CS125	Lecture 14	Fall 2014

NP-Completeness

The World if $\mathrm{P} \neq \mathrm{NP}$?

Q: If $\mathrm{P} \neq \mathrm{NP}$, can we conclude anything about any specific problems?

Idea: Try to find a "hardest" NP language.

- Want $L \in \mathrm{NP}$ such that $L \in \mathrm{P}$ iff every NP language is in P.

Reducibility

Informally, we say that a computational problem A reduces to a computational problem B (written $A \leq B$) if A can be solved (efficiently) by solving B. Thus, an (efficient) algorithm for B implies an (efficient) algorithm for A.

We have already seen many examples:

- Context-free Recognition $\leq$ Matrix Multiplication (HW3)
- Max-Flow $\leq$ Linear Programming
- Matching $\leq$ Max-Flow
- Zero-Sum Games $\leq$ Linear Programming
- $L_{\text{fact}} \leq$ Factoring (HW4)
- Factoring $\leq L_{\text{fact}}$ (HW4)

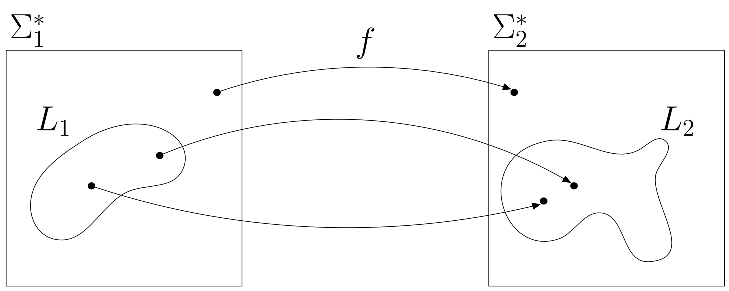
As the last bullet shows, reductions are useful not only for showing that problems can be solved efficiently, but also for giving evidence that problems are hard: under the widely believed conjecture that Factoring has no polynomial-time algorithm, we can deduce that $L_{\text{fact}} \notin \mathrm{P}$ (and hence $\mathrm{P} \neq \mathrm{NP}$). Hence "$A \leq B$" can be interpreted equivalently as saying "A is at least as easy as B" or "B is at least as hard as A".

Polynomial-Time Mapping Reductions

There are many forms of reducibility, and which one is most suitable depends on what kind of computational phenomena we are interested in studying. A very general notion is that of a *Turing reduction* (aka *oracle reduction*), where we say that $A \leq B$ if there is an algorithm that solves A given any "black box" that solves B. (For example, we add a Word-RAM instruction that will provide a solution to an instance of B written in memory in one time step. It's like programming with a library for which we have no idea how the the library functions themselves are implemented (or even if they can be implemented at all).) The polynomial-time analogue of Turing reductions are known as *Cook reductions*, and these are what we used in the reductions between FACTORING and L_{fact}.

However, for reductions between *languages*, it is often convenient to work with the following more restrictive notion of reduction (known as *polynomial-time mapping reductions* or *Karp reductions*):

Def: $L_1 \leq_P L_2$ iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ s.t. for every $x \in \Sigma_1^*$, $x \in L$ iff $f(x) \in L_2$.



- $x \in L_1 \Rightarrow f(x) \in L_2$
- $x \notin L_1 \Rightarrow f(x) \notin L_2$
- f computable in polynomial time

Proposition: If $L_1 \leq_P L_2$ and $L_2 \in$ P, then $L_1 \in$ P.

Proof:

NP-Completeness

Def: L is NP-complete iff

1. $L \in$ NP and
2. For every $L' \in$ NP, we have $L' \leq_P L$. ("L is NP-hard")

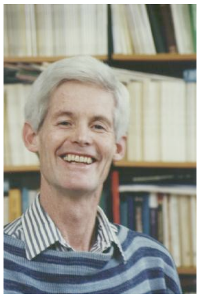
Prop: Let L be any NP-complete language. Then P = NP *if and only if* $L \in$ P.

Cook–Levin Theorem

(Stephen Cook 1971, Leonid Levin 1973)

Theorem: SAT (Boolean satisfiability) is NP-complete.

Proof: Need to show that every language in NP reduces to SAT (!) Proof next time.



More NP-complete problems

From now on we prove NP-completeness using:

Lemma: If we have the following

- L is in NP
- $L_0 \leq_P L$ for some NP-complete L_0

Then L is NP-complete.

Proof:

3-SAT

Def: A Boolean formula is in 3-CNF if it is of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where each clause C_i is a disjunction ("or") of 3 literals:

$$C_i = (C_{i1} \vee C_{i2} \vee C_{i3})$$

where each literal C_{ij} is either a variable x, or the negation of a variable, $\neg x$ (sometimes written $\overline{x}$).

e.g. $(x \vee y \vee z) \wedge (\neg x \vee \neg u \vee w) \wedge (u \vee u \vee u)$

3-SAT is the set of satisfiable 3-CNF formulas.

Theorem: 3-SAT is NP-complete

Proof: We show that SAT $\leq_P$ 3-SAT.

1. Given an arbitrary Boolean formula, e.g.

   $F = (\neg((x \vee \neg y) \wedge (z \vee w)) \vee \neg x).$
   1 2 3 4 5 6 7

2. Number the operators.

3. Select a new variable a_i for each operator.

   The variable a_i is supposed to mean "the subformula rooted at operator i is true."

4. Write a formula F_1 stating the relation between each subformula and its children subformulas.

   For example, where

   $F = (\neg((x \vee \neg y) \wedge (z \vee w)) \vee \neg x),$
   1 2 3 4 5 6 7

$$F_1 = \begin{pmatrix} & (a_3 \equiv \neg y) & \wedge & (a_7 \equiv \neg x) \\ \wedge & (a_2 \equiv x \vee a_3) & \wedge & (a_1 \equiv \neg a_4) \\ \wedge & (a_5 \equiv z \vee w) & \wedge & (a_6 \equiv a_1 \vee a_7) \\ \wedge & (a_4 \equiv a_2 \wedge a_5) & & \end{pmatrix}$$

5. Let k be the number of the main operator/subformula of F.

   (Note: $k = 6$ in the example)

   Claim: $a_k \wedge F_1$ is satisfiable iff F is satisfiable.

   Proof:

6. Write F_1 in 3-CNF to obtain F_2.

   Fact: Every function $f : \{0,1\}^k \to \{0,1\}$ can be written as a k-CNF and as a k-DNF (OR of ANDs). [albeit with possibly 2^k clauses]

   Proof:

7. Output of the reduction: $a_k \wedge F_2$.

Q: Does this prove that every Boolean formula can be converted to 3-CNF?

In contrast, 2-SAT $\in$ P

<u>Method</u> (resolution):

1. If x and $\neg x$ are both clauses, then <u>not</u> satisfiable

   e.g. $(x) \wedge (z \vee y) \wedge (\neg x)$

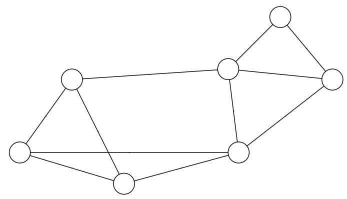
2. If $(x \vee y) \wedge (\neg y \vee z)$ are both clauses, add clause $(x \vee z)$ (which is implied).
3. Repeat. If no contradiction emerges $\Rightarrow$ satisfiable.

$O(n^2)$ repetitions of step 2 since only 2 literals/clause.

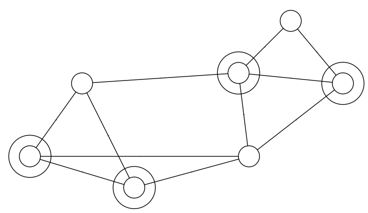
Proof of correctness: omitted

VERTEX COVER (VC)

- Instance:
  - a graph, e.g.



  - a number k (e.g. 4)
- Question: Is there a set of k vertices that "cover" the graph, i.e., include at least one endpoint of every edge?

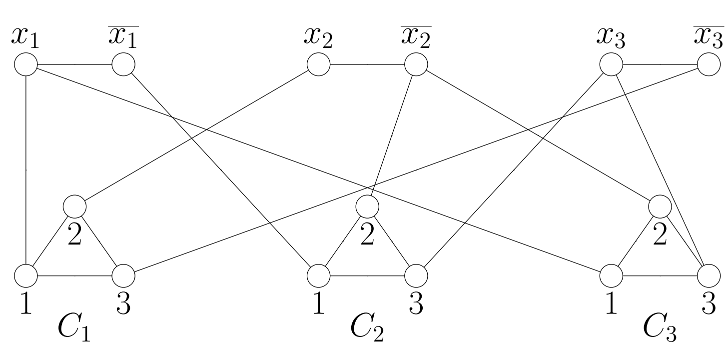


VC is NP-complete

- VC is in NP:
- 3-SAT $\leq_P$ VC:
  - Let F be a 3-CNF formula with clauses $C_1 \ldots, C_m$, variables $x_1, \ldots, x_n$.
  - We construct a graph G_F and a number N_F such that:

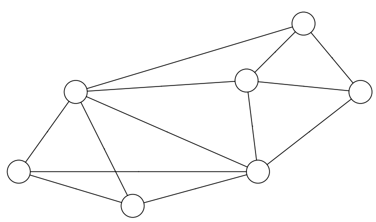
G_F has a size N_F vertex cover iff F is satisfiable

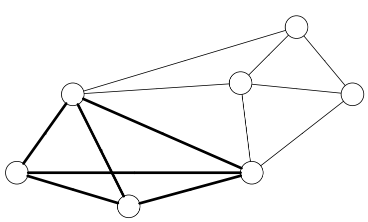
E.g. $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$



- G_F = one dumbbell for each variable, one triangle for each clause, and corner j of triangle i is connected to the vertex representing the jth literal in C_i.
- $N_F = 2m + n = 2$ (# clauses) + (# variables).
  $\Rightarrow$ 1 vertex from each dumbbell and 2 from each triangle.
- If F is satisfiable, then there is a cover of size N_F:
- If there is a cover of size N_F, then F is satisfiable:

CLIQUE

- Instance:
  - a graph, e.g. 
  - a number k (e.g. 4)
- Question: Is there a clique of size k, i.e., a set of k vertices such that there is an edge between each pair?



- Easy to see that CLIQUE $\in$ NP.

VC $\leq_P$ CLIQUE

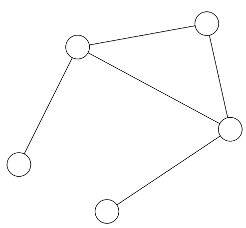
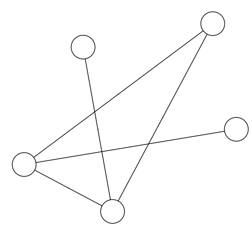
If G is any graph, let G^c be the graph with the same vertices such that:

there is an edge between x and y in G^c

iff

there is no edge between x and y in G

e.g.

$G =$  $G^c =$ 

- **Claim:** G has a k-cover iff G^c has an $(n-k)$-clique, where n is the number of vertices in G.

  (So the mapping $(G,k) \mapsto (G^c, n-k)$ is a reduction of VC to CLIQUE.)

  Proof:

INTEGER LINEAR PROGRAMMING

An integer linear program is

- A set of variables $x_1, \ldots, x_n$ which must take integer values.
- A set of linear inequalities:

  $a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \leq c_i$ $\qquad [i = 1, \ldots, m]$

e.g. $x_1 - 2x_2 + x_4 \leq 7$

$x_1 \geq 0 \qquad [-x_1 \leq 0]$

$x_4 + x_1 \leq 3$

ILP = the set of integer linear programs for which there are values for the variables that simultaneously satisfy all the inequalities.

ILP is NP-complete

ILP $\in$ NP. (Not obvious! Need a little math to prove it. Proof omitted.)

ILP is NP-hard: by reduction from 3-SAT (3-SAT $\leq_P$ ILP). Given 3-CNF Formula F, construct following ILP P as follows:

Recall: LINEAR PROGRAMMING where the variables can take *real* values is known to be in P.

More NP-complete/NP-hard Problems

- HAMILTONIAN CIRCUIT (and hence TRAVELLING SALESMAN PROBLEM) (see Sipser text for related problems)
- SCHEDULING
- CIRCUIT MINIMIZATION
- SHORT PROOF
- NASH EQUILIBRIUM WITH MAXIMUM PAYOFF
- PROTEIN FOLDING
- ⋮
- See book by Garey & Johnson for hundreds more.