CSC 363, Winter 2010 — Solutions to Long Assignment #2

Question 1: The book defines (p. 193) a Linear Bounded Automaton (LBA) as a one-tape Turing machine in which the tape head is not allowed to move past the end of the input string. Since the tape alphabet is allowed to be larger than the input alphabet, the effective amount of memory available to an LBA can be any constant times the length of the input string.

Consider a machine which is an LBA with an additional work tape, for which the head can be moved left or right as for an ordinary Turing Machine, but for which the symbol on a tape square can be changed only if the current symbol is blank. In other words, the only write operations allowed are ones that append a non-blank symbol to the end of the non-blank part of the work tape. (Arbitrary write operations are allowed on the input tape, as for an LBA, within the range of the squares originally occupied by the input string.)

Prove that any language that is decidable in polynomial time on an ordinary deterministic Turing Machine is decidable in polynomial time on a deterministic machine of the sort described above.

If a language, A, is decidable in polynomial time by an ordinary deterministic Turing Machine, then there is some Turing Machine M, and some time bound, $O(n^k)$, for which M decides A in time $O(n^k)$. We will show that M can be simulated in $O(n^{3k+1})$ time on the type of Turing Machine described above.

We can't do this simulation using just the input tape, since its $O(n)$ length may not be big enough (when $k > 1$). We need to use the work tape, which can hold any amount of information, but we need to somehow simulate the ability to write on any square. To do this, each step of M will be simulated by copying the information from the work tape that represents the tape contents of M to the end of the work tape, if necessary changing the one square that M writes to, and extending the number of squares by one if M writes to a new square.

To start, we copy the input tape to the work tape, while counting the number of symbols, in binary notation. We also record the position of M's tape head, which is 1 to start, again in binary notation. These two binary numbers can be stored on the read-write input tape, since this tape is $O(n)$ in length, and the numbers have only $O(\log n^k) = O(\log n)$ bits. (To handle very small values of n, we can store the first few bits of these numbers in the finite state of the machine.)

When simulating a step of M, we can move the the head on the work tape back to the position of M's tape head, using another counter that takes $O(\log n)$ bits, decide what transition M would take based on the symbol there, and then make a new copy of M's tape contents, with the tape square at the head changed as specified by M's transition function. To make this copy, we need to go back and forth from each successive square of the old copy to the end of the tape, which requires using another two counters to keep track of where we are, which again take only $O(\log n)$ bits. (Note that we can't keep track of where we are in the copy operation by writing markers on the work tape!) We also increment the counter holding the number of squares used on M's tape if M used another square.

We of course accept when we simulate that M would accept, and reject when we simulate that M would reject.

Since M runs in $O(n^k)$ time, it can't use more than $O(n^k)$ tape squares. Going back and forth to copy one square takes $O(n^k)O(\log n)$ time (the log factor is for doing the counting operations to see how far back to go), so one copy operation takes $O(n^k)O(n^k)O(\log n) = O(n^{2k+1})$ time. These copy operations dominate the total time, so simulating M for $O(n^k)$ steps takes $O(n^{3k+1})$ time.

Question 2: The book defines (p. 269) the SUBSET-SUM language as follows:

$$\text{SUBSET-SUM} \;=\; \{\langle S, t\rangle \mid \; S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq S \text{ we have } \textstyle\sum y_i = t\}$$

where $x_1, \ldots, x_k$ and t are positive integers, with standard binary encoding, and $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are "multisets", in which repetitions are allowed.

For example, $\langle\{12, 1, 5, 1, 1, 12\},\ 7\rangle$ is in SUBSET-SUM, because $5 + 1 + 1 = 7$, but on the other hand $\langle\{12, 1, 5, 1, 1, 12\},\ 11\rangle$ is not in SUBSET-SUM.

For this question, consider the language SUBSET-SUM-NOREP, defined in the same way as SUBSET-SUM, except $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are ordinary sets, in which repetitions are not allowed. Prove that SUBSET-SUM is polynomial time reducible to SUBSET-SUM-NOREP.

We can represent t and the elements of S in decimal notation. A reduction from SUBSET-SUM to SUBSET-SUM-NOREP can be done as follows. Given t and S with k elements, $s_1, \ldots, s_k$ (some perhaps duplicates), we map to t' and S', with S' having $2k$ elements, as follows:

$$\begin{aligned} t' &= t \times 10^k + 10^{k-1} + 10^{k-2} + \cdots + 10^0 \\ s'_i &= s_i \times 10^k + 10^{i-1}, \quad \textit{for } i = 1, \ldots k \\ s'_{k+j} &= 10^{j-1}, \quad \textit{for } j = 1, \ldots k \end{aligned}$$

Clearly there are no repetitions in S'. If for some $R \subseteq \{1, \ldots, k\}$, the sum of s_i for $i \in R$ is t, then the sum of s'_i plus s'_{k+j} for $i \in R$ and $j \in \{1, \ldots, k\} - R$ will be t'. Conversely, if some subset of S' consisting of s'_j for $j \in R'$ adds up to t', then the subset of S consisting of s_i for $i \in R' \cap \{1, \ldots, k\}$ will add up to t.

This is therefore a valid reduction. It takes only polynomial time, since t' and the elements of S' have just twice as many digits as t and the elements of S, and computing these digits is trivial.

Another valid answer to this question (though not what I intended) is to show that SUBSET-SUM-NOREP is NP-complete by modifying the proof in the book that SUBSET-SUM is NP-complete. (Note that the proof in the book produces instances of SUBSET-SUM with repetitions, so it does not show SUBSET-SUM-NOREP is NP-complete without modification.) Since SUBSET-SUM is in NP, it follows that there must be a polynomial time reduction of SUBSET-SUM to SUBSET-SUM-NOREP.

Question 3: Consider the language SUBSET-SUM-POW2 defined the same as SUBSET-SUM except that the integers $x_1, \ldots, x_k$ and t are not represented in standard binary notation. Instead, a positive integer is represented as a list $(b_1, \ldots, b_m)$ of non-negative integers, in standard binary notation, with $b_i < b_j$ if $i < j$, representing the number $2^{b_1} + \cdots + 2^{b_m}$. For example, the integer 6 is represented as $(1, 10)$, since $6 = 2^1 + 2^2$.

Prove that SUBSET-SUM-POW2 is NP-complete. Remember: You need to show *two* things to show that a language is NP-complete (see definition 7.34 on p. 276).

First, we need to show that SUBSET-SUM-POW2 is in NP. A verifier for SUBSET-SUM-POW2 can take as certificate the subset of S that adds up to t. We need to show that the verifier can confirm that the integers in this subset add up to t in polynomial time. We <u>cannot</u> do this by just converting these integers from the powers-of-two encoding to standard binary encoding, since the standard binary encoding can be exponentially larger than the powers-of-two encoding. (For example, if b_1 has k bits, 2^{b_1} will have about 2^k bits.) Instead we need to do the additions needed to verify that the sum is t in the powers-of-two notation. This is easy to do, however. When adding $2^{b_1} + \cdots + 2^{b_m}$ and $2^{b'_1} + \cdots + 2^{b'_{m'}}$, we need to just go through the b_i and b'_j in increasing order, outputing both the b_i and the b'_j when $b_i \neq b'_j$, and performing an addtion when $b_i = b'_j$ — noting that $2^b + 2^b = 2^{b+1}$. Then we compare the result with t. This takes polynomial time (indeed, just $O(n \log n)$ time).

Second, we need to show that all problems in NP can be reduced in polynomial time to SUBSET-SUM-POW2. We can do this by showing that SUBSET-SUM can be reduced in polynomial time to SUBSET-SUM-POW2, since SUBSET-SUM is proved in the book to be NP-complete. SUBSET-SUM and SUBSET-SUM-POW2 are the same language except for the difference in encoding. We just need to show that we can convert from the binary encoding of SUBSET-SUM to the powers-of-two encoding of SUBSET-SUM-POW2 in polynomial time. This is easy — for each binary number, output the b_i values for the 1's. For a number with k bits, the size of the b_i values will be $O(\log k)$, so this procedure takes time $O(n \log n)$.