VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

BJ Keertana(1BM23CS059)

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

Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

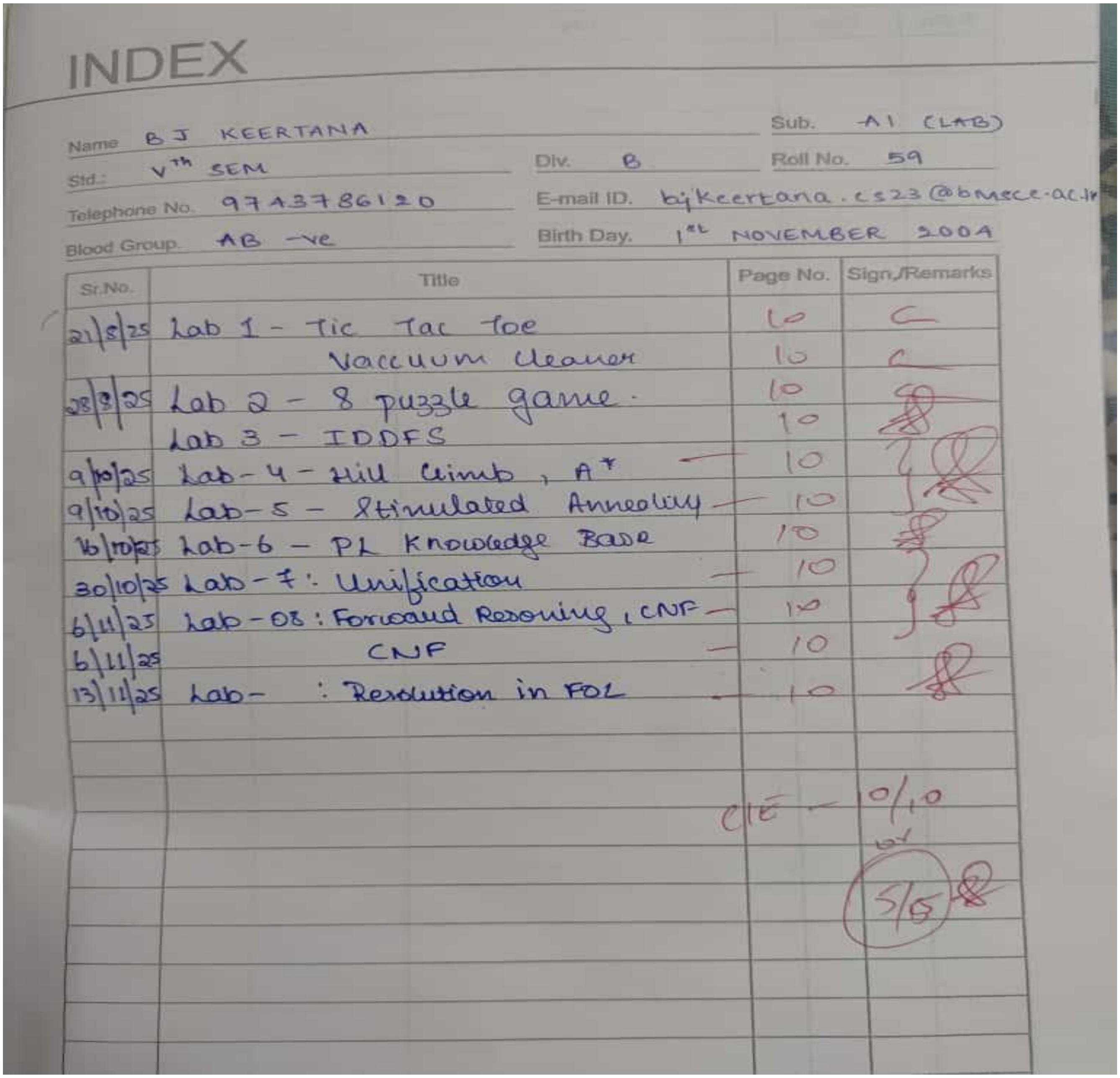
CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **BJ Keertana (1BM23CS059),** who is a 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. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/keertanabj/AI-lab-



INDEX

Name: B J KEERTANA
Sub.: AI (LAB)
Std.: V th SEM
Div.: B
Roll No.: 59
Telephone No.: 9743786120
E-mail ID.: bjkeertana.cs23@bmsce.ac.in
Blood Group.: AB -ve
Birth Day.: 1st NOVEMBER 2004

Sr.No.	Title	Page No.	Sign./Remarks
21/8/25	Lab 1 - Tic Tac Toe	10	
	Vaccuum Cleaner	10	
28/8/25	Lab 2 - 8 puzzle game.	10	
	Lab 3 - IDDFS	10	
9/10/25	Lab-4 - Hill Climb, A*	10	
9/10/25	Lab-5 - Stimulated Annealing	10	
16/10/25	Lab-6 - PL Knowledge Base	10	
30/10/25	Lab-7: Unification	10	
6/11/25	Lab-08: Forward Resoning, CNF	10	
6/11/25	CNF	10	
13/11/25	Lab- : Resolution in FOL	10	

CIE - 10/10

5/5

Program 1

Implement Tic –Tac –Toe Game Algorithm:

Lab-01

Tic-Tac-Toe.

Algorithm

Step 1: Create a 3x3 matrix and initialize all the 9 spaces NULL.

Step 2: Design create 3x3 matrix and initalize it to 0.

Step 3: Assign winning pattern = {(0,1,2) (0,3,6) (1,4,7) (0,4,8) (2,4,6) (2,5,8) (3,4,5) (6,7,8)}

Step 4: Assign X to computer & O to user

Step 5: ~~Design~~ Decide whether user or computer will start.

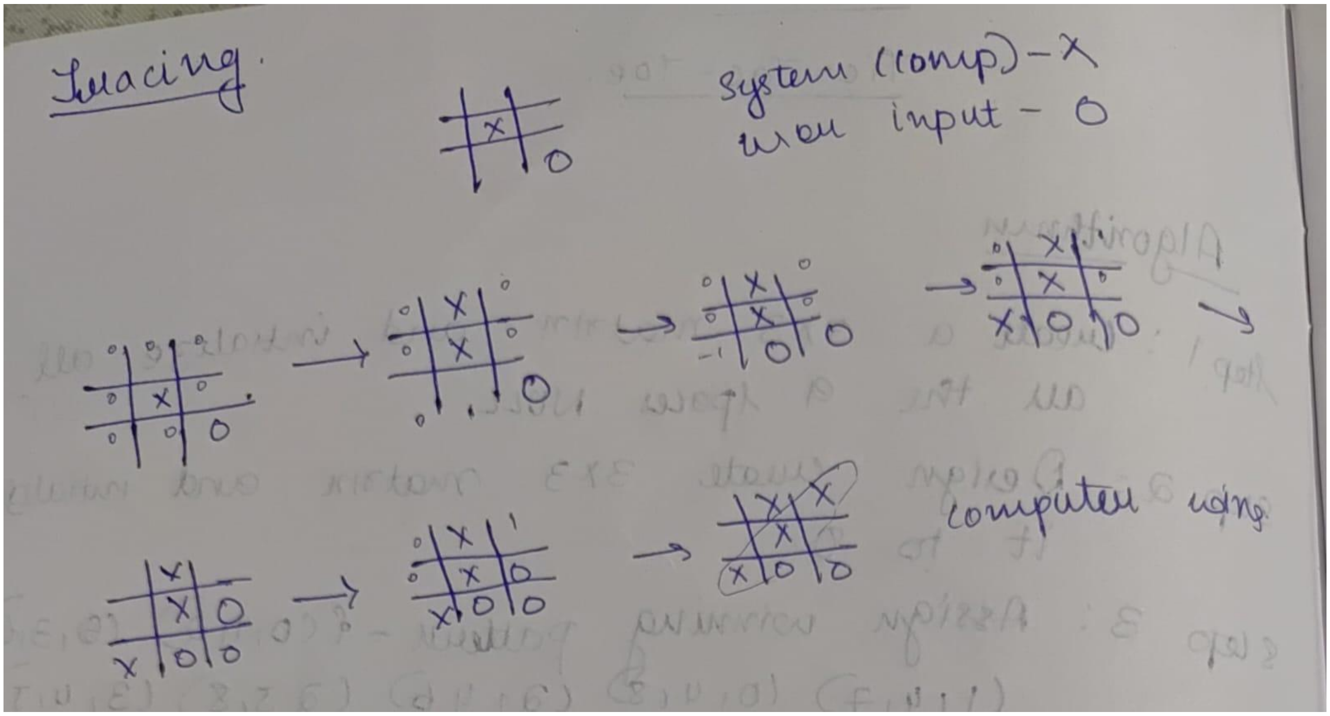
Step 6: If computer starts will be in any random space

Step 7: For each time a user or computer inserts it should check whether any O/X is meeting the winning pattern or not.

Step 8: If X is matching the winner pattern assign that position as 1, if O is matching then assign that position as -1 else 0

~~Step 9: Check the man value & insert if all are zero use -1 position or choose a pattern where the rest 2 element position in the pattern is empty.~~

Step 10: Repeat for 7 to 9 until a winner is found.

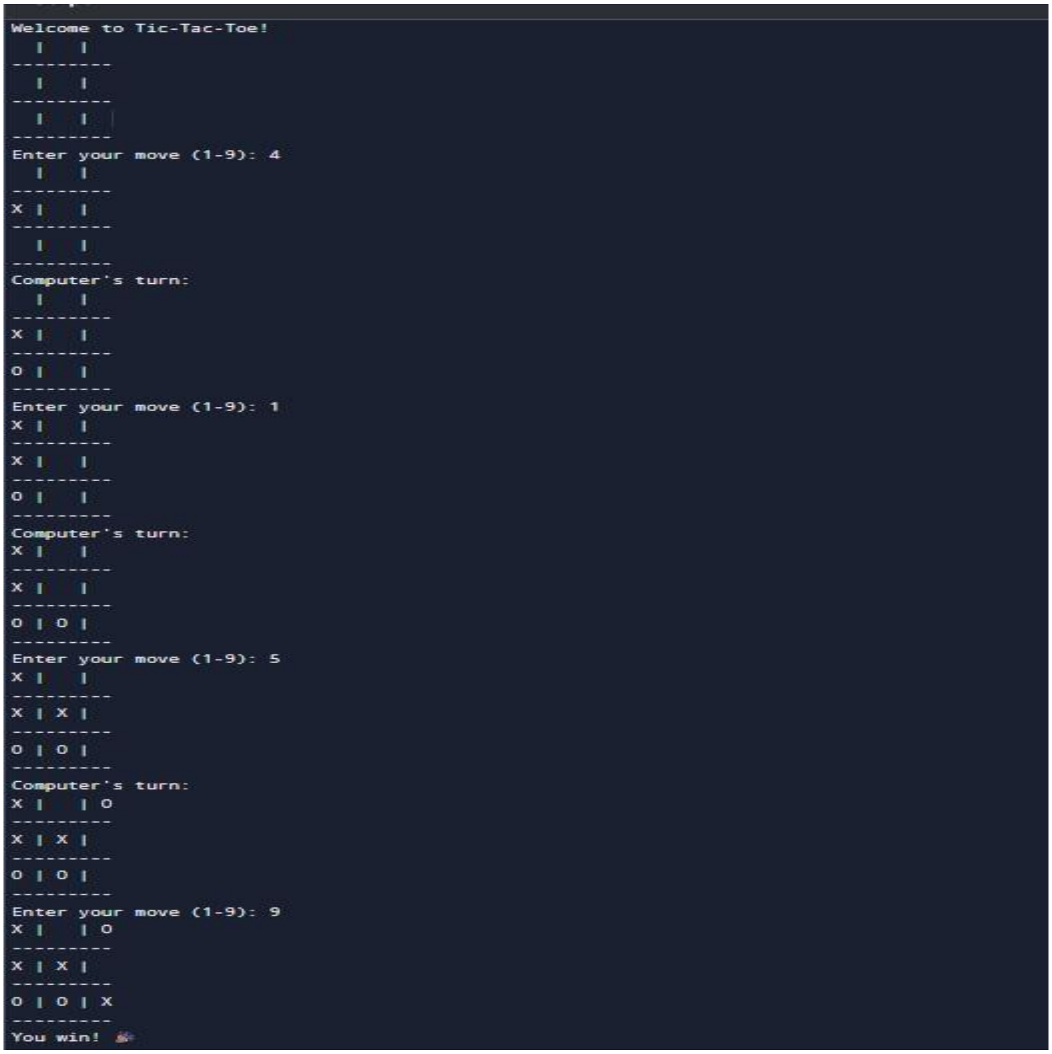


Code:

```
import random
board = [' ' for _ in range(9)]

def print_board():
print()     for i in
range(3):
        print(" " + " | ".join(board[i*3:(i+1)*3]))
if i < 2:
            print("---+---+---")
print() def
check_winner(player):
win_conditions = [
        [0,1,2], [3,4,5], [6,7,8],
        [0,3,6], [1,4,7], [2,5,8],
        [0,4,8], [2,4,6]
    ]
    for cond in win_conditions:          if
all(board[i] == player for i in cond):
            return True
return False def
is_full():
    return all(cell != ' ' for cell in board)
def player_move():     while True:
try:
```

```
            move = int(input("Enter your move (1-9): ")) - 1
if move < 0 or move >= 9:
                print("Invalid move. Choose between 1-9.")
elif board[move] != ' ':
                print("That spot is taken.")
else:
                board[move] = 'X'
break         except
ValueError:
            print("Please enter a valid number.") def
ai_move():
    empty_spots = [i for i, val in enumerate(board) if val == ' ']
    move = random.choice(empty_spots)
    board[move] = 'O'     print(f"System placed 'O' in
position {move+1}") def play_game():
print("Welcome to Tic Tac Toe!")
    print_board()     while
True:         player_move()
print_board()         if
check_winner('X'):
            print("Congratulations! You win!")
break         if is_full():            print("It's a
tie!")            break         ai_move()
print_board()         if check_winner('O'):
            print("System wins. Better luck next time!")
break         if is_full():            print("It's a tie!")
            break if __name__
== "__main__":
    play_game()
```

ScreenShots:



```
Welcome to Tic-Tac-Toe!
  |   |  
---------
  |   |  
---------
  |   |  
---------
Enter your move (1-9): 4
  |   |  
---------
X |   |  
---------
  |   |  
---------
Computer's turn:
  |   |  
---------
X |   |  
---------
O |   |  
---------
Enter your move (1-9): 1
X |   |  
---------
X |   |  
---------
O |   |  
---------
Computer's turn:
X |   |  
---------
X |   |  
---------
O | O |  
---------
Enter your move (1-9): 5
X |   |  
---------
X | X |  
---------
O | O |  
---------
Computer's turn:
X |   | O
---------
X | X |  
---------
O | O |  
---------
Enter your move (1-9): 9
X |   | O
---------
X | X |  
---------
O | O | X
---------
You win! 🎉
```

Program 2:

Solve 8 puzzle problems.
Algorithm:

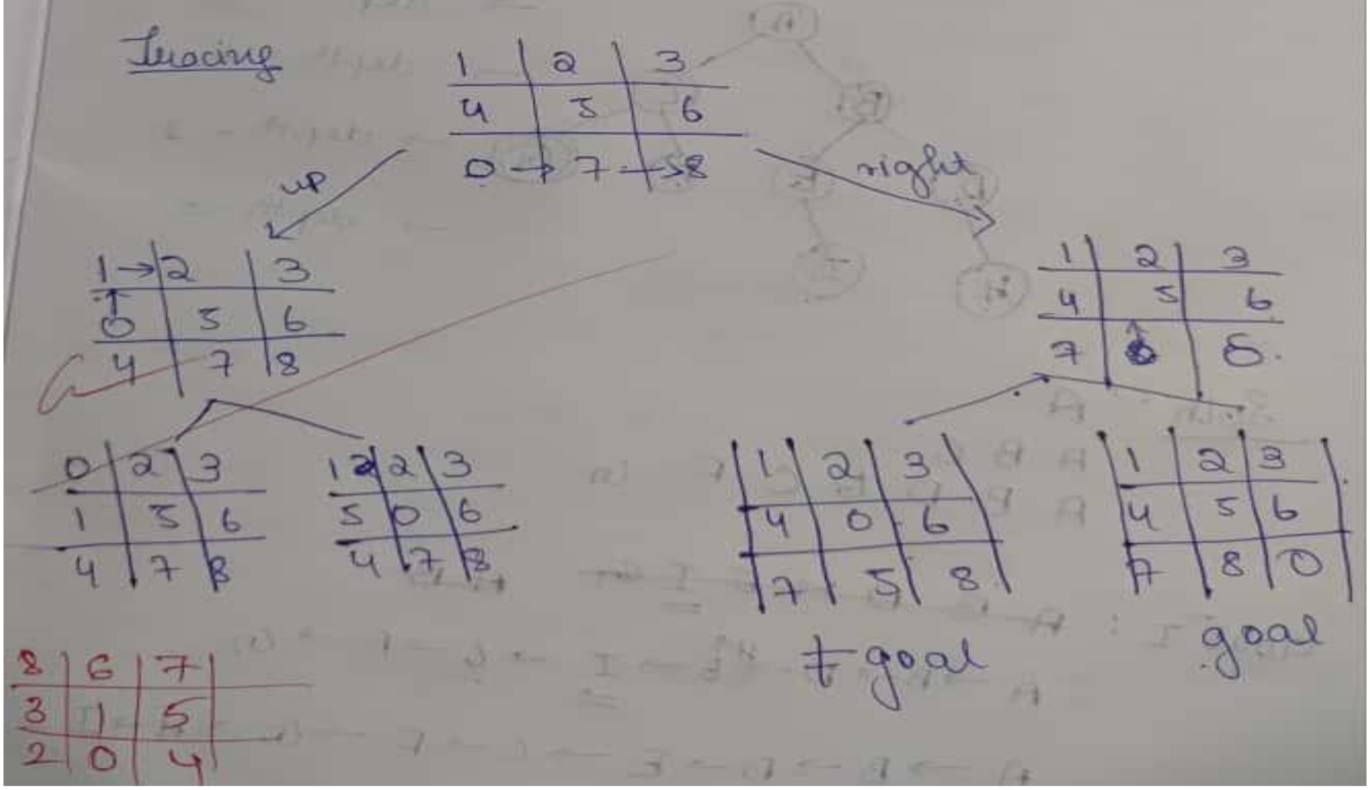
LAB 02

8 puzzle Game

Algorithm

Step 1: Start
Step 2: Initial empty queue.
Step 3: Loop until queue is empty
If state is final state stop
else,
move blank in all possible directions
& enque each state, mark as visited
and do not enque in the future
iterations
Step 4: If loop didn't break, then no possible
solution

Tracing



Code:

```
import copy def
print_board(board):
for row in board:
       print(' '.join(str(x) if x != 0 else ' ' for x in row))
print() def find_zero(board):     for i in range(3):
for j in range(3):             if board[i][j] == 0:
             return i, j def
is_solved(board):
   solved = [1,2,3,4,5,6,7,8,0]
   flat = [num for row in board for num in row]
   return flat == solved def
valid_moves(zero_pos):     i, j =
zero_pos     moves = []     if i > 0:
moves.append((i-1, j))     if i < 2:
moves.append((i+1, j))     if j > 0:
moves.append((i, j-1))     if j < 2:
moves.append((i, j+1))     return
moves def
correct_tiles_count(board):
   """Count how many tiles are in their correct position."""
count = 0     goal = [1,2,3,4,5,6,7,8,0]     flat = [num for
row in board for num in row]     for i in range(9):         if
flat[i] != 0 and flat[i] == goal[i]:
          count += 1     return count def
get_user_move(board):     zero_pos =
find_zero(board)     moves =
valid_moves(zero_pos)     movable_tiles =
[board[i][j] for (i,j) in moves]     print(f"Tiles
you can move: {movable_tiles}")     while True:
try:
          move = int(input("Enter the tile number to move (or 0 to quit): "))
if move == 0:                return None            if move in movable_tiles:
             return move
else:
             print("Invalid tile. Please choose a tile adjacent to the empty space.")
except ValueError:
          print("Please enter a valid number.") def
evaluate_move(board, tile):
   """Compare user move to all possible moves and tell if it's best/worst."""
zero_pos = find_zero(board)     moves = valid_moves(zero_pos)
movable_tiles = [board[i][j] for (i,j) in moves]     scores = {}     for t in
movable_tiles:
       temp_board = copy.deepcopy(board)
make_move(temp_board, t)         scores[t] =
```

```
correct_tiles_count(temp_board)     user_score =
scores[tile]     best_score = max(scores.values())
worst_score = min(scores.values())
   if user_score == best_score and user_score ==
worst_score:
      return "Your move is the only possible move."
elif user_score == best_score:
      return "Great! You chose the best move."
elif user_score == worst_score:
      return "Oops! You chose the worst move."
else:
      return "Your move is neither the best nor the worst." def
make_move(board, tile):
   zero_i, zero_j = find_zero(board)     for
i, j in valid_moves((zero_i, zero_j)):
if board[i][j] == tile:
         board[zero_i][zero_j], board[i][j] = board[i][j], board[zero_i][zero_j]
return def main():     board = [        [1, 2, 3],
      [4, 0, 6],
      [7, 5, 8]
   ]
   print("Welcome to the 8 Puzzle Game!")
print("Arrange the tiles to match this goal state:")
print("1 2 3\n4 5 6\n7 8  ")     while True:
      print_board(board)
if is_solved(board):
         print("Congratulations! You solved the puzzle!")
break
      move = get_user_move(board)
if move is None:
         print("Game exited. Goodbye!")
break
      feedback = evaluate_move(board, move)
     print(feedback)
make_move(board, move) if
__name__ == "__main__":
main()
```

ScreenShot:

```
Output
Welcome to the 8 Puzzle Game!
Arrange the tiles to match this goal state:
1 2 3
4 5 6
7 8
1 2 3
4   6
7 5 8

Tiles you can move: [2, 5, 4, 6]
Enter the tile number to move (or 0 to quit): 5
Great! You chose the best move.
1 2 3
4 5 6
7   8

Tiles you can move: [5, 7, 8]
Enter the tile number to move (or 0 to quit): 8
Great! You chose the best move.
1 2 3
4 5 6
7 8

Congratulations! You solved the puzzle!

=== Code Execution Successful ===
```

<u>Program 3:</u>

Implement Iterative deepening search algorithm.
Algorithm:

Using IDDFS

Algorithmn

Step 1: Check if the graph is a binary trees

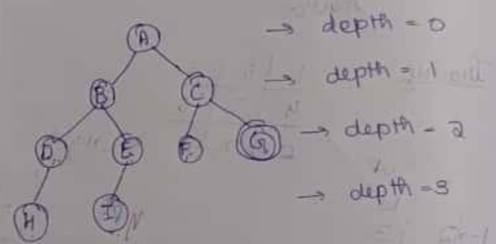
Step 2: Check from depth 0 [i.e the root node]

Step 3: do Check from depth 1 search all nodes reachable in depth 1

Step 4: Increase the depth to 2 search all nodes reachable in depth 2

Step 5: Continue the following until you fill the goal node or reach max depth.

Step 6: In each depth check the nodes from left



Soln: A

A B C

A B D E C F G

Goal I: ~~A B D H E I G~~ ~~A B~~

A → B → D → E → I → C → F → G

A → B → D → E → C → F → G → H → I

Output

Moves: ['left', 'down', 'right', 'right']

No. of moves: 4

Output Tracing

1 2 3
5 _ 6
4 7 8

1. Move left
Swap (_) left with 5

1 2 3
_ 5 6
4 7 8

2. Move down
Swap (_) down with 4

1 2 3
4 5 6
_ 7 8

3. Move right
Swap (_) with 7

1 2 3
4 5 6
7 _ 8

4. Moves Right: swap blank right

1 2 3
4 5 6
7 8 _

Code:

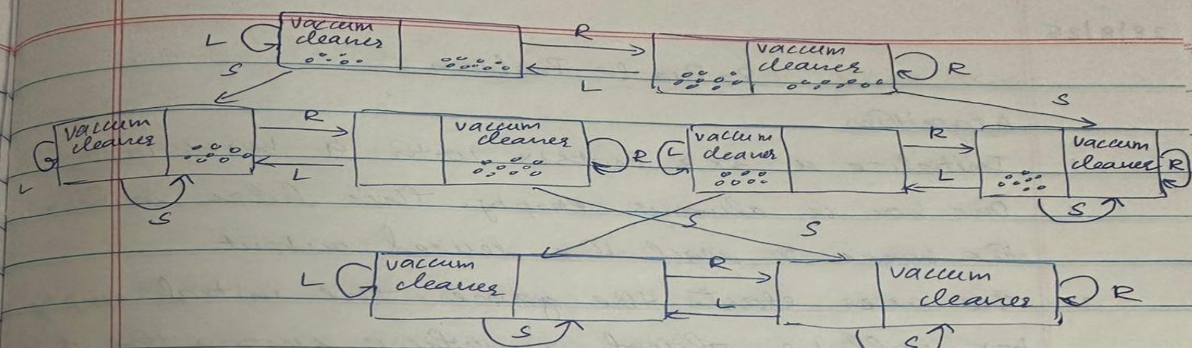
```
import copy def
get_puzzle(name):
   print(f"\nEnter the {name} puzzle (3x3, use -1 for blank):")
puzzle = []     for i in range(3):
       row = list(map(int, input(f"Row {i+1} (space-separated 3 numbers): ").split()))
puzzle.append(row)     return puzzle def move(temp, movement):     for i in
range(3):         for j in range(3):             if temp[i][j] == -1:
               if movement == "up" and i > 0:
                   temp[i][j], temp[i-1][j] = temp[i-1][j], temp[i][j]
elif movement == "down" and i < 2:
                   temp[i][j], temp[i+1][j] = temp[i+1][j], temp[i][j]
elif movement == "left" and j > 0:
                   temp[i][j], temp[i][j-1] = temp[i][j-1], temp[i][j]
elif movement == "right" and j < 2:
                   temp[i][j], temp[i][j+1] = temp[i][j+1], temp[i][j]
return temp     return temp def dls(puzzle, depth, limit,
last_move, goal):     if puzzle == goal:
       return True, [puzzle], []     if depth >= limit:         return False, [], []     for move_dir,
opposite in [("up","down"), ("left","right"), ("down","up"), ("right","left")]:
       if last_move == opposite:  # avoid direct backtracking
continue
       temp = copy.deepcopy(puzzle)         new_state = move(temp,
move_dir)         if new_state != puzzle:  # valid move             found, path,
moves = dls(new_state, depth+1, limit, move_dir, goal)             if found:
               return True, [puzzle] + path, [move_dir] + moves
return False, [], [] def ids(start, goal):
   for limit in range(1, 50):  # reasonable max depth
print(f"\nTrying depth limit = {limit}")         found,
path, moves = dls(start, 0, limit, None, goal)         if
found:
           print("Solution found!")
for step in path:             print(step)
print("Moves:", moves)
print("Path cost =", len(path)-1)
return
   print(" Solution not found within depth limit.")
start_puzzle = get_puzzle("start") goal_puzzle =
get_puzzle("goal") print("\n~~~~~~~~~~~~
IDDFS ~~~~~~~~~~~~") ids(start_puzzle,
goal_puzzle)
```

ScreenShot:

```
Output

Enter the start puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 7 8
Row 3 (space-separated 3 numbers): 5 6 -1

Enter the goal puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 5 6
Row 3 (space-separated 3 numbers): 7 8 -1

~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~

Trying depth limit = 1

Trying depth limit = 2

Trying depth limit = 3

Trying depth limit = 4

Trying depth limit = 5

Trying depth limit = 6

Trying depth limit = 7

Trying depth limit = 8

Trying depth limit = 9

Trying depth limit = 10
```

Program 4:

Implement a vacuum cleaner agent.
Algorithm:



Output :

Initial room statuses :

Room 1 : Dirty

Room 2 : Clean

Room 3 : Dirty

Room 4 : Dirty

Cleaning room 1 ....

Room 1 is clean

Room 2 is already clean

Cleaning room 3

Room 3 is clean

Cleaning room 4 ..

Room 4 is clean

All room clean!

Code:

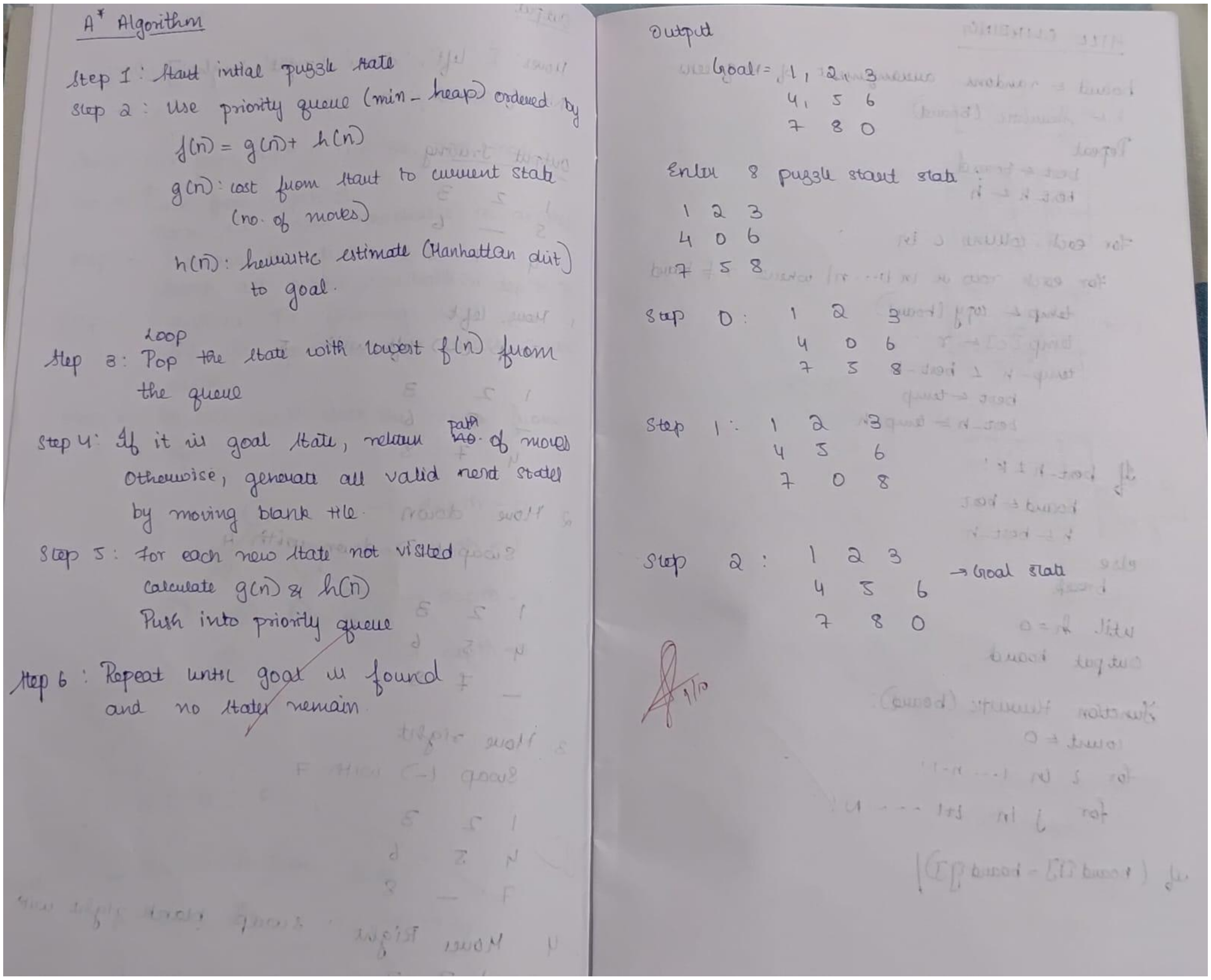
```
def show_rooms_status(rooms):     for
room_number, status in rooms.items():
        print(f"Room {room_number}: {'Clean' if status else 'Dirty'}")
def clean_room(rooms, room_number):     if rooms[room_number]:
        print(f"Room {room_number} is already clean.")
else:
        print(f"Cleaning room {room_number}...")
rooms[room_number] = True         print(f"Room
{room_number} is now clean!") def
clean_all_rooms(rooms):     print("Initial room
statuses:")     show_rooms_status(rooms)
    print("\nStarting cleaning process...\n")
for room_number in rooms:
        clean_room(rooms, room_number)
        print()
    print("Final room statuses:")
show_rooms_status(rooms) if
__name__ == "__main__":
    rooms = {
1: False,
        2: True,
        3: False,
        4: False
    }
    clean_all_rooms(rooms)
```

ScreenShot:

```
Output
Initial room statuses:
Room 1: Dirty
Room 2: Clean
Room 3: Dirty
Room 4: Dirty

Starting cleaning process...

Cleaning room 1...
Room 1 is now clean!

Room 2 is already clean.

Cleaning room 3...
Room 3 is now clean!

Cleaning room 4...
Room 4 is now clean!

Final room statuses:
Room 1: Clean
Room 2: Clean
Room 3: Clean
Room 4: Clean

=== Code Execution Successful ===
```

Program 5:

Implement A* search algorithm.
Algorithm:



A* Algorithm

Step 1: Start initial puzzle state

Step 2: Use priority queue (min-heap) ordered by
$f(n) = g(n) + h(n)$
$g(n)$: cost from start to current state (no. of moves)
$h(n)$: heuristic estimate (Manhattan dist) to goal.

Loop
Step 3: Pop the state with lowest $f(n)$ from the queue

Step 4: If it is goal state, return path no. of moves
Otherwise, generate all valid next states by moving blank tile.

Step 5: For each new state not visited
Calculate $g(n)$ & $h(n)$
Push into priority queue

Step 6: Repeat until goal is found and no states remain

Output

Goal = 1, 2, 3
4, 5, 6
7 8 0

Enter 8 puzzle start state
1 2 3
4 0 6
7 5 8

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

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

Step 2: 1 2 3
4 5 6
7 8 0 → Goal state

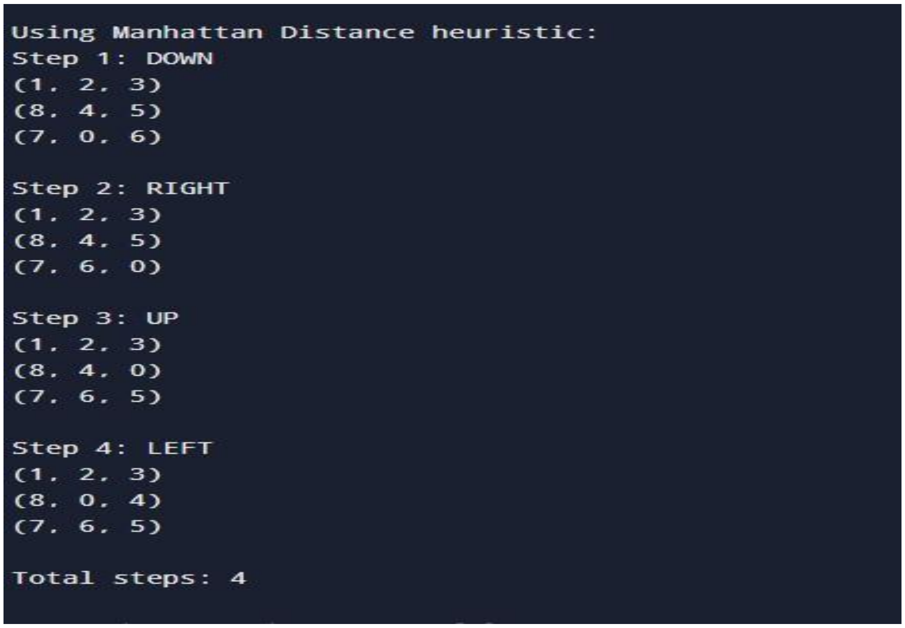
Code:

```
from heapq import heappush, heappop
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
direction_names = ["UP", "DOWN", "LEFT", "RIGHT"] def
misplaced_tiles(state):
```

count = 0 for i
in range(3): for j
in range(3): if
state[i][j] != 0 and
state[i][j] !=
goal_state[i][j]:
count += 1 return
count def
manhattan_distance(state):
distance = 0 for i
in range(3): for j
in range(3):
tile = state[i][j]
if tile != 0:
goal_x, goal_y = divmod(tile - 1, 3)
distance += abs(i - goal_x) + abs(j - goal_y) return
distance def get_neighbors_with_actions(state):
neighbors = [] for i
in range(3): for j in
range(3): if
state[i][j] == 0:
x, y = i, j break for (dx, dy), action
in zip(directions, direction_names):
nx, ny = x + dx, y + dy if
0 <= nx < 3 and 0 <= ny < 3:
new_state = [list(row) for row in state] new_state[x][y],
new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
neighbors.append((new_state, action)) return neighbors def
state_to_tuple(state):
return tuple(tuple(row) for row in state) def
reconstruct_path(came_from, current):
actions = [] states = [] while current in came_from:
prev_state, action = came_from[current]
actions.append(action) states.append(current)
current = prev_state states.append(current)
actions.reverse() states.reverse() return actions, states
def a_star_search_with_steps(initial_state, heuristic_func):
open_list = []
closed_set = set()
g_score = {state_to_tuple(initial_state): 0} f_score =
{state_to_tuple(initial_state): heuristic_func(initial_state)}
came_from = {}
heappush(open_list, (f_score[state_to_tuple(initial_state)], initial_state))
while open_list:

```
    _, current_state = heappop(open_list)
current_t = state_to_tuple(current_state)        if
current_state == goal_state:
        return reconstruct_path(came_from, current_t)
closed_set.add(current_t)        for neighbor, action in
get_neighbors_with_actions(current_state):
        neighbor_t = state_to_tuple(neighbor)
if neighbor_t in closed_set:
            continue            tentative_g = g_score[current_t] + 1            if
neighbor_t not in g_score or tentative_g < g_score[neighbor_t]:
            came_from[neighbor_t] = (current_t, action)
g_score[neighbor_t] = tentative_g                f_score[neighbor_t] =
tentative_g + heuristic_func(neighbor)                heappush(open_list,
(f_score[neighbor_t], neighbor))     return None, None def
print_path(actions, states):     for i, (action, state) in
enumerate(zip(actions, states[1:]), 1):
    print(f"Step {i}: {action}")
for row in state:
print(row)        print()
initial_state = [     [1, 2, 3],
  [8, 0, 5],
  [7, 4, 6]
]
print("Using Misplaced Tiles heuristic:") actions, states =
a_star_search_with_steps(initial_state, misplaced_tiles) if actions:
  print_path(actions, states)
print("Total steps:", len(actions)) else:
  print("No solution found.") print("\nUsing Manhattan Distance
heuristic:") actions, states = a_star_search_with_steps(initial_state,
manhattan_distance) if actions:
  print_path(actions, states)
print("Total steps:", len(actions))
else:     print("No solution found.")
```

ScreenShot:



```
Using Manhattan Distance heuristic:
Step 1: DOWN
(1, 2, 3)
(8, 4, 5)
(7, 0, 6)

Step 2: RIGHT
(1, 2, 3)
(8, 4, 5)
(7, 6, 0)

Step 3: UP
(1, 2, 3)
(8, 4, 0)
(7, 6, 5)

Step 4: LEFT
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Total steps: 4
```

b. Implement Hill Climbing Algorithm Algorithm:

HILL CLIMBING

board ← random arrangement of N-queens
h ← heuristic (board)
Repeat
best ← board
best_h ← h

for each column c in :
for each row r in [1... n] where r ≠ board
temp ← copy (board)
temp [c] ← r
temp-h < best-h
best ← temp
best_h ← temp-h

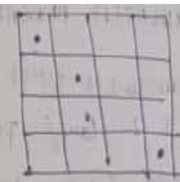
if best-h < k:
board ← best
h ← best_h
else
break

util h=0
Output board

Function Heuristic (board):
count ← 0
for i in 1....n-1:
for j in i+1 ----N
if (board[i] = board[j])

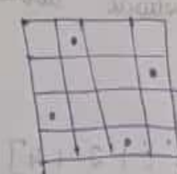
Tracing
Initial state: [1 2 2 4]

2 2 1 4 = 3
3 2 1 4 = 3
→ 1 2 1 3 = 2
4 2 1 4 = 2
1 1 1 4 = 3
1 3 1 4 = 2
1 4 1 4 = 2
1 2 3 4 = 3
1 2 4 4 = 3
1 2 1 1 = 3
1 2 1 2 = 3
2 2 1 3 = 2
3 2 1 3 = 3
→ 4 2 1 3 = 1
2 2 1 4 = 2
4 3 1 4 = 2
4 2 4 4 = 2
3 1 1 3 = 3
→ 2 4 1 3 = 0


h=3


h=2


h=1


h=0
goal

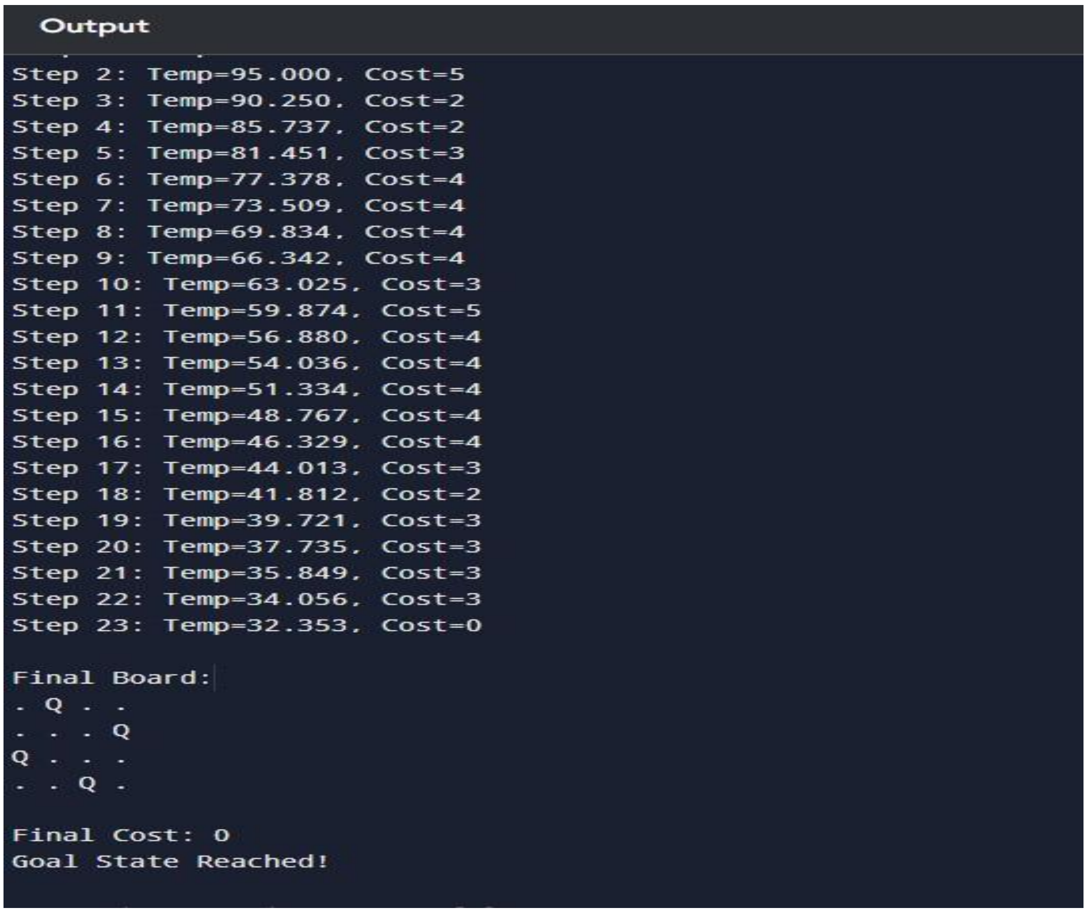
Output:
Initial state: 1 2 2 4
Goal state: 2 4 1 3

Code:

```
import random import time def
generate_initial_state(n=4):
   return [random.randint(0, n - 1) for _ in range(n)] def
calculate_conflicts(state):
   conflicts = 0     n =
len(state)     for i in
range(n):         for j in
range(i + 1, n):             if
state[i] == state[j]:
             conflicts += 1             if
abs(state[i] - state[j]) == abs(i - j):
             conflicts += 1
return conflicts def
get_neighbors(state):
   neighbors = []     n = len(state)
for col in range(n):         for row in
range(n):             if state[col] != row:
neighbor = state.copy()
neighbor[col] = row
neighbors.append(neighbor)     return
neighbors def print_board(state):
   n = len(state)     for row
in range(n):         line = ""
for col in range(n):
if state[col] == row:
             line += "Q "             else:                 line += ". "         print(line)     print() def
hill_climbing_with_steps(n=4, max_restarts=100):     for restart in range(max_restarts):
current = generate_initial_state(n)         step = 0         print(f"Restart #{restart+1}: Initial
state (Conflicts = {calculate_conflicts(current)})")         print_board(current)         while
True:
          current_conflicts = calculate_conflicts(current)
if current_conflicts == 0:
             print(f"Solution found in {step} steps!")
return current
          neighbors = get_neighbors(current)             neighbor_conflicts =
[calculate_conflicts(nbr) for nbr in neighbors]             min_conflict =
min(neighbor_conflicts)             if min_conflict >= current_conflicts:
             print("Reached local minimum, restarting...\n")
break
          best_neighbor = neighbors[neighbor_conflicts.index(min_conflict)]
step += 1             print(f"Step {step}: Conflicts = {min_conflict}")
print_board(best_neighbor)
```

current = best_neighbor
return None solution =
hill_climbing_with_steps() if
solution:
print("Final Solution:")
print_board(solution) else:
print("No solution found.")

ScreenShot:



```
Output
Step 2: Temp=95.000, Cost=5
Step 3: Temp=90.250, Cost=2
Step 4: Temp=85.737, Cost=2
Step 5: Temp=81.451, Cost=3
Step 6: Temp=77.378, Cost=4
Step 7: Temp=73.509, Cost=4
Step 8: Temp=69.834, Cost=4
Step 9: Temp=66.342, Cost=4
Step 10: Temp=63.025, Cost=3
Step 11: Temp=59.874, Cost=5
Step 12: Temp=56.880, Cost=4
Step 13: Temp=54.036, Cost=4
Step 14: Temp=51.334, Cost=4
Step 15: Temp=48.767, Cost=4
Step 16: Temp=46.329, Cost=4
Step 17: Temp=44.013, Cost=3
Step 18: Temp=41.812, Cost=2
Step 19: Temp=39.721, Cost=3
Step 20: Temp=37.735, Cost=3
Step 21: Temp=35.849, Cost=3
Step 22: Temp=34.056, Cost=3
Step 23: Temp=32.353, Cost=0

Final Board:
. Q . .
. . . Q
Q . . .
. . Q .

Final Cost: 0
Goal State Reached!
```

Program 6:

Write a program to implement Simulated Annealing Algorithm Code:

```
import random import
math def
print_board(board):
   n = len(board)
for i in range(n):
row = ["Q" if
board[i] == j else
"." for j in range(n)]
print(" ".join(row))
print() def
calculate_cost(boar
d):
   """Heuristic: number of pairs of queens attacking each other"""
n = len(board)     cost = 0     for i in range(n):         for j in range(i +
1, n):             if board[i] == board[j] or abs(board[i] - board[j]) ==
abs(i - j):
             cost += 1     return
cost def
random_neighbor(board):
   """Generate a random neighboring board by moving one queen"""     n =
len(board)     neighbor = list(board)     row = random.randint(0, n - 1)     col =
random.randint(0, n - 1)     neighbor[row] = col     return neighbor def
simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
   current_board = [random.randint(0, n - 1) for _ in range(n)]
current_cost = calculate_cost(current_board)     temperature =
initial_temp     step = 1     print("Initial Board:")
print_board(current_board)     print(f"Initial Cost:
{current_cost}\n")     while temperature > stopping_temp and
current_cost > 0:
      neighbor = random_neighbor(current_board)        neighbor_cost
= calculate_cost(neighbor)        delta = neighbor_cost - current_cost
if delta < 0 or random.random() < math.exp(-delta / temperature):
         current_board = neighbor            current_cost = neighbor_cost
print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1        temperature *= cooling_rate     print("\nFinal Board:")
print_board(current_board)     print(f"Final Cost: {current_cost}")     if
current_cost == 0:
      print("Goal State Reached!")
else:
      print("Terminated before reaching goal.")
simulated_annealing(4)
```

ScreenShot:

```
Output
Restart #1: Initial state (Conflicts = 2)
. Q Q .
. . . Q
. . . .
Q . . .

Step 1: Conflicts = 1
. Q . .
. . . Q
. . Q .
Q . . .

Reached local minimum, restarting...

Restart #2: Initial state (Conflicts = 2)
. . Q .
Q Q . .
. . . Q
. . . .

Step 1: Conflicts = 0
. . Q .
Q . . .
. . . Q
. Q . .

Solution found in 1 steps!
Final Solution:
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 7:

Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

Algorithm:

LAB-06

1. Create knowledge base using propotional logic & show that given query entails knowledge base or not.

2. ~~Q → P~~ ~~Q → P~~
P → ¬Q
~~Q → R~~
Q ∨ R

① Q → P
② P → ¬Q
③ Q ∨ R

KB - (P → Q) ∧ (Q → R) ∧ P

① Truth Table

T → true F → False

P	Q	R	Q → P	P → ¬Q	Q ∨ R	KB
F	F	F	T	T	F	F
F	F	T	T	T	T	T
F	T	F	F	T	T	F
F	T	T	F	T	T	F
T	F	F	T	T	F	F
T	F	T	T	T	T	T
T	T	F	T	F	T	F
T	T	T	T	F	T	F

i) Does KB entail R?
KB entails R as R is true when the KB conditions are true.
∴ KB entails R.

ii) KB entails R → P ?

R	P	R → P	KB
T	F	F	T
T	T	T	T

Since R → P is false
∴ KB does not entails R → P

iii) Does KB entails Q → R

Q	R	Q → R	KB
F	T	T	T
F	T	T	T

Since Q → R is true in the KB.
∴ KB entails Q → R

Algorithm

Input:
KB → A (set of PL sentences)
α → A (query)

Output:
True → if KB ⊨ α
false → otherwise

~~def tentails (KB, query)~~

function check(KB, α, symbols, model)
return true or false

if empty? (symbols) then
if PL-True ? (KB, model) then
return PL-True ? (α, model)
else return true.
else do
P ← first (symbol)
rest ← Rest (symbols)
return (TT-check (KB, α, rest, model ∪ {P = true})
and
T-check (KB, α, rest models ∪ {P = false})

Function Entails (KB, α) returns true or false
inputs KB, knowledge base, sentence in PL, α, query
symbols ← a list of proposition symbols in KB and α
return TT entails (KB, α, symbols, {})

16/10

__Output:__

Enter no of KB sentences: 3
Enter sentence 1: Q → P
Enter sentence 2: P → !Q
Enter sentence 3: Q V R
Enter queries: R, R → P, Q → R
Entailment results:
Does KB entail R → ? True
Does KB entail Q → R? ~~False~~ True
Does KB entail R → P? False

(2) Enter no of KB sentences: 1
Enter sentence: (A V C) ∧ (B V !C)
Enter query: A V B
Entailment results:
Does KB entail A V B? → true

Code:

```
import itertools
import pandas as pd variables = ['P', 'Q', 'R'] combinations =
list(itertools.product([False, True], repeat=3)) rows = [] for
(P, Q, R) in combinations:
   s1 = (not Q) or P            s2 = (not
P) or (not Q)   # P → ¬Q     s3 = Q
or R              # Q ∨ R     KB = s1
and s2 and s3     entail_R = R
   entail_R_imp_P = (not R) or P
entail_Q_imp_R = (not Q) or R     rows.append({
      'P': P, 'Q': Q, 'R': R,
      'Q → P': s1,
      'P → ¬Q': s2,
      'Q ∨ R': s3,
      'KB True?': KB,
      'R': entail_R,
      'R → P': entail_R_imp_P,
      'Q → R': entail_Q_imp_R
   })
df = pd.DataFrame(rows) print("Truth
Table for Knowledge Base:\n")
print(df.to_string(index=False))
models_true = df[df['KB True?'] == True]
print("\nModels where KB is True:\n")
print(models_true[['P', 'Q', 'R']]) def
entails(column):
   """Check if KB entails the given statement."""
return all(models_true[column]) print("\nEntailment
Results:")
print(f"KB ⊨ R ? {'Yes' if entails('R') else 'No'}")
print(f"KB ⊨ R → P ? {'Yes' if entails('R → P') else 'No'}")
print(f"KB ⊨ Q → R ? {'Yes' if entails('Q → R') else
'No'}")
```

ScreenShot:

```
Output
Truth Table for Knowledge Base:

    P      Q      R  Q → P  P → ¬Q  Q ∨ R  KB True?  R → P  Q → R
False False False   True    True  False     False   True   True
False False  True   True    True   True      True  False   True
False  True False  False    True   True     False   True  False
False  True  True  False    True   True     False  False   True
 True False False   True    True  False     False   True   True
 True False  True   True    True   True      True   True   True
 True  True False   True   False   True     False   True  False
 True  True  True   True   False   True     False   True   True

Models where KB is True:

       P      Q     R
1  False  False  True
5   True  False  True

Entailment Results:
KB ⊨ R ? Yes
KB ⊨ R → P ? No
KB ⊨ Q → R ? Yes

=== Code Execution Successful ===
```

Program 8:

Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:

LAB

Resolution in FOL

step 1: convert all sentences to CNF

step 2: Negate conclusion S & convert result to CNF

step 3: Add negate conclusion S to premise clauses.

step 4: Repeat until contradiction or no progress made:

(a) Select 2 clauses

(b) Resolve them together, performing all required unification

(c) If resolvent is empty clause, contradiction has been found

(d) If not, add resolvent to the premise

step 5: If step 4 is a success, we have proved the conclusion

Example

Premise Given:

Mother (Lulu, Fifi)

Alive (Lulu)

$\forall x \ \forall y$ Mother $(x, y) \rightarrow$ Parent (x, y)

$\forall x \ \forall y$ (Parent $(x, y) \wedge$ Alive $(x)) \rightarrow$ Older (x, y)

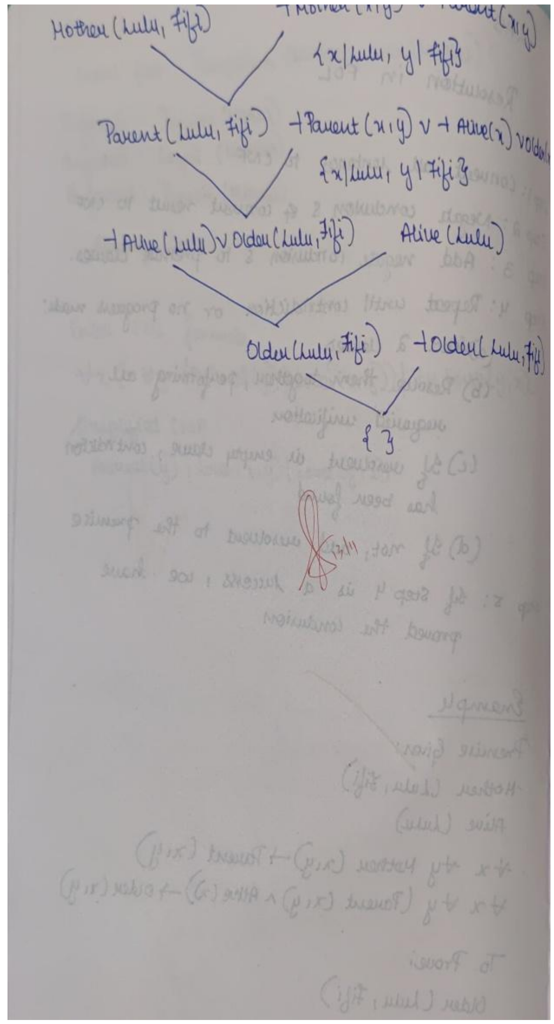
To Prove:

Older (Lulu, Fifi)

Solution:

$\neg$Mother $(x, y) \vee$ Parent (x, y)

$\neg$Parent $(x, y) \vee \neg$ Alive $(x) \vee$ Older (x, y)



Program 9:

Implement unification in first order logic.

Algorithm:

Lab 07

Implement Unification in FOL

Algorithm Unify (x, y)

Input: Two expressions x & y.

Output: A substitution set SUBST failure

1. If x & y are identical, then return
2. Else if x is a variable, then if x occurs y then return failure.
   else return {x|y}
3. Else if y is variable then if y occurs in x then return failure
   Else return {y|x}
4. Else if both x and y are both comp expression then
   If predicate/function symbols of x and y are different then
   Return failure
   Else
   S1 ← unify (argument (x), argument(y))
   if S1 = failure then return failure
   else return S1
5. Else return failure

1. $P(f(x), g(y), y)$

$P(f(g(x)), g(f(a)), f(a))$

Replace x with $g(x)$

Replace y with $f(a)$

$P(f(g(x)), g(f(a)), f(a))$

unifies = $\{y \mid f(a), x \mid g(x)\}$

2. $Q(x, f(x))$

$Q(f(y), y)$

~~Replace x with $f(y)$~~

~~Replace y win $f(x)$~~

~~$Q(f(y), f(x))$~~

~~$Q(f(y), f(x))$~~

unifier = $\{x \mid f(y), y \mid f(x)\}$

$x \mid f(y)$

$f(f(x))$

Not unifiable cause of recurring relation

3. $P(x, g(x))$

$P(g(y), g(g(x)))$

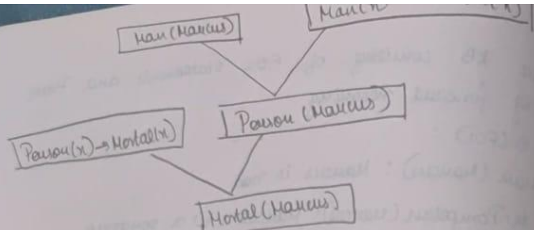
Replace x with $g(y)$

Replace y with $g(x)$

unifies $\{x \mid g(y), y \mid g(x)\}$

Program 10:

Convert a given first order logic statement into Conjunctive Normal Form (CNF).
Algorithm:



Algorithm

func FOL ask (KB, α) return substitution or false
  input: KB
  α

  repeat until new is empty
    new ← { }
    for each rule in KB do
    ($p_1 \wedge \dots p_n \Rightarrow q$) standardize variable
    for each θ such that SUBT (θ, $p_1 \wedge \dots p_n$) = SUBT (θ, $p_1' \wedge \dots p_n'$)
      for some $p_1' \dots p_n'$ in KB
    $q' \leftarrow$ SUBT (θ, q)
    if q' does not unify with statement in KB
      add q' to new
      $\phi \leftarrow$ Unify (q', α)
      if ϕ is not fail then return ϕ
    add new to KB
    return false.

CNF

10. Convert FOL into CNF

$\forall x \{\neg \forall y (Animal(y) \vee Loves(x,y))\} \vee [\exists y \, Loves(y,x)]$

$\forall x \, [\exists y \, (Animal(y) \vee \neg Loves(x,y))] \vee \{\exists y \, Loves(y,x)\}$

$\forall x \{\exists y_1 \, Animal(y_1) \wedge \neg Loves(x, y_1)\} \vee [\exists y_2 \, Loves(y_2, x)]$

$y_1 = f(x) \quad y_2 = g(x)$

$\forall x \; Animal(f(x)) \wedge \neg Loves(x, f(x)) \vee Loves(g(x), x)$

$Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(f_2(x), x)$

Algorithm

1. Elimate biconditional and implications
   $\alpha \Rightarrow \beta \qquad \neg\alpha \vee \beta$
   $\alpha \Leftrightarrow \beta \qquad (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. Move $\neg$ inwards
   $\neg(\forall x \, p) = \exists x \neg p$
   $\neg(\exists x \, p) = \forall x \neg p$
   $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   $\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$
   $\neg\neg\alpha = \alpha$
3. Standardize variables by renaming
   Each quantifier should use different variable
3. Drop universal quantifier
6. Distribute $\wedge$ over $\vee$

<u>Program 11:</u>

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

Algorithm:

Create KB consisting of FOL statements and Prove using forward reasoning

KB (FOL) :

1. Man (Marcus) : Marcus is man

2. ~~#~~ Pompeian (Marcus): Marcus is a pompeian.

3. $\forall x$ (Pompeian(x) $\rightarrow$ Roman(x))

   All pompeians are Romans

4. $\forall x$ (Roman(x) $\rightarrow$ Loyal(x))

   All Romans are Loyal.

5. $\forall x$ (Man(x) $\rightarrow$ Person(x))

   All men are persons

6. $\forall x$ (Person(x) $\rightarrow$ Mortal(x))

Query (α): Mortal (Marcus)

Soln: Man (Marcus) Pompeian (Marcus)

(3) : $\forall x$ (Pompeian(x) $\rightarrow$ Roman(x))

subst x = Marcus

Pompeian (Marcus) $\rightarrow$ Roman (Marcus)

(4) : $\forall x$ (Roman(x) $\rightarrow$ Loyal(x))

subst x : Marcus

Roman (Marcus) $\rightarrow$ Loyal (Marcus)

(5) : Man (x) $\rightarrow$ Person(x)

x = Marcus

Man (Marcus) $\rightarrow$ Person (Marcus)

(6) : $\forall x$ Person (~~Marcus~~ x) $\rightarrow$ Mortal (x)

subst x/Marcus

Person (Marcus) $\rightarrow$ Mortal (Marcus)

$\therefore$ It is true Marcus is mortal

Program 12:

Implement Alpha-Beta Pruning.

Algorithm:

Alpha - Beta Search.

```
func search (state) return action.
  v ← Max-val (state, -∞, +∞)
  return action in ACTIONS(state) with value v

func MaxVal (state, α, β) return utility value
  if terminal-test (state) then return utility (state)
    v ← ∞
  for each a in action(state) do
    v ← Max (v, Minval (Result (s, a), α, β))
    if v ≥ β then return v
    α ← Max(α, v)
  return v

function MinVal (state, α, β) return utility value
  if terminal-test (state) then return utility (state)
    v ← +∞
  for each α in Actions(state) do
    v ← MIN (v, Max-value (Result (s, a), α, β))
    if v ≤ α then return v
    β ← Min(β, v)
  return v
```

Code:

```
import math
PLAYER = "X"  # Human
AI = "O"      # Computer def
print_board(board):     for
row in board:         print(" |
".join(row))         print("-" *
9) def
available_moves(board):
   """Return list of available (row, col) moves."""
moves = []
   for i in range(3):         for j
in range(3):             if
board[i][j] == " ":
moves.append((i, j))     return
moves def
check_winner(board):
   """Return 'X' if X wins, 'O' if O wins, or None otherwise."""
for i in range(3):         if board[i][0] == board[i][1] ==
board[i][2] != " ":
         return board[i][0]         if board[0][i] ==
board[1][i] == board[2][i] != " ":
         return board[0][i]     if board[0][0] ==
board[1][1] == board[2][2] != " ":
      return board[0][0]     if board[0][2] ==
board[1][1] == board[2][0] != " ":
      return board[0][2]
return None def
is_full(board):
   return all(cell != " " for row in board for cell in row) def
minimax(board, depth, is_maximizing):
   winner = check_winner(board)
if winner == AI:
      return 1     elif winner
== PLAYER:
      return -1     elif
is_full(board):
      return 0     if
is_maximizing:
      best_score = -math.inf         for (i, j)
in available_moves(board):
         board[i][j] = AI             score =
minimax(board, depth + 1, False)
board[i][j] = " "             best_score = max(score,
best_score)         return best_score     else:
```

best_score = math.inf for (i, j)
in available_moves(board):
board[i][j] = PLAYER score =
minimax(board, depth + 1, True)
board[i][j] = " " best_score = min(score,
best_score) return best_score def
best_move(board):
"""Find the best move for the AI."""
best_score = -math.inf move =
None
for (i, j) in available_moves(board):
board[i][j] = AI score =
minimax(board, 0, False)
board[i][j] = " "
if score > best_score:
best_score = score
move = (i, j)
return move def
play_game():
board = [[" " for _ in range(3)] for _ in range(3)]
print("Tic Tac Toe - You are X, AI is O")
print_board(board) while True:
row = int(input("Enter row (0-2): "))
col = int(input("Enter col (0-2): ")) if
board[row][col] != " ":
print("Cell taken, try again.")
continue
board[row][col] = PLAYER if
check_winner(board) == PLAYER:
print_board(board)
print("You win!") break
elif is_full(board):
print_board(board)
print("It's a draw!") break
print("AI is making a move...")
move = best_move(board) if
move:
board[move[0]][move[1]] = AI
print_board(board) if
check_winner(board) == AI:
print("AI wins!")
break elif
is_full(board):
print("It's a draw!")

break if __name__
== "__main__":
play_game()

ScreenShot: