COMPSCI 501: Formal Language Theory

Lecture 11: Turing Machines

Marius Minea
marius@cs.umass.edu

University of Massachusetts Amherst

13 February 2019

Insights on Computability

Turing machines are a model of computation

Two (no longer) surprising facts:

Although simple, can describe everything
a (real) computer can do.

Although computers are powerful,
not everything is computable!

Plus: “play” / program with Turing machines!

Why should we formally define computation?

Back to 1900: David Hilbert’s 23 open problems

Tenth problem:
Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

This asks, in effect, for an *algorithm*.
And “to devise” suggests there should be one.

Must indeed an algorithm exist?

Increasingly a realization that sometimes this may not be the case.

“Occasionally it happens that we seek the solution under insufficient hypotheses or in an incorrect sense, and for this reason do not succeed. The problem then arises: to show the impossibility of the solution under the given hypotheses or in the sense contemplated.”

Hilbert, 1900

Hilbert’s *Entscheidungsproblem* (1928): Is there an algorithm that decides whether a statement in first-order logic is valid?

Church and Turing

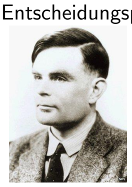
Church and Turing both showed in 1936 that a solution to the *Entscheidungsproblem* is impossible for the theory of arithmetic.

To make and prove such a statement, one needs to define *computability*.

In a recent paper Alonzo Church has introduced an idea of “effective calculability”, which is equivalent to my “computability”, but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem. The proof of equivalence between “computability” and “effective calculability” is outlined in an appendix to the present paper.

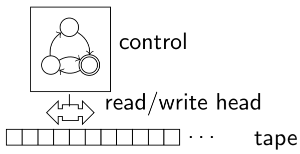
Alan Turing, 1936
On Computable Numbers, with an Application to the Entscheidungsproblem

Alonzo Church
(lambda calculus)



Alan Turing
(Turing machine)

A Turing machine, informally



control: finite-state automaton

memory: an infinite read/write tape (finite initial contents)

with a *tape head* controlled by the automaton

- *reads* symbol under the tape head
- *replaces* it with some symbol
- *moves* left / right

An example: decide $w\#w$

$\{w\#w \mid w \in \Sigma^*\}$ two identical words with a marker in between

With *computer program*: access both strings at same index, compare

`a b b a c b # a b b a c b`

With *Turing machine*:

can only access *one* symbol, then must move its "pair"

no indexing, how to find its corresponding pair?
need to "mark" cells already processed

remember symbol in first word for comparison with second
move into different *new state depending on symbol* seen
(alphabet of symbols is *finite*)

Sketch for checking matching words

`a b b a c b # a b b a c b ␣`	first symbol in word 1
`X b b a c b # a b b a c b ␣`	replace with X, move right
`X b b a c b # a b b a c b ␣`	find first non-X after #, check match
`X b b a c b # X b b a c b ␣`	replace with X, move back left
`X b b a c b # X b b a c b ␣`	find first X, move right
`X b b a c b # X b b a c b ␣`	second symbol in word 1
`X X b a c b # X b b a c b ␣`	replace with X, move right again

Insights from first example

Tape is both:
memory for *initial input*
scratch space

Usually: *overwrite input* cell by cell
with some symbol *not* in original input (here: X)

Use automaton *states to remember symbols* seen (one state per symbol)

Machine *alternates left and right* sweeps
must *detect tape ends*, to reverse

"Algorithm" composed of smaller *"subroutines"* (processing steps)
build small automata, glue them together ⇒ repeated processing

Turing Machine: Definition

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

- Q is a finite set of states
- Σ: the finite *input alphabet*, not containing ␣ (blank symbol)
- Γ: the finite *tape alphabet*, $\Sigma \cup \{␣\} \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the start state
- $q_{\text{accept}} \in Q$ is the accept state
- $q_{\text{reject}} \in Q$ is the reject state

Let's examine the definition

Tape alphabet Γ larger than the input alphabet Σ?

has to contain $␣ \in \Gamma \setminus \Sigma$
may contain other symbols (useful to mark over cells)

Can a transition write back the *same* symbol ?

YES, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ has no restrictions for symbol in Γ on the right-hand side

Can a transition write the blank ␣ symbol ?

YES, no restrictions
often, we will mark with a different symbol, keep ␣ just for ends

Can the head stay in place after a transition?

YES, if the move is L at the left tape end (can't go further)
we'll discuss how to detect that

Turing Machines: Decisions and Computations

We've formally introduced Turing machines with the goal of accepting or rejecting a string (characterizing a language).

But we could also use them for computations!

␣	␣	a	a	a	a	a	a	␣	␣	···

how many *a* symbols on the tape?

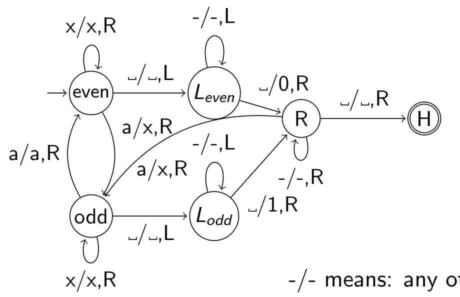
How would we write on the tape the number of *a* symbols, in *binary*?

How would you do it with a program? Can we translate it?

Converting from unary to binary

→: change every second a to x
←: write 0 or 1 according to parity
repeat until no more *a*

```
    ␣␣␣␣aaaaaa␣ → ␣␣␣␣xaxaxa␣
← ␣␣␣0xaxaxa␣ → ␣␣␣0xxxaxx␣
← ␣␣10xxxaxx␣ → ␣␣10xxxxxx␣
← ␣110xxxxxx␣ → ␣110xxxxxx␣
Halt
```



-/- means: any other symbol left unchanged

Configurations

A *configuration* describes the current snapshot of the machine
the *state* $q \in Q$ (of the control automaton)
the *tape contents*
the *position* of the tape head

The *transition function* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
describes a move from one configuration to the next

Representing configurations

We can represent *tape contents* AND *head position* by distinguishing
the string *left* of the head (possibly empty)
the string starting *under* the head, continuing *right*

$\boxed{q_7}$ (automaton state is q_7)
X X 1 0 1 # X 1 1 0 1 ␣ written as XX101#X$q_7$1101

With this notation we can precisely define a *step* of the Turing machine
i.e., configuration C_1 *yields* configuration C_2

Defining a Turing machine step

We identify and denote
symbols: b under the tape head, a left of it
words: u left of a and v right of b

$\boxed{q_i}$

$\underbrace{\texttt{X X 1 0 1 \#}}_{u}\ \underset{a}{\texttt{X}}\ \underset{b}{\texttt{1}}\ \underbrace{\texttt{1 0 1}}_{v}\ ␣$

We then define:

left move: if $\delta(q_i, b) = (q_j, c, L)$ then $ua\ q_i\ bv \overset{yields}{\Longrightarrow} u\ q_j\ acv$

right move: if $\delta(q_i, b) = (q_j, c, R)$ then $ua\ q_i\ bv \overset{yields}{\Longrightarrow} uac\ q_j\ v$

(in both cases, symbol b changes to c, and state q_i to q_j)

What happens at the two ends?

Left end: u is the empty string ϵ
left move: if $\delta(q_i, b) = (q_j, c, L)$ then $q_i\ bv \overset{yields}{\Longrightarrow} q_j\ cv$
symbol is changed, head stays in place
(cannot move left of leftmost cell)

Right end: tape is *infinite* and continues with blanks
Configuration $ua\ q_i$ is equivalent to $ua\ q_i\ ␣$
⇒ rule stays the same (head can move right, onto next ␣)
right move: if $\delta(q_i, ␣) = (q_j, c, R)$ then $ua\ q_i \overset{yields}{\Longrightarrow} uac\ q_j$

How do we recognize we're at the left end?

In processing we will *alternate* sweeps to right and left

Recognizing the right end is easy: we're at a blank ␣

How do we check not to "overrun" the left end ?

Option 1: *overwrite* with *special symbol* when we begin
may or may not need the old value of the first cell
if we do, create new tape symbol(s): a' from a, etc.

Option 2: *shift* entire tape contents
place blank ␣ or some other symbol at left end

Option 3: test by writing special symbol under head, checking if same symbol under head after left move, restoring actual symbol if needed
when moving left, check for special symbol

Some alternative TM edefinitions have a leftmost special symbol already

Exercise: Shifting tape contents

Initially: `a b b a c b ␣ ...`

Resulting: `␣ a b b a c b ␣ ...`

How to do? (and then continue processing?)
Remember *last* symbol seen, to write it in next cell to the right
$\Rightarrow$ one state per symbol of the alphabet Σ

Shifting tape contents k times
Option 1: repeat shifting once
must move k times back and forth over tape

Option 2: remember k symbols!
will need $|\Sigma|^k$ new states (all combinations of length k)
but one single pass

Our first *space-time tradeoff* !

Accepting and rejecting strings

Back to configurations:

starting configuration: $q_0\, w$ initial state q_0, tape has w, head is left

accepting configurations: all with state q_{accept}

rejecting configurations: all with state q_{reject}

accepting + rejecting = *halting* configurations
δ does not progress from halting configurations
$\Rightarrow$ could have defined $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$

A Turing machine accepts input w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that

C_1 is the start configuration $q_0\, w$

$C_i \overset{yields}{\Longrightarrow} C_{i+1} \qquad 1 \le i < k$

C_k is an accepting configuration

Recognizing a language

The set of strings accepted by machine M is the language *recognized* by M
or simply the language of M, denoted $L(M)$

Depending on the initial tape contents, a Turing machine could
end up in the accept state
end up in the reject state
loop forever (not halt)

Recognizing means accepting only and all the strings in the language,
but for other strings, the machine may either reject or **loop** forever.

$\boxed{\text{A language is } \textit{Turing-recognizable} \text{ if some Turing machine recognizes it.}}$

This corresponds to a *semialgorithm* (may or may not terminate).

Deciding a language

Ideally, we'd like a definitive answer: is a string in the language or not ?

Some Turing machines halt on all inputs, they never loop.
We call these machines *deciders*.

A decider that recognizes a language is said to *decide* that language.

$\boxed{\text{A language is (Turing-)decidable if some Turing machine decides it.}}$

Deciding a language is stronger than *recognizing* it.

More examples: Elementary Arithmetic

Let's encode multiplication as a decision problem!

Consider the alphabet $\Sigma = \{a, b, c\}$ and the language
$L = \{a^i b^j c^k \mid i \cdot j = k \text{ with } i, j, k \ge 1\}$.
How can we decide this language?

Hint: $i \cdot j = k$ means that for *each* a, the entire string of b's matches a (different) substring of c's in the result.

$$a\,a\,a\,bb\,cc\,cc\,cc$$

We now have a subproblem:
account for all b's in the string of c's
repeat for all a's

Must somehow restore string of b's in between steps

Wrap-up

Turing machines are a *model of computation*
most powerful we've seen
equivalent to *anything a real computer can do*

An *infinite tape* is used for both input and performing computation

To build a Turing machine for a problem, we use similar principles as in programming:
construct building blocks for *subproblems*
and *combine them*

Some languages are *Turing-recognizable*:
a Turing machine accepts the strings in the language
may reject or **loop** on any other input

A (smaller) set of languages is *Turing-decidable*
there is a Turing machine that always accepts or rejects