CS 4510 Automata and Complexity 4/20/2022

Lecture 21

Lecturer: Frederic Faulkner *Scribe(s):*

1 SPACE

We have now spent several lectures discussing which tasks can be done quickly. However, we know that time is not the only limited resource which a program consumes; we might also care about the memory required by the program. In the same way that computational time generalized to the number of *steps* taken by a Turing machine, memory generalizes to the number of *tape cells* used by the machine.

Definition 1 *We say that a Turing machine M is an $f(n)$-space Turing machine if M scans at most $f(n)$ tape cells on any input of length n.*

We say that a nondeterministic Turing machine N is an $f(n)$-space Turing machine if N scans at most $f(n)$ tape cells per computational branch on any input of length n.

Definition 2 $SPACE(f(n)) = \{L \mid L$ *is decided by some $f(n)$-space deterministic Turing machine*$\}$

$NSPACE(f(n)) = \{L \mid L$ *is decided by some $f(n)$-space nondeterministic Turing machine*$\}$

We can form the equivalents of P and NP for space, by looking at languages that need at most polynomial space. The classes are called, rather naturally, $PSPACE$ and $NPSPACE$:

Definition 3 $PSPACE = \cup_i SPACE(n^i)$ *is the set of languages decidable in deterministic polynomial space.*

$NPSPACE = \cup_i NSPACE(n^i)$ *is the set of languages decidable in nondeterministic polynomial space.*

If P is the set of tasks that require a reasonable amount of time to perform, then $PSPACE$ is the set of tasks which need a reasonable amount of memory. This naturally might cause one to wonder about the difference between the two. Can you have a program that runs quickly but needs an unreasonable amount of space? What about the reverse?

Here is another way of framing that question: suppose, on a homework problem, you were required to either use at most n^2 time (and unlimited space) or n^2 space (and unlimited time). Which would you prefer?

Notice that if you can only move the tape head n^2 times, you can only look at n^2 tape cells at most! So by limiting your time consumption you automatically limit your space as well. Thus we have the following theorem:

Theorem 4 $TIME(f(n)) \subseteq SPACE(f(n))$

In fact, this containment is strict, although a proof of that fact is beyond the scope of this course.

Corollary 5 $P \subseteq PSPACE$

We think that this containment is strict, but we cannot prove it. Here is one reason why we think polynomial space is better than polynomial time:

Theorem 6 $SAT \in PSPACE$

Recall that we have not found a polynomial time algorithm for SAT. However, the following algorithm is a polynomial space algorithm for SAT:

```
On input $\varphi$:

For each possible assignment of the variables of $\varphi$, evaluate $\varphi$ with that assignment,
and accept if TRUE. If every assignment has been tried, reject.
```

How much space does this use? Well, we first need to write down an assignment of the variables of φ. This uses O(n) space. Then we need to evaluate φ with that assignment. One way to do so is to re-write φ with each x_i replaced by TRUE or FALSE according to our current assignment. This again takes $O(n)$ space. Importantly, once we have tried out this assignment, it it fails, we can erase the whole thing and reuse the tape squares! So we just use $O(n)$ space overall.

This is one major difference between space and time: space is reusable! This is true in real life as well.

1.1 Bounding space by time

By showing $TIME(f(n) \subseteq SPACE(f(n))$, we can upper bound the memory usage of a Turing machine if we know its time usage. Now we investigate the reverse: if we know the space usage of a Turing machine, what can we say about its time usage?

Theorem 7 $SPACE(f(n)) \subseteq TIME(2^{O(f(n))})$

In other words, if a Turing machine uses $O(n)$ space, it uses at most $2^{O(n)}$ time. This is a much looser bound then we had for space given time. If we require that a program use a reasonable (i.e. polynomial) amount of time, we know it will also use a reasonable (i.e. polynomial) amount of memory; but the converse doesn't seem to be true! The best we can do is show that polynomial memory implies at most exponential time.

Proof. Let L be in $SPACE(f(n))$, and let M be a $O(f(n))$ space decider for L. Recall that a *configuration* of a Turing machine consists of (tape contents + tape head location + state). Now, if M ever repeats a configuration, it is stuck in an infinite loop. (Why?) But M is a decider and so is not allowed to get stuck forever. So it can never repeat a configuration while processing some input x.

So if we can bound the number of possible configurations for M on x, we can bound the runtime, since M must stop running before it runs out of unique configurations. So how many configurations of M are there?

- Current state: There are a constant c number of possible states, because the number of states of M does not depend on the input
- Tape head location: Since M uses $O(f(n))$ space, there are $O(f(n))$ possible cells for the tape head to be located over.
- Tape contents: For simplicity's sake, assume that M uses the binary tape alphabet. Each cell can contain at most 2 characters, so the total number of possibilities for the tape contents is (2 possibilities for cell 1) $\times$ (2 possibilities for cell 2) $\times$ (2 possibilities for cell 3) $\times$ ... $\times$ (2 possibilities for cell O(f(n)) = $2^{O(f(n))}$.

Then there are $c \times O(f(n)) \times 2^{O(f(n))} = O(2^{f(n)})$ possible configurations for M, so M runs in at most exponential time, completing the proof.

Corollary 8 *$PSPACE \subseteq EXPTIME$, where $EXPTIME = \cup_i TIME(2^{n^i})$*

Again, we suspect that this containment is strict, but we cannot prove it.

2 P vs. NP again

As we know, the relationship between P and NP is an open question. We suspect that nondeterminism is, in some sense, "cheating", in that it allows us to do things quickly that we cannot seem to do quickly with only deterministic choice. However, what about space? Surprisingly perhaps, non-determinism provides only a moderate advantage when it comes to memory usage.

Theorem 9 (Savitch) *If $L \in NSPACE(f(n))$, then $L \in PSPACE(f(n)^2)$.*

In other words, we can simulate an nondeterministic Turing machine by a deterministic with only a squared penalty in terms of space usage.

Proof. Let N be an $f(n)$ space NTM. We will construct M to simulate N while using at most $f(n)^2$ space.

One idea that will *not* work is to use the construction from Lecture 12 to simulate N. Recall that we had 3 tapes:

- Tape 1 held the input and was never changed
- Tape 2 held the current sequence of choices
- Tape 3 was used to simulate N deterministically by making the choices specified in tape 2.

How much space does this use? Tape 1 uses O(n) space. Tape 3 is just one branch of the computation of N. N uses at most $f(n)$ space, so tape 3 is $f(n)$. Everything seems fine so far. How much space does tape 2 use?

Well, tape 2 uses a tape cell for each choice N has to make. But we saw above that machines that use $f(n)$ space may use up to $2^{f(n)}$ steps, which could mean up to $2^{f(n)}$ choices to be stored on tape 2. So this simulation uses exponential space! But we are trying to use at most squared space, so this doesn't work.

2.1 Doing better

We can do better than this by using recursion to simulate N. Our idea is the following: we know that a machine with $f(n)$ space can take at most $2^{f(n)}$ steps. So we will check if N can get to the start configuration to the end configuration in $2^{f(n)}$ steps. We do this by picking a "middle step", and seeing if we can get from the start to the middle in $\frac{1}{2} \times 2^{f(n)}$ steps, and again from the middle to the end in the same number of steps. We then use this idea recursively.

We define the CanYield subroutine to check if machine N can get from configuration c_1 to configuration c_2 in t steps:

```
CanYield($c_1$, $c_2$, $t$):
    if $t = 1$ and $c_2$ follows from $c_1$ by a transition rule of $N$ or $c_1 = c_2$: return True
    if $t > 1$:
        for each possible configuration $c_m$ of $N$:
            $r_1$ =CanYield($c_1$, $c_m$, $\frac{t}{2}$)
            $r_2$ =CanYield($c_m$, $c_2$, $\frac{t}{2}$)
            if $r_1$ and $r_2$: return True
    return False
```

Then, here is the deterministic machine M that simulates N:

```
$M$ on input $x$:
1. if CanYield($c_{start}$, $c_{end}$, $2^{f(n)}$), accept; else, reject
```

Some notes: what exactly is the "accepting configuration"? There could be many possibilities for the contents of the tape by the time N accepts. Therefore, to standardize things, we modify N to clear its own tape and move its head all the way to the left before accepting. Thus we now have a single accepting configuration.

Also, from a time complexity perspective, this algorithm is laughably inefficient. We may end up trying every one of the exponentially many possibilities for the "middle step". Not only that, but we may spend exponentially long exploring some particular "middle step" before we realize it doesn't work.

Fortunately, we only care about space usage and not time. What is the space complexity of M?

Well, each time we make a recursive call, we have to store the current values of c_1, c_2, and t on the tape, so that we can pick up where we left off. But a configuration is primarily a snapshot of tape contents of N along with a single cell for the tape head and state, so we need $O(f(n))$ tape cells each time we make a recursive call.

Recall that we can reuse space if we are done using it. So how many calls might we need to store simultaneously? This is equivalent to asking the depth of the recursive tree. At each step, we cut t in half, so the tree will have depth $logt$. But we start with $t = 2^{f(n)}$, so the recursion depth is $f(n)$.

$f(n)$ recursive calls needing $O(f(n))$ tape cells each is $O(f^2(n))$ tape cells needed overall, completing the proof.

Corollary 10 $PSPACE = NPSPACE$.

Corollary 11 $NP \subseteq PSPACE$

The last corollary follows from the fact that $NP \subseteq NPSPACE$ (even a nondeterministic machine can't use more that n^2 tape cells in n^2 time.)

So we have that

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

.

In fact, we believe that all containments are proper, but so far the best we can do is that $P \subset EXPTIME$!

2.1.1 A small detail regarding big-O

For ease of comprehension, the proof above ignores some small details. In particular, we are not guaranteed that N uses $f(n)$ space but rather $O(f(n))$ space. From the definition of big-O, that still means that there is some constant d such that the space usage of N is bounded above by $d * f(n)$, so we can treat N as a $d * f(n)$-space Turing machine.

3 SPACE, cont.

We will finish our discussion of space complexity by introducing two new classes: Ł and NL. We use the symbol Ł to avoid confusion with the use of L to stand for any arbitrary language. Ł and NL are the deterministic and nondeterministic space classes for languages that can be decided using logarithmic space.

Definition 12 Ł $= SPACE(log(n))$

Definition 13 $NL = NSPACE(log(n))$

A question that arises almost immediately is, "How can a machine use only $log(n)$ space if writing the input itself on the tape already takes up $O(n)$ space?" The answer is that it doesn't seem quite fair to penalize the machine for "using" the tape squares of the input. So instead, when talking about sub-linear space, we use a different type of Turing machine. Instead of a single tape Turing machine, we use a two tape Turing machine where:

- One tape is the read-only input tape. The TM cannot make any changes to the contents of this tape whatsoever, including marking characters, overwriting them, etc.
- A second work tape, which begins blank and which the Turing machine can read from and write to.

On the cells used in the work tape count for the space complexity of this model of Turing machine.

3.1 An overview of L and NL

What can you do with just logarithmic space? You cannot copy the entire input to the work tape, because it is too long. In general, we think of the work tape as being able to store a finite number of *pointers*. If the input contains n variables of some type (n vertices or n SAT-phrases or n numbers or...), then you could refer to them by the numbers 1 through n. Storing a number of size $\leq n$ in binary on a tape requires $O(log(n))$ tape cells. (For example, writing 35 in binary doesn't require 35 tape cells but rather 5, to store "10011".) So we can think of a log space tape as allowing a Turing machine to think about or remember a fixed, finite number of the variables contained in the input. Logarithmic space is thus also enough space to store a counter that counts up to n, so $0^k 1^k \in$ Ł.

Recall that

$$PATH = \{\langle G\rangle, s, t \mid G \text{ contains a path from } s \text{ to } t\}$$

If we use nondeterminism, how many vertices do we need to store on the work tape at one time to determine if there is a path from s to t? Just one! At each step in the algorithm, we can guess which vertex comes next in the path from s to t. Since we always guess correctly (if possible), once we write down the new vertex we can get rid of the previous one. Since we only need to store a fixed number of vertices, we have that $PATH \in NL$.

Although we will not prove it here, we actually have that $PATH$ is complete for NL. On the other hand, we have not found a *deterministic* log-space algorithm for PATH, and so we suspect that $PATH \notin$ Ł and thus that Ł $\neq NL$. However, just like for P vs. NP, we do not yet know how to prove this fact.