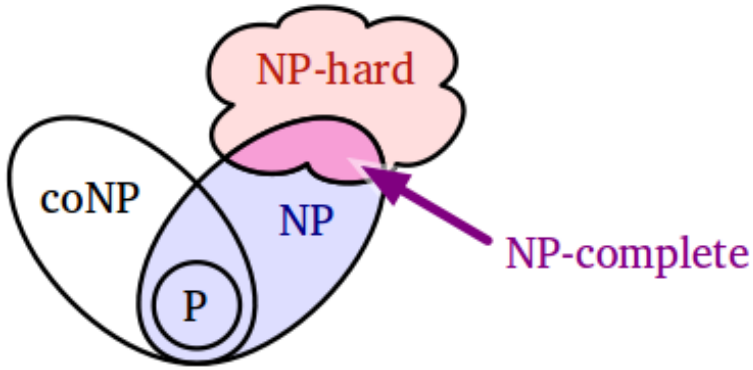
NP-completeness

Pedro Ribeiro

DCC/FCUP

2018/2019



More of what we *think* the world looks like.

Computationally Hard Problems

- **Good news:** Almost all the algorithms we've studied thus far can be solved "quickly", i.e., they are **polynomial-time algorithms**: on inputs of size n, their worst-case running time is $\mathcal{O}(n^k)$, for some (typically small) constant k.

- **Bad news:** Not all problems are like these....
  - There are problems for which there is no solution: **halting problem**
  - There are problems for which there is a solution, but not in polynomial time...

- This lecture will focus on this last set of problems. We want evidence that **some problems are intrinsically hard**. We will be particularly interested in **NP-complete** problems.

Halting Problem

The Halting Problem

Input: A program (turing-complete model) and an input
Output: Yes/No Answer: does the program halt (stop) when run?

- For instance, consider the two following programs:
  - `while (true) continue`
    Continues always in an infinite loop and never halts
  - `write "Hello World!"`
    Does halt
- On simple programs like these it is easy to decide, but on more complex program this is really problematic...
- We could run the program for a certain number of steps and see if it stops. But if the program does not halt, how do we really know if will eventually halt or continue forever?
- This is an historically important problem because one of the first to be proved as **undecidable** (by Alan Turing in 1936)

Computationally Hard Problems

Why should we care?

- **NP-complete problems** really come up all the time. Knowing they're hard lets us stop "beating our head against a wall" trying to solve them optimally, and instead:
  - Use an **heuristic**: if I can't quickly solve the problem with a good worst case time, maybe I can come up with a method for solving a reasonable fraction of the common cases.
  - Use an **approximation**: a lot of the time it is possible to come up with a fast algorithm, that doesn't solve the problem exactly but comes up with a solution we can prove is close (enough) to the optimal.
  - Use an **exponential solution**: If you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution.
  - Use a **better abstraction**: the hardness of the problem may come from ignoring some of the seemingly unimportant details of a more complicated real world problem. This can make the difference between what we can and can't solve.

Computationally Hard Problems

Some examples

- Sometimes, a problem may seem very similar to one we know that we can solve polynomially, but still they are really hard...
- **Shortest vs Longest Path**:
  Given a weighted graph, finding the shortest path between two nodes u and v can be solved polynomially (ex: Dijkstra and Bellman-Ford algorithms). However, if we want to discover the longest path between to nodes, we have an hard problem...
- **Eulerian Path vs Hamiltonian Path**:
  Given a graph, finding a path that traverses all edges exactly once (an *eulerian path*) can be solved polynomially. However, finding a path that traverses all nodes (*hamiltonian path*) is an hard problem...

Computationally Hard Problems

Some examples

- **Graph 2-coloring vs 3-coloring**:
  Given a graph, a k-coloring is a way to assign k colors to each node such that no two adjacent nodes have the same color. Deciding whether a graph admits a 2-coloring can be solved polynomially, while deciding if it admits a 3-coloring is an hard problem...

- **2-CNF satisfiability vs 3-CNF satisfiability**:
  A k-CNF (conjunctive normal formula) is an AND of clauses of ORs, each with with k boolean variables or their negations. For instance, $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF form. Knowing if 2-CNF is satisfiable (if there is an attribution of values to the variables that causes the formula to evaluate to `true`) can be solved in polynomial time, but 3-CNF satisfiability is an hard problem...

Decision vs Optimization/Search Problems

- Many problems of interests are **optimization problems**: there are many valid solutions and we wish to find the *best* solution.
- Here we will however (mainly) deal with **decision problems**, in which the answer is simply a boolean value : YES or NO.
- Nevertheless, usually **we can produce a decision version of any optimization problem** which is related to it, in the sense that if the optimization problem is "easy", than so would be the decision problem. Hence, if we can show evidence that the decision problem is "hard", we are also showing that the optimization version is "hard".

  Ex: Computing the shortest path between 2 nodes (SHORTEST-PATH) is an optimization problem. The related decision problem (PATH) is: given a graph G, nodes u and v and a constant k, is there a path between u and v of at most cost k? If we have an efficient solution for SHORTEST-PATH, than solving PATH becomes simply running that solution and checking if the answer is $\leq k$.

Classes of Problems

A first informal notion

During the lecture we will be mainly referring to three classes of problems:

- **P** (*polynomial time*). The set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved "quickly".
- **NP** (*nondeterministic polynomial time*). The set of decision problem with the following property: if the answer is `YES`, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of problems where we can verify a `YES` answer quickly, if we have the answer in front of us.
- **coNP** is essentially the opposite of NP. if the answer is `NO`, then there is a *proof* of this fact that can be checked in polynomial time.

Classes of Problems

Some examples

Let's look again at some of the problems we referred to:

- **SHORT**: knowing if there is a path of cost at most k between two nodes is NP because if we have the path itself, than checking if it is correct is just traversing it seeing if sum of the edges is $\leq k$.
- **HAMILTON**: knowing if there is an hamiltonian path is NP because if we have the path itself, than checking if it is correct is just traversing it and making sure we don't visit any nodes more than once
- **3-COLOR**: knowing if there is 3-coloring of a graph is NP because if we have the coloring, than checking it is just going trough all the edges and checking they do no not connect nodes of the same color
- **2-CNF**: knowing if 2-CNF is satisfiable is NP because if we have the values that satisfy the formula, than checking it is just (linearly) applying the ORs and ANDs and check if the result is indeed `true`.

P vs NP

- **Every problem in P is also in NP and in coNP**
  - If a problem is in P, we can verify YES answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

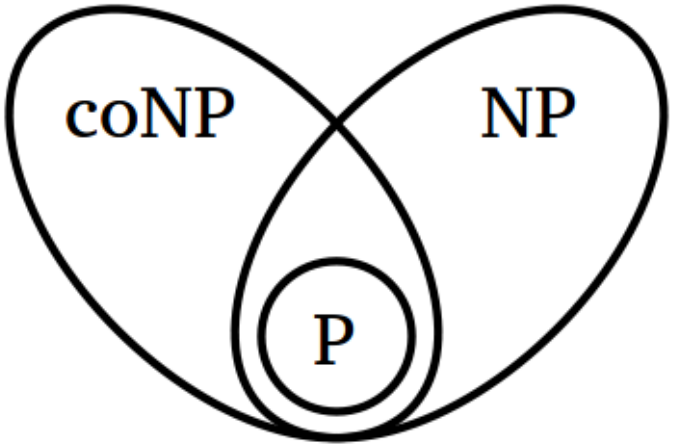
- What we don't know is if every problem in NP is also in P...

$$\mathbf{P = NP?}$$

  - Checking if a solution is correct, is *easier* than solving the problem from scratch, no? **But nobody knows how to prove it!**
  - This is one of the most important unanswered questions in Computer Science (if not from all sciences!)
  - P = NP is one of the (7) Millenium Prize problems. The Clay Mathematics Institute offers an US $1,000,000$ for anyone solving it. This reward was posted in 2000 and in fact nobody solved it (yet?)

Classes of Problems

What I (we?) think about it



What we *think* the world looks like.

(image by Jeff Erickson, UIllinois)

Classes of Problems

A typical misconception

- It is a very common mistake that **NP** corresponds to *non-polynomial*. This is really incorrect, because:
  - We are not really sure if NP problems can or cannot be solved in polynomial time ($P = NP$?)
  - There are much more harder classes of problems than NP. For example:
    - ⋆ **PSPACE**. Problems that can be solved using a polynomial amount of space.
    - ⋆ **EXPTIME**. Problems that can be solved in exponential time.
    - ⋆ **EXPSPACE**. Problems that can be solved using an exponential amount of space

      $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE$
    - ⋆ **Undecidable**. Problems we *cannot* solve.

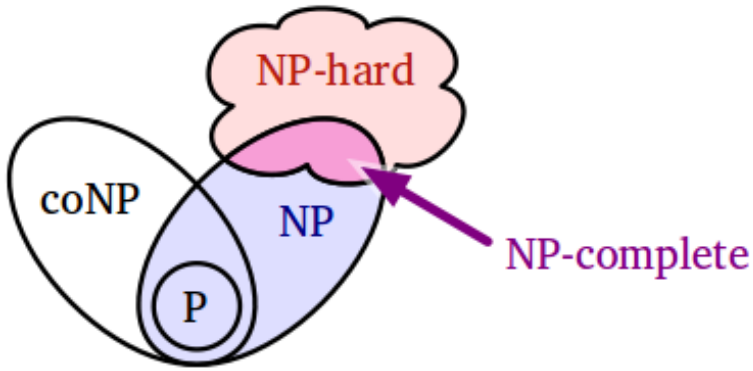
Classes of Problems

Continuing the informal notions

There are two other important classes of problems we should know:

- **NP-hard** (or NPH). A problem that is *as hard as* any problem in NP, that is, if it can be solved in polynomial time, than *every* NP problem can also be solved in polynomial time.
- **NP-complete** (or NPC). A problem that is both NP-hard and in NP.

Note that an NP-hard problem needs not to be a decision problem, nor it needs to have a way of checking solutions in polynomial time

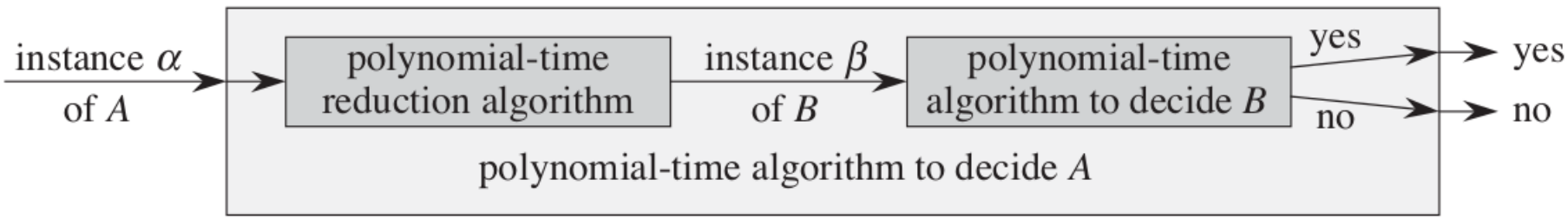


More of what we *think* the world looks like.

(image by Jeff Erickson, UIllinois)

Problem Reduction

- Consider we want to solve a problem A in polynomial time.
- Suppose we know how to solve a problem B in polynomial time.
- Suppose we have a procedure to transform any instance α of problem A into an instance β of problem B, so that
  - The transformation takes polynomial time
  - The answers are the same (the answer to α is `YES` iff the answer for β is `YES`).
- We now have a way of solving problem A in polynomial time!
  - Given any instance α of A, transform it into an instance β of B
  - Run the polynomial time algorithm for B on the instance β
  - Use the answer for β as the answer for α



Problem Reduction

- What we did is called a **problem reduction**. We reduced problem A to problem B!

Reduction

A Problem A is **poly-time** reducible to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a black-box algorithm for problem B.

- Recall we defined NP-hard as "a problem that is *as hard as* any problem in NP". What does this mean? We now have a possible answer:

  Q is NP-hard if for any other problem X in NP, X $\leq_p$ Q!

  If I can solve Q in polynomial time, than I can solve **any** problem in NP in polynomial time!

Problem Reduction

- This is very useful in order to **prove a problem is NP-complete**!
  - Suppose we have a problem A that we know is NP-hard
  - We want to show that problem B is NP-complete
  - Besides proving that B is in NP (usually "easy"), **all we need to do is to show that A $\leq_p$ B** (A is reducible to B).
  - A polynomial time algorithm for problem B would imply a polynomial solution for problem A! So B is as hard as A
  - Note that this is not valid in the "other" direction, ie, with $B \leq_p A$)

A first NP-complete problem

- All we are missing now is a "root" problem that we know is NP-complete! All other problems can "become" NP-complete by reducing to this problem...

The Cook-Levin Theorem

SAT (Boolean Satisfiability) is NP-complete

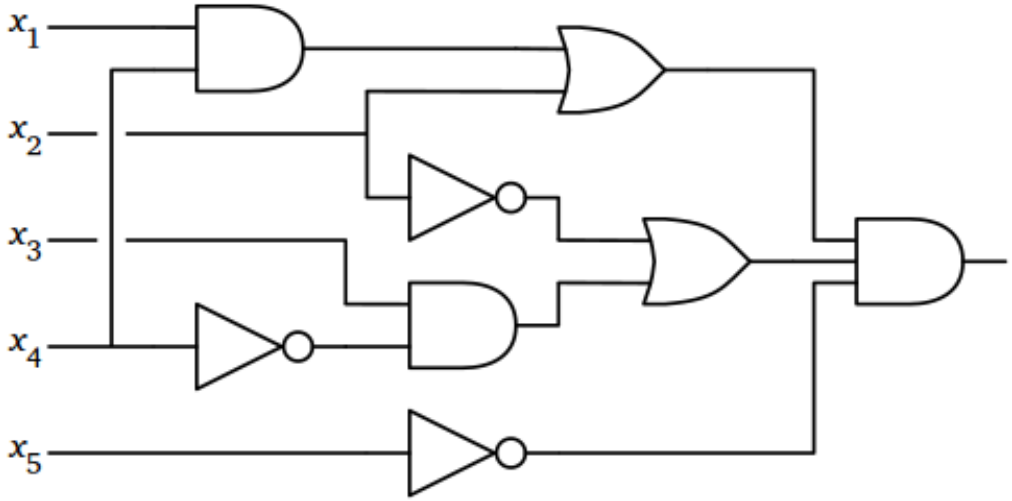
SAT - Is there an "interpretation" that satisfies a general a boolean formula written? (in other versions SAT concerns only CNF formulas)
Is there an assignment of values `TRUE` or `FALSE` to the variables in such a way that the formula evaluates do `TRUE`?

We'll give a brief sketch of a proof of why SAT is NP-complete

CIRCUIT-SAT

Consider the problem **CIRCUIT-SAT**: determining if a boolean circuit is satisfiable:

An AND gate, an OR gate, and a NOT gate.



A boolean circuit. inputs enter from the left, and the output leaves to the right.

SAT and CIRCUIT-SAT

To prove SAT is NP-complete we need to prove:

1) SAT is in NP
2) SAT is NP-hard

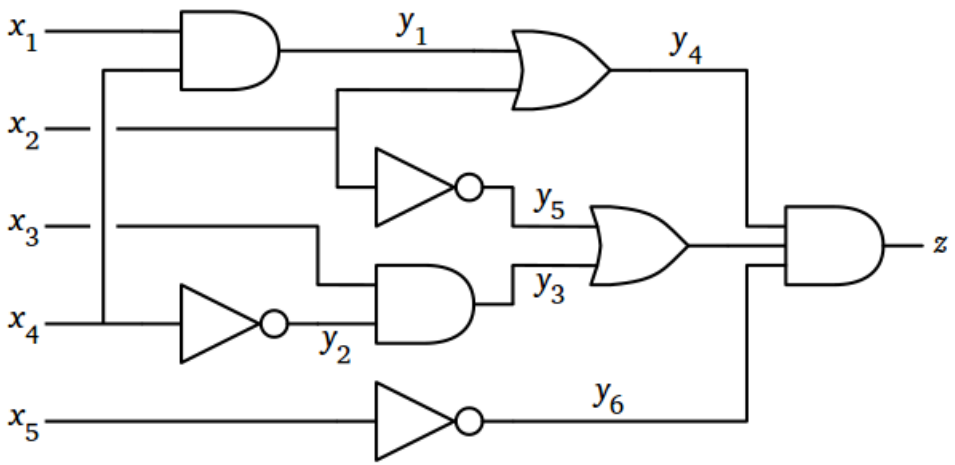
1) SAT is in NP means we have a *proof* of satisfiability we can check in polynomial time.

Imagine we have a valid assignment that satisfies a formula. To check it is correct we just need to "evaluate" the formula! And we can (trivially) do it in polynomial time

2) To prove SAT is NP-hard we will use CIRCUIT-SAT. Let's start by reducing CIRCUIT-SAT to SAT (CIRCUIT-SAT $\leq_p$ SAT). If we can show this, then all we need afterwards is to show that CIRCUIT-SAT is NP-hard (in fact it is NP-complete, like SAT)

Reducing CIRCUIT-SAT to SAT

Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates separated by ANDs. For example:



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

(image by Jeff Erickson, UIllinois)

NP-hardness of CIRCUIT-SAT

Now we still need to prove that CIRCUIT-SAT is NP-hard (and this will be our "root"). A very brief sketch of a proof, giving the intuition:

The "simple but **big idea**": every program is essentially a circuit!

Consider a problem A which is in NP. Then for every instance x we have a proof that we can verify with algorithm A in polynomial time.

We can convert the verification algorithm A into a circuit C that simulates the algorithm step by step.

Since we know that "everything" (including the number of steps A takes) is polynomially bounded by the size of x, then this circuit can be constructed in polynomial time.

We end up with a circuit C that is satisfiable if and only if the input to our problem has an YES answer!

3SAT is NP-complete

We now know that CIRCUIT-SAT and SAT are NP-complete. We can use it as the basis for proving any other problem is also NP-complete.

Let's start with a more "restricted" version of SAT:

3SAT: is a 3-CNF formula satisfiable?
(recall CNF is conjunctive normal form)

Again, what we need to prove is the following:

1) 3SAT is in NP
2) 3SAT is NP-hard

1) is "trivial". If we have the assignment that evaluates to TRUE, than we can just (linearly) evaluate the formula.

3SAT is NP-complete

Now we have two choices for problem we can reduce to 3SAT: SAT and its "circuit equivalent": CIRCUIT-SAT. Let's choose this last one.

A trivial algorithm could end up having an exponential number of clauses.

But we can do the following:

1) Make sure every gate has only two inputs at most. If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ input gates.

3SAT is NP-complete

2) Replace every gate by a CNF formula. Using the three types of gate NOT, AND, OR (we could have more gates, but they can essentially be represented by these):

$$
\begin{aligned}
a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\
a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\
a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b})
\end{aligned}
$$

3) Now transform every clause with one or two literals into a clause with three literals, introducing new variables:

$$
\begin{aligned}
a &\longmapsto (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\
a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x})
\end{aligned}
$$

(images by Jeff Erickson, UIllinois)

3SAT is NP-complete

In the end we get a 3CNF formula in which each gate corresponds to at most 5 clauses! Hence, **3SAT is NP-complete**. For instance, the formula for the circuit given before would become:

$$
\begin{aligned}
&(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\
&\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\
&\wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\
&\wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\
&\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\
&\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\
&\wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\
&\wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\
&\wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \\
&\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{aligned}
$$

(image by Jeff Erickson, UIllinois)

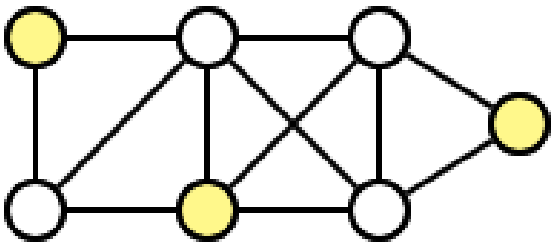
IndSet is NP-complete (from 3SAT)

We now have 3 NP-complete problems: CIRCUIT-SAT, SAT and 3SAT

Let's continue our "journey", but now going to the graphs realm.

Independent Set (IndSet): consider an unweighted graph G. An **independent set** in G is a subset of vertices of G with no edges between them.
Optimization version: What is the size of the largest independent set in G?
Decision version: Does G contain an independent set of size $\geq k$?

Example of an independent set of size 3:



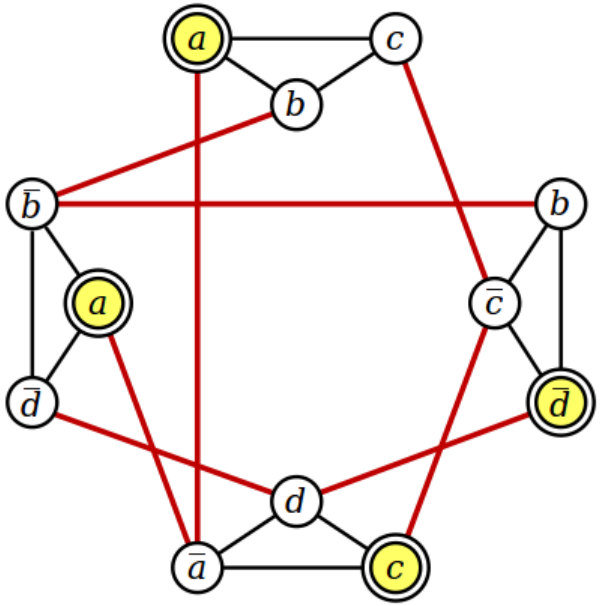
IndSet is in NP, since we can check if a solution is valid by just iterating trough all nodes and checking if there is no connection to any other node.

IndSet is NP-complete (from 3SAT)

Let's now describe a **reduction from 3SAT**. We need to describe a way to (polynomially) transform a 3CNF formula into a graph that has an independent set of a certain size only if the formula is satisfiable

Construct a graph of where we have one node for each literal of each clause. Two nodes are connected if: (1) they correspond to literals in the same clause; (2) they correspond to a variable and its negation.

For instance, the formula $(a \vee b \vee c) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee d) \wedge (a \vee \neg b \vee \neg d)$ is transformed into the following graph:



(image by Jeff Erickson, UIllinois)

IndSet is NP-complete (from 3SAT)

Now, suppose that the initial formula had k clauses. Then **the formula is satisfiable if and only if the graph has an independent set of size** k

IndSet $\rightarrow$ **Satisfying:** if the graph has an independent set of k vertices, then each vertex comes from a different clause. To obtain a satisfying assignment, we assign TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.

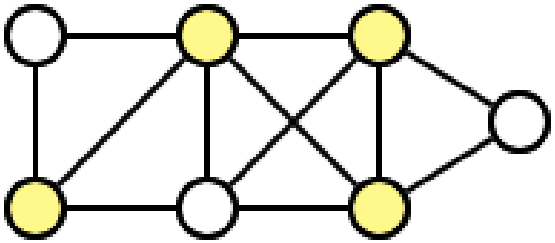
Satisfying $\rightarrow$ **IndSet:** if we have a satisfying assignement, then we can choose at least one literal in each clause that is TRUE. Those literals form an independent set in the graph.

MaxIndSet is NP-complete. ■

VERTEX-COVER is NP-complete (from IndSet)

VERTEX-COVER: a *vertex cover* is a set of nodes such that each edge it incident/adjacent to at least one of them
Optimization version: What is the size of the smallest vertex cover of G?
Decision version: Does G contain a vertex cover of size $\leq k$?

Example of a vertex cover of size 4:



VERTEX-COVER is in NP, since we can check if a solution is valid by just checking for all edges it they are covered by one of the selected nodes

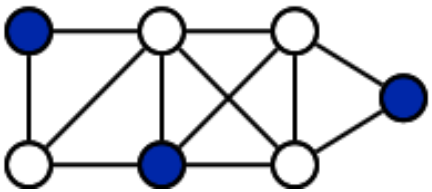
VERTEX-COVER is NP-complete (from IndSet)

Let's now describe a **reduction from IndSet**.

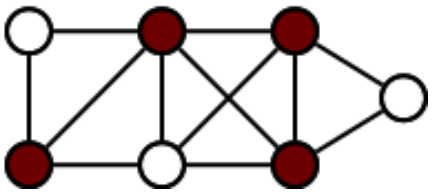
If C is a vertex cover of graph G with vertex set V, then $V - C$ is an independent set. Also, if S is an independent set, then $V - S$ is a vertex cover.

So the reduction from IndSet to VERTEX-COVER is very simple: given an instance (G, k) for IndSet, produce the instance $(G, n - k)$ for VERTEX-COVER, where $n = |V|$. In other words, to solve the question "is there an independent set of size $\geq k$, just solve the question "is there a vertex cover of size $\leq n - k$.

Independent Set



Vertex Cover

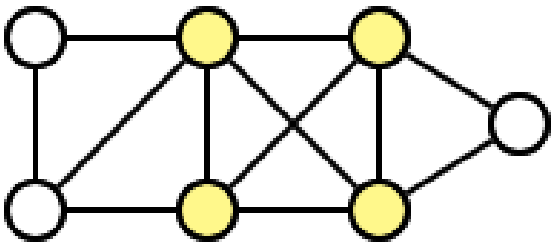


VERTEX-COVER is NP-complete. ■

CLIQUE is NP-complete (from IndSet)

Clique (CLIQUE): a *clique* is a complete graph, that is, a graph where every pair of nodes is connected by an edge.
Optimization version: What is the size of the largest subgraph of G which is a clique?
Decision version: Does G contain a clique of size $\geq k$?

Example of a clique of size 4:



Clique is in NP, since we can check if a solution is valid by just checking if all pairs of nodes are connected.

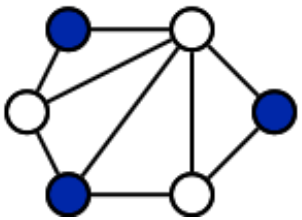
CLIQUE is NP-complete (from IndSet)

Let's now describe a **reduction from IndSet**.

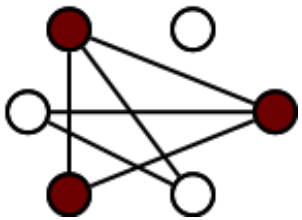
Let H be the **complement** or **inverse** of a graph G (it has the same nodes such that two distinct are connected if and only if they are not connected in G). If S is a vertex cover of graph G, then S is a clique in H.

So the reduction from IndSet to CLIQUE is very simple: given an instance (G, k) for IndSet, produce the instance (H, k) for CLIQUE. In other words, to solve the question "is there an independent set of size $\geq k$, just solve the question "is there a clique of size $\geq k$ in the complement graph".

Independent Set



Clique



CLIQUE is NP-complete. ■