Turing Machines

CSCI 2210

2023-10-23 and 2023-10-25

Context-Free Languages

We used the *pumping lemma* for *regular languages* to show that not all string languages are regular, for example:

- $0^n\,1^n$, the language of a run of 0s followed by an equal-length run of 1s,
- $w\,w^{\mathrm{R}}$, the language of even-length palindromes (strings followed by their own reversal).

We saw that these languages are *context-free*. The stack of a PDA or tree-structure of a CFG lets us remember an unbounded *quantity* of information, but only of a certain *form*.

Non-Context-Free Languages

It is natural to ask, are all string languages context free?

The answer is no.

There is a pumping lemma for context-free languages as well.

It lets us divide any long-enough string into five parts and pump the middle three.

Some simple languages that are not context-free include:

- the language of concatenations of two copies of the same string, L_{ww},
- the language of three runs of the same length, $\mathrm{L}_{a^n b^n c^n}$.

Intuitively, the problem is that the information we need to decide such languages is lost when we pop the stack.

Traversing the Memory

It seems like we could gain a lot of computational power if we could traverse the memory at will, instead of having access to just the top of a stack.

This is the insight behind the Turing machine.

Indeed, we don't even need to separate the memory from the input, because if we could traverse the memory then we could just begin by copying the input string into memory.

Doubly-Linked Lists

The kind of traversable memory we will consider can be represented by a **doubly-linked list**.

This is a 1-dimensional sequential data structure two *ends*: *left* and *right*,

and where each element

- has a left neighbor, unless it is itself at the left end,
- has a right neighbor, unless it is itself at the right end.

The list can grow by *pushing* a new element onto either the left or the right end,

and it can shrink by *popping* an element from either the left or right end.

Turing Machine

Intuitively, a (deterministic) **Turing machine** ("TM") is a finite automaton with a doubly-linked list memory. Formally, it has the following components:

input alphabet: a nonempty finite set of symbols Σ,

working alphabet: a finite set of symbols that includes the input symbols $\Gamma \supseteq \Sigma$,

state set: a nonempty finite set of states Q,

start state: a state $q_0 \in \mathrm{Q}$,

halt state: a state $q_h \in \mathrm{Q}$,

state transition partial function: a partial function $\delta : \mathrm{Q} \times \Gamma \rightharpoonup \mathrm{Q} \times \Gamma \times \mathrm{D}$ such that $q_h \notin \mathrm{dom}(\delta)$.

The set $\mathrm{D} := \{\mathrm{L}, \mathrm{R}\}$ is the two *directions* of a doubly-linked list: left and right.

Partial Functions

A **partial function** $f : \mathrm{A} \rightharpoonup \mathrm{B}$ is just a relation $f : \mathrm{A} \nrightarrow \mathrm{B}$ such that each element of the domain set is related to *at most* one element of the codomain set.

We say $x \in \mathrm{A}$ is in the **domain of definition** of f, and write "$x \in \mathrm{dom}(f)$", if $\exists\, y \in \mathrm{B} \;.\; f\,(x \pfun y)$.

Because there is at most one such y, we can also use function notation and write "$f\,(x \mapsto y)$" or "$f(x) = y$".

TM State Transition Graph

We can represent a TM M using a state transition graph.

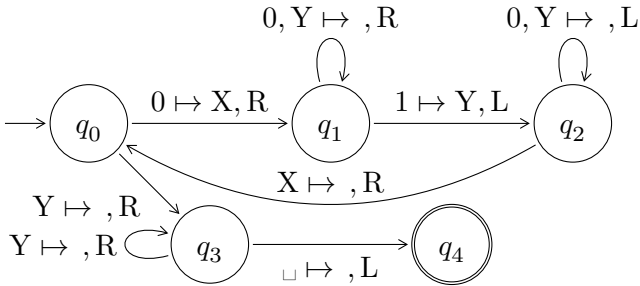
The vertex set is Q.

For vertices $q_a, q_b \in \mathrm{Q}$, symbols $x, y \in \Gamma$, and direction $d \in \mathrm{D}$, there is an edge $(x \mapsto y, d) : q_a \to q_b$ just in case $\delta\left((q_a\ , x)\right.$ -> $\left.(q_b\ , y\ , d)\right)$.

- The notation "$x \mapsto y$" indicates that we *read* x from and *write* y to the current memory location.
- We use "$\sqcup$" as shorthand for an *end* of the list, so "$\sqcup \mapsto y$" means "push y to this end of the list", and "$x \mapsto \sqcup$" means "pop x from this end of the list".
- Multiple symbols on the left of a " $\mapsto$ " is short for reading any one of them, while none stands for reading any symbol (equivalently, not reading at all).
- Leaving the right of a " $\mapsto$ " blank is short for writing the same symbol that was just read (equivalently, not writing at all).

Example: TM State Transition Graph Diagram

Let $\mathrm{M}_{0^n1^n}$ be the TM with input alphabet $\Sigma := \{0, 1\}$ specified by:



This means:

- $\Gamma := \Sigma \cup \{\mathrm{X}, \mathrm{Y}\}$,
- $\mathrm{Q} := \{q_0, \cdots, q_4\}$,
- $q_0 := q_0$,
- $q_h := q_4$,

δ	0	1	X	Y	␣
q_0	$(q_1\,,\mathrm{X}\,,\mathrm{R})$	–	–	$(q_3\,,\mathrm{Y}\,,\mathrm{R})$	–
q_1	$(q_1\,,0\,,\mathrm{R})$	$(q_2\,,\mathrm{Y}\,,\mathrm{L})$	–	$(q_1\,,\mathrm{Y}\,,\mathrm{R})$	–
q_2	$(q_2\,,0\,,\mathrm{L})$	–	$(q_0\,,\mathrm{X}\,,\mathrm{R})$	$(q_2\,,\mathrm{Y}\,,\mathrm{L})$	–
q_3	–	–	–	$(q_3\,,\mathrm{Y}\,,\mathrm{R})$	$(q_4\,,$␣$\,,\mathrm{L})$
q_4	–	–	–	–	–

Turing Machine Operational Semantics

A Turing machine M computes according to the following **operational semantics**.

Given an input string $w \in \Sigma^*$:

- M begins in the start state q_0 with only w in its memory, and its pointer on the first symbol of w (if any).
- M attempts to read from the current memory location, yielding either a symbol from Γ or the information that it has reached the end, represented by $\sqcup$.
- When M is in state q_a and reads symbol x, if $\delta\left((q_a\ ,x) \dashrightarrow (q_b\ ,y\ ,d)\right)$ then M writes symbol y, updates its pointer according to d, and transitions to state q_b.
- If $(q_a\ ,x) \notin \mathrm{dom}(\delta)$, then computation ceases.
- If M is in state q_h when computation ceases then we say that M *accepts* w, otherwise we say that M *rejects* w.

Operational Semantics in Action

Because the state, memory contents, and memory pointer all keep changing, describing the computation of a TM on an input can be verbose and tedious.

We can use a Turing machine simulator, such as the one at `https://turingmachine.io/` to help us visualize it.

```
# decides 0^n 1^n:
input : '00001111'
blank : ' '
start state : q0
table :
  q0 :
    0 : {write : X , R : q1}
    Y : {R : q3}
  q1 :
    [0 , Y] : R
    1 : {write : Y , L : q2}
  q2 :
    [0 , Y] : L
    X : {R : q0}
  q3 :
    Y : R
    ' ' : {L : q4}
  q4 :
    # halt
```

Turing Machine Rant

Following Turing's original 1936 presentation, many sources describe the memory of a TM as "*an infinitely long tape, all but a finite portion of which is blank*".

I think that this is mostly just an artifact of the state of technology in the 1930s (telegraph tape, reel-to-reel tape, etc.).

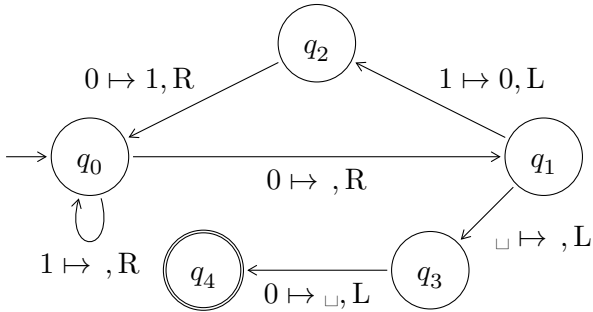
You don't need anything *infinite* to describe a Turing machine.

You just need a doubly-linked list that can be pushed and popped at each end.

Computing Functions with Turing Machines

Because a TM can leave information on the tape when it ceases computation, we can use it to compute partial functions $\Sigma^* \rightharpoonup \Gamma^*$. (Why only partial?)

For example, given two natural numbers in *unary notation* separated by a 0, we can compute their *sum*:



Computing Functions with Turing Machines ctd.

```
# computes the unary sum:
input : '11011'
blank : ' '
start state : q0
table :
  # scan to divider
  q0 :
    1 : R
    0 : {R : q1}
  # decrement second number
  q1 :
    1 : {write : 0 , L : q2}
    ' ' : {L : q3}
  # increment first number
  q2 :
    0 : {write : 1 , R : q0}
  # remove divider
  q3 :
    0 : {write : ' ' , L : halt}
  # halt
  halt :
```

Making Copies

Turing machines can copy their input:

```
# Copies a binary string.
input: '0101'
blank: ' '
start state: sep
table:
  # writes a separator after input
  sep :
    [0 , 1] : {R : sep}
    ' ' : {write : $ , L : ret}
  # reads an input symbol & replaces w/ marker
  rd :
    0 : {write : X , R : s0}
    1 : {write : Y , R : s1}
    $ : {R : halt}
  # scans right to copy a 0
  s0 :
    [0 , 1] : R
    $ : {R : w0}
  # writes a 0 to the end of the copy
  w0 :
    [0 , 1] : R
    ' ' : {write : 0 , L : ret}
  # scans right to copy a 1
  s1 :
    [0 , 1] : R
    $ : {R : w1}
  # writes a 1 to the end of the copy
  w1 :
    [0 , 1] : R
    ' ' : {write : 1 , L : ret}
  # returns to the input & replace last marker
  ret :
    [0 , 1 , $] : L
    X : {write : 0 , R : rd}
    Y : {write : 1 , R : rd}
    ' ' : {R : rd}
  # halts
  halt :
```

Conditionals

Turing machines can branch based on their input:

```
# branches on the current bit
input: '0$0101$1010$'
blank: ' '
start state: br
table:
  # branch on a bit
  br :
    0 : {R : take}
    1 : {R : skip}
  # take the first branch
  take :
    [0 , 1] : {R : take}
    $ : {R : m0}
  # skip the first branch
  skip :
    [0 , 1] : {R : skip}
    $ : {R : next}
  # take the next branch
  next :
    [0 , 1] : {R : next}
    $ : {R : m1}
  # code for machine 0 goes here
  m0 :
    [0 , 1] : R
  # code for machine 1 goes here
  m1 :
    [0 , 1] : R
```

Turing Machine Computation

You can find many more examples of arithmetic and logical functions implemented as Turing machines in the readings and at `turingmachine.io`, including:

- unary multiplication,
- binary addition and multiplication,
- deciding the non-context-free language $a^n\, b^n\, c^n$.

We will see later that any function we know how to compute can be computed, in principle, using this very simple model of computation.

Turing Machine Variants

There are several *variants* of the TM model of computation, including:

- the option to not move left or right during a state transition,
- memory that can grow and shrink at only one end,
- multiple independent memories (or “tapes”),
- a nondeterministic transition relation.

The TM model of computation is quite **robust**, in the sense that these all compute the same class of partial functions.

The way these equivalences are proved is by **simulating** one model in another.

Stationary TM Equivalence

In a stationary TM we have $\mathrm{D} := \{\mathrm{L}, \mathrm{S}, \mathrm{R}\}$, where S indicates that we *stay* on the same memory cell.

We can simulate an (itinerant?) TM in in the stationary TM model by simply not making any stationary state transitions.

To simulate a stationary TM in our usual TM model, each time we want to make a state transition $(x \mapsto y\ , \mathrm{S})$, we instead make a transition $(x \mapsto y\ , \mathrm{L})$, followed by a transition $(\ \mapsto\ \ , \mathrm{R})$.

One-Ended TM Equivalence

In a one-ended TM, we can't make a transition $(\sqcup \mapsto x\ ,\ d)$ after going left, because this *grows* the memory on the left.

Likewise, we can't make a transition $(x \mapsto \sqcup\ ,\ d)$ from the left-most cell (unless it is also the right-most cell), because this *shrinks* the memory on the left.

We can simulate a one-ended TM in our two-ended TM model by simply not making the forbidden transitions.

To simulate a two-ended TM in the one-ended TM model:

- each time we want to *push* a cell containing x to the left end of the memory, we instead *move* the memory contents one cell to the right, then return to the left end and write x to the left-most cell.
- each time we want to *pop* a cell containing x from the left end of the memory, we instead *move* the memory contents one cell to the left overwriting x, then return to the left-most cell.

Multi-Tape TM Equivalence

In an n-tape TM there are n independent memories that we read, write, and move about.

A state transition has the form:

$$(x_1\,, \cdots, x_n) \mapsto (y_1\,, \cdots, y_n), (d_1\,, \cdots, d_n) \qquad \text{where} \quad \mathrm{D} := \{\mathrm{L}, \mathrm{S}, \mathrm{R}\}$$

We can simulate an ordinary TM in the n-tape TM model, so long as $n > 0$, by ignoring all but one tape. (What happens when $n = 0$?)

Simulating an n-tape TM in our usual model is more complex.

One approach uses $2n$ *tracks*, where half of the tracks store tape contents and the other half store head positions.

The details are in Hopcroft §8.4.

Non-Deterministic TM Equivalence

In a nondeterministic Turing machine (nTM), instead of a transition partial function $\delta : \mathrm{Q} \times \Gamma \rightharpoonup \mathrm{Q} \times \Gamma \times \mathrm{D}$ we have a transition relation $\Delta : \mathrm{Q} \times \Gamma \twoheadrightarrow \mathrm{Q} \times \Gamma \times \mathrm{D}$.

By the *powerset trick* this is equivalent to a function that produces a *set* of successor state, symbol written, direction moved triples.

Simulating an ordinary TM in the nTM model is easy: every partial function is already a relation.

Going the other way is tricky. The idea is to use a multi-tape TM (which we know we can simulate with a single-tape TM) and keep track of the list of all of the states that the nTM could be in. If the nTM has more than one possible next state, we add them all to the list. We step each simulation in breadth-first order and halt if any of them does. The details are in Hopcroft §8.4.