Recitation 9: Space Complexity

In this recitation, we will:

- Review the definitions of space complexity and space complexity classes such as PSPACE and NPSPACE.
- Review how space complexity is related to time complexity.
- Use an example to practice analyzing space complexity.
- Motivate using polynomial-time reductions in the definition of PSPACE-hardness.
- Use an example to illustrate the relationship between PSPACE-hardness and NP- and coNP-hardness (the former implies the latter).

Some recitation sections reviewed the proof that $TQBF$ is PSPACE-complete. Since the proof is described in detail in the textbook (Theorem 8.9, pp. 339–341), we'll omit it in the notes here.

Space Complexity Classes

Let's begin by reviewing the definitions of space complexity for deterministic and nondeterministic TM deciders.

Definition 1. *For $f : \mathbf{N} \to \mathbf{N}$ with $f(n) \geq n$,[1] we say a deterministic TM decider M runs in $f(n)$ space if the maximum number of different tape cells M scans on any input of length n is $f(n)$.*

The space complexity class $\mathsf{SPACE}(f(n))$ *is defined to be all languages A for which there is a deterministic TM that decides A in $O(f(n))$ space.*[2]

Definition 2. *For $f : \mathbf{N} \to \mathbf{N}$ with $f(n) \geq n$, we say a nondeterministic TM decider N runs in $f(n)$ space if the maximum number of different tape cells N scans on any branch of its computation on any input of length n is $f(n)$.*

The space complexity class $\mathsf{NSPACE}(f(n))$ *is defined to be all languages A for which there is a nondeterministic TM that decides A in $O(f(n))$ space.*

Similar to polynomial-time complexity classes P and NP, we can define polynomial-space complexity classes PSPACE and NPSPACE:

[1] At this point we only define space complexity for $f(n) \geq n$. Sublinear space complexities will be defined next week using a model of computation that is based on but slightly different from the standard Turing machine. The reason is that we want to allow the TM to at least be able to read the entire input, and the current definition does not allow that when $f(n) < n$.

[2] Note that there's a big-O in the definition here.

Definition 3. PSPACE *is the class of all languages decided by a polynomial-space deterministic TM, i.e.,*

$$\mathsf{PSPACE} := \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k). \tag{1}$$

Definition 4. NPSPACE *is the class of all languages decided by a polynomial-space nondeterministic TM, i.e.,*

$$\mathsf{NPSPACE} := \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k). \tag{2}$$

Note that, while $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ is an open problem, it is known that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. This follows from Savitch's theorem, which states that simulating nondeterminism can be done with at most quadratic overhead in space. In other words, an $f(n)$-space nondeterministic TM always has an $[f(n)]^2$-space deterministic TM that decides the same language.

Relationship between space complexity and time complexity

Given that a TM runs in space $f(n)$, what can we know about its time complexity $t(n)$? Intuitively, the running time can't be too short—otherwise the TM doesn't even have enough time to reach $f(n)$ tape cells. On the other hand, since there are finitely many configurations that use at most $f(n)$ space, the TM can't take too much time or else it must repeat a configuration and the TM will loop. This is formalized in the following theorem.

Theorem 1. *If a TM runs in space $f(n)$ ($f(n) \geq n$), then its time complexity $t(n)$ must satisfy*

$$t(n) \geq f(n) \tag{3}$$

and

$$t(n) = 2^{O(f(n))}. \tag{4}$$

Proof. Equation (3)*:* Since a TM can scan at most as many tape cells as the number of steps it took, $t(n) \geq f(n)$.
Equation (4)*:* There are $N = |Q|f(n)|\Gamma|^{f(n)}$ TM configurations that have non-blank symbols only on the leftmost $f(n)$ tape cells. Thus, if a TM that uses $f(n)$ space takes more than N time steps on an input of length n, some configuration must have been repeated and the TM loops, a contradiction. Thus,

$$t(n) \leq |Q|f(n)|\Gamma|^{f(n)} = 2^{\lg|Q| + \lg f(n) + f(n)\lg|\Gamma|} = 2^{O(f(n))}. \tag{5}$$

□

Try to convince yourself that Theorem 1 also holds for nondeterministic TMs with essentially the same proof.

Due to Theorem 1, we get that a polynomial-space algorithm runs in at least polynomial time and at most exponential time.[3] Similarly, a nondeterministic polynomial-space algorithm runs in at least nondeterministic polynomial time. As a result,

$$\mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME} \quad \text{and} \quad \mathsf{NP} \subseteq \mathsf{NPSPACE}. \tag{6}$$

Furthermore, recall that $\mathsf{P} \subseteq \mathsf{NP}, \mathsf{coNP}$ and $\mathsf{PSPACE} = \mathsf{NPSPACE}$. Combining all these results together, we get

$$\mathsf{P} \subseteq \mathsf{NP}, \mathsf{coNP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXPTIME}. \tag{7}$$

Note that we get $\mathsf{coNP} \subseteq \mathsf{PSPACE}$ from the fact that $\mathsf{NP} \subseteq \mathsf{PSPACE}$ and that PSPACE is a deterministic complexity class and thus closed under complement (i.e., $\mathsf{PSPACE} = \mathsf{coPSPACE}$).

The above relationships are summarized in the following diagram:

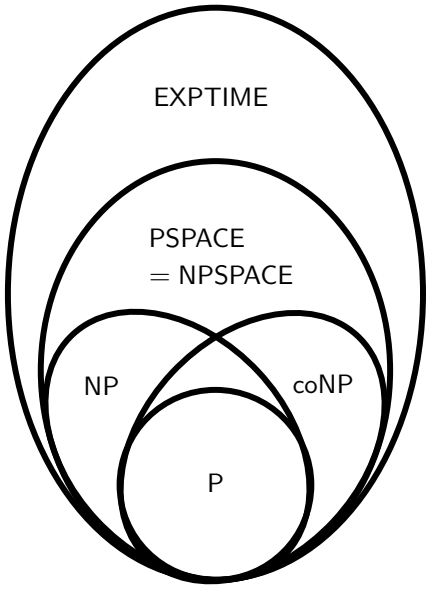

Figure 1: Relationships between time and space complexity classes.

[3] *Exponential time* refers to $2^{O(n^k)}$ for some k.

Space complexity analysis

Let's use an example to gain intuition for analyzing space complexity.

Example 1. *Let*

$$MINFORMULA = \{\langle \phi \rangle \mid \textit{boolean formula } \phi \textit{ has no smaller equivalent formula}\}. \tag{8}$$

Here, we say two formulas are equivalent if they evaluate to the same truth value on all variable assignments. We say a formula ϕ' is smaller than ϕ if the string representation of ϕ' is shorter than that of ϕ in whatever encoding we're using, i.e., $|\langle\phi'\rangle| < |\langle\phi\rangle|$.

Show that $MINFORMULA \in$ PSPACE.

Proof.
Approach 1 (Direct TM construction):

The following TM T decides $MINFORMULA$ in polynomial space using a brute-force approach:

$T =$ "On input $\langle\phi\rangle$,

1. Iterate through every formula ϕ' smaller than ϕ. For each such ϕ',
   (a) Check whether ϕ' is equivalent to ϕ by iterating through every variable assignment and checking that they always agree in truth value.
   (b) If ϕ' is equivalent to ϕ, then *reject*.
2. If no smaller equivalent formula was found, then *accept*."

Analysis: At any given moment, the tape contains $\langle\phi\rangle$, $\langle\phi'\rangle$, and the variable assignment to them. Each of these three components uses space linear in the size of $\langle\phi\rangle$, so the total space complexity is $O(n)$.
Commentary: Note that the algorithm we gave is very brute-force, which we often cannot afford to do if we want a polynomial-*time* algorithm. But space gives us a lot more freedom, allowing us to reuse the same space to try a number of things exponential in the amount of that space (this should remind you of Theorem 1!).

Approach 2 (Using NP $\subseteq$ PSPACE $=$ NPSPACE*):*

Note that deciding whether two given formulas ϕ, ϕ' are *in*equivalent is in NP since the certificate is the variable assignment on which they differ. Since NP $\subseteq$ PSPACE, there is a deterministic polynomial-space TM M for it.

Now, the following *non*-deterministic TM N decides $\overline{MINFORMULA}$ in polynomial space:

$N =$ "On $\langle\phi\rangle$,

1. Guess a smaller formula ϕ' and check whether it's *in*equivalent to ϕ using M. If not (i.e., they're equivalent), then *accept*."

This shows that $\overline{MINFORMULA} \in$ NPSPACE. But NPSPACE $=$ PSPACE, and PSPACE is closed under complement, so $MINFORMULA \in$ PSPACE, as desired.
Commentary: Because of Savitch's theorem, we have more freedom to use nondeterminism since it can be simulated by a deterministic procedure with at most quadratic overhead in space. As a result, if it

doesn't take too much time to "check if something with some property exists" by guessing that "something" nondeterministically, then we can also do the same check deterministically without using too much space. (In our case, that "something" is a variable assignment on which ϕ and ϕ' differ, or a ϕ' smaller than ϕ that's equivalent.) And an explicit construction for conducting this deterministic check was given in *Approach 1* above, i.e., you iterate through all possible things and checking whether they have the property in question. □

A question you may have about *Approach 1* above is: how exactly do you iterate through all smaller formulas ϕ' (step 1), and how exactly do you iterate through all variable assignments (step 1(a))? One approach is *recursion*.[4] We'll describe the recursive algorithm in more detail, since the kind of space complexity analysis done here applies generally to recursive algorithms. Let's illustrate with step 1(a), which tries all variable assignments to check ϕ and ϕ' agree on all of them.

[4] In practice, recursion can result in a lot of overhead (as illustrated below), so an equivalent iterative algorithm may be preferred. How would you iterate through, e.g., all possible variable assignments iteratively? *Hint:* How do you count in binary from 0 up to $2^k - 1$?

The recursive algorithm looks like:

$f =$ "On input $\langle \phi, \phi', x_1, \ldots, x_i \rangle$ where $0 \leq i \leq k$ (k is the number of variables) and $x_1, \ldots, x_i$ is a partial assignment to the variables,

1. If $i = k$, then we've finished assigning all variables, so we return whether $\phi(x_1, \ldots, x_k) = \phi'(x_1, \ldots, x_k)$.
2. Set $x_{i+1} = 0$. Recursively call f on $\langle \phi, \phi', x_1, \ldots, x_{i+1} \rangle$.
3. Set $x_{i+1} = 1$. Recursively call f on $\langle \phi, \phi', x_1, \ldots, x_{i+1} \rangle$.
4. *Accept* if both recursive calls in step 2 and step 3 accept. Otherwise, *reject*."

At any point during the execution of a recursive algorithm, the memory consists of multiple *stack frames* in a *stack*. The stack frame at the bottom of the stack consists of local variables by the top-level function call to f ($i = 0$); when that function call recursively calls f in step 2 or step 3,[5] a new stack frame is created and pushed onto the stack, which contains the local variables to that recursive call; and then when that function call further recursively calls f, yet another stack frame is created and pushed onto the stack; so on and so forth, until we reach the base case of the recursion. As a result, the number of stack frames is equal to the depth of the recursion, which in our case is $k + 1$. The space complexity is thus

[5] When a function call returns, its stack frame is destroyed, so when step 2's function call returns, the space that the step 2 function call was using can now be reused for the stack frame of step 3's function call.

$$\begin{aligned}&(\text{recursion depth}) \times (\text{space taken up by each stack frame})\\ &\qquad = (k+1) \times O(n) = O(n^2). \quad (9)\end{aligned}$$

We've used the fact that each stack frame contains the local variables $\phi, \phi', x_1, \ldots, x_{i+1}$, which take up $O(n)$ space.

So we've shown that the above recursive algorithm takes polynomial space. But before we said it would take $O(n)$ space, and now we're taking $O(n^2)$ space! The reason is that we're being quite space-inefficient here. Every time we make a recursive call, we're copying $\phi, \phi', x_1, \ldots, x_i, x_{i+1}$ into the new stack frame, which is an unnecessary waste of space. Instead, we can keep all of them as global variables that all recursive calls have access to, so that at any given point the tape has the global variables $\phi, \phi', x_1, \ldots, x_i$ ($O(n)$ space) and the $O(k)$ stack frames created so far, each of which now occupies only $O(1)$ memory, and we recover the $O(n) + O(k) = O(n)$ space complexity.

Trying all possible formulas shorter than ϕ can be done recursively as well, where we use the same recursive algorithm to try all possible strings of length less than $|\langle\phi\rangle|$ and ignore those that are not syntactically valid formulas.

Why polynomial-time reductions for defining PSPACE*-hardness?*

Recall the definition of PSPACE-hardness:

Definition 5. *A language A is* PSPACE*-hard if every language $B \in$ PSPACE is polynomial-time mapping-reducible to it, i.e., $B \leq_{\mathrm{p}} A$ $\forall B \in$ PSPACE.*

You might think it's more natural to use polynomial-*space* reductions to define PSPACE-hardness, but in fact *allowing the reduction to be as powerful as the class for which hardness is being defined defeats the purpose of defining hardness*. By that we mean that, since a polynomial-space computable function can decide polynomial-space languages, using polynomial-space reductions renders every non-trivial[6] language PSPACE-hard since the reduction is powerful enough to decide B, and we failed to formalize the notion of "the hardest problems in PSPACE".

[6] Here we'll consider any language other than $\emptyset$ and Σ^* to be non-trivial.

Lemma 1. *If* PSPACE*-hardness were defined relative to polynomial-space reductions, then* every *language A other than $\emptyset, \Sigma^*$ is* PSPACE*-hard.*

Proof. Take any $A \neq \emptyset, \Sigma^*$. Suppose $x \in A$ and $y \notin A$.

For any $B \in$ PSPACE, we have the following polynomial-space mapping reduction to A:

$$f(w) = \begin{cases} x & \text{if } w \in B \\ y & \text{if } w \notin B \end{cases}. \tag{10}$$

Since $B \in$ PSPACE, checking whether $w \in B$ can be done in polynomial space, so this reduction runs in polynomial space. □

This should remind you of two pset problems where you saw essentially the same thing:

- pset3 q2(b), where every non-trivial language is hard for the class of decidable languages under mapping reductions, since computable functions can decide decidable languages;

- pset4 q3, where every non-trivial language is P-hard under polynomial-time reductions, since polynomial-time computable functions can decide polynomial-time decidable languages.

The takeaway is that, when choosing the type of reduction for defining hardness for a complexity class, we don't want to allow the reduction to be so powerful that it can just decide all problems in that complexity class.

While this gives us an upper bound on how powerful a reduction should be allowed to be, what are some other considerations when choosing the type of reduction used to define hardness?

One thing we can say is that the reduction can't be too restricted, since otherwise we can't do transformations that we would normally consider "easy" transformations. A reasonable definition of "easy" in practice is "polynomial-time", so we can just go with that.

Another important consideration is the implications of our definition of PSPACE-hardness. For example, the reason the following theorem holds is because we had used polynomial-time reductions in the definition.

Theorem 2. *If there's a* PSPACE*-hard problem A that is in* P*, then* P $=$ PSPACE*.*

Proof. For every $B \in \mathsf{PSPACE}$, decide B in polynomial-time by transforming its input using the polynomial-*time* reduction to A and using the polynomial-time decider for A. □

Note that the resultant algorithm runs in polynomial time exactly because the reduction was required to run in polynomial time by definition. If the reduction was allowed to be more powerful (e.g., polynomial-space), then the algorithm no longer works and we would've failed to show that $B \in \mathsf{P}$.

Since the question $\mathsf{P} \stackrel{?}{=} \mathsf{PSPACE}$ is an important open problem, Theorem 2 is significant since it tells us that, to show $\mathsf{P} = \mathsf{PSPACE}$, it suffices to find just one PSPACE-hard problem decidable in polynomial time. It is thus desirable to define PSPACE-hardness relative to the kind of reduction that would allow us to have Theorem 2, and polynomial-time reductions do the job.

Another nice consequence of the definition that uses polynomial-time reductions is that Theorem 3 in the next section would hold, so we successfully capture the intuition that the hardest questions in PSPACE should be at least as hard as the hardest questions in NP since

NP $\subseteq$ PSPACE. We don't get this nice result if PSPACE-hardness were defined using a more powerful kind of reduction than that used to define NP-hardness.

PSPACE-*hardness implies* NP-*hardness and* coNP-*hardness*

As an example, let's show that the PSPACE-complete language *TQBF* is NP-hard.

Example 2. *TQBF is* NP*-hard.*

Proof. Since every PSPACE language is polynomial-time reducible to *TQBF* and NP $\subseteq$ PSPACE, this implies that every NP language is polynomial-time reducible to *TQBF*, i.e., *TQBF* is NP-hard.

(As an exercise with reductions, try to give an alternative proof using a reduction from *SAT*.) □

It is left as an exercise to the reader to show the following more general statement:

Theorem 3. *Every* PSPACE*-hard language is both* NP*-hard and* coNP*-hard.*

This captures the intuition that the hardest languages in PSPACE are at least as hard as the hardest languages in NP and coNP, which one might expect from the fact that NP, coNP $\subseteq$ PSPACE.