NP and Nondeterminism

More traditional way of viewing NP:

- Imagine a *nondeterministic* algorithm, where next step is not determined.
  - E.g. choose a random number $n$ and set $x = n$
- $L$ is in $NP$ if there is a nondeterministic algorithm $A$ that runs in polynomial time such that
  - if $x \in L$, some computation accepts (returns 1)
  - if $x \not\in L$, no computation accepts
- “runs in polynomial time” means exists $c$ such that all computations on input $x$ run in time $O(|x|^c)$.
  - Because of the nondeterminism, different computations on input $x$ may have different running times.
Connection to previous definition:

- if there’s a verification algorithm, can convert it to a nondeterministic polynomial algorithm:
  - nondeterministically try all possible verification strings $y$ such that $|y| = O(|x|^c)$
  - Can do this in PTIME with branching
- Conversely, if there’s a nondeterministic algorithm, can convert it to a verification algorithm:
  - $y$ describes the choices made along a given branch
NP, Co-NP, and PTIME

$L$ is in co-NP if $\overline{L}$ is in NP:

Examples:

- $L$ is the set of encodings of graphs that do not have Hamiltonian paths.

Major questions of complexity theory:

- Does $P = NP$?
  - Probably not, but no proof yet
- If $P = NP$, then there are PTIME algorithms for lots of problems that we don’t know how to do efficiently yet
  - E.g., factoring, scheduling, bin-packing, …
- Does $P = co-NP$?
  - Since $P$ is closed under complementation, this is true iff $P = NP$ (see homework)
- Does $NP = co-NP$?
- Does $P = NP \cap co-NP$?
  - We can’t answer any of these questions (yet)
  - Solving them gets you a Turing award …
The little we know:

- $P \subseteq \text{NP}/\text{co-NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$
- $P \neq \text{PSPACE}$
Reducibility

Key idea in complexity theory: reducibility

- Making precise the well-known mathematical idea of reducing one problem to another
- Idea: If you can reduce \( L_1 \) to \( L_2 \), then if you have an efficient algorithm to decide \( L_2 \), then you get an efficient algorithm to decide \( L_1 \)

Formal definition:

\( L_1 \subseteq \Sigma^* \) is polynomial-time reducible to \( L_2 \subseteq (\Sigma')^* \) if there is a polynomial time computable function \( f : \Sigma^* \rightarrow (\Sigma')^* \) such that \( x \in L_1 \) iff \( f(x) \in L_2 \).

**Lemma 1:** If \( L_2 \in P \) and \( L_1 \leq_P L_2 \), then \( L_1 \in P \).

**Proof:** Suppose \( A_2 \) is a PTIME algorithm that decides \( L_2 \), and \( f \) reduces \( L_1 \) to \( L_2 \)

- \( x \in L_1 \) iff \( f(x) \in L_2 \)

Let \( A_1(x) = A_2(f(x)) \).

- \( A_1 \) is PTIME, since \( A_2 \) and \( f \) are.
- \( x \in L_1 \) iff \( f(x) \in L_2 \) iff \( A_1(x) = A_2(f(x)) = 1 \).
Lemma 2: Reduction is transitive: If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Proof: Suppose $f$ reduces $L_1$ to $L_2$, $g$ reduces $L_2$ to $L_3$:

- $x \in L_1$ iff $f(x) \in L_2$
- $x \in L_2$ iff $g(x) \in L_3$.

Then $x \in L_1$ iff $g(f(x)) \in L_3$.

$g \circ f$ is PTIME computable.

Therefore $L_1 \leq_P L_3$ (using $g \circ f$)
NP- Completeness

A language $L$ is *NP-complete* if

1. $L$ is in NP and

2. $L$ is *NP hard* – i.e., $L$ is the “hardest” NP problem:
   - every language $L'$ in NP can be reduced to $L$
   - If $L' \in NP$, then $L' \leq_P L$

**Theorem:** If any NP-complete language is in P, then every language in NP is in P.

**Proof:** Suppose that $L$ is NP-complete, and $L$ is in P. If $L' \in NP$, then $L' \leq_P L$. Therefore $L'$ is in P.

There are *thousands* of known NP-complete languages.

- See Garey and Johnson (1979) for the classic compendium

We haven’t found any PTIME algorithm for any of them yet.
Proving a Language is NP-complete

General strategy for proving language $L$ is NP-complete:

- Show $L$ is in NP (usually easy)
- Reduce a known NP-complete problem $L'$ to $L$.
  - That is, show that $L' \leq_P L$
  - This means $L$ is NP-hard
    - This is because $\leq_P$ is transitive
    - If $L''$ is in NP, $L'' \leq_P L'$
    - Since $L' \leq_P L$, it follows that $L'' \leq_P L$.

Thus, it helps to have a core set of NP-complete problems to start with.

Getting off the ground is hard:

- How do you prove that every language in NP can be reduced to a particular language $L$?

For this we need a model of computation.
Turing Machines

A Turing machine (TM) can be thought of as an infinite tape, where a head can write 0s and 1s, together with some instructions for what to write.

- initially the tape has the input written on it.

Key question:

- How are instructions described?
  - i.e., what is the programming language?
- Idea: there is a finite set of states
- In a given state, the head can
  - read the symbol on the tape cell under it,
  - write a symbol (0/1) on the tape cell under it,
  - move one step left or one step right,
- Then the TM can change to a new state.
  - The new state depends on the old state and the symbol read.
  - There may be more than one possible next state (nondeterminism).
This may not like a very powerful model of computation, but ...

- Every program in a standard programming language (Java, C) corresponds to some TM

To show that a language $L$ is NP-hard, we have to show that for every language $L'$ in NP, there is a function $f_{L'}$ such that $x \in L'$ iff $f_{L'}(x) \in L$.

- Idea: since $L' \in \text{NP}$, there is a TM $M_{L'}$ that outputs 1 on input $x$ iff $x \in L$

- $f_{L'}(x)$ simulates the computation of $M_{L'}$ on $x$
Satisfiability: the canonical NP-complete problem

Propositional logic:

- Start with a set of primitive propositions \( \{p_1, \ldots, p_n\} \).
- Form more complicated formulas by closing off under conjunction (\( \land \)) and negation (\( \neg \))

Typical formula: \( \neg(p_1 \land \neg p_2) \land (p_2 \land \neg p_1) \).
Standard abbreviation: \( p \lor q \) is an abbreviation for \( \neg(\neg p \land \neg q) \).

Given a formula, we want to decide if it is true or false.

- The truth or falsity of a formula depends on the truth or falsity of the primitive propositions that appear in it. We use truth tables to describe how the basic connectives (\( \neg \), \( \land \)) work.
Truth Tables

For $\neg$:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\neg p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

For $\land$:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \land q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

For $\lor$:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\neg p$</th>
<th>$\neg q$</th>
<th>$\neg p \land \neg q$</th>
<th>$\neg(\neg p \land \neg q)$ = $p \lor q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

This means that $\lor$ is inclusive or, not exclusive or.
Equivalence

Two formulas are \textit{equivalent} if the same truth assignments make them true.

\textbf{Examples:}

\begin{itemize}
    \item Distribution Laws:
        \begin{itemize}
            \item $p \land (q_1 \lor q_2)$ is equivalent to $(p \land q_1) \lor (p \land q_2)$
            \item $p \lor (q_1 \land q_2)$ is equivalent to $(p \lor q_1) \land (p \lor q_2)$
        \end{itemize}
    \item DeMorgan’s Laws
        \begin{itemize}
            \item $\neg(p \land q)$ is equivalent to $\neg p \lor \neg q$
            \item $\neg(p \lor q)$ is equivalent to $\neg p \land \neg q$
        \end{itemize}
\end{itemize}

How do you check if two formulas are equivalent?

\begin{itemize}
    \item Fill in the truth tables for both.
\end{itemize}
Satisfiability

Is \((p_1 \lor p_2) \land (\neg p_2 \lor p_3) \land (\neg p_3 \lor p_1)\) satisfiable?

- Is there a truth assignment to the primitive propositions that makes this formula true?
  
  - Yes: \(p_1 \leftarrow T, p_2 \leftarrow T, p_3 \leftarrow T\)

- How about \((p_1 \lor p_2) \land (\neg p_2 \lor p_3) \land (\neg p_3 \lor \neg p_1)\)?
  
  - \(p_1 \leftarrow T, p_2 \leftarrow T, p_3 \leftarrow T\) doesn’t work.
  - \(p_1 \leftarrow T, p_2 \leftarrow F, p_3 \leftarrow F\) does.

- How about \((p_1 \lor p_2) \land (\neg p_2 \lor p_3) \land (\neg p_3 \lor \neg p_1) \land \neg p_1\)?
  
  - Nothing works ...

In general, you can tell if a formula is satisfiable by guess a truth assignment, and verifying that it works.

- The truth assignment is a certificate ...

Satisfiability is also NP-hard ....
Idea of proof:

- Start with a language $L'$ in NP and input $x$
- Since $L'$ is in NP, there exists $c$, $k$, and a (non-deterministic) TM $M_{L'}$ such that $M_{L'}$ accepts $L'$ using at most $c|x|^k$ steps on input $x$
- Construct formula $\varphi_{x,L'}$ that is satisfiable iff $x \in L'$
- Want $|\varphi_{x,L'}|$ to be $O(|x|^{2k})$
- Then $f_{L'}(x) = \varphi_{x,L'}$

Main ideas of construction

- $M_{L'}$ uses at most $c|x|^k$ cells on the tape
- Have propositions $p_{0,i,t}$, $p_{1,i,t}$, $p_{b,i,t}$, $i, t = 1, \ldots, c|x|^k$
  ○ cell $i$ has a 0/1/$b$ (b for blank) at step $t$
- Part of $\varphi_{x,L'}$ says that exactly one of $p_{0,i,t}$, $p_{1,i,t}$, $p_{b,i,t}$ holds at each time $t$
  \[
  (p_{0,i,t} \lor p_{1,i,t} \lor p_{b,i,t}) \land \\
  \lnot(p_{0,i,t} \land p_{1,i,t}) \land \lnot(p_{0,i,t} \land p_{b,i,t}) \land \lnot(p_{1,i,t} \land p_{b,i,t})
  \]
- Have propositions $p_{h,i,t}$, $i, t = 1, \ldots, c|x|^k$
  ○ The head is in position $i$ at time $t$
• Exactly one of $p_{h,1,t}, \ldots, p_{h,c|x|^k,t}$ holds (for all $t$)
• $p_{h,1,1}$ holds
  o The tape is initially at the far left
• If $x = x_1 \ldots x_k$, then $p_{x_1,1,1} \land p_{x_2,2,1} \land \ldots \land p_{x_k,k,1} \land p_{b,k+1,1} \land p_{b,c|x|^k,1}$ holds
  o $x$ is written out initially at the far left of the tape, followed by blanks.
• Similarly, can say that at time $c|x|^k$, there is a 1 at the far left, followed by blanks
  o $M_{L'}$ accepts $x$
• The hard part is to write the part of the formula that captures the step-by-step operation of $M_{L'}$.
  o Need proposition that talk about the current state of $M_{L'}$ and how it changes

Bottom line: We can simulate TMs that run in non-deterministic polynomial time using propositional logic.

• Satisfiability is NP complete!
• This was the first problem proved NP complete (by Steve Cook)
• Validity is co-NP complete
3-CNF Satisfiability

A literal is a primitive proposition or its negation:

- $p$ or $\neg p$

A clause is a disjunction of distinct literals:

- $p_1 \lor p_3 \lor \neg p_7 \lor p_2 \lor \neg p_5$

A formula is in CNF (conjunctive normal form) if it is a conjunction of clauses

$$(p_1 \lor \neg p_3) \land (p_1 \lor p_5 \lor \neg p_2 \lor p_7) \land (p_3 \lor \neg p_5)$$

A formula is in $k$-CNF if each clause has exactly $k$ literals.

**Theorem:** The satisfiability problem for 2-CNF formulas is in P.

**Theorem:** The satisfiability problem for 3-CNF formulas in NP-complete.

**Proof:** It’s clearly in NP. To show that it’s NP-hard, it suffices to show that the satisfiability of an arbitrary formula $\varphi$ can be reduced in polynomial to the satisfiability of a 3-CNF formula $\varphi'$. 
Three steps:

**Step 1:**

- Write a binary parse tree for \( \varphi \),
  - internal nodes are labeled with \( \neg, \land, \) and \( \lor \)
  - leaves are labeled with literals
  - An internal node represents a subformula of \( \varphi \)
  - Introduce a new primitive proposition \( q \) for each internal node
  - Write formula that says that \( q \) characterizes the formula at that node.
    * If internal node is \( \neg \) and successor is labeled by \( q' \),
      \[
      (q \land \neg q') \lor (\neg q \land q')
      \]
    * If internal node is \( \land \) and successors are \( q_1 \) and \( q_2 \):
      \[
      (q \land q_1 \land q_2) \lor (\neg q \land \neg(q_1 \land q_2))
      \]
- Let \( \varphi' \) be the conjunction of these formulas.
  - Not hard to show that \( \varphi' \) is satisfiable iff \( \varphi \) is satisfiable
Step 2: Convert \( \varphi' \) to an equivalent CNF formula, using various equivalences, where each clause has at most 3 literals:

- Using Distribution Laws, \((q \land \neg q') \lor (\neg q \land q')\) is equivalent to
  \[
  (q \lor \neg q) \land (q \lor q') \land (\neg q' \lor \neg q) \land (\neg q' \lor q')
  \]

- Using Distribution Laws and DeMorgan’s Laws, can do the same for other clauses.

- (Actually, every formula is equivalent to a CNF formula)

Step 3: Get an equi-satisfiable 3-CNF formula

- Replace a disjunct \( p_1 \lor p_2 \) by
  \[
  (p_1 \lor p_2 \lor q) \land (p_1 \lor p_2 \lor \neg q)
  \]

- The new formula is satisfiable iff the original was.