Takeaway Notes: Turing Machines

Contents

1 Introduction

In this set of takeaway notes, we generalize the notion of “membership in a language” to the entirety of computing: for some question or problem, given some input, would the answer be “yes” or ”no?” Observe that if we can interpret that problem as a language where input that belongs to the language is the same as the input for which the problem's answer is "yes" and input that does not belong to the language is the same as the input for which the problem's answer is "no," then all computing problems with "yes" or "no" answers are indeed equivalent to some language (i.e., some set of strings).

2 Central Ideas

2.1 There are More Problems Than Programs

If we now envision that a *program*—a finite string—represents some solution to a problem (i.e., some mechanism that will tell us whether a given input yields the answer "yes" or "no"), we would first realize that there are more "problems" than there are "programs"—this is a diagonalization proof that is analogous to Cantor's proof of uncountable infinity.

Fortunately for us, the vast majority of these "problems" are not at all interesting to us: they may as well be answering "yes" or "no" at random. So this assertion, though initially shocking, turns out to *not* be quite the existential crisis that we may have originally thought.

2.2 We Can Identify Specific Problems for Which Programs Do Not Exist

This problem is the famous *halting problem* first identified (and proven to have no program) by Alan Turing. It turns out, however, that "halting" is a MacGuffin: we can actually come up with infinite variants of this problem (e.g., can a program tell if another program prints "hello, world"?), all with generally the same proof that they don't exist.

The core "mind trick" in proving the halting problem lies in this fragment of pseudocode. Assume that `halts` exists—i.e., we have access to a program (or function) that can tell whether another given program will stop. Now let's say we have a program `X`, written as follows:

```
program X(P) {
  if halts(P) then {
    while true
  } else {
    return
  }
  }
```

Notice now that if we pass `X` to itself—i.e., we invoke `X(X)`—then we get a contradiction, because if `X` halts, then `X` will loop forever (`while true`), but if `X` does not halt, then `X` *will* halt (`return`).

Problems that have this characteristic—i.e., no program exists that can definitively answer "yes" or "no" for that problem—are said to be *undecidable*. And undecidability, as it turns out, is the focus of much computing theory beyond the introductory level.

2.3 The Turing Machine is a Model for Computing That Can Show Undecidability

…And so far, no other model for computing has been shown to be more powerful than the Turing machine. Or, if a Turing machine cannot solve a certain problem (i.e., recognize a certain language), we can pretty much deem that problem to be unsolvable—because there is no other machine that has been shown to solve problems which a Turing machine cannot.

3 Central Ideas, Formalized

3.1 The Formal Definition of a Turing Machine

Not surprisingly, the formal definition of a Turing machine M is yet another tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where:

- Q is a finite set of *states* that the Turing machine may be in
- Σ is a finite set of possible *input symbols* to the machine
- Γ is the set of *tape symbols* that a Turing machine can "store"—and also, $\Sigma \subset \Gamma$
- δ is (again!) a *transition function*. $\delta(q, X)$ takes a state $q \in Q$ and a tape symbol $X \in \Gamma$ such that $\delta(q, X) = (p, Y, D)$ where:
    1. $p \in Q$ is the next state of the machine

2. $Y \in \Gamma$ is the symbol that the machine writes to the current location on its tape
3. D is a *direction*, either L or R, that tells us in which direction the machine's "head" moves on the tape—the machine always moves one step at a time

- $q_0 \in Q$ is the *start state* of the machine
- $B \in \Gamma$ (but not Σ) is the *blank symbol* that initially occupies all of the Turing machine's tape cells, except for a finite number of initial input symbols
- $F \subseteq Q$ is the set of *final* or *accepting* states of the Turing machine

3.2 Instantaneous Descriptions (IDs) of a Turing Machine

Like PDAs, we need a special notation to precisely indicate the status of a Turing machine at any given time. As with PDAs, this notation is called the *ID* or instantaneous description of the machine:

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$$

q is the current state of the Turing machine. It is positioned where the machine's *tape head* currently is, which in this notation is X_i. Note that to avoid confusion, we assume that all symbols representing state are distinct from the tape symbols.

Finally, $X_1 X_2 \cdots X_n$ represents the portion of the tape between the leftmost and rightmost nonblank symbols, *unless* the tape head is beyond either of these extremes. In that case, contiguous X_ks to the right or the left of q may be blanks, and i would be either 1 or n.

3.3 A Sample Turing Machine

It should be noted that Turing machines aren't really meant to be problem-solving tools per se—they are abstractions for *thinking* about problem-solving primarily, not for actually solving problems (we have programming languages for that!). Still, it is useful to fully spec out an actual machine to get a feel for how it works.

Here is a machine that solves the problem of recognizing strings in the language $0^n 1^n, n \geq 1$. True, this is context-free and does not need the full power of a Turing machine to solve, but as you will see, the definition can get pretty involved so it's better to start small.

First, like with all programs, it is best to imagine with this machine would do informally, in order to solve the problem. We can thinking of it as "counting" the 0s and 1s by marking them off as it encounters them. It would reach a final (accepting) state if the counting matches up, then would just halt or hang in a non-accepting state if it does match up. Because the tape is one-dimensional and movement is either just one to the left or one to the right, the machine's head would zigzag back and forth over the tape.

Let's start with the tuple:

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

Then of course we have the meat of it, the δ transition function:

	Symbol				
State	0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Table 1: Transition function for a Turing machine that accepts $\{0^n 1^n \mid n \geq 1\}$

3.4 Attempted Enhancements to the Turing Machine

- Turing machines with finite data storage
- Multitrack Turing machines
- Multitape Turing machines
- Multidimensional Turing machines
- Non-deterministic Turing machines—these machines simultaneously make multiple moves for certain transitions (i.e., the result of δ is now a set and no longer a single tuple. They recognize the same set of languages as deterministic Turing machines, but *might do this faster*—i.e. polynomial time vs. exponential time, the basis for P vs. NP.

3.5 Equivalent Reductions of the Turing Machine

- Turing machines with semi-infinite tapes
- Multistack pushdown automata (two are enough)
- Counter machines

3.6 The Languages of a Turing Machine

The languages that can be accepted by a Turing machine are called *recursively enumerable* languages. Undecidable languages, such as the “diagonalizaton language” derived from a variant of Cantor’s proof of uncountable infinity, are considered non-recursively enumerable.

However, within the category of recursively enumerable languages are a distinct subdivision. *Recursive* languages are languages $L = L(M)$ for some Turing machine M such that:

1. If $w \in L$ then M accepts w (i.e., it halts on an input of w in an accept state).
2. If $w \notin L$ then M eventually halts but *not* in an accept state.

Languages that do not have this property—meaning that M never halts on strings that are not part of the language—are recursively enumerable but not recursive. The problem is that “never halts” cannot be determined (the halting problem!)—what if the machine is just taking a *really really long time* before halting?—so problems that are representable by recursive languages are considered to be *decidable*. Otherwise, they are *undecidable*. This, finally, is the formal definition of an “undecidable” problem.

3.7 The Universal Turing Machine

If we think of a Turing machine itself as being representable by a string, then the recognition of all strings that represent a Turing machine can itself be framed as a language recognition problem. . . and thus can be thought of as itself having a Turing machine that recognizes it. The machine that recognizes this language, notated as L_u, is called the *universal Turing machine*, notated as U. Thus by our original language notation from way back, $L_u = L(U)$.

The exact form of L_u is an ordered pair (M, w) where M is the encoded machine and w is a string that M accepts. Thus, if U accepts (M, w), that means that M accepts w.

3.8 P vs. NP

On a separate note, the study of languages and the acceptance of languages enters into the area of *tractability*—i.e., *how many steps*, relative to the size of the input string, does it take for a machine to accept or reject that string? Practically speaking, we only consider *polynomial-time* problems as solvable. Exponential-time problems take much longer much more quickly, to the point where we don't consider them practically solvable by a computer.

The class of problems that can be solved by a *deterministic* Turing machine in polynomial time relative to the size of the input string is the class famously known as P. The class of problems that can be solved by a *nondeterministic* Turing machine in polynomial time is NP. And now we have the great unsolved problem of computer science: is $P = NP$? It is clear that $P \subset NP$, because a deterministic Turing machine is clearly a special case of a nondeterministic Turing machine. However, it remains unknown whether every problem that a nondeterministic Turing machine can solve in polynomial time has an equivalent deterministic Turing machine that can also solve it in polynomial time (higher degrees are OK; they just can't go exponential).

There is a whole body of theory work that has gone into exploring this question. Unfortunately, we cannot get there in the time remaining, so must leave that part as a teaser for future studies of theory, if you remain interested.

4 Important Skills

The main skill in this section is the ability to "read" and "run" a given definition of a Turing machine. Everything else is proofs or understanding proofs.

5 Big Picture Points