CS 4510 Automata and Complexity February 21st 2024

Lecture 11: Turing-Completeness

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

1 Turing Completeness

For any kind of computation model, mechanical process, decision procedure C, we denote $\mathscr{L}(C)$ to be the class of languages recognizable by device C. The definition is vague on purpose, to support the diversity of systems which may be analogous to computation. It may even be too limiting to characterize computation of something by a decision procedure of a set of strings.

Definition 1.1 (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, if C satisfies the fathomability criterion 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, and check that it satisfies the fathomability criteron. Almost every single device will satisfy the fathomability criterion except those contrived not to, such as a DIA from the first problem set.

As a first example, consider the python programming language. A programming language is only a notation to abstract us away from the hardware. As you write code, you deal with the ideal language as a mental model, rather than the computers instructions. Python is Turing-complete. Why? Because you may write a Turing machine simulator in python. From this we immediately see that $\mathscr{L}(TM) \subseteq \mathscr{L}(PY)$.

Although a relatively straight forward argument, we can already make several deep remarks. First, note how we exercised the Church-Turing thesis. We did not have to prove $\mathscr{L}(PY) \subseteq \mathscr{L}(TM)$. Simulation of a python program by a Turing machine would be annoyingly complicated. Since we know we may simulate python programs in our brain, we can fathom them, we may apply the Church-Turing thesis to get this containment for free. Next, note what part of this argument was specific to python. None of it really, so all reasonable serious languages are also Turing-complete. Have you ever noticed that all serious programming languages have the same ability in terms of possibility? One may be faster in terms of efficiency or usability, but never just possibility. All serious languages are equivalent by the fact they are all Turing-complete. The fact that not one is superior to the others follows from the Church-Turing thesis. There do exist non-Turing-complete programming languages for extremely contrived use cases.

Recall the four generalizations of the Turing machine we gave last time. The Turing machine with stay, the Turing machine with a two-way tape, the multi-tape Turing machine, and the nondeterministic Turing machine. 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.We show two surprising models of computation which are Turing-complete.

1.1 Unrestricted Grammar

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

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

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

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

Any string of terminals and non-terminals with atleast one non-terminal can be replaced by any string of terminals and non-terminals. Here we lose the distinction the terminals and non-terminals. We only require there to be one non-terminal so we have a stopping condition still when no non-terminals remain in the working string.

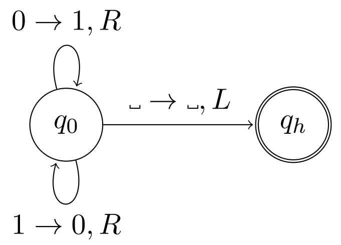
Theorem 1. Unrestricted Grammars are Turing-complete.

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 $\mathscr{L}(TM) \subseteq \mathscr{L}(UG)$.

Proof. 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 step of the transition function.

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^* as $A \to aA \mid bA \mid \varepsilon$. If we have a halting state q_h we add production $Q_h \to \varepsilon$. We also want to read blanks so we have $B \to \text{␣}B \mid \varepsilon$. If our Turing machine computes a function f, then our grammar will produce $f(\Sigma^*) = \{f(w) \mid w \in \Sigma^*\}$. $\square$

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



We would have a sequence of configurations on input 101 as

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

Then our grammar would have a sequence of productions like

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

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. It is clear that the grammar simulates the machine. You can perhaps believe we could fill in the details to get the simulation to accept or reject appropriately, rather than compute. The important part is the simulation of a Turing machine on a grammar.

By the Church-Turing thesis, we get that there is no type of grammar which is capable of producing a strictly larger class of strings. We observe a hierarchy of string rewriting systems and their power.

$$\left.\mathscr{L}(RG)\right) \quad \left.\mathscr{L}(CFG)\right) \quad \left.\mathscr{L}(CSG)\right) \quad \left.\mathscr{L}(UG)\right)\!\!\Big)$$

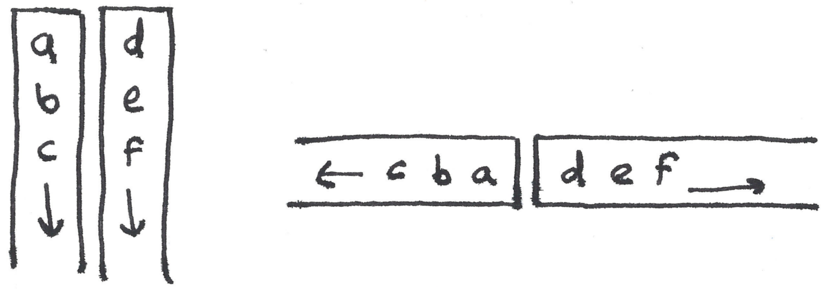
1.2 A Push-down Automata with Two Stacks

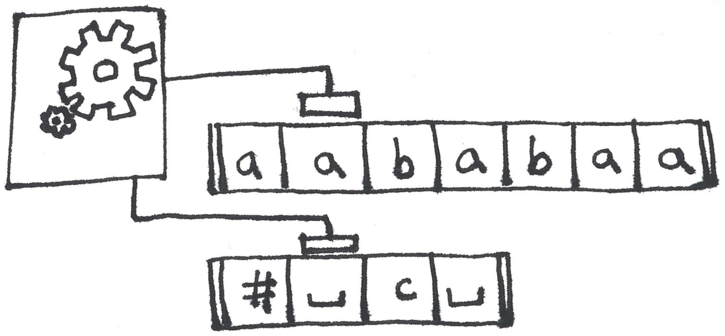
A 2PDA is defined as you might think, a PDA with two stacks. The transition function could be defined to read from them one at a time or to push and pop both simultaneously.

Theorem 2. 2PDA is Turing-complete.

We prove $\mathscr{L}(TM) \subseteq \mathscr{L}(2PDA)$. First by the Church-Turing Thesis notice that $\mathscr{L}(2PDA) \subseteq \mathscr{L}(TM)$.

Proof. 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. By the Church-Turing thesis, since the 2PDA is Turing-complete, the 3PDA cannot be strictly stronger. This implies there must exist some simulation of a 3PDA on a 2PDA. 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 actually quite small.

$$\left.\mathscr{L}(0PDA)\right) \quad \left.\mathscr{L}(PDA)\right) \quad \left.\mathscr{L}(2PDA)\right)\!\!\Bigg)$$

2 Finite Tape Turing Machine

The "infinite" nature of the tape is necessary it turns out. Suppose we model a finite tape Turing machine as one with a separate input tape which can only be moved right and read from, and a work tape of only finitely many cells.

Theorem 3. A Turing machine with a finite work tape is not Turing-complete

In fact, we prove it is regular. We can give two proofs actually.

Proof 1. Let L be decidable by a finite tape Turing machine with k space. We show L is decidable by a finite tape Turing machine with only $k-1$ space. Repetition of this argument until there are no cells left will output a DFA. Consider the execution of the k cell machine. Create the $k-1$ space machine whose states are two copies of the states of the k space machine. Rather than reading or writing to this cell, transition between the two copies of the states of the machine. Which copy you are in will implicitly keep track of what the contents of the tape cell was. □

Proof 2. Let L be decidable by a finite tape Turing machine with k space. We give a DFA to decide L. The finite tape Turing machine can only have a finitely many of configurations. There are $|\Gamma|^k$ possible writings on the tape, $|Q|$ possible states it could be in, and k possible positions of the tape head on the work tape. Create $|\Gamma|^k \cdot |Q| \cdot k$ states denoted by configurations, and wire them appropriately according to the transition function δ. The accept states of the DFA are those whose configurations contain the accept state of the finite tape Turing machine. □

2.1 Is no real computer Turing-complete?

Absolutely not. This is a very common misconception on the internet, phrased as a sort of gotcha. I claim that we are right and everyone on the internet is wrong by the fact that they do not take a rigorous enough approach to Turing-completeness.

The argument usually proceeds as follows. Every computer has finite RAM and storage. A Turing machine has an infinite tape, and therefore, can decide more than any space bounded machine. Another argument is that C is not Turing-complete, since there is a limit to the amount of memory it can address.

These arguments rely on the fact that a Turing machine supposedly has an infinite tape. Beyond a surface level inspection, we can observe that it does not actually use the infinite tape in any useful way. If we are concerned with computation, we are concerned with the machines which halt and output something. If a machine halts within k steps, it may use no more than k cells of the tape. Every halting computation ends up only using finite space. As we have seen with Cantor and his contemporaries, there is a large difference between a limit towards the infinite, and the infinite itself. The Turing machine does not use the infinite tape in any usefully infinite way. The tape is not truly infinite, but arbitrary. It can only interact with it according to a finite set of rules to change local portions. In some sense, the tape is not part of the machine. The tape is nature, an unbounded environment, and the machine acts upon nature in a local way. Computation is the evolution of an environment according to simple local steps. This definition being so general is what motivates many similar systems to be Turing complete. Conway's game of life, Neurons in the brain, electrical signals in circuitry, proteins in DNA. I could go on. Computation is such a universal process, to argue the most modern powerful devices we have are theoretically restrictive on some technicality is nothing but contrarian.

The observable universe is itself finite. There are only 10^{80} atoms. No such computer of unbounded nature can be constructed. In the modelling of a Turing machine after the human act of computation, it is observed that you are not immediately supplied with an unlimited infinite supply of scratch paper at the begging of computation. Rather that if you needed more, you could always get up and get more. The finite nature of the universe has to do with its age relative to the speed of light. If we were to use a significant portion of the universe as space for some Turing machine computation, we may explore the limits of the current observable universe, and certainly observe more.

3 Physical Realizability

A common theme in any Turing-complete model of computation is that we appeal to physics and intuition. There are common traits shared across Turing-complete computational models. We can use these are a heuristic to determine if some specific model is Turing-complete.

- Program descriptions are of finite length. The set of rules are finite and atomic.
- A constant amount of work done in unit time. If more work needs to be done, successive operations must be performed.
- There are infinitely many possible configurations, each finite in description[1].
- Universality, or simulation - Turing-complete models of computation may simulate each other (perhaps with difficulty or overhead) and branch accordingly. Not true for PDAs or DFAs.
- Determinancy - If at some intermediary step of computation, the following step is unique and has been determined by local conditions.

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. But even these models may be simulated algebraically on a Turing-machine as long as they are fathomable.

[1]note this is technically also true for push down automata