Zum Algorithmus:
Dieser Algorithmus setzt voraus, dass das gesuchte Element im Baum ist sonst kann über das Prädikat fast die Suche im letzten else-Zweig noch verkürzt werden.

3.3.2. Alpha- / Beta-Suche

Wir betrachten 2-Personen-Spiele: Zwei Personen α und β. Die Spieler machen abwechselnd Züge, bis eine Endstellung erreicht ist.
Wir nehmen an, dass das Spiel endlich ist, das heißt dass keine unendlichen Zugfolgen existieren.
In jeder Endstellung ist ein Gewinn / Verlust für jeden Spieler festgelegt. Wir sind an einem optimalen Spiel interessiert.
Wir bilden diese Situation in Informatikstrukturen wie folgt ab:

(1) Sorten: Datenstrukturen
player Sorte der Spieler ($=\{\alpha, \beta\}$)
position Sorte der Stellungen

(2) Spielzüge
fct Z = (player, position) set position
$Z(s,p)$ beschreibt die Menge der zulässigen Züge für Spieler s in Stellung p.

(3) Endstellungen
fct t = (player, position) bool
$t(s,p)$ legt fest, ob für Spieler s die Position p eine Endstellung ist.

(4) Gewinn
fct g = (player, position) bool
$g(s,p)$ legt fest, ob die Stellung p als Endstellung einen Gewinn für Spieler s darstellt.

(5) Hilfsfunktion
fct gegner = (player) player
mit gegner(α) = β und gegner(β) = α.

Wir bezeichnen eine Stellung p als sicher für Spieler s, falls wir bei optimalem Spiel immer gewinnen.

Rechenvorschrift:

position terminal

Position ist sicher, falls es eine Stellung $q \in Z(s,p)^{sic}$ gibt, die sicher ist für den Gegner

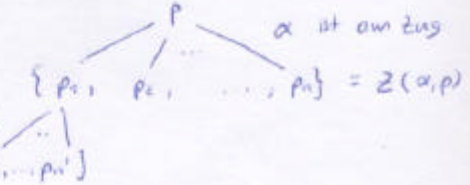
```
fct sicher = (player s, position p) bool:
    if   t(s,p)  then  g(s,p)
                 else  ∃ q ∈ Z(s,p): ¬ sicher(gegner(s), p)
fi
```

$\neg$ sicher(gegner(s), q) = {falls $\neg t$(gegner(s), q)}

$\neg \exists q' \in Z$(gegner(s), q):
  $\neg$ sicher(gegner(gegner(s)), z')) =

$\forall q' \in Z$(gegner(s), q): $\neg \neg$ sicher(gegner(gegner(s)), q')) =

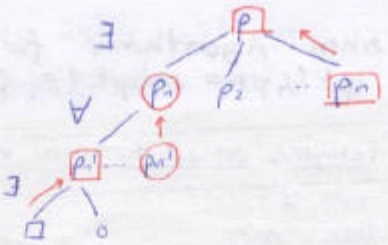
$\forall q' \in Z$(gegner(s), q): sicher(s, q')



d.h. eine Stellung ist für meinen Gegner sicher, wenn alle möglichen Stellungen die er erreichen kann für mich sicher sind.

□ Gewinn ○ Verlust

Auf ∃-Ebene setzen sich die Gewinn-Stellung nach oben durch, auf ∀-Ebene setzen sich die Verlust-Stellungen nach oben durch.



Nun betrachten wir ein leicht verändertes Problem, indem wir annehmen dass in jeder Endstellung p für einen Spieler s durch folgende geänderte Gewinn-Funktion festgelegt wird, wieviel der gewinnt (bzw. verliert):

fct g = (player, position) int

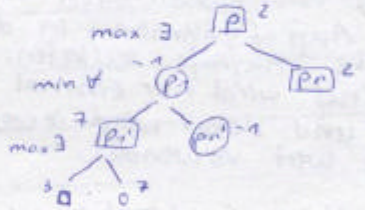
Der Rest der Deklaration, Sorten, Spielzüge, Endstellungen ~~relax~~ und Hilfsfunktion können beibehalten werden.
Die Rechenvorschrift (siehe Funktion „sicher") muss allerdings geändert werden.

```
fct opt = ( player s, position p) int :
    if t(s,p) then g(s,p)
              else max {- opt(gegner(s), q) : q ∈ Z(s,p)}
fi
```

optimaler Gewinn

Umformung:

$$-opt(gegner(s), q) = \{\text{falls } \neg t(gegner(s), q) \text{ gilt}\}$$
$$= -\max\{-opt(gegner(gegner(s)), q') : q' \in Z(gegner(s), q)\}$$
$$= \min\{opt(s, q') : q' \in Z(gegner(s), q)\}$$



Wir betrachten nun Optimierungsmöglichkeiten. Wir unterstellen, dass wir eine Abschätzung für unseren Gewinn

$opt(s,p) \leq \max opt(s,p)$

↳ Abschätzung / obere Schranke für den erzielbaren Gewinn.

Idee:
wir suchen nun ein Spielergebnis relativ zu vorgegebenen Schranken.
bei Schranke = ∞ zählt der echte Gewinn / Verlust, sonst
z.B. Schranke auf 3 setzen: Dann wird bei Gewinn > 3 nur 3 ausgezahlt, bei Verlust > 3 ebenso nur ein Verlust von 3 berechnet

Dazu definieren wir eine Sorte

eint mit Werten auf $\mathbb{Z} \cup \{-\infty, +\infty\}$ (eint steht für „erweiterter int")

Festlegung
$\max \emptyset = -\infty$
$\max(M \cup N) = \max(\max M, \max N)$
$\min \emptyset = +\infty$

Sei M eine Menge von Stellungen. Sei ferner $a = \max\{opt(s,p) : p \in M\}$
Wir definieren nun eine neue Funktion

fct setopt = (player, setposition, eint min, max) eint:

Ein Aufruf setopt(s, M, min, max): (sei min ≤ max)

$$= \begin{cases} \max & \text{falls } \max \leq a \\ a & \text{falls } \min \leq a \leq \max \\ \min & \text{falls } a \leq \min \end{cases}$$

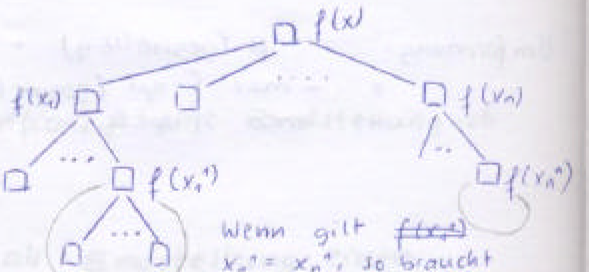
Wir geben einen Algorithmus für setopt an, der effizienter ist als opt.
Wichtig: $opt(s,p) = setopt(s, \{p\}, -\infty, +\infty)$ Einbettung:

```
fct setopt = (player s, set position m, eint min, max) eint:
    if m = ∅
    then min
    else position p = some position q : q ∈ m;                    maxopt: Abschätzung
         if maxopt(s,p) ≤ min   {d.h. opt(s,p) ≤ maxopt(s,p) ≤ min}
         then setopt(s, m \ {p}, min, max)
         else eint a = if t(s,p) then g(s,p)
                       else - setopt(gegner(s), z(s,p), - max, - min)
                       fi
              if a ≥ max then max          / andere Stellungen untersuchen
              elif a > min then setopt(s, m \ {p}, a, max)   neues min -> Intervall verkleinert sich
                           else setopt(s, m \ {p}, min, max)
              fi
         fi
    fi
```

27. 6. 2003 (Fr)

3.3.3. Dynamisches Programmieren

Idee:
Optimierung von Algorithmen mit Rekursions-Aufrufbäumen, in denen viele gleiche Aufrufe auftreten.
Jeder Aufruf wird nur einmal ausgewertet, tabelliert und dann wird jeweils der ermittelte Wert verwendet.



Wenn gilt ~~f(x..)~~ $x_1' = x_n''$, so braucht $f(x_1')$ bzw. $f(x_n'')$ nur einmal berechnet zu werden

Nachteil: Hoher Speicherplatzbedarf.
Algorithmen werden komplizierter.

Beispiel: Handlungsreisenden-Problem

Rundreise: entspricht Permutationen

Zur Übung: Programm das Permutationen erzeugt schreiben

```
fct perm = (seq node s, set node m) set seq node:
    if m = ∅ then {s}
          else node x = some node z : z ∈ m;
               ⋃             perm(insert(s, x, i), m \ {x})
          0 ≤ i ≤ length(s)
    fi
```

insert(s,x,i) sortiert den Knoten x in die Sequenz s nach Position i ein.
perm(ε, M) erzeugt die Menge aller Permutationen der Menge M.

Sei eine Abstandsfunktion gegeben: fct d = (node i, j) nat:
"Abstand zwischen i und j"

Länge einer Reise (gegeben durch Sequenz von Knoten):

```
fct sd = (seq node s) nat:
    if s = ε then 0
    elif rest(s) = ε then 0
                   else d(first(s), first(rest(s))) + sd(rest(s))
    fi
```

Aufgabe des Handlungsreisenden: Minimiere sd(s) + d(first(s), last(s))
über $s \in$ perm (ε, v) für Knotenmenge $v \neq \emptyset$.
(* Menge d. Permutationen über v)

|v|! → Aufwand steigt um ein Vielfaches bei Hinzunahme einer weiteren Stadt. (siehe Tab. n!)

n	n!	n^2	2^n	$n^2 2^n$
1	1	1	2	2
2	2	4	4	16
3	6	9	8	72
4	24	16	16	256
5	120	25	32	800
6	720	36	64	2304
7	5040	49	128	6272
8	40320	64	256	16384
9	362880	81	512	41472
10	3628800	100	1024	102400
11	39916800	121	2048	247808

Abb. Wachstumsverhalten der Funktionen $n!$, n^2, 2^n und $n^2 2^n$

```
fct mintour = (set node m, node i : i ∈ m) nat:
```
(berechnet von einem gegebenen Anfangsknoten x_0 aus alle Wege von x_0 nach i über alle Knoten in m, wobei $i \in m$)

$\min\{sd(\langle x_0 \rangle \circ s) : s \in \text{perm}(\varepsilon, m) \wedge \text{last}(\langle\rangle) = i\}$

Problem des Handlungsreisenden für Knotenmenge m und Ausgangspunkt $x_0 \notin m$:

$$\min_{x \in m} \{\text{mintour}(m, x) + d(x, x_0)\}$$

(Länge d. kürzesten Weges von x_0 nach x über alle Knoten in m; Rückkehr zu x_0)

Rechenvorschrift für mintour: (→ kürzester Weg von x_0 nach i, wobei alle Knoten durchlaufen werden)

```
fct mintour = (set node m, node i : i ∈ m) nat:
    if |m| = 1 then d(x0, i)        // d(x0,i) Abstand von x0 zu i
    else min{mintour(m\{i}, k) + d(k, i)}   // k ∈ m\{i}
    fi
```
(↘ Reduktionsprinzip: die Menge wird immer um 1 verkleinert)

Dieser Algorithmus hat folgende Reduktionsstruktur:

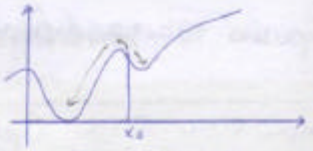
Eine Analyse zeigt, daß in jeder Schicht des Aufrufbaums nur $\prod_{n-t+1}^{n}$ verschiedene Aufrufe auf. (t)
Dies liefert $n^2 2^n$ verschiedene Aufrufe. Das heißt ein Berechnungsaufwand $O(n^2 2^n)$ gegenüber $O(n!)$ beim ursprünglichen Algorithmus.
(Siehe obige Abb. Zum Wachstumsverhalten der Funktionen: bei n = 11 statt 40 Mio Aufrufe nur 250.000)

3.3.4. Greedy-Algorithmen

Beim Handlungsreisenden-Problem: Meist sind greedy-Algorithmen nicht sinnvoll – es würde immer die nächstgelegene Stadt gewählt und dadurch evtl. günstigere Wege übersehen. Nur effektiv, wenn best. Annahmen über die Anordnungen der Städte zu treffen.
Greedy-Algorithmen meist nur für spezielle Probleme effektiv.

greedy = „gierig"

Bsp: Ziel ist das Tal. Bei Start an x_0 wandert der greedy-Alg. nach rechts ins Tal obwohl weiter links ein tieferes Tal wäre (nach einer kurzen Anfangs-Zeit)



Greedy (engl. für „gierig") Algorithmen arbeiten nach dem Prinzip der lokalen Optimierung.
Idee: Wähle den nächsten Schritt (lokal) optimal. Dies führt nicht immer zu einem globalen Optimum. (siehe Zeichnung oben)

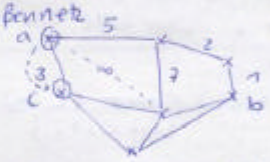
Für gewisse Problemklassen führt lokale Optimierung jedoch zu einer globalen Optimierung.

Beispiel: Dijkstras Algorithmus der kürzesten Wege

Knotenmenge $V = \{1, \dots, n\}$; Abstandsfunktion $d: V \times V \to \mathbb{N} \cup \{\infty\}$ — ∞ steht für keinen Weg
Wir suchen die Länge des kürzesten Weges zwischen zwei Punkten.

Greedy-Alg. würde c wählen.
Sei $a \to x_1 \to x_2 \to \dots \to b$ kürzester Weg
1. Fall: $c \in \{x_1, \dots, x_n\}$ liegt also auf d. kürzesten Weg
$\Rightarrow x_1 = c$
2. Fall: $c \notin \{x_1, \dots, x_n\}$

Bsp.) Straßennetz



Suche Weg von a nach b (muss nicht alle Knoten beim durchlaufen), und zwar kürzesten

Wir betten das Problem ein: Wir betrachten folgende Aufgabe:
Gegeben: Knoten x,y Knotenmenge $m \subseteq V$
Gesucht: kürzester Weg von x nach y $\quad x \to x_0 \to x_2 \to x_3 \to \dots \to x_n \to y$
wobei $x_1, \dots, x_n \in m$

```
fct dijkstra = (set node m, node x,y) enat:
  if m = ∅ then d(x,y)        ← nächstliegender Knoten zu x in der Menge m
  else node c = some node z : z ∈ m ∧ ∀ node b : b ∈ m ⇒ d(x,z) ≤ d(x,b);
       min(dijkstra(m\{c}, x,y), d(x,c) + dijkstra(m\{c}, c,y))
fi
```

c liegt nicht auf dem kürzesten Weg — Strecke bis c — Strecke von c bis y

Eine effiziente Implementierung dieses Algorithmus arbeitet mit Techniken der dynamischen Programmierung auf einer Matrix, die jeweils die Wege speichert zwischen Knoten, soweit nur Knoten in einer bestimmten vorgegebenen Menge als innere Knoten verwendet werden.

Größenordnung $O(n^3)$: viel effizienter als Durchprobieren aller Wege!