Decidability

Monday, April 3, 2023
Announcements

• HW 7 extended
  • Due Sun 4/2 11:59pm
  • Due Tue 4/4 11:59pm

• HW 8 out Wed 4/5
  • Due Tue 4/11 11:59pm

Quiz Preview

• A decider TM definition requires specifying which of the following parts?
Last Time: Turing Machines and Algorithms

• Turing Machines can express any “computation”
  • i.e., a TM represents 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 always halts
Flashback: HW 1, Problem 1

1. Come up with 2 strings that are accepted by the DFA. These strings are said to be 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, \( \varepsilon \), in the language of the DFA?
4. Come up with a string that is not in the language of the DFA.

Recall that a DFA:
\[ M = (Q, \Sigma, \delta, q_0, F) \]

You may assume:
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. \( \hat{\delta}(q_0, a\beta) \)
   b. \( \hat{\delta}(q_1, a\beta) \)
   c. \( \hat{\delta}(q_0, abc) \)
   d. \( \hat{\delta}(q_0, cd\$) \)

Remember: TMs = program (functions)

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 do it?

A function: \( \text{DFAaccepts}(B, w) \)
returns \text{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 \text{TRUE} if \( q_{\text{current}} \) is an accept state

This is just checks for an accepting computation \( \hat{\delta}(q_0, w) \in F \)!!

You had to figure out a DFA’s computation
The language of \textbf{DFAaccepts}

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

\textbf{Is this language a set of strings???}

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

\textbf{The set of strings that a Turing Machine accepts is a language ...}
Interlude: Encoding Things into Strings

Definition: A 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 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?

It doesn’t matter! 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 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

3. Assumes TM **knows** how to parse and extract parts of input
   - Description 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 $A_{DFA}$

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

A function: DFAaccepts(B, w) returns 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 TRUE if $q_{current}$ is an accept state

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

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

$$B = (Q, \Sigma, \delta, q_0, F)$$

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
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 ($\text{DFAaccepts}$)
- 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
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

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

2. Define *decider* \( M = \text{On input } w \ldots \), \( M \text{ decides } L \)

3. \( L \) is a *decidable* language

2. See \( M \) def, and examples

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 $A_{DFA}$:**

Decider input must match strings in the language!

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

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

Remember:

TM ~ program
Creating TM ~ programming
**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 $A_{DFA}$:**

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

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

**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 \text{ on input } w. \\
2. If the simulation ends in an accept state, } \text{accept}. \text{ If it ends in a nonaccepting state, } \text{reject."}$$

Where “Simulate” =

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

**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: $A_{\text{DFA}}$ is a decidable language

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Example String</th>
<th>In the $A_{DFA}$ language?</th>
<th>Accepted by $M$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle B, w \rangle$ where $B$ accepts $w$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$\langle B, w \rangle$ where $B$ rejects $w$</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Columns #2 and #3 must match

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

This is what a “See Examples” justification should look like!
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. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$

This conversion is computation

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

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

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

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

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

For \(R \in Q'\) and \(a \in \Sigma\),

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

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

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

Why is this guaranteed to halt?

Because a DFA description has only finite parts (finite states, finite transitions, etc)
**Thm:** $A_{NFA}$ is a decidable language

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

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

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 $A_{DFA}$ 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 because 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 \( \text{RegExp} \rightarrow \text{NFA} \)

... which can be used (properly!) in the TM description

**Remember:**

- TMs ~ programs
- Creating TM ~ programming
- Previous theorems ~ library

NOTE: A TM must declare “function” parameters and types ... (don’t forget it)
Flashback: \texttt{RegExp} \rightarrow \text{NFA}

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

\( R \) is a \textit{regular expression} if \( R \) 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 ...
**Thm:** $A_{\text{REX}}$ is a decidable language

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

**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 $\text{RegExpr} \rightarrow \text{NFA}$
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 for DFAs (So Far)

- $A_{\text{DFA}} = \{ \langle B, w \rangle | B$ is a DFA that accepts input string $w \}$
  - Deciding TM implements extended DFA $\delta$

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

- $A_{\text{REX}} = \{ \langle R, w \rangle | R$ is a regular expression that generates string $w \}$
  - Deciding TM uses $\text{RegExpr} \rightarrow \text{NFA} + \text{NFA} \rightarrow \text{DFA} + \text{DFA decider}$
Flashback: Why Study Algorithms About Computing

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

```javascript
function check(n) { // check if the number n is a prime
  var factor; // if the checked number is not a prime,
  // this is its first factor
  var isPrime = false;
  // try to divide the checked number by all numbers till its square root
  for (c=2; c<=Math.sqrt(n); c++) {
    if (n % c == 0) // is n divisible by c ?
      { factor = c; break; }
  }
  return factor; // end of check function
}

function communicate() { // communicate with the user
  var n = parseInt(document.getElementById('n').value); // get the checked number
  var factor; // if the checked number is not prime,
  // this is its first factor
  var isPrime = false;
  // is it a valid input?
  if (!isNaN(n) && n > 0) {
    factor = check(n); // the checked factor should be a positive number!
    if (factor == 0) {
      alert("The checked factor is not a prime!");
    } else {
      alert("The checked factor is a prime!");
    }
  } else {
    alert("The checked number should be a positive number!");
  }
  // end of communicate function
}
```

Not possible in general! But ...

RANSOMWARE ATTACK

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

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

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

$E_{DFA}$ 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., the DFA accepts no strings

We determine what is in this language ...

... by computing something about the DFA's language (by analyzing its definition)

i.e., by predicting how the DFA will behave

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

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

**Decider:**

$T =$ “On input $\langle A \rangle$, where $A$ is a DFA:

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

... this is a “reachability” algorithm ...

... check if accept states are “reachable” from start state

**Note:** Machine does not “run” the DFA!

**Termination argument?**

If loop marks at least 1 state on each iteration, then it eventually terminates because there are finite states; else loop terminates

... it computes something about the DFA’s language (by analyzing its definition)
Thm: $EQ_{DFA}$ is a decidable language

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

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

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

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

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

1. Run $A$ with input $a$, and $B$ with input $a$
   - Reject if results are different, else ...
2. Run $A$ with input $aa$, and $B$ with input $aa$
   - Reject if results are different, else ...
3. Run $A$ with input $aaa$, and $B$ with input $aaa$
   - 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: $\text{EQ}_{\text{DFA}}$ is a decidable language

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

Trick: Use Symmetric Difference
Symmetric Difference

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

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

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

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 \( C \) = Union, intersection, negation of machines \( A \) and \( B \)

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

\[ F = \text{“On input } \langle A, B \rangle, \text{ where } A \text{ and } B \text{ are DFAs:} \]

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

NOTE: This only works because: negation, i.e., set complement, and intersection is closed for regular languages.
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

Static Driver Verifier Research Platform README

Overview of Static Driver Verifier Research Platform

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

- Support additional frameworks (or APIs) and write customers
- 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: Decidable DFA Langs (i.e., algorithms)

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

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

- \( A_{\text{REX}} = \{ \langle R, w \rangle | R \text{ is a regular expression that generates string } w \} \)

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

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

Remember:
TM ~ program
Creating TM ~ programming
Previous theorems ~ library
Next Time: Algorithms (Decider TM) for CFLs?

• What can we predict about CFGs or PDAs?
**Thm:** \(A_{CFG}\) is a decidable language

\[A_{CFG} = \{\langle G, w \rangle | \ G \ \text{is a CFG that generates string} \ w\}\]

- This is a very practically important problem ...
- ... equivalent to:
  - Is there an **algorithm** to parse a programming language with grammar \(G\)?

- A Decider for this problem could ... ?
  - Try every possible derivation of \(G\), and check if it’s equal to \(w\)?
  - But this might never halt
    - E.g., what if there is a rule like: \(S \rightarrow \emptyset S\) or \(S \rightarrow S\)
    - This TM would be a **recognizer but not a decider**

**Idea:** can the TM stop checking after some length?
- I.e., Is there upper bound on the number of derivation steps?
Check-in Quiz 4/3

On gradescope