Artificial Intelligence Lab Report

Submitted by

Yashraj Sinha (1BM21CS335)
Batch: 3

Course: Artificial Intelligence
Course Code: 24CS5PCAIN
Sem & Section: 5F

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019

2023-2024

Table of contents

Program 1 - Tic Tac toe

Algorithm

4/10/24 LAB-1

(1) Implementing tic-tac-toe algorithm using python:

```
→ if isMaximizingPlayer:
      bestValue = -infinity
      for each child in node
      value = minimax(child, depth+1, false)
      bestValue = max(bestValue, value)
      return bestValue

  else
      bestValue = +infinity
      for each child in node
      value = minimax(child, depth+1, true)
      bestValue = min(bestValue, value)
      return bestValue
```

4/10/24

Code

```
import random

import math



def print_board(board):

    for row in board:

        print(" | ".join(row))

        print("-" * 9)



def check_winner(board, mark):

    # Check rows, columns, and diagonals for a win

    for row in board:

        if all(cell == mark for cell in row):

            return True



    for col in range(3):

        if all(board[row][col] == mark for row in range(3)):

            return True
```

```
    if all(board[i][i] == mark for i in range(3)) or all(board[i][2 - i] == mark for i in range(3)):
        return True

    return False

def get_available_moves(board):
    return [(r, c) for r in range(3) for c in range(3) if board[r][c] == " "]

def minimax(board, depth, is_maximizing):
    if check_winner(board, "O"):
        return 10 - depth
    if check_winner(board, "X"):
        return depth - 10
    if not get_available_moves(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for (row, col) in get_available_moves(board):
```

```
            board[row][col] = "O"
            score = minimax(board, depth + 1, False)
            board[row][col] = " "
            best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for (row, col) in get_available_moves(board):
            board[row][col] = "X"
            score = minimax(board, depth + 1, True)
            board[row][col] = " "
            best_score = min(best_score, score)
        return best_score


def computer_move(board):
    best_score = -math.inf
    best_move = None
    for (row, col) in get_available_moves(board):
        board[row][col] = "O"
```

```
            score = minimax(board, 0, False)
            board[row][col] = " "
            if score > best_score:
                best_score = score
                best_move = (row, col)
    return best_move


def main():
    print("Yashraj Sinha (1BM22CS335)")
    print("Welcome to Tic Tac Toe!")
    board = [[" " for _ in range(3)] for _ in range(3)]

    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:
            # Player's turn
            while True:
                try:
```

```
            row = int(input("Enter the row (0, 1, 2): "))
            col = int(input("Enter the column (0, 1, 2): "))

            if (row, col) not in get_available_moves(board):
                print("This spot is already taken or invalid. Try again.")
            else:
                board[row][col] = "X"
                break
        except ValueError:
            print("Invalid input. Please enter numbers 0, 1, or 2.")
else:
    # Computer's turn
    row, col = computer_move(board)
    board[row][col] = "O"
    print(f"Computer chose: ({row}, {col})")

print_board(board)

# Check for a winner
```

```
        if check_winner(board, "X"):
            print("Congratulations! You win!")
            return
        elif check_winner(board, "O"):
            print("Computer wins! Better luck next time.")
            return

    print("It's a tie!")

if __name__ == "__main__":
    main()
```

Output

```
Yashraj Sinha (1BM22CS335)
Welcome to Tic Tac Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X |   |
---------
  |   |
---------
  |   |
---------
Computer chose: (1, 1)
X |   |
---------
  | O |
---------
  |   |
---------
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X |   |
---------
  | O |
---------
  | X |
---------
Computer chose: (1, 0)
X |   |
---------
O | O |
---------
  | X |
---------
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 2
X |   |
---------
O | O | X
---------
  | X |
---------
Computer chose: (0, 2)
X |   | O
---------
O | O | X
---------
  | X |
---------
Enter the row (0, 1, 2): 2
```

```
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 0
X |   | O
---------
O | O | X
---------
X | X |
---------
Computer chose: (2, 2)
X |   | O
---------
O | O | X
---------
X | X | O
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X | X | O
---------
O | O | X
---------
X | X | O
---------
It's a tie!
```

Program 2 - 8 Puzzle BFS and DFS

Algorithm

3) Implementing 8-puzzle problem using python

BFS

Let fringe be a list containing the initial state.

LOOP

if fringe is empty return failure
Node ← remove-first (fringe)
if Node is goal
then return the path from initial state to node and add generated nodes to the head of fringe
~~END LOOP~~ else generate all successors of Node and
~~add~~ add generated node to the back of fringe.

END LOOP.

DFS

Let fringe be a list containing the initial state.

LOOP

if fringe is empty return failure
Node ← remove-first (fringe)
if Node is goal
then return the path from initial state to Node
else generate all successors of Node and add generated node to the front fringe.

END LOOP.

State Space tree : {using BFS}

Initial ⟷ **Final**

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

Initial state

1	2	3
4	5	6
0	7	8

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

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

final state.

18/10/2024

Code (BFS)

```
from collections import deque


def is_solvable(state):
    inversions = 0
    flattened = [num for row in state for num in row if num != 0]
    for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1
    return inversions % 2 == 0


def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()
```

```
def get_neighbors(state):

    rows, cols = len(state), len(state[0])

    for r in range(rows):

        for c in range(cols):

            if state[r][c] == 0:

                zero_pos = (r, c)

                break

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

    neighbors = []

    for dr, dc in directions:

        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc

        if 0 <= nr < rows and 0 <= nc < cols:

            new_state = [row[:] for row in state]

            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]

            neighbors.append(new_state)

    return neighbors



def bfs(initial, goal):

    queue = deque([(initial, [])])
```

```
    visited = set()
    visited.add(tuple(tuple(row) for row in initial))

    while queue:
        current, path = queue.popleft()
        if current == goal:
            return path + [current]

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                queue.append((neighbor, path + [current]))
    return None

def main():
    print("Yashraj Sinha (1BM22CS335)")
    print("8-Puzzle Solver Using BFS")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
```

```
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]     # Goal state


print_state(initial_state, label="Initial State:")

print_state(goal_state, label="Goal State:")


if not is_solvable(initial_state):

    print("This puzzle is not solvable.")

    return


solution = bfs(initial_state, goal_state)

if solution:

    print("Solution found in {} steps:\n".format(len(solution) - 1))

    for i, step in enumerate(solution):

        if i == 0:

            print_state(step, label="Initial State:")

        elif i == len(solution) - 1:

            print_state(step, label="Final State:")

        else:

            print_state(step, label=f"Step {i}:")
```

```python
    else:
        print("No solution exists.")


if __name__ == "__main__":
    main()
```

Code (DFS)

```
def is_solvable(state):
    inversions = 0
    flattened = [num for row in state for num in row if num != 0]
    for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1
    return inversions % 2 == 0


def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()


def get_neighbors(state):
    rows, cols = len(state), len(state[0])
```

```
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors


def dfs(initial, goal):
    stack = [(initial, [])]
    visited = set()
    visited.add(tuple(tuple(row) for row in initial))
```

```
    while stack:
        current, path = stack.pop()
        if current == goal:
            return path + [current]

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                stack.append((neighbor, path + [current]))
    return None

def main()
    print("Yashraj Sinha (1BM22CS335)")
    print("8-Puzzle Solver Using DFS")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]      # Goal state
```

```
print_state(initial_state, label="Initial State:")

print_state(goal_state, label="Goal State:")



if not is_solvable(initial_state):

    print("This puzzle is not solvable.")

    return



solution = dfs(initial_state, goal_state)

if solution:

    print("Solution found in {} steps:\n".format(len(solution) - 1))

    for i, step in enumerate(solution):

        if i == 0:

            print_state(step, label="Initial State:")

        elif i == len(solution) - 1:

            print_state(step, label="Final State:")

        else:

            print_state(step, label=f"Step {i}:")

else:

    print("No solution exists.")
```

```
if __name__ == "__main__":
    main()
```

Output (BFS)

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using BFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Output (DFS)

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using DFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Program 3 - Iterative Deepening Search

Algorithm

* Iterative Deepening Search

Pseudocode.

```
function IDS(problem) returns a solution
    inputs : problem, a problem

    for depth ← 0 to ∞ do
        result ← Depth-Limited Search
                        (problem, depth)
        if result ≠ cutoff then return result
    end.
```

Code

```
print("Yashraj Sinha (1BM22CS335)")

def is_solvable(state):
    inversions = 0
    flattened = [num for row in state for num in row if num != 0]
    for i in range(len(flattened)):
        for j in range(i + 1, len(flattened)):
            if flattened[i] > flattened[j]:
                inversions += 1
    return inversions % 2 == 0

def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()

def get_neighbors(state):
    rows, cols = len(state), len(state[0])
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors

def ids(initial, goal, depth_limit):
    def dls(state, path, depth):
        if state == goal:
```

```
            return path + [state]
        if depth == 0:
            return None
        for neighbor in get_neighbors(state):
            if tuple(tuple(row) for row in neighbor) not in visited:
                visited.add(tuple(tuple(row) for row in neighbor))
                result = dls(neighbor, path + [state], depth - 1)
                if result:
                    return result
        return None

    for depth in range(depth_limit):
        visited = set()
        visited.add(tuple(tuple(row) for row in initial))
        result = dls(initial, [], depth)
        if result:
            return result
    return None

def main():
    print("8-Puzzle Solver Using Iterative Deepening Search")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]      # Goal state

    print_state(initial_state, label="Initial State:")
    print_state(goal_state, label="Goal State:")

    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return

    depth_limit = 20
    solution = ids(initial_state, goal_state, depth_limit)
    if solution:
        print("Solution found in {} steps:\n".format(len(solution) - 1))
        for i, step in enumerate(solution):
            if i == 0:
                print_state(step, label="Initial State:")
            elif i == len(solution) - 1:
                print_state(step, label="Final State:")
            else:
```

```python
            print_state(step, label=f"Step {i}:")
    else:
        print("No solution exists within depth limit {}.".format(depth_limit))

if __name__ == "__main__":
    main()
```

Output

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using Iterative Deepening Search
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Program 4 - Vacuum Cleaner Agent

Algorithm

2) Implementing vacuum cleaner using python

```
FUNCTION vacuum_world()
    INITIALISE goal_stat as ['A' : '0', 'B' : '0']
        // 0: Clean, 1: Dirty
    SET cost to 0
    PROMPT for vacuum location (A|B) as location_input
    PROMPT for status of location_input as status_input
    SET other_location to 'B' if 'A' else 'A'
    PROMPT for status of other_location as status_input_
    compliment

    IF location_input is 'A':
        IF status_input is '1':
            SET goal_stat ['A'] to '0'
            Increment cost by 2
            IF status_input_compliment is '1':
                SET goal_stat ['B'] to '0'
        ELSE IF status_input_compliment is '1'
            Increment cost by 1
            SET goal_stat ['B'] to '0'
    ELSE:
        IF status_input is '1':
            SET goal_stat ['B'] to '0'
            Increment cost by 2
            IF status_input_compliment is '1':
                SET goal_stat ['A'] to '0'
        ELSE IF status_input_compliment is '1':
            Increment cost by 1
            SET goal_stat ['A'] to '0'.
    Print goal_state and cost
END FUNCTION
CALL vacuum_world().
```

Algo

1) Initialize the agent's starting (x,y)

2) Loop until all cells are clean:
    a. Perceive the current cell
    b. If the ~~cell~~ cell is dirty
        i. Clean the current cell
    c. else
        i. Check surrounding ~~cb~~ cells (up, down left, right), to see if any are dirty.
        ii. Move to the next dirty cells applying strategy such as BFS, DFS or other algorithms.
    d. If no dirty cell are perceived, stop.

3) End

18/10/2024

Code

```
print("Yashraj Sinha (1BM22CS335)")


def vacuum_cleaner(initial_state):
    # Initial states of rooms A and B
    room_A, room_B = initial_state


    # Trace of actions
    actions = []


    # Start in Room A
    actions.append("Starting in Room A.")


    # Check room A
    if room_A == 1:
        actions.append("Room A is dirty. Cleaning Room A.")
        room_A = 0
    else:
        actions.append("Room A is already clean.")
```

```
# Move to Room B
actions.append("Moving to Room B.")

# Check room B
if room_B == 1:
    actions.append("Room B is dirty. Cleaning Room B.")
    room_B = 0
else:
    actions.append("Room B is already clean.")

# Move back to Room A
actions.append("Returning to Room A.")

# Final state
final_state = (room_A, room_B)
actions.append("Both rooms are now clean.")

return final_state, actions
```

```
def main():
    print("Vacuum Cleaner AI")

    # Input initial states of Room A and Room B
    room_A = int(input("Enter the state of Room A (0 for clean, 1 for dirty): "))
    room_B = int(input("Enter the state of Room B (0 for clean, 1 for dirty): "))

    # Validate input
    if room_A not in (0, 1) or room_B not in (0, 1):
        print("Invalid input. Please enter 0 or 1.")
        return

    # Solve using vacuum cleaner AI
    final_state, actions = vacuum_cleaner((room_A, room_B))

    # Output actions and final state
    print("\nActions:")
    for action in actions:
```

```
        print(action)

    print("\nFinal State:")
    print(f"Room A: {'Clean' if final_state[0] == 0 else 'Dirty'}")
    print(f"Room B: {'Clean' if final_state[1] == 0 else 'Dirty'}")

if __name__ == "__main__":
    main()
```

Output

```
Yashraj Sinha (1BM22CS335)
Vacuum Cleaner AI
Enter the state of Room A (0 for clean, 1 for dirty): 1
Enter the state of Room B (0 for clean, 1 for dirty): 1

Actions:
Starting in Room A.
Room A is dirty. Cleaning Room A.
Moving to Room B.
Room B is dirty. Cleaning Room B.
Returning to Room A.
Both rooms are now clean.

Final State:
Room A: Clean
Room B: Clean
```

<u>Program 5 - </u>A* Search Algorithm and Hill Climbing Algorithm

Algorithm

4) Implementing A* algorithm using python.

→ A*

function A* search (problem) return a solution or failure

node ← a node n with n.state = problem.initial_state, n.g = 0

frontier ← a priority queue ordered by ascending g+h, only element n.

loop do

if empty? (frontier) then return failure

n ← pop (frontier)

if problem.goalTest (n.state) then return solution (n)

for each action a in problem.actions (n.states) do

n' ← ChildNode (problem, n, a)

insert (n', g(n') + h(n'), frontier)

(i) Using no. of misplaced tiles as heuristic function

$f(n) = g(n) + h(n)$

1	2	3
4	5	6
0	7	8

~~Initial State~~

1	2	3
4	5	6
7	8	0

Goal state

Output

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

Initial state

```
1 2 3        1 2 3        1 2 3
4 5 0   ->   4 5 8   ->   4 5 8
6 7 8        6 7 0        6 0 7
                            ↓
1 2 3        1 2 3        1 2 3
5 0 8   <-   0 5 8   <-   4 5 8
4 6 7        4 6 7        0 6 7
  ↓
1 2 3        1 2 3        1 2 3
5 6 8   ->   5 6 8   ->   5 6 0
4 0 7        4 7 0        4 7 8
                            ↓
1 2 3        1 2 3        1 2 3
4 5 6   <-   0 5 6   <-   5 0 6
0 7 8        4 7 8        4 7 8
  ↓
1 2 3        1 2 3
4 5 6   ->   4 5 6
7 0 8        7 8 0
```

25/10

ii) Using Manhattan distance

→ Initial state				Goal state
1 2 3				1 2 3

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

$$\left.\begin{array}{ll} \text{Tile 1 :} & \text{Manhattan distance (MD)} = 0 \\ \text{Tile 3 :} & MD = 0 \\ \text{Tile 4 :} & MD = 0 \\ \text{Tile 2 :} & MD = 1 \\ \text{Tile 7 :} & MD = 1 \\ \text{Tile 8 :} & MD = 0 \\ \text{Tile 6 :} & MD = 1 \\ f(n) = \text{Total} & MD = 3 \end{array}\right\} \text{1st iteration}$$

$$\left.\begin{array}{ll} \text{Tile 1: } \cancel{\text{Total}} & MD = \cancel{2}\,0 \\ \text{Tile 2: Total} & MD = \cancel{3}\,0 \\ \text{Tile 3: } \cancel{\text{Total}} & MD = 0 \\ \text{Tile 4: } \cancel{\text{Total}} & MD = \cancel{3}\,0 \\ \text{Tile 5: Total} & MD = \cancel{3}\,2 \\ \text{Tile 6: Total} & MD = \cancel{2}\,1 \\ \text{Tile 7: Total} & MD = \cancel{1}\,0 \\ \text{Tile 8: Total} & MD = \cancel{2}\,0 \\ f(n) = \text{Total} & MD = 3 \end{array}\right\} \text{2nd iteration}$$

8/10

$f(n)$ = Total MD = 0 → Solution in 3rd iteration

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 7 & 8 & 6 \end{array} \;\longrightarrow\; \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 0 \\ 7 & 8 & 6 \end{array} \;\longrightarrow\; \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{array}$$

5) Implementing Hill Climbing search algorithm to solve N-Queens problem.

→ function Hill_Climbing (problem) returns a state that is local maximum

current ← Make - Node (problem. Initial - State)
loop do
neighbour ← a highest value successor of current
if neighbour. value ≤ current. Value
then return current. State

Execute
current ← neighbour.

6) Show how the cost calculation of current state and neighbour nodes. And continue until you reach goal configuration of 4-Queens board.

Ans

			Q
	Q		
		Q	
Q			

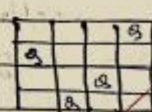
→

State	h(score)
3,1,2,0	2
1,3,2,0	1 ← this
2,1,3,0	1
0,1,2,3	6
3,2,1,0	6
3,0,2,1	1
3,1,0,2	1

State space :

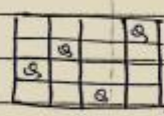
h=(2)

3,1,2,0



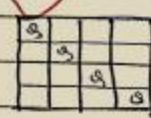
h=(1)

1,3,2,0



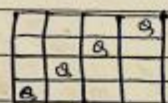
h=(1)

2,1,3,0



h = (6)

0,1,2,3

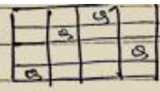


h=(6)

3,2,1,0

h = (1)

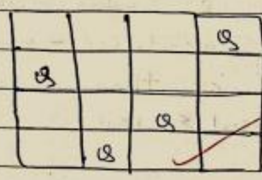
3,0,2,1

 h = (1)

3, 1, 0, 2

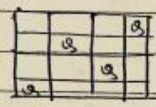
- Next step

1, 3, 2, 0

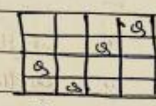
 h = (1)

<u>state</u>	<u>h (score)</u>
1, 3, 2, 0	1
3, 1, 2, 0	2
2, 3, 1, 0	2
0, 3, 2, 1	3
1, 2, 3, 0	3
1, 3, 0, 2	0 ← return solution.

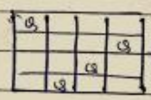
state space :

 h = (2)

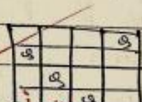
3, 1, 2, 0

 h = (2)

2, 3, 1, 0

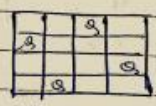
 h = (3)

0, 3, 2, 1

 h = (3)

15/11/24

1, 2, 3, 0

 h = (0)

1, 3, 0, 2

Code (A* algorithm using N – displaced Tiles)

```
import heapq

# Goal state

goal_state = (

    (1, 2, 3),

    (4, 5, 6),

    (7, 8, 0)

)



# Function to compute the heuristic (misplaced tiles)

def misplaced_tiles(state):

    misplaced = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != goal_state[i][j] and state[i][j] != 0:

                misplaced += 1

    return misplaced



# Function to get possible moves (neighbors)
```

```
def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    i, j = zero_pos

    # Possible moves: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = list(list(row) for row in state)  # Create a copy of the state
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            neighbors.append(tuple(tuple(row) for row in new_state))  # Convert back to tuple
    return neighbors

# Function to count the number of inversions in the puzzle
def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
    inversions = 0
```

```
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return None

    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tiles(initial_state), 0, initial_state, []))  # (f(n), g(n), state, path)
```

```
closed_list = set()

while open_list:
    f, g, current_state, path = heapq.heappop(open_list)
    closed_list.add(current_state)

    # If goal state is reached
    if current_state == goal_state:
        return path + [current_state]

    # Generate neighbors
    for neighbor in get_neighbors(current_state):
        if neighbor not in closed_list:
            heapq.heappush(open_list, (
                g + 1 + misplaced_tiles(neighbor),  # f(n) = g(n) + h(n)
                g + 1,  # Increment g(n) by 1 for each move
                neighbor,
                path + [current_state]
            ))
```

```
    return None  # No solution found


# Function to display the puzzle state
def display_state(state, label):
    print(f"{label} state:")
    for row in state:
        print(" ".join(str(x) for x in row))
    print()


# Example initial state (this one is solvable)
initial_state = (
    (1, 2, 3),
    (5, 6, 4),
    (7, 8, 0)
)


# Solving the puzzle
solution = a_star(initial_state)
```

```python
# Displaying the result
if solution:
    # Print Yashraj's information
    print("Yashraj Sinha (1BM22CS335)\n")

    # Print the initial state
    display_state(initial_state, "Initial")

    # Print the final state
    display_state(goal_state, "Goal")

    # Displaying the solution path
    print("Solution path:")
    for step in solution:
        display_state(step, "Step")
else:
    print("No solution found.")
```

Code (A* algorithm using Manhattan distance)

```
import heapq


# Goal state
goal_state = (
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 0)
)


# Function to compute the Manhattan distance heuristic
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_i, goal_j = divmod(tile - 1, 3)
                distance += abs(goal_i - i) + abs(goal_j - j)
```

```
    return distance


# Function to get possible moves (neighbors)
def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    i, j = zero_pos

    # Possible moves: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = list(list(row) for row in state)  # Create a copy of the state
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            neighbors.append(tuple(tuple(row) for row in new_state))  # Convert back to tuple
    return neighbors


# Function to count the number of inversions in the puzzle
```

```
def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
    inversions = 0
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return None
```

```
    open_list = []

    heapq.heappush(open_list, (0 + manhattan_distance(initial_state), 0, initial_state, []))  # (f(n),
g(n), state, path)

    closed_list = set()



    while open_list:

        f, g, current_state, path = heapq.heappop(open_list)

        closed_list.add(current_state)



        # Print the current state and its f(n) value

        print(f"State: {current_state}")

        print(f"f(n) = g(n) + h(n) = {g} + {manhattan_distance(current_state)} = {f}")

        print()



        # If goal state is reached

        if current_state == goal_state:

            return path + [current_state]



        # Generate neighbors
```

```
        for neighbor in get_neighbors(current_state):
            if neighbor not in closed_list:
                heapq.heappush(open_list, (
                    g + 1 + manhattan_distance(neighbor),  # f(n) = g(n) + h(n)
                    g + 1,  # Increment g(n) by 1 for each move
                    neighbor,
                    path + [current_state]
                ))

    return None  # No solution found

# Function to display the puzzle state
def display_state(state, label):
    print(f"{label} state:")
    for row in state:
        print(" ".join(str(x) for x in row))
    print()

# Example initial state (this one is solvable)
```

```
initial_state = (
    (1, 2, 3),
    (5, 6, 4),
    (7, 8, 0)
)


# Solving the puzzle
solution = a_star(initial_state)


# Displaying the result
if solution:
    # Print Yashraj's information
    print("Yashraj Sinha (1BM22CS335)\n")


    # Print the initial state
    display_state(initial_state, "Initial")


    # Print the final state
    display_state(goal_state, "Goal")
```

```
    # Displaying the solution path
    print("Solution path:")
    for step in solution:
        display_state(step, "Step")
else:
    print("No solution found.")
```

Code (Hill Climbing algorithm)

```
import random
print("Yashraj Sinha (1BM22CS335)")
# Function to calculate the number of attacking pairs of queens
def calculate_attacks(board):
    attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
```

```
                attacks += 1
    return attacks


# Function to generate a random initial state
def generate_initial_state(n):
    return [random.randint(0, n - 1) for _ in range(n)]


# Function to generate neighbors by moving one queen to a different row
def generate_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Make sure we are not moving the queen to its current row
                neighbor = board[:]
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors
```

```
# Hill Climbing algorithm with random restarts
def hill_climbing(n, max_restarts=100):
    for restart in range(max_restarts):
        current_state = generate_initial_state(n)
        current_attacks = calculate_attacks(current_state)

        while True:
            # Generate all neighbors
            neighbors = generate_neighbors(current_state)

            # Find the neighbor with the minimum number of attacks
            next_state = None
            next_attacks = current_attacks

            for neighbor in neighbors:
                attacks = calculate_attacks(neighbor)
                if attacks < next_attacks:
                    next_state = neighbor
                    next_attacks = attacks
```

```
            # If no improvement, return the solution or terminate
            if next_attacks == current_attacks:
                break

            current_state = next_state
            current_attacks = next_attacks

        # If a solution is found, return the current state
        if current_attacks == 0:
            return current_state

    # If no solution found after max_restarts, return None
    return None

# Function to display the board
def display_board(board):
    n = len(board)
    for i in range(n):
```

```python
        row = ['Q' if i == board[col] else '.' for col in range(n)]
        print(' '.join(row))
    print()


# Set the size of the board (N)
N = 8


# Solve the N-Queens problem with random restarts
solution = hill_climbing(N)


# Display the result
if solution:
    print(f"Solution for {N}-Queens:")
    display_board(solution)
else:
    print(f"No solution found for {N}-Queens.")
```

```
Yashraj Sinha (1BM22CS335)

Initial state:
1 2 3
5 6 4
7 8 0

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

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

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

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

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

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

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

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

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

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

<u>Output (Manhattan Distance)</u>

```
Yashraj Sinha (1BM22CS335)

Initial state:
1 2 3
5 6 4
7 8 0

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

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

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

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

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

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

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

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

Output (Hill Climbing)

```
Yashraj Sinha (1BM22CS335)
Solution for 8-Queens:
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
```

Program 6 - Simulated Annealing

Algorithm

⑥ Implement Simulated Annealing to solve N-Queen problem.

→ current ← initial state
T ← a large positive value
while T > 0 do
next ← a random neighbour of current
ΔE ← current. cost − next. cost
if ΔE > 0 then
current ← next
else
current ← next with probability $p = e^{\frac{\Delta E}{T}}$
end if
decrease T
end while
return current.

<u>Output</u>

Iteration 1 : Conflicts = 24
Iteration 2 : Conflicts = 26
⋮
Iteration 4999 : Conflicts = 0
Iteration 5000 : Conflicts = 0

Final solution : [3, 6, 2, 7, 1, 4, 0, 5]
Number of conflict in final solution : 0

15/11/24

Code

import random

import math

print("Yashraj Sinha (1BM22CS335)")

Objective function: count the number of attacking pairs of queens

```python
def calculate_attacks(board):
    attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if two queens are in the same row, column, or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                attacks += 1
    return attacks


# Function to generate a random initial state (random queen positions in each column)
def generate_initial_state(n):
    return [random.randint(0, n - 1) for _ in range(n)]


# Function to generate a neighboring solution by moving one queen in a column
def generate_neighbor(board):
    neighbor = board[:]
    column = random.randint(0, len(board) - 1)
    # Randomly select a new row for the queen in the chosen column
```

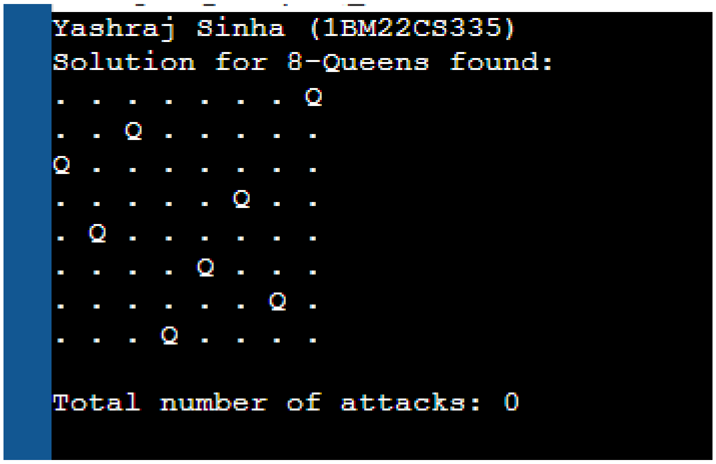
```python
    neighbor[column] = random.randint(0, len(board) - 1)
    return neighbor


# Simulated Annealing algorithm to solve the N-Queens problem
def simulated_annealing(n, max_iterations, initial_temperature, cooling_rate):
    current_state = generate_initial_state(n)
    current_attacks = calculate_attacks(current_state)
    temperature = initial_temperature

    best_state = current_state
    best_attacks = current_attacks

    for iteration in range(max_iterations):
        # Generate a neighbor solution
        neighbor = generate_neighbor(current_state)
        neighbor_attacks = calculate_attacks(neighbor)

        # Calculate the energy difference (how much worse the new state is)
        delta_attacks = neighbor_attacks - current_attacks
```

```
        # Accept the neighbor if it has fewer attacks or with a probability if it's worse
        if delta_attacks < 0 or random.random() < math.exp(-delta_attacks / temperature):
            current_state = neighbor
            current_attacks = neighbor_attacks

        # Update the best solution if necessary
        if current_attacks < best_attacks:
            best_state = current_state
            best_attacks = current_attacks

        # Cool down the temperature
        temperature *= cooling_rate

        # If no attacks, we found the solution
        if best_attacks == 0:
            break

    return best_state, best_attacks
```

```
# Function to display the board (where 'Q' is a queen and '.' is an empty space)
def display_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if i == board[col] else '.' for col in range(n)]
        print(' '.join(row))
    print()

# Parameters for Simulated Annealing
N = 8  # Set the size of the board (N x N)
max_iterations = 10000  # Higher number of iterations for better convergence
initial_temperature = 1000  # High initial temperature
cooling_rate = 0.995  # Cooling rate (temperature decreases by 0.5% every iteration)

# Solve the N-Queens problem using Simulated Annealing
solution, attacks = simulated_annealing(N, max_iterations, initial_temperature, cooling_rate)

# Output the result
```

```
print(f"Solution for {N}-Queens found:")

display_board(solution)

print(f"Total number of attacks: {attacks}")
```

Output



```
Yashraj Sinha (1BM22CS335)
Solution for 8-Queens found:
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .

Total number of attacks: 0
```

<u>Program 7 -</u> Knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

<u>Algorithm</u>

* Create a KB using prepositional logic and show that the given query entails the KB or not.

```
Initialize KB with prepositional logic statements
Input query

If forward_chaining(Knowledge_Base, query):
    print "Query is entailed by the KB".
else
    print " Not entails".

Function forward_chaining (KB, query)
    Initialize agenda with known facts from KB
    while agenda is not empty:
        pop: fact from agenda
        If fact matches query:
            Return True

        For each rule in KB
            If fact satisfies a rule's premise:
                Add the rules conclusion to
                agenda

    Return False

//Input
KB: ["A", "B", "A∧B⇒C", "C⇒D"]
query = "D"

Query is entailed by the KB
```

Code

```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
57from sympy import simplify_logic
def is_entailment(kb, query):
# Negate the query
negated_query = Not(query)
# Add negated query to the knowledge base
kb_with_negated_query = And(*kb, negated_query)
# Simplify the combined KB to CNF
simplified_kb = simplify_logic(kb_with_negated_query, form="cnf")
# If the simplified KB evaluates to False, the query is entailed
return simplified_kb == False
# Define a larger Knowledge Base
kb = [
Or(A, B),
# A ∨ B
Or(Not(A), C), # ¬A ∨ C
Or(Not(B), D), # ¬B ∨ D
Or(Not(D), E), # ¬D ∨ E
Or(Not(E), F), # ¬E ∨ F
F
# F
]
# Query to check
query = Or(C, F) # C ∨ F
# Check entailment
result = is_entailment(kb, query)
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

OUTPUT:

Is the query 'C | F' entailed by the knowledge base? Yes

Program 8 - **Knowledge base using prepositional logic and prove the given query using resolution.**

Algorithm

9) Create a knowledge base using Propositional logic and prove the given query using resolution.

Function: resolution (KB, query): return query if true or false

input: KB, the Knowledge base, set of propositions

query: a ~~s~~ state to be proven.

clauses = convert to CNF (KB, negated query = negate (query))

new-clause = set()

apply the resolution rules:

- select the clauses contain complimentary clauses
- resolve the two to form a new clause
- add the new-clause is empty{} contradiction is found

If new-clause == {} print "query if true"

else print "~~Ho~~ false"

Output:

KB: $P \vee Q$

To prove: P is true

$\neg Q \vee R$

$\neg P \vee Q$

~~R~~ $R \vee$ ~~QP~~

~~Resolution~~ $\neg R$

$(\neg Q \vee R), (\neg R) \rightarrow \neg Q$

$(\neg P \vee Q), (\neg Q) \rightarrow \neg P$

$(\cancel{\neg} R \vee P), (\neg P) \rightarrow \cancel{\neg} R$

Contradiction / ~~is~~ query is true.

Code

```
def negation(p):

    """Negate a literal."""

    if p.startswith("~"):

        return p[1:]  # remove the '~' from negated literals

    return f"~{p}"



def resolution(kb, query):

    """Perform resolution on the knowledge base to prove the query."""



    # Add the negation of the query to the knowledge base (for proof by contradiction)

    kb.append(negation(query))



    # Apply the resolution rule until we reach an empty clause (which means contradiction)

    new_clauses = set(kb)  # Keep track of all unique clauses in the knowledge base

    print(f"Initial Knowledge Base + negation of query: {kb}")



    while True:

        added_new_clause = False
```

```
# Try to resolve every pair of clauses
clauses = list(new_clauses)
for i in range(len(clauses)):
    for j in range(i + 1, len(clauses)):
        clause1 = clauses[i]
        clause2 = clauses[j]

        # Try to resolve these two clauses
        resolvent = resolve(clause1, clause2)

        if resolvent is not None:
            print(f"Resolving clauses: {clause1} and {clause2}")
            print(f"Resolved to: {resolvent}")

            # If resolvent is empty, we found a contradiction
            if not resolvent:
                return True  # Found a contradiction, so the query is provable

            # Add the new clause if it's not already in the set
```

```
                if resolvent not in new_clauses:
                    new_clauses.add(resolvent)
                    added_new_clause = True

        # If no new clause was added, resolution has terminated without a contradiction
        if not added_new_clause:
            break

    return False  # No contradiction found, so the query is not provable


def resolve(clause1, clause2):
    """Resolve two clauses if possible and return the resolvent."""
    # Split clauses into literals
    literals1 = set(clause1.split(" v "))
    literals2 = set(clause2.split(" v "))

    # Try to find complementary literals
    for literal in literals1:
```

```
        neg_literal = negation(literal)

        if neg_literal in literals2:

            # Resolve the two clauses by removing complementary literals

            new_clause = literals1.union(literals2) - {literal, neg_literal}

            return " v ".join(sorted(new_clause))  # Return the resolved clause as a string

    return None  # No resolvent found




# Example knowledge base and query (where T is provable)

kb = [

    "P v Q",          # P or Q

    "~P v R",         # Not P or R

    "Q v ~R",         # Q or Not R

    "R v T"           # R or T

]


query = "T"  # Query to prove (e.g., prove T)


# Perform resolution to prove the query
```

```
result = resolution(kb, query)


if result:
    print(f"\nQuery '{query}' is provable from the knowledge base.")
else:
    print(f"\nQuery '{query}' is not provable from the knowledge base.")
```

<u>Output</u>

```
Yashraj Sinha (1BM22CS335)
Initial Knowledge Base + negation of query: ['P v Q', '~P v R', 'Q v ~R', 'R v T', '~T']
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: ~P v R and Q v ~R
Resolved to: Q v ~P
Resolving clauses: R v T and ~T
Resolved to: R
Resolving clauses: R v T and Q v ~R
Resolved to: Q v T
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: ~P v R and Q v ~R
Resolved to: Q v ~P
Resolving clauses: R v T and ~T
Resolved to: R
Resolving clauses: R v T and Q v ~R
Resolved to: Q v T
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: ~P v R and Q v ~R
Resolved to: Q v ~P
Resolving clauses: R v T and ~T
Resolved to: R
Resolving clauses: R v T and Q v ~R
Resolved to: Q v T

Query 'T' is not provable from the knowledge base.
```

Program 9 – Unification in First Order Logic.

Algorithm

(7) Implement unification in First Order Logic

→ i> If Ψ_1 and Ψ_2 is a variable or constant then:

(a) If Ψ_1 and Ψ_2 are identical, then return NIL.

(b) Else if Ψ_1 is a variable,

- then if Ψ_1 occurs in Ψ_2 then return failure
- Else return $\{(\Psi_2/\Psi_1)\}$.

(c) Else if Ψ_2 is a variable,

- If Ψ_2 occur in Ψ_1 then return failure
- Else return $\{(\Psi_1/\Psi_2)\}$.

(d) Else return failure.

ii> If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return failure

iii> If Ψ_1 and Ψ_2 have a different number of ~~arg~~ arguments, then return failure.

iv> Set substitution set (SUBST) to NIL.

v> for i=1 to number of elements in Ψ_1

(a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2 and put the result into S.

(b) If S = failure then return Failure

(c) If S ≠ NIL then do.
- Apply S to the remainder of both L_1 and L_2.
- SUBST = APPEND(S, SUBST).

vi) Return SUBST.

& Execute

Output

Expression 1: ("Eats", "x", "Mongo", "Pizza")
Expression 2: ("Eats", "Sumit", "y", "z")

Substitutions: {'x': 'Sumit', 'y': 'Mongo', 'z': 'Pizza'}

Code

```
print("Yashraj Sinha (1BM22CS335)\n")


def occurs_check(var, term):
    """Check if a variable occurs in a term."""
    if var == term:
        return True
    elif isinstance(term, tuple):  # If the term is a function or a tuple
        return any(occurs_check(var, t) for t in term[1:])
    return False


def unify(term1, term2, substitution=None):
    """Attempt to unify two terms (or predicates)."""
    if substitution is None:
        substitution = {}

    # If both terms are the same, no unification needed
    if term1 == term2:
        return substitution
```

```
# If term1 is a variable, try to unify it with term2
if isinstance(term1, str) and term1.isupper():
    if term1 in substitution:
        return unify(substitution[term1], term2, substitution)
    if occurs_check(term1, term2):
        return None  # Avoid circular unification (occurs check)
    substitution[term1] = term2
    return substitution

# If term2 is a variable, try to unify it with term1
if isinstance(term2, str) and term2.isupper():
    return unify(term2, term1, substitution)

# If both terms are functions or predicates (tuples), unify their components
if isinstance(term1, tuple) and isinstance(term2, tuple):
    if len(term1) != len(term2):
        return None  # Different number of arguments
    for t1, t2 in zip(term1[1:], term2[1:]):
```

```python
            substitution = unify(t1, t2, substitution)
            if substitution is None:
                return None  # If any unification fails, return None
        return substitution

    return None  # If no other cases match, return None (failure)


# Example usage
term1 = ('P', 'X', 'a')  # Predicate P(X, a)
term2 = ('P', 'b', 'a')  # Predicate P(b, a)

# Attempt to unify
substitution = unify(term1, term2)
if substitution is not None:
    print("Unification succeeded with substitution:", substitution)
else:
    print("Unification failed")
```

Output

```
Yashraj Sinha (1BM22CS335)

Unification succeeded with substitution: {'X': 'b'}
```

Program 10 - Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm

10) Convert a given FOL to CNF

→ Function FOL-to-CNF:

Replace $(P \leftrightarrow Q)$ with $(P \rightarrow Q) \wedge (Q \rightarrow P)$

Replace $(P \rightarrow Q)$ with $(\neg P \vee Q)$

Replace $\neg(P \vee Q)$ with $\neg P \wedge \neg Q$

Replace $\neg(P \wedge Q)$ with $\neg P \vee \neg Q$

Replace $\neg\neg P$ with P

Rename all variable to make them unique.

Replace $\exists x[P(x)]$ with $P(c)$ if x is independent

Replace $\exists x[P(x)]$ with ~~P(y)~~ $P(f(y))$ if x depends on a universally quantified variable.

~~Remove~~ Distribute $\vee$ over $\wedge$:

Ensure all formulas is a conjunction of disjunction

Output

$\forall x (\exists y (P(x,y) \rightarrow Q(y)) \wedge \neg R(x)$

~~y = f(x)~~ $y = f(x)$

$\forall x ((\neg P(x, f(x)) \vee Q(f(x)))) \wedge \neg R(x)$

$\Rightarrow (\neg P(x, f(x)) \vee (f(x)) \wedge \neg R(x)$

Code

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf
def fol_to_cnf(fol_expr):
54"""
Converts a First-Order Logic (FOL) statement to Conjunctive Normal Form (CNF).
Arguments:
fol_expr: A sympy logical expression representing the FOL statement.
Returns:
The CNF equivalent of the input expression.
"""
# Step 1: Eliminate equivalences (A ↔ B) using (A → B) ∧ (B → A)
fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b), Implies(b, a)))
# Step 2: Eliminate implications (A → B) using (¬A ∨ B)
fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))
# Step 3: Convert to CNF
cnf_form = to_cnf(fol_expr, simplify=True)
return cnf_form
def main():
# Define propositional symbols instead of first-order predicates
P = symbols("P")
Q = symbols("Q")
R = symbols("R")
# Example 1: P → Q
fol_expr1 = Implies(P, Q)
print("Example 1: P → Q")
print("Original FOL Expression:")
print(fol_expr1)
# Convert to CNF
cnf1 = fol_to_cnf(fol_expr1)
print("\nCNF Form:")
print(cnf1)
55# Example 2: (P ∨ ¬Q) → (Q ∨ R)
fol_expr2 = Implies(Or(P, Not(Q)), Or(Q, R))
print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
print("Original FOL Expression:")
print(fol_expr2)
# Convert to CNF
cnf2 = fol_to_cnf(fol_expr2)
print("\nCNF Form:")
print(cnf2)
if __name__ == "__main__":
main()
```

OUTPUT:

Example 1: P → Q
Original FOL Expression:
Implies(P, Q)
CNF Form:
Q | ~P
Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)
CNF Form:
Q | R

Program 11 - Knowledge base consisting of first order logic statements and prove the given query using forward reasoning..

Algorithm

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

→ function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses

α, the query, an atomic sentence.

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty

new ← { }

for each rule ~~is empty~~ in KB do

($p_1 \wedge \ldots \wedge p_n \Rightarrow q$) ← STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta$, $p_1' \wedge \ldots \wedge p_n'$)

for some $p_1', \ldots, p_n'$ in KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

ϕ ← unify(q', α)

if ϕ is not fail then return

add new to KB

return false

Output

Criminal (Robert) is proven!
Inferred facts:
Missile (T_1)
Weapon (T_1)
Hostile (A)
Owns (A, T_1)
Criminal (Robert)
Enemy (A, America)
American (Robert)
Sells (Robert, T_1, A)

29/11/24

Q) Consider a vocabulary with the following symbols:

(a) Occupation (Emily, Surgeon) $\vee$ Occupation (Emily, La

(b) Occupation (Joe, Actor) $\wedge$ $\exists_o$ (Occupation (Joe, o) $\wedge$ o $\neq$ Actor)

(c) $\forall_p$ (Occupation (p, Surgeon) $\rightarrow$ Customer (Joe, p))

(d) $\exists_p$ (Occupation (p, Lawyer) $\wedge$ Customer (Joe, p))

(e) $\exists p$ (Boss(p, Emily) $\wedge$ Occupation (p. lawyer

(f) $\exists p$ (Occupation (p, Lawyer) $\wedge$ $\forall c$ (Customer (c, p)
$\rightarrow$ Occupation (c, Doctor)))

(g) $\forall p$ (Occupation (p, Surgeon) $\rightarrow$ $\exists l$ (Occupation
(l, lawyer) $\wedge$ Customer (p, l)))

29/11/24

Code

```
knowledge_base = {
"facts": {
"American(Robert)": True,
"Enemy(A, America)": True,
"Owns(A, T1)": True,
"Missile(T1)": True,
},
"rules": [
{"if": ["Missile(x)"], "then": ["Weapon(x)"]},
{"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
{"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},
{
"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
"then": ["Criminal(p)"],
},
],
}
def forward_chaining(kb):
facts = kb["facts"].copy()
rules = kb["rules"]
inferred = set()
while True:
43new_inferences = set()
for rule in rules:
if_conditions = rule["if"]
then_conditions = rule["then"]
substitutions = {}
all_conditions_met = True
for condition in if_conditions:
predicate, args = condition.split("(")
args = args[:-1].split(",")
matched = False
for fact in facts:
fact_predicate, fact_args = fact.split("(")
fact_args = fact_args[:-1].split(",")
if predicate == fact_predicate and len(args) == len(fact_args):
temp_subs = {}
for var, val in zip(args, fact_args):
if var.islower():
if var in temp_subs and temp_subs[var] != val:
break
```

```
temp_subs[var] = val
elif var != val:
break
else:
matched = True
substitutions.update(temp_subs)
break
if not matched:
all_conditions_met = False
break
44if all_conditions_met:
for condition in then_conditions:
predicate, args = condition.split("(")
args = args[:-1].split(",")
new_fact = predicate + "(" + ",".join(substitutions.get(arg, arg) for arg in args)
+ ")"
new_inferences.add(new_fact)
if new_inferences - inferred:
inferred.update(new_inferences)
facts.update({fact: True for fact in new_inferences})
else:
break
return inferred
result = forward_chaining(knowledge_base)
print('Yashraj Sinha (1BM22CS335):')
if "Criminal(Robert)" in result:
print("Proved: Robert is a criminal.")
else:
print("Could not prove that Robert is a criminal.")
```

OUTPUT:

Yashraj Sinha (1BM22CS335):
Proved: Robert is a criminal.

Program 12 - Implement Alpha-Beta Pruning.

Algorithm

Q11> Implement alpha-beta pruning

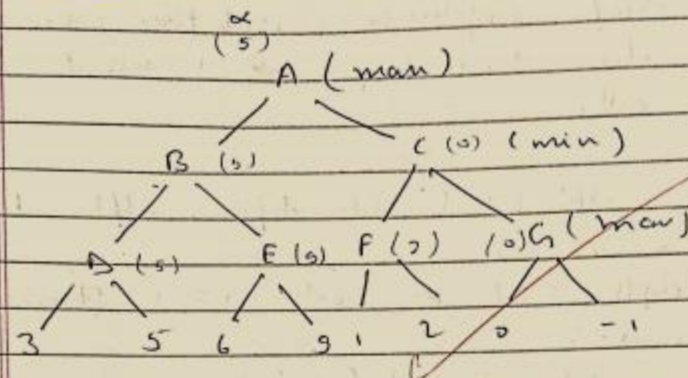
-> alpha : the ~~p~~ best option for the maximizer
Beta : the best option for the minimizer
purning: Stop exploring sub-tree when it is clear that they won't affect the outcome.

```
Function alphabeta ( node, depth, alpha, beta,
        is_maximizing
    if (depth == 0) or node is a terminal
        node:
            return evaluate (node)
    if is_maximizing ():
            maxeval = -∞
        for each child in children (node):
            eval = alphabeta (child, depth-1, alpha,
                    beta, false)
        maxeval = max (maxeval, eval)
        alpha = max (alpha, eval)
            if alpha >= beta:
                break (prune)
        return maxeval.

    else:
        minval = +∞
        for each child in children (node):
        eval = alphabeta ( child, depth-1, alpha,
                    beta, eval )
        beta = min (beta, eval)
        if alpha >= beta :
```

break (prune)

return min val

Output



Code

```
import math
def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta,
max_depth):
# Base case: when the maximum depth is reached
if depth == max_depth:
return values[node_index]
if is_maximizing_player:
best = -math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
best = max(best, val)
alpha = max(alpha, best)
# Prune the remaining nodes
if beta <= alpha:
break
return best
else:
best = math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
best = min(best, val)
beta = min(beta, best)
49# Prune the remaining nodes
if beta <= alpha:
break
return best
print("Yashraj Sinha (1BM22CS335):")
# Example usage
if __name__ == "__main__":
# Example tree represented as a list of leaf node values
values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3 # Height of the tree
result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
print("The optimal value is:", result)
```

OUTPUT:

```
Yashraj Sinha (1BM22CS335):
The optimal value is: 5
```