Computational complexity theory - an overview

Uri Feige and Ran Raz

October 29, 2003

1 Introduction

The course will present selected topics in Computational Complexity. It is assumed that students already have some background on this subject, and are familiar with concepts such as *Turing machine* and *NP-completeness*. Nevertheless, the first three lectures (given by Uri) and the associated homework assignments will be devoted to a review of the "basics" in computational complexity theory (time hierarchy, space hierarchy, relations between time and space, complete problems, nondeterminism), and are recommended also for students who do not wish to take the whole course. Based on these basics, students should be able to discuss intelligently the following question: how come computers easily outperform humans in some well defined computational tasks (such as multiplying large numbers), but still do not quite do so for other well defined computational tasks (finding the best move in a chess position). In the (roughly 8) lectures that follow Ran will discuss randomization and derandomization, and circuit complexity. The last set of lectures will be given by Uri, and will probably deal with interplay of nondeterminism and randomness (as manifested in *interactive proofs*). Many important and interesting results in computational complexity are left out of the course due to lack of time. If there are topics that you would like to see covered in the course, please let us know.

A related course is "Coping with NP-hardness", given by David Peleg this semester.

2 Models of computation

To be able to argue formally about computations, we need to model computation. Most "reasonable" models of computation are equivalent for our purposes. (*Quantum computing* may turn out to be an exception to this.) Thus for specifying algorithms, we will use a "high level" description, disregarding implementation details that vary between models of computation. In cases that we want to argue formally about how computation proceeds, we will use the model of the one-tape Turing machine.

The Turing machine has a *finite control*, and one two-way infinite *work tape* accessed by a read/write (R/W) head. Computation proceeds in time steps. Relevant parameters are:

Q - the finite set of states of the finite control.

$q_I \in Q$ - the initial state at which computation begins.

$q_f \in Q$ - the final state.

($q_a \in Q$ - an accepting final state.)

($q_r \in Q$ - a rejecting final state.)

Σ - the finite alphabet. $\bar{\mathrm{b}}$ denotes the blank character.

$\delta : (Q, \Sigma) \longrightarrow (Q, \Sigma, \{\ell, s, r\})$ - the transition function.

Initially, the input string x occupies n consecutive cells on the tape, starting at the location of the R/W head, and is delimited by $\bar{\mathrm{b}}$ on both sides. The output of the computation is the contents of the output tape when TM reaches q_f.

Instantaneous description (ID): The configuration of a Turing machine at any time step is fully described by the state of the finite control, the contents of the tape, and the location of the R/W head. A convenient representation augments the tape alphabet to $(\Sigma, Q \bigcup 0)$, and specifies on the tape (at the location of the R/W head) the state of the finite control (and a default value such as 0 elsewhere).

Tableau: The computation of a Turing machine is a dynamic process. However, it can be represented as a static object, called a tableau, which is simply the sequence of IDs that the computation undergoes, written one below the other. Simple local checks may tell us whether a given tableau indeed corresponds to a valid computation of the Turing machine. The contents of the first row as determined by the input, and the content of a cell in any other row is uniquely determined by the contents of three cells in the row above it.

Computational resources: The time of computation of a Turing machine on input x is the number of time steps that elapse until it reaches q_f. The space is the number of cells accessed on the tape. These resources are often described as a function of n, the input size. E.g., the statement that $time_M(x) \leq n^2$ means that on any input x, the running time of the Turing machine M is at most the square of the input length.

Exceptions to the use of one tape Turing machine. In certain cases it will be more advantageous to consider other models of computation. Two-tape Turing machine appear more relevant when tighter bounds on running time are of interest. A two tape TM can simulate a multitape TM with only logarithmic slowdown. A one tape TM requires quadratic slowdown when simulating a two tape TM. (It is not hard to prove quadratic lower bounds on the running time of 1-tape Turing machines, even for some simple computational problems.) In fact, we will often describe algorithms as running on multi-tape TMs (or even on *random access machines*), as this is more convenient. When dealing with sublinear space, it is useful to consider Turing machines which have a separate input tape and a separate work tape. We shall also be considering some so-called *nonuniform* models of computation, such as circuit families.

Universal Turing Machines. A Turing machine M can be encoded as a string of characters (bits). Then it can serve as input to another Turing machine U, which can simulate M's operation, based on M's description. In such a case, U is called a universal Turing Machine.

3 Languages, decision problems, and complexity classes

A language L is a set of strings over Σ^* (assume that $\Sigma = \{0, 1\}$). For example, the set of binary representations of all prime numbers is a language. We will mostly be interested in Turing machines that recognize languages, or solve the decision problem "is $x \in L$". (They can also be viewed as computing the characteristic function χ_L.)

A language is *recognized in time* $t(n)$ if there exists a Turing machine M such that for every x, $M(x) = \chi_L(x)$, and $time_M(x) \leq t(n)$.

Figuring exact time (or space) bounds for computing various functions is a very difficult task. Moreover, the bound may depend on the exact model of computation, and on the encoding scheme for the input. However, "rough" bounds on the complexity are often robust, and hold for most reasonable computational models and reasonable encoding schemes. We will use $O(.)$ and $\Omega(.)$ notation to abstract away constant factors. Moreover, in defining complexity classes, we often abstract away polynomial factors (of the form n^c, for some constant c). (Additional motivation for this abstraction can be found in the notion of *Turing reduction*, defined below.)

Some complexity classes:

P - *polynomial time*, languages that can be recognized in time $O(n^c)$. It is customary in computational complexity theory to identify *efficient* computation with polynomial time computation (disregarding the actual value of c).

PSPACE - *polynomial space*, languages that can be recognized in space $O(n^c)$.

EXPTIME - *exponential time*, languages that can be recognized in time $2^{O(n^c)}$.

LOGSPACE - *logarithmic space*, languages that can be recognized in space $O(\log n)$. Here only space on the work tape is considered, and not on the input tape.

$$LOGSPACE \subset P \subset PSPACE \subset EXPTIME$$

It is conjectured (but still remains a major open question) that these containment relations are strict (i.e., that there is a language $L \in PSPACE$ but $L \notin P$). Strict containment has been proven only for the "time hierarchy", and the "space hierarchy".

4 Nondeterministic computation.

It is an everyday phenomenon that there are problems that are hard to solve, but once someone shows you the solution, it is easy to follow/verify the solution. Is there an efficient systemic way of solving problems whose solution is easy to verify?

We model a proof system as a game between two players, a computationally limited verifier V, and a computationally unbounded prover P. Each player is modelled as a Turing machine, and both machines have access to an additional *communication tape*. Both see a common input x. The verifier uses a prespecified protocol to exchange messages with the prover over the communication tape. In the end, the verifier passes his verdict - *accept* or *reject*.

NP proof systems: The verifier is polynomial time. A language L is in NP if there exists a polynomial time verifier V and a protocol (P, V) such that:

1. *Completeness:* It is possible to prove true statements. For any $x \in L$, $V_P(x)$ accepts.
2. *Soundness:* It is impossible to prove false statements. For any $x \notin L$, for any strategy of a possibly cheating prover P', $V_{P'}(x)$ rejects.

The traditional definition of NP involves a *nondeterministic Turing machine* (NDTM) that makes nondeterministic transitions (δ is a relation instead of a function). Input x is *accepted* by the NDTM if there exists at least one accepting computation for x. $L \in NP$ iff there exists a polynomial time NDTM that accepts exactly those inputs that are in L. The notion of NDTMs makes it easier to argue formally about nondeterministic complexity classes, but is not meant to represent a realistic model of computation. In tableau representation, it is no longer the case that a content of a cell is uniquely determined by the three cells above it. Rather, it has to be consistent with the content five cells (one to its right, one to its left, and the three above it).

The *witness* to an NP statement is the sequence of nondeterministic transitions that lead to acceptance, or the convincing sequence of bits sent by the prover in a proof system.

$$P \subset NP \subset PSPACE$$

The class coNP is the complement of NP ($L \in$ coNP iff $\bar{L} \in$ NP). $x \in L$ if *every* computation of the NDTM on x is accepting. It is not known whether NP is closed under complement.

We can define similarly also nondeterministic space bounded complexity classes. Among them the most important is NL (nondeterministic LOGSPACE). Much more is known about nondeterministic space complexity classes than about nondeterministic time complexity classes. We shall see that PSPACE = NPSPACE (Savitch's theorem) and the $NL = coNL$ (the Immerman-Szelepcsenyi theorem).

5 Reductions and complete problems

A useful way of studying a complexity class is via a particular computational problem that serves as a representative for the whole class. Such a problem must be in the class, and must (in an intuitive sense) be the most difficult problem in the class, so as to represent the true complexity in the class. A way of establishing a problem as being the most difficult in a class is via reductions among problems. There are several types of reductions.

Combinatorial reductions: L_1 is many-to-one reducible to L_2 if there exists a polynomial time (or log-space) procedure p such that for any x, $x \in L_1$ if and only if $p(x) \in L_2$.

Turing reductions: L_1 is Turing-reducible to L_2 if any algorithm for recognizing L_2 can be used as a blackbox in a polynomial time algorithm for recognizing L_1 (each call to the blackbox costs one time step).

A language L is *NP-hard* if any language in NP is reducible to L. An NP-hard language that is in NP is *NP-complete*.

Locality of computation is central to proving NP-completeness results.

Satisfiability and 3SAT are NP-complete Languages. $3SAT \in P$ iff $P = NP$.

s-t-connectivity in directed graphs is NL-complete.

Sometimes, the complete problem is what makes a complexity class interesting.

It is important to note that saying the 3SAT is NP-complete is a stronger statement then saying that "$3SAT \notin P$ unless $P = NP$". As reductions are algorithmic, 3SAT being NP-complete also means that positive results about 3SAT (e.g., having a so called *zero knowledge interactive proof*, having an efficient *quantum algorithm*, if one is found) almost automatically transform to the whole of NP.

6 Probabilistic complexity classes

Important classes in this respect are BPP, RP, RL. In addition, there are complexity classes that combine randomness and nondeterminism, such as AM, IP, MIP. These classes will be discussed in later lectures.

7 Central results

Here is a list of basic results in computational complexity theory. They are stated informally. A formal statement would involve also the notion of *time-constructibility* and *space-constructibility*.

7.1 Simulation results

The following can be proved using relatively simple algorithms.

$NTIME(f(n)) \subset SPACE(f(n))$

$NSPACE(f(n)) \subset TIME(2^{f(n)})$

7.2 Diagonalization

The bounds in the following theorems can be improved.

$TIME(f(n)) \neq TIME(f(n)^2)$

$SPACE(f(n)) \neq SPACE(f(n) \log f(n))$

7.3 Nondeterministic space

Nondeterminism offers only limited savings in terms of space.

$NSPACE(f(n)) \subset SPACE(f(n)^2)$

$NSPACE(f(n)) = coNSPACE(f(n))$

We get the following list of containments:

$LOGSPACE \subset NL = coNL \subset P \subset NP \subset PSPACE = NPSPACE$

7.4 Complete problems

A surprising phenomenon is that almost all computational problems not known to be in P are known to be NP-hard.

3SAT is NP-complete.

STCON is NL-complete.

Problems involving alternating polynomial time (such as the game of *generalized geography*) are complete for PSPACE.

Problems involving alternating polynomial space (such as *generalized chess*) are complete for EXPTIME. The EXPTIME-complete problems are currently the most interesting class of computational problems for which there is proof that they have no (uniform) polynomial time algorithm.

8 Open questions

It is important to be aware that many questions remain open. The most well known is:

P=NP? (Does 3SAT have a polynomial time algorithm?)

A positive answer would imply also a positive answer to the following open question.

NP=coNP? (Is there a short witness that a 3SAT formula is not satisfiable?)

Either a positive answer to P=NP or a negative answer to NP=coNP would imply a negative answer to the following open question.

NP=EXPTIME?

A negative answer to P=NP implies also negative answers to the following open questions.

P=PSPACE?

LOGSPACE=NP?

The main open question involving space and nondeterminism is the following.

LOGSPACE=NL? (Can reachability in directed graphs be solved in logarithmic space?)

References

[1] S. Cook, “The complexity of theorem proving procedures”, 3^{rd} *STOC, 1971, 151-158.*

[2] M. Garey, D. Johnson, “Computers and intractability: a guide to the theory of NP-completeness”, *Freeman, 1979.*

[3] J. Hartmanis, R. Stearns, “On the computational complexity of algorithms”,*Trans. Amer. Math. Soc. 117, 285-306, 1965.*

[4] J. Hopcroft, W. Paul, L. Valiant, “On time versus space”, *J. ACM 24, 2, 332-337, 1977.*

[5] J. Hopcroft, J. Ullman, “Introduction to automata theory, languages, and computation”, *Addison-Wesley, Reading, MA, 1979.*

[6] D. Johnson, “A catalog of complexity classes”, in *J. van Leeuwen, editor, Handbook of Theoretical Computer Science, Volume A, “Algorithms and complexity”, 67-161, the MIT press/Elsevier, 1990.*

[7] R. Karp, “Reducibility among combinatorial problems”, in *R. Miller and J. Thatcher, editors, “Complexity of computer computations”, 85-103, Plenum Press, 1972.*

[8] C. Papadimitriou. “Computational Complexity”. Addison-Wesley, 1994.

[9] N. Pippenger, M. Fischer, “Relations among complexity measures”, *JACM, Vol. 26, No. 2, 1979, 361-381.*