Lecture 16. Extensions of TM and the Church-Turing Thesis

Extensions of TM

1. multiple tapes
2. two-way infinite tape
3. random access
4. nondeterminism

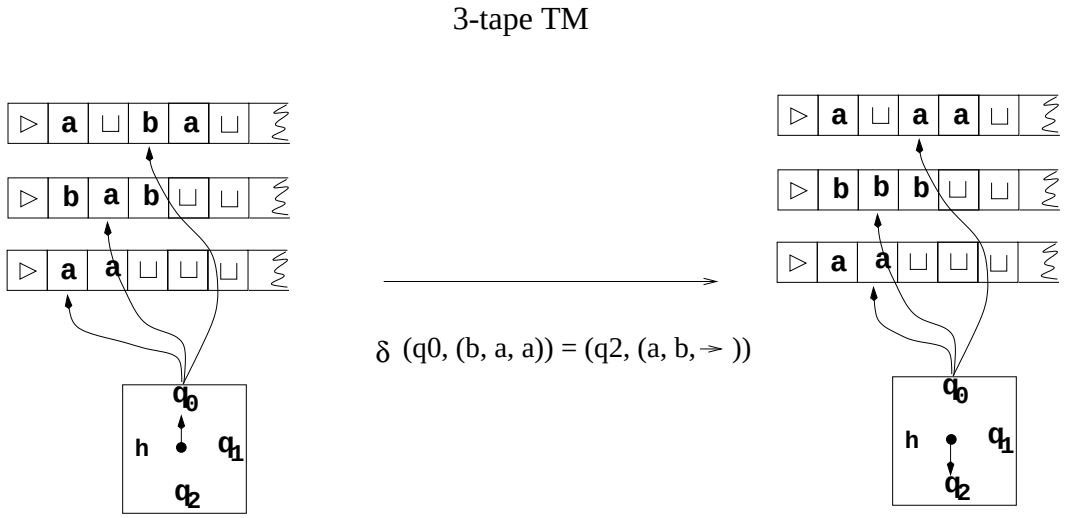
Theorem 1 *The operation of a machine allowing some or all of the above extensions can be simulated by a standard Turing machine. These extensions do not produce more powerful machines than the standard Turing machine.*

That is, any language that can be decided (or semidecided) by these extensions can also be decided (or semidecided) by a standard Turing machine.

Note: These extensions are nevertheless helpful when designing Turing machines to solve specific problems.

Multitape Turing Machine

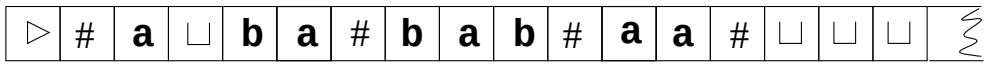
- Each tape has its own read/write head.
- Initially, the input appears on tape 1, and the other tapes start out blank.
- In each step, the TM reads the symbols pointed to by all its heads simultaneously.
- Depending on its current state and all the symbols read, the machine decides *for each head* whether to
  - write onto the current square, or
  - move $\leftarrow$, $\rightarrow$



Theorem 2 *Every multitape Turing machine has an equivalent single-tape Turing machine.*

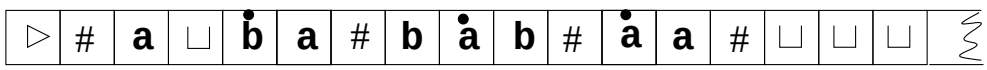
Proof:

- We shall prove that any k-tape TM M_1 can be simulated by a standard Turing machine M_2.
- M_2 uses the new symbol # as a delimiter to separate the contents of the different tapes.



- M_2 keeps track of the locations of M_1's heads by writing the tape symbol σ with a new symbol $\dot{\sigma}$ to mark the place where the head on that tape would be (i.e., add new symbols $\dot{a}, \dot{b}, \dot{\sqcup}$ to Σ for this example). Think of these as "virtual" tapes and heads.

  For example, for the configuration of 3-tape machine shown in previous slide:



Suppose M_1 has $\sigma_1 \cdots \sigma_n$ on its first tape initially.

1. M_2 puts its tape into the following format:

$$\# \dot{\sigma}_1 \ \sigma_2 \cdots \sigma_n \# \dot{\sqcup} \# \dot{\sqcup} \cdots \#$$

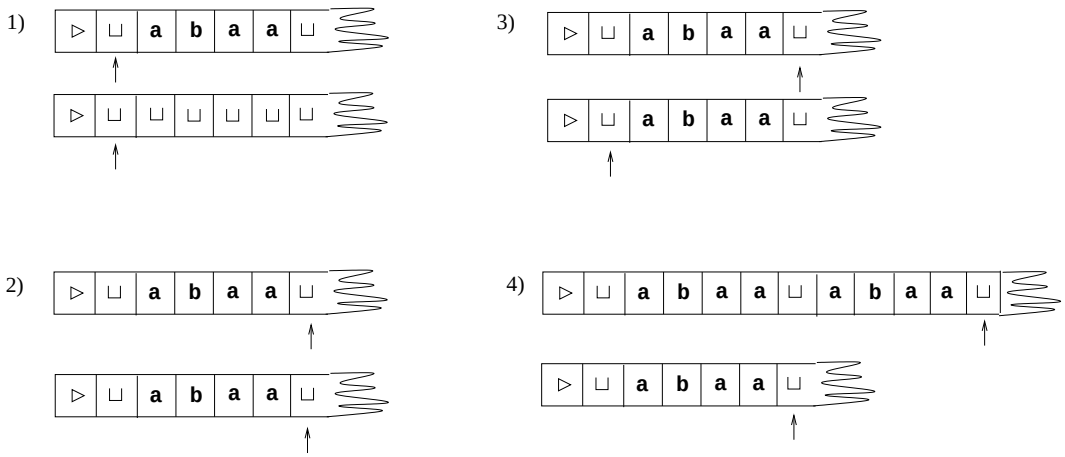
2. To simulate a single move of M_1, M_2 scans its single tape from the first # to the $(k+1)$st # and determine the symbols under the virtual heads. Then M_2 makes a second pass to update the tape cell/head at k places according to the way M_1's transition function dictates.

3. If at any point M_2 moves one of the virtual heads to the *right* onto a #, this action means that M_1 has moved the corresponding head onto the previously unread blank portion of that tape. So M_2 writes a blank symbol on this tape cell and shifts all the tape content from that cell until the rightmost #, one position to the right. Then it continues the simulation as before.

Reference: Michael Sipser's. See textbook for an alternate proof.

Example of Multitape TM

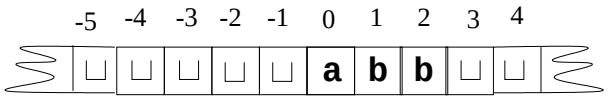
Even though k-tape TMs cannot decide any additional language than single tape TMs, it is often easier to design k-tape Turing machines than single-tape ones.
Example: Copying machine – easier to do with 2-tape machine than with 1-tape machine.



1. While moving the heads of both tapes to the right, copy each symbol on first tape to second tape, until a blank is found on first tape.
2. Move head of second tape all the way to the left until ⊔ is found.
3. While moving both heads to right, copy each symbol from second tape onto first tape. Halt when blank is found on second tape.

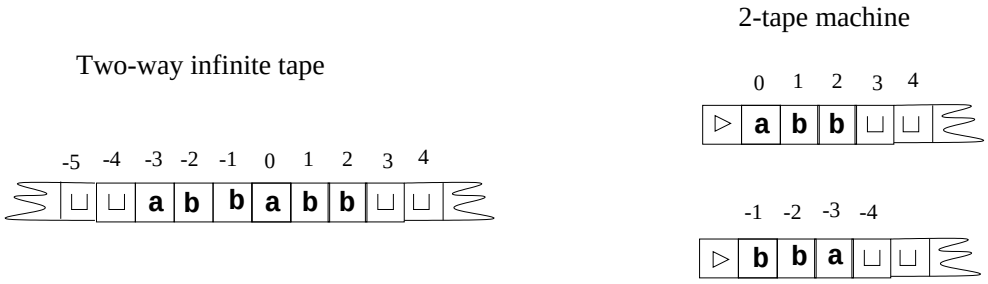
Two-way infinite tape

- Initially, all squares are blank, except for those containing the input string.



Can be simulated by a 2-tape machine:

- One tape always contains the part of the two-way tape starting from the first input symbol and the rest on its right;
- the other tape contains the part of the tape to the left of first input symbol, in reverse order.



This 2-tape machine can in turn be simulated by a standard TM.

Random Access TMs

A **random access TM** has

- k **registers**, $R_0, R_1, \ldots, R_{k-1}$, each holding a natural number; a special register is the **program counter**.
- a one-way infinite tape $T[1], T[2], T[3], \cdots$, each holding a natural number.
- a **program** which is a finite sequence of instructions.

Initially, all registers are 0, and the tape contains the input.

A random access TM is described by a **program** that consists of a sequence of **instructions** $\pi_1, \pi_2, \ldots, \pi_p$ (arithmetic operations on the registers, writing a number from a register to a tape cell, loading a number from a tape cell to a register, jump, conditional jump, halt).

Execution: The **program counter** $\kappa = 1$ initially. Then the machine executes the instruction π_κ, and set $\kappa \leftarrow \kappa + 1$ unless π_κ is a jump (or conditional jump and the condition is met). The machine halts when it reaches a halt instruction. When the machine halts, it **accepts** the input if $R_0 \neq 0$ and **rejects** if $R_0 = 0$.

A random access TM can easily simulate a standard TM: Use a register to keep track of the head position, another register to store the state. Each state has a corresponding sequence of instructions. In each step: load the tape cell, branch according to the current state, then update the state register, and either update the cell or the head position register according to the transition function.

Theorem 3 *Any language decided (or semidecided) by a random access TM can be decided (or semidecided) by a standard TM.*

Idea of proof: We will use a $(k+3)$-tape TM M' to simulate. The first tape stores the input (in binary). The second tape stores $T[1], T[2], \ldots$, in the following format: (10,1110), (111,101), (11,111), ..., (1111010,10010)\$, i.e., for each nonzero $T[i]$, the pair $(i, T[i])$ is stored somewhere (in binary) on this tape. The symbol "\$" marks the end of the tape. The next k tapes store the values of the k registers. The last tape is used as a "scratch tape". The program counter κ is encoded in the states of M'. This way, each instruction can be simulated.

Note that M' takes time (number of steps) polynomial in the running time of the random access TM and the input size.

Nondeterministic TM

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

$M = (K, \Sigma, \Delta, s, H)$

where K, Σ, s, H are as for standard TMs,

but Δ is a subset of $((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\rightarrow, \leftarrow\}))$
(i.e., it is a relation, not a function.)

The computation of a nondeterministic TM can be visualized as a tree whose branches correspond to different possibilities for the machine.

If at least one branch of the computation leads to an accepting configuration, the machine accepts the input.

Nondeterministic TM

A nondeterministic TM M is said to *decide* a language L if the following holds:

1. if $w \in L$, then *all* of M's nondeterministic computations on w halt and *one* of the computations halts in the "y" state.
2. if $w \notin L$, then *all* of M's nondeterministic computations on w halt and *none* of the computations halts in the "y" state.

A nondeterministic TM M is said to *semi-decide* a language L if the following holds:

- if $w \in L$, then *one* of M's nondeterministic computations on w halts.
- if $w \notin L$, then all of M's computations on w do not halt.

Theorem 4 *If a nondeterministic TM semi-decides (or decides) L, then there exists a standard TM that semi-decides (or decides) L.*

Idea of proof: We can simulate any nondeterministic TM N with a deterministic TM D.

- View N's computation on an input w as a tree. Each node is a configuration of N. The root of the tree is the start configuration.
- TM D searches this tree for an accepting configuration.
- A bad idea is to have D explore the tree using depth-first search, because D could go forever down one infinite branch and miss an accepting configuration on some other branch.
- Hence, we must design D to explore the tree using breadth-first search.

Proof

TM D has 3 tapes (proved to be equivalent to 1-tape TM).

- Tape 1 always contains the input string and is never altered.
- Tape 2 contains a copy of N's tape content while it takes a branch of its nondeterministic computation.
- Tape 3 keeps track of D's location in the tree.

Data representation of Tape 3

- Let b be the largest number of possible computation choices at any configuration given by N's transition function (maximum is $|K| \times |\Sigma \cup \{\rightarrow, \leftarrow\}|$). That is, b is the maximum number of children any node in N's nondeterministic computation tree can have.
- Assign an address to every node in the tree level by level. The address is a string over $\Sigma_b = \{1, 2, ..., b\}$. The address of the root node is e, the address of the root's first child is 1, that of the root's second child is 2, etc. For example, if a node has address 231, then it means we arrive at it by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child.
- Some addresses are invalid, i.e., do not correspond to any node in tree. Such an address has some symbol in the

address string that does not correspond to a real choice because a configuration has fewer than b choices.

- Tape 3 contains a string over Σ_b (i.e. an address, which may or may not be valid).
- If the address in Tape 3 is valid, D starts simulating the branch of N's computation *from the root* to the node having that address.

Description of D:

1. Initially Tape 1 contains the input w, and Tape 2 and 3 are empty.
2. Copy Tape 1 to Tape 2. (Tape 3 contains empty string which is the address of the root of N's computation tree).
3. Use Tape 2 to simulate N with input w along the branch of computation indicated by the address in Tape 3. Before each step of N, consult the next symbol on Tape 3 to determine which choice to make. If no more symbols remain on Tape 3, or if this nondeterministic choice is invalid, or if a rejecting configuration is encountered, go to 4. If an accepting configuration is encountered, halt (*accept* the input).
4. Replace the string on Tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to 2.

Note that if one of N's non-deterministic computations on w halts at y, then D will also halt at y on w. If all of N's non-deterministic computations on w halt at n, then D will also halt at n. If all of N's non-deterministic computations loop, then D will also loop. That is, D decides/semi-decides the same language as N.

The Church-Turing Thesis

- There are many ways to define variants of TM to make them easier to program, but all of them have been proved to be as powerful as the standard TM model.
- There are other computational models (e.g., unrestricted grammars, Church's λ-calculus, Post production systems, Markov algorithms, and recursive functions). All of these systems cannot define more languages or solve more problems than the standard TM model.
- The Church-Turing Thesis says that if some algorithm exists to carry out a computation (decides a language or computes a function), then the same computation can also be carried out by a Turing machine.
- The Church-Turing thesis is a statement that characterizes the nature of computation and cannot be formally proved. Nevertheless, this hypothesis now has near-universal acceptance.