CMSC330: Finite State Machines

Chris Kauffman

Last Updated:
Tue Sep 12 11:23:32 AM EDT 2023

Logistics

Assignments

- Project 1 "Intro Python"
  Due Sun 10-Sep
- First Discussion Quiz during Discussion on Fri 15-Sep
  - 20min at beginning of discussion sections
  - Paper quiz, write answers, hand it in

Goals

- Recap of Regexs
- Finite State Machines
- Determinism vs Non-Determinism

Reading

Introduction to the Theory of Computation by Michael Sipser

- Chapter 1 covers theory associated with Finite State Machines and their relation to Regular Expresssions
- For the theoretically inclined, treatment is much tighter w/ proofs than our in-class work

Prof Bakalian's Notes on FSM

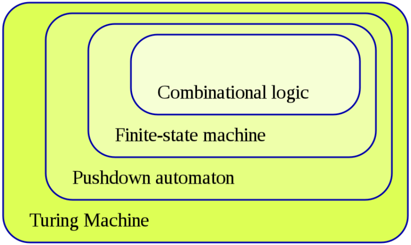
- A good summary of the topics we'll cover
- Linked on course schedule soon

Automata Theory

- Likely you've studied Boolean Logic in a previous class
- Allows the "computation" of certain outcomes based on inputs but has limits in power, does not amount to what a "computer" can do
- Example: cannot **recognize** Regular Expressions with Boolean Logic as Regexes can recognize infinite sets of strings
- **Automata Theory** is the branch of Math / CS that studies what (theoretical) machines with different properties can do
- By introducing notions of state (and time) one can build progressively more powerful machines

Levels of Computational Power

- A full course on Automata Theory would study each level, comparing, contrasting, formalizing
- Wouldn't leave much time for other fun things like Python, OCaml, Racket...
- In CMSC 330, will study **Finite State Machines (FSM)** also known as **Finite Automata (FA)** as an example of one level of power that is useful in language processing and is connected to Regular Expressions



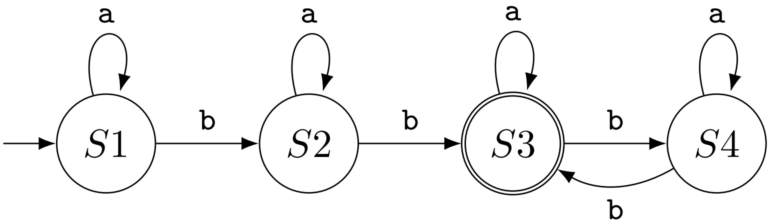
Source: Wikip "Automata Theory"

The class of problems that can be solved grows with more powerful machines.

Even-Bs: A Leading Example

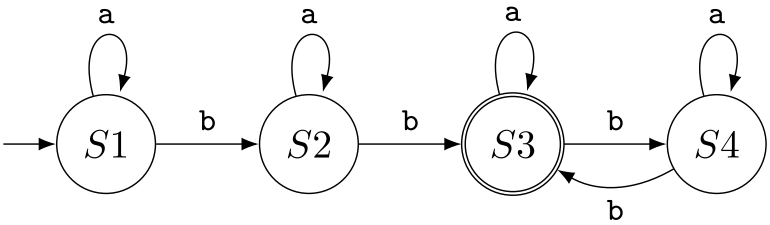
Let Even-Bs be the set of all strings composed of `a` and `b` with at least 2 `b`'s and an even number of `b`'s.

- Example members of Even-Bs are `bb, abb, aaababaa, abbabb, abba, babaaa, ...`
- **Regex** matching strings in Even-Bs: `(a*ba*ba*)+`
- **Deterministic Finite Automata (DFA)** recognizing Even-Bs

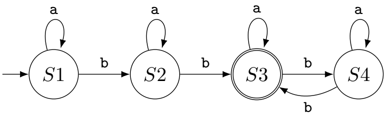


DFA Diagram Notation

- DFAs are **mathematical graphs** comprised of vertices (circles) and directed edges (arrows between circles)
- Each circle is a **state**; there are a finite number of them
- Each edge / transition is labeled with at least one item from the **input alphabet** like `a` or `b`
- There is one **start state** $S1$ in this case; note the arrow to it
- There are one or more **accept states** which are drawn with 2 circles like $S3$



Exercise: DFA Example Recognition / Rejection



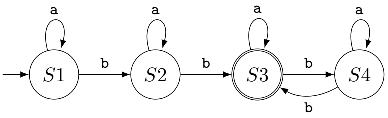

```
       v
input: abbabb
state: S1 a-> S1
        v
input: abbabb
state: S1 b-> S2
         v
input: abbabb
state: S2 b-> S3
          v
input: abbabb
state: S3 a-> S3
           v
input: abbabb
state: S3 b-> S4
            v
input: abbabb
state: S4 b-> S3
             v
input: abbabb
state: S3 ACCEPT
```

```
       v
input: bbaaba
state: S1 b-> S2
        v
input: bbaaba
state: S2 b-> S3
         v
input: bbaaba
state: S3 a-> S3
          v
input: bbaaba
state: S3 a-> S3
           v
input: bbaaba
state: S3 b-> S4
            v
input: bbaaba
state: S4 a-> S4
             v
input: bbaaba
state: S4 REJECT
```

```
       v
input: ababbba
???
???
```

Complete the state transitions

Answers: DFA Example Recognition / Rejection



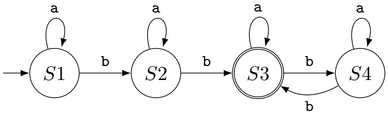

```
       v
input: abbabb
state: S1 a-> S1
        v
input: abbabb
state: S1 b-> S2
         v
input: abbabb
state: S2 b-> S3
          v
input: abbabb
state: S3 a-> S3
           v
input: abbabb
state: S3 b-> S4
            v
input: abbabb
state: S4 b-> S3
             v
input: abbabb
state: S3 ACCEPT
```

```
       v
input: bbaaba
state: S1 b-> S2
        v
input: bbaaba
state: S2 b-> S3
         v
input: bbaaba
state: S3 a-> S3
          v
input: bbaaba
state: S3 a-> S3
           v
input: bbaaba
state: S3 b-> S4
            v
input: bbaaba
state: S4 a-> S4
             v
input: bbaaba
state: S4 REJECT
```

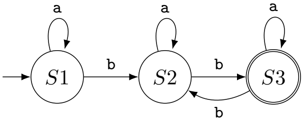
```
       v
input: ababbba
state: S1 a-> S1
        v
input: ababbba
state: S1 b-> S2
         v
input: ababbba
state: S2 a-> S2
          v
input: ababbba
state: S2 b-> S3
           v
input: ababbba
state: S3 b-> S4
            v
input: ababbba
state: S4 b-> S3
             v
input: ababbba
state: S3 a-> S3
              v
input: ababbba
state: S3 ACCEPT
```

DFAs are Not Unique

Even-Bs DFA #1



Even-Bs DFA #2



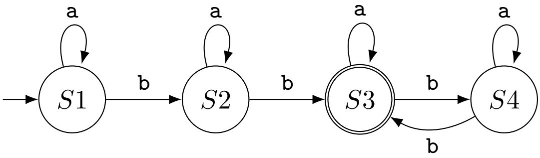
- Both these DFAs recognize the set Even-Bs but are shaped differently
- DFA Minimization finds a DFA which accepts the same input set but has a minimal number of states (subject to caveats)
- Regular Expressions are not unique either:

```
Even-Bs Regex 1: (a*ba*ba*)+
Even-Bs Regex 2: (a*ba*b)+a*
```

Finite State Machine Formalisms

Formally, a FSM is a 5-tuple (e.g. 5 parts, order matters)

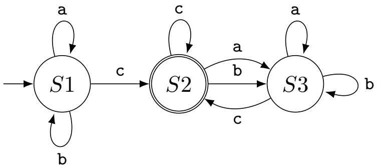
	Description	Sym	Even-Bs DFA #1
1	Alphabet: set of allowable characters	Σ	$\{a, b\}$
2	Set of States in FSM	S	$S = \{S1, S2, S3, S4\}$
3	Starting state of the FSM	s_0	$S1$
4	Set of Final / Accept States	F	$\{S3\}$
5	Set of transitions (labeled edges)[1]	δ	{(S1,a,S1), (S1,b,S2), (S2,a,S2), (S2,b,S3), (S3,a,S3), (S3,b,S4), (S4,a,S4), (S4,b,S3)}



Even-Bs DFA #1

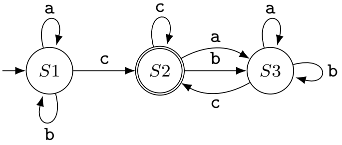
[1]The character δ is the lower-case Greek letter delta, often used to represent "change" as in a "change of state"; it's capital version is Δ

Exercise: DFA Practice

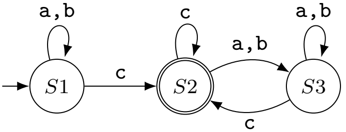


1. Show the formal 5-tuple of parts for this DFA
2. What set of strings does it accept?
3. Find a regular expression that matches that set
4. What set of strings does this Regex match?
   Regex: `[ab]*aab[ab]*`
5. Design a DFA that accepts the same set of strings

Answers: DFA Practice



Ends-C DFA

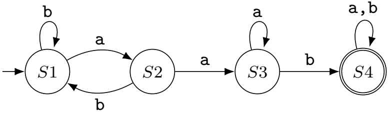


Ends-C DFA with Alt Notation

1. Show the formal 5-tuple of parts for this DFA

```
1. Alphabet: {a,b,c}
2. States: {S1,S2,S3}
3. Start: S1
4. Accept: {S2}
5. Transitions:
   {(S1,a,S1),(S1,b,S1),(S1,c,S2),
    (S2,a,S3),(S2,b,S3),(S2,c,S2),
    (S3,a,S3),(S3,b,S3),(S3,c,S2)}
```

2. What set of strings does it accept?
   Strings of `a,b,c` the end with `c`
3. Find a regular expression that matches that set
   Regex: `[abc]*c$`
   Note use of $ to denote end of input
4. What set of strings does this Regex match?
   Regex: `[ab]*aab[ab]*`
   Strings of `a,b` that contain the substring `aab`
5. Design a DFA that accepts the same set of strings



Has-AAB DFA
Adapted from Sipser Figure 1.13

DFAs in Code as Data Structures

```python
# even_Bs_dfa.py:
even_Bs_dfa = {
  "alphabet":{"a","b"},
  "nstates":4,
  "start":1,
  "accept":{3},
  "trans":[{},
           {"a":1,"b":2},
           {"a":2,"b":3},
           {"a":3,"b":4},
           {"a":4,"b":3}],
}

def dfa_match(dfa,instr):
  state = dfa["start"]
  trans = dfa["trans"]
  for i in instr:
    if not i in dfa["alphabet"]:
      return "Error"
    state = trans[state][i]
  if state in dfa["accept"]:
    return "Accept!"
  else:
    return "Reject"
```

- Encode the 5 parts of the DFA in some sort of data structure
- Python's built-in Lists, Dictionaries, Sets make this pleasant
- `dfa_match(dfa,instr)` will return Accept / Reject string using DFAs encoded as the example above
- The general goal of compiling a regular expression is to produce this kind of data structure
- Study the data structure and explain its parts

DFAs as Code

```c
// even_Bs_dfa.c:
int even_Bs_dfa(char *input){
  int pos=-1;
 S1:
  pos++;
  switch(input[pos]){
    case 'a':  goto S1;
    case 'b':  goto S2;
    case '\0': goto REJECT;
    default:   goto ERROR;
  }
 S2:
  pos++;
  switch(input[pos]){
    case 'a':  goto S2;
    case 'b':  goto S3;
    case '\0': goto REJECT;
    default:   goto ERROR;
  }
 S3:
  pos++;
  switch(input[pos]){
    case 'a':  goto S3;
    case 'b':  goto S4;
    case '\0': goto ACCEPT;
    default:   goto ERROR;
  }
 S4:
  pos++;
  switch(input[pos]){
```

- A common output option for parsing tools like Lex and Yacc is to encode state machines as positions in code
- Instruction Pointer is "state"
- Tools process a Regex or more complex language **Grammar** then generates C code that represents the state machine
- Generated C code is nigh impenetrable BUT compiles to much faster recognition routines than alternatives
- With all those `goto`'s, you know... Here be Dragons

Formal Regular Expressions

- Introduced Regexs in code somewhat informally as a pattern matching device
- Formally, Regular Expressions are
  1. ϵ, the Empty String (zero-length)
  2. $\emptyset$, the empty set of no regexs
  3. Single item like `a` from an alphabet Σ
  4. $R_1 R_2$, concatenation of two regexs
  5. $R_1 | R_2$, alternation / union of two regexs
  6. $R_1 *$, zero-or-more of a regex, its **Kleene Closure**[2]
- These 6 parts are minimal, allow construction of all the regex convenience mechanisms we've seen so far, and limit the cases of in formal proofs

  Ex: Shorthand: `[ab]+` Longhand: `(a|b)(a|b)*`

[2] Named for Stephen Kleene who studied under Alonzo Church and contributed to the development of Church's Lambda Calculus

Equivalence of FSM and Regular Expressions

Definition: A language is **Regular** if some Finite State Machine accepts it.

Using a series of proofs one can show the following:

1. A language is Regular if and only if some **Regular Expression** describes it; *shown by giving a procedure to convert a Regular Expression to a Non-deterministic Finite Automata (NFA) (Compile!)*
2. Regular Expressions are closed under 3 simple combination operations; *e.g. all regexs can that exist can be built from simpler regexs*
3. Every NFA has an equivalent DFA; *procedures exist to convert NFAs to DFAs that accept the same language; we'll study this*

Conclusion: Regular Expressions and Finite State Machines are equivalent in power, allow recognition of identical sets

If you want to see those proofs, grab a copy of Sipser's Introduction to the Theory of Computation

Nonregular Languages and the Limits Regexes/FSMs

- Before moving forward, note that Regexs / FSMs hit practical limits in power quickly and in cases we'd want to overcome
- Example: Let Equal-ABs be the set of all strings start some number n of `a` characters and are followed immediately by n `b` characters. written formally
  - Equal-ABs $= \{a^n b^n | n > 0\}$
  - Equal-ABs = `{ab, aabb, aaabbb, aaaabbbb, ...}`
- Fool's Errands:
  - Construct a DFA to accept Equal-ABs
  - Write a Regex matching Equal-ABs
  - **No such DFA or Regex Exists**
- Why do we care? Well, a similar set is **Blanced-Paren**, the set of all strings that have properly balanced parentheses
  - Balanced-Paren = `{(), (()), ((())), ...}`
- One needs a more powerful machine than FSMs / Regexs to properly recognize Equal-ABs and Balanced-Parens which is crucial for processing programming languages

—END Tue 12-Sep TOPICS—

Non-Deterministic Finite Automata

Equivalence of Power between DFAs and NFAs

Why DFA vs NFA?

Conversion from NFA to DFA

Optional: Conversion of Regex to NFA

Other Uses for Finite State Machines

Regexs in Other Languages