We illustrate polynomial time mapping reduction (also called “p-time reduction” to save ink) by discussing two examples. We approach these reductions as programming puzzles and connect to general concepts such as computation histories. After reviewing these notes, try these exercises:

Exercise: show that $4\text{-SAT} \leq_p 3\text{-SAT}$ without appealing to SAT or NP-completeness.

Exercise: show that $\text{HAMPATH} \leq_p \text{VC}$ without appealing to $3\text{-SAT}$ or NP-completeness.

What is $\leq_p$?

We say a language $A$ **p-time reduces** to a language $B$ when $w \in A \iff f(w) \in B$ for some function $f$ that is computable in polynomial time. In this case, we write $A \leq_p B$. Intuitively, this implies that $A$ can be solved just as quickly as $B$. Using yet looser language, we say that "$A$ is not more complex than $B$". For example, $\text{EPS}_{\text{TM}} \leq_p \text{A}_{\text{TM}}$ — do you see why? P-time reduction to an easy problem can help us design good algorithms, while p-time reduction from a provably hard problem can help us avoid wasting time looking for good algorithms. Let’s check our intuition:\n
\[
\begin{array}{l@{}l@{}l@{}l}
\text{T/F} & \{a^kb^k : 0 \leq k\} & \leq_p \{c\} & \text{T/F} & \{A \neq P \land A \leq_p B\} & \Rightarrow & B \notin P? \\
\text{T/F} & A \leq_p A? & \text{T/F} & (B \notin P \land A \leq_p B) & \Rightarrow & A \notin P? \\
\text{T/F} & A \leq_p B \Rightarrow B \leq_p A? & \text{T/F} & (A \notin NP \land A \leq_p B) & \Rightarrow & B \notin NP? \\
\text{T/F} & (A \leq_p B \land B \leq_p C) \Rightarrow A \leq_p C? & \text{T/F} & (B \in NP \land A \leq_p B) & \Rightarrow & A \in NP? \\
\text{T/F} & (A \in P \land A \leq_p B) \Rightarrow B \in P? & \text{T/F} & A \leq_p B \iff A \leq_m B? \\
\text{T/F} & (B \in P \land A \leq_p B) \Rightarrow A \in P? & \text{T/F} & A \leq_p B \Rightarrow A \leq_m B? \\
\end{array}
\]

The situation is thus in many ways analogous to that of mapping reduction\(^1\):

\[
\begin{align*}
\text{P} & : \text{NP} & : \leq_p : \text{SAT} & :: \\
\text{Decidable} & : \text{Recognizable} & : \leq_m : \text{A}_{\text{TM}} \\
\text{3-SAT is NP-complete.} & \\
\end{align*}
\]

In class, we proved that $\text{SAT}$ is NP-complete\(^\dagger\). We’ll now prove that $3\text{-SAT}$ is NP-complete. Since we know that $3\text{-SAT}$ is in NP, we just need to show that $3\text{-SAT}$ is NP-hard. From this immediately follows the NP-completeness of $\text{CLIQUE}$ and $\text{HAMPATH}$ — do you see why?

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

So let’s try a different approach. We will still p-time-reduce $\text{SAT} \leq_p 3\text{-SAT}$. 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 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 introduce a 3-SAT variable for each intermediate computed value in the SAT formula. For example, above we use 13 new variables to simulate the old, 5-variable formula.

\(^1\)Reading down the left then the right column, we see that the answers TTFTFT TFTTFT are correct.

\(^\dagger\)Warning: though $\text{Rec} \cap \text{coRec} = \text{Decid}$, it is unknown whether $\text{NP} \cap \text{coNP} = P$!

\(^\dagger\)This is Halloween’s Cook-Levin proof. To jog your memory, this is the day Mike revealed his tapes.
We enforce the truth of the output $z$ via a 3-SAT clause $z \lor z \lor z$. One question remains: how shall we ensure that each intermediate value is correctly computed?

To implement local computations such as $x = s \land v$, we first learn how to "program in 3-SAT". A 3-SAT clause $A \lor B \lor C$ 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 $A, B, C$ to precisely the possibilities we desire:

Using this technique, we may introduce four 3-SAT clauses to ensure that $x = s \land v$, four clauses to ensure that $w = d \lor e$, and so forth. It turns out that fewer clauses suffice for computations of negation. In addition to these clauses, we have the 3-SAT clause $z \lor z \lor z$. The resultant 3-SAT formula is satisfiable if and only if there exists a computation history starting with some $a, b, c, d, e$ such that every intermediate computation is correct and such that the output is true. In other words, the resultant 3-SAT formula is satisfiable if and only if the original SAT formula is satisfiable. QED.

**VERTEX-COVER** is NP-complete.

In solving a $k$-SAT instance, we can’t blindly set all variables to true because some variables may appear in negated form. The VERTEX-COVER language is like 2-SAT, except that no variable may appear in negated form; what stops us from setting all variables to true is that each VC instance asks whether the formula is solvable by setting at most $k$ variables to true. Thus, each instance of VC contains both a formula and a number. Here’s an example:

$$(a \lor b) \land (b \lor c) \land (c \lor a) \land (c \lor d) \land (d \lor e), \quad k = 3$$

Such formulae are conveniently depicted as graphs with one node per variable and one edge per clause; VC then asks whether some $k$ nodes touch all edges. The above example instance looks like:

---

Note: This is the strategic problem of placing a few streetlights at intersections to illuminate our city’s streets!
(a) Can we choose 3 nodes that together touch all edges?

(b) Yes! Here is one way. All 5 edges are touched.

(c) However, no 2 nodes together touch all edges.

Amazingly, VC is NP-complete! VC is clearly in NP (see why?), so we’ll just show that some language already established as NP-complete — say, 3-SAT — p-time reduces to VC. We choose 3-SAT instead of SAT or HAMPATH because 3-SAT looks the most like VC.

Our p-time reduction must translate a 3-SAT instance into a VC instance. As we program in VC, we must hence reconcile two language differences: (memory) that VC forbids negated variables and (computation) that VC’s clauses are shorter than 3-SAT’s.

Memory. We’d like to refer to a bit’s truth or falsehood, but VC only permits the former. Thus, we introduce two VC nodes $a_0$ and $a_1$ to represent a bit $a$. We connect them with an edge to ensure at least one is true, and we shrink $k$ until only one can be true. Likewise, we may store a digit using a size-10 clique. So cliques are gadgets that can implement memory! We then represent multiple variables with capacities $n_0, n_1, n_2, \ldots$ by using $\sum n_i$ nodes and setting $k = \sum (n_i - 1)$.

Computation. So clique-gadgets help implement state. How do we compute using this state? As sub-figure (b) shows, an edge between two cliques forbids a single pair of state values. We may further constrain the allowed pairs of values by introducing more edges between the cliques. Thus we may simultaneously enforce any pairwise relationships we wish among our several clique-gadgets.

So let’s introduce one 2-valued clique-gadget per 3-SAT variable and one 7-valued clique-gadget per 3-SAT clause. Each variable-gadget represents true or false; each clause-gadget represents any of the $2^3 - 1 = 7$ triplets of bits satisfying the clause. For example, the clause-gadget for $X = a \lor b \lor c$ would have 7 nodes: $abc=001$, $abc=010$, $abc=011$, $abc=100$, $abc=101$, $abc=110$, $abc=111$. We connect each of the variable-gadgets for $a, b, c$ to the clause-gadget for $X$ so that each component of $X$’s value matches the corresponding variable’s value. The VC instance is in VC if and only if:

- each variable-gadget assigns one of T or F to the corresponding 3-SAT variable and
- each clause-gadget assigns (in one of 7 satisfying ways) T or F to each of its 3-SAT var.s and
- the clause-gadgets’ assignments agree with variable-gadgets’ assignments.

But the above are true if and only if the 3-SAT instance was satisfiable. QED.

---

Our gadgets will be bigger but more interpretable than those in the book’s Theorem 7.34.

This common technique is called one hot encoding.