```
                   -> case-Anweisung

if-Anweisung -> IF Ausdruck THEN Ausdruck [ELSE Ausdruck]
case-Anweisung -> CASE Ausdruck OF case-Selektor/';'... [ELSE Ausdruck] [';'] END
case-Selektor -> case-Label/','... ':' Anweisung
case-Label -> Konstanter Ausdruck ['..' Konstanter Ausdruck]
Schleifenanweisung -> repeat-Anweisung

                   -> while-Anweisung
                   -> for-Anweisung

repeat-Anweisung -> REPEAT Anweisung UNTIL Ausdruck
while-Anweisung -> WHILE Ausdruck DO Anweisung
for-Anweisung -> FOR Qualifizierter Bezeichner ':=' Ausdruck (TO | DOWNTO)
                  Ausdruck DO Anweisung
with-Anweisung -> WITH Bezeichnerliste DO Anweisung
Prozedurdeklarationsabschnitt -> Prozedurdeklaration

                              -> Funktionsdeklaration

...

Klassentyp -> CLASS [Klassenvererbung]

              [Klassenfelderliste]
              [Klassenmethodenliste]
              [Klasseneigenschaftenliste]
              END

Klassenvererbung -> '(' Bezeichnerliste ')'
Klassensichtbarkeit -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]
Klassenfelderliste -> (Klassensichtbarkeit Objektfelderliste)/';'...
Klassenmethodenliste -> (Klassensichtbarkeit Methodenliste)/';'...
Klasseneigenschaftenliste -> (Klassensichtbarkeit Eigenschaftenliste ';')...
Eigenschaftenliste -> PROPERTY Bezeichner [Eigenschaftsschnittstelle]
                        Eigenschaftsbezeichner

Eigenschaftsschnittstelle -> [Eigenschaftsparameterliste] ':' Bezeichner
Eigenschaftsparameterliste -> '[' (Bezeichnerliste ':' Typbezeichner)/';'... ']'
Eigenschaftsbezeichner -> [INDEX Konstanter Ausdruck]

                          [READ Ident]
                          [WRITE Bezeichner]
                          [STORED Bezeichner | Konstante)]
                          [(DEFAULT Konstanter Ausdruck) | NODEFAULT]
                          [IMPLEMENTS Typbezeichner]

Schnittstellentyp -> INTERFACE [Schnittstellenvererbung]

                     [Klassenmethodenliste]
                     [Klasseneigenschaftenliste]
                     END

Schnittstellenvererbung -> '(' Bezeichnerliste ')'
requires-Klausel -> REQUIRES Bezeichnerliste... ';'
contains-Klausel -> CONTAINS Bezeichnerliste... ';'
Bezeichnerliste -> Bezeichner/','...
```

```
Qualifizierter Bezeichner -> [Unit-Bezeichner '.'] Bezeichner
Typbezeichner -> [Unit-Bezeichner '.'] <Typbezeichner>
Ident -> <Bezeichner>
ConstExpr -> <Konstanter Ausdruck>
UnitId -> <Unit-Bezeichner>
LabelId -> <Label-Bezeichner>

Number -> <Nummer>
String -> <String>
```

Die Erzeugungsregeln einer kontextfreien Grammatik erfüllen (auf den ersten Blick) auch die Bedingung, die man an eine kontextsensitive Grammtik stellt. In einer kontextfreien Grammatik können jedoch auch Produktionen der Form $A \to \boldsymbol{e}$ vorkommen, wobei $A \neq S$ oder zusätzlich A auf der rechten Seite einer Produktion vorkommt. In diesem Fall kann man in einem endlichen Verfahren die Regeln und nichtterminalen Symbole der Grammatik so abändern, daß keine Produktionen der Form $A \to \boldsymbol{e}$ mehr vorkommen außer wenn eventuell A das Startsymbol ist (A steht dann auf keiner rechten Seite einer Produktion). Die Änderung kann so erfolgen, daß weiterhin dieselbe Sprache erzeugt wird. Es gilt daher:

Satz 4.4-1:

Zu jeder kontextfreien Grammatik G gibt es eine kontextsensitive Grammatik G', die dieselbe Sprache erzeugt:

Die Klasse der kontextfreien Sprachen (Typ-1-Sprachen) über einem endlichen Alphabet Σ ist eine echte Teilmenge der kontextsensitiven Sprachen über Σ.

Die Tatsache, daß die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ echt in der Klasse der kontextsensitiven Sprachen über Σ enthalten ist, wird weiter unten bewiesen.

Der folgende Satz, der auch hier ohne Beweis aufgeführt wird, besagt, daß man jede kontextfreie Grammatik in eine Grammatik in **Normalform** überführen kann, deren Regeln eine einfache Struktur aufweisen.

Satz 4.4-2:

Jede kontextfreie Grammatik kann so umgeformt werden, daß alle Regeln die Form

$S \to \boldsymbol{e}$ oder

$A \to BC$ mit $A \in N$, $B \in N$, $C \in N$ oder

$A \to a$ mit $A \in N$, $a \in \Sigma$

haben (Chomsky-Normalform) und dabei dieselbe Sprache erzeugt wird.

Der folgende Satz (*uvwxy*-**Theorem, pumping lemma**) liefert ein Beweismittel, mit dessen Hilfe man zeigen kann, daß eine Sprache nicht kontextfrei ist.

Satz 4.4-3:

Zu jeder kontextfreien Sprache L gibt es eine Zahl $n_0 > 0$ mit der Eigenschaft:

jedes $z \in L$ mit $|z| \geq n_0$ läßt sich zerlegen in $z = uvwxy$ mit

(i) $|vwx| \leq n_0$

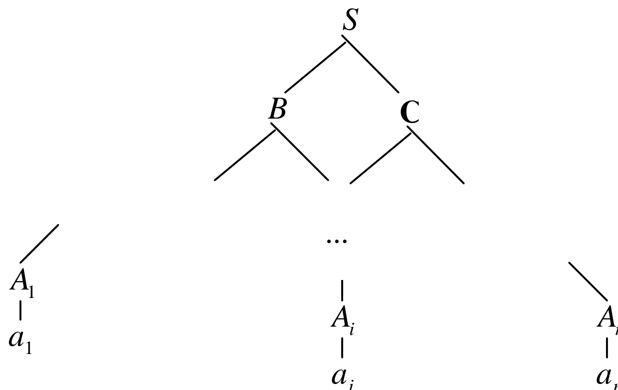
(ii) $|vx| > 0$

(iii) $uv^k wx^k y \in L$ für jedes $k \in \mathbf{N}$.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik in Chomsky-Normalform mit $L = L(G)$ und $n_0 = 2^{|N|+1}$.

Für ein Wort $z \in L$, etwa $z = a_1 \ldots a_n$ mit $a_1 \in \Sigma$, ..., $a_n \in \Sigma$ und $|z| = n \geq n_0$, hat eine Ableitung aus S die Form $S \Rightarrow BC \Rightarrow \ldots \Rightarrow a_1 \ldots a_n$. Diese Ableitung kann als Ableitungsbaum geschrieben werden, in dem jeder Knoten durch ein in der Ableitung vorkommendes Symbol markiert ist: Die Wurzel des Ableitungsbaums ist mit S markiert. Wird in der Ableitung eine Regel der Form $A \to BC$ mit $B \in N$ und $C \in N$ angewendet, so gibt es im Ableitungsbaum einen dieser Regelanwendung entsprechenden mit A markierten Knoten, dessen direkte Nachfolger mit B bzw. C markiert sind. Wird eine Regel der Form $A \to a$ mit $a \in \Sigma$ angewendet, so gibt es im Ableitungsbaum einen dieser Regelanwendung entsprechenden mit A markierten Knoten, dessen direkter Nachfolger mit a markiert ist. Da G Chomsky-Normalform hat, ist dieser Ableitungsbaum ein Binärbaum, in dem jeder innere Knoten genau zwei Nachfolger hat und der Anwendung einer Regel der Form $A \to BC$ entspricht. Die Blätter des Baums sind mit a_1, ..., a_n markiert. Ein mit a_i markiertes Blatt mit seinem mit A_i markierten Vorgänger entspricht der Anwendung der Regel $A_i \to a_i$. Nur an den Blättern werden Regeln dieser Form angewendet.



Für die Anzahl n der Blätter dieses Ableitungsbaums gilt $n = |z| \geq n_0 = 2^{|N|+1}$. Dann hat der Baum nach Satz 1.1-9 Teil 4. eine Mindesthöhe von $\lceil \log_2(n) + 1 \rceil \geq \left\lceil \log_2\left(2^{|N|+1}\right) + 1 \right\rceil = |N| + 2$.

Es gibt also einen Pfad von der Wurzel zu einem Blatt, das mit einem terminalen Zeichen a_i markiert ist, auf dem mindestens $|N|+1$ viele innere Knoten liegen, die mit nichtterminalen Symbol markiert sind. Da die Grammatik nur $|N|$ viele nichtterminale Symbole enthält, kommt auf diesem Pfad ein $A \in N$ mindestens zweimal vor. Es werde hier die Situation betrachtet, bei der dieses zum ersten Mal geschieht:
$S \Rightarrow^* w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2 \Rightarrow^* uvwxy$ mit $w_1 \Rightarrow^* u$, $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$, $w_4 \Rightarrow^* x$ und $w_2 \Rightarrow^* y$.
Da *G* kontextfrei ist, hängen diese Ableitungen nicht zusammen, man kann sie unabhängig voneinander in einer beliebigen Reihenfolge ausführen. Daher sind in *G* auch folgende Ableitungen möglich:
$S \Rightarrow^* uAy \Rightarrow^+ uw_3 A w_4 y \Rightarrow^* uvAxy \Rightarrow^* uvwxy$.
Der erste Schritt in der Teilableitung $uAy \Rightarrow^+ uw_3 A w_4 y$ erfolgt durch Anwendung einer Regel der Form $A \rightarrow BC$, daher ergibt sich
$uAy \Rightarrow uBCy \Rightarrow^* uw_3 A w_4 y$ und $BC \Rightarrow^* w_3 A w_4 \Rightarrow^* vAx$. Wären beide Teilworte *v* und *x* leer, so könnte man in *G* eine Ableitung $BC \Rightarrow^* A$ durchführen. Dieses ist nicht möglich, da *G* in Chomsky-Normalform vorliegt. Daher gilt Eigenschaft (ii).
Die Teilableitung $A \Rightarrow^* vwx$ hat höchstens $|N|$ viele Ableitungsschritte, da in *G* wegen der Chomsky-Normalform keine Ableitungen der Form $A \Rightarrow^+ B$ möglich sind. Daher gilt $|vwx| \le 2^{|N|} < n_0$ (Eigenschaft (i)).
Die in *G* mögliche Ableitung $A \Rightarrow^* vAx$ kann man beliebig oft in die Ableitung $S \Rightarrow^* uvwxy$ einbauen und mit der Ableitung $A \Rightarrow^* w$ kombinieren:
$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uv^k Ax^k y \Rightarrow^* uv^k wx^k y$ für $k \ge 1$; außerdem ist die Ableitung $S \Rightarrow^* uAy \Rightarrow^* uwy$ möglich. Insgesamt ist dieses die Eigenschaft (iii). ///

Mit Hilfe dieses Satzes läßt sich zeigen, daß die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1\}$ (aus Kapitel 4.1) nicht kontextfrei (aber kontextsensitiv ist). Es sei $n = n_0$. Für das Wort $z = a^n b^n c^n$ ist $|z| = 3n_0 \ge n_0$. Dann läßt sich *z* zerlegen in $z = uvwxy$ mit den obigen Eigenschaften (i), (ii) und (iii). Der Teil *vwx* enthält höchstens n_0 viele Zeichen und kann daher nicht gleichzeitig aus *a*'s, *b*'s und *c*'s bestehen. Eigenschaft (iii) besagt, daß auch $uv^0 wx^0 y \in L_4$ ist, d.h. $uwy \in L_4$. Es werden drei Fälle unterschieden:

1. Fall: *vwx* enthält kein Zeichen *c*. Dann ist $uvwx = a^n b^m$, $y = b^{n-m} c^n$. Da $|vx| > 0$ ist, enthält *vx* $k \ge 1$ Zeichen *a* oder *b*. Das Wort *uwy* enthält $n + m - k + n - m = 2n - k < 2n$ Zeichen *a* oder *b* und *n* Zeichen *c*. Daher ist $uwy \notin L_4$.

2. Fall: vwx enthält mindestens ein Zeichen c. Wegen $|vwx| \le n_0$ enthält es kein Zeichen a. Es ist $u = a^n b^m$, $vwx = b^{n-m} c^l$, $y = c^{n-l}$ mit $l \ge 1$. Da $|vx| > 0$ ist, enthält vx $k \ge 1$ Zeichen b oder c. Das Wort uwy enthält n Zeichen a, $m + n - m + l - k = n + l - k$ Zeichen b oder c und $n - l$ Zeichen c. Daher ist $uwy \notin L_4$.

In beiden Fällen ergibt sich ein Widerspruch. Daher ist Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1\}$ nicht kontextfrei.

Ein weiteres Beispiel ist die kontextsensitive Sprache $L = \{a^{2^i} \mid i \ge 1\}$.

Es gilt daher die in Satz 4.4-1 formulierte Aussage, daß die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ eine echte Teilmenge der Klasse der kontextsensitiven Sprachen über Σ ist.

Exemplarisch für viele interessante Entscheidungsprobleme im Zusammenhang mit kontextfreien Grammatiken und kontextfreien Sprachen sollen wieder das Wortproblem und das Leerheitsproblem, jetzt bezogen auf kontextfreie Sprachen, betrachtet werden.

Wieder wird (analog zu den Typ-0- und Typ-1-Grammatiken) ein Prädikat

$$VERIFIZIERE_G_TYP_2(w) = \begin{cases} \texttt{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-2-Grammatik darstellt} \\ \texttt{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-2-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in KG_w den Bedingungen einer kontextfreien Grammatik genügen.

Das **Wortproblem für Typ-2-Grammatiken** fragt danach, ob die Menge

$$L_{Wort_Typ-2} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE_G_TYP_2(w) = \texttt{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. In vereinfachter Darstellung wird also danach gefragt, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer kontextfreien Grammatik G über dem Alphabet Σ und eines Worts $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ gilt oder nicht. Da dieses Entscheidungsproblem für kontextsensitive Sprachen entscheidbar ist, ist es auch im kontextfreien Fall entscheidbar. Das Entscheidungsverfahren aus Kapitel 4.3 ist im Falle einer kontextfreien Grammatik jedoch zu aufwendig (exponentielles Laufzeitverhalten). In der Theorie des Compilerbaus, die sich intensiv mit der Frage beschäftigt, ob ein Wort zur Sprache einer gegebenen Grammatik gehört, wird gezeigt, daß die Frage sogar mit einem Zeitaufwand der Ordnung $O(|u|^3)$ bei gegebener Grammatik G entschieden werden kann. Auf Details soll hier verzichtet werden.

Das **Leerheitsproblem**, das für Typ-0- und Typ-1-Grammatiken nicht entscheidbar ist, fragt **für Typ-2-Grammatiken**, ob die Menge

$L_{leer_Typ-2} = \{ w \mid w \in \{0,1\}^*, VERIFIZIERE_G_TYP_2(w) = \texttt{TRUE} \text{ und } L(KG_w) = \varnothing \}$

entscheidbar ist, bzw. in vereinfachter Darstellung, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer Grammatik G entscheidet, ob $L(G) = \varnothing$ gilt oder nicht.

Der folgende (Pseudocode-) Algorithmus entscheidet bei Eingabe einer kontextfreien Grammatik die $G = (\Sigma, N, S, R)$ die Frage „$L(G) \neq \varnothing$?" . Aus diesem Algorithmus läßt sich leicht ein Entscheidungsalgorithmus für das Leerheitsproblem für Typ-2-Grammtiken gewinnen.

Eingabe: Eine kontextfreie Grammatik $G = (\Sigma, N, S, R)$

Verfahren: Aufruf der Funktion `ist_nichtleer (G)`

Ausgabe: `TRUE`, falls $L(G) \neq \varnothing$, `FALSE` sonst.

```
FUNCTION ist_nichtleer (G): BOOLEAN;
        { G = (Σ, N, S, R) }

VAR V : ...;
    W : ...;

BEGIN { ist_nichtleer }
  V := ∅;
  W := {A | A ∈ N und A → w ist eine Erzeugungsregel mit w ∈ Σ*};

  WHILE NOT (V = W) DO
    BEGIN
      V := W;
      W := {A | A ∈ N und A → w ist eine Erzeugungsregel mit w ∈ (V ∪ Σ)*} ∪ V ;
    END;

  IF S ∈ W THEN ist_nichtleer := TRUE
           ELSE ist_nichtleer := FALSE;
END   { ist_nichtleer };
```

Es läßt sich zeigen, daß die `WHILE`-Schleife höchstens *n*-mal durchlaufen wird, wenn *G n* Erzeugungsregeln enthält. Daher bricht das Verfahren ab. Die Korrektheit des Verfahrens ergibt sich aus der Behauptung

> Ein nichtterminales Symbol $A \in N$ wird im *i*-ten Durchlauf der `WHILE`-Schleife genau dann in `W` aufgenommen, wenn es ein $u \in \Sigma^*$ gibt mit $A \Rightarrow^* u$.

Zusammenfassend ergibt sich

Satz 4.4-4:

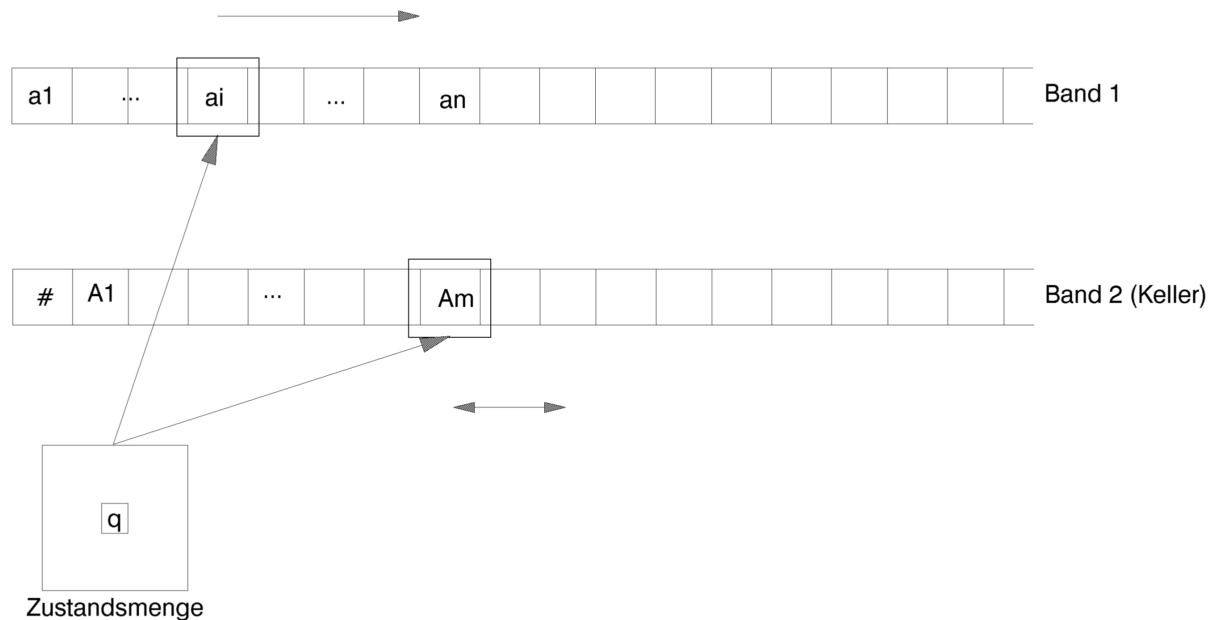
Das Wortproblem und das Leerheitsproblem für Typ-2-Grammatiken sind entscheidbar:

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-2-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-2-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

Auch zu den kontextfreien Sprachen gibt es ein Berechnungsmodell, das genau diese Sprachen erkennt und als eine Einschränkung einer Turingmaschine gesehen werden.

Ein **nichtdeterministischer Kellerautomat (NKA)** ist eine 2-NDTM, die ihre beiden Bänder in einer speziellen Weise nutzt. Vor der Angabe einer formalen Definition eines nichtdeterministischen Kellerautomaten wird das Modell informell beschrieben. Das 1. Band (Eingabeband) eines nichtdeterministischen Kellerautomaten *NKA* kann nur gelesen werden, wobei der Lesekopf nach dem Lesen eines Zeichens um eine Zelle nach rechts rückt. Das 1. Band enthält bei Start von *NKA* ein Wort über einem Eingabealphabet, der Lesekopf steht über dem ersten Zeichen des Eingabeworts. *NKA* kann auch tätig werden, wenn das Eingabeband eine leere Zeichenkette enthält; dann findet ein „spontaner (Konfigurations-) Übergang“ statt. Er kann auch Zustandsänderungen durchführen, ohne ein Symbol des Eingabebands zu lesen; man nennt diese Konfigurationsübergänge $\boldsymbol{e}$ -Überführungen. Das 2. Band heißt **Keller**; auf dem Keller bewegt sich ein Schreib/Lesekopf. Der Keller enthält zu Beginn der Berechnungsfolge von *NKA* ein spezielles Symbol #, und der Schreib/Lesekopf steht über diesem Symbol. Bei jeder Konfigurationsänderung wird das Symbol unter dem Schreib/Lesekopf durch ein endlich langes Wort ersetzt, das mit Hilfe eines Kelleralphabets gebildet wird, und der Schreib/Lesekopf rückt über das letzte Zeichen der nun im Keller befindlichen Zeichenkette. Es kann auch das leere Wort in den Keller geschrieben werden; dann wird der Inhalt des Kellers verkürzt. Auf diese Weise kann immer nur auf das zuletzt in den Keller gebrachte Zeichen zugegriffen werden. Der Schreib/Lesekopf im Keller rückt nicht über das Ende des Kellers auf Zeichen, die weiter links stehen (last-in-first-out-Prinzip).



Eine formale Definition eines nichtdeterministischen Kellerautomaten wandelt die Definition einer nichtdeterministischen Turingmaschine ab. Die Definition einer Konfiguration und des Konfigurationsübergangs wird der besonderen Arbeitsweise eines Kellerautomaten angepaßt. In der Literatur finden sich häufig Varianten der hier gegebenen Definitionen, die aber auf dieselbe Sprachklasse führen.

Ein **nichtdeterministischer Kellerautomat** *NKA* ist definiert durch
$NKA = (Q, \Sigma, \Gamma, \boldsymbol{d}, q_0, \#, F)$ mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Eingabealphabet**
3. Γ ist eine endliche nichtleere Menge: das **Kelleralphabet**
4. $\boldsymbol{d} : Q \times (\Sigma \cup \{\boldsymbol{e}\}) \times \Gamma \rightarrow \mathbf{P}_e(Q \times \Gamma^*)$ ist eine partielle Funktion, die **Überführungsfunktion**; hierbei bezeichnet $\mathbf{P}_e(Q \times \Gamma^*)$ die Menge aller endlichen Teilmengen von $Q \times \Gamma^*$
5. $q_0 \in Q$ ist der **Anfangszustand** (oder **Startzustand)**
6. $\# \in \Gamma$ ist das **Startsymbol im Keller**
7. $F \subseteq Q$ ist die **Menge der Endzustände**.

In jedem Schritt eines nichtdeterministischen Kellerautomaten ist klar, wo sich der Kopf des jeweiligen Bandes befindet: Wenn das Eingabewort w die Form $w = w_1 w_2$ hat und der Kellerautomat bereits die Zeichen in w_1 gelesen hat, dann steht der Lesekopf des 1. Bandes über

dem ersten Zeichen von w_2, und auf die Zeichen von w_1 kann der Kopf nicht mehr zugreifen. Anstelle daher in einer Konfiguration für das 1. Band den gesamten Bandinhalt und die Position des Kopfes in der Form $(w_1 w_2, |w_1| + 1)$ festzuhalten, wird der Inhalt des 1. Bandes nur durch die Zeichenkette w_2 beschrieben; implizit wird angenommen, daß der Kopf über dem ersten Zeichen von w_2 steht. Entsprechend braucht man in einer Konfiguration für das 2. Band nicht den gesamten Bandinhalt und die Position des Kopfes in der Form $(\alpha, |\alpha|)$ festzuhalten, sondern es genügt die Zeichenkette α zusammen mit der impliziten Annahme, daß der Kopf über dem zuletzt in den Keller geschriebenen Zeichen steht. Daher wird **eine Konfiguration für einen nichtdeterministischen Kellerautomaten** in der Form (q, w, α) mit $q \in Q$, $w \in \Sigma^*$ und $\alpha \in \Gamma^*$ notiert. Die Konfiguration drückt aus, daß sich der Kellerautomat gegenwärtig im Zustand q befindet, daß das Eingabeband eine Zeichenkette $w_1 w$ enthält, von der bisher die Zeichen in $w = a_1 \dots a_n$ noch nicht gelesen wurden, daß der Lesekopf des 1. Bandes über dem ersten Zeichen a_1 steht, daß sich im Keller die Zeichenkette $\alpha = A_1 \dots A_m$ befindet, und daß der Schreib/Lesekopf des 2. Bandes über A_m steht. Falls $\delta(q, a_1, A_m)$ definiert ist (das ist eine endliche Teilmenge von $Q \times \Gamma^*$) und ein Element $(q', B_1 \dots B_k)$ enthält, dann kann der Kellerautomat in eine **Nachfolgekonfiguration** übergehen, die den Zustand q' enthält, den Lesekopf auf dem 1. Band um eine Position nach rechts verrückt, im Keller das Zeichen A_m durch $B_1 \dots B_k$ ersetzt und den Schreiblesekopf im Keller auf B_k positioniert hat. Formal wird für Konfigurationen die Übergangsrelation $\Rightarrow$ definiert durch

$$(q, a_1 \dots a_n, A_1 \dots A_m) \Rightarrow \begin{cases} (q', a_2 \dots a_n, A_1 \dots A_{m-1} B_1 \dots B_k) & \text{falls } (q', B_1 \dots B_k) \in \delta(q, a_1, A_m) \\ (q', a_1 a_2 \dots a_n, A_1 \dots A_{m-1} B_1 \dots B_k) & \text{falls } (q', B_1 \dots B_k) \in \delta(q, \varepsilon, A_m) \end{cases}$$

Entsprechend bedeutet für Konfigurationen K und K' die Beziehung $K \Rightarrow^* K'$:
Es gibt Konfigurationen K_0, K_1, ..., K_l mit $l \geq 0$ und $K = K_0$, $K' = K_l$ und $K_i \Rightarrow K_{i+1}$ für $i = 0, \dots, l-1$.

Eine **Anfangskonfiguration für einen nichtdeterministischen Kellerautomaten** *NKA* hat die Form $(q_0, w, \#)$, eine **Endkonfiguration** hat die Form $(q, \varepsilon, \varepsilon)$ mit $q \in F$.

Ein Wort $w \in \Sigma^*$ wird von *NKA* **akzeptiert**, wenn $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \varepsilon)$ mit $q \in F$ gilt. Die von *NKA* **akzeptierte Sprache** ist $L(NKA) = \{ w \mid w \in \Sigma^*, \text{und } w \text{ wird von } NKA \text{ akzeptiert} \}$.

Beispiel:

Ein nichtdeterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{ w \mid w = a_1 ... a_m a_m ... a_1 \text{ , } a_i \in \{ a, b \} \}$$

Die Sprache $L = \{ w \mid w = a_1 ... a_m a_m ... a_1 \text{ , } a_i \in \{ a, b \} \}$ läßt sich beispielsweise durch die kontextfreie Grammatik $G = (\{ a, b \}, \{ S \}, S, \{ S \to aSa, S \to bSb, S \to aa, S \to bb \})$ erzeugen.

Der nichtdeterministische Kellerautomaten *NKA* wird gegeben durch $NKA = (\{ q_0, q_1, q_2 \}, \{ a, b \}, \{ A, B, \# \}, \boldsymbol{d}, q_0, \#, \{ q_2 \})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion $\boldsymbol{d}$:

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	X mit $X \in \{A, B, \#\}$	q_0	XA
	a	X mit $X \in \{A, B, \#\}$	q_1	XA
	b	X mit $X \in \{A, B, \#\}$	q_0	XB
	b	X mit $X \in \{A, B, \#\}$	q_1	XB
q_1	a	A	q_1	$\boldsymbol{e}$
	b	B	q_1	$\boldsymbol{e}$
	$\boldsymbol{e}$	$\#$	q_2	$\boldsymbol{e}$

Der Nichtdeterminismus wird in diesem Beispiel eingesetzt, um die Mitte eines Eingabewortes „zu raten".

Auch im Modell des Kellerautomaten kann man Nichtdeterminismus und Determinismus unterscheiden:

Ein **deterministischer Kellerautomat** *DKA* ist wie ein nichtdeterministischer Kellerautomat definiert mit dem Unterschied, daß die Überführungsfunktion die Form

$$\boldsymbol{d} : Q \times (\Sigma \cup \{\boldsymbol{e}\}) \times \Gamma \to Q \times \Gamma^*$$

aufweist.

Beispiel:

Ein deterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{ a^n b^n \mid n \in \mathbf{N} \}$$

Die Sprache $L = \{ a^n b^n \mid n \in \mathbf{N} \}$ läßt sich beispielsweise durch die kontextfreie Grammatik $G = (\{ a, b \}, \{ S \}, S, \{ S \to aSb, S \to \boldsymbol{e} \})$ erzeugen.

Der deterministische Kellerautomaten *DKA* wird gegeben durch $DKA = (\{ q_0, q_1, q_2 \}, \{ a, b \}, \{ a, b, \# \}, \boldsymbol{d}, q_0, \#, \{ q_0 \})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion $\boldsymbol{d}$:

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	$\#$	q_1	$\#a$
	$\boldsymbol{e}$	$\#$	q_0	$\boldsymbol{e}$
q_1	a	a	q_1	aa
	b	a	q_2	$\boldsymbol{e}$
q_2	b	A	q_2	$\boldsymbol{e}$
	$\boldsymbol{e}$	$\#$	q_0	$\boldsymbol{e}$

Satz 4.4-5:

Die Klasse der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der kontextfreien Sprachen (Typ-2-Sprachen) über Σ identisch.

Beweis:

Es ist zu zeigen, daß es zu jeder von einem nichtdeterministischen Kellerautomaten *NKA* akzeptierten Sprache $L = L(NKA)$ eine kontextfreie Grammatik G_{NKA} gibt mit $L = L(G_{NKA})$. Die umgekehrte Richtung, nämlich der Nachweis, daß es zu jeder von einer kontextfreien Grammatik *G* erzeugten Sprache $L' = L(G)$ einen nichtdeterministischen Kellerautomaten NKA_G mit $L(NKA_G) = L'$ gibt, ist von größerer praktischer Relevanz, da hierdurch implizit gezeigt wird, wie man zu einer vorgegebenen Grammatik einen Algorithmus (hier in Form eines

Kellerautomaten) entwerfen kann, der die Wörter der von der Grammatik erzeugten Sprache erkennt. Diese Beweisrichtung soll daher skizziert werden.

Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik. Ein nichtdeterministischer Kellerautomat $NKA_G = (Q, \Sigma, \Gamma, \boldsymbol{d}, q_0, \#, F)$ mit $L(NKA_G) = L(G)$ wird gegeben durch die Zustandsmenge $Q = \{ z \}$, das Kelleralphabet $\Gamma = N \cup \Sigma$, den Anfangszustand $q_0 = z$, das Startsymbol im Keller $\# = S$, die Menge der Endzustände $F = \{ z \}$ und die Überführungsfunktion $\boldsymbol{d}$, die folgendermaßen definiert ist:

(i) Für jede Erzeugungsregel $A \to \boldsymbol{a}$ aus R wird $(z, \boldsymbol{a}^R)$ in $\boldsymbol{d}(z, \boldsymbol{e}, A)$ aufgenommen; hierbei ist $\boldsymbol{a}^R$ die Konkatination der Buchstaben von $\boldsymbol{a}$ in umgekehrter Reihenfolge (Spiegelung von $\boldsymbol{a}$)

(ii) Für jedes $a \in \Sigma$ wird $(z, \boldsymbol{e})$ in $\boldsymbol{d}(z, a, a)$ aufgenommen.

Wegen (i) kann immer dann, wenn das im Keller gelesene Symbol ein nichtterminales Zeichen ist, dieses durch die rechte Seite einer Regel (in gespiegelter Reihenfolge) ersetzt werden. Ein Terminalzeichen, das gerade im Keller gelesen wird, wird entfernt, wenn es mit dem nächsten Eingabesymbol übereinstimmt. Man kann $L(NKA_G) = L(G)$ zeigen. ///

Beispiel:

Ein nichtdeterministischer Kellerautomat für eine kontextfreie Grammatik

Gegeben sei die Grammatik $G = (\Sigma, N, S, R)$ mit $\Sigma = \{ a, b, c \}$, $N = \{ S, A \}$ und der Produktionsmenge $R = \{ S \to A, A \to aAb, A \to c \}$. Die in der Konstruktion beschriebene Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
z	$\boldsymbol{e}$	S	z	A
	$\boldsymbol{e}$	A	z	bAa
	$\boldsymbol{e}$	A	z	c
	A	a	z	$\boldsymbol{e}$
	B	b	z	$\boldsymbol{e}$
	C	c	z	$\boldsymbol{e}$

Das Wort $aacbb \in L(G)$ wird durch folgenden Konfigurationsübergänge akzeptiert:

$$(z, aacbb, S) \Rightarrow (z, aacbb, A) \Rightarrow (z, aacbb, bAa) \Rightarrow (z, acbb, bA) \Rightarrow (z, acbb, bbAa)$$
$$\Rightarrow (z, cbb, bbA) \Rightarrow (z, cbb, bbc) \Rightarrow (z, bb, bb) \Rightarrow (z, b, b)$$
$$\Rightarrow (z, \boldsymbol{e}, \boldsymbol{e}).$$

Jede von einem deterministischen Kellerautomaten akzeptierte Sprache wird auch von einem nichtdeterministischen Kellerautomaten akzeptiert. Umgekehrt gibt es Sprachen, etwa $L = \{ w \mid w = a_1 ... a_m a_m ... a_1 , a_i \in \{a, b\}\}$, die nicht von einem deterministischen Kellerautomaten akzeptiert werden können. Insbesondere weisen die Klassen der deterministisch kontextfreien und die Klasse der nichtdeterministisch kontextfreien Sprachen unterschiedliche Abschlußeigenschaften auf (siehe Zusammenstellung in Kapitel 4.6).

Satz 4.4-5:

Die Klasse der deterministisch kontextfreien Sprachen über einem endlichen Alphabet Σ ist eine echte Teilmenge der Klasse der nichtdeterministisch kontextfreien Sprachen Σ.

Die Klasse der deterministisch kontextfreien Sprachen spielt eine besondere Rolle im Compilerbau. Eine derartige Sprache erlaubt die Syntaxanalyse eines Wortes (in der Anwendung ist dieses ein Programm in einer Programmiersprache), indem nur wenige Zeichen des Wortes während des Analysevorgangs im Vorgriff gelesen werden, um festzustellen, welche Produktion in einer Ableitung als nächstes anzuwenden ist bzw. daß das Wort nicht zu der von der Grammatik erzeugten Sprache gehört. Für deterministisch kontextfreie Sprachen ist das Wortproblem für ein Wort u mit einem Algorithmus entscheidbar, der eine Zeitkomplexität der Ordnung $O(|u|)$ aufweist. Die meisten Programmiersprachen sind weitgehend deterministisch kontextfrei, so daß in der Praxis die Syntaxanalyse bei Programmiersprachen sehr effizient abläuft.

4.5 Typ-3-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der die Erzeugungsregeln $R \subseteq \left\{ A \to w \mid A \in N \text{ und } w \in (N \cup \Sigma)^* \right\}$ folgende zusätzliche Bedingung erfüllen:
Alle Regeln haben die Form $A \to aB$ mit $A \in N$, $B \in N$, $a \in \Sigma$ oder $A \to \boldsymbol{e}$.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-3-Sprache** oder **rechtslineare Sprache** oder **reguläre Sprache**. Die zugehörige Grammatik heißt **Typ-3-Grammatik** oder **rechtslineare Grammatik**.

Beispielsweise ist die Sprache $L_1 = \left\{ a^i b^j \mid i \in \mathbf{N}, j \in \mathbf{N} \right\}$ aus Kapitel 4.1 regulär (rechtslinear, Typ-3-Sprache), da sie durch die Grammatik

$G_1' = (\{a,b\}, \{S,B\}, S, \{S \to \boldsymbol{e}, S \to aS, S \to bB, B \to bB, B \to \boldsymbol{e}\})$ erzeugt wird.

Der Name „rechtslineare Sprache" erklärt sich durch die Form der Erzeugungsregeln $A \to aB$ der zugehörigen Grammatik: Betrachtet man die Nichtterminalsymbole als „Variablen" (im Sinne eines Gleichungssystems) und die Terminalsymbole als „Konstanten", so liegt in der Form $A \to aB$ eine lineare Beziehung zwischen den Variablen vor, in der die Variablen rechts der Konstanten stehen.

Der Name „reguläre Sprache" weist auf eine alternative Möglichkeit hin, um eine Typ-3-Sprache zu definieren: eine derartige Sprache läßt sich durch einen „regulären Ausdruck" repräsentieren. Der Vollständigkeit soll hier die Definition eines regulären Ausdrucks und der durch ihn repräsentierten Sprache angeführt werden, wenn auch im folgenden auf dieses Konzept nicht weiter eingegangen wird.

Ein **regulärer Ausdruck über** Σ und **die durch ihn repräsentierte Sprache** wird rekursiv definiert durch:

(i) $\varnothing$ ist ein regulärer Ausdruck, der die reguläre Sprache $\varnothing$ repräsentiert

(ii) $\boldsymbol{e}$ ist ein regulärer Ausdruck, der die reguläre Sprache $\{\boldsymbol{e}\}$ repräsentiert

(iii) a aus Σ ist ein regulärer Ausdruck, der die reguläre Sprache $\{a\}$ repräsentiert

(iv) Sind p bzw. q reguläre Ausdrücke, die die regulären Sprachen P bzw. Q repräsentieren, dann ist
$(p+q)$ ein regulärer Ausdruck, der die reguläre Sprache $P \cup Q$ repräsentiert,
(pq) ein regulärer Ausdruck, der die reguläre Sprache $P \cdot Q$ repräsentiert,
$(p)^*$ ein regulärer Ausdruck, der die reguläre Sprache P^* repräsentiert

(v) Ein regulärer Ausdruck und die von ihm repräsentierte Sprache wird genau durch die Punkte (i) bis (iv) definiert.

Folgender Satz läßt sich beweisen:

Satz 4.5-1:
Eine Sprache L über einem endlichen Alphabet Σ wird genau dann durch einen regulären Ausdruck repräsentiert, wenn es eine Typ-3-Grammatik (rechtslineare Grammatik) $G = (\Sigma, N, S, R)$ mit $L = L(G)$ gibt.

Im folgenden werden daher die Begriffe „Typ-3-Sprache", „rechtslineare Sprache" und „reguläre Sprache" synonym verwendet.

Auch für die regulären Sprachen läßt sich ein Berechnungsmodell zur Akzeptanz gerade dieses Typs angeben. Man erhält es durch Einschränkung eines Kellerautomaten. Der Keller ei-

nes Kellerautomaten erlaubt die Zwischenablage von (evtl. transformierten) Eingabezeichen. Entfernt man den Keller, so kommt man auf das folgende Modell:

Ein **nichtdeterministischer endlicher Automat** *NEA* ist definiert durch
$NEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Eingabealphabet**
3. $\boldsymbol{d} : Q \times \Sigma \to \mathbf{P}(Q)$ ist eine partielle Funktion, die **Überführungsfunktion**; hierbei bezeichnet $\mathbf{P}(Q)$ die Menge aller Teilmengen von Q.
4. $q_0 \in Q$ ist der **Anfangszustand** (oder **Startzustand)**
5. $F \subseteq Q$ ist die **Menge der Endzustände**.

Entsprechend liegt ein **deterministischer endlicher Automat** *DEA* vor, wenn die Überführungsfunktion $\boldsymbol{d}$ die Form $\boldsymbol{d} : Q \times \Sigma \to Q$ aufweist.

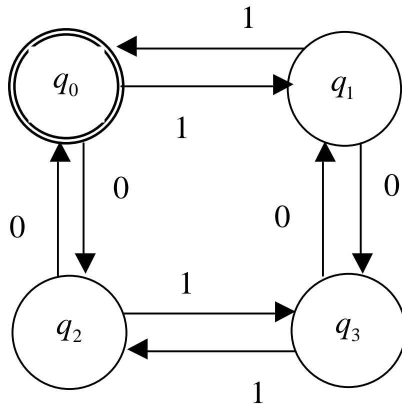
Ein deterministischer bzw. nichtdeterministischer endlicher Automat läßt sich auch in Form eines endlichen gerichteten Graphen G mit markierten Kanten als **Transitionsdiagramm** darstellen. Die Knotenmenge von G ist Q. Ist $z' = \boldsymbol{d}(z, a)$ bzw. im nichtdeterministischen Fall $z' \in \boldsymbol{d}(z, a)$, so wird eine mit a markierte Kante in G aufgenommen. Die Menge der den Endzuständen entsprechenden Knoten werden explizit markiert.

Beispiel:

Es sei $DEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ der deterministische endliche Automat mit $Q = \{ q_0, q_1, q_2, q_3 \}$, $\Sigma = \{ 0, 1 \}$, $F = \{ q_0 \}$, $\boldsymbol{d}$ gemäß folgender Tabelle:

momentaner Zustand	gelesenes Symbol	neuer Zustand
q_0	0	q_2
	1	q_1
q_1	0	q_3
	1	q_0
q_2	0	q_0
	1	q_3
q_3	0	q_1
	1	q_2

Das entsprechende Transitionsdiagramm ist



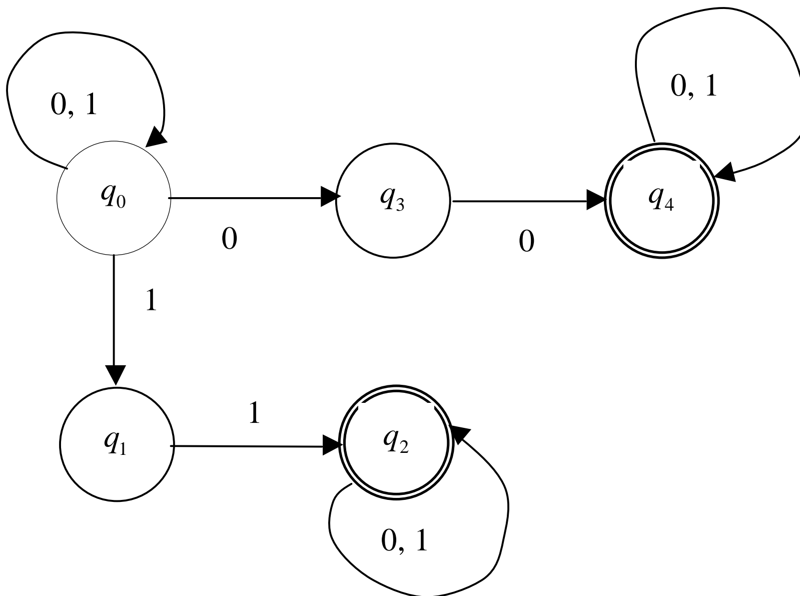
Es ist
$L(DEA) = \{ w \mid w \in \{0,1\}^* \text{ und } w \text{ enthält eine gerade Anzahl von Zeichen 0 und Zeichen 1} \}$.

Ein Beispiel eines nichtdeterministischen endlichen Automaten mit dem zugehörigen Transitionsdiagramm ist das folgende:

$NEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ mit $Q = \{ q_0, q_1, q_2, q_3, q_4 \}$, $\Sigma = \{0,1\}$, $F = \{ q_2, q_4 \}$, $\boldsymbol{d}$ gemäß folgender Tabelle:

momentaner Zustand	gelesenes Symbol	neuer Zustand
q_0	0	q_0
		q_3
	1	q_0
		q_1
q_1	1	q_2
q_2	0	q_2
	1	q_2
q_3	0	q_4
q_4	0	q_4
	1	q_4

Das zugehörige Transitionsdiagramm lautet



Es ist

$$L(NEA) = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^+ \\ \text{und } w \text{ enthält mindestens zwei aufeinanderfolgende Zeichen 0 oder Zeichen 1} \end{array} \right\}.$$

Wie bei einem Kellerautomaten durch Weglassen des Kellers kann man auch hier Konfigurationen und Konfigurationsübergänge definieren. Ein Wort $w \in \Sigma^*$ wird von *NEA* (*DEA*) **akzeptiert**, wenn $(q_0, w) \Rightarrow^* (q, \boldsymbol{e})$ mit $q \in F$ gilt. Die von *NEA* (*DEA*) **akzeptierte Sprache** ist $L(NEA) = \{ w \mid w \in \Sigma^*, \text{und } w \text{ wird von } NEA \text{ akzeptiert} \}$ (entsprechend für den deterministischen Fall).

Bei Turingmaschinen fallen Determinismus und Nichtdeterminismus zusammen, jedoch zum Preis einer exponentiell wachsenden Berechnungs- bzw. Laufzeitkomplexität. Der folgende Satz zeigt, daß auch bei den regulären Sprachen Nichtdeterminismus keine zusätzliche Berechnungsfähigkeit gegenüber dem Determinismus bringt. In den dazwischenliegenden Sprachklassen der kontextfreien und kontextsensitiven Sprachen ist der Determinismus vom Nichtdeterminismus zu unterscheiden bzw. der Unterschied zwischen Determinismus und Nichtdeterminismus ist nicht bekannt.

Satz 4.5-2:
Für jede von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache über einem endlichen Alphabet Σ gibt es einen deterministischen endlichen Automaten, der die Sprache akzeptiert.

Beweis:

Es ist bei gegebenem nichtdetermistischen endlichen Automaten $NEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ ein determistischer endlicher Automat $DEA = (Q', \Sigma, \boldsymbol{d}', q_0', F')$ anzugeben, der dieselbe Sprache akzeptiert. *DEA* wird definiert durch:
$Q' = \mathbf{P}(Q)$, $q_0' = \{q_0\}$, $\boldsymbol{d}'(Z', a) = \bigcup_{z \in Z'} \boldsymbol{d}(z, a)$ für $Z' \subseteq Q$ und $a \in \Sigma$,
$F' = \{ Z' \mid Z' \subseteq Q \text{ und } Z' \cap F \neq \emptyset \}$. ///

Der folgende Satz besagt, daß das Konzept der endlichen erkennenden Automaten das adäquate Modell für die regulären Mengen ist.

Satz 4.5-3:
Die Klasse der von (nichtdeterministischen bzw. deterministischen) endlichen Automaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der regulären Sprachen (Typ-3-Sprachen) über Σ identisch.

Beweis:

Es sei $NEA = (Q', \Sigma, \boldsymbol{d}', q_0', F')$ ein nichtdeterministischer endlicher Automat. Dann gibt es nach Satz 4.5-2 einen deterministischen endlichen Automaten $DEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ mit $L(DEA) = L(NEA)$. Es wird eine rechtslineare Grammatik $G = (\Sigma, N, S, R)$ angegeben, für die $L(G) = L(DEA)$ gilt:
$N = Q$, $S = q_0$, die Menge *R* der Regeln wird definiert durch:
R enthält genau dann eine Erzeugungsregel $q \to aq'$ mit $q \in Q$, $q' \in Q$ und $a \in \Sigma$, wenn $\boldsymbol{d}(q, a) = q'$ ist; für $q \in F$ enthält *R* außerdem die Regel $q \to \boldsymbol{e}$.

Es sei umgekehrt $G = (\Sigma, N, S, R)$ eine rechtslineare Grammatik. Dann erkennt der folgende Automat $NEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ die Sprache $L(G)$:
Es sei *A* ein nichtterminales Symbol mit $A \notin N \cup \Sigma$. Es ist

$Q = N \cup \{A\}$, $\quad q_0 = S$, $\quad F = \begin{cases} \{A\} & \text{für } \boldsymbol{e} \notin L(G) \\ \{S, A\} & \text{für } \boldsymbol{e} \in L(G) \end{cases}$, die Überführungsfunktion $\boldsymbol{d} : Q \times (\Sigma \cup \{\boldsymbol{e}\}) \to \mathbf{P}(Q)$ wird definiert durch:

$\boldsymbol{d}(B, a)$ enthält genau dann *C*, wenn *R* die Regel $B \to aC$ enthält; für jede Regel der Form $B \to \boldsymbol{e}$ wird $\boldsymbol{d}(B, \boldsymbol{e}) = \{A\}$ gesetzt. Der Automat $NEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ erfüllt noch nicht die Definition eines endlichen Automaten; denn diese läßt keine $\boldsymbol{e}$-Übergänge zu, d.h. ein endlicher Automat macht keine Überführungen, in denen kein Eingabesymbol gelesen wird. Der hier definierte Automat läßt jedoch eventuell derartige Überführungen zu. Es läßt sich jedoch zeigen (siehe angegebene Literatur), daß man *NEA* so zu einem nichtdeterministischen endlichen Automaten $NEA' = (Q', \Sigma, \boldsymbol{d}', q_0', F')$ modifizieren kann, so daß dessen Überführungsfunktion $\boldsymbol{d}' : Q' \times \Sigma \to \mathbf{P}(Q')$ die Definition eines nichtdeterministischen endlichen Automaten erfüllt und $L(NEA') = L(G)$ gilt. ///

Auch für reguläre Sprachen gibt es einen dem *uvwxy*-Theorem entsprechenden Satz, der wieder ein Beweismittel liefert, mit dessen Hilfe man zeigen kann, daß eine Sprache nicht regulär ist:

Satz 4.5-4:

Zu jeder regulären (Typ-3-, rechtslinearen) Sprache *L* gibt es eine Zahl $n_0 > 0$ mit der Eigenschaft:

jedes $z \in L$ mit $|z| \geq n_0$ läßt sich zerlegen in $z = uvw$ mit

(i) $|uv| \leq n_0$

(ii) $|v| > 0$

(iii) $uv^k w \in L$ für jedes $k \in \mathbf{N}$.

Beweis:

Die Argumentation kann ähnlich über die Ableitung eines Wortes mit Hilfe einer regulären Grammatik verlaufen wie im Beweis des *uvwxy*-Theorems. Einfacher geht es hier über die Erkennung eines Wortes mit Hilfe eines deterministischen endlichen Automaten:

Es sei regulären Sprache *L* und *DEA* ein deterministischer endlicher Automat mit $L(DEA) = L$. *DEA* enthalte n_0 viele Zustände. Bei der Akzeptanz eines Wortes $z \in L$ mit $|z| \geq n_0$ durchläuft *DEA* einschließlich des Anfangszustands $|z| + 1$ viele Zustände. Hierbei geht wesentlich ein, daß *DEA* bei jeder Überführung ein Eingabesymbol liest. In der Folge der Zustandsüberführungen vom Anfangszustand q_0 bis zu einem akzeptierenden Zustand $q_{accept} \in F$ muß sich also ein Zustand q_i wiederholen. Es sei *u* das Anfangsteilwort von *z*, das

in den Überführungen gelesen wurde, die *DEA* von q_0 aus bis zum ersten Erreichen von q_i gemacht hat. Das Teilwort v von z besteht aus den Symbolen, die *DEA* dann von q_i aus bis zum nächsten Erreichen von q_i gelesen hat. Schließlich ist w das Teilwort von z, das *DEA* liest, bis der akzeptierende Zustand q_{accept} erreicht ist. Mit dieser Zerlegung von z gelten die angegebenen drei Eigenschaften. ///

Mit Hilfe dieses Satzes läßt sich zeigen, daß die Menge der Palindrome $L = \left\{ w \mid w \in \{0,1\}^* \text{ und } w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \right\} \cup \{\boldsymbol{e}\}$ nicht regulär ist; L ist jedoch kontextfrei. Es sei $n_0 > 0$ der Wert aus obigem Satz und $n \geq n_0$ die kleinste gerade Zahl $\geq n_0$. Das Wort $z = \underbrace{1010...10}_{n \text{ Zeichen}}\underbrace{0101...01}_{n \text{ Zeichen}}$ liegt in L und läßt sich in der Form $z = uvw$ mit den Eigenschaften (i), (ii) und (iii) zerlegen. Zu beachten ist, daß die einzige Stelle, an der zwei gleiche Zeichen (00) aufeinanderfolgen, in der Mitte von z liegt. Das Teilwort uv ist Teil von $\underbrace{1010...10}_{n \text{ Zeichen}}$.

1. Fall: $v = 1$: Das Wort uv^2w ist in L und enthält zwei aufeinanderfolgende Zeichen 1. Damit uv^2w ein Palindrom ist müßten diese beiden Zeichen 1 jedoch in der Mitte des Worts liegen, da weiter rechts und links keine aufeinanderfolgenden Zeichen 1 stehen. Rechts der beiden Zeichen 1 gibt es noch die beiden Zeichen 0 (die ursprüngliche Mitte des Worts). Zu ihnen korrespondieren links der beiden Zeichen 1 keine entsprechende Zeichenfolge 00. Daher ist $uv^2w \notin L$.

2. Fall: $v = 0$: Das Wort uv^3w ist in L und enthält drei aufeinanderfolgende Zeichen 0. Damit uv^3w ein Palindrom ist müßten diese drei Zeichen 0 jedoch in der Mitte des Worts liegen, da weiter rechts und links keine aufeinanderfolgenden drei Zeichen 0 stehen. Zur ursprünglichen Mitte des Worts korrespondieren links der drei Zeichen 0 keine entsprechende Zeichenfolge 00. Daher ist $uv^3w \notin L$.

3. Fall: $|v| > 1$ und $v = 10...1$: Dann enthält Wort uv^2w zwei aufeinanderfolgende Zeichen 1 und man kann wie im 1. Fall argumentieren.

4. Fall: $|v| > 1$ und $v = 10...10$: Dann enthält Wort uv^2w zwei aufeinanderfolgende Zeichen 0 (die ursprüngliche Mitte) und links davon nur Teilzeichenketten 01 als rechts, und zwar mehr als rechts. Daher ist $uv^2w \notin L$.

In allen Fällen ergibt sich ein Widerspruch.

Die Regeln einer rechtslinearen Grammatik erfüllen die Bedingungen, die an die Regeln einer kontextfreien Grammatik gestellt werden. Jede von einer rechtslinearen Grammatik erzeugte Sprache ist daher auch eine kontextfreie Sprache. Die Sätze 4.5-2 und 4.5-3 implizieren, daß es zu jeder regulären Sprache über einem endlichen Alphabet Σ einen deterministischen

Kellerautomaten gibt, der die Sprache akzeptiert. Andererseits läßt sich mit Satz 4.5-4 zeigen, daß die deterministisch kontextfreie Sprache $L = \{a^n b^n \mid n \in \mathbf{N}\}$ nicht regulär ist. Es gilt daher:

Satz 4.5-5:
Die Klasse der regulären (Typ-3-, rechtslinearen) Sprachen über einem endlichen Alphabet Σ ist eine echte Teilmenge der deterministisch kontextfreien Sprachen über Σ.

Auch für Typ-3-Sprachen gibt es wieder viele interessante Entscheidungsprobleme. Hier sollen jedoch wieder nur das Wortproblem und das Leerheitsproblem, jetzt bezogen auf Typ-3-Sprachen, betrachtet werden. Dazu wird (analog zu den Typ-0-, Typ-1- und Typ-2-Grammatiken) ein Prädikat

$$VERIFIZIERE_G_TYP_3(w) = \begin{cases} \texttt{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-3-Grammatik darstellt} \\ \texttt{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-3-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in KG_w den Bedingungen einer Typ-3-Grammatik genügen.

Das **Wortproblem für Typ-3-Grammatiken** fragt danach, ob die Menge

$$L_{Wort_Typ-3} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE_G_TYP_3(w) = \texttt{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. In vereinfachter Darstellung wird also danach gefragt, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer rechtslinearen Grammatik G über dem Alphabet Σ und eines Worts $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ gilt oder nicht. Da dieses Problem für Typ-2-Grammatiken entscheidbar ist, gilt dieses auch für Typ-3-Grammatiken. In diesem Fall bietet sich jedoch ein einfacher Algorithmus an: Anstelle der Überprüfung $u \in L(G)$ wird untersucht, ob $u \in L(DEA)$ gilt, wobei $DEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ ein deterministischer endlicher Automat mit $L(DEA) = L(G)$ ist:

Eingabe: Ein deterministischer endlicher Automat $DEA = (Q, \Sigma, \boldsymbol{d}, q_0, F)$ und ein Wort $u \in \Sigma^*$

Verfahren: Aufruf der Funktion `accept` (DEA, u)

Ausgabe: `TRUE`, falls $u \in L(G)$, `FALSE` sonst.

```
FUNCTION accept (DEA : ...;
                 u : STRING): BOOLEAN;
         { DEA = (Q, Σ, d, q_0, F),
```

```
          u    = a1 ... an       }

VAR q : ...;
    v : STRING;
    a : CHAR;
    i : INTEGER;

BEGIN { accept }
  q := q0 ;
  v := u;

  FOR i := 1 TO Length(u) DO
    BEGIN
      a := Copy (v, 1, 1);
      Delete (v, 1, 1);
      q := d(q,a);
    END;

  IF q∈ F THEN accept := TRUE
          ELSE accept := FALSE;

END   { accept };
```

Diese Algorithmus hat eine Laufzeit der Ordnung $O(|u|)$, d.h. lineare Laufzeit.

Das **Leerheitsproblem** fragt **für Typ-3-Grammatiken**, ob die Menge

$L_{leer_Typ-3} = \{ w \mid w \in \{0,1\}^*, VERIFIZIERE_G_TYP_3(w) = \text{TRUE und } L(KG_w) = \varnothing \}$

entscheidbar ist. Diese Frage ist natürlich positiv zu beantworten, da das Leerheitsproblem für Typ-2-Grammatiken entscheidbar ist. Im Fall einer regulären L Menge kann man mit Satz 4.5-4 argumentieren. Es sei n_0 die in Satz 4.5-4 angegebene natürliche Zahl. Dann gilt:

$L \neq \varnothing$ genau dann, wenn es ein Wort $u \in L$ gibt mit $|u| < n_0$

Eine Überprüfung der Frage „$L = \varnothing$?“ kann also so ablaufen, daß man alle Wörter $u \in \Sigma^*$ mit $|u| < n_0$ darauf hin überprüft, ob $u \in L$ gilt (Wortproblem).

4.6 Eigenschaften im tabellarischen Überblick

Die folgenden Tabellen stellen wichtige Eigenschaften der behandelten Sprachklassen zusammen. Nicht alle aufgeführten Eigenschaften wurden in den vorherigen Kapiteln bewiesen; es wird vielmehr auf die angegebene Literatur verwiesen.

Beschreibungsmittel	
Typ 0	Typ-0-Grammatik Turingmaschine
Typ 1	kontextsensitive Grammatik linear beschränkter Automat
Typ 2	kontextfreie Grammatik Kellerautomat
deterministisch kontextfrei	$LR(k)$ -Grammatik deterministischer Kellerautomat
Typ 3	regulärer Ausdruck, rechtslineare Grammatik endlicher Automat

Sind das nichtdeterministische und das deterministische Modell äquivalent?	
Turingmaschine	ja
Linear beschränkter Automat	?
Kellerautomat	nein
Endlicher Automat	ja

Abschlußeigenschaften					
Operation	Schnitt $L_1 \cap L_2$	Vereinigung $L_1 \cup L_2$	Komplement $\Sigma^* \setminus L$	Produkt $L_1 \cdot L_2$	Stern L^*
Typ 0	ja	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 2	nein	ja	nein	ja	ja
det. kontextfrei	nein	nein	ja	nein	nein
Typ 3	ja	ja	ja	ja	ja

Wortproblem	
Typ 0	unlösbar
Typ 1	lösbar mit Komplexität der Ordnung $O\left(2^{O(n)}\right)$
Typ 2 (bei gegebener kfr. Grammatik)	lösbar mit Komplexität der Ordnung $O\left(n^3\right)$
deterministisch kontextfrei	lösbar mit Komplexität der Ordnung $O(n)$
Typ 3	lösbar mit Komplexität der Ordnung $O(n)$

Leerheitsproblem	
Typ 0	unlösbar
Typ 1	unlösbar
Typ 2 (bei gegebener kfr. Grammatik)	lösbar
deterministisch kontextfrei	lösbar
Typ 3	lösbar

5 Praktische Berechenbarkeit

Im vorliegenden Kapitel geht es um die Lösung von Problemen „aus der Praxis“, insbesondere um die Untersuchung des Aufwandes, den diese Lösungen erfordern. **Alle hier behandelten (Entscheidungs-) Probleme sind entscheidbar, d.h. es gibt jeweils einen Algorithmus, der bei jeder Eingabe mit einer akzeptierenden oder verwerfenden Entscheidung stoppt**. Wie ein derartiger Algorithmus formuliert wird, ob als deterministische oder nichtdeterministische Turingmaschine, RAM oder Programm in einer Programmiersprache, ergibt sich jeweils aus dem Zusammenhang; das für die Darstellung geeignetste Modell wird verwendet.

Der Begriff der Entscheidbarkeit bezieht sich auf Entscheidungsprobleme, in der Praxis hat man jedoch häufig mit Optimierungsproblemen zu tun. Beispielsweise wurde in Kapitel 1.2 das Problem des Handlungsreisenden als Optimierungsproblem, Entscheidungsproblem und Berechnungsproblem beschrieben. Der Zusammenhang zwischen diesen Problemtypen wird in Kapitel 5.1 genauer betrachtet.

Im folgenden werden zwei Beispiele für Optimierungsprobleme angeführt, die sich auf Graphen beziehen. Für beide Probleme werden Lösungsalgorithmen angegeben, die sich in ihrer Laufzeit wesentlich unterscheiden.

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Zur algorithmischen Behandlung wird der Graph $G = (V, E, w)$ in Form seiner **Adjazenzmatrix** $A(G)$ gespeichert. Diese ist definiert durch

$$A(G) = A_{(n,n)} = \lfloor a_{i,j} \rfloor_{(n,n)} \text{ mit } a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases} .$$

Es werden folgende Datentypen verwendet:

```
CONST n          = ...;        { Problemgröße }
      unendlich = ...;         { hier kann der maximal mögliche
                                 REAL-Wert verwendet werden       }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      bit_feld      = ARRAY [knotenbereich] OF BOOLEAN;
      feld          = ARRAY [knotenbereich] OF REAL;
```

Die folgende Funktion wird zur Ermittlung des Minimums zweier REAL-Zahlen eingesetzt:

```
FUNCTION min (a, b: REAL):REAL;

BEGIN { min }
  IF a <= b THEN min := a
            ELSE min := b;
END   { min };
```

Der folgende Algorithmus zur Lösung des Problems der Wege mit minimalem Gewicht in gerichteten Graphen ist ein Beispiel für ein allgemeines Lösungsprinzip: die **Greedy-Methode**. Eine (global) optimale Lösung wird schrittweise gefunden, indem Entscheidungen gefällt werden, die auf „lokalen" Informationen beruhen. Für eine Entscheidung wird hierbei nur ein kleiner (lokaler) Teil der Informationen genutzt, die über das gesamte Problem zur Verfügung stehen. Es wird die in der jeweiligen Situation optimal erscheinende Entscheidung getroffen, unabhängig von vorhergehenden und nachfolgenden Entscheidungen. Die einmal getroffenen Entscheidungen werden im Laufe des Rechenprozesses nicht mehr revidiert. Diese Methode ist jedoch nicht auf alle Optimierungsprobleme anwendbar.

Zunächst gelten alle Knoten v_i bis auf den Knoten v_1 als „nicht behandelt". Aus den nicht behandelten Knoten wird ein Knoten ausgewählt, der zu einer bisherigen Teillösung hinzugenommen wird, und zwar so, daß dadurch eine bzgl. der Teillösung, d.h. bzgl. der behandelten Knoten, optimale Lösung entsteht.

Eingabe: $G = (V, E, w)$ ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

```
VAR A       : matrix;          { Adjazenzmatrix }
    i       : knotenbereich;
    j       : knotenbereich;
    distanz : feld;
```

```
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF (v_i, v_j) ∈ E THEN A[i, j] := w((v_i, v_j))
                          ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `minimale_wege (A, distanz);`

Ausgabe: $d_i =$ `distanz[i]` für $i = 1, ..., n$.

```
PROCEDURE minimale_wege ( A           : matrix;
                          VAR distanz : feld);

VAR rest_knoten : bit_feld;      { Kennzeichnung der noch nicht
                                   behandelten Knoten: ist v_i ein
                                   nicht behandelter Knoten, so
                                   ist rest_knoten[i] = TRUE       }
    idx         : knoten_bereich;
    u           : knoten_bereich;
    exist       : BOOLEAN;

  PROCEDURE auswahl (VAR knoten   : knoten_bereich;
                     VAR gefunden : BOOLEAN);

    { die Prozedur wählt unter den noch nicht behandelten
      Knoten einen Knoten mit minimalem distanz-Wert aus
      (in knoten); gibt es einen derartigen Knoten, dann ist
      gefunden = TRUE, sonst ist gefunden = FALSE              }

      VAR  i   : knoten_bereich;
           min : REAL;

    BEGIN { auswahl }
      gefunden := FALSE;
      min      := unendlich;

      FOR i := 1 TO n DO
        IF rest_knoten [i]
        THEN BEGIN
               IF distanz [i] < min
               THEN BEGIN
                      gefunden := TRUE;
                      knoten   := i;
                      min      := distanz [i]
                    END
             END
    END { auswahl };
```

```
BEGIN { minimale_wege }
  { Gewichte zum Startknoten initialisieren und alle Knoten,
    bis auf den Startknoten, als nicht behandelt kennzeichnen: }
   FOR idx := 1 TO n DO
     BEGIN
       distanz[idx]     := A[1, idx];
       rest_knoten[idx] := TRUE
     END;
   distanz[1]     := 0;
   rest_knoten[1] := FALSE;

   { einen Knoten mit minimalem Distanzwert aus den Restknoten
     auswählen:                                                }
   auswahl (u, exist);

   WHILE exist DO
     BEGIN
       { den gefundenen Knoten aus den noch nicht behandelten
          Knoten herausnehmen:                                 }
       rest_knoten[u] := FALSE;

       { für jeden noch nicht behandelten Knoten idx, der mit
         dem Knoten u verbunden ist, den distanz-Wert auf den
         neuesten Stand bringen:                               }
       FOR idx := 1 TO n DO
         IF rest_knoten[idx]
            AND
            (A[u, idx] <> unendlich)
         THEN distanz [idx] := min (distanz[idx],
                                    distanz[u] + A[u, idx]);

       { einen neuen Knoten mit minimalem Distanzwert aus den
         Restknoten auswählen:                                 }
       auswahl (u, exist);

     END  {WHILE exist }

END { minimale_wege };
```

Es gilt bei Eintritt in die `WHILE exist DO`-Schleife folgende Schleifeninvariante:

(1) Ist der Knoten v_i ein bereits behandelter Knoten, d.h. `rest_knoten[i]=FALSE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

(2) Ist der Knoten v_i ein nicht behandelter Knoten, d.h. `rest_knoten[i]=TRUE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der bis auf v_i nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

Aus der Schleifeninvariante folgt die Korrektheit des Algorithmus.

Satz 5-1:

Ist $G = (V, E, w)$ eine Instanz des Problems der Wege mit minimalem Gewicht in gerichteten Graphen mit *n* Knoten, dann liefert das beschriebene Verfahren mit der Prozedur `minimale_wege` im Feld `distanz` die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für *i* = 1, ..., *n*. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n^2)$.

Beispiel:

Die Instanz $G = (V, E, w)$ sei gegeben durch folgende graphische Darstellung:

Die Adjazenzmatrix zu *G* lautet

$$A(G) = \begin{bmatrix} \infty & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 9 & \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & \infty & 5 & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & \infty & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 8 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

`u` wird jeweils beim Aufruf der Prozedur `auswahl (u, exist)` ausgewählt:

`u`	nichtbehandelte Knoten	`distanz [1]`	`distanz [2]`	`distanz [3]`	`distanz [4]`	`distanz [5]`	`distanz [6]`	`distanz [7]`	`distanz [8]`
	2 3 4 5 6 7 8	---	28	2	∞	1	∞	∞	∞
5	2 3 4 6 7 8		9	2	∞	---	27	∞	∞
3	2 4 6 7 8		9	---	∞		26	∞	29
2	4 6 7 8		---		18		19	∞	∞
4	6 7 8				---		19	26	25
6	7 8						---	26	20
8	7							26	---
7	leer							---	

Das zweite Besipiel löst das Handlungsreisenden-Optimierungsproblem nach der sogenannten **Branch-and-Bound-Methode**. Potentielle, aber nicht unbedingt optimale Lösungen werden systematisch erzeugt. Diese werden in Teilmengen aufgeteilt, die sich auf Knoten eines Entscheidungsbaums abbilden lassen. Es wird eine Abschätzung für das Optimum mitgeführt und während des Verfahrensverlaufs laufend aktualisiert. Potentielle Lösungen, deren Zielfunktion einen „weit von der Abschätzung entfernten Wert“ aufweisen, werden nicht weiter betrachtet, d.h. der Teilbaum, der bei einer derartigen Lösung beginnt, muß nicht weiter durchlaufen werden.

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Eine **Tour** durch *G*, d.h. eine geschlossene Kantenfolge

$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ mit $(v_{i_j}, v_{i_{j+1}}) \in E$ für $j = 1, ..., n-1$,

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht. Man kann o.B.d.A. $v_{i_1} = v_1$ setzen.

Die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist dabei die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

$$\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1}).$$

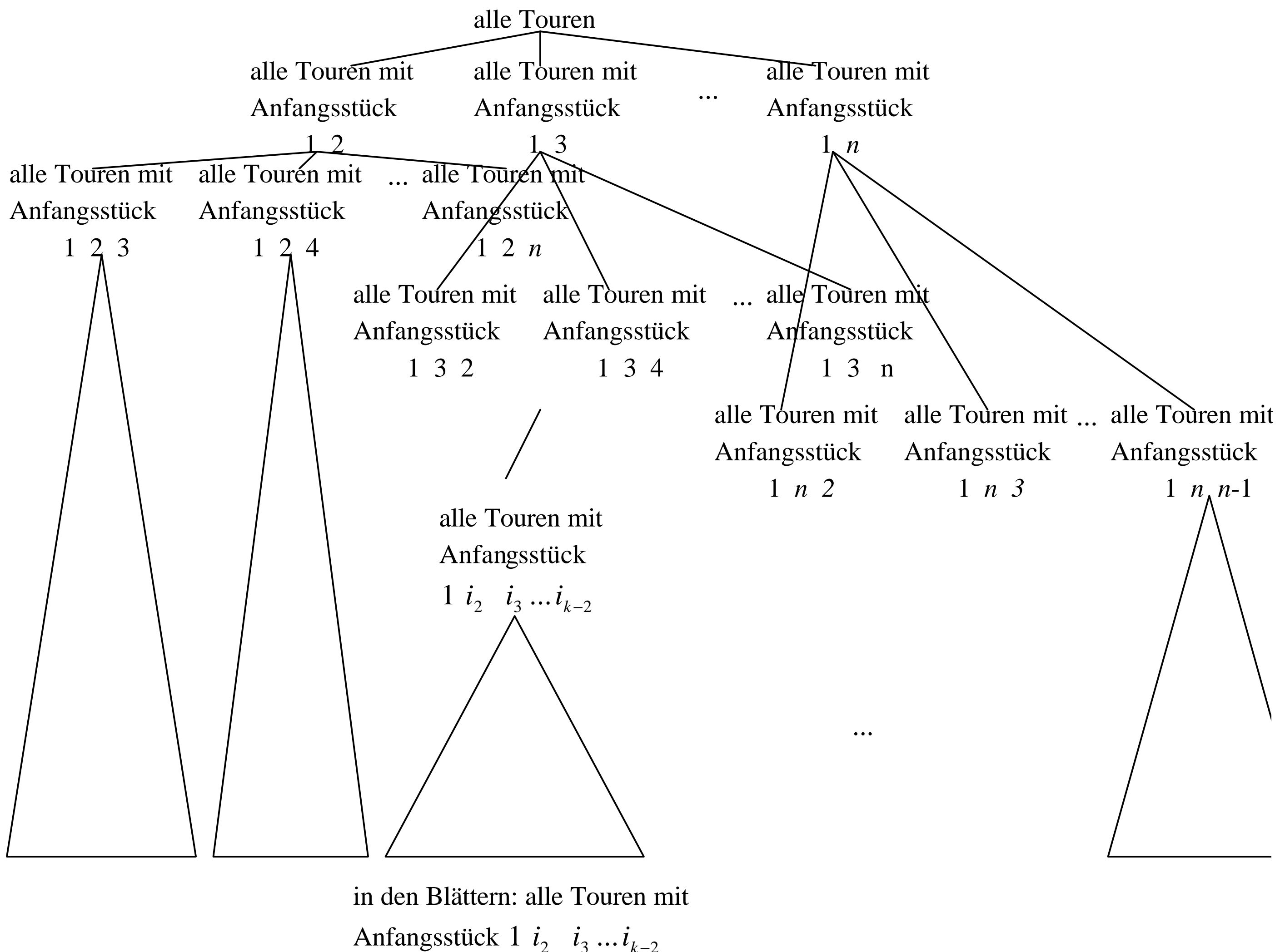
Es wird angenommen, daß eine Tour bei v_1 beginnt und endet. Außerdem wird angenommen, daß der Graph **vollständig** ist, d.h. daß $w((v_i, v_j))$ für jedes $v_i \in V$ und $v_j \in V$ definiert ist (eventuell ist $w((v_i, v_j)) = \infty$).

Im folgenden wird (zur Vereinfachung der Darstellung) die Knotenmenge $V = \{v_1, ..., v_n\}$ mit der Menge der Zahlen $\{1, ..., n\}$ gleichgesetzt (dem Knoten v_i entspricht die Zahl *i*). Eine Tour durch *G* läßt sich dann als Zahlenfolge

$$\langle 1 \ i_1 \ i_2 \ldots i_{n-1} \ 1 \rangle$$

darstellen, wobei alle i_j paarweise verschieden (und verschieden von 1) sind. Alle Touren erhält man, wenn man für i_j in $\langle 1 \ i_1 \ i_2 \ldots i_{n-1} \ 1 \rangle$ alle $(n-1)!$ Permutationen der Zahlen 2, ..., *n* einsetzt. Manche dieser Touren haben eventuell das Gewicht ∞.

Alle Touren (Zahlenfolgen der beschriebenen Art) lassen sich als Blätter eines Baums darstellen:



Alle Touren (Zahlenfolgen der beschriebenen Art) werden systematisch erzeugt (siehe unten). Dabei wird eine obere Abschätzung für das Gewicht einer optimalen Tour (Tour mit minimalem Gewicht) in der globalen Variablen `bound` mitgeführt (Anfangswert `bound` = ∞). Wird ein Anfangsstück einer Tour erzeugt, deren Gewicht größer als der Wert in der Variablen `bound` ist, dann brauchen alle Touren, die mit diesem Anfangsstück beginnen, nicht weiter betrachtet zu werden. Es wird also ein ganzer Teilbaum aus dem Baum aller Touren abgeschnitten. Ist schließlich eine Tour gefunden, deren Gewicht kleiner oder gleich dem Wert in der Variablen `bound` ist, so erhält die Variable `bound` als neuen Wert dieses Gewicht, und die Tour wird als temporär optimale Tour vermerkt. Eine Variable `opttour` nimmt dabei die Knotennummern einer temporär optimalen Tour auf; die Variable `teilfeld` dient der Aufnahme des Anfangsstücks einer Tour.

Ist bereits ein Anfangsstück $\langle 1\ i_1\ i_2 \ldots i_{k-1} \rangle$ einer Tour erzeugt, so sind die Zahlen $1,\ i_1,\ i_2, \ldots, i_{k-1}$ alle paarweise verschieden. Eine weitere Zahl i kann in die Folge aufgenommen werden, wenn

(1) i unter den Zahlen $1,\ i_1,\ i_2, \ldots, i_{k-1}$ nicht vorkommt und

(2) $w\left(\left(v_{i_{k-1}}, v_i\right)\right) < \infty$ ist und

(3) das Gewicht der neu entstehenden Teiltour $\langle 1,\ i_1,\ i_2, \ldots, i_{k-1}\ i \rangle$ kleiner oder gleich dem Wert in der Variablen `bound` ist.

Treffen (1) oder (2) nicht zu, kann i nicht als Fortsetzung der Teiltour $1,\ i_1,\ i_2, \ldots, i_{k-1}$ genommen werden, denn es entsteht keine Tour. Trifft (3) nicht zu (aber (1) und (2)), dann brauchen alle Touren, die mit dem Anfangsstück $\langle 1,\ i_1,\ i_2, \ldots, i_{k-1}\ i \rangle$ beginnen, nicht weiter berücksichtigt zu werden.

Algorithmus zur Lösung des Problems des Handlungsreisenden nach der Branch-and-bound-Methode:

Es werden wieder die bereits bekannten Deklarationen (ergänzt um neue Deklarationen) verwendet:

```
CONST n         = ...;        { Problemgröße }
      unendlich = ...;        { hier kann der maximal mögliche
                                REAL-Wert verwendet werden       }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      tourfeld      = ARRAY [1..(n+1)] OF INTEGER;
```

Die Knotennummern einer optimalen Tour werden im Feld `opttour` abgelegt.

Eingabe: $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

```
VAR A       : matrix;          { Adjazenzmatrix }
    i       : knotenbereich;
    j       : knotenbereich;
    opttour : tourfeld;
    bound   : REAL;
```

```
FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i, v_j) ∈ E THEN A[i, j] := w((v_i, v_j))
                      ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `salesman_bab (A, opttour, bound)`.

Ausgabe: `bound` gibt das minimale Gewicht einer Tour durch G an, die beim Knoten v_1 beginnt und endet; `opttour` enthält die Knotennummern einer optimalen Tour.

```
PROCEDURE salesman_bab
                  (A           : matrix;
                   VAR opttour : tourfeld; { optimale Tour         }
                   VAR bound   : REAL      { Länge der Tour        }
                   );

  VAR i : INTEGER;

  PROCEDURE  bab_g
                  (teilfeld : tourfeld;
                                       { zu ergänzendes Anfangsstück
                                         einer Tour                }
                   gewicht  : REAL;    { Gewicht des Anfangsstücks }
                   position : INTEGER  { Position, an der zu
                                         ergänzen ist              }
                  );

     VAR i  : INTEGER;
         j  : INTEGER;
         ok : BOOLEAN;
         w  : REAL;

     BEGIN {bab_g }
       IF position = n + 1
       THEN BEGIN
              w := A[teilfeld[n], 1];
              IF (w < unendlich)
                 AND
                 (gewicht + w < bound)
              THEN BEGIN
                     teilfeld[position] := 1;
                     bound    := gewicht + w;
                     opttour  := teilfeld;
                   END;
            END
```

```
    ELSE BEGIN
           FOR i := 2 TO n DO
             BEGIN
               w  := A[teilfeld[position - 1], i];
               ok := TRUE;
               { Bedingung (2):}
               FOR j := 2 TO position - 1 DO
                 IF teilfeld[j] = i
                 THEN BEGIN
                        ok := FALSE;
                        Break;
                      END;
               IF ok
                  AND
                  (w < unendlich)
                  AND
                  (gewicht + w < bound)
               THEN BEGIN
                      teilfeld[position] := i;
                      bab_g (teilfeld, gewicht + w,
                                                    position + 1);
                    END;
             END;
         END;

  END   {bab_g };

BEGIN { salesman_bab }

  FOR i := 2 TO n + 1 DO
   opttour[i]  := 0;
  opttour[1] := 1;
  bound   := unendlich;

  bab_g(opttour, 0, 2);

END   {salesman_bab };
```

Das Verfahren erzeugt u.U. alle $(n-1)!$ Folgen $\langle 1\ i_1\ \ i_2 \ldots i_{n-1}\ \ 1 \rangle$, bis es eine optimale Tour gefunden hat. In der Praxis werden jedoch schnell Folgen mit Anfangsstücken, die ein zu großes Gewicht aufweisen, ausgeschlossen. Die (worst-case-) Zeitkomplexität des Verfahrens bleibt jedoch mindestens exponentiell in der Anzahl der Knoten des Graphen in der Eingabeinstanz.

In der Praxis gilt ein Algorithmus mit exponentiellem Laufzeitverhalten als **schwer durchführbar (intractable)**, ein Algorithmus mit polynomiellem Laufzeitverhalten als **leicht**

durchführbar (tractable). Natürlich kann dabei ein Algorithmus mit exponentiellem Laufzeitverhalten für einige Eingabeinstanzen schnell ablaufen. Trotzdem bleibt die Ursache zu untersuchen, die dazu führt, daß manche Probleme „leicht lösbar“ sind (d.h. einen Algorithmus mit polynomiellem Laufzeitverhalten besitzen), während für andere Probleme bisher nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten bekannt sind.

Der Grund dafür, daß ein Verfahren mit exponentielle Laufzeit als nicht durchführbar gilt, wird aus den folgenden Tabelle sichtbar. Es werden dort jeweils fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$ und einige ausgewählte (gerundete) Funktionswerte gezeigt. Jede Funktion soll als Laufzeit für Algorithmen interpretiert werden: Der Funktionswert $h_i(n)$ gibt die Anzahl der Rechenschritte an, die bei einer Eingabeinstanz der Größe n durchlaufen werden. Selbst kleine Problemgrößen führen bereits auf eine immense Laufzeit.

Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5
i	$h_i(n)$	$h_i(10)$	$h_i(100)$	$h_i(1000)$
1	$\log_2(n)$	3,3219	6,6439	9,9658
2	$\sqrt{n}$	3,1623	10	31,6228
3	n	10	100	1000
4	n^2	100	10.000	1.000.000
5	2^n	1024	$1{,}2676506 \cdot 10^{30}$	$> 10^{693}$

Die folgende Tabelle zeigt noch einmal die fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$. Es sei $y_0 > 0$ ein fester Wert. Die dritte Spalte zeigt für jede der fünf Funktionen Werte n_i mit $h_i(n_i) = y_0$. In der vierten Spalte sind diejenigen Werte $\overline{n_i}$ aufgeführt, für die $h_i\left(\overline{n_i}\right) = 10 \cdot y_0$ gilt, d.h. dort ist angegeben, auf welchen Wert man die Problemgröße n_i vergrößern kann, wenn man die 10-fachen Laufzeit in Kauf nimmt. Wie man sieht, kann man bei der Logarithmusfunktion wegen ihres langsamen Wachstums den Wert stark vergrößern, während bei der schnell anwachsenden Exponentialfunktion nur eine additive konstante Steigerung um ca. 3,3 möglich ist. Das bedeutet, daß selbst bei einer Verzehnfachung der Rechenleistung nur eine geringfügig vergrößerte Problemgröße zu bewältigen ist.

Spalte 1	Spalte 2	Spalte 3	Spalte 4
i	$h_i(n)$	n_i mit $h_i(n_i) = y_0$	$\overline{n}_i$ mit $h_i(\overline{n}_i) = 10 \cdot y_0$
1	$\log_2(n)$	n_1	$(n_1)^{10}$
2	$\sqrt{n}$	n_2	$100 \cdot n_2$
3	n	n_3	$10 \cdot n_3$
4	n^2	n_4	$\approx 3{,}162 \cdot n_4$
5	2^n	n_5	$n_5 + 3{,}322$

5.1 Der Zusammenhang zwischen Optimierungs- und Entscheidungsproblemen

Optimierungsprobleme spielen in der Anwendung eine bedeutende Rolle; in der Theorie der Berechenbarkeit sind es eher die Entscheidungsprobleme. Zwischen beiden Typen besteht jedoch ein enger Zusammenhang.

Es sei Π ein Optimierungsproblem mit einer reellwertigen Zielfunktion:

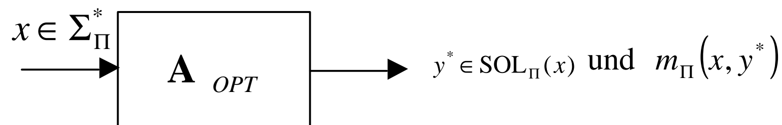
Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ eine Menge zulässiger Lösungen zuordnet
3. Spezifikation einer Zielfunktion m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den Wert einer zulässigen Lösung, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$)
bzw.
$m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

Eine Instanz x ist eine Zeichenkette $x \in \Sigma_\Pi^*$. Hier wird das Alphabet Σ_Π für das Problem „geeignet“ gewählt. Für ein Graphenproblem ist sicherlich dabei ein anderes Alphabet geeignet

als zur Darstellung Boolescher Ausdrücke. Letztlich kann man sich natürlich auf ein allgemein gültiges „Grundalphabet“ verständigen, etwa $\Sigma_0 = \{0, 1\}$.

Ein Lösungsalgorithmus $\mathbf{A}_{OPT}$ für Π hat die Form



Die von $\mathbf{A}_{OPT}$ bei Eingabe von $x \in \Sigma_\Pi^*$ ermittelte Lösung ist $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$, d.h. sie besteht aus einer optimalen Lösung y^* und dem Wert der Zielfunktion $m_\Pi(x, y^*) = m_\Pi^*(x)$ bei einer optimalen Lösung. Natürlich kann es für ein Optimierungsproblem mehrere optimale Lösungen geben; der Wert $m_\Pi^*(x)$ ist jedoch eindeutig.

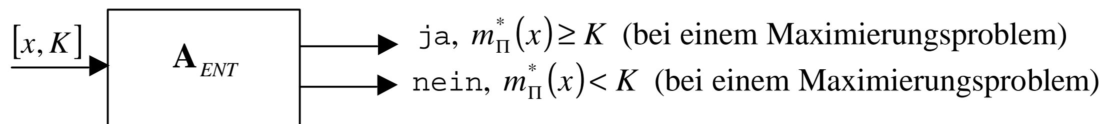
Das zu Π **zugehörige Entscheidungsproblem** Π_{ENT} wird definiert durch

Instanz: $[x, K]$
mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{R}$

Lösung: Entscheidung „ja“, falls für den Wert $m_\Pi^*(x)$ einer optimalen Lösung
$m_\Pi^*(x) \geq K$ bei einem Maximierungsproblem
(d.h. $goal_\Pi = \max$)
bzw.
$m_\Pi^*(x) \leq K$ bei einem Minimierungsproblem
(d.h. $goal_\Pi = \min$)
gilt,
Entscheidung „nein“, sonst.

Hierbei ist zu beachten, daß beim Entscheidungsproblem weder nach einer optimalen Lösung noch nach dem Wert $m_\Pi^*(x)$ der Zielfunktion bei einer optimalen Lösung gefragt wird.

Ein Lösungsalgorithmus für Π_{ENT} hat zwei mögliche Ausgänge, nämlich einen akzeptierenden und einen verwerfenden Ausgang. Der erste Ausgang wird erreicht (aktiviert), wenn bei Eingabe einer Instanz für Π_{ENT} die Entscheidung „ja“ getroffen wird, der zweite Ausgang entspricht der Entscheidung „nein“. Ein Lösungsalgorithmus $\mathbf{A}_{ENT}$ für Π_{ENT}, hier für ein Maximierungsproblem dargestellt, hat daher die Form



Aus der Kenntnis eines Algorithmus $\mathbf{A}_{OPT}$ für ein Optimierungsproblem läßt sich leicht ein Algorithmus $\mathbf{A}_{ENT}$ für das zugehörige Entscheidungsproblem konstruieren, der im wesentlichen dieselbe Komplexität besitzt:

Eingabe: [x, K]
mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{R}$

Verfahren: Man berechne $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$ und vergleiche das Resultat $m_\Pi(x, y^*) = m_\Pi^*(x)$ mit K:

Ausgabe: $\mathbf{A}_{ENT}([x, K]) = \texttt{ja}$, falls $m_\Pi^*(x) \geq K$ bei einem Maximierungsproblem ist
bzw.
$\mathbf{A}_{ENT}([x, K]) = \texttt{ja}$, falls $m_\Pi^*(x) \leq K$ bei einem Minimierungsproblem ist,
$\mathbf{A}_{ENT}([x, K]) = \texttt{nein}$ sonst.

Daher gilt:

Satz 5.1-1:

Das zu einem Optimierungsproblem Π zugehörige Entscheidungsproblem Π_{ENT} ist im wesentlichen *nicht schwieriger zu lösen* als das Optimierungsproblem.

Falls man andererseits bereits weiß, daß das Entscheidungsproblem Π_{ENT} immer nur „schwer lösbar" ist, z.B. beweisbar nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten besitzt, dann ist das Optimierungsproblem ebenfalls nur „schwer lösbar".

In den folgenden Kapiteln wird für einige *Entscheidungsprobleme* Π_{ENT} gezeigt, daß sie (vermutlich) keine schnell laufenden Lösungsverfahren besitzen. Daher können die zugehörigen Optimierungsprobleme, an deren Lösung man in der Praxis interessiert ist, ebenfalls nur mit sehr lang laufenden Verfahren angegangen werden. Die algorithmische Untersuchung von Entscheidungsproblemen ist in diesen Fällen daher für die Praxis von großem Interesse. Man hat zudem abzuwägen, ob man nicht mit einer Lösung zufrieden sein kann, die „schnell" zu

finden ist und sich der optimalen Lösung annähert. Diese Fragestellung führt auf das Gebiet der **Approximationsalgorithmen** in Kapitel 6.

In einigen Fällen läßt sich auch die „umgekehrte" Argumentationsrichtung zeigen: Aus einem Algorithmus $\mathbf{A}_{ENT}$ für das zu einem Optimierungsproblem zugehörige Entscheidungsproblem läßt sich ein Algorithmus $\mathbf{A}_{OPT}$ für das Optimierungsproblem konstruieren, dessen Laufzeitverhalten im wesentlichen dieselbe Komplexität aufweist. Insbesondere ist man dabei an Optimierungsproblemen interessiert, für die gilt: Hat $\mathbf{A}_{ENT}$ zur Lösung des zu einem Optimierungsproblem zugehörigen Entscheidungsproblem polynomielles Laufzeitverhalten (in der Größe der Eingabe), so hat auch der aus $\mathbf{A}_{ENT}$ konstruierte Algorithmus $\mathbf{A}_{OPT}$ zur Lösung des Optimierungsproblems polynomielles Laufzeitverhalten.

Ein Beispiel ist das Problem des Handlungsreisenden auf Graphen mit natürlichzahligen Gewichten:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimierungsproblem

Instanz: 1. $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein **natürlichzahliges Gewicht**; es ist $w((i, j)) = \infty$ für $(i, j) \notin E$

Als Problemgröße wird hier der Wert $k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$ genommen, da in das Verfahren wesentlich die numerischen Größen der beteiligten Kantengewichte eingeht

2. $\text{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist eine Tour durch $G\}$

3. für $T \in \text{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$

4. $goal = \min$

Lösung: Eine Tour $T^* \in \text{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

Ein Algorithmus, der bei Eingabe einer Instanz $G=(V,E,w)$ eine optimale Lösung erzeugt, werde mit $\mathbf{A}_{TSPOPT}$ bezeichnet. Die von $\mathbf{A}_{TSPOPT}$ bei Eingabe von G ermittelte Tour mit minimalen Kosten sei T^*, die ermittelten minimalen Kosten $m^*(G)$:
$\mathbf{A}_{TSPOPT}(G)=(T^*,m^*(G))$.

Das zugehörige Entscheidungsproblem lautet:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Entscheidungsproblem

Instanz: $[G,K]$

$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E\subseteq V\times V$; die Funktion $w:E\rightarrow\mathbf{N}$ gibt jeder Kante $e\in E$ ein natürlichzahliges Gewicht; es ist $w((i,j))=\infty$ für $(i,j)\notin E$;

$K\in\mathbf{N}$.

Die Problemgröße ist $k=n\cdot B$ mit $B=\max\{\lceil\log(w(e))\rceil \mid e\in E\}$.

Lösung: Entscheidung „ja“, falls es eine Tour durch G gibt mit minimalen Kosten $m^*(G)\le K$ gibt,

Entscheidung „nein“, sonst.

Ein Algorithmus, der bei Eingabe einer Instanz $[G,K]$ eine entsprechende Entscheidung fällt, werde mit $\mathbf{A}_{TSPENT}$ bezeichnet. Die von $\mathbf{A}_{TSPENT}$ bei Eingabe von $[G,K]$ getroffene `ja/nein`-Entscheidung sei $\mathbf{A}_{TSPENT}([G,K])$. Mit Hilfe von $\mathbf{A}_{TSPENT}$ wird ein Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem konstruiert. Dabei wird wiederholt Binärsuche eingesetzt, um zunächst die Kosten einer optimalen Tour zu ermitteln.

Der Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem verwendet eine als Pseudocode formulierte Prozedur `A_TSPOPT`. Der Eingabegraph kann wieder in Form seiner Adjazenzmatrix verarbeitet werden. Da Implementierungsdetails hier nicht weiter betrachtet werden sollen, können wir annehmen, daß es zur Darstellung eines gewichteten Graphen einen geeigneten Datentyp

```
TYPE gewichteter_Graph = ...;
```

und zur Speicherung einer Tour (Knoten und Kanten) in dem Graphen einen Datentyp

```
TYPE tour = ...;
```

gibt. Der Algorithmus $\mathbf{A}_{TSPOPT}$ hat folgendes Aussehen:

Eingabe: $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; ; es ist $w((i, j)) = \infty$ für $(i, j) \notin E$

```
VAR G       : gewichter_Graph;
    opttour : tour;
    opt     : INTEGER;
```

G := $G = (V, E, w)$

Verfahren: Aufruf der Prozedur `A_TSPOPT (G, opt, opttour)`.

Ausgabe: `opttour` = T^*, d.h. eine Tour mit minimalen Kosten, `opt` = die ermittelten minimalen Kosten $m^*(G)$.

```
PROCEDURE A_TSPOPT (G           : gewichter_Graph;
                    {  Graph mit natürlichzahligen Kantengewichten }
                    VAR opt     : INTEGER;
                     { Gewicht einer optimalen Tour                }
                    VAR opttour : graph;
                     { ermittelte Tour mit minimalem Gewicht       }

VAR s : INTEGER;


   PROCEDURE TSPOPT (G       : gewichter_graph;
                     VAR opt : INTEGER;)
        { ermittelt im Parameter opt das Gewicht einer
          optimalen Tour in G :                            }

      VAR min : INTEGER;
          max : INTEGER;
          t   : INTEGER;

   BEGIN { TSPOPT }
     min := 0;
     max := ∑_{e∈E} w(e) ;

     { Binärsuche auf dem Intervall [0..max]: }
     WHILE max - min >= 1 DO
       BEGIN
```

```
          t := ⌈(min+max)/2⌉;
          IF A_TSPENT([G,t]) = „ja“
          THEN max := t
          ELSE min := t + 1;
        END;

      { t enthält nun das Gewicht einer optimalen Tour in G: }
      opt := t;
    END   { TSPOPT };

BEGIN { A_TSPOPT }
  TSPOPT (G, opt);

  FOR alle e∈E DO
    BEGIN
      ersetze in G das Gewicht w(e) der Kante e durch w(e) + 1;
      TSPOPT (G, s);
      IF s > opt
      THEN { es gibt keine optimale Tour in G, die e nicht enthält }
           ersetze in G das Gewicht w(e) der Kante e durch w(e) - 1;
    END;

  { alle Kanten mit nicht erhöhten Gewichten bestimmen eine Tour
    mit minimalen Kosten                                         }
  tour := Menge der Kanten mit nicht erhöhten Gewichten und zugehörige Knoten;
END   { A_TSPOPT };
```

Der Graph G habe die Problemgröße $k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$. Es sei $m = \sum_{e \in E} w(e)$. Dann sind in obigem Verfahren höchstens $\lceil \log_2(m) \rceil$ viele Aufrufe des Entscheidungsverfahrens $\mathbf{A}_{TSPENT}([G,\mathrm{t}])$ erforderlich (Binärsuche). Es gilt

$$\lceil \log_2(m) \rceil \le \log_2(m) + 1 \le \sum_{e \in E} \log_2(w(e)) + 1 \le \sum_{e \in E} \lceil \log_2(w(e)) \rceil + 1 \le n^2 \cdot B + 1 \in O(k^2),$$

d.h. $\lceil \log_2(m) \rceil \in O(k^2)$.

Falls man also weiß, daß $\mathbf{A}_{TSPENT}$ ein in der Größe der Eingabe polynomiell zeitbeschränkter Algorithmus ist, ist das gesamte Verfahren zur Bestimmung einer optimalen Lösung polynomiell zeitbeschränkt.

Falls man umgekehrt weiß, daß es beweisbar keinen schnellen Algorithmus zur Lösung des Problems des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimierungsproblem gibt, gibt es einen solchen auch nicht für das zugehörige Entscheidungspro-

blem. In diesem Fall ist das Entscheidungsproblem genauso schwer zu lösen wie das Optimierungsproblem.

5.2 Komplexitätsklassen

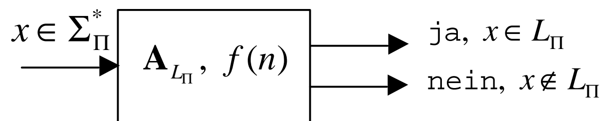
Eine Eingabeinstanz $G = (V, E, w))$ des Handlungsreisenden-Minimierungsproblem besteht aus natürlichen Zahlen für die Knotennummern, Paare natürlicher Zahlen für die Kanten mit ihren Gewichten und trennenden Klammern und dem Kommazeichen. Die Zahlen lassen sich im Binärdarstellung angeben, so daß man als Alphabet zur Formulierung der Eingabeinstanzen die Menge $\Sigma_\Pi = \{0, 1, (,)\} \cup \text{Kommazeichen}$ wählen kann. Als Problemgröße der Eingabeinstanz wird in Kapitel 5.1 der Wert $k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$ genommen, d.h. ein Wert, der proportional zu der Anzahl an Zeichen ist, um eine Eingabeinstanz zu kodieren. Dieser Ansatz berücksichtigt, daß für die Berechnung der Zeitkomplexität eines Algorithmus in der Regel das uniforme Kostenkriterium zugrunde gelegt wird, die Laufzeit aber in einigen Fällen von der numerischen Größe der Eingabe abhängt, so daß eigentlich das logarithmische Kostenkriterium angewendet werden müßte. Dadurch daß eine numerische Eingabe in der Anzahl der Zeichen, um sie darzustellen, gemessen wird, ist implizit die Laufzeitberechnung nach dem korrekt anzuwendenden Kostenkriterium gegeben.

Im folgenden wird (bei allgemeinen Betrachtungen) als **Größe einer Eingabeinstanz** $x \in \Sigma_\Pi^*$ für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ die Anzahl der Zeichen in x genommen. Die Laufzeit eines Entscheidungsalgorithmus wird dann in Abhängigkeit von $n = |x|$ beschrieben.

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in TIME(f(n))$, wenn es einen (deterministischen) Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: `ja`, falls $x \in L_\Pi$ gilt
`nein`, falls $x \notin L_\Pi$ gilt.



Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(f(n))$ liegt.

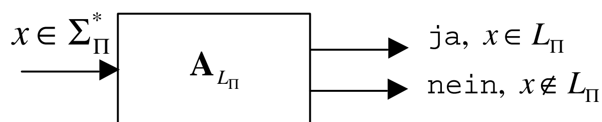
Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $TIME(f(n))$" steht synonym für $L_\Pi \in TIME(f(n))$; man sagt auch abkürzend „das Entscheidungsproblem Π liegt in $TIME(f(n))$". In diesem Sinn bezeichnet $TIME(f(n))$ die **Klasse der Entscheidungsprobleme, die in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Eine entsprechende Definition gilt für die Speicherplatzkomplexität:

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in SPACE(f(n))$, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: `ja`, falls $x \in L_\Pi$ gilt
`nein`, falls $x \notin L_\Pi$ gilt.



Hierbei werden zur Findung der Entscheidung $\mathbf{A}_{L_\Pi}(x)$ eine Anzahl von Speicherzellen verwendet, die in $O(f(n))$ liegt.

Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $SPACE(f(n))$“ steht synonym für $L_\Pi \in SPACE(f(n))$; man sagt auch „das Entscheidungsproblem Π liegt in $SPACE(f(n))$“. In diesem Sinn bezeichnet $SPACE(f(n))$ die **Klasse der Entscheidungsprobleme, die mit Speicherplatzkomplexität $O(f(n))$ gelöst werden können.**

Wichtige Komplexitätsklassen sind

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einem Polynom in der Größe der Eingabe ist:

  $$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right)$$

  Beispielsweise gibt es für jede von einer kontextfreien Grammatik G erzeugten Sprache $L(G) \subseteq \Sigma^*$ ein Entscheidungsverfahren, das für ein Wort $u \in \Sigma^*$ mit $|u| = n$ in $O\left(n^3\right)$ vielen Schritten entscheidet, ob $u \in L(G)$ gilt oder nicht (vgl. Kapitel 4.4). Daher ist die Klasse der kontextfreien Sprachen in **P** enthalten. Diese Inklusion ist echt, wie das Beispiel der Sprache $L_4 = \left\{ a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1 \right\}$ aus Kapitel 4.1 zeigt, die in **P** liegt, aber nicht kontextfrei ist.
- die Klasse der Entscheidungsprobleme, die in mit einem Speicherplatzbedarf gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist:

  $$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} SPACE\left(n^k\right)$$

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einer Exponentialfunktion in der Größe der Eingabe ist:

  $$\mathbf{EXP} = \bigcup_{k=0}^{\infty} TIME\left(2^{n^k}\right).$$

Es gilt $\mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. Die Frage, ob eine dieser Inklusionen echt ist, d.h. ob $\mathbf{P} \subset \mathbf{PSPACE}$ oder $\mathbf{PSPACE} \subset \mathbf{EXP}$ gilt, ist ein bisher ungelöstes Problem der Komplexitätstheorie. Man weiß jedoch, daß $\mathbf{P} \subset \mathbf{EXP}$ gilt.

In Kapitel 2.5 wird eine nichtdeterministische Turingmaschinen $TM = \left(Q, \Sigma, I, \boldsymbol{d}, b, q_0, q_{accept}\right)$ in zunächst unterschiedlichen Modell-Sichtweisen definiert. Entweder wird die Überführungsfunktion als partielle Funktion der Form $\boldsymbol{d} : Q \times \Sigma^k \to \mathbf{P}\left(Q \times (\Sigma \times \{L, R, S\})^k\right)$ angegeben; während der Berechnung kann dann die Turingmaschine nichtdeterministische Schritte

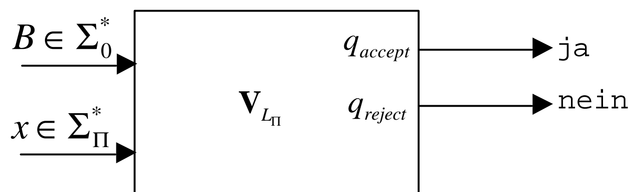
ausführen, indem sie bei Erreichen einer Konfiguration aus mehreren möglichen Folgekonfigurationen „die richtige“ auswählt. Oder die nichtdeterministische Turingmaschine wird als deterministische Turingmaschine gesehen, die mit einem „Ratemodul“ ausgestattet ist, das in nichtdeterministischer Weise zunächst eine Zusatzinformation, deren Brauchbarkeit anschließend mit Hilfe einer Überführungsfunktion der Form $\boldsymbol{d} : Q \times \Sigma^k \to Q \times (\Sigma \times \{L, R, S\})^k$ deterministisch verifiziert wird. Diese Überlegung führte schließlich auf die Definition eines Verifizierers. Die Definition wird hier noch einmal wiederholt. Da hier nur entscheidbare Probleme behandelt werden, stoppt hier der Verifizierer bei allen Eingaben.

Ein **Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π ist ein (deterministischer) Algorithmus $\mathbf{V}_{L_\Pi}$, der eine Eingabe $x \in \Sigma_\Pi^*$ und eine **Zusatzinformation** (einen **Beweis**) $B \in \Sigma_0^*$ lesen kann und die Frage „$x \in L_\Pi$?“ mit Hilfe des Beweises B entscheidet. Die Bereitstellung des Beweises B ist nicht Aufgabe des Verifizierers; B wird von außen vorgegeben.

Ein **Verifizierer** $\mathbf{V}_{L_\Pi}$ **für das Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$

Ausgabe: `ja` (accept)
`nein` (reject)



Mit $\mathbf{V}_{L_\Pi}(x, B)$, $\mathbf{V}_{L_\Pi}(x, B) \in \{$`ja`, `nein`$\}$ wird die Entscheidung von $\mathbf{V}_{L_\Pi}$ bezeichnet.

Für $x \in \Sigma_\Pi^*$ gilt die Entscheidung:
$x \in L_\Pi$, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt, so daß $\mathbf{V}_{L_\Pi}$ bei Eingabe von x und B_x mit $\mathbf{V}_{L_\Pi}(x, B_x) =$ `ja` stoppt;
$x \notin L_\Pi$, falls für alle Beweise $B \in \Sigma_0^*$ gilt: entweder stoppt $\mathbf{V}_{L_\Pi}$ bei Eingabe von x und B nicht, oder $\mathbf{V}_{L_\Pi}$ stoppt mit $\mathbf{V}_{L_\Pi}(x, B) =$ `nein`.

Der Beweis $B \in \Sigma_0^*$, den der Algorithmus $\mathbf{V}_{L_\Pi}$ zusammen mit der Eingabe $x \in \Sigma_\Pi^*$ liest, wird **auf nichtdeterministische Weise** vorgegeben. Wie er zu konstruieren ist, liegt außerhalb des Algorithmus $\mathbf{V}_{L_\Pi}$. Der Verifizierer $\mathbf{V}_{L_\Pi}$ realisiert daher einen **nichtdeterministischen Algorithmus zur Akzeptanz (Erkennung, Entscheidung)** der Menge $L_\Pi \subseteq \Sigma_\Pi^*$.

Es sei $f : \mathbf{N} \to \mathbf{N}$ eine Funktion. Falls der nichtdeterministische Algorithmus mit Hilfe des Verifizierers $\mathbf{V}_{L_\Pi}$ die ja/nein-Entscheidung bei Eingabe von $x \in \Sigma_\Pi^*$ mit $|x| = n$ nach einer Anzahl von Schritten trifft, die in $O(f(n))$ liegt, so ist der Algorithmus (der Verifizierer $\mathbf{V}_{L_\Pi}$) $f(n)$-**zeitbeschränkt**.

Zu beachten ist, daß die Laufzeit des nichtdeterministischen Algorithmus bzw. die Laufzeit des Verifizierers $\mathbf{V}_{L_\Pi}$ in Abhängigkeit von der Größe von $x \in \Sigma_\Pi^*$ gemessen wird. Natürlich ist im Zeitaufwand, den ein Verifizierer für seine Entscheidung benötigt, auch die Zeit enthalten, um die Zeichen des Beweises B zu lesen. Ist dieser $f(n)$-zeitbeschränkt, so kann er höchstens $C \cdot f(n)$ viele Zeichen des Beweises lesen (hierbei ist C eine Konstante). Wenn es also überhaupt einen Beweis B_x mit $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$ gibt, dann gibt es auch einen Beweis mit Länge $\leq C \cdot f(n)$, so daß man für den Beweis gleich eine durch $C \cdot f(n)$ beschränkte Länge annehmen kann.

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in NTIME(f(n))$, wenn es einen nichtdeterministischen $f(n)$-zeitbeschränkten Algorithmus (einen Verifizierer) zur Akzeptanz von L_Π gibt.

Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $NTIME(f(n))$" steht synonym für $L_\Pi \in NTIME(f(n))$. In diesem Sinn bezeichnet $NTIME(f(n))$ die **Klasse der Entscheidungsprobleme, die auf nichtdeterministische Weise in der Zeitkomplexität** $O(f(n))$ **gelöst werden können.**

Eine der wichtigsten Klassen, die Klasse **NP** der Entscheidungsprobleme, die nichtdeterministisch mit polynomieller Zeitkomplexität gelöst werden können, ergibt sich, wenn f ein Polynom ist:

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME(n^k)$$

In Kapitel 2.5 wird gezeigt, daß eine $T(n)$-zeitbeschränkte nichtdeterministische Turingmaschine durch eine $O(c^{T(n)})$-zeitbeschränkte deterministische Turingmaschine simuliert werden

kann. Setzt man $T(n)=n^k$, so sieht man unmittelbar, daß $NTIME\left(n^k\right) \subseteq TIME\left(2^{n^k}\right)$ gilt. Daraus folgt $\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right) \subseteq \bigcup_{k=0}^{\infty} TIME\left(2^{n^k}\right) = \mathbf{EXP}$.

Da ein polynomiell zeitbeschränkter Algorithmus $\mathbf{A}_{L_\Pi}$, wie er in der Definition von **P** vorkommt, ein spezieller polynomiell zeitbeschränkter Verifizierer ist, der für seine Entscheidung ohne die Zuhilfenahme eines Beweises auskommt, gilt

$\mathbf{P} \subseteq \mathbf{NP}$.

Insgesamt ergibt sich $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. Zusammen mit $\mathbf{P} \subset \mathbf{EXP}$ fragt sich, welche der Inklusionen echt ist. Die dabei wichtigste Frage $\mathbf{P} = \mathbf{NP}$ bzw. $\mathbf{P} \neq \mathbf{NP}$ ist bisher ungelöst (**P-NP-Problem**). Vieles spricht dafür, daß $\mathbf{P} \neq \mathbf{NP}$ gilt.

5.3 Die Klassen P und NP

Das vorliegende Kapitel behandelt Beispiele aus den Klassen **P** und **NP**.

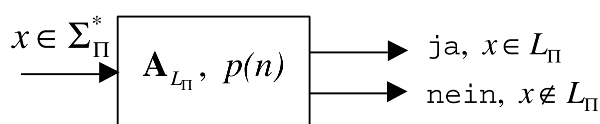
Zur Verdeutlichung des Unterschieds zwischen **P** und **NP** werden die entsprechenden Definitionen der beiden Klassen noch einmal gegenübergestellt:

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **P**, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{ja}$ gilt,
Entscheidung „$x \notin L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{nein}$ gilt.

Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.



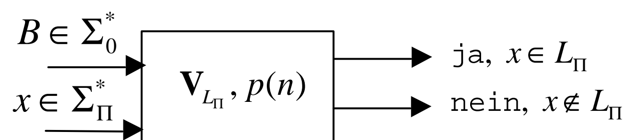
Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **NP**, wenn es einen Verifizierer $\mathbf{V}_{L_\Pi}$ und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt mit $|B_x| \leq C \cdot p(n)$ und $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$;

Entscheidung „$x \notin L_\Pi$“ falls für alle Beweise $B \in \Sigma_0^*$ mit $|B| \leq C \cdot p(n)$ gilt: $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Hierbei wird die Entscheidung $\mathbf{V}_{L_\Pi}(x, B)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.



Umgangssprachlich kann man die Klassen **P** und **NP** etwa folgendermaßen beschreiben:

Die Klasse **P** ist die Klasse der Entscheidungsprobleme, für die bei Vorgabe einer Instanz **in polynomieller Zeit** (in Abhängigkeit von der Größe der Instanz) **entschieden wird, ob die Instanz eine das Problem definierende Eigenschaft besitzt oder nicht**.

Die Klasse **NP** ist die Klasse der Entscheidungsprobleme, für die bei Vorgabe einer Instanz und eines Beweises, der zeigen soll, daß die Instanz eine das Problem definierende Eigenschaft besitzt, **der Beweis in polynomieller Zeit** (in Abhängigkeit von der Größe der Instanz) **verifiziert werden**.

Man kann die Klassen **P** bzw. **NP** auch durch deterministische bzw. nichtdeterministische Turingmaschinen charakterisieren.

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **P**, wenn es eine deterministische Turingmaschine TM_{L_Π} gibt, die bei Eingabe eines Wortes $x \in \Sigma_\Pi^*$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht. Diese Entscheidung wird in $p_{L_\Pi}(|x|)$ vielen Schritten getroffen, wobei p_{L_Π} ein

Polynom mit $p_{L_\Pi}(n) \geq n$ ist. Ist $x \in L_\Pi$, dann stoppt TM_{L_Π} im Zustand q_{accept} („ja-Zustand", „ja-Entscheidung"). Ist $x \notin L_\Pi$, dann befindet sich TM_{L_Π} nach höchstens $p_{L_\Pi}(|x|)$ in einem Zustand $q \neq q_{accept}$, für den es keinen Nachfolgezustand gibt. Dann stoppt TM_{L_Π} in einem nichtakzeptierenden Zustand („nein-Zustand", „nein-Entscheidung"). Man kann darüber hinaus annehmen, daß TM_{L_Π} nur ein Band besitzt, da jede *k*-DTM durch eine 1-DTM simuliert werden kann; diese Simulation ist von der Ordnung $O\left(p_{L_\Pi}^2(n)\right)$, also wieder polynomiell zeitbeschränkt, so daß man für die weiteren Betrachtungen dann das Polynom $C \cdot p_{L_\Pi}^2(n)$ anstelle von $p_{L_\Pi}(n)$ nimmt. Da TM_{L_Π} also nicht mehr als $p_{L_\Pi}(|x|)$ viele Schritte ausführt, werden von TM_{L_Π} auch höchstens nur die Zellen mit den Nummern 1, ..., $p_{L_\Pi}(|x|)+1$ besucht.

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **NP**, wenn die soeben beschriebene Turingmaschine mit einer nichtdeterministischen Überführungsfunktion arbeitet. Legt man dabei das in Kapitel 2.5 beschriebene Nichtstandardmodell einer Turingmaschine TM_{L_Π} mit Ratemodul zugrunde, dann kann man annehmen, daß das Band von TM_{L_Π} beidseitig unbegrenzt ist; das Eingabewort x belegt anfangs die Zellen mit den Nummern 1, ..., $|x|$, in Zelle 0 wird eine linke Markierung # geschrieben. Die Zellen mit negativen Nummern werden vom Ratemodul zur anfänglichen Erzeugung eines Beweises verwendet, der dann anschließend deterministisch verifiziert wird. In diesem Fall besucht TM_{L_Π} höchstens die Zellen $-p_{L_\Pi}(|x|)$, ..., 0, 1, ..., $p_{L_\Pi}(|x|)+1$.

Die Arbeitsweise eines nichtdeterministischen polynomiellen Entscheidungsalgorithmus kann man durch einen „normalen" deterministischen Algorithmus simulieren. Es ist jedoch bisher keine Simulation bekannt, die dabei in polynomieller Zeit arbeitet. Genauer gilt:

Satz 5.3-1:

Für jedes Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ aus **NP** mit Verifizierer $\mathbf{V}_{L_\Pi}$ und polynomieller Zeitkomplexität $p(n)$ gibt es einen deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$, der für jedes $x \in \Sigma_\Pi^*$ mit $|x| = n$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht und die Zeitkomplexität $O\left(p(n) \cdot 2^{O(p(n))}\right)$ besitzt.

Die Aussage „**P** = **NP**" würde bedeuten, daß es für jedes Problem Π aus **NP** mit polynomiell zeitbeschränktem Verifizierer $\mathbf{V}_{L_\Pi}$ sogar einen *polynomiell zeitbeschränkten* deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$ gibt, der dasselbe Ergebnis wie $\mathbf{V}_{L_\Pi}$ liefert. Das bzw. die Unmöglichkeit einer derartigen Simulation sind jedoch nicht bekannt.

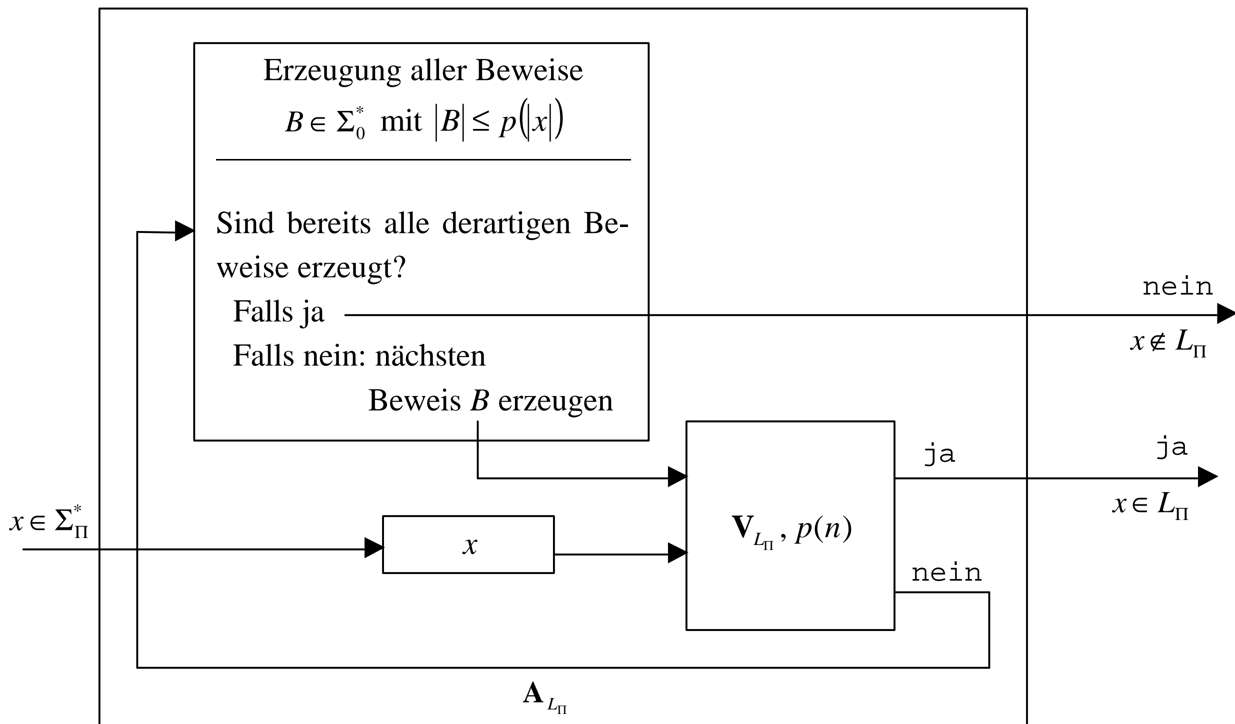
Beweis:

Der folgende Pseudocode für $\mathbf{A}_{L_\Pi}$ beschreibt die wesentlichen Aspekte der Simulation:

Bemerkung: Ein Beweis *B*, den der Verifizierer liest, ist ein Wort über einem Alphabet Σ_0 .

```
FUNCTION A_{L_Π}(x);

VAR n       : INTEGER;
    lng     : INTEGER;
    B       : Σ_0^*;
    antwort : BOOLEAN;
    weiter  : BOOLEAN;

BEGIN
  n       := size(x);        { Größe der Eingabe             }
  lng     := C * p(n);       { maximale Länge eines Beweises }
  antwort := FALSE;
  weiter  := TRUE;

  WHILE weiter DO
    IF (es sind bereits alle Wörter aus Σ_0^* mit Länge ≤ lng erzeugt worden)
    THEN weiter := FALSE
    ELSE
      BEGIN
        B:= nächstes Wort aus Σ_0^* mit Länge ≤ lng;
        IF V_Π(x,B) = ja
        THEN BEGIN
               antwort := TRUE;
               weiter  := FALSE;
             END;
      END;

  CASE antwort OF
  TRUE  : A_Π(x) := ja;
  FALSE : A_Π(x) := nein;
  END;

END;
```

Graphisch läßt sich die Simulation folgendermaßen darstellen:



///

Einige Beispiele für Probleme aus der Klasse $\mathbf{P} = \bigcup_{k=0}^{\infty} TIME(n^k)$ wurden bereits in den vorherigen Kapiteln behandelt. Dazu gehört das Problem, festzustellen, ob zwischen zwei Knoten eines gewichteten Graphen ein Pfad mit minimalem Gewicht existiert, das eine vorgegebene Schranke nicht überschreitet.

Entscheidungsprobleme leiten sich häufig von Optimierungsproblemen ab. Leider stellt sich heraus, daß für eine Vielzahl dieser Entscheidungsprobleme keine Lösungsalgorithmen bekannt sind, die polynomielles Laufzeitverhalten aufweisen. Das gilt insbesondere für viele Entscheidungsprobleme, die zu Optimierungsproblemen gehören, die für die Praxis relevant sind (Problem des Handlungsreisenden, Partitionenproblem usw.). Für diese (Optimierungs-) Probleme verfügt man meist über deterministische Lösungsalgorithmen, deren Laufzeitverhalten exponentiell in der Größe der Eingabe ist, bzw. man kann den Nachweis erbringen, daß die zugehörigen Entscheidungsprobleme in **NP** liegen. Derartige Verfahren werden als praktisch nicht durchführbar (intractable) angesehen, obwohl durch den Einsatz immer schnellerer Rechner immer größere Probleme behandelt werden können. Es stellt sich daher die Frage, wieso trotz intensiver Suche nach polynomiellen Lösungsverfahren derartige schnelle Algorithmen (bisher) nicht gefunden wurden. Seit Anfang der 1970'er Jahre erklärt eine inzwischen gut etablierte Theorie, die **Theorie der NP-Vollständigkeit**, Ursachen dieses Phänomens. Eine Einführung in diese Theorie gibt Kapitel 5.4.

Ein wichtiges Beispiel für Probleme auf der Grenze zwischen **P** und **NP** ist das Problem der Erfüllbarkeit Boolescher Ausdrücke (siehe Kapitel 1.1).

Das folgende Entscheidungsproblem liegt in **P**:

Erfüllende Wahrheitsbelegung (erfSAT)

Instanz: $[F, f]$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$, $f : V \to \{\texttt{TRUE}, \texttt{FALSCH}\}$ ist eine Belegung der Variablen mit Wahrheitswerten.

Lösung: Entscheidung „ja", falls die durch f gegebene Belegung dem Ausdruck F den Wahrheitswert `TRUE` gibt,
Entscheidung „nein", sonst.

Ein Lösungsalgorithmus setzt einfach die in der Eingabe-Instanz $[F, f]$ gelieferte Belegung f in die Formel F ein und ermittelt den Wahrheitswert der Formel gemäß den Auswertungsregeln für die beteiligten Junktoren.

Das folgende Entscheidungsproblem liegt in **PSPACE**:

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT)

Instanz: F

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$.

Lösung: Entscheidung „ja", falls es eine Belegung der Variablen von F gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert `TRUE` ergibt,
Entscheidung „nein", sonst.

CSAT liegt in **PSPACE** (und damit in **EXP)**. Für den Beweis genügt es zu zeigen, daß nacheinander alle möglichen 2^n Belegungen der n Variablen in einer Eingabe-Instanz F mit einem Speicherplatzverbrauch von polynomiell vielen Speicherzellen erzeugt, in die Formel F eingesetzt und ausgewertet werden können.

Es ist nicht bekannt, ob CSAT in **P** liegt (vieles spricht dagegen).

Die Probleme erfSAT mit einer Eingabeinstanz $[F, f]$ und CSAT mit einer Eingabeinstanz F unterscheiden sich grundsätzlich dadurch, daß bei ersterem in der Eingabeinstanz $[F, f]$ eine wesentliche Zusatzinformation, nämlich eine potentiell erfüllende Belegung f der Variablen vorgegeben ist, die nur noch daraufhin überprüft werden muß, ob sie wirklich die in der Eingabeinstanz enthaltene Formel F erfüllt. Zur Entscheidung, ob eine Eingabeinstanz F von CSAT erfüllbar ist, muß also entweder eine erfüllende Belegung f konstruiert bzw. aufgrund geeigneter Argumente die Erfüllbarkeit gezeigt werden, oder es muß der Nachweis erbracht werden, daß keine Belegung der Variablen von F die Formel erfüllt. Wenn dieser Nachweis nur dadurch gelingt, daß *alle* 2^n möglichen Belegungen überprüft werden, ist mit einem polynomiellen Entscheidungalgorithmus nicht zu rechnen. Intuitiv ist diese Entscheidungsaufgabe also schwieriger zu bewältigen, weil weniger Anfangsinformationen vorliegen, als lediglich die Verifikation einer potentiellen Lösung.

Eine einfache Überlegung zeigt, daß CSAT in **NP** liegt. Ein Verifizierer für CSAT arbeitet wie folgt: Er erhält mit einer Eingabeinstanz f (mit n Variablen) für CSAT als Zusatzinformation eine Belegung B_f der Variablen in f. Wenn f erfüllbar ist, nimmt man für B_f gerade eine erfüllende Belegung. Wenn f nicht erfüllbar ist, liefert keine Belegung B_f der Variablen den Wahrheitswert `TRUE`. Der Verifizierer überprüft lediglich (auf deterministische Weise) in $O(n)$ vielen Schritten, ob sich der Wert `TRUE` ergibt, wenn man die in B_f vorkommenden Wahrheitsheitswerte in f einsetzt; in diesem Fall wird f akzeptiert, ansonsten nicht. Insgesamt liegt polynomielles Laufzeitverhalten vor.

Da bei diesem Verfahren nicht wesentlich eingeht, ob f in konjunktiver Normalform vorliegt, sondern lediglich, ob f ein korrekter Boolescher Ausdruck und erfüllbar ist, ergibt sich, daß auch das folgende allgemeinere Problem SAT in **NP** liegt.

Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT)

Instanz: F

F ist ein Boolescher Ausdruck mit der Variablenmenge $V = \{x_1, ..., x_n\}$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von F gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert `TRUE` ergibt,

Entscheidung „nein“, sonst.

SAT liegt in **NP**.

Schränkt man das Problem CSAT auf diejenigen Booleschen Ausdrücke in konjunktiver Normalform ein, in denen jede Klausel genau 2 Literale enthält, so erhält man das Problem 2-

SAT. Entsprechend ist 3-SAT dadurch definiert, daß hier jede Klausel genau 3 Literale enthält.

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Instanz: F

$F = F_1 \wedge ... \wedge F_m$ ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{ x_1, ..., x_n \}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält.

Lösung: Entscheidung „ja", falls es eine Belegung der Variablen von F gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert `TRUE` ergibt,
Entscheidung „nein", sonst.

Man kann zeigen, daß 2-SAT in **P** liegt:

Satz 5.3-2:
Ist F eine Instanz für 2-CSAT mit n Variablen und m Klauseln, dann liefert das beschriebene Verfahren mit der Prozedur `Erfuellbarkeit_2CSAT` eine korrekte ja/nein-Entscheidung. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n \cdot m)$. Dieser Aufwand ist polynomiell in der Größe der Eingabe.

Beweis:

Ein Algorithmus zur Ermittlung einer erfüllenden Belegung kann etwa nach folgender Strategie vorgehen:

Man nehme eine bisher noch nicht betrachtete Klausel $F_i = (y_1 \vee y_2)$ von F. Falls eines der Literale während des bisherigen Ablaufs bereits den Wahrheitswert `TRUE` erhalten hat, dann wird F_i als erfüllt erklärt. Falls eines der Literale, etwa y_1, den Wert `FALSE` hat, dann erhält das Literal y_2 den Wahrheitswert `TRUE` und $\neg y_2$ den Wahrheitswert `FALSE`. F gilt dann als erfüllt. Dabei kann jedoch ein Konflikt auftreten, nämlich daß ein Literal durch die Zuweisung einen Wahrheitswert bekommen soll, jedoch den komplementären Wahrheitswert bereits vorher erhalten hat. In diesem Fall wird die vorherige Zuweisung rückgängig gemacht. Falls es dann wieder zu einem Konflikt mit diesem Literal kommt, ist die Formel nicht erfüllbar.

Der folgende Pseudocode-Algorithmus implementiert diese Strategie; aus Gründen der Lesbarkeit wird auf die exakte Deklaration der lokalen Variablen und der verwendeten Datentypen und auf deren Implementierung verzichtet.

Algorithmus zur Lösung des Erfüllbarkeitsproblems für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Eingabe: $F = F_1 \wedge ... \wedge F_m$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält, d.h. jedes F_i hat die Form $F_i = (y_{i_1} \vee y_{i_2})$

$size(F) = n$.

Der Datentyp

```
TYPE KNF_typ = ...;
```

beschreibe den Typ einer Formel in konjunktiver Normalform, der Datentyp

```
TYPE Literal_typ = ...;
```

den Typ eines Literals bzw. einer Variablen in einem Booleschen Ausdruck.

```
VAR F              : KNF_typ;
    Entscheidung : Entscheidungs_typ;

F :=
```
$F_1 \wedge ... \wedge F_m$

Verfahren: Aufruf der Prozedur

```
Erfuellbarkeit_2CSAT (F, Entscheidung);
```

Im Ablauf der Prozedur werden Variablen Wahrheitswerte `TRUE` bzw. `FALSE` zugewiesen. Zu Beginn des Ablaufs hat jede Variable noch einen undefinierten Wert; in diesem Fall wird die Variable als „noch nicht zugewiesen“ bezeichnet. Jede Klausel F_i von F gilt zu Beginn des Prozedurablaufs als „unerfüllt“ (da ihren Literalen ja noch kein Wahrheitswert zugewiesen wurde). Sobald einem Literal in F_i der Wahrheitswert `TRUE` zugewiesen wurde, gilt die Klausel als „erfüllt“.

Ausgabe: `Entscheidung = ja`, falls F erfüllbar ist,
`Entscheidung = nein` sonst.

```
PROCEDURE Erfuellbarkeit_2CSAT (F                : KNF_typ;
                                VAR Entscheidung:Entscheidungs_typ);
VAR C          : SET OF KNF_typ;
```

```
    V          : SET OF Literal_typ;
    x          : Literal_typ;
    firstguess : BOOLEAN;

BEGIN { Erfuellbarkeit_2CSAT }
  { F = F_1 ∧ ... ∧ F_m }
  C := {F_1, ..., F_m};
  erkläre alle Klauseln in C als unerfüllt;
  V := Menge der in F vorkommenden Variablen;
  erkläre alle Variablen in V als nicht zugewiesen;

  WHILE (V enthält eine Variable x) DO
    BEGIN
      x          := TRUE;
      firstguess := TRUE;
      WHILE (C enthält eine unerfüllte Klausel F_i = (y_i1 ∨ y_i2), wobei mindestens einem
              Literal ein Wahrheitswert zugewiesen ist) DO
        BEGIN
          IF (y_i1 = TRUE) OR (y_i2 = TRUE)
          THEN erkläre F_i als erfüllt
          ELSE IF (y_i1 = FALSE) AND (y_i2 = FALSE)
               THEN BEGIN
                      IF NOT firstguess
                      THEN BEGIN
                             Entscheidung := nein;
                             Exit
                           END
                      ELSE BEGIN
                             erkläre alle Klauseln in C als unerfüllt;
                             erkläre alle Variablen in V als nicht zugewiesen;
                             x          := FALSE;
                             firstguess := FALSE;
                           END;
                    END
               ELSE BEGIN
                      IF y_i1 = FALSE
                      THEN y_i2 := TRUE
                      ELSE y_i1 := TRUE;
                      erkläre F_i als erfüllt;
                    END;
        END { WHILE C enthält eine unerfüllte Klausel };
      entferne aus C die erfüllten Klauseln;
```

```
      entferne aus V die Variablen, denen ein Wahrheitswert zugewiesen wurde;
    END { WHILE V enthält eine Variable x};
  Entscheidung := ja;

END   { Erfuellbarkeit_2CSAT };
///
```

Für das zu 2-CSAT analoge Problem 3-CSAT der Erfüllbarkeit, bei dem jede Klausel genau 3 Literale enthält, ist kein polynomielles Entscheidungsverfahren bekannt. Es wird vermutet, daß es auch kein derartiges Verfahren gibt. Natürlich liegt 3-CSAT in **NP**.

Im folgenden werden einige wenige **Beispiele für Probleme** Π **aus NP** aufgeführt, von denen nicht bekannt ist , ob sie in **P** liegen. Dabei wird für Instanzen $x \in \Sigma_\Pi^*$ jeweils der Beweis $B_x \in \Sigma_0^*$ angegeben, den der Verifizierer zur `ja/nein`-Entscheidung heranzieht. Die Angabe von B_x erfolgt informell, sie muß entsprechend (beispielsweise in eine binäre Zeichenkette) übersetzt werden. Mit $size(x)$ wird die Größe einer Eingabeinstanz angegeben.

- Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) bzw. Boolescher Ausdrücke in allgemeiner Form (SAT):

  Instanz: F ist ein Boolescher Ausdruck in konjunktiver Normalform bzw. in allgemeiner Form mit der Variablenmenge $V = \{x_1, ..., x_n\}$; $size(F) = n$

  Beweis: B_F ist eine 0-1-Folge der Länge n

  Arbeitsweise des Verifizierers: Der i-te Wert in B_F wird als Belegung von x_i interpretiert, und zwar wird der Wert 0 in `FALSE` und der Wert 1 in `TRUE` übersetzt. Die so entstehende Belegung der Variablen wird in F eingesetzt und die Formel ausgewertet. Falls sich bei dieser Auswertung von F der Wert `TRUE` ergibt, wird `ja` ausgegeben, ansonsten `nein`.

- Problem des Handlungsreisenden:

  Instanz: $[G, K]$, $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{\geq 0}$; $size([G, K]) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$

  Beweis: $B_{[G,K]}$ ist eine Permutation der Zahlen 1, ..., n in Binärkodierung (mit Länge in $O(n \cdot \log(n))$)

  Arbeitsweise des Verifizierers: Überprüfung, ob die Permutation $B_{[G,K]}$ eine Rundreise in G beschreibt, deren Gewicht $\leq K$ ist. In diesem Fall wird `ja` ausgegeben, an-

sonsten `nein`. Mit Binärsuche läßt sich sogar in polynomieller Zeit überprüfen, ob es eine Rundreise mit minimalen Gewicht $\leq K$ gibt.

- Partitionenproblem:

  Instanz: $I = \{a_1, ..., a_n\}$

  I ist eine Menge von n natürlichen Zahlen; $size(I) = n \cdot \log_2(B)$ mit $B = \max\{\lceil \log(a_i) \rceil \mid i = 1, ..., n\}$

  Beweis: B_I ist eine Teilmenge von $\{1, ..., n\}$ in Binärkodierung (mit Länge in $O(n \cdot \log(n))$)

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{j \in B_I} a_j = \sum_{j \notin B_I} a_j$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- 0/1-Rucksackproblem:

  Instanz: $I = (A, M)$

  $A = \{a_1, ..., a_n\}$ ist eine Menge von n natürlichen Zahlen; $M \in \mathbf{N}$; $size(I) = n \cdot \log_2(B)$ mit $B = \max(\{\lceil \log(a_i) \rceil \mid i = 1, ..., n\} \cup \{\log(M)\})$

  Beweis: B_I ist eine 0-1-Folge $x_1, ..., x_n$ (mit Länge n)

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{i=1}^{n} x_i \cdot a_i = M$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- Problem der $\{0,1\}$-Linearen Programmierung

  Instanz: $[A, \vec{b}, \vec{c}, K]$, $A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit m Zeilen und n Spalten, $\vec{b} \in \mathbf{Z}^m$ ist ein ganzzahliger Vektor, $\vec{c} \in \mathbf{N}^n$ ist ein nichtnegativer ganzzahliger Vektor, $K \in \mathbf{N}$, $size([A, \vec{b}, \vec{c}]) = k = m \cdot n \cdot B$ mit $B = \max\{\lceil \log(e) \rceil \mid e \in A \vee e \in \vec{b} \vee e \in \vec{c}\}$

  Beweis: $B_{[A, \vec{b}, \vec{c}]}$ ist eine 0-1-Folge der Länge n (für den Vektor $\vec{x}$)

  Arbeitsweise des Verifizierers: Überprüfung, ob für $\vec{x} = B_{[A, \vec{b}, \vec{c}]}$ die Bedingungen $A \cdot \vec{x} \geq \vec{b}$ und $\sum_{i=1}^{n} c_i \cdot x_i \leq K$ gelten. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.