Recitation 08: NP-completeness

The edge of P

Intuitively, P contains the easily-solvable problems; NP, the easily-checkable problems. It seems that NP \neq \text{coNP} and P \neq NP \cap \text{coNP}. That is, most experts favor the top left world among these four worlds →

But no one has yet ruled out any of the four possibilities. Remarkably, NP contains 'maximally hard' problems (hatched areas in Venn diagrams) to which all other NP problems efficiently reduce. SAT is such an NP-complete problem, so SAT \not\in P if and only if P \neq NP.

At first glance, it might seem ‘obvious’ that P differs from NP. After all, NP includes problems that seem to require an exponentially large search. But how do we know that our problem requires a brute force search? How do we know no clever pruning is ever enough?

Take the DOMINO problem: given a 3 \times n grid with some cells crossed out, can we tile the remaining cells with dominos? A grid is tilable when both halves are. The converse also holds if we allow any of 2^3 = 8 overhang patterns. Recursively searching through overhang patterns solves DOMINO in \sim 8^n time. Here, we’ve redundantly computed \(\square\)'s tilability in both \(\square\)'s and \(\blacksquare\)'s branches. Let’s instead solve each sub-problem once, storing the answers for future look-ups. We get a p-time solution to DOMINO! P is larger than we’d guess.

This idea is called Dynamic Programming. We identify redundant sub-computations (often arising due to overlapping sub-problems), then eliminate the redundancy using a lookup table. We thus trade space for time.

Now, say we’ve been struggling to design a fast algorithm for some problem. Should we try to be more clever, as with DOMINO? Or might our problem have no p-time solution at all? In short: how do we know when to give up? Assuming P \neq NP, we have a sufficient criterion for giving up: if \(\mathcal{L}\) efficiently reduces from every NP problem, then \(\mathcal{L} \not\in P\).

To check this criterion it’s enough to efficiently reduce SAT to \(\mathcal{L}\).

More precisely, we say a language A p-time reduces to a language B when \(w \in A \iff f(w) \in B\) for some function \(f\) computable in polynomial time; then we write \(A \leq_p B\). p-time reduction is analogous to mapping reduction (imperfectly, since Rec \cap \text{coRec} = \text{Dec}):

\[
\begin{align*}
P & : \quad \text{NP} & : \quad \leq_p & : \quad \text{SAT} & : \quad \leq_m & : \quad A_{TM} \\
\text{Decidable} & : \quad \text{Recognizable} & : \quad \leq_m & : \quad A_{TM}
\end{align*}
\]

Armed with this technical tool, let’s show some problems are hard!

Let’s practice some true-or-false questions (the answers are TFFFTTF):

- SAT \leq_p A_{TM}?
- \(!\text{ETM} \leq_p A_{TM}\)?
- \(A \leq_B B \implies B \leq_p A\)?
- \((A \in P \land A \leq_p B) \implies B \in P\)?
- \((A \not\in P \land A \leq_p B) \implies B \not\in P\)?
- \((A \not\in \text{NP} \land A \leq_p B) \implies B \not\in \text{NP}\)?
- \(A \leq_p B \iff A \leq_m B\)?
3-SAT is NP-complete.

We proved in class that SAT is NP-complete. We’ll now prove that 3-SAT is NP-complete. Since we know that 3-SAT is in NP, we just need to show that SAT \( \leq_p \) 3-SAT.

Now, SAT and 3-SAT are so similar that the problem may seem trivial. It is tempting to try to distribute out a SAT formula into an equivalent 3-SAT formula, that is, one with \( \land \)s on the outside and \( \lor \)s on the inside. For example, by the distributive property, \((a \land b) \lor \neg c\) translates to \((a \lor \neg c) \land (b \lor \neg c)\). Though the two formulae are logically equivalent, there’s a problem with this translation procedure: on longer formulae, it could require exponential time! Indeed, a formula \((a \land b) \lor (c \land d) \land \cdots \land (y \land z)\) with \(n\) disjoined clauses, when distributed out, has \(2^n\) conjoined clauses!

So let’s try a different approach. We will still p-time-reduce SAT \( \leq_p \) 3-SAT. However, we’ll translate a SAT formula not to a logically equivalent 3-SAT formula but instead to an equivalently satisfiable 3-SAT formula. Just as with Cook-Levin, in which we simulated the computation history of an NTM via a SAT formula with many more variables than the NTM’s tape, we will now simulate an arbitrary SAT formula via a 3-SAT formula with many more variables than the SAT formula.

We’ll do this by using 3-SAT to simulate a digital logic circuit of \( \land \)s, \( \lor \)s, and \( \neg \)s that computes the boolean value of a SAT formula. The theme of computation histories thus strikes again!

Following Cook-Levin, we’ll introduce a 3-SAT variable for each intermediate computed value in the SAT formula. E.g. to the right we use 7 variables to simulate the old 3-variable formula. We assert the output \(z\) via a 3-SAT clause \(\lor z \lor z\). One question remains: how shall we enforce that each intermediate value is correctly computed?

To understand how to implement local computations such as \(x = s \land v\), we first examine the sorts of computations we can do using 3-SAT clauses. A 3-SAT clause \(A \lor B \lor S\) forbids the three variables from being simultaneously false and, by itself, allows all other possibilities. We may thus use multiple clauses to constrain the possible joint configurations of \(A, B, S\) to precisely the possibilities we desire.

By this technique we may introduce four 3-SAT clauses to ensure that \(x = s \land v\). Likewise for \(\lor s\). And likewise but with fewer clauses for \(\neg s\). In addition to these clauses, we assert \(z \lor z \lor z\). The resultant 3-SAT formula is satisfiable if and only if there exists a computation history starting with some \(a, b, c\) such that every intermediate computation is correct and such that the output is true. That is: our 3-SAT formula is satisfiable if and only if the SAT formula is satisfiable. QED.

Note that 3-SAT’s isn’t special — the same proof works for 4-SAT, 5-SAT, etc. But the proof fails for 2-SAT. Do you see why?
SUBSET-SUM is NP-complete.

We’ve got three dollar bills, four quarters, three dimes, and a nickel — can we make exact change for a $2.95 coffee? SUM generalizes this problem. A problem instance is a list $\ell$ of naturals together with a target natural $t$. SUM contains all $(\ell,t)$ pairs for which some (potentially non-contiguous) sub-list of $\ell$ sums to $t$. SUM is in NP — see why?

To show SUM is hard, we’ll efficiently translate a 3-CNF formula to a pair $(\ell,t)$ that is soluble if and only if the formula is satisfiable:

$$\exists s \subseteq \ell : \text{sum}(s) = t \iff \exists x \in \{0,1\}^n : c^0(x) \land c^1(x) \land \cdots$$

We’ll massage the LHS to look increasingly like the RHS. To start, let’s represent $s$ by its indicator function $\sigma$; summation, by dot product:

$$\exists \sigma \in \{0,1\}^{\vert \ell \vert} : \sigma \cdot \ell = t$$

A difference remains: 3-SAT enforces multiple constraints $\hat{c}^k(x) = true$ but SUM enforces only one: $\sigma \cdot \ell = t$. Can we stuff multiple sum constraints into just one? Yes, we can! The idea is to concatenate:

$$\begin{align*}
\sigma_0 4 + \sigma_1 4 &= 4 \\
\sigma_0 23 + \sigma_1 98 &= 121
\end{align*}$$

 iff $$\sigma_0 40023 + \sigma_1 40098 = 40121$$

Note how we buffer with zeros so that, despite carrying, distinct constraints don’t interfere; overall, the lengths expand only polynomially. This trick works in general, so we see that SUMS $\leq_p$ SUM, where SUMS contains soluble systems of linear equations between bits:

$$\exists \sigma \in \{0,1\}^d : (\sigma \cdot \ell^0 = t^0) \land (\sigma \cdot \ell^1 = t^1) \land \cdots$$

We reduce 3-SAT to SUMS simply by introducing a linear constraint for each clause. Say that $\sigma_e, \sigma_f, \sigma_g$ are SUMS bits representing an assignment to 3-SAT variables $x_e, x_f, x_g$; then the constraint

$$\sigma_e + \sigma_f + \sigma_g + \hat{\sigma}^k + \hat{\sigma}^k = 3$$

enforces (that at least one of $\sigma_e, \sigma_f, \sigma_g$ equals 1 and thus) the clause $x_e \lor x_f \lor x_g$. Here, $\hat{\sigma}^k, \hat{\sigma}^k$ are “slack bits” — see how they help?

Only one snag remains. What do we do with clauses containing negated variables? Well, let’s just partner each old bit $\sigma'_j$ with a new bit $\sigma_j$ such that $\sigma_j + \sigma'_j = 1$. Then $\sigma'_j = 1$ exactly when $x_j$ is false.

In sum, we have two bits $\sigma_j, \sigma'_j$ for each variable and two bits $\hat{\sigma}^k, \hat{\sigma}^k$ for each clause. We impose linear relations between these bits — one for each clause and one for each variable. The resulting SUMS instance is soluble if and only if the original 3-SAT instance is satisfiable. So SAT $\leq_p$ 3-SAT $\leq_p$ SUMS $\leq_p$ SUM and SUM is NP-hard. QED.

Since practical change-making involves at most twelve kinds of bills and coins, each list thereof has only polynomially many partial sums. So, despite SUM’s hardness, buying coffee is tractable. Whew!