# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**On**

**ARTIFICIAL INTELLIGENCE**

**Submitted by**

**Dhravya M(1BM21CS056)**

**in partial fulfillment for the award of the degree of**
**BACHELOR OF ENGINEERING**
**in**
**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**
**(Autonomous Institution under VTU)**
**BENGALURU-560019**
**Oct 2023-Feb 2024**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**
**(Affiliated To Visvesvaraya Technological University, Belgaum)**
**Department of Computer Science and Engineering**

CERTIFICATE

This is to certify that the Lab work entitled **"ARTIFICIAL INTELLIGENCE"** carried out by **Dhravya M (1BM21CS056)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence Lab - **(22CS5PCAIN )**work prescribed for the said degree.

**Dr. GAURI KALNOOR**
Assistant Professor
Department of CSE
BMSCE, Bengaluru

**Dr. Jyothi S Nayak**
Professor and Head
Department of CSE
BMSCE, Bengaluru

17/11/2023

① TicTacToe

```
import random
tic = [1, 2, 3, 4, 5, 6, 7, 8, 9]

def printBoard(tic):
    print(tic[0]+'|'+tic[1]+'|'+tic[2])
    print("- - - - - - - - - -")
    print(tic[3]+'|'+tic[4]+'|'+tic[5])
    print("- - - - - - - - - -")
    print(tic[6]+'|'+tic[7]+'|'+tic[8])

def isWinner(tic, pos):
    if tic[0]==tic[4] and tic[4]==tic[8] or
       tic[2]==tic[4] and tic[4]==tic[6]:
        return True
    else if tic[pos%3]==tic[pos+3] and tic[pos%3]==tic[pos+6]:
        return True
    else if tic[pos//3+1]=tic[pos//3+2] and tic[pos//3+2]==
                                            tic[pos//3+3]:
        return True
    return False

def update_user(tic):
    num = int(input("Enter a number on the board"))
    while(num not in tic):
        tic[num-1]='0'
        num = int(input("Enter a number on the board"))
    tic[num-1]=='0'

def update_comp(tic):
    for i in tic:
        if i!='x' and i!='0':
            tic[i-1]='x'
            if(isWinner(tic, i-1)==True):
                return
            else:
                tic[i-1]=i
```

```
for i in tic:
    if i!='X' and i!='O':
        tic[i-1]='O'
        if (isWinner(tic, i-1)==False):
            tic[i-1]='X'
            return
        else
            tic[i-1]=i

while (random.rand(9) not in tic):
    num
num = random.rand(9)
while (num not in tic):
    num=random.rand(9)
tic[num-1]=='X'

printBoard(tic)
count=0

if count%2==0:
    print("computer's turn")
    update_comp(tic)
    count+=1
else if count%2!=0:
    print("user's turn")
    update_user(tic)
    count+=1
if count>=5:
    if(isWinner(tic,pos)==True):
        print("Winner",tic[pos-1])
```

Output:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| X | 8 | 9 |

| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 5 | 6 |
| X | 8 | 9 |

| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 5 | X |
| X | 8 | 9 |

| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 0 | X |
| X | 8 | 9 |

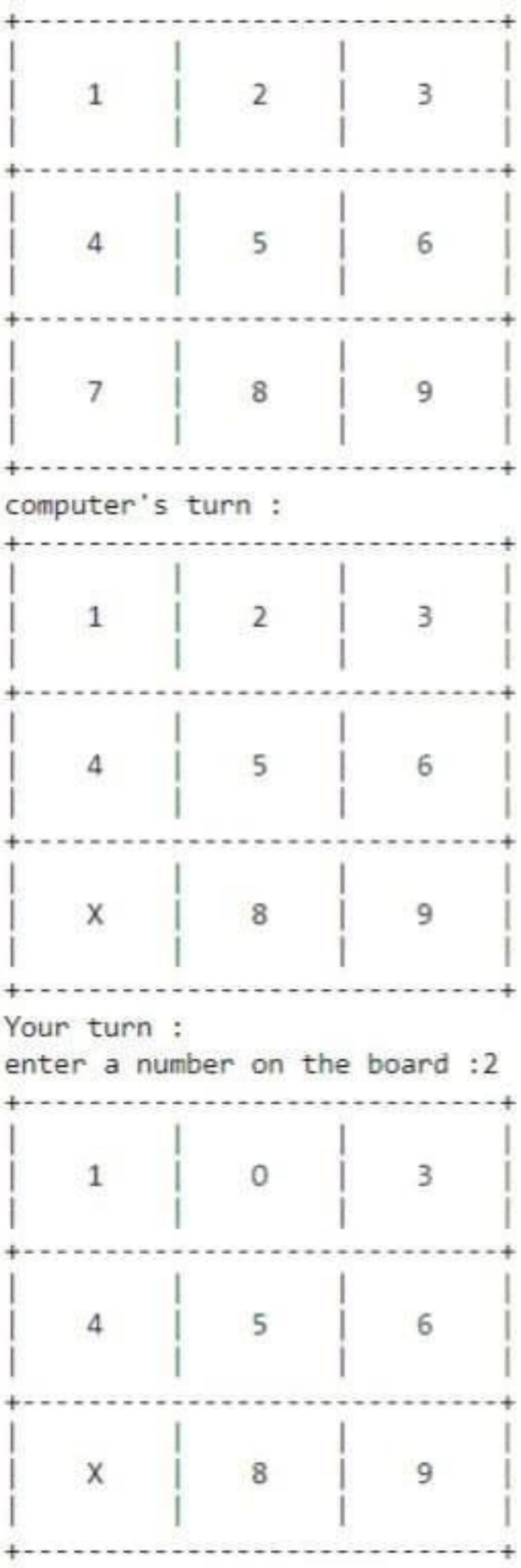
| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 0 | X |
| X | X | 9 |

| | | |
|---|---|---|
| 1 | 0 | 3 |
| 4 | 0 | X |
| X | X | O |

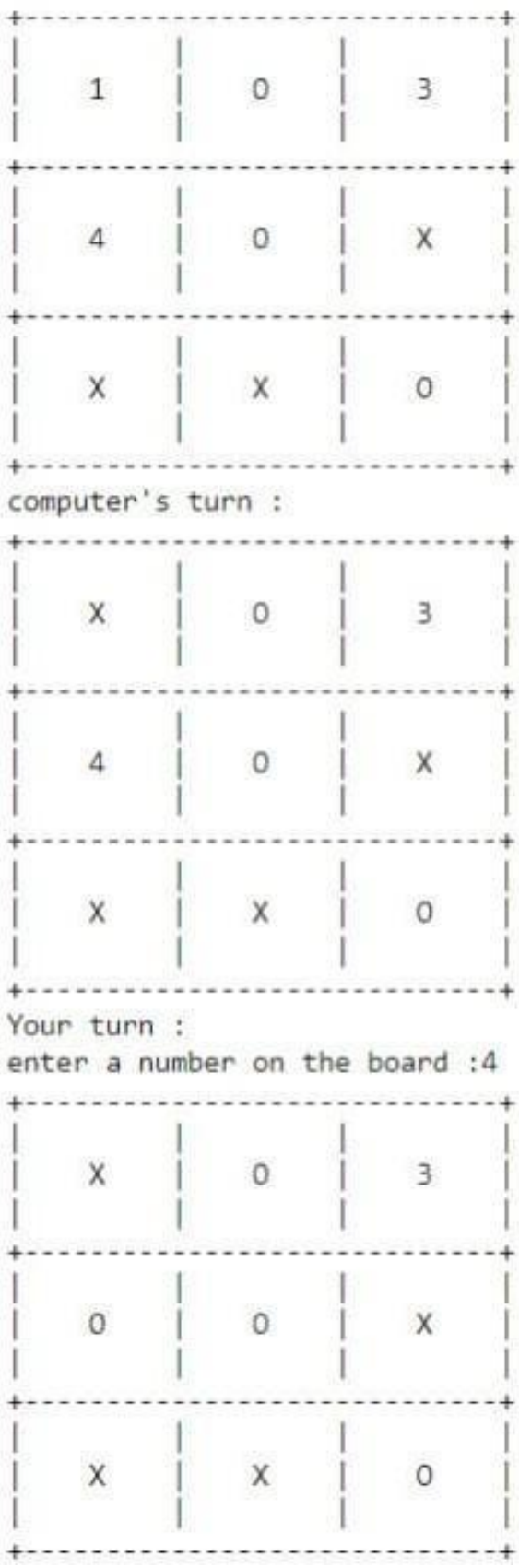
| | | |
|---|---|---|
| X | O | 3 |
| 4 | O | X |
| X | X | O |

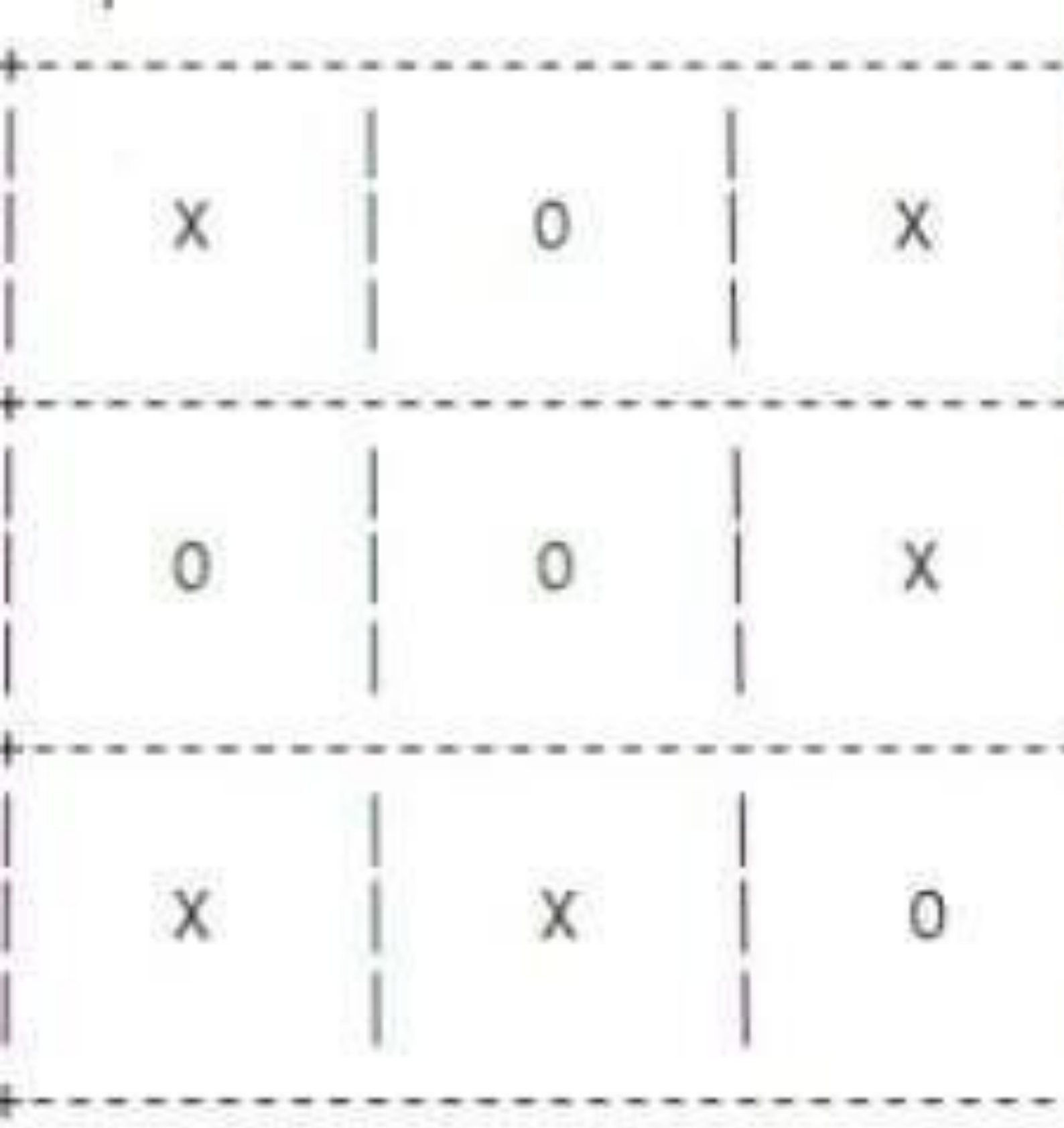
| | | |
|---|---|---|
| X | O | 3 |
| O | O | X |
| X | X | O |

| | | |
|---|---|---|
| X | O | X |
| O | O | X |
| X | X | O |


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 O 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 O X
O O X
X X O

24/11/2023 MA 24-11-23

② 8-puzzle problem using Breadth-first search

```
def bfs(src, target):
    queue = []
    queue.append(src)
    #to keep track of already visited combination
    exp = []
    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source) # insert the already visited combination
        print(source) #output screen
        if source == target:
            print("Success")
            return
        #calculate the possible moves in the puzzle
        poss_moves_to_do = []
        poss_moves_to_do = possible_moves(source, exp)

        for moves in poss_moves_to_do:
            if move not in exp and move not in queue:
                queue.append(move)
```

index

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

2 3 4

```
def possible_moves(state, visited_states):
    b = state.index(0) // 6

    #direction array consists the moves that can be made
    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')
    poss_moves_it_can = []
```

```
for i in d:        d['u','r']
    poss_moves_it_can.append(gen(state,i,b))
return [move_it_can for move_it_can in poss_move_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]
    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
source = [1,2,3,4,5,6,0,7,8]
target = [1,2,3,4,5,6,7,8,0]
bfs(source, target)  # call this to print the sequence & moves
```

Algorithm:

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
```

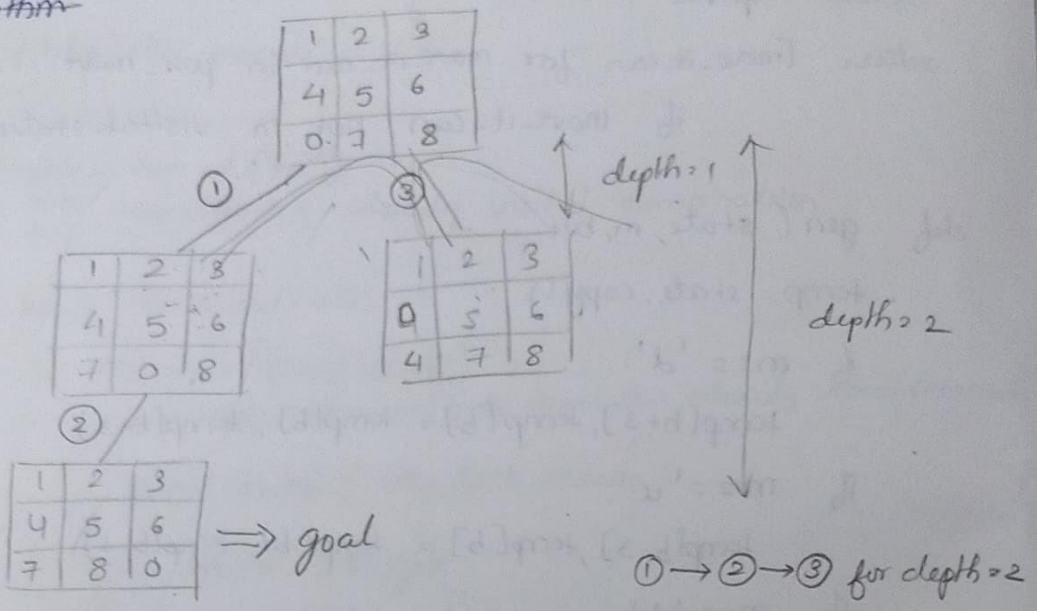
```
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8
-----------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
-----------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0
-----------
Success
```

8/12/2021 8/12/23

8-puzzle iterative deepening search algorithm

~~Algorithm~~

1)




① → ② → ③ for depth=2

Algorithm:

1) Initialize the initial state = [] and goal state for the 8puzzle

goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0] # 0 is the blank space

2) Set the depth = 1 and expand the initial state

The depth_limited_search (depth) is performed

```
if node.state = goal
    return node
else if depth=0:
    return None
else
    for neighbour in get_neighbours(state):
        child = puzzlenode (neighbour, node)
        result = depth_limited_search (depth-1)
        if result == True;
            return result
```

3) After one iteration where depth=1, increment the depth by 1 and perform depth_limited_search again

4) Here get_neighbours will generate the possible moves ~~in the same level~~ by swapping the '0' tile

5) The path traversed is printed or to reach the goal state

Code:

```
def id_dfs(puzzle, goal, get_moves):
    import itertools

    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!! It is possible to solve 8 puzzle problem")
    print("Path:", route)
else:
    print("Failed to find a solution")
```

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]]

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

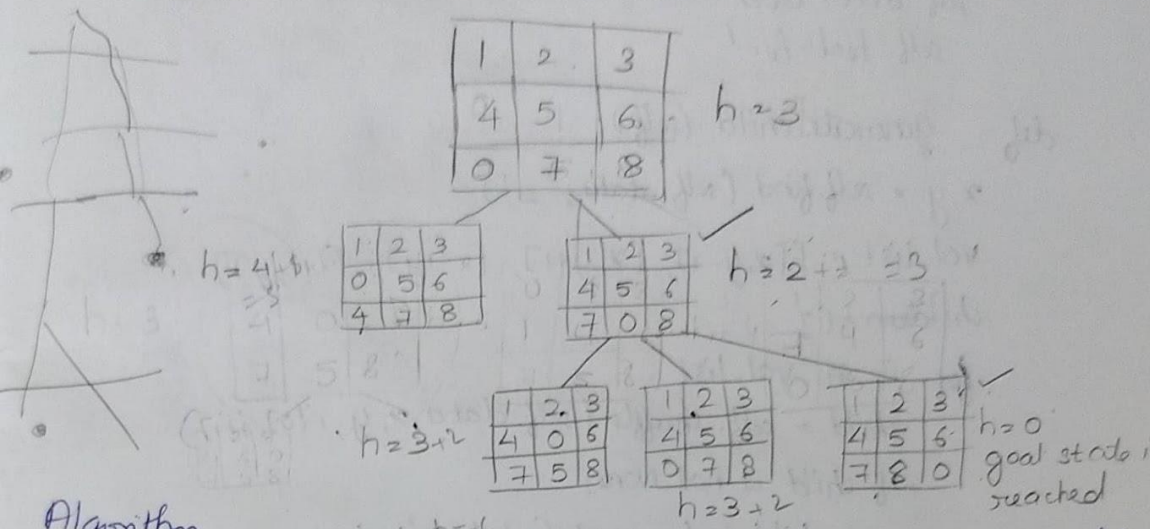
8/12/2023

# Solving 8-puzzle using A* algorithm

goal

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

h=0 (heuristics, number of misplaced tiles)




Algorithm

1) Create the initial state and goal state for the problem
In A* the heuristics is considered, lower heuristic node is considered in each step.
fvalue = hvalue + pathcost

2) Initially expand the node, find the location of empty tile and generate the nodes.
Calculate the heuristic function value
$f(x) = h(x) + g(x)$
↙ misplaced tiles. ↘ depth from starting node (path cost)

3) Maintain two list namely 'open' and 'close' the explored path, with lowest heuristics value are stored in 'open'
The nodes (states) generated are stored in the open list, sort using the f(x) values
The explored nodes are stored in close list, and removed from open.

4) The goal is reached when $h(x)=0$, implies that all the tiles are in the correct position

Code:

```
def process

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 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):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
```

```
            temp.append(t)
        return temp

    def find(self, puz, x):
        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):
        self.n = size
        self.open = []
        self.closed = []

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

    def f(self, start, goal):
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        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):
        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)
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(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]
self.open.sort(key = lambda x: x.fval, reverse=False)

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

Output:

Enter the start 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 _

Enter the goal matrix

1 2 3
4 5 6
7 8 _

```
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
```

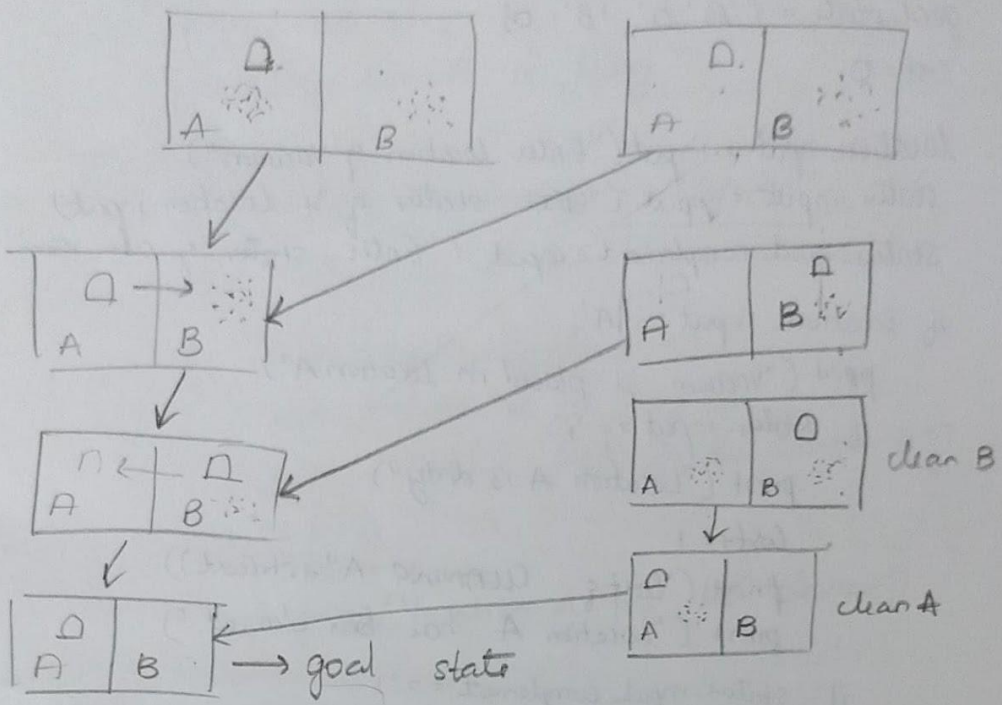
```
1 2 3
4 5 6
7 _ 8
```

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

2/12/2023

22-12-23

→ Vaccumm cleaner agent




Algorithm

1) Initialize the starting and goal state, the goal is to clean both rooms A and B ~~and tmax~~

2) If status = Dirty then clean

~~else if~~ location = A ~~then return right~~

~~else if~~ location = B ~~then return left~~

else if location = A and status = clean then return right

else if location = B and status = clean the return left

else exit

3) If both the locations are clean the vaccumm cleaner is done with it's task

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")

    if location_input == 'A':
        print("Vacuum is placed in location A")
        if status_input == '1':
            print("Location A is dirty")
            cost += 1
            print("Cost for CLEANING A" + str(cost))
            print("Location A has been cleaned.")
        if status_input_complement == '1':
            print("Location B is dirty")
            print("Moving right to the location B")
            cost += 1 #RIGHT

            cost += 1 # SUCK
            print("Location B has been cleaned.")
        else:
            print("No action" + str(cost))
            print("Location B is already clean")
    if (status_input == '0'):
        print("Location A is already clean")
        if status_input_complement == '1':
            print("Location B is Dirty")
            print("Moving RIGHT to the location B")
            cost += 1
            print("Location B has been cleaned")
        else:
            print("No action" + str(cost))
            print(cost)
            print("Location B is already clean")
```

```
else:
    print("Vaccumm is placed in location B")
    if status_input=='1':
        print("Location B is Dirty")
        cost+=1
        print("Cost for cleaning "+str(cost))
        print("Location B has been cleared.")
        if status_input_complement=='1':
            print("Location A is Dirty")
            print("Moving left to the Location A")
            cost+=1
            cost+=1
            print("Location A has been cleared")
    else:
        print(cost)
        print("Location B is already clean")
        if status_input_complement=='1':
            print("Location A is Dirty")
            print("Moving LEFT to the Location A")
            cost+=1
            cost+=1
            print("Cost for SUCK "+str(cost))
            print("Location A has been cleared")
        else:
            print("NO action "+str(cost))
            print("Location A is already clean")
    print("GOAL STATE")
    print(goal_state)
    print("Performance Measurement"+str(cost))
print("0 indicates clean and 1 indicates dirty")
vacuum_world()
```

→ Output:

```
Enter Initial Location of Vacuum (A/B~~etc~~) :B
Enter status of each room (1- dirty, 0- clean):
Status of Room 1
```

```
0 indicates clean and 1 indicates dirty
Enter Location of Vacuumb
Enter status of b1
Enter status of other room1
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

1/12/2023

Knowledge base entailment

Entailment refers to the logical relationship between a kb (a set of statement or rules) and a query.

If kb entails a statement, it means that whenever the statements in the kb are true, the given query must also be true

$KB \models Q$

Knowledge base resolution

The resolution rule involves taking two clauses that contain complementary literals and resolving them to produce a new clause

Knowledge base entailment

Kb

- If it's raining (p) then the ground is wet
- If the ground is wet (q), then the plants will grow (r)
- It's not the case that plants will grow. (not r)

Query is whether it's raining

not r - r are complement of each other

The ground is wet

which means that it is raining, because the ground is wet.

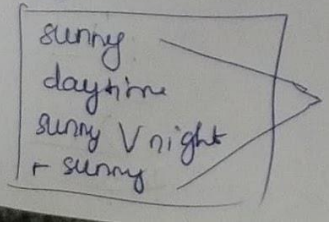
'p' entails the query

Knowledge base resolution

Input a kb and an expression, negate the expression add it to kb and find a contradiction, if contradiction is found, the negated statement is false hence the original statement is true

Is it sunny? sunny = TRUE? Prove sunny

kb




contradiction

r sunny = FALSE

sunny = TRUE

Code:

→ Knowledge_Base entailment:

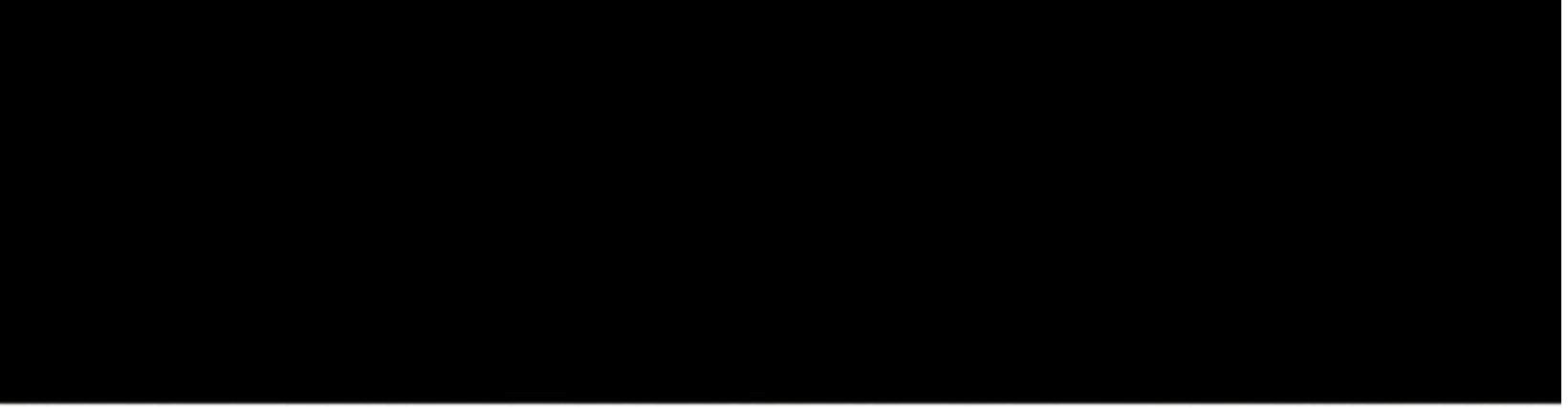
```
from sympy import symbols, And, Not, Implies, satisfiable
def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')

    knowledge_base = And(
        Implies(p, q),
        Implies(q, r),
        Not(r)
    )

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

Output:

```
Knowledge Base: r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```



```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

Knowledge_base_resolution

Code:

```
import re
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    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 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

split_terms('~PvR')

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

```
def resolve(rules, goal):
    temp = rules.copy()
    temp+ = [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given'
    step[negate(goal)] = '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(goal, 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(goal)} is assumed as
true. Hence, {goal} is true."
                                return steps
```

```
        elif len(gen) == 1:
            clause = [f'{gen[0]}']
        else:
            if contradiction(goal, 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.\
InA contradiction is found when {negate(goal)} is assumed
as true. Hence, {goal} 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
```

~~Output~~

```
rules = 'RV~P R V ~Q  ~R V P ~R V Q'
goal = 'R'
main(rules, goal)
```

Output:

| Step | Clause | Derivation |
|---|---|---|
| 1 | R V ~P | Given |
| 2 | R V ~Q | Given |
| 3 | ~R V P | Given |
| 4 | ~R V Q | Given |
| 5 | ~R | Negated conclusion |
| 6 | | Resolved R V ~P and ~R V P to R V ~R, null |

A contradiction is when ~R is assumed true, Hence R is true

```
rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
main(rules, '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.
```

12/1/2024

Unification

Eg: Knows (John, x) Knows (John, Jane)

{x/Jane}

Step1: If term1 or term2 is a variable or constant, then:

a) term1 or term2 are identical

return NIL

b) Else if term1 is a variable

if term1 occurs in term2

return FAIL

else

return {(term2/term1)}

c) else if term2 is a variable

if term2 occurs in term1

return FAIL

else

return {(term1/term2)}

d) else return FAIL

Step2: If predicate(term1) ≠ predicate(term2)

return FAIL

Step3: number of arguments ≠

return FAIL

Step4: set(SUBST) to NIL

Step5: For i=1 to the number of elements in term1

a) Call unify (ith term1, ith term2)

put result into S

b) S= FAIL

return FAIL

c) If S ≠ NIL

a. Apply S to the remainder of both L1 and L2

b. SUBST = APPEND (S, SUBST)

Step6: Return SUBST

Unification - code:

```
import re

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

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)
    for index, val in enumerate(attributes):
        if val==old:
            attributes[index]=new
    predicate = getInitialPredicate(exp)
    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):
    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: return False
    if isConstant(exp1):
        return [(exp1, exp2)]
    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 do not match. Cannot be unified")
        return False

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

→ exp1 = "knows(x)"
exp2 = "knows(Richard)"
Output:
[('x', 'Richard')]

→ exp1 = "knows(A,x,y)"
exp2 = "knows(y, mother(y))"
Output:
Substitutions:
False

→ exp1 = "knows(A,x)"
exp2 = "k(y, mother(y))"
substitution = unify(exp1, exp2)
Output:
Substitutions
False

[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)
```

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

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

12/1/2024

FOL to CNF conversion

Step1: remove_implications

return "Or (Not ({part[0]}), {part[1]})"

Step2: apply_demorgans_law

formula.replace('Not (And', 'Or(Not)').replace('Not
'And (Not')

Step3: distribute_quantifiers

formula.replace('Forall(', 'And(').replace
('Exists(', 'Or(')

Step1: Create a list of SKOLEM_CONSTANTS

Step2: Find ∀, ∃

If the attributes are lower case, replace them with a skolem constant

remove used skolem constant or function from the list

If the attributes are both lowercase and uppercase replace the uppercase attribute with a Skolem function

Step3: replace <=> with '='

transform — as $P \equiv Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Step4: replace => with '-'

Step5: Apply demorgan's law

replace ~[

as ~P&~Q if (& was present)

replace ~[

as ~P|~Q if (& was present)

replace ~~ with ''

FOL to CNF

```
print(Skolemization(fol_to_cnf("animal(y) <=> loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x [∀y [animal(y) => loves(x,y)]]
                    => [∃z [loves(z,x)]]")))
print(fol_to_cnf("[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(∼,G(x))] | [loves(F(x),z)]

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

```
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃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)
```

19/1/2024 19-1-24

→ Forward chaining

1) Input the knowledge base and the query

2) for i in KB:
   if i == query return True
   if ⇒ in i:
      split lhs and rhs part
      if lhs ~~and rhs~~ in KB:
         add rhs to KB
   return false

3) To remove variables
   ~~for~~ if i.lower():
      ~~i.replace~~ replace the variable with constants

Example:

KB

king(x) & greedy(x) ⇒ evil(x)
king(John)
greedy(John)
king(Richard)

Query
evil(x)

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}
```

```
( f"{','.join([constants.pop(0) if is Variable(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)
        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}')

kb = KB()
kb.tell('king(x) & greedy(x) => evil(x)')
kb.tell('king(John)')
kb.tell('greedy(John)')
kb.tell('king(Richard)')
kb.query('evil(x)')
```

Output:

```
Querying evil(x):
    1. evil(John)
```

```
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()
```

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

```
kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

```
Querying evil(x):
        1. evil(John)
```