Handout 8A

Anum Ahmad and Zachary Thayer

COMS 3261 Fall 2024

1 Turing Machines

1.1 Overview/Intuition

A Turing Machine is so powerful it can simulate any other "reasonable" model of computation. Here are some key changes from our previous weaker models which are useful to keep in mind.

- **Infinite Tape**: In the standard model we have our input written on the tape, followed by an infinite number of empty space characters. This gives us infinite memory! When intuitively considering if a TM can solve a problem, remembering that we have access to memory can make it simple (e.g. the problem of checking if a string contains exactly 67 ones: intuitively we know this can be done by incrementing a count every time we process a 1, then checking that count equals 67).
- **Read/Write Capability**: In addition to read, we can now write as well. The ability to write has unlocked the capability to perform computations and store intermediate results. (Another intuitive way to consider if a language can be decided by a TM is asking if a python program could solve it).
- **Unrestricted Movement**: The tape head can move both left and right, allowing the TM to access any position on the tape multiple times.

1.2 Examples

We start with the formal definition of Turing Machines. We say M is a Turing Machine if it can be written as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$$

1. Q= Finite set of states
2. Σ = Finite Alphabet
3. Γ = Tape Alphabet (Σ along with blank symbol and possibly other special symbols)

4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ transition function
5. q_0= start state
6. q_{acc} = accepting state
7. q_{rej} = rejecting state

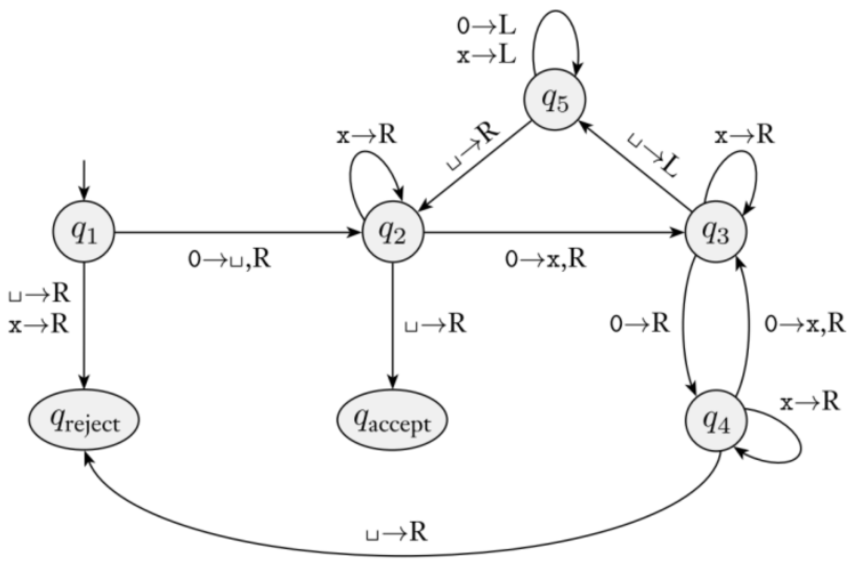
To see an example of a Turing Machine given formally, let $L = \{w \in \{0,1\}^* :$ w contains $1\}$. We construct a TM M recognizing this as follows:

1. Q= $\{q_0, q_{acc}, q_{rej}\}$
2. $\Sigma = \{0, 1\}$
3. $\Gamma = \{\Sigma \cup \text{␣}\}$
4. q_0, q_{acc}, q_{rej} given
5. $\delta(q_0, 0) = (q_0, 0, R)$, $\delta(q_0, 1) = (q_{acc}, 1, L)$, $\delta(q_0, \text{␣}) = (q_{rej}, \text{␣}, R)$

Do you see how this TM works? Essentially we are able to "halt" the TM once we see a 1 on the input and reach an accepting state.

We now give a more complex example: Let $A = \{0^{2^n} : n \geq 0\}$. We construct a TM M deciding A as follows: $\{Q, \Sigma, \Gamma, \delta, q_1, q_{acc}, q_{rej}\}$

1. Q= $\{q_1, q_2, q_3, q_4, q_5, q_{acc}, q_{rej}\}$
2. $\Sigma = \{0\}$
3. $\Gamma = \{0, X, \text{␣}\}$
4. Starting state is q_1,
5. q_{acc} as given
6. q_{rej} as given
7. We give the transition function with a state diagram:



We can explain this formal transition diagram in the following: If the string is empty, q_1 rejects it. Otherwise, a blank is placed in the first zero and moved to state q_2. From there, the states go between marking off 0's and leaving them blank as it traverses the string. If there were an even number of 0's it would end up at q_3 and traverse to the left through q_5 to q_2, where the process is repeated. On the other hand, if there is an odd number (other than 1), it will end up at q_4 and reject.

Then M recognizes (and in fact decides!) L.

As we can see, this method of constructing TMs is very cumbersome, so we often just give implementation-level or high-level descriptions. Implementation level descriptions are more of a "medium" level of formalism, where we still explain what we want the read/write head to do on the input tape, but we don't give an explicit 7-tuple and use more English sentences. Here is such an example:

Let A be defined as above; we give an implementation-level description of a TM M deciding A as follows– on input w:

1. Sweep left to right across the tape, crossing off every other 0
2. If in stage 1 we find that the tape just had one 0, accept.
3. If in stage 1 the tape contained an odd number of zeroes, and the number was greater than 1, reject.
4. Return tape head to left end of tape.
5. Go back to step 1.

Remark. One should note that for implementation-level descriptions, one can also use Non-deterministic or Multitape Turing Machines. We expand on these models in section 2.

Finally, the least formal description of a Turing Machine would be a "high-level" description, where we present how we want our Turing Machine to act using an algorithm, and we forgo considering how the read/write head should act on the input tape. Here are some examples of this:

Returning to the same language A we have defined above, a high-level description for the decider M would be– On input w:

1. Check if w is of the form 0^{2^n} for some n.
2. If the check implies w is of the correct form, accept. Otherwise, reject.

2 Two Variants of Turing Machines

While the standard Turing Machine is a powerful model, several variants exist that offer alternative perspectives but are equivalent in computational power. Namely the following:

2.1 Multi-Tape Turing Machines

This is a variant of Turing machines in which we fix some number of tapes and corresponding tape heads. At each step, the machine reads symbols from all tapes, writes symbols, and moves each tape head independently. This model can be useful for many circumstances, such as alternating between separate computations, comparing strings, copying a string to save it for later, organization, doing a fixed number of things in parallel, etc.

Example We're interested in proving recognizable languages are closed under the AtLeastOne operation, which we define as follows. Let L be some TM-recognizable language. Then AtLeastOne(L) $= \{< x\#y > \ | \text{ at least one of } x, y \text{ is in } L\}$. That is, the elements of AtLeastOne(L) are two strings separated by a #, with at least one of the two strings in the original language. Note that L has alphabet Σ, with $\# \notin \Sigma$, and AtLeastOne(L) has alphabet $\Sigma \cup \{\#\}$ (we're adding # to the new language's alphabet to be used as a separator for the inputs).

> Let M be the TM recognizing L, here's our multitape TM M', with 2 tapes, for AtLeastOne(L):
> on input $w = x\#y$, do the following:
> 1. Write everything before the # on one tape (this is x) and everything after on another (this is y).
> 2. Simulate 1 step of M (the machine that recognizes L) on tape 1 (x), if the machine accepts, accept w.
> 3. Simulate 1 step of M on tape 2 (y), if the machine accepts, then

accept w
4. Return to step 2.

Here's why this works:
If w is in AtLeastOne(L), this means x or y is in L. Since M recognizes L, it will eventually accept on one of the tapes, and thus M' will accept w.
If w is not in AtLeastOne(L), then neither one of x or y is in L. So, M will never accept on either tape, and we run indefinitely, thus M' doesn't accept w.

Note we could add rejecting criteria such as if both tapes are in a reject state output reject, but running forever is fine when the input is not in the language. To clarify, as we've written it, if both tapes say reject, steps 2 and 3 alternate with no change indefinitely.
Also note: It would be incorrect to say "simulate M on x, then M on y. If M accepts x we accept, otherwise move on to y where if it accepts there we accept. The key issue here is that M might run forever on x, so we'll never get to try y. That's why we were clever and alternated simulating M on both x and y one step at a time.

If we were trying to prove that the class of decidable languages is closed under this operation, then the above simpler algorithm would work, as if M is a decider, it is guaranteed to halt on every input, so the above would also be a decider (although, as an aside, note that we don't have a way to determine whether a given machine is a decider or not, as we will see).

Alternate Solution with NTM. Note a recap of NTM's are below in **2.2**. Our NTM N operates as follows:
on input $w = x\#y$, do the following:
1. Nondeterministically choose x or y, and simulate M on whichever was chosen, then output the result of M.

We see that if $w \in$ AtLeastOne(L), then either x or y is in L, so if we luckily choose the string which is in L, M will accept and so will N.
If $w \notin$ AtLeastOne(L), then there is no guess/choice that will make N accept.

2.2 Non-Deterministic Turing Machines

Non-deterministic Turing Machines can have multiple possible moves from a given configuration. They accept an input if and only if there exists a sequence that leads to the accept state. Nondeterminism is a very powerful (and sometimes unintuitive) concept. That said, it can make designing an algorithm simpler, since you can "guess" (at nondeterministic steps), and then just verify after the fact that the input is in the language if and only if with the right/lucky guess(es) the NTM accepts.

Example We give an NTM to decide the following language:
REACHABILITY $= \{\langle G, s, t\rangle \mid G$ is a graph, s, t are vertices, there exists a path in G from s to $t\}$

> Our NTM N operates as follows:
> On input $w = \langle G, s, t\rangle$, where G is a graph and s, t are vertices in G, do the following:
> 1. Set the current vertex to s.
> 2. Mark the current vertex as visited.
> 3. If the set of vertices connected to the current vertex by an edge are all marked, reject. Otherwise, Nondeterministically set the current vertex to be an unmarked vertex in that set.
> If the current vertex is t, accept w. Otherwise return to step 2.
>
> Why this works: Suppose that $w \in$REACHABILITY: then by definition of REACHABILITY there exists a path, or sequence of vertices connected by edges from s to t. Thus, there exists a sequence of lucky choices N can make to reach t from s, so N accepts w.
> Suppose that $w \notin$ REACHABILITY: then there exists no such guessable path from s to t, so all possible guesses make N reject w.

3 Practice Problems

1. Consider the input-output TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{halt})$ where $Q = \{q_0, q_1, q_{halt}\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \textvisiblespace\}$, and δ is given by:
$\delta(q_0, 0) = (q_0, 0, R)$, $\delta(q_0, 1) = (q_0, 1, R)$, $\delta(q_0, \textvisiblespace) = (q_1, \textvisiblespace, L)$
$\delta(q_1, 0) = (q_{halt}, 1, R)$, $\delta(q_1, 1) = (q_1, 0, L)$, $\delta(q_1, \textvisiblespace) = (q_{halt}, \textvisiblespace, L)$

   (a) Provide the complete sequence of configurations of M when ran on input 100.
   What is the output of M on this input?

   (b) What is the output of M on 10011? on input 11?

   (c) What function is computed by M?

2. Let $\Sigma = \{\#, 0, 1\}$. Provide an *implementation level* description of a input-output TM that computes the function

$$f(\#\langle x\rangle) = \begin{cases} \#\langle x/2\rangle & \text{if } x \text{ is even} \\ \#\langle 3x+1\rangle & \text{otherwise} \end{cases}$$

   where $\langle x\rangle$ stands for the binary representation of the number x.

   (For example, if the TM starts with #100 on the tape it should halt with #10 on the tape; if it starts with #11, it should halt with #1010.)

   You may use a TM with more than one tape – in this case the output should be written on the first tape.

3. Let $L = \{\langle M, k\rangle | M$ is a TM, k is a positive integer, and there exists an input to M that makes M run for at least k steps$\}$
Prove that that L is decidable.

4. Let L be a recognizable language.
Define 9/10(L) $= \{< x_1\#x_2\#...\#x_{10} > \ |$ at least 9 of $x_1, \ldots, x_{10}$ are in $L\}$.
Prove recognizable languages are closed under the 9/10 operation. Give both a high level and implementation level solution.

5. Let L be a recognizable language.
Define m/n(L) $= \{< (m, n, x_1\#x_2\#...\#x_n) > \ |$ at least m of $x_1, \ldots, x_n$ are in $L\}$.
Prove recognizable languages are closed under the m/n operation. Give a high level implementation.

6. Let CYCLE $= \{\langle G\rangle \mid G$ is a graph that contains at least one cycle$\}$. Recall that a cycle is a sequence of vertices connected by edges that starts and ends at the same vertex. Prove CYCLE is decidable.