E0 224 Computational Complexity Theory
Lecture 10 (8 Sep 2014)

Lecturer: Chandan Saha *Scribe: Sarath A Y*

---

1 Introduction

In the last lecture, we introduced the notion of space-bounded computation and defined the classes **PSPACE**, **NPSPACE**, **L** and **NL**. We also studied the configuration graph of a Turing machine and proved Savitch's Theorem. In this lecture, we define **PSPACE** completeness, and show that the language TQBF is **PSPACE**-*complete*. We also introduce the notion of log-space reduction and **NL** completeness.

2 PSPACE Completeness

To define completeness for a complexity class, we first need to define a suitable reduction. This should be guided by the complexity theoretic question that we have in our mind. (Recall that for defining **NP** completeness, we used polynomial time reduction, since we were interested in the question **P** =?**NP**). We already know that **P** $\subseteq$ **PSPACE**. Thus, an interesting question that we would like to ask is whether **P** = **PSPACE**. This suggests that for defining **PSPACE** completeness, we should use polynomial time reduction.

Definition: A language $L \subseteq \{0,1\}^*$ is **PSPACE**-*hard* if for every language $L' \in$ **PSPACE**, $L' \leq_p L$. If in addition, $L \in$ **PSPACE** then we say L is **PSPACE**-*complete*.

An immediate question that comes to our mind is whether there are any **PSPACE**-*complete* languages. To answer this, we define the following language.

$$L := \{(M, x, 1^s) : \text{ Machine } M \text{ accepts } x \text{ using at most } s \text{ cells in } M\text{'s work tape}\}$$

Theorem 2.1. *L is* **PSPACE**-complete

Proof. First, we show that $L \in$ **PSPACE**. Given an input $\{< M, x, 1^s >\}$, consider a machine M' that does the following: M' simulates M on input x and if M uses at most s cells of its work tape then M' accepts $\{< M, x, 1^s >\}$, otherwise it rejects. Note that the machine M' can simulate M with constant space overhead. Thus, M' is a deterministic TM that decides L using polynomial space. (Note that the parameter s is part of the input). Hence, we conclude that $L \in$ **PSPACE**.
Next we show that L is **PSPACE**-*hard*. Let $L' \in$ **PSPACE**, and M be a deterministic Turing machine that decides L' using polynomial space. Thus, there exists a polynomial function $p(.)$

such that M accepts a string $x \in L'$ using at most $p(|x|)$ cells of the work tape. Now, given x, we compute $f(x)$ as:

$$x \longmapsto f(x) =< M, x, 1^{p(|x|)} > \tag{1}$$

Note that $f(.)$ can be computed in polynomial time since $p(.)$ is a polynomial function. Also, observe that $x \in L' \Leftrightarrow f(x) \in L$. Hence, it follows that L is **PSPACE**-*hard*. Since $L \in$ **PSPACE** and L is **PSPACE**-*hard*, L is **PSPACE**-*complete*. □

Next, we describe some natural problems that are **PSPACE**-*complete*.

2.1 Examples of PSPACE-*complete* problems

The following are some examples of **PSPAC**-*complete* problems.

1. Given a regular expression, check if it defines the set of all strings over the alphabet.
2. Hex (board game).
3. Set of all quantified Boolean formula (Theorem 3.1).
4. Finite horizon Markovian decision processes with partially observable states (a proof can be found in [3]).

Apart from all the above problems, an extensive list of **PSPACE**-*complete* problems can be found in Appendix A8 of [2] and in [4].

3 Quantified Boolean Formula (QBF)

Definition: A QBF is a Boolean formula of the form $Q_1 x_1 Q_2 x_2 \dots Q_n x_n \phi(x_1, x_2, \dots, x_n)$ where every Q_i is either $\exists$ or $\forall$ quantifier, x_i is a Boolean variable, and $\phi(x_1, x_2, \dots, x_n)$ is an unquantified Boolean formula.

Example: $\exists x_1 \exists x_2 \dots \exists x_n \left((x_1 \vee x_2) \wedge (\neg x_3 \vee x_4 \vee x_1) \dots\right)$

Remark: Without loss of generality, we can always assume that the unquantified Boolean formula ϕ is a 3CNF. This is because any Boolean formula can be reduced to a CNF and then to a 3CNF (i.e, Cook-Levin reduction).

Unlike unquantified Boolean formulas, because of the quantifiers, a QBF is either true or false. Also, if all the Q_i's in a QBF are $\exists$ quantifier, then the QBF is true if and only if the unquantified Boolean formula ϕ is satisfiable. However, we make the following remark.

Remark: The following two languages are technically different.

$$3-\text{SAT} := \{\phi : \phi \text{ is a 3-CNF and } \phi \text{ is satisfiable}\}$$
$$3-\text{SAT}' := \{\exists x_1 \exists x_2 \ldots \exists x_n \phi(x_1, x_2, \ldots, x_n) : \phi \text{ is a 3-CNF}\}$$

3.1 QBF Game

Recall that for **NP**-*complete* problems, there exists a quickly verifiable and short certificate. It turns out that the analogous notion for **PSPACEP** completeness is the existence of a winning strategy for a two player game with perfect information (i.e., each player can see the move made by the other player)[1].

We can quantify the existence of a winning strategy for player 1 using a QBF. Given a Boolean formula $\phi(x_1, x_2, \ldots, x_{2n})$, consider the QBF

$$\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_{2n} \phi(x_1, x_2, \ldots .x_{2n}) \tag{2}$$

Note that we have alternating $\exists$ and $\forall$ quantifiers in the above formula and this can be interpreted as a two player QBF game. Given a Boolean formula ϕ over the variables $x_1, x_2, \ldots, .x_{2n}$, the QBF game proceeds as follows: the first player picks a value for x_1, then the second player picks a value for x_2, again first player picks a value for x_3 and so on. At the end, we say that the first player wins if $\phi(a_1, a_2, \ldots, a_{2n}) = 1$ where a_i's are the values picked by the players. Observe that the first player has a winning strategy *iff* the QBF (2) is true. Thus, checking the existence of a winning strategy for player 1 in a QBF game is **PSPACE**-*complete* (follows from Theorem 3.1).

Remark: Intuitively we are forced to believe that as the number of quantifications increases, the problem becomes harder (this is not really true, the complexity of the problem is actually captured by the number of alternations of quantifiers). For example, consider the problem of optimizing Boolean circuits. That is, given a Boolean circuit ϕ, we would like to find another circuit ψ that is equivalent to ϕ and whose size is strictly less that the size of ϕ. If we take a closer look at this problem, we can see that there are two quantifiers, that is a $\exists$ quantifier and a $\forall$ quantifier ($\exists$ a circuit such that $\forall$ inputs $\psi(.) = 1$ *iff* $\phi(.) = 1$). Therefore, this problem is harder than the SAT problem.

Now, we define the following language:

$$\text{TQBF} := \{\text{all true QBFs}\} \tag{3}$$

Theorem 3.1. TQBF *is* **PSPACE**-*complete.*

Proof. First, we show that TQBF $\in$ **PSPACE**. To show this, we use a recursive algorithm [1]. Define

$$\psi := Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, x_2, \ldots, x_n) \tag{4}$$

Denote the size of ϕ by m. If $n = 0$, then ϕ contains only constants, and the correctness of ϕ can be evaluated using $O(m)$ space (since as discussed in lecture 9, CIRCEVAL can be decided using logspace). Now, for $n > 0$ and given any bit $b \in \{0, 1\}$, we define a modified QBF $\psi_{|x_1=b}$ from ψ by dropping the quantifier Q_1 and replacing all occurrences x_1 in ϕ by b. Our recursive algorithm $\mathcal{A}$ does the following. If $Q_1 = \exists$, then $\mathcal{A}$ outputs 1 *iff* one of $\mathcal{A}(\psi_{|x_1=0})$ or $\mathcal{A}(\psi_{|x_1=1})$ outputs 1. If $Q_1 = \forall$, then $\mathcal{A}$ outputs 1 *iff* both $\mathcal{A}(\psi_{|x_1=0})$ and $\mathcal{A}(\psi_{|x_1=1})$ outputs 1. By defining $\mathcal{A}$ in this way, $\mathcal{A}$ always outputs the correct value of the QBF ψ. The important observation is that $\mathcal{A}$ can use same space for computing $\mathcal{A}(\psi_{|x_1=0})$ and $\mathcal{A}(\psi_{|x_1=1})$ since the algorithm only needs to remember the result (i.e., either 0 or 1) of $\mathcal{A}(\psi_{|x_1=0})$ and $\mathcal{A}(\psi_{|x_1=1})$. Using the same argument, for computing $\mathcal{A}(\psi_{|x_1=0})$ and $\mathcal{A}(\psi_{|x_1=1})$ form its smaller sized QBFs, $\mathcal{A}$ can recursively use the same space. Thus, if $s_{m,n}$ denotes the space required for computing ψ and since $\mathcal{A}$ uses only $O(m)$ space for computing the base-case (i.e., for computing ϕ when $n = 0$), we can write

$$s_{m,n} = s_{m,n-1} + O(m)$$

From the above recursion, we get $s_{m,n} = O(mn)$. Thus, we conclude that TQBF $\in$ **PSPACE**.

Next, we show that TQBF is **PSPACE**-*hard*. Let $L \in$ **PSPACE** and M be a deterministic TM that decides L using space $p(.)$ (where $p(.)$ is a polynomial function). Our goal is to find a polynomial time computable function $f(.)$ such that

$$x \in L \Leftrightarrow f(x) \in \textbf{PSPACE} \tag{5}$$

Let $G_{M,x}$ be the configuration graph of the machine M on input x. Suppose that we can define a (unquantified) Boolean formula $\phi_i(C_1, C_2)$ such that $\phi_i(C_1, C_2) = 1$ *iff* the configuration C_2 is reachable from C_1 in $G_{M,x}$ via a path of length at most 2^i (we can indeed define such a formula, we will show this later). Let m be the size of $G_{M,x}$. Since M is a poly-space machine, we have $m = O(p(|x|))$. We are trying to capture the formula $\phi_m(C_{start}, C_{accept})$ using polynomial space where C_{start} and C_{accept} are the starting and accepting configurations of the machine M respectively. Observe that, given M and x, we can easily find C_{start} and C_{accept} since they are fixed configurations.

Now, we write $\phi_i(C_1, C_2)$ as a QBF using the following idea. The configuration C_2 is reachable form C_1 via a path of length at most 2^i *iff* there is a configuration C_3 such that C_3 is reachable form C_1 via a path of length at most 2^{i-1} and C_2 is reachable from C_3 via a path of length at most 2^{i-1}. That is

$$\phi_i(C_1, C_2) = \exists C_3 \ (\phi_{i-1}(C_1, C_3) \wedge \phi_{i-1}(C_3, C_2)) \tag{6}$$

Recall that our goal is to express $\phi_m(C_{start}, C_{accept})$ using polynomial space. But from the above expression, we can see that the size of $\phi_i(.,.)$ is at least twice that of $\phi_{i-1}(.,.)$ and hence, if we inductively expand $\phi_m(.,.)$ in the above fashion, we would end up in $O(2^m)$ space for $\phi_m(.,.)$.

Instead of expanding $\phi_i(C_1, C_2)$ using (6), we can introduce additional quantified variables and write $\phi_i(C_1, C_2)$ as

$$\phi_i(C_1, C_2) = \exists C_3 \forall D_1 \forall D_2 \tag{7}$$

C_1	C_2	$\phi_0(C_1, C_2)$
$\cdots - - - * - - - \cdots$	$\cdots * - - - - - - - \cdots$	0
$\cdots - - - * - - - \cdots$	$\cdots - - * - - - - \cdots$	1
$\vdots$	$\vdots$	$\vdots$
$\cdots - * - - - - - \cdots$	$\cdots * - - - - - - - \cdots$	1
$\cdots - * - - - - - \cdots$	$\cdots - - * - - - - \cdots$	1
$\cdots - * - - - - - \cdots$	$\cdots - - - - - - - * \cdots$	0
$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$

Table 1: Table of $\phi_0(.,.)$."$-$" represents a cell of the work tape and "$*$" represents the head position.

and

$$(D_1 = C_1 \wedge D_2 = C_2) \vee (D_1 = C_3 \wedge D_2 = C_2) \Rightarrow \phi_{i-1}(D_1, D_2) \tag{8}$$

(Note that symbols "$=$" and "$\Rightarrow$" in the above expression can be replaced by standard Boolean operations. For instance $\forall x \exists y (x = y)$ is same as $\forall x \exists y (x \wedge y) \vee (\neg x \wedge \neg y)$, and $(x = y) \Rightarrow \neg x \vee y$) Observe that for expressing conditions like $D_1 = C_1$ in (8), we need only $O(m)$ space. Also, since the number of conditions (such as $D_1 = C_1$) in the expression (8) are bounded, the size of representation of (8) is $O(m) + \text{size}(\phi_{i-1})$. This together with (7) tells us that

$$\text{size}(\phi_i(.,.)) = \text{size}(\phi_{i-1}(.,.)) + O(m) \tag{9}$$

Inductively expanding $\text{size}(\phi_m(.,.))$ using the above expression, we have

$$\begin{aligned} \text{size}(\phi_m(.,.)) &= \text{size}(\phi_{m-1}(.,.)) + O(m) && (10)\\ &= \text{size}(\phi_{m-2}(.,.)) + O(m) + O(m) && (11)\\ &\vdots && (12)\\ &= \text{size}(\phi_0(.,.)) + O(m^2) && (13) \end{aligned}$$

Thus, to show $\phi_m(.,.)$ is a polynomial size formula, it suffices to show that $\phi_0(.,.)$ has polynomial size. To show this, we use the crucial property that computation is local. Recall that $\phi_0(C_1, C_2) = 1$ *iff* the machine M can go from C_1 to C_2 by applying its transition function (i.e., C_2 can be reached from C_1 in just 1 step). Now, consider Table (1). The first two columns are the set of all possible configurations of M and the third column captures $\phi_0(C_1, C_2)$. Our goal is to express $\phi_0(C_1, C_2)$ using a poly size Boolean formula. Observe that the size of the above table is $2^{O(m)}$ since total number of possible configurations for M is $2^{O(m)}$. Thus, we cannot express $\phi_0(C_1, C_2)$ directly using the above table since it takes exponential space. To represent $\phi_0(C_1, C_2)$ using polynomial space, we do the following. We group all configurations in the first column of the table such that M has the same head position in all the configurations in one group. Note that we have polynomially many groups since head position can be

encoded using $O(m)$ bits. Now, observe that a configuration C_2 cannot be obtained from C_1 (i.e., $\phi_0(C_1, C_2) \neq 1$) if the head positions of C_1 and C_2 differ by more than 1 cell. Also, if the head positions of C_1 and C_2 does not differ by more than 1 cell and the contents of work tape in C_1 and C_2 is different for at least 2 cells, then we have $\phi_0(C_1, C_2) = 0$ (aside, we observe that for any configuration C_1, $|\{C_2 : \ \phi_0(C_1, C_2) = 1\}|$ is bounded). Thus, for evaluating the formula $\phi_0(C_1, C_2)$, we just need to compare the head positions of C_1 and C_2; and if the head positions do not differ by more than 1 cell (if they do, then $\phi_0(C_1, C_2) = 0$), then we need to compare the contents of 3 cells (the cell corresponding to the head position in C_1 and the two cells adjacent to it) of C_1 and C_2 (if more than 2 cells have different contents, then we have $\phi_0(C_1, C_2) = 0$). Clearly (since each configuration can be encoded by using $O(m)$ bits and we need to only compare contents of 3 cells), these requirements can be represented by a Boolean formula by using $O(m)$ space. Thus, $\text{size}(\phi_0(C_1, C_2)) = O(m)$ and from (13), it follows that $\text{size}(\phi_m(C_1, C_2)) = O(m^2)$. Therefore, we get a polynomial size QBF $\phi_m(C_1, C_2)$ in prenex form (see the next remark), and by plugging in the values of C_{start} and C_{accept} into the formula $\phi(.,.)$ we can get $\phi_m(C_{start}, C_{accept})$. Finally, note that $x \in L$ *iff* $G_{M,x}$ has a path from C_{start} to C_{accept}, and we expressed the reachability of C_{accept} from C_{start} by using a polynomial size QBF $\phi_m(.,.)$ such that $x \in L$ *iff* $\phi_m(C_{start}, C_{accept})$ is true. Hence, we have $L \leq_p$ TQBF. This completes the proof. □

Remark: Every QBF can be written in prenex form (i.e., all the quantifiers appear in the beginning of the QBF) without much blow up in the size of the representation. This can be done by using the Boolean identities like $\neg(\forall x \phi(x)) = \exists x(\neg\phi(x))$. For instance, we have

$$\begin{aligned} \forall y(y \vee \neg\forall x \phi(x) &\Leftrightarrow \forall y(y \vee \exists x \neg\phi(x)) \\ &\Leftrightarrow \forall y(\exists x(y \vee \neg\phi(x)) \end{aligned}$$

4 NL completeness

As discussed Section 2, for defining **NL** completeness, we need a suitable reduction. The complexity theoretic question that we have is whether $\mathbf{L} = \mathbf{NL}$. This naturally suggests that the reduction should be log-space reduction, i.e., given an input string x, we have a function $f(x)$ such that $f(x)$ is computable by a deterministic TM by using at most $O(\log(|x|)$ cells in the work tape.

Remark: It can be shown that $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P}$ (Claim 4.1) . Therefore, it does not make sense to use polynomial time reduction for defining **NL** completeness, since poly time reduction is more powerful than the class **L**.

Claim 4.1. $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P}$

Proof. Clearly, we have $\mathbf{L} \subseteq \mathbf{NL}$. To show $\mathbf{NL} \subseteq \mathbf{P}$, suppose that $L_1 \in \mathbf{NL}$ and let M_1 be an NDTM that decides L_1 using logarithmic space. Since the work tape of M_1 has size $O(\log n)$,

number of vertices in the configuration graph of M_1 for any input string x will $2^{O(\log n)}$ and hence M_1 can run for at most $2^{O(\log n)} = poly(n)$ steps. Thus, it follows that $L_1 \in \mathbf{P}$ and hence $\mathbf{NL} \subseteq \mathbf{P}$. □

From our intuition, suppose that we agree upon using simple logspace reduction to define **NL** completeness. Then we have a tricky issue. Recall that, ultimately our goal is to say that if $L_1 \leq_{logspace} L_2$ and if $L_2 \in \mathbf{L}$, then $L_1 \in \mathbf{L}$. Suppose that M_{L_2} is a machine that decides L_2. Then given an input x, for deciding whether $x \in L_1$, we first compute the function $f(x)$ and run the machine M_{L_2}. But, it is not necessary that the output of function $f(.)$ uses only logspace. That is, if the output of function $f(.)$ has size more than logspace, then the machine that decides L_1 may end up using more space than just logspace and therefore we may not be able to say that $L_1 \in \mathbf{L}$.

To solve this issue, can modify the definition of logspace reduction as follows. Suppose that given the input x and an index i, we can compute the i th bit of the function $f(x)$ by using logarithmic space (such a function f is said to be implicitly computable in logspace), and we modify the definition of logspace reduction by using functions that are implicitly computable in logspace. In that case, whenever M_{L_2} needs a bit of $f(x)$ (say ith bit), then we postpone the computation of M_{L_2} and compute the i th bit of $f(x)$ by using logarithmic space. Therefore, a machine that decides L_1 will only use logarithmic space even if the size of the function $f(.)$ is more than logspace, and hence if $L_1 \leq_{logspace} L_2$ and $L_2 \in \mathbf{L}$ then we can say that $L_1 \in \mathbf{L}$. Also, observe that for $f(x)$ to be implicitly computable in logspace, size of $f(x)$ should be polynomial in the input size (because the machine that compute $f(x)$ using logarithmic space can run for at most $2^{O(log|x|)}$ steps and hence can output at most $2^{O(log|x|)} = poly(|x|)$ bits of $f(x)$). We will give a formal definition of logspace reduction and **NL** completeness in the next lecture.

References

[1] Sanjeev Arora, and Boaz Barak. *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

[2] Michael R. Garey, and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Macmillan Higher Education, 1979.

[3] Christos H. Papadimitriou, and John N. Tsitsiklis. *Complexity of Markov Decision Processes*, Mathematics of operations research, Vol. 12, No. 3, August 1987.

[4] `http://en.wikipedia.org/wiki/List_of_PSPACE-complete_problems`