**PRACTICAL-1**

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

**Code:**

import java.util.Arrays;

import java.util.InputMismatchException;

import java.util.Scanner;

public class practical1\_506 {

static String[] board;

static String turn;

static String checkWinner()

{

for (int a = 0; a < 8; a++) {

String line = null;

switch (a) {

case 0:

line = board[0] + board[1] + board[2];

break;

case 1:

line = board[3] + board[4] + board[5];

break;

case 2:

line = board[6] + board[7] + board[8];

break;

case 3:

line = board[0] + board[3] + board[6];

break;

case 4:

line = board[1] + board[4] + board[7];

break;

case 5:

line = board[2] + board[5] + board[8];

break;

case 6:

line = board[0] + board[4] + board[8];

break;

case 7:

line = board[2] + board[4] + board[6];

break;

}

if (line.equals("XXX")) {

return "X";

}

else if (line.equals("OOO")) {

return "O";

}

}

for (int a = 0; a < 9; a++) {

if (Arrays.asList(board).contains(

String.valueOf(a + 1))) {

break;

}

else if (a == 8) {

return "draw";

}

}

System.out.println(

turn + "'s turn; enter a slot number to place "

+ turn + " in:");

return null;

}

static void printBoard()

{

System.out.println("|---|---|---|");

System.out.println("| " + board[0] + " | "

+ board[1] + " | " + board[2]

+ " |");

System.out.println("|-----------|");

System.out.println("| " + board[3] + " | "

+ board[4] + " | " + board[5]

+ " |");

System.out.println("|-----------|");

System.out.println("| " + board[6] + " | "

+ board[7] + " | " + board[8]

+ " |");

System.out.println("|---|---|---|");

}

public static void main(String[] args)

{

Scanner in = new Scanner(System.in);

board = new String[9];

turn = "X";

String winner = null;

for (int a = 0; a < 9; a++) {

board[a] = String.valueOf(a + 1);

}

System.out.println("Welcome to 3x3 Tic Tac Toe.");

printBoard();

System.out.println(

"X will play first. Enter a slot number to place X in:");

while (winner == null) {

int numInput;

try {

numInput = in.nextInt();

if (!(numInput > 0 && numInput <= 9)) {

System.out.println(

"Invalid input; re-enter slot number:");

continue;

}

}

catch (InputMismatchException e) {

System.out.println(

"Invalid input; re-enter slot number:");

continue;

}

if (board[numInput - 1].equals(

String.valueOf(numInput))) {

board[numInput - 1] = turn;

if (turn.equals("X")) {

turn = "O";

}

else {

turn = "X";

}

printBoard();

winner = checkWinner();

}

else {

System.out.println(

"Slot already taken; re-enter slot number:");

}

}

if (winner.equalsIgnoreCase("draw")) {

System.out.println(

"It's a draw! Thanks for playing.");

}

else {

System.out.println(

"Congratulations! " + winner

+ "'s have won! Thanks for playing.");

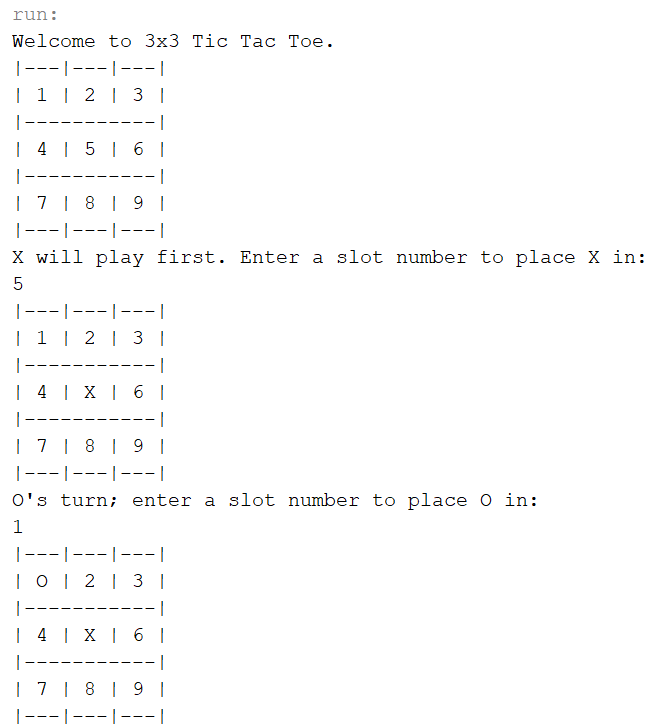
}

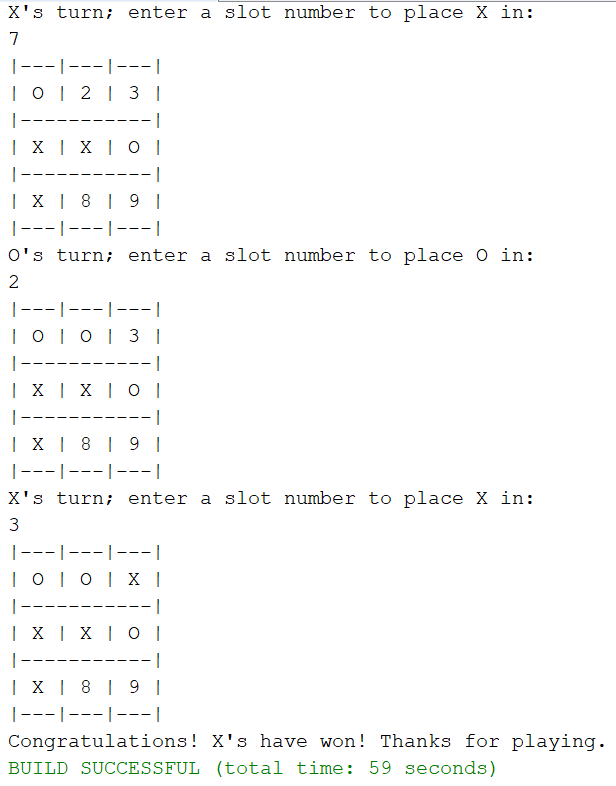
in.close();

}

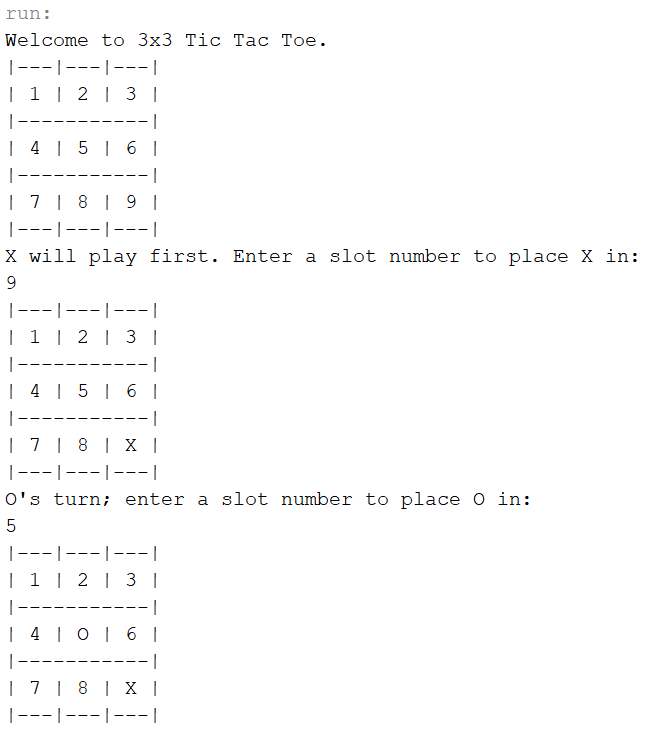
}

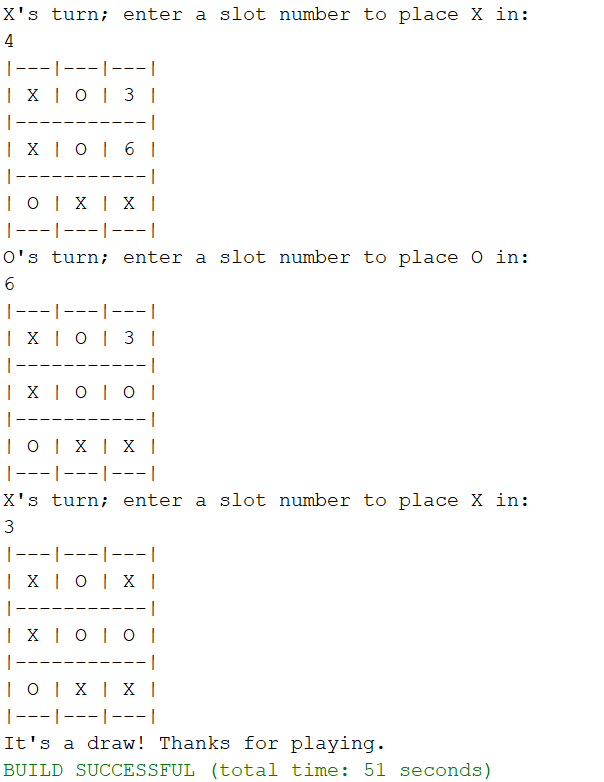
**OUTPUT:**





**GAME DRAW:**





**PRACTICAL-2**

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

**Code:**

from collections import deque

def BFS(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

q.append((0, 0))

while (len(q) > 0):

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or

u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",

path[i][1], ")")

break

q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print("No solution")

if \_\_name\_\_ == '\_\_main\_\_':

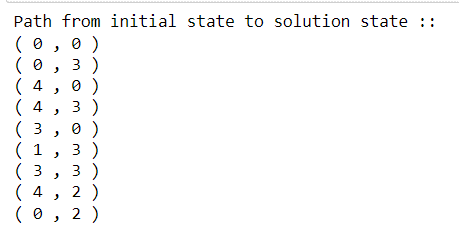
Jug1, Jug2, target = 4, 3, 2

print("Path from initial state "

"to solution state ::")

BFS(Jug1, Jug2, target)

**OUTPUT:**



**PRACTICAL-3**

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

**Code:**

from collections import deque

def BFS(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

q.append((0, 0))

while (len(q) > 0):

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or

u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

# Fill final state

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",

path[i][1], ")")

break

q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print("No solution")

if \_\_name\_\_ == '\_\_main\_\_':

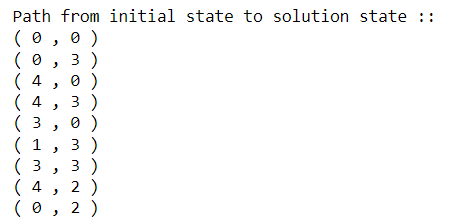
Jug1, Jug2, target = 4, 3, 2

print("Path from initial state "

"to solution state ::")

BFS(Jug1, Jug2, target)

**OUTPUT:**

****

**PRACTICAL-4**

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

**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]))

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

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

print("welcome to 8 puzzle problem. . . . . .")

printPath(start)

while start != goal:

action = input(" w s a d for up/down/left/right : ")

number = start.board.index(0)

if action == 'w':

start = start.moveUp(number)

elif action == 's':

start = start.moveDown(number)

elif action == 'a':

start = start.moveLeft(number)

elif action == 'd':

start = start.moveRight(number)

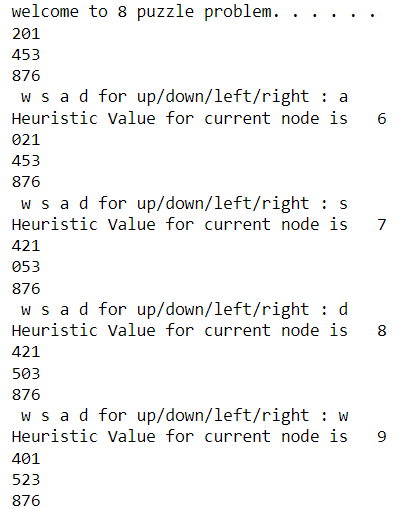
else:

print("choose right option")

print("Heuristic Value for current node is ",start.manhattan\_distance())

print(start)

**OUTPUT:**



**PRACTICAL-5**

**AIM: Write a program to Implement A\* Algorithm**

**Code:**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} #store distance from starting node

parents = {} # parents contains an adjacency map of all nodes

#distance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes

#so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

#node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to first

#n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

else:

if g[m] > g[n] + weight:

#update g(m)

g[m] = g[n] + weight

#change parent of m to n

parents[m] = n

#if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 5,

'D': 7,

'E': 3,

'F': 6,

'G': 5,

'H': 3,

'I': 1,

'J': 0

}

return H\_dist[n]

Graph\_nodes = {

'A': [('B', 6), ('F', 3)],

'B': [('A', 6), ('C', 3), ('D', 2)],

'C': [('B', 3), ('D', 1), ('E', 5)],

'D': [('B', 2), ('C', 1), ('E', 8)],

'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

'F': [('A', 3), ('G', 1), ('H', 7)],

'G': [('F', 1), ('I', 3)],

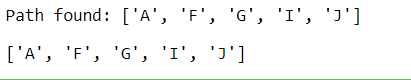
'H': [('F', 7), ('I', 2)],

'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

aStarAlgo('A', 'J')

**OUTPUT:**

****

**PRACTICAL-6**

**AIM:** **Write a program to implement mini-max algorithm for any game development.**

**Code:**

player, opponent = 'x', 'o'

def isMovesLeft(board) :

for i in range(3) :

for j in range(3) :

if (board[i][j] == '\_') :

return True

return False

def evaluate(b) :

for row in range(3) :

if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :

if (b[row][0] == player) :

return 10

elif (b[row][0] == opponent) :

return -10

for col in range(3) :

if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

if (b[0][col] == player) :

return 10

elif (b[0][col] == opponent) :

return -10

if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

if (b[0][0] == player) :

return 10

elif (b[0][0] == opponent) :

return -10

if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

if (b[0][2] == player) :

return 10

elif (b[0][2] == opponent) :

return -10

return 0

def minimax(board, depth, isMax) :

score = evaluate(board)

if (score == 10) :

return score

if (score == -10) :

return score

if (isMovesLeft(board) == False) :

return 0

if (isMax) :

best = -1000

for i in range(3) :

for j in range(3) :

if (board[i][j]=='\_') :

board[i][j] = player

best = max( best, minimax(board,depth + 1,not isMax) )

board[i][j] = '\_'

return best

else :

best = 1000

for i in range(3) :

for j in range(3) :

if (board[i][j] == '\_') :

# Make the move

board[i][j] = opponent

best = min(best, minimax(board, depth + 1, not isMax))

board[i][j] = '\_'

return best

def findBestMove(board) :

bestVal = -1000

bestMove = (-1, -1)

for i in range(3) :

for j in range(3) :

if (board[i][j] == '\_') :

board[i][j] = player

moveVal = minimax(board, 0, False)

board[i][j] = '\_'

if (moveVal > bestVal) :

bestMove = (i, j)

bestVal = moveVal

print("The value of the best Move is :", bestVal)

print()

return bestMove

# Driver code

board = [

[ 'x', 'o', 'x' ],

[ 'o', 'o', 'x' ],

[ '', '', '\_' ]

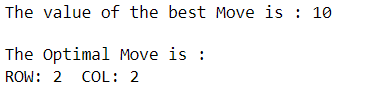
]

bestMove = findBestMove(board)

print("The Optimal Move is :")

print("ROW:", bestMove[0], " COL:", bestMove[1])

**OUTPUT:**

****

**PRACTICAL-7**

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

**Define a predicate brother(X,Y) which holds iff X and Y are brothers.**

**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.**

**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:**

**?- brother(X,Y).**

**?- cousin(X,Y).**

**?- grandson(X,Y).**

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

**Code:**

father(a,b).

father(a,c).

father(b,d).

father(b,e).

father(c,f).

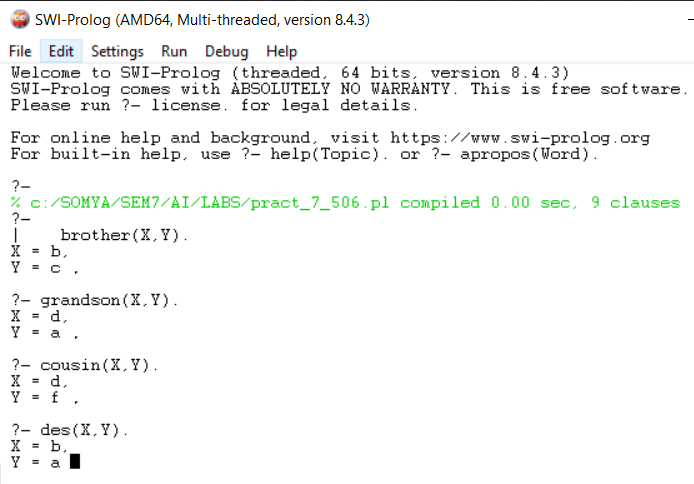
brother(X,Y) :- father(Z,X),father(Z,Y),not(X=Y).

cousin(X,Y) :- father(Z,X),father(W,Y),brother(Z,W).

grandson(X,Y) :- father(A,X),father(Y,A).

des(X,Y) :- father(Y,X).

**OUTPUT:**

****

**PRACTICAL-8**

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

**Code:**

move(1,X,Y,\_):- write('Move top disk from'),

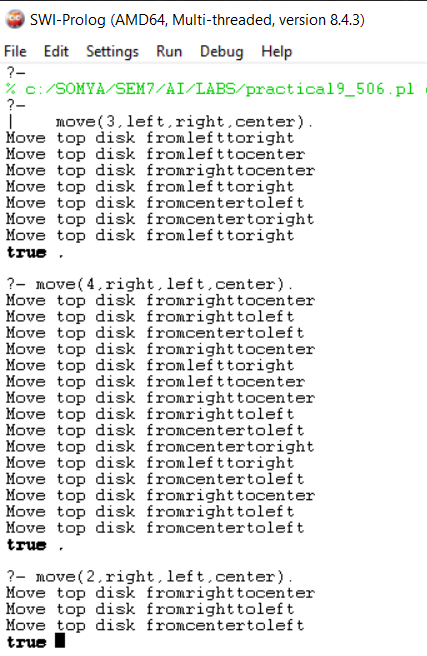
write(X), write('to'), write(Y), nl.

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:**



**PRACTICAL-9**

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

**Code:**

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]) :-

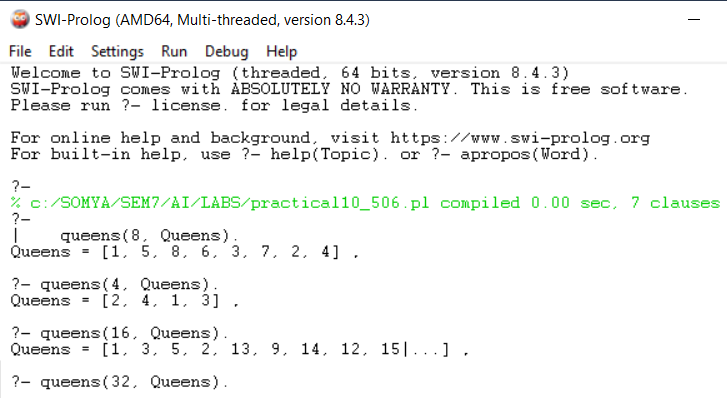
Row is Row0+1,

select(Col-Vars, [C|Cs], Board),

arg(Row, Vars, Row-Row),

queens(Board, Row, Solution).

**OUTPUT:**



**PRACTICAL-10**

**AIM:** **Write a program to solve 8 puzzle problem using Prolog.**

**Code:**

test(Plan):-

write('Initial state:'),nl,

Init= [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7), at(tile1,8), at(tile7,9)],

write\_sol(Init),

Goal= [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7), at(tile7,8), at(tile8,9)],

nl,write('Goal state:'),nl,

write(Goal),nl,nl,

solve(Init,Goal,Plan).

solve(State, Goal, Plan):-

solve(State, Goal, [], Plan).

%Determines whether Current and Destination tiles are a valid move.

is\_movable(X1,Y1) :- (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3 is X1 - Y1).

% This predicate produces the plan. Once the Goal list is a subset

% of the current State the plan is complete and it is written to

% the screen using write\_sol/1.

solve(State, Goal, Plan, Plan):-

is\_subset(Goal, State), nl,

write\_sol(Plan).

solve(State, Goal, Sofar, Plan):-

act(Action, Preconditions, Delete, Add),

is\_subset(Preconditions, State),

\+ member(Action, Sofar),

delete\_list(Delete, State, Remainder),

append(Add, Remainder, NewState),

solve(NewState, Goal, [Action|Sofar], Plan).

% The problem has three operators.

% 1st arg = name

% 2nd arg = preconditions

% 3rd arg = delete list

% 4th arg = add list.

% Tile can move to new position only if the destination tile is empty & Manhattan distance = 1

act(move(X,Y,Z),

[at(X,Y), at(empty,Z), is\_movable(Y,Z)],

[at(X,Y), at(empty,Z)],

[at(X,Z), at(empty,Y)]).

% Utility predicates.

% Check is first list is a subset of the second

is\_subset([H|T], Set):-

member(H, Set),

is\_subset(T, Set).

is\_subset([], \_).

% Remove all elements of 1st list from second to create third.

delete\_list([H|T], Curstate, Newstate):-

remove(H, Curstate, Remainder),

delete\_list(T, Remainder, Newstate).

delete\_list([], Curstate, Curstate).

remove(X, [X|T], T).

remove(X, [H|T], [H|R]):-

remove(X, T, R).

write\_sol([]).

write\_sol([H|T]):-

write\_sol(T),

write(H), nl.

append([H|T], L1, [H|L2]):-

append(T, L1, L2).

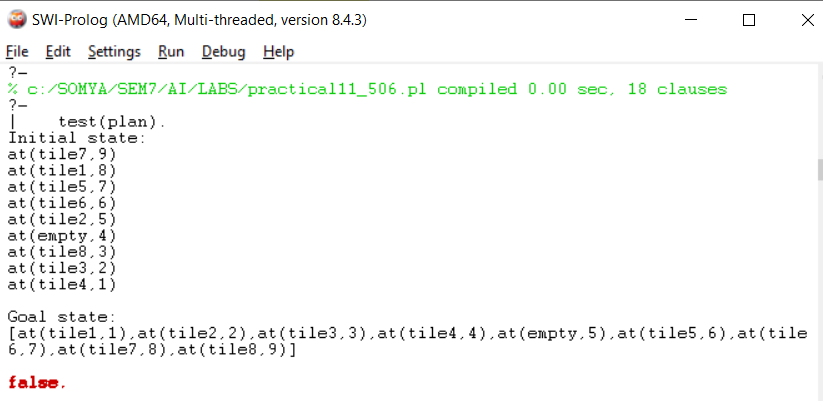
append([], L, L).

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

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

member(X, T).

**OUTPUT:**



**PRACTICAL-11**

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

**Code:**

city(boston).

city(new\_york).

city(phoenix).

city(portland).

city(tucson).

city(seattle).

city(washington).

c(boston,new\_york, 211).

c(boston,phoenix, 2690).

c(boston,portland, 3119).

c(boston,seattle,3088).

c(boston,tucson, 2632).

c(boston,washington,442).

c(new\_york,phoenix,2485).

c(new\_york,portland,2925).

c(new\_york,seattle,2894).

c(new\_york,tucson,2427).

c(new\_york,washington,237).

c(phoenix,portland,1347).

c(phoenix,seattle,1487).

c(phoenix,tucson,114).

c(phoenix,washington,2350).

c(portland,seattle,175).

c(portland,tucson,1460).

c(portland,washington,2819).

c(seattle,tucson,1602).

c(seattle,washington,2788).

c(tucson,washington,2279). cost(A,B,V):-c(A,B,V);c(B,A,V).

/\* perm(A,B): B is a permutation of A; Generator of B's \*/

perm([],[]).

perm([A|S],[A|T]):-perm(S,T).

perm([A|S],[B|T]):-perm(S,T1), exchange(A,B,T1,T).

/\* exchange A for B in set S to obtain set T\*/

exchange(A,B,[B|T],[A|T]).

exchange(A,B,[C|S],[C|T]):-exchange(A,B,S,T).

cities(P):-setof(C,city(C),P).

walk([C|W]):-cities([C|P]),perm(P,W).

ccost([A|R],V):-ccost([A|R],V,A).

ccost([A],V,F):-cost(A,F,V),!.

ccost([A,B|R],V,F):- cost(A,B,V1), ccost([B|R],V2,F), V is V1+V2.

itinerary(W,V):- walk(W),ccost(W,V).

solve(X):-setof(V-W,itinerary(W,V),B),best(B,X).

best([K-P|R],X):-best(R,L-Q),better(K-P,L-Q,X),!.

best([X],X).

better(K-P,L-\_,K-P):-K<L,!.

better(\_,X,X).

**OUTPUT:**

