COMP-330
Theory of Computation

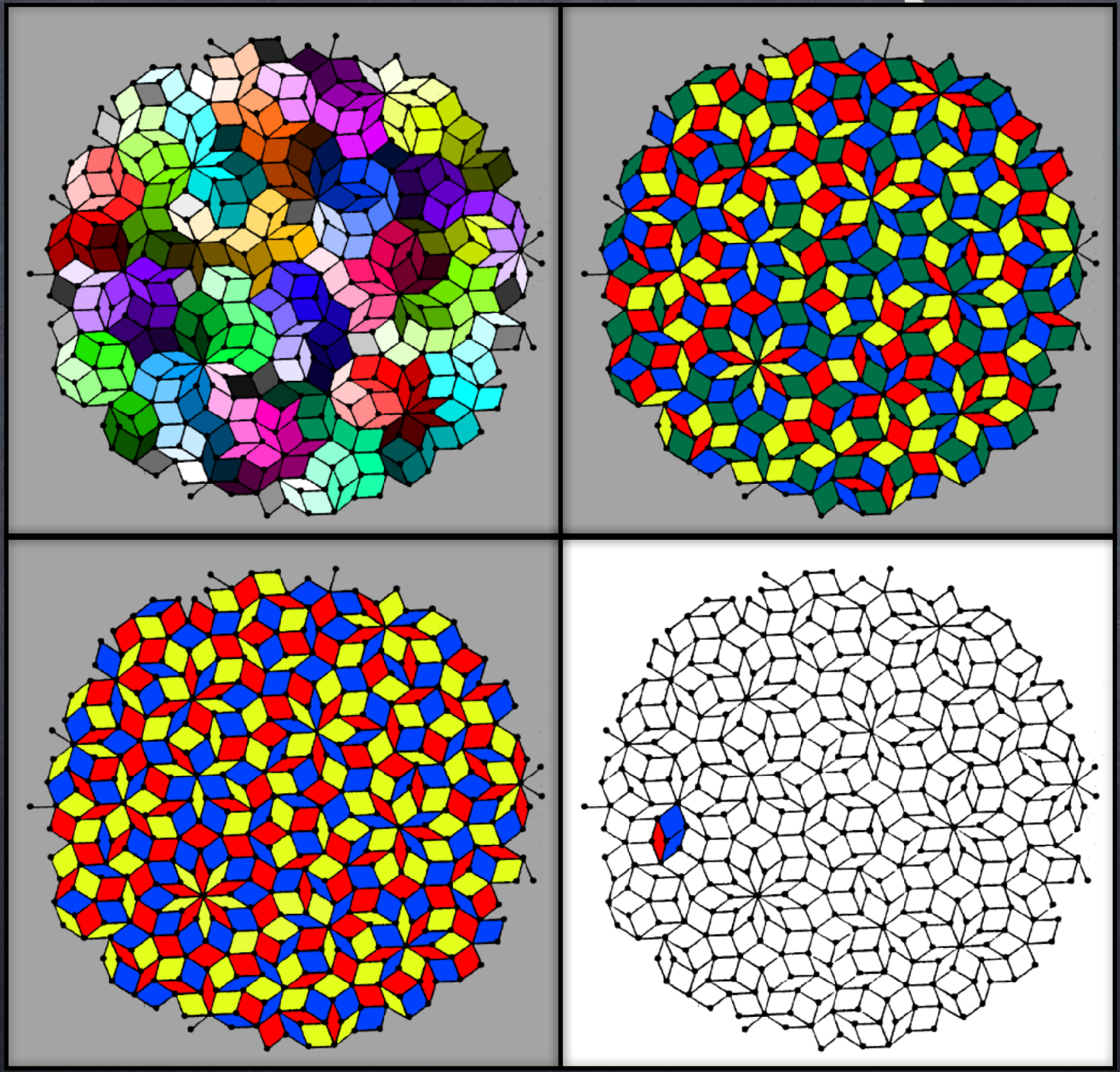
Fall 2019 -- Prof. Claude Crépeau

Lec. 22-23 :
Introduction to Complexity

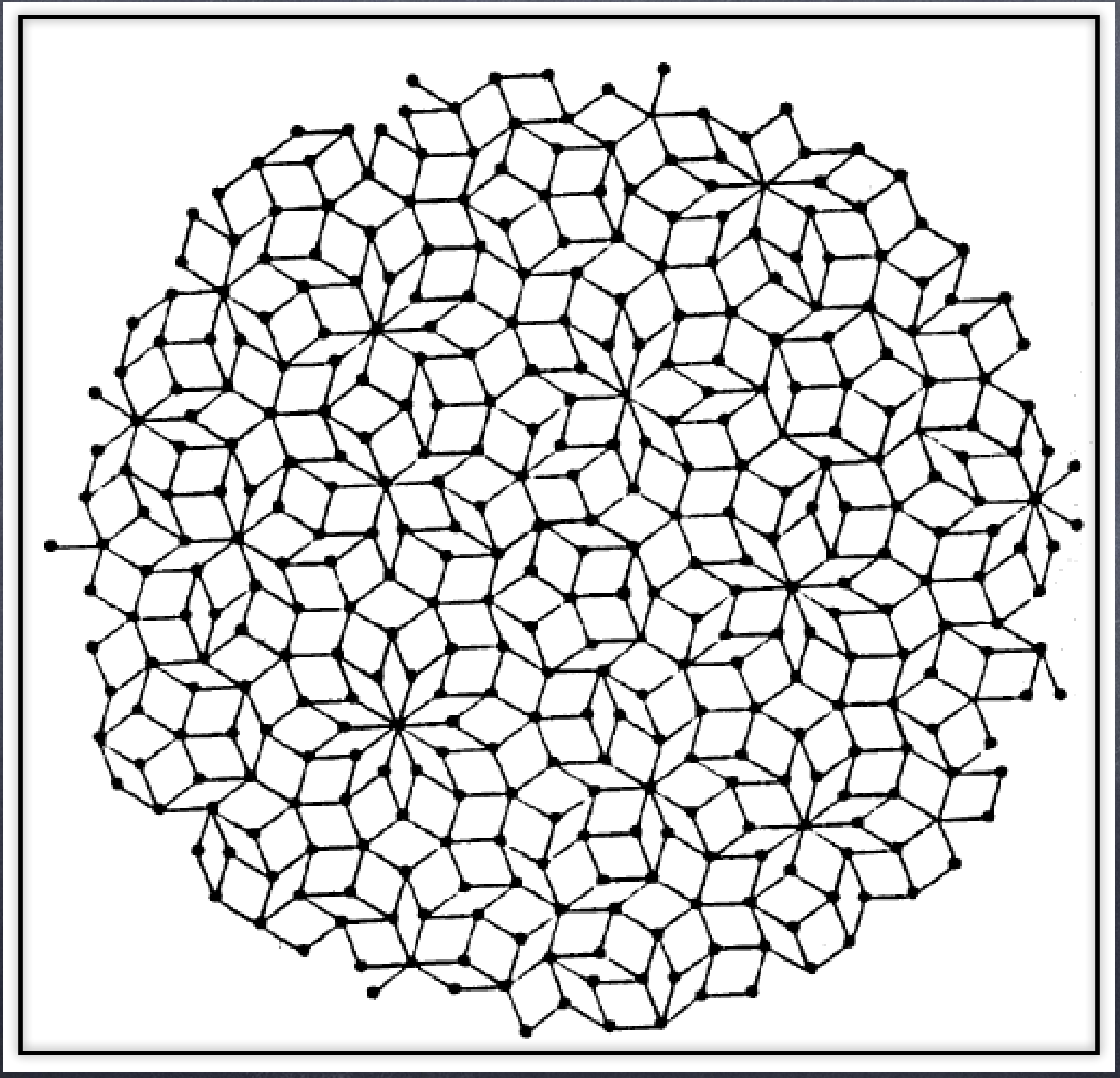
Tractable Problems

Not all problems were born equal...

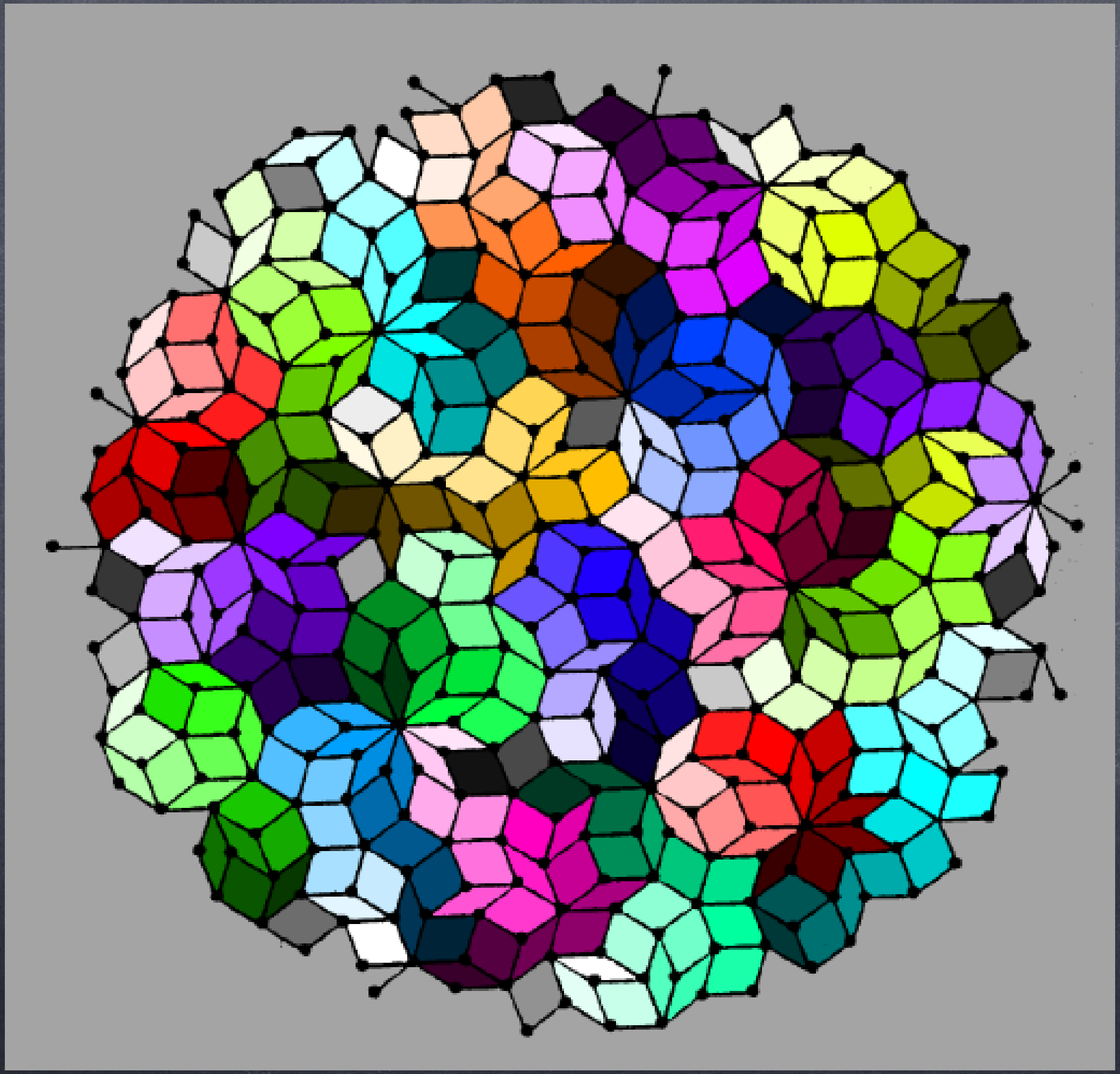
Not all problems were born equal...



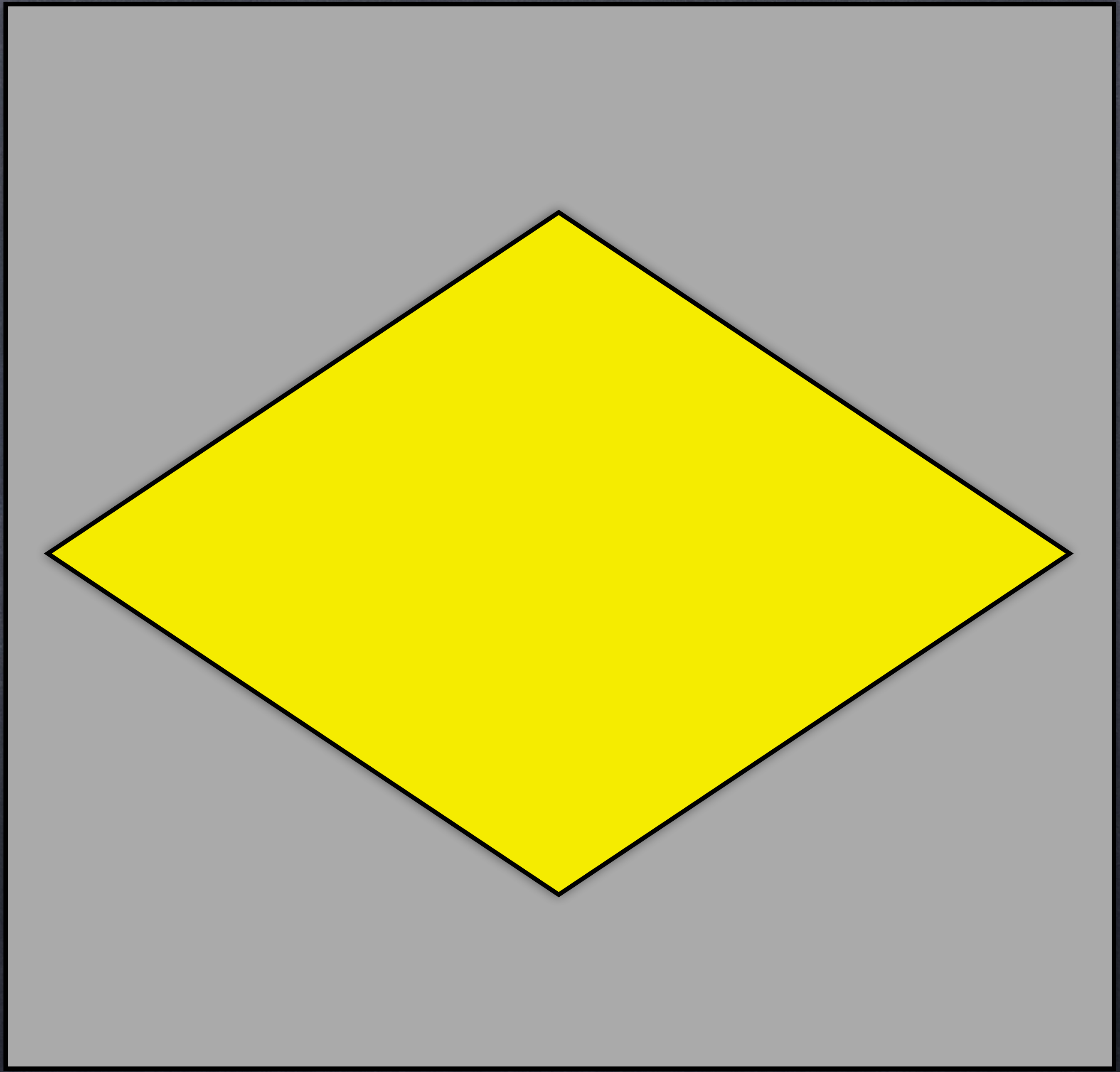
Is it possible to paint a colour on each region of a map so that no neighbours are of the same colour ?



Obviously, yes, if you can use as many colours as you like...

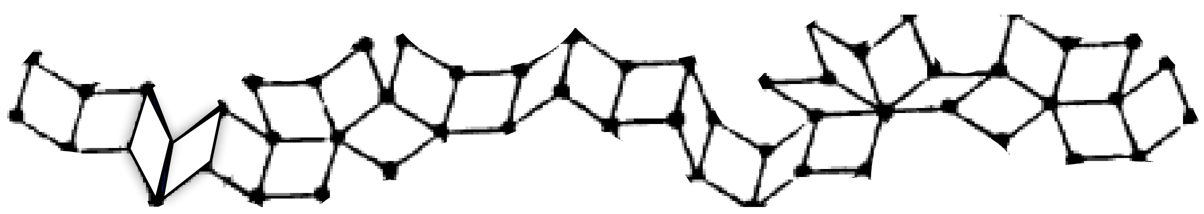


1-colouring problem



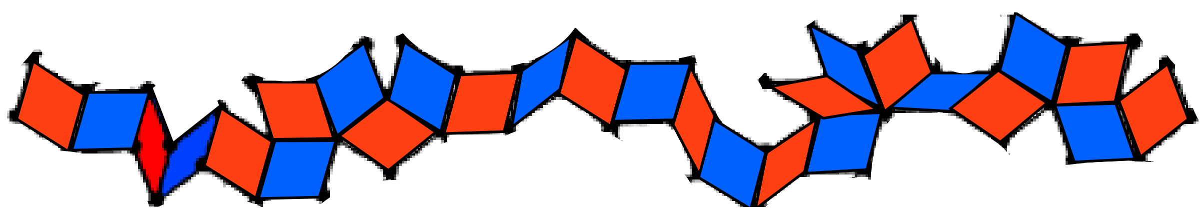
Only two maps are 1-colourable.

2-colouring problem



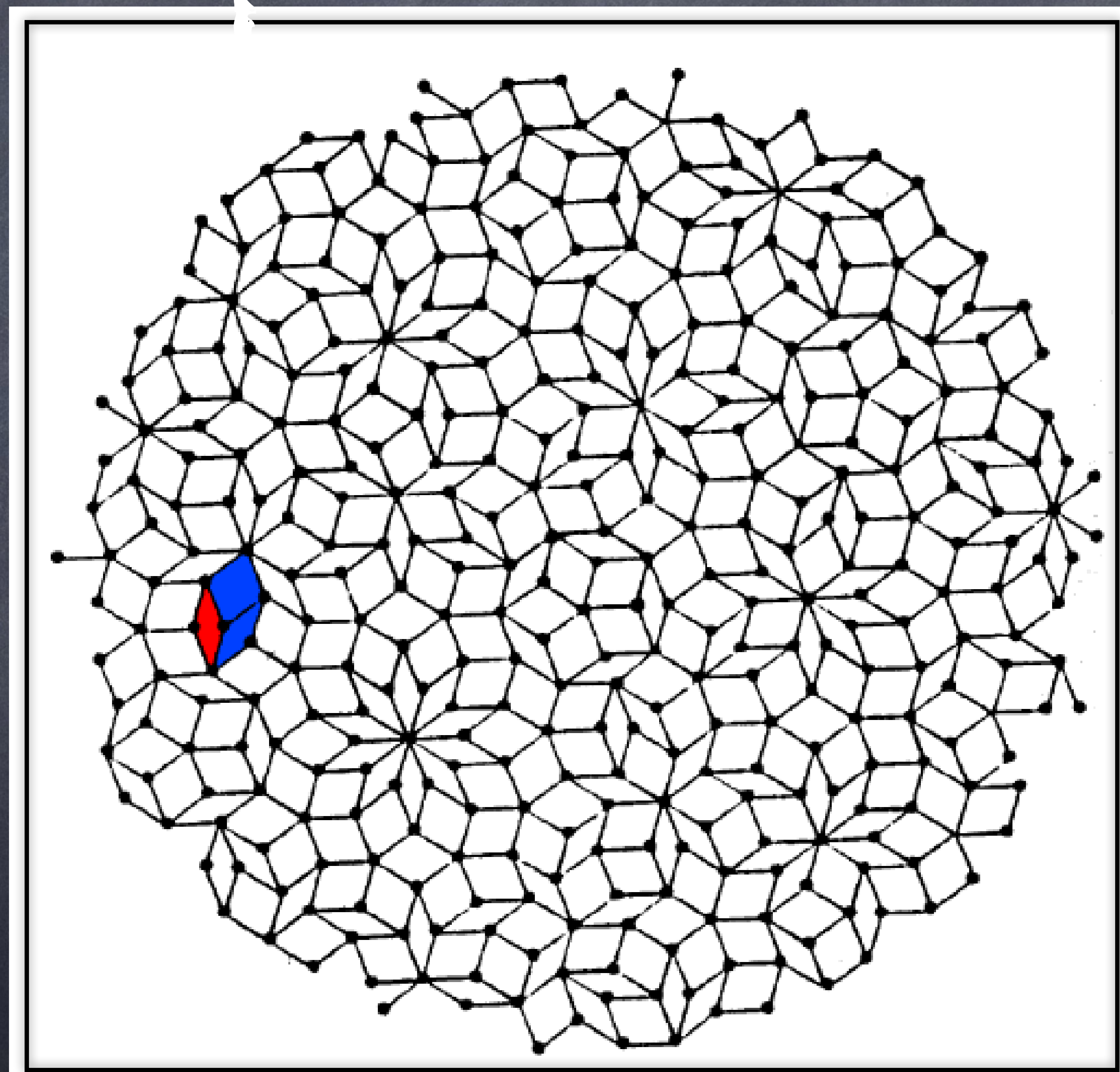
Very few maps are 2-colourable.

2-colouring problem



Very few maps are 2-colourable.

2-colouring problem



Most maps are not 2-colourable.

Tractable Problems (P)

Tractable Problems (P)

- 2-colorability of maps.

Tractable Problems (P)

- 2-colorability of maps.
- Primality testing.
  (but probably not factoring)

Tractable Problems (P)

- 2-colorability of maps.
- Primality testing.
  (but probably not factoring)
- Solving NxNxN Rubik's cube.

Tractable Problems (P)

- 2-colorability of maps.
- Primality testing.
  (but probably not factoring)
- Solving NxNxN Rubik's cube.
- Finding a word in a dictionary.

Tractable Problems (P)

- 2-colorability of maps.
- Primality testing.
  (but probably not factoring)
- Solving NxNxN Rubik's cube.
- Finding a word in a dictionary.
- Sorting elements...

Tractable Problems (P)

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.
- More formally, there exists a TM to compute solutions to the problem and there exists a polynomial Q such that the number of steps on each input x before halting is no more than Q(|x|).

Tractable Problems (P)

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.
- Computer Science studies mostly techniques to approach and find efficient solutions to tractable problems.

Tractable Problems (P)

- Fortunately, many practical problems are tractable. The name P stands for Polynomial-Time computable.
- Computer Science studies mostly techniques to approach and find efficient solutions to tractable problems.
- Some problems may be efficiently solvable but we might not be able to prove that...

Tractable Problems (P)

Tractable Problems (P)

- The name P stands for Polynomial-Time computable.

Tractable Problems (P)

- The name P stands for Polynomial-Time computable.
- Q: Why choose this level of granularity ? Why not choose linear-time for instance ?

Tractable Problems (P)

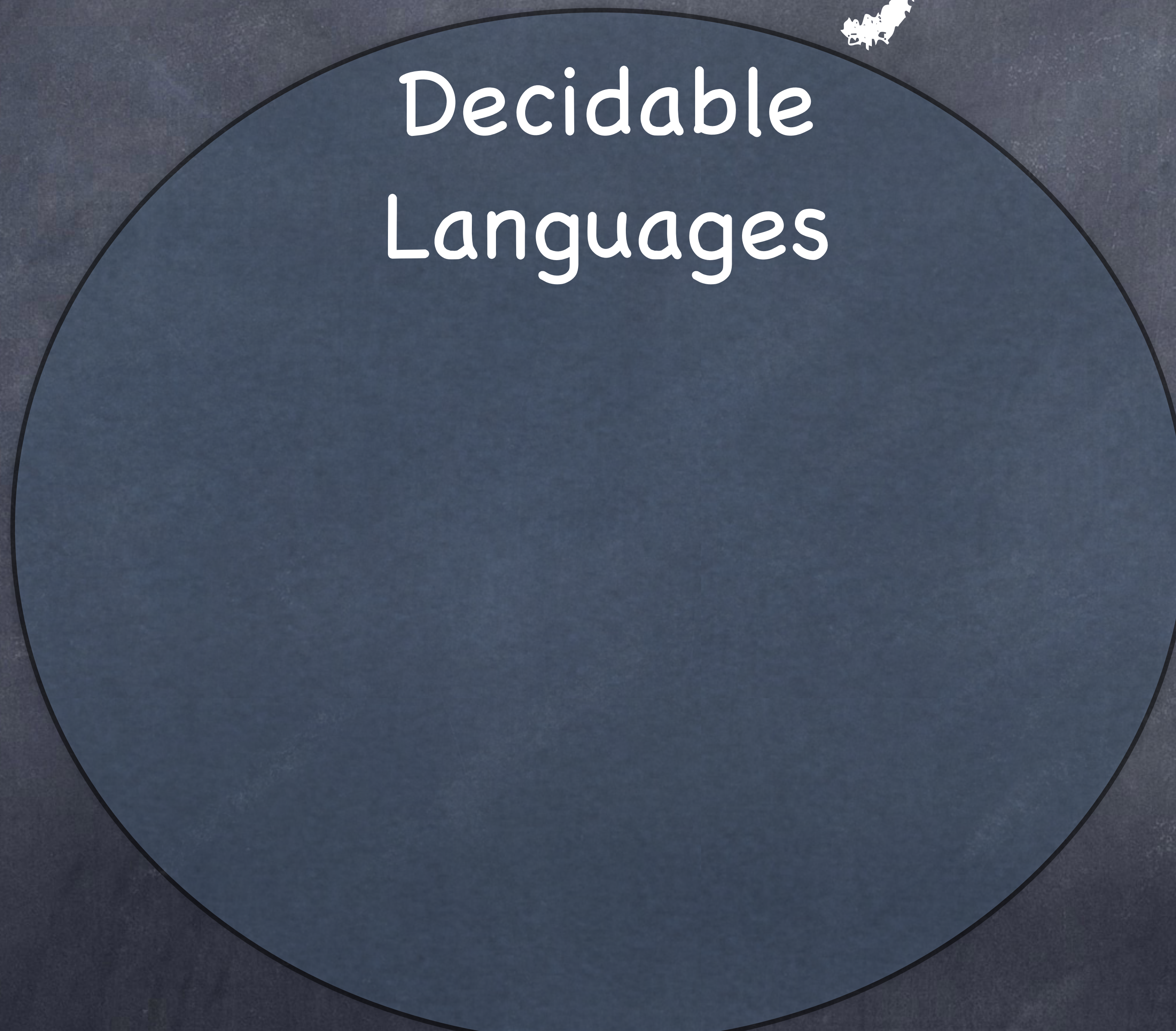
- The name P stands for Polynomial-Time computable.
- Q: Why choose this level of granularity ? Why not choose linear-time for instance ?
- A: because P is the same for all types of Turing machines and any reasonable model. This is not true of linear-time for instance...

Tractable Problems (P)

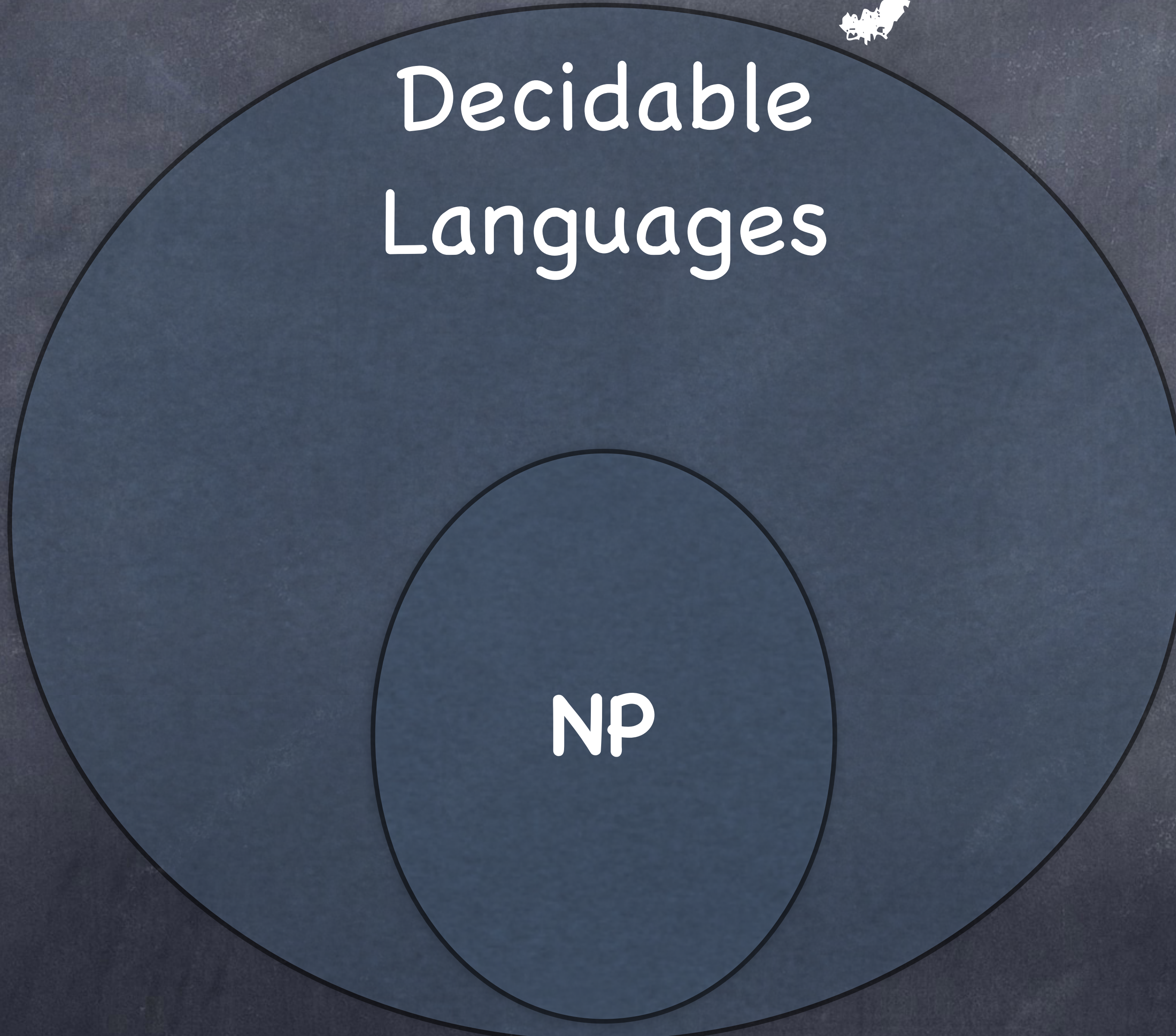
THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

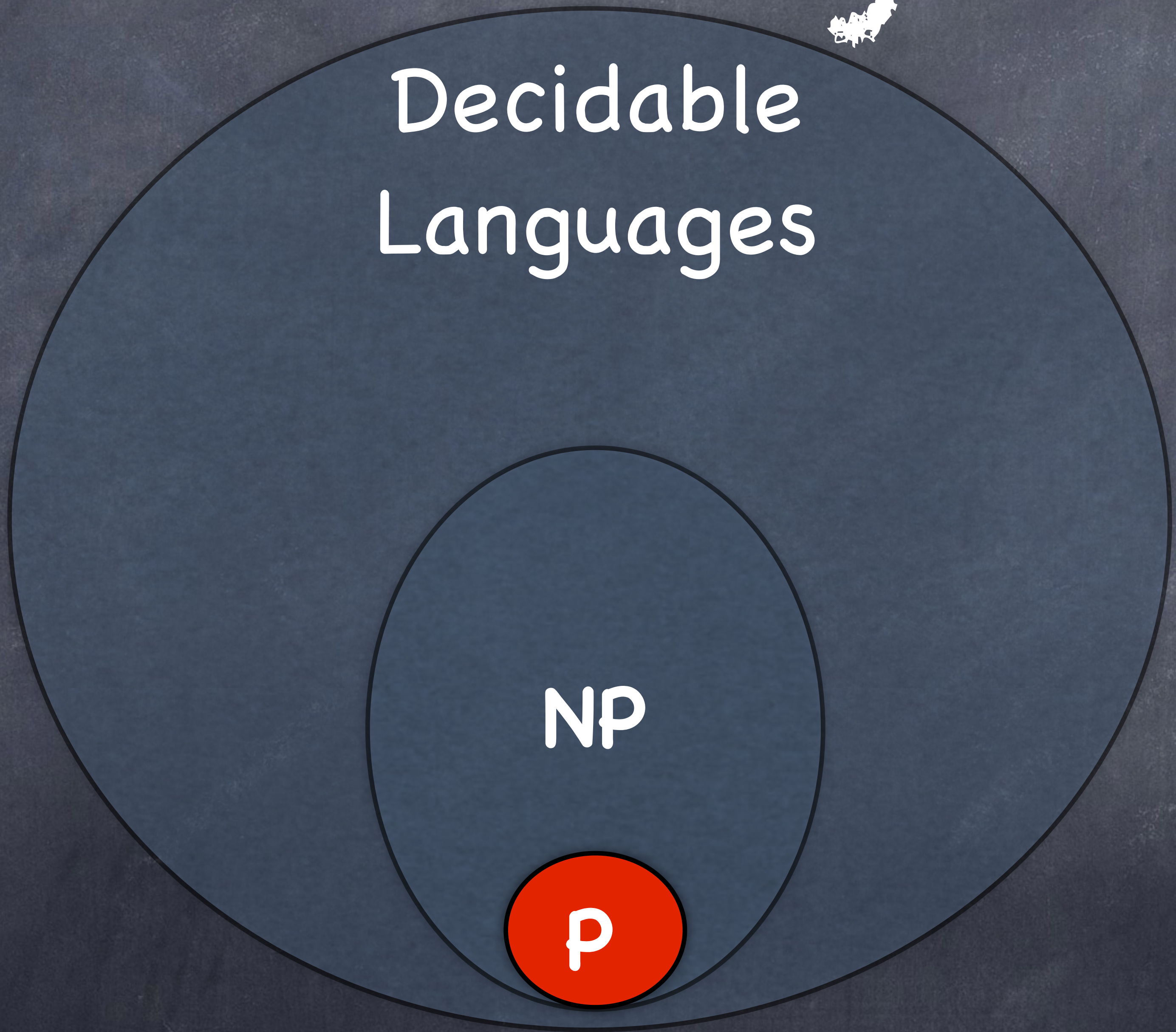
Complexity Theory



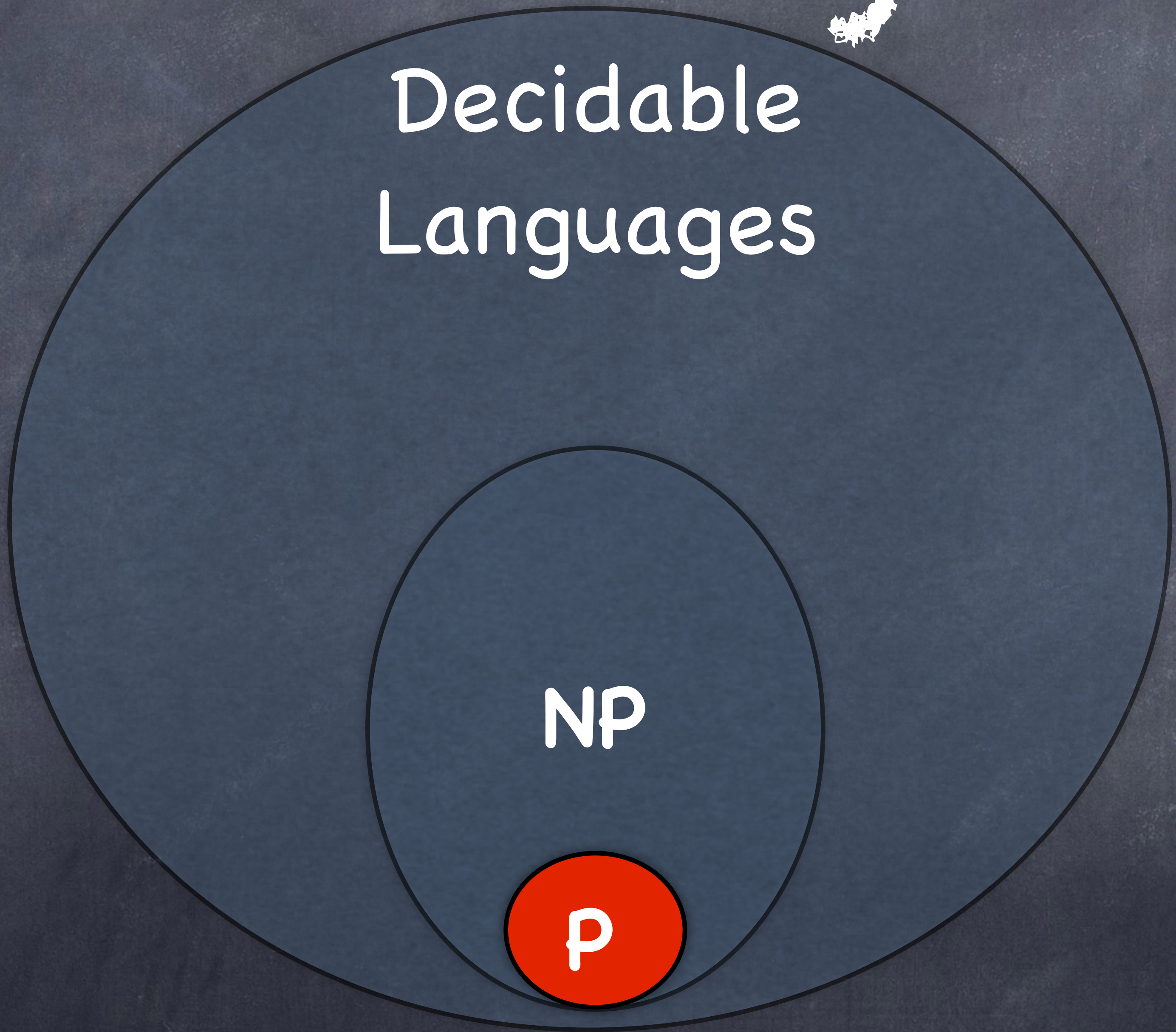
Complexity Theory



Complexity Theory

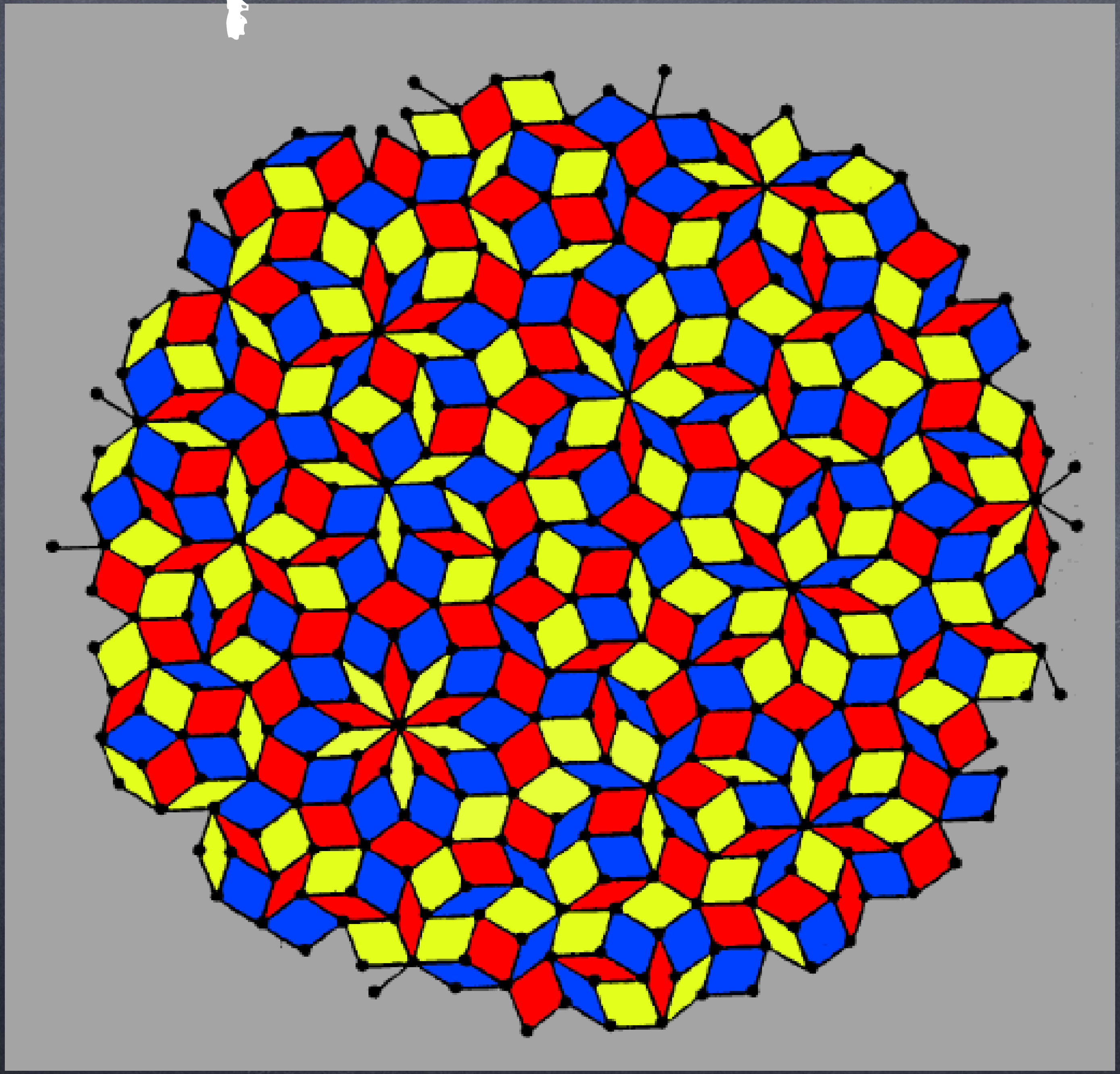


Complexity Theory



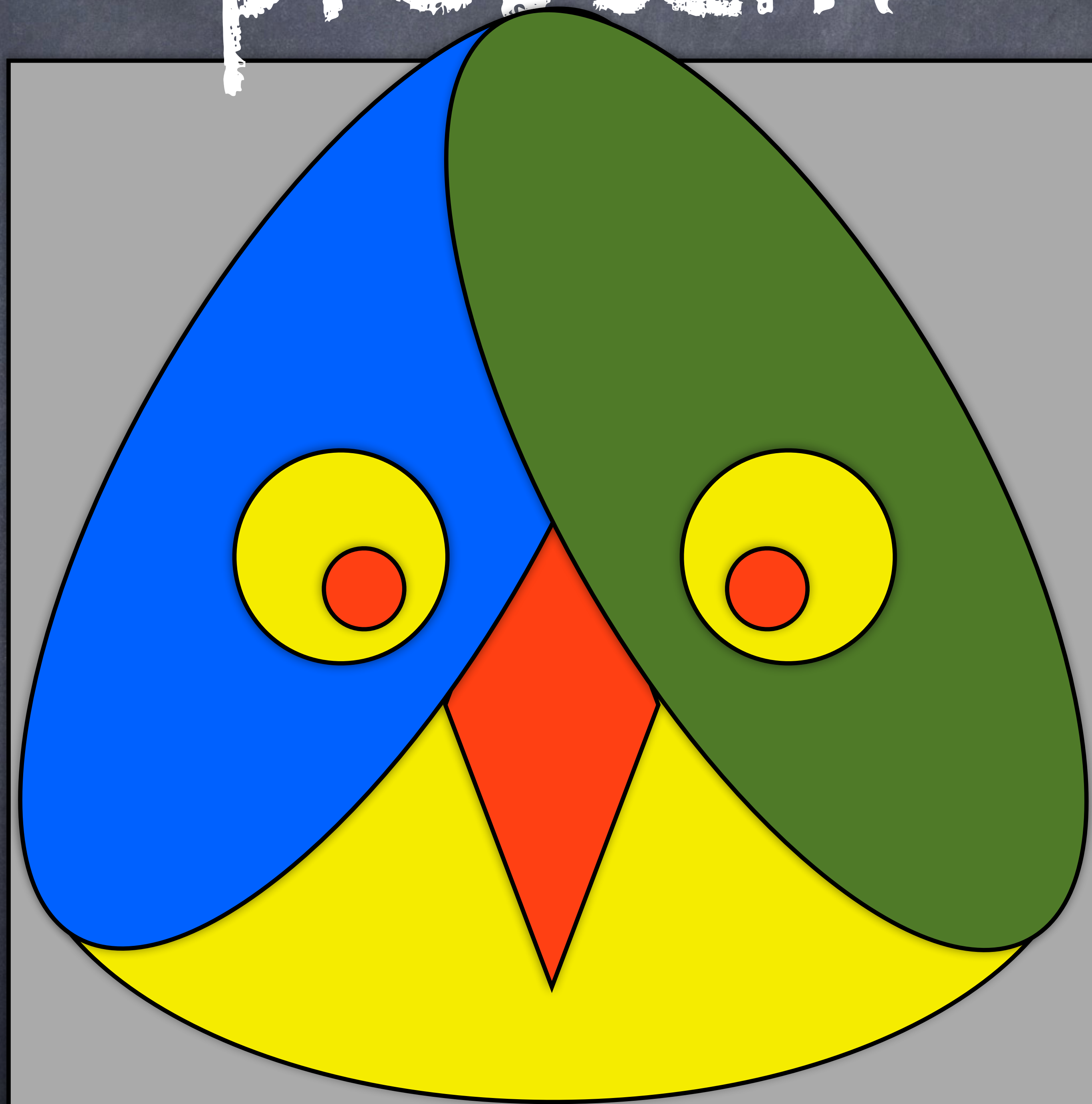
P = NP ?

3-colouring problem



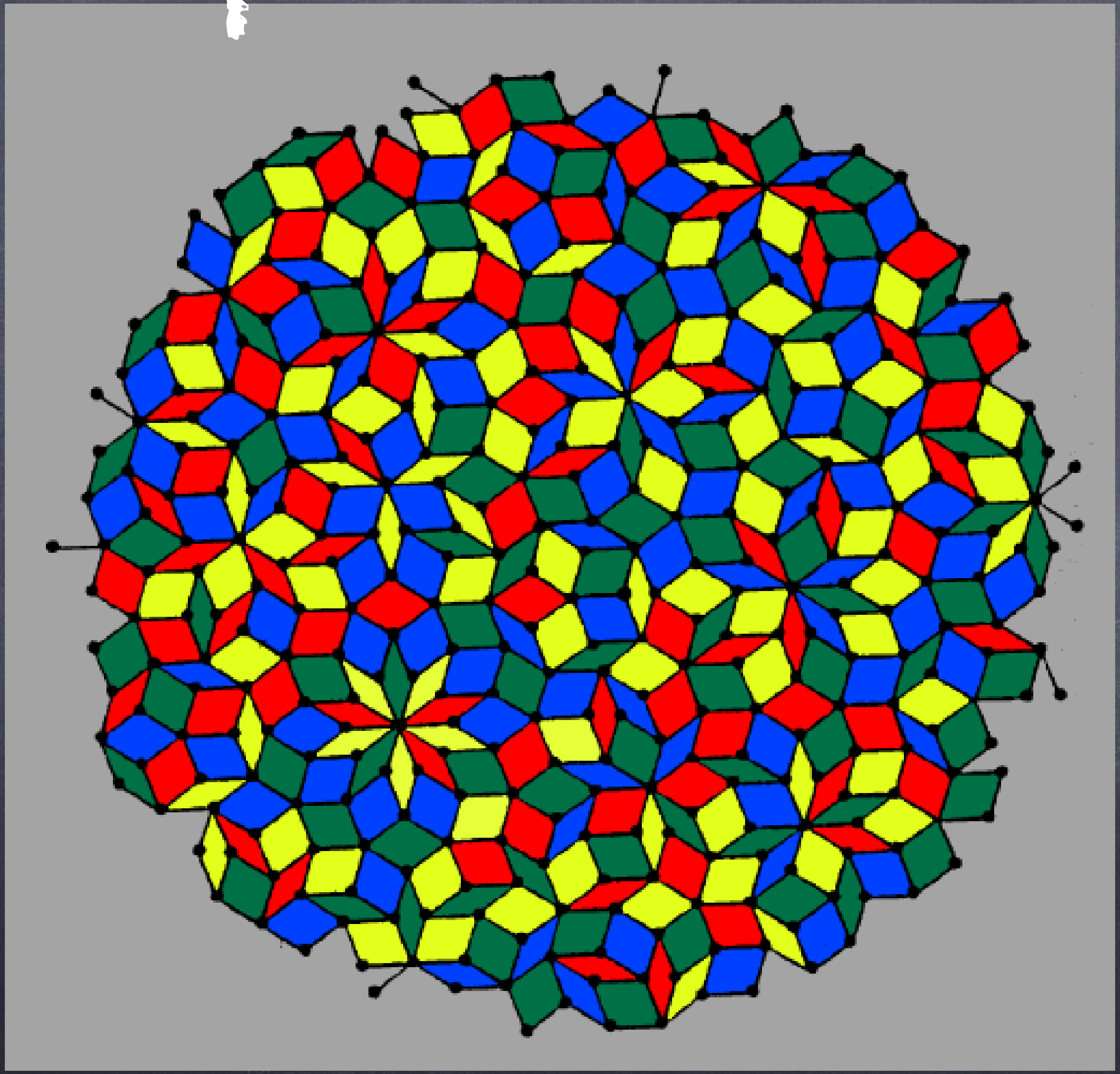
Some maps are 3-colourable.

3-colouring problem



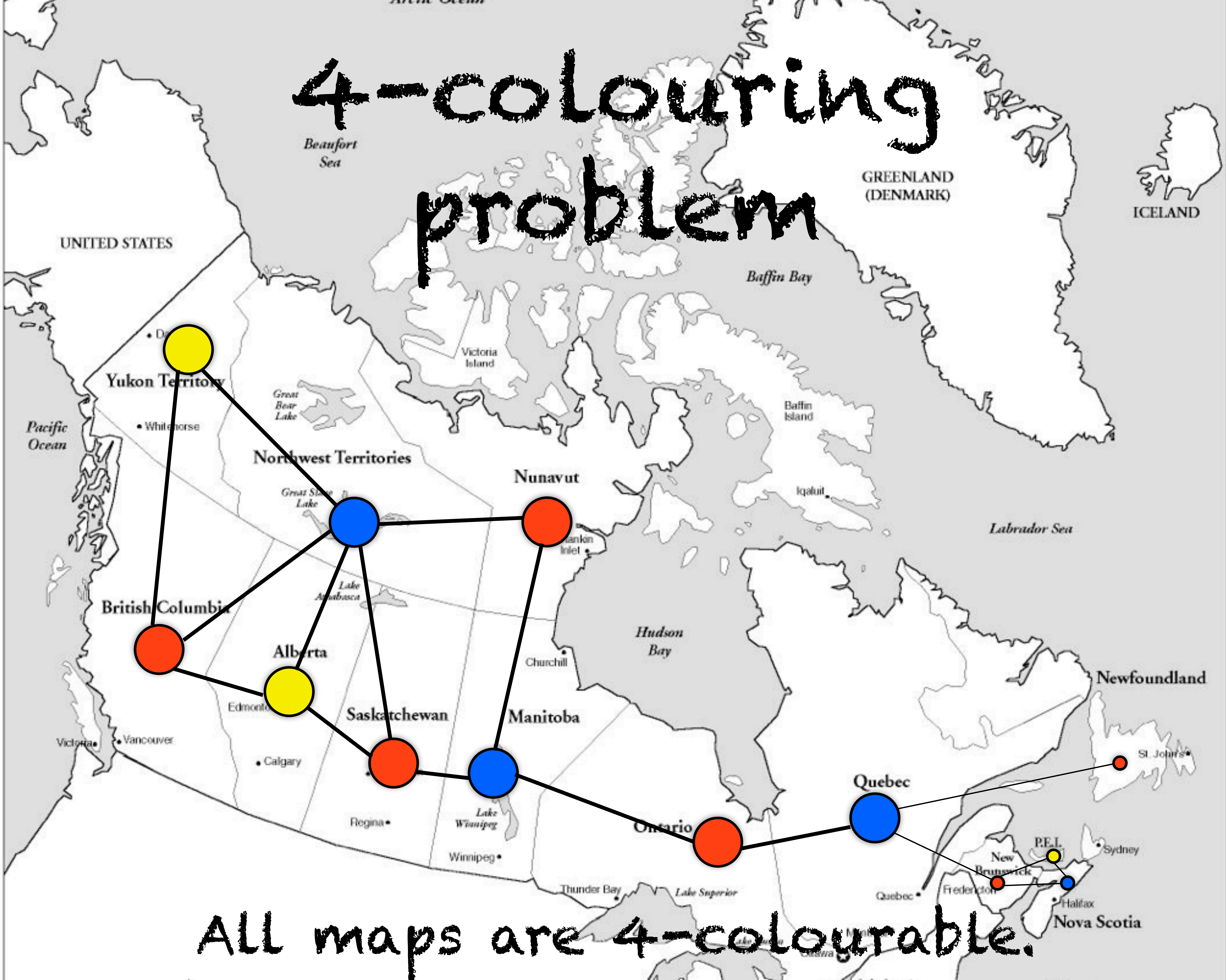
Some maps are not 3-colourable.

4-colouring problem



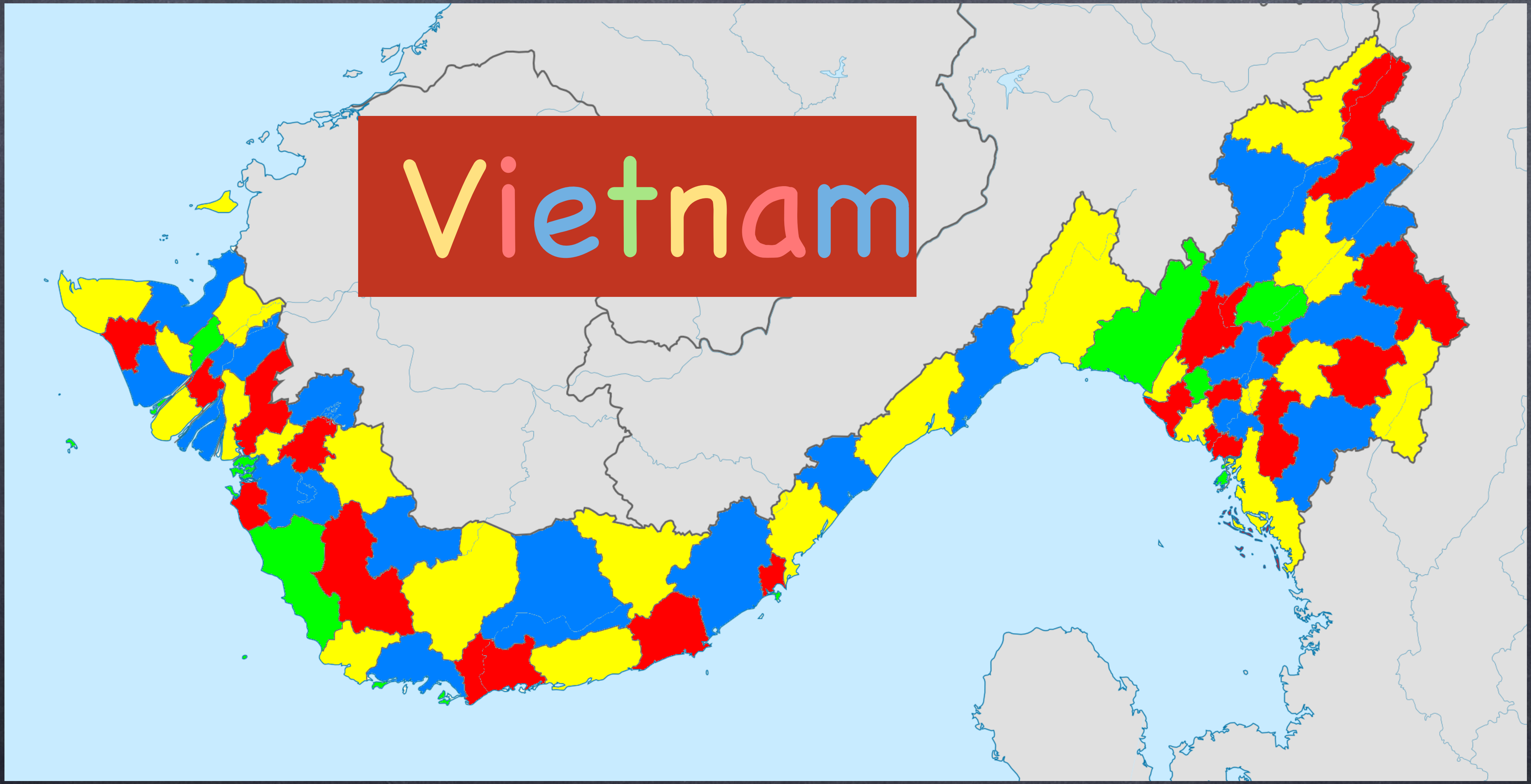
All maps are 4-colourable.

4-colouring problem



All maps are 4-colourable.

4-colouring problem



All maps are 4-colourable.

K-colouring of Maps (planar graphs)

K-colouring of Maps (planar graphs)

- K=1 only the maps with zero or one region are 1-colourable.

K-colouring of Maps (planar graphs)

- K=1 only the maps with zero or one region are 1-colourable.
- K=2 easy to decide. Impossible as soon as 3 regions touch each other.

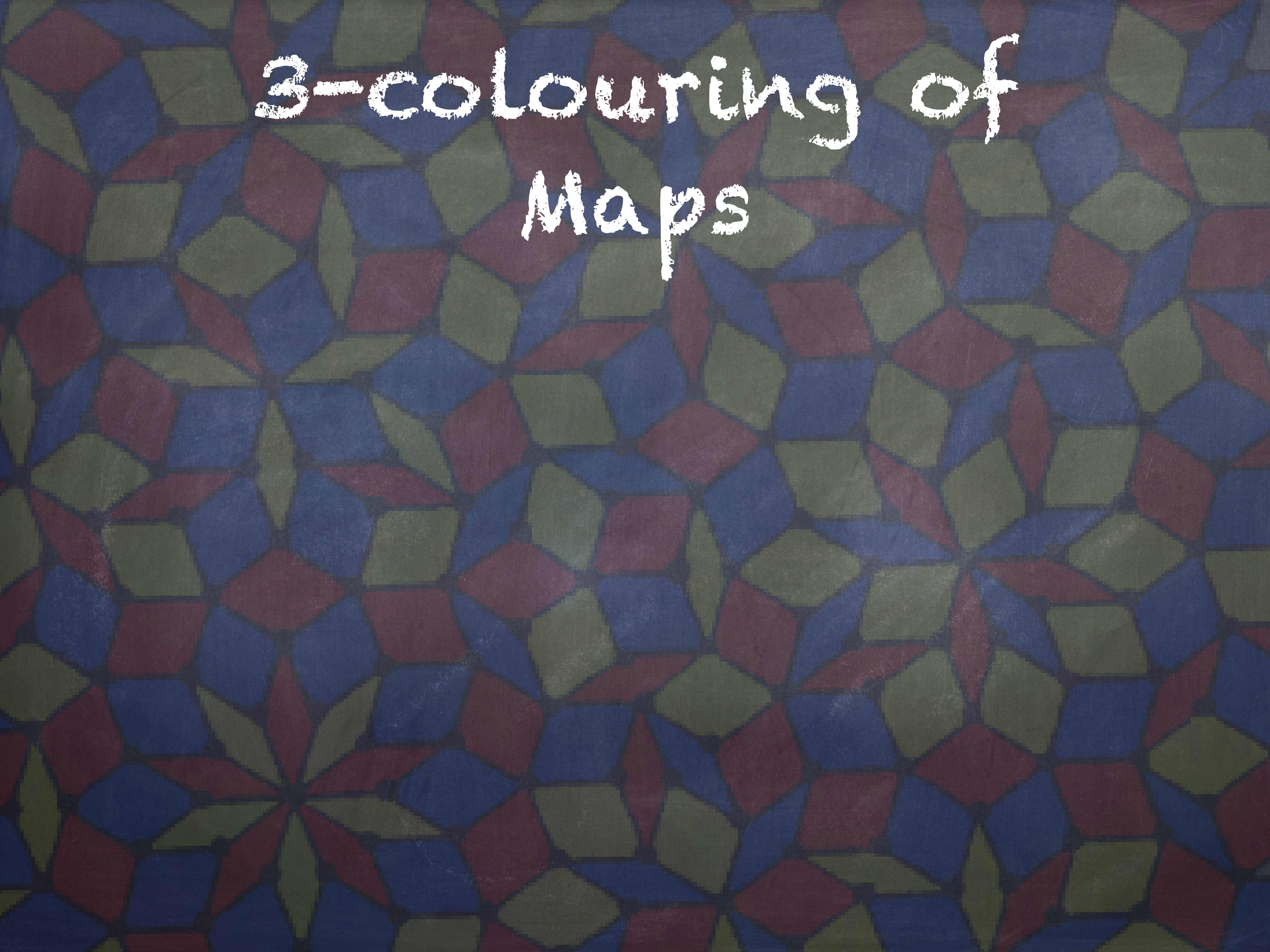
K-colouring of Maps (planar graphs)

- K=1 only the maps with zero or one region are 1-colourable.
- K=2 easy to decide. Impossible as soon as 3 regions touch each other.
- K=3 No known efficient algorithm to decide. It is easy to verify a solution.

K-colouring of Maps (planar graphs)

- K=1 only the maps with zero or one region are 1-colourable.
- K=2 easy to decide. Impossible as soon as 3 regions touch each other.
- K=3 No known efficient algorithm to decide. It is easy to verify a solution.
- K≥4 all maps are 4-colourable. (long proof) Does not imply easy to find a 4-colouring.

3-colouring of Maps



3-colouring of Maps

- Seems hard to solve in general,

3-colouring of Maps

- Seems hard to solve in general,
- Is easy to verify when a solution is given, (is in NP : guess a solution and verify it)

3-colouring of Maps

- Seems hard to solve in general,
- Is easy to verify when a solution is given, (is in NP : guess a solution and verify it)
- Is a special type of problem (NP-complete) because an efficient solution to it would yield efficient solutions to ALL problems in NP!

Examples of NP-Complete Problems

Examples of NP-Complete Problems

- SAT: given a boolean formula, is there an assignment of the variables making the formula evaluate to true ?

Examples of NP-Complete Problems

- SAT: given a boolean formula, is there an assignment of the variables making the formula evaluate to true ?
- Travelling Salesman: given a set of cities and distances between them, what is the shortest route to visit each city once.

Examples of NP-Complete Problems

- SAT: given a boolean formula, is there an assignment of the variables making the formula evaluate to true ?
- Travelling Salesman: given a set of cities and distances between them, what is the shortest route to visit each city once.
- KnapSack: given items with various weights, is there of subset of them of total weight K.

NP-Complete Problems

NP-Complete Problems

- Many practical problems are NP-complete.

NP-Complete Problems

- Many practical problems are NP-complete.
- If any of them is easy, they are all easy.

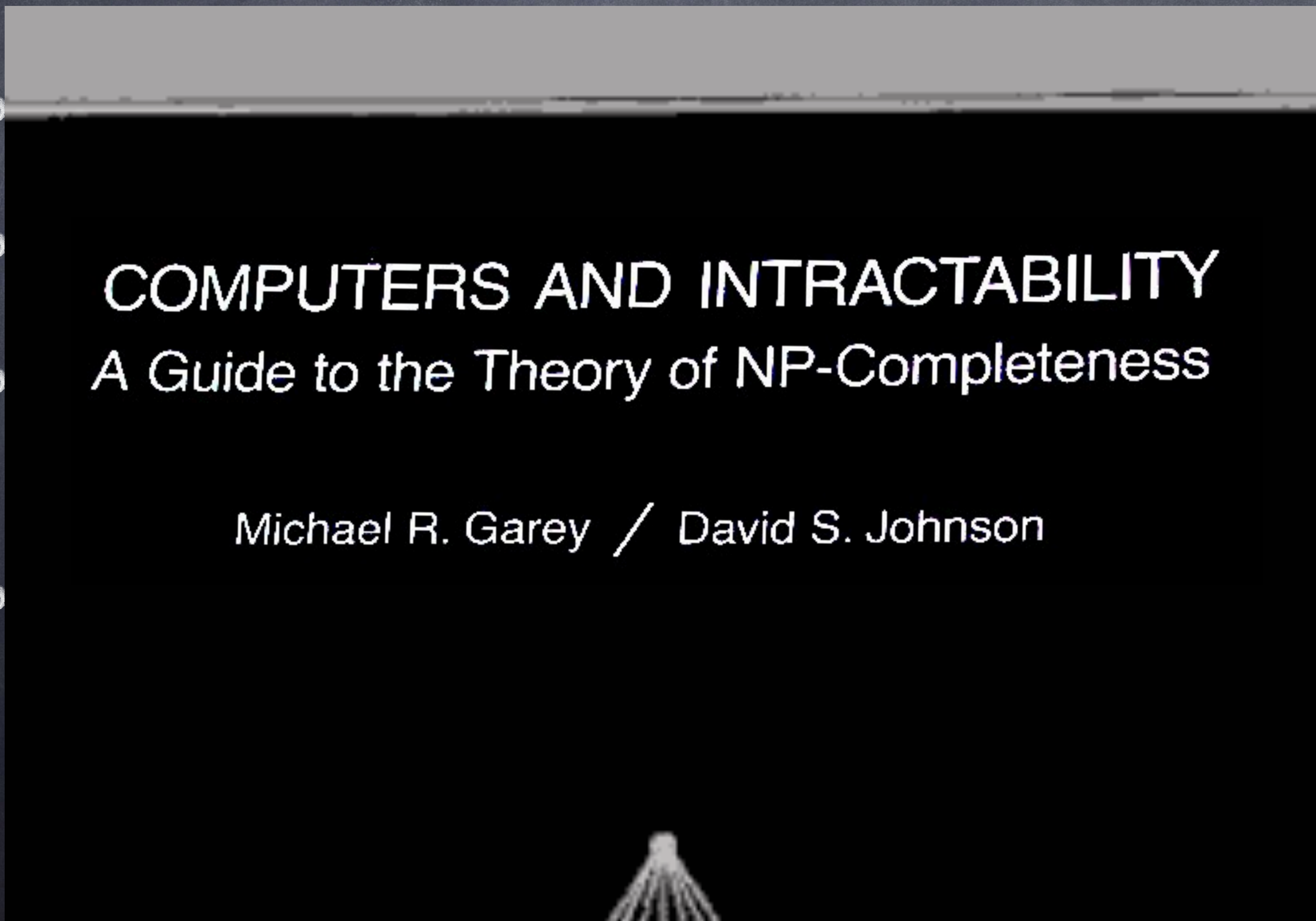
NP-Complete Problems

- Many practical problems are NP-complete.
- If any of them is easy, they are all easy.
- In practice, some of them may be solved efficiently in some special cases.

NP-Complete Problems

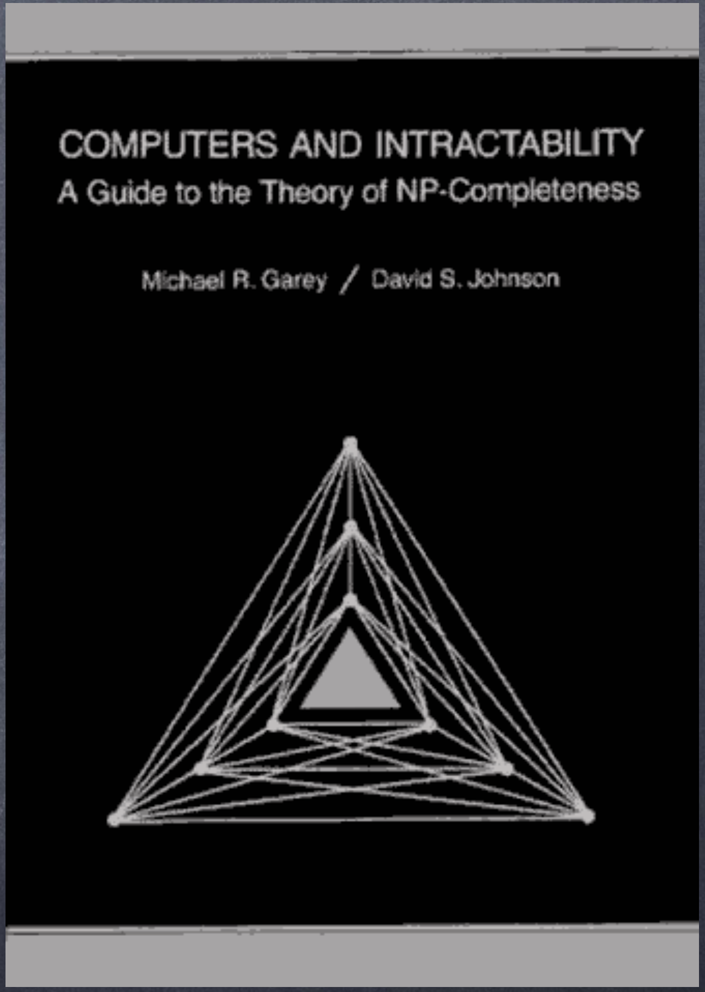
- Many practical problems are NP-complete.
- If any of them is easy, they are all easy.
- In practice, some of them may be solved efficiently in some special cases.
- Some books list hundreds of such problems.

NP-Complete Problems

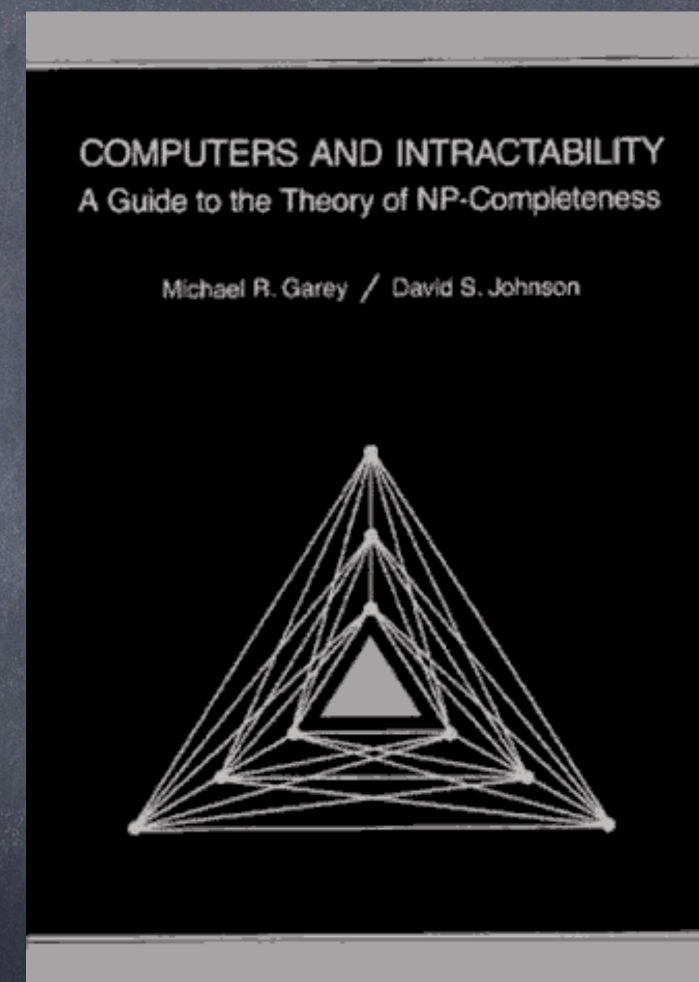


NP-Complete Problems

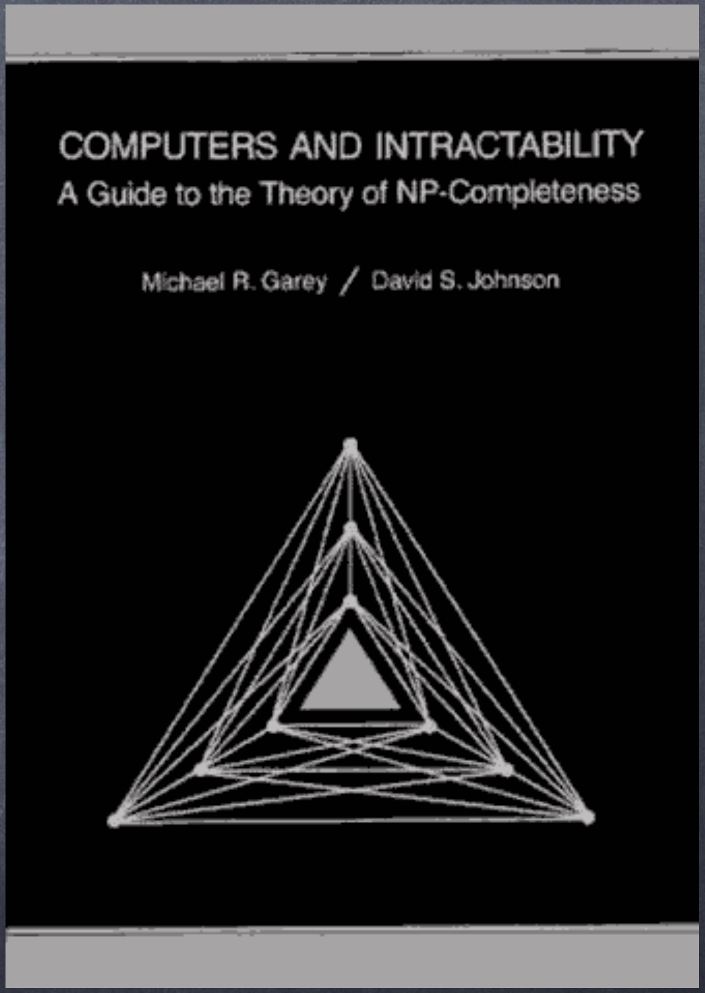
NP-Complete Problems



NP-Complete Problems

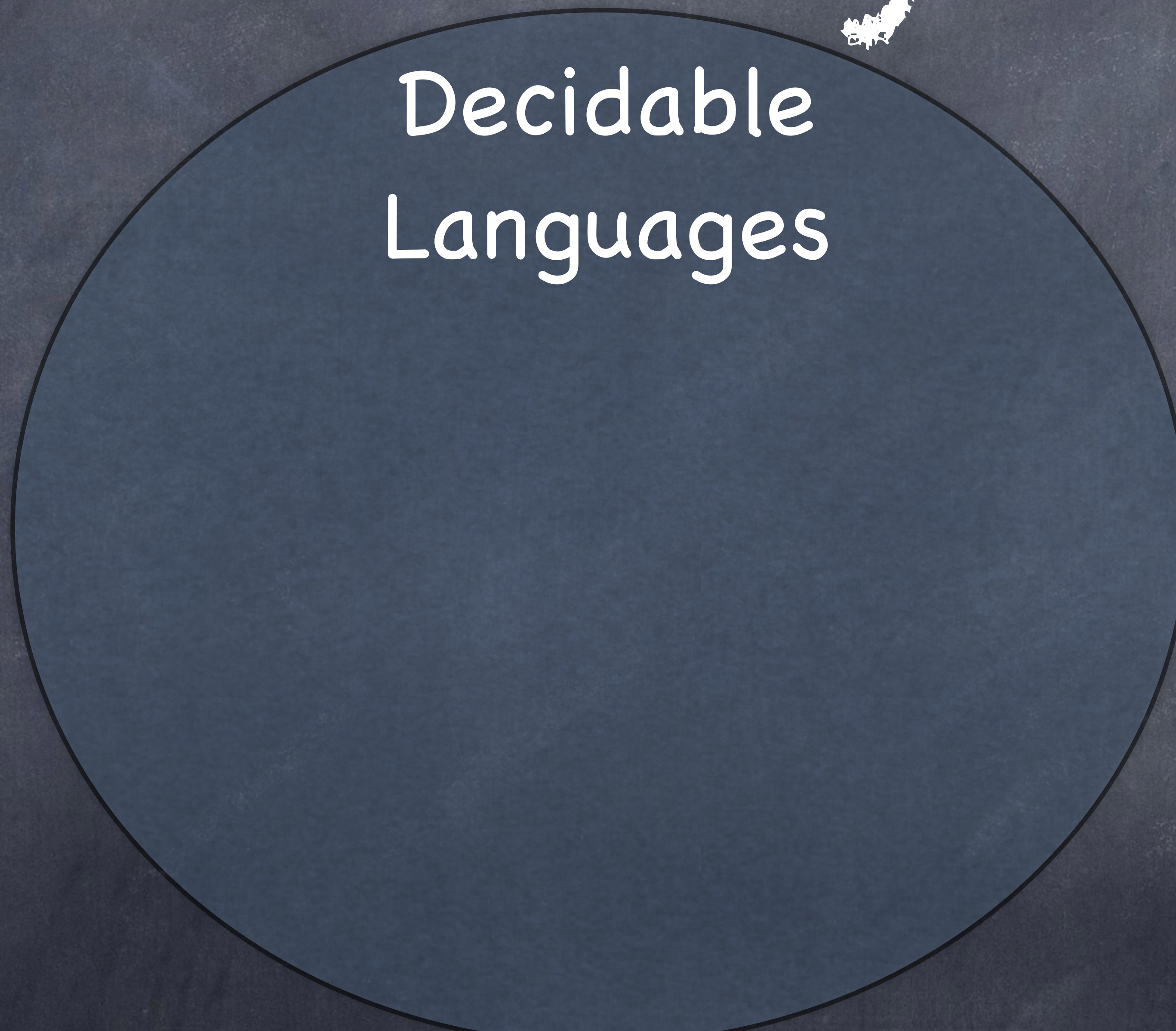


NP-Complete Problems

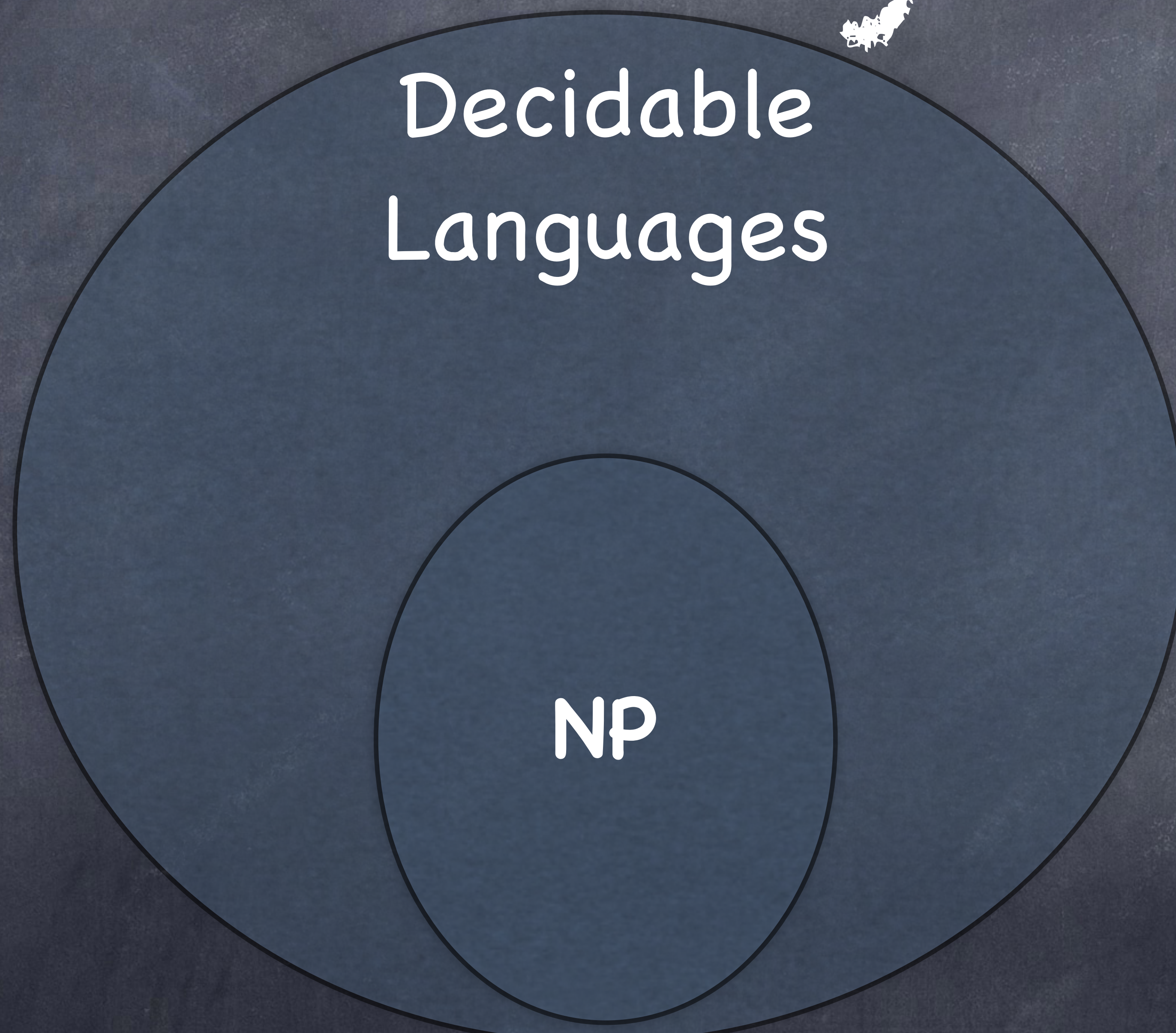


100 pages
1979 !!!

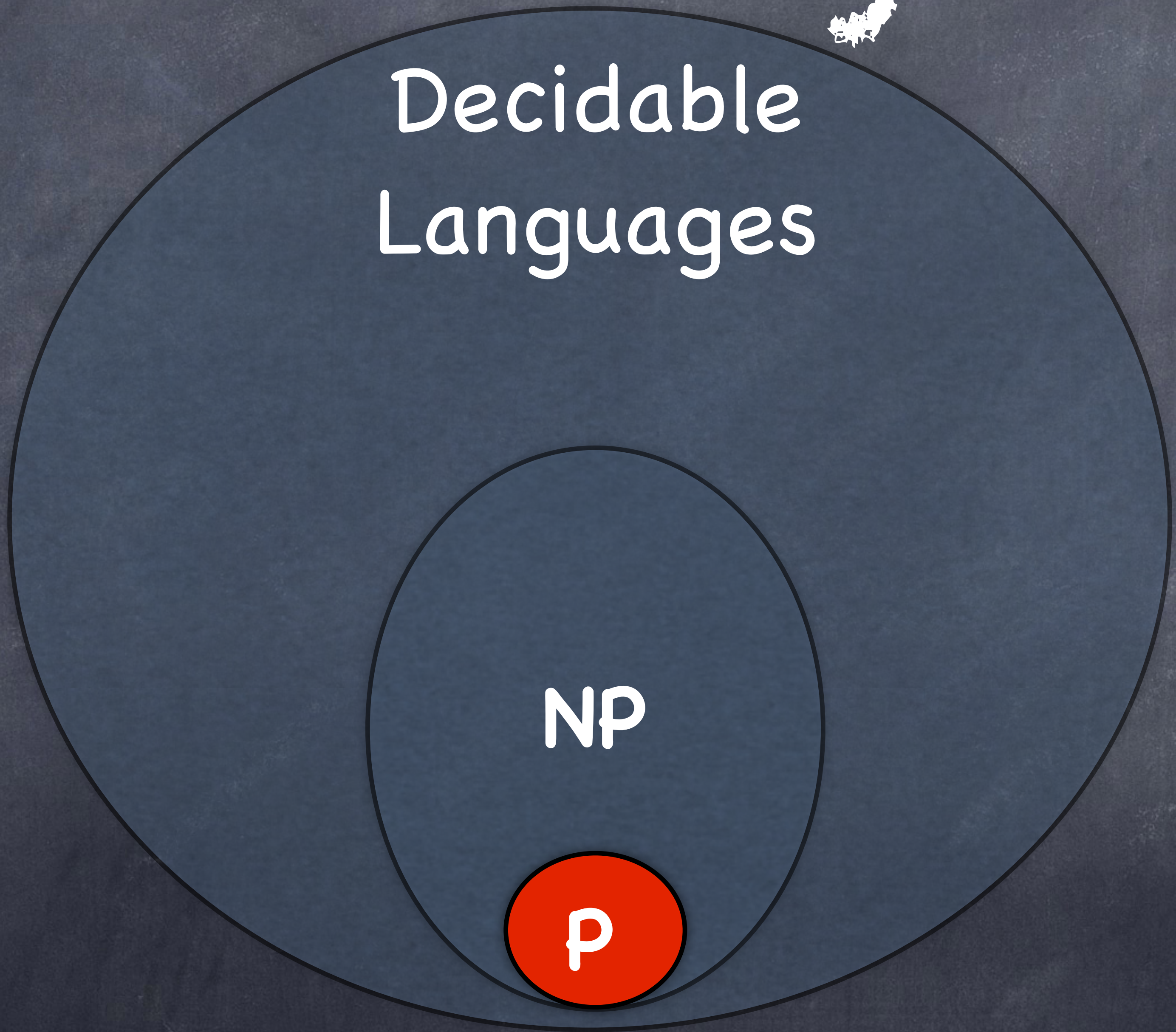
Complexity Theory



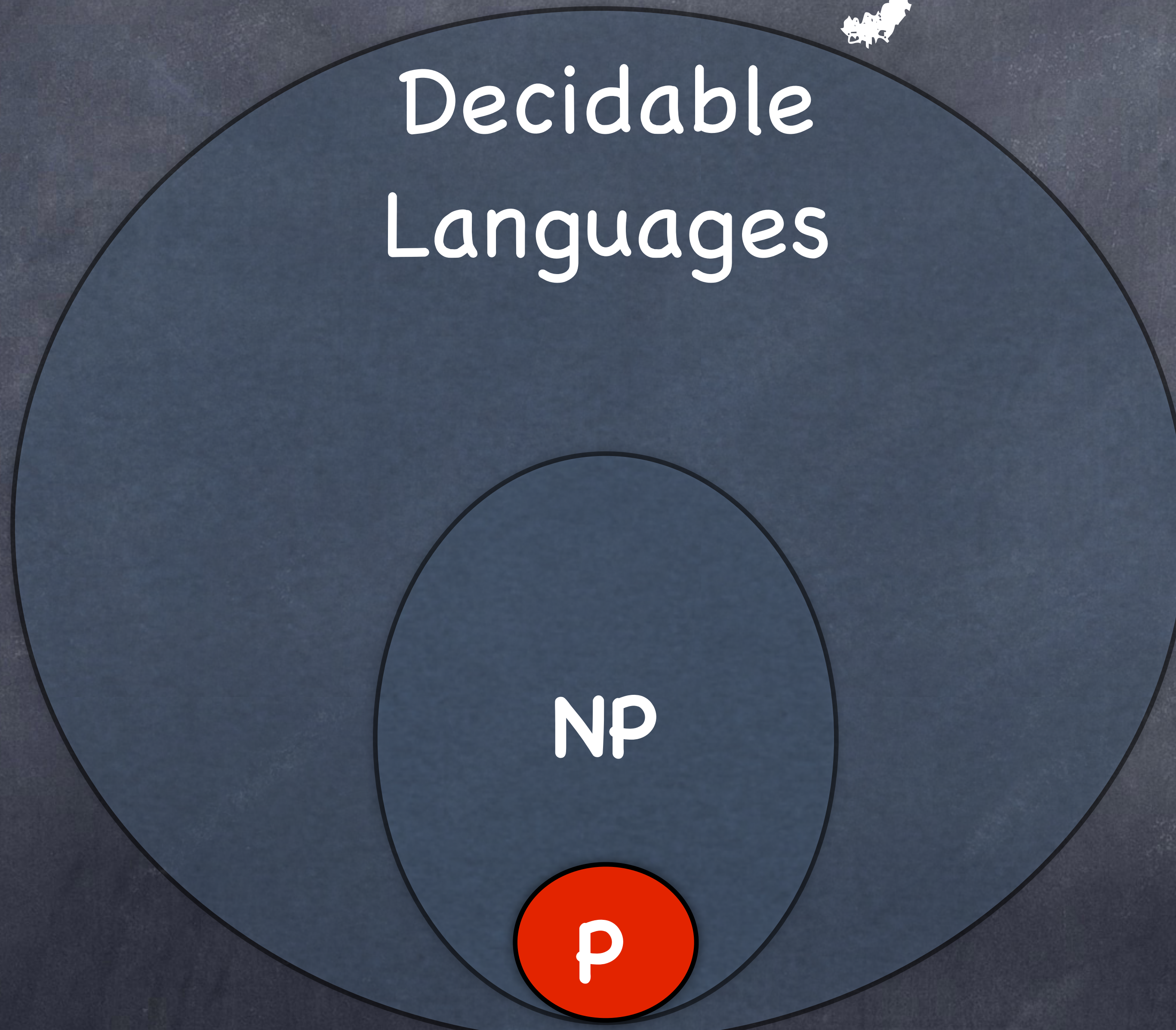
Complexity Theory



Complexity Theory



Complexity Theory



P = NP ?

Complexity Theory

P = NP ?

P vs NP

P vs NP

DEFINITION 7.7

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

P vs NP

DEFINITION 7.7

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

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

P vs NP

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

P vs NP

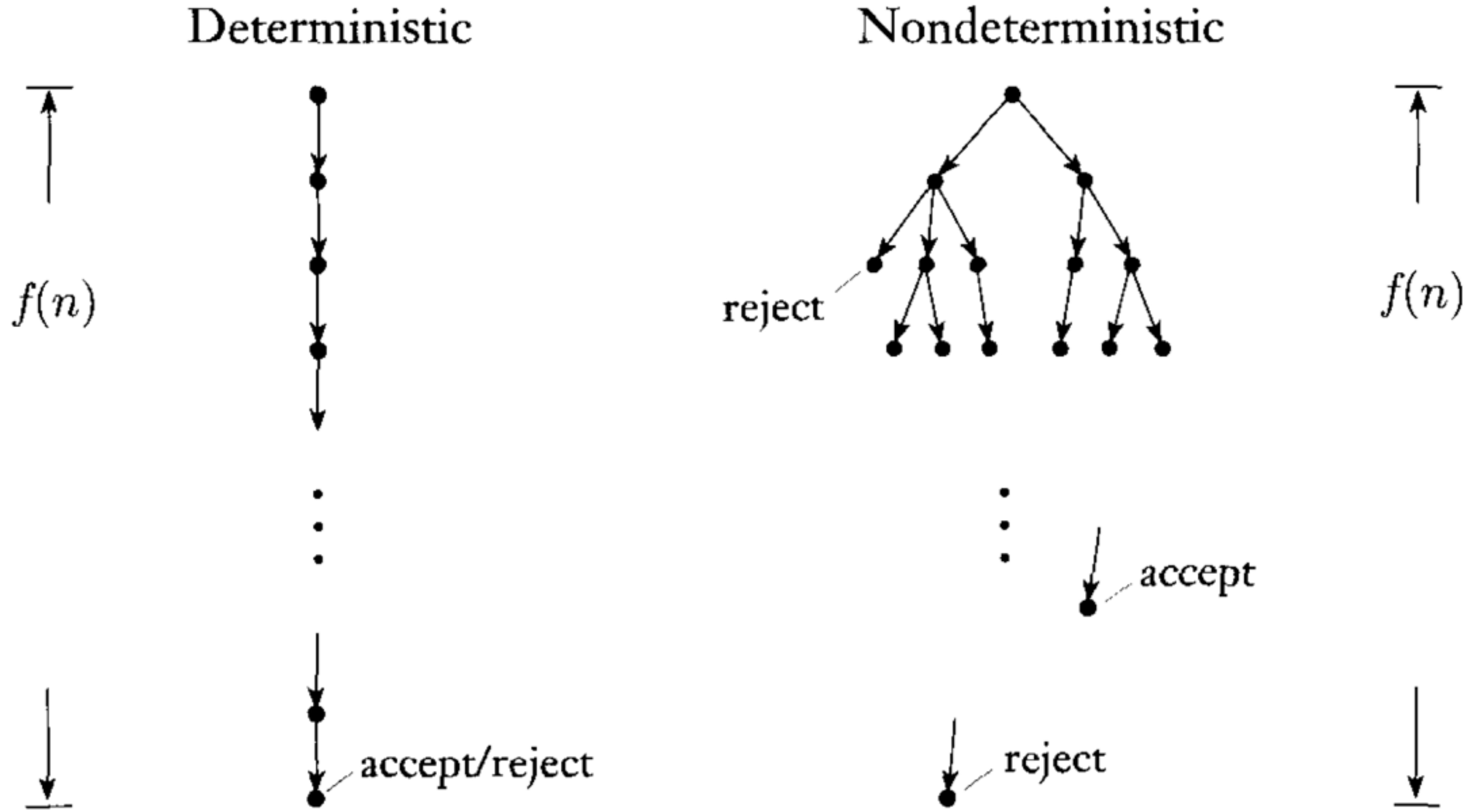


FIGURE 7.10
Measuring deterministic and nondeterministic time

P vs NP

DEFINITION 7.21

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

COROLLARY 7.22

$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$.

P vs NP

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

P vs NP

A ***clique*** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A ***k-clique*** is a clique that contains k nodes. Figure 7.23 illustrates a graph having a 5-clique

P vs NP

A *clique* in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A ***k-clique*** is a clique that contains k nodes. Figure 7.23 illustrates a graph having a 5-clique

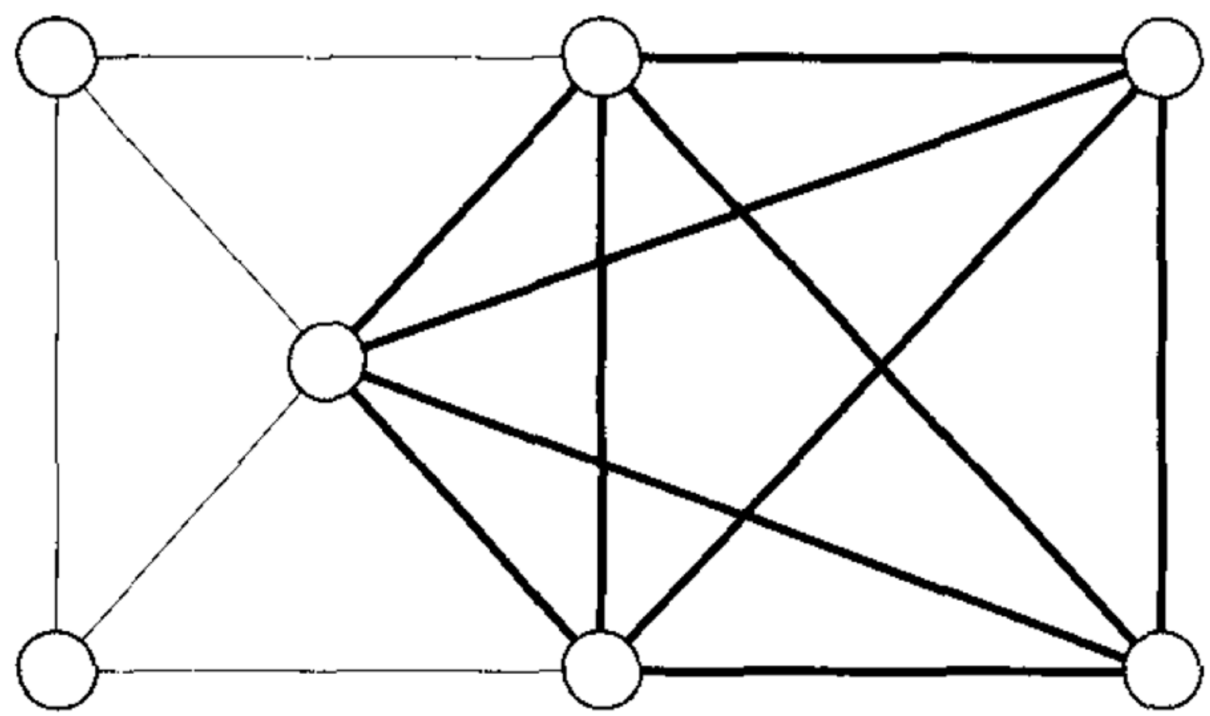


FIGURE 7.23
A graph with a 5-clique

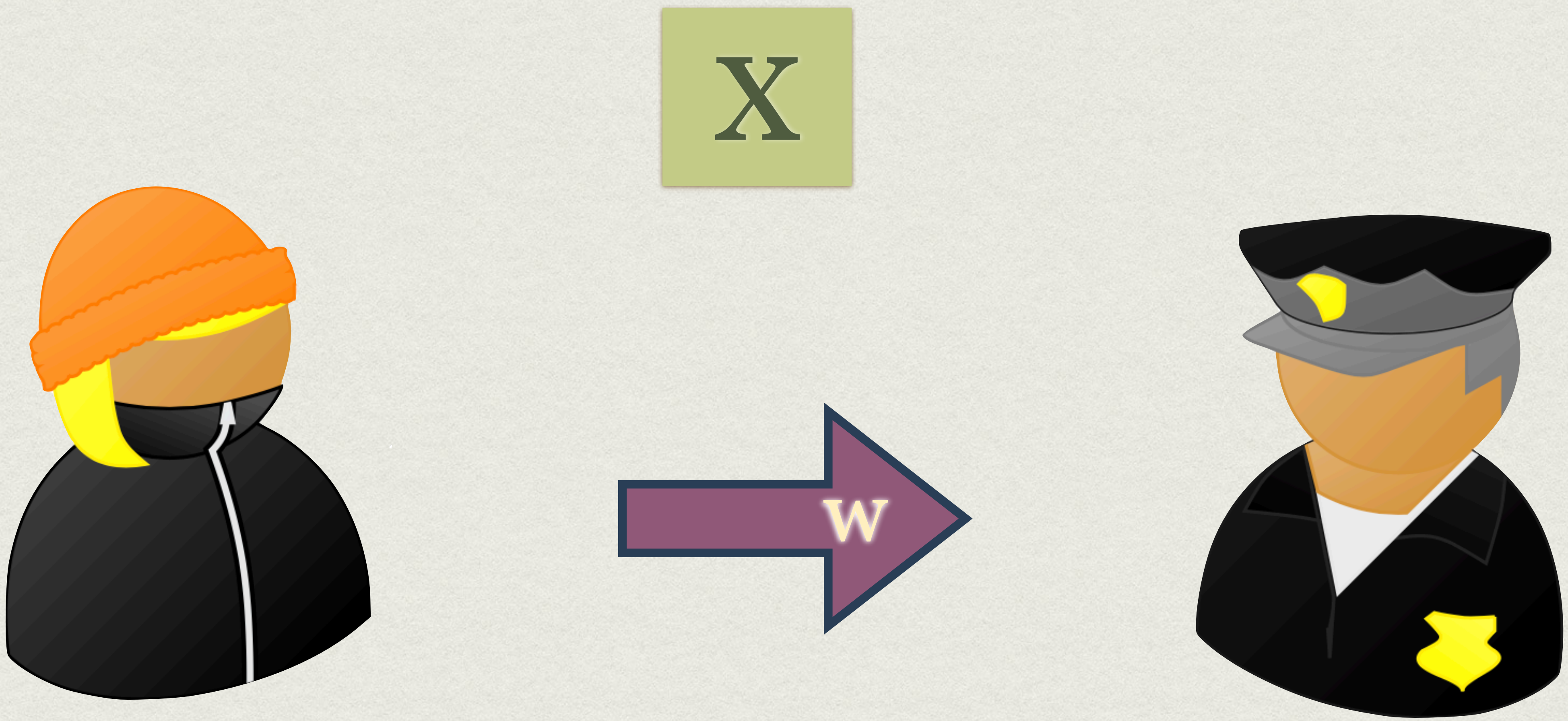
P vs NP

The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle |\ G \text{ is an undirected graph with a } k\text{-clique}\}.$$

Completeness

$\exists$  $, \forall x \in L, \exists w,$ [ (x, w) accepts]

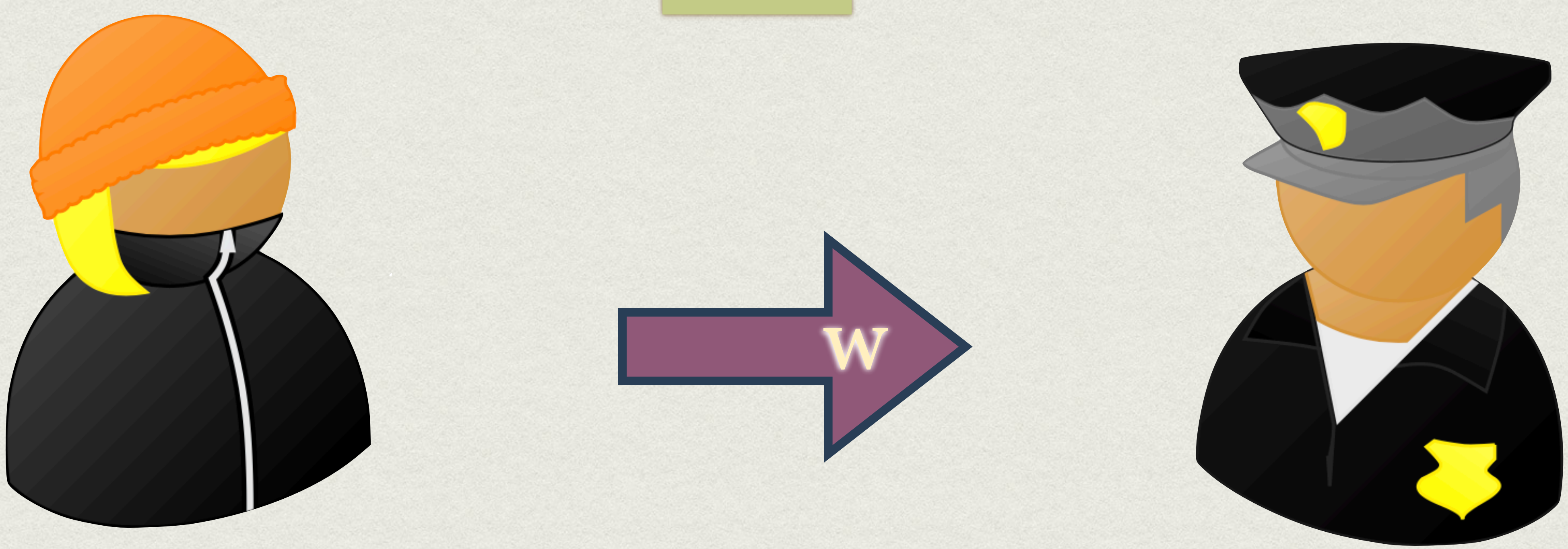


Completeness

x∈L

$\exists$  $, \forall x \in L, \exists w, [$  (x, w) accepts $]$

X

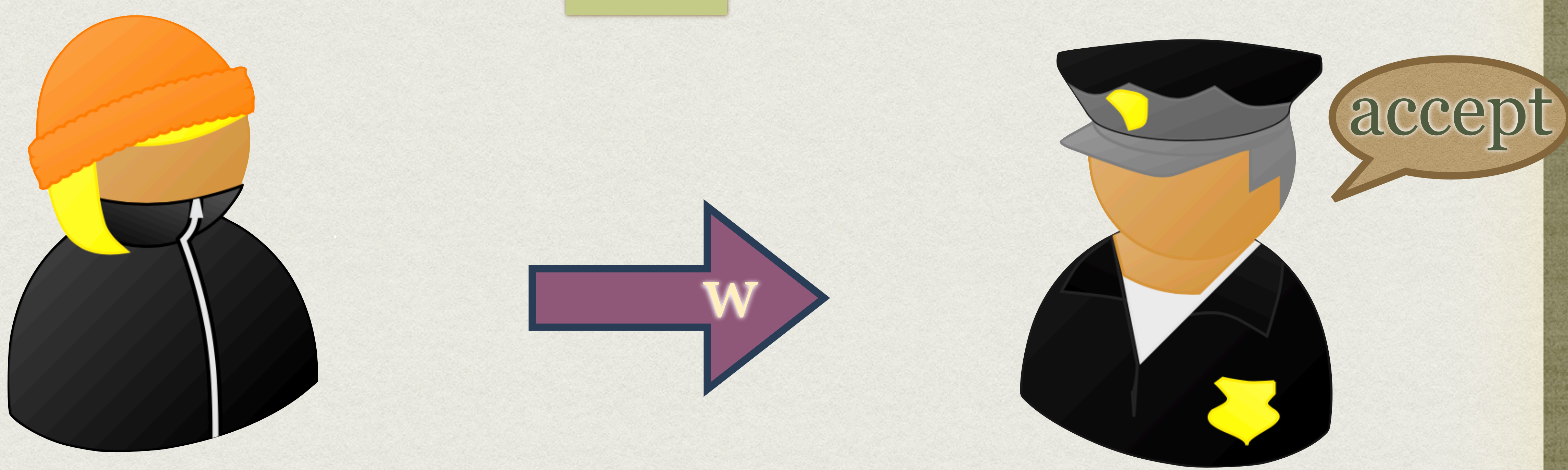


Completeness

x∈L

$\exists$, $\forall x \in L, \exists w,$ [(x, w) accepts]

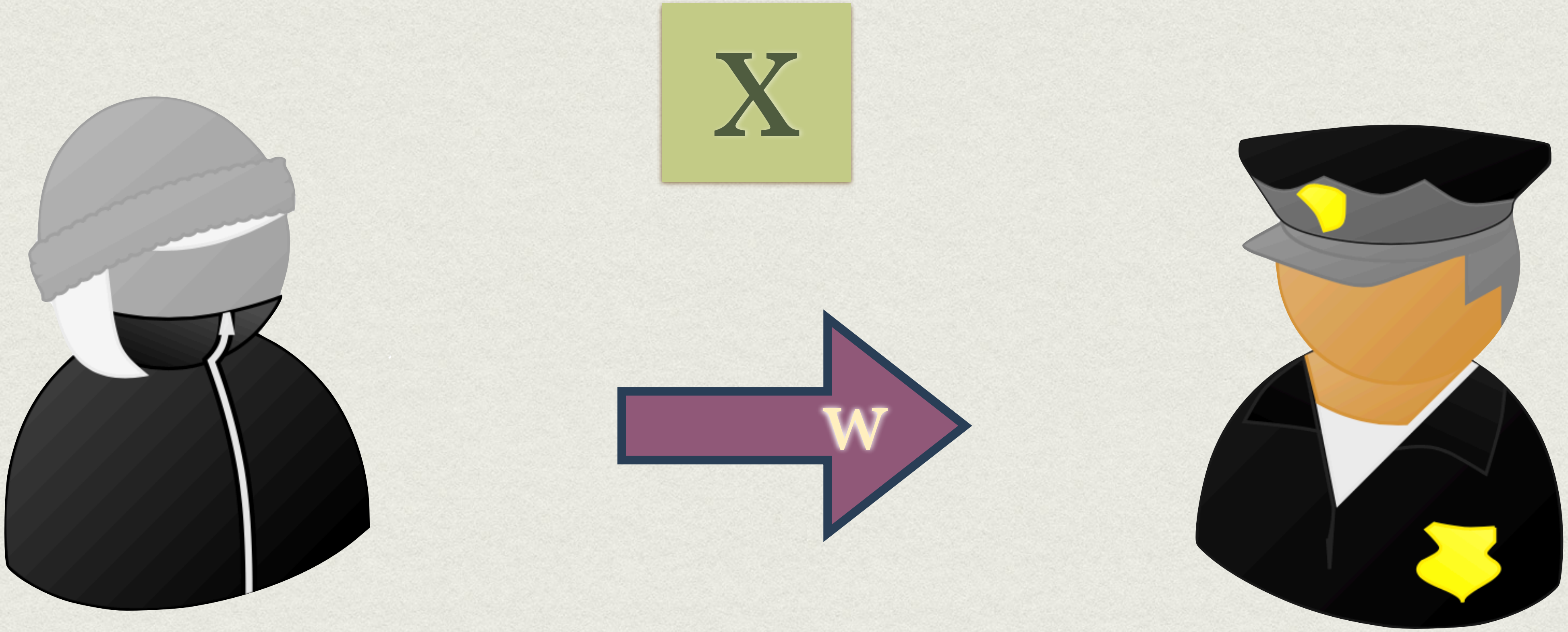
x



Soundness

$\exists\ ,\ \forall x \in L, \exists w, [\ (x, w)$ accepts $]$

and $\forall x \notin L, \forall w,\ [\ (x, w)$ rejects $]$

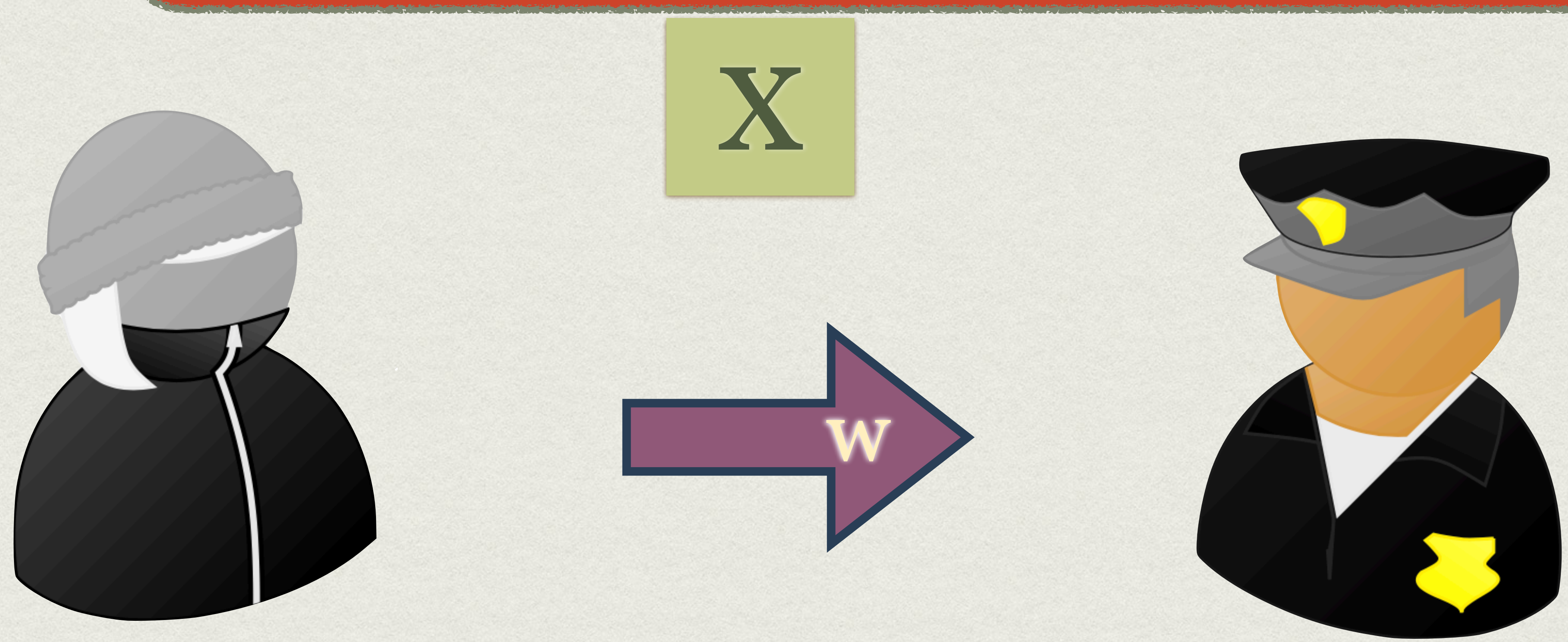


Soundness

$x \notin L$

$\exists$, $\forall x \in L, \exists w,$ [(x, w) accepts]

and $\forall x \notin L, \forall w,$ [(x, w) rejects]

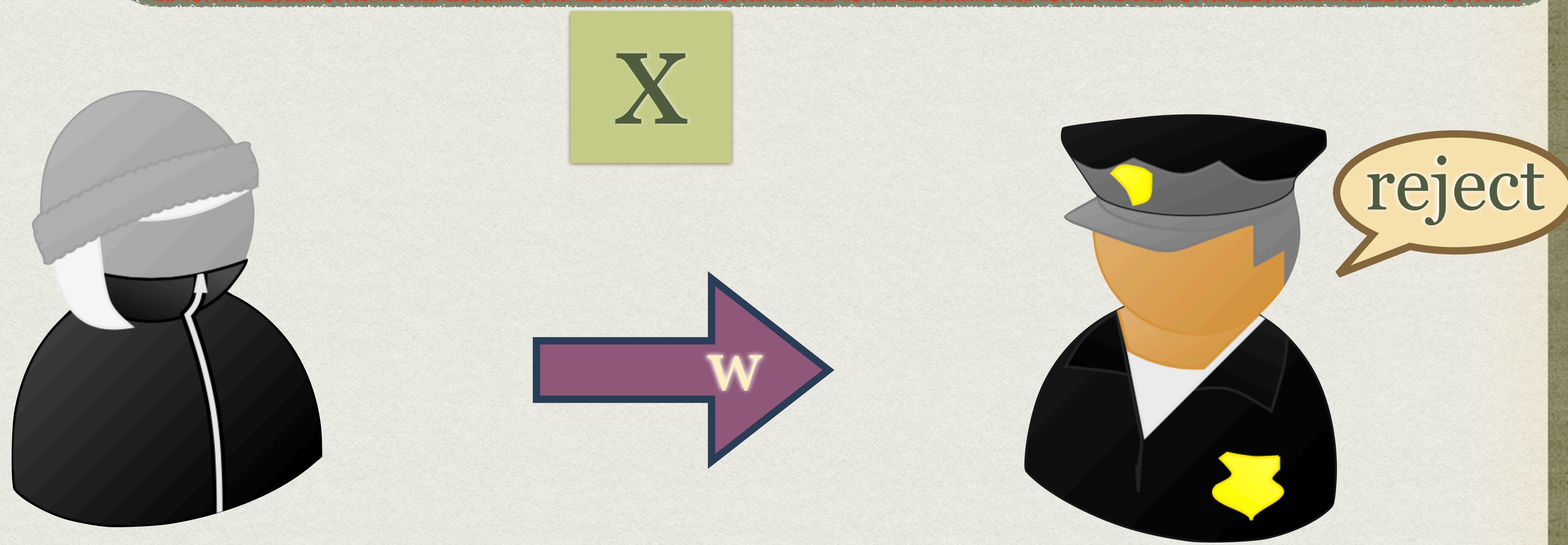


Soundness

$x \notin L$

$\exists$, $\forall x \in L, \exists w,$ [(x, w) accepts]

and $\forall x \notin L, \forall w,$ [(x, w) rejects]



THEOREM 7.24

CLIQUE is in NP.

PROOF IDEA The clique is the certificate.

PROOF The following is a verifier V for *CLIQUE*.

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

$N =$ "On input $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

SAT

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

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

Now we state the Cook–Levin theorem, which links the complexity of the *SAT* problem to the complexities of all problems in NP.

SAT

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $\boxed{x = 0,\ y = 1, \text{ and } z = 0 \text{ makes } \phi \text{ evaluate to 1.}}$ We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

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

Now we state the Cook–Levin theorem, which links the complexity of the SAT problem to the complexities of all problems in NP.

SAT

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $\boxed{x = 0, y = 1, \text{and } z = 0 \text{ makes } \phi \text{ evaluate to } 1.}$ We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

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

Now we state the Cook–Levin theorem, which links the complexity of the SAT problem to the complexities of all problems in NP.

THEOREM 7.27

Cook–Levin theorem $SAT \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$.

Poly-time Reducibility

DEFINITION 7.28

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Poly-time Reducibility

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

Poly-time Reducibility

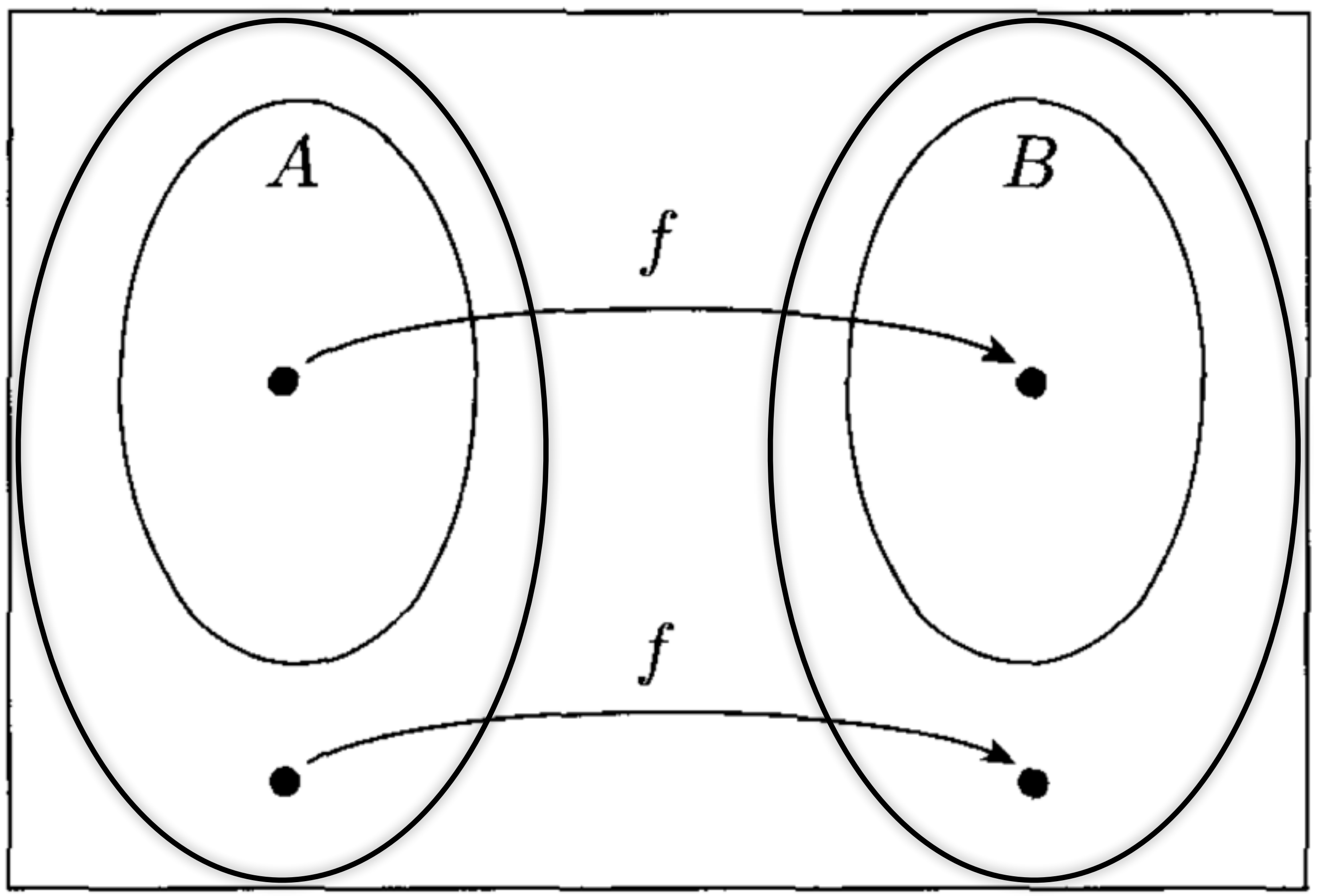


FIGURE 7.30
Polynomial time function f reducing A to B

Poly-time Reducibility

THEOREM 7.31

If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

NP-completeness

DEFINITION 7.34

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

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

THEOREM 7.35

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

PROOF This theorem follows directly from the definition of polynomial time reducibility.

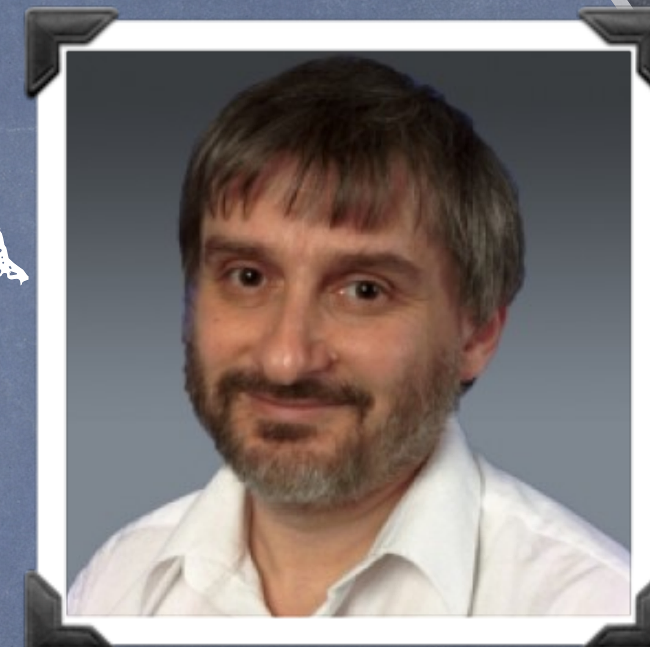
NP-completeness

THEOREM 7.36

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

PROOF We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in NP is polynomial time reducible to C.

Cook-Levin Theorem



Cook-Levin Theorem

THEOREM 7.37

SAT is NP-complete.[2]

This theorem restates Theorem 7.27, the Cook–Levin theorem, in another form.

Cook-Levin Theorem

PROOF First, we show that *SAT* is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Cook-Levin Theorem

PROOF First, we show that SAT is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Next, we take any language* A in NP and show that A is polynomial time reducible to SAT. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (For convenience we actually assume that N runs in time $n^k - 3$, but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

Cook-Levin Theorem

PROOF First, we show that *SAT* is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Next, we take any language* A in NP and show that A is polynomial time reducible to *SAT*. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (For convenience we actually assume that N runs in time $n^k - 3$, but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

*"any language A in NP" really means:

"any language A *provably* in **NP**".

A ***tableau*** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w, as shown in the following figure.

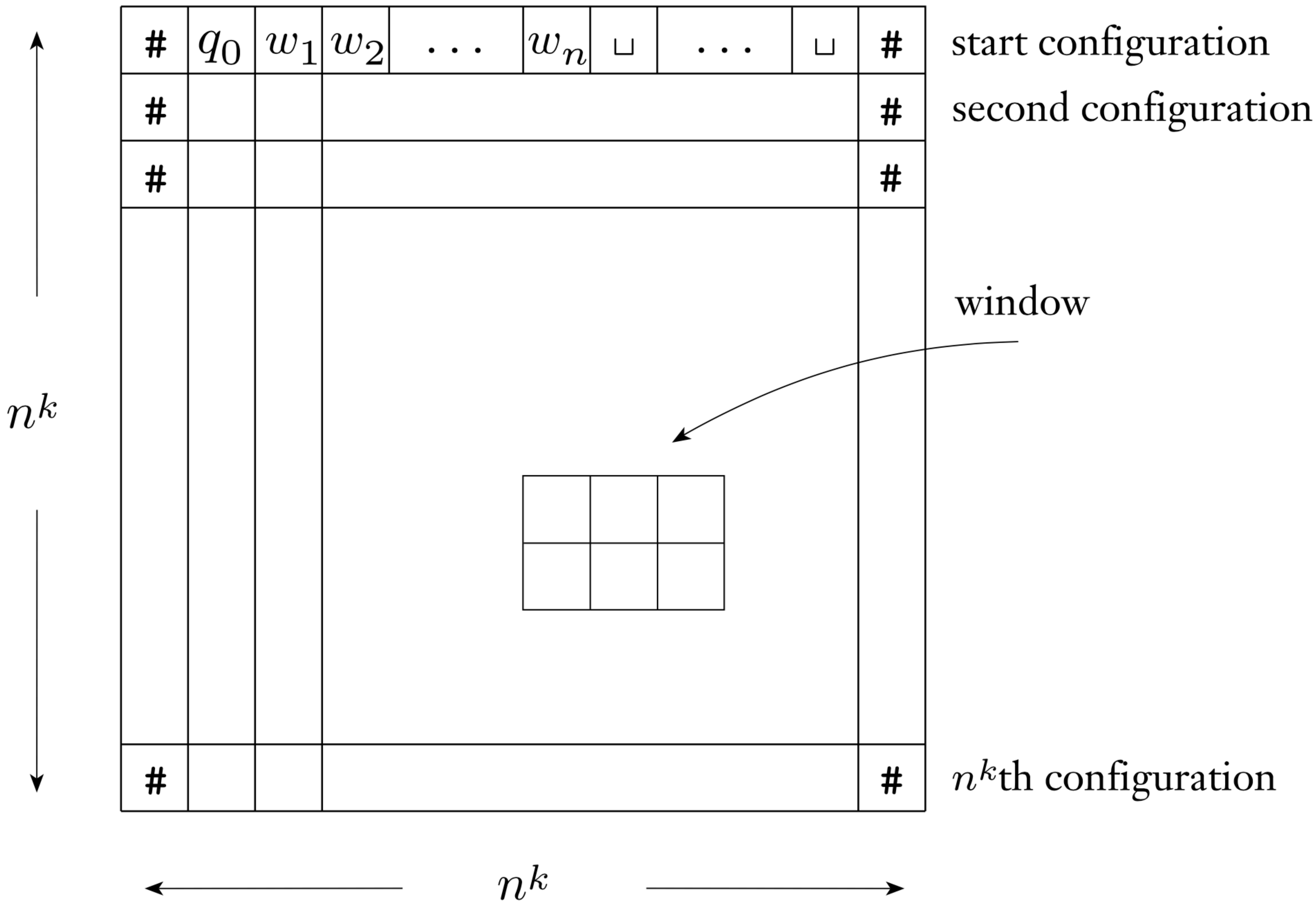
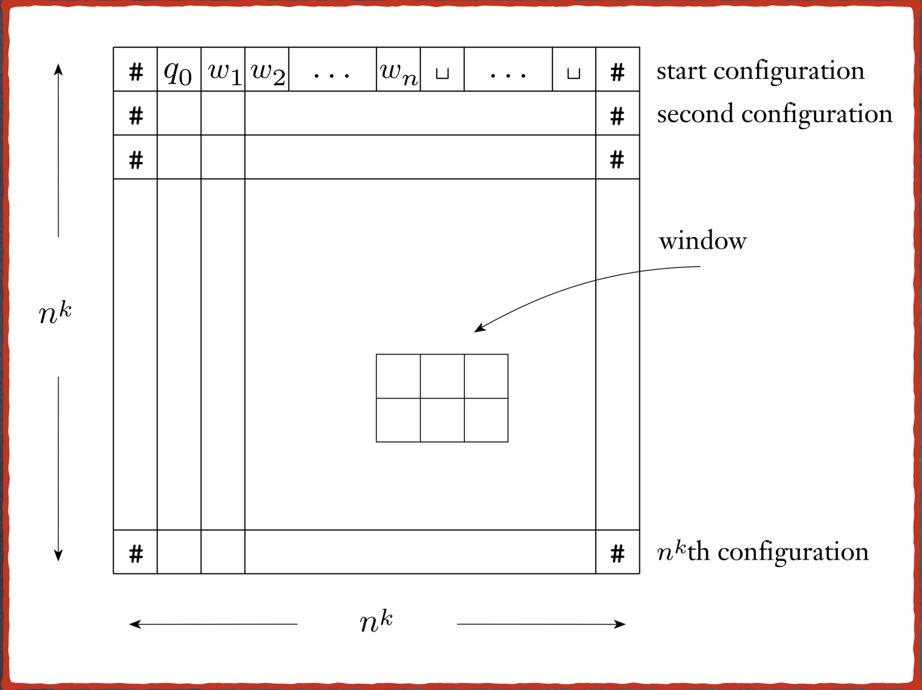
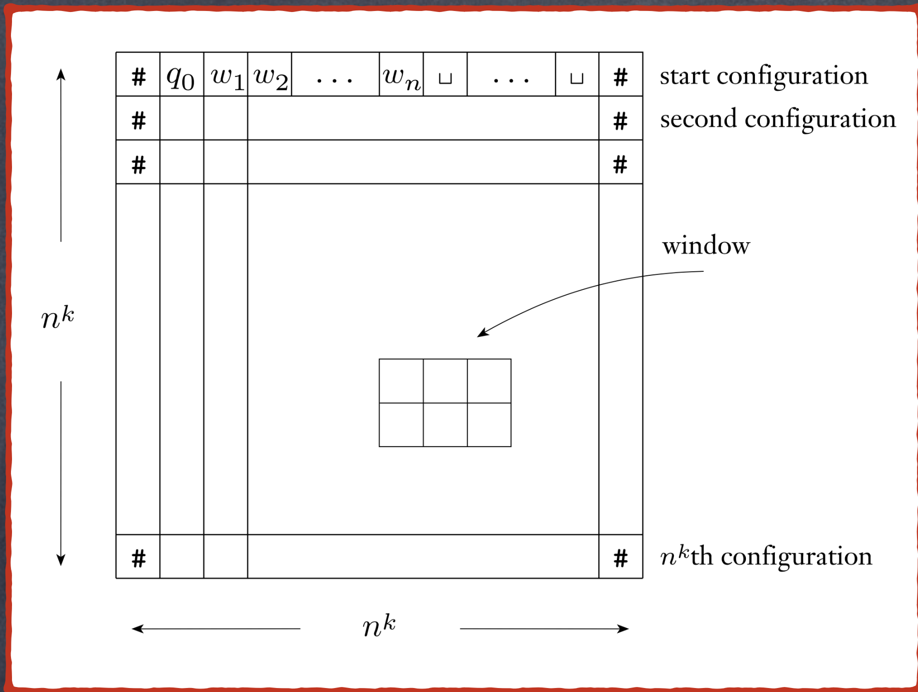


FIGURE 7.38

A tableau is an $n^k \times n^k$ table of configurations

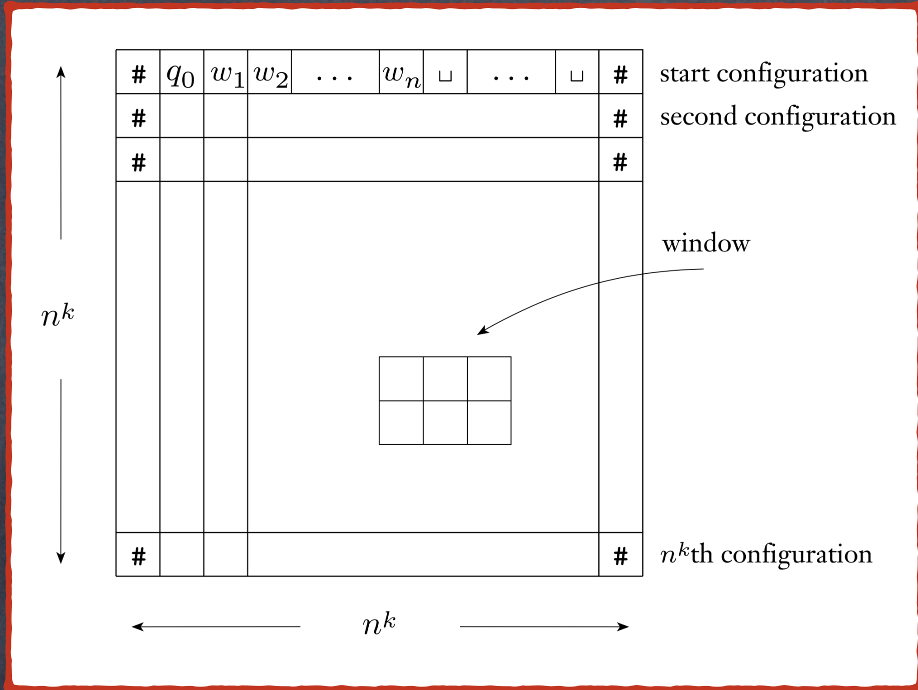
Cook-Levin Theorem





Cook-Levin Theorem

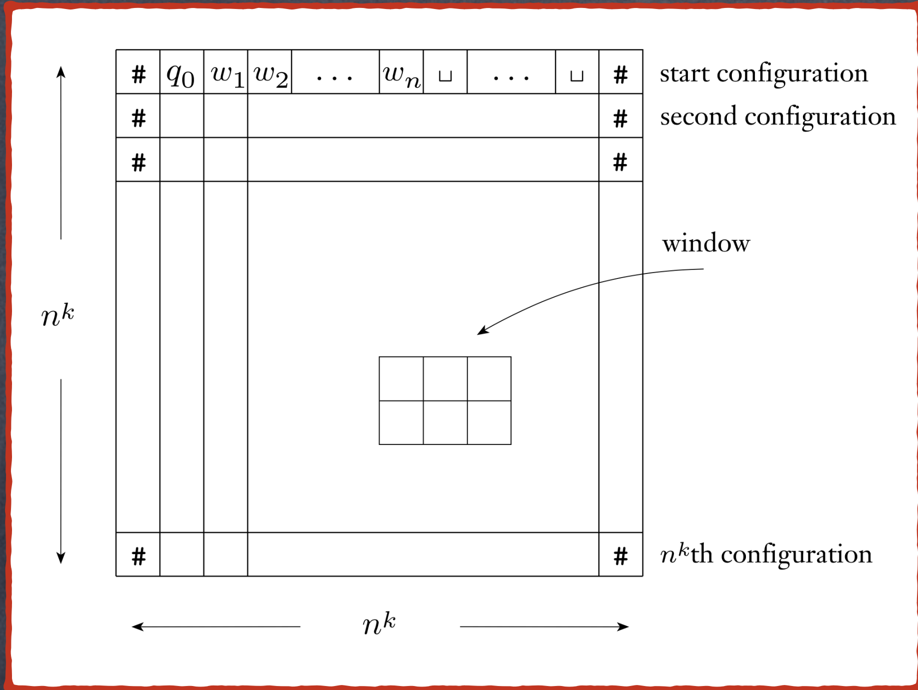
Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.



Cook-Levin Theorem

Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.

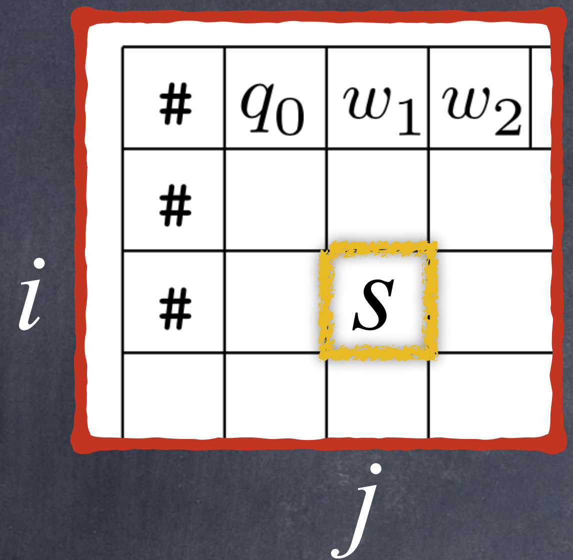


Cook-Levin Theorem

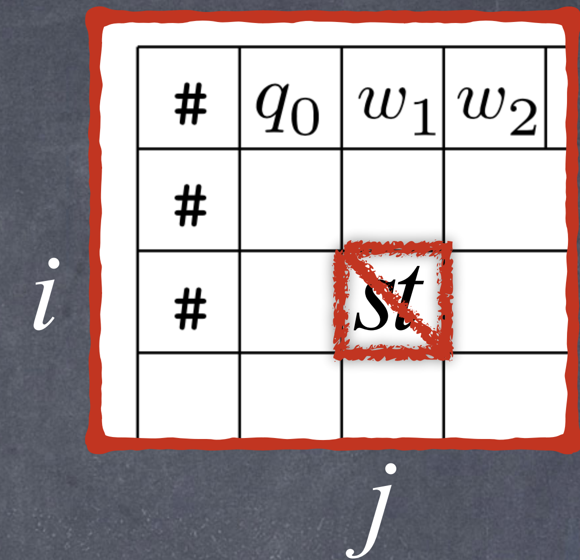
Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.

$$\phi = \phi_{cell} \cup \phi_{start} \cup \phi_{accept} \cup \phi_{move}$$

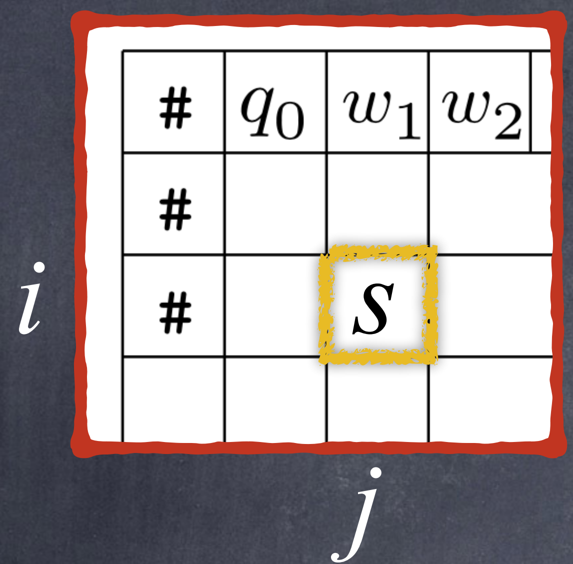


Cook-Levin Theorem: ϕ_{cell}

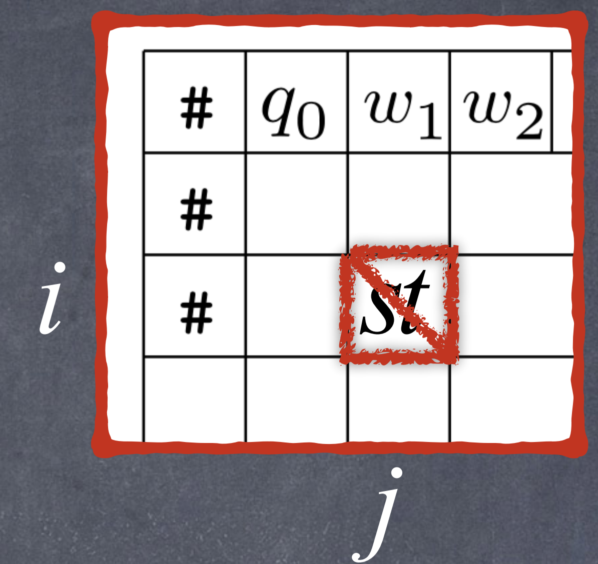


turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right].$$



Cook-Levin Theorem: ϕ_{cell}



turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1\le i,j\le n^k} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t\in C\\ s\neq t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right].$$

$$C = Q \cup \Gamma \cup \{\#\}.$$

Cook-Levin Theorem: ϕ_{cell}

The symbols $\bigwedge$ and $\bigvee$ stand for iterated AND and OR. For example, the expression in the preceding formula

$$\bigvee_{s \in C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

where $C = \{s_1, s_2, \ldots, s_l\}$. Hence ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k.

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \, .\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} .\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \, .\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = \; & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,\boxed{w_2}} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\,.\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,\boxed{w_2}} \wedge \ldots \wedge x_{1,n+2,\boxed{w_n}} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,\boxed{w_2}} \wedge \ldots \wedge x_{1,n+2,\boxed{w_n}} \wedge \\ & x_{1,n+3,\boxed{\sqcup}} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\,.\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,\boxed{w_2}} \wedge \ldots \wedge x_{1,n+2,\boxed{w_n}} \wedge \\ & x_{1,n+3,\boxed{\sqcup}} \wedge \ldots \wedge x_{1,n^k-1,\boxed{\sqcup}} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

Cook-Levin Theorem: ϕ_{start}

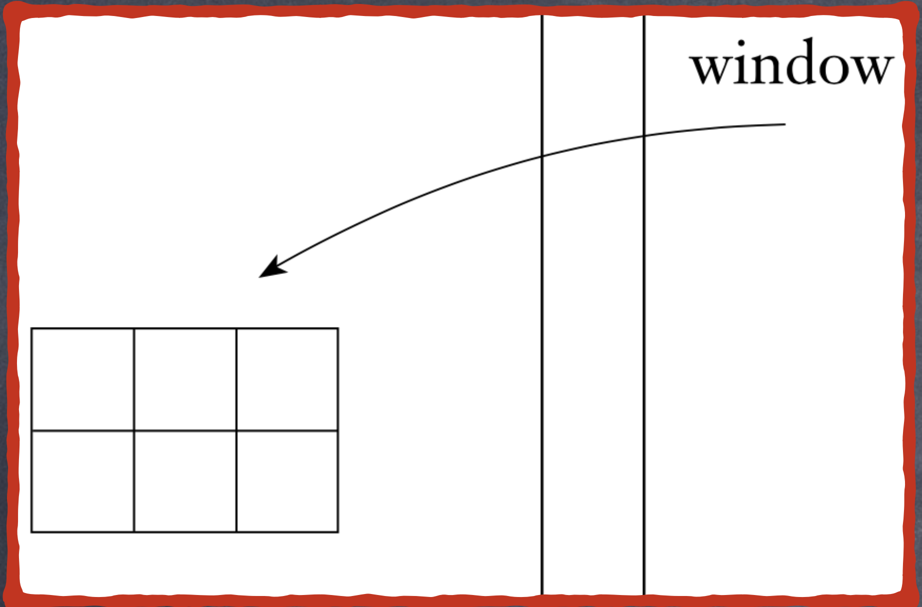
Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\boxed{\#}} \wedge x_{1,2,\boxed{q_0}} \wedge \\ & x_{1,3,\boxed{w_1}} \wedge x_{1,4,\boxed{w_2}} \wedge \ldots \wedge x_{1,n+2,\boxed{w_n}} \wedge \\ & x_{1,n+3,\boxed{\sqcup}} \wedge \ldots \wedge x_{1,n^k-1,\boxed{\sqcup}} \wedge x_{1,n^k,\boxed{\#}} .\end{aligned}$$

Cook-Levin Theorem: ϕ_{accept}

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}} .$$



Cook-Levin Theorem: ϕ_{move}

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

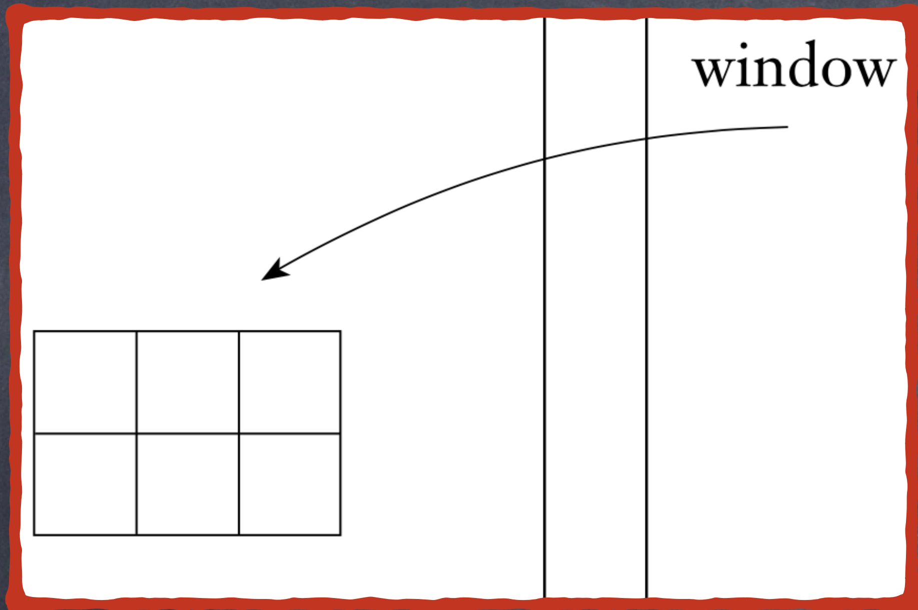
(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

FIGURE 7.39
Examples of legal windows



Cook-Levin Theorem: ϕ_{move}

(a)

a	b	a
a	a	a

(b)

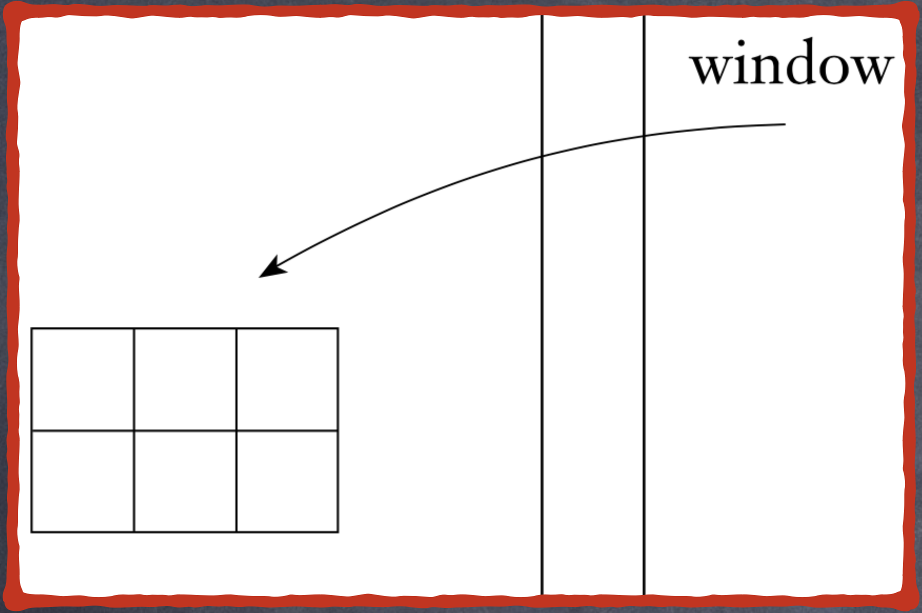
a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

FIGURE 7.40
Examples of illegal windows

Cook-Levin Theorem: ϕ_{move}



(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

$\boldsymbol{\delta}(q_1,b)=(q_1,c,L)$

FIGURE 7.40
Examples of illegal windows

Cook-Levin Theorem: ϕ_{move}

CLAIM 7.41

If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

Cook-Levin Theorem: ϕ_{move}

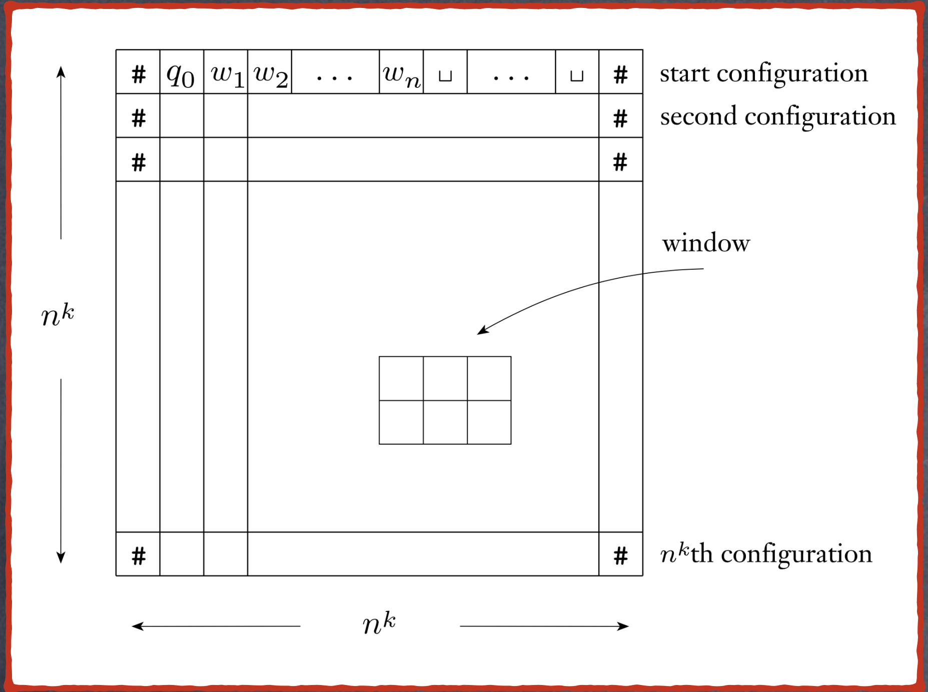
Now we return to the construction of ϕ_{move}. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula ϕ_{move} says that the settings of those six cells must be one of these ways, or

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \text{(the } (i,j) \text{ window is legal)}$$

Cook-Levin Theorem: ϕ_{move}

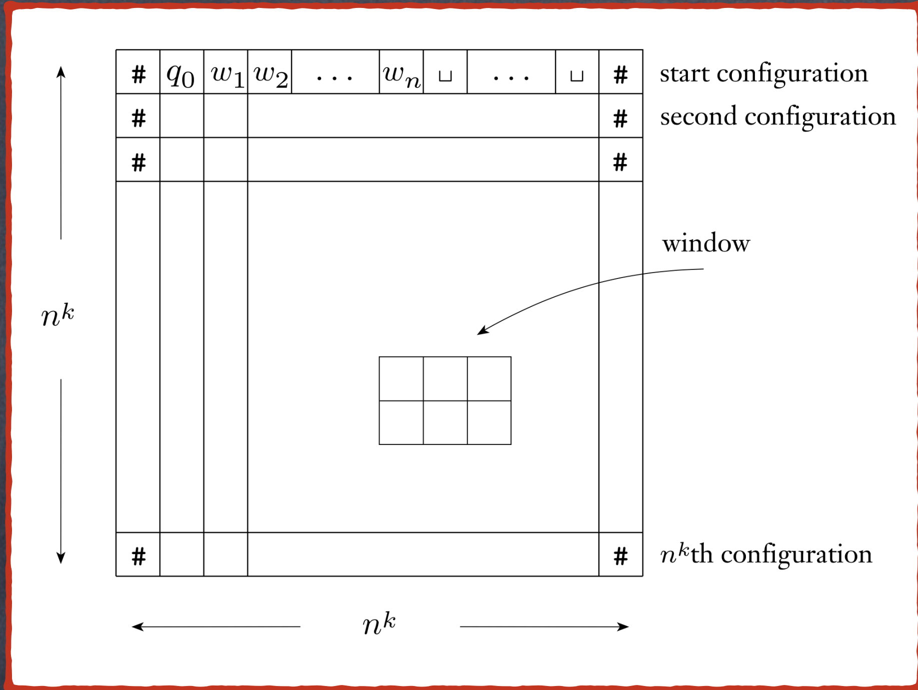
We replace the text "the (i, j) window is legal" in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1, \ldots, a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \right)$$



Cook-Levin Theorem

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.



Cook-Levin Theorem

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ.

$\langle \phi \rangle \in SAT$

iff

N accepts w within n^k steps.

NP-Complete Problems

3SAT is NP-Complete

literal is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A ***clause*** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in ***conjunctive normal form***, called a ***cnf-formula***, if it comprises several clauses connected with $\wedge$s, as in

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

It is a ***3cnf-formula*** if all the clauses have three literals, as in

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

Let $\mathit{3SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$. In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

3SAT is NP-Complete

COROLLARY 7.42

3SAT is NP-complete.

PROOF Obviously *3SAT* is in NP, so we only need to prove that all languages in NP reduce to *3SAT* in polynomial time. One way to do so is by showing that *SAT* polynomial time reduces to *3SAT*. Instead, we modify the proof of Theorem 7.37 so that it directly produces a formula in conjunctive normal form with three literals per clause.

Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1\le i,j\le n^k} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t\in C\\ s\ne t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} .\end{aligned}$$

$$\phi_{\text{accept}} = \bigvee_{1\le i,j\le n^k} x_{i,j,q_{\text{accept}}} .$$

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}} .$$

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

3SAT is NP-Complete

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} (\text{the } (i,j) \text{ window is legal})$$

Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \left(\text{the } (i,j) \text{ window is legal}\right)$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

3SAT is NP-Complete

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \text{(the } (i,j) \text{ window is legal)}$$

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

$$\bigvee_{\substack{a_1, \ldots, a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

3SAT is NP-Complete

Now that we have written the formula in cnf, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original.

3SAT is NP-Complete

For example, we replace clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, wherein z is a new variable. If some setting of the a_i's satisfies the original clause, we can find some setting of z so that the two new clauses are satisfied. In general, if the clause contains l literals,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

we can replace it with the $l - 2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

We may easily verify that the new formula is satisfiable iff the original formula was, so the proof is complete.

CLIQUE is NP-Complete

THEOREM 7.32

3SAT is polynomial time reducible to *CLIQUE*.

PROOF IDEA The polynomial time reduction f that we demonstrate from *3SAT* to *CLIQUE* converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.

CLIQUE is NP-Complete

PROOF Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \quad \cdots \quad \wedge (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

The nodes in G are organized into k groups of three nodes each called the ***triples***, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$. The following figure illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

CLIQUE is NP-Complete

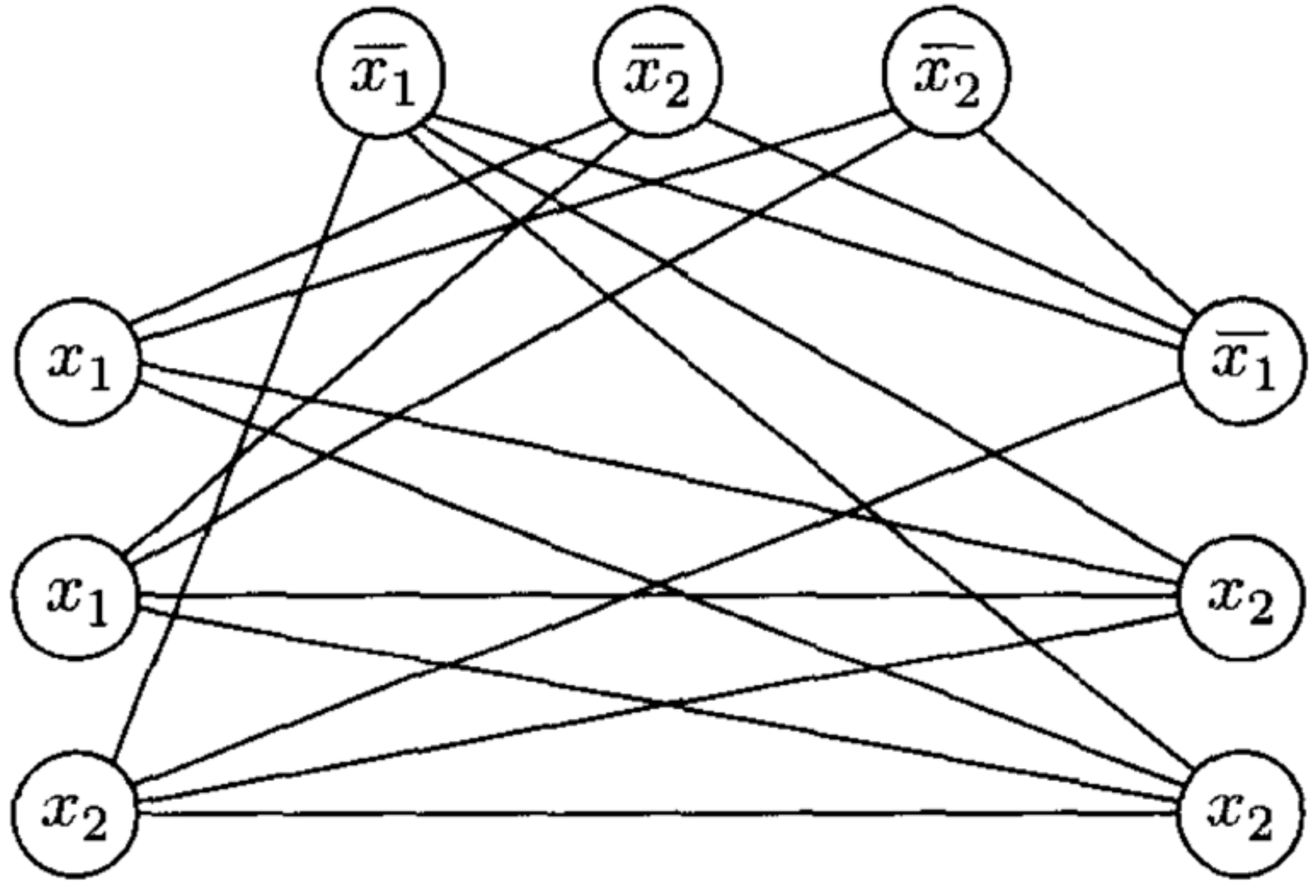


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

CLIQUE ∈ NP-Complete: ⟨ϕ⟩∈3SAT -> ⟨G,k⟩∈CLIQUE

Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a k-clique. The number of nodes selected is k, because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore G contains a k-clique.

CLIQUE ∈ NP-Complete: ⟨G,k⟩∈CLIQUE -> ⟨ϕ⟩∈3SAT

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal that is assigned TRUE. Therefore ϕ is satisfiable.

Vertex-Cover is NP-Complete

THE VERTEX COVER PROBLEM

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

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

Vertex-Cover is NP-Complete

THE VERTEX COVER PROBLEM

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

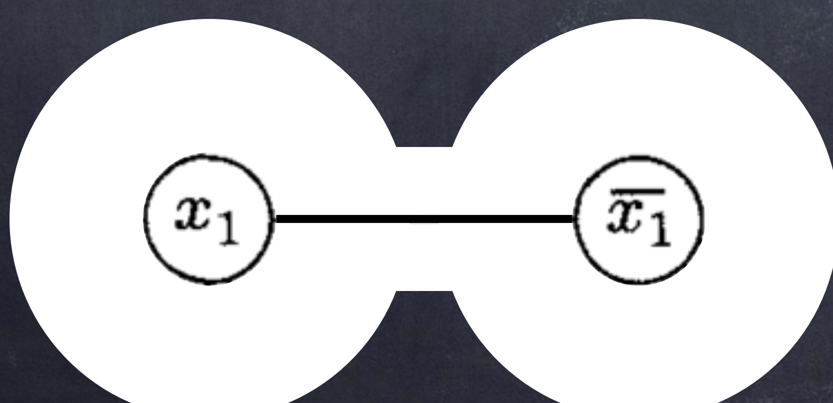
$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

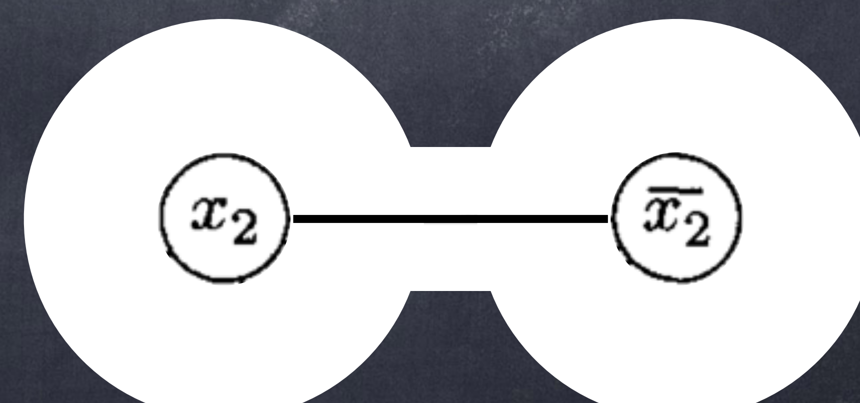
THEOREM 7.44

VERTEX-COVER is NP-complete.

Vertex-Cover is NP-Complete

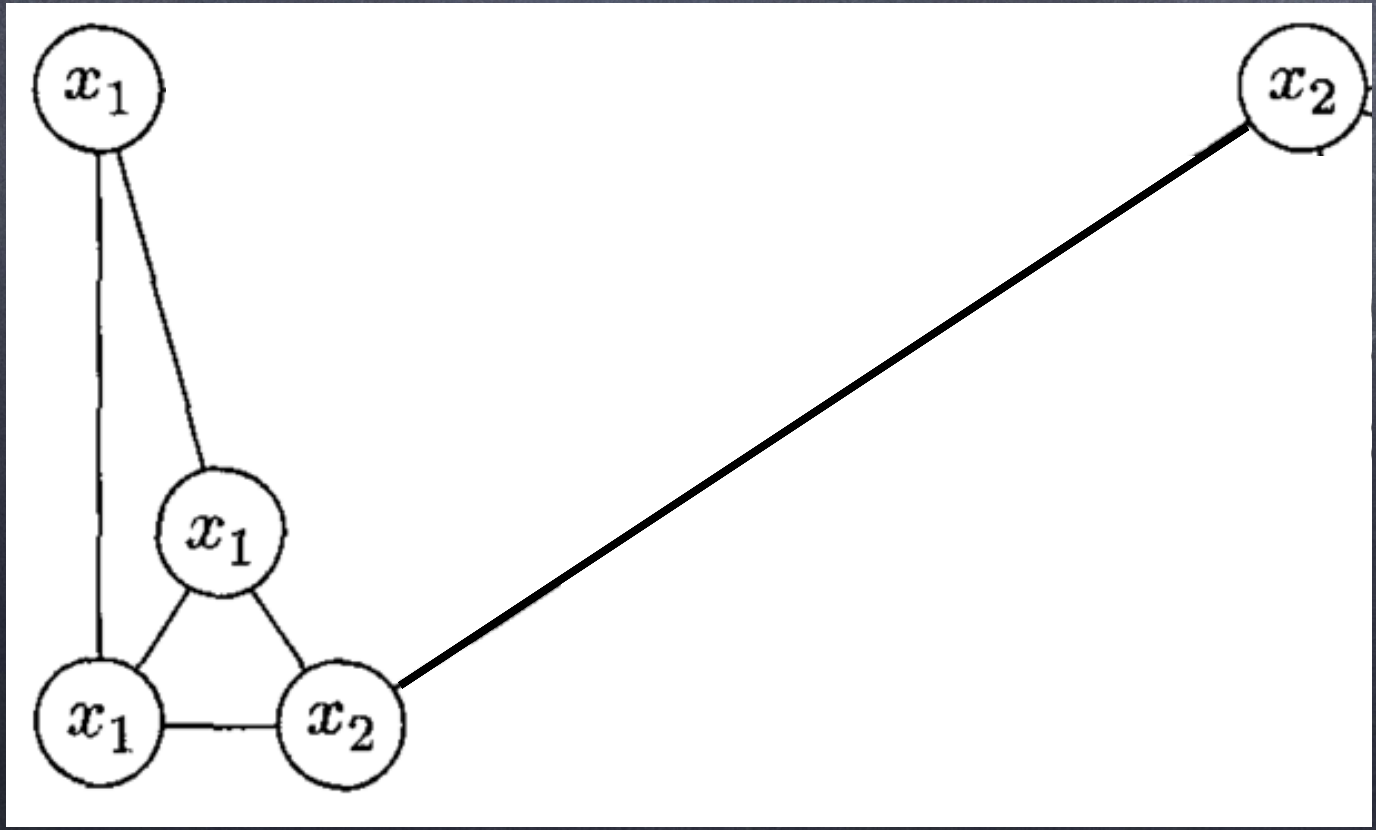
PROOF Here are the details of a reduction from *3SAT* to *VERTEX-COVER* that operates in polynomial time. The reduction maps a Boolean formula ϕ to a graph G and a value k. For each variable x in ϕ, we produce an edge connecting two nodes. We label the two nodes in this gadget x and $\overline{x}$. Setting x to be TRUE corresponds to selecting the left node for the vertex cover, whereas FALSE corresponds to the right node.





Vertex-Cover is NP-Complete

The gadgets for the clauses are a bit more complex. Each clause gadget is a triple of three nodes that are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variables gadgets that have the identical labels. Thus the total number of nodes that appear in G is $2m + 3l$, where ϕ has m variables and l clauses. Let k be $m + 2l$.



For example, if $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, the reduction produces $\langle G, k \rangle$ from ϕ, where $k = 8$ and G takes the form shown in the following figure.

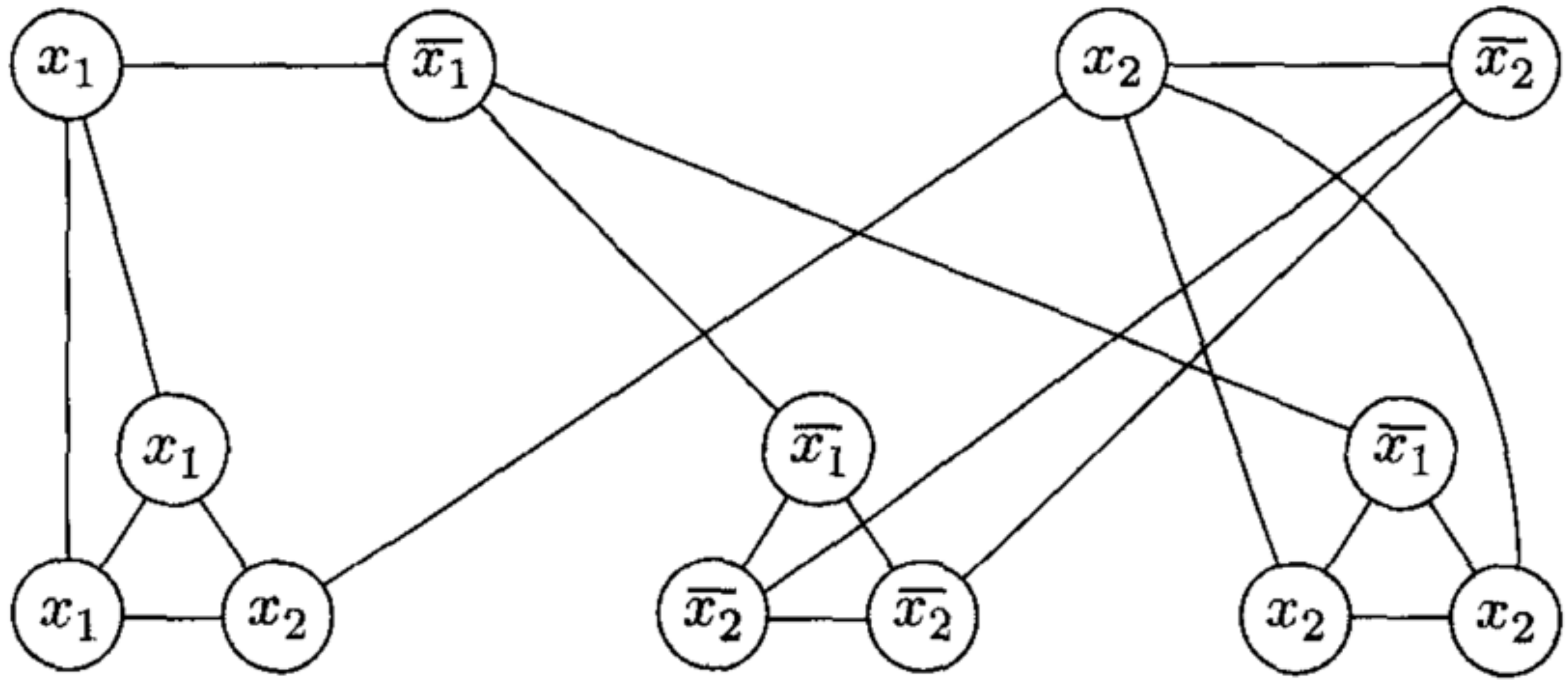


FIGURE 7.45
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Vertex-Cover ∈ NP-Complete: $\langle \phi \rangle \in$ 3SAT -> $\langle G,k \rangle \in$ V-C

To prove that this reduction works, we need to show that ϕ is satisfiable if and only if G has a vertex cover with k nodes. We start with a satisfying assignment. We first put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. Now, we have a total of k nodes. They cover all edges because every variable gadget edge is clearly covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered. Hence G has a vertex cover with k nodes.

Vertex-Cover ∈ NP-Complete: ⟨G,k⟩∈ V-C -> ⟨ϕ⟩∈ 3SAT

Second, if G has a vertex cover with k nodes, we show that ϕ is satisfiable by constructing the satisfying assignment. The vertex cover must contain one node in each variable gadget and two in every clause gadget in order to cover the edges of the variable gadgets and the three edges within the clause gadgets. That accounts for all the nodes, so none are left over. We take the nodes of the variable gadgets that are in the vertex cover and assign the corresponding literals TRUE. That assignment satisfies ϕ because each of the three edges connecting the variable gadgets with each clause gadget is covered and only two nodes of the clause gadget are in the vertex cover. Therefore one of the edges must be covered by a node from a variable gadget and so that assignment satisfies the corresponding clause.

Beyond NP-Completeness

Beyond NP-Completeness

Beyond NP-Completeness

- PSpace Completeness: problems that require a reasonable (Poly) amount of space to be solved but may use very long time though.

Beyond NP-Completeness

- PSpace Completeness: problems that require a reasonable (Poly) amount of space to be solved but may use very long time though.
- Many such problems. If any of them may be solved within reasonable (Poly) amount of time, then all of them can.

Beyond NP-Completeness

DEFINITION 8.1

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

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

Space Complexity

DEFINITION 8.2

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

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

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

THEOREM 8.5

Savitch's theorem For any[1] function $f\colon \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

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

Space Complexity

DEFINITION 8.6

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

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

We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes. However, PSPACE = NPSPACE by virtue of Savitch's theorem, because the square of any polynomial is still a polynomial.

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$

Space/Time Complexity

Space/Time Complexity

NP = PSpace ?

Space/Time Complexity

Space/Time Complexity

PSpace=EXPTime ?

Space/Time Complexity

Space/Time Complexity

P≠EXPTime

Space Complexity

DEFINITION 8.8

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

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

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

PSpace Completeness

PSpace Completeness

- Geography Game:

  Given a set of country names: Aruba, Cuba, Canada, Equador, France, Italy, Japan, Korea, Nigeria, Russia, Vietnam, Yemen.

PSpace Completeness

- Geography Game:

  Given a set of country names: Aruba, Cuba, Canada, Equador, France, Italy, Japan, Korea, Nigeria, Russia, Vietnam, Yemen.

- A two player game: One player chooses a name and crosses it out. The other player must choose a name that starts with the last letter of the previous name and so on. A player wins when his opponent cannot play any name.

Generalized Geography

Generalized Geography

- Given an arbitrary set of names: $w_1, \ldots, w_n$.

Generalized Geography

- Given an arbitrary set of names: $w_1, \ldots, w_n$.
- Is there a winning strategy for the first player to the previous game ?

Theoretical Computer Science

Theoretical Computer Science

- Challenges of TCS:

Theoretical Computer Science

- Challenges of TCS:
- FIND efficient solutions to many problems. (Algorithms and Data Structures)

Theoretical Computer Science

- Challenges of TCS:
- FIND efficient solutions to many problems. (Algorithms and Data Structures)
- PROVE that certain problems are NOT computable within a certain time or space.

Theoretical Computer Science

- <u>Challenges of TCS:</u>
- FIND efficient solutions to many problems. (Algorithms and Data Structures)
- PROVE that certain problems are NOT computable within a certain time or space.
- Consider new models of computation. (Such as a Quantum Computer)

COMP-330
Theory of Computation

Fall 2019 -- Prof. Claude Crépeau

Lec. 22-23 :
Introduction to Complexity