Program 1: TicTacToe

Choose one random between 0 & X using choice
= random.choice (0,1)
print the choice

take input from the input player 1 using
for i range (9):

print and ask square in which player will
want to play
i = input ("square:")
arr [i] = choice

store the value 0 or 1 in the square player
input in form of array of 1 to 9

Take ~~inp~~ input 4 times, 2 times for player
one and 2 times for player two without
using any condition

From 5th input start checking the condition
in horizontal, vertical & dicigonal using index
like for

Horizontal = arr[1] = arr[2] = arr[3]
vertical = arr[1] = arr[4] = arr[7]
dicagonal = arr[1] = arr[5] = arr[9]

with every input from 5th check condition
and if out of 8 conditions any one
condition get true end program

print the winner according to test input
of player if 1,5 0 mean 0 or 1 mean X

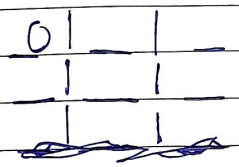
Program

```
a=[[" for _ in range(3)]
flag=0
choice= random.choice([0,1])
for i in range(x)
if (choice == 0 if choice == 1)
row = int(input(f"player {i}, enter row (0-2): "));
col = int(input(f"player {i}, enter column: "))

if 0<=row<=2 and 0<=col<=2 and a[row][col] == ":
break

else:
print("invalid move, Try again")
print(" Enter a valid input ")
[row][col] = Marks
if (i>=5)
if([arr[1] == arr[2] == arr[3]) ||
(arr[4] == arr[5] == arr[6]) ||
(arr[7] == arr[8] == arr[9]) ||
(arr[1] == arr[4] == arr[7]) ||
(arr[2] == arr[5] == arr[8]) ||
(arr[3] == arr[6] == arr[9]) ||
arr[1] == arr[5] == arr[9]) ||
arr[3] == arr[5] == arr[7]))
flag=1
break;
print("-----")
print("|" + arr[1] + '|' + arr[2] + '|' arr[3]
print("|_|_|_|")
```

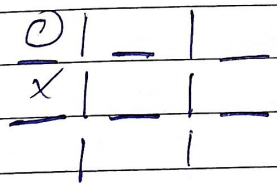
```
print ("|" + arr[4] + '|' + arr[5] + '|' arr[6]')
print ('|   |   |')
print ("|a arr [7] + '|' + arr[8] + '|' arr[9])
if flag == 1:
print ("F" player {i-3 wins: ")
else:
print ("its draw")
```

Output

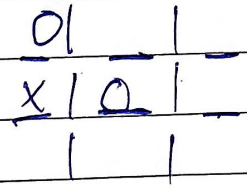
player 0
Choose the row (0-2): 0
Choose the column (0-2): 0



player 1
Choose the row (0-2): 0
Choose the column (0-2): 1



player 0
Choose the row (0-2): 1
Choose the column (0-2): 1



player 1
Choose the row (0-2): 0
Choose the column (0-2): 2

O | _ | _
X | O | X
_ | _ | _

player 0.
Choose the row (0-2): 2
Choose the column (0-2): 2

O | _ | _
X | O | X
_ | _ | O

player 0 wins.

8 puzzle program (BFS)

Algorithm

1. Create a list of 9 elements which represents 3X3 matrix with shuffled number from 1 to 8 and blank space with '0' or '-'
2. This will be a shuffled list and the goal state where we need to reach.
3. Initial state:

1	2	3
0	4	6
7	5	8

Final state

1	2	3
4	5	6
7	8	0

3. To implement A* using the BFS there will be one queue to explore the graph in bfs manner.
4. The empty space will get swap with the ~~all the (right), up)~~ numbers by using the (right) (up) (down) (left) element

There is one visited queue to record the unvisite queue

Time complexity = $\frac{\text{all possible moves}}{\text{no. of tiles}}$

$= \frac{24}{9} = 2.69$

Example

1	2	3
	4	6
7	5	8

To get.

1	2	3
4	5	6
7	8	-

right

1	2	3
4	-	6
7	5	8

up

-	2	3
1	4	6
7	5	8

down

1	2	3
7	4	6
-	5	8

down

1	2	3
4	5	6
7	-	8

up

1	-	3
4	2	6
7	5	8

~~down~~ right

1	2	3
4	5	6
7	5	8

left

1	2	3
4	5	6
-	7	8

right

1	2	3
4	5	6
-	7	8

Program-

```
from collections import deque
def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j

def generate_moves(board):
    moves = []
    blank_row, blank_col = find_blank(board)
    possible_moves = [
        (-1, 0), (1, 0), (0, -1), (0, 1)
    ]
    for dr, dc in possible_moves:
        new_row, new_col = blank_row + dr, blank_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_board = [row[:] for row in board]
            new_board[blank_row][blank_col], new_board[new_row][new_col] = new_board[new_row][new_col], new_board[blank_row][blank_col]
            moves.append(new_board)
    return moves

def solve_puzzle(initial_state, goal_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()
        visited.add(tuple(map(tuple, current_state)))
```

```
return path

possible_moves = generate_moves (current_state)
for move in possible_moves:
if tuple (map (tuple, move)) not in visited
queue.append ((move_path + [move]))

return None.

def print_step (solution_path):
if solution_path:
print("step to reach the goal:");
for step in solution_path:
print("      ");
for row in step
print(" ", end = " ")
if val == 0:
print("  ", end = " ")
else:
print (val, end = " ")
print()
print()
else
print("No solution exists for this puzzle")
```

initial :

1	2	3
	4	6
7	5	8

final state

0	1	2
3	4	5
6	7	8

solution path = solve_puzzle (initial, goal)
print_steps (solve-path)

output:

1 2 3
4 6
7 5 8

1 st

1 2 3
4 6
7 5 8

2nd

1 2 3
4 5 6
7 8

4th

1	2	3
4	5	6
7	8	

8 puzzle using A* search

Final state

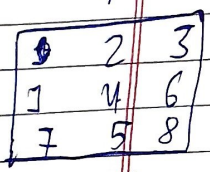
1	2	3
4	5	6
7	8	

heuristic value:
no of misplaced value by checking the goal state

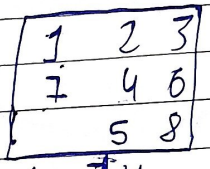
h=3

g+h = 0+3 = 3

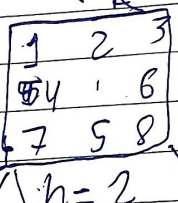
g=1



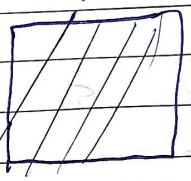
h=3



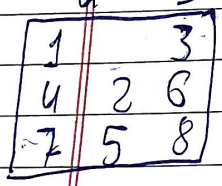
h=4



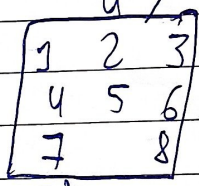
h=2



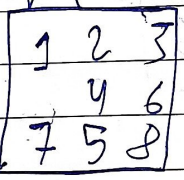
Take the minimum value and explore that state



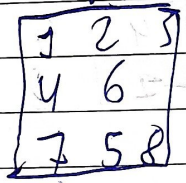
h=3



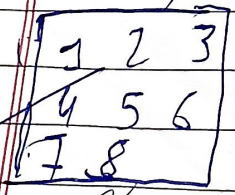
h=1 (min)



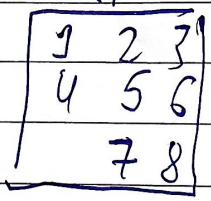
h=3



h=3



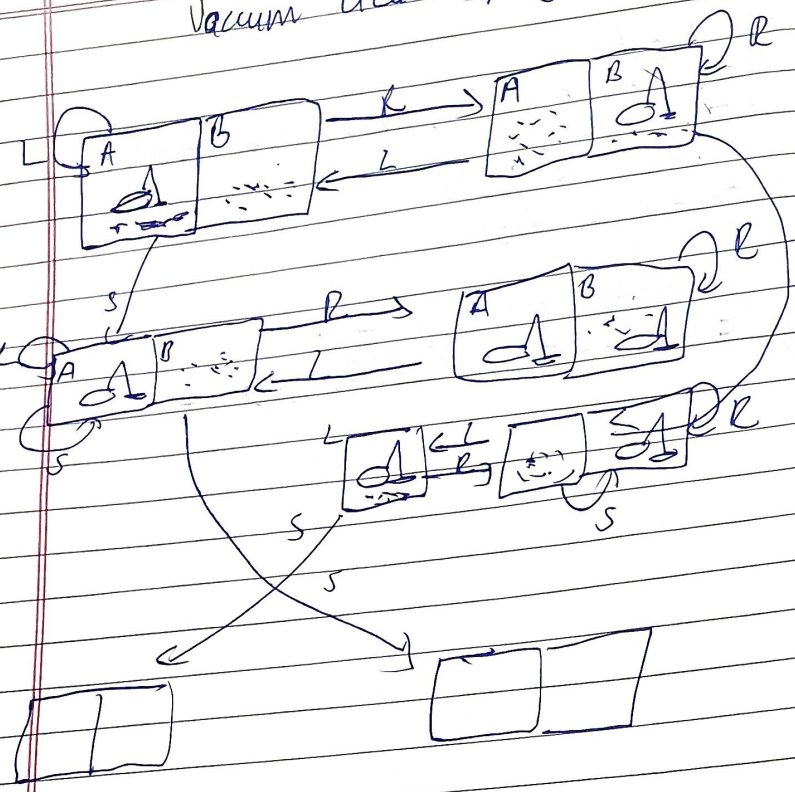
goal state



~~Algorithm~~ 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, '_')
```

Vacuum cleaner program



Code:

```
def vacuum_world ():
 goal_state = {'A': '0', 'B': '0'}
 cost = 0
 location_input = input ("Enter loc of vacuum");
 status_input = input ("Enter status of " + location_input)
 status_input_complement = input ("Enter the
 status of the other room ")
 print ("Initial location condition " + str (goal_state))
 if location_input == 'A';
 print("Vacuum is placed in location A")
 goal state ['A'] = '0';
 cost + = 1
 print ("Cost for A")
 else;
```

```
print("Vacuum is placed in loco B")
    if status_input == '1':
print("Location B has dirts")
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 for location A")
  cost += 1
print("Cost for left")
goal_state['A'] = '0'
  cost += 1
else
print("No action" + str(cost))
print("Goal state:")
print(vaccum_world())
```

Output:-

Enter location of vacuum D

Enter status of 0

Enter states of other room 1

Initial location condition {'A' = '0'

Performance measure = 2.

Knowledge based entailment

(i) Define the function weather the entailment (hypo premise a, b, c) where a, b, c are predetermined condition

(ii) Check the entailment condition based on criteria.

if premise = "humid" and as to:
return True
else if premise = "cloudy and b) to:
return true.
else if premise = "other condition" and
or
return true
else
return false.

example usage of the condition
premise condition = "humanid" or
premise condition = "cloudy" or
premise condition "other condition"
hypothesis_text = "The weather is uncomfortable.

4= Assume predetermined condition for humidity, cloudiness and other conditions like
cloudness_condition = 60
other condition = 40

5= call the function weather_entailment()

2= Check the result and print appropriate

message

Output.
knowledge base = implies (p, q) & (q, r)
Query : p
Query entails knowledge based - False

Knowledge based resolution

def
KB = P, — PVQ, PV→QVR, ↑ QVR

```
from sympy.logic.bootaly import Or, And, Not
from sympy.abc import P, Q, R

def main():
try:
    expression1 = P
    expression2 = Or (Not (P), Q)
    expression3 = Or (Not (Q), R)
    expression4 = Or (Not (Q), R)

    knowledge_base = And (expression1, expression2,
                          expression3, expression4)
    print ("Knowledge Base:")
    print (Knowledge_base)

    resolved_kb = knowledge_base.simplify ()
    print ("In Resolved Knowledge based:")
    print (resolved_kb)

    negation_of_P = Not (P)
    negation_of_Q = Not (Q)
    negation_of_R = Not (R)

    print ("In Negation of P:")
    print ("negation_of_P)

    print ("In Negation of Q:")
    print ("negation_of_Q)
```

```
print ("\n Negation of R: ")
print ( negation_of_R)

exce
except Exception as e:
print ("An error occured : {e}")

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

Output:

Knowledge Base:
P & (Q | ~P) & (R | ~Q) & (P | R | ~Q)

Resolved knowledge base:
True

Negation of P:
~P

Negation of Q:
~Q

Negation of R:
~R

Program - 8

Implement unification in first order logic

```
def unify_var (var, x, theta):
if var in theta:
return unify (theta[var], x, theta)
else if x in theta:
 return unify (var, theta[x], theta)
else:
    theta[var] = x
    return theta
def unify (x, y, theta = {}):
 if theta is None:
  return None.
else if  x == y
return theta
else if is instance (x, str) and x(0). is lower():
return unify_var (x, y, theta)
else if is instance (y, str) and y(0). is lower();
return unify_var (y, x, theta)
else if isinstance [x, list] and is instance (y, list).
if len(x) != len(y)
return None

for xi, yi in zip (x, y):
 theta = unify (xi, yi, theta)

if theta is None:
return None
return theta

else:
return None:
```

```
x= [ 'p' , 'a', 'x' ]
y= [ 'p' , 'y' , 'z' ]

result = unify (x,y)
print (result)
```

Input :
Expression 1: ['p', 'a', 'x']
Expression 2: ['p', 'y', 'z']
Output
Unification successful
Substitution theta : {'x': 'z', 'y': 'a'}

19101124

Program 9

→ Implement a given first order logic statement into conjunctive Normal form (CNF)

```
from sympy import symbols, to_cnf, parse_expr
def convert_to_cnf(logic_statement):
    parsed_statement = ~~statement~~ parse_expr(logic_
    statement)
    cnf = to_cnf(parsed_statement)
    return cnf.

if __name__ == "__main__":
    logic_statement = "(p|~q) & (~p|r)"
    cnf_result = convert_to_cnf(logic_statement)
    print("Original Statement:", logic_statement)
    print("CNF Form:", cnf_result)
```

Output

Original Statement : (A & B) | (~(C & D))

CNF Form : (A|~C) & (A|D) & (B|~C) & (B|D)

19101124

Program 10

Create a knowledge base consisting of first order logic statements & prove the given query using forward reasoning

```
from sympy import symbols, Eq, And, Or, Implies
ask satisfiable

John, Many, Alice, Bob = symbols ('John, Many, Alice, Bob')
Parent = symbols ("Parent")
Grandparent = symbols ("Grandparent")

Knowledge_base = [Eq (Parent (John, Alice), True),
                  Eq (Parent (Many, Alice), True),
                  Eq (Parent (Alice, Bob), True),
                  Implied (Parent (x, y), Grandparent (x, y)),
                  ]

query = Grandparent (John, Bob)

def forward_reasoning (knowledge_base query):
    new_facts = set()

while True:
    for fact in knowledge_base:
        if ask (fact):
            continue
        if satisfiable (fact):
            new_facts.add (fact)
            if not new_facts:
                break:
```

```
knowledge_base.extend(new_facts)
return ask(query)

result = forward_reasoning(knowledge_base, query)
print("Query", query)
print("Result", result)
```

Output:

Query: Grandparent(John, Bob)
Result: True.

25/11/24