Decidability

Monday, October 18, 2021
Announcements

• HW4 in

• HW5 out
  • Due Sun 10/24 11:59pm
Correctness of this Diagram?

HW4, Problem 5 proved that “regular” circle is correctly inside “context-free” circle.

In HW5, you’ll prove the rest.
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
Algorithms (Decidable Problems) for Regular Languages
**Flashback: Running a DFA “Program”**

Define the extended transition function: \( \hat{\delta} : Q \times \Sigma^* \rightarrow Q \)

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

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

Could you implement this as a program?

A function `DFAaccepts(B, w)` that returns `TRUE` if DFA `B` accepts string `w`

- Define “current” state \( q_{current} = \) 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
The language of $\text{DFAaccepts}$

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

But 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:** `<SOMETHING>` = string encoding for **SOMETHING**
  - A tuple combines multiple encodings, e.g., `<B, w>` (from prev slide)

**Example:** Possible string encoding for a DFA?
Interlude: Informal TMs and Encodings

An informal TM description:

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

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

- $\textbf{DFAaccepts}$ 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:
  \[ A_{\text{DFA}} \text{ is a decidable language} \]
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, either accepts or rejects it.
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
**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 \( q_{current} = \) 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 always terminates

**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: $A_{NFA}$ is a decidable language

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

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

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

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

- On input $\langle B, w \rangle$, where $B$ is an NFA and $w$ 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 $A_{DFA}$ 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

**Hint:**
• Previous theorems are a “library” of reusable TMs
• When creating a TM, try to use these theorems to help you
  • Just like you use libraries when programming!
• E.g., “Library” for DFAs:
  • $\text{NFA} \rightarrow \text{DFA}$, $\text{RegExp} \rightarrow \text{NFA}$,
  • union, 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:

$P =$ “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 $\text{RegExpr} \rightarrow \text{NFA}$
RegExpr→NFA

\( R \) 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. \( (R_1^\ast) \), where \( R_1 \) is a regular expression.

Does this conversion always halt?

Yes, because recursive call only happens on "smaller" reg exprs
**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 $\text{RegExpr} \to \text{NFA}$
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)

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

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

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

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: $E_{DFA}$ 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:
   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.”

Loop marks at least 1 state on each iteration, and there are finite states

Termination argument?

i.e., this is a “reachability” algorithm ...
... check if accept states are “reachable” from start state
Thm: $EQ_{DFA}$ is a decidable language

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

Trick: Use Symmetric Difference

I.e., Can we compute whether two (DFA) programs are "equivalent"?

(A “holy grail” of computer science)
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:** $EQ_{DFA}$ is a decidable language

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

Construct decider using 2 parts:

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

2. **Decider $T$ (from “library”) for:** $E_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$
   - 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.”

**NOTE:** This only works because: negation, i.e., set complement, and intersection is closed for regular languages.
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
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 research platform. SDV Research Platform (SDVRP) is an extension to SDV that allows:

- Support additional frameworks (or APIs) and write custom checkers.
- 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
Algorithms (Decidable Problems) for Context-Free Languages (CFLs)
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?
Chomsky Normal Form
Noam Chomsky

- He (sort of) invented this course too!
A context-free grammar is in **Chomsky normal form** if every rule is of the form

\[ A \rightarrow BC \]

\[ A \rightarrow a \]

where \( a \) is any terminal and \( A, B, \) and \( C \) are any variables—except that \( B \) and \( C \) may not be the start variable. In addition, we permit the rule \( S \rightarrow \varepsilon \), where \( S \) is the start variable.
Chomsky Normal Form: Number of Steps

To generate a string of length \( n \):

- \( n - 1 \) steps: to generate \( n \) variables
- \( + n \) steps: to turn each variable into a terminal

**Total:** \( 2n - 1 \) steps

(A finite number of steps)
Thm: Every CFG has a Chomsky Normal Form

1. Add new start variable $S_0$ that does not appear on any RHS
   - I.e., add rule $S_0 \to S$, where $S$ is old start var

\[
\begin{align*}
S & \to ASA \mid aB \\
A & \to B \mid S \\
B & \to b \mid \varepsilon \\
S_0 & \to S
\end{align*}
\]

\[
\begin{align*}
S & \to ASA \mid aB \\
A & \to B \mid S \\
B & \to b \mid \varepsilon
\end{align*}
\]
Thm: Every CFG has a Chomsky Normal Form

1. Add new start variable $S_0$ that does not appear on any RHS
   - I.e., add rule $S_0 \rightarrow S$, where $S$ is old start var

2. Remove all “empty” rules of the form $A \rightarrow \varepsilon$
   - $A$ must not be the start variable
   - Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     - E.g., if $R \rightarrow uAv$ is a rule, add $R \rightarrow uv$
     - Must cover all combinations if $A$ appears more than once in a RHS
       - E.g., if $R \rightarrow uAvAw$ is a rule, add 3 rules: $R \rightarrow uvAw$, $R \rightarrow uAvw$, $R \rightarrow uvw$
Thm: Every CFG has a Chomsky Normal Form

1. Add new start variable $S_0$ that does not appear on any RHS
   • I.e., add rule $S_0 \to S$, where $S$ is old start var

2. Remove all “empty” rules of the form $A \to \varepsilon$
   • $A$ must not be the start variable
   • Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     • E.g., if $R \to uAv$ is a rule, add $R \to uv$
     • Must cover all combinations if $A$ appears more than once in a RHS
       • E.g., if $R \to uAvAw$ is a rule, add $3$ rules: $R \to uvAw, R \to uAvw, R \to uvw$

3. Remove all “unit” rules of the form $A \to B$
   • Then, for every rule $B \to u$, add rule $A \to u$
Thm: Every CFG has a Chomsky Normal Form

1. Add new start variable $S_0$ that does not appear on any RHS
   • I.e., add rule $S_0 \rightarrow S$, where $S$ is old start var

2. Remove all “empty” rules of the form $A \rightarrow \varepsilon$
   • $A$ must not be the start variable
   • Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     • E.g., If $R \rightarrow uAv$ is a rule, add $R \rightarrow uv$
     • Must cover all combinations if $A$ appears more than once in a RHS
       • E.g., if $R \rightarrow uAvAw$ is a rule, add 3 rules: $R \rightarrow uvAw$, $R \rightarrow uAvw$, $R \rightarrow uvw$

3. Remove all “unit” rules of the form $A \rightarrow B$
   • Then, for every rule $B \rightarrow u$, add rule $A \rightarrow u$

4. Split up rules with RHS longer than length 2
   • E.g., $A \rightarrow wxyz$ becomes $A \rightarrow wB$, $B \rightarrow xC$, $C \rightarrow yz$

5. Replace all terminals on RHS with new rule
   • E.g., for above, add $W \rightarrow w$, $X \rightarrow x$, $Y \rightarrow y$, $Z \rightarrow z$
**Thm:** \( A_{\text{CFG}} \) is a decidable language

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

**Proof:** create the decider:

\( S = \) “On input \( \langle G, w \rangle \), where \( G \) is a CFG and \( w \) is a string:

1. Convert \( G \) to an equivalent grammar in Chomsky normal form.
2. List all derivations with \( 2n - 1 \) steps, where \( n \) is the length of \( w \); except if \( n = 0 \), then instead list all derivations with one step.
3. If any of these derivations generate \( w \), accept; if not, reject.”

**Termination argument?**
**Thm:** \( E_{\text{CFG}} \) is a decidable language.

\[
E_{\text{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \}
\]

Recall:

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

\( T = \) “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:
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.”

“Reachability” (of accept state from start state) algorithm
**Thm:** $E_{\text{CFG}}$ is a decidable language.

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

- Create decider that calculates reachability for grammar $G$
  - Except go backwards, start from **terminals**, to avoid looping

$R =$ “On input $\langle G \rangle$, where $G$ is a CFG:

1. Mark all terminal symbols in $G$.
2. Repeat until no new variables get marked:
3. Mark any variable $A$ where $G$ has a rule $A \rightarrow U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject.*”
Thm: $\text{EQ}_{\text{CFG}}$ is a decidable language?

$\text{EQ}_{\text{CFG}} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$

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

- Used Symmetric Difference

  \[ L(C) = \emptyset \text{ iff } L(A) = L(B) \]

  - where $C$ = complement, union, intersection of machines $A$ and $B$

- Can’t do this for CFLs!
  - Intersection and complement are not closed for CFLs!!!
Intersection of CFLs is **Not** Closed!

• If closed, then intersection of these CFLs should be a CFL:

\[
A = \{a^m b^n c^n \mid m, n \geq 0\}
\]

\[
B = \{a^n b^n c^m \mid m, n \geq 0\}
\]

• But \( A \cap B = \{a^n b^n c^n \mid n \geq 0\} \)

• Not a CFL!
Complement of a CFL is not Closed!

- If CFLs closed under complement:

  \[
  \text{if } G_1 \text{ and } G_2 \text{ context-free, then } L(G_1) \text{ and } L(G_2) \text{ context-free,}
  \]

  \[
  \text{L}(G_1) \cup \text{L}(G_1) \text{ context-free,}
  \]

  \[
  \text{L}(G_1) \cap \text{L}(G_2) \text{ context-free.}
  \]

  \*DeMorgan’s Law!*
Thm: $EQ_{\text{CFG}}$ is a decidable language?

$EQ_{\text{CFG}} = \{ (G, H) \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \}$

- No!
  - You cannot decide whether two grammars represent the same lang!

- It’s not recognizable either!
  - (We don’t know how to prove this yet)
Decidability of CFGs Recap

- \( A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates string } w \} \)
  - Convert grammar to Chomsky Normal Form
  - Then check all possible derivations of length \(2|w| - 1\) steps

- \( E_{\text{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \} \)
  - Compute “reachability” of start variable from terminals

- \( EQ_{\text{CFG}} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \} \)
  - We couldn’t prove that this is decidable!
  - (So you can’t use this theorem when creating another decider)
The Limits of Turing Machines?

• So TMs can express any “computation”
  • i.e., any (Python, Java, ...) program you write is a Turing Machine

• So why do we focus on TMs that process other machines?

• Because we also want to study the **limits** of computation
  • And a good way to test the limit of a computational model is to see what it can compute about other computational models ...

• So what are the limits of TMs? I.e., what’s here?
  • Or out here?
Next time: $A_{TM}$ is undecidable

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$
Check-in Quiz 10/18

On gradescope