Nondeterministic TMs

Friday, April 5, 2024
Announcements

• HW 7 out
  • due Mon 4/8 12pm noon EST
**Last Time:** Turing Machines

- **Turing Machines** can **read and write** to arbitrary “tape” cells.
  - Tape initially contains input string.

- The tape is infinite.
  - (to the right)

- On a transition, “head” can move left or right **1 step**.

---

Call a language **Turing-recognizable** if some Turing machine recognizes it.
Turing Machine: High-Level Description

- $M_1$ accepts if input is in language $B = \{w#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{“On input string } w:\$

1. Zig-zag across the board while avoiding positions on either side of the # symbol. Use the same symbols on either side of the # symbol to keep track of which symbols correspond.
2. When all symbols to the right of the # symbol are crossed off, check for any remaining symbols. If any symbols remain, reject; otherwise accept.

We will (mostly) define TMs using high-level descriptions, like this one. (But it must always correspond to some formal low-level tuple description)

Analogy: High-level (e.g., Python) function definitions vs Low-level assembly language
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the \textit{blank symbol} \(\square\),
3. \(\Gamma\) is the tape alphabet, where \(\square \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
Flashback: DFAs vs NFAs

A finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \(Q\) is a finite set called the states,
2. \(\Sigma\) is a finite set called the alphabet,
3. \(\delta: Q \times \Sigma \rightarrow Q\) is the transition function,
4. \(q_0 \in Q\) is the start state, and
5. \(F \subseteq Q\) is the set of accept states.

A nondeterministic finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \(Q\) is a finite set of states,
2. \(\Sigma\) is a finite alphabet,
3. \(\delta: Q \times \Sigma_e \rightarrow \mathcal{P}(Q)\) is the transition function,
4. \(q_0 \in Q\) is the start state, and
5. \(F \subseteq Q\) is the set of accept states.

Nondeterministic transition produces set of possible next states.
A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where $Q$, $\Sigma$, $\Gamma$ are all finite sets and

1. $Q$ is the set of states,
2. $\Sigma$ is the input alphabet not containing the **blank symbol** $\_\_,$
3. $\Gamma$ is the tape alphabet, where $\_\_ \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$. 

**Remember:** Turing Machine Formal Definition
A **nondeterministic Turing Machine** is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q\), \(\Sigma\), and \(\Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the **blank symbol** \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) \(\rightarrow \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})\),
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
**Thm:** Deterministic TM $\Leftrightarrow$ Non-det. TM

⇒ If a **deterministic TM** recognizes a language, then a **non-deterministic TM** recognizes the language

- **Convert:** Deterministic TM $\rightarrow$ Non-deterministic TM ...
- ... change Deterministic TM $\delta$ output to: one-element set
  - $\delta_{ntm}(q, a) = \{\delta_{dtm}(q, a)\}$
  - (just like conversion of DFA to NFA --- HW 3, Problem 1)
- **DONE!**

⇐ If a **non-deterministic TM** recognizes a language, then a **deterministic TM** recognizes the language

- **Convert:** Non-deterministic TM $\rightarrow$ Deterministic TM ...
- ... ???
Review: Nondeterminism

Deterministic computation

- start
- ...
- accept or reject

Nondeterministic computation

- Each $\bullet$ = a state (for NFA)
- every step can branch to set of states

What is a “state” for a TM?

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

- A configuration (or ID) is a “snapshot” of a PDA’s computation

- 3 components \((q, w, \gamma)\):
  - \(q\) = the current state
  - \(w\) = the remaining input string
  - \(\gamma\) = the stack contents

A sequence of configurations represents a PDA computation
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
TM Configuration = State + Head + Tape

States

Starting configuration

Config after 1 step

Config after 2 steps

accept
TM Configuration = State + Head + Tape
TM Computation, Formally

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}) \]

**Single-step**

(Right)

\[ \alpha q_1 a \beta \vdash \alpha x q_2 \beta \]

if \( q_1, q_2 \in Q \)
\[
\delta(q_1, a) = (q_2, x, R)
\]
\( a, x \in \Gamma \)
\( \alpha, \beta \in \Gamma^* \)

(Left)

\[ \alpha b q_1 a \beta \vdash \alpha q_2 b x \beta \]

if \( \delta(q_1, a) = (q_2, x, L) \)

**Multi-step**

- **Base Case**

  \( I \vdash^* I \) for any ID \( I \)

- **Recursive Case**

  \( I \vdash^* J \) if there exists some ID \( K \) such that \( I \vdash K \) and \( K \vdash^* J \)

**Edge cases:**

- Head stays at leftmost cell
  \[ q_1 a \beta \vdash q_2 x \beta \]
  if \( \delta(q_1, a) = (q_2, x, L) \)

- Add blank symbol to config
  \[ \alpha q_1 \vdash \alpha \_ q_2 \]
  if \( \delta(q_1, \_ ) = (q_2, \_ R) \)

- (L move, when already at leftmost cell)

- (R move, when at rightmost filled cell)
Nondeterminism in TMs

Deterministic computation

- start
- ... accept or reject

Nondeterministic computation

- 1011q_00111
- ... reject
- 1011q_00111
- ... accept

For TMs, each node is a configuration
Nondeterministic TM $\rightarrow$ Deterministic  

1st way

- Simulate NTM with Det. TM:
  - Det. TM keeps multiple configs on single tape
    - Like how single-tape TM simulates multi-tape

- Then run all computations, concurrently
  - i.e., 1 step on one config, 1 step on the next, ...

- Accept if any accepting config is found

- Important:
  - Why must we step configs concurrently?
    Because any one path can go on forever!
Interlude: Running TMs inside other TMs

Remember analogy: TMs are like function definitions, they can be "called" like functions ...

Exercise:
• Given: TMs $M_1$ and $M_2$
• Create: TM $M$ that accepts if either $M_1$ or $M_2$ accept

Possible solution #1:
$M = \text{on input } x$,
1. Call $M_1$ with arg $x$; accept $x$ if $M_1$ accepts
2. Call $M_2$ with arg $x$; accept $x$ if $M_2$ accepts

Possible Results for $M$

<table>
<thead>
<tr>
<th></th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M$ Expected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>accept</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>reject</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>loops</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>loops</td>
<td>loops</td>
</tr>
</tbody>
</table>

Note: This solution would be ok if we knew $M_1$ and $M_2$ were deciders (which halt on all inputs)

“loop” means input string not accepted (but it should be)
Interlude: Running TMs inside other TMs

Just an analogy: “calling” a TM actually requires “computing” how it computes ...

Exercise:
• Given: TMs $M_1$ and $M_2$
• Create: TM $M$ that accepts if either $M_1$ or $M_2$ accept

Possible solution #1:
$M = \text{on input } x$,
1. Call $M_1$ with arg $x$; accept $x$ if $M_1$ accepts
2. Call $M_2$ with arg $x$; accept $x$ if $M_2$ accepts

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>loops</td>
<td>accept</td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>loops</td>
</tr>
</tbody>
</table>

Possible solution #2:
$M = \text{on input } x$,
1. Call $M_1$ and $M_2$, each with $x$, concurrently, i.e.,
   a) Run $M_1$ with $x$ for 1 step; accept $x$ if $M_1$ accepts
   b) Run $M_2$ with $x$ for 1 step; accept $x$ if $M_2$ accepts
   c) Repeat

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M$</th>
<th>$M$ Expected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>accept</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>reject</td>
<td>accept</td>
<td>✓</td>
</tr>
<tr>
<td>accept</td>
<td>loops</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>accept</td>
<td>✓</td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>accept</td>
<td>✓</td>
</tr>
</tbody>
</table>
Nondeterministic TM $\Rightarrow$ Deterministic

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
Nondeterministic TM $\rightarrow$ Deterministic

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2

2nd way (Sipser)
Nondeterministic TM $\Rightarrow$ Deterministic

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2
    - 1-1-1

2nd way
(Sipser)
Nondeterministic TM \( \rightarrow \) Deterministic

Always has input, never changes

“Work tape” when checking each path (re-copy input here each time)

Tracks which node we are on, e.g., 1-1-2, etc.

Use 3 tapes

2\textsuperscript{nd} way (Sipser)
Nondeterministic TM $\iff$ Deterministic TM

[Check] $\Rightarrow$ If a deterministic TM recognizes a language, then a nondeterministic TM recognizes the language
  • Convert Deterministic TM $\rightarrow$ Non-deterministic TM

[Check] $\Leftarrow$ If a nondeterministic TM recognizes a language, then a deterministic TM recognizes the language
  • Convert Nondeterministic TM $\rightarrow$ Deterministic TM
Conclusion: These are All Equivalent TMs!

• Single-tape Turing Machine

• Multi-tape Turing Machine

• Non-deterministic Turing Machine
Interlude: Running TMs inside other TMs

Just an analogy: “calling” a TM actually requires “computing” how it computes ...

Exercise:
• Given: TMs $M_1$ and $M_2$
• Create: TM $M$ that accepts if either $M_1$ or $M_2$ accept

Possible solution #1:
$M = \text{on input } x,$
1. Call $M_1$ with arg $x$; accept $x$ if $M_1$ accepts
2. Call $M_2$ with arg $x$; accept $x$ if $M_2$ accepts

Possible solution #2:
$M = \text{on input } x,$
1. Call $M_1$ and $M_2$, each with $x$, concurrently, i.e.,
   a) Run $M_1$ with $x$ for 1 step; accept $x$ if $M_1$ accepts
   b) Run $M_2$ with $x$ for 1 step; accept $x$ if $M_2$ accepts
   c) Repeat

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>loops</td>
<td>accept</td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>loops</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>accept</td>
<td>loops</td>
<td>accept</td>
</tr>
<tr>
<td>loops</td>
<td>accept</td>
<td>accept</td>
</tr>
</tbody>
</table>
**Flashback: HW 1, Problem 1**

**Figuring out this HW problem (about a DFA’s computation) ... is itself (meta) computation!**

What “kind” of computation is it?

Could you write a program (function) to compute it?

A function: `DFAaccepts(B, w)` returns `TRUE` if DFA B accepts string `w`

1) Define “current” state `q_{current} = start state q_0`
2) For each input char `a_i` ... in `w`
   a) Define `q_{next} = \delta_B(q_{current}, a_i)`
   b) Set `q_{current} = q_{next}`
3) Return `TRUE` if `q_{current}` is an accept state (of B)

You had to “compute” how a DFA computes

This is “computing” the accepting computation `\hat{\delta}(q_0, w) \in F!!`
The language of **DFAaccepts**

\[ A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \} \]

**How is this language a set of strings??**

**A function: DFAaccepts(B, w) returns true if DFA B accepts string w**
Interlude: Encoding Things into Strings

Definition: A language’s elements / (Turing) machine’s input is always a string

Problem: We sometimes want TM’s (program’s) input to be “something else” ... 
  • set, graph, DFA, ...?

Solution: allow encoding “other kinds of input” as a string

Notation: \(<SOMETHING>\) = string encoding for SOMETHING
  • A tuple combines multiple encodings, e.g., \(<B, w>\) (from prev slide)

Example: Possible string encoding for a DFA?

Details don’t matter! (In this class) Just assume it’s possible

Or:
\((Q, \Sigma, \delta, q_0, F)\)
(written as string)
Interlude: High-Level TMs and Encodings

A high-level TM description:

1. Needs to say the **type** of its input
   - E.g., graph, DFA, etc.

2. Doesn’t need to say how input string is encoded

3. Assumes TM **knows how** to parse and extract parts of input
   - Definition of $M$ can refer to $B$’s $(Q, \Sigma, \delta, q_0, F)$

4. Assumes input is a **valid** encoding
   - Invalid encodings implicitly rejected
**DFAaccepts** as a TM recognizing $A_{DFA}$

$$A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$$

A function: $\text{DFAaccepts}(B, w)$ returns $\text{TRUE}$ if DFA $B$ accepts string $w$

1) Define “current” state $q_{current} = \text{start state } q_0$
2) For each input char $a_i$ ... in $w$
   a) Define $q_{next} = \delta(q_{current}, a_i)$
   b) Set $q_{current} = q_{next}$
3) Return $\text{TRUE}$ if $q_{current}$ is an accept state

Remember:
- TM ~ program (function)
- Creating TM ~ programming

**TM $M_{DFA}$**

“On input $\langle B, w \rangle$, where $B$ is a DFA and $w$ is a string:

1) Define “current” state $q_{current} = \text{start state } q_0$
2) For each input char $a_i$ ... in $w$
   a) Define $q_{next} = \delta(q_{current}, a_i)$
   b) Set $q_{current} = q_{next}$
3) **Accept** if $q_{current}$ is an accept state in $F$
The language of $\text{DFAaccepts}$

$$A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$$

- $A_{\text{DFA}}$ has a Turing machine
- But is that TM a decider or recognizer? • I.e., is it an algorithm?
- To show it’s an algo, need to prove: $A_{\text{DFA}}$ is a decidable language

What “kind” of computation is it?
How to prove that a language is decidable?
How to prove that a language is decidable?

**Statements**
1. If a **decider** decides a lang $L$, then $L$ is a **decidable** lang
2. Define **decider** $M = \text{On input } w \ldots$, \textbf{Key step} $M$ decides $L$
3. $L$ is a **decidable** language

**Justifications**
1. Definition of **decidable** langs
2. See $M$ def, and Examples Table
3. By statements #1 and #2
How to Design Deciders

• **A Decider is a TM ...**
  • See previous slides on how to:
    • write a high-level TM description
    • Express encoded input strings
    • E.g., $M = \text{On input } <B, w>$, where $B$ is a DFA and $w$ is a string: ...

• **A Decider is a TM ... that must always halt**
  • Can only accept or reject
  • Cannot go into an infinite loop

• So a **Decider** definition must include an extra termination argument:
  • Explains how every step in the TM halts
  • (Pay special attention to loops)

• Remember our analogy: **TMs ~ Programs ... so Creating a TM ~ Programming**
  • To design a TM, think of how to write a program (function) that does what you want
Next Time: $A_{DFA}$ is a decidable language

$A_{DFA} = \{ \langle B, w \rangle | \text{B is a DFA that accepts input string } w \}$

Decider for $A_{DFA}$: