**PRACTICAL-1**

**AIM:** Write a prolog Program to understand the concept of facts and queries.

**Source Code:**

% Facts about animals

animal(cat).

animal(dog).

animal(elephant).

color(cat, brown).

color(dog, black).

color(elephant, gray).

has\_legs(cat, 4).

has\_legs(dog, 4).

has\_legs(elephant, 4).

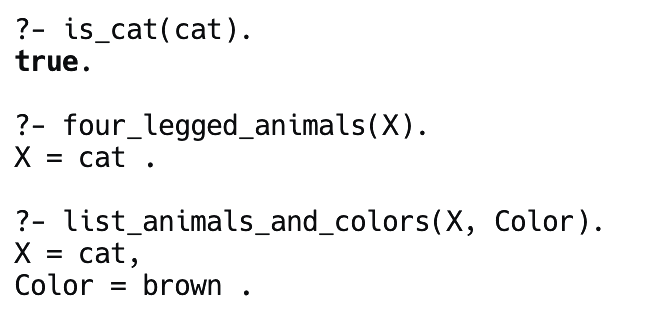
% Queries

is\_cat(X) :- animal(X), color(X, brown).

four\_legged\_animals(X) :- animal(X), has\_legs(X, 4).

list\_animals\_and\_colors(X, Color) :- animal(X), color(X, Color).

**Output:**



**PRACTICAL-2**

**AIM:** Write a prolog program to implement the following.

a. Factorial of a given number

b. Fibonacci of a given number

**Source Code:**

% Factorial of a number

factorial(0, 1).

factorial(N, F) :-

N > 0,

N1 is N - 1,

factorial(N1, F1),

F is N \* F1.

% Fibonacci of a number

fibonacci(0, 0).

fibonacci(1, 1).

fibonacci(N, F) :-

N > 1,

N1 is N - 1,

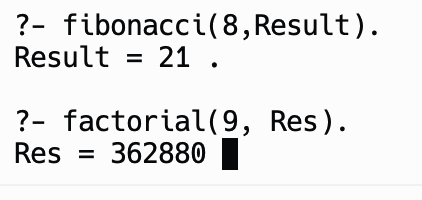
N2 is N - 2,

fibonacci(N1, F1),

fibonacci(N2, F2),

F is F1 + F2.

**Output:**

****

**PRACTICAL-3**

**AIM:** Write a program to implement Tic-Tac-Toe game problem.

**Source Code:**

% 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, \_, 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),

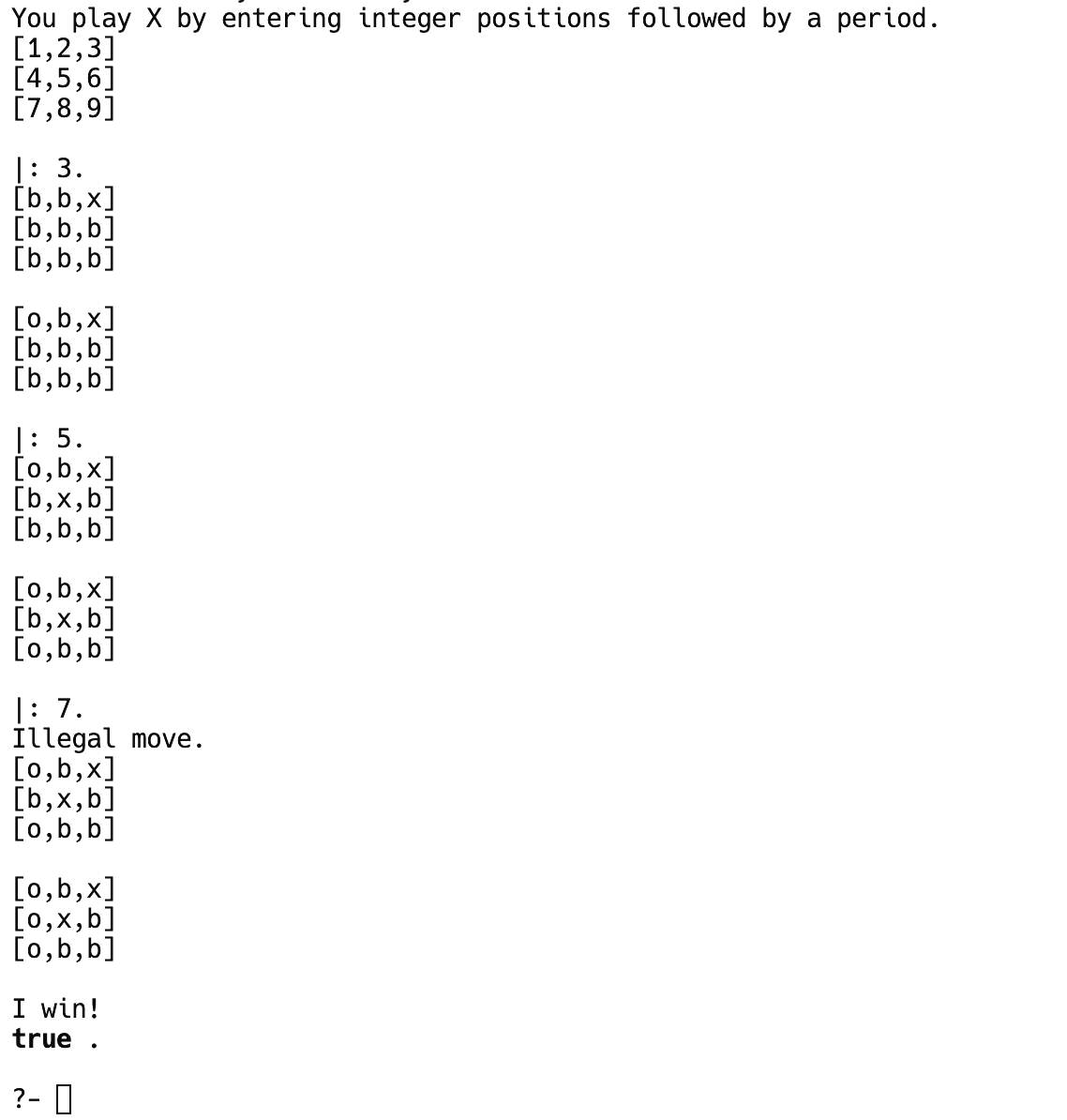
display(Newboard),

orespond(Newboard, Newnewboard),

display(Newnewboard),

playfrom(Newnewboard).

**Output:**



**PRACTICAL-4**

**AIM:** Write a program to implement BFS.(for 8 puzzle problem or Water Jug problem or any AI search problem.

**Source Code:**

eimport sys

import numpy as np

class Node:

def \_\_init\_\_(self, state, parent, action):

self.state = state

self.parent = parent

self.action = action

class StackFrontier:

def \_\_init\_\_(self):

self.frontier = []

def add(self, node):

self.frontier.append(node)

def contains\_state(self, state):

return any((node.state[0] == state[0]).all() for node in self.frontier)

def empty(self):

return len(self.frontier) == 0

def remove(self):

if self.empty():

raise Exception("Empty Frontier")

else:

node = self.frontier[-1]

self.frontier = self.frontier[:-1]

return node

class QueueFrontier(StackFrontier):

def remove(self):

if self.empty():

raise Exception("Empty Frontier")

else:

node = self.frontier[0]

self.frontier = self.frontier[1:]

return node

class Puzzle:

def \_\_init\_\_(self, start, startIndex, goal, goalIndex):

self.start = [start, startIndex]

self.goal = [goal, goalIndex]

self.solution = None

def neighbors(self, state):

mat, (row, col) = state

results = []

if row > 0:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row - 1][col]

mat1[row - 1][col] = 0

results.append(('up', [mat1, (row - 1, col)]))

if col > 0:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row][col - 1]

mat1[row][col - 1] = 0

results.append(('left', [mat1, (row, col - 1)]))

if row < 2:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row + 1][col]

mat1[row + 1][col] = 0

results.append(('down', [mat1, (row + 1, col)]))

if col < 2:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row][col + 1]

mat1[row][col + 1] = 0

results.append(('right', [mat1, (row, col + 1)]))

return results

def print(self):

solution = self.solution if self.solution is not None else None

print("Start State:\n", self.start[0], "\n")

print("Goal State:\n", self.goal[0], "\n")

print("\nStates Explored: ", self.num\_explored, "\n")

print("Solution:\n ")

for action, cell in zip(solution[0], solution[1]):

print("action: ", action, "\n", cell[0], "\n")

print("Goal Reached!!")

def does\_not\_contain\_state(self, state):

for st in self.explored:

if (st[0] == state[0]).all():

return False

return True

def solve(self):

self.num\_explored = 0

start = Node(state=self.start, parent=None, action=None)

frontier = QueueFrontier()

frontier.add(start)

self.explored = []

while True:

if frontier.empty():

raise Exception("No solution")

node = frontier.remove()

self.num\_explored += 1

if (node.state[0] == self.goal[0]).all():

actions = []

cells = []

while node.parent is not None:

actions.append(node.action)

cells.append(node.state)

node = node.parent

actions.reverse()

cells.reverse()

self.solution = (actions, cells)

return

self.explored.append(node.state)

for action, state in self.neighbors(node.state):

if not frontier.contains\_state(state) and self.does\_not\_contain\_state(state):

child = Node(state=state, parent=node, action=action)

frontier.add(child)

start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])

goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])

startIndex = (1, 1)

goalIndex = (1, 0)

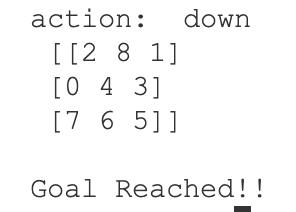
p = Puzzle(start, startIndex, goal, goalIndex)

p.solve()

p.print()

**Output:**





**PRACTICAL-5**

**AIM:** Write a program to implement DFS.(for 8 puzzle problem or Water Jug problem or any AI search problem)

**Source Code:**

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

def \_\_init\_\_(self):

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

# structure of the node

class nodes:

def \_\_init\_\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

def \_\_lt\_\_(self, nxt):

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

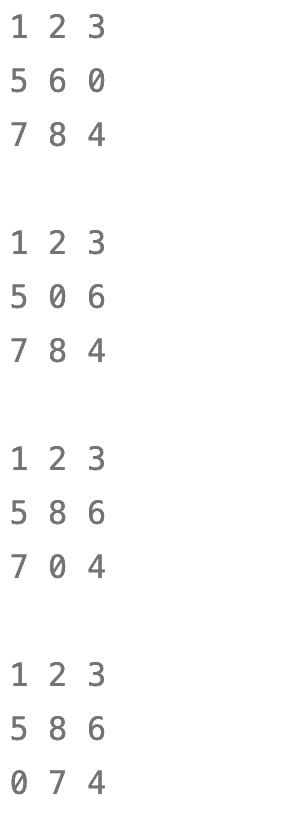
[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

**Output:**



**PRACTICAL-6**

**AIM:** Write a program to implement Single Player Game .(Using Heuristic Function)

**Source Code:**

e#include <stdio.h>

//Number of queens

int N;

//chessboard

int board[100][100];

//function to check if the cell is attacked or not

int is\_attack(int i,int j)

{

int k,l;

//checking if there is a queen in row or column

for(k=0;k<N;k++)

{

if((board[i][k] == 1) || (board[k][j] == 1))

return 1;

}

//checking for diagonals

for(k=0;k<N;k++)

{

for(l=0;l<N;l++)

{

if(((k+l) == (i+j)) || ((k-l) == (i-j)))

{

if(board[k][l] == 1)

return 1;

}

}

}

return 0;

}

int N\_queen(int n)

{

int i,j;

//if n is 0, solution found

if(n==0)

return 1;

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

{

//checking if we can place a queen here or not

//queen will not be placed if the place is being attacked

//or already occupied

if((!is\_attack(i,j)) && (board[i][j]!=1))

{

board[i][j] = 1;

//recursion

//wether we can put the next queen with this arrangment or not

if(N\_queen(n-1)==1)

{

return 1;

}

board[i][j] = 0;

}

}

}

return 0;

}

int main()

{

//taking the value of N

printf("Enter the value of N for NxN chessboard\n");

scanf("%d",&N);

int i,j;

//setting all elements to 0

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

{

board[i][j]=0;

}

}

//calling the function

N\_queen(N);

//printing the matix

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

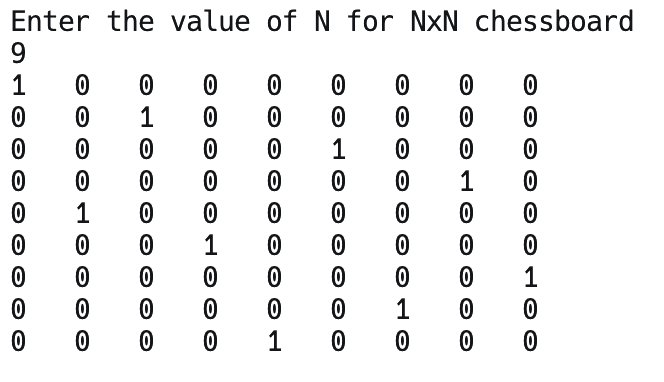
printf("%d\t",board[i][j]);

printf("\n");

}

}

**Output:**



**PRACTICAL-7**

**AIM:** Write a prolog program to perform the following operations of the list.

1. To display the elements of given list
2. To check given element is in the list or not
3. To print the last item of the list
4. To print the sum of elements of given list

**Source Code:**

display\_list([]).

display\_list([H|T]) :-

write(H), write(' '),

display\_list(T).

is\_element\_in\_list(Element, [Element|\_]).

is\_element\_in\_list(Element, [\_|T]) :-

is\_element\_in\_list(Element, T).

last\_element([X], X).

last\_element([\_|T], X) :-

last\_element(T, X).

sum\_list([], 0).

sum\_list([H|T], Sum) :-

sum\_list(T, RestSum),

Sum is H + RestSum.

list\_operations\_example :-

List = [1, 2, 3, 4, 5],

write('Here are the things in the list: '),

display\_list(List),

nl,

ElementToCheck = 3,

(is\_element\_in\_list(ElementToCheck, List) ->

write(ElementToCheck), write(' is in the list');

write(ElementToCheck), write(' is not in the list')

),

nl,

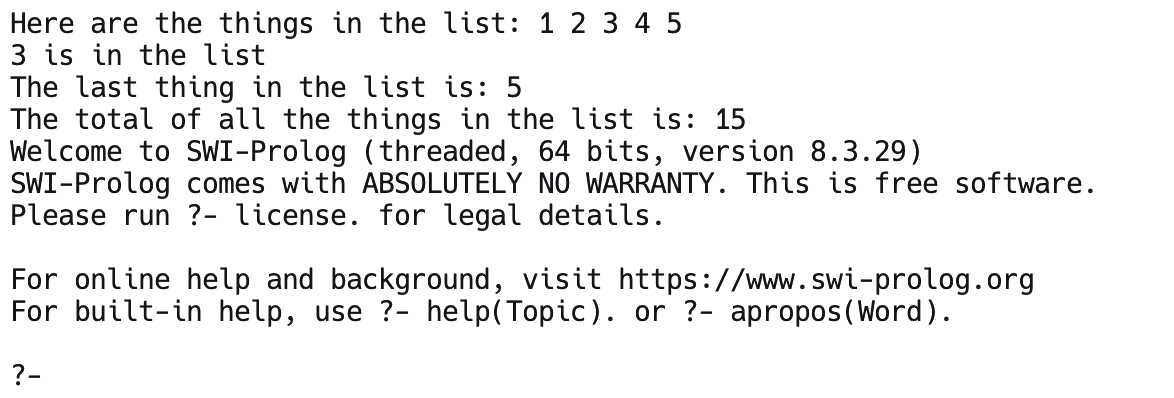
last\_element(List, Last),

write('The last thing in the list is: '), write(Last), nl,

sum\_list(List, Sum),

write('The total of all the things in the list is: '), write(Sum), nl.

**Output:**



**PRACTICAL-8**

**AIM:** Implement a Family Tree and define the following predicates:

i. parent(X,Y)

ii. father(X,Y)

iii. mother(X,Y)

iv. sister(X,Y)

v. brother(X,Y)

vi. grandfather(X,Y)

vii. grandmother(X,Y)

**Source Code:**

male(person1).

male(person2).

male(person4).

male(person6).

female(person3).

female(person5).

parent(person1, person2).

parent(person1, person4).

parent(person2, person5).

parent(person3, person5).

parent(person6, person4).

father(X, Y) :- parent(X, Y), male(X).

mother(X, Y) :- parent(X, Y), female(X).

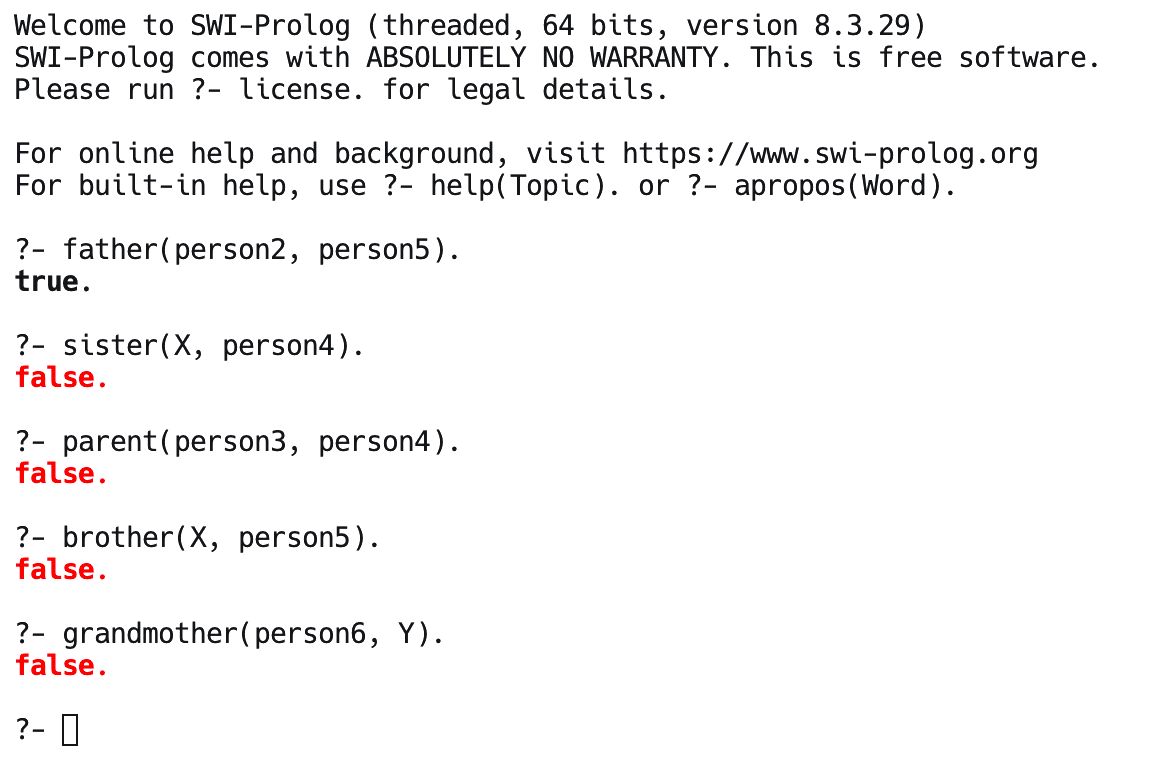
sister(X, Y) :- female(X), parent(Z, X), parent(Z, Y), X \= Y.

brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y), X \= Y.

grandfather(X, Y) :- father(X, Z), parent(Z, Y).

grandmother(X, Y) :- mother(X, Z), parent(Z, Y).

**Output:**



**PRACTICAL-9**

**AIM:** Assume given a set of facts of the form father(name1,name2) (name1 is the father of name2)

Define a predicate cousin(X,Y) which holds iff X and Y are cousins.

Define a predicate grandson(X,Y) which holds iff X is a grandson of Y.

Define a predicate descendent(X,Y) which holds iff X is a descendent of Y. Define a predicate grandparent(X,Y) which holds iff X is a grandparent of Y. Consider the following genealogical tree:

father(a,b). father(a,c). father(b,d). father(b,e). father(c,f).

Say which answers, and in which order, are generated by your definitions for the following queries in Prolog:

?- cousin(X,Y).

?- grandson(X,Y).

?- descendent(X,Y). ?-grandparent(X,Y).

**Source Code:**

father(a, b).

father(a, c).

father(b, d).

father(b, e).

father(c, f).

cousin(X, Y) :-

father(P1, X),

father(P2, Y),

P1 \= P2, % Ensure they have different fathers

siblings(P1, P2). % Check if their fathers are siblings

siblings(X, Y) :-

father(F, X),

father(F, Y),

X \= Y.

grandson(X, Y) :-

father(Y, Z),

father(Z, X).

descendant(X, Y) :-

father(Y, X).

descendant(X, Y) :-

father(Z, X),

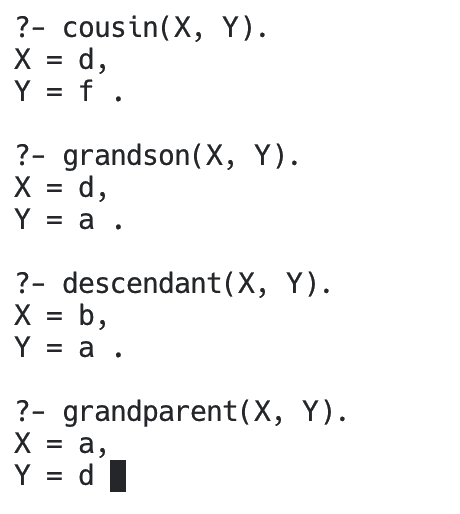
descendant(Z, Y).

grandparent(X, Y) :-

father(X, Z),

father(Z, Y).

**Output:**



**PRACTICAL-10**

**AIM:** Write a program to solve Tower of Hanoi problem using Prolog.

**Source Code:**

hanoi(N) :-

move(N, source, auxiliary, destination).

% Base case

move(0, \_, \_, \_) :- !.

% Recursive case

move(N, Source, Auxiliary, Destination) :-

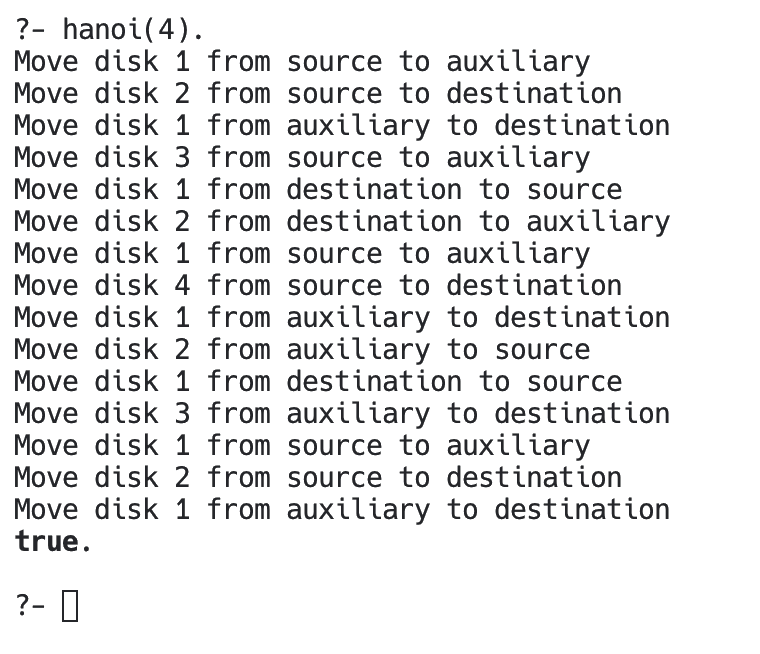
M is N - 1,

move(M, Source, Destination, Auxiliary), % Swap Auxiliary and Destination

format('Move disk ~w from ~w to ~w~n', [N, Source, Destination]),

move(M, Auxiliary, Source, Destination).

**Output:**



**PRACTICAL-11**

**AIM:** rite a program to Implement A\* Algorithm using Prolog.

**Source Code:**

import heapq

class Node(object):

"""For state representation"""

n = 0

def \_\_init\_\_(self, board, prev\_state=None):

assert len(board) == 9

self.board = board[:]

self.prev = prev\_state

self.step = 0

Node.n += 1

if self.prev:

self.step = self.prev.step + 1

def \_\_eq\_\_(self, other):

return self.board == other.board

def \_\_hash\_\_(self):

h = [0, 0, 0]

h[0] = self.board[0] << 6 | self.board[1] << 3 | self.board[2]

h[1] = self.board[3] << 6 | self.board[4] << 3 | self.board[5]

h[2] = self.board[6] << 6 | self.board[7] << 3 | self.board[8]

h\_val = 0

for h\_i in h:

h\_val = h\_val \* 31 + h\_i

return h\_val

def \_\_str\_\_(self):

string\_list = [str(i) for i in self.board]

sub\_list = (string\_list[:3], string\_list[3:6], string\_list[6:])

return "\n".join(["".join(l) for l in sub\_list])

def manhattan\_distance(self):

distance = 0

goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

for i in range(1, 9):

xs, ys = self.pos(self.board.index(i))

xg, yg = self.pos(goal.index(i))

distance += abs(xs - xg) + abs(ys - yg)

return distance

def hamming\_distance(self):

distance = 0

goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

for i in range(9):

if goal[i] != self.board[i]: distance += 1

return distance

def next(self):

next\_moves = []

i = self.board.index(0)

next\_moves = (self.moveUp(i), self.moveDown(i), self.moveRight(i),

self.moveLeft(i))

return [s for s in next\_moves if s]

def moveLeft(self, i):

x, y = self.pos(i)

if y > 0:

left\_state = Node(self.board, self)

left = self.sop(x, y - 1)

left\_state.swap(i, left)

return left\_state

def moveRight(self, i):

x, y = self.pos(i)

if y < 2:

right\_state = Node(self.board, self)

right = self.sop(x, y + 1)

right\_state.swap(i, right)

return right\_state

def moveUp(self, i):

x, y = self.pos(i)

if x > 0:

up\_state = Node(self.board, self)

up = self.sop(x - 1, y)

up\_state.swap(i, up)

return up\_state

def moveDown(self, i):

x, y = self.pos(i)

if x < 2:

down\_state = Node(self.board, self)

down = self.sop(x + 1, y)

down\_state.swap(i, down)

return down\_state

def swap(self, i, j):

self.board[j], self.board[i] = self.board[i], self.board[j]

def pos(self, index):

return (int(index / 3), index % 3)

def sop(self, x, y):

return x \* 3 + y

class PriorityQueue:

def \_\_init\_\_(self):

self.heap = []

self.count = 0

def push(self, item, priority):

entry = (priority, self.count, item)

heapq.heappush(self.heap, entry)

self.count += 1

def pop(self):

(\_, \_, item) = heapq.heappop(self.heap)

return item

def isEmpty(self):

return len(self.heap) == 0

def printPath(state):

path = []

while state:

path.append(state)

state = state.prev

path.reverse()

print("\n \n".join([str(state) for state in path]))

def astar(start, goal):

depth = 75

priotity\_queue = PriorityQueue()

h\_val = start.manhattan\_distance() + start.hamming\_distance()

f\_val = h\_val + start.step

priotity\_queue.push(start, f\_val)

visited = set()

found = False

while not priotity\_queue.isEmpty():

state = priotity\_queue.pop()

if state == goal:

found = state

break

if state in visited or state.step > depth:

continue

visited.add(state)

for s in state.next():

h\_val\_s = s.manhattan\_distance() + s.hamming\_distance()

f\_val\_s = h\_val\_s + s.step

priotity\_queue.push(s, f\_val\_s)

if found:

printPath(found)

print("Solution Found!")

else:

print("No solution found")

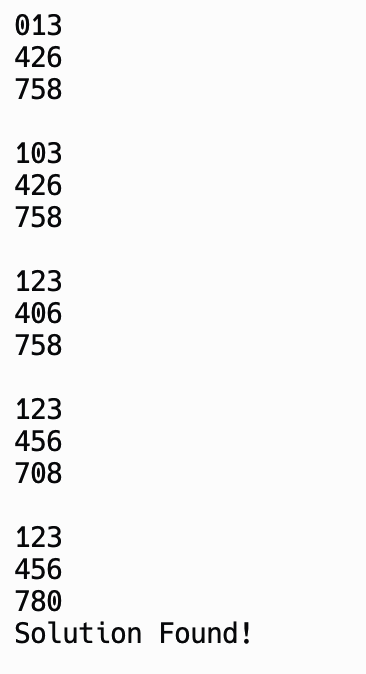
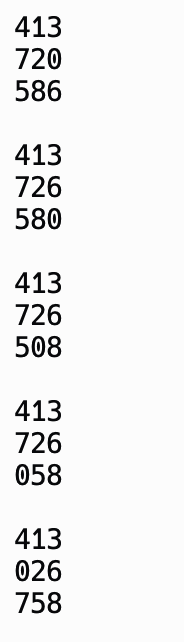
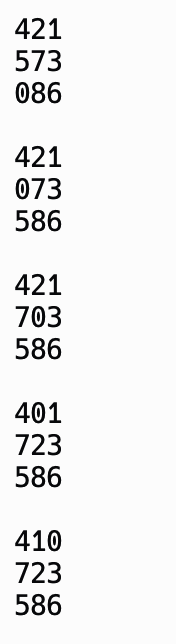
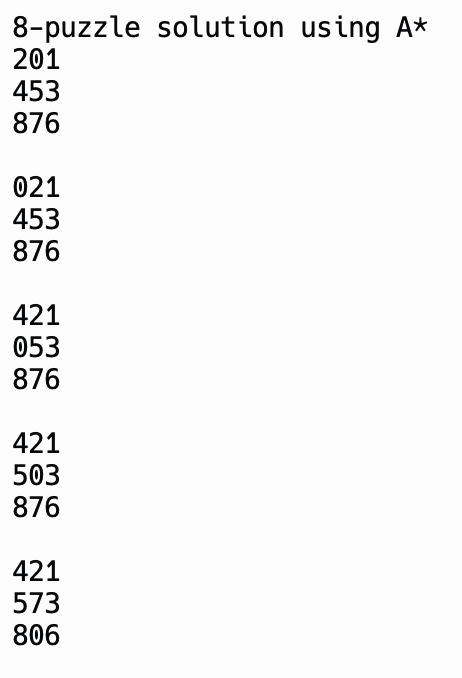
print("8-puzzle solution using A\*")

start = Node([2, 0, 1, 4, 5, 3, 8, 7, 6])

goal = Node([1, 2, 3, 4, 5, 6, 7, 8, 0])

astar(start, goal)

**Output:**



**PRACTICAL-12**

**AIM:** Write a program to solve N-Queens problem using C.

**Source Code:**

#include <stdio.h>

//Number of queens

int N;

//chessboard

int board[100][100];

//function to check if the cell is attacked or not

int is\_attack(int i,int j)

{

int k,l;

//checking if there is a queen in row or column

for(k=0;k<N;k++)

{

if((board[i][k] == 1) || (board[k][j] == 1))

return 1;

}

//checking for diagonals

for(k=0;k<N;k++)

{

for(l=0;l<N;l++)

{

if(((k+l) == (i+j)) || ((k-l) == (i-j)))

{

if(board[k][l] == 1)

return 1;

}

}

}

return 0;

}

int N\_queen(int n)

{

int i,j;

//if n is 0, solution found

if(n==0)

return 1;

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

{

//checking if we can place a queen here or not

//queen will not be placed if the place is being attacked

//or already occupied

if((!is\_attack(i,j)) && (board[i][j]!=1))

{

board[i][j] = 1;

//recursion

//wether we can put the next queen with this arrangment or not

if(N\_queen(n-1)==1)

{

return 1;

}

board[i][j] = 0;

}

}

}

return 0;

}

int main()

{

//taking the value of N

printf("Enter the value of N for NxN chessboard\n");

scanf("%d",&N);

int i,j;

//setting all elements to 0

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

{

board[i][j]=0;

}

}

//calling the function

N\_queen(N);

//printing the matix

for(i=0;i<N;i++)

{

for(j=0;j<N;j++)

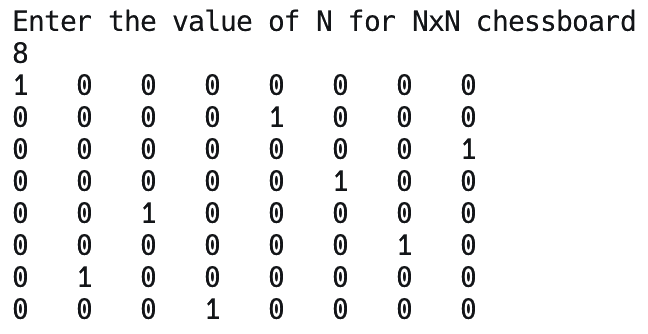
printf("%d\t",board[i][j]);

printf("\n");

}

}

**Output:**



**PRACTICAL-13**

**AIM:** Write a program to solve travelling salesman problem using C.

**Source Code:**

#include<stdio.h>

int ary[10][10], completed[10], n, cost = 0;

void takeInput() {

int i, j;

printf("Enter the number of villages: ");

scanf("%d", & n);

printf("\nEnter the Cost Matrix\n");

for (i = 0; i < n; i++) {

printf("\nEnter Elements of Row: %d\n", i + 1);

for (j = 0; j < n; j++)

scanf("%d", & ary[i][j]);

completed[i] = 0;

}

printf("\n\nThe cost list is:");

for (i = 0; i < n; i++) {

printf("\n");

for (j = 0; j < n; j++)

printf("\t%d", ary[i][j]);

}

}

void mincost(int city) {

int i, ncity;

completed[city] = 1;

printf("%d--->", city + 1);

ncity = least(city);

if (ncity == 999) {

ncity = 0;

printf("%d", ncity + 1);

cost += ary[city][ncity];

return;

}

mincost(ncity);

}

int least(int c) {

int i, nc = 999;

int min = 999, kmin;

for (i = 0; i < n; i++) {

if ((ary[c][i] != 0) && (completed[i] == 0))

if (ary[c][i] + ary[i][c] < min) {

min = ary[i][0] + ary[c][i];

kmin = ary[c][i];

nc = i;

}

}

if (min != 999)

cost += kmin;

return nc;

}

int main() {

takeInput();

printf("\n\nThe Path is:\n");

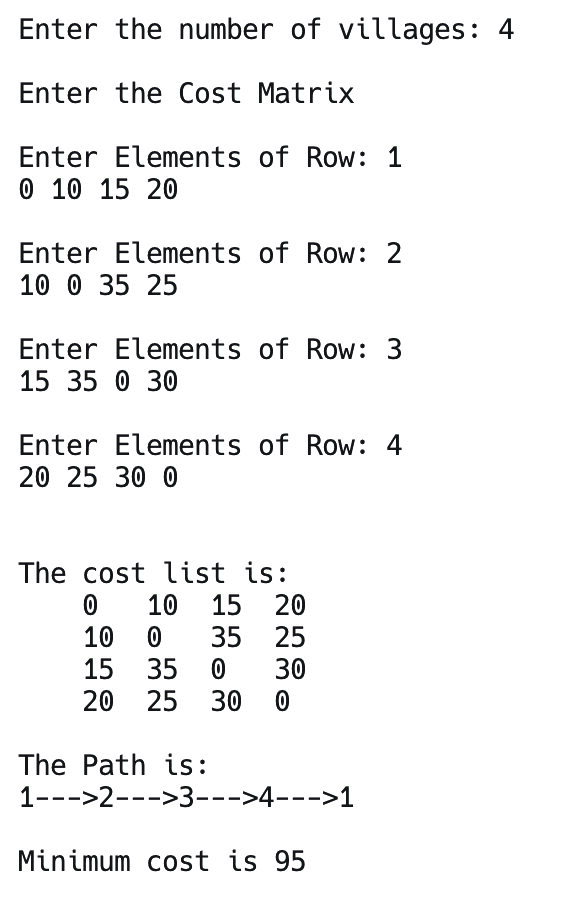
mincost(0); //passing 0 because starting vertex

printf("\n\nMinimum cost is %d\n ", cost);

return 0;

}

**Output:**



**PRACTICAL-14**

**AIM:** Write a program to solve Monkey Banana problem using Prolog.

**Source Code:**

% Predicates

in\_room(bananas).

in\_room(chair).

in\_room(monkey).

clever(monkey).

can\_climb(monkey, chair).

tall(chair).

% Rules

get\_on(X, Y):-

can\_climb(X, Y).

under(Y, Z):-

in\_room(X), in\_room(Y),

in\_room(Z), can\_climb(X, Y).

close\_to(X, Y,Z):-

get\_on(X, Y), under(Y, Z); tall(Y).

can\_reach(X, Y, Z):-

clever(X), close\_to(X, Y, Z).

can\_reach(X, Y, Z):-

clever(X), close\_to(X, Z, Y). % Added rule to consider swapping Y and Z.

**Output:**

