Recitation - Turing Machines

1 Announcements

Be sure to take advantage of the following resources :

- Homework Solution Sessions: Saturdays 15:30 - 16:30, Sundays 12:30 - 13:30.
- Weekly Review Sessions: Saturdays 12:00 - 13:00.
- Get to know your mentor and reach out to them if you need help - that's what they're here for!

2 Too Many Definitions

- Informally, a Turing machine is a finite state machine with access to a tape (memory) that is either infinite in one direction or infinite in both directions (in this course, usually infinite in both directions). Initially the input to the TM is placed on this tape. At each step, the machine makes the following decisions (based on the state it is in and the symbol it's tape-head is currently reading): change the state, write some symbol at the cell currently under the tape head, and move the tape head once cell to the left or to the right.
- Formally, we define a Turing machine to be a 7-tuple $(Q, q_0, q_{\text{acc}}, q_{\text{rej}}, \Sigma, \Gamma, \delta)$, where Q is the set of states, $q_0 \in Q$ is the start state, $q_{\text{acc}} \in Q$ is the accepting state, $q_{\text{rej}} \in Q$ is the rejecting state, Σ is the input alphabet, $\Gamma \supseteq \Sigma \cup \{\sqcup\}$ is the tape alphabet, and $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$, where $Q' = Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$ is the transition function. Once a TM reaches q_{acc} or q_{rej}, it halts, and it will keep going until it reaches one of these states.

- Unlike DFAs, TMs need not halt on all inputs. A Turing machine is called a *decider* if for all inputs $x \in \Sigma^*$, it halts and either accepts or rejects x.
- A language $L \subseteq \Sigma^*$ is called *decidable* if there exists a Turing machine M such if $x \in L$, M accepts x, and if $x \notin L$, M rejects x. Note that such a TM M halts on all inputs, and therefore is a decider TM.
- For a TM M, $L(M)$ denotes the set of all strings that M accepts. Note that for strings not in $L(M)$, M either rejects or loops forever.
- The following is an equivalent definition of decidability: $L \subseteq \Sigma^*$ is called *decidable* if there exists a **decider** Turing machine M such that $L = L(M)$.
- Let L and K be languages, where K is decidable. We say that solving L *reduces* to solving K (or simply, L reduces to K, denoted $L \leq K$), if we can decide L by using a decider for K as a subroutine (helper function).

3 Closure Ceremony

Suppose that L_1 and L_2 are decidable languages. Show that the languages $L_1 \cdot L_2$ and L_1^* are decidable as well by constructing decider TMs for them (using high-level descriptions). Proof of correctness is not required.[1]

Solution. https://www.youtube.com/embed/GbTOpeGsFw8

Let `M1` and `M2` be two Turing machines that decide L_1 and L_2 respectively.

We construct Turing machines `M3` and `M4` that decide $L_1 \cdot L_2$ and L_1^* respectively:

```
def M3(x):
    for each of the |x| + 1 ways to divide x as yz:
        simulate M1 on y
        if M1 accepts:
            simulate M2 on z
            if M2 accepts, accept
    reject
```

```
def M4(x):
    if length of x is 0:
        accept
    for each sorted list of indices [0, a1, a2, ..., |x|]:
        // the indices are a subset of {0, 1, 2, ..., |x|}
        // each list starts with 0 and ends with |x|
        string_is_good = true
        for each ordered pair of adjacent indices (p, q):
            simulate M1 on x[p:q]
            // x[p:q] is the section of x from the pth to the q-1 th character
            if M1 accepts:
                pass // i.e. keep executing
            else:
                string_is_good = false
                break
        if string_is_good:
            accept
    reject
```

Note that we've implicitly appealed to the Church-Turing thesis here. One implication of the Church-Turing thesis is that for any algorithm with precise instructions written in English, there is a corresponding TM that does the same thing. We have written our algorithms in English to show the existence of two Turing machines. ■

[1] Exercise : show that $L_1 \cup L_2$ and $L_1 \cap L_2$ are also decidable.

4 Freeze All Automata Functions

We define the language $\text{EMPTY}_{\text{DFA}}$ as follows:

$$\text{EMPTY}_{\text{DFA}} = \{\langle D \rangle : D \text{ is a DFA with } L(D) = \varnothing\}.$$

Prove that the following languages are decidable by reducing it to $\text{EMPTY}_{\text{DFA}}$.

1. NO.ODD =

   $\{\langle D \rangle : D$ is a DFA that does not accept any string containing an odd number of 1's$\}$

2. $\text{INF}_{\text{DFA}} = \{\langle D \rangle : D \text{ is a DFA with } L(D) \text{ infinite}\}$.

Solution. https://www.youtube.com/embed/au0p8xswM5o

Part 1: Let ODD be the language of all strings with an odd number of 1's. We leave it as a short exercise to show ODD is regular by drawing a DFA.

Let $M_{\text{EMPTY-DFA}}$ be a decider for $\text{EMPTY}_{\text{DFA}}$. We construct a decider M for NO.ODD as follows.

1. Given is an input $\langle D \rangle$ where D is a DFA.
2. Construct a DFA D' such that $L(D') = L(D) \cap \text{ODD}$.
3. Run $M_{\text{EMPTY-DFA}}$ on $\langle D' \rangle$ and return the answer.

Proof of correctness:

To prove the correctness, we need to argue that no matter what the input is, our TM M returns the correct answer. We do so in two cases.

Suppose that $\langle D \rangle \in$ NO.ODD. Then $L(D) \cap \text{ODD} = \varnothing$. So $M_{\text{EMPTY-DFA}}(\langle D' \rangle)$ (and therefore $M(\langle D \rangle)$) will accept as desired.

Now suppose that $\langle D \rangle \notin$ NO.ODD. Then $L(D) \cap \text{ODD} \neq \varnothing$. So $M_{\text{EMPTY-DFA}}(\langle D' \rangle)$ (and therefore $M(\langle D \rangle)$) will reject as desired.

(*Note that there is also the case when the input string does not correspond to a valid encoding of a DFA. Even though this is not explicitly written, we implicitly assume that the first thing our machine does is check whether the input is a valid encoding of an object with the expected type. If it is not, the machine rejects. If it is, then it will carry on with the specified instructions. The important thing to keep in mind is that in our descriptions of Turing machines, this step of checking whether the input string has the correct form (i.e. that it is a valid encoding) will never be explicitly written, and we don't expect you to explicitly write it either.*)

Part 2: We first prove the following lemma: If a DFA with k states accepts some string w with $|w| > k$, then it will accept infinitely many strings.

Proof of lemma:

Let D be a DFA with k states accepting the string $w = w_1 w_2 \dots w_n$, where $n > k$. By PHP, there exists i and j, $i < j$, such that $w_1 \dots w_i$ and $w_1 \dots w_j$ end on the same state $q \in Q$. It follows that $w_1 \dots w_i (w_{i+1} \dots w_j)^m$ ends on q for all $m \in \mathbb{N}$. So, all the strings of the form $w_1 \dots w_i (w_{i+1} \dots w_j)^m w_{j+1} \dots w_n$ for $m \in \mathbb{N}$ are accepted. This completes the proof of the lemma.

We now move onto the main claim. We construct a decider M for INF_{DFA} as follows:

1. Given input $\langle D \rangle$ where D is a DFA, let k be the number of states of D.
2. Construct a DFA D' such that $L(D') = \{w \in \Sigma^* : |w| > k\}$. (We leave the proof that this language is regular as an exercise).
3. Construct a DFA D'' such that $L(D'') = L(D') \cap L(D)$.
4. Let $M_{\text{EMPTY-DFA}}$ be a decider for $\text{EMPTY}_{\text{DFA}}$ and run $M_{\text{EMPTY-DFA}}$ on $\langle D'' \rangle$.
5. If $M_{\text{EMPTY-DFA}}$ accepts, reject. Else accept.

To prove the correctness, we need to argue that no matter what the input is, our TM M returns the correct answer. We do so in two cases.

Suppose $\langle D \rangle \in \text{INF}_{\text{DFA}}$. This means $L(D)$ is infinite. Since there are only finitely many strings of length at most k, D accepts some string of length greater than k. So $L(D'')$ is nonempty. This means $M_{\text{EMPTY-DFA}}(\langle D'' \rangle)$ rejects, which means $M(\langle D \rangle)$ accepts, as desired.

Suppose $\langle D \rangle \notin \text{INF}_{\text{DFA}}$. This means $L(D)$ is finite. Then by the contrapositive of our lemma it accepts no strings of length greater than k. It follows that $L(D'') = \varnothing$, so $M_{\text{EMPTY-DFA}}(\langle D'' \rangle)$ accepts and $M(\langle D \rangle)$ rejects, as desired. ■

5 Not Just Your Regular Old TM

Suppose we change the definition of a TM so that the transition function has the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, S\}$$

where the meaning of S is "stay". That is, at each step, the tape head can move one cell to the right or stay in the same position. Suppose $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is a TM of this new kind, and suppose also that M is a decider. Show that $L(M)$ is a regular language by describing a DFA solving $L(M)$. A proof of correctness is not required.

Solution. https://www.youtube.com/embed/Cj7hCX88b6I

We construct a DFA $(Q', \Sigma, \delta', q_0', F')$ that solves the language $L(M)$. The components of the 5-tuple will be specified below.

We say that a state is *pointless* if it can never be reached (no matter what the input is). Let M' be M with all the pointless states removed. Note that M' has the exact same behavior as M on all inputs. Therefore it suffices to prove $L(M')$ is regular. The reason for M' will become clear when we define the transition function of our DFA. We drop the prime and refer to M' as M from here.

We now specify how the components of our DFA are defined.

We let $Q' = Q$. And of course the input alphabet does not change. Furthermore, the initial state of the DFA will be the same as the initial state of the TM.

To define the transition function $\delta' : Q' \times \Sigma \to Q'$, we have to define what the output should be for an arbitrary $(q, a) \in Q' \times \Sigma$. To make our definition, we'll imagine the TM M is in state q when the tape head first moves right onto a symbol a of the input. There are several cases to consider.

Case 1: If (q, a) is such that $q \in \{q_{\text{acc}}, q_{\text{rej}}\}$, then define $\delta'(q, a) = q$. So the q_{acc} and q_{rej} states are sink states in the DFA.

Case 2: If we are not in case 1, then we consider whether the transition $\delta(q, a)$ moves the tape head to the right or stays. Let's consider what happens when the head moves right. In this case, the DFA's transition will proceed similarly to the TM and change the state in the same way. What is written on the tape by the TM is not important as we can never go back and read it. So if $\delta(q, a) = (q', b, R)$, then define $\delta'(q, a) = q'$.

Case 3: Suppose $\delta(q, a)$ does not move the tape head, i.e., suppose $\delta(q, a) = (q', b, S)$. In this case one of two things can happen. Either the machine reaches an accepting or rejecting state before ever moving the tape head right. Or the machine will eventually move the tape head right and enter a state q''. If the machine reaches an accepting state, define $\delta'(q, a) = q_{\text{acc}}$. If it reaches a rejecting state, define $\delta'(q, a) = q_{\text{rej}}$. Otherwise, define $\delta'(q, a) = q''$.

This completes the definition of the transition function δ'.

There is only one thing left, which is to define F'. We will have $q_{\text{acc}} \in F'$, but F' may contain other states as well. In particular, let's call a state q *good* if running M on the blank input, starting at q, ends at q_{acc}. Then we put all the good states in F'. Take a moment to understand why we are defining F' in this way. Think about what behavior we want once the TM reaches the end of the input and is pointing to the first blank symbol. At that point it will be at some state q, and note that we are calling q a *good* state if the TM eventually reaches the accepting state from that point on. ■

6 (Extra) Can You Recognize the Decider

Fix an alphabet Σ. We say that a language $L \subseteq \Sigma^*$ is *semi-decidable* (or *recognizable*) if there is a TM M such that $L = L(M)$. (Recall that a language L is decidable if there is a *decider* TM M such that $L = L(M)$.) So for all $w \in L$, M accepts, and for all $w \notin L$, M either rejects or loops forever. Show that L is decidable if and only if L and $\overline{L} = \Sigma^* \backslash L$ are semi-decidable.

Solution. https://www.youtube.com/embed/Sk5qRaNngzQ

We first prove that if L is decidable, then L and $\overline{L} = \Sigma^* \backslash L$ are semi-decidable. So suppose L is decidable. Note that if a language is decidable, then it is automatically semi-decidable. Therefore we know L is semi-decidable. So all we need to argue is that $\overline{L}$ is semi-decidable. Observe that decidable languages are closed under complementation: if we take a decider M for L, and reverse the accepting and rejecting states, then we'll have a decider for $\overline{L}$. This shows $\overline{L}$ is decidable, and therefore semi-decidable.

We now prove that if L and $\overline{L} = \Sigma^* \backslash L$ are semi-decidable, then L is decidable. Let M be a semi-decider for L and let M' be a semi-decider for $\overline{L}$. We want to describe a decider TM N for L. As a first attempt, consider the following TM.

```
def N(x):
    run M(x)
    if it accepts, accept
    run M'(x)
    if it accepts, reject
```

What is wrong with the above TM? Why doesn't it correctly decide L?

Here is a description of a correct decider N for L.

```
def N(x):
    for t = 1, 2, 3, ...:
        simulate M(x) for t steps
        if it halts and accepts, accept
        simulate M'(x) for t steps
        if it halts and accepts, reject
```

To see that this is indeed a correct decider for L, let's consider what happens for all possible inputs.

If the input x is such that $x \in L$, then we know that since M is a semi-decider for L, $M(x)$ accepts after some number of steps. We also know that M' only accepts strings that are not in L, and therefore does not accept x. So $N(x)$ will correctly accept.

If on the other hand x is not in L, then we know that x is in $\overline{L}$. Since M' is a semi-decider for $\overline{L}$, $M'(x)$ accepts after some number of steps. We also know that M only accepts strings that are in L, and therefore does not accept x. So $N(x)$ will correctly reject. ■

7 (Extra) Only $19.99! Call now!

Dr. Hyper Turing Machines Inc LLC is selling a whole host of new Turing machines, each for $19.99:

- Bi-infinite TMs - with a tape that stretches infinitely in both directions!
- Infinitely-scalable TMs - choose however many tapes heads you like!
- Quad-core TMs - now with 4 tapes (each with its own tape head)

Normal TMs usually go for $9.99 these days. Your friend (who's not very Turing-savvy) is in the market for a new Turing machine and just texted you asking you for purchasing advice. Your instincts tell you that maybe most of this is marketing hype. But some of those improvements do sound pretty compelling... Your friend doesn't use their TM for all that much - mostly just browsing the web and checking email. What should you recommend them to do?

Solution. https://www.youtube.com/embed/M-UPSalle48

If your friend isn't concerned about performance, they should just buy a normal TM. All of the above are equivalent to normal TMs. (Note that the descriptions below are informal. We are not after formal proofs in this question, but rather just build some intuition.)

- We can easily simulate a bi-infinite tape TM with a singly-infinite one. The idea is to index the cells on a bi-infinite tape with the integers and the cells of the singly-infinite tape with the naturals, and then perform the usual bijection. Given input x, we first space out x by inserting one space between each pair of adjacent characters. After this, we can simulate a move of the tape-head by moving two cells in the same direction (if it is on an odd-numbered cell), and by moving two cells in the opposite direction (if it is on an even-numbered cell). The only exception to this scheme is when we are on the first cell. If we move left from the first cell in the bi-infinite TM, we just move one cell to the right in the singly-infinite TM. How do we know that we are on the first cell? We 'mark' the symbol on the first cell at the very beginning, and keep it marked. We achieve this by adding a symbol a' for every symbol a in the alphabet.

- If M is a k-tape-head TM, then we describe a TM S that simulates M. We add to our tape alphabet the ability to 'dot' symbols (i.e. for each symbol a, add the symbol a'), demarcating where M's tape heads are. To simulate a single move of M, S first makes a pass to determine what symbols are underneath M's tape heads. S then makes a second pass and performs the appropriate transitions for each virtual tape head.

- In a 4-tape TM, we have 4 separate tapes each having their own tape heads. A single application of the transition function reads the symbols under the tape heads, changes them, and then moves left or right. What is written on the tapes and which direction the tape heads move are determined separately for each tape.

  We can simulate a 4-tape Turing machine M with a normal 1-tape Turing machine S. At step t of the computation suppose $\cdots \sqcup\sqcup\sqcup w_i \sqcup\sqcup\sqcup \cdots$ is the contents of tape i, where w_i is some string in Γ^*. We keep the contents of the 4 tapes on a single tape, with a # as a separator:

  $$\#w_1\#w_2\#w_3\#w_4\#$$

  The first # is at index 0 of the tape. We expand the tape alphabet to allow ourselves to "dot" a symbol (this keeps track of where the "virtual tape heads" are).

  To simulate a single move, S scans from the 1st # to the 5th # in order to figure out what symbols lay underneath the tape heads. S then performs a second pass to update the tapes according to the way M's transition function dictates. An edge case that we need to handle : if S ever moves one of the virtual heads onto a #, this signifies that M has moved one of its heads onto previously unread tape cell. In this case, S copies everything to the right of the tape head over 10 spots and writes blanks on the new space generated (this is kind of like requesting memory via `malloc`).

  The fact that Turing machines can be changed in so many ways and remain equivalent in power provides evidence that the Turing machine is a quite robust model of computation. This robustness may help justify Turing machines as a reasonable abstraction of computation.

■