Turing Machine Variants

It slices!
It dices!
It makes julienne fries!
Robustness

Invariance to certain changes in the model of computation

Thought Experiment:

If we changed the transition function to be $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$, where $S$ means to stay put, would our model be more “powerful”?

What is the significance of this realization?
Variant: Multitape Turing Machine

Like an ordinary Turing Machine, but with multiple tapes.

Each tape has its own head for reading and writing.

Initially the input appears on tape 1, and other tapes are blank.

Transition function is changed for reading, writing, and moving heads on some or all of the tapes simultaneously.

$$\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k,$$

Where $k$ is the number of tapes. The expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots L)$$

is a transition from state $q_i$, and heads 1 through $k$ are reading symbols $a_1$ through $a_k$, which results in a transition to state $q_j$, writing symbols $b_1$ through $b_k$, and moves (left, right, or stay put) as specified.

Is this more powerful than a regular Turing Machine?
Variant: Multitape Turing Machine: Theorem

Convert multitape TM $M$ to an equivalent single-tape TM $S$.

Say $M$ has $k$ tapes. $S$ simulates the effect of $k$ tapes by storing their information on its single tape.

Requires new symbol $#$ as a delimiter.

$S$ must keep track of the locations of all heads.

Consider a “dotted” symbol to represent the head position.

Representing three tapes with one.

What considerations need to be made?
Variant: Multitape Turing Machine: Theorem

On input $w = w_1 \ldots w_n$:

First $S$ puts its tape into the format that represents all $k$ tapes of $M$. The formatted tape contains:

$\#w_1w_2 \cdots w_n \#\# \# \cdots \#$

To simulate a single move, $S$ scans its tape from the first $\#$ to the $(k + 1)$st $\#$ two times.

1. First pass is to determine symbols under the heads, to determine the transition.
2. Second pass is to update the tapes according to transition function.

If at any point $S$ moves one of the virtual heads to the right onto a $\#$, this action signifies that $M$ has moved the corresponding head onto the previously unread blank portion of that tape.

$S$ must then write a blank symbol on this cell and shift the tape contents, from this cell until the rightmost $\#$, one unit to the right.

Finally, return to the newly blank cell that was just created, and continue the simulation.
4 Questions:

- **What** did we just prove?
- **How** did we do it?
- Do we have to prove the “other direction”? Why or why not?
- **Why** would you use a MTM?
A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

Proof:

- **Direction 1**: A single-tape TM is a special case of a multitape Turing machine.
- **Direction 2**: A multitape TM can be simulated on a single-tape TM (previously shown).
Variant: Nondeterministic Turing Machines

At any point in a computation, the machine may proceed according to several possibilities.

Transition function is of the form

$$\delta: Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Computation is a tree whose branches correspond to different possibilities for the machine. If some branch leads to the accept state, the machine accepts its input.

Can a Nondeterministic TM be simulated by a Deterministic TM?
Nondeterministic Computation

How is it represented?

“Computation is a tree whose branches correspond to different possibilities for the machine.”

All branches must be examined until an accept state is found.

Breadth-first, or depth-first?
Nondeterministic Computation

How is it simulated?

What would we need to track to perform the simulation?

- A **tape** to perform the simulation on.
- Where we are (**execution state**) in the nondeterministic computation tree.
- A way to “go back” or “reset”
Variant: Nondeterministic Turing Machines: Theorem

“Every nondeterministic TM $N$ has an equivalent deterministic TM $D$.”

$N$’s computation on $w$ is seen as a tree.

- Each branch is a branch of nondeterminism
- Each node is a configuration of $N$
- The root is the start configuration.

$D$ tries all possible branches of $N$’s computations until an accept state is found, using a breadth-first search.

$D$ has 3 tapes.

1. Input string, which is never altered.
2. Copy of $N$’s tape on some branch of computation. The simulation “tape”.
3. $D$’s location in $N$’s computation tree. The simulation “state”.

![Diagram](image-url)
Understanding Tape #3

Every node is assigned an address that is a string over the alphabet $\Gamma_b = \{1, 2, ..., b\}$, where $b$ is the most children possible for any node in the computation tree.
Understanding Tape #3

Every node is assigned an address that is a string over the alphabet $\Gamma_b = \{1, 2, \ldots, b\}$, where $b$ is the most children possible for any node in the computation tree.

Node 113, then, is the node reached by taking the first computation branch, followed by that node’s first computation branch, and then finally that node’s third computation branch.

Notice how this ordering can be used to efficiently traverse the computation tree in a breadth-first manner.
Variants: Nondeterministic Turing Machines: Theorem

How $D$ functions:

1. Initially, tape 1 contains input $w$, and tapes 2 and 3 are empty. (Tape 3, being empty, is $\varepsilon$.)
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate $N$ with input $w$ on one branch. Before each step of $N$, consult the next symbol on tape 3 to determine which choice to make (among those allowed).
4. (cont.) If no more symbols remain on tape 3, or if this choice is invalid, abort by going to step 4. Also go to step 4 if a rejecting configuration is encountered. If an accepting configuration is found, accept the input.
4. Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of $N$'s computation by going to step 2.
A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

Proof:

- **Direction 1**: Any deterministic TM is automatically a nondeterministic TM.
- **Direction 2**: A nondeterministic TM can be simulated on a deterministic TM (previously shown).
Which of these was just proved?

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

A language is Turing-decidable if and only if some nondeterministic Turing machine decides it.

What changes need to be made for both to be true?
Variant: Enumerators

An **enumerator** is a Turing Machine with an attached printer.

The TM can send **strings** to the printer.

This is where the name **recursively enumerable language** comes from (discussed earlier as an alternative for the name **Turing-recognizable language**).

**Characteristics:**
- An enumerator $E$ starts with blank input on its work tape.
- If $E$ doesn’t halt, it may print an infinite list.
- The list may be in any order.
- The list may contain duplicates.
Variant: Emulators: Proof

“A language is Turing-recognizable if and only if some enumerator enumerates it.”

Part 1, Use $E$ to simulate a TM $M$:

If $E$ enumerates language $A$, a TM $M$ recognizes it in the following way on input $w$:

1. Run $E$. Compare all strings output by $E$ with $w$.
2. If $w$ ever appears in the output of $E$, accept.

Clearly, $M$ accepts those strings that appear on $E$’s list.

Part 2, use $M$ to simulate an enumerator $E$:

If TM $M$ recognizes a language $A$, we can construct an enumerator $E$ for $A$. Say $s_1, s_2, s_3, \ldots$ is a list of all possible string in $\Sigma^*$. For $E$, ignore the input and repeat the following for $i = 1, 2, 3, \ldots$:

1. Run $M$ for $i$ steps on each input, $s_1, s_2, \ldots, s_i$.
2. If any computations accept, print out the corresponding $s_j$.

If $M$ accepts string $s$, eventually it will appear on $E$’s list.
Equivalence With Other Models

With only a handful of requirements, all models have equivalent power

- Unrestricted access to memory
- Unlimited memory
- Limit of performing finite amount of work in a single step.

Models may be very different, but still have equivalent power.

Our stepping stone to more practical applications.