Problem Set 8

Spring 10

Due: Tuesday, April 20, in class before the lecture.

Please follow the homework format guidelines posted on the class web page:

`http://www.cs.uiuc.edu/class/sp10/cs373/`

1. E.T. [**Category**: Puzzle, **Points**: 5]

   Show that the problem of deciding whether aliens exist in the universe is decidable. More precisely, show that there is a Turing machine that will print "YES" if there are aliens in the universe, and "NO" otherwise!

Solution:

A problem is decidable if there exists a Turing machine that decides it. However, we don't need to be abe to *construct one machine*. We simply have to prove the existence of a machine that decides it.

Pick a TM M_1 that always prints "YES" and halts, and pick another TM M_2 that always prints "NO" and halts. We have two cases: If aliens exist then TM M_1 does the job, if aliens don't exist then TM M_2 does the job. So in both cases there is a TM that does the job and this problem is hence decidable (of course we don't know which of these two TMs actually decides it, but we know that such a TM exists!).

In general, for any one question, or any finite number of questions, there is always a decider. Decidability makes sense only when the language is infinite. So, the set of all theorems mankind has proven is, trivially, decidable.

This does not mean that we are over-trivializing unknown problems. It just means that the theory of computation considers any finite language trivial. The theory of computation is more about the *structure* of infinite sets (languages) that can be defined using finite means (like a machine). What these sets mean is not, technically speaking, part of the study. This is often a source of great confusion in people who don't understand this difference, but use Turing's undecidability arguments in their philosophical arguments (very similar to how quantum theory is misinterpreted in many arguments).

2. Reduction à la Rice's Theorem [**Category**: Proof, **Points**: 20]

   A language $L \subseteq \Sigma^*$ is *closed under reversal* if for every $w \in L$, $w^R \in L$.

   Show that $L_{rev} = \{\langle M \rangle \mid M \textit{ is a TM and } L(M) \textit{ is closed under reversal}\}$ is undecidable.

   You may not simply appeal to Rice's theorem (however, you can adapt the proof of Rice's theorem to solve this problem).

Solution:

First note that if $|\Sigma| = 1$, then L is decidable (because every string in Σ^* will be the reverse of itself and so every language will be the reverse of itself). So assume $a, b \in \Sigma$ and $a \neq b$.

We prove that $\overline{L_{rev}}$ is not decidable which is enough to show that L_{rev} is not decidable. We have

$$\overline{L_{rev}} = \{\langle M \rangle \mid \langle M \rangle \text{ is not a TM code or, it is a TM and } L(M) \text{ is not closed under reversal}\}$$

Consider a TM M' built this way:

```
**Algorithm** $M'(x)$
1.  $v$←Simulate $M(w)$
2.  **if** $v$ = Yes and $x$ = "ab"
3.      **then return** Yes
4.      **else  return** No
```

You can see that we have these two properties:

$$\langle M, w \rangle \in A_{TM} \implies M(w) = Yes \implies L(M') = \{ab\} \implies \langle M' \rangle \in \overline{L_{rev}}$$
$$\langle M, w \rangle \notin A_{TM} \implies M(w) \neq Yes \implies L(M') = \varnothing \implies \langle M' \rangle \notin \overline{L_{rev}}$$

Note that when $M(w) \neq Yes$ we have two cases as usual, $M(w)$ may halt and return No (explicit reject) or it may never halt (implicit reject). In both cases M' will never accept (inspect the code of M' in the previous page).

So to figure out whether $\langle M, w \rangle$ is a member of A_{TM} we can just build $\langle M' \rangle$ and check whether it is a member of $\overline{L_{rev}}$ (so we have reduced A_{TM} to $\overline{L_{rev}}$). Here is the code of a decider for $\overline{L_{rev}}$ using this idea:

```
**Algorithm** $D_{A_{TM}}(\langle M, w \rangle)$
1.  $\langle M' \rangle$←Write down the code of $M'$ using $\langle M, w \rangle$
2.  **return** $D_{\overline{L_{rev}}}(\langle M' \rangle)$
```

3. Queueueueueueueue [**Category**: Proof, **Points**: 20]

A *queue automaton* is an automaton with finitely many states, that can manipulate an (unbounded) queue data-structure. Fix an input alphabet Σ and a queue alphabet Γ, where $\Sigma \subseteq \Gamma$. The input, a word in Σ^*, is given to the queue automaton in the queue, and in each step the automaton can *enqueue* a letter onto the queue, or dequeue a letter from the queue. A queue is simply a FIFO (first-in-first-out) data-structure, and can contain any number of letters. The queue automaton is non-deterministic, and accepts a word if there is some way to reach an accept state.

Show that the membership problem for queue automata in *undecidable.*

In other words, show that, given a queue automaton QA and a word $w \in \Sigma^*$, checking whether QA accepts w is undecidable.

Your answer can be at a high-level description of a reduction.

Below is a *formal* description of a queue automaton in case you want to understand the question better using a more precise description (you need not give the reduction in this kind of detail).

Let $\Gamma_\epsilon = \Gamma \cup \{eps\}$.

A queue automaton is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc})$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $q_{acc} \in Q$ is the accepting state, and $\delta \subseteq Q \times \Gamma_\epsilon \times \Gamma_\epsilon \times Q$.

Intuitively, if $(q, a, b, q') \in \delta$, then it means that the automaton can go from state q to state q' by dequeuing a from the queue and enqueing b to the queue.

Formally, a *configuration* of a queue automaton is a pair (q, x) where $q \in Q$ and $x \in \Gamma^*$ (q is the state the queue automaton is in, and x is the content of the queue, with the head of the queue being the first letter in x and the tail of the queue being the last letter in x).

We define the transitions between configurations as follows: for any $x \in \Gamma^*$, $a, b \in \Gamma$, $(q, ax) \to (q', xb)$ iff $(q, a, b, q') \in \delta$. (This captures a move that dequeues a and enqueues b.)

A word w is *accepted* by the queue automaton if there is a sequence of configurations $C_1, C_2, \ldots, C_n$ such that $C_1 = (q_0, w)$, for each $1 \le i < n$, $C_i \to C_{i+1}$, and $C_n = (q_{acc}, y)$ for some $y \in \Gamma^*$.

Solution:

First we show that for any TM M we can build a QA M_Q such that $L(M_Q) = L(M)$. The idea is that M_Q remembers all explored parts of TM tape inside its queue, in the same order as the symbols appear on the tape. It introduces a *primed* version of each tape symbol (for example a' for a). At each moment exactly one of the symbols in the queue is of the primed type: the symbol that the head of TM is pointing to. Therefore M_Q also remembers the position of the tape head inside its queue.

M_Q introduces another new tape symbol: $\sim$. This symbol is being used for *rotating* queue. Rotating queue is an operation that allows M_Q to scan all the information that it has stored in its queue (and potentially change them if it wants to). To perform a rotation M_Q enqueues $\sim$ and then repeatedly dequeues a symbol from its queue and stores it in a temporary buffer (This buffer contains just one symbol and is being simulated using M_Q's internal states). Then it dequeues some other symbol a, enqueues the buffer into the queue and overwrite the buffer with a. These operation are repeated till it dequeues the symbol $\sim$: it throws away the symbol $\sim$ and just dequeues the buffer.

Now to simulate a transition in M like $\delta(q,a) = (p,b,R)$, M_Q starts a rotation. Once it dequeues symbol a' it knows that it corresponds to the symbol that the head points to (so it should be converted to a b). Now M_Q looks at its buffer (assume it contains symbol c) and it knows that the head should now point to this cell, therefore it enqueues c' (instead of c). Then it writes b to the buffer (instead of a) and it continues the rest of the rotation. Other kinds of transitions are handled in a similar way (and at the very beginning we need a similar rotation that *puts a prime* on the very first symbol of the queue - because that is the place the head points to initially). This proves that we can simulate a TM using a QA.

Now this is the language that we are asked to show undecidable:

$$A_{QA} = \{\langle Q, w\rangle \mid Q \text{ is a QA and } Q \text{ accepts } w\}$$

We reduce A_{TM} to this set to prove it is undecidable: starting with $\langle M, w\rangle$ build the code for M_Q as described above. Since $L(M) = L(M_Q)$, M accepts w iff M_Q accepts w, therefore we can just ask whether M_Q accepts w. But this we can solve using decider of A_{QA}:

```
**Algorithm** $D_{A_{TM}}(\langle M, w\rangle)$
1.   $\langle M_Q\rangle \leftarrow$Write code for $M_Q$ as described using $\langle M, w\rangle$
2.   **return** $D_{A_{QA}}(\langle M_Q, w\rangle)$
```

4. Nondeterminism [**Category**: Construction, **Points**: 20]

For every natural number n, let n_b be the binary representation of n. For example, $5_b = 101$. Assume there is a TM, *Multiplier*, that when given inputs m_b and n_b on two tapes, outputs $(m * n)_b$ on the third tape. The TM *Multiplier* is provided for you as a black box that you can use.

Construct a *nondeterministic Turing machine* M_{comp} to decide if a natural number x, represented as x_b, is a *composite number* (a composite number is a number that is not prime). Your NTM must be a decider (i.e. halt no matter what non-deterministic choices it makes) and furthermore halt within $O(poly(|x_b|))$ steps (i.e. work in polynomial time). To do the latter, you must exploit *non-determinism.*

Describe your construction clearly (it need not be *formal*) and in sufficient detail so that is understandable, clear and easy to see it's correct.

Solution:

The idea is to non-deterministically generate two numbers a and b with leftmost digit equal to 1, at least 2 digits (we want to avoid $a = 1$ or $b = 1$) and at most $|x_b|$ digits (equivalent to guessing a and b), compute ab and compare the result with x_b and accept iff they are the same.

The key point is that this NTM will accept iff at least one of this guesses succeed and we know that a number is composite iff there is *some* factorization of that number, therefore NTM peforms exactly what we are looking for (in other words: if the number is composite there is a lucky guess that exactly guesses the factors and accepts, and if the number is prime no such lucky guess exists).

```
Algorithm $M_{comp}(x_b)$
1.  $a \leftarrow$“1”
2.  $b \leftarrow$“1”
3.  for $i \leftarrow 1$ to $|x_b| - 1$
4.      do $d \leftarrow$Non-deteministically pick from $\{$“0”, “1”, $\epsilon\}$
5.         $a \leftarrow concatenate(a, d)$
6.         $d \leftarrow$Non-deteministically pick from $\{$“0”,“1”, $\epsilon\}$
7.         $b \leftarrow concatenate(b, d)$
8.  $m \leftarrow Multiply(a, b)$
9.  if $m = x_b$
10.    then return true
11.    else return false
```

Let's analyse the running time. The amount of work done in one iteration of the **for** loop needs constant time (lines 4,5,6,7). The loop is being executed $|x_b|$ times therefore the whole loop needs time $O(|x_b|)$ to complete. Moreover at each step of the loop, $|a|$ and $|b|$ grow by at most 1, therefore at the end of the loop we have $|a| = O(|x_b|)$ and $|b| = O(|x_b|)$.

Line 8 calls the black box and we assume that the black box performs multiplication in a polynomial time bound in terms of its input length, so this will need $O(poly(|a|+|b|)) = O(poly(|x_b|))$.

The rest of the code needs constant time (lines 1,2,9,10,11), therefore the total time is: $O(1) + O(|x_b|) + O(poly(|x_b|) = O(poly(|x_b|))$.

5. Dovetailing [**Category**: Construction, **Points**: 20]

Prove that the language $L_{two} = \{ \langle M \rangle \mid |L(M)| \geq 2\}$ is Turing-recognizable. Informally, L_{two} is the set of Turing machines that accept at least *two* strings.

Solution:

We use dovetailing to prove that this languages is TM-recognizable.

```
Algorithm $A(\langle M \rangle)$
1.  if $\langle M \rangle$ is not an encoding of some TM
2.     then return reject
3.  for $i \leftarrow 1$ to $\infty$
```

```
4.        do counter := 0
5.            for $j \leftarrow 1$ to $i$
6.                do Simulate $M$ on $w_j$ for $i$ steps
7.                    if $M$ accepts
8.                        then counter := counter + 1
9.            if counter $\geq 2$
10.               then return accept
```

6. (Extra Credit) Highly Non-recognizable (NOT COMPULSORY FOR HONORS) [**Category**: Proof, **Points**: 20]

   Let $L_{ALL} = \{\langle M \rangle | M \textit{ is a } \mathsf{TM} \textit{ with input alphabet } \Sigma \textit{ and } L(M) = \Sigma^*\}$.

   Informally, L_{ALL} is the set of TMs that accept every input string.

   We want you to show that neither L_{ALL} nor its complement is TM-recognizable!

   You can assume that all strings encode some Turing machine, and hence $\overline{L_{ALL}} = \{\langle M \rangle | M \textit{ is a } \mathsf{TM} \textit{ with input alphabet } \Sigma \textit{ and } L(M) \neq \Sigma^*\}$.

   (a) Prove that $\overline{L_{ALL}}$ is not TM-recognizable.

   Hint: Be careful when using reductions to prove non-recognizability. When you reduce A to B in order to show that if B was recognizable, then A is recognizable, you create a recognizer for A using a recognizer for B. However, you must be careful *not to flip the answer given by the oracle recognizing B* as recognizable languages are not closed under complement.

Solution:

We know that A_{TM} is recognizable but not decidable, hence its complement is not TM-recognizable. We will use reduction from $\overline{A_{TM}}$ to prove that $\overline{L_{ALL}}$ is not TM-recognizable. Suppose that $\overline{L_{ALL}}$ is recognizable, then there exists a recognizer $\overline{M_{ALL}}$, such that $L(\overline{M_{ALL}}) = \overline{L_{ALL}}$.

Consider TM N:

```
Algorithm $N(\langle M, w \rangle)$
1.  Construct $M'$ as described below.
2.  Feed $M'$ to $\overline{M_{ALL}}$.
3.  if $\overline{M_{ALL}}$ accepts
4.      then return accept
5.      else return reject
```

```
Algorithm M'$(x)$
1.  Simulate $M$ on $w$. if $M$ accepts
2.      then return accept
```

```
3.       else  return reject
```

Consider the language of M'. $L(M') = \Sigma^*$ if M accepts w, and $L(M') = \emptyset$ if M does not accept w. If we feed M' to $\overline{M_{ALL}}$, it will accept it if and only if M does not accept w. This means that N is a recognizer for $\overline{A_{TM}}$. Contradiction.

(b) Prove that L_{ALL} is non-recognizable.

Hint: This direction is trickier. Assume L_{ALL} is recognizable and that M_{ALL} is a TM recognizing it. Note that we cannot assume that M_{ALL} halts on all inputs. All we know is that it accepts x iff $x \in L_{ALL}$. You may use the existence of M_{ALL} to show that $\overline{A_{TM}}$ is recognizable, which we know is false. Also, when you construct the recognizer, when it is given input $\langle M, w\rangle$, you may want to consider simulating M on w for a *finite* number of steps.

Solution:

Suppose that L_{ALL} is recognizable, then there exists a recognizer M_{ALL}, such that $L(M_{ALL}) = L_{ALL}$.

Consider TM N:

```
Algorithm N(⟨M, w⟩)
1.   Construct $M'$ as described below.
2.   Feed $M'$ to $\overline{M_{ALL}}$.
3.   if $\overline{M_{ALL}}$ accepts
4.       then return accept
5.       else  return reject
```

```
Algorithm M'(x)
1.   Simulate $M$ on $w$ for $|x|$ steps. if $M$ accepts
2.       then return reject
3.       else  return accept
```

Consider the language of M'. $L(M') = \Sigma^*$ if M does not accept w, and $L(M') = \{x \mid |x| \leq k\text{and } M \text{ accepts } w \text{ in } k \text{ steps}\}$ if M does not accept w. If we feed M' to M_{ALL}, it will accept it if and only if M does not accept w. This means that N is a recognizer for $\overline{A_{TM}}$. Contradiction.