klarer ist diese Frage für die Band-/Speicherkomplexität beantwortet:

$$P_{SPACE} = \bigcup_{i \geq 1} DSPACE(n^i)$$

$$NP_{SPACE} = \bigcup_{i \geq 1} NSPACE(n^i)$$

Nach Satz von Savitch $NSPACE(n^i) \subseteq DSPACE(n^{2i})$

Dies ergibt sofort $P_{SPACE} = NP_{SPACE}$

Es gilt $NSPACE(\log n) \subseteq P \subseteq NP \subseteq P_{SPACE}$

Probleme mit gutem Aufwand lösbar. Probleme nur mit Trick lösbar

Dies ergibt eine Klassifizierung von Problemen in solche, die praktisch lösbar sind und solche, die aufgrund des benötigten Aufwandes (exponentielle Zunahme des Aufwandes abhängig von der Größe der Eingabe) praktisch nicht mehr lösbar sind. (Vgl. Tabelle Buch S. 302)

3.1.5. Nichtdeterminismus in Algorithmen – Backtracking

Bisher hatten wir Nichtdeterminismus im wesentlichen bei Turing-Maschinen studiert. Nun behandeln wir Nichtdeterminismus auf der Ebene von Programmiersprachen.

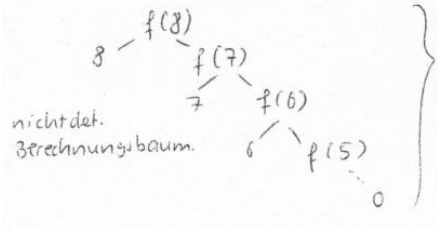
Nichtdeterminismus liegt in einem System oder Algorithmus vor, wenn bei bestimmten Schritten Wahlmöglichkeiten gegeben sind.
Achtung: Wir betrachten hier keine Annahmen über Wahrscheinlichkeiten für die Wahl der Schritte.

Für funktionale Sprachen können wir Nichtdet. durch folgende Konstruktion einführen. Wir betrachten für Ausdrücke E_1 und E_2 den nichtdet. Ausdruck $E_1 \,[]\, E_2$. Auswahlausdruck.

(Bsp. $1 \,[]\, 2$ Ausdruck dessen Wert 1 sein kann oder 2.)

Allgemein: Der Ausdruck $E_1 \,[]\, E_2$ ergibt bei Auswertung den (einen) Wert von E_1 oder den (einen) Wert von E_2

Bsp.
```
                 =f
fct wahl = (nat n) nat:
    if n=0 then 0 else n [] f(n-1) fi
```
Nichtdet. und Rekursion gemischt.



nichtdet. Berechnungsbaum.

f(8) liefert eine Zahl im Intervall von 0 bis 8. Man kann einen bel. Pfad im Berechnungsbaum auswählen, der entspricht dann einer ~~Auswa~~ Berechnung.

So kann man Nichtdeterminismus in eine funktionale Sprache bringen (durch Auswahl u. Rekursion)
Auch möglich: Wahrscheinlichkeitsverteilung für Auswahl

Bemerkung: (1) Diese Idee läßt sich auch auf Anweisungen übertragen $S_1 \,[]\, S_2$

```
fct any = (nat n) nat : n [] any(n+1)
```
any(8): Es kommt eine Zahl ≥ 8 raus.

(2) Auch die Terminierung kann von der nichtdet. Auswahl abhängen
Wir betrachten im Folgenden Algorithmen, bei denen die Terminierung gesichert ist.

Beispiel: Nichtdet. Erzeugung von Permutationen

```
fct f = (nat n) seq nat:
{ f(n) erzeugt eine Sequenz, die eine Permutation der Menge {1,...,n} ist - nichtdet
  if n=0 then ε
         else einf(f(n-1), n)
  fi
```

Prinzip $\langle p_1 \dots p_{n-1} \rangle$ Perm. von $\{1, \dots, n-1\}$
n an bel. Stelle einfügen → es entsteht

```
fct einf = (seq nat s, nat n) seq nat:
{ einf(s,n) fügt nichtdet. die Zahl n an einer bel. Stelle in s ein }
  if s = ε then <n>
         else <n> ∘ s  ▯  <first(s)> ∘ einf(rest(s), n)
  fi
```

Behauptung: f(n) erzeugt nichtdet. eine Permutation von $\{1, \dots, n\}$, wobei jede Permutation als Ergebnis auftreten kann.

Beweis: Induktion über $n \in \mathbb{N}$:
1. Fall: $n = 0$: f(n) liefert ε ✓
2. Fall: Annahme f(n-1) liefert beliebige Permutation über $\{1, \dots, n-1\}$. n wird in f(n-1) an beliebiger Position eingefügt. Dies liefert eine Permutation von $\{1, \dots, n\}$; jede Permutation kann so erzeugt werden. □

Algorithmus, der die Menge aller Permutationen erzeugt:

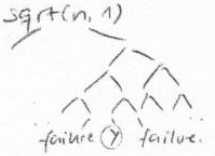
```
fct f = (nat n) set nat: if n=0 then {ε} else einf(f(n-1), n) fi
fct einf = (seq nat s, nat n) set seq nat:
  if s = ε then <n> else {<n> ∘ s} ⊂ first(s) ∘ einf(rest(s), n) fi
                                  ↖ erweitert auf Mengen
```

Bisher haben wir nur Beispiele betrachtet, wo jeder nichtdet. Berechnungspfad zu einem Ergebnis führt, das unseren Vorstellungen genügt.
Oft studieren wir nichtdet. Berechnungen (Beispiel TM), wo gewisse Pfade nicht zu einem brauchbaren Ergebnis führen. ← ≠ ⊥ Berechnung terminiert nicht.
Dazu definieren wir einen neuen Ausdruck **failure** der darstellt, dass dieses Ergebnis nicht brauchbar ist.

Beispiel:

```
fct sqrt = (nat n  nat y) nat:                    // y ist potentielles Ergebnis
  if   y² > n                then failure
  elif y² ≤ n ≤ (y+1)²       then y
                             else sqrt(n, 2*y+1) ▯ sqrt(n, 2*y)
  fi
```

sqrt(n, 1) für $n > 0$ liefert die natürlichzahlige Wurzel, wenn wir vereinbaren, dass nichtdet. Berechnungen, die mit failure enden, nicht akzeptiert werden.



Eine nichtdet. Berechnung darf nur mit failure enden, wenn alle Zweige auf failure enden.

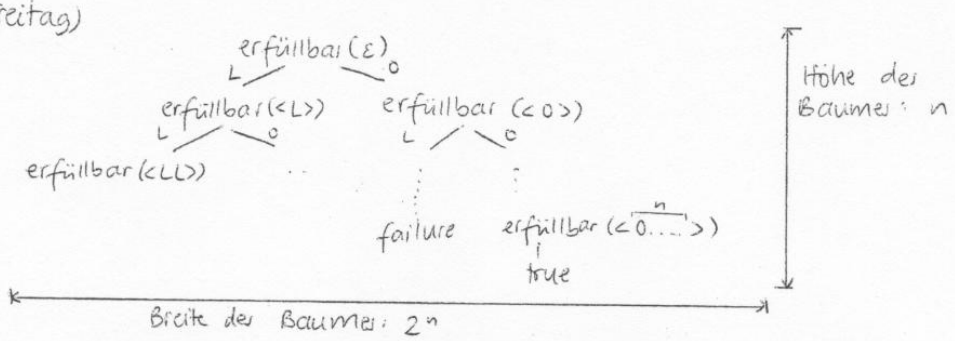
Beispiel 2: Erfüllbarkeit von aussagenlogischen Formeln
Wir betrachten eine logische Formel (Boolescher Ausdruck), der genau $x_1, \dots, x_n$ als freie Identifikatoren der Sorte Bool enthält. Der Ausdruck heißt erfüllbar, wenn es Werte $b_1, \dots, b_n \in \mathbb{B}$ gibt, so dass für $x_1 = b_1, x_2 = b_2, \dots, x_n = b_n$ der Ausdruck den Wert true liefert. Damit ist die Sequenz $\langle b_1 \dots b_n \rangle$ der Nachweis für die Erfüllbarkeit. Wir stellen den Ausdruck als Funktion dar: fct ausdruck = (seq bool b) bool: ...
Ein nichtdet. Algorithmus für die Erfüllbarkeit

```
fct erfüllbar →
```

siehe nächste Seite

```
fct erfüllbar = (seq bool s) bool:
    if |s| = n then
        if ausdruck(s) then true
                       else failure
        fi
    else erfüllbar(s ∘ <L>) [] erfüllbar(s ∘ <O>)
    fi
```

3. 6. 2003 (Freitag)



erfüllbar(ε) liefert "lauter" failure Resultate und damit insgesamt das Resultat failure falls der gegebene Ausdruck nicht erfüllbar ist, sonst aber true.

Für Programmiersprachen, die Konzepte wie nichtdet. Auswahl und "failure" anbieten, müssen spezielle Auswertungstechniken („Auswertung durch Suche") eingesetzt werden.

Stichworte: Logik Programmierung
Constraint Programmierung

z.B. Stundenplan aufstellen
Constraints: logische Beschreibungen, die Einschränkungen vorgeben, z.B. Überschneidung, HS-Gruppe, Zeit, ...

Dazu existieren eigene Theorien: (failure [] E) = E
Die Idee der nichtdet. Auswahl lässt sich weiter verallgemeinern: Wir verwenden einen Auswahloperator:

some m x : q(x) (*)

wobei q ein Prädikat sei:

fct q = (m x) bool : ...

(*) liefert einen beliebigen Wert x für den q(x) gilt. Falls ∀ m x : ¬ q(x) gilt, liefert (*) den Wert ⊥.
Dieser Operator ist nützlich um Algorithmen zu skizzieren. (hochgradig nichtdet.)

Beispiel: Hamilton-Kreis
Gegeben: Sorte {0, ..., n-1} node der Knoten im Graph
Kanten werden durch die Funktion fct g = (node, node) bool (vgl. Boolesche Matrix) beschrieben.
g(i,k) = true g.d.w. eine Kante von Knoten i zu Knoten k existiert.

Ein Hamilton-Kreis für einen Graph durch eine Teilmenge S von Knoten, ist ein geschlossener Weg, in dem alle Knoten in S genau einmal auftreten.

Rechenvorschrift für Hamiltonkreis: (⚠ hochgradig nichtdet.)

```
fct hk = (set node s, seq node q) seq node:
    if s = ∅ then
        if g(last(q), first(q)) then q                  // stelle fest ob kein geschlossen
                                else failure
        fi
    else node x = some node z : z ∈ s;
         if g(last(q), x) then hk(s \ {x}, q ∘ <x>)
                          else failure
    fi
```

Sei x aus S ($x \in S$) beliebiger Knoten aus der nichtleeren Menge S.
hk ($S \setminus \{x\}$, $\langle x \rangle$) liefert Hamilton-Kreis, falls ein solcher existiert.

3.2. NP-Vollständigkeit

Probleme in NP\P sind nach heutiger Erkenntnis nur exponentiell zu lösen. Es existieren Probleme Q in NP mit folgender Eigenschaft: jedes andere Problem in NP kann in polynomialer Zeit auf Q zurückgeführt werden.
Dann gilt: $Q \in P \Rightarrow P = NP$
Solche Probleme Q heißen NP-vollständig (NP-hart). NP-complete

3.2.1. Das Erfüllbarkeitsproblem

Ein boolescher Ausdruck ist gegeben durch eine Menge von Identifikatoren $x_1, \ldots, x_n$ der Sorte Bool und die Verknüpfungen $\neg, \wedge, \vee$ etc.
Natürlich können wir eine Darstellung von Booleschen Ausdrücken auf TM finden. Identifikatoren lassen sich durch (Binärschreibweise von) Zahlen repräsentieren.

Satz: Das Erfüllbarkeitsproblem L_{SAT} für Boolesche Ausdrücke ist NP-vollständig.

Beweisidee: (kompletter Beweis im Lehrbuch)

(1) $L_{SAT} \in NP$: Wir können eine TM konstruieren, die nacheinander alle Werte für die Identifikatoren nichtdet. festlegt und dann überprüft, ob damit der Ausdruck true hat.

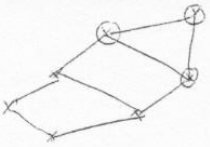
(2) Für alle Probleme $R \in NP$ existiert eine TM, die R in polynomialer Zeit in das Problem Q überführt.
Wir beschränken uns auf Probleme der formalen Sprachen, genauer, auf das Wortproblem.
Ausgangspunkt: Gegeben ist eine nichtdet. TM, die für ein gegebenes Wort in polynomialer Zeit entscheidet, ob das Wort in der Sprache ist.
Beweisidee: Wir konstruieren eine TM Ü („Übersetzer") die aus der gegebenen TM eine neue konstruiert, die dem Erfüllbarkeitsproblem entspricht, so dass der zugrunde liegende Ausdruck genau dann erfüllbar ist, wenn das Wort akzeptiert wird.
Dazu konstruieren wir einen booleschen Ausdruck, der die Konfigurationen der Ausgangs-TM durch Wahrheitswerte charakterisiert sowie die Frage, ob es sich von Konfigurationen um eine Berechnung handelt. □

3.2.2. NP-vollständige Probleme

Mittlerweise ist für eine reiche Zahl von Problemen nachgewiesen, dass sie NP-vollständig sind. Gelingt es für eines dieser Probleme einen det. polynomiell beschränkten Algorithmus anzugeben, so ist P = NP bewiesen.

Prominente Beispiele:

(1) Clique:
Gegeben: Ungerichteter Graph: V Menge der Knoten; $R \subseteq V \times V$ Menge der Kanten; $k \in \mathbb{N}$
Problem: Ex. eine Teilmenge der Kanten $C \subseteq V$ mit folgender Eigenschaft
$|C| = k$, $C \times C \subseteq R$
d.h. in C sind alle Knoten paarweise miteinander verbunden.

Bsp. 

Es ex. clique für $k=3$
Aber nicht für $k \geq 4$.

Dies entspricht der Frage, ob der Graph einen Teilgraph mit k Knoten enthält, in dem alle Knoten paarweise durch Kanten verbunden sind.

(2) Ganzzahlige Programmierung (Optimierung):
Gegeben: Ganzzahlige Matrix $A \in \mathbb{Z}^{n \times m}$
" Vektor $v \in \mathbb{Z}^n$
Gesucht: Vektor $c \in \{0,1\}^n$ so dass gilt $A \cdot c \geq v$ (elementweise)

3) Überdeckende Knotenmenge:
Gegeben: Ungerichteter Graph
Knotenmenge V
Kanten $R \subseteq V \times V$ mit $R = R^T$
$k \in \mathbb{N}$
Gesucht: Knotenmenge $U \subseteq V$ mit $|U| = k$ und U ist Überdeckung
$\forall v, w \in V: (v,w) \in R \Rightarrow \neg(v \in U \Leftrightarrow w \in U)$
Für jede Kante gilt: Ein Endpunkt ist drin und ein Endpunkt ist nicht drin
Färbung rot-blau möglich so dass jede Kante einen roten/blauen hat
Anschaulich: Es existiert eine Färbung der Knoten, so dass jede Kante einen gefärbten und einen ungefärbten Knoten verbindet.

(4) Gerichteter Hamilton-Kreis (s.o.)

(5) Problem des Handlungsreisenden
Gegeben: Menge von Knoten $V = \{1, \dots, n\}$
$dist: V \times V \to \mathbb{N}$
und eine Zahl $k \in \mathbb{N}$.
Einschränkung: Es gelte die Dreiecksungleichung
$\forall i,j,l \in \mathbb{N}: dist(i,l) + dist(l,j) \geq dist(i,j)$
Gesucht: Permutation $x_1, \dots, x_n$ der Knoten
mit $\left(\sum_{i=1}^{n-1} dist(x_i, x_{i+1})\right) + dist(x_n, x_1) \leq k$;

Folgende verwandte Problemstellungen sind oft von annähernd gleicher Komplexität
- Stelle fest, ob für ein Problem Q eine Lösung existiert
- Berechne Lösung zu Q
- Berechne „optimale" Lösung zu Q.

3.3. Effiziente Algorithmen für NP-vollst. Probleme

Auch wenn ein Problem nach heutigem Kenntnisstand nur mit exponentiellem Aufwand zu behandeln ist, ist für praktische Aufgaben die Reduzierung des Aufwandes (und sei es nur um Faktoren) von großer Bedeutung.

Im Folgenden studieren wir eine Reihe von Techniken zur Reduzierung des Aufwands.