

# FORMALE SYSTEME

## 10. Vorlesung: Grenzen regulärer Sprachen / Probleme für Automaten

Markus Krötzsch

Lehrstuhl Wissensbasierte Systeme

TU Dresden, 13. November 2017

## Nicht-reguläre Sprachen

## Minimale Automaten

Für jede reguläre Sprache  $L$

- ... gibt es einen minimalen (totalen) DFA
- ... der eindeutig ist bis auf Umbenennung von Zuständen
- ... den man durch Reduktion jedes beliebigen DFA für  $L$  erhalten kann
- ... oder auch als Myhill-Nerode-Minimalautomat

Allerdings ist der minimale DFA unter Umständen viel größer als ein minimaler NFA (dafür ist letzterer schwerer zu finden und nicht eindeutig)

## Nicht-reguläre Sprachen

Sind alle Sprachen regulär?

**Sicher nicht** (dazu gibt es zu viele Sprachen)

Sind alle Typ-0-Sprachen regulär?

**Nein**, auch das gilt nicht

Wie aber zeigt man das?

- Behauptung „Sprache  $L$  ist regulär!“  $\leadsto$  es genügt, **einen** Automaten, regulären Ausdruck oder eine Typ-3-Grammatik für  $L$  anzugeben
- Behauptung „Sprache  $L$  ist nicht regulär!“  $\leadsto$  man müsste zeigen, dass es **keinen** Automaten, **keinen** regulären Ausdruck bzw. **keine** Typ-3-Grammatik für  $L$  gibt

## Nichtregularität mit Myhill & Nerode

Myhill & Nerode liefern uns ein besseres Kriterium:

Satz (Myhill & Nerode): Eine Sprache  $L$  ist genau dann regulär, wenn  $\approx_L$  endlich viele Äquivalenzklassen hat.

Daraus folgt:

Satz: Wenn  $\approx_L$  unendlich viele Äquivalenzklassen hat, dann ist  $L$  nicht regulär.

Beispiel: Wir haben gesehen, dass die Sprache  $\{a^n b^n \mid n \geq 0\}$  unendlich viele Äquivalenzklassen erzeugt. Diese Sprache ist demnach nicht regulär.

## Nichtregularität mit Abschlusseigenschaften

Wir haben bereits gezeigt:

Satz: Wenn  $L_1$  und  $L_2$  regulär sind, dann auch  $L_1 \cap L_2$ ,  $L_1 \cup L_2$ ,  $L_1^*$  und  $\bar{L}_1$ .

Auch hier liefert die Umkehrung gute Kriterien, z.B.

Satz: Wenn  $L_1$  regulär und  $L_1 \cap L_2$  nicht regulär ist, dann ist  $L_2$  ebenfalls nicht regulär.

Beispiel: Wir betrachten über dem Alphabet  $\{a, b\}$  die Sprache

$L = \{w \mid w \text{ enthält ebenso viele } a \text{ wie } b\}$ .

Dann gilt  $L \cap \{a\}^* \{b\}^* = \{a^n b^n \mid n \geq 0\}$ .

Weil  $\{a\}^* \{b\}^*$  regulär ist und  $\{a^n b^n \mid n \geq 0\}$  nicht, kann  $L$  ebenfalls nicht regulär sein.

$a^n b^n$

Die Sprache  $a^n b^n$  gilt als Musterbeispiel nicht-regulärer Sprachen

Intuition: „Reguläre Sprachen können nicht zählen.“

Ein Automat, der diese Sprache erkennen wollte, müsste die genaue Zahl der schon gelesenen  $a$ s speichern

$\leadsto$  beliebig viel Speicher nötig

Viele Sprachdefinitionen mit Bezug zu konkreten Zahlen verhalten sich ähnlich

## Nichtregularität durch Pumpen

Ein weiteres klassisches Kriterium für Nicht-Regularität ist, dass die Wörter regulärer Sprachen beliebig „aufgepumpt“ werden können.

**Idee:**

- Jeder DFA hat nur endlich viele Zustände  $n$
- Aber manche reguläre Sprachen enthalten beliebig lange Wörter  
Wie kann ein DFA Wörter mit mehr als  $n$  Zeichen akzeptieren?
- Dann muss der DFA beim Einlesen einen Zustand mehr als einmal besuchen
- Dafür muss es in den Zustandsübergängen eine Schleife geben
- Diese Schleife kann man aber auch mehr als einmal durchlaufen

Jedes akzeptierte Wort mit  $\geq n$  Zeichen hat einen Teil, den man beliebig oft wiederholen – „aufpumpen“ – kann

# Das Pumping-Lemma

Satz (Pumping Lemma): Für jede reguläre Sprache  $L$  gibt es eine Zahl  $n \geq 0$ , so dass gilt:  
für jedes Wort  $z \in L$  mit  $|z| \geq n$   
gibt es eine Zerlegung  $z = uvw$  mit  $|v| \geq 1$  und  $|uv| \leq n$ , so dass:  
für jede Zahl  $k \geq 0$  gilt:  $uv^k w \in L$

**Beweis:** Sei  $M$  ein DFA für  $L$  mit  $|Q|$  Zuständen. Wir wählen  $n = |Q| + 1$ .  
Ein akzeptierender Lauf für ein beliebiges Wort  $z$  mit  $|z| = \ell \geq n$  muss in den ersten  $n$  Schritten einen Zustand  $p$  zweimal besuchen (sagen wir: nach  $i$  und  $j$  Schritten), hat also die Form:

$$q_0 \xrightarrow{z_1} q_1 \xrightarrow{z_2} \dots \xrightarrow{z_{i-1}} q_{i-1} \xrightarrow{z_i} p \xrightarrow{z_{i+1}} q_{i+1} \xrightarrow{z_{i+2}} \dots \xrightarrow{z_{j-1}} q_{j-1} \xrightarrow{z_j} p \xrightarrow{z_{j+1}} q_{j+1} \xrightarrow{z_{j+2}} \dots \xrightarrow{z_\ell} q_\ell$$

Die gesuchte Zerlegung ist  $u = z_1 \dots z_i$ ,  $v = z_{i+1} \dots z_j$ ,  $w = z_{j+1} \dots z_\ell$ .  
Der Lauf  $(q_0 \dots q_{i-1} p)(q_{i+1} \dots q_{j-1} p)^k (q_{j+1} \dots q_\ell)$  akzeptiert  $uv^k w$ .  $\square$

# Pumpen für Nichtregularität

Die Pump-Eigenschaft ist für Regularität **notwendig**, aber **nicht hinreichend**: man kann auch manche nicht-reguläre Sprachen pumpen.

Daher eignet sich das Pumping-Lemma eher zum Beweis der Nichtregularität

Beispiel: Sei  $L = \{a^k \mid k \text{ ist Primzahl}\}$ .  
Angenommen es gäbe einen Wert  $n$  für den diese Sprache aufgepumpt werden kann.  
Es gibt eine Primzahl  $\ell > n + 2$ . Laut Pump-Eigenschaft finden wir eine Zerlegung von  $a^\ell = uvw$  für die insbesondere gilt  $uv^{|\ell w|} w \in L$ . Aber  $uv^{|\ell w|} w = a^{(k+1)|\ell w|} \notin L$ . Widerspruch.  
 $L$  ist daher nicht regulär.

# Beispiel

Beispiel: Die Sprache von  $b^*(ab^*ab^*)^*$  (Wörter mit gerader Anzahl  $a$ s). Der Parameter  $n$  aus dem Pumping-Lemma kann 2 sein:

- Wenn  $z$  die Form  $b^y$  hat, wähle  $u = \epsilon$ ,  $v = b$  und  $w = y$
- Wenn  $z$  die Form  $ab^y$  hat, wähle  $u = a$ ,  $v = b$  und  $w = y$
- Wenn  $z$  die Form  $aa^y$  hat, wähle  $u = \epsilon$ ,  $v = aa$  und  $w = y$

Beispiel: Für endliche Sprachen ist die Eigenschaft trivial. Man wählt  $n$  einfach so groß, dass es keine Wörter  $z$  mit  $|z| \geq n$  gibt, für welche weitere Eigenschaften gefordert werden.

# Ist das Pumping-Lemma sinnvoll?

- **Kontra:** Myhill-Nerode liefert ein notwendiges **und** hinreichendes Kriterium für Regularität  
 $\leadsto$  immer anwendbar, um (Nicht-)Regularität zu zeigen
- **Pro:** Pumping funktioniert in den allermeisten praktischen Fällen auch (notfalls kann man Abschlusseigenschaften zu Hilfe nehmen)
- **Pro:** Manchmal ist der Beweis mit Pumping einfacher
- **Kontra:** Das formale Pumping Lemma ist relativ kompliziert
- **Pro:** Die Pumping-Idee ist intuitiv und wir werden sie später noch einmal bei kontextfreien Sprachen nutzen

## Algorithmen für Automaten

## Wichtige Probleme für Sprachen

Das **Wortproblem** für eine Sprache  $L$  über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein Wort  $w \in \Sigma^*$

**Ausgabe:** „ja“ wenn  $w \in L$  und „nein“ wenn  $w \notin L$

Ziel bei der Lösung des Wortproblems:

- Finde einen Algorithmus, der diese Funktion berechnet
- Die Sprache  $L$  ist **nicht** Teil der Eingabe, sondern fester Bestandteil des Algorithmus
- Es ist daher unerheblich, wie wir Sprachen darstellen (Grammatik, Automat, ...)

## Wichtige Probleme für Automaten

Das **Wortproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein FA  $M$  und ein Wort  $w \in \Sigma^*$

**Ausgabe:** „ja“ wenn  $w \in L(M)$  und „nein“ wenn  $w \notin L(M)$

- Hier ist die Sprache (gegeben als FA) Teil der Eingabe
- Die Repräsentation (DFA oder NFA?) spielt unter Umständen eine wichtige Rolle

Das **Leerheitsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein FA  $M$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $L(M) = \emptyset$  und „nein“ wenn  $L(M) \neq \emptyset$

## Wichtige Probleme für Automaten (2)

Das **Inklusionsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** zwei FAs  $M_1$  und  $M_2$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $L(M_1) \subseteq L(M_2)$ ; andernfalls „nein“

Das **Äquivalenzproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** zwei FAs  $M_1$  und  $M_2$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $L(M_1) = L(M_2)$ ; andernfalls „nein“

Manche Probleme kann man mittels anderer ausdrücken:

- $L(M_1) = L(M_2)$  gdw.  $L(M_1) \subseteq L(M_2)$  und  $L(M_2) \subseteq L(M_1)$
- $L(M) = \emptyset$  gdw.  $L(M) \subseteq L(M_0)$ , wobei  $M_0$  ein FA ist, der keine Wörter akzeptiert

Wir werden noch mehr solche **Reduktionen** sehen

## Das Leerheitsproblem für DFA und NFA

Das **Leerheitsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein FA  $\mathcal{M}$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $\mathbf{L}(\mathcal{M}) = \emptyset$  und „nein“ wenn  $\mathbf{L}(\mathcal{M}) \neq \emptyset$

**Anwendungsbeispiel:** Konsistenzprüfung einer Systemspezifikation

- In der formalen Verifikation kann (erwünschtes) Systemverhalten mit Automaten dargestellt werden
- Wörter entsprechen z.B. erlaubten Programmdurchläufen
- Leerheitstest: prüfe, ob die Spezifikation überhaupt erfüllt werden kann

## Komplexität des Leerheitsproblems

Das Leerheitsproblem ist leicht zu lösen:

**Satz:** Die Leerheit eines NFA (oder auch DFA) kann in polynomieller Zeit entschieden werden.

**Beweis:** Leerheit = „Es gibt keinen von einem Startzustand aus erreichbaren Endzustand“

Erreichbarkeit von Endzuständen kann mithilfe von Graphalgorithmen zur Erreichbarkeitsprüfung (Breadth-First-Search, Depth-First-Search, ...) getestet werden. Einfache Algorithmen laufen in  $O(|Q| \times |\delta|)$ , wobei  $|Q|$  die Zahl der Zustände und  $|\delta|$  die Gesamtzahl der Kanten im Graph des Automaten ist.  $\square$

## Das Inklusionsproblem für DFA und NFA

Das **Inklusionsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** zwei FAs  $\mathcal{M}_1$  und  $\mathcal{M}_2$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $\mathbf{L}(\mathcal{M}_1) \subseteq \mathbf{L}(\mathcal{M}_2)$ ; andernfalls „nein“

**Anwendungsbeispiel:** Verifikation einer Systemspezifikation

- Spezifiziere erwünschtes Systemverhalten als Automat  $\mathcal{M}_{\text{spec}}$
- Abstrahiere tatsächliches Systemverhalten durch Automat  $\mathcal{M}_{\text{sys}}$
- Inklusionstest  $\mathbf{L}(\mathcal{M}_{\text{sys}}) \subseteq \mathbf{L}(\mathcal{M}_{\text{spec}})$ : Prüfe, ob System Spezifikation erfüllt

## Lösen des Inklusionsproblems

Wie kann man  $\mathbf{L}(\mathcal{M}_1) \subseteq \mathbf{L}(\mathcal{M}_2)$  testen?

**Reduktion auf Leerheitsproblem:**

- $\mathbf{L}(\mathcal{M}_1) \subseteq \mathbf{L}(\mathcal{M}_2)$  gdw für jedes  $w \in \mathbf{L}(\mathcal{M}_1)$  gilt  $w \in \mathbf{L}(\mathcal{M}_2)$
- gdw es gibt kein  $w \in \mathbf{L}(\mathcal{M}_1)$  mit  $w \notin \mathbf{L}(\mathcal{M}_2)$
- gdw es gibt kein  $w \in \mathbf{L}(\mathcal{M}_1) \cap (\Sigma^* \setminus \mathbf{L}(\mathcal{M}_2))$
- gdw  $\mathbf{L}(\mathcal{M}_1) \cap (\Sigma^* \setminus \mathbf{L}(\mathcal{M}_2)) = \emptyset$
- gdw  $\mathbf{L}(\mathcal{M}_1) \cap \overline{\mathbf{L}(\mathcal{M}_2)} = \emptyset$
- gdw  $\mathbf{L}(\mathcal{M}_1) \cap \mathbf{L}(\overline{\mathcal{M}_2}) = \emptyset$
- gdw  $\mathbf{L}(\mathcal{M}_1 \otimes \overline{\mathcal{M}_2}) = \emptyset$

$\leadsto$  kann effektiv überprüft werden

## Komplexität des Inklusionsproblems

Wie aufwändig ist es,  $L(M_1 \otimes \overline{M_2}) = \emptyset$  zu testen?

- Komplementbildung: linear für DFAs, exponentiell für NFAs (Determinisierung nötig!)
- Produktautomat: polynomiell
- Leerheitstest: polynomiell

Satz: Für NFAs  $M_1$  und  $M_2$  kann  $L(M_1) \subseteq L(M_2) \dots$

- in exponentieller Zeit entschieden werden
- in polynomieller Zeit entschieden werden, wenn  $M_2$  DFA ist

- Es ist bisher nicht bewiesen, dass es im NFA-Fall keinen besseren Algorithmus gibt
- Ergebnisse aus der Komplexitätstheorie legen das aber nahe  
(das Problem ist PSpace-vollständig)
- Obwohl der Worst Case bei jedem bekannten Verfahren exponentiell ist, gibt es praktische Algorithmen, die in vielen Fällen deutlich besser sind als das obige

## Komplexität des Äquivalenzproblems

Das Äquivalenzproblem kann auf das Inklusionsproblem reduziert werden:

$$L(M_1) = L(M_2) \text{ gdw. } L(M_1) \subseteq L(M_2) \text{ und } L(M_2) \subseteq L(M_1)$$

Daraus ergibt sich eine obere Schranke der Komplexität:

Satz: Für NFAs  $M_1$  und  $M_2$  kann  $L(M_1) = L(M_2)$

- in exponentieller Zeit entschieden werden
- in polynomieller Zeit entschieden werden, wenn  $M_1$  und  $M_2$  DFAs sind

Auch hier ist nicht bekannt, ob es für den Worst Case effizientere Algorithmen geben könnte, aber es gilt als sehr unwahrscheinlich

## Das Äquivalenzproblem für DFA und NFA

Das **Äquivalenzproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** zwei FAs  $M_1$  und  $M_2$  über Alphabet  $\Sigma$

**Ausgabe:** „ja“ wenn  $L(M_1) = L(M_2)$ ; andernfalls „nein“

**Anwendungsbeispiel:** Anfrageoptimierung in Graphdatenbanken

- Graphdatenbanken erlauben die Suche nach Pfaden, die regulären Ausdrücken entsprechen
- Einmal berechnete Ergebnisse werden zwischengespeichert (Cache)
- Äquivalenztest: Prüfe, ob eine gecachete Anfrage äquivalent zu einer neu eingehenden Anfrage ist

## Das Wortproblem für reguläre Sprachen

Satz: Das Wortproblem für DFAs kann in polynomieller Zeit entschieden werden.

**Beweis:** Es genügt, den DFA für  $|w|$  Schritte zu simulieren und zu prüfen, ob danach ein Endzustand erreicht ist. Die Berechnung von  $\delta(q, a)$  ist in polynomieller Zeit möglich – die Details hängen davon ab, wie genau  $M$  in der Eingabe kodiert wurde.  $\square$

Satz: Das Wortproblem für reguläre Sprachen kann in linearer Zeit  $O(|w|)$  entschieden werden.

**Beweis:** Man kann den DFA als gegebenen Bestandteil des Problems annehmen, der nicht Teil der Eingabe ist. Die Berechnung von  $\delta(q, a)$  kann daher in konstanter Zeit erfolgen (die Zeit hängt vom Automaten ab, aber nicht von der Eingabe). Die Simulation benötigt daher insgesamt  $|w|$  Rechenschritte.  $\square$

## Das Wortproblem für NFAs

Was tun, wenn ein NFA gegeben ist?

Variante 1: NFA in DFA umwandeln (Potenzmengenkonstruktion),

Wortproblem für DFA lösen

Exponentieller Algorithmus: Potenzmengen-DFA ist exponentiell groß

Variante 2: NFA direkt mit Zustandsmengen simulieren

(vergleichbar „on-the-fly Version von Variante 1“)

Polynomieller Algorithmus: Zustandsmengen sind von linearer Größe;

Berechnung der Nachfolgemenge als Vereinigung linear vieler

$\delta$ -Ergebnismengen

Variante 3: Konstruiere einen DFA  $\mathcal{M}_w$  mit  $\mathbf{L}(\mathcal{M}_w) = \{w\}$  und prüfe ob  $\mathcal{M} \cap \mathcal{M}_w \neq \emptyset$

Polynomieller Algorithmus:  $\mathcal{M}_w$  ist linear in  $|w|$ ; Schnittmengen-DFA ist

quadratisch groß; Leerheitstest ist polynomiell in dieser Größe

## Details: Variante 2

**Eingabe:** NFA  $\mathcal{M} = \langle Q, \Sigma, \delta, Q_0, F \rangle$  und Wort  $w$

**Ausgabe:** Ist  $w \in \mathbf{L}(\mathcal{M})$ ?

- Initialisiere  $Z := Q_0$
- Für jedes Symbol  $\sigma_i$  in  $w = \sigma_1 \cdots \sigma_{|w|}$ :  
Berechne  $Z := \bigcup_{q \in Z} \delta(q, \sigma_i)$
- Für alle  $q \in F$ :  
Falls  $q \in Z$ : das Ergebnis ist „ja“
- Falls kein  $q \in F \cap Z$  gefunden wurde: das Ergebnis ist „nein“

Alle Teilberechnungen können in polynomieller Zeit ausgeführt werden, sofern  $\mathcal{M}$  „vernünftig“ kodiert wird

## Das Wortproblem für NFAs

Was tun, wenn ein NFA gegeben ist?

Variante 1: NFA in DFA umwandeln (Potenzmengenkonstruktion),

Wortproblem für DFA lösen

Exponentieller Algorithmus: Potenzmengen-DFA ist exponentiell groß

Variante 2: NFA direkt mit Zustandsmengen simulieren

(vergleichbar „on-the-fly Version von Variante 1“)

Polynomieller Algorithmus: Zustandsmengen sind von linearer Größe;

Berechnung der Nachfolgemenge als Vereinigung linear vieler

$\delta$ -Ergebnismengen

Variante 3: Konstruiere einen DFA  $\mathcal{M}_w$  mit  $\mathbf{L}(\mathcal{M}_w) = \{w\}$  und prüfe ob

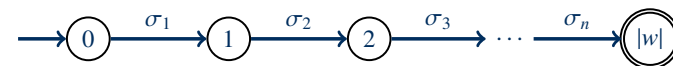
$\mathbf{L}(\mathcal{M}) \cap \mathbf{L}(\mathcal{M}_w) \neq \emptyset$

Polynomieller Algorithmus:  $\mathcal{M}_w$  ist linear in  $|w|$ ; Schnittmengen-DFA ist

quadratisch groß; Leerheitstest ist polynomiell in dieser Größe

## Details: Variante 3

Der Automat  $\mathcal{M}_w$  für  $w = \sigma_1 \cdots \sigma_{|w|}$  ist leicht gefunden:



**Eingabe:** NFA  $\mathcal{M} = \langle Q, \Sigma, \delta, Q_0, F \rangle$  und Wort  $w$

**Ausgabe:** Ist  $w \in \mathbf{L}(\mathcal{M})$ ?

- (1) Konstruiere  $\mathcal{M}_w$
- (2) Berechne Produktautomat  $\mathcal{M} \otimes \mathcal{M}_w$
- (3) Entscheide ob  $(\mathcal{M} \otimes \mathcal{M}_w) \neq \emptyset$

## Das Wortproblem für NFAs

Was tun, wenn ein NFA gegeben ist?

Variante 1: NFA in DFA umwandeln (Potenzmengenkonstruktion),  
Wortproblem für DFA lösen

Exponentieller Algorithmus: Potenzmengen-DFA ist exponentiell groß

Variante 2: NFA direkt mit Zustandsmengen simulieren  
(vergleichbar „on-the-fly Version von Variante 1“)

Polynomieller Algorithmus: Zustandsmengen sind von linearer Größe;  
Berechnung der Nachfolgemenge als Vereinigung linear vieler  
 $\delta$ -Ergebnismengen

Variante 3: Konstruiere einen DFA  $\mathcal{M}_w$  mit  $\mathbf{L}(\mathcal{M}_w) = \{w\}$  und prüfe ob  
 $\mathbf{L}(\mathcal{M}) \cap \mathbf{L}(\mathcal{M}_w) \neq \emptyset$

Polynomieller Algorithmus:  $\mathcal{M}_w$  ist linear in  $|w|$ ; Schnittmengen-DFA ist  
quadratisch groß; Leerheitstest ist polynomiell in dieser Größe

## Weitere Probleme für Automaten

Das **Endlichkeitsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein FA  $\mathcal{M}$

**Ausgabe:** „ja“ wenn  $\mathbf{L}(\mathcal{M})$  endlich ist; andernfalls „nein“

Idee wie Pumping-Lemma: unendliche Sprachen erfordern Zyklus

→ suche nach Zyklen, die auf einem Pfad von einem Start- zu einem Endzustand liegen  
(polynomiell)

Das **Universalitätsproblem** für FAs über Alphabet  $\Sigma$  besteht darin, die folgende Funktion zu berechnen:

**Eingabe:** ein FA  $\mathcal{M}$

**Ausgabe:** „ja“ wenn  $\mathbf{L}(\mathcal{M}) = \Sigma^*$ ; andernfalls „nein“

Komplement des Leerheitsproblems:  $\mathbf{L}(\mathcal{M}) = \Sigma^*$  wenn  $\mathbf{L}(\overline{\mathcal{M}}) = \emptyset$

→ Komplexität abhängig von FA-Komplementierung

## Wortproblem für NFAs – Komplexität

Variante 1: NFA in DFA umwandeln (Potenzmengenkonstruktion),  
Wortproblem für DFA lösen

Variante 2: NFA direkt mit Zustandsmengen simulieren  
(vergleichbar „on-the-fly Version von Variante 1“)

Variante 3: Konstruiere einen DFA  $\mathcal{M}_w$  mit  $\mathbf{L}(\mathcal{M}_w) = \{w\}$  und prüfe ob  
 $\mathbf{L}(\mathcal{M}) \cap \mathbf{L}(\mathcal{M}_w) \neq \emptyset$

Mit Variante 2 und 3 erhalten wir:

Satz: Das Wortproblem für NFAs kann in polynomieller Zeit entschieden werden.

## Zusammenfassung und Ausblick

Es gibt viele **nicht-reguläre Sprachen**, was man mit verschiedenen Methoden beweisen kann (Myhill-Nerode, Abschlusseigenschaften, Pumping)

$\{a^n b^n \mid n \geq 0\}$  ist **nicht regulär**, da FAs nicht zählen können

Algorithmische Probleme zu Automaten sind **Leerheit, Inklusion, Äquivalenz, Universalität, Endlichkeit** und das **Wortproblem**

Für DFAs können sie in polynomieller Zeit gelöst werden, für NFAs dagegen zum Teil nur in exponentieller

Offene Fragen:

- Wir haben schon so viel über reguläre Sprachen gelernt – könnten wir das bitte noch einmal alles zusammenfassen?
- Und was ist mit kontextfreien Sprachen?