Practical – 1

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

Code:

```
import java.util.Scanner;
public class App {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String board[][] = new String[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (i == 2) {
                    board[i][j] = "  ";
                } else {
                    board[i][j] = "___";
                }
            }
        }
        int player = 1;
        int total_moves = 0;
        print_matrix(board);
        while (true) {
            System.out.println("Turn to player " + player);int
            x = input.nextInt();
            int y = input.nextInt();
            if ((x > 3 || x < 0) || (y > 3 || y < 0) || !((board[x][y].equals("______")) ||
(board[x][y].equals(" ")))) {
                System.out.println("Invalid input ");
                print_matrix(board);
                continue;
            }
            if (player == 1) {
             board[x][y] =  "_O_";
```

```
            player = 2;
          } else {
            board[x][y] = "_X_";
            player = 1;
          }
          print_matrix(board);
          total_moves++;
          int c1 = checkVertical(board);
          int c2 = checkHorizontal(board);
          int c3 = checkDiagonal(board);
          if (c1 == 1 || c2 == 1 || c3 == 1) {
            System.out.println("player 1 has won");
            break;
          } else if (c1 == 2 || c2 == 2 || c3 == 2) {
            System.out.println("player 2 has won");
            break;
          }
          if (total_moves == 9) {
            System.out.println("Game has ended in draw");
            break;
          }
        }
      }
      static int checkVertical(String board[][]) {
        for (int i = 0; i < 3; i++) {
          if ((board[0][i].equals(board[1][i])) && (board[1][i].equals(board[2][i]))) {
            if (board[0][i].equals("_O_"))
              return 1;
            else if (board[0][i].equals("_X_"))
              return 2;
          }
        }
        return 3;
```

```
}
static int checkHorizontal(String board[][]) {
   for (int i = 0; i < 3; i++) {
      if ((board[i][0].equals(board[i][1])) && (board[i][1].equals(board[i][2]))) {if
         (board[i][0].equals("_O_"))
            return 1;
         else if (board[i][0].equals("_X_"))
            return 2;
      }
   }
   return 3;
}
static int checkDiagonal(String board[][]) {
   if ((board[0][0].equals(board[1][1])) && (board[1][1].equals(board[2][2]))) {
      if (board[1][1].equals("_O_"))
         return 1;
      else if (board[1][1].equals("_X_"))
         return 2;
   }
   if ((board[0][2].equals(board[1][1])) && (board[1][1].equals(board[2][0]))) {
      if (board[1][1].equals("_O_"))
         return 1;
      else if (board[1][1].equals("_X_"))
         return 2;
   }
   return 3;
}
static void print_matrix(String board[][]) {
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
         if (j == 2) {
            System.out.print(board[i][j]);
            break;
```

```
                }
                System.out.print(board[i][j] + "|");
            }
            System.out.println();
        }
    }
}
```

➢ **OUTPUT:**

```
PS F:\vscode_for_java\tic_toc_toe\src> java App
___|___|___
___|___|___
   |   |
Turn to player 1
0 0
_O_|___|___
___|___|___
   |   |
Turn to player 2
2 0
_O_|___|___
___|___|___
_X_|   |
Turn to player 1
1 1
_O_|___|___
___|_O_|___
_X_|   |
Turn to player 2
2 1
_O_|___|___
___|_O_|___
_X_|_X_|
Turn to player 1
2 2
_O_|___|___
___|_O_|___
_X_|_X_|_O_
player 1 has won
PS F:\vscode_for_java\tic_toc_toe\src>
```

Practical – 2

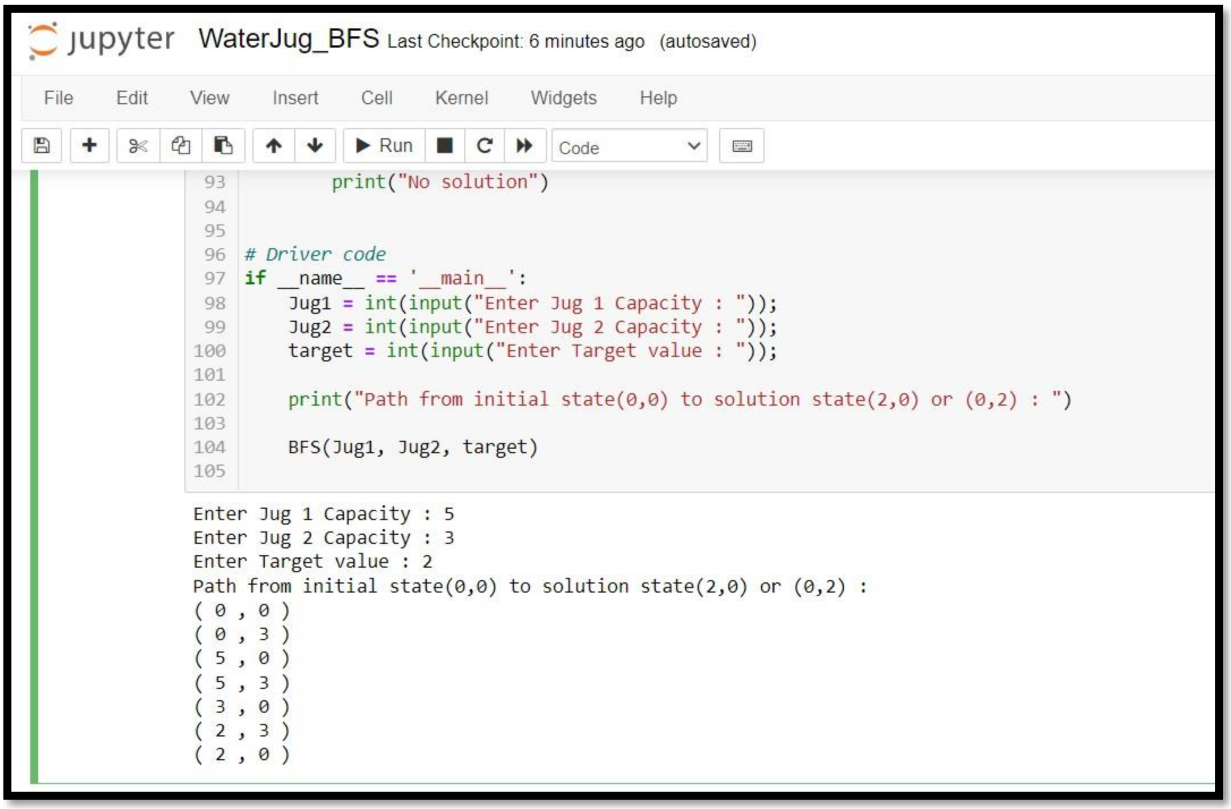
AIM: Write a program to implement BFS Water Jug problem.

Code:

```
from collections import deque
def BFS(a, b, target):
    m = {}
    isSolvable = Falsepath = []
    q = deque()
    q.append((0, 0))
    while len(q) > 0:
        u = q.popleft()
         q.pop()
        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:
                    # Fill final state
                    path.append([u[0], 0])
            else:
                if u[0] != 0:
                    path.append([0, u[1]])
            # Print the solution
            z = len(path)
            for i in range(sz):
                print("(", path[i][0], ",", path[i][1], ")")
                break
```

```
        # If we have not reached final state
        # then, start developing intermediate
        # states to reach solution state
        q.append([u[0], b])
        # Fill Jug2
        q.append([a, u[1]])
        # Fill Jug1
        for ap in range(max(a, b) + 1):
            # Pour amount ap from Jug2 to Jug1
            c = u[0] + ap
            d = u[1] - ap
            # Check if this state is possible or not
            if c == a or (d == 0 and d >= 0):
                q.append([c, d])
            # Pour amount ap from Jug 1 to Jug2
            c = u[0] - ap
            d = u[1] + ap
            # Check if this state is possible or not
            if (c == 0 and c >= 0) or d == b:
                q.append([c, d])
        # Empty Jug2 q.append([a, 0])
        # Empty Jug1 q.append([0, b])
    # No, solution exists if ans=0
    if not isSolvable:
        print("No solution")
# Driver code
if__name__== "__main__":
    Jug1 = int(input("Enter Jug 1 Capacity : "));
    Jug2 = int(input("Enter Jug 2 Capacity : "));
    target = int(input("Enter Target value : "));
    print("Path from initial state(0,0) to solution state(2,0) : ")
    BFS(Jug1, Jug2, target)
```

- **OUTPUT:**


jupyter WaterJug_BFS Last Checkpoint: 6 minutes ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run Code
        print("No solution")

# Driver code
if __name__ == '__main__':
    Jug1 = int(input("Enter Jug 1 Capacity : "));
    Jug2 = int(input("Enter Jug 2 Capacity : "));
    target = int(input("Enter Target value : "));

    print("Path from initial state(0,0) to solution state(2,0) or (0,2) : ")

    BFS(Jug1, Jug2, target)

Enter Jug 1 Capacity : 5
Enter Jug 2 Capacity : 3
Enter Target value : 2
Path from initial state(0,0) to solution state(2,0) or (0,2) :
(0 , 0)
(0 , 3)
(5 , 0)
(5 , 3)
(3 , 0)
(2 , 3)
(2 , 0)

Practical – 3

AIM: Write a program to implement DFS for Water Jug problem.

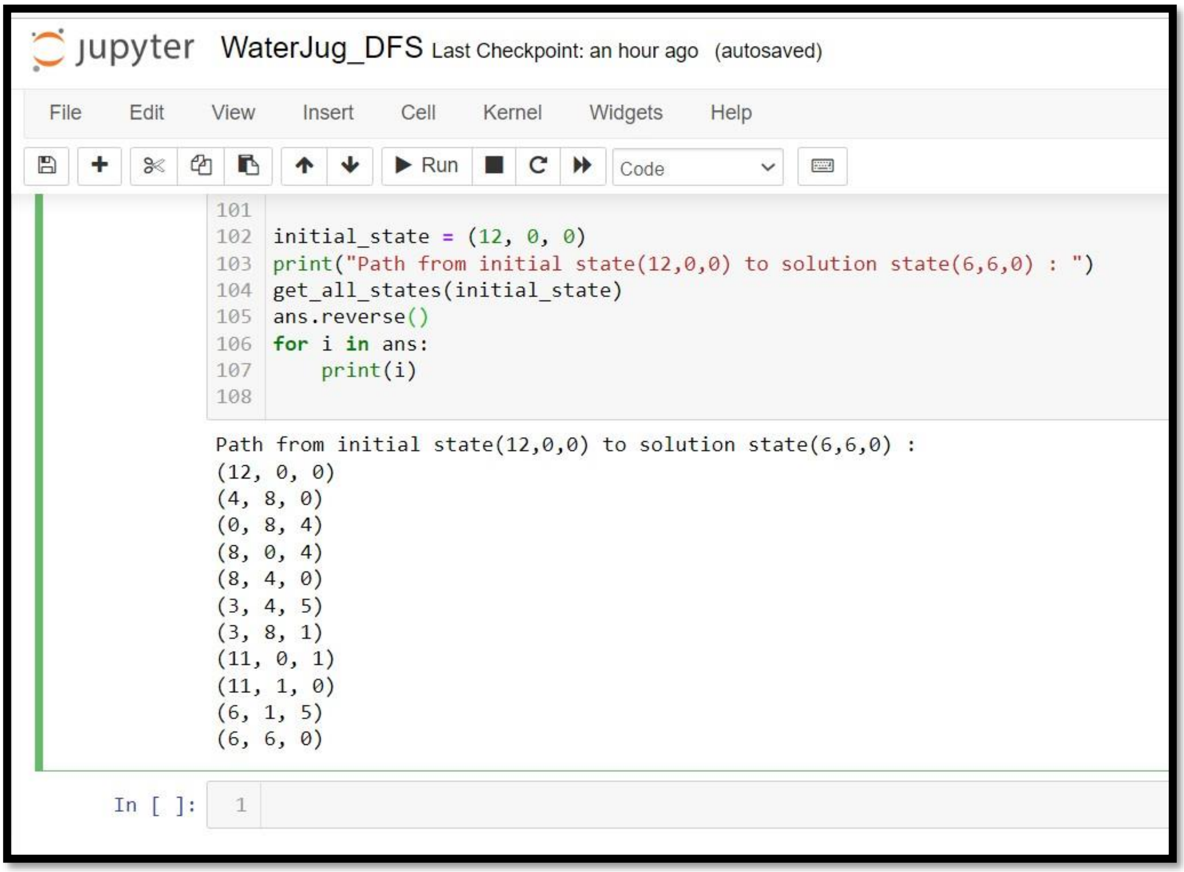
Code:

```
# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)
capacity = (12, 8, 5)
# Maximum capacities of 3 jugs -> x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = {}
# store solution path
ans = []
def get_all_states(state):
    # Let the 3 jugs be called a,b,c
    a = state[0]
    b = state[1]
    c = state[2]
    if a == 6 and b == 6:
        ans.append(state)
        return True
    # if current state is already visited earlier
    if (a, b, c) in memory:
        return False
    memory[(a, b, c)] = 1
    # empty jug a
    if a > 0:
        # empty a into b
        if a + b <= y:
            if get_all_states((0, a + b, c)):
                ans.append(state)
```

```
            return True
        else:
            if get_all_states((a - (y - b), y, c)):
                ans.append(state)
                return True
        # empty a into c
        if a + c <= z:
            if get_all_states((0, b, a + c)):
                ans.append(state)
                return True
        else:
            if get_all_states((a - (z - c), b, z)):
                ans.append(state)
                return True
    # empty jug b
    if b > 0:
        # empty b into a
        if a + b <= x:
            if get_all_states((a + b, 0, c)):
                ans.append(state)
                return   True
        else:
            if get_all_states((x, b - (x - a), c)):
                ans.append(state)
                return True #
        empty b into c if
        b + c <= z:
            if get_all_states((a, 0, b + c)):
                ans.append(state)
                return   True
        else:
            if get_all_states((a, b - (z - c), z)):
                ans.append(state)
```

```
            return True
    # empty jug c
    if c > 0:
        # empty c into a
        if a + c <= x:
            if get_all_states((a + c, b, 0)):
                ans.append(state)
                return   True
        else:
            if get_all_states((x, b, c - (x - a))):
                ans.append(state)
                return True
                #empty c into b
                if b + c <=y:
            if get_all_states((a, b + c, 0)):
                ans.append(state)
                return   True
        else:
            if get_all_states((a, y, c - (y - b))):
            ans.append(state)
                return True
    return False
initial_state = (12, 0, 0)
print("Path from initial state(12,0,0) to solution state(6,6,0) : ")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

- **OUTPUT:**




```python

initial_state = (12, 0, 0)
print("Path from initial state(12,0,0) to solution state(6,6,0) : ")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)

```

```
Path from initial state(12,0,0) to solution state(6,6,0) :
(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

Practical – 4

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

Code:

```
import random
print("# ---------------------- #")
print("| NUMBER GUESSING GAME |")
print("# ---------------------- #")
print("\n")
print("Range of Random Numbers.")
start = int(input("Enter Lower Value: "))
end = int(input("Enter Higher Value: "))
number = random.randint(start, end)
print("\n")
while True:
   guess = int(input("Guess the number: "))
   if guess > number:
      print("\nHmmm, try a lower number.  \n")
   elif guess < number:
      print("\nGo a little higher\n")
   else:
      print("Right on! Well done!")
      break
```

➢ OUTPUT:


jupyter SinglePlayerGame_HeuristicFuction Last Checkpoint: 36 minutes ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run
Code
else:
print("Right on! Well done!")
break
#----------------------#
| NUMBER GUESSING GAME |
#----------------------#
Range of Random Numbers.
Enter Lower Value: 5
Enter Higher Value: 10
Guess the number: 7
Go a little higher
Guess the number: 9
Right on! Well done!
In [ ]:

Practical – 5

AIM: Write a program to Implement A* Algorithm.

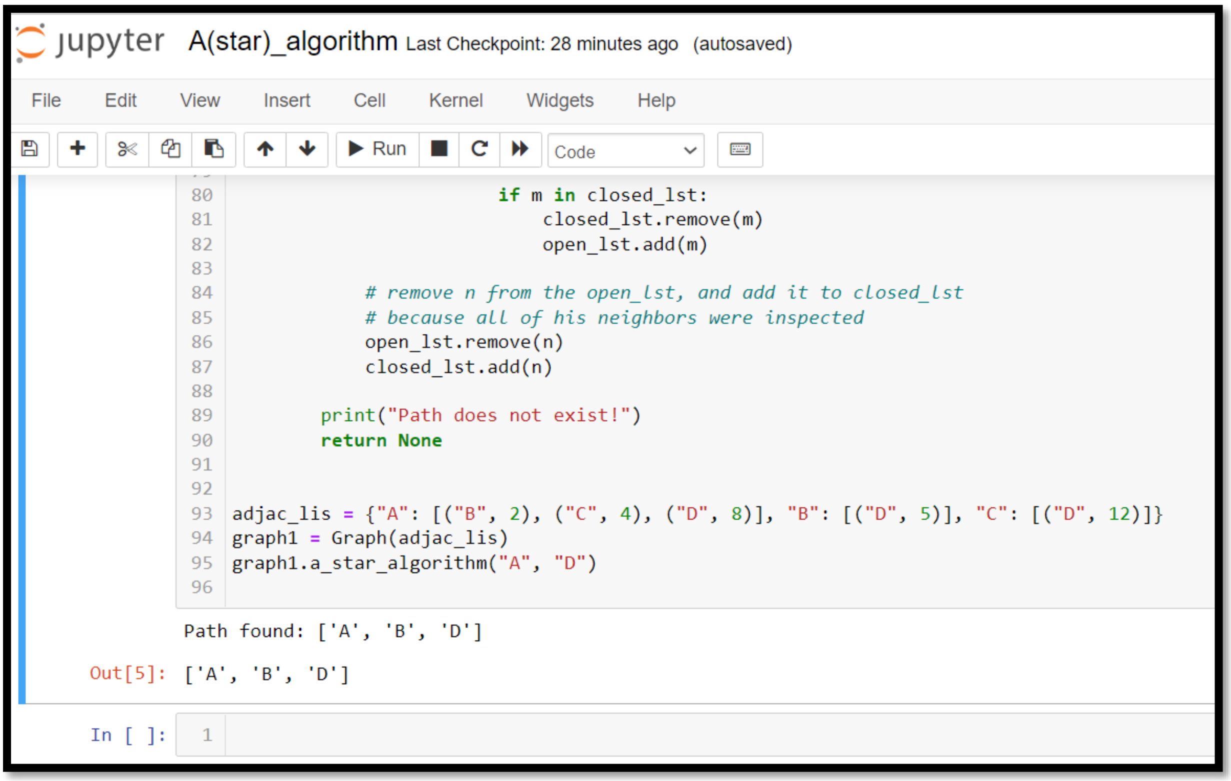
Code:

```
from collections import deque
class Graph:
    def_init_(self, adjac_lis):
        self.adjac_lis = adjac_lis
    def get_neighbors(self, v):
        return self.adjac_lis[v]
    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
        H = {"A": 1, "B": 1, "C": 1, "D": 1}
        return H[n]
    def a_star_algorithm(self, start, stop):
        # In this open_lst is a list of nodes which have been visited, but who's
        # neighbours haven't all been always inspected, It starts off with the start
        # node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = set([start])
        closed_lst = set([])
        # poo has present distances from start to all other nodes
        # the default value is +infinity
        poo = {}
        poo[start] = 0
        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start
        while len(open_lst) > 0:
            n = None
            # it will find a node with the lowest value of f() -
            for v in open_lst:
```

```
        if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):n = v
    if n == None:
        print("Path does not exist!")
        return None
    # if the current node is the stop then we start again from start
    if n == stop:
        reconst_path = []
        while par[n] != n:
            reconst_path.append(n)
        n = par[n]
        reconst_path.append(start)
        reconst_path.reverse()
        print("Path found: {}".format(reconst_path))
        return reconst_path
    # for all the neighbors of the current node dofor
    (m, weight) in self.get_neighbors(n):
        # if the current node is not presentin both open_lst and closed_lst
        # add it to open_lst and note n as it's par
        if m not in open_lst and m not in closed_lst:
            open_lst.add(m)
            par[m] = n
            poo[m] = poo[n] + weight
        # otherwise, check if it's quicker to first visit n, then m# and if it is, update par
        data and poo data
        # and if the node was in the closed_lst, move it to open_lst
        else:
            if poo[m] > poo[n] + weight:
                poo[m] = poo[n] + weight
                par[m] = n
                if m in closed_lst:
                    closed_lst.remove(m)
                    open_lst.add(m)
    # remove n from the open_lst, and add it to closed_lst
```

```
            # because all of his neighbors were inspected
            open_lst.remove(n)
            closed_lst.add(n)
        print("Path does not exist!")
        return None
adjac_lis = {"A": [("B", 2), ("C", 4), ("D", 8)], "B": [("D", 5)], "C": [("D", 12)]}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm("A", "D")
```

➢ **OUTPUT:**



Practical – 6

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

Code:

```
# Consider Tic Tac Toe game
# Find the next optimal move for a player
player, opponent = "x", "o"
# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board):
   for i in range(3): for
      j in range(3):
         if board[i][j] == "_":
            return True
   return False
# This is the evaluation function as discussed
def evaluate(b):
   # Checking for Rows for X or O victory.
   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
   # Checking for Columns for X or O victory.
   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
```

```
    # Checking for Diagonals for X or O victory.
    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
    # Else if none of them have won then return 0
    return 0
# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax):
    score = evaluate(board)
    # If Maximizer has won the game return his/her
    # evaluated score
    if score == 10:
        return score
    # If Minimizer has won the game return his/her
    # evaluated score
    if score == -10:
        return score
    # If there are no more moves and no winner then
    # it is a tie
    if isMovesLeft(board) == False:
        return 0
    # If this maximizer's move
    if isMax:
        best = -1000
```

```
        # Traverse all cells
        for i in range(3):
            for j in range(3):
                # Check if cell is empty
                if board[i][j] == "_":
                    # Make the move
                    board[i][j] = player
                    # Call minimax recursively and choose#
                    the maximum value
                    best = max(best, minimax(board, depth + 1, not isMax))
                    # Undo the move
                    board[i][j] = "_"
        return best
    # If this minimizer's move
    else:
        best = 1000
        # Traverse all cells
        for i in range(3):
            for j in range(3):
                # Check if cell is empty
                if board[i][j] == "_":
                    # Make the move
                    board[i][j] = opponent
                    # Call minimax recursively and choose#
                    the minimum value
                    best = min(best, minimax(board, depth + 1, not isMax))
                    # Undo the move
                    board[i][j] = "_"
        return best
# This will return the best possible move for the player
def findBestMove(board):
    bestVal = -1000
    bestMove = (-1, -1)
```

```
    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
     # value.
    for   i   in   range(3):
        for j in range(3):
            # Check if cell is empty
            if board[i][j] == "_":
                # Make the move
                board[i][j] = player
                # compute evaluation function for this#
                move.
                moveVal = minimax(board, 0, False)
                # Undo the move
                board[i][j] = "_"
                # If the value of the current move is
                # more than the best value, then update
                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:**

jupyter Game_MinMax Last Checkpoint: 6 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Code

```python
                # best/
                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])
```

```
The value of the best Move is : 10

The Optimal Move is :
ROW: 2  COL: 2
```

In [ ]:

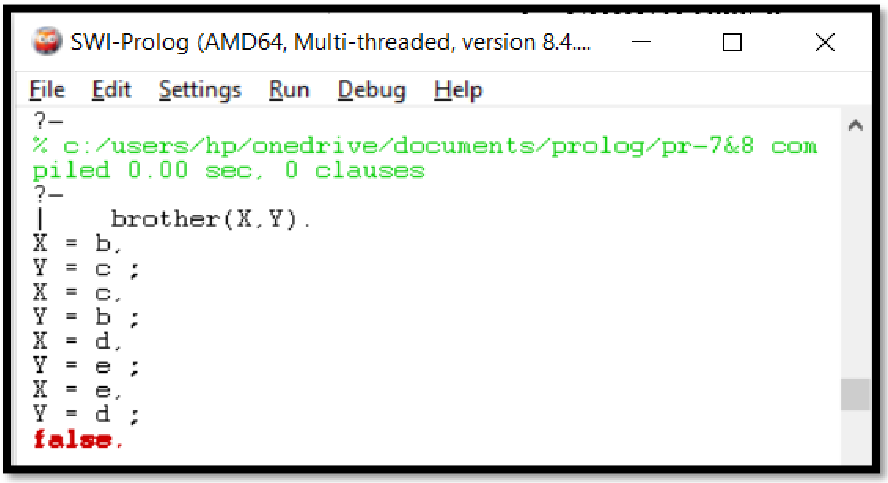
Practical – 7 & 8

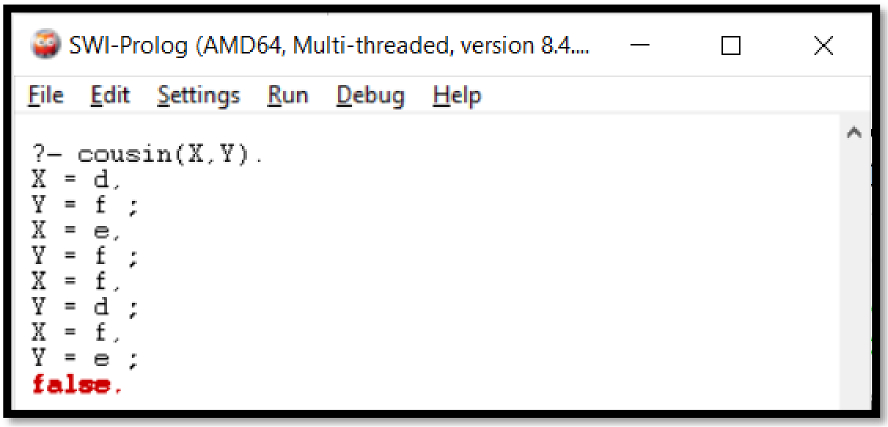
AIM: Assume given a set of facts of the form father (name1, name2) (name1 is the father of name2).
(I)Define a predicate brother (X, Y) which holds iff X and Y are brothers.
(II)Define a predicate cousin (X, Y) which holds iff X and Y are cousins.
(III)Define a predicate grandson (X, Y) which holds iff X is a grandson of Y.
(IV)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).

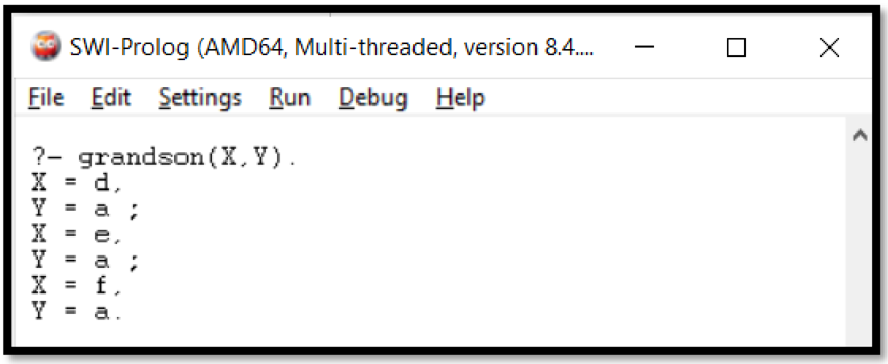
- **CODE (Prolog):**

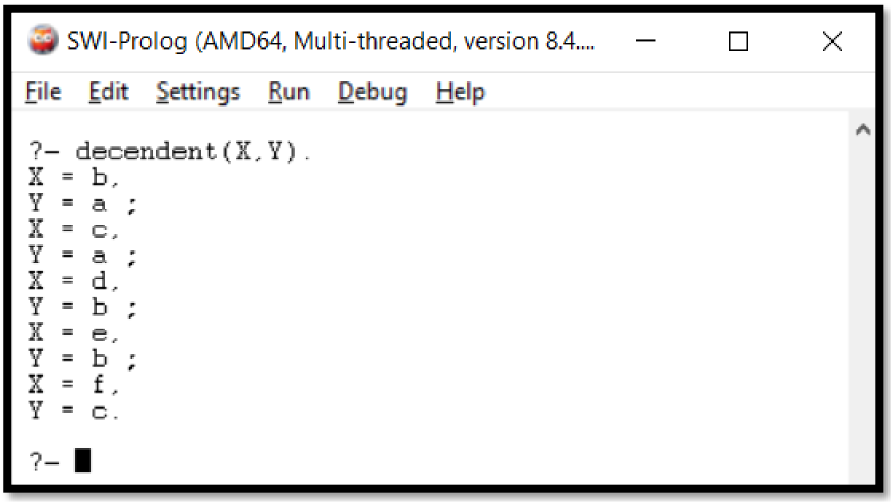
```
father(a,b).
father(a,c).
father(b,d).
father(b,e).
father(c,f).
brother(X,Y):- father(Z,X),father(Z,Y),not(X=Y).
cousin(X,Y):- father(Z,X),father(M,Y),brother(Z,M).
grandson(X,Y):- father(M,X),father(Y,M).
decendent(X,Y):- father(Y,X).
```

- **OUTPUT:**


SWI-Prolog (AMD64, Multi-threaded, version 8.4....
File Edit Settings Run Debug Help
?-
% c:/users/hp/onedrive/documents/prolog/pr-7&8 compiled 0.00 sec, 0 clauses
?-
| brother(X,Y).
X = b,
Y = c ;
X = c,
Y = b ;
X = d,
Y = e ;
X = e,
Y = d ;
false.


SWI-Prolog (AMD64, Multi-threaded, version 8.4....
File Edit Settings Run Debug Help
?- cousin(X,Y).
X = d,
Y = f ;
X = e,
Y = f ;
X = f,
Y = d ;
X = f,
Y = e ;
false.


SWI-Prolog (AMD64, Multi-threaded, version 8.4....
File Edit Settings Run Debug Help
?- grandson(X,Y).
X = d,
Y = a ;
X = e,
Y = a ;
X = f,
Y = a.


SWI-Prolog (AMD64, Multi-threaded, version 8.4....
File Edit Settings Run Debug Help
?- decendent(X,Y).
X = b,
Y = a ;
X = c,
Y = a ;
X = d,
Y = b ;
X = e,
Y = b ;
X = f,
Y = c.
?-

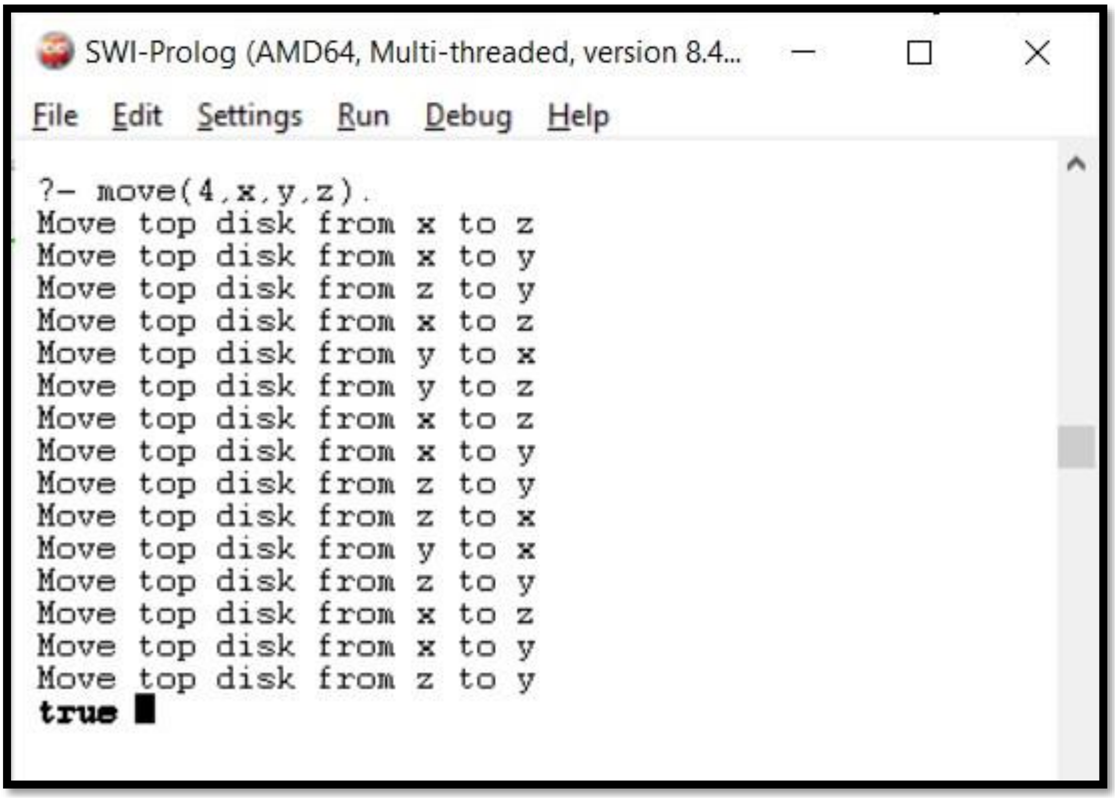
Practical – 9

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

Code (Prolog):

```
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,),
  move(1,X,Y,_),
  move(M,Z,Y,X).
```

- **OUTPUT:**



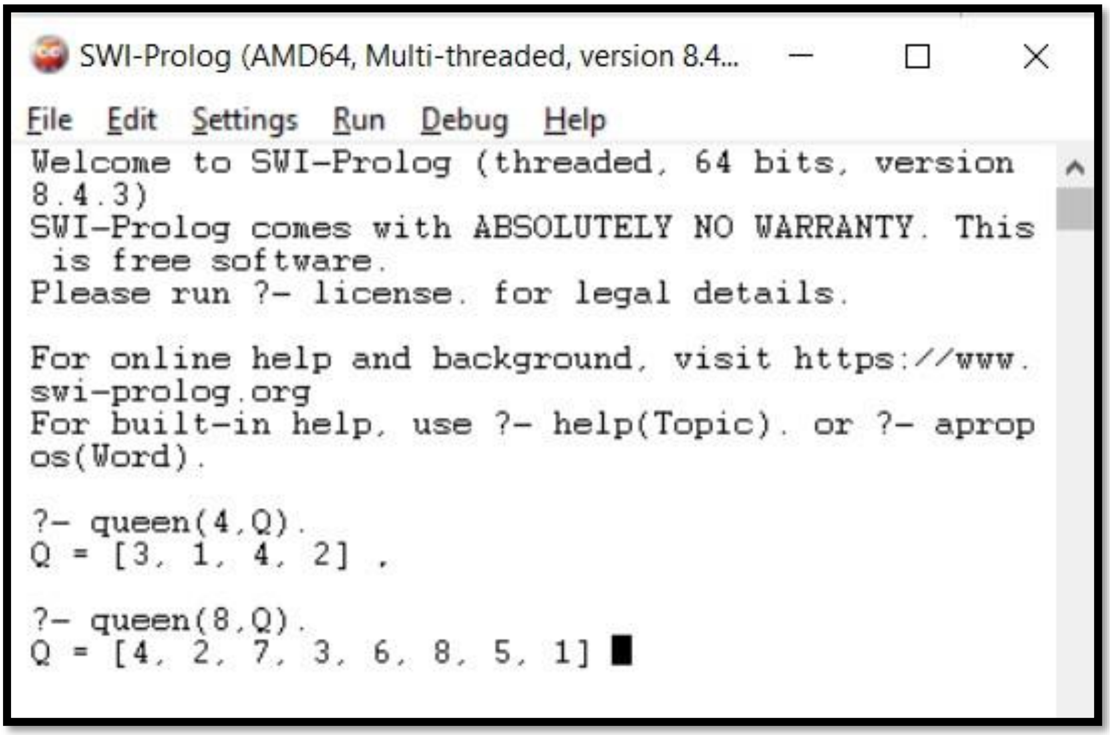
Practical – 10

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

Code (Prolog):

```
queen(N, Qs):- range(1,N,Us),queens(Us,[],Qs).
queens([],Qs,Qs).
queens(Us,Ps,Qs):- select(Q,Us,Us1),\+ attack(Q,Ps),
queens(Us1,[Q|Ps],Qs).
range(J,J,[J]).
range(I,J,[I|Ns]):- I<J,I1 is I + 1, range(I1,J,Ns).
attack(Q,Qs) :- attack(Q,1,Qs).
attack(X,N,[Y|_]) :- X is Y + N.
attack(X,N,[Y|_]) :- X is Y - N.
attack(X,N,[_|Ys]):-
N1 is N + 1 ,attack(X,N1,Ys),
    N1 is
    N+1,attack(X,N1,Ys).
go:- queens(8,Qs),write(Qs).
```

➢ **OUTPUT:**



Practical – 11

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

Code (Prolog):

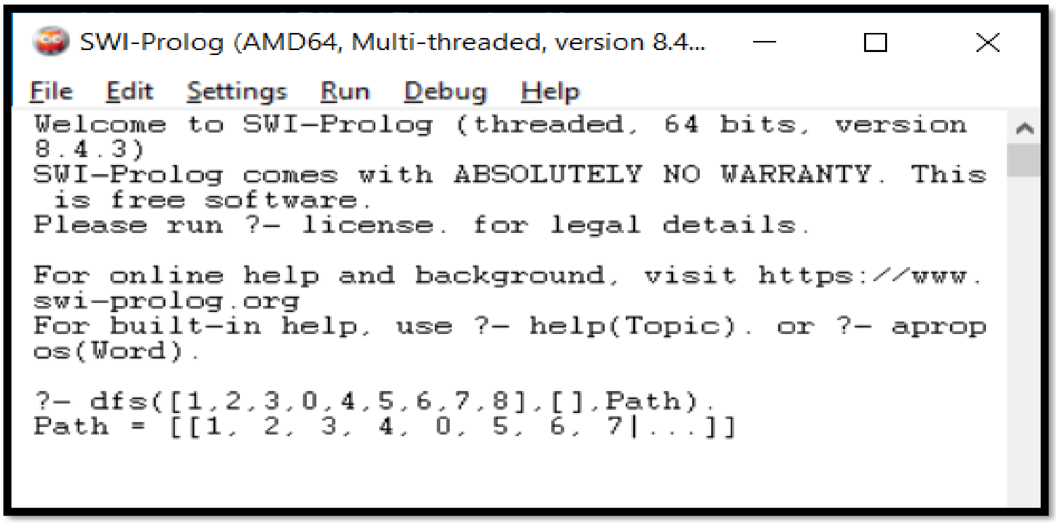
```
goal([1,2,3,4,0,5,6,7,8]).
%% move left in the top row
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
  [0,X1,X3, X4,X5,X6, X7,X8,X9]).
move([X1,X2,0, X4,X5,X6, X7,X8,X9],
  [X1,0,X2, X4,X5,X6, X7,X8,X9]).
%% move left in the middle row
move([X1,X2,X3, X4,0,X6,X7,X8,X9],
  [X1,X2,X3, 0,X4,X6,X7,X8,X9]).
move([X1,X2,X3, X4,X5,0,X7,X8,X9],
  [X1,X2,X3, X4,0,X5,X7,X8,X9]).
%% move left in the bottom row
move([X1,X2,X3, X4,X5,X6, X7,0,X9],
  [X1,X2,X3, X4,X5,X6, 0,X7,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
  [X1,X2,X3, X4,X5,X6, X7,0,X8]).
%% move right in the top row move([0,X2,X3,
X4,X5,X6, X7,X8,X9],
  [X2,0,X3, X4,X5,X6, X7,X8,X9]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
  [X1,X3,0, X4,X5,X6, X7,X8,X9]).
%% move right in the middle row
move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
  [X1,X2,X3, X5,0,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
  [X1,X2,X3, X4,X6,0, X7,X8,X9]).
  %% move right in the bottom row
move([X1,X2,X3, X4,X5,X6,0,X8,X9],
  [X1,X2,X3, X4,X5,X6,X8,0,X9]).
```

```
move([X1,X2,X3, X4,X5,X6,X7,0,X9],
  [X1,X2,X3, X4,X5,X6,X7,X9,0]).
   %% move up from the middle row
 move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
  [0,X2,X3, X1,X5,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
  [X1,0,X3, X4,X2,X6, X7,X8,X9]).
move([X1,X2,X3, X4,X5,0, X7,X8,X9],
  [X1,X2,0, X4,X5,X3, X7,X8,X9]).
   %% move up from the bottom row
  move([X1,X2,X3, X4,X5,X6, X7,0,X9],
  [X1,X2,X3, X4,0,X6, X7,X5,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
  [X1,X2,X3, X4,X5,0, X7,X8,X6]).
move([X1,X2,X3, X4,X5,X6, 0,X8,X9],
  [X1,X2,X3, 0,X5,X6, X4,X8,X9]).
 %% move down from the top row
 %% move left in the top row
 move([X1,0,X3, X4,X5,X6, X7,X8,X9],
  [0,X1,X3, X4,X5,X6, X7,X8,X9]).
move([X1,X2,0, X4,X5,X6, X7,X8,X9],
  [X1,0,X2, X4,X5,X6, X7,X8,X9]).
 %% move left in the middle row
 move([X1,X2,X3, X4,0,X6,X7,X8,X9],
  [X1,X2,X3, 0,X4,X6,X7,X8,X9]).
move([X1,X2,X3, X4,X5,0,X7,X8,X9],
  [X1,X2,X3, X4,0,X5,X7,X8,X9]).
 %% move left in the bottom row
 move([X1,X2,X3, X4,X5,X6, X7,0,X9],
  [X1,X2,X3, X4,X5,X6, 0,X7,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
  [X1,X2,X3, X4,X5,X6, X7,0,X8]).
 %% move right in the top row move([0,X2,X3, X4,X5,X6, X7,X8,X9],
```

```
    [X2,0,X3, X4,X5,X6, X7,X8,X9]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
    [X1,X3,0, X4,X5,X6, X7,X8,X9]).
 %% move right in the middle row
 move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
    [X1,X2,X3, X5,0,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
    [X1,X2,X3, X4,X6,0, X7,X8,X9]).
     %% move right in the bottom row
 move([X1,X2,X3, X4,X5,X6,0,X8,X9],
    [X1,X2,X3, X4,X5,X6,X8,0,X9]).
move([X1,X2,X3, X4,X5,X6,X7,0,X9],
    [X1,X2,X3, X4,X5,X6,X7,X9,0]).
     %% move up from the middle row
  move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
    [0,X2,X3, X1,X5,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
    [X1,0,X3, X4,X2,X6, X7,X8,X9]).
move([X1,X2,X3, X4,X5,0, X7,X8,X9],
    [X1,X2,0, X4,X5,X3, X7,X8,X9]).
     %% move up from the bottom row
  move([X1,X2,X3, X4,X5,X6, X7,0,X9],
    [X1,X2,X3, X4,0,X6, X7,X5,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
    [X1,X2,X3, X4,X5,0, X7,X8,X6]).
move([X1,X2,X3, X4,X5,X6, 0,X8,X9],
    [X1,X2,X3, 0,X5,X6, X4,X8,X9]).
 %% move down from the top row
 move([0,X2,X3, X4,X5,X6, X7,X8,X9],
    [X4,X2,X3, 0,X5,X6, X7,X8,X9]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
    [X1,X5,X3, X4,0,X6, X7,X8,X9]).
move([X1,X2,0, X4,X5,X6, X7,X8,X9],
```

```
    [X1,X2,X6, X4,X5,0, X7,X8,X9]).
  %% move down from the middle row
  move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
    [X1,X2,X3, X7,X5,X6, 0,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
    [X1,X2,X3, X4,X8,X6, X7,0,X9]).
move([X1,X2,X3, X4,X5,0, X7,X8,X9],
    [X1,X2,X3, X4,X5,X9, X7,X8,0]).
  dfsSimplest(S, [S]) :- goal(S).
  dfsSimplest(S, [S|Rest]) :- move(S, S2), dfsSimplest(S2, Rest).
  dfs(S, Path, Path) :- goal(S).
  dfs(S, Checked, Path) :-
     % try a move
     move(S, S2),
     % ensure the resulting state is new
     \+member(S2, Checked),
     % and that this state leads to the goal
     dfs(S2, [S2|Checked], Path).
```

- **OUTPUT:**



Practical – 12

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

Code (Prolog):

```
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:**