Grundlagen der Theoretischen Informatik (GTI) Zusammenfassung und Fragezettel

Michael Gregorius

15. März 2001

Vorwort

Diese Zusammenfassung wurde in Vorbereitung auf die sogenannte INF2-Prüfung geschrieben. Sie ist in Frage/Antwort-Form geschrieben, da eine Prüfung ja auch größtenteils nach diesem Schema abläuft. Als Hauptquelle diente das Buch *„Theoretische Informatik“* von *Ingo Wegener*. Es wurde versucht, die dort im Vorwort erwähnten Kapitel nochmal mit eigenen Worten und komprimiert zusammenzufassen.

Ich kann nur empfehlen, zusammen mit anderen Kommilitonen zu lernen, da man dadurch lernt, den Stoff in eigenen Worten wiederzugeben.

Inhaltsverzeichnis

1 Turingmaschinen, Churchsche These und Entscheidbarkeit

1.1 Registermaschinen und deterministische Turingmaschinen

- **Wozu dienen Rechnermodelle?**
  - ☐ Von Gegebenheiten spezieller Rechner soll **abstrahiert** werden
  - ☐ Rechnermodelle sollen Aussagen über alle schon bekannten Rechner erlauben und auch über Rechner, die wir uns bis jetzt nur vorstellen können

- **Was ist eine Registermaschine?**

Eine Registermaschine (RAM) ist ein Rechnermodell, welches sehr nahe an reale Rechner rankommt. Im Gegensatz zu realen Rechnern verfügt die RAM jedoch über einen unendlichen Speicher, den sogenannten Registern. Ein **Register** ist eine Speicherzelle, die eine beliebig große natürliche Zahl aufnehmen kann. Es gibt ein spezielles Register, den sogenannten **Akkumulator**. Die Register sind auch auf natürliche Weise durchnummeriert, so daß sie adressiert werden können. Der Akkumulator hat die Adresse 0. Im allgemeinen wird der Inhalt des i-ten Registers mit $c(i)$ bezeichnet. Desweiteren hat die RAM einen **Programmzähler** b, der auf die aktuelle Zeile des Programms zeigt. Das **Programm** ist eine Folge von Befehlen. Welche dies sind, steht in der folgenden Liste:

1. LOAD(i): $c(0) = c(i), b = b+1$
2. STORE(i): $c(i) = c(0), b = b+1$
3. ADD(i): $c(0) = c(0) + c(i), b = b+1$
4. SUB(i): $c(0) = \max\{0, c(0) - c(i)\}, b = b+1$
5. MULT(i): $c(0) = c(0) \cdot c(i)$
6. DIV(i): $c(0) = \left\lfloor \frac{c(0)}{c(i)} \right\rfloor, b = b+1$
7. GOTO j: $b = j$
8. IF $c(0)?l$ GOTO j: $b = j$, falls $c(0)?l$ wahr, sonst b=b+1
   Dabei ist $? \in \{=, <, \leq, >, \geq\}$
9. END: $b = b$

Desweiteren gibt es noch CLOAD, CADD, CSUB, CMULT und CDIV, mit denen Operationen mit Konstanten durchgeführt werden können. Desweiteren gibt es auch noch indirekte Adressierung INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT, INDDIV, wobei auf der rechten Seite der Tabelle $c(i)$ durch $c(c(i))$ ersetzt werden muß.

- **Was ist das uniforme/logarithmische Kostenmaß?**

Beim **uniformen Kostenmaß** bewertet man jede Programmzeile einer RAM mit **einer Kosteneinheit**. Dies ist jedoch nicht realistisch, wenn man mit sehr großen Zahlen operiert. Hier ist dann das **logarithmische Kostenmaß** angebrachter, welches eine Programmzeile mit der **Länge der dabei benutzten Zahlen bewertet**.

- **Was ist eine Turingmaschine?**

Eine Turingmaschine ist ein sehr primitives Rechnermodell, welches jedoch eine genauso große Rechenkraft besitzt wie eine RAM. Dabei unterscheiden sich die Rechenzeiten nur um polynomielle Faktoren. Eine Turingmaschine besteht aus folgenden „Komponenten“:

1. Q, die endliche Zustandsmenge
2. Σ, das endliche Eingabealphabet
3. $\Gamma \supseteq \Sigma$, das endliche Bandalphabet
4. $B \in \Gamma - \Sigma$, der Blankbuchstabe
5. q_0, der Anfangszustand
6. δ, die Zustandsüberführungsfunktion $\delta : (Q \times \Gamma) \to (Q \times \Gamma \times \{R, L, N\})$
7. eventuell $F \subseteq Q$, die Menge der akzeptierenden Endzustände

Die Turingmaschine (TM) verfügt ebenfalls über einen unendlichen Speicher, jedoch sieht dieser anders aus als bei der RAM. Es ist ein **beidseitig unbeschränktes Band**. Das Band hat keine Adressen, jedoch nennen wir die Position, auf die die Maschine im Startzustand zeigt 1, die rechts davon 2, usw. Zu Beginn steht auf dem Band nur die Eingabe $w = w_1 w_2 ... w_n \in \Sigma^*$, welche nur aus Zeichen des **Eingabealphabets** besteht, eingerahmt von Blanks. Die Maschine verfügt desweiteren über einen **Schreib-/Lesekopf**, der zu Beginn der Rechnung über dem ersten Buchstaben der Eingabe steht. Das „Programm“ einer Turingmaschine ist die **Zustandüberführungsfunktion** δ. Dabei bedeutet $\delta(q,a) = (q',a',d)$ z.B., daß die TM beim Lesen des Buchstabens a im Zustand q, den Buchstaben a' auf das Band schreiben, den Lesekopf in Richtung d verschieben und in den Zustand q' wechseln soll.

Für $\delta(q,a) = (q,a,N)$ hält die Maschine. Das Ergebnis steht dann unter dem Schreib-/Lesekopf und geht bis zum ersten Blank. Gibt es nur Ergebnisse vom Typ „Ja“ oder „Nein“, dann kann man das Ergebnis auch in die Zustände integrieren. Lautet die Antwort „Ja“ befindet sich die Maschine am Ende der Rechnung in einem **akzeptierenden Zustand** $q \in F$.

- **Wann heißt eine Funktion total rekusiv?**

1. Eine Funktion $f : \Sigma^* \mapsto \Sigma^*$ heißt total rekursiv (berechenbar), wenn es eine Turingmaschine gibt, die aus der Eingabe x den Funktionswert $f(x)$ berechnet.
2. Eine Funktion $f : \mathbb{N}^k \mapsto \mathbb{N}$ heißt total rekusiv, wenn es eine Turingmaschine gibt, die für Eingaben vom Typ $\text{bin}(i_1)\#\text{bin}(i_2)\#...\#\text{bin}(i_k)$ mit Ergebnis $\text{bin}(m)$ stoppt, wenn $m = f(i_1, i_2, ..., i_k)$ ist. Dabei ist $\text{bin}(j)$ die Binärdarstellung von j.

- **Wann heißt eine Sprache L total rekursiv/rekursiv aufzählbar?**

1. Eine Sprache $L \subseteq \Sigma^*$ heißt rekursiv (entscheidbar), wenn es eine Turingmaschine gibt, die auf **allen** Eingaben stoppt und die Eingabe w genau dann akzeptiert, wenn $w \in L$ ist.
2. Eine Sprache $L \subseteq \Sigma^*$ heißt rekursiv aufzählbar (semientscheidbar), wenn es eine Turingmaschine gibt, die **genau** die Eingaben w akzeptiert, die aus L sind.

Im zweiten Fall muß die Turingmaschine also nicht für alle Eingaben stoppen. Es kann vorkommen, daß sie in eine Endlosschleife gerät und niemals entscheidet, ob $w \in L$ ist. Sie darf nur nie entscheiden, daß $w \in L$ ist, wenn es nicht zur Sprache gehört.

- **Was ist eine Konfiguration?**

(1) Eine Konfiguration einer Turingmaschine ist ein String $\alpha q \beta$ mit $\alpha \in \Gamma^*$, $q \in Q$ und $\beta \in \Gamma^*$. Sie bedeutet, daß auf dem Band die Inschrift $\alpha\beta$ eingerahmt von lauter Blankbuchstaben steht, die Maschine im Zustand q ist und der Kopf auf die Zelle zeigt, die den ersten Buchstaben von β enthält.

(2) $\alpha' q' \beta'$ ist direkte Nachfolgekonfiguration von $\alpha q \beta$, wenn $\alpha' q' \beta'$ in einem Rechenschritt aus $\alpha q \beta$ entsteht, Notation $\alpha q \beta \vdash \alpha' q' \beta'$.

(3) $\alpha'' q'' \beta''$ ist Nachfolgekonfiguration von $\alpha q \beta$, wenn $\alpha'' q'' \beta''$ in endlich vielen Rechenschritten aus $\alpha q \beta$ entsteht, Notation $\alpha q \beta \overset{*}{\vdash} \alpha'' q'' \beta''$. Es gilt stets $\alpha q \beta \overset{*}{\vdash} \alpha q \beta$, da „kein Rechenschritt“ auch in der Notation „endlich viele Rechenschritte“ enthalten ist.

- **Welche Techniken gibt es zur Programmierung von Turingmaschinen?**

1. Endlicher Speicher mit schnellem Zugriff
2. Mehrspurmaschinen
3. Unterprogramme
4. Schleifen
5. Mehrbandmaschinen
6. Unbeschränkter Speicher
7. Sortieren
8. Datenstrukturen

1.2 Simulation zwischen Turingmaschinen und Registermaschinen

- **Zeige, wie eine Turingmaschine durch eine Registermaschine simuliert werden kann!**

☐ Ist **ohne Effizienzverlust** möglich

☐ Drei Register für **Zustandsnummer**, **gelesenen Buchstaben** und **Kopfposition** reserviert

☐ Die anderen Register repräsentieren jeweils eine Speicherzelle der TM (**Adressentransformation**)

☐ δ wird als ein **Programm** mit $|Q||\Gamma|$ Programmzeilen **aufgefaßt**

☐ Mit $|Q||\Gamma|$ **if-Abfragen** wird Unterprogramm gefunden, welches die Speicherabbildungsfunktion realisiert

- **Zeige, wie eine Registermaschine durch eine Turingmaschine simuliert werden kann!**

☐ **Effizienzeinbußen** sind **unvermeidbar** weil:

a) Speicherzellen der TM können nur Buchstaben aus einem **endlichen Zeichenvorrat** speichern, die einer RAM **beliebige natürliche Zahlen**

b) RAM kann **direkt** auf Speicher zugreifen, TM muß dafür **weite Wege gehen**

- ☐ **Initialisierung**
- ☐ TM hat soviele **Unterprogramme**, wie die RAM Programmzeilen hat
- ☐ **Logarithmisches Kostenmaß** muß benutzt werden, damit Simulation fair bleibt

1.3 Universelle Turingmaschinen

- **Was ist eine universelle Turingmaschine?**
  - ☐ Ist ein **fest programmierter Rechner**, der genau soviel kann wie ein **programmierbarer Rechner** $\rightarrow$ Es gibt **keinen Unterschied**
  - ☐ **Programme sind Daten**, die abgesetzt von den Daten auf denen das Programm laufen soll übergeben werden
  - ☐ Beschränkung auf $\Gamma = \{0,1,B\}$
  - ☐ Dem universellen Rechner wird eine **Gödelnummer** und die Daten gegeben
  - ☐ **Erreichter Zustand** steht auf einem **Extraband**
  - ☐ Hat dann **Zugriff auf Buchstaben und Zustand** und kann so in der Gödelnummer **das entsprechende Tupel suchen**

- **Was ist die Gödelnummer?**
  - ☐ Wird benutzt um die **Zustandsübergangsfunktion** mit $\Sigma = \{0,1\}$ darzustellen
  - ☐ Angenommen es gibt t Zustände $Q = \{q_1,...,q_t\}$ mit q_1 als **Startzustand** und q_2 als **einzigem akzeptierenden Endzustand**
  - ☐ $X_1 = 0, X_2 = 1, X_3 = B$ codiert das **Bandalphabet** und $D_1 = L, D_2 = N, D_3 = R$ die **Kopfbewegung**
  - ☐ $\delta(q_i, X_j) = (q_k, X_l, D_m) = 0^i10^j10^k10^l10^m$ mit $i,k \geq 1$ und $j,l,m \in \{1,2,3\}$
  - ☐ Sei $code(z)$ die obige Codierung der z-ten Zeile, dann sieht die **Gödelnummer** $\langle M \rangle$ wie folgt aus:
    $$111code(1)11code(2)11...11code(s)111$$
  - ☐ Universelle Turingmaschine überprüft **syntaktische Korrektheit** der Gödel-nummer
  - ☐ Gödelnummer korrekt, dann enthalten die drei Bänder:
    a) Band 1 enthält nur $w \in \Sigma^*$
    b) Band 2 enthält die Gödelnummer $\langle M \rangle$
    c) Band 3 enthält den aktuellen Zustand q_i als 0^i, zu Beginn q_1
  - ☐ Die universelle Turingmaschine kennt den Zustand in dem die simulierte Maschine ist (Band 3) und weiß, welchen Buchstaben diese liest (Kopf auf Band1). Also sucht sie den entspechenden Block 0^i10^j1 auf Band 2 und ändert den Buchstaben auf Band 1 und den Zustand auf Band 3 gemäß $0^i10^j10^k10^l10^m$. Dabei bewegt sie den Kopf auf Band 1 entsprechend 0^m.
  - ☐ Universelle TM stoppt, wenn $q_i = q_2$ oder für $0^i10^j10^i10^j100$

1.4 Die Unentscheidbarkeit des Halteproblems

- Damit spätere Beweise einfacher werden, ist es gerechtfertigt, sich ein **Problem zu konstruieren** für das man zeigt, daß es nicht rekursiv ist
- **Kanonische Ordnung** auf Wörter und Gödelnummern und **unendliche Matrix**
- **Diagonalsprache** D ist nicht rekursiv
- **Komplement** der Diagonalsprache $\overline{D}$ ist nicht rekursiv, da sonst D rekursiv wäre, jedoch ist $\overline{D}$ rekursiv aufzählbar
- Das **Halteproblem** $H = \{\langle M\rangle w | M \text{ hält auf } w\}$ ist nicht rekursiv, da man sonst $\overline{D}$ entscheiden könnte, jedoch **rekursiv aufzählbar**
- Die **universelle Sprache** $U = \{\langle M\rangle w | M \text{ akzeptiert } w\}$ ist nicht rekursiv, weil sie eine Verallgemeinerung von $\overline{D}$ ist. Jedoch ist U **rekursiv aufzählbar** und damit auch $\overline{D}$

1.5 Eigenschaften rekursiver und rekursiv aufzählbarer Sprachen

- **Rekursive Sprachen** sind gegen Durchschnitt und Vereinigung und Komplement abgeschlossen. Seien L, L_1, L_2 rekursiv und M, M_1, M_2 die entsprechenden TM:
  a) $\overline{L}$ ist **rekursiv**. M bekommt neuen Zustand q^* der von nicht akzeptierenden Zuständen erreicht wird. q^* der einzig akzeptierende Zustand, also wird $\overline{L}$ entschieden
  b) $L_1 \cup L_2$ ist **rekursiv**. M_1 und M_2 werden nacheinander simuliert. $w \in L_1 \cup L_2$, wenn eine der TM akzeptiert
  c) $L_1 \cap L_2$ ist **rekursiv**. M_1 und M_2 werden nacheinander simuliert. $w \in L_1 \cap L_2$, wenn beide TM akzeptieren. Alternativ:
  $$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$$
- **Rekursiv aufzählbare Sprachen** sind gegen Vereinigung und Durchschnitt abgeschlossen, nicht jedoch gegen Komplement. Es seien L, L_1, L_2 rekursiv aufzählbare Sprachen und M, M_1, M_2 die entsprechenden TM:
  a) $\overline{L}$ ist **nicht rekursiv**. Beispiel: $\overline{D}$ ist rekursiv aufzählbar, D ist jedoch nicht rekursiv
  b) $L_1 \cup L_2$ ist **rekursiv aufzählbar**. Paralle Simulation oder zeitbeschränk-te Schritt-für-Schritt Simulation nacheinander
  c) $L_1 \cap L_2$ ist **rekursiv aufzählbar**. Beide Maschinen M_1 und M_2 müssen in endlicher Zeit akzeptieren, also kann ich sie nacheinander simulieren
- Sind L und $\overline{L}$ rekursiv aufzählbar, ist L rekursiv. M und $\overline{M}$ parallel laufen lassen. Jenachdem, welche Maschine akzeptiert, gilt entweder $w \in L$ oder $w \notin L$

2 NP-Vollständigkeitstheorie

2.1 Die Klasse P

- **Wie ist die worst-case-Rechenzeit definiert?**

Sei M eine deterministische Turingmaschine auf dem Eingabealphabet Σ. Die worst case Rechenzeit $t_M(n)$ ist die maximale Anzahl von Rechenschritten, die M auf Eingaben aus Σ^n macht.

- **Wie ist die Klasse P definiert?**

P ist die Klasse der Probleme, für die es eine deterministische Turingmaschine M gibt, deren worst case Rechenzeit **polynomiell beschränkt** ist.

$$t_M(n) \in O(p(n)) \Leftrightarrow t_M(n) \leq k \cdot p(n)$$

- **Wann ist ein Problem nicht in P?**

  - ☐ Wenn es nicht rekursiv ist (stärkere Aussage)
  - ☐ Wenn die Länge des Ergebnisses nicht polynomiell in der Länge der Eingabe beschränkt ist. Zum Beispiel die Frage nach allen optimalen Lösungen bei Optimierungsproblemen
  - ☐ Es gibt jedoch eine Klasse von über 1000 Problemen von denen man nicht weiß, ob sie in P sind.

- **Warum genügt es meist, sich auf Entscheidungsprobleme zu beschrän-ken?**

Weil die Optimierungsvarianten oft auch in P liegen, wenn dies für die Entscheidungsvariante gilt. Bei den meisten Problemen wird die Entscheidungsvariante polynomiell oft aufgerufen und so die Optimierungsvariante gelöst.

2.2 Nichtdeterministische Turingmaschinen und die Klasse NP

- **Was ist eine nichtdeterministische Turingmaschine?**

Eine nichtdeterministische Turingmaschine NTM ist definiert wie eine DTM, nur ist die Zustandübergangsfunktion δ hier durch eine Relation auf $(Q \times \Gamma) \times (Q \times \Gamma \times \{L, N, R\})$ ersetzt.

Eine NTM arbeitet dabei wie folgt: Wenn die NTM im Zustand q den Buchstaben a liest ist für jedes (q, a, q', a', d) in der Zustandübergangsrelation der Rechenweg möglich den eine DTM für $\delta(q, a) = (q', a', d)$ machen würde. Für eine feste Eingabe w sind also **viele Rechenwege möglich**.

- **Wann akzeptiert eine Turingmaschine eine Eingabe?**

Eine NTM M akzeptiert eine Eingabe, wenn es mindestens einen akzeptierende Rechenweg gibt. Die von M akzeptierte Sprache $L = L(M)$ besteht aus allen Wörtern w, die M akzeptiert.

- **Wie ist die Rechenzeit einer nichtdeterministischen Turingmaschine definiert?**

Sei M eine NTM, die die Sprache L akzeptiert. Für ein $w \in L$ ist die Rechenzeit definiert als die Anzahl der Rechenschritte bezüglich des kürzesten akzeptierenden Rechenweges. Für $w \notin L$ beträgt die Rechenzeit definitionsgemäß 0.

- **Was ist die Klasse NP?**

Die Klasse NP (nichtdeterministisch polynomiell) ist die Klasse der Entscheidungsprobleme, für die es eine nichtdeterministische Turingmaschine M gibt, deren worst case Rechenzeit polynomiell beschränkt ist.

- **Kann man eine nichtdeterministische Turingmaschine bauen?**

Nein, man kann sie jedoch durch eine deterministische Turingmaschine **simulieren**, die nacheinander alle möglichen Rechenwege ausprobiert. Wir zeigen folgenden Satz: Für eine Sprache $L \in$ NP gibt es ein Polynom p und eine DTM M, so daß M die Sprache L in exponentieller Zeit $2^{p(n)}$ akzeptiert.

- □ Es existiert eine NTM M' für L, deren worst case Rechenzeit durch $q(n)$ beschränkt ist.
- □ M berechnet in polynomieller Zeit die Länge der Eingabe $|w|$ und berechnet $m := q(|w|)$.
- □ Es gelte $k = 3|Q||\Gamma|$, k ist also die Anzahl an Zustandsübergängen die aus einem Zustand heraus möglich sind. Denn man kann in drei Richtungen gehen, kann aus $|Q|$ neuen Zuständen wählen und kann $|\Gamma|$ verschiedene Buchstaben schreiben. Die Tripel $Q \times \Gamma \times \{L, N, R\}$ seien von 1 bis k durchnumeriert.
- □ M probiert nun alle Rechenwege der Länge m. Dies sind alles Vektoren $(i_1, ..., i_m) \in \{1, ..., k\}^m$
- □ Die Maschine schaut in der Zustandsübergangsrelation der NTM nach, ob ein Zustandsübergang gültig ist. Ist dies der Fall, wird der Übergang durchgeführt.
- □ Es gibt also k^m solcher Rechenwegvektoren und es gilt:

$$k^m = (2^{\log_2 k})^m = 2^{m \log_2 k}$$

- □ m ist polynomiell beschränkt (da es $q(|w|)$ ist) daher kann $2^{m \log_2 k}$ durch $2^{p(n)}$ abgeschätzt werden.

- **Worin besteht das Problem SAT?**

Gegeben seien zwei natürliche Zahlen n und m. Weiter seien m Klauseln über n Variablen gegeben. Eine Klausel ist die Disjunktion von einigen Literalen x_i bzw. $\overline{x_j}$ mit $i, j \in \{1, ..., n\}$. Das Problem SAT besteht nun darin, zu entscheiden, ob es eine Belegung der Variablen $x_1, ..., x_n$ gibt, so daß alle Klauseln erfüllt werden.

Beispiel:

Gegeben seien die Klauseln $C = (c_1, c_2, c_3)$ mit $c_1 = x_1 \vee \overline{x_2} \vee x_3, c_2 = \overline{x_1} \vee x_2 \vee \overline{x_4}, c_3 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$. Die Frage besteht nun darin zu entscheiden, ob es eine Belegung der $n = 4$ Variablen gibt, so daß alle $m = 3$ Klauseln wahr werden. Oder anders, gibt es eine Belegung, so daß gilt:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) = 1$$

Eine erfüllende Belegung ist zum Beispiel $(1, 1, 0, 0)$, wie man leicht selbst ausprobieren kann. Jedoch ist es nicht immer so einfach eine Lösung zu finden. Wie sich herausstellen wird ist das Problem SAT NP-Vollständig.

- **Zeige das SAT NP-vollständig ist!**

Wir wollen zeigen, daß für alle $L \in$ NP gilt:

$$\forall L \in \text{NP} : L \leq_p SAT$$

Jedoch kennen wir garnicht alle Probleme in NP. Wir wissen nur, daß für eine Sprache in NP gilt, daß sie von einer NTM in polynomieller Zeit entschieden wird. Und bleibt also nichts anderes übrig, als die Arbeitsweise einer NTM in das SAT zu codieren.

Dazu werden wir jedoch vorher die Arbeitsweise der NTM stärker strukturieren, um den Nichtdeterminismus unter Kontrolle zu bringen. Dazu benutzen wir eine RV-NTM (Rate-Verifikations-NTM), die zuerst den Nichtdeterminismus nutzt und danach deterministisch arbeitet.

Zuerst stellen wir sicher, daß unsere Maschine ein Trennsymbol in die Zelle 0 schreibt und die Zellen nach der Ratephase nicht mehr besucht, es sei denn, es werden Zufallsbits benötigt.

Zu Beginn befindet sich die RV-NTM in der Ratephase. Sie schreibt links vom Trennzeichen einige Zufallsbits und beendet dann die Ratephase, indem sie wieder an den ersten Buchstaben der Eingabe zurückwandert.

In der Verifikationsphase arbeitet die RV-NTM deterministisch. Wann immer eine normale NTM die Wahl zwischen mehr als einem Rechenweg hat $(m \geq 2)$ wandert die RV-NTM nach links über das Trennzeichen, interpretiert die ersten $\log_2 m$ Zufallsbits, die sie findet als Binärzahl i und führt dann den i-ten Rechenweg aus. Die Zufallsbits werden nach Gebrauch gelöscht.
Um zu zeigen, daß SAT NP-vollständig ist, müssen wir erst folgendes zeigen:

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

Wir raten eine zufällige Belegung der Variablen und können in polynomieller Zeit entscheiden, ob alle Elemente der Klauselmenge zu 1 evaluiert werden.
Jetzt kommt der schwierigere Teil:

$$\forall L \in \text{NP} : L \leq_p \text{SAT}$$

Wie gesagt wissen wir nur, daß für $L \in$ NP eine RV-NTM $M = (Q, \Sigma, q_0.\Gamma, \delta, F)$ existiert die eine Eingabe w in polynomieller Zeit $p(|w|)$ akzeptiert. Die Rechenzeit ist also polynomiell abhängig von der Eingabelänge.

Wir sorgen jetzt erst einmal dafür, daß die RV-NTM auch in akzeptierenden Zuständen weiter arbeitet, dabei jedoch den akzeptierenden Zustand nicht verläßt. Dann gilt nämlich, daß es für ein $w \in L$ eine Folge von Konfigurationen $K_0, ..., K_{p(|w|)}$ gibt, so daß $K_{p(|w|)}$ akzeptierend ist. Für $w \notin L$ gibt es eine solche Folge nicht. Da die Rechenzeit durch $p(|w|)$ beschränkt ist und eine Turingmaschine in einem Rechenschritt nur eine Zelle weiterwandern kann, ist auch der Platz zur Berechnung durch $O(p(|w|))$ beschränkt.

Jetzt wenden wir uns langsam aber sicher der Struktur des SAT zu. Wie drücken wir die Konfigurationen der Turingmaschine durch Boolesche Variablen aus? Eine Konfiguration besteht zu einem Zeitpunkt aus Zustand, Kopfposition und Bandinschrift.

1. Den Zustand beschreiben wir durch folgende Variable:

$$Q(i,k) \text{ mit} \begin{cases} 0 \leq i \leq p(|w|) \\ 0 \leq k \leq |Q| - 1 \end{cases}$$

   Es gilt $Q(i,k) = 1$, wenn sich die NTM zum Zeitpunkt i im Zustand q_k befindet, mit $Q = \{q_0, ..., q_{|Q|-1}\}$. Wieviele solcher Variablen können wir bilden? Es sind:

$$(p(|w|) + 1) \cdot |Q|$$

   Ihre Zahl ist also polynomiell durch $p(|w|)$ beschränkt.

2. Die Kopfposition beschreiben wir durch folgende Variable:

$$H(i,j) \text{ mit} \begin{cases} 0 \leq i \leq p(|w|) \\ -p(|w|) \leq j \leq p(|w|) \end{cases}$$

Es gilt $H(i,j) = 1$, wenn sich der Kopf der NTM zum Zeitpunkt i an Position j befindet. Wieviele solcher Variablen können wir bilden?

$$(p(|w|)+1) \cdot (2 \cdot p(|w|)+1)$$

Auch ihre Zahl ist also polynomiell durch $p(|w|)$ beschränkt.

3. Die Bandinschrift beschreiben wir durch folgende Variable:

$$S(i,j,k) \text{ mit } \begin{cases} 0 \leq i \leq p(|w|) \\ -p(|w|) \leq j \leq p(|w|) \\ 1 \leq k \leq |\Gamma| \end{cases}$$

Es gilt $S(i,j,k) = 1$, wenn sich zum Zeitpunkt i an Position j der Buchstabe a_k mit $\Gamma = \{a_1, ..., a_{|\Gamma|}\}$ befindet. Auch hier gibt es polynomiell viele Variablen:

$$(p(|w|)+1) \cdot (2 \cdot p(|w|)+1) \cdot |\Gamma|$$

Aus diesen Variablen müssen wir noch die Klauselmenge konstruieren. Zuerst werden jedoch die Bedingungen aufgezählt und dann wird gezeigt, wie diese Bedingungen durch Klauseln ausgedrückt werden können.

1. Die Maschine befindet sich zu einem Zeitpunkt in genau einem Zustand.
2. Der Kopf der Maschine befindet sich zu einem Zeitpunkt auf genau einer Position.
3. Jede Position enthält zu jedem Zeitpunkt genau einen Buchstaben.
4. Zum Zeitpunkt 0 ist die Maschine im Anfangszustand q_k nach der Ratephase und der Kopf steht auf Position 1, am Anfang der Eingabe. Für die Bandinschrift muß folgendes gelten:
   a) Links von der Position 0 stehen die Zufallsbits oder Blanks. Links von einem Blank stehen wiederum Blanks. Dadurch wird dafür gesorgt, daß die Zufallsbits einen Block bilden.
   b) An den Positionen 1 bis $|w|$ muß die Eingabe stehen. An dieser Stelle beeinflußt die Eingabe die Klauselmenge.
   c) Rechts von der Eingabe stehen nur Blanks.
5. Die letzte Konfiguration muß akzeptierend sein.
6. Es muß sichergestellt werden, daß die Rechnung korrekt verläuft, daß also die $(i+1)$-te Konfiguration die Nachfolgekonfiguration der i-ten Konfiguration ist.

Wie sehen jetzt die Klauseln aus. Für die Punkte (1)-(3) können sie nach dem selben Schema aufgebaut werden. Seien $y_1, ..., y_m$ Variablen (z.B. die Variablen zur Codierung der Zustände). Dann sieht die Klauselmenge wie folgt aus:

$$(y_1 \vee ... \vee y_m) \wedge \bigwedge_{i \neq j} (\overline{x_i} \vee \overline{x_j})$$

Beispiel: Angenommen es gäbe nur drei Variablen zur Codierung der Zustände. Dann würde die Klausel, die für die erste Bedingung (1) sorgt, wie folgt aussehen:

$$(Q_0 \vee Q_1 \vee Q_2) \wedge (\overline{Q_0} \vee \overline{Q_1}) \wedge (\overline{Q_0} \vee \overline{Q_2}) \wedge (\overline{Q_1} \vee \overline{Q_2})$$

$(Q_0 \vee Q_1 \vee Q_2)$ wird wahr, sobald die Maschine in einem Zustand ist und die Paare der Form $(\overline{Q_0} \vee \overline{Q_1})$ werden nur wahr, wenn die Maschine in genau einem Zustand ist.
Für die Bedingung (4) gibt es folgende Klauseln:

1. $Q(0,k) = 1$, wenn q_k der Startzustand ist:

   Die Maschine befindet sich **nach der Ratephase im Startzustand**.

2. $H(0,1) = 1$:

   Der **Kopf** der Maschine befindet sich **nach der Ratephase auf dem ersten Zeichen der Eingabe**.

3. $S(0,0,t) = 1$, wenn a_t das Trennsymbol ist:

   An Position 0 steht zu Beginn das Trennzeichen.

4. Für $j < 0$ gilt:

   $$S(0,j,k_0) \vee S(0,j,k_1) \vee S(0,j,k_2) = 1 \text{ mit } k_0 = 0, k_1 = 1, k_2 = B$$

   Links vom Trennzeichen stehen also Nullen, Einsen oder Blanks.

5. Desweiteren gilt $\overline{S(0,j,k_2)} \vee (S(0,j-1,k_2))$:

   Links von Blanks stehen wieder Blanks.

6. Für $1 \leq j \leq |w|$ gilt:

   $$S(0,j,k_0) = \overline{w_j} \text{ und } S(0,j,k_1) = w_j$$

   Das heißt, an diesen Stellen muß die Eingabe stehen.

7. Für $j > |w| : S(0,j,k_2) = 1$.

   Rechts von der Eingabe stehen nur Blanks.

Für (5) reicht eine Klausel der Form $Q(p(|w|),k^*)$, wenn k^* der Index des akzeptierenden Zustandes ist.

Für (6) zählen wir erst einmal die Bedingungen auf, die gelten müssen, damit die Rechnung korrekt, also gemäß der Zustandsübergangsfunktion geschieht, auf:

1. Nicht gelesene Speicherzellen werden nicht verändert.
2. Die gelesene Speicherzelle wird korrekt verändert.
3. Der Zustand wird gemäß der Zustandsübergangsfunktion geändert.
4. Die Kopfposition wird gemäß der Zustandübergangsfunktion geändert.

Im folgenden werden die Klauseln aufgelistet, die die oben genannten Forderungen repräsentieren. Es gelte für die Punkte (2)-(4) im folgenden:

$$\forall \begin{cases} 0 \leq i \leq p(|w|) \\ -p(|w|) \leq j \leq p(|w|) \\ 1 \leq k \leq |Q| \\ 1 \leq l \leq |\Gamma| \end{cases}$$

1. $$\overline{S(i,j,k)} \vee H(i,j) \vee S(i+1,j,k) \text{ mit } \forall \begin{cases} 0 \leq i \leq p(|w|) \\ -p(|w|) \leq j \leq p(|w|) \\ 1 \leq k \leq |\Gamma| \end{cases}$$

   Wenn also zum Zeitpunkt i an Position j der Buchstabe a_k steht und der Kopf nicht auf diese Position zeigt, dann muß auch zum Zeitpunkt $i+1$ der Buchstabe a_k dort stehen.

2.
$$\overline{H(i,j)} \vee \overline{Q(i,k)} \vee \overline{S(i,j,l)} \vee S(i+1,j,b(k,l))$$

$b(k,l)$ ist eine Abbildung, die den Index des Buchstaben in Abhängigkeit vom Zustand auf einen neuen Buchstabenindex abbildet. Es gilt also: $\delta(q_k,a_l) = (\cdot,a_{b(k,l)},\cdot)$. Wenn also zum Zeitpunkt i der Kopf an Position j im Zustand k ist und den Buchstaben a_k liest, so steht zum Zeitpunkt $i+1$ an der selben Position der Buchstaben $a_{b(k,l)}$.

3.
$$\overline{H(i,j)} \vee \overline{Q(i,k)} \vee \overline{S(i,j,l)} \vee Q(i+1,c(k,l))$$

$c(k,l)$ ist hierbei eine Funktion die dafür sorgt, daß die Indizes der Zustän-de in Abhängigkeit vom gelesenen Buchstaben korrekt abgebildet werden.

4.
$$\overline{H(i,j)} \vee \overline{Q(i,k)} \vee \overline{S(i,j,l)} \vee H(i+1,j+d(k,l))$$

Es gilt für $d(k,l) : k \times l \to \{-1,0,1\}$. Dabei entspricht -1 einer Linksbewegung, 0 keiner Kopfbewegung und 1 einer Rechtsbewegung.

Wir haben die Variablenmenge mit $O(p(|w|)^2)$ Variablen beschrieben und die Klauselmenge mit $O(p(|w|)^3)$ Klauseln. Variablen- und Klauselmenge sind also in polynomieller Zeit konstruierbar.

Wenn wir also ein Wort $w \in L$ haben wird es in Zeit $O(p(|w|))$ akzeptiert. Eine korrekte Rechnung erfüllt jedoch auch die Klauselmenge. Links vom Trennzeichen muß in diesem Fall ein Ratestring einer akzeptierenden Berechnung eingesetzt werden.

Wenn andererseits die Klauselmenge erfüllbar ist, handelt es sich um eine korrekte Rechnung, die w akzeptiert.

- **Welche Arten der polynomiellen Reduktion gibt es?**

Restriktion, lokale Ersetzung und Transformation mit verbundenen Komponenten.

- **Gebe Beispiele für Restriktionen an!**

Das Prinzip der Restriktion ist, daß ein **verallgemeinertes Problem** mindestens so schwer ist, wie der in ihm enthaltene **Spezialfall**. Beispiele sind:

- ☐ SAT $\leq_p$ SAT*
- ☐ KP* $\leq_p$ KP
- ☐ PAR $\leq_p$ BPP
- ☐ HC $\leq_p$ TSP

- **Gebe Beispiele für lokale Ersetzung an!**

- ☐ SAT $\leq_p$ 3-SAT
- ☐ DHC $\leq_p$ HC

- **Gebe Beispiele für Transformation mit verbundenen Komponenten an!**

- ☐ 3-SAT $\leq_p$ CLIQUE
- ☐ 3-SAT $\leq_p$ DHC

- **Zeige: 3-SAT $\leq_p$ KP*!**

Wir haben also eine Eingabe für 3-SAT und müssen dafür eine Eingabe für KP* bauen, die genau dann eine Lösung hat, wenn 3-SAT eine Lösung hat.

Sei $C = (c_1,...,c_m)$ mit $c_i = z_{i1} \vee z_{i2} \vee z_{i3}$ und $z_{ij} \in \{x_1, \overline{x_1}, ..., x_n, \overline{x_n}\}$ eine Eingabe für 3-SAT, also m Klauseln, die höchstens 3 verschiedene Literale enthalten. Dabei gibt es insgesamt $2n$ Literale, bzw. n Variablen, die entweder negiert sind oder nicht.

Für ein Objekt a_i in KP* gilt, daß das Gewicht g_i gleich dem Nutzen a_i ist ($a_i = g_i$) und daß der zu erreichende Nutzen A gleich der Gewichtsschranke G ist ($A = G$). Mit anderen Worten: KP* ist die Frage, ob der Rucksack pickepackevoll gepackt werden kann.

Wir setzen nun in Abhängigkeit von der Anzahl der Klauseln und Literalen den zu erreichenden **Nutzen** A für KP* fest:

$$A = \underbrace{4...4}_{m}\underbrace{1...1}_{n}$$

A ist also eine Dezimalzahl, die aus m Vieren gefolgt von n Einsen besteht. Gleich wird auch klar werden, warum wir gerade diese Zahl nehmen.

Nun basteln wir uns unsere **Objekte** mit ihren dazugehörigen Nutzen für KP*, ebenfalls in Abhängigkeit von den Literalen und Zahlen. Wir bauen uns $2n + 2m$ Objekte:

$$\begin{array}{ll} a_i, b_i & 1 \leq i \leq n \\ c_j, d_j & 1 \leq j \leq m \end{array}$$

Es gibt also jeweils soviele a_i- und b_i-Objekte, wie es Literale gibt und jeweils soviele c_j- und d_j-Objekte, wie es Klauseln gibt. Die c_j- und d_j-Objekte sind dabei nur Auffüllobjekte, sie sorgen am Ende erst dafür, daß wir den Nutzen A überhaupt erreichen können.

Wie sehen jetzt die einzelnen Objekte aus? Die a_i- und b_i-Objekte haben $m + n$ Dezimalstellen, die den Nutzen angeben. An der ersten m Stellen von a_i steht dabei, wie oft die Variable x_i in den m Klauseln vorkommt. Kommt also z.B. x_3 in der ersten Klausel zweimal vor, so steht eine 2 am Anfang von a_3. Bei den letzten n Stellen, steht bei a_i nur an Position $m + i$ eine 1. Für b_i läuft das ganze Analog, nur bezieht sich hier alles auf $\overline{x_i}$.

Die Objekte c_j und d_j sind wie schon gesagt Füllobjekte. Dabei besteht c_j fast nur aus Nullen, denn nur an Position j hat c_i eine 1. Bei d_j veläuft alles analog, nur hat d_j an der j-ten Position eine 2.

Über die a_i- und b_i-Objekte entscheiden wir uns, ob $x_i = 1$ oder $\overline{x_j} = 1$ sein soll. Damit wir uns wirklich entscheiden müssen, steht an den letzten Positionen bei beiden Objekten an der selben Stelle eine 1. Würden wir beiden nehmen, könnten wir den Nutzen A nicht mehr erreichen, da bei A ja an der entsprechenden Position auch nur eine 1 steht. Im Falle, daß wir beiden nehmen würden, hätten wir an der Position jedoch schon eine 2.

Im Falle der Erfüllbarkeit von 3-SAT muß in jeder Klausel mindestens ein Literal den Wert 1 ergeben, also muß in der Summe der a_i und b_i-Objekte an den ersten m Stellen mindestens eine 1 stehen. Dann könnten wir jedoch auch über die Füllobjekte die Stellen jeweils zu 4 ergänzen. Dies ist nicht der Fall, wenn ein Literal nicht erfüllbar ist. In diesem Falle, stände nämlich an einer der ersten m Stellen der Summe (ohne die Füllobjekte) eine 0. Betrachten wir nochmal A, so wird klar, warum die Zahl genau so aufgebaut wurde:

$$A = \underbrace{4...4}_{m}\underbrace{1...1}_{n}$$

Die letzen n Stellen sorgen also dafür, daß entweder $x_i = 1$ oder $x_i = 0$ gilt, nicht jedoch beides. Die ersten Stellen sorgen dafür, daß der Wert 4 nur erreichbar ist, wenn jede Klausel mindestens ein Literal enthält, dessen Wert sich zu 1 evaluiert.

- **Zeige: KP* $\leq_p$ PAR!**

Wir zeigen also, daß wir KP* lösen können, wenn wir PAR lösen können. Sei $(a_1,...,a_n,A)$ eine Eingabe für KP*. Daraus bauen wir die Eingabe $(a_1,...,a_n,S-A+1,A+1)$ für PAR. Dabei ist S die Summe aller a_i, also:

$$S = \sum_{i=1}^{n} a_i$$

Wenn KP* eine Lösung hat, gibt es also verschiedene $a_i, 1 \leq i \leq n$, die sich zu A aufsummieren. Betrachten wir nochmal die Eingabe für PAR. Welchen Gesamtwert hat die Eingabe?

$$\sum_{i=1}^{n} a_i + (S-A+1) + (A+1) = S+S-A+1+A+1 = 2S+2$$

Also müssen die Summen der ausgewählten und nicht ausgewählten Objekte jeweils genau $S+1$ ergeben. Betrachten wir nochmal die letzten beiden Objekte der PAR-Eingabe, dann fällt auf, daß sie nicht in einer Partition sein können, denn es gilt:

$$S-A+1+A+1 = S+2 > S+1$$

Also müssen die beiden in verschiedenen Partitionen sein. Die Objekte, die jedoch $S-A+1$ zu $S+1$ ergänzen haben genau den Wert A und bilden somit eine Lösung für KP*.

3 Endliche Automaten

3.1 Schaltwerke und endliche Automaten

- **Wie ist ein endlicher Automat definiert?**

Ein endlicher Automat A (DFA A) ist wie folgt definiert:

$$\text{DFA } A = (Q, \Sigma, \Omega, q_0, \delta, \gamma, F)$$

Dabei ist:

1. Q, die endliche Zustandsmenge
2. Σ, das endliche Eingabealphabet
3. Ω, das endliche Ausgabealphabet
4. $q_0 \in Q$, der Anfangszustand
5. δ, die Zustandsübergangsfunktion mit $\delta : Q \times \Sigma \to Q$
6. γ, die Ausgabefunktion mit $\gamma : Q \times \Sigma \to \Omega$
7. $F \subseteq Q$, die Menge der akzeptierenden Zustände

Automaten dieser Art heißen auch **Mealy-Automaten**. Ein Automat beginnt im Startzustand und liest dann **sequentiell** die Eingabe. Je nach dem, in welchem Zustand er sich befindet und welchen Buchstaben er liest, ändert er seinen Zustand und macht eine Ausgabe. Dabei kann die Eingabe nur einmal von links nach rechts gelesen werden. Es ist daher auch klar, daß ein DFA **in linearer Zeit** bezüglich der Eingabelänge arbeitet.

Im folgenden werden jedoch nur noch **Entscheidungsprobleme** betrachtet, da deren Betrachtung, wie auch bei der NP-Vollständigkeit, reicht. Ω und γ fallen dabei weg. Also ist ein DFA A in diesem Fall gegeben durch:

$$\text{DFA } A = (Q, \Sigma, q_0, \delta, F)$$

Weiter setzen wir δ auf $Q \times \Sigma^*$ fort. Es gilt also:

$$\delta(q, w_1 ... w_n) = \delta(\delta(q, w_1 ... w_{n-1}), w_n)$$

- **Wie kann eine Konfiguration für einen DFA beschrieben werden?**

Eine Konfiguration läßt sich in der Form Ausgabe-Zustand-Resteingabe beschreiben, also durch:

$$y' q w' \text{ mit } y' \in \Omega^*, q \in Q, w' \in \Sigma^*$$

3.2 Die Minimierung endlicher Automaten

Die Güte eines DFA, der eine Sprache L entscheidet, wird über die Menge seiner Zustände gemessen. Wenn wir nun einen DFA entworfen haben, möchten wir gerne wissen, ob er optimal ist, er also die **minimal mögliche Anzahl an Zuständen** hat. Wie wir schon bei der Theorie der NP-Vollständigkeit gesehen haben, gilt für viele Optimierungsvarianten, daß sie NP-vollständig sind. Wir könnten also Pech haben und die Optimierung eines endlichen Automaten ist NP-vollständig.

Wir haben jedoch Glück, denn wie sich herausstellen wird, ist die Minimierung endlicher Automaten sogar ziemlich **effizient durchführbar**. Man muß dafür nur zeigen, daß

der minimale Automat **dieselbe Sprache entscheidet** und daß er **tatsächlich minimal ist**.

- **Wann heißen zwei DFAs A und A' gleich?**
  Zwei DFAs A und A' heißen äquivalent, wenn sie die gleiche Sprache entscheiden, wenn also gilt:
$$L(A) = L(A')$$

- **Wie minimiert man einen endlichen Automaten?**
  Ein endlicher Automat wird in zwei Schritten minimiert:

  1. **Entfernung** überflüssiger Zustände.
  2. **Zusammenfassen** äquivalenter Zustände.

- **Wie entfernt man überflüssige Zustände?**
  Überflüssige Zustände kann man finden, indem man in der Graphendarstellung des Automaten einen DFS (**D**epth-**F**irst-**S**earch) macht. Zustände, die dabei nicht erreicht werden, sind überflüssig. Wir interpretieren den Automaten also als einen Graphen mit der Knotenmenge Q und der Kantenmenge Σ. Jeder Zustand Q kann $|\Sigma|$ viele Kanten haben, daher ist die Entfernung überflüssiger Zustände in $O(|Q| \cdot |\Sigma|)$ möglich.

- **Wann heißen zwei Zustände eines DFA äquivalent?**
  Zwei Zustände eines DFA heißen äquivalent, wenn es für das **Akzeptanzverhalten** des Automaten völlig **unerheblich** ist, **in welchem der beiden Zustände man startet**. Oder anders gesagt:
$$p \equiv q : \forall w \in \Sigma^* : \delta(p,w) \in F \Leftrightarrow \delta(q,w) \in F$$
  Mit $[p]$ wird dabei die Äquivalenzklasse der zu p äquivalenten Zustände bezeichnet. Wenn wir für einen DFA A die Äquivalenzklassen kennen, können wir also einen zu A äquivalenten Automaten A' konstruieren, den sogenannten **Äquivalenzklassenautomaten**.

- **Wie sieht ein Äquivalenzklassenautomat aus?**
  Der Äquivalenzklassenautomat A' zu einem gegebenen Automaten A sieht wie folgt aus:
$$\begin{gathered} A' = (Q', \Sigma', q_0', \delta', F') \\ Q' = \{[q] | q \in Q\} \\ \Sigma' = \Sigma \\ q_0' = [q_0] \\ \delta' = ([q], a) = [\delta(q,a)] \\ F' = \{[q] | q \in F\} \end{gathered}$$

Um zu zeigen, daß der Äquivalenzklassenautomat A' auch dieselbe Sprache wie A entscheidet, müssen wir zeigen, daß F' und δ' wohldefiniert sind. Wir betrachten das leere Wort ε bezüglich $\equiv$ und sehen, daß $p \equiv q$ nur gilt, wenn $(p,q \in F) \vee (p,q \notin F)$. In diesem Falle gibt es dann nämlich akzeptierende Zustände und nicht akzeptierende Zustände.
Jetzt gilt es noch zu zeigen:

$$p \equiv q \Rightarrow \delta(p,a) \equiv \delta(q,a) \text{ für } a \in \Sigma$$

Wenn man also in äquivalenten Zuständen einen Buchstaben a liest, landet man wieder in zwei äquivalenten Zuständen. Dies zeigen wir wie folgt:

$$\begin{aligned} p \equiv q \quad &\Rightarrow \quad \forall w \in \Sigma^* : \delta(p,w) \in F \Leftrightarrow \delta(q,w) \in F \\ &\Rightarrow \quad \forall a \in \Sigma, w \in \Sigma^* : \delta(p,aw) \in F \Leftrightarrow \delta(q,aw) \in F \\ &\Rightarrow \quad \delta(p,a) \equiv \delta(q,a) \text{ für } a \in \Sigma \end{aligned}$$

Dies bedeutet: Wenn zwei Zustände äquivalent sind, dann folgt daraus, daß sie bezüglich aller Wörter das gleiche Akzeptanzverhalten haben. Also verhalten sich die beiden Zustände auch bezüglich des Wortes aw mit $a \in \Sigma$ und $w \in \Sigma^*$ gleich. Dann ist es jedoch so, daß man auch erst $\delta(p,a)$ bzw. $\delta(q,a)$ ausführen kann und diese sich beim Lesen von w immer noch gleich verhalten, also entweder beide in einen akzeptierenden Zustand führen oder beide nicht. Daraus folgt dann, daß auch $\delta(p,a)$ und $\delta(q,a)$ gleich sind.

Jetzt muß man noch zeigen, daß A' die gleiche Sprache entscheidet wie A. Für ein $w \in \Sigma^*$ durchläuft der Automat A die Zustandsfolge $q_0, q_1, ..., q_n$ und akzeptiert w, wenn $q_n \in F$ ist. Der Automat durchläuft dementsprechend die Zustandsfolge $[q_0], [q_1], ..., [q_n]$ und akzeptiert, wenn $[q_n] \in F'$ gilt. Nach Definition ist $q_n \in F$ genau dann, wenn $[q_n] \in F'$ ist.

- **Wie berechnet man einen minimalen DFA?**

Das **Problem** bei der Berechnung der Äquivalenzklassen des DFA ist, daß in der Definition ein **Allquantor** steht, man jedoch nicht alle $w \in \Sigma^*$ ausprobieren kann, da die Menge infinit ist. Allerdings haben wir endlich viele Zustände und können daher so schlau sein, anstatt der Äquivalenz zweier Zustände deren Nichtäquivalenz zu beweisen.

Wie kann die Nichtäquivalenz zweier Zustände bewiesen werden? Man sucht sich einfach einen **Zeugen** dafür:

$$p \not\equiv q \Leftrightarrow \exists w \in \Sigma^* : (\delta(p,w) \in F \wedge \delta(q,w) \notin F) \vee (\delta(p,w) \notin F \wedge \delta(q,w) \in F)$$

Es gibt also ein Wort w, für welches man einmal in einen akzeptierenden Zustand kommt und einmal nicht.

- **Zeige, daß Zeugen für die Nichtäquivalenz zweier Zustände nicht beliebig lang werden können!**

Angenommen ein Wort $w \in \Sigma^*$ ist ein kürzester Zeuge für die Nichtäquivalenz der Zustände p und q. Es gelte $w = aw'$. Dann ist w' logischerweise ein Zeuge für die Nichtäquivalenz von $\delta(p,a)$ und $\delta(q,a)$. Wenn $\delta(p,a)$ und $\delta(q,a)$ jedoch noch einen kürzeren Zeugen $w'' \in \Sigma^*$ haben, so ist auch aw'' ein Zeuge für die Nichtäquivalenz von p und q. **Kürzere Zeugen können also als Teilstück für längere Zeugen benutzt werden.** Also kann man die Suche abbrechen, wenn man ab einer bestimmten Länge keine neuen Zeugen mehr findet.

Der folgende Algorithmus macht sich zunutze, daß man Paare mit Zeugen der Länge 0 (ε) sehr schnell findet. Paare, die also einen minimalen Zeugen der Länge 1 haben, führen nach dem Lesen dieses Zeugen, also in Zustände, deren Nichtäquivalenz mit ε bezeugt wird. Und Paare, deren minimaler Zeuge die Länge 2 hat, führen nach dem Lesen eines Buchstaben wiederum in Zustände, deren Nichtäquivalenz durch Zeugen der Länge 1 bezeugt wird, usw.

- **Wann heißt eine Äquivalenzrelation R auf Σ^* rechtsinvariant?**

Eine Äquivalenzrelation R auf Σ^* heißt rechtsinvariant, wenn gilt:

$$xRy \Rightarrow \forall z \in \Sigma^* : xzRyz$$

Der Index von R ($ind(R)$) ist die Zahl der Äquivalenzklassen bezüglich R.

- **Gebe zwei Beispiele für rechtsinvariante Äquivalenzrelationen an!**

1. Die Relation R_A, die besagt, daß zwei Wörter $x,y \in \Sigma^*$ für einen DFA A genau dann gleich sind, wenn sie vom Startzustand aus zum selben Zustand führen, also $\delta(q_0,x) = \delta(q_0,y)$. Diese Äquivalenzrelation ist auch rechtsinvariant, denn es gilt:

$$\delta(q_0,x) = \delta(q_0,y) \Rightarrow \forall z \in \Sigma^* : \delta(q_0,xz) = \delta(q_0,yz)$$

2. Die **Nerode-Relation** R_L, die besagt, daß für eine Sprache $L \subseteq \Sigma^*$ zwei Wörter $x, y \in \Sigma^*$ genau dann gleich sind, wenn für alle $z \in \Sigma^*$ gilt, daß xz und yz entweder beide zu einem akzeptierenden Zustand gehören oder beide nicht. Die Rechtsinvarianz wird gezeigt durch:

$$\begin{aligned} xR_Ly &\Rightarrow \forall w \in \Sigma^* : (xw \in L \Leftrightarrow yw \in L) \\ &\Rightarrow \forall z, w \in \Sigma^* : (xzw \in L \Leftrightarrow yzw \in L) \\ &\Rightarrow xzR_Lyz \end{aligned}$$

- **Wie lautet der Satz von Nerode?**

Der **Satz von Nerode** besagt, daß die drei folgenden Aussagen äquivalent sind:

1. $L \subseteq \Sigma^*$ wird von einem DFA akzeptiert.
2. L ist die Vereinigung von einigen Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation mit endlichem Index.
3. Die Nerode-Relation R_L hat einen endlichen Index.

- **Beweise den Satz von Nerode!**

Wir zeigen zuerst $1 \Rightarrow 2$, dann $2 \Rightarrow 3$ und zuletzt $3 \Rightarrow 1$.

$1 \Rightarrow 2$: Wir zeigen also, daß wenn L von einem DFA akzeptiert wird, L als Vereinigung von einigen Äquivalenzklassen einer Äquivalenzrelation mit endlichem Index dargestellt werden kann.

Wir betrachten die oben definierte Äquivalenzrelation R_A, bei der zwei Wörter in einer Äquivalenzklasse lagen, wenn sie vom Startzustand aus in den gleichen Zustand führen. Die Sprache L enthält alle Wörter, für die der DFA vom Startzustand aus in einen akzeptierenden Zustand gelangt. Also ist L die Vereinigung aller Äquivalenzklassen aus R_A bezüglich akzeptierender Endzustände.

$2 \Rightarrow 3$: Wir zeigen, daß wenn es eine rechtsinvariante Äquivalenzrelation R mit endlichem Index ($ind(R) < \infty$) gibt, auch R_L einen endlichen Index hat, da R_L eine Vergröberung von R darstellt und damit $ind(R_L) \leq ind(R)$ gilt. Denn alle Äquivalenzklassen von R liegen vollständig in einer Äquivalenzklasse von R_L, wie wir gleich zeigen werden. Oder anders gesagt: Die Äquivalenzklassen von R_L sind Vereinigungen von Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation R. Das heißt, daß xRy impliziert, daß xR_Ly gilt: $xRy \Rightarrow xR_Ly$:

Gelte also xRy. Da R rechtsinvariant ist gilt auch $xzRyz$. L besteht aus der Vereinigung einiger Äquivalenzklassen einer rechtsinvarianten Äquivalenzrela-tion mit endlichem Index. Also gehört eine Äquivalenzklasse entweder komplett zu L oder nicht. Daraus folgt nun für $xzRyz$: $xz \in L \Leftrightarrow yz \in L$. Dann gilt jedoch auch xR_Ly, denn es gilt ja nach Definition von R_L:

$$xR_Ly \Rightarrow \forall z \in \Sigma^* : xz \in L \Leftrightarrow yz \in L$$

$3 \Rightarrow 1$: Wir geben einen DFA für L an. Die Zustände sind dabei die verschiedenen Äquivalenzklassen von R_L. Der Startzustand des Automaten ist $q_0 = [\varepsilon]$, also die Äquivalenzklasse, die das leere Wort enthält. Aus der Defintion von R_L geht hervor, daß die Verlängerungen aller Wörter innerhalb einer Äquivalenzklasse gleich entschieden werden. Mit der Verlängerung $z = \varepsilon$ ist daher die Menge der akzeptierenden Zustände wohldefiniert. Es sind die Zustände $[w]$ mit $w \in L$. Die Zustandsübergangsfunktion definieren wir als $\delta([w], a) = [wa]$. Sie ist ebenfalls wohldefinert, was aus der Rechtsinvarianz von

R_L folgt: Sei $[w] = [w']$, also wR_Lw'. Dann gilt wegen der Rechtsinvarianz von R_L auch $waR_Lw'a$ und damit $[wa] = [w'a]$. Damit folgt schließlich:

$$\delta(q_0, w) = \delta([\varepsilon], w) = [\varepsilon w] = [w]$$

w wird also akzeptiert, wenn $[w] \in F$ gilt.

- **Zeige, daß das Entfernen überflüssiger Zustände und das Zusammenfassen von äquivalenten Zustände ausreicht, um einen Automaten zu minimieren!**

Wir haben gesehen, daß der Äquivalenzklassenautomat wohldefiniert ist und er dieselbe Sprache entscheidet, wie der nicht minimierte Automat. Der Grund dafür ist, daß äquivalente Zustände beim Lesen eines Wortes in äquivalenten Zuständen landen und wir die Sprache daher nicht „verfälschen“, indem wir diese Zustände zusammenfassen. Dies ist jedoch noch kein Beweis, daß der Automat tatsächlich minimal ist.

Jedoch haben wir mit der Äquivalenzrelation R_A gesehen, daß Zustände auch durch das gelesene Wort charakterisiert werden können. Die Nerode-Relation R_L wiederum hat gezeigt, daß die Äquivalenzklassen von Wörtern, deren Verlängerungen gleich entschieden werden, in einer Äquivalenzklasse von R_L zusammengefaßt werden können. Aus den so entstandenen Äquivalenzklassen konnte man dann einen Automaten entwerfen, der ebenfalls die Sprache L entscheidet und minimal ist.

Wir haben nämlich gezeigt, daß wenn L von einem DFA akzeptiert wurde, es eine Äquivalenzrelation R_A gibt, deren Index endlich ist: $ind(R_A) < \infty$. Danach haben wir gezeigt, daß der Index der Nerode-Relation R_L dann ebenfalls endlich ist, da R_L eine Vergröberung von R_A darstellt. Eine Vergröberung deshalb, weil mehrere Äquivalenzklassen bezüglich R_A in einer Äquivalenzklasse von R_L liegen können. Deshalb galt: $ind(R_L) \leq ind(R_A)$. Aus diesen Äquivalenzklassen konnten wir einen Automaten bauen, der dieselbe Sprache entschieden hat und so viele Zustände hatte, wie R_L Äquivalenzklassen.

3.3 Das Pumping Lemma für endliche Automaten

- **Was ist das Pumping Lemma?**

Der **Satz von Nerode** ist ein **notwendiges und hinreichendes Kriterium** um die Existenz eines DFA für eine Sprache L zu beweisen. Das Kriterium $ind(R_L) < \infty$ ist jedoch nicht immer leicht zu beweisen. Das **Pumping Lemma** ist **nur ein notwendiges Kriterium** um die Regularität von Sprachen zu beweisen. Das heißt, daß jede reguläre Sprache das Pumping Lemma erfüllen muß. Kann man also zeigen, daß eine Sprache L das Pumping Lemma nicht erfüllt, so ist die Sprache L nicht regulär.

- **Wie lautet das Pumping Lemma?**

Sei L eine Sprache, die von einem DFA akzeptiert wird. Dann gibt es eine Konstante N, so daß für alle $z \in L$ mit $z \geq N$ eine Zerlegung $z = uvw$ mit $|uv| \leq N$ und $|v| \geq N$ existiert, so daß für alle $i \geq 0$ auch $uv^iw \in L$ gilt.

- **Beweise das Pumping Lemma!**

Sei A der DFA der L akzeptiert. Dann wählen wir $N = |Q|$. Wenn der Automat nun ein Wort z mit $z \geq N$ liest, muß er **mindestens einen Zustand q zweimal erreicht**. Mit anderen Worten, in der graphischen Darstellung von A wird ein **Kreis durchlaufen**. Nun sei u der Präfix von z, nach dessen Lesen man zum ersten Mal diesen Zustand q erreicht. Und uv sei der Präfix von z, nach dessen Lesen man den Zustand q zum zweiten Mal erreicht. Dann ist der Suffix w, der zum akzeptierenden Zustand q' gehört eindeutig definiert.

Es gilt auch $uv \leq N$, da sich nach den ersten N Buchstaben ein Zustand wiederholen muß. Denn wir starten in q_0 und durchlaufen dann noch N weitere Zustände. Dies sind jedoch insgesamt $N + 1$ und wir haben nur N verschiedene, also wiederholt sich ein Zustand.

Außerdem gilt auf jeden Fall $v \geq 1$. $|v|$ ist nämlich genau dann 1 lang, wenn der Zustand der sich wiederholt eine Schlaufe hat und das Wort z die Schlaufe benutzt.

Sei nun uv^iw das Eingabewort. Dann gilt:

$$\delta(q_0, uv^iw) = \delta(q, v^iw) = \delta(q, w) = q' \in F$$

Oder ganz einfach gesagt: Weil der Automat sich nicht merkt, was er schon vorher gelesen hat, kann man vom Zustand q aus v beliebig oft lesen, da man danach eh wieder im Zustand q ist.

- **Gebe Beispiele für das Pumping Lemma!**

  □ $L = \{w \in \{0,1\}^* | w = w^R\}$:

  Sei N die Konstante aus dem Pumping Lemma. Wir wählen das Wort $0^N1^N0^N$. Jetzt können wir nur im Block der Nullen pumpen und es gilt für jede Zerlegung $uv^2w \notin L$.

  □ $L = \{0^{k^2} | k \geq 1\}$:

  Sei N die Konstante aus dem Pumping Lemma. Wir wählen das Wort 0^{N^2}. Es gilt $1 \leq |V| \leq N$. Für $i = 2$ ist dann $0^{N^2+|V|} \notin L$ wegen:

$$N^2 < N^2 + |V| \leq N^2 + N < N^2 + 2N + 1 = (N+1)^2$$

- **Was ist das verallgemeinerte Pumping Lemma und wie lautet es?**

Es gibt Sprachen, für die zum Nachweis ihrer Nichtregularität das „normale“ Pumping Lemma nicht ausreicht, da nicht erzwungen werden kann, wo gepumpt wird bzw. welches die kritische Stelle in der Sprache ist.

Das allgemeine Pumping Lemma lautet: Sei L eine Sprache, die von einem DFA akzeptiert wird. Dann gibt es eine Konstante N, so daß für jedes Wort $z \in L$ mit $|z| \geq N$ eine Zerlegung $z = tyt'$ existiert, so daß folgendes gilt: $|y| = N$ und es existiert eine Zerlegung von y mit $y = uvw$ mit $|v| \geq 1$ und für alle $i \geq 0$ gilt $tuv^iwt' \in L$.

- **Gebe eine nichtreguläre Sprache an, für die man das verallgemeinerte Pumping Lemma benutzen muß!**

$$L = \{z | (z = 1^k \text{ mit } k \geq 0) \vee (z = 0^j1^{k^2} \text{ mit } j \geq 1 \text{ und } k \geq 0)\}$$

Mit dem normalen Pumping Lemma wählen wir stets die Zerlegung $u = \varepsilon$ und $|v| = 1$. Wählt unser Gegner z.B. das Wort 01^{k^2}, so wählen wir $u = \varepsilon, v = 0, w = 1^{k^2}$. Pumpt unser Gegner ab, so bleiben wir in der Sprache, da $1^{k^2} \in L$ gilt. Sonst pumpen wir die Anzahl der Nullen beliebig auf, das Wort bleibt in der Sprache. Wir müssen also mit dem Pumping Lemma erzwingen, daß wir in den Einsen pumpen.

3.4 Nichtdeterministische endliche Automaten

- **Was ist ein nichtdeterministischer Automat?**

Ein nichtdeterministischer Automat (NFA) ist definiert wie ein DFA, nur ist hier die Zustandsüberführungsfunktion durch eine **Relation** $(Q \times \Sigma \times Q)$ ersetzt. Wenn der Automat im Zustand q den Buchstaben a liest, wechselt der Automat in alle Zustände q', für die es ein (q, a, q') in der Zustandsüberführungsrelation gibt. Man kann diese Relation auch als eine **Abbildung** von $Q \times \Sigma$ **in die Potenzmenge** der Zustände sehen, also als eine Abbildung $\delta : Q \times \Sigma \to \mathcal{P}(Q)$:

$$(q, a) = \bigcup_{(q,a,q') \in \delta} q'$$

- **Wie wird δ auf $Q \times \Sigma^*$ fortgesetzt?**
Dazu müssen wir angeben, was für das **leere Wort** gilt und was für Wörter w mit $|w| \geq 2$ gilt. Für ε gilt $\delta(q,\varepsilon) = \{q\}$. Wir werden später noch sehen, daß es NFAs gibt, die beim Lesen von ε ihren Zustand ändern können, diese Automaten jedoch nicht mächtiger sind als „normale“ NFAs. Für Wörter $w = w'a$ mit $w' \in \Sigma^*$ und $a \in \Sigma$ gelte:

$$\delta(q,w) = \bigcup_{q' \in (q,w')} \delta(q',a)$$

- **Wann akzeptiert ein NFA ein Wort?**
Ein NFA akzeptiert ein Wort w, wenn er nach dem Lesen von w in einem akzeptierenden Zustand sein kann: $\delta(q_0,w) \cap F \neq \emptyset$.

- **Gebe einen NFA für das String Matching Problem an!**
Wir suchen einen String $s = s_1...s_k \in \Sigma^*$ in einem Muster $m = m_1...m_n \in \Sigma^*$. Wie sieht ein NFA für dieses Problem aus? Der NFA muß raten können, wo der String beginnt und sieht aus wie auf Abbildung 4.4.1 im Buch „Theoretische Informatik“ von Ingo Wegener.

- **Gebe einen NFA für die Sprache L_n an, der Menge aller Wörter über $\{0,1\}$, bei denen der n-letzte Buchstabe eine 1 ist.**
Der NFA sieht ähnlich aus, wie der NFA für das String Matching Problem:

$$\text{NFA} = (\{q_0,...,q_n\},\{0,1\},q_0,\delta,\{q_n\})$$

Dabei sieht δ wie folgt aus:

δ	0	1
q_0	q_0	$\{q_0,q_1\}$
q_1	q_2	q_2
$\vdots$	$\vdots$	$\vdots$
q_{n-1}	q_n	q_n
q_n	-	-

Zwar ist jetzt bei q_n nichts angegeben, jedoch kann man sich vorstellen, daß man von q_n in einen Zustand gerät, von dem aus man keinen akzeptierenden Zustand mehr erreichen kann. Wie man sieht, kann man für Wörter, die kürzer als n sind, nicht den akzeptierenden Endzustand q_n erreichen. Für Wörter, die genau n lang sind, kann man q_n nur erreichen, wenn der erste Buchstabe eine 1 ist und für längere Wörter rät der Automat, wann das Restwort nur noch n lang ist. Rät er das Restwort zu kurz, kann der Zustand q_n nicht mehr erreicht werden, rät er das Restwort zu lang, kommen wir, wie oben vereinbart in einen Zustand, von dem aus man q_n nicht mehr erreichen kann.

Unser NFA kommt also mit $n+1$ Zuständen aus. Wie viele Zustände hat ein entsprechender DFA?

- **Zeige, daß jeder DFA für L_n mindestens 2^n Zustände benötigt!**
Dazu müssen wir zeigen, daß alle Wörter der Länge n **nicht Nerode-äquivalent** sind. Das heißt, für alle $x,y \in \{0,1\}^n$ mit $x \neq y$ gilt, daß es ein $1 \leq i \leq n$ gibt mit $x_i \neq y_i$. O.B.d.A. sei $x_i = 0$ und $y_i = 1$. Dann gilt jedoch $x_i 0^{i-1} \notin L$, während $y_i 0^{i-1} \in L$ gilt. Also sind x und y nicht Nerode-äquivalent. Da dies jedoch für beliebige x,y gilt, sind alle x,y mit $x \neq y$ nicht Nerode-äquivalent und jedes Wort bildet seine eigene Äquivalenzklasse. Wieviele Wörter w der Länge n gibt es jedoch? Genau 2^n. Also folgt daraus: $ind(R_{L_n}) \geq 2^n$.

Als Beispiel betrachten wir für die Sprache L_4 die beiden Wörter 1001 und 1011: Bis auf die dritte Stelle $i = 3$ sind die Wörter identisch. Wenn wir $i-1 = 3-1 = 2$ Nullen an das Wort 1001 hängen, dann gehört das Wort 100100 nicht zur Sprache, während das Wort

101100 zur Sprache gehört. Die Wörter sind also nicht Nerode-äquivalent. Man kann also zwei beliebige ungleiche Wörter als nicht Nerode-äquivalent nachweisen.

Man kann dies auch so interpretieren, daß der DFA sich vom bisher gelesenen String immer die letzten n Buchstaben merken muß, da er ja nicht weiß, wann die Eingabe zu Ende ist. Und dafür brauch man 2^n Zustände.

- **Können NFAs mehr als DFAs?**

Nein. Hierzu muß man zeigen, daß es für jeden NFA A einen DFA A' gibt, der im worst-case exponentiell mehr Zustände hat. Daß dieser worst-case auch tatsächlich eintreten kann zeigt die oben besprochene Sprache L_n.

Wir konstruieren den DFA über die sogenannte **Potenzmengenkonstruktion**, d.h. unser DFA merkt sich über die Potenzmenge in welchen Zuständen der NFA nach dem Lesen eines Wortes sein kann. Dieser Schritt ist eigentlich logisch, da wir ja schon bei der Definition von NFAs gesagt haben, daß wir die Zustandsübergangsrelation als eine Abbildung von $Q \times \Sigma$ in die Potenzmenge $\mathcal{P}(Q)$ interpretieren können. Der NFA A sei gegeben durch:

$$A = (Q, \Sigma, q_0, \delta, F)$$

Dann sieht der DFA A' wie folgt aus:

$$A' = (Q', \Sigma', q_0', \delta', F')$$

Dabei ist $Q' = \mathcal{P}(Q)$ und damit gilt $|Q'| = 2^{|Q|}$. Das Eingabealphabet bleibt gleich, also $\Sigma' = \Sigma$. Der neue Startzustand q_0' ist $\{q_0\}$. Es ist hier schon anzumerken, daß die Zustände durch Mengen dargestellt sind, d.h. der Startzustand ist das Element der Potenzmenge $\mathcal{P}(Q)$, welches nur den Startzustand enthält. Die Menge der akzeptierenden Endzustände F' ist die Menge der Elemente in der Potenzmenge, die mindestens einen akzeptierenden Endzustand bezüglich Q enthalten, das heißt $F' = \{q' \in Q' | q' \cap F \neq \emptyset\}$. Und für die Zustandsüberführungsfunktion δ' gilt schließlich:

$$\delta'(q', a) = \bigcup_{q \in q'} \delta(q, a)$$

$\delta'(q', a)$ ist also die Menge der Zustände, die der NFA nach dem Lesen von a erreichen kann, wenn er genau in den Zuständen aus q' sein kann.

Um zu zeigen, daß A' die gleiche Sprache wie A entscheidet, reicht es zu zeigen, daß $\delta'(q_0', w) = \delta(q_0, w)$ gilt. Dies wird per **Induktion** über die Länge von w gemacht. Für das **leere Wort** ε gilt:

$$\delta'(q_0', \varepsilon) = q_0' = \{q_0\} = \delta(q_0, w)$$

Für Wörter $|w| \geq 2$ mit $w = w'a, w' \in \Sigma^*, a \in \Sigma$ gilt:

$$\delta'(q_0', w) = \delta'(\delta'(q_0', w'), a) = \delta'(\delta(q_0, w'), a) = \bigcup_{q \in \delta(q_0, w')} \delta(q, a) = \delta(q_0, w)$$

An der Stelle $\delta'(\delta'(q_0', w'), a) = \delta'(\delta(q_0, w'), a)$ geht dabei die **Induktionsvorraussetzung** ein, d.h. man geht davon aus, daß $\delta'(q_0', w') = \delta(q_0, w')$ ist, daß also die Menge der Zustände nach dem Lesen von w' bei beiden Automaten gleich ist. Daraus folgt dann, daß auch die Menge der Zustände gleich bleibt, wenn man einen weiteren Buchstaben liest. Man zeigt, daß nichts Falsches folgt, wenn man davon ausgeht, daß die Induktionsvorraussetzung richtig ist.

Fazit: Zu jeden NFA mit n Zuständen gibt es einen entsprechenden DFA mit 2^n Zuständen, den man noch effizient minimieren kann.

- **Was ist das Problem bei der Potenzmengenkonstruktion?**

Bei der Potenzmengenkonstruktion stellt sich das Problem, daß die **Zustandsmenge** des entsprechenden DFA **sehr groß werden kann**. Wenn der NFA hunderte oder gar tausende

von Zuständen hat, kann man aus praktischer Sicht keinen minimalen DFA dafür entwerfen, da ein DFA zwischenzeitlich $2^{1000} \approx 10^{301}$ Zustände hätte, wenn der NFA 1000 Zustände hat.

Wie wir schon wissen gibt es zwei Dinge, die einen DFA unter Umständen größer machen, als er eigentlich sein müßte:

1. Überflüssige, weil nicht erreichbare Zustände.
2. Äquivalente Zustände.

Gegen die Erzeugung äquivalenter Zustände gibt es noch keinen effektiven Algorithmus, wohl aber gegen die Erzeugung überflüssiger Zustände. Dazu benutzt man eine **Queue** Q^* und ein **Dictionary** D. Zu Beginn wird $q_0' = \{q_0\}$ in die Q^* und D gepackt. Solange Q^* nicht leer ist, entnehmen wir das erste Element q' und bestimmen für jedes $a \in \Sigma$ die Menge der Zustände, die beim Lesen von a in den Zuständen in q' erreichbar sind. Wir berechnen also $\delta'(q', a)$. Für jedes solcher $\delta'(q', a)$ wird überprüft, ob wir diesen Zustand schon erzeugt haben, indem wir ihn in D suchen. Ist er Zustand noch nicht vorhanden, wird er in D und Q^* eingefügt.

Die Dictionary-Laufzeit ist, wenn wir einen 2-3-Baum benutzen, in jedem Fall $O(|Q|)$, da höchstens $2^{|Q|}$ Zustände eingefügt werden können und 2-3-Bäume eine worst-case Laufzeit von $O(\log_2 n)$ haben.

- **Was ist ein NFA mit ε-Übergängen?**

Ein NFA mit ε-Übergängen kann seinen Zustand ändern, ohne etwas zu lesen. Gilt z.B. $\delta(q, \varepsilon) = q'$, so kann der NFA, wenn er im Zustand q ist „spontan“ in den Zustand q' wechseln. Es ist anzumerken, daß es zu jedem NFA_ε mit n Zuständen einen äquivalenten NFA ohne ε-Übergänge gibt, der ebenfalls mit n Zuständen auskommt.

3.5 Effiziente Algorithmen für endliche Automaten

- **Welche Aussagen über Sprachen kann man in effizienter Zeit entscheiden?**

  - **Leerheitstest**: (Ob die erkannte Sprache $L = \emptyset$ ist)

    Mit einem **DFS vom Startzustand** aus wird im als Graphen gegebenen Automaten getestet, ob ein **akzeptierender Zustand erreicht** werden kann.

  - **Vollständigkeitstest**: (Ob die erkannte Sprache $L = \Sigma^*$ ist)

    Mit einem **DFS vom Startzustand** aus wird im als Graphen gegebenen Automaten getestet, ob ein **nicht akzeptierender Zustand** erreicht werden kann.

  - **Unendlichkeitstest**: (Ob für die erkannte Sprache $|L| = \infty$ gilt)

    Zuvor **entfernen** wir **überflüssige Zustände**. Die Sprache ist unendlich, wenn es in der Graphendarstellung des Automaten mindestens einen **Kreis** gibt. Da ein Graph jedoch exponentiell viele Kreise enthalten kann, werden die **Kanten des Graphen umgedreht** und von **akzeptierenden Zuständen** aus nach Kreisen **gesucht**. Dies wird mit Hilfe eines **DFS** gemacht, der die Kanten disjunkt in Tree-, Back-, Forward- und Cross-Kanten einteilt. Gilt $B \neq \emptyset$, so enthält der Graph Kreise und die erkannte Sprache ist damit unendlich.

- **Warum ist eine Sprache L endlich, wenn der Graph des DFA keine Kreise enthält?**

Weil dann ein Wort nur über kreisfreie Wege deren Länge durch $|Q|$ beschränkt ist, erkannt werden soll. Also ist L endlich, da es nur endlich viele solcher Wege gibt.

- **Gegen welche Operationen sind reguläre Sprachen abgeschlossen?**

- **Komplementbildung**:

  Die Menge F der akzeptierenden Zustände wird gegen $Q - F$ ausgetauscht. Wörter die vorher nicht akzeptiert wurden, werden jetzt akzeptiert und umgekehrt.

- **Vereinigung**:

  Wir bilden den Produktautomaten mit $Q = Q_1 \times Q_2$, q_0 ist das Paar von Anfangszuständen von A_1 und A_2. F ist die Menge aller (q_1, q_2) für die gilt: $q_1 \in F_1$ **oder** $q_2 \in F_2$. Für die Zustandübergangsfunktion gilt:

  $$\delta((q_1, q_2), a) = (\delta(q_1, a), \delta(q_2, a))$$

- **Durchschnitt**:

  Hier kann man entweder ein ähnliche Konstruktion wie für den Durchschnitt benutzen, oder die Regel von de Morgan anwenden. Bei der obigen Konstruktion muß man nur die Menge der akzeptierenden Zustände setzen als (q_1, q_2) mit $q_1 \in F_1$ **und** $q_2 \in F_2$. Oder man nutzt mit Hilfe des Satzes von de Morgan die Abgeschlossenheit gegen Komplement und Vereinigung aus:

  $$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$$

4 Grammatiken, die Chomsky-Hierarchie und das Wortproblem

4.1 Grammatiken und die Chomsky-Hierarchie

- Wir wollen **Regelsysteme** entwerfen, mit denen sich genau die Wörter einer Sprache ableiten lassen. Diese Systeme werden auch **Grammatiken** genannt.
- Beispiel: Grammatik für die NP-vollständige Sprache aller Graphen $G = (V,E)$, die eine $\left\lfloor \frac{|V|}{2} \right\rfloor$-elementige Clique haben. **Konstruktion** klar.

• **Woraus bestehen Grammatiken?**
Eine Grammatik G ist ein 4-Tupel $G = (V,T,S,P)$. Dabei sind:

- V, die endliche zu T disjunkte Menge an **Variablen**.
- T (oder Σ), die endliche Menge an **Terminalzeichen**. Über diesem Alphabet ist die Sprache definiert.
- $S \in V$, das **Startsymbol**.
- P, eine endliche Menge an **Produktionen**, der Form $(V \cup T)^+ \to (V \cup T)^*$. Die linke Seite der Produktion ist also nicht leer.

• **Welche Notation benutzt man für die Anwendung von Produktionen bei Grammatiken?**

- $w \to z$, wenn sich das Wort z mit **einer** Produktion aus dem Wort w ableiten läßt.
- $w \overset{*}{\to} z$, wenn sich das Wort z mit **endlich vielen** Produktionen aus dem Wort w ableiten läßt.

• **Welche Sprache $L(G)$ erzeugt eine Grammatik?**
Die von einer Grammatik G erzeugte Sprache $L(G)$ ist die Menge aller $z \in T^*$, für die $S \overset{*}{\to} z$ gilt:

$$L(G) = \bigcup_{z \in T^*} S \overset{*}{\to} z$$

• **Was ist das Wortproblem?**

- Entscheidung, ob für ein Wort $w \in T^*$ gilt: $w \in L(G)$.
- In der Anwendung: Ob w ein syntaktisch korrektes Programm ist.
- Zwiespalt: Hohe Freiheit - Wortproblem schwer, Niedrige Freiheit - nicht mehr ausdrucksstark genug

• **Wieviele Grammatikklassen hat die Chomsky-Hierarchie? Beschreibe sie!**

- Grammatiken **ohne weitere Einschränkungen** heißen Grammatiken vom Typ **Chomsky-0**.
- Bei **kontextsensitiven** Grammatiken (**Chomsky-1**) haben alle Produktionen die Form $u \to v$ mit $u \in V^+$, $v \in (V \cup T - \{S\})^+$ und $|u| \leq |v|$. Oder aber $S \to \varepsilon$. Die rechte Seite einer kontextsensitiven Grammatik ist also immer mindestens so lang, wie die linke Seite. Daher werden diese Grammatiken auch **monotone** Grammatiken genannt.

- Grammatiken, bei denen alle Produktionen die Form $A \to v$ mit $A \in V$ und $a \in (V \cup T)^*$ haben, heißen **kontextfrei** oder Grammatiken vom Typ **Chomsky-2**.
- Grammatiken, bei denen alle Produktionen die Form $A \to v$ mit $A \in V$ und $v = \varepsilon$ oder $v = aB$ mit $a \in T$ und $B \in V$ haben, heißen **rechtslinear, regulär** oder Grammatiken vom Typ **Chomsky-3**-

4.2 Chomsky-0-Grammatiken und rekursiv aufzählbare Sprachen

• **Was hat die Klasse Chomsky-0 mit der Klasse der rekursiv aufzählba-ren Sprachen zu tun?**

- Die Klasse Chomsky-0 stimmt mit der Klasse der rekusiv aufzählbaren Sprachen überein.
- Man nennt Chomsky-0-Grammatiken auch **general-rewriting-systems**.
- Grammatiken sind rückwärts arbeitende Turingmaschinen und Turingmaschinen sind rückwärts arbeitende Grammatiken.

• **Zeige, daß folgender Satz gilt: Ist L rekursiv aufzählbar, so gibt es eine Chomsky-0-Grammatik G mit L(G)=L.**
L ist rekursiv aufzählbar, also gibt es eine deterministische Turingmaschine, die genau die Wörter $w \in L$ akzeptiert. Wir werden nun einige Modifikationen vornehmen:

- M hat nur **einen akzeptierenden Endzustand** q^*.
- q^* wird erst erreicht, nachdem das **ganze Band gesäubert**, also mit Blanks beschrieben ist.
- M schreibt zu Beginn # hinter die Eingabe und benutzt die Zellen hinter # nicht mehr.
- Neue Anfangskonfiguration ist also $q_0 w_1 \ldots w_n \#$.
- q_0 kommt nur in der Anfangskonfiguration vor.

Die Grammatik G soll also vom akzeptierenden Endzustand aus, die Rechnung rückwärts simulieren und die Startkonfiguration erzeugen können, wenn $w \in L$ ist. Dazu benutzen wir folgende Regeln:

- Erzeugung der akzeptierenden **Schlußkonfiguration** mit genügend **Blankbuchstaben**. Über die Blankbuchstaben erzeugt man sich also nichtdeterministisch ein Band, welches lang genug ist:

$$S \to q^*$$
$$q^* \to q^* B$$
$$q^* \to B q^*$$

- Die **Rückwärtsrechnung**:

  Falls $\delta(q,a) = (q',a',R)$, enthält G die Produktion

  $$a'q' \to qa$$

  Falls $\delta(q,a) = (q',a',L)$, enthält G die Produktion

  $$q'ba' \to bqa$$

Falls $\delta(q,a) = (q',a',N)$, enthält G die Produktion

$$q'a' \to qa$$

- Die Konfiguration ist nun von der Form $B...Bq_0w_1...w_n\#$. Da wir nur das Eingabewort erzeugen wollen, müssen wir noch q_0, B und # löschen können. Die machen wir mit den **Schlußregeln**:

$$Bq_0 \to q_0$$
$$q_0a \to aq_0$$
$$q_0\# \to \varepsilon$$

Die Maschine gelangt nur für Wörter $w \in L$ in den akzeptierenden Endzustand q^*, also können auch nur diese Wörter nichtdeterministisch von der Grammatik erzeugt werden. Da die Grammatik jedoch auch aufgrund der Regeln für die Rückwärtsrechnung nur die Rechnungen von M rückwärts erzeugen kann gilt $L(G) = L$.

• **Zeige, daß folgender Satz gilt: Wenn L durch eine Chomsky-0-Gram-matik beschrieben wird, so gibt es eine NTM M, die genau L akzeptiert.**
Diese Richtung ist einfacher: Die NTM M schreibt das Startsymbol S an eine freie Stelle des Bandes. Danach wählt sie nichtdeterministisch eine Produktion und schaut nach, ob das so entstandene Wort mit der Eingabe w übereinstimmt. Für $w \in L(G)$ akzeptiert die Maschine, sonst nie.

4.3 Chomsky-3-Grammatiken, reguläre Sprachen und Ausdrücke

• **Zeige, daß die Klasse der von DFAs akzeptierten Sprachen mit der von Chomsky-3-Grammatiken erzeugten Sprachen übereinstimmt!**
Zu zeigen sind zwei Richtungen:

1. DFA $\to$ Chomsky-3-Grammatik:

   Gegeben sei ein DFA A_L für eine Sprache L über Σ. Unsere Grammatik sieht wie folgt aus:

$$V = Q$$
$$T = \Sigma$$
$$S = q_0$$
$$q \to aq' \text{ für } \delta(q,a) = q'$$
$$q \to \varepsilon \text{ für } q \in F$$

   Wenn ein Wort $w = w_1...w_n$ nun zur Sprache L gehört, also $w \in L$ gilt, so durchläuft der Automat die Zustände $q_0, q_1, ..., q_n$ mit $q_n \in F$. Also gilt:

$$q_0 \to w_1q_1 \to w_1w_2q_2 \to ... \to w_1w_2...w_nq_n \to w_1w_2...w_n$$

   Für jedes Wort sieht die Ableitung jedoch so aus. Und man kann die Ableitungen erst „terminieren“, wenn $q \in F$ gilt, da dann auch eine Ableitung $q \to \varepsilon$ existiert.

2. Chomsky-3-Grammatik $\to$ DFA:

   Gegeben ist also eine Chomsky-3-Grammatik, für die wir einen NFA entwerfen. Dieser kann mit Hilfe der Potenzmengenkonstruktion in einen DFA umgewandelt werden und sieht wie folgt aus:

$$Q = V$$
$$\Sigma = T$$
$$q_0 = S$$
$$\delta(A,a) = \{B | A \to aB \in P\}$$
$$F = \{A \in V | A \to \varepsilon \in P\}$$

Falls $w = w_1...w_n \in L$ hat die Ableitung folgendes Aussehen:

$$S \to w_1A_1 \to w_1w_2A_2 \to ... \to w_1...w_nA_n \to w$$

Dann gibt es jedoch auch eine akzeptierende Konfigurationenfolge:

$$Sw_1...w_n \vdash A_1w_2...w_n \vdash ... \vdash A_{n-1}w_n \vdash A_n$$

• **Wie ist die Menge der regulären Ausdrücke über dem endlichen Alphabet Σ definiert?**

1. Die Ausdrücke $\emptyset, \varepsilon$ und a für $a \in \Sigma$ sind reguläre Ausdrücke. Also die leere Sprache, die Sprache des leeren Wortes und die Sprache des einbuchstabigen Wortes.
2. Sind A_1 und A_2 reguläre Ausdrücke, so auch $A_1 + A_2$ (Vereinigung), $A_1 \cdot A_2$ (Konkatenation) und A_1^* (Kleenescher Abschluß).
3. Durch endliche Anwendung der Regeln 1 und 2 lassen sich alle regulären Ausdrücke darstellen.

• **Zeige, daß sich genau die regulären Sprachen durch reguläre Ausdrücke darstellen lassen!**
Wieder sind zwei Richtungen zu zeigen:

1. Reguläre Ausdrücke $\to$ DFA:

   Für die Ausdrücke in Regel 1 lassen sich einfach entsprechende DFAs angeben. Desweiteren haben wir schon gezeigt, daß DFAs gegen Vereinigung, Konkatenation und Kleeneschen Abschluß abgeschlossen sind.

2. DFA $\to$ Reguläre Ausdrücke:

   Diese Richtung ist schon etwas kniffeliger. Wir haben also einen DFA gegeben und sollen für die Sprache, die er entscheidet, einen regulären Ausdruck angeben. Wir gehen diese Richtung mit dem Prinzip der dynamischen Programmierung an, d.h. wir bauen aus vielen kleinen Lösungen die Gesamtlösung zusammen.

   Was sind nun die kleineren Lösungen? Wir können uns doch fragen, welche Wörter wir auf dem Weg von einem Zustand zu einem anderen lesen können. Dann interessieren uns doch alle Wörter, die wir lesen können, um vom Startzustand q_0 in einen akzeptierenden Endzustand $q \in F$ zu kommen. Dabei kommen alle Zustände als Zwischenzustände in Frage. Damit haben wir auch schon einen Ansatz für die dynamische Programmierung. Wir werden nun nach und nach immer mehr Zustände zulassen.

   Es gelte also $Q = \{1, ..., n\}$. Dann sei R_{ij}^k die Menge der Wörter, die man lesen kann, um von Zustand i nach Zustand j zu kommen, wenn die Zustände $1, ..., k$ als Zwischenzustände erlaubt sind. Das Endergebnis ist dann folgende Menge:

   $$L = \bigcup_{i \in F} R_{1i}^n$$

   Da die Vereinigung eine der erlaubte Operationen bei regulären Sprachen ist, genügt es also zu zeigen, daß ein R_{1i}^n durch einen regulären Ausdruck darstellbar ist. Dies geschieht per Induktion über die Zahl k erlaubter Zwischenzustände.

   R_{ij}^0 ist die Menge der Wörter w, für die $\delta(i, w) = j$ gilt. Es ist also die Menge der Wörter, mit denen man ohne einen Zwischenzustand zu benutzen von einen Zustand

i in einen Zustand j kommt. Dies ist eine eventuell leere Teilmenge von $\{\varepsilon\} \cup \{a | a \in \Sigma\}$. Damit ist R_{ij}^0 durch einen regulären Ausdruck darstellbar.

Die Bellmann'sche Optimalitätsgleichung für den allgemeinen Fall lautet:

$$R_{ij}^k = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}$$

Die Menge der Wörter, die man lesen kann, um vom Zustand i in den Zustand j zu gelangen, wenn als Zwischenzustände nur die Zustände $1, ..., k$ erlaubt sind ist also gleich der Menge der Wörter, die man lesen kann ohne den Zustand k zu benutzen plus die Konkatenation von Wörtern, die zum Zustand k führen, diesen eventuell beliebig oft durchlaufen und dann vom Zustand k zum Zustand j führen.

5 Kontextfreie Grammatiken, Chomsky-2

5.1 Einleitung

- **Wann ist eine Grammatik vom Typ Chomsky-2?**
Eine Grammatik ist vom Typ Chomsky-2, wenn alle Produktionen der Form $v \to u$ mit $v \in V$ und $u \in (V \cup T)^*$ sind.

- **Zeige, daß kontextfreie Grammatiken mehr können als Chomsky-3-Grammatiken!**
Die Sprachen $L_1 = \{0^i 1^i | i \geq 1\}$ und $L_2 = \{w \in \{0,1\}^* | w = w^R\}$ haben sich als nicht regulär erwiesen. Es ist jedoch leicht zwei Chomsky-2-Grammatiken dafür anzugeben. Für L_1:

$$S \to 01, S \to 0S1$$

Und für L_2:

$$S \to \varepsilon, S \to 0, S \to 1, S \to 0S0, S \to 1S1$$

- **Was ist ein Syntaxbaum?**

  - ☐ Ist die **graphische Darstellung** von Ableitungen. An der Wurzel steht das Startsymbol, in inneren Knoten Variablen. Ein Knoten A hat die Kinder $\alpha_1, ..., \alpha_n \in (V \cup T)$, wenn die Ableitung $A \to \alpha_1 ... \alpha_n$ erlaubt ist.

  - ☐ Zu jeder Ableitung gehört genau ein Syntaxbaum, während zu einem Syntaxbaum mehrere Ableitungen des selben Wortes gehören können.

- **Was ist eine Links-Ableitung (Rechts-Ableitung)?**
Eine Links-(Rechts-)Ableitung ist eine Ableitung, bei der in jedem String die linkeste (rechteste) Variable zuerst abgeleitet wird.

- **Wann heißt eine kontextfreie Grammatik eindeutig?**
Eine kontextfreie Grammatik G heißt eindeutig, wenn es zu jedem Wort $w \in L(G)$ **genau einen** Syntaxbaum gibt.

- **Wann heißt eine kontextfreie Sprache L eindeutig?**
Eine kontextfreie Sprache L heißt eindeutig, wenn es eine eindeutige kontextfreie Grammatik G gibt, so daß $L(G) = L$ gilt, eine eindeutige Grammatik also die Sprache erzeugt.

5.2 Chomsky-Normalform für kontextfreie Grammatiken

- **Wann ist eine Grammatik in Chomsky-Normalform?**
Eine Grammatik ist in Chomsky-Normalform, wenn alle Ableitungen der Form $A \to BC$ oder $A \to a$ sind, mit $A, B, C \in V$ und $a \in T$.

- **Was muß man bei Grammatiken in Chomsky-Normalform beachten?**
Man muß beachten, daß Grammatiken in Chomsky-Normalform das leere Wort nicht erzeugen können. Man kann jedoch eine neue Variable S' einführen, für die gilt:

$$S' \to \varepsilon, S' \to S$$

Dies ist nur eine kleine Störung der Chomsky-Normalform.

- **Was ist der Vorteil an der Chomsky-Normalform?**

- Das **Wortproblem** wird durch Grammatiken in Chomsky-Normalform erst wirklich effizient lösbar, da die **Syntaxbäume** solcher Grammatiken **binär** sind.
- **Syntaxbäume** haben **lineare Tiefe**, da ε-Regeln eliminiert sind und auch keine Kettenregeln mehr vorhanden sind. Ein Wort der Länge n kann daher in höchstens $n-1$ Schritten auf n Variablen auf der rechten Seite anwachsen, die in n Schritten durch Terminalzeichen ersetzt werden. Also sind höchstens $2n-1$ Ableitungen möglich.

- **Wie führt man eine Grammatik in Chomsky-Normalform über?**

Die Überführung einer kontextfreien Grammatik G in eine Grammatik in Chomsky-Normalform geschieht in **vier Schritten**:

1. **Separation**:

   Ein Grammatik heißt separiert, wenn auf der **rechten Seite** von Ableitungen **entweder nur Variablen** stehen **oder ein Terminalzeichen**. Also führt man für jedes Terminalzeichen a auf der rechten Seite eine neue Variable Y_a ein und erlaubt die Ableitung $Y_a \to a$. Ein Beispiel:

   $$A \to BaCbcD \Rightarrow A \to BY_aCY_bY_cD, Y_a \to a, Y_b \to b, Y_c \to c$$

   Wie groß ist der **Zuwachs an Größe** der Grammatik? Die **Größe** $s(G)$ einer Grammatik ist die Anzahl an Variablen und Terminalzeichen in allen Regeln. Wir erreichen einen maximalen Zuwachs an Größe, wenn in allen Regeln auf der rechten Seite nur Terminalzeichen stehen. Die Regel $A \to a_1...a_m$ mit $a_i \in T$ verursacht in der Größe der Grammatik $m+1$ Einheiten, die auch bleiben, da wir ja anstatt der Terminalzeichen jetzt Variablen benutzen. Hinzu kommen jedoch noch m Regeln, die jeweils 2 Einheiten verursachen. Also:

   $$\lim_{m\to\infty} \frac{m+1+2m}{m+1} = \lim_{m\to\infty} \frac{3m+1}{m+1} = 3$$

   Die Größe der Grammatik **verdreifacht** sich also höchstens.

2. **Beschränkung der rechten Seite auf zwei Variablen**:

   Stehen in einer separierten Grammatik in einer Ableitung auf der rechten Seite $m \geq 3$ Variablen, so führen wir $m-2$ neue Variablen ein. Eine Regel $A \to B_1...B_m$ verändern wir mit den Extravariablen $C_i, 1 \leq i \leq m-2$ zu:

   $$A \to B_1C_1, C_1 \to B_2C_2, C_2 \to B_3C_3, ..., C_i \to B_{i+1}C_{i+1}, C_{m-2} \to B_{m-1}B_m$$

   Auch hier **verdreifacht** sich die Größe der Grammatik höchstens. Aus einer Regel mit $m+1$ Variablen werden $m-1$ Regeln mit je drei Variablen:

   $$\lim_{m\to\infty} \frac{3(m-1)}{m+1} = \lim_{m\to\infty} \frac{3m-3}{m+1} = 3$$

3. **Streichung der ε-Regeln**:

   Dazu berechnen wir die Menge V' aller Variablen A für die $A \overset{*}{\to} \varepsilon$ gilt. Wir betrachten logischerweise also nur noch Regeln, bei denen auf der rechten Seite Variablen oder ε steht ($A \to \varepsilon, A \to B, A \to BC$).

   Zu Beginn werden alle Variablen A in V' aufgenommen, für die $A \to \varepsilon$ gilt, die also in **einem Schritt zum leeren Wort** abgeleitet werden können. Jedesmal, wenn ein A in V' aufgenommen wird, wird es auch in eine **Queue** Q aufgenommen und alle Seiten mit A auf der linken Seite müssen nicht mehr betrachtet werden, da wir ja schon $A \overset{*}{\to} \varepsilon$ herausgefunden haben.

Dann nehmen wir den nächsten Buchstaben A aus der Queue Q (solange diese nicht leer ist) und ersetzen in allen Regeln jedes Vorkommen von A durch ε. Entsteht dabei eine Regel $B \to \varepsilon$ wird auch B in V' aufgenommen und in Q eingefügt. Am Ende enthält V' alle Variablen für die $A \stackrel{*}{\to} \varepsilon$ gilt. Dies wird per **Induktion** über die Länge l einer kürzesten ε-Ableitung gezeigt. Für $l = 1$ wird $A \to \varepsilon$ sofort in V' aufgenommen. Für $l > 1$ gilt $A \to B \stackrel{*}{\to} \varepsilon$ oder $A \to BC \stackrel{*}{\to} \varepsilon$. Dann haben jedoch B bzw. B und C kürzere ε-Ableitungen und werden zuvor in V' aufgenommen. Dann entsteht jedoch auch die Regel $A \to \varepsilon$ und A wird in V aufgenommen.

Jetzt werden alle Regeln der Form $A \to \varepsilon$ gestrichen. Damit nicht auf einmal weniger Wörter erzeugbar sind, fügen wir zu der Regel $A \to BC$ noch die Regeln $A \to B$, falls $C \in V'$, und $A \to C$, falls $B \in V'$ ist, hinzu.

Die Größe der Grammatik wächst also höchstens um den Faktor $\frac{7}{3}$, da man vorher eine Ableitung mit drei Variablen hatte und jetzt höchstens drei Ableitungen mit sieben Variablen.

Durch die neuen Regeln können auch **nicht mehr Wörter** erzeugt werden, da falls $C \in V'$ ist, auch schon vorher $A \to BC \stackrel{*}{\to} B$ galt, wegen $C \stackrel{*}{\to} \varepsilon$. Es können auch **nicht weniger Wörter** erzeugt werden. Sei $w \in L$ und ein Syntaxbaum für w in der Grammatik mit ε-Regeln gegeben. Wenn $\varepsilon-$Ableitungen benutzt wurden, gibt es spezielle Knoten v, für die gilt, daß alle Blätter in Teilbaum mit Wurzel v ε-Blätter sind, während dies für den Vater des Knotens v nicht gilt. Der Vaterknoten repräsentiert also eine Regel $A \to BC$ und entweder gilt $B \stackrel{*}{\to} \varepsilon$ oder $C \stackrel{*}{\to} \varepsilon$. Dann können wir jedoch den Teilbaum abschneiden und stattdessen gleich die Regel $A \to C$ benutzen, wenn o.B.d.A. der linke Teilbaum nur ε-Blätter enthielt.

Zusammenfassend läßt sich also sagen, daß durch die Streichung der ε-Regeln die **Entscheidung**, ob ein Teilstring einer Ableitung zu ε abgeleitet werden soll, **früh erzwungen** wird.

4. **Elimination der Kettenregeln**:

   Bei der Elimination der ε-Regeln entstehen also Regeln der Form $A \to B$. Es kann dabei auch vorkommen, daß folgende Regel möglich ist:

   $$A_1 \to A_2 \to A_3 \to ... \to A_m \to A_1$$

   Dabei können die Variablen $A_i, 1 \leq i \leq m$ auch eventuell noch zu anderen Strings abgeleitet werden. Jedoch sind sie anscheinend alle untereinander ersetzbar, da man aus einer Variablen, ja jede andere erreicht. Daher kann man die Variablen $A_1, ..., A_m$ in allen Regeln durch eine Variable A_1 ersetzen.

5.3 Der Cocke-Younger-Kasami Algorithmus (CYK)

- **Was ist der Cocke-Younuger-Kasami Algorithmus?**

Der CYK-Algorithmus ist ein Algorithmus, mit dem sich das Wortproblem für kontextfreie Grammatiken in Chomsky-Normalform effizient entscheiden läßt. Genauer gesagt, benötigt er Zeit $O(|P|n^3)$ oder für eine feste Programmiersprache $O(n^3)$, da hier $|P|$ als eine Konstante betrachtet werden kann und nur noch unterschiedlich lange Programmtexte benutzt werden.

- **Wie funktioniert der Cocke-Younger-Kasami Algorithmus?**

Gegeben sei eine kontextfreie Grammatik G in Chomsky-Normalform. Ist die Grammatik nicht in Chomsky-Normalform, so läßt sie sich effizient konstruieren. Weiter gegeben sei $w = w_1...w_n$, das Wort für das entschieden werden soll, ob $w \in L(G)$ gilt. Wir wollen nun

die Menge V_{1n} aller Variablen A berechnen für die gilt $A \stackrel{*}{\to} w_1...w_n$. Offensichtlich ist $w \in L(G)$ genau dann, wenn $S \in V_{1n}$ gilt.

Geht man dieses Problem **Top-Down** an (also z.B. mit einem Divide-And-Conquer Algorithmus) so erhält man leicht exponentielle Kosten, da man nicht von vornherein weiß, wo man das größere Problem teilen muß. Also benutzen wir die Methode der dynamischen Programmierung und gehen **Bottom-Up** vor.

Allgemein sei V_{ij} die Menge aller Variablen A für die $A \stackrel{*}{\to} w_i...w_j$ gilt. Wir berechnen die V_{ij} jetzt nach wachsendem $l = j - i$ und beginnen mit $l = 0$. Die Menge der Variablen V_{ii} zu berechnen, ist jedoch trivial, da die Grammatik in Chomsky-Normalform gegeben ist. Es gilt nämlich $A \in V_{ii}$, wenn $A \to w_i$ gilt, A also in einem Schritt zu w_i abgeleitet werden kann. Die Berechnung **eines** w_i ist also in Zeit $O(|P|)$ möglich, da einfach nur alle Produktionen angeschaut werden müssen.

Für $l > 0$ gilt nun, daß $A \in V_{ij}$ ist, wenn es eine Ableitung $A \to BC$ und ein $k \in \{i, j-1\}$ gibt, so daß $B \in V_{ik}$ und $C \in V_{k+1,j}$ ist, also $B \stackrel{*}{\to} w_i...w_k$ und $C \stackrel{*}{\to} w_{k+1}...w_j$ gilt. Wir müssen für ein V_{ij} höchstens $|P|$ Produktionen betrachten und für jede Produktion kann das (Teil-)Wort auf höchstens $n-1$ Arten zerlegt werden. Also reicht für **ein** V_{ij} Rechenzeit $O(|P|n)$.

Wieviele Mengen V_{ij} gibt es nun? Genau $\binom{n}{2} + n = O(n^2)$. Also ist der Algorithmus nach oben durch $O(|P|n^3)$ beschränkt.

- **Wie läßt sich mit Hilfe des Cocke-Younger-Kasami Algorithmus ein Syntaxbaum konstruieren?**

Indem man für jeden Eintrag $A \in V_{ij}$ auch den Grund für den Eintrag vermerkt. Also die Regel $A \to BC$ mit dem dazugehörigen k für $l > 0$ oder $A \to w_i$ für $i = j$.

- **Wie sorgt man für einen effizienten look-up?**

Indem man für jedes V_{ij} ein Array der Länge $|V|$ anlegt,in dem man markiert, ob eine Variable in V_{ij} vorkommt oder nicht. Der look-up ist dann in Zeit $O(1)$ möglich.

5.4 Pumping Lemma und Ogden's Lemma für kontextfreie Sprachen

- **Wie lautet das Pumping Lemma für kontextfreie Sprachen?**

In Quantorenschreibweise:

$$
\begin{aligned}
&L \text{ kontextfrei } \Rightarrow \\
&\exists N \in \mathbb{N} : \\
&\forall z \in L : |z| \geq N : \\
&\exists \text{ Zerlegung } z = uvwxy \text{ mit } |vx| \geq 1 \text{ und } |vwx| \leq N : \\
&\forall i \geq 0 : uv^i wx^i y \in L
\end{aligned}
$$

Oder in Worten: Wenn L kontextfrei ist, gibt es eine Konstante $N \in \mathbb{N}$, so daß für alle Wörter $z \in L$, die mindestens so lang wie N sind, folgendes gilt: z läßt sich so in $uvwxy$ zerlegen, daß v oder x mindestens einen Buchstaben enthalten und vwx nicht länger als die Konstante N ist. Desweiteren gilt dann, daß jedes Wort $uv^i wx^i y$ in L liegt, also das Wort an zwei Stellen gleichmäßig auf- bzw. abgepumpt werden kann. Es kann jedoch auch reichen an nur einer Stelle zu pumpen, was auch nicht verwundert, da die Menge der regulären Sprachen ja eine echte Teilmenge der Menge der kontextfreien Sprachen ist und bei diesen darf nur an einer Stelle gepumpt werden.

- **Was ist Ogden's Lemma?**

Ogden's Lemma ist **quasi das verallgemeinerte Pumping Lemma** für kontextfreie Sprachen. Hier kann zusätzlich noch entschieden werden, wo gepumpt werden soll.

Wenn L kontextfrei ist, gibt es eine Konstante $N \in \mathbb{N}$, so daß für alle Wörter $z \in L$ mit $|z| \geq N$ folgendes gilt: Wenn in z mindestens N Buchstaben markiert sind, läßt sich z so in $z = uvwxy$ zerlegen, daß in vx mindestens ein Buchstabe markiert ist, in vwx höchstens N Buchstaben markiert sind und für alle $i \geq 0$ gilt: $uv^iwx^iy \in L$.

Wenn man in Ogden's Lemma alle Buchstaben des Wortes markiert, erhält man das normale Pumping Lemma für kontextfreie Sprachen. Aus der Bedingung, daß in vx mindestens ein Buchstabe markiert ist, folgt dann automatisch $|vx| \geq 1$. Und da in vwx höchstens N Buchstaben markiert sein sollen, folgt ebenfalls $|vwx| \leq N$.

- **Beweise Ogden's Lemma!**

Wir gehen davon aus, daß $L(G)$ als kontextfreie Sprache in **Chomsky-Normal-form** gegeben ist. Die zugehörigen **Syntaxbäume** sind dann **binär** und nur Väter von Blättern ($w \in T$) haben outgrad 1. Weiter wählen wir $N = 2^{|V|+1}$, also genügend lang. Für z gelte $z \in L$ und $|z| \geq N$. Wie gesagt ist der zugehörige Syntaxbaum binär und hat $|z|$ Blätter, die von links nach rechts gelesen, das Wort z ergeben. Im Wort seien mindestens N Blätter markiert.

Wir wählen jetzt einen Weg von der Wurzel (S) zu einem Blatt und wählen den Weg so, daß sich mindestens zwei Variablen wiederholen. Dies machen wir, indem wir jeweils in den **Teilbaum verzweigen**, in dem sich **mehr markierte Blätter** befinden. Knoten in deren linken und rechten Teilbäumen jeweils markierte Blätter liegen, heißen **Verzweigungsknoten**. Auf unserem Weg liegen mindestens $|V| + 1$ Verzweigungsknoten. Wir werden auch nur die **letzten** $|V| + 1$ **betrachten**.

Warum liegen jedoch mindestens $|V| + 1$ Verzweigungsknoten auf unserem Weg? Wir starten am Startknoten S. Im linken und im rechten Teilbaum zusammen liegen mindestens $N = 2^{|V|+1}$ markierte Blätter. Wenn o.B.d.A. im linken Teilbaum keine markierten Blätter liegen, so ist S eben kein Verzweigungsknoten und wir gehen zur Wurzel des rechten Teilbaums. Wenn o.B.d.A. der linke Teilbaum mindestens ein markiertes Blatt enthält, so enthält der rechte Teilbaum mindestens $2^{|V|}$ markierte Blätter, wenn der rechte Teilbaum mehr markierte Blätter enthält. Unser Weg enthält also mindestens immer die Hälfte aller markierten Blätter, da wir sonst nicht richtig verzweigen würden. Also mindestens $\log_2 2^{|V|+1} = |V| + 1$ Verzweigungsknoten.

Wir betrachten, wie schon gesagt, die letzten $|V| + 1$ Verzweigungsknoten. Da es nur $|V|$ Variablen gibt, muß also eine **doppelt vorkommen**. Wir nennen diese Variable A und die beiden Verzweigungsknoten v_1 und v_2. v_1 liege über v_2. Desweiteren erzeuge v_2 das Teilwort w und v_1 das Teilwort vwx. Damit sind auch das Präfix u und das Suffix y eindeutig.

Da v_1 Verzweigungsknoten ist, enthält mindestens v oder x einen markierten Buchstaben. Da wir von v_1 bis v_2 höchstens $|V| + 1$ Verzweigungsknoten haben, sind auch höchstens $2^{|V|+1} = N$ Knoten markiert.

Wenn man sich den Syntaxbaum aufzeichnet sieht man das die Grammatik also die Regeln $S \overset{*}{\to} uSy, A \overset{*}{\to} vAx$ und $A \overset{*}{\to} w$ enthalten muß. Also gilt $S \overset{*}{\to} uwy = uv^0wx^0y \in L$ und $S \overset{*}{\to} uAy \overset{*}{\to} uvAxy \overset{*}{\to} \ldots \overset{*}{\to} uv^iAx^iy \overset{*}{\to} uv^iwx^iy \in L$.

- **Gebe Beispiele für die Anwendung des Pumping Lemmas und Ogden's Lemma an!**

$$L = \{a^ib^ic^i | i \geq 1\}$$

Hier reicht das Pumping Lemma aus. Sei N die Konstante aus dem Pumping Lemma, dann wählen wir das Wort $a^Nb^Nc^N$. Da $vwx \leq N$ gilt, können höchstens nur a's und b's oder nur b's und c's markiert sein. Also gilt für jedes Zerlegung $uvwxy$, daß für i=2 das Wort nicht in der Sprache ist.

$$L = \{a^iba^iba^i | i \geq 0\}$$

Auch hier reicht das Pumping Lemma aus. Sei N die Konstante aus dem Pumping Lemma, dann wählen wir das Wort $a^N ba^N ba^N$. Wieder können wegen $|vwx| \leq N$ nur zwei a-Gruppen ausgewählt werden und das Wort wird uneinheitlich aufgepumpt.

5.5 Effiziente Algorithmen für kontextfreie Sprachen

• **In welcher Zeit kann man die Menge nutzloser Variablen einer Grammatik G berechnen?**
Eine Variable heißt **nutzlos**, wenn aus ihr nicht mindestens **ein Wort** $w \in T^*$ **abgeleitet werden kann** oder wenn aus ihr zwar ein Wort $w \in T^*$ abgeleitet werden kann, sie jedoch **nicht von der Startvariable aus erreichbar** ist.

Im ersten Schritt berechnen wir $V' \subseteq V$, die Menge der Variablen A, für die $A \stackrel{*}{\to} w$ mit $w \in T^*$ gilt. Wenn wir die Terminalzeichen in den Regeln durch ε ersetzen, müssen wir die Menge der Variablen berechnen für die $A \stackrel{*}{\to} \varepsilon$ gilt. Dies haben wir jedoch schon bei der Überleitung einer kontextfreien Grammatik in die Chomsky-Normalform gemacht. Für diesen Schritt brauchen wir also Zeit $O(|V|s(G))$.

Nach diesem Schritt können wir schon sagen, ob $L(G)$ die **leere Sprache** ist. Dies ist nämlich der Fall, wenn $S \notin V'$ ist, sich also kein Wort $w \in T^*$ aus S ableiten läßt.

Sei also $S \in V'$. Jetzt schauen wir, ob alle Variablen aus denen sich Wörter $w \in T^*$ ableiten lassen, überhaupt von der Startvariable S aus erreichbar sind. Die Menge aller von der Startvariable aus erreichbaren Variablen heiße $V'' \subseteq V'$. Eine Variable A wird in V'' aufgenommen, wenn es eine Regel $S \to \alpha A \beta$ mit $\alpha, \beta \in (V' \cup T)^*$ gibt. Zu Beginn wird S in V'' aufgenommen. Dann schaut man sich alle Regeln $S \to \alpha A \beta$ mit $\alpha, \beta \in (V' \cup T)^*$ an und nimmt jedes A für das die Bedingung gilt in V'' auf. Mit anderen Worten: Wenn in der Ableitung von S nur Terminalzeichen aus T und Variablen aus V' vorkommen, dann nehme alle Variablen dieser Regel in V'' auf. Desweiteren kann man ein Queue Q benutzen und alle Variablen, die in V'' aufgenommen werden auch in Q aufnehmen. Dann nimmt man die nächste Variable A aus der Queue und betrachtet alle Regeln $A \to \alpha B \beta$ mit $\alpha, \beta \in (V' \cup T)^*$. Dies macht man so lange, bis die Queue leer ist. Jeder Buchstabe wird natürlich auch nur einmal in die Queue aufgenommen. Auch hier brauchen wir höchstens Zeit $O(|V|s(G))$. Also brauchen wir insgesamt $O(|V|s(G))$.

• **In welcher Zeit kann man entscheiden, ob eine durch kontextfreie Grammatik erzeugte Sprache unendlich ist?**
In linearer Zeit.

• **Gegenüber welchen Operationen ist die Klasse der kontextfreien Sprachen abgeschlossen? Beweise?**

□ **Vereinigung**:

Gegeben seien zwei Grammatiken G_1 und G_2. Durch eventuelle Umbenennung erreichen wir, daß die Variablenmengen V_1 und V_2 disjunkt sind. Dann sieht die Grammatik für die Vereinigung der beiden Sprachen wie folgt aus, wobei S eine neue Variable ist:

$$V = V_1 \cup V_2 \cup \{S\},$$
$$P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$$

Wenn ich also ein Wort aus L_1 erzeugen möchte, wähle ich als erste Produktion nichtdeterministisch $S \to S_1$, sonst $S \to S_2$.

□ **Konkatenation**:

Wieder seien zwei Grammatiken G_1 und G_2 gegeben. S ist auch hier ein neues Startsymbol:

$$V = V_1 \cup V_2 \cup \{S\}$$
$$P = P_1 \cup P_2 \cup \{S \to S_1 S_2\}$$

□ **Kleenescher Abschluß**:

Hier muß beachtet werden, daß auch das leere Wort erzeugbar ist. Die Grammatiken seien wie oben gegeben:

$$V = V_1 \cup \{S\}$$
$$P = P_1 \cup \{S \to \varepsilon, S \to SS, S \to S_1\}$$

• **Gegenüber welchen Operationen ist die Klasse der kontextfreien Sprachen nicht abgeschlossen? Beweise?**

□ **Durchschnitt**:

Es genügt als Zeugen eine nicht kontextfreie Sprache anzugeben, die der Durchschnitt zweier kontextfreier Sprachen ist. Die beiden kontextfreien Sprachen seien L_1L_2 und L_3L_4 mit:

$$L_1 = \{a^i b^i | i \geq 1\}, L_2 = \{c\}^*, L_3 = \{a\}^*, L_4 = \{b^i c^i | i \geq 1\}$$

Es gilt jedoch $L_1L_2 \cap L_3L_4 = \{a^i b^i c^i | i \geq 1\}$ und diese Sprache kann mit Hilfe des Pumping Lemmas als nicht kontextfrei nachgewiesen werden.

□ **Komplement**:

Wären die kontextfreien Sprachen gegen Komplement abgeschlossen, dann wären sie, da sie gegen Vereinigung abgeschlossen sind, nach dem Satz von de Morgan auch gegen Durchschnitt abgeschlossen, da gilt:

$$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$$

5.6 Unentscheidbare Probleme

• **Welche beiden unentscheidbaren Probleme für kontextfreie Sprachen kennst du?**

□ Ist $L(G_1) \cap L(G_2) = \emptyset$?

□ Ist die kontextfreie Grammatik G mehrdeutig?

• **Zeige, daß die Entscheidung, ob der Schnitt zweier Grammatiken leer ist, nicht rekursiv ist!**

Dazu reduzieren wir das PKP auf dieses Problem. Das heißt: Wir können das PKP lösen, wenn wir entscheiden können, ob der Schnitt zweier kontextfreier Sprachen leer ist. Dazu bauen wir eine Eingabe für das PKP in eine Eingabe für unser Schnittproblem um:

Das PKP sei gegeben durch die Eingabe $K = ((x_1, y_1), ..., (x_k, y_k))$, also aus k Paaren von Strings $x_i, y_i \in \Sigma^*$ für $1 \leq i \leq k$. Wir bauen jetzt für die x-Strings und für die y-Strings Grammatiken. In diesen Grammatiken gilt $T = \Sigma \cup \{a_1, ..., a_k\}$. Wichtig für das PKP ist, daß die Lösung für die x-Folge und die y-Folge aus der gleichen Indexfolge gebildet wird. Dazu merken wir uns über die k neuen Variablen $\{a_1, ..., a_k\}$, welche Indexfolge wir benutzt haben, um ein Wort aus der Grammatik mit Hilfe der x- bzw. y-Strings zu bilden. Die beiden Grammatiken G_1 und G_2 sehen wie folgt aus, wobei S_1 das Startsymbol für G_1 ist und S_2 für G_2:

1. Für die x-Strings:

$$S \to a_1 x_1, ..., S \to a_k x_k, S \to a_1 S x_1, S \to a_k S x_k$$

2. Für die y-Strings:

$$S \to a_1 y_1, ..., S \to a_k y_k, S \to a_1 S y_1, S \to a_k S y_k$$

Aus welchen Wörtern bestehen nun $L(G_1)$ bzw. $L(G_2)$? $L(G_1)$ besteht aus allen Wörtern $a_{i(n)}...a_{i(1)}x_{i(1)}...x_{i(n)}$ und $L(G_2)$ aus allen Wörtern $a_{i(n)}...a_{i(1)}y_{i(1)}...y_{i(n)}$.

Das PKP hat genau dann eine Lösung, wenn die Indexfolge $i(n)...i(1)$ überein-stimmt und wenn dasselbe Wort gebildet wurde. Letzteres ist der Fall, wenn $x_1...x_n = y_1...y_n$ gilt. Also gilt insgesamt:

$$a_{i(n)}...a_{i(1)}x_{i(1)}...x_{i(n)} = a_{i(n)}...a_{i(1)}y_{i(1)}...y_{i(n)}$$

Dies ist jedoch genau dann der Fall, wenn es ein Wort gibt, welches beide Grammatiken erzeugen können. Wenn also $L(G_1) \cap L(G_2) \neq \emptyset$ ist. Mit anderen Worten: Wenn ich entscheiden könnte ob der Schnitt zweier Grammatiken leer ist, könnte ich damit entscheiden, ob das PKP eine Lösung besitzt. Dann wäre das PKP jedoch rekursiv und sogar das Halteproblem könnte entschieden werden. Wenn das Halteproblem jedoch entschieden werden könnte, wäre die Klasse der rekursiv aufzählbaren Sprachen überflüssig.

• **Zeige, daß die Entscheidung, ob eine Sprache inhärent mehrdeutig ist, nicht rekursiv ist!**

Im Beweis, daß das Schnittproblem für kontextfreie Grammatiken nicht rekursiv ist, haben wir für zwei spezielle Grammatiken gezeigt, daß ihr Schnittproblem nicht rekursiv ist. Also wird das allgemeine Schnittproblem wohl auch nicht rekursiv sein, da dieses spezielle Schnittproblem ein Spezialfall des allgemeinen Problems ist.

Die zwei Grammatiken G_1 und G_2 aus diesem Beweis benutzen wir auch hier. Wir konstruieren eine neue Grammatik G, die S als neues Startsymbol erhält. Weiter behält sie alle Regeln aus G_1 und G_2. Es kommen jedoch noch zwei neue Regeln hinzu:

$$S \to S_1, S \to S_2$$

Die Grammatiken G_1 und G_2 sind eindeutig. Wann ist jedoch unsere Grammatik G eindeutig? Doch nur, wenn gilt: $L(G_1) \cap L(G_2) = \emptyset$. Denn sonst würde es ein Wort $w \in \Sigma^*$ geben, für das gilt:

$$w \in L(G_1) \wedge w \in L(G_2)$$

Dann gilt jedoch auch:

$$(S \to S_1 \overset{*}{\to} w) \wedge (S \to S_2 \overset{*}{\to} w)$$

Das heißt, es gäbe für ein Wort zwei verschiedene Syntaxbäume und die Grammatik wäre inhärent mehrdeutig. Wenn wir also das Schnittproblem lösen könnten, könnten wir auch für diese spezielle Grammatik G entscheiden, ob sie inhärent mehrdeutig ist. Da das spezielle Schnittproblem jedoch nicht rekursiv ist, kann auch dieses Problem nicht rekursiv sein.

5.7 Eine inhärent mehrdeutige Sprache

• **Was ist eine inhärent mehrdeutige Sprache?**

Eine Grammatik G heißt eindeutig, wenn es zu jedem Wort $w \in L(G)$ genau einen Syntaxbaum gibt und eine Sprache L ist dementsprechend eindeutig, wenn es eine eindeutige Grammatik G mit $L = L(G)$. Gibt es keine solche eindeutige Grammatik heißt die Sprache **inhärent mehrdeutig**. Bei der inhärenten Mehrdeutigkeit handelt es sich also um eine **Eigenschaft der Sprache**, da es zu einer inhärent mehrdeutigen Grammatik eine äquivalente Grammatik geben kann, die eindeutig ist.

Angenommen, wir haben eine eindeutige Grammatik G, die die Regel $A \to B$ enthält. Dann können wir daraus eine inhärent mehrdeutige Grammatik konstruieren, indem wir die Regel $A \to B$ streichen und durch folgende Regeln ersetzen:

$$A \to X_1, X_1 \to B$$
$$A \to X_2, X_2 \to B$$

Es gibt dann für ein Wort $w \in L(G)$, welches die Regel $A \to B$ einmal benutzt, zwei verschiedene Syntaxbäume, in denen in einem Fall die oberen beiden Regeln benutzt werden und im anderen die unteren beiden.

Wir konstruieren im folgenden eine inhärent mehrdeutige Sprache aus zwei kontextfreien Sprachen, deren Schnitt jedoch nicht kontextfrei ist. Die Sprachen sind $L' = \{a^i b^i | i \geq 0\}\{c\}^*$ und $L'' = \{a\}^*\{b^i c^i | i \geq 0\}$. Die Vereinigung dieser Sprachen ist die Sprache $L = \{a^i b^j c^k | i = j \text{ oder } j = k\}$ und der Schnitt $L_\cap = \{a^i b^i c^i | i \geq 0\}$.

Die Wörter des Schnittes $L_\cap$ lassen sich also mit Hilfe der kontextfreien Grammatiken $G(L')$ und $G(L'')$ erzeugen.

Zuerst betrachten wir das Wort $a^n b^n c^{n+n!}$ aus L'. Hierfür gibt es logischerweise einen Syntaxbaum. Wir zeigen nun mit Hilfe des Lemma von Ogden, daß es auch einen Syntaxbaum für das Wort $a^{n+n!} b^{n+n!} c^{n+n!} \in L'$ gibt und gewinnen über das Lemma von Ogden noch zusätzliche Informationen über diesen Syntaxbaum.

Wir pumpen also $a^n b^n c^{n+n!}$ mit Hilfe des Lemma von Ogden zu $a^{n+n!} b^{n+n!} c^{n+n!}$ auf. Dazu markieren wir in $a^n b^n c^{n+n!}$ alle n a's. Es ist schonmal leicht einzusehen, daß $v, x \in a^* \cup b^* \cup c^*$ gelten muß, daß also v und x nur jeweils komplett aus a, b oder c bestehen, da man sonst das Wort „kaputtpumpt". Und einer der beiden Buchstaben muß also komplett aus a's bestehen, da in vx nach dem Lemma von Ogden mindestens ein Buchstabe markiert sein muß. Wir haben also drei mögliche Fälle:

1. $x \in a^*$:

   Da v vor x steht, gilt dann auch $v \in a^*$ und es gelte $vx = a^p$ mit $1 \leq p \leq n$. Dann ist für $i = 2$ das entstehende Wort schon nicht mehr in der Sprache.

2. $x \in c^*$:

   Hier gilt zwangläufig $v \in a^+$, da ja vx wie schon gesagt, mindestens einen markierten Buchstaben enthalten muß. Wieder ist das entstehende Wort für $i = 2$ nicht mehr in der Sprache.

3. $x \in b^*$:

   Dies ist der letztmögliche Fall. Es gelte $x = b^j$ mit $1 \leq j \leq n$. Dann muß v in den a's liegen. Es gelte also weiter $v = a^p$ mit $1 \leq p \leq n$. Dann ist $uv^2wx^2y = a^{n+p} b^{n+j} c^{n+n!}$. Damit das Wort in der Sprache liegt, muß also $j = p$ sein.

Dann gibt es in $G' = G(L')$ jedoch eine Variable A für die gilt:

$$S \overset{*}{\to} uAy \overset{*}{\to} uv^k Ax^k y \overset{*}{\to} uv^k wx^k y \in L'$$

Wieviele a's man auswählt für das Lemma von Ogden, hängt von der Variable p ab. Angenommen wir haben p a's ausgewählt. Dann setzen wir $k = \frac{n!}{p} + 1$ und erhalten so das Wort:

$$uv^k wx^k y = a^{n+n!} b^{n+n!} c^{n+n!} \in L'$$

Also haben wir für L' einen Syntaxbaum gefunden, der das Wort $a^{n+n!} b^{n+n!} c^{n+n!} \in L'$ erzeugt und in dem es eine Variable $A \overset{*}{\to} v^k wx^k$ gibt. In diesem Teilbaum werden nur die Buchstaben a und b erzeugt.

Dieselben Überlegungen kann man jetzt ausgehend vom Wort $a^{n+n!} b^n c^n \in L''$ machen. Wir erhalten auch hier einen Syntaxbaum für das Wort $a^{n+n!} b^{n+n!} c^{n+n!} \in L''$. In diesem

Syntaxbaum gibt es dann eine Variable $B \overset{*}{\to} (v')^k w' (x')^k$. Und in diesem Teilbaum werden nur die Buchstaben b und c erzeugt.

Was haben wir gemacht? Wir sind praktisch von einem Superstartsymbol gestartet und haben dann zwischenzeitlich ein Wort aus L' erzeugt und haben es dann weiter zu $a^{n+n!}b^{n+n!}c^{n+n!} \in L_\cap$ abgeleitet. Dann haben wir gezeigt, daß wir genauso gut zwischenzeitlich ein Wort aus L'' ableiten können, welches wir dann zu $a^{n+n!}b^{n+n!}c^{n+n!} \in L_\cap$ komplettieren. Wir müssen nun zeigen, daß wir zweimal dasselbe gemacht hätten, wenn die Sprache eindeutig wäre. Die Gesamt-Syntaxbäume beider Schritte müßten also isomorph sein.
Da im Teilbaum mit Wurzel A jedoch nur a's und b's erzeugt wurden und im Teilbaum mit Wurzel B nur b's und c's, müssen diese beiden Variablen auf verschiedenen Wegen liegen. Dann gibt es folgende Ableitung:

$$S \overset{*}{\to} t_1 A t_2 B t_3 \text{ mit } t_1, t_2, t_3 \in T^*$$

Es gilt jedoch auch:

$$A \overset{*}{\to} v^k w x^k \text{ mit } k \geq 0, v = a^p \text{ und } x = b^p \text{ mit } p > 0$$

Für unser B gilt ähnliches:

$$B \overset{*}{\to} (v')^k w' (x')^k \text{ mit } k \geq 0, v' = b^q \text{ und } x' = c^q \text{ mit } q > 0$$

Also gilt doch insgesamt:

$$S \overset{*}{\to} t_1 a^{pk} w b^{pk} t_2 b^{qk} w' c^{qk} t_3 \in L$$

Jede Erhöhung von k um 1 läßt die Anzahl der a's um p anwachsen, die Anzahl der c's um q und die Anzahl der b's um $p+q$. Für genügend großes k fallen wir also aus der Sprache. Dies ist jedoch ein Widerspruch zur Annahme, daß alle erzeugten Wörter in L liegen.

6 Greibach-Normalform und Kellerautomaten

6.1 Greibach-Normalform

- **Wann ist eine Grammatik in Greibach-Normalform?**

Eine Grammatik ist in Greibach-Normalform, wenn alle Ableitungen der Form $A \to a\alpha$ mit $A \in V, a \in T$ und $\alpha \in V^*$ sind.

Eine Grammtik in Greibach-Normalform kann also insbesondere nicht das leere Wort erzeugen.

- **Was ist das Besondere an der Greibach-Normalform?**

Das Besondere an der Greibach-Normalform ist, daß in **jedem Ableitungsschritt** genau **ein Buchstabe** des zu erzeugenden Wortes erzeugt wird. Benutzt man bei dieser Grammatik nur Linksableitungen, so führt dies auf natürliche Weise zum nichtdeterministischen Kellerautomaten.

- **Wie formt man eine Grammatik in Greibach-Normalform um?**

Wir zeigen zunächst zwei Methoden der Umformung, die die gegebene Grammatik nicht verändern:

1. Methode 1:

   Die Regel $A \to \alpha_1 B \alpha_2$ wird durch folgende Regeln ersetzt, wenn für B die Regeln $B \to \beta_1, B \to \beta_2, ..., B \to \beta_r$ existieren:

   $$A \to \alpha_1\beta_1\alpha_2, A \to \alpha_1\beta_2\alpha_2, ..., A \to \alpha_1\beta_r\alpha_2$$

2. Methode 2:

   Regeln der Art $A \to A\alpha_1, ..., A \to A\alpha_r$, werden durch die folgenden Regeln ersetzt, wenn für A desweiteren die Regeln $A \to \beta_1, ..., A \to \beta_s$ existieren (wobei β_1 bis β_s nicht mit A beginnen):

   $$A \to \beta_1 B, ..., A \to \beta_s B$$
   $$B \to \alpha_1 B, ..., B \to \alpha_r B$$

Wir können bei der Umformung der Grammatik davon ausgehen, daß sie in Chom-sky-Normalform ist. Die Variablen numerieren wir mit $A_1, ..., A_m$, wenn es m verschiedene Variablen gibt und die n verschiedenen Terminalzeichen mit $a_1, ..., a_n$. Desweiteren brauchen wir noch die Zusatz-Variablen $B_1, ..., B_m$. Insgesamt sei die Menge $V' = \{A_1, ..., A_m, B_1, ..., B_m\}$.

Das System der Umformung sieht wie folgt aus, wobei auf einen Beweis der Korrektheit hier verzichtet wird:

Ersetze alle Regeln $A_1 \to A_1\alpha$ nach Methode 2.
Ersetze alle Regeln $A_2 \to A_1\alpha$ nach Methode 1.
Ersetze alle Regeln $A_2 \to A_2\alpha$ nach Methode 2.
Ersetze alle Regeln $A_3 \to A_1\alpha$ nach Methode 1.
Ersetze alle Regeln $A_3 \to A_2\alpha$ nach Methode 1.
Ersetze alle Regeln $A_3 \to A_3\alpha$ nach Methode 2.
$\vdots$

Methode 1 wird dabei $\sum_{i=1}^{m-1} i = \binom{m}{2}$-mal angewandt. Methode 2 wie man leicht sieht genau m-mal. Es sei nur noch angemerkt, daß die Größe der Grammatik gewaltig anwachsen kann.

6.2 Kellerautomaten

- **Was ist ein nichtdeterministischer Kellerautomat (NPDA)?**

Kontextfreie Sprachen können mehr als die reguläre Sprachen, welche von endlichen Automaten erkannt werden und weniger als kontextsensitive Grammatiken, welche von linear bandbeschränkten Turingmaschinen erkannt werden. Maschinen für kontextfreie Sprachen müssen also mehr können als endliche Automaten (ein unendlicher Speicher steht zur Verfügung) und weniger als linear bandbeschränkte Turingmaschinen (die Eingabe darf nur einmal von links nach rechts gelesen werden).

Wie wir gesehen haben, kann eine kontextfreie Grammatik so umgeformt werden, daß in jedem Ableitungsschritt genau ein Buchstabe des zu erzeugenden Wortes erzeugt wird. Wenn wir also ein Wort $w = w_1...w_n$ erzeugen wollen, wählen wir, wenn S das Startsymbol ist, nichtdeterministisch eine Ableitung $S \to w_1\alpha$ aus. Danach brauchen wir nur die erste Variable A von α betrachten und wählen wieder nichtdeterministisch eine Regel $A \to w_2\beta$ aus. Wenn $\alpha = \alpha_1...\alpha_r$ gilt ist also folgende Ableitung möglich:

$$S \to w_1\alpha \to w_1w_2\beta\alpha_2...\alpha_r$$

Variablen, welche also in einer Ableitung vorkommen, werden also stets vor alle anderen Variablen geschrieben und es kann auch immer nur auf die erste Variable zugegriffen werden. Dieser Sachverhalt erinnert stark an einen Stack und führt zum Automaten-Modell des NPDA (**N**ondeterministic **P**ush-**D**own-**A**utomata).

- **Waraus besteht ein NPDA?**

Ein NPDA besteht aus den folgenden Komponenten:

- ☐ Q, die endliche Zustandsmenge
- ☐ Σ, das endliche Eingabealphabet
- ☐ Γ, das endliche Stackalphabet
- ☐ $q_0 \in Q$, der Anfangszustand
- ☐ $Z_0 \in \Gamma$, die Initialisierung des Stack
- ☐ δ, die nichtdeterministische Überführungsfunktion, die auf $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ definiert ist. Dabei bestehen $\delta(q,a,Z)$ bzw. $\delta(q,\varepsilon,Z)$ aus einer **endlichen Menge** von Paaren aus $Q \times \Gamma^*$
- ☐ (Eventuell) $F \subseteq Q$, die Menge akzeptierender Endzustände

- **Wie sieht eine Konfiguration eines NPDA aus?**

Eine Konfiguration (q,w,α) eines NPDA bedeutet, daß der Kellerautomat sich im Zustand $q \in Q$ befindet, noch das Wort $w \in \Sigma^*$ lesen muß und der Stack den Inhalt $\alpha \in V^*$ hat. Sei $(q,w_1...w_k,Z_1...Z_m)$ eine Konfiguration eines NPDA, dann ist die Konfiguration $(q',w_2...w_k,Z'_1...Z'_nZ_2...Z_m)$ eine **Nachfolgekonfiguration**, wenn $(q',Z'_1...Z'_n) \in \delta(q,w_1,Z_1)$ ist, also der Übergang „legal“ ist.

Ist $(q',Z'_1...Z'_n) \in \delta(q,\varepsilon,Z_1)$, dann ist auch $(q',w_1...w_k,Z'_1...Z'_nZ_2...Z_m)$ eine „legale“ Nachfolgekonfiguration.

- **Welche Akzeptanzmodi gibt es bei NPDAs?**

Es gibt zwei Akzeptanzmodi bei NPDAs:

1. Akzeptanz durch den leeren Stack:

   Ein NPDA akzeptiert ein Wort $w \in \Sigma^*$, wenn es einen zulässigen Rechenweg von (q_0,w,Z_0) nach $(q,\varepsilon,\varepsilon)$, mit $q \in Q$ gibt.

2. Akzeptanz durch akzeptierende Endzustände:

   Ein NPDA akzeptiert ein Wort $w \in \Sigma^*$, wenn es einen zulässigen Rechenweg von der Anfangskonfiguration (q_0, w, Z_0) in eine Konfiguartion (q, ε, γ), mit $q \in F$ und $\gamma \in \Gamma^*$ gibt.

- **Zeige, daß die Akzeptanzmodi für NPDAs ineinander überführbar sind!**

Zu zeigen sind zwei Umformungen:

1. Akzeptierende Endzustände $\Rightarrow$ Leerer Stack:

   Sei A_1 ein PDA, der die Sprache L mit Hilfe akzeptierender Endzustände akzeptiert. Dann sieht der PDA A_2 (leerer Stack) wie folgt aus:

   Bei A_1 kann der Stack zwischenzeitlich leer werden, ohne daß A_1 ein Wort akzeptiert und A_1 kann ein Wort auch akzeptieren, ohne daß der Stack leer ist. Unser PDA A_2 muß also in der Lage sein, den Stack zu leeren, wenn das Wort zuende gelesen wurde und A_1 einen akzeptierenden Endzustand erreicht hätte. Desweiteren muß gewährleistet sein, daß der Automat A_2 nicht zu früh akzeptiert, weil der Stack zwischenzeitlich leer ist.

   Beides erreichen wir mit Hilfe des neuen Stackbuchstaben Z_0^2, der ganz am Anfang der Rechnung geschrieben wird. Die Rechnung erfolgt dann, wie bei A_1 auf dieser neuen „Basis“. Wenn der Stack bei A_1 also zwischenzeitlich leer wird, ist er bei A_2 noch nicht leer. Durch neue Regeln wird dafür gesorgt, daß A_2 seinen Stack inklusive Z_0^2 nur leeren kann, wenn ein akzeptierender Endzustand erreicht wurde.

2. Leerer Stack $\Rightarrow$ Akzeptierende Endzustände:

   Hier gehen wir ähnlich vor, wie bei 1. Wieder legen wir einen neue Basis Z_0^2. Dann simulieren wir die Rechnung des PDA, der mit dem leeren Stack akzeptiert. Ist das Wort zu Ende gelesen und nur noch Z_0^2 befindet sich auf dem Stack, so wechseln wir mit einer ε-Bewegung in einen akzeptierenden Zustand. Es gilt also $\delta_2(q, \varepsilon, Z_0^2) = \{(q_F, \varepsilon)\}$ mit $q_F \in F$.

- **Gebe einige Sprachen an, die von PDAs akzeptiert werden!**

  □ Die Sprache $L = \{w\#w^R | w \in \{0,1\}^*\}$ wird von einem DPDA erkannt. Das Wort wird, solange # noch nicht gelesen wurde, auf dem Stack gespeichert. Wird # gelesen, so steht der letzte Buchstabe von w, der der erste Buchstabe von w^R ist, ganz oben auf dem Stack und kann daher mit dem ersten Buchstaben von w^R verglichen werden. Stimmen die Buchstaben überein, werden sie gelöscht und der nächste Buchstabe wird betrachtet. Ist der Stack leer und w^R enthält keinen Buchstaben mehr, so wird akzeptiert, sonst nicht.

  □ Die Sprache $L = \{ww^R | w \in \{0,1\}^*\}$ wird von einem NPDA erraten, da dieser die Trennstelle raten kann.

- **Gibt es zu jedem NPDA einen entsprechenden DPDA?**

Nein. Für die Sprache $L = \{ww^R | w \in \{0,1\}^*\}$ gibt es zwar einen NPDA der die Sprache erkennt, jedoch keinen DPDA.