Notes on Complexity Theory Last updated: September, 2011

Lecture 6

Jonathan Katz

1 Space Complexity

We define some of the important space-complexity classes we will study:

Definition 1

$$\begin{aligned} \mathsf{PSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{SPACE}(n^c) \\ \mathsf{NPSPACE} &\stackrel{\text{def}}{=} \bigcup_c \text{NSPACE}(n^c) \\ \mathsf{L} &\stackrel{\text{def}}{=} \text{SPACE}(\log n) \\ \mathsf{NL} &\stackrel{\text{def}}{=} \text{NSPACE}(\log n). \end{aligned}$$

We have seen that $\text{TIME}(t(n)) \subseteq \text{NTIME}(t(n)) \subseteq \text{SPACE}(t(n))$. What can we say in the other direction? To study this we look at *configurations* of a Turing machine, where a configuration consists of all the information necessary to describe the Turing machine at some instant in time. We have the following easy claim.

Claim 1 *Let M be a (deterministic or non-deterministic) machine using space $s(n)$. The number of configurations $\mathcal{C}_M(n)$ of M on any fixed input of length n is bounded by:*

$$\mathcal{C}_M(n) \leq |Q_M| \cdot n \cdot s(n) \cdot |\Sigma_M|^{s(n)}, \tag{1}$$

where Q_M are the states of M and Σ_M is the alphabet of M. In particular, when $s(n) \geq \log n$ we have $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$.

Proof The first term in Eq. (1) comes from the number of states, the second from the possible positions of the input head, the third from the possible positions of the work-tape head, and the last from the possible values stored on the work tape. (Note that since the input is fixed and the input tape is read-only, we do not need to consider all possible length-n strings that can be written on the input tape.) ■

We can use this to obtain the following relationship between space and time:

Theorem 2 *Let $s(n)$ be space constructible with $s(n) \geq \log n$. Then $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$ and $\text{NSPACE}(s(n)) \subseteq \text{NTIME}(2^{O(s(n))})$.*

Proof Let $L \in \text{SPACE}(s(n))$, and let M be a machine using space $O(s(n))$ and deciding L. Consider the computation of $M(x)$ for some input x of length n. There are at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ configurations of M on x, but if $M(x)$ ever repeats a configuration then it would cycle and never halt. Thus, the computation of $M(x)$ must terminate in at most $\mathcal{C}_M(n) = 2^{\Theta(s(n))}$ steps.

Let $L \in \text{NSPACE}(s(n))$. Then there is a non-deterministic Turing machine M deciding L and using space $O(s(n))$ on every computation path (i.e., regardless of the non-deterministic choices it makes). Consider a machine M' that runs M but only for at most $2^{O(s(n))} \geq \mathcal{C}_M(n)$ steps (and rejects if M has not halted by that point); this can be done using a counter of length $O(s(n))$ and so M' still uses $O(s(n))$ space. We claim that M' still decides L. Clearly if $M(x) = 0$ then $M'(x) = 0$. If $M(x) = 1$, consider the *shortest* computation path on which $M(x)$ accepts. If this computation path uses more than $\mathcal{C}_M(|x|)$ steps, then some configuration of M must repeat. But then there would be another sequence of non-deterministic choices that would result in a shorter accepting computation path, a contradiction. We conclude that $M(x)$ has an accepting computation path of length at most $\mathcal{C}_M(|x|)$, and so if $M(x)$ accepts then so does $M'(x)$. ■

The theorem above may seem to give a rather coarse bound for $\text{SPACE}(s(n))$, but intuitively it does appear that space is more powerful than time since space can be *re-used* while time cannot. In fact, it is known that $\text{TIME}(s(n))$ is a strict subset of $\text{SPACE}(s(n))$ (for space constructible $s(n) \geq n$), but we do not know much more than that. We conjecture that space is much more powerful than time; in particular, we believe:

Conjecture 3 $\mathcal{P} \neq \mathsf{PSPACE}$.

Note that $\mathcal{P} = \mathsf{PSPACE}$ would, in particular, imply $\mathcal{P} = \mathcal{NP}$.

1.1 PSPACE and PSPACE-Completeness

As in our previous study of $\mathcal{NP}$, it is useful to identify those problems that capture the essence of PSPACE in that they are the "hardest" problems in that class. We can define a notion of PSPACE-completeness in a manner exactly analogous to $\mathcal{NP}$-completeness:

Definition 2 *Language L' is* PSPACE-hard *if for every $L \in \mathsf{PSPACE}$ it holds that $L \leq_p L'$. Language L' is* PSPACE-complete *if $L' \in \mathsf{PSPACE}$ and L' is* PSPACE*-hard.*

Note that if L is PSPACE-complete and $L \in \mathcal{P}$, then $\mathcal{P} = \mathsf{PSPACE}$.

As usual, there is a "standard" (but unnatural) complete problem; in this case, the following language is PSPACE-complete:

$$L \stackrel{\text{def}}{=} \{(M, x, 1^s) \mid M(x) \text{ accepts using space at most } s\}.$$

For a more natural PSPACE-complete problem we turn to a variant of SAT. Specifically, we consider the language of *totally quantified boolean formulae* (denoted TQBF) which consists of quantified formulae of the form:

$$\exists x_1 \forall x_2 \cdots Q_n x_n \quad \phi(x_1, \ldots, x_n),$$

where ϕ is a boolean formula, and $Q_i = \exists$ and $Q_{i+1} = \forall$ alternate (it does not matter which is first). An expression of the above form is in TQBF if it is true: that is, if it is the case that "for all $x_1 \in \{0,1\}$, there exists an $x_2 \in \{0,1\}$ ...such that $\phi(x_1, \ldots, x_n)$ evaluates to true". More generally, if M is a polynomial-time Turing machine then any statement of the form

$$\exists x_1 \in \{0,1\}^{\mathsf{poly}(n)} \forall x_2 \in \{0,1\}^{\mathsf{poly}(n)} \cdots Q_n x_n \in \{0,1\}^{\mathsf{poly}(n)} \quad M(x_1, \ldots, x_n) = 1,$$

can be converted to a totally quantified boolean formula.

Theorem 4 TQBF *is* PSPACE*-complete.*

Proof It is not too difficult to see that TQBF $\in$ PSPACE, since in polynomial space we can try all settings of all the variables and keep track of whether the quantified expression is true.

We next show that TQBF is PSPACE-complete. Given a PSPACE machine M deciding some language L, we reduce the computation of $M(x)$ to a totally quantified boolean formula. Since M uses space n^k for some constant k, we may encode configurations of M on some input x of length n using $O(n^k)$ bits. Given an input x, we construct (in polynomial time) a sequence of totally quantified boolean formulae $\psi_0(a,b),\ldots$, where $\psi_i(a,b)$ is true iff there is a path (i.e., sequence of steps of M) of length at most 2^i from configuration a to configuration b. We then output $\psi_{n^k}(\mathsf{start},\mathsf{accept})$, where start denotes the initial configuration of $M(x)$, and accept is the (unique) accepting configuration of M. Note that $M(x)=1$ iff $\psi_{n^k}(\mathsf{start},\mathsf{accept})$ is true (using Theorem 2).

We need now to construct the $\{\psi_i\}$. Constructing ψ_0 is easy: to evaluate $\psi_0(a,b)$ we simply test whether $a=b$ or whether configuration b follows from configuration a in one step. (Recalling the proof that SAT is $\mathcal{NP}$-complete, it is clear that this can be expressed as a polynomial-size boolean formula.) Now, given ψ_i we construct ψ_{i+1}. The "obvious" way of doing this would be to define $\psi_{i+1}(a,b)$ as:

$$\exists c: \quad \psi_i(a,c) \wedge \psi_i(c,b).$$

While this is correct, it would result in a formula ψ_{n^k} of *exponential* size! (To see this, note that the size of ψ_{i+1} is roughly double the size of ψ_i.) Also, we have not made use of any universal quantifiers. Instead, we proceed a bit more cleverly and "encode" $\psi_i(a,c) \wedge \psi_i(c,b)$ in a smaller expression. Define $\psi_{i+1}(a,b)$ as:

$$\exists c \forall x,y: \quad \Big(\big((x,y)=(a,c)\big) \vee \big((x,y)=(c,b)\big) \Big) \Rightarrow \psi_i(x,y);$$

it is easy to see that constructing ψ_{i+1} from ψ_i can be done efficiently. (A technical point: although the quantifiers of ψ_i are "buried" inside the expression for ψ_{i_1}, it is easy to see that the quantifiers of ψ_i can be migrated to the front without changing the truth of the overall expression.) The key point is that whereas previously the size of ψ_{i+1} was double the size of ψ_i, here the size of ψ_{i+1} is only an $O(n^k)$ *additive* factor larger than ψ_i and so the size of ψ_{n^k} will be polynomial. ■

A very observant reader may note that everything about the above proof applies to NPSPACE as well, and so the above implies that TQBF is NPSPACE-complete and PSPACE = NPSPACE! We will see another proof of this later.

Playing games. The class TQBF captures the existence of a winning strategy for a certain player in bounded-length perfect-information games (that can be played in polynomial time). Specifically, consider a two-player game where players alternate making moves for a total of n turns. Given moves $p_1,\ldots,p_n$ by the players, let $M(p_1,\ldots,p_n)=1$ iff player 1 has won the game. (Note that M can also encode a check that every player made a legal move; a player loses if it makes the first non-legal move.) Then player 1 has a winning strategy in the game iff there exists a move p_1 that player 1 can make such that for every possible response p_2 of player 2 there is a move p_3 for player 1, $\ldots$, such that $M(p_1,\ldots,p_n)=1$. Many popular games have been proven to be PSPACE-complete. (For this to be made formal, the game must be allowed to grow without bound.)

2 Configuration Graphs and the Reachability Method

In this section, we will see several applications of the so-called *reachability method*. The basic idea is that we can view the computation of a non-deterministic machine M on input x as a directed graph (the *configuration graph* of $M(x)$) with vertices corresponding to configurations of $M(x)$ and an edge from vertex i to vertex j if there is a one-step transition from configuration i to configuration j. Each vertex in this graph has out-degree at most 2. (We can construct such a graph for deterministic machines as well. In that case the graph has out-degree 1 and is less interesting.) If M uses space $s(n) \geq \log n$, then vertices in the configuration graph of $M(x)$ can be represented using $O(s(n))$ bits.[1] If we assume, without loss of generality, that M has only a single accepting state, then the question of whether $M(x)$ accepts is equivalent to the question of whether there is a path from the initial configuration of $M(x)$ to the accepting configuration. We refer to this as the *reachability* problem in the graph of interest.

2.1 NL and NL-Completeness

We further explore the connection between graphs and non-deterministic computation by looking at the class NL. As usual, we can try to understand NL by looking at the "hardest" problems in that class. Here, however, we need to use a more refined notion of reducibility:

Definition 3 *L is* log-space reducible *to L' if there is a function f computable in space $O(\log n)$ such that $x \in L \Leftrightarrow f(x) \in L'$.*

Note that if L is log-space reducible to L' then L is Karp-reducible to L' (by Theorem 2); in general, however, we don't know whether the converse is true.

Definition 4 *L is NL-complete if (1) $L \in$ NL, and (2) for all $L' \in$ NL it holds that L' is log-space reducible to L.*

Log-space reducibility is needed[2] for the following result:

Lemma 5 *If L is log-space reducible to L' and $L' \in$ L (resp., $L' \in$ NL) then $L \in$ L (resp., $L \in$ NL).*

Proof Let f be a function computable in log space such that $x \in L$ iff $f(x) \in L'$. The "trivial" way of trying to prove this lemma (namely, on input x computing $f(x)$ and then determining whether $f(x) \in L'$) does *not* work: the problem is that $|f(x)|$ may potentially have size $\omega(\log |x|)$ in which case this trivial algorithm uses superlogarithmic space. Instead, we need to be a bit more clever. The basic idea is as follows: instead of computing $f(x)$, we simply compute the i^{th} bit of $f(x)$ whenever we need it. In this way, although we are wasting time (in re-computing $f(x)$ multiple times), we never uses more than logarithmic space. ■

[1] Note that x is fixed, so need not be stored as part of a configuration. Whenever we construct an algorithm M' that operates on the configuration graph of $M(x)$, the input x itself will be written on the input tape of M' and so M' will not be "charged" for storing x.

[2] In general, to study completeness in some class $\mathcal{C}$ we need to use a notion of reducibility computable within $\mathcal{C}$.