Example :- Water Jug Problem

Consider the following problem : you are given two jugs a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug and the ground on which water may be poured. Neither jug has any measuring marketing on it. How can you get exactly 2 gallons of water in the 4-gallon jug ?

State Representation & Initial State - we will represent a state of the problem as a tuple (x, y) the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$ our initial state (0,0)

Goal Predicate :-

State = (7, 4) where $0 \leq y \leq 3$

Operators :

we must define a set of operator that will take us from one state to another.

1. Fill 4-gal Jug $(x,y) \longrightarrow (4,y)$ where $x<4$

2. Fill 3-gal Jug $(x,y) \longrightarrow (x,3)$ where $y<3$

3. Empty 4-gal jug on ground $(x,y) \longrightarrow (0,y)$ where $x>0$

4. Empty 3-gal jug on ground $(x,y) \longrightarrow (x,0)$ where $x>0$

5. pour water from 3-gal Jug to fill 4-gal jug $(x,y) \longrightarrow (4, y-(4-x))$ where $0<x+y\geq 4$ & $4>0$

6. pour water from 4-gal Jug to fill 3-gal jug $(x,y) \longrightarrow (x-(3-y), 3)$ where $0<x+4\geq 3$ & $x>0$

7. pourall of water from 3-gal jug into 4-gal jug $(x,y) \longrightarrow (x+y, 0)$ where $0<x+y\geq 4$ and $4\geq 0$

8. pourall of water from 4-gal jug into 3-gal jug $(x,y) \longrightarrow (0, x+y)$ where $0<x+y\leq 3$ and $x\geq 0$

Through Graph Search, the following Solution is found

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	
4	0	1. Fill 4
1	3	6. pour 4 into 3-fill
1	0	4. Empty 3
0	1	8. pour all 4 into 3
4	1	1. fill 4
2	3	6. pour into 3

Date :- 22-08-2023

Experiment - 1

SHEET No. 4

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

```
Solve_dfs (state, History, []) :- final_state (state)
Solve_dfs (state, History, [Move | Moves]) :-
        Move (state, move),
        update (State, Move, State 1),
        legal (state 1),
        not (member (state 1, History)),
        Solve_dfs (state 1, [state | History], Moves)
test_dfs (problem, Moves) :-
        initial_state (problem, state), (solve_dfs [state], moves)
        Capacity (1, 10),
        Capacity (2, 7)
        initial_state (jugs, jugs (0, 0)).
        final-state (jugs, (0, 6)).
        % final-state (jugs (4, 0))
        legal (jugs (V1, V2))
        Move (jugs (V1, V2), Fill (1)) :- Capacity (1, C1), V1 < C1
                                          Capacity (2, C1), V2 < C2
```

Moves (jugs (V1, V2) FILL (2)) :- Capacity (2, C2), V2 < C2,
Capacity (2, C2), V1 < C1

Move (jugs (V1, V2), empty(1)) :- V1 > 0

Move (jugs (V1, V2), empty (2)) :- V2 > 0

Move (jugs (V1, V2), transfer (1, 2))

Move (jugs (V1, V2), transfer (2, 1))

adjust (Liquid, Excess, Liquid, 0) :- Excess = < 0

adjust (Liquid, Excess, V, Excess) :- Excess > 0, V is Liquid - Excess

update (jugs (V1, V2), FILL (1), jugs (C1, V2)) :- Capacity (1, C1)

update (jugs (V1, V2), FILL (2), jugs (V1, C2)) :- Capacity (2, C2)

update (jugs (V1, V2), empty (1), jugs (0, V2))

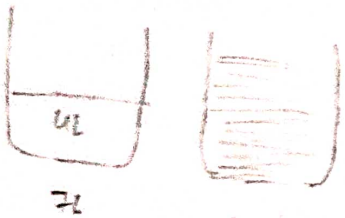
update (jugs (V1, V2), empty (2), jugs (V1, 0))

update (jugs (V1, V2), transfer (1, 2), jugs (New V1, New V2)) :-
Capacity (2, C2),
Liquid & is V1 + V2
Excess is Liquid - C2
adjust (Liquid, Excess, New V2, New V1)

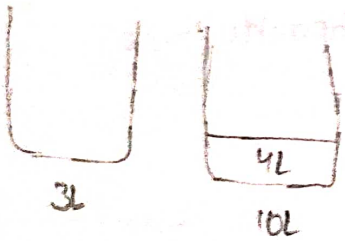
update (jugs (V1, V2), transfer (2, 1), jugs (New V1, New V2)) :-

Step :-1

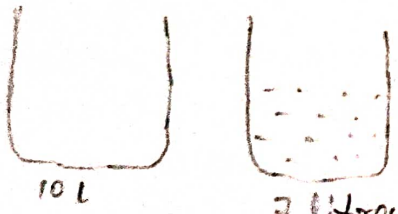
Step :-2 : Fill 7 and pour in 10L



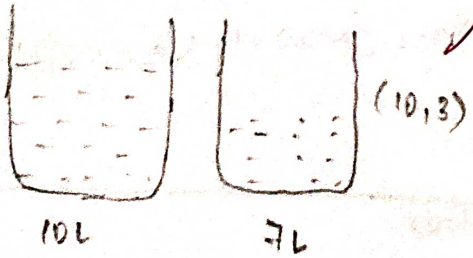
Step :-3 Empty 10L and pour in 10L



Step :-4 : till 7 again pour in 10L

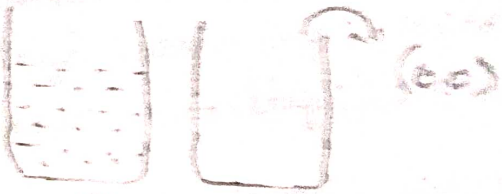


Step 5 :-

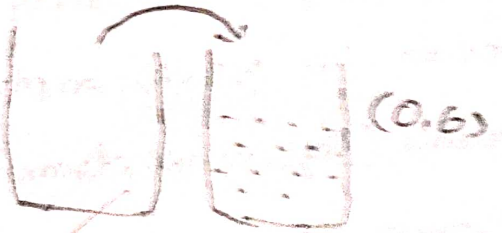


Step 6 :-

Step 7 :-



Step 8 :-



Capacity (1,c1).

Liquid is V1+V2.

Excess is Liquid - c1.

adjust (Liquid, excess, New V1, New V2)

Output

? - test-dfs (Jugs, Moves)

Moves = [fill(1), transfer(1,2), empty(2), transfer(1,2), fill(1), transfer(1,2)
empty(2), transfer(1,2)];

Moves = [fill(1), transfer(1,2), empty(2), transfer(1,2), fill(1), transfer(1,2)
empty(2), transfer(2,1)];

Moves = [fill(1), transfer(1,2), empty(2), transfer(2,1), fill(1), transfer(1,2)
empty(2), transfer(1,2)];

Moves = [Fill(1), transfer(1,2), empty(2), transfer(2,1), fill(1), transfer(1,2)
empty(2), transfer(2,1)];

Moves = [fill(1), transfer(2,1) transfer(1,2) empty(2), transfer(2,1), fill(1)
transfer(2,1), transfer(1,2), empty(...)|...];

Moves = [fill(1), transfer(2,1), transfer(1,2), empty(2), transfer(2,1), fill(1),
transfer(2,1), transfer(1,2), empty(...)|...];

false

BS
4/9/23

2) Implementation of BFS for tic-tac-toe problem using Prolog.

```
+
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" games
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,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]
% predicate to Support playing a game with the user.
X_can_win_in_one(Board):- move(Board, X, Newboard), win(Newboard, X)
% from the current board
Orespond(Board, Newboard):-
        move(Board, O, Newboard),
        win(Newboard, O),
        !.
Orespond(Board, Newboard):-
        move(Board, O, Newboard).
Orespond(Board, O, Newboard):-
```

```
        not(member(b,Board))
        !,
        write('cats game!'),nl,
        Newboard=Board.
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,G,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,F,G,H,b],6,[A,B,C,D,E,f,G,H,x]).
playo :-explain,playform([b,b,b,b,b,b,b,b,b]).
explain:-
    write('You play x by entering integer positions follows by a
          period,'),nl.
    display([1,2,3,4,5,6,7,8,9]).
playform(Board):-win(Board,x),write('You win!')
playform(Board):-win(Board,0),write('I win!').
x move(Board,N,Newboard),
display(Newboard),
display(Newnewboard),
playform(Newnewboard).
```

Output:-1

You play x by entering integer positions followed by a period

[1,2,3]

[4,5,6]

[7,8,9]

1:9.

[b,b,b]

[b,b,b]

[b,b,x]

[o,b,b]

[b,b,b]

[b,b,x]

1:

[o,b,b]

[b,b,b]

[ox,b,x]

[o,b,b]

[b,b,b]

[x,o,x]

1:3

[o, b, x]

[b, b, b]

[x, o, x]

[o, o, x]

[b, b, b]

[x, o, x]

1:6

[o, o, x]

[b, b, x]

[x, o, x]

[o, o, x]

[b, o, x]

[x, o, x]

You win!

true

ಉದಾ: 3 (1)

Date :- 14-09-2017

SHEET No. 12

Experiment :-3

Implementation of TSP using heuristic approach using prolog.

Program

```
road(guntur,tenali,9)
road(vijayawada,tenali,3)
road(vijayawada,tenali,6)
road(vijayawada,chirala,5)
road(chirala,ongole,5)
road(ongole,vijayawada,4)
get_road(Start,End,visited,Result):-
    get road(Start,End,[Start],0,visited,Result):
get_road(Start,End,waypoints,Distance,Acc,visited,Total Distance]:-
    road (Start,End,Distance),
    reverse([End|waypoints],visited),
    Total Distance is Distance Acc+Distance
get_road(Start,End,waypoints,Distance,Acc,visited Total Distance):-
    road(Start,waypoint,Distance),
    \+ member(waypoint,waypoint)
    New Distance Acc is Distance Acc+Distance,
    get.road(waypoints,End,[waypoint|waypoints],New Distance
            Acc,visited,Total,Distance)
```

Output

?- get_road (guntur, tenali, visited, Result).
visited = [guntur, tenali],
Result = 9
?- get_road (vijayawada, Ongole, visited, Result)
visited = (vijayawada, chirala, Ongole)
Result = 10

23/9/23

Date:- 25-09-2023

SHEET No. 14

Experiment - 6

Implementation of monkey banana problem using prolog

The monkey and banana problem is often used as a simple example of problem solving. Our prolog program for this problem will show how the mechanisms of matching and back tracking

There is a monkey at the door into the room. In the middle of the room a banana is hanging from the celing. The monkey is hungry and wants to get the banana but he cannot stretch high enough from the enough. All the windows of the room there is a box the monkey may use. The monkey can the climb the box. push the box around (if it is ready at box) and group the banana if standing on the box directly under the banana can be monkey get banana

* from example the initial state of the woorld is determined.

1) Monkey is at door
2) Monkey is on floor
3) Box is at window

First the goal of game is a situated in which the monkey has the banana that is any state in which is last component is 'has'!

State (-, -, -, has)

The initial State of the monkey world represented as a structured object. The three components are: horizonal position of monkey, vertical position of monkey position of box, monkey has not the banana

There are four types of moves

1) Grasp banana
2) Climb box,
3) Push box,
4) walk around

The these arguments of the relations Specify move thus

State 1 ---> State 2

State 1 is the State before the move, M is the move executed & State 2 is the State after the move

The move 'grasp' with the necessary precondition on the State before the move can be defined by the caluse:

move (State (middle, onbox, middle, hasnot) %. Before move grasp
State (middle, on box, middle, has) %. After move

This fact Says that the move the monkey has the banana and he has remained on the box in the middle of room

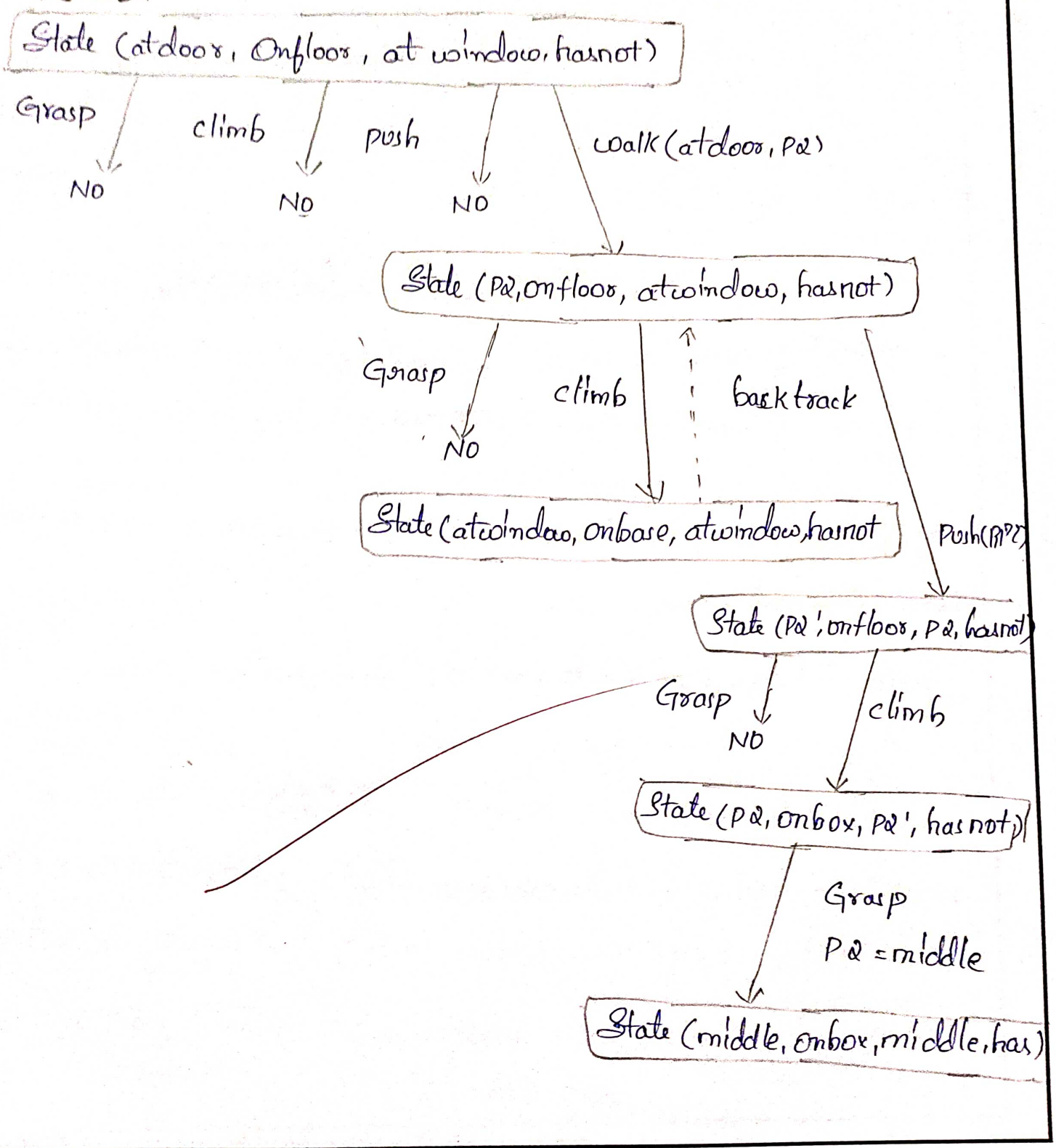
In a similar way we can express the fact the monkey on the floor can walk from the horizontal position P2. The monkey can do. All this can be defined by following predicate fact:

move(State(P1, on-floor, B, H)
    walk(P1, P2)
    State(P2, on-floor, B, H)

move M

Canget Canget Recursive formulation of canget

- The box is at some point B which remained the same after;
- The has banana status remains the same after the move


State (atdoor, Onfloor, at window, hasnot)
Grasp
climb
push
walk (atdoor, P2)
NO
NO
NO
State (P2, onfloor, atwindow, hasnot)
Grasp
climb
back track
NO
State (atwindow, onbase, atwindow, hasnot
Push(P1, P2)
State (P2', onfloor, P2, hasnot)
Grasp
climb
NO
State (P2, onbox, P2', has not)
Grasp
P2 = middle
State (middle, onbox, middle, has)

Output

?- canget (state (atdoor, onfloor, atwindow, hasnot)).

true

?- trace

true

[trace] ?- canget (state (atdoor, onfloor, atwindow, hasnot)).

Call: (10) canget (state (atdoor, onfloor, atwindow, hasnot)) ? creep

Call: (11) move (state (atdoor, onfloor, atwindow, hasnot), _20088, _20028) ? creep

Exit: (11) move (state (atdoor, onfloor, atwindow, hasnot), walk (atdoor, -20792), state (-20792, onfloor, hasnot)) ? creep

Fail: (13) move (state (atwindow, onbox, atwindow, hasnot), _25408, 24590) ? creep

Fail: (12) canget (state (atwindow, onbox, atwindow, hasnot)) ? creep

Call: (12) canget (state (-27620, onfloor, -27620, hasnot)) ? creep

Call: (13) canget (state (-27620, onbox, -27620, hasnot)) ? creep

Call: (14) move (state (-27620, onbox, -27620, hasnot), -29200, 29140) ? creep

Exit : (14) move (state (middle, onbox, middle, hasnot), grasp, state (middle, onbox, middle, has)) ? creep

Call : (14) Canget (state (middle, onbox, middle, has)) ? creep

Exit (13) Canget (state (middle, onfbox, middle, hasnot)) ? creep

Exit : (11) Canget (state (atwindow, onfloor, atwindow, hasnot)) ? creep

Exit (10) Canget (state (atdoor, onfloor, atwindow, hasnot)) ? creep

20/9/23