Computational Models

Boaz Barak

September 20, 2005

Summary. A description of the formal computational models used. For the most part, the underlying model is not important, and you can take a T-time algorithm to mean a T time program in your favorite programming language and hardware platform. However, it is good to know that there is a way to precisely formalize such statements.

When trying to formalize things precisely, we find ourselves needing to use mathematical notations that may seem daunting at first sight. I assure you that there are no difficult proofs in this handout, and any difficulty reading it is either due to my writing or to cumbersome notations.[1] Therefore I suggest you just try to read this carefully and not be intimidated by any of this formal notations.

1 Turing machines

A *Turing machine* consists of several components:

Memory tape An infinite strip of memory cells. Each cell can hold either the value 0 or 1. (For convenience, we will also allow it to contain the two symbols $\triangleright$ and $\triangleleft$, which we'll call the *start* and *end* symbols, but of course everything can be encoded using 0 and 1 only.) We index the positions of the tape by whole numbers (that is the set $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$).

Read write/head The machine has a moving read write head. At every step the head can read or write a single cell of the tape, and can stay in place or move one position to the left or right. Initially the head is placed at position 0.

Internal register In addition to the memory tape, the machine has one internal register that can contain one of a small constant number of values (you can think of this as at most 100). Initially the register contains 1. The value of this register is called the *state* of the machine.

Control code/transition function The program of the machine consists of instructions of the following form:

`if reg=`r `and cell=`v `then` *cmd*

Where r is a number between 1 and 100 (or the number of possible states), $v \in \{0, 1, \triangleright, \triangleleft\}$, and *cmd* can one or a combination of the following:

- `write` v' for $v' \in \{0, 1, \triangleright, \triangleleft\}$

[1] I wrote this in a bit of a hurry and so resorted to the rather dirty trick of using you for proof readers at the price of 10 points in the exercise (see Exercise 7 of the second handout).

- `set reg=`r' for r' a possible state (e.g. $r' \in [100]$).
- `move left`
- `move right`
- `halt`

An *execution* of the machine on some initial setting of the memory tape is obtained by iterating its control code, each time executing the command, until the command `halt` is executed.

Note that in different textbooks you may see Turing machines described in slightly different ways; however all these variations are equivalent for our concerns. In particular, it won't make a difference if the memory tape is infinite in only one direction (that is, it has an edge on the left side and continues indefinitely on the right), or if the machine has several memory tape. Note that if we are guaranteed the machine will take at most T steps before it halts, then there is no difference between using an infinite tape or a length $2T$ tape.

Definition 1 (Computing a function by a TM)**.** Suppose that $f : \{0,1\}^n \to \{0,1\}^m$ is some function. We say that a Turing machine M *computes* $f(\cdot)$ *in* T *steps* if for every $x \in \{0,1\}^n$, if given a tape that in positions $0, \ldots, n+1$ contains the symbols $\triangleright, x_1, \ldots, x_n, \triangleleft$ and contains zeros everywhere else, if we execute the Turing machine then after at most T steps it will halt and the tape will contain in positions $0, \ldots, m+1$ the symbols $\triangleright, y_1, \ldots, y_m, \triangleleft$ for $y = f(x)$.

Note: We can easily generalize this definitions to functions with two or more inputs. That is, we say that a TM computes a function f of two inputs, if on input (x, y) it outputs $f(x, y)$. When we speak of the pair (x, y) as a string we mean that it is encoded in some standard way. For example, the encoding can be $x\#y$ where $\#$ is some sort of a separator symbol (everything can of course be encode using only 0 and 1).

Asymptotic analysis. Sometimes it is more convenient not to think of computing a single finite function $f : \{0,1\}^n \to \{0,1\}^m$ but of a *family* of functions (or equivalently, a function defined on $\{0,1\}^*$). Below we have some definitions for this case:

If $f : \{0,1\}^* \to \{0,1\}^*$ is a function then we denote by f_n the restriction of $f(\cdot)$ to $\{0,1\}^n$.

Definition 2. Let $T : \mathbb{N} \to \mathbb{N}$ be some function. The class $\mathbf{Time}(T)$ is the set of functions $f : \{0,1\}^* \to \{0,1\}^*$ defined as follows: $f(\cdot) \in \mathbf{Time}(T)$ if there exists a Turing machine M such that for every $n \in \mathbb{N}$, M computes the function f_n in $T(n)$ steps.

Note that some books and papers use the notation **Dtime** instead of **Time** (the "D" stands for deterministic, emphasizing the fact that the machine's behavior is completely determined by the initial contents of the memory tape, position, and internal register). Also, some sources define the class $\mathbf{Time}(T)$ to contain only *Boolean* functions (functions with a single bit output, sometimes also called *languages*).

Definition 3. The class $\mathbf{P}$ of polynomial-time computable functions is equal to the union of $\mathbf{Time}(p)$ for all polynomials $p : \mathbb{N} \to \mathbb{N}$. In other words:

$$\mathbf{P} = \bigcup_{c,d \in \mathbb{N}} \mathbf{Time}(dn^c)$$

2 Boolean circuits

Another common model of computation used is *Boolean circuits*. The easiest way to explain a Boolean circuit is by a picture (see one the web site), but for the sake of completeness, a formal definition is below:

Boolean circuits - formal definitions:

Definition 4 (Boolean circuit)**.** A Boolean circuit with n inputs and m outputs is a directed acyclic graph (DAG) with labels on the vertices. Each vertex is labeled in one of the following labels: $\{\mathsf{in}_1, \ldots, \mathsf{in}_n, \vee, \wedge, \neg, \mathsf{out}_1, \ldots, \mathsf{out}_m\}$. For every label of the type $\{\mathsf{in}_1, \ldots, \mathsf{in}_n, \mathsf{out}_1, \ldots, \mathsf{out}_m\}$ there is exactly one vertex with this label. The vertices labeled $\mathsf{in}_1, \ldots, \mathsf{in}_n$ must be sources (i.e., have in-degree $= 0$), the vertices labeled $\mathsf{out}_1, \ldots, \mathsf{out}_m$ must be sinks (i.e., have out-degree $= 0$). Vertices labeled $\wedge$ or $\vee$ must have in-degree $= 2$, while vertices labeled $\neg$ must have in-degree $= 1$.

The *size* of a Boolean circuit is the number of vertices it contains.

If C is a Boolean circuit with n inputs and m outputs, the *function* C computes is a function $f : \{0,1\}^n \to \{0,1\}^m$ defined in the following way:let $x \in \{0,1\}^n$ be some string. For every vertex v in C we define the *value of* v with respect to x to be: **(1)** x_i, if v is labeled in_i **(2)** $a \wedge b$, if v is labeled $\wedge$ and the values of the vertices u, u' with edges into v are a and b respectively. **(3)** $a \vee b$, if v is labeled $\vee$ and the values of the vertices u, u' with edges into v are a and b respectively. **(4)** $\neq a$, if v is labeled $\neg$ and the value of the vertex u with edge into v is a. The function f maps x into $y \in \{0,1\}^m$ where y_j is the value (w.r.t. x) of the vertex labeled out_j in the circuit.

We denote that output of a circuit C on input x by $C(x)$.

Definition 5. Let $T : \mathbb{N} \to \mathbb{N}$ be some function. The class $\mathbf{Size}(T)$ is the set of functions $f : \{0,1\}^* \to \{0,1\}^*$ defined as follows: $f(\cdot) \in \mathbf{Time}(T)$ if for every $n \in \mathbb{N}$, there exists a size $T(n)$ circuit C_n that computes f_n.

Definition 6. The class $\mathbf{P_{/poly}}$ of functions computable by polynomial-sized circuits is equal to the union of $\mathbf{Size}(p)$ for all polynomials $p : \mathbb{N} \to \mathbb{N}$. In other words:

$$\mathbf{P} = \bigcup_{c,d \in \mathbb{N}} \mathbf{Size}(dn^c)$$

3 The Relative Powers of Circuits and Turing machines

Circuits and Turing machines have, up to polynomial factors, the same computational power. That is, a circuit of size T can be simulated by a Turing machine with poly(T) description and poly(T) running time and vice versa. However, when thinking about infinite families of functions (or equivalently, functions defined on $\{0,1\}^*$) then the convention (as defined above) is that the family is computed by a Turing machine if there is a *single* Turing machine that computes all members of the family. In contrast, a single circuit only has finitely many inputs, and hence we say that the family is computable by $T(n)$ sized circuits if there is a different $T(n)$-sized circuit for every member f_n of the family.

Because of this convention, the class $\mathbf{P_{/poly}}$ is bigger than the class $\mathbf{P}$, and is typically called *non-uniform* computation (because for a function $f \in \mathbf{P_{/poly}}$ there is no single uniform machine

that computes all of the f_n's). The relations between the relative powers of circuits in TM's are described in the following theorem:

First, we note that *every function* can be computed by a sufficiently large circuit. That is, we have the following theorem

Theorem 1. *Let $f : \{0,1\}^n \to \{0,1\}^m$ be some function. Then, there exists a circuit C of size at most $100mn \cdot 2^n$ that computes $f(\cdot)$.*

Proof. We'll construct m different circuits each of size at most $100n \cdot 2^n$ for each of the outputs of $f(\cdot)$ and just concatenate them together (thus increasing the size by a factor of m). This means that without loss of generality we can think of $f(\cdot)$ as having a single output. Let S be the set of $s \in \{0,1\}^n$ such that this output is 1. Then,

$$f(x) = \bigvee_{s \in S} (x = s)$$

Consider the function $g_s : \{0,1\}^n \to \{0,1\}$ where $g_s(x) = 1$ iff $x = s$. This function can be implemented by a $4n$ sized circuit (it is equal to the AND of x_i for the i's where $s_i = 1$ and $\neq x_i$ for the i's where $s_i = 0$). Therefore, since $|S| \leq 2^n$, we get that a single bit output $f(\cdot)$ can be implemented by a circuit of size at most $4n \cdot 2^n$. □

we note that any Turing machine can be simulated by a circuit:

Theorem 2. *Let M be a Turing machine that on inputs of length n runs in time at most $T(n)$ (where $T(\cdot)$ is a polynomial-time computable function). Let $f_n : \{0,1\}^n \to \{0,1\}^*$ be the restriction of the function M computes to $\{0,1\}^n$. Then there exists a circuit C of size at most $T(n)^2 n$ that computes f_n. Furthermore, the transformation of M to C can be carried out in time polynomial in $T(n)$.*

Proof Sketch. This is proven in Sipser's book. Going through all the details is a bit tedious. However, it is essentially the following:

- If M runs for at most T steps then everything it accesses during the computation is within the $2T + 1$ positions closest to the origin (i.e., positions $\{-T, -T+1, \ldots, 0, 1, \ldots, +T\}$.
- For a fixed input $x \in \{0,1\}^n$ and $0 \leq t \leq T$, define $x^{(t)} \in \{0, 1, \triangleright, \triangleleft, \mathsf{ST}_1, \ldots, \mathsf{ST}_{100}\}^{2T+2}$ to be the contents of these positions in M's memory tape after the t^{th} computation step. We put a symbol of the form ST_s in the position just after the current position of M's read/write head, where s denotes the current state of the machine. Thus $x^{(0)}$ is equal to the concatenation of 0^T and $\triangleright \mathsf{ST}_1 x \triangleleft$ with additional padding with zeros. (If M has a different number of states than 100 then we we use a different alphabet.)
- Denote by $x_i^{(t)}$ the i^{th} symbol in $x^{(t)}$. The crucial observation is that $x_i^{(t)}$ is a function of only a constant number of entries in $x^{(t-1)}$ (specifically a function of $x_i^{(t-1)}$ and $x_{i+1}^{(t-1)}$. We can express this function using a constant number of logic gates.
- When looking at the whole thing we got a circuit of size about T^2 that computes $x^{(T)}$ from $x^{(0)}$. It's not hard to encode everything in the alphabet $\{0,1\}$ and convert this to a function that computes f_n

□

Theorem 2 immediately implies that $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$ and is also at the heart of the proof that the *circuit satisfiability* problem is **NP**-complete.

We also have a simulation in the other direction: from a circuit to Turing machine. However, we need to supply the Turing machine with the description of the circuit:

Theorem 3. *Define the following function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. Given input a description of an circuit C and a valid input x for C, $f(C,x) = C(x)$. Then f is computable in polynomial-time by a Turing machine.*

Proof. We presented the algorithm to do so above. □

Theorem 3 implies that we can present the class $\mathbf{P}_{/\mathbf{poly}}$ also in the terms of Turing machines. That is, we can say that $f : \{0,1\}^* \to \{0,1\}^*$ is in $\mathbf{P}_{/\mathbf{poly}}$ if there exist polynomials $T(\cdot)$ and $T'(\cdot)$, a Turing machine M and a sequence of strings $\{a_n\}_{n\in\mathbb{N}}$ where $|a_n| \leq T(n)$ such that for every $x \in \{0,1\}^n$, $M(x, a_n)$ outputs $f(x)$ within $T'(n)$ steps. These strings are often called the *advice* strings. Note that if there would be a different advice string for every $x \in \{0,1\}^*$ then it would be trivial to compute any function, since we could just give $f(x)$ as advice. However, things are different when you need to use a single string a_n for all $x \in \{0,1\}^n$.

4 Probabilistic Computation

We can generalize both the Turing machine and the circuit model to include the ability to toss coins. In the Turing machine we'll simply change the execution to be of the form

`if reg=`r `and cell=`v `and coin=`c `then` *cmd*

where whenever this step is executed c is equal to either 0 or 1 with probability 1/2 (independently of anything that happened before in the computation).

In the circuit model, we simply add an additional input to the circuit which is chosen at random. Thus, a probabilistic circuit C is just a standard circuit with two inputs: the first is the standard normal input and the second will be a string chosen at random. We say that a probabilistic Turing machine / circuit computes a function f if on input x it outputs $f(x)$ with probability at least 2/3 (note that this probability is only over the coins of the machine, and not over the choice of x). We define the class **BPP** to be the class of functions computable by probabilistic polynomial-time Turing machines.

It turns out that, as far as computing functions is considered, probabilistic circuits are not more powerful than standard (deterministic) circuits.

Theorem 4. *Every function $f : \{0,1\}^n \to \{0,1\}^m$ that can be computed by a T-sized probabilistic Boolean circuit C can also be computed by a $1000Tn^2$-sized standard (deterministic) Boolean circuit C'.*

Proof. Suppose that the probabilistic circuit C uses k (where $k \leq T$) random bits as input. For every sequence $\overline{r} = r_1, \ldots, r_{10n^2}$ (where $r_i \in \{0,1\}^k$), consider the circuit $C_{\overline{r}}$ which acts as follows: on input $x \in \{0,1\}^n$, $C_{\overline{r}}$ computes $C(x, r_1)$, $\ldots$, $C(x, r_{10n^2})$ and if there is one value that appears more than half in this list of outputs, it outputs this value. Otherwise it outputs 0^m. Note that $C_{\overline{r}}$ does not take $\overline{r}$ as input - it is simply "hardwired" as part of the description of the circuit. Also note that the size of $C_{\overline{r}}$ is at most $1000Tn^2$.

For a fixed string $x \in \{0,1\}^n$, consider now the following probabilistic experiment: choose $r_1, \ldots, r_{10n^2}$ independently at random from $\{0,1\}^k$. Denote by Y_i the following random variable: $Y_i = 1$ if $C(x, r_i) = f(x)$ and 0 otherwise. We know that $\mathbb{E}[Y_i] \geq 2/3$ and since all of them are independent, by the Chernoff bound the probability that $Y < 0.55(10n^2)$ (where $Y = \sum_{i=1}^{10n^2} Y_i$) is less than 2^{-n}.

Now consider an experiment where x is chosen at random from $\{0,1\}^n$ and $\overline{r}$ is chosen at random from $\{0,1\}^{k10n^2}$. Define the event B (for "bad") to hold if $C_{\overline{r}}(x)$ outputs a different value than $f(x)$. From the above, we see that $\Pr[B] < 2^{-n}$. Now we make the following claim:

Claim 4.1. *There exists a sequence $\overline{r}$ such that* for every $x \in \{0,1\}^n$, $C_{\overline{r}}(x) = f(x)$.

Before proving it note that it implies the theorem since we'll let $C' = C_{\overline{r}}$.

Proof of Claim 4.1. Let $\ell = k10n^2$ and suppose otherwise, that for every $\overline{r} \in \{0,1\}^\ell$ there exists $x \in \{0,1\}^n$ such that B holds (i.e., $C_{\overline{r}}(x) \neq f(x)$). This means that there exist at least 2^ℓ pairs $(\overline{r}, x)$ such that B holds which means that

$$\Pr_{x \leftarrow_R \{0,1\}^n, \overline{r} \leftarrow_R \{0,1\}^\ell}[B] \geq \frac{2^\ell}{2^\ell 2^n} = 2^{-n}$$

Thus obtaining a contradiction. □

□

5 The class NP

Relations. A *relation* R is a set of pairs of strings (i.e., $R \subseteq \{0,1\}^* \times \{0,1\}^*$). For $x \in \{0,1\}^*$, we denote by $R(x)$ the set of y such that $(x,y) \in R$. We can think of a relation as a generalization of a function with $R(x)$ allowing zero or more possible values instead of just one. We say that a relation is *polynomially bounded* if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $(x,y) \in R$, $|y| \leq p(|x|)$.

Verifying and solving a relation. We say that a polynomially bounded relation is *polynomial-time verifiable* if there's a polynomial-time M that on inputs x, y outputs 1 iff $(x,y) \in \{0,1\}$. We call such a relation an **NP** relation. We say that it is *polynomial-time solvable* if there is a polynomial-time M that on input x outputs y such that $y \in R(x)$ if such y exists and outputs $\perp$ otherwise ($\perp$ is some symbol indicating "undefined"). We call y a *witness* for x.

Associated decision problem. For a relation R, we define the associated decision problem to R, $L(R)$ to be the Boolean function that for each $x \in \{0,1\}^n$ returns 1 iff $R(x)$ is not empty. Alternatively, we can think of $L(R)$ as the set of x's such that $R(x)$ is not empty.

Definition 7. The class **NP** consists of all decision problems associated with **NP**-relations (i.e., with polynomial-time verifiable relations).

It is not hard to see that $\mathbf{P} \subseteq \mathbf{NP}$ (by **P** here I mean only Boolean (single bit output) functions). The biggest open problem of computer science (and one of the most important questions in science in general) is whether or not $\mathbf{P} = \mathbf{NP}$.

It is not immediate, but an equivalent formulation of the **NP** vs. **P** question is whether or not all relations that are polynomially verifiable are in fact polynomially solvable (can you prove this equivalence?).

A surprising fact is that $\mathbf{P} = \mathbf{NP}$ can be resolved by giving a polynomial-time algorithm for just one particular problem. This is the *circuit satisfiability* problem.

Theorem 5. *Consider the following Boolean function* CSAT. ***Input:*** *a Boolean circuit* C *with* n *inputs and one output.* ***Output:*** 1 *iff there exists* $x \in \{0,1\}^n$ *such that* $C(x) = 1$. *Then,* $\mathsf{CSAT} \in \mathbf{NP}$. *If* $\mathsf{CSAT} \in \mathbf{P}$ *then* $\mathbf{P} = \mathbf{NP}$.

The first part of the theorem ($\mathsf{CSAT} \in \mathbf{NP}$) is quite easy (and is given as an exercise). The second part ($\mathsf{CSAT} \in \mathbf{P} \Rightarrow \mathbf{P} = \mathbf{NP}$) is proven using Theorem 2 (can you see why?). A function $f(\cdot)$ with that property (that a polynomial-time algorithm for $f(\cdot)$ implies that $\mathbf{P} = \mathbf{NP}$) is called **NP**-*hard*. If furthermore $f(\cdot)$ is itself in **NP** then it is called **NP**-*complete*. It turns out that there are many more **NP** complete problems than just CSAT.