Problemlösende Agenten

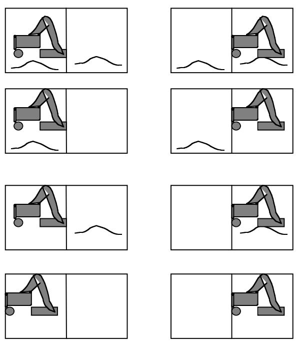
- Formuliere Ziel und Problem
- Wahrnehmung des Ausgangszustandes
- Suche einer geeigneten Handlungsfolge, um zum Ziel zu kommen
  Wahrnehmung des Zustands
  Ziel erreicht?
- Bewertung der Handlungsfolge

Suche

Suchproblem:
Anfangszustand (Anfangsknoten)
Zielzustand (Zielknoten)
Zustandsübergänge (Operatoren)
Problemraum:
Knoten und Operatoren
Operator expandiert einen Knoten n, d.h. erzeugt Nachfolge- knoten zum Knoten n.
Kriterien:
kürzesten, billigsten, schnellsten, ... Pfad finden
wann ist eine Lösung gefunden? (Abbruchkriterium)
Strategien:
Tiefen-/Breitensuche, heuristische Suche, A*

Freiburger Staubsaugerwelt

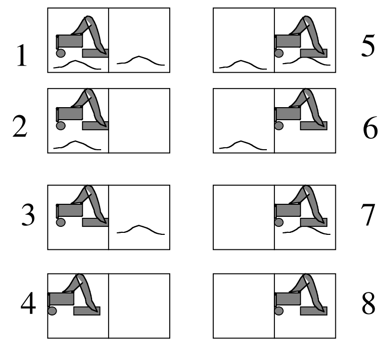
Knoten:
2 Zimmer, (¬) Schmutz
Operatoren:
ins linke Zimmer,
ins rechte Zimmer,
saugen
Ziel:
¬ Schmutz in den Zimmern



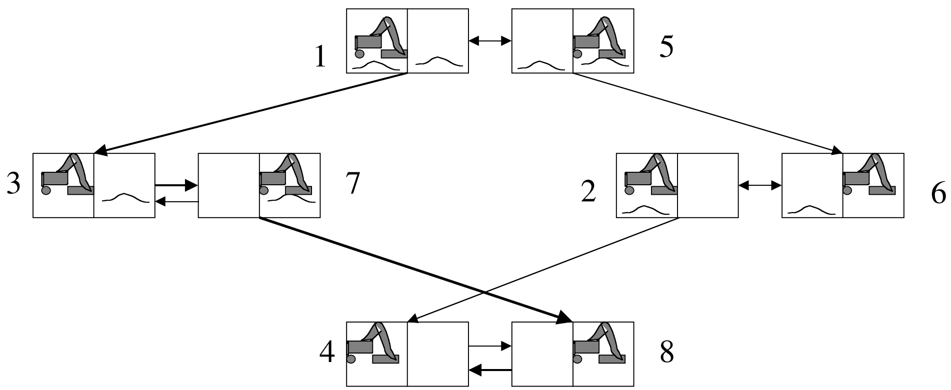
Köhler/Nebel

Einzustandsproblem

staubsaugen:
operator (1, 3). operator (5, 6).
operator (2, 4). operator (7, 8).
rechts:
operator (1, 5). operator (2, 6).
operator (3, 7). operator (4, 8).
links:
operator (5, 1). operator (6, 2).
operator (7, 3). operator (8, 4).



Problemraum



Einfache rekursive Suche

```
start(1).
operator (1, 3). operator (3, 7). operator (7, 8).
operator (2, 4). operator (4, 8).
operator (5, 6). operator (6, 2). operator (1, 5).
operator (2, 6). operator (5, 1). operator (7, 3).  operator (8, 4).

suche (Ziel):-
    start (Start),
    pfad (Start, Ziel).

pfad (Zustand, Zustand).                %Zustand gleich, also Erfolg!
pfad (Start, Ziel):-
    operator (Start, Zwischen), write (Zwischen), nl,
    pfad (Zwischen, Ziel).
```


Ergebnis

Aufruf:

suche (8).

Ergebnis:

3
7
8

Achtung:

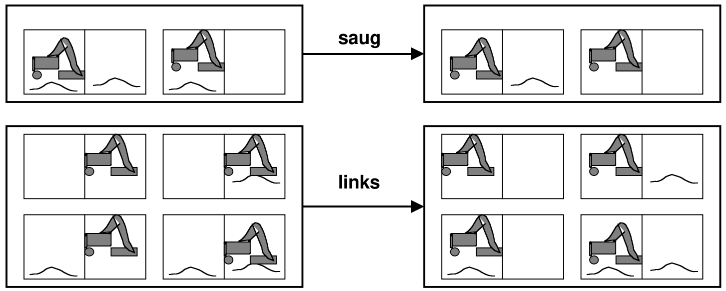
Reihenfolge der Operator-Fakten bestimmt den Pfad!

suche (4).

3
7
8
4

Wir haben den Pfad nur ausgedruckt, nicht repräsentiert!

Mehrzustandsproblem



Operatoren verwenden mehrere Zustände

operator(saug, a, schmutz, DreckB, a, sauber, DreckB).
operator(links, b, DreckA, DreckB, a, DreckA, DreckB).

in Prolog ganz einfach!

Zustände und Operatoren in Prolog

```
%Zustand: Ort, Dreck in Raum A, Dreck in Raum B
start(a, schmutz, schmutz).

%operator(Name,+Ort,+DreckA, +DreckB, -NeuOrt, -NeuA, -NeuB)
operator(saug, a, schmutz, DreckB, a, sauber, DreckB).
operator(saug, b, DreckA, schmutz, b, DreckA, sauber).
operator(rechts, a, DreckA, DreckB, b, DreckA, DreckB).
operator(links, b, DreckA, DreckB, a, DreckA, DreckB).
```


Einfache rekursive Suche

```
suche (ZielO, ZielA, ZielB):-
    start (StartO, StartA, StartB),
    pfad (StartO, StartA, StartB, ZielO, ZielA, ZielB).

pfad (Ort, A, B, Ort, A, B).  %Start = Ziel, also Erfolg!
pfad (StartO, StartA, StartB, ZielO, ZielA, ZielB):-
    operator (Name, StartO, StartA, StartB, NeuO, NeuA, NeuB),
    write (Name),
    pfad (NeuO, NeuA, NeuB, ZielO, ZielA, ZielB).
```


Suchen

Aufruf:

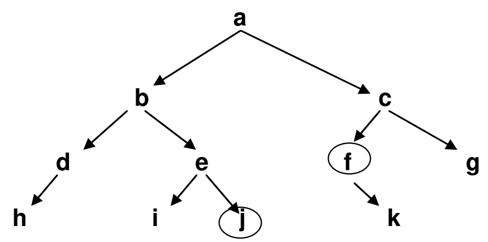
suche (X, sauber,sauber).	**suche (a, sauber, sauber)**

Ergebnis:

saug	**saug**
rechts	**rechts**
saug	**saug**
	links

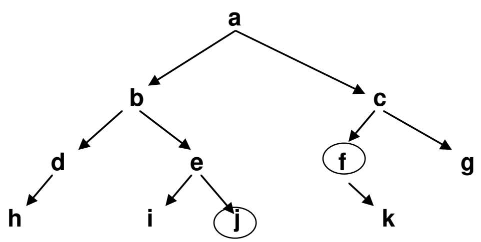
- Reihenfolge der Operator-Fakten wichtig!
- Anwendbarkeit des Operators wird durch die +Argumente bestimmt.
- Pfad wird per Rückzug gefunden.
- Pfad ist nicht repräsentiert!

Repräsentation von Graphen - Fakten



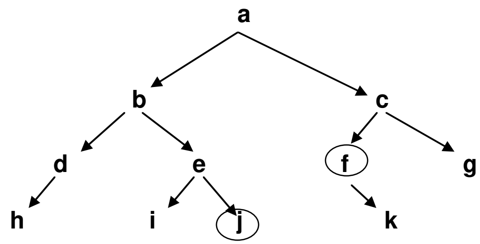

```
nachf(a,b). nachf(a,c).          ziel(j). ziel(f).
nachf(b,d). nachf(b,e).
nachf(e,i). nachf(e,j).
nachf(d,h).
nachf(c,f). nachf(c,g).
nachf(f,k).
```

Graphen als Knoten mit Liste der Nachfolger



nachf(a, [b,c]). **ziel(j). ziel(f).**
nachf(b, [d,e]).
nachf(e, [j,j]).
nachf(d, [h]).
nachf(c, [f,g]).
nachf(f, [k]).

Von Fakten zu Listen -- Rekursion




```
sammeln(Knoten, Pfad):- ziel(Knoten), write(Pfad).
sammeln(Knoten, Pfad):-
    nachf(Knoten, Nachfolger),
    sammeln(Nachfolger, [Knoten|Pfad]).
```

sammeln(a, [ ]) ergibt: **[e, b, a]**

Von Fakten zu Listen 2

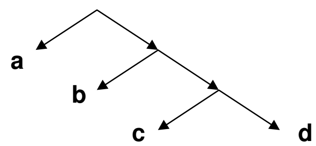
```
findall(Element, Problem, Ergebnisse):-
    call (Problem),                    % finde eine Lösung
    assertz (stack(Element)),          % trage sie ein
    fail                               % suche nach weiteren Lösungen
  ; assertz (stack(ende)),             % markiere das Ende der Lösungen
    sammle (Ergebnisse).               % sammle die Einträge
```

Aufruf zum Beispiel:

```
?-  findall (X, nachf (a, X), Nachfolger).
    Nachfolger = [b, c]
```


Behandlung von Listen

[a, b, c, d]




```
member(Element, [Element | Rest]).      %Abbruchbedingung zuerst!

member (Element,  [Erst|Rest]):-
    member (Element,Rest).              % rekursiver Abstieg über die Liste
```

eingebautes Prädikat der Basisbibliothek

<u>?- [library(basics)].</u>
member(Element, Liste).

Suche mit Listen

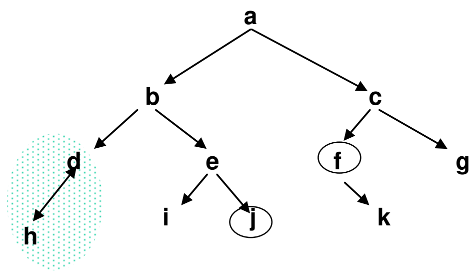
OFFEN: Liste der noch nicht expandierten Knoten

GESCHLOSSEN: Liste der expandierten Knoten

Reihenfolge der Knoten in OFFEN bestimmt Suchstrategie:

- Nachfolgeknoten vorn in die Liste --> Tiefensuche
- Nachfolger hinten in die Liste --> Breitensuche
- bester Nachfolger vorn in die Liste --> Bestensuche

Tiefensuche mit Zyklenerkennung

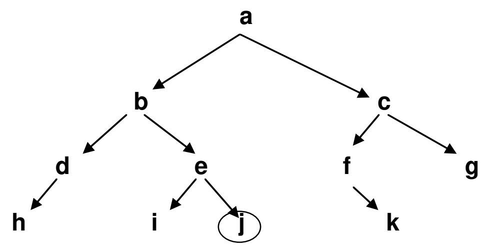



```
% tiefensuche(P, K, L)
% P - bisherigen Pfad merken!
% K - aktueller Knoten
% L - Gesamtpfad konstruieren!

tiefensuche (Pfad, K, [K|Pfad]) :- ziel(K).
tiefensuche (Pfad, K, L):-
     nachf (K, K1),                        % aktuellen Knoten expandieren
     \+ member(K1, Pfad),                  %Nachfolger nicht im bisherigen Pfad!
     tiefensuche ([K|Pfad], K1, L).        % bisherigen Pfad aktualisieren

?- tiefensuche ( [] , a, L).
L= [j, e, b, a] ; L= [f, c, a] ; no
```


Breitensuche

```
% breitensuche ( [ [aktuellOffen] | Offen], Loesung)
% K  aktueller Knoten
% Nneu alle Nachfolger eines Knotens

breitensuche ( [[K| P] | _], [K|P]) :- ziel(K).    % Erfolg
breitensuche ( [[K| P] |Offen], Loesung ):-
  findall ( [K1, K| P],
     ( nachf(K, K1), \+ member(K1, [K|P])  ) , % nichtzyklische Expansion
     Nneu),
  append (Offen, Nneu, NeuOffen), !,           % Nachfolger hinten anhängen!
  breitensuche (NeuOffen, Loesung)             % rekursiver Aufruf NeuOffen
  ;  breitensuche (Offen, Loesung).            % rekursiver Aufruf Offen
                                               % (keine Nachfolger)
```


Ergebnis



breitensuche([[a]], Geschl).

Entwicklung von Offen

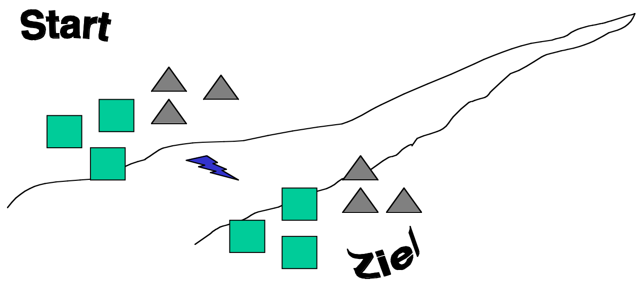
Offen:	NeuOffen:
[]	[[b, a], [c, a]]
[[c, a]]	[[c, a], [d, b, a], [e, b, a]]
[[d, b, a], [e, b, a]]	[[d, b, a], [e, b, a], [f, c, a], [g, c, a]]

Eigenschaften der Breitensuche

- Vollständig: findet eine Lösung, wenn es sie gibt
- Optimal: findet den kürzesten Pfad
- Aufwendig: Anzahl der Knoten, die expandiert werden
  b: maximaler Verzweigungsfaktor
  d: Tiefe eines Lösungspfades

  $O\left(b^d\right)$ dies ist auch der Platzbedarf

Suche mit speziellen Operatoren



3 Missionare und 3 Kannibalen sind an einem Flussufer. Alle wollen ans andere Ufer. Sie haben ein Boot, das bis zu 2 Personen befördern kann. Wenn an einem Ufer mehr Kannibalen als Missionare sind, werden die Missionare verspeist. Wie kommen alle Personen heil ans andere Ufer?

Repräsentation

- Nur ein Ufer darstellen -- das andere ist komplementär!
- Argumentstellen ausnutzen:
  Anzahl Missionare,
  Anzahl Kannibalen,
  An-Abwesenheit des Bootes
- Operator:
  Anwendbarkeitsbedingung (legal move)
  Übersetzen
  Bedingung an beiden Ufern prüfen

... in Prolog

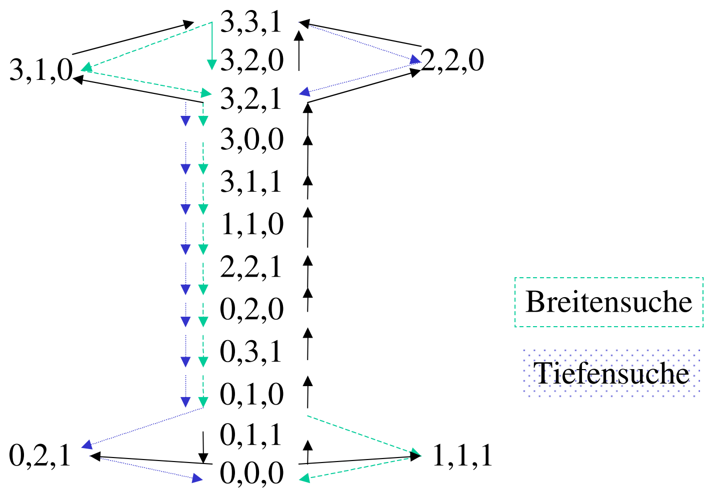
Anfangszustand z(3,3,1)

Zielzustand z(0,0,0)

10 spezielle Operatoren:

```
% nachf(+aktuellerKnoten, -Nachfolger)
nachf(z(M,K,1), z(Mneu, K,0)):-        %ein Missionar setzt über
    M > 0,                             %Anwendbarkeit
    Mneu is M - 1,                     %Übersetzen
    (K ≤ Mneu; Mneu =0),               %Bedingung am Anfangsufer
    F1 is 3-K,                         %Ankommen
    F2 is 3-Mneu,
    (F1 ≤ F2; F2 = 0).                 %Bedingungen am Zielufer
```

Problemraum



allgemeines Suchprogramm

```
suche(Offen, Geschl, Ziel):-
     member (Ziel, Offen),            % positiver Abbruch
     write(Geschl).

suche(Offen, Geschl, Ziel):-
     best(Offen, Best, RestOffen),   %Sortierung von Offen bestimmt Strategie
      findall(Nachf,
               nachf(Best, Nachf),       %alle Nachfolger des besten Knotens
               AlleNachf),
     verteile(AlleNachf,RestOffen, [Best| Geschl], NeuOffen),
                                               %keine Doppelten in NeuOffen!
     suche(NeuOffen, [Best|Geschl], Ziel).
```


Verteilung der Nachfolger

```
%verteile(+AlleNachf,+RestOffen,+Geschlossen,-NeuOffen)

verteile([ ], Offen, Geschl, Offen).                %keine Nachfolger -- fertig!
                                                            %Offen wird das Ergebnis
verteile([AK|Rest], Offen, Geschl, NeuOffen):-
     member(AK, Offen), !,                          %Nachfolger in Offen ---
     verteile(Rest, Offen, Geschl, NeuOffen).       %restliche Nachfolger
verteile([AK|Rest], Offen, Geschl, NeuOffen):-
     member(AK, Geschl), !,                         %Nachfolger in Geschl ---
     verteile(Rest, Offen, Geschl, NeuOffen).       %restliche Nachfolger
verteile([AK|Rest], Offen, Geschl, NeuOffen):-
     verteile(Rest, [AK|Offen], Geschl, NeuOffen).  % Nachfolger in Offen
                                                        %einfügen
```


Plazierung der Nachfolger in Offen

- Tiefensuche, Breitensuche, Heuristik über das Prädikat best explizieren!
  Welches Element aus der Offen-Liste wird genommen?

```
%best(+Offen, -Best, -RestOffen)
 best([Erst|Rest], Erst, Rest).          %Triviale Lösung, die die Sortierung %von
                                                              Offen übernimmt,
                                               %ergibt Breiten- oder Tiefensuche.
```