Chapter 1. Tractability

Algorithmics and Programming III

FIB

Slides by Antoni Lozano
(with editions by Enric Rodríguez)

Q1 2019–2020

1 Classes
- Decision problems
- Polynomial and exponential time
- Class NP

2 Reductions
- Motivation
- Concept of reduction
- Examples and properties

3 NP-completeness
- NP-completeness theory
- NP-complete problems

Classes

- In this course we are interested in solving hard problems efficiently.

Classes

- In this course we are interested in solving hard problems efficiently.
- But how do we justify that a problem is “hard”?

Classes

- In this course we are interested in solving hard problems efficiently.
- But how do we justify that a problem is “hard”?
- Complexity theory can help us in that.
- It classifies problems according to the resources (time, space) needed to solve them with the best of the available algorithms.
- Next we will study some basic concepts of complexity theory.

Decision problems

- In a computational problem, given an input,
  we have to produce a solution as an answer.

- To simplify our classification, we will focus on decision problems.

Decision problem

A decision problem is a problem in which the answer is YES or NO.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.
- **3-colorability**: given a graph, to determine whether it is 3-colorable.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.
- **3-colorability**: given a graph, to determine whether it is 3-colorable.
- **connectivity**: given a graph, to determine whether it is connected.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.
- **3-colorability**: given a graph, to determine whether it is 3-colorable.
- **connectivity**: given a graph, to determine whether it is connected.
- **reachability**: given a graph $G = (V, E)$ and two vertices $i, j \in V$, to determine whether there is a path in G from i to j.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.
- **3-colorability**: given a graph, to determine whether it is 3-colorable.
- **connectivity**: given a graph, to determine whether it is connected.
- **reachability**: given a graph $G = (V, E)$ and two vertices $i, j \in V$, to determine whether there is a path in G from i to j.
- **shortest path**: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j of length at **most** k.

Decision problems

Some examples of decision problems:

- **primality**: given a natural number, to determine whether it is prime.
- **sortedness**: given a list of numbers, to determine whether it is sorted.
- **3-colorability**: given a graph, to determine whether it is 3-colorable.
- **connectivity**: given a graph, to determine whether it is connected.
- **reachability**: given a graph $G = (V, E)$ and two vertices $i, j \in V$, to determine whether there is a path in G from i to j.
- **shortest path**: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j of length at **most** k.
- **longest path**: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j without repeating vertices of length at **least** k.

Decision problems

- But in some computational problems, the output is more complex: a natural number, a list of elements, ...
- By focusing on decision problems only, are we losing anything?

Decision problems

If we want to prove problems hard, focusing on decision problems is OK

Decision problems

If we want to prove problems hard, focusing on decision problems is OK

1. Non-decision problems admit variants that are decision problems.
   - Imagine that, given a graph $G = (V, E)$ and two vertices $i, j \in V$, we want to find out the distance between i and j.
   - Now recall the **shortest path** problem: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j of length at most k.

Decision problems

If we want to prove problems hard, focusing on decision problems is OK

1. Non-decision problems admit variants that are decision problems.
   - Imagine that, given a graph $G = (V, E)$ and two vertices $i, j \in V$, we want to find out the distance between i and j.
   - Now recall the **shortest path** problem: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j of length at most k.
2. If we solve the original problem, the decision one can be solved too
   - If we can compute distances, we can solve **shortest path**:
     the distance between i and j is $\leq k$
     if and only if
     there is a path in G between i and j of length at most k.

Decision problems

If we want to prove problems hard, focusing on decision problems is OK

1. Non-decision problems admit variants that are decision problems.
   - Imagine that, given a graph $G = (V, E)$ and two vertices $i, j \in V$, we want to find out the distance between i and j.
   - Now recall the **shortest path** problem: given a graph $G = (V, E)$, two vertices $i, j \in V$ and a natural number k, to determine whether there is a path in G between i and j of length at most k.
2. If we solve the original problem, the decision one can be solved too
   - If we can compute distances, we can solve **shortest path**:
the distance between i and j is $\leq k$
if and only if
there is a path in G between i and j of length at most k.
3. If the original problem is easy, so is the decision problem.
Thus if the decision problem is hard, so is the original problem.

Decision problems

Given a decision problem, we define:

- positive inputs: the ones for which the answer is YES
- negative inputs: the ones for which the answer is NO

Decision problems

Given a decision problem, we define:

- positive inputs: the ones for which the answer is YES
- negative inputs: the ones for which the answer is NO

Sometimes we'll view a decision problem as a set: the set of its positive inputs

Primality

The **Primality** problem can be described informally:

Given a natural number x, to determine whether x is prime

Or formally as the set of positive inputs:

$$\{x \in \mathbb{N} \mid x \text{ is prime } \} = \{2, 3, 5, 7, \ldots\}$$

Decision problems

Given a decision problem, we define:

- positive inputs: the ones for which the answer is YES
- negative inputs: the ones for which the answer is NO

Sometimes we'll view a decision problem as a set: the set of its positive inputs

Primality

The **Primality** problem can be described informally:

Given a natural number x, to determine whether x is prime

Or formally as the set of positive inputs:

$$\{x \in \mathbb{N} \mid x \text{ is prime } \} = \{2, 3, 5, 7, \ldots\}$$

With this description, problems are easier to manipulate mathematically. E.g., given a decision problem A with inputs E, $x \in E$ is positive for A iff $x \in A$

Decision problems

We need to measure the size of the inputs of a problem.

Decision problems

We need to measure the size of the inputs of a problem.

We observe inputs must be representable in a computer. E.g., they may be:

- natural numbers
- strings
- graphs
- logic formulas
- ...

The amount of memory this representation (a.k.a. encoding) consumes indicates how big the input is.

Decision problems

We need to measure the size of the inputs of a problem.

We observe inputs must be representable in a computer. E.g., they may be:

- natural numbers
- strings
- graphs
- logic formulas
- ...

The amount of memory this representation (a.k.a. encoding) consumes indicates how big the input is.

This motivates the following definition.

Size function

Given $x \in E$, where E is the set of inputs, the size of x, written $|x|$, is the number of symbols of a (standard) representation of x in a computer.

Decision problems

Primality

In the **Primality** problem, inputs are natural numbers: $\mathbb{N}$

If natural numbers are encoded in binary,
then the size of an input is the number of digits in base 2:

$$|x| = \text{ number of digits of } x \text{ in binary} = \lfloor \log_2 x \rfloor + 1.$$

Polynomial and exponential time

- Let $\mathcal{A}$ be an algorithm with inputs E
- Given an input $x \in E$, we'll represent the running time of $\mathcal{A}$ on x as $T(x)$
- If we group inputs of the same size, we can define the worst-case cost as

$$T(n) = \max\{T(x) \mid x \in E \wedge |x| = n\}$$

  It determines limits in time that the algorithm will not exceed.

Polynomial and exponential time

- Let $\mathcal{A}$ be an algorithm with inputs E
- Given an input $x \in E$, we'll represent the running time of $\mathcal{A}$ on x as $T(x)$
- If we group inputs of the same size, we can define the worst-case cost as

$$T(n) = \max\{T(x) \mid x \in E \wedge |x| = n\}$$

  It determines limits in time that the algorithm will not exceed.
- Let us assume that $t : \mathbb{N} \to \mathbb{N}$ is a function.
  We say algorithm $\mathcal{A}$ has cost t if its worst-case cost belongs to $\mathcal{O}(t)$.

Polynomial and exponential time

- Let $\mathcal{A}$ be an algorithm with inputs E
- Given an input $x \in E$, we'll represent the running time of $\mathcal{A}$ on x as $T(x)$
- If we group inputs of the same size, we can define the worst-case cost as

$$T(n) = \max\{T(x) \mid x \in E \wedge |x| = n\}$$

  It determines limits in time that the algorithm will not exceed.
- Let us assume that $t : \mathbb{N} \to \mathbb{N}$ is a function.
  We say algorithm $\mathcal{A}$ has cost t if its worst-case cost belongs to $\mathcal{O}(t)$.

Problem decidable in time t

We say a decision problem A is decidable in time t if
there exists an algorithm $\mathcal{A} : E \to \{0, 1\}$ of cost t such that, for all $x \in E$:

$$x \in A \Rightarrow \mathcal{A}(x) = 1$$

$$x \notin A \Rightarrow \mathcal{A}(x) = 0$$

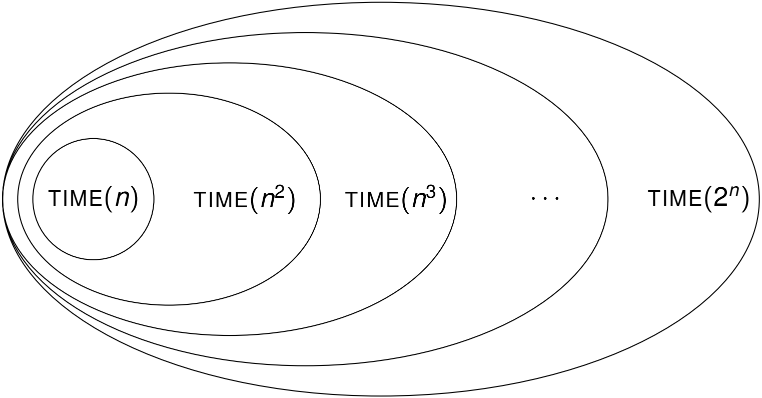
Then we say that $\mathcal{A}$ decides A in time t

Polynomial and exponential time

Class TIME(t)

Given a function $t : \mathbb{N} \to \mathbb{N}$, we will group the problems decidable in time t:

$$\text{TIME}(t) = \{A \mid A \text{ is a decision problem decidable in time } t\}.$$



Polynomial and exponential time

What is tractable with a computer and what is not?

It turns out that there is a huge difference
between having a polynomial or an exponential algorithm for a problem.

Polynomial and exponential time

Table 1 (Garey/Johnson, *Computers and Intractability*)

Comparison between polynomial and exponential functions.

cost (in μs)	10	20	30	40	50
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s	0.125 s
n^5	0.1 s	3.2 s	24.3 s	1.7 min	5.2 min
2^n	0.001 s	1.0 s	17.9 min	12.7 days	35.7 years
3^n	0.059 s	58 min	6.5 years	3855 cents.	2×10^8 cents.

Polynomial and exponential time

Table 2 (Garey/Johnson, *Computers and Intractability*)

Effect of tech improvements on polynomial and exponential algorithms.

cost	current technology	technology $\times 100$	technology $\times 1000$
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_4	$N_4 + 6.64$	$N_4 + 9.97$
3^n	N_5	$N_5 + 4.19$	$N_5 + 6.29$

Polynomial and exponential time

Class P

We define the class P as the union of all polynomial classes:

$$\mathrm{P} = \bigcup_{k>0} \mathsf{TIME}(n^k).$$

That is, a problem belongs to P if it is decidable in time n^k for some k

Class EXP

We define the class EXP as the union of all exponential classes:

$$\mathrm{EXP} = \bigcup_{k>0} \mathsf{TIME}(2^{n^k}).$$

That is, a problem belongs to EXP if it is decidable in time 2^{n^k} for some k

Polynomial and exponential time

Examples

- CONNECTIVITY $\in \mathrm{P}$
- REACHABILITY $\in \mathrm{P}$
- 2-COLOR $\in \mathrm{P}$
- SHORTEST PATH $\in \mathrm{P}$
- PRIMALITY $\in \mathrm{P}$
- 3-COLOR $\in \mathrm{EXP}$ (it is not known whether it is in P)
- LONGEST PATH $\in \mathrm{EXP}$ (it is not known whether it is in P)
- GENERALIZED CHESS $\in \mathrm{EXP}$

Polynomial and exponential time

Theorem

$\mathrm{P} \subsetneq \mathrm{EXP}$.

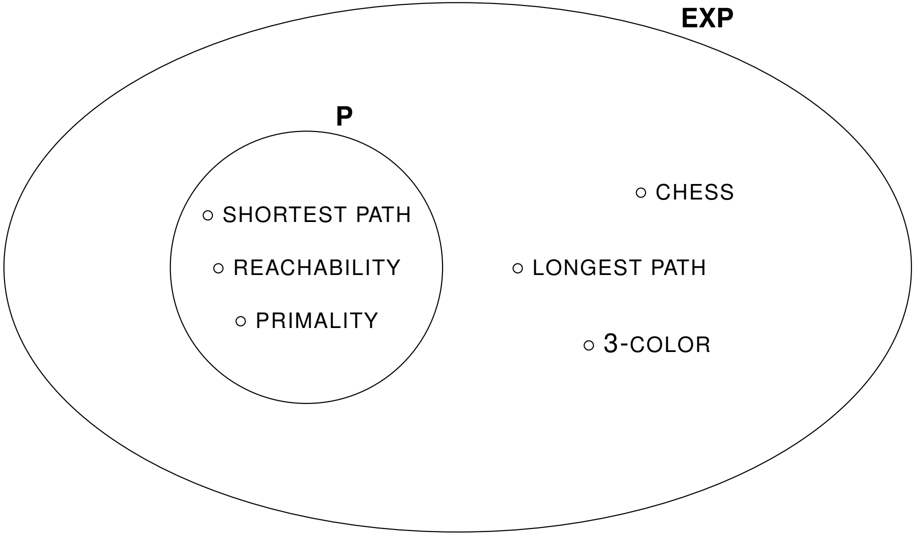
The proof of the theorem can be divided into two parts:

1. $\mathrm{P} \subseteq \mathrm{EXP}$. Obvious from the definitions:

$$\mathrm{P} = \bigcup_{k>0} \mathsf{TIME}(n^k) \subseteq \bigcup_{k>0} \mathsf{TIME}(2^{n^k}) = \mathrm{EXP}$$

2. $\mathrm{P} \neq \mathrm{EXP}$. Beyond the scope of this course.

Polynomial and exponential time



Class NP

- 3-COLOR $\in \mathrm{EXP}$ as it has an exponential-time algorithm:

3-COLORABILITY

3_COLORABLE(V, E)
    $n \leftarrow |V|$
    for each tuple $(c_1, \ldots, c_n)$ where $\forall i \leq n \;\; c_i \in \{0, 1, 2\}$
        if VALID$(V, E, (c_1, \ldots, c_n))$ **then**
            return 1
    return 0

VALID$(V, E, (c_1, \ldots, c_n))$
    for each $(i, j) \in E$
        if $c_i = c_j$ **then**
            return 0
    return 1

3-COLORABILITY

VALID$(V, E, (c_1, \ldots, c_n))$
  for each $(i, j) \in E$
    if $c_i = c_j$ **then**
      return 0
  return 1

- No known polynomial-time algo. for finding 3-colorings if there are any
- However, if we are given an assignment of colors,
  function VALID verifies it is a valid 3-coloring in polynomial time,
  and if successful this proves that the graph is 3-colorable
- 3-colorings act as certificates that the input graph is a positive instance
- Assignments of colors are small: the size is at most n
- This situation is general and motivates the following definition

Class NP

Decidability in nondeterministic polynomial time

A decision problem A defined over inputs E is decidable in nondeterministic polynomial time if there exist

- a set E'
- and a polynomial algorithm $\mathcal{V} : E \times E' \to \{0, 1\}$ (called verifier)

such that for all $x \in E$, we have

$$x \in A \iff \mathcal{V}(x, y) = 1 \text{ for some } y \in E'$$

If $x \in A$, the y s.t. $\mathcal{V}(x, y) = 1$ are called witnesses, certificates or proofs.

Class NP

Decidability in nondeterministic polynomial time

A decision problem A defined over inputs E is decidable in nondeterministic polynomial time if there exist

- a set E'
- and a polynomial algorithm $\mathcal{V} : E \times E' \to \{0, 1\}$ (called verifier)
- and a polynomial $p(n)$

such that for all $x \in E$, we have

$$x \in A \iff \mathcal{V}(x, y) = 1 \text{ for some } y \in E' \text{ such that } |y| \leq p(|x|)$$

If $x \in A$, the y s.t. $\mathcal{V}(x, y) = 1$ are called witnesses, certificates or proofs.

Class NP

To determine that problem A is decidable in nondeterministic polynomial time we have to prove that:

1. positive inputs have polynomial-size witnesses
   (first, we have to tell which are the witnesses)
2. candidate witnesses can be verified in polynomial time
   (first, we have to design the verifier)

Class NP

3-COLORABILITY

Let us consider the problem

$$3\text{-COLOR} = \{\ G \mid G \text{ is 3-colorable}\ \}$$

1. The witnesses for $G = (V, E)$ are all 3-colorings C of G of the form $C = (c_1, c_2, \ldots, c_n)$, where $n = |V|$ and $c_i \in \{0, 1, 2\}$ for all $i \leq n$.
2. The polynomial (with reasonable encodings of G and C) can be $p(n) = n$
3. The verifier is function VALID:

$\mathcal{V}(G, C)$
    for each $(i, j) \in E$
        if $c_i = c_j$ **then**
            return 0
    return 1

3-COLOR is decidable in nondeterministic polynomial time because

$$G \in 3\text{-COLOR} \Leftrightarrow \mathcal{V}(G, C)) = 1 \text{ for some } C \text{ s.t. } |C| \leq p(|G|).$$

Class NP

COMPOSITE

Let us consider the problem

$$\textsc{Composite} = \{x \mid \exists y \quad 1 < y < x \ \text{ and } y \text{ divides } x\}$$

1. The witnesses for x are all y that divide x such that $y \neq 1, x$.
2. The polynomial is $p(n) = n$
3. The verifier is

$\mathcal{V}(x, y)$
    if $(1 < y < x)$ **and** (y divides x) **then**
        return 1
    else
        return 0

COMPOSITE is decidable in nondeterministic polynomial time because

$$x \in \textsc{Composite} \Leftrightarrow \mathcal{V}(x, y) = 1 \text{ for some } y \text{ s.t. } |y| \leq p(|x|).$$

Class NP

We group problems decidable in nondeterministic polynomial time in class NP

Class NP

We define the class NP (from *Nondeterministic Polynomial time*) as:

$$\mathrm{NP} = \{A \mid A \text{ is decidable in nondeterministic polynomial time}\}.$$

How does NP compare to P and EXP?

Class NP

Main difference between P and NP:

- witnesses to problems in P can be found in polynomial time.
- witnesses to problems in NP can be verified in polynomial time.

2 and 3-colorability

1. 2-COLOR $= \{\ G \mid G$ is 2-colorable $\}$ belongs to P
2. 3-COLOR $= \{\ G \mid G$ is 3-colorable $\}$ belongs to NP

Class NP

Theorem

$\mathrm{P} \subseteq \mathrm{NP}$.

Proof

For all $A \in \mathrm{P}$, we can create verifiers $\mathcal{V}$ such that

$$\mathcal{V}(x, y) = 1 \Leftrightarrow x \in A$$

independently of y.

Hence, $A \in \mathrm{NP}$.

Class NP

Theorem

$\mathrm{NP} \subseteq \mathrm{EXP}$.

Proof

Let $A \in \mathrm{NP}$. Hence, there is a set E', a polynomial $p(n)$ and a verifier $\mathcal{V}$ s.t.

$$x \in A \iff \mathcal{V}(x, y) = 1 \text{ for some } y \in E' \text{ such that } |y| \leq p(|x|)$$

The following algorithm for A looks for a witness by brute force:

input x
for all y such that $|y| \leq p(|x|)$
  if $\mathcal{V}(x, y) = 1$ **then**
    return 1
return 0

It can be seen that the previous algorithm is exponential and decides A.

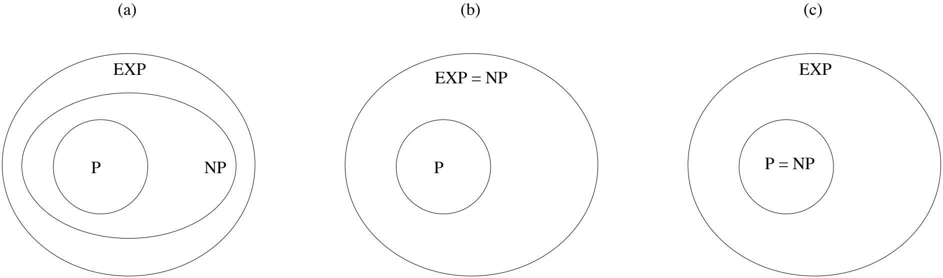
Hence, $A \in \mathrm{EXP}$.

Class NP

- It is not known whether $\mathrm{P} = \mathrm{NP}$.

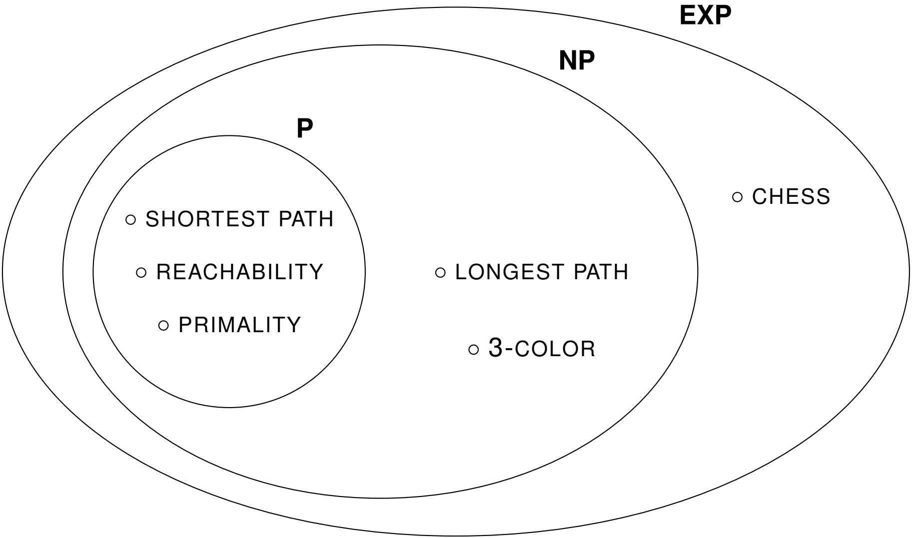
  It is one of the most important open problems in Computer Science

- We do know that either $\mathrm{P} \neq \mathrm{NP}$ or $\mathrm{NP} \neq \mathrm{EXP}$ (since we know that $\mathrm{P} \neq \mathrm{EXP}$).
- Hence, there are 3 possibilities:



It is considered that (a) is the most likely.

Class NP


EXP
NP
P
◦ SHORTEST PATH
◦ REACHABILITY
◦ PRIMALITY
◦ LONGEST PATH
◦ 3-COLOR
◦ CHESS

Motivation

- In a Sudoku puzzle, one has to complete a 9×9 grid with digits so that
  - each row,
  - each column, and
  - each of the nine 3×3 squares resulting from partitioning the grid in consecutive groups of 3 rows and 3 columns

  contains all digits from 1 to 9

- In a Sudoku puzzle, one has to complete a 9×9 grid with digits so that
  - each row,
  - each column, and
  - each of the nine 3×3 squares resulting from partitioning the grid in consecutive groups of 3 rows and 3 columns

  contains all digits from 1 to 9

- Problem SUDOKU:
  given a partially filled grid, does the puzzle have a solution?

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Motivation

- A 0-1 linear program is a constraint problem in which:
  - We have 0-1 variables x_1, ..., x_n, i.e., $x_i \in \{0, 1\}$
  - The variables are related by linear equalities and inequalities $(\leq, \geq)$

Motivation

- A 0-1 linear program is a constraint problem in which:
  - We have 0-1 variables x_1, ..., x_n, i.e., $x_i \in \{0, 1\}$
  - The variables are related by linear equalities and inequalities $(\leq, \geq)$
- For example:

$$
\begin{aligned}
x_1 + x_3 &= 1 \\
x_1 + x_2 &\leq 1 \\
x_3 + x_4 &\leq 1
\end{aligned}
$$

Motivation

- A 0-1 linear program is a constraint problem in which:
  - We have 0-1 variables x_1, ..., x_n, i.e., $x_i \in \{0, 1\}$
  - The variables are related by linear equalities and inequalities $(\leq, \geq)$
- For example:

$$
\begin{aligned}
x_1 + x_3 &= 1 \\
x_1 + x_2 &\leq 1 \\
x_3 + x_4 &\leq 1
\end{aligned}
$$

- Problem 0-1 LP:
  given a 0-1 linear program, does it have a solution?

Motivation

- Imagine we want to solve a sudoku but do not have a program for that.

  Instead, we have a program for solving problem 0-1 LP.

  Can we use that to solve problem SUDOKU?

Motivation

- Imagine we want to solve a sudoku but do not have a program for that.

  Instead, we have a program for solving problem 0-1 LP.

  Can we use that to solve problem SUDOKU?

- Let us formulate the problem of filling a Sudoku as a 0-1 linear program

Motivation

- Imagine we want to solve a sudoku but do not have a program for that.

  Instead, we have a program for solving problem 0-1 LP.

  Can we use that to solve problem SUDOKU?

- Let us formulate the problem of filling a Sudoku as a 0-1 linear program

- Let us assume that we are given a sudoku to be solved

- Let variable x_{ijk} mean "at cell of row i, column j the value is k" $(1 \leq i, j, k \leq 9)$

- If a solution assigns x_{ijk} to 1, then at row i, column j of the grid put digit k

Motivation

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

$$
\begin{array}{llllll}
x_{111} = 0 & x_{112} = 0 & x_{113} = 0 & x_{114} = 0 & x_{115} = 1 & \ldots \\
x_{121} = 0 & x_{122} = 0 & x_{123} = 1 & x_{114} = 0 & x_{115} = 0 & \ldots \\
x_{131} = 0 & x_{132} = 0 & x_{133} = 0 & x_{134} = 1 & x_{115} = 0 & \ldots \\
\ldots
\end{array}
$$

Motivation

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

$$
\begin{array}{llllll}
x_{111} = 0 & x_{112} = 0 & x_{113} = 0 & x_{114} = 0 & x_{115} = 1 & \ldots \\
x_{121} = 0 & x_{122} = 0 & x_{123} = 1 & x_{114} = 0 & x_{115} = 0 & \ldots \\
x_{131} = 0 & x_{132} = 0 & x_{133} = 0 & x_{134} = 1 & x_{115} = 0 & \ldots \\
\ldots & & & & &
\end{array}
$$

- Now let us express the constraints that these variables must satisfy

Motivation

- At each cell there is exactly one value:

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row:

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column:

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column: for all $1 \leq j, k \leq 9$,

$$\sum_{i=1}^{9} x_{ijk} = 1$$

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column: for all $1 \leq j, k \leq 9$,

$$\sum_{i=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each square:

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column: for all $1 \leq j, k \leq 9$,

$$\sum_{i=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each square: for all sq. S, $1 \leq k \leq 9$,

$$\sum_{(i,j)\in S}^{9} x_{ijk} = 1$$

Motivation

- At each cell there is exactly one value: for all $1 \leq i, j \leq 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \leq i, k \leq 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column: for all $1 \leq j, k \leq 9$,

$$\sum_{i=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each square: for all sq. S, $1 \leq k \leq 9$,

$$\sum_{(i,j) \in S}^{9} x_{ijk} = 1$$

- Fixed cells are respected:

Motivation

- At each cell there is exactly one value: for all $1 \le i, j \le 9$,

$$\sum_{k=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each row: for all $1 \le i, k \le 9$,

$$\sum_{j=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each column: for all $1 \le j, k \le 9$,

$$\sum_{i=1}^{9} x_{ijk} = 1$$

- Each value occurs exactly once in each square: for all sq. S, $1 \le k \le 9$,

$$\sum_{(i,j)\in S}^{9} x_{ijk} = 1$$

- Fixed cells are respected: for all triplets of fixed cells (i, j, k), $x_{ijk} = 1$

Motivation

- Let's call $\mathcal{F}$ the algorithm that, given a sudoku (an input of SUDOKU), produces the previous 0-1 linear program (an input of 0-1 LP)

- We observe that:
  - If x can be filled, then $\mathcal{F}(x)$ has a solution
  - If $\mathcal{F}(x)$ has a solution, then x can be filled

- So $\mathcal{F}$ allows us to solve SUDOKU using a program for 0-1 LP!

Concept of reduction

- In general, if we have two problems A and B,
  under which conditions we can use an algorithm for B to solve A?

Concept of reduction

- In general, if we have two problems A and B,
  under which conditions we can use an algorithm for B to solve A?
- Let $\mathcal{F}$ be an algorithm that transforms inputs of A into inputs of B s.t.

  (1) $x \in A \quad \Rightarrow \mathcal{F}(x) \in B$ (or equivalently $\mathcal{F}(x) \notin B \Rightarrow x \notin A$)

  (2) $\mathcal{F}(x) \in B \Rightarrow x \in A$

Concept of reduction

- In general, if we have two problems A and B,
  under which conditions we can use an algorithm for B to solve A?
- Let $\mathcal{F}$ be an algorithm that transforms inputs of A into inputs of B s.t.

  (1) $x \in A \quad \Rightarrow \mathcal{F}(x) \in B$ (or equivalently $\mathcal{F}(x) \notin B \Rightarrow x \notin A$)

  (2) $\mathcal{F}(x) \in B \Rightarrow x \in A$
- Imagine we have an algorithm $\mathcal{G}$ for solving B

Concept of reduction

- In general, if we have two problems A and B,
  under which conditions we can use an algorithm for B to solve A?
- Let $\mathcal{F}$ be an algorithm that transforms inputs of A into inputs of B s.t.

  (1) $x \in A \Rightarrow \mathcal{F}(x) \in B$ (or equivalently $\mathcal{F}(x) \notin B \Rightarrow x \notin A$)
  (2) $\mathcal{F}(x) \in B \Rightarrow x \in A$
- Imagine we have an algorithm $\mathcal{G}$ for solving B
- Now, given input x for A, we can run $\mathcal{G}$ on $\mathcal{F}(x)$.

  And then we know that:
  - If $\mathcal{F}(x) \in B$, then $x \in A$ (by (2)): so answer "yes"
  - If $\mathcal{F}(x) \notin B$, then $x \notin A$ (by (1)): so answer "no"
- So composing $\mathcal{G}$ with $\mathcal{F}$ (e.g. with Linux pipe |) we get an algorithm for A

Concept of reduction

Reductions

Let A and B be two decision problems with input sets E and E', respectively.

We say *A reduces to B in polynomial time* if
there exists a polynomial-time algorithm $\mathcal{F} : E \to E'$ such that

$$x \in A \iff \mathcal{F}(x) \in B$$

In this case we write $A \leq^p B$ (via $\mathcal{F}$),

and we say that $\mathcal{F}$ is a polynomial reduction from A to B.

Examples and properties

PARTITION reduces to SUBSET SUM

Let us consider the following two problems:

PARTITION
Given natural numbers $x_1, x_2, \ldots, x_n$,
determine whether they can be divided into two groups with the same sum.

SUBSET SUM
Given natural numbers $y_1, y_2, \ldots, y_m$ and a capacity $C \in \mathbb{N}$,
determine whether there is a selection of the y_i's that sums exactly C.

Formally:

$$\text{PARTITION} = \{(x_1, \ldots, x_n) \mid \exists S \subseteq \{1, \ldots, n\} \quad \sum_{i \in S} x_i = \sum_{i \notin S} x_i\}$$

$$\text{SUBSET SUM} = \{(y_1, \ldots, y_m, C) \mid \exists T \subseteq \{1, \ldots, m\} \quad \sum_{i \in T} y_i = C\}$$

Examples and properties

PARTITION reduces to SUBSET SUM

The algorithm

$\mathcal{F}(x_1, \ldots, x_n)$
    $S \leftarrow \sum_{i=1}^{n} x_i$
    if S is even **then**
        return $(x_1, \ldots, x_n, S/2)$
    else
        return $(x_1, \ldots, x_n, S+1)$ /* any negative input for SUBSET SUM will do */

is a polynomial reduction from PARTITION to SUBSET SUM:

$$(x_1, \ldots, x_n) \in \text{PARTITION} \Leftrightarrow \mathcal{F}(x_1, \ldots, x_n) \in \text{SUBSET SUM}.$$

Examples and properties

Properties: transitivity

For all A, B, C, if $A \leq^p B$ and $B \leq^p C$, then $A \leq^p C$.

Proof

If

- $A \leq^p B$ via $\mathcal{F}$ and
- $B \leq^p C$ via $\mathcal{G}$,

then the composition $\mathcal{G} \circ \mathcal{F}$ proves that $A \leq^p C$.

Recall that by definition $\mathcal{G} \circ \mathcal{F}(x) = \mathcal{G}(\mathcal{F}(x))$.

Examples and properties

Properties: closure of P under reductions

For all A, B, if $A \leq^p B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

Proof

If

- $\mathcal{B}$ is a polynomial algorithm for B and
- $\mathcal{F}$ is a polynomial algorithm that proves $A \leq^p B$,

then the composition $\mathcal{F} \circ \mathcal{B}$ is a polynomial algorithm for A.

Exercise

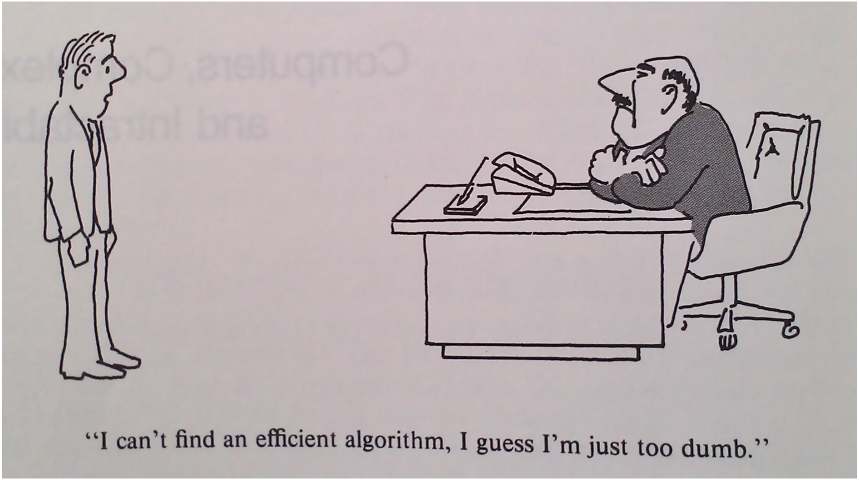
Let us consider the following collection of problems:

> k**-Colorability** (k-COLOR)
> Given an undirected graph G, determine whether vertices in G can be colored with at most k colors, so that each pair of adjacent vertices get different colors.

Prove that, for all k, it holds that:

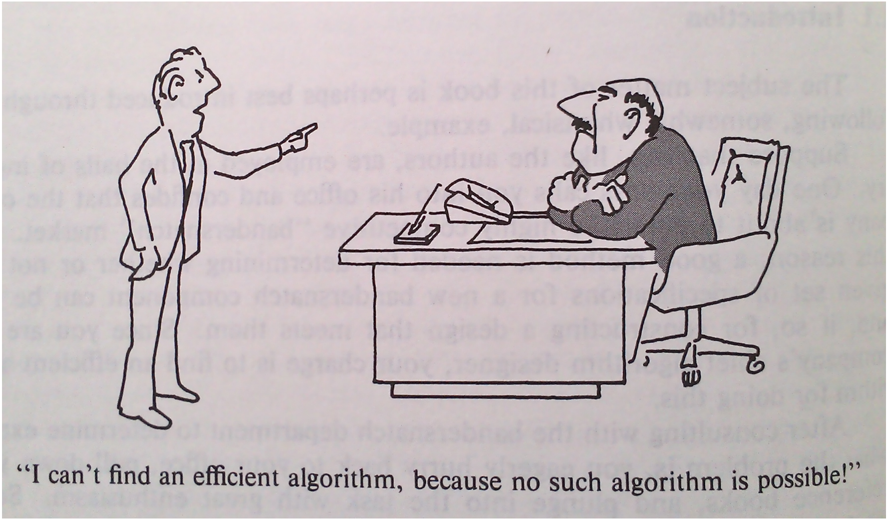
$$k\text{-COLOR} \leq^p (k+1)\text{-COLOR}.$$

NP-completeness theory



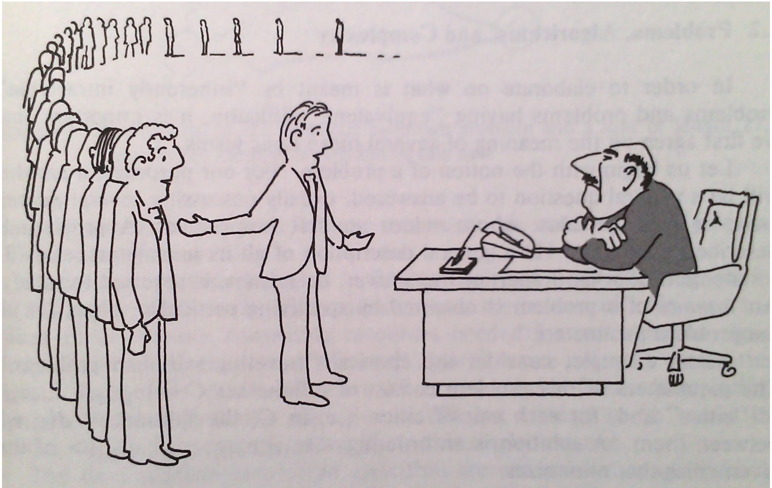
Garey & Johnson, *Computers and Intractability*

NP-completeness theory



Garey & Johnson, *Computers and Intractability*

NP-completeness theory

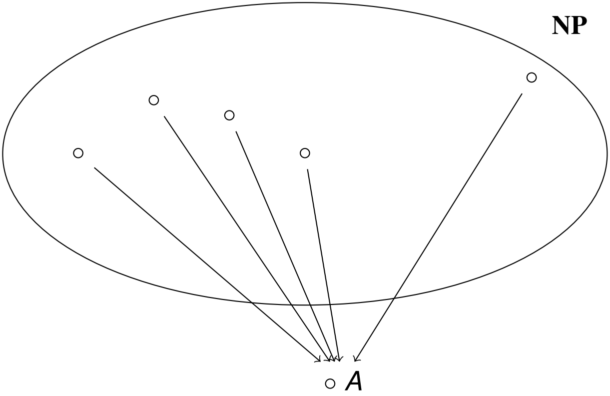


“I can’t find an efficient algorithm, but neither can all these famous people.”

Garey & Johnson, *Computers and Intractability*

NP-hard

A problem A is NP-hard if for all problem $B \in \mathrm{NP}$ we have that $B \leq^p A$.



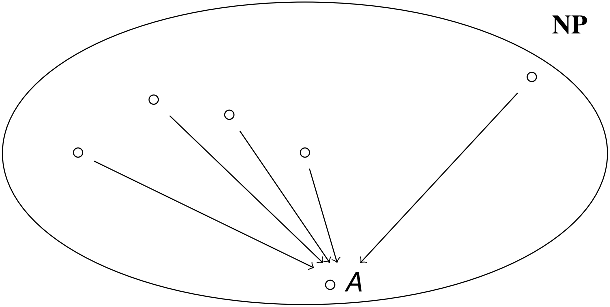
NP-completeness theory

NP-complete

A problem A is NP-complete if it is NP-hard and $A \in \mathrm{NP}$.

Class NPC

We define the class NPC as the set of all NP-complete problems



Proposition

Let A be an NP-complete problem. Then, $\mathrm{P} = \mathrm{NP}$ if and only if $A \in \mathrm{P}$.

$\boxed{\Rightarrow}$ Since A is NP-complete, $A \in \mathrm{NP}$ and hence $A \in \mathrm{P}$.

NP-completeness theory

Proposition

Let A be an NP-complete problem. Then, $\mathrm{P} = \mathrm{NP}$ if and only if $A \in \mathrm{P}$.

$\boxed{\Rightarrow}$ Since A is NP-complete, $A \in \mathrm{NP}$ and hence $A \in \mathrm{P}$.

$\boxed{\Leftarrow}$ Let $A \in \mathrm{P}$.

1. Since A is NP-complete, we know that for all $B \in \mathrm{NP}$, $B \le^p A$.
2. Due to the closure of P under reductions,
   we know that for all B such that $B \le^p A$ we have $B \in \mathrm{P}$.

Using 1 and 2, $\mathrm{NP} \subseteq \mathrm{P}$ and hence $\mathrm{P} = \mathrm{NP}$.

But... do NP-complete problems really exist?

Boolean formulas

- A Boolean formula is a formula over Boolean variables with the connectives: $\vee$ (disjunction), $\wedge$ (conjunction) and $\neg$ (negation).
- Quantifiers ($\exists, \forall$) are **not** allowed
- For example,

$$F(x, y, z) = (x \vee y \vee \neg z) \ \wedge \ \neg(x \wedge y \wedge z)$$

is a Boolean formula

NP-completeness theory

Boolean formulas

- A Boolean formula is a formula over Boolean variables with the connectives: $\vee$ (disjunction), $\wedge$ (conjunction) and $\neg$ (negation).
- Quantifiers ($\exists, \forall$) are **not** allowed
- For example,

$$F(x, y, z) = (x \vee y \vee \neg z) \ \wedge \ \neg(x \wedge y \wedge z)$$

  is a Boolean formula

Conjunctive Normal Form (CNF)

- A literal is a variable or its negation (x, $\neg x$)
- A clause is a disjunction of literals ($x \vee \neg y \vee z$)
- A Boolean formula is in CNF if it is a conjunction of clauses
- For example, $F(x, y, z) = (x \vee \neg y \vee z) \ \wedge \ (\neg x \vee \neg z)$

Satisfiability

A Boolean formula is satisfiable if
there is an assignment from vars to $\{0, 1\}$ that evaluates the formula to true

For example,

$$F(x, y, z) = (x \vee \neg y \vee z) \ \wedge \ (\neg x \vee \neg z)$$

is satisfiable (since with $x = 1$, $y = 0$, $z = 0$ we have $F(1, 0, 0) = 1$).

We define

$$\text{SAT} \ = \{ \ F \mid F \text{ is a satisfiable Boolean formula } \}$$

$$\text{CNF-SAT} \ = \{ \ F \mid F \text{ is a satisfiable Boolean formula in CNF } \}$$

Satisfiability

A Boolean formula is satisfiable if
there is an assignment from vars to $\{0,1\}$ that evaluates the formula to true

For example,

$$F(x,y,z) = (x \vee \neg y \vee z) \ \wedge \ (\neg x \vee \neg z)$$

is satisfiable (since with $x = 1$, $y = 0$, $z = 0$ we have $F(1,0,0) = 1$).

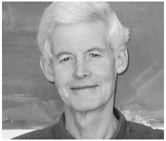
We define

$$\text{SAT} \ = \{ \ F \mid F \text{ is a satisfiable Boolean formula } \}$$

$$\text{CNF-SAT} \ = \{ \ F \mid F \text{ is a satisfiable Boolean formula in CNF } \}$$

Cook-Levin Theorem (1971)

SAT and CNF-SAT are NP-complete.



Let us sketch the proof that CNF-SAT is NP-complete. We need:

1. CNF-SAT $\in \mathrm{NP}$
2. CNF-SAT is NP-hard

NP-completeness theory

Let us sketch the proof that CNF-SAT is NP-complete. We need:

1. CNF-SAT $\in \mathrm{NP}$
2. CNF-SAT is NP-hard

(1) CNF-SAT $\in \mathrm{NP}$

- Witness candidates are assignments of Boolean variables to $\{0, 1\}$. Witnesses are those that satisfy F.
- In any reasonable encoding of a formula F with n variables, $n \leq |F|$. Since a witness candidate α has n bits, $|\alpha| = n \leq |F|$.
- Hence, choosing $p(n) = n$, we have that $|\alpha| \leq p(|F|)$.

NP-completeness theory

Let us sketch the proof that CNF-SAT is NP-complete. We need:

1. CNF-SAT $\in \mathrm{NP}$
2. CNF-SAT is NP-hard

(1) CNF-SAT $\in \mathrm{NP}$

- Witness candidates are assignments of Boolean variables to $\{0, 1\}$. Witnesses are those that satisfy F.
- In any reasonable encoding of a formula F with n variables, $n \leq |F|$. Since a witness candidate α has n bits, $|\alpha| = n \leq |F|$.
- Hence, choosing $p(n) = n$, we have that $|\alpha| \leq p(|F|)$.
- We can verify whether an assignment α satisfies F in polynomial time:
  - replace the variables by the values given by α
  - evaluate the connectives bottom up

Example

If we consider the CNF

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (x \vee \neg z)$$

and the assignment $\alpha = (1, 0, 0)$ (that is, $x = 1$, $y = 0$, $z = 0$), the verifier would evaluate:

- $F(\alpha) = (1 \vee \neg 0 \vee 0) \wedge (1 \vee \neg 0)$ (replace values)
- $F(\alpha) = (1 \vee 1 \vee 0) \wedge (1 \vee 1)$ (negations)
- $F(\alpha) = (1) \wedge (1)$ (disjunctions)
- $F(\alpha) = 1$ (conjunctions)

Lemma

Given an algorithm $\mathcal{A} : E \to \{0,1\}$ with polynomial cost, we can find in polynomial time a Boolean formula in CNF $F_{\mathcal{A}}$ such that for all $x \in E$:

$$F_{\mathcal{A}}(x) \Leftrightarrow \mathcal{A}(x) = 1$$

NP-completeness theory

Lemma

Given an algorithm $\mathcal{A} : E \to \{0, 1\}$ with polynomial cost, we can find in polynomial time a Boolean formula in CNF $F_{\mathcal{A}}$ such that for all $x \in E$:

$$F_{\mathcal{A}}(x) \Leftrightarrow \mathcal{A}(x) = 1$$

(2) CNF-SAT is NP-hard.

Let $A \in \mathrm{NP}$. Then, there is a polynomial q and a verifier $\mathcal{V}$ s.t. for all x:

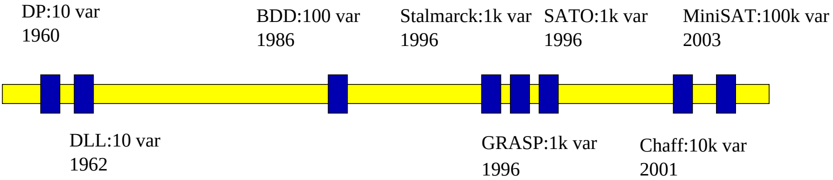
$$x \in A \Leftrightarrow \exists y \quad |y| \leq q(|x|) \wedge \mathcal{V}(x, y) = 1.$$

Let $\mathcal{V}_x(y)$ be an algorithm that checks $|y| \leq q(|x|)$ and $\mathcal{V}(x, y) = 1$. Then,

$$x \in A \Leftrightarrow \exists y \;\; \mathcal{V}_x(y) = 1 \;\Leftrightarrow\; \exists y \;\; F_{\mathcal{V}_x}(y) \;\Leftrightarrow\; F_{\mathcal{V}_x} \in \text{CNF-SAT}.$$

Hence, $A \leq^p$ CNF-SAT.

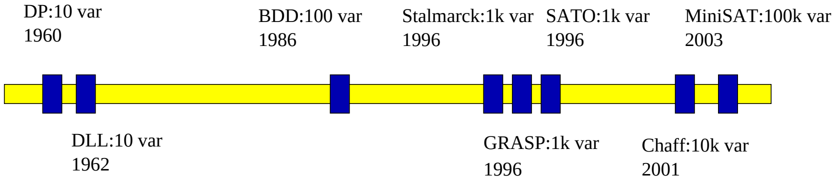
- For years, NP-completeness of CNF-SAT had only theoretical interest
- But since year $\approx$ 2000, tools for solving CNF-SAT (called SAT solvers) have improved dramatically



Modern SAT solvers handle formulas with millions of variables/clauses.

NP-completeness theory

- For years, NP-completeness of CNF-SAT had only theoretical interest
- But since year $\approx$ 2000, tools for solving CNF-SAT (called SAT solvers) have improved dramatically

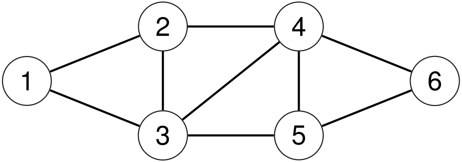


Modern SAT solvers handle formulas with millions of variables/clauses.

- So an effective alternative for solving problems in NP has emerged: to reduce them to CNF-SAT (and then use a SAT solver off-the-shelf)

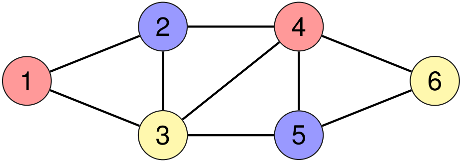
- For example, let us reduce 3-COLOR to CNF-SAT

- Let $G = (V, E)$ be a graph (the input of 3-COLOR).
  E.g.,



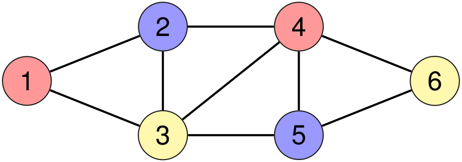
- We will produce a formula F_G in CNF (in polynomial time in $|G|$) such that
  - If G is 3-colorable, then F_G is satisfiable
  - If F_G is satisfiable, then G is 3-colorable

- Let variable x_{vk} mean "vertex v is painted with color k" $(v \in V, 0 \leq k \leq 2)$



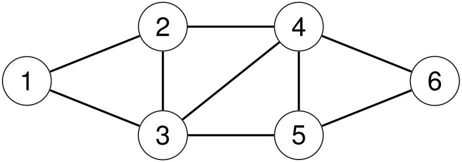
$v_{10} = 1$	$v_{20} = 0$	$v_{30} = 0$	$v_{40} = 1$	$v_{50} = 0$	$v_{60} = 0$
$v_{11} = 0$	$v_{21} = 1$	$v_{31} = 0$	$v_{41} = 0$	$v_{51} = 1$	$v_{61} = 0$
$v_{12} = 0$	$v_{22} = 0$	$v_{32} = 1$	$v_{42} = 0$	$v_{52} = 0$	$v_{62} = 1$

- Let variable x_{vk} mean "vertex v is painted with color k" $(v \in V, 0 \le k \le 2)$

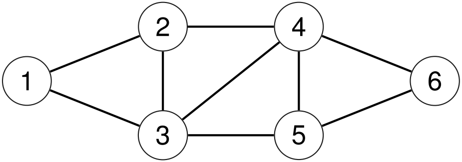


$v_{10} = 1$	$v_{20} = 0$	$v_{30} = 0$	$v_{40} = 1$	$v_{50} = 0$	$v_{60} = 0$
$v_{11} = 0$	$v_{21} = 1$	$v_{31} = 0$	$v_{41} = 0$	$v_{51} = 1$	$v_{61} = 0$
$v_{12} = 0$	$v_{22} = 0$	$v_{32} = 1$	$v_{42} = 0$	$v_{52} = 0$	$v_{62} = 1$

- Now let us express the constraints that these variables must satisfy

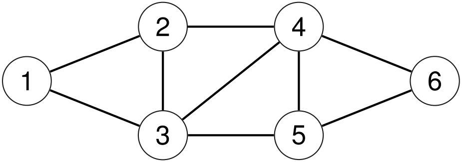


- Each vertex is painted with at least one color:



- Each vertex is painted with at least one color:
  for all $v \in V$

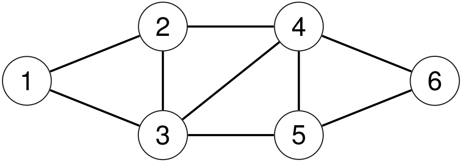
$$x_{v0} \vee x_{v1} \vee x_{v2}$$



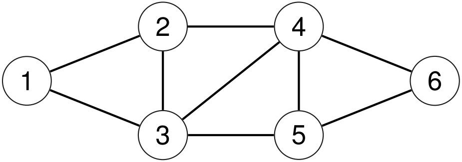
- Each vertex is painted with at least one color:
  for all $v \in V$

$$x_{v0} \vee x_{v1} \vee x_{v2}$$

$$\begin{array}{ll} v = 1: & x_{10} \vee x_{11} \vee x_{12} \\ v = 2: & x_{20} \vee x_{21} \vee x_{22} \\ v = 3: & x_{30} \vee x_{31} \vee x_{32} \\ v = 4: & x_{40} \vee x_{41} \vee x_{42} \\ v = 5: & x_{50} \vee x_{51} \vee x_{52} \\ v = 6: & x_{60} \vee x_{61} \vee x_{62} \end{array}$$

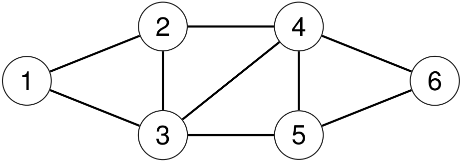


- Each vertex is painted with at most one color:



- Each vertex is painted with at most one color: for all $v \in V$ and $0 \leq k < k' \leq 2$

$$\neg x_{vk} \vee \neg x_{vk'}$$

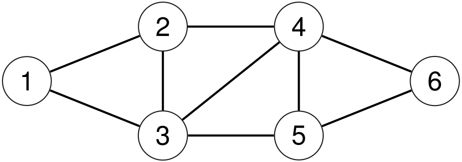


- Each vertex is painted with at most one color:
  for all $v \in V$ and $0 \leq k < k' \leq 2$

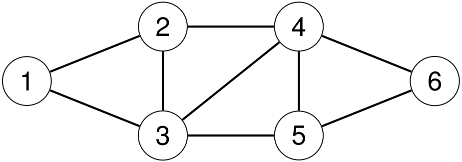
$$\neg x_{vk} \vee \neg x_{vk'}$$

$v = 1$:	$\neg x_{10} \vee \neg x_{11}$	$\neg x_{10} \vee \neg x_{12}$	$\neg x_{11} \vee \neg x_{12}$
$v = 2$:	$\neg x_{20} \vee \neg x_{21}$	$\neg x_{20} \vee \neg x_{22}$	$\neg x_{21} \vee \neg x_{22}$
$v = 3$:	$\neg x_{30} \vee \neg x_{31}$	$\neg x_{30} \vee \neg x_{32}$	$\neg x_{31} \vee \neg x_{32}$
$v = 4$:	$\neg x_{40} \vee \neg x_{41}$	$\neg x_{40} \vee \neg x_{42}$	$\neg x_{41} \vee \neg x_{42}$
$v = 5$:	$\neg x_{50} \vee \neg x_{51}$	$\neg x_{50} \vee \neg x_{52}$	$\neg x_{51} \vee \neg x_{52}$
$v = 6$:	$\neg x_{60} \vee \neg x_{61}$	$\neg x_{60} \vee \neg x_{62}$	$\neg x_{61} \vee \neg x_{62}$

NP-completeness theory

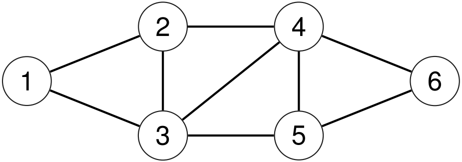


- Adjacent vertices are painted with different colors:



- Adjacent vertices are painted with different colors: for all $e = (u, v) \in E$ and $0 \leq k \leq 2$

$$\neg x_{uk} \vee \neg x_{vk}$$



- Adjacent vertices are painted with different colors:
  for all $e = (u, v) \in E$ and $0 \leq k \leq 2$

$$\neg x_{uk} \vee \neg x_{vk}$$

$$
\begin{array}{llll}
e = (1,2): & \neg x_{10} \vee \neg x_{20} & \neg x_{11} \vee \neg x_{21} & \neg x_{12} \vee \neg x_{22} \\
e = (1,3): & \neg x_{10} \vee \neg x_{30} & \neg x_{11} \vee \neg x_{31} & \neg x_{12} \vee \neg x_{32} \\
e = (2,3): & \neg x_{20} \vee \neg x_{30} & \neg x_{21} \vee \neg x_{31} & \neg x_{22} \vee \neg x_{32} \\
e = (2,4): & \neg x_{20} \vee \neg x_{40} & \neg x_{21} \vee \neg x_{41} & \neg x_{22} \vee \neg x_{42} \\
e = (3,4): & \neg x_{30} \vee \neg x_{40} & \neg x_{31} \vee \neg x_{41} & \neg x_{32} \vee \neg x_{42} \\
e = (3,5): & \neg x_{30} \vee \neg x_{50} & \neg x_{31} \vee \neg x_{51} & \neg x_{32} \vee \neg x_{52} \\
e = (4,5): & \neg x_{40} \vee \neg x_{50} & \neg x_{41} \vee \neg x_{51} & \neg x_{42} \vee \neg x_{52} \\
e = (4,6): & \neg x_{40} \vee \neg x_{60} & \neg x_{41} \vee \neg x_{61} & \neg x_{42} \vee \neg x_{62} \\
e = (5,6): & \neg x_{50} \vee \neg x_{60} & \neg x_{51} \vee \neg x_{61} & \neg x_{52} \vee \neg x_{62}
\end{array}
$$

NP-complete problems

Once we have a first NP-complete problem, more can be found via reductions

NP-complete problems

Once we have a first NP-complete problem, more can be found via reductions

For example, recall that a clique is a complete graph, i.e., it contains all possible edges among its vertices

Let us consider the following problem:

$$\text{CLIQUE} = \{ (G, k) \mid G \text{ has a clique with } k \text{ vertices } \}$$

NP-complete problems

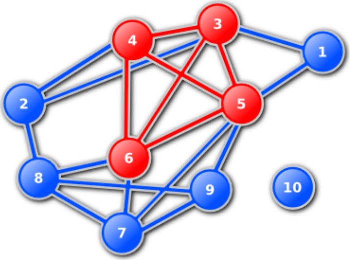
Once we have a first NP-complete problem, more can be found via reductions

For example, recall that a clique is a complete graph, i.e.,
it contains all possible edges among its vertices

Let us consider the following problem:

$$\text{CLIQUE} = \{\ (G, k) \mid G \text{ has a clique with } k \text{ vertices}\ \}$$

For instance, given graph G



we can see that

- $(G, 4) \ \in \text{CLIQUE}$
- $(G, 5) \notin \text{CLIQUE}$

NP-complete problems

Theorem

CLIQUE is NP-complete

In order to prove the NP-completeness of CLIQUE we have to see that:

1. CLIQUE $\in \mathrm{NP}$
2. CLIQUE is NP-hard

NP-complete problems

Theorem

CLIQUE is NP-complete

In order to prove the NP-completeness of CLIQUE we have to see that:

1. CLIQUE $\in \mathrm{NP}$
2. CLIQUE is NP-hard

(1) CLIQUE $\in \mathrm{NP}$

Let (G, k) be an input of CLIQUE.

- Witnesses are sets of k vertices whose induced subgraphs are complete (in the previous example, the set $C = \{3, 4, 5, 6\}$)
- The polynomial $p(n) = n$ is enough because a witness C satisfies $|C| \leq |G| \leq |(G, k)| = p(|(G, k)|)$.
- We can verify in polynomial time whether a set C of vertices is a witness: C should have k vertices and any pair of vertices of C should have an edge in G ($\leq n^2$ checks).

NP-complete problems

CLIQUE is NP-hard

We will prove that CNF-SAT $\leq^p$ CLIQUE.

Before that, assuming it is true:

- Since CNF-SAT is NP-hard, any $X \in \mathrm{NP}$ satisfies $X \leq^p$ CNF-SAT.
- By transitivity, any $X \in \mathrm{NP}$ satisfies $X \leq^p$ CLIQUE.
- Hence, CLIQUE is NP-hard.

NP-complete problems

CLIQUE is NP-hard

We will prove that CNF-SAT $\leq^p$ CLIQUE.

Before that, assuming it is true:

- Since CNF-SAT is NP-hard, any $X \in \mathrm{NP}$ satisfies $X \leq^p$ CNF-SAT.
- By transitivity, any $X \in \mathrm{NP}$ satisfies $X \leq^p$ CLIQUE.
- Hence, CLIQUE is NP-hard.

We can express this property in general:

Proposition

Let A be an NP-hard problem.

Let B be a problem such that $A \leq^p B$.

Then B is also NP-hard.

NP-complete problems

CNF-SAT $\leq^p$ CLIQUE

Let F be a Boolean formula in CNF with:

- literals $\ell_1, \ldots, \ell_n$
- clauses $C_1, \ldots, C_m$

CNF-SAT $\leq^p$ CLIQUE

Let F be a Boolean formula in CNF with:

- literals $\ell_1, \ldots, \ell_n$
- clauses $C_1, \ldots, C_m$

The reduction algorithm returns (G, m), where $G = (V, E)$ is:

- $V = \{(\ell, j) \mid \ell \text{ appears in } C_j\}$
  (a vertex represents the occurrence of a literal in a clause)

NP-complete problems

CNF-SAT $\leq^p$ CLIQUE

Let F be a Boolean formula in CNF with:

- literals $\ell_1, \ldots, \ell_n$
- clauses $C_1, \ldots, C_m$

The reduction algorithm returns (G, m), where $G = (V, E)$ is:

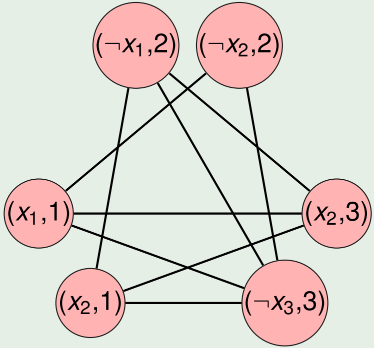
- $V = \{(\ell, j) \mid \ell \text{ appears in } C_j \}$
  (a vertex represents the occurrence of a literal in a clause)
- $E = \{ \{(\ell, i), (\ell', j)\} \mid i \neq j \wedge \neg\ell \neq \ell'\}$
  (an edge represents a pair of literals that can be simultaneously true)

Example

$F(x_1, x_2, x_3) = C_1 \wedge C_2 \wedge C_3$, where

- $C_1 = (x_1 \vee x_2)$, $C_2 = (\neg x_1 \vee \neg x_2)$, $C_3 = (x_2 \vee \neg x_3)$

The reduction returns $(G, 3)$, where G is the graph



NP-complete problems

In general, we have that

$$F \in \textsc{CNF-SAT} \Leftrightarrow (G, m) \in \textsc{CLIQUE}$$

NP-complete problems

In general, we have that

$$F \in \text{CNF-SAT} \Leftrightarrow (G, m) \in \text{CLIQUE}$$

$\boxed{\Rightarrow}$ Let α be an assignment that satisfies F.

Each clause of F contains a literal that is true in α.

As there are m clauses, we have m occurrences of literals, i.e. vertices.

Each pair of these vertices is connected with an edge:

- they belong to different clauses
- they can be simultaneously true (both are true in α)

So these vertices form a complete subgraph in G, i.e. a clique, of size m

NP-complete problems

In general, we have that

$$F \in \text{CNF-SAT} \Leftrightarrow (G, m) \in \text{CLIQUE}$$

$\boxed{\Rightarrow}$ Let α be an assignment that satisfies F.

Each clause of F contains a literal that is true in α.

As there are m clauses, we have m occurrences of literals, i.e. vertices.

Each pair of these vertices is connected with an edge:

- they belong to different clauses
- they can be simultaneously true (both are true in α)

So these vertices form a complete subgraph in G, i.e. a clique, of size m

$\boxed{\Leftarrow}$ If G has a clique with m vertices, each belongs to a different clause.

An assignment that makes the corresponding literals true will simultaneously satisfy all clauses, and so satisfy F

NP-complete problems

Definitions

Let $G = (V, E)$ be a graph.

- A set of vertices $S \subseteq V$ is an independent subset of G if there are no edges between vertices in S
- A set of vertices $S \subseteq V$ is a vertex cover of G if for any edge in E, at least one of its endpoints is in S

NP-complete problems

Exercise

Given the following problems:

- CLIQUE $= \{ (G, k) \mid G$ has a clique with k vertices $\}$
- IS $= \{ (G, k) \mid G$ has an independent subset with k vertices $\}$
- VC $= \{ (G, k) \mid G$ has a vertex cover with k vertices $\}$

prove that

1. CLIQUE $\leq^p$ IS
2. IS $\leq^p$ VC
3. VC $\leq^p$ CLIQUE

NP-complete problems

Lots of NP-complete problems have “particular cases” that are in P.

NP-complete problems

Lots of NP-complete problems have “particular cases” that are in P.

For example, in CNF-SAT,
we can fix the number of literals per clause to get a family of problems:

k-bounded satisfiability (k-SAT)

Given a Boolean formula in CNF over n variables with $\leq k$ literals per clause, to determine whether it is satisfiable.

NP-complete problems

Lots of NP-complete problems have “particular cases” that are in P.

For example, in CNF-SAT,
we can fix the number of literals per clause to get a family of problems:

***k*-bounded satisfiability** (k-SAT)

Given a Boolean formula in CNF over n variables with $\leq k$ literals per clause, to determine whether it is satisfiable.

We will see how to classify k-SAT for the different values of k.

1-bounded satisfiability (1-SAT)

Given a Boolean formula in CNF over n variables with ≤ 1 literal per clause, to determine whether it is satisfiable.

For example,

$$F(x, y, z, t) = (x) \wedge (\neg y) \wedge (z) \wedge (\neg t).$$

NP-complete problems

1-bounded satisfiability (1-SAT)

Given a Boolean formula in CNF over n variables with ≤ 1 literal per clause, to determine whether it is satisfiable.

For example,

$$F(x, y, z, t) = (x) \wedge (\neg y) \wedge (z) \wedge (\neg t).$$

1-SAT is decidable in polynomial time with the following algorithm:

input F
if F has two contradictory literals **then**
    return FALSE
else
    return TRUE

2-bounded satisfiability (2-SAT)

Given a Boolean formula in CNF over n variables with ≤ 2 literals per clause, to determine whether it is satisfiable.

For example,

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z).$$

NP-complete problems

2-bounded satisfiability (2-SAT)

Given a Boolean formula in CNF over n variables with ≤ 2 literals per clause, to determine whether it is satisfiable.

For example,

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z).$$

2-SAT is decidable in polynomial time

- transform the formula into a directed graph
- compute strongly connected components of the graph

NP-complete problems

Sketch of the algorithm

Recall the equivalences

- $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$
- $a \equiv (a \vee a) \equiv (\neg a \Rightarrow a)$
- $\neg a \equiv (\neg a \vee \neg a) \equiv (a \Rightarrow \neg a)$

NP-complete problems

Sketch of the algorithm

Recall the equivalences

- $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$
- $a \equiv (a \vee a) \equiv (\neg a \Rightarrow a)$
- $\neg a \equiv (\neg a \vee \neg a) \equiv (a \Rightarrow \neg a)$

Using them the 2-CNF Boolean formula

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z)$$

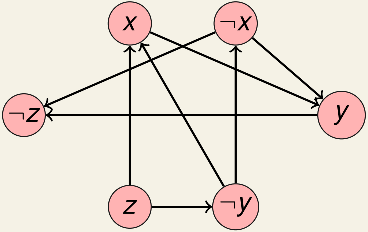
can be rewritten using implications as

$$\begin{array}{rcllll} F(x, y, z) & = & (\neg x \Rightarrow y) & \wedge (\neg x \Rightarrow \neg z) & \wedge (x \Rightarrow y) & \wedge (y \Rightarrow \neg z) \\ & & (\neg y \Rightarrow x) & \wedge (z \Rightarrow x) & \wedge (\neg y \Rightarrow \neg x) & \wedge (z \Rightarrow \neg y) \end{array}$$

The Boolean formula with implications

$$\begin{array}{rcllll} F(x,y,z) & = & (\neg x \Rightarrow y) & \wedge (\neg x \Rightarrow \neg z) & \wedge (x \Rightarrow y) & \wedge (y \Rightarrow \neg z) \\ & & (\neg y \Rightarrow x) & \wedge (z \Rightarrow x) & \wedge (\neg y \Rightarrow \neg x) & \wedge (z \Rightarrow \neg y) \end{array}$$

is transformed into a directed graph G and we apply the following lemma.



Lemma

F is unsatisfiable
iff $\exists a$ s.t. G has paths from a to $\neg a$ and from $\neg a$ to a
iff $\exists a$ s.t. a, $\neg a$ belong to the same strongly connected component of G

NP-complete problems

3-bounded satisfiability (3-SAT)

Given a Boolean formula in CNF over n variables with ≤ 3 literals per clause, to determine whether it is satisfiable.

NP-complete problems

3-bounded satisfiability (3-SAT)

Given a Boolean formula in CNF over n variables with ≤ 3 literals per clause, to determine whether it is satisfiable.

Theorem

3-SAT is NP-complete.

To prove it, we need two facts:

1. 3-SAT $\in \mathrm{NP}$ (similar to CNF-SAT)
2. 3-SAT is NP-hard: reduction CNF-SAT $\leq^p$ 3-SAT

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

NP-complete problems

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α,
  then C' can be satisfied with α and appropriate values for z_2 and z_3:

NP-complete problems

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α,
  then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α,
  then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$

NP-complete problems

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α,
  then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$
  - Else if $\alpha(a_3) = 1$, then set $\alpha(z_2) = 1, \alpha(z_3) = 0$

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α, then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$
  - Else if $\alpha(a_3) = 1$, then set $\alpha(z_2) = 1, \alpha(z_3) = 0$
- If C' is true with assignment β, some a_i will be true and C will be true with β:

NP-complete problems

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α,
  then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$
  - Else if $\alpha(a_3) = 1$, then set $\alpha(z_2) = 1, \alpha(z_3) = 0$
- If C' is true with assignment β,
  some a_i will be true and C will be true with β:
  - If $\beta(z_2) = 0$, then $\beta(a_1) = 1$ or $\beta(a_2) = 1$

NP-complete problems

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α, then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$
  - Else if $\alpha(a_3) = 1$, then set $\alpha(z_2) = 1, \alpha(z_3) = 0$
- If C' is true with assignment β, some a_i will be true and C will be true with β:
  - If $\beta(z_2) = 0$, then $\beta(a_1) = 1$ or $\beta(a_2) = 1$
  - If $\beta(z_3) = 1$, then $\beta(a_4) = 1$ or $\beta(a_5) = 1$

Example

Given a Boolean formula with a single clause $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, the reduction returns

$$C' = (a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee a_5)$$

- If C is true with assignment α, then C' can be satisfied with α and appropriate values for z_2 and z_3:
  - If $\alpha(a_1) = 1$ or $\alpha(a_2) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 0$
  - Else if $\alpha(a_4) = 1$ or $\alpha(a_5) = 1$, then set $\alpha(z_2) = \alpha(z_3) = 1$
  - Else if $\alpha(a_3) = 1$, then set $\alpha(z_2) = 1, \alpha(z_3) = 0$
- If C' is true with assignment β, some a_i will be true and C will be true with β:
  - If $\beta(z_2) = 0$, then $\beta(a_1) = 1$ or $\beta(a_2) = 1$
  - If $\beta(z_3) = 1$, then $\beta(a_4) = 1$ or $\beta(a_5) = 1$
  - If $\beta(z_2) = 1$ and $\beta(z_3) = 0$, then $\beta(a_3) = 1$

NP-complete problems

CNF-SAT $\leq^p$ 3-SAT

The following method transforms
a Boolean formula in CNF into an equisatisfiable one in 3-CNF.

Given a Boolean formula F in CNF,

1. Let F' be empty
2. For each clause $C = (a_1 \vee \cdots \vee a_k)$ in F:
   - if $k \leq 3$, add C to F'
   - if $k > 3$, add to F' the clauses

     $$(a_1 \vee a_2 \vee z_2) \wedge (\neg z_2 \vee a_3 \vee z_3) \wedge (\neg z_3 \vee a_4 \vee z_4) \ldots (\neg z_{k-2} \vee a_{k-1} \vee a_k)$$

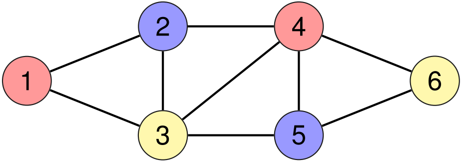
     where $z_2, \ldots, z_{k-2}$ are new variables.
3. Return F'

NP-complete problems

Recall that a graph $G = (V, E)$ is *k*-colorable if there exists a function (called a *k*-coloring)

$$C : V \to \{1, \ldots, k\}$$

such that $C(u) \neq C(v)$ for all $(u, v) \in E$.



3-coloring

NP-complete problems

With the number of colors k as an external parameter,
we can formulate the coloring problem as a function of k.

> k-**Colorability** (k-COLOR)
> Given a graph G, determine whether it is k-colorable.

For the following cases, polynomial algorithms are known:

- 1-COLOR
- 2-COLOR

NP-complete problems

CNF-SAT $\leq^p$ 3-COLOR

Let F be a Boolean formula in CNF.

We will construct a graph G that is 3-colorable if and only if F is satisfiable.

CNF-SAT $\leq^p$ 3-COLOR

Let F be a Boolean formula in CNF.

We will construct a graph G that is 3-colorable if and only if F is satisfiable.

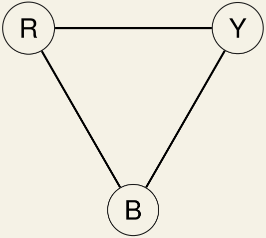
- There will be 3 special vertices called R, Y, B.

CNF-SAT $\leq^p$ 3-COLOR

Let F be a Boolean formula in CNF.

We will construct a graph G that is 3-colorable if and only if F is satisfiable.

- There will be 3 special vertices called R, Y, B.
- We add these edges:



We can assume that in any coloring, they have the colors:

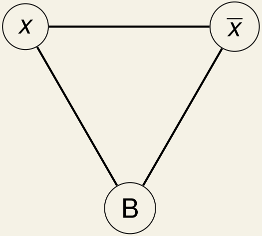
R → red, Y → yellow, B → blue

NP-complete problems

- There will be a vertex for each literal.

NP-complete problems

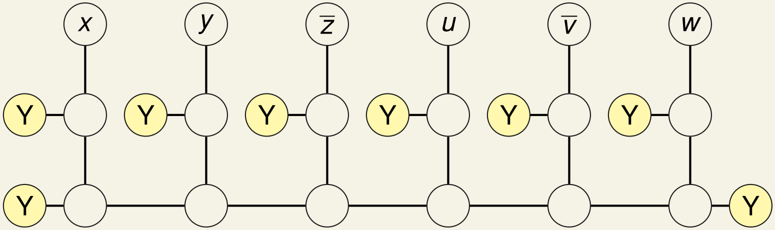
- There will be a vertex for each literal.
- We connect each literal and its negation to vertex B.



- For each clause, we add a subgraph as follows.

  For example, for a clause with an even number of literals:

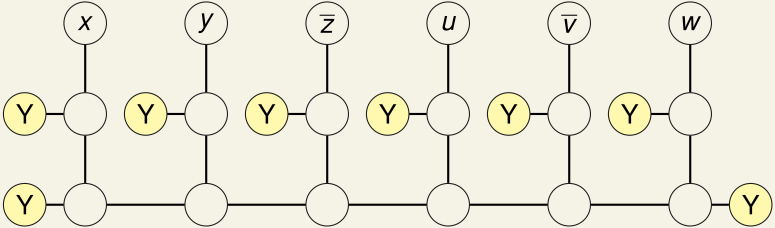
$$(x \vee y \vee \overline{z} \vee u \vee \overline{v} \vee w).$$



- For each clause, we add a subgraph as follows.

  For example, for a clause with an even number of literals:

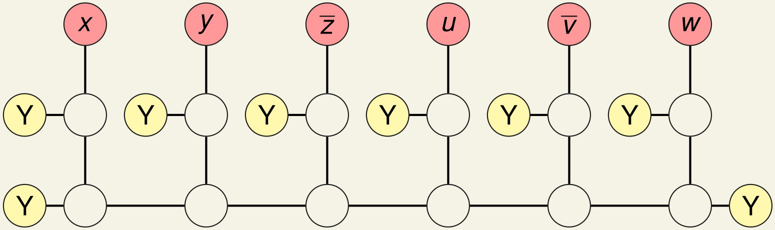
$$(x \vee y \vee \overline{z} \vee u \vee \overline{v} \vee w).$$



Property: A coloring of the upper vertices with red or yellow can be extended to a global 3-coloring if and only if at least one has yellow color.

NP-complete problems

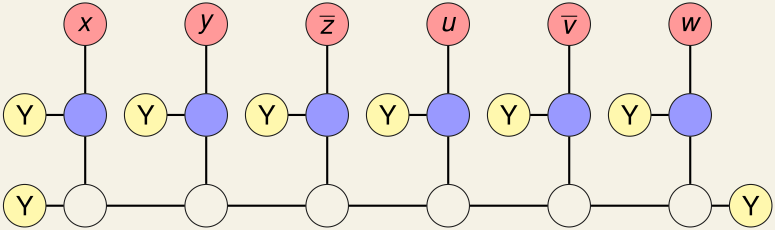
If all of the upper vertices are red....



...we cannot complete the 3-coloring.

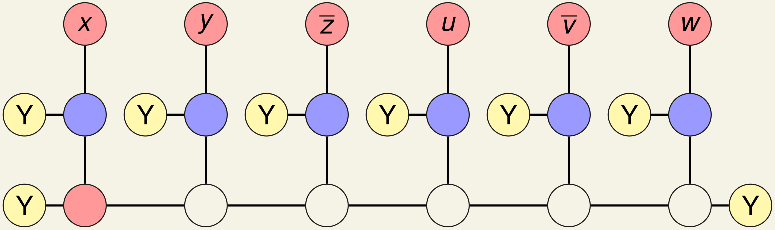
NP-complete problems

If all of the upper vertices are red....



...we cannot complete the 3-coloring.

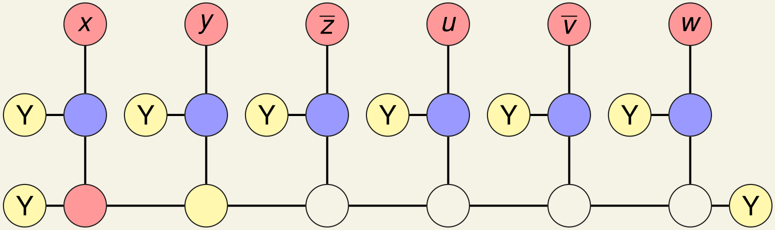
If all of the upper vertices are red....



...we cannot complete the 3-coloring.

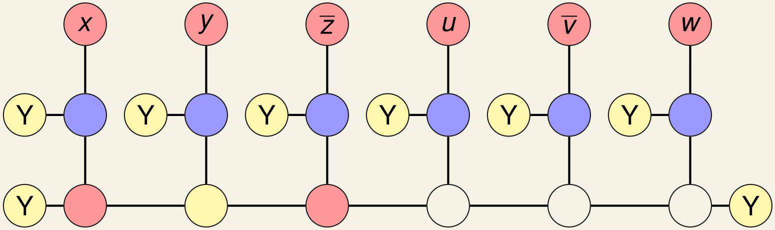
NP-complete problems

If all of the upper vertices are red....



...we cannot complete the 3-coloring.

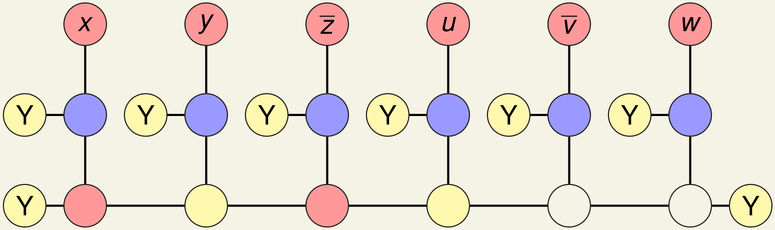
If all of the upper vertices are red....



...we cannot complete the 3-coloring.

NP-complete problems

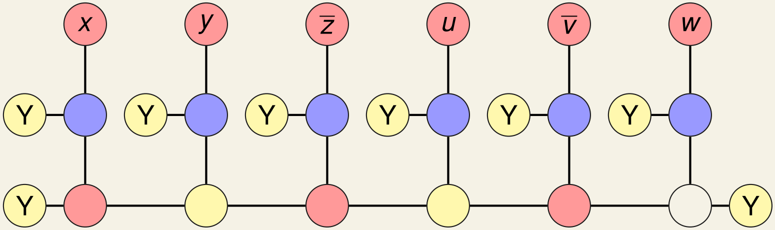
If all of the upper vertices are red....



...we cannot complete the 3-coloring.

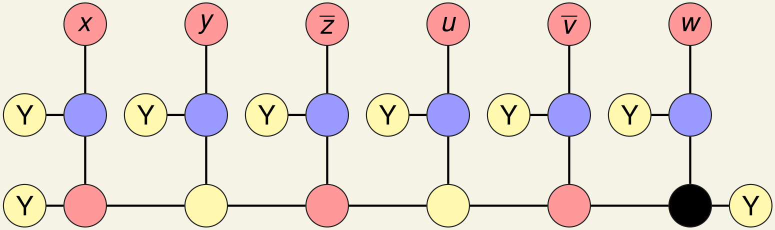
NP-complete problems

If all of the upper vertices are red....



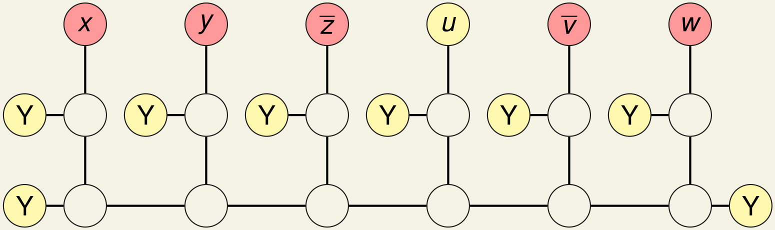
...we cannot complete the 3-coloring.

If all of the upper vertices are red....



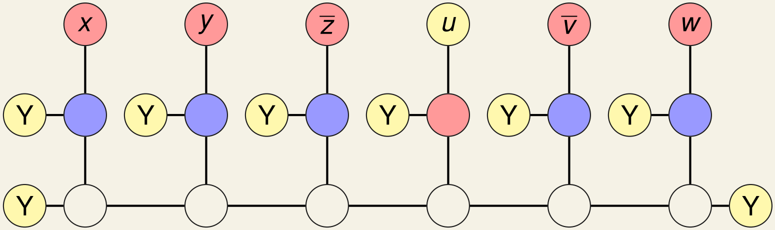
...we cannot complete the 3-coloring.

If at least one is yellow...



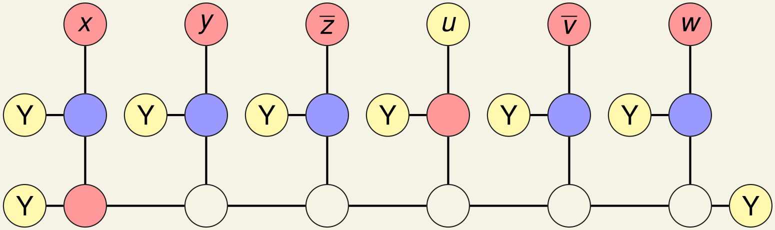
...we can obtain a global 3-coloring.

If at least one is yellow...



...we can obtain a global 3-coloring.

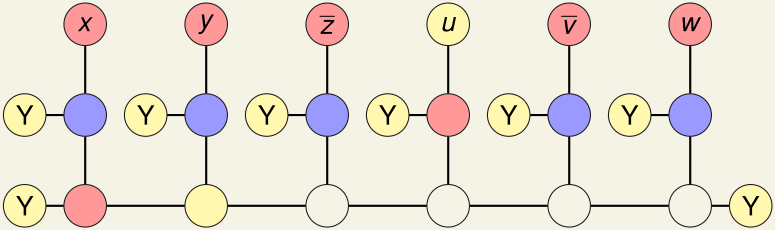
If at least one is yellow...



...we can obtain a global 3-coloring.

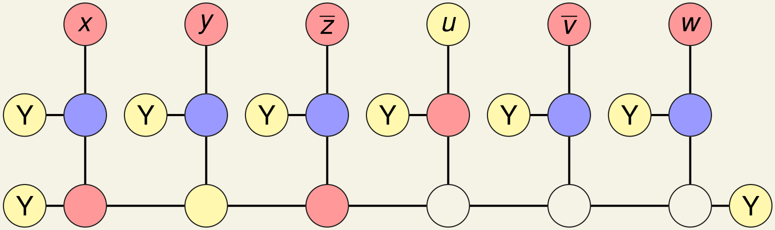
NP-complete problems

If at least one is yellow...



...we can obtain a global 3-coloring.

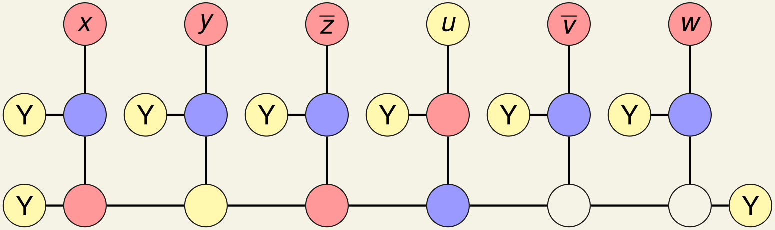
If at least one is yellow...



...we can obtain a global 3-coloring.

NP-complete problems

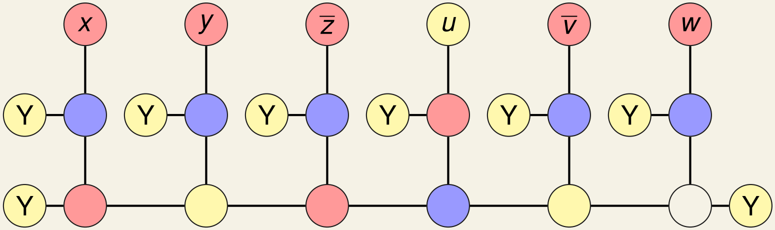
If at least one is yellow...



...we can obtain a global 3-coloring.

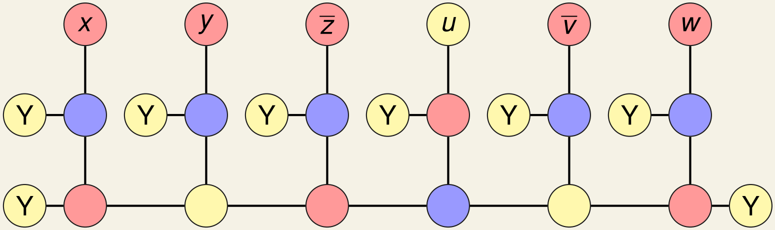
NP-complete problems

If at least one is yellow...



...we can obtain a global 3-coloring.

If at least one is yellow...

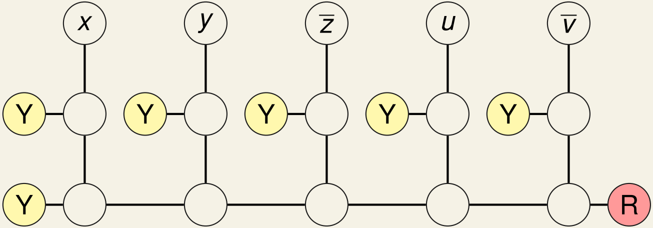


...we can obtain a global 3-coloring.

NP-complete problems

If the number of literals is odd, the rightmost vertex will be R.
For example,

$$(x \vee y \vee \overline{z} \vee u \vee \overline{v})$$



NP-complete problems

If G is the graph with all vertices and edges defined as before, then

$$F \text{ is satisfiable} \Leftrightarrow G \text{ is 3-colorable.}$$

Since G can be constructed in polynomial time, we have that

$$\text{CNF-SAT} \leq^p \text{3-COLOR}.$$

Theorem

3-COLOR is NP-complete.

NP-complete problems

For the other k-COLOR problems, we have the following.

Proposition

For all $k > 3$, 3-COLOR $\leq^p$ k-COLOR.

The reduction consists in, given a graph G,
adding to it a clique of $k - 3$ fresh vertices connected to all vertices of G.

NP-complete problems

For the other k-COLOR problems, we have the following.

Proposition

For all $k > 3$, 3-COLOR $\leq^p$ k-COLOR.

The reduction consists in, given a graph G,
adding to it a clique of $k - 3$ fresh vertices connected to all vertices of G.

Corollary

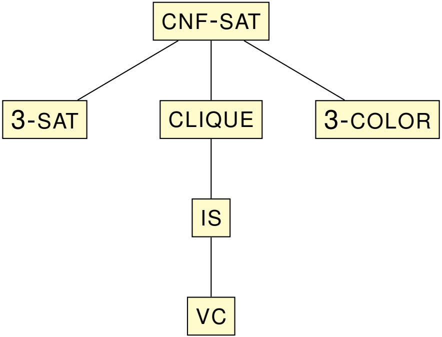
For all $k > 3$, k-COLOR is NP-complete.

Hence, we have:

- k-COLOR $\in \mathrm{P}$ for all $k \leq 2$
- k-COLOR is NP-complete for all $k \geq 3$

NP-complete problems

So far, we have seen the following tree of reductions.



But this is not the end of the story...

NP-complete problems

- There are hundreds and hundreds of known NP-complete problems

NP-complete problems

- There are hundreds and hundreds of known NP-complete problems
- You may be very familiar with some of them; e.g. see:

Gualà L.; Leucci, S. ; Natale, E.:
Bejeweled, Candy Crush and other match-three games are (NP-)hard.
IEEE Conference on Computational Intelligence and Games (2014): p. 1-8

NP-complete problems

- To see more NP-complete problems and reductions:

Garey, M.; Johnson, D. (1979),
Computers and Intractability:
A Guide to the Theory of NP-Completeness,
W. H. Freeman, ISBN 0-7167-1045-5.