

Einführung in die KI

Prof. Dr. sc. Hans-Dieter Burkhard

Vorlesung Winter-Semester 2003/04

1. Suchverfahren, Teil1:

Grundbegriffe

Problemlösen

Zustandsraumsuche

Evolutionäre Algorithmen

Durch Suche (vielleicht) lösbare Probleme

Suche nach einem günstigen Schachzug

*Edgar Allan Poe:
Maelzel's Chess-Playing
Machine,
Southern Literary
Messenger, April 1836.*

Arithmetical or algebraical calculations are, from their very nature, fixed and determinate. Certain data being given, certain results necessarily and inevitably follow. These results have dependence upon nothing, and are influenced by nothing but the data originally given...

But the case is widely different with the Chess-Player. With him there is no determinate progression. No one move in chess necessarily follows upon any one other...

It is quite certain that the operations of the Automaton are regulated by mind and by nothing else. Indeed this matter is susceptible of a mathematical demonstration, a priori.

Durch Suche (vielleicht) lösbare Probleme

Suche nach dem (optimalen) Weg zu einem Ziel

Suche nach einem günstigen Spielzug

Suche nach optimalen Parametern

Suche nach einem Fehler (Diagnose)

Suche nach einem Dokument

Suche nach einer Antwort, einem Beweis, ...

Problem:

Was genau möchte der Kunde?

Nächste Verbindung?

Kürzeste Verbindung?

Billigste Verbindung?

Wenig Umsteigen?

Suche als Methodik in der Informatik

Suche als Wiederfinden

- Datenbank
- Suchmaschine

Suche als Problemlösen

- Existiert eine Lösung?
- Finde eine Lösung.
- Was ist die beste Lösung? (Optimierung)

Suche als Wiederfinden

Indexstrukturen

- Suchbäume
- Hashtabellen
-

- Datenbanken
- Suchmaschinen
- Textsammlungen
- Knowledge-Management

Weiterführende Fragen

Suche nach ähnlichen Begriffen

Suche nach Inhalten

Erinnern als Wiederfinden oder
Erinnern als Rekonstruktion?

Suche als Problemlösen

Theoretische Informatik:

- Prinzipielle Lösbarkeit (Berechenbarkeit)
- Komplexitätsabschätzungen

z.B. Suche in Graphen

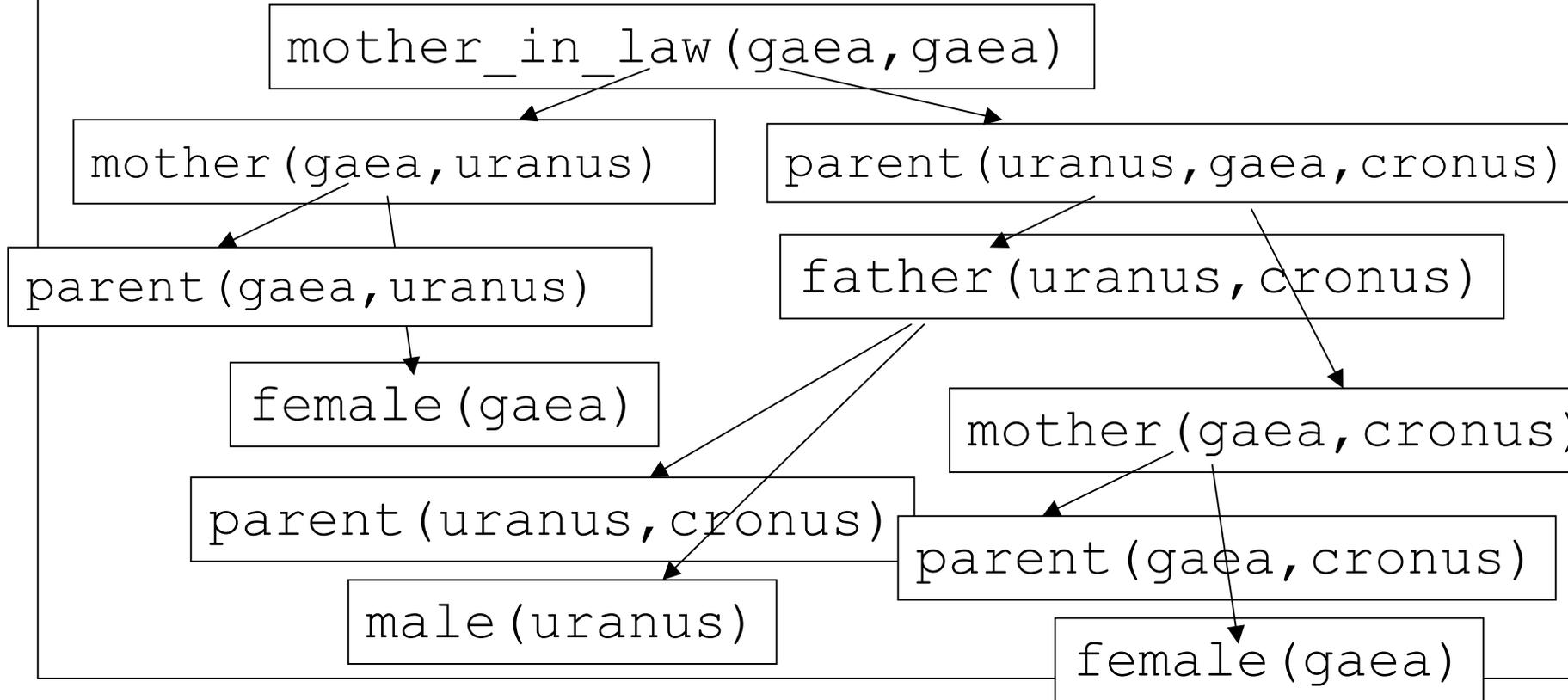
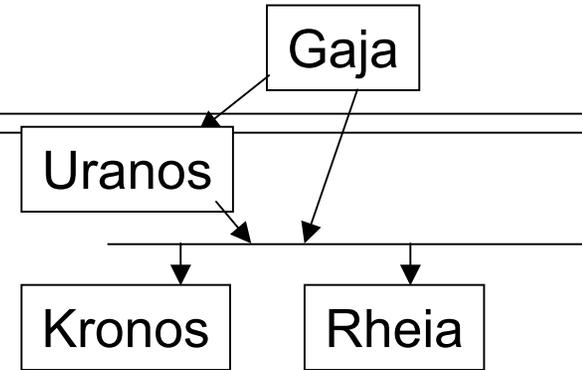
- Wege,
- Rundreise,
- Zyklen
- spannende Bäume
-

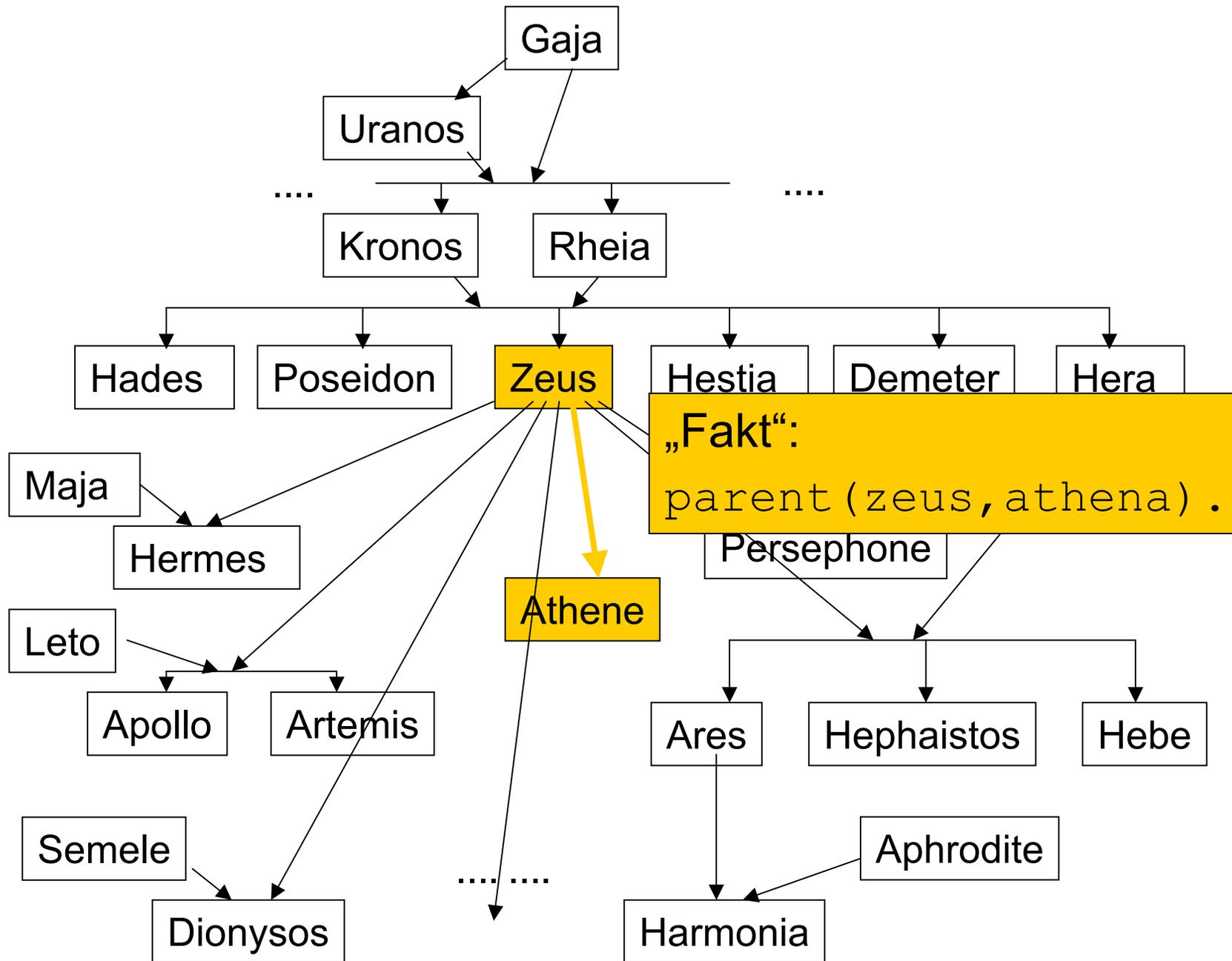
Praktische Informatik, z.B.

- Compilerbau
- Optimierung
- Theorembeweiser, PROLOG

Beweis in Prolog

```
?- mother_in_law(gaea, gaea).  
Yes.
```





Prolog-Programm:Fakten

```
parent(uranus, cronus).  
parent(gaea, cronus).  
parent(gaea, rhea).  
parent(rhea, zeus).  
parent(cronus, zeus).  
parent(rhea, heras).  
parent(cronus, heras).  
parent(cronus, hades).  
parent(rhea, hades).  
parent(cronus, hestia).  
parent(rhea, hestia).  
parent(zeus, hermes).  
parent(maia, hermes).  
...
```

```
female(gaea).  
female(rhea).  
female(heras).  
female(hestia).  
female(demeter).  
female(athena).  
female(metis).  
female(maia).  
female(persephone).  
female(aphrodite).  
female(artemis).  
female(leto).
```

```
male(uranus).  
male(cronus).  
male(zeus).  
male(hades).  
male(hermes).  
male(apollo).  
male(dionysius).  
male(hephaestus).  
male(poseidon).
```

Prolog-Programm: Regeln

```
father (X, Y) :-parent (X, Y) , male (X) .  
mother (X, Y) :-parent (X, Y) , female (X) .
```

```
parent (X, Y, Z) :-father (X, Z) , mother (Y, Z) .
```

```
son (X, Y) :-parent (X, Y) , male (Y) .
```

```
grandfather (X, Z) :-father (X, Y) , parent (Y, Z) .  
grandmother (X, Z) :-mother (X, Y) , parent (Y, Z) .
```

```
grandchild (X, Y) :-grandfather (Y, X) .  
grandchild (X, Y) :-grandmother (Y, Z) .
```

Prolog-Programm



Fahrplan-Fakten

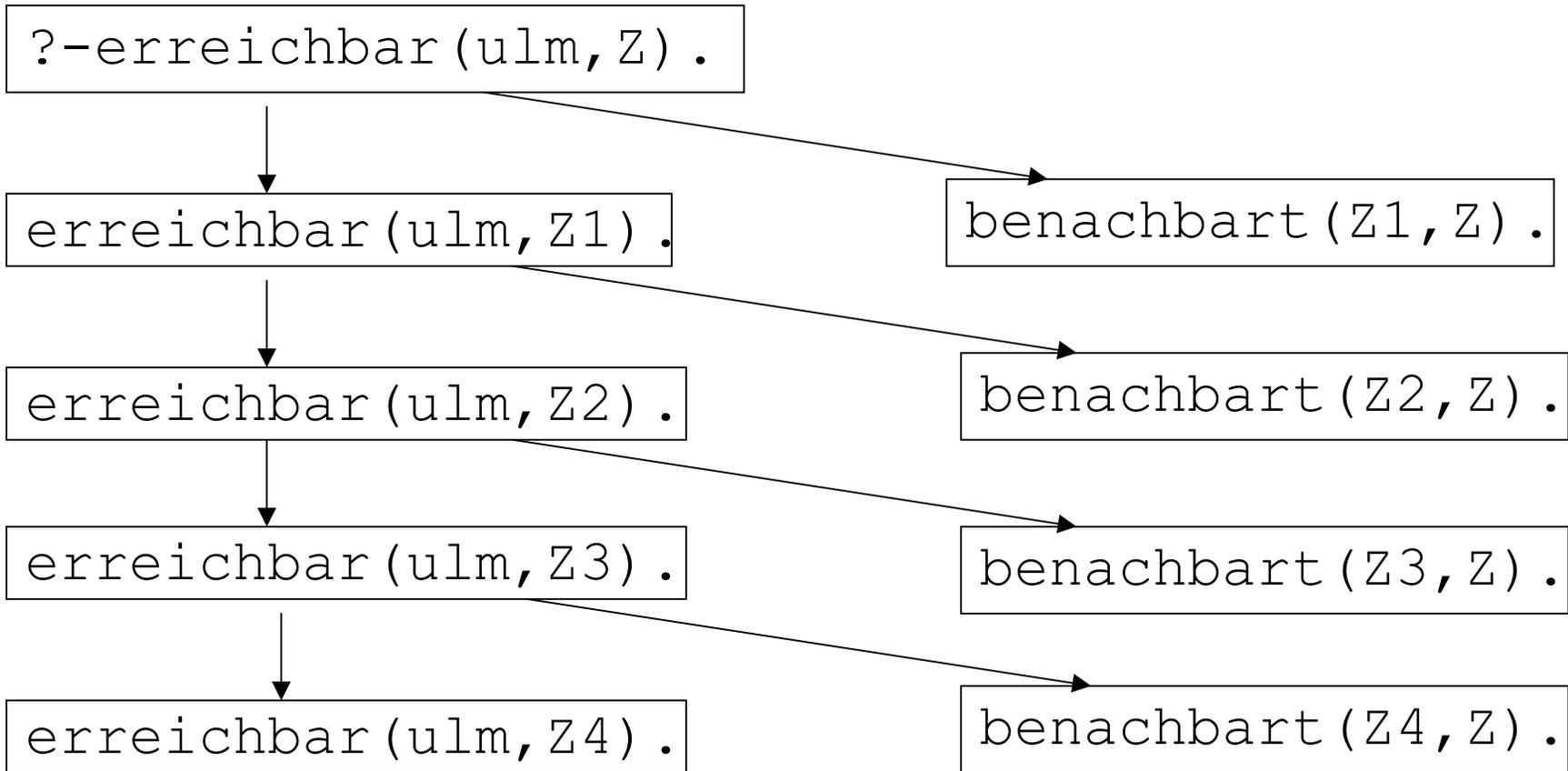
```
s_bahn(alexanderplatz,jannowitzbrücke,6:09,6:11,103).  
s_bahn(jannowitzbrücke,ostbahnhof,6:11,6:13,103).  
...
```

Suchprogramm

```
erreichbar(Start,Ziel,Zeit)  
:- s_bahn(Start,Zwischenziel,Abfahrt,Ankunft,_),  
   erreichbar(Zwischenziel,Ziel,Zeit1),  
   addiereZeit(Zeit1,Ankunft,Abfahrt,Zeit).  
...
```

Unendliche Beweisversuche

(X, Z) , benachbart (Z, Y) .



Prolog: Deklarative vs. prozedurale Semantik

Unterschiedliche Resultate bei
deklarativer und prozeduraler Semantik

?-erreichbar(ulm, Z) .

erreichbar(ulm, Z1) .

benachbart(Z1, Z) .

erreichbar(ulm, Z2) .

benachbart(Z2, Z) .

erreichbar

erreichbar(X,Y) :- erreichbar(Z,Y), benachbart(X,Z).
erreichbar(X,X).

erre

In Prolog: Links-rekursive Klauseln vermeiden

1.1 Problemlösen

Gegeben ein Problem p

Varianten:

(L): Finde Lösung(en) l des Problems p

(O): Finde Optimale Lösung(en) l_{opt} des Problems p

(E): Existieren Lösungen des Problems p ?

Menge aller Probleme: P (Problemraum)

Menge aller Lösungskandidaten: M (Lösungsraum)

Parameter-Räume

Problembeschreibung

Parameter-Raum $W_1 \times W_2 \times \dots \times W_n$

Bewertungskriterium

Lösung: $[w_1, \dots, w_n] \in W_1 \times W_2 \times \dots \times W_n$

– Muss Bewertungskriterium erfüllen

Zustandsräume

Problembeschreibung

Graphen, Bäume [Z , Op , z_{initial} , Z_{final}]

mit Zuständen Z

Operatoren $Op \subseteq Z \times Z$

Anfangszustand $z_{\text{initial}} \in Z$

Zielzuständen $Z_{\text{final}} \subseteq Z$

evtl. Kostenfunktion für Operatoren

Lösung z.B.

- Zielzustand $z \in Z_{\text{final}}$
- (kürzester) Weg zu einem Zielzustand

Spiele, Problemzerlegungen

Transformationen
zwischen den
Repräsentationen
sind möglich

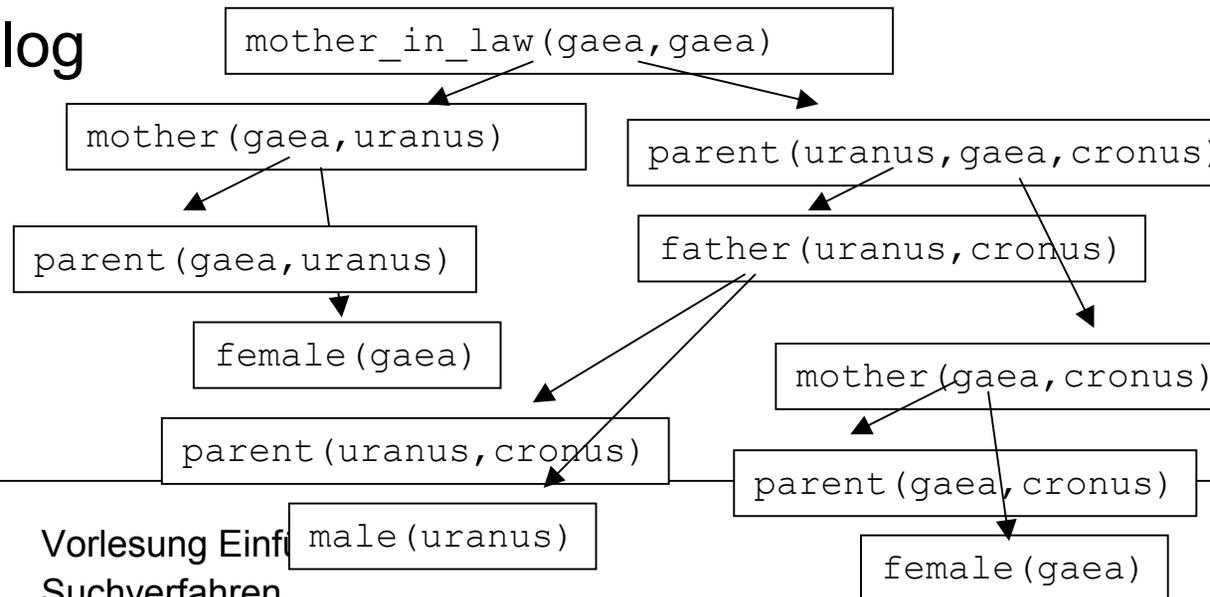
Problembeschreibung:

Und-Oder-Bäume, Spielbäume

Prologprogramm und -anfrage

Lösung z.B.

- Gewinnsituation bzw. maximaler Gewinn
- Gewinnstrategie
- Beweis in Prolog



Problemlösen: Lösungsmengen

Lösungen bzgl. (L) : $L : P \rightarrow 2^M$

Menge aller Lösungen für p : $L(p) \subseteq M$

Kriterium ϕ_p für Lösungen: $\phi_p(m) = \text{true}$ gdw. $m \in L(p)$
 $L(p) = \{ m \mid \phi_p(m) = \text{true} \}$

Lösungen bzgl. (E) : $E : P \rightarrow \{ \text{ja}, \text{nein} \}$

$E(p) = \text{ja} \iff L(p) \neq \emptyset$

Problemlösen: Lösungsmengen

Lösungen bzgl. (O) : $O: P \rightarrow 2^M$

Menge der optimalen Lösungen für p : $O(p) \subseteq M$

Vergleichskriterium:

(Halb)Ordnung bzgl. Optimalität: $R_{>} \subseteq M \times M$

mit $R_{>}(m_1, m_2)$ gdw. m_1 echt besser als m_2

$O(p) = \{ m \mid \phi_p(m) = \text{true}$

$\& \neg \exists m' \in M (\phi_p(m') = \text{true} \& R_{>}(m', m)) \}$

Problemlösen: Lösungsverfahren

Lösungsverfahren: $V: P \rightarrow 2^M$

Durch V für p ermittelte Lösungen: $V(p) \subseteq M$

Algorithmen

- Berechnen
- Manipulation von Ausdrücken
- zufälliges Probieren
- systematisches Probieren: Suchen

Vollständigkeit, Korrektheit

Für beliebige Lösungsverfahren $V: P \rightarrow 2^M$ bzgl. (L) gilt:

V ist *vollständig*, falls $L(p) \subseteq V(p)$

V ist *korrekt*, falls $V(p) \subseteq L(p)$

Für beliebige Lösungsverfahren $V: P \rightarrow 2^M$ bzgl. (O) gilt :

V ist *vollständig*, falls $O(p) \subseteq V(p)$

V ist *korrekt*, falls $V(p) \subseteq O(p)$

Ein Lösungsverfahren $V: P \rightarrow \{ \text{ja, nein} \}$ bzgl. (E) ist
korrekt, falls $V(p) = E(p)$

Systematisches Suchen

Alle Lösungskandidaten prüfen
(Lösungsraum M durchmustern)

Parameter-Räume

- *Parametersatz* $[w_1, \dots, w_n] \in W_1 \times W_2 \times \dots \times W_n$

Zustandsräume

- *Zielzustand* $z \in Z_{\text{final}} \subseteq Z$
- (*kürzester*) *Weg* zu einem Zielzustand

Spiele, Problemzerlegungen:

- *Gewinnsituation* bzw. maximaler Gewinn
- *Gewinnstrategie*

Probiervverfahren: „British-Museum-Procedure“

Generate and Test für (L):

korrekt

```
for all  $m \in M$  do
  if  $\phi_p(m) = \text{true}$  then stop(  $m$  )
stop( „no solution“ )
```

Komplexität abhängig von $\text{card}(M)$, $\text{compl}(\phi_p)$

Verfahren ist *korrekt*.

Vollständiges Verfahren durch leichte Modifikation.

Generate and Test für (E) analog .

Probiervverfahren: „British-Museum-Procedure“

Generate and Test für (O):

```
best-found := *           // * ∉ M, R>(m, *) f.a. m ∈ M
for all m ∈ M do
    if φp(m) = true & R>(m, best-found)
    then best-found := m
if best-found = { * }
then stop( „no solution“ ) else stop( best-found )
```

NP-Probleme

Komplexität abhängig von $\text{card}(M)$, $\text{compl}(\phi_p)$, $\text{compl}(R_{>})$
Verfahren ist *korrekt*.

Vollständiges Verfahren durch leichte Modifikation.

Lösungsraum M als Suchraum

- Systematik bzgl. „Generate“
 - Reihenfolge
 - Vollständigkeit der Erfassung
 - Wiederholungen vermeiden
- Heuristiken
 - Steuerung der Reihenfolge
 - Problem lokaler Kriterien
 - Beschränken (Abschneiden) des Suchraums
 - Problem Vollständigkeit/Korrektheit

1.2 Zustandsraum-Suche

Graph: $G = [Z, Op, z_{\text{initial}}, Z_{\text{final}}]$

Knoten Z („Zustände“)

Kanten $Op \subseteq Z \times Z$ („Operatoren“)

$\text{card}(Op(z, \cdot))$ endlich f.a. $z \in Z$

= Suche in Graphen

Suche in Graphen

Beispiele:

- Routenplanung
- Fahrplanauskunft
- Suche nach einem Beweis
- Suche nach Gewinnstrategie
- Planung

Modell für Problemlösen:

- Gegeben:
 - Graph $G = [V, E]$
 - „Anfangszustand“ $z_0 \in V$
 - Menge von „Zielzuständen“ $Z_f \subseteq V$

- Aufgaben:
 - Existiert ein Weg von z_0 zu einem $z_f \in Z_f$
 - Konstruiere einen Weg von z_0 zu einem $z_f \in Z$
 - Konstruiere optimalen Weg von z_0 zu einem $z_f \in Z_f$
(bzgl. eines gegebenen Optimalitätskriteriums)

Planung: Modellierung als Graph

Mögliche Aktionen: $A = \{a_1, \dots, a_n\}$

Zustände (Knoten im Graphen):

V = durch Aktionen entstehende Situationen

Ausgangssituation: Anfangszustand z_0

Situationen, in denen Planungsziel erreicht ist: Zielzustände Z_f

Zustandsübergänge (Kanten im Graphen):

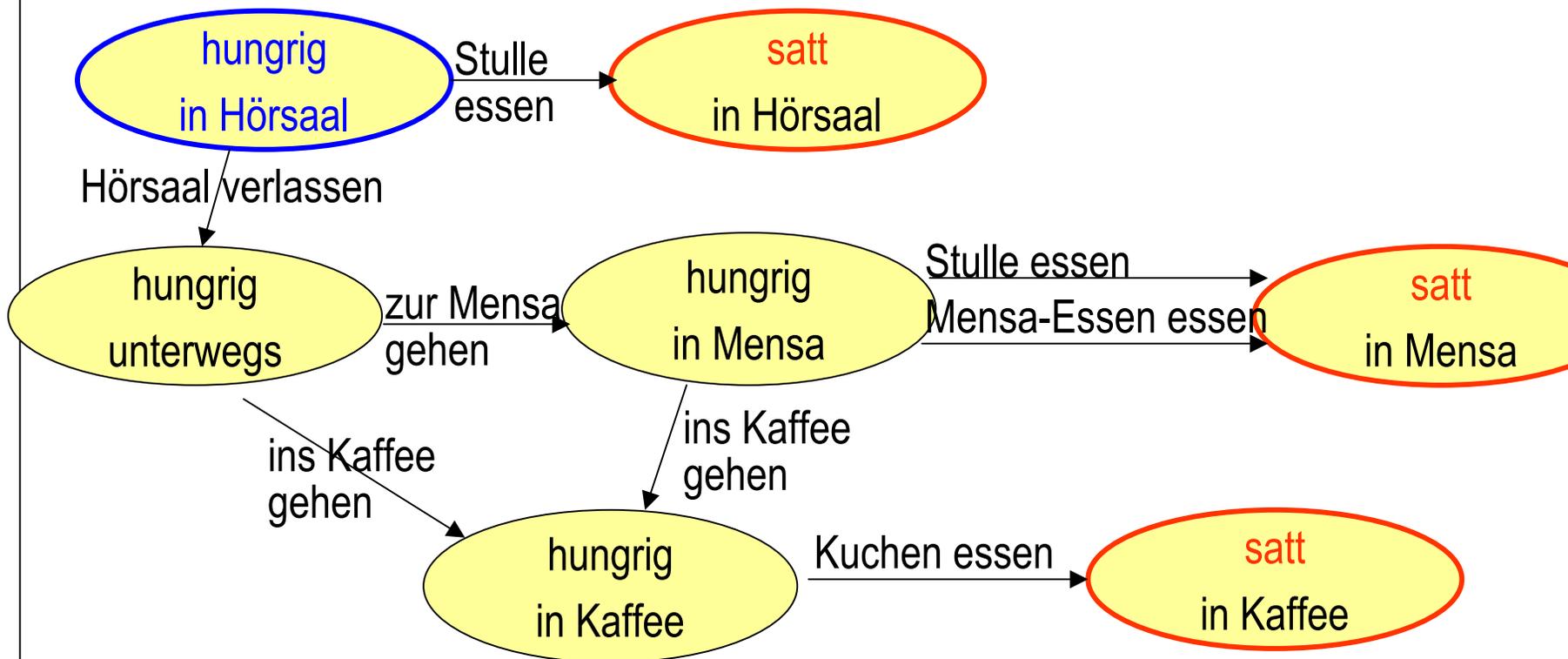
E = Übergänge zwischen Situationen durch Aktionen

G ist ein Kanten-beschrifteter Graph mit Mehrfachkanten

Planung: Modellierung als Graph

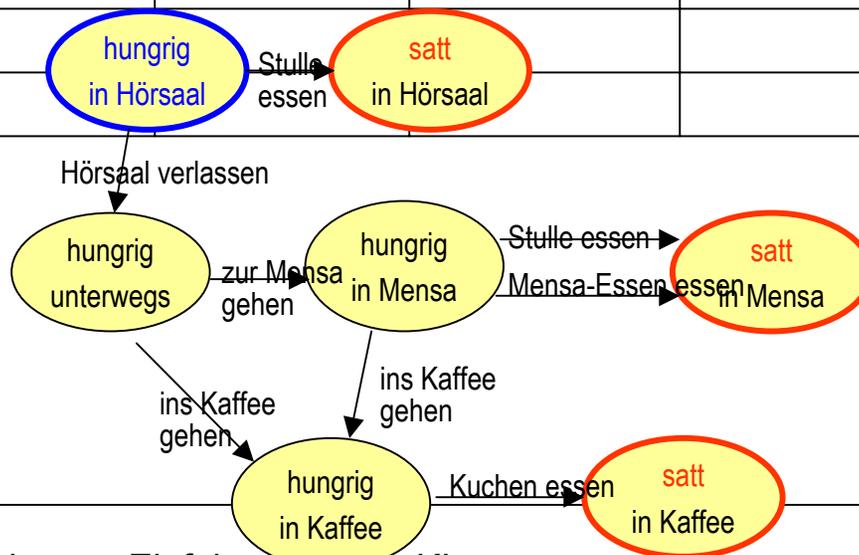
Ausgangszustand: *hungrig, im Hörsaal*

Bedingung an Zielzustände: *satt*



Übergangsmatrix

	Stulle essen	Hörsaal verlassen	zur Mensa gehen	Mensa-Essen essen	ins Kaffee gehen	Kuchen essen
hungrig, in Hörsaal	s., i.H.	h., u.				
hungrig, unterwegs			h., i.M.		h., i.K	
satt, in Hörsaal						
hungrig, in Mensa	s., i.M.			s., i.M.	h., i.K.	
hungrig, in Kaffee						s., i.K.
satt, in Mensa						
satt, in Kaffee						



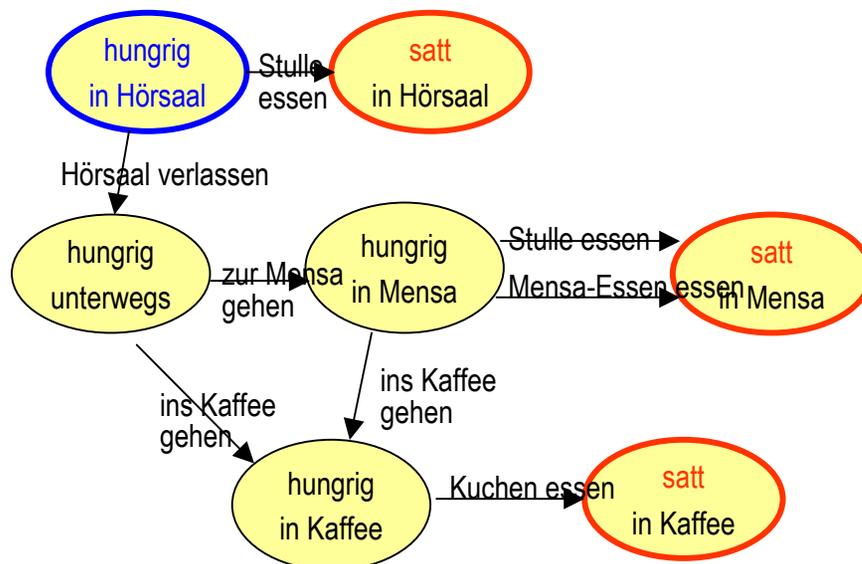
Übergangsmatrix

	Stulle essen	Hörsaal verlassen	zur Mensa gehen	Mensa-Essen essen	ins Kaffee gehen	Kuchen essen
hungrig, in Hörsaal	s., i.H.	h., u.				
hungrig, unterwegs			h., i.M.			h., i.K.
satt, in Hörsaal						
hungrig, in Mensa	s., i.M.			s., i.M.	h., i.K.	
hungrig, in Kaffee						s., i.K.
satt, in Mensa						
satt, in Kaffee						

Zeilen: Zustände z

Spalten: Aktionen a

Matrix-Element: von z durch a erreichter Zustand z'



- Graph
- Transitionssystem
- Automat
- Akzeptor

Komplexität (Anzahl der Zustände/Knoten)

8-er Puzzle: $9!$ Zustände

davon $9!/2 = 181.440$ erreichbar

15-er Puzzle: $16!$ Zustände

davon $16!/2$ erreichbar

ungarischer Würfel: $12 \cdot 4,3 \cdot 10^{19}$ Zustände

$1/12$ davon erreichbar: $4,3 \cdot 10^{19}$

Türme von Hanoi: 3^n Zustände für n Scheiben

lösbar in $(2^n) - 1$ Zügen

Dame: ca 10^{40} Spiele durchschnittlicher Länge

Schach: ca 10^{120} Spiele durchschnittlicher Länge

Go: 3^{361} Stellungen

Exponentielle vs. polynomiale Komplexität

n	n^2	n^3	2^n
10	100	1000	1024
100	10000	1000000	~ 100000000000
1000	1000000	1000000000	$\sim 10^{100}$

bei Komplexität 2^n :

Steigerung der Rechenleistung um **Faktor 1000**
ermöglicht Steigerung der Problemgröße von n auf $n+10$

Komplexitäts-Probleme

Speicher zu klein für Zustandsraum
Aufwand für Erkennen von Wiederholungen

Lösungsmethode:

„Expansion des Zustandsraumes“:

Schrittweise Konstruktion und Untersuchung von Zuständen

„konstruieren – testen – vergessen“

Expansionsstrategien

– Richtung

- Vorwärts, beginnend mit z_0
(forward chaining, data driven, bottom up)
- Rückwärts, beginnend mit Z_f
(backward chaining, goal driven, top down)
- Bidirektional

– Ausdehnung

- Tiefe zuerst
- Breite zuerst

– Zusatzinformation

- blinde Suche („uninformiert“)
- heuristische Suche („informiert“)

Güte von Suchalgorithmen

- bzgl. Komplexität des Verfahrens:
 - Zahl der Zustände insgesamt
 - Zahl der erreichbaren Zustände
 - Zahl der untersuchten Zustände
 - Suchtiefe
- bzgl. Gefundener Lösung(en)
 - **Korrektheit:**
 - alle Antworten sind korrekt
 - **Vollständigkeit:**
 - Algorithmus liefert (mind.) alle korrekten Antworten
(schwächer: bei Existenz wird eine Lösung gefunden)
 - **Optimalität:**
 - Algorithmus liefert optimale Lösung(en)

Zyklen, Maschen im Suchraum

Zustände werden mehrmals erreicht und expandiert.

Prolog:

`erreichbar(X,Y) :- erreichbar(X,Z), nachbar(Z,Y).`

`erreichbar(X,X).`

`symmetrisch(X,Y):- symmetrisch(Y,X).`

Trade-Off:

Test auf Wiederholungen: Zeit-, Speicher-aufwändig

(widerspricht „konstruieren – testen – vergessen“)

Beschränkung der Suchtiefe: unendliche Zyklen vermeiden

Suche nach einem Weg

Expansion: Schrittweise Konstruktion des Zustandsraums

Datenstrukturen :

- Liste **OPEN**:

Ein Zustand (Knoten) heißt "offen" , falls er bereits konstruiert, aber noch nicht expandiert wurde (Nachfolger nicht berechnet)

- Liste **CLOSED**:

Ein Zustand (Knoten) heißt "abgeschlossen" , falls er bereits vollständig expandiert wurde (Nachfolger alle bekannt)

Zusätzliche Informationen:

z.B. Nachfolger/Vorgänger der Knoten
(für Rekonstruktion gefundener Wege)

Schema S (Suche nach irgendeinem Weg)

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).

$OPEN := [z_0]$, $CLOSED := []$.

S1: (negative Abbruchbedingung) Falls $OPEN = []$: EXIT(„no“).

S2: (expandieren)

Sei z der erste Zustand aus $OPEN$.

$OPEN := OPEN - \{z\}$. $CLOSED := CLOSED \cup \{z\}$.

Bilde die Menge $Succ(z)$ der Nachfolger von z .

Falls $Succ(z) = \{\}$: Goto S1.

S3: (positive Abbruchbedingung)

Falls ein Zustand z_1 aus $Succ(z)$ ein Zielknoten ist: EXIT(„yes:“ z_1).

S4: (Organisation von $OPEN$)

Reduziere die Menge $Succ(z)$ zu einer Menge $NEW(z)$

durch Streichen von nicht weiter zu betrachtenden Zuständen.

Bilde neue Liste $OPEN$ durch Einfügen der Elemente aus $NEW(z)$.

Goto S1.

Variable Komponenten in Schema S :

(Re-)Organisation von OPEN in S4

– V1. Bildung der Menge NEW(z) aus Succ(z)

(Auswahl der weiter zu betrachtenden Zustände)

- alle Zustände aus Succ(z)
- einige (aussichtsreiche)
- nur die, die noch nicht in OPEN
- nur die, die nicht in CLOSED

– V2. Sortierung von OPEN

(bestimmt nächsten zu expandierenden Zustand in S2)

- NEW(z) sortieren
- NEW(z) einfügen, z.B. an Anfang oder Ende,
- OPEN (gesamte Liste) neu sortieren

– V3. Weitere Bedingungen

- Beschränkung der Suchtiefe
- Reduzierte Menge CLOSED

Blinde Suche: Tiefensuche/Breitensuche

Tiefe-Zuerst:

- V2: **NEW(z)** an den Anfang von **OPEN**

Keller

Breite-Zuerst:

- V2: **NEW(z)** an das Ende von **OPEN**

Warteschlange

Speicheraufwand für **OPEN**

für b = Verzweigungszahl(fan-out), d = Tiefe

- Tiefe-Zuerst: linear $d \cdot b$
- Breite-Zuerst: exponentiell b^d

Tiefen-/Breiten-Suche mit Test auf Wiederholungen:

$$V1: \text{NEW}(z) = \text{Succ}(z) - (\text{OPEN} \cup \text{CLOSED})$$

Für endliche Graphen:
korrekt und vollständig

Vollständig im Sinne:
findet (eine) Lösung im Fall der Existenz

Hoher Speicheraufwand für Liste **CLOSED**
(evtl. gesamter Graph)

Tiefen-/Breiten-Suche ohne Test auf Wiederholungen:

Statt Graph wird „Abgewickelter Baum“ untersucht

V1: $NEW(z) = Succ(z)$

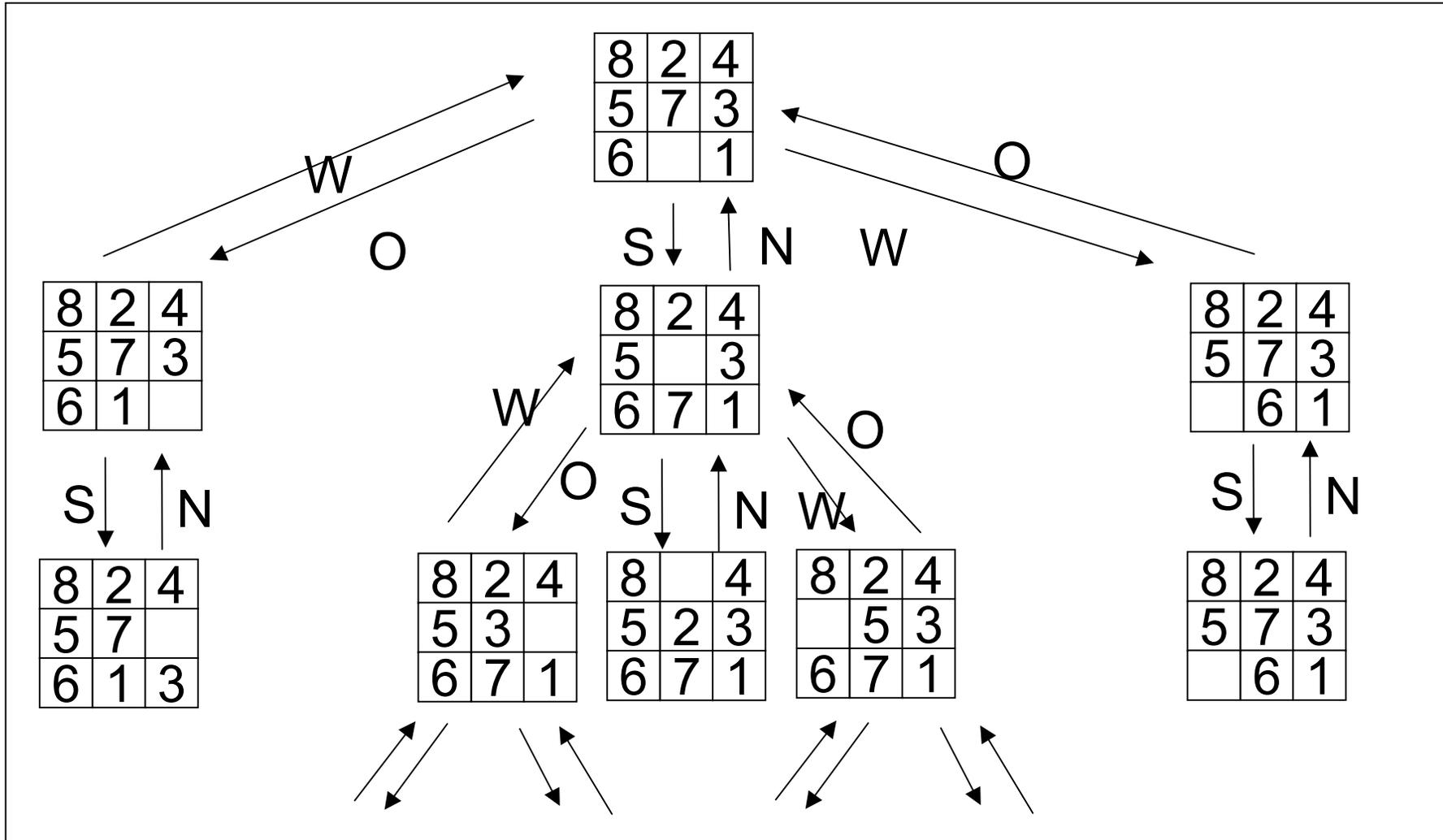
Für endliche Graphen:

Tiefe-Zuerst: korrekt, aber nicht immer vollständig

Breite-Zuerst: korrekt und vollständig

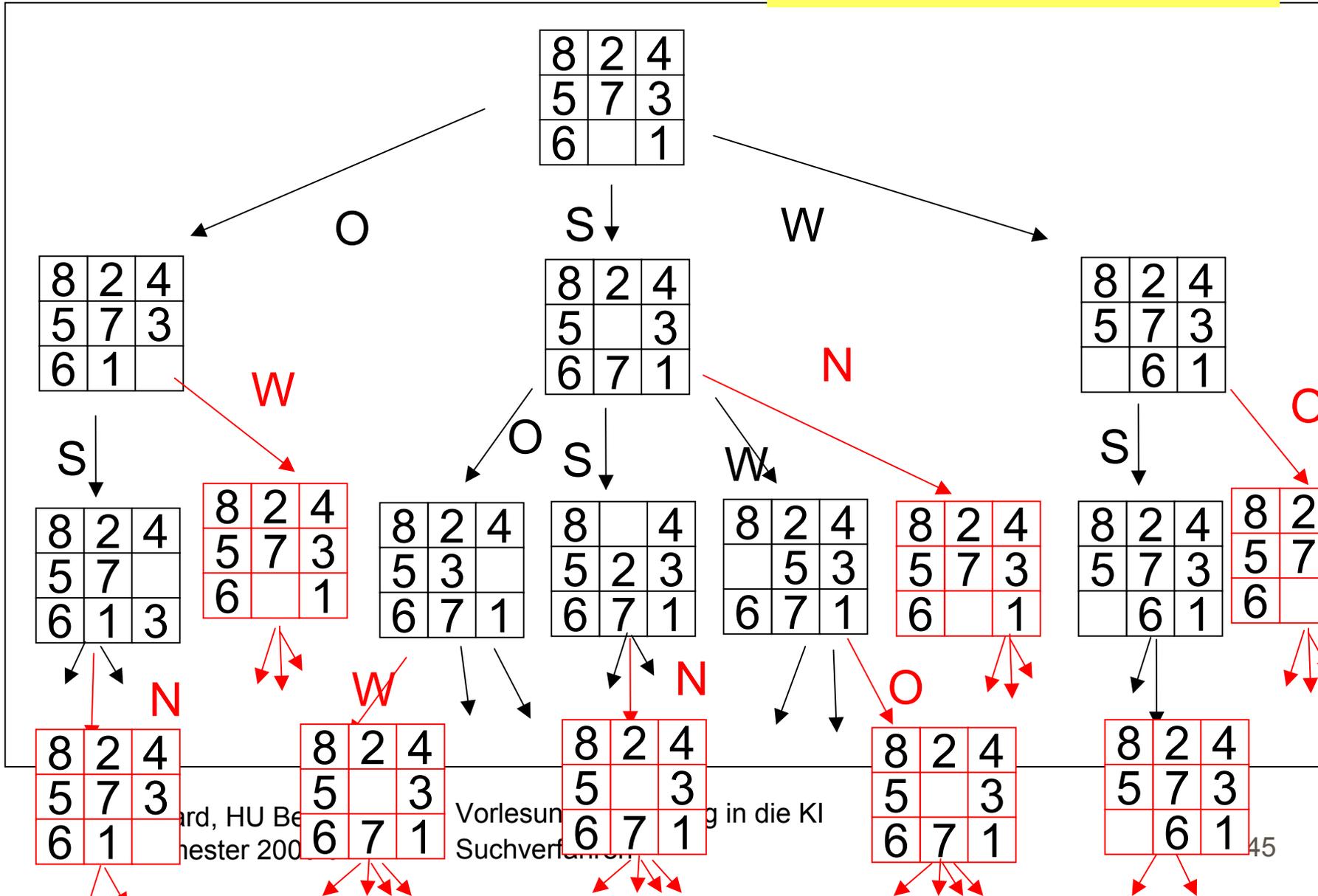
Hoher Zeitaufwand bei Zyklen/Maschen

Speicher-/Zeit-Trade-Off



Abwicklung

Durch Abwicklung
entstandener Baum



Abwicklung

Durch Abwicklung aus $G=[V,E]$ entstandener Baum

- hängt ab vom Startknoten v_0
- enthält für jeden in v_0 beginnenden Weg genau einen Knoten/eigenen Zweig
- ist endlich gdw. der mit v_0 zusammenhängende Teilgraph von G keine Zyklen enthält

$$T(G, v_0) = [K, B, v_0]$$

Knotenmenge: $K := \{ p \mid p \in V^* \text{ ist in } v_0 \text{ beginnender Weg von } G \}$

Kantenmenge: $B := \{ [p, pv] \mid p, pv \in K, v \in V \}$

(zusätzlich: entsprechende Beschriftungen der Knoten/Kanten)

Backtracking

Eine Implementierung von Tiefe-zuerst-Verfahren

Spezielle Organisation der Liste OPEN:

Referenz auf jeweils nächsten zu expandierenden Zustand in jeder Schicht

Nach Abarbeiten aller Zustände einer Schicht zurücksetzen (backtracking) auf davor liegende Schicht

Möglichkeit für Zyklenvermeidung mit reduzierter Menge CLOSED (nur für aktuellen Zweig):

- Beim Backtracking Rücksetzen von CLOSED auf früheren Stand

Tiefenbegrenzung

Tiefensuche in Schema S modifizieren:

- Maximal bis vorgegebene Tiefe d suchen
- Zusätzliches negatives Abbruchergebnis:
„es existiert keine Lösung bis zur Tiefe d “

Für endliche Graphen:

- korrekt, aber nicht immer vollständig
- Hinreichende Bedingung:

Falls Durchmesser kleiner d : vollständig

Iterative Tiefensuche

Stufenweise begrenzte Tiefensuche

- Stufe 1: begrenzte Tiefensuche bis zur Tiefe 1
- Stufe 2: begrenzte Tiefensuche bis zur Tiefe 2
- Stufe 3: begrenzte Tiefensuche bis zur Tiefe 3

• ...

„Depth-first-iterative deepening (DFID)“

DFID bis Tiefe d bei fan-out b erfordert insgesamt

$$b^d + 2 \cdot b^{d-1} + 3 \cdot b^{d-2} + \dots + i \cdot b \text{ Schritte}$$

Vergleich mit Tiefe-Zuerst/ Breite-Zuerst bis Tiefe d :

$$b^d + b^{d-1} + b^{d-2} + \dots + b \text{ Schritte}$$

DFID hat Speicherbedarf für OPEN wie Tiefe-zuerst

DFID findet Lösung wie Breite-zuerst

Bidirektionale Breitensuche

Ausgehend von Start und Ziel „parallel“ suchen bis zum Zusammentreffen.

- Aufwand: Von beiden Seiten nur halbe Tiefe
- Benötigt Kenntnis der „Vorgänger“
- Bei vielen Zielzuständen problematisch

Heuristische Suche

Schätzfunktion $\sigma(z)$: geschätzter Konstruktions-Aufwand für Erreichen eines Zielzustandes von z aus (dabei $\sigma(z) = 0$ für Zielzustände z)

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Bergsteigen/“hill climbing“ (ohne Test):

- V1: $NEW(z) = Succ(z)$ „Lokale Optimierung“
- V2: $NEW(z)$ nach Aufwand sortiert an Anfang von OPEN

Für endliche Graphen:
korrekt, aber nicht immer vollständig
(ähnlich Tiefensuche)

Heuristische Suche

Schätzfunktion $\sigma(z)$: geschätzter Konstruktions-Aufwand für Erreichen eines Zielzustandes von z aus

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Strahlensuche (ohne Test):

- V1: $NEW(z)$ = „Gute“ Auswahl aus $Succ(z)$
- V2: $NEW(z)$ nach Aufwand sortiert an Ende von $OPEN$

Für endliche Graphen:

korrekt, aber nicht immer vollständig

(analog eingeschränkter Breitensuche)

Heuristische Suche

Schätzfunktion $\sigma(z)$: geschätzter Konstruktions-Aufwand für Erreichen eines Zielzustandes von z aus

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Bestensuche/“Greedy Search“ (ohne Test):

- V1: $NEW(z) = Succ(z)$
- V2: $OPEN \cup NEW(z)$ nach Aufwand sortieren

Für endliche Graphen:
korrekt, aber nicht immer vollständig

Typische Probleme lokaler Optimierung

Vorgebirgsproblem:

steilster Anstieg führt auf
lokales Optimum ("Nebengipfel")

Plateau-Problem:

keine Unterschiede in der
Bewertung

Grat-Problem:

vorgegebene Richtungen
erlauben keinen Anstieg

Konsequenz:

zwischenzeitliche Verschlechterungen zulassen

Schätzfunktionen

Beste Schätzung: Weist korrekten Weg.

Gleichwertig: Kenntnis des gesuchten Weges.

Trade-Off

bzgl. Aufwand zur Berechnung der Schätzung

1.3 Suche nach "bestem Weg"

Bester/optimaler Weg:
Minimale Kosten

Graph: $G = [V, E]$ mit
–Anfangszustand $z_0 \in V$
–Zielzuständen $Z_f \subseteq V$

Kosten für Zustandsübergang (Kante)

$c: E \rightarrow \mathbb{R}^+$ (Kosten stets positiv!)

mit $c(e) = \text{Kosten der Kante } e \in E$

bzw. $c(z, z') = \text{Kosten der Kante } e = [z, z']$

Weg-Kosten als Summe von Kosten der Kanten.

Kosten eines Weges $s = e_1 \dots e_n \in E^*$:

$$c(e_1 \dots e_n) = \sum_{i=1, \dots, n} c(e_i)$$

Kosten eines Weges $s = z_0 z_1 \dots z_n \in Z^*$

$$c(z_0 z_1 \dots z_n) = \sum_{i=1, \dots, n} c(z_{i-1}, z_i)$$

Suche nach "bestem Weg"

Kosten für Erreichen des Zustandes z' von z aus:

– Falls z' von z erreichbar:

$$g(z, z') := \text{Min} \{ c(s) / s \text{ Weg von } z \text{ nach } z' \},$$

– Andernfalls: $g(z, z') := \infty$

Vorläufigkeit der Kostenberechnung während Expansion:

$G' = [V', E']$ sei (bekannter) Teilgraph von G

$$g'(z, z', G') := \text{Min} \{ c(s) / s \text{ Weg in } G' \text{ von } z \text{ nach } z' \}$$

$$g'(z, z', G') \geq g(z, z')$$

Suche nach "bestem Weg"

Verfahren "Generate and Test":
Alle Wege im Graphen untersuchen.

$L(z_0)$

= Menge der in z_0 beginnenden Wege $p = v_0 \dots v_n$

$L(z_0, Z_f)$

= Menge der in z_0 beginnenden Wege $p = v_0 \dots v_n$ mit $v_n \in Z_f$

Kürzesten Weg in $L(z_0, Z_f)$ bestimmen.

Suche nach "bestem Weg"

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT („yes:“ z_0).

OPEN := $[z_0]$, CLOSED := $[\]$.

Schema S (Suche nach irgendeinem Weg)

findet eventuell zuerst teure Wege

OPEN := OPEN - $\{z\}$. CLOSED := CLOSED $\cup \{z\}$.

Bilde die Menge Succ(z) der Nachfolger von z.

Lösungsidee:

Falls Succ(z) = $\{\}$: Goto S1.

Abbrechen, wenn alle offenen Wege teurer sind
als aktuell gefundene Lösung

(„yes:“ z_1)

dafür:

Durch Streichen von nicht weiter zu betrachtenden Zuständen.

Bilde neue Liste OPEN durch Einfügen der Elemente aus NEW(z).

- Positive Abbruchbedingung von Schema S verändern
- Umstellung der Schritte in Schema S

Schema S' für Suche nach "bestem Weg"

S'0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).

$OPEN := [z_0]$, $CLOSED := []$.

S'1: (negative Abbruchbedingung) Falls $OPEN = []$: EXIT(„no“).

S'2: (positive Abbruchbedingung)

Sei z der erste Zustand aus $OPEN$.

Falls z ein Zielknoten ist: EXIT(„yes:“ z).

S'3: (expandieren)

$OPEN := OPEN - \{z\}$. $CLOSED := CLOSED \cup \{z\}$.

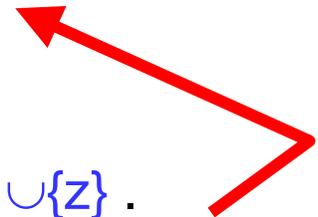
Bilde die Menge $Succ(z)$ der Nachfolger von z .

Falls $Succ(z) = \{\}$: Goto S'1.

S'4: (Organisation von $OPEN$)

- $g'(z_0, z', G')$ für alle $z' \in Succ(z)$ berechnen (im aktuellen G').
- Neue Liste $OPEN$ durch Einfügen der Elemente aus $Succ(z)$:
 $OPEN \cup Succ(z)$ sortieren nach aufsteigendem $g'(z_0, z', G')$

Goto S'1.



Schema S' für Suche nach "bestem Weg"

Satz :

Vor.: Es existiert $\delta > 0$ mit $c(z, z') > \delta$ für alle z, z'

Beh.: Falls Lösung existiert, findet Schema S', d.h.

- „Verzweigen und Begrenzen“ (Branch and bound),
 - „Uniform cost“ (bei Nilsson),
 - „Dijkstra's Algorithmus“ (1959)
- einen optimalen Weg

Verbesserungsmöglichkeiten:

Streichen aus OPEN (bzw. Succ(z)):

- Zustände aus CLOSED
- mehrmaliges Auftreten von Zuständen

Algorithmus A

Prinzip der Dynamischen Optimierung/Programmierung

A1- A3 : wie in Schema S'

A4: (Organisation von OPEN)

– $NEW(z) := Succ(z) - CLOSED$

– $g'(z_0, z', G')$ für alle $z' \in NEW(z)$ berechnen (im aktuellen G').

– Neue Liste OPEN durch Einfügen der Elemente aus $New(z)$:

$OPEN \cup NEW(z)$ sortieren nach aufsteigenden $g'(z_0, z', G')$

– Streichen von Wiederholungen in OPEN.

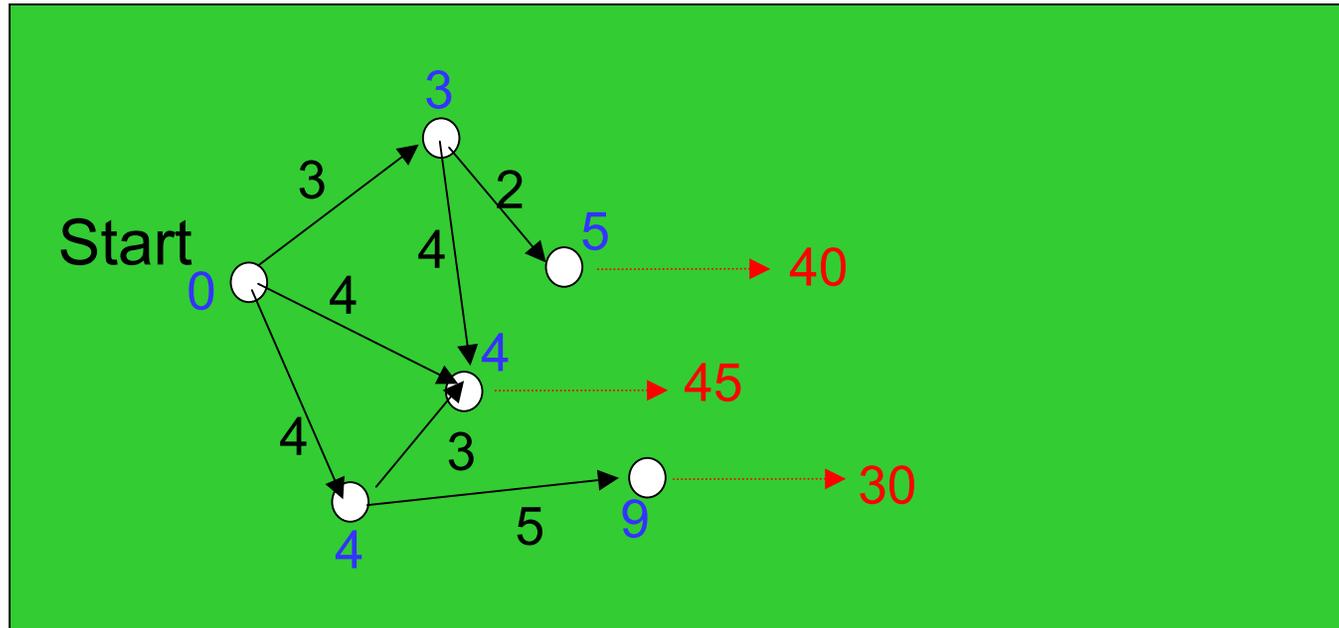
Goto S'1.

Satz :

Vor.: Ex. $\delta > 0$ mit $c(z, z') > \delta$ f.a. z, z' .

Beh.: Falls Lösung existiert, findet A einen optimalen Weg.

Heuristische Suche nach bestem Weg



$g'(z_0, z', G')$: bisher bekannte Kosten zum Erreichen von z'

$\sigma(z')$: geschätzte Kosten zum Erreichen des Ziels von z' aus

Schema S“(heur.Suche nach „bestem Weg“)

Heuristische Suche mittels Bewertungsfunktion $\sigma(z)$

Erfolgsaussichten offener Zustände z

(bzw. $\sigma(z, G')$ bei Abhängigkeit vom konstruierten G')

(Es gilt stets $\sigma(z) = 0$ für Zielzustände z .)

Schema S“:

S“1- S“3 : wie in Schema S‘

S“4: (Organisation von OPEN)

$g'(z_0, z', G') + \sigma(z')$ für alle $z' \in \text{Succ}(z)$ berechnen

Neue Liste OPEN durch Einfügen der Elemente aus Succ(z):

$\text{OPEN} \cup \text{Succ}(z)$ sortieren nach aufsteigendem $g'(z_0, z', G') + \sigma(z')$

Goto S“1.

Problem: Reihenfolge in OPEN abhängig von σ

Definition: $f(z) := \text{Min} \{ g(z, z_{\text{final}}) \mid z_{\text{final}} \in Z_{\text{final}} \}$

= tatsächliche minimale Kosten von z zu Zielzustand
($f(z_0)$ = Kosten des gesuchten optimalen Weges)

Schätzfunktion σ heisst optimistisch oder Unterschätzung,
falls $\sigma(z) \leq f(z)$ für alle $z \in Z$.

Satz :

Vor.: Ex. $\delta > 0$ mit $c(z, z') > \delta$ für alle z, z' .

σ ist optimistische Schätzfunktion

Beh.: Falls Lösung existiert, findet S“ einen optimalen Weg.

Bemerkungen:

Bedingung „Unterschätzung“ für σ nicht notwendig:
gleiche Reihenfolge wie $g' + \sigma$ liefern z.B. auch

$c_1 \cdot (g' + \sigma) + c_2$ für beliebige Konstanten c_1, c_2 .

$\sigma = 0$ ist ebenfalls Unterschätzung (vgl. Algorithmus A)

Optimale Reihenfolge bei $\sigma = f$

σ_2 *effektiver* als σ_1 falls $\sigma_1 \leq \sigma_2 \leq f$

(Hierarchien für Schätzfunktionen)

$g' = 0$: Suche nach irgendeinem Weg mit Heuristik σ

$g' = \text{Suchtiefe}$, $\sigma = 0$: Breitensuche

Bemerkungen

Verbesserungen analog Übergang von S“ nach A?

Verbesserungsmöglichkeiten:

Streichen aus **OPEN** (bzw. **Succ(z)**):

- Zustände aus **CLOSED**
- mehrmaliges Auftreten von Zuständen

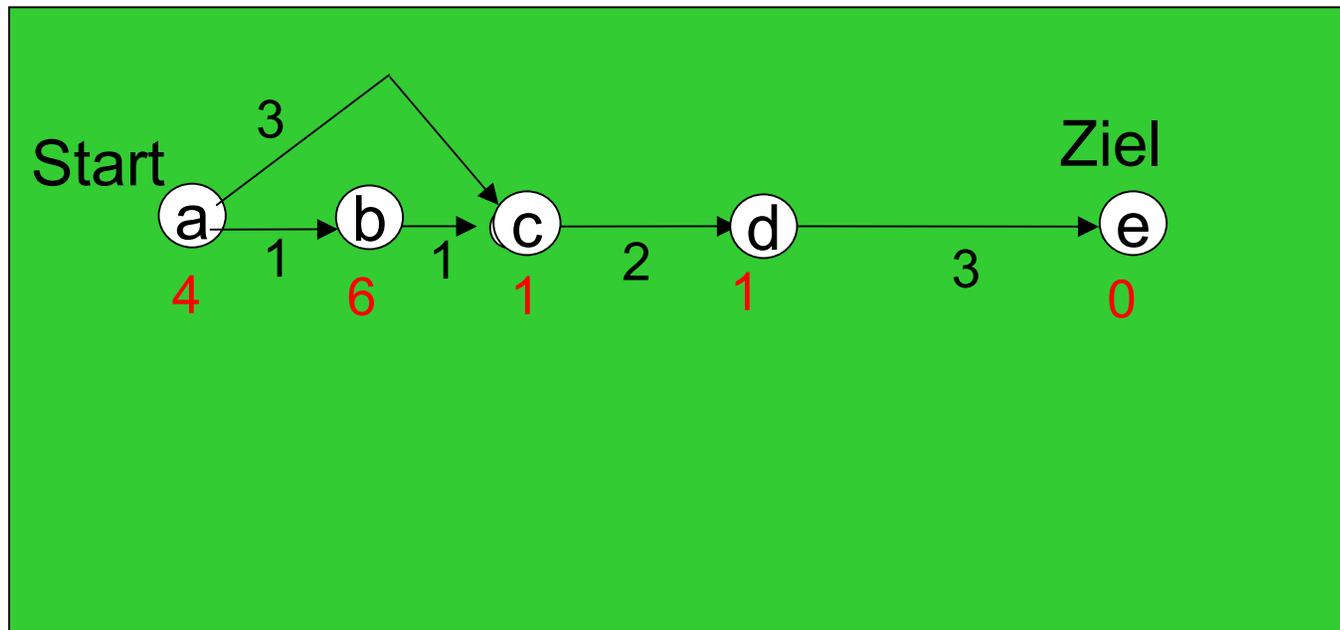


Solches Verfahren ist im allg. nicht korrekt!

Problem: **NEW(z):=Succ(z)-CLOSED**

Optimistisch (nicht konsistent – s.u.)

Streichen von Zuständen aus CLOSED führt evtl. zu falschem Ergebnis



$\sigma(z')$: geschätzte Kosten zum Erreichen des Ziels von z' aus

Algorithmus A* („weiche Form“)

A*0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).

$OPEN := \{z_0\}$, $CLOSED := \{\}$.

A*1: (negative Abbruchbedingung) Falls $OPEN = \{\}$: EXIT(„no“).

A*2: (positive Abbruchbedingung)

Sei z erster Zustand aus $OPEN$. Falls z Zielknoten: EXIT(„yes:“ z).

A*3: (expandieren)

$OPEN := OPEN - \{z\}$. $CLOSED := CLOSED \cup \{z\}$.

$Succ(z)$:= Menge der Nachfolger von z . Falls $Succ(z) = \{\}$: Goto A*1.

A*4: (Organisation von $OPEN$)

- $g'(z_0, z', G')$, $\sigma(z')$ für alle $z' \in Succ(z)$ berechnen im aktuellen G'
- $NEW(z)$ ergibt sich aus $Succ(z)$ durch Streichen aller $z' \in CLOSED$ mit $g'(z_0, z', G') \geq g'(z_0, z', G'_{alt})$
- Neue Liste $OPEN$ durch Sortieren von $OPEN \cup New(z)$ nach aufsteigendem $g'(z_0, z', G') + \sigma(z')$
- Bei mehrmaligem Auftreten eines z' in $OPEN$:
alle Vorkommen bis auf das mit minimalem $g'(z', G')$ streichen.

Goto A*1.

Algorithmus A* („weiche Form“)

A*0 - A*3 aus Algorithmus A übernommen.
Modifikation in A*4 (aber nicht bzgl. CLOSED).

Satz :

Vor.: Ex. $\delta > 0$ mit $c(z, z') > \delta$ für alle z, z' .

σ ist optimistische Schätzfunktion

Beh.: Falls Lösung existiert, findet A* (weiche Form)
einen optimalen Weg.

Algorithmus A* („harte Form“)

Ziel: A4 ohne Modifikation übernehmen
(speziell: **NEW(z):=Succ(z)-CLOSED**)

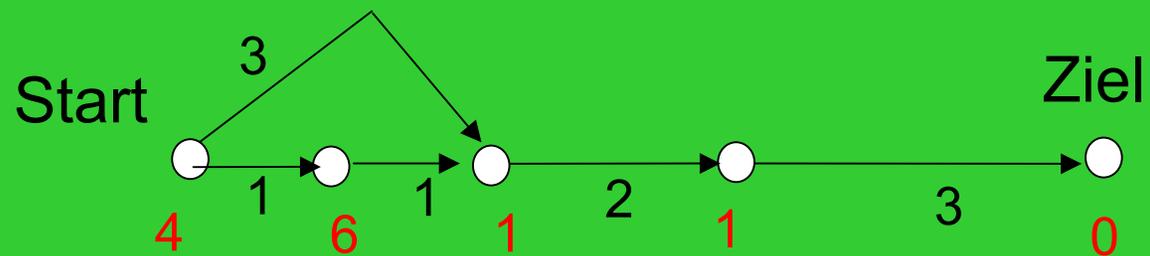
Definition:

Die Schätzfunktion σ heißt konsistent,
falls für beliebige Zustände z' , z'' gilt:

$$\sigma(z') \leq g(z', z'') + \sigma(z'')$$

Lemma: Wenn σ konsistent ist, so ist σ optimistisch.

(Umkehrung gilt i.a. nicht)



Algorithmus A* („harte Form“)

A*0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).

$OPEN := \{z_0\}$, $CLOSED := \{\}$.

A*1: (negative Abbruchbedingung) Falls $OPEN = \{\}$: EXIT(„no“).

A*2: (positive Abbruchbedingung)

Sei z erster Zustand aus $OPEN$. Falls z Zielknoten: EXIT(„yes:“ z).

A*3: (expandieren)

$OPEN := OPEN - \{z\}$. $CLOSED := CLOSED \cup \{z\}$.

$Succ(z) :=$ Menge der Nachfolger von z . Falls $Succ(z) = \{\}$: Goto A*1.

A*4: (Organisation von $OPEN$)

– $NEW(z) := Succ(z) - CLOSED$

– $g'(z_0, z', G')$, $f'(z')$ für alle $z' \in NEW(z)$ berechnen (im aktuellen G') .

– Neue Liste $OPEN$ durch Sortieren von $OPEN \cup NEW(z)$ nach aufsteigendem $g'(z_0, z', G') + \sigma(z)$

– Bei mehrmaligem Auftreten eines z' in $OPEN$:

alle Vorkommen bis auf das mit minimalem $g'(z_0, z', G')$ streichen.

Goto A*1.

Algorithmus A* („harte Form“)

A*0 - A*4 analog Algorithmus A

Satz :

Vor.: Ex. $\delta > 0$ mit $c(z, z') > \delta$ für alle z, z' .

σ ist konsistente Schätzfunktion

Beh.: Falls Lösung existiert, findet A* (harte Form)
einen optimalen Weg.

Algorithmus A* („harte Form“)

Zum Beweis:

Lemma:

Vor.: σ ist konsistente Schätzfunktion.

Beh.: Für jedes $z' \in \text{CLOSED}$ ist bei A* (harte Form):
der optimale Weg konstruiert.

Lemma:

Vor.: σ ist konsistente Schätzfunktion.

Beh.: Solange A* noch nicht gestoppt hat, gilt für jeden optimalen
Weg $z_0 \rightarrow z_1 \rightarrow z_2 \rightarrow \dots \rightarrow z_n = z$ zu einem Zielzustand z :
$$\exists i \in \{1, \dots, n\}: z_i \in \text{OPEN} \wedge g'(z_i, G') + \sigma(z_i) \leq f(z_0)$$

Algorithmus A*

Spezialfälle:

- $\sigma = 0$: Algorithmus A
- $g' = 0$: Suche nach irgendeinem Weg mit Heuristik σ
- $g' = \text{Suchtiefe}$, $\sigma = 0$: Breitensuche
(weitere Verfahren für andere σ)

Optimale Reihenfolge bei $\sigma = f$

σ_2 *effektiver* als σ_1 falls $\sigma_1 \leq \sigma_2 \leq f$

Hierarchien für (konsistente) Schätzfunktionen

Wichtungen für Einfluss von σ und g'

z.B. $c \cdot \sigma + (1 - c) \cdot g'$ mit $0 \leq c \leq 1$.

Algorithmus A*

Suchkosten ~ Zahl expandierter Zustände,
~ Berechnungskosten von σ und g'

Suchkosten vs. Lösungskosten

(optimale/suboptimale Lösungen)

Maß: „Penetranz“ := Weglänge/expandierte Knoten

Kombination von A* mit Tiefe-Zuerst (backtracking):

Iterative Deepening A* (IDA*)

mit Tiefenschranke gemäß $g'(z_0, z', G') + \sigma(z)$

Dynamische Optimierung/Programmierung

- Zustandsraum Z nicht notwendig diskret
- Diskreter Prozeß als mehrstufiger Übergang:
Anfangszustand z_1 zu Zielzustand z_n
- Kosten $g(z_1, z_2)$ für (gesamte) Überführung $z_1 \rightarrow z_2$
minimieren
- gesucht optimale „Strategie“ π
 - Strategie: Auswahl eines Operators
(Zustandsübergang) abhängig vom Zustand
- Markov-Bedingung: Kosten lokal berechenbar
- Bellmannsches Prinzip:
Wenn $z_1 \rightarrow_{op1} z_2 \rightarrow_{op2} z_3 \dots \rightarrow_{opn} z_n$ optimal ist,
so ist $z_2 \rightarrow_{op2} z_3 \dots \rightarrow_{opn} z_n$ optimal

Suche in Parameter-Räumen

Parameter: n Variable x_1, \dots, x_n mit Wertebereichen W_1, \dots, W_n

Optimalitätsfunktion $c(x_1, \dots, x_n)$

Gesucht:

$\mathbf{w} = [w_1, \dots, w_n] \in W_1 \times \dots \times W_n$ mit $c(\mathbf{w})$ minimal
(bzw. maximal)

Verfahren:

- Gradienten-Verfahren/Steilster Abstieg (bzw. Anstieg)
- Evolutionäre/Genetische Algorithmen

Schrittweises Modifizieren von Parametersätzen

„Bergsteigen“- Probleme

- Vorgebirgsproblem
- Plateau-Problem
- Grat-Problem

Simulated Annealing:
Zunächst große Schritte, später kleine

Gradienten-Verfahren/Steilster Abstieg

- Als Suchverfahren:
 - Iteration mit geeigneten Schrittlängen $l > 0$
 - jeweils in Richtung steilster Abstieg (Gradient)
- Schrittlänge l abhängig von Entfernung zum Minimum

Evolutionäre/Genetische Algorithmen

Idee aus der Natur:

Vermehrung „aussichtsreicher“ Lösungskandidaten

Kombination:

- Kreuzung
- Mutation
- Bewertung:
 - Fitness
- Auslese:
 - gemäß Wahrscheinlichkeit \sim Fitness

Genetische Algorithmen:

Parameter-Raum = $\{0,1\}^n$

Evolutionäre Algorithmen:

Parameter-Raum = \mathbb{R}^n

Charakteristika als Suchverfahren:

unterschiedliche Bereiche des Suchraums
erfassen

TU Berlin (Ingo Rechenberg)

<http://lautaro.bionik.tu-berlin.de/institut/s2foshow/>

Population, Individuum

Population: Menge von Individuen

Individuum: Durch Parametersatz beschrieben.

Genetische Algorithmen:
Parameter-Raum = $\{0,1\}$

Evolutionäre Algorithmen:
Parameter-Raum = \mathbb{R}^n

**Beschreibungsproblem:
Wahl geeigneter Parameter**

Evolutionäre/Genetische Algorithmen

Mutation:

Veränderung von Werten im Individuum $\mathbf{w} \in \text{Population}(t)$

Kombination („cross-over“): neues Individuum („Kind“) \mathbf{w}'
aus mehreren Individuen („Eltern“) $\mathbf{w} \in \text{Population}(t)$

Fitness: Nähe zu Optimalitätskriterium

Auswahl: Wahrscheinlichkeit gemäß Fitness

Viel Probieren (Problembeschreibung, Parameter, ...)

Evolutionäre/Genetische Algorithmen

Grundschema:

E1: (Start) $t:=0$, $\text{Population}(0) := \{ \mathbf{w}_1(0), \dots, \mathbf{w}_k(0) \}$
 $\text{Fitness}(\text{Population}(t)) := \{ \text{Fitness}(\mathbf{w}_1(t)), \dots, \text{Fitness}(\mathbf{w}_k(t)) \}$

E2: (Abbruch)

Falls $\text{Fitness}(\text{Population}(t))$ „gut“: $\text{EXIT}(\text{Population}(t))$

E3: (Kombination, Mutation)

$\text{Population}'(t) = \{ \mathbf{w}'_1(t), \dots, \mathbf{w}'_{k'}(t) \}$
 $:= \text{mutate}(\text{recombine}(\text{Population}(t)))$

E4: (Bewertung)

$\text{Fitness}(\text{Population}'(t)) := \{ \text{Fitness}(\mathbf{w}'_1(t)), \dots, \text{Fitness}(\mathbf{w}'_{k'}(t)) \}$

E5: (Auswahl)

$\text{Population}(t+1) = \{ \mathbf{w}_1(t+1), \dots, \mathbf{w}_k(t+1) \}$
 $:= \text{select}(\text{Population}'(t), \text{Fitness}(\text{Population}'(t)))$

$t := t+1$. Goto E3 .

Evolutionäre/Genetische Algorithmen

Karl Sims -- Virtual Creatures

Golem-Projekt Cornell University

**Humboldt-Universität:
Seminare von Manfred Hild
Projektkurs Kognitive Robotik**