University of Virginia - cs3102: Theory of Computation Spring 2010

Problem Set 5 - Comments

Problem 1: Countable and Uncountable Infinities.

a. Show that the set of all strings in $\{a,b,c\}^*$ is countable.

Answer. We can enumerate all the strings, sorted alphabetically by their length:

$$\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, \ldots$$

This provides a one-to-one mapping between the set of strings and the natural numbers, and covers every string in the set.

b. An *algebraic* number is a number that is a root of some equation of the form $c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$ (that is, a value if x that makes the value of the equation 0). For example, ϕ (the golden ratio) is algebraic since it is a root of $x^2 - x - 1$ and $\sqrt{2}$ is algebraic since it is a root of $x^2 - 2$. But, π is not algebraic (proving this is difficult). Show that the set of all algebraic numbers is countable.

Answer. The question did not specify that the coefficients must be integers, which caused some confusion (this is normally assumed for polynomials). Note that the question becomes silly if the coefficients could be real numbers, since we could describe any real number as the root of $c_0 + x$ for some real number c_0. Note that it also doesn't make any difference if the coefficients can be rational numbers instead of integers. If the coefficients are rational, there is some equivalent polynomial since we can multiply by the least common multiple of the denominators.

To show that the algebraic numbers are countable, we need a one-to-one mapping between the natural numbers and the algebraic numbers. We can produce this by enumerating all the coefficients. One way to think of this is as a path through an infinite matrix, similarly to the proof that the rational numbers are countable. Suppose there were only two coefficients, c_0 and c_1. Then, we can cover all possible values with a one-to-one mapping: (each entry in the table is (c_0, c_1)):

0, 0	0, 1	0, 2	0, 3	...
1, 0	1, 1	1, 2	1, 3	...
2, 0	2, 1	2, 2	2, 3	...
3, 0	3, 1	3, 2	3, 3	...
...	...	...	...	...

A path that starts at 0, 0 and zig-zags through the matrix will cover all entries, and provides a one-to-one mapping between all polynomials of the form $c_0 + c_1x$.

To cover all polynomials, we need to extend this to any number of coefficients. This is still countable, since we can always add more dimensions to our space and still find a path that covers the whole space.

Another way to think of this is mapping to the set of strings in $0, 1, \#$ which we also know is countable (we're just replacing the symbols from part a). We can use the 0 and 1s to encode every integer, and the #s to separate the coefficients. The set of all strings includes all polynomials represented in this way, so this proves the number of polynomials is countable.

Since each polynomial has a finite number of roots and the number of polynomials is countable, the number of algebraic numbers must also be countable.

c. A *transcendental* number is a real number that is not algebraic. Are the transcendental numbers countable or uncountable? (Support your answer with a convincing argument.)

Answer. Transcendental numbers are *uncountable*. We know the real numbers are uncountable and the algebraic numbers are countable. If we remove a countable number of items from an uncountable set, the result must be an uncountable set. Otherwise, the original set would be countable, since the union of two countable sets must be countable (just interleave the sets to get the one-to-one mapping).

Problem 2: Language Sizes. Consider the language,

$$BIGGER_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } |L(A)| > |L(B)|\}$$

The notation $|L(M)|$ means the size of the language described by the machine M. The size of a language is the number of strings in the language. For purposes of this question, you should assume the definitions about set sizes from Definition 4.12.

Is $BIGGER_{DFA}$ decidable? Either prove that it is decidable (for example, by providing a high-level description of a Turing Machine that can decide it), or prove that it is undecidable (for example, but showing that a known undecidable problem can be reduced to it).

Answer. Yes, we can decide $BIGGER_{DFA}$. To prove it, we describe how to build a TM that decides $BIGGER_{DFA}$.

Note that Definition 4.12 says that if there is a one-to-one mapping between two sets they are the same size. Since the languages can be enumerated, if they are infinite they are the same size, and $\langle A, B\rangle$ is not in $BIGGER_{DFA}$. Thus, there are two possible ways for $\langle A, B\rangle$ to be in $BIGGER_{DFA}$: (1) $L(A)$ and $L(B)$ are both finite, and $|L(A) > |L(B)|$; and (2) $L(A)$ is infinite and $L(B)$ is finite.

To decide $BIGGER_{DFA}$ we first check if any of the given DFAs are infinite. This can be done using a strategy similar to the one used in Theorem 4.1 to check if the accepted language

of a DFA is infinite. If we can find a path from the start state that goes through a cycle and eventually reaches an accepting state, the language accepted by the DFA is infinite; if no such path exists, the language is finite. See the solution to Problem 4.9 (p. 185) for the details on one possible way to do this. Another way would be to try all possible strings of length $k+1$ to length $2k$. If the language is infinite, at least one of these strings must be accepted; if none of these strings are accepted, the language must be finite.

If both A and B accept infinite languages, then they are equal in size so the input string, $\langle A, B\rangle$ is not in $BIGGER_{DFA}$. If A is infinite, and B is not, then A is bigger (all infinite values are bigger than any finite value), so $\langle A, B\rangle$ is in $BIGGER_{DFA}$.

Otherwise, they are both finite. We need to count the strings in each language to determine which is bigger. We can do this — since we know both languages are finite, from the pumping lemma we know that they cannot accept a string longer than k where k is the number of states in the DFA. We can enumerate all strings up to length k (which can be different for A and B), simulating the DFA on each string, and counting the total number of strings that are accepted. If the number accepted by A exceeded the number accepted by B, then $\langle A, B\rangle$ is in $BIGGER_{DFA}$.

Problem 3: Closure Properties. For each part, provide a clear **yes** or **no** answer, and support your answer with a brief and convincing proof.

a. If A is a Turing-recognizable language, is the complement of A a Turing-recognizable language?

Answer. **No.** We have seen a counter-example already: A_{TM} is Turing-recognizable, but its complement is not. We know this, since we know A_{TM} is undecidable. If a language and its complement are both Turing-recognizable, that would mean we could construct a machine that decides that language.

If we had a machine, C_{TM}, that could recognize the complement of A_{TM}, we could use it to construct a machine that decides A_{TM}. We know there is a machine, R_{TM} that can recognize A_{TM}. Combining C_{TM} and R_{TM}, we could construction D_{TM} that decides A_{TM}. On input w, D_{TM} simulates both R_{TM} and C_{TM} running with input w, alternating between simulating a step for R_{TM} and simulating a step for C_{TM}. Note that either w is an element of $L(R_{TM})$ or w is an element of $L(C_{TM})$, since they are complements of each other. Thus, D_{TM} will eventually simulate a step where one of these machines accepts. If the machine that accepts is R_{TM}, then accept; if it is C_{TM}, then reject. Thus, D_{TM} is a decider for A_{TM}. But, we know that ATM is undecidable. Therefore, C_{TM}, a machine that recognizes the complement if A_{TM} cannot exist.

b. If A is a Turing-decidable language, is the complement of A a Turing-decidable language?

Answer. **Yes.** Since A is Turing-decidable, there is some TM M that decides A. We can construct a TM that decides the complement of A from M simply by swapping

q_{Accept} and q_{Reject}. Since M is a decider for A, it eventually reaches one of those states on any input. To decide the complement language, just reject when M would accept, and accept when M would reject.

c. If A and B are Turing-recognizable languages, is $A \cap B$ a Turing-recognizable language?

Answer. **Yes.** Let N_A and N_B be recognizers of A and B respectively. We can construct $N_{A\cap B}$ such that on input x it simulates $N_A(x)$ and $N_B(x)$, and accepts if and only if both N_A and N_B accept. If $x \in A \cap B$ then we know that both $N_A(x)$ and $N_B(x)$ halt (and accept), and so $N_{A\cap B}(x)$ will also halt and accept. This means $A \cap B$ is Turing-recognizable.

d. If A and B are Turing-decidable language, is $A \cap B$ a Turing-decidable language?

Answer. **Yes.** Since A and B are Turing-decidable, there exist TMs M_A and M_B that decide each language. We can build a machine that decides $A \cap B$ by simulating each machine in turn. First, we copy the original input to a spare tape. Then, simulate M_A on the input. Since it decides A, we know it will eventually halt. If it rejects, reject. If it accepts, clean up the working tape (write blanks over every square), and then copy the original input from the spare tape onto the working tape. Then, simulate M_B on the input. If it accepts, accept. If it rejects, reject.

Problem 4: Undecidability. Prove that each of the following languages is undecidable. (Hint: show that you can reduce a known undecidable problem to the problem of deciding the given language.)

a. $L_{INF} = \{< M > | M \text{ describes a TM that accepts infinitely many strings}\}$

Answer. We prove L_{INF} is undecidable by reducing A_{TM} to L_{INF}.

Assume L_{INF} is decidable. Then, there exists a Turing machine M_{INF} that decides L_{INF}. For an input $< M, w >$ we can construct the machine, M_w that simulates M on w (regardless of the actual input, which is ignored). If M would accept, M_w accepts; otherwise M_w rejects. Thus, if M accepts w, the language of M_w is infinite (it accepts all strings); otherwise it is empty (it accepts no strings). Hence, if M_w is in L_{INF} then M accepts w, otherwise M rejects w. But, we know A_{TM} is undecidable. Since we could use M_{INF} to decide A_{TM}, we know M_{INF} cannot exists and L_{INF} must be undecidable.

b. $L_{HelloWorld} = \{J | J \text{ is a Java program that prints out "Hello World"}\}$

Answer.

We prove $L_{HelloWorld}$ is undecidable by showing how we can reduce $HALT_{TM}$ to it. Let R be a TM that decides $L_{HelloWorld}$. We use R to construct TM S that decides $HALT_{TM}$.

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

1. Construct JM a Java program that simulates M on w, except when M would enter q_{Accept} or q_{Reject}, print "Hello World".
2. Run R on JM. If R accepts, accept; if R rejects, reject.

Hence, we could use a decider for $L_{HelloWorld}$ to decide $HALT_{TM}$ which we know is undecidable. Hence, $L_{HelloWorld}$ must be undecidable. Note that this assumes that the original Turing machine M never prints "Hello World". If we interpret *prints* to me writing output to a standard output stream (that is, not to the TM tape), then this is true, since TMs can only write to their tape. If we interpret prints as including writing "Hello World" on the TM's tape, then we would also need to worry about modifying M so that it never does this. One way to do this would be to extend the alphabet with a new symbol, $\hat{H}$, and replace all rules that write a H with rules that write a $\hat{H}$ instead, and add a corresponding rule for every rule that reads a H to do the same thing when a $\hat{H}$ is read.

Problem 5: Unmodifiable-Input Turing Machine. (Based on a question by Ron Rivest.) Consider a one-tape Turing Machine that is identical to a regular Turing machine except the input may not be overwritten. That is, the symbol in any square that is non-blank in the initial configuration must never change. Otherwise, the machine may read and write to the rest of the tape with no constraints (beyond those that apply to a regular Turing Machine).

$$HALT_{UTM} = \{< M, w > | M \text{ is an unmodifiable-input TM and } M \text{ halts on input } w\}$$

a. What is the set of languages that can be recognized by an unmodifiable-input TM? (Support your answer with a convincing argument.)

Answer. The *regular languages*.

Since the input is unmodifiable, there is no way for a unmodifiable-input Turing machine to keep track of where it is in the input. There are two ways it can keep track of state: (1) using the finite states in the state machine, or (2) using the tape. For option 1, the number of states is finite, so if the input is sufficiently long there is no way to keep track of all input positions with a finite number of states. For option 2, we can keep track of infinitely many positions using the tape, but to use the tape we need to move past the input to the writable part of the tape. But, but moving there to read or write the position, we have lost track of the actual position in the input since there is only one tape head! Hence, there is no way to keep track of the position in the input. This means that although we have infinite state to use on the rest of the tape, there is nothing useful that can be done with it, and the machine is equivalent to a DFA.

b. Is $HALT_{UTM}$ decidable (by a regular TM)? (Support your answer with a convincing proof.)

Answer. **No.** We prove it by reducing $HALT_{TM}$ to $HALT_{UTM}$.

For any $< M, w >$, we can create the UTM machine M_w that for any input $< M, w >$, first skips over the original input, writes a marker, and then writes w on the tape and then simulates M (treating the marker as the left edge of the tape). Then, if we had a machine that decides $HALT_{UTM}$ we could use it to decide $HALT_{TM}$ by running $HALT_{UTM}$ on input $< M_w, \epsilon >$ (the second input doesn't matter).

Problem 6: Self-Rejecting DFAs. (Challenge Bonus) Define the function, $D(w)$, for DFAs similar to the function $M(w)$ we defined for Turing Machines in Class 16:

$D(w)$ = If w is a valid encoding of a DFA, the DFA described by w. Otherwise, a DFA that rejects all inputs.

Is the language SELF-REJECTING$_{DFA} = \{w \in \Sigma^* : w \notin \mathcal{L}(D(w))\}$ Turing-recognizable? Provide a convincing proof supporting your answer.

Answer. We could use a diagonalization proof similar to the one we used for SELF-REJECTING$_{TM}$ to show that SELF-REJECTING$_{DFA}$ is not decided by any DFA, but this does not answer the question of whether or not it can be recognized by a TM. In fact, it is not only Turing-recognizable but is also Turing-decidable. Here is a description of a TM that decides SELF-REJECTING$_{DFA}$:

$M_{SRDFA}(w)$ = Simulate $D(w)$ running on w and do the opposite. (If it accepts, reject. If it rejects, accept.)

We know that $D(w)$ can be computed by a TM, and that a TM can simulate any DFA. For a given input, a DFA always takes a finite number of steps, so the simulation of $D(w)$ always terminates and either accepts or rejects depending on whether the DFA accepts or rejects.

Problem 7: Random Access Memory. Random access memory (misnamed, since it is not at all random) allows a program to directly reference specified memory locations. Assume each memory location can store a single byte (8 bits) value. A random access machine (RAM) provides (at least) these two instructions:

1. **store *location value*** — write the value represented by the current square on the tape into location *location*. The *location* is any 32-bit integer, and *value* is any 8-bit value (represented by a single square on the tape).
2. **load *location*** — read the value in the location *location* and write it onto the current square on the tape. At the end of a load instruction, the square under the tape head should contain the read value. The read value should be the last value that was stored in *location* (using a **store** instruction), or 0 if no value has been stored in *location*.

a. Prove that a RAM is not more powerful than a Turing machine by showing how a TM could simulate a RAM. Your proof should include high-level descriptions of a Turing

Machine that can simulate the **load** and **store** instructions. Your description should also explain how you represent memory on the tape.

Answer. Since we already know a regular TM can simulate a multi-tape TM (see Sipser's Theorem 3.13), we will simplify our answer by describing a 2-tape TM. Note that we are not concerned with efficiency, just with proving that a TM can simulate the **load** and **store** instructions. We will use Tape 1 as the working tape, and Tape 2 as the store. Initially, Tape 2 will be empty (all blanks). We will record stored data on Tape 2 as $< location, data >$ pairs, recording the location in the first squares, and value in the second square. (Since location is defined to be a 32-bit integer, there are a finite number of locations, so we can use a different tape symbol for each location (e.g., the 32-bit number). Similarly, the data values are 8-bit values, so can be recorded using a single symbol.)

Now, to simulate a **store**: scan Tape 2 from left to right. If any record has a location matching the store location, replace the data value in the following square with the new data and transition to the finished (with the store) state. Otherwise, if a blank is reached, replace that blank with the store location, move right, and write the data value on the next square.

To simulate a **load**: scan Tape 2 from left to right. If any record has a location that matches the load location, move right one square and read the data, writing it into the current square on Tape 1.

b. Is the programming language consisting of just the **load** and **store** instructions above a *universal programming language*? (That is, is it possible to express all possible algorithms using this language.) If it is, prove it (by explaining how you could implement a universal Turing Machine using the **load/store** language. If it is not, explain convincingly why not, and describe the simplest modifications needed to make the language a universal programming language.

Answer.

No, it is not a universal programming language. For example, there is no way to implement an infinite loop using just the **load/store** language.

To make a universal programming language, we would first need to remove the restriction on the number of locations — with the 32-bit restriction, the **load/store** can only store a finite number of values, so is comparable to a DFA. If we change the location to be any natural number, then we have an unlimited number of memory locations, so can use it to simulate the tape of a Turing Machine. We still don't have enough to simulate a TM with just **load** and **store**, however. We also need some way to simulate the FSM controller. We need to add instructions that can be used to make decisions, and to enable the machine to keep going.

In fact we only need *one* opcode to be able to do all these things, but it needs three parameters. All instructions are of the form:

subleq *a b c*

Where a, b, and c are memory locations. The meaning of the **subleq** instruction is (1) replace the value in location b with $value[b] - value[a]$, then (2) if the value stored into location b is 0, jump to location c (for the next instruction). Otherwise, advance one location to the next instruction.

In fact, a TM could be simulated using only **subleq** instructions, so even without the **load** and **store** instructions it is a *universal programming language*! Although you could write all programs using just **subleq** instructions, for readability and efficiency reasons it is not recommended! (See *http://mazonka.com/subleq/* for some examples.)

Problem 8: Minds and Machines. Many people find the suggestion that a human mind is no more powerful than a Turing Machine to be disturbing, but there appear to be strong arguments supporting this position. For example, consider this argument:

> The brain is a collection of 100 billion neurons. Each neuron is a cell that has inputs (known as *dendrites*) and outputs (synapses that emit neurotransmitter chemicals). The output depends on the inputs in a deterministic way that could be simulated by a Turing Machine. The connections between neurons could also be simulated by a Turing Machine. Since all components of the brain could be simulated by a Turing Machine, the brain itself could be simulated by a Turing Machine. Hence, a human mind is no more powerful than a Turing Machine.

Write a short essay that counters this argument (although many books have been written on this question, you should limit your response to no more than one page). If you reject the premise of this question either because you do not find it disturbing to think of your mind as a Turing Machine, or you feel that the only way to counter this argument is to resort to supernatural (e.g., religious) notions, you may replace this question with Sipser's Problem 5.13.

Answer. There is, of course, no solution to this question. I would like to see answers that identify some of the difficulties in simulating a brain with a TM - for example, maybe simultaneity actually matters, or the continuous nature of chemicals cannot be represented discretely. Other good answers might estimate the size of a TM needed for an accurate brain simulator, and argue that it would be equivalent to a brain, but actually building a completely accurate simulator would involve more atoms than are available in the universe.