CS 221: Computational Complexity **Prof. Salil Vadhan**

Lecture Notes 8

February 19, 2010 **Scribe: Kevin Lee**

1 Agenda

1. Provably Intractable Problems

2 Provably Intractable Problems

Some lower bounds that we proved so far:

1. Hierarchy theorems give us unnatural problems that do not exist in certain complexity classes. These give us "unnatural" problems in $\mathbf{EXP} \backslash \mathbf{DTIME}(2^n)$ or $\mathbf{PSPACE} \backslash \mathbf{SPACE}(n)$.
2. $\exists \epsilon > 0$ such that TQBF $\notin \mathbf{SPACE}(n^\epsilon)$.
3. $\forall \epsilon > 0$ we have that SAT $\notin \mathbf{TISP}(n^{1+o(1)}, n^{1-\epsilon})$.

We want to find some more natural problems that require superpolynomial time. We can do this by finding "natural" complete problems for classes at the exponential level (like **EXP**, **NEXP**, **EXPSPACE**). Often these will be "concise" versions of problems complete for classes at the polynomial level (like **P**, **NP**, **PSPACE**). We begin with a survey of some results of this type, and then work out one example in detail.

3 Variants of Bounded Halting

To find some natural problems, we consider variations of the bounded halting problem:

1. $\left\{\left\langle M, x, 1^t\right\rangle : M \text{ accepts } x \text{ within } t \text{ steps}\right\}$ we know to be **P**-complete.
2. $\{\langle M, x, t\rangle : M \text{ accepts } x \text{ within } t \text{ steps}\}$ turns out to be **EXP**-complete.
3. $\left\{\left\langle M, 1^t\right\rangle : \exists x, M \text{ accepts } x \text{ within } t \text{ steps}\right\}$ we know to be **NP**-complete.
4. $\{\langle M, t\rangle : \exists x, M \text{ accepts } x \text{ within } t \text{ steps}\}$ turns out to be **NEXP**-complete.

Thus by writing the time bound more concisely — in binary instead of unary — the complexity goes up an exponential.

This phenomenon is like the converse of what we saw with translation/padding arguments. There we could take a problem at the exponential level, and by making the input representation less concise via padding, get a problem at the polynomial level. However, note that the polynomial-level versions of bounded halting are *not* quite the same as exponentially-padded versions of the exponential-level ones, because the description of M can be linear in the length of the input, whereas

in an exponentially padded instance, all but a small portion of the input has no information content. Indeed, it is unlikely that any exponentially padded problem is **NP**-complete (similar to the way no sparse language can be **NP**-complete).

4 Implicit Problems

4.1 Implicit Hamiltonian Cycle

A *Hamiltonian cycle* on a directed graph is a cycle that visits each vertex exactly once. Deciding whether an explicitly given graph has a Hamiltonian cycle is known to be **NP**-complete. We can consider the same problem on a graph described *implicitly* by a circuit

$$C : \{0,1\}^n \times \{0,1\}^n \mapsto \{0,1\}$$

Which takes a pair of vertices and returns 1 iff they are connected in the graph. Formally, the graph described by C is defined as

$$G = \left(\{0,1\}^n, \{(u,v) : C\,(u,v) = 1\}\right)$$

The Implicit Hamiltonian Cycle problem is the language

$$L = \{C : C \text{ describes a graph with a Hamiltonian cycle}\}.$$

This turns out to be **NEXP**-complete. It can be proven in a similar way to the standard **NP**-completeness reduction from Circuit Satisfiabilityto Hamiltonian Cycle, applied to the Implicit Circuit SAT problem, described next.

4.2 Implicit Circuit SAT

Similarly, we can define Implicit Circuit SAT as the language with

$$\left\{C : C \text{ describes a circuit } C' \text{ that is satisfiable}\right\}$$

We can describe a circuit by defining the function $C(i,j,k)$ which corresponds to whether gate k in C' have gates i, j as inputs and with what operation. Note that C' is of size exponential in the input length of C.

Theorem 1 Implicit Circuit SAT *is* **NEXP**-*complete.*

Proof Sketch: Similar to Cook-Levin (the theorem that shows the **NP**-completeness of Circuit Satisfiability). The circuit $C'(\cdot)$ should simulate the verifier $M(x,\cdot)$ for the **NEXP** language with x hardwired in. Even though C' is now exponentially large, it is highly regular (consisting of mainly of many copies of the same constant-sized circuit), and thus can be described by a poly(n)-sized circuit C. □

In general, implicit versions of problems complete at the polynomial level tend to be complete at the exponential level. However, these problems (as well as the Bounded Halting problems) are not as natural as we would like, as they still make explicit reference to computation. In the next section, we'll see a problem where "conciseness" can be achieved in a natural problem with no explicit reference to computation.

5 Regular Expressions with Exponentiation

Regular expressions are patterns that match strings. Regular expressions with exponentiation (or equivalently just squaring) are regular expressions where symbols or strings can be repeated an arbitrary number n of times using an exponentiation operator that we sometimes denote $\uparrow n$ and sometimes denote using a superscript of n. For example, the expression

$$R = (a \cup b)^* \circ (c \cup \varepsilon) \uparrow (100) = (a \cup b)^* \circ (c \cup \varepsilon)^{100}$$

generates the language:

$$L(R) = \{\text{any string formed with a's and b's followed by } \leq 100 \text{ c's}\}$$

Formally, $L(R)$ for regular expressions R with exponentiation are defined recursively. Starting with the base cases

$$L(\varepsilon) = \{\varepsilon\} \;\; L(\sigma) = \{\sigma\} \;\; L(\emptyset) = \emptyset$$

with the inductive rules

$$L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$$

$$L(R^*) = L(R)^* = \{x_1 x_2 \cdots x_k : k \geq 0, x_i \in L(R)\}$$

$$L(R^n) = L(R)^n = \{x_1 x_2 \cdots x_n : x_i \in L(R)\}$$

$$L(R_1 \circ R_2) = \{x_1 x_2 : x_1 \in L(R_1), x_2 \in L(R_2)\}$$

Any occurrence of exponentiation can of course be eliminated by expanding it into concatenation, but this will incur a size blow-up that is exponential in the bitlength of the exponent. Thus exponentiation is a source of "conciseness' in this problem.

6 $\mathbf{ALL_{REX\uparrow}}$

The problem we are interested in is

$$\text{ALL}_{\text{REX}\uparrow} = \{R : R \text{ is a regular expression with exponentiation such that } L(R) = \Sigma^*\}$$

Theorem 2 $\text{ALL}_{\text{REX}\uparrow}$ *is* **EXPSPACE***-complete under logspace mapping reductions. It is even complete for* $\mathbf{SPACE}\left(2^{O(n)}\right)$ *under linear-time, logspace reductions.*

Corollary 3 $\text{ALL}_{\text{REX}\uparrow}$ *cannot be solved in space (or time)* $2^{\epsilon n}$*, for some constant* $\epsilon > 0$*.*

Proof $\mathbf{ALL_{REX\uparrow}} \in$ EXPSPACE: We can decide $\text{ALL}_{\text{REX}\uparrow}$ of a regular expression R of length n by

1. Convert the exponents into concatenations. This results in a size blowup to $N \leq 2^n$.
2. Convert to NFA in the obvious way. This only requires linear space on N.
3. Convert to a DFA. This could require up to $2^{O(N)}$ states.
4. Check that no reject states are reachable. By Savitch's theorem, this takes space $O\left(N^2\right)$ on a graph of size $2^{O(N)}$.

Note that $2^{O(N)}$ is too much space. However, we don't need to store the entire DFA in memory, so we use the usual trick of composing space bounded algorithms to deal with this. ■

Proof $\mathbf{ALL_{REX\uparrow}}$ is EXPSPACE-hard: Given a language L decided by TM M using space 2^{n^k}, we'll give a logspace mapping $w \mapsto R$ such that

$$L(R) = \{\text{Strings that do not represent rejecting computations of } M \text{ on } w\}$$

Note that this is logically equivalent because all strings are not rejecting computations of M on w iff M does not have a rejecting computation. WLOG we assume that M is a 1-tape Turing machine.

We represent computations as $C_1 \# C_2 \# \cdots \# C_t$ where the C_i's are configurations (described by the contents of M's tape, with M's state written immediately preceding the current symbol being read) and # character is a delimeter between the configurations. Thus the computation history is a string over the alphabet $\Delta = \Gamma \cup Q \cup \{\#\}$, where Γ is the tape alphabet of M and Q is the set of states of M. WLOG we may assume that all configurations are of length exactly 2^{n^k}.

In order to have a valid reject computation, we must start with the start state, have a valid transition to the next configuration, and end at a reject state. Thus, all strings that do not represent rejecting computations can be represented by the regular expression

$$R = R_{\text{bad-start}} \cup R_{\text{bad-window}} \cup R_{\text{bad-reject}}$$

For $R_{\text{bad-start}}$, we note that at least one of the symbols in the start configuration must be incorrect. So if the expected start configuration is

$$q_0 w_1 w_2 \cdots w_n \sqcup \sqcup \cdots \sqcup$$

Then we can generate

$$R_{\text{bad-start}} = \Delta_{-q_0}\Delta^* \cup \Delta^1\Delta_{-w_1}\Delta^* \cup \Delta^2\Delta_{-w_2}\Delta^* \cdots \cup \Delta^{n+1}\left(\Delta \cup \varepsilon\right)^{2^{n^k}-n-2}\Delta_{-\sqcup}\Delta^* \cup \Delta^{2^{n^k}}\Delta_{-\#}\Delta^*$$

Where Δ represents the union of all possible symbols and Δ_{-a} represents the union of all possible symbols except for a. Most of the expressions match strings that have the wrong character in a particular place, and the second to last expression matches strings where one of the tail characters is not a blank.

Strings that do not reject the input are simply strings that do not contain the reject state. Thus,

$$R_{\text{bad-reject}} = \Delta^*_{-q_{\text{reject}}}$$

Finally, in order to obtain $R_{\text{bad-window}}$ we observe that we can figure out whether a transition is valid or not simply by looking at three consecutive symbols in each configuration, since we just need to know the value under the head, the to the left of the head, and to the right of the head. Thus,

$$R_{\text{bad-window}} = \bigcup_{\text{bad}(abc,def)} \Delta^* abc \Delta^{2^{n^k}-2} def \Delta^*,$$

where $\text{bad}(abc, def)$ means that having abc in one configuration is inconsistent with having def in same positions of the next configuration, according to the transition function of the TM M.

Note that the total length of R is dependent on the exponents in it, since everything else remains constant with regard to the length of x. The largest exponents are 2^{n^k}, so the length of R is $O\left(n^k\right)$, which is polynomial in n. ■

7 Variations of $\mathbf{ALL_{REX}}$

What happens when we allow or disallow exponentiation and starring?

	$\uparrow$	$\neg\uparrow$
$*$	**EXPSPACE**-complete	**PSPACE**-complete
$\neg*$	**coNEXP**-complete	**coNP**-complete

Above the results without $*$ actually refer to the *equivalence* problem for regular expressions (deciding whether $L(R_1) = L(R_2)$) since a regular expression without $*$ can't possibly generate all strings.

Note that in both rows we see that the exponentiation operator increases the complexity by an exponential.