Rationale Agenten

Rationalität des Agentenverhaltens hängt ab von

- Performanzmaß (auf dem Ergebnis der Agenten*funktion*)
- vorgegebenem Wissen des Agenten über die Umgebung
- ausführbaren Aktionen
- aktueller Perzept-Folge

Ein Agent ist rational, wenn er stets die Aktion ausführt, die seine Performanz unter der aktuellen Perzept-Folge und seinem gegebenen Wissen maximiert.

- rational $\neq$ allwissend, hellseherisch, erfolgreich
- rational $\Rightarrow$ explorierend, lernend, autonom

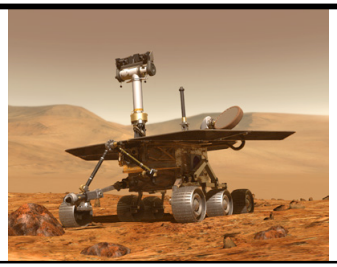
PEAS-Beschreibungen

Performance, **E**nvironment, **A**ctuators, **S**ensors –
Informelle Sammlung Rationalitäts-relevanter Merkmale

Agententyp	Perf.-Maß	Umgebung	Aktuatoren	Sensoren
 Staubsauger	billig, leise, gründlich, energiesparend, bei Abwesenheit,…	Haus, Treppen, Steckdosen, Menschen, Spielzeug, …	Räder, Lader, Saugpumpe, Lautsprecher, Müllauswurf, …	Stoßmelder, Dreckdetektor, Kamera, Mikro, Odometrie, …
 Marsroboter	zuverlässig, robust, ausdauernd, selbstbootend, …	Sand, Staub, Steine, Hitze, Kälte, Stoß, Strahlung, …	Räder, Kameramotoren, Probenehmer, Sender, …	Odometrie, Kamera, Chemiemodul, Empfänger, …

http://marsrovers.jpl.nasa.gov/home/

Eigenschaftsspektren von Umgebungen

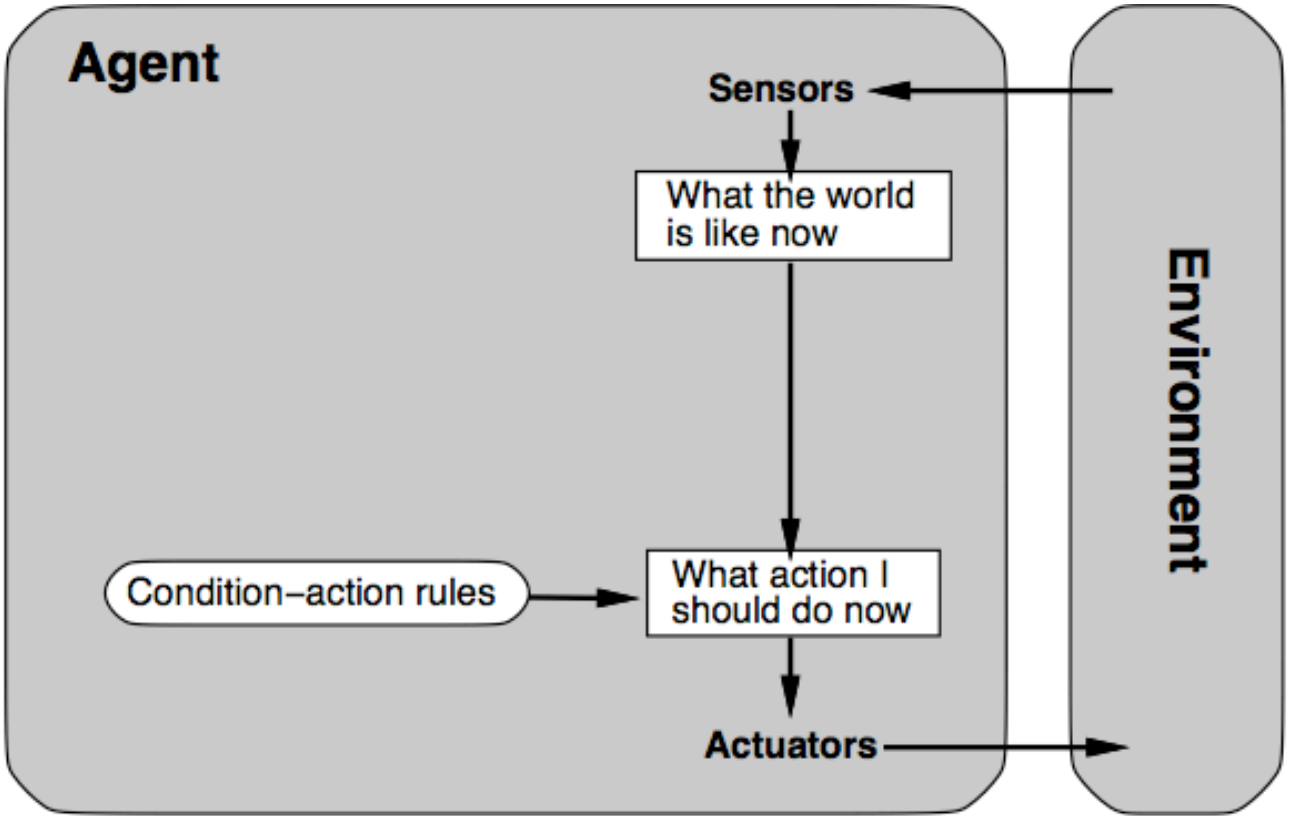
			Schach
vollständig/partiell beobachtbar	partiell	partiell	vollständig
deterministisch/ stochastisch	„mäßig stochastisch“	„mäßig deterministisch“	„strategisch“
episodisch nicht-episodisch	nicht-episodisch	nicht-episodisch	nicht-episodisch
statisch/dynamisch	dynamisch	dynamisch	„semi-statisch“
diskret/ kontinuierlich	kontinuierlich	kontinuierlich	diskret
mono-/multi-agentisch	multi-agentisch(?)	mono-agentisch	bi-agentisch (2)

Agententypen

- einfacher Reflex-Agent (s.o. REFLEX-VACUUM-AGENT!)
- Reflex-Agent mit innerem Zustand
- Ziel verfolgender Agent
- Nutzen maximierender Agent

Alle in mehr oder weniger explorierenden, lernenden, autonomen Varianten machbar!

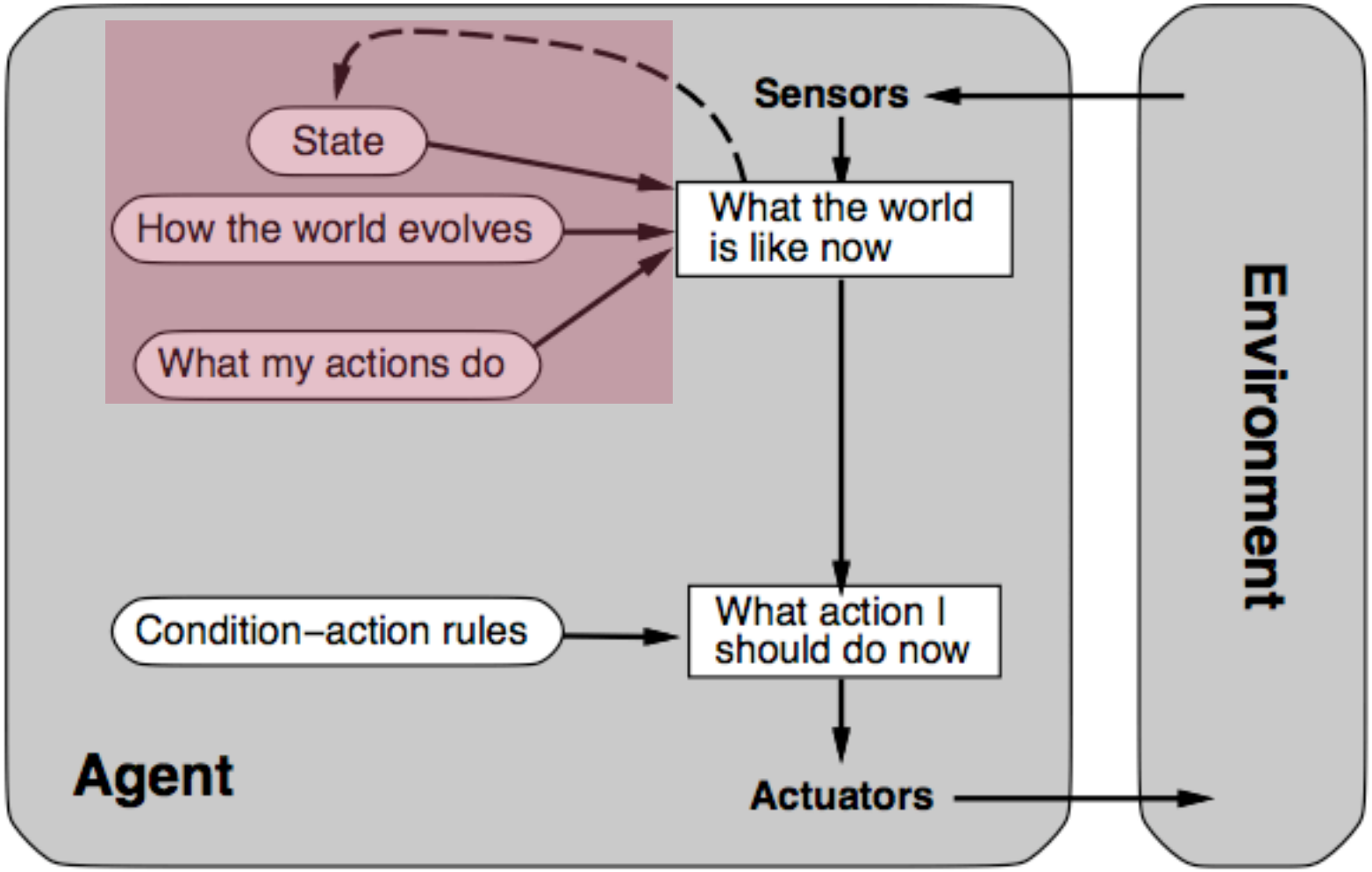
Programmschema: Einfacher Reflex-Agent



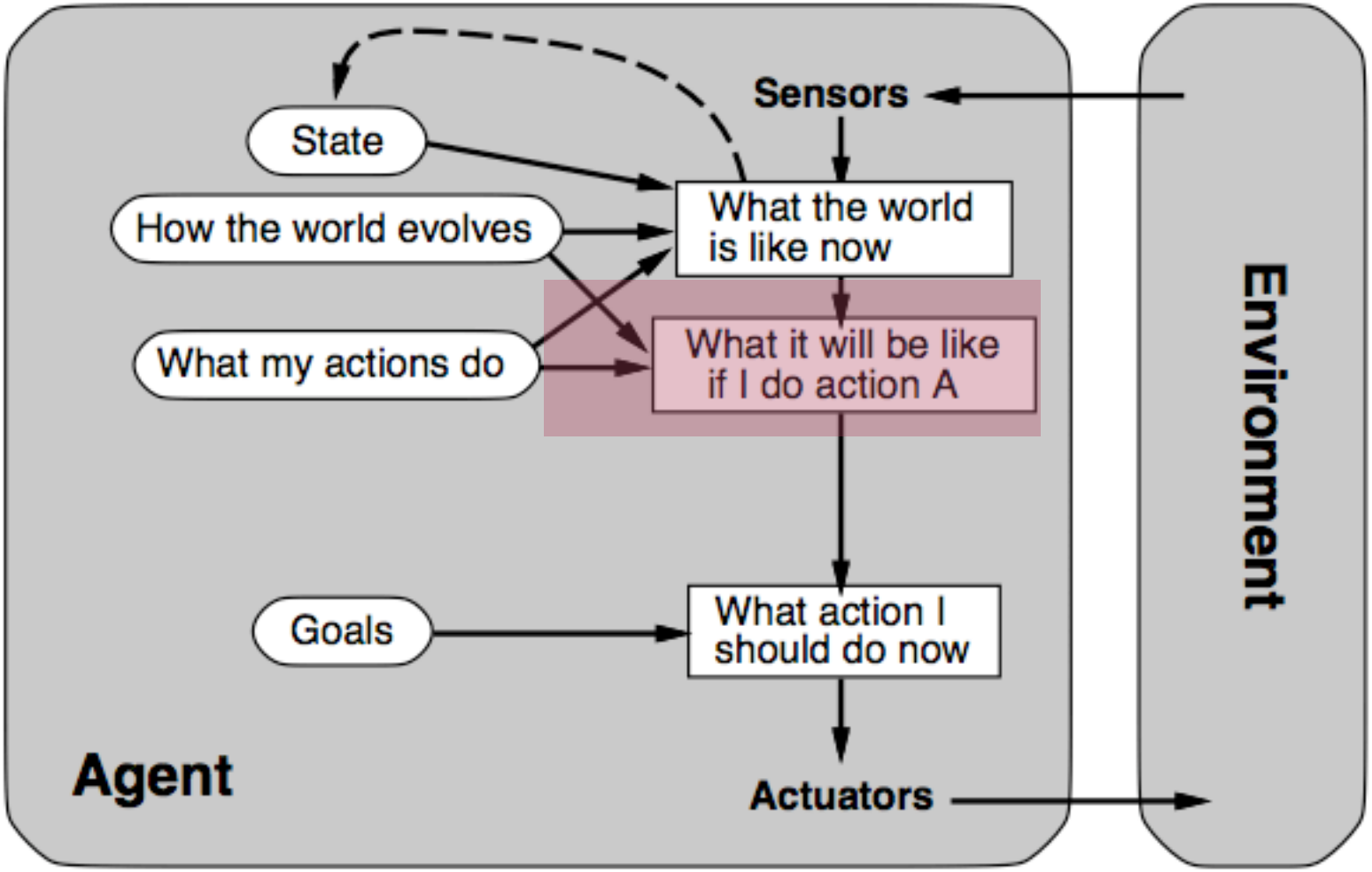
function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

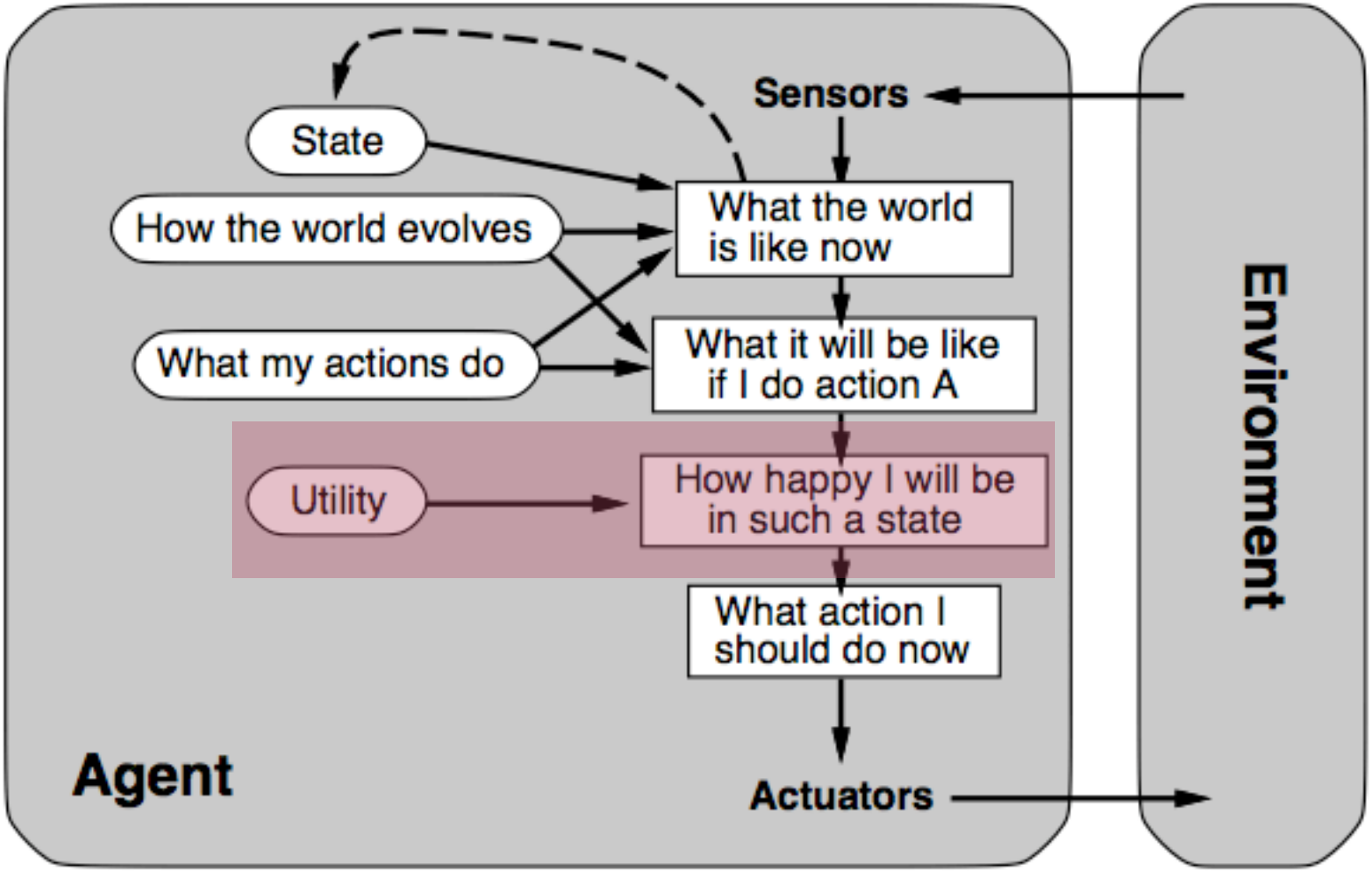
Programmschema: Reflexagent mit Zustand



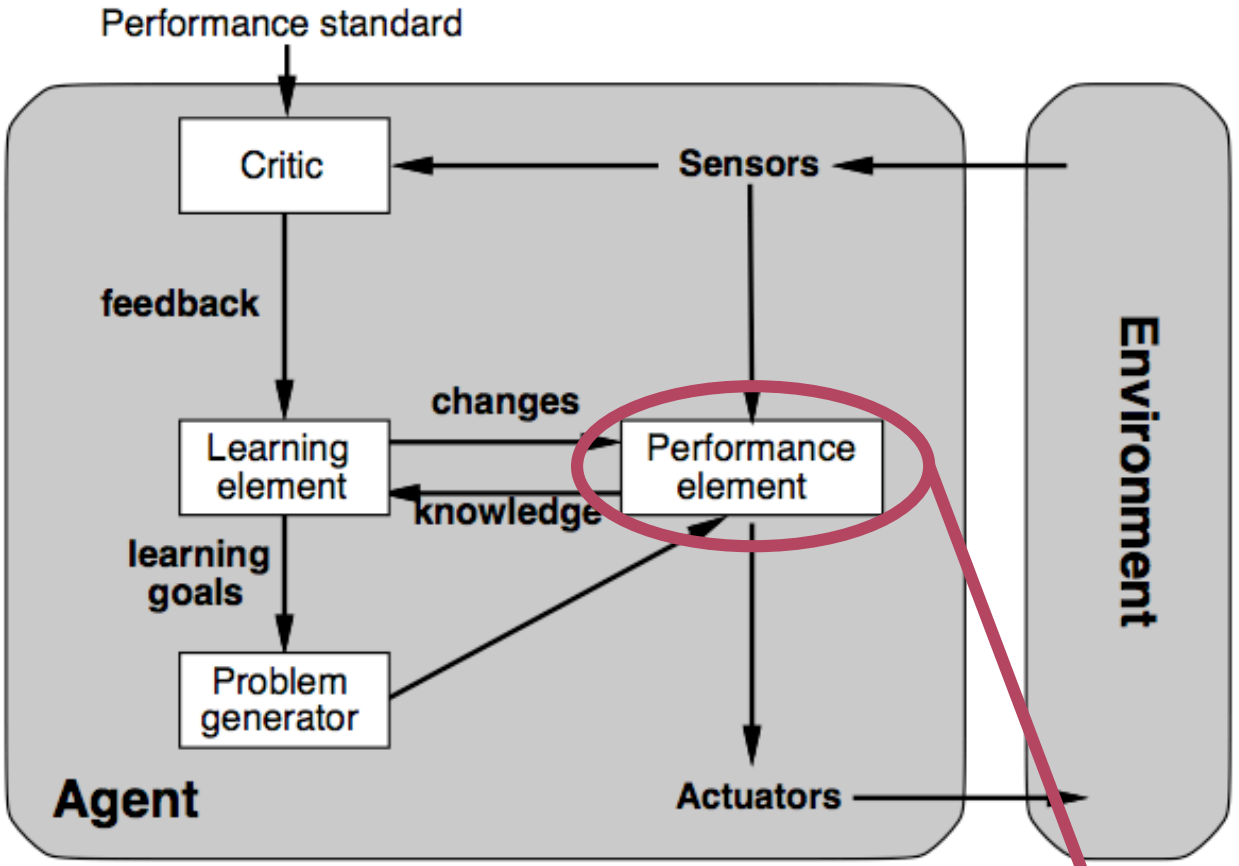
Programmschema: Ziele verfolgender Agent



Programmschema: Nutzen maximierender Agent



Programmschema: Lernender Agent

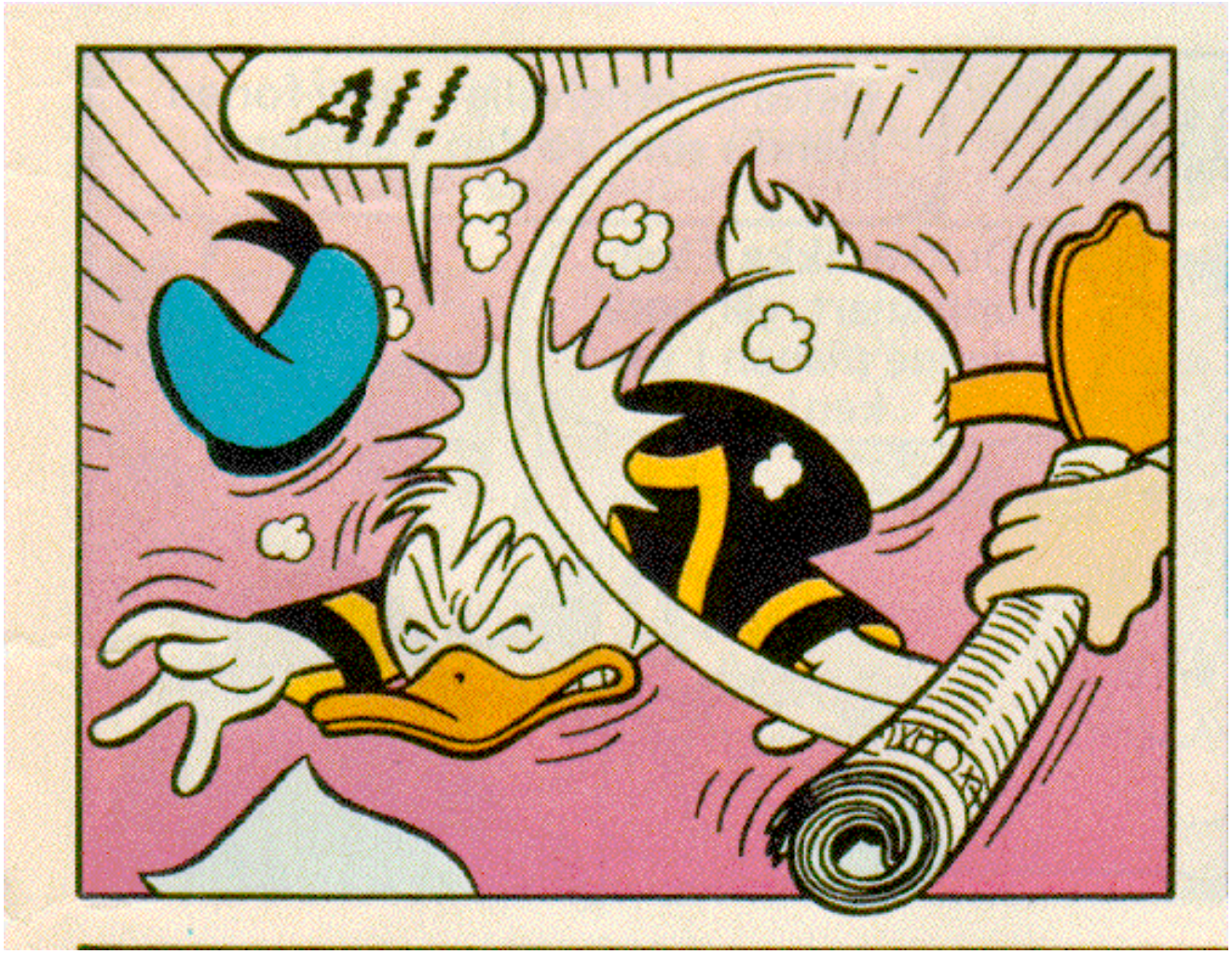


Eines der vorgenannten Agentenprogramme

Vorlesungsteil 1 – Was haben wir gelernt?

- Was ist KI, und was nicht? (grob)
- Agenten als Strukturrahmen für KI-Systeme
- Merkmale von Agenten und ihren Umgebungen
- Grobe Unterstrukturen von Agenten

Bilder-Suche mit Google nach „AI“



Gliederung

1. KI im Allgemeinen und in dieser Vorlesung
2. **Heuristische Suche**
3. Logik und Inferenz
4. Wissensrepräsentation
5. Handlungsplanung
6. Lernen
7. Sprachverarbeitung
8. Umgebungswahrnehmung

- **Suche im Allgemeinen**
- **Uninformierte Suche**
- **Heuristiken**
- **Constraint Satisfaction**

Suche in der Informatik

Typische Problemstellung

„Ist ein Datensatz in einer Datenbank vorhanden?“

Naive Lösung

Alle Datensätze der Reihe nach durchsuchen.

Zeit: $O(n)$ für n Datensätze

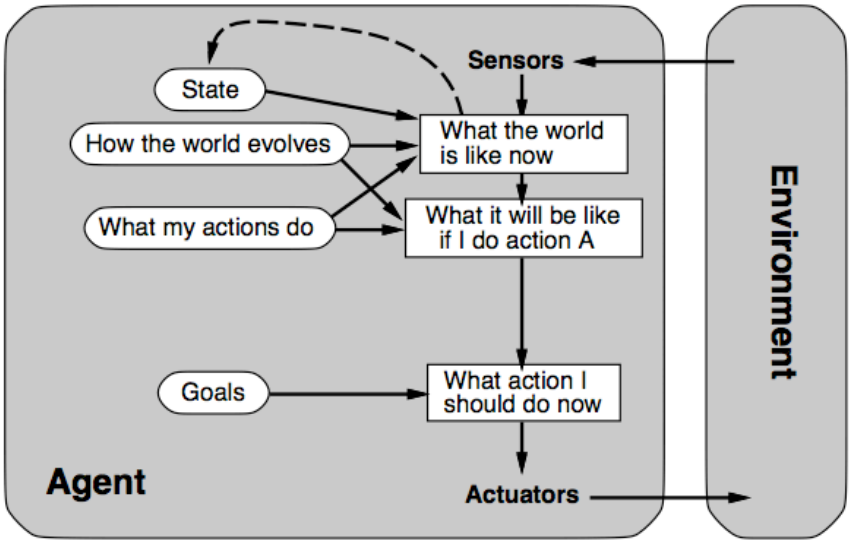
Bessere Lösung (wie in Informatik A gelernt)

Datensätze clever sortiert speichern (z.B. Baum);
Sortierung beim Suchen nutzen.

Zeit: $O(\log n)$

⇒ D.E. Knuth, Bd.3: *Sorting and Searching*; Informatik A

Suche in der KI



Typische Problemstellung

Ziele verfolgender Agent überlegt
den nächsten Schritt,
und den Folgeschritt,
und den … … — bis zum Ziel

Lösungsweg konstruieren,
nicht Ziel nachschlagen!

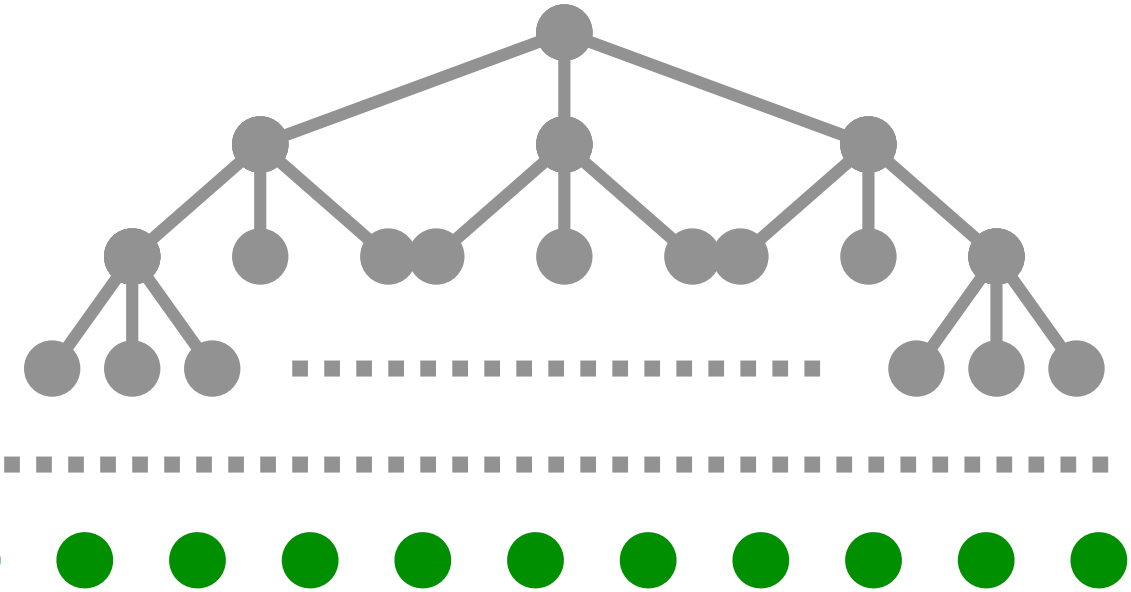
Naive Lösung

Alle Sequenzen von Schritten der Reihe nach durchprobieren.

Zeit: …

Exkurs über Knoten und Blätter in Bäumen

Verzweigungsfaktor b=3



Tiefe

d=0

d=1

d=2

d=3

d

$O(b^d)$ # Knoten der Tiefe d: b^d

$O(b^d)$ # alle Knoten bis einschl. Tiefe d: $\sum_{i=0}^{d} b^i = \frac{b^{d+1}-1}{b-1}$

Komplexität der naiven Suche

Alle Sequenzen von Schritten der Reihe nach durchprobieren.

Zeit: $O(b^d)$ bei „erster“ Lösung in Tiefe d

Speicher: dito (alle Knoten im Speicher)

Bessere Lösungen … folgen!

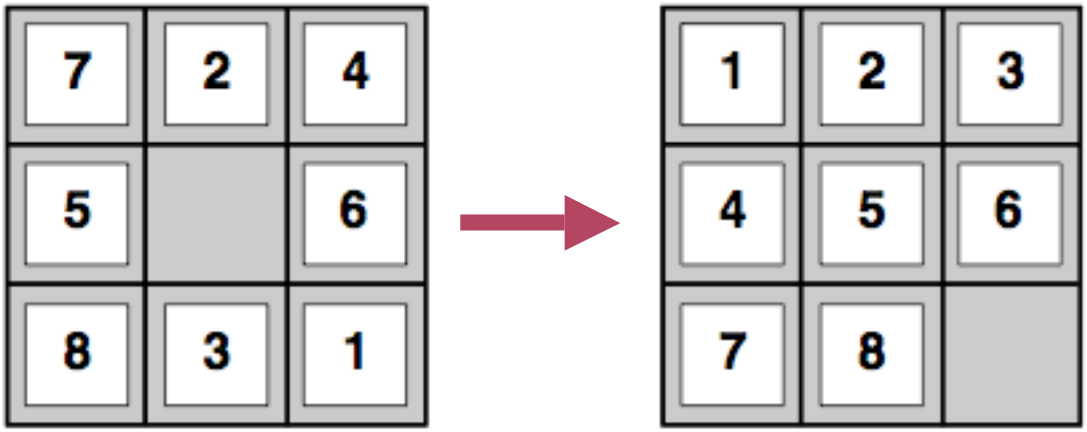
Problem-Solving Agent

```
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
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Inhalt dieses Kapitels

Beispielproblem I: Verschiebespiel



Startzustand **Zielzustand**

Zustand

Sequenz der Zahlen/ Leerfeld auf den 9 Feldern

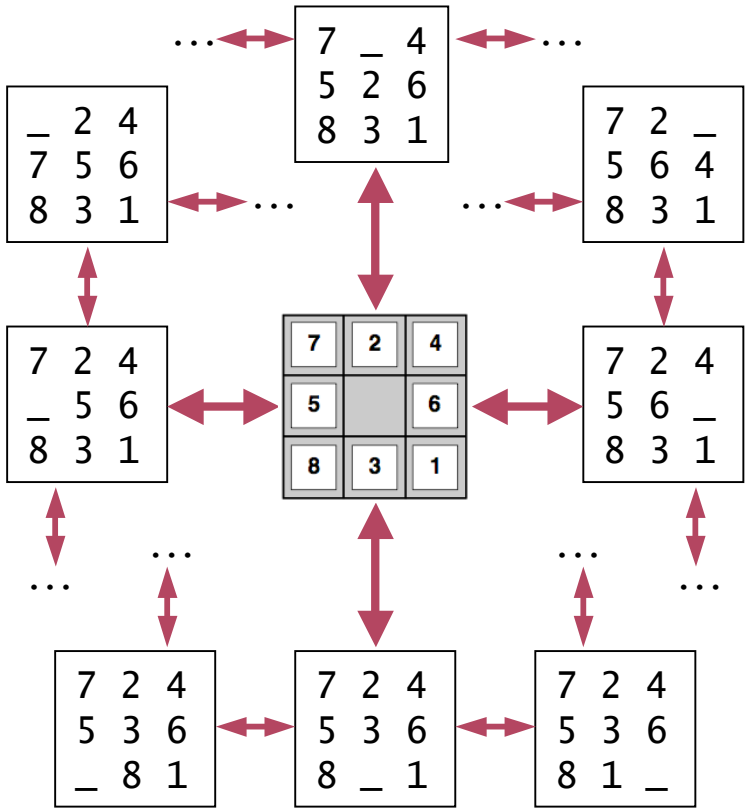
Aktionen

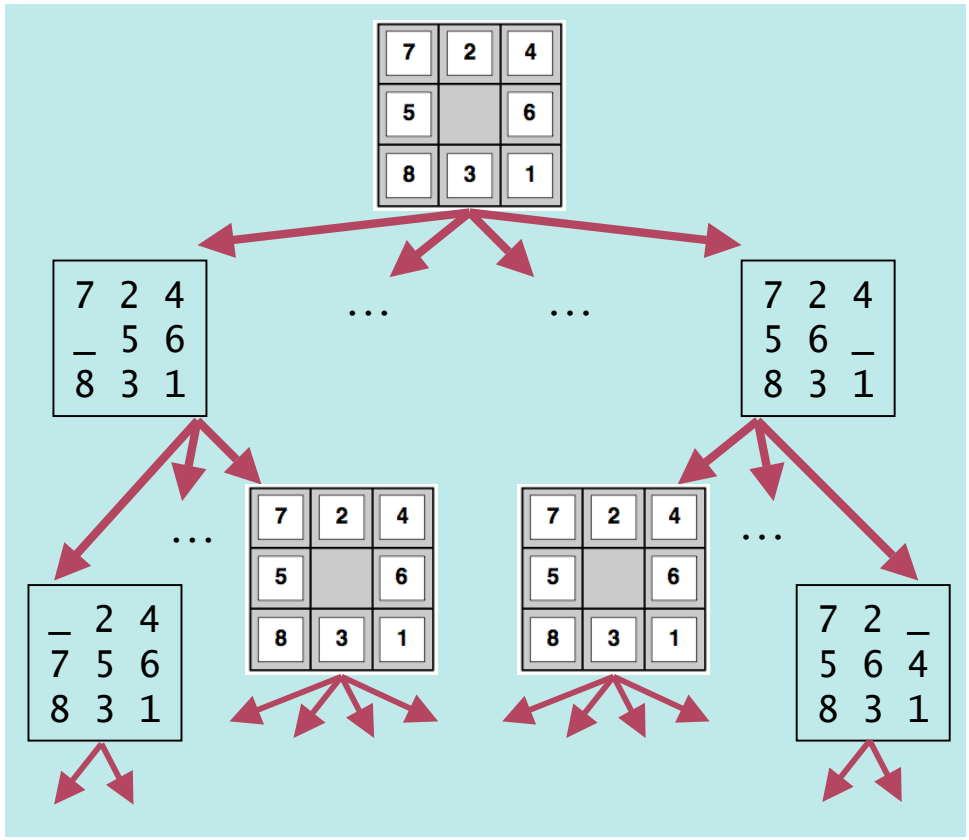
Left, Right, Up, Down (Verschiebung des Leerfelds)

Kosten

Konstant (1) pro Aktion

Problemraum und Suchraum





Traversiere **Graph** als **Baum**

Zyklen werden zu unendlichen Pfaden!

Baumsuche im Allgemeinen

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

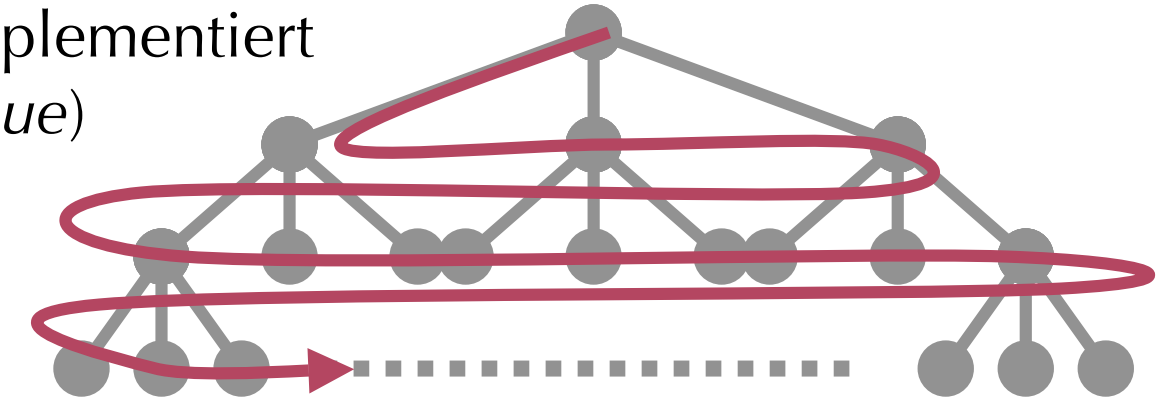
EXPAND ermittelt die Nachfolgezustände
und „verpackt“ sie als Knoten im Suchbaum
(Knotenkonstruktor, Fig.3.9)

Achtung Falle: Zieltest erst bei Expansion, nicht schon beim Einfügen!
Erkennt Ziel der Tiefe d erst auf Ebene $(d+1)$

Breitensuche

Siehe Vorlesung Informatik A!

Funktion INSERT-ALL implementiert
als Warteschlange (*queue*)

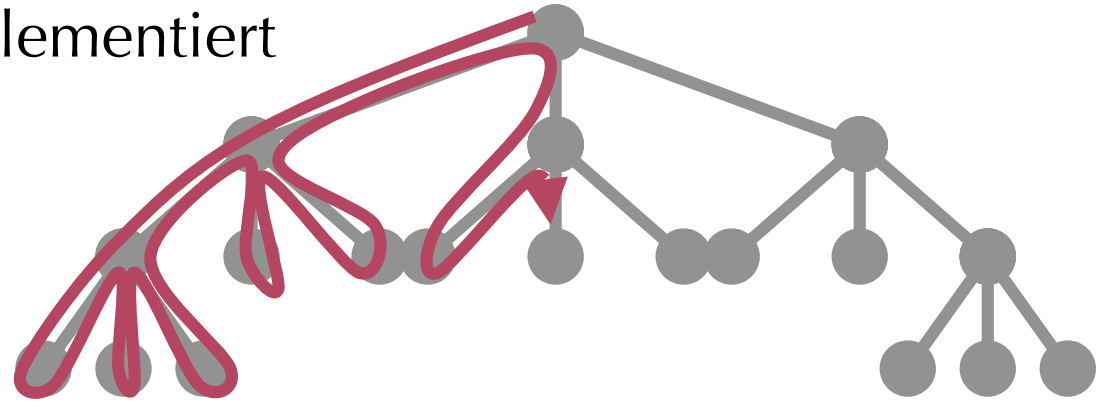


- ☹ Zeitbedarf: $O(b^{d+1})$ (Exponent d bei konstanten Aktionskosten!)
- ☹ Speicherbedarf: $O(b^{d+1})$ (Exponent d bei konstanten Aktionskosten!)
- ☺ Vollständig: Wenn Lösung existiert, wird sie gefunden
- ☺ Optimal bei konstanten Aktionskosten

Tiefensuche

Siehe Vorlesung Informatik A!

Funktion INSERT-ALL implementiert
als Keller (*stack*)



- ☹ Zeitbedarf: $O(b^m)$, wenn m Maximaltiefe des Baums
- ☺ Speicherbedarf: $O(bm)$
- ☹ Unvollständig
- ☹ Nicht optimal

Tiefensuche „taucht ab“
auf unendlichen Suchpfaden!