UMB CS 622
More Decidability

Friday, April 12, 2024

Halting TMs, a.k.a., “algorithms”

... that predict:
- Reg lang computation
- CFL computation
Announcements

• HW 8 out
  • Due Wed April 17 12pm noon

• No class Monday 4/15 – Patriot’s Day

In-class participation question
• Which of the following rules are valid for a grammar in Chomsky Normal Form?
Decidable Languages About DFAs

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

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

- $A_{REX} = \{ \langle R, w \rangle | R \text{ is a regular expression that generates string } w \}$
  - Decider TM: uses $\text{RegExp2NFA}$ algorithm + $A_{NFA}$ decider

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

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 we: look at **simpler** 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)

Key question we are studying:
Compute (predict) something about the runtime computation of a program, by analyzing only its source code?

Analogy
DFA’s description: a program’s source code ::
DFA’s language : a program’s runtime computation

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.”

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: TM $T$ is doing a different computation on DFAs! (It doesn’t “simulate” the DFA!)

Instead: compute something about DFA’s language (runtime computation) by analyzing its description (source code)
Thm: $EQ_{DFA}$ is a decidable language

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

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

 Replace “DFA” with “program” = A “holy grail” of computer science!
Thm: $E_{DFA}$ is a decidable language

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

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 ...
   - ...

(meta) compute how the DFA would compute

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

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

\[ L(C) = \emptyset \text{ iff } L(A) = L(B) \]
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 $C = \text{Union, intersection, negation of machines } A \text{ and } B$

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 $L(A) = L(B)$

**NOTE:** This only works because regular langs closed under negation, i.e., set complement, **union** and **intersection**
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:**
   $$L(C) = \left( L(A) \cap \overline{L(B)} \right) \cup \left( \overline{L(A)} \cap L(B) \right)$$
   
   - Construct $C$ = Union, intersection, negation of machines $A$ and $B$

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

$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 $E_{DFA}$ 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 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

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 $\text{NFA}\to\text{DFA}$ decider + $A_{DFA}$ decider

- $A_{REX} = \{ \langle R, w \rangle | \ R \text{ is a regular expression that generates string } w \}$
  - **Decider:** Uses $\text{RegExpr}\to\text{NFA}$ 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: $A_{\text{CFG}}$ is a decidable language

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

- This is a very practically important problem ...
- ... equivalent to:
  - Algorithm to parse “program” $w$ for a programming language with grammar $G$?

- A Decider for this problem could ... ?
  - Try: for every possible derivation of $G$, check if result is $w$?
  - But this might never halt
    - E.g., what if there are rules like: $S \to \emptyset S$ 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?
Chomsky Normal Form
Noam Chomsky

He came up with this hierarchy of languages
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

- \( S \rightarrow AB \)
- \( A \rightarrow AB \)
- \( A \rightarrow a \)
- \( B \rightarrow b \)

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

To generate string of length: 3
- Use \( S \) rule: 1 time; \( A \) rule: 1 time; \( A \) or \( B \) rules: 3 times
- \( S \Rightarrow AB \Rightarrow AAB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab \)
- Derivation total steps: \( 1 + 1 + 3 = 5 \)

To generate string of length: 4
- Use \( S \) rule: 1 time; \( A \) rule: 2 times; \( A \) or \( B \) rules: 4 times
- \( S \Rightarrow AB \Rightarrow AAB \Rightarrow AAAB \Rightarrow aAAB \Rightarrow aaAB \Rightarrow aab \Rightarrow aaab \)
- Derivation total steps: \( 3 + 4 = 7 \)

\[ \cdots \]
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!)

Makes the string long enough
Convert string to terminals

Chomsky normal form

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

$A \rightarrow a$
Use $n$ times
**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.”

---

**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

$$
S \rightarrow ASA \mid aB \\
A \rightarrow B \mid S \\
B \rightarrow b \mid \varepsilon
$$

\[
S_0 \rightarrow S
\]

\[
S \rightarrow ASA \mid aB \\
A \rightarrow B \mid S \\
B \rightarrow b \mid \varepsilon
\]
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: $R \to uvAw$, $R \to uAvw$, $R \to uvw$

\[
\begin{align*}
S_0 & \to S \\
S & \to ASA \mid aB \mid a \\
A & \to B \mid S \mid \varepsilon \\
B & \to b \mid \varepsilon \\
\end{align*}
\Rightarrow
\begin{align*}
S_0 & \to S \\
S & \to ASA \mid aB \mid a \mid SA \mid AS \mid S \\
A & \to B \mid S \mid \varepsilon \\
B & \to b \\
\end{align*}
\]
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: $R \rightarrow uvAw, R \rightarrow uAvw, R \rightarrow uvw$
3. Remove all “unit” rules of the form $A \rightarrow B$
   • Then, for every rule $B \rightarrow u$, add rule $A \rightarrow u$
**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: $A_{CFG}$ 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.”

Termination argument:
Step 1: any CFG has only a finite # rules
Step 2: $2n - 1 =$ finite # of derivations to check
Step 3: checking finite number of derivations

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

Recall:

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

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

$T =$ “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

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

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

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.”

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

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

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

- Used Symmetric Difference

- 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!

**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:

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

- From the assumption
- Union of CFLs is closed
- From the assumption
- DeMorgan’s Law!

But intersection is not closed for CFLs (prev slide)
Thm: $EQ_{\text{CFG}}$ is a decidable language?

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

• 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

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

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

- \( EQ_{CFG} = \{ \langle G, H \rangle | G \text{ and } H \text{ are CFGs and } L(G) = L(H) \} \)
  - 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 explore the limits of computation, we have been studying ...
  ... computation about other computation ...
  • Thought: Is there a decider (algorithm) to
determine whether a TM is an decider?

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

$$A_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$$