

Computational Complexity

The most sharp distinction in the theory of computation is between computable and non-computable functions; that is, between *impossible* and *possible*. From the example of the halting problem, we have seen that some problems are indeed impossible for a computer to solve.

However, for practical purposes, it is often more important to make a different (if somewhat less clear-cut) distinction: between *hard* and *easy* problems. This is the subject of computational complexity.

What we are calling a *problem* is actually a class of questions which share a particular structure; specific questions within this class we call *instances* of the problem.

Decision Problems

The type of problems we will consider are *decision problems*: given a particular instance, the computer returns the answer *yes* or *no*. Generally, one is being asked if a particular mathematical object has a particular property or not. For instance, rather than asking for the shortest path passing through a group of cities (the famous *Traveling Salesman Problem*), we ask “is there a path shorter than X ?”

Needless to say, we usually want to know more than if a solution shorter than X exists; we would actually like to know the path. But there are two reasons we consider decision problems.

First, they are easier to analyze; we can think of them as evaluating a function which returns a single bit. Returning a path would require us to consider how paths are represented, and other complications.

Second, it turns out that if one can solve the decision problem, one can solve the optimization problem with only a polynomial amount of extra overhead. We call this a *reduction* of the optimization problem to the decision problem. (More on reduction soon.)

For a given problem, there are often particular instances which are easy to solve. We define the difficulty of a problem to be the *worst case* difficulty; that is, the difficulty of the *hardest* instance of a particular size.

Hard vs. Easy

An instance of a problem will be characterized by a *size*, which is an integer n . For instance, in the Traveling Salesman Problem (TSP) the size n is the number of cities; for factoring, the size is the *number of digits* of the number to be factored; for sorting, it is the number of objects to be sorted. Usually, n is proportional to the length of the description for the instance in question. (For instance, it could be the number of bits needed to describe it.)

Each instance of a problem is described in a standard way as a series of zeros and ones; and we consider Turing machines which input this description and output the answer “yes” or “no” after some number of steps. The program of such a Turing machine corresponds to an *algorithm* for solving the problem.

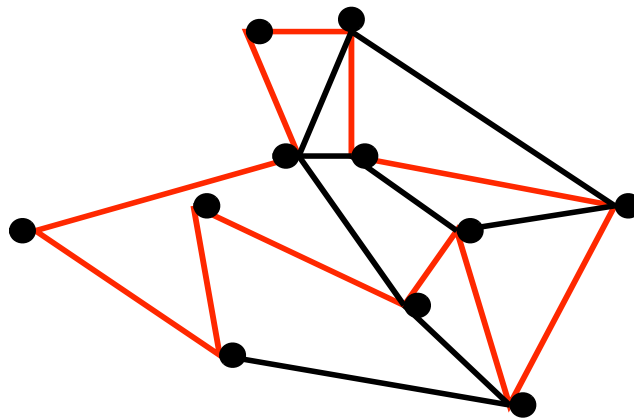
In calling a problem hard or easy, we are interested in the *functional dependence* of the number of steps on n . For the most part we will only be concerned with the coarsest distinction: does the number of steps scale up in a way bounded by a *polynomial* in n , or *exponentially* (more properly, superpolynomially). We will consider polynomial algorithms to be easy, and exponential ones to be hard. At a greater level detail, we may say, e.g., that an algorithm has complexity $O(n)$, $O(n \log n)$, $O(n^2)$, $O(\exp(n^{1/3}))$ or $O(2^n)$.

Obviously, for small values of n a lot depends on the prefactors of these scaling laws; an exponential algorithm might be much easier than an $O(n^5)$ algorithm for small n . But for large problems, polynomial algorithms will always win out.

It isn't always obvious if a problem is hard or easy. Here are two examples:

The Euler Cycle Problem. Given a connected graph G , is there is a closed path that traverses every edge exactly once?

The Hamiltonian Cycle Problem. Given a connected graph G , is there is a closed path that traverses every vertex exactly once?



These problems sound almost identical. But the Euler Cycle Problem is easy, while the Hamiltonian Cycle Problem (HC) is hard—in fact it is *NP-complete*.

To P, or not To P?

Computational complexity theory groups decision problems (and others) into broad classes based on their difficulty. The simplest category are *problems which are solvable on a Turing machine in polynomial time*. The class of such problems is called P.

In principle, such problems could still be extremely difficult: a problem of $O(n^k)$ for very large k could still take far longer to solve than the lifetime of the universe. But most common algorithms in the class P have fairly low complexity: searching ($O(n)$), sorting ($O(n \log n)$), addition ($O(n)$), multiplication ($O(n^2)$), the Euler Cycle Problem ($O(n^3)$), the Connecting Path problem ($O(n^3 \log n)$). These problems are reasonably easy for modern computers.

NP

Then there are problems for which no efficient algorithm is known. The most well-known class of these problems is NP: the class of all problems solvable in polynomial time by a *nondeterministic Turing machine* (NTM).

What is an NTM? We can illustrate the idea with the Traveling Salesman Problem. One method of solving the TSP is to simply try each possible path in turn until one finds the shortest, and then see if it is shorter than the requested length. On an ordinary Turing machine, this would take a very long time—exponential in the number of cities visited.

An NTM, however, has the following useful ability: rather than trying each possibility in turn, it can simply *guess* the shortest path—and it always guesses correctly! (Needless to say, such a machine would be impossible to build in reality.)

An NTM, then, guesses the best answer, and then checks if it solves the problem. In order for such a machine to be efficient, this checking procedure must take a time no more than polynomial in n .

This gives another, less fanciful way of defining NP: the class NP are those decision problems for which a “yes” result can be proven by providing a binary string of at most polynomial length which can be checked by a TM in polynomial time. For TSP, such a string would describe a path shorter than the threshold; for HC, the string would describe the Hamiltonian cycle. Such a proof is called a *witness* for the problem.

Needless to say, any problem in P is also in NP. It is widely believed, however, that the converse is not true.

We can frame this as what is called a *Merlin-Arthur game*. Merlin is an all-powerful wizard, able to solve any computable problem instantly; Arthur is an ordinary mortal with an ordinary TM. Merlin wishes to convince Arthur that the answer to a decision problem is “yes.” He does so by providing a witness that Arthur can check in polynomial time. If he cannot provide one, the answer is “no.”

(This kind of procedure can be used to define a whole hierarchy of complexity classes called Π_k of increasingly difficult* problems, where $P = \Pi_0$ and $NP = \Pi_1$. But this is getting beyond the scope of this course!)

*In fact, there is no proof that these problems *are* more difficult. If you can prove whether or not $P = NP$, the Clay Institute has a million-dollar check waiting for you!

Oracles and problem reduction

An oracle is like a special-purpose version of Merlin: in is a *black box* that, given an input X , immediately returns the value of a function $f(X)$, regardless of how difficult computing $f(X)$ might be. We imagine this as being a special add-on feature to the Turing machine, which in addition to its usual actions can choose to invoke the oracle. Such a call is called a *query*.

Computer scientists often like to consider problems with oracles, since it is often easier to bound the number of oracle queries than the overall computational complexity. This is called *query complexity*.

This may seem a bit artificial, but it isn't necessarily. We might wish to deduce some property of the function $f(X)$, where the definition of f doesn't make this obvious. Then it makes sense to treat $f(X)$ as an oracle.

A question one might ask, then, is the complexity of a particular problem *if* I have access to an oracle for the solution of a *different* problem. For instance: can I solve the Traveling Salesman Problem efficiently if I have an oracle for the Hamiltonian Cycle Problem?

If we can solve problem A efficiently (i.e., in polynomial time) given an oracle for problem B, then we say that problem A is *reducible* to problem B. Much of complexity theory is concerned with determining which problems are reducible to which other problems.

NP-complete problems

Remarkably, there are certain problems in NP to which *all* other problems in NP are reducible. We say that such problems are *NP-complete*. Given an efficient algorithm to solve one such problem, we would have an efficient algorithm to solve *any* problem in NP.

The canonical NP-complete problem is *satisfiability* (SAT): *given* a Boolean formula f in n Boolean variables x_1, \dots, x_n , whose length is polynomial in n , is there some assignment of values to these variables which *satisfies* f , that is, makes $f(x_1, \dots, x_n) = 1$?

We can see intuitively why any problem in NP can be rephrased in terms of SAT. Let the variables x_1, \dots, x_n be the binary string representing the witness for an NP problem, and $f(x_1, \dots, x_n)$ the polynomial-time checking procedure. It is always possible to express f as a polynomial-length formula.

There are many problems which have been shown to be NP-complete. For instance, the Hamiltonian Cycle and Traveling Salesman problems are both NP-complete. So is a problem called 3-SAT, which is just like SAT, except that the formula is required to be in *conjunctive normal form* with clauses containing no more than three literals each:

$$f(x_1, \dots, x_n) = (\cdot) \& (\cdot) \& \dots \& (\cdot),$$

$$(\cdot) = \text{e.g., } (x_i \vee x_j \vee \neg x_k).$$

(2-SAT, by contrast, is in P.)

There are problems in NP which are *not* NP-complete. An important example of this is *factoring an integer into prime factors*. Another is the *discrete logarithm*. We will return to these problems when we study quantum algorithms.

NP-complete problems are considered to be the most difficult problems in NP, since an ability to solve them implies an ability to solve any other NP problem. However, it is by no means true that *every instance* of such a problem is difficult. It is often the case that *typical* instances can actually be solved fairly efficiently. Remember, difficulty is defined by the worst case efficiency.

Curiously, however, for some problems that aren't NP-complete, a large proportion of instances are difficult. Factoring is an example of this. It is quite easy to produce a number of length n whose prime factors are of length $\sim n/2$, and which would take longer than the lifetime of the universe to factor using the best known classical algorithm.

Probabilistic algorithms and BPP

All of the Turing machines so far—even the *nondeterministic* ones—have acted in a non-probabilistic fashion. That is, they return the answer to the problem with probability one.

We now consider a *probabilistic* Turing machine (PTM), one with access to random bits. We can run algorithms on such a machine which succeed in answering problems only with some probability $p > 1/2$. (In fact, we can boost the probability to $p > 1 - \epsilon$ for any $\epsilon > 0$ with only polynomial overhead, simply by running the program several times.)

Unlike NTMs, PTMs can easily be built. And perhaps surprisingly, they make possible some efficient algorithms which are not readily convertible to deterministic algorithms.

A good example of this is *primality testing*. We wish to determine if some n -bit number q is prime. The heart of the usual primality testing algorithm is *Fermat's Little Theorem*, which states that if q is prime, then

$$a^{q-1} = 1 \pmod{q}, \quad 0 \leq a < q.$$

Unfortunately, this can be true for composite q for some values of a . We can't efficiently check all a values; but it turns out that if we *sample* a uniformly from the interval $[0, q-1]$, there is a very high probability of determining if q is prime with only a polynomial number of samples. (In fact, this test isn't perfect, so in the most common algorithm it is supplemented by an additional probabilistic test, the Miller-Rabin test.)

The class of all problems solvable in polynomial time on a PTM with probability $p > 1/2$ is called BPP (for Bounded Probabilistic Polynomial). It is not known if $P = BPP$.

PSPACE

We have been looking at restrictions on the performance of algorithms in *time* by putting bounds on the total number of steps. We might try an analogous restriction in *space*, by asking what problems can be solved by a TM using only a polynomial length of tape. This complexity class is PSPACE.

It turns out that this restriction is not very restrictive. Not only does PSPACE include all the problems in P and NP, it includes many much harder problems as well! There are problems in PSPACE that not only cannot be solved in polynomial time, the solutions cannot even be *checked* in polynomial time.

The canonical problem in PSPACE is *quantified satisfiability* or QSAT. It is superficially similar to ordinary SAT. Given a Boolean formula $f(x_1, \dots, x_n)$ of polynomial length in n binary variables (where I assume n to be even), the answer is the truth value of

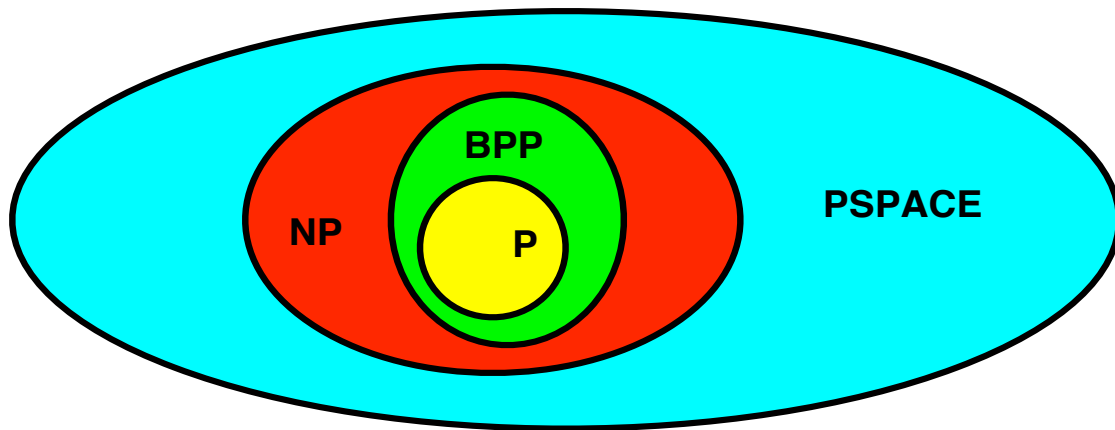
$$\exists x_1 \forall x_2 \exists x_3 \cdots \exists x_{n-1} \forall x_n f(x_1, \dots, x_n) = 1.$$

By contrast, ordinary SAT can be written

$$\exists(x_1, \dots, x_n) f(x_1, \dots, x_n) = 1.$$

QSAT is in fact *PSPACE-complete*, just as SAT is NP-complete. We can think of QSAT as a generalization of a Merlin-Arthur game; x_1, x_2, \dots represent successive moves in a game, and the question is whether the person who goes first will always win. The Boolean function represents the victory criterion.

Here is a diagram of the complexity classes we've looked at, with the most widely-believed inclusions:



However, relatively little has been proven. $P \subseteq NP \subseteq PSPACE$, but it is not known if $P \neq NP$. In fact it hasn't been proven that $PSPACE \neq P$! It is widely believed, though, that these inclusions are strict. Also, while $P \subseteq BPP$, it is not known if $BPP \subseteq NP$ (though it is known that $BPP \subseteq PSPACE$). It is widely thought that $P = BPP$, because probabilistic algorithms still work well with *pseudorandom* numbers. (We will see how quantum computers fit into this complexity scheme.)

Complexity theory and circuits

Our tour of complexity theory has been framed entirely in terms of Turing Machines; but we started by modeling computation in terms of Boolean circuits, which generalize more easily to the quantum case.

A TM can be given for a problem that will solve instances of *any* size n . A given circuit, by contrast, will usually only solve instances of a single size n (or at best of size $\leq n$). We can make a connection to complexity theory by defining a *uniform family* of circuits for each problem. For every value of n , we define a circuit of size polynomial in n which solves all instances of the problem of size n . Moreover, there must be a TM which, given n as the input, outputs a description of the circuit in a time polynomial in n . It turns out that the problems for which there is a uniform family of circuits is equivalent to P.

The requirement that the circuits can be generated by a TM is important. If we remove that assumption, we get a new class (called P/poly) of *nonuniform* circuit families, which includes P, but also problems not in P. Indeed, by some definitions this class includes noncomputable functions! Of course, for practical purposes, we would have no idea how to construct the successive members of a nonuniform circuit family.

There is no logical reason why naturally-occurring phenomena might not solve some noncomputable functions for us. Indeed, it has been speculated that the outcome of the laws of physics may be noncomputable. No demonstration of this, however, is known.

Next time: Quantum algorithms.