CS 710: Complexity Theory Fall 2017

Introduction

University of Wisconsin–Madison Dieter van Melkebeek

In this lecture we start with a high-level overview of the course, and then review the computational model of a Turing machine, the resources of time and space, and some of the standard complexity classes that are typically covered in an undergraduate theory of computing course.

1 Course Overview

This course provides a graduate-level introduction to computational complexity theory, the study of the power and limitations of efficient computation.

In the first part of the course we focus on the standard setting, in which one tries to realize a given relation between inputs and outputs in a time- and space-efficient way. We develop models of computation that represent the various capabilities of digital computing devices, including parallelism, randomness, and quantum effects. We also introduce models based on the notions of nondeterminism, alternation, and counting, which precisely capture the power needed to efficiently compute important types of relations. The meat of this part of the course consists of intricate connections between these models, as well as some separation results.

In the second part, depending on the interest of the students, we may study other computational processes that arise in diverse areas of computer science, each with their own relevant efficiency measures. Specific possible topics include:

- proof complexity, interactive proofs, and probabilistically checkable proofs – motivated by verification,
- pseudorandomness and zero-knowledge – motivated by cryptography and security,
- computational learning theory – motivated by artificial intelligence,
- communication complexity – motivated by distributed computing, and
- query complexity – motivated by databases.

All of these topics have grown into substantial research areas in their own right. We cover the main concepts and some of the key results.

2 Machine Model

The deterministic Turing machine is the model of computation we use to capture our intuitive notion of a computer. A Turing machine is depicted in Figure 1. The finite control has a finite number of states that it can be in at any time, and can read from and/or write to the various memory tapes. Based on the current state and the contents of the tapes, the finite control changes its state and alters the contents of the tapes.

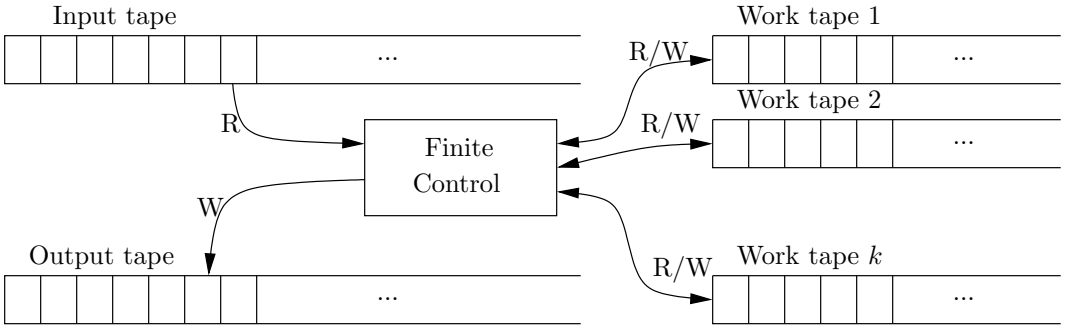

Figure 1: An illustration of a deterministic Turing machine. The finite control has read-only access to the input tape, write-only access to a one-way output tape, and read-write access to a constant k many work tapes.

Definition 1 (sequential access Turing machine). *A* sequential access deterministic Turing machine M *is defined by a tuple:* $M = (Q, q_{start}, H, \Sigma, \Gamma, \llcorner, \delta)$*, where* Q *is a finite set of possible states of the finite control,* $q_{start} \in Q$ *is the start state,* $H \subseteq Q$ *is the set of halting states,* Σ *is a finite input and output alphabet,* Γ *is a finite work-tape alphabet,* $\llcorner \in \Gamma \setminus \Sigma$ *is the blank symbol, and* δ *is the finite control's transition function.*

The transition function has the form

$$\delta : (Q \setminus H) \times (\Sigma \cup \{\llcorner\}) \times \Gamma^k \to Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^k \times \{L, R\} \times \{L, R\}^k.$$

The argument of δ *represents the current state, the current symbol being scanned on the input tape, and the current symbol being scanned on each work tape. The value of* δ *represents the next state of the finite control, a symbol to write to the output tape (possibly none, represented by the empty string* ϵ*), symbols to write to each of the work tapes, and which direction to move the head on the input and work tapes.*

We often use the binary alphabet $\{0, 1\}$ for Σ, and the binary alphabet plus the blank $\llcorner$ for Γ.

We think of the operation of a Turing machine on an input $x \in \Sigma^n$ as consisting of three steps. First, the machine is initialized as follows: For each $i \in [n]$, set the ith cell of the input tape to x_i, leave all other tape cells blank, set the head of each tape to position 0, and put the finite control in state q_{start}. Second, the machine is allowed to run one step at a time by repeatedly applying the transition function δ, where the output symbol produced (if any) is appended to the output tape. Third, if the machine ever halts by entering a state in H, the computation is finished and we read off the output from the output tape. We use $M(x)$ to denote the output of M on input x when the computation halts.

At first sight, the definition of a Turing machine may seem to be too restrictive to correspond to our intuitive notion of computing. However, the Turing machine has been shown to be just as powerful and roughly as efficient as traditional computers (we discuss this more later in this lecture).

The above sequential access model is sufficiently accurate for most of the results we develop in this course. However, sometimes it is important to enable indirect memory addressing. For that reason, our default model of a Turing machine is the following variation.

Definition 2 (random access Turing machine). *A* random access Turing machine *is a Turing machine that functions as in Definition 1 with regards to its output and sequential access work tapes. The input tape and any fixed number of work tapes may be* random access tapes *rather than sequential access tapes. Each random access tape has an associated sequential access* index work tape. *The tape head of a random access tape is moved by a special jump operation, whose only effect is to move the tape head to the location specified by the index tape and to change the state of the finite control.*

Note that each tape is either a sequential tape or a random access tape but not both. We think of the sequential access tapes as performing operations that are usually performed in registers on modern computers such as arithmetic, while the random access tapes are used for storage. We choose the random access Turing machine as our basic model of computation for this course as it more closely models modern computers than sequential access Turing machines (see Exercise 1 below for an example).

3 Time and Space

The Turing machine was originally defined and used in the theory of computability (also known as recursion theory). In this setting, the goal is to determine which mappings from inputs to outputs can be executed by a computer with unlimited resources. For example, a famous early result is that the Halting Problem is not computable by any Turing machine. It may seem surprising at first that there are uncomputable mappings. A closer inspection makes this fact obvious: there are only countably many Turing machines, while there are uncountably many mappings from inputs to outputs.

In contrast to the setting of computability, complexity theory is concerned with tasks that computers can execute *efficiently*. To this end, we consider the amount of resources a Turing machine uses during its computation. The two standard resources to consider are time and space.

Definition 3 (time and space). *Let M be a Turing machine and x an input to M. Then*

$t_M(x)$ is the number of steps until M halts on input x,
$t_M(n) = \max(t_M(x) | x \in \Sigma^n)$,
$s_M(x)$ is the sum over all work tapes of the index range of the cells visited until M halts,
$s_M(n) = \max(s_M(x) | x \in \Sigma^n)$.

$t_M(x)$ corresponds to the time used by M on input x, while $s_M(x)$ corresponds to the amount of memory used. $t_M(n)$ and $s_M(n)$ correspond to the worst performance of M on an input of length n; this choice reflects our focus on worst-case complexity in this course.

Exercise 1. Recall that a palindrome is a string that reads the same from left to right as from right to left. Construct a Turing machine M that outputs whether a given binary string is a palindrome and runs in time $t_M(n) = O(n \log n)$ and space $s_M(n) = O(\log n)$.

One can show that for any *sequential access* Turing machine M with the required input-output behavior, the product $t_M(n)s_M(n)$ is at least $\Omega(n^2)$ [1]. Thus, this exercise provides an example where the random access model captures reality more accurately than the sequential access model.

Note that in the definition of $s_M(x)$ we do not count the input tape or output tape memory cells that are used. This choice in definition is the reason we distinguish between the different types

of tapes in the first place, and allows us to consider Turing machines that achieve non-trivial tasks in sub-linear space. A definition including input tape usage in $s_M(x)$ would preclude non-trivial sub-linear space algorithms as then the entire input could not even be read. Usually, a Turing machine uses at least $\Omega(\log n)$ space as that is the amount of space required to index into the input. Similarly, a Turing machine usually runs for at least n steps as that amount is needed to read the entire input.

A possible alternative to the definition for $s_M(x)$ only counts the number of work-tape cells that are visisted during the computation. Both definitions coincide for the sequential access model but not for the random access model. Our definition counts all cells that are in between some work-tape cells that are visited, which may include unvisitedcells. Our definition is more natural from the perspective that if you want to run the algorithm on a computer, you'll need one with that much memory. If an algorithm uses 10K space with the alternative definition, it may not run on a machine with 10K of RAM, at least not without modification (see Exercise 2 below); it does with the our definition.

Using our definition of space the *configuration* of a machine (consisting of contents of the work tapes, the positions of all tape heads, and the state of the finite control) can be described using $O(s_M(x)+\log(|x|))$ space, while this is not true with the alternative definition. On the other hand, whereas with the alternate definition the space is always bounded by the time, this it not the case with our definition. In fact, $s_M(x)$ can be exponential in $t_M(x)$. However, using adequate data structures one can keep the space from being much larger than the time, without increasing the time by much.

Exercise 2. Show that for any Turing machine M there exists a Turing machine M' with the same input-output behavior such that for every input x, $t_{M'}(x) = O(t_M(x)\operatorname{polylog}(t_M(x)))$ and $s_{M'}(x) = O(t_M(x)\operatorname{polylog}(t_M(x)))$.

Like the distinction between the sequential access and the random access model, for most of our purposes the choice in definition of s_M does not affect the result statements.

4 Standard Setting

In the first part of the course we consider a computational problem as realizing a given relation between instances and solutions. We call this the *standard computational setting*. We typically think of instances and solutions as abstract objects, which we – sometimes implicitly – encode as strings over the alphabet Σ. This way, a computational problem is captured by a mathematical relation $R \subseteq \Sigma^* \times \Sigma^*$.

Definition 4 (computing a relation). *Given a* relation $R \subseteq \Sigma^* \times \Sigma^*$*, we say that a Turing machine* M computes R *if the following holds. For all instances* $x \in \Sigma^*$*,* M *on input* x *halts and outputs a* $y \in \Sigma^*$ *such that* $(x, y) \in R$ *if such a* y *exists, or halts and indicates there is no such* y *otherwise.*

Indicating whether there is a solution can be accomplished via a bit in the output, or by having multiple halting states and designating a subset $A \subseteq H$ as those indicating that there is a solution. The states in A is referred to as "accepting", and the other ones as "rejecting".

As an example, consider the shortest path problem. In this problem, we we wish to compute a shortest path between two given vertices in a given graph. Here, the input x is a description of the

graph and source and destination vertices; the output y is the description of a shortest path from the source to the destination in the graph, or an indication that there is no path from the source to the destination.

A function is a special case of a relation, where for a given x there is at most one y such that $(x, y) \in R$. Factoring is an example, where the input x represents an positive integer, and the output y represents the unique prime factorization, say listing the prime factors from smallest to largest.

A special case of a function is a Boolean function, where the codomain is $\{0, 1\}$. The underlying computational problem is referred to as a *decision problem*. It can be specified equivalently as the corresponding *language*, which consists of all instances that map to 1. For example, testing primality is a decision problem, with the set of primes as the corresponding language.

There is a natural way to associate any computational problem $R \subseteq \Sigma^* \times \Sigma^*$ to a language, namely the set of all instances $x \in \Sigma^*$ for which there exists $y \in \Sigma^*$ such that $(x, y) \in R$. The corresponding decision problem is called the *decision version* of R. It is often the case that the complexity of computing a relation is captured by the computational complexity of its decision version or of another closely related decision problem. For example, we can turn the factoring problem into a decision problem by trying to compute the i^{th} bit of the factorization. More precisely, on input (z, i, b), we try to decide whether the i^{th} bit of the prime factorization of z is b.

There is also a natural way to associate a Turing machine with a language.

Definition 5 (language accepted by a Turing machine). *Given a Turing mahcine M with accepting states A, the language accepted by M, denoted $L(M)$ is the set of inputs $x \in \Sigma^*$ on which M halts in a state in A.*

5 Goal in the Standard Setting

A natural goal in the standard computational setting is, given a relation R between instances and solutions, find the smallest t_M and s_M over all Turing machines M that compute R. In finding the "smallest," however, there are several issues to consider.

Comparability. What if between two Turing machines one is better on one input, and the other machine is better on a different input. In that case it is unclear which is the better machine to have.

Because of this, we instead look at the dual of the goal we originally stated: rather than trying to find the minimal $t_M(n)$ for a given relation, we instead try to determine which relations can be computed given a certain time and/or space bound.

Hardwiring. It is possible to "hard-wire" the solution to a finite number of input instances into the finite control of a Turing machine. This contributes to the comparability issue above. For any finite subset in our relation, there *is* a Turing machine that computes extremely quickly and with little space on that subset.

We resolve this issue by considering only the *asymptotic* behavior of the machines.

Speedups. We can "speedup" (reduce) memory usage by increasing the alphabet size: instead of $\{0, 1\}$ we can have $\{00, 01, 10, 11\}$ which would halve the number of cells in use. The space usage

can be decreased by any constant factor by increasing the size of the tape alphabet to a suitably large constant.

The number of computation steps needed can be reduced in a similar, but somewhat more complex, way. If we raise the alphabet size to the power $b \in \mathbb{N}$, we can perform all the operations that the original machine executed on a block of b consecutive cells in one step. If we additionally read the contents of the block on the left, and the block on the right, then we have enough information to carry out at least b steps of the original machine in the sequential access model. This means we can, asymptotically, reduce the running time by any constant factor.

Because of these constant factor speedups, we ignore constant factors in running time and space usage. This is formally realized through the "big-O" notation.

Robustness with respect to the model. Recall our discussion about the sequential vs random access model. We would rather have our results to be apply to both models. There are also several choices that we made in our models, e.g., that an index tape in a random access Turing machine gets erased after each use. We would like our results to be independent of those choices, as well.

Recall the *Church-Turing thesis* – that any relation computable on a physically realizable computing device can also be computed on a Turing machine. This belief underscores the use of the Turing machine in computability theory as the computing device to be studied. The Church-Turing thesis holds for all known models.

While sufficient for computability, the above Church-Turing thesis makes no mention of resources. The *Strong Church-Turing thesis* states that any relation computable on a physically realizable computing device can be computed by our model of a Turing machine with a polynomial overhead in time and a constant overhead in space. More precisely, if some machine uses $t(n)$ time to compute the relation, there is a Turing machine using $\text{poly}(t(n), n)$ time; and if there is a machine using $s(n)$ space, there is a Turing machine using $O(s(n) + \log n)$ space. Those overhead factors seem like the best possible. For example, one can show that any one-tape Turing machine (where a single tape is used as input tape, work tape, and output tape) deciding palindromes has to run for $\Omega(n^2)$ steps, whereas our model can do it in $O(n \log n)$ steps.

For that reason, robustness with respect to the model requires that time bounds are closed under polynomial overhead, and space bounds under constant-factor overhead.

Note that the Strong Church-Turing thesis is a much bolder statement than the Church-Turing thesis. For example, it is open whether it holds for randomized machines (believed to be the case) and for quantum machines (believed not to be the case as factoring can be computed in polynomial time on a quantum machine but is conjectured not be be computable in polynomial time on deterministic or randomized machines).

Robustness with respect to the encoding. Consider the shortest path problem. The input to the problem includes a graph, which must be represented somehow as a string. Standard methods of representing a graph include an adjacency matrix and an adjacency list. Note that the number of bits of the former representation may be quadratic in the latter. This affects the running time and space usage as these are functions of the size of the input. An instance can also be made artificially large (say, by *padding* with 0's), in which case the running time and space usage are artificially decreased as functions of the size of the input.

We always assume a "reasonable" encoding of the instances and solutions, in which case the time and space usage do not depend too much on the particular encoding. Typically, natural

encodings can be computed from one another in polynomial time or more efficiently.

6 Complexity Classes

Given the comparability, hardwiring, and speedup issues mentioned in the previous section, we consider the following general definition of time-bounded and space-bounded complexity classes.

Definition 6. *For a given $t : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, we define:* $\mathrm{DTIME}(t(n))$ *as the class of languages accepted in $O(t(n))$ time by a random access Turing machine, and* $\mathrm{DSPACE}(s(n))$ *as the class of languages accepted in $O(s(n))$ space by some random access Turing machine.*

Given the robustness issues, the following are the standard (deterministic) complexity classes.

Definition 7 (standard deterministic complexity classes).

$$\begin{aligned} \mathrm{P} &= \cup_{c>0}\mathrm{DTIME}(n^c), \\ \mathrm{EXP} &= \cup_{c>0}\mathrm{DTIME}(2^{n^c}), \\ \mathrm{E} &= \cup_{c>0}\mathrm{DTIME}(2^{c\cdot n}), \\ \mathrm{L} &= \mathrm{DSPACE}(\log n), \\ \mathrm{PSPACE} &= \cup_{c>0}\mathrm{DSPACE}(n^c). \end{aligned}$$

Note that all classes are robust with respect to the choice of model, and most are also robust with respect to the encoding. The only exception is E.

Complexity theorists are quite interested in relationships between these standard complexity classes. We will later argue the following relationships:

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}.$$

The relationship $\mathrm{P} \subseteq \mathrm{PSPACE}$ follows from Exercise 2. The other inclusions require some thought. It remains open whether any of these containments is proper.

References

[1] Alan Cobham. The recognition problem for the set of perfect squares. In *Proceedings of the 7th Annual Symposium on Switching and Automata Theory*, pages 78–87, 1966.