1.Factorial Of A Number.

Problem definition:

Find out the factorial of a given number

Algorithm:

Algorithm Factorial(N)

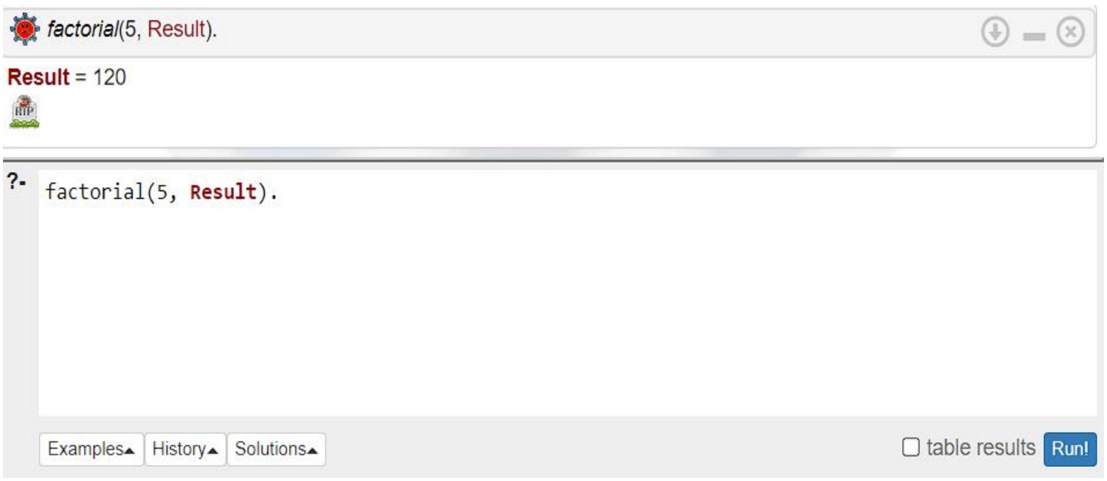
IF N = 0 THEN:
    Return 1
Else
    Return Factorial(N-1)*N

End If

- **Prolog Code** :

```
factorial(0, 1).
factorial(N, Result) :-
N > 0,
N1 is N - 1,
factorial(N1, SubResult),
Result is N * SubResult
```

- **Output:**



2. GCD of two numbers.

- **Problem definition**:
  Find out the Greatest Common Divisor of two numbers

- **Algorithm**:
  Input: Two integers A and B
  Output: An integer which is the GCD of A and B
  Function gcd(a, b)
  if a = 0
  return b
  end if
  while b ≠ 0
  if a > b
  a ← a – b
  else
  b ← b – a
  end if
  End while
  return a

- **Prolog Code** :

```
gcd(X,0,X).
gcd(X,Y,Z):-
R is mod(X,Y),
gcd(Y,R,Z).
```

- **Output:**

3.Prime Number.

- **Problem Definition:**
  Find Out Either e given number prime or not.

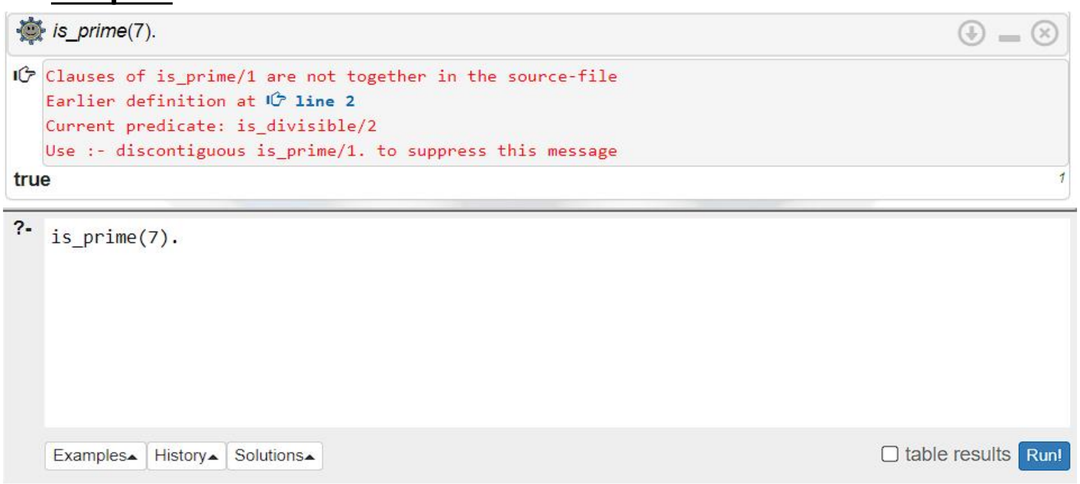
- **Algorithm:**

```
function isPrime(N):
  if N <= 1:
    return false

  for i from 2 to square root of N:
    if N is divisible by i:
      return false

  return true
```

- **Prolog Code:**

```
 % Base case: 0 and 1 are not prime
is_prime(0) :- false.
is_prime(1) :- false.

% Predicate to check if N is divisible by any number up to the square root of N
is_divisible(N, Div) :-
N > 1,
N1 is N - 1,
between(2, N1, Div),
0 is N mod Div.

% A number is prime if it's not divisible by any number other than 1 and itself
is_prime(N) :-
N > 1,
not(is_divisible(N, _)).
```

- **Output:**



4. Fibonacci series.

- **Problem definition:**

  Find out the Fibonacci series

- **Algorithm**:

Input: integer N such that N>0
Output: A sequence of integer numbers

Function Fib is

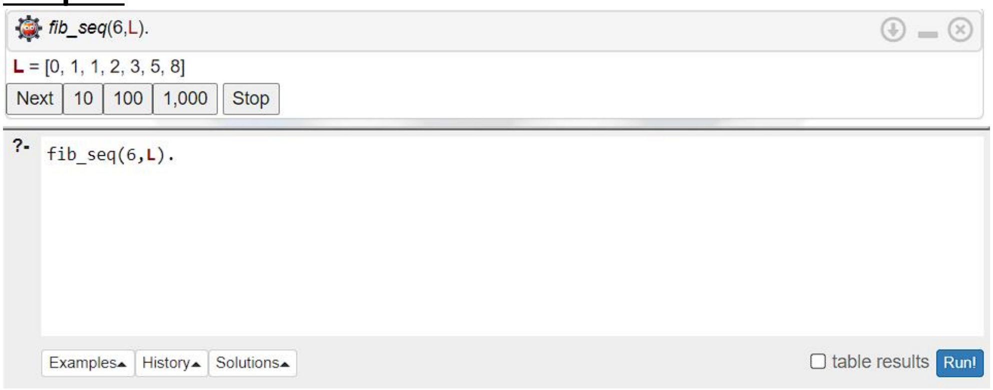
IF N=0 THEN:
return 0
ELSE if N=1
return 1
ELSE
return Fib(N-1)+Fib(N-2)
END IF

END Fib

- **Prolog Code** :

```
fib_seq(0,[0]).                % <- base case 1
fib_seq(1,[0,1]).              % <- base case 2
fib_seq(N,Seq) :-
  N > 1,
  fib_seq_(N,SeqR,1,[1,0]),     % <- actual relation (all other cases)
  reverse(SeqR,Seq).           % <- reverse/2 from library(lists)
fib_seq_(N,Seq,N,Seq).
fib_seq_(N,Seq,N0,[B,A|Fs]) :-
  N > N0,
  N1 is N0+1,
  C is A+B,
  fib_seq_(N,Seq,N1,[C,B,A|Fs]). % <- tail recursion
```

- **Output:**



5.Family Tree Problem.

- **Problem Definattion:**
  Execute Family Tree in Class.

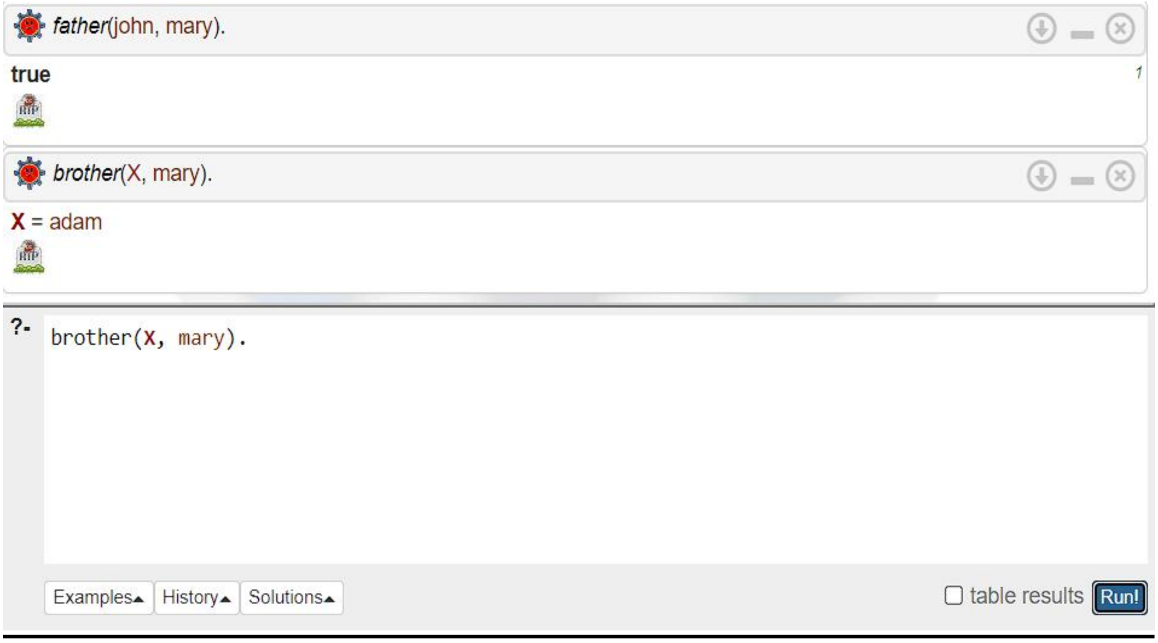
- **Algorithm:**

```
Facts:
   parent(john, mary).
   parent(john, adam).
   parent(eve, mary).
   parent(eve, adam).
   parent(mary, julie).
   parent(adam, peter).

Rules:
   father(Father, Child) :-
      parent(Father, Child),
      male(Father).

   mother(Mother, Child) :-
      parent(Mother, Child),
      female(Mother).

   sibling(Person1, Person2) :-
      parent(Parent, Person1),
      parent(Parent, Person2),
      Person1 \= Person2.

   brother(Brother, Person) :-
      sibling(Brother, Person),
      male(Brother).

   sister(Sister, Person) :-
      sibling(Sister, Person),
      female(Sister).

   grandparent(Grandparent, Grandchild) :-
      parent(Grandparent, Parent),
      parent(Parent, Grandchild).

Predicates for Genders:
   male(john).
   male(adam).
   male(peter).

   female(eve).
   female(mary).
   female(julie).
```

- **Prolog Code:**

```
% Facts defining the relationships in the family tree
parent(john, mary).
parent(john, adam).
parent(eve, mary).
parent(eve, adam).
parent(mary, julie).
parent(adam, peter).

% Rules for defining various relationships
father(Father, Child) :-
   parent(Father, Child),
   male(Father).

mother(Mother, Child) :-
   parent(Mother, Child),
   female(Mother).

sibling(Person1, Person2) :-
   parent(Parent, Person1),
   parent(Parent, Person2),
   Person1 \= Person2. % Ensuring they are not the same person

brother(Brother, Person) :-
   sibling(Brother, Person),
   male(Brother).

sister(Sister, Person) :-
   sibling(Sister, Person),
   female(Sister).

grandparent(Grandparent, Grandchild) :-
   parent(Grandparent, Parent),
   parent(Parent, Grandchild).

% Predicates for genders (can be extended with more details)
male(john).
male(adam).
male(peter).

female(eve).
female(mary).
female(julie).
```

- **Output:**



6.Towers Of Hanoi.

- **Problem definition:**

  Find out the solution of the famous towers of Hanoi problem

- **Algorithm:**

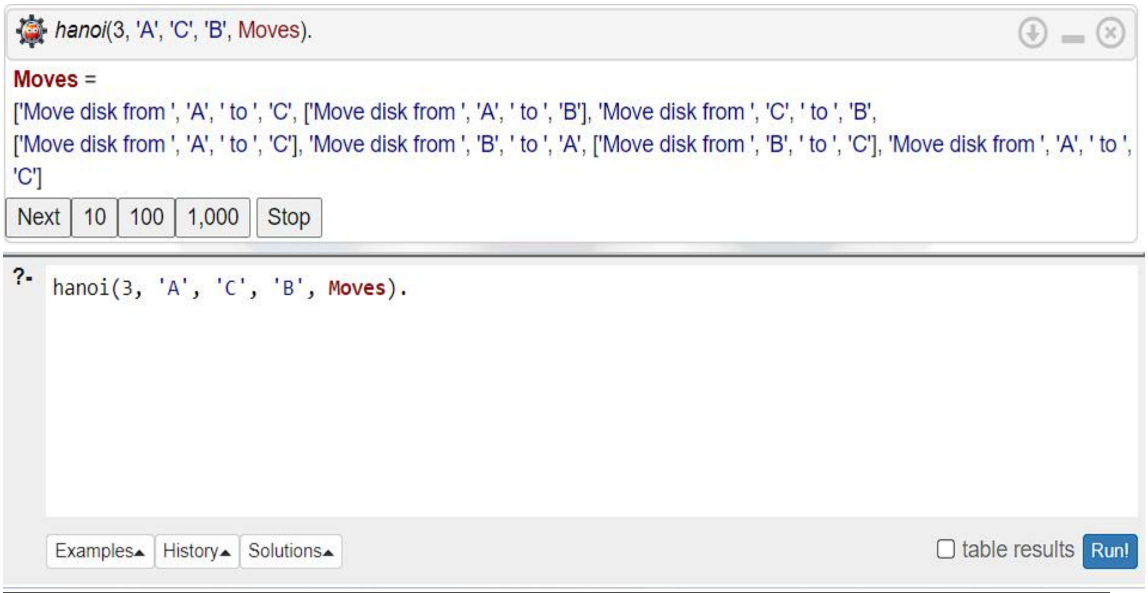
```
FUNCTION MoveTower(disk, source, dest, spare):
        IF disk == 0, THEN:
move disk from source to dest
        ELSE:
MoveTower(disk - 1, source, spare, dest)   // Step 1 above
        move disk from source to dest           // Step 2 above
MoveTower(disk - 1, spare, dest, source)   // Step 3 above
        END IF
```

- **Prolog Code** :

```
% Define the predicate for Towers of Hanoi
hanoi(1, Start, End, _ , Moves) :-
  % Base case: When there's only one disk, move it from Start to End
  append(['Move disk from ', Start, ' to ', End], [], Moves).
hanoi(N, Start, End, Aux, Moves) :-
  % Recursive case: Move N-1 disks from Start to Aux using End as the auxiliary peg
  N > 1,
  N1 is N - 1,
```

```
hanoi(N1, Start, Aux, End, FirstMoves),
% Move the largest disk from Start to End
append(['Move disk from ', Start, ' to ', End], [], Move),
% Move the N-1 disks from Aux to End using Start as the auxiliary peg
hanoi(N1, Aux, End, Start, LastMoves),
% Concatenate the moves
append(FirstMoves, [Move | LastMoves], Moves).
```

- **Output:**



7.Eight Queens.

- **Problem definition:**

  Find out the solution of the famous 8 queen problem

- **Algorithm:**

```
//initialize row, then find
//first acceptable solution for self and
//neighbor
//advance advance row and find next
//acceptable solution
//canAttack-see whether a position can
//be attacked by self or neighbors

//initialize


//row:current row number (changes)
//column:column number (fixed)
//neighbor:neighbor to left (fixed)


function queen.initialize(col, neigh)->boolean

/* initialize our column and neighbor values */

column := col
neighbor := neigh

/* start in row 1 */

row := 1
return findSolution;

end



function queen.findSolution ->boolean

/* test positions */

while neighbor.canAttack(row, column) do
if not self.advance then
```

```
        return false

/* found a solution */

return true

end


function queen.advance -> boolean

/* try next row */

if row<8 then begin
row :=  row + 1
return self.findSolution
end

/* cannot go further */
/* move neighbor to next solution */

if not neighbor.advance then
return false

/* start again in row 1 */

row := 1
return self.findSolution

end


procedure print
neighbor.print
write row, column
end



Function queen.catattack(testrow,testcolumn) ->boolean
/*test for same row*/
if row=testrow then
return true
        /*test diagonals*/
```

columndifference:=testcolum,n-column

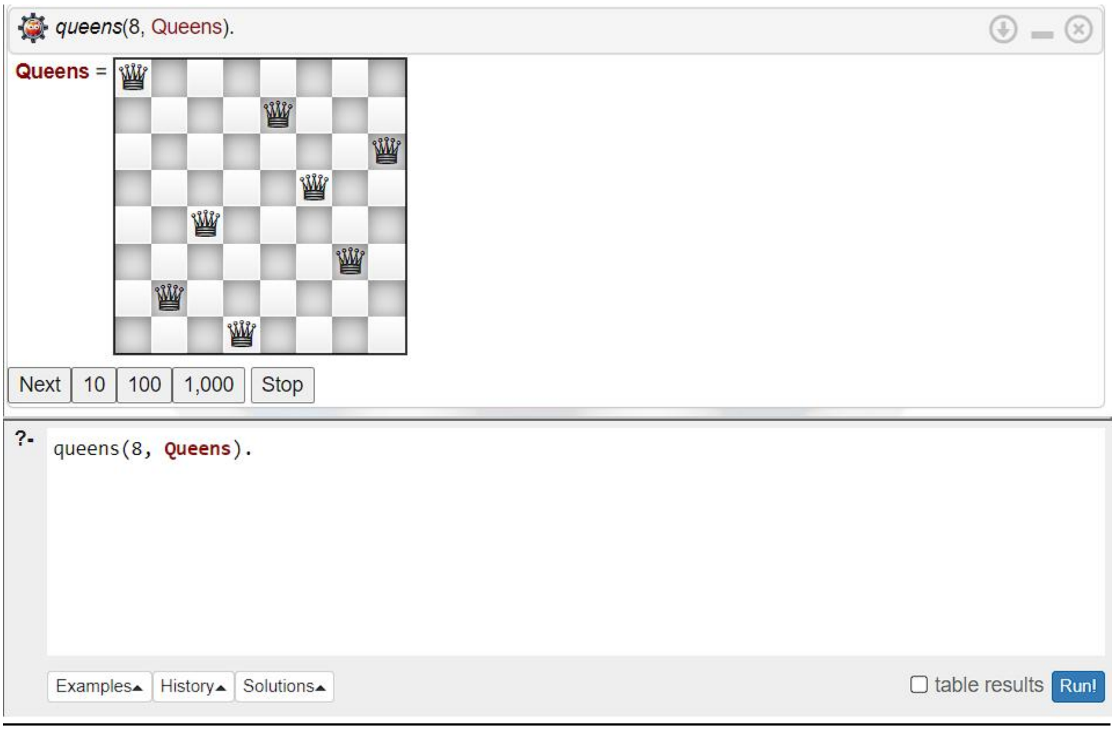
If (row+columndifference =testrow) or (row-columndifference=testrow) then
return true
/*we cant attack,see if neighbour can*/

return neighbour.canattack(testrow,testcolumn)

- **Prolog Code** :

```
% render solutions nicely.
:- use_rendering(chess).

%%   queens(+N, -Queens) is nondet.
%
%   @param          Queens is a list of column numbers for placing the queens.
%   @author Richard A. O'Keefe (The Craft of Prolog)

queens(N, Queens) :-
   length(Queens, N),
     board(Queens, Board, 0, N, _, _),
     queens(Board, 0, Queens).

board([], [], N, N, _, _).
board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
     Col is Col0+1,
     functor(Vars, f, N),
     constraints(N, Vars, VR, VC),
     board(Queens, Board, Col, N, VR, [_|VC]).

constraints(0, _, _, _) :- !.
constraints(N, Row, [R|Rs], [C|Cs]) :-
     arg(N, Row, R-C),
     M is N-1,
     constraints(M, Row, Rs, Cs).

queens([], _, []).
queens([C|Cs], Row0, [Col|Solution]) :-
     Row is Row0+1,
     select(Col-Vars, [C|Cs], Board),
     arg(Row, Vars, Row-Row),
     queens(Board, Row, Solution).


/** <examples>

?- queens(8, Queens).

*/
```

- **Output:**


queens(8, Queens).
Queens =
Next
10
100
1,000
Stop
?-
queens(8, Queens).
Examples
History
Solutions
table results
Run!

8.Water Jug Problem.

Problem Definition:

You are given two empty jugs with capacities **Jug1Capacity** and **Jug2Capacity** in units of water. The goal is to measure a specific quantity of water **TargetQuantity** using these jugs. The following actions can be performed:

- Fill a jug completely.
- Empty a jug completely.
- Pour water from one jug into another until the pouring jug is empty or the receiving jug is full.

The problem is to determine whether it's possible to measure the TargetQuantity of water using the given jugs and, if possible, provide the sequence of actions to achieve this goal.

- **Algorithm:**

function WaterJugProblem(Jug1Capacity, Jug2Capacity, TargetQuantity):
initialize a stack to store states (initially empty)
initialize a set to store visited states (initially empty)
push initial state (0, 0) to the stack (representing both jugs empty)

while the stack is not empty:
current_state = pop state from the stack
if current_state matches the TargetQuantity:
return sequence of actions to achieve the TargetQuantity

if current_state is not in visited states:
mark current_state as visited

for each possible action in Fill Jug 1, Fill Jug 2, Empty Jug 1, Empty Jug 2,
Pour Jug 1 to Jug 2, Pour Jug 2 to Jug 1:
new_state = apply action to current_state (simulate the action)
if new_state is valid and not visited:
push new_state to the stack

return "TargetQuantity cannot be achieved with the given jugs"

function apply action to current_state:
simulate the action on current_state to generate a new_state

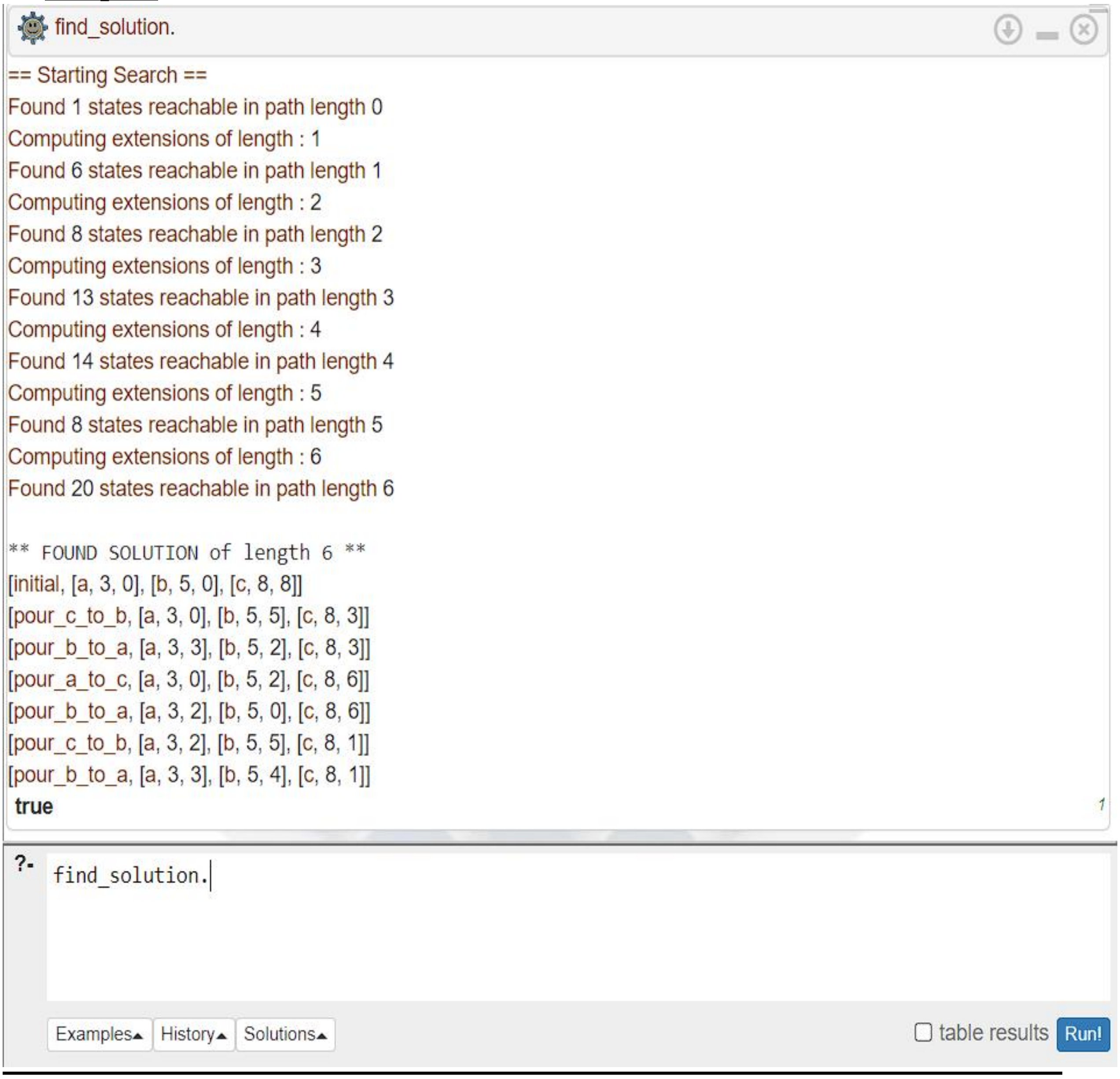
return new_state if the action is feasible (within jug capacities and non-negative values)
otherwise, return an invalid state

- **Prolog Code:**

```
:- 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:



```
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

?- find_solution.
```

9. Missionaries and Cannibals problem.

- **Problem Definition:**
- There are three missionaries and three cannibals on one side of the river.
- A boat is available that can carry at most two individuals at a time.
- The goal is to move all the missionaries and cannibals to the other side of the river without violating the following constraints:
- At any point, the number of cannibals on either side of the river cannot exceed the number of missionaries, otherwise, the cannibals would overpower and eat the missionaries.
- The boat must always have at least one person (missionary or cannibal) to operate it.

- **Algorithm:**

```
function solveMissionariesAndCannibals():
initialize an empty stack to store states
push initial state (3, 3, 0) to the stack
initialize an empty set to store visited states
add initial state (3, 3, 0) to the visited set

while stack is not empty:
current_state = pop state from stack
if current_state is the goal state (0, 0, 1):
return sequence of actions to reach the goal state

for each possible action in MoveOne, MoveTwo, MoveOneBack, MoveTwoBack:
new_state = apply action to current_state
if new_state is valid and not in visited states:
push new_state to stack
add new_state to visited states

return "No solution found"

function apply action to current_state:
simulate the action on current_state to generate a new_state

return new_state if the action is feasible (adheres to constraints)
otherwise, return an invalid state
```

- **Prolog Code:**

```
% Represent a state as [CL,ML,B,CR,MR]
start([3,3,left,0,0]).
goal([0,0,right,3,3]).

legal(CL,ML,CR,MR) :-
% is this state a legal one?
ML>=0, CL>=0, MR>=0, CR>=0,
(ML>=CL ; ML=0),
(MR>=CR ; MR=0).

% Possible moves:
move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
% Two missionaries cross left to right.
MR2 is MR+2,
ML2 is ML-2,
legal(CL,ML2,CR,MR2).

move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
% Two cannibals cross left to right.
CR2 is CR+2,
CL2 is CL-2,
legal(CL2,ML,CR2,MR).

move([CL,ML,left,CR,MR],[CL2,ML2,right,CR2,MR2]):-
%  One missionary and one cannibal cross left to right.
CR2 is CR+1,
CL2 is CL-1,
MR2 is MR+1,
ML2 is ML-1,
legal(CL2,ML2,CR2,MR2).

move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
% One missionary crosses left to right.
MR2 is MR+1,
ML2 is ML-1,
legal(CL,ML2,CR,MR2).

move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
% One cannibal crosses left to right.
CR2 is CR+1,
CL2 is CL-1,
legal(CL2,ML,CR2,MR).
```

```
move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
% Two missionaries cross right to left.
MR2 is MR-2,
ML2 is ML+2,
legal(CL,ML2,CR,MR2).

move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
% Two cannibals cross right to left.
CR2 is CR-2,
CL2 is CL+2,
legal(CL2,ML,CR2,MR).

move([CL,ML,right,CR,MR],[CL2,ML2,left,CR2,MR2]):-
%  One missionary and one cannibal cross right to left.
CR2 is CR-1,
CL2 is CL+1,
MR2 is MR-1,
ML2 is ML+1,
legal(CL2,ML2,CR2,MR2).

move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
% One missionary crosses right to left.
MR2 is MR-1,
ML2 is ML+1,
legal(CL,ML2,CR,MR2).

move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
% One cannibal crosses right to left.
CR2 is CR-1,
CL2 is CL+1,
legal(CL2,ML,CR2,MR).


% Recursive call to solve the problem
path([CL1,ML1,B1,CR1,MR1],[CL2,ML2,B2,CR2,MR2],Explored,MovesList) :-
move([CL1,ML1,B1,CR1,MR1],[CL3,ML3,B3,CR3,MR3]),
not(member([CL3,ML3,B3,CR3,MR3],Explored)),
path([CL3,ML3,B3,CR3,MR3],[CL2,ML2,B2,CR2,MR2],[[CL3,ML3,B3,CR3,MR3]|Explored],[
 [[CL3,ML3,B3,CR3,MR3],[CL1,ML1,B1,CR1,MR1]] | MovesList ]).

% Solution found
path([CL,ML,B,CR,MR],[CL,ML,B,CR,MR],_,MovesList):-
output(MovesList).
```

```
% Printing
output([]) :- nl.
output([[A,B]|MovesList]) :-
output(MovesList),
write(B), write(' -> '), write(A), nl.

% Find the solution for the missionaries and cannibals problem
find :-
path([3,3,left,0,0],[0,0,right,3,3],[[3,3,left,0,0]],_).
```

- **<u>Output:</u>**

10.Breadth First Search.

- **Problem Definition:**
  Given a tree or graph, perform a Breadth-First Search to traverse the structure and visit all nodes in a breadthward motion.

- **Algorithm:**

```
function BFS(graph, startNode):
create an empty queue
create an empty set to keep track of visited nodes

enqueue startNode into the queue
add startNode to the set of visited nodes

while the queue is not empty:
currentNode = dequeue from the queue
process currentNode (visit or perform required operation)

for each neighbor of currentNode:
if neighbor is not visited:
enqueue neighbor into the queue
add neighbor to the set of visited nodes
```

- **Prolog Code:**

```
% solve( Start, Solution):
%    Solution is a path (in reverse order) from Start to a goal

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

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

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

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

extend( [Node | Path], NewPaths)  :-
```

```
bagof( [NewNode, Node | Path],
( s( Node, NewNode), \+ member( NewNode, [Node | Path] ) ),
NewPaths),
!.

extend( Path, [] ).          % bagof failed: Node has no successor
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
s(c,g).
s(d,h).
s(e,i).
s(e,j).
goal(j).
goal(f).
```

- <u>**Output:**</u>

11.Depth First Search.

- **Problem Definition:**
  Given a tree or graph, perform a Depth-First Search to traverse the structure and visit all nodes in a depthward motion.

- **Algorithm:**

```
function DFS(Graph, StartNode):
create an empty stack to store nodes to be explored
create an empty set to keep track of visited nodes

push StartNode onto the stack
add StartNode to the set of visited nodes

while the stack is not empty:
currentNode = pop from the stack
process currentNode (visit or perform required operation)

for each neighbor of currentNode:
if neighbor is not visited:
push neighbor onto the stack
add neighbor to the set of visited nodes
```

- **Code:**

```
% solve( Node, Solution):
%    Solution is an acyclic path (in reverse order) between Node and a goal

solve( Node, Solution)  :-
depthfirst( [], Node, Solution).

% depthfirst( Path, Node, Solution):
%   extending the path [Node | Path] to a goal gives Solution

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

depthfirst( Path, Node, Sol)  :-
s( Node, Node1),
\+ member( Node1, Path),              % Prevent a cycle
depthfirst( [Node | Path], Node1, Sol).

depthfirst2( Node, [Node], _)  :-
goal( Node).
```

```
depthfirst2( Node, [Node | Sol], Maxdepth) :-
Maxdepth > 0,
s( Node, Node1),
Max1 is Maxdepth - 1,
depthfirst2( Node1, Sol, Max1).

goal(f).
goal(j).
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
s(c,g).
s(d,h).
s(e,i).
s(e,j).
```

- **<u>Output:</u>**