COM364 Automata Theory
Lecture Note*7 - Turing Machines

Kurtuluş Küllü

May 2018

In this course so far, we discussed FA and PDA as models of computation (as theoretical machines). FA can solve many problems but they have limited memory. The collection of languages they can recognize are called regular languages. PDA has unlimited memory and as a result can recognize a larger set of languages called context-free languages. However, PDA can only use their memory in a last in first out fashion. We have shown that there are languages that PDA cannot recognize. We now start looking at a more powerful model of computation called the *Turing machine* (TM). This model was developed by and named after Alan Turing. This model also has unlimited memory and can do more with it. It is a more accurate model of a general purpose computer. In fact, a TM can do everything that a real computer can do. But it also has limitations and there are problems that it cannot solve.

Let us first describe a TM in an informal way. A TM has an infinite tape to use as memory (Figure 1). It has a tape head that can read and write symbols from/to the tape as it moves left or right on it. When the machine starts, the tape contains only the input string and all other areas are blank. The machine does its computation often by writing to and reading from the tape until it goes into an accepting or rejecting state. Once such a state is reached, we have an *accept/reject* output. It is also possible that the machine never reaches an accepting or rejecting state. In this case, it goes on forever and never halts.

Example: Consider a machine M_1 that will test whether its input is a member of $B = \{w\#w \mid w \in \{0,1\}^*\}$ or not. In other words, in the input is a string in B, M_1 should accept and otherwise, it should reject.

Let us put ourselves in place of the machine. Because we can move left and right on the tape as we want, we can check whether every symbol has a match before and after the # symbol by going back and forth. When making these moves, we should place marks on the tape to remember where we were last. If there is a mismatch of symbols at any point, the machine can reject. If we match all the symbols without any mismatch, then the input will be accepted.

For a better understanding, let us look at the stages of such a computation with the example string 0110#0110. In Table 1 below, you see the contents of the tape at each row. The top row is the status of the tape at the beginning. The underlined symbol indicates the place of the tape head (read/write head) and the symbol ⊔ is a symbol to indicate blank places.

Going from row1 to row2, the machine marks the first symbol of the string before going to the right to check if it matches with a symbol after #. This is necessary because we want to be able

*Based on the book "Introduction to the Theory of Computation" by Michael Sipser.

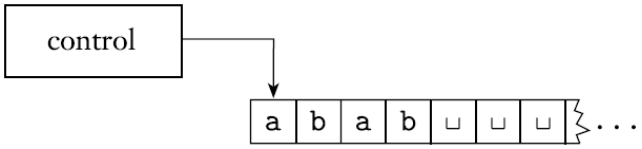

Figure 1: A diagram for a Turing machine (TM).

row1:	$\underline{0}$	1	1	0	#	0	1	1	0	$\sqcup$	...
row2:	x	$\underline{1}$	1	0	#	0	1	1	0	$\sqcup$	...
row3:	0	1	$\underline{1}$	0	#	0	1	1	0	$\sqcup$	...
row4:	x	1	1	$\underline{0}$	#	0	1	1	0	$\sqcup$	...
row5:	0	1	1	0	$\underline{\#}$	0	1	1	0	$\sqcup$	...
row6:	x	1	1	0	#	$\underline{0}$	1	1	0	$\sqcup$	...
row7:	x	1	1	0	$\underline{\#}$	x	1	1	0	$\sqcup$	...
row8:	x	1	1	$\underline{0}$	#	x	1	1	0	$\sqcup$	...
⋮											
row11:	$\underline{x}$	1	1	0	#	x	1	1	0	$\sqcup$	...
row12:	x	$\underline{1}$	1	0	#	x	1	1	0	$\sqcup$	...
row13:	x	x	$\underline{1}$	0	#	x	1	1	0	$\sqcup$	...
⋮											
row17:	x	x	1	0	#	x	$\underline{1}$	1	0	$\sqcup$	...
row18:	x	x	1	0	#	$\underline{x}$	x	1	0	$\sqcup$	...
⋮											
row24:	x	x	x	$\underline{0}$	#	x	x	1	0	$\sqcup$	...
⋮											
row29:	x	x	x	0	#	x	$\underline{x}$	x	0	$\sqcup$	...
⋮											
row40:	x	x	x	x	#	x	x	$\underline{x}$	x	$\sqcup$	...
⋮											
row44:	x	x	x	$\underline{x}$	#	x	x	x	x	$\sqcup$	...
⋮											
row50:	x	x	x	x	#	x	x	x	x	$\underline{\sqcup}$	...
										accept.	

Table 1: An example TM computation.

to continue with the next symbol when we come back. Between rows 2 and 6, the machine only moves right (without making any changes) until it reaches the symbol to match on the right side of #. After row 6, it marks the matching symbol and starts going back until row 11 where it sees the x symbol and stops going left. Next, we do the same procedure for the next symbol between rows 12 and 22. Machine continues in this fashion as the symbols keep matching. Eventually, there will be an x symbol on the left of # (row44) and the machine starts going right one last time here until it reads ⊔ (row50). This situation will result in the machine to go into an accepting state and the input will be accepted.

We described how M_1 should work but we didn't give all the details. For example, we didn't specify the states and transitions between them but if you understand the algorithm described above, you can actually work out these details. We will do so after we give a formal definition for Turing machines.

Formal Definition

A Turing machine (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, and Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the *blank symbol* ⊔,
3. Γ is the tape alphabet where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function where L and R means a left or right tape head move respectively,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accepting state, and
7. $q_{\text{reject}} \in Q$ is the rejecting state where $q_{\text{reject}} \neq q_{\text{accept}}$.

Initially, a TM has its input on the leftmost squares of the tape and the rest of the tape is filled with blank symbols. The tape head starts at the leftmost square. If M ever tries to go left from the leftmost square, it stays on the same square and continues. If M ever enters the accepting or rejecting state, it halts. Otherwise, it goes on forever.

While a TM is computing, there can be a change of state, a change of tape contents, and a change of tape head location. The status of these three things (state, tape contents, and tape head location) is called the machine *configuration*. A TM configuration is often represented with a string in the form uqv where q is the current state, uv is the tape contents, and the tape head is located at the first symbol of string v. For example, if a machine configuration is given as $100q_31011$, we know that the TM is currently in state q_3, its tape contains the string 1001011, and the tape head is on the second 1.

Definition: A language is *Turing-recognizable* (or sometimes called *recursively enumerable*) if some TM recognizes it (i.e., accepts the strings in that language).

Remember that there are three possible outcomes for a TM working on an input. The TM can accept its input, reject it, or it can go on forever without halting. The above definition states that strings in the language are accepted but it does not specify what happens to those not in the language. The machine can be rejecting such a string or it can be working without halting. If a machine halts on all inputs, it is called a *decider*. For such TMs instead of the word "recognize", we use the word "decide" and we say that the TM *decides* the language.

Definition: A language is *Turing-decidable* (or *recursive*) if some TM decides it.

Note: All Turing-decidable languages are Turing-recognizable.

Example: Let us look at a machine M_2 that decides the language $\left\{0^{2^n} \mid n \in \mathbb{N}\right\}$ (the language consisting of all strings of 0s whose length is a power of 2).

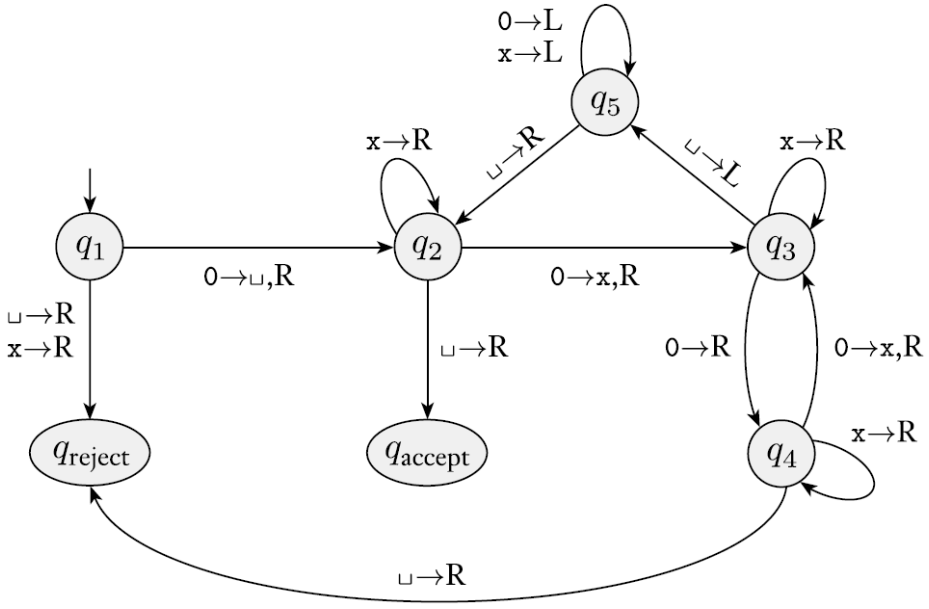

Figure 2: State diagram for M_2.

First, we give an informal description of how the TM will work and then we will look at the state diagram and formal description. M_2 will work as follows:

1. Go from left to right on the tape and mark every other 0 (mark one, skip the next, so on).
2. If there was only a single 0 in step 1, *accept*.
3. If there was more than a single 0 and the number of 0s is odd, *reject*.
4. Go back left to the beginning and return to step 1.

Let's try this on some examples using X to mark symbols. If we have eight 0s at the beginning as 00000000, after the first pass, we have 0X0X0X0X. After the second pass, we will have 0XXX0XXX. Then, we will have 0XXXXXXX. And finally, in the fourth pass, because there is a single 0, the string will be accepted.

However, if we have six 0s, we first get 0X0X0X. Then in the second pass, while we are marking to get 0XXX0X, we realize that there were three 0s (odd number) in step 3 and the machine rejects.

This algorithm works as each iteration halves the number of 0s. By doing so repeatedly, we will either get a single 0 (accept), or more but odd number of 0s (reject).

When you think about how these steps can be implemented, you realize that while marking the 0s from left to right, you need to keep track of whether the number of 0s is even or odd. Although it is not common to show TMs using state diagrams, for a better understanding, the state diagram for M_2 is given in Figure 2. You will realize that the transition from q_1 to q_2 replaces the first 0 with ␣ instead of X. This is so only because we want to be able to tell when we reach the left-end of the tape later when we are coming back to the left. We could use a separate symbol such as # for this task but blank symbol works and allows us to keep the stack alphabet small.

Drawing a state diagram or giving the formal description is often cumbersome for even the simplest machines. For example, if you try to do so for the machine M_1 we discussed at the beginning, you will realize that you need ten states. Because of this, we often discuss TMs at a formal algorithmic level instead of state diagrams or formal definitions using sets and functions.

Example: $C = \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \text{ and } (i = j \text{ or } i = k)\}$. Try to describe how a TM M_3 that recognizes language C can work (**Hint:** Go over how M_1 works, can you adapt it to this language?).

Variants

There are some variations on the TM definition that are important to know. For example, we can change the definition so that the machine has multiple tapes instead of a single tape or we can speak about determinism and nondeterminism. One of the interesting points about TMs is that many variants have the same power as the original definition, meaning that they recognize the same set of languages. For example, allowing a TM to have multiple tapes doesn't change the set of languages that can be recognized. A TM with one tape can do anything that a multiple-tape TM can.

Theorem: Every multi-tape TM has an equivalent single-tape TM.

Proof Idea: We show how a single-tape TM can simulate a multi-tape TM (See textbook).

Nondeterministic TMs are defined in the expected way. At any point, the machine has several possible choices it can follow. The computation of a nondeterministic TM can be viewed as a tree where each branch corresponds to one possible computation track.

Theorem: Every nondeterministic TM has an equivalent deterministic TM.

Proof Idea: We show how a deterministic TM can simulate a nondeterministic one by trying all possible branches of computation (See textbook). A key point is that the branches should not be explored in a depth-first fashion as it could cause the simulation to run infinitely. Also, because we know that a multi-tape TM can be simulated by a single-tape one, we can take advantage of having multiple tapes to make things easier.

Importance and Last Remarks

In 1900, mathematician David Hilbert identified 23 mathematical problems as a challenge for mathematicians. One of them (his tenth) problem was about finding integral (integer) roots of polynomials. Specifically, he was asking whether there is a "process according to which whether a polynomial has integral roots can be determined with a finite number of operations". So, in a way Hilbert was asking for an algorithm. He assumed that such an algorithm exists and it only needs to be found. The concept of an algorithm was not defined at the time. Today, we know that no algorithm exists for this task. But finding this required a mathematical definition for what an algorithm is. The definition came from the works of Alan Turing and Alonzo Church. Separately, Turing described an algorithm using his machines and Church used a notational system called λ-calculus. Later, these definitions are shown to be equal and they are combined under the name Church-Turing thesis. It has become the basis for proving that some problems are algorithmically unsolvable.

The modern computers we have today are not more powerful than a TM in terms of what they can do and do not. So, if a problem is shown to be undecidable by a TM (e.g. halting problem), we know that we cannot come up with an algorithm to solve that problem as it is. But often, we are able to solve by simplifying or altering the problem (i.e., develop algorithms that work in some/most cases or that produce probabilistically correct solutions).

A key concept used a lot in proofs for problems (languages), decidability or computability is *reduction*. Reduction involves two problems A and B, and one of these, let's say A is our main concern. In other words, assume that we want to prove something about problem A. If we can find a way to convert problem A to B, we say that A can be *reduced* to B. This is useful because if A can be reduced to B, solving A cannot be harder than solving B. Also, if we know that A is undecidable and that we can reduce A to B, then B must also be undecidable.

Beyond stating that languages are decidable or not, TMs are also used to speak about how fast (or in how much space) we can solve problems. For example, there are two famous classes of problems called P and NP. P is the class of problems (languages) that are decidable in polynomial time on a deterministic TM. NP, on the other hand, is the class of problems (languages) that are decidable in polynomial time on a nondeterministic TM. In practice, problems in P class are often problems that we can solve in reasonable amount of time even with large inputs. However, class NP (especially subsets of it called NP-hard and NP-complete) is generally linked with problems for which the best algorithms take very long time as input becomes larger. We know that P is a subset of NP ($P \subseteq NP$) but it is still an open question whether there is a problem in NP that is not in P (whether $P \subset NP$ or $P = NP$).