Notes for CS538 Lecture on 2/1/00

Detlef Ronneburger *

June 8, 2000

Nondeterministic Time Hierarchy Theorem

As mentioned in the previous lecture it is not possible to use a straight forward diagonalization to separate nondeterministic time classes for two different time bounds. It is widely believed that nondeterministic time bounded computations are not generally closed under complement. Thus it seems like a nondeterministic Turing machine cannot simulate another nondeterministic machine on the same input and reverse the answer.

However with a slightly more complicated argument it is possible to show the following theorem.

Theorem (Nondeterministic Time Hierarchy Theorem). *Let T be some time constructible function. Then for any function t for which $t(n+1) \in o(T(n))$ we know*

$$\mathsf{NTIME}[t(n)] \subsetneq \mathsf{NTIME}[T(n)] \quad .$$

Proof. We will construct a tally set A (i.e. $A \subseteq \{1\}^*$) where $A \in \mathsf{NTIME}[T] - \mathsf{NTIME}[t]$.

In order to define the Turing machine that decides A we partition $\mathbb{N}$ into larger and larger chunks. To do that we will use the following two recursively defined functions $start : \mathbb{N} \to \mathbb{N}$ and $end : \mathbb{N} \to \mathbb{N}$.

$$start(i) := \begin{cases} 1 & \text{if } i = 1 \\ end(i-1) + 1 & \text{if } i > 1 \end{cases} \tag{1}$$

$$end(i) := 2^{2^{i+start(i)+T(i)}} \tag{2}$$

So the natural numbers $\mathbb{N} = \{\, start(1), \ldots, end(1), start(2), \ldots, end(2), \ldots \}$ are split up into consecutive segments that grow double exponentially in size.

*detlef@paul.rutgers.edu

Since $end(i)$ is so much bigger than $start(i)$ there is a universal deterministic Turing machine can decide in time linear in $\mathcal{O}(end(i))$ given i and j, whether M_j accepts $1^{start(i)}$ in $\leq T(1^{start(i)})$ steps.

Having a deterministic machine that can simulate our nondeterministic machine on a smaller input enables us to construct a delayed diagonalization.

We will diagonalize against each machine M_j by fooling it on one of the inputs $x \in \left\{ 1^{start(i)}, \ldots, 1^{start(i)} \right\}$ where $i = j \cdot 2^k$ for a sufficiently big constant k which depends on the functions t and T.

Algorithm (defining hard language A).

(1) 1 ***input:*** 1^n ;
(2) 2 ***given:*** suitably large constant k;
(3) 3 find i such that $i = j \cdot 2^k$ and $start(i) \leq n \leq end(i)$, otherwise ***reject***;
(4) 4 `if` $(n = end(i))$ `then`
(5) 5 ***accept*** $\iff$ M_j does not accept $1^{start(i)}$ in time $\leq T(start(i))$;
(6) 6 `else`
(7) 7 ***accept*** $\iff$ universal NTM accepts $(j, 1^{n+1})$ in time $\leq T(n)$;
(8)

Computation only takes places in Lines 3, 5 and 7. In Line 3 we can deterministically just check all possible values for i in increasing order and verify the conditions. Thus this will take at most linear time in n.

As we noted above it is possible to deterministically simulate within time $\leq T(n)$ the machine M_j on $1^{start(i)}$ and check whether it accepts in at most $T(start(i))$ steps. Thus Line 5 can be performed within the resource bound.

Finally we can use the fact that nondeterministically simulating an NTM does not cause any relevant overhead and thus Line 7 only requires time $T(n)$ as well. Therefore the above algorithm defines a language $A \in \mathsf{NTIME}[T(n)]$.

Now we will prove by contradiction that $A \notin \mathsf{NTIME}[t(n)]$. Assume that the nondeterministic Turing machine M_j decides A in at most $t(n)$ steps.

Let us consider the input 1^n such that $n = start(i)$ and $i = j \cdot 2^k$.

$$\begin{aligned}
1^n \in A &\iff 1^{start(i)} \in A \\
&\iff \text{universal NTM accepts } (j, 1^{n+1}) \text{ in } \leq T(n) \text{ steps} \\
&\iff M_j \text{ accepts } 1^{n+1} \\
&\iff 1^{n+1} \in A \\
&\iff \text{universal NTM accepts } (j, 1^{n+2}) \text{ in } \leq T(n+1) \text{ steps} \\
&\iff M_j \text{ accepts } 1^{n+2} \\
&\iff 1^{n+2} \in A \\
&\iff \ldots \\
&\iff 1^{end(i)} \in A \\
&\iff M_j \text{ does not accepts } 1^{start(i)}) \text{ in } \leq T(start(i)) \text{ steps.} \\
&\iff 1^{start(i)} \notin A
\end{aligned}$$

All equivalences directly follow from the algorithm defining A. The only one that might require explanation is

$$M_j \text{ accepts } 1^{n+1} \iff \text{universal NTM accepts } (j, 1^{n+1}) \text{ in } \leq T(n) \text{ steps} \quad .$$

If M_j accepts 1^{n+1} then it does so in time $t(n+1)$ which is less than $T(n)$ for sufficiently large n. So the universal NTM will accept the input after simulating M_j for $t(n+1) < T(n)$ steps.

If M_j does not accept 1^{n+1} then it also does so in time $t(n+1) < T(n)$. So again the universal NTM will able to simulate M_j and reject the input after $t(n+1) < T(n)$ steps.

As observed above this method only works for sufficiently large n. However by selecting the constant k big enough, depending on the functions t and T, we will only consider those large enough inputs. All the inputs that are too small will be rejected since we will not be able to find an $i = j \cdot 2^k$ such that $start(i) \leq n \leq end(i)$.

Clearly the statement $1^{start(i)} \in A \iff 1^{start(i)} \notin A$ is a contradiction and thus M_j cannot properly decide A. Therefore $A \notin \mathsf{NTIME}[t(n)]$. □

Note that this does not give us any information for very large functions t and T. So for example it is not known whether

$$\mathsf{NTIME}[2^{2^{2^n}}] \subsetneq \mathsf{NTIME}[(2^{2^{2^n}})^2] \quad .$$

This is due to the fact that for $t(n) = 2^{2^{2^n}}$ and $T(n)(2^{2^{2^n}})^2$ the property $t(n+1) \in o(T(n))$ does not hold.

However there are oracles known relative to which those to classes are disjoint and oracles relative to which they coincide.

Reducibility

Reducibility is a powerful tools that allows us to

(i) solve problems using solutions from similar problems and

(ii) use the known hardness of a problem to show hardness of another problem.

So for example Stockmeyer gave precise hardness calculation in his dissertation for the problem WS1S which is a set of certain logical formulas. He used the following theorem for his(???) hardness result.

Theorem. *There is a set $A \in \mathsf{DSPACE}[2^n]$ that can only be computed by circuits of size at least $2^{\frac{n}{2}}$.*

Proof. Define the language with the following algorithm.

Algorithm (defining hard language A).

```
input:  x, n = |x|;
    foreach (s ∈ {0,1}^n)
    /* check all possible candidates for characteristic string λ_A.   */
        cycle through all circuits C of size ≤ 2^(n/2) and check if they agree with s;
        if (no circuit agrees with s) then
        /* no small circuit computes set with characteristic string s*/
            accept ⟺ s(x) = 1;
        else
            next s;      /* try next characteristic string */
```

It is not hard to see that this algorithm runs in linear exponential space and yet diagonalizes against every possible circuit of size $\leq 2^{\frac{n}{2}}$. □

Now Stockmeyer showed that there is an easily computable function f such that (???)

$$x \in A \iff f(x) \in \text{WS1S} \quad .$$

So if WS1S was computable by small circuits then membership for A would be easy to decide by computing $f(x)$ and then using the small circuit for WS1S to decide whether $f(x) \in$ WS1S or not. From this we could directly conclude whether $x \in A$.

However we proved above that A is not easy to compute and therefore WS1S cannot have small circuits.

This type of argument is an example of a reduction from the problem A to the problem WS1S (more precisely it is a *many-one reduction*).

The idea is that we want to say one problem is reducible to another problem, if it is "at least as easy" as the other problem. In particular we would like meet the following two properties.

(i) If a function f is computable in time $\mathcal{O}(n^2)$ then we want to consider it to be "easy".

(ii) If two functions f and g are "easy", then the function $f \circ g$ should also be viewed as "easy".

Unfortunately those two demands imply that we have to consider any function computable in time n^k for any k as "easy". This leads to a slight distortion since for practical purposes a function computable in time n^{2^100} is certainly not "easy" to compute for most inputs.

However it turns out that the notion described above is very useful for theoretical considerations. Thus we will define it here.

Definition. A problem A is *polynomial time many-one reducible* to a problem B (notation $A \leq_m^{\mathsf{P}} B$) if there is a function f computable in time n^k for some k such that

$$x \in A \iff f(x) \in B \quad .$$

We will further say to problems are *polynomial time many-one equivalent* ($A \equiv_m^{\mathsf{P}} B$) if $A \leq_m^{\mathsf{P}} B$ and $B \leq_m^{\mathsf{P}} A$.

In particular this definition implies that if $B \in \mathsf{DTIME}[t(n)]$ and $A \leq_m^{\mathsf{P}} B$ then $A \in \mathsf{DTIME}[t(n)^k]$. The same holds if B being contained in a nondeterministic time class or in a space bounded complexity class.

Since we consider any problem A that is $\leq_m^{\mathsf{P}}$-reducible to a set B to be at least as "easy" as B we would like to consider complexity classes that are closed under this reduction. In other words we are interested in complexity classes C where $A \leq_m^{\mathsf{P}} B$ and $B \in \mathsf{C}$ implies $A \in \mathsf{C}$.

This consideration leads to the definition of the following complexity classes.

Definition 1.

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}[n^k] = \mathsf{DTIME}[n^{\mathcal{O}(1)}] \qquad \mathsf{NP} = \bigcup_{k \in \mathbb{N}} \mathsf{NTIME}[n^k] = \mathsf{NTIME}[n^{\mathcal{O}(1)}]$$

$$\mathsf{EXP} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}[2^{n^k}] = \mathsf{DTIME}[2^{n^{\mathcal{O}(1)}}] \qquad \mathsf{NEXP} = \bigcup_{k \in \mathbb{N}} \mathsf{NTIME}[2^{n^k}] = \mathsf{NTIME}[2^{n^{\mathcal{O}(1)}}]$$

$$\mathsf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{DSPACE}[n^k] \qquad \mathsf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{NSPACE}[n^k]$$

$$\mathsf{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{DSPACE}[2^{n^k}] \qquad \mathsf{NEXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{NSPACE}[2^{n^k}]$$