CSCI 5170: Computational Complexity **Lecture 1**
The Chinese University of Hong Kong, Spring 2016 12 January 2016

Computational complexity is the mathematical study of computational efficiency. It is concerned with identifying models of *efficient* computation (natural, engineered, or abstract) and understanding their power, their limitations, and their interrelationships.

The emphasis on efficiency highlights the contrast between computational complexity and computability. Computability theory played an extremely influential role in the early development of computational complexity theory. The very name "complexity theory" (which I do not find descriptive or representative) bears a striking resemblance to "computability theory." Many textbooks on the subject adopt the perspective, at least early on, of complexity as a refinement of computability.

Modern computational complexity theory has grown into a vast subject that is tied to other areas of mathematics (combinatorics, analysis, logic, algebra) and computer science (learning theory, cryptography, data analysis, quantum computing) at least as closely as it is to computability. While many of the questions from the early days — most notably, the "P versus NP problem" — still play a central role, the problems of interest, the motivation for their study, and the methods applied are now significantly different than what was standard 20 years ago.

Computational complexity has an unfair reputation of being irrelevant for all but "those who want to do theory." In reality, insights from complexity affect virtually all areas of modern computer science, as well as parts of other disciplines (mathematics, physics, information theory, economics).

My aim in this course is to highlight those concepts and methods in computational complexity which I believe have wider significance. These should give you the ability to "see" efficient computation everywhere in the world and in your research, identify suitable models, and reason about them.

To get a sense of what is an efficient model of computation, what we might want to know about it, and what kinds of methods are "legitimate" for answering these questions, let's start with one of the simplest examples: the decision tree. But before that we need to introduce and motivate some conventions for describing computational problems.

1 Computational problems

In complexity we model the computational problems we want to solve as functions. The function takes as its input a data item or a sequence of items as they may appear, say, in a computer's memory, hard drive, or "on the cloud". Its answer is the outcome of the computation (which we will assume always terminates) in a similar format. For example, the problem "What is the fastest way from Wan Chai to CUHK by bus?" might have maps and bus schedules as its input and your itinerary as its output.

Data items are usually described by bit sequences. Not only are bits the storage and processing unit of choice for most general-purpose computers, but even if this is not the case it is usually easy to convert whatever representation the data was stored in into bit representation. This is not where the real computational difficulty lies.[1]

We will represent computational problems as functions that map a sequence of input bits into one or more output bits. Even in the case when the inputs and outputs are objects of another type

[1]However, in some parts of computational complexity (e.g., arithmetic complexity) it is more natural to work with other representations.

we will think of them as being represented by sequences of bits. For example, if we study the problem of finding the prime factorization of a positive integer n, then n is given in binary, and its prime factorization is also described as a sequence of bits, one for each prime factor, with a suitable convention for representing the whole sequence as a single bit string.

A *computational problem* is a function whose domain is one set of bit sequences and whose range is another set of bit sequences. Should the domain and range be finite or infinite? This choice turns out to be surprisingly important. Let us try to motivate it by some examples.

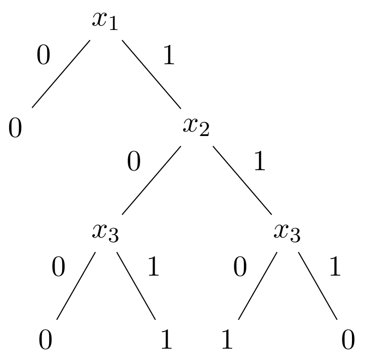
When we talk about algorithms for "finding a shortest path" or "factoring an integer" in the abstract, we would like our algorithm to work in principle for all inputs, so an infinite domain and range seem better suited for such study. You may object, however, that in practice we are unlikely to ever see any input an output that is more than say 2^{500} bits long, so shouldn't it be enough to restrict our domain and range to the sets $\{0,1\}^{2^{500}}$? While this is a reasonable objection, it is difficult to imagine anything interesting about these problems that happens only for inputs and outputs that are at most 2^{500} bits long; any algorithm designed for inputs that are 2^{500} bits long should in principle also work for arbitrarily long inputs. It is more natural to represent the domain and/or range of such problems as infinite sets, in which case the problem itself is a function from $\{0,1\}^*$ to $\{0,1\}^*$, the set of all possible bit sequences. The part of complexity theory that studies such problems is called (for reasons to be explained later) *uniform complexity.* The computational study of algorithms, and also proofs, is more natural in the setting of uniform complexity.

On the other hand, a cryptographic function like AES-256 is an algorithm that may only take inputs that are 256 bits long and always produces outputs that are 256 bits long. It is a specific design that is believed to have certain cryptographic properties and is simply not equipped to handle inputs or produce outputs of a different length. In (applied) cryptography, it is common to fix the size of the problem to be solved (e.g., "exchange a 1024-bit key") and then design an algorithm for a problem of this size. We model such problems as functions $f\colon \{0,1\}^n \to \{0,1\}^m$ for some *a priori* fixed lengths n and m. The part of complexity theory that studies them is called *non-uniform complexity* or *circuit complexity.* Apart from cryptography, non-uniform computational models are also more common in computational learning theory. For example, if we want to train a neural network to recognize shapes, we usually decide in advance on the size of the network and then run our learning algorithm of choice to calculate its relevant parameters.

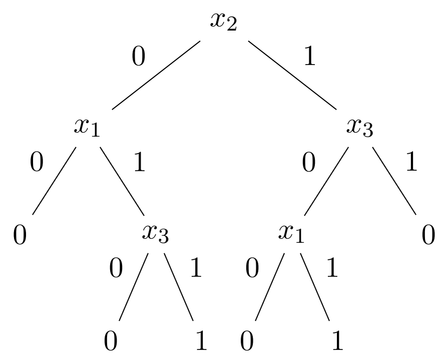
A *decision problem* is a computational problem with a yes/no answer, like "is graph G connected?". Such problems are represented by functions $f\colon \{0,1\}^* \to \{0,1\}$ and $f\colon \{0,1\}^n \to \{0,1\}$ in uniform and non-uniform complexity, respectively.

2 Decision trees

A *decision tree* for inputs of length n is a rooted binary tree whose vertices and edges are labeled as follows. Each of its internal nodes is labeled by one of the variables $x_1, \dots, x_n$, and its two outgoing edges are labeled by the values 0 and 1 respectively. Each leaf is labeled by a 0 or by a 1. Here is an example:

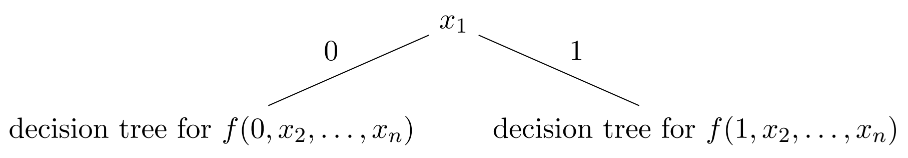


A decision tree computes a function $f\colon \{0,1\}^n \to \{0,1\}$ in a natural way: Query the variable at the root, follow the edge labeled by its value, and continue until a leaf is reached, then output its value. The above decision tree computes the function x_1 AND (x_2 XOR x_3). The following decision tree computes the same function:



The first decision tree is in some sense preferable than the second one as it is smaller. This is our first example of a *complexity measure*: The *size* of a decision tree is its number of leaves.[2]

It is not difficult to see that *any* function $f\colon \{0,1\}^n \to \{0,1\}$ can be computed by a sufficiently large decision tree like this:


x_1
0
1
decision tree for $f(0, x_2, \dots, x_n)$
decision tree for $f(1, x_2, \dots, x_n)$

The size of this decision tree is 2^n. Can we do much better?

The counting argument

We will now show that in general, the answer is "no":

Theorem 1. *For all n there exists a function $f\colon \{0,1\}^n \to \{0,1\}$ that requires decision tree size at least $2^n/4n$.*

[2]We could have also defined size as number of nodes, which is twice the number of leaves minus one. Working with leaves will be more convenient for us.

Proof. First, we show that the number of decision trees of size s with n variables is at most $(n \cdot (2s)^2)^{2s}$. A tree of size s has $2s-1$ nodes. To specify a decision tree it is sufficient to describe each node's label and the identities if its left and right children (if they exist). There are n possibilities for each label and $2s$ possibilities for the identity of each child (we reserve one possibility for "no child"), which gives a total of $(n \cdot (2s)^2)^{2s-1}$ choices. So the number of decision trees is at most $(n \cdot (2s)^2)^{2s-1}$.

On the other hand, the number of functions from $\{0,1\}^n$ to $\{0,1\}$ is 2^{2^n}. If 2^{2^n} were larger than $(n \cdot (2s)^2)^{2s-1}$, or equivalently if 2^n were larger than $(2s-1)\log(n \cdot (2s)^2)$, then there would be at least one function that is not computed by any decision tree of size s. You can verify that this is the case when $s = 2^n/4n$. □

This type of proof is called a counting argument. It has several appealing features. First, it applies to virtually any non-uniform model of computation that one can think of. Indeed, the proof uses almost nothing particular to decision trees. Second, it is quite insensitive to details: I could have refined the counting by distinguishing between internal nodes and leaves, but the end result would have been almost the same (try this at home). Third, there is a variation of the counting argument which not only tells us that there exists a "hard" function for decision trees, but also that a vast majority of such functions are hard:

Theorem 2. *For all $n \geq 2$, the probability that a random function $F\colon \{0,1\}^n \to \{0,1\}$ can be computed by a decision tree of size $2^n/8n$ is at most $2^{-2^{n-1}}$.*

Proof. To bound the probability that a random function is computable by some decision tree of size s, we apply a union bound:

$$\Pr_F[\text{some decision tree of size } s \text{ computes } F] \leq \sum_T \Pr[T \text{ computes } F]$$

where the summation ranges over all decision trees T of size s. The probability of any single such tree computing F is exactly 2^{-2^n}, as the value of the function computed by T must match the value of a random function at all points. From the proof of Theorem 1 the number of decision trees of size s is at most $(n \cdot (2s)^2)^{2s-1}$, so the expression on the right is at most $(n \cdot (2s)^2)^{2s-1} \cdot 2^{-2^n} = 2^{(2s-1)\log(n \cdot (2s)^2) - 2^n}$. When $s = 2^n/8n$, $(2s-1)\log(n \cdot (2s)^2)$ is at most 2^{n-1} by a very similar calculation as in the proof of Theorem 1 and we get the desired result. □

The main drawback of the counting argument is that it does not give us hold of an *explicit* function f that is hard for decision trees. Intuitively, "explicit" means that despite the existence of f with the desired property, we are at a loss when it comes to "writing down" a "nice expression" for it. The term "explicit" has a precise meaning, and we will define it properly once we become more comfortable with complexity-centric ways of thinking about computation.

An explicit hard function

The *parity function* on n bits is the function

$$PARITY(x_1, x_2, \ldots, x_n) = x_1 \text{ XOR } x_2 \text{ XOR } \cdots \text{ XOR } x_n.$$

It takes value zero when an even number of its inputs are ones, and value one otherwise.

Theorem 3. *$PARITY$ requires decision tree size 2^n.*

Proof. Suppose $PARITY$ on n bits can be computed by a decision tree T of size $s(n)$, $n \geq 1$. Then so can its complement function $\overline{PARITY}(x) = \text{NOT } PARITY(x)$: The decision tree for $\overline{PARITY}$ is obtained from T by flipping the value at each leaf.

Let x_i be the variable queried at the root node r of T. Since T computes parity, the two subtrees rooted by the children of r must compute the parity function on $n-1$ bits and its complement, respectively. It follows that $s(n) \geq 2 \cdot s(n-1)$. Since $s(0) = 1$, $s(n)$ must be at least 2^n. □

Theorem 3 is in some sense more satisfying than Theorem 1: Here is an explicit function, one that we can clearly write down and calculate at will, that cannot be computed by small decision trees. (The lower bound in Theorem 3 also better than the one in Theorem 1, but that is an unusual feature of this specific example.) What more could be possibly want? Well, one drawback of this argument is that it is specifically tailored to the $PARITY$ function. What if we want to know, say, the decision tree size of some other function, like $MAJORITY$ (assuming n is odd)?

$$MAJORITY(x_1, \dots, x_n) = \begin{cases} 1, & \text{if the input has more 1s than 0s,} \\ 0, & \text{if the input has more 1s than 1s.} \end{cases}$$

3 Random restrictions

A more general approach for proving lower bounds on decision tree size is to identify a relevant property that all small decision tree satisfy, but the "hard function" of interest does not satisfy. This is not an easy task, and much of the effort in computational complexity is devoted to identifying such properties for various models of computation.

One such property that is relevant for decision trees and some other sufficiently simple models of computation is "simplification under random restrictions." A *restriction* of the variables x_1 up to x_n is a partial assignment that gives each of the variables the value 0, the value 1, or leaves it unassigned. Such a restriction can be succinctly represented as a string ρ in $\{0, 1, \star\}^n$, where a $\star$ means that the corresponding variable is unassigned; for example, $01\star0\star$ is the restriction $x_1 = 0$, $x_2 = 1$, $x_4 = 0$, x_3 and x_5 unassigned.

Given a function $f\colon \{0,1\}^n \to \{0,1\}$ and a restriction $\rho \in \{0,1,\star\}^n$, the restricted function $f|_\rho(x)$ is obtained by substituting the variables assigned by ρ with the corresponding constants, for example

$$f|_{01\star0\star}(x_3, x_5) = f(0, 1, x_3, 0, x_5).$$

The input size of $f|_\rho$ equals the number of stars in ρ.

A δ*-random restriction* is a restriction that sets each coordinate independently to $\star$ with probability $\delta \in [0, 1]$ and 0 and 1 with probability $(1-\delta)/2$ each. Some very simple functions, like the AND of n bits, are typically "killed" by random restrictions when n is large: If any of the input is restricted to zero the AND function vanishes, so the probability that AND does *not* vanish is at most $(1/2 + \delta/2)^n$, which is exponentially small in n for say $\delta = 1/10$. By the same reasoning, the OR of n bits is also typically killed by a random restrictions, and so are ORs and ANDs of *literals* (possibly negated variables), such as

$$x_1 \text{ AND } (\text{NOT } x_2) \text{ AND } (\text{NOT } x_3) \text{ AND } x_4 \text{ AND } (\text{NOT } x_5) \cdots \text{ AND } x_n.$$

All these functions have decision trees of size n. Is it the case that random restrictions kill decision trees as well? Not always so, but they do simplify them greatly:

Theorem 4. *For every f computable by a decision tree of size s and a δ-random restriction ρ, the probability that $f|_\rho$ requires a decision tree of depth at least d is at most $s \cdot (1/2 + \delta/2)^d$.*

The *depth* of a decision tree is the maximum length of a path from root to leaf. This theorem reduces the problem of showing that f has no small decision tree to showing that the related function f_ρ has no *shallow* decision tree (with sufficient probability). But depth is a much easier complexity measure to analyze than size.

For example, it is immediate that the decision tree depth of $MAJORITY$ on n bits is at least $(n+1)/2$: any shorter root-to-leaf path can query at most half the inputs, which is insufficient to determine their majority value. In fact, the decision tree depth must be n: If the inputs "seen" by the decision tree are a 0, then a 1, then a 0, then a 1, and so on, then the majority cannot be determined until all n bits are read.

To apply this theorem, we need to analyze the decision tree depth not of $MAJORITY$ itself, but of its random restrictions. Before we do so let's prove Theorem 4.

Proof of Theorem 4. Let T be a decision tree of size s for f and p be a root-to-leaf path in T. We will assume, without loss of generality, that all variables that appear along p are distinct. We will say that p is killed by ρ if there exists a variable along p that ρ fixes to a value different from the one on its outgoing edge. For example, the root-to-leaf path

$$x_2 \xrightarrow{0} x_4 \xrightarrow{0} x_1 \xrightarrow{1} 0$$

is killed by the restriction $0{\star}00$ as the value of x_1 in ρ is inconsistent with the value on its outgoing edge. After substituting the non-starred variables in ρ by their values in T and removing all the paths that are killed, we obtain a decision tree for the function $f|_\rho$.

To conclude that the decision tree depth of $f|_\rho$ is at most d, it is therefore enough to show that all paths of length d or more are killed by the restriction. The probability that a given path p is *not* killed by ρ equals $(1/2+\delta/2)^{\text{length of } p}$, as for the path to survive each value along p must be either unfixed by ρ or fixed to the value along its outgoing edge. As there are at most s such paths, we can upper bound the probability of the complement event by a union bound:

$$\begin{aligned}\Pr[f|_\rho \text{ requires decision depth at least } d] &\le \Pr[\text{some length} \ge d \text{ path of } T \text{ is not killed by } \rho] \\ &\le \sum\nolimits_{\text{paths } p \text{ in } T \text{ of length } \ge d} \Pr[p \text{ is not killed by } \rho] \\ &\le s \cdot (1/2+\delta/2)^d. \end{aligned}$$ □

On the other hand, even after a random restriction, $MAJORITY$ is not too likely to have small decision tree depth:

Claim 5. *When $\delta = 1/2$, with probability $\Omega(1/n)$, $MAJORITY|_\rho$ on n inputs (for $n-1$ a multiple of 4) has decision tree depth at least $(n+1)/2$.*

Proof. Consider the event that exactly $(n-1)/4$ of the inputs are fixed to 0, exactly $(n-1)/4$ are fixed to 1, and the remaining $(n+1)/2$ are unfixed by ρ. If this is the case, $MAJORITY|_\rho$ is a majority of its unfixed $(n+1)/2$ inputs, and so it has decision tree depth at least $(n+1)/2$. The probability of ρ having exactly $(n+1)/2$ stars is $1/2^n \cdot \binom{n}{(n+1)/2}$, which is $\Omega(1/\sqrt{n})$. Conditioned on this, the probability of ρ having an equal number of zeros and ones is $1/2^{(n-1)/2} \cdot \binom{(n-1)/2}{(n-1)/4}$, which is also $\Omega(1/\sqrt{n})$. Multiplying the two we obtain the desired $\Omega(1/n)$ probability lower bound. (The $\Omega(1/n)$ can in fact be improved to $\Omega(1)$ by a slightly different analysis.) □

From Theorem 4 and Claim 5 it follows that if $MAJORITY$ on n bits has a decision tree of size s then $s \cdot (3/4)^{(n+1)/2} = \Omega(1/n)$, so $s = \Omega((4/3)^{n/2}/n) = \omega(1.154^n)$. Namely, computing majority on n bits requires decision trees of size exponential in n. (We did choose the best possible value of δ here; a more careful analysis should give a better exponent base.)

4 Disjunctive normal form

A formula in *disjunctive normal form* (DNF) of size s is an OR of s *clauses*, each of which is an AND of (distinct) literals. For example, distinctness of two n-bit strings $x, y \in \{0,1\}^n$ can be expressed as a DNF of size $2n$:

$$\begin{aligned} DISTINCT(x,y) = (x_1 \text{ AND NOT } y_1) \text{ OR } (\text{NOT } x_1 \text{ AND } y_1) \\ \text{OR } \cdots \text{ OR } (x_n \text{ AND NOT } y_n) \text{ OR } (\text{NOT } x_n \text{ AND } y_n). \end{aligned}$$

In terms of size, DNFs capture a larger class of efficient computations than decision trees:

Theorem 6. *If f has a decision tree of size s, then it has a DNF of size at most s.*

Proof. Let T be a decision tree for f of size s. For every path p of T that leads to a 1-leaf introduce a clause c_p that looks like this: Each variable x_i in p appears in c_p as literal x_i if its outgoing edge is a 1-edge and as literal NOT x_i if its outgoing edge is a 0-edge. The resulting DNF accepts exactly those inputs that lead to 1-leaves of T, so it computes the function f. $\square$

So DNFs are at least as strong a model of computation as decision trees. Are they stronger? To answer this question in the positive, we need to come up with a function that requires a large decision tree but admits a small DNF. It turns out that our usual suspects from the previous section — “most functions”, $PARITY$, and $MAJORITY$ — are not of much help here. I suggest that you try writing small DNFs for these functions to get some intuition why this is so. In the next lecture, we will in fact show that the PARITY and MAJORITY functions require cannot be represented by DNFs of size sub-exponential in the input length n.

Could it then be the case that DNFs of size s are *equivalent* to decision trees of size s, or maybe that they can be represented by decision trees of somewhat larger size, say s^2 or s^{10}? Not so: In the homework you will show that $DISTINCT$ (for input size $2n$) requires decision trees of size 2^n or more. This example shows that there can be an *exponential gap* in the size of the smallest DNF and the smallest decision tree for the same function. In the homework you will also show that the gap can never be more than exponential.

To conclude, today we saw two examples of non-uniform models of computation, namely decision trees and DNFs. We saw several ways to argue that certain functions that cannot be computed by small decision trees. We then stated a *separation* between these two models: DNFs of a given size are at least as powerful as decision trees of that size, but in the other direction there exist DNFs of size s that require decision tree size $2^{\Omega(s)}$ for infinitely many s.