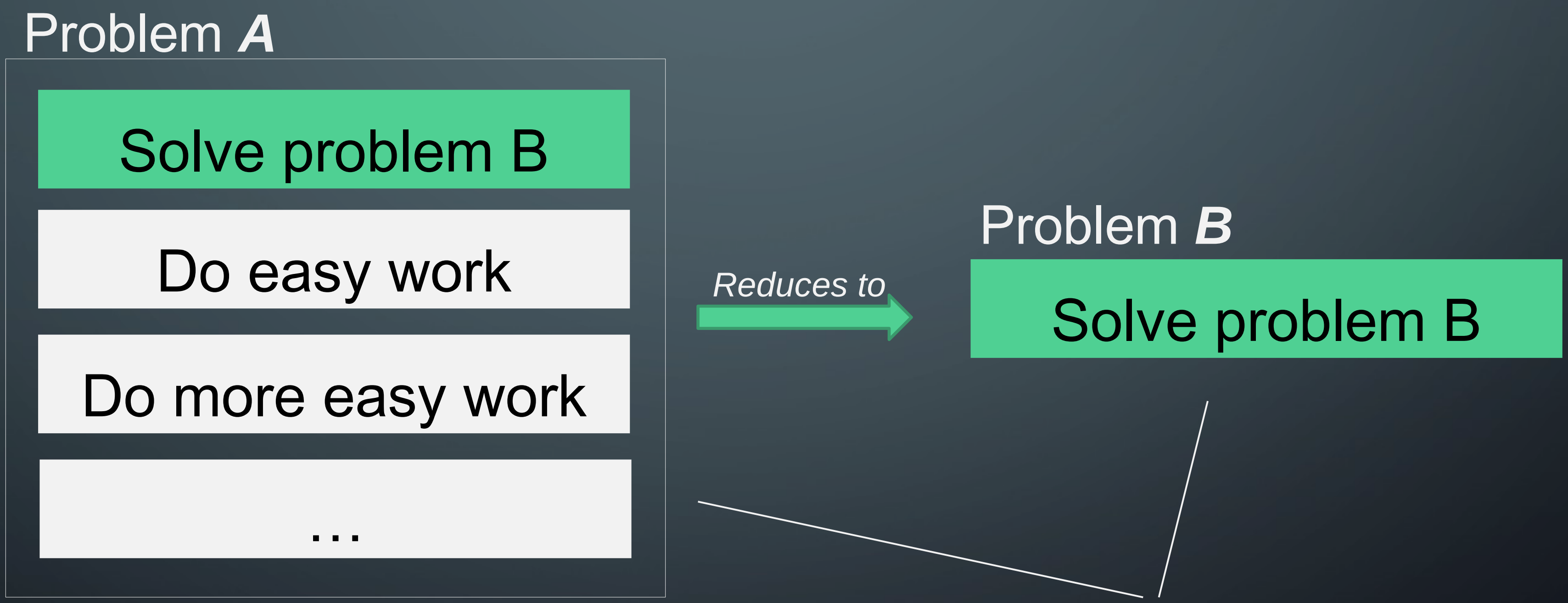
*means that problem A is harder than problem B, shown through a **reduction**, which we will see in a moment.*



MORE ON REDUCTIONS: MAPPING REDUCTIONS

WHAT WE HAVE ALREADY SEEN

Reduction: A reduction exists between problems ***A*** and ***B*** if a solution to ***B*** can be used to develop a solution for ***A***.

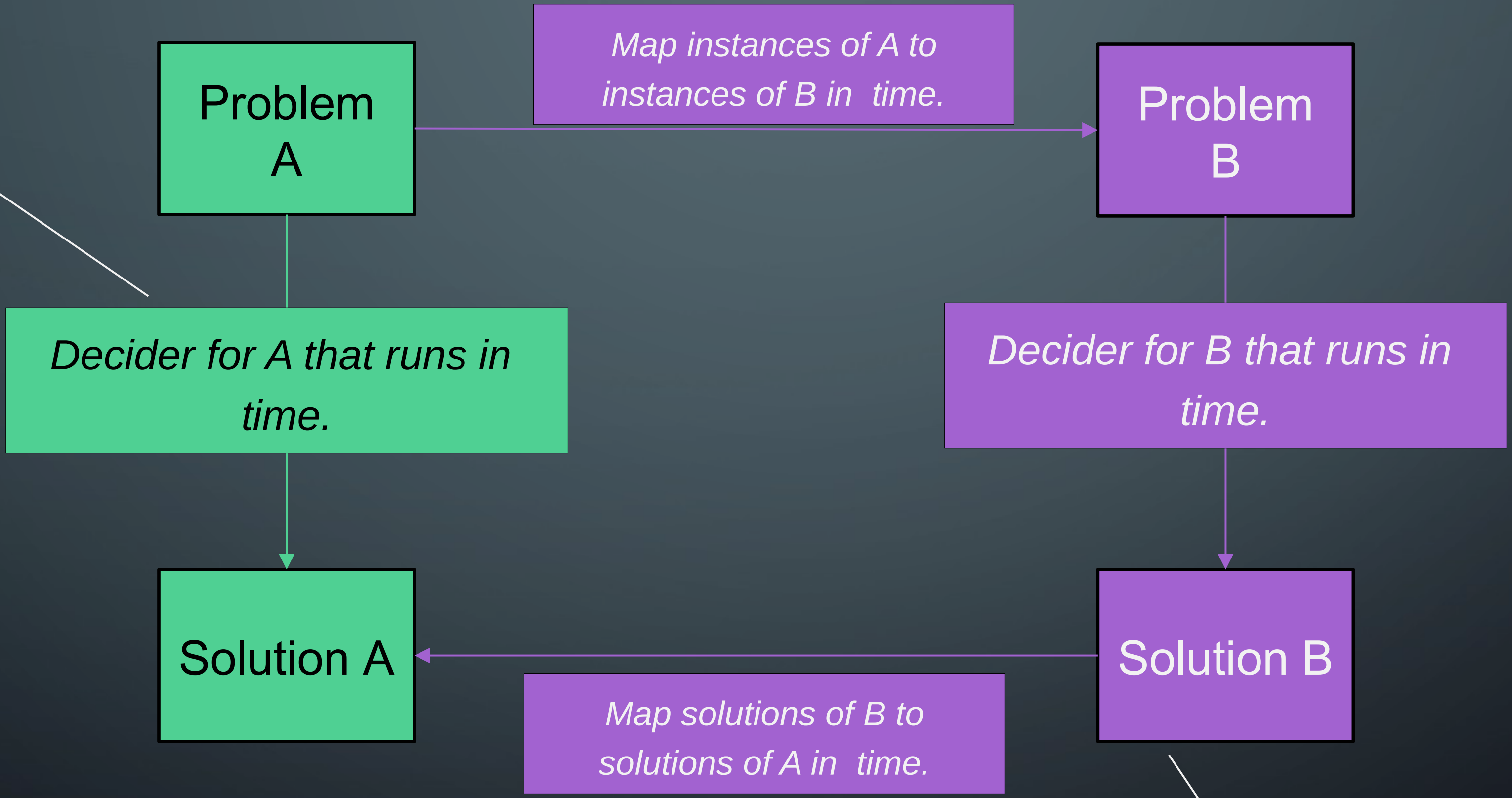


This kind of reduction involves the decidability of Problems A and B. If B is decidable then A is decidable!

MAPPING REDUCTION

*A **mapping reduction** uses a reduction function R() to map instances of one problem (A) to instances of another problem (B) such that for any input string ,*

One way (green route) to solve A is to use the decider in time



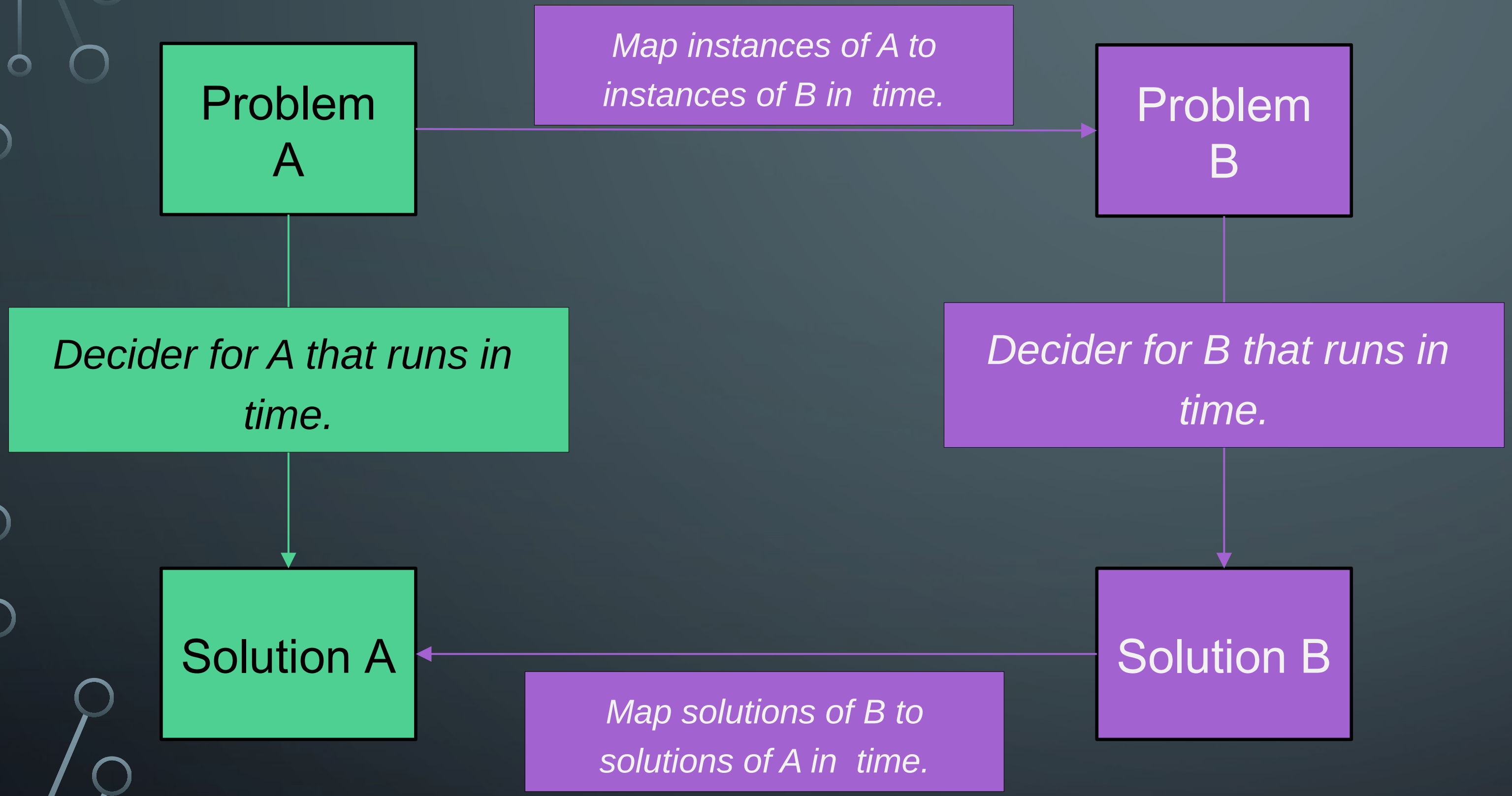
Problem A
Map instances of A to instances of B in time.
Problem B
Decider for A that runs in time.
Decider for B that runs in time.
Solution A
Solution B
Map solutions of B to solutions of A in time.

Another way to solve A is to use the purple path. Takes:

REDUCTIONS YOU'VE PROBABLY SEEN BEFORE!

Reduction:	*Details:*
Max-Flow Min-Cut	*No conversion necessary. Value of maximum flow is equal to capacity of minimum cut on the same, unaltered graph.*
Bi-Partite Matching Max-Flow	*Conversion involved adding capacities to edges, adding source and sink node, adding edges to / from source / sink node, etc.*
FindMedian Sorting	*No conversion necessary. Sort the list, then pull out the middle element in the array.*
FindMin Sorting	*No conversion necessary. Sort the list, then pull the first element in the array. Note that this one is a reduction to a HARDER problem. So won't be used in practice.*

RUNTIME COMPARISON



Problem A

Map instances of A to instances of B in time.

Problem B

Decider for A that runs in time.

Decider for B that runs in time.

Solution A

Solution B

Map solutions of B to solutions of A in time.

Which Algorithm is faster?

A_n

$R_{AB} + B_n + R\,S_{BA}$

If , then this represents a **valid reduction** *and*

If , then this is the best algorithm for A (or equally the best)

RUNTIME COMPARISON

Which Algorithm is faster?

A_n

$R_{AB} + B_n + RS_{BA} \in \Theta(B_n)$

B_n A_n

Not surprisingly, if these two algorithms have same overall runtime, then either can be used (they are equivalent).

Harder Problems (fastest algorithm has slower runtime)

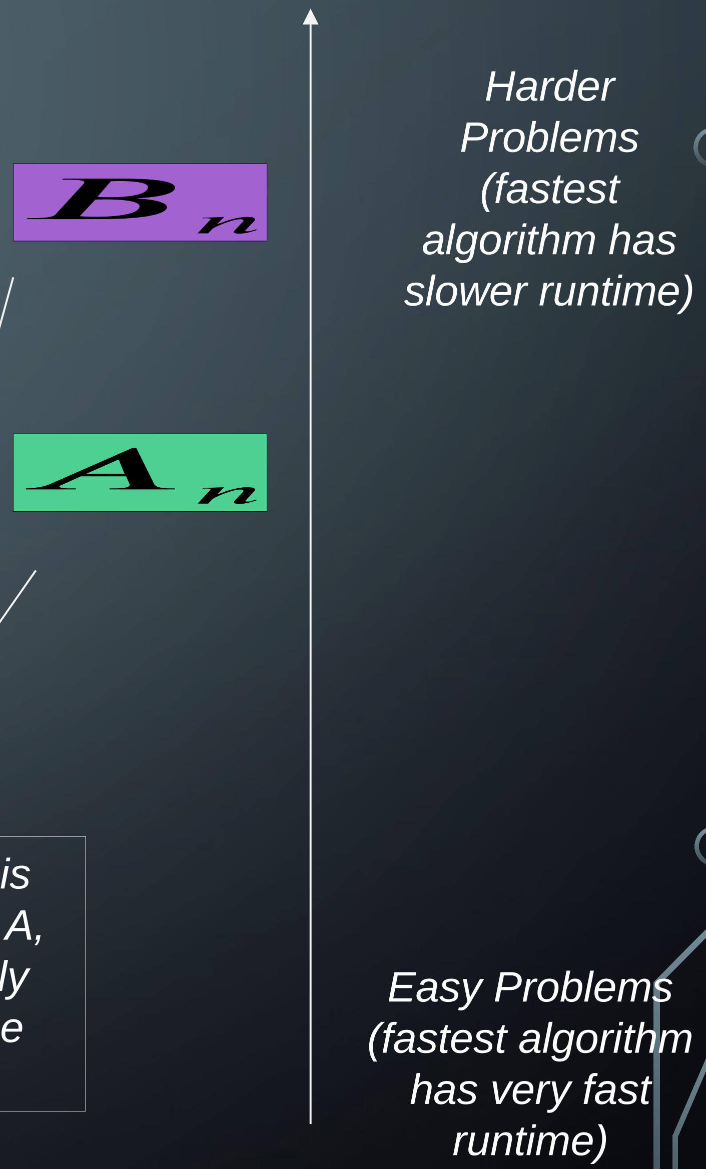
Easy Problems (fastest algorithm has very fast runtime)

RUNTIME COMPARISON

Which Algorithm is faster?

A_n

$R_{AB} + B_n + RS_{BA} \in \Theta(B_n)$


B_n
A_n
Harder Problems (fastest algorithm has slower runtime)
Easy Problems (fastest algorithm has very fast runtime)

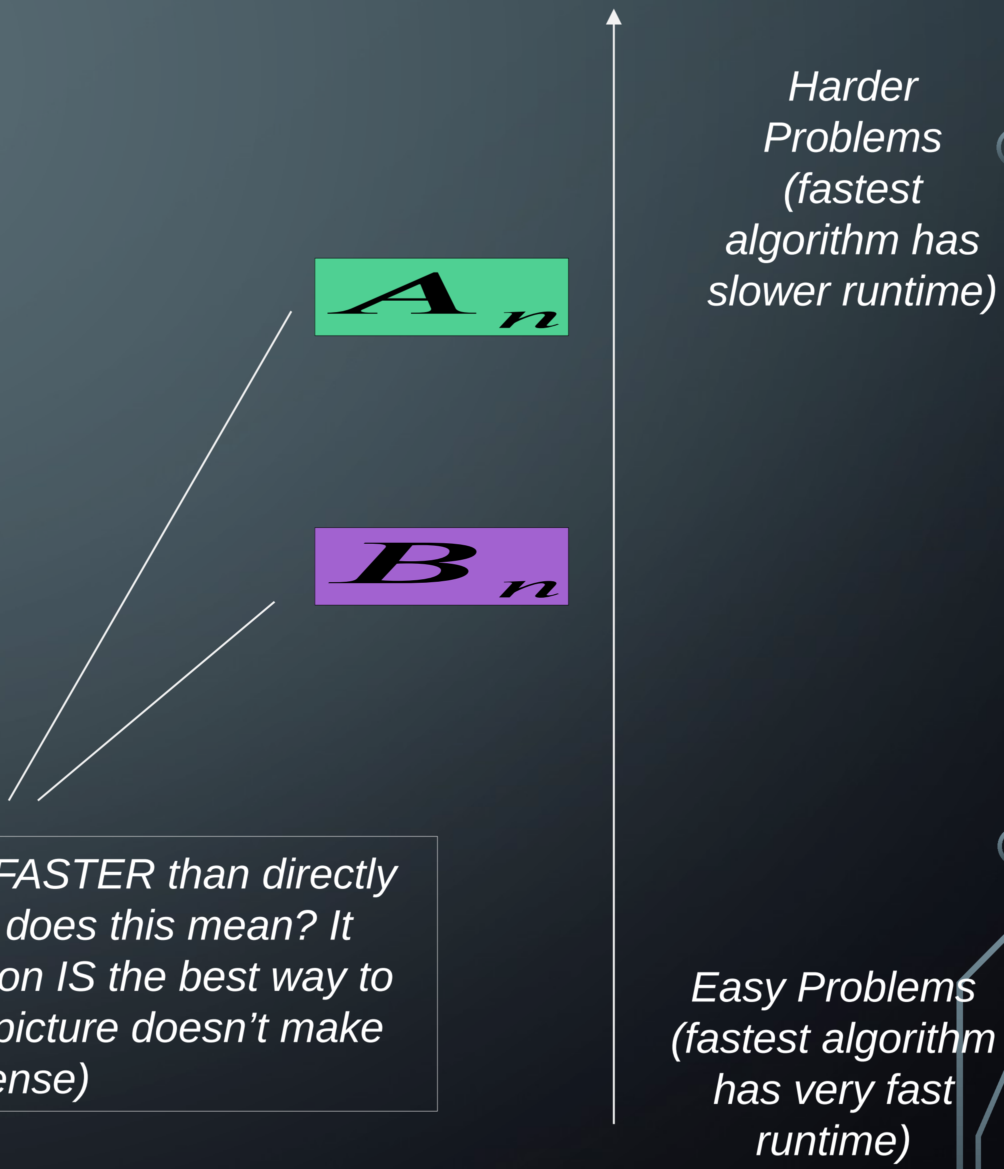
If solving A through reduction is SLOWER than directly solving A, this means problem B is simply harder than problem A (but the reduction is still valid)

RUNTIME COMPARISON

Which Algorithm is faster?

A_n

$R_{AB} + B_n + RS_{BA} \in \Theta(B_n)$


A_n
B_n
Harder Problems (fastest algorithm has slower runtime)
Easy Problems (fastest algorithm has very fast runtime)

If the reduction is FASTER than directly solving A, What does this mean? It means the reduction IS the best way to solve A (and this picture doesn't make sense)

RUNTIME COMPARISON

Which Algorithm is faster?

$$A_n$$

$$R_{AB} + B_n + RS_{BA} \in \Theta(B_n)$$

OLD

$$A_n = B_n$$

…and the direct algorithm for A is obsolete. The reduction through problem B is the direct way to solve A

Harder Problems (fastest algorithm has slower runtime)

Easy Problems (fastest algorithm has very fast runtime)

RUNTIME COMPARISON

Suppose time goes on, and somebody find a FASTER way to solve B in time, how will the picture to the right change as a result?

Harder Problems (fastest algorithm has slower runtime)

$$A_n = B_n$$

A now has a faster algorithm also! So improving B’s algorithm improves A’s. They are linked in this direction!

$$A'_n = R_{AB} + B'_n + RS_{BA}$$

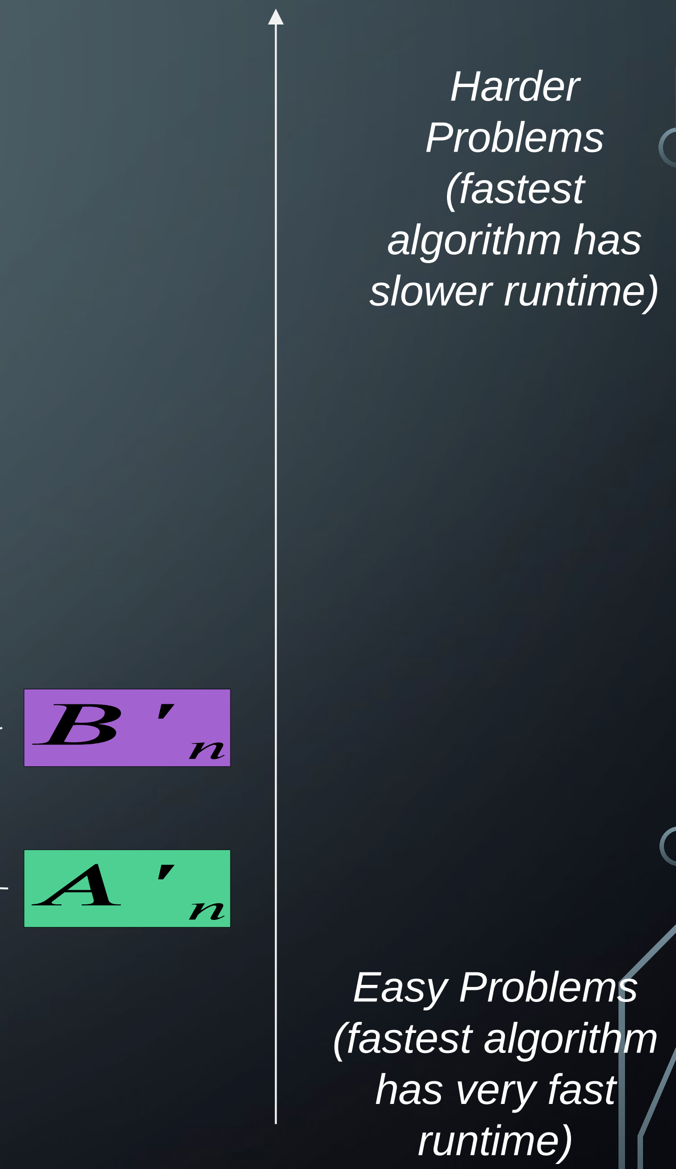
*This is ONLY true if the reduction stays **valid**, meaning the conversion is still fast:*

Easy Problems (fastest algorithm has very fast runtime)

RUNTIME COMPARISON

Now suppose time goes on and someone finds a VERY fast algorithm for A. What could happen?

Now, the reduction may still be valid, but we are back to B being strictly harder than A



Harder Problems (fastest algorithm has slower runtime)
B'_n
A'_n
Easy Problems (fastest algorithm has very fast runtime)

BIG PICTURE

So, via reduction

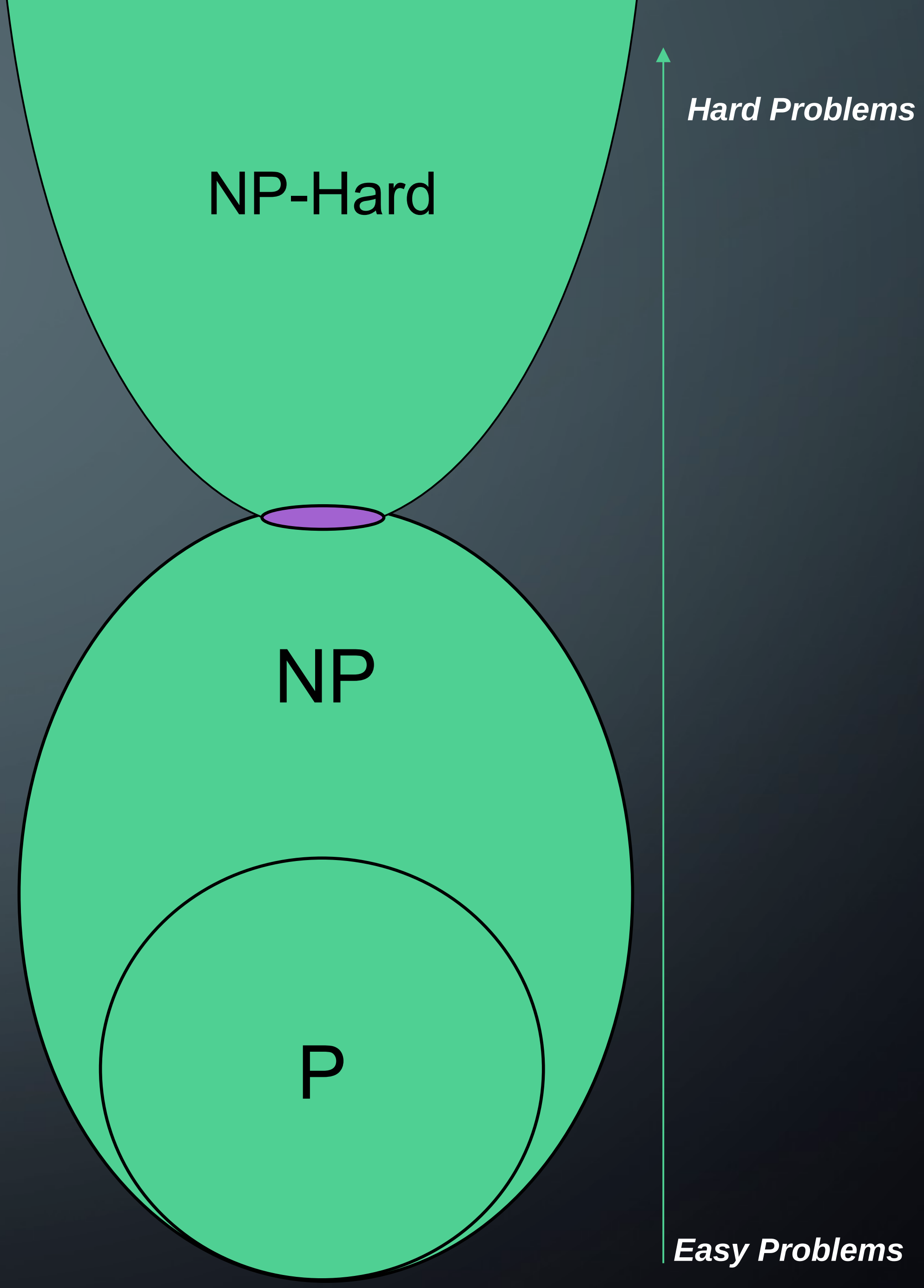
*A **valid** reduction establishes that B is at least as hard as A*

Some related facts!

If valid reductions exist in both directions: and , then the two problems are equally as hard

NP-Complete problems are the hardest in NP, so by definition there is a valid reduction from anything in NP to them.

How fast does a reduction between NP-Complete problems need to be? Just some polynomial. Why? We write this as



PROVING NP-COMPLETENESS

Usually we do the bolded ones

But for second step, we need a known NP-Complete problem. What was the first one?

To prove a problem A is NP-Complete, show that:

How? Either:

Solve in Polynomial time with an NTM

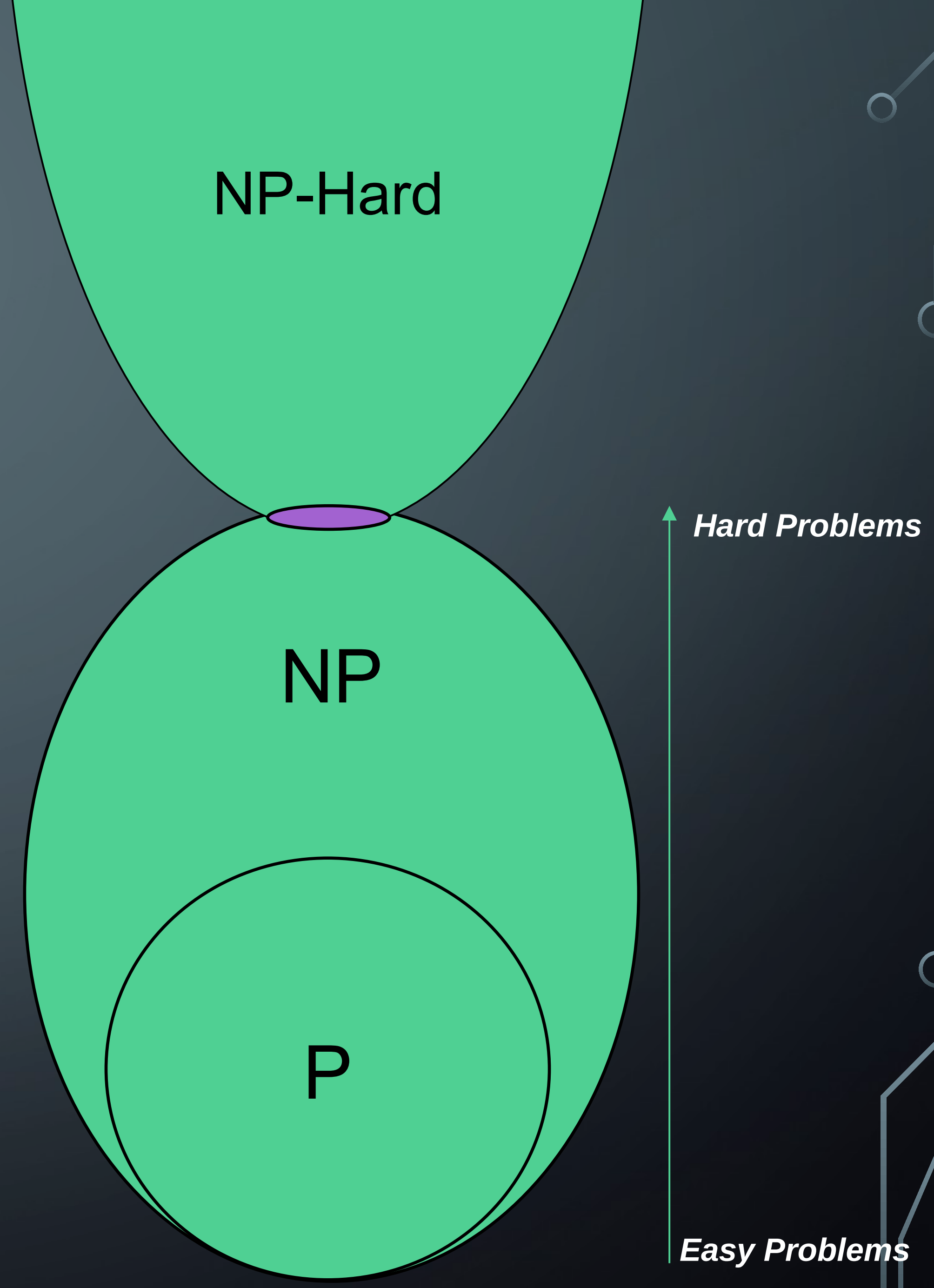
Verify in Polynomial time with a DTM

1. Is NP-Hard

How? Either:

Show that

Pick known NP-Complete problem B and show



COOK-LEVIN THEOREM

COOK-LEVIN THEOREM

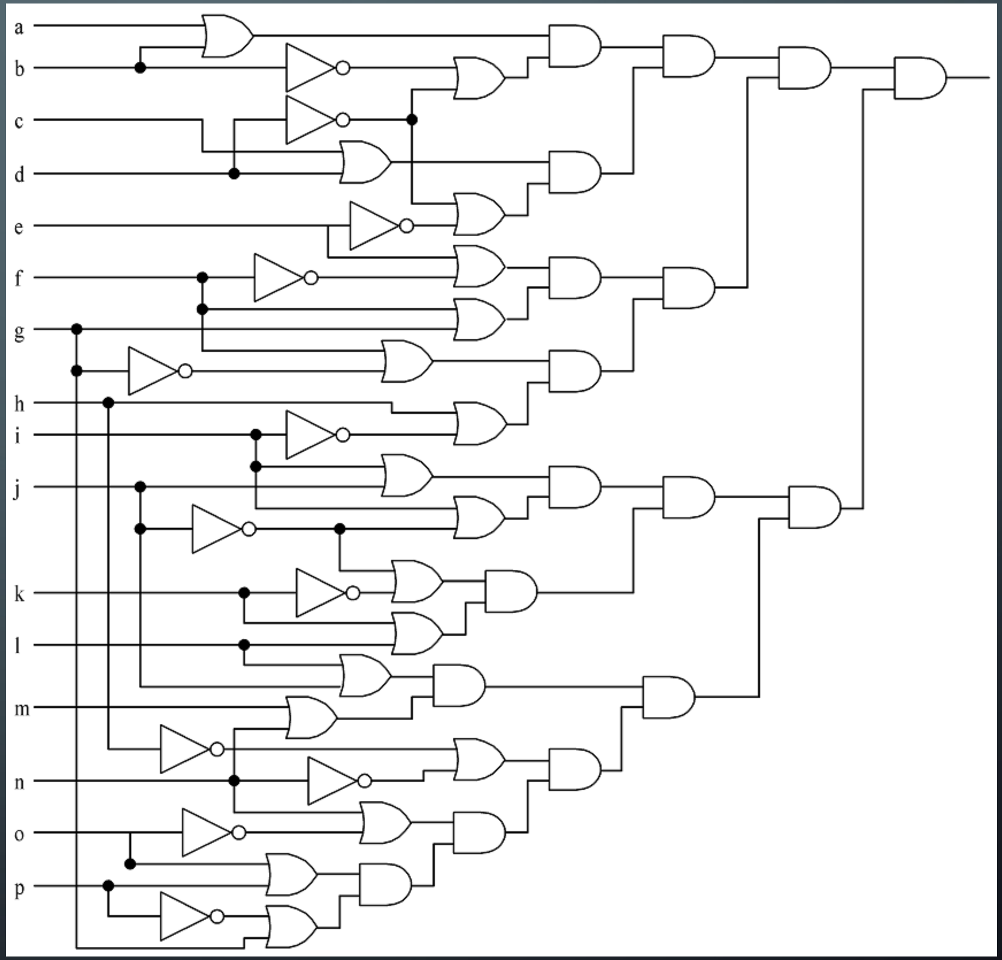
Cook-Levin Theorem: The Satisfiability (SAT) problem is NP-Complete

Incredibly famous theorem. Established the first known NP-Complete problem!

Developed independently by Stephen Cook (US) and Leonid Levin (USSR) in 1971 & 1973

CIRCUIT SATISFIABILITY (CIRCUIT-SAT)

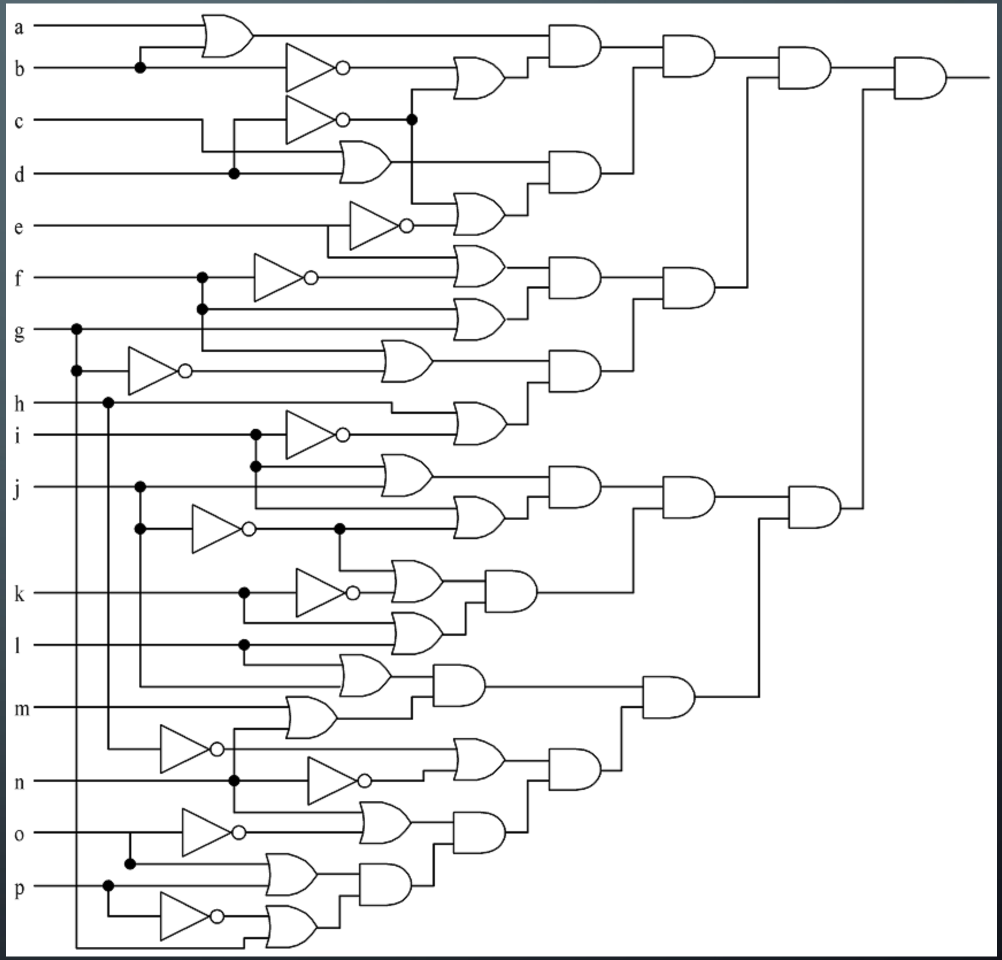
Given a circuit with boolean inputs, AND, OR, and NOT gates…is it possible to assign values to the input such that the output is TRUE?



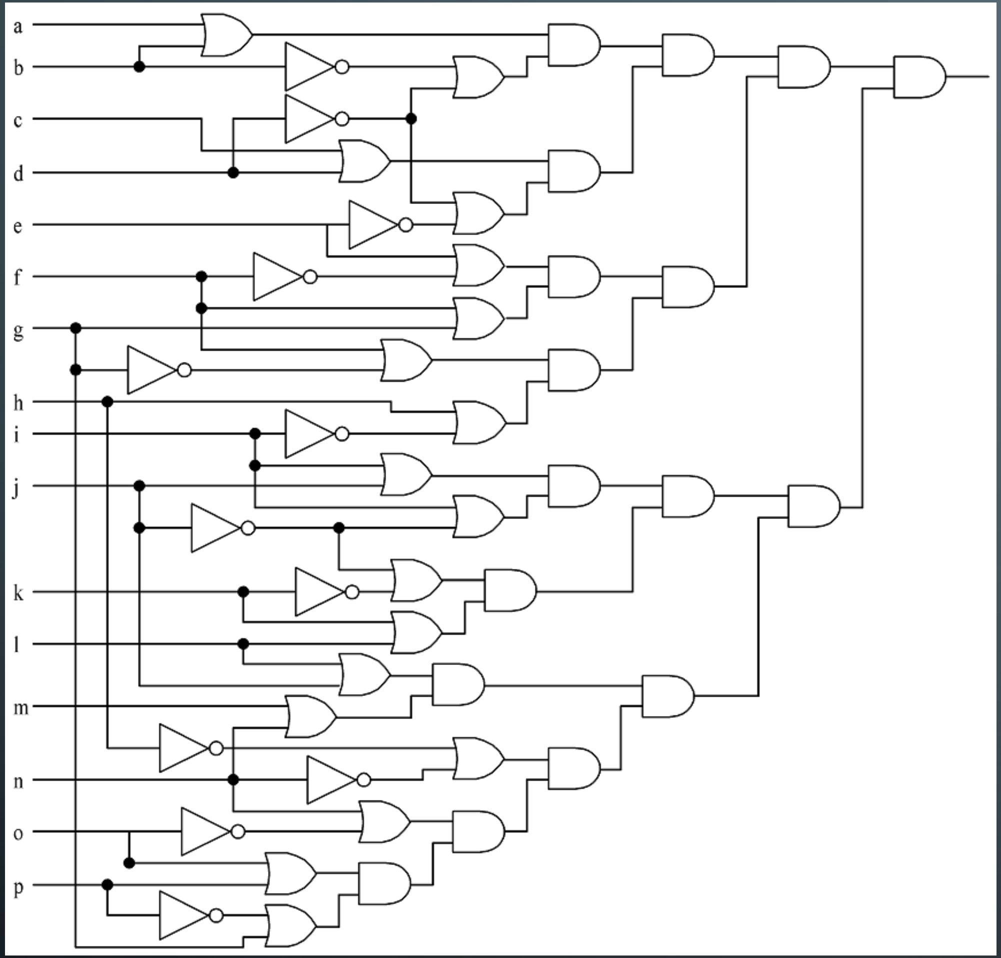
CIRCUIT SATISFIABILITY (CIRCUIT-SAT)

Solutions:

- 1110111110011001
- 1010111111011001
- 0110111110111001
- 0110111110011001
- 1110111111011001
- 1010111110011001
- 1010111110111001
- 0110111111011001
- 1110111110111001



CIRCUIT-SAT VS SAT



```
(v[0] || v[1]) && (!v[1] || !
v[3]) && (v[2] || v[3]) && (!
v[3] || !v[4]) && (v[4] || !
v[5]) && (v[5] || !v[6]) &&
(v[5] || v[6]) && (v[6] || !
v[15]) && (v[7] || !v[8]) && (!
v[7] || !v[13]) && (v[8] ||
v[9]) && (v[8] || !v[9]) && (!
v[9] || !v[10]) && (v[9] ||
v[11]) && (v[10] || v[11]) &&
(v[12] || v[13]) && (v[13] || !
v[14]) && (v[14] || v[15])
```

These are two variations of the exact same problem. We will stick with the right side (SAT) from now on

PROOF OF THE COOK-LEVIN THEOREM

SAT ∈ NPC

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

Show that
OR

Here, we must use the second (bold) option because there are not any NPC problems that exist yet! Ugh!!

SAT ∈ NPC

Let’s do this one first:

Provide a verifier TM that runs in Polynomial Time

Needs to be polynomial runtime, is it? Yes!

Verifier:

Given variables V, formula F, and potential values for each variable V’:

1. *Scan over formula F for first operator (Op) that should be applied (deepest in parens and/or lowest precedence)*
2. *Find the two variables X and Y on each side of Op, this gives X Op Y (example: V1 AND V7)*
3. *Apply operator Op to the values X and Y given by V’ or by result of a previous operation and replace X Op Y with this Boolean result.*
4. *Loop back to step 1 until only one Boolean remains. This Boolean is true if and only if the solution V’ is verified.*

SAT ∈ NPC

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

Show that
OR

This part is done!!

SAT IS NP-HARD

Show that
OR

As we stated. before, we have to use the second option because there (when this proof was done) are no NP-Complete problems yet!

SAT IS NP-HARD

Choose arbitrary

NTM Decider for x

Reduce problem x

$\longrightarrow$

To an instance of SAT

$x_1 \wedge \overline{x_2} \vee (\overline{x_3} \wedge x_2) \ldots$

How are we going to do this?

SAT IS NP-HARD

Choose arbitrary | *Reduce problem x* | *To an instance of SAT*

NTM Decider for x → *Tape moved right AND 1 written to first cell of tape AND …*

***<u>IDEA</u>**: For any generic problem x in NP, it has a decider NTM. Convert that NTM into a Boolean expression that describes the operation of the machine. Why is this a valid reduction?*

VARIABLES WE NEED

Variable	Meaning	How many
T_{ijk}	True if tape cell *i* contains symbol *j* at step *k* of the computation	$O(p(n)^2)$
H_{ik}	True if the M’s read/write head is at tape cell *i* at step *k* of the computation	$O(p(n)^2)$
Q_{qk}	True if M is in state *q* at step *k* of the computation	$O(p(n))$

Some constraints:

$$q \in Q$$

Note that p(n) is the time the original NTM takes and

CREATE A CONJUNCTION ‘B’ OF…

Expression	Conditions	Interpretation	How many
T_{ij0}	Tape cell i initially contains symbol J	Initial tape state; blank symbols above n	O(p(n))
Q_{s0}		Initial state of the NTM	1
H_{00}		Initial position of the read/write head	1
$T_{ijk} \rightarrow \neg T_{ij'k}$	j != j’	One symbol per tape cell	$O(p(n)^2)$
$T_{ijk} = T_{ij(k+1)} \vee H_{jk}$		Tape remains unchanged unless written	$O(p(n)^2)$
$Q_{qk} \rightarrow \neg Q_{q'k}$	$q \neq q'$	Only one state at a time	O(p(n))
$H_{jk} \rightarrow \neg H_{j'k}$	$i \neq i'$	Only one head position at a time	$O(p(n)^2)$
$(H_{ik} \wedge Q_{qk} \wedge T_{i\sigma k}) \rightarrow (H_{(i+d)(k+1)} \wedge Q_{q'(k+1)} \wedge T_{i\sigma'(k+1)})$	$(q, \sigma, q', \sigma', d) \in \delta$	Possible transitions at computation step k when head position is at position I	$O(p(n)^2)$

IS THE REDUCTION VALID?

NTM for x accepts iff and only if SAT equation can be satisfied

If there is an accepting computation for the NTM on input I, then B is satisfiable by assigning T_{ijk}, H_{jk}, and Q_{jk} their intended interpretations.

The time and space complexity of the reduction is polynomial

Yes!

The number of sub-expressions is:

$$\mathbf{2p(n) + 4p(n)^2 + 3 = O(p(n)^2)}$$

and each is computed in less than that.

SAT $\in$ NPC

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

Thus, it is proven!!

OTHER NP-COMPLETE PROBLEMS (REDUCTIONS)

3-SAT

3-SAT

3-SAT = Can a provided Boolean expression in 3-Conjunctive-Normal Form (3-CNF) be satisfied?

$$V = \left(v_1 \vee v_2 \vee \overline{v_3}\right) \wedge \left(v_4 \vee \overline{v_1} \vee v_2\right) \wedge \left(v_4 \vee \overline{v_3} \vee \overline{v_1}\right) \wedge \ldots$$

Each Clause contains a disjunction (OR) of exactly 3 literals (or negated literals)

The expression must be a conjunction (AND) of multiple clauses

Is it easier to decide 3-SAT because the format is simpler?

SHOWING THAT

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

This one, as usual, is not difficult.

This time we can reduce from a concrete, known, NPC problem. We only have SAT so far, so that is what we will choose!

SHOWING THAT

Provide a verifier TM that runs in Polynomial Time

This is trivial. The verifier we developed for SAT will also work for 3SAT.

SHOWING THAT

Need to show 3SAT is at least as hard as SAT. How? Show a reduction.

Given a generic SAT input, can we convert it into an equivalent formula in 3SAT?

SAT input x:

e.g.,

$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

$\Rightarrow$

Equivalent 3SAT formula:

e.g.,

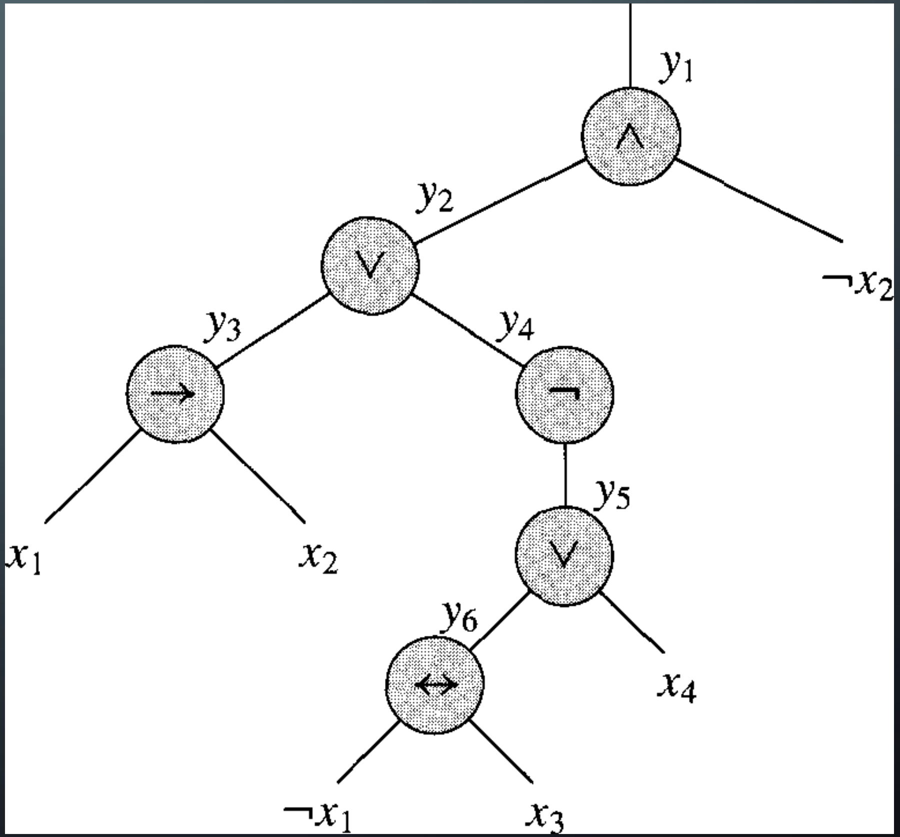
$\phi'_i = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) \ldots$

CONVERTING SAT TO 3-SAT, STEP 1

Input:

$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

***<u>Step 1</u>*: Parse the expression into an expression tree**



CONVERTING SAT TO 3-SAT, STEP 2

***Step 2**: Introduce a variable for each internal node. This variable will represent whether or not that subtree expression evaluated to True or False*

We can then re-write our expression:

$\phi' = \quad y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

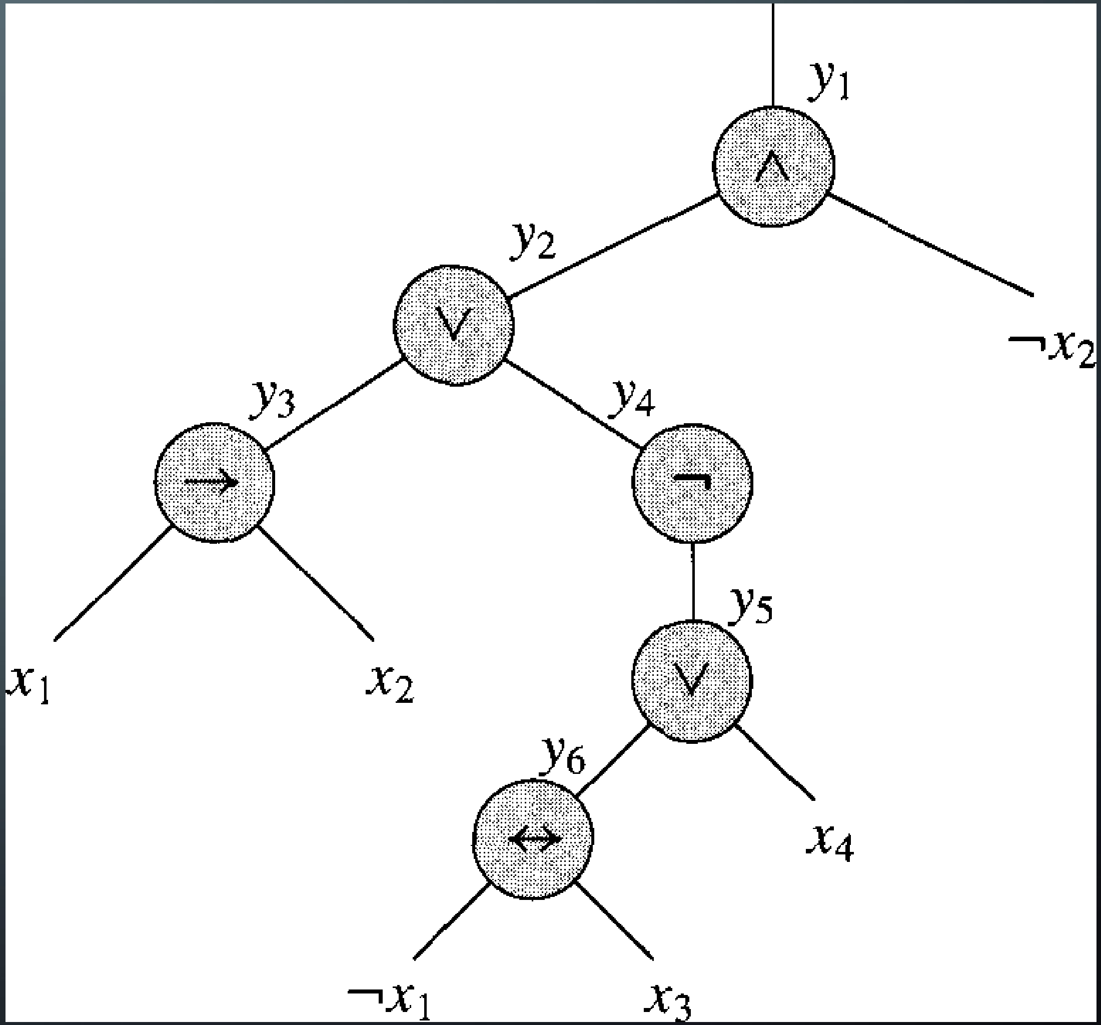
$\wedge (y_2 \leftrightarrow (y_3 \vee y_4))$

$\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))$

$\wedge (y_4 \leftrightarrow \neg y_5)$

$\wedge (y_5 \leftrightarrow (y_6 \vee x_4))$

$\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$



CONVERTING SAT TO 3-SAT, STEP 3

Step 3:

Build a truth table for each clause ϕ'_i:

- $\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

  $\wedge (y_2 \leftrightarrow (y_3 \vee y_4))$

  $\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))$

  $\wedge (y_4 \leftrightarrow \neg y_5)$

  $\wedge (y_5 \leftrightarrow (y_6 \vee x_4))$

  $\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$

y_1	y_2	x2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

CONVERTING SAT TO 3-SAT, STEP 4 / 5

Step 4: For each clause, construct a DNF (disjunctive normal form) for when it is False (based on truth table)

y_1	**y_2**	**x_2**	**$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$**
1	**1**	**1**	**0**
1	1	0	1
1	**0**	**1**	**0**
1	**0**	**0**	**0**
0	1	1	1
0	**1**	**0**	**0**
0	0	1	1
0	0	0	1

$\neg\phi'_i =$

$(y_1 \wedge y_2 \wedge x_2) \vee$

$(y_1 \wedge \neg y_2 \wedge x_2) \vee$

$(y_1 \wedge \neg y_2 \wedge \neg x_2) \vee$

$(\neg y_1 \wedge y_2 \wedge \neg x_2)$

Step 5: Take this formula and negate it to get all the instances where the clause is true in CNF (conjunctive normal form).

$\neg\phi'_i = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

Negate formula

$\phi'_i = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$

CONVERTING SAT TO 3-SAT, STEP 6

$$\phi'_i = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Step 6: Almost done. This works but some clauses may have only 1 or 2 literals (3 are required for every single clause). Add dummy variables to force each clause to have three literals.

Case 1: Clause has 3 literals

$$(\upsilon_i \vee \upsilon_j \vee \upsilon_k)$$

Do nothing, already fine

$$(\upsilon_i \vee \upsilon_j \vee \upsilon_k)$$

Case 2: Clause has only 2 literals

$$(\upsilon_i \vee \upsilon_j)$$

Becomes:

Introduce dummy variable p

$$(\upsilon_i \vee \upsilon_j \vee p) \wedge (\upsilon_i \vee \upsilon_j \vee \neg p)$$

Case 3: Clause has only 1 literal

$$(\upsilon_i)$$

Becomes:

Introduce dummy variables p and q

$$(\upsilon_i \vee p \vee q) \wedge (\upsilon_i \vee \neg p \vee q) \vee (\upsilon_i \vee p \vee \neg q) \wedge (\upsilon_i \vee \neg p \vee \neg q)$$

SHOWING THAT

To show that , we must show both that:

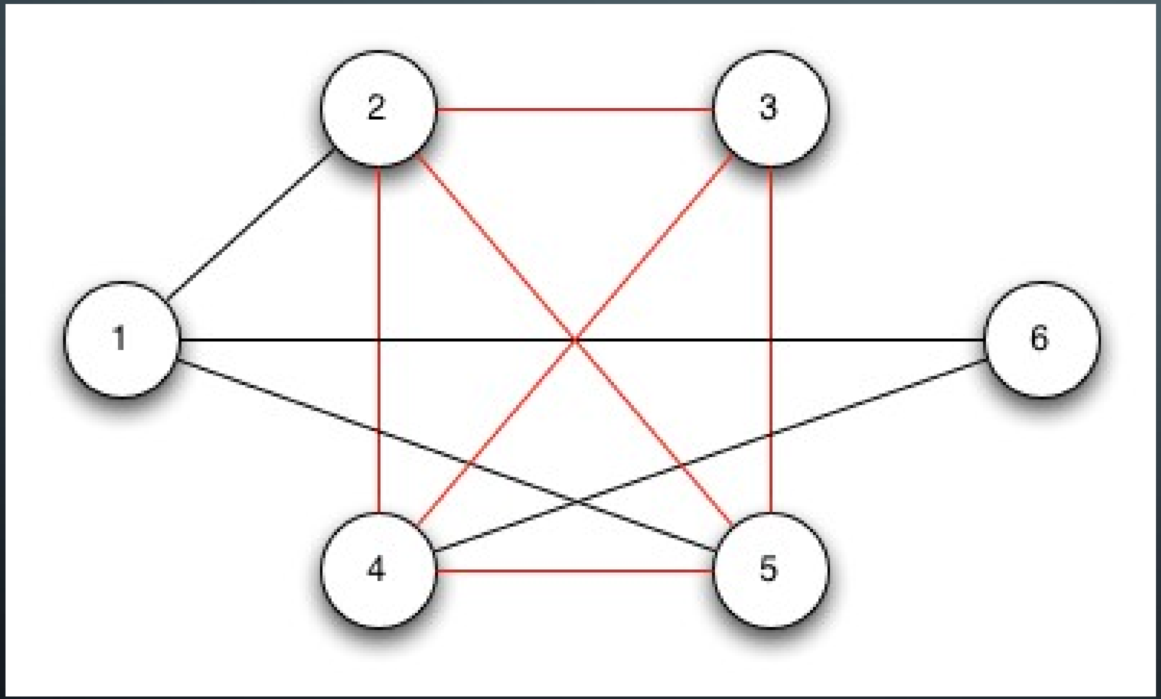
Provide a verifier TM that runs in Polynomial Time

We are done!!

CLIQUES

CLIQUE

A **Clique** in a graph G is a set of nodes such that each one is connected to each other in the set

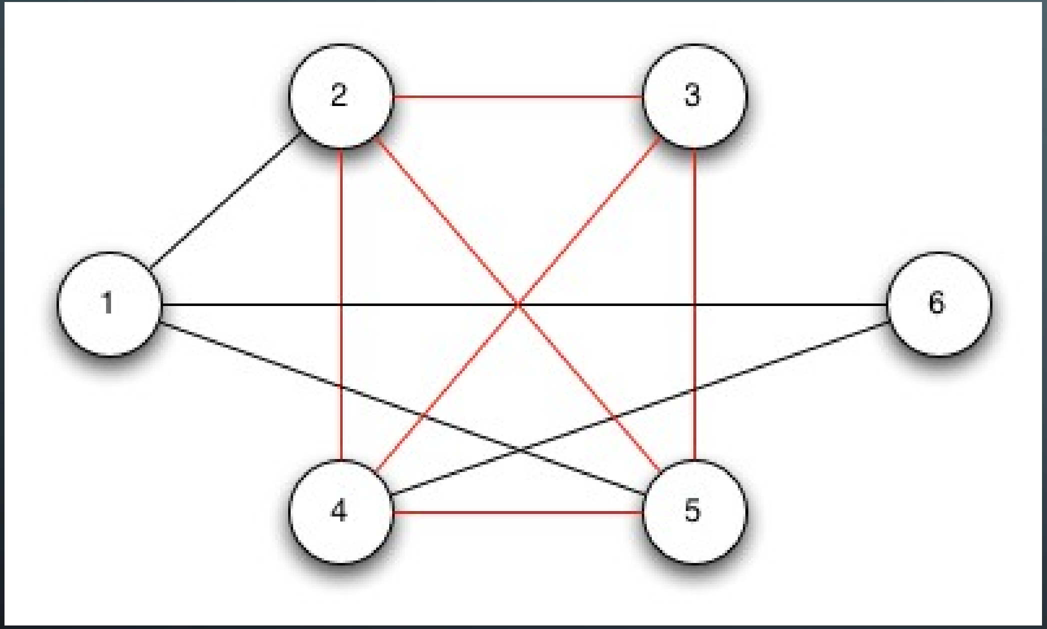


In other words, it is a maximal sub-graph of G

Problem: Find the maximum size clique in a graph G

CLIQUE

A **Clique** in a graph G is a set of nodes such that each one is connected to each other in the set



*Can we frame this as a **Decision Problem**?*

Given a graph G and an integer k, return Yes iff G has a click of size k or larger.

SHOWING THAT

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

As usual, this one is pretty simple

For this one, we can choose SAT or 3-SAT

SHOWING THAT

Provide a verifier TM that runs in Polynomial Time

Verifier:

Given G, k, and a subset of nodes

1. Verify that number of nodes in V' is k or larger
2. For each pair of nodes (p,q) in V':
    1. check that edge p,q exists in G
    2. If not, return **NO**
3. Return **YES**

SHOWING THAT

3-SAT

We choose 3-SAT

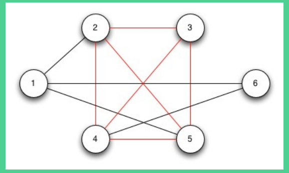
<u>Goal</u>: *Given a generic 3-SAT input, can we convert it into graph and integer k such that the 3-SAT formula is satisfiable IFF the graph has a click of at least size k?*

Input: 3SAT formula:

e.g.,

$\phi'_i = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\ldots$

Graph G and integer k



Converting a Boolean formula into a graph is strange, right? Let's see how it works!

, INTUITION

Consider this 3-SAT formula:

$$\theta = \left(x_1 \vee \overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee x_2 \vee x_3\right)$$

__TIP__: When doing a reduction, think about the "spirit" of how the problems relate to each other

With a 3-Sat formula, we have:

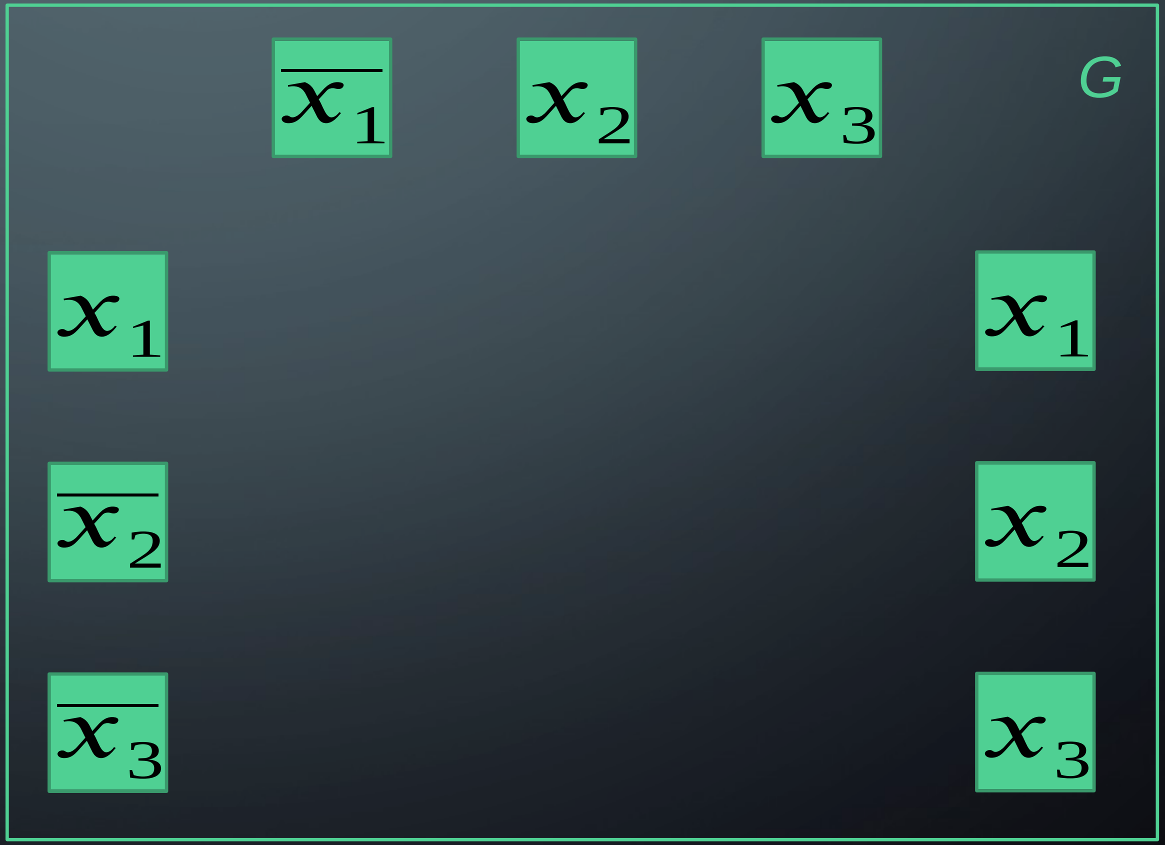
1. *A bunch of "things" (variables)*
2. *Some can be assigned TRUE without issue (they are "connected")*
3. *Each clause must have a TRUE item that is connected (valid) with the other items in the other clauses*

, STEP 1

Consider this 3-SAT formula:

$$\theta = \left(x_1 \vee \overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee x_2 \vee x_3\right)$$

Step 1: *Create a graph G with nodes where each variable in represents a node in G*

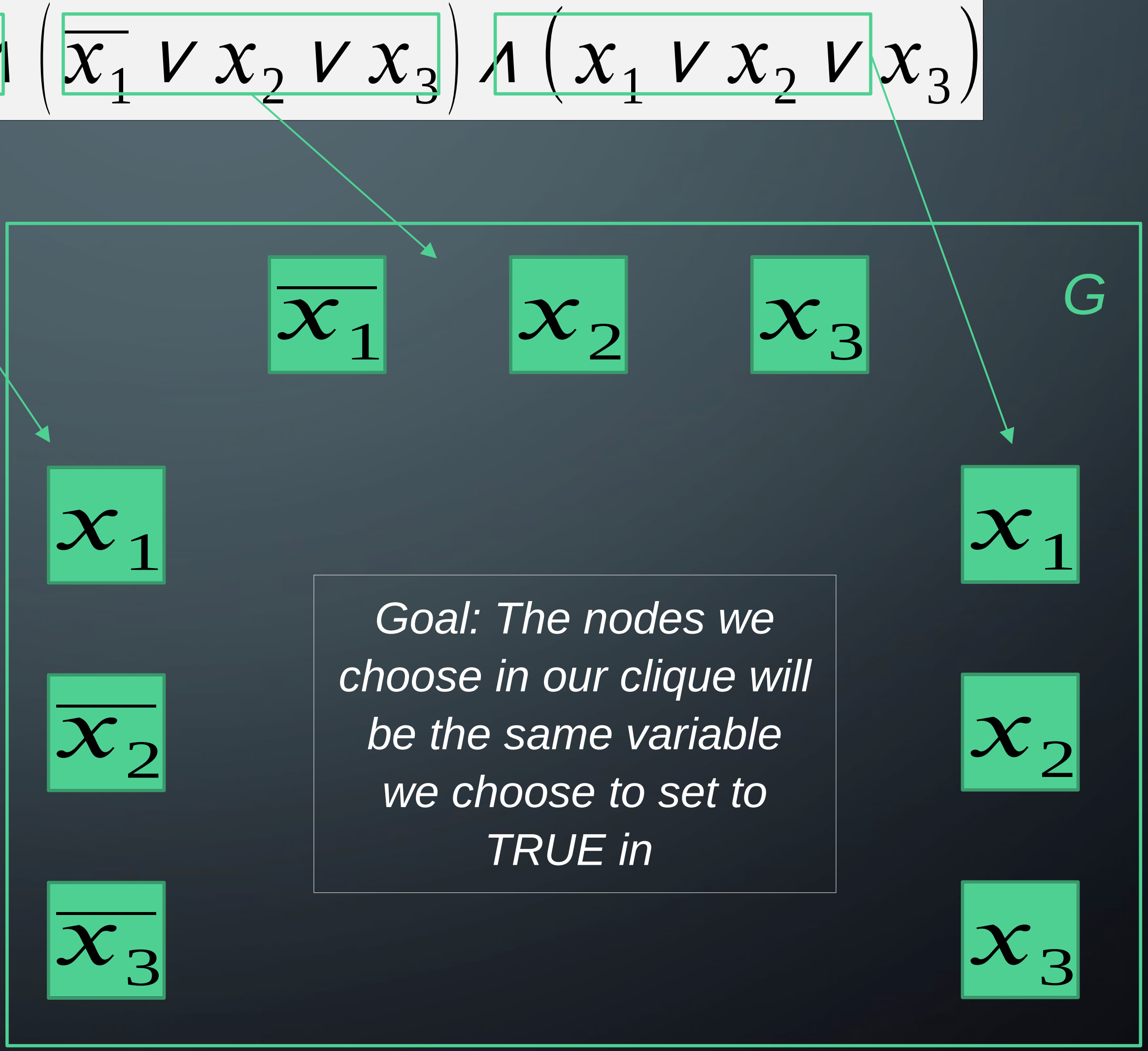


, STEP 1

Consider this 3-SAT formula:

$$\theta = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Step 1: *Create a graph G with nodes where each variable in represents a node in G*



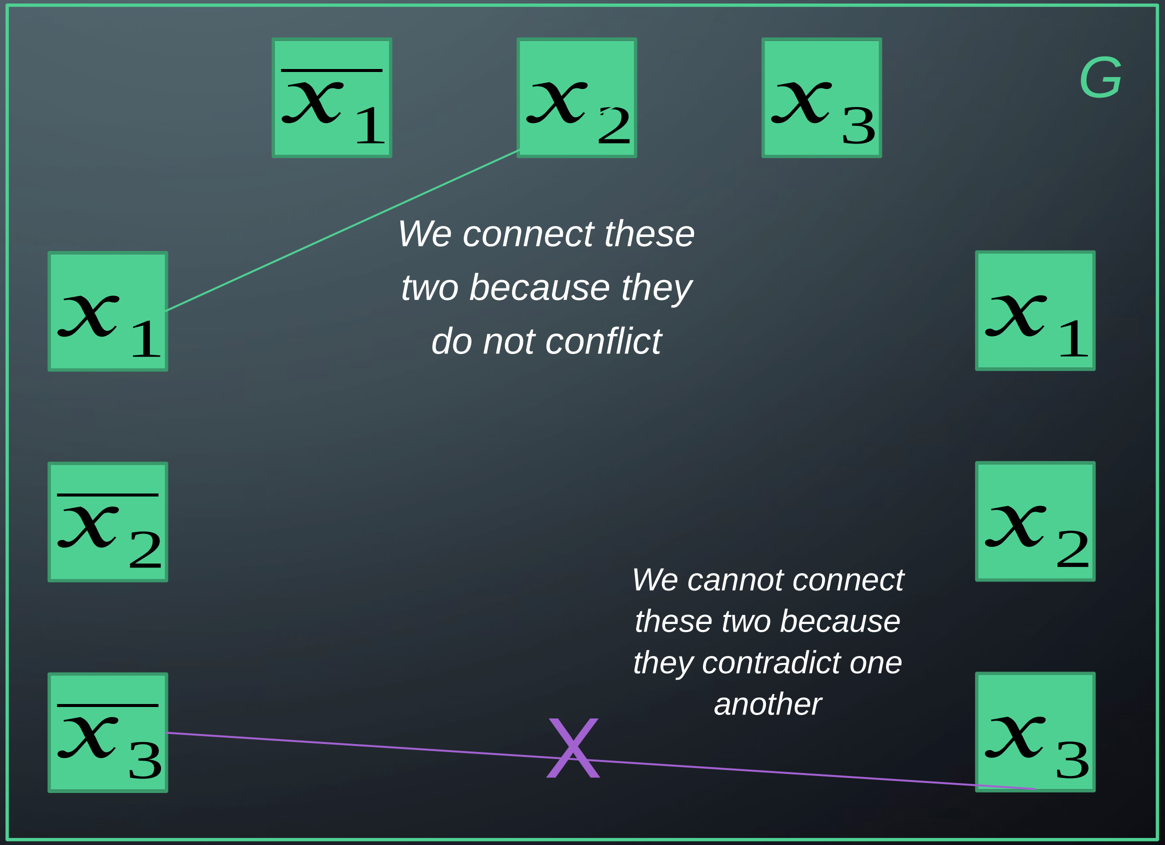
Goal: The nodes we choose in our clique will be the same variable we choose to set to TRUE in

, STEP 2

Consider this 3-SAT formula:

$$\theta = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Step 2: *Connect any two nodes that are in different clauses AND can be set to true at the same time*



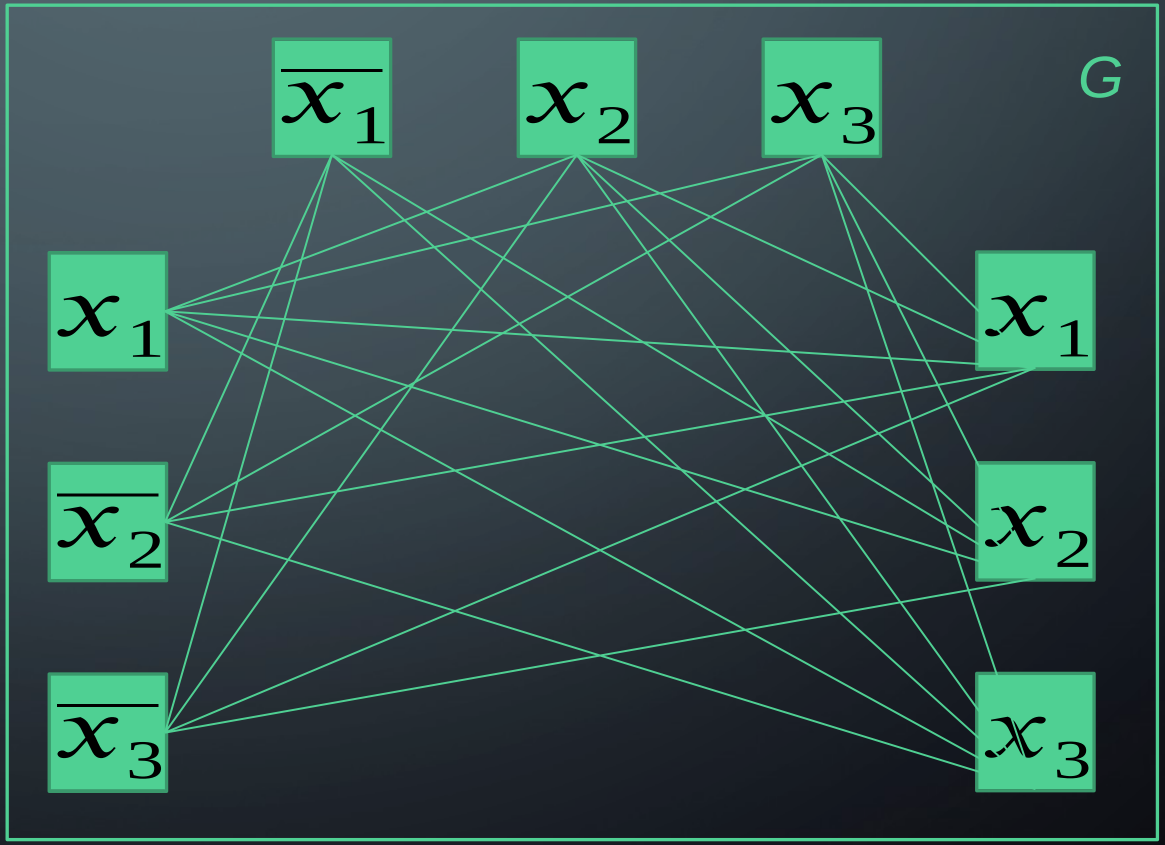
G
$\overline{x_1}$
x_2
x_3
x_1
$\overline{x_2}$
$\overline{x_3}$
x_1
x_2
x_3
We connect these two because they do not conflict
We cannot connect these two because they contradict one another
X

, STEP 2

Consider this 3-SAT formula:

$$\theta = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

***Step 2**: Connect any two nodes that are in different clauses AND can be set to true at the same time*

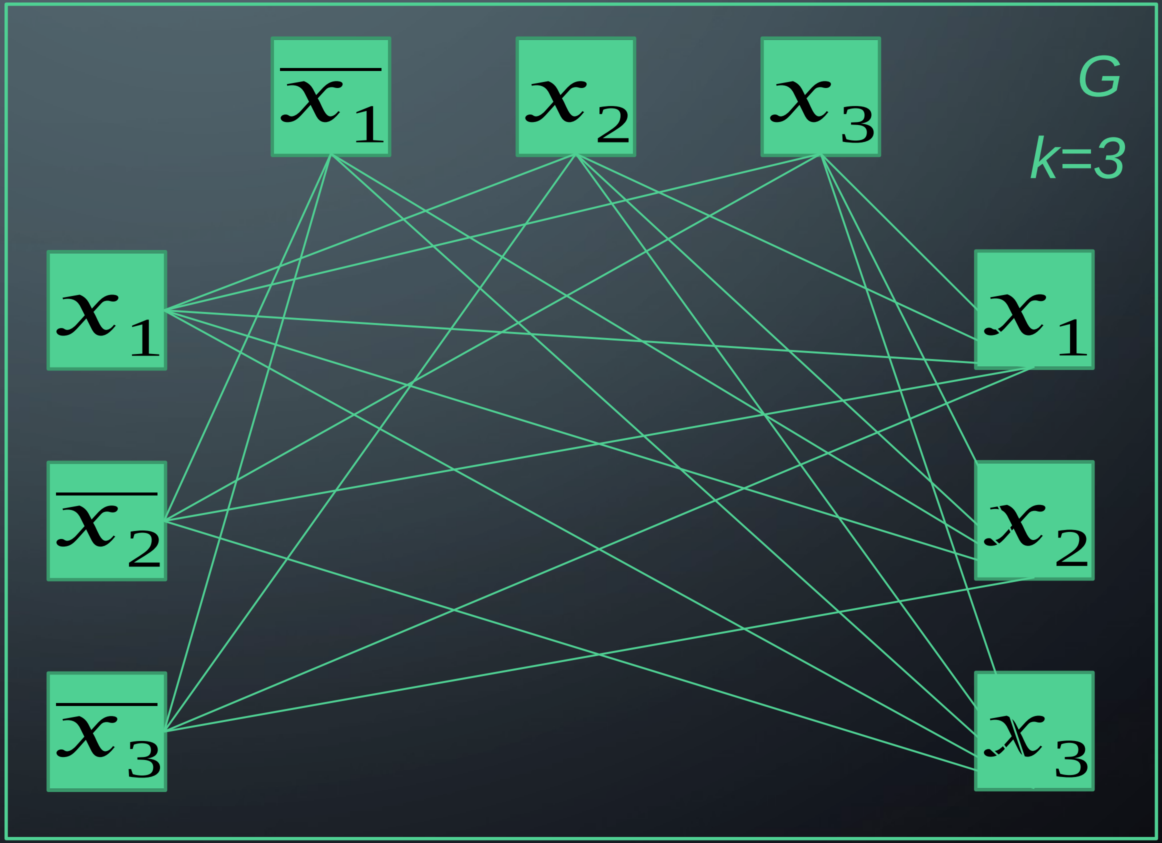


, STEP 3

Consider this 3-SAT formula:

$$\theta = \left(x_1 \vee \overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee x_2 \vee x_3\right)$$

Step 3: *Set k equal to the number of clauses in*



, PROOF

Consider this 3-SAT formula:

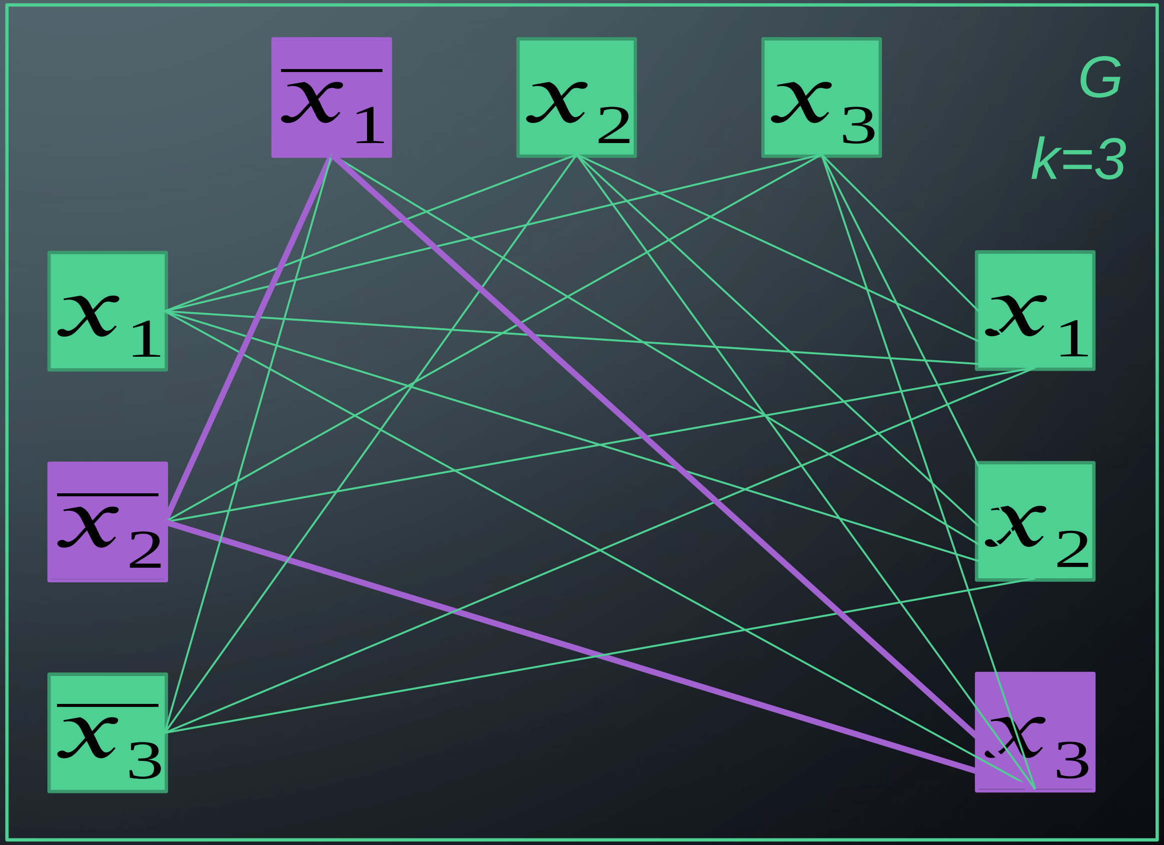
$$\theta = \left(x_1 \vee \overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee x_2 \vee x_3\right)$$

Claim:

is satisfiable IFF G contains a clique of size 3

Intuition:

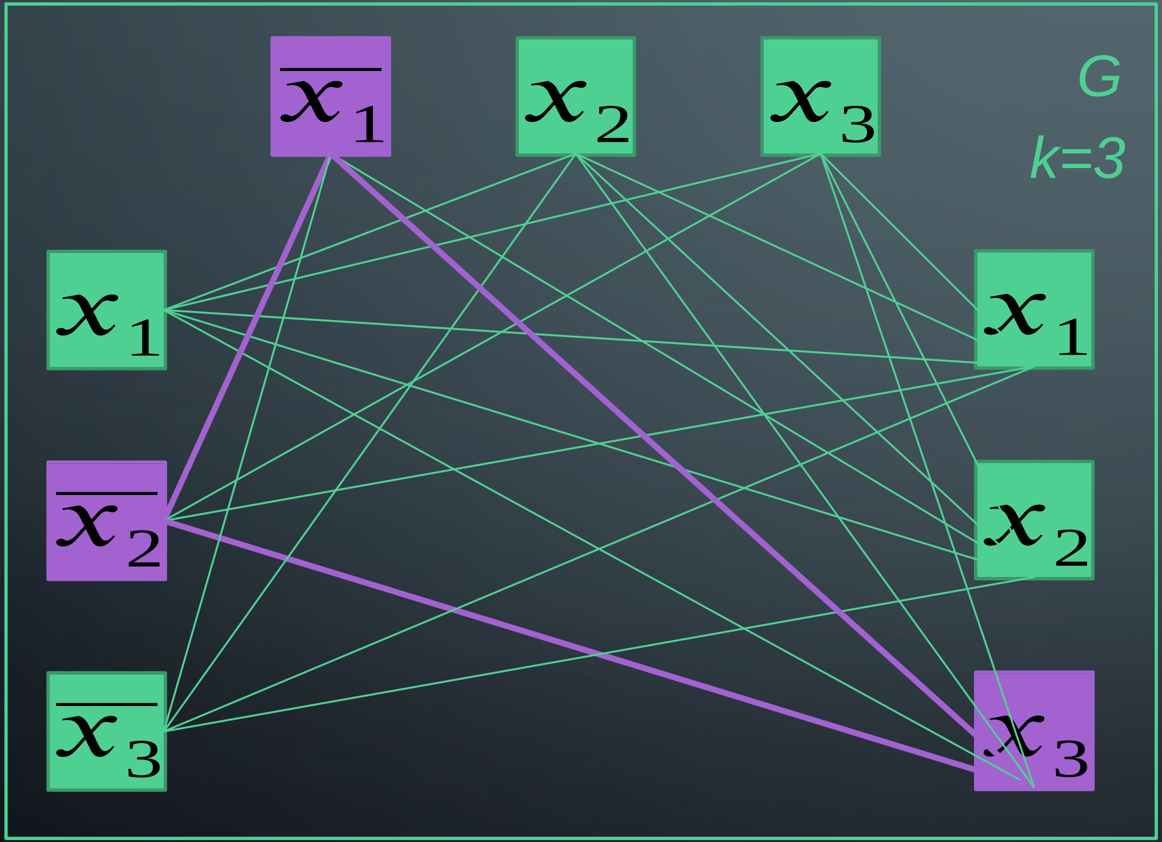
One clique of size 3 is shown. The nodes in the clique represent three variables, one per clause, that can be set to TRUE without issue.



, PROOF

Consider this 3-SAT formula:

$$\theta = \left(x_1 \vee \boxed{\overline{x_2}} \vee \overline{x_3}\right) \wedge \left(\boxed{\overline{x_1}} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee x_2 \vee \boxed{x_3}\right)$$



Direction 1:
is satisfiable → G contains a clique of size k

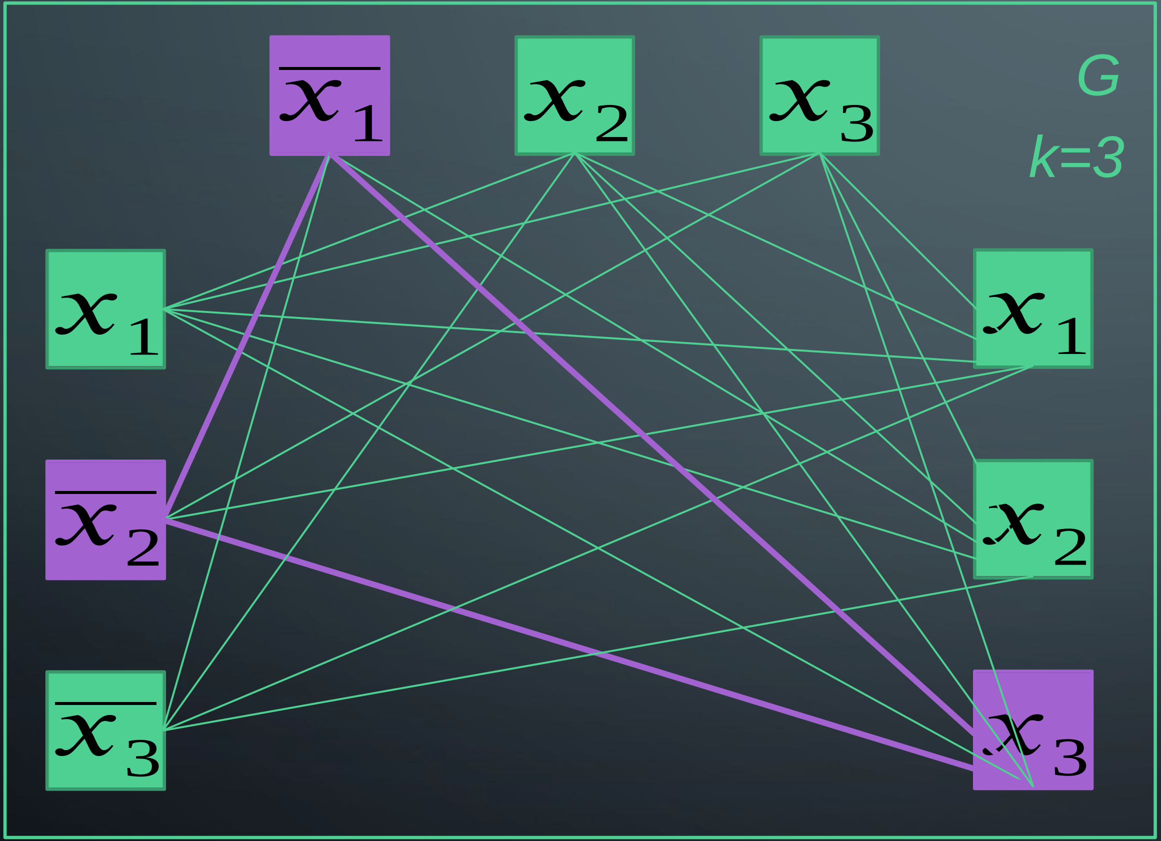
Proof:
is satisfiable
This means at least one variable is true in each clause
Take one true variable from each clause (k total)
Find their nodes in G
These nodes MUST be a clique of size k
- *Each of the k nodes is connected to each other:*
  - *They are in a different clause*
  - *They can both be assigned true*

Q.E.D.

, PROOF

Consider this 3-SAT formula:

$$\theta = (x_1 \vee \boxed{\overline{x_2}} \vee \overline{x_3}) \wedge (\boxed{\overline{x_1}} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \boxed{x_3})$$



***Direction 2*:**

G contains a clique of size k → is satisfiable

***Proof*:**

G contains a clique of size k

Select the k nodes

Find their respective variables in

Each of these variables must be in a different clause

By how G was constructed

Each variable can be set to TRUE without issue

By definition of how edges were added to G

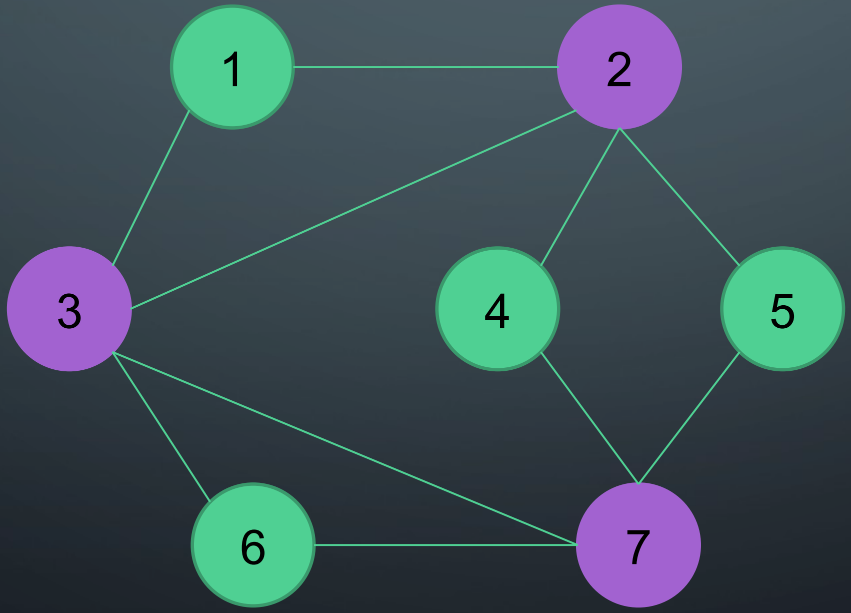
Thus, these variables must satisfy

VERTEX COVER

VERTEX COVER

A ***Vertex Cover (VC)*** on a graph G = (V,E) is a subset of vertices $S \subseteq V$ such that every edge in the graph is connected to at least one vertex in S

Decision Problem: Does a given graph G have a vertex cover of size k or smaller?



The purple nodes represent a vertex cover of size 3 on this graph. Notice that every edge touches one of these nodes

SHOWING THAT

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

As usual, this one is pretty simple

Let’s use Clique this time

SHOWING THAT

Provide a verifier TM that runs in Polynomial Time

Given graph , integer k and subset :

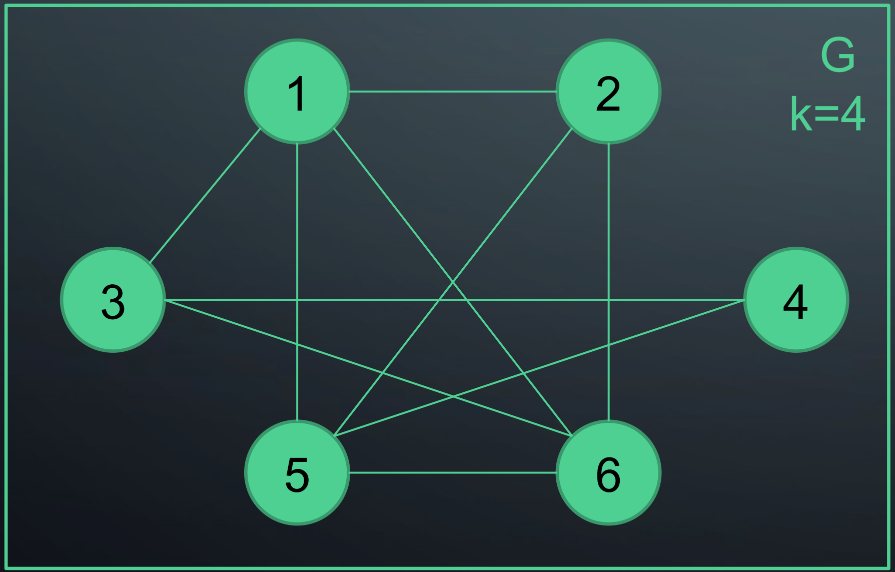
```
Verify that , if not reject
For each edge
    Check that , if not reject
else accept
```

SHOWING THAT

Given a graph G, integer k, and looking for a clique of size k

$\longrightarrow$

graph G', integer k', and looking for a vertex cover of size k'



$\longrightarrow$

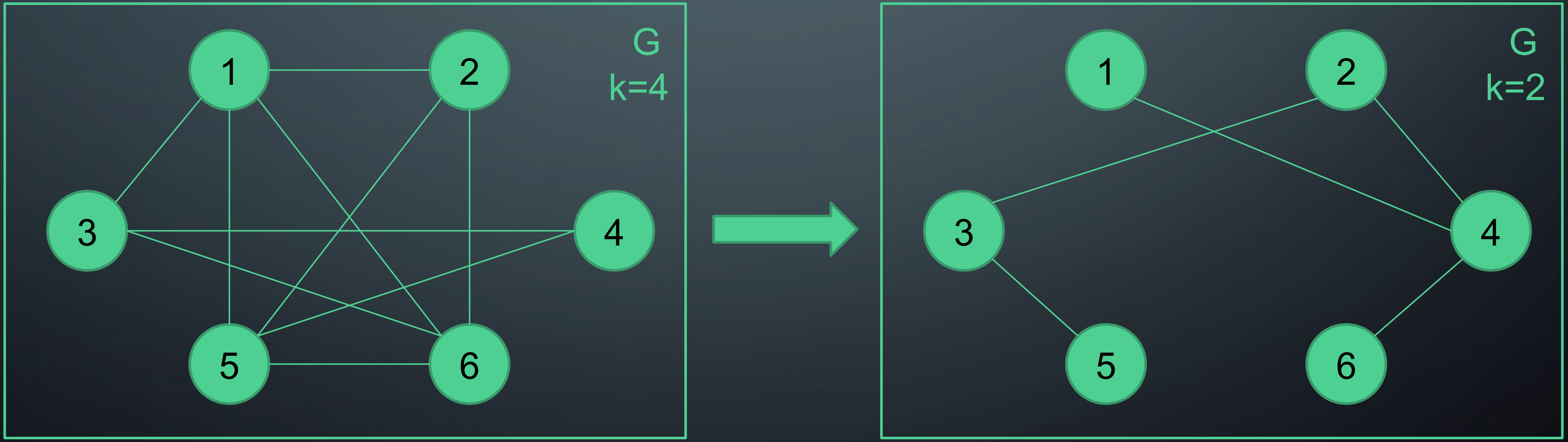
SHOWING THAT

Given a graph G, integer k, and looking for a clique of size k

$\Rightarrow$

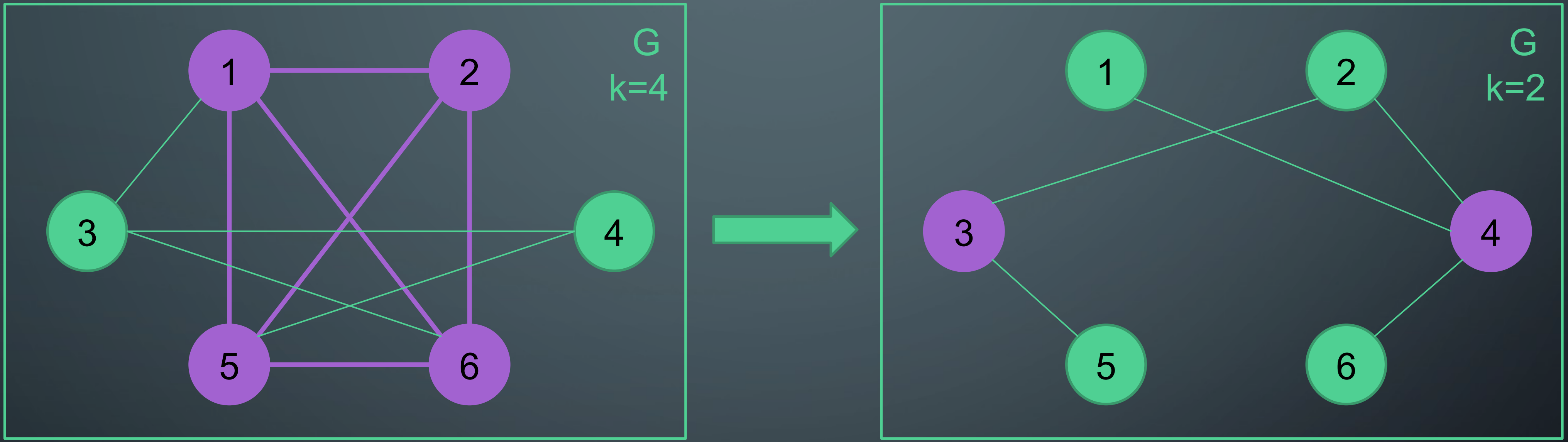
graph G', integer k', and looking for a vertex cover of size k'

Simply flip the edges that exist in G and set k to



SHOWING THAT

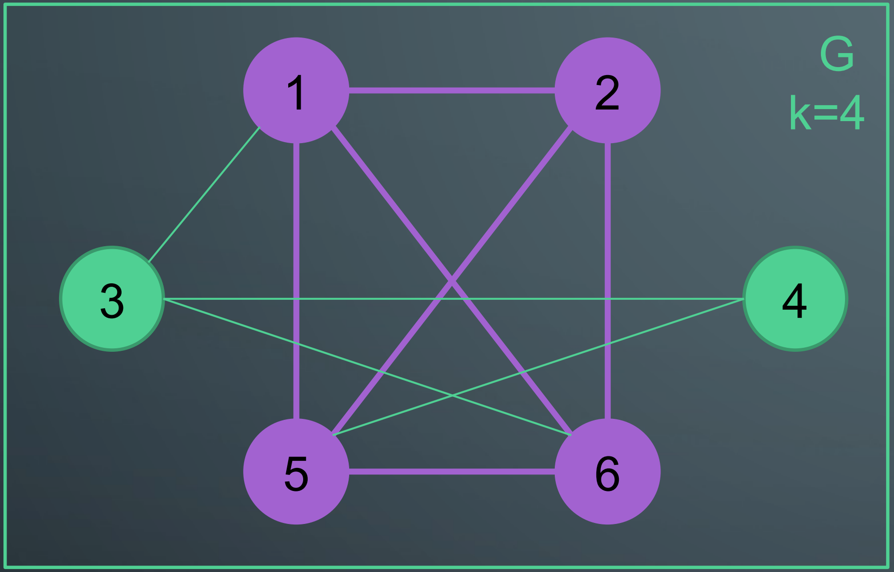
Claim: G has a clique of size k IFF G’ has a VC of size

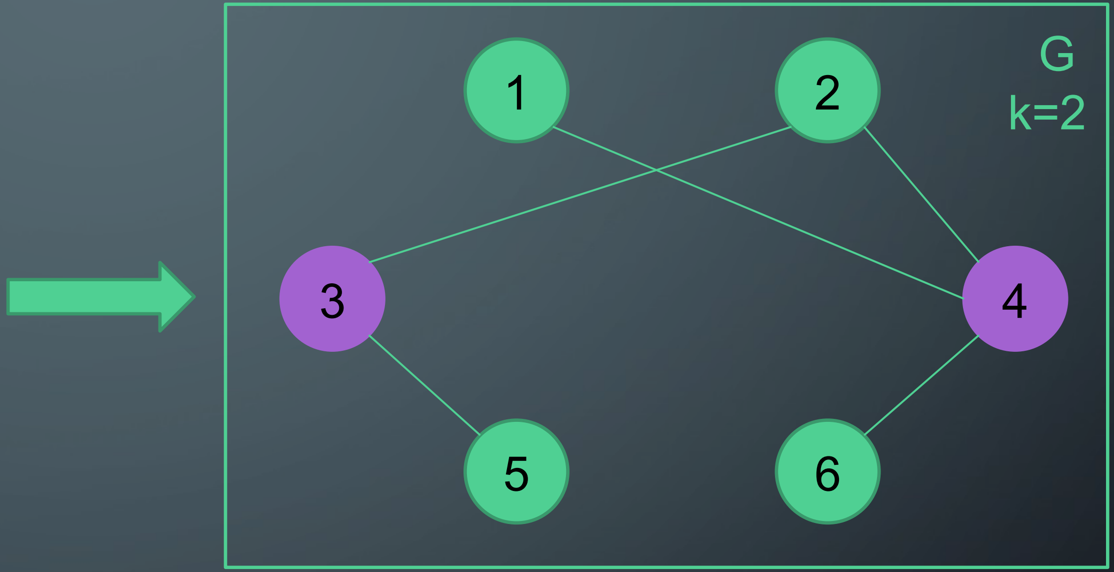


…and if the clique in G is nodes , then the cover in G’ is exactly the nodes

SHOWING THAT

Claim: G has a clique of size k IFF G' has a VC of size



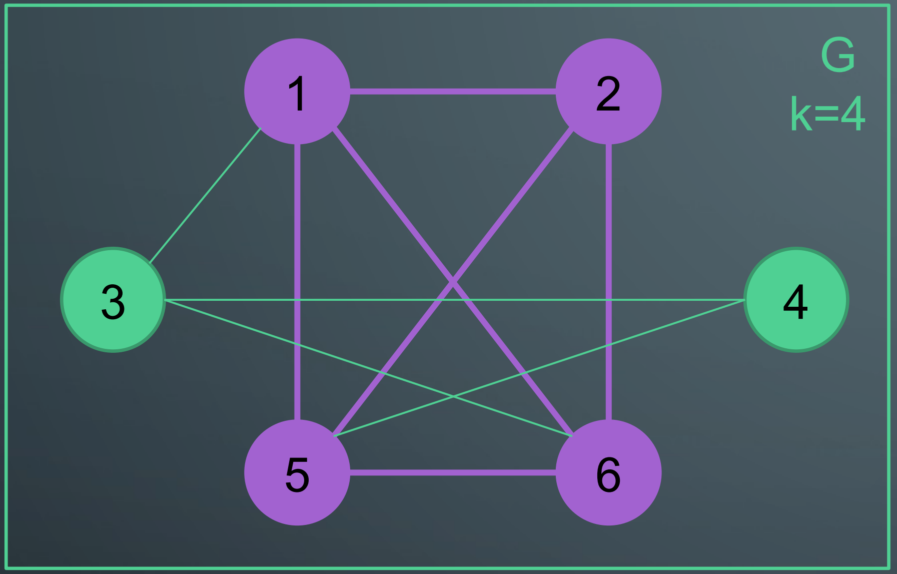


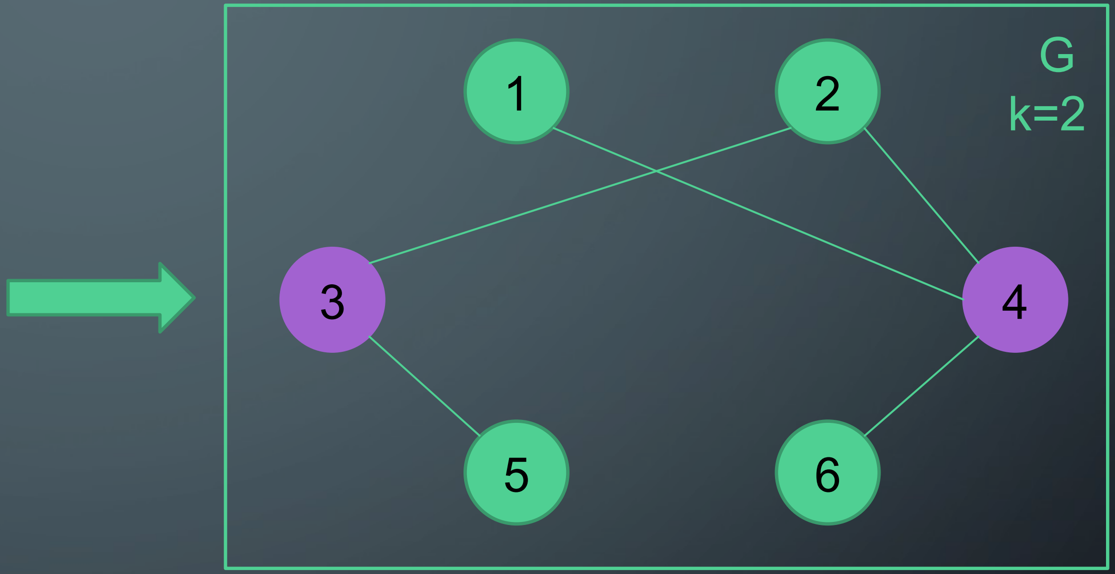
Proof Direction 1:

Suppose G has a clique of size k
Consider nodes in G'
In G, every edge between nodes in V' existed (clique), so none of these edges appear in G'
Thus every edge in G' touches a node that was not in the clique, which is the exact set
Q.E.D.

SHOWING THAT

Claim: G has a clique of size k IFF G' has a VC of size

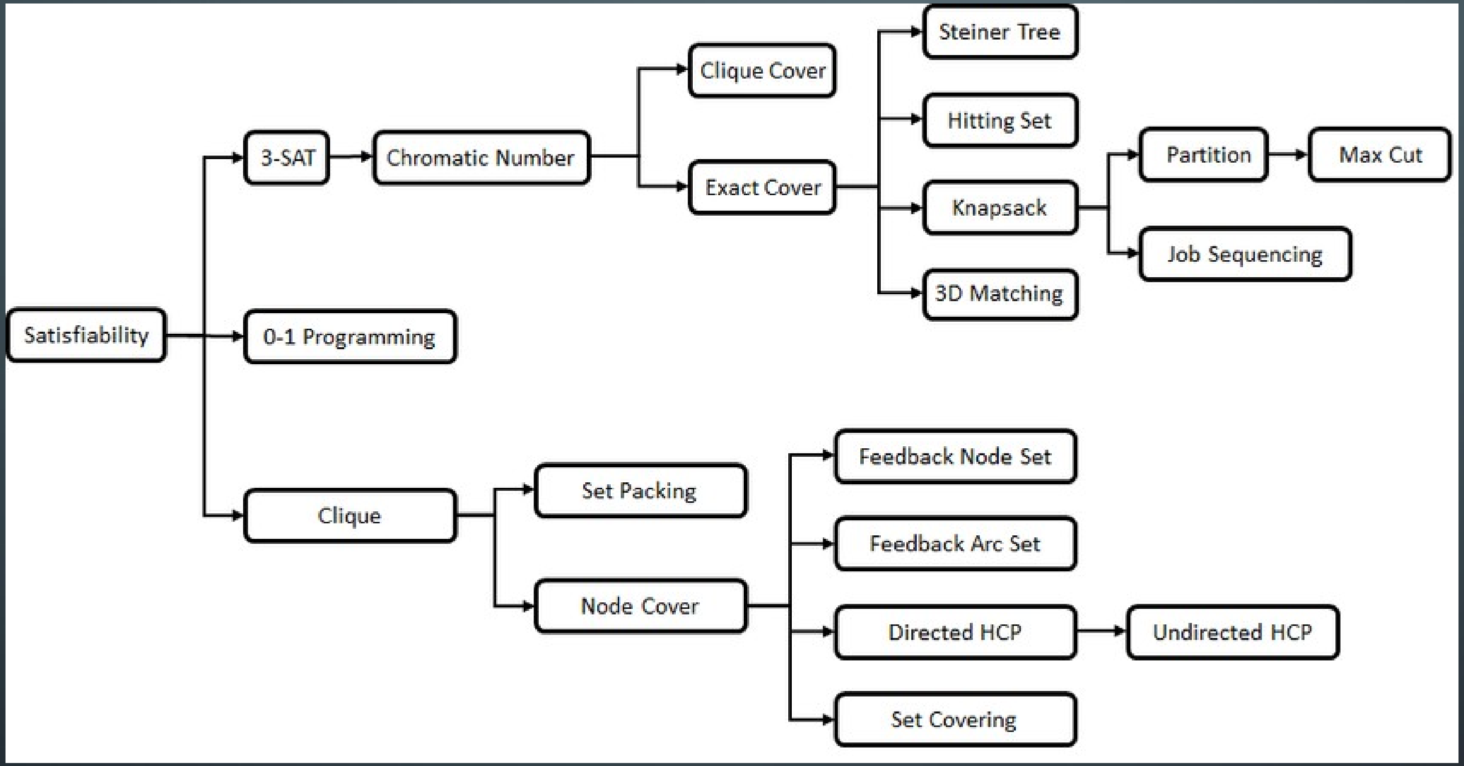




Proof Direction 2: Suppose G' has a cover of size
Consider the k nodes in G
In G', no edge between nodes in V'' exists, otherwise V' would not be a vertex cover
Thus, in G every edge between nodes in V'' exists. This is definition of a clique
Q.E.D.

MORE ON REDUCTIONS

MORE REDUCTIONS!



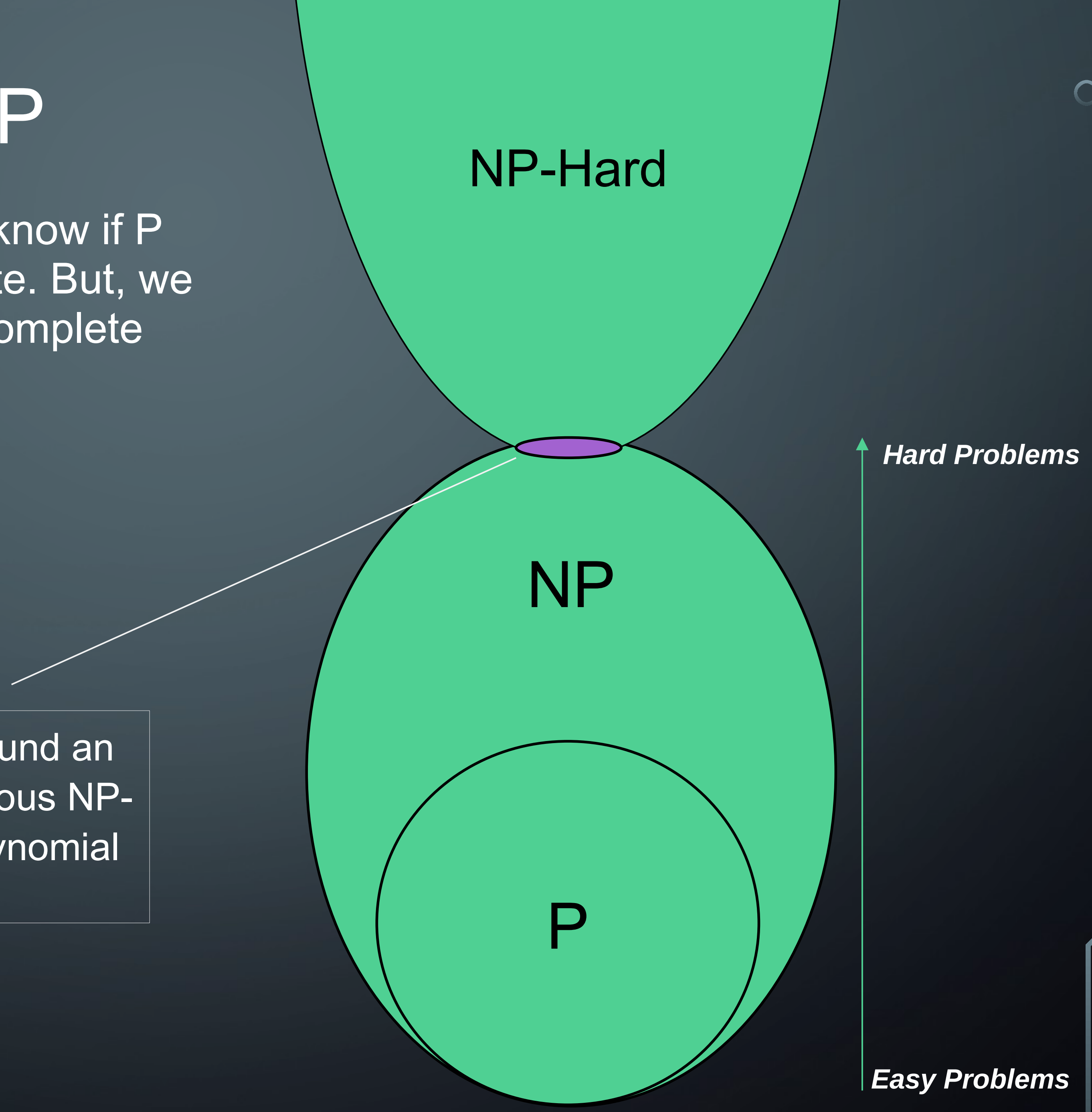
In 1972, Richard Karp showed a number of problems were NP-complete

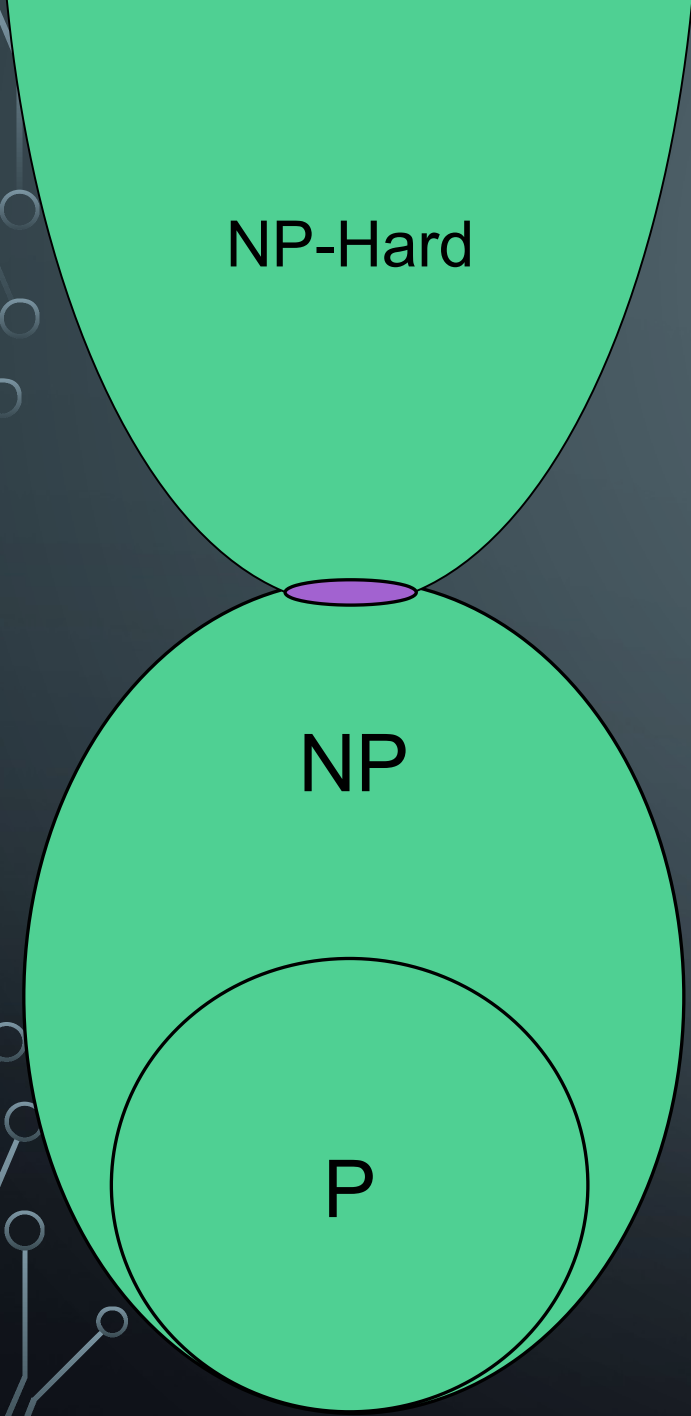
The problems were known to be "hard", but how "hard" was not really quantified until then

DOES P=NP

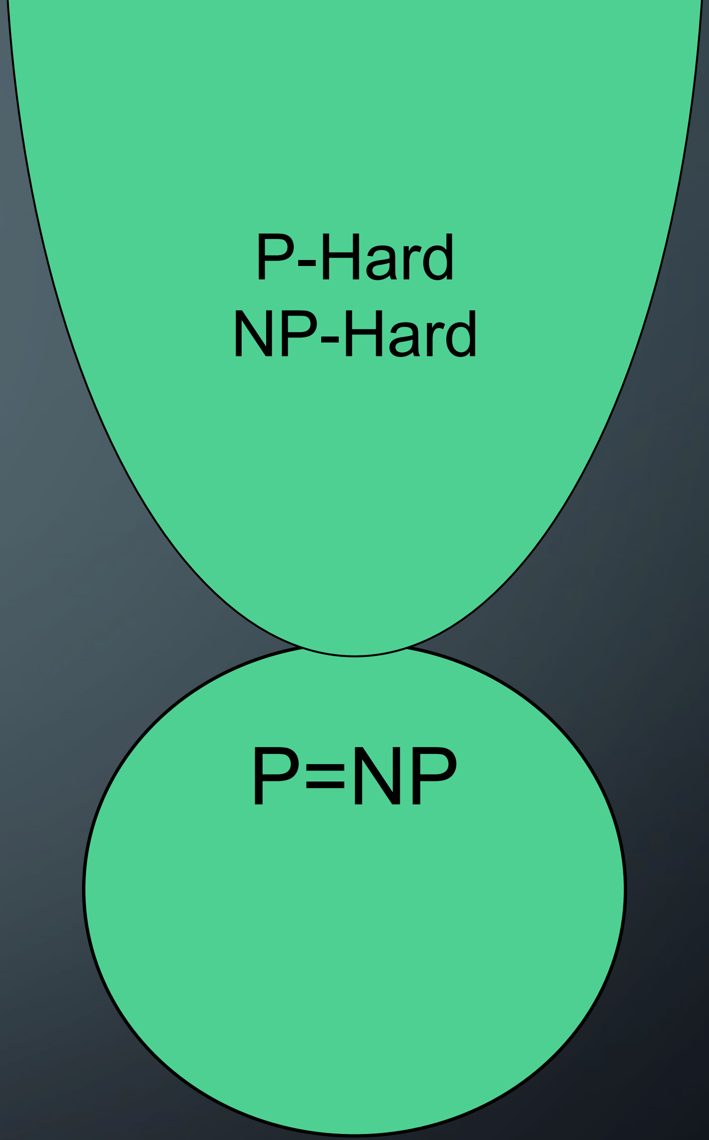
To this day, we still do not know if P and NP are distinctly separate. But, we have a lot of known NP-Complete problems

What would happen if someone found an algorithm to solve one of these famous NP-Complete problems that ran in polynomial time?





If someone finds a polynomial time algorithm to ANY np-complete problem, then

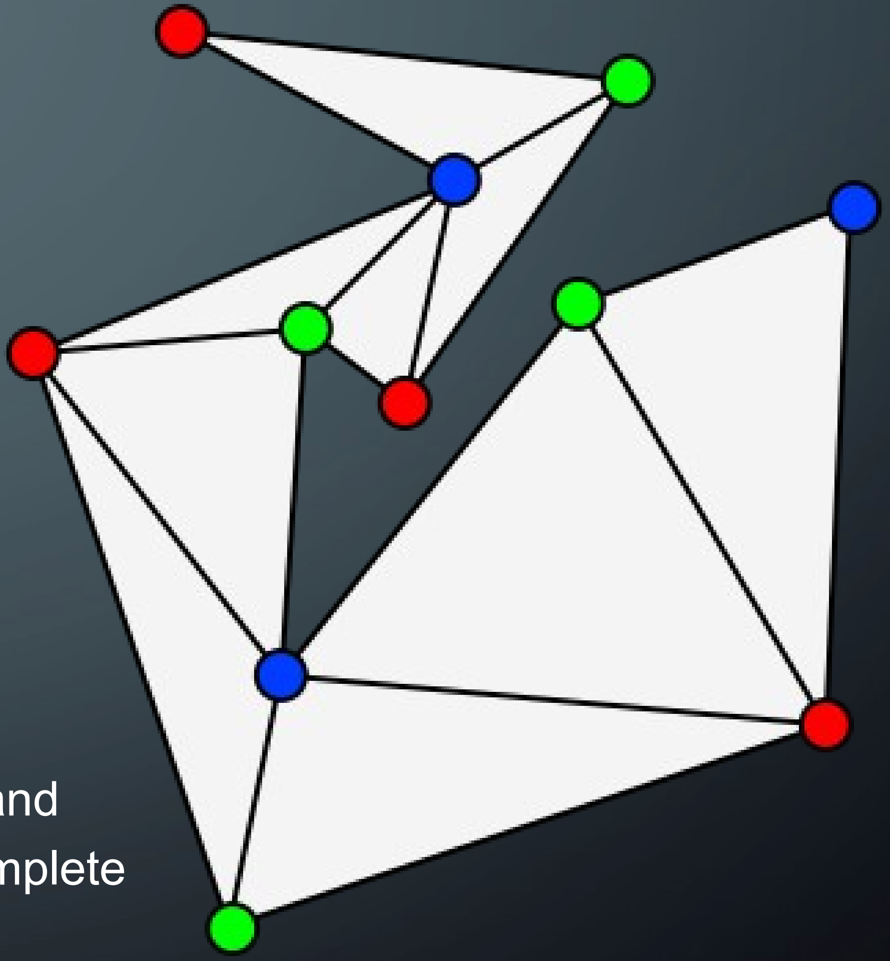


Suddenly, through various reductions there is a fast (polynomial) algorithm for every NP problem!

ANOTHER REDUCTION: 3-COLORING

3-COLORING

***Problem Statement*:**
Given graph G, and three colors c1, c2, c3 (not really given as input), can we color the graph with these colors such that no adjacent nodes have the same color.



Turns out that 3-Coloring is NP-Complete, and problems like this should start “feeling” NP-Complete to you.

SHOWING THAT

To show that , we must show both that:

Provide a verifier TM that runs in Polynomial Time

3

As usual, this one is pretty simple

Let's use 3-SAT this time

SHOWING THAT

Provide a verifier TM that runs in Polynomial Time

```
**Given graph , and color assignments C for each node in V:**

Verify that only 3 unique colors exist in C, if not reject
Verify that each node was assigned exactly one color in C, if not reject

For each edge
    Check that , if not reject

else accept
```

$3SAT \leq_p 3C$

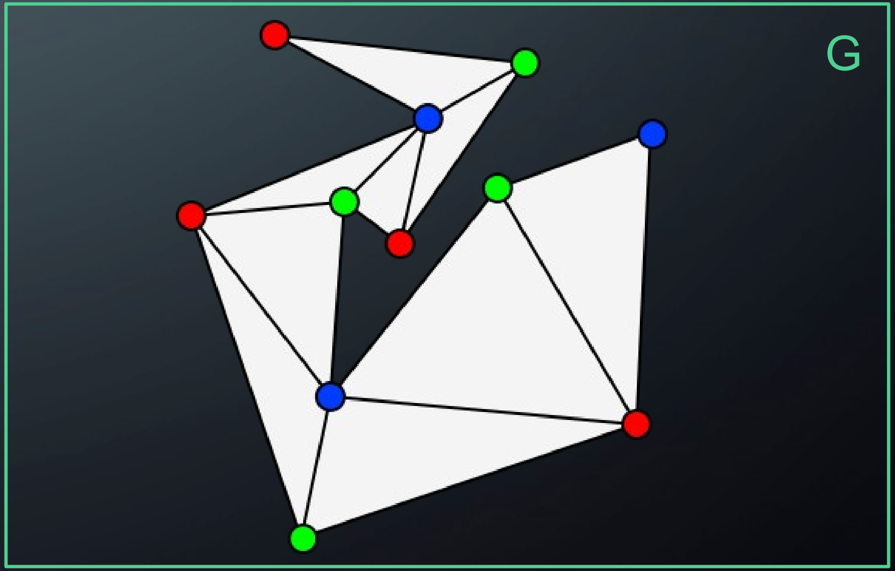
Given a boolean formula in 3-CNF that we want to test satisfiability on

$\rightarrow$

graph G that is 3-Colorable if and only if is satisfiable

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

$\rightarrow$



$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

The graph we construct needs to:

- Model the fact that variables can only be set to True and False.

- Model the variables and the fact that each variable XOR its negation can be True.

- Model the fact that at least one variable per clause must be chosen.

$3SAT \leq_p 3C$

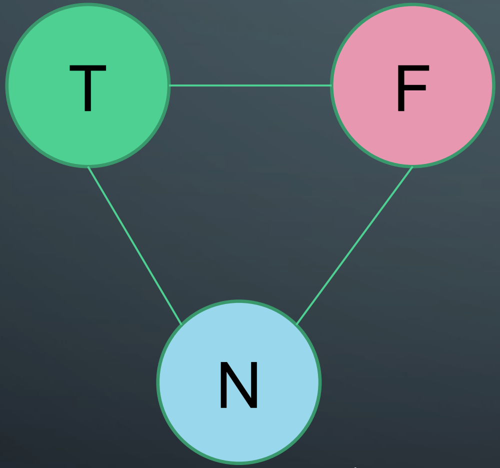
$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

The graph we construct needs to:

- ***Model the fact that variables can only be set to True and False.***

- Model the variables and the fact that each variable XOR its negation can be True.

- Model the fact that at least one variable per clause must be chosen.



Whatever color these top two nodes are assigned will represent True / False for the remainder of the coloring.

Notice that if we connect a variable (node) to this Neutral node, then that variable MUST take on the color assigned to True or False

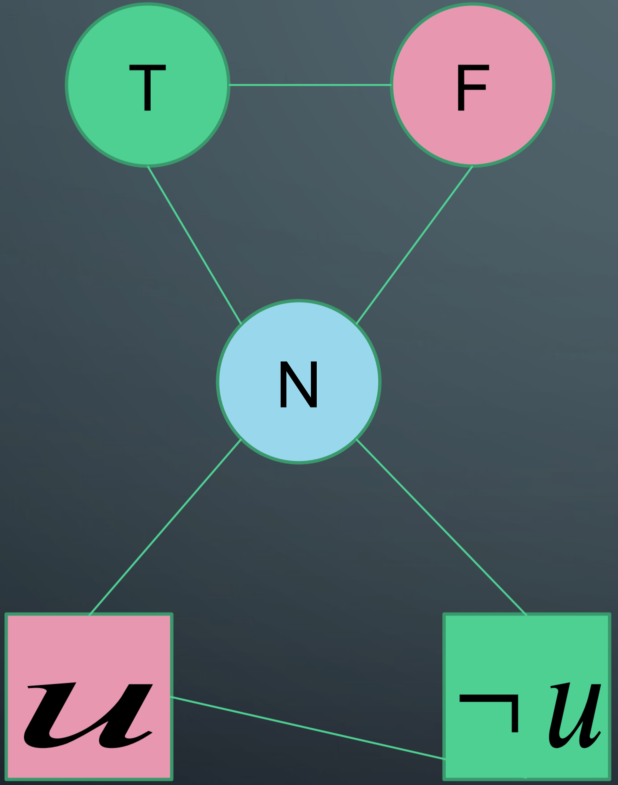
$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

The graph we construct needs to:

- ***<u>Model the variables and the fact that each variable XOR its negation can be True.</u>***

- Model the fact that at least one variable per clause must be chosen.



This variable cannot take the Neutral color so it must be the opposite of whatever u took. One is true, the other is false.

This variable is connect to the Neutral, so it MUST take the True color or the false color.

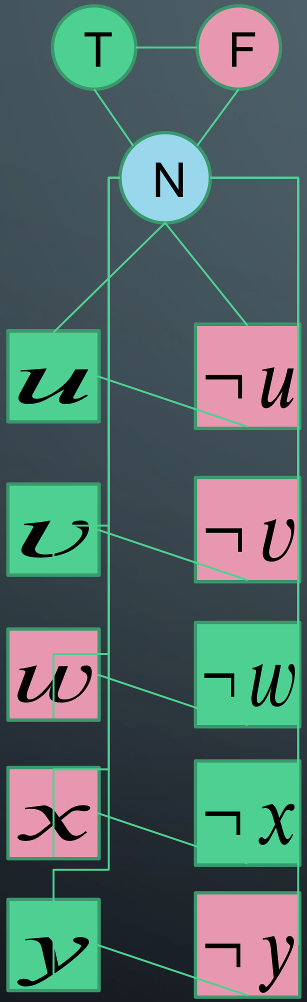
$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

The graph we construct needs to:

- ***Model the variables and the fact that each variable XOR its negation can be True.***

- Model the fact that at least one variable per clause must be chosen.



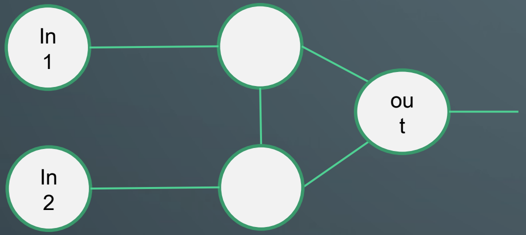
So far, so good. By assigning every node one of three colors, we can effectively choose which variables to set to True / False!

$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

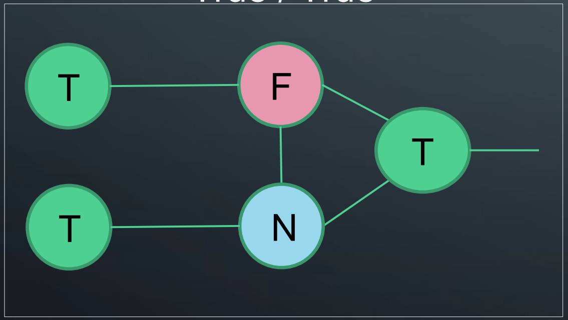
The graph we construct needs to:

- ***Model the fact that at least one variable per clause must be chosen.***

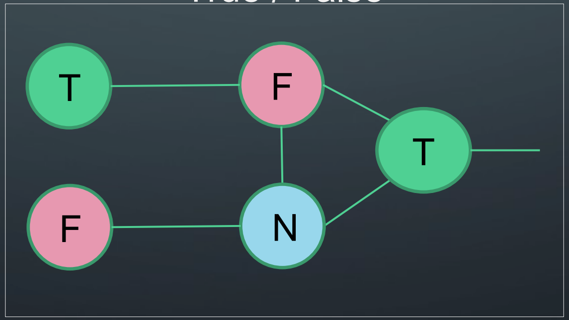


Claim:
Three fully-connected nodes can act as an OR gate. The output node can be colored with the True color IFF at least one of the input nodes is colored with the true color.

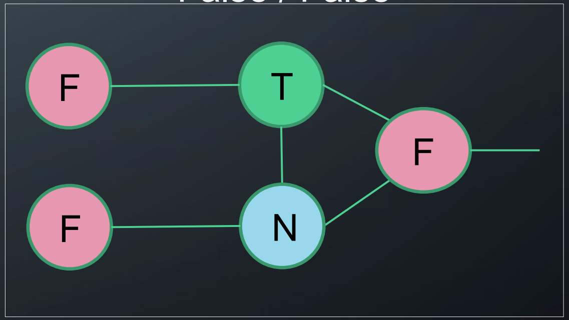
True / True



True / False



False / False



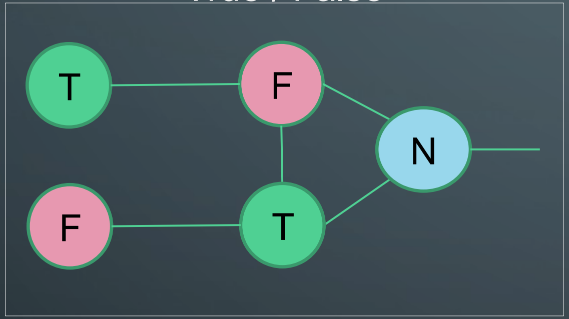
$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

The graph we construct needs to:

- ***Model the fact that at least one variable per clause must be chosen.***

True / False



Quick Aside:
Notice that in some cases, we can color the output to the neutral color. We will handle this issue in a moment.

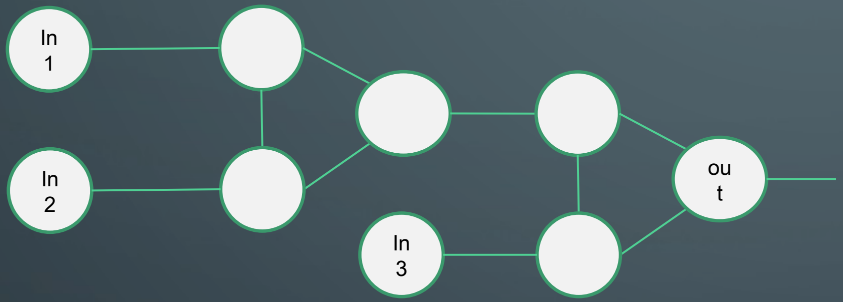
But, it is still the case that we CAN color the output True if and only if one of the input nodes is colored True.

$3\,SAT \leq_p 3\,C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

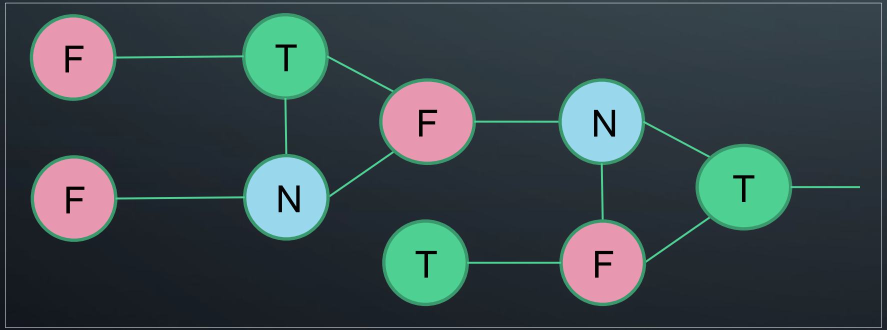
The graph we construct needs to:

- ***Model the fact that at least one variable per clause must be chosen.***

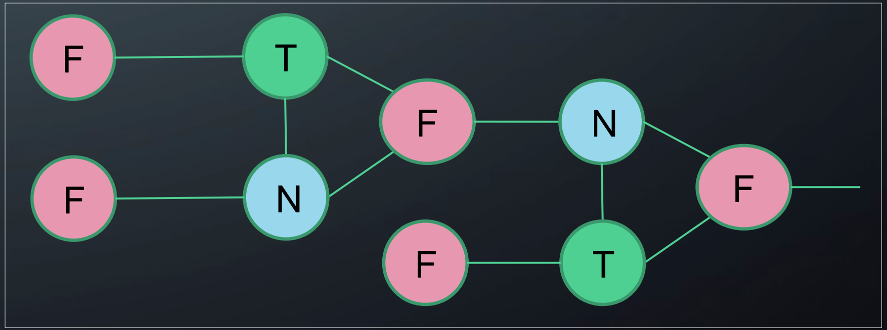


Corollary:
We can combine two of these widgets to produce an OR gate across three variables. The output is colorable as TRUE if and only if one of the three inputs is colored TRUE

Example 1: False / False / True



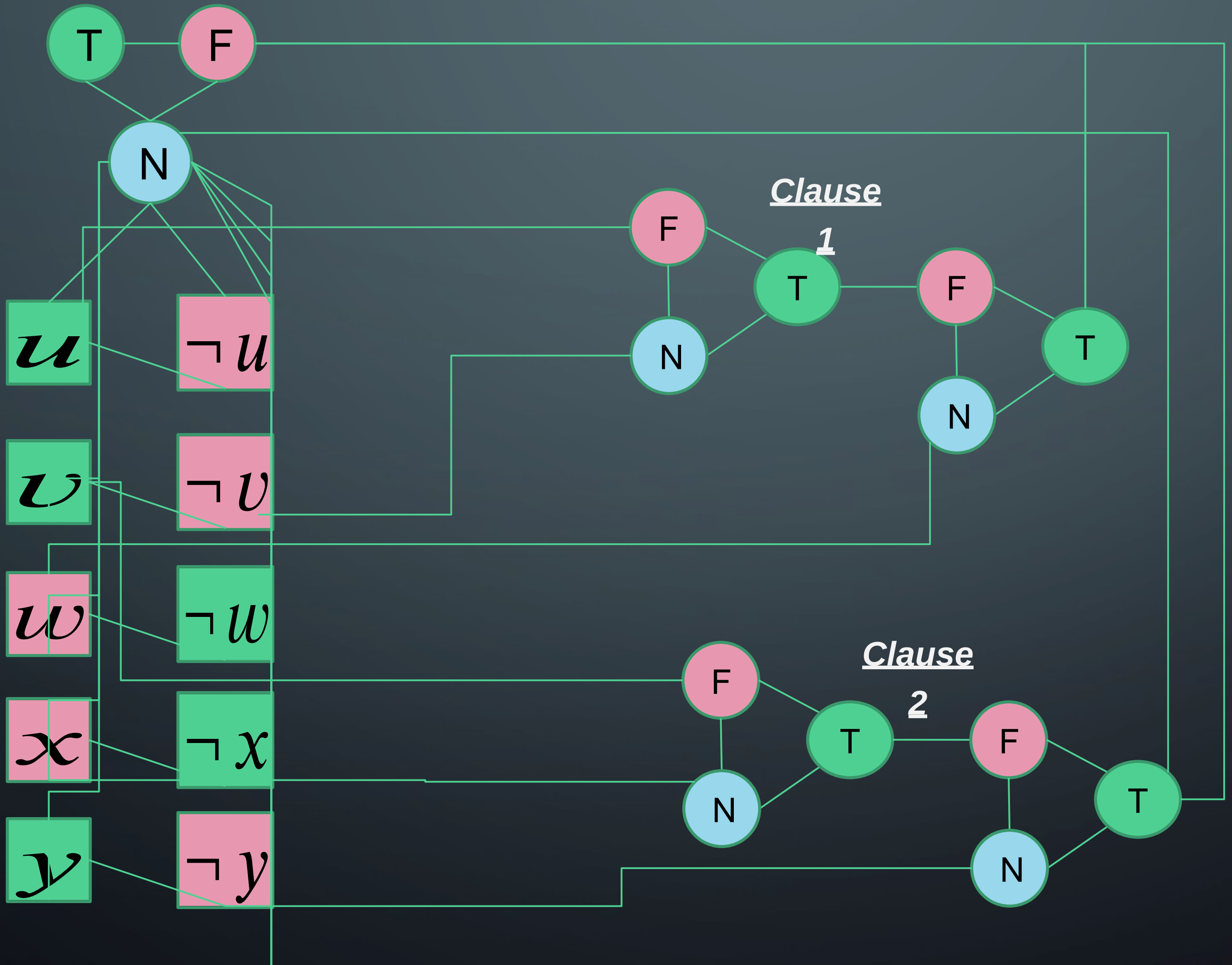
Example 2: False / False / False



$3SAT \leq_p 3C$

$$\theta = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$$

Notice that the outputs of the gates are connected to the False and Neutral terminals. This is because we NEED the output of each clause to be colored True!



(VERY INFORMAL) PROOF OF REDUCTION

- Sat(Φ) → G is 3-Colorable
  - Assume Φ is satisfiable
  - 3 colors (true, false, base)
  - Color B,T,F with these colors
  - Color variable nodes with T and F depending on their satisfying values for Φ
  - Or gates always colorable so that they represent correct OR (output is true iff one or more inputs true)
  - Thus G is 3-Colorable

- G is 3-Colorable → Sat(Φ)
  - Assume G is 3-Colorable
  - Color the graph
  - Let the colors of the B,T,F nodes represent base, true, and false respectively.
  - Re-arrange OR gate colors slightly if necessary so output is always T or F
  - Let variable assignments be the color they were given
  - These assignments satisfy Φ

CONCLUSIONS / OTHER COMPLEXITY CLASSES

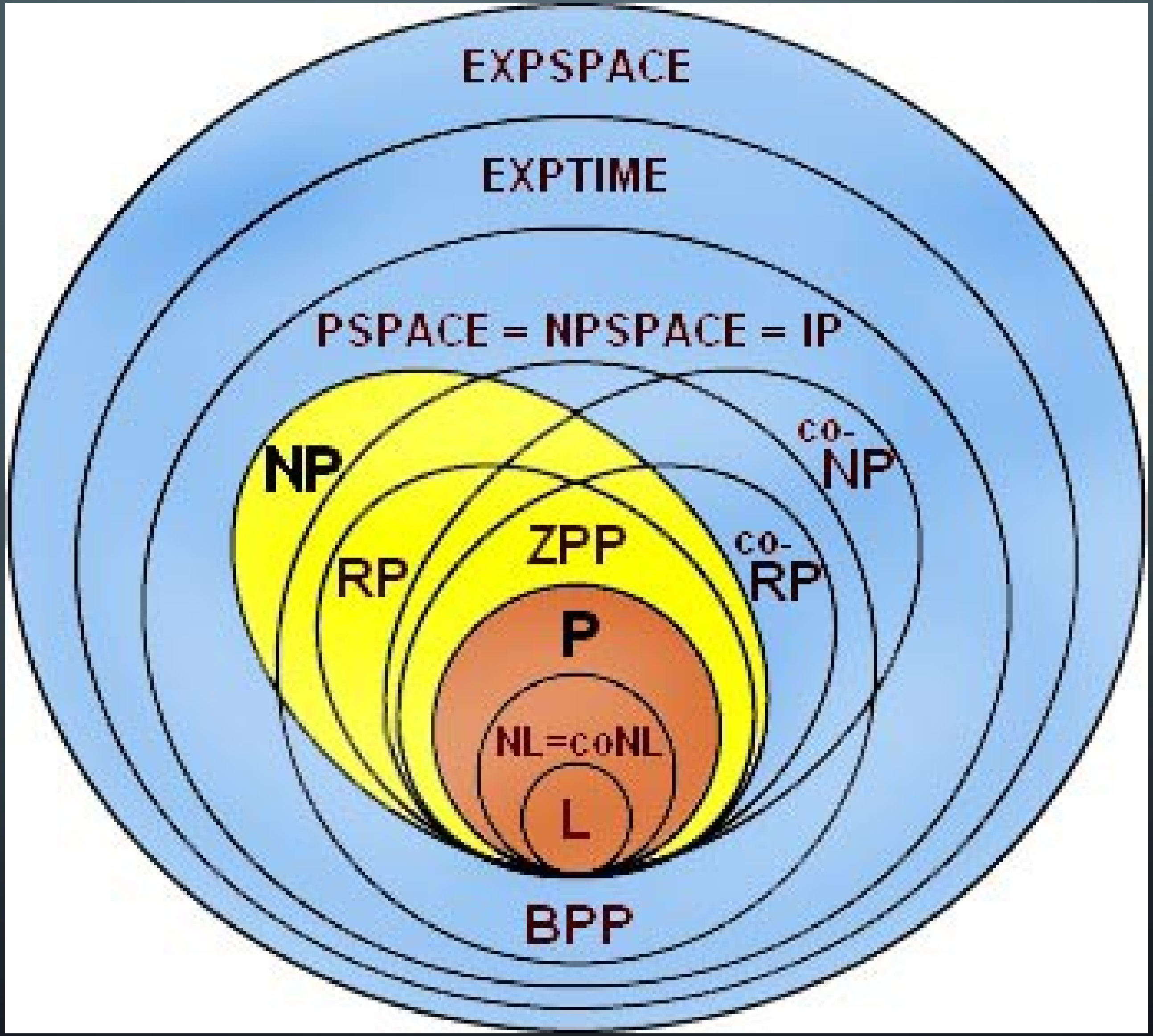
A COUPLE COMPLEXITY CLASSES WE WON'T SEE:

- EXPTIME
  - Deterministic exponential time
- NEXPTIME
  - Non-Deterministic exponential time
- PSPACE
  - Deterministic Polynomial Space
- NPSPACE
  - Non-Deterministic Polynomial Space
- EXPSPACE
  - Deterministic Exponential Space
- NEXPSPACE
  - Non-Deterministic Exponential Space

PSPACE = NPSPACE and EXPSPACE = NEXPSPACE

(WOAH! That's pretty cool!)

COMPLEXITY CLASS DIAGRAM



CONCLUSIONS!

In this module, we learned:

1. Problem types (function, decision, verification), runtimes of DTMs and NTMs, relationships between DTM and NTM runtimes for types of problems.

2. The basic complexity classes (P, NP, NP-Hard, NPC) and how they relate to one another.

3. What a reduction is and how it is used to compare the difficulty of two different problems.

4. How to prove that a problem is NP-Complete.