CS 710: Complexity Theory 1/19/2010

Lecture 1: Introduction

Instructor: Dieter van Melkebeek Scribe: Jeff Kinne

Today we begin a review of basic Complexity Theory material that is typically covered in an undergraduate theory of computing course. In this first lecture, we introduce the course, define the model of computation we use to formalize our intuitive notion of a computer, and explore issues that arise concerning the computational model.

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 mapping of inputs to 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, quantum effects, and non-uniformity. 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 mappings. The meat of this part of the course consists of intricate relationships between these models, as well as some separation results.

In the second part we study other computational processes that arise in diverse areas of computer science, each with their own relevant efficiency measures. Specific 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,
- query complexity – motivated by databases.

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

2 The Standard Setting

2.1 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 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.

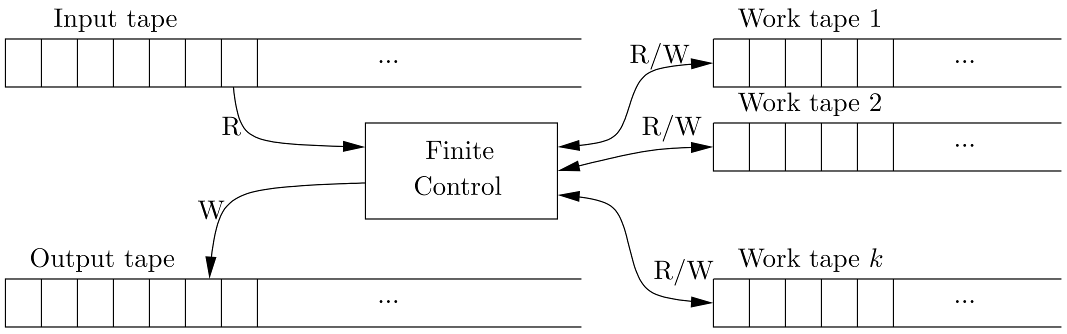

Figure 1: An illustration of a deterministic Turing machine. The finite control has read-only access to the input tape, write-only access to a one-way output tape, and read-write access to a constant k many work tapes.

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

The transition function has the form

$$\delta : Q \setminus \{q_{halt}\} \times \Sigma \times \Gamma^k \to Q \times \{\Sigma \cup \{\epsilon\}\} \times \Gamma^k \times \{L, R\} \times \{L, R\}^k.$$

The input to 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 output represents the next state of the finite control, a symbol to write to the output tape (possibly empty), symbols to write to each work tape, and which direction to move the head on the input and work tapes.

Both Σ and Γ contain a blank symbol, which is used to denote empty tape cells. We often use the binary alphabet, which in addition consists of the symbols 0 and 1.

We think of the use of a Turing machine as consisting of three steps. First, the machine is initialized as follows: place input on input tape with tape head on first symbol of input, work tapes are empty with their tape heads on the left-most cell of the tape, the output tape is empty, and the finite control is set to q_{start}. Second, the machine is allowed to run one step at a time by repeatedly applying the transition function δ. Third, if the machine ever halts by entering q_{halt}, the computation is finished and we can read the output from the output tape. We use $M(x)$ to denote the output of M on input x when this 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 model of Turing machine we use in this course is a modification of the definition given above. The definition above is a bit too restrictive as it does not allow indirect memory addressing, which real computers rely on.

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 that moves its tape head to the location specified by its index tape and re-initializes the index tape. The tape head position of a random access tape is not altered by the transition function, but the contents of the memory cell are read and written by the transition function.*

Notice 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.

2.2 Computing a Relation

The main setting in the first part of the course is the Turing machine's ability to compute various relations.

Definition 3. *Given a* relation $R \subset \Sigma^* \times \Sigma^*$*, we say that a Turing machine M* computes R *if the following holds. For all inputs $x \in \Sigma^*$, M on input x halts and outputs a $y \in \Sigma^*$ such that $(x, y) \in R$ if such a y exists and indicates "no" if no such y exists.*

Indicating no can be accomplished by some encoding scheme on the output or by having two different halt states.

As an example relation, consider the shortest path problem. For this relation, we wish to compute a shortest path between two vertices in a graph. Here, the input is a description of the graph and source and destination vertices; the output is the description of a path through the graph. Notice that 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 is an integer n, and the output is the unique prime factorization of n listing the prime factors from smallest to largest.

We are often interested in a particular kind of relation, called a decision problem.

Definition 4 (decision problem)**.** *A* decision problem *is a relation where the output is always either 1 or 0. In this case, an output of 1 indicates that the input has some property, and 0 indicates the input does not have the property. We refer to the set of inputs with corresponding output 1 as a* language*.*

As an example, consider the problem of determining if a string is a palindrome. We define the language PALINDROMES as the set of strings that read the same front to back as they do from back to front. The corresponding relation assigns 1 to each palindrome and 0 to each non-palindrome.

It is often the case that the complexity of computing a relation is captured by the complexity of computing a 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 output. That is, on input $\langle n, i\rangle$, we try to compute the i^{th} bit of the description of the prime factorization of n.

3 Time and Space Complexity

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 relations are computable. 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 relations. A closer inspection makes this fact obvious: there are only countably many Turing machines, while there are uncountably many different relations.

In contrast to the setting of computability, complexity theory is concerned with the efficiency with which a relation is computable. 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 5. *Let M be a Turing machine and x an input to M. Then*

$$\begin{aligned}
&t_M(x) = \textit{the number of steps until } M \textit{ halts on input } x,\\
&t_M(n) = \max(t_M(x) | x \in \Sigma^n),\\
&s_M(x) = \textit{the sum over all work tapes of the maximum cell touched until } M \textit{ halts},\\
&s_M(n) = \max(s_M(x) | x \in \Sigma^n).
\end{aligned}$$

$t_M(x)$ corresponds to the time used by M on input x, while $s_M(x)$ corresponds to the amount of memory used. A possible alternative to the definition we have given for $s_M(x)$ would be to only count the number of work-tape cells that are touched. Our definition counts all cells that are left of some work-tape cell that is touched, and hence counts even unused cells that are left of some used cell. 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; it does with the our definition. Also notice that the configuration of a machine (the positions of its tapes, contents of its work tapes, and its internal state) can be described using $O(s_M + \log(|x|))$ space, while this is not true with the alternative definition. For most purposes, the choice in definition does not effect the power or efficiency of the model.

We are often interested in the worst-case complexity of a Turing machine, and hence have defined the worst-case measures $t_M(n)$ and $s_M(n)$. These correspond to the worst performance of M on any input of length n.

Notice in the definition of $s_M(x)$ that we do not count the input tape or output tape memory cells that are used. This choice in definition is the reason we distinguished between the different types of tapes in the first place, and allows us to consider Turing machines that compute non-trivial relations using 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. We will consider algorithms that use at least logarithmic space as this is the amount required to index into the input. For time usage, linear is the smallest we consider as we must at least read the entire input.

3.1 The Goal of Complexity Theory

The main goal of complexity theory is to characterize the amount of time and space required to compute a given relation. In other words, find a machine M so that either $t_M(n)$ or $s_M(n)$ is minimal over all machines that solve the relation. However, a number of issues arise when asking this question.

- *Hard wiring.* The solution to any finite subset of the possible inputs can be hard-wired into the transition function of a Turing machine. By using a lookup table, any finite subset of possible inputs can be solved very fast with low space usage.

  Because of this issue, we focus on asymptotic run-time and space usage.

- *Constant factor speedups.* Consider a Turing machine that uses a certain amount of space using the binary alphabet. By changing to the alphabet $\{0,1\}^{10}$, the machine can store the information of 10 of its original tape cells into 1 of the new tape cells. The finite control of the machine can be modified to work properly in the new setting, thus reducing the space usage of the machine roughly by a factor of 10. The space usage can be decreased by any constant factor by increasing the size of the tape alphabet to a suitably large constant. Notice that if we had used the alternative definition of space usage that only counts memory cells that are touched during the computation, then this argument would only work for random access machines for algorithms that have a high amount of memory locality.

  A similar argument can be made for time usage. The argument is a bit more complicated because the finite control must read both the current cell and immediately adjacent cells to properly mimic the original Turing machine. It may seem that random access operations would destroy any possible time speedup. An argument to get around this potential problem makes use of the fact that indexing operations will still be sped up and an algorithm making many random access operations will also spend a large amount of time indexing those operations.

  Because of these constant factor speedups, we ignore constant factors in running time and space usage.

- *Incomparable behavior.* Even modulo the above issues, a fundamental problem remains in attempting to determine the optimal time and space usage to compute a relation. It may be the case that there are two machines M_1 and M_2 so that on some input lengths t_{M_1} is smaller while on other input lengths t_{M_2} is smaller. In this case, a third machine can be created to simulate both machines and thus achieve near optimal run-time over all inputs. However, because of the hard-wiring issue already discussed, given a finite set of inputs, there is always a Turing Machine that decides these instances correctly very fast and with small space usage. Each of these machines is nearly optimal on some different set of inputs, and it is not in general possible to create a machine combining the optimal performance from each of these infinitely many machines. Thus, in general, there is no single single Turing machine with minimal run-time or space-usage on all input lengths.

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

Definition 6. *For a given $t : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, we define:*
DTIME$(t(n))$ = { *Decision problems solvable in $O(t(n))$ time by some random access Turing machine* },
DSPACE$(s(n))$ = { *Decision problems solvable in $O(s(n))$ space by some random access Turing machine* }.

The main goal of complexity theory can then be restated as follows: given a time or space bound $t(n)$, which relations can be computed on a Turing machine in this amount of time or space? Given this new goal, a number of issues still remain.

- *Model dependence.* We would like our results to be independent of the particular choices we have made in defining the Turing machine. 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 that is studied. Note that the Church-Turing thesis is not violated by any of the known models of computation.

In the setting of complexity theory, we consider the *Strong Church-Turing thesis* - 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. Namely, if some machine uses $t(n)$ time to compute the relation, there is a Turing machine using $\text{poly}(n, t(n))$ time; and if there is a machine using $s(n)$ space, there is a Turing machine using $O(\log n + s(n))$ space. Notice that the Strong Church-Turing thesis is a much bolder statement than the Church-Turing thesis. In particular, it is widely believed to be violated by quantum machines (although it is debatable if these are physically realizable), and may even be violated by randomized machines (although this is believed not to be the case).

Consider the language PALINDROMES and a single tape sequential Turing machine. It can be shown that the trivial algorithm of scanning back and forth across the input string is the best possible, with a running time of $\Theta(n^2)$. A similar result shows that on a multi-tape sequential access Turing machine, the product of the space and time usage to solve palindromes is $\Omega(n^2)$. However, PALINDROMES can be decided in simultaneous quasi-linear time and logarithmic space on random access machines.

Notice that this result implies that a single-tape Turing machine must incur roughly a quadratic overhead in time if it attempts to simulate our model of a random access multi-tape Turing machine. That is, different physically realizable models of computation often differ by a polynomial factor in the time to solve a relation and in a constant factor in the space needed. Hence the Strong Church-Turing thesis stated above certainly can not be strengthened.

Also notice that the quadratic lower bound for PALINDROMES on sequential access Turing machines is model dependent: it is true on certain models but not on others. We have chosen the model of the random access Turing machine to avoid such model dependent results. Each of the results we present in this course will be model independent.

- *Input representation.* The input representation is clear for the PALINDROMES language. However, for many relations there is some choice that can be made. Consider the shortest path problem. The input to the problem includes a graph, which must be represented somehow on the input tape. Standard methods of representing a graph include an adjacency matrix and an adjacency list. Note there may be a linear factor difference in the size of these. This effects the running time and space usage because these are functions of the size of the input. If an input is made to be artificially large (say, by padding with unnecessary 0's) then the running time and space usage are artificially decreased as functions of the size of the input.

  For this reason, we always assume a "reasonable" encoding of the input. We leave this notion qualitative, although it can be quantified. As long as we choose a reasonable encoding of the input, the running time and space usage do not depend too much on the input encoding.

Because of the above issues, we define complexity classes that are robust with respect to both the model and the input representation - namely the complexity class remains the same regardless of the particular model or reasonable input encoding used.

Definition 7. *The following are definitions of standard complexity classes.*

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

Notice that each of these is robust with respect to the model under the strong Church-Turing thesis. Each of the classes other than E also is robust with respect to input representation.

It may seem that restricting the space usage to logarithmic in the case of the complexity class L is too great. For graph problems, logarithmic space is only enough to remember a constant number of vertices within the graph. However, it turns out that many interesting problems can be solved within L. For example, a recent result showed that the problem of determining connectivity in an undirected graph can be achieved using logarithmic space on deterministic machines.

We will later show the following relationships between complexity classes:

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

It is open whether any of these containments is proper.