Theoretische Informatik

Frühjahrssemester 2020

Matthias Krause

2020/03/17, 16:35

Universität Mannheim

Berechenbarkeit

Berechnungsprobleme

Berechnungsprobleme sind Relationen $\Pi \subseteq \Sigma^* \times \Sigma^*$, wobei

- Σ endliches Arbeitsalphabet,
- $(x, y) \in \Pi$ bedeutet, dass y ist Lösung zur Eingabe x bzgl. Π ist.
- $X_\Pi = \{x, \exists y, (x, y) \in \Pi\}$ Menge zulässiger Eingaben,
- $Y_\Pi = \{y, \exists x, (x, y) \in \Pi\}$ Menge möglicher Lösungen

Arten von Berechnungsproblemen Π

- Funktionen $\Pi : \Sigma^* \longrightarrow \Sigma^*$, Beispiele: Addition, Multiplikation.
- Entscheidungsprobleme $=$ Funktionen mit Ausgabe $\{0, 1\}$, Beispiel Primzahltest.
- Optimierungsprobleme: Für jede Eingabe x ist eine Lösung y aus einem Raum $L_\Pi(x)$ möglicher Lösungen zu x gesucht, die bzgl. einer vorgegebenen Zielfunktion auf $L_\Pi(x)$ den optimalen Wert hat, Bsp: Kürzeste Rundfahrten in Entfernungsgraphen.

Eingabeformate, Eingabelänge

Eingaben $x \in X_\Pi$ haben i.d.R. typische Eingabeformate, denen eine Parameter $|x|$, die Eingabelänge, zugeordnet ist.

- Worte, Strings $x \in \Sigma^*$, $|x|$ gleich Anzahl Zeichen von x.
- $m \times n$-Matrizen A, $|A| = nm$.
- Graphen $G = (V, E)$, Eingabelänge $|V|$ oder $|E|$.
- Binärzahlen n, $|n| = \lfloor \log_2(n) \rfloor + 1$ (Anzahl der Binärstellen bei führender 1).

5 Notation von Berechnungsproblemen $\Pi = (\Pi_n)_{n \in \mathbb{N}} \subseteq X \times Y$, $X, Y \subseteq \Sigma^*$

$$\Pi_n = \{(x, y) \in \Pi, |x| = n\}.$$

d.h. $\Pi_n \subseteq X_n \times Y$, $X_n = \{x \in X, |x| = n\}$.

Berechnungsapparate

Berechnungsapparate sind Maschinen, die taktweise auf der Grundlage eines inneren Programms arbeiten, sich auf ein Arbeitsalphabet Σ beziehen, und die pro Takt

- Eingabezeichen aus Σ einlesen können,
- Ausgabezeichen aus Σ ausgeben können,
- Operationen aus einer vorgegebenen endlichen Menge von Grundoperationen ausführen, und
- stoppen können.

Berechnungen

Ein Berechnungsapparat A führt auf jeder Eingabe $x \in \Sigma^*$ eine eindeutig bestimmte Folge von Rechentakten, **die Berechnung von A auf x**, aus, die entweder

- nach endlich vielen Takten stoppt: Dann bezeichnet
  - $A(x) \in \Sigma^*$ die während der Berechnung erzeugte Ausgabe, und
  - $time_A(x) \in \mathbb{N}$ die Anzahl der Takte der Berechnung auf x,

  oder
- unendlich ist, d.h. nicht stoppt bzw. auf x nicht anhält ($time_A(x) = \infty$).

Definition 1

Ein Berechnungsapparat A berechnet ein Berechnungsproblem $\Pi \subseteq \Sigma^* \times \Sigma^*$, falls A auf allen Eingaben $x \in \mathbf{X}_\Pi \subseteq \Sigma^*$ stoppt, und für alle $x \in \mathbf{X}_\Pi$ gilt, dass die Ausgabe $A(x)$ Lösung zu x bezüglich Π ist, d.h. $(x, A(x)) \in \Pi$.

Berechnungsapparate und Berechnungsmodelle

Definition 2

Unter einem Berechnungsmodell $\mathcal{M}$ versteht man eine Menge gleichartiger Berechnungsapparate.

Beispiele: Modell der Schaltkreise, der endlichen Automaten, der Turing-Maschinen (TMs), der Random Access Maschinen (RAMs), der Binären Entscheidungsgraphen (BDDs) usw.

Definition 3

Berechnungsapparate heißen **nicht uniform**, wenn sie nur Eingaben einer festen Eingabelänge verarbeiten (Schaltkreise, BDDs). Berechnungsapparate heißen **uniform**, wenn sie Eingaben beliebiger Eingabelänge verarbeiten (TMs, RAMs, endliche Automaten).

Berechnungsmodelle heißen **nicht uniform** (bzw. **uniform**) wenn sie aus nicht uniformen bzw. uniformen Berechnungsapparaten bestehen.

Kostenmaße für Berechnungsapparate (Beispiele)

Uniformen Berechnungsapparaten A werden oft die Kostenfunktion Zeitbedarf und Speicherplatz zugeordnet

- **worst-case Berechnungszeit**: $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \max\{time_A(x), |x| = n\}.$$

- **average-case Berechnungszeit**: $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \mathbf{E}_{x \in_{P_n} X_n}[time_A(x)],$$

  P_n Wahrscheinlichkeitsverteilung auf $X_n = \{x \in X, |x| = n\}$.

Nichtuniformen Berechnungsapparaten wird oft ein Kostenwert zugeordnet (Z.B. Schaltkreisgröße oder -tiefe, BDD-Größe oder -Breite)

Komplexität von Problemen in uniformen Modellen

Sei $\mathcal{M}$ ein uniformes Berechnungsmodell und

$$\Pi = (\Pi_n)_{n \in \mathbb{N}} \subseteq X \times Y$$

ein Berechnungsproblem, wobei $X, Y \subseteq \Sigma^*$, $\Pi_n \subseteq X_n \times Y$, $X_n = \{x \in X, |x| = n\}$.

Definition 4

Die **Komplexität von** Π bezüglich $\mathcal{M}$ entspricht den Kosten eines kostenminimalen Berechnungsapparats $A \in \mathcal{M}$, der Π berechnet.

Beispiel: Ein Berechnungsproblem Π hat die Zeitkomplexität $t(n)$ (kurz $\Pi \in \mathit{TIME}_{\mathcal{M}}(t(n))$, falls

- Es existiert ein Apparat $A^* \in \mathcal{M}$ für Π mit $\mathit{time}_{A^*} \in \Theta(t(n))$,
- Für alle $A \in \mathcal{M}$ für Π gilt $\mathit{time}_A \in \Omega(t(n))$.

Asymptotisches Wachstum von Funktionen I

Alle kommenden Aussagen beziehen sich auf monoton wachsende Funktionen $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}^+$.

Definition 5

- Es sei $f(n) = O(g(n))$, falls (informal) f **wächst asymptotisch nicht schneller als** g, d.h. (formal) es existiert eine Konstante $C \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ so dass $f(n) \leq C \cdot g(n)$ für alle $n \geq n_0$.
- Es sei $f(n) = \Omega(g(n))$, falls (informal) f **wächst asymptotisch nicht langsamer als** g, d.h. (formal) es existiert eine Konstante $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ so dass $f(n) \geq c \cdot g(n)$ für alle $n \geq n_0$.
- $f(n) = \Theta(g(n))$ falls $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$, d.h. (informal) f **wächst asymptotisch genauso wie** g

Asymptotisches Wachstum von Funktionen II

Definition 6

- Es sei $f(n) = o(g(n))$, falls (informal) f **wächst asymptotisch echt langsamer als** g, d.h. (formal) für alle Konstanten $c \in \mathbb{R}^+$ existiert $n_0 \in \mathbb{N}$ so dass $f(n) < c \cdot g(n)$ für alle $n \geq n_0$.
- Es sei $f(n) = \omega(g(n))$, falls (informal) f **wächst asymptotisch echt schneller als** g, d.h. (formal) für alle Konstanten $C \in \mathbb{R}^+$ existiert $n_0 \in \mathbb{N}$ so dass $f(n) > C \cdot g(n)$ für alle $n \geq n_0$.

Bemerkung: Obiges wird auch als O-Notation für Funktionen bezeichnet, es beinhaltet die Vernachlässigung von Konstanten und additiven Termen niedrigerer Wachstumsordnung (Bsp.: $5n^2 + 3n + 7 = O(n^2)$). Geeignet zur Notationen von Laufzeiten von Algorithmen, da die Kosten pro Rechenschritt oft implementierungs- und hardwareabhängige Konstanten sind.

Wachstumsordnungen

- $\Theta(n)$ Lineares Wachstum
- $\Theta(n^2)$ Quadratisches Wachstum
- $\Theta(n^3)$ Kubisches Wachstum
- $O(1)$ Durch Konstante beschränktes Wachstum
- $\Theta(\log(n))$ Logarithmisches Wachstum (gerechtfertigt durch Logarithmengesetz $\log_a(x) = \log_a(b) \cdot \log_b(x)$, d.h. $\log_a(n) = \Theta(\log_b(n))$ für alle $a, b \geq 1$.)
- f polynomiell beschränkt, falls existiert $k > 0$ mit $f(n) = O(n^k)$.

Abgeleitete Notationen

- Exponentielles Wachstum: $f(n) = \exp(\Omega(n))$, genau dann, wenn existiert eine Konstante $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ mit $f(n) \geq 2^{C \cdot n}$ für $n \geq n_0$. (gerechtfertigt durch Potenzgesetz $a^n = 2^{\log_2(a) \cdot n}$, d.h. $a^n = exp(\Theta(n))$ für alle $a > 1$.)
- Schwach exponentielles Wachstum $\exp(\Omega(n^\delta))$, $0 < \delta < 1$.
- Polynomielle Beschränktheit $f(n) = n^{O(1)}$ genau dann, wenn existiert eine Konstante $C \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ mit $f(n) \leq n^C$ für $n \geq n_0$.
- Polylogarithmische Beschränktheit $f(n) = \log^{O(1)}(n)$
- ...

Asymptotisches Wachstum von Polynomen

Lemma 7

Für f, g existiere der Grenzwert $\gamma = \lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R} \cup \{\infty\}$. Dann gilt

- *Falls $0 < \gamma < \infty$ so $f(n) = \Theta(g(n))$,*
- *falls $\gamma = 0$ so $f(n) = o(g(n))$,*
- *falls $\gamma = \infty$ so $f(n) = \omega(g(n))$.*

Anwendung auf Polynome $p(n) = p_k n^k + p_{k-1} n^{k-1} + \cdots + p_1 n + p_0$ und $q(n) = q_r n^r + q_{r-1} n^{r-1} + \cdots + q_1 n + q_0$.

- Falls $k = r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = \frac{p_k}{q_r}$, also $p(n) = \Theta(q(n))$.
- Falls $k < r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = 0$, also $p(n) = o(q(n))$.
- Falls $k > r$ so $\lim_{n\to\infty} \frac{p(n)}{q(n)} = \infty$, also $p(n) = \omega(q(n))$.

Asymptotischen Wachstum, exponentiell versus polynomiell

Theorem 8

Exponentielle Funktionen wachsen echt schneller als polynomielle, d.h. für alle Konstanten $k \in \mathbb{N}^+$ gilt $n^k = o(e^n)$.

Beweis: Erinnerung $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ und

$$p(n) = \sum_{i=0}^{k+1} \frac{n^i}{i!} \; < \; e^n.$$

Also $p(n) = \Theta(n^{k+1})$, d.h. $n^k = o(p(n))$, also $n^k = o(e^n)$.

Anforderungen an formale uniforme Berechnungsmodelle

Ein formales uniformes Berechnungsmodell sollte einfach und formal gut handhabbar sein und die wesentlichen Aspekte von Berechnungen auf realen Rechnern widerspiegeln.

Insbesondere sollten genau die Probleme berechenbar sein, die auch auf realen Rechnern berechenbar sind, und genau dies Probleme effizient berechenbar sein, die auch auf realen Rechnern effizient berechenbar sind.

Das Modell der Turing Maschinen (TMs) ist das am häufigsten verwendete formale uniforme Berechnungsmodell.

TMs beruhen auf der zeichenweisen Verabeitung von Information, wobei die Zeichen aus einem festgelegten endlichen Arbeitsalphabet stammen.

Turing Maschinen (TMs)

benannt nach dem Erfinder, dem englischen Pionier der Theoretischen Informatik und Kryptanalyse, Alan Turing (1912-1954)

- Definiert über endlichem Arbeitsalphabet Σ, das idR ein Trennzeichen $\#$ enthält (z.B. $\Sigma = \{0, 1, \#\}$)
- TMs bestehen aus CPU, Programmspeicher und Hauptspeicher, der aus einem oderen mehreren linearen TM Bändern besteht
- TM Bänder unterteilt in Felder, in denen Zeichen aus Σ stehen
- Pro Band ein Lese/Schreibkopf, der von CPU gemäß des TM Programms gesteuert wird
- Taktweise arbeitend, pro Takt kann jeder Kopf in Abhängigkeit von den aktuell eingelesenen Zeichen ein neues Zeichen schreiben, und stehenbleiben oder ein Feld nach links oder rechts gehen

TMs, Formale Definition

Definition 9

Eine k-Band TM über Σ ist ein Tripel $M = (Q, q_0, \delta)$, wobei

- Q endliche Zustandsmenge
- $q_0 \in Q$ Anfangszustand
- $\delta : Q \times \Sigma^k \xrightarrow{\supset} \Sigma^k \times MOVE^k \times Q$
  Zustandsüberführungsfunktion, wobei $MOVE = \{L, R, N\}$ (geh nach links (L) oder rechts (R) oder bleibe wo Du bist (N)).

Interpretation von δ-Instanzen

$$(q, (x_1, \cdots, x_k); (y_1, \cdots, y_k), (m_1, \cdots, m_k), q')$$

Falls M im Zustand q, und die k Köpfe lesen $(x_1, \cdots, x_k)$, so schreiben diese $(y_1, \cdots, y_k)$, bewegen sich gemäß $(m_1, \cdots, m_k)$, und M geht in Zustand q' über.

Beispiel 1: Inkrementieren von Binärzahlen

Eine 1-Band TM $M = (\{q_0, q_1, q_2\}, q_0, \delta)$ über $\Sigma = \{0, 1, \#\}$, die eine Binärzahl $x = (x_{n-1}, \cdots, x_0)$ um 1 erhöht.

$$\delta = \{(q_0, 1; 0, L, q_0), (q_0, 0; 1, R, q_1), (q_0, \#; 1, R, q_1),$$
$$(q_1, 0; 0, R, q_1), (q_1, 1; 1, R, q_1), (q_1, \#; \#, L, q_2)\}$$

In der Start- und Endkonfiguration steht der Kopf auf dem least significant bit x_0.

Beispiel 2: Addition von Binärzahlen

Eine 3-Band TM $M = (\{q_0, q_1, q_2\}, q_0, \delta)$, die zwei Binärzahlen addiert (liest nur von Band 1 und 2, schreibt nur auf Band 3).

$$\delta = \{(q_0, 0\#; 0, L, q_0), (q_0, \#0; 0, L, q_0), (q_0, 00; 0, L, q_0),$$

$$(q_0, \#\#; \#, N, q_2),$$

$$(q_0, 1\#; 1, L, q_0), (q_0, \#1; 1, L, q_0), (q_0, 01; 1, L, q_0), (q_0, 10; 1, L, q_0),$$

$$(q_0, 11; 0, L, q_1),$$

$$(q_1, 0\#; 1, L, q_0), (q_0, \#0; 1, L, q_0), (q_1, 00; 1, L, q_0), (q_1, \#\#; 1, L, q_2),$$

$$(q_1, 1\#; 0, L, q_1), (q_1, \#1; 0, L, q_1), (q_1, 01; 0, L, q_1), (q_1, 10; 0, L, q_1),$$

$$(q_1, 11; 1, L, q_1)\}$$

Beispiele: Vergleich von Binärzahlen

Eine read-only 2-Band TM $M = (\{q_0, q_{acc}, q_{rej}\}, q_0, \delta)$, die zwei Binärzahlen $x = (x_{n-1}, \cdots, x_0)$ (auf Band 1) und $y = (y_{n-1}, \cdots, y_0)$ (auf Band 2) vergleicht, und genau dann akzeptiert wenn $x \leq y$.

$$\delta = \{(q_0, 00; R, q_0), (q_0, 11; R, q_0), (q_0, \#\#; N, q_{acc}),$$

$$(q_0, 01; R, q_{acc}), (q_0, 10; R, q_{rej})\}.$$

Achtung: Ausgabeverhalten geregelt über akzeptierende bzw. verwerfende Stoppzustände.

Turing Maschinen versus Endliche Automaten

Theorem 10

Jeder DFA $A = (Q, q_0, F, \delta)$ *über* Σ *kann direkt durch eine 1-Band TM* $M = (Q, q_0, \tilde{\delta})$ *über* $\Sigma \cup \{\#\}$ *simuliert werden,*

$$\tilde{\delta} = \{(q, a; R, \delta(q, a)), q \in Q, a \in \Sigma\}.$$

Beweisidee: M startet die Berechnung auf $x = (x_1, \cdots, x_n)$ auf Bandinschrift $\#x_1 \cdots x_n\#$ mit Kopf auf x_1, geht in jedem Schritt nach rechts, wechselt die Zustände gemäß A und stoppt bei Erreichen des rechten Begrenzers $\#$. □

Theorem 11

Jede read-only TM mit Ausgabe $0, 1$ *kann durch einen DFA simuliert werden.* □

Beweisidee: siehe [We99], Kap. 4.5

Konfigurationen von TMs

Definition 12

Sei $M = (Q, q_0, \delta)$ eine k-TM über Σ. **K**onfigurationen von M beschreiben den Gesamtzustand von M zu einem gegebenen Zeitpunkt, also den aktuellen Zustand, die aktuellen Bandinschriften und die aktuellen Kopfpositionen, d.h.

$$Conf(Q, k, \Sigma) = Q \times (\Sigma^*)^k \times \mathbb{Z}^k.$$

- Eine Instanz $I \in \delta$ heißt anwendbar auf $K \in Conf(Q, k, \Sigma)$, falls die linke Seite von I konsistent mit dem aktuellen Zustand und den aktuellen Zeichen an den aktuellen Kopfpositionen in K ist.
- Ist $I \in \delta$ anwendbar auf $K \in Conf(Q, k, \Sigma)$, so bezeichnet $I(K) \in Conf(Q, k, \Sigma)$ die entsprechende Nachfolgekonfiguration von K (Bezeichnung $K \xrightarrow{M} I(K)$)

Konfigurationen und Berechnungen

- K heißt **Stoppkonfiguration**, falls kein $I \in \delta$ auf K anwendbar ist.
- Die Folge $K \xrightarrow[M]{} K_1 \xrightarrow[M]{} K_2 \xrightarrow[M]{} \cdots$ heißt Berechnung von M auf $K \in Conf(k, \Sigma)$.
- Schreibweise $K \xrightarrow[M]{*} K'$, falls die Berechnung von M auf K die Konfiguration K' enthält.
- Die Berechnung heißt abbrechend (oder haltend) wenn sie eine Stoppkonfiguration erreicht, nicht abbrechend (oder haltend) sonst.
- $time_M(K)$ bezeichnet die Anzahl der Takte der Berechnung von M auf K bis zum Erreichen einer Stoppkonfiguration, $time_M(K) = \infty$ falls M hält nicht auf K.

Berechnungsmodus von TMs

Es sei Σ ein endliches Alphabet, $\Pi \subseteq X_\Pi \times Y_\Pi$ ein Berechnungsproblem über Σ, wobei $X_\Pi, Y_\Pi \subseteq \Sigma^*$.

Definition 13

Eine k-TM $M = (Q, q_0, \delta)$ über dem Alphabet Σ', $\Sigma \subseteq \Sigma'$, berechnet Π falls

- Für alle Eingaben $x \in X_\Pi$ ist eine Startkonfiguration $K_0(x) \in \mathit{Conf}(k, \Sigma)$ definiert.
- Für alle $y \in Y_\Pi$ sind Stoppkonfigurationen $K(y) \in \mathit{Conf}(k, \Sigma)$ definiert, die y eindeutig ausweisen.
- Für alle $x \in X_\Pi$ hält M auf $K_0(x)$ in der Stoppkonfiguration $K(y)$ (d.h. $K_0(x) \xrightarrow[M]{*} K(y)$), wobei $y \in Y_\Pi$ eine Lösung für x bezüglich Π ist (d.h. $(x, y) \in \Pi$).

Zeitverhalten von TMs

Sei M eine k-TM über Σ , die bezüglich einer Menge zulässiger Eingaben $X \subseteq \Sigma^*$ definiert ist. Für alle $x \in X$ sei eine Startkonfiguration $K_0(x) \in Conf(k, \Sigma)$ definiert.

Definition 14

- Die **worst case Laufzeit** von M ist definiert als

$$time_M(n) = \max\{time_M(K_0(x)), |x| = n\}.$$

- Die **average case Laufzeit** von M ist definiert als

$$time_M(n) = \mathbf{E}[time_M(K_0(x)],$$

  bezogen auf eine Wahrscheinlichkeitsverteilung auf $X_n = \{x \in X, |x| = n\}$.

Simulation k-TMs durch 1-TMs

Theorem 15

Jede $t(n)$-zeitbeschränkte k-TM M über Σ kann durch $O(t^2(n))$-zeitbeschränkte 1-TM M' über Σ^{2k} simuliert werden.

Beweisidee: Konfigurationen

- In den Spuren $1, 3, 5, \cdots, 2k-1$ des M'-Bands stehen die aktuellen Bandinschriften von M.
- In den Spuren $2, 4, 6, \cdots, 2k$ sind die aktuellen Kopfpositionen vermerkt (zB durch Sonderzeichen unter entsprechendem Feld).

M-Instanzen I werden simuliert, indem der M'-Kopf von links nach rechts und zurück über die gesamte Inschrift geht und diese gemäß I aktualisiert, Zeitbedarf $O(t(n))$ pro M-Takt (da die Gesamtlänge der Bandinschrift durch $O(t(n))$ beschränkt ist).

Simulation von TMs über Σ durch TMS über $\{0,1\}$

Theorem 16

Jede $t(n)$-zeitbeschränkte 1-TM M über einem endlichen Arbeitsalphabet Σ kann durch $O(t(n))$-zeitbeschränkte 1-TM M' über $\{0,1\}$ simuliert werden.

Beweisidee: Sei $k := \lfloor \log_2 |\Sigma| \rfloor$. Kodieren alle Σ-Zeichen a durch Blöcke $c(a) \in \{0,1\}^k$. Kodieren so auf dem M'-Band zeichenweise die Inschriften des M-Bandes.

Simulation einer M-Instanz I: Aktualisieren den dem aktuellen Zeichen entsprechenden Block gemäß I und gehen mit dem M'-Kopf ggf. auf das Anfangszeichen des linken oder rechten Nachbarblocks. Zeitbedarf pro M-Takt ist $O(1)$.

Reale Rechner und Turing Maschinen, I

Die Hauptkomponenten eines realen Rechners sind sein Speicher (typischerweise verteilt in Cash, Arbeitsspeicher, Festplatte usw.) und ein (oder mehrere) Prozessoren.

Im Speicher werden (Adresse,Datenblock)-Paare gespeichert, wobei Adressen und Datenblöcke eine vorgegebene Länge l haben (bei modernen Rechnern $l = 64$).

Computerprogramme werden kompiliert in Maschinencode Programme, d.h., endliche durchnumerierte Folgen von Maschinencode Befehlen.

Beispiel $C_3 = ADD(C_1, C_2)$ heißt: Lade Block X von Adresse C_1 und Block Y von Adresse C_2 in den Akkumulator des Prozessors, berechne dort $Z = X + Y \mod 2^l$ und speichere Z unter Adresse C_3 ab.

Reale Rechner und Turing Maschinen, II

Reale Rechner können in verschiedener Weise durch TMs simuliert werden.

Eine einfache (aber wenig effiziente) Möglichkeit ist, eine TM mit Arbeitsalphabet $\Sigma = \{0,1\}^l \times \{0,1\}^l$ zu wählen und die (Adresse, Datenblock)-Paare direkt in den Feldern eines TM-Bandes zu schreiben.

Die Simulation von $C_3 = ADD(C_1, C_2)$ wäre:

- Suche die Blöcke (C_1, X) und (C_2, Y) auf dem Arbeitsband und kopiere X und Y auf Hilfsbänder.
- Berechne auf Hilfsbändern $Z = X + Y$
- Schreibe einen Block (C_3, Z) aufs Arbeitsband bzw. überscheibe einen bestehenden Block (C_3, Z') durch (C_3, Z)

Problem ist, dass beim Laden und Speichern das gesamte Arbeitsband abgelaufen werden muss, d.h. jeder Maschinenbefehl kostet $O(t(n))$, Gesamtzeit $O(t^2(n))$.

Resumee und Church'sche Thesen

Alonzo Church (1903-1995), US Mathematiker, Logiker, Philosoph, Theoretische Informatik Pionier, Lehrer von Alan Turing

- **Church'sche These:** Die Menge der berechenbaren Probleme ist in allen vernünftigen Rechnermodellen gleich und stimmt mit der Menge der im intuitiven Sinne berechenbaren Probleme überein.
- **Erweiterte Church'sche These:** Die Menge der in Polynomialzeit berechenbaren Probleme ist in allen vernünftigen Rechnermodellen gleich und stimmt mit der Menge der im intuitiven Sinne effizient berechenbaren Probleme überein.
- **Insbesondere** sind 1-TMs über $\{0, 1, \#\}$ ein vernünftiges Rechnermodell.
- $REK = \{\Pi, \Pi \text{ ist berechenbar}\}$
- $PTIME = \{\Pi, \Pi \text{ ist in Zeit } n^{O(1)} \text{ berechenbar}\}$

Universalität

Wir haben bislang nur solche Turing Maschinen betrachtet, auf denen genau ein Algorithmus zur Lösung eines vorgegebenen Berechnungsproblems (zum Beispiel des Inkrementierens einer Binärzahl) implementiert ist.

Das widerspricht der wichtigen Eigenschaft der **Programmierbarkeit** realer Rechner.

D.h., reale Rechner sind insoweit universell, als dass Algorithmen für beliebige berechenbare Problem auf ihnen implementierbar sind.

Dieses Prinzip der **Universalität** (d.h., der Programmierbarkeit) lässt sich mittels der Konzepts der **Universellen Turing Maschinen (UTMs)** auch auf Turing Maschinen übertragen.

Universelle Turing Maschinen (UTM)

Arbeitsweise einer UTM M_U:

- UTMs sind **programmierbare** TMs
- **Eingabe:** $([M], x)$, - $[M]$ Programm einer TM M, x zulässige Eingabe für M.
- **Arbeitsweise:** M_U simuliert M auf x taktweise
- **Ausgabe:** $M(x)$ falls M auf x hält

Das TM-Programm $[M]$ wird in spezieller Weise als $0, 1$-String, die **Gödelnummer** von M, kodiert.

(**Kurt Friedrich Gödel**, 1906-1978, bedeutender Mathematiker, Logiker, bekannt durch 1. und 2. Unvollständigkeitssatz und Arbeiten zur Kontinuumshypothese)

Gödelisierung von TMs

Sei $M = (Q, q_1, \delta)$ 1-TM über Σ mit

- $\Sigma = \{X_1, X_2, X_3\}$, wobei $X_1 = 0$, $X_2 = 1$, $X_3 = \#$.
- $Q = \{q_1, \cdots, q_k\}$
- $MOVE = \{M_1, M_2, M_3\}$ mit $M_1 = L$, $M_2 = R$, $M_3 = N$.
- $\delta = \{I_1, \cdots, I_s\}$, wobei
- M-Instanzen die Gestalt $I = (q_i, X_a; X_b, M_r, q_j)$, $1 \leq i,j \leq k$, $1 \leq a,b,r \leq 3$ haben.

Dann ist das TM-Programm $[M]$ in folgender Weise definiert:

- $[M] = 111code(I_1)11code(I_2)11 \cdots 11code(I_s)111$, wobei
- $code(I) = 0^i 10^a 10^b 10^r 10^j$.

Beispiel: $code(q_1, 0; 1, R, q_4) = 01010010010000$.

Arbeitsweise von UTMs

- M_U ist 3-TM, Band-1 simuliert das M-Band, Band-2 (read-only) enthält $[M]$, Band-3 den aktuellen M-Zustand
- **Startkonfiguration** für Eingabe (z, x): z auf Band-2, x auf Band-1, 0 auf Band-3
- **Phase-1**: M_U prüft, ob z eine syntaktisch korrekte Gödelnummer $[M]$ einer 1-TM M ist (Übung).
- **Phase-2**: Taktweise Simulation von M auf x. Hier: Simulation eines Rechenschritts $K \xrightarrow{M} K'$:
  - Lese X_a auf Band-1 und 0^i auf Band-3
  - Suche Instanz $0^i 1 0^a 1 0^b 1 0^r 1 0^j$ auf Band-2.
  - schreibe X_b auf Band-1, bewege 1-Kopf gemäß M_r und aktualisiere Band-3 zu 0^j.
- Beobachtung $time_{M_U}([M], x) = O(time_M(x))$.

REK ist abzählbar

Betrachten Berechnung von Sprachen $L \subseteq \{0,1\}^*$ durch 1-Akzeptoren, d.h. 1-TMs über $\{0,1,\#\}$, die nur akzeptieren oder verwerfen.

- $REK = \{L \subseteq \{0,1\}^*$,L berechenbar$\}$ Menge der berechenbaren (rekursiven) Sprachen.
- **Beobachtung 1:** *REK* ist abzählbar (Zuordnung der kleinsten Gödelnummer eines 1-Akzeptors, der *L* berechnet).
- **Beobachtung 2:** Potenzmenge von $\{0,1\}^*$ ist überabzählbar (gleichmächtig mit Menge der rellen Zahlen).
- **Also:** Fast alle Sprachen sind nicht berechenbar.
- Kann man Sprachen explizit definieren, die nicht berechenbar sind?

Rekursiv Aufzählbare Probleme

Definition 17

Eine Sprache $L \subseteq \Sigma^*$ heißt **rekursiv aufzählbar**, falls es eine TM gibt, die genau die Eingaben $x \in \Sigma^*$ akzeptiert, für die $x \in L$ gilt.

- $ENUM = \{L \subseteq \Sigma^*, \text{L rekursiv aufzählbar}\}$ Menge der rekursiv aufzählbaren Sprachen.
- TMs, die eine Sprache L rekursiv aufzählen, dürfen auf Eingaben $x \notin L$ verwerfen oder nicht stoppen.
- **Beobachtung:** $REK \subseteq ENUM$, wir werden bald sehen, dass Inklusion echt.
- **Beispiel:** die Universalsprache $U = \{([M], x), M(x) = 1\}$ ist rekursiv aufzählbar.
- *ENUM* ist abzählbar (Übungsaufgabe).

Nichtberechenbarkeit der Diagonalsprache D

Definition 18

D besteht aus den Gödelnummern der 1-Akzeptoren M über $\{0, 1, \#\}$, die die Eingabe $[M]$ verwerfen.

Theorem 19

$D \notin REK$.

Beweis: Angenommen $D \in REK$ und M_D berechne D. Betrachten das Berechnungsverhalten von M_D auf $[M_D]$. M_D akzeptiert $[M_D]$ genau dann, wenn $[M_D] \in D$ genau dann, wenn M_D verwirft $[M_D]$, Widerspruch. □

Diese Beweismethode heißt **Diagonalisierung**.

Nichtberechenbarkeit des Halteproblems H

Definition 20

$H = \{([M], x);\ M \text{ hält auf } x\}$.

Theorem 21

$H \notin REK$.

Beweis: Angenommen $H \in REK$ und M_H berechnet H. Konstruieren mittels M_H eine TM M_D für D:

- Sei $[M]$ Eingabe für M_D.
- Falls $M_H([M], [M]) = 0$ sei $M_D([M]) = 0$, ansonsten $M_D([M]) = \neg M([M])$. $\square$

Nichtberechenbarkeit des speziellen Halteproblems H_ϵ

Definition 22

$H_\epsilon = \{[M];\ M \text{ hält auf leerem Band } \epsilon\}$.

Theorem 23

$H_\epsilon \notin REK$.

Beweis: Angenommen $H_\epsilon \in REK$ und M_ϵ berechnet H_ϵ. Konstruieren mittels M_ϵ eine TM M_H für H:

- Sei $([M], x)$ Eingabe für M_H.
- M_H benutzt TM M_x, die auf Eingabe ϵ einfach x aufs Band schreibt
- M_H modifiziert M zu TM $\tilde{M}$, die auf leerem Band zuerst M_x und dann M auf x startet.
- **Beobachtung:** $M_H([M], x) = M_\epsilon([\tilde{M}])$. $\square$

Nichtberechenbarkeit der Universalsprache U

Definition 24

$U = \{([M], x);\ \text{M(x)=1}\}$.

Theorem 25

$U \notin REK$.

Beweis: Angenommen $U \in REK$ und M_U berechnet U. Konstruieren mittels M_U eine TM M_H für H:

- Sei $([M], x)$ Eingabe für M_H.
- M_H benutzt TM $\bar{M}$, die auf genau den Eingaben hält auf denen M hält, und genau die Eingaben akzeptiert, die M verwirft.
- **Beobachtung:** Es gilt $M_H(M, x) = 1$ genau dann wenn $M_U([M], x) = 1$ oder $M_U([\bar{M}], x) = 1$. $\square$

Systematische Nichtberechenbarkeitsbeweise: Reduktionen

Definition 26

- Eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt **rekursive Reduktion** von der Sprache $L \subseteq \{0,1\}^*$ auf die Sprache $L' \subseteq \{0,1\}^*$, falls
  - f ist berechenbar.
  - Für alle $x \in \{0,1\}^*$ gilt $x \in L$ genau dann, wenn $f(x) \in L'$.
- Die Sprache L heißt **rekursiv reduzierbar** auf L' (Schreibweise $L \leq_{rek} L'$), falls eine rekursive Reduktion von L auf L' existiert.

Theorem 27

Aus $L \leq_{rek} L'$ und $L' \in REK$ folgt $L \in REK$.

Beweis: Konstruktion einer TM M für L mittels TMs M_f für eine Reduktion f von L auf L' und einer TM M' für L': Gegeben Eingabe x, berechne $M_f(x) = f(x)$ und akzeptiere x genau dann wenn $M'(f(x)) = 1$. □

Beispiele für rekursive Reduktionen

1.) $D \leq_{rek} H$, Konstruktion einer rekursiven Reduktion f von D auf H:

- $f([M]) := ([\tilde{M}], [M])$, wobei $\tilde{M}$ wiefolgt definiert:
- $\tilde{M}$ benutzt TM M^*, die niemals hält.
- $\tilde{M}$ startet M auf x und startet M^* falls $M(x) = 1$.
- $\tilde{M}$ hält genau dann auf x wenn M auf x hält und $M(x) = 0$.

2.) $H \leq_{rek} H_\epsilon$, Konstruktion einer rekursiven Reduktion f von H auf H_ϵ:

- $f([M], x) := ([\tilde{M}])$, wobei $\tilde{M}$ wiefolgt definiert:
- $\tilde{M}$ startet M_x auf dem leeren Band, und danach M auf x.
- $\tilde{M}$ hält genau dann auf dem leeren Band wenn M auf x hält.

Partielle Berechenbarkeit

Definition 28

- Eine TM M berechnet partiell eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1,*\}$, falls $M(x) = f(x)$ für alle $x \in \{0,1\}^*$ für die $f(x) \in \{0,1\}$. (Bedeutung von $*$ ist don't care)
- $\mathcal{R} = \{f : \{0,1\}^* \longrightarrow \{0,1,*\},\ f \text{ partiell berechenbar}\}$

- $REK \subseteq \mathcal{R}$
- berechenbar gleich partiell berechenbar für Funktionen $f : \{0,1\}^* \longrightarrow \{0,1\}$.
- Spezielle Funktion $u \in \mathcal{R}$, $u(x) = *$ für alle $x \in \{0,1\}^*$. Jede TM M berechnet partiell u.

Der Satz von Rice

Definition 29

Für alle $S \subseteq \mathcal{R}$ sei $L_S = \{[M], M \text{ berechnet partiell ein } f \in S\}$

Theorem 30

Für alle nichtleeren $S \subseteq \mathcal{R} \setminus \{u\}$ gilt $L_S \notin REK$.

Beweis: Wir zeigen $H_\epsilon \leq_{rek} L_S$. Sei $[M]$ Eingabe für H_ϵ:

- Sei M_f eine TM, die eine Funktion $f \in S$ berechnet.
- Konstruieren mittels $[M]$ und $[M_f]$ eine TM $\tilde{M}$, die auf Eingabe x folgendermaßen arbeitet:
- $\tilde{M}$ startet M auf ϵ und danach M_f auf x.
- **Beobachtung:** $F(M) := \tilde{M}$ ist rekursive Reduktion von H_ϵ auf L_S ($\tilde{M}$ berechnet entweder $f \in S$ oder $u \notin S$). $\square$

Das POST'sche Korrespondenzproblem (PKP)

Emil Leon Post (1897-1954), US-Mathematiker jüdisch-polnischer Herkunft, bahnbrechende Arbeiten zur Logik und Berechenbarkeitstheorie

Definition 31

- Zulässige PKP-Eingaben sind Folgen $(X, Y) = \big((x^1, y^1), \cdots, (x^t, y^t)\big)$ von Paaren von Worten über einem vorgegebenem endlichen Alphabet Σ.
- Für endliche Indexfolgen $I = (i^1, \cdots, i^p)$, $1 \leq i^r \leq t$ für alle r, $1 \leq r \leq p$, sei $X(I) = (x^{i^1} \cdots x^{i^p})$, $Y(I)$ entsprechend.
- Es gilt $(X, Y) \in PKP$ falls es eine Indexfolge I gibt, die Lösung von (X, Y) ist, d.h. für die $X(I) = Y(I)$.

PKP Beispiele und Spezialfall

- **Beispiel 1:** $(X, Y) = ((1, 111), (10111, 10), (10, 0))$ hat Lösung $(2, 1, 1, 3)$.
- **Beispiel 2:** $(X, Y) = ((10, 101), (011, 11), (101, 011))$ hat keine Lösung.
- **Beispiel 3:** $(X, Y) = ((001, 0), (01, 011), (01, 101), (10, 001))$ hat Lösungen I, für $|I| \geq 66$.

Definition 32

Das modifizierte PKP, MPKP, ist definiert wie das PKP, allerdings wird nach der Existenz von Lösungen $I = (i^1, \cdots, i^p)$ gefragt, für die $i^1 = 1$.

Die Nichtberechenbarkeit von PKP

Theorem 33

PKP $\notin$ *REK.*

Beweisidee: Zeigen zunächst $MPKP \leq_{rek} PKP$ (ÜA) und dann $H \leq_{rek} MPKP$ durch Angabe einer Reduktion f, $f([M], x) = (X, Y)$, so dass eine MPKP-Lösung I für (X, Y) genau dann existiert, wenn M auf x hält:

- Folge J heiße **Teillösung** für (X, Y), falls $X(J)$ Präfix von $Y(J)$ oder umgekehrt.
- **Beobachtung:** Jedes Anfangsstück einer Lösung I für (X, Y) ist Teillösung für (X, Y).
- Es gilt $(x_1, y_1) = (\#, \#\#K_0(x))$, wobei $K_0(x)$ Kodierung der Startkonfiguration von M auf x.

Die Nichtberechenbarkeit von PKP (Fortsetzung)

M-Instanzen werden so in Wortpaare für (X, Y) übersetzt, dass:

- Für alle Teillösungen J gilt

$$X(J) = \#\#K_0(x)\#\#K_1\#\#\cdots\#\#K_s$$

$$Y(J) = \#\#K_0(x)\#\#K_1\#\#\cdots\#\#K_s\#\#K_{s+1}$$

  wobei für alle r, $0 \leq r \leq s$, K_{r+1} die M-Nachfolgekonfiguration von K_r (oder im Fall $r = s$ ein Anfangsstück derselben) kodiert.

- Für alle Lösungen I gilt

$$X(I) = Y(I) = \#\#K_0(x)\#\#K_1\#\#\cdots\#\#K_t\#\#\#$$

  wobei zusätzlich K_t eine Stoppkonfiguration kodiert.

- Details siehe Buch von Ingo Wegener, Kap. 2.8.

NP-Vollständigkeit als Methode zum Nachweis der Schwierigkeit von Problemen

Leichte, unlösbare und schwierige Probleme

- **Effizient lösbare Probleme:** Sortieren, Berechnung minimaler Spannbäume, kürzester Wege usw., Primzahltesten, Lineare Optimierung $\cdots$
- **Unlösbare (d.h. nicht berechenbare) Probleme:** Verifikation nichttrivialer Eigenschaften von Computerprogrammen, POSTsches Korrespondenzproblem, $\cdots$
- **Schwierige Probleme:** d.h. Probleme, die nur in exponentieller Zeit gelöst werden können, SAT, 3-SAT, Berechnung kürzester Rundfahrten, maximaler Cliquen, Lineare ganzzahlige Optimierung, Überdeckungsprobleme, Berechnung diskreter Logarithmen und der Faktorisierung ganzer Zahlen, $\cdots$

Nachweis der Schwierigkeit von Problemen

Wie weist man nach, dass ein gegebenes Problem schwierig ist?

1.) **Empirisch** ... trotz langjährigen intensive Bemühungen der weltweiten wissenschaftlichen Community findet niemand einen subexponentiellen Algorithmus.

2.) **Absolut** ... jemand findet einen Beweis, dass alle Algorithmen, die dieses Problem berechnen, exponentielle Laufzeit haben müssen.

3.) **Relativ** ... man beweist, dass wenn dieses Problem einen subexponentiellen Algorithmus hat, so trifft das auch auf eine riesige Klasse von Problemen zu, für die trotz langjähriger intensiver Bemühungen der weltweiten wissenschaftlichen Community niemand einen subexponentiellen Algorithmus gefunden hat.

Die NP-Vollständigkeitstheorie liefert Schwierigkeitsnachweise vom Typ 3.

Schwierig erscheinendende Graphprobleme I: Das Cliquenproblem

- **Eingabe:** Ungerichteter Graph $G = (V, E)$
- **Ausgabe:** Eine maximale Clique $V' \subseteq V$ von G

Definition 34

Eine Teilmenge $V' \subseteq V$ heißt **Clique** (oder vollständiger Untergraph) von G, falls für alle $v \neq w \in V'$ gilt $(v, w) \in E$.

Entscheidungsvariante: Eingabe ungerichteter Graph $G = (V, E)$ und $k \in \mathbb{N}$, entscheide ob G eine Clique mit mind. k Knoten hat

Schwierig erscheinendende Graphprobleme II: Das Hamiltonkreis Problem (HC)

- **Eingabe:** Ungerichteter Graph $G = (V, E)$
- **Frage:** Hat G einen Hamiltonkreis?

Definition 35

Ein Hamiltonkreis in G ist ein einfacher Kreis mit $|V|$ Kanten, der alle Knoten von G enthält.

- **Bemerkung 1:** Das verwandte **Eulerkreis Problem**, ob $G = (V, E)$ einen Eulerkreis, d.h. einen Kreis der jede Kante von G genau einmal enthält, besitzt, ist effizient lösbar.
- **Bemerkung 2:** benannt nach **Sir William Rowan Hamilton** (1805-1865), irischer Mathematiker und Physiker, Erfinden der Hamiltonschen Mechanik, formulierte das Problem für den Dodekaedergraphen

Schwierig erscheinendende Graphprobleme III: Kürzeste Rundfahrten

- **Eingabe:** Gewichteter Graph über n Knoten, gegeben als Entfernungsmatrix $D = (d_{i,j})_{i,j=1}^{n}$.
- **Ausgabe:** Eine kürzeste Rundfahrt bzgl. D, d.h. eine Permutation $\pi \in \mathcal{S}_n$, so dass

$$\pi(D) = d_{\pi(1),\pi(2)} + \cdots + d_{\pi(n-1),\pi(n)} + d_{\pi(n),\pi(1)}$$

  minimal.

Dieses Problem wird auch als **Travelling Salesman Problem (TSP)** bezeichnet.

Schwierige Probleme: Überdeckungsprobleme

- **Eingabe:** Mengensystem $\mathcal{U} = \{U_1 \cdots, U_m\}$ aus Teilmengen eines Universums U, $U = \bigcup_{i=1}^{m} U_i$.
- **Ausgabe:** Eine minimale Überdeckung, d.h. eine minimale Teilmenge $I \subseteq \{1, \cdots, m\}$ so dass $U = \bigcup_{i \in I} U_i$.
- **Entscheidungsvariante:** Entscheide für $U_1, \cdots, U_m$ und $k \in \mathbb{N}$ ob eine Überdeckung von U mit höchstens k Teilmengen existiert.
- **Spezialfall:** $\mathcal{U} = Pl(f)$, Menge der Primimplikanten einer Booleschen Funktion f, Ausgabe minimale DNF-Formel für f.
- **Anmerkung:** $\bigcup_{m \in Pl(f)} m^{-1}(1) = f^{-1}(1)$.

Schwierige Probleme: Das Rucksack Problem (RP)

- **Eingabe:** n Objekte für den Rucksack, Gewichtsvektor $w = (w_1, \cdots, w_n) \in \left(\mathbb{R}^+\right)^n$, Nutzenvektor $c = (c_1, \cdots, c_n) \in \left(\mathbb{R}^+\right)^n$, Gewichtsschranke W.
- **Ausgabe:** Menge $I \subseteq [n] = \{1, \cdots, n\}$ von Objekten, $w(I) = \sum_{i \in I} w_i \leq W$, $c(I) = \sum_{i \in I} c_i$ maximal.
- **Entscheidungsvariante:** Eingabe (n, c, w, W) wie oben, zusätzlich Nutzenschranke C, entscheide ob Menge $I \subseteq [n]$ von Objekten existiert, mit $w(I) \leq W$ und $c(I) \geq C$.
- **Spezialfall PARTITION:** Entscheide für $a = (a_1, \cdots, a_n) \in \left(\mathbb{R}^+\right)^n$ ob Menge $I \subseteq [n]$ mit $a(I) = a([n] \setminus I)$ existiert.

Schwierige zahlentheoretische Berechnungsprobleme

- **DiscreteLog:** Eingabe n-Bit Primzahl p, Erzeugendes g der Gruppe $\mathbb{Z}_p^*$ und $y \in \mathbb{Z}_p^*$, berechne eindeutiges $x \in \mathbb{Z}_{p-1}$ mit $y = g^x \mod p$. (d.h. $x = \log_g(y)$ in $\mathbb{Z}_p^*$).
- **Faktorisierung:** Eingabe n-Bit Zahl m, berechne die Primfaktorenzerlegung von m.
- Die Berechnung von $g^x \mod p$ für gegebenes x, sowie die Integer Multiplikation sind effizient berechenbar.

Das Satisfiability Problem SAT

Wir definieren im Folgenden das SAT-Problem

für aussagenlogische Formeln

Aussagenlogische Formeln (AL-Formeln), Syntax

Definition 36

Aussagenlogische (AL-Formeln) über einer Menge $\{X_1, \cdots, X_n\}$ von AL-Variablen sind definiert als:

- Die Konstanten 0 und 1, sowie die AL-Variablen X_i, $1 \leq i \leq n$, sind AL-Formeln über $\{X_1, \cdots, X_n\}$.
- Ist F eine AL-Formel über $\{X_1, \cdots, X_n\}$, so ist $\neg(F)$ AL-Formel über $\{X_1, \cdots, X_n\}$.
- Sind F, G AL-Formeln über $\{X_1, \cdots, X_n\}$, so auch $(F) \wedge (G)$ und $(F) \vee (G)$.

Beispiele

- Literale X_i, $\neg X_i$
- $(\neg X_1 \wedge X_2) \vee (\neg X_2 \wedge X_1)$
- $\neg X_1 \vee X_2$

Aussagenlogische Formeln (AL-Formeln), Semantik

Definition 37

AL-Formeln F über $\{X_1, \cdots, X_n\}$ berechnen Boolesche Funktionen $F : \{0,1\}^n \longrightarrow \{0,1\}$.

Für alle Belegungen $b = (b_1, \cdots, b_n) \in \{0,1\}^n$ von $\{X_1, \cdots, X_n\}$ gelte:

- Falls $F = 0$ bzw. $F = 1$, so gilt $F(b) = 0$ bzw. $F(b) = 1$.
- Falls $F = X_i$ so gilt $F(b) = b_i$.
- Falls $F = \neg(G)$ so gilt $F(b) = 1 - G(b)$.
- Ist $F = (G) \wedge (H)$ so gilt $F(b) = \min\{G(b), H(b)\}$.
- Ist $F = (G) \vee (H)$ so gilt $F(b) = \max\{G(b), H(b)\}$.

Die Tabelle $\{(b, F(b)), b \in \{0,1\}^n\}$ heißt **Wahrheitstabelle** der AL-Formel F über $\{X_1, \cdots, X_n\}$.

Spezielle AL-Formeln über $\{X_1, \cdots, X_n\}$

- Literale X_i, $\neg X_i$
- Monome $M = L_1 \wedge L_2 \wedge \cdots \wedge L_m$, L_j Literale
- Klauseln $C = L_1 \vee L_2 \vee \cdots \vee L_m$, L_j Literale
- Formeln in disjunktiver Normalform (DNF-Formeln) $D = M_1 \vee \cdots \vee M_k$, M_j Monome
- Formeln in konjunktiver Normalform (CNF-Formeln) $C = C_1 \wedge \cdots \wedge C_k$, C_j Klauseln

Achtung: Monome und Klauseln können als **vereinfacht** vorausgesetzt werden, d.h. für all $i = 1, \cdots, n$ kommt höchstens ein Literal aus $\{X_i, \neg X_i\}$ in einem Monom bzw. einer Klausel vor.

Das liegt an den Regeln $X_i \wedge X_i = X_i \vee X_i = X_i$, $\neg X_i \wedge \neg X_i = \neg X_i \vee \neg X_i = \neg X_i$, $X_i \wedge \neg X_i = 0$, $X_i \vee \neg X_i = 1$.

Das Erfüllbarkeitsproblem (Satisfiability, SAT)

Definition 38

- Eine AL-Formel $F = F(X_1, \cdots, X_n)$ heißt **erfüllbar**, falls eine erfüllende Belegung $b \in \{0,1\}^n$ (d.h., eine für die $F(b) = 1$ gilt, existiert.
- Die Sprache SAT besteht aus allen erfüllbaren CNF-Formeln $C = C_1 \wedge \cdots \wedge C_m$.
- Die Sprache k-SAT ($k \geq 1$) besteht aus allen erfüllbaren CNF-Formeln $C = C_1 \wedge \cdots \wedge C_m$, deren Klauseln höchstens k Literale enthalten.

Das SAT Problem ist ein schwieriges Berechnungsproblem, das in vielen wichtigen Anwendungszusammenhängen (besonders im Bereich künstliche Intelligenz) gelöst werden muss.

SAT ist berechenbar

Der Algorithmus **Vollständige Suche** für SAT:

Eingabe: CNF-Formel $C = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ über $\{X_1, \cdots, X_n\}$.

1 **For all** $b \in \{0,1\}^n$
2 **do if** $C_1(b) = \cdots = C_m(b) = 1$
3 **then output** C **erfüllbar, stop**
4 **output** C **nicht erfüllbar**

- Die Berechnung von Zeile 2 kostet $O(m \cdot n)$ (ÜA).
- Die Worst Case Rechenzeit vollständiger Suche ist $\Theta(m \cdot n \cdot 2^n)$.

Effizient verifizierbare Beweise

Definition 39

Ein Entscheidungsproblem Π hat ein **System effizient verifizierbarer Beweise** $(Proof_\Pi, V_\Pi)$, falls

- $Proof_\Pi$ ordnet jeder Eingabe x für Π eine Menge $Proof_\Pi(x)$ **möglicher Beweise** zu.
- $V_\Pi = V_\Pi(x, y)$ bezeichnet einen Verifikationsalgorithmus polynomieller Laufzeit in $|x|$, der für alle Eingaben x für Π und jeden möglichen Beweis $y \in Proof_\Pi(x)$ testet, ob y ein Beweis für die Behauptung $\Pi(x) = 1$ ist.
- Es gilt dass $\Pi(x) = 1$ genau dann, wenn ein möglicher Beweis $y \in Proof_\pi(x)$ mit $V_\Pi(x, y) = 1$ existiert.

Ein entsprechendes Kommunikationsszenario

Ein System effizient verifizierbarer Beweise $(\mathit{Proof}_\Pi, V_\Pi)$ für ein Entscheidungsproblem Π definiert das folgende Spiel zwischen einer (potentiell genialen) Alice und einem *normal boy* Bob, der nur Probleme in P lösen kann.

- **Alice** behauptet: Für diese Eingabe x gilt $\Pi(x) = 1$!
- **Bob:** Beweise es !
- **Alice** präsentiert $y \in \mathit{Proof}_\Pi(x)$.
- **Bob** verifiziert durch Ausführung von $V_\Pi(x, y)$ ob y die Behauptung $\Pi(x) = 1$ beweist.

Achtung: Die Effizienz von V_Π bedingt, dass die Bitlänge der Beweise in $\mathit{Proof}_\Pi(x)$ polynomiell in $|x|$ beschränkt ist.

Formaleres Modell: Polynomielle Rate + Verifiziere Algorithmen

Ein **Polynomieller Rate + Verifiziere Algorithmus** für ein Entscheidungsproblem Π (kurz PRV-Algorithmus) ist eine Paar (p, V) für das folgendes gilt:

- $p : \mathbb{N} \longrightarrow \mathbb{N}$ ist eine polynomiell beschränkte Funktionen.
- V ist ein Polynomialzeitalgorithmus mit Ausgabe $\{0, 1\}$, dessen Eingabe Paare (x, y) mit $y \in \{0, 1\}^{p(|x|)}$ sind.
- Es gilt $\Pi(x) = 1$ genau dann, wenn ein $y \in \{0, 1\}^{p(|x|)}$ mit $V(x, y) = 1$ existiert (in dem Fall heißt y **Beweis für** $\Pi(x) = 1$**)**.

Beobachtung: Jeder PRV-Algorithmus (p, V) für Π definiert ein System effizient verifizierbarere Beweise $(Proof_\pi, V)$ für Π mit $V_\pi = V$ und $Proof_\pi(x) = \{0, 1\}^{p(|x|)}$.

Effizient verifizierbare Beweise, Beispiele

Beispiel: Effizient verifizierbare Beweise für *SAT*:

- **Eingabe:** CNF-Formul $C = C_1 \wedge \cdots \wedge C_m$ über $\{x_1, \cdots, x_n\}$
- **Beweise:** $Proof_{SAT}(C) = \{0,1\}^n$ (entsprechen den möglichen 0, 1-Belegungen von $\{x_1, \cdots, x_n\}$)
- **Verifikation** $V(C, b) = 1$ genau dann, wenn b erfüllt C, d.h., $C_j(b) = 1$ für alle $j = 1, \cdots, m$

Beispiel: Effizient verifizierbare Beweise für *HC*:

- **Eingabe:** Ungerichteter Graph $G = (V, E)$ über $\{v_1, \cdots, v_n\}$
- **Beweise:** $Proof_{HC}(G) = \mathcal{S}_n$, die Menge der Permutationen über $\{1, \cdots, n\}$ (entsprechend den möglichen Hamiltonkreisen).
- **Verifikation** $V(G, \pi) = 1$ genau dann, wenn Kreis $\{(v_{\pi(1)}, v_{\pi(2)}), \cdots, (v_{\pi(n-1)}, v_{\pi(n)}), (v_{\pi(n)}, v_{\pi(1)})\} \subseteq E$

Effizient verifizierbare Beweise, weitere Beispiele

Beispiel: Effizient verifizierbare Beweise für *Clique*:

- **Eingabe:** Ungerichteter Graph $G = (V, E)$ und Schranke $k \leq |V|$.
- **Beweise:** $Proof_{Clique}(G, k) = \{V';\ V' \subseteq V, |V'| = k\}$ entsprechend den möglichen k-Cliquen in G.
- **Verifikation:** $V((G, k), V') = 1$ genau dann, wenn $(v, v') \in E$ für alle $v \neq v' \in V'$, d.h. falls V' Clique in G.

Beispiel: Effizient verifizierbare Beweise für *TSP*:

- **Eingabe:** Distanzmatrix $D = (d_{i,j})_{i,j=1}^{n}$ für n Städte, und Schranke d.
- **Beweise:** $Proof_{TSP}(D, d) = \mathcal{S}_n$ entsprechend den möglichen Rundfahrten
- **Verifikation:** $V((D, d), \pi) = 1$ genau dann, wenn $d_{\pi(1),\pi(2)} + \cdots + d_{\pi(n-1),\pi(n)} + d_{\pi(n),\pi(1)} \leq d$.

Effizient verifizierbare Beweise, weiteres Beispiel

Beispiel: Effizient verifizierbare Beweise für RP:

- **Eingabe:** $I = (n, w, c, W, C)$ entsprechend n Objekten, d.h., Gewichtsvektor $w = (w_1, \cdots, w_n)$, Nutzenvektor $c = (c_1, \cdots, c_n) \in \mathbb{N}^n$, Gewichtsschranke $W \in \mathbb{N}$, Nutzenschranke $C \in \mathbb{N}$
- **Beweise:** $Proof_{RP}(n, w, c, W, C) = \{I; I \subseteq \{1, \cdots, n\}\}$ entsprechend den Teilmengen der Menge der n Objekte.
- **Verifikation:** $V((n, w, c, W, C), I) = 1$ falls $w(I) = \sum_{i \in I} w_i \leq W$ und $c(I) = \sum_{i \in I} c_i \geq C$.

Die Klassen P und NP

Definition 40

P bezeichne die Klasse der Entscheidungsprobleme, die einen Polynomialzeitalgorithmus (d.h., eine polynomiell zeitbeschränkte Turing-Maschine) haben.

Definition 41

NP bezeichne die Klasse der Entscheidungsprobleme, die ein System effizient verifizierbarer Beweise bzw. einen Polynomiellen Rate + Verifiziere Algorithmus haben.

- **Beobachtung:** $P \subseteq NP$ (leichte Übungsaufgabe)
- **Beobachtung:** $SAT, Clique, HC, TSP, RP$ usw. liegen in NP.
- **Anmerkung:** Die Klasse NP läßt sich auch mittels nichtdeterministischer Turingmaschinen bzw. randomisierter Algorithmen definieren.

Background PNP-Problem

Das Problem $P \neq NP$? berührt ein wichtiges Problem im Bereich künstliche Intelligenz:

Ist es wesentlich komplizierter, einen Beweis für ein mathematisches Problem zu finden, als die Korrektheit eines gegebenen Beweises zu verifizieren?

$P \neq NP$ würde nahelegen, dass Beweisfindung wesentlich komplizierter ist als Beweisverifizierung.

$P = NP$ würde nahelegen, dass kreative Schaffensprozesse (wie die Findung eines Beweises für ein schwieriges mathematisches Problem) im Prinzip durch Computer effizient simulierbar sind.

Polynomielle Reduzierbarkeit

Polynomielle Reduzierbarkeit (Schreibweise $L \leq_{pol} L'$) bildet eine Relation zwischen Sprachen L und L'.

$L \leq_{pol} L'$ bedeutet, dass L nicht wesentlich schwieriger zu berechnen ist, als L'.

Definition 42

- Eine Abbildung $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt polynomielle Reduktion der Sprache $L \subseteq \{0,1\}^*$ auf die Sprache $L' \subseteq \{0,1\}^*$ falls f in polynomieller Zeit berechenbar ist, und für alle Eingabe $x \in \{0,1\}^*$ gilt

$$x \in L \iff f(x) \in L'.$$

- L heißt polynomiell reduzierbar auf L' (Schreibweise $L \leq_{pol} L'$), falls eine polynomielle Reduktion von L auf L' existiert.

Eigenschaften polynomieller Reduzierbarkeit

Lemma 43

Es sei f polynomielle Reduktion von $L \subseteq \{0,1\}^*$ auf $L' \subseteq \{0,1\}^*$.

(1) *Aus $L' \in NP$ folgt $L \in NP$.*

(2) *Aus $L' \in P$ folgt $L \in P$.*

Beweis (1): Es sei $(Proof_{L'}, V_{L'})$ ein System effizient verifizierbarer Beweise für L'. Wir konstruieren daraus ein System $(Proof_L, V_L)$ effizient verifizierbarer Beweise für L:

- Für alle $x \in \{0,1\}^*$ sei $Proof_L(x) = Proof_{L'}(f(x))$
- Für alle $x \in \{0,1\}^*$ und $\pi \in Proof_L(x) = Proof_{L'}(f(x))$ sei

$$V_L(x, \pi) = V_{L'}(f(x), \pi).$$

Man rechnet leicht nach, dass $(Proof_L, V_L)$ ein System effizient verifizierbarer Beweise für L ist.

Eigenschaften polynomieller Reduzierbarkeit, Beispiel

Beweis (2): Es sei A' ein Polynomialzeitalgorithmus für L' und A_f ein Polynomialzeitalgorithmus für f.

Wir konstruieren einen Polynomialzeitalgorithmus A für L:

Für alle $x \in \{0,1\}^*$ sei $A(x) = A'(A_f(x))$. $\square$

Beispielreduktion *HC* auf *TSP*:

f ordnet jeder *HC*-Eingabe $G = (V, E)$ über $V = \{v_1, \cdots, v_n\}$ die *TSP*-Eingabe (D, n) zu, wobei $D = (d_{i,j})_{i,j=1}^n$ mit $d_{i,j} = 1$ falls $(v_i, v_j) \in E$ und $d_{i,j} = 2$ falls $(v_i, v_j) \notin E$.

Man rechnet leicht nach, dass für alle Permutationen $\pi \in \mathcal{S}_n = \mathit{Proof}_{HC}(G) = \mathit{Proof}_{TSP}(D, n)$ gilt

$$V_{HC}(G, \pi) = V_{TSP}((D, n), \pi),$$

woraus folgt, dass $G \in HC$ genau dann, wenn $(D, n) \in TSP$.

NP-Vollständigkeit

Definition 44

- Eine Sprache $L \in NP$ heißt NP-vollständig, falls für alle $L' \in NP$ gilt $L' \leq_{pol} L$.
- NPC bezeichnet Klasse NP-vollständigen Sprachen.

Theorem 45

Falls $P \neq NP$ so gilt $P \cap NPC = \emptyset$, d.h. NP-vollständige Probleme haben keinen Polynomialzeitalgorithmus.

Beweis: Sei $L \in P \cap NPC$ so gilt für alle $L' \in NP$ dass $L' \leq_{pol} L$ also $L' \in P$ also $P = NP$. $\square$

Wichtige Frage: Gibt es NP-vollständige Probleme?

Der Satz von Cook

Stephen A. Cook, geb.1939, lehrt an der University of Toronto.

Theorem 46

(Cook 1971) SAT ist NP-vollständig.

Beweis: Sei $L \in NP$. Beschreiben einen Polynomialzeitalgorithmus, der jeder Eingabe $x \in \{0,1\}^*$ eine CNF-Formel C^x zuordnet. Konstruktion basiert auf:

- Es gibt Polynome $P = P(n)$ und $T = T(n)$, sowie eine *spezielle* TM M, so dass für alle $x \in \{0,1\}^*$ gilt: $x \in L$ genau dann, wenn ein $y \in \{0,1\}^{P(|x|)}$ existiert, so dass $M(x,y) = 1$.
- $M = (Q, q_0, \delta)$ ist 1-Band TM über $\{0,1,\#\}$, die auf jeder Eingabe x, y, $x \in \{0,1\}^*$, $y \in \{0,1\}^{P(|x|)}$ genau $T(|x|)$ Takte rechnet, $Q = \{q_0, \cdots, q_S\}$, q_S einziger akzeptierender Zustand.

Beweis Satz von Cook, II

Sei $n \in \mathbb{N}$ beliebig, $x \in \{0,1\}^n$, $P = P(n)$, $T = T(n)$. Es gilt $x \in L$ genau dann, wenn $\exists y \in \{0,1\}^P$, M-Konfigurationen $K_0 \xrightarrow{M} K_1 \xrightarrow{M} \cdots \xrightarrow{M} K_T$ mit $K_0 = K_0^M(x,y)$, K_T akzeptierende Stoppkonfiguration. Kodieren M-Konfigurationen durch Belegungen folgender Aussagevariablen:

- $H = \{H_{j,b}^t, t \in \{0,\cdots,T\}, j \in \{-T,\cdots,T\}, b \in \{0,1,\#\},\}$,
- $R = \{R_j^t, t \in \{0,\cdots,T\}, j \in \{-T,\cdots,T\}\}$,
- $Q = \{Q_s^t, s \in \{0,\cdots,S\}\}$.

$Q_s^t = 1$ bzw. $R_j^t = 1$ bzw. $H_{j',b}^t = 1$ genau dann, wenn M zum Zeitpunkt t im Zustand q_s ist, der Kopf auf Position j und im Feld j' Zeichen b steht.

Beweis Satz von Cook, III

Konstruieren CNF-Formel C^x über $H \cup R \cup Q \cup Y$, $Y = \{Y_1, \cdots, Y_P\}$, so dass die erfüllenden Belegungen von C_x genau den akzeptierenden Berechnungen von M auf x, y, y Belegung von Y, entsprechen. Es gilt $C^x = C_1^x \wedge C_2^x \wedge C_3^x \wedge Q_S^T$, wobei

- C_1^x erfüllt genau dann wenn Belegung korrekt eine Konfigurationenfolge $K_0, \cdots K_T$ kodiert,
- C_2^x erfüllt genau dann wenn $K_0 = K_0(x, y)$, y Belegung von Y,
- C_3^x erfüllt genau dann wenn $K_t \xrightarrow{M} K_{t+1}$ für alle $t = 0, \cdots, T-1$.

Beweis Satz von Cook, Konstruktion C_1^x

C_1^x ist erfüllt wenn zu jedem Zeitpunkt t genau eine R^t- und genau eine Q^t-Variable und für alle $j = -T, \cdots, T$ genau eine H_j^t-Variable erfüllt ist. Die Konstruktion benutzt dass

$$F(z_1, \cdots, z_m) = \left(\bigvee_{i=1}^{m} z_i \right) \wedge \bigwedge_{1 \leq i \neq j \leq m} (\neg z_i \vee \neg z_j),$$

genau dann erfüllt, wenn genau eine z-Variable erfüllt.

$$C_1^x = \bigwedge_{t=0}^{T} \Big(F(R^t_{-T}, \cdots, R^t_T) \wedge F(Q^t_0, \cdots, Q^t_S) \wedge \bigwedge_{j=-T}^{T} F\left(H^t_{j,\#}, H^t_{j,0}, H^t_{j,1}\right) \Big).$$

Beweis Satz von Cook, , Konstruktion C_2^x

C_2^x erfüllt wenn $K_0 = K_0^M(x, y)$, y Belegung von Y. Konstruktion benutzt dass

$$sel(y, z_0, z_1) = (y \vee z_0) \wedge (\neg y \vee z_1)$$

genau dann erfüllt wenn aus $y = 0$ folgt $z_0 = 1$ und aus $y = 1$ folgt $z_1 = 1$.

$$C_2^x = \bigwedge_{j=-T}^{0} H_{j,\#}^0 \wedge \bigwedge_{j=1}^{n} H_{j,x_j}^0 \wedge H_{n+1,\#}^0 \wedge \bigwedge_{j=1}^{P} sel(y_j, H_{n+1+j,0}^0, H_{n+1+j,1}^0) \wedge$$

$$\bigwedge_{j=n+P+2}^{T} H_{j,\#}^0.$$

Beweis Satz von Cook, Konstruktion C_3^x

C_3^x genau dann erfüllt, wenn $K_t \xrightarrow{M} K_{t+1}$ für alle $t = 0, \cdots, T-1$. Sei $\delta = \{I_1, \cdots, I_U\}$, $I_u = (q_{i_u}, b_u | b'_u, m_u, q_{j_u})$,

$$C_3^x = \bigwedge_{t=0}^{T-1} \bigwedge_{u=1}^{U} \bigwedge_{j=-T}^{T} \left(\left(Q_{i_u}^t \wedge R_j^t \wedge H_{j,b_u}^t \right) \longrightarrow \left(H_{j,b'_u}^{t+1} \wedge R_{m_u(j)}^{t+1} \wedge Q_{j_u}^{t+1} \right) \right).$$

- Es sei $m(j) = j - 1$ für $m = L$, $m(j) = j + 1$ für $m = R$, $m(j) = j$ für $m = N$.
- Man beachte dass

$$(z_1 \wedge z_2 \wedge z_3) \longrightarrow (z_4 \wedge z_5 \wedge z_6) \equiv (\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_4) \wedge$$

$$(\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_5) \wedge (\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_6). \square$$

Reduktionsmethode zum Zeigen von *NP*-Vollständigkeit

Theorem 47

Séi $L' \in NPC$, $L \in NP$, $L' \leq_{pol} L$, dann $L \in NPC$.

Beweis: Es bezeichne $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ eine polynomielle Reduktion vom NPC-Problem L' auf das *NP*-Problem L.

Es sei $\tilde{L} \in NP$ beliebig fixiert, und es bezeichne $g : \{0,1\}^* \longrightarrow \{0,1\}^*$ eine polynomielle Reduktion vo $\tilde{L}$ auf L'. Diese existiert da $L' \in NPC$.

Die Abbildung $f \circ g$ ist eine polynomielle Reduktion von $\tilde{L}$ auf L, da $f \circ g \in PTIME$ und

$$f(g(x)) \in L \Longleftrightarrow g(x) \in L' \Longleftrightarrow x \in \tilde{L}.$$

Also gilt $\tilde{L} \leq_{pol} L$ für alle $\tilde{L} \in NP$, d.h. L ist *NP*-vollständig. □

Eine Polynomielle Reduktion von SAT auf $3SAT$, I

Theorem 48

$3SAT \in NPC$.

Beweis: Eine polynomiellen Reduktion von SAT auf $3SAT$:

Sei $C = C_1 \wedge \cdots \wedge C_m$ eine CNF-Formel über $X = \{x_1, \cdots, x_n\}$. Wir konstruieren $3CNF$-Formel

$$D = D_1 \wedge \cdots \wedge D_m$$

über $X \cup Y^1 \cup \cdots \cup Y^m$ mit

$$C \in SAT \Longleftrightarrow D \in 3SAT.$$

Hierbei bezeichnet D^j für alle $j = 1, \cdots, m$ eine 3CNF-Formel über den Variablen $X \cup Y^j$, Y^j ist eine Menge von Hilfsvariablen die nur in D_j vorkommt.

Polynomielle Reduktion SAT auf $3SAT$,II

Konstruktion von D_j über $X \cup Y^j$ für $j = 1, \cdots, m$:

- Annahme: $C_j = L^j_1 \vee \cdots \vee L^j_{s_j}$.
- Falls $s_j \leq 3$ so ist $D_j = C_j$,
- Falls $s_j \geq 4$ so ist gilt $Y^j = \{y^j_1, \cdots, y^j_{s_j-3}\}$.
- D_j besteht dann aus den $s_j - 2$ Klauseln

$$D_j = (L^j_1 \vee L^j_2 \vee y^j_1) \ \wedge \ (\neg y^j_1 \vee L^j_3 \vee y^j_2) \ \wedge \cdots$$

$$\cdots \wedge \ (\neg y^j_{s_j-4} \vee L^j_{s_j-2} \vee y^j_{s_j-3}) \ \wedge \ (\neg y^j_{s_j-3} \vee L^j_{s_j-1} \vee L^j_{s_j}).$$

Lemma 49

Für alle Belegungen b von X gilt: Es existiert genau dann eine Belegung c von Y^j mit $D_j(b,c) = 1$ wenn $C_j(b) = 1$.

Beweis des Lemmas:

Annahme: $C_j(b) = 0$, d.h. $L^j_k(b) = 0$ für alle $k = 1, \cdots, s_j$.

Dann hat $D_j|_b$ die Gestalt

$$y^j_1(\neg y^j_1 \vee y^j_2) \vee \cdots \vee (\neg y^j_{s_j-4} \vee y^j_{s_j-3})\neg y^j_{s_j-3},$$

d.h. $D_j|_b$ ist nicht erfüllbar.

$SAT \leq_{pol} 3SAT$, (III)

Annahme: $C_j(b) = 1$, d.h. $L_k^j(b) = 1$ für mindestens ein k aus $1, \cdots, s_j$.

Dann hat $D_j|_b$ die Gestalt $E \wedge F_1 \wedge \cdots \wedge F_r \wedge G$,

wobei alle Teilformeln paarweise disjunkte Variablenbereiche haben, und

- E vom Typ $z_1(\neg z_1 \vee z_2) \wedge \cdots \wedge (\neg z_{t-1} \vee z_t)$,
- alle F_u vom Typ $(\neg u_1 \vee u_2)(\neg u_2 \vee u_3) \wedge \cdots \wedge (\neg u_{t-1} \vee u_t)$,
- und G vom Typ $(\neg v_1 \vee v_2)(\neg v_2 \vee v_3) \wedge \cdots \wedge (\neg v_{t-1} \vee v_t) \wedge \neg v_t$

ist.

Da alle Teilformeln erfüllbar sind, ist auch $D_j|_b$ erfüllbar. Somit gilt $D \in SAT$ genau dann, wenn es eine Belegung von b gibt, die alle C_j erfüllt (d.h. $C \in SAT$). $\square$

Clique ist *NP*-vollständig.

Theorem 50

Clique $\in$ *NPC.*

Beweis: Wir konstruieren eine polynomielle Reduktion von 3*SAT* auf *Clique*. Sei $C = C_1 \wedge \cdots \wedge C_m$ 3*CNF*-Formel über $X = \{x_1, \cdots, x_n\}$, konstruieren Graph $G = (V, E)$, $|V| = 3m$, mit C genau dann erfüllbar, wenn G eine Clique der Größe m hat.

- OBdA sei $C^j = L_1^j \vee L_2^j \vee L_3^j$, für alle $j = 1, \cdots, m$.
- $V = V^1 \cup V^2 \cup \cdots \cup V^m$, $V^j = \{v_1^j, v_2^j, v_3^j\}$.
- Es sei $(v_a^j, v_b^k) \in E$, falls $j \neq k$ und $L_a^j \neq \neg L_b^k$.
- **Beobachtung:** Jede Teilmenge $\{v_{a_1}^{j_1}, \cdots, v_{a_m}^{j_m}\}$ ist genau dann m-Clique, wenn $j_k \neq j_l$ für alle $1 \leq k \neq l \leq m$, und $\{L_{a_1}^{j_1}, \cdots, L_{a_m}^{j_m}\}$ sind simultan erfüllbar. $\square$

The NP-Completeness of Knapsack

Definition 51

The special knapsack problem KP^* consists of input instances $w = (w_1, \cdots, w_n), W$, where $w_i, W \in \mathbb{N}$, for which there is a subset $I \subseteq \{1, \cdots, n\}$ fulfilling $\sum_{i \in I} w_i = W$.

One can easily show that $KP^* \leq_{pol} KP$.

Theorem 52

$3SAT \leq_{pol} KP^*$, *i.e.,* $KP^* \in NPC$.

Proof: We assign to each $3CNF$-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X = \{x_1, \cdots, x_n\}$ numbers a^i, b^i for $i = 1, \cdots, n$, and c^j, d^j for $j = 1, \cdots, m$ and a weight number W. All these numbers are written as vectors of length $n + m$ over $\{0, \cdots, 9\}$ and interpreted as decimal numbers with $n + m$ digits.

The *NP*-Completeness of *Knapsack* (2)

For all $i = 1, \cdots, n$ the numbers a^i and b^i encode the occurences of literal x_i resp. $\neg x_i$ in C, where c^j and d^j denote helping numbers.

- For all $i = 1, \cdots, n$ let $a^i_i = b^i_i = 1$ and $a^i_k = b^i_k = 0$ for all k, $1 \leq k \leq n$, $k \neq i$, i.e., the leading n components of a^i and b^i correspond to the i-th unit vector.
- For all $i = 1, \cdots, n$ and $j = 1, \cdots, m$ let $a^i_{n+j} = 1$ (resp. $b^i_{n+j} = 1$) iff literal x_i (resp. literal $\neg x_i$) occurs in clause C_j, otherwise let $a^i_{n+j} = 0$ (resp. $b^i_{n+j} = 0$).
- For all $j = 1, \cdots, m$, for the number c^j (resp. d^j) it holds $c^j_{n+j} = 1$ (resp. $d^j_{n+j} = 2$) and $c^j_k = d^j_k = 0$ for all $k = 1, \cdots, n+m$, $k \neq n+j$.

The weight W is defined by $W_i = 1$ for $i = 1, \cdots, n$, and $W_{n+j} = 4$ for $j = 1, \cdots, m$.

The *NP*-Completeness of *Knapsack* (3)

Obviously, the numbers a^i, b^i as well as $c^j, d^{j'}$ and W can be constructed from C in polynomial time in n and m.

We show now that there are subsets I, I' of $\{1, \cdots, n\}$ and J, J' of $\{1, \cdots, m\}$ such that

$$\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} + \sum_{j \in J} c^j + \sum_{j' \in J'} d^{j'} = W \tag{1}$$

if and only if $C \in 3SAT$.

The leading n ones of W imply that for all $i = 1, \cdots, n$ either $i \in I$ or $i \in I'$, i.e., by determining I we determine an assignment $\beta^I \in \{0,1\}^n$ by $\beta^I_i = 1$ if $i \in I$ and $\beta^I_i = 0$ if $i \in I'$.

We show that I and $I' = [n] \setminus I$ can be completed by sets J, J' of $\{1, \cdots, m\}$ fulfilling (1) iff β^I satisfies C. Note that this completes the proof of our theorem.

The *NP*-Completeness of *Knapsack* (4)

Suppose first that $C(\beta^I) = 1$. Then each clauses C_j contains a literal x_i for which $i \in I$ or a literal $\neg x_i$ for which $i \in I'$.

This implies that for all $j = 1, \cdots, m$ it holds that

$$\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} \right)_{n+j} \in \{1, 2, 3\}.$$

The desired sets J, J' can be constructed as follows.

- If $\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} \right)_{n+j} = 1$ then add j to J and to J'.
- If $\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} \right)_{n+j} = 2$ then add j only to J'.
- If $\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} \right)_{n+j} = 3$ then add j only to J.

The *NP*-Completeness of *Knapsack* (5)

It can be easily checked that $I, I' = [n] \setminus I, J, J'$ satisfy (1).

Suppose now that $C(\beta') = 0$, i.e., there is a clause C_j not containing any literal x_i, $i \in I$, and not containing any literal $\neg x_{i'}$, $i' \in I' = [n] \setminus I$. This implies that

$$\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} \right)_{n+j} = 0.$$

It can be easily shown that for all choices of J, J' it holds that

$$\left(\sum_{i \in I} a^i + \sum_{i' \in I'} b^{i'} + \sum_{j \in J} c^j + \sum_{j' \in J'} d^{j'} \right)_{n+j} < 4. \ \square$$

The *NP*-Completeness of *Partition*

Theorem 53

It holds $KP^* \leq_{pol}$ Partition, i.e., Partition is NP-complete.

Proof: We assign to each input instance $(w_1, \cdots, w_n), W$ for KP^* the input instance $(w_1, \cdots, w_n, S - W + 1, W + 1)$ for *Partition*, where $S = \sum_{i=1}^{n} w_i$.

It can be easily shown that this defines a polynomial reduction from KP^* to *Partition*. □

The *NP*-Completeness of *TSP* and *HC*

The *NP*-Completeness of *TSP* and *HC* follows from the reductions

$$3SAT \leq_{pol} DHC \leq_{pol} HC \leq_{pol} TSP.$$

We have already shown the third reduction.

DHC (Directed Hamiltonian Circuit) denotes the problem to decide if a given directed graph G contains a DHC, i.e., a directed circuit which visits each node exactly once.

The reduction $DHC \leq_{pol} HC$ is left to the reader as an exercise.

We show

Theorem 54

$3SAT \leq_{pol} DHC.$

The *NP*-Completeness of *DHC* (1)

The main component of the reduction from 3*SAT* to *DHC* is a graph $G^* = (V^*, E^*)$ with six nodes and nine edges.

- V^* consists of the entry nodes u_1, u_2, u_3 and the exit nodes U_1, U_2, U_3.
- E^* contains the two antiparallel 3-cycles $(u_1, u_2), (u_2, u_3), (u_3, u_1)$ and $(U_1, U_3), (U_3, U_2), (U_2, U_1)$ interconnected via $(u_1, U_1), (u_2, U_2), (u_3, U_3)$.

Let us fix a 3*CNF*-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X = \{x_1, \cdots, x_n\}$. We suppose w.l.o.g. that each clause C_j contains exactly three Literals, $C_j = L_1^j \vee L_2^j \vee L_3^j$.

The *NP*-Completeness of *DHC* (2)

We assign to C a graph $G = (V, E)$ over $6m + n$ nodes consisting of

- m disjoint components $G_1, \cdots, G_m$. Each G_j is a copy of G^* over the nodes $u^j_1, u^j_2, u^j_3, U^j_1, U^j_2, U^j_3$.
- n additional nodes $v_1, \cdots, v_n$.
- Additionally, for all $i = 1, \cdots, n$ there is an $(i, 1)$-path and an $(i, 0)$-path. The $(i, 1)$-path (resp. the $(i, 0)$-path) starts in node v_i, visits all components G_j for which $x_i \in C_j$ (resp. $\neg x_i \in C_j$) and ends in node $v_{(i+1 \mod n)}$.

We still have to describe how an $(i, 1)$-path (resp. $(i, 0)$-path) visits a component G_j.

The *NP*-Completeness of *DHC* (3)

Let $C_{j_1}, \cdots, C_{j_s}$ denote the clauses containing x_i (resp. $\neg x_i$), where $j_1 < j_2 < \cdots < j_s$.

Fix values $k_1, \cdots, k_s$ from $\{1, 2, 3\}$ such that the literals $L^{j_1}_{k_1}, \cdots, L^{j_s}_{k_s}$ equal x_i (resp. $\neg x_i$).

Then the $(i, 1)$-path (resp. the $(i, 0)$-path) consists of the edges

$$(v_i, u^{j_1}_{k_1}), (U^{j_1}_{k_1}, u^{j_2}_{k_2}), \cdots, (U^{j_{s-1}}_{k_{s-1}}, u^{j_s}_{k_s}), (U^{j_s}_{k_s}, v_{(i+1 \mod n)}).$$

Note that E consists of the edges inside the G_j components and the edges of all $(i, 1)$- and $(i, 0)$-pathes.

The *NP*-Completeness of *DHC* (4)

We have to show that $C \in 3SAT$ if and only if $G \in DHC$.

Therefor we try to construct a *DHC* in G.

Let the *DHC* start in v_1. It has to choose and to follow either the $(1,1)$-path or the $(1,0)$-path and reaches v_2 and so on.

Thus, each *DHC*-candidate identifies exactly one assignment $\beta \in \{0,1\}^n$: For all $i = 1, \cdots, n$ let $\beta_i = 1$ if the *DHC*-candidate follows, starting from v_i, the $(i,1)$-path and $\beta_i = 0$ if it follows the $(i,0)$-path.

Let us call this *DHC*-candidate the *DHC*(β)-candidate.

Under which circumstances can the *DHC*(β)-candidate be extended to a *DHC* in G?

The *NP*-Completeness of *DHC* (5)

If there is a clause C_j for which $C_j(\beta) = 0$ then non of the β_i-paths visits G_j, this $DHC(\beta)$-candidate can not be extended to a *DHC*.

Let us now suppose that $C_j(\beta) = 1$ for all $j = 1, \cdots, m$, i.e., $C(\beta) = 1$ and $C \in SAT$.

We have to show that this $DHC(\beta)$-candidate be extended to a *DHC*.

For doing this we have to observe that the following three possibilities are the only possibilities how a *DHC* in G visits a component G_j:

The *NP*-Completeness of *DHC* (6)

(1) The *DHC* catches all six nodes with one visit to G_j.

$$\cdots \to u^j_1 \to u^j_2 \to u^j_3 \to U^j_3 \to U^j_2 \to U^j_1 \to \cdots .$$

(2) The *DHC* catches four nodes with one visit to G_j.

$$\cdots \to u^j_1 \to u^j_2 \to U^j_2 \to U^j_1 \to \cdots .$$

(2) The *DHC* catches two nodes with one visit to G_j: $\cdots \to u^j_1 \to U^j_1 \to \cdots$.

Observe that in all theses cases it holds that if the DHC enters G_j via u^j_k then it leaves G_j via U^j_k.

The *NP*-Completeness of *DHC* (7)

Let us now suppose that $C_j(\beta) = 1$ for all $j = 1, \cdots, m$. We describe how the *DHC*(β)-candidate can be extended to a *DHC*. We fix a component G_j and distinguish three cases.

- Suppose that β satisfies exactly one literal in C_j. Then exactly one (i, β_i)-path visits G_j. This (i, β_i)-path has to visit G_j via (1).
- Suppose that β satisfies exactly two literals in C_j. Then one of the two (i, β_i)-paths visiting G_j has to do this via (2) and one via (3).
- Suppose that β satisfies all three literals in C_j. Then all three (i, β_i)-paths visiting G_j have to do this via (3). □

The *NP*-Completeness of *HC* (1)

Theorem 55

It holds DHC $\leq_{pol}$ HC, i.e., HC is NP-complete.

Proof: We describe a polynomial reduction from *DHC* to *HC*. We assign to each directed graph $G = (V, E)$ an undirected graph $G' = (V', E')$ as follows

$$V' = \bigcup_{v \in V} \{v_l, v_m, v_r\}$$

$$E' = \bigcup_{v \in V} \{\{v_l, v_m\}, \{v_m, v_r\}\} \cup \bigcup_{(v,w) \in E} \{\{v_r, w_l\}\}\,.$$

The *NP*-Completeness of *HC* (2)

Suppose first that $G \in DHC$, i.e. G contains a *DHC*. By assigning to each edge (v, w) of this DHC the three edges $\{v_r, w_l\}, \{w_l, w_m\}, \{w_m, w_r\}$ of E', we obtain a *HC* in G'.

Suppose now that G' contains a *HC* and fix an edge $\{u_r, v_l\}$ in this *HC*. We go through the *HC* in the direction defined by taking first u_r and then v_l.

The next nodes along *HC* must be v_m and v_r, followed by some new w_l, otherwise v_m would never be reachable by the *HC*, and so on.

Each $\{s_r, t_l\}$ edge occuring in the *HC* corresponds to an edge (s, t) in E. It can be easily checked that all edges of this type in the *HC* form a *DHC* in G. □

Orakel-Algorithmen, Turing-Reduzierbarkeit

Erinnerung: Ein Berechnungsproblem Π ist **effizient berechenbar**, wenn es einen **Polynomialzeitalgorithmus für** Π gibt.

Ein Berechnungsproblem Π ist, **relativ zu einem Berechnungsproblem** Π'**, effizient berechenbar**, wenn es einen **Polynomialzeitalgorithmus für** Π **mit Zugriff auf ein** Π'**-Orakel** gibt.

Ein Π'-Orakel ist ein Unterprogramm, das Π' effizient löst. D.h., für alle Eingaben $x \in \Sigma^*$ liefert es ein y mit $(x, y) \in \Pi'$ in minimal möglicher Zeit $O(|x| + |y|)$.

Wir sagen, dass Π **Turing-reduzierbar auf** Π' ist (Schreibweise: $\Pi \leq_T \Pi'$), falls es einen Polynomialzeitalgorithmus für Π mit Zugriff auf ein Π'-Orakel gibt.

Formale Definition: Orakel-Turingmaschinen

Definition 56

Sei $\Pi, \Pi' \subseteq \Sigma \times \Sigma$ Berechnungsprobleme.

- Eine Orakel-Turingmaschine mit Orakel Π' (kurz Π'-OTM) ist eine TM mit einem zusätzlichen Orakelband und einem Orakelbefehl. Steht auf dem Orakelband $x \in \Sigma^*$, so bewirkt der Orakelbefehl, dass eine Lösung y mit $(x, y) \in \Pi'$ in Zeit $O(|x| + |y|)$ auf das Orakelband geschrieben wird.
- Es sei Π Turing-reduzierbar auf Π' (Schreibweise $\Pi \leq_T \Pi'$) falls es eine polynomiell zeitbeschränkte Π'-OTM für Π gibt.

Lemma 57

Aus $\Pi' \in$ *PTIME und* $\Pi \leq_T \Pi'$ *folgt* $\Pi \in$ *PTIME.* $\square$

Orakel-Algorithmen: Beispiele (1)

Lemma 58

Es seien $L, L' \subseteq \Sigma^*$ Sprachen. Dann folgt aus $L \leq_{pol} L'$ dass $L \leq_T L'$.

Beweis: Es bezeichne $f : \Sigma^* \longrightarrow \Sigma^*$ eine polynomielle Reduktion von L auf L'. Die L'-OTM für L arbeitet auf Eingaben $x \in \Sigma^*$ wiefolgt:

- Berechne $f(x)$ und schreibe $f(x)$ aufs Orakelband.
- Akzeptiere x genau dann wenn das Orakel $f(x)$ akzeptiert.

Lemma 59

Die Entscheidungsvarianten von Clique, Rucksack, TSP usw. sind auf ihre Optimierungsvarianten Turing-reduzierbar.

Turing-Reduzierbarkeit, Beispiele (2)

Beweis: zeigen *DecClique* $\leq_T$ *MaxClique* durch Konstruktion einer *OptClique*-OTM für *DecClique*) :

- Eingabe $G = (V, E)$ ungerichteter Graph, k untere Schranke für Cliquengröße.
- Schreibe G aufs Orakelband. Der Orakelaufruf liefert eine maximale Clique $V' \subseteq V$ von G.
- Akzeptiere G, k falls $|V'| \geq k$. □

In vielen Fällen gilt auch die Umkehrung

Lemma 60

Die Optimierungsvariantenvarianten von Clique, Rucksack, TSP usw. sind auf ihre Entscheidungsvarianten Turing-reduzierbar.

Turing-Reduzierbarkeit, Beispiele (3)

Beweis: Zeigen $\mathit{MaxClique} \leq_T \mathit{DecClique}$ durch Konstruktion einer polynomiellen *DecClique*-OTM für *MaxClique*, die auf Eingaben $G = (V, E)$ wiefolgt arbeitet.

- Testen mittels Orakelaufrufen für $k = 1, \cdots, |V|$ ob $(G, k) \in \mathit{DecClique}$ und ermitteln so $\mathit{CliqueSize}(G) = \max\{|V'|, V' \subseteq V \text{ Clique in } G\}$.
- Entfernen solange Kanten $e \in E$, für die $\mathit{CliqueSize}(G \setminus \{e\}) = \mathit{CliqueSize}(G)$ bis eine Clique der Größe $\mathit{CliqueSize}(G)$ übrigbleibt. □

Das Phänomen der Turing-Reduzierbarkeit eines Optimierungsproblems auf seine Entscheidungsvariante heißt **Selbstreduzierbarkeit**. Gilt auch für *TSP*, *RP* und viele andere Probleme.

NP-schwere, -leichte und -äquivalente Probleme

Definition 61

Sei Π eine Berechnungsproblem

- Π heißt *NP*-schwer, falls ein *NPC*-Problem L mit $L \leq_T \Pi$ existiert.
- Π heißt *NP*-leicht, falls ein *NP*-Problem L mit $\Pi \leq_T L$ existiert.
- Π heißt *NP*-äquivalent, falls Π sowohl *NP*-leicht als auch *NP*-schwer ist.

Lemma 62

- *Aus Π NP-schwer und $\Pi \in$ PTIME folgt $P = NP$.*
- *Aus Π NP-leicht und $P = NP$ folgt $\Pi \in$ PTIME.*
- *NPC-Probleme sind NP-äquivalent.* $\square$

One-Way Funktionen

Definition 63

Sei $F : \Sigma^* \longrightarrow \Sigma^*$ eine Funktion mit $|x| = |F(x)|$ für alle $x \in \Sigma^*$. F heißt One-Way Funktion, falls $F \in PTIME$, jedoch $F^{-1} \notin PTIME$ (F^{-1} bezeichnet das Umkehrproblem zu F, für gegebenes $y \in \Sigma^*$ ein x mit $F(x) = y$ zu berechnen).

- One-Way Funktionen spielen eine große Rolle in der Kryptographie, Beispiele:
- Multiplikation von n-Bit Zahlen (Umkehrproblem Faktorisierung)
- Modulare Exponentiation $F(x) = g^x \mod p$, wobei p n-Bit Primzahl, g Erzeugendes von $\mathbb{Z}_p^*$ (Umkehrfunktion Diskreter Logarithmus).
- Auf der Unmöglichkeit, $m = p \cdot q$ (p, q n-Bit Primzahlen, $n \geq 1024$) zu faktorisieren, beruht z.B. die Sicherheit des RSA-Systems.

Eine kryptographische Konsequenz von $P = NP$

Theorem 64

Sei $F : \Sigma^* \longrightarrow \Sigma^*$ in PTIME. Dann folgt aus $P = NP$, dass auch $F^{-1} \in$ PTIME. D.h. falls $P = NP$ gibt es keine One-Way Funktionen.

Beweisidee: Sei $\Sigma = \{0, 1\}$. Wir identifizieren $\{0, 1\}^n$ mit $\{0, \cdots, 2^n - 1\}$ und nehmen OBdA an, dass $|x| = |F(x)|$ für alle $x \in \{0, 1\}^*$.

Definieren Test $\tilde{F} = \tilde{F}(y, k)$ mit $\tilde{F}(y, k) = 1$ falls existiert $x \geq k$ mit $F(x) = y$. Es gilt $\tilde{F} \in NP$.

Aus $P = NP$ folgt, dass $\tilde{F}(y, k)$ einen Polynomialzeitalgorithmus hat.

$F^{-1}(y)$ kann mittels n $\tilde{F}$-Tests ermittelt werden (binäre Suche), also gilt $F^{-1} \in PTIME$. $\square$

Erzeugung von Sprachen durch Grammatiken

Grammatiken

Definition 65

Grammatiken $G = (T, V, S, P)$ bestehen aus vier Komponenten, die wiefolgt definiert sind:

- T ist ein endliches Alphabet von Terminalsymbolen, $T \neq \emptyset$.
- V ist ein endliches Alphabet von Hilfssymbolen, $V \neq \emptyset$, $V \cap T = \emptyset$.
- $S \in V$ bezeichnet das Startsymbol.
- Ein Paar (l, r) mit $l \in (V \cup T)^+$ und $r \in (V \cup T)^*$ heißt Ableitungsregel (oder Produktion) über V, T (alternative Schreibweise $l \to r$).
- P bezeichnet eine endliche Menge von Regeln über V, T.

Beispiel: $G = (\{0, 1\}, \{S\}, S, \{S \to 0S1, S \to \epsilon\})$.

Ableitung von Worten mittels Grammatiken

Definition 66

Es sei $G = (T, V, S, P)$ eine Grammatik.

- Eine Regel $(l, r) \in P$ heißt anwendbar auf ein Wort $z \in (V \cup T)^*$, falls z das Wort l als Teilwort enthält. Die Regel (l, r) auf z anwenden heißt: Ersetze $z = ulv$ durch $z' = urv$.
- $z' \in (V \cup T)^*$ heißt direkt G-ableitbar aus $z \in (V \cup T)^*$, falls z' Ergebnis der Anwendung einer Regel aus P auf z ist, Schreibweise $z \underset{G}{\longrightarrow} z'$.
- z' heißt G-ableitbar aus z, falls es eine endliche Folge $z^1, \cdots, z^s$ von Worten gibt mit $z^1 = z$, $z^s = z'$ und $z^r \underset{G}{\longrightarrow} z^{r+1}$ für alle $r = 1, \cdots, s-1$ (Schreibweise $z \overset{*}{\underset{G}{\longrightarrow}} z'$).

Erzeugung von Sprachen durch Grammatiken

Definition 67

Es sei $G = (T, V, S, P)$ eine Grammatik. Dann bezeichnet $L(G) = \{z \in T^*; S \xrightarrow[G]{*} z\}$ die von G erzeugte Sprache.

Beispiele:

- $G_1 = (\{0,1\}, \{S\}, S, \{S \to 0S1, S \to \epsilon\})$,
$$L(G) = \{0^n 1^n; n \geq 0\}.$$
- Die Sprache der Regulären Ausdrücke über $\{0,1\}$ wird erzeugt durch $G_2 = (\{0,1,e,\cdot,+,^*,(,),\emptyset\}, \{S\}, S, P)$ mit
$$P = \{S \to 0, 1, e, \emptyset, (S) \cdot (S), (S) + (S), (S)^*\}.$$
- $G_3 = (\{0,1\}, \{S,T\}, S, P)$ mit
$$P = \{S \to 0S, 1T, T \to 0T, 1S, \epsilon\}.$$

Grammatiken vom Chomsky Typ

Definition 68

- Grammatiken ohne weitere Einschränkungen heißen Chomsky-0 Grammatiken.
- $G = (T, V, S, P)$ heißt Chomsky-1 oder kontextsensitiv, wenn alle Regeln die Gestalt $S \to \epsilon$ oder $l \to r$ mit $l \in V^+$, $r \in ((V \cup T) \setminus \{S\})^*$, $|l| \leq |r|$ haben.
- $G = (T, V, S, P)$ heißt Chomsky-2 oder kontextfrei, wenn alle Regeln die Gestalt $A \to r$ mit $A \in V$ und $r \in (V \cup T)^*$ haben.
- $G = (T, V, S, P)$ heißt Chomsky-3, wenn alle Regeln die Gestalt $A \to r$ mit $r = \epsilon$ oder $r = aB$ mit $a \in T$ und $B \in V$ haben.
- Für alle $i = 1, 2, 3, 4$ bezeichne $\mathcal{L}_i$ die Menge der Sprachen, die durch eine Chomsky-i Grammatik erzeugbar sind.

Die Chomsky Hierarchie

Noam Chomsky (*1928, Prof.em. des M.I.T.) ist ein weltweit bekannter linker Intellektueller und Linguist, Kognitionswissenschaftler und Informatiker.

Theorem 69

Es gilt $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$.

Zum Beweis: Es ergibt sich direkt aus den Definitionen, dass jede Chomsky-3 Grammatik eine Chomsky-2 Grammatik und jede Chomsky-1 Grammatik eine Chomsky-0 Grammatik ist.

Zudem lässt sich jede Chomsky-2 Grammatik in eine äquivalente Chomsky-1 Grammatik umformen (Übungsaufgabe).

Wir werden im Folgenden sehen, dass die Klassen der Chomsky Hierarchie teilweise mit uns bereits bekannten Klassen zusammenfallen, und dass alle Inklusionen in der Chomsky Hierarchie echt sind.

Wichtige Probleme mit Bezug zu Grammatiken

Definition 70

- Das **Wortproblem** bezüglich einer Grammatik $G = (T, V, S, P)$ besteht in der Entscheidung, ob ein gegebenes Wort $z \in T^*$ in $L(G)$ liegt.
- Das **Parsingproblem** bezüglich G besteht in der Entscheidung, ob ein gegebenes Wort $z \in T^*$ in $L(G)$ liegt, und, falls ja, in der Berechnung einer Ableitung, d.h. einer Folge $z^1, \cdots, z^s$ von Worten aus $(V \cup T)^*$, für die gilt

$$S \underset{G}{\longrightarrow} z^1 \underset{G}{\longrightarrow} z^2 \underset{G}{\longrightarrow} \cdots \underset{G}{\longrightarrow} z^s \underset{G}{\longrightarrow} z.$$

Anwendung: Höhere Programmiersprachen basieren auf Grammatiken, für die das Parsing Problem effizient lösbar ist. Das Parsen eines gegebenen Programms ist die Basis für das Kompilieren in ein äquivalentes Maschinencode-Programm.

Chomsky-0 Grammatiken

Wir gehen nun der Frage nach, für welche Stufen der Chomsky-Hierarchie das Wort- und Parsingproblem effizient lösbar ist und für welche Grammatiken G die Sprache $L(G)$ über genügend Ausdruckskraft zur Formulierung von Algorithmen besitzen. Wir starten mit der Klasse $\mathcal{L}_0$.

Theorem 71

Jede rekursiv aufzählbare Sprache lässt sich durch eine Grammatik erzeugen, d.h. $ENUM \subseteq \mathcal{L}_0$.

Damit ist klar, dass uneingeschränkte Grammatiken zu ausdrucksstark sind für Programmiersprachen, denn $\mathcal{L}_0$ enthält nicht berechenbare Probleme.

Beweis von Theorem 71: Es sei $L \subseteq \{0,1\}^*$ eine rekursiv aufzählbare Sprache und M eine Einband TM, die genau die Worte aus L akzeptiert.

Der Beweis von Theorem 71, Teil I

Wir setzen OBdA voraus, dass $M = (Q, q_0, \delta)$ über dem Bandalphabet $\{0, 1, \#, \$\}$ definiert ist und die folgenden Eigenschaften hat:

- M hat genau einen akzeptierenden Endzustand q^* hat. Zudem wird der Anfangszustand q_0 von M nur zum Zeitpunkt 0 angenommen.
- Die Anfangskonfiguration zu jeder Eingabe $z = (z_1, \cdots, z_{|z|}) \in \{0, 1\}^*$ ist $(q_0, z_1, \cdots, z_{|z|}, \$)$, d.h., der Kopf steht auf dem linkesten Eingabezeichen und rechts des rechtesten Eingabezeichens steht das Sonderzeichen \$.
- Die \$-Marke wird nie nach rechts überschritten.
- Die einzig auftretende akzeptierende Stoppkonfiguration entspricht dem leeren Band im Zustand q^*.

Der Beweis von Theorem 71, Teil II

Wir konstruieren eine Grammatik $G = (T, V, S, P)$ mit $L(G) = L$ und $T = \{0, 1\}$, $V = Q \cup \{S, \$, \#\}$ und $P = P_1 \cup P_2 \cup P_3$. Wir konstruieren G so, dass sich genau die Konfigurationen von M mittels G aus S ableiten lassen, die zu einer akzeptierenden Stoppkonfiguration führen. Ableitungen verfolgen Berechnungen somit schrittweise zurück, Regeln in P entsprechen umgekehrten Zustandsübergängen von M.

- Regelgruppe P_1 erzeugt akzeptierende Stoppkonfigurationen $(\#, \cdots, \#, q^*, \#, \cdots, \#)$ mittels der Regeln $S \to q^*$, $q^* \to \#q^*, q^*\#$.
- Regelgruppe P_2 realisiert die direkte Vorgängerrelation von M-Konfigurationen, indem jede Instanz von δ in eine Menge von Regeln übersetzt wird.

Der Beweis von Theorem 71, Teil III

- Regelgruppe P_2:
  - Instanzen der Form $\delta(q, a) = (a', R, q')$ werden übersetzt in die Regel $a'q' \to qa$.
  - Instanzen der Form $\delta(q, a) = (a', L, q')$ werden übersetzt in die Regelmenge $\{q'ba' \to bqa; b \in \{0, 1, \$, \#\}\}$.
  - Instanzen der Form $\delta(q, a) = (a', N, q')$ werden übersetzt in die Regel $q'a' \to qa$.

  Man beachte, dass mittels P_1 und P_2 genau die Anfangskonfigurationen aus S ableitbar sind, die zur akzeptierenden Stoppkonfiguration führen, d.h. Eingaben aus L entsprechen.
- Regelgruppe P_3 leitet aus Startkonfigurationen $(q_0, z_1, \cdots, z_{|z|}, \$)$ das Wort $z = (z_1, \cdots, z_{|z|})$ ab. Das geschieht durch die Regeln $q_o 0 \to 0 q_0$, $q_o 1 \to 1 q_0$, $q_0 \$ \to \epsilon$.

□

Die Umkehrung

Theorem 72

Jede Sprache, die durch eine Grammatik erzeugbar ist, ist rekursiv aufzählbar, d.h. $\mathcal{L}_0 = ENUM$.

Beweis: Es sei $G = (V, T, S, P)$ eine beliebige Grammatik. Wir beschreiben einen Algorithmus M, der genau die Worte aus $L(G)$ akzeptiert und auf allen anderen Worten aus T^* nicht anhält.

M berechnet auf Eingabe $z \in T^*$ für $i = 0, 1, \cdots$ alle Worte der Menge $L(G)_i \subseteq (V \cup T)^*$ bestehend aus allen Worten, die mittels i Anwendungen von Regeln aus P aus S abgeleitet werden können. M stoppt akzeptierend, sobald $z \in L(G)_i$ für ein $i \geq 1$ festgestellt wird.

Offensichtlich gilt $L(G)_0 = \{S\}$. Für alle $i \geq 1$ berechnet M die Menge $L(G)_i$ aus $L(G)_{i-1}$ indem für alle Worte $w \in L(G)_{i-1}$ jede Regel aus P in jeder möglichen Weise auf w angewendet wird. □

Chomsky-3 Grammatiken

Wir gehen nun ans andere Ende der Chomsky Hierarchie und untersuchen, ob Chomsky-3 Grammatiken zur Erzeugung Höherer Programmiersprachen geeignet sind.

Theorem 73

Eine Sprache ist genau dann durch eine Chomsky-3 Grammatik erzeugbar, wenn sie regulär, d.h., durch einen endlichen Automaten berechenbar, ist.

Beweis: Wir fixieren zunächst einen DFA $A = (Q, q_0, F, \delta)$ über einem endlichen Arbeitsalphabet T. Wir ordnen A folgende Chomsly-3 Grammatik $G = (T, Q, q_0, P)$ zu:

(1) Für alle $q \in Q$ und $t \in T$ enthält P die Regel $q \to tq'$, wobei $q' = \delta(q, t)$.

(2) Für alle $q \in F$ enthält P die Regel $q \to \epsilon$.

Der Beweis von Theorem 73

Offensichtlich ist mittels Regelgruppe (1) für alle Worte $z \in T^*$ das Wort zq mit $q = \delta^*(q_0, z)$ aus dem Startsymbol q_0 ableitbar. Genau dann wenn $q \in F$ gilt, ist auch z aus dem Startsymbol q_0 ableitbar (Regelgruppe (2)). Damit gilt $L(G) = L(A)$.

Wir ordnen nun einer beliebige Chomsky-3 Grammatik $G = (T, V, S, P)$ einen NFA $N = (V, S, F, \delta)$ zu, der folgendermaßen definiert ist:

(1) Für alle $A \in V$ und $t \in T$ gelte genau dann $B \in \delta(A, t)$ wenn P die Regel $A \to tB$ enthält.

(2) Für alle $A \in V$ gelte genau dann $A \in F$ wenn P die Regel $A \to \epsilon$ enthält.

Es gilt $z \in L(G)$ genau dann wenn $S \xrightarrow[G]{*} zB$ und P enthält $B \to \epsilon$. Das gilt genau dann wenn $B \in \delta^*(S, z)$ und $B \in F$ also $z \in L(N)$. □

Chomsky-3 Grammatiken und Programmiersprachen

Da mit Chomsky-3 Grammtiken nur reguläre Sprachen erzeugt werden können, sind diese nicht ausdrucksstark genug um als Grammatiken für Höhere Programmiersprachen zu dienen.

So sollte es mittels einer derartige Grammatik möglich sein, zu entscheiden, ob alle öffnenden Klammern in einem Programmtext auch wieder geschlossen werden. Das entspricht der Sprache $D_1 \subseteq \{(,)\}^*$, die wiefolgt definiert ist.

Für alle Klammerausdrücke $w \in \{(,)\}^*$ bezeichne $d(w)$ die Differenz zwischen der Anzahl der öffnenden und der Anzahl der schließenden Klammern in w.

Es gelte, dass $w \in D_1$ wenn $d(w) = 0$ und $d(w') \geq 0$ für jeden Präfix w' von w.

Damit enthält D_1 genau die korrekt geklammerten Ausdrücke.

Nichtregularität der Klammersprache D_1

Lemma 74

D_1 ist nicht regulär.

Beweis: Wir nehmen an, dass es einen DFA $A = (Q, q_0, F, \delta)$ mit $L(A) = D_1$ gibt.

Wir bezeichnen mit $(^n$ das Wort aus n öffnenden Klammern, $)^m$ entsprechend. Es gilt dass $(^n)^m \in D_1$ genau dann, wenn $n = m$.

Offensichlich muss es Zahlen $n \neq m$ mit $1 \leq n, m \leq |Q| + 1$ geben mit $\delta^*(q_0, (^n) = \delta^*(q_0, (^m)$. Das bedeutet jedoch, dass

$$\delta^*(q_0, (^n)^m) = \delta^*(q_0, (^m)^m),$$

was einen Widerspruch darstellt, da $(^n)^m$ verworfen und $(^m)^m$ akzeptiert werden muss. □

Kontextsensitive Grammatiken

Die Ausdruckskraft von Chomsky-1 Grammatiken wird charakterisiert durch

Theorem 75

Eine Sprache ist genau dann durch eine kontextsensitive Grammatik erzeugbar, wenn sie durch eine linear platzbeschränkte nichtdeterministische Turing Maschine berechenbar ist (ohne Beweis hier). □

Damit gilt, dass alle Sprachen in $\mathcal{L}_1$ berechenbar sind. Allerdings enthält $\mathcal{L}_1$ auch *NP*-vollständige Probleme.

Damit kann nicht davon ausgegangen werden, dass das Wortproblem von $\mathcal{L}_1$-Sprachen effizient lösbar ist.

Es verbleiben kontextfreie Grammatiken als einzig mögliche Kandidaten zum Erzeugen höherer Programmiersprachen.

Kontextfreie Grammatiken

Einführung und Beispiele

Wir analysieren die Ausdruckskraft von kontextfreien Grammatiken.

Diese ist ausreichend, um die notwendigen Konstrukte Höhere Programmiersprachen zu erzeugen.

Andererseits sind das Wortproblem und das Parsingproblem zu kontextfreien Grammatiken effizient lösbar.

Tatsächlich werden spezielle kontextfreie Grammatiken (deterministisch kontextfreie Grammatiken bzw. LR-Grammatiken) zur Erzeugung von Programmiersprachen praktisch eingesetzt.

Erinnerung: Alle Regeln in kontextfreien Grammatiken $G = (T, V, S, P)$ haben die Gestalt $A \to z$ mit $A \in V$ und $z \in (V \cup T)^*$.

Beispiel 1: Die Sprache der Palindrome in $\{0, 1\}^*$ kann durch die kontextfreie Grammatik mit den Regeln $S \to 0, 1, \epsilon, 0S0, 1S1$ erzeugt werden.

Weitere Beispiele

Beispiel 2: Die (nichtreguläre) Sprache $\{0^n 1^n; n \geq 0\}$ kann durch die kontextfreie Grammatik mit den Regeln $S \to \epsilon, 0S1$ erzeugt werden.

Beispiel 3: Für Worte $z \in \{0,1\}^*$ bezeichne $|z|_0$ bzw. $|z|_1$ die Anzahl der Nullen bzw. Einsen in z.

Lemma 76

Die Sprache $\{z \in \{0,1\}^+; |z|_0 = |z|_1\}$ *kann durch die kontextfreie Grammatik mit den Regeln*

$$S \to 0B, 1A; A \to 0, 0S, 1AA; B \to 1, 1S, 0BB$$

erzeugt werden.

Induktionsbeweis: Wir zeigen, dass (1) für alle $n \geq 1$ aus A genau die Worte $z \in \{0,1\}^{2n-1}$ mit $|z|_0 = |z|_1 + 1$ abgeleitet werden können, dass

Beweis von Lemma 76, Teil I

(2) für alle $n \geq 1$ aus B genau die Worte $z \in \{0,1\}^{2n}$ mit $|z|_1 = |z|_0 + 1$ abgeleitet werden können, und dass (3) für alle $n \geq 1$ aus S genau die Worte $z \in \{0,1\}^{2n}$ mit $|z|_0 = |z|_1$ abgeleitet werden können.

Die Aussage gilt offensichtlich für $n = 1$.

Wir fixieren ein beliebiges $n \geq 2$ und setzen voraus, dass Aussagen (1), (2), (3) für alle $n' < n$ gilt.

Es sei $z \in \{0,1\}^{2n-1}$, das aus A abgeleitet werden kann. Wir fixieren eine Ableitung von z aus A. Ist $A \to 0S$ die erste Regel entlang dieser Ableitung, so gilt nach Induktionsvoraussetzung $z = 0z'$ mit $|z'|_1 = |z'|_0$, also $|z|_0 = |z|_1 + 1$. Ist $A \to 1AA$ die erste Regel, so gilt nach Induktionsvoraussetzung $z = 1uv$ mit $|u|_0 = |u|_1 + 1$ und $|v|_0 = |v|_1 + 1$, also $|z|_0 = |z|_1 + 1$.

Mit der entsprechenden Begründung gilt, dass $|z|_1 = |z|_0 + 1$ falls $z \in \{0,1\}^{2n-1}$ aus B abgeleitet werden kann.

Beweis von Lemma 76, Teil II

Es sei $z \in \{0,1\}^{2n-1}$, das aus S abgeleitet werden kann. Wir fixieren eine Ableitung von z aus S. Ist $S \to 1A$ die erste Regel entlang dieser Ableitung, so gilt nach Induktionsvoraussetzung $z = 1z'$ mit $|z'|_0 = |z'|_1 + 1$, also $|z|_0 = |z|_1$. Ist $S \to 0B$ die erste Regel entlang dieser Ableitung, so gilt nach Induktionsvoraussetzung $z = 0z'$ mit $|z'|_1 = |z'|_0 + 1$, also $|z|_0 = |z|_1$.

Wir fixieren nun ein beliebiges $z \in \{0,1\}^{2n-1}$ mit $|z|_0 = |z|_1 + 1$ und zeigen, dass z aus A abgeleitet werden kann. Gilt $z = 0z'$, also $|z'|_0 = |z'|_1$, so benutzen wir $A \to 0S$ gefolgt von der laut Induktionsvoraussetzung existierenden Ableitung von z' aus S. Gilt $z = 1z'$, also $|z'|_0 = |z'|_1 + 2$, so schreiben wir $z' = uv$ mit $|u|_0 = |u|_1 + 1$ und $|v|_0 = |v|_1 + 1$. Wir benutzen Regel $A \to 1AA$ gefolgt von den laut Induktionsvoraussetzung existierenden Ableitungen von u aus A und von v aus A.

Beweis von Lemma 76, Teil III

Mit entsprechender Begründung kann man zeigen, dass jedes $z \in \{0,1\}^{2n-1}$ mit $|z|_1 = |z|_0 + 1$ aus B ableitbar ist.

Wir fixieren nun ein beliebiges $z \in \{0,1\}^{2n}$ mit $|z|_0 = |z|_1$ und zeigen, dass z aus S abgeleitet werden kann.

Gilt $z = 0z'$, also $|z'|_1 = |z'|_0 + 1$ so benutzen wir $S \to 0B$ gefolgt von der gerade nachgewiesenen Ableitung von z' aus B. Gilt $z = 1z'$, also $|z'|_0 = |z'|_1 + 1$ so benutzen wir $S \to 1A$ gefolgt von der gerade nachgewiesenen Ableitung von z' aus A. $\square$

Beispiel 4: Die Klammersprache D_1 kann durch die kontextfreie Grammatik mit den Regeln $S \to (), (S), SS$ erzeugt werden.

Das liegt daran, dass jedes Wort $z \in D_1$ entweder als $(^n)^n$, $n \geq 1$, oder als $(^n uv)^n$ mit $n \geq 0$ und $u, v \in D_1$ geschrieben werden kann (Übungsaufgabe).

Syntaxbäume

Ableitung von Worten $z \in T^*$ aus Hilfssymbolen $A \in V$ mittels kontextfreien Grammatiken $G = (T, V, S, P)$ können durch sogenannte Syntaxbäume veranschaulicht werden.

Es sei $A \xrightarrow[G]{} y_1 \cdots y_m \xrightarrow[G]{} \cdots \xrightarrow[G]{} z$ eine derartige Ableitung mit $y_1 \cdots y_m \in (V \cup T)^*$.

Man beobachte zunächst, dass $z = z^1 \cdots z^m$ mit $y_j \xrightarrow[G]{*} z^j$ für alle $j = 1, \cdots, m$.

Der Syntaxbaum zu $A \xrightarrow[G]{*} z$ hat die Wurzel mit Label A.

Die Nachfolger der Wurzel sind von links nach rechts für $j = 1, \cdots, m$ ein y_j-Blatt falls $y_j \in T \cup \{\epsilon\}$ bzw. die Wurzel des Syntaxbaums zur Ableitung $y_j \xrightarrow[G]{*} z^j$ falls $y_j \in V$.

Die Chomsky Normalform

Definition 77

Eine kontextfreie Grammatik $G = (T, V, S, P)$ heißt in Chomsky Normalform (kurz: CNF-Grammatik) falls alle Regeln in P die Gestalt $A \to BC$ oder $A \to a$ für $A, B, C \in V$ und $a \in T$ haben.

Achtung: Mittels einer CNF-Grammatik $G = (T, V, S, P)$ kann nicht das leere Wort erzeugt werden. Dazu muss man zu P die neuen Regeln $S^* \to \epsilon$ und $S^* \to S$ für ein neues Startsymbol $S^* \notin V$ hinzufügen.

Theorem 78

Für jede kontextfreie Grammatik G existiert eine äquivalente Grammatik der Größe $O(|G|^3)$ in Chomsky Normalform[1].

[1] Unter der Größen $|G|$ versteht man die Summe der Größen der Regeln. Die Größe einer Regel ist die Summe der Längen der beiden in ihr vorkommenden Worte

Der Beweis von Theorem 78

Wir wandeln schrittweise eine gegebene kontextfreie Grammatik $G = (T, V, S, P)$ mit $\epsilon \notin L(G)$ in äquivalente Grammatiken G_1, G_2, G_3 und G_4 um, wobei G_4 eine CNF-Grammatik ist.

1. Wir erhalten G_1 aus G, indem wir für alle Terminalsymbole $a \in T$ ein neues Hilfssymbol Y_a zu V hinzufügen, alle Vorkommen von a in rechten Seite von Regeln in G der Länge größer 1 durch Y_a ersetzen und die Regel $Y_a \to a$ hinzufügen.
2. Wir erhalten G_1 aus G_2, idem wir für alle Regeln $A \to B_1 B_2 \cdots B_m$ mit $m \geq 3$ und $A, B_1, \cdots, B_m \in V$ neue Hilfssymbole $C_1, \cdots, C_{m-2}$ zu V hinzufügen, und die Regel $A \to B_1 B_2 \cdots B_m$ durch die Regelfolge $A \to B_1 C_1, C_1 \to B_2 C_2, \cdots, C_{m-3} \to B_{m-2} C_{m-3}, C_{m-2} \to B_{m-1} B_m$ ersetzen.

Man sieht leicht ein, dass $L(G) = L(G_1) = L(G_2)$.

Die Konstruktion von G_3, I

Wir konstruieren G_3 aus G_2, indem wir die $A \to \epsilon$ Regeln in G_2 ersetzen.

Wir berechnen dafür zuerst die Menge $V' = \{A \in V; A \xrightarrow[G]{*} \epsilon\}$.

Für alle $A \in V'$ und $r \geq 1$ bezeichne $l(A)$ die Länge einer kürzesten Ableitung von ϵ aus A mittels G_2, und V'_r die Menge aller $A \in V'$ mit $l(A) = r$.

Man beobachte, dass $V'_1 = \{A \in V, A \to \epsilon \in G_2\}$.

V'_1 kann somit direkt berechnet aus G_2 werden.

Für $r > 1$ gilt, dass $A \in V'_r$ genau dann, wenn eine Regel $A \to B$ oder $A \to BC$ oder $A \to CB$ mit $B \in V'_{r-1}$ und $C \in V'_s$ für $s \leq r - 1$ existiert.

Demenstrechend kann V'_r direkt aus G_2 und $\bigcup_{s=1}^{r-1} V'_s$ berechnet werden.

Die Konstruktion von G_3, II

Wir erhalten G_3 aus G_2, indem wir alle Regeln $A \to \epsilon$ aus G_2 streichen und für alle Regeln $A \to BC$ in G_2 die Regel $A \to B$ hinzufügen, falls $C \in V'$ und die Regel $A \to C$ hinzufügen, falls $B \in V'$.

Wir müssen $L(G_2) = L(G_3)$ zeigen.

Wir fixieren zunächst ein Wort $z \in L(G_2)$ und eine Ableitung von z aus S mittels G_2.

Wir markieren im zugehörigen Syntaxbaum alle Knoten v, für die

- alle von v aus erreichbaren Blätter ϵ-Blätter sind,
- das auf den Vorgängerknoten von v aber nicht zutrifft.

Das entspricht einer Regel $A \to BC$ oder $A \to CB$ wobei B das Label von v ist und $B \in V'$ gilt. Ersetzt man für alle markierten Knoten v die Regel $A \to BC$ durch $A \to C$, so erhält man eine Ableitung von z aus S mittels G_3.

Die Konstruktion von G_3, Teil III, und von G_4, Teil I

Umgekehrt können wir in jedem Syntaxbaum bezüglich G_3 alle neuen G_3-Regeln $A \to B$ jeweils durch die entsprechenden G_2-Regeln $A \to BC$ bzw. $A \to CB$ mit $C \in V'$ sowie die zugehörige G_2-Ableitung von ϵ aus C ersetzen.

Die Konstruktion von G_4 aus G_3: Wir konstruieren G_4 indem wir die Kettenregeln $A \to B$ in G_3 ersetzen. Diese bilden einen gerichteten Graphen $\mathcal{G}$ über der Knotenmenge V.

Zunächst ersetzen wir sämtliche Kreise $A_1 \to A_2 \to \cdots \to A_s \to A_1$ in dem wir alle A_i, $i = 2, \cdots, s$, an allen Vorkommen in G_3 durch A_1 ersetzen und entstehende Regeln $A_1 \to A_1$ streichen.

Danach ist $\mathcal{G}$ azyklisch und wir können die Knoten in $\mathcal{G}$ topologisch sortieren.

Die verbliebenen Kettenregeln haben also die Gestalt $B_i \to B_j$ mit $i < j$, $1 \leq i \leq t$.

Die Konstruktion von G_4, Teil II

Man beobachte, dass B_t nicht als linke Seite einer Kettenregel vorkommt.

Entsprechend ersetzen wir für alle Regeln der Gestalt $B_t \to \alpha$ in allen Regeln $B_i \to B_t$ die rechte Seite B_t durch α.

Das selbe tun wir für $i = t - 1, \cdots, 2$ in dieser Reihenfolge für B_i, d.h. wir ersetzen für alle Regeln der Gestalt $B_i \to \alpha$ in allen Regeln $B_j \to B_i$ die rechte Seite B_i durch α.

Auf diese Weise entfernen wir alle Kettenregeln, G_4 ist also in CNF.

Anmerkung: Man kann zeigen, dass für G_1, G_2, G_3 jeweils die Größe und die Laufzeit zu ihrer Berechnung $O(|G|)$ beträgt.

Die Entdeckung eines Kreises in $\mathcal{G}$ sowie die Berechnung der topologischen Knotensortierung ist in Zeit $O(|G_3|)$ möglich, was eine Laufzeit von $O(|G_3|^2) = O(|G|^2)$ für die Berechnung von G_4 aus G_3 und damit für die Berechnung von G_4 aus G ergibt. □

Der Cocke-Younger-Kasami Algorithmus, I

Theorem 79

$\mathcal{L}_2 \subseteq P$.

Beweis: Wir beschreiben einen Polynomialzeitalgorithmus für das Wort- und das Parsingproblem für CF-Grammatiken, den **Cocke-Younger-Kasami (CYK) Algorithmus**:

Es sei $G = (V, T, S, P)$ eine CNF-Grammatik, $n \geq 1$ und $w = (w_1, \cdots, w_n) \in T^n$.

Der CYK-Algorithmus entscheidet in Zeit $O(|P| \cdot n^3))$ ob $w \in L(P)$. Falls ja wird ein Syntaxbaum für w bzgl. P ausgegeben.

Für alle $1 \leq i \leq j \leq n$ berechnen wir $V_{i,j}$, die Menge der Variablen $A \in V$ mit

$A \xrightarrow[G]{*} (w_i, \cdots, w_j)$.

Offensichtlich gilt $V_{i,i} = \{A; A \to w_i \in P\}$, diese lassen sich also in Zeit $O(n \cdot |P|)$ berechnen.

Der Cocke-Younger-Kasami Algorithmus, II

Außerdem gilt $w \in L(P)$ genau dann, wenn $S \in V_{1,n}$.

Wir fixieren nun eine beliebige Zahl L, $1 \leq L \leq n$, und setzen voraus, dass wir $V_{k,l}$ für alle k, l, $1 \leq k \leq l \leq n$, $k - l \leq L - 1$ bereits berechnet haben.

Wir fixieren beliebige i, j, $1 \leq i \leq j \leq n$, $j - i = L$, und berechnen $V_{i,j}$.

Offensichtlich ist $A \in V_{i,j}$ genau dann, wenn ein k, $i \leq k < j$, und eine P-Regel $A \to BC$ existiert, so dass $B \in V_{i,k}$ und $C \in V_{k+1,j}$.

Das läßt sich in Zeit $O(L \cdot |P|)$ testen.

Die Gesamtlaufzeit für den CYK-Algorithmus beträgt also $O(n^3 \cdot |P|)$. $\square$

Das Pumping Lemma für CNF-Grammatiken

Theorem 80

Für alle $L \in \mathcal{L}_2$ existiert eine Zahl n so dass für alle Worte $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ mit $|w| \geq 1$ und $|vwx| \leq n$ existiert, so dass für alle $j \geq 0$ gilt dass auch $uv^j wx^j y \in L$.

Beweis: Es sei $G = (V, T, S, P)$ eine CNF-Grammatik für L, wir setzen $n = 2^{|V|+1}$, fixieren $z \in L$ mit $|z| \geq n$ und betrachten einen Syntaxbaum T für z.

T hat mindestens $n = 2^{|V|+1}$ Blätter, also $depth(T) \geq |V| + 1$. Wir fixieren einen Weg von der Wurzel zu einem Blatt in T mit mindestens $|V| + 1$ inneren Knoten. Unter den letzten $|V| + 1$ inneren Knoten muss eine Variable $A \in V$ doppelt vorkommen.

Also gilt $S \xrightarrow[G]{*} uAy$, $A \xrightarrow[G]{*} vAx$ und $A \xrightarrow[G]{*} w$, wobei $|w| \geq 1$ und $|vwx| \leq n$. $\square$

Eine einfache Sprache außerhalb $\mathcal{L}_2$

Lemma 81

Die Sprache $L = \{a^m b^m c^m ; m \geq 1\}$ hat keine kontextfreie Grammatik.

Beweis: Wir nehmen an, dass L eine CNF-Grammatik G hat und wählen n wie in Theorem 80.

Dann hat das Wort $a^n b^n c^n$ eine Zerlegung $uvwxy$ wie in Theorem 80.

Da $|vwx| \leq n$ gilt, enthält vwx höchstens zwei verschiedene Buchstaben.

Damit kann $uv^j wx^j y$ für kein $j \geq 2$ in L liegen. □

Lemma 82

Die Sprache $L = \{a^m b^m c^m ; m \geq 1\}$ hat eine $\mathcal{L}_1$-Grammatik.

Eine $\mathcal{L}_1$-Grammatik für $L = \{a^m b^m c^m; m \geq 1\}$

Wir betrachten $G = (V, T, S, P)$ mit

- $V = \{S, S_1, A_1, A_2, B_1, B_2, C_1, C_2, D\}$
- $P = P_1 \cup P_2 \cup P_3$
- $P_1 = \{S \to abc; DB_1C_1S_1; S_1 \to A_1B_1C_1S_1, A_1B_1D\}$
- $P_2 = \{B_1A_1 \to A_1B_1; C_1A_1 \to A_1C_1; C_1B_1 \to B_1C_1\}$
- $P_3 = \{DA_1 \to aA_2; A_2A_1 \to aA_2; A_2B_1 \to aB_2; B_2B_1 \to bB_2; B_2C_1 \to bC_2; C_2C_1 \to cC_2; C_2D \to cc\}$

Lemma 83

G erzeugt $L = \{a^m b^m c^m; m \geq 1\}$.

Beweis von Lemma 83,I

Man beobachte zunächst, dass alle linken Seiten von Regeln aus P_1 disjunkt mit den rechten Seiten der Regeln aus P_1. Sollte also eine P_2-Regel direkt vor einer P_1-Regel angewendet werden, so kann man diese Regeln vertauschen.

Gleiches gilt für Regeln aus P_1 und P_3 und für Regeln aus P_2 und P_3.

Folglich kann jedes Wort in $L(G)$ mittels einer in Phasen $1, 2, 3$ unterteilten Ableitung erzeugt werden, wobei in Phase i nur P_i-Regeln ausgeführt werden, $i = 1, 2, 3$.

Nach Phase 1 wird entweder abc oder ein Wort der Gestalt $DB_1C_1(A_1B_1C_1)^{m-2}A_1B_1D$ für $m \geq 2$ abgeleitet.

Nach Phase 2 wird daraus ein Wort der Gestalt DwD, wobei w $(m-1)$-mal das Zeichen A_1, m-mal das Zeichen B_1 und $(m-1)$-mal das Zeichen C_1 enthält.

Beweis von Lemma 83,II

Auf dieses Wort DwD ist bestenfalls die Regel $DA_1 \to aA_2$ anwendbar, danach bestenfalls die Regel $A_2A_1 \to aA_2$ oder $A_2B_1 \to aB_2$ usw.

Man überlegt sich leicht, dass sich aus DwD genau dann ein Wort aus $\{a, b, c\}^*$ (nämlich das Wort $a^m b^m c^m$) ableiten lässt, wenn

$$DwD = DA_1^{m-1} B_1^m C_1^{m-1} D.$$

Genauso leicht überlegt man sich, dass sich das gewünschte Ergebniswort $DA_1^{m-1} B_1^m C_1^{m-1} D$ von Phase 2 aus dem Ergebniswort

$$DB_1C_1(A_1B_1C_1)^{m-2}A_1B_1D$$

von Phase 1 mittels Regeln aus P_2 ableiten lässt. □

Kellerautomaten und kontextfreie Sprachen

Grammatiken in Greibach Normalform

Definition 84

Eine Grammatik $G = (V, T, S, P)$ ist in Greibach Normalform (kurz, eine GNF-Grammatik), falls alle Regeln die Gestalt $A \to a\alpha$ mit $A \in V$, $a \in T$ und $\alpha \in V^*$ haben.

Offensichtlich sind GNF-Grammatiken stets kontextfrei.

Beispiel 1: In $\mathcal{L}_3$-Grammatiken haben alle Regeln die Gestalt $A \to aB$ oder $A \to a$ mit $A, B \in V$, $a \in T$, sie sind also in GNF.

Beispiel 2: Die bereits betrachtete Grammatik mit den Regeln $S \to 0B, 1A; A \to 0, 0S, 1AA; B \to 1, 1S, 0BB$ für die Sprache aller Worte mit genausoviel Einsen wie Nullen ist in GNF.

Theorem 85

Jede CNF-Grammatik G mit p Regeln lässt sich effizient in eine äquivalente GNF-Grammatik mit höchstens p^3 Regeln umwandeln (Siehe das Buch von Wegener, Kapitel 7.1). □

Kellerautomaten

Analog zu den regulären Sprachen können auch kontextfreie Sprachen sowohl durch Grammatiken als auch durch ein spezielles Automatenmodell, in diesem Fall die nichtdeterministischen **Kellerautomaten** (englisch auch *pushdown automata (PDAs)*), charakterisiert werden.

Ein Kellerautomat ist ein Automat, der, wie endliche Automaten, die Eingabe von links nach rechts von einem Eingabeband einliest und dabei seinen Zustand wechselt.

Zusätzlich kann er ein spezielles Arbeitsband, einen sogenannten *stack* oder Kellerspeicher, benutzen, der nach dem *last-in-first-out* (Lifo) Prinzip benutzt wird.

Achtung: Gäbe es keine Einschränkung bezüglich des Arbeitsbandes, so hätten Kellerautomaten die gleiche Berechnungskraft wie beliebige Turingmaschinen.

Kellerautomaten, formale Definition

Definition 86

Ein (nichtdeterministischer) Kellerautomat A über dem Eingabealphabet Σ ist ein 5-Tupel $A = (Q, \Gamma, q_0, Z_0, \delta)$, wobei

- Q ist endliche Zustandsmenge und $q_0 \in Q$ der Anfangszustand
- Γ ist das endliche Stack-Alphabet und $Z_0 \in \Gamma$ ist das Initialsymbol.
- δ bezeichnet eine endliche Menge von Instruktionen der Gestalt $I = (q, a, Z|q', Y)$ oder $I = (q, \epsilon, Z|q', Y)$, wobei $a \in \Sigma$, $q, q' \in Q$, $Z \in \Gamma$ und $Y \in \Gamma^*$.

Arbeitsweise (informal): In jedem Schritt wird ggf. ein Zeichen eingelesen und der Eingabekopf geht ein Feld nach rechts, wird das oberste Stacksymbol gestrichen ($Y = \epsilon$) oder durch ein neues Wort aus Γ^* ersetzt, und der Zustand gewechselt.

Arbeitsweise (formal), Teil I

Definition 87

- Eine Konfiguration $K = (q|w|\gamma)$ von $A = (Q, \Gamma, q_0, Z_0, \delta)$ ist gegeben durch den aktuellen Zustand q, den noch einzulesenden Teil w der Eingabe und den Stackinhalt $\gamma \in \Gamma^*$.
- Startkonfiguration zur Eingabe $x \in \Sigma^*$ ist $K_0(x) = (q_0, x, Z_0)$.
- Eine Instruktion $I = (q, a, Z|q', Y) \in \delta$ ist anwendbar auf eine Konfiguration $K = (\tilde{q}|w_1, \cdots, w_t|\gamma_1, \cdots, \gamma_s)$, falls $q = \tilde{q}$, $a = w_1$ und $Z = \gamma_1$. Das Resultat der Anwendung ist die Konfiguration $K' = (q'|w_2, \cdots, w_t|Y\gamma_2, \cdots, \gamma_s)$.
- Eine Instruktion $I = (q, \epsilon, Z|q', Y) \in \delta$ ist anwendbar auf eine Konfiguration $K = (\tilde{q}|w_1, \cdots, w_t|\gamma_1, \cdots, \gamma_s)$, falls $q = \tilde{q}$ und $Z = \gamma_1$. Das Resultat der Anwendung ist die Konfiguration $K' = (q'|w_1, \cdots, w_t|Y\gamma_2, \cdots, \gamma_s)$.

Arbeitsweise (formal), Teil II

Definition 88

- Ist eine Instruktion I auf eine Konfiguration K anwendbar und liefert K', so schreiben wir das $K \xrightarrow[A]{I} K'$.
- Eine Konfiguration $(q|w|\gamma)$ heißt Stopp-Konfiguration wenn keine Instruktion aus δ anwendbar ist. Das gilt insbesondere wenn $w = \epsilon$ oder $\gamma = \epsilon$.
- Eine Stopp-Konfiguration $(q|\epsilon|\epsilon)$ heißt **akzeptierende Stopp-Konfiguration**.
- Die Sprache $L(A) \subseteq \Sigma^*$ besteht aus allen Worten $x \in \Sigma^*$ mit $K_0(x) \xrightarrow[A]{*} (q|\epsilon|\epsilon)$ für ein $q \in Q$, d.h. für die es eine akzeptierende Berechnung von A auf $K_0(x)$ gibt.

Deterministische Kellerautomaten

Kellerautomaten $A = (Q, \Gamma, q_0, Z_0, \delta)$ sind im Allgemeinen nichtdeterministisch, das heißt für Zustände $q \in Q$, Stacksymbole $Z \in \Gamma$ und $a \in \Sigma$ kann es mehrere Instruktionen mit linker Seite $(q, a, Z|\cdots)$ oder $(q, \epsilon, Z|\cdots)$ geben.

Definition 89

Ein Kellerautomaten $A = (Q, \Gamma, q_0, Z_0, \delta)$ heißt deterministisch, falls es für alle $q \in Q$, $Z \in \Gamma$ und $a \in \Sigma$ höchstens eine Instruktion mit linker Seite $(q, a, Z|\cdots)$ oder $(q, \epsilon, Z|\cdots)$ gibt.

Beispiel 1: Wir betrachten $L = \{w\#w^R, w \in \{0,1\}^*, |w| \geq 1\}$, wobei für $n \in \mathbb{N}$ und $w = (w_1, \cdots, w_n)$ gelte $w^R = (w_n, \cdots, w_1)$.

Wir konstruieren eine Kellerautomaten $A = (Q, \Gamma, q_0, Z_0, \delta)$ für L mit $Q = \{q_0.q_1\}$, $\Gamma = \{Z_0, 0, 1\}$.

Idee: A kopiert die Eingabe in den Stack bis zum ersten $\#$ und vergleicht den Rest der Eingabe zeichenweise mit dem Stackinhalt.

Beispiele

Zu Beispiel 1: Deterministischer PDA

$$\delta = \{(q_0, 0, Z_0|0, q_0), (q_0, 1, Z_0|1, q_0),$$
$$(q_0, 0, 0|00, q_0), (q_0, 1, 0|10, q_0), (q_0, 0, 1|01, q_0), (q_0, 1, 1|11, q_0),$$
$$(q_0, \#, 0|0, q_1), (q_0, \#, 1|1, q_1),$$
$$(q_1, 0, 0|\epsilon, q_1), (q_1, 1, 1|\epsilon, q_1)\}.$$

Beispiel 2: Nichtdeterministischer PDA für $L = \{ww^R, w \in \{0,1\}^*, |w| \geq 1\}$:

$$\delta = \{(q_0, 0, Z_0|0, q_0), (q_0, 1, Z_0|1, q_0),$$
$$(q_0, 0, 0|00, q_0), (q_0, 1, 0|10, q_0), (q_0, 0, 1|01, q_0), (q_0, 1, 1|11, q_0),$$
$$(q_0, \epsilon, 0|0, q_1), (q_0, \epsilon, 1|1, q_1)$$
$$(q_1, 0, 0|\epsilon, q_1), (q_1, 1, 1|\epsilon, q_1)\}.$$

Theorem 90

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann kontextfrei, wenn Sie durch einen PDA berechnet werden kann.

Beweis: Wir beweisen hier nur die Hinrichtung. Der Beweis der Rückrichtung kann im Buch von Wegener, Kapitel 7.3, nachgelesen werden.

Wir konstruieren aus einer GNF-Grammatik $G = (V, T, S, P)$ für L einen PDA $A = (\{q_0\}, q_0, V, S, \delta)$ für L, wobei δ für alle Regeln $A \to a\alpha$ in P die Instruktion $(q_0, a, A|\alpha, q_0)$ enthält.

Man sieht leicht ein, dass für alle $w \in \Sigma^*$ und alle $\alpha \in V^*$ gilt, dass bezüglich A α genau dann als Stackinhalt nach Einlesen von w auftauchen kann, wenn $w\alpha$ mittels G aus S abgeleitet werden kann. □

Algorithmen für das SAT-Problem

Das Erfüllbarkeitsproblem (Satisfiability, SAT)

Definition 91

- Die Sprache *SAT* besteht aus allen erfüllbaren CNF-Formeln $C = C_1 \wedge \cdots \wedge C_m$.
- Die Sprache *k-SAT* ($k \geq 1$) besteht aus allen erfüllbaren CNF-Formeln $C = C_1 \wedge \cdots \wedge C_m$, deren Klauseln höchstens k Literale enthalten.

- *SAT* ist ein Problem, das in vielen Schlüsselanwendungen der Informatik gelöst werden muss (insbesondere im Bereich AI).
- *SAT* und 3-*SAT* sind *NP*-vollständig.
- Wir behandeln hier Heuristiken für das *SAT*, die besser als vollständige Suche sind (Iterierte Restriktion, Resolution), sowie effiziente Algorithmen für eingeschränkte *SAT*-Varianten (Hornformel-*SAT* und 2-*SAT*).

SAT-Algorithmen: Vollständige Suche

Eingabe: CNF-Formel $C = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ über $\{X_1, \cdots, X_n\}$.

```
1 For all  b ∈ {0,1}^n
2     do if for all j = 1 to m
3           C_j(b) = 1
2       then output C ∈ SAT, stop
4 output C ∉ SAT
```

Die Berechnung von $C_j(b)$:

```
1 For  i = 1 to n
2     do if (b_i = 0) ∧ (X̄_i ∈ C_j) or (b_i = 1) ∧ (X_i ∈ C_j)
3           then output C_j(b) = 1, stop
4 output C_j(b) = 0
```

Die Worst Case Rechenzeit vollständiger Suche ist $\Theta(n2^n)$.

Eingeschränkte CNF-Formeln (1)

Definition 92

Für alle $i = 1, \cdots, n$ bezeichne $C|_{\neg X_i}$ (bzw. $C|_{X_i}$) die AL-Formel über $\{X_1, \cdots, X_n\} \setminus \{X_i\}$, die aus C daduch entsteht, dass alle Vorkommen von $\neg X_i$ (bzw. von X_i) durch 1 und alle Vorkommen von X_i (bzw. von $\neg X_i$) durch 0 ersetzt werden.

Für alle Belegungen $\tilde{b}$ von $\{X_1, \cdots, X_n\} \setminus \{X_i\}$ bezeichne $(\tilde{b}, 0)$ (bzw. $(\tilde{b}, 1)$) die entsprechende Belegung von $\{X_1, \cdots, X_n\}$, die zusätzlich X_i mit 0 (bzw. mit 1) belegt.

Lemma 93

Für alle Belegungen $\tilde{b}$ von $\{X_1, \cdots, X_n\} \setminus \{X_i\}$ gilt $C|_{\neg X_i}(\tilde{b}) = C(\tilde{b}, 0)$ bzw. $C|_{X_i}(\tilde{b}) = C(\tilde{b}, 1)$, d.h. $C \in SAT$ genau dann, wenn $C|_{\neg X_i} \in SAT$ oder $C|_{X_i} \in SAT$. $\square$

Eingeschränkte CNF-Formeln (2)

Berechnung von $C|_{\neg X_i}$ (bzw. $C|_{X_i}$):

- Streiche alle Klauseln, die $\neg X_i$ enthalten (bzw. alle, die X_i enthalten),
- streiche alle Vorkommen von X_i (bzw. von $\neg X_i$).

Lemma 94

(1) *Existiert ein Literal L, das in keiner Klausel von C vorkommt, so gilt $C \in SAT$ genau dann, wenn $C|_{\neg L} \in SAT$.*

(2) *Kommen Literale L und $\neg L$ als 1-Klauseln in C vor, so ist $C \notin SAT$.*

(3) *Kommt ein Literal L (aber nicht $\neg L$) als 1-Klauseln in C vor, so ist $C \in SAT$ genau dann, wenn $C|_L \in SAT$.*

Beweis des Lemmas, Beispiel eingeschränkte Formeln

Beweis: Die Beweise von (2),(3) sind einfach, wir beweisen (1) und nehmen an, dass Literal L nicht in C vorkommt.

$C|_L$ entsteht durch Streichen aller Literale $\neg L$ in C.

$C|_{\neg L}$ durch Streichen aller Klauseln, in denen $\neg L$ vorkommt.

Also ist die Menge der Klauseln in $C|_{\neg L}$ eine Teilmenge der Menge der Klauseln von $C|_L$.

D.h., aus $C|_L \in SAT$ folgt $C|_{\neg L} \in SAT$. D.h. $C \in SAT$ genau dann, wenn $C|_{\neg L} \in SAT$. $\square$

Beispiel:

- $C = (X_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee X_3 \vee X_4) \wedge (X_2 \vee \bar{X}_3 \vee \bar{X}_4)$
- $C|_{X_1} = (X_3 \vee X_4) \wedge (X_2 \vee \bar{X}_3 \vee \bar{X}_4)$
- $C|_{\neg X_2} = (\bar{X}_1 \vee X_3 \vee X_4) \wedge (\bar{X}_3 \vee \bar{X}_4)$.

SAT-Algorithmen: Schema Iterierte Restriktion

SolveSAT(C)

```
1  C ← SimpleRestrict(C)
2  If SimpleTest(C) = true
3     then output SolveSimpleSAT(C)
4     else X ← ChooseSplittingVariable(C)
5          if SolveSAT(C|X=0) = 1
6             then output 1
7             else output SolveSAT(C|X=1)
```

- *SimpleRestrict*(C) testet ob, eine der drei Bedingungen in Lemma 94 vorliegen, und setzt C auf 0 (bei Vorliegen (2)) bzw. schränkt C entsprechend ein (bei Vorliegen (1) oder (3)). *SimpleRestrict*(C) wiederholt das, bis keine der drei Bedingungen vorliegt.

Erläuterungen Iterierte Restriktion, Einfache SAT-Eingaben

- *SimpleTest*(C) testet ob C **einfach** ist, d.h., eine Form hat, die einen effizienten SAT-Test erlaubt, *SolveSimpleSAT*(C) führt diesen Test dann aus.
- *ChooseSplittingVariable*(C) wählt eine passende Variable, bzgl. derer eingeschränkt wird.

Einfach sind z.B. CNF-Formeln,

- die konstant sind ($0 \notin SAT$, $1 \in SAT$),
- für die in allen Klauseln positive bzw. in allen Klauseln negative Literale vorkommen (dann ist C erfüllbar durch $(1, 1, \cdots, 1)$ bzw. $(0, 0, \cdots, 0)$, d.h. $C \in SAT$),
- die 2-CNF- oder Hornformeln sind,
- ...

Ein effizienter 2-SAT Algorithmus, Vorbetrachtungen

Definition 95

Es sei $G = (V, E)$ ein gerichteter Graph, $V_1, \cdots, V_s$ seien die starken Zusammenhangskomponenten von G. Der Faktorgraph $\tilde{G} = (\tilde{V}, \tilde{E})$ sei wiefolgt definiert:

- Es sei $\tilde{V} = \{\tilde{v_1}, \cdots, \tilde{v_s}\}$.
- Es gelte $(\tilde{v_p}, \tilde{v_q}) \in \tilde{E}$ genau dann, wenn es eine Kante $(v, w) \in E$ mit $v \in V_p$ und $w \in V_q$ gibt.

Lemma 96

- *Die starken Zusammenhangskomponenten von G können in Zeit $O(|V| + |E|)$ berechnet werden.*
- *Der Faktorgraph $\tilde{G} = (\tilde{V}, \tilde{E})$ azyklisch.*
- *Es existiert eine topologische Knotensortierung auf $\tilde{G}$, die in Zeit $O(|\tilde{V}| + |\tilde{E}|)$ berechnet werden kann.* □

Ein effizienter 2-SAT Algorithmus (1)

Sei $C = C_1 \wedge \cdots \wedge C_m$ eine 2-CNF-Formel, die OBdA nur aus 2-Klauseln besteht, $C_j = L_{j,1} \vee L_{j,2}$ für $j = 1, \cdots, m$.

- Ersetzen alle C_j durch die äquivalente Formel $(\neg L_{j,1} \rightarrow L_{j,2}) \wedge (\neg L_{j,2} \rightarrow L_{j,1})$.
- Definieren Graphen $G = (V, E)$ mit
  - $V = \{X_1, \neg X_1, \cdots, X_n, \neg X_n\}$ und
  - $E = \{(\neg L_{j,1}, L_{j,2}), (\neg L_{j,2}, L_{j,1}); j = 1, \cdots, m\}$.
- **Beobachtung:** Jede Belegung $b \in \{0,1\}^n$ markiert alle Knoten $L \in V$ mit $L(b) \in \{0,1\}$.
- **Fakt:** Belegung $b \in \{0,1\}^n$ erfüllt C genau dann, wenn es in G bezüglich der Markierung durch b keine $(1,0)$-Kanten gibt.

Ein effizienter 2-SAT Algorithmus (2)

Es bezeichne $SCC(G) = \{V_1, \cdots, V_s\}$ die Menge der starken Zusammenhangskomponenten von G.

Lemma 97

Für $b \in \{0,1\}^n$ gelte $C(b) = 1$. Dann gilt für alle starken Zusammenhangskomponenten $V_p \in SCC(G)$, dass b entweder alle Literale in V_p erfüllt, oder gar keines.

Beweis: Wir fixieren $b \in \{0,1\}^n$ mit $C(b) = 1$. Angenommen es existiert $V_p \in SCC(G)$ und $L, L' \in V_p$ sodass $L(b) = 1$ und $L'(b) = 0$. Es existiert ein gerichteter Weg von L nach L', und entlang dieses Weges muss eine $(1,0)$-Kante existieren, Widerspruch zu $C(b) = 1$. $\square$

Ein effizienter 2-SAT Algorithmus (3)

Lemma 98

Es existiert ein Index i, $1 \leq i \leq n$*, und eine starken Zusammenhangskomponenten* $V_p \in SCC(G)$*, so dass* $X_i \in V_p$ *und* $\neg X_i \in V_p$*. Dann gilt* $C \notin SAT$*.*

Beweis: folgt direkt aus Lemma 97. □

Wir zeigen nun die Umkehrung von Lemma 98.

Lemma 99

Für alle i, $1 \leq i \leq n$*, und alle starken Zusammenhangskomponenten* $V_p \in SCC(G)$ *gelte, dass aus* $X_i \in V_p$ *folgt, dass* $\neg X_i \notin V_p$*. Dann gilt* $C \in SAT$*.*

Ein effizienter 2-SAT Algorithmus (4)

Setzen ab jetzt voraus, dass $\{X_i, \neg X_i\} \not\subseteq V_p$ für alle $V_p \in SCC(G)$.

Definition 100

Wir nennen $b \in \{0,1\}^n$ semierfüllend, falls für alle starken Zusammenhangskomponenten $V_p \in SCC(G)$ gilt, dass b entweder alle Literale in V_p erfüllt, oder gar keines.

- **Wissen:** $C(b) = 1 \implies b$ semierfüllend (Lemma 97).
- **Beobachtung 1:** Für alle Kanten $(L, L') \in E$ gilt $(\neg L', \neg L) \in E$.
- **Folgerung:** Für alle $W \in SCC(G)$ existiert $\neg W \in SCC(G)$ mit $\neg W = \{\neg L, L \in W\}$.

Ein effizienter 2-SAT Algorithmus (5)

Betrachten nun den Faktorgraphen $\tilde{G} = (\tilde{V}, \tilde{E})$

- **Beobachtung 3:** Für alle $\tilde{v} \in \tilde{V}$ existiert Knoten $\neg\tilde{v} \in \tilde{V}$ gemäß der entsprechenden Relation auf den starken Zusammenhangskomponenten.
- **Beobachtung 4:** Jede semierfüllende Belegung $b \in \{0,1\}^n$ markiert die Knoten $\tilde{v} \in \tilde{V}$ mit $b(\tilde{v}) \in \{0,1\}$, je nach dem ob alle Literale der entsprechenden starke Zusammenhangskom- ponente durch b erfüllt oder nicht erfüllt werden. Es gilt zudem $b(\neg\tilde{v}) = \neg b(\tilde{v})$.
- **Beobachtung 5:** Eine semierfüllende Belegung $b \in \{0,1\}^n$ erfüllt C genau dann, wenn durch b in $\tilde{G}$ keine $(1,0)$-Kanten entstehen.

Ein effizienter 2-SAT Algorithmus (5)

Wir setzen voraus, dass $\tilde{V} = \{\tilde{v_1}, \cdots, \tilde{v_s}\}$ **topologisch sortiert** ist, d.h. $i < j$ für alle Kanten $(\tilde{v_i}, \tilde{v_j}) \in \tilde{E}$.

Wir konstruktion einer semierfüllenden Belegung $b \in \{0,1\}^n$, für die $C(b) = 1$:

1 **For** $i \leftarrow 1$ **to** s
2 **do if** $\tilde{v_i}$ **unmarkiert**
3 **then** $b(\tilde{v_i}) \leftarrow 0$, $b(\neg\tilde{v_i}) \leftarrow 1$

Lemma 101

Der mittels b markierte Graph $\tilde{G}$ hat keine $(1,0)$-Kanten, d.h. $C(b) = 1$.

Ein effizienter 2-SAT Algorithmus (6)

Beweis von Lemma 101:

Angenommen, $\tilde{E}$ enthält Kante $(\tilde{v}_i, \tilde{v}_j)$ mit $b(\tilde{v}_i) = 1$, $b(\tilde{v}_j) = 0$.

Es gilt $i < j$, also gilt $\tilde{v}_j \neq \neg\tilde{v}_i$, da ansonsten $\tilde{v}_i$, der Knoten mit der kleineren Nummer, mit 0 markiert worden wäre.

Es sei nun $\tilde{v_p} = \neg\tilde{v}_i$ und $\tilde{v_q} = \neg\tilde{v}_j$.

Laut Definition von b gilt, dass $b(\tilde{v_p}) = 0$, also $p < i$, und $b(\tilde{v_q}) = 1$, also $q > j$, also $p < q$.

Es gilt jedoch auch dass $(\tilde{v_q}, \tilde{v_p}) \in \tilde{E}$, Widerspruch. $\square$

Ein effizienter 2-SAT Algorithmus, Zusammenfassung

Eingabe: $C = C_1 \wedge \cdots \wedge C_m$ 2-CNF-Formel über $\{X_1, \cdots, X_n\}$ wie beschrieben.

1 Konstruiere $G = (V, E)$ aus C wie beschrieben.
2 Konstruiere $SCC(G) = \{V_1, \cdots, V_s\}$ von G wie beschrieben.
3 **If** $\exists i, p (1 \leq i \leq n, 1 \leq p \leq s)$ mit $\{X_i, \neg X_i\} \subseteq V_p$
4 **then output** $C \notin SAT$
5 **else output** $C \in SAT$

Laufzeit: $O(n + m)$ bei geschickter Wahl der Algorithmen und Datenstrukturen.

Beispiele

$$(x_1 \vee x_2)(x_1 \vee x_3)(x_2 \vee x_3)(\neg x_1 \vee \neg x_2)(\neg x_1 \vee \neg x_3)(\neg x_2 \vee \neg x_3)$$

Der Graph ist bipartit mit Partien $\{x_1, x_2, x_3\}$ und $\{\neg x_1, \neg x_2, \neg x_3\}$ und stark zusammenhängend.

$$(x_1 \vee \neg x_2)(x_1 \vee \neg x_3)(x_2 \vee \neg x_3)(\neg x_1 \vee x_2)(\neg x_1 \vee x_3)(\neg x_2 \vee x_3)$$

Der Graph hat zwei starke Zusammenhangskomponenten $\{x_1, x_2, x_3\}$ und $\{\neg x_1, \neg x_2, \neg x_3\}$.

Hornformeln

Alfred Horn (1918-2001), US-amerikanischer Mathematiker und Logiker, legte mit Arbeiten über Hornformeln die Grundlage für die Logikprogrammierung.

Definition 102

Eine Klausel heißt **Hornklausel**, falls sie höchstens ein unnegiertes Literal enthält. Eine CNF-Formel heißt **Hornformel**, falls sie nur aus Hornklauseln besteht

- Hornklauseln haben Gestalt $X_i \equiv 1 \to X_i$ oder, für $r \geq 1$,

$$\neg X_{i_1} \vee \cdots \vee \neg X_{i_r} \vee X_{i_{r+1}} \equiv X_{i_1} \wedge \cdots \wedge X_{i_r} \to X_{i_{r+1}}$$

$$\neg X_{i_1} \vee \cdots \vee \neg X_{i_r} \equiv X_{i_1} \wedge \cdots \wedge X_{i_r} \to 0.$$

Beobachtung: Hornformeln ohne unnegierte 1-Klauseln der Gestalt $1 \to X_i$ sind immer erfüllbar (durch $(0, 0, \cdots, 0)$).

Beispiel Hornformel

$$(\neg A \vee \neg B \vee E) \wedge C \wedge A \wedge$$
$$(\neg C \vee B) \wedge (\neg A \vee D) \wedge (\neg C \vee \neg D \vee A)$$
$$\wedge \neg E \wedge \neg G$$

Ein effizienter SAT-Algorithmus für Hornformeln

Eingabe: C Hornformel über $\{X_1, \cdots, X_n\}$:

1. $b := (0, 0, \cdots, 0) \in \{0,1\}^n$
2. **while** $\exists (1 \to X_i) \in C$
3. **do choose** $i, (1 \to X_i) \in C$
4. **if** $(X_i \to 0) \in C$ **then stop** $C \notin SAT$
5. **else** $b_i := 1$, $C := C|_{X_i=1}$
6. **stop** $C \in SAT$, $C(b) = 1$

Laufzeit $O(n|C|)$, korrekt da, falls $(1 \to X_i) \in C$, $C \in SAT$ genau dann, wenn $C|_{X_i=1} \in SAT$.

Der Resolutionsalgorithmus, Vorbetrachtungen

Der Resolutionsalgorithmus ist ein Algorithmus für *SAT*, der grundlegend für viele Algorithmen im Bereich Logische Systeme ist.

Definition 103

Es seien $\mathbf{C} = C_1 \wedge \cdots \wedge C_m$ eine CNF-Formel über $\{X_1, \cdots, X_n\}$ und C, C' weitere Klausel über $\{X_1, \cdots, X_n\}$.

C heißt **Klausel in C** (Schreibweise $C \in \mathbf{C}$), falls existiert j, $1 \leq j \leq m$, mit $C = C_j$.

C heißt **Klausel von C**, falls $C \wedge \mathbf{C} \equiv \mathbf{C}$, d.h. $C^{-1}(0) \subseteq \mathbf{C}^{-1}(0)$.

C' heißt **Verkürzung von** C **(Schreibweise** $C' \subseteq C$**)**, wenn alle Literale in C' auch in C vorkommen.

Lemma 104

Gilt $C', C \in \mathbf{C}$ *und* $C' \subseteq C$ *so kann* C *in* $\mathbf{C}$ *gestrichen werden.*

Beweis: Es gilt $C^{-1}(0) \subseteq {C'}^{-1}(0) \subseteq \mathbf{C}^{-1}(0)$. □

Idee des Resolutionsalgorithmus

Definition 105

Eine CNF-Formel **C** heißt vereinfacht, falls für alle $C', C \in \mathbf{C}$ gilt $C' \not\subseteq C$.

Simplify(**C**) bezeichnet die CNF-Formel, bei der für alle Vorkommen von $C', C \in \mathcal{C}$ gilt $C' \subseteq C$ die Klausel C gestrichen wird.

Grundidee des Resolutionsalgorithmus: Finde eine Klausel C von **C** und vereinfache $C \wedge \mathbf{C}$, d.h. streiche alle Verlängerungen von C in **C**.

Definition 106

Eine Klausel C von **C** heißt Primklausel von **C**, falls alle Klauseln $C' \subset C$ keine Klauseln von **C** sind.

PC(**C**) bezeichnet die Menge aller Primklauseln von **C**.

Lemma 107

(1) *Es gilt* $\mathbf{C} \notin SAT$ *genau dann, wenn* $PC(\mathbf{C}) = \{0\}$.

(2) *Es gilt stets*

$$\mathbf{C} \equiv \bigwedge_{C \in PC(\mathbf{C})} C.$$

Beweis: (1) ist offensichtlich, wir zeigen (2):

Es gilt $\mathbf{C}(b) = 0$ genau dann, wenn $C_j(b) = 0$ für ein j, $1 \leq j \leq m$.

Wir fixieren Primklausel $C \subseteq C_j$ von $\mathbf{C}$.

Dann gilt $C(b) = 0$ und damit $\bigwedge_{C \in PC(\mathbf{C})} C(b) = 0$.

Andererseits gilt $\bigwedge_{C \in PC(\mathbf{C})} C(b) = 0$ genau dann, wenn $C(b) = 0$ für eine Primklausel C von $\mathbf{C}$, woraus $\mathbf{C}(b) = 0$ folgt. □

Anmerkung: Der Resolutionsalgorithmus berechnet $\bigwedge_{C \in PC(\mathbf{C})} C$

Resolventen

Definition 108

Eine Klausel R heißt **Resolvent** der Klauseln K und N, falls eine Variable X existiert mit $K = X \vee K'$ und $N = \neg X \vee N'$ und $R = K' \vee N'$.

R heißt **Resolvent einer CNF-Formel C** über $X_1, \cdots, X_n$, falls R Resolvent zweier Klauseln aus **C** ist.

Lemma 109

Ist R Resolvent von **C***, so ist R Klausel von* **C***, d.h.* $\mathbf{C} \wedge R \equiv \mathbf{C}$.

Beweis: Sei $R = K' \vee N'$ Resolvent zu C, wobei $K = X_i \vee K' \in C$ und $N = \neg X_i \vee N' \in C$. Wir zeigen dass R Klausel zu $K \wedge N$, d.h. $R \wedge K \wedge N \equiv K \wedge N$: Sei $R(b) = 0$. Dann ist $N'(b) = K'(b) = 0$ und somit entweder $K(b) = 0$ oder $N(b) = 0$ je nach dem ob $b_i = 0$ oder $b_i = 1$. $\square$

Abgeschlossenheit gegen Resolventenbildung

Definition 110

Eine CNF-Formel **C** heißt **abgeschlossen gegen Resolventenbildung**, falls für jeden Resolventen K von **C** gilt, dass K oder eine Verkürzung von K bereits in **C** enthalten ist.

Theorem 111

C *ist genau dann abgeschlossen gegen Resolventenbildung, wenn* **C** *aus allen Primklauseln von* **C** *besteht.*

Beweis: Angenommen, **C** besteht aus allen Primklauseln von C. Wir zeigen die Abgeschlossenheit von **C**.

Sei R ein Resolvent von **C**. Dann ist R Klausel von **C** und es existiert also eine Primklausel C von **C** mit $C \subseteq R$.

Gemäß der Annahme gilt $C \in$ **C**.

Beweis des Abgeschlossenheitssatzes (2)

Wir nehmen nun an, dass $\mathbf{C}$ eine abgeschlossene CNF-Formel über $\{X_1, \cdots, X_n\}$ ist und zeigen

$$\mathbf{C} = \bigwedge_{C \in PC(\mathbf{C})} C \tag{2}$$

per Induktion über n. Für $n = 1$ sind 0, 1, X_1, $\neg X_1$ die einzigen abgeschlossenen Formeln, für die offensichtlich (2) erfüllt ist.

Wir fixieren nun ein beliebiges $n > 1$ und beobachten, dass

$$\mathbf{C} = (X_n \vee \mathbf{C}^0) \wedge (\neg X_n \vee \mathbf{C}^1) \wedge \mathbf{C}^2.$$

Man bemerke, dass $\mathbf{C}^0 \wedge \mathbf{C}^2$ und $\mathbf{C}^1 \wedge \mathbf{C}^2$ abgeschlossene CNF-Formeln für $\mathbf{C}|_{X_n=0}$ und $\mathbf{C}|_{X_n=1}$ über $X_1, \cdots, X_{n-1}$ sind.

Damit gilt laut Induktionsvorausetzung für $\beta = 0, 1$, dass

$$\mathbf{C}^\beta \wedge \mathbf{C}^2 = \bigwedge_{C \in PC(\mathbf{C}|_{X_n=\beta})} C. \tag{3}$$

Beweis des Abgeschlossenheitssatzes (3)

Die Mengen $PC(\mathbf{C})$, $PC(\mathbf{C}|_{X_n=0})$ und $PC(\mathbf{C}|_{X_n=1})$ stehen in folgender Bezeihung zueinander. Es sei K eine beliebige Klausel über $X_1, \cdots, X_{n-1}$.

(a) Es gilt $X_n \vee K \in PC(\mathbf{C})$ genau dann, wenn $K \in PC(\mathbf{C}|_{X_n=0})$ und $\neg X_n \vee K \in PC(\mathbf{C})$ genau dann, wenn $K \in PC(\mathbf{C}|_{X_n=1})$.

(b) Aus K ist Klausel von $\mathbf{C}$ folgt K ist Klausel von $\mathbf{C}|_{X_n=0}$ und von $\mathbf{C}|_{X_n=1}$.

Beweis von (a): Es ist $X_n \vee K$ Klausel von $\mathbf{C}$ genau dann, wenn für alle $b \in \{0,1\}^n$ aus $X_n \vee K(b) = 0$ folgt $\mathbf{C}(b) = 0$. Letzteres ist äquivalent dazu, dass aus $K(b) = 0$ folgt $\mathbf{C}|_{X_n=0}(b) = 0$. Also ist $X_n \vee K$ Klausel von $\mathbf{C}$ genau dann, wenn K Klausel von $\mathbf{C}|_{X_n=0}$.

Wäre $X_n \vee K \in PC(\mathbf{C})$ und $K' \subset K$ Klausel von $\mathbf{C}|_{X_n=0}$ so wäre also auch $X_n \vee K'$ Klausel von $\mathbf{C}$, Widerspruch zu $X_n \vee K \in PC(\mathbf{C})$.

Beweis Abgeschlossenheitssatz (4)

Beweis von (b): Für alle $b \in \{0,1\}^n$ gilt $K(b) = K(\tilde{b})$, wobei $\tilde{b}$ aus b durch die Negation von b_n entsteht.

Ist K Klausel von $\mathbf{C}$ so folgt aus $K(b) = K(\tilde{b}) = 0$, dass
$\mathbf{C}(b) = \mathbf{C}|_{X_n=b_n}(b) = 0 = \mathbf{C}(\tilde{b}) = \mathbf{C}|_{X_n=\neg b_n}(b)$.

Also folgt aus $K(b) = 0$ dass $\mathbf{C}|_{X_n=0}(b) = \mathbf{C}|_{X_n=1}(b) = 0$, d.h., aus K ist Klausel von $\mathbf{C}$ folgt K ist Klausel von $\mathbf{C}|_{X_n=0}$ und von $\mathbf{C}|_{X_n=1}$.

Wir beweisen nun (2) mittels (a) und (b):

Fall 1: Es sei $X_n \vee K \in PC(\mathbf{C})$. Dann ist $K \in PC(\mathbf{C}|_{X_n=0})$ und kommt deshalb laut (3) in $\mathbf{C}^0 \wedge \mathbf{C}^2$ vor.

K kommt aber nicht in $\mathbf{C}^2$ vor, ansonsten käme K in $\mathbf{C}$ vor und $X_n \vee K$ wäre keine Primklausel von $\mathbf{C}$. Also kommt K in $\mathbf{C}^0$ und damit $X_n \vee K$ in $\mathbf{C}$ vor.

Genauso zeigt man, dass aus $X_n \vee K \in PC(\mathbf{C})$ folgt, dass $X_n \vee K$ in $\mathbf{C}$ vorkommt.

Beweis Abgeschlossenheitssatz (5)

Fall 2: Es gelte $K \in PC(\mathbf{C})$.

Wir nehmen an, dass $K \notin \mathbf{C}^2$ und beweisen das Gegenteil.

$\mathbf{C}^2$ enthält keine echte Verkürzung von K, ansonsten $K \notin PC(\mathbf{C})$.

K ist jedoch Klausel von $\mathbf{C}|_{X_n=0}$ und von $\mathbf{C}|_{X_n=1}$.

Also müssen wegen (3) sowohl $\mathbf{C}^0 \wedge \mathbf{C}^2$ als auch $\mathbf{C}^1 \wedge \mathbf{C}^2$ und damit sowohl $\mathbf{C}^0$ als auch $\mathbf{C}^1$ Verkürzungen L bzw. L' von K enthalten.

Damit enthält $\mathbf{C}$ die Klauseln $X_n \vee L$ und $\neg X_n \vee L'$.

Der entsprechende Resolvent $L \vee L'$ ist eine Verkürzung von K oder gleich K.

Da $\mathbf{C}$ abgeschlossen ist, muss $\mathbf{C}^2$ den Resolventen $L \vee L'$ (oder eine Verkürzung davon), also K oder eine Verkürzung von K, enthalten, Widerspruch zur Annahme.

K kommt also doch in $\mathbf{C}^2$ und damit auch in $\mathbf{C}$ vor. $\square$

Der Resolutionsalgorithmus

Definition 112

NRes(**C**) enthalte alle Resolventen von **C**, die nicht in **C** enthalten sind und für die auch keine Verkürzung bereits in **C** enthalten ist

Eingabe: **C** CNF-Formel über $X_1, \cdots, X_n$.

1. **C** $\leftarrow$ *Simplify*(**C**)
2. **while** $NRes(C) \neq \emptyset$
3. **do C** $\leftarrow$ *Simplify*(**C** $\wedge$ *NRes*(**C**))

- Der Algorithmus bricht ab, da jede der 3^n möglichen Klauseln höchstens einmal zu **C** hinzugefügt wird.
- Der Algorithmus liefert zudem eine abgeschlossene zur Eingabe-CNF **C** äquivalente CNF-Formel, und damit $\bigwedge_{C \in PC(\mathbf{C})} C$.

Resolutionsbeweise

- **Beobachtung:** Falls $\mathbf{C} \notin SAT$, so gibt der Resolutionsalgorithmus 0 aus, d.h. der Resolutionsalgorithmus löst das SAT-Problem.
- **Folgerung:** $\mathbf{C} \notin SAT$ genau dann, wenn eine Folge von Klauseln $K_1, \cdots, K_s$ existiert, so dass
  - Für alle $j = 1, \cdots, s$ gilt: $K_j \in \mathbf{C}$ oder es existieren i, r, $1 \leq i, r < j$ mit K_j ist Resolvent von K_i und K_r.
  - $K_s = 0$.
- Eine derartige Folge heißt **Resolutionsbeweis** für $C \notin SAT$.

Beispiel

$$\mathcal{C} = K_1 \wedge K_2 \wedge K_3 \wedge K_4 \wedge K_5 =$$
$$= (A \vee B \vee C) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (B \vee \neg C) \wedge (A \vee \neg B)$$

Resolutionsbeweis für $\mathcal{C} \notin SAT$:

- $K_6 = A \vee \neg C = Res(K_4, K_5)$ bezüglich Variable B
- $K_7 = A \vee C = Res(K_1, K_5)$ bezüglich Variable B
- $K_8 = \neg A \vee \neg C = Res(K_3, K_4)$ bezüglich Variable B
- $K_9 = C = Res(K_2, K_7)$ bezüglich Variable A
- $K_{10} = \neg C = Res(K_6, K_8)$ bezüglich Variable A
- $K_{11} = 0 = Res(K_9, K_{10})$ bezüglich Variable C.