Chapter Two: Finite Automata

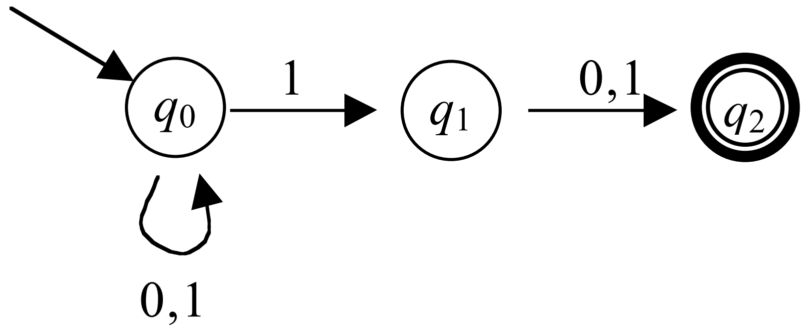
The problem with implementing NFAs is that, being nondeterministic, they do not really define computational procedures for testing language membership. To implement an NFA we must give a computational procedure that can look at a string and decide whether the NFA has at least one sequence of legal transitions on that string leading to an accepting state. This seems to require searching through all legal sequences for the given input string—but how?

One approach is to implement a direct backtracking search. Another is to convert the NFA into a DFA and implement that instead. This conversion is both useful and theoretically interesting: the fact that it is always possible shows that in spite of their extra flexibility, NFAs have exactly the same power as DFAs. They can define exactly the regular languages.

Outline

- 6.1 NFA Implemented With Backtracking Search
- 6.2 NFA Implemented With Bit-Mapped Parallel Search
- 6.3 The Subset Construction
- 6.4 NFAs Are Exactly As Powerful As DFAs
- 6.5 DFA Or NFA?

An NFA Example



- $L(N)$ is the language strings over the alphabet {0,1} that have a 1 as the next-to-last symbol
- We will implement it with backtracking search in Java
- We will use a three-dimensional transition array
- **`delta[s,c-'0']`** will be an array of 0 or more possible next states

```
/**
 * A nondeterministic finite-state automaton that
 * recognizes strings of 0s and 1s with 1 as the
 * next-to-last character.
 */
public class NFA1 {

  /*
   * The transition function represented as an array.
   * The entry at delta[s,c-'0'] is an array of 0 or
   * more ints, one for each possible move from
   * state s on character c.
   */
  private static int[][][] delta =
     {{{0},{0,1}}, // delta[q0,0], delta[q0,1]
      {{2},{2}},   // delta[q1,0], delta[q1,1]
      {{},{}}};    // delta[q2,0], delta[q2,1]
```

```
/**
 * Test whether there is some path for the NFA to
 * reach an accepting state from the given state,
 * reading the given string at the given character
 * position.
 * @param s the current state
 * @param in the input string
 * @param pos index of the next char in the string
 * @return true iff the NFA accepts on some path
 */
private static boolean accepts
    (int s, String in, int pos) {
  if (pos==in.length()) { // if no more to read
    return (s==2); // accept iff final state is q2
  }
```

```
  char c = in.charAt(pos++); // get char and advance
  int[] nextStates;
  try {
    nextStates = delta[s][c-'0'];
  }
  catch (ArrayIndexOutOfBoundsException ex) {
    return false; // no transition, just reject
  }

  // At this point, nextStates is an array of 0 or
  // more next states.  Try each move recursively;
  // if it leads to an accepting state return true.

  for (int i=0; i < nextStates.length; i++) {
    if (accepts(nextStates[i], in, pos)) return true;
  }

  return false; // all moves fail, return false
}
```

```
  /**
   * Test whether the NFA accepts the string.
   * @param in the String to test
   * @return true iff the NFA accepts on some path
   */
  public static boolean accepts(String in) {
    return accepts(0, in, 0); // start in q0 at char 0
  }
}
```

Not object-oriented: all static methods
All recursive search information is carried in the parameters
Usage example:

```
if (NFA1.accepts(s)) ...
```

Outline

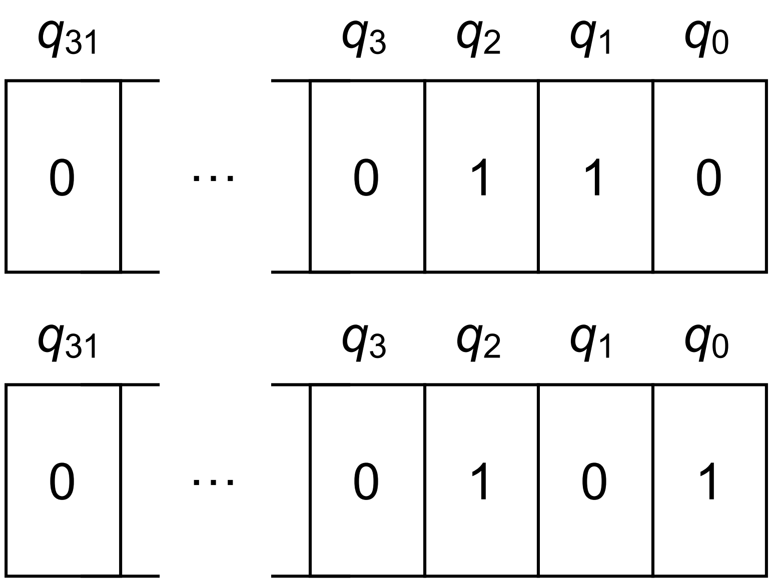
- 6.1 NFA Implemented With Backtracking Search
- 6.2 NFA Implemented With Bit-Mapped Parallel Search
- 6.3 The Subset Construction
- 6.4 NFAs Are Exactly As Powerful As DFAs
- 6.5 DFA Or NFA?

Parallel Search

- The previous implementation was a backtracking search
  - Try one sequence of moves
  - If that fails, back up a try another
  - Keep going until you find an accepting sequence, or run out of sequences to try
- You can also search all sequences at once
- Instead of keeping track of one current state, keep track of the set of all possible states

Bit-Coded Sets

- We'll use machine words to represent sets
- One bit position for each state, with a 1 at that position if the state is in the set



The set $\{q_1,q_2\}$

The set $\{q_0,q_2\}$

Bit-Coded Sets in Java

- The `<<` operator in Java shifts an integer to the left
- We're using `1<<i` for state *i*
- The `|` operator combines integers using logical OR

q_{31}		…	q_3	q_2	q_1	q_0
0		…	0	1	1	0

The set $\{q_1,q_2\}$
Java: `1<<1 | 1<<2`

q_{31}		…	q_3	q_2	q_1	q_0
0		…	0	1	0	1

The set $\{q_0,q_2\}$
Java: `1<<0 | 1<<2`

```
/**
 * A nondeterministic finite-state automaton that
 * recognizes strings with 0 as the next-to-last
 * character.
 */
public class NFA2 {

  /*
   * The current set of states, encoded bitwise:
   * state i is represented by the bit 1<<i.
   */
  private int stateSet;

  /**
   * Reset the current state set to {the start state}.
   */
  public void reset() {
    stateSet = 1<<0; // {q0}
  }
```

```
/*
 * The transition function represented as an array.
 * The set of next states from a given state s and
 * character c is at delta[s,c-'0'].
 */
static private int[][] delta =
   {{1<<0, 1<<0|1<<1}, // delta[q0,0] = {q0}
                       // delta[q0,1] = {q0,q1}
    {1<<2, 1<<2}, // delta[q1,0] = {q2}
                  // delta[q1,1] = {q2}
    {0, 0}}; // delta[q2,0] = {}
             // delta[q2,1] = {}
```

```java
/**
 * Make one state-set to state-set transition on
 * each char in the given string.
 * @param in the String to use
 */
public void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    int nextSS = 0; // next state set, initially empty
    for (int s = 0; s <= 2; s++) { // for each state s
      if ((stateSet&(1<<s)) != 0) { // if maybe in s
        try {
          nextSS |= delta[s][c-'0'];
        }
        catch (ArrayIndexOutOfBoundsException ex) {
          // in effect, nextSS |= 0
        }
      }
    }
    stateSet = nextSS; // new state set after c
  }
}
```

```
 /**
   * Test whether the NFA accepted the string.
   * @return true iff the final set includes
   *         an accepting state
   */
  public boolean accepted() {
    return (stateSet&(1<<2))!=0; // true if q2 in set
  }
}
```

Usage example:

```
NFA2 m = new NFA2();
m.reset();
m.process(s);
if (m.accepted()) ...
```

Larger NFAs

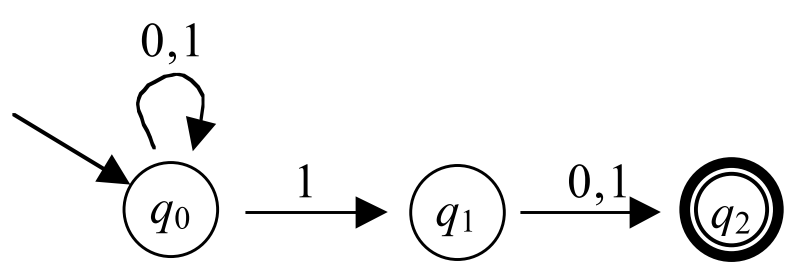
- Generalizes easily for NFAs of up to 32 states
- Easy to push it up to 64 (using **`long`** instead of **`int`**)
- Harder to implement above 64 states
- Could use an array of $\lceil n/32 \rceil$ **`int`** variables to represent *n* states
- That would make **`process`** slower and more complicated

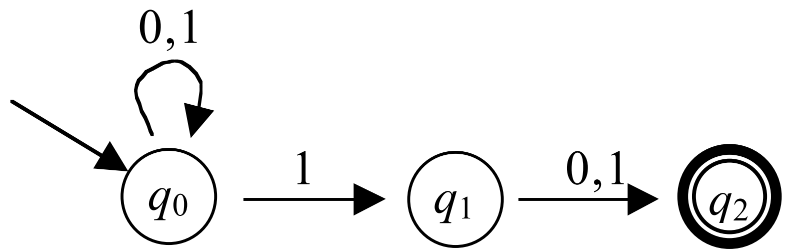
Outline

- 6.1 NFA Implemented With Backtracking Search
- 6.2 NFA Implemented With Bit-Mapped Parallel Search
- 6.3 The Subset Construction
- 6.4 NFAs Are Exactly As Powerful As DFAs
- 6.5 DFA Or NFA?

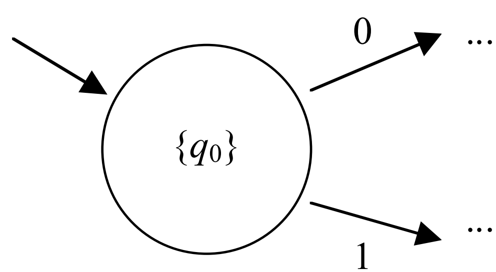
From NFA To DFA

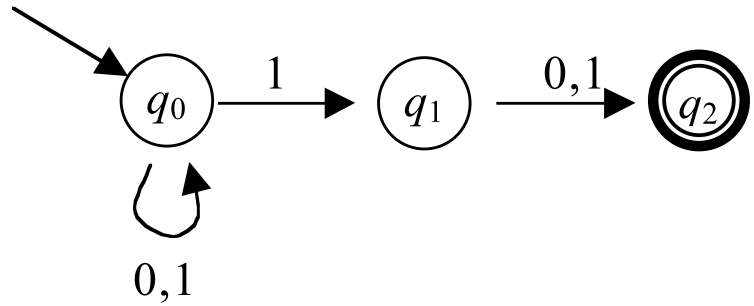
- For any NFA, there is a DFA that recognizes the same language
- Proof is by construction: a DFA that keeps track of the set of states the NFA might be in
- This is called the *subset construction*
- First, an example starting from this NFA:



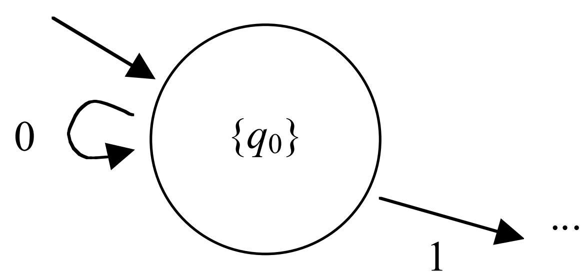


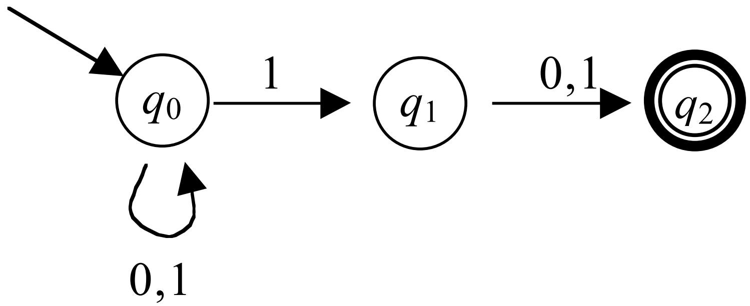
- Initially, the set of states the NFA could be in is just $\{q_0\}$
- So our DFA will keep track of that using a start state labeled $\{q_0\}$:



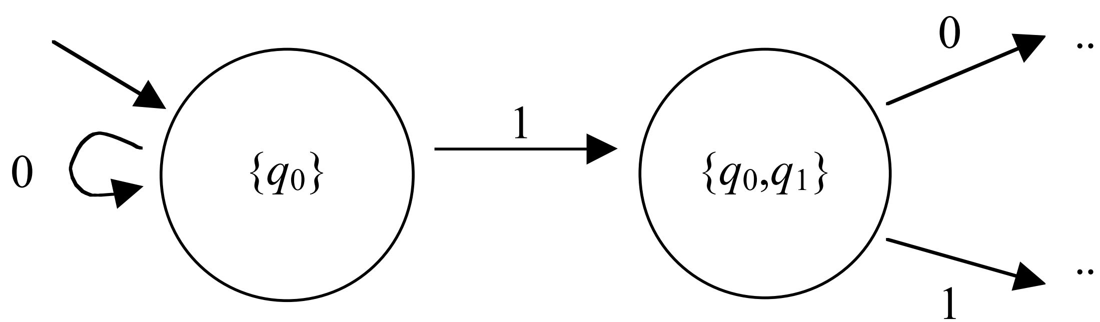


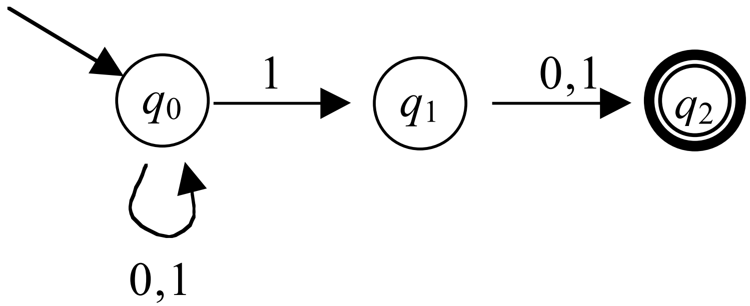
- Now suppose the set of states the NFA could be in is $\{q_0\}$, and it reads a 0
- The set of possible states after reading the 0 is $\{q_0\}$, so we can show that transition:



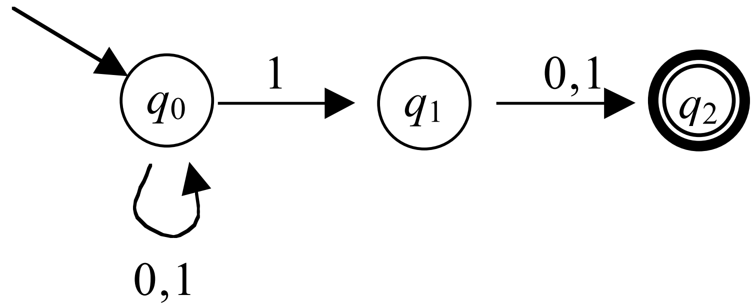


- Suppose the set of states the NFA could be in is $\{q_0\}$, and it reads a 1
- The set of possible states after reading the 1 is $\{q_0,q_1\}$, so we need another state:

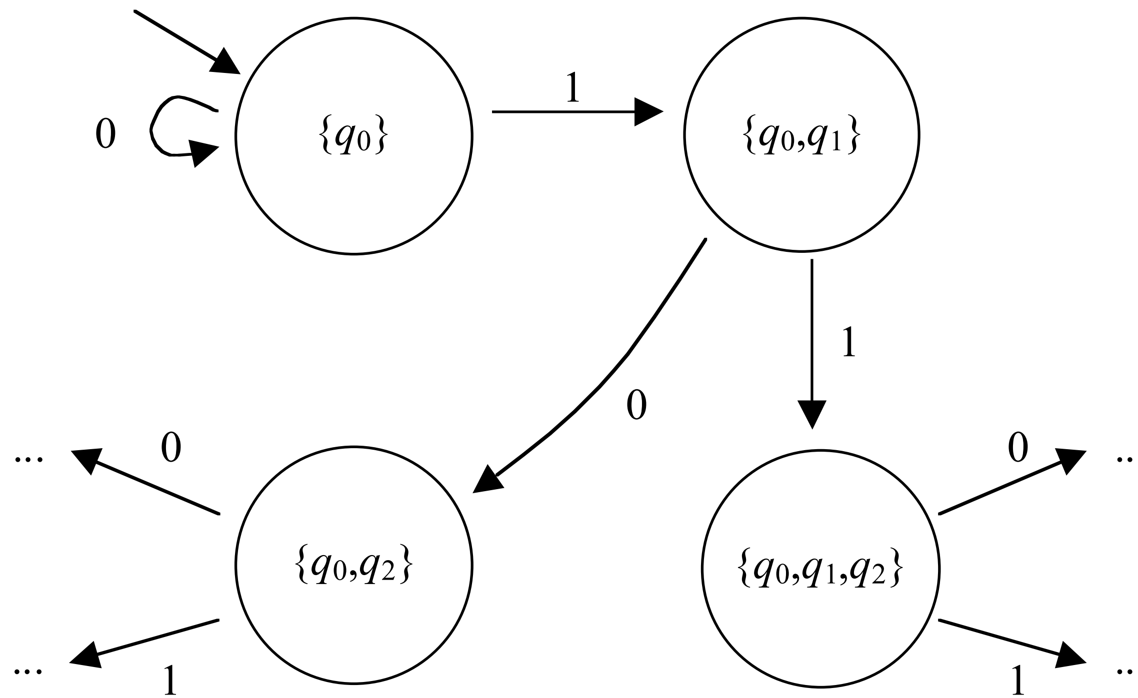




- From $\{q_0,q_1\}$ on a 0, the next set of possible states is $\delta(q_0,0) \cup \delta(q_1,0) = \{q_0,q_2\}$
- From $\{q_0,q_1\}$ on a 1, the next set of possible states is $\delta(q_0,1) \cup \delta(q_1,1) = \{q_0,q_1,q_2\}$
- Adding these transitions and states, we get…



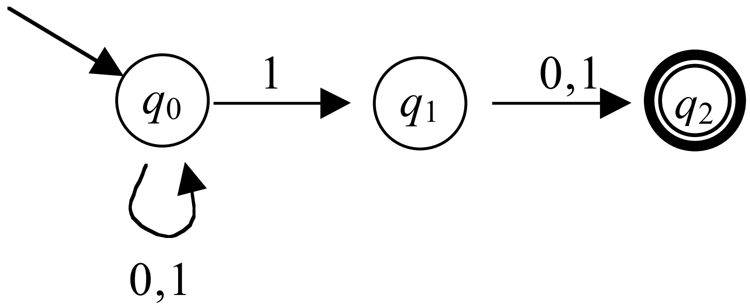
q_0
1
q_1
0,1
q_2
0,1



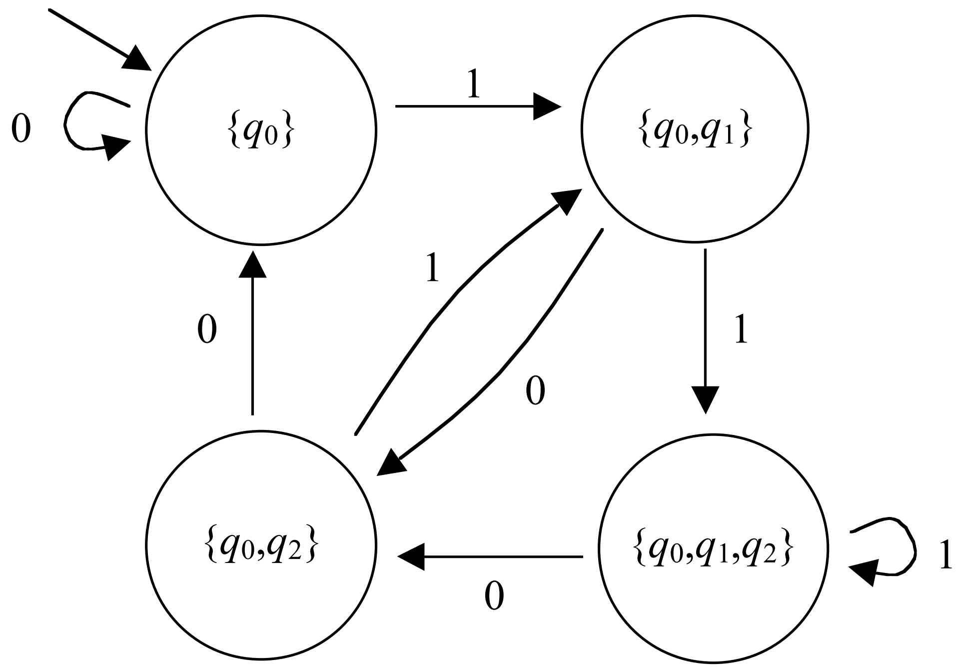
0
$\{q_0\}$
1
$\{q_0,q_1\}$
1
0
...
0
$\{q_0,q_2\}$
...
1
$\{q_0,q_1,q_2\}$
0
...
1
...

And So On

- The DFA construction continues
- Eventually, we find that no further states are generated
- That's because there are only finitely many possible sets of states: $P(Q)$
- In our example, we have already found all sets of states reachable from $\{q_0\}$…



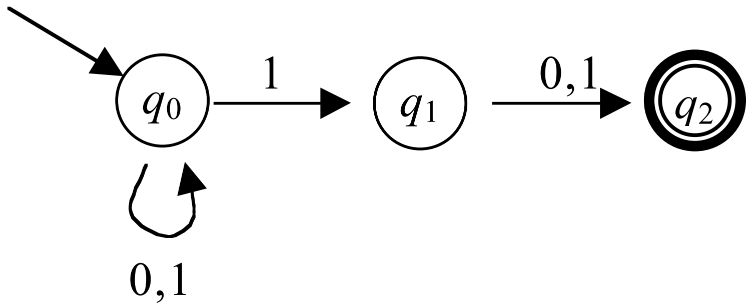
q_0
1
q_1
0,1
q_2
0,1



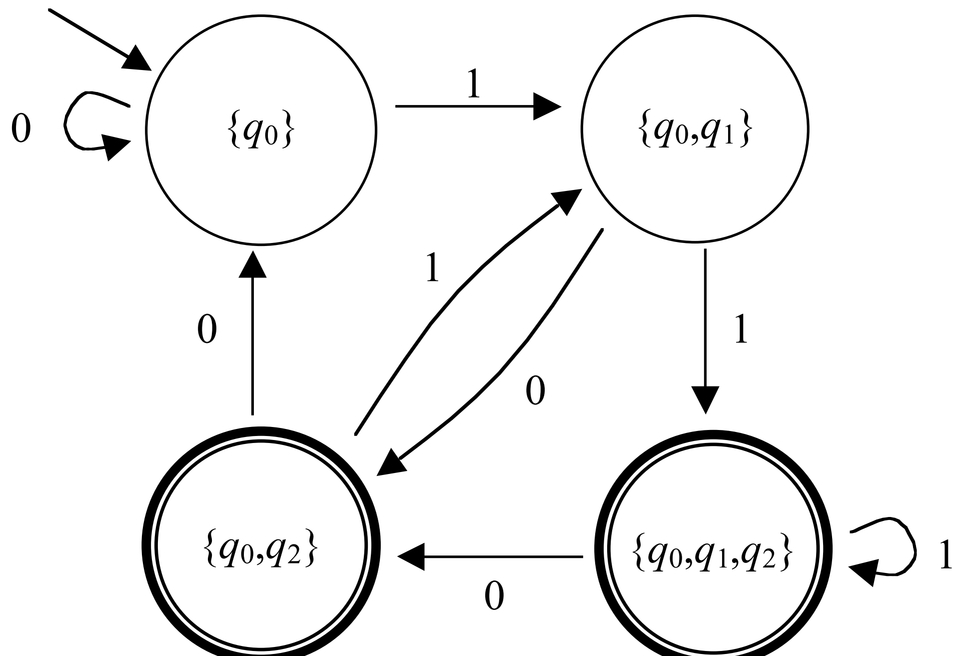
0
$\{q_0\}$
1
$\{q_0,q_1\}$
1
0
0
1
$\{q_0,q_2\}$
$\{q_0,q_1,q_2\}$
0
1

Accepting States

- It only remains to choose the accepting states
- An NFA accepts *x* if its set of possible states after reading *x* includes at least one accepting state
- So our DFA should accept in all sets that contain at least one NFA accepting state



q_0
1
q_1
0,1
q_2
0,1



0
$\{q_0\}$
1
$\{q_0,q_1\}$
1
0
1
0
$\{q_0,q_2\}$
$\{q_0,q_1,q_2\}$
1
0

Lemma 6.3

If $L = L(N)$ for some NFA N, then L is a regular language.

- Suppose L is $L(N)$ for some NFA $N = (Q_N, \Sigma, \delta_N, q_N, F_N)$
- Construct a new DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$, where:

$$Q_D = P(Q_N)$$

$$\delta_D(R,a) = \bigcup_{r \in R} \delta_N^*(r,a), \text{ for all } R \in Q_D \text{ and } a \in \Sigma$$

$$q_D = \delta_N^*(q_N, \varepsilon)$$

$$F_D = \{R \in Q_D \mid R \cap F_N \neq \{\}\}$$

Lemma 6.3, Proof Continued

- By construction we have, for all $x \in \Sigma^*$,

$$\delta_D^*(q_D, x) = \delta_N^*(q_N, x)$$

jump can be bridged
by routine induction

- D simulates N's behavior on each input x
- D accepts if and only if N accepts
- $L = L(N) = L(D)$
- L is a regular language

Implementation Note

- The subset construction defined the DFA transition function by

$$\delta_D(R,a) = \bigcup_{r \in R} \delta_N^*(r,a)$$

- This is exactly what `process` computed in its inner loop, in our bit-mapped implementation in Java
- So that implementation really just computes the subset construction, one step for each input symbol

Start State Note

- In the subset construction, the start state for the new DFA is

$$q_D = \delta_N^*(q_N, \varepsilon)$$

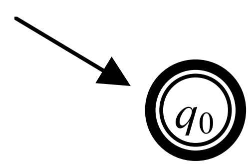
- Often this is the same as $q_D = \{q_N\}$, as in our earlier example
- But the difference is important if there are ε-transitions from the NFA's start state

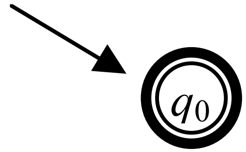
Unreachable State Note

- The formal subset construction generates all states $Q_D = P(Q_N)$
- These may not all be reachable from the DFA's start state
- In our earlier example, only 4 states were reachable, but $|P(Q_N)| = 8$
- Unreachable states don't affect $L(D)$
- When doing the construction by hand, it is usually better to include only the reachable states

Empty-Set State Note

- The empty set is a subset of every set
- So the full subset construction always produces a DFA state for {}
- This is reachable from the start state if there is some string x for which the NFA has no legal sequence of moves: $\delta_N^*(q_N,x) = \{\}$
- For example, this NFA, with $L(N) = \{\varepsilon\}$





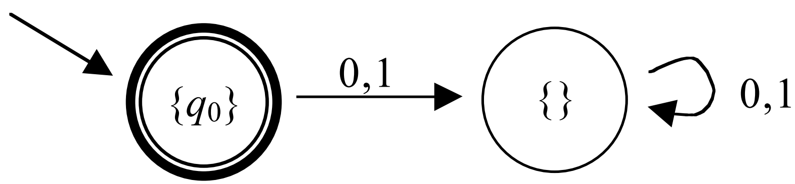
$$\delta_D\left(\{q_0\},0\right) = \bigcup_{r\in\{q_0\}} \delta_N^*\left(r,0\right) = \{\ \}$$

$$\delta_D\left(\{q_0\},1\right) = \bigcup_{r\in\{q_0\}} \delta_N^*\left(r,1\right) = \{\ \}$$

$$\delta_D\left(\{\ \},0\right) = \bigcup_{r\in\{\ \}} \delta_N^*\left(r,0\right) = \{\ \}$$

$$\delta_D\left(\{\ \},1\right) = \bigcup_{r\in\{\ \}} \delta_N^*\left(r,1\right) = \{\ \}$$

- $P(\{q_0\}) = \{\ \{\}, \{q_0\}\ \}$
- A 2-state DFA



Trap State Provided

- The subset construction always provides a state for {}
- And it is always the case that

$$\delta_D(\{\ \},a) = \bigcup_{r \in \{\ \}} \delta_N^*(r,a) = \{\ \}$$

so the {} state always has transitions back to itself for every symbol *a* in the alphabet

- It is a non-accepting trap state

Outline

- 6.1 NFA Implemented With Backtracking Search
- 6.2 NFA Implemented With Bit-Mapped Parallel Search
- 6.3 The Subset Construction
- 6.4 NFAs Are Exactly As Powerful As DFAs
- 6.5 DFA Or NFA?

From DFA To NFA

- This direction is much easier
- A DFA is like a special case of an NFA, with exactly one transition from every state on every symbol
- So it is easy to show that whenever there is a DFA M with $L(M) = L$ (so L is regular), there is an NFA N with $L(N) = L$
- There's just a little technicality involved in changing the type of the δ function

Lemma 6.4

If L is any regular language, there is some NFA N for which $L(N) = L$.

- Let L be any regular language
- By definition there must be some DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $L(M) = L$
- Define a new NFA $N = (Q, \Sigma, \delta', q_0, F)$, where $\delta'(q,a) = \{\delta(q,a)\}$ for all $q \in Q$ and $a \in \Sigma$, and $\delta'(q,\varepsilon) = \{\}$ for all $q \in Q$

jump can be bridged by routine induction

» Now $\delta'^*(q,x) = \{\delta^*(q,x)\}$, for all $q \in Q$ and $x \in \Sigma^*$

» Thus $L(N) = L(M) = L$

Theorem 6.4

A language L is $L(N)$ for some NFA N if and only if L is a regular language.

- Follows immediately from the previous lemmas
- Allowing nondeterminism in finite automata can make them more compact and easier to construct
- But in the sense of Theorem 6.4, it neither weakens nor strengthens them

DFA, Pro And Con

- Pro
  - Faster and simpler
- Con
  - There are languages for which DFA-based implementation takes exponentially more space than NFA-based
  - Harder to extend for non-regular constructs
- Example: scanner in a compiler
  - Speed is critical
  - Token languages do not usually bring out the exponential-size pathology of DFAs

NFA, Pro And Con

- Pro
  - Easier to extend for non-regular language constructs
  - No exponential-space pathologies
- Con
  - Slower and trickier
- Example: regular-expression programming language features (Perl, Python, Ruby, etc.)
  - Need to handle non-regular constructs as well as regular ones
  - More about these when we look at regular expressions

Hybrids

- Some applications use both techniques in combination
  - Use DFA techniques for purely regular parts, but switch to NFA techniques when non-regular extensions are needed
  - Use NFA techniques but cache frequently-used state sets and transitions
- A spectrum of techniques, not just two points