Lecture 10: Turing Machine Variants

Ryan Bernstein

1 Introductory Remarks

- I still haven't graded assignment 2 because this week is the worst and I also am the worst

1.1 Recapitulation

Last lecture, we introduced the Turing machine. Turing machines are capable of computing anything that can be algorithmically computed. Like finite and pushdown automata, they're controlled by a set of states and a transition function. However, they have the following new properties:

- Rather than being constrained to moving across the input strictly from left to right, Turing machines have the power to move in either direction across an infinitely long input tape. Initially, all cells on the input tape to the right of the input string are populated by a blank character ␣.
- Turing machines are capable of writing to the tape as well as reading from it
- Turing machines have a single accept state q_{accept} and a single reject state q_{reject}. Entering either of these states immediately terminates computation. If the machine is not in either of these states, it has neither accepted nor rejected the string

Since state diagrams of Turing machines grow very complex very quickly, we construct them using informal natural-language descriptions instead.

Since Turing machines may never enter q_{accept} or q_{reject}, they may not accept or reject a given input string, instead looping forever. This means that with the introduction of these machines, we've introduced not one, but two new classes of language.

A Turing machine M *decides* a language L if and only if:

1. $\forall s(s \in L \rightarrow \ M \ ACCEPT\text{s} \ s)$
2. $\forall s(s \notin L \rightarrow \ M \ REJECT\text{s} \ s)$

We say that a language L is *Turing-decidable* if and only if there exists some Turing machine that decides it.

A Turing machine M *recognizes* a language L if and only if:

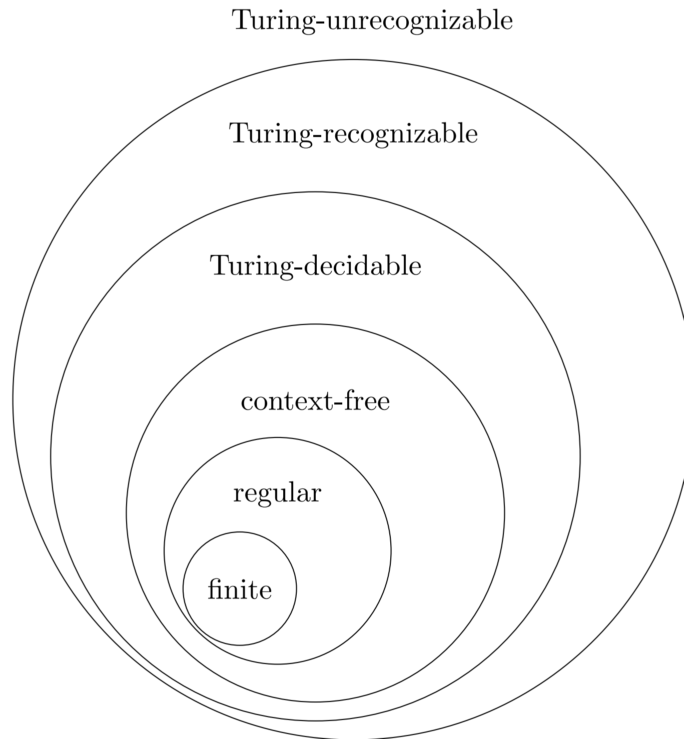
1. $\forall s(s \in L \rightarrow \ M \ ACCEPT\text{s} \ s)$

In other words, a recognizer M will $ACCEPT$ any string in L, but may not $REJECT$ any string that is not a member of L. If at an arbitrary point in time we inspect M during its processing of a string s and

find that it is in neither q_{accept} or q_{reject}, it's not clear whether $s \notin L(M)$ or whether M simply needs more time to process before $ACCEPT$ing s.

We say that a language L is *Turing-recognizable* if and only if there exists some Turing machine that recognizes it.

At the end of the last lecture, we also showed that Turing machines could emulate pushdown automata and were thus capable of deciding any context-free language. Since we've already seen examples of Turing machines that decide non-context-free languages, we know that both Turing-decidable languages and Turing-recognizable languages are proper supersets of context-free languages. This completes our hierarchy of languages:



The one thing that we hadn't addressed was nondeterminism — since nondeterministic pushdown automata are capable of deciding more languages than are deterministic ones, we need to show that the power of Turing machines is *not* affected by nondeterminism to prove that we can emulate nondeterministic PDAs. This will be addressed today.

2 Turing Machine Variants

Today, we're going to talk about variants of Turing machines. We do this for the same reason we introduced NFAs and GNFAs after DFAs: if we can show equivalence between a standard Turing machine and any other type of machine, then we can prove decidability or recognizability of a language using whichever variant is most convenient for the problem at hand. Like these other types of machines, we have one method for showing equivalence: provide methods for converting a standard Turing machine to the new type of machine and vice versa.

Turing machines constructed for emulation look a lot like the machines that we constructed on Tuesday.

Since we know very little about the language being decided, though, we instead provide an algorithm for emulating each possible type of action that a machine of the emulated class might take.

During the following proofs, it helps to remember that all we're doing is determining what Turing machines are capable of doing. We don't care about efficiency at all, and if you are somebody who worries about that sort of thing, we'll be doing some things that will probably make you cringe.

2.1 Turing Machines That Can Stay Put

First, a quick example that will 1) demonstrate our construction method and 2) make our lives slightly easier. Currently, we require that on each transition, a Turing machine moves its tape head either left or right. To make proofs easier, we'll allow our Turing machine to execute a transition without moving its tape head. Our transition function will therefore be of the form $\delta_{stay} : Q \times \Gamma \to Q \times \Gamma_\epsilon \times \{L, R, S\}$ rather than the original $\delta : Q \times \Gamma \to Q \times \Gamma_\epsilon \times \{L, R\}$. However, we'll first need to prove that this does not allow the Turing machine to decide any language that it might otherwise not have been able to.

It's clear that a standard Turing machine is not more powerful than a Turing machine with stay-put capabilities. It's simply a machine with stay-put capability that contains no transitions that don't move the tape head.

We can emulate a stay-put machine with a standard Turing machine as follows: $S =$ "On input w:

1. Emulate all transitions labeled $a \to b, L$ or $a \to b, R$ as usual
2. Emulate any transition labeled $a \to b, S$ as two transitions:
   (a) $a \to b, R$
   (b) $\Sigma \to \epsilon, L$"

2.2 Multi-tape Turing Machines

A multi-tape Turing machine is exactly what it sounds like. Each tape has its own independent tape head. The δ function of a multi-tape Turing machine with k tapes is therefore of the form:

$$\delta_{multitape} : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

We'll assume that the first tape contains the input string as usual, and that all other tapes are initialized blank.

Again, it's clear that a standard single-tape Turing machine is not more powerful than a multi-tape machine: we can simply think of a single-tape machine as a multi-tape machine where $k = 1$.

We'll emulate a multi-tape machine by writing all of the tape contents to a single tape. We'll also need a way to keep track of the location of each tape head. To do this, we'll mark the location of each tape head with a circle.

We can construct a single-tape Turing machine S that simulates a machine with k tapes as follows:

$S =$ "On input w:

1. Write the initial contents of all tapes onto the input tape. Separate the contents of each tape using a delimiting character #. To begin, assume that each new tape contains only a single blank character.

   The formatted tape will then look something like the following:

$$\#\dot{w}_1 w_2 ... w_n \# \dot{\sqcup} \# \dot{\sqcup} \# ... \#$$

2. To simulate each move of the multi-tape machine, first scan the full tape from left to right to determine which characters are under each tape head and determine the appropriate transition to take. Then make another pass that updates the contents of each tape and the location of each virtual tape head.
3. If any tape head moves to the right and encounters a #, it has attempted to move off the end of its tape. To address this, write a $\dot{\sqcup}$ in that space and then shift every subsequent character on the tape one space to the right."

2.2.1 Uses for Multi-Tape Turing Machines

Now that we know that multi-tape Turing machines are equivalent in power to single-tape Turing machines, we can prove that languages are Turing-decidable (or Turing-recognizable) using whichever is easiest. As an example, here is a way that we can use a multi-tape Turing machine to decide a language that we've already seen in a much more intuitive way.

Example 1 Construct a multi-tape Turing machine to decide $A = \{0^n 1^n 2^n\}$.

Construct a machine with two tapes. The first is the input tape. The second is the counter tape, which is initialized as entirely blank.

$M =$ "On input w:

1. If $w = \epsilon$, $ACCEPT$
2. If w is not of the form `0+1+2+`, $REJECT$
3. For each zero:
   (a) Write a 1 to the counter tape and move both tape heads one space to the right
4. Return the counter tape head to the beginning of the counter tape
5. For each 1 on the input tape, move both tape heads one space to the right
   - If the input tape head reads a two while the counter tape head reads a one, $REJECT$
   - If the input tape head reads a one while the counter tape head reads a blank, $REJECT$
6. Return the counter tape head to the beginning of the counter tape
7. For each 2 on the input tape, move both tape heads one space to the right
   - If either tape head reads a blank before the other, $REJECT$
8. $ACCEPT$"

While it's true that this is a problem that we've already solved, this machine represents an algorithm that's much closer to the way that we as humans might solve this problem. Rather than zig-zagging back and forth across the input tape, we simply count the number of zeros and then make sure that it matches the number of ones and twos.

Worksheet Exercise Show that a Turing machine is equivalent in power to a pushdown automaton with two stacks.

2.3 Nondeterministic Turing Machines

The following proof is incredibly long. While its conclusion may be important on an assignment or exam, the low-level details of the proof itself will not be, so don't ruin your arm trying to write everything down if you don't want to. This proof is also available in the book and in the online lecture notes.

Nondeterminism changes Turing machines in much the same way as it did the previous types of machines that we looked at: the machine is now able to follow multiple transitions on the same state/input pair instead of being restricted to just one. The transition function is therefore of the form $\delta_{NDTM} : (Q \times \Gamma_\epsilon) \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon \times \{L, R, S\})$.

Nondeterminism allows us to execute multiple *branches* of computation at once, and as such, we can view the computation of any nondeterministic machine as a tree. At each state-input pair, we have zero or more states to which we can transition (and zero or more associated actions that we may perform on the tape).

Because Turing machines may loop forever, this tree may have infinite depth. Therefore, we'll attempt a breadth-first traversal of this tree to search for an accepting state. To do this, we'll need to establish a branching factor, which is the maximal number of transitions that originate from any single state/input pair (and therefore the maximal number of branches that any single node in our tree will contain). We'll call this branching factor b.

We can use this branching factor to assign an address to every node in the computation tree. The address of the starting state will be ϵ. The node one step down the first transition branch from this start state will have address 1. The node one step down the second transition branch from this start state will have address 2. We continue in this manner until we reach branch b. We can combine these numbers to find the address of nodes further down. Every character in an address describes one transition. The node reached by following the second transition from the starting node, the first transition from the next node, and the third transition from the next therefore has address 213.

Having established this convention, we can execute a breadth-first search of the address space by constructing a counter. This counter starts with 1, then goes to 2, and continues until it reaches b. At this point, it has generated the addresses of every node that can be reached in one step from the starting state. It then continues with addresses $11, 12, ..., 1b$, which generates the address of every node that can be reached in *two* steps from the start state. By combining this counter with a method for navigating to a given address, we can execute a breadth-first traversal of the entire computation tree.

To simulate a nondeterministic Turing machine M, we'll construct a multi-tape Turing machine D that has three tapes. The first tape holds a copy of the input string that is never modified. The second tape is the simulation tape, on which we execute whatever modifications were made by the transitions we've followed in N. The third tape is the address tape, which is written to by the counter that we just described.

$D =$ "On input w:

1. Initialize the string on the address tape to ϵ
2. Copy the contents of the input tape to the simulation tape
3. Use the simulation tape to simulate a single branch of N on input w. For each step, consult the address tape to determine which nondeterministic branch to take.
   (a) If we reach the end of the address tape, go to step 4
   (b) If the address represents a state to which no transition exists, go to step 4
   (c) If this address leads to q_{reject}, go to step 4 UNLESS
   (d) If *all* addresses at this depth are invalid or lead to q_{reject}, $REJECT$
   (e) If this address leads to q_{accept}, $ACCEPT$
4. Replace the contents of the address tape with the next address. Erase the contents of the simulation tape and go to step 2"

This machine simulates a nondeterministic machine by taking each transition, erasing the tape, and starting over. It is horribly inefficient. But all that we're concerned about is the fact that this nondeterminism doesn't change the set of languages that a Turing machine can decide or recognize.

2.3.1 Uses for Nondeterminism

Before the midterm, we showed that the language $\{ww \mid w \in \{a,b\}^*\}$ was non-context-free. On Tuesday, we constructed a Turing machine that shows that $\{w\#w \mid w \in \{a,b\}^*\}$ was Turing-decidable. In this case, the delimiting # character was necessary because we didn't know where to break the string.

The machines we introduced today provide two methods of deciding $\{ww\}$ without the delimiting character:

1. We could use (or simulate) a multi-tape Turing machine to determine how to break the input string exactly in half
2. We could use (or simulate) a nondeterministic Turing machine to break the string at every possible point and see if the latter section of the input matched the former section exactly.