Declarative Programming

4: blind and informed search of state space, proving as search process
State space search: blocks world

Each configuration of the stacks can be seen as a state of the problem. Alternatively, the state of the gripper can be considered, too. The edges of the graph represent the possible moves.

The state of the blocks world can be represented by a couple of relations between the objects of the world. Alternatively, it can be represented by a list of stacks, where each stack is a list of blocks, the topmost block being the first element.

1.1 Example: Blocksworld

A simple world of blocks on a table. Blocks can be stacked on top of each other, or be placed on the table. A robot gripper can pick one block at a time if there is no other block on top of it.

The objective is to restack the blocks from a start configuration to the goal configuration. The best solution would take the least number of movements.
State space search:
8-puzzle

1.2 Example: 8-Puzzle
- The puzzle has eight square tiles that can be moved one square at a time. I.e. only a tile adjacent to the gap can be moved.
- The objective is to rearrange the position of the tiles to a given pattern in the least number of moves.

1.3 State-space representation in Prolog
- State-space can be represented by the relation $s(X, Y)$ which is true if there is a legal move from node $X$ to node $Y$. We say $Y$ is the successor of $X$.
- In addition, a cost for the transition can be associated with moves: $s(X, Y, \text{Cost})$.
- This relation can be represented in the program explicitly by a set of facts, or by rules:
State space search: graph representation

**state space**

- state = node, state transition = arc
- goal nodes and start nodes
- cost associated with arcs between nodes

**solution**

- path from start to goal node
- optimal if cost over path is minimal

**search algorithms**

- completeness: will a solution always be found if there is one?
- optimality: will highest-quality solution be found when there are several?
- efficiency: runtime and memory requirements
- blind vs informed: does quality of partial solutions steer process?
State space search:
Prolog skeleton for search algorithms

\[ \text{search}(\text{Agenda}, \text{Goal}):= \]
\[ \text{next}(\text{Agenda}, \text{Goal}, \text{Rest}), \]
\[ \text{goal} \text{(Goal)}. \]

\[ \text{search}(\text{Agenda}, \text{Goal}):= \]
\[ \text{next}(\text{Agenda}, \text{Current}, \text{Rest}), \]
\[ \text{children}(\text{Current}, \text{Children}), \]
\[ \text{add}(\text{Children}, \text{Rest}, \text{NewAgenda}), \]
\[ \text{search}(\text{NewAgenda}, \text{Goal}). \]

reached, but untested states

succeeds if the goal state Goal can be reached from a state on the Agenda

goal state for which \text{goal}(\text{Goal}) succeeds

selects a candidate state from the Agenda

expands the current state
State space search: depth-first search

next/3 implemented by taking first element of list

search_df([Goal|Rest],Goal):-
goal(Goal).

search_df([Current|Rest],Goal):-
children(Current,Children),
append(Children,Rest,NewAgenda),
search_df(NewAgenda,Goal).

children(Node,Children):-
findall(C,arc(Node,C),Children).

first-in, last-out agenda treated as a stack

add/3 implemented by prepending children of first element on agenda to the remainder of the agenda

arc(1,2). arc(1,8). arc(1,6).
arc(2,7). arc(2,12). arc(2,4).
arc(12,9). arc(12,15). arc(6,3).
arc(6,11). arc(11,0). arc(11,5).
State space search:
depth-first search with paths

keep path to node on agenda, rather than node

children([Node|RestOfPath],Children):-  
    findall([Child,Node|RestOfPath],arc(Node,Child),Children).

?- search_df([[initial_node]],PathToGoal).

only requires a change to children/3
AND
way search_df/2 is called
State space search: depth-first search with loop detection

keep list of visited nodes

search_df_loop([Goal|Rest], Visited, Goal):-
  goal(Goal).
search_df_loop([Current|Rest], Visited, Goal):-
  children(Current, Children),
  add_df(Children, Rest, Visited, NewAgenda),
  search_df_loop(NewAgenda, [Current|Visited], Goal).

add_df([], Agenda, Visited, Agenda).
add_df([Child|Rest], OldAgenda, Visited, [Child|NewAgenda]):-
  not(element(Child, OldAgenda)),
  not(element(Child, Visited)),
  add_df(Rest, OldAgenda, Visited, NewAgenda).
add_df([Child|Rest], OldAgenda, Visited, NewAgenda):-
  element(Child, OldAgenda),
  add_df(Rest, OldAgenda, Visited, NewAgenda).
add_df([Child|Rest], OldAgenda, Visited, NewAgenda):-
  element(Child, Visited),
  add_df(Rest, OldAgenda, Visited, NewAgenda).

add current node to list of visited nodes

do not add a child if it’s already on the agenda

do not add already visited children
State space search: depth-first search using Prolog stack

**vanilla**

```
search_df(Goal, Goal):-
    goal(Goal).
search_df(CurrentNode, Goal):-
    arc(CurrentNode, Child),
    search_df(Child, Goal).
```

**depth bounded**

```
search_bd(Depth, Goal, Goal):-
    goal(Goal).
search_bd(Depth, CurrentNode, Goal):-
    Depth > 0, 
    NewDepth is Depth-1,
    arc(CurrentNode, Child),
    search_bd(NewDepth, Child, Goal).
?- search_df(10, initial_node, Goal).
```

**iterative deepening**

```
search_id(CurrentNode, Goal):-
    search_id(1, CurrentNode, Goal).
search_id(Depth, CurrentNode, Goal):-
    search_bd(Depth, CurrentNode, Goal).
search_id(Depth, CurrentNode, Goal):-
    NewDepth is Depth+1,
    search_id(NewDepth, CurrentNode, Goal).
```

- Use Prolog call stack as agenda
- Might loop on cycles
- Do not exceed depth threshold while searching
- Always halts, but no solutions beyond threshold
- Increase depth bound on each iteration
- Less memory than BFS
- Complete and solutions on, but upper parts of search space not that bad for full trees: number of nodes at a single level is smaller than all nodes above it
State space search: breadth-first search

next/3 implemented by taking first element of list

first-in, first-out agenda treated as a queue

add/3 implemented by appending children of first element on agenda to the remainder of the agenda

search\_bf([Goal|Rest],Goal):- goal(Goal).
search\_bf([Current|Rest],Goal):- children(Current,Children), append(Rest,Children,NewAgenda), search\_bf(NewAgenda,Goal).

children(Node,Children):- findall(C, arc(Node,C), Children).
State space search: 
dfs vs bfs

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
<th>Completeness</th>
<th>Shortest Solution Path</th>
<th>Best Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>breadth-first</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
</tr>
<tr>
<td>depth-first</td>
<td>$b^m$</td>
<td>$bm$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
</tr>
<tr>
<td>depth-limited</td>
<td>$b^l$</td>
<td>$bl$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
</tr>
<tr>
<td>iterative deepening</td>
<td>$b^d$</td>
<td>$bd$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
<td>$\sqrt{}$</td>
</tr>
</tbody>
</table>

$l$ = depth-limit, $b$ = branching factor of search space, $d$ = depth of search space, $m$ = depth of shortest path solution

- Spirals away from start node, # candidate paths to be remembered grows exponentially with depth.
- Might be second child of root node.
State space search: water jugs problem

- fill a jug from the pool
- empty a jug into the pool
- pour one jug into another until one poured from is empty or the one poured into is full

operations

4L in a jug

goal

operations

[The Art of Prolog, Sterling and Shapiro]
State space search: implementing the search

as a generic algorithm for state space problems

visited states

sequence of transitions to reach goal from current state

until now, we only had unnamed arcs

multiple named transitions out of a state

solve_dfs(State,History,[]) :-
  final_state(State).
solve_dfs(State,History,[Move|Moves]) :-
  move(State,Move),
  update(State,Move,State1),
  legal(State1),
  not(member(State1,History)),
  solve_dfs(State1,[State1|History],Moves).

test_dfs(Problem,Moves) :-
  initial_state(Problem,State),
  solve_dfs(State,[State],Moves).

[The Art of Prolog, Sterling and Shapiro]
State space search: encoding water jugs problem

starting and goal states

\begin{verbatim}
initial_state(jugs,jugs(0,0)).
final_state(jugs(4,V2)).
final_state(jugs(V1,4)).
\end{verbatim}

possible transitions out of a state

\begin{verbatim}
moves(jugs(V1,V2),fill(1)).
moves(jugs(V1,V2),fill(2)).
moves(jugs(V1,V2),empty(1)) :- V1>0.
moves(jugs(V1,V2),empty(2)) :- V2>0.
moves(jugs(V1,V2),transfer(2,1)).
moves(jugs(V1,V2),transfer(1,2)).
\end{verbatim}

empty first jug (1), but only if it still contains water (C1)
State space search: encoding water jugs problem

states a transition can lead to

update(jugs(V1,V2),fill(1),jugs(C1,V2)) :-
   capacity(1,C1).
update(jugs(V1,V2),fill(2),jugs(V1,C2)) :-
   capacity(2,C2).
update(jugs(V1,V2),empty(1),jugs(0,V2)).
update(jugs(V1,V2),empty(2),jugs(V1,0)).
update(jugs(V1,V2),transfer(2,1),jugs(W1,W2)) :-
   capacity(1,C1),
   Liquid is V1 + V2,
   Excess is Liquid - C1,
   adjust(Liquid,Excess,W1,W2).
update(jugs(V1,V2),transfer(1,2),jugs(W1,W2)) :-
   capacity(2,C2),
   Liquid is V1 + V2,
   Excess is Liquid - C2,
   adjust(Liquid,Excess,W2,W1).

adjust(Liquid, Excess,Liquid,0) :- Excess =< 0.
adjust(Liquid,Excess,V,Excess) :-
   Excess > 0,
   V is Liquid - Excess.

capacity(j1,8).
capacity(j2,5).
legal(jugs(C1,C2)).
Proving as a search process: 

df agenda-based meta-interpreter

prove(true):- !.
prove((A,B)):-
  !,
  clause(A,C),
  conj_append(C,B,D),
  prove(D).
prove(A):-
  clause(A,B),
  prove(B).

prove_df_a(Goal) :-
  prove_df_a([Goal]).
prove_df_a([true|Agenda]).
prove_df_a([[(A,B)|Agenda]]) :-
  !,
  findall(D,(clause(A,C),conj_append(C,B,D)),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
prove_df_a([A|Agenda]) :-
  findall(B,clause(A,B),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).

conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):-
  not(X=true),
  not(X=(One,TheOther)).
conj_append((X,Xs),Ys,(X,Zs)):-
  conj_append(Xs,Ys,Zs).

true: empty conjunctions
single term: singleton conjunction

swapping arguments of append/3 turns this into a breadth-first meta-interpreter!
Proving as a search process: 
**bf agenda-based meta-interpreter**

```prolog
prove_bf(Goal):-
    prove_bf_a([a(Goal,Goal)],Goal).
prove_bf_a([a(true,Goal)|Agenda],Goal).
prove_bf_a([a((A,B),G)|Agenda],Goal):-!,
    findall(a(D,G),(clause(A,C),conj_append(C,B,D)),Children),
    append(Agenda,Children,NewAgenda),
    prove_bf_a(NewAgenda,Goal).
prove_bf_a([a(A,G)|Agenda],Goal):-
    findall(a(B,G),clause(A,B),Children),
    append(Agenda,Children,NewAgenda),
    prove_bf_a(NewAgenda,Goal).
```

**problem:**
findall(Term,Goal,List) creates new variables in the instantiation of Term for the unbound variables in answers to Goal

```prolog
?- findall(Body,clause(foo(Z),Body),Bodies).
Bodies = [bar(_G336)].
```

**trick:**
store a(Literals,OriginalGoal) on agenda where OriginalGoal is a copy of the Goal

Goal will be instantiated with the correct answer substitutions.

This time with answer substitution.

breadth-first
### Proving as a search process:
*forward vs backward chaining of if-then rules*

<table>
<thead>
<tr>
<th>Backward chaining</th>
<th>Forward chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>from head to body</td>
<td>from body to head</td>
</tr>
</tbody>
</table>

- Search starts from where we want to be towards where we are
- e.g. Prolog query answering

- Search starts from where we are to where we want to be
- e.g. model construction

What’s more efficient depends on structure of search space (cf. discussion on practical uses of var)
Proving as a search process:
**forward chaining - bottom-up model construction**

```
model(M):- model([],M).
model(M0,M):-
is_violated(Head,M0),!,
disj_element(L,Head),
model([L|M0],M).
model(M,M).

is_violated(H,M) :-
cl((H:-B)),
satisfied_body(B,M),
not(satisfied_head(H,M)).
```

- **model of clauses defined by cl/1**
- **grounds literal from head**
- **no more violated clauses (note the !)**
- **grounds literal from body**
- **add a literal from the head of a violated clause to the current model**
- **a violated clause: body is true in the current model, but the head not**
Proving as a search process:
forward chaining - auxiliaries

body is a conjunction of literals

satisfied_body(true,M).
satisfied_body(A,M) :-
    element(A,M).
satisfied_body((A,B),M) :-
    element(A,M),
    satisfied_body(B,M).

satisfied_head(A,M):-
    element(A,M).
satisfied_head((A;B),M) :-
    element(A,M),
    satisfied_head(B,M).

, and ; are right-associative operators:
a;b;c=(a,;(b,c))

disj_element(X,X):-
    not(X=false),
    not(X=(One;TheOther)).
disj_element(X, (X;Ys)):-
    disj_element(X, (Y;Ys)):-
    disj_element(X, Ys).

disj_element(X, Xs):-
    not(X=false),
    not(X=(One;TheOther)).

disj_element(X, (X;Ys)):-
    disj_element(X, (Y;Ys)):-
    disj_element(X, Ys).

false = empty disjunction

single disjunct
Proving as a search process: forward chaining - example

\[
\text{cl(}\{(\text{married}(X) ; \text{bachelor}(X) : -\text{man}(X), \text{adult}(X))\}.
\text{cl(}\{(\text{has}_{-}\text{wife}(X) : -\text{married}(X), \text{man}(X))\}.
\text{cl(}\{(\text{man}(\text{paul}) : -\text{true})\}.
\text{cl(}\{(\text{adult}(\text{paul}) : -\text{true})\}.
\]

\[\text{?- model(M) M = [}\text{has}_{-}\text{wife(}\text{paul}\text{)}, \text{married(}\text{paul}\text{)}, \text{adult(}\text{paul}\text{), man(}\text{paul}\text{)]; M = [}\text{bachelor(}\text{paul}\text{), adult(}\text{paul}\text{), man(}\text{paul}\text{)]}\]

two minimal models as there is a disjunction in the head
Proving as a search process: forward chaining - range-restricted clauses

Our simple forward chainer cannot construct a model for following clauses:

- `cl((man(X);woman(X):-true)).`
- `cl((false:-man(maria))).`
- `cl((false:-woman(peter))).`  

an unground `man(X)` will be added to the model, which leads to the second clause being violated—which cannot be solved as it has an empty head.

works only for clauses for which grounding the body also grounds the head.

add literal to first clause, to enumerate possible values of `X`

- `cl((man(X);woman(X):-person(X))).`
- `cl((person(maria):-true)).`
- `cl((person(peter):-true)).`
- `cl((false:-man(maria))).`
- `cl((false:-woman(peter))).`

?- model(M)  
M = [man(peter),person(peter),woman(maria),person(maria)]

range-restricted clause: all variables in head also occur in body

can be ensured by adding predicates that quantify over each variable’s domain.
Proving as a search process: forward chaining - subsets of infinite models

cl((append([],Y,Y):-list(Y))).
cl((append([X|Xs],Ys,[X|Zs]):-thing(X),append(Xs,Ys,Zs))).
cl((list([]):-true)).
cl((list([X|Y]):-thing(X),list(Y))).
cl((thing(a):-true)).
cl((thing(b):-true)).
cl((thing(c):-true)).

model_d(D,M):-
    model_d(D, [], M).

model_d(0,M,M).
model_d(D,M0,M):-
    D>0,
    D1 is D-1,
    findall(H,is_violated(H,M0),Heads),
    satisfy_clauses(Heads,M0,M1),
    model_d(D1,M1,M).

satisfy_clauses([],M,M).

satisfy_clauses([H|Hs],M0,M):-
    disj_element(L,H),
    satisfy_clauses(Hs, [L|M0], M).