Final Reflection

Gabe Orosan-Weine

June 2023

1 Computability

What problems can be computed? To approach this question, we start with the definition of a Turing machine.

Definition 1.1 (Turing Machine)**.** A Turing machine is described by a string containing a 7-tuple, $(Q, \sum, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where $Q, \sum, \Gamma$ are all finite sets and
1. Q is the set of states,
2. $\sum$ is the input alphabet not containing the blank symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\sum \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
4. $q_0 \in Q$ is the start state,
5. $q_{accept} \in Q$ is the accept state, and
6. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

A Turing machine has a tape which extends infinitely to the right, on which the input string is written, a head on the tape which keeps track of the symbol under it, an internal state, and a transition function which tells the machine what to set the current symbol to, what to set the internal state to, and whether to move the head left or right based on the symbol under the head and the current internal state. Both the tape alphabet (the input alphabet together with the blank symbol) and the set of internal states must be finite. Within the set of internal states are distinct start, accept, and reject states; the machine halts and returns the result when the internal state reaches either the accept or reject state.

A language being undecidable means that a Turing machine can be constructed that halts on every input of strings over the alphabet, accepting strings in the language and rejecting strings not in the language. For a language to be recognizable, it is only necessary that the Turing machine halts on strings in the language; strings which are not in the language can be rejected or go into an infinite loop. Thus the set of decidable languages is a subset of the set of recognizable languages.

While in the language reflection, we saw that Turing machines are capable of deciding languages like $\{a^n b^n c^n | n \in \mathbb{N}\}$, there are problems which cannot be decided by Turing machines, the most famous of which is the halting problem:

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a } TM \text{ and } M \text{ halts on input } w\}.$$

If a Turing machine is processing an input, it is not possible to know if it will continue indefinitely or if it will eventually come to a result and halt. Naturally, if it does halt then you have you answer; this idea can be used to construct a simple Turing machine which recognizes $HALT_{TM}$.

Lemma 1.1. $HALT_{TM}$ is recognizable.

Proof. Let R be a Turing machine that, given $\langle M, w\rangle$, runs M on w and accepts if it gets any result. R recognizes $HALT_{TM}$ since if M halts on w then M will return some result and R will accept. If M does not halt on w then R will be stuck in an infinite loop and not return anything. Thus R recognizes $HALT_{TM}$. □

However, it is not possible to construct a decider for $HALT_{TM}$.

Lemma 1.2. $HALT_{TM}$ is undecidable.

Proof. Assume $HALT_{TM}$ is decidable, so we can construct the decider H, where

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ halts on } w, \\ reject & \text{if } M \text{ does not halt on } w. \end{cases}$$

Let S be a Turing machine that, given a string description of a Turing machine m (where m describes M), runs H on M and m. Then,

$$S(\langle m\rangle) = \begin{cases} accept & \text{if } M \text{ halts on } m, \\ reject & \text{if } M \text{ does not halt on } m. \end{cases}$$

Note that S halts on every input since $HALT_{TM}$ halts on every input by assumption. Then define the Turing machine OPP, which takes in a string description of a Turing machine and runs S on m, going into an infinite loop if S accepts and accepting of S rejects.

$$OPP(\langle m\rangle) = \begin{cases} loop & \text{if } S \text{ accepts } m, \\ accept & \text{if } S \text{ rejects } m. \end{cases}$$

We know S either accepts or rejects every input as mentioned above, so this is a complete description of OPP, which must either accept or infinitely loop on a given input. Consider what happens when you run of OPP on input $\langle opp\rangle$, where *opp* is the string describing OPP. For the result to be *accept*, S must reject *opp*, meaning OPP does not halt on *opp* which would be a contradiction. For $OPP(\langle m\rangle)$ to loop on *opp*, S must accept *opp* meaning OPP halts on *opp*, which is a contradiction. Therefore OPP does not loop or accept on input *opp* which we have established it must. This is a contradiction, so we must not be able to construct decider for $HALT_{TM}$. □

It turns out that there are many languages which are not even recognizable. We have established that $HALT_{TM}$ is recognizable but not decidable, which implies that its complement is not recognizable.

Theorem 1.3. Let L be a language that is recognizable but not decidable. L^c is not recognizable.

Proof. Assume that L^c is recognizable; let R_c be a Turing machine that recognizes L^c and let R recognize L. Construct the Turing machine T which runs R and R_c in parallel, returning the result if it comes from R and the opposite if it comes from R_c. We will show that T decides L. Given an input string s, if s is in L then R will accept s after a finite number of steps, so T will accept s in a finite number of steps as well. If s is not in L then L^c will accept it after a finite number of steps, at which point we know s is in the complement of L so T will appropriately reject s. s is either in L or its complement so this proves that T decides L. □

Therefore, there are languages which are not even Turing-recognizable, one of which is $(HALT_{TM})^c$. If a Turing machine is bound to keep computing forever on a given input, there are certain problems for which no way of knowing beforehand or at any step in the computing process. A more general result, which can also be used to prove the halting problem is undecidable, is Rice's theorem.

Theorem 1.4 (Rice's Theorem)**.** Let P be a language consisting of Turing machine descriptions where P contains some, but not all, Turing machine descriptions and $L(M_1) = L(M_2)$ implies that $\langle M_1 \rangle \in P$ if and only if $\langle M_1 \rangle \in P$. P is undecidable.

Less formally, Rice's Theorem states that any nontrivial semantic property of the language of a Turing machine, such as whether it is empty, or contains a certain string, is undecidable. This is an even more significant result when combined with the Church-Turing Thesis: any problem which can be solved by computation can be solved by a Turing machine. While it is not possible to create a formal proof for the Church-Turing Thesis, we have many reasons to believe that it is true. There have been multiple systems of computation devised apart from Turing machines, including λ-calculus, which have been shown to compute an equivalent class of problems as Turing machines. To get an idea of when this concept could come into play, consider the Collatz conjecture, which asks whether the function $f(x)$ eventually returns 1 on every natural number input, where

$$f(x) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ 3n+1 & \text{if } n \text{ is odd.} \end{cases}$$

Given the fact that f can be expressed so simply, we might assume that we can determine whether it halts on a given input beforehand. But from what we know about Turing machines, it becomes clear that if there is a natural number which does not eventually land on 1, we may have no way to compute

it. The idea that certain problems are not possible to compute has wide-reaching implications across math, computer science, and beyond.

2 Complexity

Within the class of problems that can be computed, there are further ambiguities. To understand why, we introduce the notion of time complexity.

Definition 2.1 (Time Complexity)**.** Let M be a deterministic Turing machine that halts on all inputs. The time complexity of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the time complexity of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.

In theoretical computer science, we often categorize time complexities by how they behave as n approaches infinity.

Definition 2.2. Let f and g be functions, $f, g : N \to R^+$. $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

We call this big O notation, which groups together functions whose asymptotic behavior is within a constant factor of each other. This allows us to define the traditional groupings of time complexity as follows.

Definition 2.3 (Time complexity class)**.** Let $t : N \to R^+$ be a function. The time complexity class, $TIME(t(n))$ is the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

If all solutions to a problem can be verified in polynomial time, can a solution necessarily be found in polynomial time? We begin with the requisite definitions.

Definition 2.4 (P)**.** The class P of languages that are decidable in polynomial time by a Turing machine is

$$P = \bigcup_k TIME(n^k).$$

Definition 2.5 (Verifier)**.** A verifier for a language A is an algorithm V, where

$$A = \{w | V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w. A language A is polynomially verifiable if it has a polynomial time verifier.

Definition 2.6 (NP)**.** NP is the class of languages that have polynomial time verifiers.

Arguably the largest unsolved problem in computer science today is the question of whether $P = NP$. One concept which could be useful in an eventual proof is NP-completeness.

Definition 2.7. A language B is NP-complete if it satisfies two conditions:
1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

NP-complete problems are at least as complex as NP problems since any polynomial time solution to an NP-complete problem would imply a polynomial time solution to all NP problems. On the other hand, if it was shown that a single NP problem had no polynomial-time decider then it would prove that P is a strict subset of NP, and thus $P \neq NP$. While most computer scientists believe that such a problem exists, a proof has yet to be found.

If a proof were to be found that $P = NP$, it could have significant implications for cryptography, as modern cryptosystems like RSA work based on the assumption that although a given set of factors of a large number can be verified in polynomial time, it is much harder to find such factors for large products of primes. Overall, the question of what can be (efficiently) computed is an active area of research with a rich history, and has implications across many other fields.

3 Personal reflection

I found all the subjects we covered fairly interesting, but I am particularly curious about complexity and information theory. I had come across Big O notation before, but I never had a good grasp on exactly what constituted an operation or a certain time complexity class. Not that I have a grasp of the formal definition of Turing machines and Big O notation, I feel much more confident in my ability to determine the time complexity of a given algorithm. I also find these topics interesting because I see the concepts of complexity and information as fundamental to life and intelligence.

Part of the reason I enjoy math is because learning the math behind something gives me confidence that I actually understand the concepts behind it, rather than just having a heuristic understanding that applies to the case in front of me. While it is easy to come up with explanations for things like the evolution of life and intelligence on Earth, I find the perspective of trying to understanding it computationally much more compelling because I see it as having the potential to reach a more satisfying conclusion. I see complexity and information theory as instrumental in this pursuit because they provide rigorous ways of classifying and deriving concrete properties about algorithms that apply to a wide range of problems.

A lot of the concepts felt challenging when I first came across them, mainly because they were just of a different form from the kind of ideas that I have learned in math and computer science in the past. However, I found the book's definitions generally quite helpful and I feel comfortable with pretty much everything we have covered in regards to finite automata, Turing machines, and classes of languages. I would say I have a less rigorous understanding of complexity classes, specifically NP, because I am not comfortable enough with the

logical prerequisites (specifically, the concept of a witness) to really wrap my head around the formal definition.

A lot of what motivated me to take the class was a desire to understand the notion of computability and why certain problems are not computable. Partially because I understand it better now, and partially because my interests have shifted slightly, I come away from the class more curious about the more measurable side of complexity and its implications. I would say that I am satisfied with how the knowledge I have gained through the class has deepened my understanding of computability, which was my main goal coming in.

As I graduate and go on to graduate school to study statistics, I definitely think the ideas of time complexity and information will stick with me. My interest in statistics comes from wanting more rigorous tools for understanding complex systems (along with the hope that it will be applicable to artificial intelligence in the future). Since intelligence comes from compressing knowledge, information theory could provide useful insights about what can be learned from a given set of data, and how. Information/complexity theory can be applied to machine learning algorithms to classify problems that can be learned by a given algorithm and the expected rate of convergence to a solution. This is the main field of research I would like to contribute to in my academic career: understanding statistical properties of different learning algorithms, which I think the methods for analyzing algorithms that we have learned about in this class will be useful for.