Experiment - 1

1. Implementation of DFS for water jug problem using LISP/PROLOG

Bb_planner.pl Program:

```
%%  bb_planner.pl    Brandon Bennett (25/11/2014)
%%  Updated 16/11/2018
%%  Updated 16/11/2019
%%    Changes:
%%    1) goal_state now called with the solution search.
%%       (Previously goal was determined prior to search, which is
%%       less flexible. Now it can search for several potential goals
%%       within a single goal.
%%    2) equivalent_states is now only used by the loop checker, not
%%       when testing for goals, so goal_state predicate needs to be
%%       true for all acceptable goals.
%%  Updated 12/05/2022
%%    Change: eliminated some retundant backtracking in the 'solution'
%     predicate.

%% This code implements a breadth-first search strategy for
%% transition-based search/planning problems.
%% It has an option to automatically eliminate loops and redundant
%% diversions by discarding any path whose end state is the same as
%% that of some shorter path.

%% To use the algorithm on a particular problem, you need to define
%% a number of problem-specific predicates that give the intitial
%% and goal states and describe the possible transitions between
%% states. This is explained in detail at the end of this file.

:- use_module( library(lists) ).

find_solution :-
        initial_state( Initial ),
        write( '== Starting Search ==' ), nl,
        solution( [[Initial]], StateList ),
        length( StateList, Len ),
        Transitions is Len -1,
```

```
        format( '~n** FOUND SOLUTION of length ~p **', [Transitions] ), nl,
        showlist( StateList ), !.

%find_solution :-
%        write( '!! FAILED: No plan reaches a goal  !!' ), nl, fail.

%% Base case for finding solution.
%% Find a statelist whose last state is the goal or
solution( StateLists, StateList ) :-
        member( StateList, StateLists ),
        last( StateList, Last ),
        goal_state(Last),
        report_progress( StateLists, final ).

%% Recursive rule that looks for a solution by extending
%% each of the generated state lists to add a further state.
solution( StateLists, StateList ) :-
        report_progress( StateLists, ongoing ),
        extend( StateLists, Extensions ), !,
        solution( Extensions, StateList ), !.

solution( _, _ ) :- !,
        write( '!! Cannot extend statelist !!' ), nl,
        write( '!! FAILED: No plan reaches a goal  !!' ), nl,
        fail, !.

%% Extend each statelist in a set of possible state lists.
%% If loopcheck(on) will not extend to any state previously reached
%% in any of the state lists, to avoid loops.
extend( StateLists, ExtendedStateLists ) :-
   setof( ExtendedStateList,
          StateList^Last^Next^( member( StateList, StateLists ),
                                last( StateList, Last ),
                                transition( Last, Next ),
                                legal_state( Next ),
                                no_loop_or_loopcheck_off( Next, StateLists ),
                                append( StateList, [Next], ExtendedStateList )
                              ),
           ExtendedStateLists
         ).
```

```
poss_empty_setof( X, G, S ) :- setof( X, G, S), !.
poss_empty_setof(_,_, []).


no_loop_or_loopcheck_off( _, _) :- loopcheck(off), !.
no_loop_or_loopcheck_off( Next, StateLists ) :-
                  \+( already_reached( Next, StateLists ) ).

%% Check whether State (or some equivalent state) has already been
%% reached in any state list in StateLists.
already_reached( State,  StateLists ) :-
        member( StateList, StateLists ),
        member( State1, StateList ),
        equivalent_states( State, State1 ).

%% Print out list, each element on a separate line.
showlist([]).
showlist([H | T]) :- write( H ), nl, showlist( T ).

%% Report progress after each cycle of the planner:
report_progress( StateLists, Status ) :-
    length( StateLists, NS ),
    StateLists = [L|_], length( L, N ),
    Nminus1 is N - 1,
    write( 'Found '), write( NS ),
    write( ' states reachable in path length ' ), write(Nminus1), nl,
    ( Status = ongoing ->
      (write( 'Computing extensions of length : ' ), write(N), nl)
      ; true
    ).


%% To run this you need to define the following predicates:

% initial_state( SomeState ).
% goal_state( AnotherState ).

% Specify possible transitions from any state S1
```

```
% transition( S1, S2 ) :- conditions.
%        :                    :         % specify as many as needed
% transition( S1, S2 ) :- conditions.

% You can add a further condition on what states are valid:
% legal_state( S ) :- conditions.
% If no special conditions are needed just use:
% legal_state( _ ). % Allow any state

% You can tell the planner that some state representations are equivalent.
% equivalent_states( S1, S2 ) :- conditions.
% If all distinct state expressions represent different states, just use:
% equivalent_states( S, S ).
% The equivalent_states predicate is only used when checking if a generated
% state is equivalent to an already reached state, when loopcheck is on.

% You must tell the planner whether to check for and discard repeated states.
% Specify one of:
% loopcheck(off).
% loopcheck(on).
% Eliminating loops can greatly prune the search space.
% But looking for loops can use a lot of processing time, and may not be
% worth doing (especailly if loops cannot occur!).

% To run each time file is loaded, add the following command to the
% the end of your program file.
% :- find_solution.

% This special SWISH comment adds the find_solution query to the examples
%  menu under the console window. So you can use that instead when running
%  in SWISH. (But you first need to define the initial state, goal state,
%  transition relation etc., as explained above
/** <examples>
?- find_solution.
*/
```

Waterjug.pl Program:

```
:- include(bb_planner).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  bb_planner Example:  A Measuring Jugs Problem
%%  Last modified 16/11/2019
%%  Changes: 1) Defined goal muliple possible goal_state options, which now
%%           works because of update to bb_planner
%%           2) Simplified and added explanation to the pour/4 predicate.

%%% There are three jugs (a,b,c), whose capacity is respectively:
%%% 3 litres, 5 litres and 8 litres.
%%% Initially jugs a and b are empty and jug c is full of water.

%%%% Goal: Find a sequence of pouring actions by which you can measure out
%%% 4 litres of water into one of the jugs without spilling any.

%%%  State representation will be as follows:
%%%  A state is a list:  [ how_reached, Jugstate1, Jugstate2, Jugstate3 ]
%%%  Where each JugstateN is a lst of the form: [jugname, capcity, content]
initial_state( [initial, [a,3,0], [b,5,0], [c,8,8]]).

%% Define goal state to accept any state where one of the
%% jugs contains 4 litres of water:
goal_state( [_, [a,_,4], [b,_,_], [c,_,_]]).
goal_state( [_, [a,_,_], [b,_,4], [c,_,_]]).
goal_state( [_, [a,_,_], [b,_,_], [c,_,4]]).

% Is it possible to get to this state?
%goal_state( [_, [a,_,_], [b,_,3], [c,_,3]]).
% Or this one?
%goal_state( [_, [a,_,_], [b,_,_], [c,_,6]]).

% What if I want to share out the water equally between two people?


%%% The state transitions are "pour" operations, where the contents of
%%% one jug is poured into another jug up to the limit of the capacity
%%% of the recipient jug.
%%% There are six possible pour actions from one jug to another:
```

```
transition( [_, A1,B1,C], [pour_a_to_b, A2,B2,C] ) :- pour(A1,B1,A2,B2).
transition( [_, A1,B,C1], [pour_a_to_c, A2,B,C2] ) :- pour(A1,C1,A2,C2).
transition( [_, A1,B1,C], [pour_b_to_a, A2,B2,C] ) :- pour(B1,A1,B2,A2).
transition( [_, A,B1,C1], [pour_b_to_c, A,B2,C2] ) :- pour(B1,C1,B2,C2).
transition( [_, A1,B,C1], [pour_c_to_a, A2,B,C2] ) :- pour(C1,A1,C2,A2).
transition( [_, A,B1,C1], [pour_c_to_b, A,B2,C2] ) :- pour(C1,B1,C2,B2).

%%% The pour operation is defined as follows:
% Case where there is room to pour full contents of Jug1 to Jug2
% so Jug 1 ends up empty and its contents are added to Jug2.
pour( [Jug1, Capacity1, Initial1], [Jug2, Capacity2, Initial2], % initial jug states
     [Jug1 ,Capacity1, 0],   [Jug2, Capacity2, Final2]        % final jug states
   ):-
      Initial1 =< (Capacity2 - Initial2),
      Final2 is Initial1 + Initial2.

% Case where only some of Jug1 contents fit into Jug2
% Jug2 ends up full and some water will be left in Jug1.
pour( [Jug1, Capacity1, Initial1], [Jug2, Capacity2, Initial2], % initial jug states
     [Jug1 ,Capacity1, Final1],   [Jug2, Capacity2, Capacity2] % final jug states
   ):-
      Initial1 > (Capacity2 - Initial2),
      Final1 is Initial1 - (Capacity2 - Initial2).

%% Define the other helper predicates that specify how bb_planner will operate:
legal_state( _ ).              % All states that can be reached are legal
equivalent_states( X, X ).     % Only identical states are equivalent.
loopcheck(on).                 % Don't allow search to go into a loop.

%% Call this goal to find a solution.
%:- find_solution.

% This special comment adds the find_solution query to the examples menu
% under the console window.
/** <examples>
?- find_solution.
*/
```

Output:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.3)
File   Edit   Settings   Run   Debug   Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/admin/Downloads/bb_jugs.pl compiled 0.02 sec, 30 clauses
?- find_solution.
== Starting Search ==
Found 1 states reachable in path length 0
Computing extensions of length : 1
Found 6 states reachable in path length 1
Computing extensions of length : 2
Found 8 states reachable in path length 2
Computing extensions of length : 3
Found 13 states reachable in path length 3
Computing extensions of length : 4
Found 14 states reachable in path length 4
Computing extensions of length : 5
Found 8 states reachable in path length 5
Computing extensions of length : 6
Found 20 states reachable in path length 6

** FOUND SOLUTION of length 6 **
[initial,[a,3,0],[b,5,0],[c,8,8]]
[pour_c_to_b,[a,3,0],[b,5,5],[c,8,3]]
[pour_b_to_a,[a,3,3],[b,5,2],[c,8,3]]
[pour_a_to_c,[a,3,0],[b,5,2],[c,8,6]]
[pour_b_to_a,[a,3,2],[b,5,0],[c,8,6]]
[pour_c_to_b,[a,3,2],[b,5,5],[c,8,1]]
[pour_b_to_a,[a,3,3],[b,5,4],[c,8,1]]
true.

?-
```

Experiment - 2

Implementation of BFS for tic-tac-toe problem using LISP/PROLOG/Java.

Tictoctoe.pl Program.

```
% A Tic-Tac-Toe program in Prolog.
% To play a game with the computer, type
% playo.
% To watch the computer play a game with itself, type
% selfgame.

% Predicates that define the winning conditions:

win(Board, Player) :- rowwin(Board, Player).
win(Board, Player) :- colwin(Board, Player).
win(Board, Player) :- diagwin(Board, Player).

rowwin(Board, Player) :- Board = [Player,Player,Player,_,_,_,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,Player,Player,Player,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,_,_,_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,_,_,Player,_,_,Player,_,_].
colwin(Board, Player) :- Board = [_,Player,_,_,Player,_,_,Player,_].
colwin(Board, Player) :- Board = [_,_,Player,_,_,Player,_,_,Player].

diagwin(Board, Player) :- Board = [Player,_,_,_,Player,_,_,_,Player].
diagwin(Board, Player) :- Board = [_,_,Player,_,Player,_,Player,_,_].

% Helping predicate for alternating play in a "self" game:

other(x,o).
other(o,x).

game(Board, Player) :- win(Board, Player), !, write([player, Player, wins]).
game(Board, Player) :-
  other(Player,Otherplayer),
  move(Board,Player,Newboard),
  !,
  display(Newboard),
```

```
  game(Newboard,Otherplayer).

move([b,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).
move([A,b,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).
move([A,B,b,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).
move([A,B,C,b,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).
move([A,B,C,D,b,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).
move([A,B,C,D,E,b,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).
move([A,B,C,D,E,F,b,H,I], Player, [A,B,C,D,E,F,Player,H,I]).
move([A,B,C,D,E,F,G,b,I], Player, [A,B,C,D,E,F,G,Player,I]).
move([A,B,C,D,E,F,G,H,b], Player, [A,B,C,D,E,F,G,H,Player]).

display([A,B,C,D,E,F,G,H,I]) :- write([A,B,C]),nl,write([D,E,F]),nl,
 write([G,H,I]),nl,nl.

selfgame :- game([b,b,b,b,b,b,b,b,b],x).

% Predicates to support playing a game with the user:

x_can_win_in_one(Board) :- move(Board, x, Newboard), win(Newboard, x).

% The predicate orespond generates the computer's (playing o) reponse
% from the current Board.

orespond(Board,Newboard) :-
  move(Board, o, Newboard),
  win(Newboard, o),
  !.
orespond(Board,Newboard) :-
  move(Board, o, Newboard),
  not(x_can_win_in_one(Newboard)).
orespond(Board,Newboard) :-
  move(Board, o, Newboard).
orespond(Board,Newboard) :-
  not(member(b,Board)),
  !,
  write('Cats game!'), nl,
  Newboard = Board.

% The following translates from an integer description
```

```
% of x's move to a board transformation.

xmove([b,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).
xmove([A,b,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).
xmove([A,B,b,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).
xmove([A,B,C,b,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).
xmove([A,B,C,D,b,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).
xmove([A,B,C,D,E,b,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).
xmove([A,B,C,D,E,F,b,H,I], 7, [A,B,C,D,E,F,x,H,I]).
xmove([A,B,C,D,E,F,G,b,I], 8, [A,B,C,D,E,F,G,x,I]).
xmove([A,B,C,D,E,F,G,H,b], 9, [A,B,C,D,E,F,G,H,x]).
xmove(Board, N, Board) :- write('Illegal move.'), nl.

% The 0-place predicate playo starts a game with the user.

playo :- explain, playfrom([b,b,b,b,b,b,b,b,b]).

explain :-
  write('You play X by entering integer positions followed by a period.'),
  nl,
  display([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :- win(Board, x), write('You win!').
playfrom(Board) :- win(Board, o), write('I win!').
playfrom(Board) :- read(N),
  xmove(Board, N, Newboard),
  display(Newboard),
  orespond(Newboard, Newnewboard),
  display(Newnewboard),
  playfrom(Newnewboard).
```

Output:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.3)
File  Edit  Settings  Run  Debug  Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
Warning: c:/users/admin/desktop/ttt.pl:86:
Warning:    Singleton variables: [N]
% c:/Users/admin/Desktop/TTT.pl compiled 0.00 sec, 46 clauses
?- selfgame.
[x,b,b]
[b,b,b]
[b,b,b]

[x,o,b]
[b,b,b]
[b,b,b]

[x,o,x]
[b,b,b]
[b,b,b]

[x,o,x]
[o,b,b]
[b,b,b]

[x,o,x]
[o,x,b]
[b,b,b]

[x,o,x]
[o,x,o]
[b,b,b]

[x,o,x]
[o,x,o]
[x,b,b]

[x,o,x]
[o,x,o]
[x,o,b]

[player,x,wins]
true.

?-
```

Experiment - 3

Implementation of TSP using heuristic approach using Java/LISP/Prolog

TSP.pl Program.

```
/* tsp(Towns, Route, Distance) is true if Route is an optimal solution of */
/* length Distance to the Travelling Salesman Problem for the Towns, */
/* where the distances between towns are defined by distance/3. */
/* An exhaustive search is performed using the database. The distance */
/* is calculated incrementally for each route. */
/* e.g. tsp([a,b,c,d,e,f,g,h], Route, Distance) */
tsp(Towns, _, _):-
 retract_all(bestroute(_)),
 assert(bestroute(r([], 2147483647))),
 route(Towns, Route, Distance),
 bestroute(r(_, BestSoFar)),
 Distance < BestSoFar,
 retract(bestroute(r(_, BestSoFar))),
 assert(bestroute(r(Route, Distance))),
 fail.
tsp(_, Route, Distance):-
 retract(bestroute(r(Route, Distance))), !.
/* route([Town|OtherTowns], Route, Distance) is true if Route starts at */
/* Town and goes through all the OtherTowns exactly once, and Distance
*/
/* is the length of the Route (including returning to Town from the last */
/* OtherTown) as defined by distance/3. */
route([First|Towns], [First|Route], Distance):-
 route_1(Towns, First, First, 0, Distance, Route).
route_1([], Last, First, Distance0, Distance, []):-
 distance(Last, First, Distance1),
 Distance is Distance0 + Distance1.
route_1(Towns0, Town0, First, Distance0, Distance, [Town|Towns]):-
 remove(Town, Towns0, Towns1),
 distance(Town0, Town, Distance1),
 Distance2 is Distance0 + Distance1,
 route_1(Towns1, Town, First, Distance2, Distance, Towns).
distance(X, Y, D):-X @< Y, !, e(X, Y, D).
distance(X, Y, D):-e(Y, X, D).
```

```
retract_all(X):-retract(X), retract_all(X).
retract_all(X).
/*
 * Data: e(From,To,Distance) where From @< To
 */
e(a,b,11). e(a,c,41). e(a,d,27). e(a,e,23). e(a,f,43). e(a,g,15). e(a,h,20).
e(b,c,32). e(b,d,16). e(b,e,21). e(b,f,33). e(b,g, 7). e(b,h,13).
e(c,d,25). e(c,e,49). e(c,f,35). e(c,g,34). e(c,h,21).
e(d,e,26). e(d,f,18). e(d,g,14). e(d,h,19).
e(e,f,31). e(e,g,15). e(e,h,34).
e(f,g,28). e(f,h,36).
e(g,h,19).
```

Output:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.3)
File   Edit   Settings   Run   Debug   Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
Warning: c:/users/admin/desktop/tsp.pl:36:
Warning:    Singleton variables: [X]
% c:/Users/admin/Desktop/TSP.pl compiled 0.00 sec, 37 clauses
?- e(From,To,Distance).
From = a,
To = b,
Distance = 11 .

?- e(b,f,Distance).
Distance = 33.

?- e(a,f,Distance).
Distance = 43.

?-  e(a,g,Distance).
Distance = 15.

?-  e(c,f,Distance).
Distance = 35.

?-
```

Experiment - 4

Implementation of Simulated Annealing Algorithm using LISP/PROLOG

SA.pl Program.

```
/*This is the data set.*/

edge(a, b, 3).
edge(a, c, 4).
edge(a, d, 2).
edge(a, e, 7).
edge(b, c, 4).
edge(b, d, 6).
edge(b, e, 3).
edge(c, d, 5).
edge(c, e, 8).
edge(d, e, 6).
edge(b, a, 3).
edge(c, a, 4).
edge(d, a, 2).
edge(e, a, 7).
edge(c, b, 4).
edge(d, b, 6).
edge(e, b, 3).
edge(d, c, 5).
edge(e, c, 8).
edge(e, d, 6).
edge(a, h, 2).
edge(h, d, 1).

/* Finds the length of a list, while there is something in the list it increments N
        when there is nothing left it returns.*/

len([], 0).
len([H|T], N):- len(T, X), N is X+1 .

/*Best path, is called by shortest_path.  It sends it the paths found in a
```

```
 path, distance format*/

best_path(Visited, Total):- path(a, a, Visited, Total).


/*Path is expanded to take in distance so far and the nodes visited */

path(Start, Fin, Visited, Total) :- path(Start, Fin, [Start], Visited, 0, Total).

/*This adds the stopping location to the visited list, adds the distance and then calls recursive
          to the next stopping location along the path */

path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, StopLoc, Distance), NewCostn is Costn + Distance, \+ member(StopLoc,
CurrentLoc),
    path(StopLoc, Fin, [StopLoc|CurrentLoc], Visited, NewCostn, Total).

/*When we find a path back to the starting point, make that the total distance and make
          sure the graph has touch every node*/

path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, Fin, Distance), reverse([Fin|CurrentLoc], Visited), len(Visited, Q),
    (Q\=7 -> Total is 100000; Total is Costn + Distance).

/*This is called to find the shortest path, takes all the paths, collects them in holder.
          Then calls pick on that holder which picks the shortest path and returns it*/

shortest_path(Path):-setof(Cost-Path, best_path(Path,Cost), Holder),pick(Holder,Path).

/* Is called, compares 2 distances. If cost is smaller than bcost, no need to go on. Cut it.*/

best(Cost-Holder,Bcost-_,Cost-Holder):- Cost<Bcost,!.
best(_,X,X).

/*Takes the top path and distance off of the holder and recursively calls it.*/

pick([Cost-Holder|R],X):- pick(R,Bcost-Bholder),best(Cost-Holder,Bcost-Bholder,X),!.
pick([X],X).

/*?-shortest_path(Path).*/
```

Output:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.3)
File   Edit   Settings   Run   Debug   Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
Warning: c:/users/admin/desktop/travelling.pl:33:
Warning:    Singleton variables: [H]
% c:/Users/admin/Desktop/Travelling.pl compiled 0.00 sec, 33 clauses
?- shortest_path(Path).
Path = 20-[a, h, d, e, b, c, a].

?-
```

Experiment - 5

5. Implementation of DFS for water jug problem using LISP/PROLOG

HC8PP.pl Program:

```
% Simple Prolog Planner for the 8 Puzzle Problem

% This predicate initialises the problem states. The first argument
% of solve/3 is the initial state, the 2nd the goal state, and the
% third the plan that will be produced.

test(Plan):-
   write('Initial state:'),nl,
      Init= [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7),
at(tile1,8), at(tile7,9)],
   write_sol(Init),
      Goal= [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7),
at(tile7,8), at(tile8,9)],
   nl,write('Goal state:'),nl,
   write(Goal),nl,nl,
   solve(Init,Goal,Plan).

solve(State, Goal, Plan):-
   solve(State, Goal, [], Plan).

%Determines whether Current and Destination tiles are a valid move.
is_movable(X1,Y1) :- (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3 is X1 - Y1).


% This predicate produces the plan. Once the Goal list is a subset
% of the current State the plan is complete and it is written to
% the screen using write_sol/1.

solve(State, Goal, Plan, Plan):-
   is_subset(Goal, State), nl,
   write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
```

```
    act(Action, Preconditions, Delete, Add),
    is_subset(Preconditions, State),
    \+ member(Action, Sofar),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    solve(NewState, Goal, [Action|Sofar], Plan).

% The problem has three operators.
% 1st arg = name
% 2nd arg = preconditions
% 3rd arg = delete list
% 4th arg = add list.

% Tile can move to new position only if the destination tile is empty & Manhattan distance = 1
act(move(X,Y,Z),
    [at(X,Y), at(empty,Z), is_movable(Y,Z)],
    [at(X,Y), at(empty,Z)],
    [at(X,Z), at(empty,Y)]).


% Utility predicates.

% Check is first list is a subset of the second

is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).
is_subset([], _).

% Remove all elements of 1st list from second to create third.

delete_list([H|T], Curstate, Newstate):-
    remove(H, Curstate, Remainder),
    delete_list(T, Remainder, Newstate).
delete_list([], Curstate, Curstate).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).
```

```
write_sol([]).
write_sol([H|T]):-
   write_sol(T),
   write(H), nl.

append([H|T], L1, [H|L2]):-
   append(T, L1, L2).
append([], L, L).

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

Output:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.3)
File  Edit  Settings  Run  Debug  Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/admin/Desktop/HC8PP.pl compiled 0.00 sec, 18 clauses
?- test(Plan).
Initial state:
at(tile7,9)
at(tile1,8)
at(tile5,7)
at(tile6,6)
at(tile2,5)
at(empty,4)
at(tile8,3)
at(tile3,2)
at(tile4,1)

Goal state:
[at(tile1,1),at(tile2,2),at(tile3,3),at(tile4,4),at(empty,5),at(tile5,6),at(tile6,7),at(tile7,8),at(tile8,9)]

false.

?-
```