Regular Expressions and Inductive Proofs

Wed Feb 17, 2021
Logistics

• HW2 solutions posted

• HW3 due Sunday 2/21 11:59pm EST
  • Mostly a repeat of HW1-2 tasks, but for NFAs
  • Note: last question is non-coding

• Coding in this class:
  • Forces you to be precise
  • Reinforces that we are studying **computation**
    • and meta-computation!
    • Proof by construction = algorithm = computation by a more powerful computer!
    • (see next slide)
  • As computational models get complex, we will transition to on-paper proofs

• Questions?
Review: HW2, Intersection Problem

Password Requirements

- Passwords must have a minimum length of ten (10) characters - but more is better!
- Passwords **must include at least 3** different types of characters:
  - upper-case letters (A-Z)
  - lower-case letters (a-z)
  - symbols or special characters (%, &, *, $, etc.)
  - numbers (0-9)
- Passwords cannot contain all or part of your email address
- Passwords cannot be re-used

Combination of these machines is also a state machine.

But what kind of computer is needed to perform the combining?
def DFA_Intersection(DFA1, DFA2):

    DFA = {'states': set(), 'sigma': set(), 'delta': {}, 'start': '', 'accepts': set()}

    DFA['states'] = set(it.product(DFA1['states'], DFA2['states']))

    DFA['sigma'] = set.union(DFA1['sigma'], DFA2['sigma'])

    DFA['start'] = (DFA1['start'], DFA2['start'])

    DFA['accepts'] = set(it.product(DFA1['accepts'], DFA2['accepts']))

    for state in DFA['states']:
        DFA['delta'][state] = {}
        for string in DFA['sigma']:
            DFA['delta'][state][string] = (DFA1['delta'][state[0]][string], DFA2['delta'][state[1]][string])

    return DFA

M1_I_M2 = DFA_Intersection(M1, M2)  # M1 and M2 intersection
M3_I_M4 = DFA_Intersection(M3, M4)  # M3 and M4 intersection
DFA_Final = DFA_Intersection(M1_I_M2, M3_I_M4)  # Final DFA i.e. intersection of M1,M2,M3,M4

# String check condition.
if run(DFA_Final, string):
    sys.stdout.write("valid")
Flashback: Levels of Computational Power

Construction of password validation machine from smaller state machine???

Password validation

We’ll start here
Review: HW2, Intersection Problem: A different answer

Password checking **not** computed by state machine (no call to “run” function)

```python
def intersection(dfa1, dfa2, dfa3, dfa4, password):
    flag1 = 0
    flag2 = 0
    flag3 = 0
    flag4 = 0
    for char in password:
        if (char in dfa1.alphabet): flag1 = 1
        elif (char in dfa2.alphabet): flag2 = 1
        elif (char in dfa3.alphabet): flag3 = 1
    i = len(password)
    j = len(dfa4.alphabet)
    if (i >= j): flag4 = 1
    if (flag1 and flag2 and flag3 and flag4):
        print("valid")
    else:
        print("invalid")
```

Last time: Regular Expressions

- **Regular expressions** are widely used by programmers
  - But they can only match **regular languages**
  - So to *properly* use reg. exps, you must know what is/isn’t a regular lang!

---

RegEx match open tags except XHTML self-contained tags

- I need to match all of these opening tags:
  - `<p> <a href="foo">`
  - `<br /> <hr class="foo" />`

- But not these:
  - `<br />
  - `<hr class="foo" />

- You can’t parse [X]HTML with regex. Because HTML can’t be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions.
**Last time**: Big Picture Road Map

- In this course, we must formally prove the equivalence:
  - Regular Languages $\Leftrightarrow$ Regular Expressions [Today!]

- To do so, we need to prove these ops are closed under reg langs:
  - Union (done!)
  - Concatenation (done!)
  - Kleene star (done!)

- To prove closure properties, we using NFAs:
  - Need NFA $\Leftrightarrow$ DFA equivalence theorem (done!)
By the end of class today ...

• We’ll have proven that all these are equivalent:
  • Deterministic Finite Automaton (DFA)
  • Non-deterministic Finite Automaton (NFA)
  • Generalized Non-deterministic Finite Automaton (GNFA)
  • Regular Expressions

• They all represent a regular language!
Regular Expressions, Formal Definition

**Definition 1.52**

Say that $R$ is a *regular expression* if $R$ is

1. $a$ for some $a$ in the alphabet $\Sigma$, 
   - (A lang containing a) length-1 string
2. $\varepsilon$, 
   - (A lang containing) the empty string
3. $\emptyset$, 
   - The empty set (i.e., a lang containing no strings)
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, or 
6. $(R_1^*)$, where $R_1$ is a regular expression.
Regular Expression: Concrete Example

*Entire reg expr:* represents lang whose strings are strings from these langs concat’ed together (implicit concat op)

- the lang {"0","1"}
- \((0 \cup 1)0^*\)
- the lang {"\"","0","00", ...}
- the lang {"0"}
- the lang {"1"}

*Operator Precedence:*
- Parens
- Star
- Concat (sometimes implicit)
- Union
Thm: A lang is regular iff some reg expr describes it

• => If a language is regular, it is described by a reg expr

• <= If a language is described by a reg expr, it is regular
  • Easy!
  • For a given regexp, construct the equiv NFA!
  • See Lemma 1.55

How to show that a lang is regular?

Construct DFA or NFA!
Lemma 1.55: Regexp -> NFA

**Definition 1.52**

Say that $R$ 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.

Constructions from before!
Thm: A lang is regular iff some reg expr describes it

• => If a language is regular, it is described by a reg expr
  • Hard!
  • Need to convert DFA or NFA to Regular Expression
  • Need something new: a GNFA

• <= If a language is described by a reg expr, it is regular
  • Easy!
  • Construct the NFA! (Done)
Generalized NFAs (GNFAs)

- GNFA = NFA with regular expression transitions
GNFA->Regexp function

• On GNFA input $G$:
  • If $G$ has 2 states, return the regular expression transition, e.g.:

$$ (R_1) (R_2)^* (R_3) \cup (R_4) $$

• Else:
  • “Rip out” one state, and “repair”, to get $G'$ (has one less state than $G$)
  • Recursively call GNFA->Regexp($G'$)

A recursive (function) definition!
Recursive (Inductive) Definitions

• (at least) two parts:
  • Base case
  • Inductive case
    • Self-reference must be “smaller” than the whole

• Example: factorial function

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

What’s the base case?
Self-reference smaller than the whole
GNFA->Regexp function

• On GNFA input G:
  • If G has 2 states, return the regular expression transition, e.g.:
    
    ![Diagram showing the transition from qi to qj with expression (R1)(R2)* (R3) ∪ (R4)]

    **Base case**

    **Inductive case**
    • Else:
      • “Rip out” one state, and “repair”, to get G’ (has one less state than G)
      • **Recursively** call GNFA->Regexp(G’)

  **Equivalent Regular expression**

  **GNFA**

  **A recursive (function) definition!**
GNFA->Regexp function: “Rip/repair” step

To convert GNFA to reg expr: “rip out” states, and then “repair”, until only 2 states remain
GNFA->Regexp function: “Rip/repair” step

Before: two paths from $q_i$ to $q_j$:
1. Not through $q_{rip}$
2. Through $q_{rip}$

\[
(R_1) (R_2)^* (R_3) \cup (R_4)
\]

after
GNFA-\(\rightarrow\)Regexp function: “Rip/repair” step

**Before:**
- \(q_i\) to \(q_j\)
- \(q_{\text{rip}}\)
- \(R_1\), \(R_2\), \(R_3\), \(R_4\)

**After:**
- 
**Paths from \(q_i\) to \(q_j\):**
  1. Not through \(q_{\text{rip}}\)
  2. Through \(q_{\text{rip}}\)

**Expression:**
\[(R_1)(R_2)^* (R_3) \cup (R_4)\]
GNFA->Regexp function: “Rip/repair” step

Before:
- path through $q_{rip}$ has 3 transitions
- One is self loop
GNFA->Regexp function: “Rip/repair” step

Before:
- path through $q_{\text{rip}}$ has 3 transitions
- One is self loop

After:
- Self loop becomes star operation
- Others are concat’ed together

This course requires formal correctness, i.e., proof

This is “informal” correctness
Need to prove GNFA->Regexp “correct”

• Where “correct” means:

\[ \text{LANGOF} ( G ) = \text{LANGOF} ( \text{GNFA->Regexp} ( G ) ) \]

• i.e., GNFA->Regexp must not change the language!
Kinds of Mathematical Proof

• Proof by construction

• Proof by contradiction

• Proof by induction
  • Use to prove properties of recursive (inductive) defs or functions
Proof by Induction

• To prove that a property $P$ is true for a thing $x$
  • First, prove that $P$ is true for the base case of $x$ (usually easy)
  • Then, prove the induction step:
    • Assume the induction hypothesis (IH):
      • $P(x)$ is true, for some $x_{\text{smaller}}$ that smaller than $x$
      • and use it to prove $P(x)$
    • The key is $x_{\text{smaller}}$ must be smaller than $x$

• Why can we assume IH is true???
  • Because we can always start at base case,
  • Then use it to prove for slightly larger case,
  • Then use that to prove for slightly larger case ...
Need to prove GNFA->Regexp “correct”

• Where “correct” means:

$$\text{LANGOF}(G) = \text{LANGOF}(\text{GNFA->Regexp}(G))$$

• i.e., GNFA->Regexp must not change the language!
GNFA->Regexp is correct

**Def:** GNFA->Regexp: input G is a GNFA with n states:
- If n = 2: return the reg expr on the transition
- Else (G has n > 2 states):
  - “Rip” out one state to get G’
  - Recursively Call GNFA->Regexp(G’)

**Proof** (by induction on size of G):
GNFA->Regexp is correct

Def: GNFA->Regexp: input G is a GNFA with n states:
   If \( n = 2 \): return the reg expr on the transition
   Else (G has \( n > 2 \) states):
      “Rip” out one state to get G’
      Recursively Call GNFA->Regexp(G’)

• Proof (by induction on size of G):
  ➢ Base case: G has 2 states
    • \( \text{LANGOF} ( G ) = \text{LANGOF} ( \text{GNFA->Regexp} ( G ) ) \) is true!
GNFA->Regexp is correct

**Def:** GNFA->Regexp: input \( G \) is a GNFA with \( n \) states:
- If \( n = 2 \): return the reg expr on the transition
- Else (\( G \) has \( n > 2 \) states):
  - “Rip” out one state to get \( G' \)
  - Recursively Call GNFA->Regexp\((G')\)

**Proof** (by induction on size of \( G \)):
- **Base case:** \( G \) has 2 states
  - \( \text{LANGOF}(G) = \text{LANGOF}(\text{GNFA->Regexp}(G)) \) is true!
- **IH:** Assume \( \text{LANGOF}(G') = \text{LANGOF}(\text{GNFA->Regexp}(G')) \)
  - For some \( G' \) with \( n-1 \) states
**GNFA->Regexp is correct**

**Def:** GNFA->Regexp: input G is a GNFA with n states:
   If \( n = 2 \): return the reg expr on the transition
   Else (G has \( n > 2 \) states):
   “Rip” out one state to get \( G' \)
   Recursively Call GNFA->Regexp(\( G' \))

**Proof** (by induction on size of G):
- **Base case:** G has 2 states
  - \( \text{LANGOF}(G) = \text{LANGOF}(\text{GNFA->Regexp}(G)) \) is true!
- **IH:** Assume \( \text{LANGOF}(G') = \text{LANGOF}(\text{GNFA->Regexp}(G')) \)
  - For some \( G' \) with \( n-1 \) states
- **Induction Step:** Prove it’s true for G with \( n \) states
GNFA->Regexp is correct

**Def:** GNFA->Regexp: input $G$ is a GNFA with $n$ states:
- If $n = 2$: return the reg expr on the transition
- Else ($G$ has $n > 2$ states):
  - “Rip” out one state to get $G'$
  - Recursively Call GNFA->Regexp($G'$)

**Proof** (by induction on size of $G$):
- **Base case:** $G$ has 2 states
  - $\text{LANGOF}(G) = \text{LANGOF}(\text{GNFA->Regexp}(G))$ is true!
- **IH:** Assume $\text{LANGOF}(G') = \text{LANGOF}(\text{GNFA->Regexp}(G'))$
  - For some $G'$ with $n-1$ states
- **Induction Step:** Prove it’s true for $G$ with $n$ states
  - After “rip” step, we have exactly a GNFA with $n-1$ states
  - And we know $\text{LANGOF}(G') = \text{LANGOF}(\text{GNFA->Regexp}(G'))$ from the IH!
GNFA→Regexp is correct

**Def:** GNFA→Regexp: input $G$ is a GNFA with $n$ states:
- If $n = 2$: return the reg expr on the transition
- Else ($G$ has $n > 2$ states):
  - “Rip” out one state to get $G'$
  - Recursively Call GNFA→Regexp($G'$)

**Proof** (by induction on size of $G$):
- **Base case:** $G$ has 2 states
  - $\text{LANGOF}(G) = \text{LANGOF}(\text{GNFA→Regexp}(G))$ is true!
- **IH:** Assume $\text{LANGOF}(G') = \text{LANGOF}(\text{GNFA→Regexp}(G'))$
  - For some $G'$ with $n-1$ states
- **Induction Step:** Prove it’s true for $G$ with $n$ states
  - After “rip” step, we have exactly a GNFA with $n-1$ states
  - And we know $\text{LANGOF}(G') = \text{LANGOF}(\text{GNFA→Regexp}(G'))$ from the IH!
  - To go from $G$ to $G'$: need to prove correctness of “rip” step
GNFA->Regexp: “rip” step correctness

Before:
- $q_i \xrightarrow{R_1} q_{rip} \xrightarrow{R_2} q_j$
- $q_i \xrightarrow{R_4} q_j$

These are equivalent

After:
- $(R_1)(R_2)^* (R_3) \cup (R_4) \xrightarrow{} q_j$

Must prove:
- Every string accepted before, is accepted after
- 2 cases:
  - Accepted string does not go through $q_{rip}$
    - Acceptance unchanged (both use $R_4$ transition part)
  - String goes through $q_{rip}$
    - Acceptance unchanged?

Mostly done this already!
Just need to state more formally
Thm: A lang is regular iff some reg expr describes it

• => If a language is regular, it is described by a reg expr
  • Hard!
  • Need to convert DFA or NFA to Regular Expression
  • Use GNFA->Regexp to convert GNFA to regular expression! (Done!)

• <= If a language is described by a reg expr, it is regular
  • Easy!
  • Construct the NFA! (Done)

Now we may confidently use regular expressions to represent regular langs.
Check-in Quiz 10/17
On gradescope

End of Class Survey 10/17
See course website