80240233: Computational Complexity **Lecture 1**
ITCS, Tsinghua Univesity, Fall 2007 9 October 2007

Instructor: Andrej Bogdanov **Notes by:** Andrej Bogdanov

The aim of computational complexity theory is to study the amount of resources necessary – such as time, memory, randomness, "quantumness" – to perform given computations. The origins of this theory can be traced back to the early 1970s when it was realized that certain simple problems would take an inordinate amount of time to solve on any computer, even though these problems are in principle solvable. Moreover, the inherent difficulty of these problems does not have anything to do with the computing technology that was available in the 1970s. With today's technology we can do no better, and complexity theory indicates that even 100 years from now we will be stuck at the same place. It appears that nature imposes intrinsic obstacles at performing certain computations, and a central question in complexity theory is to understand why and how these obstacles arise.

To initiate a study of computation we must agree on some essentials. First, what are the computational problems and processes that we are interested in studying? And second, what is a good model that will allow us to reason about computation in a way that is general enough to encompass all computing technology of the day?

1 Computational problems

The answer to the first question depends somewhat on your personal taste. However unlike computability theory, which mostly studies problems *about* computations, much of the focus of modern complexity theory has been on "natural" problems that are inspired by other scientific disciplines and areas of life – be it mathematics, economics, physics, or sociology – and is somewhat unique in its ability to yield interesting insights on problems arising from such a wide range of sources. Of course, complexity theory also studies problems arising from computations, but these are often used just as a stepping stone to say something about problems that scientists and engineers from other disciplines might be interested in.

Here are some examples of problems that complexity theory is interested in.

Decision problems A decision problem (or a language) is one that can be answered by "yes" or "no":

MATCHING (PERFECT MATCHING) Given an undirected graph, does it contain a perfect matching, i.e. a pairing of its vertices such that each pair is connected by an edge?

SAT (BOOLEAN FORMULA SATISFIABILITY) Given a boolean formula in conjunctive normal form (CNF), for instance

$$(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_4})$$

is there an assignment that satisfies the formula (i.e. it satisfies all the clauses simultaneously)?

For instance the assignment

$$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$$

satisfies the above formula.

As a standard convention, we represent inputs to decision problems as binary strings; for instance a graph of n vertices can be represented as a string of length $\binom{n}{2}$, where each position in the string indicates a potential edge in the graph, and the value at this position indicates the presence or absence of this edge. Sometimes several representations are possible – for instance a graph can be represented as an adjacency matrix or an adjacency list – and usually it will not matter which representation we choose.

We will denote decision problems either as subsets L of the set $\{0,1\}^*$ of all possible strings – for instance, for the matching problem, L is the set of all strings representing graphs that have a perfect matching – or as functions $L : \{0,1\}^* \to \{0,1\}$ where $L(x) = 1$ if the string x is in L and 0 otherwise.

Search problems In a search problem we not only want to know if a solution exists, but find the actual solution as well:

FIND-MATCHING Given a graph, find a perfect matching if one exists.

FIND-SAT Given a boolean formula, fina an assignment that satisfies it if possible.

We will represent search problems as relations R over pairs of strings $(x, y) \in \{0,1\}^* \times \{0,1\}^*$. The input x represents the problem (the graph) while y represents potential solutions (the perfect matching). The pair (x, y) satisfies R if y is a solution of x (graph x contains the matching y).

Optimization problems Optimization problems ask for the best possible solution to a problem. A decision or search problem can have several optimization variants.

MAX-MATCHING Given a graph, find a maximum matching, i.e. the largest perfect matching present in one of its subgraphs.

MAX-SAT Given a boolean formula, find an assignment that satisfies as many of its clauses as possible.

MIN-EQUIV-SAT Given a boolean formula, find the smallest formula that is equivalent to it, i.e., one that shares the same set of satisfying assignments.

Sometimes we may be willing to settle for an *approximate* optimization – namely instead of looking for the best solution, we may be happy with one that is close enough:

APPROX-SAT Given a boolean formula, find an assignment that satisfies 90% of the maximum possible number of clauses that can be satisfied.

Counting problems Counting problems ask for the number of solutions of a given instance:

#MATCHING Given a graph, count the number of perfect matchings it contains.

#SAT Given a boolean formula, count how many satisfying assignments it has.

Now let us jump ahead a little bit and see what kind of insights complexity theory can give us about these problems.

At a very general level, we are looking at two problems – MATCHING and SAT – of the following kind: We are given some object (a graph, a formula) and want to check, or count, or find if the object contains structures of a certain type (a matching, a satisfying assignment). In principle, we can solve both of these problems by trying out all possible structures of interest. However, even for moderately sized objects (a 100 node graph, or a 100 clause formula) this would take an inordinate amount of time even on very fast computers.

In 1965 Edmonds found an ingenious way of finding perfect matchings in graphs that takes much less effort than looking at all possible matchings in the graph. For an n vertex graph, Edmonds' algorithm performs a sequence of about n^3 operations which consist of marking certain edges in the graph and looking at their neighbors, at the end of which the maximum matching is highlighted. With today's technology we can run this algorithm on graphs with tens or hundreds of thousands of nodes.

In contrast to the MATCHING problem, for the SAT problem we know of no better algorithm than one that essentially performs an exhaustive search of all possible assignments. Most complexity theorists (and computer scientists in general) believe that no amount of ingenuity will help us here; rather it is a fundamental fact of nature that SAT cannot be solved by any algorithm substantially better than exhaustive search.

This is the famous "P versus NP" question – the most famous problem in complexity theory, and one to which we don't know the answer to. So if complexity theory fails to explain why solving SAT requires exhaustive search, what is it good for? What complexity theory *can* do – and is very good at – is present evidence for the hardness of SAT in relation to other questions we might be interested in. Here are some examples.

- Suppose our intuition is incorrect and the SAT problem is in fact tractable. What can we then say about FIND-SAT, which is apparently harder than SAT? It turns out that if we can solve the decision problem SAT, then we can also solve the search problem FIND-SAT. How about MAX-SAT? It turns out that we can solve this one as well.
- The problem MIN-EQUIV-SAT also looks harder than SAT. Solving it seems to require two levels of exhaustive search: One that loops over all possible instances of SAT and another one that checks that the two instances are equivalent. However, it again turns out that if SAT is tractable, then so is MIN-EQUIV-SAT.
- How about APPROX-SAT? This appears easier than MAX-SAT. Again it turns out that if we can do one we can do the other. This is a consequence of the "PCP theorem", one of the greatest achievements of complexity theory.
- Finally, what about counting solutions, namely #SAT? Here complexity theory gives us evidence to believe that #SAT is in fact harder than SAT. So even if we were wrong all along and one day found a clever algorithm for SAT, #SAT might be still standing.

- More surprisingly, complexity theory shows that #MATCHING is *just as hard as* #SAT; so for the matching problem deciding, finding, and optimizing is easy, but counting solutions seems hard.

2 Models of computation

To reason rigourously about the complexity of computations we need a model that faithfully captures the resources that we have at our disposal. This appears somewhat difficult as the resources we have available change dramatically over time. In the early 1800s the mathematician Gauss was admired for his prowess at performing difficult calculations. In the early 1900 mechanical calculating devices put the best humans to shame, and by the 1950s electronic computers were already doing calculations that noone thought possible before. Today any ordinary cell phone can do much much more than a 1950s computer. Given such huge disparities among the devices at our disposal, how can we hope to make any sort of general statement about computation?

Complexity takes the perspective that the difference in resources between various computational devices is insignificant compared to the vast gap between the tractable and the intractable. For problems like MATCHING, it might matter what kind of computational device we use; perhaps Gauss could solve graphs of size 10, the 1950 machines could do size 100 while today's computers can solve problems of size 10,000 or more. However, for instances of SAT the differences are much less dramatic; maybe Gauss could do size 8, while today we can do problems up to size 50 or so.

When trying to separate the tractable from the intractable it does not really matter what model of computation we use, as long as it is a reasonable one – in the sense that it can both simulate and be simulated by a realistic computational device in a relatively efficient way. A standard model which is convenient for most purposes in complexity theory is the Turing Machine. This is a device with a finite control and several infinite tapes, one of which contains the input to the computation, one that is reserved for the output, and one or several working tapes. Each of the tapes contains a moving head, and at each step of the computation the machine decides what to do – move to a different control state, move the heads left or right, modify the contents of the work tape, write on the output tape – only as a function of its current control state and the contents of the tapes where the heads point to. The most important property of this definition is that *computation is local*: What happens next is determined by applying some local rule to what we have computed up to now. At some point the machine goes into a special halting state and the computation is over.

For decision problems, we require that at the end of the computation the output tape contains either the symbol 0 (for "no", or "reject") or the symbol 1 (for "yes", or "accept"). We say that machine M *decides* decision problem $L : \{0,1\}^n \to \{0,1\}$ if for every input x, $M(x) = L(x)$, namely the machine always halts with output $L(x)$.

The most important resource we will be interested in is the *computation time* of the machine, namely number of steps it takes for a machine on a given input to reach the halting state.

In complexity theory we are interested in proving lower bounds on the amount of resources required for a Turing Machine to produce its output. For instance we might aim to prove something like this:

> ”For every Turing Machine with 1000 states there exists a SAT instance of size 50 on which the machine fails to find a satisfying assignment in time 10^6.”

However attempts to prove concrete statements of this type have not met with much success, nor have they yielded particularly interested insights on the nature of computational hardness. It is much more common to reason about computations *asymptotically*; we see what happens to the running time of computations not for particular inputs, but as the size of the input grows. So we typically look at statements like this:

> ”For every Turing Machine there exists a sufficiently large number n and a SAT instance of size n on which the machine fails to find a satisfying assignment in time $O(n^5)$.”

I find this to be a somewhat unsatisfactory aspect of complexity theory; perhaps the size n of the instance for which the statement might hold is much too large to be of any practical significance. However at the present stage of development these are the only kinds of questions that we have a hope at answering. The asymptotic nature of the questions and theorems of complexity theory is not something that reflects our objectives and desires, but merely our current level of understanding.

3 P and NP

The next thing we need is a notion of efficiency. The convention in complexity theory is to consider a computation efficient if the running time of the computation is bounded by some polynomial of the input size. Such Turing Machines are said to run in polynonmial time.

There are several reasons for choosing polynomial time as a notion of efficiency. First, we want to do better than exhaustively running through all possible solutions, a procedure that typically takes time exponential in the size of the inputs. Polynomials grow slower than exponentials, so an algorithm that runs in polynomial time must be using something about the structure of the problem in question that allows it to sidestep the ”brute force” procedure of going through all possible solutions.

Of course, we could also consider more restrictive or more liberal notions of efficiency. However if the notion is too restrictive – for instance linear time – then it becomes too dependent on the details of our model of computation. Polynomial time does not seem to suffer from this problem. The *Cobham-Edmonds thesis* (also known as the *extended Church-Turing thesis*) states that any reasonable model of computation can be simulated on any other with a slowdown that is at most polynomial in the size of the input. On the other hand more liberal notions – for instance time $n^{O(\log n)}$ – do not seem to capture too many ”natural” computations outside polynomial time; we don’t seem to gain much by considering these as efficient.

There is another reason why polynomial-time appears to be a good notion of efficiency. In practice we sometimes compose algorithms by feeding the output of one algorithm as an input to another. Polynomial-time computation behaves well under this kind of operation. If both algorithms run in polynomial time, so will their composition.

It is important to keep in mind that not all computations that can be done in polynmomial time are considered efficient in practice. Rather, the choice of polynomial time as a notion of efficiency is an operational choice that allows us to reason about tractability versus hardness without thinking too much about details of the computational model we are using.

In the setting of decision problems, this notion of efficient computation is captured by the class P. The class P consists of all decision problems that can be decided by a Turing Machine that runs in polynomial time.

Many of the problems we are interested in solving are described by NP-relations. We say a relation R is an NP-relation if it is efficiently computable and it describes a search problem whose solutions are short (if they exist). In other words,

1. There is a polynomial-time algorithm that, on input (x, y), decides if $(x, y) \in R$, and
2. There is a polynomial p such that if $(x, y) \in R$, then $|y| \leq p(|x|)$.

If $(x, y) \in R$, then y is called a *witness* or *certificate* for x. For instance in the SAT problem, x is a boolean formula and y is a satisfying assignment for x; thus y witnesses the fact that x is satisfiable.

A decision problem L is an NP decision problem if L is the decision version of some NP-relation R. The class NP consists of all NP decision problems.

There is an alternate description of NP in terms of an imaginary computational device called a non-deterministic Turing Machine (NTM). This machine contains a special tape called a non-determinism tape which is one-way read-only and is initialized at the beginning of the computation. We say that NTM N accepts its input x if there exists a setting of the non-determinism tape that makes $N(x)$ accept, and we say N rejects x otherwise. The running time of N on input x is the maximum running time of N over all possible settings of the non-determinism tape.

It is not hard to see that $L \in$ NP iff L is decided by some polynomial-time NTM. The non-determinism tape of N corresponds to the witness y for x, and N computes the NP-relation $R(x, y)$.

The non-deterministic Turing Machine is not a realistic model of computation. However it is a useful tool for describing and reasoning about problems in the class NP. We will also see other settings where introducing imaginary, unrealistic devices will help us gain insight about computations.