



Nov.	17 18 19 20 21 22 23	Einführung
	24 25 26 27 28 29 30	Aussagenlogik
	31 1 2 3 4 5 6	Normalformen
	7 8 9 10 11 12 13	Resolution
Dez.	14 15 16 17 18 19 20	Kompaktheit
	21 22 23 24 25 26 27	DPLL, Satz von Cook
	28 29 30 1 2 3 4	Strukturen und FO
	5 6 7 8 9 10 11	Prädikatenlogik
	12 13 14 15 16 17 18	Komplexität von FO
Jan.	19 20 21 22 23 24 25	Weihnachten
	26 27 28 29 30 31 1	Neujahr
	2 3 4 5 6 7 8	Normalformen
	9 10 11 12 13 14 15	Definierbarkeit
	16 17 18 19 20 21 22	EF-Spiele
Feb.	23 24 25 26 27 28 29	EF-Spiele
	30 31 1 2 3 4 5	Sequenzkalkül AL
	6 7 8 9 10 11 12	Sequenzkalkül FO
	13 14 15 16 17 18 19	Ausblick

Wiederholung

Der aussagenlogische Resolutionskalkül

Theorem. Eine Menge \mathcal{C} von Klauseln hat genau dann eine Resolutionswiderlegung, wenn \mathcal{C} unerfüllbar ist. D.h. es gilt

$$\mathcal{C} \vdash_R \square \quad \text{gdw.} \quad \mathcal{C} \text{ ist unerfüllbar} \quad \text{gdw.} \quad \mathcal{C} \models \perp$$

Lemma. (Korrektheit des Resolutionskalküls)

Wenn eine Menge \mathcal{C} von Klauseln eine Resolutionswiderlegung hat, ist \mathcal{C} unerfüllbar.

Lemma. (Vollständigkeit des Resolutionskalküls)

Jede unerfüllbare Klauselmenge \mathcal{C} hat eine Resolutionswiderlegung.

Der Kompaktheitssatz der Aussagenlogik

Theorem. (Kompaktheits- oder Endlichkeitssatz)

Sei $\Phi \subseteq AL$ eine Formelmenge und $\psi \in AL$ eine Formel.

1. Φ ist erfüllbar gdw. jede endliche Teilmenge $\Phi' \subseteq \Phi$ erfüllbar ist.
2. $\Phi \models \psi$ gdw. eine endliche Teilmenge $\Phi_0 \subseteq \Phi$ existiert mit $\Phi_0 \models \psi$.

Bemerkung. Der Satz kann auch für überabzählbare Variablenmengen und somit überabzählbare Formelmengen bewiesen werden.

Erinnerung.

Eine Menge Φ aussagenlogischer Formeln ist **erfüllbar**, wenn es eine Belegung β gibt, die zu allen $\varphi \in \Phi$ passt und alle $\varphi \in \Phi$ erfüllt.

Wir schreiben $\beta \models \Phi$.

DPLL Algorithmen

Der Algorithmus arbeitet auf Formeln $\varphi := \bigwedge_{i=1}^n \bigvee_{j=1}^{n_i} L_{i,j}$ in KNF.

Repr. φ als Klauselmenge: $\Phi := \{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{n,1}, \dots, L_{n,n_n}\}\}$.

Basis Algorithmus DPLL(Φ, β)

Eingabe. Klauselmenge $\Phi := \{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{n,1}, \dots, L_{n,n_n}\}\}$

Partielle Belegung β mit $\text{def}(\beta) \subseteq \text{var}(\Phi)$.

Ausgabe. $\beta' \supseteq \beta$ mit $\beta' \models \Phi$ oder unerfüllbar, wenn kein β' existiert.

Algorithmus. Wenn $\llbracket \Phi \rrbracket^\beta = 1$, gib β zurück.

Wenn $\llbracket \Phi \rrbracket^\beta = 0$, gib unerfüllbar zurück.

Sonst (d.h. wenn der Wert von Φ unter β unbestimmt ist)

reduce(β, Φ);

branch(β); Wähle unbelegte Variable X aus $\text{var}(\Phi)$ und Wahrheitswert t .

$\beta' := \text{DPLL}(\Phi, \beta[X \mapsto t])$.

wenn $\beta' \neq \text{unerfüllbar}$, gibt β' zurück

sonst gib $\text{DPLL}(\Phi, \beta[X \mapsto 1 - t])$ zurück.

Einheitsresolution

Die Funktion $\text{reduce}(\beta, \Phi)$ dient der Reduktion der Formel nach jedem Schritt.

Meistens wird nur Einheitsresolution verwendet.

Unit Clause Propagation.

Enthält Φ Klausel $C = \{L_1, \dots, L_k\}$ in der alle bis auf 1 Literal L_i durch β belegt werden, so muss jede erfüllende Belegung $\beta' \supseteq \beta$ das Literal L_i mit 1 belegen.

Ist $L_i = X$, können wir also direkt $[X \mapsto 1]$ zu β hinzufügen.

Ist $L_i = \neg X$, fügen wir $[X \mapsto 0]$ zu β hinzu.

Dies wird solange wiederholt, bis es keine Einer-Klauseln mehr gibt.

Basis DPLL(Φ, β)

Ausgabe.

$\beta' \supseteq \beta$ mit $\beta' \models \Phi$ oder unerf.

Algorithmus.

$\llbracket \Phi \rrbracket^\beta = 1 \rightsquigarrow \text{return } \beta$.

$\llbracket \Phi \rrbracket^\beta = 0 \rightsquigarrow \text{return unerf.}$

Sonst

$\text{reduce}(\beta, \Phi);$

$\text{branch}(\beta):$

Wähle $X \in \text{var}(\Phi)$, $t \in \{0, 1\}$.

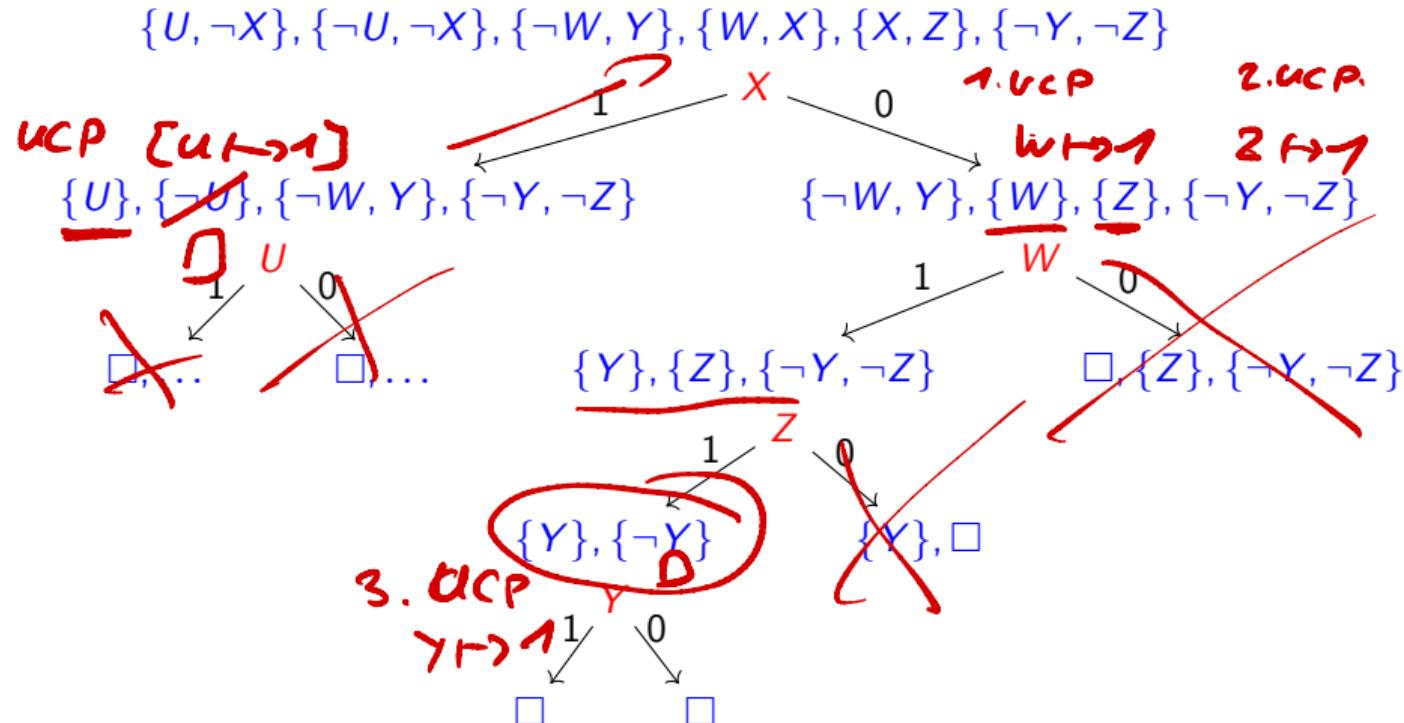
$\beta' := \text{DPLL}(\Phi, \beta[X \mapsto t]).$

wenn $\beta' \models \Phi$, return β'

sonst

return $\text{DPLL}(\Phi, \beta[X \mapsto 1 - t])$

Beispiel für den DPLL-Algorithmus



Der DPLL-Algorithmus

In praktischen Implementierungen des Algorithmus' werden verschiedene Optimierungen verwendet.

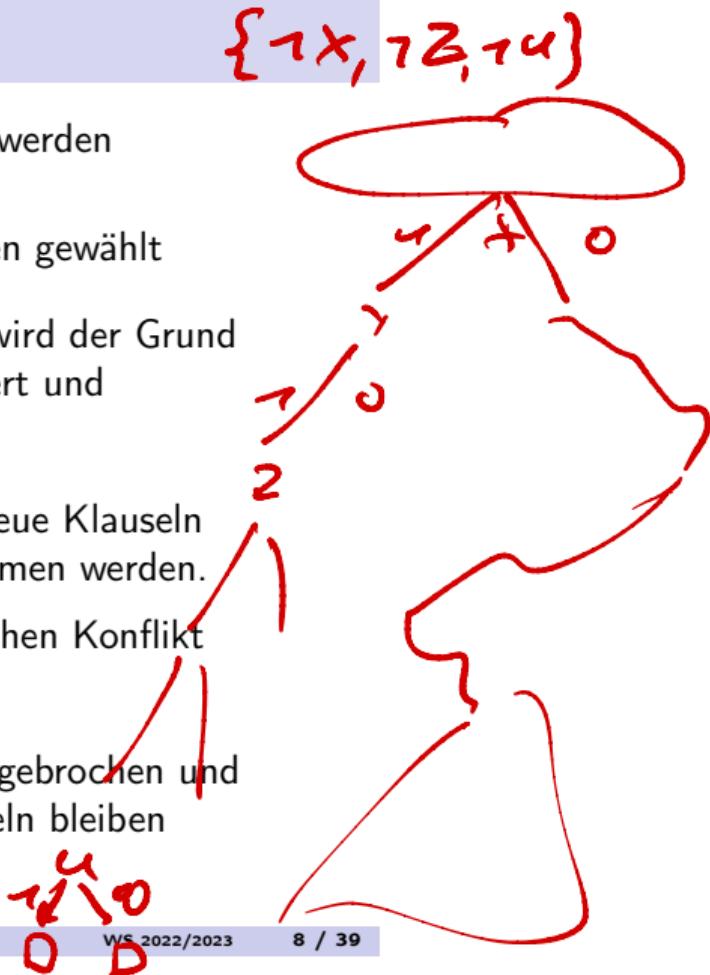
Auswahlregel: Welches Literal wird beim Verzweigen gewählt

Conflict Analysis: Bei einem Backtracking Schritt wird der Grund der Unerfüllbarkeit (Conflict) analysiert und intelligenter zurück gesprungen.

Clause Learning: Aus der Konfliktanalyse werden neue Klauseln generiert, die zur Formel hinzugenommen werden.

Dies soll verhindern, dass in den gleichen Konflikt hineingelaufen wird.

Random restarts: Bisweilen wird ein DPLL-Lauf abgebrochen und neu angefangen. Die gelernten Klauseln bleiben erhalten.



Der Satz von Cook: NP-Vollständigkeit von SAT

NP-Vollständigkeit von SAT

Definition.

Das aussagenlogische Erfüllbarkeitsproblem (SAT) ist das Problem

SAT

Eingabe. Eine aussagenlogische Formel $\varphi \in AL$

Problem. Entscheide, ob φ erfüllbar ist.

Satz.

(Cook 1970, Levin 1973)

SAT ist NP-vollständig.

6.1 NP-Vollständigkeit

Schwere und leichte Probleme

3-Färbbarkeit

Definition (3-Färbbarkeit).

Ein Graph G ist **3-färbbar**, wenn es eine Funktion

$$c : V(G) \rightarrow \{C_1, C_2, C_3\}$$

gibt, so dass

$$c(u) \neq c(v) \text{ für alle Kanten } \{u, v\} \in E(G).$$

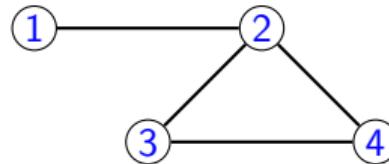
Wir nennen $c : V(G) \rightarrow \{1, 2, 3\}$ eine **3-Färbung** von G .

Das Problem **3-COL**. Das zugehörige Entscheidungsproblem:

3-COL

Eingabe. Graph G .

Problem. Entscheide, ob G 3-färbbar ist.



3-Färbbarkeit

Definition (3-Färbbarkeit).

Ein Graph G ist **3-färbbar**, wenn es eine Funktion

$$c : V(G) \rightarrow \{C_1, C_2, C_3\}$$

gibt, so dass

$$c(u) \neq c(v) \text{ für alle Kanten } \{u, v\} \in E(G).$$

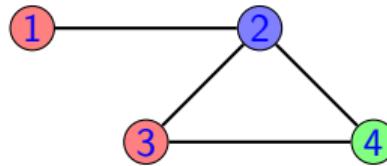
Wir nennen $c : V(G) \rightarrow \{1, 2, 3\}$ eine **3-Färbung** von G .

Das Problem **3-COL**. Das zugehörige Entscheidungsproblem:

3-COL

Eingabe. Graph G .

Problem. Entscheide, ob G 3-färbbar ist.



Leicht oder schwer?

Entscheidungsproblem 3-COL.

3-COL.

Eingabe. Ein Graph G .

Ausgabe. 1, wenn G 3-färbbar ist, sonst 0.

3-Färbung.

$$c : V(G) \rightarrow \{C_1, C_2, C_3\}$$

mit

$$c(u) \neq c(v)$$

für alle $\{u, v\} \in E(G)$.

Frage. Wie schwer (algorithmisch gesehen) ist es, das Problem zu lösen?

Kann man bei einer Eingabe G mit n Knoten z.B. in

- Linearzeit $O(n)$ lösen?
- polynomieller Zeit $O(n^d)$ lösen, für eine Konstante d ?
- in exponentieller Zeit $2^{O(n)}$ lösen?
- oder in Zeit $O(2^{2^{2^{2^{2^n}}}})$? Oder gar $O\left(2^{\cdot^{\cdot^{\cdot^2}}} \right) n$?

Schummeln ist nicht.

Überprüfen einer vorgeschlagenen Lösung für $3\text{-COL}(G)$.

Sei $c : V(G) \rightarrow \{C_1, C_2, C_3\}$ eine Funktion.

Wir können sehr leicht überprüfen, ob c eine 3-Färbung von G ist:

Überprüfe für jede Kante $\{u, v\} \in E(G)$, ob $c(u) \neq c(v)$.

Schummeln ist nicht.

Überprüfen einer vorgeschlagenen Lösung für $3\text{-COL}(G)$.

Sei $c : V(G) \rightarrow \{C_1, C_2, C_3\}$ eine Funktion.

Wir können sehr leicht überprüfen, ob c eine 3-Färbung von G ist:

Überprüfe für jede Kante $\{u, v\} \in E(G)$, ob $c(u) \neq c(v)$.

Zahl der möglichen Lösungen.

Es gibt $3^{|V(G)|}$ mögliche Funktionen $c : V(G) \rightarrow \{C_1, C_2, C_3\}$.

Wir können also alle Funktionen nacheinander überprüfen.

Dies liefert einen Algorithmus, der 3-COL in Zeit $3^{O(n)} \cdot n^c$ löst.

Schummeln ist nicht.

Überprüfen einer vorgeschlagenen Lösung für $3\text{-COL}(G)$.

Sei $c : V(G) \rightarrow \{C_1, C_2, C_3\}$ eine Funktion.

Wir können sehr leicht überprüfen, ob c eine 3-Färbung von G ist:

Überprüfe für jede Kante $\{u, v\} \in E(G)$, ob $c(u) \neq c(v)$.

Zahl der möglichen Lösungen.

Es gibt $3^{|V(G)|}$ mögliche Funktionen $c : V(G) \rightarrow \{C_1, C_2, C_3\}$.

Wir können also alle Funktionen nacheinander überprüfen.

Dies liefert einen Algorithmus, der 3-COL in Zeit $3^{O(n)} \cdot n^c$ löst.

Offenes Problem. Aber geht es auch in polynomieller Zeit?

Komplexitätsklassen

Komplexitätsklassen.

Eine Komplexitätsklasse \mathcal{C} ist eine Klasse algorithmischer Probleme.

Formal: Klasse von Sprachen über einem Alphabet.

Intuitiv: Klasse von Problemen, die mit Hilfe eines bestimmten Ressourceneinsatzes gelöst werden können.

Beispiel.

Die Klasse EXP aller Probleme P die bei Eingaben der Größe n in Zeit $2^{O(n^d)}$, für ein $d \geq 1$, gelöst werden können.

Wir haben gesehen, dass $3\text{-COL} \in \text{EXP}$.

Komplexitätsklassen

Komplexitätsklassen.

Eine Komplexitätsklasse \mathcal{C} ist eine Klasse algorithmischer Probleme.

Formal: Klasse von Sprachen über einem Alphabet.

Die Klasse “PTIME”. Klasse aller Probleme, die durch einen Algorithmus gelöst werden können, dessen Laufzeit auf Eingaben der Länge n durch $p(n)$ beschränkt ist.

Hierbei ist $p(x)$ ein festes Polynom (unabhängig von n).

Formal ist **PTIME** die Klasse aller Sprachen, die von einer **deterministischen** Turing-Maschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.

Komplexitätsklassen

Komplexitätsklassen.

Eine Komplexitätsklasse \mathcal{C} ist eine Klasse algorithmischer Probleme.

Formal: Klasse von Sprachen über einem Alphabet.

Die Klasse “PTIME”. Klasse aller Probleme, die durch einen Algorithmus gelöst werden können, dessen Laufzeit auf Eingaben der Länge n durch $p(n)$ beschränkt ist.

Hierbei ist $p(x)$ ein festes Polynom (unabhängig von n).

Formal ist PTIME die Klasse aller Sprachen, die von einer **deterministischen** Turing-Maschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.

Bemerkung. Probleme in PTIME werden meistens durch explizite Konstruktion oder Berechnung der Lösung gelöst.

Komplexitätsklassen

Komplexitätsklassen.

Eine Komplexitätsklasse \mathcal{C} ist eine Klasse algorithmischer Probleme.

Formal: Klasse von Sprachen über einem Alphabet.

Die Klasse "PTIME". Klasse aller Probleme, die durch einen Algorithmus gelöst werden können, dessen Laufzeit auf Eingaben der Länge n durch $p(n)$ beschränkt ist.

Hierbei ist $p(x)$ ein festes Polynom (unabhängig von n).

Formal ist PTIME die Klasse aller Sprachen, die von einer **deterministischen** Turing-Maschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.

Beispiele.

- Berechnen maximaler Flüsse in Netzwerken.
- 2-Färbbarkeit von Graphen.
- Das Auswertungsproblem der Aussagenlogik.

Bemerkung. Probleme in PTIME werden meistens durch explizite Konstruktion oder Berechnung der Lösung gelöst.

Komplexitätsklassen

Die Klasse "NP". Klasse aller Probleme,

- die von einer *nicht-deterministischen* Turingmaschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.
- bei denen
 1. die *Größe einer Lösung* für eine Eingabe der Größe n *polynomiell* beschränkt ist und
 2. für die eine vorgeschlagene Lösung in *Polynomialzeit* überprüft werden kann.

D.h. wir können für eine Eingabe x eine Lösung raten und diese dann in polynomieller Zeit (in $|x|$) verifizieren.

Komplexitätsklassen

Die Klasse "NP". Klasse aller Probleme,

- die von einer *nicht-deterministischen* Turingmaschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.
- bei denen
 1. die *Größe einer Lösung* für eine Eingabe der Größe n *polynomiell* beschränkt ist und
 2. für die eine vorgeschlagene Lösung in *Polynomialzeit* überprüft werden kann.

D.h. wir können für eine Eingabe x eine Lösung raten und diese dann in polynomieller Zeit (in $|x|$) verifizieren.

Bemerkung.

Probleme in *NP* werden meistens durch Suchverfahren gelöst.

D.h. man berechnet die Lösung nicht, sondern sucht geschickt nach einer Lösung und überprüft, dass diese auch stimmt.

Komplexitätsklassen

Die Klasse "NP". Klasse aller Probleme,

- die von einer *nicht-deterministischen* Turingmaschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.
- bei denen
 1. die *Größe einer Lösung* für eine Eingabe der Größe n *polynomiell* beschränkt ist und
 2. für die eine vorgeschlagene Lösung in *Polynomialzeit* überprüft werden kann.

D.h. wir können für eine Eingabe x eine Lösung raten und diese dann in polynomieller Zeit (in $|x|$) verifizieren.

Bemerkung.

Probleme in *NP* werden meistens durch Suchverfahren gelöst.

D.h. man berechnet die Lösung nicht, sondern sucht geschickt nach einer Lösung und überprüft, dass diese auch stimmt.

Beispiele.

- Das SAT-Problem ist in NP.

Eingabe: $\varphi(X_1, \dots, X_n) \in \text{AL}$.

Gesucht: Belegung β von X_1, \dots, X_n mit $[\![\varphi]\!]^\beta = 1$

Größe von β polynomiell in Größe von φ .

Wir können in polynomieller Zeit überprüfen, ob $[\![\varphi]\!]^\beta = 1$

Komplexitätsklassen

Die Klasse "NP". Klasse aller Probleme,

- die von einer *nicht-deterministischen* Turingmaschine mit polynomieller Zeitschranke $p(n)$ gelöst werden können.
- bei denen
 1. die *Größe einer Lösung* für eine Eingabe der Größe n *polynomiale* beschränkt ist und
 2. für die eine vorgeschlagene Lösung in *Polynomialzeit* überprüft werden kann.

D.h. wir können für eine Eingabe x eine Lösung raten und diese dann in polynomieller Zeit (in $|x|$) verifizieren.

Bemerkung.

Probleme in *NP* werden meistens durch Suchverfahren gelöst.

D.h. man berechnet die Lösung nicht, sondern sucht geschickt nach einer Lösung und überprüft, dass diese auch stimmt.

Beispiele.

- Das SAT-Problem ist in NP.
- 3-COL.
- Hat G ein dominating set der Größe k ?
- Hat G einen Hamiltonkreis?
- Set Cover
- Rucksackprobleme
- Constraint Satisfaction Problems
- Integer Programming

NP-Vollständigkeit

Die Komplexitätsklassen P und NP. Die Probleme in P können (im Prinzip) effizient gelöst werden, d.h. in Polynomialzeit. Bei Problemen in NP wissen (und glauben) wir das nicht.

NP-Vollständigkeit

Die Komplexitätsklassen P und NP. Die Probleme in P können (im Prinzip) effizient gelöst werden, d.h. in Polynomialzeit. Bei Problemen in NP wissen (und glauben) wir das nicht.

Wir können aber Probleme in NP lösen, indem wir eine kurze, d.h. polynomiell große, Lösung raten und dann effizient überprüfen.

NP-Vollständigkeit

Die Komplexitätsklassen P und NP. Die Probleme in P können (im Prinzip) effizient gelöst werden, d.h. in Polynomialzeit. Bei Problemen in NP wissen (und glauben) wir das nicht.

Wir können aber Probleme in NP lösen, indem wir eine kurze, d.h. polynomiell große, Lösung raten und dann effizient überprüfen.

$P \subseteq NP$. Man sieht sofort, dass jedes Problem in P auch in NP liegt.

Es sind also nicht alle Probleme in NP „schwer“.



NP-Vollständigkeit

Die Komplexitätsklassen **P** und **NP**. Die Probleme in **P** können (im Prinzip) effizient gelöst werden, d.h. in Polynomialzeit. Bei Problemen in **NP** wissen (und glauben) wir das nicht.

Wir können aber Probleme in **NP** lösen, indem wir eine kurze, d.h. polynomiell große, Lösung **raten** und dann effizient **überprüfen**.

P \subseteq **NP**. Man sieht sofort, dass jedes Problem in **P** auch in **NP** liegt.

Es sind also nicht *alle* Probleme in **NP** „schwer“.

Woran erkennt man die „schweren“ Probleme in **NP**?

NP-Vollständigkeit

Die Komplexitätsklassen **P** und **NP**. Die Probleme in **P** können (im Prinzip) effizient gelöst werden, d.h. in Polynomialzeit. Bei Problemen in **NP** wissen (und glauben) wir das nicht.

Wir können aber Probleme in **NP** lösen, indem wir eine kurze, d.h. polynomiell große, Lösung **raten** und dann effizient **überprüfen**.

P \subseteq **NP**. Man sieht sofort, dass jedes Problem in **P** auch in **NP** liegt.

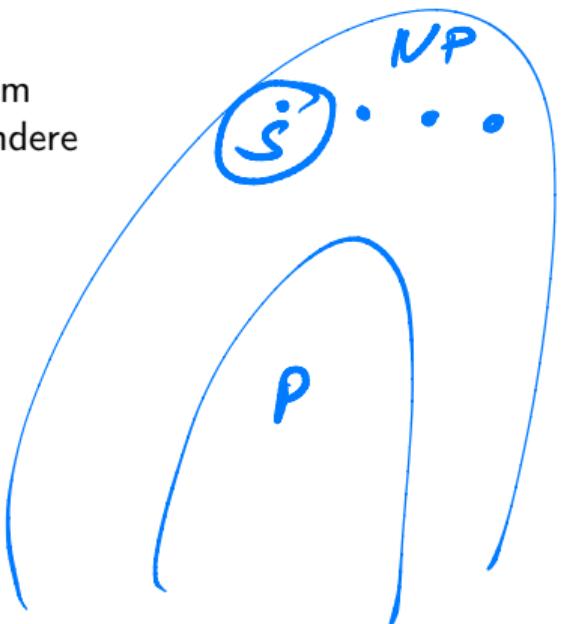
Es sind also nicht *alle* Probleme in **NP** „schwer“.

Woran erkennt man die „schweren“ Probleme in **NP**? **Gar nicht**

NP-Vollständigkeit

Definition.

Ein Problem L in NP ist *NP-vollständig*, wenn man mit einem Polynomialzeitalgorithmus für L als Unterprogramm jedes andere Problem in NP auch in Polynomialzeit lösen könnte.



NP-Vollständigkeit

Definition.

Ein Problem L in NP ist *NP-vollständig*, wenn man mit einem Polynomialzeitalgorithmus für L als Unterprogramm jedes andere Problem in NP auch in Polynomialzeit lösen könnte.

Wenn also ein einziges *NP-vollständiges* Problem in Polynomialzeit gelöst werden kann, dann auch alle anderen NP -Probleme, d.h. es gilt $P = NP$.

Beispiele NP-vollständiger Probleme

Wie zeigt man NP-Vollständigkeit?

Um zu beweisen, dass ein Problem L NP-vollständig ist, müssen wir folgendes zeigen:

1. $L \in \text{NP}$ und
2. falls L in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Beispiele NP-vollständiger Probleme

Wie zeigt man NP-Vollständigkeit?

Um zu beweisen, dass ein Problem L NP-vollständig ist, müssen wir folgendes zeigen:

1. $L \in \text{NP}$ und
2. falls L in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Reduktionen. Wenn man schon ein NP-vollständiges Problem M kennt, dann reicht es zu zeigen, dass ein Polynomialzeitalgorithmus für L benutzt werden kann, um M in Polynomialzeit zu lösen.

Nur wo bekommt man das „erste“ NP-vollständige Problem her?

Beispiele NP-vollständiger Probleme

Wie zeigt man NP-Vollständigkeit?

Um zu beweisen, dass ein Problem L NP-vollständig ist, müssen wir folgendes zeigen:

1. $L \in \text{NP}$ und
2. falls L in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Reduktionen. Wenn man schon ein NP-vollständiges Problem M kennt, dann reicht es zu zeigen, dass ein Polynomialzeitalgorithmus für L benutzt werden kann, um M in Polynomialzeit zu lösen.

Nur wo bekommt man das „erste“ NP-vollständige Problem her?

Die Logik zur Rettung. Dies gelang Ende der 60er Jahre *Stephen Cook* für das SAT Problem, der für seine bahnbrechenden Arbeiten zur NP-Vollständigkeit den Turing-Award erhielt.

Beispiele NP-vollständiger Probleme

Wie zeigt man NP-Vollständigkeit?

Um zu beweisen, dass ein Problem L NP-vollständig ist, müssen wir folgendes zeigen:

1. $L \in \text{NP}$ und $\varphi_{\text{GAL}} \leq_p L$
 2. falls L in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.
- $SAT \leq_p \varphi_{\text{GAL}}$

$\varphi_{\text{GAL}} \leq_p SAT$

Reduktionen. Wenn man schon ein NP-vollständiges Problem M kennt, dann reicht es zu zeigen, dass ein Polynomialzeitalgorithmus für L benutzt werden kann, um M in Polynomialzeit zu lösen.

Nur wo bekommt man das „erste“ NP-vollständige Problem her?

Die Logik zur Rettung. Dies gelang Ende der 60er Jahre **Stephen Cook** für das SAT Problem, der für seine bahnbrechenden Arbeiten zur NP-Vollständigkeit den Turing-Award erhielt.

Stephen Cook



(c) Jiří Janíček, CC-BY-SA-3.0

Richard Karp



(c) Rama,
Cc-by-sa-2.0-fr

Alan Turing



6.2 Der Satz von Cook

NP-Vollständigkeit von SAT

Definition.

Das aussagenlogische Erfüllbarkeitsproblem (SAT) ist das Problem

SAT

Eingabe. Eine aussagenlogische Formel $\varphi \in AL$

Problem. Entscheide, ob φ erfüllbar ist.

Satz.

(Cook 1970, Levin 1973)

SAT ist NP-vollständig.

NP-Vollständigkeit von SAT

Definition.

Das aussagenlogische Erfüllbarkeitsproblem (SAT) ist das Problem

SAT

Eingabe. Eine aussagenlogische Formel $\varphi \in AL$

Problem. Entscheide, ob φ erfüllbar ist.

Satz.

(Cook 1970, Levin 1973)

SAT ist NP-vollständig.

NP-Vollständigkeit. Um zu beweisen, dass SAT NP-vollständig ist, müssen wir folgendes zeigen:

1. $SAT \in NP$ und
2. falls SAT in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Beweis des Satzes von Cook

Satz. SAT ist NP-vollständig.

(Cook 1970, Levin 1973)

NP-Vollständigkeit. Um zu beweisen, dass SAT NP-vollständig ist, müssen wir folgendes zeigen:

1. $\text{SAT} \in \text{NP}$
2. falls SAT in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Beweis des Satzes von Cook

Satz. SAT ist NP-vollständig.

(Cook 1970, Levin 1973)

NP-Vollständigkeit. Um zu beweisen, dass SAT NP-vollständig ist, müssen wir folgendes zeigen:

1. $\text{SAT} \in \text{NP}$
2. falls SAT in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

$\text{SAT} \in \text{NP}$.

Eingabe: $\varphi(X_1, \dots, X_n) \in \text{AL}$.

Gesucht: Belegung β von X_1, \dots, X_n mit $\llbracket \varphi \rrbracket^\beta = 1$

Größe von β polynomiell in Größe von φ . *Bool*

Wir können in polynomieller Zeit überprüfen, ob $\llbracket \varphi \rrbracket^\beta = 1$ *Bool L* *NP*

Beweis des Satzes von Cook

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

Beweis des Satzes von Cook

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

1. Sei $L \in \text{NP}$. Dann gibt es ein Polynom $p(n)$ und eine *nicht-deterministische* TM M , die L in Zeit $O(p(n))$ löst.

Beweis des Satzes von Cook

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

1. Sei $L \in \text{NP}$. Dann gibt es ein Polynom $p(n)$ und eine *nicht-deterministische* TM M , die L in Zeit $O(p(n))$ löst.
2. Für jede feste NTM M und jedes Polynom $p(n)$, zeigt man nun:

Zu jedem Eingabewort w der Länge n existiert eine Formel φ_w mit:

M akzeptiert w in Zeit $p(n)$ gdw. φ_w erfüllbar ist.

φ_w kann in Zeit $O(n^d)$ konstruiert werden, für ein festes $d = d(M, p)$.

Beweis des Satzes von Cook

Eins: $\text{SAT} \sim_{\text{P}} \varphi_L$

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

Eins: w

Probleme? Welche?

e. $\varphi_L \text{ GEGEN}$

- Sei $L \in \text{NP}$. Dann gibt es ein Polynom $p(n)$ und eine *nicht-deterministische* TM M , die L in Zeit $O(p(n))$ löst.

- Für jede feste NTM M und jedes Polynom $p(n)$, zeigt man nun:

Zu jedem Eingabewort w der Länge n existiert eine Formel φ_w mit:
 M akzeptiert w in Zeit $p(n)$ gdw. φ_w erfüllbar ist.

Alg: Eingabe w

1. Konstruiere φ_w
 2. teste φ_w CSAT

Wenn ja, dann NT
 $w \in L$
 sonst nicht.

φ_w kann in Zeit $O(n^d)$ konstruiert werden, für ein festes $d = d(M, p)$.

- Wenn man also SAT in pol. Zeit lösen könnte, dann könnte man auch in pol. Zeit entscheiden, ob M eine Eingabe w der Länge n in Zeit $p(n)$ akzeptiert. Also könnte man L in pol. Zeit entscheiden.

6.3 Das Wortproblem endlicher Automaten als SAT-Problem

Das Wortproblem regulärer Sprachen

Beispiel. Wir betrachten das Wortproblem regulärer Sprachen.

Sei Σ ein Alphabet.

Sei $\mathcal{A} := (Q, \Sigma, q_0, \Delta, F)$ ein endlicher, nicht-deterministischer Automat.

Wortproblem für \mathcal{A} .

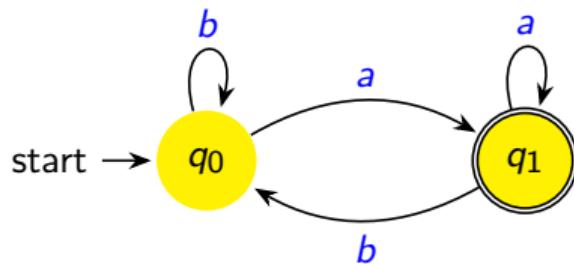
Eingabe. $w \in \Sigma^*$

Ausgabe. 1 wenn \mathcal{A} das Wort w akzeptiert, sonst 0.

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



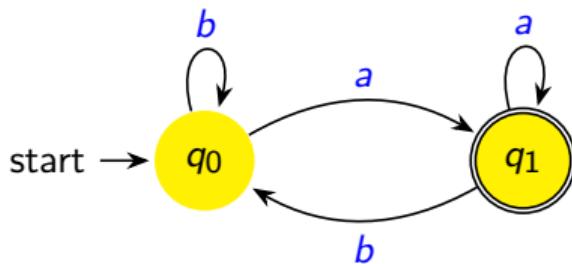
Wort w : a b a b a

Lauf von \mathcal{A} :

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



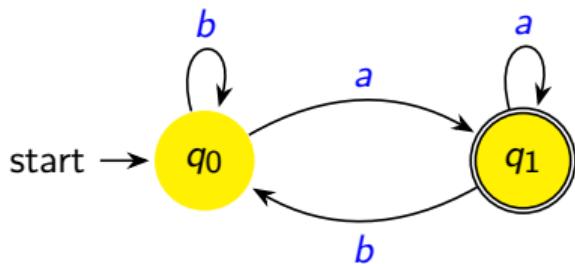
Wort w : a b a b a

Lauf von \mathcal{A} : q_0

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



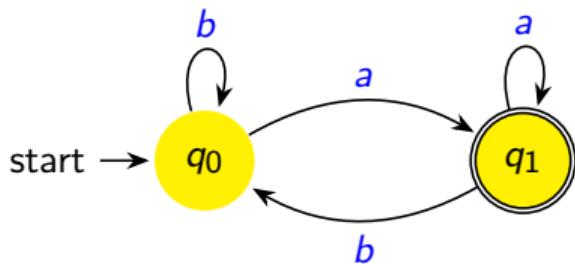
Wort w : a b a b a

Lauf von \mathcal{A} : q_0 q_1

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



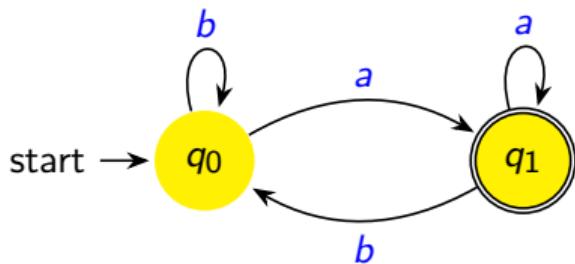
Wort w : $a \quad b \quad a \quad b \quad a$

Lauf von \mathcal{A} : $q_0 \quad q_1 \quad q_0$

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



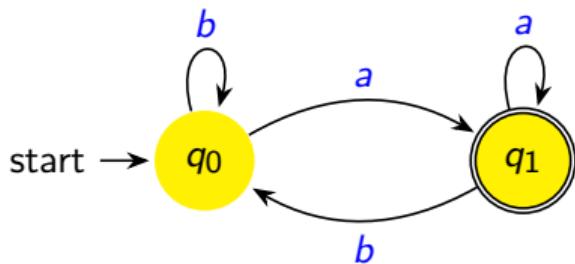
Wort w : a b a b a

Lauf von \mathcal{A} : q_0 q_1 q_0 q_1

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



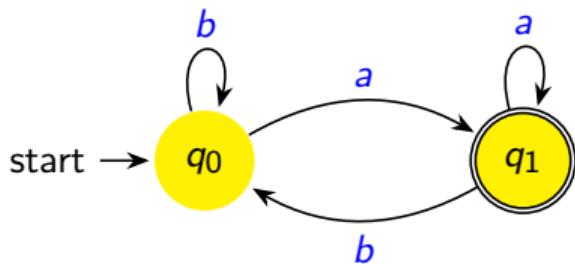
Wort w : $a \quad b \quad a \quad b \quad a$

Lauf von \mathcal{A} : $q_0 \quad q_1 \quad q_0 \quad q_1 \quad q_0$

Beispiel für das Wortproblem

Beispiel

Sei $Q := \{q_0, q_1\}$ und $F := \{q_1\}$. Sei $\Sigma := \{a, b\}$.



Wort $w :$ a b a b a

Lauf von $\mathcal{A} :$ q_0 q_1 q_0 q_1 q_0 q_1

Das Wortproblem regulärer Sprachen

Beispiel. Wir betrachten das Wortproblem regulärer Sprachen.

Sei Σ ein Alphabet.

Sei $\mathcal{A} := (Q, \Sigma, q_0, \Delta, F)$ ein endlicher, nicht-deterministischer Automat.

Wortproblem für \mathcal{A} .

Eingabe. $w \in \Sigma^*$

Ausgabe. 1 wenn \mathcal{A} das Wort w akzeptiert, sonst 0.

Das Wortproblem regulärer Sprachen

Beispiel. Wir betrachten das Wortproblem regulärer Sprachen.

Sei Σ ein Alphabet.

Sei $\mathcal{A} := (Q, \Sigma, q_0, \Delta, F)$ ein endlicher, nicht-deterministischer Automat.

Wortproblem für \mathcal{A} .

Eingabe. $w \in \Sigma^*$

Ausgabe. 1 wenn \mathcal{A} das Wort w akzeptiert, sonst 0.

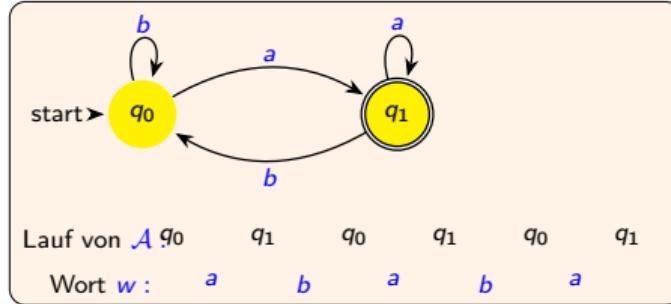
Reduktion auf SAT. Wir wollen das Problem auf SAT reduzieren.

Zu jedem $w \in \Sigma^*$ müssen wir eine Formel $\varphi_w \in \text{AL}$ konstruieren, so dass

\mathcal{A} akzeptiert w gdw. φ_w ist erfüllbar.

Formalisierung in der Aussagenlogik

Reduktion auf SAT. Zu jedem $w \in \Sigma^*$ müssen wir eine Formel $\varphi_w \in \text{AL}$ konstruieren, so dass \mathcal{A} akzeptiert w gdw. φ_w ist erfüllbar.

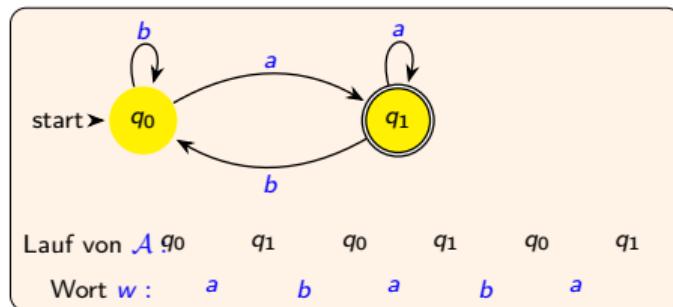


Formalisierung in der Aussagenlogik

Reduktion auf SAT. Zu jedem $w \in \Sigma^*$ müssen wir eine Formel $\varphi_w \in \text{AL}$ konstruieren, so dass \mathcal{A} akzeptiert w gdw. φ_w ist erfüllbar.

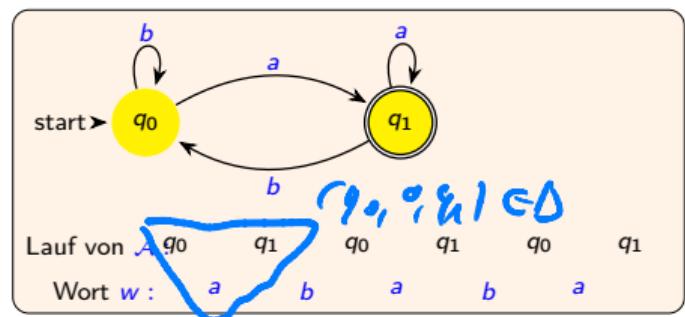
Idee. Die Formel φ_w soll so aus w und \mathcal{A} konstruiert werden, dass

- eine Belegung β von $\text{var}(\varphi_w)$ einem möglichen Lauf $\rho(\beta)$ von \mathcal{A} auf w entspricht und
- wenn $\beta \models \varphi_w$, dann ist $\rho(\beta)$ ein akzeptierender Lauf von \mathcal{A} auf w .



Umsetzung in der Aussagenlogik

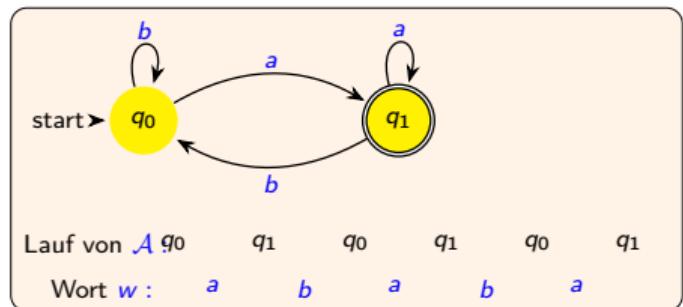
- Wir müssen das Wort $w = a_1, \dots, a_n$ durch Variablen kodieren.
- Wir brauchen Variablen, die bestimmen, in welchem Zustand der Automat nach dem i -ten Buchstaben ist.



Umsetzung in der Aussagenlogik

- Wir müssen das Wort $w = a_1, \dots, a_n$ durch Variablen kodieren.
- Wir brauchen Variablen, die bestimmen, in welchem Zustand der Automat nach dem i -ten Buchstaben ist.
- Die Formel muss sicherstellen, dass die so kodierte Zustandsfolge auch mit dem Automat \mathcal{A} übereinstimmt.

D.h. wenn eine Belegung der Variablen behauptet, dass \mathcal{A} nach dem i -ten Buchstaben im Zustand q ist und nach dem $i+1$ -ten im Zustand q' , dann muss $(q, a_{i+1}, q') \in \Delta$ sein.

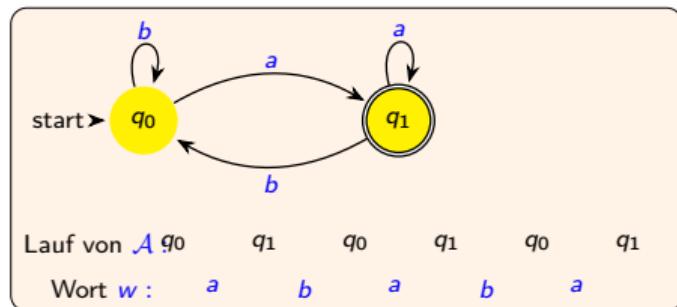


Umsetzung in der Aussagenlogik

- Wir müssen das Wort $w = a_1, \dots, a_n$ durch Variablen kodieren.
- Wir brauchen Variablen, die bestimmen, in welchem Zustand der Automat nach dem i -ten Buchstaben ist.
- Die Formel muss sicherstellen, dass die so kodierte Zustandsfolge auch mit dem Automat \mathcal{A} übereinstimmt.

D.h. wenn eine Belegung der Variablen behauptet, dass \mathcal{A} nach dem i -ten Buchstaben im Zustand q ist und nach dem $i+1$ -ten im Zustand q' , dann muss $(q, a_{i+1}, q') \in \Delta$ sein.

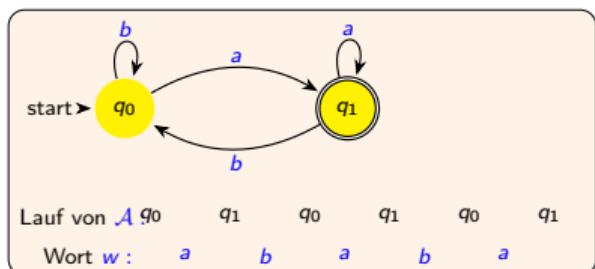
- Und dann muss der letzte Zustand noch in F liegen.



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

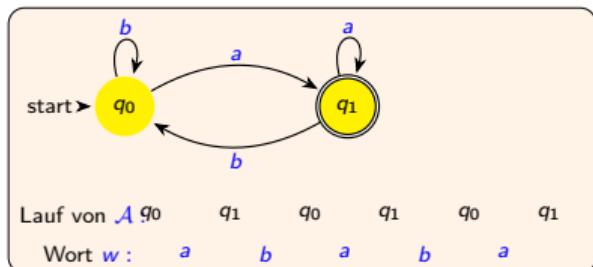
- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

\uparrow_{q_1}

s_4^a
 s_5^b



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“

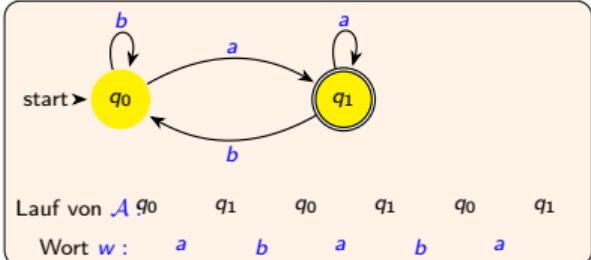
$$\varphi_{w, \text{Eingabe}} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$$

$$S_1^a \wedge \bigwedge_{c \in \Sigma \setminus \{a\}} \neg S_1^c$$

$$S_1^b \wedge \neg S_1^a$$

Variablen.

Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1



$$q_1 > q_0$$

Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

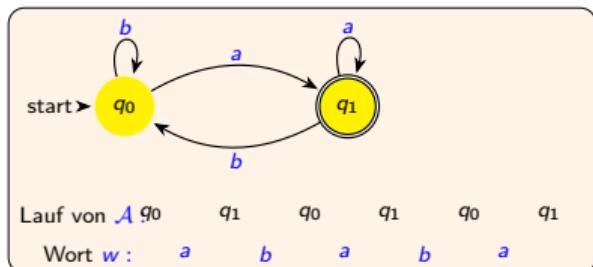
- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“
 $\varphi_{w,Eingabe} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$
- „Nach jedem Buchstaben ist der Automat in genau einem Zustand“
 $\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

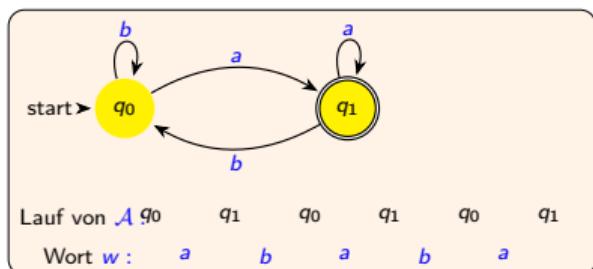
Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“
 $\varphi_{w,Eingabe} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$
- „Nach jedem Buchstaben ist der Automat in genau einem Zustand“
 $\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$
- „Zustandsübergänge kodieren gültigen Lauf“

$$\varphi_{w,Lauf} := Q_0^{q_0} \wedge$$

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“
 $\varphi_{w,Eingabe} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$

- „Nach jedem Buchstaben ist der Automat in genau einem Zustand“

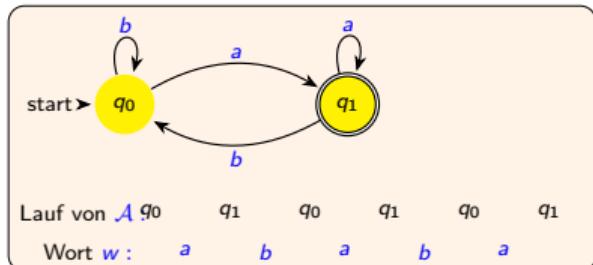
$$\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$$

- „Zustandsübergänge kodieren gültigen Lauf“

$$\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$$

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

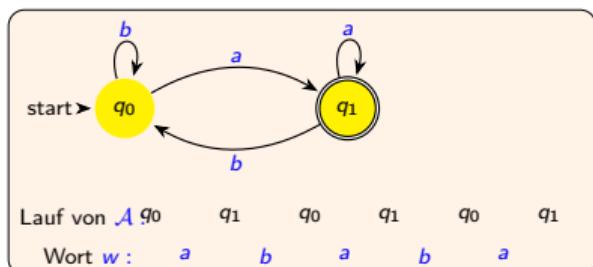
- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“
 $\varphi_{w,Eingabe} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$
- „Nach jedem Buchstaben ist der Automat in genau einem Zustand“
 $\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$
- „Zustandsübergänge kodieren gültigen Lauf“
 $\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$
- „Der letzte Zustand ist in F “
 $\varphi_{w,end} := \bigvee_{q \in F} Q_n^q$

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1



Lauf eines Automaten

Variablen. Sei $w = a_1, \dots, a_n$ ein Wort der Länge n und $\Sigma := \{a, b\}$.

- $\{S_i^c : c \in \Sigma, 1 \leq i \leq n\}$ „ $S_i^c = 1$ gdw. $a_i = c$ “
- $\{Q_i^q : q \in Q, 0 \leq i \leq n\}$ „ $Q_i^q = 1$, wenn \mathcal{A} nach i Schritten in Zustand q ist“

Formel φ_w .

- „Kodieren des Eingabeworts durch Variablen“
 $\varphi_{w,Eingabe} := \bigwedge_{i=1}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$
- „Nach jedem Buchstaben ist der Automat in genau einem Zustand“

$$\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$$

- „Zustandsübergänge kodieren gültigen Lauf“
 $\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$

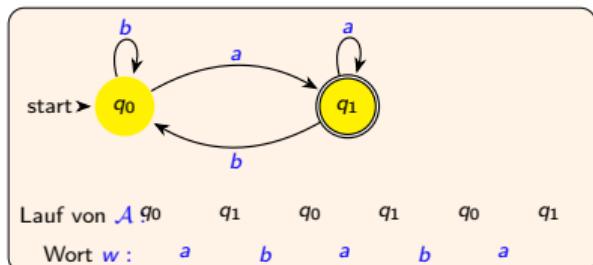
- „Der letzte Zustand ist in F “

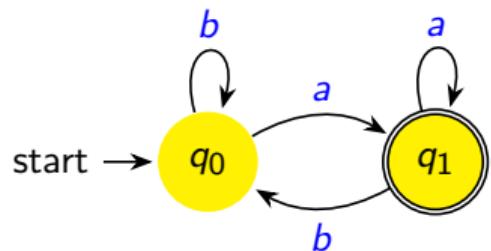
$$\varphi_{w,end} := \bigvee_{q \in F} Q_n^q$$

$$\varphi_w := \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}$$

Variablen.

Pos	0	1	2	3	4	5
S^a		1	0	1	0	1
S^b		0	1	0	1	0
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1





	Position:	1	2	3	4	5
	Wort w :	a	b	a	b	a
	Lauf von \mathcal{A} :	q_0	q_1	q_0	q_1	q_0

Variablen.

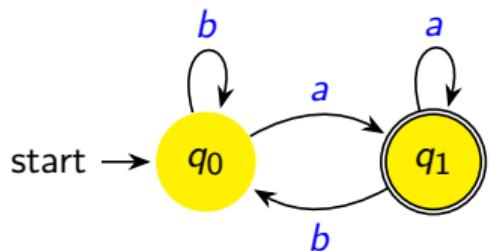
Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

Formel φ_w .

$$\begin{aligned}
 \varphi_{w,Eingabe} &:= \bigwedge_{i=0}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c) \\
 \varphi_{w,Zustand} &:= \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right) \\
 \varphi_{w,Lauf} &:= Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right) \\
 \varphi_{w,end} &:= \bigvee_{q \in F} Q_n^q \\
 \varphi_w &:= \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}
 \end{aligned}$$

Variablen.

$$S_1^a, S_1^b, \dots, S_5^a, S_5^b, \quad Q_0^{q_0}, Q_0^{q_1}, \dots, Q_5^{q_0}, Q_5^{q_1}$$



	Position:	1	2	3	4	5
	Wort w :	a	b	a	b	a
	Lauf von \mathcal{A} :	q_0	q_1	q_0	q_1	q_0

Variablen.

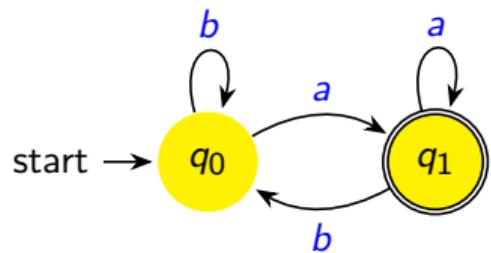
Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

Formel φ_w .

$$\begin{aligned}
 \varphi_{w,Eingabe} &:= \bigwedge_{i=0}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c) \\
 \varphi_{w,Zustand} &:= \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right) \\
 \varphi_{w,Lauf} &:= Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right) \\
 \varphi_{w,end} &:= \bigvee_{q \in F} Q_n^q \\
 \varphi_w &:= \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}
 \end{aligned}$$

„Kodieren des Eingabeworts durch Variablen“

$$\varphi_{ababa,Eing.} := (S_1^a \wedge \neg S_1^b) \quad \wedge \quad (S_2^b \wedge \neg S_2^a) \quad \wedge \quad (S_3^a \wedge \neg S_3^b) \quad \wedge \quad \dots$$



Position: 1 2 3 4 5

Wort w : a b a b a

Lauf von \mathcal{A} : q0 q1 q0 q1 q0 q1

Variablen.

Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

Formel φ_w .

$$\varphi_{w,Eingabe} := \bigwedge_{i=0}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$$

$$\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$$

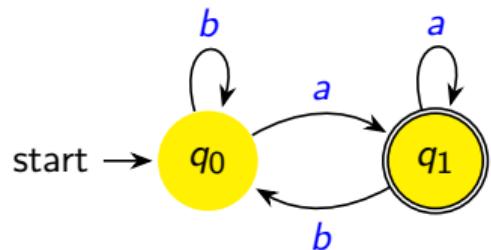
$$\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$$

$$\varphi_{w,end} := \bigvee_{q \in F} Q_n^q$$

$$\varphi_w := \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}$$

„Nach jedem Buchstaben ist der Automat in genau einem Zustand“

$$((Q_0^{q_0} \wedge \neg Q_0^{q_1}) \vee (\neg Q_0^{q_0} \wedge Q_0^{q_1})) \wedge ((Q_1^{q_0} \wedge \neg Q_1^{q_1}) \vee (\neg Q_1^{q_0} \wedge Q_1^{q_1})) \dots$$



	Position:	1	2	3	4	5
	Wort w :	a	b	a	b	a
	Lauf von \mathcal{A} :	q_0	q_1	q_0	q_1	q_0

Variablen.

Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

Formel φ_w .

$$\varphi_{w,Eingabe} := \bigwedge_{i=0}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$$

$$\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$$

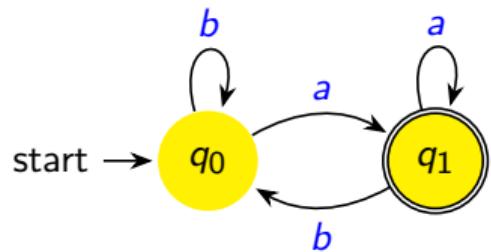
$$\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$$

$$\varphi_{w,end} := \bigvee_{q \in F} Q_n^q$$

$$\varphi_w := \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}$$

„Zustandsübergänge kodieren gültigen Lauf“

$$\begin{aligned}
 Q_0^{q_0} \wedge & \left(\left((Q_0^{q_0} \wedge S_1^a \wedge Q_1^{q_1}) \vee (Q_0^{q_0} \wedge S_1^b \wedge Q_1^{q_0}) \vee (Q_0^{q_1} \wedge S_1^a \wedge Q_1^{q_1}) \vee \right. \right. \\
 & \left. \left. (Q_0^{q_1} \wedge S_1^b \wedge Q_1^{q_0}) \right) \wedge \dots
 \end{aligned}$$



Position:	1	2	3	4	5
Wort w :	a	b	a	b	a
Lauf von \mathcal{A} :	q_0	q_1	q_0	q_1	q_0

Variablen.

Pos	0	1	2	3	4	5
S^a	1	0	1	0	1	
S^b	0	1	0	1	0	
Q^{q_0}	1	0	1	0	1	0
Q^{q_1}	0	1	0	1	0	1

Formel φ_w .

$$\varphi_{w,Eingabe} := \bigwedge_{i=0}^n (S_i^{a_i} \wedge \bigwedge_{c \in \Sigma \setminus \{a_i\}} \neg S_i^c)$$

$$\varphi_{w,Zustand} := \bigwedge_{i=0}^n \left(\bigvee_{q \in Q} (Q_i^q \wedge \bigwedge_{q' \in Q \setminus \{q\}} \neg Q_i^{q'}) \right)$$

$$\varphi_{w,Lauf} := Q_0^{q_0} \wedge \bigwedge_{i=1}^n \left(\bigvee_{(q,a,q') \in \Delta} (Q_{i-1}^q \wedge S_i^a \wedge Q_i^{q'}) \right)$$

$$\varphi_{w,end} := \bigvee_{q \in F} Q_n^q$$

$$\varphi_w := \varphi_{w,Eingabe} \wedge \varphi_{w,Zustand} \wedge \varphi_{w,Lauf} \wedge \varphi_{w,end}$$

„Der letzte Zustand ist in F “

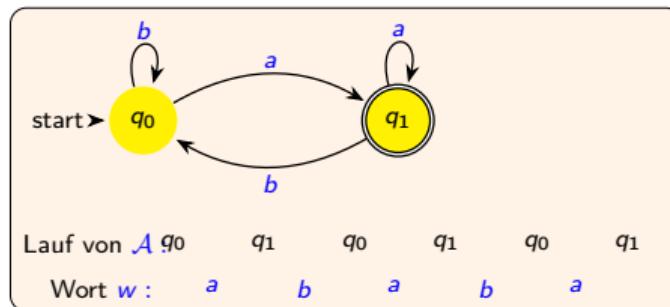
$$\varphi_{ababa,end} := Q_5^{q_1}$$

Formalisierung in der Aussagenlogik

Reduktion auf SAT. Zu jedem $w \in \Sigma^*$ müssen wir eine Formel $\varphi_w \in \text{AL}$ konstruieren, so dass \mathcal{A} akzeptiert w gdw. φ_w ist erfüllbar.

Idee. Die Formel φ_w soll so aus w und \mathcal{A} konstruiert werden, dass

- eine Belegung β von $\text{var}(\varphi_w)$ einem möglichen Lauf $\rho(\beta)$ von \mathcal{A} auf w entspricht und
- wenn $\beta \models \varphi_w$, dann ist $\rho(\beta)$ ein akzeptierender Lauf von \mathcal{A} auf w .



Reduktion des regulären Wortproblems auf SAT

Theorem. Für jedes endliche Alphabet Σ und jede reguläre Sprache $\mathcal{L} \subseteq \Sigma^*$ kann das Wortproblem für \mathcal{L} in Polynomialzeit auf SAT reduziert werden.

D.h., es existiert ein Algorithmus $\mathcal{A}_{\mathcal{L}}$ der auf Eingabe $w \in \Sigma^*$ der Länge n in Zeit $O(n^c)$, für eine Konstante c , eine Formel φ_w berechnet, so dass

$$w \in \mathcal{L} \text{ gdw. } \varphi_w \text{ erfüllbar ist.}$$

Reduktion des regulären Wortproblems auf SAT

Theorem. Für jedes endliche Alphabet Σ und jede reguläre Sprache $\mathcal{L} \subseteq \Sigma^*$ kann das Wortproblem für \mathcal{L} in Polynomialzeit auf SAT reduziert werden.

D.h., es existiert ein Algorithmus $\mathcal{A}_{\mathcal{L}}$ der auf Eingabe $w \in \Sigma^*$ der Länge n in Zeit $O(n^c)$, für eine Konstante c , eine Formel φ_w berechnet, so dass

$$w \in \mathcal{L} \text{ gdw. } \varphi_w \text{ erfüllbar ist.}$$

Folgerung. Wir können also das Wortproblem für \mathcal{L} mit Hilfe eines SAT-Lösers lösen, indem wir zunächst die Formel φ_w ausrechnen und diese dann an den SAT-Löser als Eingabe geben.

Der Satz von Cook

Beweis des Satzes von Cook

Satz. SAT ist NP-vollständig. (Cook 1970, Levin 1973)

NP-Vollständigkeit. Um zu beweisen, dass SAT NP-vollständig ist, müssen wir folgendes zeigen:

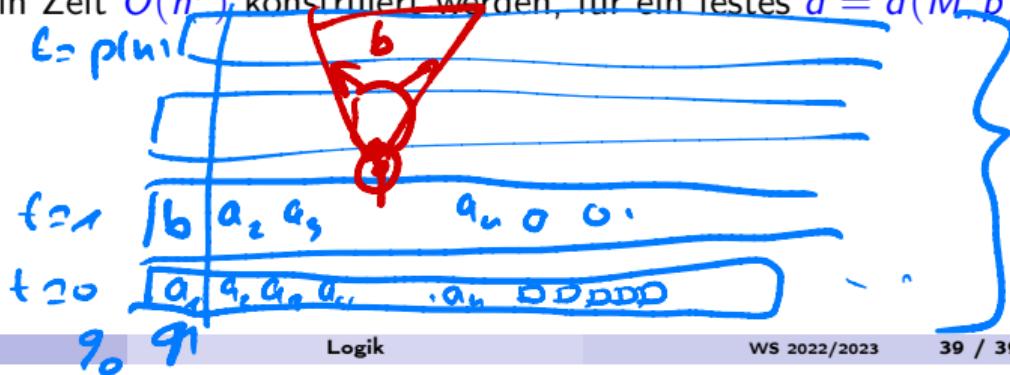
1. $\text{SAT} \in \text{NP}$
2. falls SAT in Polynomialzeit gelöst werden kann, dann können alle NP-Probleme in Polynomialzeit gelöst werden.

Beweis des Satzes von Cook

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

1. Sei $L \in \text{NP}$. Dann gibt es ein Polynom $p(n)$ und eine *nicht-deterministische* TM M , die L in Zeit $O(p(n))$ löst.
2. Für jede feste NTM M und jedes Polynom $p(n)$, zeigt man nun:
Zu jedem Eingabewort w der Länge n existiert eine Formel φ_w mit:
 M akzeptiert w in Zeit $p(n)$ gdw. φ_w erfüllbar ist.

φ_w kann in Zeit $O(n^d)$ konstruiert werden, für ein festes $d = d(M, p)$.



Beweis des Satzes von Cook

NP-Härte von SAT. Wir müssen zeigen, dass wenn SAT in Polynomialzeit lösbar wäre, dann wären alle Probleme $L \in \text{NP}$ in PTIME.

1. Sei $L \in \text{NP}$. Dann gibt es ein Polynom $p(n)$ und eine *nicht-deterministische* TM M , die L in Zeit $O(p(n))$ löst.
2. Für jede feste NTM M und jedes Polynom $p(n)$, zeigt man nun:
Zu jedem Eingabewort w der Länge n existiert eine Formel φ_w mit:
 M akzeptiert w in Zeit $p(n)$ gdw. φ_w erfüllbar ist.
 φ_w kann in Zeit $O(n^d)$ konstruiert werden, für ein festes $d = d(M, p)$.
3. Wenn man also SAT in pol. Zeit lösen könnte, dann könnte man auch in pol. Zeit entscheiden, ob M eine Eingabe w der Länge n in Zeit $p(n)$ akzeptiert. Also könnte man L in pol. Zeit entscheiden.