VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Suhas B P(1BM23CS345)

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
Aug 2025 to Dec 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 **Suhas B P (1BM23CS345),** 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. 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/1BM23CS345/AI

INDEX

Name Suhas B.P Std Sec

Roll No. Subject AI School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	18/08/25	Implementation Tic - Tac-Toe		10
2	25/08/25	Vaccume.		10
3	1/09/25	8 puzzle using BFS & DFS.		15/9/2
4		Iterative deepning.		
5	9/8/25	A* Algorithm.		
		Hill climbing.		
6	9/10/25	Stimulated Annealing		
7	6/10/25	Propositional Logic		8
8	13/10/25	Unification		13/10/25
		First order logic.		
9	20/10/25	First order Logic		8
10	27/10/25	Minmax & Alpha beta.		30/10/25

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Code:
Implement Tic –Tac –Toe Game

```
def print_board(board):
    for row in board:
        print("|".join(row))
    print()

def check_winner(board, player):
    for row in board:
        if all(s == player for s in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        row, col = map(int, input(f"Player {current_player}, enter row and col (0-2, space separated):
").split())
        if board[row][col] == " ":
            board[row][col] = current_player
            if check_winner(board, current_player):
                print_board(board)
                print(f"Player {current_player} wins!")
```

```
            break
        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = "O" if current_player == "X" else "X"
    else:
        print("Cell already taken, try again.")
```

Vacuume is in room D
D is dirty
cleaning D.....
Moving vacum to A

Vacum is in room A
A is dirty.
cleaning A....

Cost = 6
{'A': 0, 'B': 0, 'C': 0, 'D': 0}

lay_game()

```
Player X, enter row and col (0-2, space separated): 2 2
X|O|
 |X|
 |O|X

Player X wins!
```

Suhas B P (1BM23CS345)

```
Player X, enter row and col (0-2, space separated): 2 1
X|O|X
X|O|O
O|X|X

It's a draw!
```

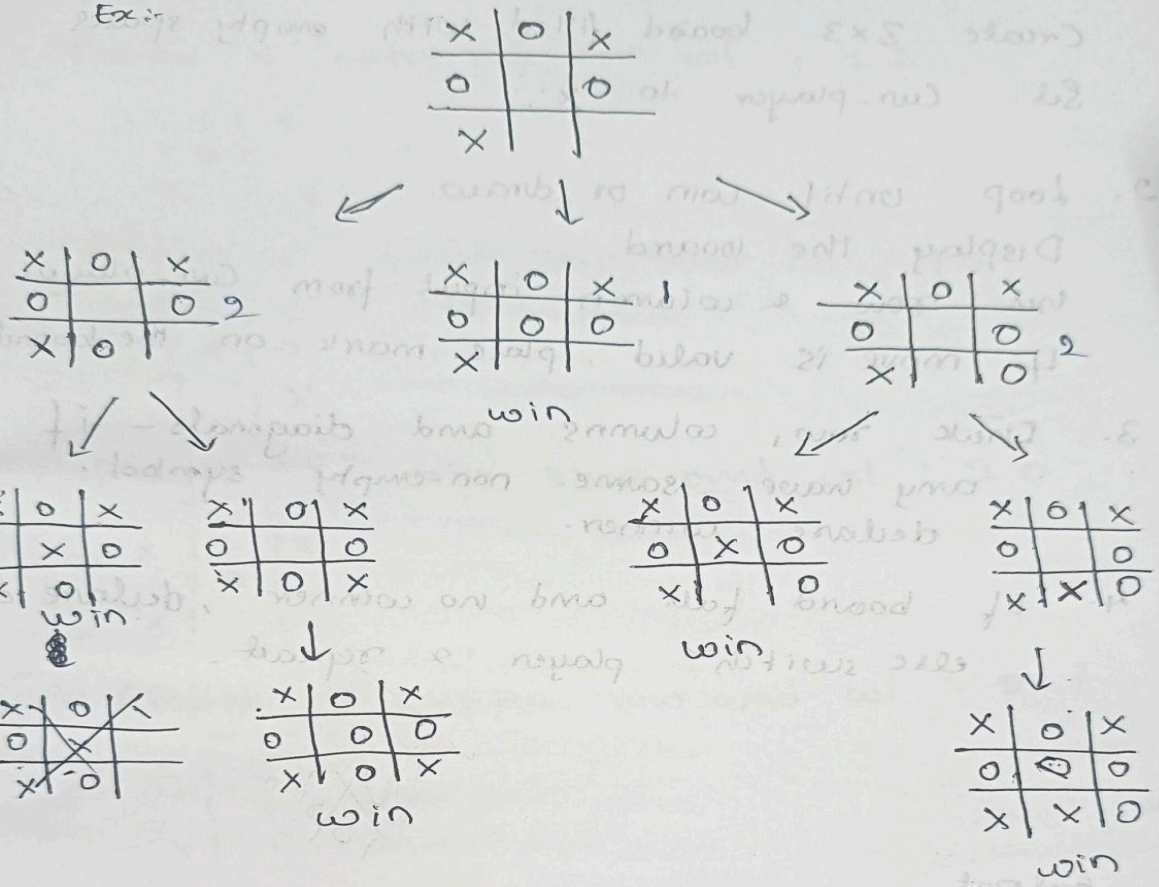
Suhas B P (1BM23CS345)

```
Player X, enter row and col (0-2, space separated): 2 1
X|O|X
X|O|O
O|X|X

It's a draw!
```

Suhas B P (1BM23CS345)

1) Implementation of Tic – Tac – Toe.

Ex :-



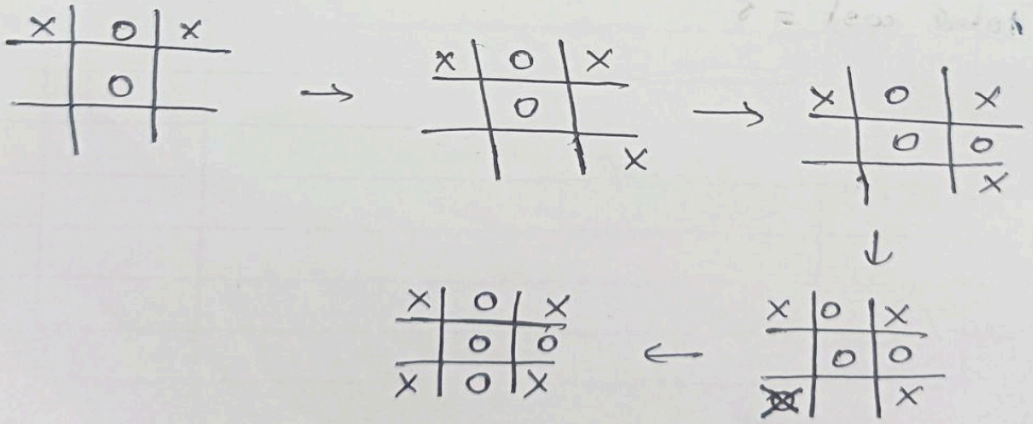
Total cost = 5

Code :

1. Initialize
   Create 3×3 board filled with empty spaces
   Set Cur-player to 'X'.
2. Loop until win or draw.
   Display the board
   Get row & column input from Cur-player
   If move is valid, place mark on the board.
3. Check row, columns and diagonals - if any have same non-empty symbol.
   declare winner.
4. If board full and no winner, declare draw
   else switch player & repeat.

10/3/25

Output



O wins.

Out put

```
X | O | X
  | O |
  |   |
```

Player 'X' enter row and col : 2 2.

```
X | O | X
  | O |
  |   | X
```

Player 'O' enter row and col : 1 2

```
X | O | X
  | O | O
  |   | X
```

Player 'X' enter row and col : 2 0

```
X | O | X
  | O | O
X |   | X
```

Player 'O' enter row and col : 2 -1

```
X | O | X
  | O | O
X | O | X
```

Player 'O' wins.

~~count = 1~~

Implement vacuum cleaner agent

```
rooms = {
    'A': int(input("Enter state of A (0 for clean, 1 for dirty): ")),
    'B': int(input("Enter state of B (0 for clean, 1 for dirty): ")),
    'C': int(input("Enter state of C (0 for clean, 1 for dirty): ")),
    'D': int(input("Enter state of D (0 for clean, 1 for dirty): "))
}

start = input("Enter starting location (A, B, C, or D): ").upper()
order = ['A', 'B', 'C', 'D']

if start not in order:
    print("Invalid starting location.")
    exit()

start_index = order.index(start)
visited_order = order[start_index:] + order[:start_index]

cost = 0

for i in range(len(visited_order)):
    current = visited_order[i]
    print(f"\nVacuum is in room {current}")

    if rooms[current] == 1:
        print(f"{current} is dirty.")
        print(f"Cleaning {current}...")
        rooms[current] = 0
        cost += 1
    else:
        print(f"{current} is clean.")

    if i < len(visited_order) - 1:
        next_room = visited_order[i + 1]
        print(f"Moving vacuum to {next_room}")
        cost += 1

print(f"\nCost: {cost}")
print(rooms)
```

Start coding or generate with AI.

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter state of C (0 for clean, 1 for dirty): 1
Enter state of D (0 for clean, 1 for dirty): 1
Enter starting location (A, B, C, or D): B

Vacuum is in room B
B is clean.
Moving vacuum to C

Vacuum is in room C
C is dirty.
Cleaning C...
Moving vacuum to D

Vacuum is in room D
D is dirty.
Cleaning D...
Moving vacuum to A

Vacuum is in room A
A is dirty.
Cleaning A...

Cost: 6
{'A': 0, 'B': 0, 'C': 0, 'D': 0}
```

Suhas B P (1BM23CS345)

Lab -2

1) Vaccume cleaner
For two Rooms

Step 1 : Start.
Step 2 :- Let two rooms are 'A' and 'B'
Step 3 :- Let vaccume in Room A
Step 4 :- Ask user to which room he want to clean.
Step 5 :- Move vaccume to user_said room
if dirt_is_present : clean
else raise message room cleaned.
Loop : step 4
Step 6 : Stop.

Output.

Enter State of A (0 for clean, 1 for dirty) : 1
Enter state of B (0 for clean, 1 for dirty) : 0
Enter state of C (0 for clean, 1 for dirty) : 1
Enter state of D (0 for clean, 1 for dirty) : 1
Enter starting location (A, B, C, or D) : B
Vaccume is in room B
B is clean.
Moving vaccume to C
C is dirty
cleaning C.....
Moving vaccum to D

Program 2

Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm
Algorithm:

Code:

```
                a) DFS
goal_state = '123456780'
                moves = { 'U': -3,
'D': 3,
'L': -1,
'R': 1
}

invalid_moves = {
0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
3: ['L'],                          5: ['R'],
6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction): index = state.index('0')
if direction in invalid_moves.get(index, []): return None

new_index = index + moves[direction] if new_index < 0 or new_index >= 9:
return None

state_list = list(state)
state_list[index], state_list[new_index] = state_list[new_index], state_list[index] return
''.join(state_list)

def print_state(state): for i in range(0, 9, 3):
print(' '.join(state[i:i+3]).replace('0', ' ')) print()

def dfs(start_state, max_depth=50): visited = set()
stack = [(start_state, [])] # Each element: (state, path)

while stack:
current_state, path = stack.pop()
```

```
if current_state in visited: continue

# Print every visited state print("Visited state:") print_state(current_state)

if current_state == goal_state: return path
visited.add(current_state) if len(path) >= max_depth:
continue

for direction in moves:
new_state = move_tile(current_state, direction) if new_state and new_state not in visited:
stack.append((new_state, path + [direction])) return None
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'): print("Start state:")
print_state(start) result = dfs(start)
if result is not None: print("Solution found!") print("Moves:", ' '.join(result))
print("Number of moves:", len(result)) print("1BM23CS345 Suhas B P\n")
current_state = start
for i, move in enumerate(result, 1):
current_state = move_tile(current_state, move) print(f"Move {i}: {move}")

print_state(current_state)
else:
print("No solution exists for the given start state or max depth reached.")
else:
print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```
Output:
Enter start state (e.g., 724506831): 123456078 Start state:
1 2 3
4 5 6
7 8

Visited state:
1 2 3
4 5 6
7 8

Visited state:
1 2 3
4 5 6
7 8

Visited state:
1 2 3

4 5 6
7 8

Solution found! Moves: R R Number of moves: 2 1BM23CS345 Suhas B P

Move 1: R
1 2 3
4 5 6
7 8

Move 2: R
1 2 3
4 5 6
7 8

Week-3

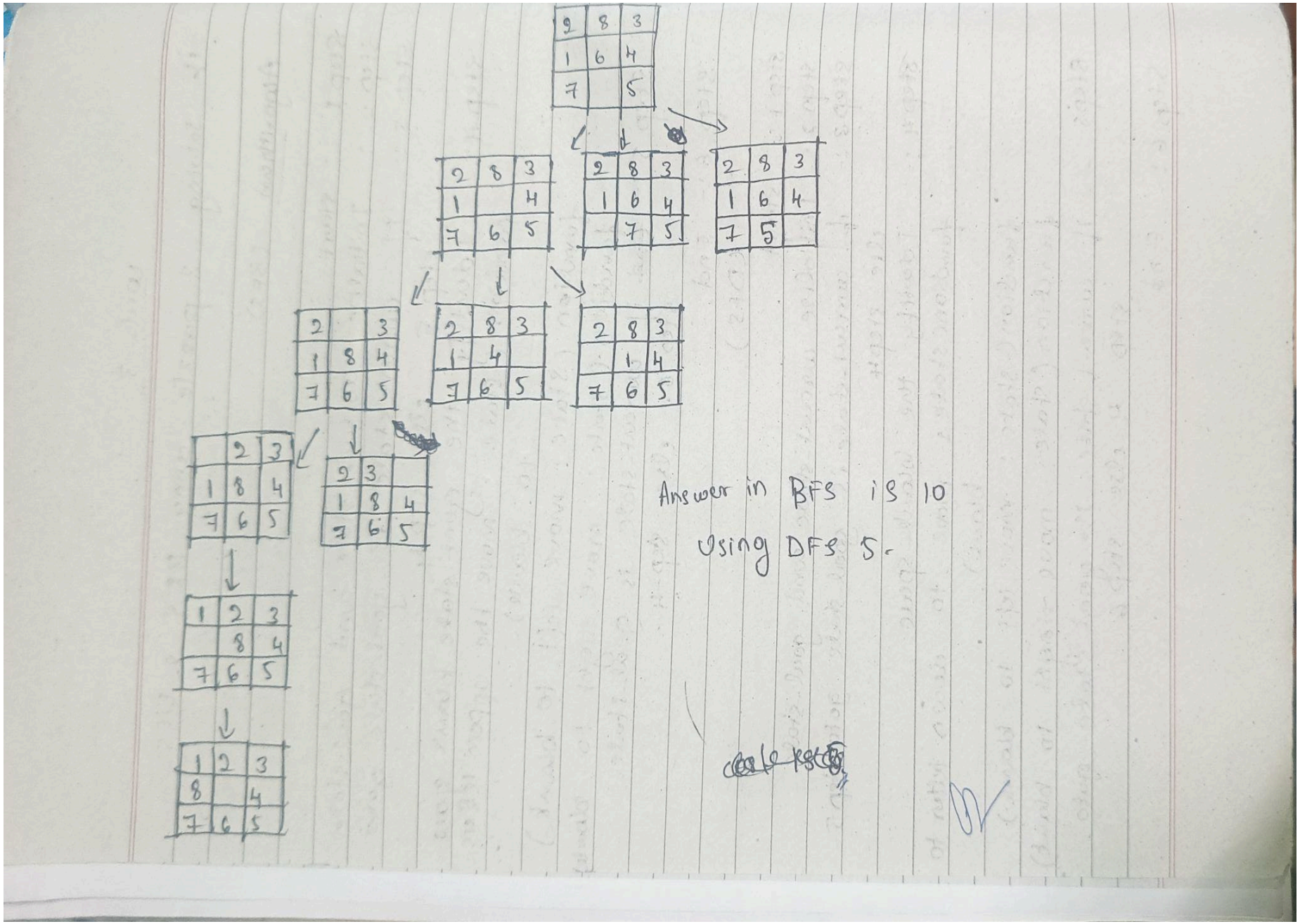
1) Solving 8 puzzle using BFS & DFS.

Algorithem (BFS)

Step 1 :- Start.
Step 2 :- Initialize current_state and goal_state
Step 3 :- If current_state is goal_state goto step 5 else step 4
Step 4 :- Identify the ~~goal state~~ blank space
function (state : move the upper letter to blank).
function (State : move left to blank)
function (State : move right to blank),
Step 5 :- ~~End~~ if current_state is goal_state.
step 6. else step-4.
Step 6 :- End.

(DFS)

Step 1 :- Start.
Step 2 :- Initialize current_state and goal_state.
Step 3 :- if current_state is goal_state goto step 5.
else step 4.
Step 4 :- Identify the blank_space.
function(state : move to down letter to blank).
function(State : move left to blank)
function(State : move right to blank)
Step 5 :- if current_state is goal_state goto step 4 else step 6.
Step 6 :- End.


Answer in BFS is 10
Using DFS 5.

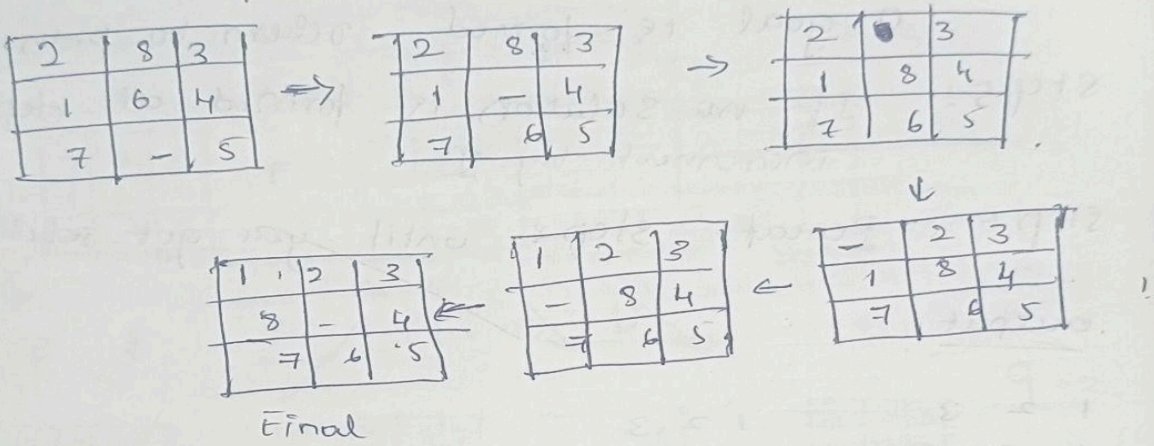
Output

Initial state:

2	8	3
1	6	4
7	-	5

Soln

Solution found in Five moves.



Final

A. Iterative Deepening Search goal_state = '123456780'

```
moves = { 'U': -3,
'D': 3,
'L': -1,
'R': 1
}

invalid_moves = {
0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
3: ['L'],       5: ['R'],
6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction): index = state.index('0')
if direction in invalid_moves.get(index, []): return None

new_index = index + moves[direction] if new_index < 0 or new_index >= 9:
return None

state_list = list(state)
state_list[index], state_list[new_index] = state_list[new_index], state_list[index] return
''.join(state_list)

def print_state(state): for i in range(0, 9, 3):
print(' '.join(state[i:i+3]).replace('0', ' ')) print()

def dls(state, depth, path, visited, visited_count): visited_count[0] += 1 # Increment visited states
count if state == goal_state:
```

```
return None visited.add(state)
for direction in moves:
new_state = move_tile(state, direction)
if new_state and new_state not in visited:
result = dls(new_state, depth - 1, path + [direction], visited, visited_count) if result is not None:
return result

visited.remove(state) return None

def iddfs(start_state, max_depth=50):
visited_count = [0] # Using list to pass by reference for depth in range(max_depth + 1):
visited = set()
result = dls(start_state, depth, [], visited, visited_count) if result is not None:
return result, visited_count[0] return None, visited_count[0]

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'): print("Start state:")
print_state(start)

result, visited_states = iddfs(start,15)
print(f"Total states visited: {visited_states}") if result is not None:
print("Solution found!") print("Moves:", ' '.join(result)) print("Number of moves:", len(result))
print("1BM23CS345 Suhas B P\n")

current_state = star
```

```
for i, move in enumerate(result, 1):
 current_state = move_tile(current_state, move) print(f"Move {i}: {move}") print_state(current_state)
 else:
 print("No solution exists for the given start state or max depth reached.")
 else:
 print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 123405678 Start state:
1 2 3
4  5
6 7 8
            Total states visited: 24298
Moves: R D L L U R D R U L L D R R
Number of moves: 14 1BM23CS302 Santhosh N

Move 1: R
1 2 3
4 5
6 7 8

Move 2: D
1 2 3
4 5 8
6 7

Move 3: L
1 2 3
4 5 8
6  7

Move 4: L
1 2 3
4 5 8
6 7
```

Move 5: U
1 2 3
5 8
4 6 7

Move 6: R
1 2 3
5 8
4 6 7

Move 7: D
1 2 3
5 6 8
4 7

Move 8: R
1 2 3
5 6 8
4 7

Move 9: U
1 2 3
5 6
4 7 8

Move 10: L
1 2 3
5 6
4 7 8

Move 11: L
1 2 3
5 6
4 7 8

Move 12: D
1 2 3
4 5 6
7 8

Move 13: R
1 2 3
4 5 6
7 8

Move 14: R
1 2 3
4 5 6
7 8

Iterative deepning

Algorithem

Step 1 :- start

Step 2 : set deapth unit d=0.

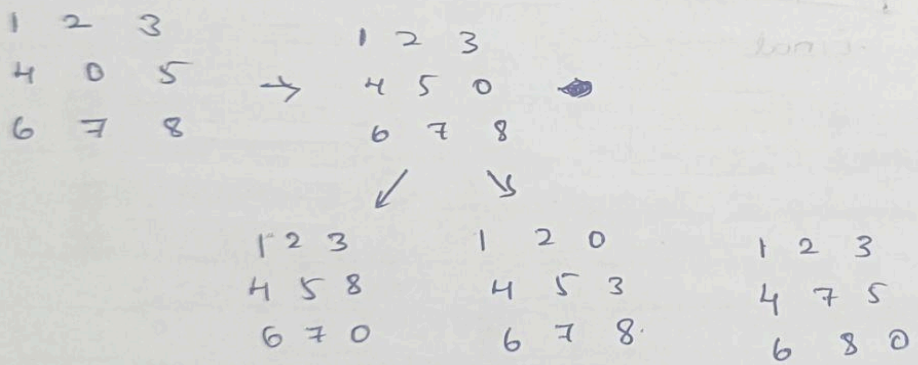
Step 3 :- perform the depth ~~or~~ limited DFS with current depth limited.

Step 4 :- Explore search tree to depth only if a goal is formed return to path.

Step 5 :- If no solution is formed at depth d increment by 1

Step 6 :- Repeat step 3 until you got solution.

outpat.



At depth = 2 solution is found.

Program 3

Implement A* search algorithm

Code:
MisplaceType
goal_state = '123804765' moves = {
'U': -3,
‘D': 3,
'L': -1,
'R': 1
}

invalid_moves = {
0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
3: ['L'], 5: ['R'],
6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction): index = state.index('0')
if direction in invalid_moves.get(index, []): return None

new_index = index + moves[direction] if new_index < 0 or new_index >= 9:
return None

state_list = list(state)
state_list[index], state_list[new_index] = state_list[new_index], state_list[index] return
''.join(state_list)

def print_state(state): for i in range(0, 9, 3):
print(' '.join(state[i:i+3]).replace('0', ' ')) print()

def misplaced_tiles(state):
"""Heuristic: count of tiles not in their goal position (excluding zero)."""" return sum(1 for i, val in
enumerate(state) if val != '0' and val != goal_state[i])

heapq.heappush(open_set, (misplaced_tiles(start_state), 0, start_state, [])) visited = set()

while open_set:
f, g, current_state, path = heapq.heappop(open_set) visited_count += 1

if current_state == goal_state: return path, visited_coun0t
if current_state in visited: continue

```
visited.add(current_state)

for direction in moves:
new_state = move_tile(current_state, direction) if new_state and new_state not in visited:
new_g = g + 1
new_f = new_g + misplaced_tiles(new_state)
heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction])) return None,
visited_count
# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'): print("Start state:")
print_state(start)
result, visited_states = a_star(start) print(f"Total states visited: {visited_states}") if result is not None:
print("Solution found!")
print("Moves:", ' '.join(result)) print("Number of moves:", len(result)) print("1BM23CS345 Suhas B
P\n")

current_state = start
g = 0 # initialize cost so far
for i, move in enumerate(result, 1):
new_state = move_tile(current_state, move)
g += 1
h = misplaced_tiles(new_state) f = g + h
print(f"Move {i}: {move}") print_state(new_state)
print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n") current_state = new_state
else:
print("No solution exists for the given start state.")
else:
print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 283164705 Start state:
2 8 3
1 6 4
7  5

Total states visited: 7 Solution found!
Moves: U U L D R Number of
Move 1: U
2 8 3
1  4
7 6 5
g(n) = 1, h(n) = 3, f(n) = g(n) + h(n) = 4 Move 2: U
2  3
```

1 8 4
7 6 5
g(n) = 2, h(n) = 3, f(n) = g(n) + h(n) = 5 Move 3: L
2 3
1 8 4
7 6 5
g(n) = 3, h(n) = 2, f(n) = g(n) + h(n) = 5 Move 4: D
1 2 3
8 4
7 6 5
g(n) = 4, h(n) = 1, f(n) = g(n) + h(n) = 5 Move 5: R
1 2 3
8 4
7 6 5

g(n) = 5, h(n) = 0, f(n) = g(n) + h(n) = 5
b.Manhattan distance.
import heapq
goal_state = '123456780' moves = {
'U': -3,
'D': 3,
'L': -1,
'R': 1
}

invalid_moves = {
0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
3: ['L'], 5: ['R'],
6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction): index = state.index('0')
if direction in invalid_moves.get(index, []): return None

new_index = index + moves[direction] if new_index < 0 or new_index >= 9:
return None

state_list = list(state)
state_list[index], state_list[new_index] = state_list[new_index], state_list[index] return
''.join(state_list)

def print_state(state): for i in range(0, 9, 3):
print(' '.join(state[i:i+3]).replace('0', ' ')) print()

def manhattan_distance(state): distance = 0

```
for i, val in enumerate(state): if val == '0':continue
goal_pos = int(val) - 1
current_row, current_col = divmod(i, 3) goal_row, goal_col = divmod(goal_pos, 3)
distance += abs(current_row - goal_row) + abs(current_col - goal_col) return distance


heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, [])) visited = set()

while open_set:
f, g, current_state, path = heapq.heappop(open_set) visited_count += 1

if current_state == goal_state: return path, visited_count

if current_state in visited: continue
visited.add(current_state)

for direction in moves:
new_state = move_tile(current_state, direction) if new_state and new_state not in visited:
new_g = g + 1
new_f = new_g + manhattan_distance(new_state) heapq.heappush(open_set, (new_f, new_g,
new_state, path + [direction]))
return None, visited_count # Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'): print("Start state:")
print_state(start)

result, visited_states = a_star(start) print(f"Total states visited: {visited_states}")
if result is not None:
print("Solution found!") print("Moves:", ' '.join(result)) print("Number of moves:", len(result))
print("1BM23CS345 Suhas B P\n")

current_state = start
g = 0 # initialize cost so far
for i, move in enumerate(result, 1):
new_state = move_tile(current_state, move)
g += 1
h = manhattan_distance(new_state) f = g + h
print(f"Move {i}: {move}") print_state(new_state)
print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n") current_state = new_state
else:
print("No solution exists for the given start state.")
else:
print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:
Enter start state (e.g., 724506831): 123678450 Start state:
1 2 3
6 7 8
4 5

Total states visited: 21 Solution found!
Moves: L U L D R R U L D R
Number of moves: 10 1BM23CS345 Suhas B P

Move 1: L
1 2 3
6 7 8
4 5
g(n) = 1, h(n) = 9, f(n) = g(n) + h(n) = 10 Move 2: U
1 2 3
6 8
4 7 5

g(n) = 2, h(n) = 8, f(n) = g(n) + h(n) = 10

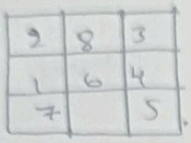
Move 3: L
1 2 3
6 8
4 7 5
g(n) = 3, h(n) = 7, f(n) = g(n) + h(n) = 10 Move 4: D
1 2 3
4 6 8
7 5
g(n) = 4, h(n) = 6, f(n) = g(n) + h(n) = 10 Move 5: R
1 2 3
4 6 8
7 5
g(n) = 5, h(n) = 5, f(n) = g(n) + h(n) = 10 Move 6: R
1 2 3
4 6 8
7 5
g(n) = 6, h(n) = 4, f(n) = g(n) + h(n) = 10 Move 7: U
1 2 3
4 6
7 5 8
g(n) = 7, h(n) = 3, f(n) = g(n) + h(n) = 10 Move 8: L
1 2 3
4 6
7 5 8

g(n) = 8, h(n) = 2, f(n) = g(n) + h(n) = 10

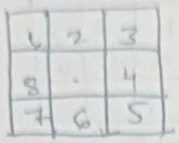
Move 9: D
1 2 3
4 5 6
7 8
g(n) = 9, h(n) = 1, f(n) = g(n) + h(n) = 10 Move 10: R
1 2 3
4 5 6
7 8

g(n) = 10, h(n) = 0, f(n) = g(n) + h(n) = 10

A* Algorithm

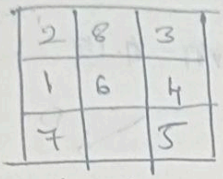


Initial

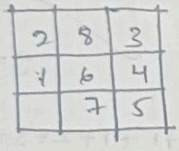


Final

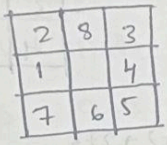
Solution :



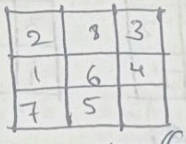
R U L



m = 6



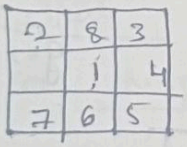
m = 4



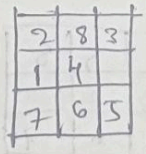
m = 6

g = 1

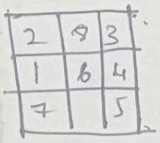
R L D U



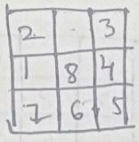
m = 5



m = 5



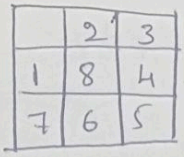
m = 5



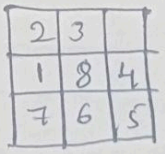
m = 3

g = 2

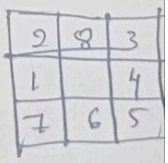
R D



m = 2



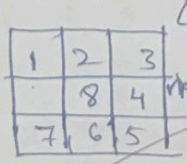
m = 4



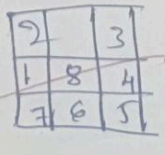
m = 4

g = 3

D L

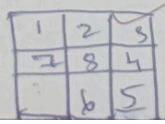


m = 1

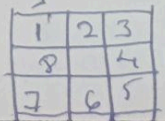


m = 3

D R



m = 2



m = 0

g = 4

Misplaced

1	5	8
3	2	
4	6	7

Initial

1	2	3
4	5	6
7	8	

Final

1	5	8
3	2	
4	6	7

m = 15

1+3+1+1+3

g=1

L U D

1	5	8
3		2
4	6	7

m = 14

2+3+1+1+2+2+3+2

1	5	
3	2	8
4	6	7

1+3+1+1+2+2+2

m = 12

1	5	8
3	2	7
4	6	

g=1

1+3+1+1+2+3+3

14

L D

1		5
3	2	8
4	6	7

1+3+1+2+2+2+2

m = 13

1	5	8
3	2	
4	6	7

g=2

1+3+1+1+2+2+3

m=13

Algorithm for misplace & Manhattan Distance

S1 : Start

S2 : Evaluate nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to goal node.

S3 : calculate f(n) = g(n) + h(n)

S4 : f(n) is evaluation function which gives the cheapest solution cost.

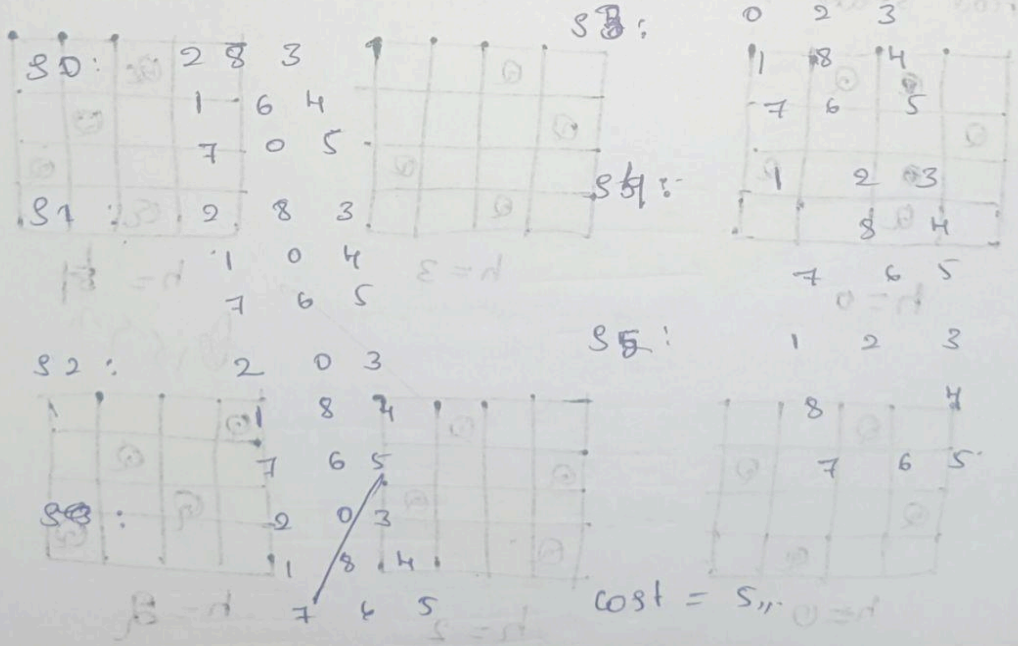
S5 : g(n) is exact cost to reach node n from the initial state.

S6 : h(n) is an estimation of the assumed cost from current state (n) to reach the goal.

S7 : End.

Output

1) For Misplace and Manhattan.



Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

```
import random
import time

def print_board(state):
"""Prints the chessboard for a given 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 compute_heuristic(state):
"""Computes the number of attacking pairs of queens.""" h = 0
n = len(state) for i in range(n):
for j in range(i + 1, n):
if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j): h += 1
return h

def get_neighbors(state):
"""Generates all possible neighbors by moving one queen in its column.""" neighbors = []
n = len(state)
for col in range(n): for row in range(n):
if row != state[col]: neighbor = list(state) neighbor[col] = row
neighbors.append(neighbor) return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
"""Hill climbing algorithm with verbose output at each step.""" current = initial_state
current_h = compute_heuristic(current)

step = 0

print(f"Initial state (heuristic: {current_h}):") print_board(current)
time.sleep(step_delay)

while True:
```

```
neighbors = get_neighbors(current) next_state = None
next_h = current_h

for neighbor in neighbors:
h = compute_heuristic(neighbor) if h < next_h:
next_state = neighbor next_h = h

if next_h >= current_h:
print(f"Reached local minimum at step {step}, heuristic: {current_h}") return current, current_h

current = next_state current_h = next_h step += 1
print(f"Step {step}: (heuristic: {current_h})") print_board(current)
time.sleep(step_delay)

def solve_n_queens_verbose(n, max_restarts=1000):
"""Solves N-Queens problem using hill climbing with restarts and verbose output.""" for attempt in
range(max_restarts):
print(f"\n=== Restart {attempt + 1} ===\n")
initial_state = [random.randint(0, n - 1) for _ in range(n)] solution, h = hill_climb_verbose(initial_state)
if h == 0:
print(f"⬛ Solution found after {attempt + 1} restart(s):") print_board(solution)
return solution else:
print(f"+ No solution in this attempt (local minimum).\n") print("Failed to find a solution after max
restarts.")

return None

# --- Run the algorithm ---
if   name                    == "  main  ":
N = int(input("Enter the number of queens (N): ")) solve_n_queens_verbose(N)
print("1BM23CS345 Suhas B P")
```

Output:

```
Enter the number of queens (N): 4

=== Restart 1 ===

Initial state (heuristic: 3):
Q . Q .
. Q . .
. . . Q
. . . .

Step 1: (heuristic: 1)
. . Q .
```

. Q . .
. . . Q
Q . . .

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).

=== Restart 2 ===

Initial state (heuristic: 3):
. Q . .
. . Q .
. . . . Q . . Q

Step 1: (heuristic: 1)
. Q . .
. . Q .
Q . . .
. . . Q

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).

=== Restart 3 ===

Initial state (heuristic: 2):
. . . .
. Q . Q
. . . . Q . Q .

Step 1: (heuristic: 1)
. Q . .
. . . Q
. . . . Q . Q .

Step 2: (heuristic: 0)
. Q . .
. . . Q
Q . . .
. . Q .

Reached local minimum at step 2, heuristic: 0
  Solution found after 3 restart(s):

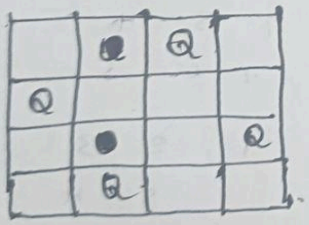
. Q . .
. . . Q
Q . . .
. . Q .
1BM23CS345 Suhas B P

Lab-5 Hill climbing

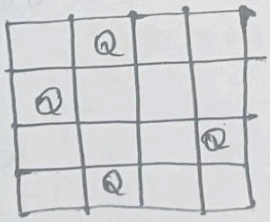
1) Define current state as initial state.
2) Loop until goal state is reached, or no more operation is applied.
3) Apply on operand
4) compare new state with goal state
5) Quit
6) Evaluate newstate with its heuristics
7) compare.
8) if newstate is close to goal state then update current state.
9) display.
10) End.

4-Queens problem.

Initial state.

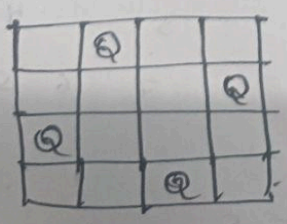


h=0

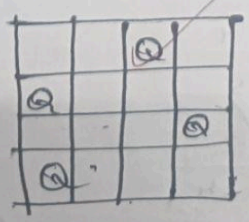


h=3

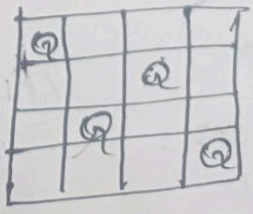
h=



h=0



h=2



h=2

Program 5

Simulated Annealing to Solve 8-Queens problem

Code:

```
import random import math

def compute_heuristic(state): """Number of attacking pairs.""" h = 0
n = len(state) for i in range(n):
for j in range(i + 1, n):
if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

h += 1
return h

def random_neighbor(state):
"""Returns a neighbor by randomly changing one queen's row.""" n = len(state)
neighbor = state[:]
col = random.randint(0, n - 1) old_row = neighbor[col]
new_row = random.choice([r for r in range(n) if r != old_row]) neighbor[col] = new_row
return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
"""Simulated Annealing with dual acceptance strategy."""
current = [random.randint(0, n - 1) for _ in range(n)] current_h = compute_heuristic(current)
temperature = initial_temp

for step in range(max_iter): if current_h == 0:
print(f" Solution found at step {step}") return current

neighbor = random_neighbor(current) neighbor_h = compute_heuristic(neighbor) delta = neighbor_h
- current_h

if delta < 0:
current = neighbor current_h = neighbor_h
else:
# Dual acceptance: standard + small chance of higher uphill move probability = math.exp(-delta /
temperature)
if random.random() < probability: current = neighbor
current_h = neighbor_h

temperature *= cooling_rate
if temperature < 1e-5: # Restart if stuck temperature = initial_temp
```

```
current = [random.randint(0, n - 1) for _ in range(n)] current_h = compute_heuristic(current)

print("+ Failed to find solution within max iterations.") return None

# --- Run the algorithm ---
if   name                    == "  main  ":
N = int(input("Enter number of queens (N): ")) solution = dual_simulated_annealing(N)

if solution:
print("Position format:")
print("[", " ".join(str(x) for x in solution), "]") print("Heuristic:", compute_heuristic(solution))
print("1BM23CS345 Suhas B P")
```

Output:

```
Enter number of queens (N): 8
   Solution found at step 675 Position format:
[ 3 0 4 7 5 2 6 1 ]
Heuristic: 0
1BM23CS3345 Suhas B P
```

Simulated Annealing

1. current ← initial state.
2. T ← a large positive value.
3. while T > 0 do.
4. next ← a random neighbour of current
5. ΔE ← current.cost − next.cost
6. if ΔE > 0 then
7. current ← next
8. else current ← next with probability $p = e^{\frac{\Delta E}{T}}$
9. end if decrease T.
10. end while.
13. return current.

Output

The best position found is : [5, 1, 1, 4, 7, 5, 0, 2]
The number of queens that are not attacking each other is : 5

15/9/25

Program 6

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

Code:

```
from itertools import product

# ----------------- Propositional Logic Symbols ----------------- class Symbol:
def  init (self, name): self.name = name

def  invert (self): # ~P return Not(self)

def  and (self, other): # P & Q return And(self, other)

def  or (self, other): # P | Q return Or(self, other)

def  rshift (self, other): # P >> Q (implication) return Or(Not(self), other)

def   eq  (self, other): # P == Q (biconditional) return And(Or(Not(self), other), Or(Not(other), self))

def eval(self, model): return model[self.name]

def symbols(self): return {self.name}

def  repr (self): return self.name

class Not:
def  init (self, operand): self.operand = operand

def eval(self, model):
return not self.operand.eval(model)

def symbols(self):
return self.operand.symbols()
```

```
    def   repr  (self):
    return f"~{self.operand}"

    def  invert (self): # allow ~~A return Not(self)


class And:
    def _init_(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) and self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def _repr_(self):
        return f"({self.left} & {self.right})"

    def _invert_(self): # allow ~(A & B) return
        Not(self)


class Or:
    def _init_(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) or self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def _repr_(self):
        return f"({self.left} | {self.right})"

    def _invert_(self): # allow ~(A | B) return
        Not(self)
```

```
tt_entails(kb, alpha, show_table=False):
    symbols = sorted(list(kb.symbols() | alpha.symbols()))
    if show_table:
        print_truth_table(kb, alpha, symbols)
    return tt_check_all(kb, alpha, symbols, {})


def tt_check_all(kb, alpha, symbols, model):
    if not symbols: # all symbols assigned
        if kb.eval(model): # KB is true
            return alpha.eval(model)
        else:
            return True # if KB is false, entailment holds
    else:
        P, rest = symbols[0], symbols[1:]

        model_true = model.copy()
        model_true[P] = True
        result_true = tt_check_all(kb, alpha, rest, model_true)

        model_false = model.copy()
        model_false[P] = False
        result_false = tt_check_all(kb, alpha, rest, model_false)

        return result_true and result_false


# ---------- Truth Table Printer ----------
def print_truth_table(kb, alpha, symbols):
    header = symbols + ["KB", "Query"]
    print(" | ".join(f"{h:^5}" for h in header))
    print("-" * (7 * len(header)))

    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = kb.eval(model)
        alpha_val = alpha.eval(model)
        row = [str(model[s]) for s in symbols] + [str(kb_val), str(alpha_val)]
        print(" | ".join(f"{r:^5}" for r in row))
```

```
    print()
C = Symbol("C")
T = Symbol("T")


c = T | ~T

# KB: P → Q
kb1 = ~ (S|T)
# Query: Q
alpha1 = S & T


print("Knowledge Base:", kb1)
print("Query:", alpha1)
print()
result = tt_entails(kb1, alpha1, show_table=True)
print("Does KB entail Query?", result)
```

Output:

```
Knowledge Base: (P | (Q & P))
Query: (Q | P)

  P  |  Q  | KB  | Query
--------------------
False | False | False | False
False | True | False | True
True | False | True | True
True | True | True | True

Does KB entail Query? True
```

6) Create a knowledge base using propositional logic & show that the given query entails the knowledge base / not

Truth table for connectives

P	Q	¬P	P∧Q	P∨Q	P⇒Q
false	false	true	false	false	true
false	true	true	false	true	false
false	false	true	false	true	true
true	true	false	true	true	

propositional inference: Enumeration method.

$\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	A ∨ C	B ∨ ¬C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	T	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

$KB \models \alpha$ holds (KB entails α)

Algorithm

1) List all variables
   - Find all the symbols that appear in KB & α

Ex: A, B, C

2) Try every possibility:

* Each symbol can be true / false
* So we test all combination.

3) Create KB.

For each combination, see if KB is true

4) Check α

* if KB is true, then α must also true, then α must also be true
* If KB is false, we don't care about α in that row.

5) Final decision.

* If in all cases, where KB is true, α is also true → KB entails α
* If in any case KB is true but false → KB does not entail.

Output

KB; Not

Enter Query (alpha): t

Truth Table

Not	I	KB	alpha
True	True	True	True
True	False	True	False
False	True	False	True
False	False	False	False

Result: False.

Program 7

Implement unification in first order logic

Code:

```
class
   UnificationError(
   Exception): pass

def occurs_check(var, term):

   """Check if a variable occurs in a term (to prevent infinite
   recursion)""" if var == term:
      return True
   if isinstance(term, tuple): # Term is a compound
      (function term) return any(occurs_check(var,
      subterm) for subterm in term)
   return False

def unify(term1, term2, substitutions=None):
   """Try to unify two terms, return the MGU (Most
   General Unifier)""" if substitutions is None:
      substitutions = {}

   # If both terms are equal, no further
   substitution is needed if term1 == term2:
      return substitutions

   # If term1 is a variable, we substitute
   it with term2 elif isinstance(term1,
   str) and term1.isupper():
      # If term1 is already
      substituted, recurse if
      term1 in substitutions:
         return unify(substitutions[term1], term2,
      substitutions) elif occurs_check(term1,
      term2):
         raise UnificationError(f"Occurs check fails: {term1}
      in {term2}") else:
         substitutions[term
```

```
        1] = term2 return
        substitutions

    # If term2 is a variable, we substitute
    it with term1 elif isinstance(term2,
    str) and term2.isupper():
        # If term2 is already
        substituted, recurse if
        term2 in substitutions:
            return unify(term1, substitutions[term2],
        substitutions) elif occurs_check(term2,
        term1):
            raise UnificationError(f"Occurs check fails: {term2}
        in {term1}") else:
            substitutions[term
            2] = term1 return
            substitutions

    # If both terms are compound (i.e., functions), unify their
    parts recursively elif isinstance(term1, tuple) and
    isinstance(term2, tuple):
        # Ensure that both terms have the same "functor" and number of arguments

        # if len(term1) != len(term2):
        #     raise UnificationError(f"Function arity mismatch: {term1} vs {term2}")

        for subterm1, subterm2 in zip(term1, term2):
            substitutions = unify(subterm1, subterm2,

        substitutions) return substitutions

    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# Define the terms
as tuples term1 =
('p', 'b', 'X', ('f',
('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
```

```
    # Find the MGU
    result = unify(term1,
    term2) print("Most
    General Unifier
    (MGU):") print(result)
except
    UnificationError
    as e:
    print(f"Unification
    failed: {e}")
finally:
```

Output:

```
Most General Unifier (MGU):
{'Z': 'b', 'X': ('f', 'Y'),
'Y': ('g', 'Z')}
```

Lab - 4

Unification Algorithm

Unification :- It is process to find substitution make different FOL (first order logic).

* Unify (knows (John, x), knows (John, Jane))

θ = x/Jane

x/Jane.

Unify (knows (John, Jane), know (John, Jane)).

2) Unify (knows (John, x), know (y, Bill))

θ = y/John

knows (John, x), know (John, Bill)

θ = x/Bill

knows (John, Bill), knows (John, Bill).

3) Find MGU of {P(b, x, f(g(z)))}

{P(z, f(y), f(y))}

θ = b/z

~~knows {P(z, x~~

θ = x/f(y)

θ = g(z)/y.

4) Find MGU of {Q(a, g(x,a), f(y)), Q(a, g(f(b),a), x)}

$\theta = x / f(b)$

Unify {Q(a, g(f(b),a), f(y))}

$\theta = f(y) / x$

Unify {Q(a, g(f(b),a), x), Q(a, g(f(b),a), x)}

5) Find MGU of {P(f(a), g(y)), P(x, x)}

$\theta = f(a) / x$

{P(x, g(y)), P(x, x)}

unification fails.

6) Unify {prime(11) & prime(y)}

$\theta = 11 / y$

Unify {prime(y) & prime(y)}.

Algorithm

Unify(ψ_1, ψ_2)

Step 1: If ψ_1 / ψ_2 is a variable or constant, then

a) If ψ_1 occurs then return NIL.

b) Else if ψ_1 is variable.

a. then if ψ_1 occurs in ψ_2, then returns Failure.

{ψ_2 / ψ_1}.

c) Else if ψ_2 is a variable
   a. If ψ_2 occure in ψ_1 then return Failure.
   b. Else return $\{(\psi_1 / \psi_2)\}$.
d) Else return $\{(\psi_1/\psi_2)\}$ Failure.

Step 2: If initial predicate symbol in ψ_1 & ψ_2 are same, then return failure.

Step 3: If ψ_1 & ψ_2 have different number of arguments then return failure.

Step 4: Set substitution set (SUBST) to NIL.

Step 5: For i = 1 to number of element in ψ_1
   a) If S = failure then return Failure.
   b) If S ≠ NIL then do,
      a. Apply S to the remainder of both ψ_1 & ψ_2
      b. SUBST = APPEND(S, SUBST).

Step 6: Return SUBT.

Output,

Input: $P(b, x, f(g(z)))$ & $P(z, f(y), f(y))$

Trace;

Unify b with z

Unify x with f(y)

Unify f(g(z)) with f(y)

Unify g(z) with y

Final MGU:

$z \rightarrow b$

$x \rightarrow f(g(z))$

$y \rightarrow g(z)$

Program 8

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

```
facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Missile(T1)",
    "Owns(A, T1)"
}


rules = [
    ("Enemy(x, America)", "Hostile(x)"),
    ("Missile(x)", "Weapon(x)"),
    ("Missile(x) ⋀ Owns(A, x)", "Sells(Robert, x, A)"),
    ("American(p) ⋀ Weapon(q) ⋀ Sells(p, q, r) ⋀ Hostile(r)", "Criminal(p)")
]

def apply_rules(facts):
    new_facts = set()


    if "Enemy(A, America)" in facts and "Hostile(A)" not in facts:
        print("Step 1: Enemy(A, America) → Hostile(A)")
        new_facts.add("Hostile(A)")


    if "Missile(T1)" in facts and "Weapon(T1)" not in facts:
        print("Step 2: Missile(T1) → Weapon(T1)")
        new_facts.add("Weapon(T1)")


    if "Missile(T1)" in facts and "Owns(A, T1)" in facts and "Sells(Robert, T1, A)"
not in facts:
        print("Step 3: Missile(T1) ⋀ Owns(A, T1) → Sells(Robert, T1, A)")
        new_facts.add("Sells(Robert, T1, A)")

    if {"American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"} <=
facts and "Criminal(Robert)" not in facts:
        print("Step 4: American(Robert) ⋀ Weapon(T1) ⋀ Sells(Robert, T1, A) ⋀
Hostile(A) → Criminal(Robert)")
        new_facts.add("Criminal(Robert)")

    return new_facts


step = 1
while True:
    new_facts = apply_rules(facts)
    if not new_facts:
        break
    facts |= new_facts
    step += 1
```

```
print("\n✅ Final Facts:")
for fact in facts:
    print(fact)
```

```
Step 1: Enemy(A, America) → Hostile(A)
Step 2: Missile(T1) → Weapon(T1)
Step 3: Missile(T1) ⋀ Owns(A, T1) → Sells(Robert, T1, A)
Step 4: American(Robert) ⋀ Weapon(T1) ⋀ Sells(Robert, T1, A) ⋀ Hostile(A) →
Criminal(Robert)

✅ Final Facts:
Weapon(T1)
Criminal(Robert)
Hostile(A)
Sells(Robert, T1, A)
Missile(T1)
Enemy(A, America)
Owns(A, T1)
American(Robert)
```

First order logic

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

Rules

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

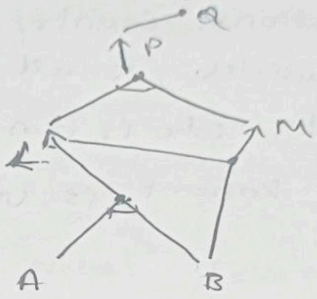
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

fact $\{A, B\}$

Prove Q



Algorithm

function FOL-FC-ASK (KB, α) returns a substitution/fa

inputs: KB, the knowledge base, a set of FOL de

α, the query, an atomic sentence

Local variable: new, the new sentence inferred of each

repeat until new is empty

new $\leftarrow \{\}$

for each rule in KB do

$(P_1 \wedge \dots \wedge P_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIAB

for each θ such that SUBST(θ, $P_1 \wedge \dots \wedge P_n$) = SUBST($\theta$, $P_1' \wedge \dots$

for some $P_1' \dots P_n'$ in KB.

$q' \leftarrow$ SUBST(θ, q)

If q' does not unify with some sentence already in KB / new then.

Add q to KB
$\phi \leftarrow$ UNIFY($\forall(\phi), \alpha$)
if ϕ is not fail then return ϕ
Add new to KB
return false.

Problem

As per law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America has some missiles, & all the missiles were sold to it by Robert, who is an American citizen.

Prove that "Robert is criminal"

Representation in FOL.

it is a crime for an American to sell weapons to hostile nations

Lets say P, q & r are variables

American (P) $\wedge$ weapon(q) $\wedge$ sells(P, q, r) $\wedge$ Hostile(r) $\Rightarrow$ Criminal(P)

Country A has some missiles.

$\exists x$ owns(A, x) $\wedge$ Missile(x)

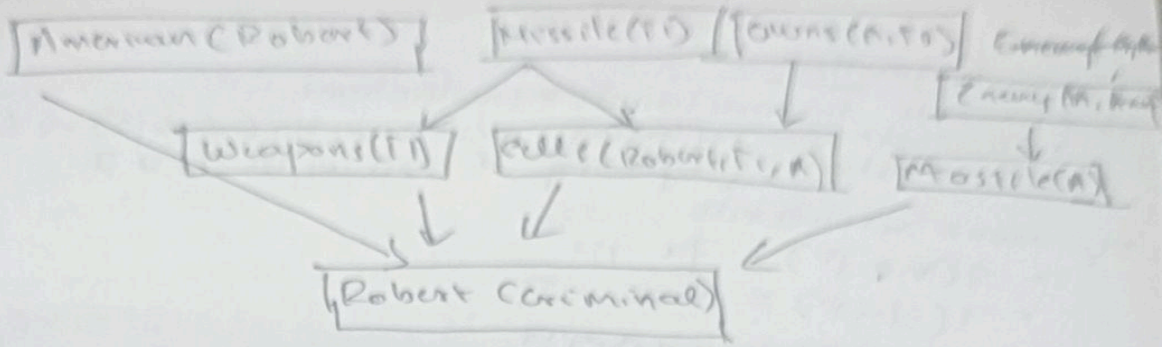
Existential Instantiation, introducing a new constant T1

owns(A, T1)

Missile(T1)

All of the missiles were sold to country A by Robert

$\forall x$ Missile(x) $\wedge$ owns(A, x) $\Rightarrow$ sells(Robert, x, A)



. American(p) ∧ Weapon(a) ∧ Sells(p,a,r) ∧ Hostile(r)

=> Criminal(p).

Output →

All condition are met Robert is Criminal.

Final Facts:

Enemy(A, America)

American(Robert)

Hostile(A)

Criminal(Robert).

13/10/25

Program 9

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

```
class Literal:
    def __init__(self, name, negated=False):
        self.name = name
        self.negated = negated

    def __repr__(self):
        return f"¬{self.name}" if self.negated else self.name

    def __eq__(self, other):
        return isinstance(other, Literal) and self.name == other.name and
self.negated == other.negated

    def __hash__(self):
        return hash((self.name, self.negated))
print('Suhas B P (1BM23CS345)')
def convert_to_cnf(sentence):
    return sentence  # Placeholder for CNF conversion logic

def negate_literal(literal):
    return Literal(literal.name, not literal.negated)

def resolve(clause1, clause2):
    resolvents = []
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1.name == literal2.name and literal1.negated !=
literal2.negated:
                new_clause = (set(clause1) - {literal1}) | (set(clause2) -
{literal2})
                resolvents.append(frozenset(new_clause))
    return resolvents

def prove_conclusion(premises, conclusion):
    cnf_premises = [convert_to_cnf(p) for p in premises]
    cnf_conclusion = convert_to_cnf(conclusion)
    negated_conclusion = frozenset({negate_literal(lit) for lit in cnf_conclusion})

    clauses = set(frozenset(p) for p in cnf_premises)
    clause_list = list(clauses)
    id_map = {}
    parents = {}
    clause_id = 1
```

```
    for c in clause_list:
        id_map[c] = f"C{clause_id}"
        clause_id += 1

    id_map[negated_conclusion] = f"C{clause_id} (¬Conclusion)"
    clauses.add(negated_conclusion)
    clause_list.append(negated_conclusion)
    clause_id += 1

    print("\n--- Initial Clauses ---")
    for c in clause_list:
        print(f"{id_map[c]}: {set(c)}")

    print("\n--- Resolution Steps ---")

    while True:
        new_clauses = set()
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                for r in resolvents:
                    if r not in id_map:
                        id_map[r] = f"C{clause_id}"
                        parents[r] = (id_map[clause_list[i]],
id_map[clause_list[j]])
                        clause_id += 1

                    print(f"{id_map[r]} = RESOLVE({id_map[clause_list[i]]},
{id_map[clause_list[j]]}) -> {set(r)}")

                    if not r:
                        print(f"\nEmpty clause derived! ({id_map[r]})")
                        print("\n--- Resolution Tree ---")
                        print_resolution_tree(parents, id_map, r)
                        return True

                    new_clauses.add(r)

        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived.")
            return False

        clauses |= new_clauses
        clause_list = list(clauses)

def print_resolution_tree(parents, id_map, empty_clause):
    """Recursively print resolution tree leading to the empty clause."""
    def recurse(clause):
        if clause not in parents:
            print(f" {id_map[clause]}: {set(clause)}")
```

```
            return
        left, right = parents[clause]
        print(f" {id_map[clause]} derived from {left} and {right}")
        for parent_clause, parent_name in zip([k for k, v in id_map.items() if v in
[left, right]], [left, right]):
            recurse(parent_clause)

def parse_literal(lit_str):
    lit_str = lit_str.strip()
    if lit_str.startswith("¬") or lit_str.startswith("~"):
        return Literal(lit_str[1:], True)
    return Literal(lit_str, False)

def get_user_input():
    premises = []
    num_premises = int(input("Enter number of premises: "))
    for i in range(num_premises):
        clause_str = input(f"Enter clause {i+1} (e.g., A, ¬B, C): ")
        literals = {parse_literal(l) for l in clause_str.split(",")}
        premises.append(literals)

    conclusion_str = input("Enter conclusion (e.g., C or ¬C): ")
    conclusion = {parse_literal(l) for l in conclusion_str.split(",")}
    return premises, conclusion

if __name__ == "__main__":
    premises, conclusion = get_user_input()

    if prove_conclusion(premises, conclusion):
        print("\n Conclusion can be proven from the premises.")
    else:
        print("\n Conclusion cannot be proven from the premises.")
```

```
Suhas B P (1BM23CS345)
Enter number of premises: 4
Enter clause 1 (e.g., A, ¬B, C): A
Enter clause 2 (e.g., A, ¬B, C): -A
Enter clause 3 (e.g., A, ¬B, C): -B
Enter clause 4 (e.g., A, ¬B, C): c
Enter conclusion (e.g., C or ¬C): c

--- Initial Clauses ---
C1: {A}
C2: {-B}
C3: {c}
C4: {-A}
C5 (¬Conclusion): {¬c}
```

```
--- Resolution Steps ---
C6 = RESOLVE(C3, C5 (¬Conclusion)) -> set()

Empty clause derived! (C6)

--- Resolution Tree ---

 Conclusion can be proven from the premises.
```

Algorithm

1) Eliminate biconditionals & implications
   - Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2) Move $\neg$ inwards
   - $\neg(\forall x\ p) \equiv \exists x\ \neg p$
   - $\neg(\exists x\ p) \equiv \forall x\ \neg p$,
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,

3) Standardize variables apart by renaming them

4) Skolemize: each existential variable is replace by Skolem constant / skolem function.
   - For instance, $\exists x$ Rich(x) becomes Rich(G1) where G1 is a new Skolem constant.
   - "Everyone has a heart" $\forall x$ person(x) $\Rightarrow \exists y$ Heart(y) $\wedge$ Has(x,y) becomes $\forall x$ person(x) $\Rightarrow$ Heart(H(x)) $\wedge$ Has(x, H(x))

5) Drop universal qualifiers
   - for instance, $\forall x$ person(x) becomes person(x).

6) Distribute $\wedge$ over $\vee$:
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

Proof by Resolution:

Given Premises: John likes all kind of food. Apples & vegetables are food. Anything anyone eats & not killed is food. Harry eats everything that Anil eats. Anil eats peanuts & still alive. Harry not killed. Anyone who is not killed implies Anil is alive.

prove by resolution: John likes peanuts.

Representation in FOL:

a) $\forall x$: food(x) $\rightarrow$ likes(John, x)

b) food(Apple) $\wedge$ food(vegetables)

c) $\forall x\ \forall y$: eats(x, y) $\wedge$ $\neg$ killed(x) $\rightarrow$ food(y)

d) eats(Anil, peanuts) $\wedge$ alive(Anil)

e) $\forall x$: eats(Anil, x) $\rightarrow$ eats(Harry, x)

f) $\forall x$: alive(x) $\rightarrow$ $\neg$ killed(x)

g) $\forall x$: $\neg$ killed(x) $\rightarrow$ alive(x)

backspace

prove : b) likes (John, Peanuts)

Proof by Resolution:

a) ¬food (x) V likes (John, x)
b) food (Apple)
c) food (Vegetable)
d) ¬eats (y, z) V killed (y) V food (z)
e) eats (Anil, peanuts)
f) alive (Anil)
g) ¬eats (Anil, w) V eats (Harry, w)
h) killed (g) V alive (g)
i) ¬alive (k) V ¬killed (k)
j) likes (John, Peanuts).

Output

Goal Prove likes (John, Peanuts),
Negated Goal : { 'Not likes (John, Peanuts)' }
Step 1: Resolving Negated Goal with clause a
{'Not likes (John, Peanuts)'} Result 1: {Not food(Peanuts)}

Step 2: Resolving Result 1 with clause d
Result 2: {'killed(y)', 'Not eats(y, Peanuts)'}
Step 3: Resolving Result 2 with clause e
Result 3: {killed(Anil)'}
Step 4 : Resolving Result 3 with clause i
Result 4 : {Not alive(Anil)'}
Step 5 : Resolving Result 4 with clause f
{Not alive (Anil)'} => {alive(Anil)'} Result 5 : set()

Question: The original goal likes (John, Peanuts) is True

¬likes(John, Peanuts)
¬food(x) V likes(John, x)
{Peanuts/x}
¬food(Peanuts)
¬eats(y, z) V killed(y) V food(z)
{Peanuts/z}
¬eats(y, Peanuts) V killed(y)
eats (Anil, Peanuts)

Hence proved.

Program 10

Implement Alpha-Beta Pruning.

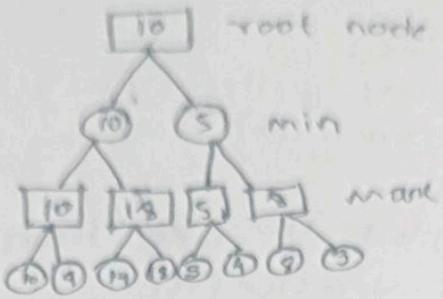
```
import math

def alpha_beta(node, depth, alpha, beta, maximizingPlayer, game_tree):


    if depth == 0 or isinstance(game_tree[node], int):
        return game_tree[node]

    if maximizingPlayer:
        maxEval = -math.inf

        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, False,
game_tree)

            maxEval = max(maxEval, eval)

            alpha = max(alpha, eval)

            if beta <= alpha:
                break

        return maxEval

    else:
        minEval = math.inf

        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, True, game_tree)

            minEval = min(minEval, eval)

            beta = min(beta, eval)

            if beta <= alpha:
                break
```

```
        return minEval


game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': 3,
    'E': 5,
    'F': 2,
    'G': 9
}

best_value = alpha_beta(
    'A', depth=3, alpha=-math.inf, beta=math.inf,
    maximizingPlayer=True, game_tree=game_tree
)

print("Best value for maximizer:", best_value)
```

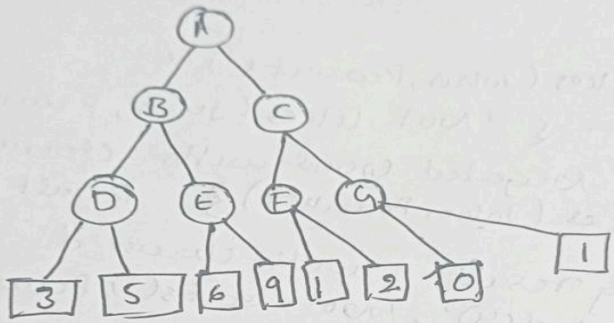
output:

```
Best value for maximizer: 3
```

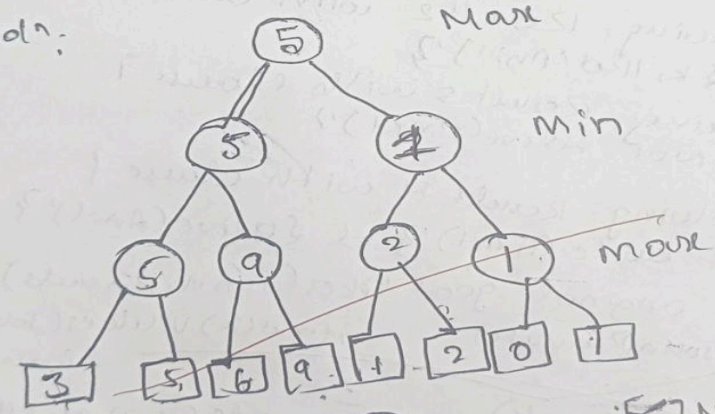
Min - Max Algorithm



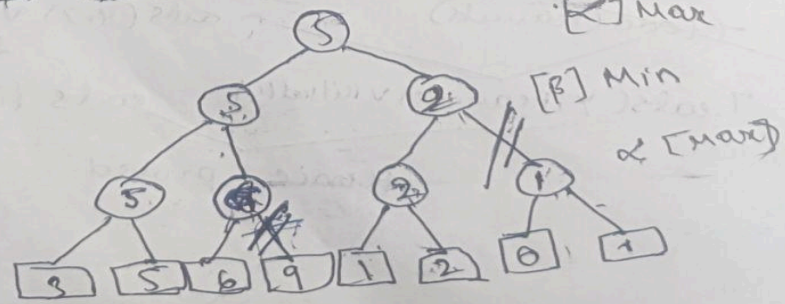
Solve using min-max & Alpha-Beta



Soln:



$\alpha \geq \beta$



Algorithm

function Alpha-Beta search(state) returns an action
  $v \leftarrow$ MAX-value(state, $-\infty$, $+\infty$)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in Actions(state) do
    $v \leftarrow$ MAX(v, MIN-value(Result(s,a), α, β))
    if $v \geq \beta$ then return v
    $\alpha \leftarrow$ MAX(α, v)
  return v.

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each a in Actions(state) do
    $v \leftarrow$ MIN(v, MAX-value(Result(s,a), α, β))
    if $v \leq \alpha$ then return v
    $\beta \leftarrow$ MIN(β, v)
  return v.

output

Final optimal ~~too~~ value for root node is 5

30/10/25