Grundlagen der Künstlichen Intelligenz

3 Problemlösen durch Suche

Problemlösende Agenten, Problemformulierungen, Problemtypen, Suchstrategien

Volker Steinhage

Inhalt

- Problemlösende Agenten
- Problemformulierungen
- Problemtypen
- Beispielprobleme
- Suchstrategien

Zielorientierte Agenten

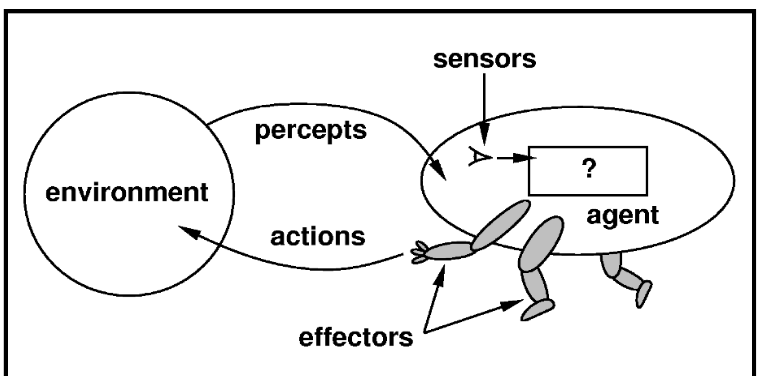
Vorlesung 1, Folie 43:

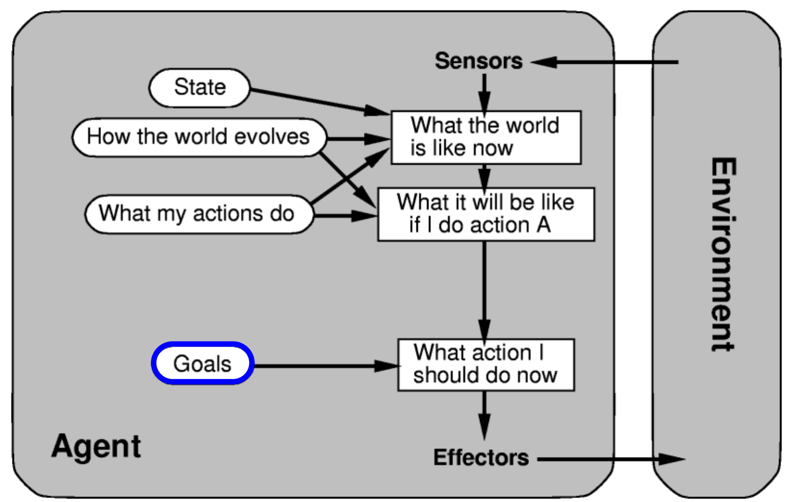
Ein rationaler Agent agiert so, dass er seine gegebenen Ziele erreicht unter der Voraussetzung, dass

(1) seine Eindrücke von der Umwelt richtig sind

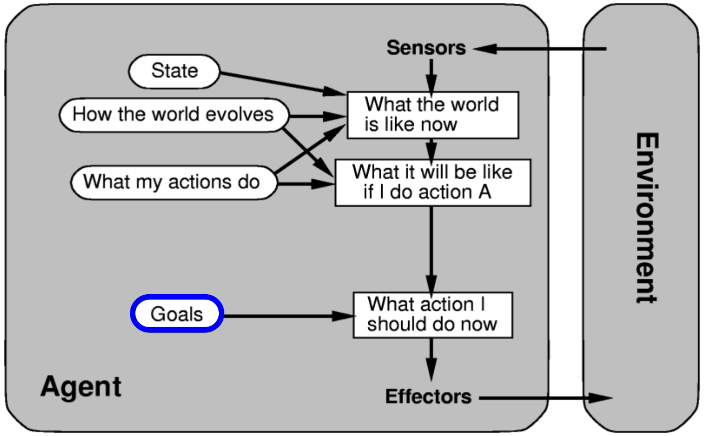
(2) seine Überzeugungen richtig sind

Also zunächst Betrachtung zielorientierter Agenten:





Problemlösende Agenten (1)



Aufgabenstellung:

- Gegeben: Ein aktueller Startzustand
- Gewünscht: Erreichen eines Zielzustandes

Lösungsansatz:

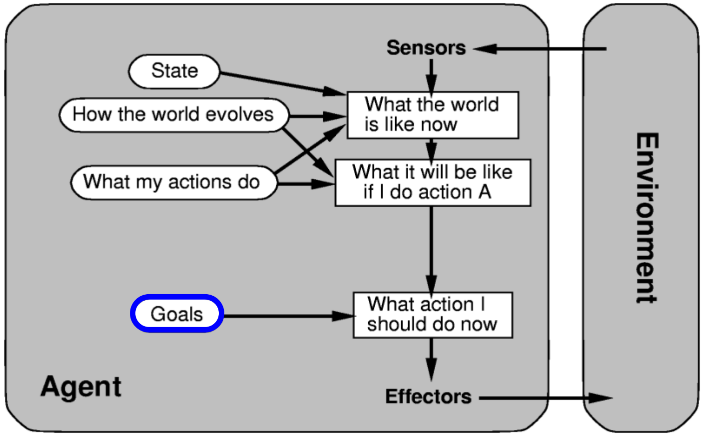
- Präzisierte Zielformulierung:
  - welche Leistungskriterien müssen in einem Zielzustand erfüllt sein?
  - … ist aber nur ein Teil des zu lösenden Problems: welche Aktionen, Perzepte und Zustände stehen überhaupt zur Verfügung?

→ Präzise Problemformulierung
umfasst Spezifikation aller mögl. Zustände (Weltzustands-raum), der Menge aller Startzustände, der Menge aller Zielzustände, sowie alle verfügbaren Aktionen und Perzepte

Eine angemessene Problemformulierung ist der erste Schritt zur Problemlösung

S. auch PEAS-Spezifikation aus Vorl. 2

Problemlösende Agenten (2)



Was ist eine Lösung?

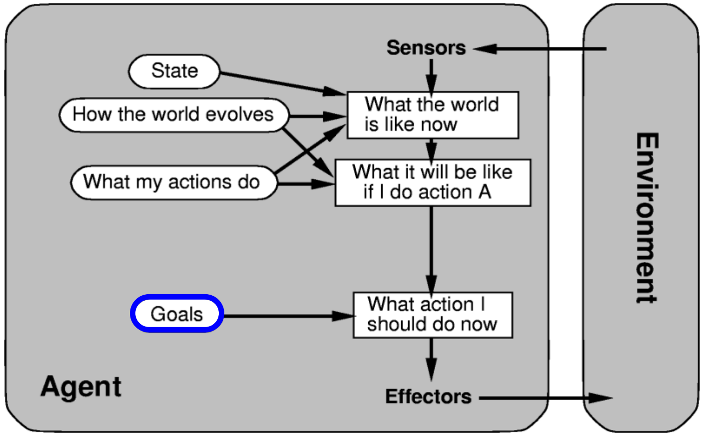
- Wir beginnen mit einem aktuellen Startzustand *s*
- Wir terminieren mit einem Zielzustand *z*

⤳ Lösung: eine geeignete Folge von Aktionen, die vom Startzustand *s* zum Zielzustand *z* führen

Wie wird eine Lösung ermittelt?

- Der Prozess, eine geeignete Folge von Aktionen zu ermitteln, die vom Startzustand *s* zum Zielzustand *z* führen, wird als Suche bezeichnet
- Ein Suchalgorithmus nimmt ein Problem als Eingabe und gibt eine Aktionsfolge als Lösung zurück

Problemlösende Agenten (2)



Lösung und Ausführung?

- Nachdem eine Lösung gefunden wird, können die Aktionen ausgeführt werden

  → Ausführungsphase

→ Agentenentwurf nach dem Schema:

Formulieren – Suchen – Ausführen

1) Formulierung von Problem und Ziel
2) Suche nach Lösung
3) Ausführung der Lösung

   Gehe zu 1) zur nächsten Aufgabe

Bildquelle: Colourbox, (https://www.colourbox.de/, 23.04.2021)

Ein einfacher problemlösender Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Lösung = Aktionsfolge

Herleitung der Aktionsfolge

Abarbeitung der Aktionsfolge

Der Agenten-Design nach *Formulate-Search-Execute*-Schema zeigt implizit folgende Umgebungsanforderungen (s. Vorl. 2): vollständige Beobachtbarkeit, determist. Aktionen, statische und diskret Zustände

Problemformulierung

1) Festlegen des *Weltzustandsraums*

   - *durch Abstraktion*: nur Betrachtung der *relevanten* Aspekte

   - *mit Bestimmung des Problemtyps*: abhängig vom verfügbaren *Wissen* über Weltzustände und Aktionen

2) Festlegen von *Startzuständen*: Weltzustände mit Starteigenschaften

3) Festlegen einer Nachfolgefunktion zur Überführung (Transformation) von Weltzuständen durch geeignete *Operatoren* / *Aktionen*

4) Festlegen eines *Zieltests* zur Überprüfung, ob die Beschreibung eines Zustands einem Zielzustand entspricht

5) Bestimmung von *Pfadkosten*: was kostet die Ausführung einer Aktion und folglich die Ausführung einer Aktionsfolge (s. Folgefolie)

Kosten

- Pfad: Folge von Aktionen, die von einem Zustand zu einem anderen führen

- Pfadkosten: Kostenfunktion g über Pfaden

  - entspricht i. A. der Summe der Kosten der einzelnen Aktionen

- Lösung: Pfad von einem Anfangs- zu einem Zielzustand

- Suchkosten: Zeit- und Speicherbedarf der Suche, um eine Lösung zu finden

- Gesamtkosten: Suchkosten + Pfadkosten, also Kosten

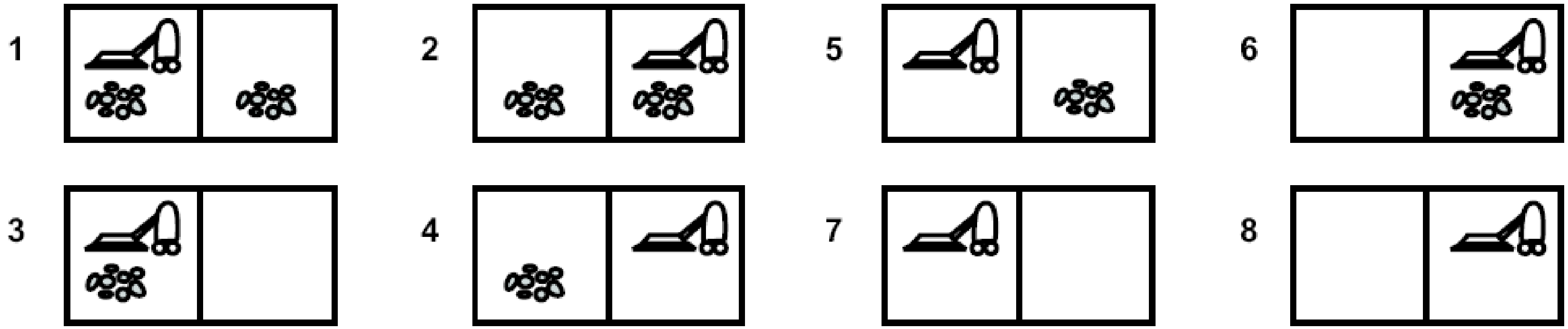
  - *für das Suchen* (Suchkosten, *Offline*-Kosten) +

  - *für die Ausführung* (Pfadkosten, *Online*-Kosten)

Formulate-Search-Execute-Schema

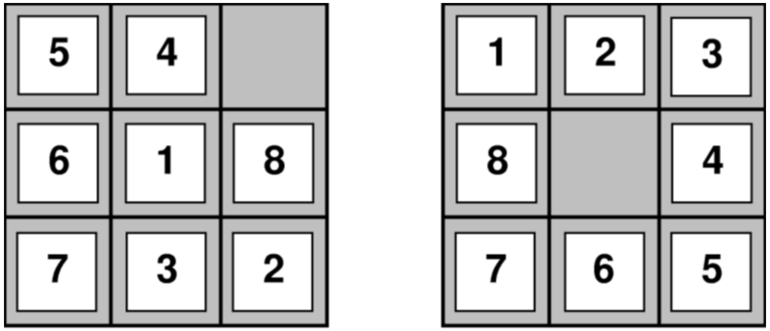
Problemformulierung für die abstrakte Staubsaugerwelt

- *Weltzustandsraum:* 2 Agentenpositionen, in beiden Räumen Schmutz oder kein Schmutz $\leadsto$ $2 \cdot 2^2 = 8$ Weltzustände



- *Startzustände:* jeder beliebige Zustand
- *Aktionen:* links (*L*), rechts(*R*), saugen(*S*)
  (Aktionen *L* im linken Raum, *R* im rechten Raum und *S* in gesaugtem Raum führen wieder zum Ausgangszustand)
- *Zielzustände:* kein Schmutz in beiden Räumen $\leadsto$ Zustände 7 und 8
- *Pfadkosten:* pro Aktion 1 Kosteneinheit
- Lösungsbeispiel: $Z_1 \rightarrow_S Z_5 \rightarrow_R Z_6 \rightarrow_S Z_8$ bzw. als Aktionsfolge: *S, R, S*

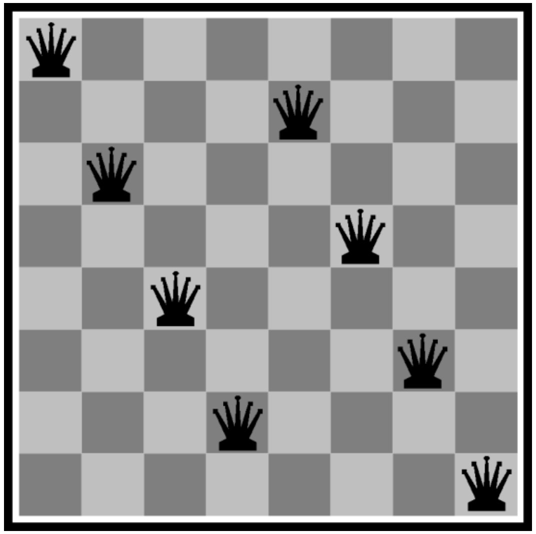
Problemformulierung für 8-Puzzle



Start State Goal State

- Zustände:
  - Beschreibung der Lage jedes der 8 Felder und aus Effizienzgründen des leeren Feldes
- Operatoren:
  - „Verschieben" des leeren Feldes nach links, rechts, oben und unten
- Zieltest:
  - Entspricht aktueller Zustand dem rechten Bild?
- Pfadkosten
  - Jeder Schritt kostet 1 Einheit

Problemformulierung für *8-Damen-Problem* (1)



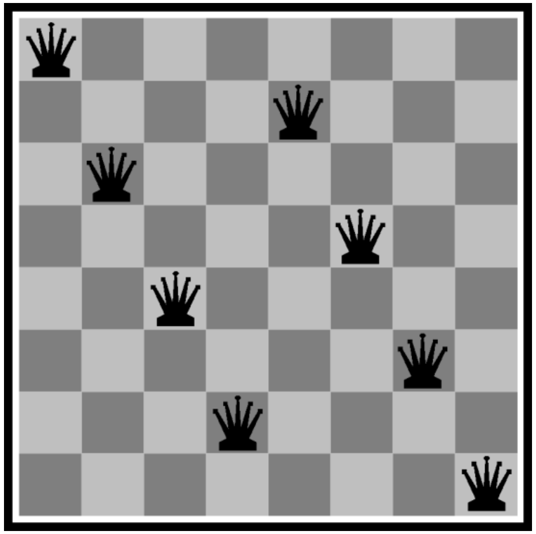
Bemerkung: Beispiel stellt keine Lösung dar.

- Zieltest:
  - 8 Damen auf dem Brett, keine angreifbar.
- Pfadkosten: 0 $\leadsto$ nur die Lösung interessiert!
- Darstellung 1:
  - Zustände: beliebige Anordnung von 0 bis 8 Damen
  - Operatoren: setze eine der Damen aufs Brett

$\rightarrow$ Problem: $64 \cdot 63 \cdot ... \cdot 57 \approx 3 \cdot 10^{14}$ Aktionsfolgen zu untersuchen!

Problemformulierung für *8-Damen-Problem* (2)

- Darstellung 2:
  - Zustände: Anordnung von 0 bis 8 Damen in unangreifbarer Stellung
  - Operatoren: Setze jede Dame *möglichst links* unangreifbar auf das Brett
  - → Vorteil: sehr viel weniger Aktionsfolgen für das 8-Damen-Problem: 2057
  - → Problem: immer noch 10^{52} Folgen für das 100-Damen-Problem (10^{400} in Darstg. 1)

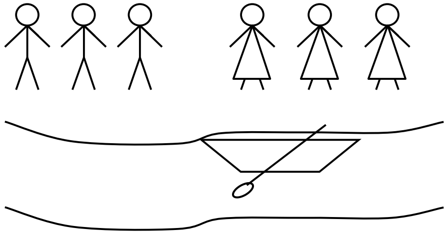


⇝ noch effizientere Darstellungen und effiziente Verfahren nötig!

Problemformulierung für *Missionare und Kannibalen*

Informelle Problembeschreibung:

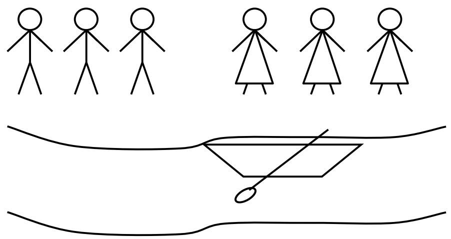
- An einem Fluss sind 3 Kannibalen und 3 Missionare, die alle den Fluss überqueren wollen



- Es steht ein Boot zur Verfügung, das maximal zwei Personen aufnimmt
- Es darf keine Situation eintreten, in der an einem Ufer Missionare *und* Kannibalen stehen *und* die Kannibalen dabei zahlenmäßig überlegen sind

→ Finde eine Aktionsfolge, die alle Missionare und Kannibalen wohlbehalten an das andere Ufer bringt

Formalisierung des *MuK-Problems*



- **Zustände:** Tripel (m,k,b) mit $0 \leq m, k \leq 3$ und $0 \leq b \leq 1$ für Variablen m, k und b für die Zahlen der Missionare, Kannibalen bzw. Boote am Ausgangsufer

→ **Anfangszustand:** (3,3,1)

- **Aktionen:** Von jedem Zustand aus können mit dem Boot entweder (1) ein Missionar, (2) ein Kannibale, (3) zwei Missionare, (4) zwei Kannibalen oder (5) einer von jeder Sorte übersetzen $\rightsquigarrow$ 5 Operatoren
  → nicht jeder Zustand ist so erreichbar [z.B. (0,0,1)] und einige sind nicht zulässig

- **Endzustand:** (0,0,0)

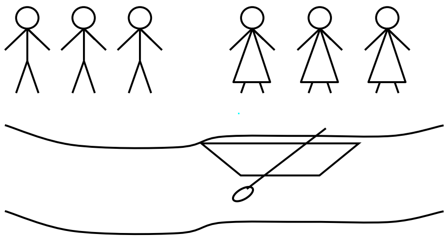
- **Pfadkosten:** 1 Kosteneinheit pro Flussüberquerung

Lösung des *MuK-Problems* durch Suche

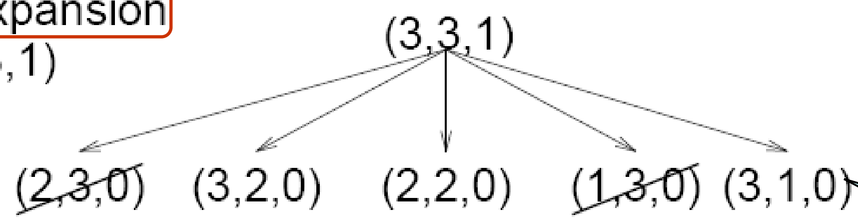
Ausgehend vom ***Anfangszustand*** schrittweise alle ***Folgezustände*** erzeugen

→ ***Suchbaum***:

(a) Anfangszustand (3,3,1)



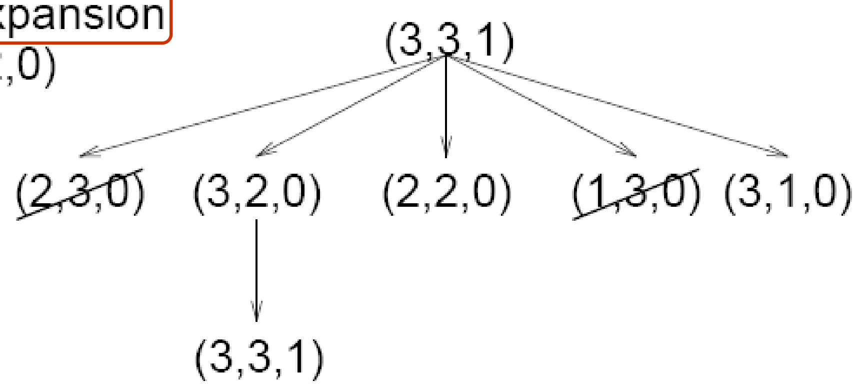
(b) nach Expansion von (3,3,1)



*Erste Ableitungsebene hat im Suchbaum die **Tiefe 1***

Anfangszustand und die Folgezustände bilden die Knoten des Suchbaums

(c) nach Expansion von (3,2,0)



Allgemeine Suchprozedur

```
function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Beachte: Zieltest erfogt auf den zur Expansion gewählten Knoten,

nicht auf den gerade expandierten Knoten *(resulting nodes)*!

[(1)] General-Search entspricht Tree-Search in 2. Auflage (s. Kommentar in übernächster Folie).

[(2)] Expansion = Erzeugen von Folgezuständen

Implementierung des Suchbaums

Knoten sind mehr als Zustände!

Datenstruktur für Knoten im Suchbaum umfasst:

- *State*: korrespondierender Zustand im Zustandsraum
- *Parent-Node*: Vorgängerknoten
- *Operator*: Operator/Aktion, der den aktuellen Knoten erzeugt hat
- *Depth*: Tiefe im Suchbaum = Anzahl der Knotenexpansionen entlang des Pfades vom Ausgangsknoten aus
- *Path-Cost*: Pfadkosten bis zu diesem Knoten

Funktionen zur Knotenexpansion durch eine *Warteschlange* (*Queue*):

- *Make-Queue(Elements)*: Erzeugt eine Queue
- *Empty?(Queue)*: Testet auf Leerheit
- *Remove-Front(Queue)*: Gibt erstes Element zurück
- *Queuing-Fn(Queue, Elements)*: Fügt neue Elemente ein (verschiedene Möglichkeiten)

Allgemeine Suche ... konkretisiert

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure   (1)

   nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
   loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
   end
```

- *Queuing Function* implementiert *strategy*
- *nodes* implementiert Warteschlange
- *state(node)* liefert Zustandsbeschreibung vom Knoten *node*
- *EXPAND(node,OPERATORS[problem])* erzeugt alle Nachfolgeknoten von *node* über die zulässigen Operatoren

[1] General-Search entspricht Tree-Search in 2. Auflage, ist aber stimmiger mit der Queue-Terminologie.

Bewertung von Suchstrategien

Kriterien:

- Vollständigkeit: Wird immer eine Lösung gefunden, sofern es eine gibt?
- *Optimalität:* Findet das Verfahren immer die *beste* Lösung?
  Beste Lösung: minimale Suchtiefe bzw.
  minimale Pfadkosten
- *Zeitkomplexität:* Wie lange dauert die Suche nach einer Lösung (im schlechtesten Fall)?
- *Platzkomplexität:* Wie viel Speicher benötigt die Suche (im schlechtesten Fall)?

Grundsätzliche Ansätze für Suchstrategien

Strategien:

- Uninformierte oder blinde Suche: keine problemspezifische Information!
  - Breitensuche, uniforme Kostensuche, Tiefensuche
  - tiefenbeschränkte Suche, iterative Tiefensuche
  - bidirektionale Suche

- Informierte oder heuristische Suche
  - $\rightsquigarrow$ nächste Vorlesung

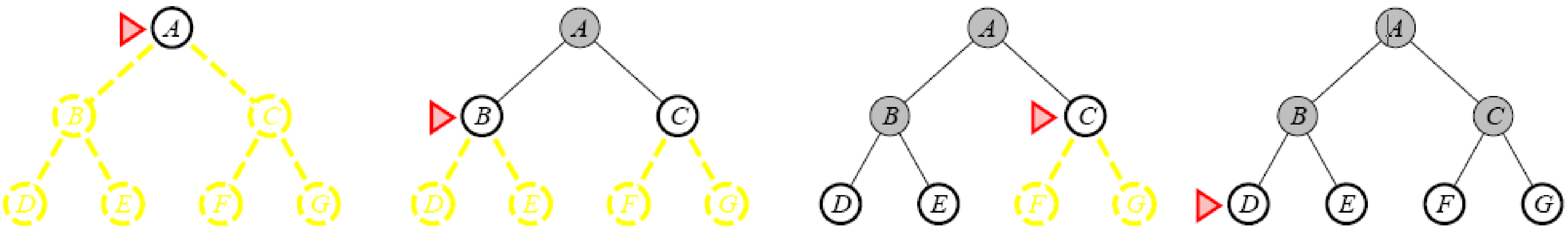
Breitensuche (1)

Expandiere Knoten in der Reihenfolge, in der sie erzeugt werden

→ *Queue-Fn* = *Enqueue-at-end*

→ *FIFO*-Warteschlange (*First-in-First-out)*

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```



- Findet immer die flachste Lösung

  → vollständig (bei endlicher *Lösungstiefe,* d.h. endl. Tiefe eines Zielknotens)

- Die Lösung ist optimal, wenn die Pfadkostenfunktion eine nichtfallende Funktion der Knotentiefe ist (z.B. wenn jede Aktion identische, nicht-negative Kosten hat).

Breitensuche (2)

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Allerdings sind die Suchkosten sehr hoch: Sei b der *maximale Verzweigungsfaktor* und $d > 0$ die Tiefe des kürzesten *Lösungspfads*. Dann werden maximal $O(b^{d+1})$ Knoten erzeugt:

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Schlechtester Fall: Alle Knoten der Tiefe d (bis auf den Zielknoten) werden expandiert

Beispiel: $b = 10$, 10.000 Knoten/Sec; 1.000 Bytes/Knoten:

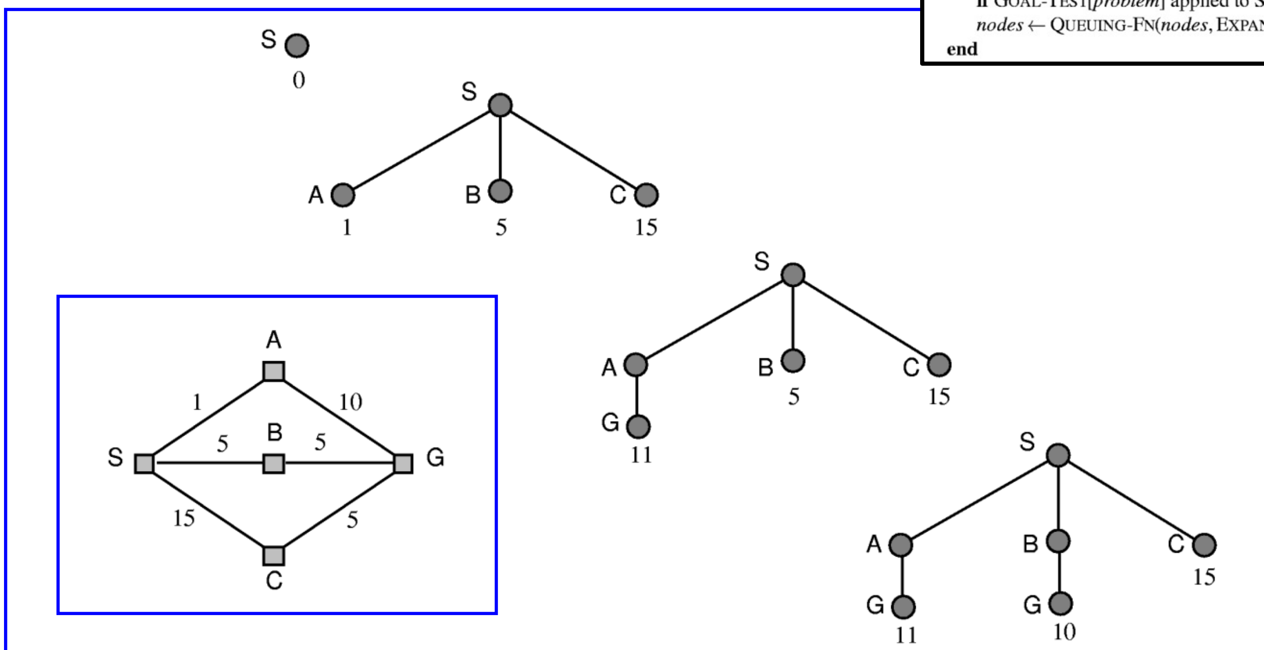
Tiefe	Knoten	Zeit	Speicher
2	1.100	0,11 Sekunden	1 Megabyte
4	111.100	11 Sekunden	100 Megabyte
6	10^7	19 Minuten	10 Gigabyte
8	10^9	31 Stunden	1 Terabyte (10^{12} Byte)
10	10^{11}	129 Tage	100 Terabyte (10^{14} Byte)
12	10^{13}	35 Jahre	10 Petabyte (10^{16} Byte)
14	10^{15}	3.523 Jahre	1 Exabyte (10^{18} Byte)

Uniforme Kostensuche

- Breitensuche ist optimal, wenn alle Schrittkosten gleich sind
- Uniforme Kostensuche als Generalisierung expandiert statt der flachsten Knoten die Knoten *n* mit den <u>geringsten Pfadkosten</u> *g(n)*

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Bspl.: Weg von *S* nach *G*:



(a) (b)

Findet immer die günstigste Lösung, falls $\forall\, n$: $g(succ(n)) \geq g(n)$. Zeit- und Speicher-komplexität: $O(b^{\lceil C^*/\varepsilon \rceil})$ mit Kosten C^* für optimalen Pfad und geringsten Aktionskosten ε.

Tiefensuche (1)

Expandiere immer einen nicht expandierten Knoten mit maximaler Tiefe

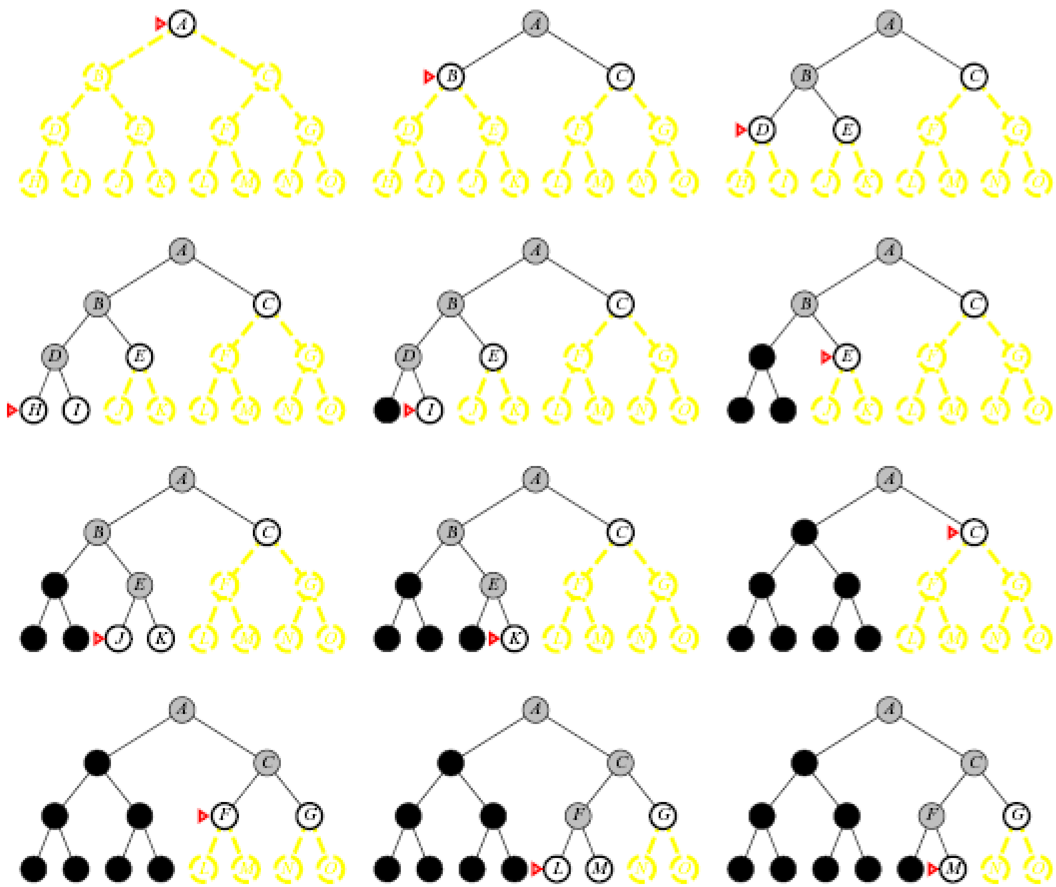
→ *Queue-Fn* = *Enqueue-at-front*

→ *LIFO*-Warteschlange für *Last-in-First-out*

Beispiel für Baum mit

max. Suchtiefe *m* = 3:

Abgearbeitete Knoten (schwarz) können aus dem Speicher entfernt werden!



Tiefensuche (2)

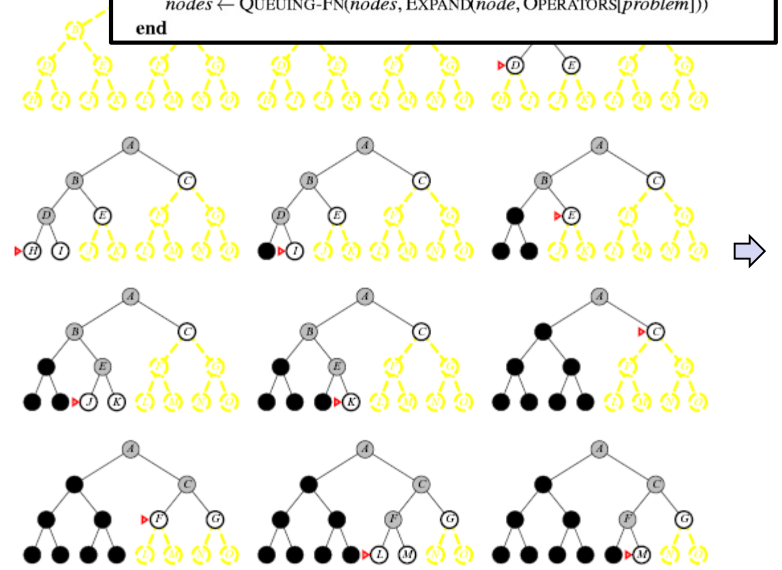
```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

- Benötigt nur ***Speicher*** für $b \cdot m + 1$ Knoten bei max. Verzweigung *b* und max. *Suchtiefe m*

  → Speicherkomplexität O(b·m)

- Die sog. *Backtracking*-Variante generiert immer nur *einen* Nachfolger. Der Vorgänger merkt sich dafür, welche Nachfolger alternativ als nächste zu erzeugen wären

  → *Speicherkomplexität O(m)*



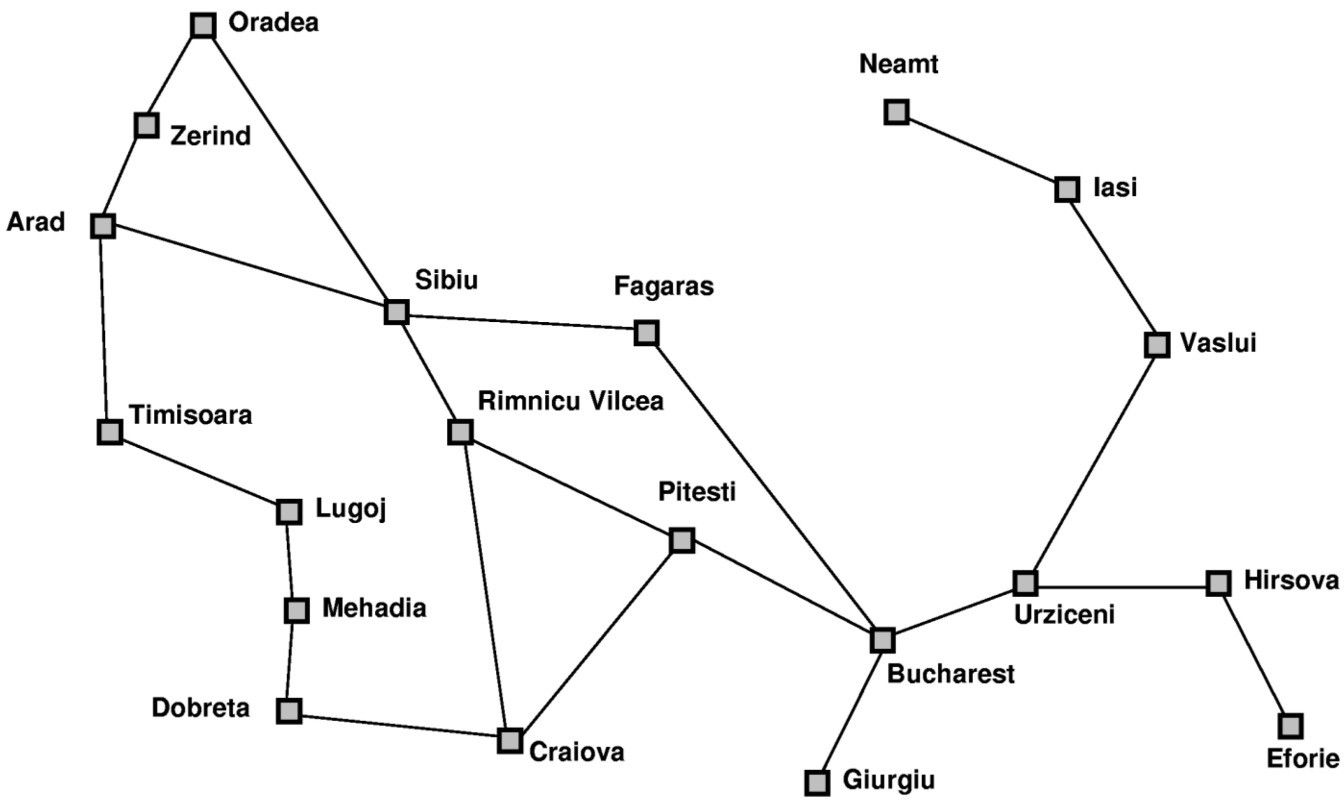
s. oben:
EXPAND(*nodes*,OPERATORS[*problem*] in *function* GENERAL-SEARCH

- Im schlechtesten Fall ist *Zeitkomplexität* $O(b^m)$
- Kann die Lösung bei unendlich tiefen Bäumen verfehlen → unvollständig
- Kann die optimale Lösung verfehlen → nicht optimal

Tiefenbeschränkte Suche

Es wird nur bis zu einer vorgegebenen Pfadlänge Tiefensuche durchgeführt

(z.B. bei *Routenplanung* mit max. n Teilstrecken für alle Städteverbindungen ist Suchtiefe $m > n$ sicher nicht sinnvoll)



Im Bspl. reicht max. Suchtiefe $m = 9$, da alle Städtepaare über max. 9 Teilstrecken verbunden sind $\leadsto$ *Durchmesser* des Problems

Iterative Tiefensuche (1)

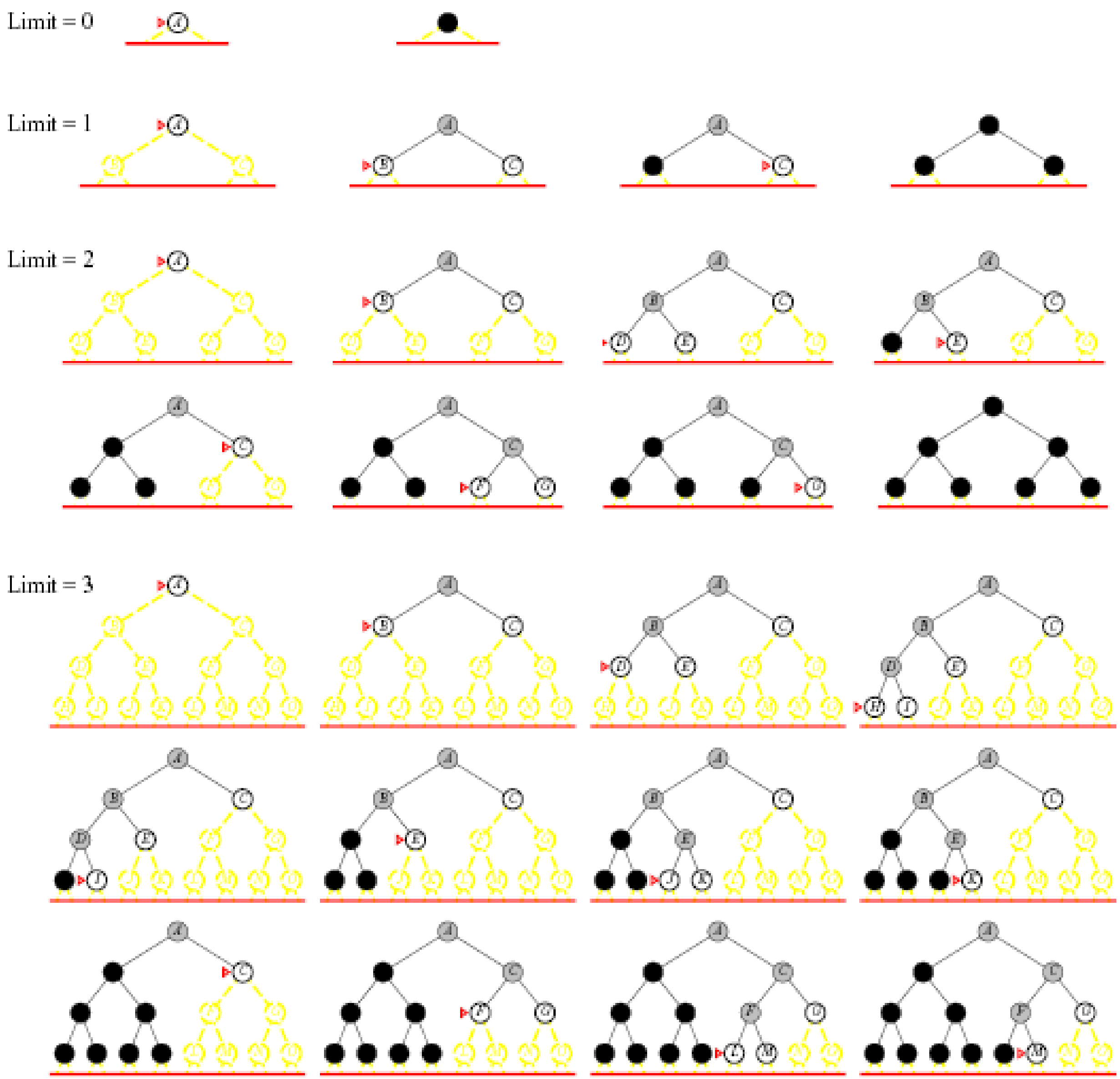
Iterative Tiefensuche

- kombiniert Tiefen- und Breitensuche,
- ist optimal und vollständig wie Breitensuche, braucht aber weniger Speicherplatz

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution sequence
    inputs: *problem*, a problem

    for *depth* $\leftarrow$ 0 **to** ∞ **do**
      if DEPTH-LIMITED-SEARCH(*problem*, *depth*) succeeds **then return** its result
    end
    return failure

Beispiel



Iterative Tiefensuche (2)

Zahl der erzeugten Knoten allgemein (bei max. Verzweigungsfaktor *b* und Lösungstiefe *d*):

Breitensuche	$b + b^2 + ... + b^{d-1} + b^d + (b^{d+1} - b)$
Iterative Tiefensuche	$d \cdot b + (d-1) \cdot b^2 + ... + 2 \cdot b^{d-1} + 1 \cdot b^d$

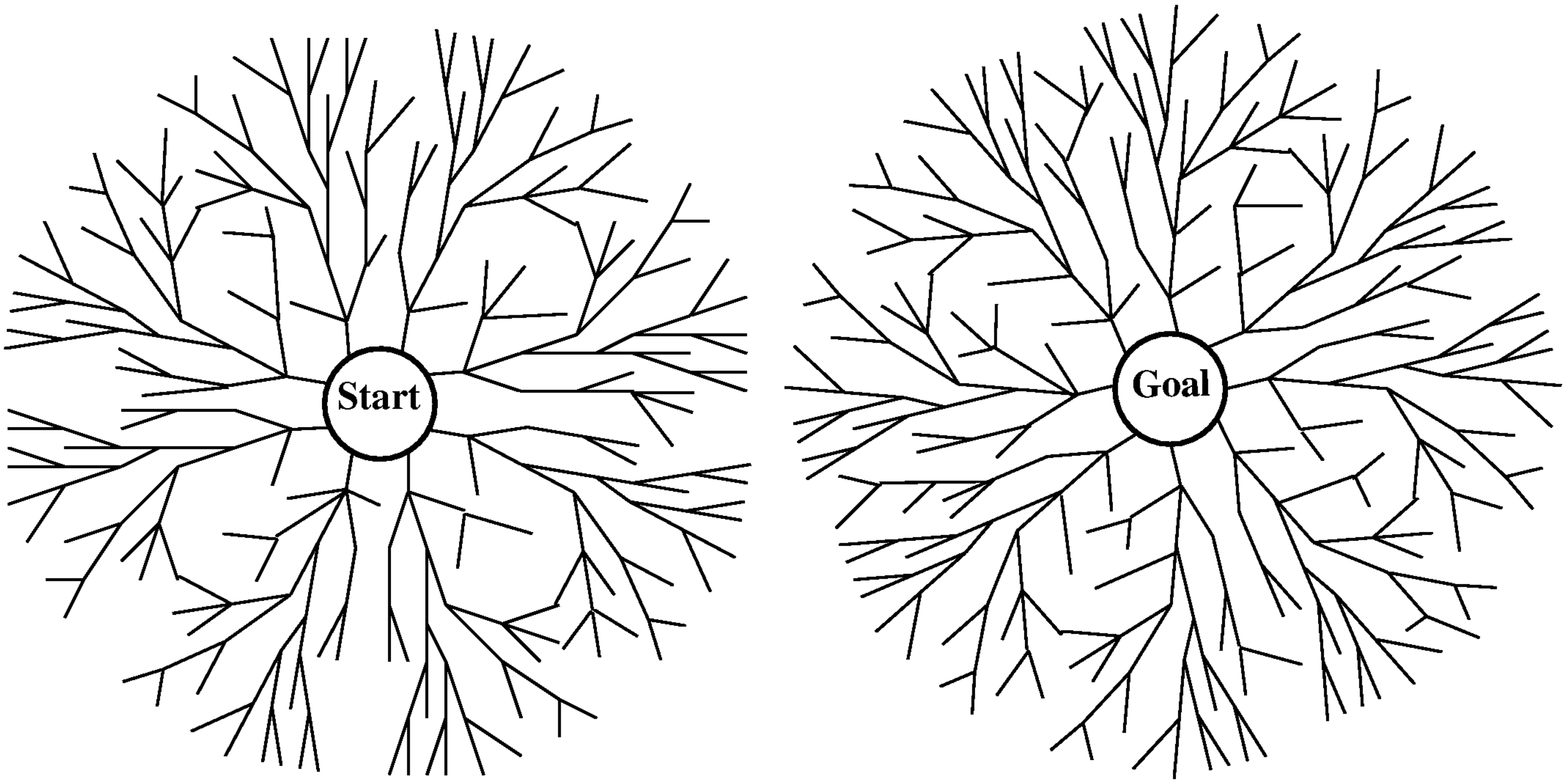
Beispiel für $b = 10$, $d = 5$:

Breitensuche	10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100
Iterative Tiefensuche	50 + 400 + 3.000 + 20.000 + 100.000 = 123.450

Zeitkomplexität: $O(b^d)$, *aber Platzkomplexität*: $O(b \cdot d)$

→ Iterative Tiefensuche ist bzgl. der Zeitkomplexität in derselben Größenordnung wie die Breitensuche und i.allg. die bevorzugte Suchmethode bei großen Suchräumen mit unbekannter maximaler Suchtiefe.

Bidirektionale Suche (1)



- Sofern Vorwärts- und Rückwärtssuche symmetrisch sind, erreicht man Suchzeiten gemäß der Argumentation $O(2 \times b^{d/2}) = O(b^{d/2})$
- Z.B. bei Breitensuche mit $b = 10$, $d = 6$ nur 22.200 Knoten statt 11.111.100

Bidirektionale Suche (2)

Zu beachten ist:

- Die Operatoren sind nicht immer oder ggf. schwer umkehrbar (entspr. Berechnung der Vorgängerknoten)
- In manchen Fällen gibt es sehr viele Zielzustände, die nur unvollständig beschrieben sind (z.B. Schach)
- Man braucht effiziente Verfahren, um zu testen, ob sich die Suchverfahren ”getroffen“ haben
- Welche Art der Suche wählt man für jede Richtung (im Bild: Breitensuche, die z.B. selbst wieder hohe Komplexität hat)?

Vergleich der Suchstrategien

Kriterien:

- Zeitkomplexität
- Platzkomplexität
- Optimalität
- Vollständigkeit

Variable:

b: max. Verzweigungsfaktor, d: Tiefe der Lösung,

m: maximale Tiefe des Suchbaums (Suchtiefe), l: Tiefenlimit

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Diskussion des Schemas *Formulieren – Suchen – Ausführen*

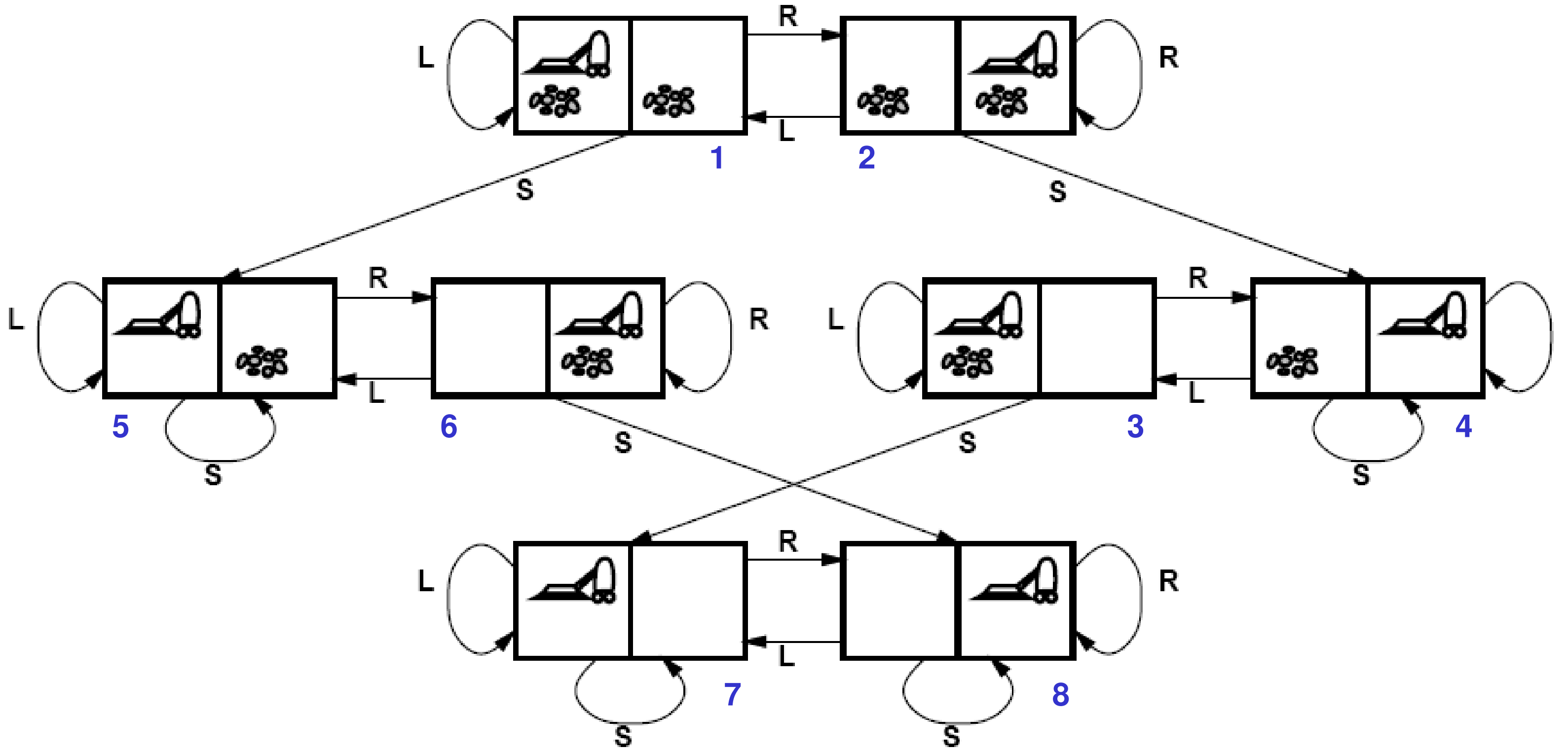
- Das Design-Schema *Formulieren – Suchen – Ausführen* setzt eine vollständig beobachtbare Umwelt und deterministische Aktionen voraus

→ Der Agent weiß immer eindeutig, in welchem Weltzustand er ist und in welchen Zustand er durch jede Aktion kommen wird.

- Diese Voraussetzungen sind nicht immer gegeben. Wir unterscheiden hier drei Problemklassen
  - Einzustandsprobleme
  - Mehrzustandsprobleme
  - Kontigenzprobleme

Problemtypen: Staubsaugerwelt als Einzustandsproblem

Bei vollständ. Beobachtbarkeit und determinist. Aktionen weiß der Agent immer, wo er ist und ob Schmutz vorliegt. Problemlösen reduziert sich auf die Suche nach einem Pfad von einem Anfangszustand zu einem Zielzustand:

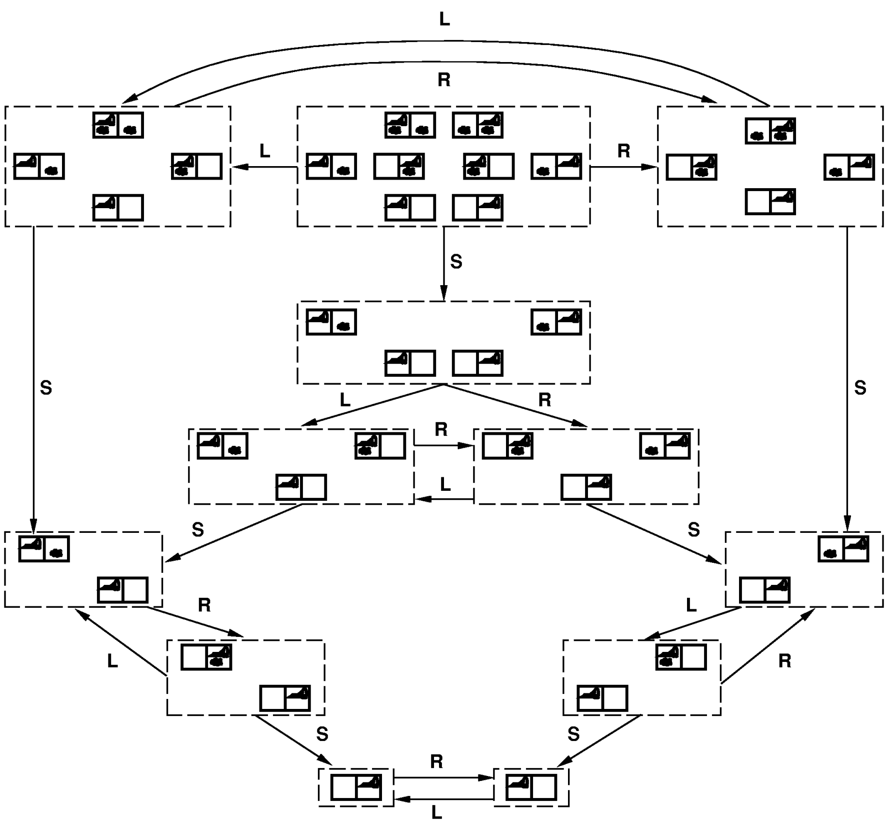


→ *Zustände* für die Suche: die Weltzustände 1 – 8

→ Lösung durch eines der besproch. Suchverfahren

Problemtypen: Staubsaugerwelt als Mehrzustandsproblem

Die Aktionen seien deterministisch, aber der Agent besitze keine Sensoren und weiß somit von Beginn an nicht, wo er ist und wo Schmutz ist. Trotzdem kann er das Problem lösen. Zustände sind dann sog. *Glaubenszustände* (*Belief States*):



Ein *Belief State* beschreibt die *Menge aller möglichen* physischen Zustände.

(Bei vollständig beobachtbarer Umwelt beschreibt dagegen jeder Glaubenszustand genau einen physischen Zustand.)

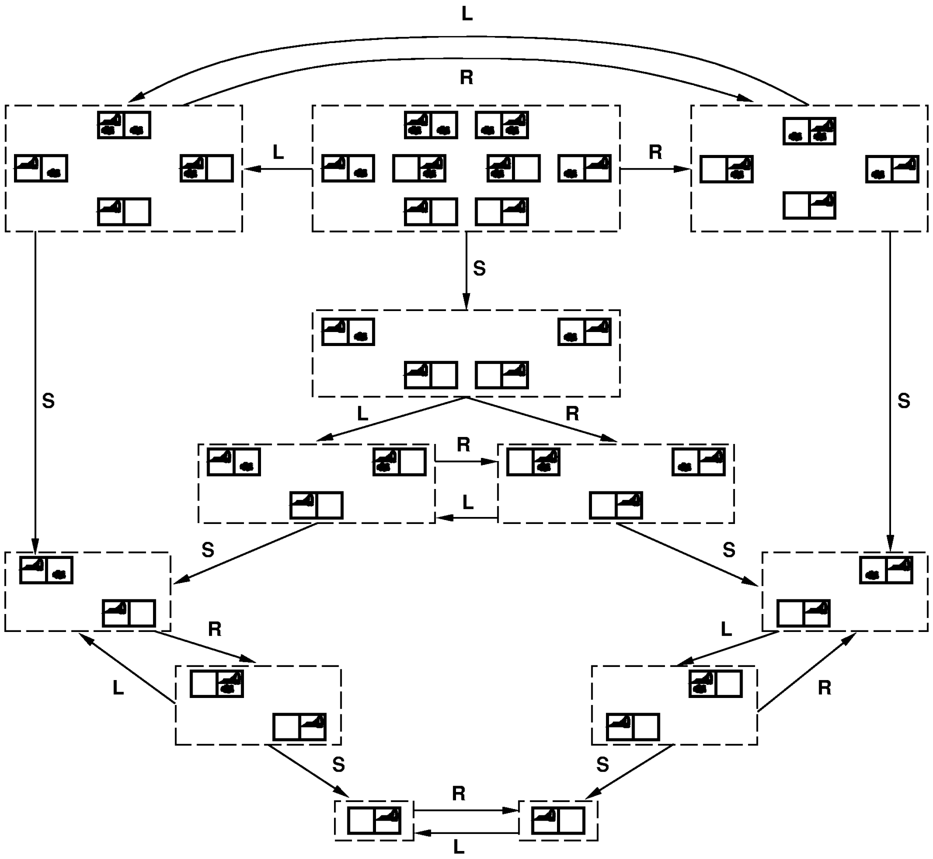
Eine Lösung ist jeder Pfad zu einem Belief State, dessen Elemente alle Zielzustände sind.

Zustände für die Suche: hier Zahl der erreichbaren Zustände = 12

Prinzipiell Worst Case: Potenzmenge der acht Weltzustände: $2^8 = 256$ Zustände

Problemtypen: Staubsaugerwelt als Mehrzustandsproblem

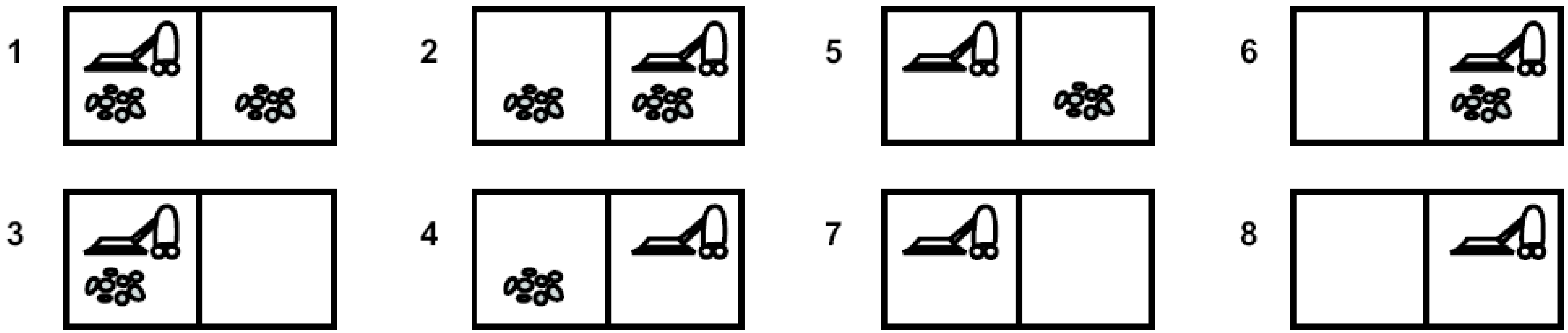
Eine Lösung des Mehrzustandsproblems erzwingt quasi die sukzessive Reduktion der Gesamtmenge aller möglichen Weltzustände über kleinere Potenzmengen letztlich auf solche Potenzmengen, die nur aus phys. Zielzuständen bestehen (hier Zustände 7 und 8).



Lösungen hier z.B. die Aktionssequenzen *R,S,L,S* und *L,S,R,S*.

Problemtypen: Staubsaugerwelt als Mehrzustandsproblem mit Unsicherheit

- Der Agent ist sensorlos *und* Aktionen sind nicht-deterministisch
- Bspl.: Die Aktion *Saugen* bewirke *im Defektfall* den Verlust von Schmutz auf sauberer Fläche – ansonsten wird vorhandener Schmutz beseitigt

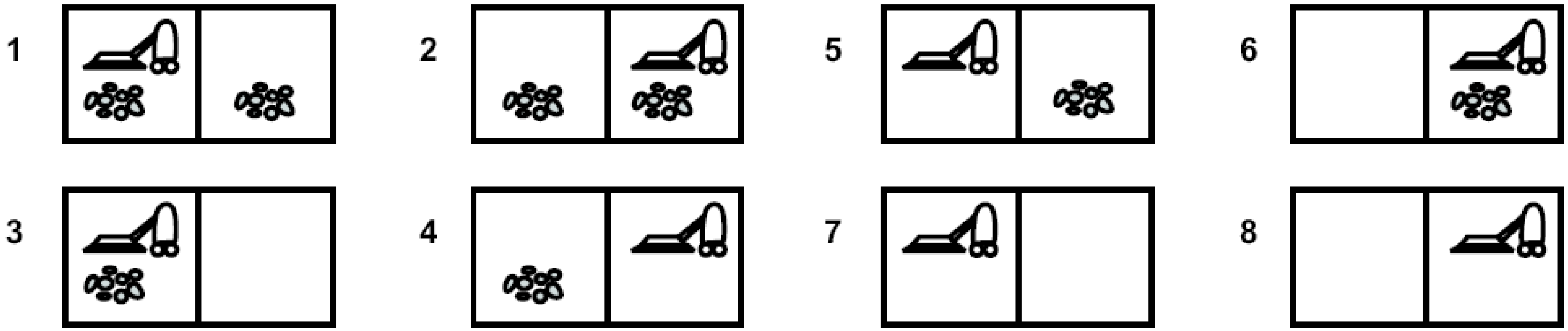


- Bspl.: vom Zustand {4} wird über Aktion *Saugen* der Zustand {2,4} erreicht
- Aber: Vom Startzustand {1,2,3,4,5,6,7,8} führt *Saugen* wieder zum selben Zustand {1,2,3,4,5,6,7,8} → die Menge der mögl. Zustände ist nicht reduzierbar
- Das Problem ist somit zwar formal gefasst und es existiert auch ein Pfad zu einer Lösung, aber die Umsetzung ist nicht einlösbar, da der Agent einfach nicht wissen kann, was seine Aktionen tatsächlich bewirken

Problemtypen: Staubsaugerwelt als Kontingenzproblem

- Die Aktionen seien nicht-determinstisch, aber der Agent kann seine Welt nach einer Aktion durch Sensoren neu erfassen

→ Er muss seine Aktionsfolge *vom tatsächlichen Effekt der Einzelaktionen* abhängig machen. Eine Lösung ist ein Baum von Aktionsfolgen anstatt eines einzelnen Pfades



Bspl.:

- Die unsichere Aktion *Saugen* bewirke im Defektfall weiterhin den Verlust von Schmutz auf sauberer Fläche – ansonsten wird vorhandener Schmutz beseitigt
- Vom Zustand {1,3} werde die Aktionsfolge S,R,S erzeugt. Aktion S führt zu {5,7}. Aktion R führt zu {6,8}. Liegt nach R der Zustand 6 vor, wird durch Aktion S der Zielzustand 8 erreicht. Die bedingte Aktionsfolge S, R, *if* (R,schmutzig) *then* S wäre dann die Lösung

Problemtypen: Staubsaugerwelt als Kontingenzproblem

- Die Lösungsmethoden für Kontingenzprobleme erfordern einen anderen Lösungsentwurf, in dem der Agent handeln kann, bevor er (durch Suche nach Lösungen) weiterplanen kann
- Es handelt sich also um Ansätze einer Verzahnung (Interleaving) von Suche und Ausführung anstelle einer vollständigen Trennung und Sequentialisierung gemäß dem Schema *Formulieren – Suchen – Ausführen*
- Wir werden dies bei Spielproblemen gegen Gegner kennenlernen (Vorl. 5)
- Als eine extreme Variante von Kontingenzproblemen finden wir Explorationsprobleme, bei denen die Zustände und die Aktionen der Umwelt unbekannt sind

Zusammenfassung

- Bevor ein Agent beginnen kann, eine Lösung zu suchen, muss er sein Ziel und darauf aufbauend sein Problem definieren.

- Eine Problembeschreibung umfasst fünf Komponenten: Zustandsraum, Anfangszustand, *Operatoren*, *Zieltest* und *Pfadkosten*. Ein *Pfad* vom Anfangszustand zu einem Zielzustand ist eine *Lösung* im Sinne einer *Aktionsfolge*.

- Es existiert ein *genereller Suchalgorithmus*, der benutzt werden kann, um Lösungen zu finden. Spezifische Varianten des Algorithmus benutzen verschiedene *Suchstrategien*.

- Suchalgorithmen werden auf Basis der Kriterien *Vollständigkeit*, *Optimalität*, *Zeit-* und *Platzkomplexität* beurteilt.

- Folgende Problemtypen wurden angesprochen: *Einzustandsprobleme, Mehrzustandsprobleme (mit Unsicherheit), Kontingenzprobleme.*