Chapter Nine: Advanced Topics in Regular Languages

There are many more things to learn about finite automata than are covered in this book. There are many variations with interesting applications, and there is a large body of theory. Especially interesting, but beyond the scope of this book, are the various algebras that arise around finite automata. This chapter gives just a taste of some of these advanced topics.

Outline

- 9.1 DFA Minimization
- 9.2 Two-Way Finite Automata
- 9.3 Finite-State Transducers
- 9.4 Advanced Regular Expressions

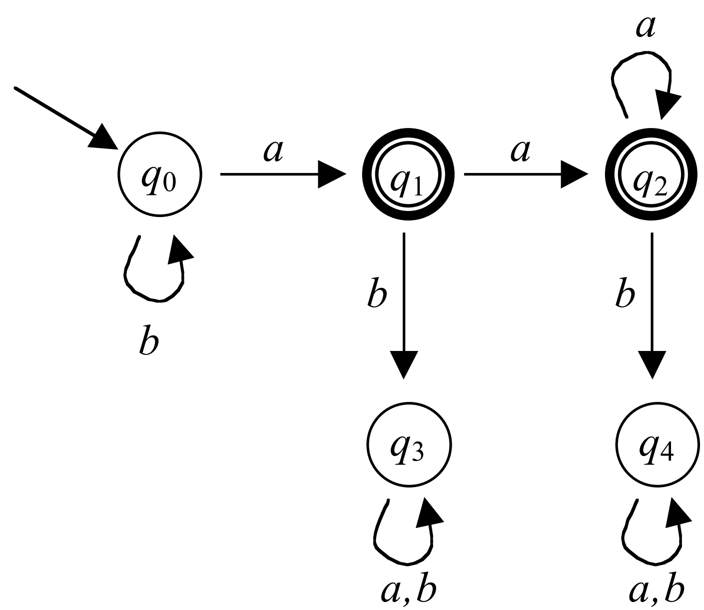
DFA Minimization

- Questions of DFA size:
  - Given a DFA, can we find one with fewer states that accepts the same language?
  - What is the smallest DFA for a given language?
  - Is the smallest DFA unique, or can there be more than one "smallest" DFA for the same language?
- All these questions have neat answers…

Eliminating Unnecessary States

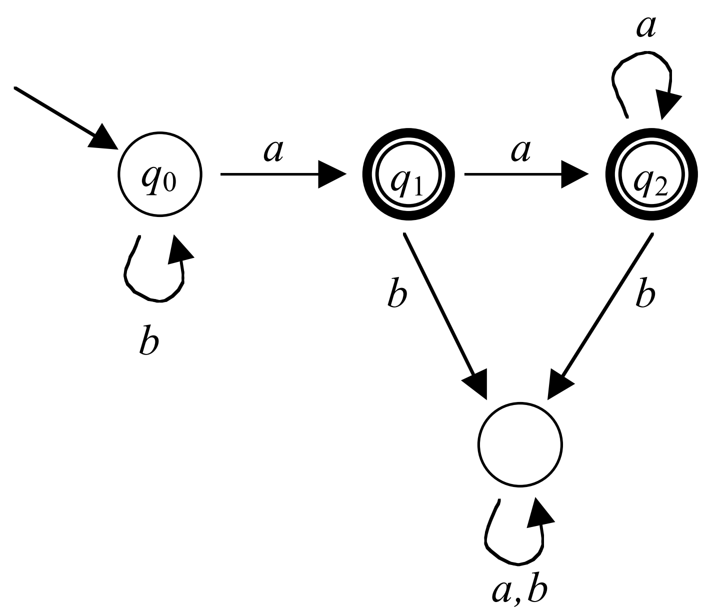
- Unreachable states, like some of those introduced by the subset construction, can obviously be eliminated
- Even some of the reachable states may be redundant…

Example: Equivalent States



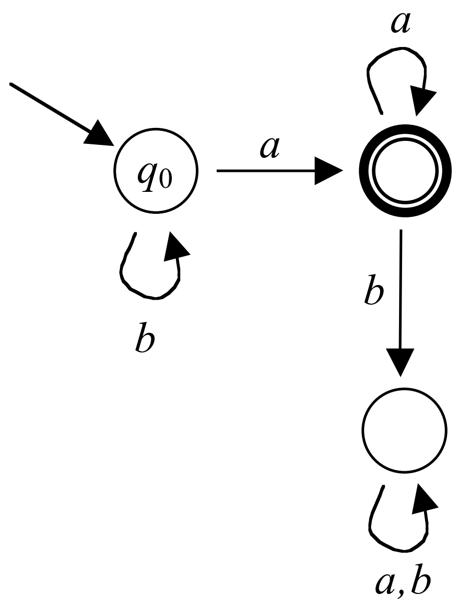
- In both q_3 and q_4, the machine rejects, no matter what the rest of the input string contains
- They're equivalent and can be combined…

Still More Equivalent States



- In both q_1 and q_2, the machine accepts if and only if the rest of the string consists of 0 or more *a*s
- They're equivalent and can be combined…

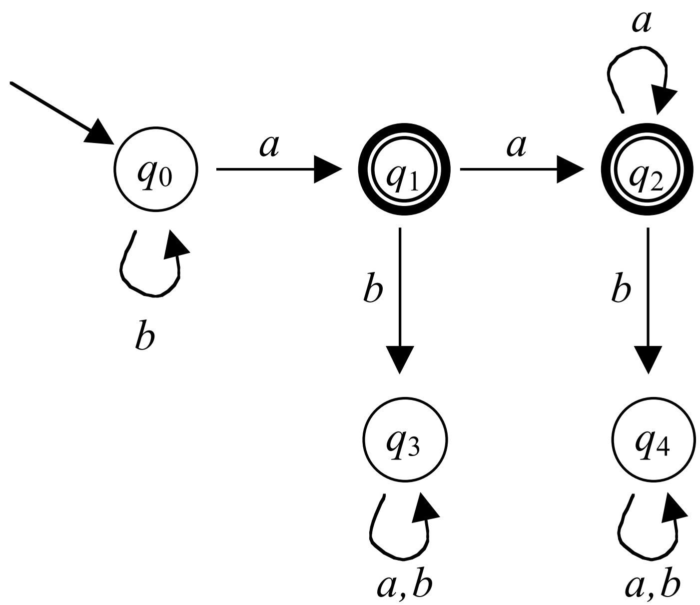
Minimized



- No more equivalencies
- This is a minimum-state DFA for the language, $\{xay \mid x \in \{b\}^* \text{ and } y \in \{a\}^*\}$

State Equivalence

- Informally: two states are equivalent when the machine's decision after any remaining input will be the same from either state
- Formally:
  - Define a little language $L(M,q)$ for each state q: $L(M,q) = \{x \in \Sigma^* \mid \delta^*(q,x) \in F\}$
  - That's the language of strings that would be accepted if q were used as the start state
  - Now we can define state equivalence: q and r are equivalent if and only if $L(M,q) = L(M,r)$



- We have:
  - $L(M,q_0) = \{xay \mid x \in \{b\}^* \text{ and } y \in \{a\}^*\}$
  - $L(M,q_1) = \{x \mid x \in \{a\}^*\}$
  - $L(M,q_2) = \{x \mid x \in \{a\}^*\}$
  - $L(M,q_3) = \{\}$
  - $L(M,q_4) = \{\}$
- So $q_1 \equiv q_2$ and $q_3 \equiv q_4$

DFA Minimization Procedure

- Two steps:
  - 1. Eliminated states that are not reachable from the start state
  - 2. Combine all equivalent states, so that no two remaining states are equivalent to each other
- Step 2 is the construction of a new DFA whose states are the equivalence classes of the original: the *quotient construction*

Theorem 9.1

Every regular language has a unique minimum-state DFA, and no matter what DFA for the language you start with, the minimization procedure finds it.

- (Stated here without proof)
- Resulting DFA is unique up to isomorphism
  - That is, unique except perhaps for state names, which of course have no effect on $L(M)$
- So our minimization procedure is safe and effective
  - Safe, in that it does not change $L(M)$
  - Effective, in that it finds the structurally unique smallest DFA for $L(M)$

Automating Minimization

- Is there an algorithm that can efficiently detect equivalent states and so perform the minimization?
- Yes: a DFA with state set Q and alphabet Σ can be minimized in $O(|\Sigma|\ |Q| \log |Q|)$ time
- (reference in the book)

Minimizing NFAs

- Results are not as clean for NFAs
- We can still eliminate unreachable states
- We can still combine equivalent states, using a similar definition of equivalence
- But the result is not necessarily a unique minimum-state NFA for the language
- (reference in the book)

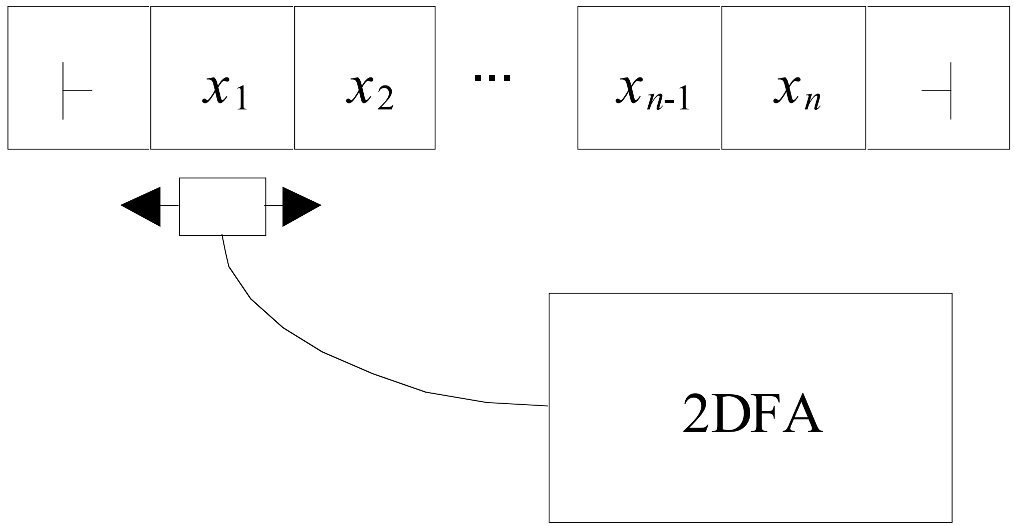
Outline

- 9.1 DFA Minimization
- 9.2 Two-Way Finite Automata
- 9.3 Finite-State Transducers
- 9.4 Advanced Regular Expressions

Two-Way Finite Automata

- DFAs and NFAs read their input once, left to right
- We can try to make these models more powerful by allowing re-reading
- Treat the input like a tape, and allow the automaton to move its read head left or right on each move
  - Two-way deterministic finite automata (2DFA)
  - Two-way nondeterministic finite automata (2NFA)

2DFA Example



- Input string $x_1x_2\ldots x_{n-1}x_n$
- Special endmarker symbols frame the input
- The head can't move past them

2DFA Differences

- Transition function returns a pair of values:
  - The next state
  - The direction (*L* or *R*) to move the head
- Computation ends, not at the end of the string, but when a special state is entered
  - One accept state: halt and accept when entered
  - One reject state: halt and reject when entered
- Can these define more than the regular languages?

Theorems 9.2.1 And 9.2.2

There is a 2DFA for a language *L* if and only if *L* is regular.

There is a 2NFA for a language *L* if and only if *L* is regular.

- (Stated here without proof)
- So adding two-way reading to finite automata does not increase their definitional power
- A little more tweaking will give us more power later:
  - adding the ability to write as well as read yields LBAs (linear bounded automata), which are much more powerful
  - Adding the ability to write unboundedly far past the end markers yields the Turing machines, still more powerful

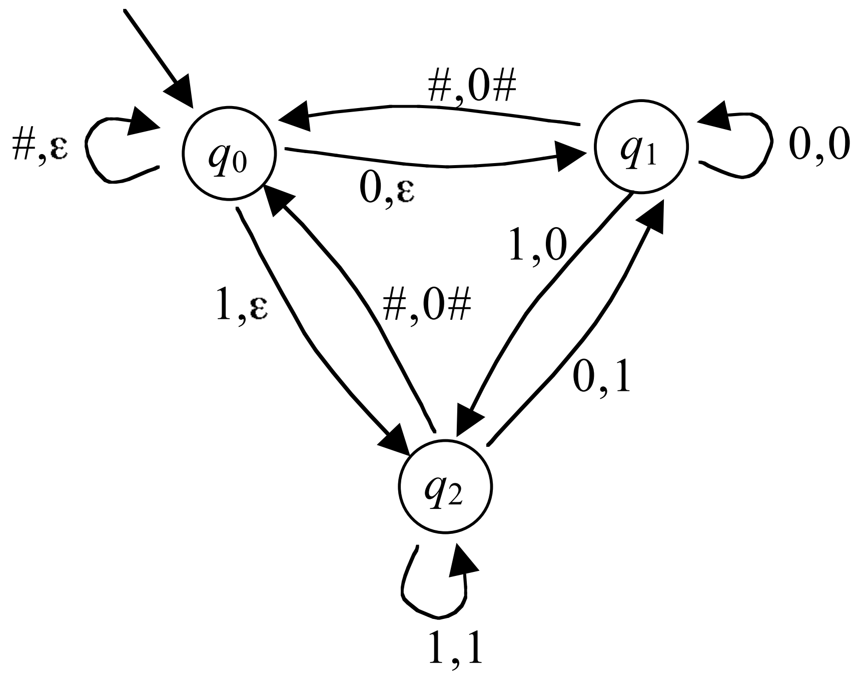
Outline

- 9.1 DFA Minimization
- 9.2 Two-Way Finite Automata
- 9.3 Finite-State Transducers
- 9.4 Advanced Regular Expressions

Finite-State Transducers

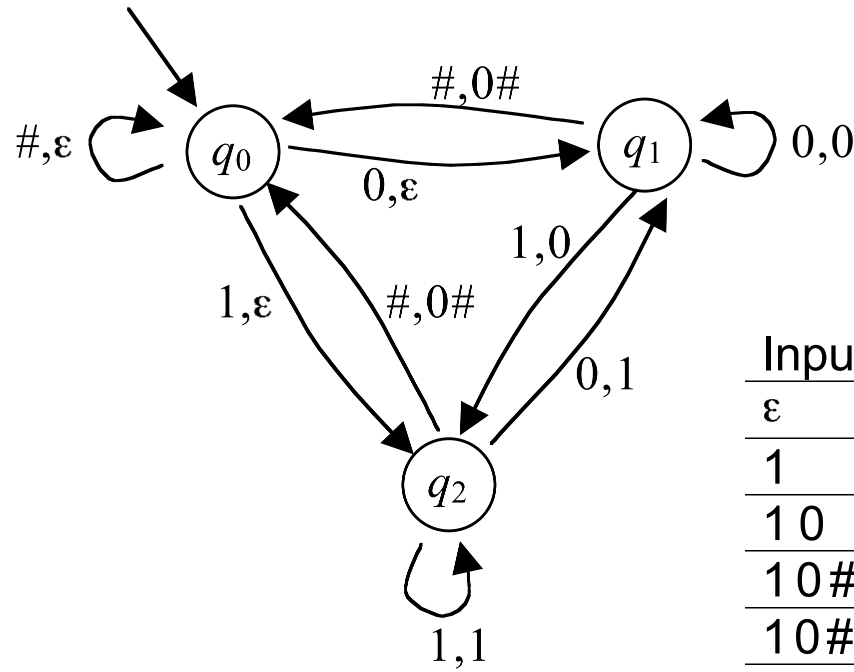
- Our DFAs and NFAs have a single bit of output: accept/reject
- For some applications we need more output than that
- Finite-state machines with string output are called *finite-state transducers*

Output On Each Transition



- A transition labeled *a*,*x* works like a DFA transition labeled *a*, but also outputs string *x*
- Transforms input strings into output strings

Action For Input 10#11#



Input read	State	Output so far
ε	q_0	ε
1	q_2	ε
10	q_1	1
10#	q_0	10#
10#1	q_2	10#
10#11	q_2	10#1
10#11#	q_0	10#10#

Transducers

- Given a sequence of binary numbers, it outputs the same numbers rounded down to the nearest even
- We can think of it as modifying an input signal
- Finite-state transducers have signal-processing applications:
  – Natural language processing
  – Speech recognition
  – Speech synthesis
  – (reference in the book)
- They come in many varieties
  – deterministic / nondeterministic
  – output on transition / output in state
  – software / hardware

Outline

- 9.1 DFA Minimization
- 9.2 Two-Way Finite Automata
- 9.3 Finite-State Transducers
- 9.4 Advanced Regular Expressions

Regular Expression Equivalence

- Chapter 7 grading: decide whether a regular expression is a correct answer to a problem
- That is, decide whether two regular expressions (your solution and mine) define the same language
- We've seen a way to automate this:
  – convert to NFAs (as in Appendix A)
  – convert to DFAs (subset construction)
  – minimize the DFAs (quotient construction)
  – compare: the original regular expressions are equivalent if the resulting DFAs are identical
- But this is extremely expensive

Cost of Equivalence Testing

- The problem of deciding regular expression equivalence is PSPACE-complete
  - Informally: PSPACE-complete problems are generally believed (but not proved) to require exponential time
  - More about this in Chapter 20
- The problem gets even worse when we extend regular expressions...

Regular Expressions With Squaring

- Add one more kind of compound expression:
  - $(r)^2$, with $L((r)^2) = L((r)(r))$
- Obviously, this doesn't add power
- But it does allow you to express some regular languages much more compactly
- Consider $\{0^n \mid n \bmod 64 = 0\}$, without squaring and with squaring:

(0000000000000000000000000000000000000000000000000000000000000000)*

$(((((0000)^2)^2)^2)^2)^*$

New Complexity

- The problem of deciding regular expression equivalence, when squaring is allowed, is EXPSPACE-complete
  - Informally: EXPSPACE-complete problems require exponential space and at least exponential time
  - More about this in Chapter 20
- Cost is measured as a function of the input size -- and we've compressed the input size using squaring
- The problem gets even worse when we extend regular expressions in other ways...

Regular Expressions With Complement

- Add one more kind of compound expression:
  - $(r)^C$, with $L((r)^C)$ defined to be the complement of $L(r)$
- Obviously, this doesn't add power; regular languages are closed for complement
- But it does allow you to express some regular languages much more compactly
- See Chapter 9, Exercise 3

Still More Complexity

- The problem of deciding regular expression equivalence, when complement is allowed, requires NONELEMENTARY TIME
  - Informally: the time required to compare two regular expressions of length *n* grows faster than

$$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$$

    for any fixed-height stack of exponentiations
  - More about this in Chapter 20

Star Height

- The star height of a regular expression is the nesting depth of Kleene stars
  - $a+b$ has star height 0
  - $(a+b)^*$ has star height 1
  - $(a^*+b^*)^*$ has star height 2
  - etc.
- It is often possible, and desirable, to simplify expressions in a way that reduces star height
  - $\varnothing^*$ defines the same language as ε
  - $(a^*+b^*)^*$ defines the same language as $(a+b)^*$

Star Height Questions

- Is there some fixed star height (perhaps 1 or 2) that suffices for any regular language?
- Is there an algorithm that can take a regular expression and find the minimum possible star height for any equivalent expression?

Star Height Questions

- For basic regular expressions, the answers are known:
  - Is there some fixed star height (perhaps 1 or 2) that suffices for any regular language? -- No, we need arbitrary star height to cover the regular languages
  - Is there an algorithm that can take a regular expression and find the minimum possible star height for any equivalent expression? -- Yes, there is an algorithm for minimizing star height
- When complement is allowed, these questions are still open

Generalized Star-Height Problem

- In particular, when complement is allowed, it is not known whether there is a regular language that requires a star height greater than 1!
- This is one of the most prominent open problems surrounding regular expressions