NP-Completeness

1 Definitions

Three rationales for calling polynomial-time algorithms *tractable*:

- Few polytime algorithms have degrees much more than 5 or 6.
- Stability under models of computation: serial random access machines (SRAM), Turing machines, poly-size parallel machines.
- Closure under addition, multiplication, constant-time composition.

1.1 Problems and Decisions

An *abstract problem* Q is a binary relation between a set I of problem *instances* and a set S of *solutions*. Informally, the abstract problem is a set of pairs of the form (*instance, solution*). We can further simplify this to the set of *decision problems* having *yes/no* answers by mapping the set of instances to the set $\{0,1\}$. The instances may need to be augmented with a parameter k; then we can recast optimization problems by doing a binary search over the parameter k. For computation, the problem instance is encoded into binary, and the length of the encoding is considered to be the *size* of the instancen.

One can argue that a problem (language) is *accepted* in polynomial time iff it is *decided* in polynomial time. If a yes/no answer can be found in polytime, we can certainly output the "yes" part in polytime. In the other direction, the proof is not constructive — it says that if a Turing machine exists that accepts in polynomial time, then another Turing machine, one with a clock, can try to accept but will say "no" and halt after $c \times n^k$ operations for the appropriate c and k.

1.2 The Class P

Now we can define the class P to be the set of concrete (encoded) decision problems solvable in polynomial time. Instances (strings) which correspond to no syntactic or meaningful problem can be mapped arbitrarily to 0. Also, the encoding can make a difference: it suffices to prohibit unary encodings, because alphabets of order higher than unary differ in efficiency only by constants. Two encodings are polynomially related if each can be computed from the other in polynomial time. (This implies that the second encoding can't be longer than polynomial in the size of the first.) As in Turing theory, we denote by $< X >$ the encoding of an object X, and by $| < X > |$ its length.

2 Verification

2.1 The Class NP

Verification of a solution can be much quicker than finding the solution, e.g. Hamilton Cycle. The *complexity class* **NP** is the class of languages that can be verified by a polynomial time algorithm. "NP" stands for "nondeterministic polynomial" time and refers to the time it would take a nondeterminisitic Turing machine to solve the problem by guessing a solution and confirming it in polynomial time. Clearly $\mathrm{P} \subseteq \mathrm{NP}$; it is not known whether $\mathrm{P} = \mathrm{NP}$, but it is believed not. Nor is it known whether NP is closed under complementation, i.e. whether $L \in \mathrm{NP} \Rightarrow \bar{L} \in \mathrm{NP}$. If we define the complexity class **co-NP** as the set of languages L such that $\bar{L} \in \mathrm{NP}$ then the closure question can be rephrased as whether $\mathrm{NP} = \text{co-NP}$. (Since P is closed under complementation, it follows that $\mathrm{P} \subseteq (\mathrm{NP} \cap \mathrm{co}-\mathrm{NP})$, but equality is not known.) The current belief is expressed in figure 36.2, p. 928 of CLR. It is important to understand that showing (via a certificate) that a certain property holds can be more or less difficult that showing that it doesn't hold. The difference is understandable as between showing existence and nonexistence. The latter involves a universal quantifier "for all instances, a certificate does not exist."

2.2 Certificates

Verification takes the form of a *certificate*, which is a binary string encoding the alleged solution. A verification algorithm takes an instance x and a certificate y and outputs 1 iff the certificate is a solution of the instance of the problem.

3 Reducibility and NP-completeness

The NP-complete problems form a class of "equivalent difficulty", where equivalence means the existence of polynomial-time transformation or reduction. The NP-complete problems have the property that *any* problem in NP can be polynomial-time reduced to one (and therefore to all) of the NP-complete problems. Problem Q can be polynomial-time reduced to problem Q' ($Q \leq_p Q'$) iff there exists a polynomial-time function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that $\forall x \in \{0,1\}^*,\ x \in Q$ iff $f(x) \in Q'$. Polynomial reduction preserves polynomial-time solution. It means that one problem is no harder than another, to within a polynomial-time difference.

3.1 NP-Complete Problems

Now we can state the definition: A language $L \subseteq \{0,1\}^*$ is **NP-complete** if $L \in \mathrm{NP}$ and $\forall L' \in \mathrm{NP}\ L' \leq_p L$. A language that satisfies just the second condition is called **NP-hard**.

4 Cook's Theorem

Version 1: Circuit Satisfiability is NP-complete:
An instance of CIRCUIT-SAT is a Boolean circuit (**and**s, **or**s, **not**s,) with some number of inputs and one output. The decision problem is whether there exists a combination of 0/1 inputs that makes the output 1. To prove this is in NP, trace the intermediate 0/1 values through the circuit. If each gate has bounded input (say 2) this takes time proportional to the number of gates in the circuit. To prove it is as hard as any problem in NP, observe that the entire state of a computer, called a *configuration* (registers, counters, memory, etc.) is just a Boolean vector, and the computer itself is a logic circuit C that transforms one configuration to another in a deterministic way. The program and inputs to the computer, once loaded, simply determine the initial configuration. So the history of any computation bounded by a polynomial time function $T(n)$ can be visualized as concatenating $T(n)$ copies of C, with the initial condition presented at the inputs of the first copy. The output of the computation is one distinguished bit of the $T(n)^{th}$ copy. The breadth $B(n)$ of the circuit is polynomial in its size, so if we can decide $B(n) \times T(n)$ Boolean values (bits,) we can decide the problem instance in polynomial time.

Version 2: The original proof by Cook used the problem of Boolean Satisfiabilty, namely whether an arbitrary Boolean *conjuctive normal form,* consisting of a conjunction (AND) of disjunctions (ORs,) of symbols and their negations, has a satisfying assignment. The successful computation by a Turing machine of a solution of a problem instance can be encoded as a collection of conditions to be satisfied, described in terms of the syntactic correctness of the input, the possible successions of ID's (instantaneous descriptions of the Turing machine — the analogues of computer configurations), including the guaranteed halting of the computation. The formula is "TRUE" if and only if all the requirements are satisfied. Moreover, the formula has a size which is polynomial in the size of the original problem instance, and the formula can be computed in polynomial time. Using DeMorgan's laws, the logic formula expressing decidability by a NTM can be converted to a conjunction of clauses each of which is the disjunction of literals, known as conjunctive normal form.

4.1 Satisfiability

It was soon shown that the same conditions could be met by a conjunction of clauses each of which is the disjunction of exactly three literals or negations of literals — or conjunctive normal form with exactly three literals per clause, known as "3-CNF-SAT." The proof is as follows: Any disjunction $(a_1 \vee a_2 \vee \cdots \vee a_k)$ can be replaced by triples by the recursive application of the rule $(a_1 \vee a_2 \vee \cdots \vee a_k = (a_1 \vee a_2 \vee Z) \wedge (a_3 \vee a_v \cdots \vee \bar{Z})$ where Z is a new symbol. It is *not* possible to go down to two symbols per clause, because 2-CNF-SAT is in P.
Proof 1 (casual): Scan the variables; any variable that occurs in only one state (complemented or not) can be set to the appropriate constant (0 or 1.) For each remaining variable x, and for each other variable y, determine whether clauses of both types $(x \vee y)$ and $(x \vee y')$ occur. If so, set $x = 1$ everywhere. If $(x' \vee z)$ and $(x' + z')$ both occur, set

$x = 0$ everywhere. If for some both occur, the expression is not satisfiable. If all variables pass this test, the expression is satisfiable.
Proof 2 Assume each variable appears in both polarities (otherwise set to 0 or 1.) Eliminate one variable at a time via the following lemma: **Lemma** Let E be an x-free 2-CNF expression. Then $E_1 = E \prod_i (x' \vee a_i) \prod_j (x \vee b_j)$ is satisfiable if and only if $E_2 = E \prod_{i,j} (a_i \vee b_j)$ is satisfiable.
Proof: $\Rightarrow$ is easy. Conversely, consider any assignment that satisfies E_2. If all $a_i = 1$, then $_1$ is clearly satisfiable. On the other hand, if any $a_i = 0$ then all $b_j = 1$, and again E_1 is satisfiable. Thus x' can be eliminated, although the expressions must be cleaned up between eliminations to avoid exponential buildup. After each cleanup, there can be at most $\binom{n}{2}$ clauses, so the elimination is polynomial. This technique is called "resolution" in logic.

4.2 Reductions

Once there exists one NP-complete problem (created from the definition of NP, other problems can be proved NP-complete by

1. Showing that they are in NP (have polynomial-time certificates, and
2. reducing earlier ones to them. Notice that it says nothing to go in the other direction, i.e. to reduce a new problem to a known NP-complete problem.

Here's an example, with some of the intricacies:

FORMULA-SAT asks whether an arbitrary Boolean formula has a satisfying assignment. It is clearly in NP: the certificate exhibits the assignment, and it takes polynomial time to substitute the values and evaluate the entire expression. The reduction of CIRCUIT-SAT to FORMULA-SAT seems straightforward, but it's not. (Problem 36.4.1, p. 945 of CLR.) What goes wrong is that a circuit consisting of n levels, each with a pair of gates feeding both the gates below it expands to an exponential number of clauses in a naive expansion into a formula. The correct way is to assign a variable to each *wire* in the circuit, and to describe the operation of each gate as $z \Leftrightarrow (w \circ x \circ \cdots \circ y)$ where z is the output wire, w, $x, \ldots, y$ are the inputs, and $\circ$ is the operator. These are all ANDed ($\wedge$) together with the total circuit output. Clearly the reduction and the size of its result are polynomial in the circuit size.

5 Some NP-complete Problems

About the same time that Cook proved the existence of an NP-complete problem, Karp produced a collection of well-known, empirically intractable problems from Operations Research and computer science, and provided reductions among them. This effort soon exploded into an industry. Here are some classic examples:

- CLIQUE: Does a graph contain K_k, the complete subgraph of size k? Reduction of 3-CNF-SAT with k clauses. For each clause put three vertices in G; connect two vertices if they occur in different clauses and they are not complements. A clique then corresponds to a set of k literals all of which can be set to 1.

- VERTEX-COVER: Does a graph have a subset of k vertices adjacent to every edge in the graph? Reduction of $(n-k)$-CLIQUE. G has a clique of size $(n-k$ iff $\bar{G}$ has a vertex cover of size k. Let $V' \subseteq V$ be a clique in G and consider any edge $(u,v) \in \bar{E}$ and therefore $\notin E$. At least one of u, v is in $V - V'$, otherwise, with both ends in the clique V', $(u,v) \in E$. Therefore the edge (u,v) is covered by a vertex in $V - V'$.

- SUBSET-SUM (same as 0-1 KNAPSACK): Does a set S of integers have a subset summing to k? Reduction of VERTEX-COVER. Given a graph construct the following matrix: Place an $|E|$-dimensional identity matrix below the $|V| \times |E|$ incidence matrix of the graph. Append on the right a column of $|V|$ 1's followed by $|E|$ 0's. The weights of the SUBSET-SUM problem are sums of powers of 4 (1,4,16 and so on, left to right) using the rows of this matrix as coefficients. So the i^{th} weight is

$$4^{|E|} + \sum_{j=0}^{|E|-1} b_{ij} 4^j \text{ and the target sum is } k4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j.$$

  The only way to compose the target sum from the weights is to select the k rows corresponding to the vertex cover plus all the lower rows corresponding to the graph edges. See Fig. 36.14, p. 952 in CLR.

- HAM-CYCLE: Does a graph have a Hamilton Cycle (tour visiting every vertex?) By reduction of 3CNF-SAT. The CNF is replaced by a graph with "gadgets" that enforce paths analogous to the clauses of the CNF.

- TSP: Given an edge-(integer)-weighted graph, does there exist a Hamilton cycle of edge cost $\leq k$? By reduction of HAM-CYCLE to a TSP with unit edge weights.

- Subgraph Isomorphism: Given two graphs, is G_1 isomorphic to a subgraph of G_2. The intuition here is that all $\binom{|G_2|}{|G_1|}$ subgraphs potentially must be considered, which is exponential in $|G_1|$. The actual reduction is from HAM-CYCLE: given G with $|V| = n$, if we could solve SUBGRAPH-ISOM we could ask whether an $n - cycle$ is isomorphic to any subgraph of G.