Artificial Intelligence Lab Report

Submitted by

Valmika G (1BM20CS180)
Batch: D1

Course: Artificial Intelligence
Course Code: 20CS5PCAIP
Sem & Section: 5D

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

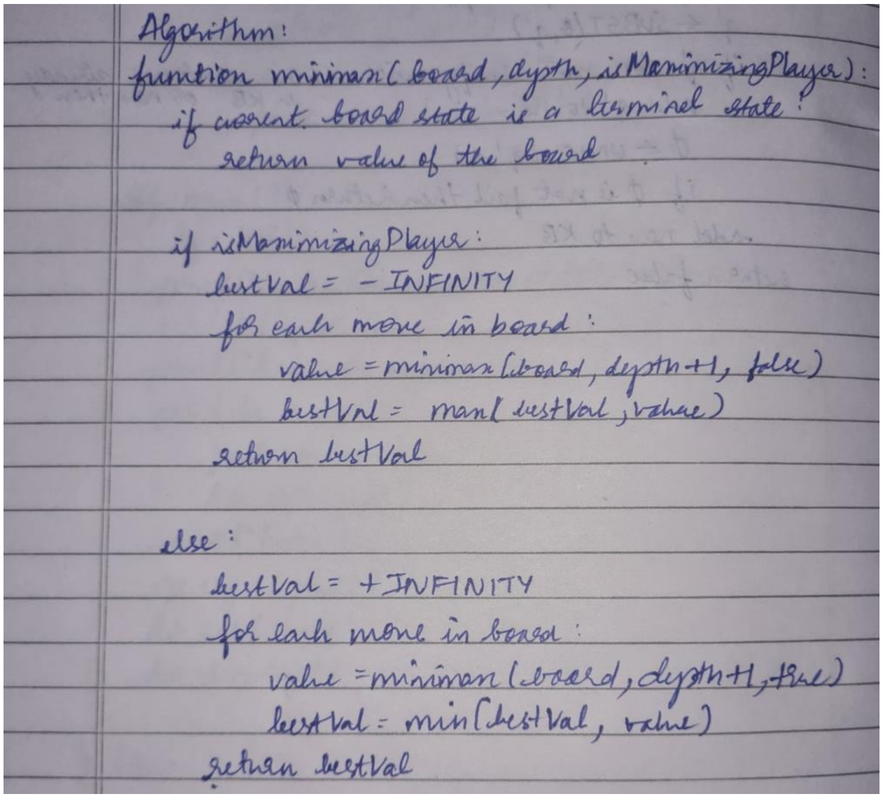
B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of contents

9/11/22 **Lab Program 1**

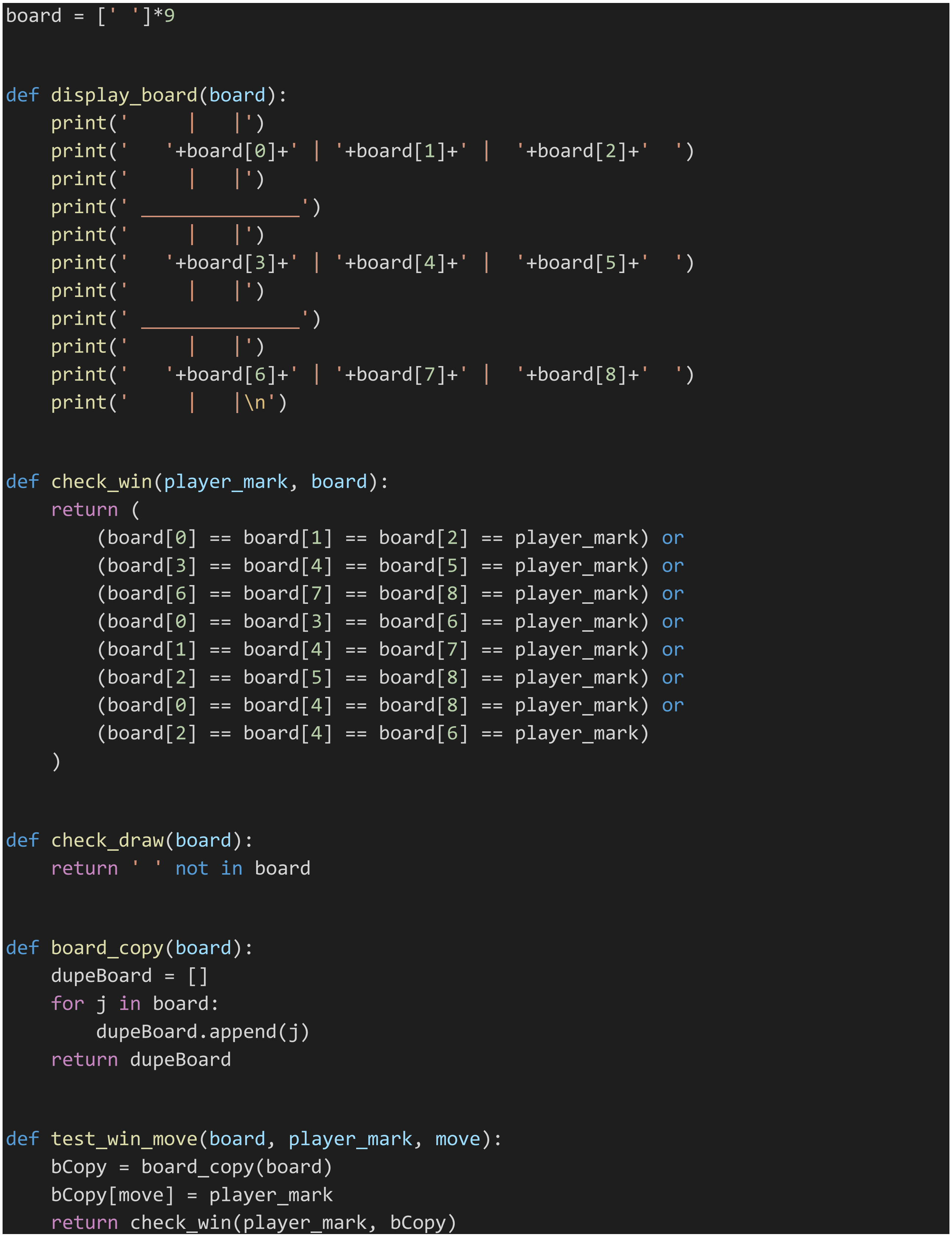
Implement Tic –Tac –Toe Game

Algorithm:

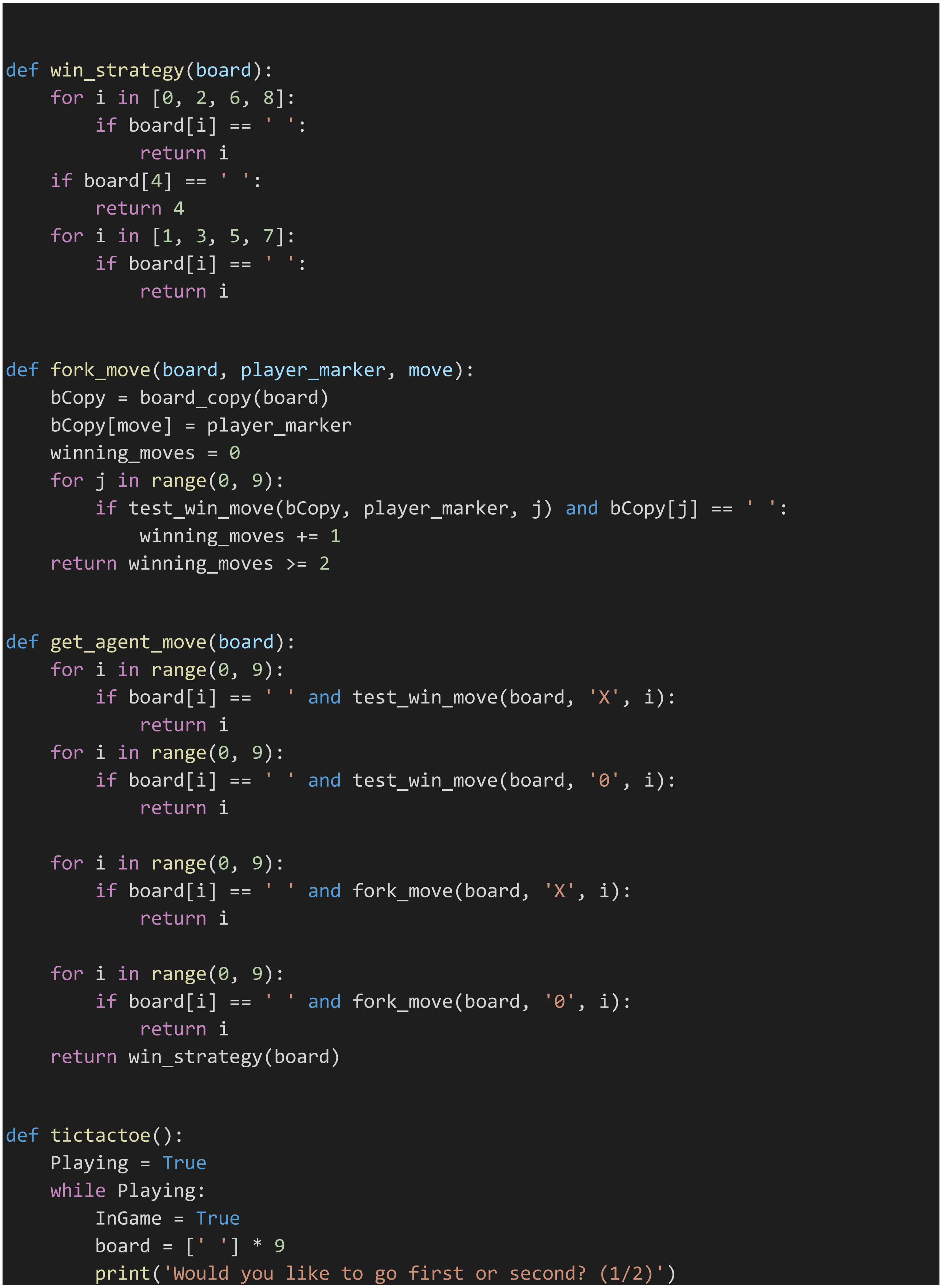


```
Algorithm:
funtion minimax( board, depth, isMaximizingPlayer):
  if current board state is a terminal state:
    return value of the board

  if isMaximizingPlayer:
    bestVal = -INFINITY
    for each move in board:
        value = minimax(board, depth+1, false)
        bestVal = max( bestVal, value)
    return bestVal

  else:
    bestVal = +INFINITY
    for each move in board:
        value = minimax(board, depth+1, true)
        bestVal = min(bestVal, value)
    return bestVal
```

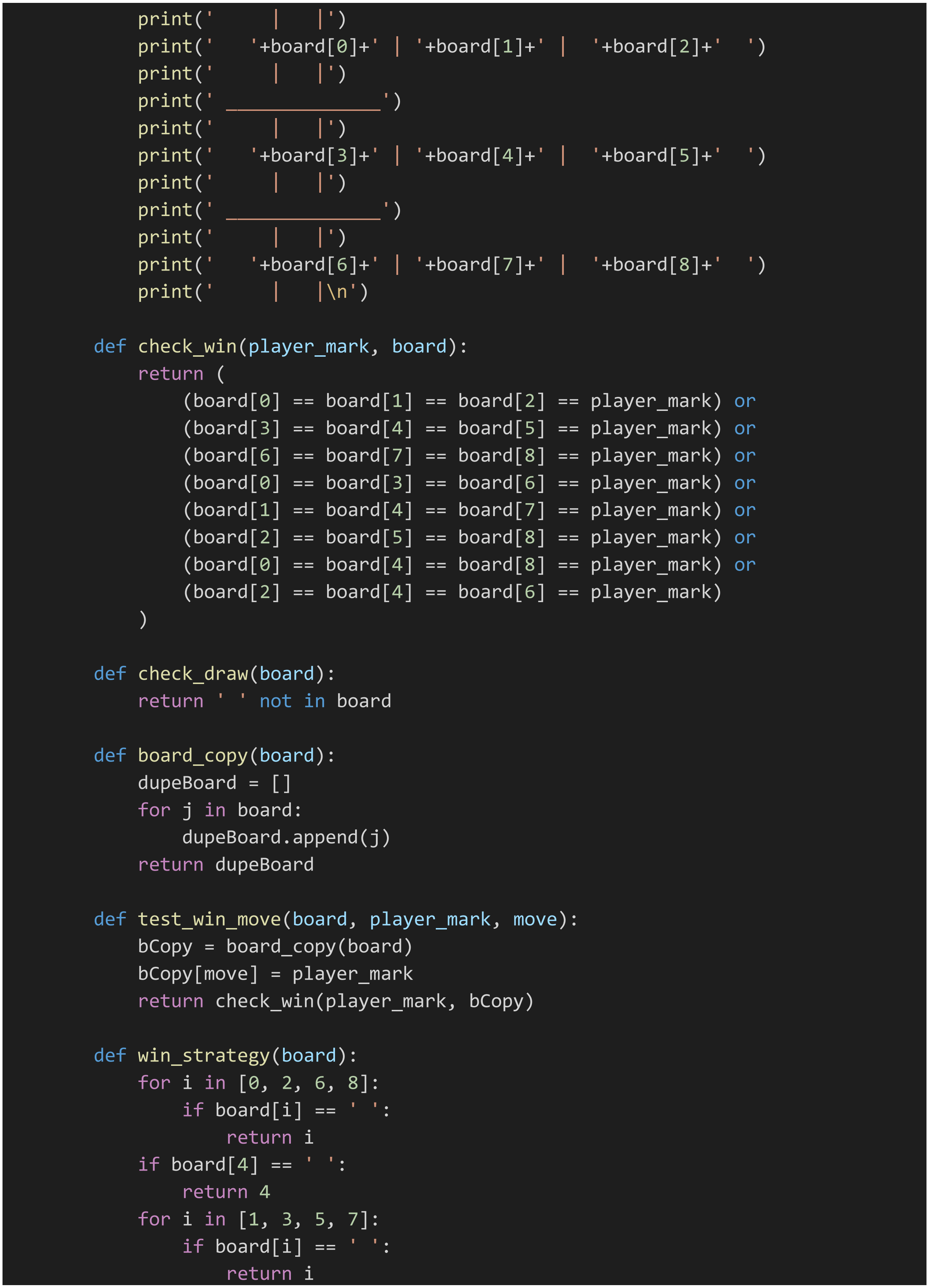
Code:



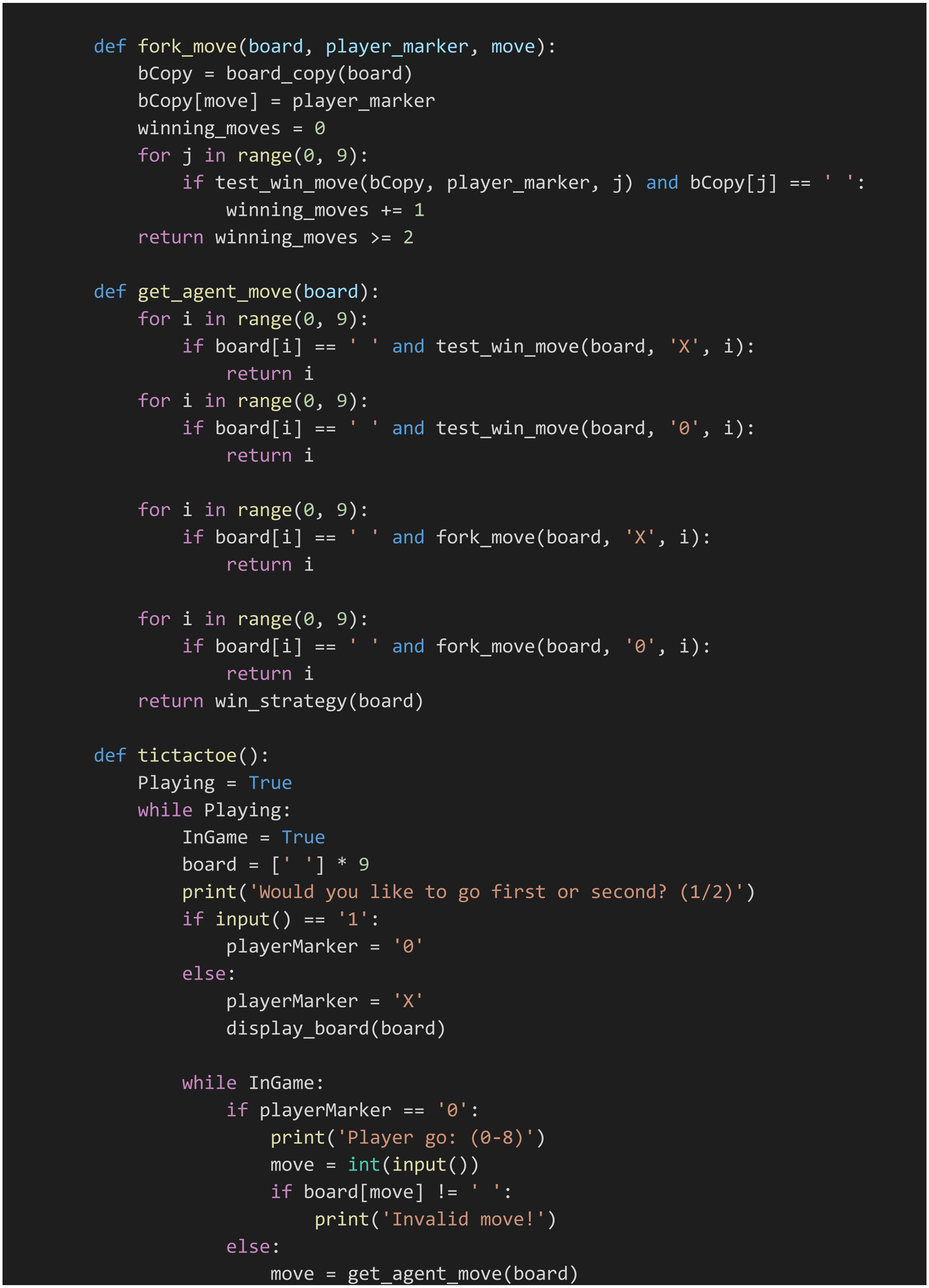
```
board = [' ']*9


def display_board(board):
    print('     |   |')
    print('   '+board[0]+' | '+board[1]+' |  '+board[2]+'  ')
    print('     |   |')
    print(' _____________')
    print('     |   |')
    print('   '+board[3]+' | '+board[4]+' |  '+board[5]+'  ')
    print('     |   |')
    print(' _____________')
    print('     |   |')
    print('   '+board[6]+' | '+board[7]+' |  '+board[8]+'  ')
    print('     |   |\n')


def check_win(player_mark, board):
    return (
        (board[0] == board[1] == board[2] == player_mark) or
        (board[3] == board[4] == board[5] == player_mark) or
        (board[6] == board[7] == board[8] == player_mark) or
        (board[0] == board[3] == board[6] == player_mark) or
        (board[1] == board[4] == board[7] == player_mark) or
        (board[2] == board[5] == board[8] == player_mark) or
        (board[0] == board[4] == board[8] == player_mark) or
        (board[2] == board[4] == board[6] == player_mark)
    )


def check_draw(board):
    return ' ' not in board


def board_copy(board):
    dupeBoard = []
    for j in board:
        dupeBoard.append(j)
    return dupeBoard


def test_win_move(board, player_mark, move):
    bCopy = board_copy(board)
    bCopy[move] = player_mark
    return check_win(player_mark, bCopy)
```



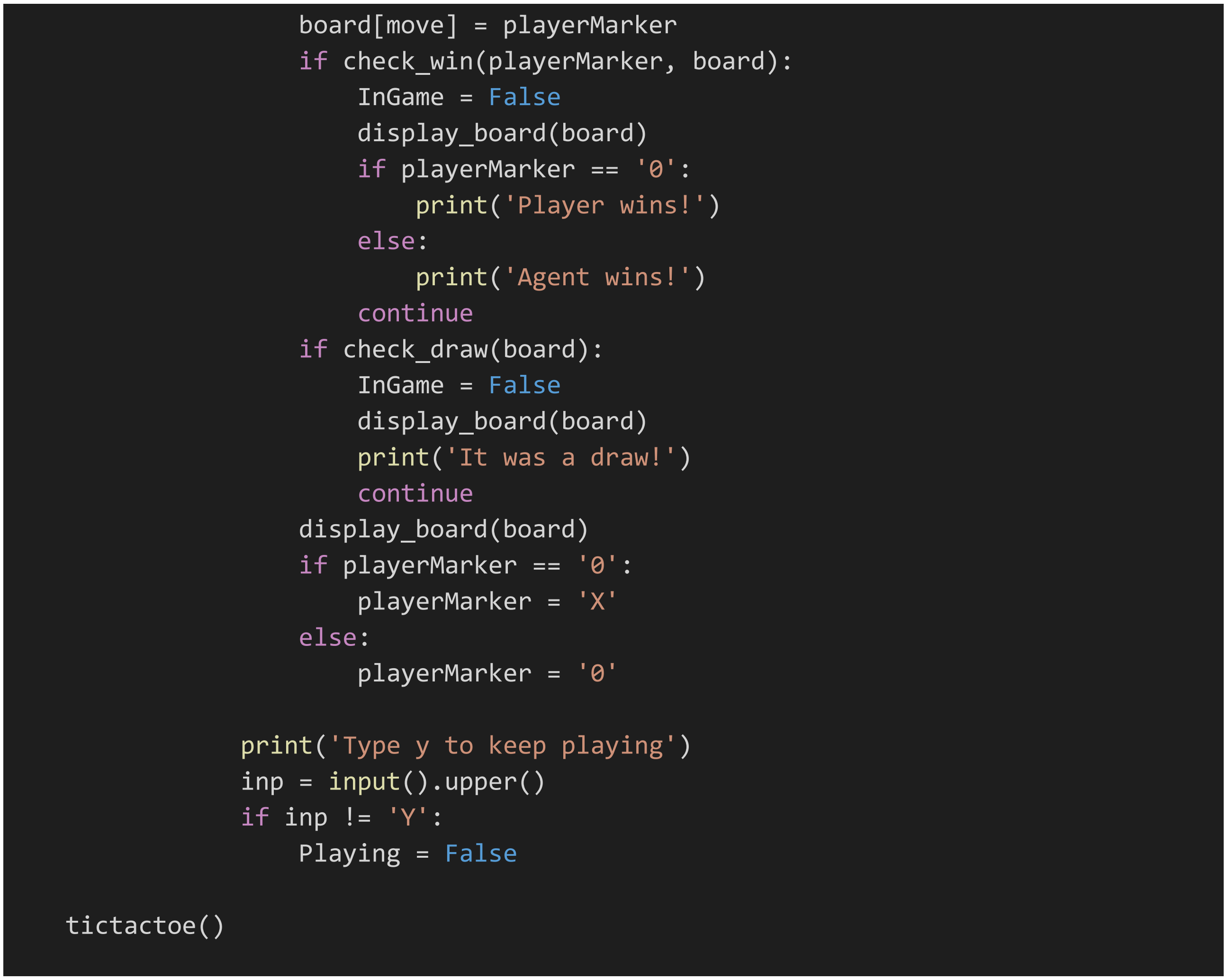
```python
def win_strategy(board):
    for i in [0, 2, 6, 8]:
        if board[i] == ' ':
            return i
    if board[4] == ' ':
        return 4
    for i in [1, 3, 5, 7]:
        if board[i] == ' ':
            return i


def fork_move(board, player_marker, move):
    bCopy = board_copy(board)
    bCopy[move] = player_marker
    winning_moves = 0
    for j in range(0, 9):
        if test_win_move(bCopy, player_marker, j) and bCopy[j] == ' ':
            winning_moves += 1
    return winning_moves >= 2


def get_agent_move(board):
    for i in range(0, 9):
        if board[i] == ' ' and test_win_move(board, 'X', i):
            return i
    for i in range(0, 9):
        if board[i] == ' ' and test_win_move(board, '0', i):
            return i

    for i in range(0, 9):
        if board[i] == ' ' and fork_move(board, 'X', i):
            return i

    for i in range(0, 9):
        if board[i] == ' ' and fork_move(board, '0', i):
            return i
    return win_strategy(board)


def tictactoe():
    Playing = True
    while Playing:
        InGame = True
        board = [' '] * 9
        print('Would you like to go first or second? (1/2)')
        if input() == '1':
            playerMarker = '0'
        else:
            playerMarker = 'X'
            display_board(board)

        while InGame:
            if playerMarker == '0':
                print('Player go: (0-8)')
                move = int(input())
                if board[move] != ' ':
                    print('Invalid move!')
            else:
                move = get_agent_move(board)
            board[move] = playerMarker
            if check_win(playerMarker, board):
                InGame = False
                display_board(board)
                if playerMarker == '0':
                    print('Player wins!')
                else:
                    print('Agent wins!')
                continue
            if check_draw(board):
                InGame = False
                display_board(board)
                print('It was a draw!')
                continue
            display_board(board)
            if playerMarker == '0':
                playerMarker = 'X'
            else:
                playerMarker = '0'

        print('Type y to keep playing')
        inp = input()
        if inp != 'y' and inp != 'Y':
            Playing = False


tictactoe()

class Tic_Tac_Toe:
    def __init__(self):
        board = [' ']*9

        def display_board(board):
```



```
    print('     |   |')
    print('   '+board[0]+' | '+board[1]+' |  '+board[2]+'  ')
    print('     |   |')
    print(' ______________')
    print('     |   |')
    print('   '+board[3]+' | '+board[4]+' |  '+board[5]+'  ')
    print('     |   |')
    print(' ______________')
    print('     |   |')
    print('   '+board[6]+' | '+board[7]+' |  '+board[8]+'  ')
    print('     |   |\n')

def check_win(player_mark, board):
    return (
        (board[0] == board[1] == board[2] == player_mark) or
        (board[3] == board[4] == board[5] == player_mark) or
        (board[6] == board[7] == board[8] == player_mark) or
        (board[0] == board[3] == board[6] == player_mark) or
        (board[1] == board[4] == board[7] == player_mark) or
        (board[2] == board[5] == board[8] == player_mark) or
        (board[0] == board[4] == board[8] == player_mark) or
        (board[2] == board[4] == board[6] == player_mark)
    )

def check_draw(board):
    return ' ' not in board

def board_copy(board):
    dupeBoard = []
    for j in board:
        dupeBoard.append(j)
    return dupeBoard

def test_win_move(board, player_mark, move):
    bCopy = board_copy(board)
    bCopy[move] = player_mark
    return check_win(player_mark, bCopy)

def win_strategy(board):
    for i in [0, 2, 6, 8]:
        if board[i] == ' ':
            return i
    if board[4] == ' ':
        return 4
    for i in [1, 3, 5, 7]:
        if board[i] == ' ':
            return i
```

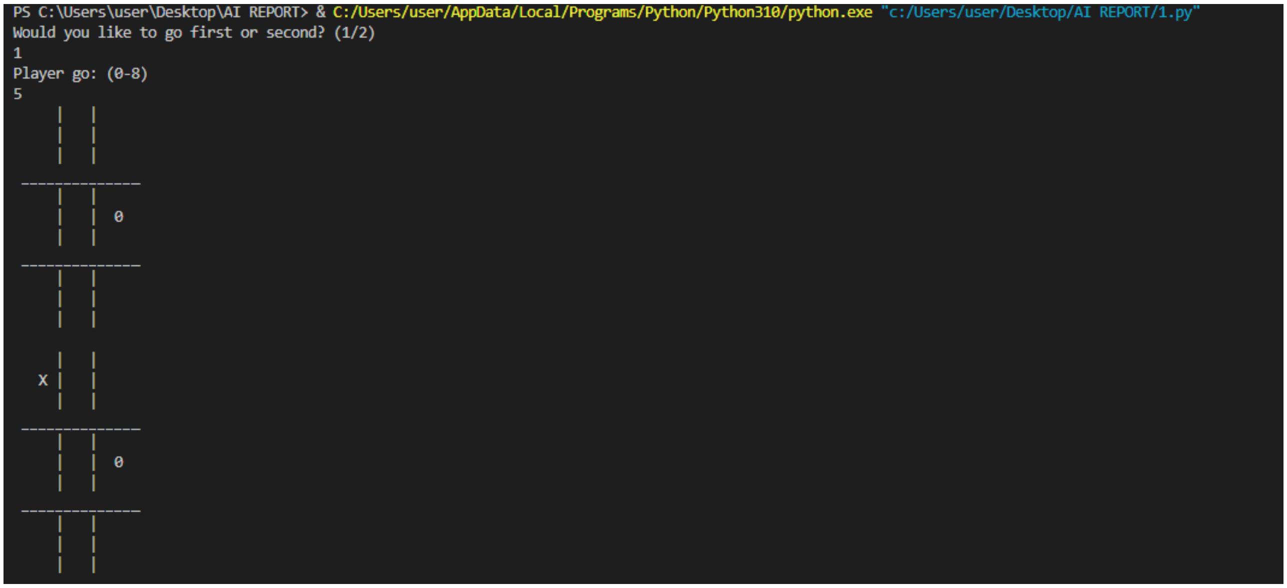


```
def fork_move(board, player_marker, move):
    bCopy = board_copy(board)
    bCopy[move] = player_marker
    winning_moves = 0
    for j in range(0, 9):
        if test_win_move(bCopy, player_marker, j) and bCopy[j] == ' ':
            winning_moves += 1
    return winning_moves >= 2

def get_agent_move(board):
    for i in range(0, 9):
        if board[i] == ' ' and test_win_move(board, 'X', i):
            return i
    for i in range(0, 9):
        if board[i] == ' ' and test_win_move(board, '0', i):
            return i

    for i in range(0, 9):
        if board[i] == ' ' and fork_move(board, 'X', i):
            return i

    for i in range(0, 9):
        if board[i] == ' ' and fork_move(board, '0', i):
            return i
    return win_strategy(board)

def tictactoe():
    Playing = True
    while Playing:
        InGame = True
        board = [' '] * 9
        print('Would you like to go first or second? (1/2)')
        if input() == '1':
            playerMarker = '0'
        else:
            playerMarker = 'X'
            display_board(board)

        while InGame:
            if playerMarker == '0':
                print('Player go: (0-8)')
                move = int(input())
                if board[move] != ' ':
                    print('Invalid move!')
            else:
                move = get_agent_move(board)
```

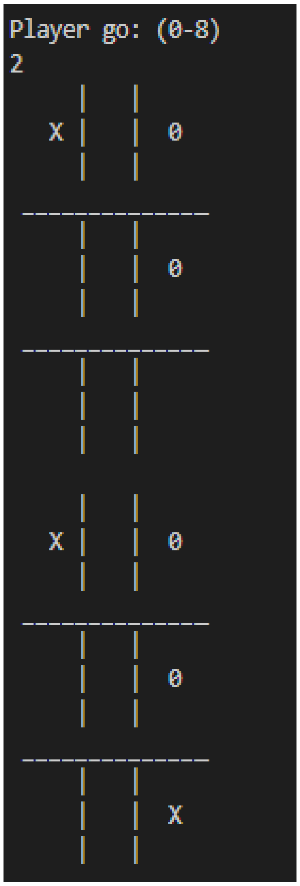


```
                board[move] = playerMarker
                if check_win(playerMarker, board):
                    InGame = False
                    display_board(board)
                    if playerMarker == '0':
                        print('Player wins!')
                    else:
                        print('Agent wins!')
                    continue
                if check_draw(board):
                    InGame = False
                    display_board(board)
                    print('It was a draw!')
                    continue
                display_board(board)
                if playerMarker == '0':
                    playerMarker = 'X'
                else:
                    playerMarker = '0'

            print('Type y to keep playing')
            inp = input().upper()
            if inp != 'Y':
                Playing = False

tictactoe()
```

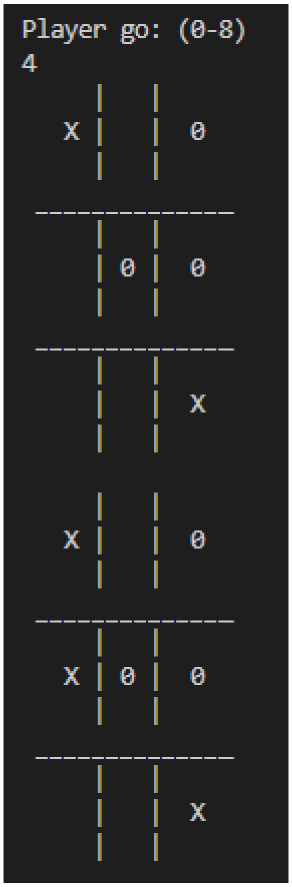
Output:



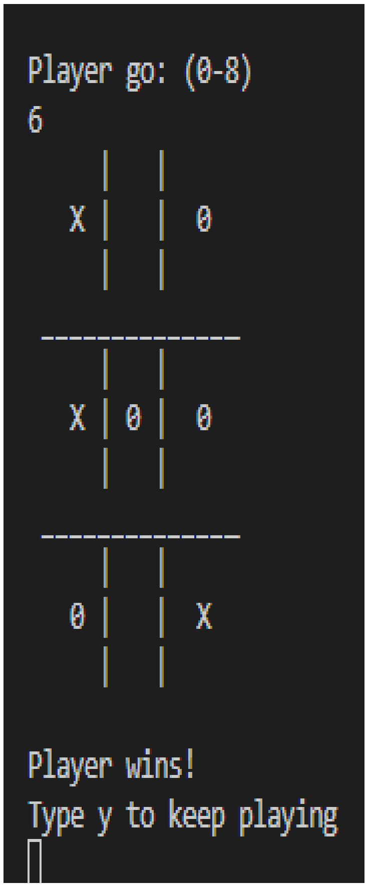
```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/1.py"
Would you like to go first or second? (1/2)
1
Player go: (0-8)
5
     |   |
     |   |
     |   |
 -------------
     |   |
     |   |  0
     |   |
 -------------
     |   |
     |   |
     |   |

     |   |
   x |   |
     |   |
 -------------
     |   |
     |   |  0
     |   |
 -------------
     |   |
     |   |
     |   |
```



```
Player go: (0-8)
2
     |   |
   X |   |  0
     |   |
 -------------
     |   |
     |   |  0
     |   |
 -------------
     |   |
     |   |
     |   |

     |   |
   X |   |  0
     |   |
 -------------
     |   |
     |   |  0
     |   |
 -------------
     |   |
     |   |  X
     |   |
```

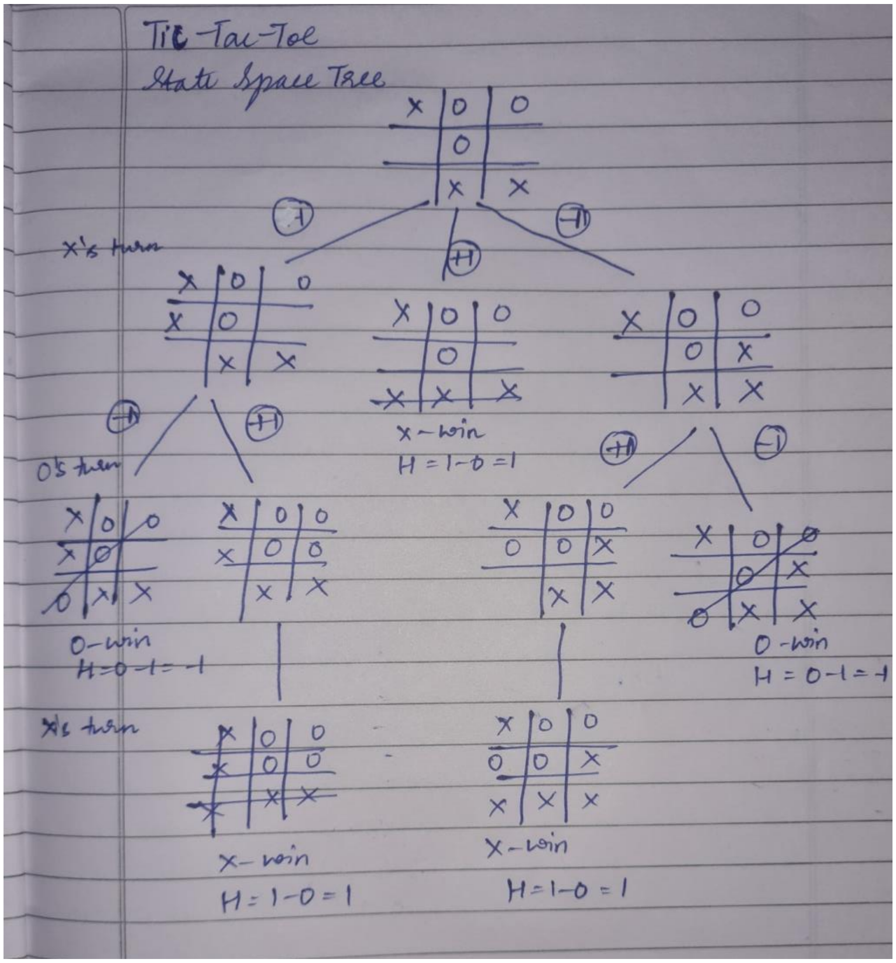


```
Player go: (0-8)
4
     |   |
   X |   |  0
     |   |
 -------------
     |   |
     | 0 |  0
     |   |
 -------------
     |   |
     |   |  X
     |   |

     |   |
   X |   |  0
     |   |
 -------------
     |   |
   X | 0 |  0
     |   |
 -------------
     |   |
     |   |  X
     |   |
```



```
Player go: (0-8)
6
     |   |
   X |   |  0
     |   |
 -------------
     |   |
   X | 0 |  0
     |   |
 -------------
     |   |
   0 |   |  X
     |   |

Player wins!
Type y to keep playing
```

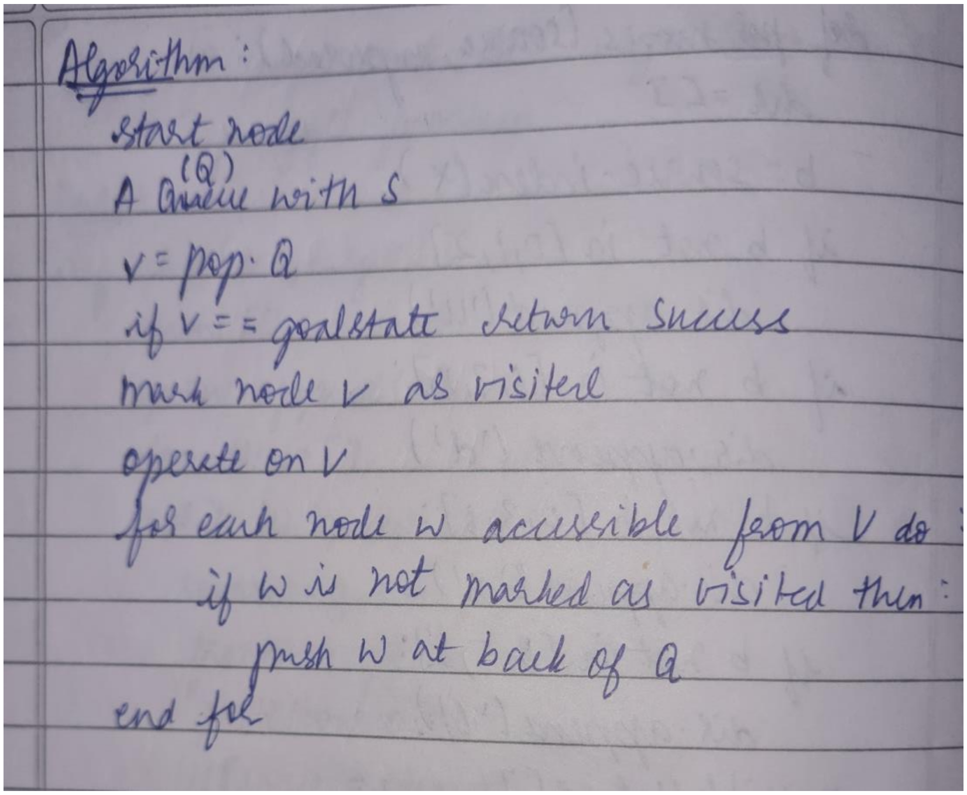
State Space Tree:



16/11/22 **Lab Program 2**

Solve 8 Puzzle Using BFS

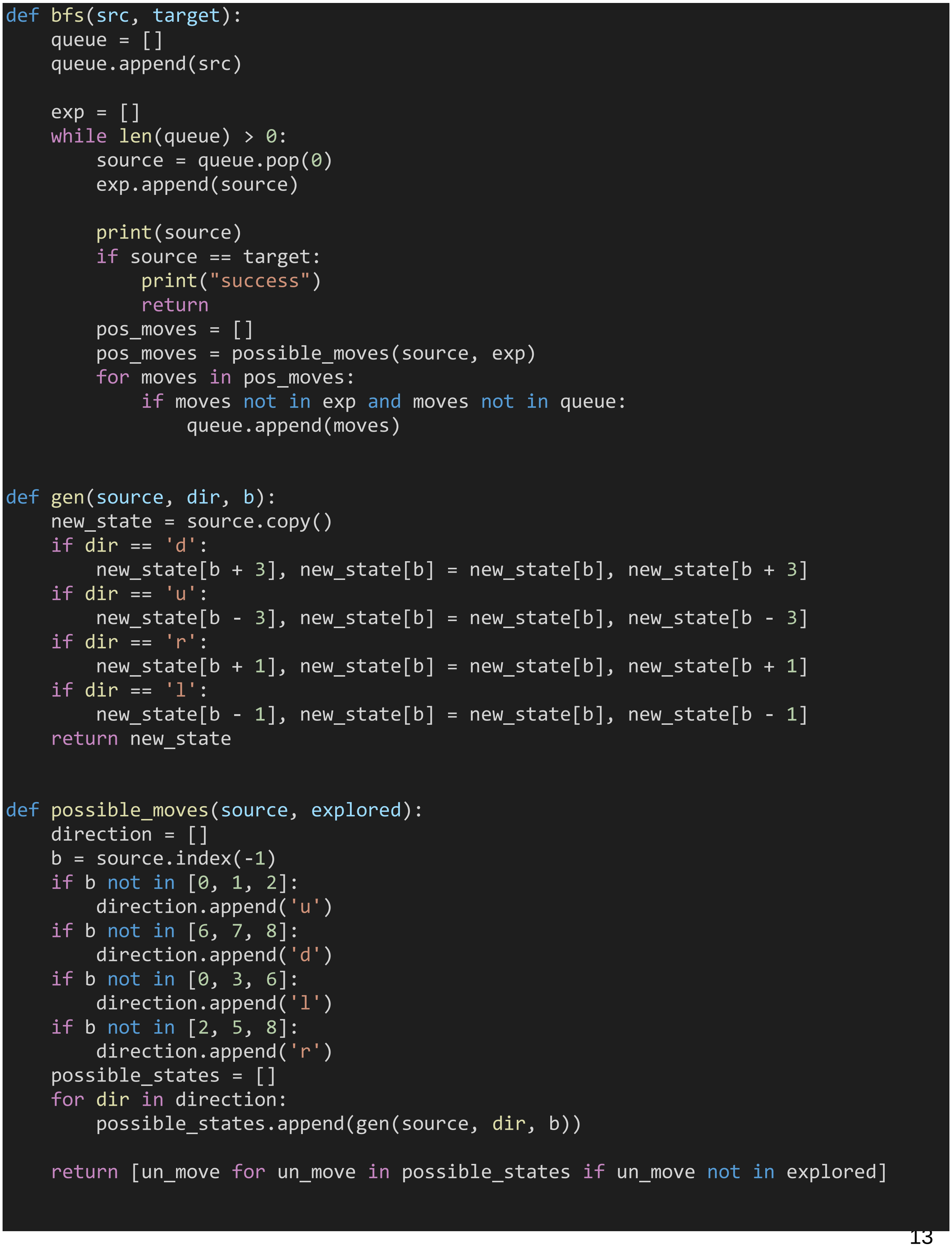
Algorithm:



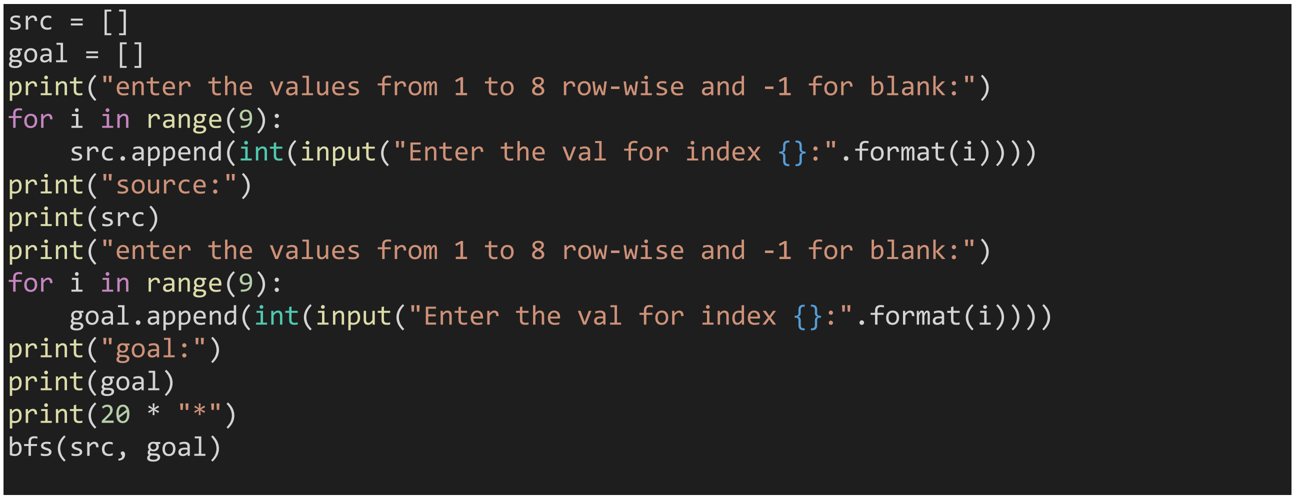
Algorithm :

```
start node
A Queue (Q) with S
v = pop.Q
if v == goalstate return Success
mark node v as visited
operate on v
for each node w accessible from v do :
    if w is not marked as visited then :
        push w at back of Q
end for
```

Code:

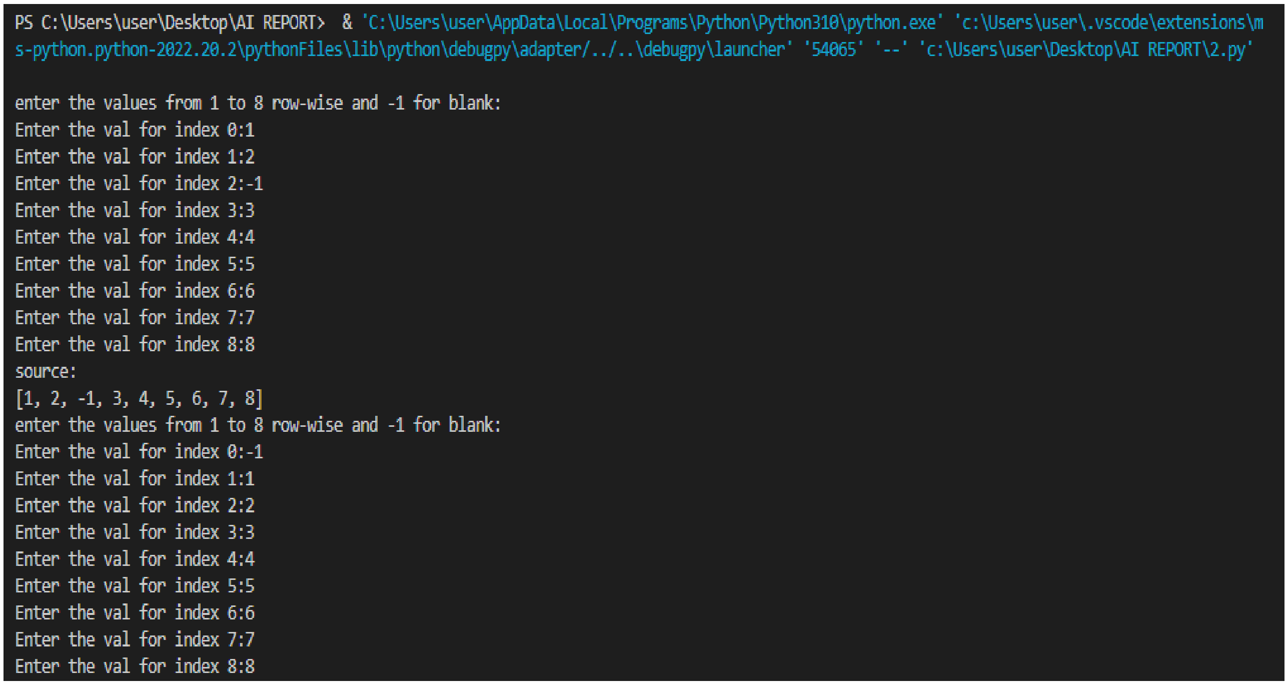


```
def bfs(src, target):
    queue = []
    queue.append(src)

    exp = []
    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print(source)
        if source == target:
            print("success")
            return
        pos_moves = []
        pos_moves = possible_moves(source, exp)
        for moves in pos_moves:
            if moves not in exp and moves not in queue:
                queue.append(moves)


def gen(source, dir, b):
    new_state = source.copy()
    if dir == 'd':
        new_state[b + 3], new_state[b] = new_state[b], new_state[b + 3]
    if dir == 'u':
        new_state[b - 3], new_state[b] = new_state[b], new_state[b - 3]
    if dir == 'r':
        new_state[b + 1], new_state[b] = new_state[b], new_state[b + 1]
    if dir == 'l':
        new_state[b - 1], new_state[b] = new_state[b], new_state[b - 1]
    return new_state


def possible_moves(source, explored):
    direction = []
    b = source.index(-1)
    if b not in [0, 1, 2]:
        direction.append('u')
    if b not in [6, 7, 8]:
        direction.append('d')
    if b not in [0, 3, 6]:
        direction.append('l')
    if b not in [2, 5, 8]:
        direction.append('r')
    possible_states = []
    for dir in direction:
        possible_states.append(gen(source, dir, b))

    return [un_move for un_move in possible_states if un_move not in explored]
```

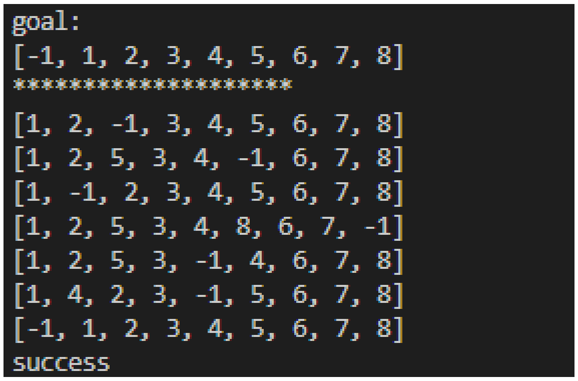


```python
src = []
goal = []
print("enter the values from 1 to 8 row-wise and -1 for blank:")
for i in range(9):
    src.append(int(input("Enter the val for index {}:".format(i))))
print("source:")
print(src)
print("enter the values from 1 to 8 row-wise and -1 for blank:")
for i in range(9):
    goal.append(int(input("Enter the val for index {}:".format(i))))
print("goal:")
print(goal)
print(20 * "*")
bfs(src, goal)
```

Output:

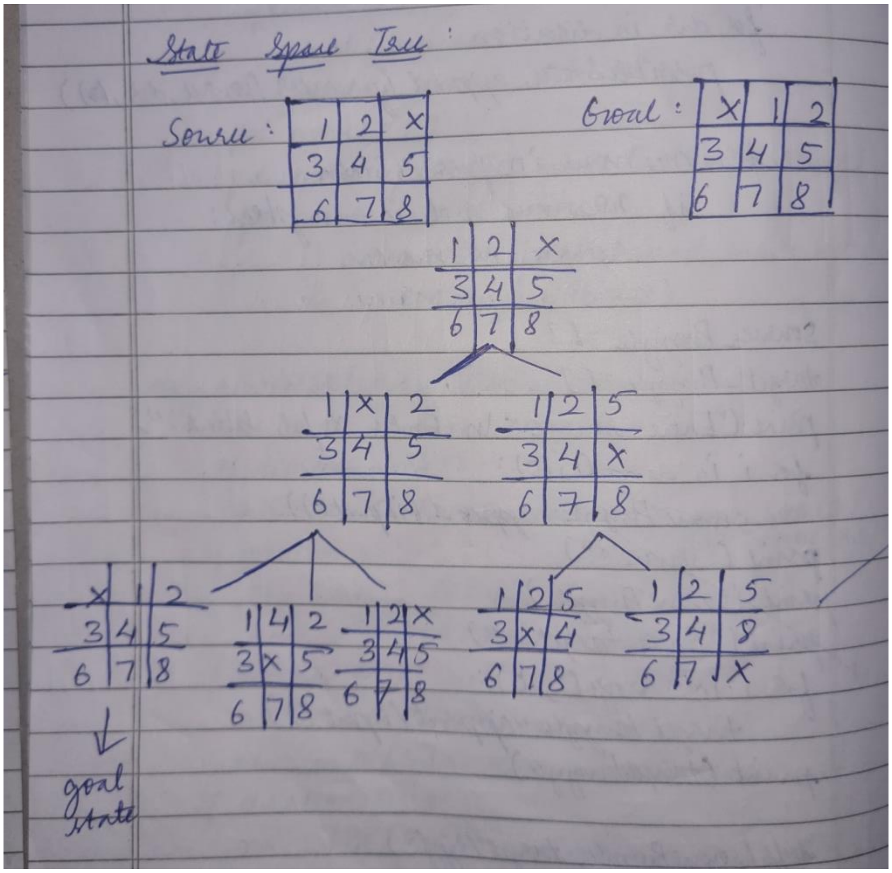


```
PS C:\Users\user\Desktop\AI REPORT>  & 'C:\Users\user\AppData\Local\Programs\Python\Python310\python.exe' 'c:\Users\user\.vscode\extensions\ms-python.python-2022.20.2\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '54065' '--' 'c:\Users\user\Desktop\AI REPORT\2.py'

enter the values from 1 to 8 row-wise and -1 for blank:
Enter the val for index 0:1
Enter the val for index 1:2
Enter the val for index 2:-1
Enter the val for index 3:3
Enter the val for index 4:4
Enter the val for index 5:5
Enter the val for index 6:6
Enter the val for index 7:7
Enter the val for index 8:8
source:
[1, 2, -1, 3, 4, 5, 6, 7, 8]
enter the values from 1 to 8 row-wise and -1 for blank:
Enter the val for index 0:-1
Enter the val for index 1:1
Enter the val for index 2:2
Enter the val for index 3:3
Enter the val for index 4:4
Enter the val for index 5:5
Enter the val for index 6:6
Enter the val for index 7:7
Enter the val for index 8:8
```



```
goal:
[-1, 1, 2, 3, 4, 5, 6, 7, 8]
********************
[1, 2, -1, 3, 4, 5, 6, 7, 8]
[1, 2, 5, 3, 4, -1, 6, 7, 8]
[1, -1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 5, 3, 4, 8, 6, 7, -1]
[1, 2, 5, 3, -1, 4, 6, 7, 8]
[1, 4, 2, 3, -1, 5, 6, 7, 8]
[-1, 1, 2, 3, 4, 5, 6, 7, 8]
success
```

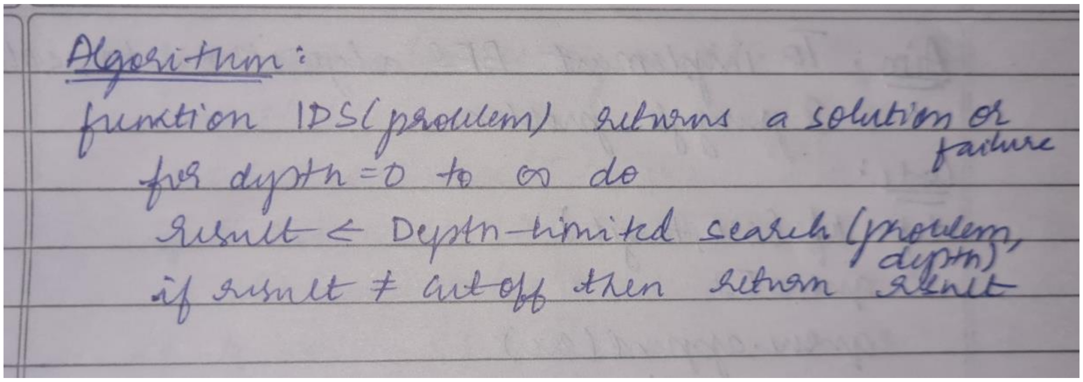
State Space Tree:



23/11/22 **Lab Program 3**

Implement Iterative deepening search algorithm

Algorithm:



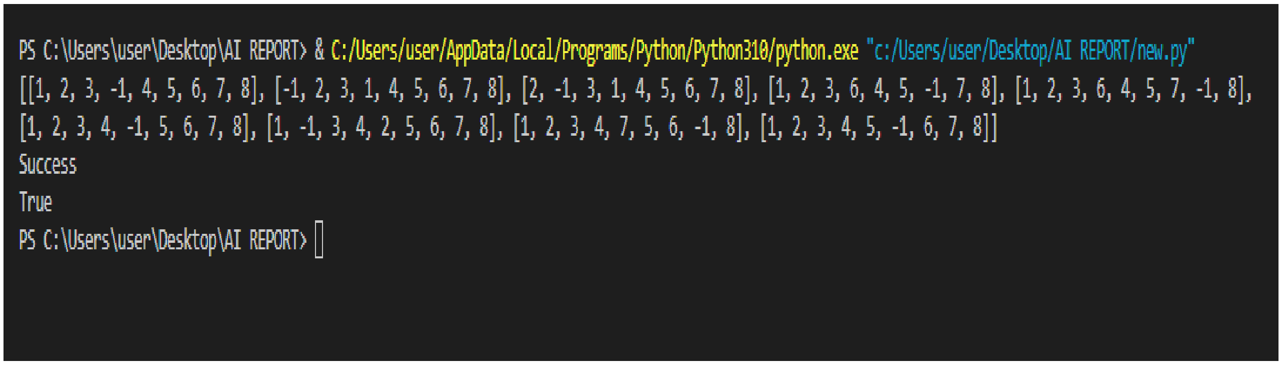
Algorithm:

```
function IDS(problem) returns a solution or failure
    for depth = 0 to ∞ do
        result ← Depth-limited search(problem, depth)
        if result ≠ cutoff then return result
```

Code:

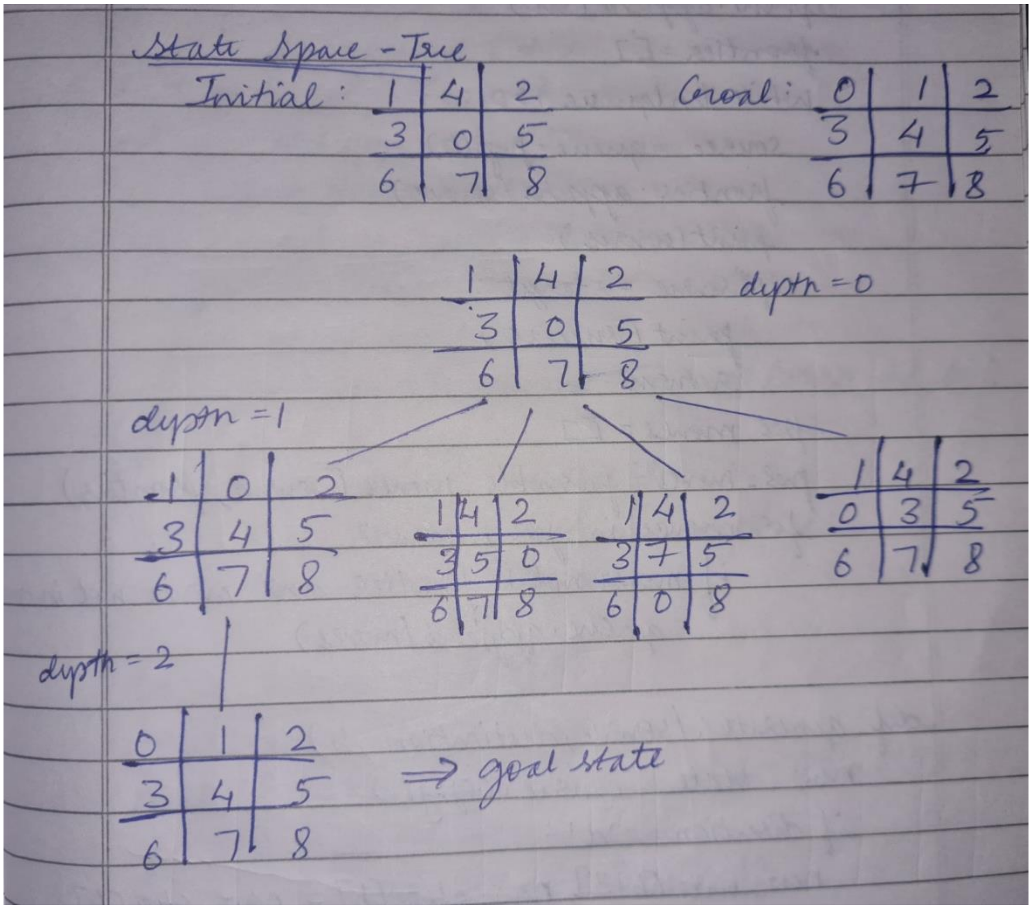
```
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5, -1, 6, 7, 8]


def iddfs(src, target, depth):
    for limit in range(0, depth+1):
        visited_states = []
        visited_states.append(src)
        if dfs(src, target, limit, visited_states):
            print(visited_states)
            print("Success")
            return True
    return False


def gen(state, m, b):
    temp = state[:]
    if m == 'l':
        temp[b], temp[b-1] = temp[b-1], temp[b]
    if m == 'r':
        temp[b], temp[b+1] = temp[b+1], temp[b]
    if m == 'd':
        temp[b], temp[b+3] = temp[b+3], temp[b]
    if m == 'u':
        temp[b], temp[b-3] = temp[b-3], temp[b]
    return temp


def next_state(state):
    blank = state.index(-1)
    moves = []
    if blank >= 3:
        moves.append('u')
    if blank <= 5:
        moves.append('d')
    if (blank % 3) > 0:
        moves.append('l')
    if (blank % 3) < 2:
        moves.append('r')
    return moves, blank


def dfs(src, target, limit, visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False
    moves, blank = next_state(src)
    for move in moves:
        nextmove = gen(src, move, blank)
        if not nextmove in visited_states:
            visited_states.append(nextmove)
            if dfs(nextmove, target, limit-1, visited_states):
                return True
    return False


print(iddfs(src, target, 2))
```

Output:



```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
[[1, 2, 3, -1, 4, 5, 6, 7, 8], [-1, 2, 3, 1, 4, 5, 6, 7, 8], [2, -1, 3, 1, 4, 5, 6, 7, 8], [1, 2, 3, 6, 4, 5, -1, 7, 8], [1, 2, 3, 6, 4, 5, 7, -1, 8],
[1, 2, 3, 4, -1, 5, 6, 7, 8], [1, -1, 3, 4, 2, 5, 6, 7, 8], [1, 2, 3, 4, 7, 5, 6, -1, 8], [1, 2, 3, 4, 5, -1, 6, 7, 8]]
Success
True
PS C:\Users\user\Desktop\AI REPORT> 
```

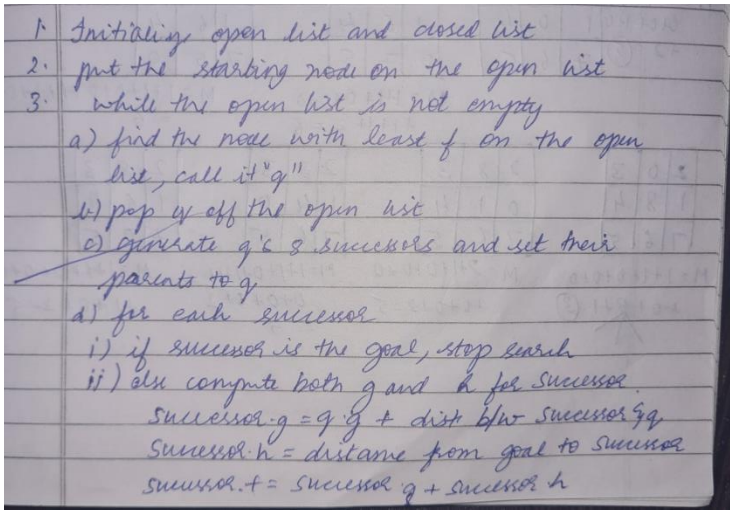
State Space Tree:



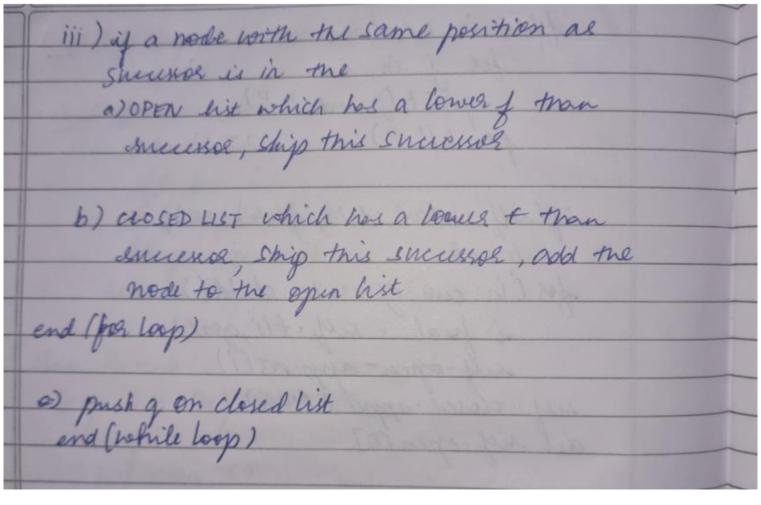
30/11/22 **Lab Program 4**

Implement A* search algorithm

Algorithm:



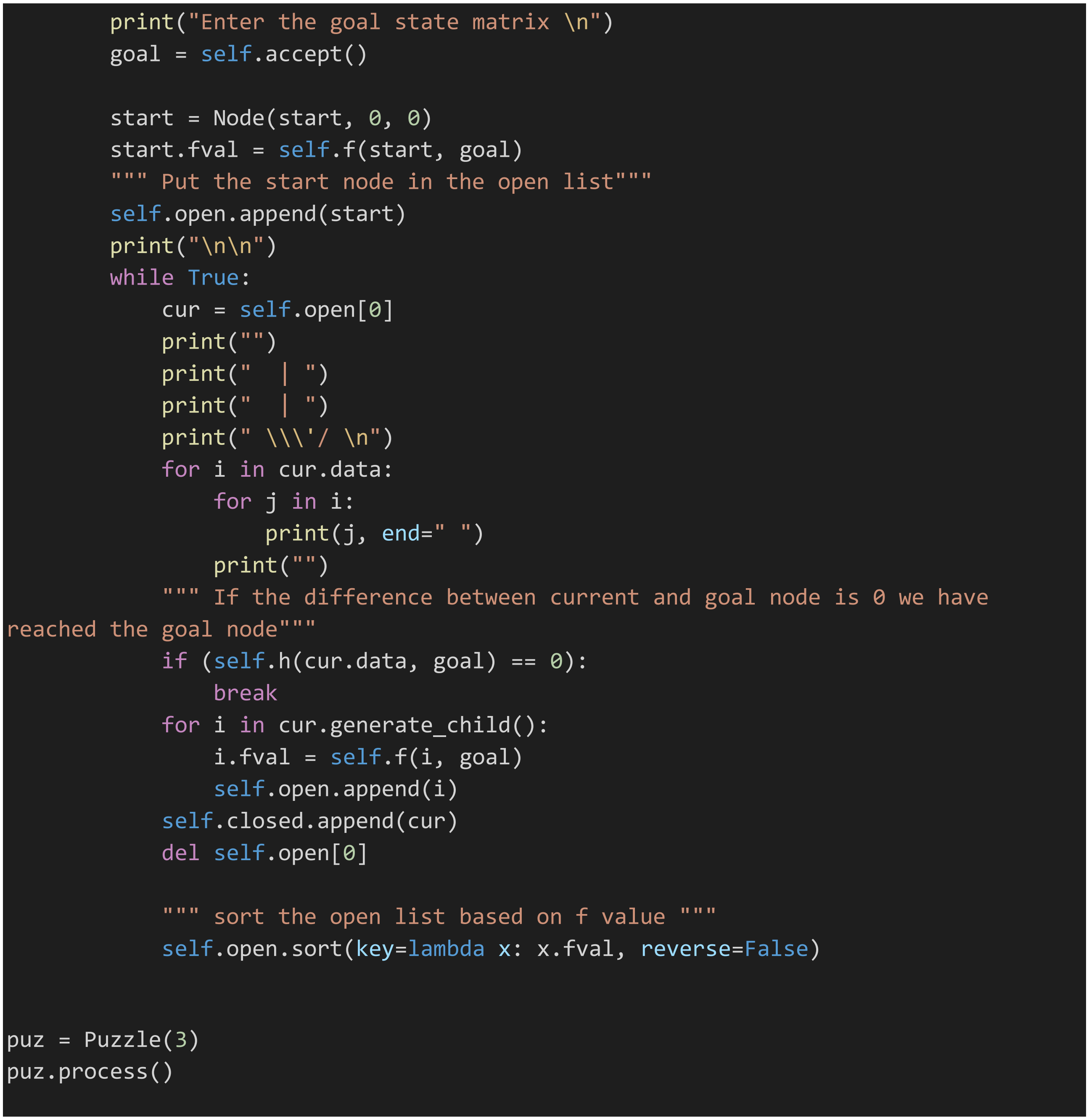
```
1. Initialize open list and closed list
2. put the starting node on the open list
3. while the open list is not empty
   a) find the node with least f on the open
      list, call it "q"
   b) pop q off the open list
   c) generate q's 8 successors and set their
      parents to q
   d) for each successor
      i) if successor is the goal, stop search
      ii) else compute both g and h for successor
            successor.g = q.g + dist b/w successor & q
            successor.h = distance from goal to successor
            successor.f = successor.g + successor.h
```



```
      iii) if a node with the same position as
           successor is in the
           a) OPEN list which has a lower f than
              successor, skip this successor

           b) CLOSED LIST which has a lower f than
              successor, skip this successor, add the
              node to the open list
   end (for loop)

   e) push q on closed list
   end (while loop)
```

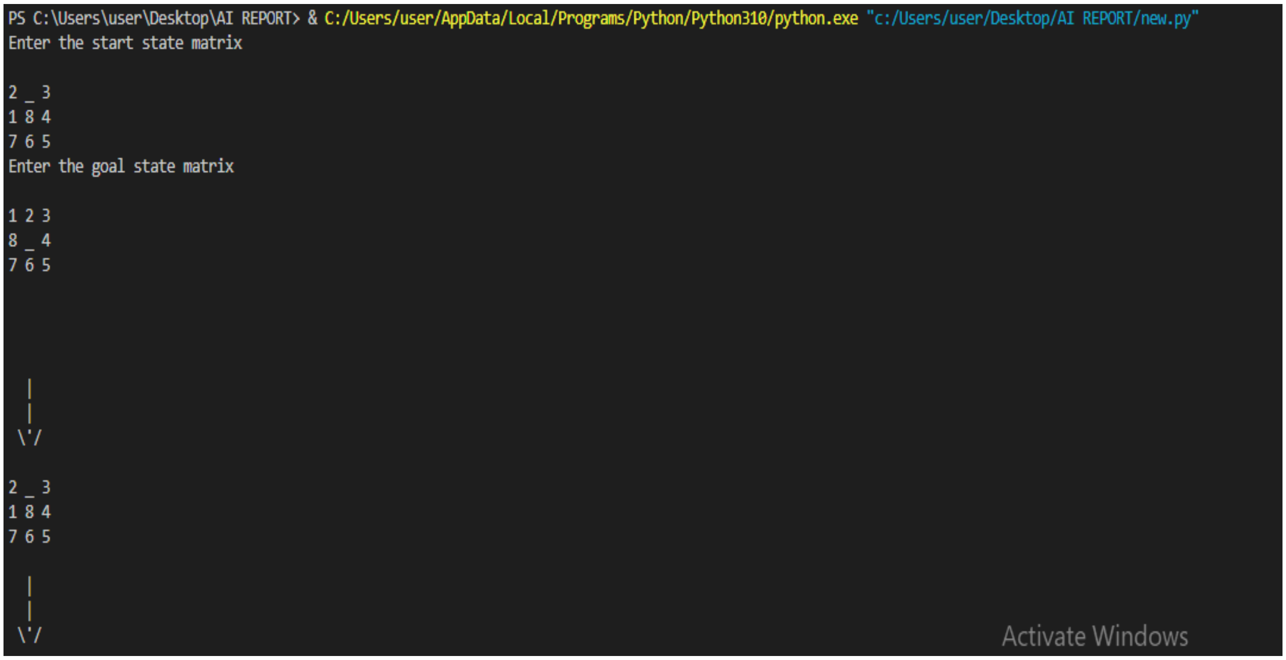
Code:

```
class Node:
    def __init__(self, data, level, fval):
        """ Initialize the node with the data, level of the node and the calculated
fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x, y = self.find(self.data, '_')
        """ val_list contains position values for moving the blank space in either
of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level + 1, 0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
        """ Move the blank space in the given direction and if the position value
are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self, root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        """ Specifically used to find the position of the blank space """
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j


class Puzzle:
    def __init__(self, size):
        """ Initialize the puzzle size by the specified size,open and closed lists
to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
```

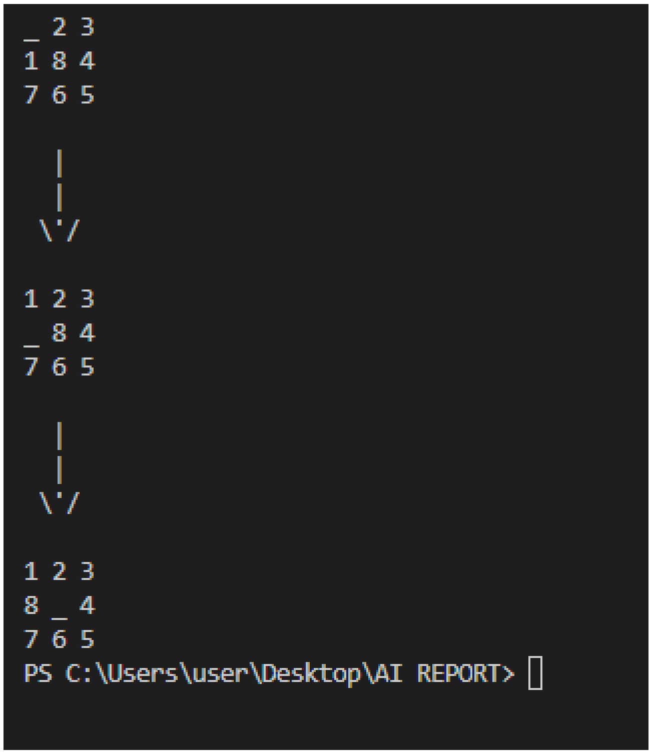


```
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have
reached the goal node"""
            if (self.h(cur.data, goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the open list based on f value """
            self.open.sort(key=lambda x: x.fval, reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

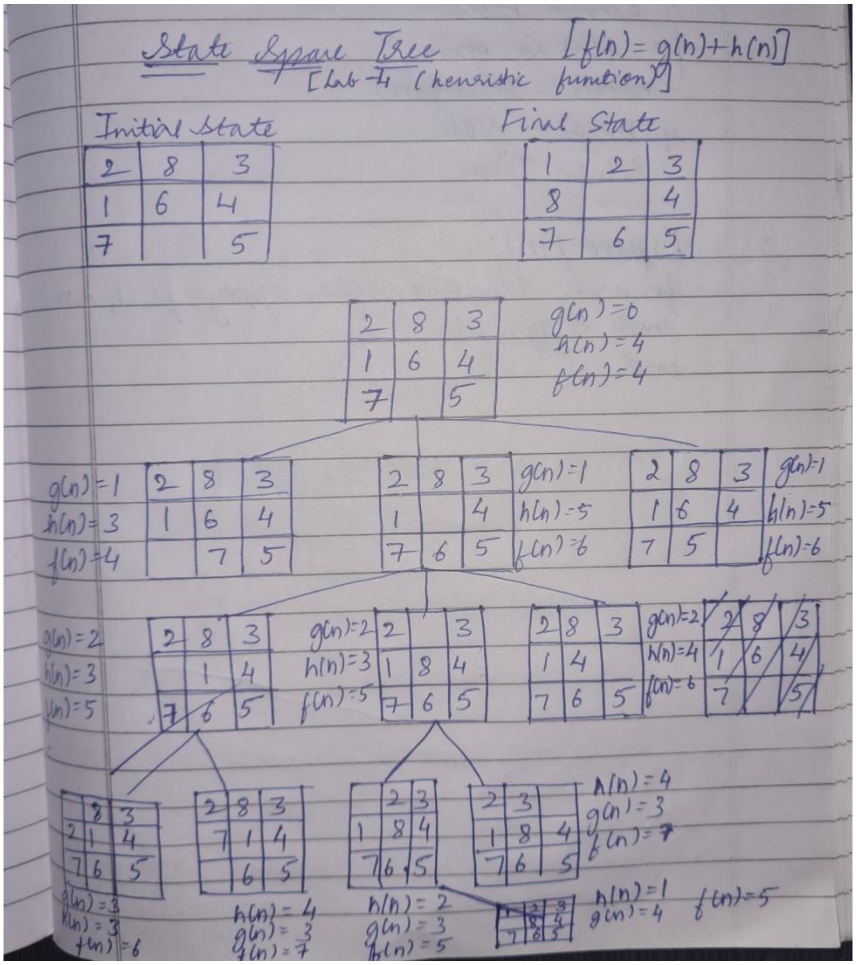


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter the start state matrix

2 _ 3
1 8 4
7 6 5
Enter the goal state matrix

1 2 3
8 _ 4
7 6 5




  |
  |
 \'/

2 _ 3
1 8 4
7 6 5

  |
  |
 \'/
```



```
_ 2 3
1 8 4
7 6 5

  |
  |
 \'/

1 2 3
_ 8 4
7 6 5

  |
  |
 \'/

1 2 3
8 _ 4
7 6 5
PS C:\Users\user\Desktop\AI REPORT> 
```

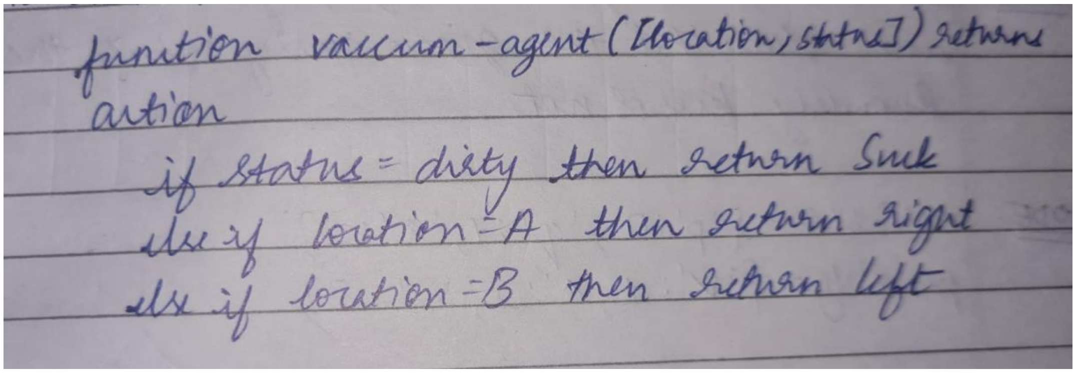
State Space Tree:



7/12/22 **Lab Program 5**

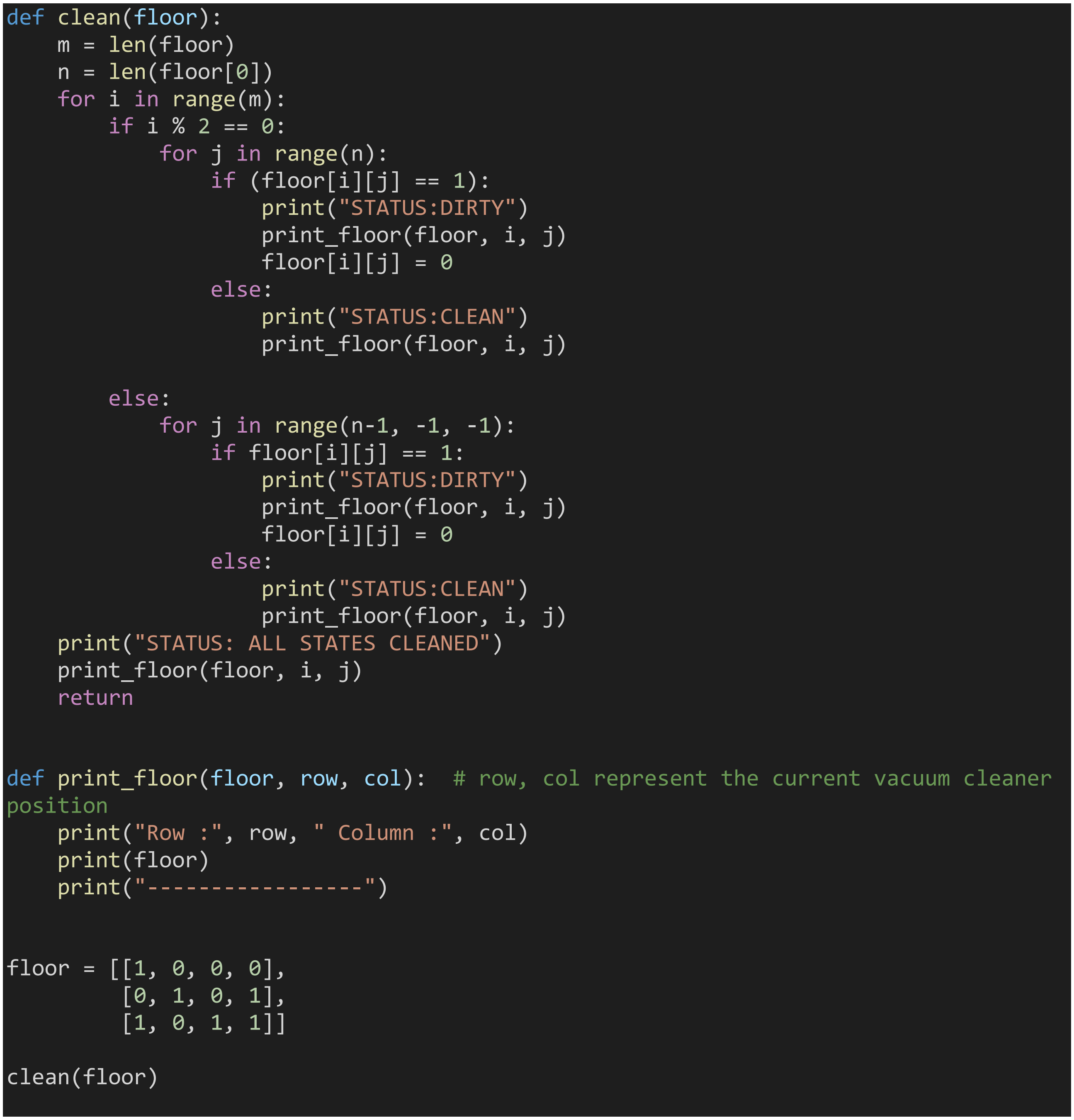
Implement vacuum cleaner agent

Algorithm:



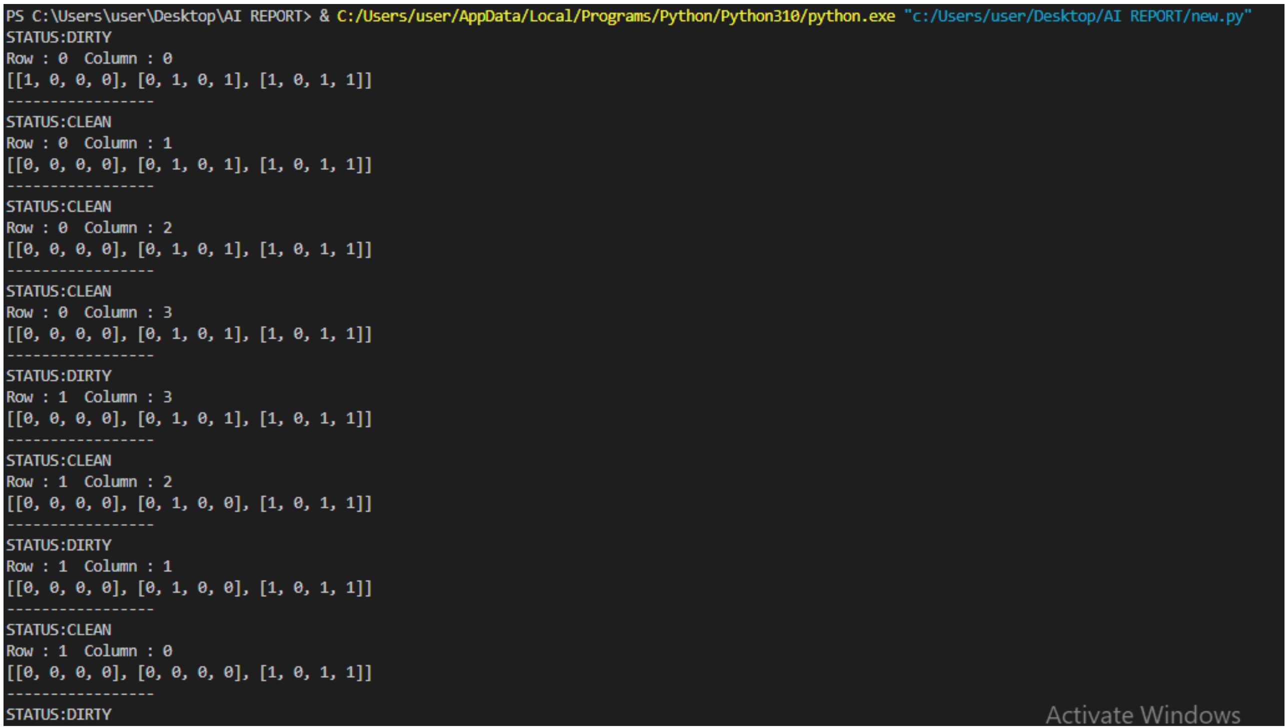
```
function vaccum-agent ([location, status]) returns
action
    if status = dirty then return Suck
    else if location = A then return right
    else if location = B then return left
```

Code:

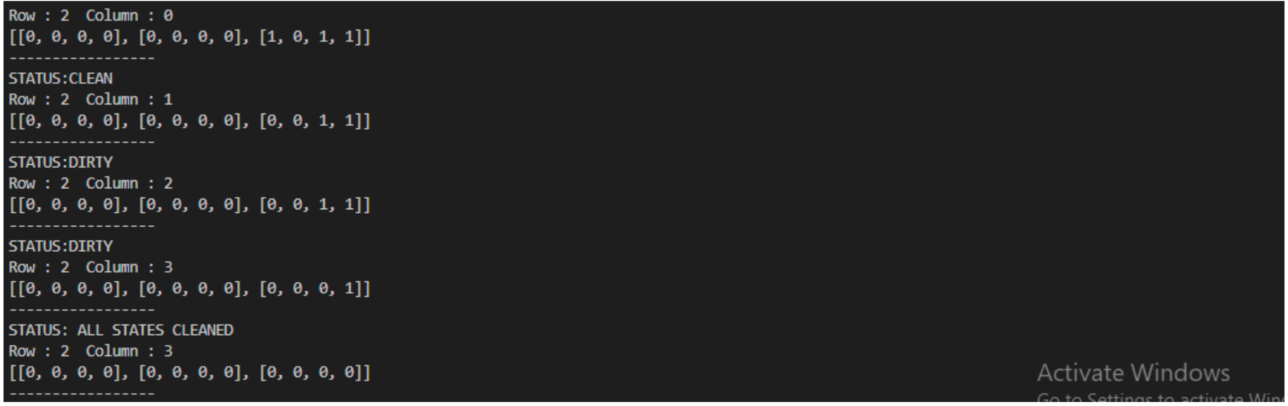


```
def clean(floor):
    m = len(floor)
    n = len(floor[0])
    for i in range(m):
        if i % 2 == 0:
            for j in range(n):
                if (floor[i][j] == 1):
                    print("STATUS:DIRTY")
                    print_floor(floor, i, j)
                    floor[i][j] = 0
                else:
                    print("STATUS:CLEAN")
                    print_floor(floor, i, j)

        else:
            for j in range(n-1, -1, -1):
                if floor[i][j] == 1:
                    print("STATUS:DIRTY")
                    print_floor(floor, i, j)
                    floor[i][j] = 0
                else:
                    print("STATUS:CLEAN")
                    print_floor(floor, i, j)
    print("STATUS: ALL STATES CLEANED")
    print_floor(floor, i, j)
    return


def print_floor(floor, row, col):  # row, col represent the current vacuum cleaner
position
    print("Row :", row, " Column :", col)
    print(floor)
    print("-----------------")


floor = [[1, 0, 0, 0],
         [0, 1, 0, 1],
         [1, 0, 1, 1]]

clean(floor)
```

Output:

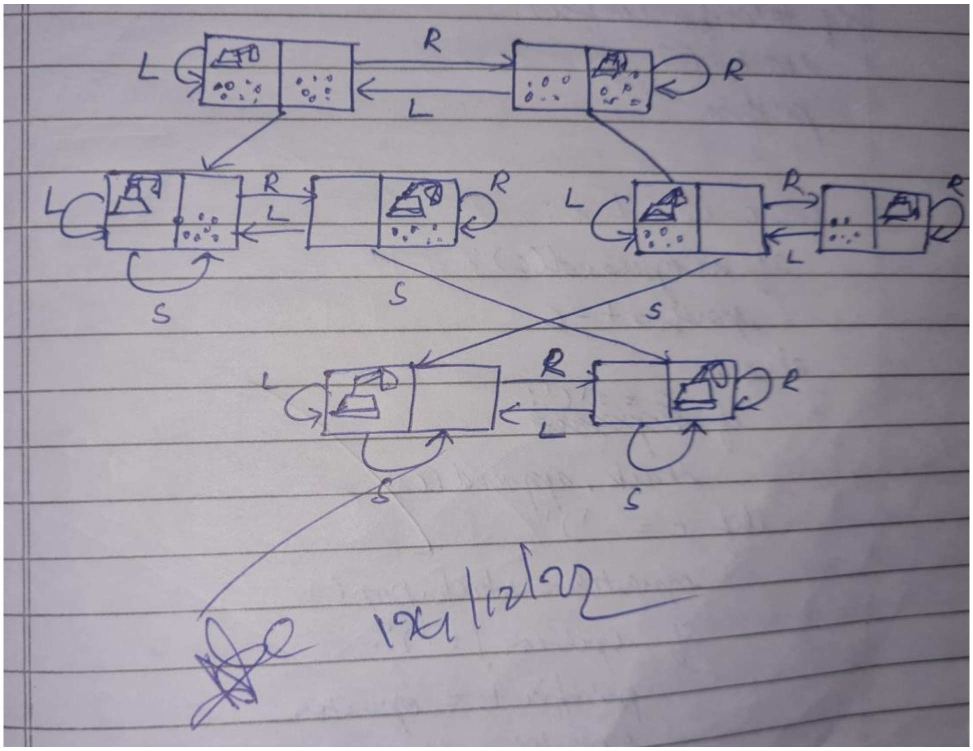


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
STATUS:DIRTY
Row : 0  Column : 0
[[1, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 0  Column : 1
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 0  Column : 2
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 0  Column : 3
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
-----------------
STATUS:DIRTY
Row : 1  Column : 3
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 1  Column : 2
[[0, 0, 0, 0], [0, 1, 0, 0], [1, 0, 1, 1]]
-----------------
STATUS:DIRTY
Row : 1  Column : 1
[[0, 0, 0, 0], [0, 1, 0, 0], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 1  Column : 0
[[0, 0, 0, 0], [0, 0, 0, 0], [1, 0, 1, 1]]
-----------------
STATUS:DIRTY
```



```
Row : 2  Column : 0
[[0, 0, 0, 0], [0, 0, 0, 0], [1, 0, 1, 1]]
-----------------
STATUS:CLEAN
Row : 2  Column : 1
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 1]]
-----------------
STATUS:DIRTY
Row : 2  Column : 2
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 1]]
-----------------
STATUS:DIRTY
Row : 2  Column : 3
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 1]]
-----------------
STATUS: ALL STATES CLEANED
Row : 2  Column : 3
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
-----------------
```

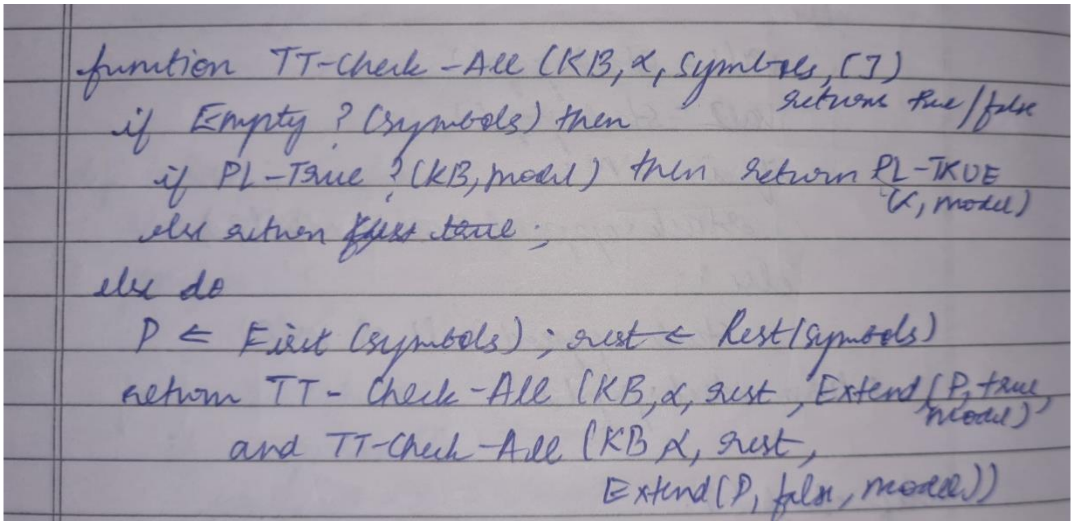
State Space Tree:



28/12/22 **Lab Program 6**

Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.

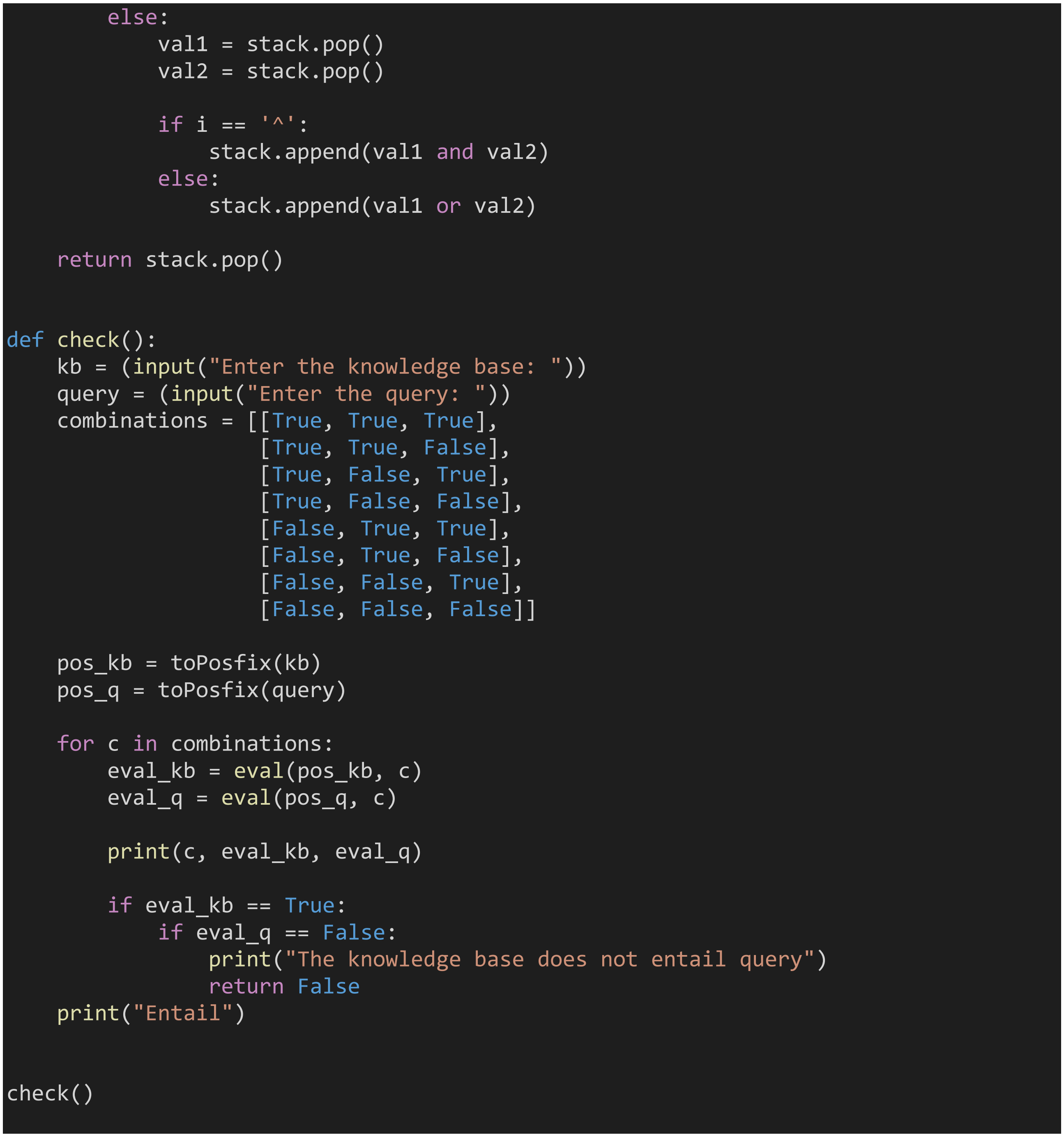
Algorithm:



```
function TT-Check-All (KB, α, symbols, [ ])    returns true/false
  if Empty ? (symbols) then
    if PL-True ? (KB, model) then return PL-TRUE (α, model)
    else return ~~false~~ true;
  else do
    P ⇐ First (symbols) ; rest ⇐ Rest (symbols)
    return TT-Check-All (KB, α, rest, Extend (P, true, model))
        and TT-Check-All (KB α, rest,
                          Extend (P, false, model))
```

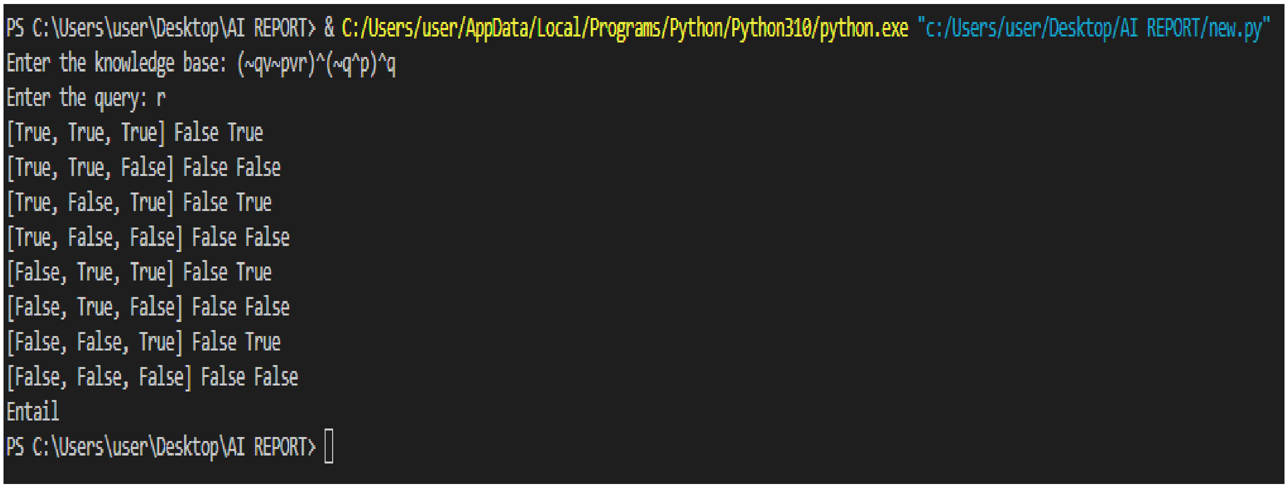
Code:

```
variable = {'p': 0, 'q': 1, 'r': 2}
priority = {'v': 1, '^': 2, '~': 3}


def isoperand(c):
    return c.isalpha() and c != 'v'


def haslessEqual(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False


def toPosfix(infix):
    stack = []
    posfix = ''

    for c in infix:
        if isoperand(c):
            posfix += c
        else:
            if c == '(':
                stack.append(c)
            elif c == ')':
                operator = stack.pop()

                if operator != ')':
                    posfix += operator
                    operator = stack.pop()
            else:
                while len(stack) != 0 and haslessEqual(c, stack[-1]):
                    posfix += stack.pop()

                stack.append(c)

    while len(stack) != 0:
        posfix += stack.pop()

    return posfix


def eval(post, comb):
    stack = []
    for i in post:
        if isoperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
```



```
        else:
            val1 = stack.pop()
            val2 = stack.pop()

            if i == '^':
                stack.append(val1 and val2)
            else:
                stack.append(val1 or val2)

    return stack.pop()


def check():
    kb = (input("Enter the knowledge base: "))
    query = (input("Enter the query: "))
    combinations = [[True, True, True],
                    [True, True, False],
                    [True, False, True],
                    [True, False, False],
                    [False, True, True],
                    [False, True, False],
                    [False, False, True],
                    [False, False, False]]

    pos_kb = toPosfix(kb)
    pos_q = toPosfix(query)

    for c in combinations:
        eval_kb = eval(pos_kb, c)
        eval_q = eval(pos_q, c)

        print(c, eval_kb, eval_q)

        if eval_kb == True:
            if eval_q == False:
                print("The knowledge base does not entail query")
                return False
    print("Entail")


check()
```

OUTPUT:

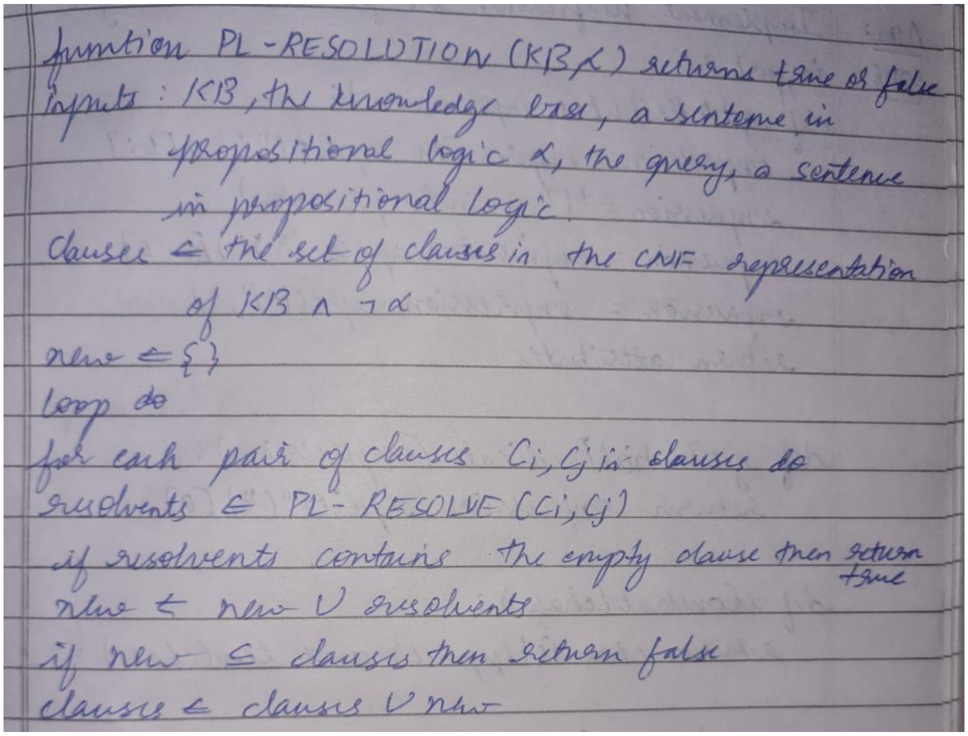


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter the knowledge base: (~qv~pvr)^(~q^p)^q
Enter the query: r
[True, True, True] False True
[True, True, False] False False
[True, False, True] False True
[True, False, False] False False
[False, True, True] False True
[False, True, False] False False
[False, False, True] False True
[False, False, False] False False
Entail
PS C:\Users\user\Desktop\AI REPORT> 
```

4/1/22 **Lab Program 7**

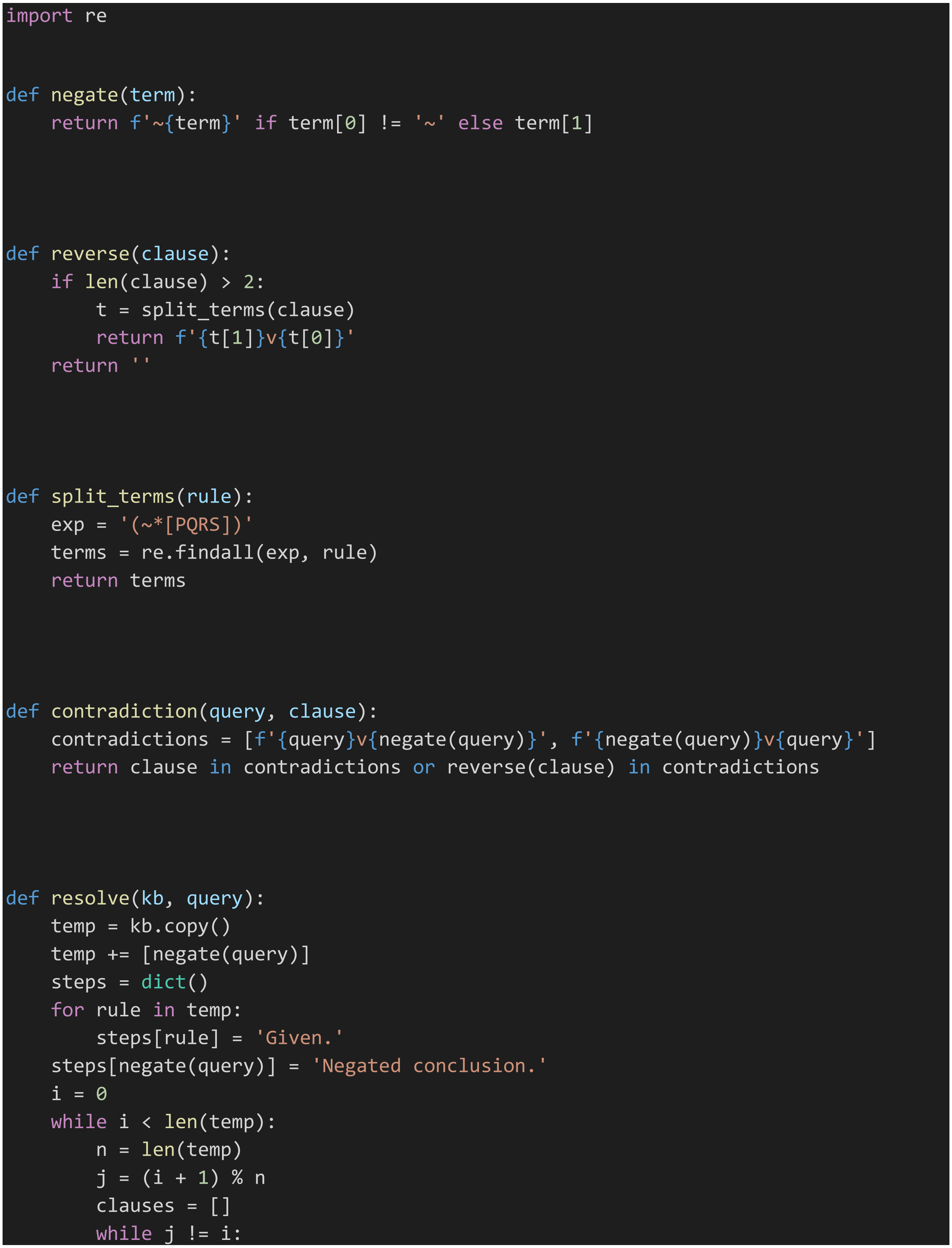
Create a knowledgebase using prepositional logic and prove the given query using resolution

Algorithm:

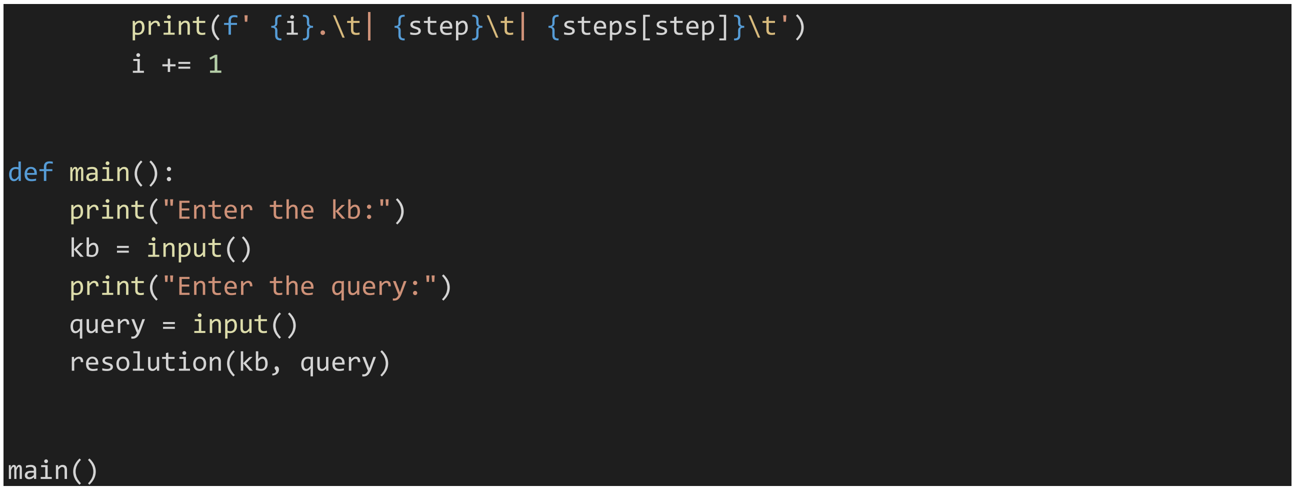


```
funtion PL-RESOLUTION (KB, $\alpha$) returns true or false
inputs : KB, the knowledge base, a sentence in
        propositional logic $\alpha$, the query, a sentence
        in propositional logic.
clauses $\leftarrow$ the set of clauses in the CNF representation
        of KB $\wedge \neg \alpha$
new $\leftarrow$ { }
loop do
for each pair of clauses Ci, Cj in clauses do
resolvents $\leftarrow$ PL-RESOLVE (Ci, Cj)
 if resolvents contains the empty clause then return true
 new $\leftarrow$ new $\cup$ resolvents
if new $\subseteq$ clauses then return false
clauses $\leftarrow$ clauses $\cup$ new
```

Code:

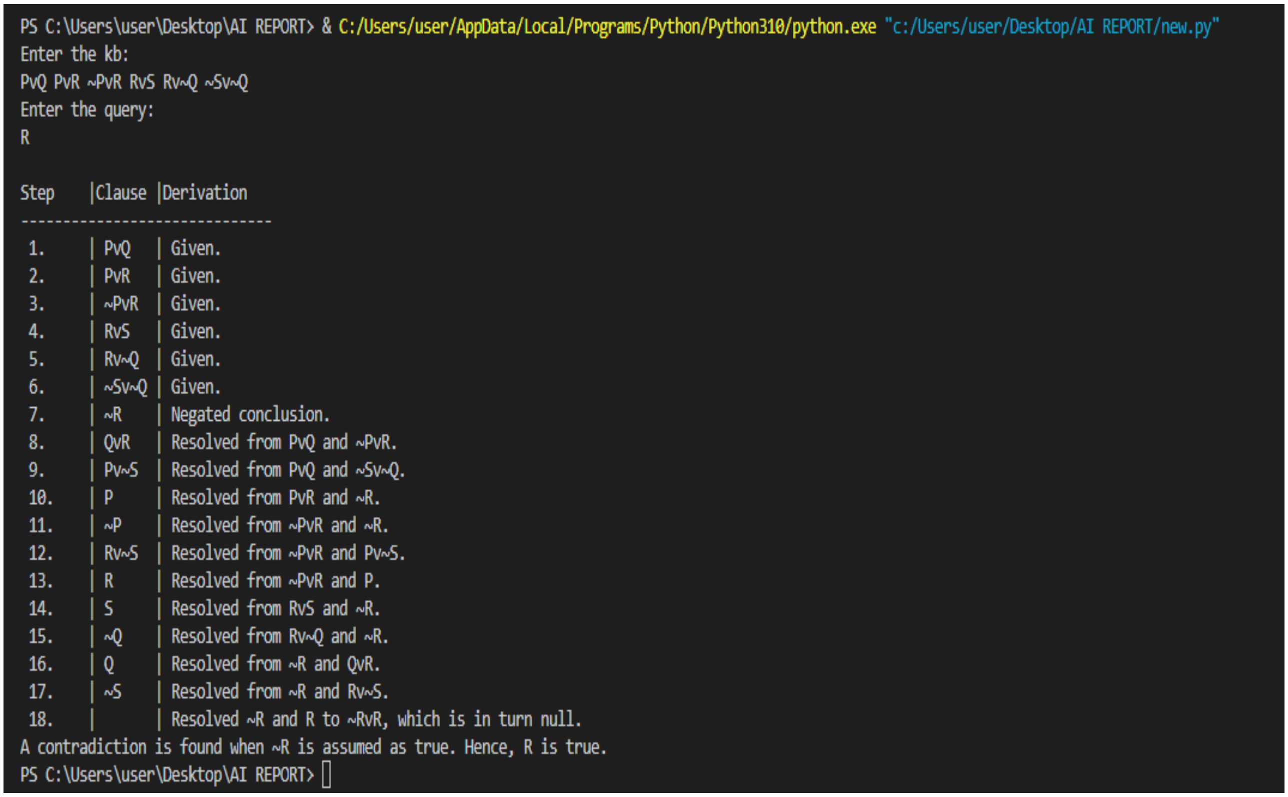


```
import re


def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]




def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''




def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms




def contradiction(query, clause):
    contradictions = [f'{query}v{negate(query)}', f'{negate(query)}v{query}']
    return clause in contradictions or reverse(clause) in contradictions




def resolve(kb, query):
    temp = kb.copy()
    temp += [negate(query)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(query)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(query, f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when {negate(query)} is assumed as true. Hence, {query} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if contradiction(query, f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                            \nA contradiction is found when {negate(query)} is assumed as true. Hence, {query} is true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps



def resolution(kb, query):
    kb = kb.split(' ')
    steps = resolve(kb, query)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
```



```
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1


def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb, query)


main()
```

Output:

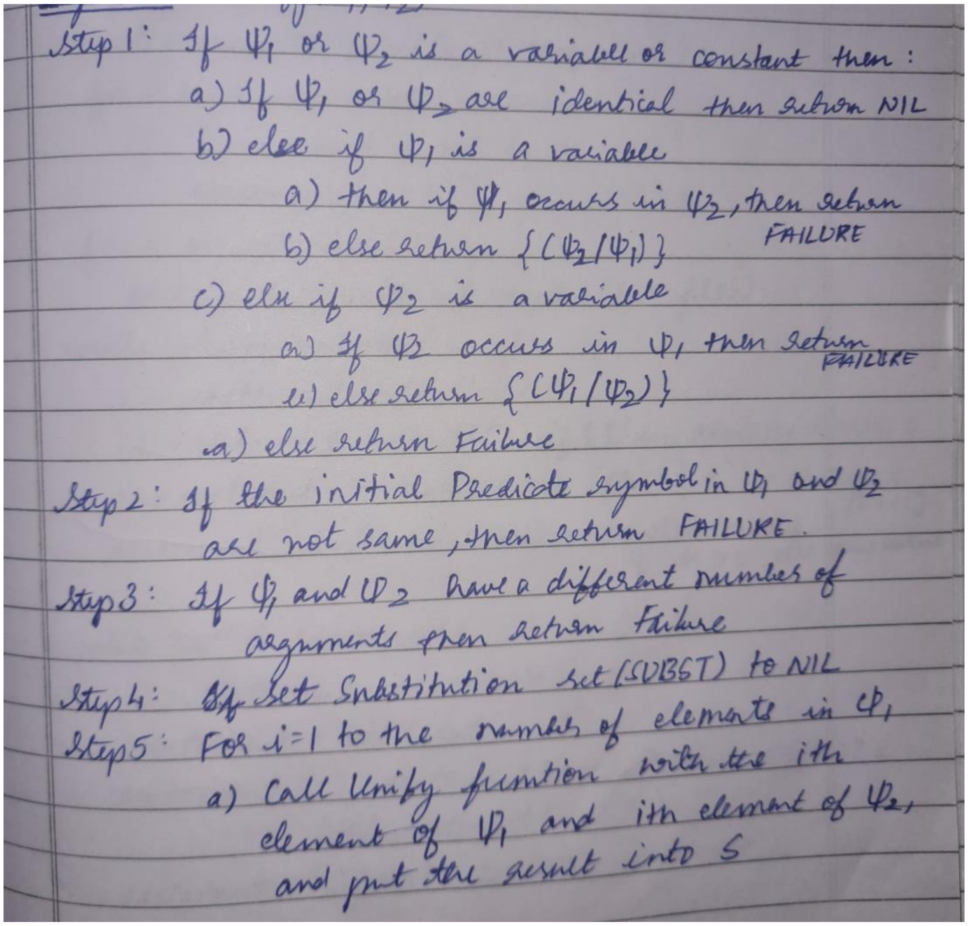


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter the kb:
PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
Enter the query:
R

Step    |Clause |Derivation
-----------------------------
 1.     | PvQ   | Given.
 2.     | PvR   | Given.
 3.     | ~PvR  | Given.
 4.     | RvS   | Given.
 5.     | Rv~Q  | Given.
 6.     | ~Sv~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PvQ and ~PvR.
 9.     | Pv~S  | Resolved from PvQ and ~Sv~Q.
 10.    | P     | Resolved from PvR and ~R.
 11.    | ~P    | Resolved from ~PvR and ~R.
 12.    | Rv~S  | Resolved from ~PvR and Pv~S.
 13.    | R     | Resolved from ~PvR and P.
 14.    | S     | Resolved from RvS and ~R.
 15.    | ~Q    | Resolved from Rv~Q and ~R.
 16.    | Q     | Resolved from ~R and QvR.
 17.    | ~S    | Resolved from ~R and Rv~S.
 18.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
PS C:\Users\user\Desktop\AI REPORT> 
```

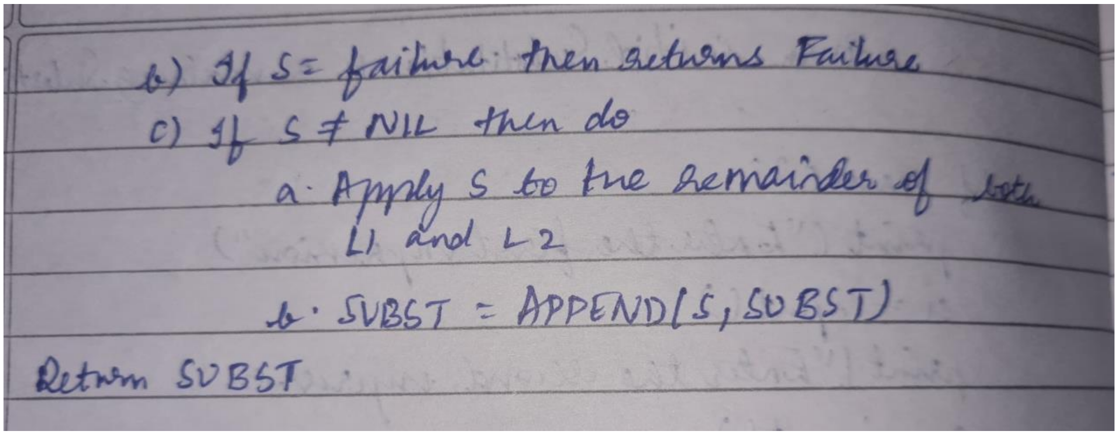
11/1/23 **Lab Program 8**

Implement unification in first order logic

Algorithm:



```
Step 1: If $\Psi_1$ or $\Psi_2$ is a variabll or constant then :
        a) If $\Psi_1$ or $\Psi_2$ are identical then return NIL
        b) else if $\Psi_1$ is a variable
              a) then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
              b) else return $\{(\Psi_2/\Psi_1)\}$
        c) else if $\Psi_2$ is a variable
              a) If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE
              b) else return $\{(\Psi_1/\Psi_2)\}$
        a) else return Failure
Step 2: If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$
        are not same, then return FAILURE.
Step 3: If $\Psi_1$ and $\Psi_2$ have a different number of
           arguments then return Failure
Step 4: Set Substitution set (SUBST) to NIL
Step 5: For i=1 to the number of elements in $\Psi_1$
        a) Call Unify function with the ith
           element of $\Psi_1$ and ith element of $\Psi_2$,
           and put the result into S
```

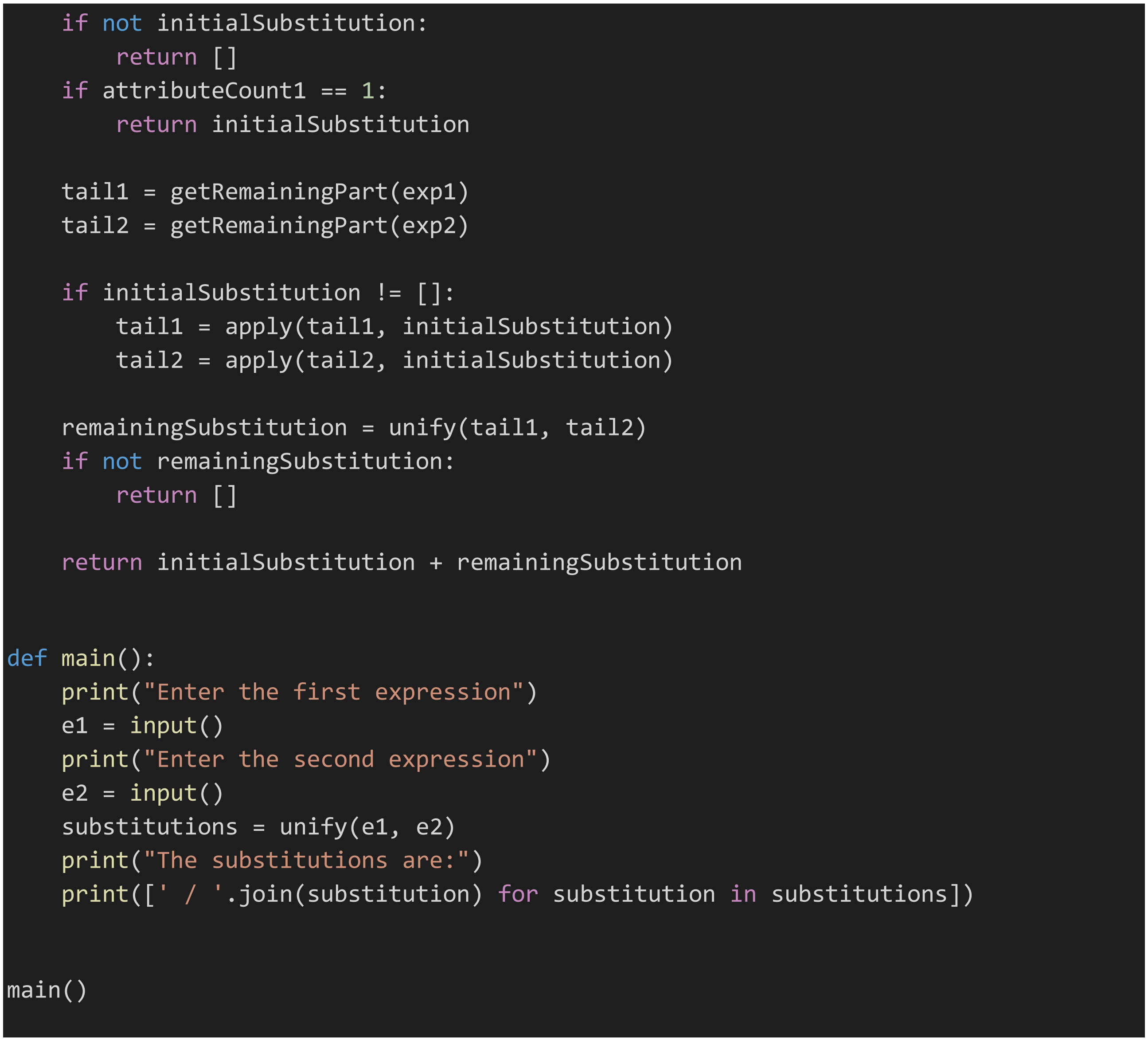


```
        b) If S = failure then returns Failure
        c) If S ≠ NIL then do
              a. Apply S to the remainder of both
                 L1 and L2
              b. SUBST = APPEND(S, SUBST)
Return SUBST
```

Code:

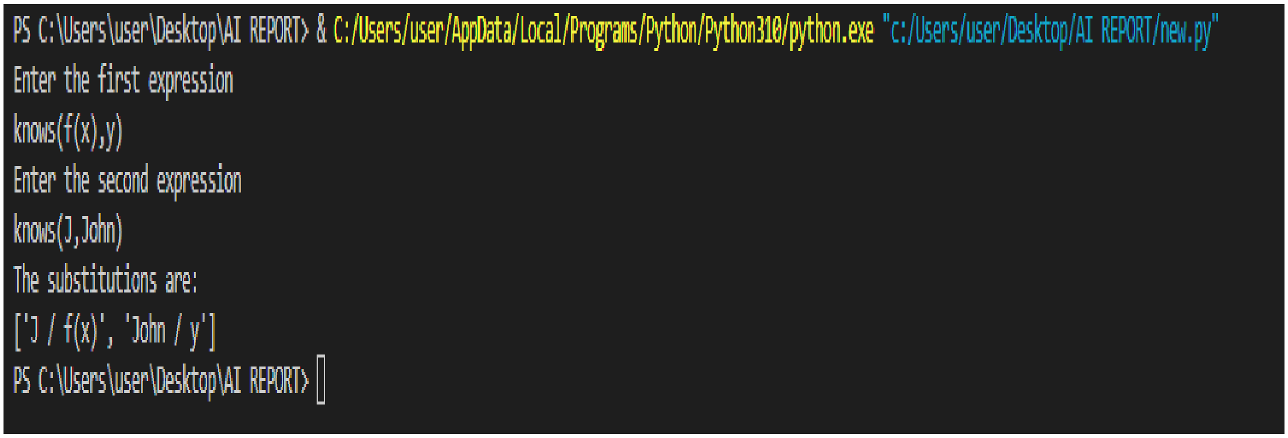
```
import re


def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes


def getInitialPredicate(expression):
    return expression.split("(")[0]


def isConstant(char):
    return char.isupper() and len(char) == 1


def isVariable(char):
    return char.islower() and len(char) == 1


def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"


def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp


def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
```

```python
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression


def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(
            f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
```



```python
    if not initialSubstitution:
        return []
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution


def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])


main()
```

Output:

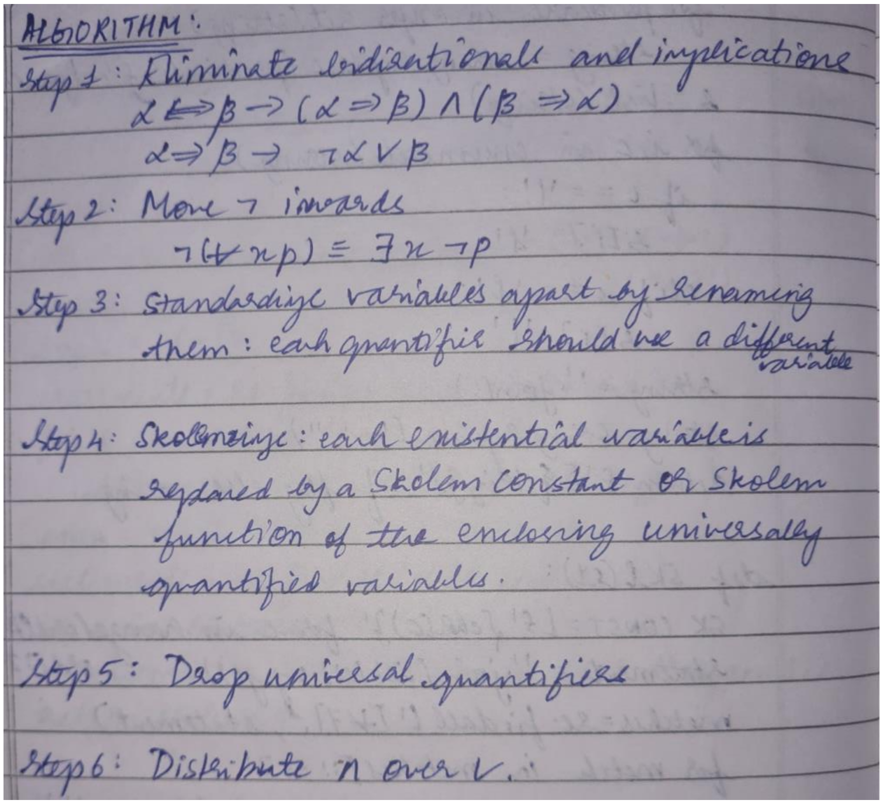


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter the first expression
knows(f(x),y)
Enter the second expression
knows(J,John)
The substitutions are:
['J / f(x)', 'John / y']
PS C:\Users\user\Desktop\AI REPORT> 
```

18/1/23 **Lab Program 9**

Convert given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



ALGORITHM:

Step 1: Eliminate bidirectionals and implications

$\alpha \Leftrightarrow \beta \rightarrow (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

$\alpha \Rightarrow \beta \rightarrow \neg \alpha \vee \beta$

Step 2: Move ¬ inwards

$\neg(\forall x p) \equiv \exists x \neg p$

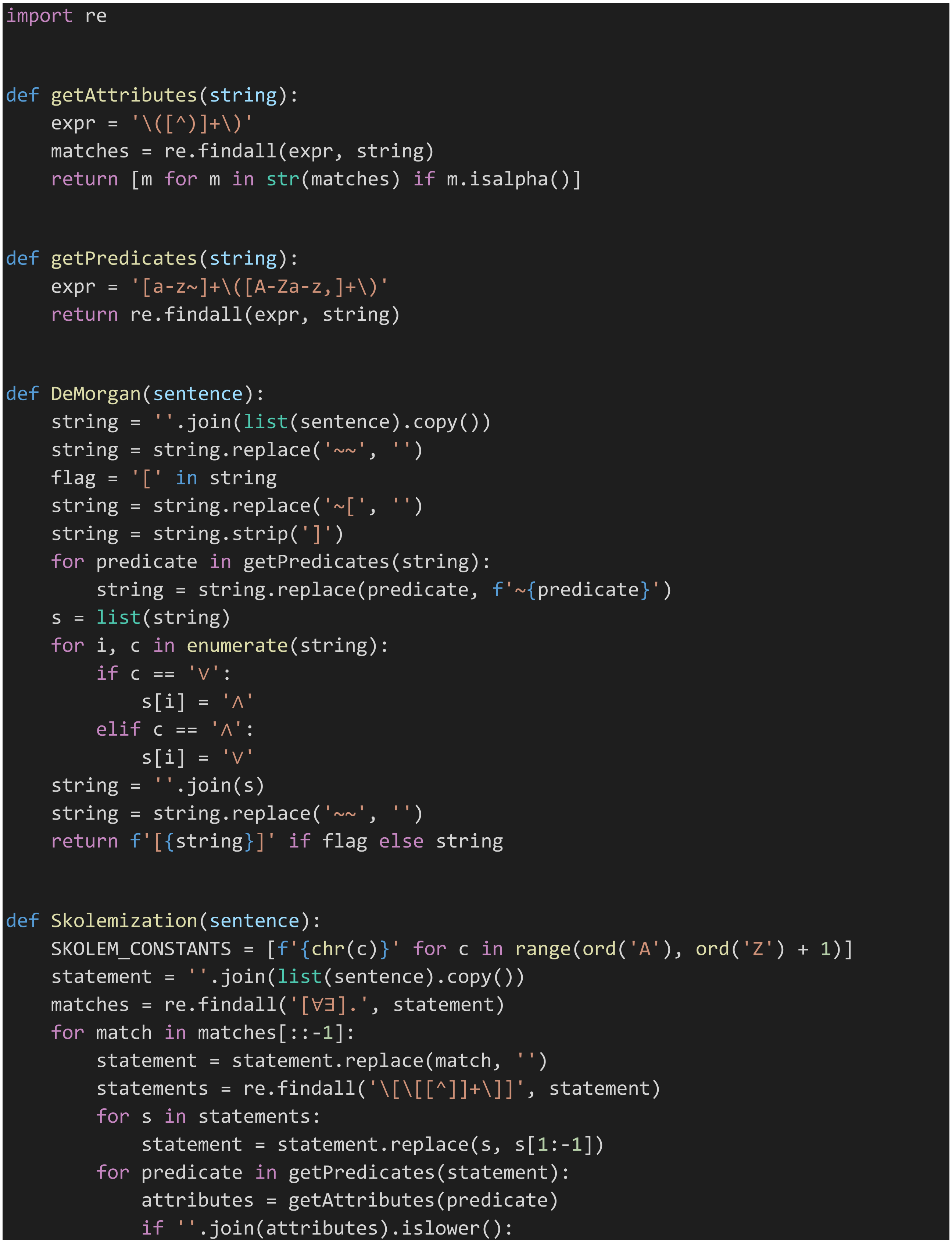
Step 3: Standardize variables apart by renaming them: each quantifier should use a different variable

Step 4: Skolemize: each existential variable is replaced by a Skolem constant or Skolem function of the enclosing universally quantified variables.

Step 5: Drop universal quantifiers

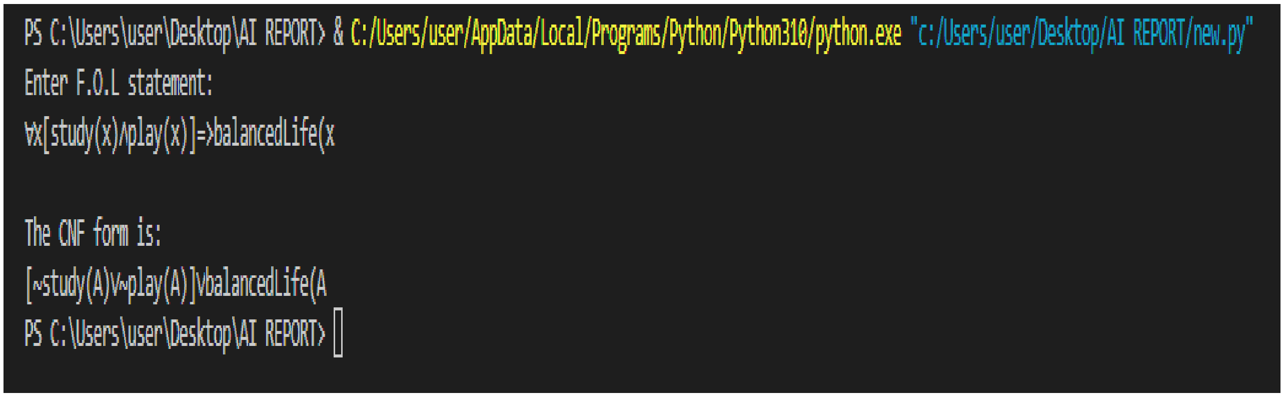
Step 6: Distribute ∧ over ∨.

Code:



```python
import re


def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]


def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)


def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '∨':
            s[i] = '∧'
        elif c == '∧':
            s[i] = '∨'
    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string


def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z') + 1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(
                    match[1], SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(
                    aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else
match[1]})')
    return statement


def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i + 1:] + ']∧[' +
statement[i + 1:] + '=>' + statement[
            :i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i + 1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i + 1], statement[i +
                                                  2] = '∃', statement[i + 2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i + 1], s[i + 2] = '∀', s[+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀', '[~∀')
    statement = statement.replace('~[∃', '[~∃')
    expr = '(~[∀|∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement


fol = input("Enter F.O.L statement:\n")
print("\nThe CNF form is:")
print(Skolemization(fol_to_cnf(fol)))
```

Output :

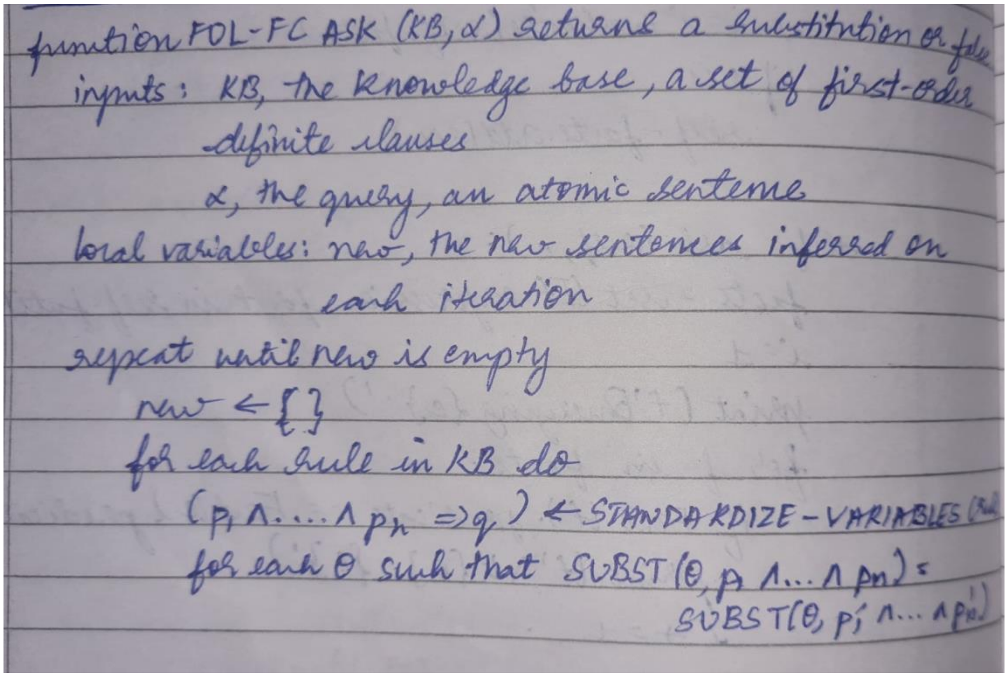


```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter F.O.L statement:
∀x[study(x)∧play(x)]=>balancedLife(x

The CNF form is:
[~study(A)∨~play(A)]∨balancedLife(A
PS C:\Users\user\Desktop\AI REPORT> 
```

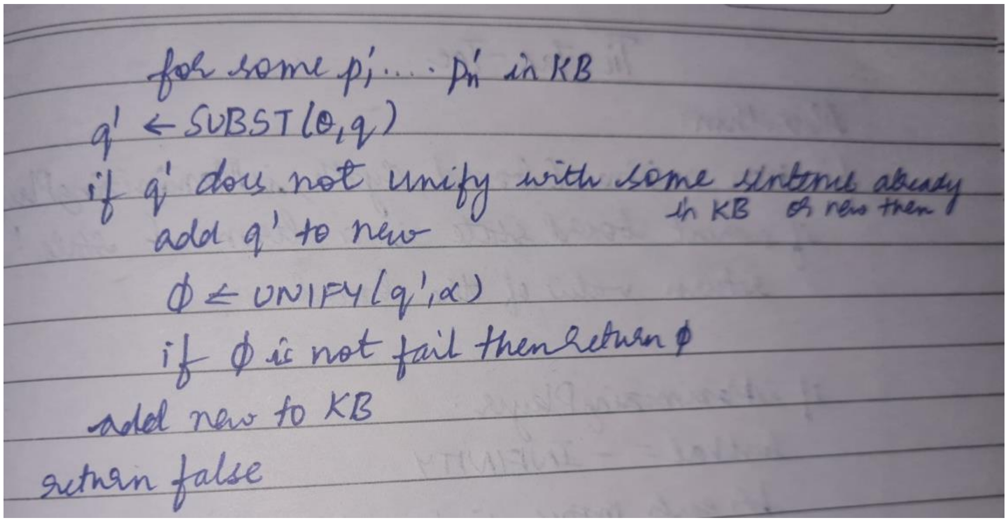
18/1/23 **Lab Program 10**

Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



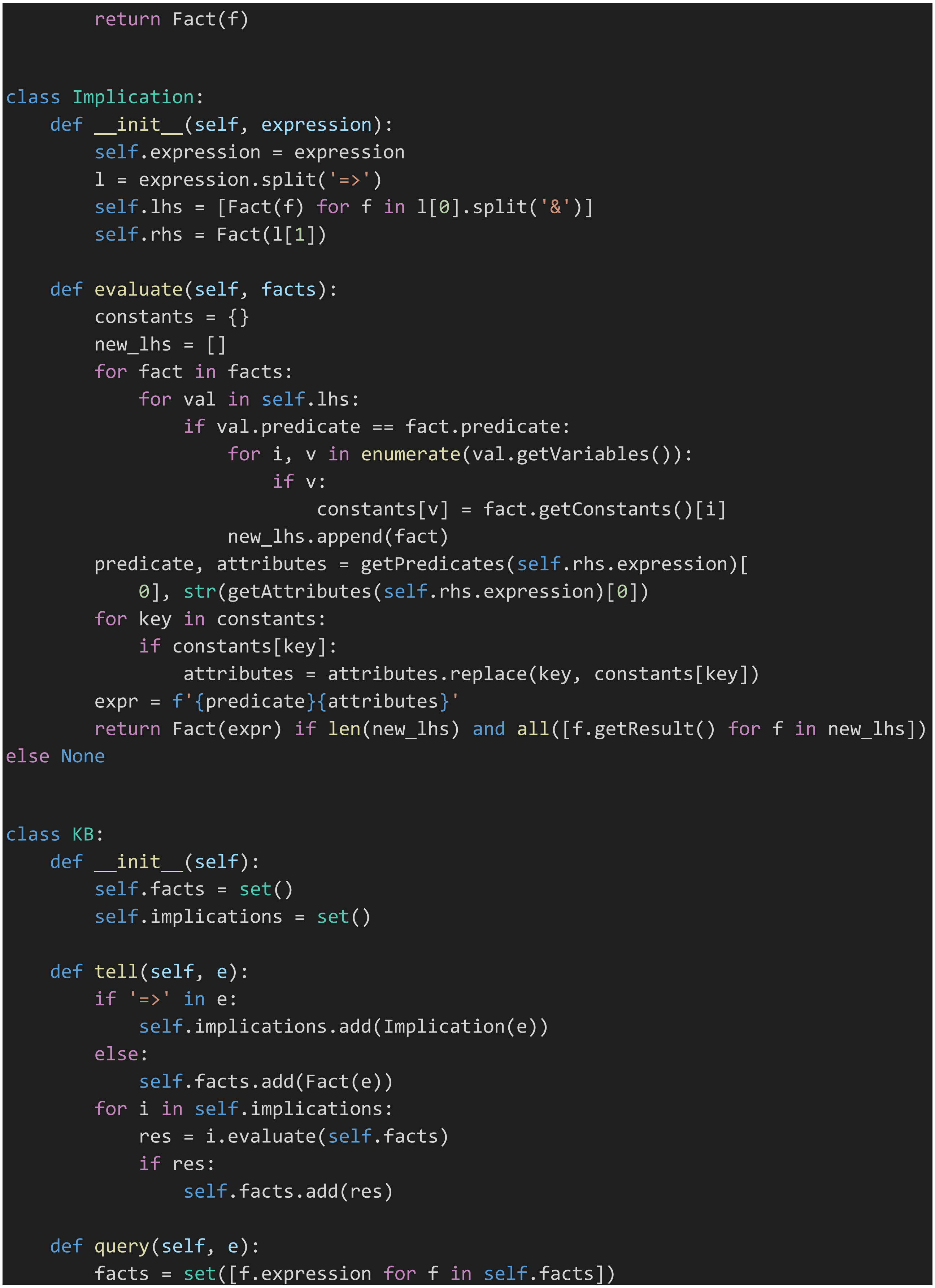
```
function FOL-FC ASK (KB, $\alpha$) returns a substitution or false
    inputs: KB, the knowledge base, a set of first-order
                definite clauses
            $\alpha$, the query, an atomic sentence
    local variables: new, the new sentences inferred on
                each iteration
    repeat until new is empty
        new ← { }
        for each rule in KB do
            ($p_1 \wedge \ldots \wedge p_n \Rightarrow q$) ← STANDARDIZE-VARIABLES (ru
            for each $\theta$ such that SUBST($\theta$, $p_1 \wedge \ldots \wedge p_n$) =
                                    SUBST($\theta$, $p_1' \wedge \ldots \wedge p_n'$)
```



```
                for some $p_1' \ldots p_n'$ in KB
            q' ← SUBST($\theta$, q)
            if q' does not unify with some sentence already
                                    in KB or new then
                add q' to new
                $\phi$ ← UNIFY(q', $\alpha$)
                if $\phi$ is not fail then return $\phi$
        add new to KB
    return false
```

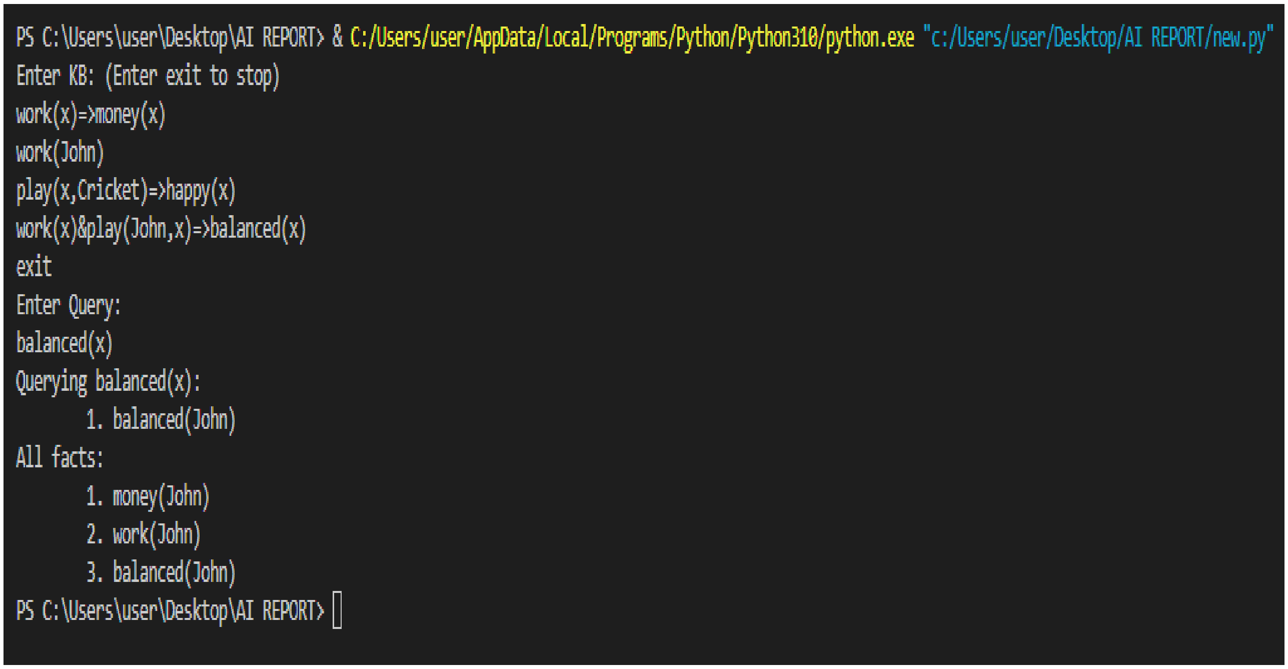
Code:

```
import re


def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()


def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches


def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)


class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p
for p in self.params])})"
```



```
        return Fact(f)


class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[
            0], str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs])
else None


class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i + 1}. {f}')


def main():
    kb = KB()
    print("Enter KB: (Enter exit to stop)")
    while True:
        t = input()
        if (t == 'exit'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()


main()
```

Output:



```
PS C:\Users\user\Desktop\AI REPORT> & C:/Users/user/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/user/Desktop/AI REPORT/new.py"
Enter KB: (Enter exit to stop)
work(x)=>money(x)
work(John)
play(x,Cricket)=>happy(x)
work(x)&play(John,x)=>balanced(x)
exit
Enter Query:
balanced(x)
Querying balanced(x):
        1. balanced(John)
All facts:
        1. money(John)
        2. work(John)
        3. balanced(John)
PS C:\Users\user\Desktop\AI REPORT> 
```