Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 11: Rice’s Theorem, General Grammars

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2021

Topics:

- Rice's Theorem
- Unrestricted grammars
- ... and their relationship to Turing machines
- Context-sensitive grammars
- * A glimpse beyond: Computational complexity

Recap

- *Church–Turing thesis:* Intuitive notion of algorithms $\equiv$ Turing machines.
- Formal language $\equiv$ Yes/No decision problem.
- A language is semi-decidable (also called recursively enumerable) if it can be recognised by some Turing machine.
- A language is decidable (also called recursive) if it can be recognised by some machine that halts on all inputs.
- A language is undecidable if it is not decidable.
- An undecidable language may still be semi-decidable.

- The “acceptance” decision problem for Turing machines is

  Given a Turing machine M and a string w.
  Does M accept w?

- The formal language representing this is the universal language

$$U = \{c_M w \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

- The language U is semi-decidable but not decidable.

Rice's Theorem

11.1 Rice's theorem

- Rice's Theorem states that *all* decision problems concerning the languages recognised by Turing machines[1] are undecidable.
- Let us denote the family of all semi-decidable (i.e. recursively enumerable) languages by **RE**.
- A *semantic property*[2] $\mathbf{S}$ of Turing machines is any family of semi-decidable languages, i.e. $\mathbf{S} \subseteq$ **RE**.
- A machine M *has property* $\mathbf{S}$ if $\mathcal{L}(M) \in \mathbf{S}$.
- Examples of semantic properties:
  - **NE** $= \{L \subseteq \{0,1\}^* \mid L \neq \emptyset\}$
  - **ALLSTRINGS** $= \{L \subseteq \{0,1\}^* \mid L = \{0,1\}^*\} = \{\{0,1\}^*\}$
  - **EVEN** $= \{L \subseteq \{0,1\}^* \mid |x|$ is even for all $x \in L\}$
  - **ONLY**$_w = \{L \subseteq \{0,1\}^* \mid x \in L \Leftrightarrow x = w\} = \{\{w\}\}$
  - **EMPTYSET** $= \{L \subseteq \{0,1\}^* \mid L = \emptyset\} = \{\emptyset\}$

[1] i.e. the input-output behaviours of computer programs

[2] or "specification"

- A semantic property is *trivial* if
  - $\mathbf{S} = \emptyset$ (no machine has this property) or
  - $\mathbf{S} = \mathbf{RE}$ (all machines have this property)
- A property $\mathbf{S}$ is *decidable* if the language

$$\text{codes}(\mathbf{S}) = \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \in \mathbf{S}\}$$

  is decidable.
- In other words: A semantic property is decidable if one can algorithmically decide whether a given Turing machine has the property.[3]

Theorem 11.1 (Rice 1953)

All non-trivial semantic properties of Turing machines are undecidable.

[3] equivalently "a given computer program matches the specification"

Example:

- Let us consider the *non-emptiness problem* for Turing machines from Lecture 10:

  Given a Turing machine M.

  Does the machine accept any strings?

- The corresponding semantic property is $\textbf{NE} = \{L \in \textbf{RE} \mid L \neq \emptyset\}$.
- The property is non-trivial because:
  - $\textbf{NE} \neq \emptyset$ (witness any semi-decidable language $L \neq \emptyset$)
  - $\textbf{NE} \subsetneq \textbf{RE}$ (since $\emptyset \in \textbf{RE} \setminus \textbf{NE}$)
- Thus by Rice's theorem, the language

$$\begin{aligned}\text{codes}(\textbf{NE}) &= \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \in \textbf{NE}\} \\ &= \{c_M \mid M \text{ is a Turing machine and } \mathcal{L}(M) \neq \emptyset\}\end{aligned}$$

  is undecidable. (Note that this is precisely the result in Lemma 10.5.)

Theorem 11.1

All non-trivial semantic properties of Turing machines are undecidable.

Proof

- A simple generalisation of the proof of Lemma 10.5.
- Let $\mathbf{S}$ be any non-trivial semantic property.
- We can assume that $\emptyset \notin \mathbf{S}$; in other words, machines that recognise the empty language do not have the property.[a]
- As $\mathbf{S}$ is non-trivial, there is a Turing machine $M_{\mathbf{S}}$ that has the property $\mathbf{S}$, i.e. one for which $\mathcal{L}(M_{\mathbf{S}}) \neq \emptyset$ and $\mathcal{L}(M_{\mathbf{S}}) \in \mathbf{S}$ hold.

[a] If $\emptyset \in \mathbf{S}$, we can first show that the property $\bar{\mathbf{S}} = \textbf{RE} \setminus \mathbf{S}$ is undecidable and then conclude that also $\mathbf{S}$ is undecidable because if we could decide codes$(\mathbf{S})$, we could also decide codes$(\bar{\mathbf{S}})$ as $c_M \in$ codes$(\bar{\mathbf{S}})$ iff $c_M \notin$ codes$(\mathbf{S})$.

- We now prove that codes$(\mathbf{S})$ is undecidable by reducing the undecidable language U to it.
- Let (M,w) be any instance of the Turing machine acceptance problem, encoded as the string $c_M w$.
- From input $c_M w$ construct (the code for) a Turing machine M^w that on any input string x works as follows:
  - First run machine M on string w, and then
    - if M accepts w, run $M_{\mathbf{S}}$ on x
    - if M rejects w (or doesn't halt), reject x (or don't halt)
- Now M^w recognises the language

$$\mathcal{L}(M^w) = \begin{cases} \mathcal{L}(M_{\mathbf{S}}) & \text{if } w \in \mathcal{L}(M) \\ \emptyset & \text{if } w \notin \mathcal{L}(M) \end{cases}$$

- Thus M accepts w if and only if M^w has the property $\mathbf{S}$. That is, $c_M w \in U$ if and only if $c_{M^w} \in$ codes$(\mathbf{S})$.
- Therefore, codes$(\mathbf{S})$ is an undecidable language.

General Grammars

11.2 Unrestricted grammars

- A generalisation of context-free grammars.
- The left-hand sides of rules can now include multiple symbols.
- As will be shown, can generate all semi-decidable languages.

Definition 11.1

An *unrestricted grammar* is a quadruple[a]

$$G = (V, \Sigma, P, S),$$

where

- V is a finite set of *variables*;
- Σ is a finite set, disjoint from V, of *terminals*;
- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ is a finite set of *rules* (also called productions), where $(V \cup \Sigma)^+ = (V \cup \Sigma)^* \setminus \{\varepsilon\}$;
- $S \in V$ is the *start variable*.

A rule $(\omega, \omega') \in P$ is usually written as $\omega \to \omega'$.

[a] Note the minor streamlining of the structure of the definition from Lecture 5.

- A string $\gamma \in (V \cup \Sigma)^*$ *yields* a string $\gamma' \in (V \cup \Sigma)^*$ in the grammar G, denoted by

$$\gamma \underset{G}{\Rightarrow} \gamma'$$

  if

  - the grammar contains a rule $\omega \to \omega'$ such that
  - $\gamma = \alpha\omega\beta$ and $\gamma' = \alpha\omega'\beta$ for some $\alpha, \beta \in (V \cup \Sigma)^*$.

- A string $\gamma \in (V \cup \Sigma)^*$ *derives* a string $\gamma' \in (V \cup \Sigma)^*$ in the grammar G, denoted by

$$\gamma \underset{G}{\Rightarrow}^* \gamma'$$

  if there is a sequence of strings $\gamma_0, \gamma_1, \ldots, \gamma_n$ for some $n \geq 0$ such that

$$\gamma = \gamma_0, \qquad \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n, \qquad \gamma_n = \gamma'.$$

- If the grammar G is clear from the context, we can simply write $\gamma \Rightarrow \gamma'$ and $\gamma \Rightarrow^* \gamma'$ instead of $\gamma \underset{G}{\Rightarrow} \gamma'$ and $\gamma \underset{G}{\Rightarrow}^* \gamma'$, respectively.

Example:

An unrestricted grammar for the non-context-free language $\{a^k b^k c^k \mid k \geq 0\}$:

$$
\begin{array}{rcl}
S & \to & LT \mid \varepsilon \\
T & \to & ABCT \mid ABC \\
BA & \to & AB \\
CB & \to & BC \\
CA & \to & AC
\end{array}
\qquad
\begin{array}{rcl}
LA & \to & a \\
aA & \to & aa \\
aB & \to & ab \\
bB & \to & bb \\
bC & \to & bc \\
cC & \to & cc
\end{array}
$$

A derivation of string $aabbcc$ in the grammar:

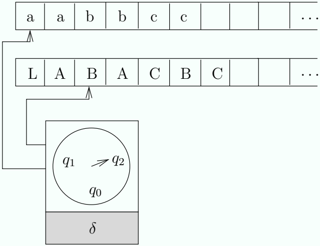
$$
\begin{array}{lllllll}
\underline{S} & \Rightarrow & L\underline{T} \Rightarrow LABC\underline{T} & \Rightarrow & LABC\underline{A}BC & \Rightarrow & LA\underline{BA}CBC \\
 & \Rightarrow & LAAB\underline{CB}C & \Rightarrow & \underline{LA}ABBCC & \Rightarrow & \underline{aA}BBCC \\
 & \Rightarrow & a\underline{aB}BCC & \Rightarrow & aa\underline{bB}CC & \Rightarrow & aab\underline{bC}C \\
 & \Rightarrow & aabb\underline{cC} & \Rightarrow & aabbcc & &
\end{array}
$$

Theorem 11.2

If a language L can be generated with an unrestricted grammar, then it can be recognised with a Turing machine.

Proof

Let $G = (V, \Sigma, P, S)$ be an unrestricted grammar generating language L. We can design a two-tape nondeterministic Turing machine M_G recognising L as follows:



- On tape 1 the machine stores a copy of the input string.
- Tape 2 holds the current string that the machine tries to rewrite to match the one on tape 1.
- In the beginning, the machine writes the start variable S on tape 2.

The computation of machine M_G is composed of stages. In each stage, the machine:

1. Moves the read/write-head of tape 2 *nondeterministically* to *some* position on the tape.
2. Chooses *nondeterministically* a rule in G that it tries to apply at the selected position. (The rules of G are encoded in the transitions of M_G.)
3. If the left-hand side of the chosen rule matches the symbols on the tape, M_G rewrites these symbols with the ones in the right-hand side of the rule. Otherwise M_G rejects.
4. At the end of the stage, M_G compares the strings on tapes 1 and 2. If they are the same, the machine acceps and halts. Otherwise, the machine executes the next stage (loops back to step 1).

Theorem 11.3

If a language L can be recognised with a Turing machine, then it can be generated with an unrestricted grammar.

Proof

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a (deterministic one-tape) Turing machine recognising language L. We can design an unrestricted grammar G_M generating L based on the following idea.

- The variables of G_M include (among others) symbols for all the states $q \in Q$ of M.
- A configuration $(q, u\underline{a}v)$ of M will be represented as a string $[uqav]$.
- Based on the transitions of M, G_M will have rules that ensure

$$[uqav] \underset{G_M}{\Rightarrow} [u'q'a'v'] \quad \text{if and only if} \quad (q, u\underline{a}v) \underset{M}{\vdash} (q', u'\underline{a'}v').$$

- Thus M accepts the input x if and only if for some $u, v \in \Sigma^*$:

$$[q_0 x] \underset{G_M}{\Rightarrow}^* [uq_{\text{acc}}v]$$

The rules in G_M comprise three types:

1. Rules with which one can derive from the start variable S any string of form $x[q_0x]$, where $x \in \Sigma^*$ and '$[$', 'q_0' and '$]$' are variables in G_M.
2. Rules that allow one to derive from the string $[q_0x]$ a string $[uq_{\text{acc}}v]$ if and only if M accepts x.
3. Rules that enable one to rewrite any string of form $[uq_{\text{acc}}v]$ to the empty string.

Deriving a string $x \in \mathcal{L}(M)$ can then be done as follows:

$$S \overset{(1)}{\Rightarrow^*} x[q_0x] \overset{(2)}{\Rightarrow^*} x[uq_{\text{acc}}v] \overset{(3)}{\Rightarrow^*} x$$

Let us thus define the grammar $G = (V, \Sigma, P, S)$, where

$$V = (\Gamma \setminus \Sigma) \cup Q \cup \{S, T, [,], E_L, E_R\} \cup \{X_a \mid a \in \Sigma\}$$

and the rules in P include the following three sets:

1. Producing the initial configuration:

$$\begin{array}{lcll} S & \to & T[q_0] & \\ T & \to & \varepsilon & \\ T & \to & aTX_a & (a \in \Sigma) \\ X_a[q_0 & \to & [q_0X_a & (a \in \Sigma) \\ X_ab & \to & bX_a & (a, b \in \Sigma) \\ X_a] & \to & a] & (a \in \Sigma) \end{array}$$

2. Simulating the transitions of M ($a,b \in \Gamma$, $c \in \Gamma \cup \{\,[\,\}$):

Transitions:	*Rules:*
$\delta(q,a) = (q',b,R)$	$qa \rightarrow bq'$
$\delta(q,a) = (q',b,L)$	$cqa \rightarrow q'cb$
$\delta(q,\triangleright) = (q',\triangleright,R)$	$q[\ \rightarrow\ [q'$
$\delta(q,\triangleleft) = (q',b,R)$	$q]\ \rightarrow\ bq']$
$\delta(q,\triangleleft) = (q',b,L)$	$cq]\ \rightarrow\ q'cb]$
$\delta(q,\triangleleft) = (q',\triangleleft,L)$	$cq]\ \rightarrow\ q'c]$

3. Emptying an accepting configuration:

$$
\begin{array}{lcll}
q_{\text{acc}} & \to & E_L E_R & \\
q_{\text{acc}}[& \to & E_R & \\
aE_L & \to & E_L & (a \in \Gamma) \\
[E_L & \to & \varepsilon & \\
E_R a & \to & E_R & (a \in \Gamma) \\
E_R] & \to & \varepsilon &
\end{array}
$$

11.3 Context-sensitive grammars

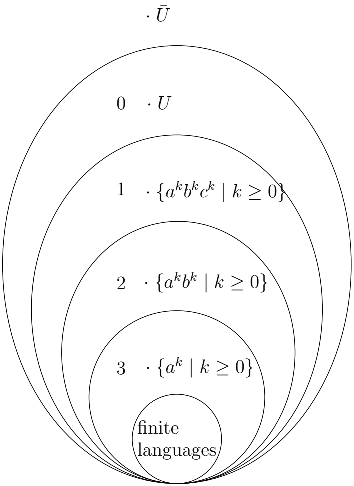
- An unrestricted grammar is *context-sensitive* if all its rules are of form $\omega \to \omega'$, where $|\omega'| \geq |\omega|$, or $S \to \varepsilon$, where S is the start variable.
- In addition, it is required that if the grammar has the rule $S \to \varepsilon$, then the start variable S does not occur on the right-hand side of any rule.
- A language L is *context-sensitive* if it can be generated with some context-sensitive grammar.
- *A normal form for context-sensitive grammars:* Each context-sensitive language can be generated with a grammar whose rules are of form $S \to \varepsilon$ and $\alpha A \beta \to \alpha \omega \beta$, where A is a variable and $\omega \neq \varepsilon$.
- A rule $\alpha A \beta \to \alpha \omega \beta$ can be interpreted as the application of a rule $A \to \omega$ "in the context" α_β.

Theorem 11.4

A language L is context-sensitive if and only if it can be recognised with a non-deterministic Turing machine that does not use more tape space than was already allocated for the input.

- The machines in Theorem 11.4 are called *linear bounded automata*.
- It is an open problem whether the non-determinism in Theorem 11.4 is necessary or not. (The “LBA ?= DLBA” problem.)

11.4 Recap: The Chomsky hierarchy



A classification of grammars, languages generated by grammars and recogniser automata classes:

Type-0: unrestricted grammars / semi-decidable languages / Turing machines

Type-1: context-sensitive grammars / context-sensitive languages / linear bounded automata

Type-2: context-free grammars / context-free languages / pushdown automata

Type-3: right and left linear grammars / regular languages / finite automata

* A Glimpse Beyond: Computational Complexity

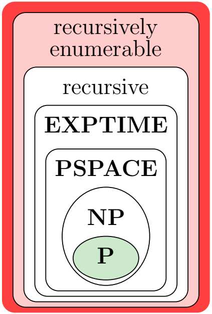
* Computational complexity

- So far: only what is decidable (solvable with computers) and what is not.
- But some problems are “more decidable than others”.
- For instance, finding a smallest element in an array is/seems much easier than solving sudokus.

102							

10		16			12				15				4	5	1
	11		14	13			5	10				16			
			12									11			
	1		9			7	4			11	8	13		12	
5	10	14				11				9		3		4	
		9	7		4	6					15	1	11	13	16
				16		5	3						2	15	9
			6				7	2							
14		13		1		2		9		16		8	6		
16				7	14	9		8	1		2	5			
2		8			6	4		13	3		5	14		1	
			4								7				
			16	14						1		12			7
			11						14	5					
		2			10		6	11	7		13	9	5		
3		12	15									2		10	

- In fact, the set of decidable problems can be divided in many smaller *complexity classes*:

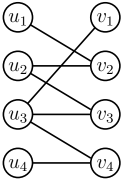


- **P** — problems that can be solved in polynomial time ($\approx$ always efficiently) with deterministic Turing machines / algorithms.
- **NP** — problems that can be solved in polynomial time with *non-deterministic Turing machines*.
- **PSPACE** — problems that can be solved with a polynomial amount of extra space (possibly in exponential time).
- **EXPTIME** — problems that can be solved in exponential time.
- and many more...

Example: a nontrivial, but efficiently solvable problem

Definition (PERFECT MATHING)

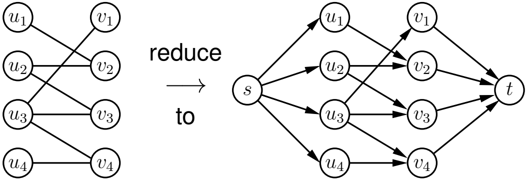
INSTANCE: Bipartite graph $B = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, and $E \subseteq U \times V$.
QUESTION: Does B have a *perfect matching*, i.e. a 1-1 pairing of vertices?



Example: a nontrivial, but efficiently solvable problem

Definition (PERFECT MATHING)

INSTANCE: Bipartite graph $B = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, and $E \subseteq U \times V$.
QUESTION: Does B have a *perfect matching*, i.e. a 1-1 pairing of vertices?



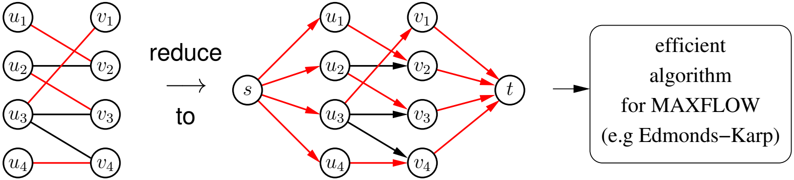
We can solve a PERFECT MATCHING instance by

1. *Polynomial-time reducing* it to a MAXFLOW instance so that: the MAXFLOW instance has a flow of n units if and only if the PERFECT MATCHING instance has a perfect matching.

Example: a nontrivial, but efficiently solvable problem

Definition (PERFECT MATHING)

INSTANCE: Bipartite graph $B = (U, V, E)$, where $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$, and $E \subseteq U \times V$.
QUESTION: Does B have a *perfect matching*, i.e. a 1-1 pairing of vertices?



We can solve a PERFECT MATCHING instance by

1. *Polynomial-time reducing* it to a MAXFLOW instance so that: the MAXFLOW instance has a flow of n units if and only if the PERFECT MATCHING instance has a perfect matching.
2. Solving the resulting MAXFLOW instance.
3. The reduction is linear-time and Edmonds-Karp alg. works in $O(VE^2)$.

Example: a not-so efficiently solvable problem

Definition (propositional satisfiability, SAT)

INSTANCE: A Boolean formula ϕ in conjunctive normal form.
QUESTION: Is there a truth assignment that satisfies ϕ?

Example

$(x) \land (\neg x \lor y) \land (\neg x \lor \neg z) \land (\neg x \lor \neg y \lor \neg z)$ is satisfiable with $\{x \mapsto \mathbf{true}, y \mapsto \mathbf{true}, z \mapsto \mathbf{false}\}$.

$(x) \land (\neg x \lor y) \land (\neg x \lor \neg z) \land (\neg x \lor \neg y \lor z)$ is unsatisfiable.

- Even the best known SAT algorithms, with sophisticated pruning techniques can perform very badly on some instances (although they can solve many relevant problems efficiently).
- No polynomial-time algorithm for SAT is known despite several decades of effort in trying to find one.

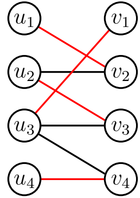
Problem class NP (Non-deterministic Polynomial time)

Two alternative ways to characterise problems in **NP**:

1. Problems that can be solved in *polynomial time* with *non-deterministic* Turing machines ($\approx$ algorithms that can guess perfectly).
2. Problems whose solutions (when they exist) are
   - reasonably small (i.e., of polynomial size), and
   - easy to check (i.e., in polynomial time).

   but *not necessarily easy to find* (or prove non-existent)!

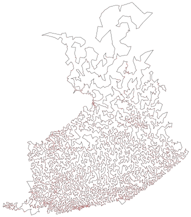
PERFECT MATCHING



SAT

$$
\begin{gathered}
(x \vee y \vee \neg z) \wedge \\
(\neg x \vee \neg y \vee v) \wedge \\
(x \vee w \vee z) \wedge \\
(y \vee \neg w \vee \neg z) \wedge \\
\dots
\end{gathered}
$$

TRAVELLING SALESPERSON

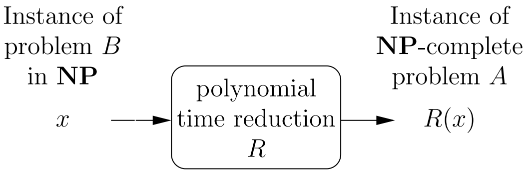


GENERALISED SUDOKUS

10		16			12				15				4	5	1
	11		14	13			5	10				16			
			12									11			
	1		9			7	4			11	8	13		12	
5	10	14				11				9		3		4	
		9	7		4	6					15	1	11	13	16
				16		5	3						2	15	9
			6				7	2							
14		13		1		2		9		16		8	6		
16				7	14	9		8	1		2	5			
2		8			6	4		13	3		5	14		1	
			4								7				
			16	14						1		12			7
			11						14	5					
		2			10		6	11	7		13	9	5		
3		12	15									2		10	

NP-complete problems

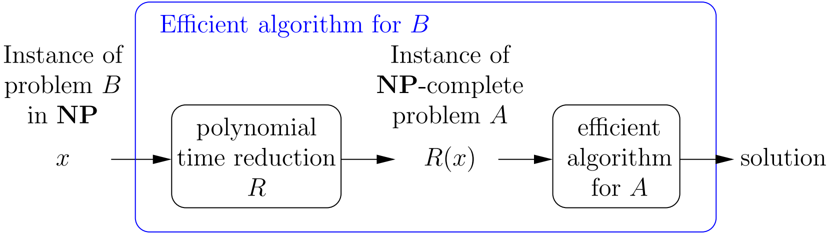
- A problem A in **NP** is **NP**-*complete* if every other problem B in **NP** can be reduced to it with a polynomial time computable reduction.



Property: x has a solution in B if and only if $R(x)$ has a solution in A.

NP-complete problems

- A problem A in **NP** is **NP**-*complete* if every other problem B in **NP** can be reduced to it with a polynomial time computable reduction.



Property: x has a solution in B if and only if $R(x)$ has a solution in A.

☞ If an **NP**-complete problem A can be solved in polynomial time, then *all the problems in* **NP** *can*.

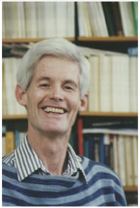
☞ **NP**-complete problems are the *most difficult ones* in **NP**!

☞ We *do not know*(!!!) whether **NP**-complete problems can be solved efficiently or not.

The Cook–Levin theorem

Theorem (S. A. Cook 1971, L. Levin 1973)

SAT is **NP***-complete.*



Stephen Cook (1939–)

Leonid Levin (1948–)

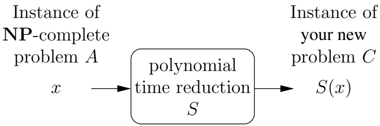
Richard Karp (1935–)

- R. Karp soon (1972) listed the next 21 **NP**-complete problems.
- Since then, 1000's of problems have been shown **NP**-complete.
- E.g. TRAVELLING SALESPERSON, GENERALISED SUDOKUS etc. are **NP**-complete.
- Classic text: Garey and Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

How to prove a new problem NP-complete?

Given: a new problem C that you suspect **NP**-complete.
To prove that C is **NP**-complete:

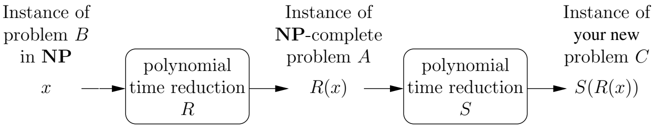
1. show that C is in **NP**,
2. take any *existing* **NP***-complete problem* A, and
3. reduce A *to your problem* C.



How to prove a new problem NP-complete?

Given: a new problem C that you suspect **NP**-complete.
To prove that C is **NP**-complete:

1. show that C is in **NP**,
2. take any *existing* **NP***-complete problem* A, and
3. reduce A *to your problem* C.



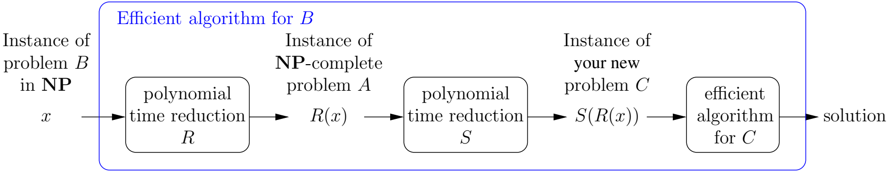
Polynomial time reductions compose: any B in **NP** reduces to C!

☞ Your problem C is **NP**-complete.

How to prove a new problem NP-complete?

Given: a new problem C that you suspect **NP**-complete.
To prove that C is **NP**-complete:

1. show that C is in **NP**,
2. take any *existing* ***NP****-complete problem* A, and
3. reduce A *to your problem* C.



Polynomial time reductions compose: any B in **NP** reduces to C!

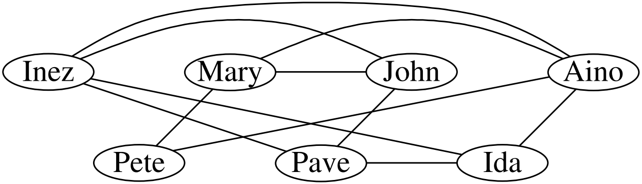
☞ Your problem C is **NP**-complete.

☞ If your problem C can be solved in polynomial time, then so can A and all the problems in **NP**.

Proving NP-completeness: an example

Definition (PARTYING WITH STRANGERS)

INSTANCE: A network of students and a positive integer K, where a network consists of (i) a finite set of students and (ii) a symmetric, binary "X knows Y" relation among them.
QUESTION: Is it possible to arrange a party with (at least) K students, none of whom know each other?

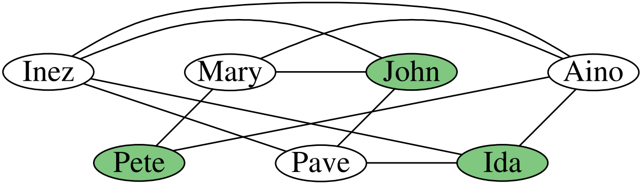


with $K = 3$?

Proving NP-completeness: an example

Definition (PARTYING WITH STRANGERS)

INSTANCE: A network of students and a positive integer K, where a network consists of (i) a finite set of students and (ii) a symmetric, binary "X knows Y" relation among them.
QUESTION: Is it possible to arrange a party with (at least) K students, none of whom know each other?

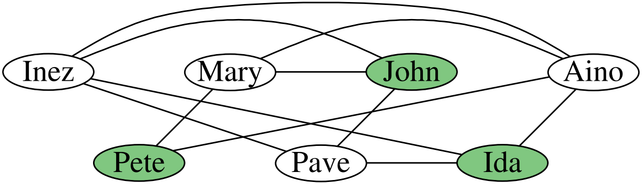


with $K = 3$?

Proving NP-completeness: an example

Definition (PARTYING WITH STRANGERS)

INSTANCE: A network of students and a positive integer K, where a network consists of (i) a finite set of students and (ii) a symmetric, binary "X knows Y" relation among them.
QUESTION: Is it possible to arrange a party with (at least) K students, none of whom know each other?



with $K = 3$?

Definition (INDEPENDENT SET)

INSTANCE: An undirected graph $G = (V, E)$ and an integer K.
QUESTION: Is there an independent set $I \subseteq V$ with $|I| = K$?

Theorem

INDEPENDENT SET *is* **NP**-*complete.*

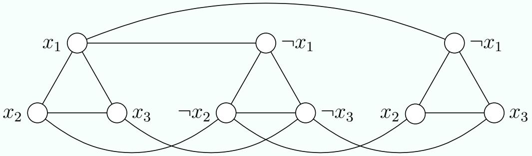
Proof

Reduction *from* 3SAT.

The SAT formula ϕ:

$(x_1 \lor x_2 \lor x_3) \land$
$(\neg x_1 \lor \neg x_2 \lor \neg x_3) \land$
$(\neg x_1 \lor x_2 \lor x_3)$

The corresponding graph G with $K = 3$:



1. If ϕ is satisfiable, then G has an independent set of size K.
2. If G has an independent set of size K, then ϕ is satisfiable.

$\Rightarrow$ ϕ is satisfiable if and only if G has an independent set of size K.

☞ **If we can solve INDEPENDENT SET efficiently, then we can solve SAT and all other problems in NP efficiently as well.**

Theorem

INDEPENDENT SET *is* **NP**-*complete.*

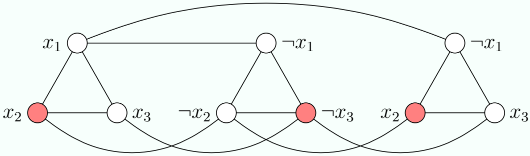
Proof

Reduction *from* 3SAT.

The SAT formula ϕ:

$(x_1 \vee x_2 \vee x_3) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee x_2 \vee x_3)$

The corresponding graph G with $K = 3$:



1. If ϕ is satisfiable, then G has an independent set of size K.
2. If G has an independent set of size K, then ϕ is satisfiable.

$\Rightarrow$ ϕ is satisfiable if and only if G has an independent set of size K.

☞ **If we can solve INDEPENDENT SET efficiently, then we can solve SAT and all other problems in NP efficiently as well.**

Theorem

INDEPENDENT SET *is* **NP**-*complete.*

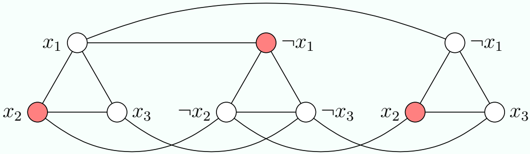
Proof

Reduction *from* 3SAT.

The SAT formula ϕ:

$(x_1 \vee x_2 \vee x_3) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee x_2 \vee x_3)$

The corresponding graph G with $K = 3$:



1. If ϕ is satisfiable, then G has an independent set of size K.
2. If G has an independent set of size K, then ϕ is satisfiable.

$\Rightarrow$ ϕ is satisfiable if and only if G has an independent set of size K.

If we can solve INDEPENDENT SET efficiently, then we can solve SAT and all other problems in NP efficiently as well.

Theorem

INDEPENDENT SET *is* **NP***-complete.*

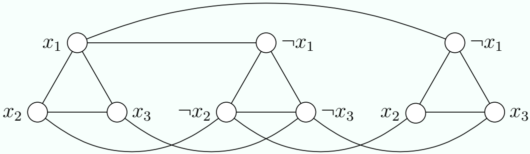
Proof

Reduction *from* 3SAT.

The SAT formula ϕ:

$(x_1 \vee x_2 \vee x_3) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee x_2 \vee x_3)$

The corresponding graph G with $K = 3$:



1. If ϕ is satisfiable, then G has an independent set of size K.
2. If G has an independent set of size K, then ϕ is satisfiable.

$\Rightarrow$ ϕ is satisfiable if and only if G has an independent set of size K.

☞ **If we can solve INDEPENDENT SET efficiently, then we can solve SAT and all other problems in NP efficiently as well.**

NP-completeness: Significance

- *Can* **NP***-complete problems be solved in polynomial time?*

  One of the seven 1M$ Clay Mathematics Institute Millenium Prize problems, see

  http://www.claymath.org/millennium-problems/

- What to do when a problem is **NP**-complete?
  - Attack special cases that occur in practice
  - Develop backtracking search algorithms with efficient heuristics and pruning techniques
  - Develop approximation algorithms
  - Apply incomplete local search methods
  - ...

Some further courses:

- CS-E3190 Principles of Algorithmic Techniques
- CS-E4530 Computational Complexity Theory
- CS-E4320 Cryptography and Data Security
- and so on...