Decidability

Monday, April 8, 2024

UMB CS 420

Turing-recognizable

decidable

can be recognized by a Turing Machine

context-free

regular

language that can be described by a context-free grammar

language that can be recognized by a finite automaton
Announcements

• HW 7 extended
  • Due Mon 4/8 12pm noon
  • Due Wed 4/10 12pm noon

• HW 8 out
  • Due Wed 4/17 12pm noon

• No class Mon 4/15 (Patriots Day)

Quiz Preview (after class)
• What are the required parts of a **decider** TM definition?
Previously: Turing Machines and Algorithms

• Turing Machines can express more “computation” (than other prev machines)
  • Analogy: a TM models a (Python, Java) program (function)

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

• Deciders = Algorithms
  • I.e., an algorithm is a program that (for any input) always halts
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 $q_{\text{current}} = \text{start state } q_0$
2) For each input char $a_i$ ... in w
   a) Define $q_{\text{next}} = \delta_B(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) Return TRUE if $q_{\text{current}}$ is an accept state (of $B$)

This is “computing”: whether we have accepting computation $\hat{\delta}(q_0, w) \in F$ !!
The language of \textit{DFAaccepts}

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

A function: \textit{DFAaccepts}(B, w) returns \textbf{true} if DFA B accepts string w

How is this language a set of strings???
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: \(<\text{SOMETHING}>\) = string encoding for SOMETHING
• A tuple combines multiple encodings, e.g., \(<B, w>\) (from prev slide)

Example: Possible string encoding for a DFA?

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 TM\s 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
   - **Assume 1:** input is a **valid** encoding
     - Invalid encodings implicitly rejected
   - **Assume 2:** TM **knows how** to parse and extract parts of input

\[ M = \text{“On input } \langle B, w \rangle, \text{ where } B \text{ is a DFA and } w \text{ is a string:} \]

\[ B = (Q, \Sigma, \delta, q_0, F) \]

Definition of TM $M$ can use:

Details don’t matter! (In this class) Just assume it’s possible
**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 **true** if DFA $B$ accepts string $w$

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

**Previously**

Remember:

- **TM ~ program (function)**
- Creating **TM ~ programming**

"On input $B$ is a DFA and $w$ is a string:"

Definition of TM $M$ can use: $B = (Q, \Sigma, \delta, q_0, F)$

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

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

- \( A_{\text{DFA}} \) has a Turing machine
- Is the 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
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** $M = \text{On input } w \ldots$, $M \text{ decides } L$

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

**Justifications**

1. Definition of **decidable** langs

2. See $M$ def, and Equiv. 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
Thm: $A_{DFA}$ is a decidable language

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

Decider for $A_{DFA}$:

- **M** = “On input $\langle B, w \rangle$, where $B$ is a DFA and $w$ is a string:
  1. Simulate $B$ on input $w$.
  2. If the simulation ends in an accept state, **accept**. If it ends in a nonaccepting state, **reject**.”

Where “Simulate” =
- Define “current” state $q_{current} = \text{start state } q_0$
- For each input char $x$ in $w$ ...
  - Define $q_{next} = \delta(q_{current}, x)$
  - Set $q_{current} = q_{next}$

Remember:
- TM ~ program
- Creating TM ~ programming

Decider input must match strings in the language!
**Thm:** $A_{\text{DFA}}$ is a decidable language

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

**Decider for $A_{\text{DFA}}$:**

1. Simulate $B$ on input $w$.
2. If the simulation ends in an accept state, \textit{accept}. If it ends in a nonaccepting state, \textit{reject}.

**NOTE:** A TM must declare “function” parameters and types ... (don’t forget it)

Undeclared parameters can’t be used! (This TM is now invalid because $B$, $w$ are undefined!)

... which can be used (properly!) in the TM description
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

$$M = \text{“On input } \langle B, w \rangle \text{, where } B \text{ is a DFA and } w \text{ is a string:}$$

1. Simulate $B$ on input $w$.
2. If the simulation ends in an accept state, accept. If it ends in a nonaccepting state, reject.”

Where “Simulate” =
- Define “current” state $q_{current} =$ start state $q_0$
- For each input char $x$ in $w$ ...
  - Define $q_{next} = \delta(q_{current}, x)$
  - Set $q_{current} = q_{next}$

**Termination Argument:** Step #1 always halts because the simulation input is always finite, so the loop has finite iterations and always halts

**Deciders must have a termination argument:**
Explains how every step in the TM halts (we typically only care about loops)
Thm: \( A_{DFA} \) is a decidable language

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

Decider for \( A_{DFA} \):

\[ M = \text{“On input } \langle B, w \rangle \text{, where } B \text{ is a DFA and } w \text{ is a string:} \]

1. Simulate \( B \) on input \( w \).
2. If the simulation ends in an accept state, \textit{accept}. If it ends in a nonaccepting state, \textit{reject}.”

Termination Argument: \textbf{Step #2 always halts because we are checking only the state of the result (there’s no loop)}

Deciders must have a \textit{termination argument}: Explains how every step in the TM halts (we typically only care about loops)
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

$$M = \text{“On input } \langle B, w \rangle, \text{ where } B \text{ is a DFA and } w \text{ is a string:}$$

1. Simulate $B$ on input $w$.
2. If the simulation ends in an accept state, accept. If it ends in a nonaccepting state, reject.”

(New for TMs) “called” machine column(s)  
“Actual” behavior  
“Expected” behavior

<table>
<thead>
<tr>
<th>Example Str</th>
<th>$B$ on input $w$?</th>
<th>$M$?</th>
<th>In $A_{DFA}$ lang?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle D_1, w_1 \rangle$</td>
<td>Accept</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>$\langle D_2, w_2 \rangle$</td>
<td>Reject</td>
<td>Reject</td>
<td>No</td>
</tr>
</tbody>
</table>

Columns must match!

A good set of examples needs some Yes’s and some No’s

This is what a “Equivalence Table” justification should look like!

(typically only needed when called machine could loop)
Thm: $A_{\text{NFA}}$ is a decidable language

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

Decider for $A_{\text{NFA}}$:
Flashback: NFA→DFA

Have: \( N = (Q, \Sigma, \delta, q_0, F) \)

Want to: construct a DFA \( M = (Q', \Sigma, \delta', q_0', F') \)

1. \( Q' = \mathcal{P}(Q) \).

2. For \( R \in Q' \) and \( a \in \Sigma \),
\[
\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)
\]

3. \( q_0' = \{q_0\} \)

4. \( F' = \{R \in Q' | R \) contains an accept state of \( N\} \)

This conversion is computation!

So it can be computed by a (decider?) Turing Machine.
Turing Machine **NFA→DFA**

**TM NFA→DFA** = On input <N>, where \( N \) is an NFA and \( N = (Q, \Sigma, \delta, q_0, F) \)

1. **Write to the tape:** \( \text{DFA } M = (Q', \Sigma, \delta', q_0', F') \)

   Where: \( Q' = \mathcal{P}(Q) \).

   For \( R \in Q' \) and \( a \in \Sigma \),
   \[
   \delta'(R, a) = \bigcup_{r \in R} \delta(r, a)
   \]

   \( q_0' = \{q_0\} \)

   \( F' = \{R \in Q' \mid R \text{ contains an accept state of } N\} \)

**New TM Variation!**
Doesn’t accept or reject, Just writes “output” to tape

Why is this guaranteed to halt?
Because a DFA description has only finite parts (finite states, finite transitions, etc)
**Thm:** $A_{\text{NFA}}$ is a decidable language

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

**Decider for $A_{\text{NFA}}$:**

1. Convert NFA $B$ to an equivalent DFA $C$, using the procedure $\text{NFA} \rightarrow \text{DFA}$.
2. Run TM $M$ on input $\langle C, w \rangle$. ($M$ is the $A_{\text{DFA}}$ decider from prev slide.)
3. If $M$ accepts, accept; otherwise, reject.

**Termination argument:** This is a decider (i.e., it always halts) because:
- Step 1 always halts bc there’s a finite number of states in an NFA
- Step 2 always halts because $M$ is a decider
How to Design Deciders, Part 2

Hint:

• Previous theorems are a “library” of reusable TMs
• When creating a TM, try to use this “library” to help you
  • Just like libraries are useful when programming!
• E.g., “Library” for DFAs:
  • NFA→DFA, RegExp→NFA
  • Union operation, intersect, star, decode, reverse
  • Deciders for: $A_{DFA}$, $A_{NFA}$, $A_{REX}$, ...
**Thm:** \( A_{\text{REX}} \) is a decidable language

\[ A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \} \]

**Decider:**

\[ P = \text{"On input } \langle R, w \rangle \text{, where } R \text{ is a regular expression and } w \text{ is a string:} \]

1. Convert regular expression \( R \) to an equivalent NFA \( A \) by using the procedure \( \text{RegExpr} \rightarrow \text{NFA} \)

\[ \text{... which can be used (properly!) in the TM description} \]

\[ \text{NOTE: A TM must declare "function" parameters and types ... (don't forget it)} \]

**Remember:**
- TMs ~ programs
- Creating TM ~ programming
- Previous theorems ~ library
Flashback: \textit{RegExpr} → NFA

... so guaranteed to always reach base case(s)

\[ R \text{ is a regular expression if } R \text{ is} \]

1. \( a \) for some \( a \) in the alphabet \( \Sigma \),
2. \( \varepsilon \),
3. \( \emptyset \),
4. \( (R_1 \cup R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions,
5. \( (R_1 \circ R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions,
6. \( (R_1^*) \), where \( R_1 \) is a regular expression.

Yes, because recursive call only happens on “smaller” regular expressions ...

Does this conversion always halt, and why?
Thm: $A_{\text{REX}}$ is a decidable language

$$A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \}$$

Decider:

$P =$ “On input $\langle R, w \rangle$, where $R$ is a regular expression and $w$ is a string:

1. Convert regular expression $R$ to an equivalent NFA $A$ by using the procedure $\text{RegExpr\rightarrow NFA}$
2. Run TM $N$ on input $\langle A, w \rangle$ (from prev slide)
3. If $N$ accepts, accept; if $N$ rejects, reject.”

Termination Argument: This is a decider because:
- Step 1: always halts because converting a reg expr to NFA is done recursively, where the reg expr gets smaller at each step, eventually reaching the base case
- Step 2: always halts because $N$ is a decider
Decidable Languages About DFAs

- $A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$
  - Decider TM: implements $B$ DFA’s extended $\delta$ fn

- $A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \}$
  - Decider TM: uses $\text{NFA}\rightarrow\text{DFA}$ algorithm + $A_{\text{DFA}}$ decider

- $A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \}$
  - Decider TM: uses $\text{RegExpr}\rightarrow\text{NFA}$ algorithm + $A_{\text{NFA}}$ decider

Remember:
- TMs $\sim$ programs
- Creating TM $\sim$ programming
- Previous theorems $\sim$ library
Flashback: Why Study Algorithms About Computing

To predict what programs will do (without running them!)

Not possible for all programs! But ...

RANSOMWARE ATTACK

???
Predicting What Some Programs Will Do ...

What if: look at simpler computation models ... like DFAs and regular languages!
Thm: $E_{\text{DFA}}$ is a decidable language

$E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$

$E_{\text{DFA}}$ is a language ... of DFA descriptions, i.e., $(Q, \Sigma, \delta, q_0, F)$ ...

... where the language of each DFA ... must be $\emptyset$, i.e., DFA accepts no strings

Is there a decider that accepts/rejects DFA descriptions ...

... by predicting something about the DFA's language (by analyzing its description)

Key question we are studying:

Compute (predict) something about the runtime computation of a program, by analyzing only its source code?

Analogy

DFA's description: a program's source code ::
DFA's language    : a program's runtime computation

Important: don’t confuse the different languages here!
**Thm:** $E_{\text{DFA}}$ is a decidable language

$$E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$$

**Decider:**

$T = \text{"On input } \langle A \rangle, \text{ where } A \text{ is a DFA:}$$

1. Mark the start state of $A$.
2. **Repeat** until no new states get marked:
   3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject.*

---

... this is a “reachability” algorithm ... 
... check if accept states are “reachable” from start state

---

**Note:** TM $T$ does not “run” the DFA!

---

... it computes something about the DFA’s language (runtime computation) by analyzing it’s description (source code)
Thm: $E_{\text{DFA}}$ is a decidable language

$E_{\text{DFA}} = \{\langle A, B \rangle \mid \text{A and B are DFAs and } L(A) = L(B)\}$

I.e., Can we compute whether two DFAs are “equivalent”?

Replacing “DFA” with “program” = A “holy grail” of computer science!
Thm: $EQ_{DFA}$ is a decidable language

$$EQ_{DFA} = \{ \langle A, B \rangle | A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$$

A Naïve Attempt (assume alphabet \{a\}): 

1. Simulate:
   - $A$ with input $a$, and
   - $B$ with input $a$
   - Reject if results are different, else ... 

2. Simulate:
   - $A$ with input $aa$, and
   - $B$ with input $aa$
   - Reject if results are different, else ... 

• ...

This might not terminate! (Hence it's not a decider)

Can we compute this without running the DFAs?
Thm: $EQ_{DFA}$ is a decidable language

$$EQ_{DFA} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$$

Trick: Use Symmetric Difference
Symmetric Difference

\[ L(C) = \left( L(A) \cap \overline{L(B)} \right) \cup \left( \overline{L(A)} \cap L(B) \right) \]

\[ L(C) = \emptyset \text{ iff } L(A) = L(B) \]
Thm: $EQ_{\text{DFA}}$ is a decidable language

$$EQ_{\text{DFA}} = \{ \langle A, B \rangle | A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$$

Construct **decider** using 2 parts:

1. **Symmetric Difference algo:** $L(C) = \left( L(A) \cap \overline{L(B)} \right) \cup \left( \overline{L(A)} \cap L(B) \right)$
   - Construct $C = \text{Union, intersection, negation of machines } A \text{ and } B$

2. **Decider $T$ (from “library”) for:** $E_{\text{DFA}} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$
   - Because $L(C') = \emptyset$ iff $L(A) = L(B)$

**NOTE:** This only works because: regular langs closed under negation, i.e., set complement, union and intersection
Thm: $EQ_{DFA}$ is a decidable language

$EQ_{DFA} = \{ \langle A, B \rangle \mid A$ and $B$ are DFAs and $L(A) = L(B) \}$

Construct decider using 2 parts:

1. Symmetric Difference algo: $L(C) = \left( L(A) \cap \overline{L(B)} \right) \cup \left( \overline{L(A)} \cap L(B) \right)$.
   - Construct $C =$ Union, intersection, negation of machines $A$ and $B$

2. Decider $T$ (from “library”) for: $E_{DFA} = \{ \langle A \rangle \mid A$ is a DFA and $L(A) = \emptyset \}$
   - Because $L(C) = \emptyset$ iff $L(A) = L(B)$

$F =$ “On input $\langle A, B \rangle$, where $A$ and $B$ are DFAs:

1. Construct DFA $C$ as described.
2. Run TM $T$ deciding $E_{DFA}$ on input $\langle C \rangle$.
3. If $T$ accepts, accept. If $T$ rejects, reject.”

Termination argument?
Predicting What Some Programs Will Do ...

SLAM is a project for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine.

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." Bill Gates, April 18, 2002. Keynote address at WinHec 2002

Overview of Static Driver Verifier Research Platform

Static Driver Verifier (SDV) is a compile-time static verification tool. Static Driver Verifier Research Platform (SDVRP) is an extension to SDV that allows:

- Support additional frameworks (or APIs) and write custom plugins.
- Experiment with the model checking step.

Model checking

From Wikipedia, the free encyclopedia

In computer science, model checking or property checking is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness). This is typically
Summary: Algorithms About Regular Langs

- \( A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \} \)
  - **Decider:** Simulates DFA by implementing extended \( \delta \) function

- \( A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \} \)
  - **Decider:** Uses \( \text{NFA} \rightarrow \text{DFA} \) decider + \( A_{\text{DFA}} \) decider

- \( A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \} \)
  - **Decider:** Uses \( \text{RegExpr} \rightarrow \text{NFA} \) decider + \( A_{\text{NFA}} \) decider

- \( E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \)
  - **Decider:** Reachability algorithm

- \( E_{\text{DFA}} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \} \)
  - **Decider:** Uses complement and intersection closure construction + \( E_{\text{DFA}} \) decider

Remember:
- TMs ~ programs
- Creating TM ~ programming
- Previous theorems ~ library
Next: Algorithms (Decider TMs) for CFLs?

• What can we predict about CFGs or PDAs?