Quantum Complexity Theory II

In the last lecture, we saw what it means to compute a function efficiently, and saw subproblems of ISING GROUND STATE and k-SAT for which we know efficient algorithms. We also saw that the best known algorithms or these problems in their full generality are not efficient, and in fact are not qualitatively better than taking a brute-force search approach. Today, we'll develop a theory that explains to some extent the difficulty in designing efficient algorithms for these problems.

The complexity class NP

Let us revisit the language ISING GROUND STATE. If someone gave us a state σ claiming that it had energy $H(\sigma)$ at most the required threshold k, we would be able to efficiently check whether that is true. In fact, whenever an instance (G, c, b, k) is in the language, there is such a state—the ground state would be one such. When the instance is not in the language, there is no such state, and any claimed state can efficiently be checked and rejected.

The problem k-SAT enjoys the same property. If we are presented with an assignment, and we can check efficiently whether it satisfies the given formula or not. By definition, whenever the formula is satisfiable, there is evidence supporting this (e.g., a satisfying assignment), and when the formula is not satisfiable any claimed evidence (like a putative satisfying assignment) can be checked and rejected efficiently.

This property is common to a wide array of problems, including the travelling salesman problem, graph isomorphism, scheduling with constraints, etc., and is the defining characteristic of the complexity class **NP**.

Formally, we say that a language L is in **NP** if the following conditions are met. There is an efficient verification algorithm, i.e., uniform family of polynomial-size circuits $\{C_n\}$, that take two arguments x, y as input. The two inputs to C_n are of length n, and $p(n)$, respectively, where $p(n)$ is a polynomial in n. Furthermore, for every n, and for any input x of length n,

1. if $x \in L$, there is a "witness" y of length $p(n)$ such that $C_n(x, y) = 1$, i.e., the verification procedure accepts the witness.
2. if $x \notin L$, every putative witness y of length $p(n)$ is rejected by the verification procedure, i.e., $C_n(x, y) = 0$.

We may think of the witness as a "proof" that x is in the language, and the verification algorithm as a proof system, which checks whether a proof is correct or not. The two conditions above correspond to completeness and soundness of the proof system: every correct statement $x \in L$ has a valid proof, whereas no incorrect statement has one. Moreover, a putative proof can be checked efficiently. In other words, **NP** consists of languages which admit short, efficiently verifiable proofs.

The name **NP** is an abbreviation for "non-deterministic polynomial time", and comes from its original definition in terms of *non-deterministic* Turing machines. It should not be mistaken for "non-polynomial time" eventhough we do not know of efficient solutions to many of the problems in the class, and it is widely believed that **P** $\neq$ **NP**.

Note that every language in **P** is in **NP** by the above definition, as for these languages we do not need the aid of a proof to efficiently verify whether $x \in L$. There are a plethora of problems not known to be in **P** which are in **NP**; some were mentioned above, others are integer linear programming, decoding linear error-correction codes, factoring integers, etc. Whether **P** $\neq$ **NP** remains one of the defining open problems of the field of theoretical computer science today. Besides being of interest for practical computation, it touches upon a fundamental question about mathematics itself: is devising a proof harder than checking that a putative proof is correct?

We mention in passing that there are problems that are not known to have short proofs, and therefore are not known to be in **NP**. For example, we do not know of short proofs of unsatisfiability of boolean formulae, or that the permanent of a matrix is bounded by some threshold.

NP-hardness

In the early 1970s, Cook and Levin independently proved a remarkable fact about 3-SAT that greatly simplifies the study of problems in **NP**. They showed that 3-SAT is **NP**-complete, which essentially means that if there is a polynomial time algorithm for this problem, then every language in **NP** has an efficient algorithm.

A *many-one* or *Karp reduction* from a language L_1 to another language L_2 is an efficient algorithm that given an instance x for the problem L_1 produces an instance y of the problem L_2 such that $x \in L_1$ iff $y \in L_2$. Since y is produced by an efficient, i.e., polynomial-time algorithm, it is only polynomially longer than x. When there is such a reduction, a polynomial time algorithm for L_2 implies a polynomial time algorithm for L_1, and we say that L_1 *reduces to* L_2. Clearly, reducibility is a transitive relation, so that if L_1 reduces to L_2, and L_2 reduces to L_3, then L_1 reduces to L_3.

We say that a language L is **NP**-*hard* if every language in **NP** reduces to L (via Karp reductions). We say that a language is **NP**-*complete* if it is also contained in **NP**.

Theorem 1 (Cook-Levin) *3-SAT is **NP**-complete.*

Proof : Since we have seen that 3-SAT is contained in **NP**, we need only prove that it is **NP**-hard.

Consider some language L in **NP**, and an instance x of the problem of length n. By definition, there is a polynomial time constructible verification circuit C_n that takes as input x and a "witness" y of length m, a polynomial in n. Recall that there is a assignment for y such that $C_n(x, y) = 1$ iff $x \in L$.

Let the gates in C_n be $g_1, \ldots, g_T$, so that the circuit has size T. We construct a 3-CNF formula ϕ over n variables (x'_i), m variables (y'_j), and a further T auxiliary variables (g'_k) with a number of clauses proportional to $n + m + T$.

The formula consists of three types of clause: input, output, and propagation. The output clause is simply g'_T. This is to ensure that if ϕ is satisfiable, then the output of the verification circuit is 1.

There are n input clauses: if the input bit $x_i = 1$, then we include the clause x'_i, otherwise we include $\neg x_i$. These clauses ensure that the variables x'_i take the values of the input bits x_i.

The propagation clauses ensure that the variable g'_k takes the value computed by the gate g_k. Suppose $g_k = \neg$, and that the input wire is z, with the corresponding variable z' in our construction. Then we include the clauses $(\neg z' \vee \neg g'_k)$ and $(z' \vee g'_k)$. These are true iff g'_k is the negation of z'. Suppose $g_k = \wedge$, and that the input wires are z_1, z_2, with the corresponding variables z'_1, z'_2 in our construction. Then we include the clauses $(z'_1 \vee \neg g'_k)$, $(z'_2 \vee \neg g'_k)$, and $(\neg z'_1 \vee \neg z'_2 \vee g'_k)$. All three clauses are simultaneously true iff g'_k takes the value $z'_1 \wedge z'_2$, as intended.

Summing up, we see that $\phi(x', y', g')$ is satisfiable iff there is an assignment to y such that $C_n(x, y) = 1$. Equivalently, iff $x \in L$. This means that L reduces to 3-SAT, which is what we set out to prove. ■

The problem 3-SAT is only one of hundreds of problems known to be **NP**-complete today. There are only a handful of problems in **NP** that are neither known to be in **P** nor known to be **NP**-complete. Examples of such problems include integer factorisation and graph isomorphism.

The Cook-Levin theorem allows us to easily show **NP**-hardness of other problems: we need only design a Karp-reduction from 3-SAT to the problem, and invoke transitivity. A sequence of such reductions from 3-SAT terminating with ISING GROUND STATE shows that the latter problem is also **NP**-complete, even when the underlying graph structure is planar, and the interactions are in the set $\{-1, 0, +1\}$ and the strengths are all 1. As an illustration, we describe the first reduction in this sequence, which shows that the CLIQUE problem is also **NP**-complete.

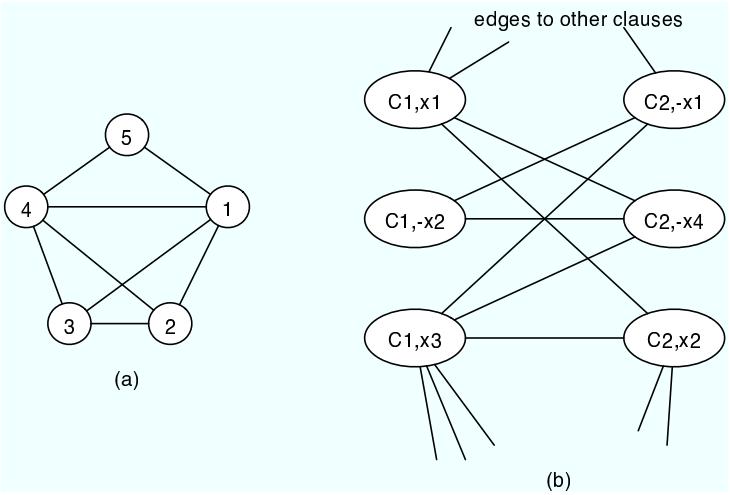

Figure 1: (a) A graph with a clique of size 4, and (b) a part of the graph obtained from a 3-SAT instance with clauses $(x_1 \vee \neg x_2 \vee x_3)$ and $(\neg x_1 \vee \neg x_4 \vee x_2)$.

A clique in a graph is a subset of vertices all of which are connected by edges. Its size is the number of vertices in the subset.The graph in Figure 1(a) has a clique $\{1, 2, 3, 4\}$ of size 4, but not one of size 5.

The input to CLIQUE is a graph $G = (V, E)$, and an integer $k \geq 1$, and the problem is to determine whether there is a clique of size k in G.

Theorem 2 CLIQUE *is* ***NP****-complete.*

Proof : That CLIQUE is in **NP** is straightforward. We show **NP**-hardness by reducing 3-SAT to CLIQUE. Given a 3-CNF formula ϕ, we construct a graph G, and a threshold k as follows.

For each clause c in ϕ, and each literal l in c, we have a vertex (c, l) in G. There is an edge between two vertices (c, l) and (c', l') iff $c \neq c'$, and l, l' are not a variable and its negation. We set k to be the number of clauses in ϕ. We see that the number of edges, and therefore the size of the instance G, k of CLIQUE is at most quadratic in the number of clauses in ϕ. Figure 1(b) depicts a portion of a graph obtained from a formula in this manner.

We claim that ϕ is satisfiable iff G has a clique of size k. Suppose ϕ has satisfying assignment a. Consider the subset of vertices (c, l), exactly one for each clause c such that the literal l is true under assignment a. These vertices form a clique in G, of size k.

Now suppose that G has a clique of size k. Note that no two vertices with the same clause can be part of a clique, so the clique has exactly one vertex for each clause in ϕ. Moreover, the set of literals in these vertices does not contain both a variable and its negation. Setting all these literals to true, we get a satisfying assignment for ϕ.

All the above imply that 3-SAT is reducible to CLIQUE, and the theorem follows. ■

Probabilistic algorithms

A major development in the design of algorithms was the discovery in the late 1970s that using randomness during the course of a computation could lead to significantly simpler and more efficient algorithms. In many cases, including random numbers to guide computation allows the design of efficient algorithms where no such deterministic algorithm is known. For testing primality of an integer, for example, until 2002 the only known polynomial time algorithms were randomized. Similarly, the only known efficient way

of estimating the partition function of the Ising model in the ferromagnetic case, for general graphs, is randomized, and the fastest known, albeit exponential time, algorithm for k-SAT is also randomized.

To see how using randomness may help, consider the problem of checking whether the product AB of two given $n \times n$ matrices A, B over $\mathbb{Z}_2$ equals a third matrix C. The straightforward method of solving this would be to explicitly multiply the two matrices A, B, and check the result against C. This takes time $\mathrm{O}(n^\omega)$, where $\omega < 2.38$ is a constant, using a sophisticated method due to Coppersmith and Wingorad. (The straightforward method would have complexity $\mathrm{O}(n^3)$.) Here is a simpler and faster alternative. Pick two vectors $u, v \in \mathbb{Z}_2^n$, and check if $u^{\mathrm{T}}ABv = u^{\mathrm{T}}Cv$. Since matrix-vector multiplication takes time $\mathrm{O}(n^2)$, this check can be performed in the same amount of time. If $AB = C$, the check is always passed. If $AB \neq C$, the check detects inequality with probability at least $1/4$. (We leave the proof of this fact as an exercise to the reader.) By repeating the check t times, and reporting "unequal" if we detect this in any one of the t trials, we can amplify the probability of detection to at least $1 - (3/4)^t$.

Another example is that of computing a prime number larger than a given integer n. It is known that there is a prime between n and $2n$, and therefore we may run through numbers $n+1, n+2, \ldots$ and output the first prime we find. Recall that testing primality is in **P**, so that each check is efficient. However, we may run the primality algorithm up to $\Omega(n)$ times, which is *exponential* in the input length $\log_2 n$. The use of randomness allows an efficient solution: we pick a random number in the interval $[n, 2n)$, and check for primality. If we succeed, we stop, otherwise we repeat the process up to a total of $t \log_2 n$ times. Since there are $\Omega(n/\log_n)$ primes in this interval, we succeed with probability at least $1/\log_2 n$ in any one iteration. So the probability that we fail to find a prime within $t \log_2 n$ iterations is at most $\exp(-\Omega(t))$. The net time complexity is polynomial.

Yet another example is that of polynomial identity testing. In this problem, we are presented with a polynomial over the integers, described by an arithmetic circuit $C(x)$ with n indeterminates $x = (x_i)$. This is a circuit with multiplication and linear combination gates, much like the boolean circuits we have studied. The problem is to determine if the polynomial defined by the circuit is identically 0. The intermediate results of the circuit may have exponentially many monomials, so explicitly computing the polynomial is inefficient. Here is a randomized test, that *is* efficient. Pick a prime p larger than $2d = 2^{m+1}$, where m is the multiplication gates in the circuit. The number d is an upper bound on the degree of the polynomial described by the circuit. Pick independently, and uniformly at random, n numbers $r_1, \ldots, r_n \in \mathbb{Z}_p$. Evaluate $C(r)$, and check if it is 0. If $C(x)$ is identically 0, then the check passes with probability 1. If not, by the Schwartz-Zippel Lemma, the probability that $C(r) \neq 0$ is at least $1/2$. As before, we may drive up the probability of detection by repeating the test.

These are all examples of *bounded-error probabilistic* polynomial time (or **BPP**) algorithms. We say a language L is in the complexity class **BPP** if there is a uniform family of polynomial-size circuits $\{C_n\}$, that take two arguments x, r as input. The two inputs to C_n are of length n, and $p(n)$, respectively, where $p(n)$ is a polynomial in n. The input r is is chosen uniformly at random from $\{0,1\}^{p(n)}$, and x is an instance of the problem L. For every n, and for any input x of length n,

1. if $x \in L$, $\quad \Pr_r[C_n(x, r) = 1] \geq 2/3$.
2. if $x \notin L$, $\quad \Pr_r[C_n(x, r) = 1] \leq 1/3$.

In other words, the output of the circuit is the correct answer with probability at least $2/3$ in either case. The choice of the values $2/3, 1/3$ for the probabilities above is not set in stone. We may allow probabilities q_1, q_0, respectively, such that $q_1 - q_0 \geq 1/q(n)$ for some polynomial q.

Suppose we start with a probabilistic algorithm A that accepts with probability at least q_1 when the input is in the language, and accepts with probability at most q_0 when it is not, where q_1, q_0 are as above (for inputs of length n). We can amplify the probability of correctness to $2/3$ by running independent trials polynomially many times. In fact, we can amplify it all the way to $1 - \exp(-s(n))$ for any polynomial s. We run A a total of $t = \mathrm{O}(q(n)\, s(n))$ times, using independent random bits r for each trial. We output 1 iff the number of outcomes of the t trials is at least $t(q_1 + q_0)/2$. A straightforward "tail bound" on the Binomial distribution (the Hoeffding-Chernoff bound) tells us that the probability we output the incorrect answer is at most $\exp(-s(n))$.

Of course, **BPP** includes all languages in **P**. Does randomness help us solve **NP**-hard problems? The answer to this is not known, but indications are that it does not. For example, integer factorisation is in **NP**, but we do not know of an efficient probabilistic algorithm for it. There is growing evidence that the use of randomness may be eliminated entirely using "pseudo-random" number generators, so that **BPP** may equal **P**. Nonetheless, we do not know whether certain problems in **BPP**, such as polynomial identity testing, are also in **NP**, let alone in **P**.

You might ask whether it is reasonable to find sources of uniformly random bits in nature, i.e., whether the model of probabilistic algorithms is reasonable. There is a well-developed theory of randomness extraction which enables us to efficiently distil nearly uniform random bits from imperfect sources, which suggests it is. This is precisely what lead to the reformulation of the Church-Turing thesis to essentially state that all computers are created equal, and in particular equal to the *probabilistic* Turing machine.