Time and Space Classes
Exposition by William Gasarch

1 Deterministic Turing Machines

Turing machines are a model of computation. It is believed that anything that can be computed can be computed by a Turing Machine. The definition won't look like much, and won't be used much; however, it is good to have a rigorous definition to refer to.

Def 1.1 A *Turing Machine* is a tuple $(Q, \Sigma, \delta, s, h)$ where

- Q is a finite set of states. It has the states s, q_{acc}, q_{rej}.
- Σ is a finite alphabet. It contains the symbol #.
- $\delta : Q - \{q_{acc}, q_{rej}\} \times \Sigma \to Q \times \Sigma \cup \{R, L\}$
- $s \in Q$ is the start state, q_{acc} is the accept state, q_{rej} is the reject state.

We use the following convention:

1. On input $x \in \Sigma^*$, $x = x_1 \cdots x_n$, the machine starts with tape

$$\#x_1x_2 \cdots x_n\#\#\#\# \cdots$$

that is one way infinite.

2. The head is initially looking at the x_n.
3. If $\delta(q, \sigma) = (p, \tau)$ then the state changes from q to p and the symbol σ is overwritten with τ. The head does not move.
4. If $\delta(q, \sigma) = (p, L)$ then the state changes from q to p and the head moves Left one square. overwritten with τ. The head does not move. (Similar for $\delta(q, \sigma) = (p, R)$.
5. If the machine is in state h then it is DONE.
6. If the machine halts in state q_{acc} then we say M ACCEPTS x. If the machine halts in state q_{rej} then we say M REJECTS x.

Important Note: We can code Turing machines into numbers in many ways. The important think is that when we do this we can, given a number i, extract out which Turing Machine it corresponds to (if it does not correspond to one then we just say its the machine that halts in one step on any input). Hence we can (and will) say things like

- Let $M_1, M_2, M_3, \ldots$ be a list of all Turing Machines.
- Run $M_i(x)$. This is easy- given i, we can find M_i, — that is, find the code for it, and then run it on x.

Def 1.2 A function f is COMPUTABLE if there is a Turing Machine M such that

$$(\forall x)[M(x) = f(x)]$$

Def 1.3 A set A is DECIDABLE if there is a Turing Machine M such that
For all x:

$$x \in A \rightarrow M(x)\text{ACCEPTS}$$

$$x \notin A \rightarrow M(x)\text{REJECTS}$$

2 Nondeterministic Turing Machines

Def 2.1 A *Turing Machine* is a tuple $(Q, \Sigma, \delta, s, h)$ where

- Q is a finite set of states
- Σ is a finite alphabet. It contains the symbol #.
- $\delta : Q - \{q_{acc}, q_{rej}\} \times \Sigma \rightarrow 2^{\Sigma \rightarrow Q \times \Sigma \cup \{R,L\}}$
- $s \in Q$ is the start state
- $h \in Q$ is the halt state.

So if the machine is in a configuration there are MANY choices of what it can do.

Def 2.2 Let M be a nondet TM. The set of x that M ACCEPTS is

$$\{x \mid \text{ SOME choice of moves of the nondet machine leads to } q_{acc} \}.$$

(We do not define the notion of M computing a function. There are several ways to do this, but they will not be useful for us.)

We can define a MULTITAPE Turing Machine. We leave it to you to define formally. Note that each machine has a constant number of tapes.

3 Time and Space Classes

Def 3.1 Let $T(n)$ be a computable function (think of it as increasing). A is in DTIME($T(n)$) if there is a MULTITAPE TM M that decides A and also, for all x, $M(x)$ halts in time $\leq T(|x|)$. *Convention:* By DTIME($T(n)$) we really mean DTIME($O(T(n))$. They are actually equivalent by having your TM just take bigger steps.

Note that this is unfortunately machine dependent. It is possible that if we allow 2-tapes instead of one it would change how much you can do. We won't have to deal with this much since we will usually define classes in terms of multi-tape machines, and we will allow some slack on the time bound, like: DTIME($n^{O(1)}$).

It is known that a Multitape DTIME($T(n)$) machine can be simulated by (1) a 1-tape DTIME($T(n)^2$) TM, and also0 (2) a 2-tape DTIME($T(n) \log T(n)$) TM.

Def 3.2 Let $S(n)$ be a computable function (think of it as increasing). A is in DSPACE($S(n)$) if there is a TM M that decides A and also, for all x, $M(x)$ only uses space $S(|x|)$. *Convention:* By DSPACE($S(n)$) we really mean DSPACE($O(S(n))$. They are actually equivalent by having your TM just take bigger steps. *Convention:* When dealing with space classes we will have an input tape which is read-only and a separate worktape. When dealing with space-bounded TMs computing functions we will also have a write-only output tape.

It is known that a Multitape DSPACE($S(n)$) machine can be simulated by a 1-tape DSPACE($S(n)$) TM.

Def 3.3 Let $T(n)$ be a computable function (think of it as increasing). A is in NTIME($T(n)$) if there is a Nondet TM M that decides A and also, for all x, $M(x)$, on any path, halts in time $\leq T(|x|)$. *Convention:* By NTIME($T(n)$) we really mean NTIME($O(T(n))$. They are actually equivalent by having your TM just take bigger steps.

Def 3.4 Let $S(n)$ be a computable function (think of it as increasing). A is in NSPACE($S(n)$) if there is a Nondet TM M that decides A and also, for all x, $M(x)$, on any path, only uses space $\leq S(|x|)$. *Convention:* By NSPACE($S(n)$) we really mean NSPACE($O(S(n))$. They are actually equivalent by having your TM just take bigger steps.

4 The Only Theorem about Time Classes Worth Mentioning

In the next section we will discuss what is wrong with these classes. But before that I will prove one theorem bout DTIME($T(n)$) which is interesting. In the next section

I will discuss why this theorem is interesting even though, in general DTIME($T(n)$) is not.

Important Note: Imagine doing the following: Take a list of TMs $M_1, M_2, \ldots$ and then bound M_i by $T(n)$. That is, when you run M_i if it has not halted by $T(|x|)$ steps then shut it off and declare its answer to be 0. To save on notation we will also call this list

$$M_1, M_2, M_3, \ldots$$

KEY- if a set is in DTIME($T(n)$ then there is an i such that M_i decides it in time $T(n)$.

And now the one interesting theorem.

Theorem 4.1 *Let $T(n)$ be any computable function. Then there exists a decidable set $A \notin$ DTIME($T(n)$).*

Proof: We give an algorithm for the set A which will define A and show that A is computable.

Warning: A is a contrived set. It was constructed for the sole purpose of being decidable but not in DTIME($T(n)$.

Let $M_1, M_2, \ldots$, be a list of all Turing Machines.

We define the set A.

1. Input(x)
2. Run $M_x(x)$ for $T(|x|)$ steps.
   (a) If $M_x(x)$ halts within $T(|x|)$ steps and outputs 1 (which means yes) then output 0 (which means no)
   (b) If $M_x(x)$ has any other behavious (e.g. does not halt within $T(|x|)$ steps, halts and says 1, halts and outputs 1993) then output 1.

Let

$$A = \{x : \text{ The above algorithm outputs 1 on input x } \}$$

We claim that $A \notin$ DTIME($T(n)$).

Assume that $A \in$ DTIME($T(n)$). Then there is a machine M_x such that, for all y:

- $M_x(y) = A(y)$.
- $M_x(y)$ halts within $T(|y|)$ steps

Look at $M_x(x)$. It halts within $T(|x|)$ steps. Hence the algorithm above would KNOW what $M_x(x)$ is AND intentionally given the OTHER answer. Hence M_x and A differ on x. So M_x DOES NOT decide A. Contradiction. ∎

5 Whats Wrong with the Time Space Classes and How to Fix the Problem

The problem with DTIME($T(n)$) and the other classes is that they are *model dependent*. When talking about DTIME(n^2) you really need so say if you are working with a 1-tape Turing Machine or a 2-tape Turing Machine or other variants. An example look at the language

$$PAL = \{w : w = w^R\}.$$

On a 2-tape Turing Machine $PAL \in$ DTIME(n): Copy the string to the other tape and then compare them. On a 1-tape Turing Machine it seems to require n^2 time, and in fact one can prove this. So on a 2-tape Turing Machine $PAL \in$ DTIME(n^2) and, for all $T(n)$ such that $T(n) << n^2$, $PAL \notin$ DTIME($T(n)$). To say all of this properly we would need a subscript to denote how many tapes. But it doesn't stop there! What about if we have a 2-dimensional Turing Machine? 3-dimensional? k-dimensional? What if we have a Turing Machine that is allowed to do the Hokey Pokey and Turn itself about (cause THATS what its all about!).

We are NOT going to go there. We are NOT going to define TIME and SPACE on all variants of Turing Machines So what to do instead? There are a few options:

1. If you want to study SORTING then use a model of computation that fits that problem. Decision trees works there. More generally let the model fit the application. This works for particular problems and even classes of problems. The Pointer Machine Models are one success story. One can even view DFA's in this light. The PRO of this approach is that we can make fine distinctions: n vs $n \log n$ that matter for the real world. problems people care about. The CON is that we can't prove or even define large classes of problems. This work is very interesting and given how I change this course around it may be in this course someday; but not today.

2. We noted above that PAL is in DTIME(n) on 1-tape TM's but requires DTIME(n^2) on 2-tape TM's. Whats a square factor between friends? What is we decide *we don't care about polynomial factors*. It turns out that ALL variants of Turing machines are equivalent up to poly time and space. Hence we can define P, NP by a 1-tape TM confident that the definition will not change if you go to 2-tape or even a TM that does the hokey pokey. The PRO of this approach is that we can prove theorems about classes of problems. The CON is that we cannot make fine distinctions. To us n and n^{100} are the same. This is the approach we take. We will need to use 1-Tape TM's a few times but not much.

Def 5.1 For all of the definitions below, 1-tape and multitape are equivalent. This is important in the proof of Cooks theorem and later in the proof that a particular lang is EXPSPACE complete and hence not in P.

1. $P = \text{DTIME}(n^{O(1)})$.
2. $NP = \text{NTIME}(n^{O(1)})$. This is equivalent of just using 1-tape TM's. (This is equivalent to our quantifier definition.)
3. $EXP = \text{DTIME}(2^{n^{O(1)}})$.
4. $NEXP = \text{NTIME}(2^{n^{O(1)}})$.
5. $L = \text{DSPACE}(O(\log n))$.
6. $NL = \text{NSPACE}(O(\log n))$.
7. $PSPACE = \text{DSPACE}(n^{O(1)})$.
8. NSPACE = $\text{NSPACE}(n^{O(1)})$.
9. $EXPSPACE = \text{DSPACE}(2^{n^{O(1)}})$.
10. $NEXPSPACE = \text{NSPACE}(2^{n^{O(1)}})$.

6 Easy Relations Between Classes

The following theorem is trivial.

Theorem 6.1 *Let $T(n)$ and $S(n)$ be computable functions (think of as increasing).*

1. $\text{DTIME}(T(n)) \subseteq \text{NTIME}(T(n))$.
2. $\text{DSPACE}(S(n)) \subseteq \text{NSPACE}(S(n))$.
3. $\text{DTIME}(T(n)) \subseteq \text{DSPACE}(T(n))$.
4. $\text{NTIME}(T(n)) \subseteq \text{NSPACE}(T(n))$.

The following theorem is easy but not trivial.

Theorem 6.2 *Let $T(n)$ and $S(n)$ be computable functions (think of as increasing).*

1. $\text{NTIME}(T(n)) \subseteq \text{DTIME}(2^{O(T(n))})$. *(Just simulate ALL possible paths.)*
2. $\text{NTIME}(T(n)) \subseteq \text{DSPACE}(O(T(n)))$. *(Just simulate ALL possible paths- keep a counter for which path you are simulating.)*

7 Sophisticated Relations Between Classes

Theorem 7.1 *Let $S(n)$ be computable functions (think of as increasing). Then*

$$\text{NSPACE}(S(n)) \subseteq \text{DSPACE}(S(n)^2).$$

Proof: Let $A \in \text{NSPACE}(S(n)$ via TM M. Given x we want to determine if SOME path in $M(x)$ goes to an accept state.

We will assume M has 1 worktape. The modification for many tapes are easy.

A *configuration* (henceforth config) is a snapshot of the $S(n)$ squares of the worktape, the state, and where the head is. It does not include the input.

Note that we CANNOT write down all of possible configs. Even so, consider the following: There are $2^{O(S(n)}$ configurations. Consider them to be nodes of a graph (which we CANNOT write down). There is an edge from C to D if from C you can get, in one step to D. Since M is nondeterministic there may be more than one. Note that you have to know x as well as C and D to tell if there is an edge.

We can now restate the problem: Given a directed graph by being given a program E that will, given two nodes $(a, b$ can tell if (a, b) is an edge, and given two nodes s, t (in our case s is the start configuration and t is the accepting configuration which we can assume is unique) determine if there is a path from s to t. The graph has N nodes. (In our case $N = 2^{O(S(n))}$.) Do the problem in $O((\log N)^2)$ space).

We assume the graph is fixed and write a program E that will, give a, b, t, tell if there is a path from a to b of length $\leq t$.

$ALG(a, b, t)$

1. If $t = 0$ then if $a = b$ output YES, else output NO.
2. If $t = 1$ then if $E(a, b) = 1$ output YES, else output NO.
3. For all $v \in V$

   If $ALG(a, v, t/2) = YES$ AND $ALG(v, b, t/2) = YES$ then output YES

The depth of recursion is $O(\log t)$ and each recursion need to store a, b, t which is $O(\log t)$. Hence the program runs in space $O((\log t)^2)$.

We run this on OUR graph with $t = 2^{S(n)}$ and get the result. ∎

Corollary 7.2 $PSPACE = NPSPACE$.

What do we know about NL ? Using the above we get

$$\text{NSPACE}(\log n) \subseteq \text{DSPACE}((\log n)^2)) \subseteq \text{DTIME}(2^{(\log n)^2}).$$

Can we do better? YES!

Theorem 7.3 $NL \subseteq P$.

Proof: Let $A \in NL$ via TM M. Given x we want to determine if SOME path in $M(x)$ goes to an accept state.

We will assume M has 1 worktape. The modification for many tapes are easy.

A *configuration* is as in the last theorem. KEY- we CAN write down all possible configs. In the last theorem we had a graph IMPLICITLY. Here we have one EXPLICITLY.

Write down all of possible config. There are only $2^{O(\log n)}$ of them which is some poly, say n^c. Consider them to be nodes of a graph. We draw a directed graph from u to v if from u you CAN go to v in one step (note that this depends on both u and x).

$x \in A$ iff there is a path from the start config to an accept config in the graph. This can be determined in time poly in the size of the graph which is poly in n^c so poly. ∎

Note 7.4 So we know $NL \subseteq P$. Do we think $NL \neq P$ or $NL = P$? Do we think this is hard to determine? We think $NP \neq P$. And we think it will be hard to determine. Why do we think that? There has been work no proving that some problems are hard to determine. NL vs P was one of them. It was part of my PhD thesis.

8 Time and Space Hierarchy Theorems

Important Note: Imagine doing the following: Take a list of TMs $M_1, M_2, \ldots$ and then bound M_i by $T(n)$. That is, when you run M_i if it has not halted by $T(|x|)$ steps then shut it off and declare its answer to be 0. To save on notation we will also call this list

$$M_1, M_2, M_3, \ldots$$

KEY- if a set is in DTIME($T(n)$ then there is an i such that M_i decides it in time $T(n)$.

KEY- if M_i decides a set then it is in DTIME($T(n)$).

Hence we have a list that represents all of DTIME($T(n)$).

Does more time help? YES but the proof involves some details we will skip.

Theorem 8.1 *(The Time Hierarchy Theorem) For all computable increasing $T(n)$ there exists a set A such that*

$$A \in \text{DTIME}(T(n) \log T(n))) - \text{DTIME}(T(n)).$$

Proof: Let $M_1, M_2, \ldots,$ represent all of DTIME($T(n)$) as described above.

We construct a set A to NOT be in DTIME($T(n)$). We will want A and to DISAGREE with M_1, to DISAGREE with M_2, etc. Lets state this in terms of REQUIREMENTS

R_i : A and M_i differ on some string.

We want A to satisfy all of these requirements.

Here is our algorithm for A. It will be a subset of 0^*.

1. Input 0^i.
2. Run $M_i(0^i)$. If the results is 1 then output 0. If the results is 0 then output 1.

Note that, for all i, M_i and A DIFFER on 0^i. Hence every R_i is satisfied. Therefore $A \notin$ DTIME($T(n)$).

How do we get $A \in$ DTIME($T(n)) \log T(n)$)? It is KNOWN that any multitape DTIME($T(n)$) TM can be SIMULATED by a 2-tape TM in time $T(n) \log T(n)$. So we use this to run $M_i(0^i)$ in the algorithm. The entire algorithm can then be one in time $T(n) \log T(n)$. ∎

Corollary 8.2 *$P \subset EXP$.*

The following theorem is proved similarly:

Theorem 8.3 *(The Space Hierarchy Theorem) Let S_1 and S_2 be computable increasing functions. Assume $\lim_{n\to\infty} \frac{S_1(n)}{S_2(n)} = \infty$. Then there exists a set A such that*

$$A \in \text{DSPACE}(S_1(n))s \subseteq \text{DSPACE}(S_1(n)).$$

Corollary 8.4 *$L \subset PSPACE$.*

There are also nondeterministic versions of the hierarchy theorems which we state without proof. The proofs are similar but a bit more technical.

Theorem 8.5 *(The Nondet Time Hierarchy Theorem) For all computable increasing $T(n)$ there exists a set A such that*

$$A \in \text{NTIME}(T(n) \log T(n))) - \text{NTIME}(T(n)).$$

Corollary 8.6 *$NP \subset NEXP$.*

Theorem 8.7 *(The Nondet Space Hierarchy Theorem) Let S_1 and S_2 be computable increasing functions. Assume $\lim_{n\to\infty} \frac{S_1(n)}{S_2(n)} = \infty$. Then there exists a set A such that*

$$A \in \text{NSPACE}(S_1(n))s \subseteq \text{NSPACE}(S_1(n)).$$

Corollary 8.8 $NL \subset$ NSPACE.

Note that NSPACE $= PSPACE$ and $NL \subseteq P$. Hence we have

$$NL \subseteq P \subseteq PSPACE$$

and

$$NL \subset PSPACE.$$

Therefore we have that EITHER
$NL \subset P$ OR $P \subset PSPACE$. We think both are true.