INDEX

Name: Ncha Bhaskar kamath Sub: AI LAB

Std.: V SEM Div. USN Roll No: 1BM21CS113

Telephone No. E-mail ID.

Blood Group. Birth Day.

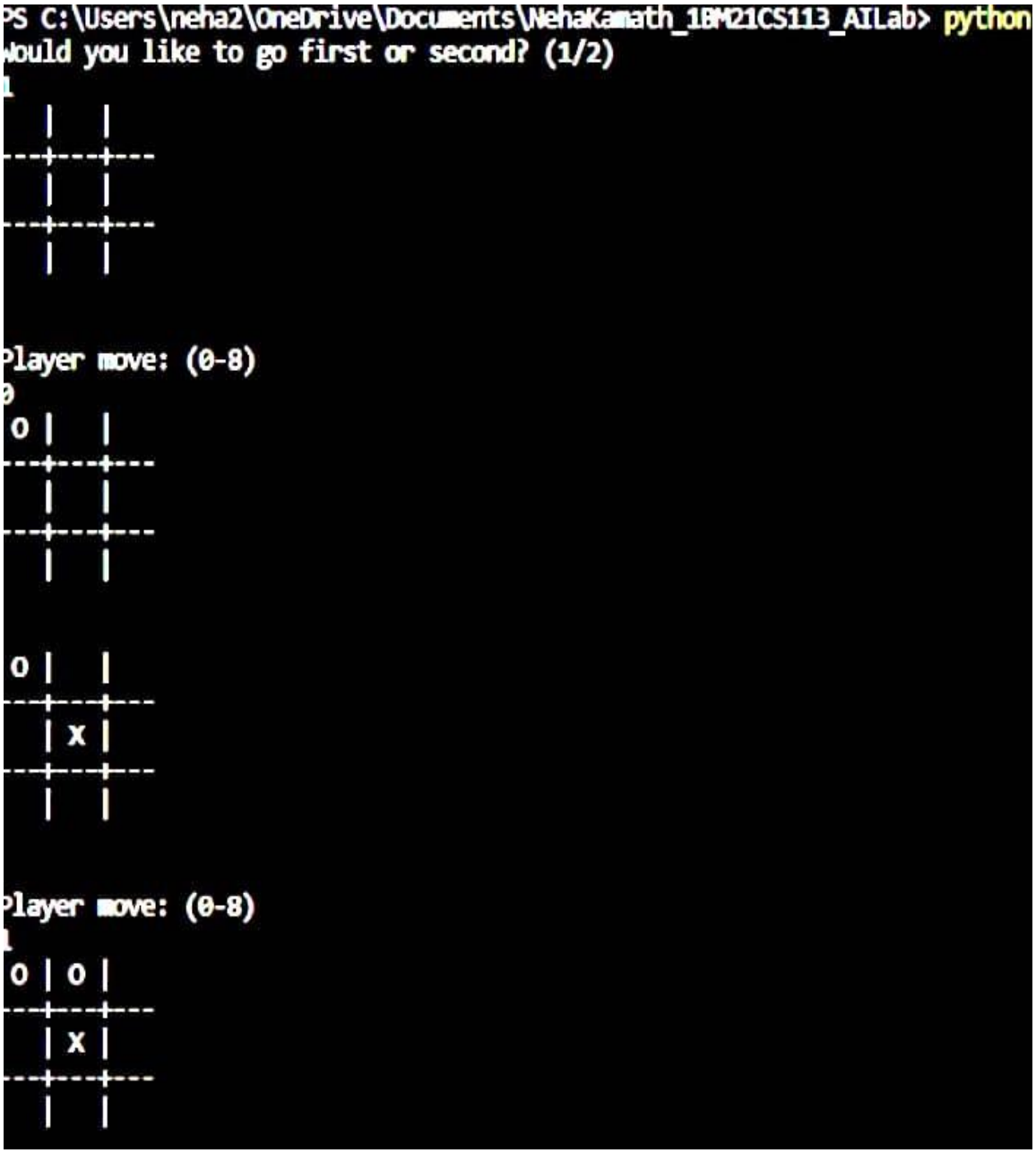
Sr.No.	Title	Page No.	Sign./Remarks
1	Python programming course on Kaggle	1	
2	Basic python programs	2-4	
3.	Tic Tac Toe game implementation	5-7.	
4.	8 Puzzle problem using BFS.	8-11	
5.	8 Puzzle problem using A* algorithm	12-15	
6.	8 Puzzle problem using ID-DFS	16-17	
7	Vacuum cleaner	18-20	
8	KB- entailment	21-22	
9.	KB - resolution	22	
10.	Unification	23-24	
11.	FOL to CNF	25-26.	
12.	Forward Chaining	27-29	

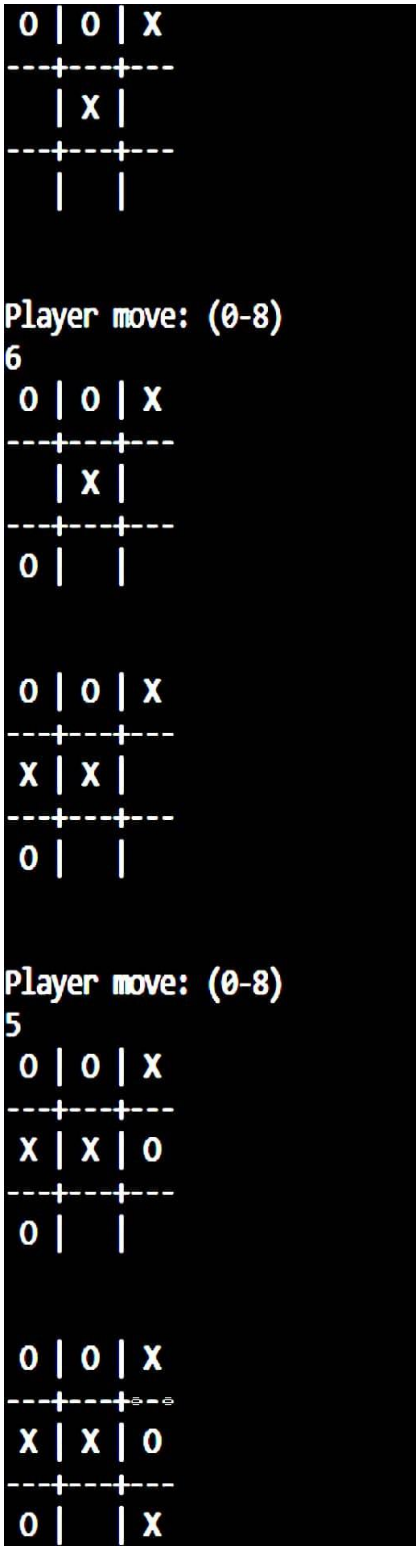
17-11-23

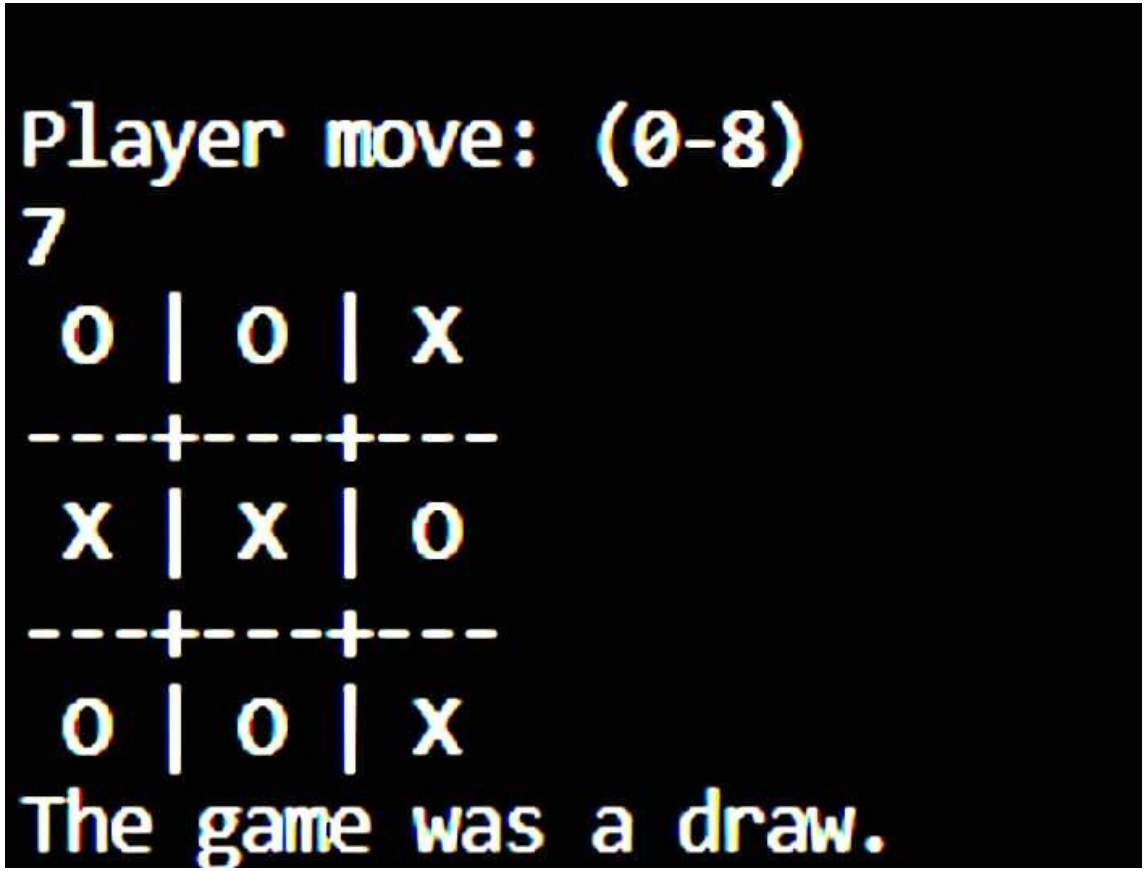
Program 1: Implement Tic Tac Toe Game

```
board = [" " for x in range(10)]
def insertLetter(letter, pos):
    board[pos] = letter

def spaceIsFree(pos):
    return board[pos] == ' '

def printBoard(board):
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')

def isWinner(bo, le):
    return (bo[1] == le and bo[2] == le and bo[3] == le)
            or
           (bo[4] == le and bo[5] == le and bo[6] == le)
            or
           (bo[7] == le and bo[8] == le and bo[9] == le)
            or
           (bo[1] == le and bo[4] == le and bo[7] == le)
            or
           (bo[2] == le and bo[5] == le and bo[8] == le)
            or
           (bo[3] == le and bo[6] == le and bo[9] == le)
            or
           (bo[1] == le and bo[5] == le and bo[9] == le)
            or
           (bo[3] == le and bo[5] == le and bo[7] == le)
```

```
def playerMove():
    run = True
    while run:
        move = int(input("Enter the position where you want to place: "))
        if(move > 0 and move < 10):
            if spaceIsFree(move):
                run = False
                insertLetter('X', move)
            else:
                print("Enter another position, this position is occupied.")
        else:
            print("Enter position within the range.")

def compMove():
    run = True
    while run:
        move = random.randrange(1,10)
        if spaceIsFree(move):
            run = False
            insertLetter('O', move)
        else:
            continue

def main():
    while (board.count(' ') <= 10):
        playerMove()
        printBoard(board)
        if(isWinner(board, 'X')):
            print("You won")
            break
        else:
            compMove()
            printBoard(board)
            if(isWinner(board, 'O')):
                print("Computer won")
                break
    else:
        print("This is a tie")
```







23-11-23 Program-2 8-Puzzle problem using breadth first search

8 puzzle problem (N puzzle problem / sliding puzzle problem) - Uninformed approach

~~N-puzzle~~

In the problem, the square will have N+1 tiles where N = 8, 15, 24 and so on.

N = 8 means the square will have 9 tiles (3 rows & 3 columns)

In this problem, ~~we have~~ initial state / initial configuration (start state) will given and we have to reach the goal state / goal configuration.

Suppose:

Initial state:

1	2	3
	4	6
7	5	8

Goal state:

1	2	3
4	5	6
7	8	

The puzzle can be solved by moving the tiles one by one in the single empty space & thus achieve the goal state.

Rules

- The empty can only move in 4 directions:
  1) up 2) down 3) right 4) left
- It cannot move diagonally
- Can take only one step at a time.

0	x	0
x	#	x
0	x	0

Tiles at 0 — no. of possible moves = 2.
Tiles at x — no of possible moves = 3
Tiles at # — no of possible moves = 4.

This problem can be solved using breadth first search approach:

↳ Uninformed / non-heuristic search

- This approach explores all nodes (does not use intelligence).

Complexity: $O(b^d)$ where
b - branching factor
d - depth factor

Worst case — 3^{20}.

For 8 puzzle problem.

Branching factor b = $\frac{\text{all possible moves of empty tile at each position}}{\text{no of tiles}}$

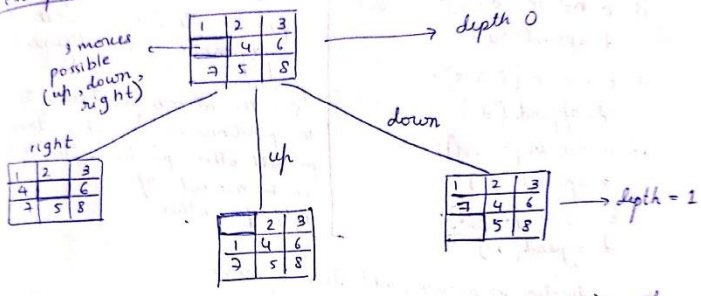
$= \frac{24}{9} = 2.67 \sim 3$

depth factor - initially will be 0

As we explore the nodes, the depth factor increases.

Every time, we check the possible moves of the empty tile, branch it, consider all possible cases and finally to the goal state.

For example



We continue branching these nodes.

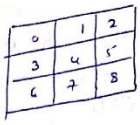
Since this is breadth first search, branch for possibilities of 'right' node, then 'up' node, then 'down' node and then only move to branch in the next level.

Code

```
import numpy as np
import pandas as pd
import os

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
    poss_moves_to_do = []
    poss_moves_to_do = possible_moves(source, exp)
    for move in poss_moves_to_do:
        if move not in exp and move not in queue:
            queue.append(move)
```

Same BFS code

```
def possible_moves(state, visited_states):
    # index of empty spot
    b = state.index(0)

    # directions array
    d = []
    # add all the possible directions
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')

    # if direction is possible, add state to move
    pos_moves_it_can = []
    # for all possible directions, find the state if that move is played
    #  by calling 'gen' function
    # Now for each direction, I want to generate different states of the matrix
    for i in d:
        pos_moves_it_can.append(gen(state, i, b))   # which I append in pos_moves_it_can

    return [move_it_can for move_it_can in pos_moves_it_can if
            move_it_can not in visited_states]

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

src = [1,2,3,4,5,6,0,7,8]
target = [1,2,3,4,5,6,7,8,0]
bfs(src, target)
```



This is to find the possible directions when the empty node is at different positions.
See the above matrix. If the positions are not 0, 1, 2, then for all other positions, the tile can be moved up.
Similarly for others.

To move down, I shift 3 indices front.
If I am at position 5, and I want to move down, I do 5+3 = 8.

24-11-21

Output

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

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

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

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

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

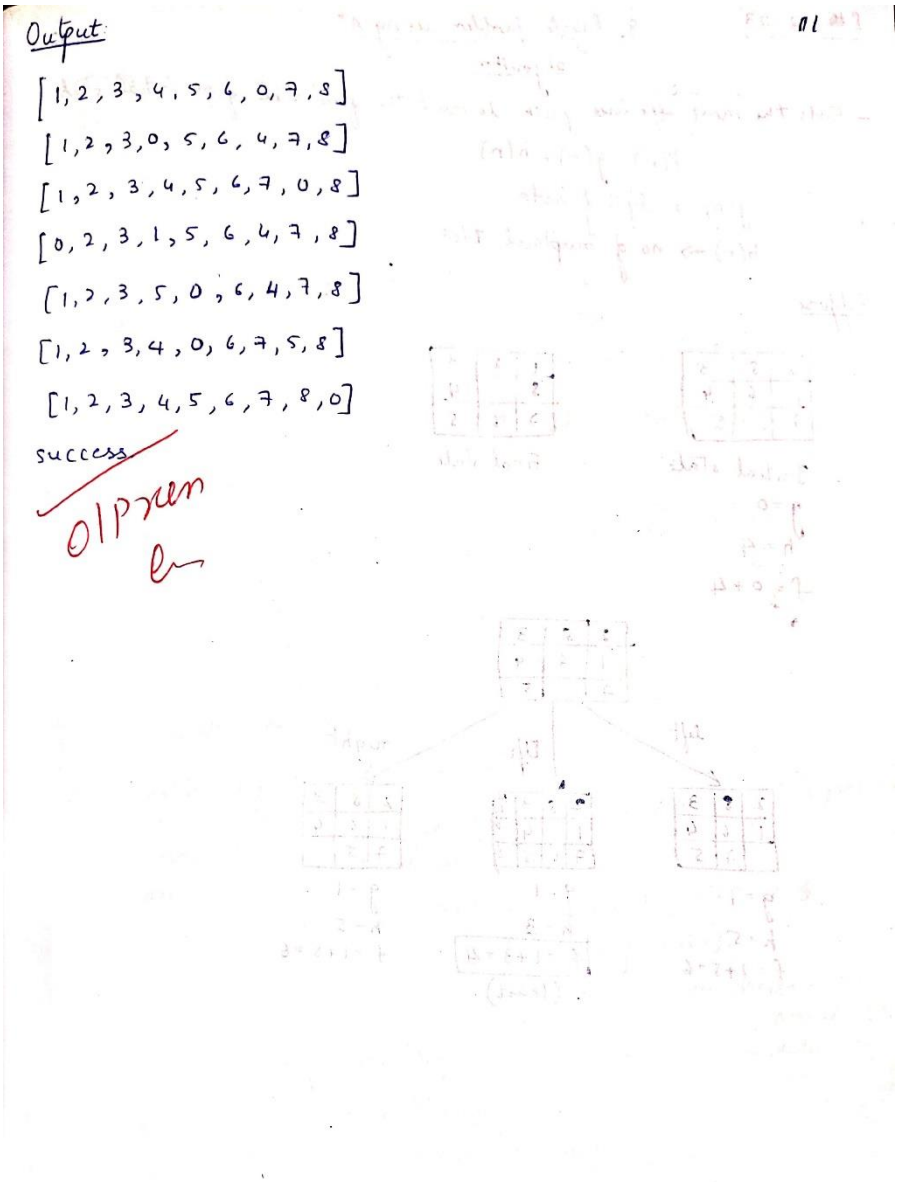
[1, 2, 3, 4, 0, 6, 7, 5, 8]

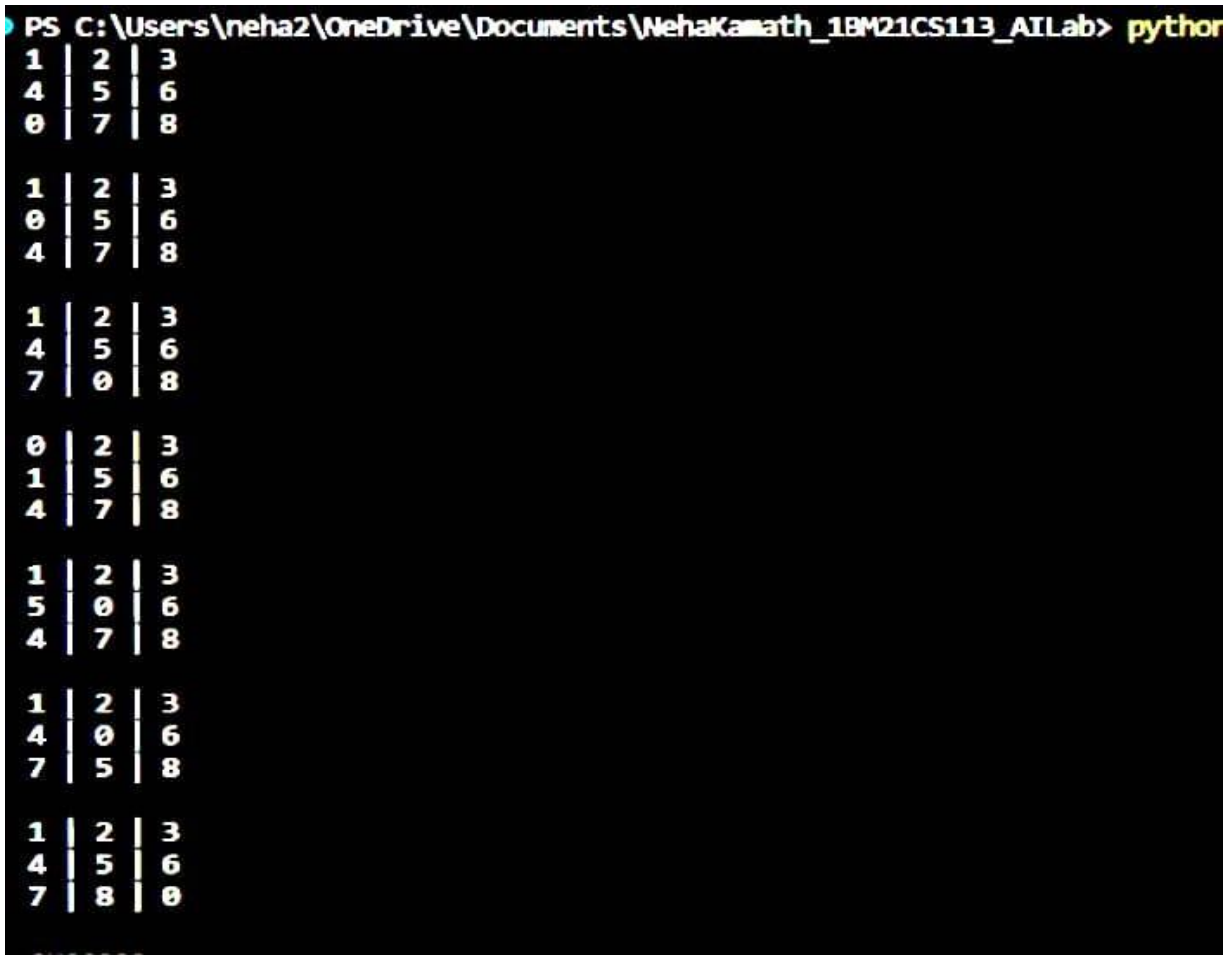
[1, 2, 3, 4, 5, 6, 7, 8, 0]

success

Olpxen

e





8-12-23

8-Puzzle problem using A* algorithm

- Finds the most cost-effective path to reach the final state from initial state.

$f(n) = g(n) + h(n)$

$g(n) \rightarrow$ depth of node

$h(n) \rightarrow$ no of misplaced tiles.

Suppose:

2	8	3
1	6	4
7		5

Initial state

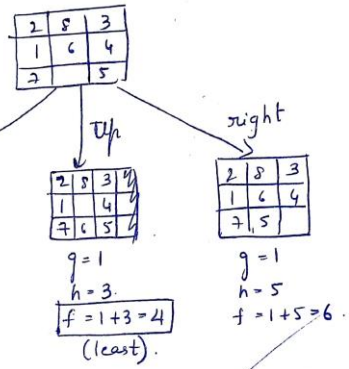
$g = 0$

$h = 4$

$f = 0 + 4$

1	2	3
8		4
7	6	5

Final state



Code

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):  # to generate child nodes from the given node by moving blank space either in the 4 directions.
        x, y = self.find(self.data, '_')  # "find" to find position of blank space
        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 & if the pos. values are out of limits, then return None"""
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):  # checks if new position (x2, y2) is within the bounds of puzzle
            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):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp
```

```
0 1 2
0 1 2 3
1 4 5 6
2 7 8 9
```

Two lists are maintained.

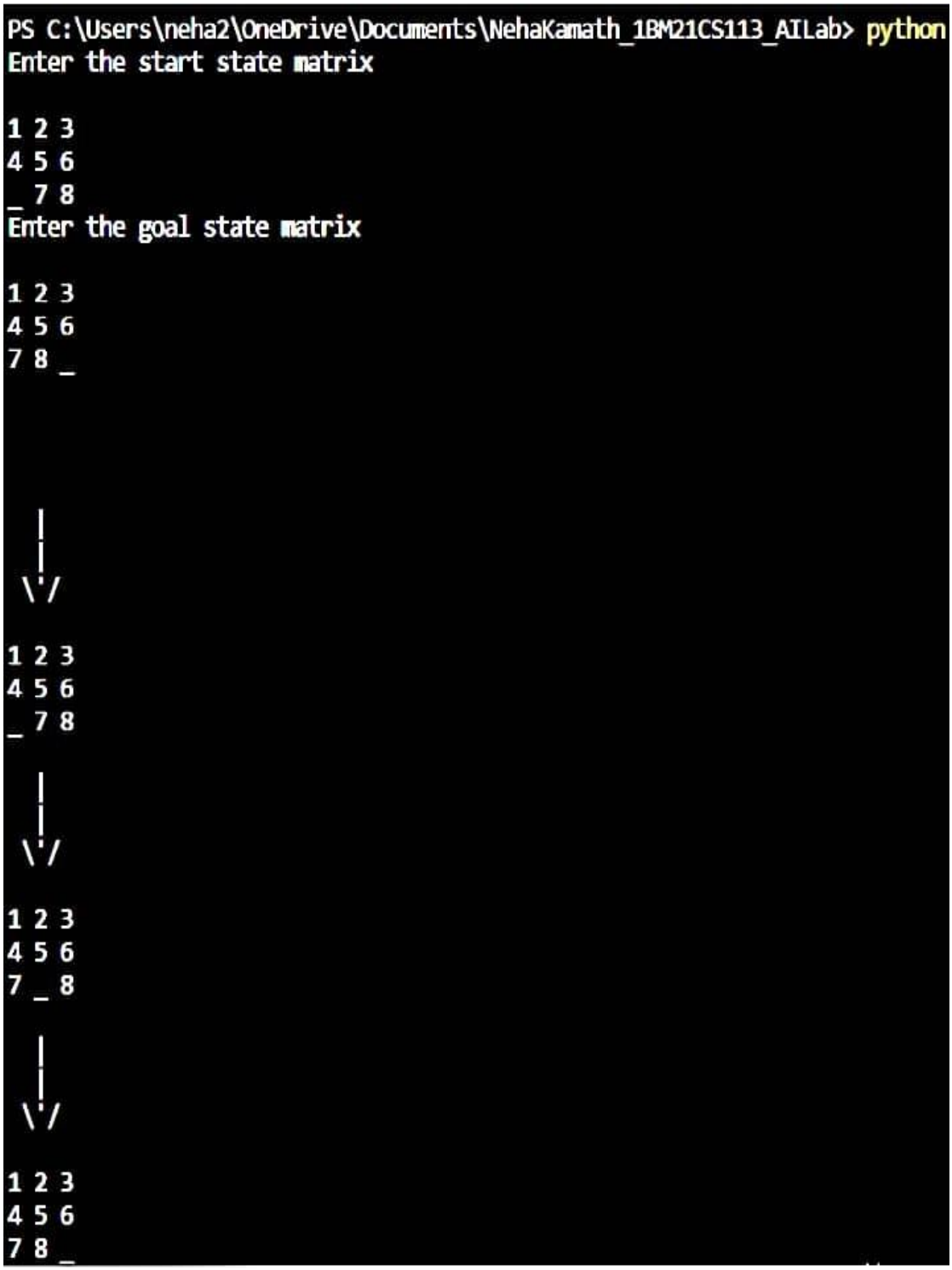
open list → contains all the nodes that are generated & not existing in closed list

closed list – each node ~~explored~~ explored after its neighbouring nodes are discarded

So after expanding a node, it is pushed into the closed ~~state~~ list and the newly generated states are pushed in open list.

```
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 &
          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 func to calculate heuristic value
            f(x) = h(x) + g(x) """
        return self.h(start, data, goal) + start.level

    def h(self, start, goal):
        """ Calculates the difference b/w 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 & goal puzzle state"""
    print("Enter start state matrix :\n")
    start = self.accept()
    print("Enter 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/\n")
        for i in cur.data:
            for j in i:
                print(j, end=" ")
            print("")
        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()
```



8-12-23

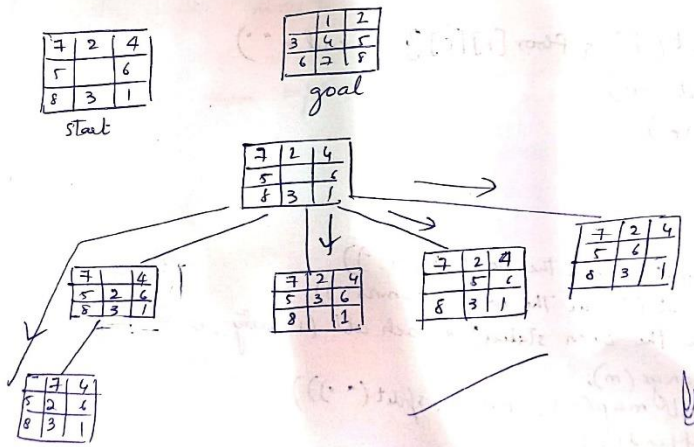
8-Puzzle problem using ID-DFS:

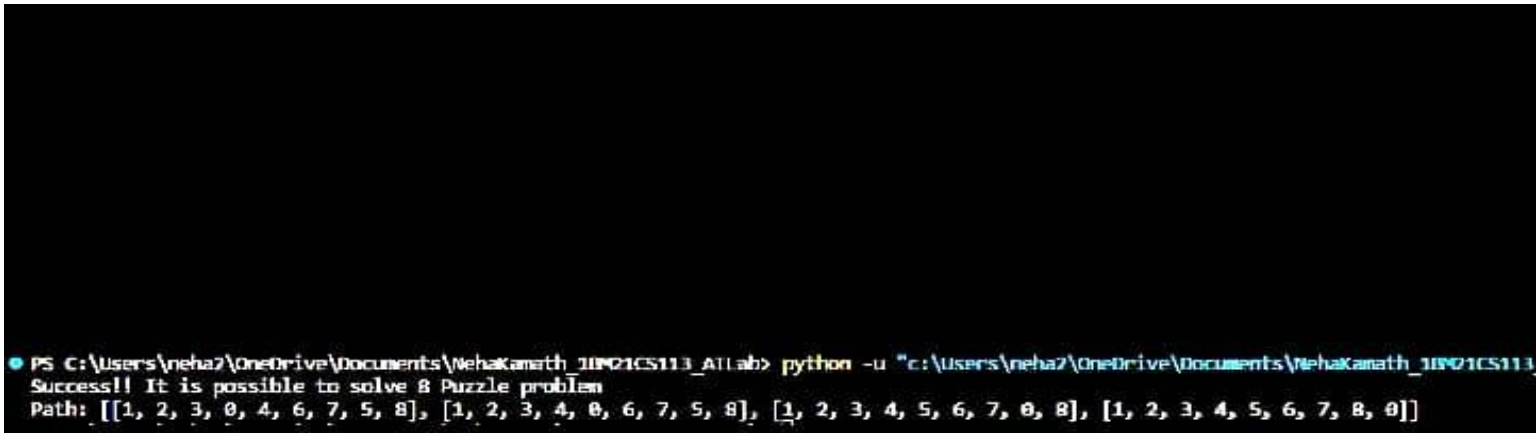
Code:

```
def id-dfs(puzzle, goal, get-moves):
    import itertools
# get-moves → possible-moves
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get-moves(route[-1]):
            if move not in route:
                next-route = dfs(route + [move], depth-1)
                if next-route:
                    return next-route

    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route

def possible-moves(state):
    b = state.index(0)
    d = []
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')

    pos-moves = []
    for i in d:
        pos-moves.append(generate(state, i, b))
return pos-moves
```

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

initial = [1,2,3,0,4,6,7,5,8]
goal = [1,2,3,4,5,6,7,8,0]
route = id_dfs(initial, goal, possible_moves)
if route:
    print("Success!!")
    print("Path:", route)
else:
    print("Failed to find a solution")
```

ID_DFS - Combination of BFS and DFS
- DFS in BFS manner





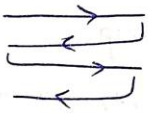
22-12-23

Vacuum cleaner Agent:

Code: (One room).

```
def clean(floor):
    i, j, row, col = 0, 0, len(floor), len(floor[0])
    for i in range(row):
        if (i%2 == 0):
            for j in range(col):
                if(floor[i][j] == 1):
                    print_F(floor, i, j)
                    floor[i][j] = 0
                    print_F(floor, i, j)
        else:
            for j in range(col-1, -1, -1):
                if(floor[i][j] == 1):
                    print_F(floor, i, j)
                    floor[i][j] = 0
                print_F(floor, i, j)
```

// The vacuum cleaner cannot move diagonally or jump to positions directly. So for all even rows, the vacuum cleaner moves from left to right and for all odd rows, it moves from right to left, as shown below.



If a grid in a room is dirty (i.e floor[i][j] then print the grid and clean it by setting it to 0.

```
def print_F(floor, row, col):
    print("The floor matrix is as below:")
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r == row and c == col:
                print(f">{floor[r][c]}<", end = "")
            else:
                print(f" {floor[r][c]} ", end = "")
        print(end = '\n')
    print(end = '\n')

def main():
    floor = []
    m = int(input("Enter the no. of rows: "))
    n = int(input("Enter the no. of columns: "))
    print("Enter the clean status of each cell (1-dirty, 0-clean)")
    for i in range(m):
        f = list(map(int, input().split(" ")))
        floor.append(f)
    print()
    clean(floor).
```

Two rooms:

Logic:

In the main function, take input for 2 rooms.

~~If room1~~

Initially, start from room 1 and inspect every grid.

If room 1 is clean i.e

[room1] = 0 , move to room 2 and clean it.

Room 1

0	1	1
0	0	1
1	0	0

Room 2

0	1	0
0	0	0
0	0	1

Room 1 → dirty → so clean it.

Check if room 1 is completely clean (ie all grids are 0)

if clean, check if room 2 is already clean, else move to room 2.

If room 2 → dirty

Clean it by calling 'clean' function

If completely clean, return True and ~~move to room 1~~

~~and move to~~ check if room 1 is clean, else move to room 1, else

if room 1 also is completely clean, return true and exit.

Room 1
Status: clean ——move to——→ Room 2.

(Instead of grid, ~~take~~ implement this as just one grid in a room)

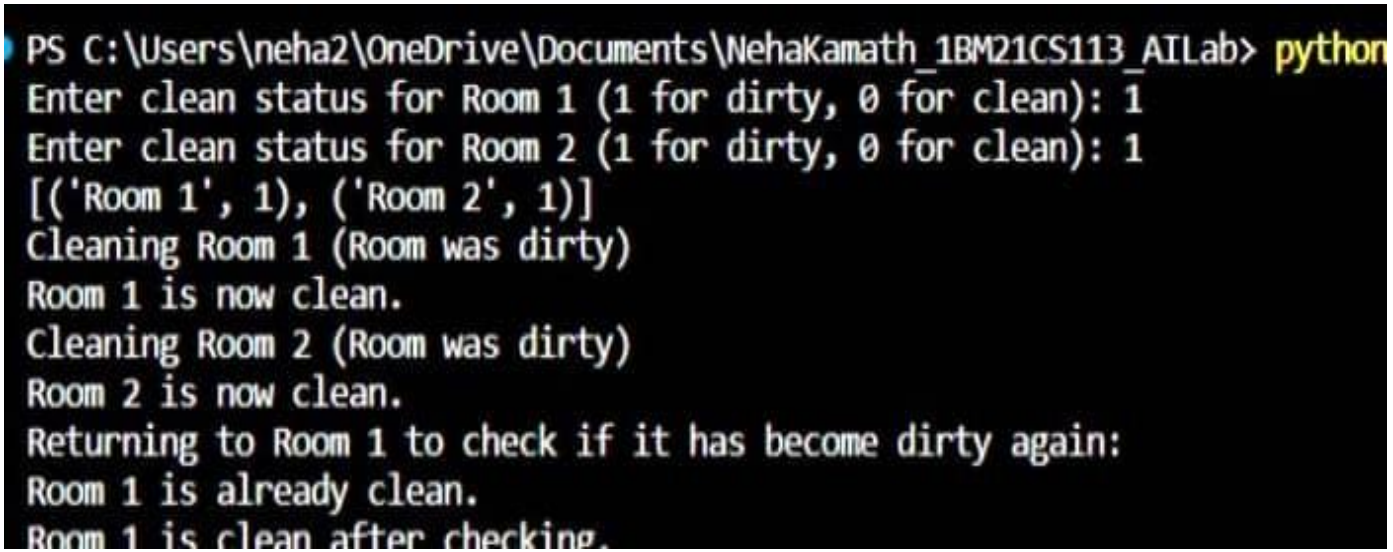
2. Code for 2 rooms:

```
def clean_room(room_name, is_dirty):
    if is_dirty:
        print(f"Cleaning {room_name} (Room was dirty).")
        print(f"{room_name} is now clean.")
        return 0
    else:
        print(f"{room_name} is already clean.")
        return 0

def main():
    rooms = ["Room 1", "Room 2"]
    room_statuses = []

    for room in rooms:
        status = int(input(f"Enter the clean status for {room} (1 for dirty, 0 for clean): "))
        room_statuses.append((room, status))

    for i, (room, status) in enumerate(room_statuses):
        room_statuses[i] = (room, clean_room(room, status))

    print(f"Returning to {Room[0]} to check if it has become dirty again.")
    room_statuses[0] = (rooms[0], clean_room(rooms[0], room_statuses[0][1]))
    print(f"{rooms[0]} is {'dirty' if room_statuses[0][1] else 'clean'} after checking.")

if __name__ == "__main__":
    main()
```

Output:

```
Enter clean status for room 1 : 1
Enter clean status for room 2 : 0
[('Room 1', 1), ('Room 2', 0)]
Cleaning Room 1 (Room was dirty)
Room 1 is now clean.
Room 2 is already clean.
Returning to room 1 to check if it has become dirty again:
Room 1 is already clean.
Room 1 is clean after checking.
```

12/12/23



29-12-23

KB - Entailment

```
from sympy import symbols, And, Not, Implies, satisfiable
def create_KB():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')
    KB = And(Implies(p, q), Implies(q, r), Not(r))
    return KB
def query_entails(KB, query):
    en = satisfiable(And(KB, Not(query)))
    return not en
if __name__ == "__main__":
    kb = create_KB()
    query = symbols('p')
    result = query_entails(kb, query)
print("KB:", kb)
print("Query:", query)
print("result")
```

(KB ∧ ¬ query)

			a	b	c	d	e	
p	q	r	p → q	q → r	¬ r	a ∧ b ∧ c	¬ p	d ∧ e
T	T	T	T	T	F	F	F	F
T	T	F	T	F	T	F	F	F
T	F	T	F	T	F	F	F	F
T	F	F	F	T	T	F	T	F
F	T	T	T	T	F	F	T	F
F	T	F	T	F	T	F	T	F
F	F	T	T	T	F	F	T	T
F	F	F	T	T	T	T	T	T

∴ satisfiable

∴ ~~Not Entail~~ KB does not entail α.

$\alpha \models \beta$ iff in every model where α is true, β is true.

$\alpha \vdash \beta$ if $(\alpha \wedge \neg \beta)$ is unsatisfiable.

KB: $\neg r \wedge (p \to q) \wedge (q \to r)$

query: p.

~~If A comes do~~

1) Jim rides a bike to school every morning

2) Jim is good at riding bikes

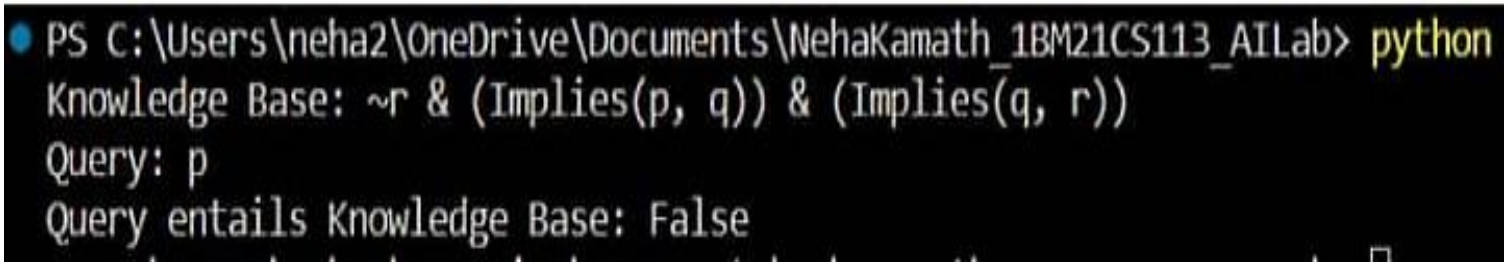
(1) does not entail (2).

Output:

KB: ~r & (Implies(p, q) & Implies(q, r))

query: P

Result: False.



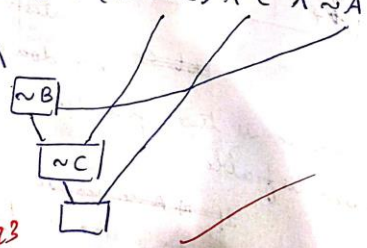
8

KB - resolution

```
def negate_literal(literal):
    if literal[0] == '~':
        return literal[1:]
    else
        return '~' + literal

def resolve(c1, c2):
    resolved_clause = set(c1) | set(c2)
    for literal in c1:
        if negate_literal(literal) in c2:
            resolved_clause.remove(literal)
            ——————————— (negate_literal(literal))
    return tuple(resolved_clause)

def resolution(KB)
    new_clauses = set()
    for i, c1 in enumerate(KB)
        for j, c2 ——— (KB)
        if i != j:
            new_clause = resolve(c1, c2)
            if len(new_clause) > 0 & new_clause not in
                new_clauses.add(new_clause)
```

Output:

KB: $(A \vee \sim B) \wedge (B \vee \sim C) \wedge C \wedge \sim A$



29/12/23

```
79    rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
80    goal = 'R'
81    main(rules, goal)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\neha2\OneDrive\Documents\NehaKamath_1BM21CS113_AILab> python -u "c:\Us

Step    |Clause |Derivation
-------------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
PS C:\Users\neha2\OneDrive\Documents\NehaKamath_1BM21CS113_AILab>
```

Code

```
import re
def getAttributes(expression):
    expression = expression.split(",")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    ____ = re.split("(?<!\(.),(?!.\))", expression)
    return expression

def unify(exp1, exp2):
    if exp1 == exp2: #Checks the entire expression
        return[]
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp1):
        return [(exp1, exp2)]  # returns substitution list.
    if isConstant(exp2):
        return ([exp2, exp1)].
    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
        else
            return [(exp2, exp1)]
    if isVariable(exp2):
        if checkOccurs(exp2, exp1):
            return False
        else return[(exp1, exp2)]
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates dont match!");
        return False
```

hears(John, x) —①
hears(John, Jane). —②

knows(John, x)
knows(Jane, y).

```
attributeCount1 = len(getAttributes(exp1))
--------- 2 = ---------- 2
if attributeCount1 != attributeCount2:
    return False

head1 = getFirstPart(exp1)
head2 = ----------(exp2)
initialSubstitution = unify(head1, head2)
if not initialSubstitution:
    return False
if attributeCount1 == 1:
    return initialSubstitution

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

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

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

initialSubstitution.extend(remainingSubstitution)
return initialSubstitution

exp1 = "knows(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

Output:

```
[('X', 'Richard')]
```

```
exp1 = "knows(A, x)"
exp2 = "knows(y, mother(y))"
```

Output: Substitutions:

```
[('A', 'y'), ('mother(y)', 'x')]
```

Unification — making 2 expressions look identical.

Conditions:

- The predicate should be the same
- The no. of arguments in both the expressions must be the same.
- If a similar variables are present in the same exp, then unification fails.

$P(x, F(y))$ — ①

$P(a, F(g(z)))$ — ②

$[a/x]$ (x with a)

$P(a, F(y))$, $P(a, F(g(z)))$ $[g(z)/y]$

$P(a, F(g(z)))$, $P(a, F(g(z)))$

```
107    exp1 = "knows(A,x)"
108    exp2 = "knows(y,Y)"
109    substitutions = unify(exp1, exp2)
110    print("Substitutions:")
111    print(substitutions)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\neha2\OneDrive\Documents\NehaKamath_1BM21CS113_AILab> python
Substitutions:
[('A', 'y'), ('Y', 'x')]
```

19-1-24

FOL to CNF

Create a list of skolem constants

```
import re
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 = 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 statements:
        i = statement.index('-')
        br = ---------- ('[' if '[' in statement
        new_statement = '~' + statement[br:i] + ']' + 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(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))
exps = '~\[[^]]+\]'
statements = re.findall(exps, statement)
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement
```

```
print(Skolemization(fol_to_cnf("animal(y) <=> loves(x,y)")))
```

("∀x [∀y [animal(y) => loves(x,y)] => [∃z [loves(z,x)]]]")

("[american(x) & weapon(y) & sells(x,y,z) & hostile(z)] => criminal(x)")

Output:

[~animal(y) | loves(x,y)] & [~loves(x,y) | animal(y)]

[animal(G(x)) & ~loves(x, G(x))] | [loves(F(x), x)]

[~american(x) | ~weapon(y) | ~sells(x,y,z) | ~hostile(z)] | ~~hostile(z)~~ criminal(x).

Explanation:

$\forall x \; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$.

$A \Leftrightarrow B$

↳ Replace with $(A \Rightarrow B) \wedge (B \Rightarrow A)$

$A \Rightarrow B$

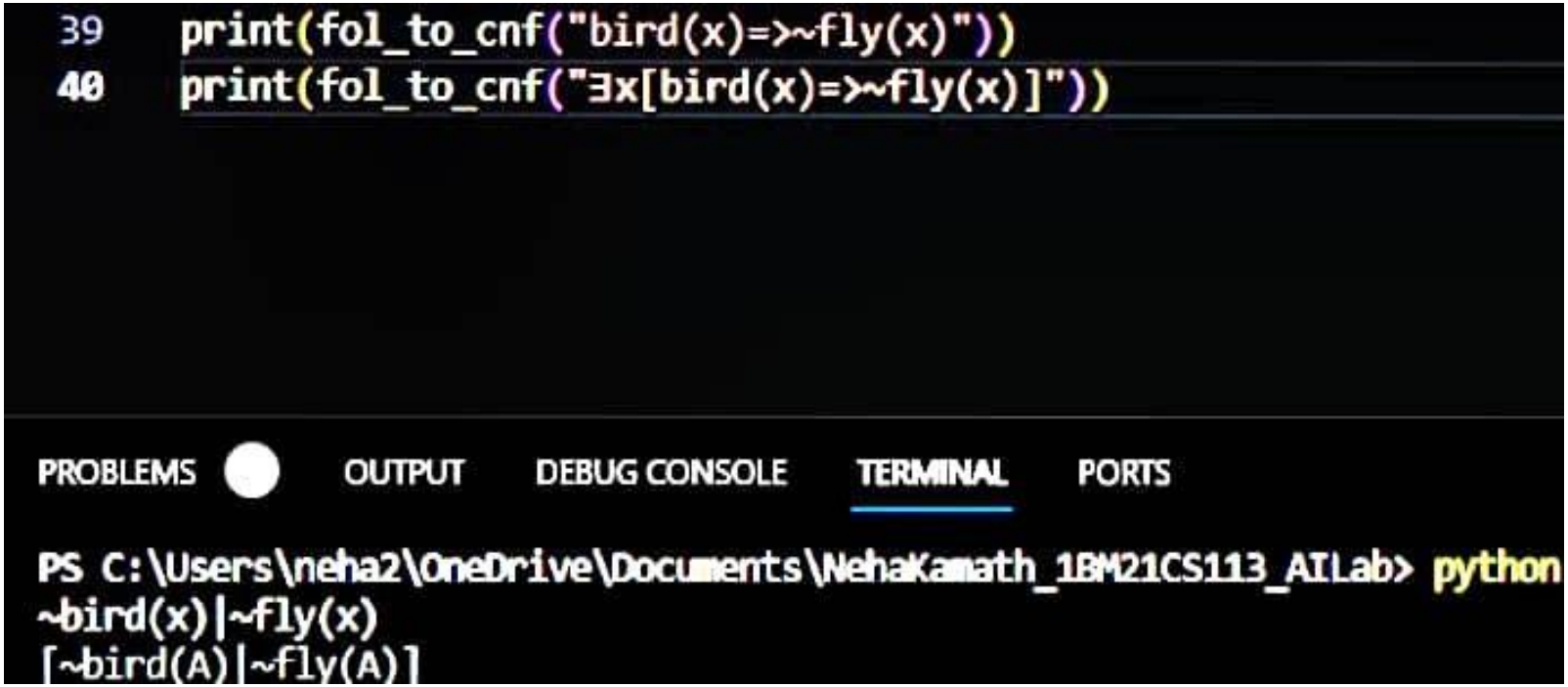
↳ $\sim A \vee B$

$\sim[A] \rightarrow$ demorgan

$\sim [King(Richard) \wedge Greedy(Richard)] \vee Evil(Richard)$

$\sim King(Richard) \vee \sim Greedy(Richard) \vee Evil(Richard)$

19/11/24



```
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
        ~~constant~~ 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 {c}:')
for f in facts:
    if Fact(f).predicate == Fact(c).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}')
```

```
kb = KB()
kb.tell('missile(x) ⇒ weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x, America) ⇒ hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono, America)')
kb.tell('owns(Nono, M1)')
kb.tell('missile(x) & owns(Nono, x) ⇒ sells(West, x, Nono)')
kb.tell('american(x) & weapon(y) & sells(x,y,z) &
        hostile(z) ⇒ criminal(x)')

kb.query('criminal(x)')
kb.display()
```

Output:

```
Querying criminal(x):
    1) criminal(West)
All facts:
    1. american(West)
    2. Sells(West, M1, Nono)
    3. Missile(M1)
    4. enemy(Nono, America)
    5. criminal(West)
    6. weapon(M1)
    7. owns(Nono, M1)
    8. hostile(Nono).
```

Forward chaining:
Starts with the base state and uses the inference rules and available knowledge in the forward direction till it reaches the end state. The process is iterated till the final state is reached

$A \wedge B \Rightarrow C$

A

B

Query : C

Snake
24/1/24