



OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

INF

FAKULTÄT FÜR
INFORMATIK

Intelligente Systeme

Zustandsagenten, Problemlösen und Suchen

Prof. Dr. R. Kruse C. Moewes G. Ruß

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

Institut für Wissens- und Sprachverarbeitung

Fakultät für Informatik

Otto-von-Guericke Universität Magdeburg

Übersicht

1. Zustandsagenten

Beispiel: Roboter in Gitterwelt

Rückgekoppelte Neuronale Netze

Umgebungsmodelle

Blackboard-Systeme

2. Problemlösende Agenten

3. Uninformierte Suche

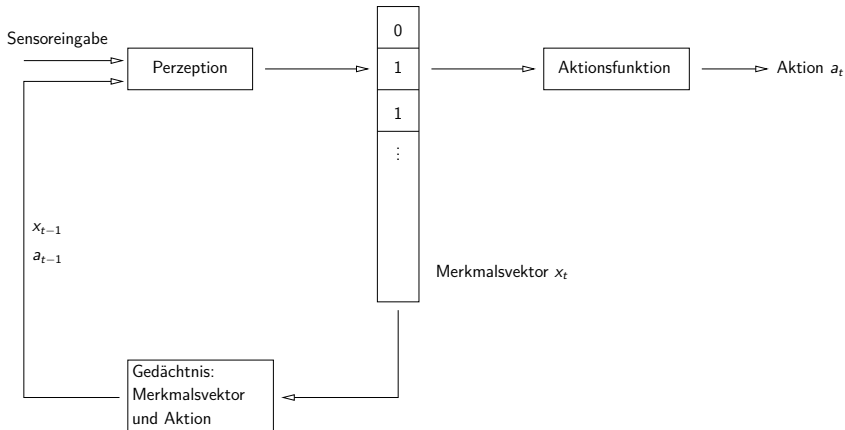
Zustandsagenten

bisher: S-R-Agenten mit unmittelbarer Reaktion auf Sensorreize jetzt:

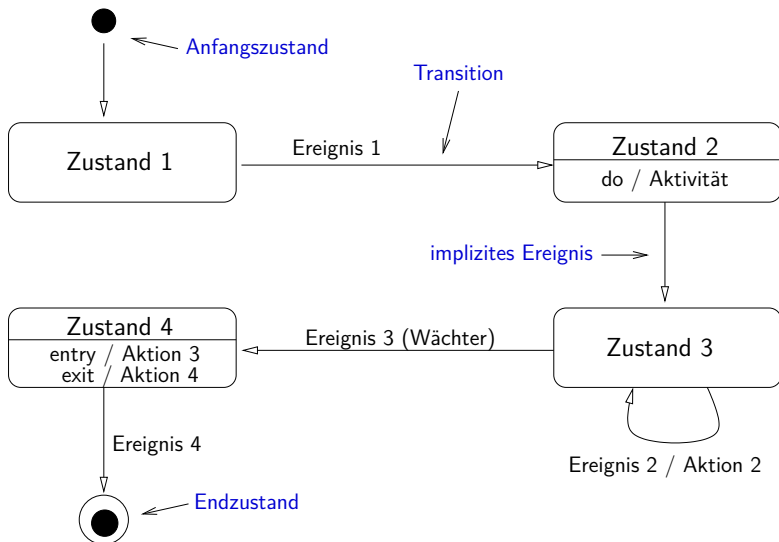
Ausnutzung von Sensorinformationen aus Vergangenheit

Zustandsagenten

- ▶ Darstellung der Umgebung mit Merkmalsvektoren
- ▶ Beispiel:



Zustandsagenten



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)
 - ▶ $s_j^t = 1 \leftrightarrow$ Feld s_j^t ist nicht frei
- ▶ Aufgabe: Wandverfolgung
- ▶ Idee: nutze Merkmalsvektor des jeweils vorherigen Zeitpunkts

Beispiel: Roboter in Gitterwelt (2)

Definition der Merkmalsvektoren:

- ▶ $w_i^t = s_i^t$ für $i = 2, 4, 6, 8$
- ▶ $w_1^t = 1 \leftrightarrow w_2^{t-1} = 1$ and $a_{t-1} = \text{east}$
- ▶ $w_3^t = 1 \leftrightarrow w_4^{t-1} = 1$ and $a_{t-1} = \text{south}$
- ▶ $w_5^t = 1 \leftrightarrow w_6^{t-1} = 1$ and $a_{t-1} = \text{west}$
- ▶ $w_7^t = 1 \leftrightarrow w_8^{t-1} = 1$ and $a_{t-1} = \text{north}$

⇒ (teilweiser) Ausgleich eingeschränkter Sensorinformationen möglich!

Beispiel: Roboter in Gitterwelt (3)

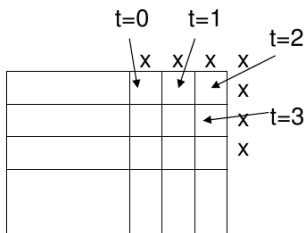
Sinnvolle Aktionen zur Wandverfolgung:

- ▶ $w_2^t \wedge \neg w_4^t \rightarrow \text{east}$
- ▶ $w_4^t \wedge \neg w_6^t \rightarrow \text{south}$
- ▶ $w_6^t \wedge \neg w_8^t \rightarrow \text{west}$
- ▶ $w_8^t \wedge \neg w_2^t \rightarrow \text{north}$
- ▶ $w_1^t \wedge \neg w_2^t \rightarrow \text{north}$
- ▶ $w_3^t \wedge \neg w_4^t \rightarrow \text{east}$
- ▶ $w_5^t \wedge \neg w_6^t \rightarrow \text{south}$
- ▶ $w_7^t \wedge \neg w_8^t \rightarrow \text{west}$
- ▶ 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



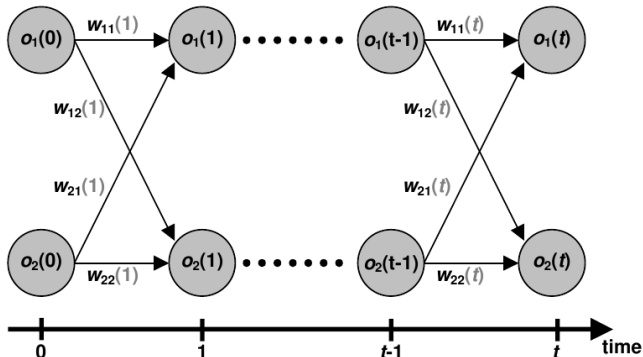
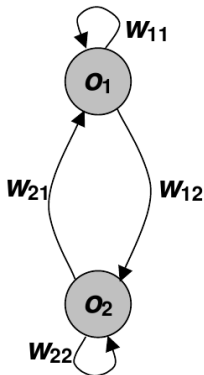
Anmerkung: Situation $t=2'$ z.B. bei Sensorstörung (alle Sensoren $s_i=0$)

Rekurrente Neuronale Netze

- ▶ rückgekoppelte neuronale Netzen
- ▶ Verbindungen von oberen zu unteren Schichten erlaubt
- ▶ vgl. MLP: nur Verbindungen von „unten“ nach „oben“
- ▶ Rückkopplung \Rightarrow Information kann gespeichert werden
- ▶ Training: modifizierte Backpropagation
- ▶ Anwendung: komplexe Differentialgleichungssysteme

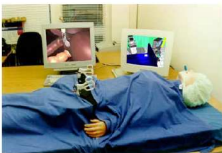
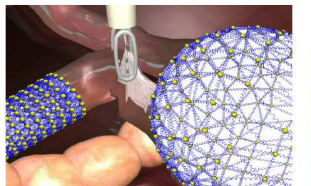
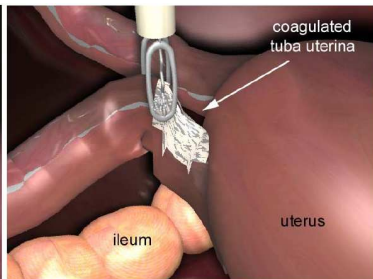
Rekurrente Netze: Lernverfahren

Idee der Lernverfahren: Entfalten des Netzes über die Zeit



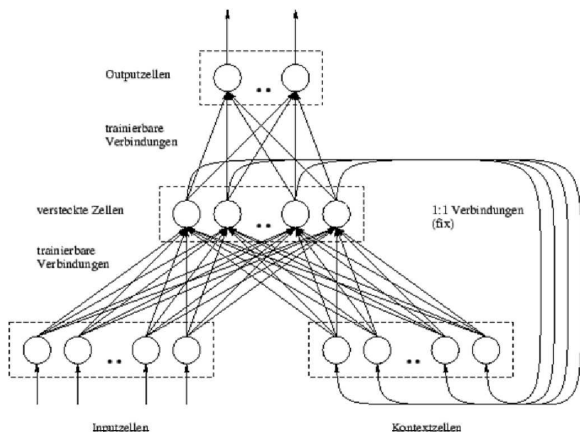
SUSILAP-G

Surgical Simulator for LAParoscopy in Gynaecology [Radetzky et al., 1999]



Einfache RNNs: Elman-Netze

- ▶ Einführung einer Kontextschicht
- ▶ ermöglicht Speichern von Informationen

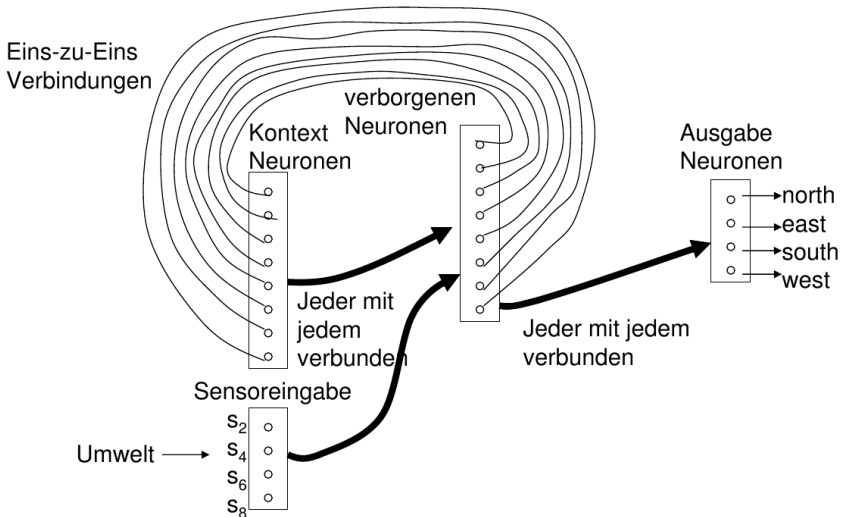


Beispiel: Elman-Netz für Roboter

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

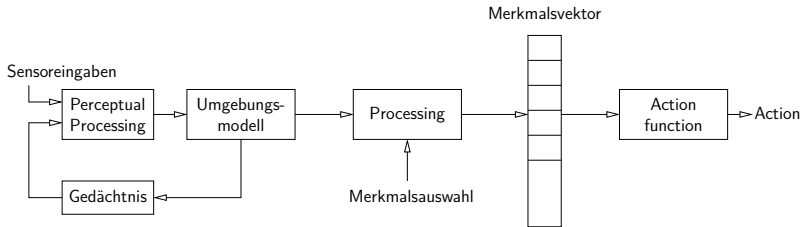
Beispiel: Elman-Netz für Roboter



Umgebungsmodelle

- ▶ bisher: nur Informationen über 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

Umgebungsmodelle: Beispiel für Gitterwelt



Umgebungsmodelle: Beispiel für Gitterwelt

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:

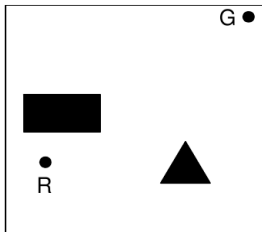
- ▶ durch Zielfeld erzeugt: $p_a(x^{(\rho)}) = k_1 \cdot d(x^{(\rho)})^2$
- ▶ k_1 : konstanter Faktor, d : Abstand zum Zielfeld

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

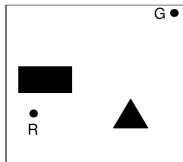


Roboter (R)

Ziel (G)

Hindernisse (◆)

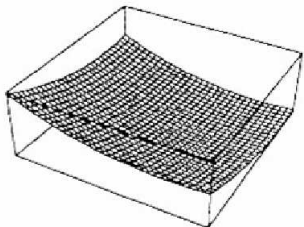
Potentialfelder: Potentialfeldkomponenten



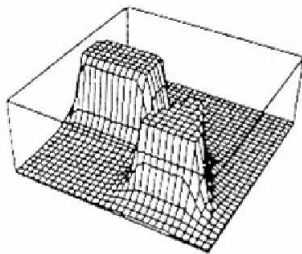
Roboter (R)

Ziel (G)

Hindernisse (◆)

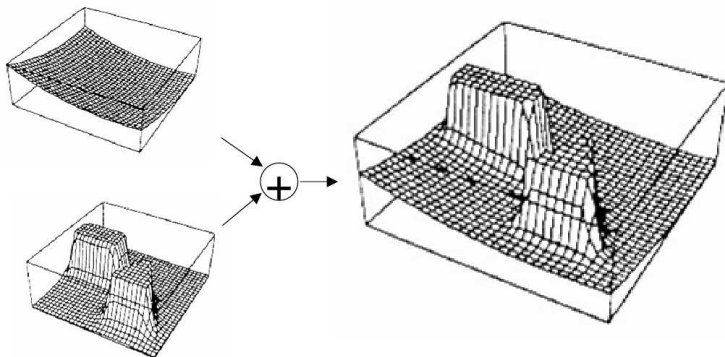


Ziel

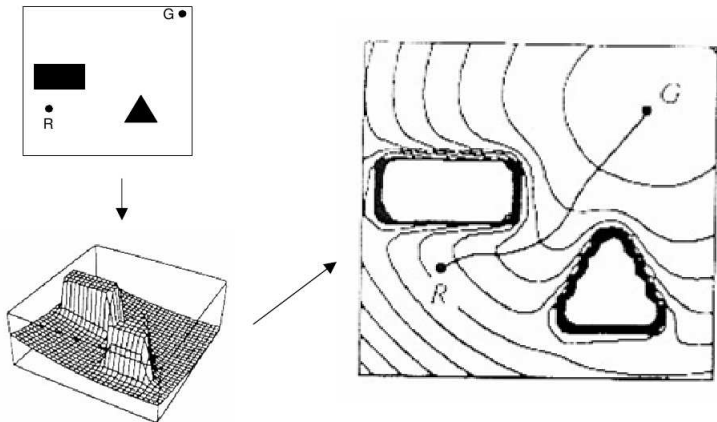


Hindernisse

Potentialfelder: Gesamtes Potentialfeld

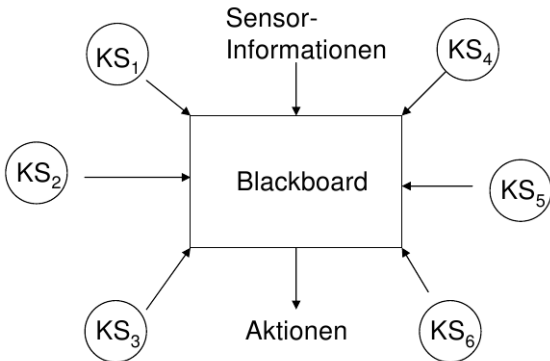


Äquipotentiallinien (für Gradientenverfahren)



Blackboard-Systeme

- ▶ Blackboard: Spezielle Datenstruktur
- ▶ Knowledge Source (KS): Programm zum Lesen und Schreiben des Blackboards



Blackboard-Systeme: Knowledge Source

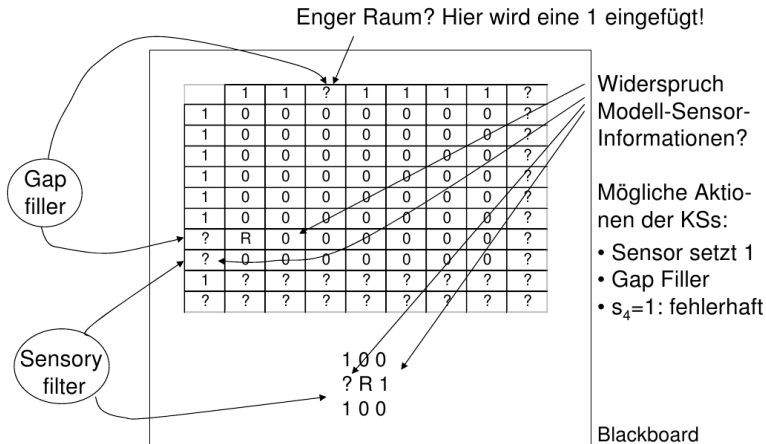
Bestandteile:

1. Bedingungsteil (berechnet Wert eines Merkmals)
2. 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 Knowledge Sourcen:
 - ▶ 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. Zustandsagenten
- 2. Problemlösende Agenten**
3. Uninformierte Suche

Problemlösende Agenten

eingeschränkte Form eines generellen Agenten:

Algorithmus 1 SIMPLE-PROBLEM-SOLVING-AGENT

Eingabe: Wahrnehmung *percept*

Ausgabe: eine Aktion *action*

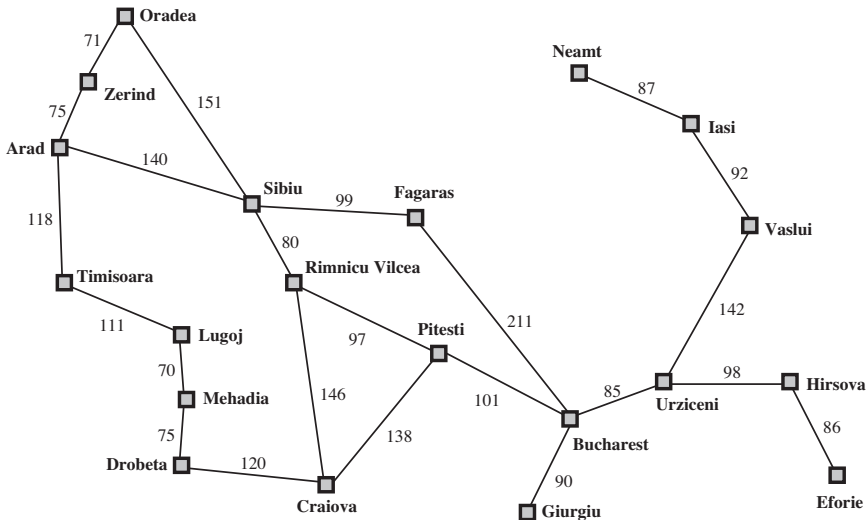
```
1: static seq: Aktionssequenz (anfangs leer)
2: static state: momentane Beschreibung der Welt
3: static goal: Ziel (anfangs null)
4: static problem: Problembeschreibung
5: state ← UPDATE-STATE(state, percept)
6: if seq is empty {
7:   goal ← FORMULATE-GOAL(state)
8:   problem ← FORMULATE-PROBLEM(state,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

- ▶ entspricht Problemlösen "offline"
- ▶ Lösung mit „geschlossenen Augen“
- ▶ "online": Handeln ohne vollständiges Wissen

Beispiel: Rumänien

- ▶ Urlaub in Rumänien, momentan in Arad
- ▶ Rückflug: morgen von Bukarest
- ▶ Formulieren des Ziel: in Bukarest sein
- ▶ Formulieren des Problems: Fahren
 - ▶ Zustände: verschiedene Städte
 - ▶ Aktionen: Fahrten von einer zu anderen Stadt
- ▶ Lösung finden: Sequenz von Städten, z.B. Arad, Sibiu, Fagaras, Bukarest

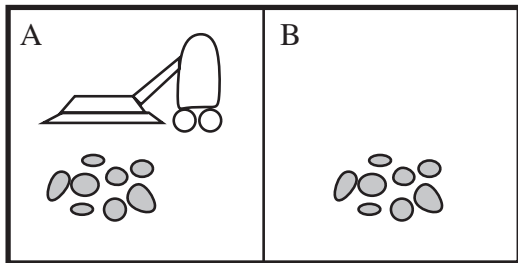
Beispiel: Rumänien



Arten von Problemen

- ▶ deterministisch (vollständig beobachtbar)
 - ▶ Agent weiß genau in welchem Zustand er sein wird
 - ▶ Lösung ist eine Sequenz
- ▶ nicht beobachtbar \Rightarrow **konformantes Problem**
 - ▶ Agent hat u.U. keine Ahnung wo er ist
 - ▶ Lösung (falls existent) ist eine Sequenz
- ▶ nichtdeterministisch (teilw. beobachtbar) \Rightarrow **Zufallsproblem**
 - ▶ Wahrnehmungen: neue Infos über momentan Zustand
 - ▶ Lösung: ungewisser Plan oder Strategie
 - ▶ oftmals verschränkte Suche/Ausführung
- ▶ unbekannter Zustandsraum \Rightarrow **Explorationsproblem** (“online”)

Staubsaug-Welt



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

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

Ein Staubsaug-Agent

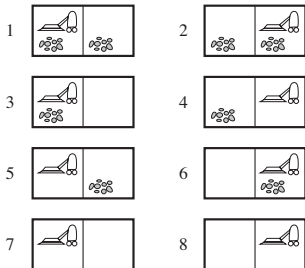
Sequenz von Wahrnehmungen	Aktion
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮

Algorithmus 2 REFLECTIVE-VACUUM-AGENT

```
1: if staus = Dirty {  
2:   return Suck  
3: } else {  
4:   if location = A {  
5:     return Right  
6:   } else {  
7:     return Left  
8:   }  
9: }
```

Beispiel: Staubsaug-Welt

- ▶ **deterministisch**,
Start in #5, Lösung: [*Right*, *Suck*]
- ▶ **konformant**
 - ▶ Start in {1, 2, 3, 4, 5, 6, 7, 8}
 - ▶ z.B. *Right* nach {2, 4, 6, 8}
 - ▶ Lösung: [*Right*, *Suck*, *Left*, *Suck*]
- ▶ **Zufall**
 - ▶ Start in #5
 - ▶ Murphys Gesetz: *Suck* kann einen sauberen Teppich beschmutzen!
 - ▶ lokal Abtastung: Schmutz, nur Ort
 - ▶ Lösung: [*Right*, if dirt then *Suck*]



Ansatz für deterministische Probleme

Problem: definiert durch 4 Begriffe

1. **Anfangszustand**, z.B. *Arad*

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

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

3. **Zieltest**

- ▶ explizit, z.B. $x = \textit{Bukarest}$
- ▶ implizit, z.B. $\textit{NoDirt}(x)$

4. **Wegkosten** (zusätzlich)

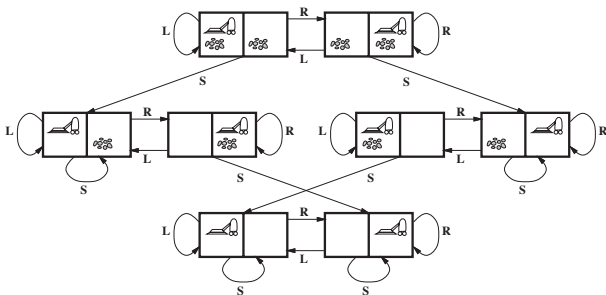
- ▶ Summe der Abstände, Anzahl ausgeführter Aktionen, etc.
- ▶ $c(x, a, y)$ seien Schrittkosten mit ≥ 0

Lösung: Sequenz von Aktionen vom Anfangs- zum Endzustand

Wahl des Zustandsraums

- ▶ reale Welt ist sehr komplex
- ⇒ *Abstraktion* des Zustandsraums für Problemlösen
- ▶ (abstrakter) Zustand = Menge realer Zustände
 - ▶ (abstrakte) Aktion = komplexe Kombination realer Aktionen
z.B. „*Arad* → *Zerind*“ ist komplexe Menge möglicher Strecken,
Umleitungen, Raststätten, usw.
für Realisierbarkeit: jeder reale Zustand „*in Arad*“ führt zu realem
Zustand „*in Zerind*“
 - ▶ (abstrakte) Lösung = Menge aller realen Pfade, die realen
Lösungen entsprechen
 - ▶ abstrakte Aktion sollte „leichter“ sein als Realität

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*)

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

Baumsuchalgorithmen

grundlegende Idee:

- ▶ offline, simulierte Durforstung des Zustandsraums
- ▶ Erzeugung von Nachfolgern bereits erkundeter Zustände (sog. expandierte Zustände)

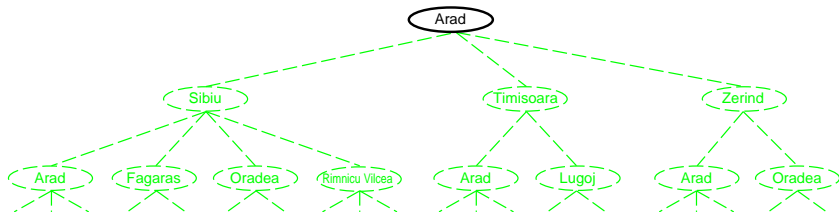
Algorithmus 3 TREE-SEARCH

Eingabe: Problembeschreibung *problem*, Vorgehensweise *strategy*

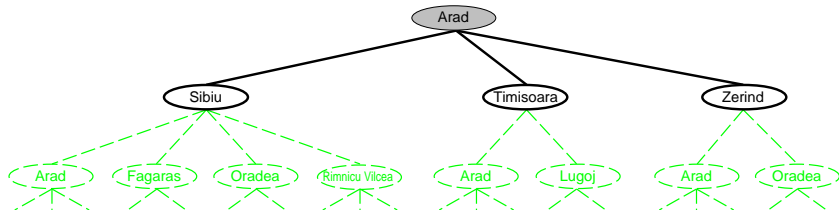
Ausgabe: Lösung oder Fehler

```
1: initialize search tree using initial state of problem
2: while true {
3:   if there are no candidates for expansion {
4:     return failure
5:   }
6:   choose leaf node for expansion according to strategy
7:   if node contains goal state {
8:     return corresponding solution
9:   } else {
10:    expand node and add resulting nodes to search tree
11:  }
12: }
```

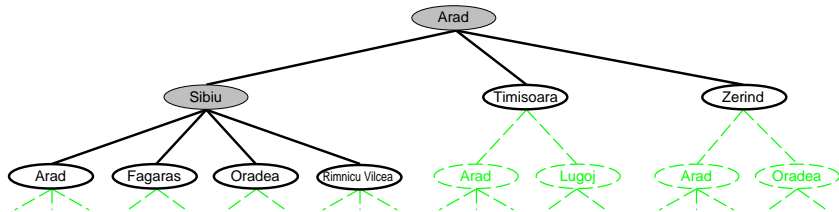
Beispiel: Baumsuche



Beispiel: Baumsuche

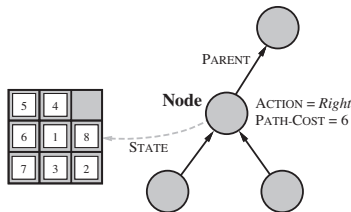


Beispiel: Baumsuche



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 zugehörigen Zustände

Implementierung: Generelle Baumsuche

Algorithmus 4 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

Algorithmus 5 EXPAND

Eingabe: Knoten *node*, Problembeschreibung *problem*

Ausgabe: eine Menge von Knoten

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

Suchstrategien

- ▶ Strategie = 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 ∞)

Übersicht

1. Zustandsagenten

2. Problemlösende Agenten

3. Uninformierte Suche

Breitensuche

Tiefensuche

Beschränkte Tiefensuche

Iterative Tiefensuche

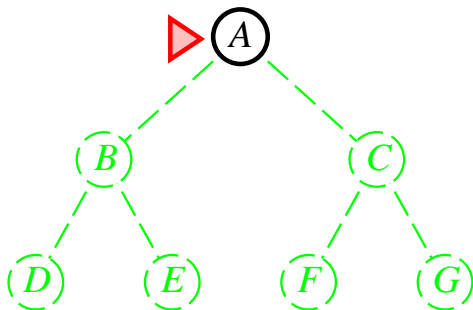
Uninformierte Suchstrategien

Uninformierte Suchstrategien nutzen nur verfügbare Informationen der Problemdefinition.

- ▶ Breitensuche
- ▶ uniforme Kostensuche
- ▶ Tiefensuche
- ▶ beschränkte Tiefensuche
- ▶ iterative Tiefensuche

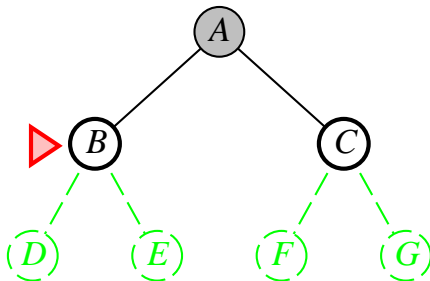
Breitensuche (engl. *breadth-first search*)

- ▶ Expandiere „seichtesten“, nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



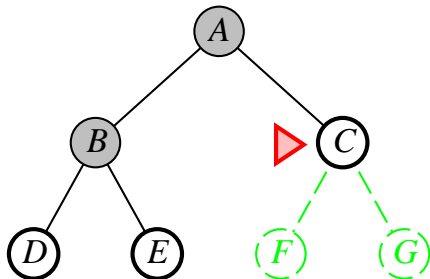
Breitensuche (engl. *breadth-first search*)

- ▶ Expandiere „seichtesten“, nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



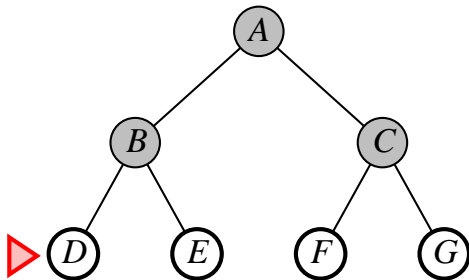
Breitensuche (engl. *breadth-first search*)

- ▶ Expandiere „seichtesten“, nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche (engl. *breadth-first search*)

- ▶ Expandiere „seichtesten“, nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche: 8-Puzzle

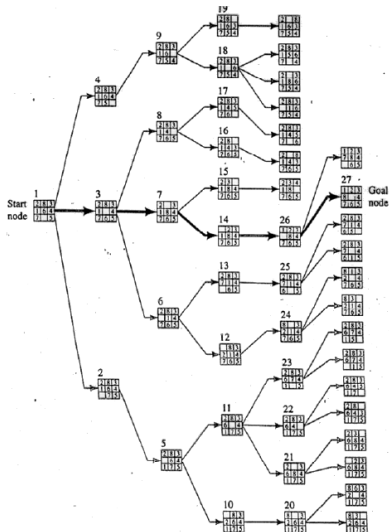
Start

2	8	3
1	6	4
7		5



Ziel

1	2	3
8		4
7	6	5



Breitensuche: Eigenschaften

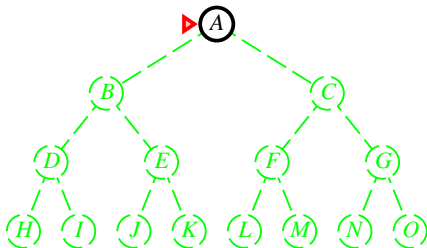
- ▶ vollständig: falls b endlich
- ▶ Zeit: $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, also exponentiell in d
- ▶ Speicher: $O(b^{d+1})$ (behält jeden Knoten im Speicher)
- ▶ optimal: falls Kosten = 1 pro Schritt, generell nicht
- ▶ größtes Problem: Speicher
- ▶ leichte Erzeugung von Knoten mit 100 MB/s, also 24 h \mapsto 8.5 TB

Uniforme Kostensuche

- ▶ Expandiere nicht-expandierten Knoten mit geringsten Kosten
- ▶ Implementierung: *fringe* = Schlange absteigend sortiert nach Wegkosten
- ▶ äquivalent zur Breitensuche falls Schrittkosten alle gleich
- ▶ vollständig: falls Schrittkosten $\geq \epsilon$
- ▶ Zeit: # Knoten mit $g \leq$ Kosten der optimalen Lösung $O(b^{\lceil C^*/\epsilon \rceil})$
wobei C^* Kosten der optimalen Lösung
- ▶ Speicher: # Knoten mit $g \leq$ Kosten der optimalen Lösung
 $O(b^{\lceil C^*/\epsilon \rceil})$
- ▶ optimal: ja, denn Knoten expandieren in aufsteigender Reihenfolge von $g(n)$

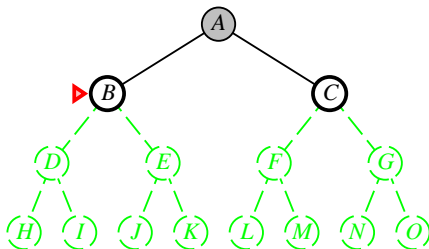
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



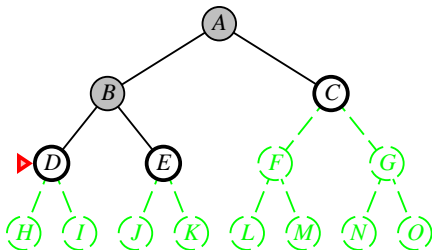
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



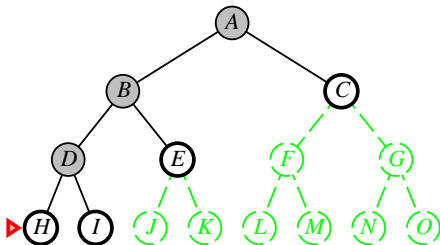
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



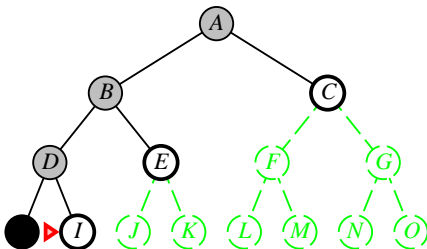
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



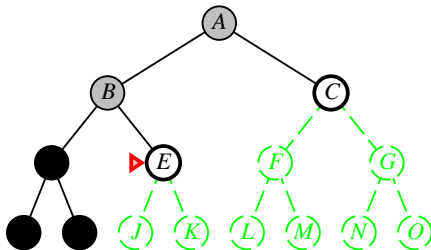
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



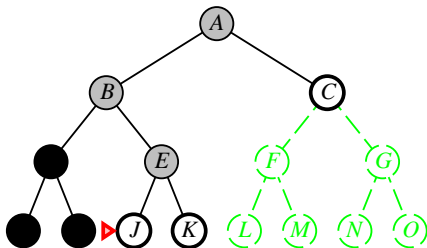
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



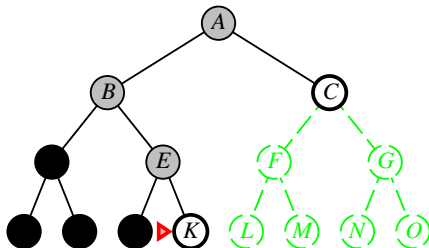
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



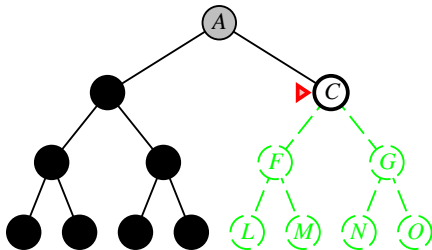
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



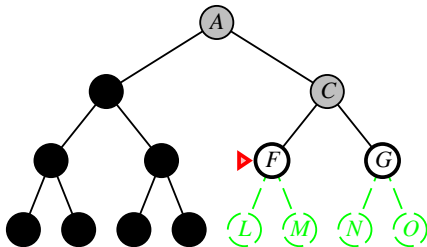
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



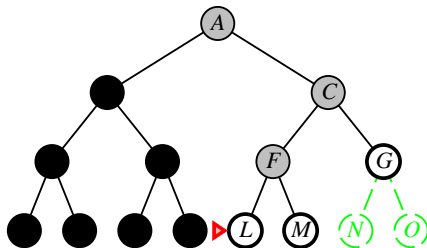
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



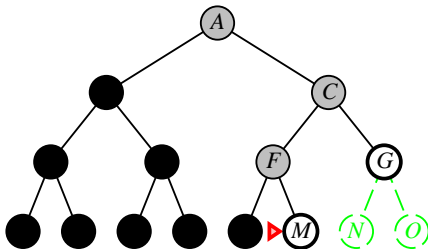
Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

- ▶ Expandiere tiefsten nicht-expandierten Knoten!
- ▶ Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche: Eigenschaften

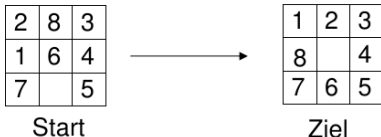
- ▶ vollständig:
 - ▶ nein, versagt für unendlich-tiefe Räume (oder Räume mit Schleifen)
 - ▶ ja, für endliche Räume bei Vermeidung sich wiederholender Zustände im Pfad
- ▶ Zeit:
 - ▶ $O(b^m)$, schrecklich falls $m \gg d$
 - ▶ falls viele Lösungen, dann u.U. viel schneller als Breitensuche
- ▶ Speicher: $O(bm)$, also linearer Speicher!
- ▶ optimal: nein

Beschränkte Tiefensuche

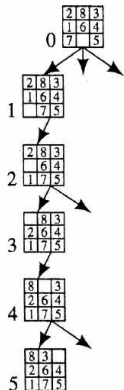
- ▶ Tiefensuche mit Tiefenbegrenzung l
- ⇒ Knoten in Tiefe l haben keine Nachfolger

Beispiel: 8-Puzzle

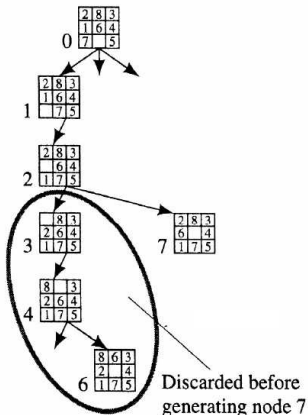
- ▶ Operationsreihenfolge: links, oben, rechts, unten
- ▶ Tiefenbegrenzung: 5



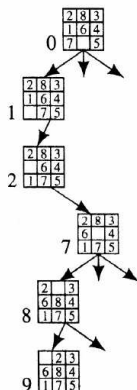
Beschränkte Tiefensuche: 8-Puzzle



(a)

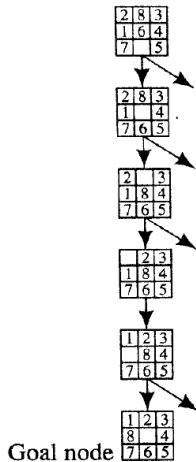
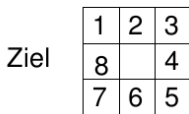
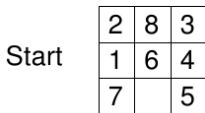


(b)



(c)

Beschränkte Tiefensuche: 8-Puzzle



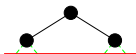
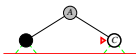
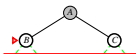
Iterative Tiefensuche

Limit = 0



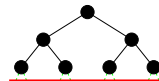
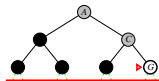
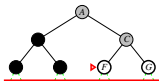
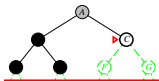
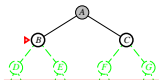
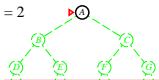
Iterative Tiefensuche

Limit = 1



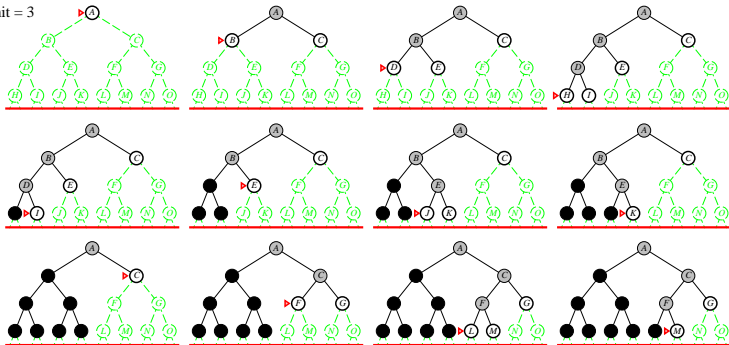
Iterative Tiefensuche

Limit = 2



Iterative Tiefensuche

Limit = 3



Iterative Tiefensuche: Eigenschaften

- ▶ vollständig: ja
- ▶ Zeit: $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- ▶ Speicher: $O(b \cdot d)$
- ▶ optimal:
 - ▶ ja, falls Schrittkosten = 1
 - ▶ kann um uniforme Kostenbäume erweitert werden
- ▶ numerischer Vergleich für $b = 10$ und $d = 5$ (Lösung im am weitesten rechten Blatt):

$$N(\text{IDS}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$


$$N(\text{BFS}) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$$

- ▶ IDS besser, weil andere Knoten bei Tiefe $d = 5$ nicht expandiert
- ▶ BFS kann modifiziert werden, um auf Ziel zu testen wenn Knoten generiert wird

Zusammenfassung der Algorithmen

Kriterium	Breiten- suche	uniforme Kostensuche	Tiefen- suche	beschränkte Tiefensuche	iterative Tiefensuche
vollständig?	ja*	ja*	nein	ja, falls $l \geq d$	ja
Zeit	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Speicher	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
optimal?	ja*	ja	nein	nein	ja*

Literatur

-  Radetzky, A., Bartsch, W., Grospietsch, G., and Pretschner, D. P. (1999). SUSILAP-G: Ein Operationssimulator zum Training minimal-invasiver Eingriffe in der Gynäkologie. *Zentralblatt für Gynäkologie*, 121(2).