CIS 511 Homework 4

Stephen Phillips, Dagaen Golomb

February 24, 2015

Problem 1

We want to show that finding if a Turing Machine has a *useless state* is Turing Decidable. We formulate this as a language:

$$L = \{\langle M, q\rangle \mid q \in Q(M),\ \forall s \in \Sigma^* M(q) \text{ does not enter } q\}$$

Now we reduce this to A_{TM} to show that it is undecidable. First as usual, we suppose toward contradiction that there was a decider of this language, which we will denote D. We build a Turing Machine in the following manner:

function $N(\langle M, x\rangle)$
  Build the description for the following machine
  function $A(y)$
    Ignore input y
    Simulate $M(x)$, and output what it outputs
  end function
  Simulate $D(\langle A, q_A\rangle)$, where q_A is the accept state of A, and output what it outputs
end function

If $M(x)$ halts and accepts, then we eventually reach the accept state of A, by construction. Also by construction we never reach the accept state if $M(x)$ loops or if it rejects. Since D is a decider, and building the machine description takes finite time, N always halts. Therefore N accepts if and only if D accepts, which in turn accepts if and only if $\langle M, x\rangle \in A_{TM}$. Therefore, N decides A_{TM}, a contradiction.

Problem 2

Note that the set of decidable languages is countable. If a language is decidable, there must exist a Turing Machine which decides it. We proved in class the set of TMs is countable (via enumerating descriptions of all combinations of bits starting at length 1 and increasing onward). The set of TM's that decides subset languages in $\{1\}^*$ is a subset of all TM's and is therefore countable. Thus, the set of decidable languages in $\{1\}^*$ is countable.

Now, using diagonalization, we can prove that the set of all languages of subsets of $\{1\}^*$ is uncountable. This will show that there must be some language that is a subset of $\{1\}^*$ that is not decidable since the set of TM's deciding languages of this form is countable.

Picture the set of languages that are a subset of $\{1\}^*$ as a bit string with 0 if they include a particular string in $\{1\}^*$ and 0 otherwise. Suppose this set of languages is countable. Then we could list all such languages as so:

ϵ	1	11	111	...
0	0	0	0	
1	0	0	0	
0	1	0	0	
1	1	0	0	
$\vdots$				$\ddots$

Now create the languages where if the i^{th} entry in the i^{th} row is one, that string is not in the language. Conversely, if it is a zero it is in the language. Now, we have created a language that is clearly composed of substrings of $\{1\}^*$, so it should be in the table. However, this language conflicts with every table entry, by construction, by at least one string. But this table is supposed to list all languages of the given form. This is a contradiction. Therefore, the set of languages that are a subset of $\{1\}^*$ is uncountable.

Thus, we have shown that there must exist a language that is a subset of $\{1\}^*$ that cannot be decided.

Problem 3

We want to show that the intersection of two context-free languages is undecidable using A_{TM}.

We modify the statment of A_{TM} that $\langle M, w\rangle$ accepts to use the computation paths of M. Specifically we use the following format:

$$L = \{w_1 w_1^{\mathcal{R}} \# \ldots \# w_n w_n^{\mathcal{R}} \mid w_1, \ldots, w_n \text{ is the computation path of } M \text{ on } w\}$$

Now we want to form this as the intersection of context-free languages. It is possible to build a PDA to accept strings where w_i follows from w_{i-1}. The idea is that we push the symbols of w_{i-1} on the stack and mark the state/head and the letter that follows. Then we check to make sure that $w_{i-1} = w_i^{\mathcal{R}}$ except for around the state/head non-deterministically check what rule it applied in the surrounding symbols. So we ensure that $w_{i-1}^{\mathcal{R}} \# w_i$ gives a valid step in the computation of the Turing machine. It is also easy to make the PDA check that the first part w_1 is the starting state and that w_n is in the accepting state.

The problem is that once we've done this we can't ensure that the terms between the # symbols are the same. Meaning the language we have decided is this:

$$L_1 = \{w_1 w_2^{\mathcal{R}} \# w_3 w_4^{\mathcal{R}} \# \ldots \# w_{n-1} w_n^{\mathcal{R}} \mid w_{i+1} \text{ follows from } w_i,\ w_1 \text{ initial state},\ w_n \text{ final state}\}$$

In this we have no way to guarantee that $\# w_{i-1}^{\mathcal{R}} w_i \#$ terms are in the form $\# w^{\mathcal{R}} w \#$, which was required by our original language. This is easy to see using the pumpling lemma for context-free languages. Fortunately for us though there is another context free language that does ensure this:

$$L = \{w_1 w_1^{\mathcal{R}} \# \ldots \# w_n w_n^{\mathcal{R}} \mid w_i \in \Sigma^*\}$$

This is can easily be shown to be context free. In fact here is the CFG:

$$\begin{aligned} S &\to A\#S|A \\ A &\to XAX|XX \\ X &\to \sigma \in \Sigma \end{aligned}$$

So what we want is the intersection of these two languages. If it is empty then there is no valid computational path on $\langle M, w\rangle$. So if we could decide if the intersection of two context-free languages was empty, we could decide A_{TM}, a contradiction. Therefore determining the if the intersection between two context-free languages is empty is undecidable.

Problem 4

(a) This can be done using dynamic programming.

Start by defining $T[q_0, 0] = 1$ to account for the only string of length zero (ϵ). All other $T[q, 0]$ for $q \neq q_0$ are set to 0.

Now, we can compute any $T[q, i]$ for $i \geq 1$ as follows:

$T[q, i] \leftarrow 0$
for all $q' \in Q$ **do**
  for all $a \in \Sigma$ **do**
    if $\{(q', a) \to q\} \in \delta$ **then**
      $T[q, i] \leftarrow T[q, i] + T[q', i - 1]$
    end if

end for
end for

(b) The answer to the problem is the number of strings of length n that are accepted by M, i.e. the ones that end in the accept state. So the answer is $\sum_{f \in F} T[f, n]$.

(c) The algorithm runs in time polynomial in the *value* of n, since increasing n by one requires $O(|Q| \cdot |\Sigma|)$ more steps. However, the value of n can be encoded in $\log(n)$ bits so the running time in the *size* of the input is exponential since 1 extra bit can hold up to a number double in value.

Problem 5

Show that the language

$$ISO = \{\langle G, H\rangle \mid G \text{ and } H \text{ are isomorphic graphs}\}$$

is in NP.

To do this we need to show a verifier V for ISO. As one might expect, the certificate for a member of the language $\langle G, H\rangle$ is the isomorphism $\phi : V_G \to V_H, v \to \phi(v)$ between them. The size of such an isomorphism would be about $2n$ where n is the number of nodes in G and H, so it polynomial in the size of G and H. We also need to check that the verifier V using this runs in polynomial time. A simple algorithm to use this is the following:

function $V(\langle G, H\rangle, \phi)$
  If the number of nodes or edges in G differ from the number of nodes or edges in H, reject
  for Nodes v in G **do**
    If the number of edges (v, u) in G differ from the number of edges $(\phi(v), u')$ in H, reject
    for Edges (v, u) in G **do**
      If $(\phi(v), \phi(u))$ is not in H, reject
    end for
  end for
  Accept
end function

By asserting that the size of the graphs are the same, that each node v has the same degree as $\phi(v)$ and maps to the corresponding vertices, we have shown that the isomorphism is correct. If there is no isomorphism then at least one of these tests will fail. Therefore $V(\langle G, H\rangle, y)$ will accept for some input y if and only if there is an isomorphism between G and H.

Problem 6

We can create an algorithm that reduces the length of b by 1 bit every iteration, therefore running in time polynomial in b. Let $b = b_1 b_2 ... b_n$ (in binary representation, where b_i is the i^{th} lowest order bit). The algorithm works as follows: We keep a running total of the current "answer" according to the bits we have seen so far. If the bit is not set we need to double our running answer (and modulo it) since it is evidence that the string is one bit longer which means twice as large. On the other hand, if the bit is set we must do the above but also multiply by a since this is adding a^i for bit i. This is known as "exponentiation by squaring."

The algorithm is as follows:

$x \leftarrow 1$
$l = \text{length}(b)$
for $i = 1...l$ **do**
  if $b_i = 0$ **then**
    $x \leftarrow x^2 \pmod p$
  else
    $x \leftarrow a \cdot x^2 \pmod p$
  end if
end for

if $x = c \pmod{p}$ **then**
    accept
else
    reject
end if

Note that the for loop runs $O(l)$, the *length* of b, times. In the loop, each multiplication can be done in time $O(n^2)$ in the length of the numbers using naive methods, and even faster using sophisticated ones. The same goes for division, being computable in $O(n^2)$ in the length of the numbers. Due to the above, modulo can be implemented in $O(n^2)$ as well. Therefore, the entire algorithm runs in polynomial time, in particular in $O(n^3)$ of the length of the longest number.

Problem 7

For a language A to be NP-complete, it needs to satisfy two conditions:

- $B \in$ NP
- $\forall L \in$ NP, $A \leq_p B$

We want to show that if P = NP, then every language $A \in P$ exept $A = \varnothing$ and $A = \Sigma^*$ is NP-complete.

The basic idea is that since we have polynomial time algorithms for all NP-complete problems like SAT, we map each language to SAT, solve it, then map that output to appropriate inputs for the original language, i.e. use $L \leq_p$ SAT then that $L \leq_p$ SAT (kind of a hack).

We know that SAT $\in NP$, and since by assumption P = NP, SAT $\in P$. That means there must be a polynomial time turing machine M_{SAT} that accepts SAT in polynomial time. We also know that SAT is NP-complete, so for every language $L \in$ NP there is a polynomial time function $f_L^{SAT} : \Sigma^* \to \Sigma^*$ that satisfies $w \in L \iff f_L^{SAT}(w) \in$ SAT.

Given a language $A \in$ P besides the empty and full languages, we know there exists strings $s_{acc} \in A$ and $s_{rej} \notin A$. Since we assume P = NP, we know $A \in$ NP. Using all of this we create the following map for a language $L \in$ P:

function $f_L^A(x)$
    Let $y = f_L^{SAT}(x)$
    Simulate $M_{SAT}(y)$, and record its output
    If it accepted, output s_{acc}
    Otherwise, output s_{rej}
end function

This machine satisfies $x \in L \iff f_L^A(x) \in A$ since it outputs $s_{acc} \in A$ if M_{SAT} accepted and $s_{rej} \in A$ if M_{SAT} rejected, and we know M_{SAT} accepts if and only if $f_L^{SAT}(x) \in$ SAT. We also know this runs in polynomial time, since $f_L^{SAT}(x)$ runs in polynomial time and $M_{SAT}(y)$ runs in polynomial time, and the rest are just a constant number of operations. Therefore this is a polynomial time map from arbitrary language L to arbitrary language A and so.

Since A satisfies both the conditions to be NP-complete, and A was any language in P (with the two exceptions), then any language in P is NP-complete if P = NP.