```
/* -*- Mode: Prolog -*-

   Demonstration of six distinct search methods to generate
   an executable plan for getting from A to B.

   Operates in a simplififed blocks world environment similar
   to that of Winograd's SHRDLU agent.

   Michael E. Sparks, 25 Nov 2020

   SAMPLE USAGE:

%%% 1) Find path through the Sussman Anomaly:

?- init_sussman(Init), goal_sussman(Goals).
Init = [clear(b), on(b, 1), clear(2), clear(c), on(c, a), on(a, 3)],
Goals = [clear(a), on(a, b), on(b, c), on(c, 1), clear(2), clear(3)].

?- init_sussman(Init), goal_sussman(Goals),
|    time(soln_ME(Init,Goals,Plan)), !, fail.

  1  move(b,1,2)
  2  move(c,a,1)
  3  move(b,2,c)
  4  move(a,3,b)

% 636,134 inferences, 0.103 CPU in 0.103 seconds (100% CPU, 6159272 Lips)
false.

?- init_sussman(Init), goal_sussman(Goals),
|    time(soln_MEGR(Init,Goals,Plan)), !, fail.

  1  move(b,1,2)
  2  move(c,a,1)
  3  move(b,2,c)
  4  move(a,3,b)

% 447,685 inferences, 0.073 CPU in 0.073 seconds (100% CPU, 6138735 Lips)
false.

?- init_sussman(Init), goal_sussman(Goals),
|    time(soln_DFSidMEGR(Init,Goals,Plan)), !, fail.
```

```
  1  move(b,1,2) -> [on(c,a),clear(1),on(b,2),clear(c),on(a,3),clear(b)]
  2  move(c,a,1) -> [on(b,2),clear(c),on(a,3),clear(b),clear(a),on(c,1)]
  3  move(b,2,c) -> [on(a,3),clear(b),clear(a),on(b,c),on(c,1),clear(2)]
  4  move(a,3,b) -> [clear(a),on(a,b),on(b,c),on(c,1),clear(2),clear(3)]

% 581,507 inferences, 0.088 CPU in 0.088 seconds (100% CPU, 6629881 Lips)
false.

?- goal_sussman(G), init_sussman(I),
|     time(soln_BFS_backward(I,G,P)), !, fail.

  1  [clear(2),clear(b),clear(c),on(a,3),on(b,1),on(c,a)]&move(b,1,2)
  2  [clear(1),clear(b),clear(c),on(a,3),on(b,2),on(c,a)]&move(c,a,1)
  3  [clear(a),clear(b),clear(c),on(a,3),on(b,2),on(c,1)]&move(b,2,c)
  4  [clear(2),clear(a),clear(b),on(a,3),on(b,c),on(c,1)]&move(a,3,b)
  5  [clear(2),clear(3),clear(a),on(a,b),on(b,c),on(c,1)]&none

% 268,426 inferences, 0.047 CPU in 0.047 seconds (100% CPU, 5768678 Lips)
false.

?- goal_sussman(G), init_sussman(I),
|     time(soln_BFS_forward(I,G,P)), !, fail.

  1  [clear(2),clear(b),clear(c),on(a,3),on(b,1),on(c,a)]&move(b,1,2)
  2  [clear(1),clear(b),clear(c),on(a,3),on(b,2),on(c,a)]&move(c,a,1)
  3  [clear(a),clear(b),clear(c),on(a,3),on(b,2),on(c,1)]&move(b,2,c)
  4  [clear(2),clear(a),clear(b),on(a,3),on(b,c),on(c,1)]&move(a,3,b)
  5  [clear(2),clear(3),clear(a),on(a,b),on(b,c),on(c,1)]&none

% 162,434 inferences, 0.028 CPU in 0.028 seconds (100% CPU, 5797362 Lips)
false.

?- goal_sussman(G), init_sussman(I), time(soln_BFS_bidir(I,G,P)).

  1  [clear(b),on(b,1),clear(2),clear(c),on(c,a),on(a,3)]&move(b,1,2)
  2  [clear(1),clear(b),clear(c),on(a,3),on(b,2),on(c,a)]&move(c,a,1)
  3  [clear(a),clear(b),clear(c),on(a,3),on(b,2),on(c,1)]&move(b,2,c)
  4  [clear(2),clear(a),clear(b),on(a,3),on(b,c),on(c,1)]&move(a,3,b)
  5  [clear(a),on(a,b),on(b,c),on(c,1),clear(2),clear(3)]&none

% 21,374 inferences, 0.008 CPU in 0.008 seconds (100% CPU, 2680793 Lips)
```

```
false.


%%% 2) Solve a simple blocks world puzzle (3 blocks, 3 places):

?- init_state(Init), goal_state(Goals),
|     write("means-ends:"), nl, soln_ME(Init,Goals,Plan),
|     write("means-ends with goal regression:"),
|     nl, soln_MEGR(Init,Goals,Plan),
|     write("DFSid using means-ends with goal regression:"),
|     nl, soln_DFSidMEGR(Init,Goals,Plan), !, fail.
means-ends:

  1  move(c,a,2)
  2  move(b,3,c)
  3  move(a,1,b)

means-ends with goal regression:

  1  move(c,a,2)
  2  move(b,3,c)
  3  move(a,1,b)

DFSid using means-ends with goal regression:

  1  move(c,a,2) -> [on(b,3),clear(c),on(a,1),clear(b),clear(a),on(c,2)]
  2  move(b,3,c) -> [on(a,1),clear(b),clear(a),on(b,c),on(c,2),clear(3)]
  3  move(a,1,b) -> [clear(1),clear(a),on(a,b),on(b,c),on(c,2),clear(3)]

false.


%%% 3) Solve a less simple blocks world problem (4 blocks, 3 places)

?- assertz(block(d)), init_state1(I), goal_state1(G),
|     time(soln_BFS_backward(I,G,P)), !, fail.

  1  [clear(3),clear(b),clear(d),on(a,2),on(b,c),on(c,1),on(d,a)]&move(d,a,b)
  2  [clear(3),clear(a),clear(d),on(a,2),on(b,c),on(c,1),on(d,b)]&move(a,2,3)
  3  [clear(2),clear(a),clear(d),on(a,3),on(b,c),on(c,1),on(d,b)]&move(d,b,2)
  4  [clear(a),clear(b),clear(d),on(a,3),on(b,c),on(c,1),on(d,2)]&move(b,c,a)
  5  [clear(b),clear(c),clear(d),on(a,3),on(b,a),on(c,1),on(d,2)]&move(c,1,d)
```

```
  6  [clear(1),clear(b),clear(c),on(a,3),on(b,a),on(c,d),on(d,2)]&move(b,a,c)
  7  [clear(1),clear(a),clear(b),on(a,3),on(b,c),on(c,d),on(d,2)]&move(a,3,b)
  8  [clear(1),clear(3),clear(a),on(a,b),on(b,c),on(c,d),on(d,2)]&none

% 16,805,005 inferences, 2.584 CPU in 2.584 seconds (100% CPU, 6504050 Lips)
false.

?- init_state1(I), goal_state1(G),
|     time(soln_BFS_forward(I,G,P)), !, fail.

  1  [clear(3),clear(b),clear(d),on(a,2),on(b,c),on(c,1),on(d,a)]&move(d,a,b)
  2  [clear(3),clear(a),clear(d),on(a,2),on(b,c),on(c,1),on(d,b)]&move(a,2,3)
  3  [clear(2),clear(a),clear(d),on(a,3),on(b,c),on(c,1),on(d,b)]&move(d,b,2)
  4  [clear(a),clear(b),clear(d),on(a,3),on(b,c),on(c,1),on(d,2)]&move(b,c,a)
  5  [clear(b),clear(c),clear(d),on(a,3),on(b,a),on(c,1),on(d,2)]&move(c,1,d)
  6  [clear(1),clear(b),clear(c),on(a,3),on(b,a),on(c,d),on(d,2)]&move(b,a,c)
  7  [clear(1),clear(a),clear(b),on(a,3),on(b,c),on(c,d),on(d,2)]&move(a,3,b)
  8  [clear(1),clear(3),clear(a),on(a,b),on(b,c),on(c,d),on(d,2)]&none

% 17,785,102 inferences, 2.542 CPU in 2.542 seconds (100% CPU, 6996214 Lips)
false.

?- init_state1(I), goal_state1(G), time(soln_BFS_bidir(I,G,P)).

  1  [clear(b),on(b,c),on(c,1),clear(d),on(d,a),on(a,2),clear(3)]&move(d,a,b)
  2  [clear(3),clear(a),clear(d),on(a,2),on(b,c),on(c,1),on(d,b)]&move(a,2,3)
  3  [clear(2),clear(a),clear(d),on(a,3),on(b,c),on(c,1),on(d,b)]&move(d,b,2)
  4  [clear(a),clear(b),clear(d),on(a,3),on(b,c),on(c,1),on(d,2)]&move(b,c,a)
  5  [clear(b),clear(c),clear(d),on(a,3),on(b,a),on(c,1),on(d,2)]&move(c,1,d)
  6  [clear(1),clear(b),clear(c),on(a,3),on(b,a),on(c,d),on(d,2)]&move(b,a,c)
  7  [clear(1),clear(a),clear(b),on(a,3),on(b,c),on(c,d),on(d,2)]&move(a,3,b)
  8  [clear(1),clear(3),clear(a),on(a,b),on(b,c),on(c,d),on(d,2)]&none

% 337,949 inferences, 0.061 CPU in 0.061 seconds (100% CPU, 5502505 Lips)
false.


%%% 4) Solve a less simple blocks world problem (5 blocks, 3 places)
       (Consider increasing SWI Prolog's stack space via --stack-limit:
         $ swipl --stack-limit=32g -s blox_wurld.pro )

   Not only does the bidirectional BFS "smoke" the competition,
```

```
   but it also finds a second, equally optimal solution. In contrast
   to the Sussman Anomaly, backward chaining performed much better
   than forward chaining.

?- goal_state2(G).
G = [clear(a), on(a, b), on(b, c), on(c, d), on(d, e), on(e, 1), clear(2), clear(3)].

?- init_state2(Init), goal_state2(Goals), assertz(block(e)),
|     time(soln_DFSidMEGR(Init,Goals,Plan)), !, fail.

  1  move(a,d,b) -> [on(e,c),clear(d),on(c,1),clear(a),on(d,2),clear(e),on(a,b),on(b,3)]
  2  move(e,c,d) -> [on(c,1),clear(a),on(e,d),on(d,2),clear(e),on(a,b),on(b,3),clear(c)]
  3  move(c,1,a) -> [on(e,d),clear(1),on(d,2),clear(e),on(c,a),on(a,b),on(b,3),clear(c)]
  4  move(e,d,1) -> [on(d,2),clear(e),on(c,a),clear(d),on(a,b),on(b,3),clear(c),on(e,1)]
  5  move(d,2,e) -> [on(c,a),clear(d),on(a,b),clear(2),on(b,3),clear(c),on(d,e),on(e,1)]
  6  move(c,a,d) -> [on(a,b),clear(2),on(b,3),clear(c),clear(a),on(c,d),on(d,e),on(e,1)]
  7  move(a,b,2) -> [on(b,3),clear(c),on(a,2),clear(b),clear(a),on(c,d),on(d,e),on(e,1)]
  8  move(b,3,c) -> [on(a,2),clear(b),clear(a),on(b,c),on(c,d),on(d,e),on(e,1),clear(3)]
  9  move(a,2,b) -> [clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),clear(2),clear(3)]

% 16,256,690,298 inferences, 2452.890 CPU in 2452.991 seconds (100% CPU, 6627567 Lips)
false.

?- init_state2(I), goal_state2(G),
|     time(soln_BFS_backward(I,G,P)), !, fail.

  1  [clear(a),clear(b),clear(e),on(a,d),on(b,3),on(c,1),on(d,2),on(e,c)]&move(a,d,b)
  2  [clear(a),clear(d),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,c)]&move(e,c,d)
  3  [clear(a),clear(c),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,d)]&move(c,1,a)
  4  [clear(1),clear(c),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,d)]&move(e,d,1)
  5  [clear(c),clear(d),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,1)]&move(d,2,e)
  6  [clear(2),clear(c),clear(d),on(a,b),on(b,3),on(c,a),on(d,e),on(e,1)]&move(c,a,d)
  7  [clear(2),clear(a),clear(c),on(a,b),on(b,3),on(c,d),on(d,e),on(e,1)]&move(a,b,2)
  8  [clear(a),clear(b),clear(c),on(a,2),on(b,3),on(c,d),on(d,e),on(e,1)]&move(b,3,c)
  9  [clear(3),clear(a),clear(b),on(a,2),on(b,c),on(c,d),on(d,e),on(e,1)]&move(a,2,b)
  10  [clear(2),clear(3),clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1)]&none

% 4,124,212,616 inferences, 533.958 CPU in 534.044 seconds (100% CPU, 7723848 Lips)
false.

?- init_state2(I), goal_state2(G), retractall(path_BFS(_)),
|     time(soln_BFS_forward(I,G,P)), !, fail.
```

```
  1   [clear(a),clear(b),clear(e),on(a,d),on(b,3),on(c,1),on(d,2),on(e,c)]&move(a,d,b)
  2   [clear(a),clear(d),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,c)]&move(e,c,d)
  3   [clear(a),clear(c),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,d)]&move(c,1,a)
  4   [clear(1),clear(c),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,d)]&move(e,d,1)
  5   [clear(c),clear(d),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,1)]&move(d,2,e)
  6   [clear(2),clear(c),clear(d),on(a,b),on(b,3),on(c,a),on(d,e),on(e,1)]&move(c,a,d)
  7   [clear(2),clear(a),clear(c),on(a,b),on(b,3),on(c,d),on(d,e),on(e,1)]&move(a,b,2)
  8   [clear(a),clear(b),clear(c),on(a,2),on(b,3),on(c,d),on(d,e),on(e,1)]&move(b,3,c)
  9   [clear(3),clear(a),clear(b),on(a,2),on(b,c),on(c,d),on(d,e),on(e,1)]&move(a,2,b)
  10  [clear(2),clear(3),clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1)]&none

% 31,689,751,887 inferences, 3825.759 CPU in 3825.946 seconds (100% CPU, 8283258 Lips)
false.

?- init_state2(I), goal_state2(G), time(soln_BFS_bidir(I,G,P)).

  1   [clear(e),on(e,c),on(c,1),clear(a),on(a,d),on(d,2),clear(b),on(b,3)]&move(a,d,b)
  2   [clear(a),clear(d),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,c)]&move(e,c,d)
  3   [clear(a),clear(c),clear(e),on(a,b),on(b,3),on(c,1),on(d,2),on(e,d)]&move(c,1,a)
  4   [clear(1),clear(c),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,d)]&move(e,d,1)
  5   [clear(c),clear(d),clear(e),on(a,b),on(b,3),on(c,a),on(d,2),on(e,1)]&move(d,2,e)
  6   [clear(2),clear(c),clear(d),on(a,b),on(b,3),on(c,a),on(d,e),on(e,1)]&move(c,a,d)
  7   [clear(2),clear(a),clear(c),on(a,b),on(b,3),on(c,d),on(d,e),on(e,1)]&move(a,b,2)
  8   [clear(a),clear(b),clear(c),on(a,2),on(b,3),on(c,d),on(d,e),on(e,1)]&move(b,3,c)
  9   [clear(3),clear(a),clear(b),on(a,2),on(b,c),on(c,d),on(d,e),on(e,1)]&move(a,2,b)
  10  [clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),clear(2),clear(3)]&none


  1   [clear(e),on(e,c),on(c,1),clear(a),on(a,d),on(d,2),clear(b),on(b,3)]&move(e,c,a)
  2   [clear(b),clear(c),clear(e),on(a,d),on(b,3),on(c,1),on(d,2),on(e,a)]&move(c,1,b)
  3   [clear(1),clear(c),clear(e),on(a,d),on(b,3),on(c,b),on(d,2),on(e,a)]&move(e,a,1)
  4   [clear(a),clear(c),clear(e),on(a,d),on(b,3),on(c,b),on(d,2),on(e,1)]&move(a,d,c)
  5   [clear(a),clear(d),clear(e),on(a,c),on(b,3),on(c,b),on(d,2),on(e,1)]&move(d,2,e)
  6   [clear(2),clear(a),clear(d),on(a,c),on(b,3),on(c,b),on(d,e),on(e,1)]&move(a,c,2)
  7   [clear(a),clear(c),clear(d),on(a,2),on(b,3),on(c,b),on(d,e),on(e,1)]&move(c,b,d)
  8   [clear(a),clear(b),clear(c),on(a,2),on(b,3),on(c,d),on(d,e),on(e,1)]&move(b,3,c)
  9   [clear(3),clear(a),clear(b),on(a,2),on(b,c),on(c,d),on(d,e),on(e,1)]&move(a,2,b)
  10  [clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),clear(2),clear(3)]&none

% 2,475,629 inferences, 0.405 CPU in 0.405 seconds (100% CPU, 6110324 Lips)
false.
```

```
%%% 5) A "least simple" blocks world challenge:

?- assertz(block(f)), assertz(block(g)).
true.

?- init_state3(I), goal_state3(G), time(soln_BFS_bidir(I,G,P)).

  1  [clear(e),on(e,c),on(c,1),clear(f),on(f,a),on(a,d),on(d,g),on(g,2),clear(b),on(b,3)]&move(b,3,e)
  2  [clear(3),clear(b),clear(f),on(a,d),on(b,e),on(c,1),on(d,g),on(e,c),on(f,a),on(g,2)]&move(f,a,3)
  3  [clear(a),clear(b),clear(f),on(a,d),on(b,e),on(c,1),on(d,g),on(e,c),on(f,3),on(g,2)]&move(a,d,f)
  4  [clear(a),clear(b),clear(d),on(a,f),on(b,e),on(c,1),on(d,g),on(e,c),on(f,3),on(g,2)]&move(b,e,a)
  5  [clear(b),clear(d),clear(e),on(a,f),on(b,a),on(c,1),on(d,g),on(e,c),on(f,3),on(g,2)]&move(e,c,d)
  6  [clear(b),clear(c),clear(e),on(a,f),on(b,a),on(c,1),on(d,g),on(e,d),on(f,3),on(g,2)]&move(c,1,b)
  7  [clear(1),clear(c),clear(e),on(a,f),on(b,a),on(c,b),on(d,g),on(e,d),on(f,3),on(g,2)]&move(e,d,1)
  8  [clear(c),clear(d),clear(e),on(a,f),on(b,a),on(c,b),on(d,g),on(e,1),on(f,3),on(g,2)]&move(d,g,e)
  9  [clear(c),clear(d),clear(g),on(a,f),on(b,a),on(c,b),on(d,e),on(e,1),on(f,3),on(g,2)]&move(c,b,d)
  10  [clear(b),clear(c),clear(g),on(a,f),on(b,a),on(c,d),on(d,e),on(e,1),on(f,3),on(g,2)]&move(b,a,c)
  11  [clear(a),clear(b),clear(g),on(a,f),on(b,c),on(c,d),on(d,e),on(e,1),on(f,3),on(g,2)]&move(a,f,b)
  12  [clear(a),clear(f),clear(g),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),on(f,3),on(g,2)]&move(f,3,a)
  13  [clear(3),clear(f),clear(g),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),on(f,a),on(g,2)]&move(g,2,3)
  14  [clear(2),clear(f),clear(g),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),on(f,a),on(g,3)]&move(f,a,g)
  15  [clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),clear(2),clear(f),on(f,g),on(g,3)]&none

% 1,608,997,574 inferences, 259.822 CPU in 259.834 seconds (100% CPU, 6192682 Lips)
false.
*/

%%%%% OBJECT-TYPE CONSIDERATIONS:

% the Sussman Anomaly:
goal_sussman([clear(a),on(a,b),on(b,c),on(c,1),clear(2),clear(3)]).
init_sussman([clear(b),on(b,1),clear(2),clear(c),on(c,a),on(a,3)]).

% a simple blocks world challenge:
goal_state([clear(1),clear(a),on(a,b),on(b,c),on(c,2),clear(3)]).
init_state([clear(c),on(c,a),on(a,1),clear(2),clear(b),on(b,3)]).

% a less simple blocks world challenge (must assert block d):
goal_state1([clear(1),clear(3),clear(a),on(a,b),
             on(b,c),on(c,d),on(d,2)]).
```

```
init_state1([clear(b),on(b,c),on(c,1),clear(d),
             on(d,a),on(a,2),clear(3)]).

% another blocks world challenge (must assert blocks d-e):
goal_state2([clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),
             clear(2),clear(3)]).
init_state2([clear(e),on(e,c),on(c,1),clear(a),on(a,d),on(d,2),
             clear(b),on(b,3)]).

% a least simple blocks world challenge (must assert blocks d-g):
goal_state3([clear(a),on(a,b),on(b,c),on(c,d),on(d,e),on(e,1),
             clear(2),clear(f),on(f,g),on(g,3)]).
init_state3([clear(e),on(e,c),on(c,1),clear(f),on(f,a),on(a,d),
             on(d,g),on(g,2),clear(b),on(b,3)]).

% blocks are movable ("personalty")
:- dynamic
       block/1. % we can add/remove blocks at will

block(a).
block(b).
block(c).

% places are static ("realty")
place(1).
place(2).
place(3).

% All objects in this microworld are supportive,
% though it could be extended to include, for instance,
% pyramid-shaped objects (which can only be clear and
% placed on a supportive structure).
supportive(X) :-
    block(X)
    ;
    place(X).

%%%%% VERB-TYPE CONSIDERATIONS:

% predicate establishes prerequisites for the operation
can(move(Block,From,To),[clear(Block),on(Block,From),clear(To)]) :-
    block(Block),
```

```
    supportive(From),
    supportive(To),
    To \== Block, % all objects are distinct
    From \== To,
    Block \== From.

% "positive" consequences of a move action
adds(move(X,From,To),[clear(From),on(X,To)]).

% "negative" consequences of a move action
subtracts(move(X,From,To),[clear(To),on(X,From)]).

% will the action allow us to achieve a particular goal?
achieves(Action,Goal) :-
    adds(Action,Goals),
    member(Goal,Goals).

% transform world using the specified action
apply(BeginState,Action,EndState) :-
    subtracts(Action,NegConsequences),
    delete_all(BeginState,NegConsequences,MidState),
    adds(Action,PosConsequences),
    append(PosConsequences,MidState,EndState).

% does the action undo already-satisfied goals?
preserves(Action,Goals) :-
    subtracts(Action,NegConsequences),
    \+ (member(Goal,NegConsequences),
        member(Goal,Goals)).

% Identify updated goals when regressing through an action.
% This establishes what the goal set might have been just prior
% to having achieved the goals, such that executing the action
% from the RegressedGoals state resulted in Goals being realized.
regress(Goals,Action,RegressedGoals) :-
    adds(Action,PosConsequences),
    delete_all(Goals,PosConsequences,ResidualGoals),
    can(Action,Prereqs), % prereqs assumed to be satisfied
    union_of_goals(Prereqs,ResidualGoals,RegressedGoals).

%%%%% NOUN-TYPE CONSIDERATIONS:
```

```
% checks whether all goals are met in current state
% (nothing prevents State from being a superset of Goals)
satisfied(_,[]).

satisfied(State,[Goal|Goals]) :-
    member(Goal,State),
    satisfied(State,Goals).

% checks whether current & goal states are literally equal
satisfied2(CurrState,GoalState) :-
    quicksort(CurrState,CurrStateSorted),
    quicksort(GoalState,GoalStateSorted),
    CurrStateSorted == GoalStateSorted.

% Is a contemplated goal possible in light of overall goals?
% This captures a bit more of the "physics" of block world.
impossible(on(X,X),_). % X can't sit on itself

impossible(on(X,Y),Goals) :-
    member(on(X,Y0),Goals),
    Y \== Y0 % X can't sit on two objects at once
    ;
    member(on(X0,Y),Goals),
    X \== X0 % two objects can't sit on Y at once
    ;
    member(clear(Y), Goals). % something's on it, so not clear

impossible(clear(X),Goals) :-
    member(on(_,X),Goals). % clear, so nothing's on it

% Backwards generating, heuristic-unaware node expansion
% policy for state-space search.
predecessor_4SSS(RegressedGoals,Action,Goals) :-
    member(Goal,Goals),
    achieves(Action,Goal),
    can(Action,_), % assures values are bound to variables
    preserves(Action,Goals),
    regress(Goals,Action,RegressedGoals).

% Forwards generating, heuristic-unaware node expansion
% policy for state-space search.
successor_4SSS(State,Action,NextState) :-
```

```
    can(Action,Prereqs), % allowed at present
    satisfied(State,Prereqs),
    \+ impossible(Action,NextState), % allowed in future
    apply(State,Action,NextState).

% purge elts of 2nd list from 1st to give 3rd: L1 - L2 = L3.
delete_all([],_,[]).

delete_all([X|L1],L2,L3) :-
    member(X,L2),
    !,
    delete_all(L1,L2,L3).

delete_all([X|L1],L2,[X|L3]) :-
    delete_all(L1,L2,L3).

% Is list L not a subset of any list from a list of lists?
not_found_in(_,[]) :- !.

not_found_in(L,[H|T]) :-
    delete_all(L,H,[]), !, fail
    ;
    not_found_in(L,T).

% Check that no set is literally equal to another.
% (SWI's merge sort implementation, sort/2, collapses
% redundant elts, which is not what we want. Let's use
% our own quicksort/2 instead.)
all_disjoint([_|[]]) :- !.

all_disjoint([L|T]) :-
    quicksort(L,Ls),
    list_is_disjoint(Ls,T),
    all_disjoint(T).

% !!!caller's responsible for sorting arg1 in advance!!!
list_is_disjoint(_,[]) :- !.

list_is_disjoint(L1s,[L2|Rest]) :-
    quicksort(L2,L2s),
    L1s \== L2s,
    list_is_disjoint(L1s,Rest).
```

```
% combines state w/ applicable action (BFS contexts)
% (alternatively, we could have just used '/',
% as with our N-Queens prototype)
:- op(400,yfx,'&').

% adapted for use with BFS
all_disjoint2([_|[]]) :- !.

all_disjoint2([L&A|T]) :-
    quicksort(L,Ls),
    list_is_disjoint2(Ls&A,T),
    all_disjoint2(T).

% adapted for use with the backwards BFS data structure;
% !!!caller's responsible for sorting L1s in advance!!!
list_is_disjoint2(_,[]) :- !.

list_is_disjoint2(L1s&Action,[L2&_|Rest]) :-
    quicksort(L2,L2s),
    L1s \== L2s,
    list_is_disjoint2(L1s&Action,Rest).

% adapted for use with the forwards BFS data structure;
% !!!caller's responsible for sorting L1s in advance!!!
list_is_disjoint3(_,[]) :- !.

list_is_disjoint3(L1s,[L2&_|Rest]) :-
    quicksort(L2,L2s),
    L1s \== L2s,
    list_is_disjoint3(L1s,Rest).

% lexicographically sort a list, the elts
% of which are potentially redundant
quicksort([],[]) :- !.

quicksort([Pivot|Tail],Sorted) :-
    partition(Pivot,Tail,Smaller,Larger),
    quicksort(Smaller,SortedSmaller),
    quicksort(Larger,SortedLarger),
    append(SortedSmaller,[Pivot|SortedLarger],Sorted).
```

```
partition(_,[],[],[]) :- !.

partition(Pivot,[X|T],[X|Smaller],Larger) :-
    X @=< Pivot, !,
    partition(Pivot,T,Smaller,Larger).

partition(Pivot,[X|T],Smaller,[X|Larger]) :-
    X @> Pivot, !, % the .GT. check's technically unnecessary
    partition(Pivot,T,Smaller,Larger).

% non-redundantly combine elts of first
% and second lists to give third...ensuring
% the union is possible in blocks world.
union_of_goals([],Goals,Goals) :- !.

union_of_goals(Goals,[],Goals) :- !.

union_of_goals([TestGoal|_],Goals,_) :-
    impossible(TestGoal,Goals),
    !, fail. % cut-fail combo (a red cut)

union_of_goals([X|L1],L2,L3) :-
    member(X,L2), !, % A U B = A + B - (A ^ B)
    union_of_goals(L1,L2,L3).

union_of_goals([X|L1],L2,[X|L3]) :-
    union_of_goals(L1,L2,L3).

%%%%% PRINTING-TYPE CONSIDERATIONS

% pretty printing routines for the plans generated
display_all([],_) :-
    nl, nl.

display_all([Plan|Rest]) :-
    display(Plan,1),
    display_all(Rest).

display([],_) :-
    nl, nl.

display([Move|Rest],Step) :-
```

```
    nl,
    tab(2), write(Step),
    tab(2), write(Move),
    Step1 is Step + 1,
    display(Rest,Step1).

display([],_,_) :-
    nl, nl.

display([Move|RestMoves],[State|RestStates],Step) :-
    nl,
    tab(2), write(Step),
    tab(2), write(Move),
    write(" -> "), write(State),
    Step1 is Step + 1,
    display(RestMoves,RestStates,Step1).

%%%%% PLANNING/SEARCHING CONSIDERATIONS

%%% I) basic means-ends analysis

soln_ME(Init,Goals,Plan) :- % alias
    soln_per_means_ends_planning(Init,Goals,Plan).

soln_per_means_ends_planning(Init,Goals,Plan) :- % wrapper
    path(Init,Goals,Plan,_),
    display(Plan,1).

path(BeginState,Goals,[],EndState) :-
    satisfied(BeginState,Goals), !,
    EndState = BeginState.

path(BeginState,Goals,Path,EndState) :-
    append(Path,_,_),
    append(PrePath,[Action|PostPath],Path),
    member(Goal,Goals),
    \+ member(Goal,BeginState),
    achieves(Action,Goal),
    can(Action,Prereqs),
    path(BeginState,Prereqs,PrePath,MidState1),
    apply(MidState1,Action,MidState2),
    path(MidState2,Goals,PostPath,EndState).
```

```
%%% II) means-ends analysis with goal regression

soln_MEGR(Init,Goals,Plan) :-
    soln_per_means_ends_planning_with_goal_regression(Init,Goals,Plan).

soln_per_means_ends_planning_with_goal_regression(Init,Goals,Plan) :-
    path1(Init,Goals,Plan),
    display(Plan,1).

path1(State,Goals,[]) :-
    satisfied(State,Goals).

path1(State,Goals,Path) :-
    append(PrePath,[Action],Path),
    member(Goal,Goals),
    achieves(Action,Goal),
    can(Action,_), % assures values are bound to variables
    preserves(Action,Goals),
    regress(Goals,Action,RegressedGoals),
    path1(State,RegressedGoals,PrePath).

/*
%%% III) depth-first search w/ iterative deepening using
         means-ends analysis with goal regression

   We know from experiments with the path & path1 predicates that
   we can get from I to G in the "super-simple" blocks world
   puzzle in three steps. Manual backwards chaining demonstrates
   that the means-ends, goal regression-based predecessor predicate
   defined above can be used to implement a state-space search:

?- init_state(I), goal_state(G), predecessor_4SSS(P1,_,G),
|     predecessor_4SSS(P2,_,P1), predecessor_4SSS(P3,_,P2),
|     all_disjoint([G,P1,P2,P3]), satisfied(P3,I).
I = [clear(c), on(c, a), on(a, 1), clear(2), clear(b), on(b, 3)],
G = [clear(1), clear(a), on(a, b), on(b, c), on(c, 2), clear(3)],
P1 = [on(a, 1), clear(b), clear(a), on(b, c), on(c, 2), clear(3)],
P2 = [on(b, 3), clear(c), on(a, 1), clear(b), clear(a), on(c, 2)],
P3 = [on(c, a), clear(2), on(b, 3), clear(c), on(a, 1), clear(b)] .

   We now proceed to define a recursive formulation for this
```

```
   that generalizes to solutions of arbitrary lengths. We'll
   try this using depth-first search w/ iterative deepening.
*/

% for recording DFSidMEGR solutions via state/ side effects
% (dynamic predicates should be used with extreme caution;
% they are often frowned upon, too (see Sterling & Shapiro))
:- dynamic
        path_DFSidMEGR/2.

% sets a "reasonable" bound on DFS max depth;
% adjust or call path2 directly if something
% else is desired.
soln_DFSidMEGR(Init,Goal,Plan) :-
    path2(Init,Goal,Plan,20,1).

% returns an optimal solution, if any solution exists
path2(_,_,_,ConstMaxDepth,CurrMaxDepth) :-
    CurrMaxDepth > ConstMaxDepth,
    !, fail.

path2(Init,Goal,Plan,_,CurrMaxDepth) :-
    path2aux(CurrMaxDepth,0,Init,Goal,[Goal],[]),
    !,
    path_DFSidMEGR(Steps,States),
    Plan = Steps,
    display(Steps,States,1),
    retract(path_DFSidMEGR(Steps,States)).

path2(Init,Goal,Plan,ConstMaxDepth,CurrMaxDepth) :-
    NextMaxDepth is CurrMaxDepth + 1,
    path2(Init,Goal,Plan,ConstMaxDepth,NextMaxDepth).

path2aux(_,_,Init,Curr,CurrOnward,Steps) :-
    satisfied2(Curr,Init),
    !,
    CurrOnward = [_|FutureStates],
    % I'm not sure of alternative mechanisms for returning an
    % accumulator variable's value back to a calling predicate
    % from the bottom of a stack of tail-recursive calls;
    % it's not clear that last-call optimization even allows
    % for such a thing. (In C, you'd just dereference a pointer.)
```

```
    % We can, however, accomplish this using side effects/ state
    % via Prolog's assert/retract mechanism.
    asserta(path_DFSidMEGR(Steps,FutureStates)).

path2aux(MaxDepth,Depth,Init,Curr,CurrOnward,Steps) :-
    Depth1 is Depth + 1,
    Depth1 =< MaxDepth,
    predecessor_4SSS(Prev,Action,Curr), % only generates backwards
    quicksort(Prev,PrevSort),
    list_is_disjoint(PrevSort,CurrOnward), % novel configuration
    path2aux(MaxDepth,Depth1,Init,Prev,
             [Prev|CurrOnward],[Action|Steps]).

/*
%%%   IV) Backward-chaining breadth-first search
          (leveraging means-ends w/ goal regression machinery)

   The DFSid framework we've explored above performs many
   redundant calculations as it progressively deepens its
   search. Although this kind of brute-force searching in
   general isn't practical for any but the simplest of tasks
   (due to combinatorial complexity), this is still a situation
   in which the caching/ memoization afforded by breadth-first
   search would likely yield a better solution.
*/

% database for recording BFS-based solutions
:- dynamic
       path_BFS/1.

soln_BFS_backward(Init,Goal,Plan) :-
    quicksort(Init,Init1),
    quicksort(Goal,Goal1),
    bfs_backward([[Goal1&"none"]],Init1),
    path_BFS(Plan),
    display(Plan,1).

% breadth-first search approach to state-space search
bfs_backward([[CurrState&Action|FutureStates]|_],InitState) :-
    satisfied2(CurrState,InitState),
    !,
    % record at front of database
```

```
    asserta(path_BFS([CurrState&Action|FutureStates])).

bfs_backward([HeadPath|OtherPaths],InitState) :-
    expand_backwards_for_bfs(HeadPath,HeadPathChildren),
    append(OtherPaths,HeadPathChildren,NewPathSet),
    bfs_backward(NewPathSet,InitState).

expand_backwards_for_bfs([Current&NextAction|Futures],Children) :-
    findall(
        [PreviousSorted&LastAction,
         Current&NextAction|Futures],
        (
            % predicate only expands nodes backwards
            predecessor_4SSS(Previous,LastAction,Current),
            quicksort(Previous,PreviousSorted),
            % novel child?
            list_is_disjoint2(PreviousSorted&LastAction,
                              [Current&NextAction|Futures])
        ),
        Children0),
    sort(Children0,Children).

%%% V) Forward-chaining breadth-first search
%%%    (leveraging means-ends w/ goal regression machinery)

soln_BFS_forward(Init,Goal,Plan) :-
    quicksort(Init,Init1),
    quicksort(Goal,Goal1),
    bfs_forward([[Init1]],Goal1),
    path_BFS(Plan),
    display(Plan,1).

bfs_forward([[CurrState|PastStates]|_],GoalState) :-
    satisfied2(CurrState,GoalState),
    !,
    reverse([CurrState&"none"|PastStates],Path),
    asserta(path_BFS(Path)).

bfs_forward([HeadPath|OtherPaths],GoalState) :-
    expand_forwards_for_bfs(HeadPath,HeadPathChildren),
    append(OtherPaths,HeadPathChildren,NewPathSet),
    bfs_forward(NewPathSet,GoalState).
```

```
expand_forwards_for_bfs([Current|Past],Children) :-
    findall(
        [NextSorted,Current&Action|Past],
        (
            successor_4SSS(Current,Action,Next),
            quicksort(Next,NextSorted),
            % novel child?
            list_is_disjoint3(NextSorted,[Current&Action|Past])
        ),
        Children0),
    sort(Children0,Children).

/*
%%%   VI) Bidirectional breadth-first search

   Lastly, let's try a search by alternately extending
   forward and backward chains, with the hope that they
   will eventually overlap to compose an executable plan.
*/

soln_BFS_bidir(Init,Goal,Plans) :-
    bfs_bidirectional("backward",[[Init]],[[Goal&"none"]],Plans),
    !, % "one" and done! (unless multiple optimal solns exist)
    display_all(Plans).

bfs_bidirectional(_,PathsForward,PathsBackward,Plans) :-
    findall(
        Plan,
        (member(X,PathsForward),
         [CurrStateF|PastStates] = X,
         member(Y,PathsBackward),
         [CurrStateB&Action|FutureStates] = Y,
         CurrStateF == CurrStateB,
         reverse(PastStates,PastStatesR),
         append(PastStatesR,
                [CurrStateF&Action|FutureStates],
                Plan),
         all_disjoint2(Plan),
         nonvar(Plan)
        ),
        Plans),
```

```
    length(Plans,PlansL),
    PlansL > 0.

bfs_bidirectional(Dir,PathsForward,PathsBackward,Plans) :-
    Dir == "forward", !,
    expand_full_tier(Dir,PathsForward,PathsForwardExp),
    bfs_bidirectional("backward",PathsForwardExp,PathsBackward,Plans).

bfs_bidirectional(Dir,PathsForward,PathsBackward,Plans) :-
    Dir == "backward", % sanity check (given cut in clause above)
    expand_full_tier(Dir,PathsBackward,PathsBackwardExp),
    bfs_bidirectional("forward",PathsForward,PathsBackwardExp,Plans).

% generate all child nodes from all parents at given level
expand_full_tier(_,[],[]) :- !.

% ?- init_sussman(I), expand_full_tier("forward",[[I]],F1).
expand_full_tier(Dir,[HeadPath|RestPaths],ExpandedPaths) :-
    Dir == "forward", !,
    expand_forwards_for_bfs(HeadPath,ExpandedHeadPath),
    expand_full_tier(Dir,RestPaths,ExpandedRestPaths),
    append(ExpandedHeadPath,ExpandedRestPaths,ExpandedPaths).

% ?- goal_sussman(G), expand_full_tier("backward",[[G&"none"]],B1).
expand_full_tier(Dir,[HeadPath|RestPaths],ExpandedPaths) :-
    Dir == "backward", % sanity check
    expand_backwards_for_bfs(HeadPath,ExpandedHeadPath),
    expand_full_tier(Dir,RestPaths,ExpandedRestPaths),
    append(ExpandedHeadPath,ExpandedRestPaths,ExpandedPaths).
```