Turing Machine Variants

Arjun Chandrasekhar

Turing Completeness

- **Def:** A model of computation is **Turing complete** if it is equivalent to the Turing machine model
- We need to show that a language can be recognized by a Turing machine if and only if it can be recognized by a machine from the other model
- This involves two directions:
  1. Show that every language that can be recognized by a Turing machine can be recognized by a machine from the other model
  2. Show that every language that can be recognized by a machine from the other model can be recognized a Turing machine

Stationary Turing Machine

- **Def:** A **stationary Turing machine (stationary TM)** is a like a normal Turing machine, but the machine has the option to stay in place after reading a character.
- The transition function is $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$

Stationary Turing Machine

- Let's prove that stationary Turing machines are equivalent to Turing machine.
  - That is, L can be recognized by a Turing machine if and only if L is recognized by a stationary TM
- There are **two** directions to this proof
  1. If L is recognized by a normal TM, it can recognized by a stationary TM
  2. If L is recognized by a stationary TM, it can be recognized by a TM

Stationary Turing Machine

($\Rightarrow$) If L is recognized by a normal TM, it can be recognized by a stationary TM

- Let M be the TM that recognizes L
- M **is** a stationary TM that simply chooses not to stay in place
- Thus, L can be recognized by a stationary TM

Stationary Turing Machine

($\Leftarrow$) If L is recognized by a stationary TM, it can be recognized by a normal TM

- Let M be the stationary TM that recognizes L
- **Technique:** create a normal TM that simulates M
- Create a machine M_2 behaves as M would, with one exception
- If M is supposed to stay in place, M_2 will move left and then move right before proceeding

2-hop Turing Machine

- **Def:** A **2-hop Turing Machine** is a like a normal Turing machine, but it can move up to 2 spaces left or right.
  - The transition function is $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, LL, RR\}$
- Let's prove that 2-hop Turing machines equivalent to Turing machines.
  - What are the two directions?

2-hop Turing Machine

(⇒) If L is recognized by a normal TM, it can be recognized by a 2-hop TM

- Let M be the TM that recognizes L
- M **is** a 2-hop TM that chooses to only move one square at a time.

2-hop Turing Machine

($\Leftarrow$) If L is recognized by a 2-hop TM, it can be recognized by a normal TM

- Let M be the 2-hop TM that recognizes L
- We will design a normal TM M_2 to simulate M
- M_2 operates as M would. If M tries to hop two spaces right or left, M_2 will perform the two hops over two consecutive steps

2-tape Turing Machine

- **Def:** A **2-tape Turing machine** is a TM with 2 different tapes
  - Each tape has a separate tape head
  - The two tape heads share a common state, but they move independently

2-tape Turing Machine

2-tape Turing Machine

- **Def:** A **2-tape Turing machine** is a TM with 2 different tapes
  - Each tape has a separate tape head
  - The two tape heads share a common state, but they move independently
  - The transition function is $\delta : Q \times \Gamma^2 \to Q \times \Gamma^2 \times \{L, R\}^2$
- Let's prove that 2-tape Turing machines are equivalent to (1-tape) Turing machines
  - What are the two directions?

2-tape Turing Machine

($\Rightarrow$) If L is recognized by a 1-tape TM, it can be recognized by a 2-tape TM

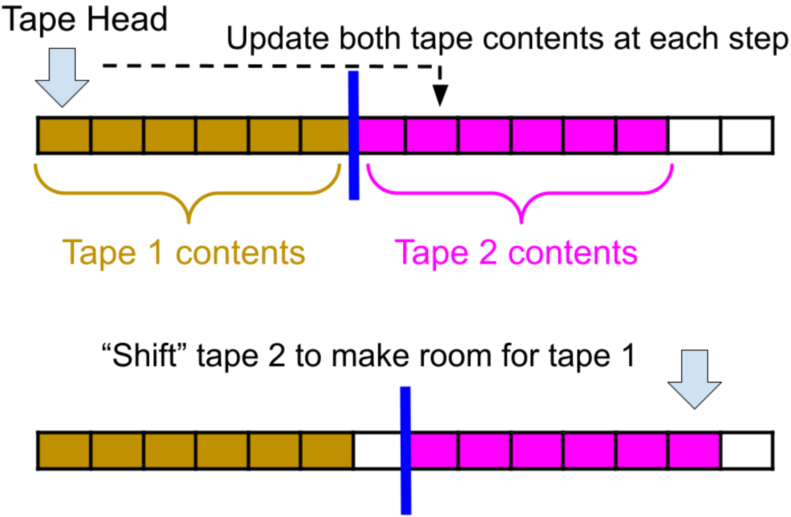
- Let M be the 1-tape TM that recognizes L
- M **is** a 2-tape TM that ignores the second tape

2-tape Turing Machine

($\Leftarrow$) If L is recognized by a 2-tape TM, it can be recognized by a 1-tape TM

- Let M be the 2-tape TM that recognizes L
- We will design a 1-tape TM called M_2 to recognize L
- M_2 will use its single tape to keep track of both of M's tapes.
- At every step, M_2 simulates both tape heads of M
- If needed, M_2 can always push the second tape farther downstream to make more room for the first tape

2-tape Turing Machine



Non-deterministic Turing Machine

- **Def:** A **non-deterministic Turing machine** is a normal TM, but it can make several different choices at each step
  - The transition function is $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$
  - There are many possible computation paths for the same string
  - The machine accepts if at least one computation path accepts
- Let's show non-deterministic TMs are equivalent to deterministic TMs
  - What are the two directions?
  - Keep in mind that some computation paths may not halt

Non-deterministic Turing Machine

($\Rightarrow$) If L is recognized by a deterministic TM, it can be recognized by a non-deterministic TM

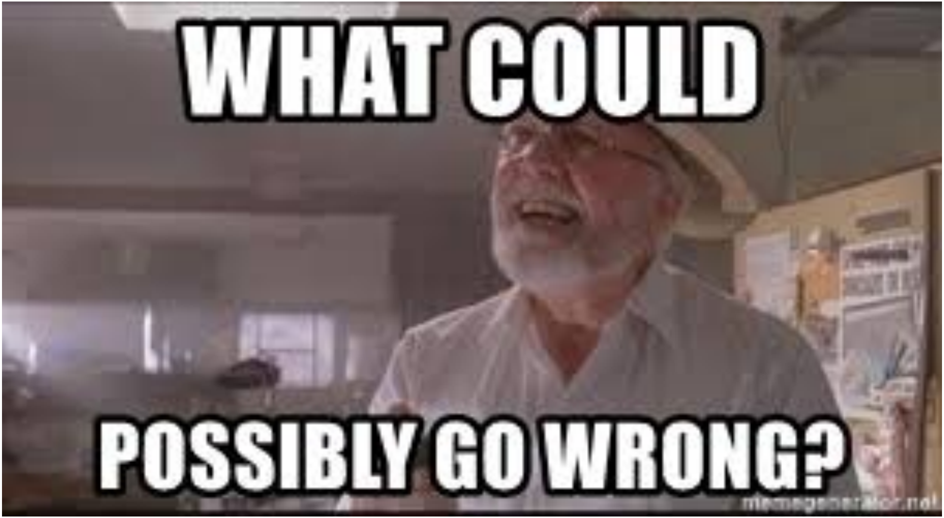
- Let M be the machine that recognizes L
- M **is** a non-determinstic TM that only has one computation path

Non-deterministic Turing Machine

($\Leftarrow$) If L is recognized by a non-deterministic TM, it can be recognized by a deterministic TM

- Let M be the non-deterministic TM that recognizes L
- Design a deterministic TM called M_2 to simulate M
- We run M, and try all possible computation paths.
- If any computation path accepts, M_2 accepts, otherwise it rejects

Non-deterministic Turing Machine

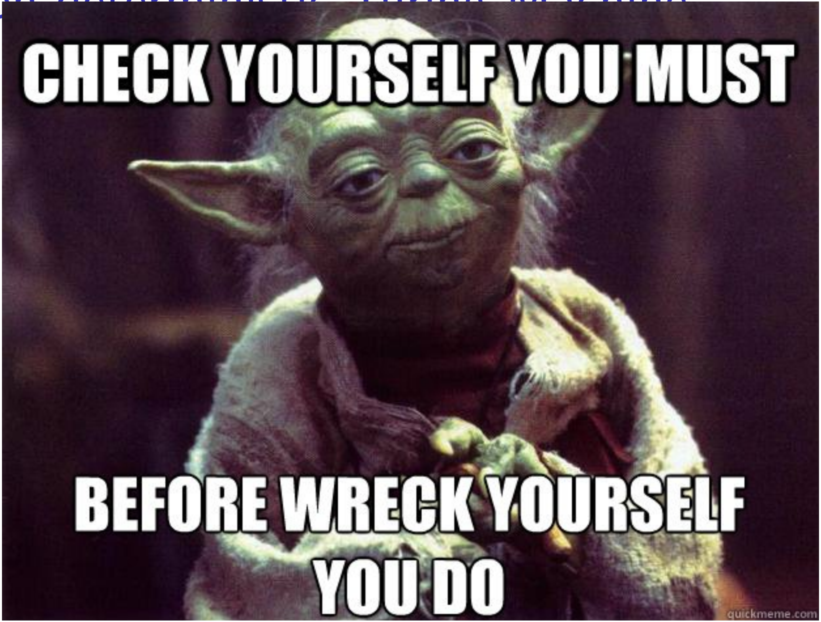


Non-deterministic Turing Machine

($\Leftarrow$) If L is recognized by a non-deterministic TM, it can be recognized by a deterministic TM

- Let M be the non-deterministic TM that recognizes L
- Design a deterministic TM called M_2 to simulate M
- ~~We run M, and try all possible computation paths.~~
  We might get stuck on a path that loops forever!
- If any computation path accepts, M_2 accepts, otherwise it rejects

Non-deterministic Turing Machine

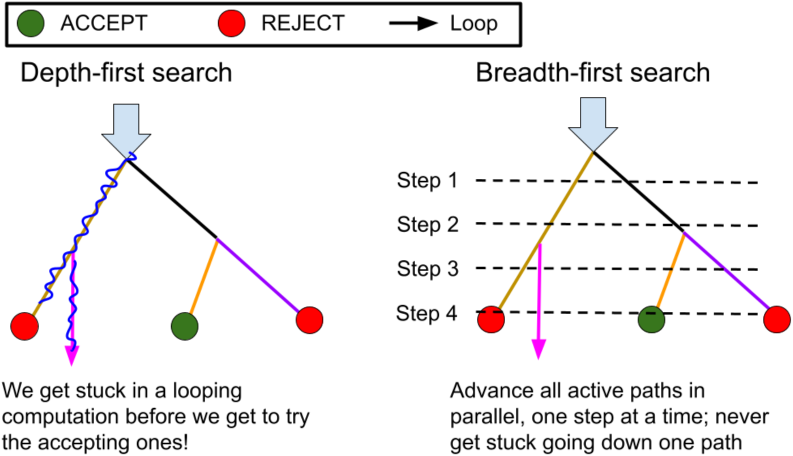


Breadth-first search

Technique: We run M, and test out all possible computation paths <u>in parallel</u>

- Keep track of all the current computation paths
- Run each current path for one step (rather than running any one path to completion)
- "breadth-first search"

Breadth-first search

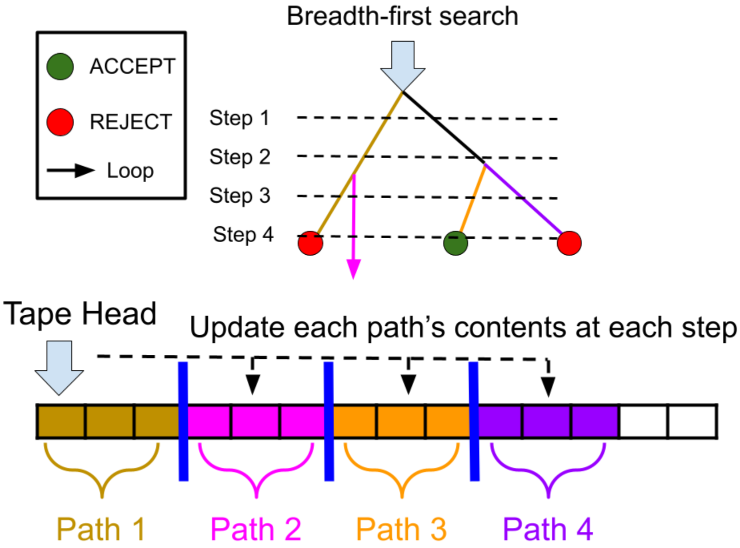


Non-deterministic Turing Machine

($\Leftarrow$) If L is recognized by a non-deterministic TM, it can be recognized by a deterministic TM

- Let M be the non-deterministic TM that recognizes L
- Design a deterministic TM called M_2 to simulate M
- M_2 tries out all possible computation paths of M <u>in parallel</u>
- If any computation path accepts ever, M_2 accepts, otherwise it rejects

Non-deterministic Turing Machine



Enumerator

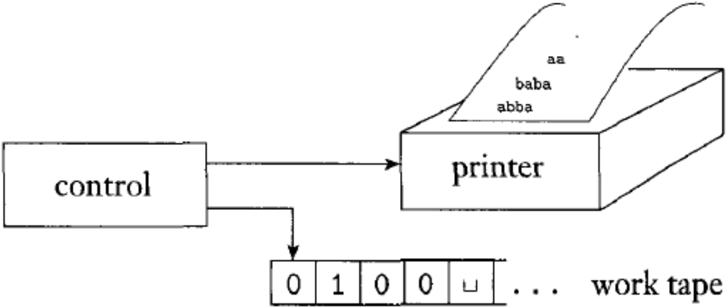
Def: An **enumerator** is a Turing machine with an attached printer

- At any point in time the TM may ask the printer to print a string

Enumerator

Def: An **enumerator** is a Turing machine with an attached printer

- At any point in time the TM may ask the printer to print a string



Recursively Enumerable Languages

- Let L be a language, and let E be an enumerator
- We say E **enumerates** L if E prints out every string in L
- If we give the enumerator infinite time, it will <u>eventually</u> print out every string in the language
- **Def:** If L can be enumerated, we say L is **recursively enumerable (RE)**

Recursively Enumerable Languages

Proposition: The following language is recursively enumerable

$$L = \{p | p \in \mathbb{N}, p \text{ is prime}\}$$

1. For $i = 0, 1, 2, \ldots$
   1.1 Check if i is prime
   1.2 If i is prime, print it out

Recursively Enumerable Languages

Theorem: A language L is Turing-recognizable if and only if L is recursively enumerable.

There are two directions:

1. If L is Turing-recognizable, there is an enumerator E that enumerates L
2. If L is RE, then some machine M can recognize L

Recursively Enumerable Languages

($\Rightarrow$) If L is recursively enumerable, L is Turing-recognizable

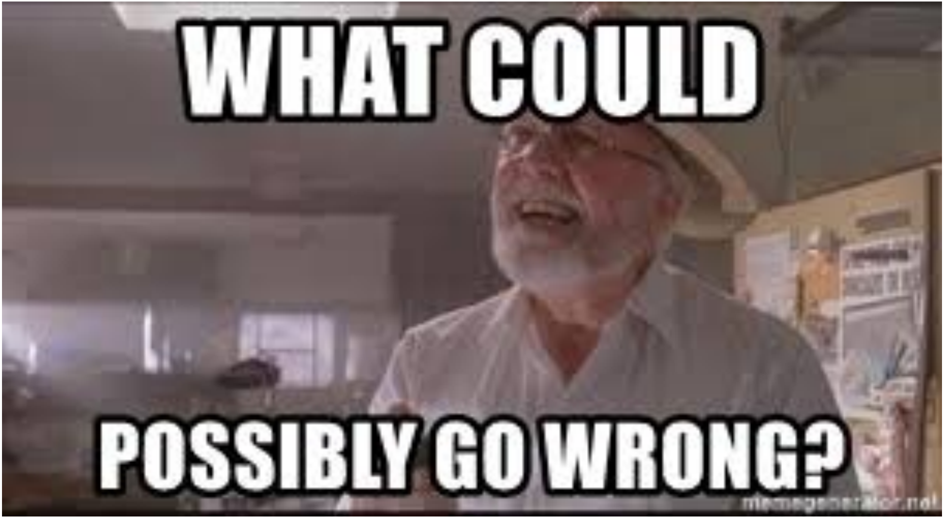
- We know some machine E enumerates L
- Design a machine M to recognize L
- On input w, do the following:
  1. Run E to enumerate L
  2. If E ever prints out w, then $w \in L$ so immediately accept
  3. If E never prints out w, then $w \notin L$ and M will run forever (which is OK)

Recursively Enumerable Languages

($\Leftarrow$) If L is Turing-recognizable, then L is recursively enumerable

- We know some machine M recognizes L
  - If $w \in L$, M accepts w
  - If $w \notin L$ then M rejects or loops
- We design a machine E to enumerate L
  1. Go through all $w \in \Sigma^*$ one at a time and run M on each w
  2. If M accepts w, print out w.
  3. After processing w, move on the the next string

Recursively Enumerable Languages



Recursively Enumerable Languages

($\Leftarrow$) If L is Turing-recognizable, then L is recursively enumerable

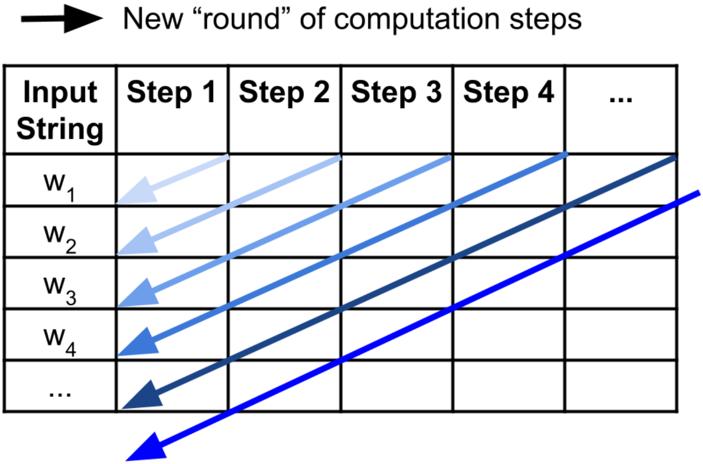
- We know some machine M recognizes L
  - If $w \in L$, M accepts w
  - If $w \notin L$ then M rejects or loops
- We design a machine E to enumerate L
  1. Go through all $w \in \Sigma^*$ one at a time and run M on each w
  2. ~~If M accepts w, print out w.~~
     This may run forever!
  3. After processing w, move on the the next string

Dovetailing

Technique: Run a machine M in parallel on all possible strings through dovetailing.

- Let $w_1, w_2, \cdots \in \Sigma^*$
- Let $S(i, j)$ represent step i in M's computation on string w_j
  1. Run $S(1, 1)$
  2. Run $S(1, 2), S(2, 1)$
  3. Run $S(1, 3), S(2, 2), S(3, 1)$
  4. Run $S(1, 4), S(2, 3), S(3, 2), S(4, 1)$
  5. ...

Dovetailing



Recursively Enumerable Languages

($\Leftarrow$) If L is Turing-recognizable, then L is recursively enumerable

- We know some machine M recognizes L
  - If $w \in L$, M accepts w
  - If $w \notin L$ then M rejects or loops
- We design a machine E to enumerate L
  1. Run M in parallel on all strings $w \in \Sigma^*$ (dovetailing)
  2. Whenever M accepts a string w, print out w (but keep running the other strings)