CS 3719 (Theory of Computation and Algorithms) – Turing machines

Antonina Kolokolova*

February 6 2019

1 Turing machines

Now we are moving on to the model of computation which we will use for the rest of the class: the Turing machine.

Alan Turing was working on a problem posed by Hilbert: does there exist an algorithm that for any statement in the language of mathematics would state if this statement is provable? The original problem asked for an algorithm that for any statement of mathematics would state whether it is true or false; Gödel has shown (his famous Incompleteness Theorem) that there are statements of mathematics for which such answer cannot be given. There were several mathematicians working on this problem at that time; notably, Alonco Church solved this problem (to give a negative answer) at about the same time, by inventing lambda-calculus. Turing's approach is somewhat more computational: he defined a model of computation which we now call the Turing machine, equivalent to Church's model in terms of power, and used it to show undecidability results, thus giving a negative answer to Hilbert's problem.

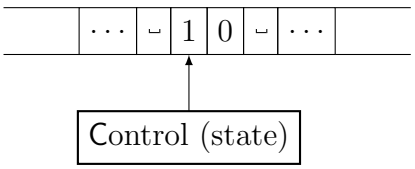
Definition 1 (Church-Turing thesis)**.** *Anything computable by an algorithm of any kind (our intuitive notion of algorithm) is computable by a Turing machine.*

Since this statement talks about an intuitive notion of algorithm we cannot really prove it; all we can do is that whenever we think of a natural notion of an algorithm, show that this can be done by a Turing machine.

*The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364.

2 Definition of a Turing machine

The weakness of finite automata was the lack of memory; pushdown automata had some memory, but only a limited access to it. A natural next step would be to take a finite automaton and add an unrestricted-access unlimited memory to it. This is essentially the intuition behind the definition of a Turing machine.



Memory is given to a Turing machine in form of an infinite tape. The Turing machine has a "head" which points to a cell on the tape. The head can both read and write in that cell, and can move in either direction. In the simplest definition, every cell contains just one symbol, and in one step of the computation the head can move by one position to the left or to the right.

Definition 2. *Formally, a Turing machine is a 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. *Here,* Q *is a finite set of states as before, with three special states* q_0 *(start state),* q_{accept} *and* q_{reject}. *The last two are called the halting states, and they cannot be equal.* Σ *a finite input alphabet.* Γ *is a tape alphabet which includes all symbols from* Σ *and a special symbol for blank,* $\sqcup$. *Finally, the transition function is* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *where* L, R *mean move left or right one step on the tape.*

At the start of the computation the input is written on the tape and the head points to the first symbol of the input. Sometimes Turing machines are defined to have a tape which is only infinite in one direction; in that case, the first symbol of the input is in the first cell of the tape. We will prove soon that these two definitions are equivalent.

The simplest Turing machine halts on an input by entering the state q_{accept} or q_{reject} in some point in the computation. Usually we assume that there are no transitions from q_{accept} or q_{reject}. It *accepts* its input if it halts in q_{accept}. For simplicity, we will often shorten q_{accept} as q_a, and q_{reject} as q_r.

Example 1. Recall that we have shown that the language $\{ww | w \in \{a, b\}\}$ is not context-free. Here we will show how to design a Turing machine accepting this language.

For simplicity, let's first design a Turing machine accepting the language $\{w\#w | w \in \{a, b^*\}$. Then we will say how to modify it to accept our original language.

We will be constructing a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$. The symbols in the input alphabet are $\Sigma = \{a, b, \#\}$.

M works as follows. It starts in a state q_0 pointing to the first symbol of the first copy of w. A symbol in the cell it is pointing to can be one of $\{a, b, \#, \sqcup\}$. If it is a or b, then M has to remember it (by going to q_1 or q_2, respectively) and scan till it is past the # sign. Then, it has to see if the corresponding first symbol of the second copy of w is a or b. After that, M

	a	b	$\#$	$\sqcup$	o
q_0	(q_1, o, R)	(q_2, o, R)	$(q_3, \#, R)$	q_r	(q_0, o, R)
q_1	(q_1, a, R)	(q_1, b, R)	$(q_4, \#, R)$	q_r	(q_1, o, R)
q_2	(q_2, a, R)	(q_2, b, R)	$(q_5, \#, R)$	q_r	(q_2, o, R)
q_3	q_r	q_r	q_r	q_a	(q_3, o, R)
q_4	(q_6, o, L)	q_r	q_r	q_r	(q_4, o, R)
q_5	q_r	(q_6, o, L)	q_r	q_r	(q_5, o, R)
q_6	(q_6, o, L)	(q_6, o, L)	(q_6, o, L)	$(q_0, \sqcup, R)$	(q_6, o, L)

Figure 1: A transition table for TM M recognizing $\{w\#w | w \in \{a.b\}^*\}$.

returns to the second symbol of the first copy and repeats till it runs out of $a's$ and $b's$. If at that point it ran out of symbols on both sides of #, then accept, otherwise reject.

How do we make sure that the same symbol is not counted twice? Let's introduce a new symbol into Γ, call it o (the name does not matter, as long as it does not occur in Σ and is not $\sqcup$). We will use it to mark the cells that we have already visited on both sides of #, so we don't count them again.

Now, each iteration of the computation proceeds as follows. Start in state q_0; then move right changing to q_1, q_2 or q_3 depending on whether the cell contained a, b or #. In the latter case, we have matched all symbols in the first copy of w and need to check if there is anything left in the second copy; so scan right staying in q_3, and reject if seeing a or b; accept if got to a $\sqcup$ (end of input). In the first two cases, mark the cell with o and move right until # is found. Then, change state: from q_1 to q_4 and from q_2 to q_5: here, q_4 will be looking for an a and q_5 for a b. In these two states, skip over o's; if a wrong symbol is seen first, reject, otherwise change to the "going-back state" q_6.

If M has a doubly-infinite tape, then in q_6 M can move left until it hits a $\sqcup$ before the input, at which point it changes to q_0 and moves right. If the tape has a beginning, we'll need to use an extra state: scan till # in q_6 and then till the rightmost o of the first copy of w in q_7. The table lists all the transitions for the doubly-infinite tape case.

This Turing Machine accepts a language $\{w\#w | w \in \{a, b\}^*\}$. How can we modify it to accept the language $\{ww | w \in \{a, b\}^*\}$? The simplest way to do it is to find the middle (by going back-and-forth from the beginning to end marking cells; in this case we'd want different markers for a and b). Then, insert # in the middle and move the second copy one cell left. We are back to the $\{w\#w\}$ case, except need a different name for q_0 and our a and b became some kind of marked $\dot{a}$ and $\dot{b}$, so need to rename the columns (and add rejects for seeing old a and b)

Let us define more formally what is a *computation* of a Turing machine. Recall that for regular languages we defined a computation as a sequence of states; for grammars we talked

about derivations. To describe a computation of a Turing machine we need to specify, at every point in time, its state, head position, and (non-blank) content of the tape. This information we will call a *configuration* of a Turing machine. For convenience, we will write a configuration as the string that is the sequence of non-blank symbols of the tape (assuming there are no blanks between non-blanks), with the symbol for the current state in a position right before the cell it points to. For example, the starting configuration of a Turing machine on input 0011 is $q_0 0011$; after a transition $(q_0, 0) \to (q_5, 1, R)$ the machine goes to the configuration $1q_5011$. Now, a Turing machine M *accepts* a string w if there is a sequence of configurations starting from $q_0 w$ and ending in a configuration containing q_a, with every configuration in the sequence resulting from a previous one by a transition in δ of M.

A Turing machine M *recognizes* a language L if it accepts all and only strings in L: that is, $\forall x \in \Sigma^*$, M accepts x iff $x \in L$. As before, we write $\mathcal{L}(M)$ for the language accepted by M.

Definition 3. *A language L is called* Turing-recognizable *(also* recursively enumerable, r.e*, or* semi-decidable*) if $\exists$ a Turing machine M such that $\mathcal{L}(M) = L$.*

A language L is called decidable *(or* recursive*) if $\exists$ a Turing machine M such that $\mathcal{L}(M) = L$, and additionally, M halts on all inputs $x \in \Sigma^*$. That is, on every string M either enters the state q_a or q_r in some point in computation.*

It is possible to define a Turing machine producing an output; in that case, the Turing machine halts in an accepts state, with the tape clear except for the output and head pointing to the first symbol of the output.

Remark 1. One reason that these languages are called "recursively enumerable" is that it is possible to "enumerate" all strings in such a language by a special type of Turing machine, enumerator. This Turing machine starts on blank input and runs forever; as it runs, it outputs (e.g., on some additional tape or part of the main tape) all the strings in the language. It is allowed to print the same string multiple times, as long as it does not print anything not in the language and eventually print every string that is in the language. See Sipser's book for a formal definition and a proof of equivalence.

Example 2 (+1 operation)**.** Here we will show a Turing machine M which takes as its input a string in binary and adds 1 to it. That is, it will halt in an accept state pointing to the first non-empty symbol, and the content of the tape will be input plus 1.

$M = (Q, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q_0, q_a, q_r)$.

To add 1 to a binary number, the usual algorithm is to start from the last bit, add 1 to it, and propagate the possible carry as much as needed, to the closest 0 or blank from the end. For example, to add 1 to string 1011 we need to propagate the carry to the second symbol, 0, while flipping the 1s.

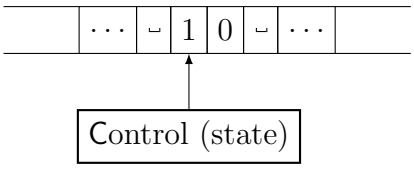
	0	1	␣
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_c, ␣, L)$
q_c	$(q_n, 1, L)$	$(q_c, 0, L)$	$(q_a, 1, R)$
q_n	$(q_n, 0, L)$	$(q_n, 1, L)$	$(q_a, ␣, R)$

Our Turing machine implements this algorithm as follows. It starts by scanning the input to the end of the input string, staying in q_0. After that, it starts moving backward, using two states: a "carry" q_c state and "no carry" state q_n. In the "carry" state, it flips a bit, and if the bit was a 1, remains in the carry state, otherwise goes to non-carry. In no carry state it scans back to the start of the string and there goes to the accept state.

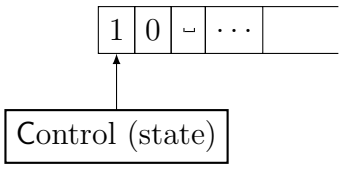
3 Variants of Turing machines

3.1 One-way vs. 2-way infinite tape

There are two ways of defining a Turing machine's tape. Either we assume that the tape is infinite in both left and right direction (like in the example before), or, as in Sipser's book, we allow the tape to be infinite only to the right, and require the input to start in the very first cell of the tape. Here we will show that it does not matter: both definitions are equivalent. That is, any Turing machine with 1–way infinite tape can be transformed into a Turing machine with two-way infinite tape, and vice verse.



Two-way tape Turing machine



One-way tape Turing machine

Lemma 1. *For every TM M with a one-way infinite tape there exists a TM M' with two-way infinite tape accepting the same language.*

Proof. A Turing machine with one-way infinite tape behaves exactly like a two-way infinite tape TM, except when it is in its rightmost cell then any transition that tries to move left stays in the same place instead. We will simulate it as follows. Using a similar idea to marking the bottom of the stack in a pushdown automaton, start by putting a special symbol \$ at the blank cell to the left of the input. Now, every time M' moves onto this

cell it has to come back to the previous place, remembering the state it was in. So, we add transitions $(q_i, \$) \to (q_i, \$, R)$ for every state $q_i \in Q$. Also, we will start in a state q'_0 with new transitions $(q'_0, a) \to (q'_0, a, L)$ $\forall a \in \Sigma$, and $(q'_0, \sqcup) \to (q_0, \$, R)$.

Finally, if this is a Turing machine producing output then it matters in which position its head is when it accepts/rejects its input; in this case, we need special treatment for the halting states. So the q_a and q_r of M would not be the halting states of M': instead, it will treat them as normal states and have a transition $(q_a, \$) \to (q'_a, \$, R)$ and similar for q_r where q'_a, q'_r are accepting and rejecting states of M'. But what should we do if M halts not at the start of the tape moving left? Then we need two transitions moving, say, right and then back and entering the halting state on the last step. That is, accepting sequence will be $(q_a, c) \to (q_a, c, R)$, then $(q_a, d) \to (q'_a, d, L)$. Here, $c, d \in \Gamma - \{\$\}$ are any non-\$ symbols and q_a remembers that the state was accepting; we'll use a different state say q_r for reject.

So, $M' = (Q', \Sigma, \Gamma \cup \{\$\}, q'_0, q'_a, q'_r)$, where $Q' = Q \cup \{q'_0, q_a, q_r, q'_a, q'_r\}$. In δ', there are all transitions in δ, plus there is an additional transition on \$ from every state, transitions for setting the marker and handling halting states as described above.

□

Lemma 2. *For every TM M with a two-way infinite tape there exists a TM M' with one-way infinite tape accepting the same language.*

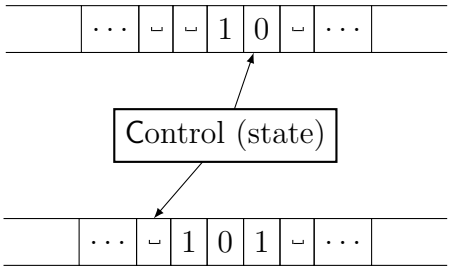
Proof sketch. The first thing that M' needs to do is to mark the start of its tape. Now, there are two ways it can imitate the two-way infinite tape. First, it can move the whole string to the right every time M moves left to the marker. There we need a new state for every pair i, c where i is remembering that the state trying to move left was q_i and c is a symbol being moved. Another set of states, say $q_{i,\sqcup}$, will scan the string backwards ending with $(q_{i,\sqcup}, \$) \to (q_i, \$, L)$. In that case, those states are used also to move the string to the right after placing the marker.

A different way to simulate two-way infinite tape is to have a "double" symbol in each cell, e.g. (a, b), where a is the symbol in the i^{th} cell to the right of the start on the two-way infinite tape and b is the i^{th} symbol to the left of the start. Then, double each state to the "left-state" and "right-state", change the transition table to use the double symbols and reverse the direction of transition when working with the second symbols in "left-state" copies of the states. Of course, at the marker M' switches from looking at the first symbol to the second if moving left, and from second to the first moving right.

□

3.2 Multi-tape Turing machine

Often it is convenient to describe a Turing machine using several tapes, each dedicated to a specific function. For example, if we want to simulate a random-access machine, we can have a dedicated "address tape" where the machine writes the address of the cell it wants to visit next.



Let us define a k-tape Turing machine to have k tapes, with a separate head on each tape, and a transition function $\delta\colon Q\times\Gamma^k \to Q\times\Gamma^k\times\{L,R\}^k$. That is, now for every state there are k characters the heads are pointing to, and, respectively, k-tuple of characters to overwrite the current ones and k-tuple of directions, one for each tape. Note that there is just one state, not k.

Here we will show that this definition is equivalent to the traditional one-tape Turing machine. There are several ways to do this simulation, similar to two ways of simulating a two-way tape Turing machine by a one-way tape TM. In the first case, we write strings corresponding to (non-blank part of) the tapes one after another, with a delimiter between strings on different tapes. Then, whenever a new symbol needs to be added, the rest of the tapes is shifted (just like putting a new symbol to the left of the used part of the tape in simulating two-way tape TM by one-way). Here, we use special symbols ("marked" versions of symbols of Γ) to represent a symbol with head pointing to it.

Another, a more interesting way is to expand the alphabet to make a symbol of each k-tuple (for k tapes) of symbols of the original alphabet (also with marked versions of symbols of Γ). This is similar to the second way of simulating two-way tape TM, where we used new symbols for the pairs of symbols from Γ; again, we need markers for the head positions.

In both cases, the Turing machine works by scanning all tape from the beginning to the end noting head positions and symbols under them. On the way back, it makes the changes according to the transition table.

3.3 Non-deterministic Turing machines

In non-deterministic computation, the transition function δ, rather than giving exactly one choice for the next step, gives a (possibly empty, but finite) set of choices. So the non-deterministic computation is not a sequence, but rather a tree, and it is successful (accepting) if at least one leaf is accepting. We call one branch of this tree a computational path. Each such path can be described by a sequence of choices the Turing machine made. The arity d of the tree is the maximum over all $q \in Q, a \in \Gamma$ of $|\delta(q,a)|$. So each node in the tree can be described by a sequence of numbers each from 1 to d saying which transition was chosen from a set (in, say, lexicographic order). Of course, some sequences do not correspond to

any node, but it is OK as long as any node has a unique sequence describing it.

Recall that non-deterministic Finite Automata recognize the same class of languages as deterministic ones; however, for pushdown automata there are languages that cannot be recognized without non-determinism. Which of the two cases holds for the Turing machines?

We will show here that it is possible to simulate a non-deterministic Turing machine by a deterministic one, although this simulation is not efficient. This efficiency issue is one of the main open problems in theoretical computer science, the famous P vs. NP question, whether efficient non-deterministic Turing machine computation can always be simulated by an efficient deterministic one. We will define precisely what we mean by efficient in a few lectures.

In order to simulate non-deterministic computation by deterministic we need to find if there is an accepting leaf in its computation tree. Since it is a Turing machine, there may be infinite branches in the tree (corresponding to infinite loops of the Turing machine), so we cannot do a depth-first search of this tree. However, we can do a breadth-first search. If we don't care at all about efficiency, we can do the simulation by just remembering which node we were in last, and restarting the computation from the beginning, making non-deterministic choices according to the path to the next node.

For simplicity, let's use the multi-tape Turing machine we just defined. Suppose our TM has three tapes: an input tape on which it will not write, a work tape, and a "address" tape, which in this case will keep track of the current computation branch. The address of a node here is a the sequence of choices that led to this node. The machine will work as follows: at every stage of the computation, starting with writing 0 on the address tape, it will run the computation from the start to the next non-deterministic choice following the sequence of choices written on the address tape. When it reaches a choice not on the tape, it will increment the address tape to the next node in the breadth-first order, and restart the computation. If one of the choices leads to a non-existing transition or reject state, abort the computation, increment the node and restart. If q_{accept} is reached, accept.

Since this Turing machine will eventually get to all nodes in every level, if there is an accepting leaf it will be found. Otherwise it will run forever. A check can be added to see if all nodes on the last level aborted and then reject.