IT-306

ARTIFICIAL INTELLIGENCE

LAB FILE

DELHI TECHNOLOGICAL UNIVERSITY

Submitted to:
Ms. Shivani

Submitted by:
Varun Kumar
2K19/IT/140

INDEX

S.no	TOPICS
1	Implement Tic-Tac-Toe problem using Min-max algorithm.
2	Implement Tic-Tac-Toe problem using random number.
3	Write a program to implement Breadth first search for water jug problem.
4	Write a program to implement Depth first search for water jug problem.
5	Write a program to implement A* algorithm for 8-Puzzle problem.
6	Write a program to implement BFS algorithm for 8-Puzzle problem.
7	Write a program to solve crypt arithmetical problems.
8	Compare different searching algorithms.
9	Write a Prolog program that: (a) Computes factorial of a number (b) Computes area and circumference of a circle

PROGRAM - 1

AIM: Implement Tic-Tac-Toe problem using Min-max algorithm.

THEORY:
Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

ALGORITHM:
Assuming X is the "turn taking player":

- If the game is over, return the winner.
- Otherwise get a list of new game states for every possible move.
- Create a scores list.
- For each of these states add the minimax result of that state to the scores list.
- If it's X's turn, choose the maximum score from the scores list and make move.
- If it's O's turn, choose the minimum score from the scores list and make move.

CODE:

```
import random

class TicTacToe(object):
    winning_combos = (
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    )

    winners = ('X-win', 'Draw', 'O-win')

    def __init__(self, board=[]):
        """
        Initialize the tic tac toe board
        :param board: 1-D list of board positions
        """
        if len(board) == 0:
            self.board = [0 for i in range(9)]
        else:
            self.board = board

    def print_board(self):
        for i in range(3):
```

```
        print(
            "| " + str(self.board[i * 3]) +
            " | " + str(self.board[i * 3 + 1]) +
            " | " + str(self.board[i * 3 + 2]) + " |"
        )

def check_game_over(self):
    #Check if the game is over or there is a winner
    if 0 not in [element for element in self.board]:
        return True
    if self.winner() != 0:
        return True
    return False

def available_moves(self):
    #To check what all possible moves are remaining for a player
    return [index for index, element in enumerate(self.board) if element == 0]

def available_combos(self, player):
    return self.available_moves() + self.get_acquired_places(player)

def X_won(self):
    return self.winner() == 'X'

def O_won(self):
    return self.winner() == 'O'

def is_tie(self):
    return self.winner() == 0 and self.check_game_over()

#check winner of game
def winner(self):
    for player in ('X', 'O'):
        positions = self.get_acquired_places(player)
        for combo in self.winning_combos:
            win = True
            for pos in combo:
                if pos not in positions:
                    win = False
            if win:
                return player
    return 0

#To get the positions already acquired by a particular player
def get_acquired_places(self, player):
      return [index for index, element in enumerate(self.board) if element == player]
```

```
    def make_move(self, position, player):
        self.board[position] = player

    #minimax algorithm main function
    def minimax(self, node, player):
        if node.check_game_over():
            if node.X_won():
                return -1
            elif node.is_tie():
                return 0
            elif node.O_won():
                return 1
        best = 0
        for move in node.available_moves():
            node.make_move(move, player)
            val = self.minimax(node, get_enemy(player))
            node.make_move(move, 0)
            if player == 'O':
                if val > best:
                    best = val
            else:
                if val < best:
                    best = val
        return best

#Driver function to apply minimax algorithm
def determine(board, player):
    a = 0
    choices = []
    if len(board.available_moves()) == 9:
        return 4
    for move in board.available_moves():
        board.make_move(move, player)
        val = board.minimax(board, get_enemy(player))
        board.make_move(move, 0)
        if val > a:
            a = val
            choices = [move]
        elif val == a:
            choices.append(move)
    try:
        return random.choice(choices)
    except IndexError:
        return random.choice(board.available_moves())
```

```
def get_enemy(player):
    if player == 'X':
        return 'O'
    return 'X'

#main function
if __name__ == "__main__":
    board = TicTacToe()
    print('Board positions are like this: ')
    for i in range(3):
        print(
            "| " + str(i * 3 + 1) +
            " | " + str(i * 3 + 2) +
            " | " + str(i * 3 + 3) + " |"
        )
    print('Type in the position number you to make a move on..')
    while not board.check_game_over():
        player = 'X'
        player_move = int(input("Your Move: ")) - 1
        if player_move not in board.available_moves():
            print('Please check the input!')
            continue
        board.make_move(player_move, player)
        board.print_board()
        print()
        if board.check_game_over():
            break
        print('Computer is playing.. ')
        player = get_enemy(player)
        computer_move = determine(board, player)
        board.make_move(computer_move, player)
        board.print_board()
    if board.winner() != 0:
        if board.winner() == 'X':
            print ("Congratulations you win!")
        else:
            print('Computer Wins!')
    else:
        print("Game tied!")
```

OUTPUT:

```
Board positions are like this:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
Type in the position number you to make a move on..
Your Move: 7
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| X | 0 | 0 |

Computer is playing..
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| X | O | 0 |
Your Move: 4
| 0 | 0 | 0 |
| X | 0 | 0 |
| X | O | 0 |
```

```
Computer is playing..
| O | 0 | 0 |
| X | 0 | 0 |
| X | O | 0 |
Your Move: 5
| O | 0 | 0 |
| X | X | 0 |
| X | O | 0 |

Computer is playing..
| O | O | 0 |
| X | X | 0 |
| X | O | 0 |
Your Move: 3
| O | O | X |
| X | X | 0 |
| X | O | 0 |

Congratulations you win!
```

PROGRAM – 2

AIM: Implement tic-tac-toe problem using random number.

THEORY:

Tic-tac-toe (American English), Xs and Os is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players.

ALGORITHM:

play_game() is the main function, which:

- Calls create_board() to create a 9×9 board and initializes with 0.
- For each player (1 or 2), calls the random_place() function to randomly choose a location on board and mark that location with the players variable, alternatively.
- Print the board after each move.
- Evaluate the board after each move to check whether a row or column or a diagonal has the same player number. If so, displays the winner name. If after 9 moves, there are no winner then displays -1.

CODE:

```
# for display the tic tac toe board
def print_board(a):
   print("",a[1]," │",a[2]," │ ",a[3]," ")
   print("───────┼───────┼───────")
   print("",a[4]," │",a[5]," │ ",a[6]," ")
   print("───────┼───────┼───────")
   print("",a[7]," │",a[8]," │ ",a[9]," ")

def print_instructions():
   print("\n----------- WELCOME TO TIC TAC TOE ------------\n\n")
   print_board(pos)
   print()

   players[0] = input("Player 1 : ")
   players[1] = input("Player 2 : ")

   print("\n-------- Instructions ---------")
   print("->",players[0],"you will using X")
   print("->",players[1],"you will using 0")
   print("-> Turn starts from",players[0])
   print("-> Potisions are like :-")
   print("  1 │  2 │ 3  ")
```

```
    print("────┼────┼────")
    print("  4 │ 5  │ 6  ")
    print("────┼────┼────")
    print("  7 │ 8  │ 9  ")
    print("-> press S to start the game")
    flag = input()
    return flag

def startgame():
    turn = 0
    for i in range(9):
        if turn % 2 == 0:
            print("\nthis is ur turn",players[0])
            p = int(input("Please Enter postion : "))
            v = 'x'
            pos[p] = v
            print_board(pos)
            winner = checkwin(v)
            if winner == "nobody":
                turn = 1
                continue
            else:
                print("\n\nHurray !!,",players[0],"you win ♥♥")
                break
        else:
            print("\nthis is ur turn",players[1])
            p = int(input("Please Enter postion : "))
            v = '0'
            pos[p] = v
            print_board(pos)
            winner = checkwin(v)
            if winner == "nobody":
                turn = 0
                continue
            else:
                print("\n\nHurray !!,",players[1],"you win ♥♥")
                break
    else:
        print("\n\nGame is Tie")

# check for winner
def checkwin(v):
    for i in winning_conditions:
        if (pos[i[0]], pos[i[1]], pos[i[2]]) == (v,v,v):
            winner = players[0]
            break
```

```
        elif (pos[i[0]], pos[i[1]], pos[i[2]]) == (v,v,v):
            winner = players[1]
            break
    else:
        winner = "nobody"
    return winner

# main code
pos = ['',' ',' ',' ',' ',' ',' ',' ',' ',' ']
players = ['','']
winning_conditions = [(1,2,3),(4,5,6),(7,8,9),(1,4,7),(2,5,8),(3,6,9),(1,5,9),(3,5,7)]
flag = print_instructions()
if flag == 's' or flag == 'S':
    startgame()
else:
    print("Invalid Entry")
```

OUTPUT:

```
----------- WELCOME TO TIC TAC TOE ------------

     |     |
_____|_____|_____
     |     |
_____|_____|_____
     |     |

Player 1 : Tanmay
Player 2 : Sushant

-------- Instructions ---------
-> Tanmay you will using X
-> Sushant you will using 0
-> Turn starts from Tanmay
-> Potisions are like :-
  1  |  2  | 3
_____|_____|_____
  4  | 5   | 6
_____|_____|_____
  7  | 8   | 9
-> press S to start the game
S
```

```
this is ur turn Tanmay
Please Enter postion : 1
 x  |    |
____|____|____
    |    |
____|____|____
    |    |

this is ur turn Sushant
Please Enter postion : 2
 x  | 0  |
____|____|____
    |    |
____|____|____
    |    |

this is ur turn Tanmay
Please Enter postion : 5
 x  | 0  |
____|____|____
    | x  |
____|____|____
    |    |

this is ur turn Sushant
Please Enter postion : 4
 x  | 0  |
____|____|____
 0  | x  |
____|____|____
    |    |

this is ur turn Tanmay
Please Enter postion : 9
 x  | 0  |
____|____|____
 0  | x  |
____|____|____
    |    | x

Hurray !!, Tanmay you win ♥♥
```

PROGRAM - 3

AIM: Write a program to implement Breadth first search for water jug problem.

THEORY:

Water Jug Problem - Given an m litter jug and a n litter jug. Both the jugs are initially empty. The jugs
don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litters of water where d is less than n.

The operations that can be perform are:

1. Empty a Jug, (X, Y) -> (0, Y) Empty Jug 1
2. Fill a Jug, (0, 0) -> (X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node).

ALGORITHM:

- Define a state space that contains all possible configurations of the water jugs and even some of the unreachable ones.
- Specify one or multiple states within that space that describe all the possible situations from which we can initiate the problem-solving process. These states are referred to as the initial states.
- Specify one or more states which are regarded as the acceptable solutions to the problem. These states are known as goal states.
- Specify a set of rules also known as predictions that describe the actions (operators) allowed and a well-defined control strategy for aligning the order of application of these predictions.

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 = int(input("Enter Jug1 value : "))
        Jug2 = int(input("Enter Jug2 value : "))
        target = int(input("Enter target value : "))
        print("Path from initial state "
            "to solution state ::")
        BFS(Jug1, Jug2, target)
```

OUTPUT

```
Enter Jug1 value : 5
Enter Jug2 value : 4
Enter target value : 3
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 4 )
( 5 , 0 )
( 5 , 4 )
( 4 , 0 )
( 1 , 4 )
( 4 , 4 )
( 5 , 3 )
( 0 , 3 )
```

PROGRAM - 4

AIM: Write a program to implement Depth first search for water jug problem.

THEORY:

Water Jug Problem - Given an m litter jug and a n litter jug. Both the jugs are initially empty. The jugs
don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litters of water where d is less than n.
The operations that can be perform are:

1. Empty a Jug, (X, Y) -> (0, Y) Empty Jug 1
2. Fill a Jug, (0, 0) -> (X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it.

ALGORITHM:

- Select a node. Since we have selected the node, add it to the visited list.
- Look at all the adjacent nodes. Add those nodes which have not been visited to the stack.
- Then pop the top node and repeat the first two steps by backtracking.

CODE:

```
visited = []
ans = []
jug1 = int(input("Enter capacity of Jug1 : "))
jug2 = int(input("Enter capacity of Jug2 : "))
target = int(input("Enter target value : "))
def state(a , b):
    if( (a,b) in visited):
        return False

    if((a,b) == (target , 0)):
        visited.append((a,b))
        ans.append((a,b))
        return True

    visited.append((a,b))

    #Fill jug1 and jug2
    if state(jug1 , b):
        ans.append((a,b))
        return True
```

```
    if state(a , jug2):
        ans.append((a,b))
        return True

    #empty Jug1 and jug2
    if state(0 , b):
        ans.append((a,b))
        return True
    if state(a , 0):
        ans.append((a,b))
        return True

    #transfer
    t = min(b , jug1 - a)
    if state(a + t , b - t):
        ans.append((a,b))
        return True

    t = min(a , jug2 - b)
    if state(a - t , b + t):
        ans.append((a,b))
        return True

    return False

state(0,0)
if len(ans) == 0:
    print("NO sol")
else:
    ans.reverse()
    print(ans)
```

OUTPUT:

```
Enter capacity of Jug1 : 4
Enter capacity of Jug2 : 3
Enter target value : 2
[(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
```

PROGRAM - 5

AIM: Write a program to implement A* algorithm for 8-Puzzle problem.

THEORY:

A* is a computer algorithm that is widely used in path finding and graph traversal. The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes.

ALGORITHM:

- We first move the empty space in all the possible directions in the start state and calculate the f-score for each state.
- After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list.
- A state with the least f-score is selected and expanded again.
- This process continues until the goal state occurs as the current state.

CODE:

```
from copy import deepcopy
import numpy as np
import time

# takes the input of current states and evaluvates the best path to goal state
def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)

# this function checks for the uniqueness of the iteration(it) state, weather it has been previously
traversed or not.
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state
```

```
def manhattan(puzzle, goal):
    a = abs(puzzle // 3 - goal // 3)
    b = abs(puzzle % 3 - goal % 3)
    mhcost = a + b
    return sum(mhcost[1:])

# will calcuates the number of misplaced tiles in the current state as compared to the goal state
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0

#3[on_true] if [expression] else [on_false]
# will indentify the coordinates of each of goal or initial state values
def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos

# start of 8 puzzle evaluvation, using Manhattan heuristics
def evaluvate(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],
1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

     # initializing the parent, gn and hn, where hn is manhattan distance function call
    costg = coordinates(goal)
    parent = -1
    gn = 0
    hn = manhattan(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]
    priority = np.array( [(0, hn)], dtpriority)

    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        priority = np.delete(priority, 0, 0)
        # sort priority queue using merge sort,the first element is picked for exploring remove from
queue what we are exploring
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
```

```
        # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
                # The all function is called, if the node has been previously explored or not
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable ! \n")
                        exit
                    # calls the manhattan function to calcuate the cost
                    hn = manhattan(coordinates(openstates), costg)
                     # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal
state
                    fn = gn + hn
                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                      # Checking if the node in openstates are matching the goal state.
                    if np.array_equal(openstates, goal):
                        print(' The 8 puzzle is solvable ! \n')
                        return state, len(priority)

    return state, len(priority)

# start of 8 puzzle evaluvation, using Misplaced tiles heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],
1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is misplaced_tiles  function call
    parent = -1
```

```
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   # We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)

    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        # sort priority queue using merge sort,the first element is picked for exploring.
        priority = np.delete(priority, 0, 0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
         # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        # Increase cost g(n) by 1
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                 # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
                # The check function is called, if the node has been previously explored or not.
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
                    # calls the Misplaced_tiles function to calcuate the cost
                    hn = misplaced_tiles(coordinates(openstates), costg)
                    # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal
state
                    fn = gn + hn
                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                    # Checking if the node in openstates are matching the goal state.
```

```
                if np.array_equal(openstates, goal):
                    print(' The 8 puzzle is solvable \n')
                    return state, len(priority)

    return state, len(priority)

 # User input for initial state
puzzle = []
print(" Input vals from 0-8 for start state ")
for i in range(0,9):
    x = int(input("enter vals :"))
    puzzle.append(x)

 # User input of goal state
goal = []
print(" Input vals from 0-8 for goal state ")
for i in range(0,9):
    x = int(input("Enter vals :"))
    goal.append(x)

n = int(input("1. Manhattan distance \n2. Misplaced tiles"))

if(n ==1 ):
    state, visited = evaluvate(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))
if(n == 2):
    state, visited = evaluvate_misplaced(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))
```

OUTPUT:

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :0
enter vals :3
enter vals :4
enter vals :5
enter vals :6
enter vals :7
enter vals :8
 Input vals from 0-8 for goal state
Enter vals :0
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
1. Manhattan distance
2. Misplaced tiles2
 The 8 puzzle is solvable

   1 2 0
   3 4 5
   6 7 8

   1 0 2
   3 4 5
   6 7 8

   0 1 2
   3 4 5
   6 7 8
Steps to reach goal: 2
Total nodes visited:  2

Total generated: 5
```

PROGRAM - 6

AIM: Write a program to implement bfs algorithm for 8-Puzzle problem.

THEORY:

N puzzle problem is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

Breadth-First Search (BFS) starts by examining the first node and expands one layer at a time, for example, all nodes "one hop" from the first node; once those are exhausted it proceeds to all nodes "two hops" from the first node and so forth.

ALGORITHM:

- Generate a graph map of all possible scenarios from the given initial state.
- Find the goal state (final state) in that map.
- Backtrack to the initial state to find the path.

CODE:

```
import 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("States 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:

```
Start State:
 [[1 2 3]
 [8 0 4]
 [7 6 5]]

Goal State:
 [[2 8 1]
 [0 4 3]
 [7 6 5]]

States Explored:  358

Solution:

action:  up
 [[1 0 3]
 [8 2 4]
 [7 6 5]]

action:  left
 [[0 1 3]
 [8 2 4]
 [7 6 5]]

action:  down
 [[8 1 3]
 [0 2 4]
 [7 6 5]]
```

```
action:  right
 [[8 1 3]
 [2 0 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 4 0]
 [7 6 5]]

action:  up
 [[8 1 0]
 [2 4 3]
 [7 6 5]]

action:  left
 [[8 0 1]
 [2 4 3]
 [7 6 5]]

action:  left
 [[0 8 1]
 [2 4 3]
 [7 6 5]]

action:  down
 [[2 8 1]
 [0 4 3]
 [7 6 5]]

Goal Reached!!
```

PROGRAM - 7

AIM: Write a program to solve crypt arithmetical problems.

THEORY:
Crypt arithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols. In crypt arithmetic problem, the digits (0-9) get substituted by some possible alphabets or symbols. The task in crypt arithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

ALGORITHM:

- First, create a list of all the characters that need assigning to pass to Solve
- If all characters are assigned, return true if puzzle is solved, false otherwise
- Otherwise, consider the first unassigned character
- for (every possible choice among the digits not in use)
- make that choice and then recursively try to assign the rest of the characters
  if recursion successful, return true
- if !successful, unmake assignment and try another digit
- If all digits have been tried and nothing worked, return false to trigger backtracking

CODE:

```
#include <bits/stdc++.h>
using namespace std;

vector<int> use(10);
// structure to store char and its corresponding integer
struct node
{
   char c;
   int v;
};

// function check for correct solution
int check(node* nodeArr, int count, string s1, string s2, string s3)
{
   int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;
   // calculate number corresponding to first string
   for (i = s1.length() - 1; i >= 0; i--)
   {
      char ch = s1[i];
      for (j = 0; j < count; j++)
         if (nodeArr[j].c == ch)
            break;
```

```
        val1 += m * nodeArr[j].v;
        m *= 10;
    }
    m = 1;

    // calculate number corresponding to second string
    for (i = s2.length() - 1; i >= 0; i--)
    {
        char ch = s2[i];
        for (j = 0; j < count; j++)
            if (nodeArr[j].c == ch)
                break;

        val2 += m * nodeArr[j].v;
        m *= 10;
    }
    m = 1;

    // calculate number corresponding to third string
    for (i = s3.length() - 1; i >= 0; i--)
    {
        char ch = s3[i];
        for (j = 0; j < count; j++)
            if (nodeArr[j].c == ch)
                break;

        val3 += m * nodeArr[j].v;
        m *= 10;
    }

    // sum of first two number equal to third return true
    if (val3 == (val1 + val2))
        return 1;

    return 0;
}

// Recursive function to check solution for all permutations
bool permutation(int count, node* nodeArr, int n, string s1, string s2, string s3)
{
    if (n == count - 1)
    {
        // check for all numbers not used yet
        for (int i = 0; i < 10; i++)
        {
            // if not used
```

```
            if (use[i] == 0)
            {
                nodeArr[n].v = i;
                // if solution found
                if (check(nodeArr, count, s1, s2, s3) == 1)
                {
                    cout << "\nSolution found: ";
                    for (int j = 0; j < count; j++)
                        cout << " " << nodeArr[j].c << " = "
                            << nodeArr[j].v;
                    return true;
                }
            }
        }
        return false;
    }

    for (int i = 0; i < 10; i++)
    {
        // if ith integer not used yet
        if (use[i] == 0)
        {
            nodeArr[n].v = i;
            use[i] = 1;
            // call recursive function
            if (permutation(count, nodeArr, n + 1, s1, s2, s3))
                return true;

            use[i] = 0;
        }
    }
    return false;
}

bool solveCryptographic(string s1, string s2, string s3)
{
    // count to store number of unique char
    int count = 0;

    // Length of all three strings
    int l1 = s1.length();
    int l2 = s2.length();
    int l3 = s3.length();

    // vector to store frequency of each char
    vector<int> freq(26);
```

```
    for (int i = 0; i < l1; i++)
        ++freq[s1[i] - 'A'];

    for (int i = 0; i < l2; i++)
        ++freq[s2[i] - 'A'];

    for (int i = 0; i < l3; i++)
        ++freq[s3[i] - 'A'];

    // count number of unique char
    for (int i = 0; i < 26; i++)
        if (freq[i] > 0)
            count++;

    // solution not possible for count greater than 10
    if (count > 10)
    {
        cout << "Invalid strings";
        return 0;
    }

    node nodeArr[count];

    // store all unique char in nodeArr
    for (int i = 0, j = 0; i < 26; i++)
    {
        if (freq[i] > 0)
        {
            nodeArr[j].c = char(i + 'A');
            j++;
        }
    }
    return permutation(count, nodeArr, 0, s1, s2, s3);
}

// Driver function
int main()
{
    string s1,s2,s3;
    cout<<"Enter first String: ";
    cin>>s1;
    cout<<"Enter second String: ";
    cin>>s2;
    cout<<"Enter added String: ";
    cin>>s3;
```

```
    if (solveCryptographic(s1, s2, s3) == false)
        cout << "\nNo solution";
    return 0;
}
```

OUTPUT:

```
Enter first String: SEND
Enter second String: MORE
Enter added String: MONEY
Solution found:  D = 1 E = 5 M = 0 N = 3 O = 8 R = 2 S = 7 Y = 6
```

PROGRAM - 8

Comparison of different searching algorithms

Algorithm	BFS (Breadth-first Search)	DFS (Depth-first Search)	UCS (Uniform-cost Search)	BFS(greedy search) (Best-first Search)	A* search
Time Complexity	$O(b^{d})$	$O(b^{m})$	$O(b^{1+[C*/e]})$	$O(b^{d})$	$O(b^{d})$
Space Complexity	$O(b^{d})$	O(bm)	$O(b^{1+[C*/e]})$	$O(b^{d})$	$O(b^{d})$
Optimality	Yes	No	Yes	Yes	No
Completeness	Yes	No	Yes	Yes	Yes

Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Depth-first Search:

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$$f(n)= g(n).$$

Were, $h(n)$= estimated cost from node n to the goal.

A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function $h(n)$, and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses $g(n)+h(n)$ instead of $g(n)$.

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number.

PROGRAM - 9

AIM: Write a Prolog program that:
(a) Computes factorial of a number.
(b) Computes area and circumference of a circle.

THEORY:

Factorial of a Number

The factorial of a number is the product of all the integers from 1 to that number. Prolog Factorial function definition is also similar to a normal factorial function.
Factorial of n is n!= 1 * 2* 3 * 4 *... * n
Factorial function consist of two clauses.
First clause is unit class with no body and the second clause is a rule as it has a body.
Body of the second clause is on to the right side of ':-' which is read as 'if' and it consists of literals separated by commas, each read as 'and'.

Area and Circumference of a Circle

Area of circle is the region occupied by the circle in a two dimensional plane. Area = πr^2.
The circumference is the length of the boundary of the circle. C = $2\pi r$
Write is used to write on output and read is used to take input nl is used for going to next line in output.

ALGORITHM:

Factorial of a Number

- The recursive relation between N and factorial M reviews rules for particular relation in the top to bottom order.
- Factorial(0,1) i.e., factorial of 0 is generally 1.
- Factorial(N,M), if any temporary value N1 is assigned to N-1.
- Factorial(N1,M1), and is factorial of N1 is M1.
- M is NM1 i.e., assigning M to N*M1, then value of N is M.

Area and Circumference of a Circle

- Take input radius to calculate area and circumference of circle.
- Apply value of R to calculate Area using Area = πr^2.
- And circumference of circle using C = $2\pi r$.

CODE:

Factorial of a Number

```
factorial(0, 1).
factorial(N, F) :- N > 0, Prev is N -1, factorial(Prev, R), F is R * N.
```

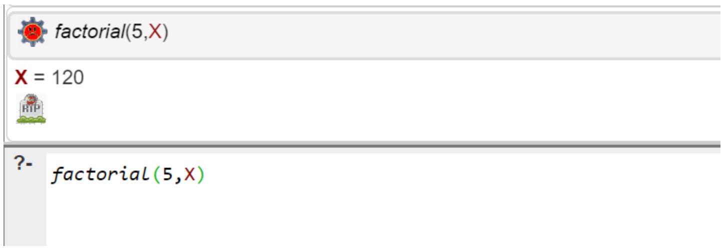
Area and Circumference of a Circle

```
circle:-
    write('Radius '),read(R),
```

```
write('Area is '),A is 3.14*R*R,write(A),nl,
write('Circumference is '),C is 2*3.14*R,write(C),nl.
```

<u>OUTPUT:</u>

<u>Factorial of a Number</u>



<u>Area and Circumference of a Circle</u>