Lecture 3: Finite Automata

Dr Kieran T. Herley

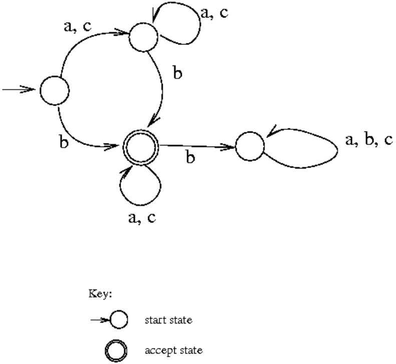
Department of Computer Science
University College Cork

2018/19

Summary

Deterministic finite automata (DFA). Definition and operation of same. DFAs as string classifiers or pattern recognizers. Algorithm for simulation of DFAs.

Finite Automata

- A *finite automaton* (aka finite state machine) is a simple abstract device that categorizes input strings as either *accepted* or *rejected*.
- 

Operation of (deterministic) finite automaton:

- Place token on start state.
- For each input symbol in turn, move token along (unique) edge whose label matches that symbol.
- Accept/reject input if token in accept/non-accept state at the end.

Deterministic Finite Automata

Definition

Deterministic Finite Automaton (DFA) [a] consists of

Σ an alphabet

S a finite set of states

s_0 a start state

A a set of accept states and

$T = (S, E)$ a directed graph in which each edge in E is labelled with one or more elements of Σ and no two edges bearing the same label emanate from the same state.

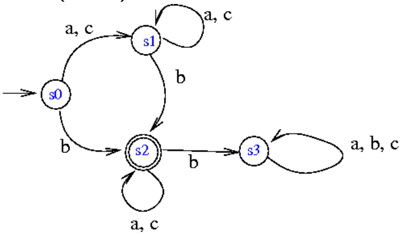
[a] Will consider *nondeterministic* finite automata (NFA) later

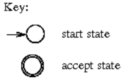
Example cont'd

-

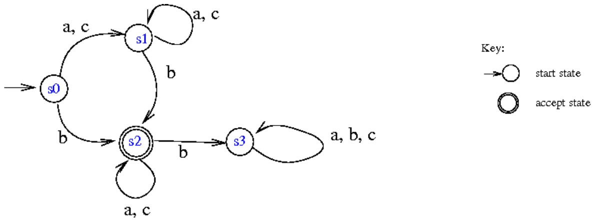
$$
\begin{aligned}
\Sigma &= \{a, b, c\} \\
S &= \{s_0, s_1, s_2, s_3\} \\
A &= \{s_2\}
\end{aligned}
$$

- $T = (S, E)$





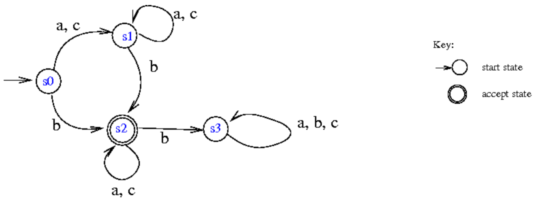
Aside



Transition graph often represented by a transition table:

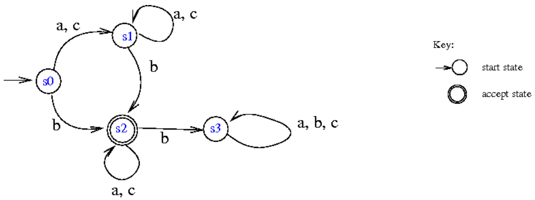
	a	b	c
s_0	s_1	s_2	s_1
s_1	s_1	s_2	s_1
s_2	s_2	s_3	s_2
s_3	s_3	s_3	s_3

Illustration of FA in Action



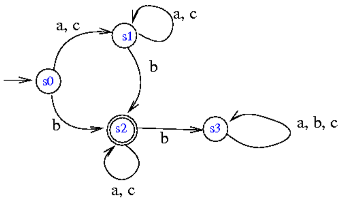
Input: $a \quad c \quad a \quad c \quad c \quad a$

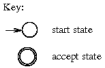
Illustration of FA in Action



Input: $a \quad c \quad a \quad c \quad c \quad a$ Reject!
Input: $a \quad c \quad b \quad c \quad c \quad a$

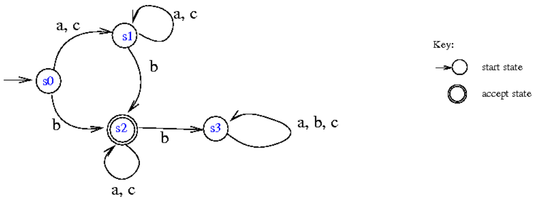
Illustration of FA in Action





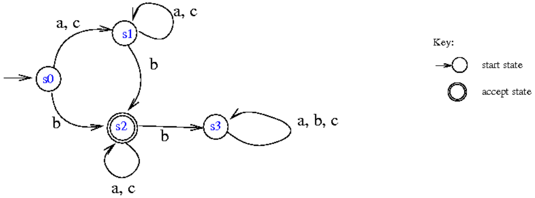
Input: *a* *c* *a* *c* *c* *a* Reject!
Input: *a* *c* *b* *c* *c* *a* Accept!
Input: *a* *a* *b* *b* *c* *a*

Illustration of FA in Action



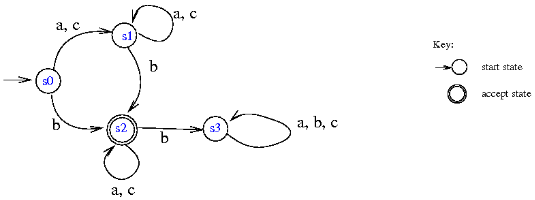
Input: a c a c c a Reject!
Input: a c b c c a Accept!
Input: a a b b c a Reject!
Input: b a c b c a

Illustration of FA in Action



Input: a c a c c a Reject!
Input: a c b c c a Accept!
Input: a a b b c a Reject!
Input: b a c b c a Reject!
Input: b b c a c a

Illustration of FA in Action



Input: a c a c c a Reject!
Input: a c b c c a Accept!
Input: a a b b c a Reject!
Input: b a c b c a Reject!
Input: b b c a c a Reject!

Observation

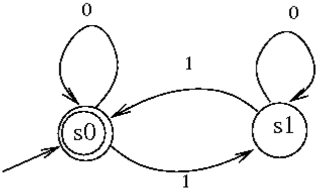
Accepts all strings with exactly one b.

Formal Criterion

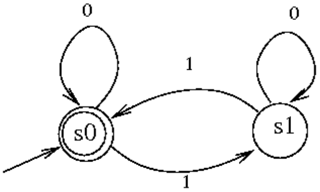
Definition

The automaton *accepts* string $x_1 x_2 \cdots x_n$ if there is a path in T from the start state to one of the accept states such that for each i, the ith edge on the path bears label x_i. All other strings are *rejected*.

Pic



Example

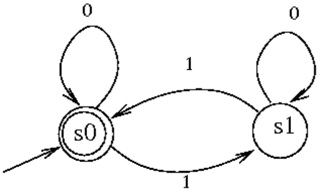
- 

- Accepts 01001:

$$s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_1 \xrightarrow{0} s_1 \xrightarrow{0} s_1 \xrightarrow{1} s_0$$

- Rejects 010101.

Example

- 

- Accepts 01001:

$$s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_1 \xrightarrow{0} s_1 \xrightarrow{0} s_1 \xrightarrow{1} s_0$$

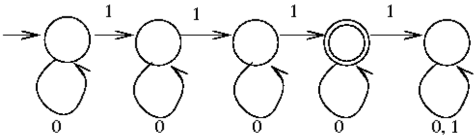
- Rejects 010101.
-

Observation

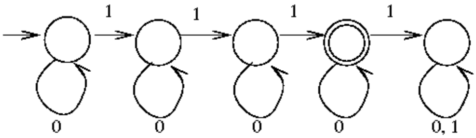
Accepts strings of even parity, rejects odd.

How to prove this?

Another Example

- 

Another Example

- 

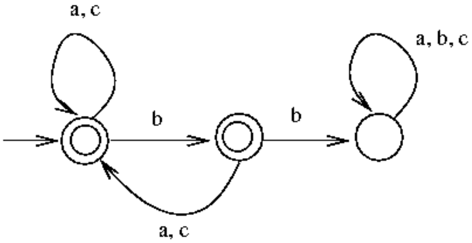
-

Claim

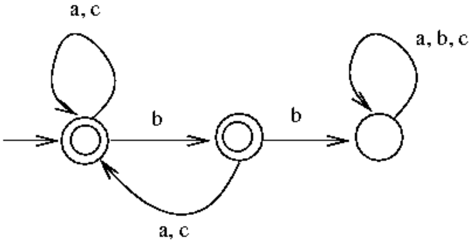
Accepts all strings containing exactly three 1*s.*

- Each '0' leaves state unchanged, each '1' moves one state to right
- Rightmost state is a "trap state": once entered there is no way of "escaping" and reaching the accept state to its left.

Yet Another Example

- 

Yet Another Example

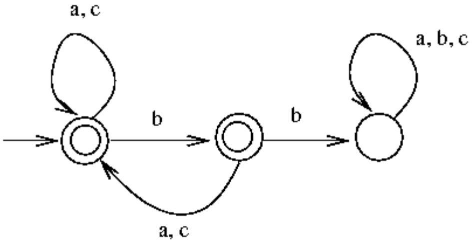
- 

-

Claim

Accepts strings that do not contain two consecutive bs

Yet Another Example

- 

-

Claim

Accepts strings that do not contain two consecutive bs

Algorithm to Simulate DFA

```
Algorithm DfaAccept(M, X):
    s ← M.start
    ch ← X.nextChar()
    while ch ≠ eof do
        s ← M.moveTo(s, ch)
        ch ← X.nextChar()
    if s in M.accept then
        return true
    else
        return false
```

Algorithm to Simulate DFA

```
Algorithm DfaAccept(M, X):
    s ← M.start
    ch ← X.nextChar()
    while ch ≠ eof do
        s ← M.moveTo(s, ch)
        ch ← X.nextChar()
    if s in M.accept then
        return true
    else
        return false
```

M = Automaton

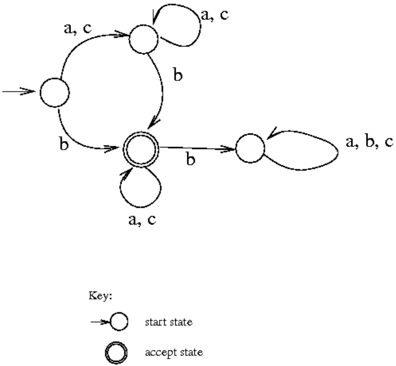
M.start start state
M.accept accept states
M.moveTo(..) transition function

X = input

X.nextChar() returns next char
eof Special end-of-input char

Algorithm to Simulate DFA

Algorithm DfaAccept(M, X):
  s $\leftarrow$ M.start
  ch $\leftarrow$ X.nextChar()
  while ch $\neq$ eof **do**
    s $\leftarrow$ M.moveTo(s, ch)
    ch $\leftarrow$ X.nextChar()
  if s in M.accept **then**
    return true
  else
    return false



Algorithm emulates "token-tracing" process:

- *s* encodes state holding token
- moveTo emulates state transitions

Algorithm to Simulate DFA

```
Algorithm DfaAccept(M, X):
    s ← M.start
    ch ← X.nextChar()
    while ch ≠ eof do
        s ← M.moveTo(s, ch)
        ch ← X.nextChar()
    if s in M.accept then
        return true
    else
        return false
```

Theorem

DfaAccept returns true if and only if automaton M accepts string X.

- For DFAs, string-path correspondance is unique
- Sequence of s values implicitly defines path
- and *vice versa*

Algorithm to Simulate DFA

```
Algorithm DfaAccept(M, X):
    s ← M.start
    ch ← X.nextChar()
    while ch ≠ eof do
        s ← M.moveTo(s, ch)
        ch ← X.nextChar()
    if s in M.accept then
        return true
    else
        return false
```

Claim

DfaAccept(M, X) *runs in* $O(|X|)$ *time i.e. linear in the input length.*

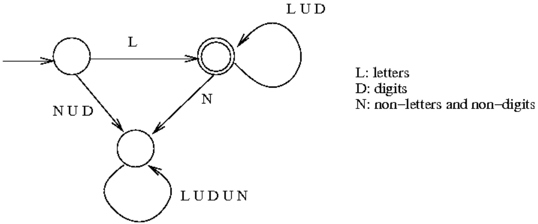
Representing the Transition Function

- Transition function typically represented as table
- Example:

	a	b	c	$\cdots$	z	$\cdots$
s_0						
s_1						
s_2						
$\vdots$						
s_n						

- For large alphabets and state sets, tables can be huge; also typically sparse.
- Permits fast lookup ($O(1)$), but space hungry.
- (Could also represent function using (say) map.)

FAs As Pattern Recognizers

- Since FAs categorize strings (accept/reject), can think of them as language recognizers: device to characterize which strings belong to some set and which don't.
- 

- Accepts strings consisting of letters and digits that begin with a letter i.e. valid Pascal identifiers

Aside – String Matching

Setting

Text T[1..n]

a	b	a	a	b	b	a	b	a	b	b	b	b
1	2	3	4	5	6	7	8	9	10	11	12	13

Pattern P[1..m]

a	b	a	b	b
1	2	3	4	5

Matches P occurs with *shift* s if $P[1..m] = T[s+1..s+m]$

Goal Locate all occurrences of P in T.

Naive Algorithm

Algorithm NaiveStringMatch(T, P):
  for s ←0 **to** n − m **do**
    if P [1.. m] = T[s+1..s+m] **then**
      print "pattern occurs with shift " s

Running Time $= O(nm)$ (SLOW)

DFAs and String Matching

Recall DFA can act as simple pattern recognition device, accepting some strings, rejecting others

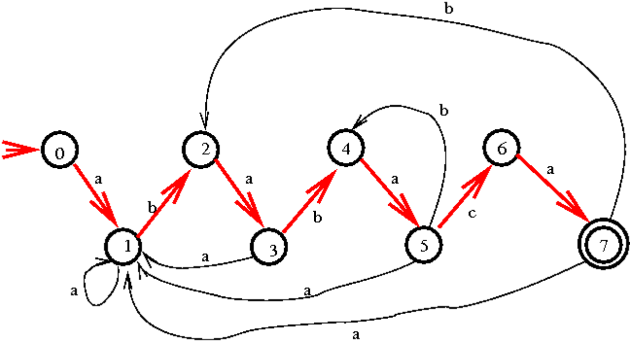
Fact

For every string p there is a simple DFA $M(p)$ that "recognizes" it. (Not obvious, but true.)

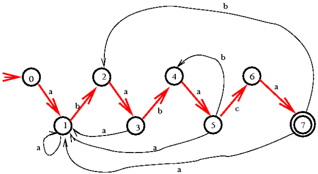
Cunning Plan

- Build a DFA that "recognizes" string p
- Run input through DFA
- Watch for token entering accept state (signifies pattern occurrence)

M(ababaca) a DFA for ababaca



M(ababaca) cont'd



Notes

- Complete DFA has exactly one transition per state for each alphabet symbol
- Implicit transitions of for $s_? \rightarrow s_0$ are not shown to avoid cluttering diagram

Fascinating Fact Beginning at any state s, by following transitions labelled a-b-a-b-a-c-a you end up in the accept state!

Corollary DFA detects (by entering accept state) each occurrence of pattern ababaca in the input string