Water Jug problem using BFS

Program:-

```
go :-
	start(Start),
	solve(Start, Solution),
	reverse(Solution, L),
	print(L, _).

solve(Start, Solution) :-
	breadthfirst([[Start]], Solution).

% breadthfirst([Path1, Path2, ...], Solution):
% Solution is an extension to a goal of one of the paths

breadthfirst([[Node | Path] | _], [Node | Path]) :-
	goal(Node).

breadthfirst([Path | Paths], Solution) :-
	extend(Path, NewPaths),
	append(Paths, Paths1),
	breadthfirst( Solution).

extend([Node | Path], NewPaths) :-
	findall([NewNode, Node | Path],
		(next_state(Node, NewNode), \+ member(NewNode, [Node | Path])),
		NewPaths),
	!.

extend(_, []).

% States are represented by the compound term (4-gallon jug, 3-gallon jug);
% In the initial state, both jugs are empty:

start((0, 0)).

% The goal state is to measure 2 gallons of water:
goal((2, _)).
goal((_, 2)).

% Fill up the 4-gallon jug if it is not already filled:
next_state((X, Y), (4, Y)) :- X < 4.

% Fill up the 3-gallon jug if it is not already filled:
next_state((X, Y), (X, 3)) :- Y < 3.

% If there is water in the 3-gallon jug (Y > 0) and there is room in the 4-gallon jug (X < 4), THEN use it
to fill up
% the 4-gallon jug until it is full (4-gallon jug = 4 in the new state) and leave the rest in the 3-gallon
jug:
next_state((X, Y), (4, Z)) :-
	Y > 0, X < 4,
	Aux is X + Y,
	Aux >= 4,
	Z is Y - (4 - X).
```

```
% If there is water in the 4-gallon jug (X > 0) and there is room in the 3-gallon jug (Y < 3), THEN use it to fill up
% the 3-gallon jug until it is full (3-gallon jug = 3 in the new state) and leave the rest in the 4-gallon jug:
next_state((X, Y), (Z, 3)) :-
	X > 0, Y < 3,
	Aux is X + Y,
	Aux >= 3,
	Z is X - (3 - Y).

% There is something in the 3-gallon jug (Y > 0) and together with the amount in the 4-gallon jug it fits in the
% 4-gallon jug (Aux is X + Y, Aux =< 4), THEN fill it all (Y is 0 in the new state) into the 4-gallon jug (Z is Y + X):
next_state((X, Y), (Z, 0)) :-
	Y > 0,
	Aux is X + Y,
	Aux =< 4,
	Z is Y + X.

% There is something in the 4-gallon jug (X > 0) and together with the amount in the 3-gallon jug it fits in the
% 3-gallon jug (Aux is X + Y, Aux =< 3), THEN fill it all (X is 0 in the new state) into the 3-gallon jug (Z is Y + X):
next_state((X, Y), (0, Z)) :-
	X > 0,
	Aux is X + Y,
	Aux =< 3,
	Z is Y + X.

% Empty the 4-gallon jug IF it is not already empty (X > 0):
next_state((X, Y), (0, Y)) :-
	X > 0.

% Empty the 3-gallon jug IF it is not already empty (Y > 0):
next_state((X, Y), (X, 0)) :-
	Y > 0.

action((_, Y), (4, Y), fill1).
action((X, _), (X, 3), fill2).
action((_, Y), (4, Z), put(2, 1)) :- Y \= Z.
action((X, _), (Z, 3), put(1, 2)) :- X \= Z.
action((X, _), (Z, 0), put(2, 1)) :- X \= Z.
action((_, Y), (0, Z), put(2, 1)) :- Y \= Z.
action((_, Y), (0, Y), empty1).
action((X, _), (X, 0), empty2).

print([], _).

print([H | T], 0) :-
	write(start), tab(4), write(H), nl,
	print(T, H).

print([H | T], Prev) :-
	action(Prev, H, X),
	write(X), tab(4), write(H), nl,
```

```
print(T, H).
```

Output:-

```
% v:/CSMSS all/7th sem all notes/Ai notes/BFS.pl compiled 0.02 sec, 28 clauses
?- go.
start    0,0
fill2    0,3
put(2,1)    3,0
fill2    3,3
put(2,1)    4,2
true
```

Water Jug problem using DFS

Program:-

```
% Solve the Water Jug Problem using DFS

% solve_dfs(State, History FinalState):
% If the State is the FinalState, return an empty list of Moves.
solve_dfs(State, History, [], State) :- % Removed FinalState
    true. % Always succeeds

% solve_dfs(State, History FinalState):
% Move to the next state, update the history, and continue searching.
solve_dfs(State, History, [Move | Moves], FinalState) :-
    move(State, Move),
    update(State, Move, State1),
    legal(State1),
    not(member(State1, History)),
    solve_dfs(State1, [State1 | History], Moves, FinalState).

% Query to find a solution to the Water Jug Problem.
solve_water_jug_problem(FinalState, Moves) :- % Pass FinalState as an argument
    initial_state(jugs(0, 0)),
    solve_dfs(jugs(0, 0), [jugs(0, 0)], Moves, FinalState). % Use FinalState as the final goal

% Define the capacity of the jugs as constants.
capacity(1, 4).
capacity(2, 3).

% Define initial states.
initial_state(jugs(0, 0)).

% Define the legal states.
legal(jugs(V1, V2)) :- V1 >= 0, V2 >= 0.

% Define the available moves.
move(jugs(V1, V2), fill(1)) :- V1 < 4.
move(jugs(V1, V2), fill(2)) :- V2 < 3.
move(jugs(V1, V2), empty(1)) :- V1 > 0.
move(jugs(V1, V2), empty(2)) :- V2 > 0.
move(jugs(V1, V2), transfer(1, 2)) :- V1 > 0, V2 < 3.
move(jugs(V1, V2), transfer(2, 1)) :- V2 > 0, V1 < 4.

% Define how to update the state after a move.
update(jugs(V1, V2), fill(1), jugs(4, V2)).
update(jugs(V1, V2), fill(2), jugs(V1, 3)).
update(jugs(V1, V2), empty(1), jugs(0, V2)).
update(jugs(V1, V2), empty(2), jugs(V1, 0)).
update(jugs(V1, V2), transfer(1, 2), jugs(NewV1, NewV2)) :-
    Liquid is V1 + V2,
    (Liquid =< 3, NewV1 = 0, NewV2 = Liquid;
     Liquid > 3, NewV1 = Liquid - 3, NewV2 = 3).
update(jugs(V1, V2), transfer(2, 1), jugs(NewV1, NewV2)) :-
    Liquid is V1 + V2,
    (Liquid =< 4, NewV1 = Liquid, NewV2 = 0;
     Liquid > 4, NewV1 = 4, NewV2 = Liquid - 4).
```

```
% Adjust the liquid between the jugs.
adjust(Liquid, Excess, Liquid, 0) :- Excess =< 0.
adjust(Liquid, Excess, V, Excess) :- Excess > 0, V is Liquid - Excess.
```

OUTPUT:-

```
% v:/CSMSS all/7th sem all notes/Ai notes/DFS1.pl compiled 0.02 sec, 21 clauses
?- solve_water_jug_problem(jugs(2, 0), Moves).
Moves = [fill(1), fill(2), empty(1), transfer(2, 1), fill(2), transfer(2, 1), empty(1), transfer(2, 1)]
```

Eight Queen Problem

Program:-

```prolog
:- use_module(library(clpfd)).

n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- no_threat(Q, Qs, 1), safe_queens(Qs).

no_threat(_, [], _).
no_threat(Q1, [Q2|Qs]) :-
        Q1 #\= Q2,
        abs(Q1 - Q2) #\= D,
        D1 #= D + 1,
        no_threat(Q1, Qs, D1).

% Define a predicate to solve the N-Queens problem and print the solution.
solve_n_queens(N) :-
        n_queens(N, Qs),
        label(Qs),
        format('Solution for ~w-Queens: ~w~n', [N, Qs]).

% Example: solve the 8-Queens problem
:- solve_n_queens(8).
```

OUTPUT:-

```
% v:/CSMSS all/7th sem all notes/Ai notes/eight_queens.pl compiled 0.22 sec, 7 clauses
?- solve_n_queens(8).
Solution for 8-Queens: [1,5,8,6,3,7,2,4]
```

8-Puzzle Problem:

program :

```
% Simple Prolog Planner for the 8 Puzzle Problem
% This predicate initializes 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_sol(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('Solution Plan:'), nl,
write_sol(Plan).
solve(State, Goal, Sofar, Plan):-
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 a new position only if the destination tile is empty & Manhattan distance =
1Vaibhav Tawale[CS4256]
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 if the 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 the first list from the second to create the 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.
```

Output:

```
% v:/CSMSS all/7th sem all notes/Ai notes/puzzle.pl compiled 0.00 sec, 14 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(tile8,9)
at(tile7,8)
at(tile6,7)
at(tile5,6)
at(empty,5)
at(tile4,4)
at(tile3,3)
at(tile2,2)
at(tile1,1)
```

Robot traversal problem:

Program:

```
% Simple Prolog Planner for Robot Traversal Problem
% This predicate initializes 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 = [position(0, 0), direction(north)],
write_sol(Init),
Goal = [position(2, 3)],
nl, write('Goal state:'), nl,
write_sol(Goal), nl, nl,
solve(Init, Goal, Plan).
% Robot can move forward, backward, turn left, or turn right.
act(move_forward, [position(X, Y), direction(north)], [], [position(X, Y1), direction(north)]) :-
Y1 is Y + 1.
act(move_backward, [position(X, Y), direction(south)], [], [position(X, Y1), direction(south)]) :-
Y1 is Y - 1.
act(move_left, [position(X, Y), direction(west)], [], [position(X1, Y), direction(west)]) :-
X1 is X - 1.
act(move_right, [position(X, Y), direction(east)], [], [position(X1, Y), direction(east)]) :-
X1 is X + 1.
% Means-end analysis to determine actions needed to achieve the goal.
solve(State, Goal, Plan):-
solve(State, Goal, [], Plan).
solve(State, Goal, Plan, Plan):-
is_subset(Goal, State), nl,
write('Solution Plan:'), nl,
write_sol(Plan).
solve(State, Goal, Sofar, Plan):-
applicable(Action, State),
\+ member(Action, Sofar),
apply(Action, State, NewState),
solve(NewState, Goal, [Action|Sofar], Plan).
% Utility predicates.
% Check if the 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 the first list from the second to create the 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.
% Determine applicable actions based on the current state.
```

```
applicable(Action, State):-
act(Action, Preconditions, _, _),
is_subset(Preconditions, State).
% Apply an action to the current state to produce a new state.
apply(Action, State, NewState):-
act(Action, _, Delete, Add),
delete_list(Delete, State, Remainder),
append(Add, Remainder, NewState).
```

Output:

```
?- 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(tile8, 9)
at (tile7, 8)
at (tile6, 7)
at (tile5, 6)
at (empty,5)
at (tile4, 4)
at (tile3, 3)
at (tile2, 2)
at (tile1, 1)
```

Traveling Salesman Problem with Genetic Algorithm:

Program:

```
:- use_module(library(random)).
% Define the number of cities
num_cities(5).
% Define the population size for the GA
population_size(10).
% Define the mutation rate for the GA
mutation_rate(0.1).
% Define cities
city(0, 0).
city(1, 2).
city(3, 1).
city(4, 3).
city(2, 4).
% Generate a random route
generate_random_route(Route) :-
num_cities(NumCities),
length(Route, NumCities),
numlist(0, NumCitiesMinusOne, Cities),
random_permutation(Cities, Route).
% Calculate the total distance of a route
calculate_total_distance(Route, TotalDistance) :-
append(Route, [Route[0]], ClosedRoute), % Close the route
calculate_total_distance_helper(ClosedRoute, TotalDistance).
calculate_total_distance_helper([City1, City2 | Rest], TotalDistance) :-
city(City1, X1-Y1),
city(City2, X2-Y2),
DX is X1 - X2,
DY is Y1 - Y2,
Distance is sqrt(DX*DX + DY*DY),
calculate_total_distance_helper([City2 | Rest], RestDistance),
TotalDistance is Distance + RestDistance.
calculate_total_distance_helper([_], 0).
% Perform crossover between two parent routes to produce a child route
crossover(Parent1, Parent2, Child) :-
length(Parent1, Length),
random_between(1, Length, CrossoverPoint),
append(Prefix, Suffix, Parent1),
append(Prefix, RestParent2, Parent2),
append(RestParent2, Suffix, Child).
% Mutate a route by swapping two cities
mutate(Route, MutatedRoute) :-
mutation_rate(MutationRate),
(random_float < MutationRate ->
random_permutation(Route, MutatedRoute) ;
MutatedRoute = Route
).
% Create an initial population
initialize_population(StartCity, Population) :-
population_size(PopSize),
findall(Route, (between(1, PopSize, _), generate_initial_route(StartCity, Route)), Population).
generate_initial_route(StartCity, Route) :-
num_cities(NumCities),
length(Route, NumCities),
numlist(0, NumCitiesMinusOne, Cities),
random_permutation(Cities, ShuffledCities),
select(StartCity, ShuffledCities, Route).
% Evolutionary algorithm iteration
```

```
evolve_population([], []).
evolve_population([Parent1, Parent2 | Rest], [Child1, Child2 | NewPopulation]) :-
crossover(Parent1, Parent2, Child1),
crossover(Parent2, Parent1, Child2),
mutate(Child1, MutatedChild1),
mutate(Child2, MutatedChild2),
evolve_population(Rest, NewPopulation).
% Perform the GA iterations
ga_iteration(Population, NewPopulation) :-
evolve_population(Population, Children),
append(Population, Children, CombinedPopulation),
sort_population(CombinedPopulation, SortedPopulation),
take_best(SortedPopulation, PopulationSize, NewPopulation).
% Sort the population based on fitness (total distance)
sort_population(Population, SortedPopulation) :-
predsort(compare_fitness, Population, SortedPopulation).
compare_fitness(Order, Route1, Route2) :-
calculate_total_distance(Route1, Fitness1),
calculate_total_distance(Route2, Fitness2),
compare(Order, Fitness1, Fitness2).
% Take the best N individuals from the population
take_best([], _, []).
take_best(Population, N, BestPopulation) :-
length(Population, Length),
MaxN is min(N, Length),
take_best_helper(Population, MaxN, BestPopulation).
take_best_helper(_, 0, []).
take_best_helper([Individual | Rest], N, [Individual | BestRest]) :-
N > 0,
N1 is N - 1,
take_best_helper(Rest, N1, BestRest).
% Main function
tsp_genetic(StartCity, OptimalTour) :-
initialize_population(StartCity, Population),
ga_iterations(Population, max_generations, OptimalTour).% Perform GA iterations
ga_iterations(Population, 0, BestRoute) :-
find_best_route(Population, BestRoute).
ga_iterations(Population, GenerationsLeft, BestRoute) :-
ga_iteration(Population, NewPopulation),
NextGenerationsLeft is GenerationsLeft - 1,
ga_iterations(NewPopulation, NextGenerationsLeft, BestRoute).
% Print the total distance of the best route
print_total_distance(BestRoute) :-
calculate_total_distance(BestRoute, Fitness),
writeln('Total Distance:'),
writeln(Fitness).
```

Output:

```
v:/CSMSS all/7th sem all notes/Ai notes/tsp_genetic.pl compiled 0.02 sec, 30 clauses
- tsp_genetic(0, OptimalTour), print_best_route(OptimalTour), print_total_distance(OptimalTour
    Best Route:
    [0, 4, 1, 3, 2]

    Total Distance:
    10.472
```