INDEX

Batch- 3.

Name Mahavir Nahata

Standard 5th sem Section B Roll No. 1BM21CS100

Subject ~~IOT~~ AI Lab. (Batch-3).

Sl.No.	Date	Title	Page No	Teacher's Sign
01	9/11/23	Basic python program.		17.11.23
02	17/11/23	Tic-Tac Toe game.		
03	24/11/23	8-puzzle search using BFS		
04	08/12/23	8-puzzle using A* search & ID DFS		8/12/2023
~~05~~				
05	22/12/23	Vacuum cleaner		22/12/23
06	29/12/23	Knowledge-Base entailment		29/12/23
07	29/12/23	Knowledge-Base Resolution		29/12/23
08	19/01/24	Unification program.		19/1/24
09	19/01/24	FOL to CNF		19/1/24
10	19/1/24	Forward reasoning.		19/1/24

① Age program:

```
age = int (input ("Enter age: "))
if age <= 12 :
    print("child")
elif age > 12 and age <= 17 :
    print(" Teenager")
elif age >= 18 and age < 65 :
    print ("Adults")
else :
    print ("Older Adults")
```

output :

Enter the age : 12
child .

② Multiplication table :

```
num = int (input ("Enter no:"))
for i in range (1, 11, 1)
    print (num, " * ", i, " =", (num * i))
```

Output :

Enter no : 2

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20

③ Pattern 1 : (1, 22, 333, ...)

```
num = int(input("Enter no: "))
for x in range(num)
    n = x+1
    for y in range (x+1)
        print(n, end = " ")
    print("\n")
```

output:

```
1
2 2
3 3 3
```

④ Pattern 2 (1, 12, 123, ...)

```
num = int(input("Enter no: "))
for x in range(num)
    n = 1
    for y in range(x+1)
        print(n, end = " ")
        n = n+1
    print("\n")
```

output:

```
1
1 2
1 2 3
```

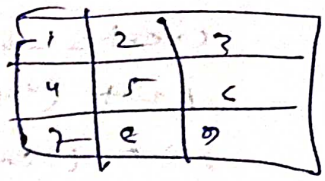
⑤ Integer reversal:

```
num = int(input("Enter the int:"))
numrev = 0
while(num>0)
    numrev = numrev*10 + num%.10
    num = num//10
print(n num)
```

17/1/25

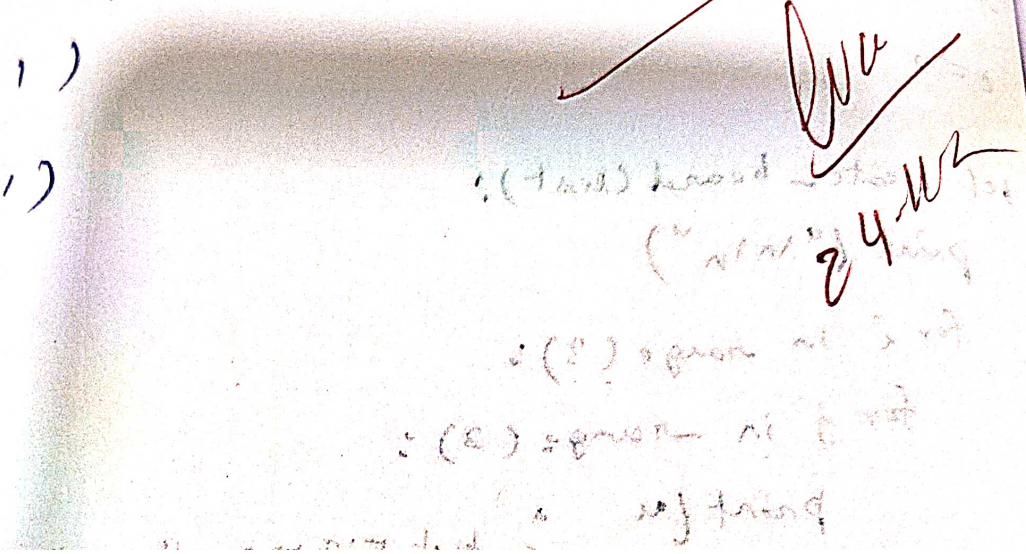
Tic Tac Toe game :

17/11/2023

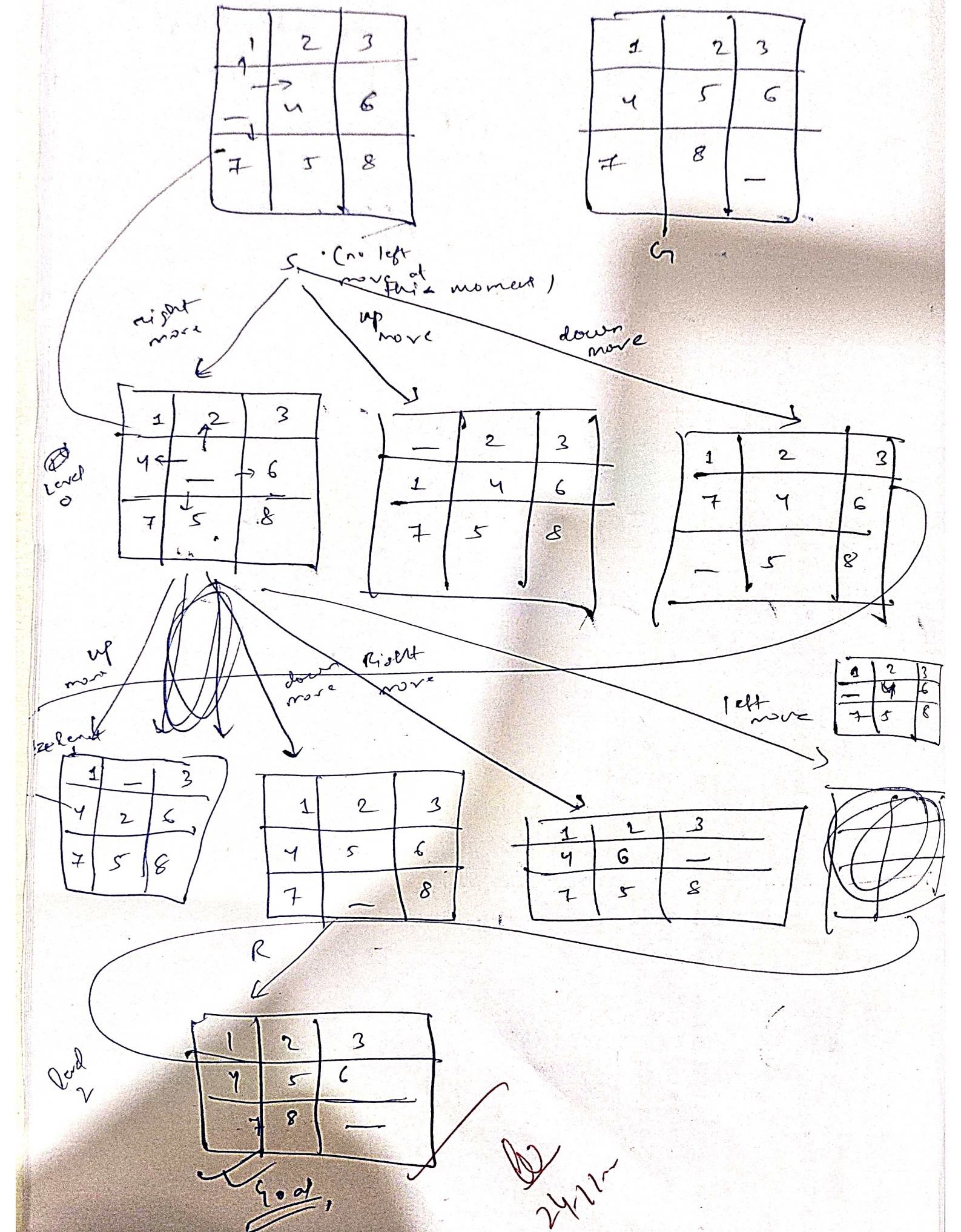



```
import math
list=[[1,2,3],[4,5,6],[7,8,9]]
c=0

def create_board(list):
    print("\n\n")
    for i in range(3):
        for j in range(3):
            print("  "+list[i][j],"  ",end="|")

def check_winner(list):
    for i in range(2):
        if(list[i][0]==list[i][1] and list[i][1]==list[i][2]):
            return True
        elif(list[0][i]==list[1][i] and list[1][i]==list[2][i]):
            return True
    x=list[0][0]
    y=list[0][2]
    count=0
    count1=0
    for i in range(3):
        for j in range(3):
            if(i==j and list[i][j]==x):
                count+=1
            if(i+j==2 and list[i][j]==y):
                count1+=1

def play(k,i):
    n=k/3
    m=(k%3)
    if(list[n][m]=='x' or list[n][m]=='o'):
        print("player",i,"re")
        c=c-1
    else:
        list[n][m]='x'
```

```
while(1):
    create_board(list)
    print("player", cx.2+1, "chance to play 1")
    print("enter pos")
    x=int(input())
    play(x, cx.2+1)
```



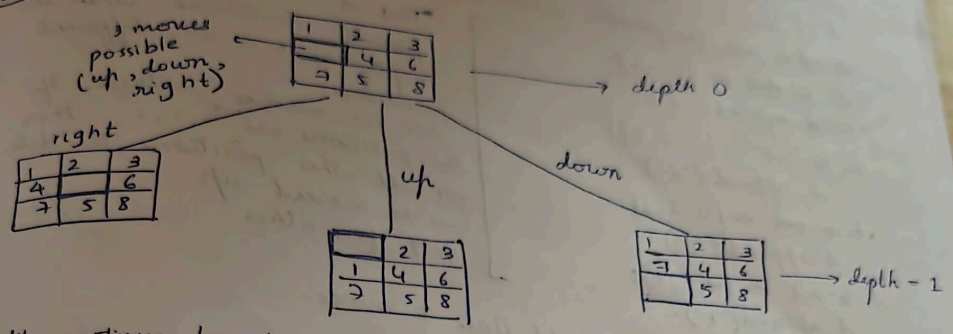
8 puzzle problem using BF



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')
```

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.

```
    # 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
    for i in d:                          # Now for each direction, I want to generate different state
        pos_moves_it_can.append(gen(state, i, b))   # of the mat which I append in pos_moves

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

— here, once I get a state which is not in visited_states, I return

```
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
```

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.

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

14-11-21

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]

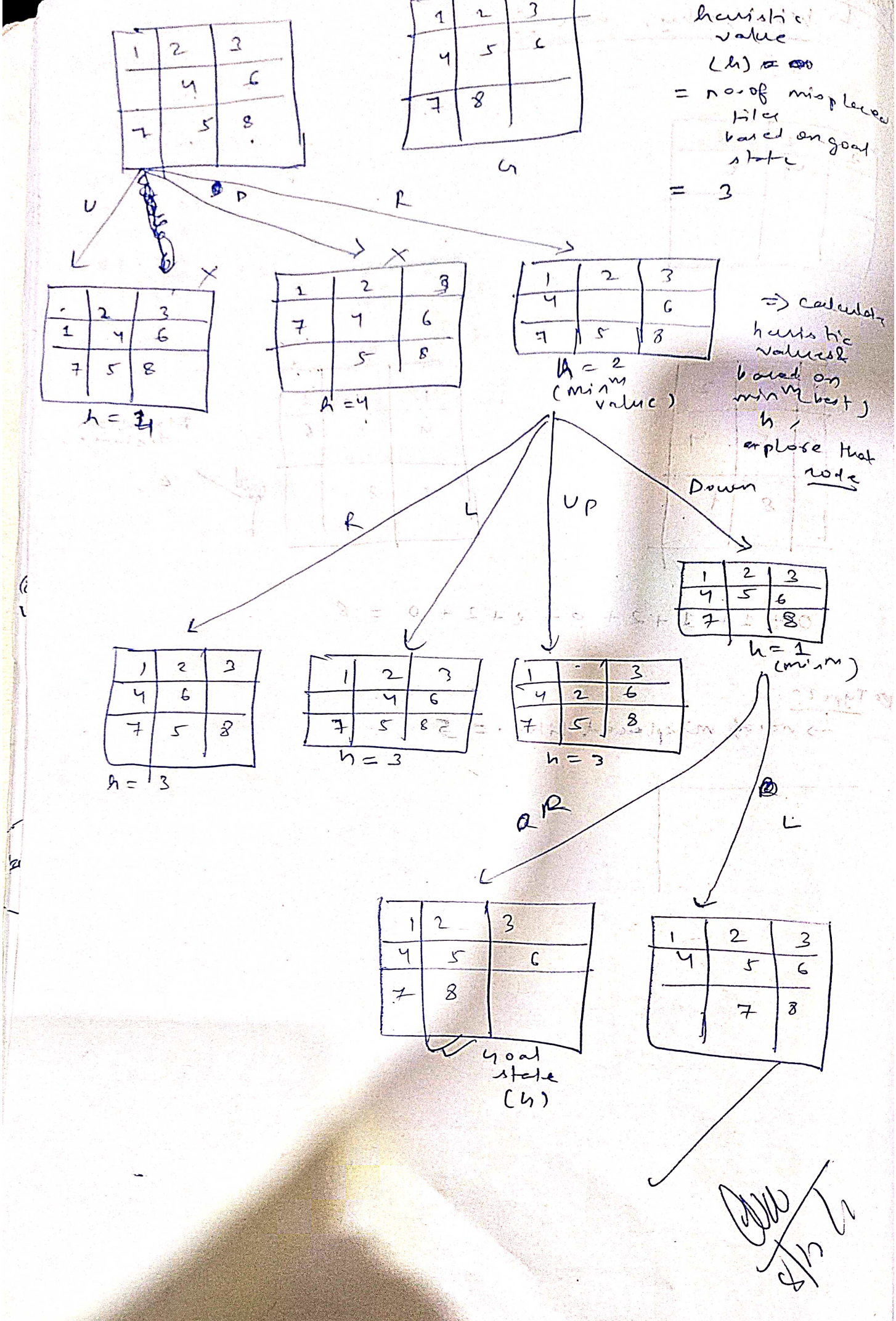
[0, 2, 3, 1, 5, 6, 4, 7, 8]

[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


Heuristic value (h) = no. of misplaced tiles based on goal state
= 3
U
D
R
X
h = 4
h = 4
h = 2 (min^m value)
⇒ Calculate heuristic values based on min^m best h, explore that node
Down
R
L
UP
h = 1 (min^m)
h = 3
h = 3
h = 3
R
L
goal state (h)

code :

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

def generate_child(self) :
    x, y = self.find(self.data, '_')
    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) :
    if x2 >= 0 & x2 < len(self.data) and y2 >= 0 :
        temp_puz = [ ]
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        return temp_puz
    else :
        return None

def process(self) :
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter goal state matrix \n")
    goal = self.accept()
    self.open()
    self.open.append(start)
```

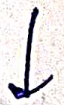
Output:

Enter start state matrix

1 2 3
4 5 6
- 7 8

Enter goal state matrix

1 2 3
4 5 6
7 8 -



1 2 3
4 5 6
- 7 8

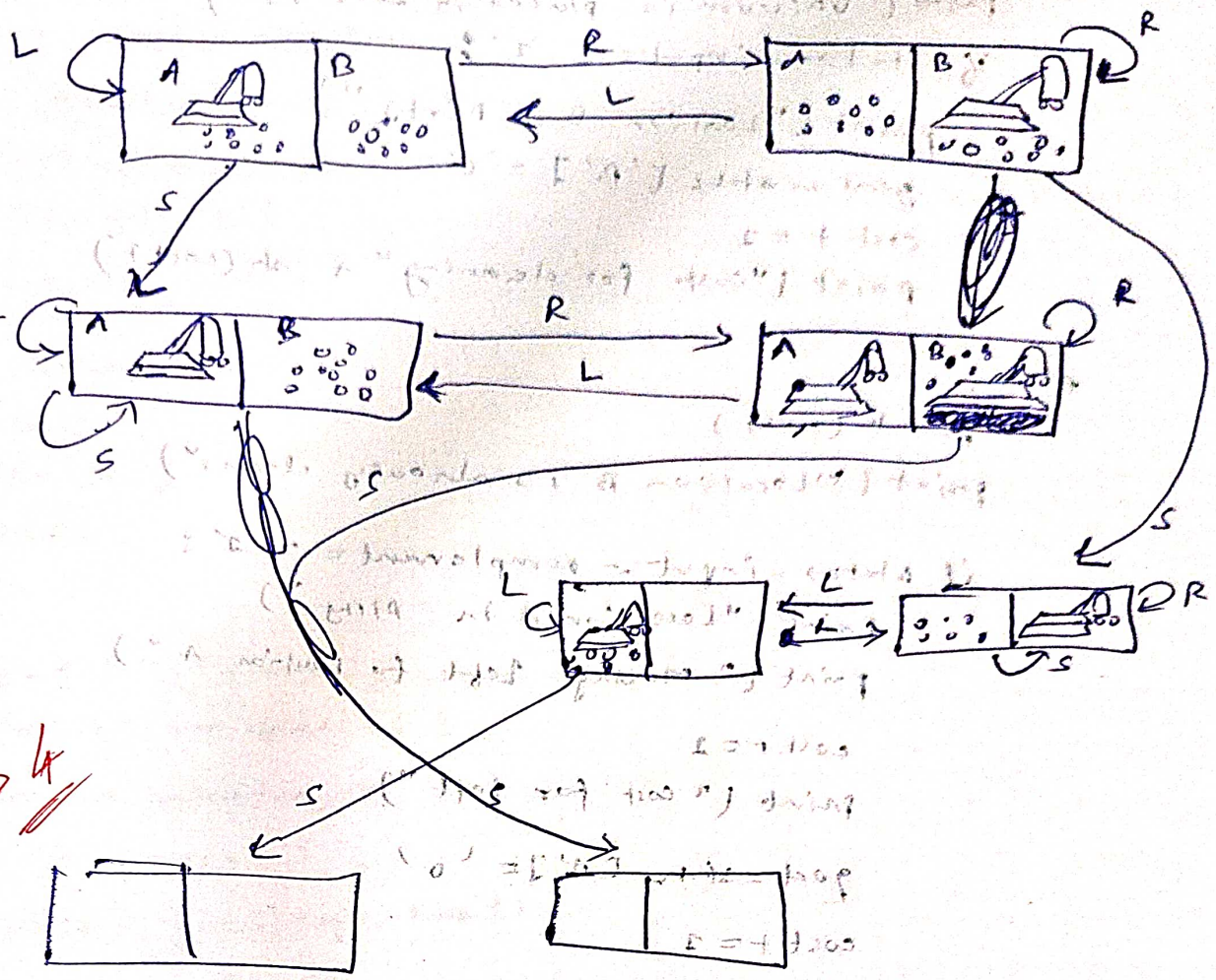
1 2 3
4 5 6
7 - 8

1 2 3
4 5 6
7 8 -

Vacuum cleaner program:

2×2^2 states = 8 states

Actions: → L, R, Suck (S)



Proceed →

Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")
    print("Initial location condition" + str(goal_state))

if location_input == 'A':
    print("vacuum is placed in location A")
    if status_input == '1':
        print("Location A is dirty")
        goal_state['A'] = '0'
        cost += 1
    print("cost for A")
```

```
else:
    print("Vacuum is placed in location B")
    if status_input == '1':
        print("Location B is Dirty")
        goal_state['B'] = '0'
        cost += 1
        print("cost for cleaning" + str(cost))
else:
    print(cost)
    print("Location B is already clean")

if status_input_complement == '1':
    print("Location A is Dirty")
    print("Moving left to Location A")
    cost += 1
    print("cost for left")
    goal_state['A'] = '0'
    cost += 1

else:
    print("No action" + str(cost))

print("Goal state:")
print(goal_state)

vacuum_world()
```

output:

Enter location of vacuum D

Enter status of D

Enter status of other room 1

Initial location condition { 'A': '0', 'B': '0' }

Performance measure: 2

22/12/23

8) Knowledge - Based Entailment :

```
from sympy import symbols.
def create_knowledge_base():
    p = symbols('p')
    q = symbols("q")
    r = symbols('r')

knowledge_base = (
    implies(p, s)
    implies(q, r)
    Not(r)
)

def query_entails(knowledge_base, query):
    entailment = satisfiable()
    return not entailment
if __name__ == "__main__":
    kb = create_knowledge_base()
    query = symbols('p')
    print("query:", query)
    print("query entails knowledge Base", result)

def tell(kb, rule):
    kb.append(rule)

combinations = [(T, T, T), (T, T, F), (T, F, T), (T, F, F), (F, T, T),
                (F, T, F), (F, F, T), (F, F, F)]

def ask(kb, q):
    for c in combinations:
    s = all(rule(c)) for rule in kb
    f = q(c)
    print(s, f)
    if s != f & s != false
        return "Does not entail"
entails().
```

output :

Knowledge base : implies(p, w) & (q, r)

query = p

query entails knowledge base : False

8) KB Resolution :

```
def tell(kb, rule):
    kb.append(rule)

combinations = [(T,T,T), (T,T,F), (T,F,T),
                (T,F,F), (F,T,T), (F,F,F)]

def ask(kb, q):
    for c in combinations:
        s = all(rule(c) for rule in kb)
        f = q(c)
        print(s, f)

kb = []

rule_str = input("Enter Rule 1 as lambda func :")
r1 = eval(rule_str)
tell(kb, r1)

query_str = input("Enter query as lambda")
q = eval(query_str)
result = ask(kb, q)
print(result)
```

Smeha
29/12/23

output :

```
Base : (q, r)
No entailed.
```

8) 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 = statement.replace("=>", '-')
expr = '\[([^\]]+)\]'
statements = re.findall(expr, statement)
for i,s in enumerate(statements):
    if '[' in s & ']' not in s
        statements[i]+ = ']'

for s in statements:
    statement = st.replace(s, fol_to_CNF(s))

while '-' in statements:
        i = statement.index('-')
        br = ('[') if '[' in this
```

output:

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

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

Explanation:

$\forall x$ king(x) $\wedge$ greedy(x) $\Rightarrow$ evil(x)

king(Richard) $\wedge$ greedy(Richard) $\Rightarrow$ evil(Richard)

A <=> B

is replaced with $(A \Rightarrow B) \wedge (B \Rightarrow A)$

Sneha

1/2/24

8) unification :

```
def unify ( x, y, bindings = [ ])
    if bindings is None :
        return None
    elif x == y :
        return bindings
    elif is instance (x, str) and x . startswith ('?') :
        return unify . var ( x, y, bindings )
    elif is instance (x, str) and instance (y, list) and len
        return unify . var ( x, y, bindings)
        for x_elem y_elem in zip (x,y) :
            bindings = unify (x_elem, y_elem )
    else :
        return None

def unify_var ( var, x, bindings) :
    if var in bindings :
        return bindings
    elif x in bindings :
        return unify ( var, bindings [x] )
                    ↳ variable
                    to store variable one
                    after binding
    else :
        new_bindings = bindings . copy ()
        new . bindings [var] = x
        return new_bindings .
```

output:

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

19/1/24

Forward Chaining

```
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