CSE 531: Computational Complexity I **Winter 2016**

Lecture 7: PSPACE-completeness, Savitch's Theorem

January 27, 2016

Lecturer: Paul Beame *Scribe: Paul Beame*

1 Space Complexity

In the last class we proved the following theorem.

Theorem 1.1. *For all space-constructible $S(n) \geq \log_2 n$,*

$$\mathsf{NTIME(S(n))} \subseteq \mathsf{DSPACE(S(n))} \subseteq \mathsf{NSPACE(S(n))} \subseteq \mathsf{DTIME(2^{O(S(n))})},$$

where $\mathsf{DTIME(2^{O(S(n))})} = \bigcup_{\mathsf{c}} \mathsf{DTIME(2^{cS(n)})}$.

Definition 1.2. *We can now define some specific space complexity classes:*

$$\begin{aligned} \mathsf{L} &= \mathsf{DSPACE}(\log \mathsf{n}) && \text{“logspace”} \\ \mathsf{NL} &= \mathsf{NSPACE}(\log \mathsf{n}) && \text{“nondeterministic logspace”} \\ \mathsf{PSPACE} &= \bigcup_{k\geq 1} \mathsf{DSPACE(n^k)} \\ \mathsf{NPSPACE} &= \bigcup_{k\geq 1} \mathsf{NSPACE(n^k)} \end{aligned}$$

Applying the theorem to these definitions and using what we know already, we obtain:

Corollary 1.3.

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}.$$

We consider some examples of functions computable using only logarithmic space:

$$MAJORITY = \{x \mid x \text{ has at least as many 1's as 0's}\} \in \mathsf{L}$$

since the TM need only maintain the two counters and compare them at the end of the computation. We can also consider functions such as $Sort$ where

$$Sort_n : \{1, \ldots, n^2\}^n \to \{1, \ldots, n^2\}^n$$

which takes a list n integers represented in binary ($2 \log 2n$ bits each) and outputs them in order from smallest to largest. Note that this is only possible because the output tape is not counted in the space bound. (Note that the configuration graph does not contain the output tape since it is never read.) Though most sorting algorithms such a Mergesort, Quicksort, Radix sort require a linear amount of space we can compute $Sort$ in logarithmic space using one of the simplest algorithms, Selection sort. To implement Selection sort one only needs:

- an index i to the currently scanned element in the list
- the largest value v_o that has already been output
- the candidate next output value, which is the value of the smallest value $v_{min} > v_o$ scanned so far.
- the number n_{min} of times that v_{min} has been observed so far.

which is only $O(\log n)$ bits on the work tape.

Define $PATH = \{[G, s, t] \mid G$ is a directed graph with a path from s to $t\}$.

Lemma 1.4. $PATH \in \mathsf{NL}$.

Proof. The general idea is for the NTM to guess and verify the path from s to t in G but there is a problem with doing this as we have done for NP since there is not enough room to write down a guess.

The solution is to start with $u = s$ repeatedly guess the next vertex v on the path, verify that (u, v) is an edge of G, and then set $u = v$. If $v = t$ is ever reached then the algorithm halts and accepts. This has one problem: if the graph has a cycle reachable from s, then the algorithm may run forever. To avoid this, the algorithm first computes n, the number of vertices in G and stops there search after n edges have been traversed. The total space is now $O(\log n)$. □

Just as for time, a small amount of additional space allows us to compute new things.

Theorem 1.5. *If $g \geq \log_2 n$ is a space-constructible function and $f(n)$ is $o(g(n))$ then*

$$\mathsf{SPACE(f(n))} \subsetneq \mathsf{SPACE(g(n))} \text{ and } \mathsf{NSPACE(f(n))} \subsetneq \mathsf{NSPACE(g(n))}.$$

Proof. Both proofs follows a similar outline to the deterministic time hierarchy theorem using

$$D = \{[M]01^k \mid M \text{ on input } x = [M]01^k \text{ does } \textbf{not} \text{ accept } x \text{ in } g(|x|) \text{ space}\}$$

but are much simpler since (1) one can simulate $S(n)$ space-bounded TMs with a universal TM that uses only $O(S(n))$ space on a single tape and the same applies to universal NTMs and (2) one can complement the outcome of a nondeterministic computation by trying all computation bounds without only a constant factor increase in the space bound. □

Corollary 1.6. $\mathsf{L} \neq \mathsf{PSPACE}$ *and* $\mathsf{NL} \neq \mathsf{NPSPACE}$.

We will show later in this lecture that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. It is, however, consistent with our knowledge that $\mathsf{P} = \mathsf{PSPACE}$ and $\mathsf{L} = \mathsf{NP}$. Refuting either of these would be easier than proving that $\mathsf{P} \neq \mathsf{NP}$ and might be a step towards it.

Definition 1.7. *(1) B is* PSPACE-hard *iff* $\forall A \in \mathsf{PSPACE}$, $A \leq_P B$.

(2) B is PSPACE-complete *iff*
(a) $B \in \mathsf{PSPACE}$, *and*
(b) B *is* PSPACE-*hard.*

PSPACE-complete problems include a quite wide variety of questions of practical interest, for example:

- Safety properties of computer systems: "Does this computer systems get into a bad state?"
- Planning problems in AI: "Is there a plan involving a sequence of allowable actions that lets me reach a goal state?"
- Existence of winning strategies in games, e.g. $n \times n$-Checkers.

We can define a generic PSPACE-complete problem just as we did with $EXP\text{-}COM$, but we are interested in more natural examples.

Quantified Boolean Formulas (QBF) We can define first order logic with quantifiers over bit variables $x_i \in \{0, 1\}$. In this way, if φ is a Boolean formula then we can give alternative definitions of problems we have already considered using QBF:

$$[\varphi] \in SAT \Leftrightarrow \exists x_1 \ldots \exists x_n \varphi(x_1, \ldots, x_n) \text{ is true and}$$
$$[\varphi] \in TAUT \Leftrightarrow \forall x_1 \ldots \forall x_n \varphi(x_1, \ldots, x_n) \text{ is true.}$$

In general, formulas in *QBF* are expressible as

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi(x_1, \ldots, x_n)$$

where φ is a Boolean formula, each $Q_i \in \{\exists, \forall\}$ and each $x_i \in \{0, 1\}$.

We therefore have the definition:

$$TQBF = \{[\Phi] \mid \Phi \text{ is a QBF that evaluates to true}\}.$$

Theorem 1.8. *$TQBF$ is* PSPACE-*complete.*

Proof. We first prove that $TQBF$ is in PSPACE. To do this, on input $Q_1x_1Q_2x_2\ldots Q_nx_n\varphi(x_1,\ldots,x_n)$ consider the complete binary tree that branches on $x_1,\ldots,x_n$ in turn. The algorithm will be the analogue of the DFS brute force search of all branches in the case of satisfiability or tautology which computes the formula's value on the assignment at each of the leaves. On input $\Phi = Q_1x_1Q_2x_2\ldots Q_nx_n\varphi(x_1,\ldots,x_n)$ The general recursion $Eval$ maintains an assignment $\vec{b}$ that is initially empty.

```
$Eval(\Phi, i)$:
if $i = n + 1$ then return $\varphi(\vec{b})$
else
    $b_i \leftarrow 0$
    if $Q_i = \exists$ then
      if $Eval(\Phi, i + 1) = \mathbf{true}$ then
        return **true**
      else
        $b_i \leftarrow 1$
        return $Eval(\Phi, i + 1)$
      endif
    elseif $Q_i = \forall$ then
      if $Eval(\Phi, i + 1) = \mathbf{false}$ then
        return **false**
      else
        $b_i \leftarrow 1$
        return $Eval(\Phi, i + 1)$
      endif
    endif
endif.
```

Clearly this algorithm only needs to maintain $\vec{b}$ and the current value of i in addition to its input (except for computing $\varphi(\vec{b})$) and its total space use is polynomial (even linear) in the input size.

It remains to show that $TQBF$ is PSPACE-complete. Let $A \in \mathsf{PSPACE}$. Therefore there is a normal-form TM M_A that decides A using space at most $S(n)$ that is $O(n^k)$ for some k.

$T(n)$ be an upper bound on the maximum number of configurations possible for M_A on inputs of length n. By construction, $T(n)$ is at most $2^{cS(n)}$ for some constant c depending on A. We therefore have:

$$\begin{aligned} x \in A \Leftrightarrow &\text{there is a path in } G_{M_A,x} \text{ from } C_0 \text{ to } C_{accept} \\ \Leftrightarrow &\text{there is a path of length} \leq T(|x|) \text{ in } G_{M_A,x} \text{ from } C_0 \text{ to } C_{accept} \end{aligned}$$

The general idea of the argument is to define a QBF formula ψ_t that takes the (binary encodings of) two configurations C and D of $G_{M_A,x}$ expresses that there is a path in $G_{M_A,x}$ of length $\leq t$ between them. The reduction will map x to $\phi_{T(|x|)}([C_0],[C_{accept}])$. It remains to define the formulas ϕ_t. Now

$$\psi_1([C],[D]) \equiv ([C]=[D]) \vee \varphi_{M_A,x}([C],[D])$$

where $\varphi_{M,x}$ is the size $O(S(n))$ CNF formula expressing the edge relation on $G_{M,x}$. (For convenience we drop the encoding brackets around configurations from now on and simply write $\psi_t(C,D)$ instead.) We will also assume that t is a power of 2 and round T up to the nearest power of 2 since we consider paths with *at most* T steps.

We know that there is a path from C to D of length at most t if and only if there is some middle vertex C_m such that there are paths of length at most $t/2$ from C to C_m and from C_m to D. Therefore would could try to define $\psi_t(C,D)$ to be $\exists C_m(\psi_{t/2}(C,C_m) \wedge \psi_{t/2}(C_m,D))$ where $\exists C_m$ expresses $O(S(n))$ existential Boolean quantifiers. Unfortunately, this is not sufficiently efficient; we can see that the size of ψ_t is roughly $O(S(n))$ plus 2 times the size of $\psi_{t/2}$. When expanded out we get that the total size of ψ_T would be roughly $\Theta(S \cdot T)$ which is exponential and certainly not polynomial-time computable. However, we have only used $\exists$ quantifiers so we have much more flexibility. There is a trick that lets us express an equivalent statement using only one copy of $\varphi_{t/2}$. Define

$$\begin{aligned}\varphi_t(C,D) \equiv \exists C_m \forall C_1 \forall C_2 (((C_1 = C) \wedge (C_2 = C_m)) \vee ((C_1 = C_m) \wedge (C_2 = D)) \\ \rightarrow \psi_{t/2}(C_1,C_2).\end{aligned}$$

The size of ψ_t is now that of $\psi_{t/2}$ plus $O(S(n))$. The total size is then $O(S(n)\log T(n))$ which is $O(S^2(n))$, i.e. $O(n^{2k})$ which is polynomial. It is also very easy to construct ψ_T given input x and the reduction is polynomial time. □

Theorem 1.9 (Savitch's Theorem)**.** *For any* $S(n) \geq \log_2 n$, $\mathsf{NSPACE(S(n))} \subset \mathsf{DSPACE(S^2(n))}$.

Proof. We use the same idea as in the proof of the PSPACE-completeness of $TQBF$. (The idea originated with Savitch's Theorem which pre-dates NP-completeness.) Let $A \in \mathsf{NSPACE(S(n))}$ and let M_A be an $O(S(n))$ space normal-form NTM deciding membership in A. As before we have

$$x \in A \;\Leftrightarrow\; \text{there is a path of length} \leq T(|x|) \text{ in } G_{M_A,x} \text{ from } C_0 \text{ to } C_{accept}$$

where $T(n)$ is $2^{cS(n)}$ for some constant c (since $S(n) \geq \log_2 n$) and we are going to use the idea of reuse of resources on the two sides of a middle configuration C_m.

Define the recursive function $Reach(i,C,D)$ which will output 1 iff there is a path of length at most 2^i from C to D in $G_{M_A,x}$ as follows.

```
$Reach(i, C, D)$:
if $i = 0$ then return 1 if $C = D$ or if $C$ can lead to $D$ in 1 step of $M_A$
if $i \geq 1$ then
    for all $C_m$ in $G_{M_A,x}$
        if $Reach(i-1, C, C_m) = 1$ and $Reach(i-1, C_m, D) = 1$ then return 1
    endfor
    return 0
```

The largest i for which we will need to call $Reach$ is at most $cS(n)$ since $T \leq 2^{cS(n)}$. The depth of recursion for a call of $Reach(i, C, D)$ is i and the amount of storage need on the stack is only the sizes needed to store a constant number of configurations plus the value of i. This totals $O(S(n)$ per level and $O(S(n))$ levels for $O(S^2(n))$ overall.

Thus far, we have needed to be able to compute the value of $i = cS(n)$ for the first call, which would require that $S(n)$ be constructible in space $O(S^2(n))$. However, we can also do this without needing to construct $S(n)$. Indeed the algorithm just needs to know M_A and does not need to know the bound $S(n)$ at all.

The general idea is that the algorithm can try all space bounds starting at space $S = 1, 2, \ldots$, incrementing the space bound by 1 each time. It will be looking at larger and larger configurations. If it finds a path using space bound S then the input should be accepted. However, if it does not find a path, then it is possible that the space bound S was simply too small. The idea will be to check this also at every step. At the bottom level of recursion, whenever it is checking $Reach(0, C, D)$ the algorithm will check whether or not any configuration reachable from C in 1 step is actually larger than S. If so, it will know that the computation using space bound S is not sufficient to reject the input and the algorithm must increase the space bound. If S was sufficient and a path was not found with space bound S, then the algorithm rejects. Because M_A has space bound $O(S(n))$, the algorithm will eventually finish with S at most $cS(n)$ as required. $\square$

Corollary 1.10. PSPACE = NPSPACE.