

Intelligente Systeme

Erweiterungen der Fähigkeiten eines Agenten

Prof. Dr. R. Kruse C. Braune C. Moewes

{kruse,cmoewes,russ}@iws.cs.uni-magdeburg.de

Institut für Wissens- und Sprachverarbeitung
Fakultät für Informatik
Otto-von-Guericke Universität Magdeburg

Erweiterung der Fähigkeiten von Agenten

Bisher: S-R-Agenten mit unmittelbarer Reaktion auf Sensorreize
Jetzt: Ausnutzung von Sensorinformationen aus Vergangenheit

Übersicht

1. Temporale Informationen

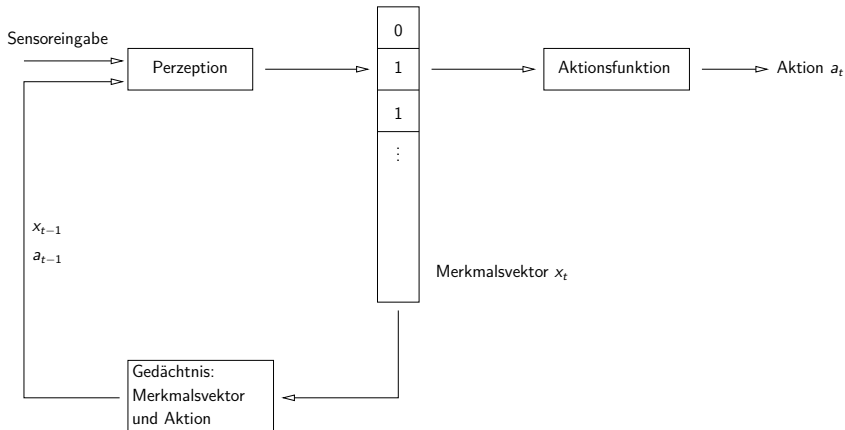
Beispiel: Roboter in Gitterwelt

2. Räumliche Informationen

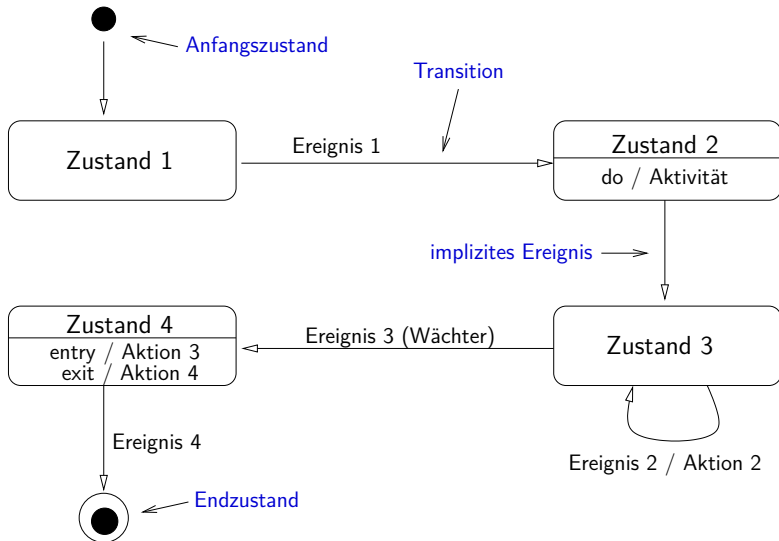
3. Informationsfusion

4. Problemlösung

Integration von zeitlichen Informationen



Zustandsagenten (erweitert)



Beispiel: Roboter in Gitterwelt (1)

Roboter in Gitterwelt mit begrenzter Sensorinformation

Sensoreingabe zum Zeitpunkt t :

- $s_2^t, s_4^t, s_6^t, s_8^t$ (d.h. nur 4 statt bisher 8 Sensoren)
- $s_i^t = 1 \leftrightarrow$ Feld s_i^t ist nicht frei

Aufgabe: Wandverfolgung

Idee: nutze Merkmalsvektor des jeweils vorherigen Zeitpunkts

Beispiel: Roboter in Gitterwelt (2)

Definition der Merkmalsvektoren:

$$w_i^t = s_i^t \text{ für } i = 2, 4, 6, 8$$

$$w_1^t = 1 \leftrightarrow w_2^{t-1} = 1 \text{ and } a_{t-1} = \text{east}$$

$$w_3^t = 1 \leftrightarrow w_4^{t-1} = 1 \text{ and } a_{t-1} = \text{south}$$

$$w_5^t = 1 \leftrightarrow w_6^{t-1} = 1 \text{ and } a_{t-1} = \text{west}$$

$$w_7^t = 1 \leftrightarrow w_8^{t-1} = 1 \text{ and } a_{t-1} = \text{north}$$

⇒ (teilweiser) Ausgleich eingeschränkter Sensorinformationen möglich!

Beispiel: Roboter in Gitterwelt (3)

Sinnvolle Aktionen zur Wandverfolgung:

$$w_2^t \wedge \neg w_4^t \rightarrow \text{east}$$

$$w_4^t \wedge \neg w_6^t \rightarrow \text{south}$$

$$w_6^t \wedge \neg w_8^t \rightarrow \text{west}$$

$$w_8^t \wedge \neg w_2^t \rightarrow \text{north}$$

$$w_1^t \wedge \neg w_2^t \rightarrow \text{north}$$

$$w_3^t \wedge \neg w_4^t \rightarrow \text{east}$$

$$w_5^t \wedge \neg w_6^t \rightarrow \text{south}$$

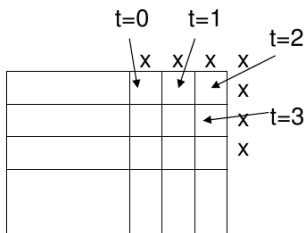
$$w_7^t \wedge \neg w_8^t \rightarrow \text{west}$$

$$\text{alle } w_i = 0 \rightarrow \text{north}$$

Beispiel: Roboter in Gitterwelt (4)

Implementierung (Beispiel einer Bewegung):

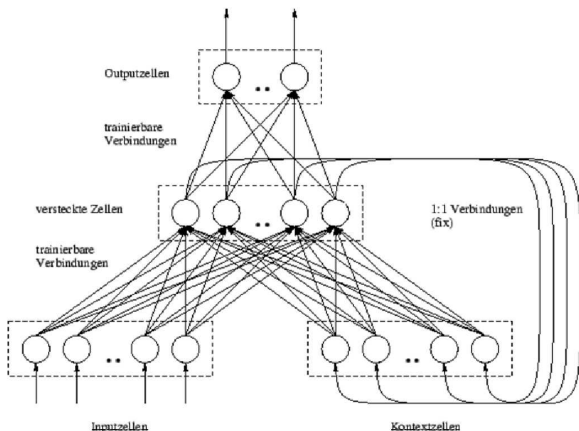
	Sensoreingabe				Merkmalsvektor								Aktion
t	s_2^t	s_4^t	s_6^t	s_8^t	w_1^t	w_2^t	w_3^t	w_4^t	w_5^t	w_6^t	w_7^t	w_8^t	a_t
0	1	0	0	0	0	1	0	0	0	0	0	0	east
1	1	0	0	0	1	1	0	0	0	0	0	0	east
2	1	1	0	0	1	1	0	1	0	0	0	0	south
2'	0	0	0	0	1	0	0	0	0	0	0	0	north



Anmerkung: Situation t=2' z.B. bei Sensorstörung (alle Sensoren $s_i=0$)

Einfache RNNs: Elman-Netze

Einführung einer Kontextschicht
ermöglicht Speichern von Informationen



Beispiel: Elman-Netz für Roboter

Elman-Netz für Roboter in Gitterwelt:

8-dimensionale Merkmalsvektoren (i.A. Anzahl der Dimensionen unbekannt)

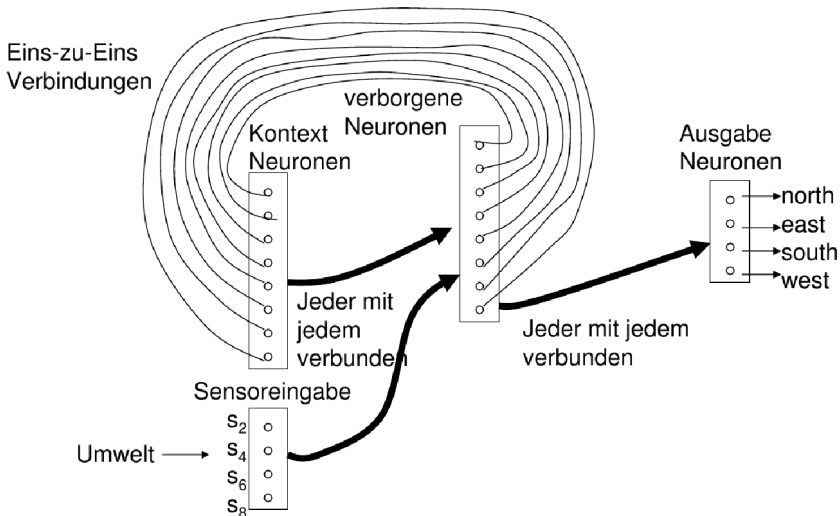
4 Eingaben (Sensoren)

4 Ausgaben (Richtungen; Ausgabe mit größtem Wert wird gewählt)

Training durch Backpropagation

„Lernfähige“ Automaten

Beispiel: Elman-Netz für Roboter



Übersicht

1. Temporale Informationen

2. Räumliche Informationen

3. Informationsfusion

4. Problemlösung

Integration von räumlichen Informationen

Bisher: nur Informationen über einen sehr kleinen
Umgebungsausschnitt:

- Unmittelbare Nachbarschaft

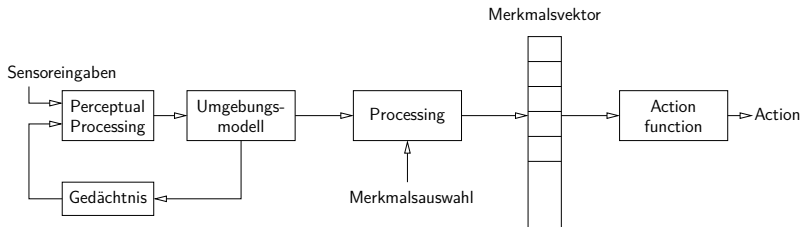
- Gespeichert in Merkmalsvektor

Idee der Umgebungsmodelle:

- Speicherung möglichst aller bereits gesammelter Informationen
über Umgebung

- Nutzung geeigneter Datenstrukturen wie z.B. Landkarten

Beispiel: Umgebungsmodell für Gitterwelt (1)



Beispiel: Umgebungsmodell für Gitterwelt (2)

1	1	1	1	1	?	?
1	0	0	0	0	0	?
1	0	0	0	0	0	?
1	0	0	R	0	0	?
1	0	0	0	0	0	?
1	?	?	?	?	?	?
?	?	?	?	?	?	?

1: belegt, 0: frei, ?: unbekannt, R: Roboter

Mögliche Aktion basierend auf Informationen: *go west (or north)*
and follow wall

Umgebungsmodelle: Aktionen

Aktionen z.B. über zwei-dimensionale Potentialfelder bestimmen
Potentialfelder: Überlagerung von anziehenden (“attractive”) und abstoßenden (“repulsive”) Komponenten

Bewegung des Roboters: absteigender Richtung des Gradienten (lokale Minima!)

Bewegungsrichtung: vorberechnet oder online (z.B. in sich ändernden Umgebungen)

Umgebungsmodelle: Potentialfeld für Gridworld

anziehende Komponente:

Durch Zielfeld erzeugt: $p_a(x^{(\rho)}) = k_1 \cdot d(x^{(\rho)})^2$

k_1 : konstanter Faktor, d : Abstand zum Zielfeld

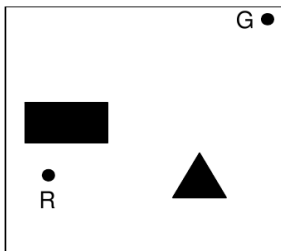
abstoßende Komponente(n):

Durch Hindernisse erzeugt: $p_r(x^{(\rho)}) = \frac{k_2}{d(x^{(\rho)})^2}$,

wobei k_2 konstanter Faktor, d Abstand zum Hindernis

Insgesamt: $p = p_a + p_r$

Potentialfelder: Beispielumgebung

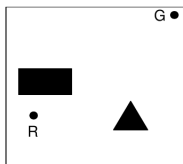


Roboter (R)

Ziel (G)

Hindernisse (◆)

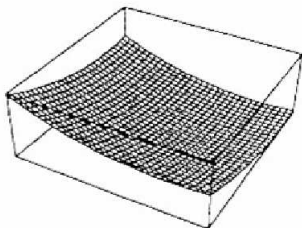
Potentialfelder: Potentialfeldkomponenten



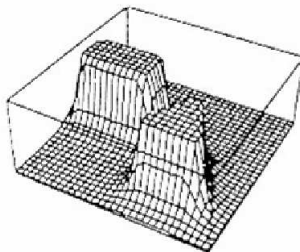
Roboter (R)

Ziel (G)

Hindernisse (◆)

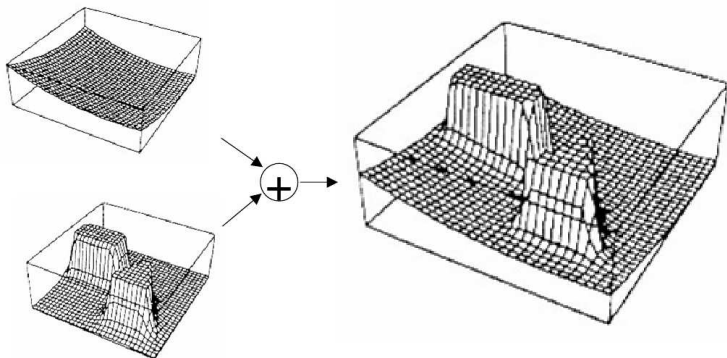


Ziel

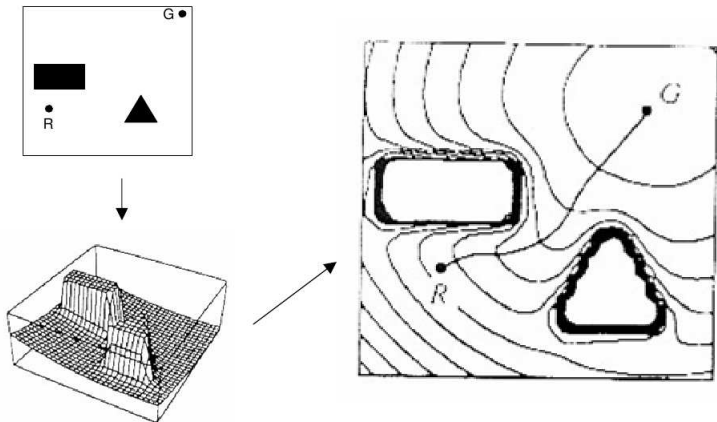


Hindernisse

Potentialfelder: Gesamtes Potentialfeld



Äquipotentiallinien (für Gradientenverfahren)



Übersicht

1. Temporale Informationen

2. Räumliche Informationen

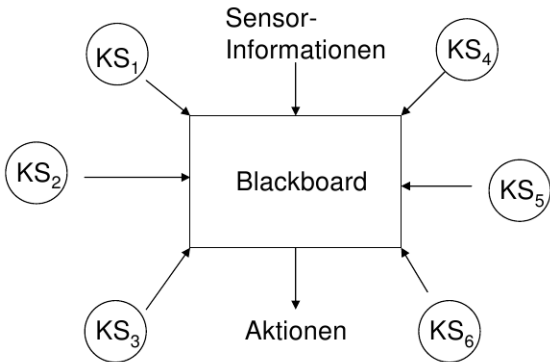
3. Informationsfusion

4. Problemlösung

Beispiel: Blackboard-Systeme

Blackboard: Spezielle Datenstruktur

Knowledge Source (KS): Programm zum Lesen und Schreiben des Blackboards



Blackboard-Systeme: Knowledge Source

Bestandteile:

Bedingungsteil (berechnet Wert eines Merkmals)

Aktionsteil (Programm zum Lesen/Schreiben des Blackboards und/oder zum Ausführen externer Aktionen)

Konfliktlöser entscheidet bei Ausführung von zwei KS, welche gewählt wird

KS = „Experte“ eines Teils des Blackboards, den es überwacht

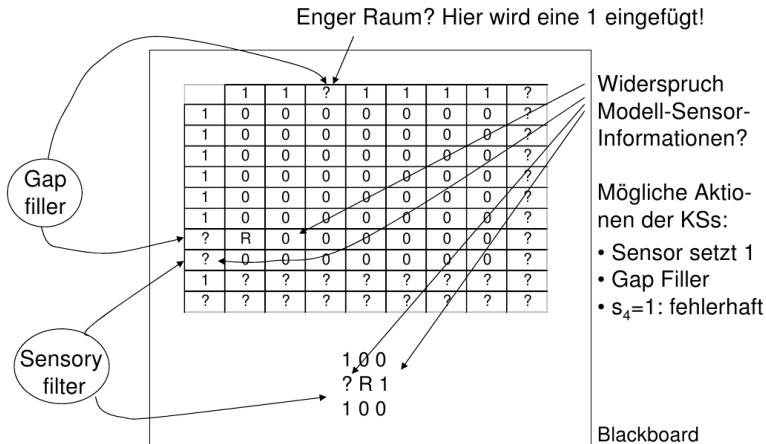
Blackboard-Systeme: Beispiel (1)

Weltmodell im Roboter: kann unvollständig/falsch sein (Grund: Sensorfehler)

Mögliche Wissensquellen:

- Lückenfüller (Gap Filler): sucht nach engen Räumen (tight spaces) im gelernten Umgebungsmodell und markiert Feld bzw. korrigiert ggf. vorhandene Fehler
- Sensorfilter (Sensory Filter): vergleicht Sensorinformationen mit gelerntem Umgebungsmodell und versucht Fehler zu beseitigen

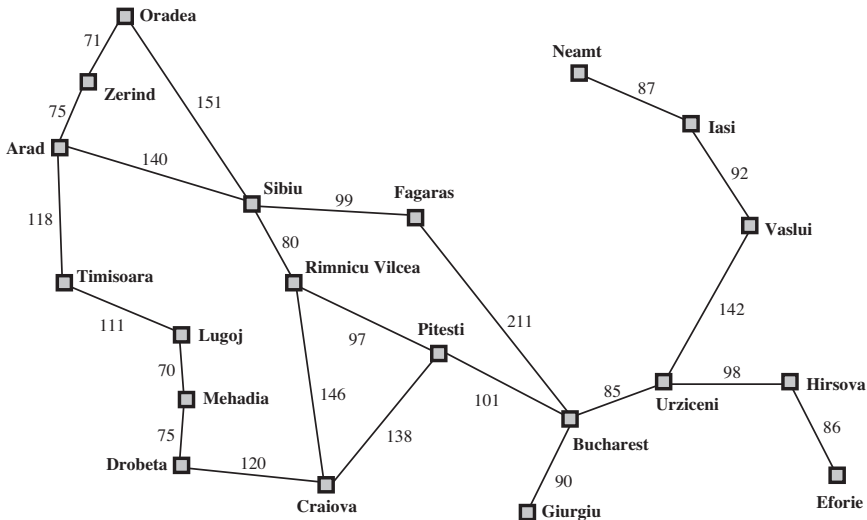
Blackboard-Systeme: Beispiel (2)



Übersicht

1. Temporale Informationen
2. Räumliche Informationen
3. Informationsfusion
- 4. Problemlösung**

Beispiel: Routenplanung



Problemlösende Agenten

SIMPLE-PROBLEM-SOLVING-AGENT

Eingabe: Wahrnehmung *percept*

Ausgabe: eine Aktion *action*

- 1: **static** *seq*: Aktionssequenz (anfangs leer)
- 2: **static** *state*: momentane Beschreibung der Welt
- 3: **static** *goal*: Ziel (anfangs null)
- 4: **static** *problem*: Problembeschreibung
- 5: *state* \leftarrow UPDATE-STATE(*state*, *percept*)
- 6: **if** *seq* is empty {
- 7: *goal* \leftarrow FORMULATE-GOAL(*state*)
- 8: *problem* \leftarrow FORMULATE-PROBLEM(*state*, *goal*)
- 9: *seq* \leftarrow SEARCH(*problem*)
- 10: }
- 11: *action* \leftarrow RECOMMENDATION(*seq*, *state*)
- 12: *seq* \leftarrow Remainder(*seq*, *state*)
- 13: **return** *action*

Arten von Problemen

deterministisch (vollständig beobachtbar)

Agent weiß genau in welchem Zustand er sein wird

Lösung ist eine Sequenz von Aktionen

nicht beobachtbar \Rightarrow **konformantes Problem**

Agent hat u.U. keine Ahnung in welchem Zustand er sich befindet

Lösung (falls existent) ist eine Sequenz von Aktionen

nichtdeterministisch (teilw. beobachtbar) \Rightarrow **Zufallsproblem**

Wahrnehmungen: neue Infos über momentan Zustand

Lösung: ungewisser Plan oder Strategie

Oftmals hängen Suche und Ausführung voneinander ab

unbekannter Zustandsraum \Rightarrow **Explorationsproblem** (“online”)

Wissen über einen bestimmten Teil des Zustandsraum kann durch Suchverfahren maximiert werden

Ein klares Ziel, das zu erreichen wäre, gibt es nicht

Ansatz für deterministische Probleme

Ein **Problem** sei definiert durch 4 Begriffe

Anfangszustand, z.B. *Arad*

Nachfolgerfunktion $S(x)$ = Menge aller
Aktions-Zustands-Paare, z.B.

$S(\textit{Arad}) = \{\langle \textit{Arad} \rightarrow \textit{Zerind}, \textit{Zerind} \rangle, \dots\}$

Zieltest

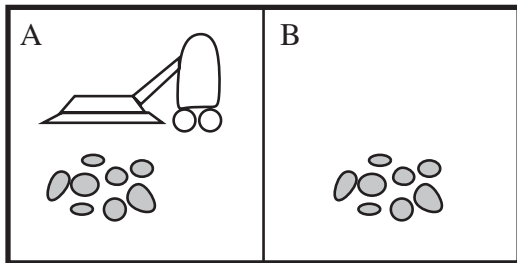
- Explizit, z.B. $x = \textit{Bukarest}$
- Implizit, z.B. $\textit{NoDirt}(x)$

Wegkosten (zusätzlich)

- Summe der Abstände, Anzahl ausgeführter Aktionen, etc.
- $c(x, a, y)$ seien Schrittkosten mit ≥ 0

dann ist eine **Lösung** eine Sequenz von Aktionen vom Anfangs- bis zum Endzustand

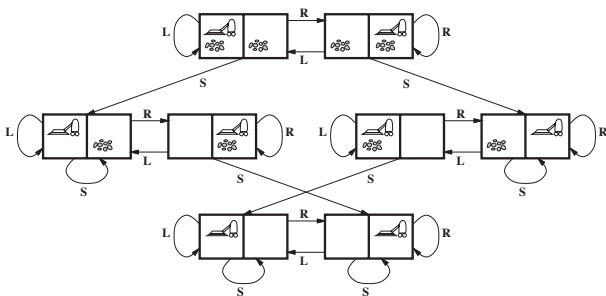
Staubsaug-Welt



Wahrnehmung: Ort und Status, z.B. [A, *Dirty*]

Aktionen: *Left*, *Right*, *Suck*, *NoOp*

Beispiel: Zustandsgraph der Staubsaug-Welt



Zustände: ganzzahliger Schmutz und Ort des Agenten

Aktionen: *Left*, *Right*, *Suck*, *NoOp*

Zieltest: kein Schmutz

Pfadkosten: 1 pro Aktion (0 für *NoOp*)

Ein Staubsaug-Agent

Sequenz von Wahrnehmungen	Aktion
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
:	:

REFLECTIVE-VACUUM-AGENT

```

1: if staus = Dirty {
2:   return Suck
3: } else {
4:   if location = A {
5:     return Right
6:   } else {
7:     return Left

```

Beispiel: Das 8-Puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Zustände: ganzzahlige Positionen der Plättchen

Aktionen: bewege Lücke *links*, *rechts*, *hoch*, *runter*

Zieltest: Zielzustand (gegeben)

Pfadkosten: 1 pro Zug

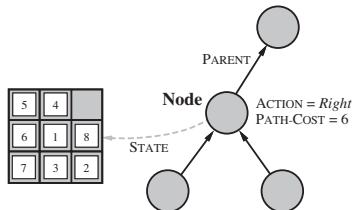
Hinweis: optimale Lösung der n -Puzzle-Familie ist NP-schwer

Implementierung: Zustände vs. Knoten

Zustand = physikalische Konfigurierung

Knoten = Datenstruktur (Teil eines Suchbaums mit Eltern, Kinder, Tiefe, Pfadkosten $g(x)$)

Zustände haben keine Eltern, Kinder, Tiefe, oder Pfadkosten $g(x)$



EXPAND-Funktion erzeugt neue Knoten

SUCCESSOR-Funktion erzeugt zugehörige Zustände

Implementierung: Generelle Baumsuche

TREE-SEARCH

Eingabe: Problembeschreibung *problem*, Rand *fringe*

Ausgabe: Lösung oder Fehler

```
1: seq ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
2: while true {
3:   if fringe is empty {
4:     return failure
5:   }
6:   node ← REMOVE-FRONT(fringe)
7:   if GOAL-TEST(problem, STATE(node)) {
8:     return node
9:   }
10:  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
11: }
```

Implementierung: Generelle Baumsuche

EXPAND

Eingabe: Knoten *node*, Problembeschreibung *problem*

Ausgabe: eine Menge von Knoten

- 1: **for each** *action*, *result* in $\text{SUCCESSOR}(\text{problem}, \text{STATE}[\text{node}])$ {
 - 2: $s \leftarrow \text{new NODE}$
 - 3: $\text{PARENT-NODE}[s] \leftarrow \text{node}$
 - 4: $\text{ACTION}[s] \leftarrow \text{action}$
 - 5: $\text{STATE}[s] \leftarrow \text{result}$
 - 6: $\text{PATH-COST}[s] \leftarrow \text{PATH-COST}[\text{node}] + \text{STEP-COST}(\text{node}, \text{action}, s)$
 - 7: $\text{DEPTH}[s] \leftarrow \text{DEPTH}[\text{node}] + 1$
 - 8: add *s* to *successors*
 - 9: }
 - 10: **return** *successors*
-

Problemlösung durch geeignete Suche

Suchstrategie = Reihenfolge der Expansion von Nachfolgerknoten
Bewertung anhand von

Vollständigkeit: Wird immer 1 Lösung gefunden falls eine existiert?

Zeitkomplexität: Anzahl der Knoten erzeugt/expandiert

Speicherkomplexität: maximale Anzahl von Knoten im Speicher

Optimalität: Wird immer 1 Lösung mit geringsten Kosten gefunden?

Zeit- und Speicherkomplexität gemessen anhand von

b maximaler Verzweigungsfaktor des Suchbaums

d Tiefe der Lösung mit geringsten Kosten

m maximale Tiefe des Zustandsraums (eventuell ∞)