**COMPUTER SCIENCE ENGINEERING(AI&ML)**

**DEPARTMENT OF COMPUTER SCIENCE**

**&**

**ENGINEERING(AI&ML)**

**ARTIFICIAL INTELLIGENCE LAB MANUAL**



**SAI SPURTHI INSTITUTE OF TECHNOLOGY**

Accredited by NACC-‘B’ Grade

B.Gangaram

Sathupally- 507303, Khammam Dist,Telangana

|  |
| --- |
| COMPUTER SCIENCE ENGINEERING (AI&ML)    SAI SPURTHI INSTITUTE OF TECHNOLOGY  B.GANGARAM  SATHUPALLY-507303,Khammam Dist    C E R T I F I C A T E  *This is certify that it is a bonafide record of practical*  *Work of Mr / Miss……………………………………………*  *Roll No…………of……………. B.Tech ……..………….sem.*  *In the Department of…………………….Engineering,*  *In………………………………….Laboratory during the*  *academic year………………………………………….*    *No.ofExperiments Conducted---------- No. of Experiments Attended………*    *Signature of the Staff Member Signature of Head of the Dept*  *Signature of the External Examinar* |

**List of Experiments (AI)**

1) Write a program in prolog to implement simple facts and Queries

2) Write a program in prolog to implement simple arithmetic

3) Write a program in prolog to solve Monkey banana problem

4) Write a program in prolog to solve Tower of Hanoi

5) Write a program in prolog to solve 8 Puzzle problems

6) Write a program in prolog to solve 4-Queens problem

7) Write a program in prolog to solve Traveling salesman problem

8) Write a program in prolog for Water jug problem

**Experiment :1**

Write a program in prolog to implement simple facts and Queries.

**THEORY:**

Now that we have some facts in our Prolog program, we can consult the program in the listener

and query, or call, the facts. This chapter, and the next, will assume the Prolog program

contains only facts. Queries against programs with rules will be covered in a later chapter.

Prolog queries work by pattern matching. The query pattern is called a goal. If there is a fact

that matches the goal, then the query succeeds and the listener responds with 'yes.' If there is

no matching fact, then the query fails and the listener responds with 'no.'

Prolog's pattern matching is called unification. In the case where the logicbase contains only

facts, unification succeeds if the following three conditions hold.

● The predicate named in the goal and logicbase are the same.

● Both predicates have the same arity.

● All of the arguments are the same.

**CODE:**

food(burger).

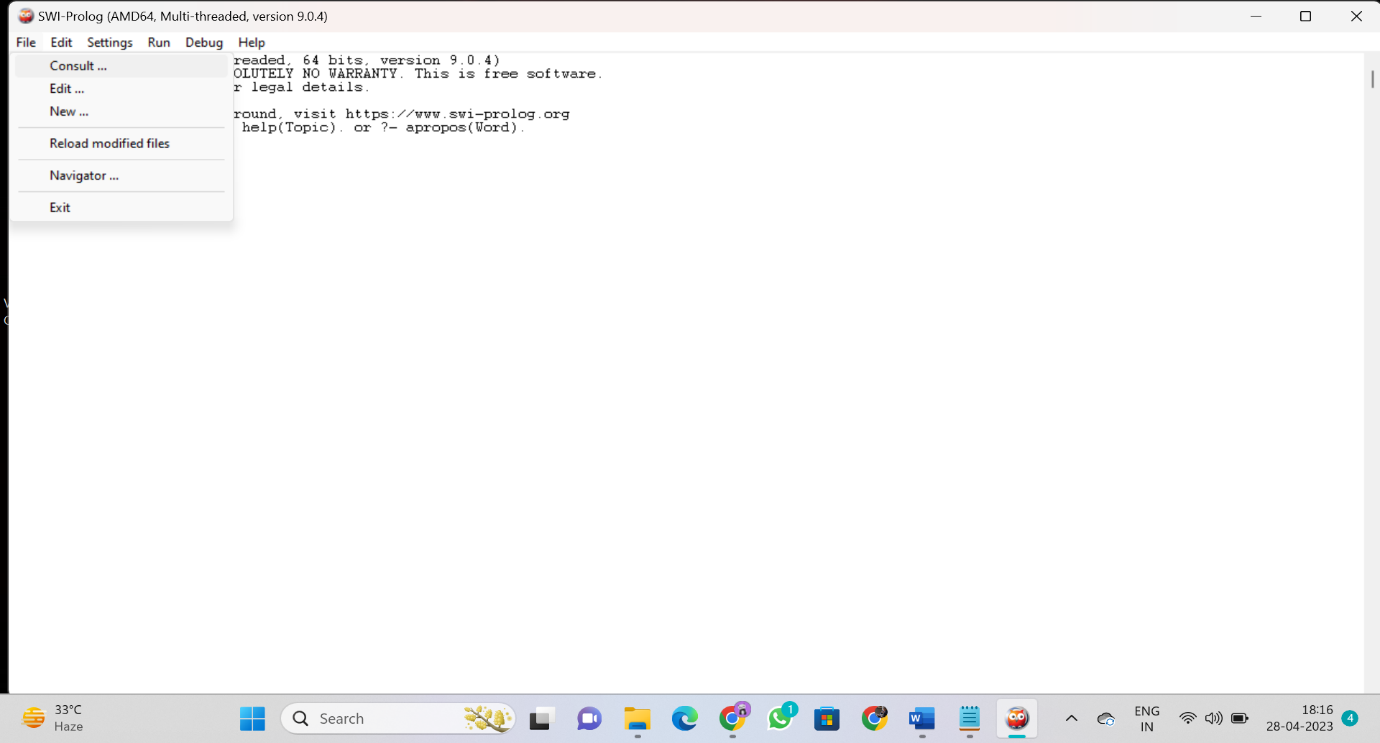
food(sandwich).

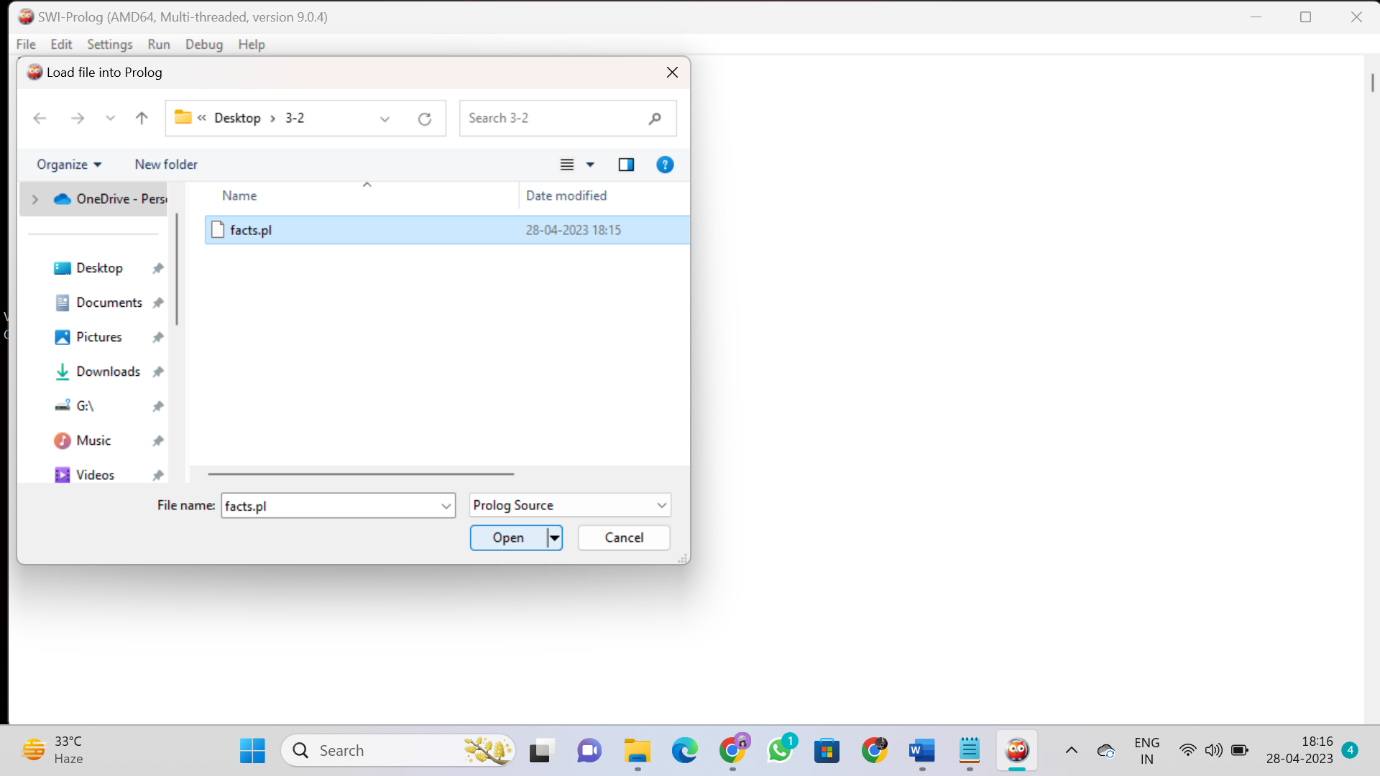
food(pizza).

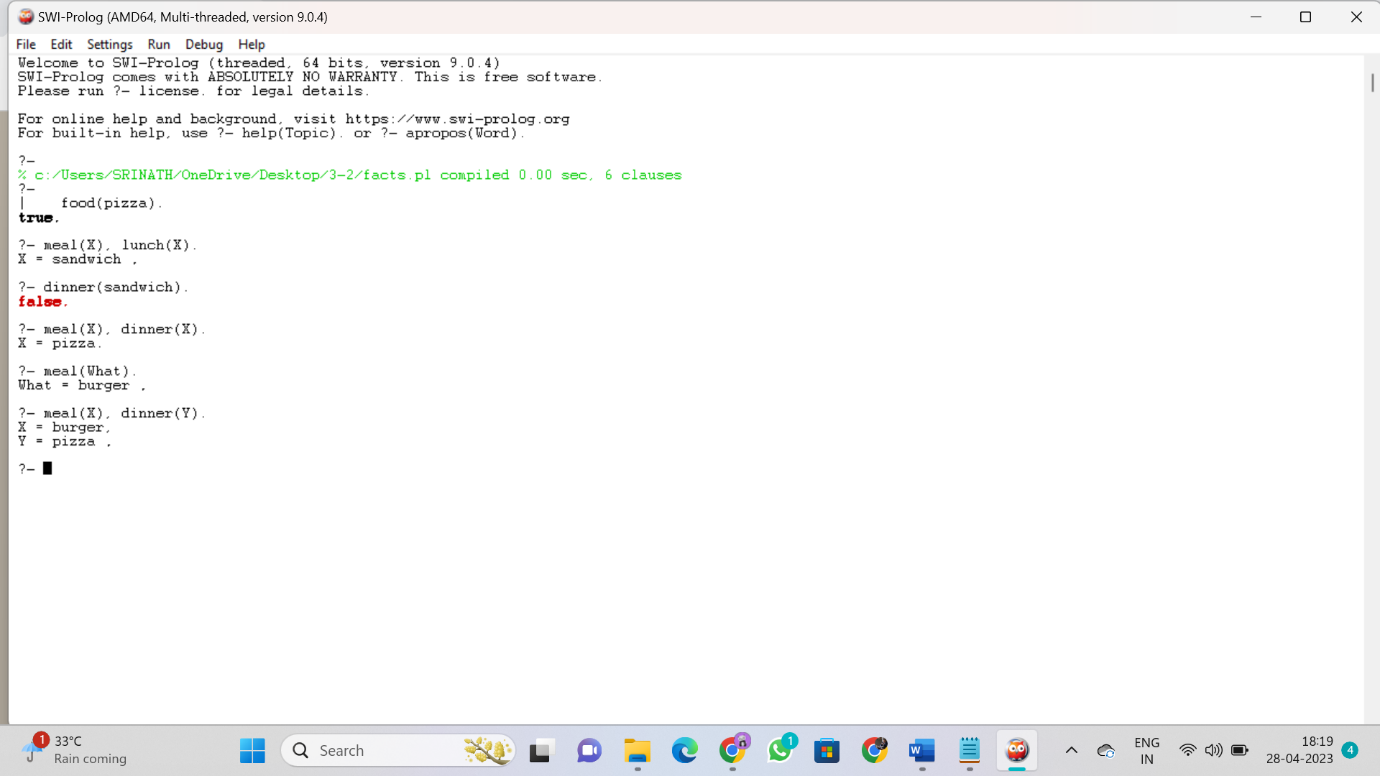
lunch(sandwich).

dinner(pizza).

meal(X) :- food(X).







**OUTPUT:**

?-

| food(pizza).

true.

?- meal(X), lunch(X).

X = sandwich .

?- dinner(sandwich).

false.

?- meal(X), dinner(X).

X = pizza.

?- meal(What).

What = burger .

?- meal(X), dinner(Y).

X = burger,

Y = pizza .

**Experiment :2**

Write a program in prolog to implement simple arithmetic

**THEORY:**

Prolog is not the programming language of choice for carrying out heavy-duty mathematics. It does, however, provide arithmetical capabilities. The pattern for evaluating arithmetic

expressions is (where Expression is some arithmetical expression)

X is Expression

The variable X will be instantiated to the value of Expression. For example,

?-X is 10+5.

X=15 ?

Yes

It is important to note that Expression is not evaluated by itself. You have to supply a variable (followed by the infix operator is/2) to collect a result.

Other pre-defined Prolog arithmetic infix operators are

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| X > Y | X is greater than Y |
| X < Y | X is less than Y |
| X >= Y | X is greater than or equal to Y |
| X =< Y | X is less than or equal to Y |
| X =:= Y | the X and Y values are equal |
| X =\= Y | the X and Y values are not equal |

You can see that the ‘=<’ operator, ‘=:=’ operator and ‘=\=’ operators are syntactically different from other languages.

**Arithmetic Operators in Prolog:**

Arithmetic operators are used to perform arithmetic operations. There are few different types of arithmetic operators as follows –

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| \*\* | Power |
| // | Integer Division |
| mod | Modulus |

Let us see one practical code to understand the usage of these operators.

**Code:**

add(X, Y, Z) :-

Z is X + Y.

subtract(X, Y, Z) :-

Z is X - Y.

multiply(X, Y, Z) :-

Z is X \* Y.

divide(X, Y, Z) :-

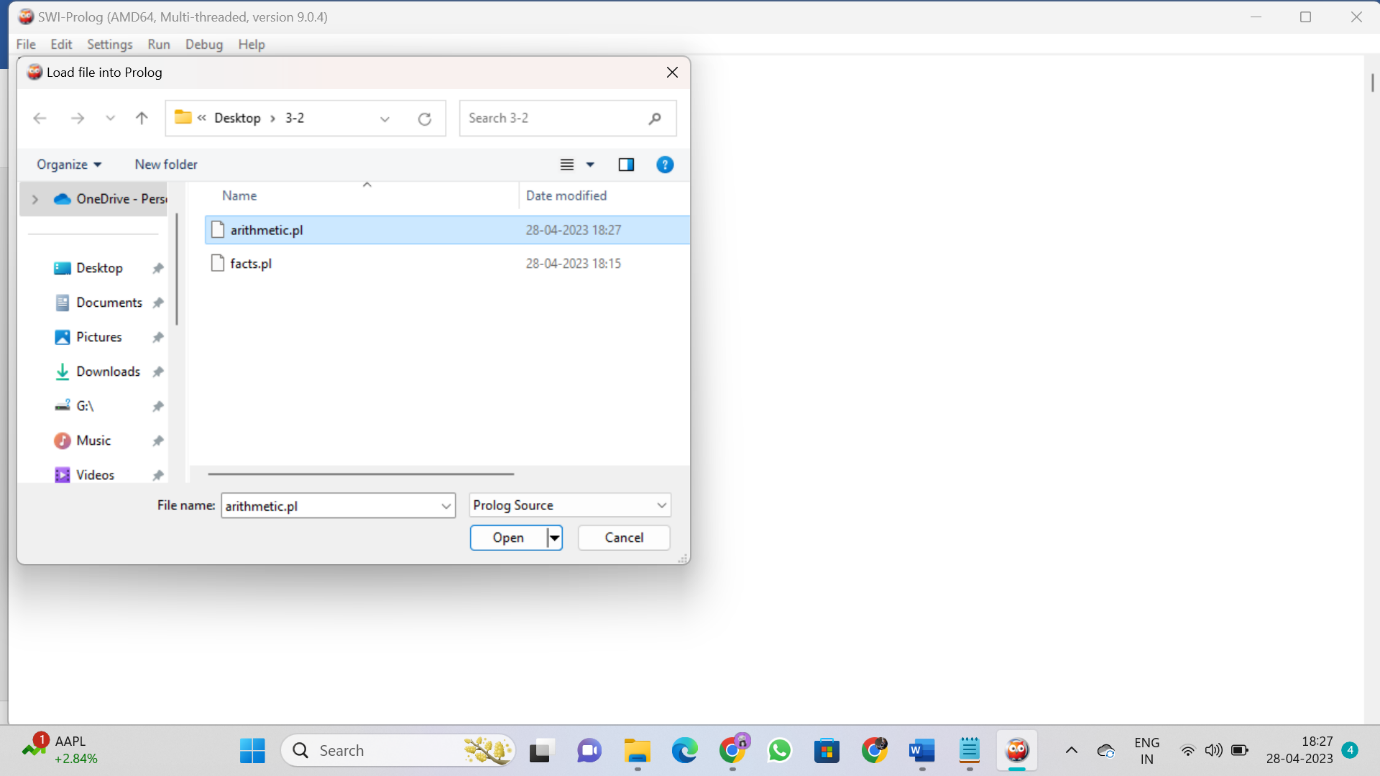
Y =\= 0,

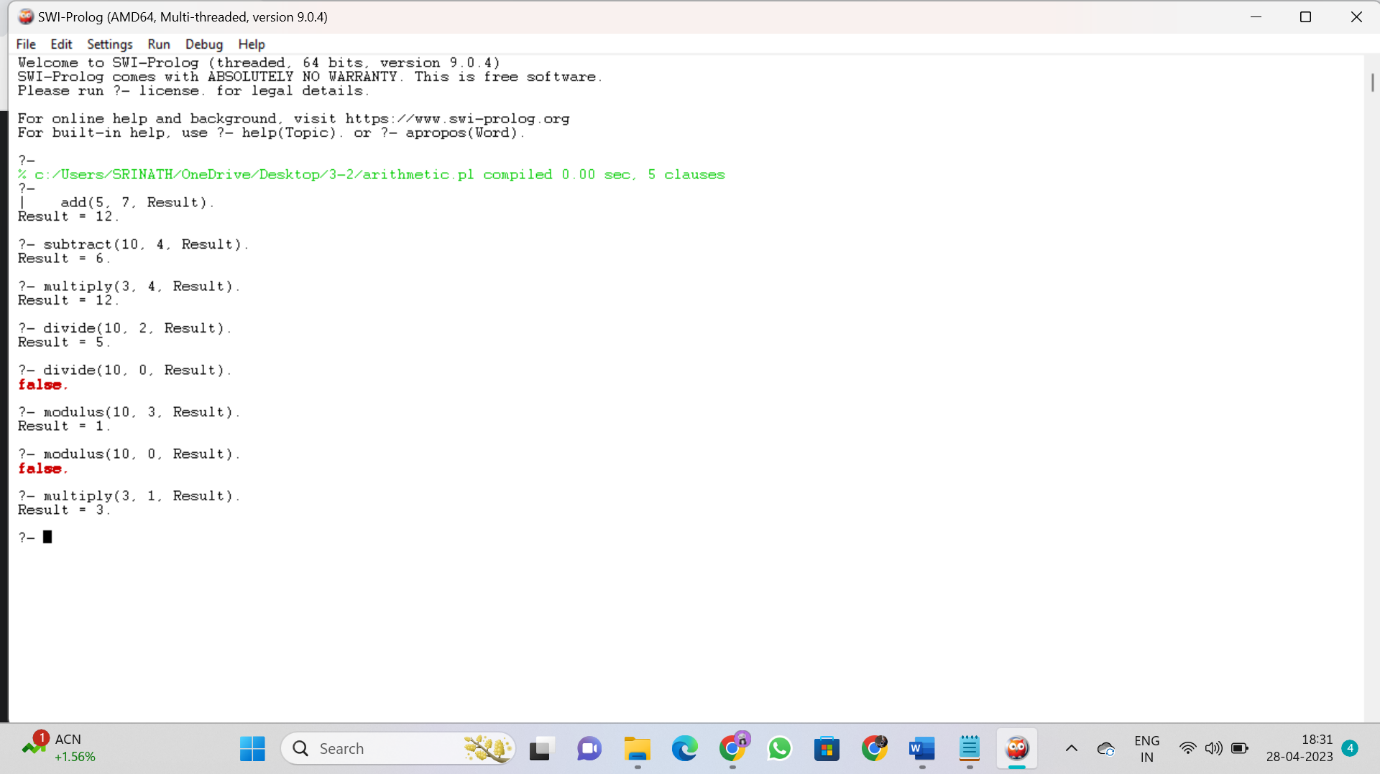
Z is X / Y.

modulus(X, Y, Z) :-

Y =\= 0,

Z is X mod Y.





**OUTPUT:**

?-

| add(5, 7, Result).

Result = 12.

?- subtract(10, 4, Result).

Result = 6.

?- multiply(3, 4, Result).

Result = 12.

?- divide(10, 2, Result).

Result = 5.

?- divide(10, 0, Result).

false.

?- modulus(10, 3, Result).

Result = 1.

?- modulus(10, 0, Result).

false.

?- multiply(3, 1, Result).

Result = 3.

**Experiment :3**

Write a program in prolog to solve Monkey banana problem

**THEORY:**

Suppose the problem is as given below −

* A hungry monkey is in a room, and he is near the door.
* The monkey is on the floor.
* Bananas have been hung from the center of the ceiling of the room.
* There is a block (or chair) present in the room near the window.
* The monkey wants the banana, but cannot reach it.



## So how can the monkey get the bananas?

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case −

* Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
* If the block position is not at the center, then monkey can drag it to the center.
* If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
* When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Now, let us see how we can solve this using Prolog. We will create some predicates as follows −

We have some predicates that will move from one state to another state, by performing action.

* When the block is at the middle, and monkey is on top of the block, and monkey does not have the banana (i.e. ***has not*** state), then using the ***grasp*** action, it will change from ***has not*** state to ***have*** state.
* From the floor, it can move to the top of the block (i.e. ***on top*** state), by performing the action ***climb***.
* The **push** or **drag** operation moves the block from one place to another.
* Monkey can move from one place to another using **walk** or **move** clauses.

Another predicate will be canget(). Here we pass a state, so this will perform move predicate from one state to another using different actions, then perform canget() on state 2. When we have reached to the state ‘**has>**’, this indicates ‘**has banana**’. We will stop the execution.

**CODE:**

on(floor,monkey).

on(floor,chair).

in(room,monkey).

in(room,chair).

in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,chair).

push(monkey,chair):-

strong(monkey).

under(banana,chair):-

push(monkey,chair).

canreach(banana,monkey):-

at(floor,banana);

at(ceiling,banana),

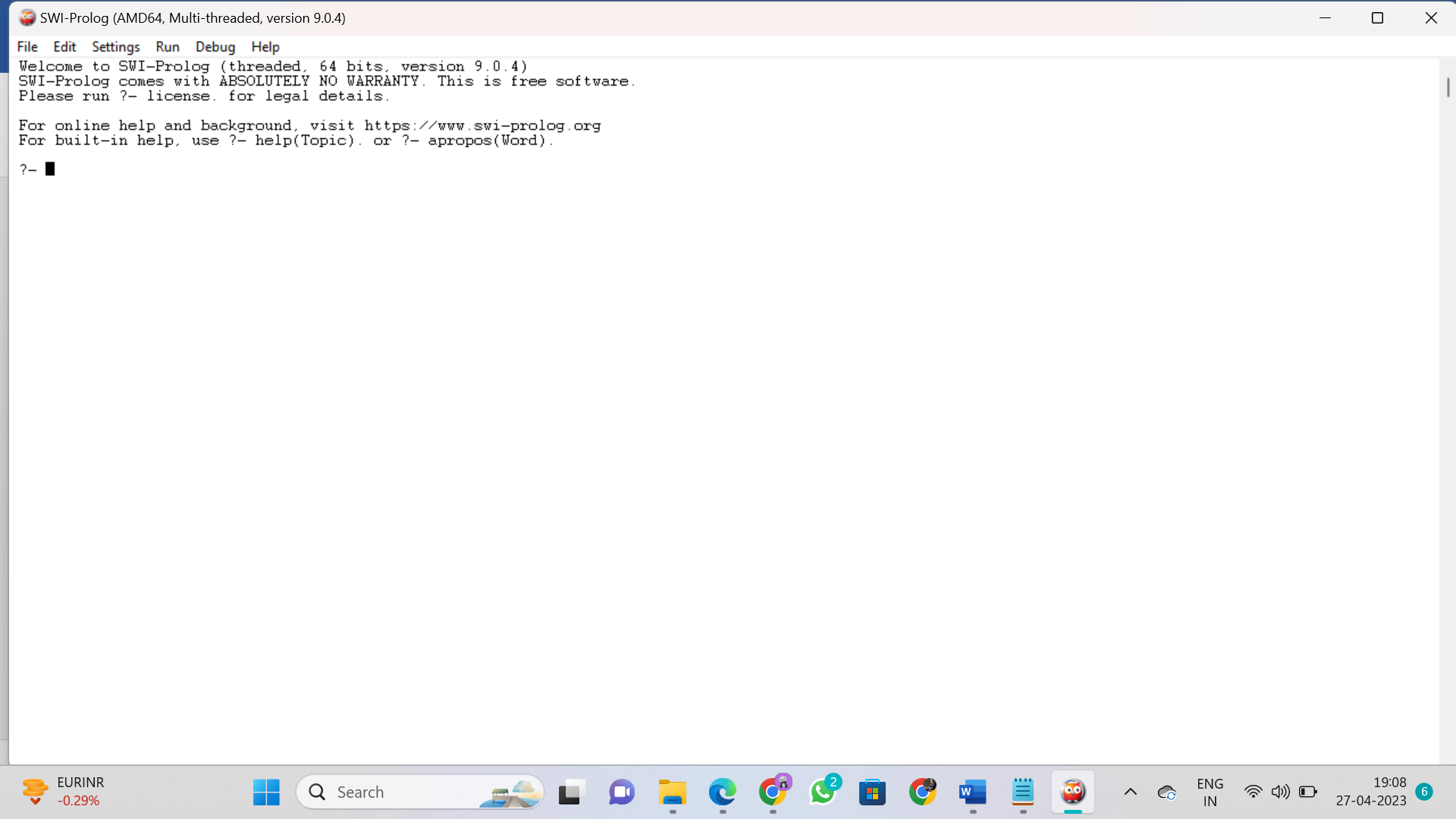
under(banana,chair),

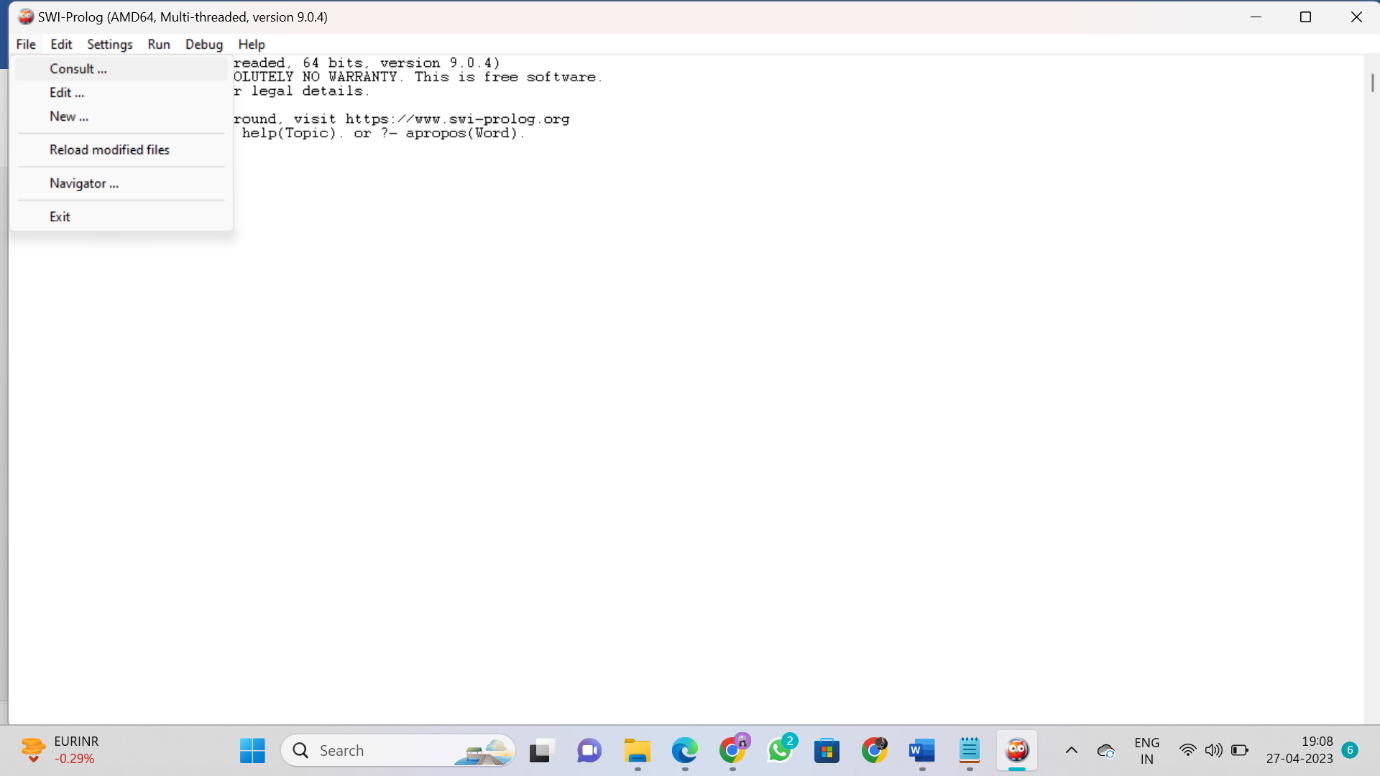
climb(monkey,chair).

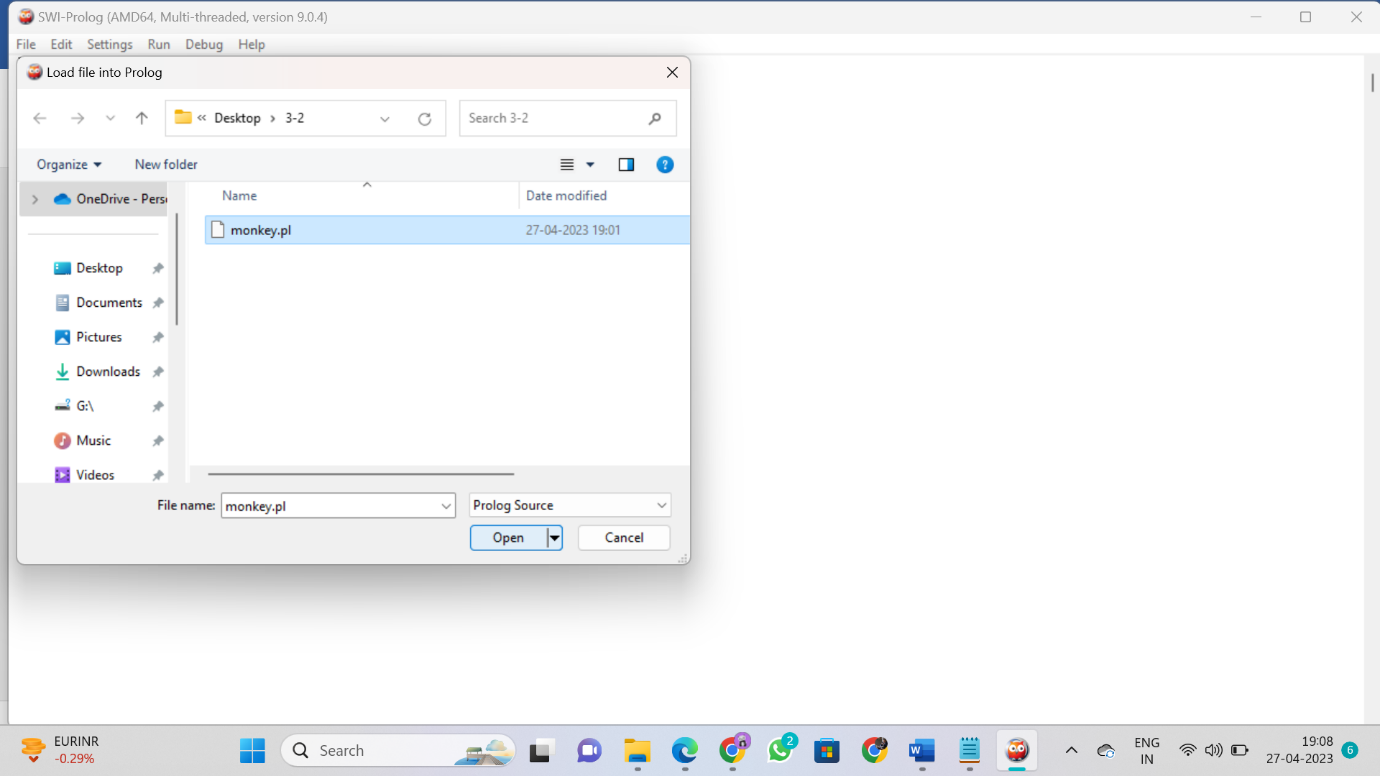
canget(banana,monkey):-

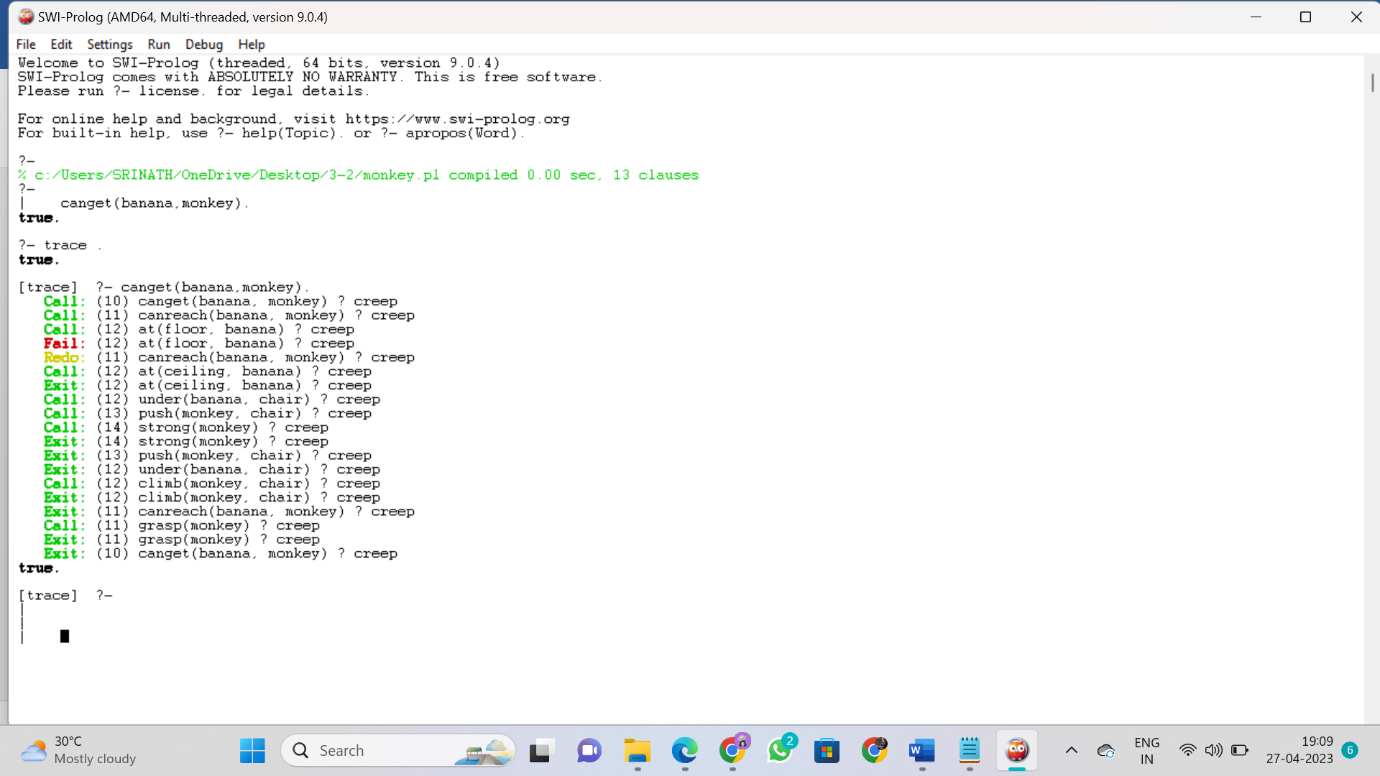
canreach(banana,monkey),grasp(monkey).

**OUTPUT:**









? - canget(banana,monkey).

true.

?- trace .

true.

[trace] ?- canget(banana,monkey).

Call: (10) canget(banana, monkey) ? creep

Call: (11) canreach(banana, monkey) ? creep

Call: (12) at(floor, banana) ? creep

Fail: (12) at(floor, banana) ? creep

Redo: (11) canreach(banana, monkey) ? creep

Call: (12) at(ceiling, banana) ? creep

Exit: (12) at(ceiling, banana) ? creep

Call: (12) under(banana, chair) ? creep

Call: (13) push(monkey, chair) ? creep

Call: (14) strong(monkey) ? creep

Exit: (14) strong(monkey) ? creep

Exit: (13) push(monkey, chair) ? creep

Exit: (12) under(banana, chair) ? creep

Call: (12) climb(monkey, chair) ? creep

Exit: (12) climb(monkey, chair) ? creep

Exit: (11) canreach(banana, monkey) ? creep

Call: (11) grasp(monkey) ? creep

Exit: (11) grasp(monkey) ? creep

Exit: (10) canget(banana, monkey) ? creep

true.

**Experiment :4**

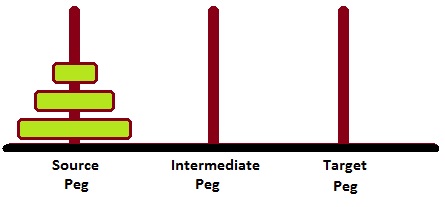
Write a program in prolog to solve Tower of Hanoi

**THEORY:**

Towers of Hanoi Problem is a famous puzzle to move N disks from the source peg/tower to the target peg/tower using the intermediate peg as an auxiliary holding peg. There are two conditions that are to be followed while solving this problem −

* A larger disk cannot be placed on a smaller disk.
* Only one disk can be moved at a time.

The following diagram depicts the starting setup for N=8 disks.



To solve this, we have to write one procedure move(N, Source, Target, auxiliary). Here N number of disks will have to be shifted from Source peg to Target peg keeping Auxiliary peg as intermediate.

**tower of hanor(using recursion method)**

Here n=8 the formula is 2^n-1.

We are taking 3disks.

2 n-1=28-1=7 steps we are taking

∴ Let take rode 1=A , rod 2=B, rod 3=C.

**Step 1** : shift First disk from 'A'to 'B'.

**Step 2** : shift Second disk from 'A'to 'C'.

**Step 3** : shift first disk from 'B'to 'C'.

**Step 4** : Move disk 3 from 'A'to 'C'.

**Step 5** : Move disk 1 from 'B'to 'A'.

**Step 6** : Move disk from 'A'to 'C'.

**Step 7** : Move disk 1 from 'A'to 'C'.

∴ The patterns is

* Shift n-1 disks from 'A'to 'B'
* Shift last disk from 'A'to 'C'
* Shift 'n-1' disks from 'B'to 'C']

**CODE:**

move(1,X,Y,\_) :-

write('Move top disk from '), write(X), write(' to '), write(Y), nl.

move(N,X,Y,Z) :-

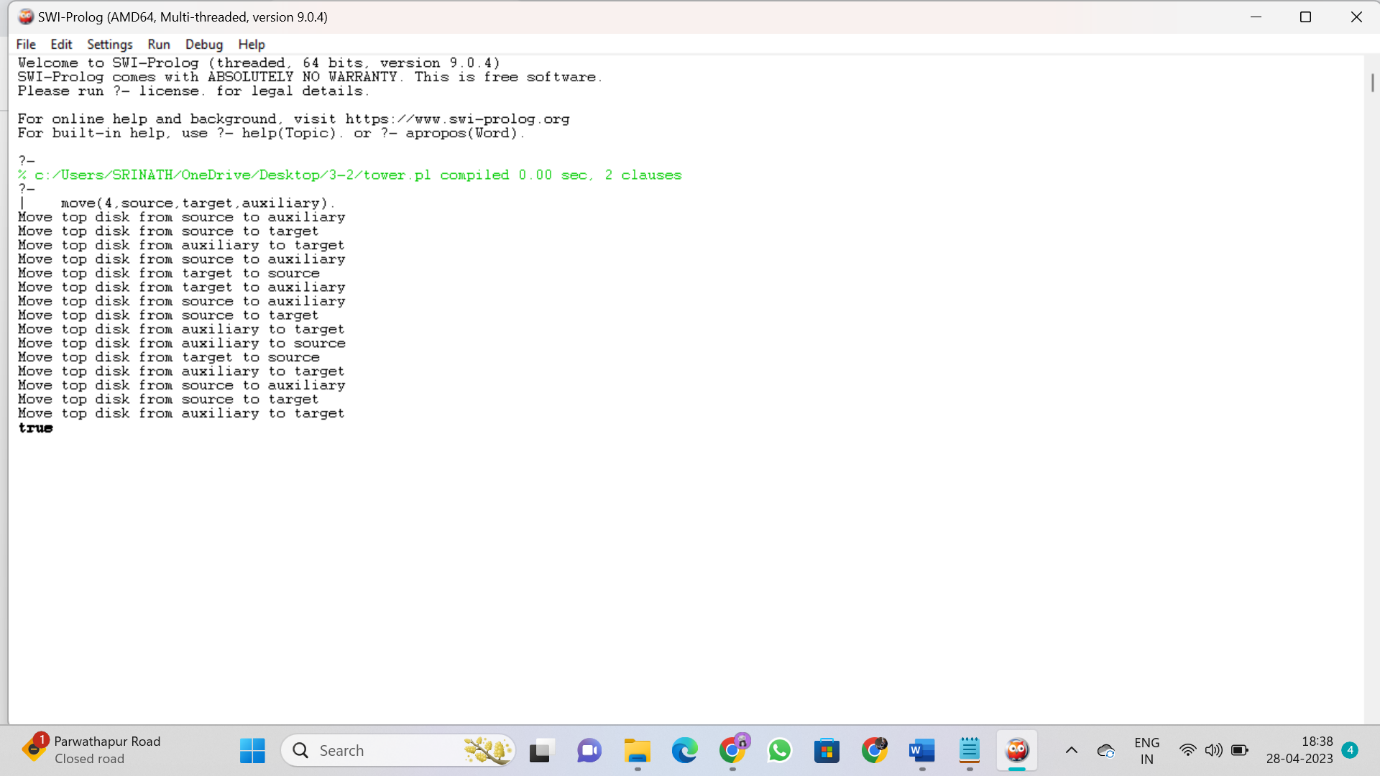
N>1,

M is N-1,

move(M,X,Z,Y),

move(1,X,Y,\_),

move(M,Z,Y,X).



**OUTPUT:**

move(4,source,target,auxiliary).

Move top disk from source to auxiliary

Move top disk from source to target

Move top disk from auxiliary to target

Move top disk from source to auxiliary

Move top disk from target to source

Move top disk from target to auxiliary

Move top disk from source to auxiliary

Move top disk from source to target

Move top disk from auxiliary to target

Move top disk from auxiliary to source

Move top disk from target to source

Move top disk from auxiliary to target

Move top disk from source to auxiliary

Move top disk from source to target

Move top disk from auxiliary to target

True

**Experiment :5**

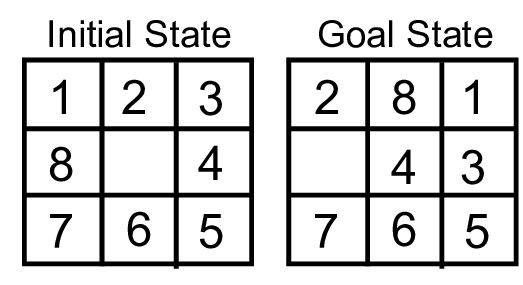
Write a program in prolog to solve 8 Puzzle problems

**THEORY:**

**Definition**:

“It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state”.

**Procedure:**



The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle i s a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it.

A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as two situations given below.

**Figure            (Starting State)                              (Goal State)**

The state of 8-puzzle is the different permutation of tiles within the frame. The operations are the permissible moves up, down, left, right. Here at each step of the problem a function f(x) will be defined which is the combination of g(x) and h(x).

i.e. **F(x)=g(x) + h (x)**

Where

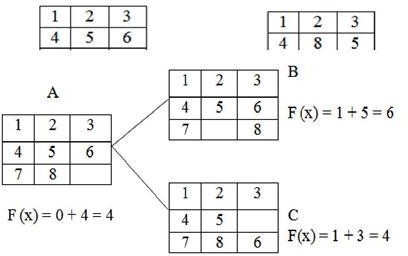
**g (x):**how many steps in the problem you have already done or the current state from the initial state.

**h (x):**Number of ways through which you can reach at the goal state from the current state or Or

**h (x):**is the heuristic estimator that compares the current state with the goal state note down how many states are displaced from the initial or the current state. After calculating the **f** value at each step finally take the smallest f (x) value at every step and choose that as the next current state to get the goal

Let us take an example.

**Figure     (Initial State)                               (Goal State)**



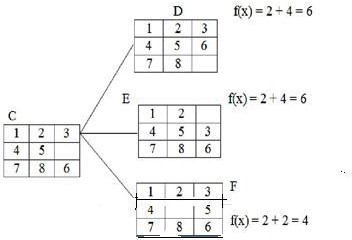
**Step 1:**

**f (x)**is the step required to reach at the goal state from the initial state. So in the trayeither 6 or 8 can change their portions to fill the empty position. So there will be two possible current states namely B and C. The f (x) value of B is 6 and that of C is 4. As 4 is the minimum, so take C as the current state to the next state.

**Step 2:**

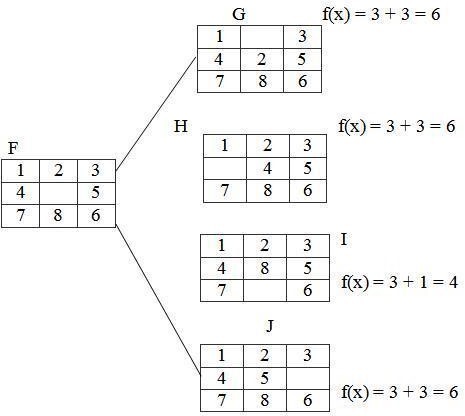
In this step, from the tray C three states can be drawn. The empty position will contain either 5 or 3 or 6. So for three different values three different states can be obtained. Then calculate each of their f (x) and take the minimum one.

Here the state F has the minimum value i.e. 4 and hence take that as the next current state.



**Step 3:**

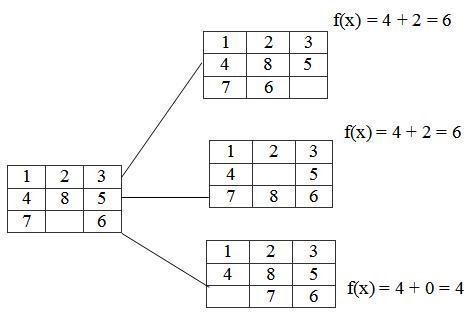
The tray F can have 4 different states as the empty positions can be filled with b4 values i.e.2, 4, 5, 8.

****

**Step 4:**

In the step-3 the tray I has the smallest f (n) value. The tray I can be implemented in 3 different states because the empty position can be filled by the members like 7, 8, 6.

Hence, we reached at the goal state after few changes of tiles in different positions of the trays.



**CODE:**

ids :-

start(State),

length(Moves, N),

dfs([State], Moves, Path), !,

show([start|Moves], Path),

format('~nmoves = ~w~n', [N]).

dfs([State|States], [], Path) :-

goal(State), !,

reverse([State|States], Path).

dfs([State|States], [Move|Moves], Path) :-

move(State, Next, Move),

not(memberchk(Next, [State|States])),

dfs([Next,State|States], Moves, Path).

show([], \_).

show([Move|Moves], [State|States]) :-

State = state(A,B,C,D,E,F,G,H,I),

format('~n~w~n~n', [Move]),

format('~w ~w ~w~n',[A,B,C]),

format('~w ~w ~w~n',[D,E,F]),

format('~w ~w ~w~n',[G,H,I]),

show(Moves, States).

% Empty position is marked with '\*'

start( state(6,1,3,4,\*,5,7,2,0) ).

goal( state(\*,0,1,2,3,4,5,6,7) ).

move( state(\*,B,C,D,E,F,G,H,J), state(B,\*,C,D,E,F,G,H,J), right).

move( state(\*,B,C,D,E,F,G,H,J), state(D,B,C,\*,E,F,G,H,J), down ).

move( state(A,\*,C,D,E,F,G,H,J), state(\*,A,C,D,E,F,G,H,J), left ).

move( state(A,\*,C,D,E,F,G,H,J), state(A,C,\*,D,E,F,G,H,J), right).

move( state(A,\*,C,D,E,F,G,H,J), state(A,E,C,D,\*,F,G,H,J), down ).

move( state(A,B,\*,D,E,F,G,H,J), state(A,\*,B,D,E,F,G,H,J), left ).

move( state(A,B,\*,D,E,F,G,H,J), state(A,B,F,D,E,\*,G,H,J), down ).

move( state(A,B,C,\*,E,F,G,H,J), state(\*,B,C,A,E,F,G,H,J), up ).

move( state(A,B,C,\*,E,F,G,H,J), state(A,B,C,E,\*,F,G,H,J), right).

move( state(A,B,C,\*,E,F,G,H,J), state(A,B,C,G,E,F,\*,H,J), down ).

move( state(A,B,C,D,\*,F,G,H,J), state(A,\*,C,D,B,F,G,H,J), up ).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,D,F,\*,G,H,J), right).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,D,H,F,G,\*,J), down ).

move( state(A,B,C,D,\*,F,G,H,J), state(A,B,C,\*,D,F,G,H,J), left ).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,\*,D,E,C,G,H,J), up ).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,C,D,\*,E,G,H,J), left ).

move( state(A,B,C,D,E,\*,G,H,J), state(A,B,C,D,E,J,G,H,\*), down ).

move( state(A,B,C,D,E,F,\*,H,J), state(A,B,C,D,E,F,H,\*,J), left ).

move( state(A,B,C,D,E,F,\*,H,J), state(A,B,C,\*,E,F,D,H,J), up ).

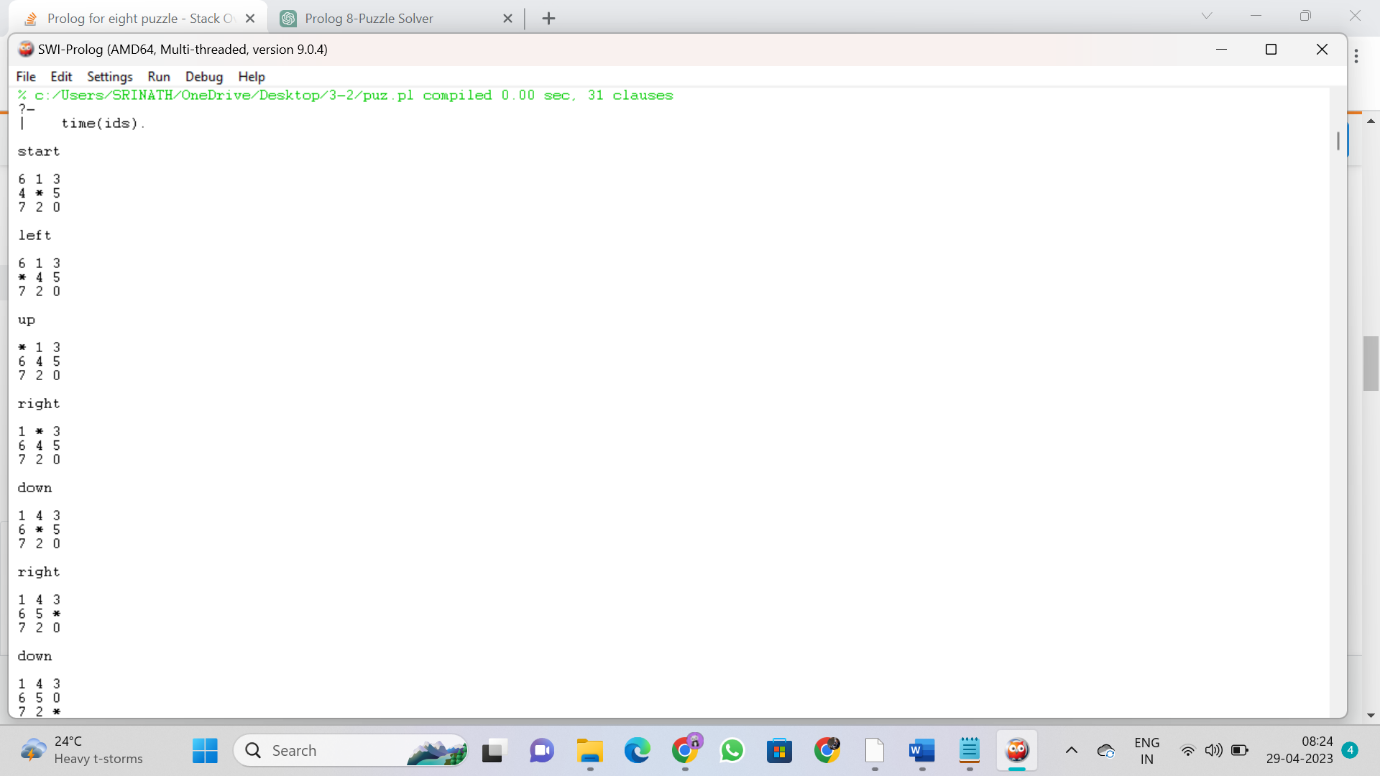
move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,E,F,\*,G,J), left ).

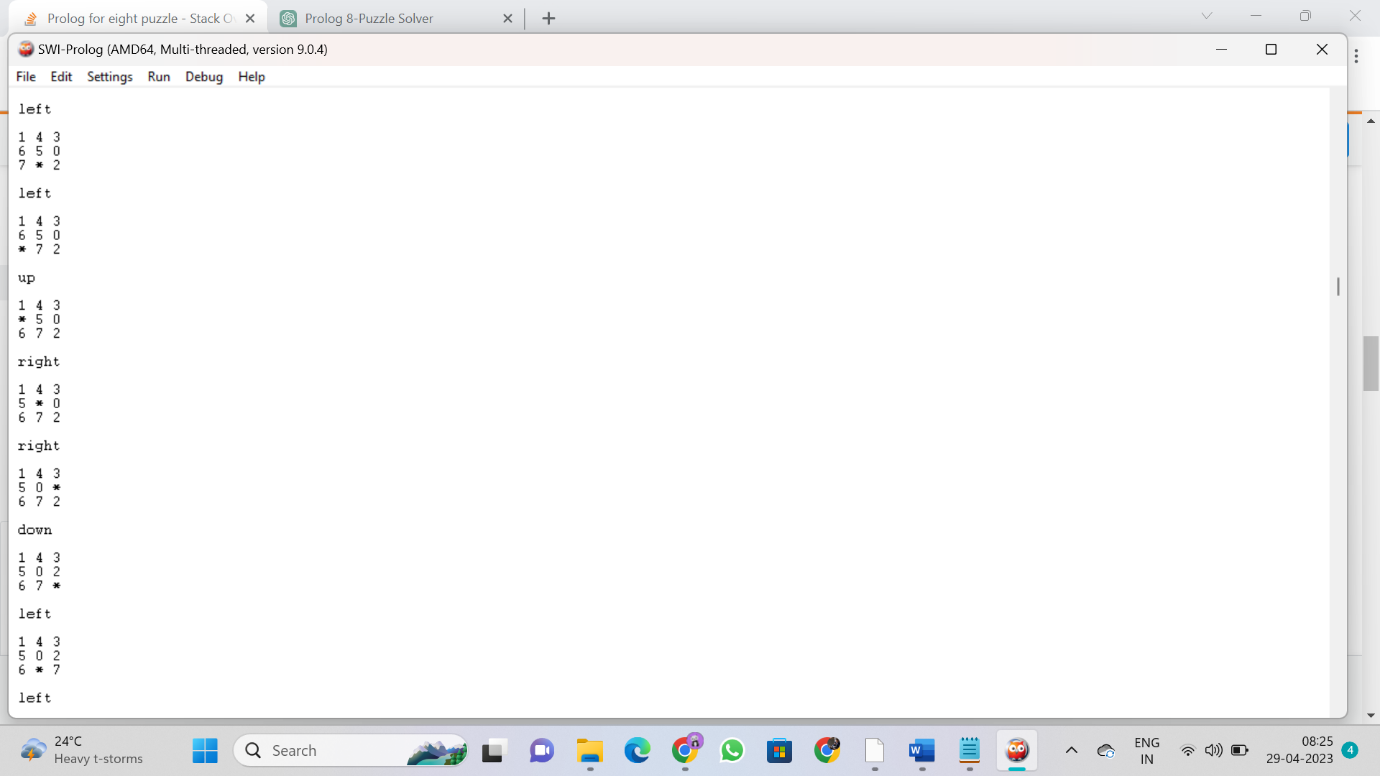
move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,\*,F,G,E,J), up ).

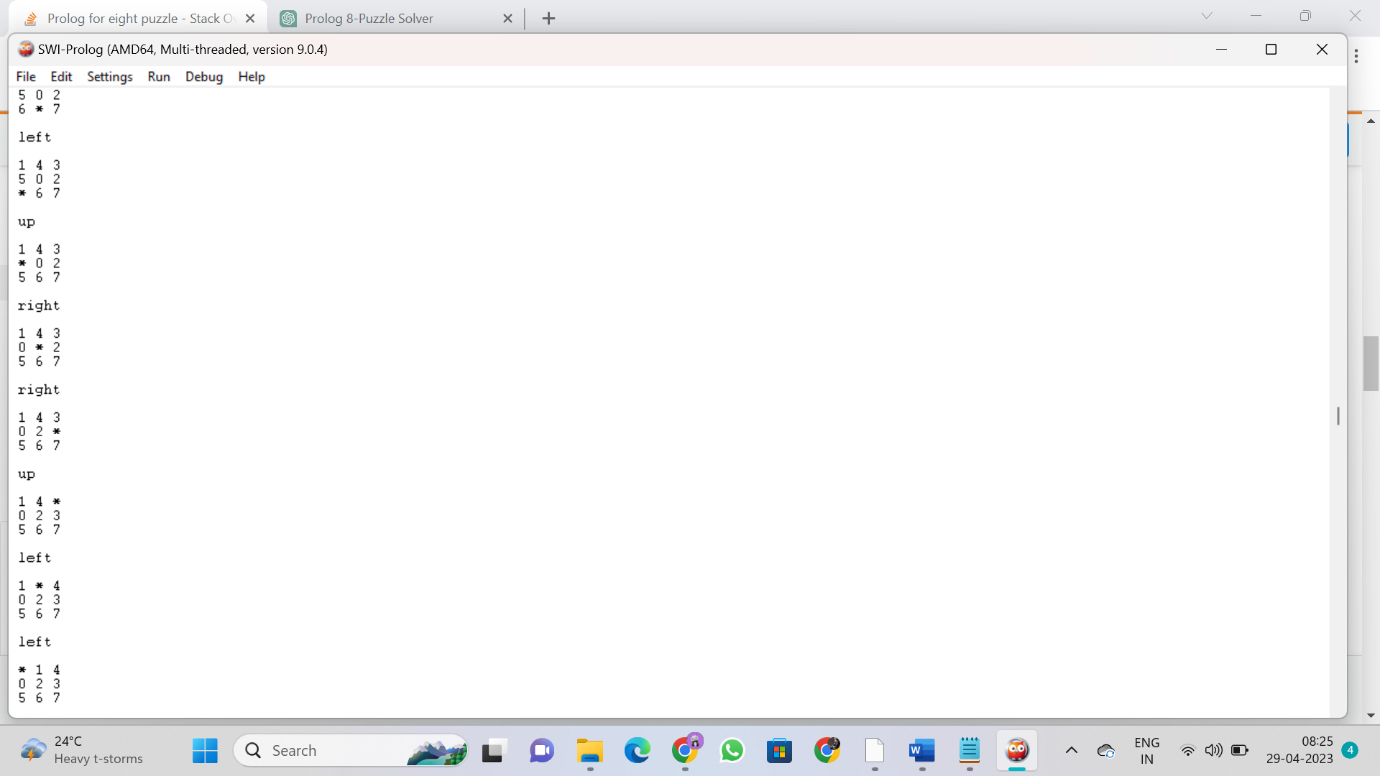
move( state(A,B,C,D,E,F,G,\*,J), state(A,B,C,D,E,F,G,J,\*), right).

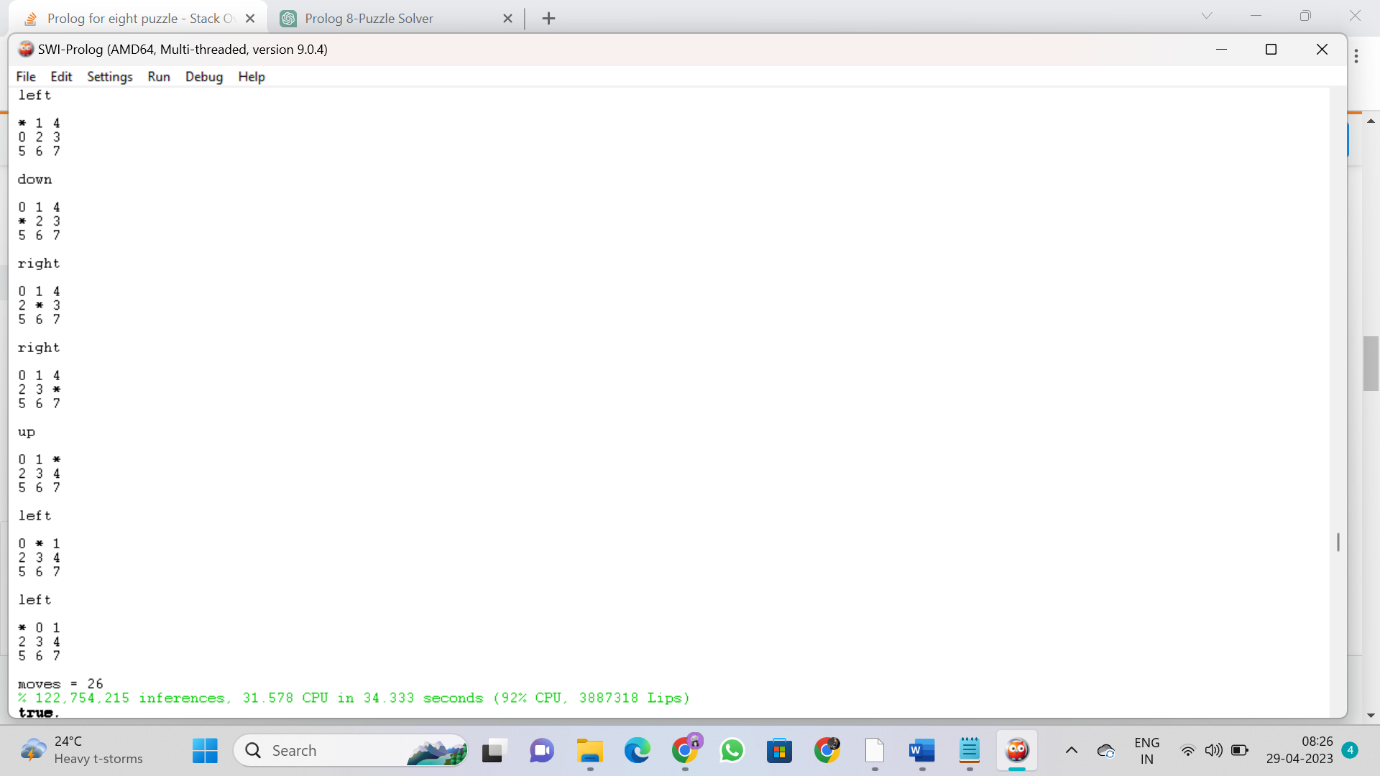
move( state(A,B,C,D,E,F,G,H,\*), state(A,B,C,D,E,\*,G,H,F), up ).

move( state(A,B,C,D,E,F,G,H,\*), state(A,B,C,D,E,F,G,\*,H), left ).









**OUTPUT:**

?-

| time(ids).

**start**

6 1 3

4 \* 5

7 2 0

**left**

6 1 3

\* 4 5

7 2 0

**up**

\* 1 3

6 4 5

7 2 0

**right**

1 \* 3

6 4 5

7 2 0

**down**

1 4 3

6 \* 5

7 2 0

**right**

1 4 3

6 5 \*

7 2 0

**down**

1 4 3

6 5 0

7 2 \*

**left**

1 4 3

6 5 0

7 \* 2

**left**

1 4 3

6 5 0

\* 7 2

**Up**

1 4 3

\* 5 0

6 7 2

**right**

1 4 3

5 \* 0

6 7 2

**right**

1 4 3

5 0 \*

6 7 2

**down**

1 4 3

5 0 2

6 7 \*

**left**

1 4 3

5 0 2

6 \* 7

**left**

1 4 3

5 0 2

\* 6 7

**up**

1 4 3

\* 0 2

5 6 7

**right**

1 4 3

0 \* 2

5 6 7

**right**

1 4 3

0 2 \*

5 6 7

**up**

1 4 \*

0 2 3

5 6 7

**left**

1 \* 4

0 2 3

5 6 7

**left**

\* 1 4

0 2 3

5 6 7

**down**

0 1 4

\* 2 3

5 6 7

**right**

0 1 4

2 \* 3

5 6 7

**right**

0 1 4

2 3 \*

5 6 7

**up**

0 1 \*

2 3 4

5 6 7

**left**

0 \* 1

2 3 4

5 6 7

**left**

\* 0 1

2 3 4

5 6 7

**moves = 26**

**% 122,754,215 inferences, 31.578 CPU in 34.333 seconds (92% CPU, 3887318 Lips)**

**true.**

**Experiment :6**

Write a program in prolog to solve 4-Queens problem

**THEORY:**

n **4- queens problem**, we have 4 queens to be placed on a 4\*4 chessboard, satisfying the constraint that no two queens should be in the same row, same column, or in same diagonal.

The solution space according to the external constraints consists of 4 to the power 4, 4-tuples i.e., **Si = {1, 2, 3, 4}** and **1<= I <=4**, whereas according to the internal constraints they consist of **4!** solutions i.e., permutation of **4**.

## Solution of 4 – queen’s with the help of backtracking

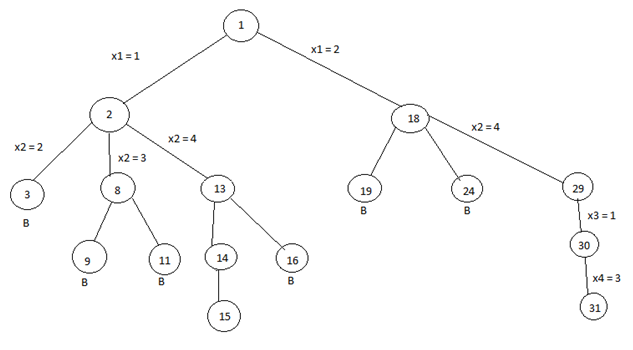
We can solve 4-queens problem through backtracking by taking it as a bounding function .in use the criterion that if (x1, x2, ……., xi) is a path to a current E-node, then all the children nodes with parent-child labelings x (i+1) are such that (x1, x2, x3, ….., x(i+1)) represents a chessboard configuration in which no queens are attacking.

So we start with the root node as the only live node. This time this node becomes the E-node and the path is (). We generate the next child. Suppose we are generating the child in ascending order. Thus the node number 2 is generated and path is now 1 i.e., the queen 1 is placed in the first row and in the first column.

Now, node 2 becomes the next E-node or line node. Further, try the next node in the ascending nodes i.e., the node 3 which is having x2 = 2 means queen 2 is placed in the second column but by this the queen 1 and 2 are on the same diagonal, so node 3 becomes dead here so we backtrack it and try the next node which is possible.

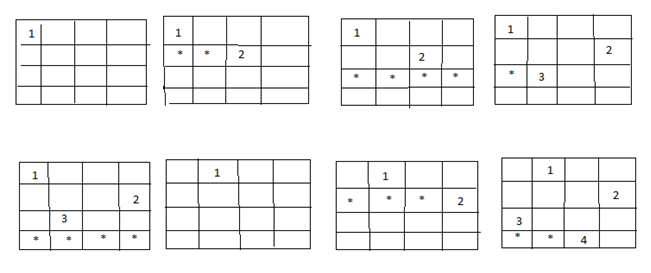
Here, the x2 = 3 means the queen 2 is placed in the 3rd column. As it satisfies all the constraints so it becomes the next live node.

After this try for next node 9 having x3 = 2 which means the queen 3 placed in the 2nd column, but by this the 2 and 3 queen are on the same diagonal so it becomes dead. Now we try for next node 11 with x3 = 4, but again the queens 2 and 3 are on the same diagonal so it is also a dead node.



**\*** The B denotes the dead node.

We try for all the possible positions for the queen 3 and if not any position satisfy all the constraints then backtrack to the previous live node.

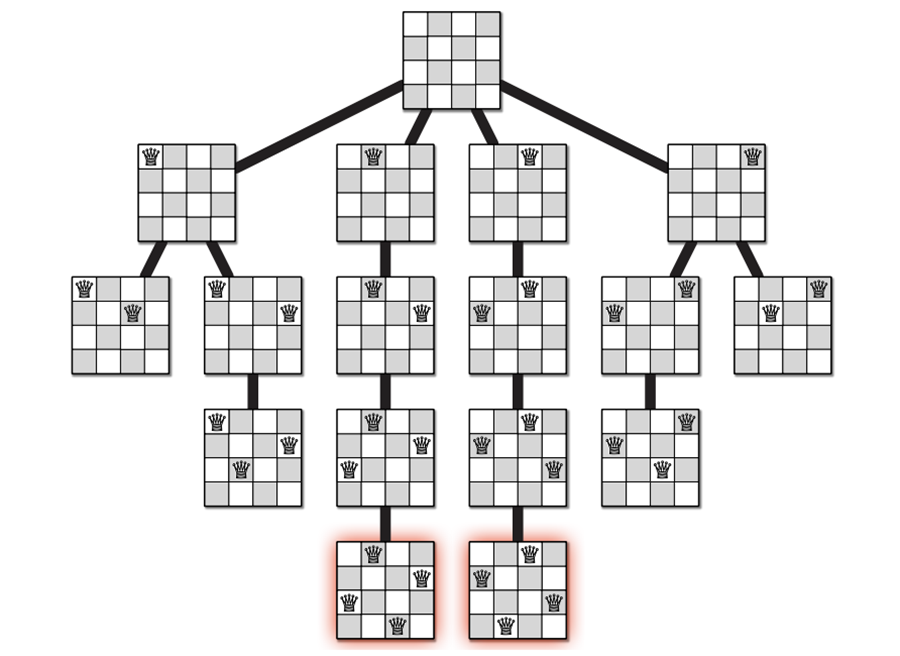


Now, the node13 become the new live node with x2 = 4, means queen 2 is placed in the 4th column. Move to the next node 14. It becomes the next live node with x3 = 2 means the queen 3 is placed in the 2nd column. Further, we move to the next node 15 with x4 = 3 as the live node. But this makes the queen 3 and 4 on the same diagonal resulting this node 15 is the dead node so we have to backtrack to the node 14 and then backtrack to the node 13 and try the other possible node 16 with x3 = 3 by this also we get the queens 2 and 3 on the same diagonal so the node is the dead node.

So we further backtrack to the node 2 but no other node is left to try so the node 2 is killed so we backtrack to the node 1 and try another sub-tree having x1 = 2 which means queen 1 is placed in the 2nd column.

Now again with the similar reason, nodes 19 and 24 are killed and so we try for the node 29 with x2 = 4 means the queen 2 is placed in the 4th column then we try for the node 30 with x3 = 1 as a live node and finally we proceed to next node 31 with x4 = 3 means the queen 4 is placed in 3rd column.

Here, all the constraints are satisfied, so the desired result for 4 queens is {2, 4, 1, 3}.



**Code :**

domain(X) :- member(X, [1,2,3,4]).

no\_threat((X1,Y1),(X2,Y2)) :-

X1 \= X2,

Y1 \= Y2,

Xdiff is abs(X1 - X2),

Ydiff is abs(Y1 - Y2),

Xdiff \= Ydiff.

no\_threat(\_,[]).

no\_threat(Q,[Q1|QT]) :- no\_threat(Q,Q1), no\_threat(Q,QT).

solve(Q1,Q2,Q3,Q4) :-

domain(Q1), domain(Q2), domain(Q3), domain(Q4),

no\_threat((1,Q1),(2,Q2)),

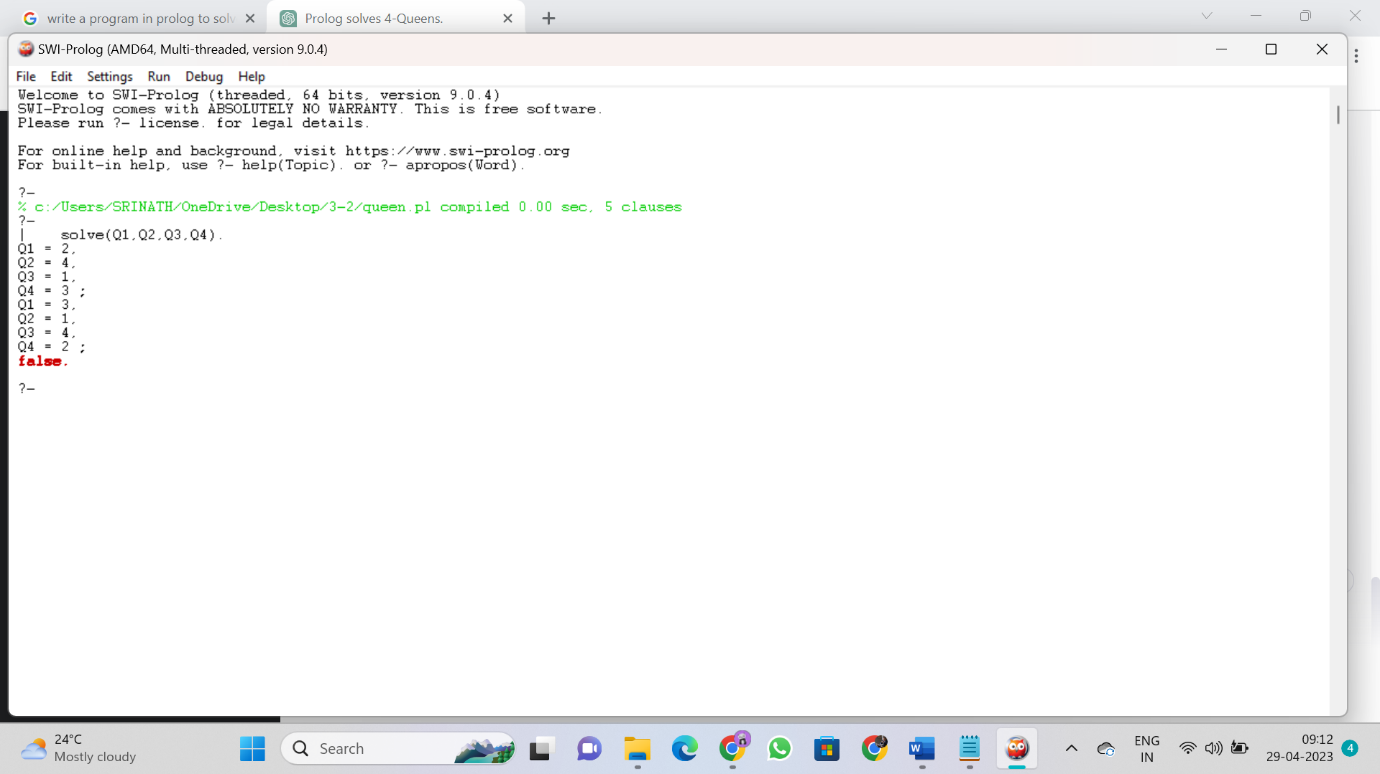
no\_threat((1,Q1),(3,Q3)),

no\_threat((1,Q1),(4,Q4)),

no\_threat((2,Q2),(3,Q3)),

no\_threat((2,Q2),(4,Q4)),

no\_threat((3,Q3),(4,Q4)).



**OUTPUT:**

?-

| solve(Q1,Q2,Q3,Q4).

**Q1 = 2,**

**Q2 = 4,**

**Q3 = 1,**

**Q4 = 3 ;**

***Q1 = 3,***

***Q2 = 1,***

***Q3 = 4,***

***Q4 = 2 ;***

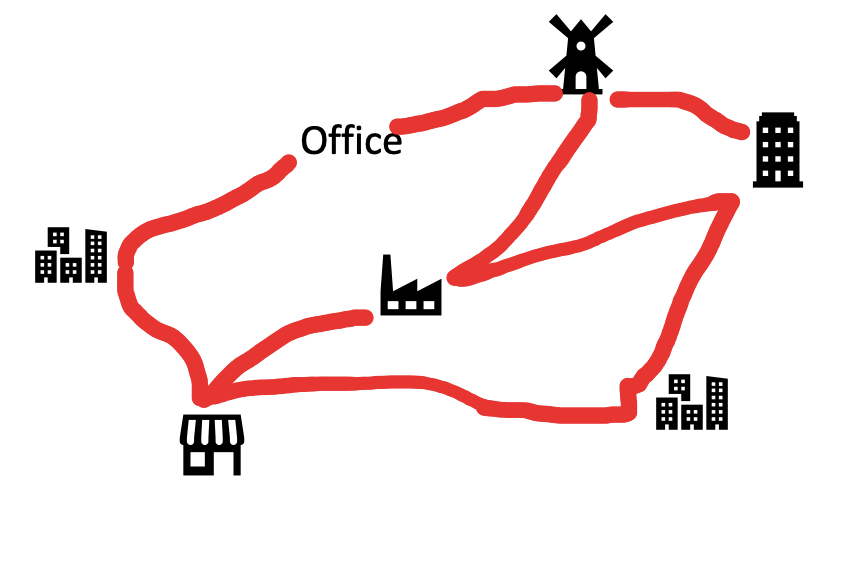
false.

**Experiment :7**

Write a program in prolog to solve Traveling salesman problem

## THEORY: Traveling Salesman Problem (TSP)

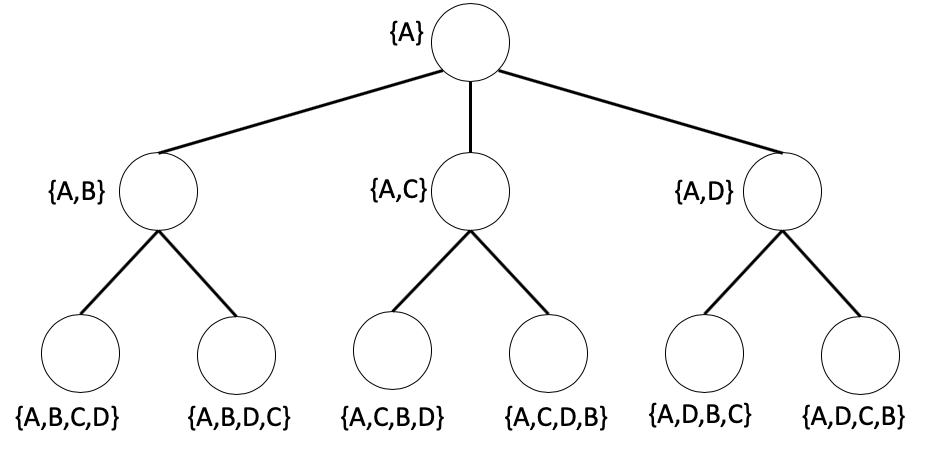
Consider the following situation. You are given a list of n cities with the distance between any two cities. Now, you have to start with your office and to visit all the cities only once each and return to your office. What is the shortest path can you take? This problem is called the Traveling Salesman Problem (TSP).



**Problem Formulation of TSP**

To make the problem simple, we consider 3-city-problem.  
Let’s call the office ( A )and the 3 cities ( B ) ( C ) ( D ) respectively. We initialize the problem state by {A} means the salesman departed from his office. As an operator, when he visited city-B, the problem state is updated to {A, B}, where the order of elements in { } is considered. When the salesman visited all the cities, {A, B, C, D} in this case, the departed point A is automatically added to the state which means {A, B, C, D, A}. Therefore, the initial state of this TSP is {A} and the final state(goal) is {A, X1, X2, X3, A} where traveled distance is minimized.

Taking each state as a node of a tree structure, we can represent this TSP as the following tree search problem.

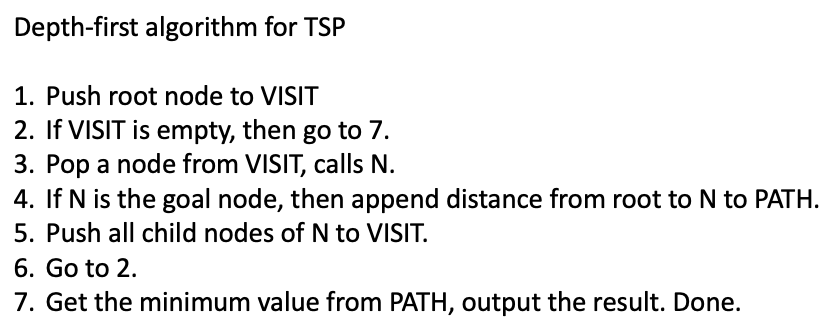


**Brute-force search**

**Depth-first search**

The depth-first search algorithm starts at the root node and explores as deep as possible along each branch before taking backtracking. In our TSP, when a state node with all city labels is visited, its total distance is memorized. This information will later be used to define the shortest path.

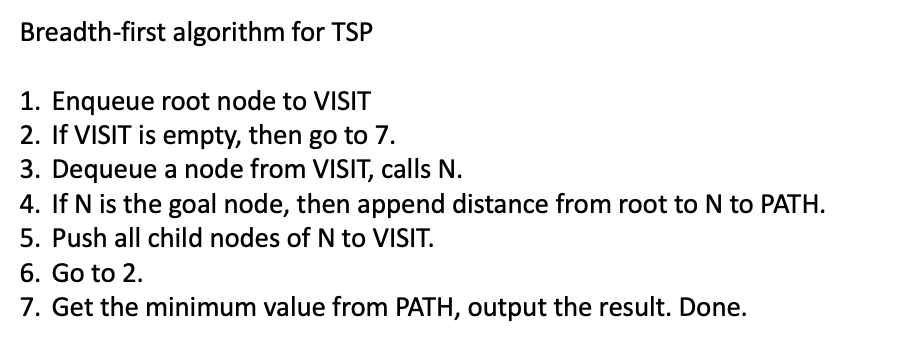
Let VISIT be a ***stack*** to save visited nodes, PATH be a set to save distances from the root node to the goal. The depth-first algorithm can be written as



**Breadth-first search**

The depth-first search algorithm starts at the root node and explores all of the nodes at the present depth level before moving on to the nodes at the next depth level. In our TSP, when a state node with all city labels is visited, its total distance is memorized. This information will later be used to define the shortest path.

Let VISIT be a ***queue*** to save visited nodes, PATH be a set to save distances from the root node to the goal. The breadth-first algorithm can be written as



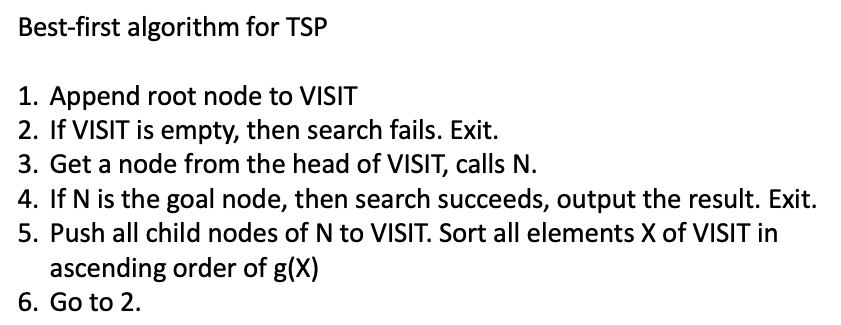
**Heuristic Search**

In Brute-force search, all nodes are visited and the information from each node (distance from a node to a node) is not considered. This leads to a large amount of time and memory consumption. To solve this problem, a heuristic search is a solution. The information of each state node is used to consider visiting a node or not. This information is represented by a heuristic function which commonly set up by user’s experiences. For example, we can define the heuristic function by the distance from the root node to the present visit node, or the distance from the present visit node to the goal node.

**Best-first search**

In the Best-first search, we use the information of distance from the root node to decide which node to visit first. Let g(X) be the distance from the root node to node-X. Therefore, the distance from the root node to node-Y by visiting node-X is g(Y)=g(X)+d(X, Y), where d(X, Y) is the distance between X and Y.

Let VISIT be a ***list*** to save visited nodes. The best-first algorithm can be written as



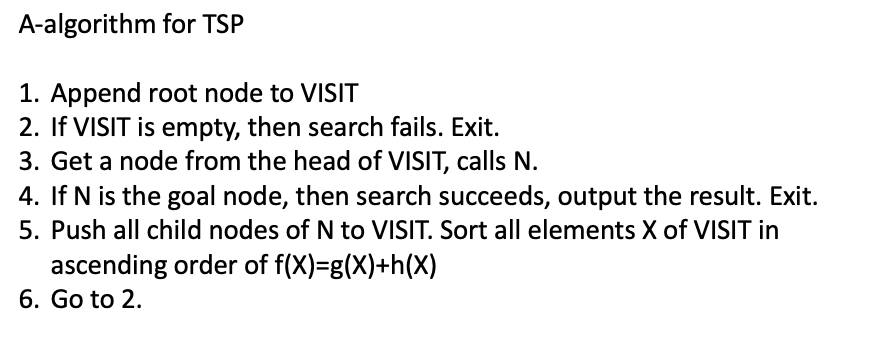
**A-algorithm (A\*-algorithm)**

In the A algorithm search, we use the information of distance from the present visit node to the goal as a heuristic function, h(X). Let g(X) be the distance from the root node to node-X. In this case, we consider the priority of node visit order by f(X)=g(X)+h(X).

In real-world problems, it is impossible to obtain the exact value of h(X). In that case, an estimation value of h(X), h’(X), is used. However, setting h’(X) takes risks in falling into a local optimal answer. To prevent this problem, choosing h’(X) which h’(X)≤h(X) for all X is recommended. In this case, it is known as A\*-algorithm and it can be shown that the obtained answer is the global optimal answer.

In our experiment described in the following part, we are setting h’(X) as the sum of the minimum distance of all possible routes from each city which is not presented in the current visit node label, and the present city. For example, if the present node is {A, D}, then city-B and C is not present in the label. Therefore, h’(D)=min(all possible route distance from C)+min(all possible route distance from B)+min(all possible route distance from D).

Let VISIT be a ***list*** to save visited nodes. The A-algorithm can be written as



**Code:**

cities([city(1,2), city(2,5), city(3,1), city(4,6)]).

distance(city(X1,Y1), city(X2,Y2), D) :-

DX is X2 - X1,

DY is Y2 - Y1,

D is sqrt(DX\*DX + DY\*DY).

tour\_distance([City], 0).

tour\_distance([City1, City2|Cities], Dist) :-

distance(City1, City2, D),

tour\_distance([City2|Cities], Dist1),

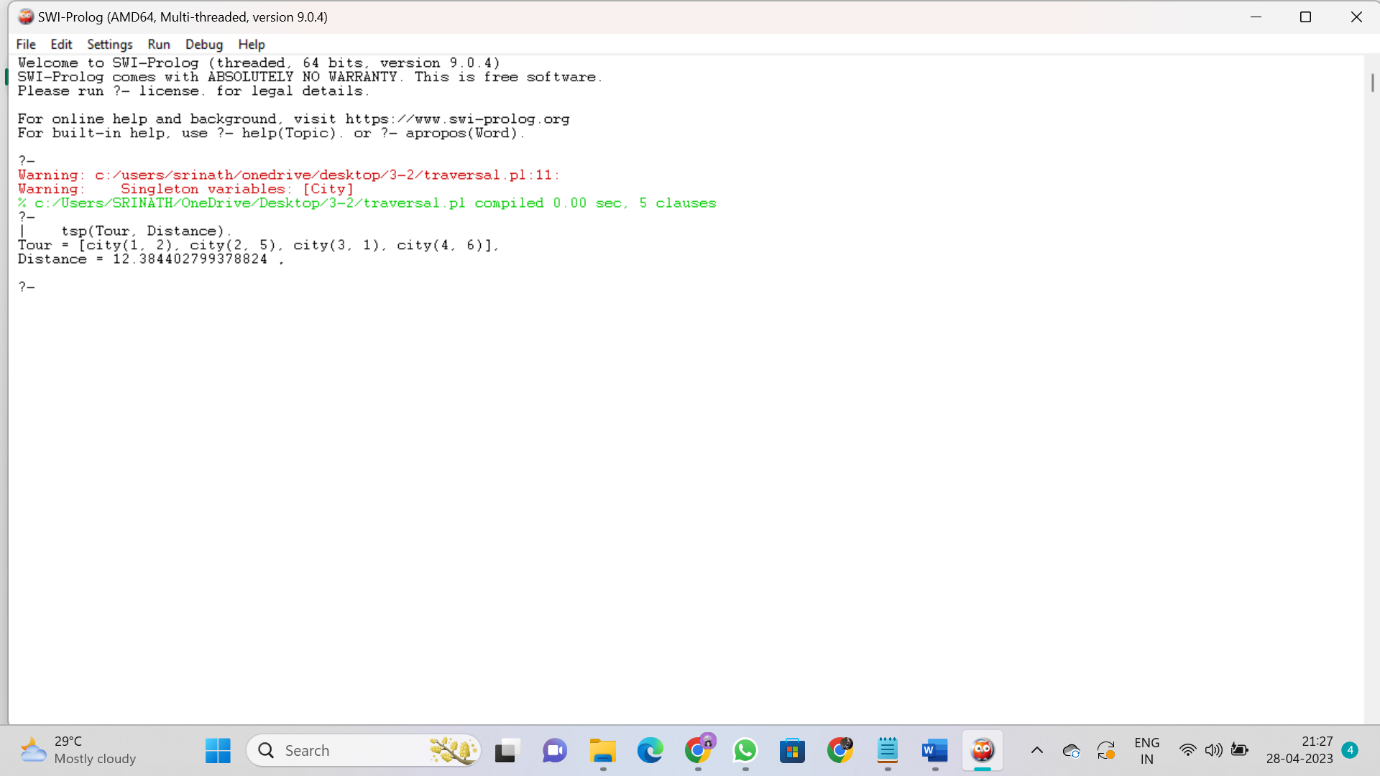
Dist is Dist1 + D.

tsp(Tour, Distance) :-

cities(Cities),

permutation(Cities, Tour),

tour\_distance(Tour, Distance).



**Output:**

? | tsp(Tour, Distance).

Tour = [city(1, 2), city(2, 5), city(3, 1), city(4, 6)],

Distance = 12.384402799378824 .

**Experiment :8**

Write a program in **prolog for Water jug problem.**

**THEORY :**

## Introduction:

* Water Jug Problem is a classic problem in Artificial Intelligence (AI) that involves finding a way to measure specific amounts of water using two jugs with different capacities.
* The goal is to reach a specific target amount of water in one of the jugs, without exceeding its capacity, by transferring water from one jug to another.
* This problem can be solved using a state space search algorithm.

## State Space Search:

* State space search is a technique used in AI to find the solution to a problem by searching through the state space of a problem.
* In this approach, the problem is represented as a graph with nodes as states and edges as transitions.
* The search algorithm starts at an initial state and explores the graph to reach the goal state.
* In the water jug problem, the state space can be represented as a tuple (x, y) where x and y are the amounts of water in the two jugs.
* The initial state is (0, 0) and the goal state is (x, y) or In my case, it is (2, y) where x and y are the desired amounts of water.

## Production Rules:

Production rules, also known as rules or heuristics, are used in state space searches to define the actions that can be taken to move from one state to another. In the water jug problem, there are six possible actions:

1. Fill the first jug to its capacity.
2. Fill the second jug to its capacity.
3. Empty the first jug.
4. Empty the second jug.
5. Pour water from the first jug to the second jug until either the first jug is empty or the second jug is full.
6. Pour water from the second jug to the first jug until either the first jug is full or the second jug is empty.

"You are given two jugs, a 4-gallon one and a 3-gallon one. Neither have any measuring markers

on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2

gallons of water into the 4-gallon jug?".

**Production Rules:-**

R1: (x,y) --> (4,y) if x < 4

R2: (x,y) --> (x,3) if y < 3

R3: (x,y) --> (x-d,y) if x > 0

R4: (x,y) --> (x,y-d) if y > 0

R5: (x,y) --> (0,y) if x > 0

R6: (x,y) --> (x,0) if y > 0

R7: (x,y) --> (4,y-(4-x)) if x+y >= 4 and y > 0

R8: (x,y) --> (x-(3-y),y) if x+y >= 3 and x > 0

R9: (x,y) --> (x+y,0) if x+y =< 4 and y > 0

R10: (x,y) --> (0,x+y) if x+y =< 3 and x > 0

**Code :**

member(X,[X|\_]).

member(X,[Y|Z]):-member(X,Z).

move(X,Y,\_):-X=:=2,Y=:=0,write('done'),!.

move(X,Y,Z):-X<4,\+member((4,Y),Z),write("fill 4 jug"),nl,move(4,Y,[(4,Y)|Z]).

move(X,Y,Z):-Y<3,\+member((X,3),Z),write("fill 3 jug"),nl,move(X,3,[(X,3)|z]).

move(X,Y,Z):-X>0,\+member((0,Y),Z),write("pour 4 jug"),nl,move(0,Y,[(0,Y)|Z]).

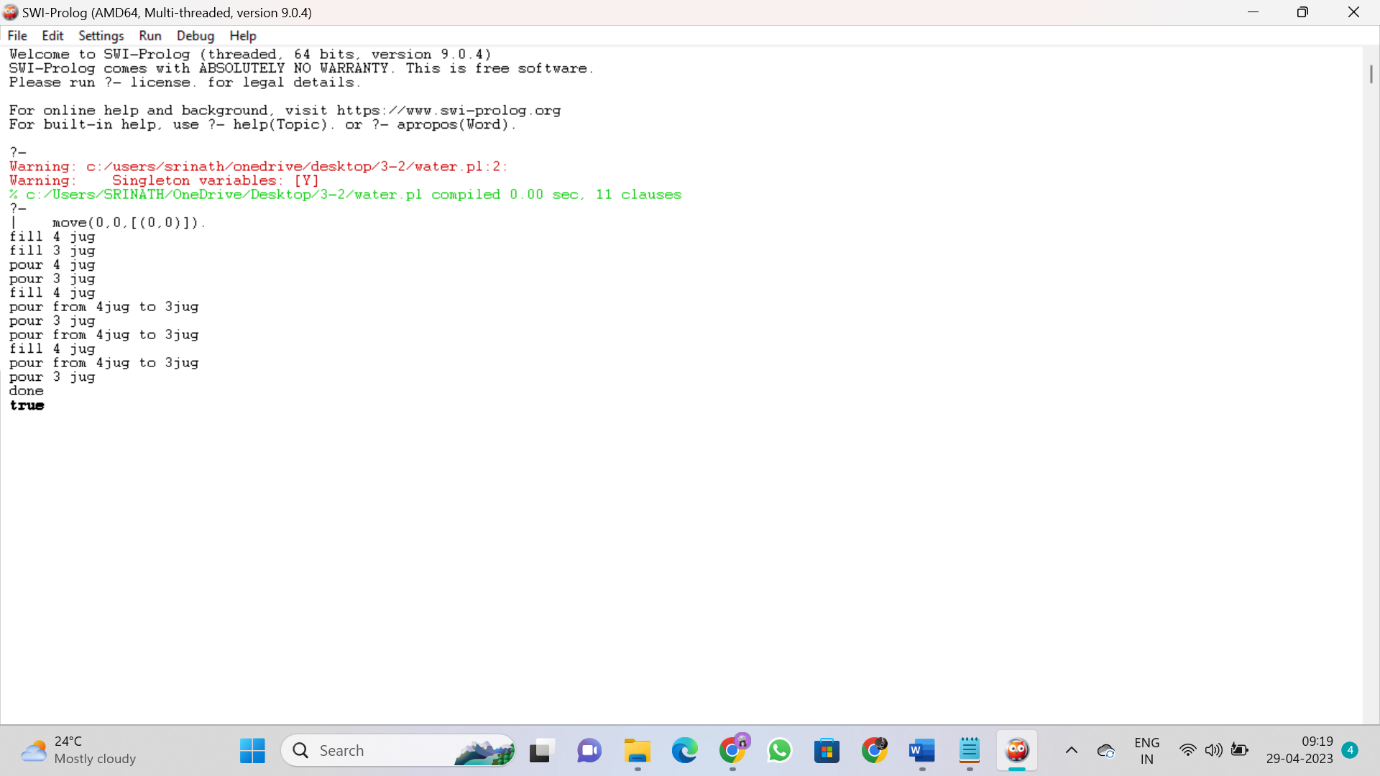
move(X,Y,Z):-Y>0,\+member((X,0),Z),write("pour 3 jug"),nl,move(X,0,[(X,0)|Z]).

move(X,Y,Z):-P is X+Y,P>=4,Y>0,K is 4-X,M is Y-K,\+member((4,M),Z),write("pour from 3jug to 4jug"),nl,move(4,M,[(4,M)|Z]).

move(X,Y,Z):-P is X+Y,P>=3,X>0,K is 3-Y,M is X-K,\+member((M,3),Z),write("pour from 4jug to 3jug"),nl,move(M,3,[(M,3)|Z]).

move(X,Y,Z):-K is X+Y,K<4,Y>0,\+member((K,0),Z),write("pour from 3jug to 4jug"),nl,move(K,0,[(K,0)|Z]).

move(X,Y,Z):-K is X+Y,K<3,X>0,\+member((0,K),Z),write("pour from 4jug to 3jug"),nl,move(0,K,[(0,K)|Z]).



**Output:**

?-

| move(0,0,[(0,0)]).

fill 4 jug

fill 3 jug

pour 4 jug

pour 3 jug

fill 4 jug

pour from 4jug to 3jug

pour 3 jug

pour from 4jug to 3jug

fill 4 jug

pour from 4jug to 3jug

pour 3 jug

done

true