Space Complexity and Interactive Proof Systems

Sections 8.0, 8.1, 8.2, 8.3, 10.4

Based on Powerpoint slides by Giorgi Japaridze, Villanova University

Space complexity defined

Definition 8.1 Let **M** be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of **M** is the function **f: N→N**, where **f(n)** is the maximum number of tape cells that **M** scans on any input of length **n**. If the space complexity of **M** is **f(n)**, we also say that **M** runs in space **f(n)**.

If **M** is a nondeterministic TM wherein all branches halt on all inputs, we define its space complexity **f(n)** to be the maximum number of tape cells that **M** scans on any branch of its computation for any input of length **n**.

SPACE and NSPACE

Definition 8.2 Let **f: N→R$^+$** be a function. The space complexity classes ***SPACE(f(n))*** and ***NSPACE(f(n))*** are defined as follows.

SPACE(f(n)) = {**L** | **L** is a language decided by an **O(f(n))** space deterministic Turing machine}

NSPACE(f(n)) = {**L** | **L** is a language decided by an **O(f(n))** space nondeterministic Turing machine}.

Space is more powerful than time

Example 8.3. Space is more powerful than time, because it can be reused. E.g., while we believe that SAT has no polynomial (let alone linear) time algorithm, it can be easily decided in linear space:

```
M_1 = "On input <$\phi$>, where $\phi$ is a Boolean formula:
        1. For each truth assignment to the variables $x_1,...,x_m$ of $\phi$:
        2.      Evaluate $\phi$ on that truth assignment.
        3. If $\phi$ ever evaluated to 1, accept; if not, reject."
```

Each iteration of the loop needs extra memory only for remembering the current truth assignment, and for evaluating ϕ on that assignment. This takes **O(n)** space. This space can then be recycled during the next iteration. Thus, the overall space complexity here remains linear.

What is the time complexity of this algorithm, by the way?

Nondeterministic space is not much more powerful than deterministic space

Theorem 8.5 (Savitch's Theorem)

For any function **$f: N \rightarrow R^+$**, where **$f(n) \geq n$**, we have

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f^2(n)).$$

As the proof of this theorem reveals, a deterministic TM can simulate a nondeterministic TM using only a little amount of extra space.

That is due to the fact that space can be recycled. As time cannot be recycled, the same trick fails to work with time (otherwise we would have a proof of P=NP).

Proof of Savitch's theorem (i)

Proof. Fix an arbitrary NTM **N**.

Let **t** be a positive integer, and let c_1 and c_2 be two configurations of **N**. We cay that c_1 ***can yield*** c_2 in $\leq t$ steps if **N** can go from c_1 to c_2 in **t** or fewer steps. The following is a deterministic recursive algorithm deciding the "can yield" problem when **t** is a power of **2** ($t=2^p$ for some $p\geq 0$).

```
CANYIELD($c_1$, $c_2$, $2^p$) = "On input $c_1$, $c_2$ and $p$, where $p\geq 0$ and $c_1$,$c_2$
are configurations that use at most $f(n)$ space (i.e. in which the head is
at a $\leq f(n)$th cell, and all non-blank cells are among the first $f(n)$ cells):
    1. If $p=0$, then test if $c_1=c_2$ or $c_1$ yields $c_2$ in one step according to the
       rules of N. Accept if either test succeeds; reject if both fail.
    2. If $p>0$, then for each configuration $c_m$ of N that uses space $\leq f(n)$:
    3.     Run CANYIELD($c_1$, $c_m$, $2^{p-1}$).
    4.     Run CANYIELD($c_m$, $c_2$, $2^{p-1}$).
    5.     If steps 3 and 4 both accept, then accept.
    6. If haven't yet accepted, reject."
```

Proof of Savitch's theorem (ii)

We assume that (or otherwise modify **N** so that) **N** clears its tape before halting and goes to the beginning of the tape, thereby entering a (fixed) configuration called $\mathbf{c_{accept}}$. And we let $\mathbf{c_{start}}$ be the start configuration of **N** on input **w**.

Next, where **n** is the length of **w**, we select a constant **d** so that **N** has no more than $\mathbf{2^{df(n)}}$ configurations that use **f(n)** cells of the tape. Then $\mathbf{2^{df(n)}}$ provides an upper bound on the running time of **N** on **w** (for otherwise a configuration would repeat and thus **N** would go into an infinite loop, contrary to our assumption that this does not happen).

Hence, **N** accepts **w** if and only if it can get from $\mathbf{c_{start}}$ to $\mathbf{c_{accept}}$ within $\mathbf{2^{df(n)}}$ or fewer steps. So, the following (deterministic) machine **M** obviously simulates **N**, i.e. **M** accepts **w** iff **N** does:

M = "On input **w**:

1. Output the result of **CANYIELD**$\mathbf{(c_{start}, c_{accept}, 2^{df(n)})}$."

Proof of Savitch's theorem (iii)

It remains to analyze the space complexity of **M**.

Whenever **CANYIELD** invokes itself recursively, it stores the current stage number and the values of c_1, c_2 and p on a stack so that these values can be restored upon return from the recursive call. Each level of the recursion thus uses $O(f(n))$ additional space.

Next, each level of the recursion decreases p by 1. And, as initially p starts out equal to $df(n)$, the depth of the recursion is $df(n)$. Therefore the total space used is $df(n) \times O(f(n)) = O(f^2(n))$, as promised.

We are done. Or are we?

One technical difficulty arises in this argument because **M** needs to know the (**w**-depending) value of $f(n)$ when it calls **CANYIELD**. Not to worry, we can handle that.

Proof of Savitch's theorem (iv)

A way to overcome the difficulty is to modify **M** --- call the modified version **M'** --- so that it tries **$f(n) = 1,2,3,...$**

For each value **$f(n)=i$**, **M'** uses **CANYIELD** to determine whether the accept configuration is reachable, and *accepts* if yes. Thus, if **N** accepts **w** within **$\leq 2^j$** steps, **M'** will also accept **w** when trying **$f(n)=i$** for a certain **$i \leq j$**.

Next, to guarantee that **M'** does not keep increasing **i** (and thus the used space) indefinitely in the cases when **N** does not accept **w**, **M'** does the following before passing from **i** to **i+1**. Using **CANYIELD**, it checks whether any configuration (at all) utilizing **i+1** cells of the tape is reachable from **c_{start}**. If not, there is no point in trying **i+1** and greater **i**'s, and **M'** *rejects*.

Obviously the greatest possible **i** that **M'** will have to try before halting is **$f(n)$**. So, remembering the current **i** (which is necessary for **M'** to work properly) will only take **$O(\log f(n))$** space. And, while trying an **i**, **M'** uses no more additional space than **M** does. This space can be recycled on every iteration, so the space complexity of **M'** remains **$O(f^2(n))$** .

PSPACE defined

Definition 8.6 **PSPACE** is the class of languages that are decidable in polynomial space on a deterministic TM. In other words,

$$\mathbf{PSPACE} = \mathbf{SPACE}(n) \cup \mathbf{SPACE}(n^2) \cup \mathbf{SPACE}(n^3) \cup \ldots$$

NPSPACE can be defined similarly. However, the latter is not a very interesting class because, as an immediate corollary of Savitch's theorem, it coincides with **PSPACE** (squaring polynomial space again yields polynomial space).

This is what we know (why?):

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}{=}\mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}.$$

We, however, do not know whether any of the three $\subseteq$s can be replaced by $=$. Another set of huge open problems! It can be proven however that

$$\mathbf{P} {\neq} \mathbf{EXPTIME}.$$

So, at least one of the three containments must be proper ($\subseteq$ but not $=$), even though we do not know which one(s)!

PSPACE-completeness defined

Definition 8.8 A language **B** is ***PSPACE-complete*** iff it satisfies two conditions:

1. **B** is in PSPACE, and
2. every **A** in PSPACE is polynomial time reducible to **B**.

If **B** merely satisfies condition 2, we say that it is ***PSPACE-hard***.

Why do we still appeal to polynomial time reducibility and not, say, polynomial space reducibility, philosophically speaking?

A reduction must be *easy* relative to the class (of difficult problems) that we are defining. Only then it is the case that if we find an easy way to solve a (PSPACE-, NP- or whatever-) complete problem, easy solutions to other (reducible to it) problems would also be found. If the reduction itself is hard, it does not at all offer an easy way to solve problems.

The TQBF problem

Universal quantifier **$\forall$**: **$\forall x P(x)$** means "for any **$x \in \{0,1\}$**, **$P(x)$** is true"

Existential quantifier **$\exists$**: **$\exists x P(x)$** means "for some **$x \in \{0,1\}$**, **$P(x)$** is true"

We consider ***fully quantified Boolean formulas*** (in the *prenex* form). These are Boolean formulas prefixed with either **$\forall x$** or **$\exists x$** for each variable **x**.

Examples (true or false?):

$\forall x(x \vee -x)$

$\exists x(x \vee -x)$

$\exists x(x \wedge -x)$

$\exists x \exists y(x \wedge y)$

$\forall x \forall y\ (x \vee y)$

$\forall x \exists y\ ((x \wedge y) \vee (-x \wedge -y))$

$\exists x \forall y\ ((x \wedge y) \vee (-x \wedge -y))$

$\exists z \forall x \exists y\ ((x \wedge y \wedge z) \vee (-x \wedge -y \wedge z))$

TQBF = {<ϕ> | ϕ is a true fully quantified Boolean formula}

(**T**rue **Q**uantified **B**oolean **F**ormulas)

The PSPACE-completeness of TQBF – proof idea

Theorem 8.9 TQBF is PSPACE-complete.

Proof idea. To show that **TQBF∈PSPACE**, we give an algorithm that assigns values to the variables and recursively evaluates the truth of the formula for those values.

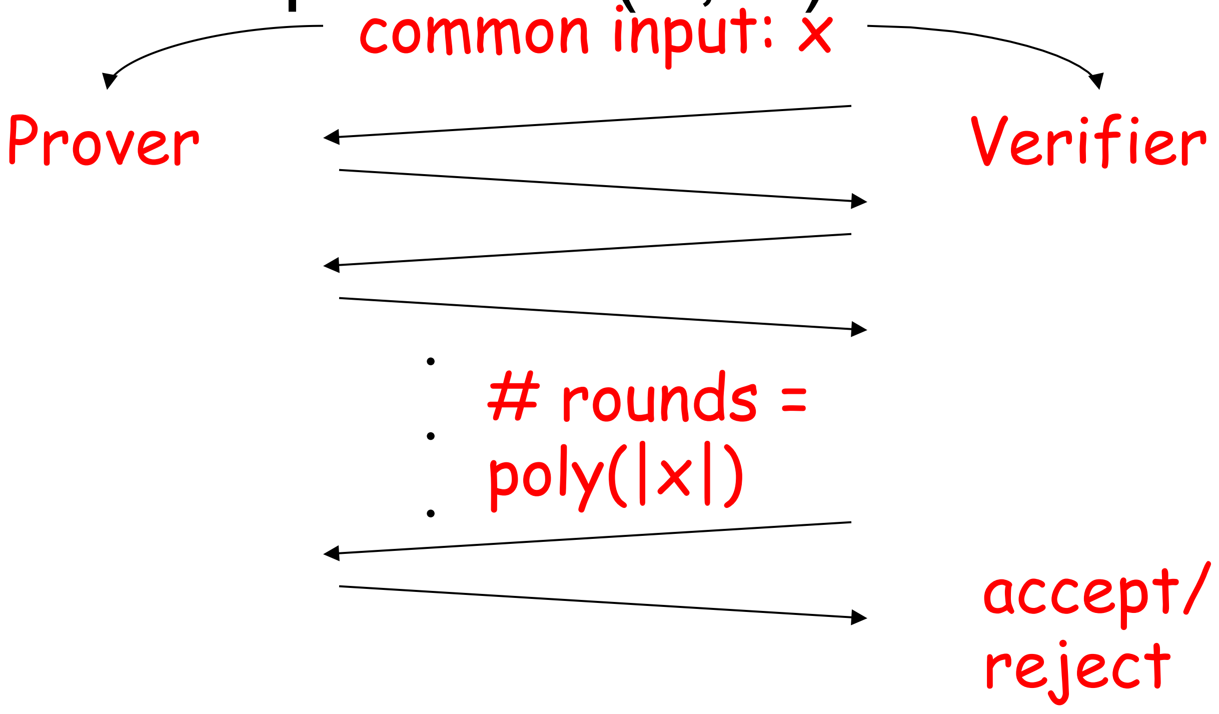
To show that **$A\leq_p$TQBF** for every **A∈PSPACE**, we begin with a polynomial-space machine **M** for **A**. Then we give a polynomial time reduction that maps a string **w** to a formula **ϕ** that encodes a simulation of **M** on input **w**. **ϕ** is true iff **M** accepts **w** (and hence iff **w∈A**).

A first, naive, attempt to do so could be trying to precisely imitate the proof of the Cook-Levin theorem. We can indeed construct a **ϕ** that simulates **M** on input **w** by expressing the requirements for an accepting tableau. As in the proof of the Cook-Levin theorem, such a tableau has polynomial width **$O(n^k)$**, the space used by **M**. But the problem is that the height of the tableau would be exponential!

Instead, we use a technique related to the proof of Savitch's theorem to construct the formula. The formula divides the tableau into halves and employs the universal quantifier to represent each half with the same part of the formula. The result is a much shorter formula. **End of proof idea**

Interactive Proofs

- **interactive proof system** for L is an interactive protocol (P, V)



Interactive Proof Systems

Interactive proof systems provide a way to define a probabilistic analog of the class NP, much as probabilistic polynomial time algorithms provide a probabilistic analog of P.

Remember the formulation of NP in terms of polynomial time verifiability. Let us rephrase it by creating two entities: a Prover that finds the proofs of membership and a Verifier that checks them. Think of the Prover as if it were *convincing* the Verifier of **w**' s membership in **A**. We require the verifier to be a polynomial time bounded machine; otherwise it could figure out the answer itself. We don' t impose any computational bounds on the Prover because finding the proof may be time-consuming.

Take the SAT problem for example. A Prover can convince a polynomial time Verifier that a a formula is satisfiable by supplying a satisfying assignment. Can a Prover similarly convince a computationally limited Verifier that a formula is *not* satisfiable? The answer, surprisingly, is yes, provided we give the Prover and Verifier two additional features. First, they are permitted to engage in a *two-way* dialog. Second, the Verifier may be a *probabilistic* polynomial time machine that reaches the correct answer with a high degree of, but not absolute, certainty.

Such a Prover and Verifier constitute an interactive proof system.

Graph nonisomorphism

Convincing the Verifier that two graphs are isomorphic is easy: just present the isomorphism. But how could the Prover convince the Verifier that two graphs G_1,G_2 are *not* isomorphic? Well, it can! Consider the following protocol.

The Verifier randomly selects either G_1 or G_2 and then randomly reorders its nodes to obtain a new graph **H**. The Verifier sends **H** to the Prover. The Prover must respond by declaring whether G_1 or G_2 was the source of **H**. This concludes the protocol.

If G_1 and G_2 are indeed nonisomorphic, the Prover can always carry out the protocol because the Prover could identify whether **H** came from G_1 or G_2. However, if the graphs were isomorphic, **H** might have come from either G_1 or G_2 , so even with unlimited computational power, the Prover would have no better than 50-50 chance of getting the correct answer. Thus if the Prover is able to answer correctly consistently (say in 100 repetitions of the protocol), the Verifier has convincing evidence that the graphs are actually nonisomorphic.

The Verifier

We define the ***Verifier*** to be a function **V** with three inputs:

1. **Input string**. The objective is to determine whether this string is a member of some language.
2. **Random input**. For convenience in making the definition, we provide the Verifier with a randomly chosen input string instead of the equivalent capability to make probabilistic moves during its computation.
3. **Partial message history**. A function has no memory of the dialog that has been sent so far, so we provide the memory externally via a string representing the exchange of messages up to the present point. We use the notation $\mathbf{m_1\#...\#m_i}$ to represent the exchange of messages $\mathbf{m_1}$ through $\mathbf{m_i}$.

The Verifier's output is either the next message $\mathbf{m_{i+1}}$ in the sequence, or *accept* or *reject*, designating the conclusion of the interaction. Thus **V** has the functional form
$\mathbf{V}\colon\ \Sigma^* \times \Sigma^* \times \Sigma^* \to \Sigma^* \cup \{\textit{accept, reject}\}$.

$\mathbf{V\ (w,\ r,\ m_1\#...\#m_i)\ =\ m_{i+1}}$ means that the input string is **w**, the random input is **r**, the current message history is $\mathbf{m_1}$ through $\mathbf{m_i}$, and the Verifier's next message to the Prover is $\mathbf{m_{i+1}}$.

The Prover

We define the ***Prover*** to be a function **P** with two inputs:

1. **Input string**.
2. **Partial message history**.

The Prover' s output is the next message to the Verifier. **P** has the functional form
$\mathbf{P}\colon \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$.

$\mathbf{P}(w, m_1\#\ldots\#m_i) = m_{i+1}$ means that the Prover sends m_{i+1} to the Verifier after having exchanged messages m_1 through m_i.

The interaction

Next we define the interaction between the Prover and the Verifier. For particular strings **w** and **r**, we write $(V \leftrightarrow P)(w,r) = \text{accept}$ if a message sequence m_1 through m_k exists for some **k** whereby

1. for $0 \leq i < k$, where **i** is an even number, $V(w, r, m_1\#\ldots\#m_i) = m_{i+1}$;
2. for $0 < i < k$, where **i** is an odd number, $P(w, r, m_1\#\ldots\#m_i) = m_{i+1}$; and
3. the final message m_k in the message history is *accept*.

To simplify the definition of the class IP we assume that the lengths of the Verifier's random input and each of the messages exchanged between the Verifier and the Prover are $p(n)$ for some polynomial **p** that depends only on the Verifier. Furthermore we assume that the total number of messages exchanged is at most $p(n)$.

The following definition gives the probability that an interactive proof system accepts an input string **w**. For any string **w** of length **n**, we define

$$\Pr[V \leftrightarrow P \text{ accepts } w] = \Pr[(V \leftrightarrow P)(w,r) = \text{accept}],$$

where **r** is a randomly selected string of length $p(n)$.

The class IP

Definition 10.28 Say that a language **A** is in **IP** if some polynomial time function **V** and arbitrary function **P** exist, where for every function **P'** and string **w**:

1. $w \in A$ implies $\Pr[V \leftrightarrow P \text{ accepts } w] \geq 2/3$, and
2. $w \notin A$ implies $\Pr[V \leftrightarrow P' \text{ accepts } w] \leq 1/3$.

We can amplify the success probability of an interactive proof system by repetition, as we did in Lemma 10.5, to make the error probability exponentially small.

An import of the following remarkable theorem is that, for any language in PSPACE, a Prover can convince a probabilistic polynomial time verifier about the membership of a string in the language, even though a conventional proof of membership might be exponentially long.

Theorem 10.29 IP = PSPACE.