10/10

6.840

Theory of Computation

2004/12/02

Dan Ports

drkp@mit.edu

Problem 6-1

Let B be the language of properly-nested parentheses and brackets. We show that B is in L. The following procedure decides whether a string of parentheses and brackets is properly nested:

We first scan over the string from left to right, maintaining a counter of the current depth. Every time we encounter a '(' or '[' character, we increment the counter, and every time we encounter a ')' or ']', we decrement the counter. If at any time the counter goes negative, or if the counter is not equal to zero once the end of the string is reached, we reject. If neither happens, then we have ensured that the string is a properly nested sequence, ignoring the distinction between parentheses and brackets. Note that this phase requires only a logarithmic amount of space, since we need only maintain the counter. Since the input string is of length n, it contains at most n open parentheses or brackets, so the counter fits in $O(\log n)$ bits.

We next need to ensure that each open parenthesis or bracket is associated with a close symbol of the same type. To check whether the ith open symbol is associated with a close symbol of the same type, we first scan over the string counting open symbols until we find the ith. We then record the type of this symbol. We initialize a depth counter to zero and scan through the string from the current position, incrementing and decrementing it when we encounter open and close symbols as before. When we encounter a close symbol and the depth counter is zero, we have found the close symbol associated with the ith open symbol. If the open and close symbols are not of the same type, we reject. Note that this requires only logarithmic space, for the same reason. We perform this check for all values of i from 1 to the number of open parentheses (which is $O(n)$), and we can reuse the space, so this can be performed for all i in logarithmic space. If all of these checks pass, we accept. This accepts all properly nested parentheses and brackets, so the problem is in L.

10/10

6.840
Theory of Computation
2004/12/02

Dan Ports
drkp@mit.edu

Problem 6-2

a) To check whether $\langle x, y, z \rangle \in ADD$, we need to verify the addition $x + y = z$. We can do this using logarithmic space. We will do so by verifying each bit of the addition in turn. Let x_i be the ith lowest order bit, and similarly y_i and z_i. We also maintain a single bit c to track the carry state, which is initially zero. Then for each i from 1 to the number of bits in the input numbers, we add the bits $x_i + y_i + c$. The result is a two-bit integer. The high bit is the new value for c, and the low bit should equal z_i. If at any time the low bit does not equal the corresponding z_i, then $x + y \neq z$, and we reject. Otherwise, if the loop completes without rejecting, we accept.

This algorithm requires logarithmic space. Performing bit-addition requires constant space, as does holding the carry bit c. So the space bound is dominated by the requirement for holding the counter i. The value of i is bounded above by the number of bits in each input integer, which is $O(n)$, so i requires $O(\log n)$ bits. Thus $ADD \in \mathsf{L}$.

b) We can also check whether $x + y$ is a palindrome using logarithmic space. Let z be the sum, and let m be the number of digits in it. (Note that $m = \Theta(n)$, and that we can determine m using logarithmic space by performing the addition without keeping track of the result, simply counting the number of bits required.) As before, let z_i be the ith lowest order bit of z. If z is a palindrome, then $z_i = z_{m-i+1}$.

Our algorithm is to check, for each k from 1 to $\frac{m}{2}$, whether $z_k = z_{m-k+1}$. We can do so in much the same method as verifying an addition in ADD. We maintain a carry bit c initially equal to zero. Then for each i from 1 to m, we add the bits $x_i + y_i + c$. The result is z_i. If $i = k$ or $i = m - k + 1$, we save z_i in a register. Once we have iterated over all i, we reject if $z_k \neq z_{m-k+1}$. As before, for each k, this requires only constant space plus logarithmic space for keeping track of i. Performing this check for all k also requires logarithmic space, since the space for each iteration can be reused. If we have iterated over all k without rejecting, then z is a palindrome, and we accept. So $PAL - ADD \in \mathsf{L}$.

6.840
Theory of Computation
2004/12/02

Dan Ports
drkp@mit.edu

10

Problem 6-3

Lemma 1. *STRONGLY-CONNECTED* $\in$ NL.

Proof. We show that *STRONGLY-CONNECTED* is in coNL. We nondeterministically guess two vertices in the graph that are not connected. We then verify that there is no path between these vertices. We can do this using logarithmic space because *PATH* $\in$ coNL. So *STRONGLY-CONNECTED* $\in$ coNL = NL. □

Theorem 2. *STRONGLY-CONNECTED is* NL*-hard.*

Proof. We reduce *PATH* to *STRONGLY-CONNECTED*. Let $\langle G, s, t\rangle$ be an instance of *PATH*. Generate a new graph G' that is identical to G except that for every vertex v there is an edge $(v \to s)$ and an edge $(t \to v)$. Then $\langle G, s, t\rangle \in$ *PATH* iff $\langle G', s, t\rangle \in$ *STRONGLY-CONNECTED*. First, if there is a path from s to t in G', then G' is strongly connected. For any pair of vertices $\langle u, v\rangle$, there is a path $u \to s \to t \to v$. If $\langle G, s, t\rangle \notin$ *PATH*, there is no path from s to t in G, and this remains true in G'. So $\langle G', s, t\rangle \notin$ *STRONGLY-CONNECTED*.

It is easy to see that this reduction can be performed using a log-space transducer. The transducer simply outputs the same graph G, then iterates over all vertices v adding the $(v \to s)$ and $(t \to v)$ edges. This requires only logarithmic space on the work tape to keep track of the index of v. So *STRONGLY-CONNECTED* is NL-hard. □

Corollary 3. *STRONGLY-CONNECTED is* NL*-complete.*

Proof. It is NL-hard and in NL, so it is NL-complete. □

Dan Ports
drkp@mit.edu

Problem 6-4

a) A_{LBA} is PSPACE-complete. Furthermore, the reduction from any problem in PSPACE to A_{LBA} requires polynomial space; i.e. it can be computed by a log-space transducer. Thus if it is in NL, then every problem in PSPACE is also in NL. We know there are problems in PSPACE that are not in NL by the hierarchy theorems. So $A_{\mathsf{LBA}} \notin \mathsf{NL}$.

b) If $A_{\mathsf{LBA}} \in \mathsf{P}$, then every problem in PSPACE is in P. It is not currently known whether this is the case, so it is unknown whether $A_{\mathsf{LBA}} \in \mathsf{P}$.

6.840
Theory of Computation
2004/12/02

Dan Ports
drkp@mit.edu

10

Problem 6-5

a)

Theorem 4. *If* $A \in \mathsf{TIME}(n^6)$ *then* $pad(A, n^2) \in \mathsf{TIME}(n^3)$.

Proof. Suppose $A \in \mathsf{TIME}(n^6)$. Then we will show how to decide whether a string B of length n is in $pad(A, n^2)$ in $O(n^3)$ time. Observe that to be in $pad(A, n^2)$, B must consist of a substring s of length $\sqrt{n}$ followed by $(n - \sqrt{n})$ '#' symbols. We can easily verify that this is the case by scanning over B (in linear time). Then $B \in pad(A, n^2)$ if and only if s is in A. Since the string s has length $\sqrt{n}$, and $A \in \mathsf{TIME}(n^6)$, deciding $s \in A$ requires $O\left((\sqrt{n})^6\right) = O(n^3)$ time. Thus, $pad(A, n^2) \in \mathsf{TIME}(n^3)$. □

+5

b)

Theorem 5. *If* $\mathsf{NEXPTIME} \neq \mathsf{EXPTIME}$, *then* $\mathsf{P} \neq \mathsf{NP}$.

Proof. Suppose there is some language L that is in NEXPTIME but not in EXPTIME. Then $L \in \mathsf{NTIME}\left(2^{n^k}\right)$ for some k. By the same reasoning as above, if $L \in \mathsf{NTIME}\left(2^{n^k}\right)$ then $pad(L, 2^{n^k}) \in \mathsf{NTIME}(n)$. So $pad(L, 2^{n^k}) \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then $pad(L, 2^{n^k}) \in \mathsf{TIME}(n^l)$ for some l. But then we can solve L by padding the input to length 2^{n^k} and solving the resulting problem. This requires time $O(n^l)$ in the size of the padded problem, so time $O\left(\left(2^{n^k}\right)^l\right) = O\left(2^{n^{kl}}\right)$. Thus $L \in \mathsf{EXPTIME}$. But we defined L to not be in EXPTIME. So $\mathsf{P} \neq \mathsf{NP}$ unless NEXPTIME = EXPTIME. □

+5

6.840

Theory of Computation

2004/12/02

Dan Ports

drkp@mit.edu

Problem 6-6

(10)

Theorem 6. *UNIQUE-SAT* $\in \mathsf{P}^{SAT}$

Proof. Let ϕ be a Boolean formula. We give a polynomial-time procedure for determining whether ϕ has a unique satisfying solution using a SAT oracle.

First, we query the oracle with ϕ to determine whether $\phi \in SAT$. If not, we reject, since it has no satisfying solution at all. If it does, we then determine what the satisfying solution is. We do so one variable at a time.

Let $x_1, x_2, \cdots, x_m$ be the variables in ϕ. Suppose that we have determined the values of all variables with index less than i. We try both possibilities for variable x_i (true or false) to determine which one leads to a satisfying assignment. To test whether a particular assignment leads to a satisfying assignment, we add clauses to ϕ to force variable assignments: for each l, if x_l is assigned to be true, we add the clause "$\wedge x_l$", and if it assigned to be false, we add the clause "$\wedge \overline{x_l}$". We query the SAT as to whether ϕ with forcing clauses for all variables with index $\leq i$ is satisfiable; if so, then a satisfying assignment with these variable assignments exists. This determines a value of x_i in a satisfying assignment. We repeat this process for i from 1 to m, to find a satisfying assignment for all variables. Note that this process requires $\Theta(m) = O(n)$ time and space. ✓

Now we determine whether there exists any *other* satisfying assignment for ϕ. We do so by augmenting ϕ with a new clause containing $\overline{x_i}$ if x_i is true in the satisfying assignment we found, and x_i if $x_i is false$. (For example, "$\wedge(\overline{x_1} \vee \overline{x_2} \vee x_3 \vee x_4)$" if x_1 and x_2 are true and x_3 and x_4 in the satisfying assignment.) This forces at least one variable to take a different value. We then query the SAT oracle with this augmented formula. If it returns true, we reject; otherwise we accept. The oracle accepts only if there is a satisfying assignment with some variable different, i.e. if the satisfying assignment is non-unique. So *UNIQUE-SAT* $\in \mathsf{P}^{SAT}$. ✓ □