NP-Completeness

The World if $P \neq NP$?

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

Idea: Try to find a “hardest” NP language.

– Want $L \in NP$ such that $L \in P$ 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**
- **Factoring $\leq L_{\text{fact}}$**

Here $L_{\text{fact}} = \{\langle N, m \rangle : N \text{ has a factor in } \{2, \ldots, m\}\}$. The last bullet follows since, to factor $N$, we can iteratively try to find one factor $x$ then recurse on both $x$ and $N/x$. To find a single factor, we can use a subroutine solving $L_{\text{fact}}$ and binary search on $m$ (recall to be efficient, our running time should be polylogarithmic in $N$, since the input length is $\lceil \log N \rceil$ bits to write down $N$). 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 $L_{\text{fact}} \notin P$ (and hence $P \neq \text{NP}$). 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 $L_{\text{fact}}$.

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 $f : \Sigma_1^* \rightarrow \Sigma_2^*$ s.t. for every $x \in \Sigma_1^*$, $x \in L_1$ 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:**

Suppose that
• \( f \) is a reduction of \( L_1 \) to \( L_2 \) computable in time \( T_1 \), a polynomial.

• \( L_2 \) is decidable in time \( T_2 \), a polynomial.

To decide whether \( x \in L_1 \):

1. Compute \( f(x) \). [takes time \( T_1(|x|) \)]
2. Decide whether \( f(x) \in L_2 \). [takes time \( T_2(|f(x)|) \)]

But we know that \( |f(x)| \leq T_1(|x|) \), since the length of the output of a TM can’t be longer than the time in which it runs.

Thus, \( T_2(|f(x)|) \leq T_2(T_1(|x|)) \).

So total time \( \leq T_1(|x|) + T_2(T_1(|x|)) \), a polynomial.

**NP-Completeness**

**Def:** \( L \) is NP-complete iff

1. \( L \in \text{NP} \) and

2. For every \( L' \in \text{NP} \), we have \( L' \leq_p L \). (“\( L \) is NP-hard”)

**Prop:** Let \( L \) be any NP-complete language. Then \( P = \text{NP} \) 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:** Since by hypothesis $L \in$ NP, it suffices to show that every $L' \in$ NP reduces to $L$.

- $L' \leq_P L_0$ since $L_0$ is NP-complete;
- $L_0 \leq_P L$ by hypothesis; and so
- $L' \leq_P L$ by transitivity.

Thus, $L$ is NP-complete.

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 $C_j$ is a disjunction (“or”) of 3 literals:

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

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

e.g. $(x \lor y \lor z) \land (\neg x \lor \neg u \lor w) \land (u \lor u \lor u)$

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.

\[ F = (\neg((x \lor \neg y) \land (z \lor w)) \lor \neg x). \]

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

\[ F = (\neg((x \lor \neg y) \land (z \lor w)) \lor \neg x), \]

\[ F_1 = \left( \begin{array}{c} (a_3 \equiv \neg y) \land (a_7 \equiv \neg x) \\ \land (a_2 \equiv x \lor a_3) \land (a_1 \equiv \neg a_4) \\ \land (a_5 \equiv z \lor w) \land (a_6 \equiv a_1 \lor a_7) \\ \land (a_4 \equiv a_2 \land a_5) \end{array} \right) \]

5. Let \( k \) be the number of the main operator/subformula of \( F \).
   (Note: \( k = 6 \) in the example)

   **Claim:** \( a_k \land F_1 \) is satisfiable iff \( F \) is satisfiable.

6. Write \( F_1 \) in 3-CNF to obtain \( F_2 \).

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

   **Proof:** Write the truth table for \( f \). To obtain a \( k \)-DNF, for each row of the table for which \( f(x) = 1 \), we obtain a clause which ANDs all the literals in that row. We then OR these together over all such \( x \). To obtain a \( k \)-CNF, we first build a \( k \)-DNF as in the last sentence for the function \( \neg f \). This is the OR of many clauses:

\[ C_1 \lor \ldots \lor C_m. \]

Each \( C_i \) is an AND of \( k \) literals. We then use De Morgan’s laws to obtain \( \neg(\neg f) \), which yields

\[ \overline{C_1 \lor \ldots \lor C_m} = \overline{C_1} \land \ldots \land \overline{C_m}, \]

which is a \( k \)-CNF.

7. Output of the reduction: \( a_k \land F_2 \).
Exercise: Note the above ingredients give us a CNF in which each clause has at most 3 literals. Some may have just 1 or 2. Show how to extend such clauses to have exactly 3 literals, from 3 distinct variables (hint: add new dummy variables and more clauses).

In contrast, 2-SAT ∈ P

Method (resolution):

1. If \(x\) and \(\neg x\) are both clauses, then not satisfiable
   
   e.g. \((x) \land (z \lor y) \land (\neg x)\)

2. If \((x \lor y) \land (\neg y \lor z)\) are both clauses, add clause \((x \lor z)\) (which is implied).

3. Repeat. If no contradiction emerges ⇒ 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.

  ![Graph Example]

  - 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 ≤_p VC:
  - Let F be a 3-CNF formula with clauses C₁,...,Cₘ, variables x₁,...,xₙ.
  - We construct a graph Gₘ and a number Nₘ such that:

  \( Gₘ \) has a size Nₘ vertex cover iff F is satisfiable

  E.g. \( F = (x₁ ∨ x₂ ∨ ¬x₃) ∧ (¬x₁ ∨ ¬x₂ ∨ x₃) ∧ (x₁ ∨ ¬x₂ ∨ x₃) \)

  - \( Gₘ \) = one dumbbell for each variable, one triangle for each clause, and corner \( j \) of triangle \( i \) is connected to the vertex representing the \( j \)th literal in \( C_i \).
  - \( Nₘ = 2m + n = 2 \) (# clauses) + (# variables).
    \( ⇒ \) 1 vertex from each dumbbell and 2 from each triangle.
  - Exercise: Show that \( F \) is satisfiable iff there is a cover of size \( Nₘ \).

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** $\text{CLIQUE} \in \text{NP}$.

$$\text{VC} \leq_p \text{CLIQUE}$$

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.**

$G = \begin{array}{c}
\text{vertices} \\
\text{edges}
\end{array}$

$G^c = \begin{array}{c}
\text{vertices} \\
\text{edges}
\end{array}$

**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.)

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

  e.g. $x_1 - 2x_2 + x_4 \leq 7$
  \[
  \begin{align*}
  x_1 & \geq 0 \\
  x_4 + x_1 & \leq 3
  \end{align*}
  \]

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. The reason is that an integer solution might have really big integers – we need to make sure they only need a polynomial number of bits. 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.

If the variables are $x_1, \ldots, x_n$, then we have the constraints $0 \leq x_1, \ldots, x_n \leq 1$. Also, if there are $m$ clauses, we have constraints $c_1, \ldots, c_m \geq 1$, one for each clause. We also have a separate constraint for each clause. If the $i$th clause is, for example, $x_i \lor \bar{x}_i \lor x_j$, then we have a constraint $c_i \leq x_i + (1 - x_i) + x_j$.

**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.