TURING MACHINE VARIANTS

BURTON ROSENBERG
UNIVERSITY OF MIAMI

Contents

1. Multi-tape Turing Machines

The theory of Turing Machines is supposed to be a Theory of Computation. If each variant of a Turing Machine had different computing power, then the theory lacks generality. However, a Turing Machine's power is largely unaffected by the details of its definition.

There are two variants of the Turing Machine that interest us especially in this course: the multi-tape Turing Machine and the non-deterministic Turing Machine. We will show them equal in power to the Turing Machine.

To show that a machine model $\mathcal{M}$ equivalent to a Turing Machine by proposing a general method of transcribing $\mathcal{M}$ into a Turing Machine program that calculates the same values $\mathcal{M}$ calculates every step fo the way. This is called a *simulation*. Our Python Turing Machine simulation is a simulation (although one that shows that my

Date: March 26, 2020.

Mac is as powerful as a Turing Machine, since my Mac can simulate in a generalized way any Turing Machine computation).

1.1. **The simulator tape.** In class I have proposed on method of simulation. It consisted of a tape format that interleaves the k tapes of the mutli-tape machine into a single tape. And it consisted of a *soft-state* that drives the transitions in the simulating machine so that it acts, transition by transition, as the multi-tape machine would act.

To summarize the tape discussion, the tape is divided into three areas: the interleaved area, the staging area, and the soft-state area.

1.1.1. *The Interleaved Area.* The the three areas are placed in sequence on the tape, with the interleaved area coming last. It needs to be able to get unboundedly long. The other areas are fixed sized and fixed format.

The interleaved area contains the contents of all k tapes in a cycle so that the zero-th cell of each of the k tapes is placed on into the first $2k$ cells of the interleaved tape (we will explain why $2k$, not k, in a moment), the next cell of each of the k tapes in the next $2k$ cells, and so on.

The interleaved tape will also record the k head positions of all k tapes. We use $2k$ cells for k symbols to leave every other cell for a marker that says whether the head is over that particular symbol. A bit of wizard hackery suggests we do this by leaving the cell either blank, or with the symbol from a set K of k elements that are completely new and distinct symbols. This makes the code for finding a tape head very compact.

$$\delta_{11}\sigma_{11}\delta_{21}\sigma_{21}\delta_{31}\sigma_{31}\dots$$

Where σ_{ij} is the symbol in position j on tape number i; and δ_{ij} is either a blank or the i symbol the set K if the head position on tape i is j.

1.1.2. *The Staging Area.* It can be quite a mess writing a Turing Machine program. One must release one's inner Wizard for all the Hackery possible. My inner Wizard recommended that I break down the simulation into three phases that repeat with each simulation step: *gather*, *soft-state look-up*, *execution*.

The *gather phase* goes out into the interleaved area, finds the symbols under the head of each of the k cells, and copies them into fixed locations in the staging area.

The *execute phase* will find k tape actions and k tape symbols in the staging area,

and will copy the symbols into the interleaved area, and update the head markers in the interleaved area, moving them back or forward by $2k$ steps, according to the actions in the staging area

The staging area is the second area on the tape, just before the interleaved area, and it has format,

$$a_1\sigma_1 a_2\sigma_2 \dots a_k\sigma_k$$

Where a_i is the action on the i tape retrieved from soft-state lookup, or is disregarded in the gather phase, and σ_i is the current symbol on the i tape retrieved by the gather phase, or to be written, in the execute phase.

1.1.3. *The Soft-State Area.* The *soft-state area* contains the state number, written in binary, left-justified in a fixed width field. The width is set during the construction of the simulator to be wide enough to contain the largest state number. It is the the leftmost area on the tape and is bounded on both sides by a dedicated tape marker. In my implementation the colon, that we have been using as the left-end of tape marker.

$$: b_1b_2b_3 \dots b_n :$$

Where ":" is a dedicated symbol, and b_i is either 0, 1 or a blank. It is a unique string for each state in the multi-tape machine we are trying to simulate, and can be thought of as if we numbered the states in that machine, and this is the state number.

For convenience of the simulator, the string of all zeros is the unique accepting state; the string of all ones is the unique rejecting state, and the start state is state number 1.

1.1.4. *Summary.*

$$: b_1b_2b_3 \dots b_n : a_1\sigma_1 a_2\sigma_2 \dots a_k\sigma_k\delta_{11}\sigma_{11}\delta_{21}\sigma_{21}\delta_{31}\sigma_{31} \dots$$

Where,

- ":" is a dedicated symbol marking the left end of the tape, and the start and end of the soft-state area,
- b_i is either 0, 1 or a blank, and represents the current state of the simulated machine,
- a_i is the action on the i tape retrieved from soft-state lookup, or is disregarded in the gather phase,
- σ_i is the current symbol on the i tape retrieved by the gather phase, or to be written, in the execute phase,

- δ_{ij} is either a blank or the i symbol the set K if the head position on tape i is j,
- and σ_{ij} is the symbol in position j on tape number i.

1.2. **Soft-state lookup.** The gather and execute phases are programmed in the notebook (class CSC 427, Spring 2020). Here I discuss how the simulator captures the computation of the simulated machine, and carries out all its actions in proxy.

The essential idea is that for every state transition in the simulated machine, we create a state in the simulator. For each transition,

$$(q, \sigma_1, \sigma_2, \ldots, \sigma_k) \longrightarrow (q', \sigma'_1, a_1, \sigma'_2, a_2, \ldots, \sigma'_k, a_k)$$

Create a state and give it a number.

The simulating machine navigates to a given state, which is a state-symbols combination in the multi-tape machine, by scanning left to right the soft-state area, branching one way or another on 0 or 1, and skipping over the training blanks until the colon-marker, and then continuing its branching as it passes over each of the current tape symbols in the staging area. This is always a fixed number of symbols, k. At the end of this, the simulator is now in the required state.

From that state, issue a sequence of transitions that write q' into the soft-state area, overwriting q, and writes the σ_i and a_i into the staging area. Then we are done with the lookup.

In fact, what we will have in our Turing Machine simulator, is a capture of the state transition information of the multi-tape machine to simulate, in a large tree. The top part is a binary tree and is navigated downwards going left or right according to the 0-1 pattern in the soft-state area (a 0-1 pattern is also known as a binary number). The the bottom part will branch according to the family of tape symbols in each of the k tapes.

This is a totally general structure. And the rest of the Turing Machine simulator is a generic house-keeping of the tape, the gather and execute phases, and with each round of simulation, dropping through this tree to map the input to the transition to the output of the transition.

2. A Universal Machine

We introduce the notion of a *Universal Turing Machine*, and suggest that it is an alternative to the above construction. From the universal T.M. falls out easily the

simulation of a DFA on a T.M., giving the (perhaps unsurprising) result that T.M. compute a superset of DFA's.

— The idea of a universal machine is completely familiar. It is a common computer. Our Turing Machines are hard wired up to do a task. Early computers, such as the IMB 407 did this too, figure 4 figure 5 figure 6.

However, now programs are sent as text, and the text and data both reside in the memory, on "the tape". A Turing Machine can be created that is in this way programmable. We can do it with the idea of the multi-tape simulator described in the previous section. What we do differently is rather creating a complex of states to encode the transitions, we leave a textual description of the collection of transitions on a separate, read-only, tape.

Given a transition,

$$(q, \sigma_1, \sigma_2, \ldots, \sigma_k) \to (r, a_1, \tau_1, a_2, \tau_2, \ldots, a_k, \tau_k)$$

encode this as a sequence of tape symbols,

$$\langle q \rangle \sigma_1 \sigma_2 \ldots \sigma_k \langle r \rangle a_1 \tau_1 a_2 \tau_2 \ldots a_k \tau_k$$

where $\langle q \rangle$ is the binary representation of the numbering of state q, written zero padded to a fixed length. This length is large enough to contain the largest state number. That way the encoding of every state transition share a common length.

A separate tape contains these encodings, one after the other. An left end of tape marker is useful, as each simulation step requires going left to left tape end, then attempting to match the encoded transition against the soft state and staging area of the simulation tape.

When a match is found, the second half of the transition encoding is copied to the soft state and staging area. Hence our universal Turing Machine is a simple modification of the multi-tape simulator, replace hard state to wire up the transitions with a list of symbols, that are matched under program control.

2.1. **Example.** FIgure 2 gives an example of a Turing Machine program. This says how a Turing Machine should be built. It gives the states the TM will have, how transitions go between states, and the actions taken on transition.

We want to reformat that language for our universal machine. First we number the states, as given in Table 3. Then we write out a long stream of symbols, the concatenation of encodings for every state transition in the description, see Figure 1.

```
# A-star-B-star:
# a*^b^*

start: s
accept: a
reject: r

state: s
    _ _ n a # accept the empty string
    a a r a_seen
    b b r b_seen

# any number of a's and end or some b's
state: a_seen
    a a r a_seen
    _ _ n a
    b b r b_seen

# any number of b's until end
state: b_seen
    b b r b_seen
    _ _ n a
    a a n r
```

FIGURE 1. Turing machine program in our syntax

state	**number**	**code**
s	0	0000
a	1	0001
r	2	0010
a_seen	3	0011
b_seen	4	0100

FIGURE 2. Numbering the states

3. Simulation of a DFA

Simulation of a DFA by a Turing Machine is immediate from the previous section.

Use the above construction, however the state descriptor tape contains only a sequence of the simplified elements,

$$:: \langle q_1 \rangle \sigma_1 \langle q_1' \rangle : \langle q_2 \rangle \sigma_2 \langle q_2' \rangle : \ldots : \langle q_n \rangle \sigma_n \langle q_n' \rangle$$

```
0000_0001_r
0000a0011ar
0000b0100br
0011a0011ar
0011_0001_n
0011b0100br
0100b0100br
0010_1000_n
0100a0010an
```

FIGURE 3. Program in tape format for universal machine

4. Nondeterministic Turing Machines

In both the Finite Automata and Push Down Automata, we gave thought to *nondetermism*. This important concept in computing will continue to intrigue us, for at least as we explore the P versus NP problem.

The prior results were that adding non-determinism to a finite automata is inconsequential to the language that results. The quick proof is that we can lift the machine from a machine of states to a machine of sets of states, and lift with that the transition function to work simultaneously on all states interior to a set of states.

In this was we follow the parallel threads of a nondeterministic finite automata, simultaneously, in the trajectory of a sequence of single states. The parallelism is bounded. If there are n states in the original machine, the maximum parallelism is 2^n threads of computation, one for each state combination possible after each step in the machine.

The situation of Push Down Automata's was different. If one insists that it proceed deterministically, the class of languages recognizable is restricted. Compilers need deterministic context free grammars, and these are not the general context free grammar. But compiler writers will deal with any inconvenience by cheating.

For instance, C language is not even context free, because of the typedef class (see section 5.10.3, Harbison and Steele, Fifth edition).

For Turing Machines, the class of languages recognizable (or decidable) is not changed by adding non-determinism to the Turing Machine definition.

4.1. **The Turing Search Machine.** We will prove this by the introduction of what a call the *Turing Search Machine*. A non-deterministic TM program is easily re-written for a Turing Search Machine. The machine will then weed through the non-determinism, choice by choice, until it either finds a halting state (accept or reject), or continues indefinitely.

Add to the Turing Machine simulator at,
`https://github.com/burtr/Workbook/tree/master/fa-sim`
the following modifications.

(1) There will be three tapes,
   (a) The usual work tape,
   (b) A read only input tape. The computation begins with the input written onto this tape, and it is never modified.
   (c) A search tape which enumerates one by one, breath first, all possible non-deterministic choices to be made during a run.
(2) The tag vocabulary will be broadened to include a "choice" tag. A choice tag has a name argument, as does the state tag; and the name can be used interchangeably with a state name.
(3) The line following a choice tag gives three states or choices. Which of three becomes the next state depends on the symbol under the head of the search tape.
(4) The search tape is a binary sequence of finite length. Following the last 0 or 1 is an (effectively) infinite sequence of blanks. It is initialized at the beginning of the computation with a single zero in the leftmost cell of the tape, and the head above it.
(5) The search tape supports the following operations,
   (a) It can be queried during a run using a "choice" transition. The choice transition will look at the character under the head of the search tape, and go to one of three states depending on whether the head is over a 0, 1 or blank. This action also advances the head one to the right.
   (b) A count action. This will bring the head of the search tape back to the left end and will increment the binary string on the tape to the next binary sequence numerically. The count action will also have other effects (described separately).
(6) The list of actions is expanded from L, R, N to include the new action: Z, for "Zählen" (in honor of Turing naming his paper in German).

4.2. **Example.** Consider the recognition of $(ab)^*(ac)^*$. This would be done on a NFA with two non-deterministic jumps — one after each ab whether to return to

look for another *ab* or move on to look for *ac*; likewise whether another *ac* should be matched or do we predict the end of the input.

On a search machine, here we go:

```
start :s
accept: a
reject: r

state: s
a a N c1

state: ab
a a R ab_1

state: ab_1
b b R c1

state: ac
a a R ac_1

state: ac_1
c c R c2

choice: c1
ab ac  try_again

choice: c2
ac end  try_again

state: end
_ _ N a

state_ try_again
_ _ Z s
a a Z s
b b Z s
```

5. Appendix

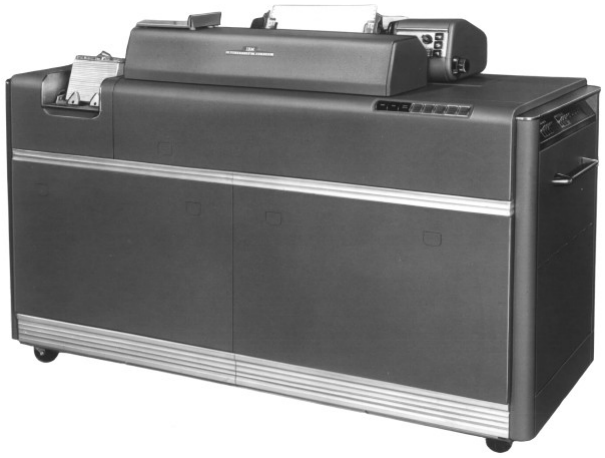

Figure 4. From: http://www.columbia.edu/cu/computinghistory/407.html

Figure 5. From: http://www.columbia.edu/cu/computinghistory/407.html

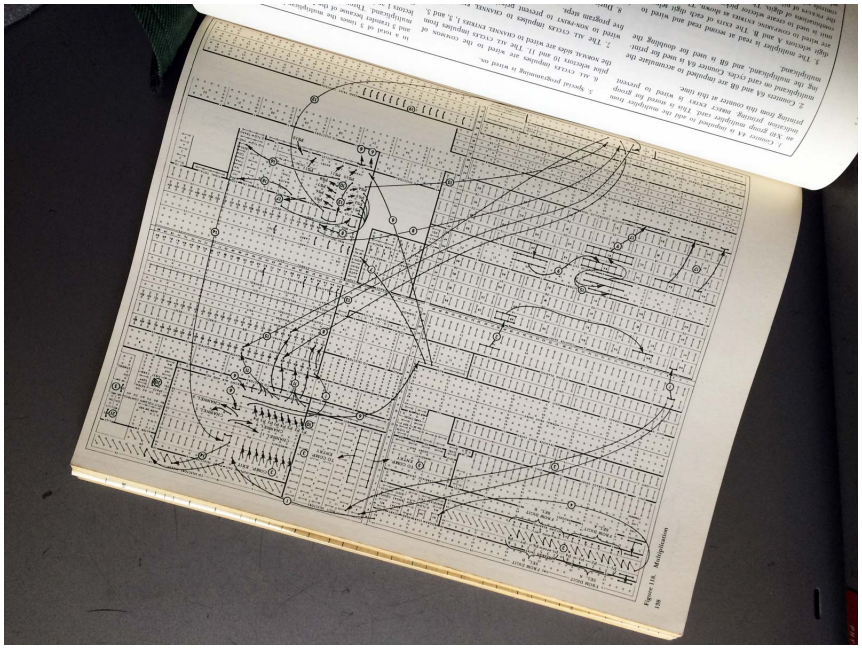

FIGURE 6. From: http://www.columbia.edu/cu/computinghistory/407.html