UMB CS 420
Decidability for CFLs

November 8, 2022
Announcements

• HW 7 in
  • Due Mon Nov 7 11:59pm EST

• HW 8 out
  • Due Mon Nov 14 11:59pm EST
Last Time: Decider Turing Machines

• 2 classes of Turing Machines
  • Recognizers (all TMs): may loop forever
    • A TM that loops on an input does not accept that input
  • Deciders (subset of TMs) (algorithms) always halt
    • And then either accept or reject

• Decider definitions must include a termination argument:
  • Explains (informally) why every step in the TM halts
  • (Pay special attention to loops)
Last Time: Decidable Languages About DFAs

- \( A_{\text{DFA}} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts input string } w \} \)
  - Decider TM: implements \( B \) DFA's extended \( \delta \) fn

- \( A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is an NFA that accepts input string } w \} \)
  - Decider TM: uses NFA→DFA algorithm + \( A_{\text{DFA}} \) decider

- \( A_{\text{REX}} = \{ \langle R, w \rangle | R \text{ is a regular expression that generates string } w \} \)
  - Decider TM: uses RegExpr→NFA algorithm + \( A_{\text{NFA}} \) 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 for all programs! But...

RANSOMWARE ATTACK

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

What if: 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 $\emptyset$, 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)

The key question we are studying:
Can we compute something about the computation of a program, by analyzing only its source code?

Analogy
DFA’s description : a program’s source code ::
DFA’s language : what the program computes

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

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

**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.”

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

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

**Termination argument?**

Note: Machine does not “run” the DFA!

... it computes something about the DFA’s language (computation) by analyzing it’s description (source code)
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

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

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 simulating running the DFAs?
Thm: $EQ_{\text{DFA}}$ is a decidable language

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

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

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

1. Construct DFA \( C \) as described.
2. Run TM \( T \) deciding \( E_{\text{DFA}} \) 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
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 Research Platform README

Static Driver Verifier (SDV) is a compile-time static verification research platform. It 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: Algorithms About Regular Langs

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

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

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

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

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

---

Remember:

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

• What can we predict about CFGs or PDAs?
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:
  • Algorithm to parse “program” $w$ for PL 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 are rules like: $S \rightarrow \theta S$ 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 Example

- $S \rightarrow AS \mid AB$
- $A \rightarrow a$
- $B \rightarrow b$

To generate string of length: 2
- Use $S$ rule: 1 time; Use $A$ or $B$ rules: 2 times
- $S \Rightarrow AB \Rightarrow aB \Rightarrow ab$
- Derivation total steps: $1 + 2 = 3$

To generate string of length: 3
- Use $S$ rule: 2 times; $A$ or $B$ rules: 3 times
- $S \Rightarrow AS \Rightarrow AAB \Rightarrow aAB \Rightarrow aB \Rightarrow ab$
- Derivation total steps: $2 + 3 = 5$

To generate string of length: 4
- Use $S$ rule: 3 times; $A$ or $B$ rules: 4 times
- $S \Rightarrow AS \Rightarrow AAS \Rightarrow AAAB \Rightarrow aAAB \Rightarrow aaB \Rightarrow aaab$
- Derivation total steps: $3 + 4 = 7$

...
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!)

$A \rightarrow BC$ Use $n-1$ times
$A \rightarrow a$ Use $n$ times

Makes the string long enough
Convert string to terminals
Thm: $A_{CFG}$ is a decidable language

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

Proof: create the decider:

\[ S = \text{“On input } \langle G, w \rangle, \text{ where } G \text{ is a CFG and } w \text{ is a string:} \]

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.”

We first need to prove this is true for all CFGs!

Step 1: Conversion to Chomsky Normal Form is an algorithm ...
Step 2:
Step 3:

Termination argument?
### Thm: Every CFG has a Chomsky Normal Form

**Proof:** Create algorithm to convert any CFG into 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

\[
\begin{align*}
S & \rightarrow ASA | aB \\
A & \rightarrow B | S \\
B & \rightarrow b | \varepsilon \\
\end{align*}
\]

\[
\begin{align*}
S_0 & \rightarrow S \\
S & \rightarrow ASA | aB \\
A & \rightarrow B | S \\
B & \rightarrow b | \varepsilon \\
\end{align*}
\]
**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 $S_0 \rightarrow S$, where $S$ is old start var

2. Remove all “empty” rules of the form $A \rightarrow \epsilon$
   - $A$ 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: $R \rightarrow uvAw$, $R \rightarrow uAv$, $R \rightarrow uvw$

3. Remove all “unit” rules of the form $A \rightarrow B$
   - Then, for every rule $B \rightarrow u$, add rule $A \rightarrow u$
Termination argument of this algorithm?

**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 $A \rightarrow \epsilon$
   - $A$ 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:** $A_{CFG}$ is a decidable language

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

**Proof:** create the decider:

$S = \text{"On input } \langle G, w \rangle, \text{ where } G \text{ is a CFG and } w \text{ is a string:}\n1. \text{ Convert } G \text{ to an equivalent grammar in Chomsky normal form.}\n2. \text{ List all derivations with } 2n - 1 \text{ steps, where } n \text{ is the length of } w; \\text{ except if } n = 0, \text{ then instead list all derivations with one step.}\n3. \text{ If any of these derivations generate } w, \text{ accept; if not, reject."} \n
**Termination argument:**

**Step 1:** any CFG has only a finite # rules
**Step 2:** $2n - 1 = \text{finite # of derivations to check}$
**Step 3:** only 1 step

We first need to prove this is true for all CFGs!
**Thm:** $E_{CFG}$ is a decidable language.

Recall:

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

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

$T =$ “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

Can we compute “reachability” for a CFG?
Thm: $E_{CFG}$ is a decidable language.

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

Proof: create **decider** that calculates reachability for grammar $G$.

- Go backwards, start from **terminals**, to avoid getting stuck in looping rules.

$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.”

Loop marks 1 new variable on each iteration or stops: it eventually terminates because there are a finite # of variables.

Termination argument?
Thm: $EQ_{\text{CFG}}$ is a decidable language?

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

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

• Used Symmetric Difference

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

• where $C = \text{complement, union, intersection of machines } A \text{ and } B$
• Can’t do this for CFLs!
  • Intersection and complement are not closed for CFLs!!!
Intersection of CFLs is **Not** Closed!

**Proof** (by contradiction), *Assume* intersection is closed for CFLs.

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

- *... which is not a CFL!* (So we have a contradiction)
Complement of a CFL is not Closed!

- **Assume** CFLs closed under complement, then:
  
  \[
  \text{if } G_1 \text{ and } G_2 \text{ context-free, then:
  }
  \]

  \[
  \overline{L(G_1)} \text{ and } \overline{L(G_2)} \text{ context-free}
  \]

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

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

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

  \[
  \text{But intersection is not closed for CFLS (prev slide)}
  \]

  \[
  \text{From the assumption}
  \]

  \[
  \text{Union of CFLs is closed}
  \]

  \[
  \text{From the assumption}
  \]

  \[
  \text{DeMorgan’s Law!}
  \]
Thm: $\text{EQ}_{\text{CFG}}$ is a decidable language?

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

- No!
  - There’s no algorithm to decide whether two grammars are equivalent!

- It’s not recognizable either! (Can’t create any TM to do this!!!)
  - (details later)

- I.e., this is an impossible computation!
Summary Algorithms About CFLs

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

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

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

- TMs represent all possible “computations”
  - i.e., any (Python, Java, ...) program you write is a TM

- But some things are not computable? i.e., some langs are out here?

- To test the limit of a computational model, we can see...
  ... what it can compute about other computational models

  - Thought: Is there a decider (algorithm) to determine whether a TM is an decider?

  Hmm, this doesn’t feel right...
Next time: Is $A_{TM}$ decidable?

$$A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$$
Check-in Quiz 11/8

On gradescope