A tourist guide through computational complexity

Why bother proving a problem to be hard ?

Stéphane Vialette

`Stephane.Vialette@ens.fr`

LGM

École Normale Supérieure

Computational complexity

Computational Complexity studies:

- the efficiency of algorithms;
- the inherent "*difficulty*" of problems of practical and/or theoretical importance.

The time complexity of a problem is the number of steps that it takes to solve an instance (as a function of the size of the instance).

Review of order notation

$f(n) = \mathcal{O}(g(n))$ iff $\exists c\, \exists n_0\, \forall n \geq n_0, \;\; f(n) \leq c\, g(n)$

$f(n) = \Omega(g(n))$ iff $\exists c\, \exists n_0\, \forall n \geq n_0, \;\; f(n) \geq c\, g(n)$

$f(n) = \Theta(g(n))$ iff $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$

Review of order notation

$f(n) = \mathcal{O}(g(n))$ iff $\exists c\, \exists n_0\, \forall n \geq n_0,\ \ f(n) \leq c\, g(n)$

$f(n) = \Omega(g(n))$ iff $\exists c\, \exists n_0\, \forall n \geq n_0,\ \ f(n) \geq c\, g(n)$

$f(n) = \Theta(g(n))$ iff $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$

- If $f(n) = \mathcal{O}(g(n))$ then f has at most rate of growth $g(n)$
- If $f(n) = \Omega(g(n))$ then f has at least rate of growth $g(n)$
- If $f(n) = \Theta(g(n))$ then f has rate of growth $g(n)$

Review of order notation

Note that for sufficiently large n:

$$\log n < n < n \log n < n^2 < n^3 < 2^n$$

$\log n$ is interpreted as base-2 logarithm (it does not really matter, since $\log_2 n = log_{10}n/log_{10}2$ and constants are ignored as already mentioned).

This is sometimes stated as:

$$\mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \mathcal{O}(2^n)$$

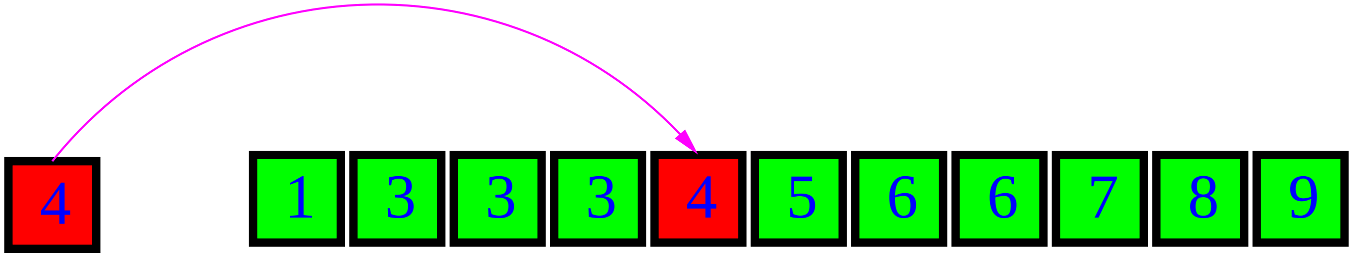
Polynomial time algorithms

A *polynomial time algorithm* or a *good algorithm* is one that runs in $O(p(n))$ time for some polynomial $p(n)$.

Polynomial time algorithms

A *polynomial time algorithm* or a *good algorithm* is one that runs in $O(p(n))$ time for some polynomial $p(n)$.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half

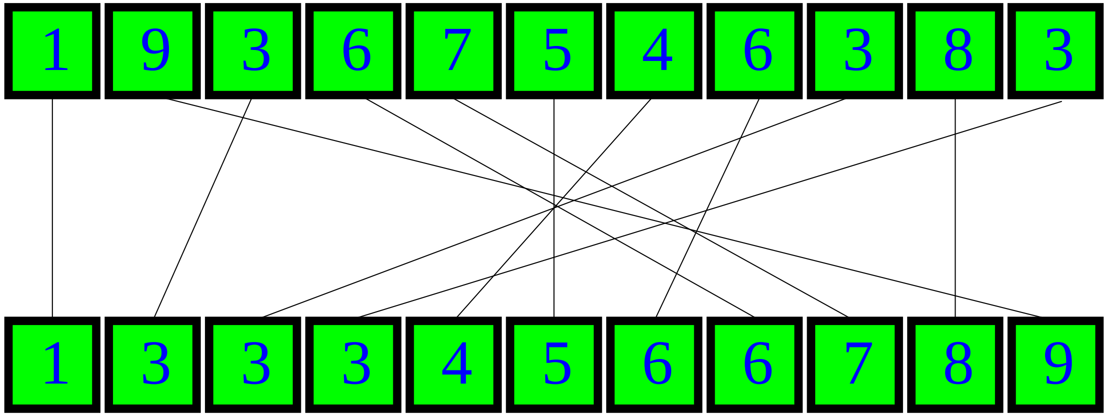


Running time is $\mathcal{O}(\log n)$

Polynomial time algorithms

A *polynomial time algorithm* or a *good algorithm* is one that runs in $O(p(n))$ time for some polynomial $p(n)$.

Quicksort: An in-place sort algorithm that uses the divide and conquer paradigm

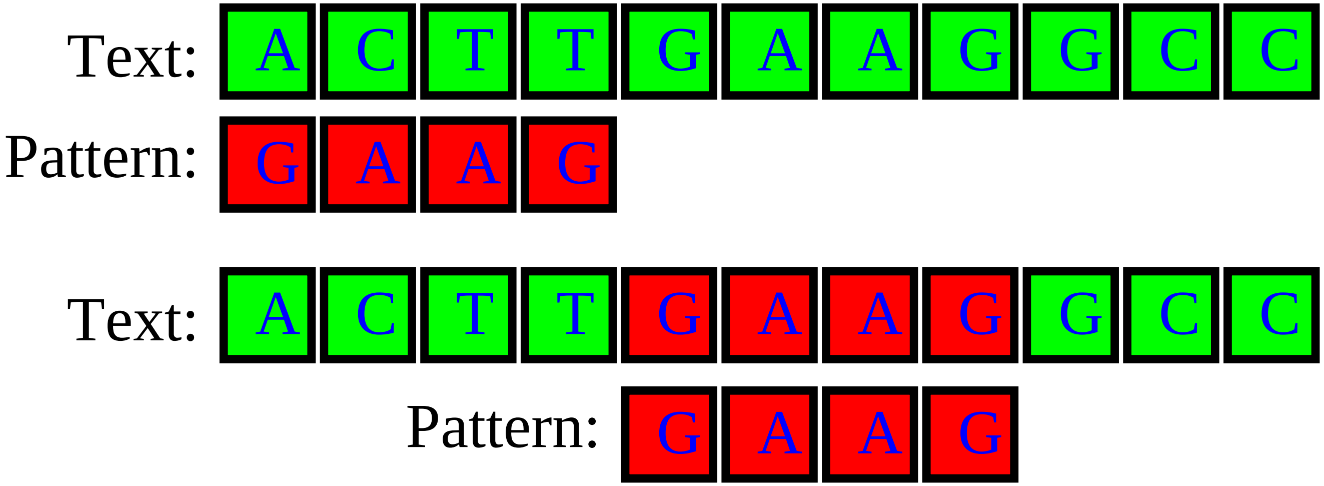


Quicksort has running time upper-case $\Theta(n^2)$ in the worst case, but it is typically $\mathcal{O}(n \log n)$.

Polynomial time algorithms

A *polynomial time algorithm* or a *good algorithm* is one that runs in $O(p(n))$ time for some polynomial $p(n)$.

String matching (brute force algorithm): Check whether an occurrence of the pattern starts there or not



Text: A C T T G A A G G C C
Pattern: G A A G

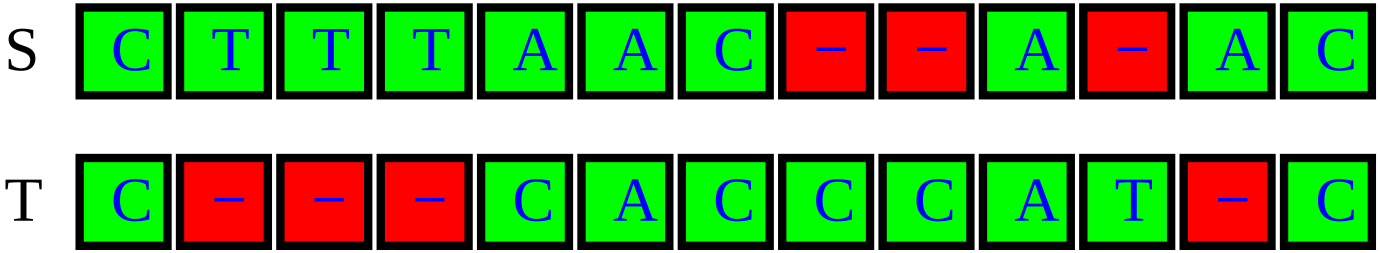
Text: A C T T G A A G G C C
Pattern: G A A G

Searching phase in $\mathcal{O}(m^n)$ time complexity

Polynomial time algorithms

A *polynomial time algorithm* or a *good algorithm* is one that runs in $O(p(n))$ time for some polynomial $p(n)$.

Optimal Alignment: Compute an optimal alignment of S and T.



S	C	T	T	T	A	A	C	–	–	A	–	A	C
T	C	–	–	–	C	A	C	C	C	A	T	–	C

Running time is $\mathcal{O}(m^n)$

The Fundamental Question

Do polynomial time algorithms exist for all problems ?

- The answer to this question is not known
- Is P $=$ NP ?
- It is one of the great mysteries of modern computer science

I can’t find an efficient algorithm . . .

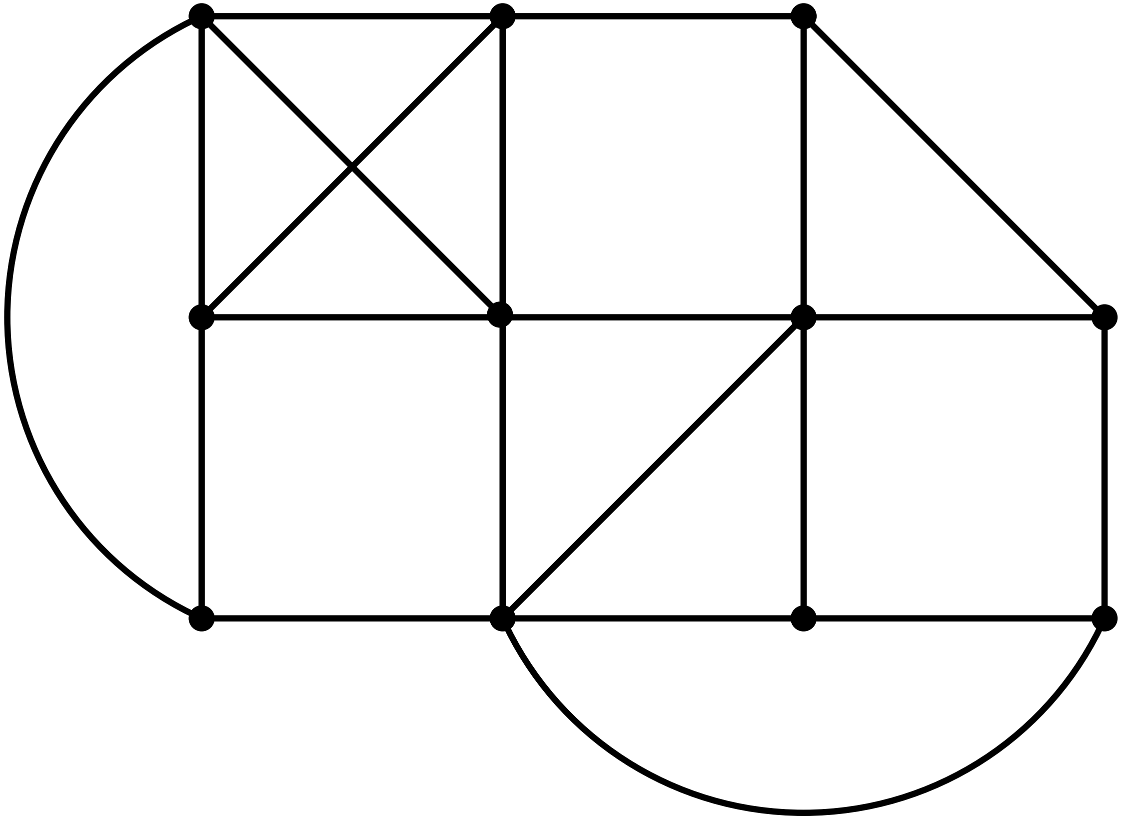
“I can’t find an efficient algorithm, I guess I’m just too dumb.”

“I can’t find an efficient algorithm, because no such algorithm is possible.”

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

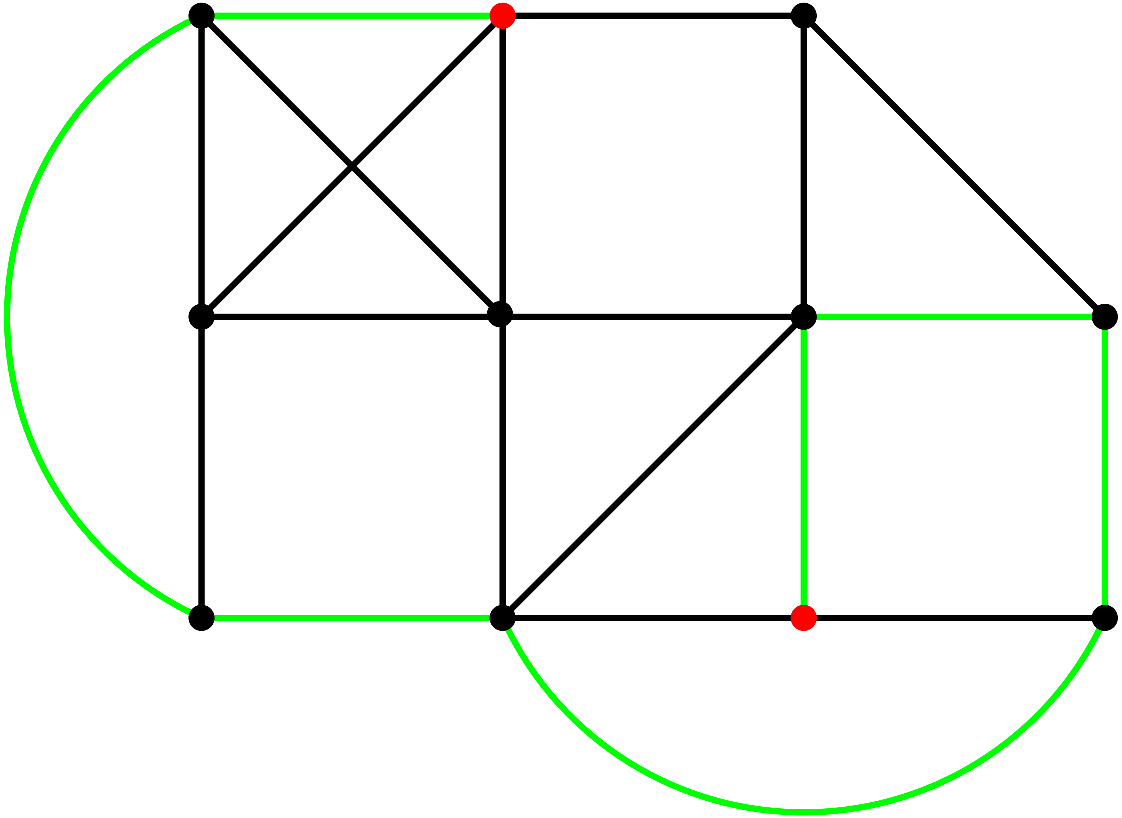
"*Easy*" problems

Graph $G = (V, E)$



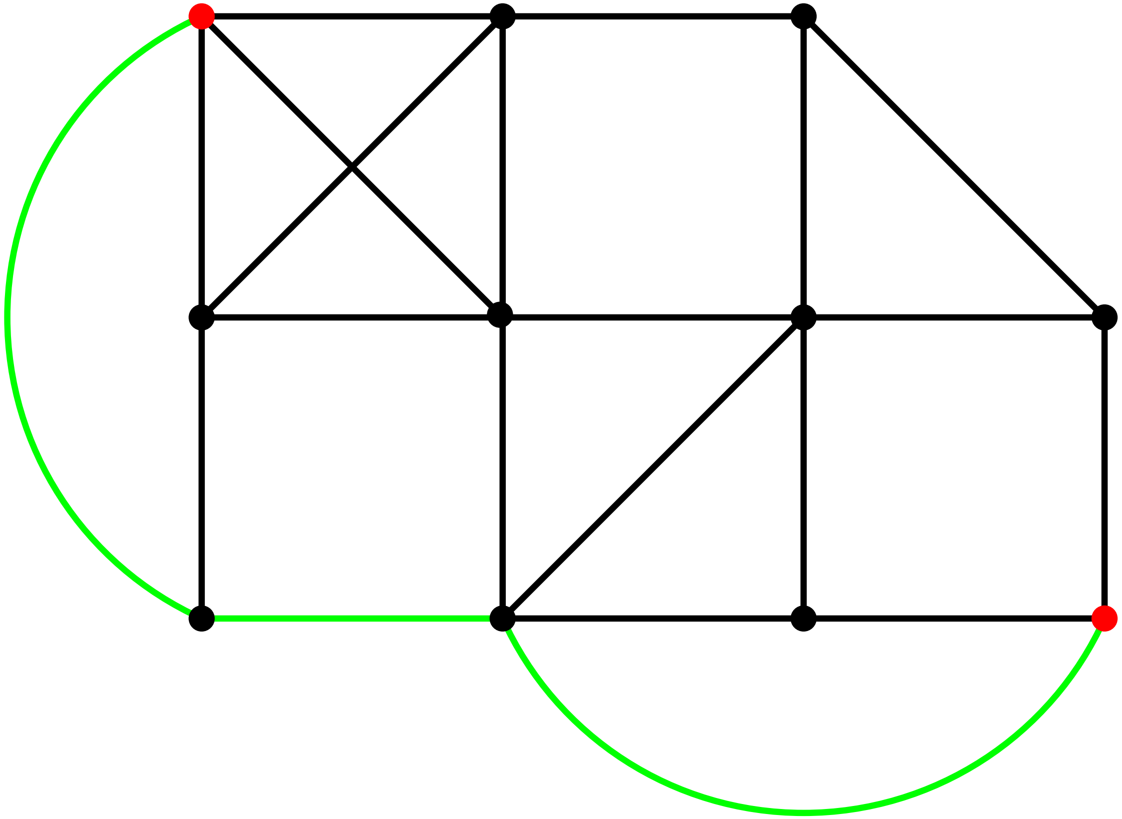
"*Easy*" problems

Is there exist a path between two vertices in G ?



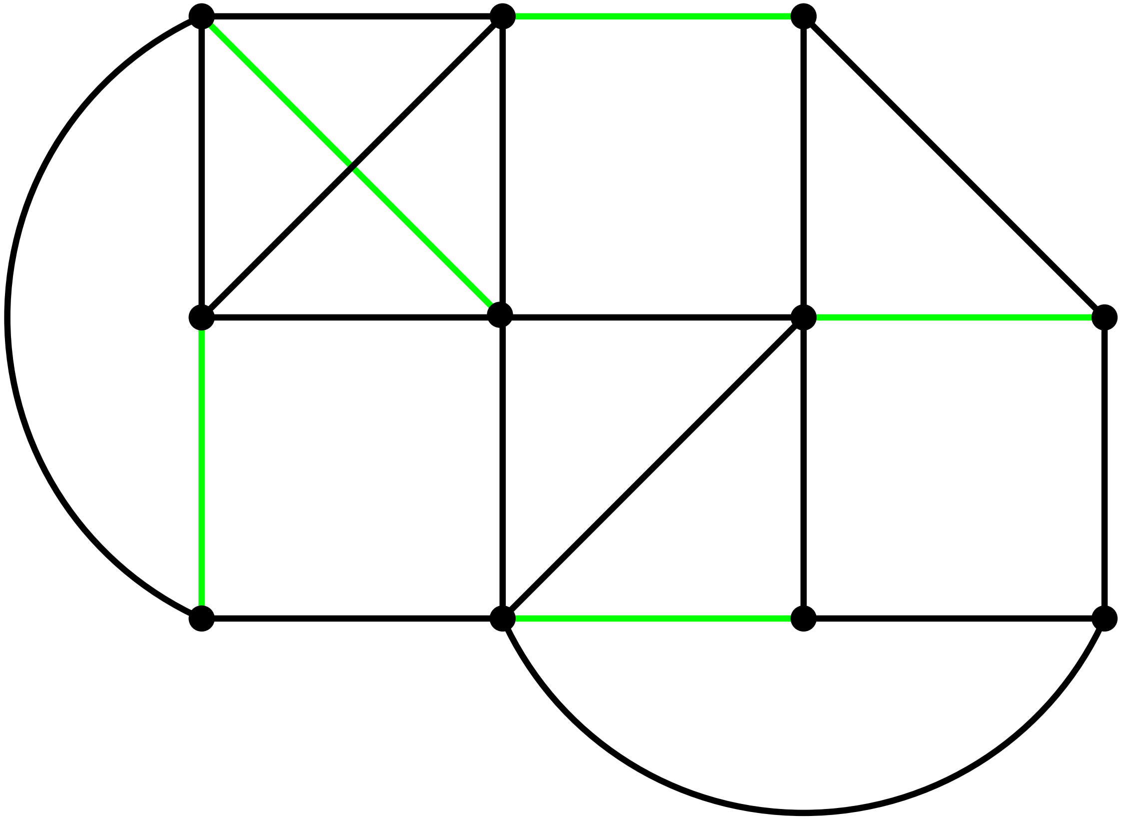
"*Easy*" problems

Finding a *shortest path* between two vertices in G



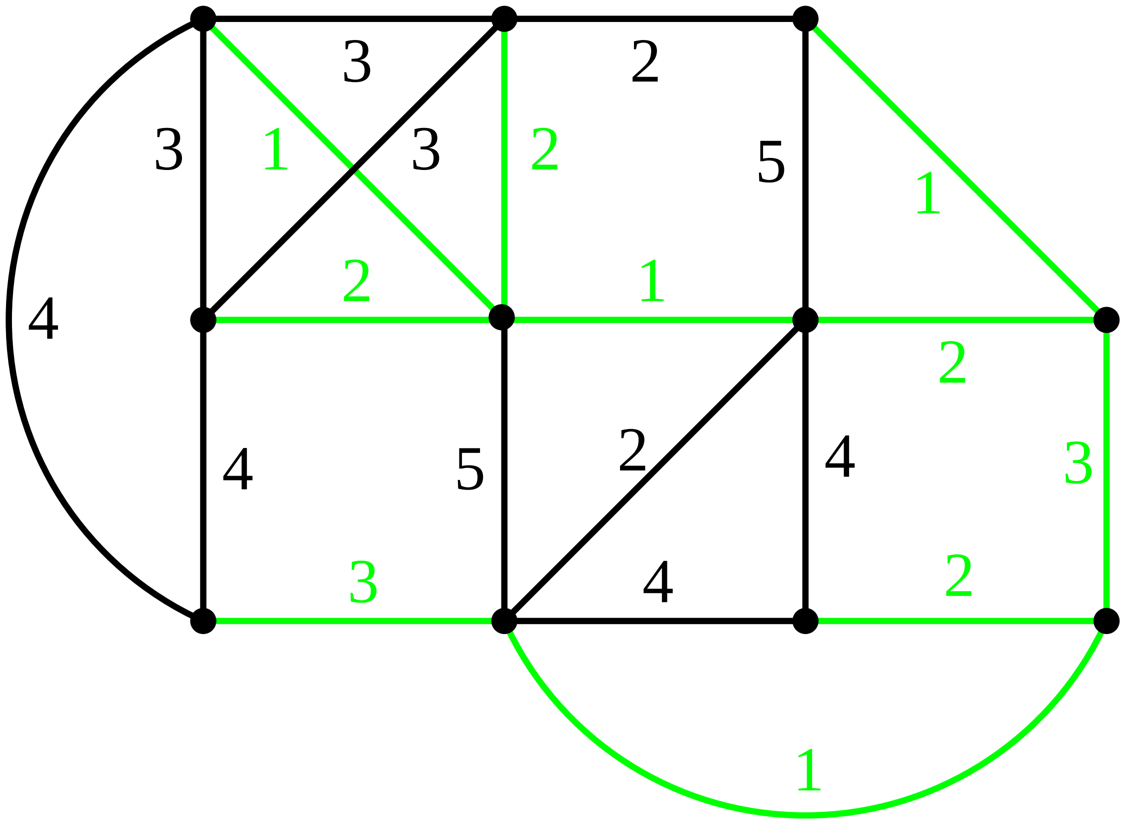
"*Easy*" problems

Finding a maximum *matching* in G



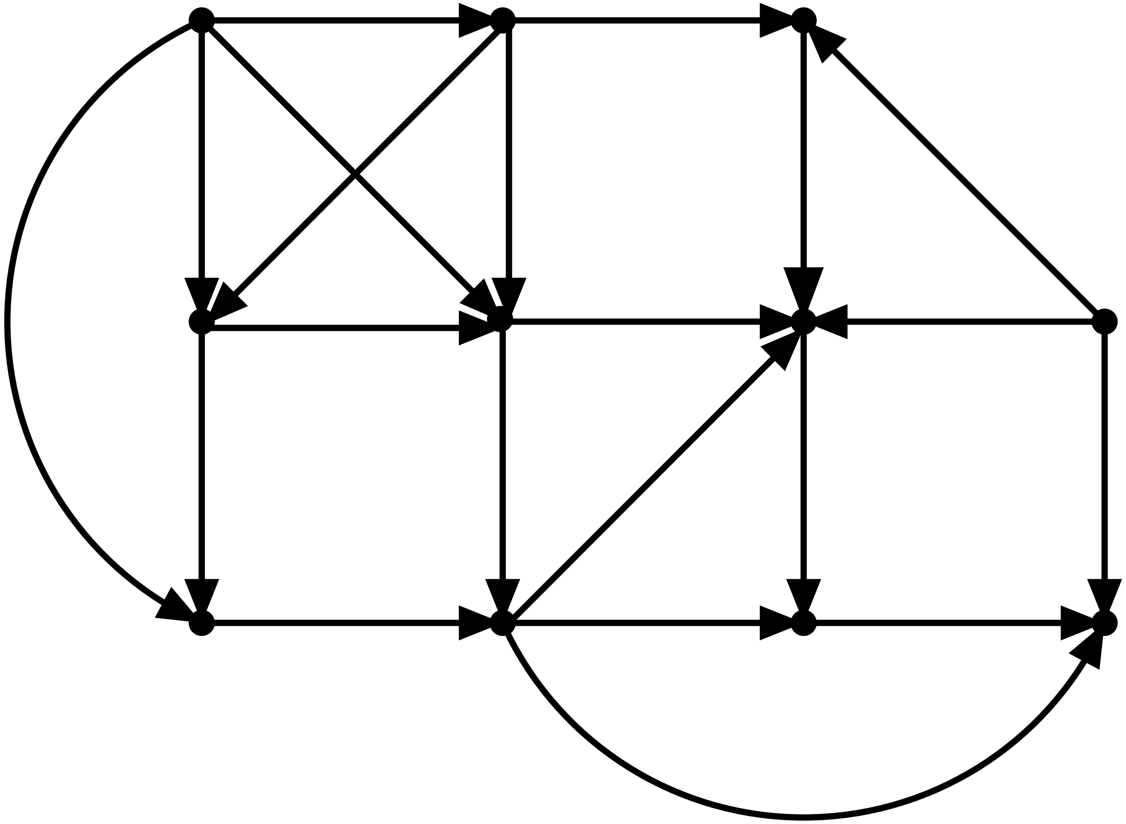
"*Easy*" problems

Finding a minimum weight *spanning tree* in G



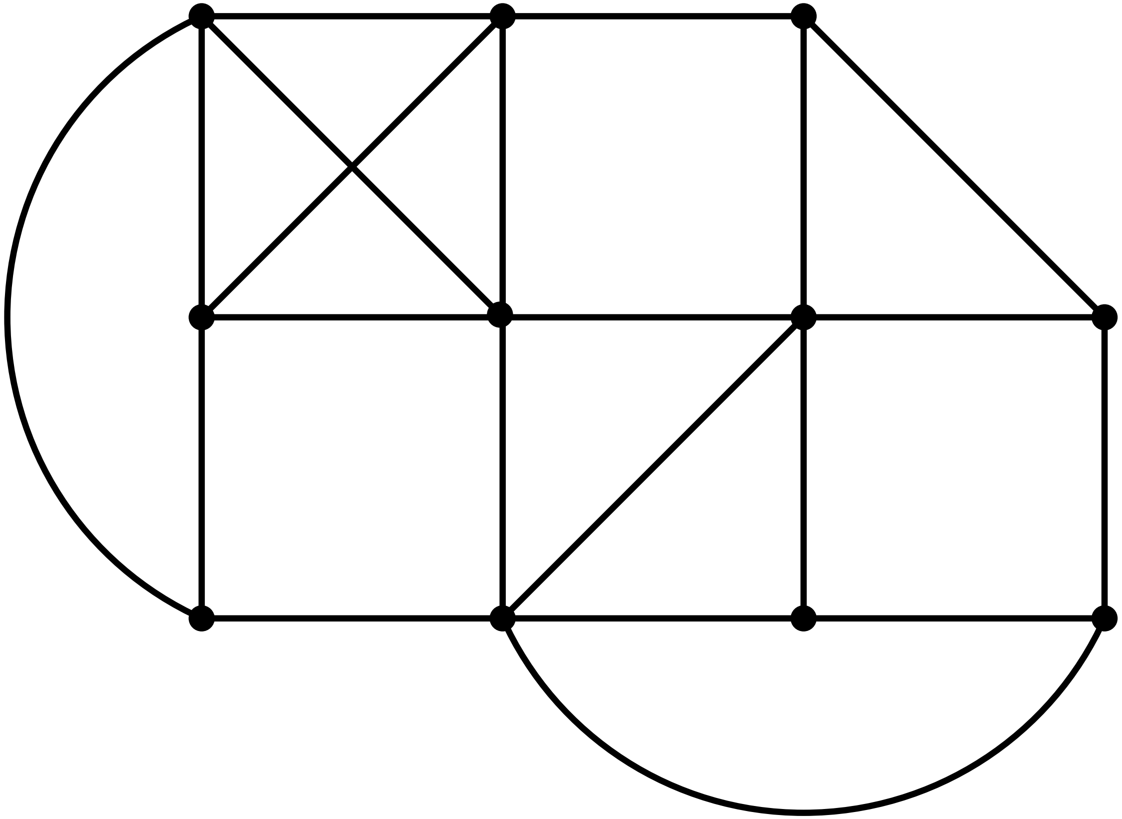
"*Easy*" problems

Is the directed graph G *acyclic* ?



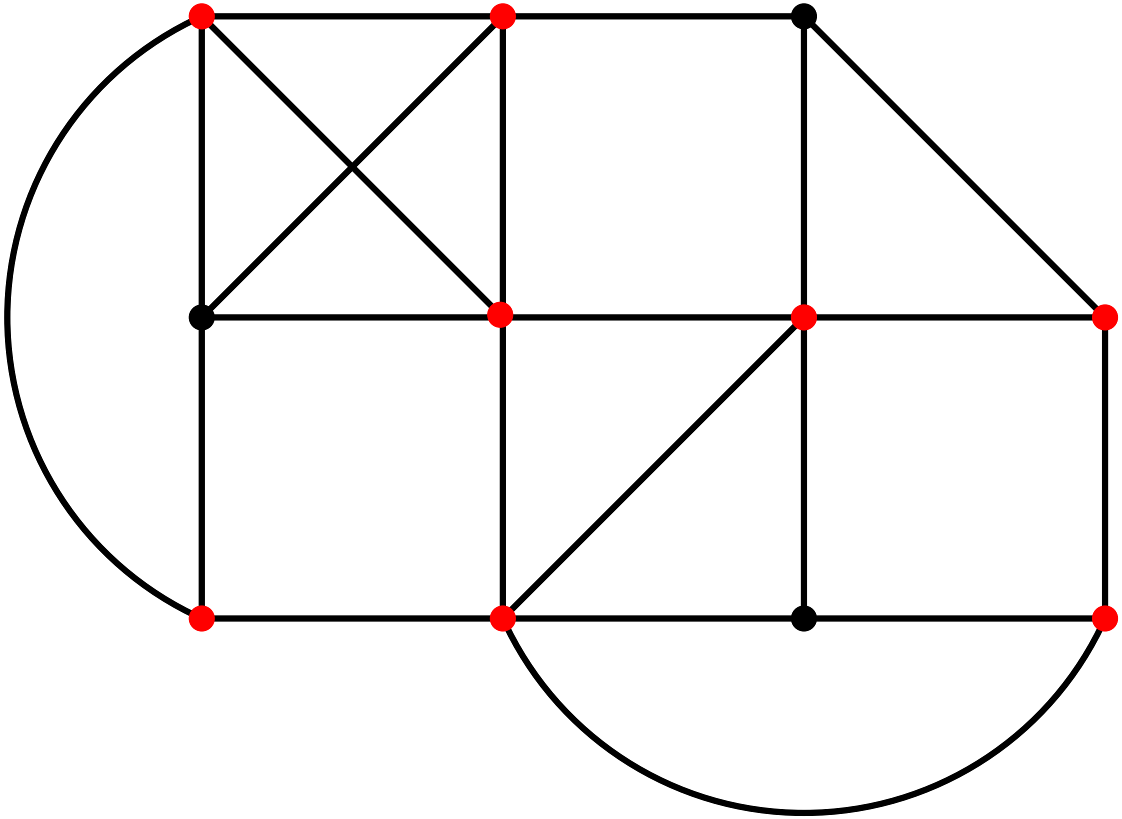
"*Hard*" problems

Graph $G = (V, E)$



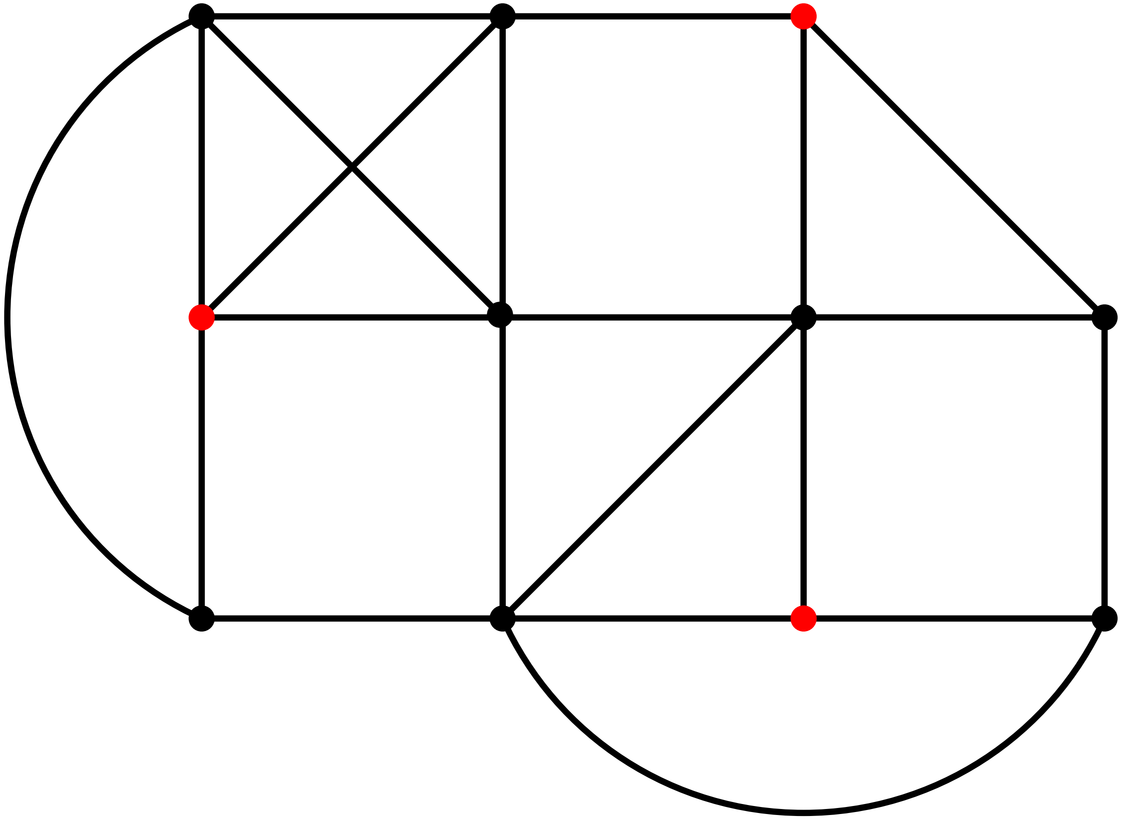
"*Hard*" problems

Finding a minimum *vertex cover* in G



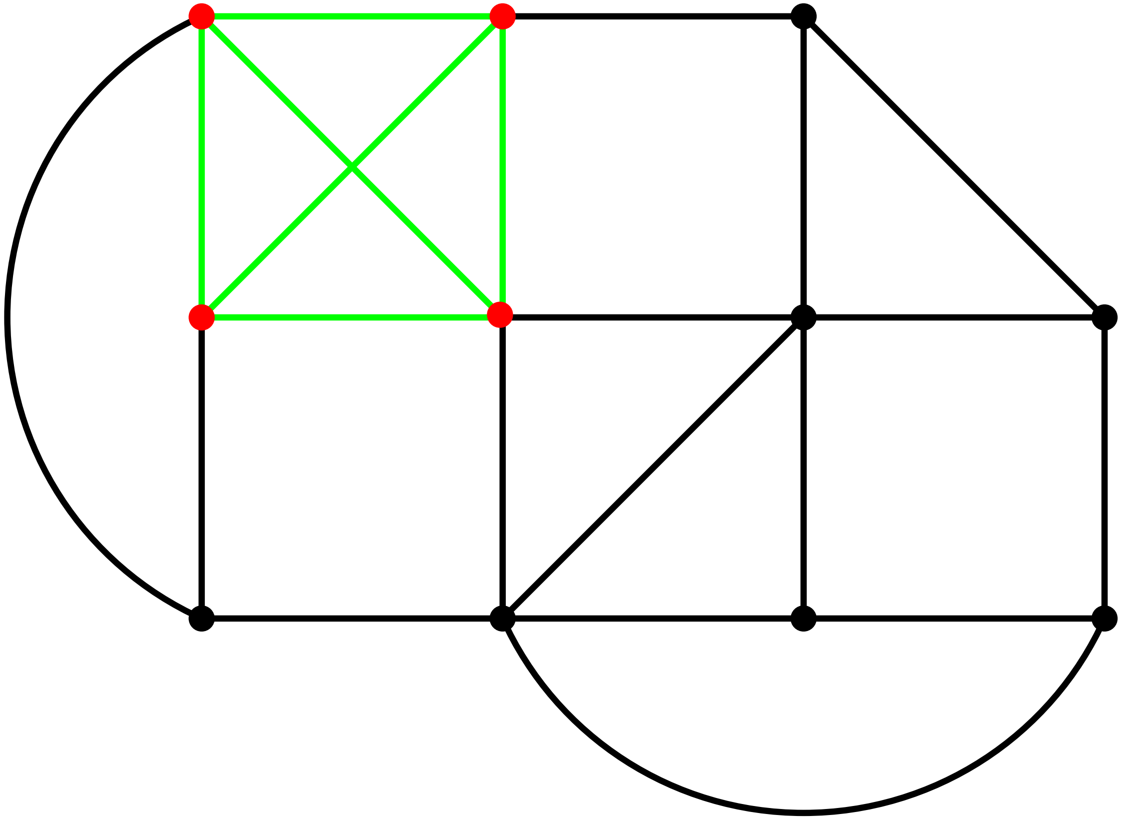
"*Hard*" problems

Finding a maximum *independent set* in G



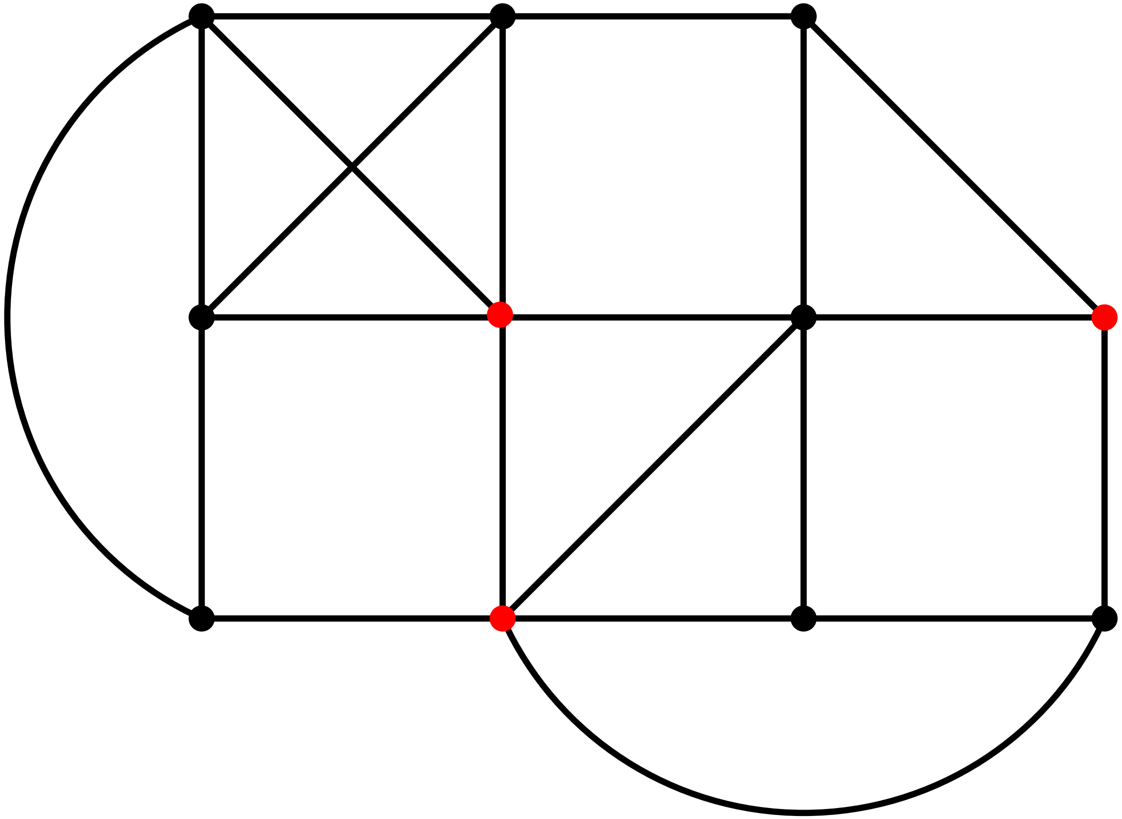
"*Hard*" problems

Finding a maximum *clique* in G



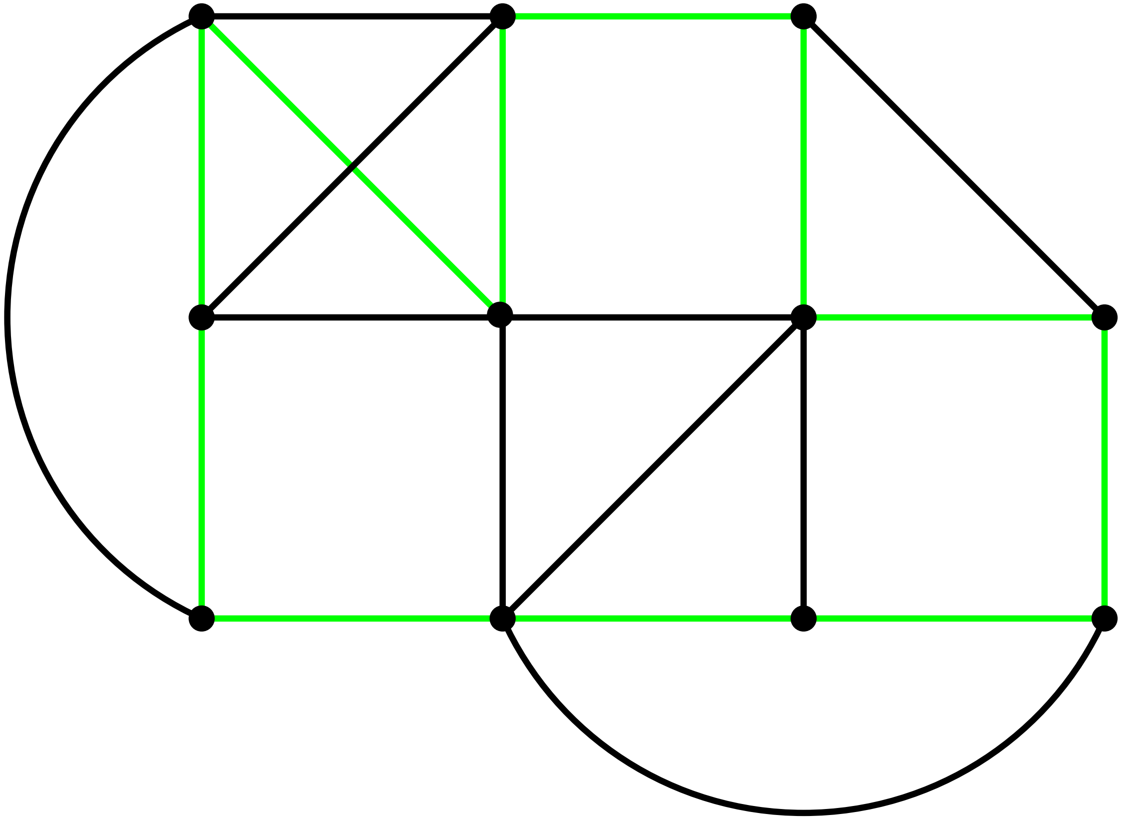
"*Hard*" problems

Finding a minimum *dominating set* in G



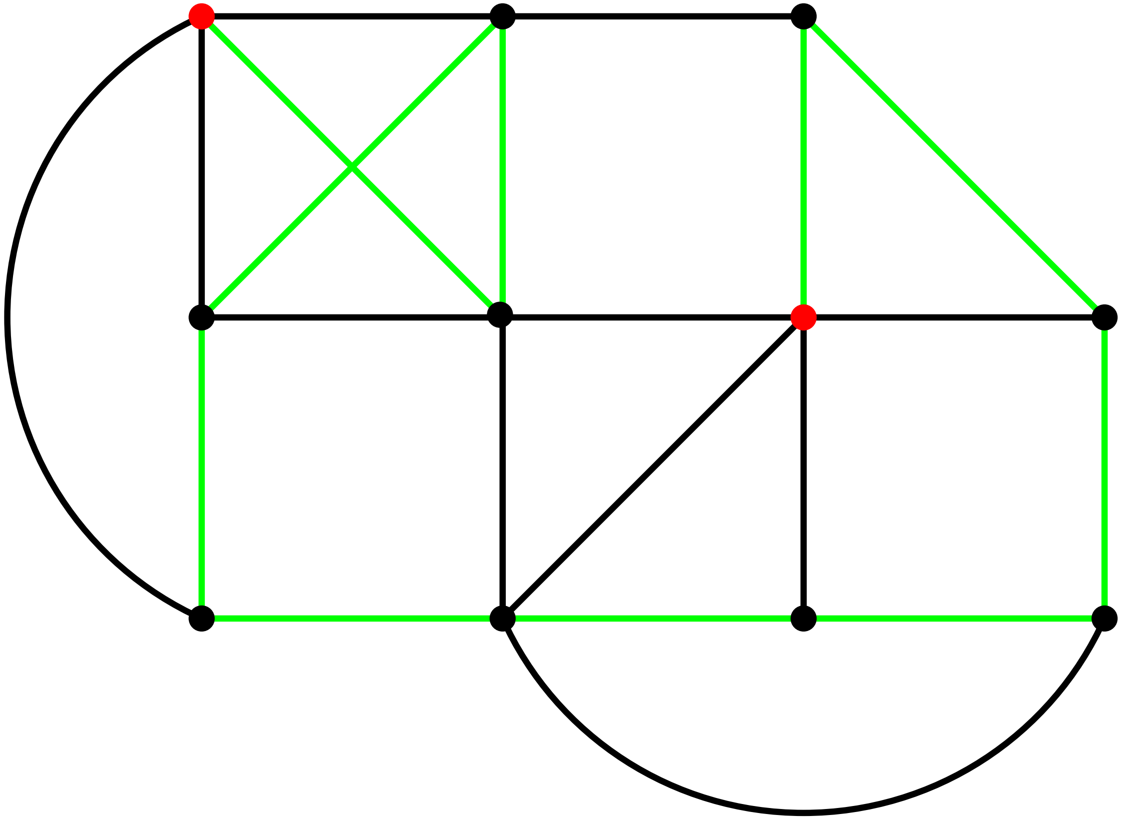
"*Hard*" problems

Finding an *hamiltonian circuit* in G



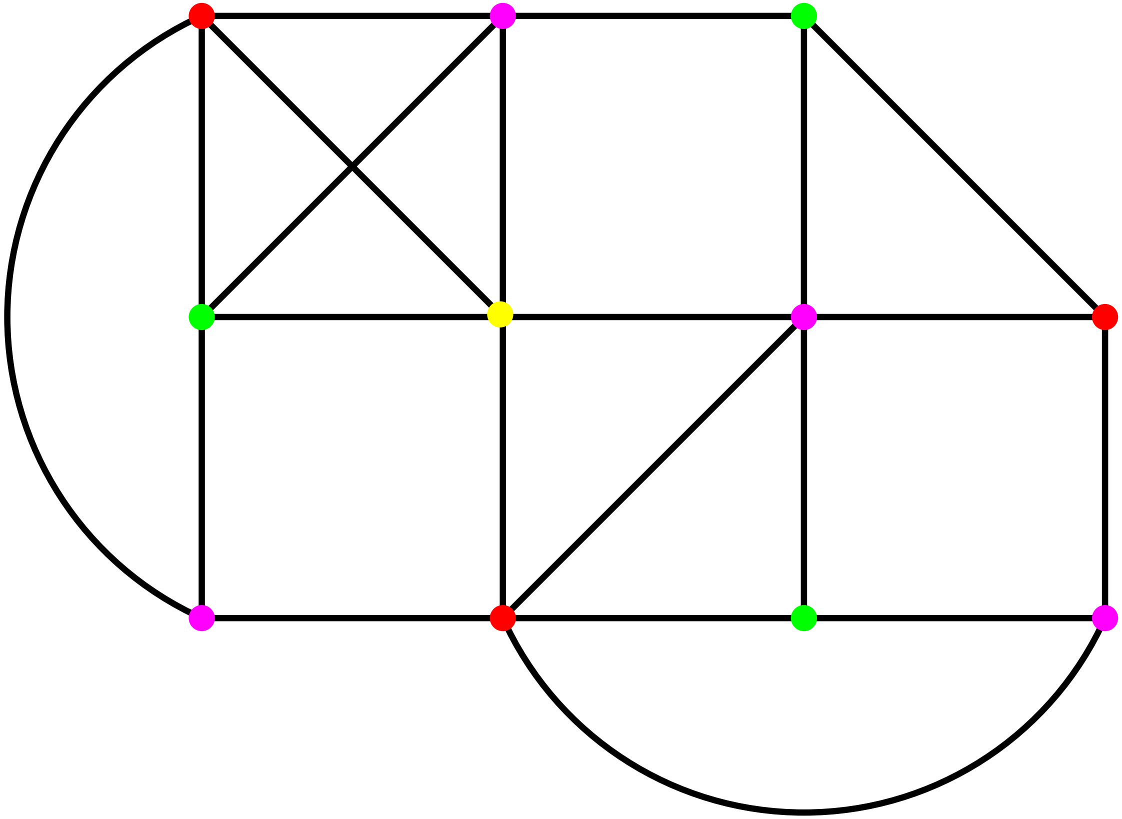
"*Hard*" problems

Finding an *hamiltonian path* between two vertices in G



"*Hard*" problems

Finding a minimum coloring of G



"*polynomial*" is a synonym to practical

Time complexity	Size of Largest Problem Instance solvable in 1 Hour		
	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
2^n	N_4	$N_4 + 6.64$	$N_4 + 9.97$
3^n	N_5	$N_5 + 4.19$	$N_5 + 6.29$

- The effect of improved technology is multiplicative in polynomial-time algorithms and only additive in exponential-time algorithms.
- The situation is much worse than that shown in the table if complexities involve factorials.

Decision problems

A decision problem is a problem where the answer is always "*YES*" or "*NO*".

- An arbitrary problem can always be reduced to a decision problem.
- Complexity theory often makes a distinction between "*YES*" answers or "*NO*" answers.

Classes P and NP

The class P consists of all those recognition problems for which a polynomial-time algorithm exists.

For the class NP, we simply require that any "*yes*" answer is "*easily*" verifiable. That is, both the encoding of the answer and the time it takes to check its validity must be "*short*", i.e. polynomially bounded. Formally we say that any "*yes*" instance of the problem has the "*succinct certificate*" property.

NP stands for "*Non deterministic Polynomial*", because of an alternative (and equivalent) definition based on the notion of non-deterministic algorithms.

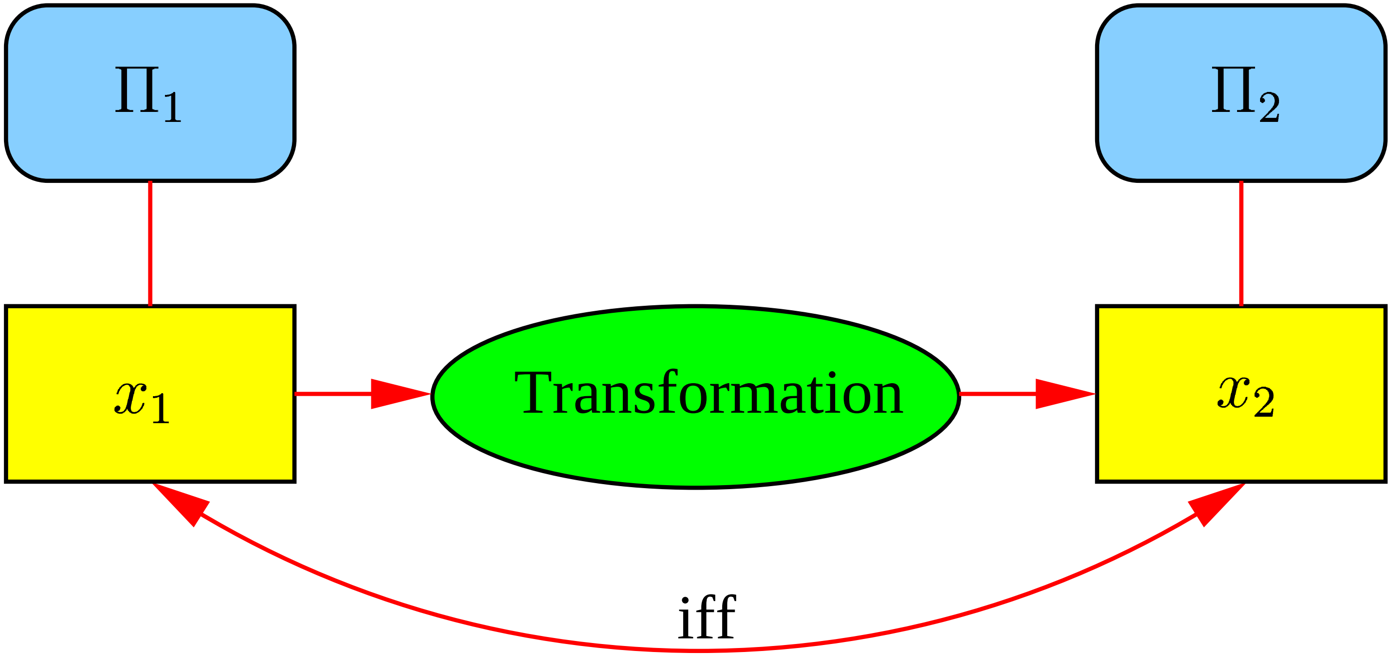
"$\in$ P": *easy*" and "$\notin$ NP": "*hard*"

This is not always true in practice:

- It ignores constant factors. A problem that takes time $10^{100}n$ is in P (in fact, it's linear time), but is completely intractable in practice. A problem that takes time $10^{-1000}2^n$ is not in P (in fact, it's exponential time), but is very tractable for values of n up into the thousands.
- It ignores the size of the exponents. A problem with time n^{1000} is in P, yet intractable. A problem with time $2^{n/1000}$ is not in P, yet is tractable for n up into the thousands.
- It only considers worst-case times. There might be a problem that arises in the real world. Most of the time, it can be solved in time n, but on very rare occasions you'll see an instance of the problem that takes time 2^n. This problem might have an average time that is polynomial, but the worst case is exponential, so the problem wouldn't be in P.

Polynomial time transformation

A decision problem Π_1 *polynomially transforms* to another decision problem Π_2 if, given any instance x_1 of Π_1 we can construct a corresponding instance x_2 of Π_2 within polynomial (in $|x_1|$) time such that x_1 is a "*yes*" instance of Π_1 *if and only if* x_2 is a "*yes*" instance of Π_2.



NP-hard and NP-complete problem

A decision problem Π is NP-hard if all other problems in NP polynomially transform to Π.

A decision problem Π in NP is NP-complete if all other problems in NP polynomially transform to Π.

- This definition was given by Stephen Cook in 1971.
- At first it seems rather surprising that NP-complete problems should even exist, but in a celebrated theorem Cook proved that the Boolean satisfiability problem is NP-complete.
- If a problem Π is NP-complete, then it has a formidable property: If there is an efficient algorithm for Π, then there is an efficient algorithm for every problem is NP.

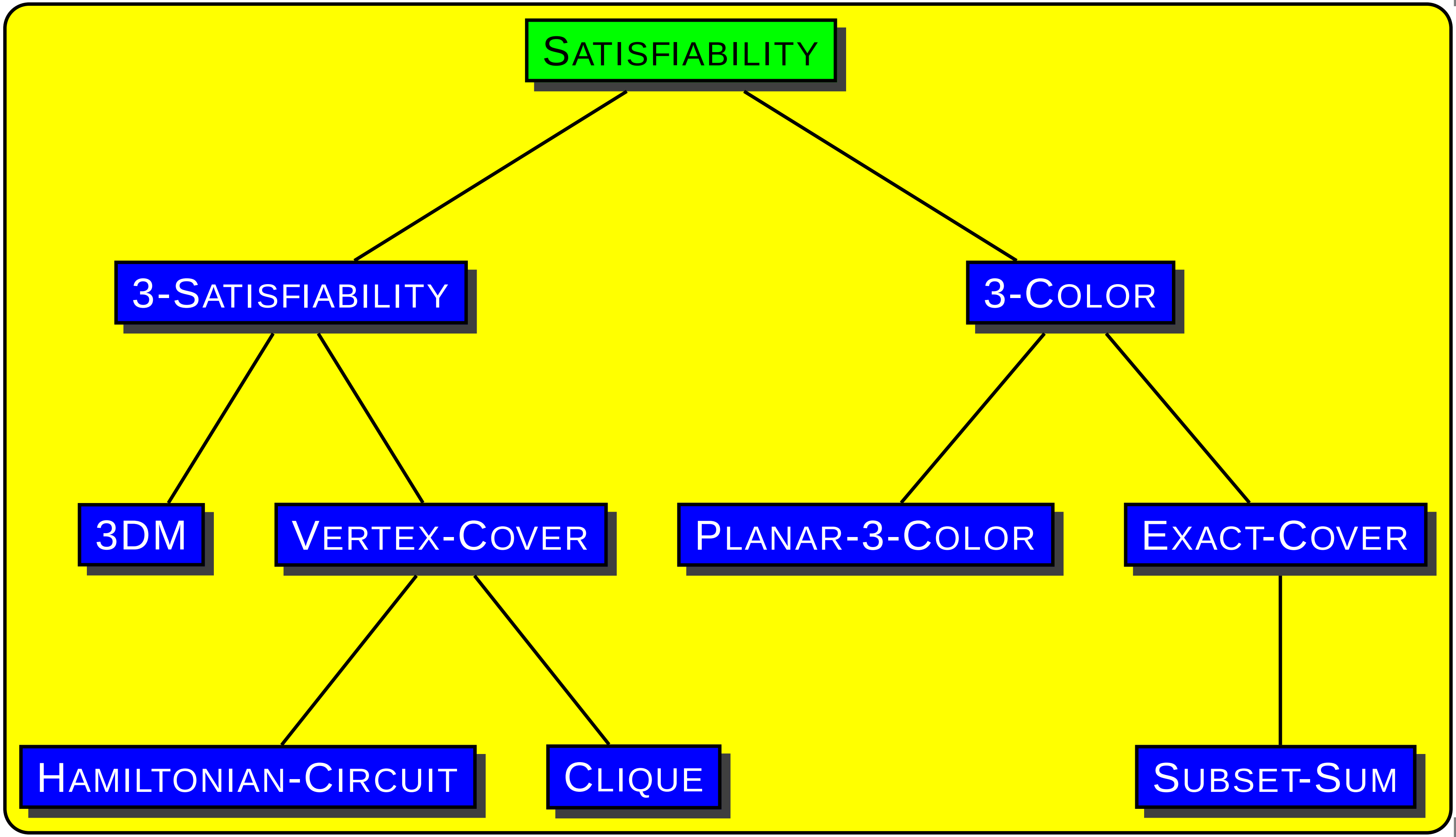
Proving NP-complete results

In order to prove that a problem is NP-complete, we must show two things:

1. That the problem is in NP.
2. That all other problems in NP polynomially transform to our problem.

In practice, Part 2 is usually carried out by showing that a known NP-complete problem is polynomially transformable to the problem at hand.

Dick Karp (1972)



SATISFIABILITY

3-SATISFIABILITY

3-COLOR

3DM

VERTEX-COVER

PLANAR-3-COLOR

EXACT-COVER

HAMILTONIAN-CIRCUIT

CLIQUE

SUBSET-SUM

LONGEST-COMMON-SUBSEQUENCE (LCS)

LCS

Instance: *A set of* n *strings* $S_1, S_2, \ldots, S_n$ *over an alphabet* $\mathcal{A}$ *and a positive integer* L.
Question: *Is there a string* $x \in \mathcal{A}^*$ *of length at least* L *that is a subsequence of* S_i *for* $1 \leq i \leq n$ *?*

LONGEST-COMMON-SUBSEQUENCE (LCS)

LCS

Instance: *A set of n strings $S_1, S_2, \ldots, S_n$ over an alphabet $\mathcal{A}$ and a positive integer L.*
Question: *Is there a string $x \in \mathcal{A}^*$ of length at least L that is a subsequence of S_i for $1 \leq i \leq n$?*

$$S_1 = AAGGGATTCATAGT$$

$$S_2 = ATATAGTGAAACATCG$$

$$S_3 = GAAGCTACAATGAGCC$$

$$S_4 = AGGACCCAATGACGG$$

LONGEST-COMMON-SUBSEQUENCE (LCS)

LCS

Instance: *A set of n strings $S_1, S_2, \ldots, S_n$ over an alphabet $\mathcal{A}$ and a positive integer L.*
Question: *Is there a string $x \in \mathcal{A}^*$ of length at least L that is a subsequence of S_i for $1 \leq i \leq n$?*

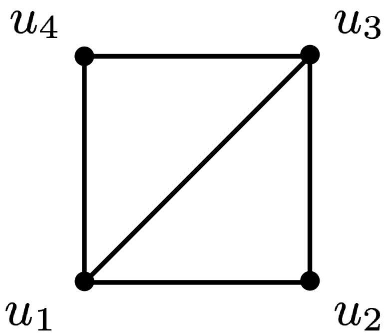
$$S_1 = AA\boxed{G}\boxed{G}G\boxed{A}TT\boxed{C}\boxed{A}\boxed{T}A\boxed{G}T$$

$$S_2 = ATATA\boxed{G}T\boxed{G}\boxed{A}AA\boxed{C}\boxed{A}\boxed{T}C\boxed{G}$$

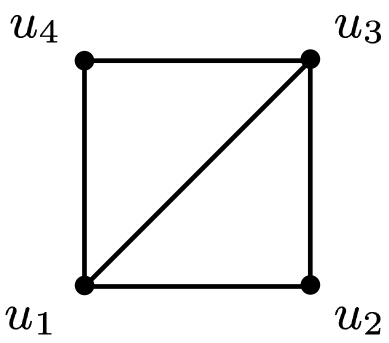
$$S_3 = \boxed{G}AA\boxed{G}CT\boxed{A}\boxed{C}\boxed{A}A\boxed{T}GA\boxed{G}CC$$

$$S_4 = A\boxed{G}\boxed{G}\boxed{A}CC\boxed{C}\boxed{A}A\boxed{T}\boxed{G}ACGG$$

LCS is NP-complete



LCS **is** NP-**complete**



Consider a vertex $u_i \in V$ and suppose that u_i is adjacent to vertices

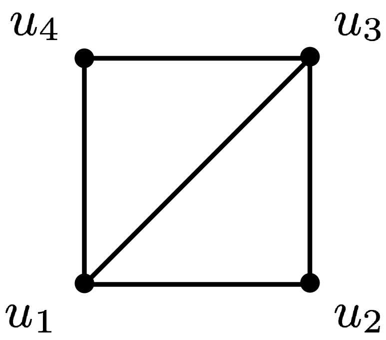
$$u_{i_1},\ u_{i_2},\ \ldots,\ u_{i_q} \qquad (i_1 < i_2 < \ldots < i_q)$$

Let p be the unique index such that $0 \le p \le q$ and $i_p < i < i_{p+1}$.

$$s_i = \overbrace{u_{i_1}\ u_{i_2}\ \ldots\ u_{i_p}}^{\text{adj. vertices that go } \textit{before}\ u_i} \qquad u_i \qquad \overbrace{u_1\ \ldots\ u_{i-1}\ u_{i+1}\ \ldots u_n}^{\text{all vertices but } u_i}$$

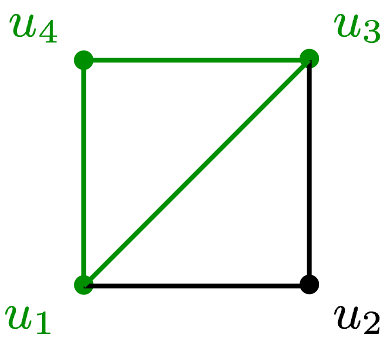
$$t_i = \underbrace{u_1\ \ldots\ u_{i-1}\ u_{i+1}\ \ldots u_n}_{\text{all vertices but } u_i} \qquad u_i \qquad \underbrace{u_{i_{p+1}}\ \ldots\ u_{i_{q-1}}\ u_{i_q}}_{\text{adj. vertices that go } \textit{after}\ u_i}$$

LCS is NP-complete



$$\begin{array}{llll}
s_1 = & & u_1 & u_2\ u_3\ u_4 \\
t_1 = & u_2\ u_3\ u_4 & u_1 & u_2\ u_3\ u_4 \\
s_2 = & u_1 & u_2 & u_1\ u_3\ u_4 \\
t_2 = & u_1\ u_3\ u_4 & u_2 & u_3 \\
s_3 = & u_1\ u_2 & u_3 & u_1\ u_2\ u_4 \\
t_3 = & u_1\ u_2\ u_4 & u_3 & u_4 \\
s_4 = & u_1\ u_3 & u_4 & u_1\ u_2\ u_3 \\
t_4 = & u_1\ u_2\ u_3 & u_4 &
\end{array}$$

LCS is NP-complete



$s_1 = u_1 \; u_2 \; u_3 \; u_4$

$t_1 = u_2 \; u_3 \; u_4 \; u_1 \; u_2 \; u_3 \; u_4$

$s_2 = u_1 \; u_2 \; u_1 \; u_3 \; u_4$

$t_2 = u_1 \; u_3 \; u_4 \; u_2 \; u_3$

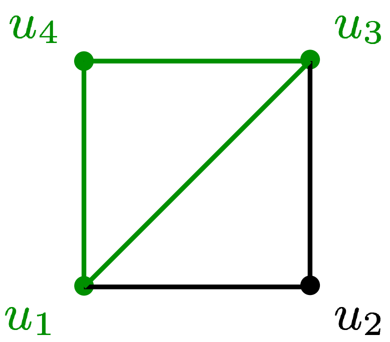
$s_3 = u_1 \; u_2 \; u_3 \; u_1 \; u_2 \; u_4$

$t_3 = u_1 \; u_2 \; u_4 \; u_3 \; u_4$

$s_4 = u_1 \; u_3 \; u_4 \; u_1 \; u_2 \; u_3$

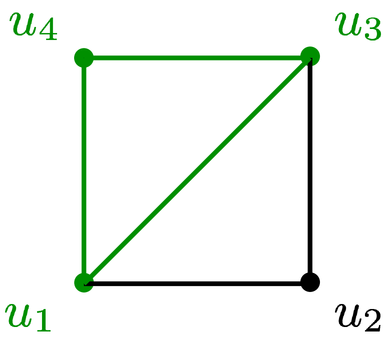
$t_4 = u_1 \; u_2 \; u_3 \; u_4$

LCS is NP-complete



$$\begin{array}{rlll}
s_1 = & & \boxed{u_1} & u_2\ \boxed{u_3}\ \boxed{u_4} \\
t_1 = & u_2\ u_3\ u_4 & u_1 & u_2\ u_3\ u_4 \\
s_2 = & u_1 & u_2 & u_1\ u_3\ u_4 \\
t_2 = & u_1\ u_3\ u_4 & u_2 & u_3 \\
s_3 = & u_1\ u_2 & u_3 & u_1\ u_2\ u_4 \\
t_3 = & u_1\ u_2\ u_4 & u_3 & u_4 \\
s_4 = & u_1\ u_3 & u_4 & u_1\ u_2\ u_3 \\
t_4 = & u_1\ u_2\ u_3 & u_4 &
\end{array}$$

LCS is NP-complete



$$s_1 = \boxed{u_1} \quad u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$t_1 = u_2 \; u_3 \; u_4 \quad \boxed{u_1} \quad u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$s_2 = u_1 \quad u_2 \quad u_1 \; u_3 \; u_4$$

$$t_2 = u_1 \; u_3 \; u_4 \quad u_2 \quad u_3$$

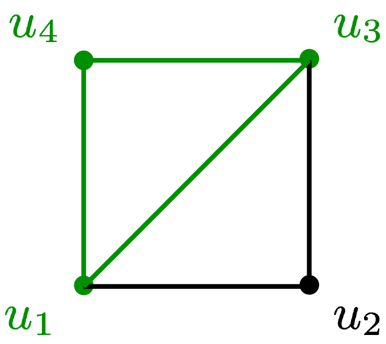
$$s_3 = u_1 \; u_2 \quad u_3 \quad u_1 \; u_2 \; u_4$$

$$t_3 = u_1 \; u_2 \; u_4 \quad u_3 \quad u_4$$

$$s_4 = u_1 \; u_3 \quad u_4 \quad u_1 \; u_2 \; u_3$$

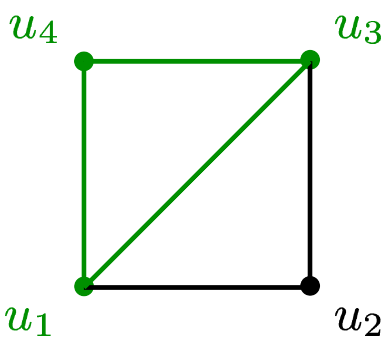
$$t_4 = u_1 \; u_2 \; u_3 \quad u_4$$

LCS is NP-complete



$$\begin{array}{llll}
s_1 = & & \boxed{u_1} & u_2\ \boxed{u_3}\ \boxed{u_4} \\
t_1 = & u_2\ u_3\ u_4 & \boxed{u_1} & u_2\ \boxed{u_3}\ \boxed{u_4} \\
s_2 = & \boxed{u_1} & u_2 & u_1\ \boxed{u_3}\ \boxed{u_4} \\
t_2 = & u_1\ u_3\ u_4 & u_2 & u_3 \\
s_3 = & u_1\ u_2 & u_3 & u_1\ u_2\ u_4 \\
t_3 = & u_1\ u_2\ u_4 & u_3 & u_4 \\
s_4 = & u_1\ u_3 & u_4 & u_1\ u_2\ u_3 \\
t_4 = & u_1\ u_2\ u_3 & u_4 &
\end{array}$$

LCS is NP-complete



$$s_1 = \qquad \boxed{u_1} \quad u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$t_1 = u_2 \; u_3 \; u_4 \qquad \boxed{u_1} \quad u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$s_2 = \boxed{u_1} \qquad u_2 \quad u_1 \; \boxed{u_3} \; \boxed{u_4}$$

$$t_2 = \boxed{u_1} \; \boxed{u_3} \; \boxed{u_4} \qquad u_2 \quad u_3$$

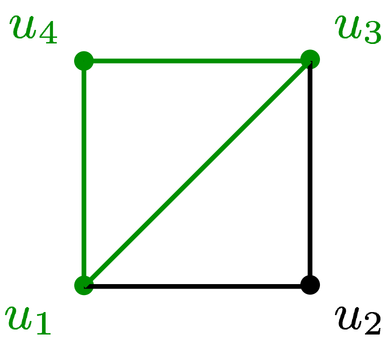
$$s_3 = u_1 \; u_2 \qquad u_3 \quad u_1 \; u_2 \; u_4$$

$$t_3 = u_1 \; u_2 \; u_4 \qquad u_3 \quad u_4$$

$$s_4 = u_1 \; u_3 \qquad u_4 \quad u_1 \; u_2 \; u_3$$

$$t_4 = u_1 \; u_2 \; u_3 \qquad u_4$$

LCS is NP-complete



$s_1 = \boxed{u_1} \; u_2 \; \boxed{u_3} \; \boxed{u_4}$

$t_1 = u_2 \; u_3 \; u_4 \; \boxed{u_1} \; u_2 \; \boxed{u_3} \; \boxed{u_4}$

$s_2 = \boxed{u_1} \; u_2 \; u_1 \; \boxed{u_3} \; \boxed{u_4}$

$t_2 = \boxed{u_1} \; \boxed{u_3} \; \boxed{u_4} \; u_2 \; u_3$

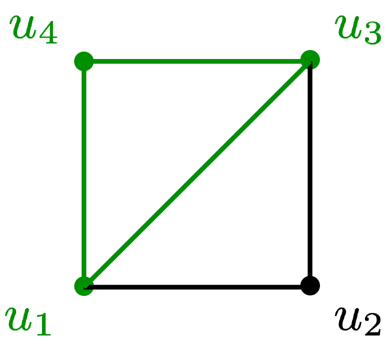
$s_3 = \boxed{u_1} \; u_2 \; \boxed{u_3} \; u_1 \; u_2 \; \boxed{u_4}$

$t_3 = u_1 \; u_2 \; u_4 \; u_3 \; u_4$

$s_4 = u_1 \; u_3 \; u_4 \; u_1 \; u_2 \; u_3$

$t_4 = u_1 \; u_2 \; u_3 \; u_4$

LCS is NP-complete



$$s_1 = \boxed{u_1} \; u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$t_1 = u_2 \; u_3 \; u_4 \; \boxed{u_1} \; u_2 \; \boxed{u_3} \; \boxed{u_4}$$

$$s_2 = \boxed{u_1} \; u_2 \; u_1 \; \boxed{u_3} \; \boxed{u_4}$$

$$t_2 = \boxed{u_1} \; \boxed{u_3} \; \boxed{u_4} \; u_2 \; u_3$$

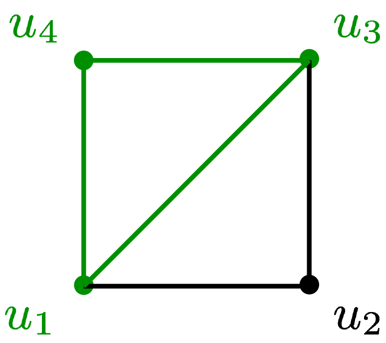
$$s_3 = \boxed{u_1} \; u_2 \; \boxed{u_3} \; u_1 \; u_2 \; \boxed{u_4}$$

$$t_3 = \boxed{u_1} \; u_2 \; u_4 \; \boxed{u_3} \; \boxed{u_4}$$

$$s_4 = u_1 \; u_3 \; u_4 \; u_1 \; u_2 \; u_3$$

$$t_4 = u_1 \; u_2 \; u_3 \; u_4$$

LCS is NP-complete



$s_1 = \boxed{u_1}\ u_2\ \boxed{u_3}\ \boxed{u_4}$

$t_1 = u_2\ u_3\ u_4\ \boxed{u_1}\ u_2\ \boxed{u_3}\ \boxed{u_4}$

$s_2 = \boxed{u_1}\ u_2\ u_1\ \boxed{u_3}\ \boxed{u_4}$

$t_2 = \boxed{u_1}\ \boxed{u_3}\ \boxed{u_4}\ u_2\ u_3$

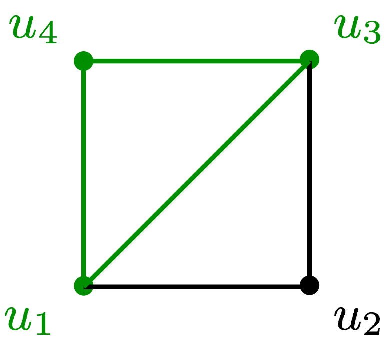
$s_3 = \boxed{u_1}\ u_2\ \boxed{u_3}\ u_1\ u_2\ \boxed{u_4}$

$t_3 = \boxed{u_1}\ u_2\ u_4\ \boxed{u_3}\ \boxed{u_4}$

$s_4 = \boxed{u_1}\ \boxed{u_3}\ \boxed{u_4}\ u_1\ u_2\ u_3$

$t_4 = u_1\ u_2\ u_3\ u_4$

LCS is NP-complete



$$\begin{aligned}
s_1 &= \quad \boxed{u_1} \quad u_2\ \boxed{u_3}\ \boxed{u_4} \\
t_1 &= u_2\ u_3\ u_4 \quad \boxed{u_1} \quad u_2\ \boxed{u_3}\ \boxed{u_4} \\
s_2 &= \boxed{u_1} \quad u_2 \quad u_1\ \boxed{u_3}\ \boxed{u_4} \\
t_2 &= \boxed{u_1}\ \boxed{u_3}\ \boxed{u_4} \quad u_2 \quad u_3 \\
s_3 &= \boxed{u_1}\ u_2 \quad \boxed{u_3} \quad u_1\ u_2\ \boxed{u_4} \\
t_3 &= \boxed{u_1}\ u_2\ u_4 \quad \boxed{u_3} \quad \boxed{u_4} \\
s_4 &= \boxed{u_1}\ \boxed{u_3} \quad \boxed{u_4} \quad u_1\ u_2\ u_3 \\
t_4 &= \boxed{u_1}\ u_2\ \boxed{u_3} \quad \boxed{u_4}
\end{aligned}$$

Complexity classes P and NP

The biggest open question in theoretical computer science concerns the relationship between those two classes:

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

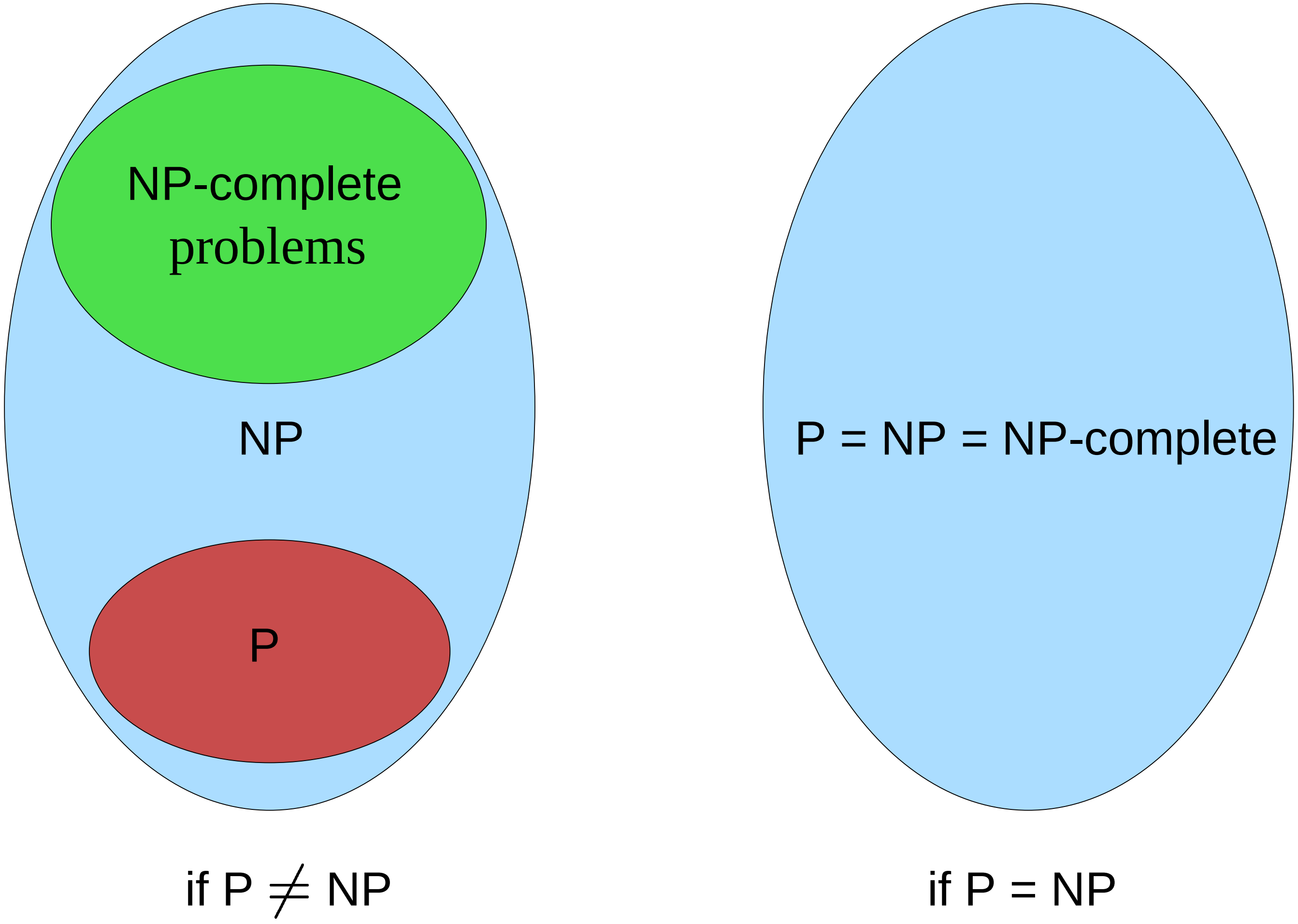
- most people think that the answer is probably "*no*";
- some people believe the question may be undecidable from the currently accepted axioms.

Consensus opnion: $\mathrm{P} \neq \mathrm{NP}$

In essence, the $\mathrm{P} = \mathrm{NP}$ question asks: if positive solutions to a *YES*/*NO* problem can be verified quickly, can the answers also be computed quickly?

NP **map**

The suggested map of the NP world:



Coping with NP-hardness

NP-hard optimization problems can not be efficiently solved in an exact way unless $\mathsf{P} = \mathsf{NP}$.

Classical methods to deal with intractable problems:

- *Approximation algorithms*
- *Probabilistic algorithms*
- *Special cases*
- *Exponential algorithms*
- *Local search*
- *Heuristics*

Approximation

If we want to solve an NP-hard optimization problem by means of an efficient (polynomial time) algorithm, we have to accept that the algorithm does not always return an optimal solution but rather an approximate one.

- Historically: MULTIPROCESSOR-SCHEDULING and BIN-PACKING
- Poor quality of approximation:
  - Lack in ability to design good approximation algorithm, or
  - Structural properties of the problem

Performance ratio

Given an optimization problem Π, for any instance x of Π and for any feasable solution y of x, the *performance ratio* of y with respect to x defined as

$$R(x,y) = \max\left(\frac{\mathsf{Algo}(x,y)}{\mathsf{opt}(x)}, \frac{\mathsf{opt}(x)}{\mathsf{Algo}(x,y)}\right)$$

Both in the case of minimization problems and of maximization problems, the value of the performance ratio $R(x,y)$ is

- equal to 1 in the case of an optimal solution;
- arbitrary large in case of poor approximate solution.

r-Approximate algorithm

Given an optimization problem Π and an approximation algorithm Algo for Π, we say that Algo is an r-approximate algorithm for Π if, given any input instance x of Π, the performance ratio of the approximate solution Algo(x) is bounded by r, that is:

$$R(x, \text{Algo}(x)) \leq r$$

- A 1-approximate algorithm is an exact algorithm
- Any desired performance ratio can be obtained ?

NPO

NPO: *Optimization*

The class NPO is the set of all NP optimization problems

- The goal of an NPO problem is to find an *optimum solution*
- *minimization* or *maximization*
- feasible solutions are *short* and *easy to recognize*

APX

APX: *Approximable*

The subclass of NPO problems that admit constant-factor approximation algorithms. (I.e., there is a polynomial-time algorithm that is guaranteed to find a solution within a constant factor of the optimum cost.)

⚠ Equals the closure of MaxSNP and of MaxNP under PTAS reduction.

- SHORTEST-COMMON-SUPERSTRING
- VERTEX-COVER for $\Delta \geq 3$
- MAX-CUT for $\Delta \geq 3$

PTAS

PTAS: *Polynomial-Time Approximation Scheme*

The subclass of NPO problems that admit an approximation scheme in the following sense. For any $\varepsilon > 0$, there is a polynomial-time algorithm that is guaranteed to find a solution whose cost is within a $1 + \varepsilon$ factor of the optimum cost. (However, the exponent of the polynomial might depend strongly on ε.)

- NEAREST-STRING
- TRAVELING-SALESMAN in the Euclidean plane.
- MAX-INDEPENDENT-SET for planar graphs

FPTAS

FPTAS: *Fully Polynomial-Time Approximation Scheme*

The subclass of NPO problems that admit an approximation scheme in the following sense. For any $\varepsilon > 0$, there is an algorithm that is guaranteed to find a solution whose cost is within a $1 + \varepsilon$ factor of the optimum cost. Furthermore, the running time of the algorithm is polynomial in n (the size of the problem) and in $1/\varepsilon$.

- MAXIMUM-INTEGER-K-CHOICE-KNAPSACK
- MINIMUM-MULTIPROCESSOR-SCHEDULING for constant number of processor

Inclusions

$$\mathsf{FPTAS} \subseteq \mathsf{PTAS} \subseteq \mathsf{APX} \subseteq \mathsf{NPO}$$

- These inclusions are strict if and only if P $\neq$ NP.

Between APX and NPO

$$\mathsf{PTAS} \subseteq \mathsf{APX} \subseteq \mathsf{log-APX} \subseteq \mathsf{poly-APX} \subseteq \mathsf{exp-APX} \subseteq \mathsf{NPO}$$

⚠ If P $\neq$ NP then exp-APX is strictly contained in NPO.

- SHORTEST-COMMON-SUPERSEQUENCE
- LONGEST-PATH

Using the compendium

http://www.nada.kth.se/~viggo/problemlist/compendium.html

LONGEST COMMON SUBSEQUENCE

- INSTANCE: Finite alphabet Σ, finite set R of strings from Σ^*.
- SOLUTION: A string $w \in \Sigma^*$ such that w is a subsequence of each $x \in R$, i.e. one can get w by taking away letters from each x.
- MEASURE: Length of the subsequence, i.e., $|w|$.
- GOOD NEWS: Approximable within $O(m \log m)$, where m is the length of the shortest string in R [223].
- BAD NEWS: Not approximable within $n^{1/4-\varepsilon}$ for any $\varepsilon > 0$, where n is the maximum of $|R|$ and $|\Sigma|$ [77], [275] and [68].
- COMMENT: Transformation from MAX-INDEPENDENT-SET. APX-complete if the size of the alphabet Σ is fixed [275] and [89]. Variation in which the objective is to find the shortest maximal common subsequence (a subsequence that cannot be extended to a longer common subsequence) is not approximable within $|R|^{1-\varepsilon}$ for any $\varepsilon > 0$ [170].
- Garey and Johnson: SR10

Approximate the mult alignment problem

1. Find $S_1 \in T$ that minimizes

$$\sum_{S \in T - S_1} D(S_1, S)$$

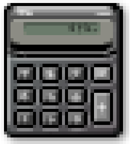
2. Add the remaining strings $S_2, \ldots, S_k$ one at a time to a multiple alignment that initially contains only S_1
    - Suppose $S_1, S_2, \ldots S_{i-1}$ has already aligned as $S'_1, S'_2, \ldots S'_{i-1}$
    - Align S'_i and S_i to produce S''_1 and S'_1.
    - Adjust $S'_2, \ldots, S'_{i-1}$ by adding spaces to these columns where spaces were added to get S''_1 from S'_1
    - Replace $S'1$ by S''_1

Analysis of the center star algorithm

TIME ANALYSIS
The preceding algorithm runs in $\mathcal{O}(k^2m^2)$ time where k is the number of sequences and m is the maximum length.

ERROR ANALYSIS
The preceding algorithm produces an alignment whose SP value alignment is less than twice that of the optimal SP value alignment.

$$\frac{\mathsf{ALGO}(x)}{\mathsf{opt}(x)} \leq \frac{2(k-1)}{k} < 2$$



For small value of k the approximation is significantly better than a factor of 2.

2	4	6	8	10	12	14	16	18	20	30	40
1	1.5	1.66	1.75	1.8	1.83	1.85	1.87	1.88	1.9	1.93	1.95

Parameterized complexity

- To deal with problems that are NP-hard or worse.
- Solutions produced by approximation algorithms or heuristics are in many cases not satisfying in practice.
- Exact algorithms often yield more expressive results.

Restrict the *combinatorial explosion* of NP-hard problems to a part of the input, the *parameter*.

⚠ A central area is computational biology where we encounter many examples of fixed-parameter algorithms.

Fixed-parameter tractability (FPT)

A problem is *fixed-parameter tractable* (in FPT) w.r.t parameter k when it has an algorithm with running time

$$f(k)\,p(n)$$

where f is an arbitrary function in k and p is a polynomial in the input size n.

- f is exponential or worse.
- p may be even linear.
- design of efficient algorithms.

Example

The NP-complete VERTEX-COVER problem is to determine whether there is a subset of vertices $V' \in V$ with k or fewer vertices (the parameter) such that each edge in E has at least one of its endpoints in V'.

The VERTEX-COVER problem is *fixed-parameter tractable* (in FPT).

- Solvable in time $\mathcal{O}(kn + 1.3248^k k^2)$.
- Efficiently solvable for small values of k.

Fixed-parameter intractability

Some problems appear to be *fixed-parameter intractable*.

- It is not known whether the CLIQUE problem can be solved in time $f(k)\,p(n)$ where f might be an arbitrary fast growing function only depending on k.
- Unless P $=$ NP the well-founded conjecture is that no such algorithm exist.
- The best known algorithm soling the CLIQUE problem runs in time $\mathcal{O}(n^{ck/3})$ where c is the exponent on the time bound for multiplying two integer $n \times n$ matrices.

W hierarchy

Hierarchy of W classes

$$\text{FPT} \subseteq \text{W}[1] \subseteq \text{W}[2] \subseteq \ldots \text{W}[\text{P}]$$

- Many parameterized problems have been proved to be complete for W[1] and W[2]
  - VERTEX-COVER is in FPT
  - INDEPENDENT-SET is complete for W[1]
  - DOMINATING-SET is complete for W[2]
- The class W[P] contains the parameterized problems that reduce to the family of decision circuit of any weft and depth
- While no W[t] class with $t \geq 3$ seems to be well populated, there are several W[P]-complete problems.

LONGEST-COMMON-SUBSEQUENCE

n : number of sequences

L : size of the longest common subsequence

parameter	unbounded alph.	parameterized alph.	fixed alph.
n	W[t]-hard for $t \geq 1$	W[t]-hard for $t \geq 1$	W[1]-complete
L	W[2]-hard	FPT	FPT
n et L	W[1]-complete	FPT	FPT

Questions . . .

- Why bother proving a problem to be NP-hard?

Questions . . .

- Why bother proving a problem to be NP-hard?
- When to attempt such a proof ?

Questions . . .

- Why bother proving a problem to be NP-hard?
- When to attempt such a proof ?
- What are the steps in proving a problem to be NP-complete?

Questions . . .

- Why bother proving a problem to be NP-hard?
- When to attempt such a proof ?
- What are the steps in proving a problem to be NP-complete?
- Does there exist any short-cut for proving NP-hardness?

Questions . . .

- Why bother proving a problem to be NP-hard?
- When to attempt such a proof ?
- What are the steps in proving a problem to be NP-complete?
- Does there exist any short-cut for proving NP-hardness?
- How to deal with a problem once it is proved to be NP-hard?

Questions . . .

- Why bother proving a problem to be NP-hard?
- When to attempt such a proof ?
- What are the steps in proving a problem to be NP-complete?
- Does there exist any short-cut for proving NP-hardness?
- How to deal with a problem once it is proved to be NP-hard?
- Do I need parameterized complexity ?