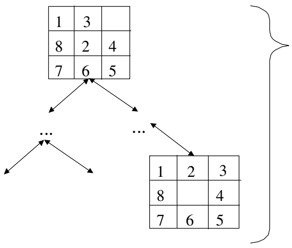
Acht-Rätsel - Zustandsräume



Zustandsraum : gerichteter Graph
- Problemsituationen : Knoten
- Zustandsübergänge : Kanten
- Lösung : Weg vom Start zum Ziel

Repräsentation des Zustandsraums:
- Menge von Fakten n(K1,K2,Kosten)
- Regel zur Berechnung des Nachfolgerknotens

Repräsentation der Problemsituation:
- Wahl einer kompakten und bzgl. der erforderlichen Operationen effizienten Datenstruktur

Repräsentation des Acht-Rätsels in Prolog

Der **Zustandsraum** wird durch Prädikate repräsentiert:

```
n(Knoten, Knoten1, Kosten)     %wahr, wenn es eine gerichtete Kante mit den
                               %Kosten zwischen Knoten und Knoten1 gibt.
                               %hier: Kosten = 1
```

Die **Problemsituation** wird durch eine Liste von Positionen der Scheiben dargestellt. Das erste Element entspricht der Position der leeren Scheibe, das zweite der Pos. der Scheibe 1, ...

3	1	2	3
2	8		4
1	7	6	5
	1	2	3

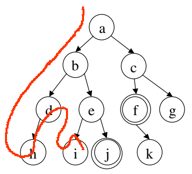
Bsp. [2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]

Zielsituation

Auf die explizite Angabe der einzelnen Prädikate n(...) kann verzichtet werden:

```
n([Leer|L], [B|L1], 1) :- tausche(Leer, B, L, L1).
```


Tiefensuche



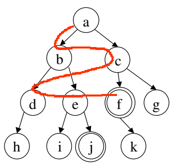

```
?- loese(a, Loes).
Loes = [j, e, b, a];
Loes = [f, c, a];
no
```

```
loese(Knoten, Loesung) :- tiefensuche([], Knoten, Loesung).

tiefensuche(Pfad, Knoten, [Knoten|Pfad]) :- ziel(Knoten).
tiefensuche(Pfad, Knoten, Loes) :-
    n(Knoten, Knoten1, 1),                  %Zustandsraum.
    not member(Knoten1, Pfad),              %Zyklus verhindern.
    tiefensuche([Knoten|Pfad], Knoten1, Loes).
```

Verbesserungen zur Tiefensuche

```
loese(Start, Loesung) :-
        groesster(Max_Tiefe),
        tiefensuche2([Start], Loesung, Max_Tiefe).

tiefensuche2([Knoten|Pfad], [Knoten|Pfad], Max_Tiefe) :- ziel(Knoten).
tiefensuche2([Knoten|Pfad], Loesung, Max_Tiefe) :-
        Max_Tiefe >0,
        n(Knoten, Knoten1, 1),
        not member(Knoten1, Pfad),      %Zyklus verhindern
        Max1 is Max_Tiefe -1,
        tiefensuche2([Knoten1,Knoten|Pfad], Loesung, Max1).

% Bsp. -----------
n(a,b,1). n(b,d,1). n(d,h,1). n(b,e,1). n(e,j,1). n(e,i,1).
n(a,c,1). n(c,f,1). n(f,k,1). n(c,g,1).
ziel(j). ziel(f).
groesster(Max_Tiefe) :- Max_Tiefe = 2.  %liefert nur ein Ergebnis.
```


Breitensuche

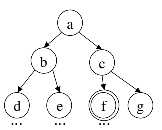


Bsp. Breitensuche mit „umgekehrten Anwärterpfaden"

[[a]]
[[b, a], [c,a]]   % [b,a] --> [[d, b, a], [e, b, a]]
[[c, a], [d, b, a], [e, b, a]]
...
[[f, c, a], [g, c, a]...]

```
loese(Knoten, Loesung) :- breitensuche( [[Knoten]], Loesung).
breitensuche([[Knoten|Pfad] | _], [Knoten | Pfad] ) :- ziel(Knoten).
breitensuche([[Kn|Pfad] | Pfade], Loesung) :- bagof(
        [M,Kn | Pfad], ( n(Kn, M, 1), not member(M, [Kn|Pfad]) ), Neue_Pfade ),
                                        %Neue_Pfade soll Zyklen verhindern
append(Pfade, Neue_Pfade, Pfade1), !,
        breitensuche(Pfade1, Loesung)
        ;breitensuche(Pfade, Loesung).      %wenn Kn keine Nachfolger hat
```


Verbesserung der Breitensuche



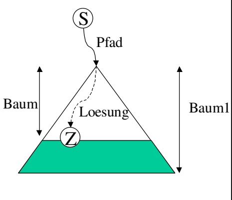
Bsp. Repräsentation der Anwärtermenge
vorher: [[d, b, a], [e, b, a], [f, c, a], [g, c, a]]
nun: b(a, [b(b, [bl(d), bl(e)]], b(c, [bl(g)])])

Schlüsselrelation des Programms breitensuche2:
erweitere(+Pfad, +Baum, Baum1, Geloest, Loesung)

erzeugt zwei Arten von Lösungen:

entweder: Loesung = Pfad erweitert zu einem Ziel

oder Baum1 = Baum, der an seinem unteren Rand um eine Ebene erweitert ist, wobei Baum1 keine „toten" Äste von Baum enthält.

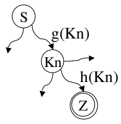


Programm breitensuche2(Baum, Loesung):

```
loese(Start, Loesung) :-  breitensuche2(bl(Start), Loesung).

breitensuche2(Baum, Loesung) :-
        erweitere([], Baum, Baum1, Geloest, Loesung),
        ( Geloest = ja
        ; Geloest = nein, breitensuche2(Baum1, Loesung)
        ).

erweitere(P, bl(Kn), _, ja, [Kn|P]) :-   ziel(Kn).
erweitere(P, bl(Kn), b(Kn, Unter), nein, _) :-
        bagof( bl(M), ( n(Kn, M, 1), not member(M,P)), Unter).
erweitere(P, b(Kn, Unter), b(Kn, Unter1), Geloest, Loes) :-
        erweitere_alle([Kn|P], Unter, [], Unter1, Geloest, Loes).

erweitere_alle( _, [], [B|Be], [B|Be], nein, _).            %mind. ein Baum wachsen
erweitere_alle(P, [B|Be], Be1, Unter1, Geloest, Loes) :-
        erweitere(P, B, B1, Geloest1, Loes),
        ( Geloest1 = ja, Geloest = ja
        ; Geloest1 = nein, !,
          erweitere_alle(P, Be, [B1|Be1], Unter1, Geloest, Loes)
        )
     ;erweitere_alle(P, Be, Be1, Unter1, Geloest, Loes).
```


Heuristische Suche

„Abschätzen" der Kosten eines besten Lösungspfades von S nach Z über kn:
f(Kn) = g(Kn) + h(Kn)   mit
g(Kn) = Summe der Kosten der Kanten von S nach Kn
h(Kn) = heuristische Schätzung abhängig vom Problem

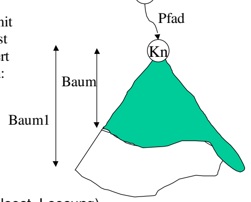


Anwendung auf Acht-Rätsel:
h(Pos,H)

/* Pos ist eine Brettstellung, H = totdist + 3* folge mit
totdist : totale Distanz der acht Scheiben in pos von ihren Heimatfeldern
folge : Summe aller Scheibenbewertungen, wobei eine Scheibe
in der Mitte 1,
nicht in der Mitte 0, wenn ihr richtiger Nachfolger im Uhrzeigersinn folgt
sonst 2 zählt.*/

BestFirstSearch

* bl(Kn, F/G) repräsentiert ein Blatt; G ist g(Kn) und F ist f(Kn)=G+h(Kn)
* b(Kn, F/G, Unter) repräsentiert einen Baum mit nichtleeren Unterbäumen; G ist g(Kn) und F ist der **erneuerte** f-Wert von Kn, das ist der f-Wert des meistversprechendsten Nachfolger von Kn: Unter ist eine Liste der Unterbäume geordnet nach wachsenden f-Werten.

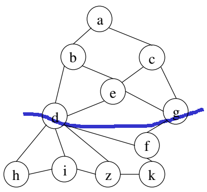


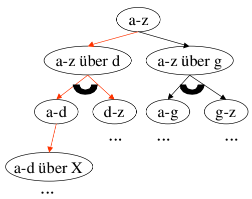
Schlüsselrelation (nun 6 Argumente):
erweitere(+P, +Baum, +Grenze, Baum1, Geloest, Loesung)

Mit Grenze wird der maximale f-Wert übergeben.
Baum 1 ist die Erweiterung von Baum innerhalb der Grenze.
Geloest : ja, nein(=Grenze überschritten), nie(=kann nicht wachsen)

Best-First-Search und das Acht-Rätsel

UND/ODER-Graphen

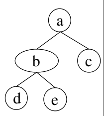




Ein UND/ODER-Graph ist ein gerichteter Graph, in dem Knoten Problemen entsprechen, und gerichtete Kanten Beziehungen zwischen diesen Problemen angeben. Zwischen den von einem Knoten „wegführenden" Kanten kann eine ODER oder eine UND-Relation bestehen.

Suchen auf UND/ODER-Graphen

```
a :- b.      %a ist ein ODER-Knoten mit Nachfolgern b und c
a :- c.
b :- d, e.   %b ist ein UND-Knoten mit Nachfolgern d und e
c. d. e.     %sind Zielknoten
?- a.        %Anfrage, ob das Problem a gelöst werden kann.
```



Zu behebende Mängel:

1. schwer erweiterbar bzgl. Gewichtung
2. keine Ausgabe des Lösungsbaums
3. Gefahr von Endlosschleifen

1. Schritt: Def. einer eigenen Notation:

```
:- op(600, xfx, -->)
:- op(500, xfx, :)

a --> oder: [b, c].
b --> und: [d, e].
ziel(c). ziel(d). ziel(e).
```

%Def. von loese(Knoten, Loesungsbaum):

```
loese(Knoten, Knoten) :- ziel(Knoten).     %Baum ist der Zielknoten selbst
loese(Knoten,Knoten--->Baum) :-
    Knoten---> oder:Knoten2,               %ODER-Knoten, dann ist Baum
    member(Knoten1,Knoten2),               %von der Gestalt
    loese(Knoten1,Baum).                   % Knoten--->Unterbaum
loese(Knoten,Knoten---> und: Baeume) :-
    Knoten---> und:Knoten1,                %UND-Knoten, dann ist Baum von
    loese_alle(Knoten1,Baeume).            %der Gestalt
                                           %Knoten-->und:[U1,U2,..]
loese_alle([],[]).
loese_alle([Knoten|Knoten1], [Baum|Baeume]) :-
    loese(Knoten,Baum),
    loese_alle(Knoten1,Baeume).
```