Künstliche Intelligenz

Prof. Dr. Dirk Krechel
Hochschule RheinMain

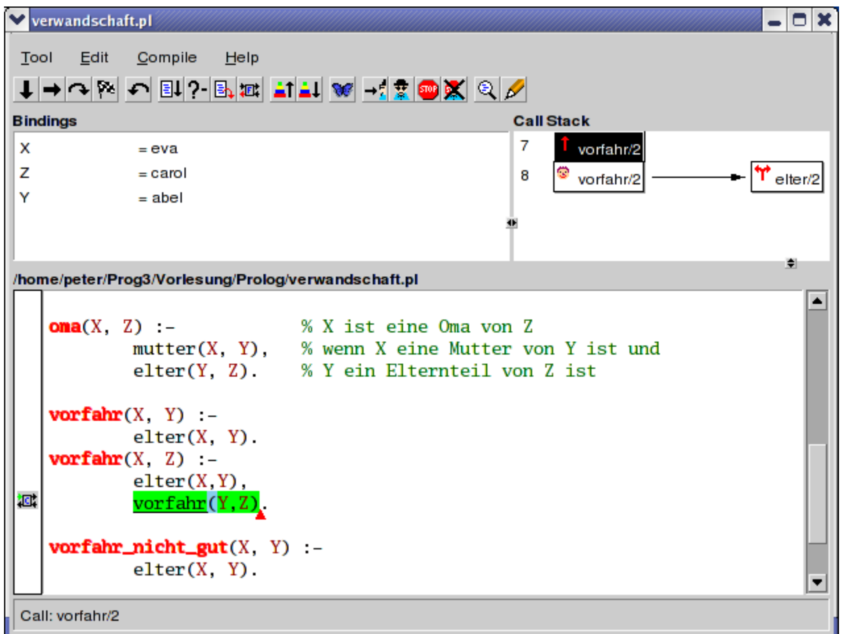
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

Inhalte

- **Symbolische Verfahren, Logik**
  - Aussagenlogik, Prädikatenlogik
  - **Horn Logik, Prolog**
- Suchen und Bewerten
  - Problemlösen durch Suche
    - Uninformierte Suche
    - Heuristische Suche
    - Spielbäume

Prolog bei der Ausführung beobachten

- Aufrufe und Variablen-belegung bei der Tiefensuche
  - Text-basiert mit trace.
  - Graphisch mit guitracer.

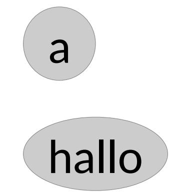



```
?- trace.
[trace]  ?- vorfahr(X, carol).
   Call: (8) vorfahr(_G315, carol) ? creep
   Call: (9) elter(_G315, carol) ? creep
   Exit: (9) elter(lilith, carol) ? creep
   Exit: (8) vorfahr(lilith, carol) ? creep
X = lilith ;
   Redo: (8) vorfahr(_G315, carol) ? creep
   Call: (9) elter(_G315, _L192) ? creep
   Exit: (9) elter(adam, kain) ? creep
   Call: (9) vorfahr(kain, carol) ? creep
   Call: (10) elter(kain, carol) ? creep
   Fail: (10) elter(kain, carol) ? creep
   Redo: (9) vorfahr(kain, carol) ? creep
   …
   Redo: (10) vorfahr(susi, carol) ? creep
   Call: (11) elter(susi, _L214) ? creep
   Fail: (11) elter(susi, _L214) ? creep
No
?- guitracer.
% The graphical front-end will be used for subsequent tracing
Yes
?- trace.
Yes
[trace]  ?- vorfahr(X, carol).
```

Datenstrukturen sind Terme

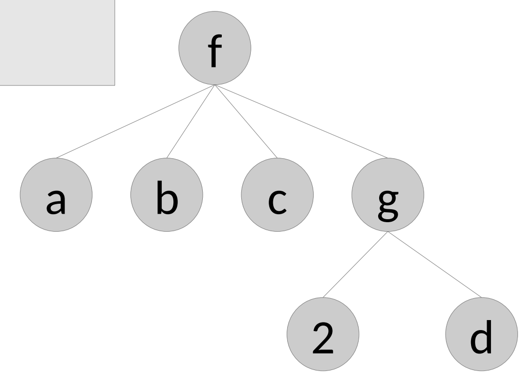
- Atome
  - Kleingeschriebene Wörter
  - Funktoren ohne Parametern

```
?- X = a, Y = hallo.
X = a
Y = hallo
```



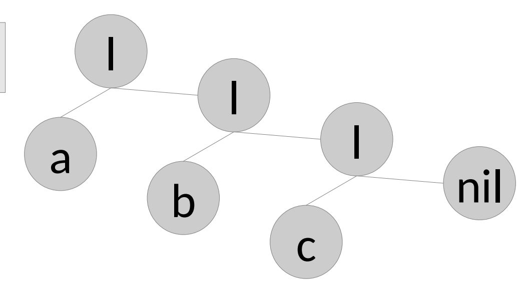
- Terme
  - Baumstruktur
  - Funktoren mit mehreren Argumenten
  - Argumente sind Terme
  - Atome sind Terme

```
?- X = f(a,b,c, g(2, d)).
X = f(a, b, c, g(2, d))
```



- Listen
  - Darstellbar als Baum
  - Spezielle Notation in Prolog möglich

```
l(a, l(b, l(c, nil)))
```

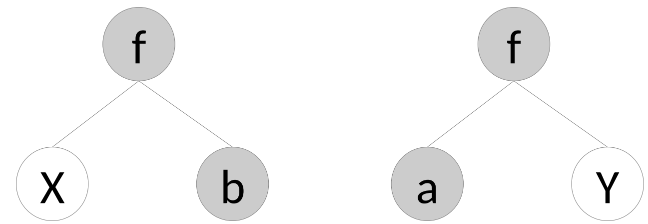



```
?- X = [a,b,c], Y = [a | [b | [c | []]]].
X = [a, b, c]
Y = [a, b, c]
Yes
```

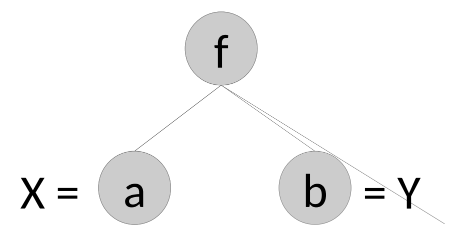
Unifikation

- =
  - Nicht Zuweisung sondern Unifikation
  - Versucht eine Variablenbelegung
    zu finden zwei Terme gleich zu machen
  - Variablen an beliebigen Stellen im Term

```
?- f(X, a) = f(b, Y).
```



- Unifikationsmethode
  - Gleichungsmenge
  - Wenn Funktoren nicht gleich, FAIL
  - Ersetze Funktionsgleichung durch
  - Entferne triviale Gleichungen zwischen Atomen
  - Wenn eine Seite eine Variable ist, dann ersetze
    jedes Vorkommen der Variable durch andere Seite
  - Ergebnis wie Robinson-Unifkation



- Zyklische Terme vermeiden
  - Je nach Prolog erlaubt,
    mit Occurscheck verboten

```
?- f(X) = X.
X = f(**)
Yes
?- unify_with_occurs_check(f(X),X).
No
```

Unifikation – Beispiele

- Reihenfolge beliebig, Menge von Gleichungen, Vereinfachung
  - { X = f(Y), Y= a }
    { X = f(a), Y=a }
- Gleichsetzen der Unterterme, Vereinfachung
  - { f(X,a) = f(b,Y) }
    { X = b, a = Y }
  - { f(X, a, g(Y, X)) = f(c, a, Z) }
    { X = c, a = a, g(Y,X) = Z }
    { X = c, g(Y, X) = Z}
    { X = c, g(Y, c) = Z}
- Nicht immer eine Lösung
  - { f(X,b) = f(a,X) }
    { X =a, b = X}   % Widerspruch

```
?- X = f(Y), Y = a.
X = f(a)
Y = a
Yes
?- f(Y) = X, Y = a.
Y = a
X = f(a)
Yes

?- f(X, a) = f(b, Y).
X = b
Y = a
Yes

?- f(X,a,g(Y, X)) = f(c,a,Z).
X = c
Z = g(Y, c)
Yes

?- f(X, b) = f(a, X).
No
```

Terminierung

- Terminierung nicht garantiert
  - Aufgrund der Strategie endlose Inferenzketten möglich
  - Im Beispiel liefert Anfrage
    `?- p(a).`
    kein Ergebnis
  - Es wird immer wieder (nach Umbenennung der Variablen X) mit der ersten Regel resolviert, aber kein Fortschritt erzielt

```
?- p(a).
ERROR: Out of local stack
```

- Wie beim Programmieren denken
  - Reihenfolge beachten
  - Problemreduktion,wie Rekursion
    - Problem muss kleiner werden Ordnung
    - Es darf nur endlich viele Schritte bis zum trivialen Fall geben, diskrete Ordnung

```
p(X) :- p(X).
p(a).
```

```
p(X) :- p(f(X)).
p(a).
```

wäre ok

```
p(f(X)) :- p(X).
p(a).
```

```
p(a).
p(X) :- p(X).
```

```
?- p(b).
ERROR:
```

```
p(a).
```

```
?- p(b).
false
```

nicht herleitbar

Negation und Closed World Assumption

- Negation
  - Nicht unterstützt in Hornlogik
  - Nicht unterstützt in Prolog
- Negation as Failure
  - Wenn mit den vorhandenen Regeln eine Anfrage nicht bewiesen werden kann, dann wird angenommen die Anfrage gilt nicht
  - Statt Negation besser „nicht herleitbar“ $\nvdash P(x)$
  - Eingebautes Prädikat \+ sieht ähnlich aus wie $\nvdash$
    (hieß früher not())
    - Metaprädikat (hat Prädikat als Argument)
    - Vermeiden!
  - ACHTUNG: Eine solche Aussage kann sich ändern,
    wenn neue Fakten hinzu kommen, also wird **nicht bewiesen** $\vdash \neg P(x)$
    keine Negation in Hornlogik
- Closed World Assumption
  - Alles was nicht als wahr gezeigt werden kann ist falsch

```
p(a).
```

```
?- p(b).
false.
?- \+(p(b)).
true.
```

Arithmetik

- Arithmetik
  - Repräsentation von natürlichen Zahlen durch Terme
    - 0 durch `0`
    - 1 durch `succ(0)`
    - 2 durch `succ(succ(0))`
    - …
  - Addition
    - Neutrales Element 0
    - x+y = (x-1) + (y+1)
    - (oft) terminierend, da kleiner werdend und gegen 0
    - Idealerweise „Sorte“ festlegen
- Anfragen
  - Rechnen
  - Aber auch umstellen!
  - Und Lösungen aufzählen!

```
zahl(0).
zahl(succ(X)) :-
   zahl(X).

add(0, X, X).
add(succ(X), Y, succ(Z)) :-
   add(X, Y, Z).
```

```
add(0, X, X) :-
      zahl(X).
```

```
?- zahl(X).
X = 0 ;
X = succ(0) ;
X = succ(succ(0)) ;
X = succ(succ(succ(0))) .

?- add(succ(succ(0)), succ(0), X).
X = succ(succ(succ(0))).

?- add(succ(succ(0)), X, succ(succ(succ(succ(0))))).
X = succ(succ(0)).

?- add(X, Y, succ(succ(0))).
X = 0,   Y = succ(succ(0)) ;
X = succ(0),   Y = succ(0) ;
X = succ(succ(0)),   Y = 0 ;
false.
```

Programmieren mit Listen – member

- member
  - X ist Element einer Liste, wenn es das erste Element der Liste ist
  - X ist Element einer Liste, wenn es in der Liste außer dem ersten ist
- Kontrollstrukturen
  - Rekursion statt Schleifen
  - Mehrere Regeln statt Verzweigung
- Verwendung
  - Test
  - Aufzählung
- Achtung
  - Kann durch starre Tiefensuche unendlich lange laufen

```
member(X, [ X | _]).
member(X, [ _ | L]) :-
   member(X, L).
```

```
?- member(a,[a,b,c]).
Yes
?- member(c,[a,b,c]).
Yes
?- member(d,[a,b,c]).
No
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
?- member(a,L).
L = [a|_G246] ;
L = [_G245, a|_G249] ;
L = [_G245, _G248, a|_G252] ;
L = [_G245, _G248, _G251, a|_G255];
...
```

Programmieren mit Listen – append

- append – Listen zusammenfügen
  - Eine Liste an die leere Liste angefügt ist die Liste
  - Das erste Element der ersten Liste ist das erste Element der zusammengefügten Liste
  - Die Restliste der zusammengefügten Liste ist die erste Liste ohne erstes Element angefügt an die zweite Liste

```
append([], L, L).
append([X|L0], L1, [X|L2]) :-
   append(L0, L1, L2).

% append ist eingebaut in SWI-Prolog
```

- Beispiel ?- append([a,b], [c,d], L).

?- *X=a*, *L0=[b]*, *L1=[c,d]*, *L=[a|L2]*,
append([b], [c,d], L2).

Neue Variablen in Regel 2 und Anwendung Regel 2

?- X =a, L0=[b], L1=[c,d], L=[a,*b|L2'*],
X' =b, *L0'=[]*, *L1'=[c,d]*, *L2=[b|L2']*,
append([], [c,d], L2').

Neue Variablen in Regel 2 und Anwendung Regel 2

?- X =a, L0=[b], L1 =[c,d], L= [a,b,*c,d]*,
X' =b, L0'=[], L1'=[c,d], L2= [b,*c,d]*,
L2'=[c,d].

Neue Variablen in Regel 1 und Anwendung Regel 1, keine weiteren Prädikate

?- L = [a,b,*c,d]*.

Lösung nach Elimination nicht sichtbarer Variablen

Beispiele mit append

- Füge zwei Listen zusammen
- Welche Liste muss man anfügen?
- Welche Liste, außer dem ersten Element, muss man anfügen?
- Welches …
- Welche Möglichkeiten gibt es zwei Listen zusammenzufügen um eine vorgegebene Liste zu erhalten?

```
?- append([a,b,c], [d,e,f], Z).
Z = [a, b, c, d, e, f]

?- append([a,b,c], X, [a,b,c,d,e,f]).
X = [d, e, f]

?- append([a,b,c], [d|X], [a,b,c,d,e,f]).
X = [e, f]

?- append([a,b,c], [X|[e,f]], [a,b,c,d|Z]).
X = d
Z = [e, f]

?- append(X, Y, [a,b,c,d,e,f]).
X = []                 Y = [a, b, c, d, e, f] ;
X = [a]                Y = [b, c, d, e, f] ;
X = [a, b]             Y = [c, d, e, f] ;
X = [a, b, c]          Y = [d, e, f] ;
X = [a, b, c, d]       Y = [e, f] ;
X = [a, b, c, d, e]    Y = [f] ;
X = [a, b, c, d, e, f] Y = [] ;
```

Weitere Listen-Prädikate

- Listen-Bibliothek
  - Wird in SWI-Prolog automatisch bei Bedarf geladen
  - Viele sinnvolle Listen-Prädikate
  - Parameter in Doku. annotiert: ? Ein/Ausgabe, + Eingabe, - Ausgabe
    Hinweis auf sinnvolle Verwendung
- Auszug
  - `nth0(?Index, ?List, ?Elem)`
    Elem ist Element der Liste an Stelle Index (ab 0 gezählt)
    nth1 wie nth0 nur ab 1 gezählt
  - `delete(+List1, ?Elem, ?List2)`
    In List2 sind alle Elemente von List1 außer Elem,
    List1 muss instanziierte Liste sein
  - `select(?Elem, ?List, ?Rest)`
    Elem ist Element der Liste, Rest ist Liste ohne Elem
  - `permutation(?List1, ?List2)`
    List1 ist eine Permutation von List2

Cut

- Ausführungsstrategie
  - Anfrage von links nach rechts
  - Regeln von oben nach unten
  - Tiefensuche
- Problem
  - Endloser Abstieg bei Tiefensuche
  - Absehbare erfolglose Suche in Teilbaum
- Lösung
  - Tiefensuche abschneiden
  - Cut-Operator, !
- Cut
  - Achtung, keinerlei logische Entsprechung ausschließlich operational
  - Man verliert Möglichkeit aufzuzählen
  - Vermeiden

```
?- lebt(X).
X = rose;
X = lilie;
X = hund;
...
```

```
lebt(X) :- blume(X).
lebt(X) :- tier(X).
blume(rose).
blume(lilie).
tier(hund).
tier(katze).
tier(maus).
```

```
[trace]  ?- lebt(lilie).
   Call: (6) lebt(lilie) ? creep
   Call: (7) blume(lilie) ? creep
   Exit: (7) blume(lilie) ? creep
   Exit: (6) lebt(lilie) ? creep
true ;
   Redo: (6) lebt(lilie) ? creep
   Call: (7) tier(lilie) ? creep
   Fail: (7) tier(lilie) ? creep
   Fail: (6) lebt(lilie) ? creep
false.

[trace]  ?-
```

Wenn was eine Blume ist, dann ist es kein Tier, absehbar erfolglos

```
lebt(X) :- blume(X),!.
```

```
?- lebt(X).
X = rose.
?-
```

```
[trace]  ?- lebt(lilie).
   Call: (6) lebt(lilie) ? creep
   Call: (7) blume(lilie) ? creep
   Exit: (7) blume(lilie) ? creep
   Exit: (6) lebt(lilie) ? creep
true.
```

Cut – Beispiel

- Beispielprogramm
  - Einstellige Prädikate p, q, r
- Beispielanfragen
  - p(1).
    - Ja
    - Cut wird nicht abgearbeitet
    - p(1) als Fakt führt zum Erfolg
  - p(2).
    - Ja
    - Cut wird abgearbeitet
    - Erfolg wegen r(2)
  - p(3).
    - Nein
    - Cut wird abgearbeitet
    - Cut verhindert Backtracking

```
p(X) :-
        q(X),
        !,
        r(X).
p(1).
p(2).
p(3).
q(2).
q(3).
r(2).
```

Cut und Negation

- Negation as Failure
  - Wir nehmen an, dass not(P(X)) gilt, wenn P(X) nicht beweisbar
- Selbst implementierbar
  - cut verwenden
  - call verwenden
    - Ruft ein Prädikat, versucht eine Aussage zu beweisen
    - Metaprädikat, das Eval von Prolog
  - Versucht P zu zeigen
    - Wenn es klappt, dann Cut (nicht mehr über Stelle zurück Backtracking) fail, forciert Fehlschlag
    - Wenn es nicht klappt, dann zweite Klausel; es klappt
- Nicht sehr intuitiv
  - Zählt zum Beispiel nicht auf
  - Vermeiden, nicht selbst machen sondern \+ nehmen

```
not(P) :-
        call(P),
        !,
        fail.
not(P).
```

```
?- not(lebt(lilie)).
false.

?- not(lebt(haus)).
true.

?- not(lebt(X)).
false.
```

Meta-Prädikate

- Meta-Prädikate
  - Prädikate
  - Arbeiten mit Prädikaten als Argumenten statt Termen
  - Nicht mehr Prädikatenlogik erster Stufe
  - Nur für Spezialaufgaben, vermeiden
  - Bekannte Beispiele: call, not
- Weitere Beispiele
  - apply(:Goal, +List): Fügt Listenelemente als Parameter an Goal an und ruft es
  - call_with_depth_limit(:Goal, +Limit, -Result): Tiefenbeschränkte Suche, für iterative deepening
  - findall(+Template, :Goal, -Bag): Sucht alle Lösungen für Goal und sammelt in Bag die Bindungen von Template für jede Lösung
  - … spezifisch je Implementierung, Dokumer

In Dokumentation Parameterannotation:
+ Eingabe, instanziiert
- Ausgabe, Variable
? Ein/Ausgabe
: Prädikat

```
?- apply(append, [[1,2], [3,4], X]).
X = [1, 2, 3, 4].
```

```
?- findall(X, append(X, Y, [1,2,3,4]), L).
L = [[], [1], [1, 2],
    [1, 2, 3], [1, 2, 3, 4]].
```

Modulsystem

- Modul definieren
  - Erste Zeile Direktive
  - :- module(<name>, <export>)
  - <name> ist modulname
  - <export> ist Liste exportierter Prädikate
    - Je Prädikat die Stelligkeit
- Beispiele
  - Listen rumdrehen, nur Prädikat reverse nach außen
- Modul verwenden
  - Direktive
  - :- use_module(<name>) im Programmcode
  - library(<name>) für Suche Modul in Suchpfad
  - Wird in globalen Namensraum importiert

```
:- module(reverse, [
     reverse/2  % (?L1, ?L2)
   ]).

reverse(L0, L1) :-
   rev(L0, [], L1).

rev([], L, L).
rev([H|L0], L1, L2) :-
   rev(L0, [H|L1], L2).
```

```
?- use_module(reverse).
% reverse compiled into reverse
true.

?- reverse([1,2,3], X).
X = [3, 2, 1].

?- use_module(library(ordsets)).
%  library(oset) compiled into oset
% library(ordsets) compiled into ordsets
true.

?- ord_add_element([], b, X),
   ord_add_element(X, a, Y).
X = [b],
Y = [a, b].
```

Eingebaute Arithmetik

- Berechnung
  - = ist Termgleichheit/Unifikation
  - is für Auswertung und Zuweisung
- Arithmetische Vergleichsoperationen
  - <,>,>=
    wie gewohnt
  - =< statt <=!
    (<= als Implikation verwendet)
  - Arithmetische Gleichheit
    =:= gleich
    =\= ungleich
- Nur Grundterme
  - Achtung: Keine Vari-ablen in arithmetischen Ausdrücken

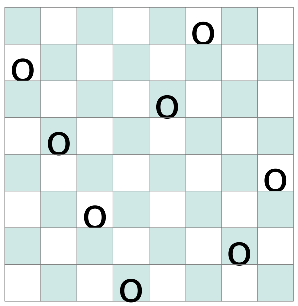
```
? 2+3 = 3+2.
No
?- X is 3+2.
X = 5

?- 2+3 >= 1+5.
No
?- 2+5 >= 1+5.
Yes
?- 2+3 < 1+5.
Yes
?- 2+3 >= 3+1.
Yes

?- 3+2 =:= 2+3.
Yes
```

```
? X = 3+2.
X = 3+2
?- X is 3+2, X=5.
X = 5

?- 2+3 > 3+1.
Yes
?- 2+3 <= 3+4.
ERROR: Syntax error:
Operator expected

?- 2+3 =< 3+4.
Yes

?- 3+2 =\=
2+3.
No
?- 3+2 =\=
2+4.
Yes
```

```
?- X = 3, X+2 =:= 2+X.
X = 3
Yes
?- X+2 =:= 2+X, X = 3.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

Beispiel – N-Damen

- N-Damen Problem
  - N Damen auf einem Schachbrett der Größe NxN verteilen (Alg. u. Datenstrukturen)
  - Generieren und Testen



```
?- ndamen(8,Damen).
Damen = [4, 7, 3, 8, 2, 5, 1, 6]
Yes

?- findall(Damen, ndamen(8,Damen), Loes),
   length(Loes, LenLoes).
Loes = [[4, 7, 3, 8, 2, 5, 1, 6], ...]
LenLoes = 92
```

```
ndamen(N, Ds) :-
   range(N, NL, Ds),
   permutation(NL, Ds), % generate
   sicher(Ds).        % test

% Zahlen 1..N, N Variablen
range(0, [], []).
range(N, [N|L], [_|Ds]) :-
   N >= 0,
   N1 is N-1,
   range(N1, L, Ds).

sicher([]).
sicher([D|Ds]) :-
   sicher(Ds, 1, D),
   sicher(Ds).

% Dame sicher wenn Diagonale frei
sicher([], _, _).
sicher([TD|Ds], N, D) :-
   TD + N =\= D,
   TD - N =\= D,
   N1 is N+1,
   sicher(Ds, N1, D).
```

CLP(R)

- Arithmetik über reelle Zahlen
  - Gleichungen und Ungleichungen (Constraints) über Variablen, die reelle Zahlen sein können
  - In SWI-Prolog
    - Bibliothek importieren
    - (Un-)Gleichungen in {}
- Beispiele
  - Gleichung mit zwei Unbekannten
  - Ein Zug fährt mit 100km/h wie weit in 80 Minuten
  - Lineares Gleichungssystem mit eindeutiger Lösung

```
?- use_module(library(clpr)).
Yes

?- {X + 3 = Y}, {Y = 2}.
X = -1.0
Y = 2.0
Yes

?- {60*KMM = 100}, {80*KMM =
Strecke}.
F = 1.66667
Strecke = 133.333
Yes

?- {3*X + 4*Y - 2*Z = 8,
      X - 5*Y + Z = 10,
    2*X + 3*Y -Z = 20}.
X = 15.5
Y = 8.25
Z = 35.75
Yes
```

Verarbeitungsmodell

- Verarbeitungsmodell
  - Hinzufügen von Constraints (statt/zusätzlich zu Termgleichungen)
  - Bei Backtracking rückgängig machen
  - Bei Ergebnis Projektion auf sichtbare Variablen
- Beispiel – Zug
  - Programm sammelt Constraints
  - Berechne Strecke bei gegebener Zeit und Geschwindigkeit
  - Berechne Zeit bei gegebener Geschwindigkeit und Strecke
  - Berechne Zeit und Strecke bei gegebener Geschwindigkeit
    - Geht auch – Formel!

```
zug(MIN, KMH, Strecke) :-
   {60*KMM = KMH},
   {MIN*KMM = Strecke}.
```

```
?- zug(80, 100, S).
S = 133.333
Yes
?- zug(Min, 120, 300).
Min = 150.0
?- zug(10, 120, 100).
No
?- zug(Min, 120, Strecke).
{Strecke=2.0*Min}
...
Yes
```

Grenzen von CLP(R)

- Zahlbereich R
  - Fließpunktzahlen und nicht R
  - Alternative Q
- Nur lineare Constraints
  - Zumindest nur lineare Constraints immer korrekt
  - Nichtlineare Constraints werden immer akzeptiert
    - Lösen SEHR aufwendig/ praktisch unmöglich
    - Aktiv sobald durch Instanziierung linear
  - Beispiel:
    - X*X = 1
      X ist entweder 0 oder 1,
      wird nicht erkannt
    - Sobald instanziiert wird, wird geprüft

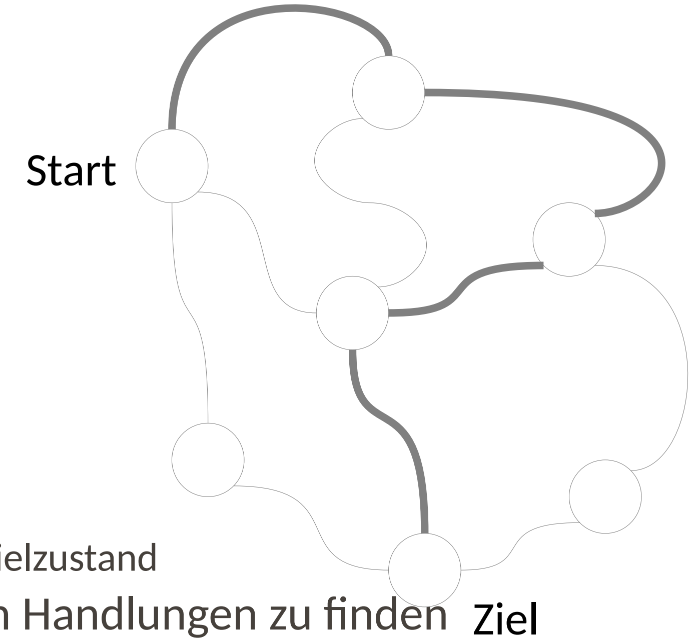
```
?- {X*X = X}.
{-X+X^2=0.0}
Yes

?- {X*X = X}, {X > 0.5}.
{X>0.5}
{-X+X^2=0.0}
Yes

?- {X*X = X}, {X > 2}. % nicht loesbar!
{X>2.0}
{-X+X^2=0.0}
Yes

?- {X*X = X}, {X > 0.5}, {X=1}.
X = 1.0
Yes
?- {X*X = X}, {X=3}.
No
```

Inhalte

- Symbolische Verfahren, Logik
  - Aussagenlogik, Prädikatenlogik
  - Horn Logik, Prolog
- **Suchen und Bewerten**
  - **Problemlösen durch Suche**
    - **Uninformierte Suche**
    - Heuristische Suche
    - Spielbäume

Suchen

- Modellierung
  - Viele Probleme darstellbar als
    - Zustandsraum (Umgebung), (un)endliche Menge diskreter Zustände
    - Ausgewiesener Startzustand und Zielzustand
  - Handlungskette zur Zielerreichung
    - Sequentielle Ausführung (Reihenfolge) von elementaren Schritten
    - Jeder Schritt/Handlung, (un)endliche Menge von Operatoren,
überführt Zustand in anderen Zustand
    - Beginn bei Startzustand, letzter Schritt überführt in Zielzustand
  - Problemlösen heißt eine richtige Reihenfolge von Handlungen zu finden die Startzustand in Zielzustand überführt



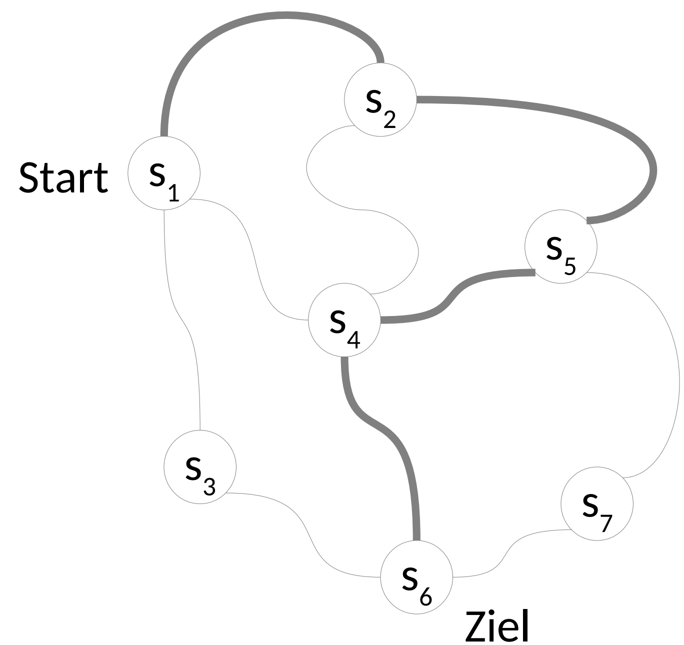
- Problemlösungsstrategien
  - Uninformierte Suche
  - Heuristische Suche
  - Stochastische Suche

Suchen – Begriffsbildung

- Suchproblem
  - *Zustandsraum* $S = \{s_1, ..., s_n\}$, Menge von Zuständen
  - *Operatoren* (Zustandsübergange) als partielle Funktionen $o: S \rightarrow S$
    - Ein Operator o führt einen Zustand s_i in einen Zustand s_j wenn $o(s_i) = s_j$
  - *Initialzustand* $s_I \in S$, Startzustand
  - *Zielbeschreibung* $G: S \rightarrow Bool$, Ziel ist mit s erreicht wenn G(s) gilt
    - Alternativ Menge von Zielzuständen S_G angeben
- Pfade
  - Ein *Pfad* p ist eine geordnete Operatorenfolge $<o_1,...,o_n>$
  - Hintereinanderausführung: $p(s_i) = s_j$ gdw $o_n(o_{n-1}(....o_1(s_i)...)) = s_j$
  - Zustand s_j ist *erreichbar* von s_i falls ein Pfad p von s_i nach s_j existiert
- Pfadkosten g(p)
  - Meist Summe der einzelnen Operatoren $g(<o_1,...,o_n>) = g(o_1)+...+g(o_n)$
  - Kosten können von besuchten Zuständen abhängen
  - Kosten können uniform sein, Pfadkosten = Länge des Pfades

Suchproblem und Lösung

- Ein *Suchproblem* besteht aus
  - Zustandsmenge S
  - Ein Initialzustand S_I
  - Zielbeschreibung G
  - Operatorenmenge O
  - Pfadkostenfunktion g
- Eine *Lösung* eines Suchproblems ist
  - ein Pfad p, der den Initialzustand S_I in einen Zustand $s_G \in S_G$ überführt, der die Zielbedingung erfüllt
- Lösungskosten einer Lösung ist
  - der Wert der Pfadkostenfunktion von S_I nach s_G



$S = \{s_1, s_2, s_4, s_4, s_5, s_6, s_7\}$

$S_I = s_1$

$S_G = \{s_6\}$

$s_G = s_6$

Lösung $p = <o_{12}, o_{25}, o_{54}, o_{46}>$

Uniforme Kosten: $\forall i,j: g(o_{ij}) = 1$

$g(p) = 4$

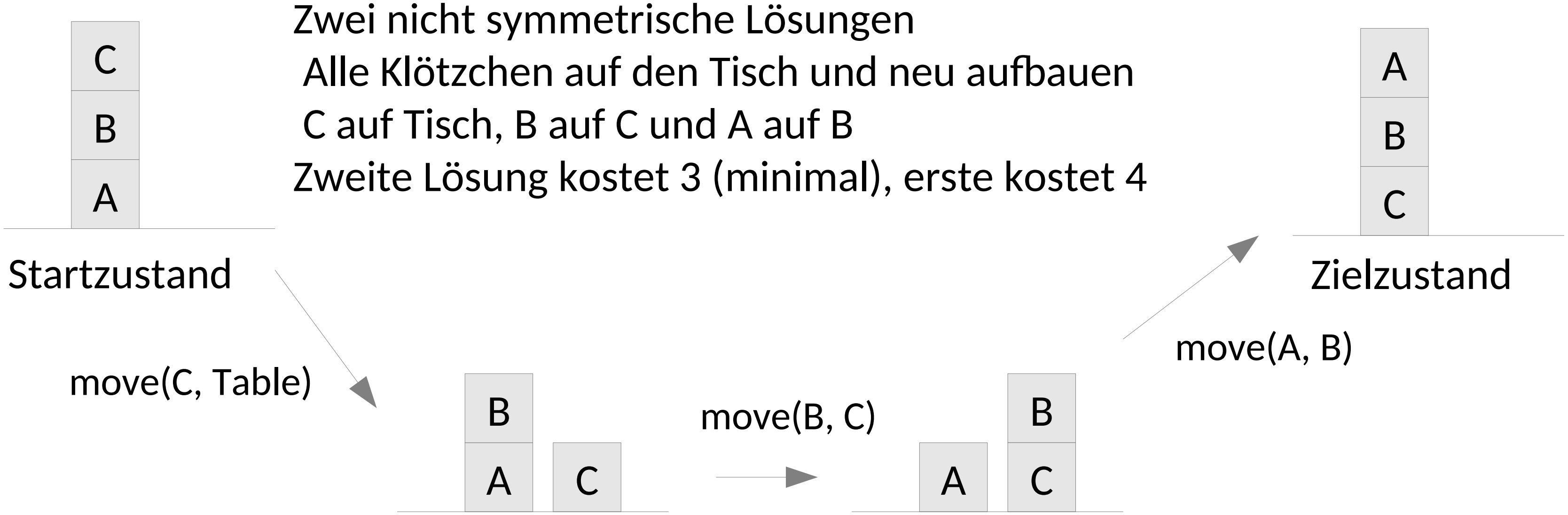
Suchprobleme – Eigenschaften

- Problem: Riesige Suchgraphen, Beispiel PL1
- Ziel: schnell Lösungspfad finden
  - Suchgraph verkleinern
  - „Geschickt“ suchen
- Eigenschaften
  - Zustandsraum durch Operatoren verbunden ist
    Baum oder Graph (Zyklen)
    - Zum Beispiel etwas wegnehmen oder wegnehmen und zurücklegen
  - Äquivalente Zustände
    - Zustände, die dasselbe Problem anders beschreiben
    - Beispiele: Symmetrie, Listen statt Mengen bei beliebiger Reihenfolge
  - Kostenfunktionen, ein Pfad oder optimaler Pfad
- Probleme
  - Zyklen vermeiden, wenn unvermeidlich erkennen
    - Keine Pfade mit Zyklen
  - Äquivalente Zustände vermeiden
    - Problemmodellierung anpassen

Suchprobleme – Beispiel Blocksworld

- Zustandsmenge: Anordnung von Klötzchen auf einem Tisch
- Operatoren: Klötzchen vom Turm nehmen oder drauf legen
- Startzustand, Zielzustand: Zwei beliebige Anordnungen
- Pfadkosten: Uniform, Anzahl der Klötzchenbewegungen

Zwei nicht symmetrische Lösungen
 Alle Klötzchen auf den Tisch und neu aufbauen
 C auf Tisch, B auf C und A auf B
Zweite Lösung kostet 3 (minimal), erste kostet 4



Pfad der zweiten Lösung: <move(C, Table), move(B,C), move(A, B)>

Suchprobleme – Beispiel 8-Puzzle

- Zustandsmenge: Anordnung von 8 Kacheln auf 3x3 Feld
- Operatoren: Bewegung von Kacheln auf freies Feld
- Startzustand, Zielzustand: Zwei beliebige Anordnungen
- Pfadkosten: Uniform, Anzahl der Kachelbewegungen

5	4	
6	1	8
7	3	2

Startzustand

NP-vollständig

9!/2 erreichbare Zustände

bei 4x4 ~1.3 Billionen

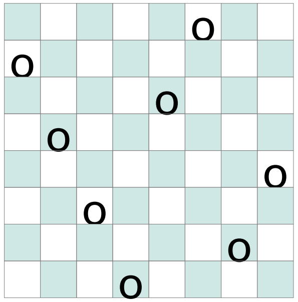
bei 5x5 ~10^{25} –
noch heute schwer
zu lösen

1	2	3
8		4
7	6	5

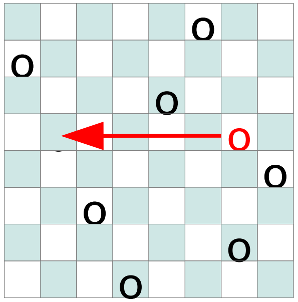
Zielzustand

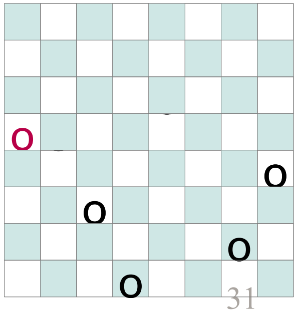
Suchprobleme – N-Damen

- N-Damen Problem
  - N Damen auf einem Schachbrett der Größe NxN verteilen
  - Constraints
    - Keine Dame attackiert eine andere
  - Problemgrößen
    - Standardbeispiel hat Größe 8



- Formulierungen – unterschiedlicher Zustandsraum
  - Vollständig: Ändern
    - Alle Damen auf dem Brett (potentiell sich angreifend)
    - Jeder Operator setzt eine Dame um
    - Größe des Zustandsraums 64*63*…*57 ~ 1.8×10^{14}
  - Inkrementell: Erweitern
    - Je eine Dame in einer Reihe
    - Damen auf unteren Reihen (ohne Lücken)
    - Jeder Operator fügt Dame nicht attackierend hinzu
    - Deutlich kleinerer Suchraum: 2057



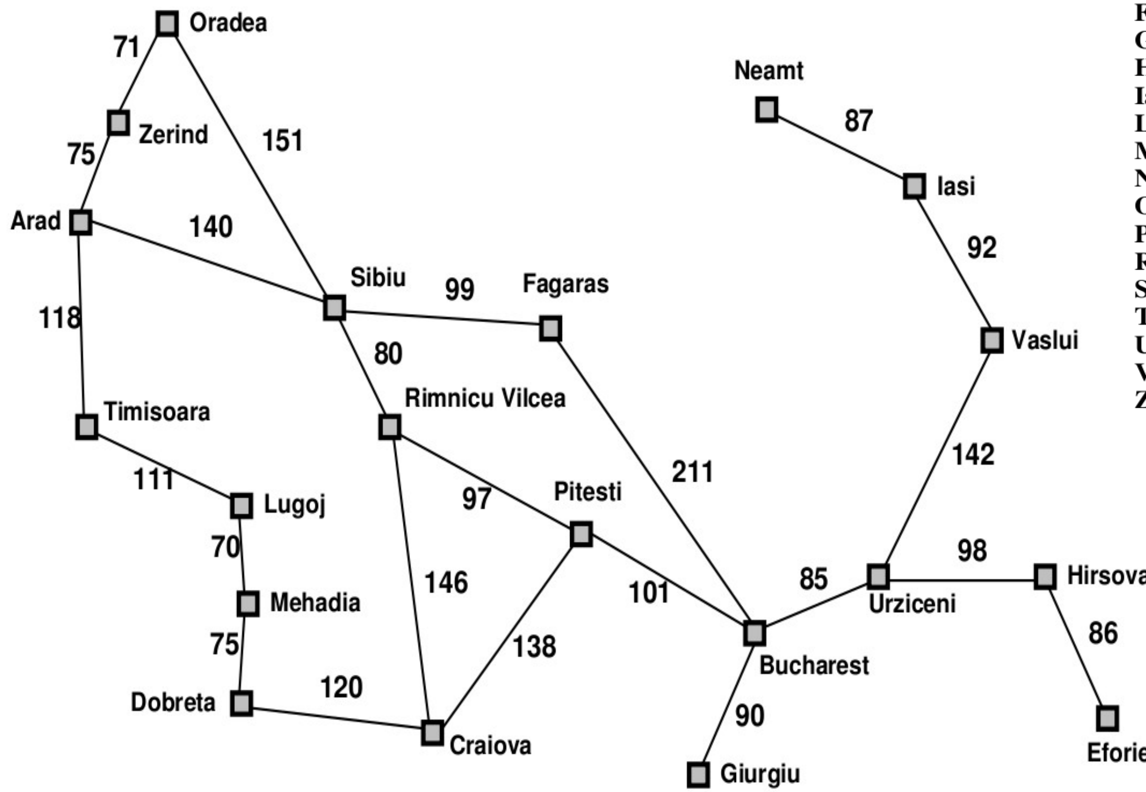


Suchprobleme – Routenplanung

- Zustandsmenge: Städte
- Operatoren: Strassenfahrt Stadt zu Stadt
- Startzustand, Zielzustand: Zwei beliebige Städte
- Pfadkosten: je Übergang unterschiedlich, Strassen-kilometer, Gesamtkosten Gesamt-strassen-kilometer

Luftlinienentfernung zu Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



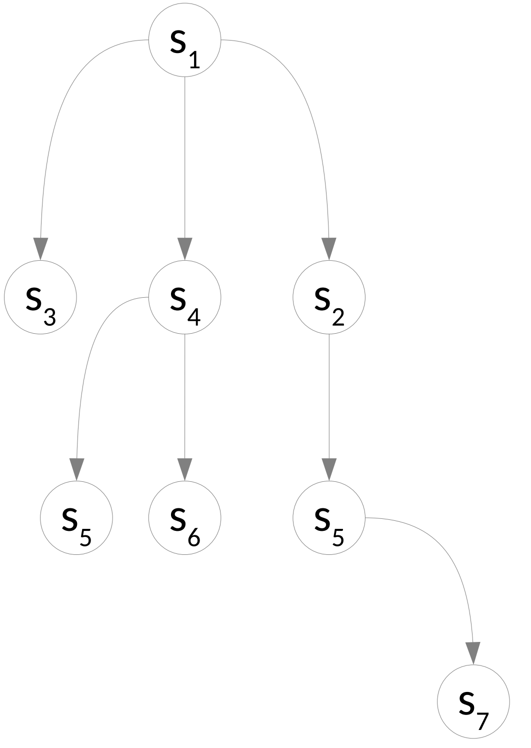
Quelle: Artificial Intelligence, Russel, Norvig, http://www.cs.berkeley.edu/~russell/slides/, für alle weiteren Routenplanerbeispiele

Uninformierte Suche

- Suchen: Finde Pfad vom Startzustand zu einem Zielzustand
- Ansatz: Systematische Suche im Zustandsraum
  - Beginne beim Initialzustand (Vorwärtsgerichtete Suche)
  - Bestimme Nachfolgezustände
    - Mögliche Operatoren anwenden bis Zustand, der Zielbedingung erfüllt, erreicht
  - Suchstrategie
    - Bestimmt die Reihenfolge, in der Nachfolgezustände betrachtet werden
  - Uniformiert
    - Kein Verwenden von zusätzliches Bereichswissen
- Rahmenbedingungen
  - Vollständigkeit?: Wenn eine Lösung existiert, dann wird sie gefunden
  - Optimalität?: Finde die „beste“ Lösung, zum Beispiel geringste Kosten
  - Technische Rahmenbedingungen
    - Zeitkomplexität: Wie lange dauert es?
    - Speicherkomplexität: Wie viele Zustände muss ich gleichzeitig halten?

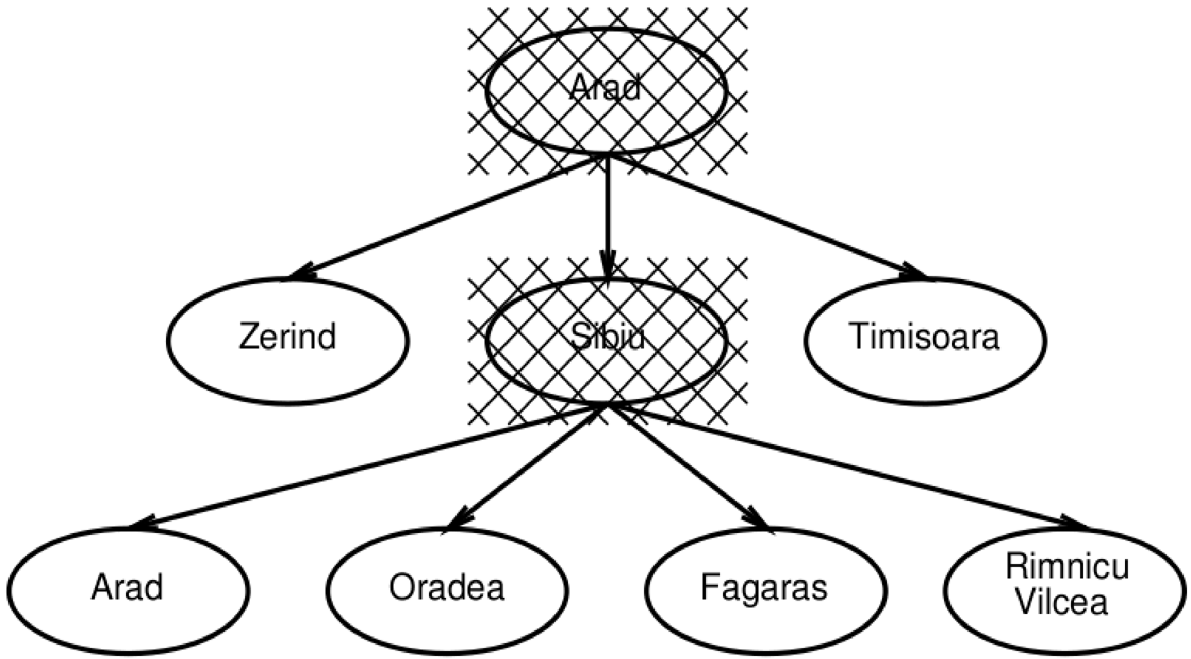
Suchbaum

- Wird während des Suchens implizit oder explizit aufgebaut
- Besteht aus Knoten und Kanten
  - Knoten repräsentieren Zustände
  - Kanten repräsentieren Operatoranwendungen
  - Nachfolgerknoten wurde durch Operator von Vorgängerknoten aus erreicht
  - Wurzel ist Startzustand
  - Blattknoten sind noch nicht *expandiert* oder kein Operator ist mehr anwendbar
  - Zielknoten ist ein Blatt
- Pfadkosten und Tiefe
  - Bei jedem Knoten die Kosten des einen Pfads Wurzel bis Knoten
  - Tiefe ist Pfadkosten bei uniformen Kosten
- [Könnte auch ein Suchgraph sein]



Beispiel – Expansion Routenplanung

- Suche Route von Arad nach Bucharest
  - Arad und Sibiu sind beide expandiert, alle Nachfolger sind im Suchbaum



- Achtung
  - Zustandsraum != Knoten im Suchbaum
  - Es können welche fehlen (die noch nicht erreicht worden sind)
  - Es könnten doppelte vorkommen (ein Graph, im Beispiel Arad)
  - Zustandsraum ist unendlich, man kann auf Strassen hin und her fahren

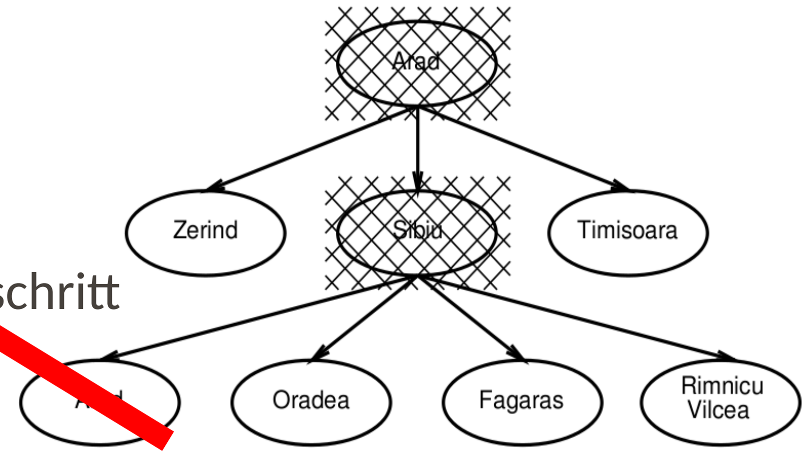
Universeller Suchalgorithmus

- Warteschlange
  - Blätter, noch abzuarbeitende Knoten
  - Meist deque, double ended queue
- Solange noch Knoten abzuarbeiten
  - Hole Knoten aus Queue
  - Expandiere Knoten
    - Alle möglichen Nachfolger durch Menge von Operatoren
    - Vermeide Wiederholungen
  - Füge nur noch nicht erreichte Knoten hinzu
    - Vermeide Endlosschleifen in der Berechnung
  - Strategie – Art des Hinzufügens an Warteschlange
    - Ans Ende (gegenüber der Stelle an der rausgeholt wird), Breitensuche
    - An den Anfang (an der Stelle an der rausgeholt wird), Tiefensuche

```
def search((start, expand, strategy, is_goal)
  queue = [start]
  reached = [start]
  while queue:
    state = queue.pop()
    if is_goal(state):
      return state # eine Lösung
    reached.push(state)
    ex = expand(state, ops)
    newex = [s for s in ex if ex not in reached]
    queue = strategy(queue, newex)
  return None # keine Lösung
```

Vermeide wiederholte Zustände

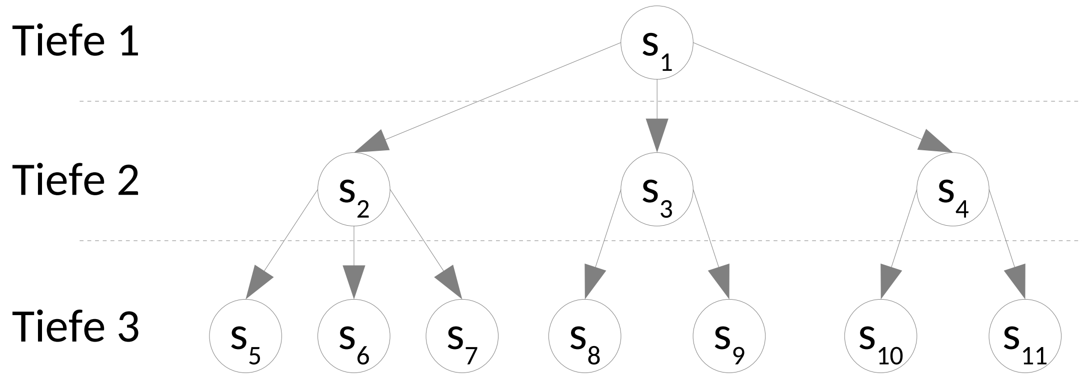
- Wiederholte Zustände
  - Wiederholung falls mehrere Pfade zu einem Zustand
  - Immer dann, wenn Operationen umkehrbar



- Wiederholte Zustände bringen keinen Fortschritt
  - Bei Erkennen Wiederholung kann Suchbaum beschnitten werden
  - Verursacht aber zusätzlichen Berechnungsaufwand
- Mögliche Ansätze
  - Modellerierung so, dass Zustände sich nicht wiederholen, oft reicht es nicht zu einem Zustand zurückkehren von dem man gerade gekommen ist
  - Nicht zu Zustand zurückkehren, der auf Pfad zum Knoten liegt
    - Man muss sich nur Knoten auf dem Pfad merken
  - Nicht zu einem Zustand gehen, der bereits im Suchbaum vorhanden ist
    - Speicherung aller Zustände, hohe Speicherkomplexität
    - Effizient implementierbar (Mengen mit Bäumen oder Hashing)

Breitensuche

- Strategie
  - Erst alle Knoten einer Tiefe, dann das nächsttiefere Level
  - Reihenfolge: 1,2,3,4,5,6….
    - Reihenfolge innerhalb der Tiefe kann variiert werden, z. B. 1, 4,3,2, 11, 10, …
  - Umsetzung: Einfügen der neuen Knoten ans Ende der Warteschlange



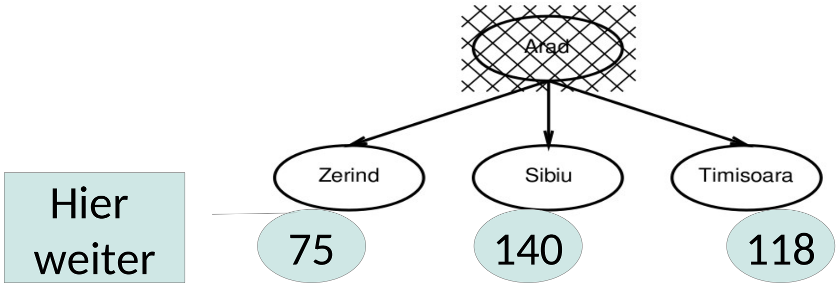

```
def breadthfirst(queue, nodes): return queue+nodes
```

- Eigenschaften
  - Vollständige Strategie, Optimalität bei uniformen Pfadkosten
  - Aufwand: Annahme fester Verzweigungsgrad b (Anzahl Nachfolger) $1 + b + b^2 + ... + b^{d-1} \in O(b^d)$, bei Lösung auf Tiefe d
    Fast alle Knoten im Speicher, Speicheraufwand größtes Problem
  - Beispiel: b=10, 100 Byte/Knoten, 1000 Knoten pro Sekunde

Tiefe	Knoten	Zeit	Speicher
4	11.111	11s	1 MB
8	10^8	31h	11 GB
10	10^{10}	128d	1 TB
12	10^{12}	35y	111 TB
14	10^{14}	3500y	11 PB

Uniforme Kostensuche

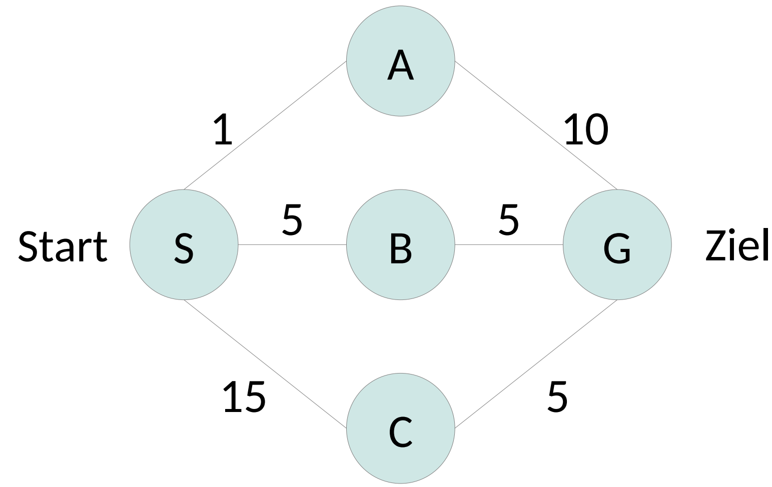
- Strategie
  - Expandiere Knoten mit geringsten Kosten in Warteschlange
    - Ersetzte Warteschlange durch Prioritätswarteschlange oder
    - Füge Knoten nach Kosten aufsteigend sortiert ein
  - Sinnvoll, wenn Kosten je Schritt unterschiedlich sind, zum Beispiel Routenplanung



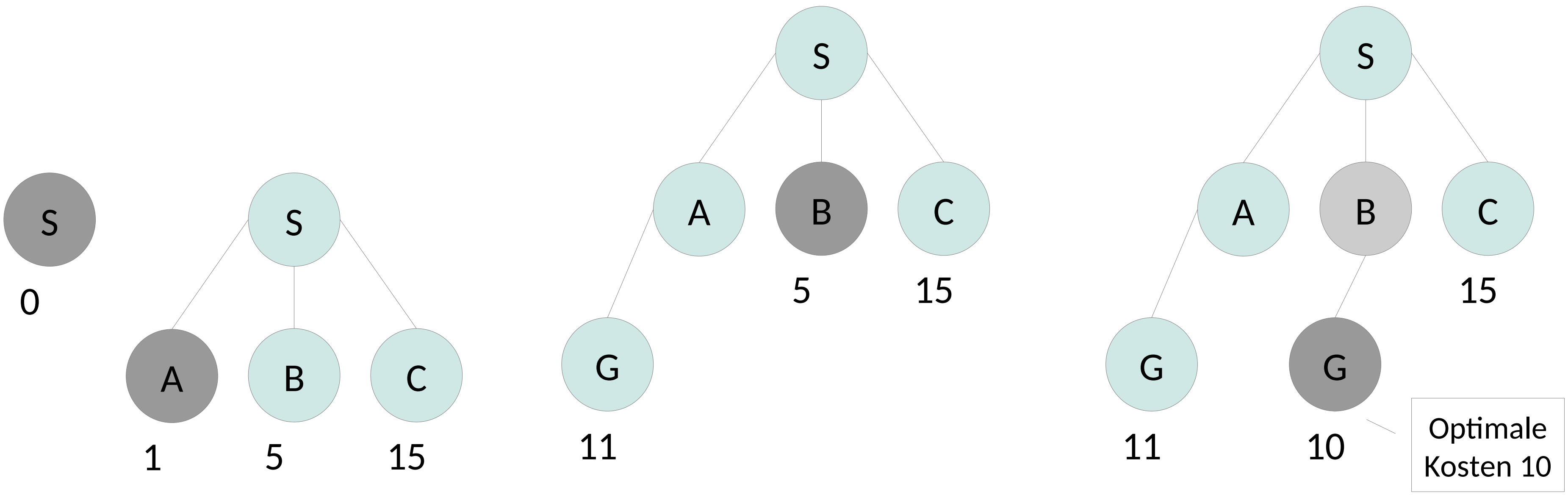
- Eigenschaften
  - Vollständige Strategie
  - Optimalität bei positiven Pfadkosten
    - Beachte, dass erst aufgehört wird, wenn Zielzustand aus Queue kommt, und nicht gleich wenn er in der Menge der expandierten Knoten ist
  - Aufwand wie bei Breitensuche:
    Zeitkomplexität $O(b^d)$
    Speicherkomplexität $O(b^d)$

Uniforme Kostensuche – Beispiel

- Zustandsraum mit Kosten je Operatoranwendunge
  - 5 Zustände: S, A, B, C, G
  - Startzustand S
  - Zielzustand G

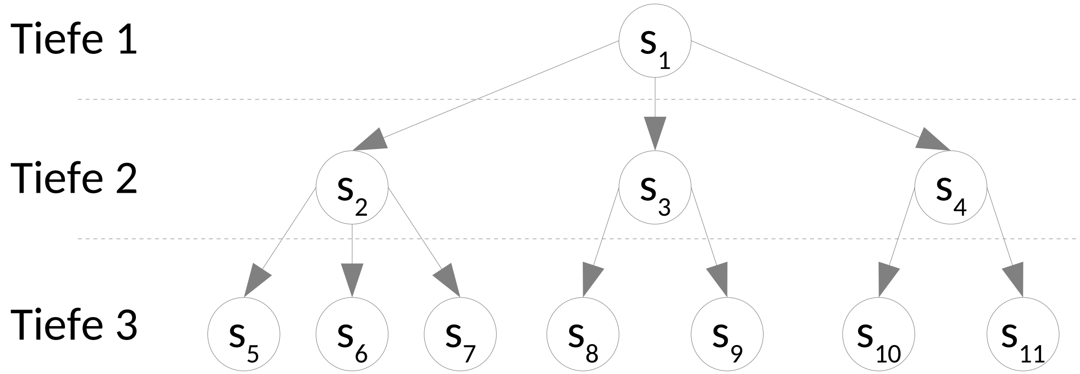


- Suchbaum



Tiefensuche

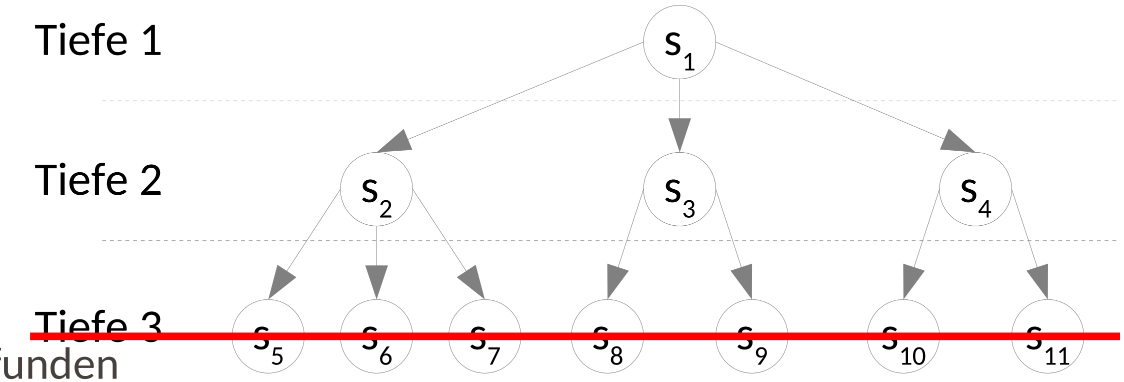
- Strategie
  - Immer den zuletzt hinzugefügten Knoten zuerst
    - Erst in die Tiefe,
      nur bei Misserfolg in die Breite



  - Reihenfolge: 1,2,5,6,7,3,8,9,4, ….
    - Reihenfolge innerhalb der Tiefe kann variiert werden, z. B. 1,4,11,10,2,7, …
  - Umsetzung: Einfügen der neuen Knoten an Anfang der Warteschlange
- Eigenschaften
  - Nicht vollständig bei unendlichen Suchbäumen
    - Kann in unendlichem Pfad stecken bleiben
    - Häufig effektiv, wenn es viele Lösungen gibt
  - Keine Optimalität
  - Aufwand:
    Zeitkomplexität $O(b^d)$
    Speicherkomplexität $O(d*b)$, viel besser als Breitensuche

Beschränkte Tiefensuche

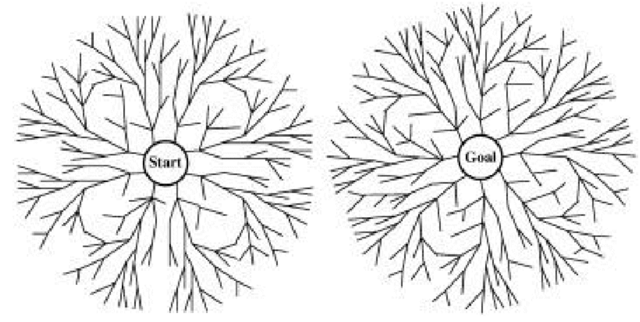
- Idee: Abschneiden des Suchbaums bei Erreichen einer bestimmten Tiefe t



  - Falls Lösung bis Tiefe t existiert, dann wird diese gefunden
  - Speicherkomplexität bleibt mit O(b*t) gering!
  - Keine Optimalität
  - Implementierung: Tiefe mitführen, ab Tiefe t nicht mehr in Warteschlange
- Iterative Tiefensuche (iterative deepending)
  - Beschränkte Tiefensuche schrittweise mit höherer Tiefe wiederholen
  - Kombination der Vorteile von Tiefensuche und Breitensuche
    - Vollständig und optimal bei uniformen Kosten
    - Entgegen der Intuition nicht signifikant mehr Rechenaufwand, gleiche Zeitkomplexität $O(b^d)$
    - Geringer Speicherbedarf O(d*b)
  - Gut geeignet für große Suchräume ohne Tiefenbeschränkung

Bidirektionale Breitensuche

- Idee: Gleichzeitig suchen
  - Suche beginnt sowohl vom Startzustand als auch vom Zielzustand (wenn der eindeutig ist)
  - Ende ist erreicht, wenn sich zwei Suchzweige in der Mitte treffen

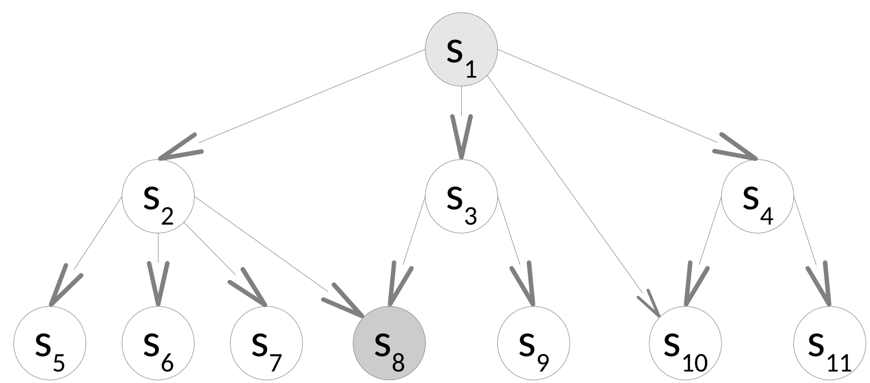


Quelle: Artificial Intelligence, Russel, Norvig

- Voraussetzung
  - Operatoren müssen umkehrbar sein, Vorgängerzustände sind zu bestimmen für Gesamtlösung
- Umsetzung
  - Terminierungsbedingung aufwendiger: Test ob Knoten schon im anderen Suchbaum enthalten
- Eigenschaften
  - Wenn b in beide Richtungen gleich ist, dann Zeitkomplexität: $O(b^{d/2})$, Speicherkomplexität: $O(b^{d/2})$
  - Vollständigkeit und Optimalität bei uniformen Pfadkosten

Vorwärtssuche versus Rückwärtssuche

- Suchrichtung
  - Statt von Start zu Ziel kann man auch von Ziel zu Start suchen
  - Frage der Modellierung
- Voraussetzung
  - Umkehrung der Operatoren muss möglich sein
  - Definition von Umkehroperatoren (o^{-1})
- Welche Richtung wählen?
  - Problemabhängig
  - Falls unterschiedlilcher Verwzeigungsgrad, dann meist geringerer Verzweigungsgrad vorteilhaft
- Beispiel
  - s_1 nach s_8 Verzweigungsgrad 4
  - s_8 nach s_1 Verzweigungsgrad 2



Uninformierte Suchverfahren – Vergleich

Kriterium	Breiten-suche	Uniforme Kostensuche	Tiefen-suche	Iterative Tiefensuche	Beschränkte Tiefensuche	Bidirektionale Breitensuche
Zeit	$O(b^d)$	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^t)$	$O(b^{d/2})$
Speicher	$O(b^d)$	$O(b^d)$	$O(b*m)$	$O(b*d)$	$O(b*t)$	$O(b^{d/2})$
Optimalität	ja[1]	ja	nein	ja[1]	Nein	ja[1]
Vollständig-keit	ja	ja	nein	ja	ja, wenn td	ja

b = Verzweigungsgrad
d = Tiefe der Lösung
m = Maximale Tiefe des Suchbaums
t = Tiefenlimit MaxTiefe

[1] nur bei uniformen Kosten