Finite Automata

Deterministic, Nondeterministic, and State Minimization

Computer Science Theory

Slide Set 2

Table of Contents

Part 1: Intuitive Introduction to DFAs

Self-Study Header: The Automatic Door Controller

Finite automata are excellent models for computers with an extremely limited amount of memory[cite: 784].

Part 1: Intuitive Introduction to DFAs

Self-Study Header: The Automatic Door Controller

Finite automata are excellent models for computers with an extremely limited amount of memory[cite: 784].

Example: Automatic Door Controller

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door[cite: 785]. There are four possible input conditions: "FRONT", "REAR", "BOTH", and "NEITHER"[cite: 785].

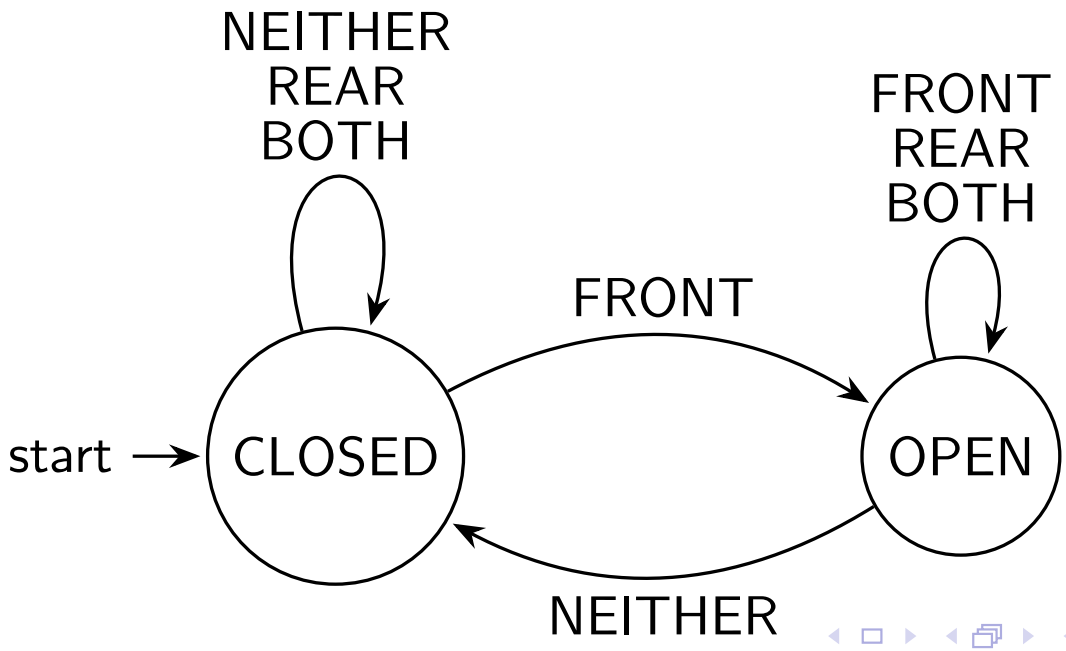
Part 1: Intuitive Introduction to DFAs

Self-Study Header: The Automatic Door Controller

Finite automata are excellent models for computers with an extremely limited amount of memory[cite: 784].

Example: Automatic Door Controller

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door[cite: 785]. There are four possible input conditions: "FRONT", "REAR", "BOTH", and "NEITHER"[cite: 785].



Formal Definition of a DFA

To resolve any ambiguities, we require a mathematically precise definition.

Formal Definition of a DFA

To resolve any ambiguities, we require a mathematically precise definition.

The 5-Tuple Definition

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where[cite: 788]:

1. Q is a finite set called the states[cite: 788],
2. Σ is a finite set called the alphabet[cite: 788],
3. $\delta : Q \times \Sigma \to Q$ is the transition function[cite: 788],
4. $q_0 \in Q$ is the start state[cite: 788], and
5. $F \subseteq Q$ is the set of accept states[cite: 788].

Formal Definition of a DFA

To resolve any ambiguities, we require a mathematically precise definition.

The 5-Tuple Definition

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where[cite: 788]:

1. Q is a finite set called the states[cite: 788],
2. Σ is a finite set called the alphabet[cite: 788],
3. $\delta : Q \times \Sigma \to Q$ is the transition function[cite: 788],
4. $q_0 \in Q$ is the start state[cite: 788], and
5. $F \subseteq Q$ is the set of accept states[cite: 788].

Crucial Detail: δ is a *total function*. For every state and every symbol in the alphabet, exactly one transition must be defined[cite: 788].

Extended Transition Function

The transition function δ strictly handles single symbols. To describe the processing of entire strings, it is convenient to introduce the extended transition function $\delta^* : Q \times \Sigma^* \to Q$[cite: 307].

Extended Transition Function

The transition function δ strictly handles single symbols. To describe the processing of entire strings, it is convenient to introduce the extended transition function $\delta^* : Q \times \Sigma^* \to Q$[cite: 307].

Recursive Definition of δ^*

We define δ^* recursively as follows:

1. $\delta^*(q, \lambda) = q$ [cite: 307] (where λ or ε is the empty string).
2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ [cite: 307] for all $w \in \Sigma^*$ and $a \in \Sigma$.

Extended Transition Function

The transition function δ strictly handles single symbols. To describe the processing of entire strings, it is convenient to introduce the extended transition function $\delta^* : Q \times \Sigma^* \to Q$[cite: 307].

Recursive Definition of δ^*

We define δ^* recursively as follows:

1. $\delta^*(q, \lambda) = q$ [cite: 307] (where λ or ε is the empty string).
2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ [cite: 307] for all $w \in \Sigma^*$ and $a \in \Sigma$.

Language Recognition: A language is called a regular language if some finite automaton recognizes it[cite: 793]. Formally, the language recognized by a DFA M is $L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$[cite: 308].

Designing DFAs: Case Study 1

Goal: Construct a DFA accepting all strings over $\{0, 1\}$ with an **odd number of 1s**[cite: 795].

Designing DFAs: Case Study 1

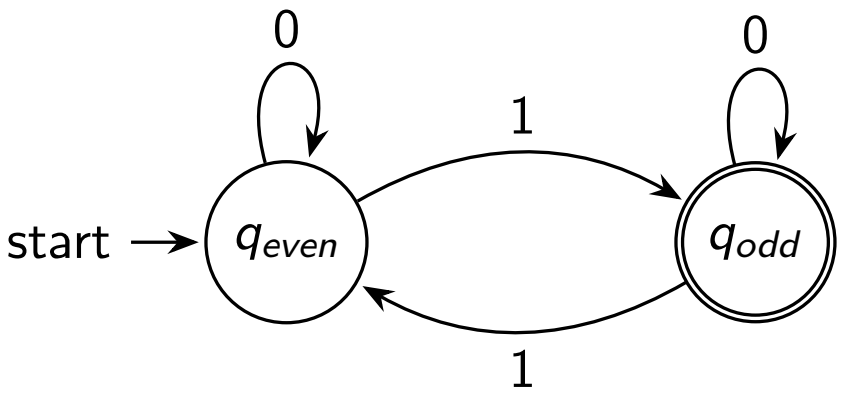
Goal: Construct a DFA accepting all strings over $\{0, 1\}$ with an **odd number of 1s**[cite: 795].

Thought Process: We don't need to remember the entire string. We only need to remember whether the number of 1s seen so far is even or odd[cite: 795, 796].

Designing DFAs: Case Study 1

Goal: Construct a DFA accepting all strings over $\{0, 1\}$ with an **odd number of 1s**[cite: 795].

Thought Process: We don't need to remember the entire string. We only need to remember whether the number of 1s seen so far is even or odd[cite: 795, 796].



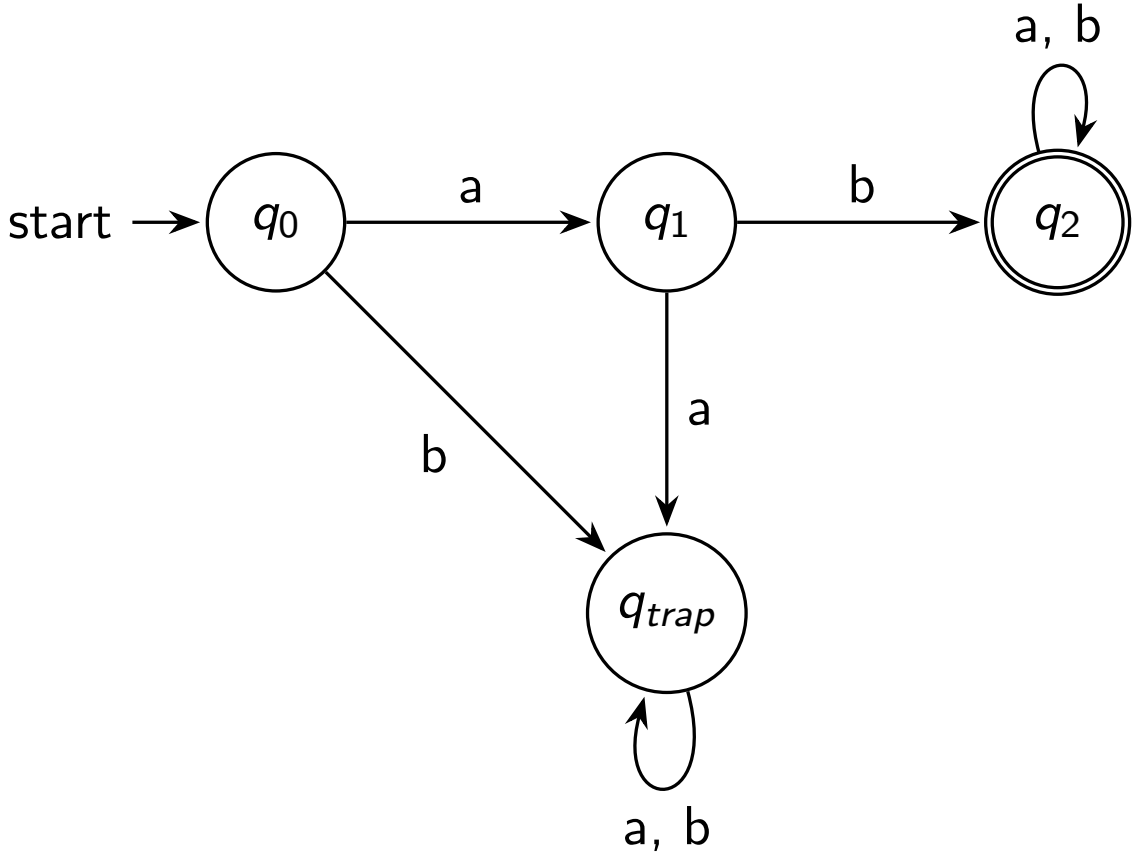
- **States:** $Q = \{q_{even}, q_{odd}\}$.
- **Start State:** q_{even} (0 is an even number).
- **Accept State:** $F = \{q_{odd}\}$.

Designing DFAs: Case Study 2 & Trap States

Goal: Construct a DFA over $\{a, b\}$ accepting strings starting with the prefix '**ab**'.

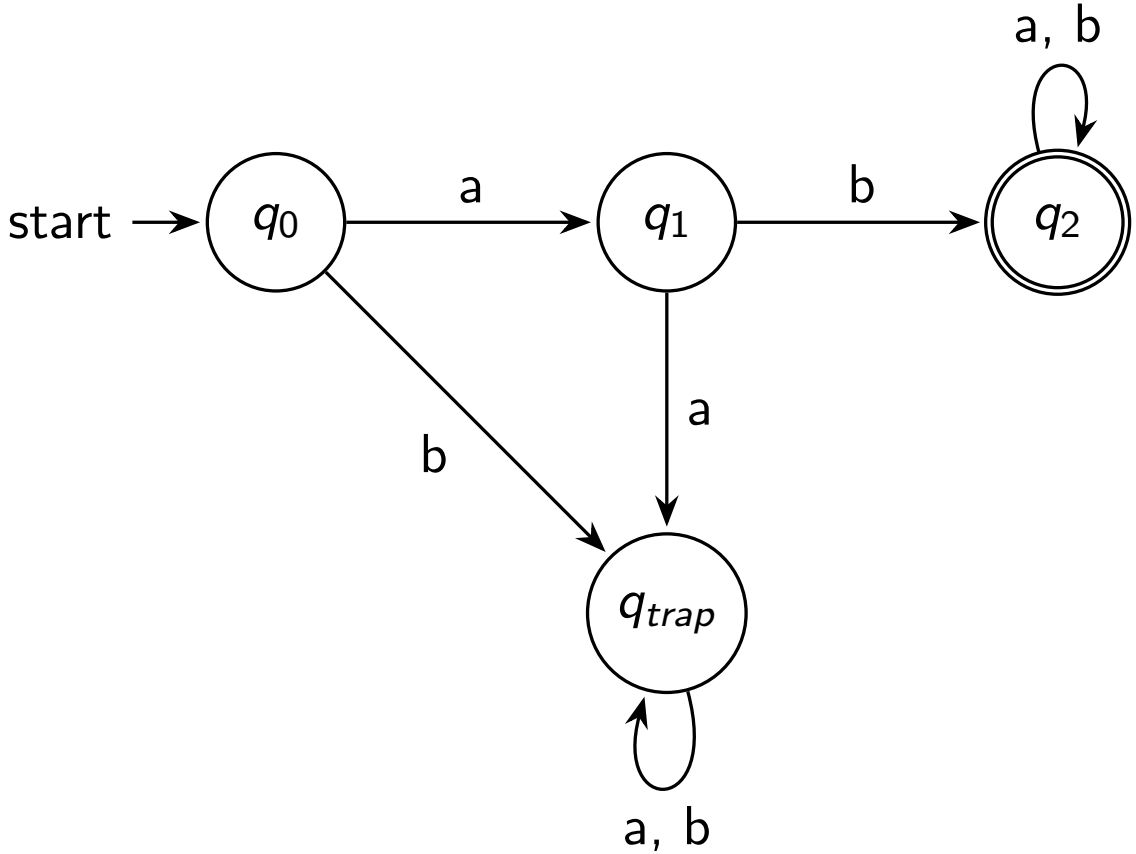
Designing DFAs: Case Study 2 & Trap States

Goal: Construct a DFA over $\{a, b\}$ accepting strings starting with the prefix '**ab**'.



Designing DFAs: Case Study 2 & Trap States

Goal: Construct a DFA over $\{a, b\}$ accepting strings starting with the prefix '**ab**'.



Teacher's Corner: The Trap State

Students often draw partial machines that stop if the wrong input is read. However, δ must be a total function! If an invalid sequence is encountered, the

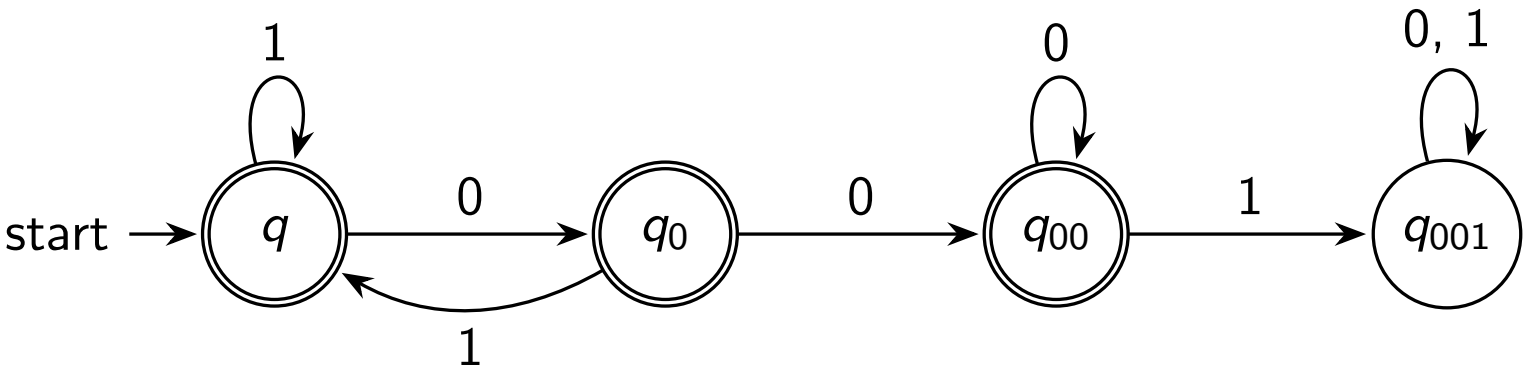
Designing DFAs: Case Study 3

Goal: Construct a DFA over $\{0, 1\}$ recognizing strings that **do not contain** the substring '001'.

Designing DFAs: Case Study 3

Goal: Construct a DFA over $\{0, 1\}$ recognizing strings that **do not contain** the substring '001'.

Strategy: First, build a machine that *does* recognize strings containing '001'[cite: 796]. Then, simply flip the accept and non-accept states!



By making the target state q_{001} a non-accept (trap) state, and all intermediate progress states as accepting, we accept everything until the forbidden substring arrives.

Part 3: Introduction to Nondeterminism

Self-Study Header: The Power of Choice

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton[cite: 800]. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol[cite: 801].

Part 3: Introduction to Nondeterminism

Self-Study Header: The Power of Choice

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton[cite: 800]. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol[cite: 801].

- **Parallel Computation:** If multiple choices exist, the machine "splits" into multiple copies, exploring all paths in parallel[cite: 801].
- **Acceptance:** The machine accepts if *at least one* branch reaches an accept state at the end of the input[cite: 801].
- **Empty Transitions:** An NFA can feature arrows labeled with ε (or λ), allowing it to branch into multiple states without consuming an input symbol[cite: 801].

Formal Definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The crucial difference lies in the transition function.

Formal Definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The crucial difference lies in the transition function.

The Transition Function δ

Let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. The transition function takes a state and an input symbol or the empty string and produces the **set** of possible next states[cite: 806].

$$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$$

Formal Definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The crucial difference lies in the transition function.

The Transition Function δ

Let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. The transition function takes a state and an input symbol or the empty string and produces the **set** of possible next states[cite: 806].

$$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$$

Here, $\mathcal{P}(Q)$ (or 2^Q) is the power set of Q. Since the output is a set of states:

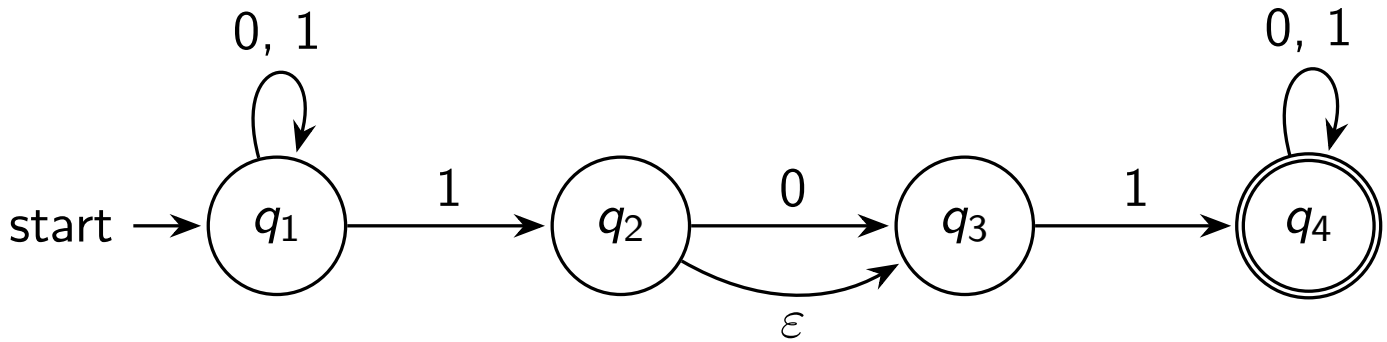
- It can return a single state: $\{q_1\}$
- It can return multiple states: $\{q_1, q_2\}$
- It can return the empty set: $\emptyset$ (the computational branch "dies" [cite: 801]).

NFA Example

Goal: Accept strings containing either '101' or '11' as a substring[cite: 803].

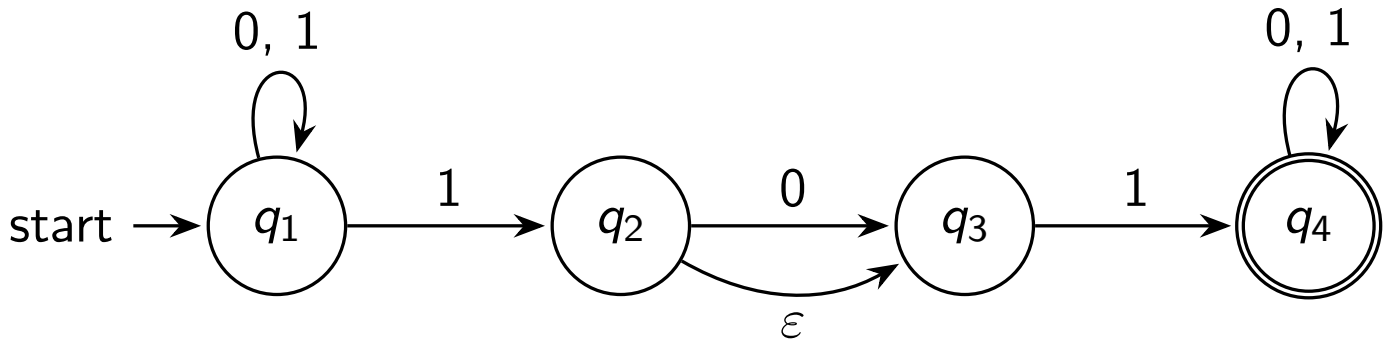
NFA Example

Goal: Accept strings containing either '101' or '11' as a substring[cite: 803].



NFA Example

Goal: Accept strings containing either '101' or '11' as a substring[cite: 803].



Why is this an NFA?

1. From q_1, reading a '1' leads to either q_1 or q_2.
2. From q_2, an ε-transition jumps to q_3 without consuming input.
3. From q_2, there is no valid path for the input '1', causing that branch to die.

Part 4: Equivalence Theorem

Theorem: Every nondeterministic finite automaton has an equivalent deterministic finite automaton[cite: 808].

Part 4: Equivalence Theorem

Theorem: Every nondeterministic finite automaton has an equivalent deterministic finite automaton[cite: 808].

This is a surprising but highly useful result. It tells us that adding nondeterminism *does not increase* the computational power of a finite automaton—they both recognize exactly the class of regular languages.

Part 4: Equivalence Theorem

Theorem: Every nondeterministic finite automaton has an equivalent deterministic finite automaton[cite: 808].

This is a surprising but highly useful result. It tells us that adding nondeterminism *does not increase* the computational power of a finite automaton—they both recognize exactly the class of regular languages.

Proof Idea: The Subset Construction Algorithm

If the NFA has k states, we construct a DFA whose states represent subsets of the NFA's states. Every state of M (the DFA) is a set of states of N (the NFA)[cite: 808]. The DFA simulates all parallel branches of the NFA simultaneously.

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

- **States (Q'):** $Q' = \mathcal{P}(Q)$. The DFA tracks the *set* of states the NFA could currently be in.

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

- **States (Q'):** $Q' = \mathcal{P}(Q)$. The DFA tracks the *set* of states the NFA could currently be in.
- **ε-Closure ($E(R)$):** For any set of states R, $E(R)$ is the collection of states reachable from R by going along zero or more ε arrows[cite: 809].

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

- **States (Q'):** $Q' = \mathcal{P}(Q)$. The DFA tracks the *set* of states the NFA could currently be in.
- **ε-Closure ($E(R)$):** For any set of states R, $E(R)$ is the collection of states reachable from R by going along zero or more ε arrows[cite: 809].
- **Start State (q_0'):** $q_0' = E(\{q_0\})$. We begin by calculating the ε-closure of the NFA's start state.

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

- **States (Q'):** $Q' = \mathcal{P}(Q)$. The DFA tracks the *set* of states the NFA could currently be in.
- **ε-Closure ($E(R)$):** For any set of states R, $E(R)$ is the collection of states reachable from R by going along zero or more ε arrows[cite: 809].
- **Start State (q_0'):** $q_0' = E(\{q_0\})$. We begin by calculating the ε-closure of the NFA's start state.
- **Transition Function (δ_M):** For a state $R \in Q'$ and symbol $a \in \Sigma$:

$$\delta_M(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$$

The Subset Construction Algorithm

Let $N = (Q, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We build an equivalent DFA $M = (Q', \Sigma, \delta_M, q_0', F')$.

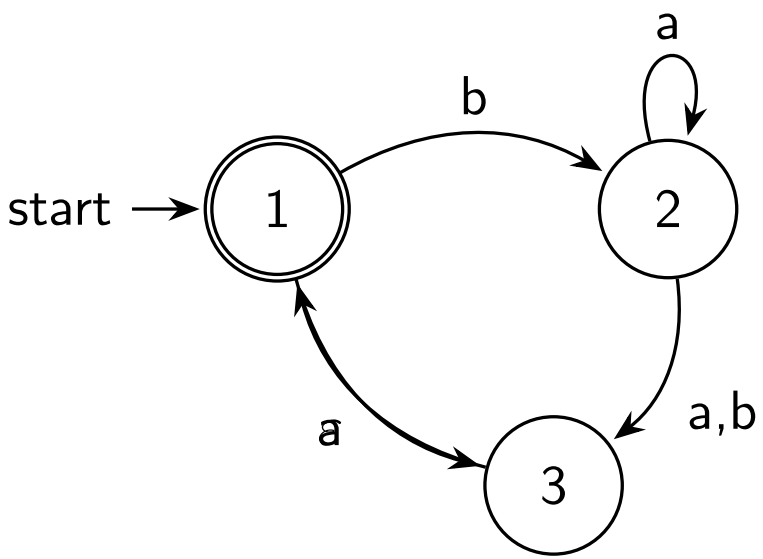
- **States (Q'):** $Q' = \mathcal{P}(Q)$. The DFA tracks the *set* of states the NFA could currently be in.
- **ε-Closure ($E(R)$):** For any set of states R, $E(R)$ is the collection of states reachable from R by going along zero or more ε arrows[cite: 809].
- **Start State (q_0'):** $q_0' = E(\{q_0\})$. We begin by calculating the ε-closure of the NFA's start state.
- **Transition Function (δ_M):** For a state $R \in Q'$ and symbol $a \in \Sigma$:

$$\delta_M(R, a) = \bigcup_{r \in R} E(\delta_N(r, a))$$

- **Accept States (F'):** Any subset $R \in Q'$ that contains at least one accept state of N.

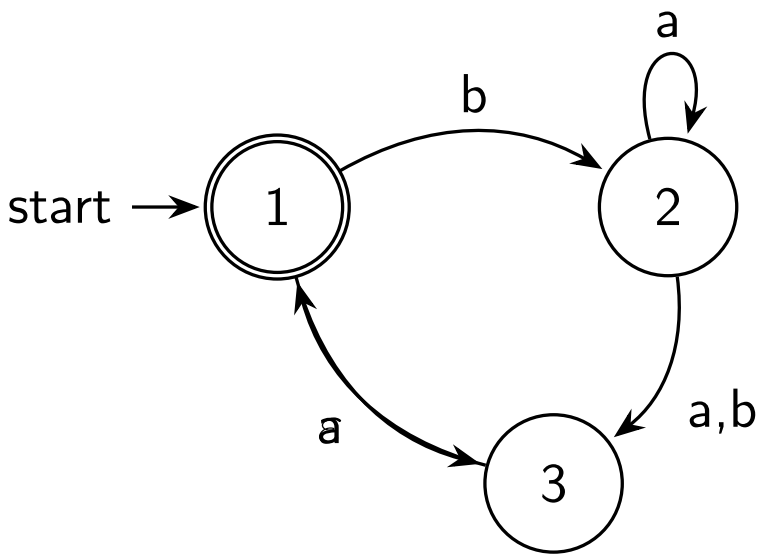
Worked Example: NFA to DFA

Consider the following NFA N_4 (states 1, 2, 3)[cite: 809]:



Worked Example: NFA to DFA

Consider the following NFA N_4 (states 1, 2, 3)[cite: 809]:

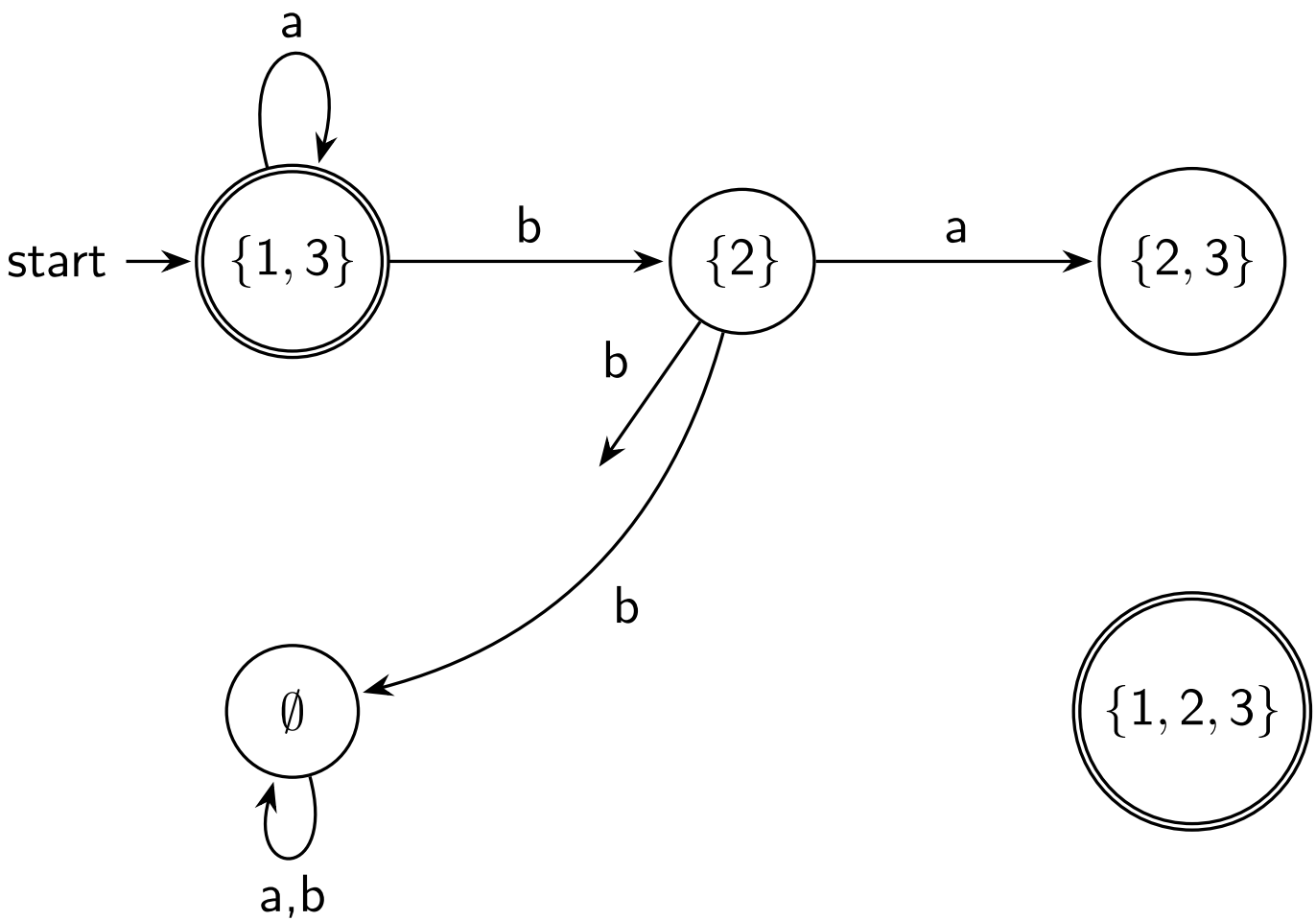


Let's trace the Subset Construction:

- **Start State:** $E(\{1\}) = \{1, 3\}$ (since $1 \xrightarrow{\varepsilon} 3$).
- **From $\{1, 3\}$ on 'a':** $\delta_N(1, a) = \emptyset$, $\delta_N(3, a) = \{1\}$. Closure $E(\{1\}) = \{1, 3\}$. So, $\delta_M(\{1, 3\}, a) = \{1, 3\}$.
- **From $\{1, 3\}$ on 'b':** $\delta_N(1, b) = \{2\}$, $\delta_N(3, b) = \emptyset$. Closure $E(\{2\}) = \{2\}$. So, $\delta_M(\{1, 3\}, b) = \{2\}$.

Worked Example: NFA to DFA (Completed)

Continuing the process from the previous slide, we generate all reachable subsets:



(Diagram abbreviated for space. Note that many of the $2^3 = 8$ possible states are simply unreachable from the start state, minimizing our conversion effort!) [cite: 810]

Part 5: Motivation for State Reduction

Self-Study Header: Finding the Smallest Equivalent DFA

For a given regular language, there are infinitely many DFAs that recognize it. For practical applications (like compiling and text searching), we want the **minimal** DFA—the one with the fewest possible states.

Part 5: Motivation for State Reduction

Self-Study Header: Finding the Smallest Equivalent DFA

For a given regular language, there are infinitely many DFAs that recognize it. For practical applications (like compiling and text searching), we want the **minimal** DFA—the one with the fewest possible states.

Indistinguishable States

Two states p and q of a dfa are called indistinguishable if ... $\delta^*(p, w) \in F$ implies $\delta^*(q, w) \in F$ ... and vice versa for all $w \in \Sigma^*$[cite: 330].

Part 5: Motivation for State Reduction

Self-Study Header: Finding the Smallest Equivalent DFA

For a given regular language, there are infinitely many DFAs that recognize it. For practical applications (like compiling and text searching), we want the **minimal** DFA—the one with the fewest possible states.

Indistinguishable States

Two states p and q of a dfa are called indistinguishable if ... $\delta^*(p, w) \in F$ implies $\delta^*(q, w) \in F$ ... and vice versa for all $w \in \Sigma^*$[cite: 330].

If two states are indistinguishable, they represent redundant memory. We can merge them into a single state without changing the language recognized by the DFA.

The "Mark" Algorithm

The procedure *mark* can be implemented by partitioning the states into equivalence classes[cite: 332].

1. **Initialize:** Remove all unreachable states.

The "Mark" Algorithm

The procedure *mark* can be implemented by partitioning the states into equivalence classes[cite: 332].

1. **Initialize:** Remove all unreachable states.
2. **Base Partition:** Draw a table of all pairs of states (p, q). Mark pairs where $p \in F$ and $q \notin F$ (or vice versa) as **distinguishable**.

The "Mark" Algorithm

The procedure *mark* can be implemented by partitioning the states into equivalence classes[cite: 332].

1. **Initialize:** Remove all unreachable states.
2. **Base Partition:** Draw a table of all pairs of states (p, q). Mark pairs where $p \in F$ and $q \notin F$ (or vice versa) as **distinguishable**.
3. **Iterate:** For every unmarked pair (p, q) and every symbol $a \in \Sigma$:
   - Compute $r = \delta(p, a)$ and $s = \delta(q, a)$.
   - If (r, s) is marked, then mark (p, q).

The "Mark" Algorithm

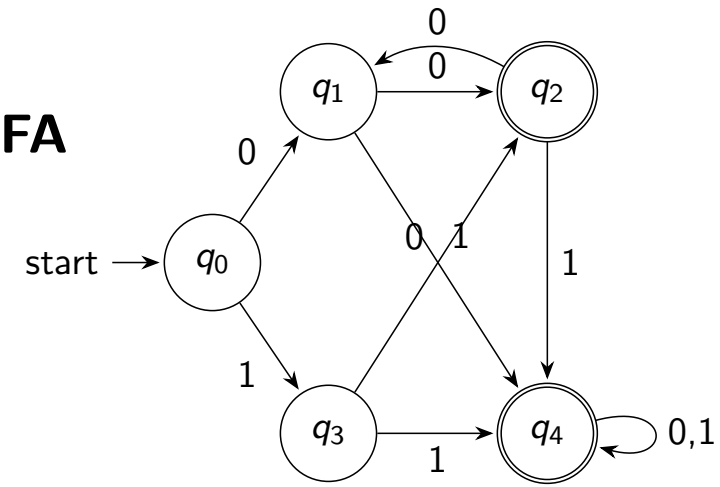
The procedure *mark* can be implemented by partitioning the states into equivalence classes[cite: 332].

1. **Initialize:** Remove all unreachable states.
2. **Base Partition:** Draw a table of all pairs of states (p, q). Mark pairs where $p \in F$ and $q \notin F$ (or vice versa) as **distinguishable**.
3. **Iterate:** For every unmarked pair (p, q) and every symbol $a \in \Sigma$:
   - Compute $r = \delta(p, a)$ and $s = \delta(q, a)$.
   - If (r, s) is marked, then mark (p, q).
4. **Terminate:** Repeat step 3 until a full pass makes no new marks.
5. **Merge:** Unmarked pairs are equivalent. Merge them into a single state.

Minimization Example

Consider a DFA with equivalent states[cite: 332, 333].

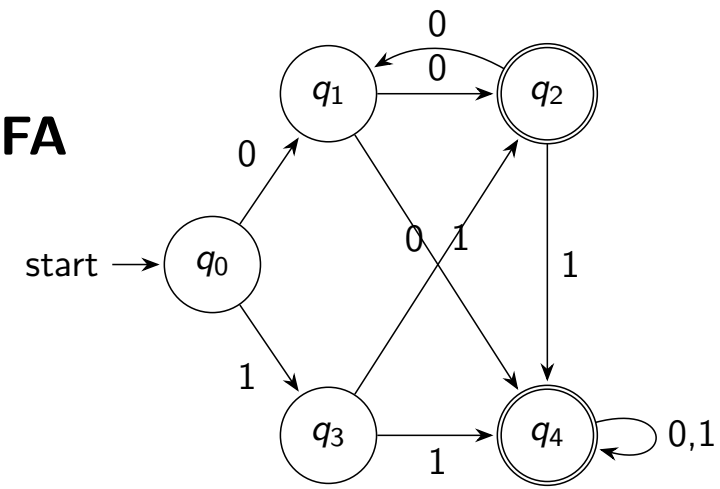
Original DFA



Minimization Example

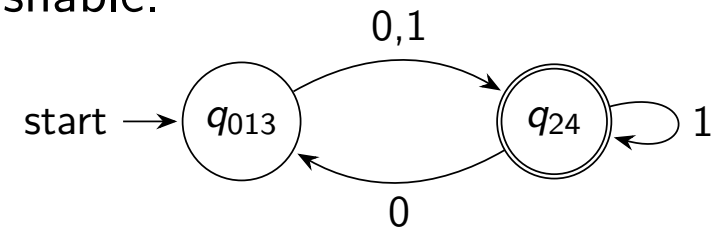
Consider a DFA with equivalent states[cite: 332, 333].

Original DFA



Minimal DFA

After running the marking algorithm, we find that $\{q_0, q_1, q_3\}$ are indistinguishable, and $\{q_2, q_4\}$ are indistinguishable.



Checkpoint Exercises

1. **DFA Design:** Draw the state diagram of a DFA over the alphabet $\Sigma = \{0, 1\}$ that recognizes the language: $\{w \mid w \text{ contains at least two 0s}\}$.

2. **NFA Trace:** Consider an NFA with states $\{q_0, q_1, q_2\}$. The start state is q_0.
   - $\delta(q_0, \varepsilon) = \{q_1\}$
   - $\delta(q_1, \varepsilon) = \{q_2\}$
   - $\delta(q_0, a) = \{q_0\}$

   What is the ε-closure of the start state q_0? That is, find $E(\{q_0\})$.

3. **Minimization Logic:** If a DFA has one accept state and one non-accept state, can it ever be minimized further? Why or why not?

Hints for Checkpoint Exercises

Hint for Exercise 1 (DFA Design):

- Think of the states as the memory of what you have seen so far.
- You need to remember: "Seen zero 0s", "Seen one 0", and "Seen two or more 0s". These become your three states.
- Once you hit "Seen two or more 0s", what happens if you see more 0s or 1s? (It should be an accepting trap state).

Hint for Exercise 2 (ε-closure):

- A state is always in its own ε-closure.
- The closure is transitive! If you can reach q_1 from q_0, and q_2 from q_1 via ε, you can reach q_2 from q_0 without consuming input.

Hint for Exercise 3 (Minimization):

- The first step of the marking algorithm is to mark all pairs (p, q) where one is final and the other is not. Are there any unmarked pairs left to potentially merge?