CS 4510 Automata and Complexity February 23rd 2023

Lecture 10: Church-Turing Thesis

Lecturer: Abrahim Ladha *Scribe(s): Rishabh Singhal*

1 Introduction

The Church-Turing Thesis cannot be proved. Most agree that it is some kind of definition or worse, a "working hypothesis". Here, we will give the closest thing to a proof possible. Recall the narrative of our class. First we did regular languages, that was level one. We pumped out of those to get some context-free languages, that was level two. Now we are on level three, Turing machines. The Church-Turing Thesis essentially says that there is no level four. Our argument has two parts:

(I.) There is no computing device strictly more powerful than the human mind.

(II.) Turing Machines are equivalent in power to the human mind.

Together, these imply that a Turing Machine, although incredibly simple, is an excellent choice for us to reason about computation. A philosophical argument is unlike a proof. We want to make a convincing argument of some statement. The way we will do so is make atomic jumps, each convincing, then argue that the composition must be convincing.

2 Part I: Our Own Limits

Suppose our space of the languages looks like the following:

$$\mathscr{L}(NFA)\Big)\ \ \mathscr{L}(CFG)\Big)\ \ ...\ \ \mathscr{L}(Human)\Big)\ \ \mathscr{L}(?)$$

where $\mathscr{L}(Human)$ is the set of languages recognizable by a human. It's a pretty big class. If I give you a description of a language and a word, and you can tell if that word is in the language, then that language is in $\mathscr{L}(Human)$. Its not even yet clear if there are languages outside of this class. Could something exist with $\mathscr{L}(Human) \subsetneq \mathscr{L}(?)$ Our first observation is that if it existed, then we could not build it. Otherwise, we could understand it. Anything humans have managed to build, they do by understanding it. Even if we do not understand why some things happen, we understand what happens. So suppose then it is somehow an Alien device.

Assume to the contrary this alien device exists in a useful way and $\mathscr{L}(Human) \subsetneq \mathscr{L}(Alien)$[1]. Then there does not exist a simulation of $\mathscr{L}(Alien) \subseteq \mathscr{L}(Human)$. I claim that it it is then unfathomable. It is beyond our comprehension and therefore useless. A contradiction.

[1] If this computer wasn't strictly stronger, but its power somehow orthogonal, we could augment the alien computer with a human counterpart creating a strictly more powerful computer.

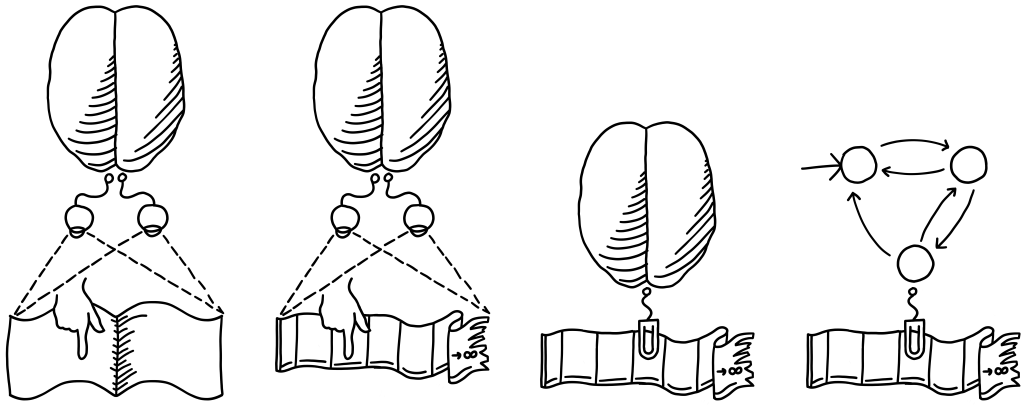


Figure 1: Successive simplications of computation as made in the Direct Appeal to Intuition

Note how we really couldn't argue that such a computer couldn't exist. Only that we couldn't build it, or it would have no effect on us in any measurable, meaningful, or interactive way. If something exists in no way which is fathomable, it doesn't make sense to even discuss it, like an agnostic God. The core of our argument here is that we have been able to simulate every computer so far in our head. Given a DFA and a word, you can run the DFA on the word in your mind. Given psuedocode, you can follow along line by line. If there was a device which we couldn't do this with, it may as well not exist relative to us.

3 Part II: The Direct Appeal to Intuition

These notes have been typeset separately: https://ladha.me/files/fancy-turing-notes.pdf

4 Church Turing Thesis

We now give our statement of the Church-Turing Thesis. For any kind of computation model, mechanical process, decision procedure C,

$$\forall C \ \mathscr{L}(M) \subseteq \mathscr{L}(TM)$$

Its useful to rephrase this as

$$\nexists C \ (TM) \subsetneq \mathscr{L}(C)$$

In human words, the Turing machine is the ultimate computer. The entire complexity and confusion of the mind, at least in our framework of decision problems, can be simplified to the humble Turing machine. This pathetic three button typewriter is equivalent in power to any realizable computer, and there is no greater. It is the supreme.

5 Evidence

We will now show a more traditional argument in favor of the Church-Turing Thesis. We will attempt to generalize the Turing machine. We will show that each generalization is

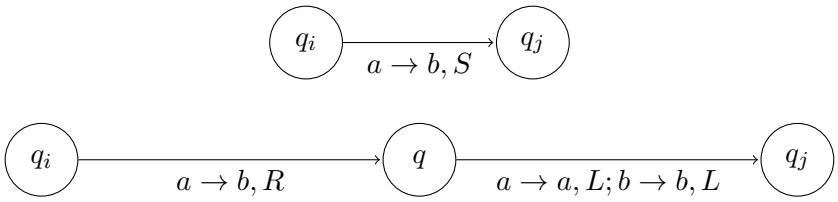
infact, just equivalent.

5.1 Turing machine with Stay

Consider Turing machines with a stay instruction. Instead of moving either left or right, we allow the tape head to remain on the same cell. Its transition function would be defined like:

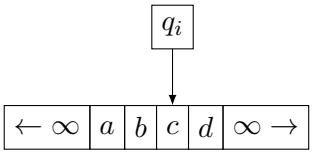
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

It is obvious that $\mathscr{L}(TM) \subseteq \mathscr{L}(stayTM)$. Let's prove $\mathscr{L}(stayTM) \subseteq \mathscr{L}(TM)$. If a Turing machine with stay has a normal L, R move, we leave it alone. If a Turing machine with stay has a S move, we can simulate it as a sequence of L, R moves. Specifically we will choose to move right, then back left[2].



5.2 Turing machine with Two Way Tape

Our Turing machine's tape is only infinite in one direction, so lets generalize it to be infinite in two. This can be called a bidirectional, doubly infinite, or two way tape.



It is true that $\mathscr{L}(TM) \subseteq \mathscr{L}(2wayTM)$, but the simulation needs an extra sentence. We put our one way tape on the two way tape and add a special marker leftmost of our tape. If we ever read it, we force ourselves right. Now lets show $\mathscr{L}(2wayTM) \subseteq \mathscr{L}(TM)$. There were many excellent simulation suggestions in class, but the elegant one is to just fold the tape in half! Here we extend our tape alphabet to cover pairs as so.

$$\Gamma^2 = \left\{ \frac{a}{b} \mid a, b, \in \Gamma \right\}$$

If you are in the right half of the tape, the transition function will essentially ignore the bottom half of the symbol. If we attempt to move into the left half, we start ignoring the tops and looking at the bottoms. We also flip every L, R move.

[2]The only reason we don't move left then right is the case we are on the leftmost square.

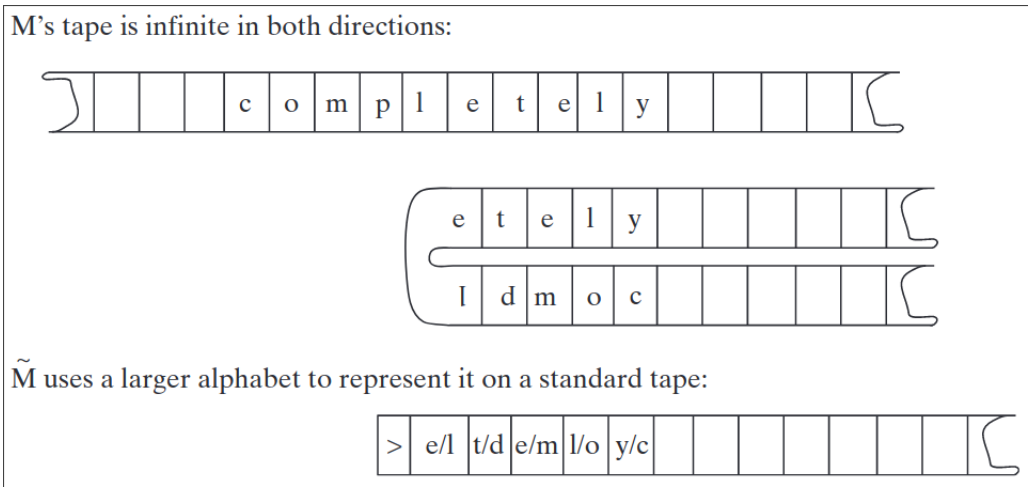


Figure 2: Folded tape in half, from the Arora-Barak book

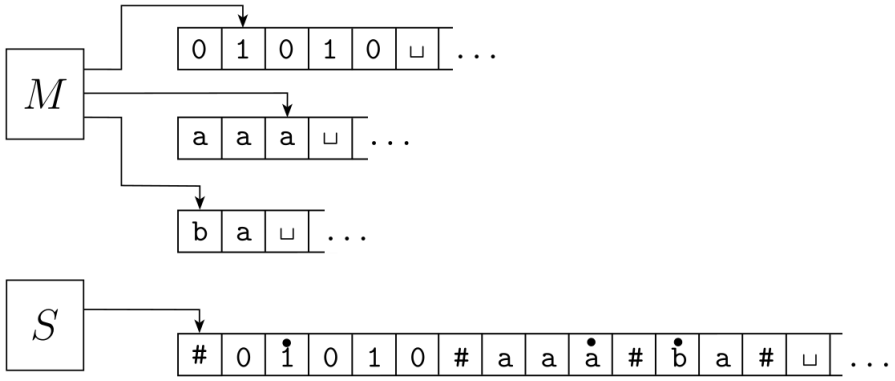


Figure 3: A Three tape Turing machine being simulated on a single tape, from the Sipser book

5.3 Multi-Tape Turing Machine

If you think about it, the negative half of a bidirectional tape is like a second tape. Lets define a Turing machine with multiple tapes. A multi-tape Turing machine is just that. It has k tapes it may read, right and move on all independently. Its transition function would look like

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

At some state, it will read the position of its k tape heads, makes k writes, and moves each head either left or right independently. Certainly $\mathscr{L}(TM) \subseteq \mathscr{L}(kTM)$ by simply ignoring all tapes except the first one.

We want to show $\mathscr{L}(kTM) \subseteq \mathscr{L}(TM)$. In order to do so, we are going to simulate the k tapes on a single tape. We initialize our single tape as

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

The dot represents the position of that tape head over its tape. As we simulate a read, write, move of each tape head, we will scan over our tape making the appropriate adjustments. If we need more space than allocated, we pause or simulation, enter a shifting subroutine, insert blanks appropriately, and then continue. What was essential for this simulation was that at any time stamp, only a finite amount of space has been used. A Turing machine cannot use the infinite nature of the tape in any useful way.

5.4 Nondeterministic Turing machine

A nondeterministic Turing machine is defined exactly as you might think. At some state reading some symbol, there are more than one outgoing transitions which could be taken. There exists more than one computation path. Its transition function would be defined as

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Its certainly true that $\mathscr{L}(TM) \subseteq \mathscr{L}(NTM)$ as a nondeterministic Turing machine is simply a generalization of a Turing machine. We now show the reverse way, that $\mathscr{L}(NTM) \subseteq \mathscr{L}(TM)$.

We can view a nondeterministic computation like a rooted tree. Each node in this computation tree can be represented by a configuration, with the initial configuration representing the root, $q_0 w_1 ... w_n$. Our deterministic simulator is going to attempt to search this tree for an accepting computation, if one exists. If our nondeterministic machine has some accepting configuration in this tree, then there exists an accepting computation path and this computation path halts. Our deterministic simulator will find this accepting configuration, and also halt. If our nondeterministic machine has no accepting configuration in this tree, then on all computation paths it either rejects or loops. So our tree may go on forever. Similarly, our deterministic simulator will halt or search the tree forever.

Recall the many tree traversal algorithms. An initial idea is to use DFS: Depth first search. This is not a good idea. If the first computation path we find is some infinite loop, we may miss an easy accepting configuration. The next idea is then to use BFS: Breadth first search. Our deterministic simulator is going to implicitly perform BFS on the computation tree.

Recall how BFS works on a rooted tree. It uses a queue. We pop an element off the queue and push its children. Just as a refresher, consider the following example. Here the traversal of the elements goes layer by layer from our initial node.

We are going to use the tape kind of like a queue. We will pop (read, then erase) a configuration off the front of the tape, compute the possible next configurations, and push (write) them to the end of the tape. For example if our nondeterministic machine had the following structure:

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$

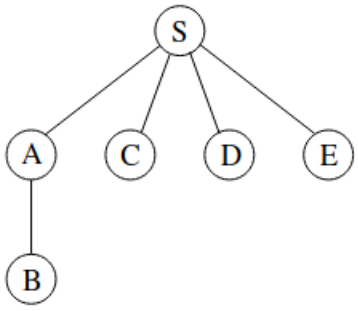
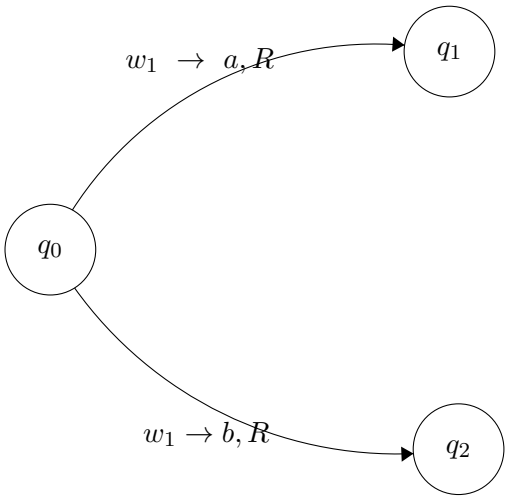
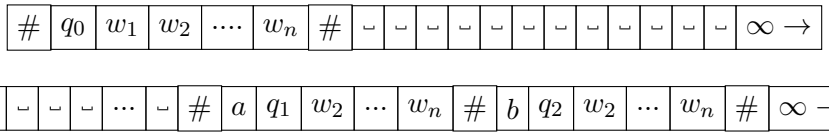


Figure 4: BFS, from the Dasgupta-Papadimitriou-Vazirani book



Then our tape would transform like:



The configuration $q_0 w_1 ... w_n$ yields two configurations nondeterministically, those being $aq_1 w_2 ... w_n, bq_2 w_2 ... w_n$. We would read the initial configuration on the tape, compute its two next configurations, and append those on our tape. We would then erase the initial configuration. We would then next look at the first configuration on the tape and repeat. While we are doing this, we check if a configuration contains an accept state, and if it does, we accept. If it contains a reject state, we toss the configuration and continue searching the others. Its worth noting the Sipser book does a slightly different more detailed simulation, still using BFS but with three tapes. I recommend you read it. We have shown that nondeterministic Turing machines are no more powerful than Turing machines. Keep this simulation in mind when we discuss the resource bounded variant of the question: $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

6 Turing Completeness

For any kind of computation model, mechanical process, decision procedure C, we define it to be Turing-complete if

$$\mathscr{L}(TM) \subseteq \mathscr{L}(C)$$

Recall that by the Church-Turing Thesis, we get the reverse implication for free, that $\mathscr{L}(C) \subseteq \mathscr{L}(TM)$. All we need in order to show that a computer is equivalent in power to a Turing machine is to be able to simulate a Turing machine on it. As a brief example, Python is Turing-complete since I believe you could write a Turing machine simulator in it. We could have applied this to the previous four generalizations to immediately get that they must be Turing-complete, but I wanted to actually work through some of the simulations. You may hear the phrase "Turing-complete" in pop culture[3] In practice, this can mean wildly different things which I won't expand on here. I only ask you not get confused. Now we show two surprising models of computation which are Turing-complete.

6.1 Unrestricted Grammar

An unrestricted Grammar is just that. Recall that for a CFG we had productions of the form

$$V \to (V \cup \Sigma)^*$$

A single nonterminal gets substring replaced by an arbitrary string of terminals and nonterminals. In comparison, an unrestricted grammar places no restrictions on the left hand side. It has productions of the form

$$(V \cup \Sigma)^* \to (V \cup \Sigma)^*$$

Any string of terminals and nonterminals can be replaced by any string of terminals and nonterminals. Here we lose the distinction the terminals and nonterminals as well. A string of terminals is not necessarily "terminal", and there may be more productions to follow.

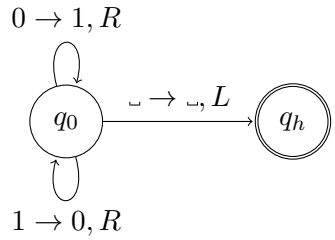
We show that unrestricted grammars are Turing complete. By the Church-Turing thesis, you could write a simulator for an unrestricted grammar, so $\mathscr{L}(UG) \subseteq \mathscr{L}(TM)$. We want to show its Turing complete, so given a Turing machine, we will construct an unrestricted grammar to simulate it. This will prove $\mathscr{L}(TM) \subseteq \mathscr{L}(UG)$.

For any computation of a Turing machine, there exists a sequence of configurations. Our simulation idea is that the sequence of working strings of our grammar will be analogous to this sequence of configurations. The next production we can apply with our unrestricted grammar will be analogous to the next configuration.

If our states were $\{q_0, ..., q_k\}$ then our nonterminals will be $\{S, A, Q_0, .., Q_k\}$ If in our Turing machine $abq_icd \vdash abc'q_jd$, we add productions $Q_ic \to c'Q_j$. Similarly if $abq_icd \vdash aq_jbc'd$ then we add the set of productions $aQ_ic \to Q_jac', bQ_ic \to Q_jbc'$. We add our production to set things up as $S \to Q_0AB$ and then A produces Σ^*, perhaps like $A \to aA \mid bA \mid \varepsilon$. If we have a halting state q_h we add production $Q_h \to \varepsilon$. We also want to read blanks so we have $B \to \sqcup B \mid \varepsilon$

Lets do an example. Recall the Turing machine which simply computes the bitflip of its input and halts.

[3] https://xkcd.com/2556/



We would have a sequence of configurations on input 101 as

$$q_0101 \vdash 0q_001 \vdash 01q_01 \vdash 010q_0\sqcup \vdash 01q_h0$$

Then our grammar would have a sequence of productions like

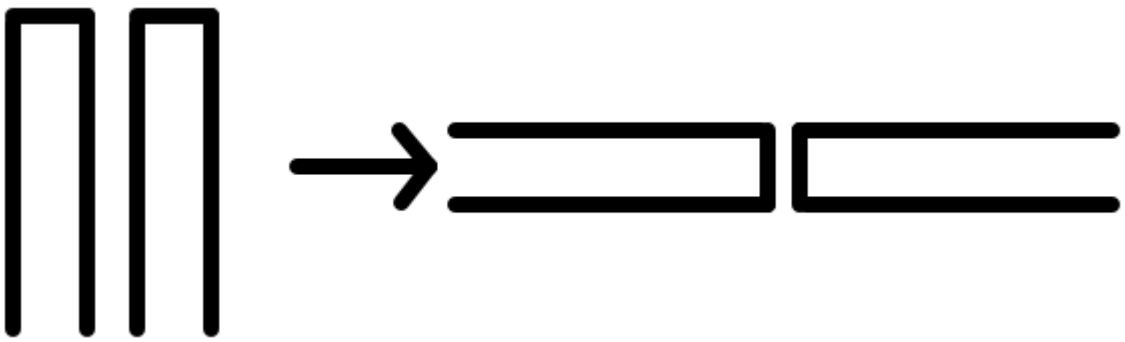
$$S \implies Q_0AB \stackrel{*}{\implies} Q_0101\sqcup \implies 0Q_001\sqcup \implies 01Q_01\sqcup \implies 010Q_0\sqcup \implies 01Q_h0\sqcup \implies 010\sqcup$$

Notice that for any two sequential configurations, only a small local part of the string is changed. This will be essential for many proofs in the future. Its clear that the grammar simulates the machine but we have been vague on the details. If our Turing machine computes a function f, then our grammar will produce $f(\Sigma^*) = \{f(w) \mid w \in \Sigma^*\}$. You can perhaps believe we could fill in the details to get the simulation to accept or reject appropriately, rather than compute everything. Or perhaps compute only the string we wanted. The important part is the simulation.

6.2 $\mathscr{L}(TM) \subseteq \mathscr{L}(2PDA)$

A 2PDA is defined as you might think, a PDA with two stacks. The transition function could be defined to read from them one at a time or to push and pop both simultaneously. Either way, we claim that a PDA with two stacks is Turing complete. First by the Church-Turing Thesis notice that $\mathscr{L}(2PDA) \subseteq \mathscr{L}(TM)$. We will show $\mathscr{L}(TM) \subseteq \mathscr{L}(2PDA)$ by simulation of a Turing machine on a PDA with two stacks.

Recall the intuitive limitation of a PDA with one stack. If you need to read deep into the stack, you have to pop things out losing them into the ether. With two stacks, instead of popping them out and losing them, just push them into the second stack. We join our two stacks together to form a bidirectional tape! The proof becomes obvious by the following change in perspective:



Denote one stack as the "left" stack and the other as the "right" stack. For some Turing machine right move of the form $a \to b, R$, we pop a from the right stack and push b to the left stack. For some Turing machine left move of the form $c \to d, L$, we pop c from the right stack, push d to the right stack. Then we pop whatever is on top of the left stack and push it to the right stack.

Here, we get a interesting observation. A PDA with one stack (PDA) is strictly stronger than a PDA with no stacks (NFA). A PDA with two stacks (Turing-complete) is strictly stronger than a PDA with one stack (context-free). But a PDA with three stacks is not strictly stronger than a PDA with two stacks, its equivalent. In the language of set theory:

$$\mathscr{L}(0PDA) \subsetneq \mathscr{L}(1PDA) \subsetneq \mathscr{L}(2PDA) = \mathscr{L}(3PDA) = \mathscr{L}(4PDA) = ...$$

Two stacks is the limit!. Its just enough to get all computation. The "degrees of freedom" or amount of power a machine needs to be Turing-complete is relatively small.

7 Physical Realizability

A common theme in any Turing-complete model of computation is that we appeal to physics and intuition. They all share some common traits.

- Program descriptions are of finite length
- A constant amount of work done in unit time. If more work needs to be done, successive operations must be performed

This is the heart of the Church-Turing Thesis. Our understanding of algorithms is invariant in the way we choose to represent them. As long as some basic requirements are met, many models of computation are Turing-complete. There do exist theoretical "Super-Turing" models of computation, but these are explicitly unrealistic on purpose. For example, they allow $\Gamma = \mathbb{Q}$. This would allow you to encode any string into a single cell, and compare arbitrarily long strings in constant time. Something totally infeasible and unintuitive.