Kapitel 3. Problem-Lösen durch Suchen

Übersicht

3.1 Problem-Formulierung

3.2 Beispiele

3.3 Such-Strategien (allgemein)

3.4 Uninformierte Suche

Zeitplan:

- 45 Minuten

1 Einleitung

2 Intelligente Agenten

3 Problem-Lösen durch Suchen

Gesucht: **Agenten, die Probleme lösen.** Diese sind offensichtlich eine Teilklasse der Ziel-basierten Agenten.

Struktur: **Formuliere, Suche, Führe aus.**

Formuliere:

- **Ziel:** eine Menge von Zuständen,
- **Problem:** Zustände, Aktionen, die Zustande ineinander überführen, Transitionen

Suche: Welche Folge von Aktionen ist hilfreich?

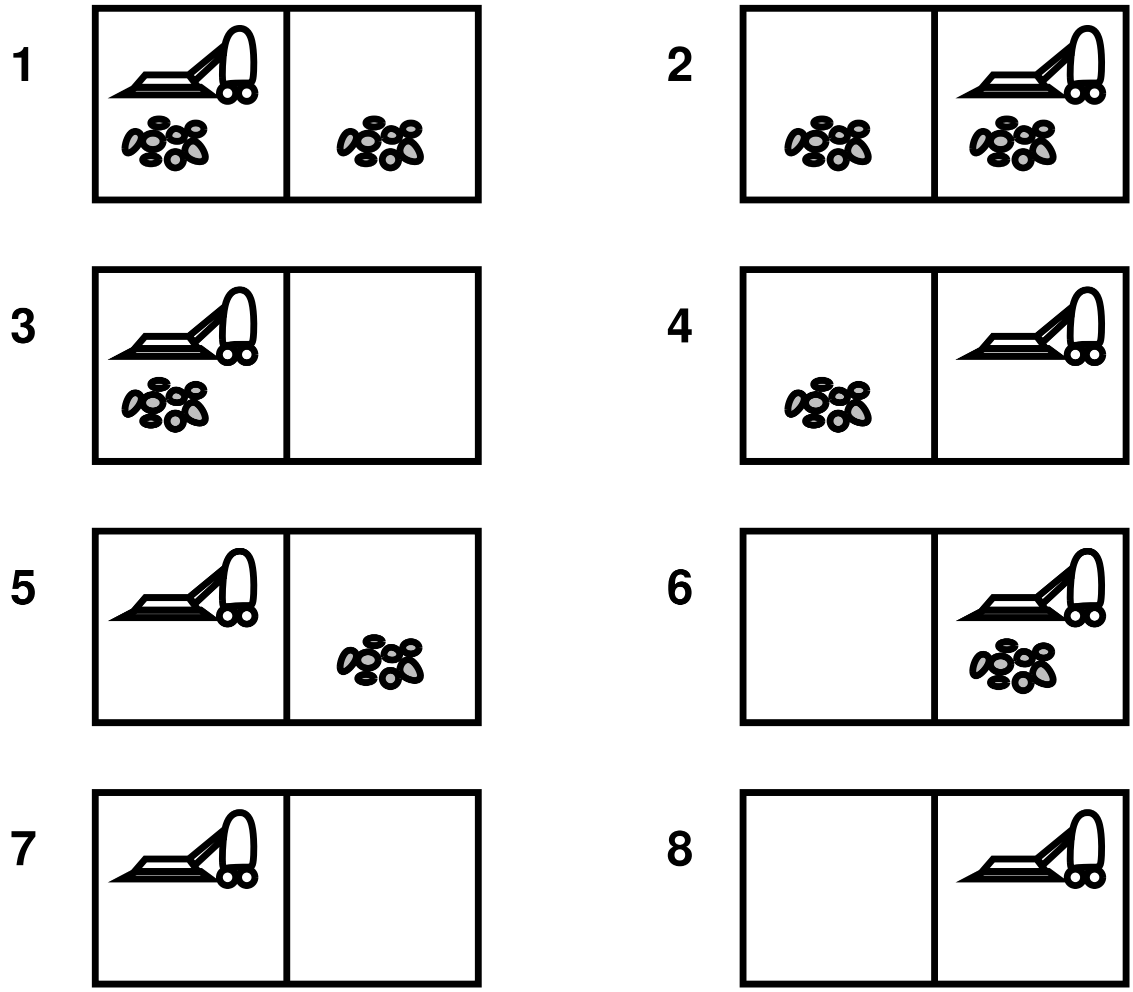
Führe aus: Man wende die Lösungsfolge von Aktionen auf den Anfangszustand an.

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

  state ← UPDATE-STATE(state, p)
  if s is empty then
     g ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, g)
     s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

3.1 Problem-Formulierung

Wir unterscheiden zwischen 4 verschiedenen Typen:

1. **1-Zustand-Probleme:** Aktionen sind vollständig beschrieben. Vollständige Information durch Sensoren zur eindeutigen Bestimmung des gegenwärtigen Zustands.
2. **Multiple-Zustand-Probleme:** Aktionen sind vollständig beschrieben, aber der Anfangszustand liegt nicht fest.
3. **Kontingenz-Probleme:** Manchmal gibt es keine fixe Sequenz als Lösung, sondern man muss den ganzen Baum betrachten. ($\leadsto$ Übung: Murphy's Gesetz, $\leadsto$ Kapitel 7. Planen)
4. **Explorations-Probleme:** Nicht einmal die Effekte der Aktionen sind bekannt. Man muss in der Welt suchen, nicht im abstrakten Modell.


1
2
3
4
5
6
7
8

Definition 3.1 (1-Zustand-Problem)
Ein 1-Zustand-Problem besteht aus

- *einer Menge von Zuständen, darunter einer der als* Anfangszustand *ausgezeichnet ist,*
- *einer Menge von n* Aktionen *(wir nennen sie auch* Operatoren*), die, in einem Zustand ausgeführt, zu einem anderen Zustand überleiten:*

$$\textit{Operator}_i\textit{: Zustände} \mapsto \textit{Zustände}, \ \ i = 1, \ldots, n$$

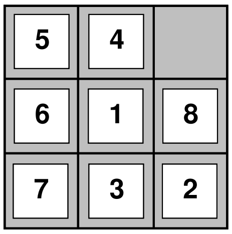
- *einer Menge von* Ziel-Zuständen*, oder einem Ziel-Test, der, angewandt auf einen Zustand, bestimmt ob es sich um einen Ziel-Zustand handelt oder nicht,*
- *einer Kosten Funktion g, die jedem Pfad im Zustandsraum (Menge der erreichbaren Zustände) gewisse Kosten zuordnet. Üblicherweise ist g additiv.*

Wie geht das für Multiple Zustand-Probleme?
($\rightsquigarrow$ Übung)

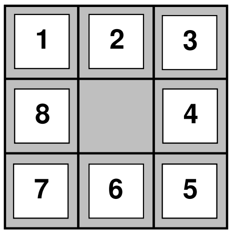
Wie wählt man Aktionen und Zustände aus?
($\rightsquigarrow$ Abstraktion)

3.2 Einige Beispiele

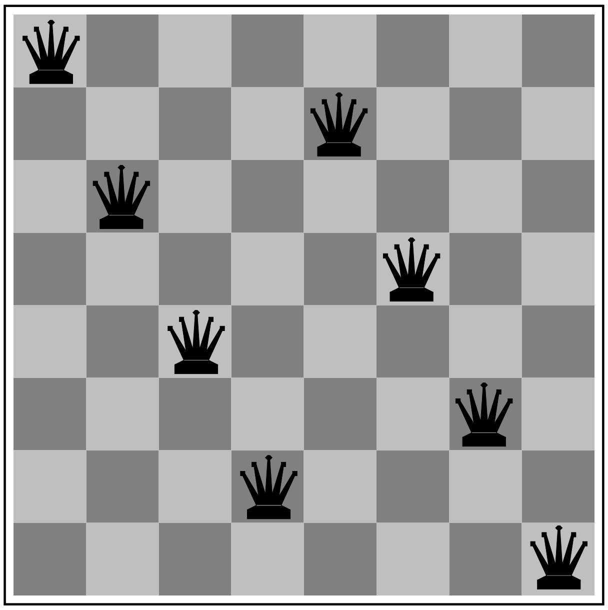
Spiel-Beispiele:



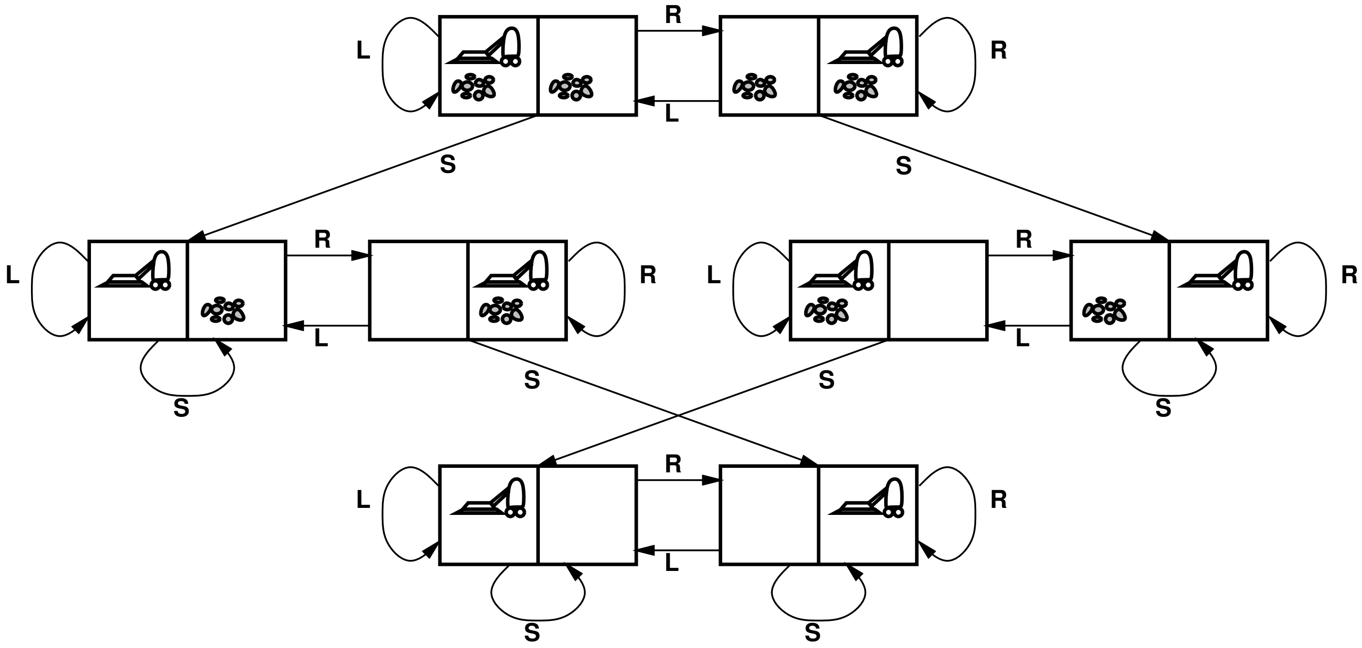
Start State



Goal State



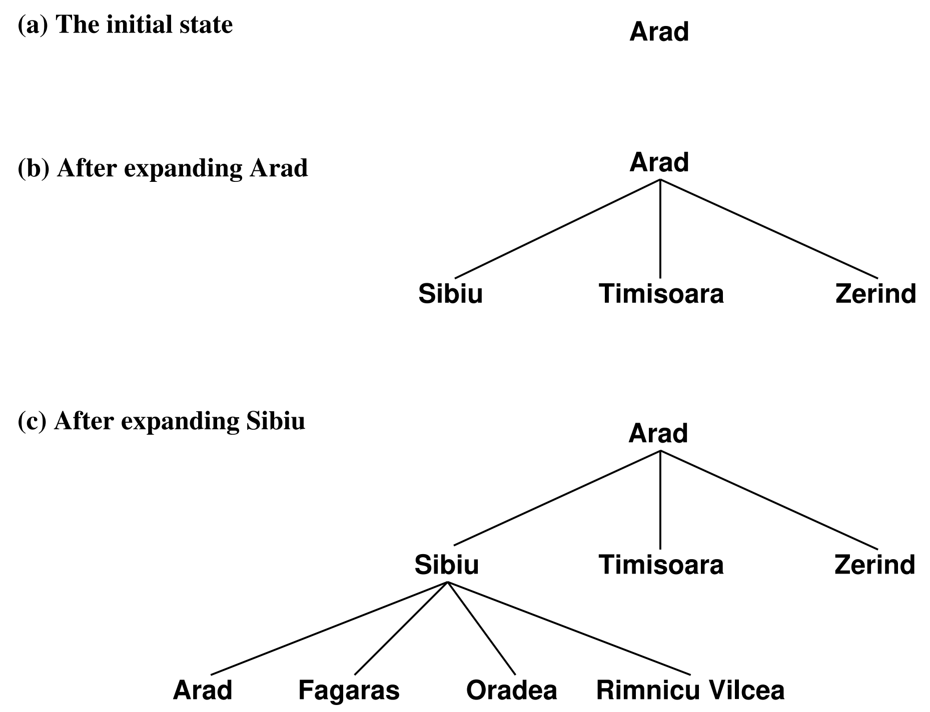
Staubsauger Welt mit Murphy's Gesetz



- Missionare vs. Kannibalen

3.3 Such-Strategien (allgemein)

Prinzip: Auswählen, Testen, Expandieren. ($\rightsquigarrow$ Such-Baum)




```
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
```

Wichtig: Zustands-Raum versus Such-Baum

Während der Zustandsraum üblicherweise endlich ist, ist der Such-Raum meist abzählbar unendlich.

Definition 3.2 (Datentyp Knoten)
Der Datentyp Knoten *ist durch die Komponenten* Zustand, Elter-Knoten, Operator, Tiefe *und* Pfad-Kosten *definiert.*

- Ein Knoten ist ein Buchhalter bezüglich einer Problem-Instanz und bezüglich eines bestimmten Algorithmus.
- Ein Zustand ist ein Schnappschuss der Welt.

Beachten Sie die *rekursive* Abhängigkeit (Knoten/Elter-Knoten). Falls wir die *Tiefe* weglassen, müssen wir einen ausgezeichneten Knoten $root$ einführen, die *Wurzel*. Dann ist die Tiefe **definierbar**. Umgekehrt ist $root$ durch die Tiefe **definierbar**: der Knoten, der sich selbst als Elter hat und die Tiefe 0.

Wir wollen versuchen, die Funktion GENERAL-SEARCH etwas konkreter zu machen.

Design-Entscheidung: Die zu expandierenden Knoten stellen wir nicht als Menge sondern als **Queue** dar.

Make-Queue(Elements)	**Remove-Front(Queue)**
Empty?(Queue)	**Queing-Fn(Queue,Elements)**

```
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
```

Was sind interessante Anforderungen an Such-Strategien?

- **Vollständigkeit**,
- **Zeit-Komplexität**,
- **Speicher-Komplexität**,
- **Optimalität (bez. Pfad-Kosten)**

Wir unterscheiden zwischen

Uninformierte vs. informierte Suche.

3.4 Uninformierte Suche

Breadth-First-Suche: "Knoten der kleinsten Tiefe werden zuerst expandiert",

$$\text{Queing-Fn}(Queue, Expand(\ldots)) \text{ : neue Knoten ans Ende}$$

Vollständig: Ja.
Optimal: Ja, falls alle Operatoren dieselben Kosten haben.

Für einen konstanten Verzweigungsfaktor b gilt: In der Tiefe d gibt es maximal

$$1 + b + b^2 + \ldots + b^{d-1} + b^d\text{-viele Knoten.}$$

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Uniform-Cost-Suche: "Knoten n mit kleinsten Pfad-Kosten $g(n)$ werden zuerst expandiert",

Queing-Fn($Queue, Expand(\ldots)$) : neue Knoten werden gemäss ihrer Pfad-Kosten mit denen in der Queue verglichen und entsprechend eingefügt

Vollständig: Ja, falls jeder Operator die Pfadkosten um mindestens einen Betrag $\delta > 0$ anwachsen lässt (siehe unten).

Falls alle Operatoren die gleichen Kosten haben (insbesondere wenn $g(n) =$ Tiefe(n)) gilt: Uniform-Cost=Breadth-First.

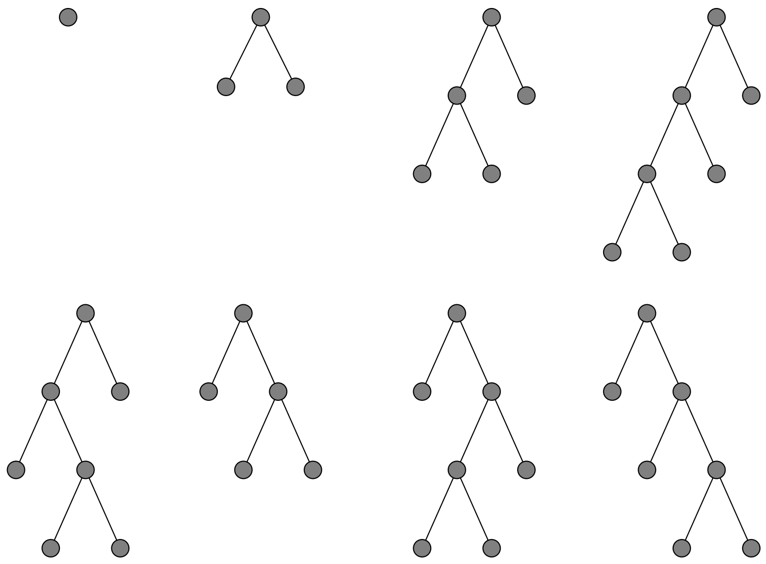
Theorem 3.3 (Optimalität von Uniform-Cost)
Gilt $\exists\delta > 0 : \; g(succ(n)) \geq g(n) + \delta$ *so ist Uniform-Cost optimal.*

Depth-First-Suche: “Knoten der grössten Tiefe werden zuerst expandiert”.

Queing-Fn($Queue, Expand(\ldots)$) : neue Knoten werden am Anfang eingefügt

Falls Verzweigungsfaktor b konstant und max. Tiefe m:

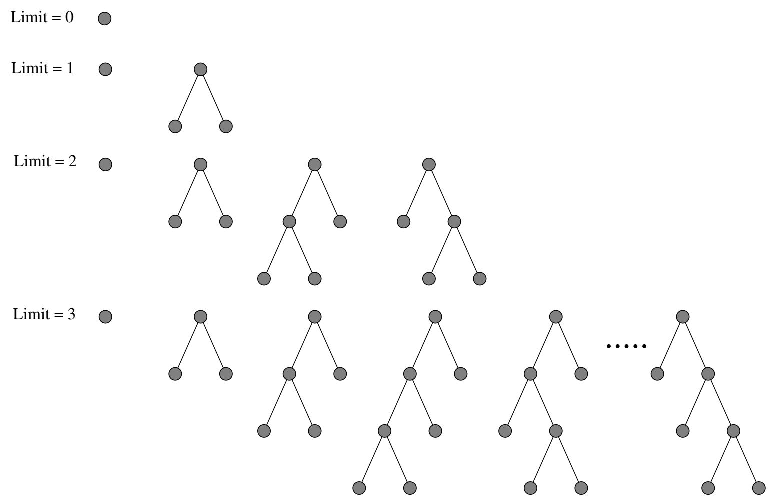
- **Speicherkomplexität:** $b \times m$,
- **Zeitkomplexität:** b^m.



Depth-First mit Tiefenbeschränkung d: "gehe höchstens in Tiefe d".

Insgesamt gibt es $1 + b + b^2 + \ldots + b^{d-1} + b^d$-viele Expansionsschritte.

Iterative Deepening: "Tiefe wird immer grösser"



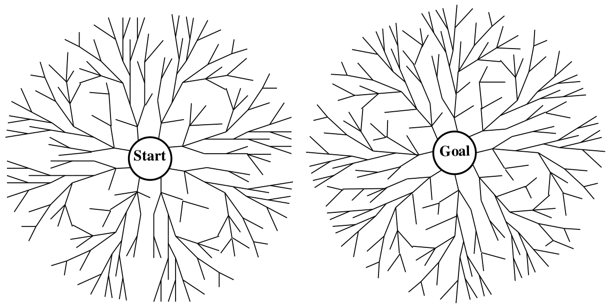

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure
```

Wieviele Expansionsschritte gibt es insgesamt?

Genau $(d+1) \times 1 + d \times b + (d-1) \times b^2 + \ldots + 2 \times b^{d-1} + 1 \times b^d$-viele.

Man beachte, daß nur etwa doppelt soviele Schritte im Vergleich zu Depth-First mit Tiefenbeschränkung ausgeführt werden ($\rightsquigarrow$ geometrische Reihe). Der Aufwand für mehrfach berechnete Knoten hält sich also in Grenzen.

Bidirektionale Suche:



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes