UMB CS 420

Decidability

Thursday, November 11, 2022

Diagram:
- Turing-recognizable
- Decidable
- Context-free
- Regular

Trust in the accuracy of the diagram.
Announcements

• HW 7 out
  • due Mon 11/7 11:59pm
  • Note the extra day
Last Time: Turing Machines and Algorithms

- Turing Machines can express any “computation”
  - i.e., a Turing Machine 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

Remember: TMs = programs
Flashback: HW 1, Problem 1

1. Define current
2. Define q
3. For a i Define q next \(= \delta(q_{current}, a_i)\)
4. Set q current \(= q_{next}\)
5. Return TRUE if q current is an accept state

Remember: TMs = programs

Doing this HW about computation ... is itself (meta) computation!

A Turing Machine represents computation

Let’s do this with a Turing Machine:

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 \(a_i\)...
  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
The language of $\text{DFAaccepts}$

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

A Turing Machine represents computation a language ...

But a language is 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 any TM input as a string

Notation: \(<\text{SOMETHING}> = \text{string encoding for SOMETHING}\)
  - 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 say how input string is encoded
2. Assumes TM knows how to parse and extract parts of input
3. Assumes input is a valid encoding
   • Invalid encodings implicitly rejected
The language of **DFAaccepts**

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

- \( A_{\text{DFA}} \) \text{(DFAaccepts)} has 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:
  \( A_{\text{DFA}} \) 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 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 $A_{DFA}$:**

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

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:
- TMs = programs
- Creating TM = programming
**Thm:** $A_{DFA}$ is a decidable language

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

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

**Termination Argument:** This is a decider (i.e., it always halts) because the 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: $A_{NFA}$ is a decidable language

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

Decider for $A_{NFA}$:
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. \( F' = \{ R \in Q' | R \text{ contains an accept state of } N \} \)

This construction is computation

So it can be computed by a (decider) Turing Machine

Why is this guaranteed to halt?

(Could you implement this conversion algorithm as a program?)
**Thm:** $A_{NFA}$ is a decidable language

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

**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 $\text{NFA} \rightarrow \text{DFA}$
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 bc there’s a finite number of states in an NFA
- Step 2 always halts because $M$ is a decider

*Remember: TMs = programs Creating TM = programming Previous theorems = library*
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:
    • \( \text{NFA} \rightarrow \text{DFA}, \text{RegExp} \rightarrow \text{NFA} \)
    • Union operation, intersect, star, decode, reverse
    • Deciders for: \( A_{\text{DFA}}, A_{\text{NFA}}, A_{\text{REX}}, ... \)
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 \( \text{RegExpr}\rightarrow\text{NFA} \)

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

Remember:

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

\text{NOTE: A TM must declare "function" parameters ... (don't forget it)}
Flashback: \textbf{RegExpr\to 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.

Does this conversion always halt, and why?

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

\[ \rightarrow \]
**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 \( \text{RegExpr} \rightarrow \text{NFA} \)
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)

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

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

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

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

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

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

- $E_{\text{DFA}}$ is a language of DFA descriptions, i.e., $(Q, \Sigma, \delta, q_0, F)$ ...
- We determine what is in this language ...
- ... where the language of each DFA must be $\emptyset$, i.e., the DFA accepts no strings
- ... 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: $E_{\text{DFA}}$ 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:
   - 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.”

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

l.e., this is a “reachability” algorithm ...

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

Note: Machine does not “run” the DFA!

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

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

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 $a a$, and $B$ with input $a a$
   - Reject if results are different, else ...

3. Run $A$ with input $a a a$, and $B$ with input $a a a$
   - Reject if results are different, else ...

   - ...
Thm: $EQ_{DFA}$ is a decidable language

$EQ_{DFA} = \{ \langle A, B \rangle | A$ and $B$ 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: \( \text{EQ}_{\text{DFA}} \) is a decidable language

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

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:** \( \text{EDFA} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \)
   - 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 \( \text{EDFA} \) on input \( \langle C \rangle \).
3. If \( T \) accepts, accept. If \( T \) rejects, reject.”
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 platform. Research Platform (SDVRP) is an extension to SDV that allows

- Support additional frameworks (or APIs) and write custom
- 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)

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

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

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

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

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

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: $A_{CFG}$ is a decidable language

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

• 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: $S \rightarrow \emptyset S$ 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 11/3
On gradescope