The Class of Languages $\mathcal{P}$

- If a (deterministic) TM M has some polynomial $p(n)$ such that M never makes more than $p(n)$ moves when presented with input of length n, then M is said to be a *polynomial-time TM.*
- $\mathcal{P}$ is the set of languages that are accepted by polynomial-time TM's.
- Equivalently, $\mathcal{P}$ is the set of problems that can be solved by a real computer by a polynomial-time algorithm.
    - Why? Because while $T(n)$ steps on a computer may become $T^3(n)$ steps on a TM, $T(n)$ cannot be a polynomial unless $T^3(n)$ is.
    - Many familiar problems are in $\mathcal{P}$: graph reachability (transitive closure), matrix multiplication (is this matrix the product of these other two matrices?), etc.

The Class of Languages $\mathcal{NP}$

- A nondeterministic TM that never makes more than $p(n)$ moves in any sequence of choices for some polynomial p) is said to be a *polynomial-time NTM.*
- $\mathcal{NP}$ is the set of languages that are accepted by polynomial-time NTM's.
- Many problems are in $\mathcal{NP}$ but appear not to be in $\mathcal{P}$: TSP (is there a tour of all the nodes in a graph with total edge weight $\leq k$?), SAT (does this Boolean expression have a satisfying assignment of its variables?), CLIQUE (does this graph have a set of k nodes with edges between every pair?).
- One of the great mathematical questions of our age: Is there anything in $\mathcal{NP}$ that is not in $\mathcal{P}$?

NP-Complete Problems

If we can't resolve the "$\mathcal{P} = \mathcal{NP}$ question, we can at least demonstrate that certain problems in $\mathcal{NP}$ are "hardest," in the sense that if any one of them were in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

- Called *NP-complete* problems.
- Intellectual leverage: each NP-complete problem's apparent difficulty reenforces the belief that they are all hard.

Method for Proving NP-complete Problems

- Polynomial-time reductions (PTR): take time that is some polynomial in the input size to convert instances of one problem to instances of another.
  - Of course, the same algorithm converts non-instances of one to non-instances of the other.
- If P_1 PTR to P_2, and P_2 is in $\mathcal{P}$, then so is P_1.
  - Why? Combine the PTR and P_2 test to get a polynomial-time algorithm for P_1.
- Start by showing *every* problem in $\mathcal{NP}$ has a PTR to SAT (= satisfiability of a Boolean formula).
  - Thus, if SAT is in $\mathcal{P}$, everything in $\mathcal{NP}$ is in $\mathcal{P}$; i.e., $\mathcal{P} = \mathcal{NP}$!
- Then, more problems can be proven NP-complete by showing that SAT PTRs to them, directly, or indirectly.
  - Key point: the composition of any finite number of PTR's is a PTR.
- Don't forget that you also need to show the problem is in $\mathcal{NP}$ (usually easy, but necessary).

Reduction of Any L in $\mathcal{NP}$ to SAT

Assume $L = L(M)$ for some NTM M that is time-bounded by polynomial $p(n)$.

- Key idea: if w, of length n, is in L, then there is a sequence of $p(n) + 1$ ID's, each of length $p(n) + 1$, that demonstrates acceptance of w.
  - Well not exactly: the accepting sequence might be shorter. If so, extend $\vdash$ to allow $\alpha \vdash \alpha$ if α is an ID with an accepting state.
  - Still not exactly: some ID's will be shorter than $p(n) + 1$ symbols. Pad those out with blanks.
- Now, we can imagine a square array of symbols X_{ij}, for i and j ranging from 0 to $p(n)$, where X_{ij} is the symbol in position j of the ith ID.

- Given string w, construct a Boolean expression that says "these X_{ij}'s represent an accepting computation of w.

  - **Very Important**: The construction must be carried out in time polynomial in $n = |w|$. In fact, we need only $O(1)$ work per X_{ij} [or $O\big(p^2(n)\big)$ total]

  - Another important principle: The output cannot be longer than the amount of time taken to generate it, so we are saying that the Boolean expression will have $O(1)$ "stuff" per X_{ij}.

- The propositional variables in the desired expression are named $y_{i,j,Y}$, which we should interpret as an assertion that the symbol X_{ij} is Y.

- The desired expression $E(w)$ is $S \wedge M \wedge F =$ starts, moves, and finishes right.

Starts Right

- S is the AND of each of the proper variables: $y_{0,0,q_0} \wedge y_{0,1,a_1} \wedge \cdots \wedge y_{0,n,a_n} \wedge y_{0,n+1,B} \wedge \cdots \wedge y_{0,p(n),B}$

  - Here, q_0 is the start state, $w = a_1, \ldots, a_n$, and B is the blank.

Moves Right

Key idea: the value of $X_{i,j}$ depends only on the three symbols above it, to the northeast, and the northwest.

- However, since the components of a move (next state, new symbol, and head direction) must come from the same NTM choice, we need rules that say: when $X_{i-1,j}$ is the state, then all three of $X_{i,j-1}$, X_{ij}, and $X_{i,j+1}$ are determined from $X_{i-1,j-1}$, $X_{i-1,j}$, and $X_{i-1,j+1}$ by one choice of move.

- We also have rules that say when the head is not near X_{ij}, then $X_{ij} = X_{i-1,j}$.

- Details in the reader. The essential point is that we can write an expression for each X_{ij} in $O(1)$ time.

  - Therefore, this expression is $O(1)$ long, independent of n.

Finishes Right

Key idea: we defined the TM to repeat its ID once it accepts, so we can be sure the last ID has an accepting state if the TM accepts.

- F is therefore the OR of all variables $y_{p(n),j,q}$ where q is an accepting state.

Conjunctive Normal Form

We now know SAT is NP-complete. However, when reducing to other problems, it is convenient to use a restricted version of SAT, called 3SAT, where the Boolean expression is the AND of *clauses*, and each clause consists of exactly 3 *literals*.

- A literal is a variable or a negated variable.
  - E.g., x or $\neg y$. We shall sometimes use the common convention where $\bar{x}$ represents the negation of x.
- A *clause* is the OR of literals.
  - E.g., $(x \vee \bar{y} \vee z)$.
  - We shall often follow common convention and use $+$ for $\vee$ in clauses, e.g., $(x + \bar{y} + z)$, and also use juxtaposition (like a multiplication) for $\wedge$.
- An expression that is the AND of clauses is in *conjunctive normal form* (CNF).

CSAT

Satisfiability for CNF expressions is NP complete.

- The proof (reduction from SAT) is simple, because the expression $S \wedge M \wedge F$ we derived is already the product (AND) of expressions whose size is $O(1)$, i.e., not dependent on $n = |w|$.
- First, we can push all the $\neg$'s down the expression until they apply only to variables; i.e., any expression is converted to an AND-OR expression of literals.
  - Use *DeMorgan's laws*: $\neg(E \wedge F) = (\neg E) \vee (\neg F)$ and $\neg(E \vee F) = (\neg E) \wedge (\neg F)$.
  - Changes the size of the expression by only a constant factor (because extra $\neg$'s and possibly parentheses are introduced).

- Next, distribute the OR's over the AND's, to get a CNF expression.
  - This process can **exponentiate** the size of an expression.
  - However, since this process only needs to be applied to expressions of size $O(1)$, the result may be huge expressions, but expressions whose lengths are independent of n and therefore still $O(1)$!

3-CNF and 3SAT

- A Boolean expression is in 3-CNF if it is the product (AND) of clauses, and each clause consists of exactly 3 literals.
  - Example: $(x + \bar{y} + z)(\bar{x} + w + \bar{z})$.
- The problem *3SAT* is satisfiability for 3-CNF expressions.

Reducing CSAT to 3SAT

It would be nice if there were a way to turn any CNF expression into an equivalent 3-CNF expression, but there isn't.

- Trick: we don't have to turn a CNF expression E into an equivalent 3-CNF expression F, we just need to know that F is satisfiable if and only if E is satisfiable.
- We turn E into 3-CNF F in polynomial time by introducing new variables.
  - If the clause is too long, introduce extra variables. Example: $(u+v+w+x+y+z)$ becomes $(u + v + a)(\bar{a} + w + b)(\bar{b} + x + c)(\bar{c} + y + z)$.
  - A clause of only two, like $(x + y)$ can become $(x + y + a)(x + y + \bar{a})$.
  - A clause of one, like (x) can become $(x + a + b)(x + a + \bar{b})(x + \bar{a} + b)(x + \bar{a} + \bar{b})$.
  - See the reader for explanations of why these transformations preserve satisfiability, and can be carried out in polynomial time.
- Thus 3SAT is NP-complete. This problem plays a role similar to PCP for proving NP-completeness of problems.