We illustrate polynomial time mapping reduction (also called “p-time reduction” to save ink) by discussing two examples. We will approach these reductions as programming puzzles, thus connecting to general concepts such as computation histories and representation of state.

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, \( E_{TM} \leq_p A_{TM} \) — 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*

\[
\begin{align*}
\text{T/F} & \quad \{a^k b^k : 0 \leq k \} \leq_p \{\varepsilon\} \quad \text{T/F} & \quad (A < P \land A \leq_p B) \implies B \not\in P \\
\text{T/F} & \quad A \leq_p A? \quad \text{T/F} & \quad (B \not\in P \land A \leq_p B) \implies A \not\in P \\
\text{T/F} & \quad A \leq_p B \implies B \leq_p A? \quad \text{T/F} & \quad (A \not\in NP \land A \leq_p B) \implies B \not\in NP \\
\text{T/F} & \quad (A \leq_p B \land B \leq_p C) \implies A \leq_p C? \quad \text{T/F} & \quad (B \in NP \land A \leq_p B) \implies A \in NP \\
\text{T/F} & \quad (A \in P \land A \leq_p B) \implies B \in P? \quad \text{T/F} & \quad A \leq_p B \iff A \leq_m B \\
\text{T/F} & \quad (B \in P \land A \leq_p B) \implies A \in P? \quad \text{T/F} & \quad A \leq_p B \implies A \leq_m B
\end{align*}
\]

The situation is thus in many ways analogous to that of mapping reduction†:

\[
\text{P} : \quad \text{NP} : \quad \leq_p : \quad \text{SAT} :: \\
\text{Decidable} : \quad \text{Recognizable} : \quad \leq_m : \quad A_{TM}
\]

3-SAT is NP-complete. In class, we proved 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 3-SAT is NP-hard. From this immediately follows the NP-completeness of CLIQUE and HAMPATH — do you see why?

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 \( \forall s \) on the outside and \( \exists s \) 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 \( (a \land b) \lor (c \land d) \lor \cdots \lor (y \land z) \) 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 with many more variables than the SAT formula. We’ll do this by using 3-SAT to simulate a digital logic circuit of \( \forall s \), \( \exists s \), and \( \neg s \) that computes the boolean value of a SAT formula. The theme of computation histories thus strikes again!

Following Cook-Levin, we will introduce a 3-SAT variable for each intermediate computed value in the SAT formula. For example, in the above, we use 13 new variables to simulate the old, 5-variable formula. We may enforce the truth of the output \( z \) via a 3-SAT clause \( z \lor z \lor z \). One question remains: how shall we enforce that each intermediate value is correctly computed?

---

* Reading down the left then the right column, we see that the answers TTFTFT TFFTFT are correct.
† Warning: though Rec ∩ coRec = Decid, it is unknown whether NP ∩ coNP = P!
‡ This is Halloween’s Cook-Levin proof. To jog your memory, this is the day Mike revealed his tapes.
(a) A computation history for the SAT formula 
\((a \land b \land \neg(b \lor c)) \lor \neg(d \lor e)\).

(b) We need to verify correct transitions and a correct output using only 3-SAT clauses.

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. In doing so, we will learn how to “\textbf{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:

(a) A cube of possibilities for 3 boolean variables. Without any constraints, all 8 corners are allowed. We seek to allow only the corners in which \(A \land B = C\).

(b) We progress by introducing the clause \(A \lor B \lor \neg C\). The clause rules out the corner \(A = 0, B = 0, C = 1\). This is good, because in that corner, \(A \land B \neq C\)!

(c) We achieve our goal by introducing further clauses, each ruling out one configuration. Only 4 corners remain: precisely those for which \(A \land B = C\) is true.

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.

\textbf{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.\footnote{This is the strategic problem of placing a few streetlights at intersections to illuminate our city’s streets!} The above example instance looks like:
(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 $\parallel$. So cliques are gadgets that can implement memory! We then represent multiple variables with capacities $n_0, n_1, n_2, \cdots$ by using $\sum n_i$ nodes and setting $k = \sum (n_i - 1)$.

(a) We implement a 4-valued part $X$ of memory via a size-4 clique allotted 3 true nodes. The unique false node determines the gadget’s value. Here, $X = 2$.

(b) We implement constraints involving two parts of memory by connecting the two cliques. For example, an edge between the nodes $X=0$ and $Y=3$ forbids the configuration $X=0,Y=3$. Thus, of $4 \times 5 = 20$ pairs of values, only 19 are now allowed. We may impose multiple constraints to more fully relate $X$ and $Y$.

Computation. Now that we know how to implement state using clique-gadgets, how do we compute using these gadgets? As sub-figure (b) above shows, a single edge between two cliques forbids a single pair of values between two clique-gadgets. We may constrain the allowed pairs of values further by introducing more edges between the two cliques. This allows us to enforce simultaneously 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

- 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.

---

* We will give a construction similar to Theorem 7.34 in the book. Because we try here to motivate each design choice, our gadgets will be bigger but more interpretable than the book’s.

∥ This common technique is called one hot encoding.