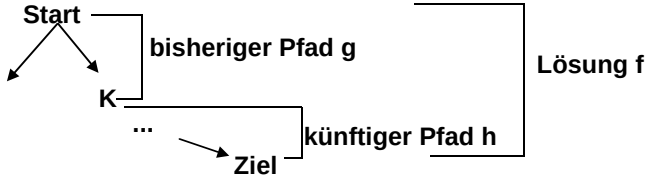
Heuristische Suche

- Kosten von Expansionen (Kanten, Übergängen) können unterschiedlich sein.
- Einige Expansionen (Kanten, Übergänge) sind vielversprechender als andere.
- Information über den Sachbereich soll ausgenutzt werden! (Bisher: uninformierte Suche.)

Kosten

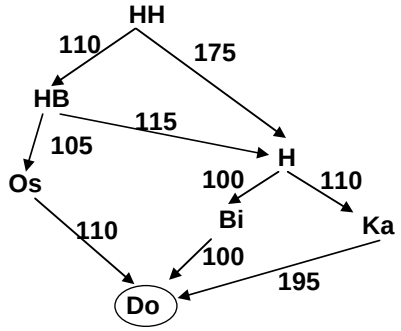
Gleichmäßige Kosten bei jeder Expansion:
- Länge des Pfades zur Lösung ergibt die Kosten
- Länge des Gesamtpfades ergibt sich aus bisherigem und zukünftigem Pfad

Wie schätzt man die Länge des künftigen Pfades ab?



Informierte Suche

Luftlinie:
HH - Do 290 km
HB - Do 200 km
Os - Do 100 km
H - Do 190 km
Bi - Do 90 km
Ka - Do 180 km



Luftlinie unterschätzt tatsächliche Entfernung.

Bergsteigen

- Die Nachfolgeknoten werden nach ihrer Entfernung vom Ziel angeordnet:
  je näher ein Zustand dem Ziel, desto besser ist er.
- Manchmal muß man sich aber vom Ziel entfernen, um es zu erreichen.
  Z.B. muß man eine Person wieder zurückfahren, um das Missionare-Kannibalen-Problem zu lösen.
- Vorgebirgsproblem: ein Knoten wird nicht expandiert, obwohl er zum Ziel führt, weil ein anderer eine bessere Bewertung hat (näher am Ziel ist, aber nicht zum Ziel führt).

Beispiel Bergsteigen

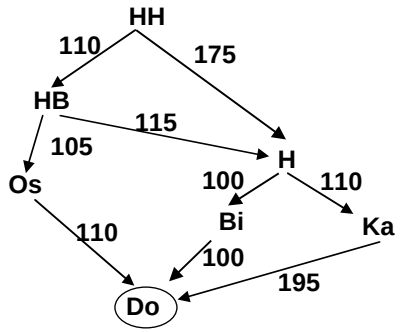
Luftlinie:
HH - Do 290 km
HB - Do 200 km
Os - Do 100 km
H - Do 190 km
Bi - Do 90 km
Ka - Do 180 km

h(HH)=290
h(HB)=200
h(H)=190
h(BI)= 90
h(Ka)=180

HH --> DO
H

BI

DO



Kein guter Weg!

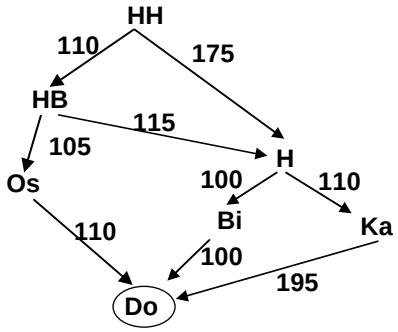
A*

g(n) zurückgelegter Weg vom Anfang bis n
h(n) geschätzter Abstand zum Ziel
h*(n) tatsächliche Kosten
f(n) = g(n) + h(n)

Wenn h(n) < h*(n) - Kosten zu niedrig geschätzt - werden mehr Knoten als nötig expandiert.
Wenn h(n) > h*(n) - Kosten zu hoch geschätzt - werden eventuell die Knoten, die zum Ziel führen, nicht expandiert.
Wenn h die untere Grenze von h* trifft, heißt das Suchverfahren A*.

Beispiel A*

Luftlinie:
HH - Do 290 km
HB - Do 200 km
Os - Do 100 km
H - Do 190 km
Bi - Do 90 km
Ka - Do 180 km



f(**HH**)=g(HH)+h(HH)=0+290
f(**HB**)=110+200=310
f(H)=175+190=365
f(**OS**)= 215+100=315
f(H)=225+190=415
f(**DO**)=325+0=325

optimaler Pfad!

f*(HH) sind die tatsächlichen Kosten des Gesamtpfades: 0+h*(HH)=325

Monotonie

Jeder Schritt von einem Knoten zum anderen kostet mindestens $\varepsilon > 0$.

- Die geschätzten Kosten f(n) können bei einem Nachfolgeknoten von n nicht kleiner werden!
  $f(n_i) \leq f(n_j)$, n_j Nachfolger von n_i
- Die Kosten des bisherigen Pfades werden nicht weniger: $g(n_i) \leq g(n_j)$, n_j Nachfolger von n_i
- Die geschätzen Kosten des Restpfades auch nicht!
  $h(n_i) \leq h(n_j)$, n_j Nachfolger von n_i

Zulässigkeit

Ein Suchverfahren ist zulässig (admissible), wenn
es für jeden Graphen einen optimalen Pfad findet und dann anhält , falls es einen solchen Pfad gibt.

Ist A* zulässig?
Beweis in 4 Schritten:
1) Lemma: jeder Knoten des optimalen Pfads ist mal in Offen.
2) A* hält nicht an, bevor das Ziel erreicht ist.
3) A* hält an.
4) Der Pfad, den A* findet, ist optimal.

Lemma

Wenn $h(n) \leq h^*(n)$ für alle Knoten n,
dann gibt es immer für jeden optimalen Pfad einen Knoten n‘ dieses Pfades in OFFEN und
es gilt $f(n') \leq f^*(n')$.

Vorüberlegungen:
$h \leq h^*$, also wird jeder relevante Knoten (und einige mehr) betrachtet.
Für jeden optimalen Pfad gibt es einen Knoten n' in Offen mit $f(n') \leq f^*(s)$.
Kein Knoten aus Offen kostet mehr als der Gesamtpfad, weil $h(n) \approx f^*(s) - g^*(n)$, h also wirklich den Restpfad schätzt.

Beweis

Sei $n_0 , n_1, ... , n_k$ ein optimaler Pfad von n_0 nach n_k .

Sei n‘ ein Knoten in Offen. Es muss einen Knoten in Offen geben, wenn es einen optimalen Pfad gibt und n_k noch nicht in Geschlossen ist, so dass der Algorithmus anhält.

Wenn n‘ der erste Knoten in Offen ist, sind alle Vorgänger von n‘in Geschlossen und g(n‘)=g*(n‘).

Also gilt: **f(n‘)** = g*(n‘) + h(n‘) ≤ g*(n‘) + h*(n‘) = **f*(n‘)**

Theorem A* ist zulässig

Wenn $h(n) \leq h^*(n)$ für alle Knoten n und das Suchproblem ist monoton,

dann ist A* zulässig.

Vorüberlegung:

fieser Fall 1: A* hält an, bevor das Ziel erreicht ist.

fieser Fall 2: A* hält nie an.

fieser Fall 3: A* findet einen Lösungspfad, der nicht optimal ist.

Beweis -- FF1

Wenn A* anhält, bevor das Ziel erreicht ist, dann muss Offen leer sein.
A* hält nur an, wenn Offen leer ist oder das Ziel erreicht ist.

Unser Lemma sagt, dass Offen immer einen Knoten enthält, der auf dem optimalen Pfad liegt bis das Ziel erreicht ist.

Der Fall kann nicht vorkommen.

Beweis -- FF2

A* hält an, weil nur endlich viele Knoten expandiert werden und nur endlich viele Pfade zu diesen durchlaufen werden, bis der optimale Pfad gefunden ist.

Knoten, die mehr als f*(s) / ε Schritte vom Start entfernt sind, werden nie geöffnet. Sie wären teurer als irgendein Knoten n‘ in Offen, der auf dem optimalen Pfad liegt.

Knoten in Geschlossen werden nie wieder betrachtet.

Somit hält A* immer nach endlich vielen Schritten an.

Beweis -- FF3

Was wäre, wenn A* einen Lösungspfad findet, der nicht optimal ist?

Der hätte höhere Kosten als der optimale:

$$f^*(t) = g(t) > f^*(s)$$

Laut Lemma muss es vorm anhalten einen Knoten n‘ in Offen gegeben haben, so dass

$$f(n') \leq f^*(s) \leq f(t).$$

Dann wählt A* den Knoten n‘ und nicht t.

q.e.d.

A* in Prolog

```
astern(Start,Ziel,Pfad,Weglaenge):-
        suche([Start],[],Ziel,Pfad,Weglaenge).

%%  suche(+Offen, +Geschlossen, +Ziel, -Pfad, -Wert)
suche([], _Geschl, _Ziel, _N, _W):-      % Misserfolg
        write('Schade'),  nl,
        !,
        fail.
suche(Offen, Geschl, Ziel, N, W):-
        best(Offen, Best, RestOffen), % Offen sortieren, besten Knoten zuerst
        ( Best = Ziel ->                                        % Erfolg
          f(Ziel, N, W)                                         % optimaler Pfad
        ;                                                       % weitersuchen
          findall(Succ, nachf(Best,Succ), AllSuccs),          % Besten expandieren
          verteile(AllSuccs, RestOffen, [Best | Geschl], NeuOffen),
                                                % Nachfolger in Offen ohne Doppelte
          suche(NeuOffen, [Best | Geschl], Ziel, N, W)
        ).
```

verteile

```
%%  verteile(+AlleNachfolger, +OffenOhneBest, +Geschlossen, -NeuOffen)
verteile([], Offen, _Geschl, Offen).                    % gibt nichts mehr zu verteilen

verteile([AK | Rest], Offen, Geschl, NeuOffen):-  % Knoten ist schon in Offen
        member(AK, Offen),
        !,                                              % roter CUT!
        verteile(Rest, Offen, Geschl, NeuOffen).        % nun den Rest verteilen

verteile([AK | Rest], Offen, Geschl, NeuOffen):-  % Knoten ist schon in
                                                  % Geschlossen
        member(AK,Geschl),
        !,                                              % roter CUT!
        verteile(Rest, Offen, Geschl, NeuOffen).        % nun den Rest verteilen

verteile([AK | Rest], Offen, Geschl, NeuOffen):-  % Knoten in Offen eintragen
        verteile(Rest, [AK | Offen], Geschl, NeuOffen).
```


best

```
%%  best(+Offen, -Best, -RestOffen)
%%
%%          hier wird f = g + h berechnet

best(Liste, BesteElem, RestListe):-                     % nach Heuristik sortieren
                                                        % Heuristik ist hier f=g+h
        sortiere(Liste, [BesteElem | RestListe], f).
```


sortiere (1)

```
%%  Schnellsortierung für Werte von g, um den minimalen Wert zu finden,
%%  und für Offen nach der A* Heuristik
%%  sortiere(+ZuTun, -SortierteListe, +Vergleichskriterium)
%%
%%  Rekursive Variante von Quicksort,  Split immer auf dem ersten Element;
%%  in Vergleich steht f oder min:  f sortiert nach der f Heuristik von A*,
%%  min sortiert Zahlen aufsteigend

sortiere([SplitElem | RestListe], SortierteListe, Vergleich):-
        split(RestListe, SplitElem, Klein, Gross, Vergleich),
        !,          %% ein gefundener Split ist genug!
                    %% roter Cut!
        sortiere(Klein, Kleinsortiert, Vergleich),
        sortiere(Gross, Grosssortiert, Vergleich),
        append(Kleinsortiert, [SplitElem | Grosssortiert], SortierteListe).

sortiere([],[], _Vergleich).
```


sortiere (2)

```
split([ X | Xs], SplitElem, [X | KleinSortiert], GrossSortiert, f):-
        f(SplitElem,_Pfad,WertY),
        f(X,_P,WertX),
        WertX =< WertY,
        split(Xs, SplitElem, KleinSortiert, GrossSortiert, f).

split([ X | Xs], SplitElem, KleinSortiert, [X | GrossSortiert], f):-
        f(SplitElem,_Pfad,WertY),
        f(X,_P,WertX),
        WertX > WertY,
        split(Xs, SplitElem, KleinSortiert, GrossSortiert, f).

split([ X | Xs], SplitElem, [X | KleinSortiert], GrossSortiert, min):-
        X =< SplitElem,
        split(Xs, SplitElem, KleinSortiert, GrossSortiert, min).

split([ X | Xs], SplitElem, KleinSortiert, [X | GrossSortiert], min):-
        X > SplitElem,
        split(Xs, SplitElem, KleinSortiert, GrossSortiert, min).

split([], _SplitElem, [], [], _Vergleich).              % Abbruch fuer beide Varianten
```

f = g + h

```
gstern([Knoten, hh],_Knotenliste, G):-  % Zusammensetzen der Pfadkosten
        g([Knoten, hh], G).

gstern([Knoten, Vor | Rest],Knotenliste, Gesamt):-
        g([Knoten, Vor], Weg),
        \+ member(Vor,Knotenliste),
        gstern([Vor | Rest],[Vor | Knotenliste], Vorweg),
        Gesamt is Weg + Vorweg.

f(hh, _Pfad, 290).                      % Hack fuer HH

f(Knoten, [Knoten | Pfad], Wert):-      % f mit minimalem g
        h(Knoten, _Ziel, H),
        findall(G, gstern([Knoten | _P],[], G), AlleG),
        sortiere(AlleG, [GMin | _], min),
        Wert is H + GMin,
        gstern([Knoten | Pfad],[], GMin).
```


Wegbeispiel HH → DO

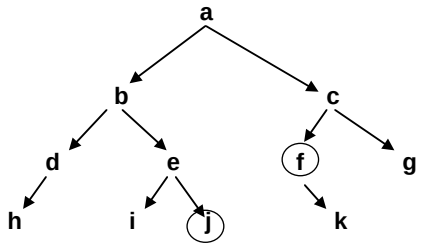
```
%%  Der Graph

nachf(hh,hb).
nachf(hh,h).
...

%%  Die Kosten für die Kanten

g([hb,hh],110).
g([h,hb],115).
...

%%  Die Schätzwerte für die Heuristik

h(hh,do,290).
h(hb,do,200).
...
```


nur g, nur h

Wir betrachten **nur h**,
berücksichtigen also nicht bisher gegangenen Pfad -
wir erhalten Bergsteigen.
a und c werden expandiert

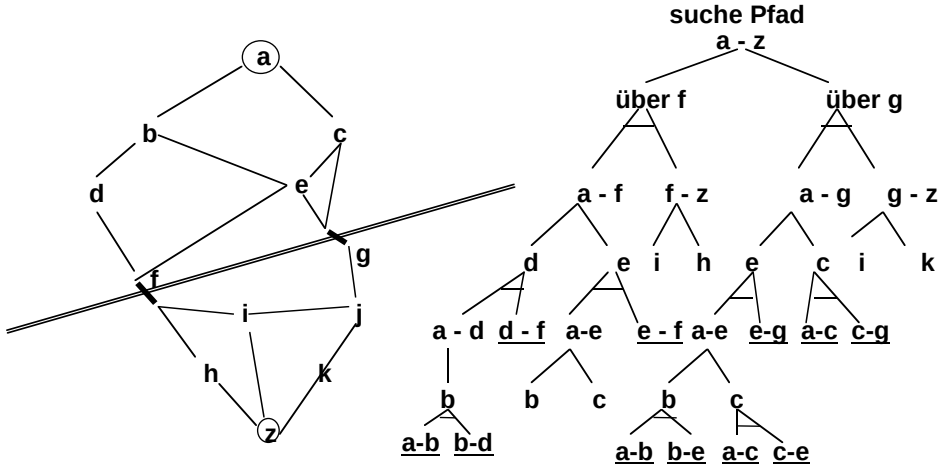
Wir betrachten **nur g**,
berücksichtigen also nicht den zukünftigen Pfad -
wir erhalten eine Breitensuche.
a, b, c, d, e, f (g) werden expandiert.

Wir expandieren immer den ersten Nachfolger,
wir erhalten eine uninformierte Tiefensuche.
a, b, d, (h), e, i werden expandiert.

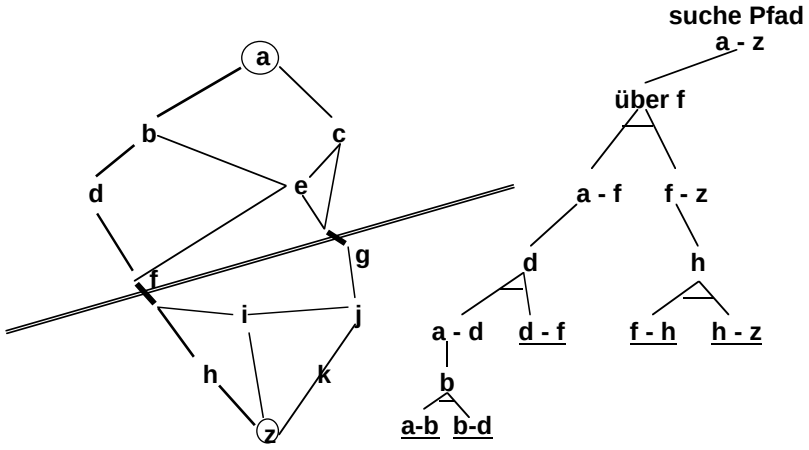


Problemzerlegung

Straßenkarte mit zwei Brücken über den Fluß



Eine Lösung



Problemzerlegung

Statt wie bei den Missionaren und Kannibalen eine Folge von Operatoranwendungen zu suchen, mit der der Start- in den Zielzustand überführt wird,
zerlegt man jetzt das Gesamtproblem in Teilprobleme, für die man eine Lösung sucht.
Alternative Zerlegungen werden als Entscheidungspunkte angegeben. Man fährt über f ODER über g.
Alle Teilprobleme müssen gelöst werden, damit das Gesamtproblem gelöst ist.
Man muß von a nach f UND von f nach z finden.

Verwenden der Prologkontrolle

In Prolog kann man das Problem leicht codieren:

```
pfad(a,z) :- pfad(a,f), pfad(f,z), write('f').
pfad(a,z) :- pfad(a,g), pfad(g,z), write('g').
pfad(a,f) :- pfad(a,d), pfad(d,f), write('d').
pfad(a,f) :- pfad(a,e), pfad(e,f), write('e').
pfad(a,d) :- pfad(a,b),pfad(b,d), write('b').

pfad(a,b). pfad(b,d). pfad(d,f).

pfad(a,e) :- pfad(a,c), pfad(c,e), write('c').

pfad(a,c). pfad(c,e). pfad(e,f).

pfad(f,z) :- pfad(f,h), pfad(h,z), write('h').

pfad(f,h). pfad(h,z).

pfad(f,z) :- pfad(f,i), pfad(i,z), write('i').

pfad(f,i). pfad(i,z).
```

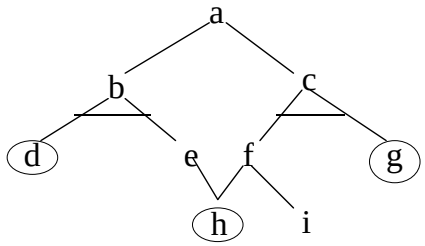
Prolog findet hier 4 Lösungen:

```
b d h f
b d i f
c e h f
c e i f
```

Klauseln sind ge-ODER-t.
Literale sind ge-UND-et.

Explizite Darstellung

```
:- op(600, xfx, -->). %Infixoperator
:- op(500, xfx, :).    % Infixoperator

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



Tiefensuche in UND-ODER-Bäumen

```
%loese(+Knoten, -Loesungsbaum)
%findet für einen Startknoten einen Baum zu einem Zielknoten


loese_alle( [], []).

loese_alle ([ ], [ ]).
loese_alle ([ Knoten | Knoten1], [Baum |Baeume]) :-
      loese ( Knoten, Baum),
      loese_alle (Knoten1, Baeume).

loese(Knoten, Knoten) :- ziel (Knoten).
loese(Knoten, Knoten --> Baum) :-
      Knoten --> oder:Knoten2,          %ODER-Knoten
      member(Knoten1, Knoten2),         %Nachfolger wählen
      loese(Knoten1, Baum).
loese(Knoten, Knoten --> und: Baeume) :-
      Knoten --> und: Knoten1,          %UND-Knoten
      loese_alle(Knoten1, Baeume).      %alle Nachfolger lösen
```