

17/W/23

Algo :-

- (1) choose one random b/w 0 & X using
choice = random.choice (0, 1)
print the choice
- (2) take input from the player 1 using
for i range (9):
print and ask
- (3) print and ask square in which
player want to play:
 $j = \text{input}("square : ")$, $k = \text{input}("0 or X")$
 $\text{arr}[j] = \text{choice } k$
- (4) store the value 0 or 1 in the
square they player input in
form of array of 1 to 9.
- (5) Take input 4 times, 2 times for
player one and 2 times for
player two without using
any condition
- (6) From 5th input start checking
the condition in Horizontal,
Vertical & diagonal using
index like for
Horizontal = $\text{arr}[1] == \text{arr}[2] == \text{arr}[3]$
Vertical = $\text{arr}[1] == \text{arr}[4] == \text{arr}[7]$
Diagonal = $\text{arr}[1] == \text{arr}[5] == \text{arr}[9]$

- (7) with every input from 5th check condition and if out of 8 conditions any one condition get true end program.
- (8) print the winner according to last input of player if it's 0 mean O or 1 mean X.

1. Program :-

```
a = [[ '' for _ in range(3) ]  
flag = 0  
choice = random.choice([0, 1])  
for i in range(9):
```

```
if choice == 0 || choice == 1  
    row = int(input("player {i}:  
        - enter row(0-2):"))
```

```
    col = int(input("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")  
if a[row][col] == "None":
```

```
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(' 1   2   ')
print(' '+arr[0]+arr[1]+arr[2]+'|'+arr[3]+arr[4]+arr[5])
print(' 1   1   ')
print(' '+arr[6]+arr[7]+arr[8]+'|'+arr[9]+arr[10])
print(' 1   1   ')
```

```
if flag == 1:
    print("player 1 wins!")
else:
    print("it's draw")
```

Output

player 0

choose the row (0-2) : 0

choose the column (0-2) : 0

0	1	
1		
1	1	

player 1

choose the row (0-2) : 0

choose the column (0-2) : 1

0	-	-
X	-	-
1	1	-

player 0

choose the row (0-2): 1

choose the column (0-2): 1

O		
	-	-
X	O	1
	-	-

player 1

choose the row (0-2): 2

choose the column (0-2): 1

O		
	-	-
X	1	X
	-	-

player 0

choose the row (0-2): 2

choose the column (0-2): 2

O		
X	O	X
	-	-
	O	

player 0 wins

Q 18/11
X

1	2	3
4	5	6
7	8	9

computer's turn :

1	2	3
4	5	6
X	8	9

Your turn :

enter a number on the board :2

1	0	3
4	5	6
X	8	9

1	0	3
4	5	X
X	8	9

Your turn :

enter a number on the board :5

1	0	3
4	0	X
X	8	9

computer's turn :

1	0	3
4	0	X
X	X	9

Your turn :

enter a number on the board :9

1	0	3
4	0	x
x	x	0

computer's turn :

x	0	3
4	0	x
x	x	0

Your turn :

enter a number on the board :4

x	0	3
0	0	x
x	x	0

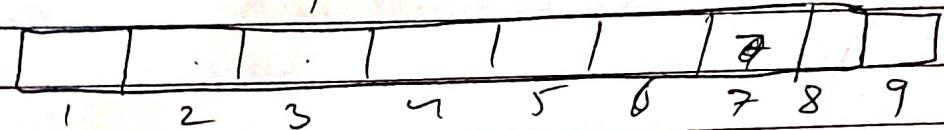
x	0	x
0	0	x
x	x	0

2

8-puzzle using BFS

Algorithm

1. Create a list of 9 element which represents 3×3 matrix with shuffled number from 1 to 8 and ~~1 to 8~~ blank space with 0



2. There will be a shuffled queue list and the goal state where we need to check.
3. To implement using the BFS there will be one queue to explore the original grid in breadth-first manner.
4. The empty space will get swap with the number with it's (up ↑), (down ↓), (right →), (left ←) element.
5. There is one visited queue to keep record of the visited state to avoid revisiting.
6. To check if the queue is empty or it reached it's goal state.

7. With each iteration of loop it will check if it's reached goal state or need to generate more successors of pattern.

1. ~~Swap~~

1. Creating queue and board.

def board(board)

for r in board:

print(r)

print()

2. Checking empty places

for i in range(3)

for j in range(3):

if board[i][j] == 0:

3. To check if the move is valid.

if $0 \leq n < 3$ and $0 \leq y < 3$

4. To swap the elements

if move == 'up'

board[n2][y1], [n1][y2]

= board[n2][y2], board[n1][y1]

5. Queue traversal

q = queue()

8. get the blank position.

We can get the possible moves

move $[0, 1]$ = Right $[0, -1]$ left
 $[1, 0]$ = down $[-1, 0]$ up

Check if the move is valid.
and swap

if current state == goal state.
~~return~~ return path.

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

8-12-23

3

8 puzzle using iterative deepening search algorithm.

① initialize the initial state and final state of 8 puzzle

for i in range m:

 for j in range n:

 initial[i][j] = x;

Goal state

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

② Find the blank space in the 8 puzzle by finding 0

for i in range(3)

 for j in range(3)

 if initial[i][j] == 0:

 return i, j

③ Write the possible action that can be applied on the 8 puzzle

if blank row > 0:

 Move up

if blank row < 2:

 Move down;

if blank row > 0:

 Move left

if blank col < 2:

 Move right

(4) Apply the action by swapping the position of all the possible action

if action = up

new state [b-row][b-col]; new state [b-row-1][b-col]
~~= new state [blank]~~
~~- (new state [b-row-1][b-col]), new state [b-row][b-col]~~

(5) Then check if the state is equal to the goal state.

goaltest (state, goal-state):

return state == goal-state

(6) Apply iterative deepening search algorithm

Run a while loop to trace the depth limit

(while \neq depth-limit)

{

3

(7) Apply the action on the puzzle to find possible written in step 3 & 4
~~no~~

node = Puzzle (initial, node, action)

(8) Get the node & depth and
and repeat the processes using
recursion.

result = dis(node, depth - 1);

(9) if the depth limit is others
the end the search there only

if depth_limit == 0;
return "solved already";

if not then check goal state with
every iteration of while loop

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

→

1	2	3
4	7	5
6	8	

1	0	3
4	2	5
6	7	8

off *flag*

1	2	3
0	4	5
6	7	8



4

8 puzzle using A* Search.

- (1) Take the Input of initial state stack and the final state of 8 puzzle from user

initial = [int(i) for i in s.]

goal = [int(i) for i in s.]

- (2) Identify the empty tile position.

for i in initial :

if i == 0 :

pos = i;

break;

- (3) Find the row and column of the empty position

1	2	3	4	0	5	6	7	8
•	•	•	3	4	5	6	7	8

row = pos // 3;

col = pos % 3;

- (4) Apply action on the empty tile.

for move in (0, 1, 0, -1)

- (5) Check the move is valid or not

if $0 \leq \text{new row} \leq 3$ for row

if $0 \leq \text{new col} \leq 3$ for col.

(6) function to check if all the misplaced tile is placed goal (node, goal)
if (node == goal.state)
goalstate = 1
3

(7) Run a while loop till the goal state is reached or not
while (goal.state != 1)
and to track the depth.

(8) calculate $f(n)$ by using $f(n) = g(n) + h(n)$
where

$g(n)$ = the value of the loop

~~for i in node:~~
~~if goal.state[i] != node[i]:~~
~~a++~~
 $h(n) = a$

(9) compare $f(n)$ at each depth
and for the node which is min
apply action Step 4 & 5

Result = (node, f(n))

Step 12

8-puzzle problem using T.DFS

Code

```

def iddfs(puzzle, goal, structure)
    input em
    
    def dfs(route, depth):
        if depth == 0:
            return route
        if route == (-1) == goal:
            return route
        for move in getMoves('while c'):
            if move not in route:
                nextmove = dfs(route + move, depth - 1)
                if nextmove:
                    return nextmove
        return None
    
    for depth in iterfunc(count()):
        route = dfs(puzzle, depth)
        if route:
            return route
    
    def possibleMove(state):
        b = state[0]
        d = []
        if b not in [0, 1]:
            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')
        return d
    
```

permove = ()

for i in d

permove.append(generate(m, i))
return permove

def generate(state.. m, b)

temp = state.copy()

if m == 'd':

temp[i][b] = temp[b]

return temp

uint : a = [1, 2, 3, 0, 4, 6, 7, 8]

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

rank = id = def (init:a, goal, possible),

if route:

print("success")

printf("path": route)

else:

print("failed to find sol")

Output :

success

path [C 1 7 0 4 6 7 8] (1, L)]

Output:

Success!! It is possible to solve 8 Puzzle problem

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

5

solve puzzle problem using A* algo

Code

import heapq

def get_blank_pos(state)

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def get_neighbours(state, moves)

row, col = get_blank_pos(state)

new_row, new_col = row + move

[+col+move(i)])

if 0 <= new_row < 3 and 0 <= new_col < 3:

new_state[link(row) for row in state]

new_state[new_row][new_col];

new_state[new_row]

return new_state

return None

def is_good(state)

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

return state == goal_state

def heuristic(state)

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0
goal = rows, goal_id
= division of state[i][j]

distance += abs(i - goal_id)

return distance;

def solve_puzzle(initial_state; now, now)

open_set[start_node].

closed_set = Set()

while openset:

current_state, parent, new
heappop, heap appendne.)

if goal(current_state).

path()

while parent:

path.insert(0, (move; uni_state))

closed_set.add(Kmple((map(tuple, curstate)))

new_ver = [(0, 1), (1, 0), (0, -1), (-1, 0)]

initial_state = [(1, 2, 3)], [4, 5, 6], [7, 8]

Setpath = solve_puzzle(initial_state)

```
if sol path  
    print ("move", mac)
```

```
    for row in state:  
        print (row)
```

```
    print ("---").
```

```
else
```

```
    print ("NO col")
```

Output

1 2 3

~~4~~ 5 6

7 8

1 2 3

~~4~~ 5 6

7 8

1 2 3

4 5 6

7 8 0

4 2 3

4 5 6

7 8 0

123
456
780

Output:

Enter the start state matrix

1 2 3
4 5 6
— 7 8

Enter the 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 —

29/12/23

6 Knowledge based Entailment

1. Define the function
weather entailment(hypo, premise, a, b, c)
where a, b, c are pre-determined
condition
2. Check the Entailment condition based
on Criteria.

~~if premise == ''
if a > 70:
premise = ''.~~

~~if premise == "humid" and a > 70:
return True.~~

~~else if premise == "cloudy" and b > 50:
return True.~~

~~elif premise == "other condition" and c > 50:
return True.~~

~~else~~

~~return False.~~

3. Example usage of the function.

premise condition = "Humid" or

premise condition = "Cloudy" or

premise condition = "Other condition"

hypothesis. Let's = "The weather is ~~un~~ uncomfortable".

4 Assume predetermined conditions
for humidity, cloudiness and
other conditions like
humidity-condition = 75
cloudiness-condition = 60
other-condition = 40

5 call the function weather_environemt()
6 check the result and print
appropriate message.

Program

Code

def weather_entailment(hypothesis, premise, a, b, c)

 if premise == "humid" and a > 0:
 return True

 elif premise == "cloudy" and b > 0:
 return True

 elif premise == "other-condition" and c > 35:
 return True

else

 return False

precond

premise-condition = "humid" or "cloudy"
or "other-condition"

hypothesis-text = "The weather is un.comfortable"

humidity-condition = 75

Cloudness-condition = 50

Other-condition = 30

Result = weather_entailment(hypothesis-text,
premise-condition, humid condition,
cloudness-condition, other condition)

if result:

 print("The hypothesis is entailed
 by 3 premise-condition's conditions")

ex:

prin(F "The hypothesis is entailed by the premise (condition) considered")

output

The hypothesis is entailed by human condition.

The weather is uncomfortable.

~~Q1~~

25/12/23

the hypothesis is entailed by human condition

Knowledge Base: $\neg r \wedge (\text{Implies}(p, q)) \wedge (\text{Implies}(q, r))$

Query: p

Query entails Knowledge Base: False

19/01/24

Unification

1. For input take two expression E_1 & E_2 that is needed to unify
2. Substitution — initially set to an empty Substitution
3. The procedure to deal with cases
 - (a) if E_1 is equal E_2 , return the current substitution

if $E_1 == E_2$

return sigma

- b) if E_1 is a variable and E_1 is not equal to E_2 check if E_1 occurs in E_2 and if it does, return failure.

elif is variable(E_1):

return unify var(E_1, E_2, σ)

- c) if E_1 is a variable and E_1 is not equal to E_2 check if E_2 occurs in E_1 and if it does, return failure

elif is variable(E_2):

return unify var(E_2, σ)



unify-var(var, exp, sigma)

{

if var in sigma:

return unify(sigma[var], exp, sigma)

elif exp contains var (expression, var):
return none

else

sigma[var] = expression.

3

c) If E_1 & E_2 are complex terms
(non variable) with same factor
Recursively unify

elif is_complex(E_1) and is_complex(E_2)

4. if predicate(term1) ≠ predicate(term2),
return FAIL

5. Number of arguments ≠ return fail

6. set(SUBST) to NIL

7. for i=1 to no. of elements in term1

a) call unify(ith element in term1, ith element in term2)

put result into S

b) S = FAIL

result = fail

c) if $S \neq \text{NUL}$

- a. Apply S to the remainder of $L_1 \& L_2$
- b. $\text{SUBST} \rightarrow \text{append} (S \text{ SUBST})$

8. Return SUBST

8

FOL to CNF conversion

1. Create a list of Skolem constants
2. Find A, Z
 if the attributes are lower case
 , replace them with a Skolem constant.
 Remove used skolem constant or
 function from the list
 if the attribute are both lower case
 and uppercase. replace the uppercase
 attribute with a Skolem function.
3. replace \leq with ' $-$ '
 transform - as $Q = (P =) \cap (A =) \rho$
4. replace $=$ with ' $-$ '
5. Apply deMorgan's law
 replace $\sim [$
 as $\sim P \& \sim Q$ if ($\&$ was present)
 replace $\sim [$
 as $\sim P \mid \sim Q$ if (\mid was present)
 replace $\&$ with ' $,$ '

Unification code

```
def is_variable(term):  
    return isinstance(term, str) and  
    term.islower()
```

```
def is_list(term):  
    return isinstance(term, list)
```

```
def unify_var(var, expression, substitution):  
    if var in substitution:  
        return unify(substitution[var],  
                     expression, substitution)
```

```
    elif expression == var:  
        return None
```

```
    else:  
        substitution[var] = expression  
        return substitution
```

```
def expression_contains_var(expression, var):  
    if expression == var:  
        return True
```

```
    elif is_list(expression):  
        return any(expression_contains_var(elem, var)  
                  for elem in expression)  
    return False
```

def unify(expression1, expression2,
 substitution = None):

 if substitution is None:

 Substitution = {}

 if expression1 == expression2:
 return substitution

 if is_variable(expression1):

 return unify_var(expression1, expression2,
 substitution)

 if is_variable(expression2):

 return unify_var(expression2, expression1,
 substitution)

 elif is_list(expression1) and is_list(expression2):
 return unify_list(expression1, expression2,
 substitution)

 return None

Expression1 = ['likes', 'John', 'pizza']

Expression2 = ['likes', 'x', 'pizza']

result = unify(Expression1, Expression2)

Output:

```
[9] exp1 = "knows(x)"
    exp2 = "knows(Richard)"
    substitutions = unify(exp1, exp2)
    print("Substitutions:")
    print(substitutions)

Substitutions:
[('Richard', 'x')]

[7] exp1 = "knows(A,x)"
    exp2 = "k(y,mother(y))"
    substitutions = unify(exp1, exp2)
    print("Substitutions:")
    print(substitutions)

Predicates do not match. Cannot be unified
Substitutions:
False
```

23

```
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)

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

exp1 = "knows(A,x)"
exp2 = "knows(y)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)

Substitutions:
False
```

FOL to CNF code.

```
def getAttributes(string)
    expr = '\([^\)]+\)'
    matches = re.findall(expr, string)
    return [m for m in matches
            if m[0].isalpha()]
```

~~def-gt predicates (coming)~~

$$\text{expr}' = [a-z_2] + 1(A-z_a-z_j-1)'$$

return re.findall(expr, string)

def Skolemisierung (Statement)

'SKOLEM. Constants = [f'chr(c)3' for c in range(ord('A'), ord('Z') + 1)]

```
matches = re.findall("[\r\n].*", stamen1)
```

for main markers (-1):

Statement = Statement.replace(maven, "")

for predicate in getpredicates(syntax):

attribute = getAttribute (predicate)

if ''join_attributes.ipynb'':

Statement = memory + response

(Maten[], SKOLEM-CONSTANTS. POP())

return statement

def fol_to_cnf(fol):

statement = fol.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] + ')' + statement[br+1:]

statement = statement[:br] + new_statement

if br > 0:

else

new_statement

return skolemization(statement)

~~call by value~~
complete program yet.
I wrote it.

Output:

```
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("forall[y][animal(y)=>loves(x,y)]=>[exists[z][loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))

[~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)]|criminal(x)
```