Decidability

Wednesday, March 9, 2022
Announcements

• HW 6 due Sun 3/20 11:59pm
  • After Spring Break

• No class next week
Last Time: Turing Machines and Algorithms

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

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

• Deciders = Algorithms
  • i.e., an algorithm is any program that always halts
Flashback: HW 1, Problem 1

1. DFA Formal Description

1. Come up with a formal description for this DFA.

Recall that a DFA’s formal description has five components, e.g.,

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

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

2. Then do the following computations using extended transition function and say whether computation represents an accepting computation (some of these may be tricky so be careful here, you may want to review the definition of an accepting computation):

   a. \( \delta(q_0, \varepsilon) \)
   
   b. \( \delta(q_0, a) \)

You had to “do” (meta) computations (e.g., on paper, in your head), to compute the DFA’s computation!
**Flashback: DFA Computations**

Define the extended transition function:

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

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

**Recursive case:** \[ \hat{\delta}(q, a_1w_{\text{rest}}) = \hat{\delta}(\delta(q, a_1), w_{\text{rest}}) \]

**Remember:** TMs = programs

Could you implement this (meta) computation as a program?

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

- Define “current” state \( q_{\text{current}} \) = start state \( q_0 \)
- For each input char \( a_i \)...
  - Define \( q_{\text{next}} = \delta(q_{\text{current}}, a_i) \)
  - Set \( q_{\text{current}} = q_{\text{next}} \)
- Return **TRUE** if \( q_{\text{current}} \) 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 language is a set of strings
Interlude: Encoding Things into Strings

• A Turing machine’s input is always a string

• So anything we want to give to TM must be **encoded** as string

**Notation:** \(<\text{SOMETHING}>\) = string encoding for \text{SOMETHING}

• A tuple combines multiple encodings, e.g., \(<B, w>\) (from prev slide)
Interlude: Informal TMs and Encodings

An informal TM description:

1. Doesn’t need to describe exactly how input string is encoded
   • Think of it as implicit parsing: the TM parses the input but we don’t care how

2. Assumes input is a “valid” encoding
   • Invalid encodings are implicitly rejected
The language of DFAaccepts

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

- What kind of language is this?
- What kind of machine accepts this language?
- DFAaccepts:
  - Define “current” state $$q_{current} = \text{start state } q_0$$
  - For each input char $$a_i$$ ...
    - Define $$q_{next} = \delta(q_{current}, a_i)$$
    - Set $$q_{current} = q_{next}$$
    - Return TRUE if $$q_{current}$$ is an accept state

- DFAaccepts is a Turing machine
- But is it a decider or recognizer?
  - i.e., is it an algorithm?
- To show it’s an algo, need to prove:

$$A_{DFA}$$ 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 Design Deciders

• If TMs = Programs ...
  ... then Creating a TM = Programming

• E.g., if HW asks “Show that lang $L$ is decidable” ...
  • .. you must create a TM that decides $L$; to do this ...
  • ... think of how to write a (halting) program that does what you want

• Deciders must also include a termination argument:
  • Explains how every step in the TM halts
  • (Pay special attention to loops)
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 $A_{DFA}$:

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

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

Termination Argument: This is a decider (i.e., it always halts) because the input is always finite, so the loop has finite iterations and always halts

Remember:
- TMs = programs
- Creating TM = programming

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

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

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. \( F' = \{R \in Q' \mid R \text{ contains an accept state of } N\} \)
**Thm:** $A_{NFA}$ is a decidable language

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

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

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

1. Convert NFA $B$ to an equivalent DFA $C$, using the procedure $\text{NFA} \to \text{DFA}$
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 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

• If TMs = Programs ...
  ... then Creating a TM = Programming

• E.g., if HW asks “Show that lang $L$ is decidable” ...
  • .. you must create a TM that decides $L$; to do this ...
  • ... think of how to write a (halting) program that does what you want

• Deciders must have a termination argument

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

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

**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 $\text{RegExpr} \rightarrow \text{NFA}$

**Remember:**

- TMs = programs
- Creating TM = programming
- Previous theorems = library
**Flashback:** \( \text{RegExp} \rightarrow \text{NFA} \)

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

R is a *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.

Does this conversion always halt, and why?

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 $\text{RegExp} \rightarrow \text{NFA}$
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
DFA TMs Recap (So Far)

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

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

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

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

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

function communicate() {
  var i = parseInt(document.getElementById("input").value); // get the checked number
  var factor; // if the checked number is not a prime, this is its first factor
  if (isNaN(i)) {
    alert("The checked input should be a single positive number");
    return;
  }
  factor = check(i);
  if (factor == 0) {
    alert(i + " is a prime");
  } else {
    alert(i + " is not a prime, " + n + " = " + factor + " * " + (n / factor));
  }
}
```

RANSOMWARE ATTACK

Not possible in general! But ...
Predicting What Some Programs Will Do ...

What if we look at weaker 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 $\{\}$, i.e., the DFA accepts no strings

We determine what is in this language ...

... by computing some property of a DFA’s language

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

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:**

\[ T = \text{“On input } \langle A \rangle, \text{ where } A \text{ is a DFA:} \]
\begin{enumerate}
    
    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.*
\end{enumerate}

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

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

Note: Machine does not “run” the DFA!
Thm: $EQ_{\text{DFA}}$ is a decidable language

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

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

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

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:** $EQ_{DFA}$ 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

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

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.”
Predicting What Some Programs Will Do ...

SLAM is a project for checking that software satisfies critical behavioral properties on 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. The Static Driver Verifier Research Platform (SDVRP) is an extension to SDV that allows:

- Support additional frameworks (or APIs) and write custom drivers.
- 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 done using model checking tools that generate proofs about the correctness of the system's behavior.

Its “language”
Summary: Decidable DFA Langs (i.e., algorithms)

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

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

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

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

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

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

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

$$A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \}$$

• This a is 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: $S \to 0S$ or $S \to S$
    • 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 3/9
On gradescope