PAGE NO :
DATE :

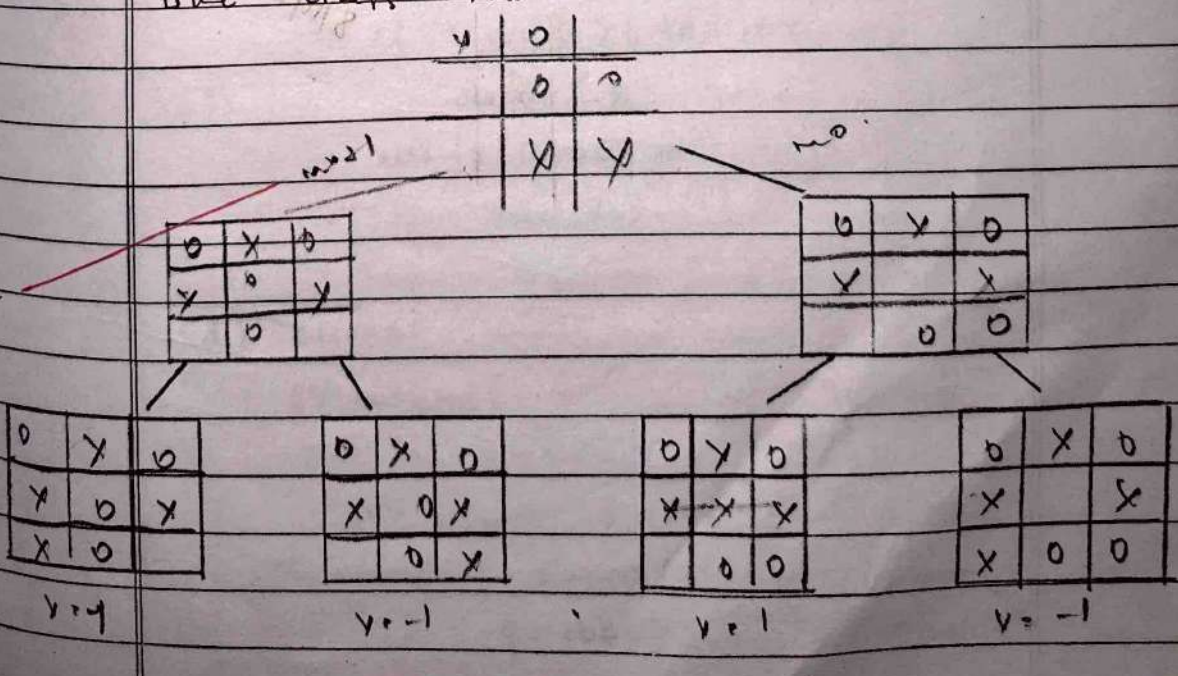
Tic Tak Toe Game

In order to be able to create game AI that attempts to win the game, we attach a numerical value to each possible end result To the board position where X has a line of three so that may wins with three O's in a row we attach the value +1.

Game playing in AI is an active area of Research and has many practical applications, including game development and develop more effective decision making systems.

Mini max search procedure

It is a depth-first depth limited search procedure. It is used for games like chess and tik tac toe.



Algorithm

function minimax (board, depth, isMax) {
if current board state is a final state,
return.

if is Maximizing Player.
best val = -INF
for each move in board
value = minimax (board, depth+1, false)
best val = max (best val, val)
return best val.
else
best val = +INF
for each move in board
value = minimax (board, depth+1, true)
best val = min (best val, value)
return best val.

8/11/2023

PAGE NO :
DATE :

PROGRAM - 1.

Tic Tac Toe

```
import math
import copy
    X = "X"
    O = "O"
    EMPTY = NONE
def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]
def player(board):
    count_O = 0
    count_X = 0
    for y in [0, 1, 2]:
        for x in board[y]:
            if x == O:
                count_O = count_O + 1
            elif x == X:
                count_X = count_X + 1
    if count_O < count_X:
        return O
    elif count_X < count_O
        return X.

def actions(board):
    possible_moves = set()
    for i in [0, 1, 2]:
        for j in [0, 1, 2]
            if board[i][j] == EMPTY:
                possible_moves.add((i, j)).
```

```
def result (board, action):
    i = action[0]
    j = option[i]
    if type (action) == list:
        action = [i, j]
    if action in actions (board):
        if player (board) = X:
            board[i][i] = X
        elif player (board) == 0:
            board[i][j] = 0
        return board.

def winner (board)
    if (board[0][0] == board[0][1] == board[0][2]
    if (board[1][0] == board[1][1] == X
    or board[2][0] == board (2][1] == X);
    return X

    if (board [0][0] == board [0][1] == 0 or
    or board[1][0] == board [1][1] == 0 or
      bord[2][0] == board [1][2] == board[2][2].

    for i in [0.1.2.
        s2=[]
        for j in [0.1.2]
            s2.append (board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0].

    strike D=[]
    for i in [0,1],
        strike D.append (board[j][i]).
```

```
if (strike[0] == strike[1] == strike[2]):
    return strike[0]
if (board[0][2] == board[1][1] == board[2][0]):
    return board[0][2].
return none.

def terminal(board):
    Full = True;
    for i in [0,1,2]:
        for j in board[i]:
            if j is None:
                Full = False.

if Full:
    return True.
if (winner(board) in not None):
    return True.
    return False.
def utility(board)
    if (winner(board) == X):
        return 1
    elif (winner(board) == 'O':
        return -1
    else
        return 0

def minimax-helper(board)
    is MaxTurn = True
    if player(board) == X else False
    if terminal(board)
        return utility(board)
    scores=[]
```

```
for move in actions (board)
    result (board. move)
    scores. append ( minimax_helper (board)).
    board. [move (0)]. [move (1)] = EMPTY.
return max (scores) if is maxTurn else
def minmax (board).
    is maxTurn = True. if player (board)
    bestScore = None.
if is maxTurn:
    bestScore = math. inf.
    for move in actions (board)
      result (board. move)
      score = minimax_helper (board).
    board [move [0]] [move [1]] = EMPTY.
  if (score > score):
    best score = score.
    best move = move.
  return best move.
else.
  bestScore = math. inf.
  for move in actions (board);
    result (board, move).
    score = minimax. helper (board)
  board [move [0]] [move [1]] = EMPTY.
  if (score is best score).
      best score = score
      best move = move.
    return best move.
def printboard (board)
    for Row in board.
      print (row).
game_board = initial state()
```

print ("Initial_board").
print ("Initial_board")
printboard (game_board).

While not Terminal (game_board):
    if player (gameboard) == 'X'.
user_input = input ("Enter your name (
                    row, column): ")
row, column = map (int user_input.).
result (game_board, row, col).

else
    print ('AI is making a move').
    move = minimax (copy.deepcopy (game board))
    result (game_board, move).

print ("Current Board").
move = minimax (copy.deepcopy (game board))
result (game board, move).

if winner (game_board) is not None:
    print ("The winner is:" {winner (game)}").
else
    print ("It is a tie").

PROGRAM - 2.

```
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("sucess").
            return
        poss_moves_to_do = []
        pos_moves_to_do = possible_moves(c.exp)

for move in poss_moves_to_do:
        if move not in exp & move not

int queue:
        queue.append(move)
def possible_moves(state, visited_states):
    # indx of empty spot
    b = state.indx

    # 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']

pos_moves_it_con=[]

    for i in d:
pos_moves_it_con.append(gen(state,i,b))
    return [move_it_con for move_it_con in
    pos_moves_it_con if move_it_con 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-3], temp[b] =
temp[b], temp[b-1]
```

PAGE NO :
DATE :

if m==i :

temp[b+1], temp[b] =
temp[b], temp[b+1]

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

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

bfs(src, target)

O/P - ?

PROGRAM-3.

```
def vaccum_world():
Dirty
    goal_state = {'A':'0','B':'0'}
    cost = 0

    location_input = input("Enter location of
    vaccum")
    status_input = input("Enter status of "
    + location_input)
    status_input_complement = input("Enter
    status of other room")
    print("Initial location condition" + str(goal))

    if location_input == 'A':
     print("Vaccum is placed in location A").

    if status_input == '1':
     print("Location A is Dirty.")

     goal_state['A'] = '0'
     cost += 1
     print("cost for cleaning A "+ str(cost))
     print("Location A has been cleaned.").
        if status_input_complement == '1':
     print("Moving right to the B")
     print("Moving for cost Right.");
```

goal_state['B'] = '0'
cost1 = 7.
else:
print("Location B is already clean.")
if status_input == '0':
print("Location A is already clean")
if status_input_complement == '1':
print("moving Right to the B")
cost += 1.
print("cost for moving Right" +str(cost))

goal_state['B'] = '0'
cost += 1.
print(cost)
print("Location B is already clean.")
else:
print("Vellum 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))
print("Location B has been cleaned").

if status_input_complement == '1':
print("Location A is Dirty").
goal_state[A] = 0.
cost += 1.
print("cost of cleaning A" +str(cost))
print("Location A has ").

if status input == 0:
print("Location A is already clean").

```
print("Location B is Dirty.")
print("Moving Right to the loc B")
cost+=1
print("cost for moving Right")
goal_state['B']='0'
cost+=1
print("No action B as cleaned.")
```

else

```
print("No action "+str(cost))
print(cost)
print("location B is already clean")
```

else

```
print("Vacuum is placed in loc B")
if status_input=='1':
print("Location B is dirty.")
goal_state['B']='0'
cost+=1
print("cost for cleaning "+str(cost))

if status_input_complement=='1':
print("location A is Dirty")
print("moving LEFT to the loc A")
cost+=1
print("cost for moving left"+str(cost))
goal_state['A']='0'
cost+=1
print("cost for suck")
print("location A has been cleaned.")
```

else:

```
print (cost)
print ("location B is already clean.")
if status_input_complement == '1'
print ("location A is Dirty.").
print ("moving left").
cost += 1.
print ("cost for moving left" + str(cost)).
goal_state['A'] = '0'
cost += 1.
print ("cost for suck :" ).
print ("location A has been cleaned.").
else:
print ("No action" + str(cost))
print ("location A is already clean.").

print ("Goal State").
print (goal_state).
print ("Performance Measurement :")
```

Vacuum world is.

OUTPUT

Enter location of Vacuum A
Enter status of A
Enter status of other room B.
Initial location condition {A' 0', B: '0'}.
Vacuum is placed in location B
0
Location B is already clean.
No action.

PROGRAM-11.

IDDSF

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

Visited : 1 2 3
4 5 6
7 0 8

up ↑

temp[b-3], temp[b]
= temp[b-3], temp[b]

↓ down.

temp[b+3], temp[b]
= temp[b] [temp[b+3]]

1 2 3
0 4 5
6 7 8

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

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

PAGE NO :
DATE :

PROGRAM - 5.

A*

```
def print_b (src)
    state = src.copy()
    state [state.index (-1)]
    print (
        f" "
    { state [0] } { state [1] } { state [2] }
    { state [3] } { state [4] } { state [6] }.
    { state [6] } { state [7] } { state [8] }.
        " "
    )
def h (state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target [i]:
            count = count +
        return count.
def astar (state, target):
    states = [src]
    g = 0
    visited_states = []
while len (states):
    print (f"move: {g}").
    moves = []
    for state in states:
        visited. states. append (states)
    print_b (states)
        if (state == target)
```

20-12-23

PAGE NO:
DATE:

=> Create a knowledgebase using propositional logic and show that the given entails the knowledge base or not.

¬ not
∨ or
∧ and

$(p \vee q) \wedge (\neg r \vee p)$.

Expression Result => (p or q) and (not r or p)
p and R.

p	q	r	Exp (KB)	Query (p, r)
T	T	T	T	T
T	T	F	T	F
T	F	T	T	T
T	F	F	T	F
F	T	T	F	F
F	T	F	F	F
F	F	T	F	F
F	F	F	F	F

10

For example :-

=> (T or T) & (¬T or T) : T

=> (F or T) & (¬T or F) : F

20/12/23

PAGE NO:
DATE:

```
expression_result = ((not q or not p or r) and
                     (not q and p) and q)
return expression_result.

def generate_truth_table.
  print ("q | p | r | Expression (KB) | Query (r)").
  print ("... | ... | ... | ... | ...").

for q in [True, False]
  for p in [True, False]:
    for r in [True, False]:
      expression result = evaluate_expression (q,p,r)
      query_result = r
print (f" {q} | {p} | {r} | expression_result ").

def query_entails_knowledge ():
  for q in [True, False]:
    for p in [True, False]:
      for r in [True, False]:
        expression_result = evaluate_expression
                                (q, p, r).
  query_result = r
  if expression_result & not query_result
      return false.
  return true.

def main().
  generate_truth_table()
    if query_entails_knowledge ():
    print ("Query entails knowledge.")
    else
      print ("Query does not entail").

if __name__ == "main"
```

27/12/23.

PROGRAM-7

Create a knowledgebase using propositional logic and prove the given query using resolution.

Expression: $R \vee \neg P \quad R \vee \neg Q \quad \neg R \vee P \quad \neg R \vee Q$

$(R \text{ or not } P) \quad (R \text{ or not } Q) \quad (\text{not } R \text{ or } P) \quad (\text{not } R \text{ or } Q)$

Step	Clause	Derivation
1	$R \vee \neg P$	G
2	$R \vee \neg Q$	G
3	$\neg R \vee P$	G
4	$\neg R \vee Q$	G.
5	$\neg R$	Negate conclusion.
6		Resolved $R \vee \neg P$ & $R \vee P$ to $R \vee R$, which is R

A contradiction is found when $\neg R$ is assumed as true. Hence R is true.

PROGRAM.

```
!pip install z3-solver
from z3 import Implies, Not, Bool, solver

#Define z3 Boolean symbols.
p, q, r = Bool('p'), Bool('q'), Bool('r')

# knowledge base (KB)
KB = Implies(p, q) & Implies(q, r) & Not(p)

# Q.
query = r

def prove_query_with_resolution(
        knowledge_base, query):
    s = solver()
    s.add(Not(Implies(knowledge_base, query)))
    result = s.check()
    return result == sat.

if proof_result:
    print("The query is proved to be true").

else
    print("The query is not proved to be true").
```

```
import re.

def main (rule, goal)
  rules = rules.split (',')
  steps = resolve (rules, goal)
  print ('In step \t clause \t Derivation\t')
  print ('-' * 30)
   i=1
   for step in steps
  print (f'{i} . \t {step2}\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_term (rule):
     exp = ('~' * [PQRS])
    Terms = re.findall (exp, rule)
     return Terms

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

  def resolve (rule, goal)
     temp = rules.copy ()
```

```
steps = dict()
for rule in temp:
    steps[rule] = 'Given'
steps[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]}'):
                            temp.append(f'{gen[0]} v {gen[1]}')
                            steps[':'] = f"Resolved {temp[i]} &
{temp[j]} to {temp[-1]} which is in
turn null. A contradiction is formed
when {negate(goal)} is assumed as
true."
        for clause in clauses:
            if clause not in temp:
                temp.append(clause)
                steps[clause] = f'Resolved from {temp[i]}
& {temp[j]}.'
```

temp[j].

j = (j+1)+n

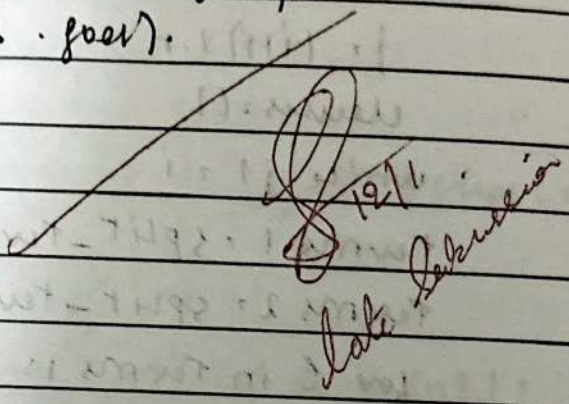
i+=1

return steps.

rules = input ("Enter KB : ")

goal = input ("Enter query")

main (rules, goal).



PAGE NO :
DATE :

10-1-24

Implement unification in FOL.

Condition

* Predicate symbols must same
* Number of arguments in both expressions must be identical.
* Unification will fail if there are two similar variables present in the same expression.

```
def unify(expr1, expr2):
    # Split expressions into function arguments
    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 2)
    if func1 != func2:
        print("Expression cannot be unified Different functions.")
        return NONE
    args1 = args1.rstrip(')').split(',')
    args2 = args2.rstrip(')').split(',')
    substitution = {}
    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower():
            substitution[a1] = a2
        elif a1.islower() and not a2.islower():
            substitution[a1] = a2
        elif a1 != a2:
            print("Expressions cannot be unified incompatible argument")
```

return None

return substitution

```
def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return substitution

# Main program
if __name__ == "__main__":
    # Sample input
    expr1 = input("Enter the first expression: ")
    expr2 = input("Enter the second expression: ")
    # Unify expressions
    substitution = unify(expr1, expr2)

    # Display result
    if substitution:
        print("The substitutions are:")
        for key, value in substitution.items():
            print(f"{key} / {value}")

    # Apply substitution to original expr
    expr1_result = apply_substitution(expr1, substitution)
    expr2_result = apply_substitution(expr2, substitution)

    print(f"Unified expression 1: {expr1_result}")
    print(f"Unified exp 2: {expr2_result}")
```

Le show

o/p to

17-1-24.

PAGE NO:
DATE:

→ Convert the given first order logic statement into conjuctive normal form.

Input : $\forall x$ food(x) $\Rightarrow$ likes (John, x)

Step 1 : i) eliminate $\Rightarrow$ and rewrite

$\forall x \neg$ food(x) $\vee$ likes (John x)

Step 2: Move negation '$\neg$' inwards.

Step 3: Standardize variable apart by renaming them

Step 4: Standardize each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variable

$\forall x \neg$ food(x) $\vee$ likes (John, x)
where x is the new skolem constant

Step 5: Drop universal quantifiers

$\neg$ food(x) $\vee$ likes (John, x)

Step 6: De morgen application:
(Here we don't use De morgen law)

Example 2: $\forall x [\exists z [loves(x,z)]]$

→ No "~" or negation in FOL statements so continue.

$\forall x [\exists z [loves(x,z)]]$

→ Step 4: Skolemize - $z = B(x)$.

new skolem function: B.

$\forall x [\exists x [loves(x, B(x))]]$

→ Step 5: Drop universal quantifiers.

$[loves(x, B(x))]$

→ Code:

```
def getAttributes(string):
    expr = '([~])+'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if (x)]

def getPredicates(string):
    expr = '[a-zA-Z]+([A-Za-z,])+'
    return re.findall(expr, string)

def Demorgen(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
```

24-1-24

PAGE NO:
DATE:

Create a knowledge base of FOL and given query using forward reasoning.

FOL

① - Query criminal (x)

1. criminal (West)

All facts:

1. criminal (West)
2. weapon (M1)
3. hostile (NONO)
4. american (West)
5. enemy (NONO, America)
6. sells (West, M1, NONO)
7. owns (NONO, M1)
8. Missile (M1).

KB = KB()
kb.tell('missile(x)
kb.tell('Missile(x
kb.tell('enemy(x
kb.tell('american
kb.tell('owns
kb.tell('missile(x
kb.tell('american(x
kb.query('criminal(x)
kb.display

② Querying evil(x):

1. evil(John).

All facts

1. evil(John)
2. greedy(John).
3. king(John)
4. king(Richard)

KB = KB()
kb.tell('king(x)&
kb.tell('king(John)
kb.tell('greedy(
kb.tell('king(Rich
kb.query('evil(x)')

Hall

```
for i in self.implicands:
    res = i.evaluate(self)
    if res:
        self.facts.add(res)

def query(self,e):
    facts = set([f.exp for f in self.f])
    i = 1
    print(f'Querying {e}:')
    for f in facts:
        if f.factf).predicate == fact(e).predicate
        print(f'{i}: {f}')
        i += 1

def display(self):
    print("All facts:")
    for i, f in enumerate(set([f.exp for f
    print(f'{i+1}: {f}')
kb = KB()
kb.tell('missile(x) & weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x, America) & hostile
kb.tell('owns(NONO,M1)')
kb.tell('owns(NONO,M1)')
x) & sell(West, x, NONO)')
kb.tell('enemies(x) & weapon(y)
kb.display()
```