CS 6743 **Lecture 2** **Fall 2007**

1 Worst case complexity

We now formally define the notion of worst case time complexity of Turing machines.

Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to *halt* (i.e., terminate in one of the two final states) on input x. (Each step is an execution of one instruction of the machine, and we define $t_M(x) = \infty$ if M never halts on input x.)

Definition 1 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

If T_M is polynomial in $|x|$, we say that the Turing machine runs in polynomial time; if it is linear in $|x|$, then in linear time.

Same for the *space* complexity, but instead of the number of steps $t_M(x)$ we maximize the number of tape cells visited.

2 Linear Speed-up theorem and multitape Turing machines

Last time we mentioned that a k-tape Turing machine can be simulated by a 1-tape Turing machine with quadratic increase in time complexity. Here is the outline of the proof.

Theorem 1 *If a k-tape Turing machine M decides a language L in time T_M, then there exists a 1-tape Turing machine M' that decides L in time $O((T_M)^2)$.*

Proof:

c*	a	a		
b*		b		
c	b*			

Imagine tapes "glued together" to form one tape in which each cell consists of a vertical stack of k cells. In order to record a position of the head on an i^{th} tape, we add a marker $*$ to a symbol in the corresponding track. E.g., in the example to the left, the head positions are on the first cell in the first two tapes, and on the second cell in the third tape.

Now, introduce a new symbol into the tape alphabet Γ' of the new TM M', in which each symbol will encode a k-tuple of symbols from Γ, with additional head markers. That is, if Γ has s symbols, then Γ' will have $(2s)^k$ additional symbols. The transition function δ is adjusted accordingly to accommodate the transitions on all the new symbols.

Now, to emulate one step of the k-tape TM M, the new TM M' will 1) scan all of the non-blank portion of the tape to determine where the head positions are and go to a corresponding state 2) scan the tape the second time to change the values in the cells and head positions according to the state obtained at the previous pass. Since the maximal number of cells in use is the running time of M, T_M, the maximum time M' will spend on this is $2T_M$. Therefore, one step of the new machine is linear in the running time of the old machine, and the total time will be $O((T_M)^2)$.

□

A similar reasoning can be used to prove the Linear Speed-Up theorem, showing that for any Turing machine M there is another deciding the same language that runs in a constant fraction time of the original (plus the size of the input). More formally,

Theorem 2 *If a language L is decided by some TM in time $T_M(n)$, then, for any $\epsilon > 0$, there is a TM M' that decides L in time $\epsilon T_M(n) + n + 2$.*

Proof sketch:

As in the case of k-tape to 1-tape reduction, encode k-tuples of symbols of M as single symbols of M', increasing the alphabet. First, rewrite the input in this form (this takes $n + n/k + 2$ steps). Now, to simulate k steps of the old machine, notice that k steps of the old machine affect only three cells of the new machine. It takes 6 steps of the machine to read and overwrite these cells with new information. Now, set $k = 6/\epsilon$. Then the total time of M' on input of size n is at most $\epsilon T_M(n) + n + 2$, as promised.

□

3 Review of computability and arithmetic hierarchy

Definition 2 *A language L is* decidable *if there exists a Turing machine which halts on all inputs and accepts all and only strings in L. Otherwise, a language is called* undecidable. *A language which is a set of strings accepted by some (not necessarily halting on all inputs) Turing machine is called semi-decidable.*

Recall the classic example of an undecidable language, the halting problem. The language HALT is defined as follows:

$$HALT = \{< M > | M \text{ is a Turing machine that halts on blank tape}\}.$$

Here, $< M >$ is some standard encoding of M.

It can be proven using diagonalization argument that $HALT$ is undecidable. However, $HALT$ is semi-decidable. One way of seeing it is to consider a Turing machine $M'_{HALT}(< M >, y)$ which takes as an additional input a string y encoding a sequence of steps (a computation) of $< M >$. On such an input M'_{HALT} runs through the steps listed in y checking that they are indeed valid steps of M on an initially blank tape, and that the last state in y is q_{accept} or q_{reject}. This Turing machine M'_{HALT} halts on all inputs, moreover, its running time is linear in $|y|$ (provided $|y|$ is sufficiently larger than $| < M > |$). Therefore, semi-decidable can be viewed as a set of languages which are "verifiable" – given a certificate, one can decide the solution.

Another way of writing it is to write L as a set definable by a formula $\exists y \phi(< M >, y)$, where ϕ is a formula true on $(< M >, y)$ if y is a correct sequence of steps of M ending in a halting state. This kinds of formulae (with one unbounded existential quantifiers) are called Σ_1 formulae. Note that this definition can be extended: a language L is co-semi-decidable if it can be defined by a formula of the form $\forall y \phi(< M >, y)$ (called Π_1 formulae). If a language is definable by both Σ_1 and Π_1 formulae, then it is decidable, since if a language and its complement are both semi-decidable then the language itself is decidable.

Of course, in this form it is easy to introduce arbitrary alternations of quantifiers, leading to more levels of complexity. The formulae of the form $\exists y_1 \forall y_2 \exists y_3 ... y_n$ are said to be on the n^{th} level of the *arithmetic hierarchy*. This hierarchy is strict: each subsequent level properly contains the previous, Σ_{i+1} properly contains both Π_i and Σ_i.

4 Complexity classes

If we take the definitions from the previous section and add the words "polynomially bounded" to them, we will obtain the main complexity classes. **Complexity started as computability with the polynomial time bound.** In particular:

- Replacing "decidable" by "polynomial-time decidable": obtain class P of polynomial-time decidable languages.
- In definition of semi-decidable, adding "polynomially verifiable" and "polynomial-size certificate" gives us the complexity class NP.
- Adding polynomial-time decidable to the formula and polynomial size to all y_i in the definition of arithmetic hierarchy gives us the polynomial-time hierarchy.

Modern complexity theory is a much richer field than just polynomial-bounded analogue of computability. Even in this course we will cover much more than these classes. But the most fundamental (or at least most well-known) question in complexity theory, P vs NP, can already be phrased with the definitions above. The computability analogue of this question, whether the class of decidable languages is different from semi-decidable, was solved using diagonalization. But we will see that diagonalization cannot be used to resolve P vs. NP.

Remark 1 We will see that in the logic setting the correspondences are slightly different. Although NP and polynomial-time hierarchy remain the same, the lowest class will be AC^0 rather than P.

5 Reductions

The most generic type of reduction (which is not very useful in complexity theory) is a *Turing* reduction. There, $A \leq_T B$ if A can be solved given a black-box access to an algorithm for B (this algorithm does not have to exists, the assumption is that if B can be solved, then so can A). This definition allows for usage of the result of the run of an algorithms: one can take its complement, run it multiple times and so on. In particular, in that case the halting problem is reducible to its own complement, which already does not respect the boundary of such classes as semi-decidable problems.

A more useful definition of reducibility is a *mapping* reducibility. There, $A \leq_m B$ if there is a computable function transforming an instance of A into an instance of B so that the new instance is in B iff the original was in A. More formally, there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that $f(x) \in B$ iff $x \in A$.

As before, we can put restrictions on the complexity of the function f. It does not make sense to use f more complex than the problem from which we are reducing: in that case, f can solve the problem and map it to a predefined string in B (or not in B). With this in mind, we define the following reducibilities:

- $A \leq_p B$: f is computable in polynomial time
- $A \leq_l B$: f is computable in logarithmic space.
- $A \leq_{fo} B$: f is definable by a first-order formula

Originally, NP-completeness results were proven using polynomial-time reducibility, but many were later shown to require much less complex reductions.