```
/* Prolog solution to the farmer's wolf-chicken-grain problem.

   Michael E. Sparks, 5 Dec 2020

SAMPLE USAGE:

?- init(I), goal(G), setof(P,soln_DFSid(I,G,P),Ps).

  0  BARC-W:[chicken,farmer,grain,wolf]  <--Rt1-->  BARC-E:[]
  1  BARC-W:[grain,wolf]  <--Rt1-->  BARC-E:[chicken,farmer]
  2  BARC-W:[farmer,grain,wolf]  <--Rt1-->  BARC-E:[chicken]
  3  BARC-W:[wolf]  <--Rt1-->  BARC-E:[chicken,farmer,grain]
  4  BARC-W:[chicken,farmer,wolf]  <--Rt1-->  BARC-E:[grain]
  5  BARC-W:[chicken]  <--Rt1-->  BARC-E:[farmer,grain,wolf]
  6  BARC-W:[chicken,farmer]  <--Rt1-->  BARC-E:[grain,wolf]
  7  BARC-W:[]  <--Rt1-->  BARC-E:[chicken,farmer,grain,wolf]


  0  BARC-W:[chicken,farmer,grain,wolf]  <--Rt1-->  BARC-E:[]
  1  BARC-W:[grain,wolf]  <--Rt1-->  BARC-E:[chicken,farmer]
  2  BARC-W:[farmer,grain,wolf]  <--Rt1-->  BARC-E:[chicken]
  3  BARC-W:[grain]  <--Rt1-->  BARC-E:[chicken,farmer,wolf]
  4  BARC-W:[chicken,farmer,grain]  <--Rt1-->  BARC-E:[wolf]
  5  BARC-W:[chicken]  <--Rt1-->  BARC-E:[farmer,grain,wolf]
  6  BARC-W:[chicken,farmer]  <--Rt1-->  BARC-E:[grain,wolf]
  7  BARC-W:[]  <--Rt1-->  BARC-E:[chicken,farmer,grain,wolf]

I = [[chicken, farmer, grain, wolf], []],
G = [[], [chicken, farmer, grain, wolf]],
Ps = [[[[], [chicken, farmer, grain, wolf]], [[chicken, farmer], [grain, wolf]],
 [[chicken], [farmer, grain, wolf]], [[chicken, farmer, grain], [wolf]], [[grain
], [chicken|...]], [[farmer|...], [...]], [[...|...]|...], [...|...]], [[[], [ch
icken, farmer, grain, wolf]], [[chicken, farmer], [grain, wolf]], [[chicken], [f
armer, grain|...]], [[chicken, farmer|...], [grain]], [[wolf], [...|...]], [[...
|...]|...], [...|...]|...]].

*/

% main user interface, uses a depth-first/
% breadth-first hybrid searching approach
soln_per_depth_first_search_with_iterative_deepening(Init,Goal,Plan) :-
    path(Init,Goal,Plan),
    reverse(Plan,Plan1),
    display(Plan1,0).

% create alias for full predicate name
soln_DFSid(Init,Goal,Plan) :-
    soln_per_depth_first_search_with_iterative_deepening(Init,Goal,Plan).

% returns a legal path from initial node to goal node
path0(N,N,[N]).

path0(Init,Goal,[Goal|Path]) :-
    path0(Init,Penultimate,Path),
    successor(Penultimate,Goal),
    \+ member(Goal,Path).

% Let's improve our code safety by limiting the depth
% to which iterative deepening can plunge.
% There’s risk that the limit is set too shallow to find a solution,
% so a practitioner should generally set this high enough that
% the stack can be put to good use for effective search.
% The flip side is the deeper the search, the longer the run time,
% so it shouldn’t be set so high that efficiency suffers.
% (A limit of 13 is the minimum to find solutions here.)
max_depth(32).

path(Init,Goal,Path) :-
    max_depth(Dlim),
    call_with_depth_limit(path0(Init,Goal,Path),Dlim,_),
    nonvar(Path). % solutions must be fully instantiated.

% State representation is a list of two lists,
% being the left and right shores.
% Shore constituents should be recorded in lexicographic order.
init([X,[]]) :-
    quicksort([wolf,grain,chicken,farmer],X),
    !.

goal([[],X]) :-
    quicksort([farmer,wolf,chicken,grain],X),
    !.

% checks whether shore is allowed when farmer’s away
disallowed(Shore) :-
    member(wolf,Shore),
    member(chicken,Shore),
    \+ member(farmer, Shore),
    !.

disallowed(Shore) :-
    member(chicken,Shore),
    member(grain,Shore),
    \+ member(farmer, Shore).

% data selectors tailored for our state representation
left(State,Lshore) :-
    State = [Lshore,_].

right(State,Rshore) :-
    State = [_,Rshore].

% case in which farmer’s on left shore and
% carries an object to the right shore.
successor(BeginState,EndState) :-
    left(BeginState,Lbegin),
    right(BeginState,Rbegin),
    member(farmer,Lbegin),
    member(X,Lbegin),
    X \= farmer,
    delete_all(Lbegin,[X,farmer],Lend0),
    \+ disallowed(Lend0), % shore’s safe sans farmer
    quicksort(Lend0,Lend),
    left(EndState,Lend),
    append([X,farmer],Rbegin,Rend0),
    quicksort(Rend0,Rend),
    right(EndState,Rend).
```

```
% symmetric case in which farmer’s on right shore and
% carries an object to the left shore.
successor(BeginState,EndState) :-
    left(BeginState,Lbegin),
    right(BeginState,Rbegin),
    member(farmer,Rbegin),
    member(X,Rbegin),
    X \= farmer,
    delete_all(Rbegin,[X,farmer],Rend0),
    \+ disallowed(Rend0),
    quicksort(Rend0,Rend),
    right(EndState,Rend),
    append([X,farmer],Lbegin,Lend0),
    quicksort(Lend0,Lend),
    left(EndState,Lend).

% case in which farmer’s on left shore and
% carries NO object to the right shore.
successor(BeginState,EndState) :-
    left(BeginState,Lbegin),
    right(BeginState,Rbegin),
    member(farmer,Lbegin),
    delete_all(Lbegin,[farmer],Lend0),
    \+ disallowed(Lend0),
    quicksort(Lend0,Lend),
    left(EndState,Lend),
    append([farmer],Rbegin,Rend0),
    quicksort(Rend0,Rend),
    right(EndState,Rend).

% symmetric case in which farmer’s on right shore and
% carries NO object to the left shore.
successor(BeginState,EndState) :-
    left(BeginState,Lbegin),
    right(BeginState,Rbegin),
    member(farmer,Rbegin),
    delete_all(Rbegin,[farmer],Rend0),
    \+ disallowed(Rend0),
    quicksort(Rend0,Rend),
    right(EndState,Rend),
    append([farmer],Lbegin,Lend0),
    quicksort(Lend0,Lend),
    left(EndState,Lend).

% purge elts of second list from first to give third:
% 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).

% lexicographically sort a list
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).

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

display([Move|Rest],Step) :-
    nl,
    tab(2), write(Step), tab(2),
    left(Move,Left), write("BARC-W:"), write(Left),
    write(" <--Rt1--> "),
    right(Move,Right), write("BARC-E:"), write(Right),
    Step1 is Step + 1,
    display(Rest,Step1).
```