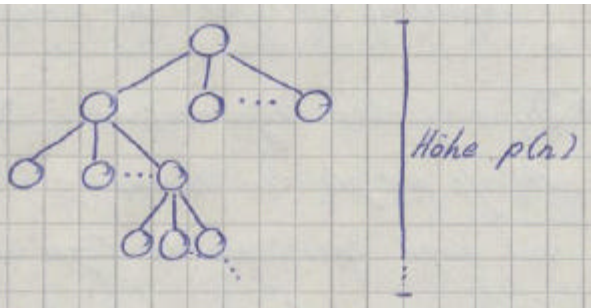


Anzahl Knoten $\sim k^{p(n)}$ $k \in \mathbb{N}$, $k > 1$

Für NP-vollständige Probleme müssen wir mit exp. Aufwand rechnen.
Dabei sind Reduktionen im Aufwand die einzige Möglichkeit, gewisse Probleme noch halbwegs vernünftig algorithmisch behandeln zu können.
Dies führt auf die Frage, wie ein vorgegebener Algorithmus effizienter gemacht werden kann.
Techniken zur Aufwandsreduzierung beim Rechnen mit großen Aufrufbäumen:

(1) Branch and Bound (geschichtes Back-Tracking):
Der Lösungsraum (als Baum angeordnet) wird geschickt organisiert:
(a) Einfach auszuwertende Zweige werden zuerst durchlaufen
(b) Frühzeitig (ohne sie ganz zu durchlaufen) Zweige (Teilbäume) können als uninteressant erkannt werden und werden nicht vollständig durchlaufen ("pruning")

(2) Approximative Verfahren: Oft sind optimale Lösungen für ein Problem gesucht (Rundreise von Handlungsreisenden mit minimaler Länge, Scheduling mit optimaler Verl., etc.) In solchen Fällen sind oft Näherungslösungen auch akzeptabel. Statt des absoluten Optimums wird eine

Näherung berechnet, die hinreichend gut ist.

(3) Dynamisches Programmieren: Treten im Aufrufbaum Aufrufe mit gleichen Parametern öfter auf, so können wir durch Tabellieren der Ergebnisse das mehrfache (aufwändige) Aufrufen vermeiden. Im Extremfall rechnen wir statt Top-Down (es wird eine Kaskade von rekursiven Aufrufen erzeugt) Bottom-Up: Wir erzeugen eine Tabelle mit Ergebnissen, indem wir die einfachen Fälle zuerst eintragen und dann schrittweise die Tabelle vervollständigen (Nachteil: Hoher Speicheraufwand, komplexe Datenstrukturen)

(4) Probabilistische Algorithmen: Diese Algorithmen verwenden Zufallszahlen (Zufallsgeneratoren)
Varianten:

(a) Algorithmen, die nur mit einer bestimmten Wahrscheinlichkeit ein korrektes oder optimales Ergebnis liefern.

(b) Algorithmen, bei denen die Terminierung innerhalb einer Zeitschranke nur mit einer bestimmten Wahrscheinlichkeit gesichert ist.

(5) Einschränkung der Problemklasse:
Häufig können eigentlich NP-vollständige Problemstellungen durch Einschränkungen auf Teilprobleme zurückgeführt werden, die Polynomial lösbar sind.

(6) Heuristische Methoden: Dies sind Methoden, die ad-hoc eingesetzt, ohne genaue Angaben warum oder wie gut sie funktionieren.

Für eine angemessene Behandlung NP-vollständiger Probleme ist besonderes Geschick des Programmierers erforderl.

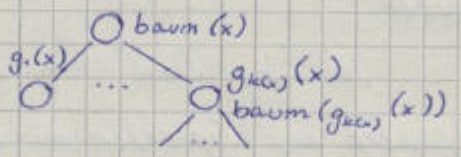
3.3.1 Geschicktes Durchsuchen von Baumstrukturen

Wir betrachten eine spezielle Problemstellung:
Wir minimieren eine Funktion

$$f: M \to O$$

wobei O eine linear geordnete Menge sei, über M. Dabei stellen wir uns vor, dass wir die Menge als Baum anordnen können.

```
fct baum = (m x) set m:
  if vollständig(n) then {x}
                    else ⋃_{i∈{1..k(x)}} baum(g_i(x))
  fi
```



baum (x) erzeugt eine Menge, die baumartig angeordnet werden kann

```
fct suchemin = (set m s) m:
  if |s| = 1 then some m x: x∈s
             else m x = some m z: z∈s;
                  m y = suchemin (s \ {x})
```

```
    if f(x) ≤ f(y) then x
                   else y
    fi
fi
Aufruf:
    suchemin (baum (x))
```

Schema umfaßt:

(1) Erfüllbarkeitsproblem:

$\mathbb{O} = \mathbb{B}$ mit true ≤ false

Sequenzen werden baumartig erzeugt!

(2) Handlungsreisender:

Permutationen lassen sich baumartig erzeugen

Dieses Verfahren ist sehr schematisch und in der Regel ineffizient.
Durch geschicktes Verbinden von Suche und Erzeugen können wir den Aufwand reduzieren.
Beispiel: Erfüllbarkeit
Gegeben:

a: Seq Bool → Bool

definiert auf Sequenzen der Länge n.
Wir nehmen an, daß wir eine Approximationsfkt. fail(a,s) besitzen, die die für alle Booleschen Sequenzen s mit $|s| \leq n$ folgende Eigenschaft hat:

$$\boxed{fail(a,s) \wedge |s \circ z| = n \rightarrow a(s \circ z) = false}$$

Folgender Algorithmus notiert dieses aus:

```
fct suche = (seq bool x) seq bool:
    if length(x) = n then x
                     else seq bool s = x◇<L>;
                          if fail(a,s) then suche(x◇<0>)
                                       else seq bool y = suche(s);
                                            if a(y) then y
                                                    else suche(x◇<0>)
                                            fi
                          fi
    fi
```

Erkenntnis, daß wir keine Chance durch Verfeinerung des ges. Satzes zu finden

Suche in anderem Baum

Erfolg in einem Teilbaum
Suche wird abgebrochen