

Problem Set 5 - Undecidability

Problem 1: 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. Show that a Turing Machine can simulate random access memory. Your machine should be able to simulate these two instructions:

1. **store** $\langle location \rangle$ — write the value represented by the current square on the tape into location $\langle location \rangle$. The $\langle location \rangle$ is any 32-bit integer, and $\langle value \rangle$ is any 8-bit value (represented by a single square on the tape).
 2. **load** $\langle location \rangle$ — read the value in the location $\langle location \rangle$ 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 $\langle location \rangle$ (using a **store** instruction), or 0 if no value has been stored in $\langle location \rangle$.
- a. Provide an implementation-level description of a Turing Machine that can simulate the **load** and **store** instructions. Your description should explain how you represent memory on the tape and provide implementation-level descriptions of how you simulate the two instructions.

Answer:

Since we have already shown how a regular TM can simulate a multi-tape TM, 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 $\langle location, data \rangle$ 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 non-negative integer, 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.

A TM could be simulated using only **subleq** instructions, so 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 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.

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 construct 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. Since A and B are Turing-recognizable, there exist TMs M_A and M_B that recognize each language. We can build a machine that recognizes $A \cap B$ by simulating each machine. First, we copy the original input to a spare tape. Then, simulate M_A on the input. Since it recognizes A , if the input is in A it will eventually accept. If it accepts, then erase the working tape with all blanks, copy the original input from the spare tape onto the working tape, and simulate M_B on the input. The result is the result to M_B . Note that it is okay that the simulate of M_A may never finish — all we need to construct is a TM that recognizes $A \cap B$. If M_A would not terminate, then the simulator doesn't terminate. This is consistent with the input not being in the language $A \cap B$.

- 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. $L_{INF} = \{ \langle M \rangle \mid M \text{ describes a TM that accepts infinitely many strings} \}$

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 $\langle M, w \rangle$ 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 exist and L_{INF} must be undecidable.

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

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.

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).

$HALT_{UTM} = \{ \langle M, w \rangle \mid M \text{ is an unmodifiable-input TM and } M \text{ halts on input } w \}$

- 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. (Discussed in Class 19).

- 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 $\langle M, w \rangle$, we can create the UTM machine M_w that for any input $\langle M, w \rangle$, 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 $\langle M_w, \epsilon \rangle$ (the second input doesn't matter).