Introduction to Complexity Theory*
Lecture 19: Dtime vs Dspace

Notes taken by Tamar Seeman and Reuben Sumner

June 13, 1999

Summary: In this lecture, we prove that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. That is, we show how to simulate any given deterministic multi-tape Turing Machine (TM) of time complexity t, using a deterministic TM of space complexity $t/\log t$.

1 Introduction

We begin by defining Dtime(t) and Dspace(t).

Definition 1 (Dtime): *$Dtime(t(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM M which decides whether or not $x \in L$ in less than $t(|x|)$ Turing machine steps.*

Definition 2 (Dspace): *$Dspace(s(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM which decides whether or not $x \in L$ while never going to the right of cell $s(|x|)$ on any of its tapes.*

Since any Turing machine step can move the head(s) by at most one position, it is immediately clear that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot))$. Furthermore NP is easily in $Dspace(p(\cdot))$ for some polynomial p, but is not believed to be in $Dtime(q(\cdot))$ for any polynomial q. Thus it seems that space is much more powerful than time. In this lecture we will further refine this intuition by proving that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. It follows that $Dtime(t(\cdot))$ is a strict subset of $Dspace(t(\cdot))$, since it has already been shown that $Dspace(o(t))$ is a strict subset of $Dspace(t)$.

Note: The multi-tape TM consists of one read-only, bi-directional input tape, one optional write-only output tape (in the case of a machine to decide a language we will not include such a tape and determine acceptance based on the exact state that the Turing machine terminates in) together with a fixed number of bi-directional read/write work tapes. The number of work tapes is irrelevant for $Dspace()$, though, since a TM M with k work tapes can be simulated by a TM M' with just a single work tape and the same amount of space. This is done by transforming k work tapes into one work tape with k tracks. This transformation then simulates the work of the original Turing machine by using polynomially more time, but the same amount of space. However, in the case of $Dtime()$ the number of tapes *does* matter.

---
* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

2 Main Result

Theorem 3 *If $t(\cdot)$ is constructible in space $t(\cdot)/\log t(\cdot)$ and $t(\cdot)$ is at least linear, then $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$.*

In order to make the proof of this theorem as readable as possible we state some results (without proof) where they are needed so that their motivation is clear, but we prove them only later so as not to disturb the flow of the proof.

Proof: Let L be a language accepted by a TM M in $Dtime(t(\cdot))$. Let x be an input and $t = t(|x|)$. Divide each tape into $t^{1/3}$ blocks of $t^{2/3}$ cells. (This ensures that $(\#\text{ blocks})^2 \ll t$, a necessary feature.) Similarly, partition time into periods of $t^{2/3}$ steps. During a particular period, M visits at most two blocks of space on each tape, since the number of steps M can take does not exceed the size of a single block.

Lemma 2.1 *Without loss of generality such a machine moves between block boundaries only on the very last move of a time period. This holds with at most a linear increase in the amount of time used, and a constant number of additional tapes.*

Therefore without loss of generality we assume that in each time period our machine stays within the same block on each work tape.

Our goal now is to compute the final state of the machine, which will indicate whether or not our input is in the language. We therefore have to somehow construct the blocks that we are in at the final time period while never storing the complete state of the work tapes of the machine since these would potentially exceed our space bound. To do so, we establish a dependency graph between different states and find a sufficiently efficient method of evaluating the graph node corresponding to the final machine state.

We introduce some notation to describe the computation of the machine.

$h_i(j)$ is the block location of the i^{th} tape head during the j^{th} period. $h_i(j) \in \left\{1, \ldots, t^{1/3}\right\}$.

$h(j)$ is the vector $h_1(j), h_2(j), \ldots, h_k(j)$ for a k-tape machine.

$c_i(j)$ is the content of block $h_i(j)$ on the i^{th} tape, together with the Turing machine state at the end of the period and head position on tape i. $c_i(j) \in \{0,1\}^{t^{2/3}} \times O(1) \times \{0,1\}^{O(\log t)}$. [1]

$c(j)$ is the vector $c_1(j), c_2(j), \ldots, c_k(j)$.

$l_i(j)$ is the last period where we visited block $h_i(j)$ on tape i. This is $\max\{j' < j \text{ such that } h_i(j') = h_i(j)\}$.

In order to compute $c(j)$ we will need to know:

- $c(j-1)$ to determine what state to start our computation in.
- $c_1(l_1(j)), \ldots, c_k(l_k(j))$ so that we know the starting content of the blocks we are working with.

It is immediately clear then that we need at most $k+1$ blocks from past stages of computation to compute $c(j)$.

Define a directed graph $G = (V, E)$ where $V = \{1, \ldots, t^{1/3}\}$ and $E = \{(j-1, j) | j > 1\} \cup \{(l_i(j), j)\}$. So vertex i represents knowledge of $c(i)$. There is an edge $(j-1) \to j$ since $c(j)$ depends on $c(j-1)$.

Figure 1: An interesting pebbling problem

Similarly there is an edge $l_i(j) \to j$ since $c(j)$ depends on $c_i(l_i(j))$ for all i. Hence this graph represents exactly the dependency relationship that we have just described.

Consider the example in Figure 1. The most obvious way to reach node 6 would be

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and keep state 2
- calculate state 4 from state 3 and keep state 3 as well as still keeping state 2
- calculate state 5 from states 3 and 4 and erase states 3 and 4
- calculate state 6 from states 2 and 5 and erase states 2 and 5

Notice that when calculating state 5 we had in memory states 2,3 and 4, a total of three prior states. We can do better.

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and *erase* state 2
- calculate state 4 from state 3 and keep state 3
- calculate state 5 from states 3 and 4 and then erase both states 3 and 4
- calculate state 1 from the starting state (for the second time!)
- calculate state 2 from state 1 and erase state 1
- calculate state 6 from states 2 and 5 and then erase both

This second calculation required two more steps to compute but now we never needed to remember more than two previous states, rather than three. It is exactly this tradeoff of time versus space which enables us to achieve our goal.

In general on a directed acyclic graph of bounded degree we can play a pebble game. The rules of the game are simple:

1. A pebble may be placed on any node all of whose parents have pebbles.
2. A pebble may be removed from any node.
3. A pebble may be placed on any node having no parents. This is a special case of the first rule.

[1]We assume that $\Sigma = \{0, 1\}$, where Σ is the tape alphabet

The goal of the pebble game is to pebble a given target node while minimizing the number of pebbles used at any one time.

This game corresponds to calculations of our Turing machine. A pebble on vertex j in the game corresponds to a saved value for $c(j)$. When the game tells us to pebble node j then we can recover the contents of $c(j)$ as follows:

1. Load contents of $c_1(l_1(j)), \ldots, c_k(l_k(j))$ from storage. Since there was an edge from $l_i(j) \to j$ in the graph, Rule 1 guarantees that node $l_i(j)$ has a pebble on it. Since $l_i(j)$ has a pebble in the pebble game, then in our computation we have $c(l_i(j)$ and therefore $c_i(l_i(j))$ in storage.

2. Load the starting state and head position for period j from $c(j-1)$, again guaranteed to be available by the pebble game.

3. Based on all the above, it is easy to reconstruct $c(j)$.

To compute the answer of the original Turing machine we calculate $c(t^{1/3})$.

Theorem 4 *For any directed acyclic graph $G = (V, E)$ with in-degree bound d, and any $v \in V$, we can pebble v while never using more than $O(|V|/\log|V|)$ pebbles simultaneously.*

Notice, however, that the theorem does not state anything about how efficient it is to compute this pebbling.

Now to simulate the machine M on input x we proceed as follows.

1. Compute $t = t(|x|)$

2. Loop over all possible guesses of $h(1), \ldots, h(t^{1/3})$. For each one do:

   (a) Compute and store the dependency graph.

   (b) Execute a pebbling strategy to reach node $t^{1/3}$ on the graph as follows:

      i. Determine next pebble move

      ii. if the move is "Remove i" then erase $c(i)$

      iii. if the move is "Pebble i" then set the machine to the initial state for period i by loading the contents of the work tape from storage. Calculate $c(i)$. Verify that in step 2 we did guess $h(i+1)$ correctly. If not, then abort the whole calculation so far and return to the next guess in step 2. Otherwise save $c(i)$ for future use.

   (c) Having just executed "Pebble $t^{1/3}$", terminate and accept or reject according to the original Turing machine.

We need to show that all of the above computation can be performed within our space bound.

- Step 1 can be performed within our space bound by our hypothesis regarding the (space constructability of the) function $t(\cdot)$.

- In step 2a we store a graph $G = (V, E)$ where $|V| = t^{1/3}$ and $|E| \leq |V| \cdot (k+1)$. A simple list representation of the graph will therefore take $(k+1) \cdot t^{1/3} \cdot \log_2(t^{1/3}) \ll t/\log t$ space.

- Step 2(b)ii actually frees space rather than using space.

...	$Rev(b_i(j-2))$ $b_i(j-1)$ $Rev(b_i(j))$	$Rev(b_i(j-1))$ $b_i(j)$ $Rev(b_i(j+1))$	$Rev(b_i(j))$ $b_i(j+1)$ $Rev(b_i(j+2))$	...

Figure 2: The new work tape

- Step 2(b)iii needs space only for k blocks copied from storage, and space to store the result. Since our pebble game guarantees that we will never need more than $t^{1/3}/\log t^{1/3}$ pebbles, we will never need more than $(t^{1/3}/\log t^{1/3}) \cdot (k+1) \cdot t^{2/3} = O(t/\log t)$ space, our space bound.

So aside from step 2(b)i all of step 2b can be calculated within our space bound.

We need to describe now how to perform step 2(b)i. Consider a non-deterministic algorithm. We set a counter to be the maximum number of possible pebble moves to reach our target. Since it never makes sense to return to an earlier pebbling configuration of the graph, we can bound the number of steps by $2^{t^{1/3}}$. Such a counter will only require $t^{1/3}$ space, which is within our bound. We now non-deterministically make our choice of valid pebbling move and decrement the counter. We accept if we pebble the target vertex, reject if the counter reaches its limit first. The dominant space required by the routine is therefore the space required to store a working copy of the graph, $O(t^{1/3} \cdot \log t)$.

In Lecture 5 we proved that $Nspace(s) \subseteq Dspace(s^2)$. Therefore, the above non-deterministic subprogram which runs in $Nspace(t^{1/3} \cdot \log t)$ can be converted to a deterministic subprogram running in $Dspace(t^{2/3} \cdot \log^2 t \ll t/\log t)$. ■

3 Additional Proofs

3.1 Proof of Lemma 2.1

Our new machine M' which simulates the operation of M works as follows. Firstly each work tape is replaced by a three-track work tape. Three additional work tapes are added:

- one uses a unary alphabet and is used as a counter
- the second is a k-track "copy" tape
- the last is a k-track binary "position" tape

After calculating the block bound $t^{2/3}$, store it in unary on the counter tape. Now mark the work tapes into blocks of length $t^{2/3}$ by putting a special end-of-block marker where needed (each block will thus be one cell larger to accommodate the end of cell marker). Let $b_i(j)$ and $b'_i(j)$ denote the contents of the j^{th} block of tape i in the original machine and new machine respectively. Then $b'_i(j)$ is the tuple $(Reverse(b_i(j-1)), b_i(j), Reverse(b_i(j+1)))$. Here $Reverse$ means reversing the order of the tape cells. When simulating the computation we start by working on the middle track. If we see the end-of-block marker when going left, then we start using the first track instead and reverse the directions of each head move (on this track). Similarly, if we go off the end of the block while going right we switch to using the last track and again reverse the direction of head moves. Throughout we keep moving the head on the counter tape until it indicates that we have reached the end of the period.

At the end of the period we have to do some housekeeping. First we save the state, either within the state itself or on a special tape somewhere; either way this is only a constant amount of information. We store the head position within each block on the "position" tape as follows. Scan left along all work tapes not moving past the beginning of the block. For all work tapes not at the beginning of the block write a 1 in the corresponding track of the "position" tape, and for all others write a 0. Continue in this way until every head position is at the start of block for all the work tapes; the number of 1's on track i of the "position" tape will thus equal the head position, within the block, on work tape i. This takes one additional time period.

Consider Figure 2. At this point only the center block is up to date; the values for $b_i(j-1)$ and $b_i(j)$ in the left block are 'stale' since we may have modified them in our last time period. Similarly the values for $b_i(j)$ and $b_i(j+1)$ in the right block are also 'stale'. We update these stale values by scanning back and forth over all three cells and using the "copy" tape as a temporary area to store the contents of one block (all three tracks) at a time. Altogether this process requires about 6 time periods.

Finally we return the heads to their correct positions, but this time if they were working on the first track of the working block we place them in the corresponding cell in the previous block, and if they were in the last track then we place them in the next block. This may require an additional time period or two. Altogether this simulation of a single time period has cost us a little bit of extra space (more work tapes) and a constant factor increase of about 14 in the number of periods. ■

3.2 Proof of Theorem 4 (Pebble Theorem)

Denote by $R_d(p)$ the minimum number of edges in a di-graph of in-degree bound d which requires p pebbles (that is, at least one node in the graph needs p pebbles). We will first show that $R_d(p) = \Omega(p \log p)$ and then that this fact implies our theorem.

Consider a graph $G = (V, E)$ with the minimal possible number of edges requiring p pebbles. Let V_1 be the set of vertices which can be pebbled using at most $p/2$ pebbles. Let G_1 be the subgraph induced by V_1 having edge set $E_1 = E \cap (V_1 \times V_1)$. Let $V_2 = V - V_1$ and let G_2 be the subgraph induced by V_2 and having edge set $E_2 = E \cap (V_2 \times V_2)$. Let $A = E - E_1 - E_2$ be all edges between the two subgraphs. Notice that there cannot be any edge from V_2 to V_1 since then the vertex at the head of the edge (in V_1) would require at least as many pebbles as the vertex at its tail, which is more than $p/2$.

3.2.1 There exists $v \in V_2$ which requires $p/2 - d$ pebbles in G_2

Assume not. Then we show that we can pebble any node in G with fewer than p pebbles, contradicting the hypothesis regarding G. For any node $v \in V_2$ we have assumed that in G_2 we can pebble it with $< p/2 - d$ pebbles. Invoke the strategy to pebble v in the graph G_2 on the original graph G. Whenever we want to pebble a vertex $u \in G_2$ that has a parent in G_1, bring a pebble to the parent in G_1, using $\leq p/2$ pebbles. Since some u might have as many as d parents in G_1, we actually use as many as $d - 1 + p/2$ pebbles when pebbling the d^{th} parent in G_1 (using $d-1$ pebbles to cover the first $d-1$ parents and $p/2$ to do the actual pebbling). Once all the parents in G_1 are pebbled, pebble u and then lift the pebbles on all the vertices in G_1. Thus we can pebble our target v using at most $(p/2 - d) + (p/2 + d - 1) = p - 1$ pebbles. Since we can also pebble any $v \in V_1$ with at most $p/2 < p$ we can pebble any $v \in V$ with fewer than p pebbles, a contradiction to our original choice of G.

3.2.2 If v with in-degree k requires m pebbles then it has a parent needing $m - k$

We show that in general, for any $G = (V, E)$ where G is an acyclic di-graph, any node $v \in V$ requiring m pebbles, with an in-degree of k, has a parent u requiring at least $m - k + 1$ pebbles. Suppose to the contrary that each parent needs $m - k$ or fewer pebbles. Then simply bring a pebble to each of them in arbitrary order, each time removing all pebbles not on parents of v. When bringing a pebble to the i^{th} parent we have $i - 1$ pebbles covering the other parents and use at most $m - k$ pebbles for the pebbling itself. Thus at any time we use at most $m - k + i - 1$ pebbles. Over all parents then, the maximum number of pebbles that we use is $m - k + k - 1 = m - 1$ which may occur only when pebbling the k^{th} parent. Now, however, we have k pebbles on the parents of v and we can pebble v itself having used at most $m - 1$ pebbles, a contradiction.

3.2.3 There exists $u \in G_1$ which requires $p/2 - d$ pebbles (in both G and G_1)

Consider any $v \in G_2$. Repeatedly replace v by a parent in G_2 until you can go no further. Since G is acyclic this is guaranteed to stop. When it stops, since the new v requires, by virtue of being in G_2, more than $p/2 > 0$ pebbles, v has at least one parent in G_1 (and no parents in G_2). Let $m > p/2$ be the number of pebbles needed to pebble v and let k be the in-degree of v in the original graph G. By the above claim v has a parent which requires $m - k + 1 > p/2 - k + 1$ pebbles. Futhermore, since $k \leq d$ we see that v has a parent requiring at least $p/2 - d + 1$ pebbles. Therefore (as v has no parents in G_2) there is a vertex $u \in G_1$ requiring at least $p/2 - d + 1 > p/2 - d$ pebbles.

3.2.4 $|E_2| + |A| \geq R_d(p/2 - d) + \frac{p}{4d}$

Since G_2 requires at least $p/2 - d$ pebbles, $|E_2| \geq R_d(p/2 - d)$. If $|A| \geq \frac{p}{4} \geq \frac{p}{4d}$ then we are done.

Otherwise $|A| < \frac{p}{4}$. We will ignore $|A|$ and show that $|E_2| \geq R_d(p/2 - d) + \frac{p}{4d}$. Pebble each of the $< p/4$ vertices in V_1 with children in V_2 in succession. Independently pebbling each would require at most $p/2$ pebbles. By pebbling one at a time we can pebble them all using at most $p/4 - 1 + p/2 = 3p/4 - 1$ pebbles. When done we are left with less than $p/4$ pebbles on the graph, leaving more than $3p/4$ pebbles free. Since we know that there must exist a $v \in G_2$ requiring p pebbles in G then it must require at least $3p/4$ pebbles in G_2. Consider a vertex v with out-degree of 0 which requires $3p/4$ pebbles in G_2, and remove it. As proven in section 3.2.2 the resulting graph must still require at least $3p/4 - d$ pebbles. Repeating this process i times, we have a graph requiring $3p/4 - i \cdot d$ pebbles with at least i fewer edges. So for $i = \frac{p}{4d}$ times, we have a graph requiring at least $3p/4 - p/4 = p/2$ pebbles with $\frac{p}{4d}$ fewer edges. This subgraph will have at least $R_d(p/2) \geq R_d(p/2 - d)$ edges. Together with the $\frac{p}{4d}$ that we removed we see that $|E_2| + |A| \geq |E_2| \geq R_d(p/2 - d) + \frac{p}{4d}$ as required.

3.2.5 Putting it together

So $|E| = |E_1| + |E_2| + |A|$. By Section 3.2.3 we have $|E_1| \geq R_d(p/2 - d)$. By Section 3.2.4 we have $|E_2| + |A| \geq R_d(p/2 - d) + \frac{p}{4d}$. Therefore $R_d(p) = |E| \geq 2R_d(p/2 - d) + \frac{p}{4d}$, where the equality is due to the hypothesis that G has minimum number of edges among grapohs requiring p pebbles.

To solve the recurrence notice that

$$R_d(p) \geq 2R_d(\frac{p}{2} - d) + \frac{p}{4d} \geq 2R_d(\frac{p}{2} - 2d) + \frac{p}{4d}$$

For any i we get

$$R_d(\frac{p}{2^i} - 2d) \geq 2R_d(\frac{\frac{p}{2^i} - 2d}{2} - d) + \frac{\frac{p}{2^i} - 2d}{4d} = 2R_d(\frac{p}{2^{i+1}} - 2d) + \frac{\frac{p}{2^i} - 2d}{4d}$$

So

$$R_d(p-2d) \geq 2^i R_d(\frac{p}{2^i}-2d) + \sum_{j=0}^{i-1} 2^j \frac{\left(\frac{p}{2^j}-2d\right)}{4d} = 2^i R_d(\frac{p}{2^i}-2d) + \sum_{j=0}^{i-1} \frac{p-2^{j+1}d}{4d}$$

Setting $i = \log_2(p/2d)$ we get

$$\begin{aligned}
R_d(p) \geq R_d(p-2d) &\geq 2^i R_d(0) + \sum_{j=0}^{\log_2(p/2d)-1} \frac{p-2^{j+1}d}{4d} \\
&\geq \sum_{j=0}^{\log_2(p/2d)-1} \frac{p-2^{\log_2(p/2d)}d}{4d} \\
&= \log_2(p/2d)\frac{p-\frac{p}{2d}d}{4d} \\
&= \log_2(p/2d)\frac{p/2}{4d} \\
&= \Omega(p\log p)
\end{aligned}$$

So, for some constant $c > 0$, $R_d(p) \geq c \cdot p \log p$.

Now consider our original question of how many pebbles one needs to pebble a graph of n vertices. With $p = kn/\log n$ pebbles, we can certainly pebble all graphs with less than $c \cdot p \log p$ edges. Now,

$$\begin{aligned}
cp\log p &= c\frac{kn}{\log n}\log\frac{kn}{\log n} \\
&= c\frac{kn}{\log n}(\log k + \log n - \log\log n) \\
&> ckn\left(1 - \frac{\log\log n}{\log n}\right) \\
&> ckn/2
\end{aligned}$$

for all sufficiently large n such that $\frac{\log\log n}{\log n} < 1/2$. Letting $k = \frac{2d}{c}$, we can pebble all graphs with less than $\frac{cn}{2} \cdot \frac{2d}{c} = dn$ edges, using $p = kn/logn$ pebbles. Since any graph on n vertices, with in-degree bound of d, has less than dn edges, it may be pebbled by $O(n \log n)$ pebbles. ■