Non Deterministic Automata

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

March 23, 2021

[1] Mostly based on Nicholas Mainardi's material.

ND-FSA

- A Non Deterministic FSA is formally defined as a quintuple $(\mathbf{Q}, \mathbf{I}, \delta, q_0, \mathbf{F})$, where:
  - $\mathbf{Q}$, is the set of states of the automata
  - $\mathbf{I}$ is the alphabet of the input string which will be checked
  - $\delta : \mathbf{Q} \times \mathbf{I} \mapsto \wp(\mathbf{Q})$ the transition function
  - $q_0 \in \mathbf{Q}$ the (unique) initial state from where the automaton starts
  - $\mathbf{F} \subseteq \mathbf{Q}$ the set of final accepting states of the automaton
- The transitive closure δ^* is defined as:
  $\delta^*(q, \epsilon) = \{q\}, \delta^*(q, y.i) = \bigcup_{q' \in \delta^*(q,y)} \delta(q', i)$
- The string x is accepted $\iff$ $\delta^*(q_0, x) \cap F \neq \emptyset$

FSA determinization

It is always possible to make a ND-FSA deterministic! $\Longrightarrow$ FSA and ND-FSA are equivalent! Indeed, using a subscript D to mark the elements of the FSA and the subscript N to mark the elements of the ND-FSA, we can derive the former from the latter with the following construction:

- $\mathbf{Q_D} = \wp(\mathbf{Q_N})$
- $\delta_D(q_D, i) = \bigcup_{q_N \in \mathbf{q_D}} \delta_N(q_N, i)$
- $\mathbf{F_D} = \{q_d \mid \mathbf{F_N} \cap q_d \neq \emptyset\}$

The idea is that since ND-FSA goes to a set of states for the same input character, we can consider this set as a new state and the automaton becomes deterministic. The cost is a potential exponential factor in the number of states, since $\wp(\mathbf{Q}) = 2^{|\mathbf{Q}|}$

The Purpose of ND-FSA

A legitimate question: if ND-FSA and FSA are equivalent, why do we care about ND-FSA?

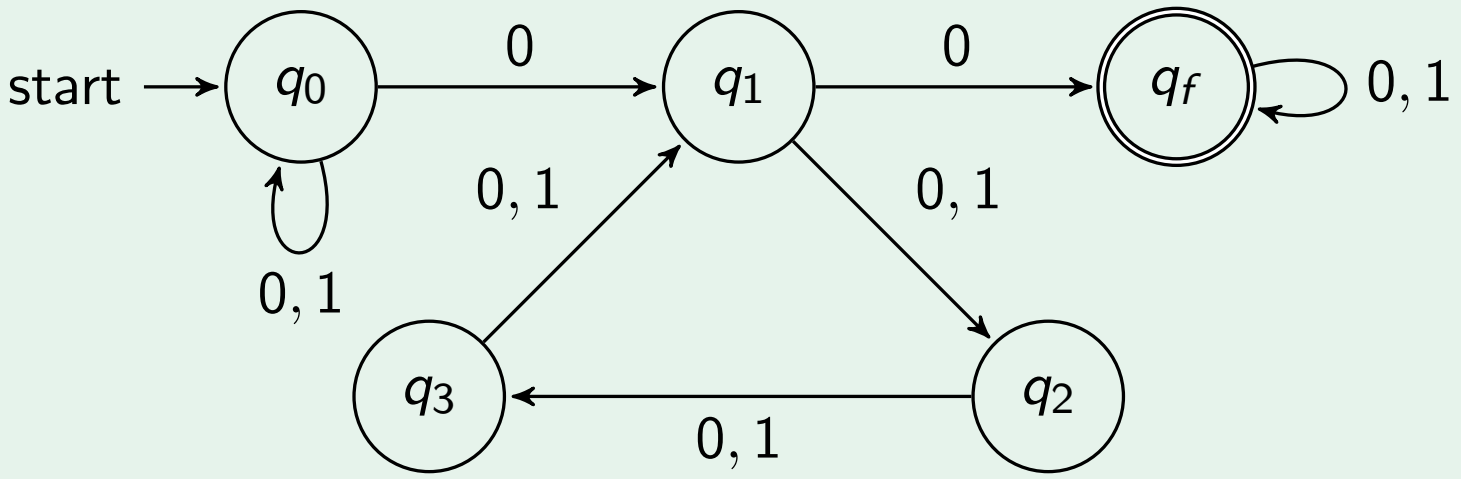
$$\Downarrow$$

Because they can be way easier to design for some languages!
Therefore, we can employ this additional strategy to build a FSA for a language:

1. Design a ND-FSA able to recognize it
2. Derive a deterministic FSA using the algorithm we have just seen

FSA Determinization Example

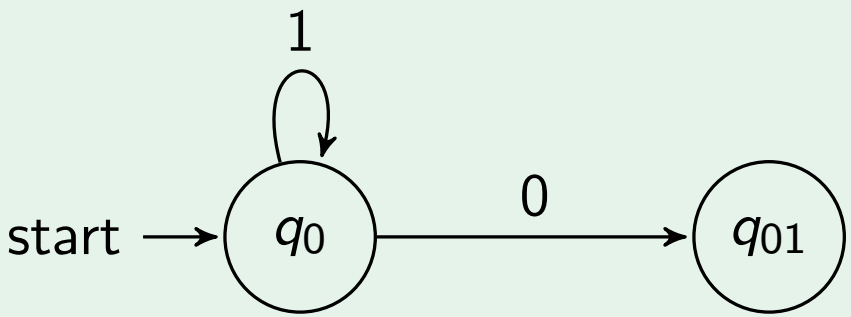
ND-FSA for $L = (0 \mid 1)^*0(0|1)^{3n}0(0 \mid 1)^* \mid n \geq 0$



- Deterministic version? Difficult to be designed from scratch
- Let's try the determinization algorithm ...

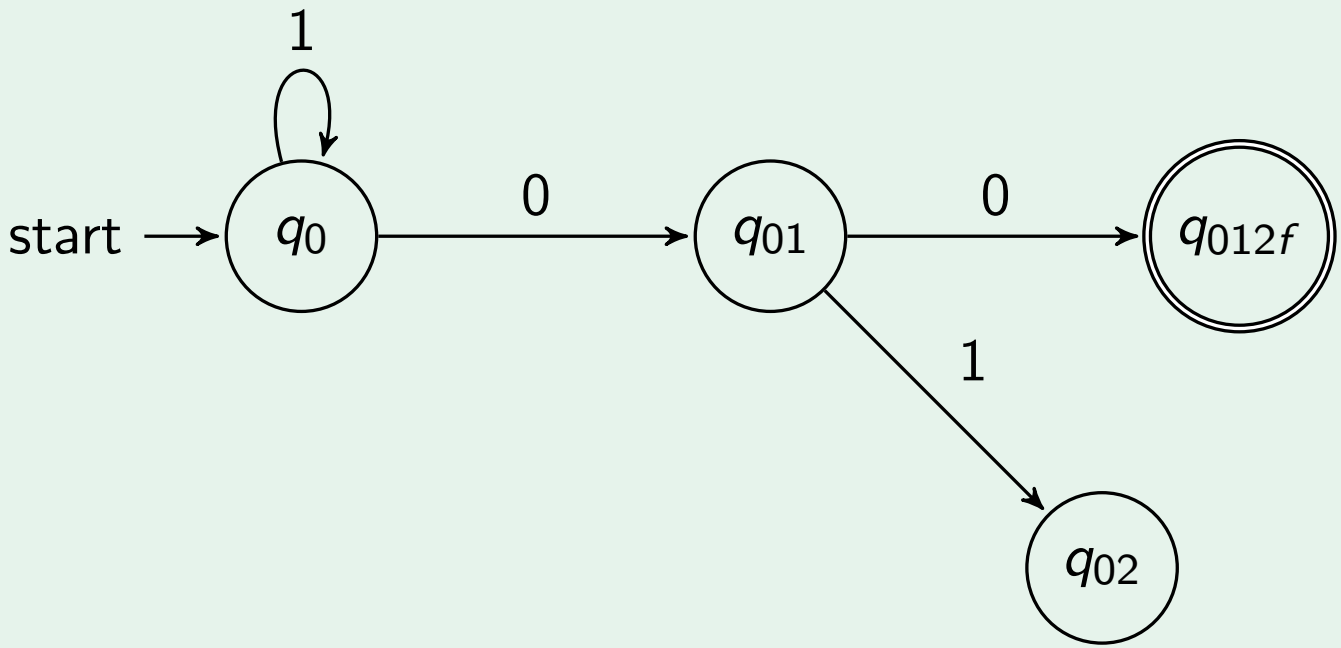
FSA Determinization Example

Expanding q_0



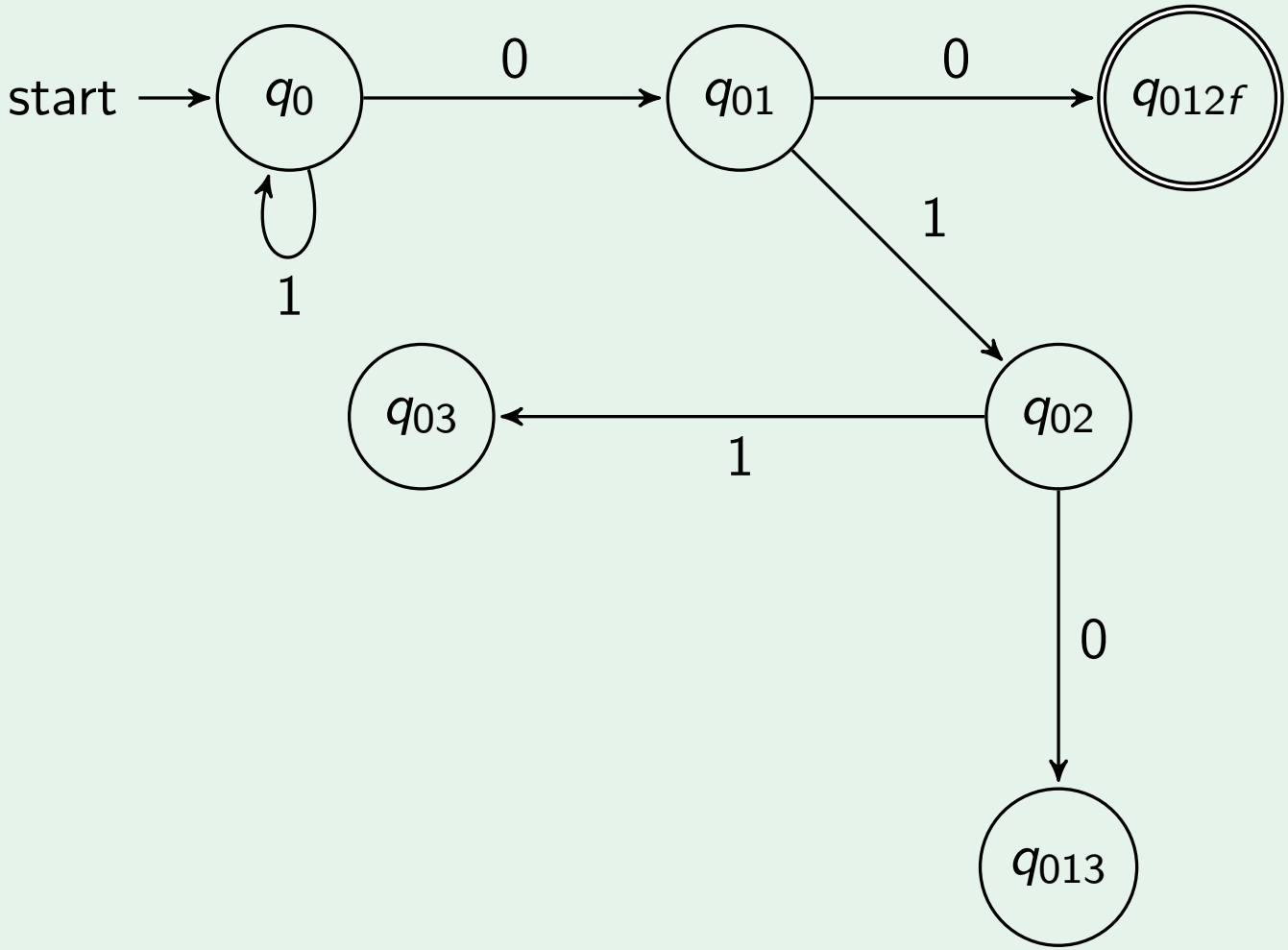
FSA Determinization Example

Expanding q_{01}



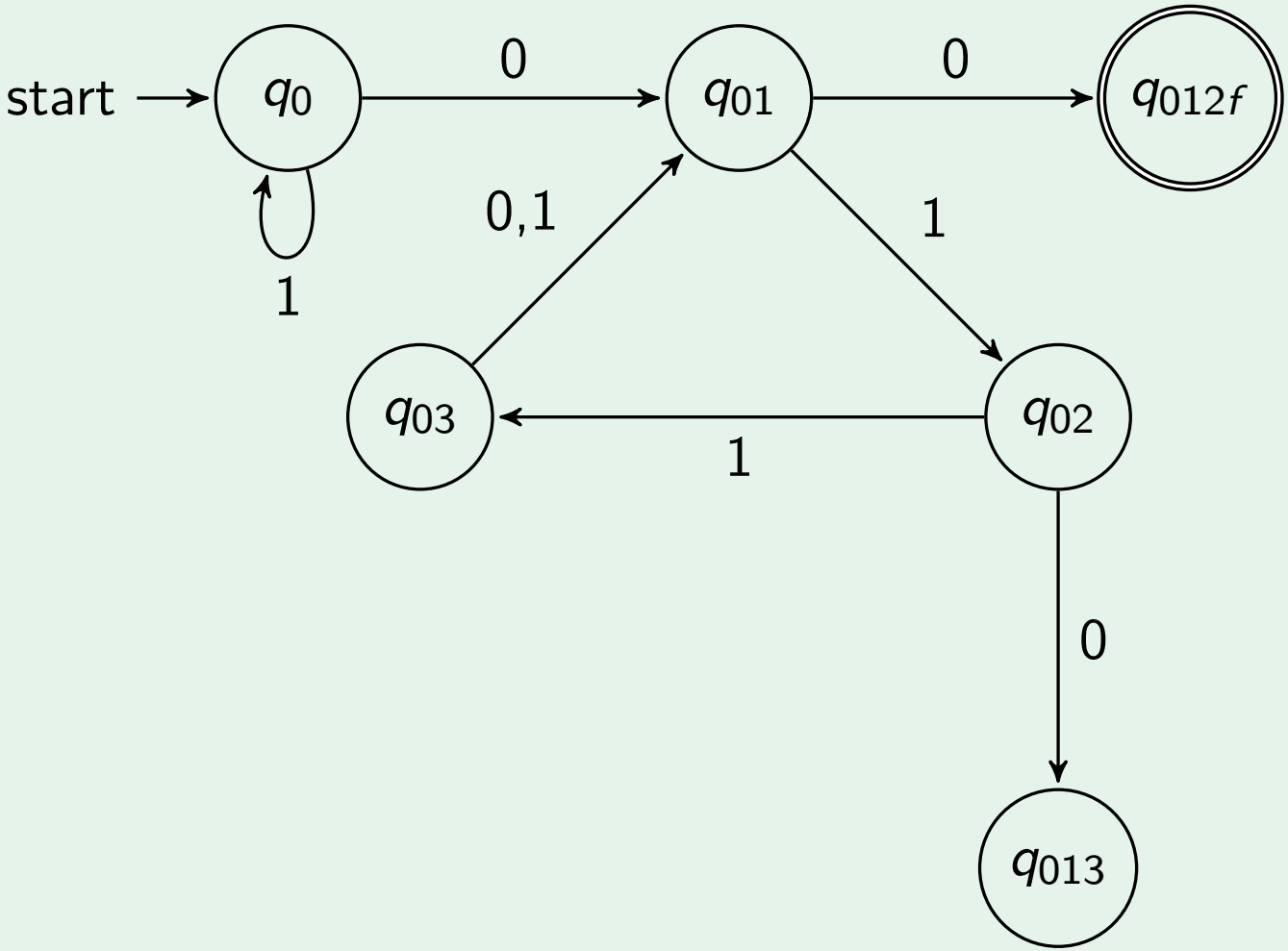
FSA determinization Example

Expanding q_{02}



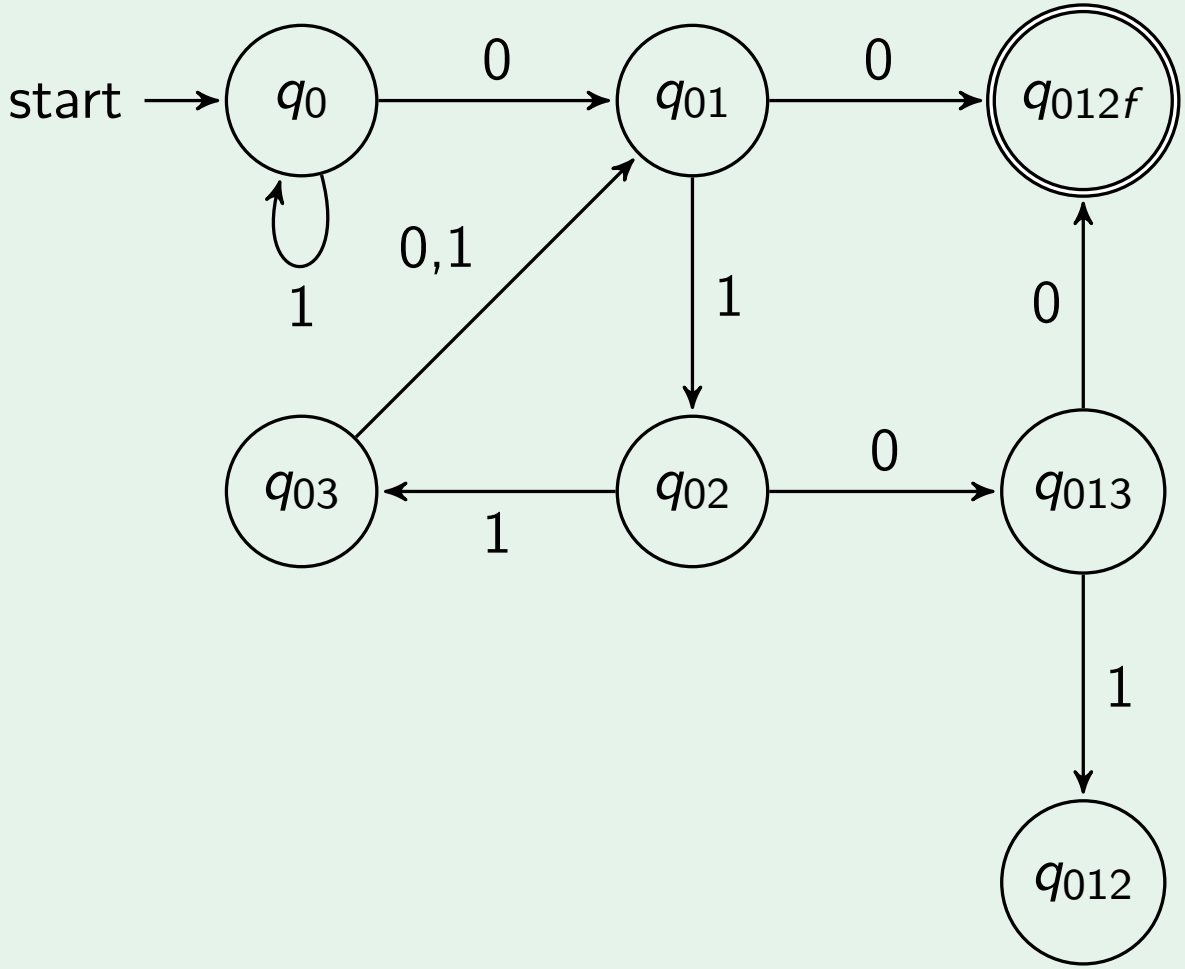
FSA determinization Example

Expanding q_{03}



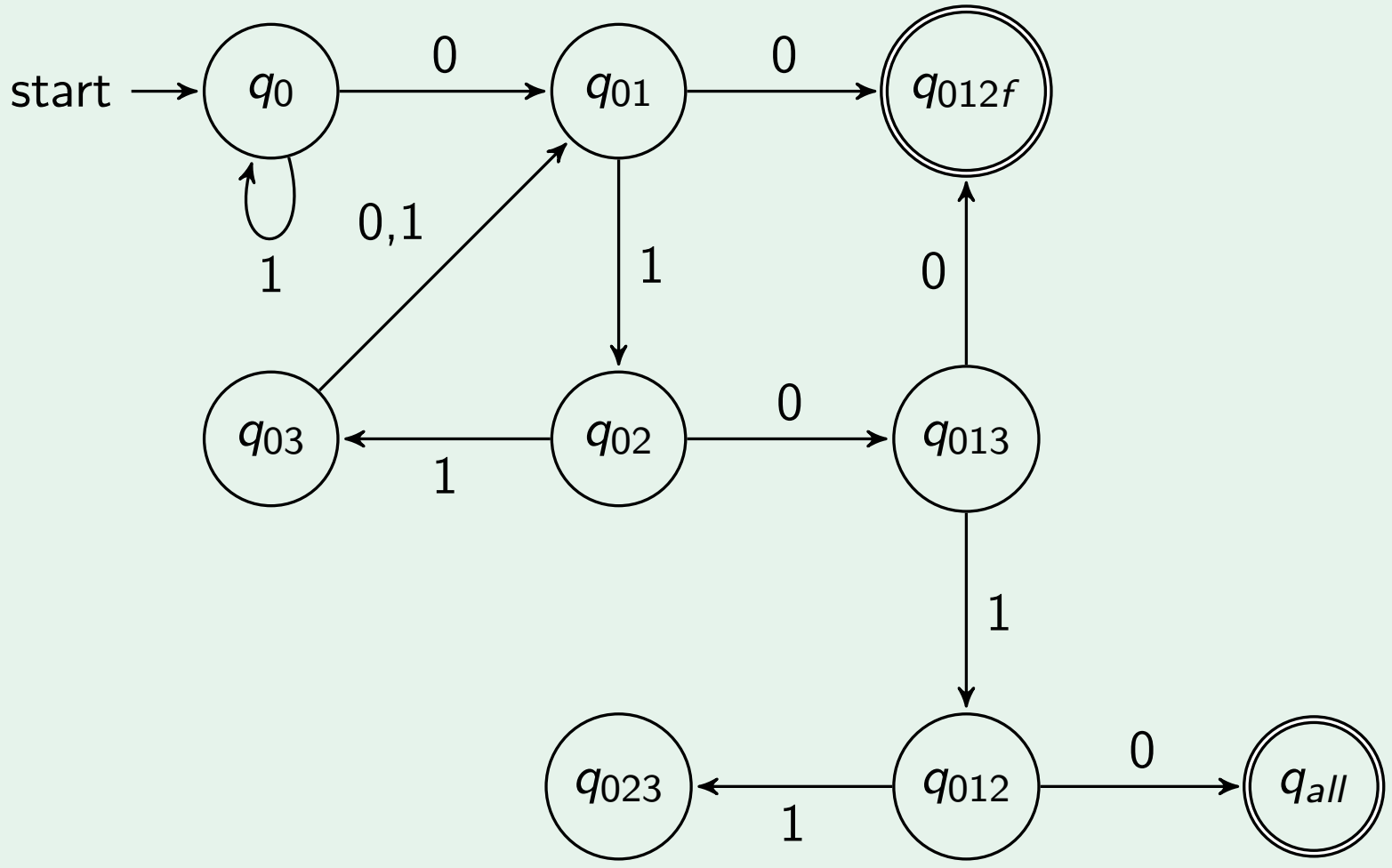
FSA determinization Example

Expanding q_{013}



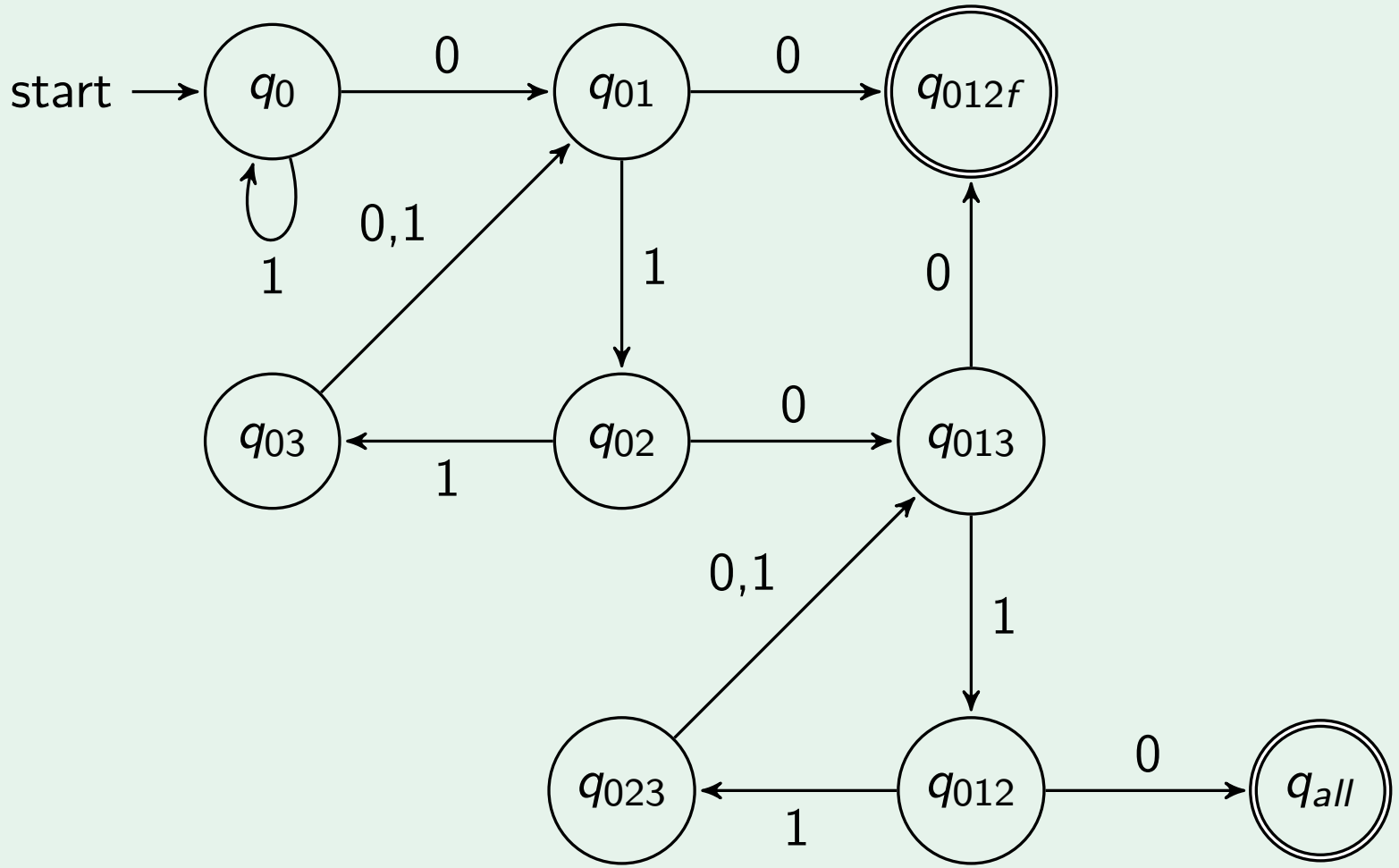
FSA determinization Example

Expanding q_{012}



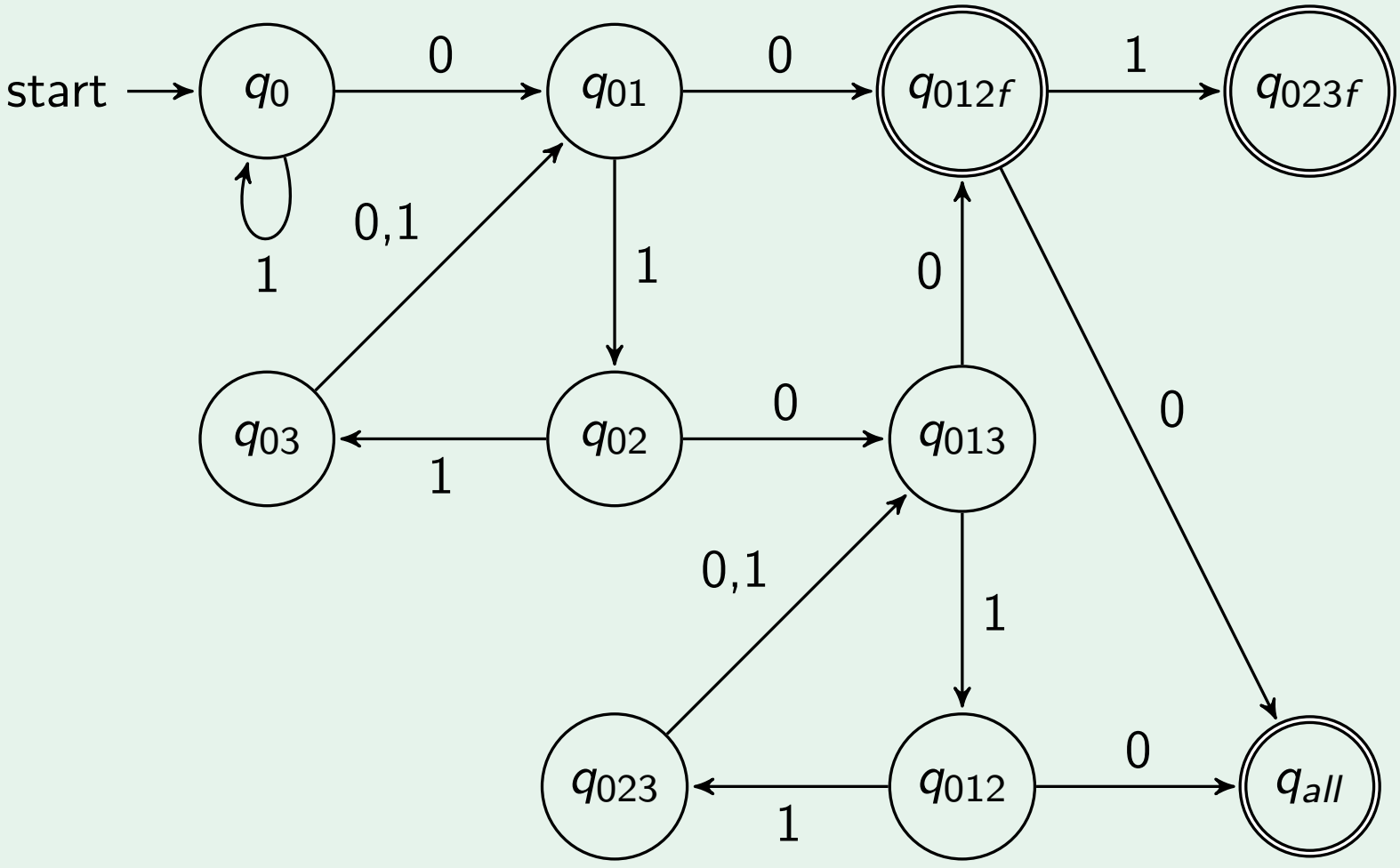
FSA determinization Example

Expanding q_{023}



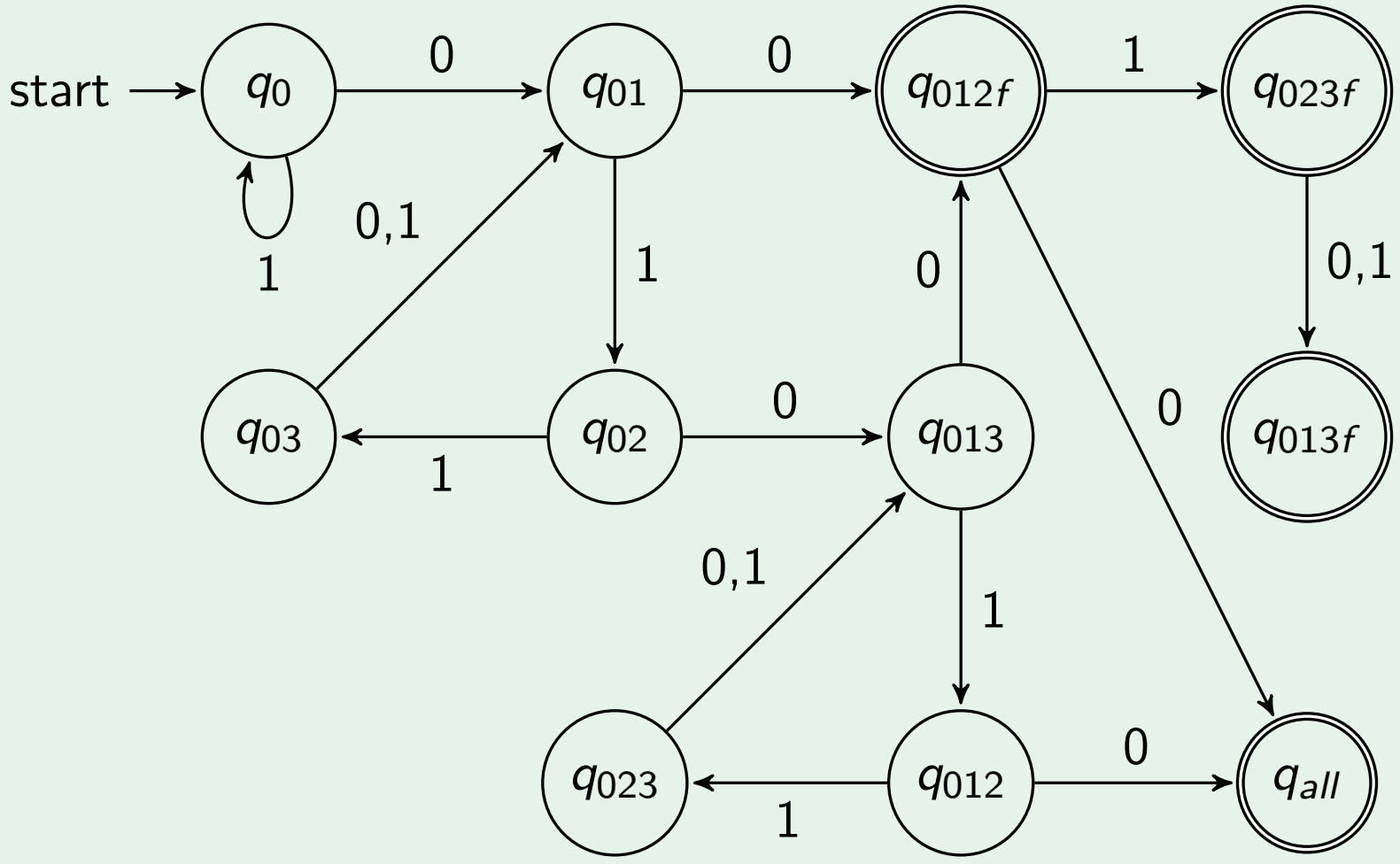
FSA determinization Example

Expanding q_{012f}



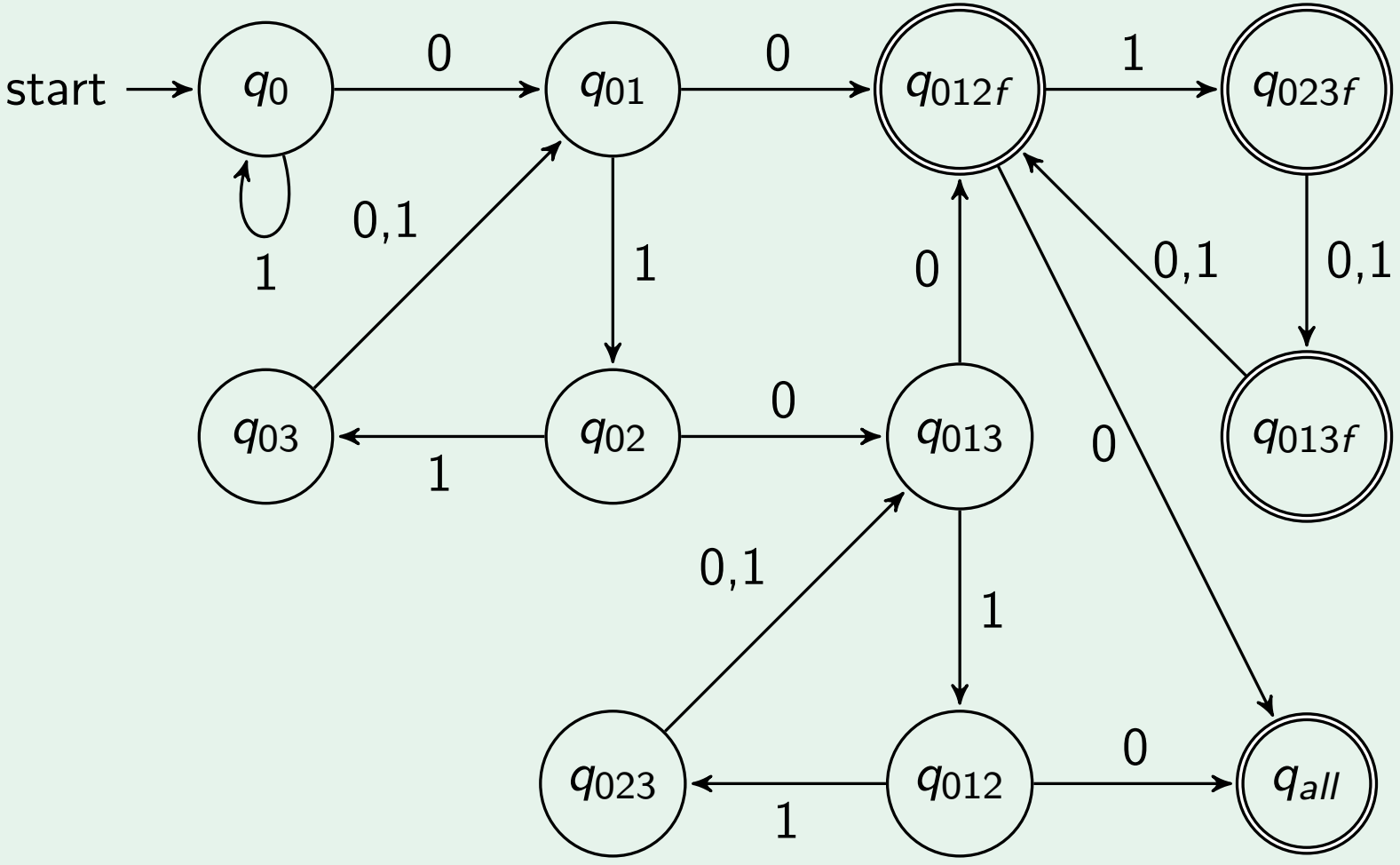
FSA determinization Example

Expanding q_{023f}



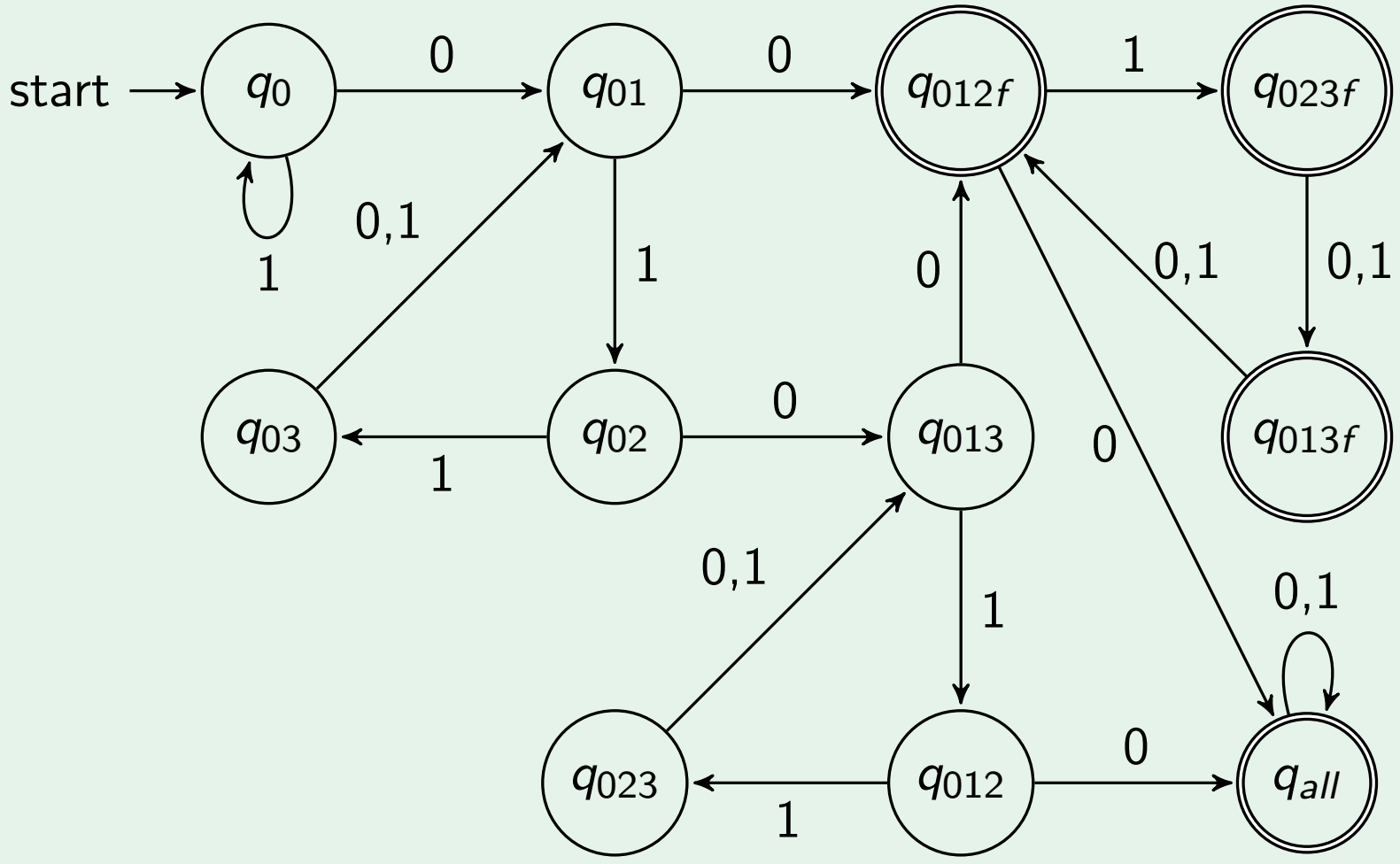
FSA determinization Example

Expanding q_{013f}



FSA determinization Example

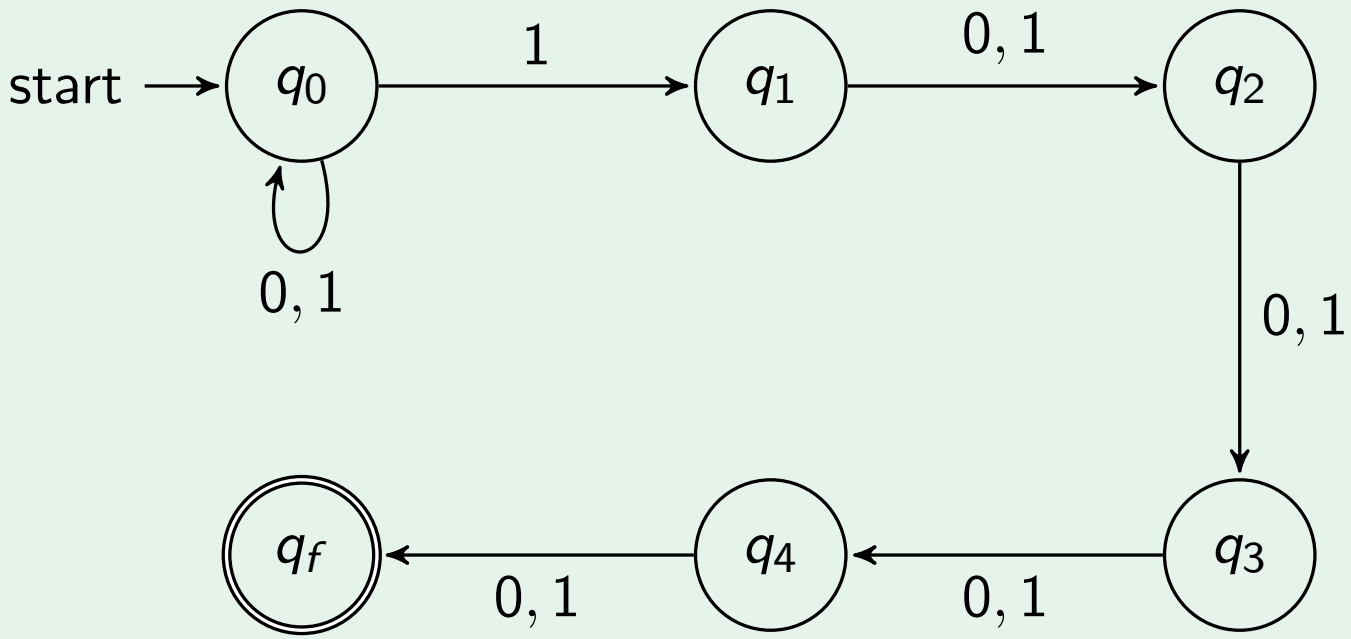
Expanding q_{all}



The Purpose of ND-FSA

An Example

Design an automaton able to recognize this language
$L = \{0|1\}^*1\{0|1\}^4$



The purpose of ND-FSA

Instead, how would have we recognized this language with a deterministic FSA?...

$$\Downarrow$$

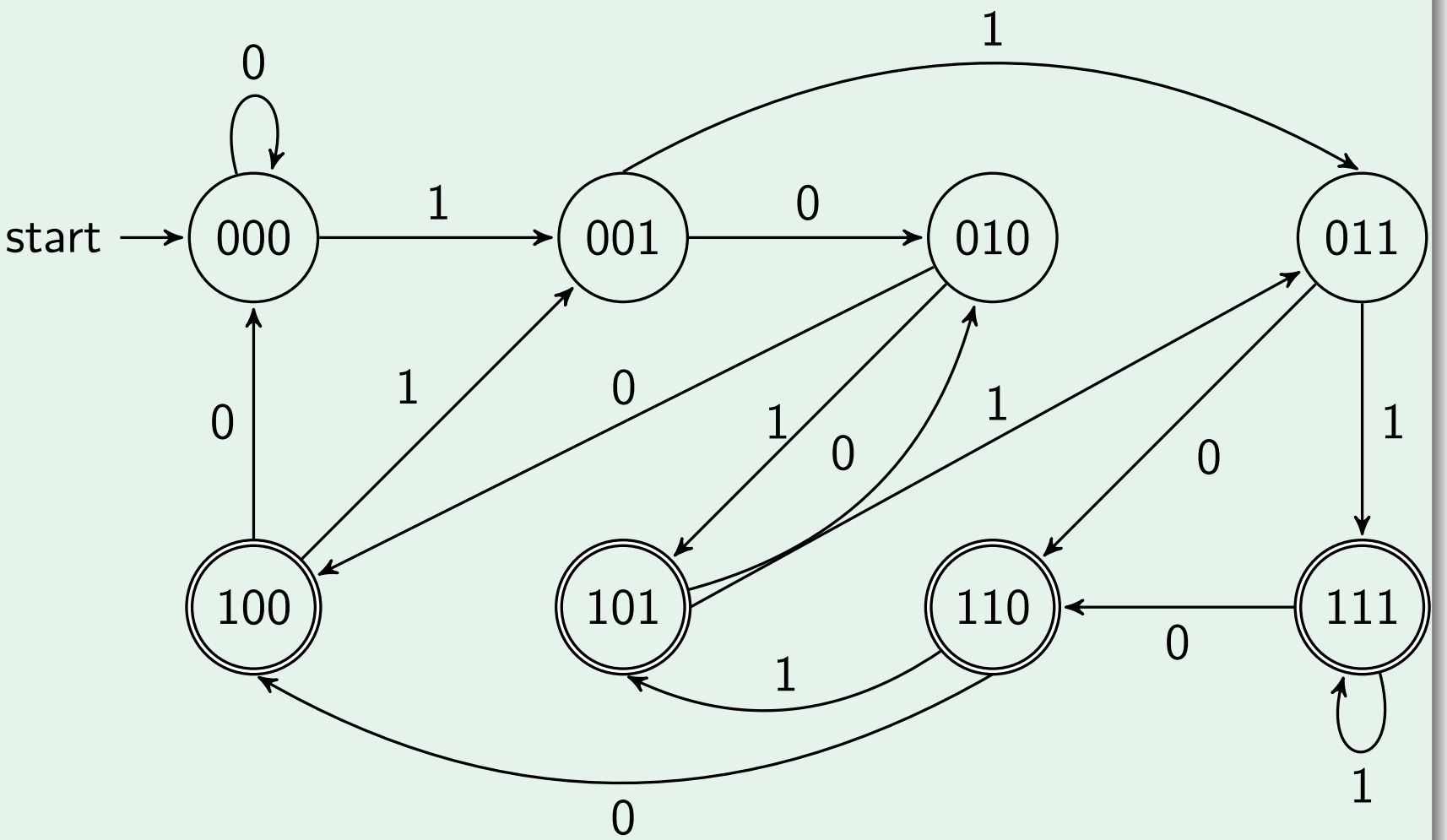
.. We would have needed a state for each possible sequence of the last 5 digits read, with accepting states being only the ones corresponding to sequences where the fifth-to-the-last digit is 1

$$\Downarrow$$

$2^5 = 32$ states, with 16 of them being accepting ones.

The purpose of ND-FSA

Deterministic FSA for the language $L = \{0|1\}^*1\{0|1\}^2$



Non-Deterministic PushDown Automaton

Definition

Recall the deterministic requirement of PushDown Automaton:

$$\exists \alpha \in \Gamma, q \in Q(\delta(q, \epsilon, \alpha) \neq \perp \implies \nexists i \in I(\delta(q, i, \alpha) \neq \perp))$$

If we remove this requirement and we modify the transition function to:

$$\delta : \mathbf{Q} \times \{\mathbf{I} \cup \epsilon\} \times \mathbf{\Gamma} \mapsto \wp_F(\mathbf{Q} \times \mathbf{\Gamma}^*)$$

We obtain a Non-Deterministic PushDown Automaton - NPDA

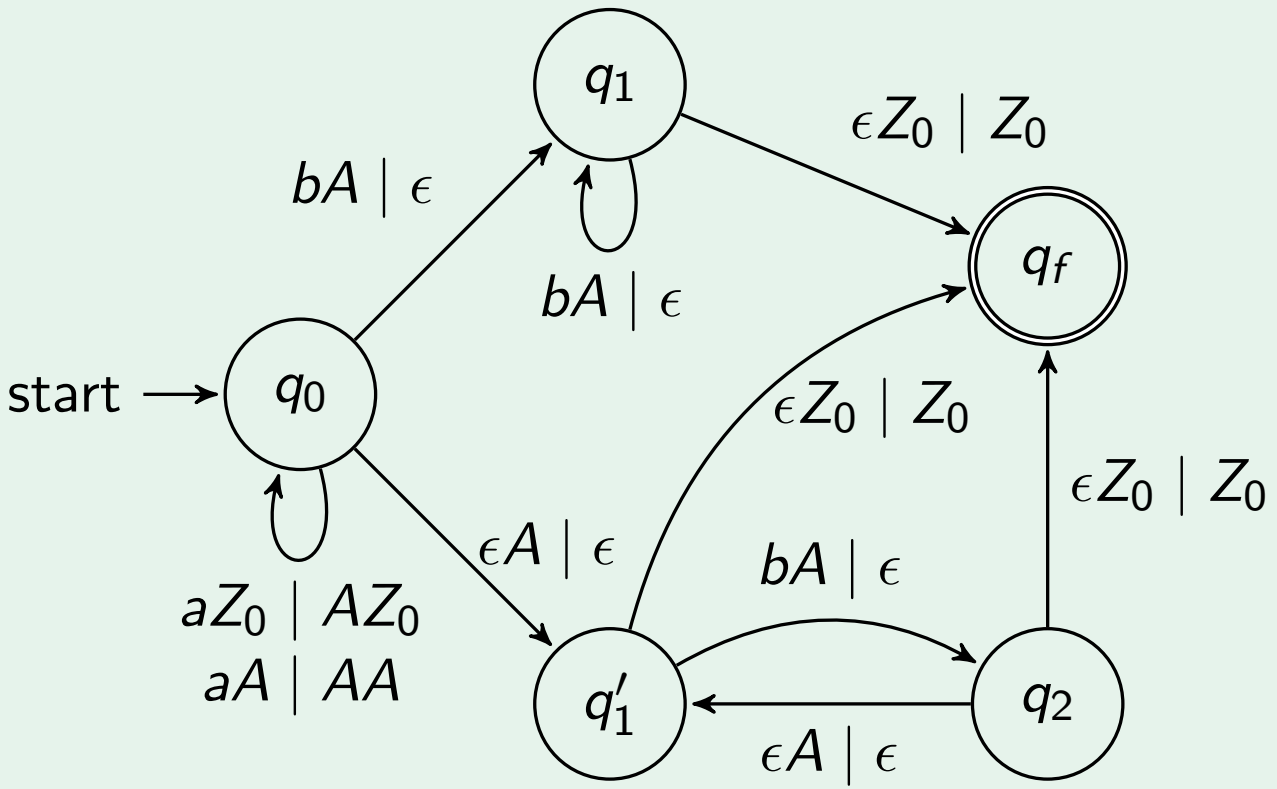
NPDA are more powerful than PDA:

1. Union of two languages recognizable by a PDA can be trivially performed
2. They can recognize languages where "guessing" about the structure of a string is needed

NPDA - Recognizing Union

$L = \{a^n b^n \cup a^n b^{\frac{n}{2}} \mid n \geq 1\}$

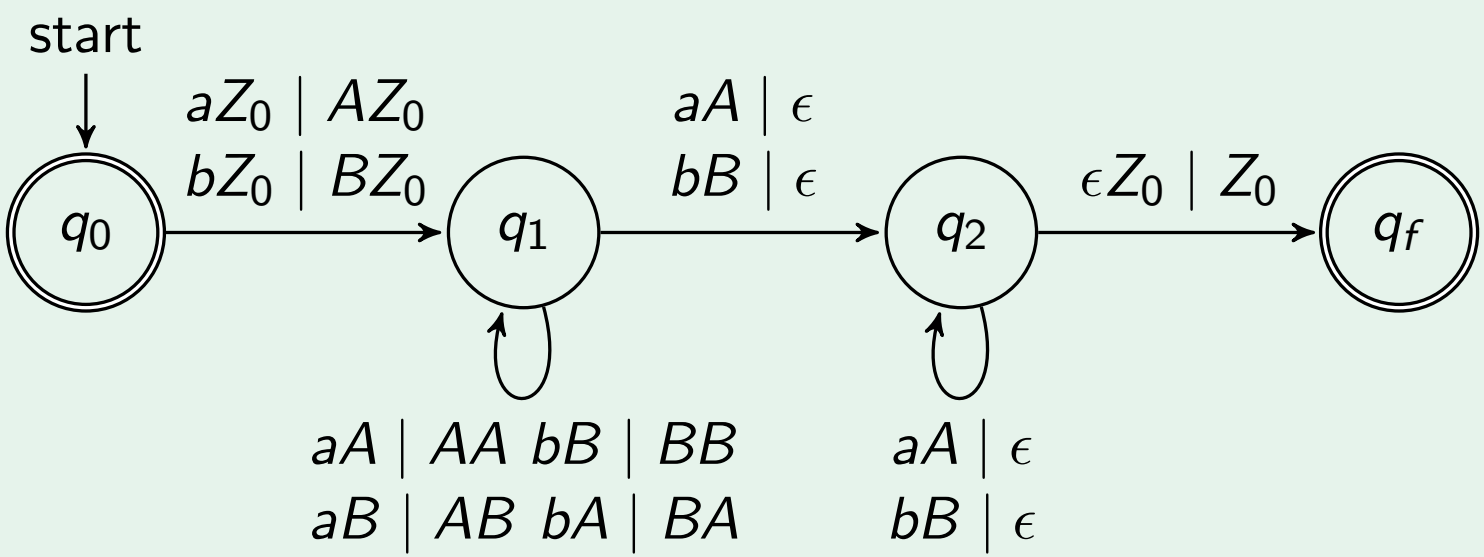
This language is a classical counterexample to show that PDA are not close w.r.t the union of languages. But with a NPDA...



NPDA - Exploiting Non-Deterministic Guessing

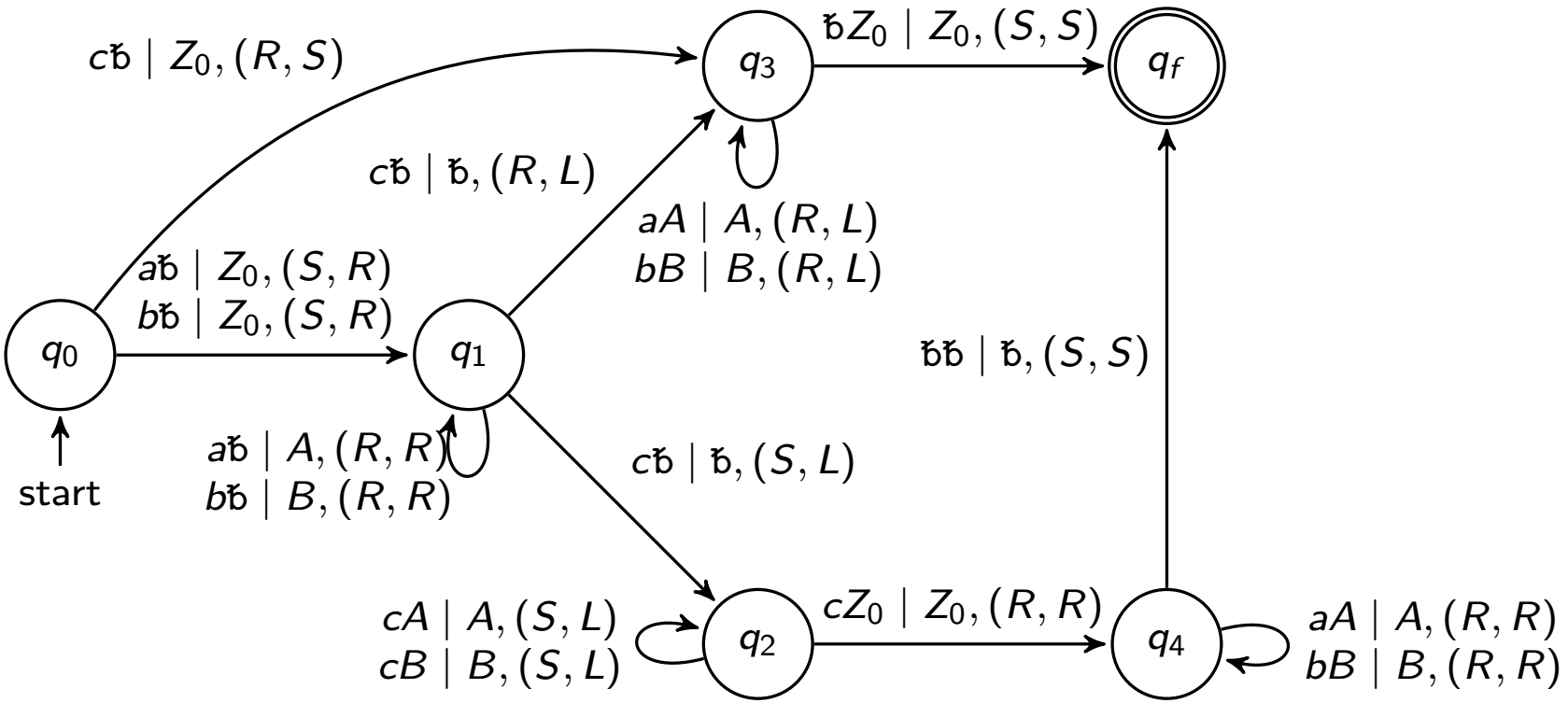
$L = \{ww^R \mid w \in \{a, b\}^*\}$

The problem with a PDA is that there is no marker which tells when w is finished, and thus we do not not know when we can start popping the stack to recognize w^R. No longer an issue with a NPDA:



Exploiting Non-Determinism on TM

Consider again the enriched parrot language:
$L = \{wcw \mid w \in \{a, b\}^*\} \cup \{wcw^R \mid w \in \{a, b\}^*\}$. Its recognition with a one tape TM is straightforward with a non-deterministic TM:



ND-TM Equivalence to Deterministic TM

Recall

The enriched parrot language can be recognized with a 1 tape deterministic TM too! How?

1. The TM starts recognizing the sublanguage wcw^R, using the tape as a stack
2. In case of a failure, the head of the tape is moved back to the first cell, while the head of the input tape is moved back to the first cell after the one containing the c character.
3. The tape can now be used as a queue to recognize the parrot language.

- At step 2, the TM performs backtracking, which is exactly the same behavior of a Non-Deterministic TM!
- Backtracking is the core mechanism to simulate a ND-TM with a deterministic one!

Complementing TM Exploiting Non-Determinism

Suppose we want to design an automaton able to recognize the language $L = \{a^n b^n c^n \mid n \geq 1\}^C$.

- We can exploit a kind of "divide et impera" strategy
- We split the language in sublanguages, and we design an automaton for each of these sublanguages
- Then, we merge these sublanguages by allowing a non-deterministic choice among these 3 automata

How to split the language?

$L_1 : (a^+ b^+ c^*)^C$

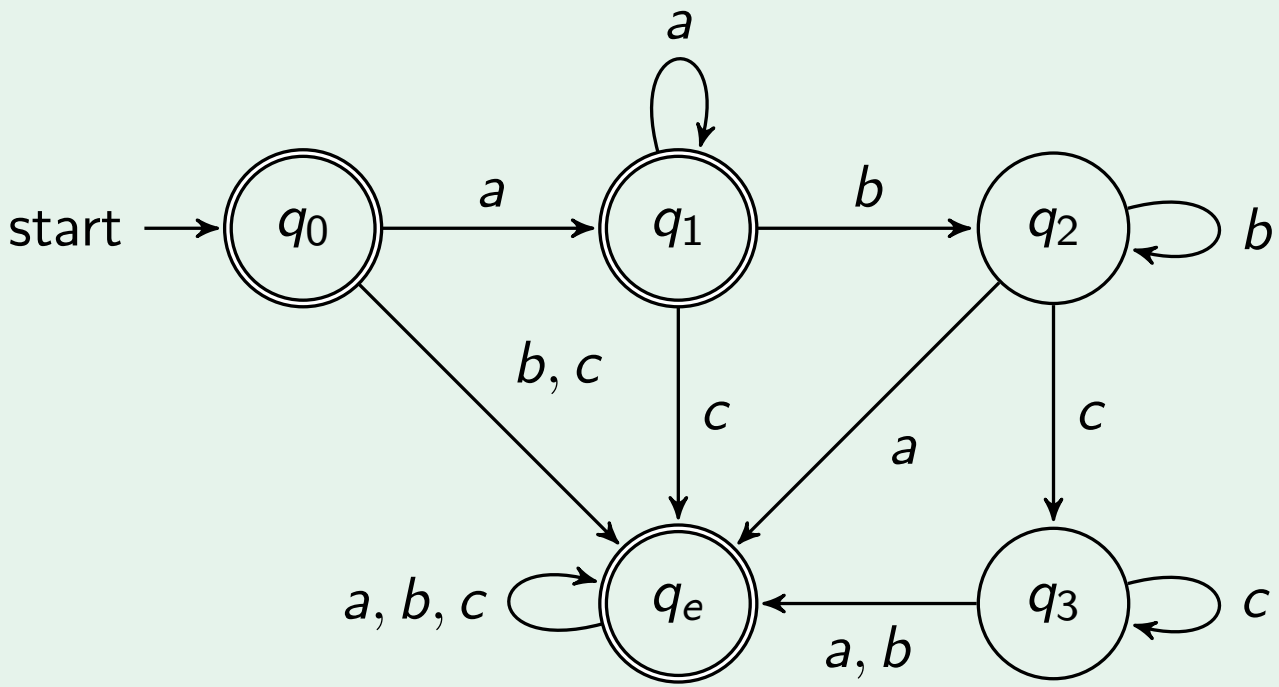
$L_2 : \{a^n b^m c^* \mid n \neq m\}$

$L_3 : \{a^* b^n c^m \mid n \neq m\}$

Complementing TM Exploiting Non-Determinism

Automaton for L_1

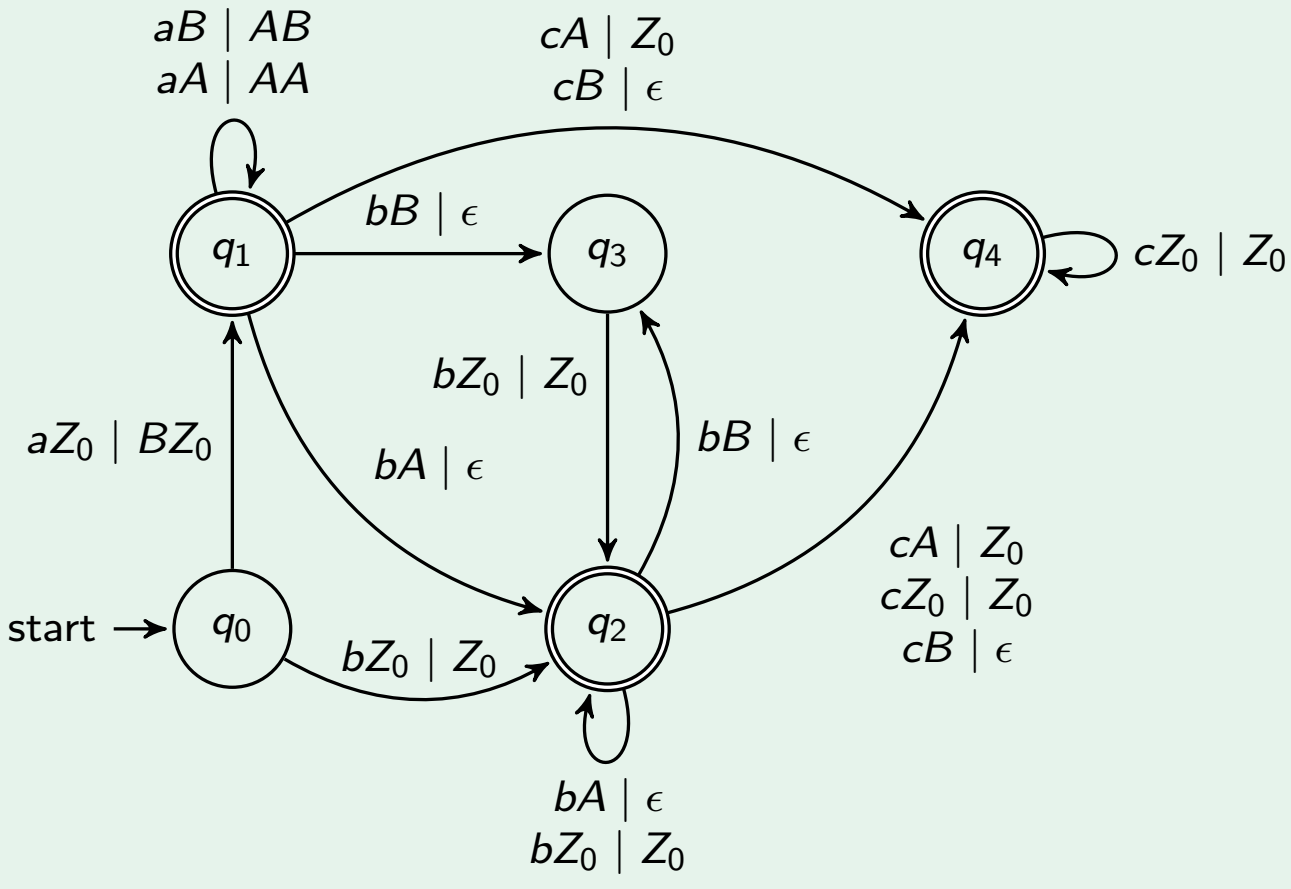
A FSA is sufficient!



Complementing TM Exploiting Non-Determinism

Automaton for L_2

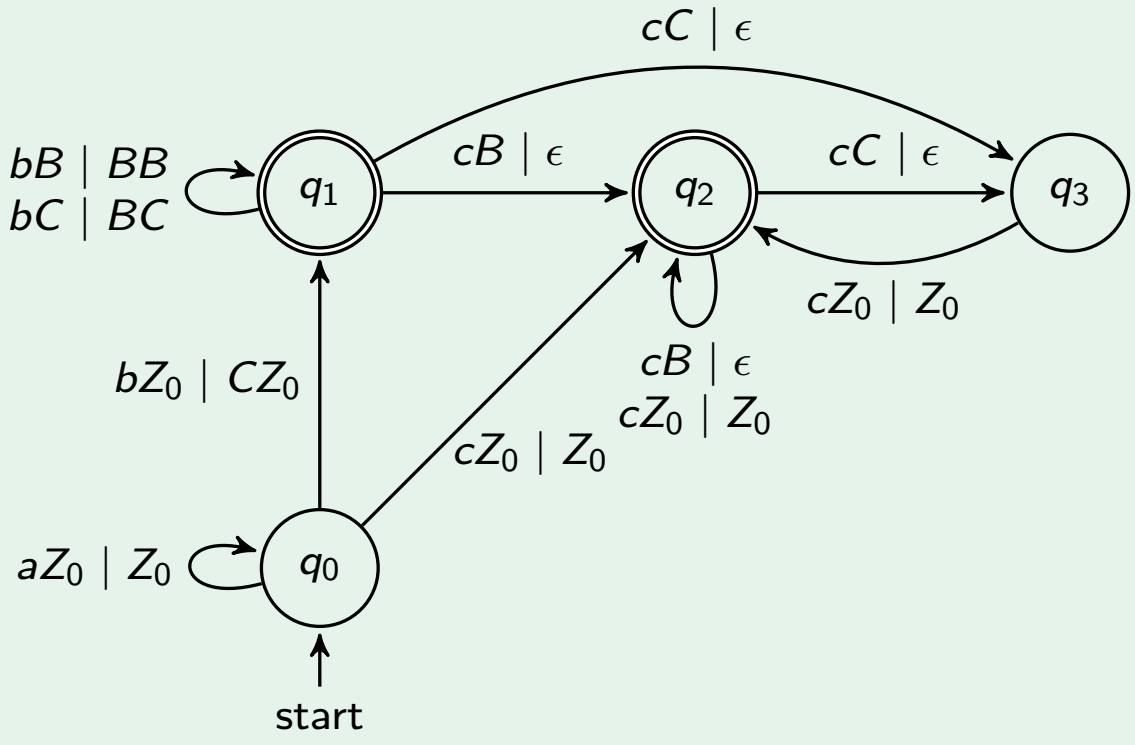
A Deterministic PDA is sufficient!



Complementing TM Exploiting Non-Determinism

Automaton for L_3

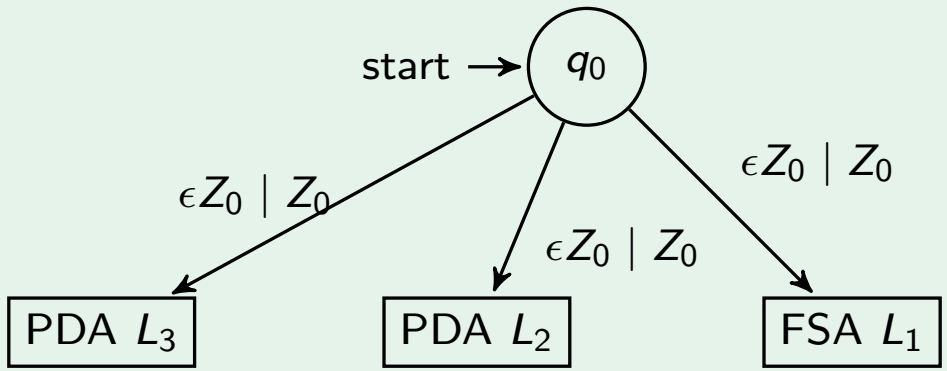
A Deterministic PDA is sufficient!



Complementing TM Exploiting Non-Determinism

We can now merge the 3 automata with a non-deterministic choice among the 3 initial states of the automata:

Recognizer for $L = \{a^n b^n c^n \mid n \geq 1\}^C$



The resulting automaton is a Non-Deterministic PDA

NB For the sake of correctness, the FSA for L_1 must be turned into a PDA which does not alter the stack

Complementing Parrot Language

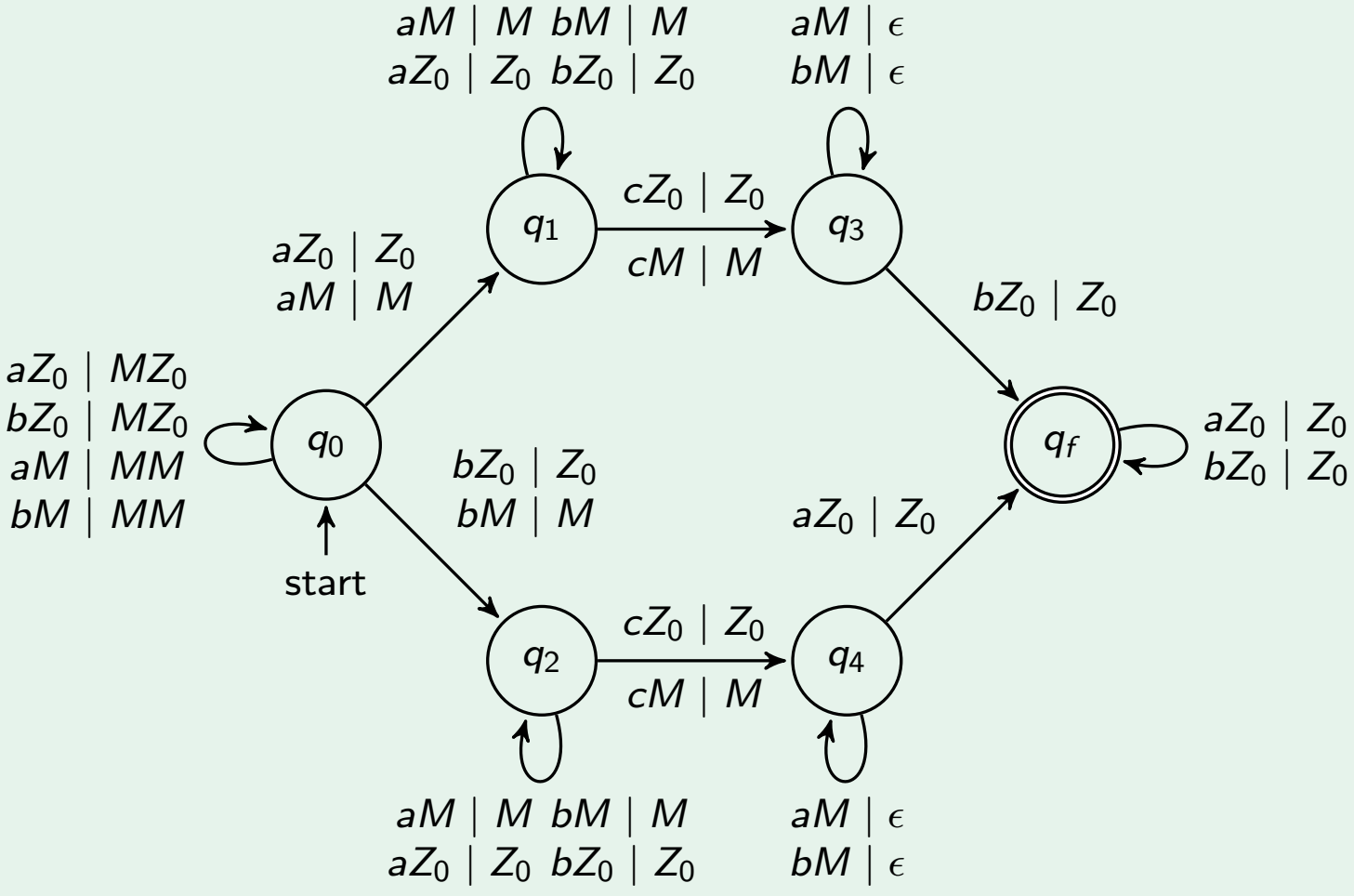
We want to design the automation with minimum computational power required to recognize the language
$L = \{wcw \mid w \in \{a, b\}^*\}^C$

- Again, we split in different sublanguages, each breaking one constraint imposed by the original language on its strings
- Which sublanguages?
  1. $L_1 = \{x \mid \exists \alpha, \beta, \gamma, \delta \in \{a, b\}^* ((x = \alpha.b.\beta.c.\gamma.a.\delta \vee x = \alpha.a.\beta.c.\gamma.b.\delta) \wedge |\alpha| = |\gamma|)\}$: break the constraint that each character in the first half of the string is equal to the corresponding character in the second half of the string
  2. $L_2 = \{(a \mid b)^* c (a \mid b)^*\}^C$: break the structure of string
  3. $L_3 = \{x \mid \exists \alpha, \beta \in \{a, b\}^* (x = \alpha.c.\beta \wedge |\alpha| \neq |\beta|)\}$: break the constraint that the string is split by the single character c in two equally long portions
- Then, we merge these sublanguages with a non deterministic choice among the 3 corresponding automata

Complementing Parrot Language

Automaton for L1

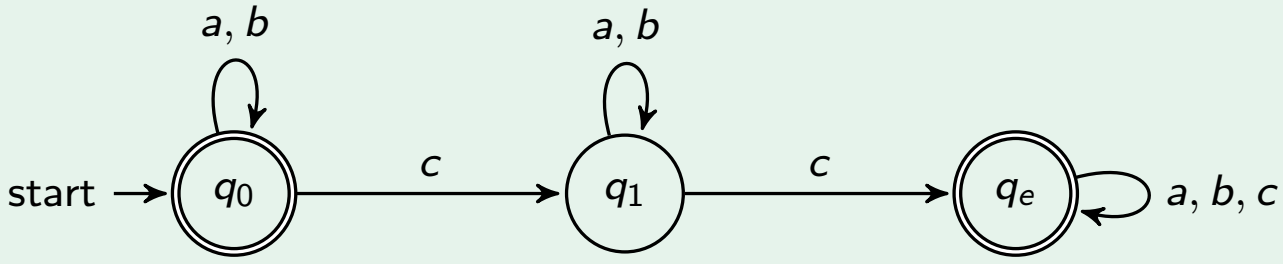
A NPDA is sufficient:



Complementing Parrot Language

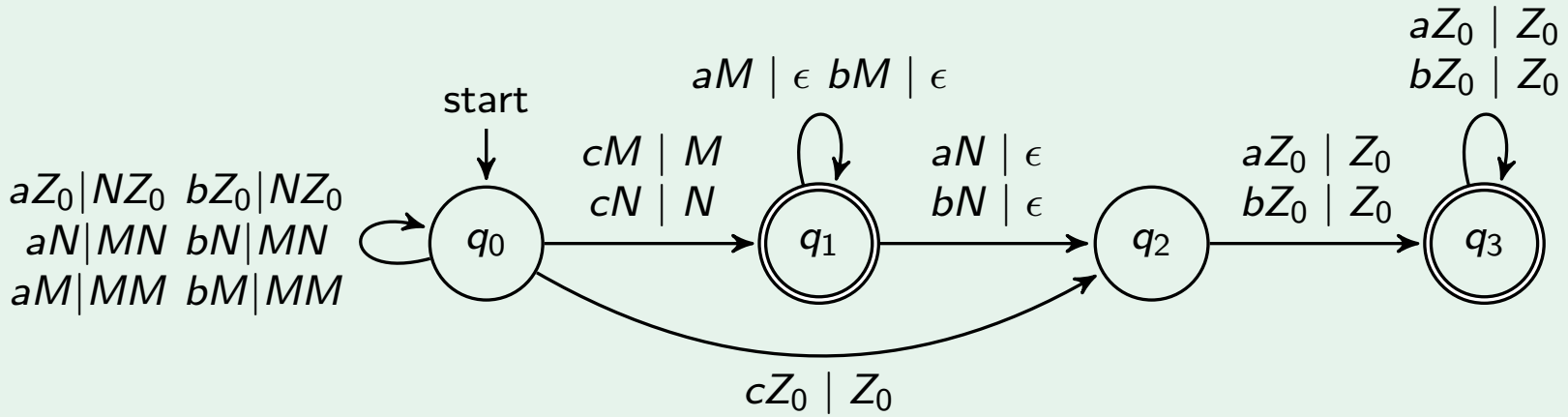
Automaton for L2

A FSA is sufficient:



Automaton for L3

A DPDA is sufficient:



A Trickier Variant

We want to recognize $L = (ww \mid w \in \{a, b\}^*)^C$. Which automaton?

Main Idea: Decomposition Into Sub-Languages! How?

$$\Downarrow$$

$L = L_1 \cup L_2 \cup L_3$

$L_1 : \{x \mid |x| = 2k + 1, k \geq 0\}$

$L_2 : \{x \mid \exists \alpha, \beta, \gamma (x = \alpha.a.\beta.b.\gamma \wedge |\alpha.\gamma| = |\beta|)\}$

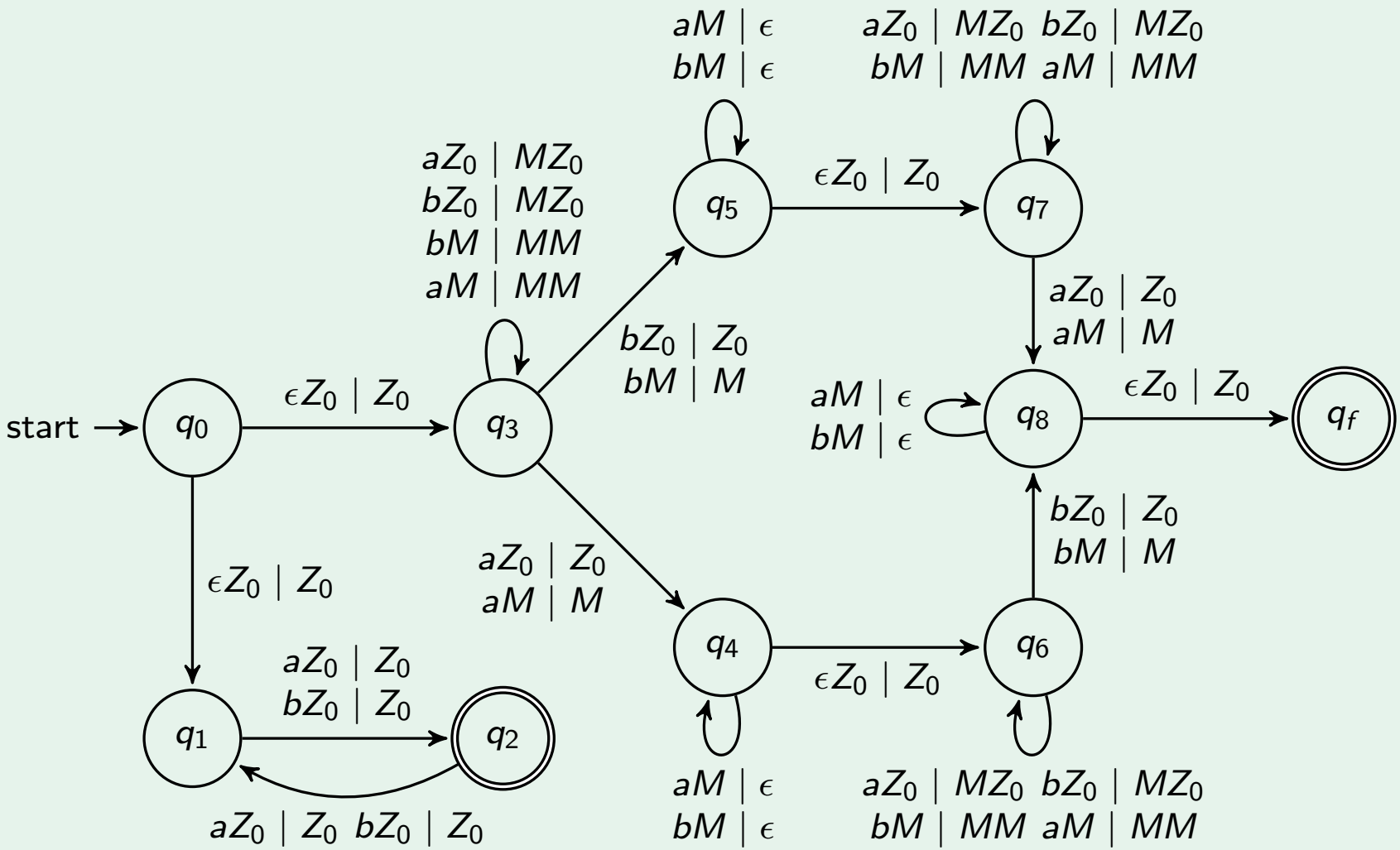
$L_3 : \{x \mid \exists \alpha, \beta, \gamma (x = \alpha.b.\beta.a.\gamma \wedge |\alpha.\gamma| = |\beta|)\}$

Can we recognize L_2 and L_3 with a NPDA?

Yes, we can split β in two parts: $\beta = \beta_1.\beta_2, |\beta_1| = |\alpha| \wedge |\beta_2| = |\gamma|$

A Trickier Variant

Recognizer for $L = (ww \mid w \in \{a, b\}^*)$



Determining The Minimum Computational Power Required

Determine the weakest automaton able to recognize the following languages:

1. $L_1 = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$
2. $L_2 = \{1a^n b^n \mid n \geq 1\} \cup \{2a^n b^{2n} \mid n \geq 1\}$
3. $L_3 = \{a^n 1 b^n \mid n \geq 1\} \cup \{a^n 2 b^{2n} \mid n \geq 1\}$
4. $L_4 = \{a^n b^n 1 \mid n \geq 1\} \cup \{a^n b^{2n} 2 \mid n \geq 1\}$

L_1, L_4 : NPDA! We cannot determine the number of b to be counted

L_2, L_3 : PDA! 1 and 2 allows to determine the number of b to be counted

Determining the Minimum Computational Power Required

Determine the weakest automaton able to recognize the following languages:

1. $L_1 = \{a^n b^p b^p \mid n \geq 1, p \geq 1\} \cup \{a^n b^p a^n \mid n \geq 1, p \geq 1\}$
2. $L_2 = \{a^n b^n \mid n \geq 1\} \cup \{b^n c^n \mid n \geq 1\}$
3. $L_3 = \{a^n b^n c^+ \mid n \geq 1\} \cup \{a^+ b^n c^n \mid n \geq 1\}$

$$\Downarrow$$

1. $L_1 = \{a^n b^{2p} \mid n \geq 1, p \geq 1\} \cup \{a^n b^p a^n \mid n \geq 1, p \geq 1\} \Rightarrow$ PDA: if the number of b is odd, then the sequence a^n after b^p is mandatory, otherwise it is optional
2. L_2: Recognize $a^n b^n$ (respectively $b^n c^n$) if the string starts with a (resp. b) $\Rightarrow$ PDA
3. L_3: The sub-languages can neither be recognized simultaneously nor be distinguished at the beginning of the string $\Rightarrow$ NPDA

Determining the Minimum Computational Power Required

What about complements of L_1, L_2 and L_3?

1. L_1^C, L_2^C: PDA are closed with respect to complement. Both L_1 and L_2 are recognized by a PDA $\Rightarrow$ their complements are recognized by a PDA too
   - May they be recognized by a FSA? No, as FSA are closed w.r.t. complement too!
2. $L_5 = L_3^C$: The complement of a language recognized by a NPDA cannot be recognized by a PDA. Why?
   - If L_5 is recognized by a PDA, then its complement (i.e., L_3) must be recognized by a PDA too
   - Thus L_5 can be recognized by either a NPDA or a TM
   - $L_5 = \{a^n b^n c^+ \mid n \geq 1\}^C \cap \{a^+ b^n c^n \mid n \geq 1\}^C$
   - Necessary to simultaneously verify that the number of a is different from the number of b and that the number of b is different from c $\Rightarrow$ Impossible with a stack $\Rightarrow$ TM

What about $L_1^C \cup L_2^C$?

- $L_1^C \cup L_2^C = (L_1 \cap L_2)^C = \{a^{2p} b^{2p} \mid p \geq 1\}^C \Rightarrow$ PDA