UMB CS420
Nondeterministic TMs
Wednesday, March 29, 2023
Announcements

• HW 7 out
  • due Sun 4/2 11:59pm EST

Quiz Preview

• Which of the following are equivalent to a single-tape deterministic TM?
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 $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1$ = “On input string $w$:

1. Zig-zag across the input, counting positions on either side of the # symbol, marking the same symbol at both positions. Do not cross off symbols. Cross off symbols on the # side to keep track of which symbols correspond.

2. When all symbols to the right of the # have been crossed off, check for any remaining symbols to the left. 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 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 \textit{states},
2. \(\Sigma\) is a finite set called the \textit{alphabet},
3. \(\delta: Q \times \Sigma \rightarrow Q\) is the \textit{transition function},
4. \(q_0 \in Q\) is the \textit{start state}, and
5. \(F \subseteq Q\) is the \textit{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 \textit{start state}, and
5. \(F \subseteq Q\) is the \textit{set of accept states}.

Nondeterministic transition produces \textit{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 \(q_{\text{reject}} \neq q_{\text{accept}}\).
A non-deterministic 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. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\)
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 $\delta$ fn output to a one-element set
     • $\delta_{nmt}(q, a) = \{\delta_{dtm}(q, a)\}$
     • (just like d2n conversion of DFA to NFA --- HW 2, Problem 2)
   • 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

- ... 
- reject 

In nondeterministic computation, every step can branch into a 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 \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
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

0 1 1 0 0 0 # 0 1 1 0 0 0 □ ...

x 1 1 0 0 0 # 0 1 1 0 0 0 □ ...

x 1 1 0 0 0 # x 1 1 0 0 0 □ ...

x 1 1 0 0 0 # x 1 1 0 0 0 □ ...

x 1 0 0 0 # x 1 1 0 0 0 □ ...

accept

x x x x x x # x x x x x □ ...
TM Configuration = State + Head + Tape

Textual representation of “configuration” (use this in HW)

1st char after state is current head position
TM Computation, Formally

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

**Single-step**

(Left)

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

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

**Extended**

- Base Case
  \[ I \vdash^* I \text{ for any ID } I \]

- Recursive Case
  \[ I \vdash^* J \text{ if there exists some ID } K \]
  \[ \text{such that } I \vdash K \text{ and } K \vdash^* J \]

**Edge cases:**

Head stays at leftmost cell

Add blank symbol to config

(Left)

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

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

(L move, when already at leftmost cell)

(R move, when at rightmost filled cell)

(Left)

\[ \alpha q_1 \vdash \alpha \times q_2 \]

if \( \delta(q_1, \times) = (q_2, \times, R) \)
Nondeterminism in TMs

Deterministic computation

- start
- ... (multiple transitions)
- accept or reject

Nondeterministic computation

- reject
- accept

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

• 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: If TMs are like function definitions, then 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:
$M =$ 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

Note: This solution would be ok if we knew $M_1$ and $M_2$ were *deciders* (which halt on all inputs)
Interlude: Running TMs inside other TMs

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 if $M_1$ accepts
   b) Run $M_2$ with $x$ for 1 step; accept if $M_2$ accepts
   c) Repeat
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

2nd way
(Sipser)

Nondeterministic computation

reject

accept
Nondeterministic TM $\to$ 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
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
Nondeterministic TM $\rightarrow$ Deterministic

2nd way
(Sipser)

Use 3 tapes

$D$

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.

$0 \ 0 \ 1 \ 0 \ \square \ldots$ input tape

$x \ x \ # \ 0 \ 1 \ x \ \square \ldots$ simulation tape

$1 \ 2 \ 3 \ 3 \ 2 \ 3 \ 1 \ 2 \ 1 \ 1 \ 3 \ \square \ldots$ address tape
Nondeterministic TM $\iff$ Deterministic TM

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

$\Leftarrow$ 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
Turing Machines as Algorithms
Turing Machines and Algorithms

• **Turing Machines** can express any “computation”
  • i.e., a Turing Machine models (Python, Java) programs (functions)!

• 2 classes of Turing Machines
  • **Recognizers** may loop forever
  • **Deciders** always halt

• **Deciders = Algorithms**
  • i.e., an algorithm is any program that always halts

Remember: TMs = program (functions)
Flashback: HW 1, Problem 1

1. Come up with 2 strings that are accepted by the DFA. These strings are in the language recognized by the DFA.
2. Come up with 2 strings that are not accepted (rejected) by the DFA. These strings are not in the language recognized by the DFA.
3. Is the empty string, $\epsilon$, in the language of the DFA?
4. Come up with a formal description for this DFA.

Recall that a DFA's formal description is a tuple of five components, e.g. $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$.

You may assume that the alphabet contains only the symbols from the diagram.

5. Then for each of the following, say whether the computation represents an accepting computation or not (make sure to review the definition of an accepting computation). If the answer is no, explain why not:
   
   a. $\delta(q_0, a\$b)$
   b. $\delta(q_1, a\$b)$
   c. $\delta(q_0, abc)$
   d. $\delta(q_0, cd\$)$

To "figure out" this computation ... you had to "do" (meta) computations (e.g., in your head)

This represents computation by a DFA
**Flashback: DFA Computations**

Define the extended transition function: 

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow Q \]

**Base case:** \( \hat{\delta}(q, \epsilon) = q \)

**Recursive case:** 

\[ \hat{\delta}(q, a_1 w_{rest}) = \hat{\delta}(\delta(q, a_1), w_{rest}) \]

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 \( a_i \)...
   a) Define \( q_{next} = \delta(q_{current}, a_i) \)
   b) Set \( q_{current} = q_{next} \)
3) Return TRUE if \( q_{current} \) is an accept state

Could you implement this (meta) computation as an algorithm?

First char

Last chars

Single transition step

Remember:

**TMs = program** (functions)
The language of $\text{DFAaccepts}$

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

- But I thought a language is defined as a set of strings???
- A function: $\text{DFAaccepts}(B, w)$ returns $\text{TRUE}$ if DFA $B$ accepts string $w$
Interlude: Encoding Things into Strings

Definition: A Turing machine’s input is always a string

Problem: A TM’s (program’s) input could also be: list, graph, DFA, ...?

Solution: encode other kinds of TM input as a string

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

Example: Possible string encoding for a DFA?

But in this class, we don’t care about what the encoding is! (Just that there is one)

\((Q, \Sigma, \delta, q_0, F)\)

(written as string)
Interlude: High-Level TMs and Encodings

A high-level TM description:

1. Doesn’t need to describe exactly how input string is encoded
2. Assumes input is a “valid” encoding
   - Invalid encodings are implicitly rejected
The language of \textbf{DFAaccepts}

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

\textbf{DFAaccepts} is a Turing machine recognizing language \( A_{\text{DFA}} \) i.e.,

- its inputs strings look like \( \langle B, w \rangle \) where
  - \( B \) is a DFA description
  - \( w \) is any string
- \textbf{DFAaccepts} accepts string \( \langle B, w \rangle \) if
  - DFA \( B \) would end in accept state if run with input string \( w \)

But is \textbf{DFAaccepts} a \textbf{decider} or \textbf{recognizer}?

- i.e., is it an \textbf{algorithm}?
- To show it’s an algo, need to \textbf{prove}:

\( A_{\text{DFA}} \) is a decidable language
How to prove that a language is decidable?

- Create a Turing machine that **decides** that language!

**Remember:**

- A **decider** is Turing Machine that always halts
  - i.e., for any input, it either accepts or rejects it.
  - It must never go into an infinite loop
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** $M = \text{On input } w \ldots$, $M$ decides $L$

3. $L$ is a **decidable** language

**Justifications**
1. Definition of **decidable** langs

2. See examples

3. By statements #1 and #2
How to prove that a language is decidable?

- Create a Turing machine that **decides** that language!

**Remember:**
- A **decider** is Turing Machine that always halts
  - i.e., for any input, it either accepts or rejects it.
  - It must never go into an infinite loop

**Deciders must also include a termination argument:**
- Explains how every step in the TM halts
- (Pay special attention to loops)
Next Time: $A_{DFA}$ is a decidable language

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

Decider for $A_{DFA}$:
Check-in Quiz 3/29
On gradescope